CONTENTS

Units	Page No.
1. Introduction	1
2. Data Structures	31
3. Advanced Design and Analysis Techniques	78
4. Graph Algorithms	115
5. Selected Topic	160

SYLLABUS

C-129 DESIGN AND ANALYSIS OF ALGORITHM

UNIT I
INTRODUCTION:
Algorithms, Analysis of Algorithms, Design of Algorithms, and Complexity of Algorithms, Asymptotic Notations, Growth of function, Recurrences Sorting in polynomial Time: Insertion sort, Merge sort, Heap sort, and Quick sort sorting in Linear Time: Counting sort, Radix Sort, Bucket Sort Medians and order statistics.

UNIT II
Elementary Data Structure: Stacks, Queues, Linked list, Binary Search Tree, Hash Table
Advanced Data Structure: Red Black Trees, Splay Trees, Augmenting Data Structure Binomial Heap, BTree, Fibonacci Heap, and Data Structure for Disjoint Sets Union-find Algorithm, Dictionaries and priority Queues, mergeable heaps, concatenable queues.

UNIT III
Advanced Design and Analysis Techniques: Dynamic programming, Greedy Algorithm, Backtracking, Branch-and-Bound, Amortized Analysis.

UNIT IV
Graph Algorithms: Elementary Graph Algorithms, Breadth First Search, Depth First Search, Minimum Spanning Tree, Kruskal's Algorithms, Prim's Algorithms, Single Source Shortest Path, All pair Shortest Path, Maximum flow and Travelling Salesman Problem.

UNIT V
Randomized Algorithms, String Matching, NP-Hard and NP-Completeness Approximation Algorithms, Sorting Network, Matrix Operations, Polynomials and the FFT, Number Theoretic Algorithms, Computational GeometrOR.

UNIT

1

INTRODUCTION

NOTES

STRUCTURE

1.0 Objectives
1.1 Algorithms
1.2 Growth Function-Asymptotic Function
1.3 Grown Function-Asymptotic Notation (O, Ω, θ)
1.4 Designing of Algorithm
1.5 Master Theorem
1.6 Heap Sort (Sorting and Order Statistics)
1.7 Sorting is Linear Time
1.8 Median and Order Statistics Sorting
- *Summary*
- *Glossary*
- *Review Questions*
- *Further Readings*

1.0 OBJECTIVES

After going through this unit, you will be able to:

- discuss about analysis and design of algorithms.
- illustrate the applications of growth and grown function (asymptotic function and notation) and recurrences sorting in polynomial time.
- describe merge sort, heap sort and quick sort.
- define sorting in linear time.

1.1 ALGORITHMS

An algorithm is a finite set of instruction that if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following:

1. *Input.* Zero or more quantities are externally suplied.
2. *Output.* At least one quantity is produced.
3. *Definiteness.* Each instruction is clear and unambigous.
4. *Finiteness.* After finite numbers of step algorithm must terminate.
5. *Effectiveness.* Every instruction must be basic and must be feasible.

NOTES

Example:

```
Tower of Hanoi (n, x, y, z)
"move n disk from tower x to y".
{
If (n ≥ 1)
Tower of Hanoi (n – 1, x, z, y);
Write ("move top disk from tower", x, " to top of tower," y),
Tower of Hanoi (n – 1) z, y, x);
}
}
```

Space Complexity. Space complexity of an algorithms is the amount of memory it needed to run to completion of program.

Time Complexity. The time T(*p*) taken by a program *p* is the sum of the compile time and the run or (execution) time. The compile time does not depends on the instance characteristics. If run time denoted by *t*(*p*) then

$$t_p(n) = C_a \text{ ADD } (n) + C_s \text{ SUB } (n) + C_m \text{ MULT } (n) + C_d \text{ DIV } (n) + \ldots\ldots$$

when n denotes instance characteristics, and C_a, C_s, C_m, C_d ... and so on denoted the time needed addition subtraction and *b* on.

The time complexity of an algorithm is the amount of computer time it needed to run to completion.

Example:

```
a + b + b * c + (a + b – c)/(a + b) + 4
Also
abc (a, b, c)
{
return a + b + b * c + (a + b – c)/(a + b) + 4
}
Space needed by following component is seen by following component.
ISum (a, n)
{                                        // Iterative method
S: = 0.0;
for i: = 1 to n do
S: – S + a(i);
return S;
}
RSum (a, n)
{
If (n ≤ 0) then return 0;
else return RSum (a; n – 1) + a[n];
}
```

Analysis of Algorithms. Analysis of Algorithm depends of various factor's such as

– memory

– communication bandwidth

– computer hardware

But most often used is the computation time that an algorithm's require for completing the given task. Algorithm is machine and language independent. These are the only important durable and original part of (machine) C.S. (Critical Section).

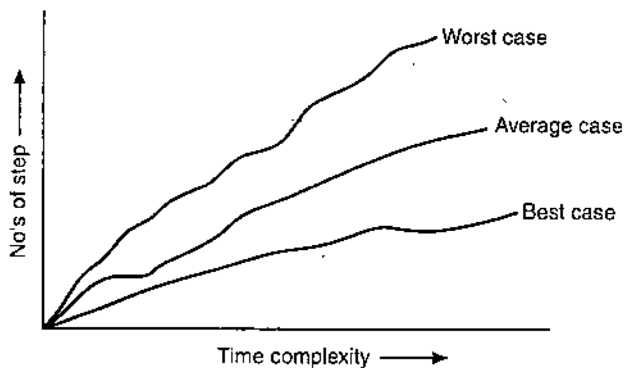
If we are using any arithmetic operation then it uses one time computation. but if we are using loop which are not in single step loops then counting of run time exceed. One that condition three cases arises:

1. Worst Case
2. Average Case
3. Best Case.

1. *Worst Case.* Worst case time complexity is the function defined by maximum amount of time needed by an algorithm for an input of size 'n' thus it is the function defined by the maximum number of steps taken on any instance of size 'n'.

2. *Average Case.* The average case time complexity is the execution of an algorithm having typical input data of size 'n'. Thus it is the function defined by the average number of steps taken on any instance of size n.

3. *Best Case.* The best case time complexity is the minimum amount of time that an algorithm requires for an input of size 'n'. Thus it is the function defined by minimum number of step taken on any instance of size n.



Example of Insertion Sort

Number's are

(80), 40, 50, 20, 90, 30, 70, 60

40, 80, (50), 20, 90, 30, 70, 60

40, 50, 80, (20), 90, 30, 70, 60

20, 40, 50, 80, 90, (30), 70, 60

20, 30, 40, 50, 80, 90, (70), 60

20, 30, 40, 50, 70, 80, 90, (60)

20, 30, 40, 50, 60, 70, 80, 90

NOTES

Algorithm of Insertion Sort

Element of Array is 'A' in an ascending order. Array consist of n-element i is index value and t is temporary storage.

Step 1. Look until length [A]

Repeat step 1, 2 for $i \leftarrow 2, 3, 4 \ldots n$

Set $i \leftarrow A[i]$

$P \leftarrow i - 1$

temporary variable is set to new value, pointer is adjusted.

Step 2. Loop, comparision

Repeat while ($P > 0$ and $t < A[P]$)

End of loop (step 2)

Step 3. Inserting element in appropriate place

Set $A[P + 1] \leftarrow t$

End of step 1 loop

Step 4. Finished

Return.

1.2 GROWTH FUNCTION-ASYMPTOTIC FUNCTION

Theta "θ" Notation

The function $f(n) = \theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constant C_1, C_2 and n_0 such that

$$C_1 g(n) \le f(n) \le C_2 g(n) \text{ for all } n, n \ge n_0$$

Example 1. *The function $f(n) = 3n + 2 = \theta(n)$.*

Sol. $3n \le 3n + 2 \le 4n$ for all $n \ge 2$

where $C_1 = 3$, $C_2 = 4$, $n_0 = 2$

Example 2. *Function $f(n) = 5n + 7 = \theta(n)$.*

Sol. $5n \le 5n + 7 \le 6n$ for all $n \ge 7$

where $C_1 = 5$, $C_2 = 6$, $n_0 = 7$

on given condition of theta

$$C_1 g(n) \le f(n) \le C_2 g(n) \quad \text{for all } n \ge n_0$$

Note. In theta notation lower and upper bound both exist where lower bound = $C_1 g(n)$ and upper bound = $C_2 g(n)$

Omega "Ω" Notation

The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constant C and n_0 such that $f(n) \ge C * g(n)$ for all n, $n \ge n_0$

Ω notation provides an asymptotic lower bound.

Example 1. *Function $f(n) = 3n + 2$.*

Sol. $3n + 2 \ge 3n$ for $n \ge 1$

where $C = 3$ and $n_0 = 1$

and $\Omega(n) = 3n + 2$

NOTES

Example 2. *Function* $f(n) = 3n^2 + n + 1$.

Sol. $3n^2 + n + 1 \geq 3n^2$ for $n \geq 1$

$$C = 3 \text{ and } n_0 = 3$$

and $\Omega(n^2) = 3n^2 + n + 1$

i.e., complexity of above function be n^2.

1.3 GROWN FUNCTION-ASYMPTOTIC NOTATION (O, Ω, θ)

Big 'oh'

The function $f(n) = O(g(n))$

(read as "f of n is big oh of g of n") Iff there exist positive constants C and n_0 such that

$$f(n) \leq C * g(n) \text{ for all } n, n \geq n_0$$

On big oh notation we find out only asymptotic upper bound.

Example 1. *The function* $f(n) = 10n^2 + 4n + 2$

Sol. $10n^2 + 4n + 2 \leq 11n^2$

for all $n \geq 5$

$$O(n^2) = 10n^2 + 4n + 2$$

$O(n)$ is called linear and $O(n^2)$ is called quadratic.

$O(n^3)$ is called qubic etc.

Example 2. The function $f(n) = 3n + 2$ $O(n) = 2$

Sol. $3n + 2 \leq 4n$ for all $n \geq 2$ $C = 4, n_0 = 2$

$$\boxed{O(f(n) = n}$$

Theorem. *If* $f(n) = a_m n^m + \ldots a_1 n + a_0$ *then show*

$$f(n) = O(n^m)$$

Proof.
$$f(n) \leq \sum_{i=0}^{m} (a_i)\, n^i$$

$$\leq n^m \sum_{i=0}^{m} (a_i)\, n^{i-m}$$

$$\leq n^m \sum_{i=0}^{m} (a_i) \text{ for } n \leq 1$$

for fixed value of m

So $f(n) = O(n^m)$

Omega (Ω)

Function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") If $f(n)$ positive constants C and n_0 so that

$$f(n) \geq C\, g(n) \text{ for all } n, n \geq n_0$$

NOTES

Example:

$$f(n) = 3n + 2 \geq 3n \quad \text{for } n \geq 1$$

Little "oh"

The function $f(n) = O(g(n))$ (read as "f of n is little oh of g of n") iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Example 1. *Function $f(n) = 3n + 2$*

Sol.

$$g(n) = n^2$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{3n+2}{n^2}$$

Example 2.

$$f(n) = 7n + 5$$

$$7n < 7n + 5 \text{ for all } n$$

where C = 2

Thus $\quad f(n) = \Omega(n)$

Example 3.

$$f(n) = 2n^3 + n^2 + 2n$$

$$2n^3 < 2n^3 + n^2 + 2n \quad \text{for all } n$$

where C = 2

So $\quad f(n) = \Omega(n^3)$

Exponential

$$f(n) = 2^n + 6n^2 + 3n$$

So $\quad 2^n (2^n + 6n^2 + 3n) \quad \text{for all } n$

where $\quad C = 1$

So $\quad f(n) = \Omega(2^n)$

Example 1.

$$f(n) = 16$$

$$15 * 1 \leq f(n) \leq 16 * 1$$

$$C_1 = 15, C_2 = 16, n_0 = 0$$

Example 2.

$$f(n) = 3n + 5$$

$$3n < 3n + 5 \text{ for all } n \; C_1 = 3$$

and $\quad 3n + 5 \leq 4n \text{ for } n \geq 5$

$$C_2 = 4, n_0 = 5$$

Thus

$$3n < 3n + 5 \leq 4n$$

$$C_1 = 3, C_2 = 4, n_0 = 5$$

So $\quad f(n) = \Theta(n).$

Big Theta Notation (Θ)

The lower and upper bound for the function f is provided by the big theta notation (Θ).

Definition. Consider 'g' be the function from the non-negative integer in to the positive real number's. Then

$$\Theta(g) = O(g) \cap \Theta(g)$$

i.e., the set of function is both are both in $O(g)$ and $\Omega(g)$.

The $\Theta(g)$ is the set of function 'f' such that for same positive constant C_1 and C_2 and a number exists such that

$$C_1(g(n)) \le f(n) \le C_2(g(n)) \text{ for all } n, n \ge n_0$$

By $f \in \Theta(g)$ we mean "f is order g".

Theorem. If $f(n) = a_m + n^m - n^{m-1} f(n-1) + ... a_1 n + a_0$ and $a_m > 0$, then $f(n) = 0\,(n^m)$

Proof. Using sum of the function we can write above function as

$$f(n) \le \sum_{i=0}^{m} |a_i| n^i$$

$$f(n) \le n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\le n^m \sum_{i=0}^{m} |a_i| \text{ for } m \ge 1$$

So $$f(n) = 0\,(n^m)$$

Theorem Big Oh Ratio Theorem

If $f(n)$ and $f'(n)$ be two function such that $\lim_{n \to \infty} \frac{f(n)}{f'(n)}$ exists, then a function $f \in 0\,(f')$ if $\lim_{n \to \infty} \frac{f(n)}{f'(n)} = C < \infty$; also include the condition where limit is 0.

Proof. Let $f(n) = 0\,(f'(n))$ then for every n; $n \ge n_0$ and C is a positive value and n_0 lies with in limited interval and is constant value then

$$\frac{f(n)}{f'(n)} \le C.$$

Hence $$\lim_{n \to \infty} \frac{f(n)}{f'(n)} \le C$$

Suppose, $\lim_{n \to \infty} \frac{f(n)}{f'(n)} \le C$, then it means '$n_0$' exist and constant.

So $$f(n) \le \max\{1, C\} * f'(n)$$

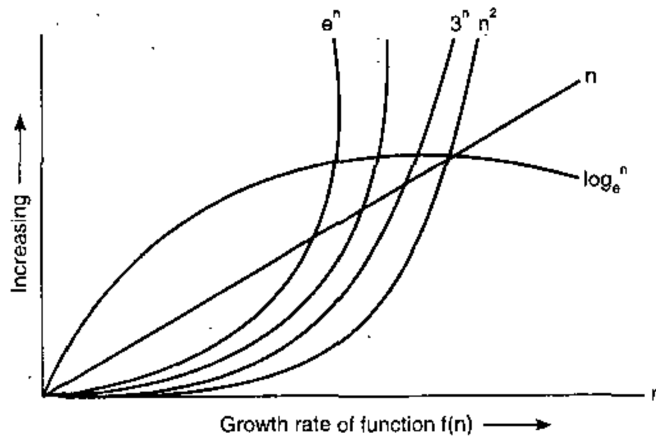
for every n, $$n \ge n_0$$

By Diagram

$$\lim_{n \to \infty} \frac{f(n)}{f'(n)} = \begin{cases} 0 \; f(n) \text{ rises and lower than } f'(n) \\ \infty \text{ rises faster than } f'(n) \\ \text{else both rises or increase with same rate} \end{cases}$$

NOTES

Graph given below shows the growth or rise rate of function.



Where $f(n) = n^2$

$f(n) = 2^n$

$f(n) = 3^n$

$f(n) = \log_e^n$

$f(n) = n$

$f(n) = 3^n$

1.4 DESIGNING OF ALGORITHM

There are many way of designing an algorithm. In which some popular and frequently used approaches are:

1. Divide and conquer approach.
2. Using dynamic programming approach.
3. Greedy Approach
4. Back Tracking approach
5. Branch and Bound
6. Approximate algorithm
7. Randomized algorithm
8. Genetic Algorithm
9. Parallel algorithm

On following above (2), (3) and (4) and rest will be discussed on unit (3) and first approach (divide and conquer is explained here.

1. *Divide and Conquer.* Recursive call of an algorithm to source a problem, they call them selves one or more than one time to deal with closely related sub problems. These algorithm follow *divide and conquer approach.*

They break a problem to sub problem and then re-combine them after getting the solution of the problem and get arranged in original form.

- Divide and conquer problem concepts involves three step at each level they are:

(*i*) Divide (*ii*) Conquer (*iii*) Combine

NOTES

(*i*) *Divide.* Given problem partitioned in to many sub problem

(*ii*) *Conquer.* In this part sub problem we solve the problem recursively call. If a problem is small then problem is treated as a sub-problem.

(*iii*) *Combine.* In this part we arrange all sub-problem in original problem manner respectively.

Example of Divide and Conquer. Merge sort is best suited example of divide and conquer where problem can divided in many sub-problem and latter an after solving we merge all sub problem.

Merge Sort

Merge sort work on principal of divide and conquer where following function takes place.

(*i*) *Divide.* Divide the n-elements series in to two sub series of $n/2$ length.

(*ii*) *Conquer.* Sort the given series using merge. Sort technique recursively calling.

(*iii*) *Combine.* Merge the two sub series after producing sort.

Merge Sort Algorithm

// where A is an Array

Merge sort (A, m, s, r)

$n_1 \leftarrow S - m + 1$

$n_2 \leftarrow r - s$

create array B $[1, 2 \ldots n_1 + 1]$ and C $[1, \ldots n_2 + 1]$

for $i \leftarrow 1$ to n_1

do B $[i] \leftarrow$ A $[m + i - 1]$

for $j \leftarrow 1$ to n_2

do C $[j] \leftarrow$ A $[s + j]$

B $[n_1 + 1] \leftarrow \infty$

C $[n_2 + 1] \leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

for $k \leftarrow m$ to r

do B $[i] \leq$ C $[j]$

then A $[k] \leftarrow$ C$[i]$

$i \leftarrow i + 1$

else A $[k] \leftarrow$ C$[j]$

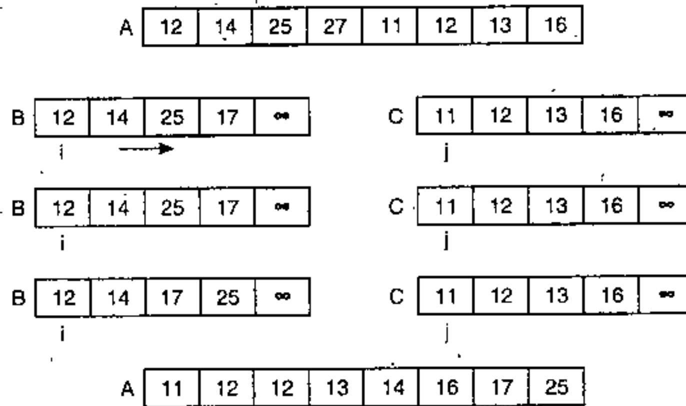
$j \leftarrow j + 1$

Example. *Sort the following number's using merge sort technique*

12, 14, 25, 27, 11, 12, 13, 16

Sol.

NOTES



1.5 MASTER THEOREM

Let $a \geq 1$ and $b \geq 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = a\,T(n/b) + f(n)$$

$T(n)$ being bounded asymptotically as follows

(1) If $f(n) = 0\,(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$ then $T(n) = \theta\,(n^{\log_b a})$

(2) If $f(n) = \theta\,(n^{\log_b a})$ then $T(n) = \theta\,(n^{\log_b a} \log n)$

(3) If $f(n) = \Omega\,(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$, and if $a\,f(n/b) \leq C\,f(n)$ for some constant $C < 1$ and all sufficiently large n. Then $T(n) = \theta(f(n))$

Now we compair the function (above all 3) $f(n)$ with the function $n^{\log_b a}$. The solution to the recurrence is determined by the larger of the two functions.

Case 1. The function $n^{\log_b a}$ is the larger, then the solution is

$$T(n) = \theta\,(n^{\log_b a})$$

Case 2. The two functions are the same size, use multiply by a logarithmic factor, and the solution is

$$T(n) = \theta\,(n^{\log_b a} \log n) = \theta(f(n) \log n)$$

Case 3. The function $f(n)$ is the larger, then the solution is

$$T(n) = \theta(f(n)).$$

Proof of Master's Theorem

Master theorem proof can be analyzes by the recurrence theorem.

$$T(n) = a\,T(n/b) + f(n)$$

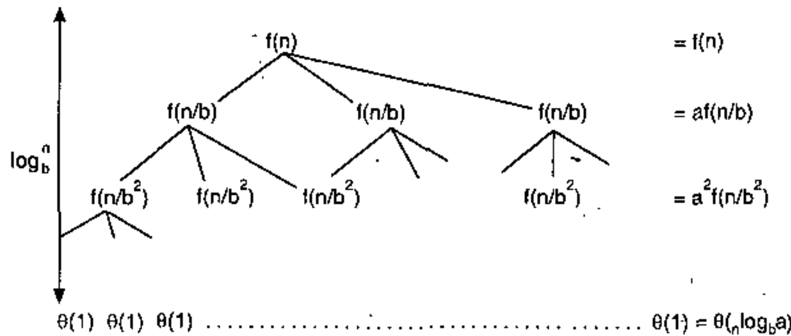
where n is exact power of n , where $b > 1$ and b is need not an integer.

The analysis is parted in to 3 lemmas.

First part. The first reduces the problem of solving the problem of solving the master algorithm by recurrence which an expression that contains a summation.

Second part. It determines bounds on summation *i.e.*, it determines lower and upper bounds {greatest lower bound ($g.l.b$) and least upper bound ($l.u.b$)}.

Third part. This part include first and second part togather to prove a version of master theorem where n is an exact power of b.



$$\text{Total} = \theta(n^{\log_b a}) + \sum_{i=0}^{\log_b^{n-1}} a^i f(n/b^i)$$

$$\boxed{\text{Total} = \theta(n^{\log_b a}) + \sum_{i=0}^{\log_b^{n-1}} a^i f(n/b^i)}$$

Theorem 1. Let $a \geq 1$ and $b \geq 1$ be a constant and let $f(n)$ be a non-negative function defined on exact power of b. Define T(n) on exact power of b by the recurrence

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ a\,T(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

when i is a positive integer, then

$$T(n) = \theta(n^{\log_b a}) + \sum_{i=0}^{\log_b^{n-1}} a^i f(n/b^i)$$

Proof. In above problem we used the recurrsion tree. Where root of tree has cost $f(n)$ and it has a children, each with cost $f(n/b)$. Each of the children contains a children with cost $f(n/b^2)$, and there are a^2 nodes and that being distance 2 from the root and there are a^i nodes with distance i from the root, and each has cost $f(n/b^i)$, and each have cost

$$T(1) = \theta(1)$$

and each leaf is at depth $\log_b{}^n$ and $\frac{n}{b^{\log_b{}^n}} = 1$ and tree contains

$$a^{\log_b{}^n} = n^{\log_b{}^n} \text{ leaves.}$$

which can get the summation of recurrsion tree at each level of the tree and cost of level i of internal nodes is $a^i f(n/b^i)$.

So total cost of all internal nods

$$= \sum_{i=0}^{\log_b^{n-1}} a^i f(n/b^i)$$

NOTES

By divide and conquer algorithm, this sum represents the cost of dividing problem in sub problem and combining all sub-problems have the leave cost with size $1 = \theta(n^{\log_b a})$.

Theorem 2. Let $a \geq 1$ and $b > 1$, be constants and let $f(n)$ be non-negative function defined on exact power of b. A function $f'(n)$ defined our exact power of b by

$$f'(n) = \sum_{i=0}^{\log_b^{n-1}} a^i f(n/b^i)$$

Then be bounded ($g.l.b$ & $l.u.b$) asymptotically for exact power of b as follows

(*i*) If $\quad f(n) = O(n^{\log_b^{a-\varepsilon}})$

$f'(n) = O(n^{\log_b^a})$ for $\varepsilon > 0$, then

(*ii*) If $\quad f(n) = \theta(n^{\log_b^a})$, then $f'(n) = \theta(n^{\log_b^a} \log n)$

(*iii*) If $af(n/b) \leq c.f(n)$ for same constant $c < 1$ and for $n \geq b$, then

$f'(n) = \theta(f(n))$.

Proof. The above all 3 condition being proved one by one

Case (*i*) To prove

$$f(n) = O(n^{\log_b^{a-\varepsilon}})$$

which implies that $f(n/b^i) = O(n/b^i)^{\log_b^{a-\varepsilon}}$ it yields the

$$f'(n) = O\left(\sum_{i=0}^{\log_b^{n-1}} a^i \left(\frac{n}{b^i}\right)^{\log a-\varepsilon}\right)$$

we bound upper limit and lower limit bath and summation within o-notation **and** simplifying factorization, which leaves the geometric series

$$\sum_{i=0}^{\log_b^{(n-1)}} a^i \left(\frac{n}{b^i}\right)^{\log_b^{a-\varepsilon}} = n^{\log_b^{a-\varepsilon}} \sum_{i=0}^{\log_b^{n-1}} \left(\frac{ab^\varepsilon}{b^{\log_b^a}}\right)$$

$$= n^{\log_b^{a-\varepsilon}} \sum_{i=0}^{\log_b^{n-1}} (b^\varepsilon)^i$$

$$= n^{\log_b^{a-\varepsilon}} \left(\frac{b^{\varepsilon \log_b^n} - 1}{b^\varepsilon - 1}\right)$$

$$= n^{\log_b^{a-\varepsilon}} \left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right)$$

where b and ε are constant and above expression can be reduced in to

$= n^{\log_b a-\varepsilon} O(n^\varepsilon) = O(n^{\log_b a})$ and it yields the

$f'(n) = O(n^{\log_b a})$ **Proved**

Case (*ii*) and (*iii*) do yourself.

Theorem 3. Let $a \geq 1$ and $b \geq 1$ be constant and let $f(n)$ be non-negative function defined on exact power of b. Defined $T(n)$ on exact power of b by the recurrence

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ a\,T(n/b) + f(b) & \text{if } b = b^i \end{cases}$$

where i is positive integer. Then $T(n)$ can be (upper and lower bounded) bounded asymptotically for exact power of b as follows

NOTES

(*i*) If $f(n) = O(n^{\log_b^{a-\varepsilon}})$ for constant $\varepsilon > 0$, then $T(n) = \theta(n^{\log_b^a})$

(*ii*) If $f(n) = \theta(n^{\log_b^a})$, then $T(n) = \theta(n^{\log_b^a} \log n)$

(*iii*) If $f(n)$ $\Omega(n^{\log_b^{a+\varepsilon}})$ for $\varepsilon > O$ and if $af(n/b) \leq cf(n)$ for constant $c < 1$ and all sufficiently large n, then $T(n) = \theta(f(n))$.

Proof. Using theorem 2 we get following proof

Case (*i*) $T(n) = \theta(n^{\log_b^a}) + O(n^{\log_b^a})$

$T(n) = \theta(n^{\log_b^a})$

Case (*ii*) $T(n) = \theta(n^{\log_b^a}) + \theta(n^{\log_b^a} . \log n)$

$T(n) = \theta(n^{\log_b^a} . \log n)$

Case (*iii*) Using theorem 2 we conclude that

$T(n) = \theta(n^{\log_b^a}) + \theta(f(n))$

$T(n) = \theta(f(n))$

Since $f(n) = \Omega(n^{\log_b^{a+\varepsilon}})$.

Example of Master's Method

Example 1. *Consider the following recurrence*

$$T(n) = T(3n/4) + 1.$$

find asymptotic bound.

Sol. Comparing with master method.

$T(n) = a\,T(n/b) + f(n)$

$a = 1, f(n) = 1, b = 4/3$

Case 1. $(n)^{\log_b^a} = (n)^{\log_{4/3}^1} = (n)^{o} = 1$

Case 2. $T(n) = \theta(f(n) . \log n)$

$T(n) = \theta(\log n)$

Example 2. *Consider the following*

$$T(n) = 4T(n/2) + n$$

Sol. Using master method

$a = 4, b = 2, f(n) = n$

$(n)^{\log_b^a} = (n)^{\log_2^4} = (n^2)$

Case 1. Since, $f(n) = n\varepsilon\, O(n^{2-\varepsilon})$

$T(n) = O(n^2)$

Case 2. $T(n) = \theta(n^2)$

NOTES

1.6 HEAP SORT (SORTING AND ORDER STATISTICS)

The binary heap data structure is an array object that can be viewed as a nearly complete binary tree.

Heap are two kind of binary tree heaps they are

1. Max heap
2. Min heap.

In both kinds the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max heap, the max heap property is that for every node i other than the root.

$$\text{heap [Parent } (n)] \geq \text{heap } [i]$$

is for max heap, *where parent value will largest value, other than heap nodes*

$$\text{heap [Parent } (n)] \leq \text{heap } [i]$$

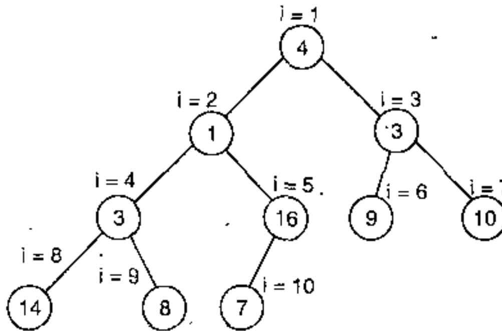
is the min heap, where parent node will be smallest value from rest node value.

"A max or min heap is complete binary tree with the property where the value of each node will be largest or smallest according to max or min heap respectively".

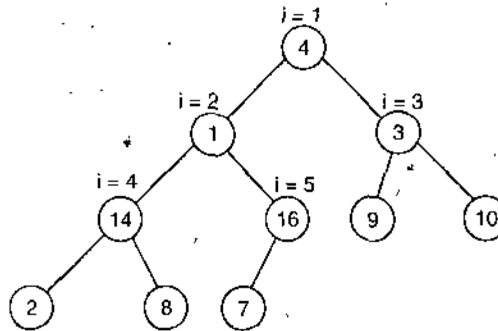
Array A

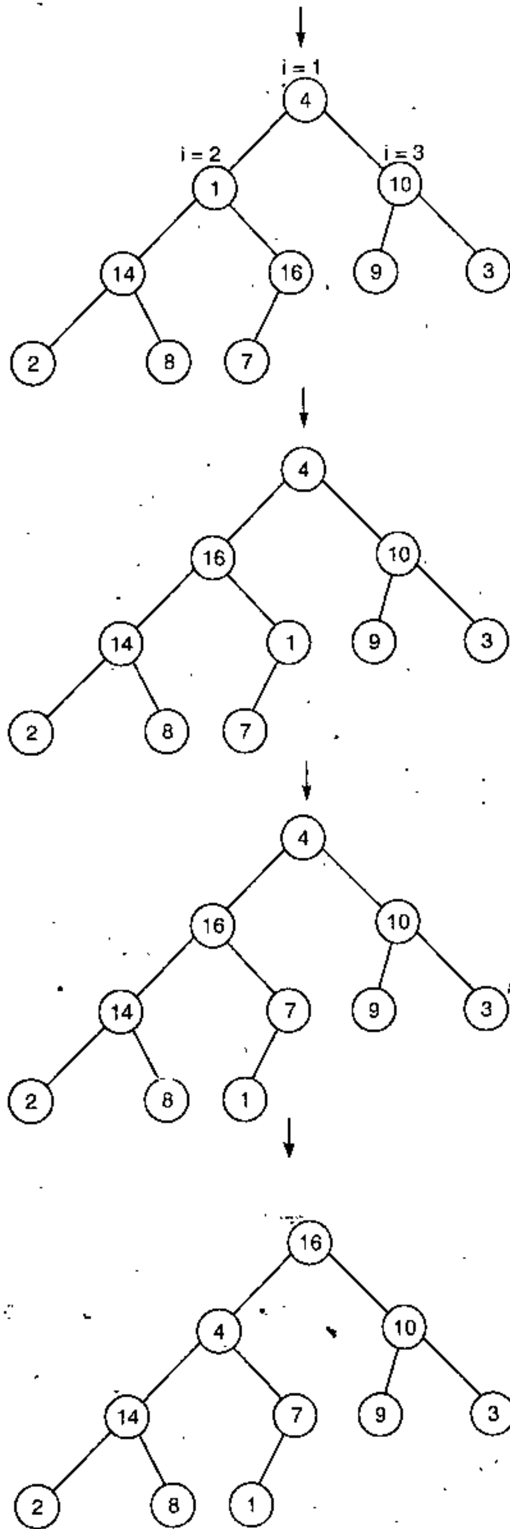
4	1	3	2	16	9	10	14	8	7

Max heap



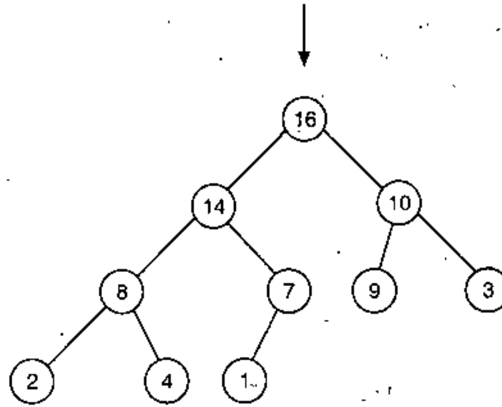
i = 5, 4, 3, 2, 1



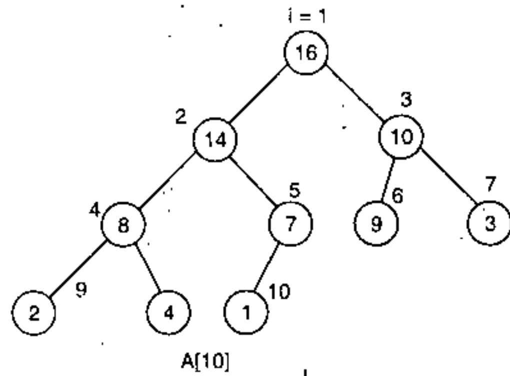


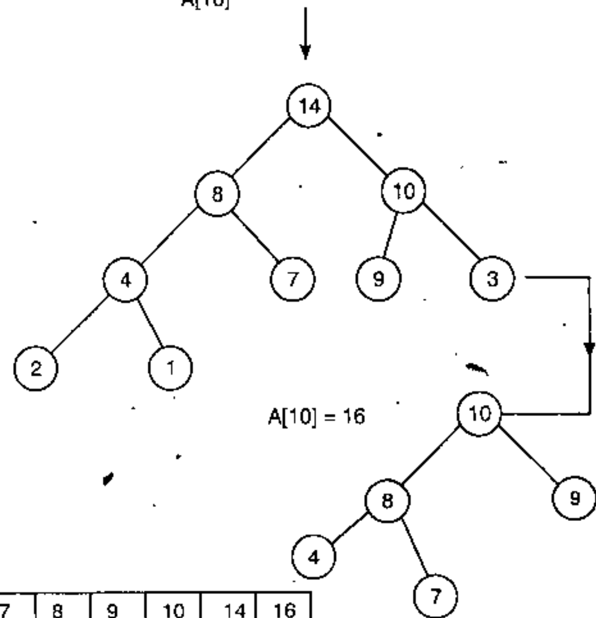
NOTES

NOTES



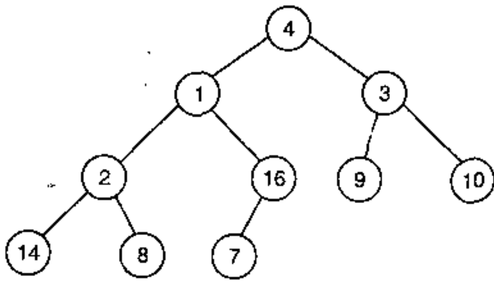
Sorting of Max Heap





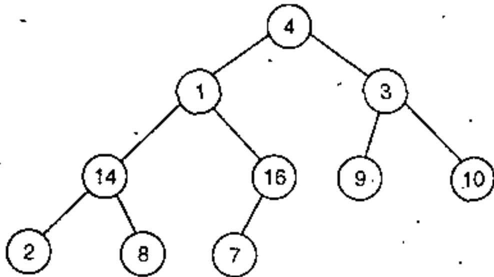
A	1	2	3	4	7	8	9	10	14	16

Nos. are 4, 3, 2, 16, 9, 10, 14, 8, 7

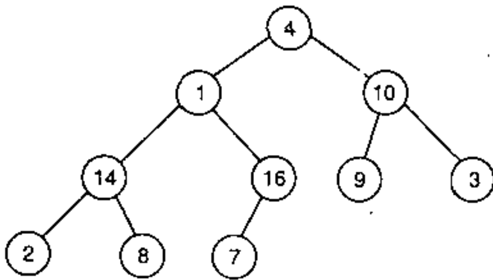


Heap sort (A)
Build-max heap (A)
Max heapify (A, 5)
Max heapify (A, 4)
$i \neq$ largest
Max heapify (A, 8)

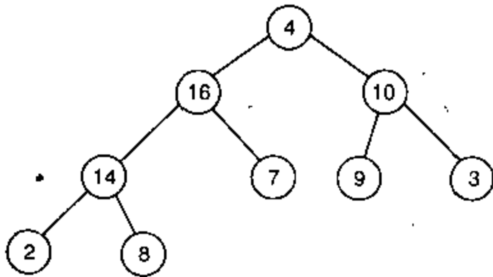
NOTES



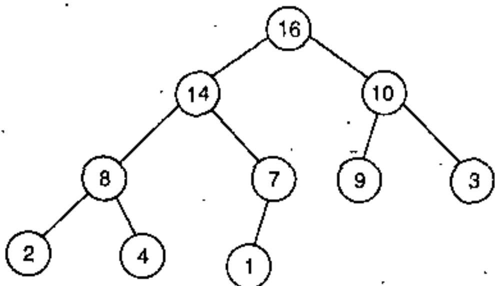
Max heapify (A, 3)
(3) $i \neq$ largest (7)
Max heapify (A, 7)



Max heapify (A, 2)
$i = 2$
largest = 5 (node)
Max heapify (A, 5)



$i = 5$
largest = 10
Max heapify (A, 10)



Max heapify (A, 1)

NOTES

Heap Sort

```
Heap sort (A)
{
n = length of (A);
build max heap (A);
for (i = n, i > = 2, i ...)
{
swop (A[i], A[1]);
Max heapify (A, 1);
}
```

Build-Max-Heap (A)

```
{
n = length of (A);
for (i = n/2; i ≥ 1; i ...)
do
Max heapify (A, i);
}
```

Max heapify (A, i)

```
{
l = 2i;
r = 2i -1;
if (l ≤ heap-size (A) and A[l] > A[i]
then largest = l;
else largest = i;
if (r ≤ Heap-size [A] and A[r] > A [largest]
then largest = r;
if largest ≠ i;
then exchange A[i] ←→ A[largest]
Max heapify (A, largest)
}
```

Quick Sort

Quick sort is similar to the quick sort and is based on the divide-and-conquer paradigm. There is three steps divide-and-conquer process for sorting a array $A[p... n]$

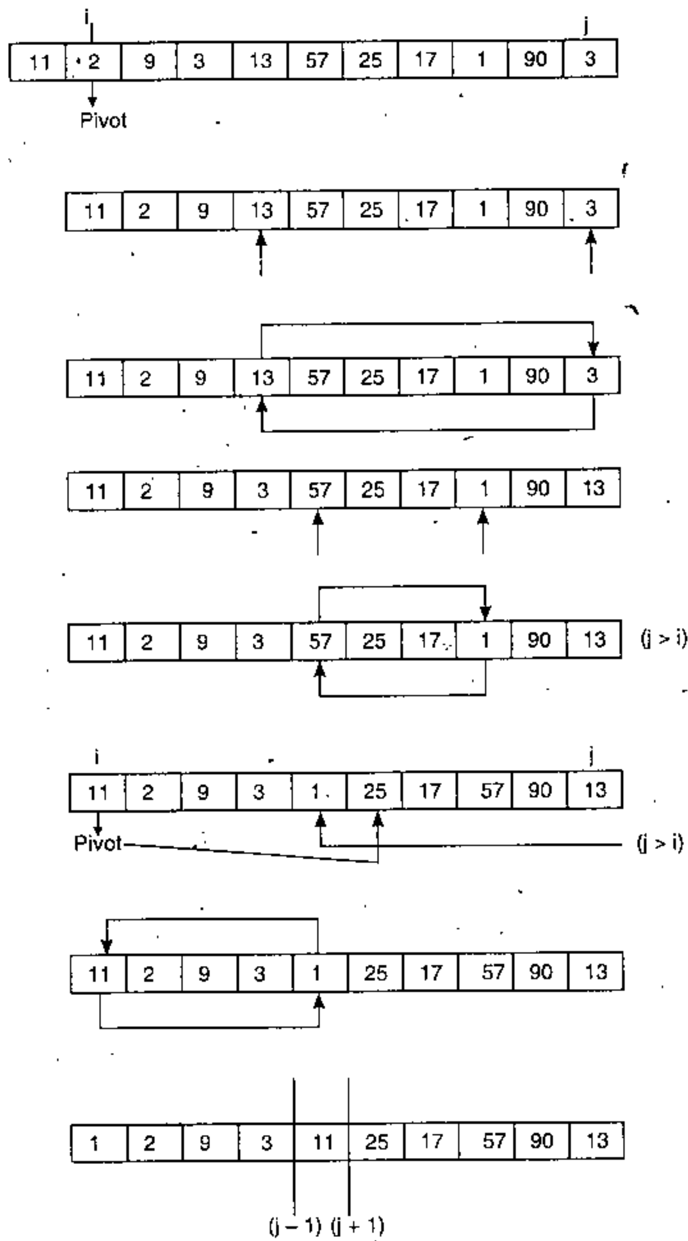
Divide. Partition the array $A[p...n]$ into two sub array $A[p...m]$ and $A[m +2 ... n]$ such that each element of $A[p...m]$ is less than or equal to $A[m+1]$. Which is in turn less than or equal to each element of series of $A[m + 1, ...n]$ and for each sub array we proceed same low.

Conquer. Sort the sub array $A[p...m]$ and $A[m + 2, ...n]$ by recurrsively calling the quick-sort algorithm.

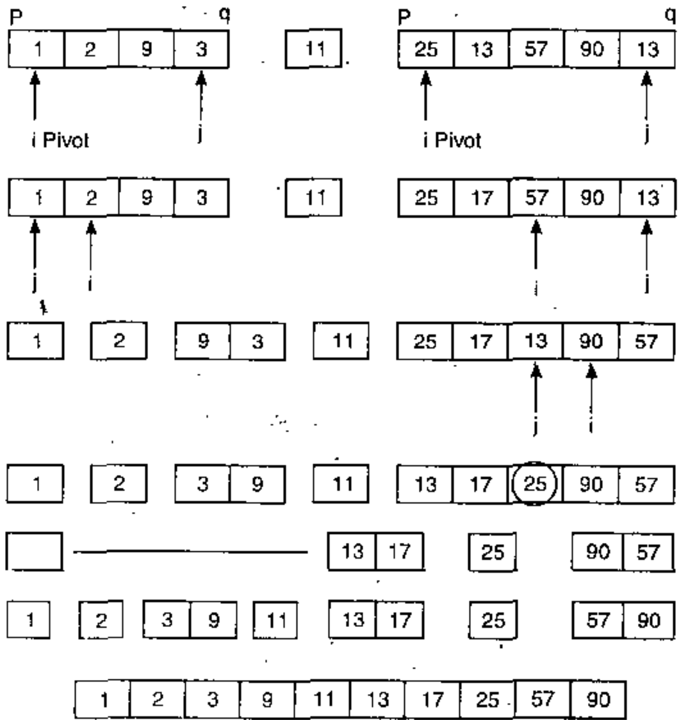
Combine. Since sub array are sorted in place no work is needed to combine them. The entire array A[p, r] is now sorted.

The value of A[m + 1] will be is sorted place *i.e.*, that are in appropriate position no sorting is needed for A[m + 1].

NOTES



NOTES



Algorithm

Quick Sort (p, q)

{

If $(p < q)$ then

{

// divide p into two sub problems

j; = partition $(a, p, q + 1)$

// J is the position of the partioning element

Quick Sort $(p, j-1)$

Quick Sort $(j + 1, q)$

}

}

// Sub Array A[p, r] in place

Partition (A,p,r)

$x \leftarrow$ A [r]

$i \leftarrow p - 1$

for $j \leftarrow p$ to $r - 1$

do if A[j] $\leq x$

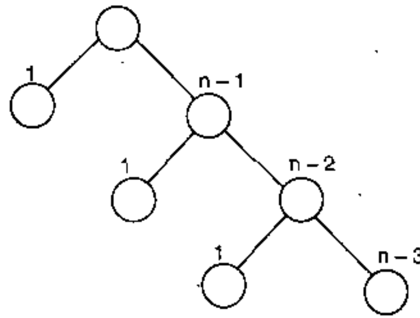
then $i \leftarrow i + 1$

exchange A[i] $\longleftrightarrow$ A[j]

exchange A[$i + 1$] $\longleftrightarrow$ A[r]

return $i + 1$;

Quick-Sort Worst Case Complexity



At a^{th} Level Time for portion of file.

$$T(n) = \max [T(a) + T(n - (a - 1)) + \theta(n)$$
$$0 \le a \le n - 1$$

By substitution

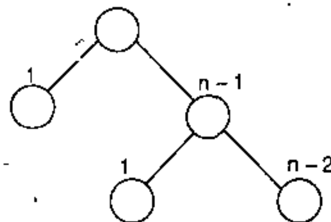
$$T(n) = cn2$$
$$T(n) \le \max [ca^2 + c(n - (a - 1))^2] + \theta cn$$
$$0 \le a \le n - 1$$
$$\le \max [c(a^2 + (n - a + 1)^2) + \theta(n)$$
$$0 \le a \le n - 1$$
$$(a = n - 1)$$
$$\le \max c[(n - 1)^2 + (n - n + 1 - 1)^2] + \theta(n)$$
$$\le \max c[(n - 1)^2 + \theta(n)$$
$$T(n) = \theta(n^2)$$

//Alternative method

Quick Sort-Worst Cases

$$T(n) = P(n) + T(o) + T(n - 1)$$
$$= c.n + o + T(n - 1)$$
$$= c.n + c.\ (n - 1) + T(n - 2)$$
$$= c.n + c.(n - 1) + c.(n - 2) + T(n - 3)$$
$$= c(1 + 2 + 3 + \ldots + n) + T(o)$$
$$= c.n\ (n + 1)/2$$
$$= o(n^2)$$

//P(n) = time to portion the given file.



Best Case of Quick-Sort

When file is always divided in half (*i.e.*, $n/2$ part)

$$T(n) = P(n) + 2\ T(n/2)$$
$$T(n) = c.n + 2T\ (n/2)$$

NOTES

By substitution method $T(n) = o(n\log_2{}^n)$

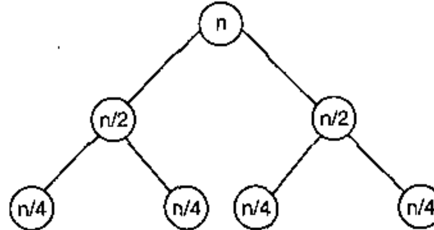
$$T(n) \le cn\log_2{}^n$$

$$T(n) \le c.n + 2\left(c\frac{n}{2}\log_2\frac{n}{2}\right)$$

$$T(n) \le c.n + cn\log_2{}^n - c.n\log_2{}^2$$

$$T(n) \le cn\log_2{}^n$$

$$T(n) = O(n\log_2{}^n)$$



1.7 SORTING IS LINEAR TIME

In linear time sorting, it can be categorised in following two category:

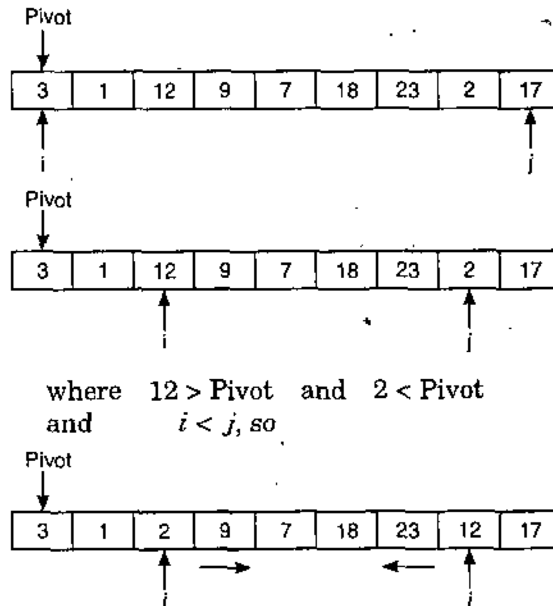
- Comparision based sorting
- Non-comparision based sorting.

Comparision Based Sorting

In this sorting, sorting performs with comparision of values where two key value play an important role.

Example:

In quick-sort there is a pivot value and two variable i and j are the key value's and i will be from left side and i approaches from right side and if i > pivot and i < pivot then they interchange (exchange) the value, where $i < j$.



Lower Bound for Sorting. In comparision based sorting, sorted elements or number is based only on comparision between input number and we generate a tree for

there element that is called decision tree and there run time is $\Omega(\log n)$ and comparision of sequence $\{n_1, n_2, ..., n_n\}$ have the following property

$$n_i < n_j, n_i \le n_j, n_i = n_j \quad \text{or} \quad n_j > n_i, n_j \ge n_i, n_j = n_i$$

NOTES

Non-comparision Based Sorting

Non-comparision based sorting have three type's of sort

(*i*) Bucket sort

(*ii*) Counting sort

(*iii*) Radix sort

(*i*) Bucket Sort

Bucket sort taken place when input be uniformaly drawn or taken.

Bucket sort assumes that the input consist of integer in a small range.

On bucket sort principal behind the concept is that numbers are randomely generated. So there will be chance of repeating element be very less or rase.

Idea of bucket sort is to divide in the interval [0, 1) in to n equal-sized sub intervals or buckets and then distribute the number on bucket.

And then we sort each bucket using any sorting method.

We assume that there are n-elements on array A and each element A[i] in the array satisfies $0 \le A[i] < 1$.

And code required an auxiliary array B[0, n – 1] of link list.

Bucket Sort

Bucket Sort (A)

$n \leftarrow$ length(A)

for $i \leftarrow 1$ to n

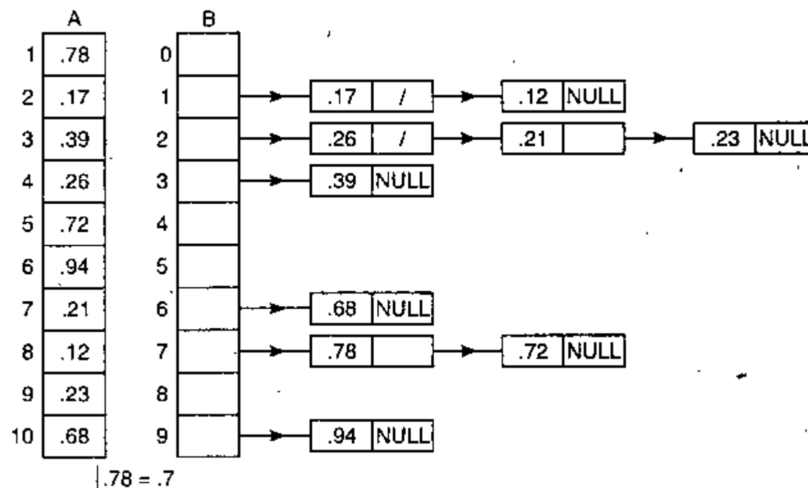
do

insert A[i] into list B [$\lfloor n(i) \rfloor$]

for $i \leftarrow 0$ to $n - 1$

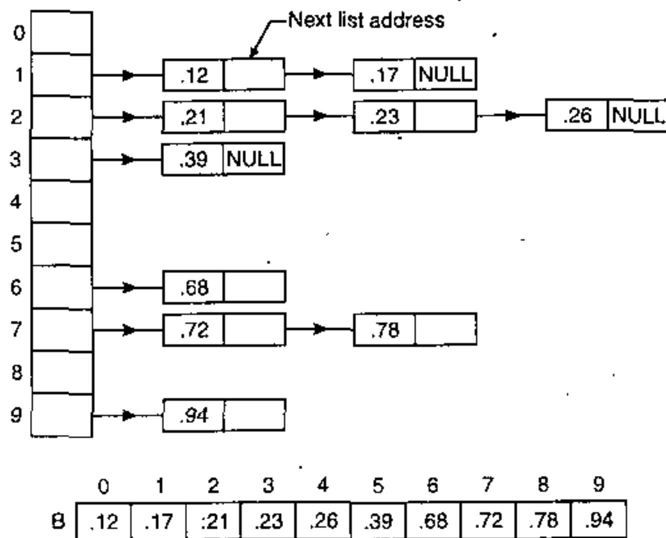
do

sort list B[i] with insertion sort concatenate the lists B[0], B[1], ... B[n – 1] together in order.



NOTES

Sorted Bucket



Complexity

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O(ni)^2$$

Taking expectation of both side with using linearity of expectation, we have

$$E(a + b) = E(a) + E(b)$$

$$E(T(n)) = E\left[\theta(n) + \sum_{i=0}^{n-1} O(ni)^2\right]$$

$$= \theta(n) + \sum_{i=0}^{n-1} E\,[O(ni)^2] \quad \text{By linearity of expectation}$$

$$= \theta(n) + \sum_{i=0}^{n-1} O\,(E\,[ni]^2) \quad \text{Solving this we get}$$

$$= \theta(n) + 2 - \frac{1}{n}$$

So time complexity of bucket sorting is linear on n, *i.e.*, $\theta(n)$.

Example. *Bucket sort, with bucket of equal size, runs in average O(n) time even if the input numbers are not chosen from uniform distribution.*

Ans. Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time as long as the input has property that the sum of the squares of the bucket sizes is linear in the total number of elements.

(ii) Counting Sort

This sort assume that numbers given for sorting are integer numbers where they have the range in between 0 to k. Where k is the highest value for sorting *i.e.*, highest number for sort this sorting technique in general is used for small number due to that if value of

k be large than array c which have index from 0 to k have many index value, so it become complicated to evaluate the all values position any many more memory for sorting be not used but occupied the space.

NOTES

(iii) Radix Sort

This sort is the algorithm used for the card-sorting machines and used in computer museum.

"It is an integer sort algorithm which assume that each and every integer consist of d digits, and each digits is range from 1, 2, 3, ... k portion.

i.e., 10, 13

where d is 10,

$\because$ 10^n where $n = 1$

Example of Counting Sort

Array

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

there find greatest number $k = 5$

	1	2	3	4	5	6	7	8
B								

This one will be find sorted list

and we take another temporary array

	0	1	2	3	4	5
C	2	0	2	3	0	1

from 0 to K

	0	1	2	3	4	5
C	2	2	4	7	7	8

	0	1	2	3	4	5
C	1	2	4	6	7	8

	1	2	3	4	5	6	7	8
B		0					3	

A	2	5	3	0	2	3	0	3

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Counting sort (A, B, K)

for $i \leftarrow$ 0 to K

do c [i] $\leftarrow$ 0

for $j \leftarrow$ 1 to length [A]

do c [A [i]] $\leftarrow c$ [A [i]] + 1

c [i] contains the number of element equal to i.

do

c [i] $\leftarrow c$ [i] + c [i – 1]

c [i] containing the number less than or equal to i.

for $j \leftarrow$ length [A] down to 1

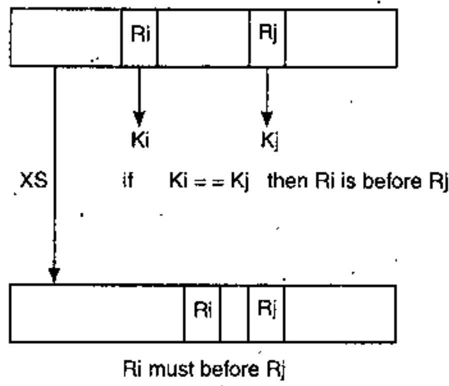
do B [c [A [j]]] ← A [i]
c [A [i]] ← c [A [i]] – 1

Radix Sort

NOTES

Radix sort (A, d)
for i ← 1 to d
do
use a stable sort to sort array A on digit i.
// Where d is digit numbers
// Help sorting is not stable sort
// Quick sort is stable sort.

Example:



Ri must before Rj

And on sorting concept

$$n\% \, 10^i$$

Example:

129	721	721	129
357	893	129	354
657	354	435	357
739	435	739	435
435	357	354	657
721	657	357	721
354	129	657	739
893	739	893	893
$n\%10$	$n\%100$	$n\%100$	

We divide number by 10^i where $i = 1, 2, \ldots\ldots$ highest degree of the number and arrange then according to increasing remainder value, if remainder value is same, them which number come first, that is written first and this process is repeated upto highest degree of number *i.e.,* for *435* we divide upto 10^3 *i.e.,* upto 1000 from 10. Sequentially 10, 100 then 1000.

NOTES

1.8 MEDIAN AND ORDER STATISTICS SORTING

Median is the half way of the point *i.e.*, it lies in between minimum and maximum set of elements. When ever n is odd then median is unique.

$$\text{median} = \left[\frac{\text{total number of element}}{2}\right]$$

whenever n is odd,

i.e., when value of set is

$$S = \{3, 27, 9, 18, 13, 37, 48\}$$

$$\text{then median} = \left[\frac{7}{2}\right] = [3.5] = 4$$

median will be 4th position value that is 18.

When element of set is even the median is not unique it will more than one value, that is 2 median value.

Example:

If set of sequence S is

$$S = \{13, 18, 17, 6, 14, 5, 19, 15\}$$

where number θ

$$\text{So median} = \frac{n}{2}, \frac{n}{2} + 1$$

hence median will be 6 and 14.

$$\text{Median class of odd number of classes} = \frac{n}{2}$$

$$\text{Median class of even number of classes} = \frac{n}{2} \text{ and } \left(\frac{n}{2} + 1\right)$$

Whenever numbers are given in interval the median can be calculated as follows

$$\text{Median} = L + \left(\frac{\frac{N}{2} - f}{f}\right) \times h$$

where

L = lower limit of median class

N = E cfi (frequency sum)

F = Cumulative frequency preceeding the median class.

f = Frequency of median class

h = class interval.

STUDENT ACTIVITY

1. Discuss time and space complexity of a program.

2. What is algorithm ? Why do we study algorithms ?

NOTES

SUMMARY

1. An algorithm is a finite set of instruction that if followed, accomplishes a particular task.
2. Space complexity of an algorithms is the amount of memory it needed to run to completion of program.
3. The time complexity of an algorithm is the amount of computer time it needed to run to completion.
4. If we are using any arithmetic operation then it uses one time computation. but if we are using loop which are not in single step loops then counting of run time exceed.
5. Worst case time complexity is the function defined by maximum amount of time needed by an algorithm for an input of size 'n' thus it is the function defined by the maximum number of steps taken on any instance of size 'n'.
6. The average case time complexity is the execution of an algorithm having typical input data of size 'n'. Thus it is the function defined by the average number of steps taken on any instance of size n.
6. The best case time complexity is the minimum amount of time that an algorithm requires for an input of size 'n'. Thus it is the function defined by minimum number of step taken on any instance of size n.
7. In linear time sorting, it can be categorised in following two category:
   - Comparision based sorting
   - Non-comparision based sorting.

GLOSSARY

- *Max or Min Heap:* A max or min heap is complete binary tree with the property where the value of each node will be largest or smallest according to max or min heap respectively.
- *Quick Sort:* Quick sort is similar to the quick sort and is based on the divide-and-conquer paradigm.
- *Counting Sort:* This sort assume that numbers given for sorting are integer numbers where they have the range in between 0 to K.
- *Median:* Median is the half way of the point *i.e.,* it lies in between minium and maximum set of elements.

REVIEW QUESTIONS

1. Which of the following statement are true? Prove your answer.

   (*a*) $n^2 \in O(n^3)$ (*b*) $n^2 \in \Omega(n^3)$

   (*c*) $2^n \in \theta(2^{n+1})$ (*d*) $n! \in \theta(n+1)!$
2. Arrange the following growth rates in the increasing order.

   $O(n^3)$, $O(1)$, $O(n^2)$, $O(n \log n)$, $O(n^2 \log n)$, $\Omega(n^{0.5})$, $\Omega(n \log n)$, $\theta(n^3)$, $\theta(n^{0.5})$.
3. What do you mean by analysis of an algorithm?

NOTES

4. Analyse the insertion sort in worst case.
5. What is divide and conquer approach? Analyse the merge sort algorithm?
6. Define the following terms:
   (*a*) θ-notation
   (*b*) O-notation
   (*c*) Ω notation
   (*d*) Little-oh notation
   (*e*) Little-omega (ω) notation.
7. What is recurrence?
8. What is stable sorting algorithm? Give the names.
9. Write an algorithm median(s) to get the median element from the sequence S of n elements.

FURTHER READINGS

- Sachin Dev Goyal, *'Design and Analysis of Algorithm'*, University Science Press.
- Hari Mohan Pandey, *'Design Analysis and Algorithm'*, University Science Press.

UNIT

2

DATA STRUCTURES

NOTES

STRUCTURE

2.0 Objectives
2.1 Introduction
2.2 Stacks
2.3 Queue
2.4 Linked Lists
2.5 Binary Tree
2.6 Hash Table
2.7 Binary Search Tree (BST)
2.8 Querying a Binary Search Tree
2.9 Insertion and Deletion
2.10 Red-Black Tree
2.11 Augmenting Data Structure
2.12 Binomial Heap
2.13 Data Structure for Disjoint Set
2.14 Splay Trees
2.15 Priority Queue
- *Summary*
- *Glossary*
- *Review Questions*
- *Further Readings*

2.0 OBJECTIVES

After going through this unit, you will be able to:

- discuss about stacks and queues.
- define types of linked list.
- describe the Binary search tree, Red-black tree, Splay tree and B tree.
- explain augmenting data structure and data structure box disjoint sets.

PART I: ELEMENTARY DATA STRUCTURES

2.1 INTRODUCTION

Data structure is representation of the logical relationship existing between individual elements of data. In other words, a data structure is a way of organizing all data items

that considers not only the elements stored but also their relationship to each other. There are the following four things to specifies data structures.

(*i*) Organization of data; (*ii*) Accessing methods; (*iii*) Degree of associativity; and (*iv*) Processing alternatives for informations.

The most commonly used operations on data structure are the following:

SEARCH (S, *k*): A query that, given a set S and a key value *k*, returns a pointer *x* to an element in S such that key $[x] = k$, or NIL if no such element belongs to S. Searching operation finds the presence of the desired data item in the list of data item. It may also find the location of all elements that satisfy certain conditions.

INSERT (S, *x*): A modifying operation that augments the set S with the element pointed to by *x*. We assume that any fields in element *x* needed by the set implementation have already been initialized.

DELETE (S, *x*): Given a pointer *x* to an element in the set S, remove *x* from S. This operation destroys memory space allocated for the specified data structure Malloc ( ) and Free ( ) function of *c* language are used for these two operations respectively.

MINIMUM (S): A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM (S): A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

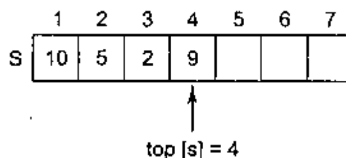
SUCCESSOR (S, *x*): A query that, given an element *x* whose key is form a totally ordered set S, returns a pointer to the next integer element in S, or NIL if *x* is the maximum element.

PREDECESSOR (S, *x*): A query that, given an element *x* whose key is form a totally ordered set S, returns a pointer to the next integer element is S or NIL if *x* is the minimum element.

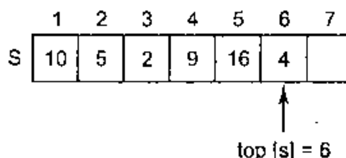
2.2 STACKS

A stack is a non-primitive data structure. In a stack, the element deleted from the set is the one most recently inserted. Stack implements a last in, first out, or LIFO, policy. The insert operation on a stack is often called PUSH, and the Delete operation, which does not take an element argument, is often called POP.

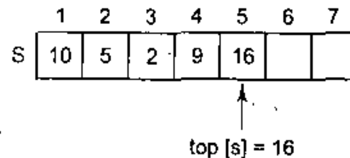
For example, an array implementation of a stack S. Stack S has 4 elements. The top element is 9.



Now, in the above figure, we calls PUSH (S, 16) and PUSH (S, 4).



After this, we call POP (S) has returned the element 4, which is one most recently pushed. Element 4 still appears in the array, it is no longer in the stack the top is element 16.



NOTES

We can implement a stack of at most n elements with an array S [1 ... n]. The array has an attribute top [S] that indexes the most recently inserted element. The stack consists of elements S[1 .... top [S], where S [1] is the element at the bottom of the stack and S [top (S)] is the element at the top.

When top [S] = 0, the stack contains no elements and is empty. The stack can be tested for emptiness. If an empty stack is poped, we say the stack underflow, which is normally an error. If top [S] exceeds n, the stack overflows.

The stack operations can each be implemented as:

STACK-EMPTY (S)

1. If top [s] = 0
2. then return TRUE
3. else return FLASE

PUSH (S, x)

1. top [S] ← top [S] + 1
2. S [top (S)] ← x

POP (S)

1. If STACK-EMPTY (S)
2. then error "underflow"
3. else top [S] ← top [S] – 1
4. return S [top [S] + 1]

PUSH and POP operation takes O (1) time.

Applications of Stacks

There are basically three types of notations for an expression:

1. Infix notation
2. Prefix notation
3. Postfix notation.

In Infix notation, where the operator is written in-between the operands. For example, The expression to add two numbers C and D is written in infix notation as:

C + D

In the prefix notation, a notation in which the operator is written before the operands, it is also called polish notation is the honor of the mathematician Jan Lukasiewicz who developed this notation. For example,

+ CD

As the operator '+' is written before the operands C and D, this notation is called Prefix.

The postfix notation, the operators are written after the operands, so it is called the postfix notation. It is also called as suffix notation or reverse polish notation. For example:

CD +

Example. *Convert the following infix expression to postfix form for*

*A + [(B + C) + (D + E) * F] | G*

Sol.

A + [(B + C) + (D + E) * F] | G

A + [(BC +) + (DE +) * F] | G

A + [(BC +) + (DE + F *)] | G

A + [BC + (DE + F* +)] | G

A + [BC + DE + F * + G |]

ABC + DE + F * + G | +   Postfix form

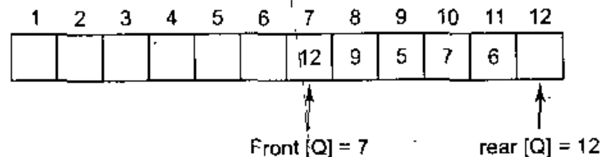
2.3 QUEUE

Queue means a line. A queue is logically a first in first out (FIFO) type of list. In a *queue, new elements are added to the queue from one end called* REAR end, and the elements are always removed from other end called the FRONT end. For example, the people standing in a railway reservation row. Each new person comes and gets the ticket first and get out of the row from the front end.

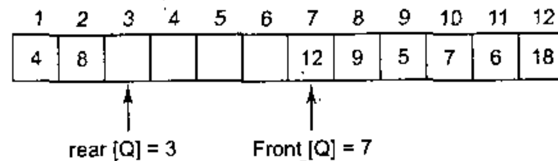
Queues can also be implemented in two ways:

(*i*) Using arrays   (*ii*) Using pointer.

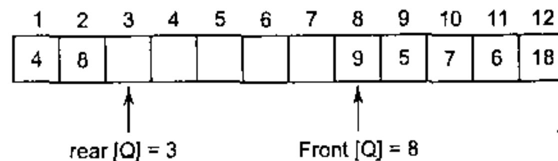
A queue implemented using an array Q [1 ... 12] in the following figure; the queue has 5 elements in locations Q (7 ... 11).



We call the INSERT operation on a queue ENQUEUE and we call the DELETE operation DEQUEUE. The configuration of the queue after the calls ENQUEUE (Q, 18), ENQUEUE (Q, 4), and ENQUEUE (Q, 8).



Now, the configuration of queue after the call DEQUEUE (Q) returns the key value 12 formerly at the front of the queue. The new head has key 9.



Algorithm for inserting elements in queue.

ENQUEUE (Q, K)

1. Q [rear [Q]] ← *x*
2. If rear [Q] = length [A]
3. *then rear [Q] ← 1*
4. else rear [Q] ← rear [Q] + 1

NOTES

ENQUEUE operation takes O (1) time. It is noticeable that the error checking for underflow and overflow has been omitted.

Algorithm for deleting elements in queue.

DEQUEUE (Q)

1. $x \leftarrow$ Q [Front [Q]]
2. If front [Q] = length [Q]
3. then front [Q] $\leftarrow$ 1
4. else front [Q] $\leftarrow$ front [Q] + 1
5. return x.

DEQUEUE operation also takes O (1) time. It is note that the underflow and overflow conditions are not checked.

Example. *Show how to implement a queue using two stacks. Analyze the running time of the queue operations.*

Sol. To implement a queue using two stacks. There are two stacks and it is denoted by A_1 and A_2. The ENQUEUE operation is implemented whenever a push operation call on A_1 and the DEQUEUE operation is simply implemented when a pop operation call on A_2. If A_2 is empty, successively pop A_1 and push A_2. It reverses the order of A_1 onto A_2. The running time in worst case is O (n).

Q-INSERT (x)

1. if (top = max)
2. then error "overflow"
3. else A_1 [top + 1] $\leftarrow x$

Q-DELETE ( )

1. if (top = 0)
2. then error "under flow".
3. Flag $\leftarrow$ 0 (return)
4. else S $\leftarrow$ 0
5. while (top < 1)
6. do $x \leftarrow$ pop (A_1)
7. PUSH (A_2, x)
8. S $\leftarrow$ S + 1
9. top $\leftarrow$ top – 1
10. $y \leftarrow$ POP (A_2)
11. top $\leftarrow$ top – 1
13. While (S ≠ 0)
14. do X $\leftarrow$ POP (A_2)
15. PUSH (A_1, X)
16. top $\leftarrow$ top + 1
17. return y.

NOTES

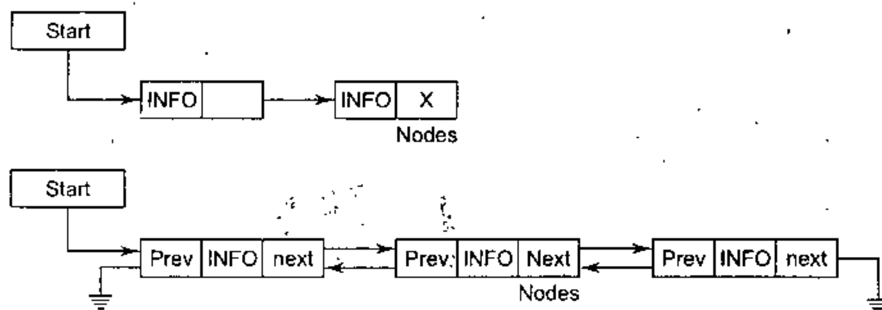
2.4 LINKED LISTS

Linked lists are special list of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each element is called a node, which has two parts. INFO part which stores the information and POINTER which points to the next element.

There are following types of linked lists:

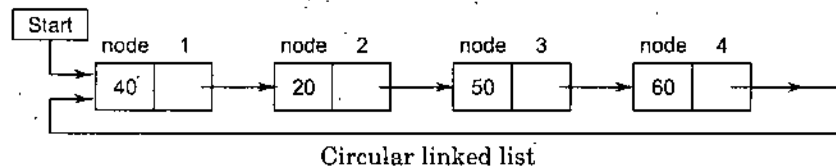
1. Singly linked lists
2. Doubly linked lists
3. Circular linked list
4. Circular doubly linked lists.

The following figure shows both types of linked list (Singly linked lists and Doubly linked lists).



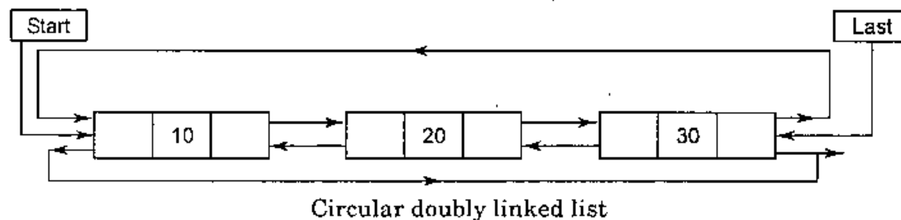
In singly linked list nodes have one pointer (Next) pointing to the next node, whereas nodes of doubly lists have two pointers (prev and next). Prev points to the previous node and next points to the next node in the list.

A circular linked list is one which has no beginning and no end. A singly linked list can be made a circular linked list by simply sorting the address of the very first node in the link field of the last node.



Circular linked list

A circular doubly linked list is one which has both the successor pointer and predecessor pointer in circular manner.



Circular doubly linked list

Searching a Linked List

LIST-SEARCH (L, K) is a process to finds the first element with key K in list L by a simple linear search returning a pointer to this element. If no objects with key K appears in the list, then NIL is returned. LIST-SEARCH procedure takes $\theta(n)$ time to search a list of n objects in the worst case.

NOTES

LIST-SEARCH (L, K)

1. $x \leftarrow$ head [L]
2. While $x \neq$ NIL and key [x] $\neq$ K
3. do $x \leftarrow$ next [x]
4. return x.

Inserting into a Linked List

LIST-INSERT procedure is used to insert a new node x onto the front of the linked list.

LIST-INSERT (L, K)

1. next [x] $\leftarrow$ head [L]
2. if head [L] $\neq$ NIL
3. then prev [head [L]] $\leftarrow x$
4. head [L] $\leftarrow x$
5. prev [x] $\leftarrow$ NIL

The running time for LIST-INSERT on a list of n elements is O (1) time.

Deleting from a Linked List

LIST-DELETE operation is used to delete an item (an element or node) x from a linked list L. It must be given a pointer to x, and it then splices x out of the list by updating pointers.

LIST-DELETE (L, x)

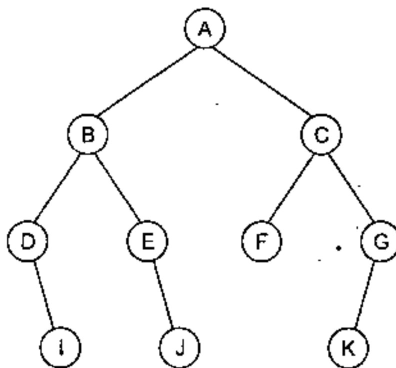
1. If prev [x] $\neq$ NIL
2. then next [prev (x)] $\leftarrow$ next [x]
3. else head [L] $\leftarrow$ next [x]
4. if next [x] $\neq$ NIL
5. then prev [next [x]] $\leftarrow$ Prev [x]

LIST-DELETE operation runs in 0 (1) time, but if we want to delete an node or element with a given key, θ (n) time is required in the worst case because we must first call LIST-SEARCH.

2.5 BINARY TREE

A binary tree is a finite set of data items which is either empty or consists of a single item called the root and two disjoint binary trees called the left subtree and right subtree.

In a binary tree, the maximum degree of any node is at most two. That means, there may be a zero degree node or a one degree node and two degree node.



A binary tree.

NOTES

Example. *Augment a linked list data structure, where each node maintains the number of preceding nodes.*

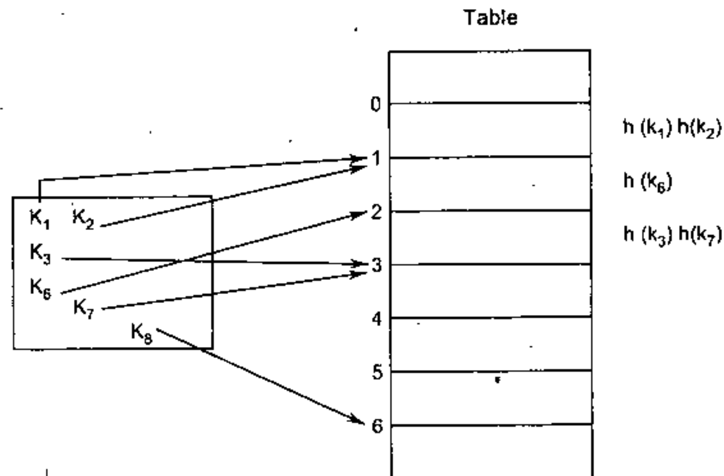
Sol. LIST-INSERT (L, X)

1. next [x] ← head [L]
2. If head L ≠ NIL
3. then prev [head [L]] ← x
4. head [L] ← x
5. prev [N] ← NIL

2.6 HASH TABLE

We have seen that the dictionary operations INSERT, SEARCH and DELETE takes θ (n) time if the data are searched linearly and there are n items stored in records. Hashing is an approach, in which we compute the location of the desired record in order to retrieve it in a single access. This avoid the unnecessary comparison. In this method, the location of the desired record present in the search table depends only on the given key but not on other keys.

A hash table is an effective data structure where we store a key value after applying the hash function it is arranged in the form of an array that is addressed via., a hash function. The hash table is divided into a number of buckets and each bucket is in turn capable of storing a number of records. Thus we can say that a bucket has number of slots and each slot is capable of holding one record.



The time required to locate any element in the hash table is O (1). It is constant and it is not depend on the number of data elements stored in the table.

Hash Function

The basic idea is hashing is the transformation of a key into the corresponding location in the hash table. This is done by a hash function. A hash function can be defined as a function that take key as input and transforms it into a hash index. It is denoted by H.

$$H: K \rightarrow M$$

where H is a hash function
K is a set of keys
M is a set of memory addresses.

Sometimes, a function H may not yield distinct values, it is possible that two different keys K_1 and K_2 will yield the same hash address. This situation is called hash collision.

NOTES

Types of Hash Function

Different hash functions are available. To choose the hash function $H: K \rightarrow M$ there are two things to consider. Firstly, the function H should be very easy and quick to compute. Secondly, the function H should distribute the keys to the number of locations of hash table with less no of collisions.

(*i*) Division reminder method

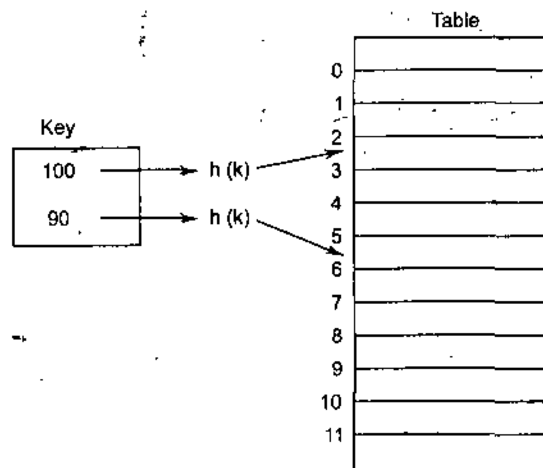
(*ii*) Mid square method

(*iii*) Folding method.

Division Reminder Method

In this method for creating hash function, the key value k is divided by a number m larger than the number n of keys in k and the reminder of this division is taken as index into the hash table, *i.e.*, the hash function is,

$$h(k) = k \bmod m$$



Example. *Consider a hash table with 12 slots i.e., m = 12 then hash function h (k) = k mod m will map the key 100 to slot 1*

Since, $h(100) = 100 \bmod 12 = 4$

Similarly, $h(90) = 90 \bmod 12 = 6$

Mid Square Method

In this mid square method, first the key is squared. Then the hash function is defined by

$$h(k) = p$$

where p is obtained by deleting digits from both sides of k^2.

Example. *Consider a hash table with 50 slots i.e., m = 50 and key values k = 1632, 1739, 3123.*

Sol.

k:	1632	1739	3123
k^2:	2663424	3024121	9753129
$h(k)$:	34	41	31

The hash values are obtained by taking the fourth and fifth digits counting from right.

Folding Method

In folding method, the key k is partitioned into a number of parts $k_1, k_2, \ldots k$, where each part, except possibly the last, has the same number of digits as the required address. Then the parts are added together, ignoring the last carry *i.e.*,

$$H(k) = k_1 + k_2 + \ldots + k_r$$

where the leading-digits carrier, if any are ignored.

Example. *Consider a hash table with 100 slots i.e., m = 100 and key values k = 7325, 76321, 1623, 7613.*

Sol.

NOTES

k	Parts	Sum of parts	$h(k)$
7325	73, 25	98	98
76321	76, 32, 1	109	09
1623	16, 23	39	39
7613	76, 13	89	89

Universal Hashing

The universal hasing is used to select the hash function at random from a carefully designed class of functions at the beginning of execution. Randomization guarantees that no single input will always evoke the worst case behaviour, like as quick sort poor performance occurs only when the compiler chooses a random hash function that causes the set of identifiers to hash poorly, but the probability of this situation occurring is small and is the same for any set of identifiers of the same size.

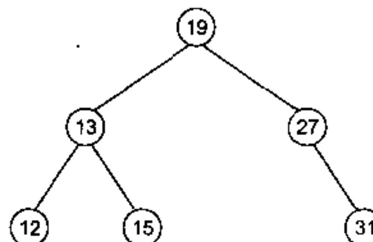
Let us consider H is a finite collection of hash functions that map a given universe U of key into the range (0, 1, ..., $m-1$). We say that a collection is universal it. For each pair of distinct keys $k, y \in$ U, the number of hash function $h \in$ H for which $h(k) = h(y)$ is at most H1 | M.

2.7 BINARY SEARCH TREE (BST)

A search tree can be used both as a dictionary and as a priority queue. Basic operations on a binary search tree with n nodes, such operations run in $\theta(\lg n)$ worst-case time. A binary search tree is organized in a binary tree. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a key field and satellite data, each node contains fields left, right and p that points to nodes the corresponding to its left child, its right child, and its parent. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is only the node in the tree whose parent field is NIL.

A Binary Search Tree (BST) is a binary tree which is either empty or satisfies the following rules:

1. The value of the key in the left child or left subtree is less than the value of the root.
2. The value of the key in the right child or right subtree is more than or equal to the value of the root.



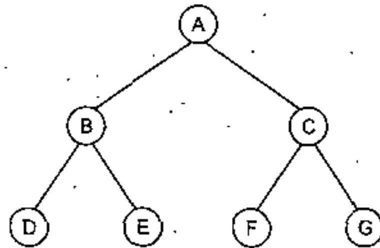
Binary Search Tree

In the above figure, the key of the root is 19, the keys 12, 13 and 15 in its left subtree are no longer than 19, and the keys 27 and 31 in its right subtree are no smaller than 19. The binary search tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk. This algorithm is so named because the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree. Similarly, in case of a preorder tree walk, it prints the root before the values in either subtree, and in postorder tree walk, it prints all the elements in a binary search tree T, we call INORDER-TREE-WALK (root [T]).

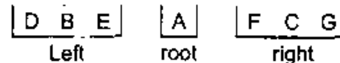
INORDER-TREE-WALK (x)

1. if $x \neq$ NIL
2. then INORDER-TREE-WALK (left [x])
3. point key [x]
4. INORDER-TREE-WALK (right [x])

Example,



The inorder traversal of the bring tree is



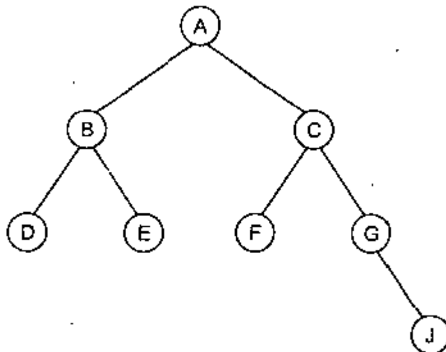
INORDER-TREE-WALK (x) takes θ (n) time to walk an n-node binary search tree, since after the initial call, the procedure is called recursively exactly twice for each node in the tree — once for its left child and once for its right child.

PREORDER-TREE-WALK (x)

1. if $x \neq$ NIL
2. then print key [x]
3. PREORDER-TREE-WALK (left [x])
4. PREORDER-TREE-WALK (right [x])

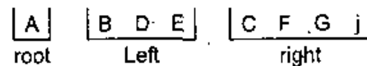
It also takes θ (n) time.

Example:



A binary tree

The preorder traversal of binary tree is



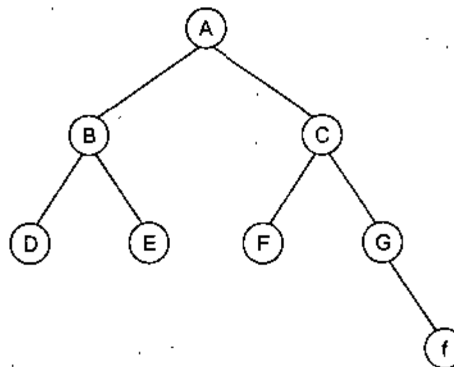
NOTES

POSTORDER-TREE-WALK (x)

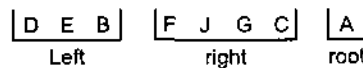
1. if $x \neq$ NIL
2. then POSTORDER-TREE-WALK (left [x])
3. POSTORDER-TREE-WALK (right [x])
4. print key [x]

The running time of POSTORDER-TREE-WALK is θ (n).

Example:



The POSTORDER traversal of binary tree is.



2.8 QUERYING A BINARY SEARCH TREE

SEARCH operation is a common operation which are performed on a binary search tree for a key sorted in the tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR.

Searching

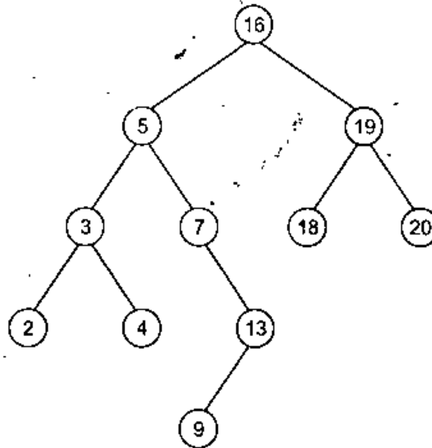
We use the following procedure to search for a node with a given key in a binary search tree

TREE-SEARCH (x, k)

1. if x = NIL or k = key [x]
2. then return x
3. if k < key [x]
4. then return TREE-SEARCH (left [x], k)
5. else return TREE-SEARCH (right [x], k)

NOTES

Example. *To search for the key 13 in the following tree:*



The procedure begins its search at the root and trace a path downward in the tree. We follow the path 16 → 5 → 7 → 13 from the root. We compares the key k with key (x). If the two keys are equal, the search terminates. If k is smaller than keys, the search continues in the left subtree of x_1. If k is larger than key $[x]$, the search continues in the right subtree. In the above example, we compare 13 with 16 *i.e.*, 13 < 16, then search continues in the left subtree of 6, continue this process until we found the key 13.

The running time of TREE-SEARCH is O (h), where h is the height of the tree.

The same procedure can be written iteratively by unrolling the recursion into a while loop.

ITERATIVE-TREE-SEARCH (x, k)

1. while $x \neq$ NIL and $k \neq$ key $[x]$
2. do if $k <$ key $[x]$
3. then $x \leftarrow$ left $[x]$
4. else $x \leftarrow$ right $[x]$
5. return x.

Minimum and Maximum

An element in a binary search tree whose key is a minimum can always be found by following left child pointers from the root until a NIL is encountered.

TREE-MINIMUM (x)

1. While left $[x] \neq$ NIL
2. do $x \leftarrow$ left $[x]$
3. return x.

The pseudocode for TREE-MAXIMUM is symmetric.

TREE-MAXIMUM (x)

1. While right $[x] \neq$ NIL
2. do $x \leftarrow$ right $[x]$
3. return x.

The running time of TREE-MINIMUM and TREE-MAXIMUM takes O (h) time on a tree of height h.

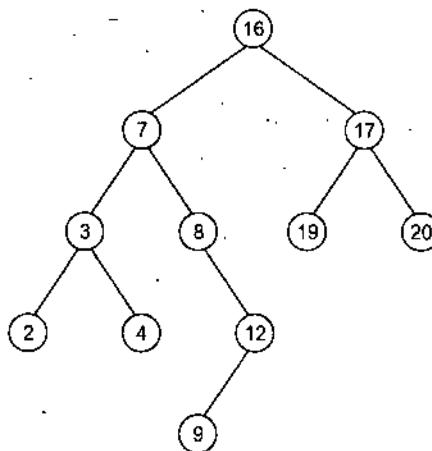
Successor and Predecessor

Suppose a node is given in a binary search tree, since it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than key $[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

```
TREE-SUCCESSOR (x)
 1. if right [x] ≠ NIL
 2. then return TREE-MINIMUM (right [x])
 3. y ← p [x]
 4. while y ≠ NIL and x = right [y]
 5. do x ← y
 6. y ← p [y]
 7. return y.
```

In the above procedure, TREE-SUCCESSOR is divided into two cases. If the right subtree of node x is non-empty, then the successor of x is just the leftmost node in the right subtree, which is found in line 2 by calling TREE-MINIMUM (right $[x]$).

Example:



Binary Search Tree

In the above Binary Search Tree, the successor of the node with key 16 is the node with key 19. On the other hand, if the right subtree of node is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x.

The running time of TREE-SUCCESSOR on a tree of height h is O (h). The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in O (h) time.

2.9 INSERTION AND DELETION

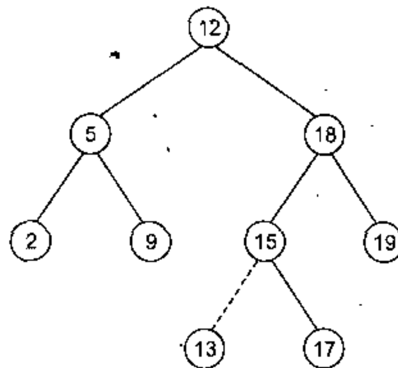
NOTES

Insertion: If you want to insert a new key or value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which key $[z] = v$, left $[z]$ = NIL and right $[z]$ = NIL. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.

TREE-INSERT (T, z)

1. $y \leftarrow$ NIL
2. $x \leftarrow$ root [T]
3. while $x \neq$ NIL
4. do $y \leftarrow x$
5. if key $[z]$ < key $[x]$
6. then $x \leftarrow$ left $[x]$
7. else $x \leftarrow$ right $[x]$
8. p $[z] \leftarrow y$
9. if y = NIL
10. then root [T] $\leftarrow z$
11. else if key $[z]$ < key $[y]$
12. then left $[y] \leftarrow z$
13. else right $[y] \leftarrow z$

For example: Let us consider we insert an item with key 13 into a binary search tree.



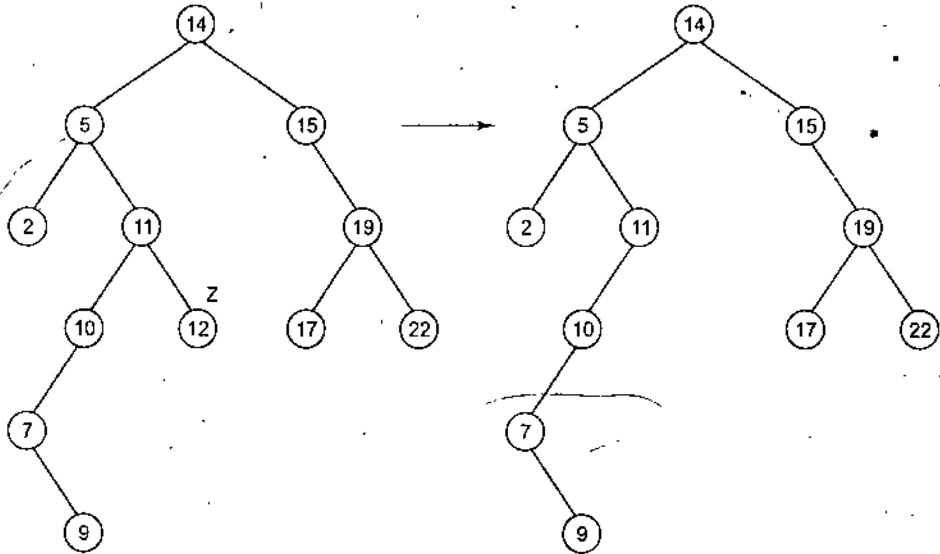
The dashed line indicates the link in the tree that is added to insert the item. TREE-INSERT works just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. While loop causes these two pointers to move down the tree, going left or right depending on the comparison of key $[z]$ with key $[x]$, until x is set to NIL. This NIL occupies the position where we wish to place the input item z. line 8-13 set the pointers that cause z to be inserted. The procedure TREE-INSERT takes O (h) time on a tree of height h.

Deletion: There are three cases to deleting a given node z from a binary search tree.

Case Ist: Suppose we want to delete a node z, which has no children, we just remove it.

For example,

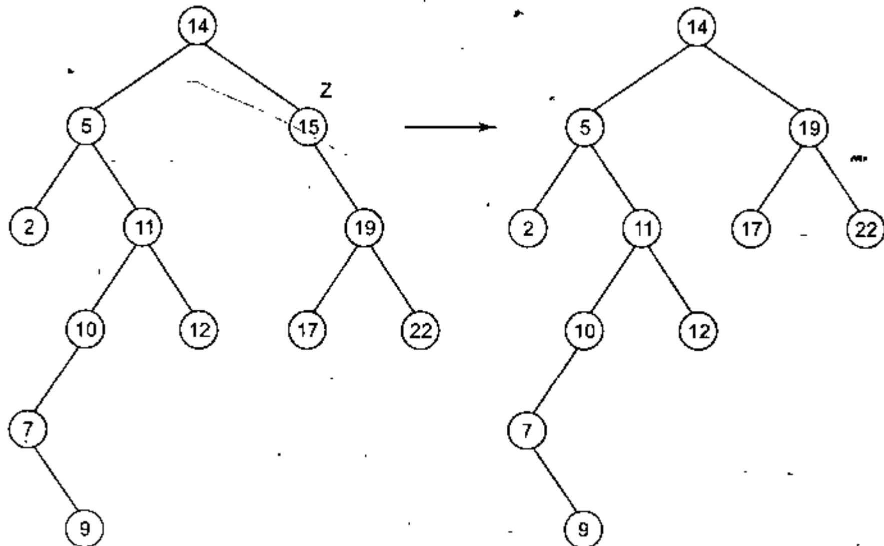
NOTES



In the above figure, 13 is a node which has no left or right children. So we can simply remove this node.

Case IInd: Suppose we want to delete a node z which has only one child, it may be left or right child. Then we splice out z.

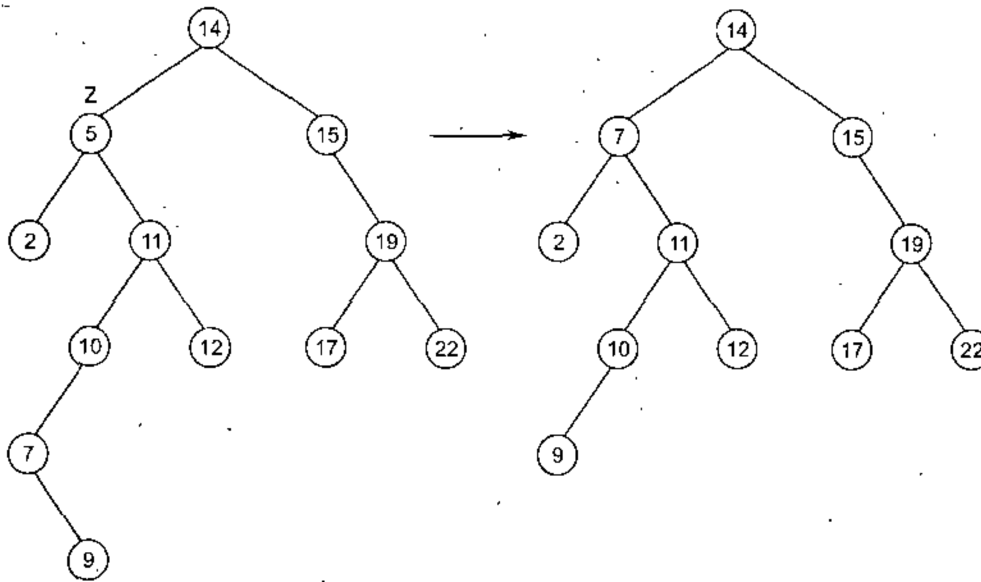
For example: In the following figure, 15 is a node which has only one child, *i.e.*, 19.



Case IIIrd: Suppose we want to delete a node z which has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

For example: In the following figure, 5 is a node which has two children 2 and 11. Then 5 is replace by 7 and we remove 5.

NOTES



The following code for TREE-DELETE organizes these three cases a little differently.

```
TREE-DELETE (T, z)
 1. if left [z] = NIL or right [z] = NIL
 2. then y ← z
 3. else y ← TREE-SUCCESSOR (z)
 4. if left [y] ≠ NIL
 5. then x ← left [y]
 6. else x ← right [y]
 7. if x ≠ NIL
 8. then P [x] ← P [y]
 9. if P [ y] = NIL
10. then root [T] ← x
11. else if y = left [P [y]]
12. then left [P [y]] ← x
13. else right [P [y]] ← x.
14. if y ≠ z
15. then key [z] ← key [y]
16. copy y's satellite data into z.
17. return y.
```

The procedure of TREE-DELETE runs in O (h) time on a tree of height h.

PART II: ADVANCED DATA STRUCTURE

2.10 RED-BLACK TREE

Red-black tree is a binary search tree with one extra bit of storage per node *i.e.*, colour which can be either red or black.

NOTES

This tree is approximately balanced tree.

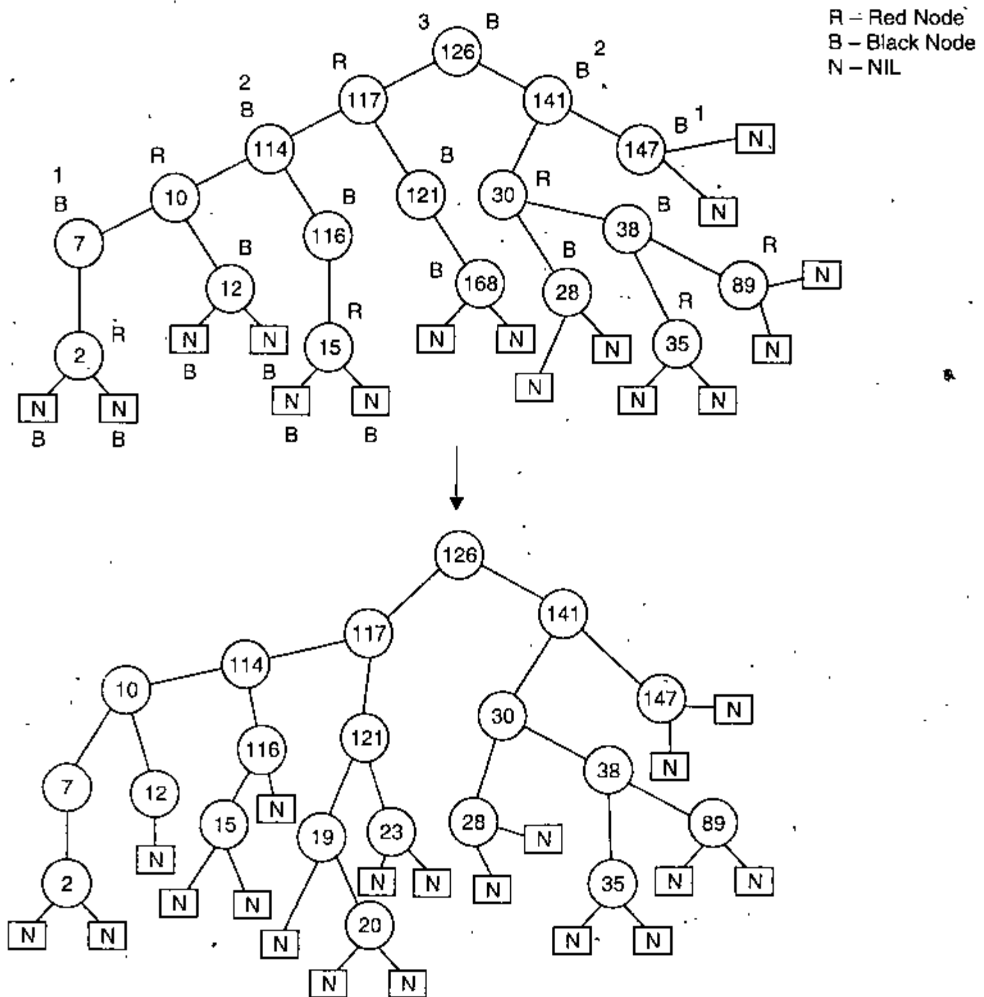
Each node of tree contains fields colours, key, left, right and P.

If notes contains the value nil. (NIL) we shall regards the NIL as an external node.

A binary search tree is red-black tree if it satisfy the following property they are:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all path from the node to descendant leaves contains the same number of black nodes.

Example:



Theorem. A *Red-Black tree with n internal nodes has height at most 2 log (n + 1).*

Proof. Now we show sub tree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

For root $bh(x) = 0$ (*i.e.*, no sub tree)

$2^{\circ} - 1 = 0$ (no internal nodes)

From induction we consider x has the height and has internal nodes with 2 children.

Each child has a black-height $bh(x)$ or $bh(x) - 1$ depending on colour is red or black.

Since height of child of x is less than the height it self so from *induction hypothesis* we conclude each child has $2^{bh(x)-1} - 1$ internal nodes.

So sub tree rooted at x, contains atleast

$$(2^{bh(x)-1} - 1)\,(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1 \text{ Internal nodes}$$

and h be height of tree. According to property four the black height of root must be $h/2$.

So

$$n \geq 2^{h/2} - 1 \quad \text{and} \quad (n + 1) \geq 2^{h/2}.$$

taking log.

$$\log_2 (n + 1) \geq h/2 \quad \Rightarrow \quad \boxed{h \leq 2 \log_2 (n + 1)}$$

Insertion of Red-Black Tree

Insertion is a process by which we push or insert an element in to red-black tree and n node in red-black tree can be inserted in to $O(\log_2 n)$ time.

When R-B tree is blank then element inserted in to root label and its left and right child and parent are nil.

Algorithm of Red-black Tree Insertion

Step 1. Red-black (A, n)

- $i \leftarrow$ nil [A]
- $i \leftarrow$ root [A]
- while $j \neq$ nil [A]
- do

  $j \leftarrow i$
- if temp [n] < temp [j]
- then $j \leftarrow$ left [j]
- else $j \leftarrow$ right [j]
- K [n] $\leftarrow i$

Step 2

- if i = nil [A]
- then root [A] $\leftarrow$ K
- else if temp [A] $\leftarrow$ K
- then left [i] $\leftarrow$ K
- else right [i] $\leftarrow$ K
- left [K] $\leftarrow$ nil [A]

- right [K] ← nil [A]
- colour [K] ← Red

NOTES

Step 3

- Red-black – fix (A, n)

Step 1

Red-black fix (A, n)

- while colour [K(n)] = Red
- do if K[n] = left [K[K[n]]
- then i ← right [K[K[n]]]

Step 2

- if colour [i] = Red
- then colour [K(n)] = Black
- colour [i] ← Black
- colour [K[K[n]]] ← Red
- n ← K[K[n]]
- else if n = right ← K[n]
- left-rotation (A, n)
- colour [K[n]] ← Black
- colour [K[K[n]]] ← Red
- Right rotation (A, K[K[n]])
- else (otherwise as then statement with right, and left interchange)

Step 3

- colour [root [A]] ← Black
  // colour of root node always black

Deletion of Red-Black Tree Node

Deletion are process where remove one or more node of R-B tree removed from tree and after that node are re-arranged using left-right rotation of Red-Black tree algorithm it happens due to that when we remove any node of R-B tree it may rotate black height of an Red-Black tree and deletion can be take place any where of R-B tree.

- At root label
- At node label
- Any intermediate node label.

Algorithm for R-B tree node deletion

Red-Black-Delete (A, n)

Step 1

- if left (n) = nil [A] or right [n] = nil [A]
- then i ← n
- else i ← tree-continue (n)
- if left [i] ≠ nil [A]

NOTES

- then $j \leftarrow$ left [i]
- else $j \leftarrow$ right [i]

Step 2

- K[j] $\leftarrow$ K[i] //Assigning value of K[i] into K[j]
- if K[i] = nil [A]
- then root [A] $\leftarrow i$
- else if j = left [K[i]]
- then left (K[i]) $\leftarrow j$
- else right [K[i]] $\leftarrow j$
- if $i \neq n$
- then temp [n] $\leftarrow$ temp [i]
- copy successor data in to n
- if colour [i] = Black
- then Red-Black-Delete-Funct (A, j)

Step 3

- Return i

Red-Black-Delete-Funct (A, j)

Step 1

- if ($j \neq$ root (A) && colour [j] = Black)
- if (j = left (K[j]))
- then $m \leftarrow$ right ([K[x]])
- if colour (m) = Red
- then colour [m] $\leftarrow$ Black
- colour K[j] $\leftarrow$ Red
- left-rotation (A, K[j])
- $m \leftarrow$ right (A, K[j])
- if (colour [left [m]]) = Black && right [m] = Black)
- then colour [m] $\leftarrow$ Red

Step 2

- $j \leftarrow$ K[j]
- else if colour [right [m]] $\leftarrow$ Black
- then colour [left [m]] $\leftarrow$ Black
- colour [m] $\leftarrow$ Red
- right-rotation (A, m)
- $m \leftarrow$ right [K[j]], colour [K[j]] $\leftarrow$ Black
- colour [K[j]] $\leftarrow$ Black
- Left-rotation (A, K[j])
- $j \leftarrow$ root [A]
- else (otherwise remain same with left & right interchange)

Step 3

- colour [j] $\leftarrow$ Black

2.11 AUGMENTING DATA STRUCTURE

NOTES

- Introduction
- Dynamic order statistics
- Retriving an element with a given rank.
- Determining rank of an element
- How augment data structure
- Augment Red-Black trees.

Augmenting Data Type

Generally we do not require any more data from existing data structure. For example stack, queue, link-list (Linear, Doubly, Circular), hash table etc., if we create new data structure it requires more effort for new data structure so use can augment the existing data structure by adding some additional information.

At rare case we need to create entirely new type of data structure. It implies that if we entirely change the data structure, it result new text book of data structure.

Dynamic Order Statistics

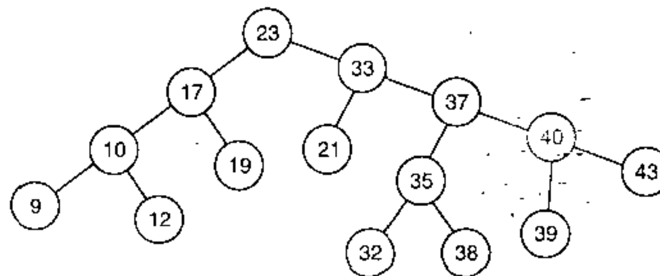
Now in this section we will check the support and result of data structure in general order statistics operation by the help of order statics tree (T).

A data structure that can support fast order statistic operation. A order statistics tree (T) is simply Red-Black tree with an additional information of parent [n], left [n], right [n] *i.e.*, information with parent and neighbour in node n. Who is left child of node n and who is right child of node n with Red or Black colour, and key value of that node.

If we want any changes with current data structure and want to perform some operation like addition or deletion of a node value in given set S. Then time complexity for that node = O (log n).

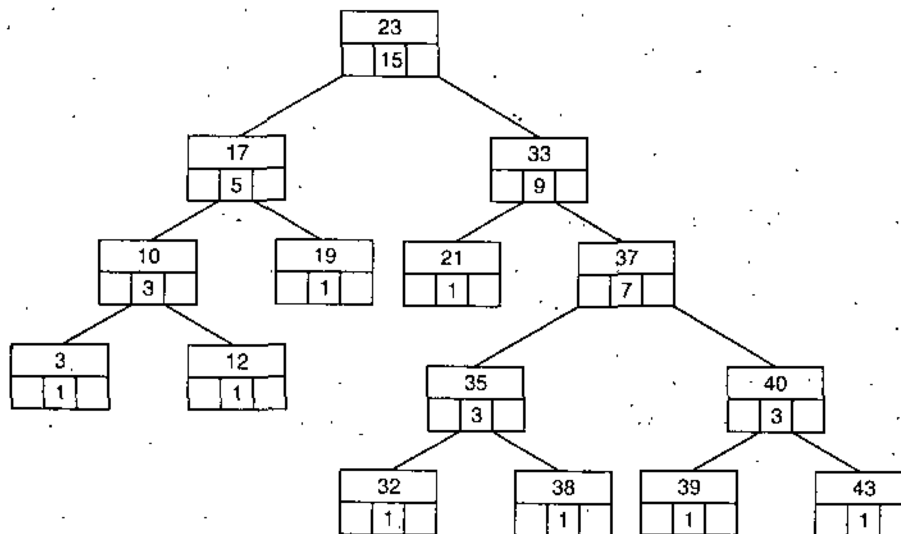
If number a given sub tree contains some changes in their internal node then time taken for any operation O (log n).

Example of sub tree



then for above sub tree identity will be

$$\text{size}[n] = \text{size}[\text{left}[n]] + \text{size}[\text{right}[n]] + 1$$



NOTES

Retriving an Element with a given Rank

In this section we will discuss the implementation of two-order statistic qurry, and analysis the size information during insertion and deletion of node. Here we use a term called rank, rank of the element as the position of removal node which can be resulted during in order traversal the worst case run time of retriving an element is 0 (log n).

Algorithm of Retriving an Element

order-statistic-rank (T, n)

// where T is traversal tree

// n is node which we remove.

Step 1

$m \leftarrow$ size [left [n]] + 1

K $\leftarrow n$

if ((K $\neq$ root [T]) && (K = right P [K]))

// P [K] Parent of K.

Step 2

then $m \leftarrow m$ + size [left [P [K]]] + 1

K $\leftarrow$ P [K]

Step 3

return (K),

order-statics-choose (n, K)

Step 1

$m \leftarrow$ size [left [n]] + 1

if (K = m)

then return (n)

Step 2

else if (K < m)

then return order-statistics (left [n], K)

NOTES

else return (order-statistics (right [n], $i - m$))

How to Augment a Data Structure ?

Augment process support for additional functionality of data structure which frequency used for algorithm designing following four steps involve in augmenting of data structure they are :

1. We select an underlying data structure
2. We resolved the property, which will be added to improve the operation functionality.
3. After adding the additional property, we verify the additional property functional operability in existing data structure.
4. Developing in operation of data structure.

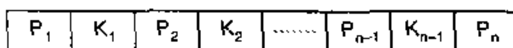
Augment Red-Black Tree

We can augment Red-Black tree (T) of data structure and if we add some additional operation then we can improve the operational functionality of addition and deletion function.

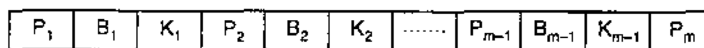
Note. Let F be a field that augment a Red-Black tree (T) of m nodes, and suppose that the contents of f for a node n car be computed using only the information in node n, left [n], right [n], including f [left [n]] and f [right [n]]. Then, we can maintain the value of f in all nodes of T during insertion and deletion without asymptotically affecting the O (log n) performance of these operations.

B⁺-Tree

- B-tree indices are similar to B⁺-tree indices. B-tree eliminates the redudant storage of search-key values.
- B-tree allows search key values to appear only once. A B-tree that represents the same search key as the B⁺-tree.
- Since search keys are not repeated in the B-tree. We may be able to store the index in fewer tree nodes than in the corresponding B⁺-tree index.
- Since search keys that appear in non-leaf needs appear no where else in the B-tree.
- A generalized B-tree leaf node appear in figure given below and non-leaf node appear.



Leaf Node



Non-Leaf Node

- Leaf node are similar to B⁺-tree leaf node.

B^+-Tree

B-tree has additional constraints that ensure that tree is balanced or not maintain operation like insertion deletion throughout.

Addition and deletion algorithm become more complex.

NOTES

When *B-tree of order P* used structure for searching following constraints or rule considerable they are :

1. Each internal node in B-tree is of form

$$< P_1, < K_1, P_{r_1} >, P_2, < K_2, P_{r_2} > \ldots\ldots < K_{q-1}, P_{r_{q-1}} >, P_q >$$

where $q \le P$

P_i is tree pointer, P_{ri}-is data pointer, K_i search value field.

2. With in each node $K_1 < K_2 \ldots\ldots < K_{q-1}$.
3. For all search key field value X in the sub tree pointed at by P_i (*i.e.*, its sub tree) we have

$$K_{i-1} < X < K_i \quad \text{for} \quad K_i < q$$

$$X < K_i \quad \text{for} \quad i = 1 \text{ ; and } K_i - 1 X \text{ for } i = q.$$

4. Each node has at most P sub tree.
5. Each node, Except root and leaf node has atleast [P/2] tree pointers the root node has atleast two tree pointer unless it is only node on tree.
6. A node with q tree pointers $q \le P$ has $(q - 1)$ search key field value (and $q - 1$ data pointers).
7. All leaf node will be at some level leaf node have same structure as internal nodes except that all of their tree pointers P_i are null.

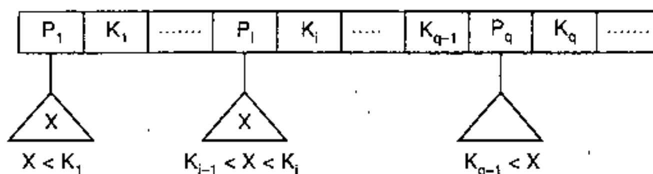
Where P : is a pointer to a child node and K : is the search value.

Two constraints must be hold at all times on search tree

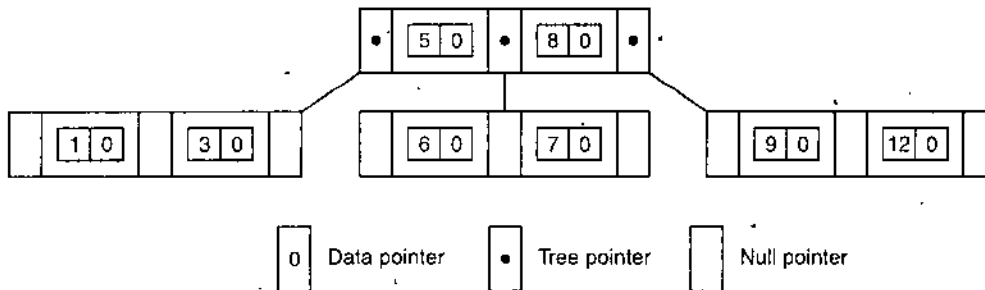
1. Within each node $K_1 < K_2 \ldots\ldots < K_{q-1}$
2. For all value of X in the sub tree pointed at by P_i. We have $K_{i-1} < X < K_i$ for $1 < i < q$

$\& X < K_i$ for $i = 1$ & $K_{i-1} < X$ for $i = q$

Example :



Note. Tree formate on B-tree are:



2.12 BINOMIAL HEAP

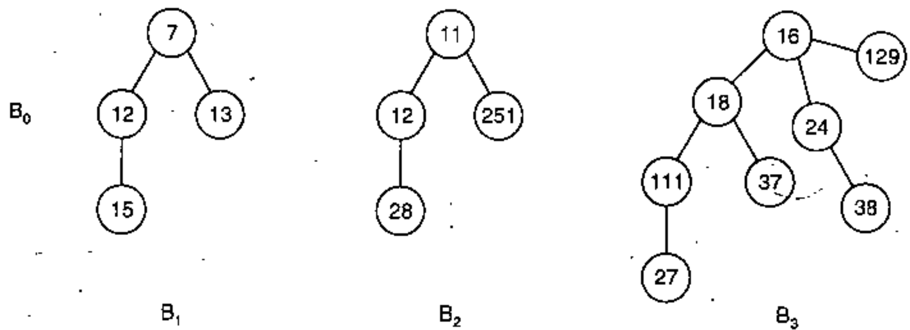
NOTES

Binomial heap is a set of Binomial trees that satisfies the following Binomial heap properties : -

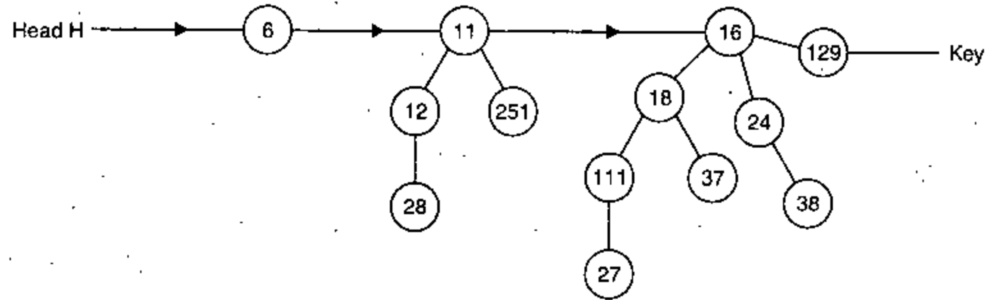
1. Each binomial tree in H obeys the min-heap property the key of a node is greater than or equal to then key of its parent. We say that each such tree is *min-heap ordered.*
2. For any non-negative integer K there is of most one Binomial tree contains the smallest key in the tree.

B_3	B_2	B_1	B_0
1	1	0	1

Example. Show Binomial heap H with B nodes Binary representation of 13 is 1101



B-Nodes B-H



Binomial heap and Fibonacci heap are known as mergeable heap. Which support the following operation.

1. **Make-Heap ( ).** Creates and returns new heap containing no elements.
2. **Insert (H, x).** Insert node x, whose key field has already been filled in, into heap H.
3. **Minimum (x).** Returns a pointer a pointer to the node in heap H. Whose key is minimum.
4. **Extract-min (H).** Deletes the node from heap H. Whose key is minimum.
5. **Union (H_1, H_2).** Creates and returns a new heap, that contains all the node of heaps H_1 and H_2. Heaps H_1 and H_2 'destroyed' by this operation.

6. **Decrease-key (H, x, K).** Assigns to node x with in heap H the new key value K. Which is assign or assumed to be no greater than its current key value.
7. **Delete (H, x).** Delete node x from heap H.

Why we use Binomial heap or fibonacci heap in compression of heap (Binary Heap) ?

Procedure	*Binary heap (worst case)*	*Binomial heap (worst case)*	*Fibonacci heap*
Make-Heap	θ (1)	θ (1)	θ (1)
Insert	θ (log n)	0 (log n)	θ (1)
Minimum	θ (1)	0 (log n)	0 (1)
Xtract-Min	θ (log n)	θ (log n)	0 (log n)
Union	θ (n)	0 (log n)	θ (1)
Decrease-Key	θ (log n)	θ (log n)	θ (1)
Delete	θ (log n)	θ (log n)	0 (log n)

Binomial Heap. Binomial heap is collection of Binomial trees and proving some key properties.

Binomial Tree. The Binomial tree B_k (k is no. of elements) is an ordered tree.

B_0 consist of single node.

The Binomial tree B_k consist of 2 B_{k-1} Binomial tree that are linked together.

B-tree

B-tree are special case of well known tree data structure. A tree is formed of nodes and each node of tree except for root have one parent node and several (zero or more) child nodes.

Root has no parent.

A node that has no any child node is called leaf node.

A non leaf node is called internal node.

The level of a node is always one more than the level of its parents.

Level of root node be zero.

A sub tree of node consist of that node and all its descendant nodes *i.e.,* child node.

Search Tree and B-tree

Search tree is different from multi level index.

A search tree of order P is a tree s.t. each node consist at most (P – 1) search tree and P pointers in order

$$< P_1, K_1, P_2, K_2, \ldots\ldots P_{q-1}, K_{q-1}, P_q >$$
$$\text{where } q \leq P$$

Properties of B-tree

1. *There are 2^k nodes*

Proof. Binomial tree B_k consist of two copy of B_{k-1} so B_k has

$$2^{k-1} + 2^{k-1} = 2^k \text{ nodes}$$

2. *Height of tree is k.*

NOTES

Proof. Because of the way in which the two copy of B_{k-1} are linked to form B_k, the maximum depth of a node in B_k is one greater than the maximum depth in B_{k-1}, By inductive Hypothesis the maximum depth is

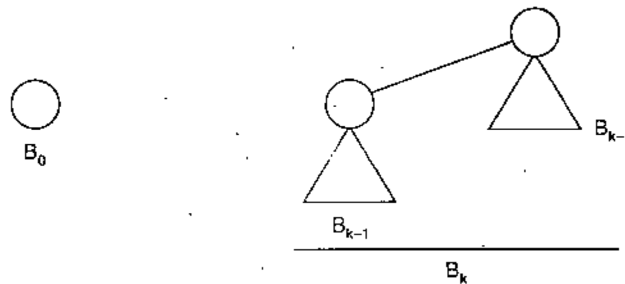
$$(k - 1) + 1 = k.$$

3. *There are exactly i^{th} nodes at depth $i = 0, 1, \ldots\ldots k$*

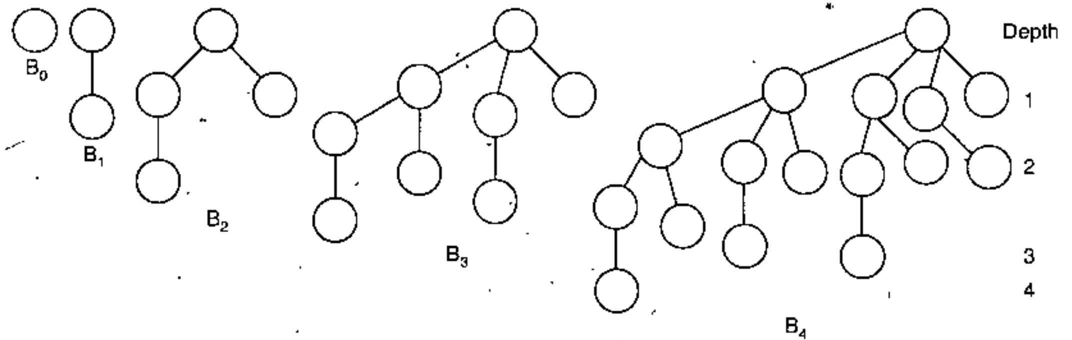
Proof. Let D (k, i) be the number of nodes at depth i of Binomial tree B_k.

Since B_k is composed of two copy of B_{k-1} linked together, a node at depth in B_{k-1} appears in B_k once at depth i and once at depth $(i + 1)$.

Example 1:



Example 2:



So number of nodes at depth $i - 1$ in B_{k-1}

$$D(k, i) = D(k - 1, i) \rightarrow D(k - 1, i - 1)$$

$$= \binom{k-1}{i} + \binom{k-1}{i-1}$$

$$= \binom{k}{i}$$

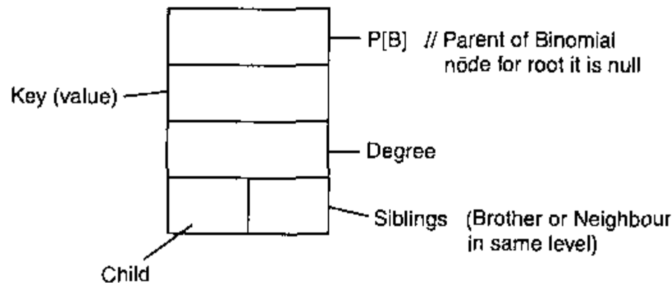
4. *The root has degree k, which is greater than that of any other node, more over if the children of the root are numbered from left to right by $k - 1, k - 2, \ldots\ldots 0$ child i is the root of a sub tree B_i.*

Operation of Binomial-Heap

1. Creating a new Binomial heap
2. Inserting a node in Binomial heap
3. Extracting the node with minimum key
4. Finding minimum key

5. Decreasing a key value
6. Union of two Binomial heap.

1. Creating a new Binomial heap. To create an empty binomial heap, we call create- Binomial heap ( ) function, and allocate and return an object to Binomial heap B. Where head of Binomial heap [B] = NIL and running time to create Binomial heap (empty) θ (1).



2. Inserting a node in Binomial heap. Insertion procedure push or insert a node with data field in Binomial heap and we assume an node is already inserted before inserting a new node in B-heap.

Binomial-Heap-Insertion (B, *n*)

Step 1

// B′ ← Make a new Binomial heap (B′)

- B′ ← Binomial-Heap ( )
- P[*n*] ← NIL // Parent of node of now B-heap is null
- Child [*n*] ← NIL
- Sibling [– *n*] ← NIL // No-sibling
- degree [*n*] ← 0 // Due to no child
- Lead [B′] ← *n*
- B ← Binomial-Heap-union (B, B′)

// After making a new Binomial heap we merge it with old our Binomial heap which is already exist.

3. Extracting the node with Minimum key. To find out minimum key at Binomial heap at given Binomial heap (Min-Binomial-Heap) we search the minimum value at root level of each Binomial tree due to that each Binomial tree is min-Binomial tree and minimum value will at root level so we search only at sibling of each and every root and after getting minimum key we extract that minimum key value and then we make that Binomial tree min-Binomial tree by placing minimum value of that Binomial tree at root level of that tree hence now Binomial tree obtained after extracting min key value.

Binomial-Heap-Min-Extract (B, *n*)

Step 1. Find min value at root level of each Binomial tree and remove it.

Step 2. B′ ← Make-Binomial-Heap ( )

Step 3. Reverse the order of the link list of *n*'s children and set head [B′] to point of the head of obtain list.

Step 4. B ← Binomial-Heap-Union (B, B′)

Step 5. return *n*

NOTES

4. Finding Minimum key. At Binomial heap minimum key will be at root level of Binomial tree. Since they are min-Binomial tree and minimum value will be at root level of Binomial tree or its sibling.

Binomial-Heap-min (B)

Step 1. $i \leftarrow$ NIL // initilization

Step 2. $j \leftarrow$ head [B]

Step 3. min $\leftarrow \infty$ // Assume min value is ∞ and then compare it at root level of each B-tree.

Step 4. While $n \neq$ NIL

Step 5. do if key [n] < min

Step 6. Then min $\leftarrow$ key [n]

Step 7. $i \leftarrow j$

Step 8. $j \leftarrow$ sibling [n]

Step 9. Return i

5. Decreasing a Key value. In this operation we select the node which value we went to decrease and then compare the key value of that node from new value if new value < present key value then we remove the value (key value) of that node and insert new value. If key value < new value then error occures due to that inserting value is greater than present value.

Binomial-Heap-Dec-Key (B, n, m)

// Where B is Binomial Heap

// n is old key value

// m is new key value

Step 1. If m > key [n]

Step 2. Then print "inserting key value is greater than present key value"

Step 3. Key [n] $\leftarrow m$

Step 4. $i \leftarrow n$

Step 5. $j \leftarrow$ K[i]

Step 6. if ($j \neq$ NIL && key [i] < key [j])

Step 7. Then exchange key [i] $\leftrightarrow$ Key [j]

Step 8. $i \leftarrow j$

Step 9. $j \leftarrow$ K[i]

6. Union of two Binomial heap. Union function is join function when we call this function then 2 Binomial heap merge and make a single Binomial heap and adjust the element of Binomial tree, and make min-Binomial tree and from there combination a Binomial heap yield.

Binomial-Heap-union (B_1, B_2)

Step 1

- B $\leftarrow$ make-Binomial-Heap (B)
- head [B] $\leftarrow$ Binomial-Heap merge (B_1, B_2)
- if head [B] = NIL
- then return (B)

NOTES

Step 2

- Prev-n ← NIL
- n ← NIL
- n ← head [B]
- next-n ← Sibling [B]
- if (next-n ≠ NIL)
- if (degree [n] ≠ degree [next-n] or sibling [next-n ≠ NIL && degree [sibling [next-n]] = degree [n]
- then prev-n ← n
- n ← next-n
- else if key [n] ≤ Key [next-n]
- then sibling [n] ← sibling [next-n]

Step 3

- temp (next-n, n)
- else if prev-n = NIL
- then head [B] ← next-n
- else sibling [prev-n] ← next-n
- temp (n, next-n)
- n ← next-n
- next-n ← sibling [n]

Step 4

return [B]

temp (n, m)

K[n] ← m

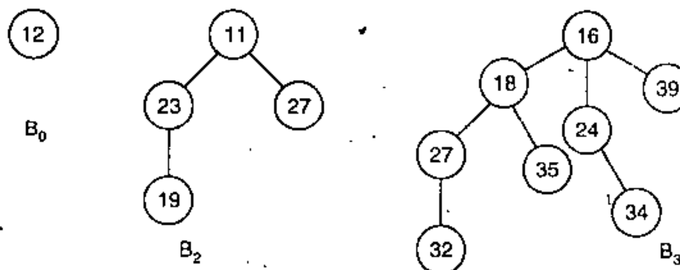
Sibling [n] ← child [m]

child [m] ← n

degree [n] ← degree [m] + 1

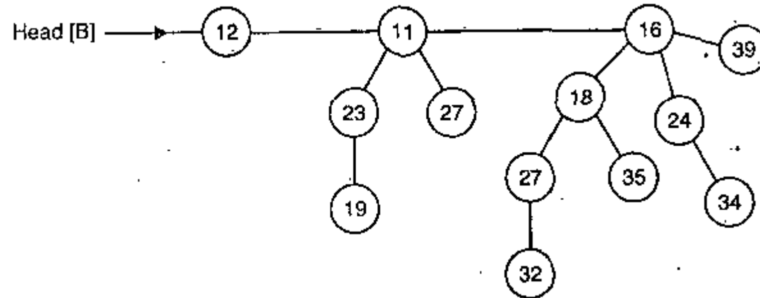
Example:

The Binomial tree are



NOTES

Then Binomial heap is

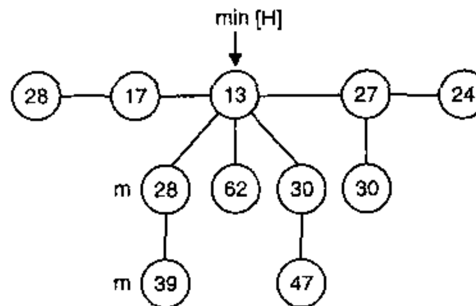


Binomial head default set at B_0 level and Fibonacci heap head at default set at *min value of root* level of Fibonacci heap.

Structure of Fibonacci Heap

It like a Binomial heap, a Fibonacci heap is a collection of min heap ordered tree.

Example:



Mark [x] means if a node losses or delete its child than if marked.

The Boolean value field mark [x] indicates whether a node x has lost its child since last time x was made the child of another node.

Newly created child are unmarked and a node x becomes unmarked whenever it is made of child of another node.

Potential Function of Fibonacci Heap

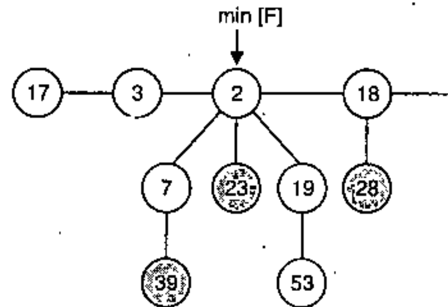
Potential function of Fibonacci heap is sum of the potentials of its constituent Fibonacci heap

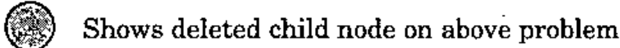
i.e., it is sum of individual potential function of Fibonacci heap. It is sum of total marked and unmarked node of Fibonacci heap potential function can be calculated as

$$\phi(F) = t(F) + 2m(F)$$

t (F) ≡ *no's of node at root level of* F-heap

m (F) ≡ No's of marked node



 Shows deleted child node on above problem

$\phi(F) = 4 + 2 * 3$

$\phi(F) = 10$

where total no's of node at root level

{17, 3, 2, 18} = 4

total marked node {23, 28, 39} = 3

* Marked node are those node which looses there child *i.e.*, child has been deleted of that node, so we marked those node.

NOTES

Operations Performs in Fibonacci Heap

1. Create an empty Fibonacci heap
2. Insertion of a node in Fibonacci heap
3. Extracting minimum key value from Fibonacci heap
4. Union of 2 Fibonacci heap
5. Decreasing a key value from Fibonacci heap
6. Delete a node from fibonacci heap.

1. Create an empty Fibonacci heap. To create an empty Fibonacci heap we just call the make-Fibonacci-Heap (F) proceed the creation of Fibonacci heap and where n [H] = 0 and min [H] = NIL. Which show what there are no any element.

2. Insertion of an node in Fibonacci heap. Insertion is a process by which we insert or push a node value in Fibonacci heap (F) and assume that heap already allocated a node with key [n] value.

Fibonacci Heap-insertion (F, n)

Step 1

- degree [n] ← 0
- p [n] ← NIL // Parent of that node is NIL
- child [n] ← NIL // Child of that node NIL
- Left [n] ← n // Self indicating
- right [n] ← n // Self indicating

Step 2

- mark [n] ← False // No child has been deleted
- for each and every node m of that root list of F.
- do n ← m

NOTES

5. Decreasing Key value of Fibonacci heap. We firstly search that node, which node key value is decreased and then compare with present key value and decreasing key value. If present key value [n] < inserting key value [m] then error occur *i.e.*, inserting value is greater than present value else case we remove the present value and insert new value and compare with its parent value if parent value > present value (after insertion) then value should be interchange.

Algorithm

Fibonacci-Heap-Decrease-Key (F, n, m)

// n present key value & m is new value

Step 1

- if m > Key [n]
- then print "new value is greater than present value"
- Key [n] $\leftarrow m$
- $i \leftarrow$ P[n]
- if $i \neq$ NIL and key [n] < key [i]

Step 2

- then temp (F, n, i)

Step 3

- temp function (F, i)
- if key [n] < key [min [F]]

Step 4

- then min [F] $\leftarrow n$

  temp (F, n, i)
- Remove n from child list of i
- add n in root list of F // we insert at root label
- P[n] $\leftarrow$ NIL // Parent of n is nil
- mark [n] $\leftarrow$ false

  // No child node has been deleted.

  temp-function (F, i)

Step 1

- $j \leftarrow$ P[i]
- if $j \neq$ NIL
- then if (mark [i] = False)

  // *i.e.*, child node has been not deleted of i

Step 2

- then mark [i] $\leftarrow$ true

  // Child of i has been deleted.
- else temp (F, i, j)

Step 3

- temp function (F, j)

6. Delete node from Fibonacci heap. In this operation we delete or remove a node from list. This process is also called pop operation after removing a node from heap we adjust the child node of its node.

NOTES

Fibonacci-Heap-Deletion (F, n)

Step 1

- if n = min [F]

Step 2

- then Fibonacci-Heap-temp (F)

Step 3

- else $i \leftarrow p$ [n]

Step 4

- if $i \neq$ NIL

Step 5

- then temp (F, n, i)

Step 6

- temp-function (F, i)

Step 7. then adjust child of n at root of F.

Step 8. delete n from root of F

Fibonacci-Heap-temp (F)

Step 1. $j \leftarrow$ min [F]

if $j \neq$ NIL

then child of n add to root of F

P[n] $\leftarrow$ NIL

delete n from Fibonacci heap F.

Step 2. if n = right [n]

then min [F] $\leftarrow$ NIL

else min [F] $\leftarrow$ right [n]

del-function (F)

n [F] $\leftarrow n$ [F] – 1

Step 3. return (F)

del-Function (F)

Step 1. For $i \leftarrow$ 0 to degree (n (F))

do K[i] $\leftarrow$ NIL

do

$j \leftarrow m$

$l \leftarrow$ degree [j]

while K [l] $\neq$ NIL

do a $\leftarrow$ K [l]

if key [j] > key [a]

then interchange $j \leftrightarrow a$

Fibonacci-Heap-join (F, a, j)

K [l] $\leftarrow$ NIL

$l \leftarrow l + 1$

NOTES

K [l] ← j

min [F] ← NIL

for i ← 0 to degree (n [F])

do if K[i] ≠ NIL

then K[i] be in root Fibonacci heap F.

if min [F] = Nil or key [K[i]] < key [min [F]]

then min [F] ← K[i]

2.13 DATA STRUCTURE FOR DISJOINT SET

Stack

Queue

Link list

Stack

Stack works in principal of FILO or LIFO (Last in first out). It is defined as a list in which insertion and deletion are performs only at top of the stack called (TOS).

Stack can be implemented by

- Array
- Pointer (link list)

And performs 2 main operation PUSH and POP, where PUSH is similar to insertion and POP is similar to deletion.

Push Operation

Push operation is insertion operation, in which we insert an element at top of stack (TOS) and if stack is full then we want to insert more element then "overflow" condition rises *i.e.*, no more element could be inserted.

Algorithm

Push (s, n)

Step 1. // Check for stack overflow

if TOS ≥ Size

Print "stack overflow"

Step 2. else

TOS ← TOS + 1

// increase pointer value by one

Step 3. S [TOS] ← Value

// insert value at Top

Step 4. return

POP Operation

Pop operation delete the topmost element of stack if stack is already empty and we want to remove further more element then condition occures with name "underflow stack" *i.e.*, no element in stack.

NOTES

Algorithm

Pop (S, n)

Step 1. if TOS = 0

Print "stack underlying underflow"

Step 2. Value ← S [TOS]

Tos ← Tos – 1

// delete the value at top level

Step 3. return

Queue

Queue is a linear data structure. Which work's on principal of 'FIFO' first in first out. Queue is used in operating system time sharing and it is used to system process scheduling like round robin technique.

Queue can implemented by 2 technique:

1. Using array
2. Using pointer (link-list).

Following 2 operation (main) performs in queue with the help of front and rear, front work like pop & rear as push operator.

Algorithm for Push

Step 1. if rear ≥ Size

spring "overflow queue"

// To check the condition of queue

Step 2. rear ← rear + 1

// Incrementing rear pointer

Step 3. Q [rear] ⇐ value

if front = 0

front = 1

Step 4. return

Algorithm for Pop Operation

Step 1. if front = 0

print "queue underflow

Step 2. value = Q [front]

// Remove an element

Step 3. if (front = rear)

front = 0

rear = 0

Step 4. else

front = front + 1

Step 5. return (value)

Link-List

Linked-List is collection of nodes each node have 2 part

1. Information
2. Pointer to next node

Using link-list following operation performs

1. Deletion of information
2. Insertion of information
3. Updating of information
4. Display of information.

Building Link-List

To create link-list following steps involved.

1. Declare the structure that defines the list entries.
2. Declare the variable *start* and * node or (* start and * node).
3. Assume start next = NULL to signifies an empty list.
4. *Find each list entry.*

(*i*) Find end of the list so that node → next = NULL.

(*ii*) Allocate memory for the new entry by assigning it to node → next.

(*iii*) Assign node the value of node → next.

(*iv*) Assign the member value to node.

(*v*) Assign node → next = NULL to indicate the end of the list.

Structure of Structure Node

```
Structure node
{
Data type information;
Struct node * next;
}
Start, * node;
```

2.14 SPLAY TREES

A splay tree is a self-balancing binary search tree with the additional unusual property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O (lg (*n*)) amortized time. For many non-uniform sequences of operations, splay tree perform better than search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Sleator and Robert Tarfan.

All normal operations on a binary search tree are combined with one basic operation, called splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a bottom-up algorithm can combine the search and the tree reorganization.

Advantages and Disadvantages

Good performance for a splay tree depends on the fact that it is self-balancing, and indeed self optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. This is an advantage for nearly all practical applications, and is particularly useful for implementing caches; however it is important

to note that for uniform access, a splay tree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree.

Splay tree also have the advantage of being considerably simpler to implement than other self-balancing binary search tree, such as red-black tree or AVL trees, while average-case performance is just as efficient. Also, splay trees don't need to store any book keeping data, thus minimizing memory requirements. However, these other data structures provide worst-case time guarantees, and can be more efficient in practice for uniform access. One worst case issue with the basic splay tree algorithm is that of sequentially accessing all the elements of the tree in the sorted order. This leaves the tree completely unbalanced (this takes n accesses-each a O (log n) operation). Reaccessing the first item triggers an operation that takes O (n) operations to rebalance the tree before returning the first them. This is a significant delay for that final operation, although the amortized performance over the entire sequence is actually O (lg n). However, recent research shows that randomly rebalancing the tree can avoid this unbalancing effect and give similar performance to the other self-balancing algorithms.

NOTES

It is possible to create a persistent version of splay trees which allows access to both the previous and new versions after an update. This requires amortized O (log n) space per update.

Contrary to other types of self-balancing trees, splay trees work well with nodes containing identical keys. Even with identical keys, performance remains amortized O (log n). All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operations can return the left most or right most node of a given key.

The Splay Operation

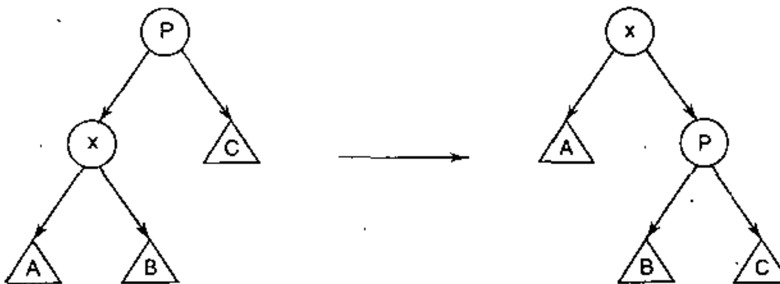
When a node is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step can depend on three factors:

1. Whether x is the left or right child of its parent node, P.
2. Whether P is the root or not, and if not.
3. Whether P is the left or right child of its parent, g (the grandparent of x).

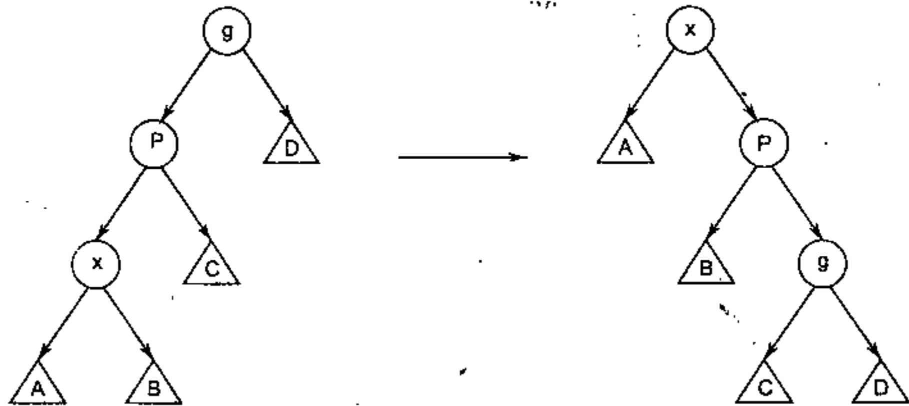
The three types of splay steps are:

1. **Zig Step:** This step is done when P is the root. The tree is rotated on the edge between x and P. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when x has odd depth at the beginning of the operation.

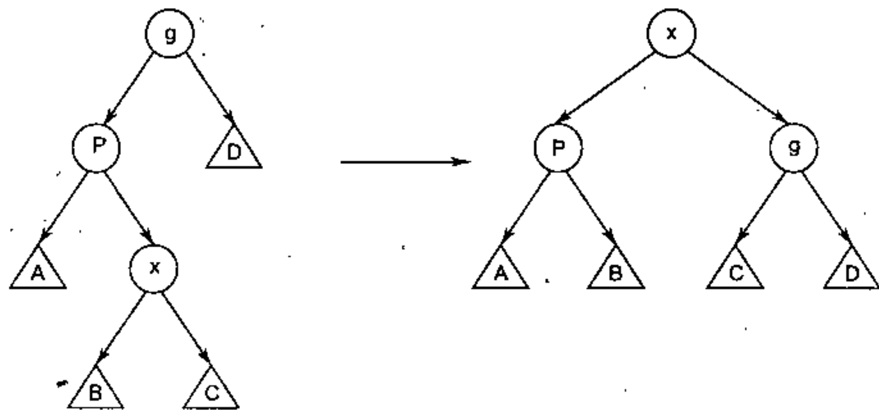


2. **Zig-zig Step:** This step is done when P is not the root and x and P are either both right children or are both left children. The picture below shows the

case where x and P are both left children. The tree is rotated on the edge joining P with its parent g, then rotate the edge joining x with P. Note that zig-zig steps are the only thing that differentiate splay trees from the rotate to root method introduced by Allen and Munro prior to the introduction of splay trees.



3. **Zig-zag Step:** This step is done when P is not the root and x is a left child and P is a right child or vice versa. The tree is rotated on the edge between x and P, then rotated on the edge between x and its new parent g.



2.15 PRIORITY QUEUE

A queue in which we can insert or delete (remove) items from any position depending on some priority is known as a **priority queue.** For example, in a multiuser system, the CPU is needed by many programs and it is utilised by the programs one at a time depending on some priority. The CPU is first used by the program having the highest priority. A priority queue can be splitted into several queues if needed. Figures shown below illustrate a priority queue:

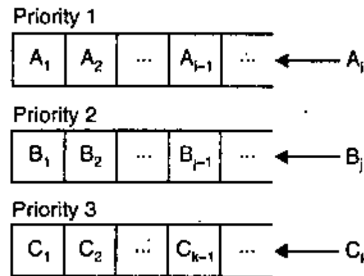
Task identification

A_1	A_2	...	A_{i-1}	B_1	B_2	...	B_{j-1}	C_1	C_2	...	C_{k-1}	...
1	1		1	2	2		2	3	3	...	3	...

Priority

A_i B_j C_k

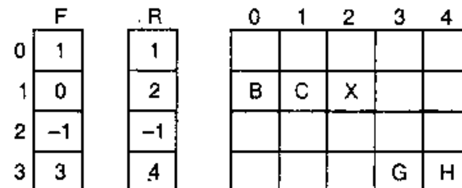
A priority queue as a single queue with insertions allowed at any position.



A priority queue viewed as a set of queues.

Priority Queue using Array

To maintain a priority queue in memory, we use multidimensional array, *i.e.*, use a separate queue for each level of priority (priority number). Each queue will appear as a separate circular array and have its own set of pair of pointers called *Front* and *Rear*. Assume that each queue is of same size. So, we need only a two-dimensional array in which number of rows is equal to number of priorities and the elements are added to the respective queue depending upon its priority number. Figure shown below illustrates a priority queue of size 5.

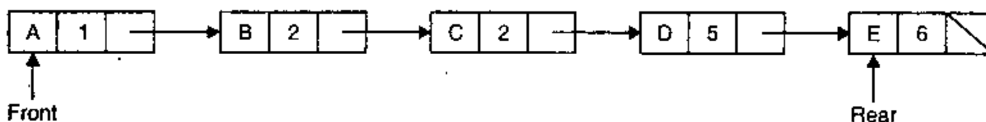


Priority queue using arrays.

In the above priority queue, F[*i*] and R[*i*] contain the front and rear elements of row *i* of queue, *i.e.*, the row maintains the queue of elements with a priority number *i*. Whenever we are performing the insertion operation, We have to read the data item along with priority number. Then it will search for that row for empty cell, if it is found then place the item in that cell and rear and front pointers are modified as in a circular queue. If it is full, then it indicates overflow condition for that row or priority number. To delete an item from the priority queue, first it will check for the element in first row, if it is not empty, simply it deletes the element like normal queue. If it is empty, then it check for next row, if it not empty, delete the element from that row otherwise it checks for next row. If all front and rear items are – 1, then it indicates underflow condition if we try to delete the element. So, the deletions are taken first from first row to last row.

Priority Queue using Singly Linked List

In a singly linked list, the data items are maintained according to the priority. Each node in this queue contains three fields. One is actual data item, the second field is the priority number of the data item and the third field is link to next item in the list. At any point of time the highest priority element is in the front of the list, so that directly we can delete the item from the front of the list. If the two data items have the same priority, then we can consider their entry sequence in FIFO order. Figure shown below illustrates the implementation of priority queue using linked list.



Priority queue.

STUDENT ACTIVITY

1. What is a circular queue? How do represent it?

2. Write short notes on hash function.

SUMMARY

NOTES

1. A stack is a non-primitive data structure. In a stack, the element deleted from the set is the one most recently inserted.
2. In Infix notation, where the operator is written in-between the operands.
3. In the prefix notation, a notation in which the operator is written before the operands, it is also called polish notation is the honor of the mathematician Jan Lukasiewicz who developed this notation.
4. The postfix notation, the operators are written after the operands, so it is called the postfix notation.
5. Queue means a line. A queue is logically a first in first out (FIFO) type of list. In a queue, new elements are added to the queue from one end called REAR end, and the elements are always removed from other end called the FRONT end.
6. Linked lists are special list of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element.
7. LIST-SEARCH (L, K) is a process to finds the first element with key K in list L by a simple linear search returning a pointer to this element.
8. A hash table is an effective data structure where we store a key value after applying the hash function it is arranged in the form of an array that is addressed via., a hash function.
9. The universal hasing is used to select the hash function at random from a carefully designed class of functions at the beginning of execution.
10. *B-tree* are special case of well known tree data structure. A tree is formed of nodes and each node of tree except for root have one parent node and several *(zero or more) child nodes.*

    Search tree different from multi level index.
11. *Stack* works in principal of FILO or LIFO (Last in first out). It is defined as a list in which insertion and deletion are performs only at top of the stack called (TOS).
12. Push operation is insertion operation, in which we insert an element at top of stack (TOS) and if stack is full then we want to insert more element then "overflow" condition rises *i.e.,* no more element could be inserted.
13. Pop operation delete the top most element of stack if stack is already empty and we want to remove further more element then condition occures with name "underflow stack". *i.e.,* no element in stack.
14. A splay tree is a self-balancing binary search tree with the additional unusual property that recently accessed elements are quick to access again.

GLOSSARY

- *Data structure:* It is representation of the logical relationship existing between individual elements of data.
- *Stack:* Stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called top of the stack.

NOTES

- *Queue:* A queue is an ordered list in which all insertions take place at one end, the rear, where as all deletion takes place at other end, the front. Therefore, 'Queues' work on the concept of First In First Out (FIFO) Principle.
- *Red-black tree:* It is a binary search tree with one extra bit of storage per node *i.e.,* colour which can be either red or black.
- *Binomial heap:* Binomial heap is collection of binomial trees and proving some key properties.

REVIEW QUESTIONS

1. Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.
2. Implement the dictionary operations INSERT, DELETE and SEARCH using simply linked circular lists: What are the running time of your procedure?
3. Write a C program to reverse a singly linked list.
4. How is the queue different from the stack?
5. Evaluate the following prefix expressions:
   (*i*) 9 10 17 * +
   (*ii*) 40 25 + 8 5 * 4 +
6. Transform the following to infix:
   (*i*) ABC ∧
   (*ii*) ABC * + DE | F *
7. Differentiate between prefix and post fix expressions.
8. Draw the 11-item hash table resulting from hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 and 5, using the hash function
   $$h(i) = (2i + 5) \bmod 11$$
9. What are hash tables? How do we make a good choice of hash function.
10. Draw the 11-item hash table resulting from hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 and 5, using the hash function
    $$h(i) = (2i + 5) \bmod 11$$
11. What are hash tables? How do we make a good choice of hash function.
12. Suppose that a binary search tree is constructed by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.
13. Give a recursive version of the TREE-INSERT procedure.
14. *Give an algorithm to count the number of leaf nodes in a binary tree t.* What is its computing time?
15. Write TREE-PREDECESSOR procedure.
16. Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.
17. For the set of keys {1, 4, 5, 10, 16, 17, 21}, draw binary search trees of height 2, 3, 4, 5 and 6.
18. Give recursive algorithm that perform preorder and postorder tree walks in $\theta(n)$ time on a tree of n nodes.

19. What is splay tree? Explain its advantage.
20. Insert items with the following (in order) into an initially empty splay tree and draw the tree after each operation:

    3, 5, 7, 12, 14, 16, 19, 23, 28.

21. What is a priority queue?

NOTES

FURTHER READINGS

- Hari Mohan Pandey, '*Design Analysis and Algorithm*', University Science Press.
- Gyanendra Kumar Dwivedi, '*Analysis and Design of Algorithm*', University Science Press.

UNIT

3

ADVANCED DESIGN AND ANALYSIS TECHNIQUES

STRUCTURE

3.0 Objectives
3.1 Dynamic Programming
3.2 Back Tracking
3.3 Greedy Method
3.4 Amortized Analysis
3.5 Branch and Bound
3.6 Applications
3.7 Knapsack Problem
3.8 Integer Programming
3.9 Quadratic Assignment Problem
3.10 Maximum Satisfiability Problem (Max-Sat)
- *Summary*
- *Glossary*
- *Review Questions*

3.0 OBJECTIVES

After going through this unit, you will be able to:

- describe the steps in developing a Dynamic Programming Algorithm
- discuss the terms used in Back Tracking
- define Greedy Algorithms
- explain Amortized Analysis
- describe Branch and Bound (BB).

3.1 DYNAMIC PROGRAMMING

This is the most powerful method of designing technique for optimization. This technique was developed by Richard Bellman. Dynamic programming algorithm stores the results for small subproblems and looks them up, rather than recomputing them, when it needs them later to solve larger sub problems.

"Dynamic programming is an algorithm for design method that can be used when solution can be viewed decision sequence of result".

Steps in Developing a Dynamic Programming Algorithm

1. Divide, sub problem
2. Table storage
3. Combine bottom-up computation

1. Divide, Sub Problem. In this topic problem is divided in several small problem's and each problem is called sub problem.

2. Table Storage. These sub problem result stored in table and used whenever is required.

- Principle of optimality
- Polynomial breakup
- Shortest path

Principle of Optimality

According to principle of optimality optimal sequence of decision has the property that whenever the initial state and decision are, the remaining must constitute an optimal decision must constitute an optimal decision sequence with regard to the state resulting from the first decision.

Polynomial Breakup

Given problem being divided into several problem for efficient performance of dynamic problem and total number of sub problem should be a polynomial number.

Shortest Path

Let G = (V, E) be a directed graph with n vertices.

Let C be a cost adjacency matrix for G such that $C(i, i) = 0$ for all i.

All pair shortest path problem is to determine a matrix A such that $B(i, j)$ is the length of a shortest path from i to j.

We assume that there is no cycle of negative length.

From i to k and k to j must be shortest paths from i to k and k to j.

Let us examine a shortest path from i to j in G.

This path originates at vertex i and goes through some intermediate vertices and terminates at vertex j.

Let k be an intermediate vertex on this shortest path then the subpath Let $Bk(i, j)$ represent the length of a shortest path from i to j going through no vertex of index greater than k,

A (i, j) represent shortest distance between i and j.

$$\mathbf{B(i, j) = \min\{\min_{1 \le k \le n}\{Bk-1\,(i, k) + Bk-1\,(k, j)\}, cost\,(i, j)\}}$$

Clearly

$$B0(i, j) = \text{cost}\,(i, j)$$

$$\mathbf{B(i, j) = \min\{Bk-1\,(i, j), Bk-1\,(i, k) + Bk-1\,(k, j)\}\ k \ge 1}$$

Characterize the structure of an optimal solution.

Recursively define the value of an optimal solution.

Compute the value of an optimal solution in a bottom-up fashion.

Construct an optimal solution from computed information.

Knapsack Problem

- Knapsack Problem can be viewed as sequence of result. Where condition should be satisfied and profit should be maximised.
- Where condition must be satisfied as follows $\Sigma w_i n_i \le m$ and $0 \le n_i \le 1$ and $1 \le i \le n$ where m is the maximum weight of conditions.
- And profit can be computed as $= \Sigma P_i n_i$

Example. *Assume that m = 30 and n = 3. While weights are (10, 20, 30) and profit are (12, 20, 24). Then find the optimal solution using knap-sack technique.*

Sol. Given that $m = 30, n = 3,$

and $(n_1, n_2, n_3) = \{1, 1, 0\}$

weights are $\{w_1, w_2, w_3\} = \{10, 20, 30\}$

and profits $\{P_1, P_2, P_3\} = \{12, 20, 24\}$

we assume that $\{n_1, n_2, n_3\} = \{1, 1, 0\}$

which provide highest (optimal) profit.

$$\Sigma w_i n_i \le m \text{ condition}$$
$$w_1n_1 + w_2n_2 + w_3n_3 \le 30$$
$$10 \times 1 + 20 \times 1 + 30 \times 0 \le 30$$
$$10 + 20 + 0 \le 30 \Rightarrow 30 \le 30 \ \{30 = 30\} \text{ condition satisfies.}$$

Then

for profit $\Sigma P_i n_i = 12 \times 1 + 20 \times 1 + 24 \times 0$

optimal profit $(\Sigma P_i n_i) = \mathbf{32}$ **Ans.**

0/1 Knapsack Problem

Knapsack problem is sequence of result. Where $g_i(y)$ value be optimal value and solution obtained by it, called optimal solution $g_i(y)$ can be calculated as follows:

$$g_i(n) = \max\{g_{i+1}(n), g_{i+1}(n - w_{i+1}) + p_{i+1}\}$$
$$g_i(n) = 0 \text{ if } i = m \text{ and } n \ge 0,$$
$$\boxed{g_i(n) = \max\{g_{i+1}(n), g_{i+1}(n - w_{i+1}) + p_{i+1}}$$
$$g_i(n) = -\infty \text{ if } \quad i = m \text{ and } n < 0$$

Example. *Calculate value of $g_0(5)$ where n = 5, m = 3, and weight are $\{w_1, w_2, w_3\}$ = {2, 3, 4} and profits are $\{P_1, P_2, P_3\}$ = {1, 2, 5}.*

Sol.

$$g_0(5) = \max\{g_1(5), g_1(5-2) + 1\}$$
$$g_0(5) = \max\{g_1(5), g_1(3) + 1\} \quad \ldots(1)$$
$$g_1(5) = \max\{g_2(5), g_2(5-3) + 2\}$$
$$g_1(5) = \max\{g_2(5), g_2(2) + 2\} \quad \ldots(2)$$
$$g_2(5) = \max\{g_3(5), g_3(5-4) + 5\}$$
$$g_2(5) = \max\{g_3(5), g_3(1) + 5\} \quad \ldots(3)$$
$$g_i(n) = 0 \quad \text{if} \quad i = m \text{ and } n \ge 0$$
$$g_i(n) = -\infty \quad \text{if} \quad i = m \text{ and } n < 0$$
$$g_2(5) = \max(0, 0 + 5\}$$
$$g_2(5) = 5 \quad \ldots(4)$$
$$g_2(2) = \max\{g_3(2), g_3(2-4) + 5\}$$

$g_2(2) = \max\{0, -\infty + 5\}$

$g_2(2) = 0$ ...(5)

Putting value of equations (4) and (5) at equation (2) we get

$g_1(5) = \max\{g_2(5), g_2(2) + 2\}$

$g_1(5) = \max\{5, 0 + 2\}$

$g_1(5) = 5$ ...(6)

$g_1(3) = \max\{g_2(3), g_2(3 - 3) + 2\}$

$g_1(3) = \max\{g_2(3), g_2(0) + 2\}$ ...(7)

$g_2(3) = \max\{g_3(3), g_3(3 - 4) + 5\}$

$g_2(3) = \max\{g_3(3), g_3(-1) + 5\}$

Since $g_i(n) = 0$ if $i = m$ and $n \geq 0$

$g_i(n) = -\infty$ if $i = m$ and $n < 0$

$g_2(3) = \max\{0, -\infty + 5\}$

$g_2(3) = 0$ ...(8)

$g_2(0) = \max\{g_3(0), g_3(0 - 4) + 5\}$

$g_2(0) = \max\{0, -\infty + 5\}$

$g_2(0) = 0$ ...(9)

From equations (8), (9) and (7) we get

$g_1(3) = \max\{g_2(3), g_2(0) + 2\}$

$g_1(3) = \max\{0, 0 + 2\}$

$g_1(3) = 2$ ...(10)

Putting value of equations (6) and (10) in to (1) we get

$g_0(5) = \max\{g_1(5), g_1(3) + 1\}$

$g(5) = \max\{5, 2 + 1\}$

$g_0(5) = \max\{5, 3\}$

$\boxed{g_0(5) = 5}$ **Ans.**

3.2 BACK TRACKING

Back tracking can be described as an organized exhaustive search which often avoids searching all possibilities. This technique is generally suitable for solving problems where a potentially large, but finite number of solutions have to be inspected.

Here the desired solution is to express as an n-tuple $(x_1, x_2, x_3, ..., x_n)$ where the x_i are chosen from some finite S_i.

If m_i is the size of set S_i. Then there are $m = m_1 m_2 ... m_n$ tuples that are possible candidates. The back tracking algorithm has its virtue the ability to yield the same answer with far fewer than m trials. Its back idea is to build up the solution vector one component at a time and to use some criterion function $P_i(x_1 x_2 ... x_i)$ to test whether the vector being formed has any chance of success.

If it is realize that partial vector $(x_1, x_2 ... x_i)$ can in no way lead to an optimal solution, then $m_{i+1} m_{i+2} ... m_n$ possible test vectors can be ignored entirely.

NOTES

Terms Used in Back Tracking

Explicit constraints. These are rules that restrict each x_i to take values only from a given set.

e.g.,
$$X_i > = 0$$
$$x_i = 0 \quad \text{or} \quad 1$$

Implicit constraints. These are rules which describes the way in which the x_i must relate to each other.

Solution space. It is the set of all tuples that satisfy the explicit constraints.

State space tree. The solution space is organized as a tree called state space tree.

Live node. A node which has been generated and all of whose children have not been generated is called a live node.

Dead node. It is a generated node which is not to be expanded further or all of whose children have been generated.

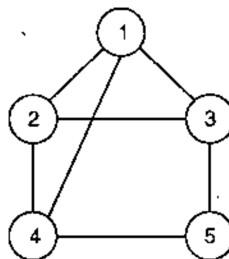
E-node. The live node whose children are currently being generated is called E-node.

Bounding function. Is a function created which is used to kill live nodes without generating all its children.

Graph coloring. Let G be a graph and m be a given positive integer, we want to discover whether the node of G can be colored in such a way that no two adjacent node have the same color. Yet only m color are used to color vertices. This is termed as "**m-colorable decision problem**".

- If d is degree of given graph G(V, E). Then it can colored with $d + 1$ color.
- The m-colorability optimization problem asks for the smallest integer m for which the graph G can colored.
- Integer is referred to as the *'chromatic number'* of graph G(V. E)

Example 1.



Let G(V, E) be a graph.

Then calculate maximum numbers of color needed to color the graph G(V, E).

Sol. Degree of Graph G(V, E) = 3

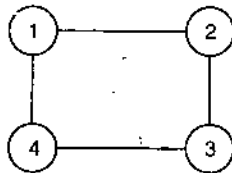
(Since one vertex is attached to maximum 3 vertex)

Hence maximum number of color needed to color the graph $= d + 1$

$$= 3 + 1 = 4$$

But graph can be closed from 3 color distinctly.

Example 3.



Calculate numbers of color needed to color differently adjacent.

Sol. Degree of graph = 2.

Maximum numbers of color needed to color the graph = $d + 1$

$= 3 + 1 = 4.$

But graph can be colored from 2 color distinctly.

Bicycle Problem

Consider a combination lock for a bicycle that consists of a set of N switches each of which can be 'on' or 'off'.

Exactly one setting of all the switches with [N/2] or more in the 'on' position, will open the lock. Suppose we have forgotten this combination and must get the lock open.

Suppose also that you are willing to try (if necessary) all combinations. We need an algorithm for systematically generating these combinations.

If we ignore the $n/2$ condition (given clue), there are $2n$ possible combinations for the lock. We model each possible combination with an n-tuple of 0's and 1's. 1 for the switch on 0 for switch off.

We can represent all combinations as binary tree.

1 → 11 → 111 → 1111 (check) → 111 (back tr.)

→ 1110 (check) → 111 (back tr.) → 11 (back tr.) → 110 → 1101 (check)

→ 110 (back tr.) → 1100 → 110 (back tr.) → 11 (back tr.) → 1 (back tr.)

→ 10 → 101 → 1011 (check) → 101 (back tr.) → 1010 (check) → 101 (back tr.)

→ 10 (back tr.) → 100 → 1001 (check) → 100 (back tr.) → 10 (back tr.)

→ 1 (back tr.) → root → 0....

Using this binary tree we can now state a back track procedure for generating only those combinations with at least $[n/2]$ switches on.

This algorithm amounts to traversal of the tree.

Move down the tree as far as possible to the left until we can move no further.

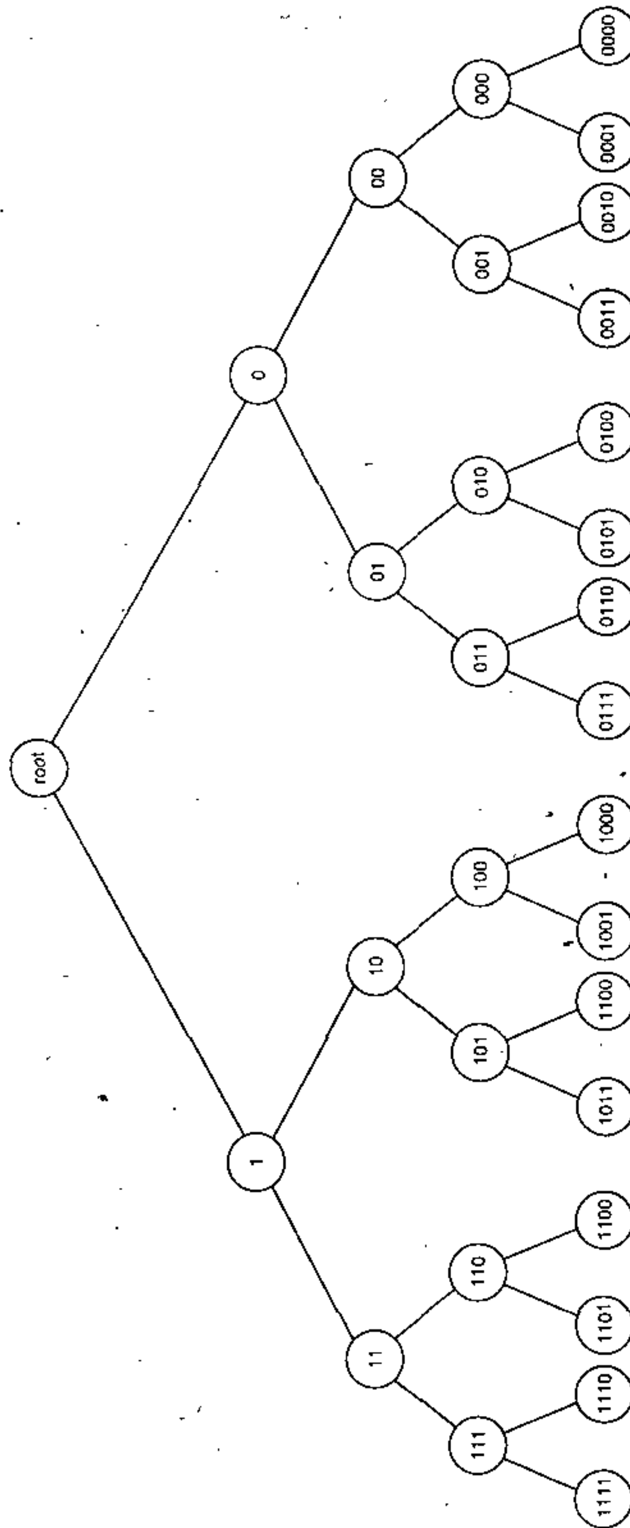
Upon reaching an end vertex, try the corresponding combination.

If it fails, back track up one level and see if we can move down again along an unsearched branch. If it is possible take the leftmost unused branch. If not back track up one more level and try to move down from this vertex.

Before moving down check if it is possible to satisfy the $[n/2]$ restriction at any successor vertex.

Algorithm terminates when we return to the root and there are no unused branches remaining.

NOTES

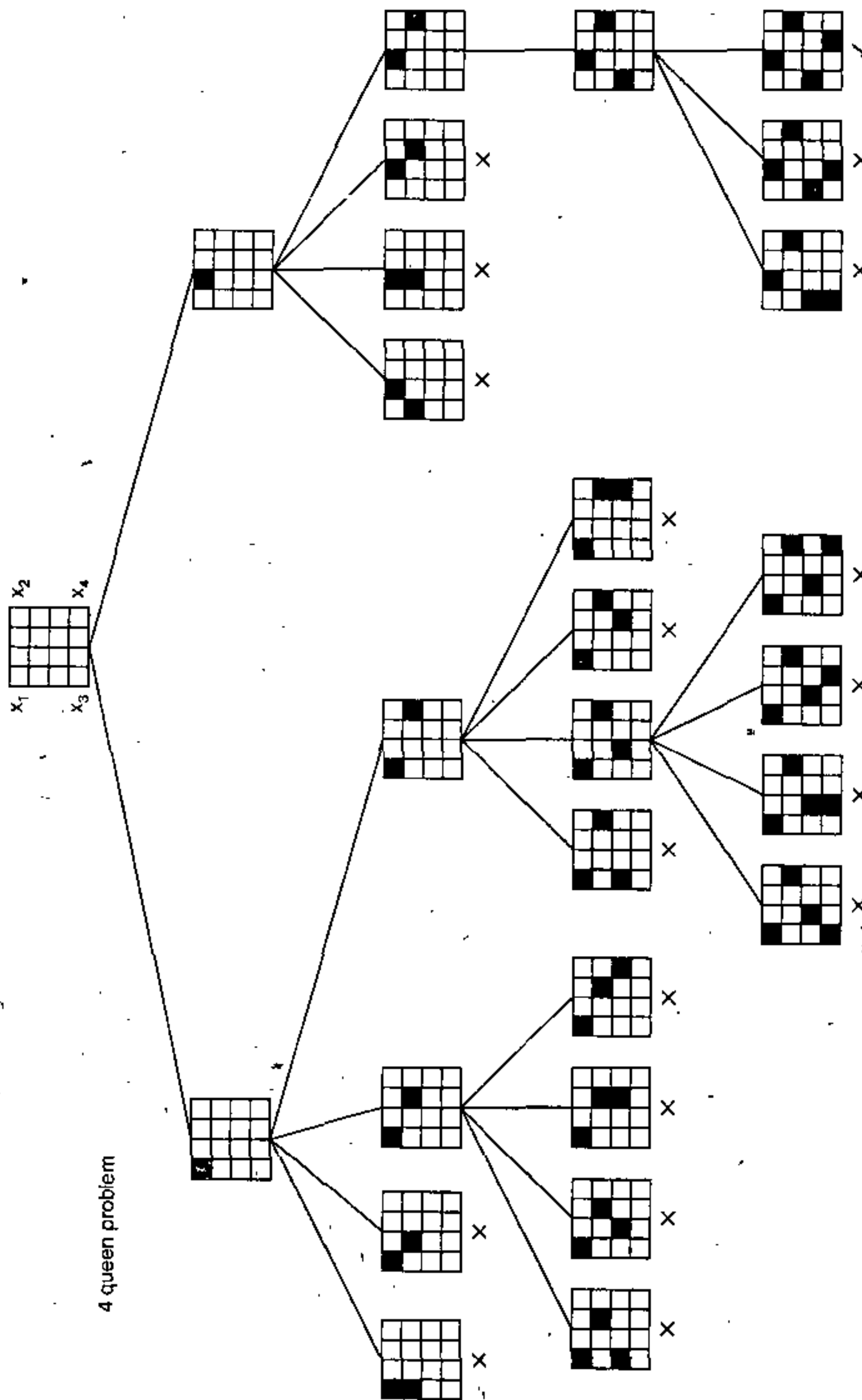


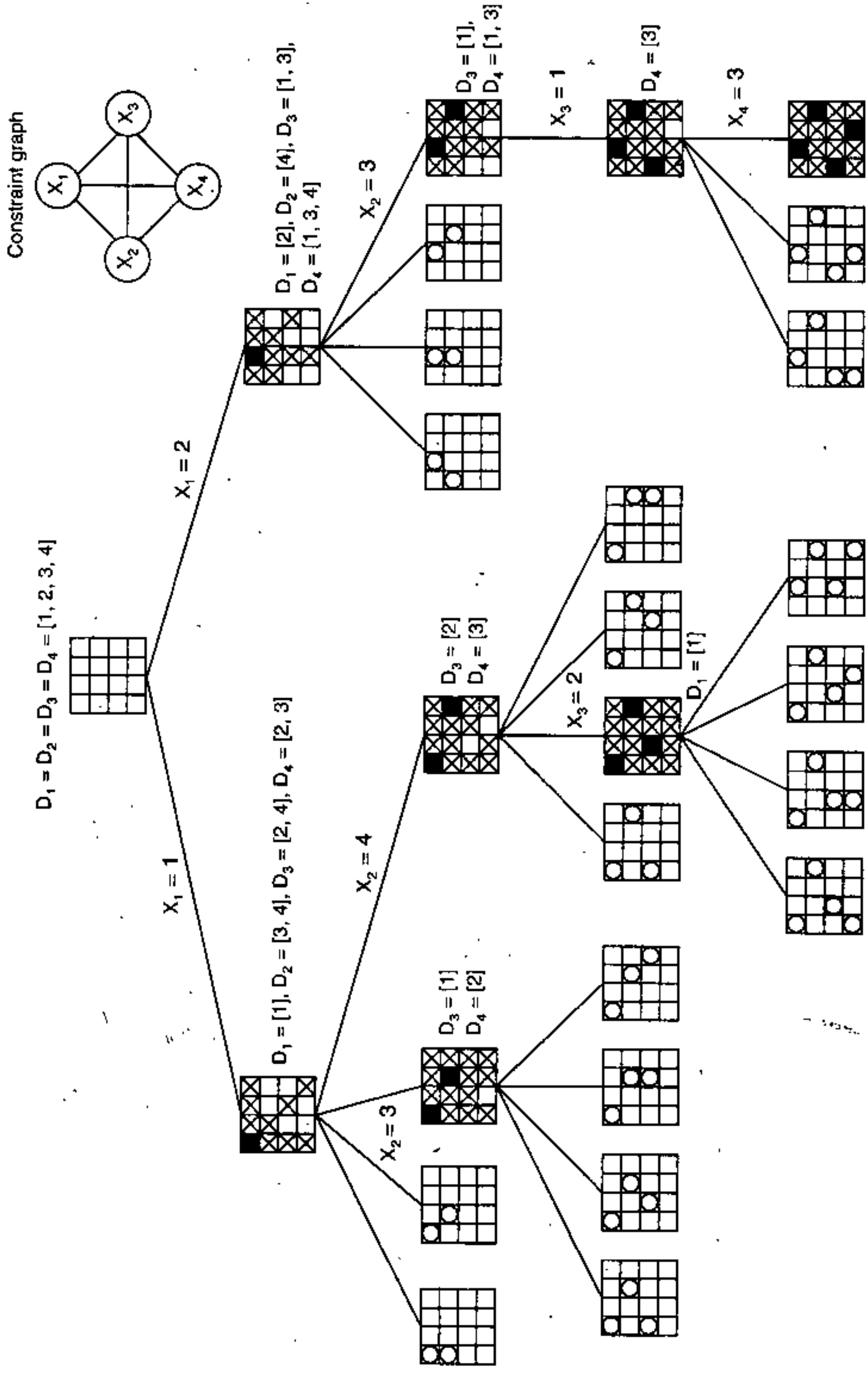
4-Queen Problem

To place 4-queens on an 4 by 4 chessboard so that no two "attack" *i.e.*, no two queens of them are on the same row, column, or diagonal. To solve it by back tracking approach we must see the solution as an n-tuple $(x_1, ..., x_n)$ where each x_i is coming from a finite set. Let us number the queens 1 to 4 also number rows and columns from 1 to 4.

Since each queen has to be in different row we can assume that queen 1 moves in row 1, queen 2 moves in row 2 and so on. Using this convention, any non attacking placement of queens can be represented as (c_1, c_2, c_3, c_4). c_i represents the column number in which queen i is placed.

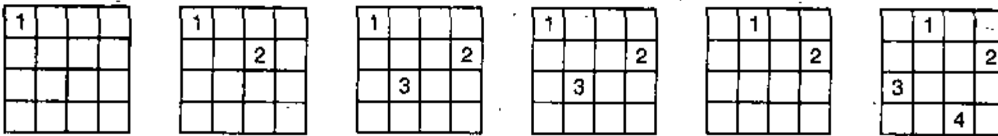
Similarly one position give the mirror image of fourth position and 2 give the mirror image of third position.



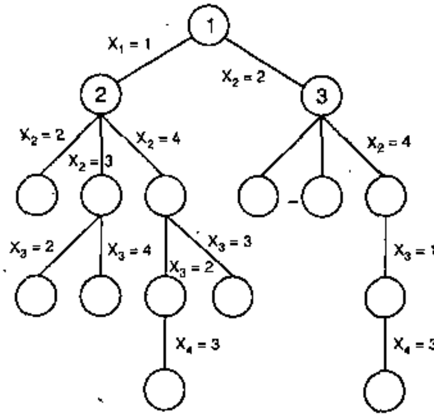


And queen can be placed at following possible position.

NOTES

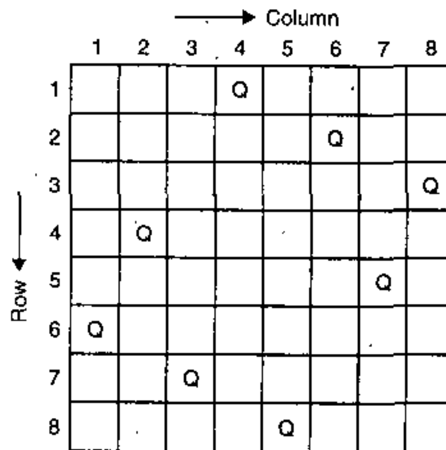


And there placed position can be represented in the form of tree.



8-Queen Problem

8-queen problem is similar to 8 × 8 chessboard where no two queen are placed at attacking position. *i.e.,* they will not placed diagonally, same, row wise column wise same,



Make tree similarly as four queen problem.

Sum of Subsets Problem

- Suppose we are given n distinct positive numbers and we desire to find all combinations of these numbers whose sums are m. This is called sum of subset problem.
- In sum of subset problem edge are labeled such that an edge from a level i node to a level $i + 1$ node represents a value for x_i. At each node, the solution space is partitioned into sub solution spaces. The solution space is defined by

NOTES

all paths from the root node to any node in the tree, since any such path corresponds to a subset satisfying the explicit constraints.

- In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.
- We will take on left side $x_i = 1$ and on right side $x_i = 0$ at each level.
- And bounding function is $B_k(x_1, x_2, ..., x_k)$ = true if $\sum_{i=1}^{k} w_i n_i + \sum_{i=k+1}^{n} w_i \geq m$
- If $\sum_{i=1}^{n} w_i n_i + \sum_{i=k+1}^{n} w_i \leq m$ then we will go further *i.e.,* node be live node. Other wise we have to back track the other nodes.

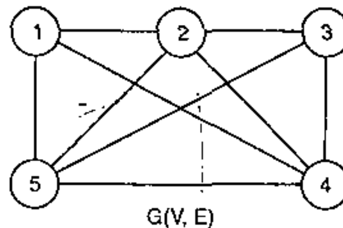
Algorithm of Sum of Subset

- Case Study 2: Sum of Subsets (cont.)
- The following incarnation *Sum* of the general back tracking algorithm
- Has two additional formal parameters:
- *s* for the sum of elements that has been selected.
- *r* for the overall sum of the remaining stock.
- **Algorithm** *Backtrack* (*n*)
- {for each child *x* [*n*] of *x* [*n* – 1] in T do
- If *x* [1 ..*n*] does not meet the boundary B
- then {if *x* [1..*n*] is a path to a leaf of T
- then {if the leaf meets the implicit constraint C
- then output the leaf and the path *x* [1..*n*]
- }
- else Backtrack (*n* + 1)
- }
- }
- **Algorithm** Sum (*n, s, r*)
- {*x* [*n*]: = 1;
- if (*s* < = M) & (*r* > = M – *s*) & (*w* [*n*] < = M – *s*)
- then {if M = *s* + *w* [*n*]
- then output all *i* [1, *n*] that *x*[*i*] = 1
- else Sum (*n* + 1, *s* + *w*[*n*], *r* – *w* [*n*]
- };
- *x*[*n*];=0;
- if (*s*<=M) & (*r*>=M–*s*) & (*w*[*n*]<=M–*s*)
- then { if M = *s*
- then output all *i*Î[1, *n*] that *x* [*i*] = 1
- else Sum (*n* + 1, *s*, *r* – *w*[*n*])
- }
- }

Hamiltonian Cycle

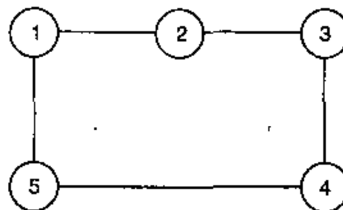
Let G(V, E) be a connected graph with n-verties. A hamiltonian cycle is a round trip path along n-edge of G(V, E) where V is vertex and E is an edge. $(u, v) \in E$, $u \in v_i$ and $v \in v_{i+1}$ and at hamiltonian cycle each and every vertex is visited and return to its starting position (it is mostly similar to travelling sales person problem). *i.e.*, if a hamiltonian cycle begin at some vertex $v_1 \in G$ and vertex of G are visited in order $v_1, v_2, ..., v_n, v_{n+1}$ and edge $(v_i, v_{i+1}) \in E$ for $1 \le i \le n$ and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

Example.



Graph G(V, E) is given, construct Hamiltonian cycle for it.

Sol.



0–1 Knapsack Problem

Given n positive weights w_i, n positive profits p_i and a positive number M which is the knapsack capacity. The problem calls for choosing a subset of the weights such that is maximized.

$$\sum_{1 \le i \le n} w_i x_i \le M \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i$$

We select the value of X_i such that profit should be maximized. Two possible tree organization is possible. One correspond to the fixed tuple size formulation and the other to the variable tuple size formulation a good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtained by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far then this live node may be killed.

Fixed Tuple Size Formulation

If at a node Z the value of x_i, $1 \le i \le k$ have already been determined, then an upper bound for Z can be obtained by relaxing the requirement

$$x_i = 0 \quad \text{or} \quad 1 \quad \text{to} \quad 0 \le x_i \le 1 \quad \text{for } k + 1 \le i \le n$$

And it can be solved by simple greedy method.

Example: P = (11, 21, 31, 33, 43, 53, 25, 35)

W = (1, 11, 21, 23, 33, 43, 55, 65)

NOTES

M = 110

N = 6

Relaxing 0, 1 condition and using greedy approach solution is (1, 1, 1, 1, 1, 21/43).

Having 164.88 profit.

This value will work as an upper bound for the required solution.

3.3 GREEDY METHOD

This method is used to design an algorithm. Greedy method provide optimal result, and it is also provide fast and non-optimal solution, which are feasible. This method less powerful in comparision of dynamic programming technique and not possesing the wide area of application by using this technique we can few problem which are most known problem they are knap-sack problem, travelling sales person problem, Job sequencing problem etc. The most useful terms for solving problem is:

- Feasible solution
- Local optimal choice
- Unalterable problem.

Feasible Solution

- Here we check the feasibility of problem *i.e.,* given problem can be solved by greedy approach or not and try to get one solution of the problem.
- If a problem have n-inputs and require to obtain a subset that satisfies some constraints that subset satisfies these constraint is called feasible solution.

Local Optimal Choice

- It we find the feasible solution of given problem, then feasible solution is called local optimal solution of that problem.
- The selection process of the problem solution is called optimization measurement of the problem.

Unalterable Problem

- When we choose some feasible solution for a problem then choice of that feasible solution can not be change.

Algorithm for Greedy Technique

Greedy (a, b)

// Greedy problem contains exactly b inputs.

Step 1.

Result ← ϕ

// initially assume that there is no (technique)

// solution of problem

do i ← 1 to b

n = choose (a)

while feasible (choose, m) then

Step 2. Choose = union (choose, m)

Step 3. Return choose

Knapsack Problem

Knapsack problem provide the optimal feasible solution of given problem, where condition must be satisfies *i.e.*, total sum of weight with multiplication of selected n_i will not exceed the knap-sack limit, then we calculate the maximum profit in the n_i value.

condition $\quad \sum w_i n_i \le m$
$\qquad 1 \le i \le j$

profit = $\sum p_i n_i$

where $1 \le i \le j$

j is equal to number of objects and $0 \le n_i \le 1$.

Example. *For following instance calculate knap-sack solution where $j = 3$, $m = 30$ and weight are (10, 12, 15) and profit are (20, 28, 22).*

Sol. Given $(w_1, w_2, w_3) = \{10, 12, 15\}$ and profit $\{p_1, p_2, p_3\} = \{20, 28, 22\}$ then

Solution No.	$\{n_1, n_2, n_3\}$	$m \ge \sum w_i n_i$	$\sum p_i n_i$
1.	$\left(1, 1, \frac{8}{15}\right)$	30	59.73
2.	$\left(\frac{3}{10}, 1, 1\right)$	30	56
3.	$\left(1, \frac{1}{4}, 1\right)$	30	49
4.	$\left(\frac{3}{5}, 1, \frac{2}{5}\right)$	30	48.80

Hence optimal profit = 59.73

and $\quad n_i$ *i.e.*, $(n_1, n_2, n_3) = \left(1, 1, \frac{8}{15}\right)$.

Theorem. If $p_1/w_1 \ge p_2/w_2 \ge \ldots p_n/w_n$ then selecting items w.r.t., this ordering gives optimal solution.

Proof. Let $x = (x_1 \ldots\ldots x_n)$ be the solution so generated. If all the x_i's are one then clearly the solution is optimal. So let j be the least index such that $x_j \ne 1$.

So, $X_i = 1$ for all $1 \le i < j$ and $x_i = 0$ for $j < i \le n$ and $0 \le x_j < 1$.

Let $y = (y_1, y_2, \ldots\ldots, y_n)$ be an optimal solution. By obvious reasons $\sum w_i y_i = m$

Let k be the least index such that $y_k \ne x_k$. It is easy to reason that $y_k < x_k$.

Now suppose we increase y_k to x_k and decrease as many of $(y_{k+1} \ldots\ldots y_n)$ as necessary so that total capacity is still m. This results in a new solution

$$Z = (z_1, z_2, \ldots\ldots, z_n)$$

where $\quad z_i = x_i \quad 1 \le i \le k$

and $\quad \sum_{k < i \le n} w_i (y_i - z_i) = w_k(z_k - y_k)$

$$\sum_{1 \le i \le n} p_i z_i = \sum_{1 \le i \le n} p_i y_i + (z_k - y_k)\, w_k\, p_k/w_k - \sum_{1 \le i \le n} (y_i - z_i)\, w_i\, p_i/w_i$$

NOTES

$$\geq \sum_{1 \leq i \leq n} p_i y_i + [(z_k - y_k) w_k - \sum_{k < i \leq n} (y_i - z_i) w_i] p$$

$$= \sum_{1 \leq i \leq n} p_i y_i$$

If $\Sigma p_i z_i > \Sigma p_i y_i$ then y could not have been optimal solution.

- If these sums are equal then either $z = x$ and x is optimal or $z \neq x$.
- In the latter case repeated use of above argument y can be transformed into x and thus x too is also optimal.

Minimum Spanning Tree Problem

Minimum spanning tree problem can be analyzed by following technique.

- Kruskal's algorithm
- Prim's algorithm
- Both are based on greedy algorithm
- Algorithm and proof of getting optimal solution

Entropy and Its Importance

The measurement of contents of information in the message obtained is termed as entropy of information. The greater the entropy the higher is the information. If the entropy of an message is high then it also have the higher potential for compression.

It is related with probability of occurrence of characters and is defined by a mathematical formula

$$E = \log_2 \text{(character probability)}$$

i.e., $$E = -\log_2 P$$

The entropy can also be used to determine the number of bits present in a stream of message.

Example. *Character 'a' has a probability* $\frac{1}{8}$. *If as message string in "a a a a a a" determine number of bit saved.*

Sol. $$E = -\log_2 p = -\log_2 \left(\frac{1}{8}\right) = 3$$

$$= -\log_2 1 - (-\log_2 8) = 3$$

Total numbers of character = 6

Numbers of bit required = 6 × 3 = 18

Now in ASCII code bit required = 8 × 6 = 48

Bit saved = 48 – 18 = 30

Note. If P(A) and P(B) be probability of occurrence of event A and B. Then self information S_i can be expressed as

$$S_i(A) = -\log_2 (P(A)),\ S_i(B) = -\log_2 (P(B))$$

and $$S_i(AB) = -\log_2 (P(AB))$$

$$S_i(AB) = -(\log_2 P(A) + \log_2 P(B))$$
$$S_i(AB) = S_i(A) + S_i(B)$$

Entropy Function

To determine degree of randomness or disorder in the data. The Shannon proposed following entropy function. Suppose there are 'n' possible of outcomes of an event and P_i denotes the probability of ith outcomes, the entropy may be computed as

$$\text{Entropy of message stream} = -\sum_{i=1}^{n} P_i * \log_2 (P_i)$$

Example. *Compute the self-information and entropy of following message stream, "AA BA CDA CDB ABC AB".*

Sol. Total numbers of character in message (N) = 15 total number of characters, their probability and self-information (entropy) is

Character	*Probability*	*Self-information* $= -\log_2 p$
'A'	6/15	$-\log_2 6/15 = 1.32$
'B'	4/15	1.90
'C'	3/15	2.32
'D'	2/15	2.90

$$\text{Entropy of Message} = \frac{1}{N} * \sum_{i=1}^{n} \text{entropy of character}$$

$$= \frac{1}{15} * (1.32 + 1.32 + 1.90 + 1.32 + 2.32 + 1.32 + 2.32 + 2.90 + 1.90 + 1.32 + 1.90 + 2.32 + 1.32 + 1.90)$$

$$= 1.88$$

or

$$\text{Entropy of message} = -\sum_{i=1}^{n} P_i \log (P_i)$$

$$= \frac{6}{15} * 1.32 + \frac{4}{15} * 1.90 + \frac{3}{15} * 2.32 + \frac{2}{15} * 2.90$$

$$= .528 + .506 + .464 + .386$$

$$= 1.88$$

Motivating Factors for Data Compression

Shannon's work in information theory has been widely accepted in communication and data compression the concept of entropy and self information are used to develop the efficient code. These well requires less amount of information bits to represent the data.

Example. *Suppose message consist four character 'A', 'B', 'C', 'D' and they sent over communication channels and probability of occurrence of these character is pa, pb, pc, pd respectively.*

Sol. $pa + pb + pc + pd = 1$

$$\text{Total number of bits } (m) = \log_2 N$$

$$= \log_2 4 = 2$$

NOTES

If probability of occurrence is equal then it will be $\frac{1}{4} = 0.25$

$$\text{Entropy} = \sum_{i=1}^{n} p_i * \log_2 (p_i)$$

$$= (.25 * \log_2 (.25) + .25 * \log_2 (.25) + .25 * \log_2 (.25) + .25 * (\log_2 .25))$$

$$= 2$$

Therefore size of well required two bits to represent an four (4) character.

If message consist 100 character.

Then total numbers of bits required = 2 * 100 = 200 bits.

Character	*Code*
A	00
B	01
C	10
D	11

Static Coding

In static coding, fixed sized codes are allocated to each symbol, each symbol can be uniquely identified by its corresponding code. It is also possible to compute the minimum number of bits required to represent a symbol.

Suppose there are m symbols which are used to constitute a message or text.

Let N ≡ Minimum number of digits required to represent m distinct symbol.

Let i be the base of numbers system

$$N = \lceil \log_i (m) \rceil$$

In digital computer system. We represent the data in binary form, thus the minimum number of bits required to uniquely represent a symbol will be $N = \lceil \log_2 (m) \rceil$.

Example. *Suppose a message is composed of five symbols 'a', 'b', 'c', 'd', 'e' compute the following:*

1. *Find numbers of bits required to represent/code each symbol uniquely.*
2. *Generate the code for all symbols.*
3. *Find the coded form of message string 'bddac'.*

Sol. Total number of distinct symbol M = 5

Total number of bits required $N = \lceil \log_2 (m) \rceil$

$$N = \lceil \log_2 5 \rceil = \lceil \log_2 5 \rceil = 3$$

Thus 3 bit code will be required to represent each symbol or the minimum number of bits required to represent a symbol uniquely is 3.

Using 3 digit the following unique code may be generated

Static Code	*Symbol*
000	*a*
001	*b*
010	*c*
011	*d*
100	*e*
⋮	⋮
111	

Total number of unique code generated = $2^3 = 8$ we may assign any five codes to these symbols as mentioned in the table above.

Using the scheme, coded string *'bddac'* is as follows

001 011 011 000 010

Thus string *bddac* would required = 5 * 3 = 15 digits to code.

Dynamic Coding (Variable Size Coding)

Dynamic coding is done using variable size code. This method is based on the principle of identifying the symbols which appear frequently.

Example. *Suppose a text or message may be composed of four symbols. These symbol are a, b, c, d frequency distribution of occurrence of each symbol is as under*

Symbol	*Frequency*
'a'	15
'b'	10
'c'	70
'd'	5

Suppose a text containing 1000 symbols has to be compressed compute following:

1. Total numbers of bits required to represent the whole text using ASCII codes.

2. Total numbers of bits required to represent the whole text using fixed size code / static code generate the static code for all symbols.

3. Total number of bits required to represent the whole text using dynamic coding / variable length codes. Consider the frequency distribution generate the dynamic codes for all symbols.

Sol. 1. Number of bits used in ASCII code = 8 bits

total number of symbols in text = 1000

total number of bits to represent the whole text = 1000 × 8 = 8000 bits.

2. Total number of distinct symbol M = 4.

Total number of bits required to represent each symbol

$$N = \lceil \log_2 (m) \rceil = \lceil \log_2 4 \rceil = 2.$$

NOTES

Thus 2 bit code is required to represent each symbol. The code may be allocated as follows:

Symbol	*Code*
a	00
b	01
c	10
d	11

Total number of bits required to represent a text in this scheme = 1000 * N = 1000 * 2 = 2000 bits

3. C is most frequency symbol, it may be given one bit code. Similarly *a*, *b*, and *d* be in 2, 3 and 4 bits.

Total bits required to represent text containing 1000 symbol = 1 * total occurrence symbol '*c*' + 2*. Total occurrence of symbol '*a*' + 3 * total occurrence of symbol '*b*' + 4 * total occurrence of symbol '*d*'

= 1 * 700 + 2 * 150 + 3 * 100 + 4 * 50

= 700 + 300 + 300 + 200 = 1,500 bits.

Data Processing Concepts

Data processing mainly consist 3 main step thats contribute or make data processing possible they are

1. Input
2. Output
3. Process.

When these 3 steps combines togather then they are referred as system. System can broadly be defined as group of interrelated components to achieve common goal. Each element plays equal and important role in functioning of system.

Data processing activity can be categorised on following 5 parts.

(*i*) Collection

(*ii*) Conversion

(*iii*) Manipulation

(*iv*) Storage

(*v*) Communication.

Huffman Algorithms

It generate variable length code variable length code in such a way that high frequency symbol are represented with minimum number of bits and low frequency symbol are represented with high number of bits.

Step for Generating Huffman Tree

1. Compute the total number of symbol
2. Calculate their frequency

3. Arrange them descending order
4. Construct huffman tree.
   (*i*) Pick up two symbols with minimum frequency.
   (*ii*) Create two tree nodes of Binary tree and name them.
   (*iii*) Create a parent node having frequency equal to the sum of frequency of child nodes.
   (*iv*) Delete there 2 frequency from list.
   (*v*) Add parent nodes to the list
   (*vi*) Re-arrange the list in descending order.
5. Repeat step 4(*i*) to 4(*vi*) until list become empty.
6. Generate Huffman tree.

Example.

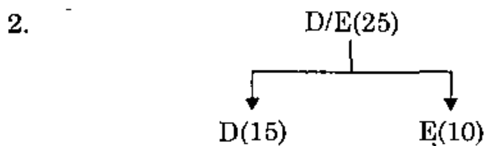
Symbol	*Frequency*
A	*40*
B	*20*
C	*15*
E	*10*
D	*15*

Construct Huffman tree.

Sol. 1.

A	40
B	20
C	15
D	15
E	10

2.



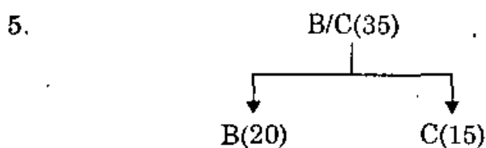
3.

A	40
B	20
C	15
D/E	25

4.

A	40
D/E	25
B	20
C	15

5.



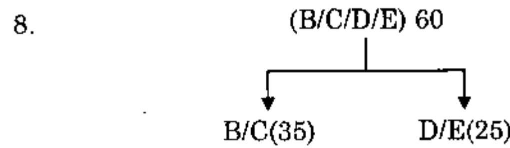
6.

A	40
B/C	35
D/E	25

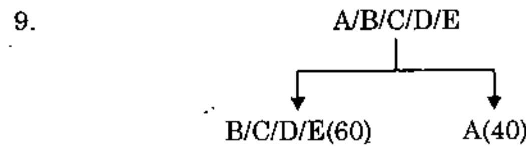
7.

B/C D/E	60
A	40

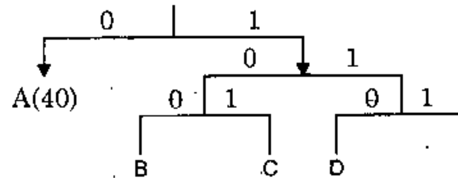
8.



9.



So final



Assign 0 to left 1 to right **Huffman code**

Symbol	*Frequency*	*Code*	*Size*
A	40	0	1
B	30	100	3
C	20	101	3
D	15	110	3
E	10	111	3

Total number of bits required

$$= 1 \times 40 + 3 \times 30 + 3 \times 20 + 3 \times 15 + 3 \times 10$$
$$= 40 + 90 + 60 + 45 + 30$$
$$= 265$$

Example. *The symbols and their frequency are*

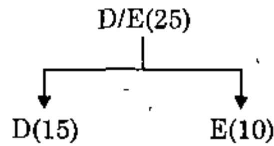
Symbol	*Frequency*
A	*30*
B	*30*
D	*15*
E	*10*
C	*15*

Construct Huffman tree and generate code.

Sol. 1. Sorted list is:

Symbol	*Frequency*
A	30
B	30
C	15
D	15
E	10

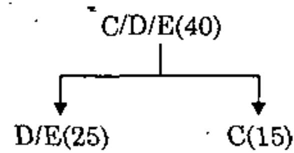
2. Pick symbol with minimum frequency they are D and E.



Combined frequency is 25.

Symbol	*Frequency*
A	30
B	30
D/E	25
C	15

3. Combine C, D/E

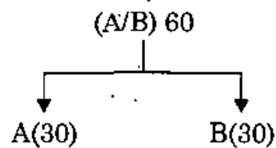


Symbol	*Frequency*
C/D/E	40
A	30
B	30

Sorted list as shown below:

D/E/C, A, B

4. Pick up two symbols with minimum frequency. These symbols are 'A' and 'B'. Assign them tree node as given below

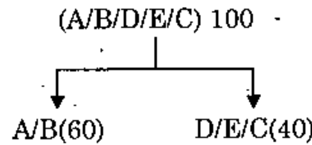


This subtree is called 'A/B' and combined frequency is 60.

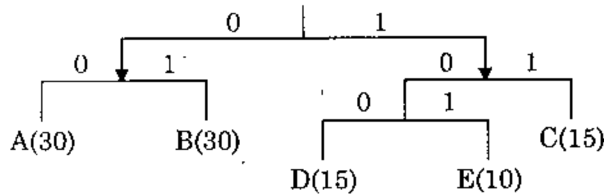
NOTES

5. Update sorted list as shown below:

[(A/B), (D/E/C)]



6.



7. Assign the bit 0 and 1 to left and right subtree

Symbol	*Frequency*	*Huffman Code*	*Size of Huffman Code*
A	30	00	2
B	30	01	2
C	15	11	2
D	15	100	3
E	10	101	3

Using Huffman code, the total number of bits required to represent a text of 100

$= 30 \times 2 + 30 \times 2 + 15 \times 2 + 15 \times 3 + 10 \times 3$

$= 225$

$$\text{Entropy} = -\sum_{i=1}^{n} P_i \log_2 P_i$$

$$= -(.30 \log(.30) + .30 \log_2(.30) + .15 \log_2(.15) + .15 \log_2(.15) + .10 \log_2(.10))$$

Entropy = ?

Numbers of bits required in 100 text = 225/100 = 2.25

Redundency = numbers of bits required in 100 – entropy

= 2.25 – entropy

Extended Huffman Code

The difference in entropy and average size of code length seres as a indicator for redundency in the codes. Higher the difference, higher the redundency our overall objective for designing Huffman code is that it gives minimum redundency in all cases, we are not yielded with code which eliminates the redundant information up to considerable amount.

NOTES

Example. *Suppose a transmitter transmit three symbols. There symbols (A, B, C) have probability P(A) = .85, P(B) = .10, P(C) = .05 then calculate entropy?*

Sol.

Symbol	*Probability*	*Huffman Code*
A	.85	0
B	.10	10
C	.05	11

Average size of Huffman code/symbol = 1.15 bits

We may compute the entropy as

$$\text{Entropy} = -\sum_{i=1}^{N} P_i * \log_2 (P_i)$$

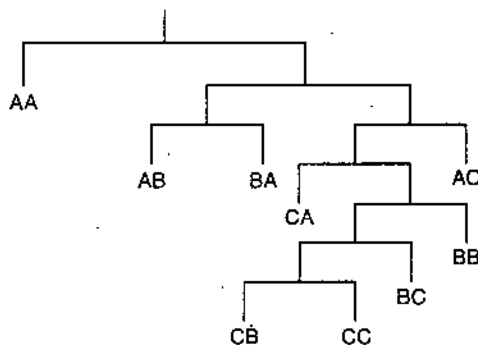
$$= .199 + .332 + .216$$

$$= .747 \text{ bit/symbol.}$$

Difference in entropy and average Huffman code size is considerable high. The redundency for this code is .403 which is 54% of the entropy.

Extended Huffman Group is

Group	*Probability*	*Huffman Code*
AA	.85 × .85 = .722	0
AB	.85 × .10 = .085	100
AC	.85 × .05 = .042	111
BA	.10 × .85 = .085	101
BB	.10 × .10 = .01	11011
BC	.10 × .05 = .005	110101
CA	.05 × .85 = .042	1110
CB	.05 × .10 = .005	1101000
CC	.05 × .05 = .0025	1101001



Average length of group of symbols = Total numbers of bits/100

NOTES

We may also calculate the revised entropy considering the group as a character as under

$$\text{Entropy} = -\sum_{i=1}^{n} P_i \log_2 (P_i)$$

$$= 1.490 \text{ bits/group}$$

The redundancy = 1.658 – 1.490 = .168 bits/group. Therefore redundancy here is only 11.2% of the entropy value. It proves that if we generate the extended binary code, we are left with only 11.2% redundancy.

Huffman Encoding—Time Complexity

- Sort keys $O(n \log n)$
- Repeat n times
  - — Form new sub-tree $O(1)$
  - — Move sub-tree (binary search) $O(\log n)$
  - — Total $O(n \log n)$
- Overall $O(n \log n)$

Theorem. Let C be a given alphabet with frequency $f(c)$. Let x and y be two characters in C with minimum frequencies. Let C′ oe the alphabet obtained from C as

$$C' = C - \{x, y\} \cup \{z\}.$$

Frequencies for new set is same as for C except that $f(z) = f(x) + f(y)$.

Let T′ be any optimal tree representing optimal code for C′. Then the tree T obtained from T′ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Proof. For any $C \in C - \{x, y\}$

$$dT(C) = dT'(C) \text{ but for } x \text{ and } y$$

$$dT(x) = dT(y) = dT'(z) + 1$$

We have

$$f(x)\, dT(x) + f(y)\, dT(y) = (f(x) + f(y))\, (dT(z) + 1)$$

$$= f(z)\, dT(z) + (f(x) + f(y))$$

$$= B(T) + B(T') + f(x) + f(y)$$

From which we conclude that

$$B(T) = B(T') + f(x) + f(y)$$

or

$$B(T') = B(T) - f(x) - f(y)$$

We now prove by contradiction.

Suppose that T is not optimal for C then there is another optimal tree T″ such that B(T″) < B(T)

Without any loss of generality we can assume that x and y are siblings here.

Let T‴ be the tree obtained from T″ with the common parent of x and y replaced by a leaf z with freq $f(z) = f(x) + f(y)$

- Then $B(T''') = B(T'') - f(x) - f(y)$

$$< B(T) - f(x) - f(y)$$

$$= B(T')$$

This contradicts the assumption that T′ represents an optimal prefix code for C′.

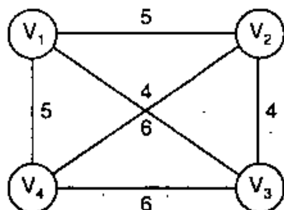
Thus T must represent an optimal prefix code for the alphabet C.

Travelling Sales Person Problem

In this problem a sales person visit exactly n cities in such a manner all cities must be exactly visited only once and sales person return to the that city in which he started the journey with minimum cost no one city visited twice except starting point. To evaluate the cost of visiting of each city can be calculated considering following point.

1. Travelling person can be start journey from any city, that will be assume V_1.
2. Then choose the adjacent city from city V_1, with minimal cost.
3. When we visited m cities from total n cities then remaining city will be $(n - m)$.
4. This process will be repeated until unless each and every city must be visited with minimal cost.
5. When all city are visited then return back to starting city.

Example. *Suppose a travelling person started his journey from city V_1 then calculate minimal cost of journey of graph G(V, E).*



Sol. Given graph G(V, E) can be expressed in form of matrix like

$$M_1 = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{matrix} V_1 & V_2 & V_3 & V_4 \end{matrix} \\ \begin{bmatrix} 0 & 5 & 4 & 5 \\ 5 & 0 & 4 & 6 \\ 4 & 4 & 0 & 6 \\ 5 & 6 & 6 & 0 \end{bmatrix} \end{array}$$

Starting with V_1 then select minimum cost from V_1 to $\{V_2, V_3, V_4\}$

$$M_2 = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{matrix} V_1 & V_2 & V_3 & V_4 \end{matrix} \\ \begin{bmatrix} 0 & 5 & 4 & 5 \\ 5 & 0 & 4 & 6 \\ 4 & 4 & 0 & 6 \\ 5 & 6 & 6 & 0 \end{bmatrix} \end{array} \qquad V_1 - V_3$$

Here V_3 to V_1 and V_2 have same cost, but we select V_2 since V_1 is already visited

$$M_3 = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{matrix} V_1 & V_2 & V_3 & V_4 \end{matrix} \\ \begin{bmatrix} 0 & 5 & 4 & 5 \\ 5 & 0 & 4 & 6 \\ 4 & 4 & 0 & 6 \\ 5 & 6 & 6 & 0 \end{bmatrix} \end{array} \qquad V_1 - V_3 - V_2$$

We will not select that city which have been already visited. So we will choose V_2 to V_4

NOTES

$$M_4 = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array}\begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{bmatrix} 0 & 5 & 4 & 5 \\ 5 & 0 & 4 & 6 \\ 4 & 4 & 0 & 6 \\ 5 & 6 & 6 & 0 \end{bmatrix} \end{array} \qquad V_1 - V_3 - V_2 - V_4$$

Then path from V_1 to all city

$$V_1 \xrightarrow{4} V_3 \xrightarrow{4} V_2 \xrightarrow{6} V_4 \xrightarrow{5} V_1$$

Total cost = 4 + 4 + 6 + 5 = 19

Some time it may be happen that cost from the travelling sales person technique is greater than general approach.

Job Sequencing with Dead Lines

We are given n jobs, associated with each job i there is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. This profit p_i will be earned only if job is completed before its deadline. Each job needs processing for one unit time on a single machine available.

Feasible solution is a subset of jobs each which can be completed without crossing any deadline. Optimal solution is that feasible solution with maximum profit.

Example. $n = 4, p = (80, 10, 25, 30)$ and $d = (2, 1, 2, 1)$

Possible feasible solutions:

Subset	*Sequence*	*Value*
(1, 2)	2, 1	90
(1, 3)	1, 3 or 3, 1	105
(1, 4)	4, 1	110
(2, 3)	2, 3	35
(3, 4)	4, 3	55
(1)	1	80
(2)	2	10
(3)	3	25
(4)	4	30

Greedy approach objective is to maximize Σp_i so next job to select in J is the one that increases Σp_i the most, subject to the restriction that set J is feasible. Feasibility test becomes very easy through the following result.

Theorem. *Let J be set of k jobs and S =* $i_1, i_2, ..., i_k$ *a permutation of jobs in J such that* $d_{i1} \leq d_{i2} \leq d_{ik}$. *Then J is feasible iff the jobs in J can be processed in the order S without violating any deadline.*

Proof. We need to show that if J is feasible set then a permutation S = $i_1, i_2, ..., i_k$ with $d_{i1} \leq d_{i2} \leq d_{ik}$ is also feasible. If J is a feasible set then there exists S′ = $r_1, r_2, ..., r_k$ such that $d_{rq} \geq q, 1 \leq q \leq k$, assume S ≠ S′. Let a be the least index such that $ra \neq rb$. Let $i_a = r_b$ for some $b > a$ in S′ we can interchange r_a and r_b, since $dra.drb$ the resulting permutation s''

S:	$i_1\, i_2 \ldots$	$i_a \ldots$	i_k
S′:	$r_1\, r_2 \ldots$	$r_a \ldots$	r_k

$d_{ia} < = d_{ra}$

Since S′ is feasible sequence, if we interchange r_a and r_b we get another feasible sequence. And new sequence is closer to S contain deadline.

Theorem. *Greedy approach always obtains an optimal solution.*

Proof. Let i be the set obtained by greedy method and let J be the set of jobs in the optimal solution. We will show that i and j have both have same profit values. We assume that $i \neq j$ also j cannot be subset of i as j is optimal also i can not be subset of j by the algorithm. So there exists jobs a in i such that a is not in j and b in j but not in i, Let a be a highest profit job such that a is in I and not in j. Clearly $p_a \geq p_b$ for all jobs that are in j but not in i. Since if $p_b > p_a$ then greedy approach would consider job b before job a and included into I. Let S_i and S_j be the sequences for feasible sets i and j. We will assume that common jobs of i and j are being processed in same time intervals. Let i be a job scheduled to t to $t + 1$ in S_i and t' to $t' + 1$ in S_j. If $t < t'$ then interchange the job if any in $[t', t' + 1]$ in S_i with job i. Similarly if $t' < t$ similar transformation can be done in S_j now consider the interval $[t_a, t_a + 1]$ in S_i in which job a is scheduled. Let b the job (if any) scheduled in S_j in this interval. $p_a \geq p_b$ from the choice of a So scheduling a from t_a to $t_a + 1$ in S_j and discarding job b gives a feasible schedule for the set $j' = j - \{b\} + \{a\}$. Clearly j' has profit no less than that of j and differs from i in one less job than j does. By repeatedly using this approach j can be transformed into i without decreasing the profit value.

3.4 AMORTIZED ANALYSIS

Binary Counter

- Count (in binary) from 0 to n:

$$0 \rightarrow 1 \rightarrow 10 \rightarrow 11 \rightarrow 100 \rightarrow 101 \rightarrow \ldots \rightarrow \text{bin}(n)$$

where bin (n) denotes the binary representation of n. Assume c_i cost of a single counting step from i to $i + 1$. $c_i = 1 +$ number of trailing '1's bin (i) because this is exactly the number of bits which change in this step. The total cost of counting from 0 to n is then $T_n = \Sigma_i = 0_{n-1}\, c_i$. Since all number between 0 to n can be represented using at most $1 + \log n$ bits. We have

$$c_i \leq 2 + \log n \text{ for all I} = 1, 2, \ldots, n - 1$$

Thus $T_n = O(n \log n)$

But this is overly pessimistic, because such a worst case will not happen for every operation, Only very few steps have cost $\log n$. Actually half of the steps have only cost 1 (The cost of incrementing an even number). So we should try to find a better estimate for the total cost of a sequence of *n operations*. Not just multiplying *n with the worst case cost of a single operation*. This is called amortized analysis. If any sequence of n operations takes time at most T_n then we say that each single operation needs amortized time = T_n/n.

Motivation

In amortized analysis we look for the time bound for a sequence of operations, instead of the cost of a single operation. The time bound for one operation (called amortized cost) is calculated as: The worst case total time $T(n)$ divided by the number of operations *i.e.*, $T(n)/n$. Note: This is different from usual notion of "average case analysis".

Here we do not take any assumptions about inputs being chosen at random.

Let us again take the example of binary counter

000000	during counting:
000001	Bit 0 flips every time
000010	Bit 1 flips every other time
000011	So we can conclude that
000100	For bit k, it flips every $2k$ time.
000101	
000110	
000111	

So total bits flipped in N operations (sums), when the counter counts from 1 to N will be :

$$T(N) = \Sigma\ k = 0 \log (N)\ N/2k < N\ \Sigma k = 0\ \infty\ 1/2k = 2N.$$

So the amortized cost will be T(N)/N = 2.

Amortized Time Analysis

The name amortized comes from business accounting practice of spreading a large cost, which was actually incurred in a single time period, over multiple time periods that are related to the reason for incurring the cost. In case of algorithm analysis. Large cost of one operation is spread out over many operations, where the others are less expensive.

Techniques used for amortized analysis

- Aggregate method
- Accounting method
- Potential method

Aggregate Method

Example. Stack operations

Push (S, x): Pushes object x into the stack S.

Pop (S): Pops the top of the stack S and returns the popped object.

Multipop (S, k): Removes the k top objects of the stack S or pops the entire stack if it contains less than k objects.

Time for each operation:

Since push and pop operations runs in O(1) time, let us consider the cost of each to be 1. What is running time of multipop (S, k) on a stack of s objects? The actual running time is linear in number of pop operations actually executed multipop (S, k).

1. While not stackempty (S) and $k \neq 0$
2. do pop (S)
3. $k \leftarrow k - 1$

Thus total cost of multipop (S, k) in a stack of s objects is min (S, k)

Analysis of a Sequence of Operations

A sequence of n push, pop and multipop operations on an initially empty stack. The worst case cost of a multipop in the sequence is O(n) (since the stack is atmost n).

The worst case time of any stack operation is therefore O(n). Hence a sequence of n operations costs O(n2).

Aggregate Method of Amortized Analysis

Each object in the stack can be popped at most once of each time it is pushed. Therefore the number of times that pop can be called on a nonempty stack including calls in multipop, is atmost the number of push operations which is at most n. For any value of n, any sequence of n push, pop and multipop operations takes a total of $O(n)$ time.

The amortized cost of an operation is:

$$O(n)/n = O(1)$$

Accounting Method

In this method we assign different charges to different operations of the sequence. While in aggregate method same amortized cost is assigned to each operation. With operations we assign more or less than the actual cost, (called amortized cost). When the amortized cost of the operation exceeds its actual cost, the difference is assigned to specific object as credit. This credit is used later on to pay for operations whose amortized cost is less than the actual cost.

Amortized Costs

The total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence (all possible sequences of operations). *Thus the total credit associated with the data structure must be non negative at all times.*

Example. In the example of counting, we know that there is only one bit which flips from 0 to 1. If we pay 2 rupees for each (0 – 1) flip by paying one for the operation and "pinning" one rupee (as credit) on that bit. We can use this pinned rupee to pay the followed (1 – 0) flip without paying any extra money for that operation.

Observation

In the counting 000000...000001 ... 000010 ... 000011

000100...000101 ... 000110 ... 000111

1. A number of bits changed from 1 to 0
2. Exactly one bit changed from 0 to 1.

So at any step counting we have enough money (as credit) to flip all the bits from 1 to 0. For flipping the only bit from 0 to 1 we are charging fresh Rs. 2. So at any step total money left after paying for the work is always non negative.

Potential Method

Potential method work as prepaid work as potential energy that performs further computation in punture. It used in data structure for specific job inspite whole program *i.e.,* performs like an object of program.

Following stack operation performed by potential method.

- Push
- Pop
- Multipop
- Potential function ϕ on the stack work as the number of element it has, we assume initially stack D is empty then

$$\phi(D_0) = 0 \quad \text{and} \quad \phi(D_i) \geq 0 \text{ for all } i$$

NOTES

Push operation

To insert i^{th} element on stack D. Then following function work as

$$\phi(D_j) - \phi(D_{j-1}) = \phi(D_j) + 1 - \phi(D_{j-1}) + 1$$

and cost to push an element is

$$C_j = C_j + [\phi(D_j)] - \phi(D_{j-1})]$$
$$= 1 + 1$$
$$= 2.$$

Multipop operation

Assume that j^{th} operation on the stack is multipop (D, i) and

$$m = \min(i, S)$$

Objects are popped from the stack. The actual cost of the operation is m and potential difference are:

$$\phi(D_j) - \phi(D_{j-1}) = -m$$

Cost for multipop:

$$C_j = C_j + \phi(D_j) - \phi(D_{j-1})$$
$$= m - m$$
$$= 0$$

As we can calculated the amortized cost of an ordinary pop operation is 0.

Potential method

Actually "pinning rupees in both the examples was just demo of potential method. In this analysis with potential, we define a potential function $\phi(D_i)$ to represent the potential energy after the it operation. Here D_i is the resulting data structure after the it operation. Therefore the amortized cost C* I of the ith operation with respect to the potential function is defined by: $C^*i = C_i + \varphi(D_i) - \varphi(D_{i-1})$

And the total amortized cost is

$$\Sigma i = \ln C^*i = \Sigma i = \ln (C_i + \varphi(D_i) - \varphi(D_{i-1}))$$
$$= \varphi(D_n - D_{n-1}) + \Sigma i = \ln C_i$$
$$\Sigma i = \ln C^*I = -(\phi(D_n - D_{n-1})) + \Sigma i = \ln C^*I$$

If $\varphi(D_n) \geq \varphi(D_0)$

We have $\Sigma i = \ln C_i \leq \Sigma i = \ln C^*i$

and so $\Sigma i = \ln C^*i$ can be used as upper bound for the total cost.

So suitable potential function should be such that it is always non-negative.

Example 1. Stack operation we define the potential energy as follows:

$\phi(S)$ = number of items in the stack

So we have

$\phi(S_0) = 0$ and $\phi(S_n) = 0$ (if assumed at the end stack again becomes empty)

Also $\phi(S_i) \geq 0$ for all i. Which meets the non negative requirement for potential function.

Stack operation. We define the potential energy as follows:

$\phi(S)$ = number of items in the stack

So we have

$\phi(S_0) = 0$ and $\phi(S_n) = 0$ (if assumed that at the end stack again becomes empty)

Also $\phi(S_i) \geq 0$ for all i, which meets the non-negative requirement for potential function.

Example 2.

Binary Counter. Here we take the potential as the number of 1' s in that binary number.

For and i^{th} counting step. Let the first k bits from the right is 1 and $(k + 1)^{th}$ bit is 0. So we have:

$$C^* = C + \Delta\varphi = (k + 1) + (1 - k) = 2$$
$$\Sigma N C < \Sigma N C^* = 2N$$

So the amortized cost is bounded by $2N/N = 2$.

3.5 BRANCH AND BOUND

Branch and Bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinational optimization. Branch and bound is a general optimization technique that applies where the greedy method and dynamic programming fails.

The method was first proposed by **A.H. Land and A.G. Doig** in 1960 for linear programming.

We assume that the goal is to find the minimum value of a function $f(x)$ (*e.g.*, the cost of manufacturing a certain product), where x ranges over some sest S of admissible or candidate solutions (the search space or feasible region).

A branch-and-bound procedure requires two tools. The first one is a splitting procedure that, given a set S of candidates, returns two or more smaller sets S_1, S_2 whose union covers S′. Note that the minimum of $f(x)$ over S is min $\{v_1, v_2, ...\}$, where each v_i is the minimum of $f(x)$ within S_i. This step is called branching, since its recursive application defines a tree structure (the search tree) whose nodes are the subset of S.

Another tool is *a* procedure that computes upper and lower bounds for the minimum value of $f(x)$ within a given subset S. This step is called **bounding**.

The key idea of the BB algorithm is: if the lower bound for some tree node (set of candidates) A is greater than the upper bound for some other node B, then A may be safely discarded from the search. This step is called **pruning**, and is usually implemented by maintaining a global variable in that records the minimum upper bound seen among all subregions examined so far. Any node whose lower bound is greater than m can be discarded. The recursion stops when the current candidate set S is reduced to a single element; or also when the upper bound for set S matches the lower bound. Either way, any element of S will be a minimum of the function within S.

3.6 APPLICATIONS

This approach is used for a number of NP-hand problems such as:

- Knapsack problem
- Integer programming
- Travelling salesman problem (TSP)
- Quadratic assignment problem (QAP)
- Non-linear programming
- Maximum satisfiability problem (MAX-SAT).

NOTES

3.7 KNAPSACK PROBLEM

The knapsack problem is a problem in combinational optimization. It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip. Given a set of items, each with a cost and a value, determine the number of each item to include in a collection so that the total cost is less than a given limit and the total value is as large as possible.

Suppose we have n kinds of items, 1 through n. Each item j has a value p_j and a weight ω_j. The maximum weight that we can carry in the bag is c. The 0 – 1 knapsack is a special case of the original knapsack problem in which each item of input cannot be subdivided to fill a container in which that input partially fits.

The 0 – 1 knapsack problem restricts the number of each kind of item, x_j, to 0 to 1.

Mathematically the 0 – 1 knapsack problem can be formulated as

$$\text{Maximize } \sum_{j=1}^{n} p_j x_j$$

$$\text{Subject to } \sum_{j=1}^{n} w_j x_j \leq c, x_i = 0 \text{ or } 1, j = 1, \ldots\ldots, n$$

The bounded knapsack problem restricts the number of each item to a specific value. Mathematically, the bounded knapsack problem can be formulated as:

$$\text{Maximize } \sum_{j=1}^{n} p_j x_j$$

$$\text{Subject to } \sum_{j=1}^{n} w_j x_j \leq c_1\ 0 \leq x_j \leq b_j, j = 1, \ldots\ldots, n.$$

The unbounded knapsack problem places no bounds on the number of each item. Of particular interest is the special case of the problem with these properties:

- It is a decision problem.
- It is a 0/1 problem
- For each item, the cost equals the value: $\overline{c = v}$.

3.8 INTEGER PROGRAMMING

If the unknown variables are all required to be integers, then the problem is called an integer programming (IP) or integer linear programming (ILP) problem. In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in many practical situations (those with bounded variables) NP-hard, 0-1 integer programming or binary integer programming (BIP) is the special case of integer programming where variables are required to be 0 or 1. This problem is also classified as Np-hard.

3.9 QUADRATIC ASSIGNMENT PROBLEM

The quadratic assignment problem (QAP) is one of fundamental combinational optimization problems in the branch of optimization or operations research in Mathematics, from the category of the facilities location problems.

The problem models the following real-life problem:

There are a set of n facilities and a set of n locations for each pair of location, a distance is specified and for each pair of facilities a weight or flow is specified. The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows.

3.10 MAXIMUM SATISFIABILITY PROBLEM (MAX-SAT)

The maximum satisfiability problem (MAX-SAT) asks for the maximum number of clauses which can be satisfied by any assignment. It is an NP-hard problem.

There are several extensions to MAX-SAT;

- The weighted maximum satisfiability problem asks for the maximum weight which can be satisfied by any assignment, given a set of weighted clauses.
- The partial maximum satisfiability problem (PMAX-SAT) asks for the maximum number of clauses which can be satisfied by any assignment of a given subset of clauses. The rest of the clauses must be satisfied.
- The soft satisfiability problem (Soft-SAT), given a set of SAT problems asks for the maximum number of sets which can be satisfied by any assignment.

STUDENT ACTIVITY

1. Explain the technique of amortized analysis.

2. What is branch and bound?

NOTES

SUMMARY

1. 8-queen problem is similar to 8 × 8 chessboard where no two queen are placed at attacking position.
2. A hamiltonian cycle is a round trip path along n-edge of G(V, E) where V is vertex and E is an edge.
3. Greedy method provide optimal result, and it is also provide fast and non-optimal solution, which are feasible.
4. Knapsack problem provide the optimal feasible solution of given problem, where condition must be satisfies *i.e.*, total sum of weight with multiplication of selected n_i will not exceed the knapsack limit, then we calculate the maximum profit in the n_i value.
5. Dynamic coding is done using variable size code. This method is based on the principle of identifying the symbols which appear frequently.
6. Potential method work as prepaid work as potential energy that performs further computation in punture. It used in data structure for specific job inspite whole program *i.e.,* performs like an object of program.
7. Branch and Bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinational optimization. Branch and bound is a general optimization technique that applies where the greedy method and dynamic programming fails.
8. The quadratic assignment problem (QAP) is one of fundamental combinational optimization problems in the branch of optimization or operations research in Mathematics, from the category of the facilities location problems.

GLOSSARY

- *Dynamic programming:* Dynamic programming is an algorithm for design method that can be used when solution can be viewed decision sequence of result.
- *Backtracking:* It can be described as an organized exhaustive search which often avoids searching all possibilities.
- *Solution space:* It is the set of all tuples that satisfy the explicit constraints.
- *State space:* The solution space is organized as a tree called state space tree.
- *Live node:* A node which has been generated and all of whose children have not been generated is called a live node.
- *Dead node:* It is a generated node which is not to be expanded further or all of whose children have been generated.
- *E-node:* The live node whose children are currently being generated is called E-node.
- *Bounding function:* Is a function created which is used to kill live nodes without generating all its children.

NOTES

REVIEW QUESTIONS

1. Explain about dynamic programming.
2. Explain application of back tracking.
3. Find all possible solution for 8-queens problem.
4. Discuss about back tracking technique.
5. State the objective of carrying amortized analysis. Discuss with a suitable example.
6. Write short notes on:

   (*a*) Knapsack problem

   (*b*) Travelling sales person problem.
7. Consider three items along with their respect weights and ponds (values) as $I = \{I_1, I_2, I_3\}$, $w = \{6, 5, 4\}$, $p = \{7, 6, 5\}$.

   the knapsack has the maximum capacity $w = 9$, we have to pick this knapsack using the branch and bound technique so as to give the maximum possible value while considering all constraints.

FURTHER READINGS

- Gyanendra Kumar Dwivedi, '*Analysis and Design of Algorithm*', University Science Press.
- Sachin Dev Goyal, '*Design and Analysis of Algorithm*', University Science Press.

NOTES

UNIT

4

GRAPH ALGORITHMS

STRUCTURE

4.0 Objectives
4.1 Graph
4.2 Breadth First Search (BFS)
4.3 Depth First Search (DFS)
4.4 Spanning Tree
4.5 Minimum Spanning Graph (MSG)
4.6 Single Source Shortest Path
4.7 All Pairs Shortest Paths
4.8 The Floyd-Warshall Algorithm
4.9 Transitive Closure
4.10 Johnson's Algorithm
4.11 Maximum Flow
4.12 Max-Flow Min-Cut Theorem
4.13 The Ford-Fulkerson Method
4.14 The EDMONDS-KARP Algorithm
- *Summary*
- *Glossary*
- *Review Questions*

4.0 OBJECTIVES

After going through this unit, you will be able to:

- explain graph algorithms
- discuss about breadth and depth first search
- describe the minimum spanning tree
- define Kruskal's and Prism's algorithms
- describe the types of single source shortest path
- define maximum flow.

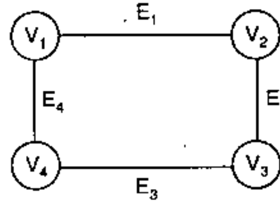
4.1 GRAPH

A graph G (V_1, E) consist of non empty set of vertices (V). Where E are edge and V are vertices. A graph G (V_1, E) consist of:

NOTES

(1) Set of objects $V = \{V_1, V_2, V_3 \dots V_n\}$. Whose elements are called vertex, nodes or vertices of graph (G).

(2) A set of edge $E = \{E_1, E_2, \dots E_n\}$ of unordered pair of distinct vertices and this is called edge of G such that each edge E_i is defined by pair (V_i, V_{i+1}) of vertex. Example of the graph G (V, E) is:



Graph are mainly of following kinds:

- Sparse graph
- Dense graph
- Undirected graph
- Directed graph
- Incident graph
- Complete graph
- Null graph
- Adjacency graph
- Disjoint graph
- Sub-graph
- Euler graph
- Weighted graph
- Regular graph etc.

Sparse Graph

A sparse graph is presentation of adjacency list.

Dense Graph

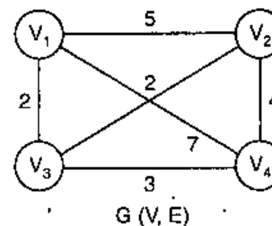
A dense graph is presentation of adjacency matrix.

Undirected Graph

Unordered pair of vertices is called undirected edge and graph G (V, E) is called undirected graph. If, graph possess undirected edge.

Undirected edge can be viewed as two way edge.

Example of undirected graph G (V, E)



Graph G (V, E) is undirected graph. Where cost of each edge are:

$$V_1 \overset{5}{—} V_2, \quad V_2 \overset{5}{—} V_1, \quad V_1 \overset{2}{—} V_3, \quad V_3 \overset{2}{—} V_1$$
$$V_1 \overset{2}{—} V_4, \quad V_4 \overset{2}{—} V_1$$

Similarly

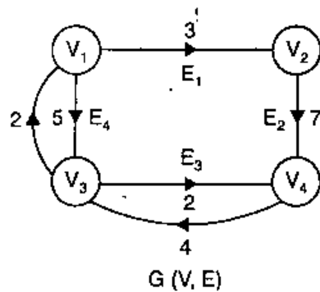
$$V_2 \overset{4}{—} V_4, \quad V_4 \overset{4}{—} V_2, \quad V_4 \overset{3}{—} V_3, \quad V_3 \overset{3}{—} V_4$$
$$V_2 \overset{2}{—} V_3, \quad V_3 \overset{2}{—} V_2$$

Directed Graph

Directed graph is 'one way' graph where cost of one node to another are not necessary to equal to *vice versa.*

i.e., if cost of $V_1 - V_3 = 5$

but cost from $V_3 - V_1 = 2$



G (V, E)

Hence "A graph G (V, E) with an edge which is associated an ordered pair of V × V is called a directed graph edge and such graph G (V, E) which consist directed edge is called directed graph".

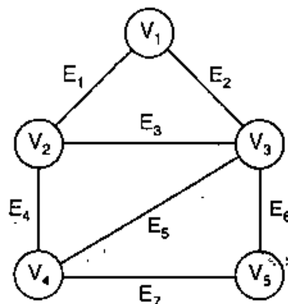
In above graph G (V, E):

$$E_1 = \{V_1\ V_2\} \neq \{V_2, V_1\}$$
$$E_2 = \{V_2, V_4\} \neq \{V_4, V_2\}$$
$$E_3 = \{V_3, V_4\} \neq \{V_4, V_3\}$$
$$E_4 = \{V_1, V_3\} \neq \{V_3, V_1\}.$$

Incident Graph

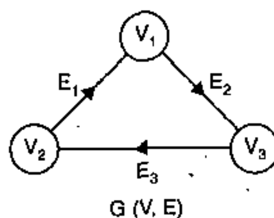
If graph G (V, E) on directed or undirected graph and edge $E_i \in E$ which joins the vertex V_i and V_{i+1} then edge E_i is said to incident edge at vertex V_i and V_{i+1}, and such type of graph is called incident graph G (V, E).

Example:



NOTES

- Then graph G (V, E) for edge E_1, E_2 is incident in to V_1 and V_2 vertex above graph is undirected graph.



G (V, E)

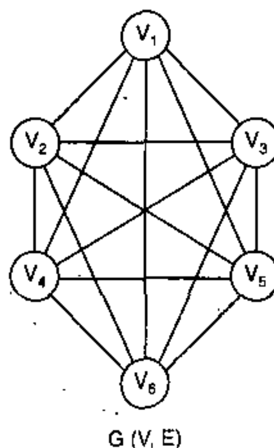
Where E is incident on V_2 and V_1.

Complete Graph

A graph G (V, E) is called complete graph if every vertex is adjacent to all other vertex of the graph G (V, E).

i.e., all vertex are connected to each other and no one is isolated.

Example:



G (V, E)

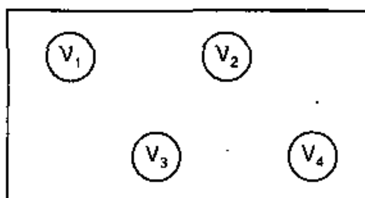
- In graph G (V, E), each and every node are connected to all other's vertex.
- This graph is similar to mesh topology.

Null Graph

A graph G (V, E) is called null graph if it does not possess any edge. *i.e.*, every vertex are isolated no two vertex are connected.

Example:

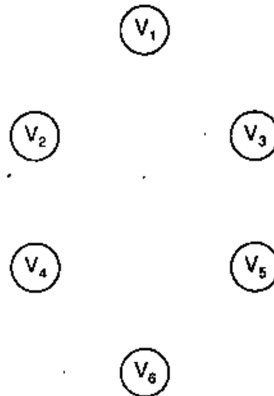
1. Let G (V, E) be an null graph then graph can be expressed as below:



No one edge are connected.

NOTES

2.



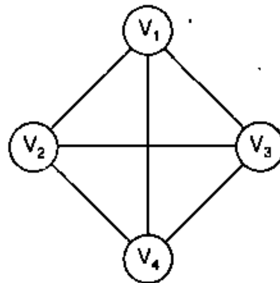
At example (2) every vertex are isolated. Hence it is Null graph.

Adjacency Matrix

The adjacency graph $A = [a_{mk}]_{n \times n}$ of graph G is $n \times n$ binary matrix, where n are the number of vertex with no parallel edges, and number of adjacency graph 'A' can be represented as:

$$a_{mk} = 1 \text{ if } m, k \text{ forms an edge}$$
$$= 0 \text{ if } m, k \text{ does not form an edge.}$$

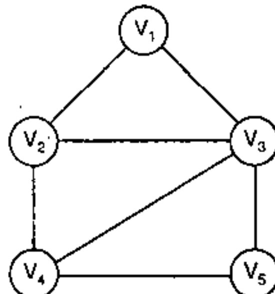
Example 1. *If graph G (V, E) be given below then make adjacency matrix.*



Sol. Adjacency matrix

$$A = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & V_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Example 2. *Then construct adjacency matrix A.*



Then construct adjacency matrix A.

Sol.

$$A = \begin{array}{c} \\ V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{array} \begin{array}{c} \begin{array}{ccccc} V_1 & V_2 & V_3 & V_4 & V_5 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

NOTES

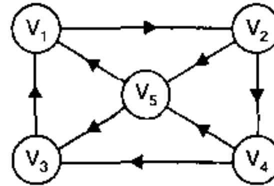
When 2 vertex are directly connected then we simply placed 1 at that corresponding position at matrix. Otherwise place 0.

Hamiltonian path and Hamiltonian circuit. The path obtained by removing anyone edge from a Hamiltonian circuit is called a Hamiltonian path. Hamiltonian path is a graph G (V, E) visit every vertex of graph.

Hamiltonian path is a subgraph of Hamiltonian circuit. So we conclude that Hamiltonian circuit always consist Hamiltonian path.

Hamiltonian circuit is a closed walk in which each vertex is traversed only once except the starting state it is similar to travelling sales person problem.

Example:



Hamiltonian circuit with Hamiltonian path.

Disjoint Graph

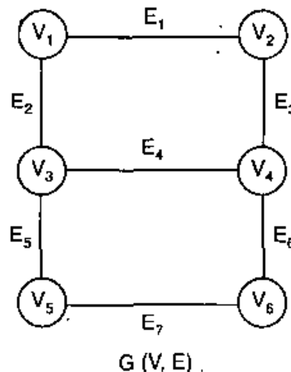
If a given graph G (V, E) have 2 sub-graph G_1 (V_1, E_1) and G_2 (V_2, E_2) and vertex of G_1 and G_2 are distinct *i.e.*,

$V(G_1) \cap V(G_2) = \phi$ then such vertex are called disjoint vertex and if

$E(G_1) \cap E(G_2) = \phi$ then such edges are called disjoint edge of the graph G (V, E).

Hence a graph is called disjoint graph if it contains disjoint vertex and disjoint edge.

Example. *If graph G (V, E) is an graph with G_1 (V_1, E_1) and G_2 (V_2, E_2) sub-graph then constant disjoint graph of graph G (V, E) given below.*



Sol. Graph G (V, E) can be divided in to 2 sub-graph $G_1 (V_1, E_1)$ and $G_2 (V_2, E_2)$

and $G(V_1) \cap G(V_2) = \phi$

and $G(E_1) \cap G(E_2) = \phi$

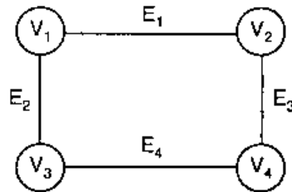
so $G(V_1) = \{V_1, V_2, V_3, V_4\}$

$G(E) = \{E_1, E_2, E_3, E_4\}$

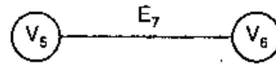
$G(V_2) = \{V_5, V_6\}$

$G(E) = \{E_7\}$

$G_1 (V_1, E_1) =$



and graph $G_2 (V_2, E_2) =$



NOTES

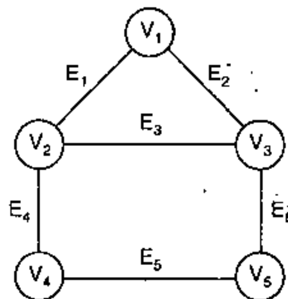
Sub-graph

Consider a graph G (V, E). A graph $G_1 (V_1, E_1)$ is called sub-graph when

$$V_i \subset V_j \quad \forall V_i \in G \quad \text{and} \quad V_j \in G_1$$
$$E_i \subset E_j \quad \forall E_i \in G \quad \text{and} \quad E_j \in G_1$$

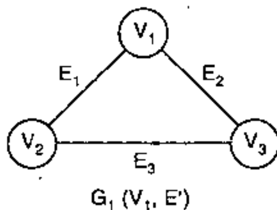
Sub-graph of a graph G (V, E) is similar to sub-set of set.

Example:

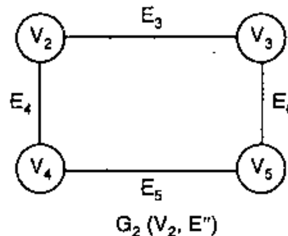


Let G (V, E) be a graph where $V = \{V_1, V_2, V_3 V_4, V_5\}$ and $E = \{E_1, E_2, E_3, E_4, E_5, E_6\}$. Find sub-graph of graph G (V, E).

Sol. Graph $G_1 (V_1', E')$ and graph $G_2 (V_2'', E'')$ are 2 sub-graph of graph G (V, E)



$G_1 (V_1, E')$



$G_2 (V_2, E'')$

where $V_1' = \{V_1, V_2, V_3\}$ $\quad V_2' = \{V_2, V_3, V_4, V_5\}$

$E' = \{E_1, E_2, E_3\}$ $\quad E'' = \{E_3, E_4, E_5, E_6\}$

The graph G_1 and G_2 are 2 sub-graph of Graph G.

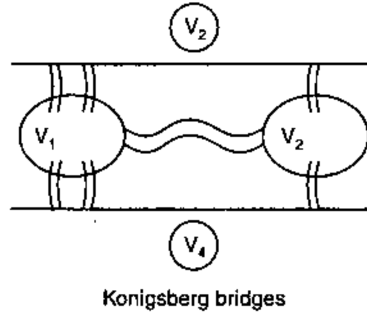
NOTES

Euler Graph

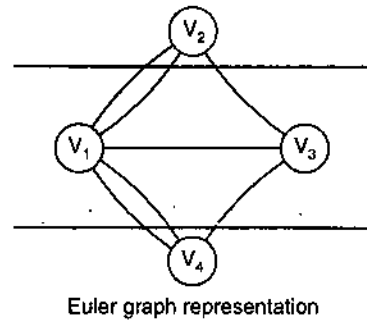
Euler gives the concept of closed walk running through every edge of graph G (V, E) exactly once, this walk is called Euler line and graph that consist Euler line is called Euler graph.

i.e., if some closed (path) walk in a graph contains all the graph (line) then walk is called Euler line and graph will be Euler graph.

Eulerian circuit in a graph to be a circuit that traverses each edge in the graph once and only once.



Konigsberg bridges

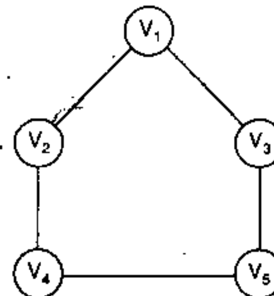


Euler graph representation

Complement Graph

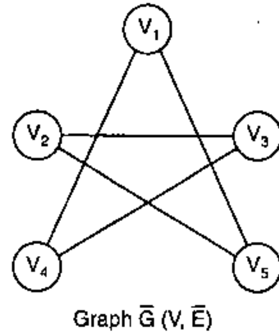
The complement graph $\overline{G}$ of simple graph G (V, E) is simple graph with vertex set V (G) defined by $V_i, V_j \in E(\overline{G})$ iff $V_i, V_j \notin E(G)$.

Example: If a graph G (V, E)



is given then complement graph of graph G (V, E)

NOTES



Graph $\bar{G}$ (V, $\bar{E}$)

In complement graph all connected edge of graph G (V, E) becomes disconnected and disconnected edge become connected.

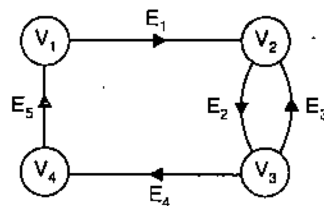
Degree of Graph G (V, E)

In directed graph G (V, E) for any vertex V_i the number of edge which have V_i as the initial vertex is called '**out degree**' of vertex V_i denoted by $d^+ (V_i)$. The number of edge which have V_i as their terminal vertex is called the "**indegree**" of V_i denoted by $d^* (V_i)$. Sum of the out degree and the indegree of vertex n is called its total degree or degree of the graph.

We can say that the number of edges incident on a vertex V_i, with self-loops counted twice is called the degree, d (V_i) of vertex V_i, in case of undirected graph, the degree of a vertex V_i is equal to the number of edges incident with V_i.

The degree of a vertex as sometimes also referred to as *valency*. A vertex V with zero indegree is called a source and a vertex V with out degree is called sink.

Example:



Degree of V_1 $\quad d\,(V_1) = 2$

$d\,(V_2) = 3$

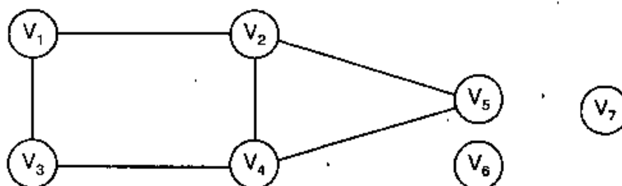
$d\,(V_3) = 3$

$d\,(V_4) = 2$

Isolated Graph

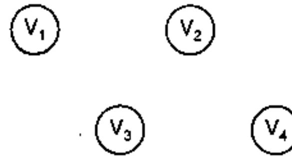
A vertex of degree zero is called an isolated vertex, *i.e.*, vertex having no incident edge is called an isolated vertex. Vertex V_6 and V_7 are isolated vertex and graph is isolated graph.

Example:



NOTES

While null-graph containing only isolated vertex *i.e.*, there will be no edge.



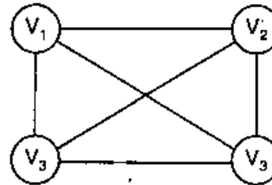
G (V) called null-graph.

Regular Graph

A graph in which all vertices are of equal degree is called a regular graph or simply regular. A graph G is regular of degree n or n-regular if every vertex has degree n.

3-regular graph is called a cubic graph. The connected 0-regular graph is trivial graph with one vertex and no edges.

Example:

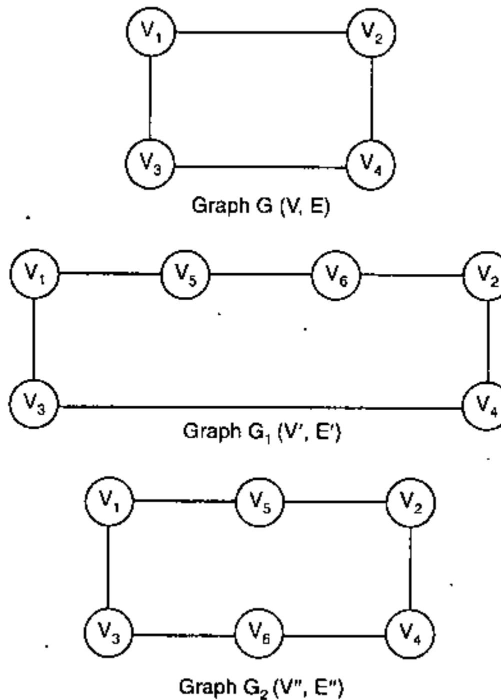


Graph G (V, E) with degree 3 of each vertex is called cubic graph.

Homeomorphic Graph

A graph by dividing an edge of a given graph G with additional vertex two graph G_1 and G_2 are said to be homeomorphic if they can be obtained from the same graph or isomorphic graph.

Example:



Graph G (V, E)

Graph G_1 (V′, E′)

Graph G_2 (V″, E″)

Graph G_1 and G_2 are homeomorphic but they are not isomorphic.

Isomorphic Graph (Isomorphism)

Two graph G (V, E) and G′ (V′, E′) are thought of as equivalent and called isomorphic if they have identical behaviour in term of graph-theoretic properties.

i.e., Two graph G (V, E) and G′ (V′, E′) are isomorphic if there exist (∃) a function $f: V(G) \to V(G')$ from vertices of G to the vertices of G′ such that

(*i*) f is one-one (injective)

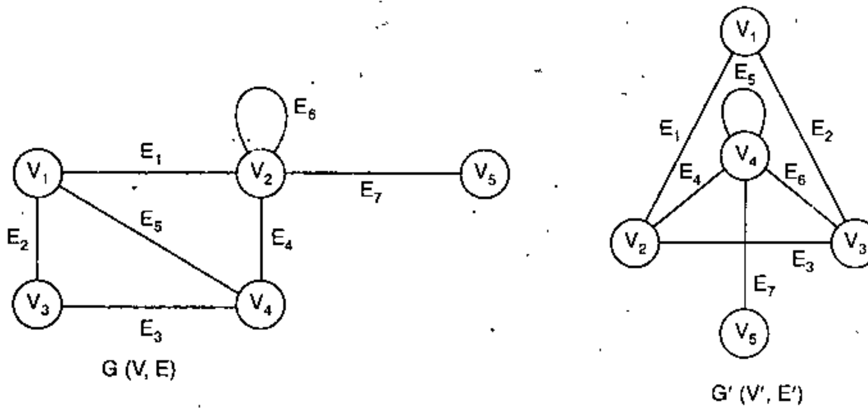
(*ii*) f is onto (surjcetive)

(*iii*) For each of vertices V_i and V_j of G, $\{V_i, V_j\} \in E$ if and only if $\{f(V_i), f(V_j)\} \in E'$.

If above three properties satisfy from $f: G \to G'$ then this properties is called isomorphic graph from G to G′.

The (*iii*) condition say that function and preserves adjacency *i.e.*, V_i and V_j are adjacent in G iff $f(V_i)$ and $f(V_j)$ are adjacent in G.

Example:

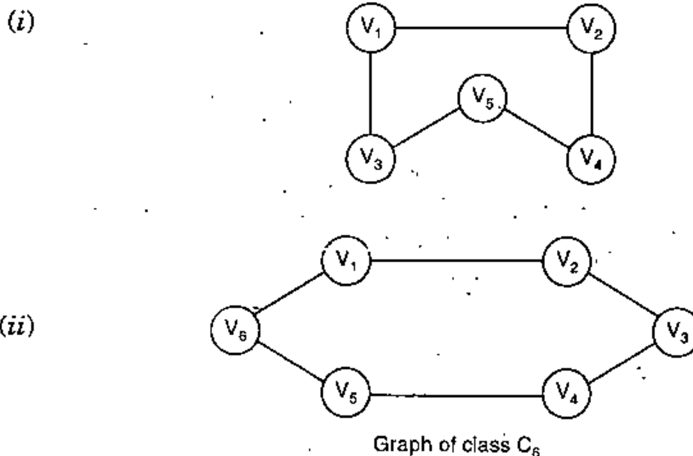


$f: G \to G'$ is called isomorphic graph.

Cycle Graph

A connected graph G (V, E) whose edge from a cycle of length n is called a cycle graph of order n. Cycle graph are denoted by C_n.

Example: A graph G (V, E), with C_5 class



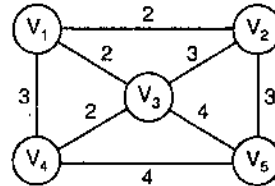
Graph of class C_6

NOTES

Wheel Graph

A wheel of order n is a graph obtained by joining a single new vertex to each vertex of cycle graph of order $(n - 1)$. Wheels of order n are denoted by w_n. A graph of class w_5 is given below.

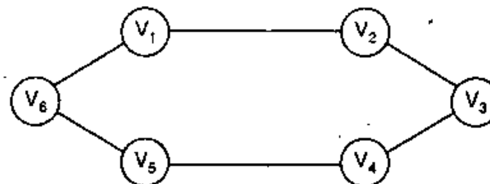
Example:



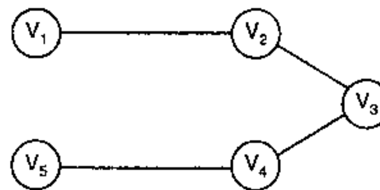
Path Graph

If we remove an edge from C_n graph, A path graph of order n is obtained, A path graph of order n is denoted by P_n.

Example. A graph G (V, E) is cycle graph then we remove one vertex and then it becomes path graph G′ (V′, E′)



Cycle graph G (V, E)



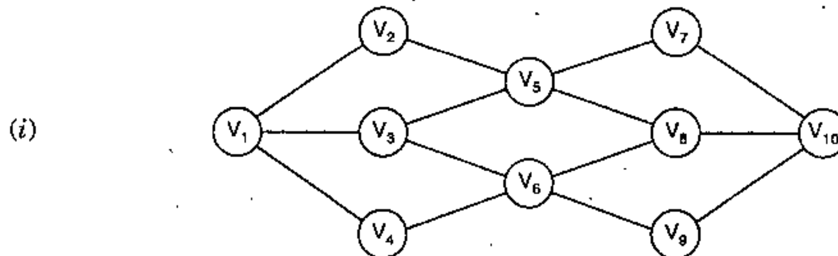
Path graph G′ (V′, E′)

Bipartite Graph

A Bipartite graph is a graph G (V, E), whose set of vertices can be partitioned into two non-empty subsets G_1 and G_2. Such that there is no edge in E joining two vertices in G_1 or two vertices in G_2.

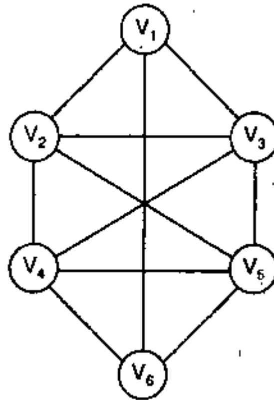
Example:

(*i*)



Above given graph can partitioned in to 2 subset.

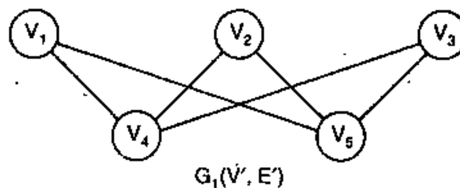
(ii)



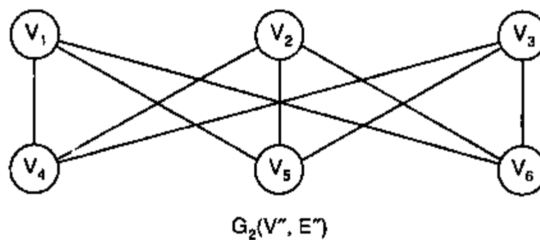
Complete Bipartite Graph

A complete bipartite graph is a bipartite graph in which every vertex of G_1 is adjacent to every vertex of G_2. A complete bipartite graph G (V, E) denoted as $[B]_{m \times n}$ or B_{mn}. If it may be partitioned into set G_1 and G_2 such that the number of vertices of G_1 *i.e.*, $|G_1| = m$ and $|G_2| = n$. The normal order of m and n is such that $m \leq n$. Hence graph B_{mn} has $m.n$ edges.

Example:



G1(V′, E′)



G2(V″, E″)

Graph G_1 and G_2 be bipartite graph of graph G (V, E).

Graph Traversal:

- Breadth first search
- Depth first search

4.2 BREADTH FIRST SEARCH (BFS)

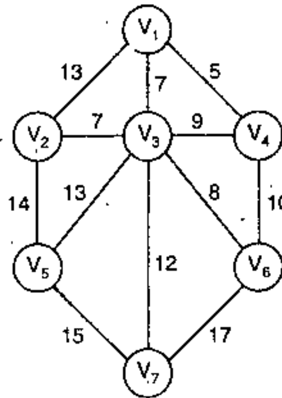
Breadth first search is elementary searching technique of a graph G (V, E), in this technique we visit or search the vertex and a function visit = L enable in respect of that node.

It discover every vertex that is reachable from source 'S' or not. It also computes the distance from S to each reachable vertex. It also produce a "breadth first tree" with root S that contains all reachable vertices. For any vertex v reachable from S, the path

in the breadth first tree from S to v corresponds to a ***shortest path*** from S to v in graph G (V, E) *i.e.,* a path containing the smallest number of edges. The algorithm works on both directed and undirected graph.

Breadth first search maintain a queue, FIFO (First-in-first-out) where the unvisited element are kept initially vertex 'S', the start vertex, is kept in queue. Next, the adjacent vertices are kept in queue after processing the start vertices.

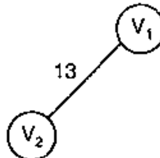
Example of BFS. *If a graph G (V, E) are given below then show how all node's will be discover.*



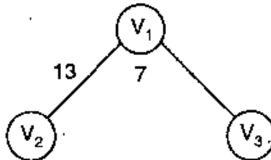
Sol. Graph G (V, E) will be started visiting or discover the vertex from V_1.

Step 1.

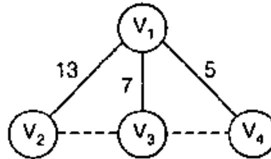
Step 2.



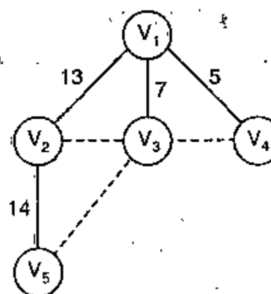
Step 3.



Step 4.

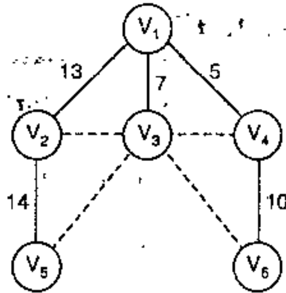


Step 5.

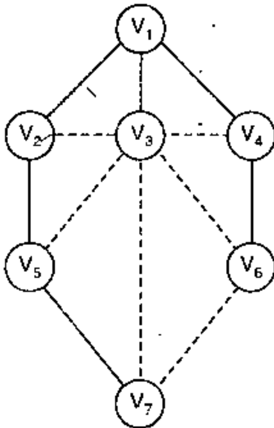


NOTES

Step 6.



Step 7.



Algorithm for BFS

Breadth first search (v)

Step 1. $w \leftarrow v$
visit [v] $\leftarrow$ L
loop
{
For every vertex u adjacent from m.
do
if (visit [u] = 0)
then
{
add u to L
// u is visited
visit [u] $\leftarrow$ 1
}
}
if L is empty then
return
// There is not any unexplored or unvisited node delete w from L
}
until (false) ;
}
Now at vertex v, if any node is visited then visit [node] = 1

4.3 DEPTH FIRST SEARCH (DFS)

NOTES

In depth first search, graph G (V, E), the traversal can be started from any node (S), as from name it search deeply at one side then started the other side with same scenario in DFS, edge are explored out of the most recently discovered vertex u that still has unexplored edge leaving it. When all of u's edges have been explored, the search brack-track to explore edge leaving the vertex from which u was discovered. (This process is cost effective, due to that using back-track process we visited. Those vertex also which had been already visited.) This process continues until we have discovered vertex remain, then one of them is selected as a new source and the search is repeated from that source, this process is repeated up to all vertex discovered.

Algorith For DFS

Depth-first search (u)

Step 1. visit [u] $\leftarrow$ L

For each vertex v adjacent from u

do

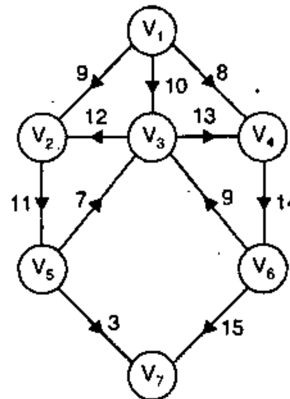
if (visit [v] = = 0)

then

Depth-first search (v).

Note. Graph given is undirected graph G (V, E) with n vertex and array visit [ ] initially set to zero and all vertex must be visited which are reachable from v.

Example of DFS:

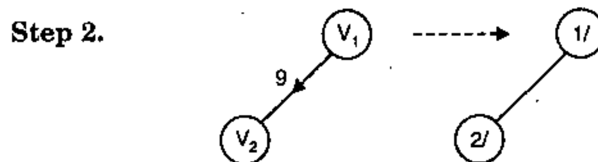


Using DFS technique discover all nodes.

Sol. Assume we are starting visit from node V_1.

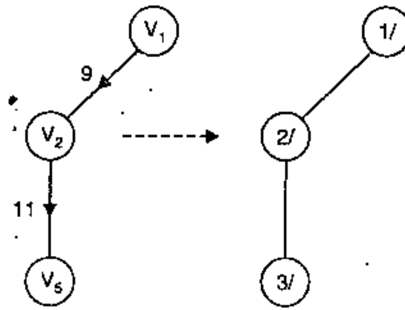
Step 1.

Step 2.

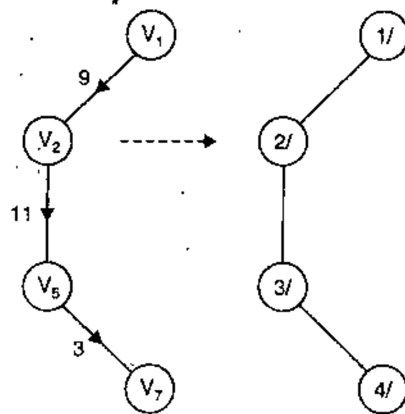


NOTES

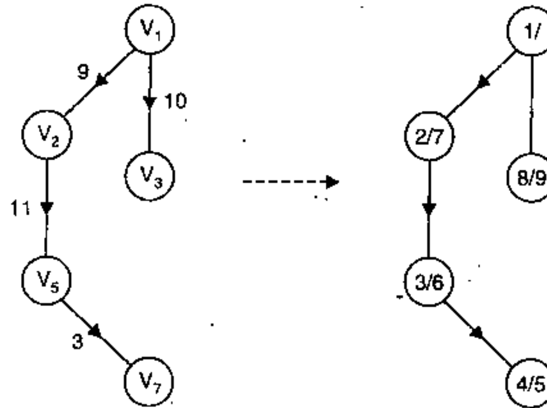
Step 3.



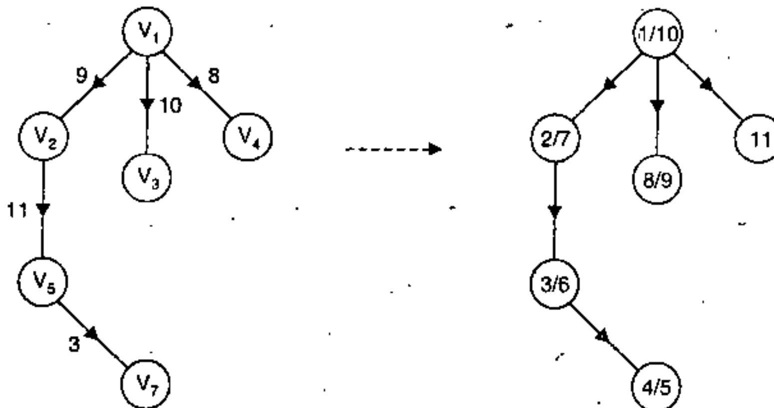
Step 4.



Step 5.

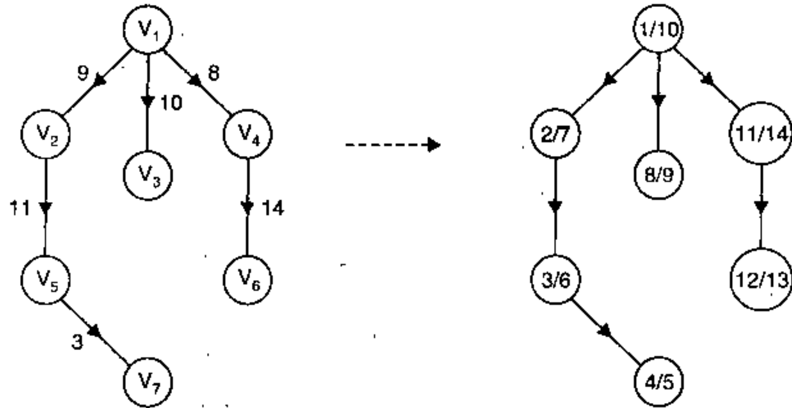


Step 6.



NOTES

Step 7.



4.4 SPANNING TREE

A sub graph S of a connected graph G (V, E) is called a spanning tree of G if

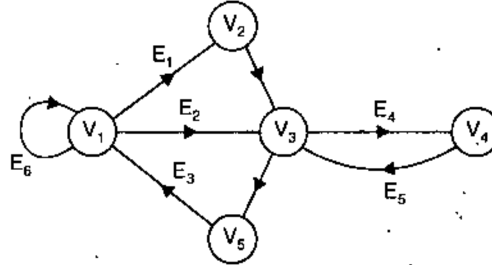
(*i*) S is a tree

(*ii*) S contains all vertices of G.

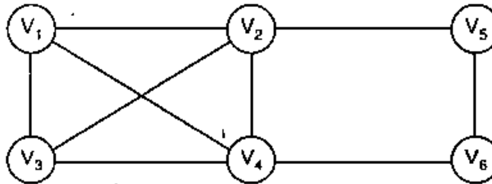
The sub graph in heavy lines in figure given below is the spanning tree of the graph G.

Spanning trees some time also referred to as a skeleton or scaffolding of G.

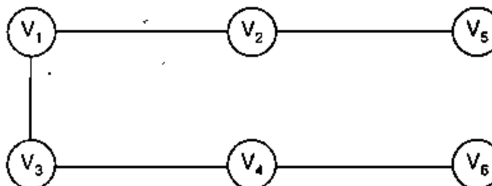
Example:



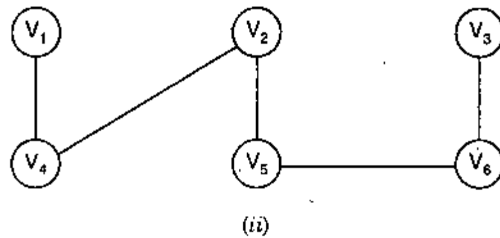
Example. *Draw spanning trees of graph given below.*



Sol. Spanning tree of given graph is:



(*i*)



(ii)

NOTES

4.5 MINIMUM SPANNING GRAPH (MSG)

Minimum spanning graph G (V, E) of weighted graph are sum of weight of each edge, where every vertex are covered with minimum cost. Hence MST are traversel of every vertex with minimum cost. Where graph G (V, E) is undirected graph.

Application of spanning trees arises from property that spanning tree is a minimal sub graph G_1 of G such that V (G_1) = V (G) and G_1 is connected.

Any connected graph with n vertices must have at least $(n - 1)$ edges and all connected graph with $(n - 1)$ edge are trees.

These are 2 algorithm to compute the minimum cost of weighted undirected graph. They are

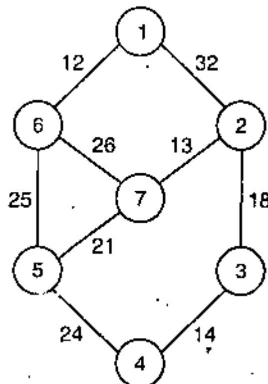
- Kruskal's algorithm
- Jarnik-Prims algorithm

Kruskal's Algorithm

In Kruskal's algorithm the optimization criteria mentioned that edge of graph G (V, E) are considered in non-decreasing (increasing) order of cost.

i.e., we firstly join that edge which have minimum weight (cost) and then other in-increasing order and consider that no look has been created.

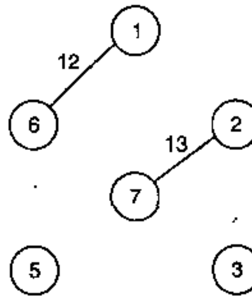
Example of Kruskal's algorithm. *If a graph G (V, E) have following weight then construct Kruskal's minimum spanning tree.*



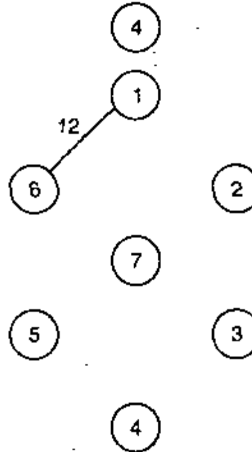
Graph G (V, E) which is undirected graph with cost of each edge.

Sol.

Step 1.

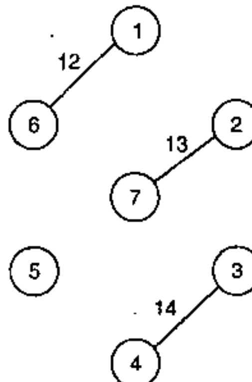


Step 2.

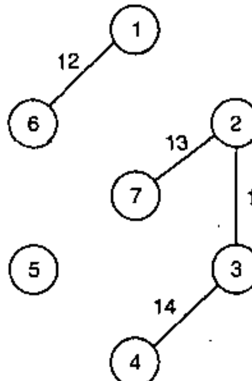


Minimum cost edge will be select firstly so we firstly join (6) —12— (1) then (7) —13— (2) with respective cost.

Step 3.

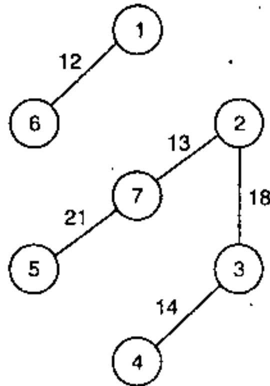


Step 4.

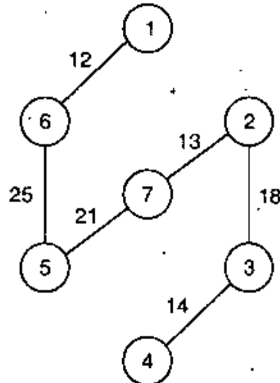


NOTES

Step 5.



Step 6.



Then weight 24 will joint (4)———(5) which forms closed loop so we avoid it, then (6)—25—(5) are connected with cost.

Hence total cost = 12 + 25 + 21 + 13 + 18 + 14 = **103.**

Kruskal's Algorithm

Kruskal's (E, m cost, n, S)

Step 1. Construct the heap of each vertex for $i \leftarrow 1$ to n
do parent [i] $\leftarrow -1$
// For each vertex is in different set
$i \leftarrow 0$
minimum cost $\leftarrow 0$
while (($i < n-1$) and (Heap $\neq$ empty))
do
Delete the edge (V_i, V_{i+1}) with minimum cost and adjust heap again.
$j \leftarrow$ find (V_j)
$k \leftarrow$ find (V_{i+1})
if ($j \neq k$) then
$i \leftarrow i + 1$
S [i, 1] $\leftarrow V_i$
S [i, 2] $\leftarrow V_{i+1}$
Minimum cost $\leftarrow$ minimum cost + cost [V_i, V_{i+1}]
union (j, k)

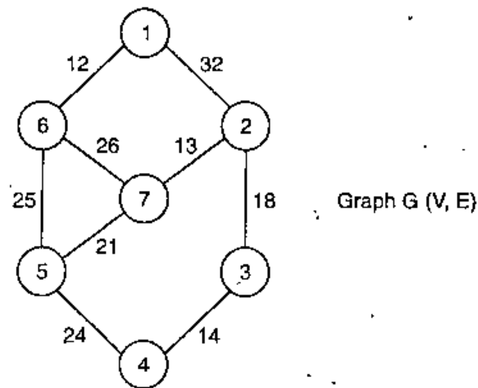
if ($i \neq n - 1$) then write ("No spanning tree")
else
return minimum cost

NOTES

Jarnik-Prims Algorithm

Using Jarnik-Prims algorithm we construct minimum cost spanning tree by adding edge to edge using this method first we select that edge which have minimum cost, then we choose another vertex connected to connected edge with minimum cost, run time of prims algorithm is 0 ((V + E). log V). This algorithm work on principle of Greed method.

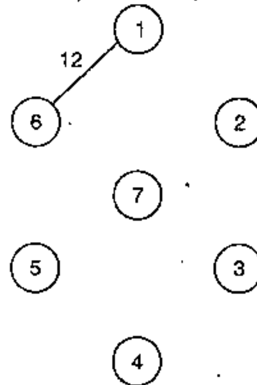
Example of Prims algorithm



Construct the minimum cost spanning tree of graph G (V, E)

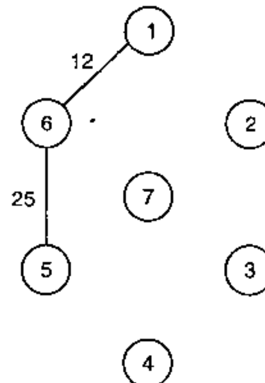
Sol.

Step 1.



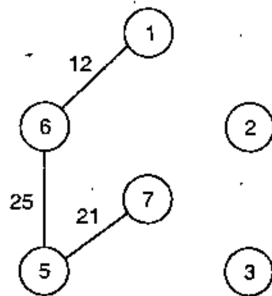
Firstly we add those vertex which have minimum cost then

Step 2.



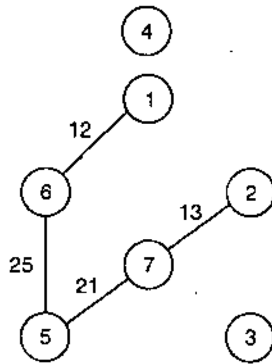
Since (1) —32— (2), (6) —26— (7), (6) —25— (5) the minimum cost is in between (6) —— (5), so we connected it.

Step 3.

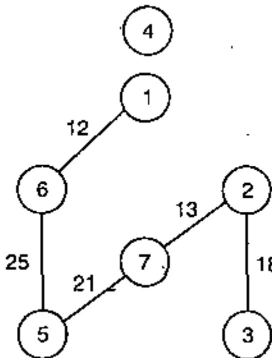


NOTES

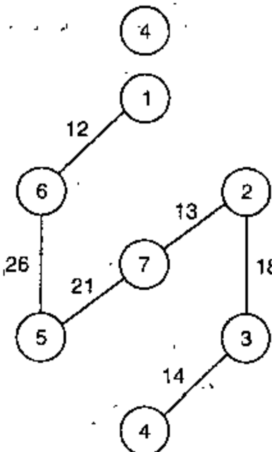
Step 4.



Step 5.



Step 6.



NOTES

When we connected the vertex it should also considered that closed path does not form.

Algorithm of prims

```
Jarnik-prims (E, n, s)
{
Assume (u, v) be an edge of minimum cost in E.
minimum cost ← cost [u, v]
S [1, 1] ← u
S [1, 2] ← v
For i ← L to n
do // Initialization neighbour
if (cost [i, u] < cost [i, v] )
then near [i] ← u
else near [i] ← v
near [v] = near [u] = 0
for i ← 2 to n – 1
do
{
Let j be an index such that near [j] ≠ 0, and
cost [j, near [j]] is minimum.
S [i, 1] ← j
S [i, 2] ← near [j]
Minimum cost = minimum – cost + cost [j, near [j]]
near [j] = 0
for u ← 1 to n
do
if ((near [u] ≠ 0) and cost [u, near [u]] > cost [u, j])
then near [u) ← j
}
return minimum – cost
}
```

4.6 SINGLE SOURCE SHORTEST PATH

Following algorithm are studies under single source shortest path. They are:

- Bellman-Ford algorithm
- Dijkstra's algorithm.

Bellman-Ford Algorithm

According to this algorithm, it solve the problem of single-source shortest path with assuming that weight of edge may be negative. Where weight function of graph G (V, E) are $w : E \rightarrow R$, and E are edge and w are weight boolean value are returned by Bellman-Ford algorithm. If cycle form during the traversal of vertex, then problem has no solution and return value be negative. If traversal done successfully then some positive value will be return *i.e.*, L. Bellman-Ford graph is directed graph.

Example of Bellman-Ford algorithm

Bellman-Ford (G, u, S)

Step 1. Source (G, S)

Step 2. for i = 1 to V [G] – 1

do for each $(m, n) \in$ E (G)

Step 3. do temp (m, n, u)

for each $(m, n) \in$ E (G)

do if $d\ [m] > d\ [n] + u\ (m, n)$

then return false

return true

Source (G, S)

For each vertex $v \in$ V [G]

do $d\ [v] = \infty$

$k\ [v]$ = Nil

d [S] = 0

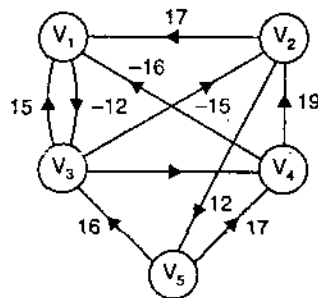
temp (m, n, u)

If $d\ [m] > \text{d}\ [n] + u\ [m, n]$

then $d\ [m] = d\ [n] + 4\ [m, n]$

$k\ [n] = m$.

Example:

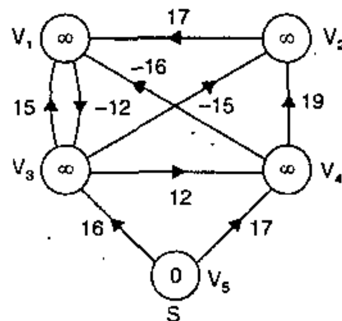


Source (S)

Find the solution of single source path using Bellman-Ford algorithm.

Sol.

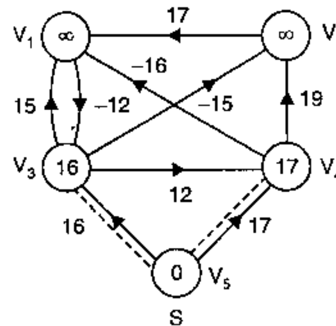
Step 1.



NOTES

We assume V_5 as an source node, so distance from that node to other we assume ∞ and value of that node becomes 0.

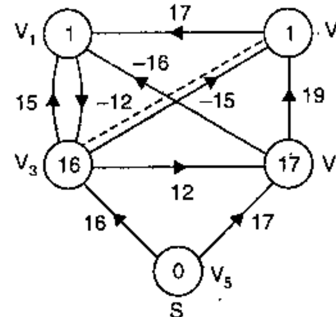
Step 2.



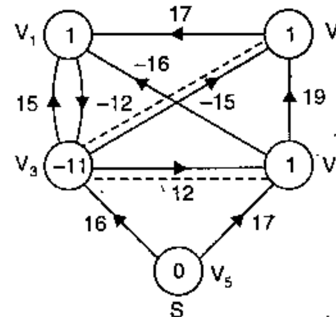
Distance from S to V_3 = 10 and 5 to V_4 = 17

So value of that edge placed their

Step 3.



Step 4.



Dijkstra's Algorithm

This algorithm solves single-source shortest path, and performs repeated relaxation on all the edges of a given graph only once. It start from any vertex with minimum cost of edge and that vertex cost set 0 and rest are set $+\infty$. Then we traverse the rest vertex V – {S} where {S} = {starting vertex} and V = $\{V_1, V_2 \ldots V_i\}$ and S is called source *i.e.*, starting vertex or node where $S \subseteq V$.

Correctness of Dijkstra's algorithm. Dijkstra's algorithm, run on a weighted graph G (V, E) with positive weight. Where weight function are word source {S}, terminate with $d[v] = \delta(S, v)$ for all $V \in V$. This problem include three main functions.

1. *Initialization:* Initialize, $S = \phi$, so invariant is trivially true.

2. *Maintenance:* Do yourself.

3. *Termination:* At termination $m = \phi$. which, along with our earlier invariant that

$$m = V - S, \text{ implies that } S = V$$

Thus $d\,|v| = \delta(S, v)$ for all vertex $v \in V$.

NOTES

Algorithm For Dijkstra's

Dijkstra's (G, w, S)

// G is given graph with vertex V and edge E

// w is weight of 2 connecting vertex

// S is source, where we start the traversing.

Step 1. Source (G, S)

S ← φ // initially we does not select any vertex

m ← V (G)

while $m \neq \phi$

do

v ← Extract – minimum (m)

S ← S ∪ {v}

for each vertex $u \in$ Adj {v}

do set-function (v, u, w)

Step 2. Source (G, S)

For each $u \in$ V (G)

do d [u] ← ∞

x [v] ← Nil

d [s] ← 0

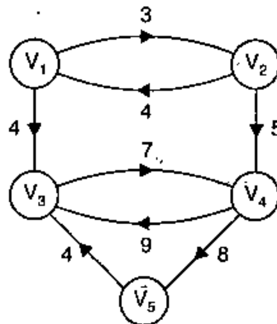
Step 3. Set-function (v, u, w)

do if d [u] > d [v] + w [v, u]

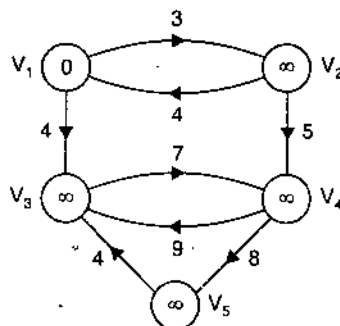
Then d [u] ← d [v] + w [v, u]

x [u] ← v

Example of Dijkstra's algorithm. *Let graph G (V, E) is directed graph with each edge weight. Then find shortest path with minimum cost using Dijkstra's algorithm.*

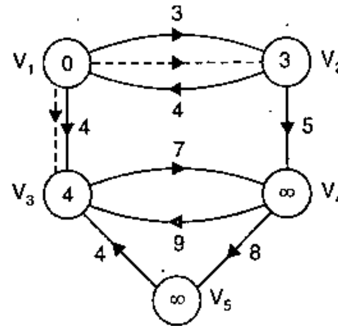


Sol. We can write above graph as:

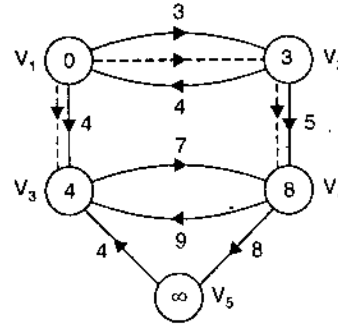


Where starting node will be zero and rest node have ∞ value. We start traverse from V_1 of minimum value and then traverse rest {V – S} edge, where |S| = V_1 and V = {V_1, V_2, V_3, V_4, V_5}.

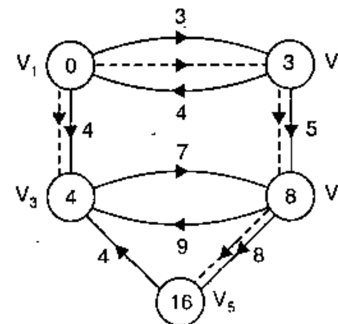
NOTES



From source S = [V_1] we have 2 way $V_1 - V_2$ and $V_1 - V_3$ and cost of the edge are 3 and 4 respective so cost from source have been placed at node.



At node V_4 cost are 4 + 7 and 3 + 5. Since 3 + 5 = 8 is minimum so at node V_4 we placed the value 8.



Hence each and every vertex have been traverse from source S = |V|.

4.7 ALL PAIRS SHORTEST PATHS

In **all pairs shortest path (APSP) problem,** given a directed graph G = (V, E), where each edge (v, w) has a non-negative cost C [v, w], for all pairs of vertices (v, w). Find the lowest cost path from v to w.

- A generalization of the single-source-shortest path problem.
- Use Dijkstra's algorithm, varing the source node among all the node in the graph.

  We will consider a slight extension to this problem: Find the **lowest cost path** between each pair of vertices.
- We must recover the path itself, and not just the cost of the path.

4.8 SHORTEST PATHS AND MATRIX MULTIPLICATION

A dynamic-programming algorithm is used for the all pairs shortest paths problem on a directed graph G = (V, E). Each major loop of the dynamic program will invoke an operation that is similar to matrix multiplication.

Structure of a Shortest Path

We start by characterizing the structure of an optimal solution. Suppose that for all pairs shortest-paths problem on a graph G = (V, E). The graph is represented by an adjacency matrix $W = (\omega_{ij})$. Consider a shortest path P from vertex i to vertex j, and suppose that P contains at most m edges. Assuming that there are no negative-weight cycles, m is finite. If $i = j$, then P has weight 0 and no edges. If vertices i and j are distinct, then we decompose path P into $i \xrightarrow{P'} k \rightarrow j$, where path P′ now contains at most m-1 edges.

Note: P′ is a shortest path from i to k, and so $\delta(i, j) = \delta(i, k) + \omega_{kj}$.

A recursive solution to the all-pairs shortest-paths problem.

Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. When $m = 0$, there is a shortest path from i to j with no edges if and only if $i = j$. Thus,

$$l_{ij}^{(0)} = \begin{cases} 0 & ; \text{ if } i = j \\ \infty & ; \text{ if } i \neq j \end{cases}$$

We compute $l_{ij}^{(m)}$ as the minimum of $l_{ij}^{(m-1)}$ (the weight of the shortest path from i to j consisting of at most m – 1 edges) and the minimum weight of any path from i to j. Consisting of at most m edges, obtained by looking at all possible predecessors k of j; for $m \geq 1$. Thus, we recursively define

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)}, \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\} \right)$$

$$= \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}.$$

What are the actual shortest-path weights $\delta(i, j)$?

If the graph contains no negative-weight cycles, then for every pair of vertices i and j for which $\delta(i, j) < \infty$. There is a shortest path from i to j that is simple and thus contains at most n – 1 edges. A path from vertex i to vertex j with more than n – 1 edges can not have lower weight than a shortest path from i to j. The actual shortest-path weights are given by:

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots\ldots$$

NOTES

Computing the shortest-path weights bottom up.

Taking the matrix $W = (\omega_{ij})$ as input, we compute a series of matrices $L^{(1)}, L^{(2)}, \ldots\ldots, L^{(n-1)}$, where for $m = 1, 2, \ldots\ldots, n-1$, we have $L^{(m)} = \left(l_{ij}^{(m)}\right)$. The final matrix $L^{(n-1)}$ contains the actual shortest path weights. For all vertices $i, j \in V$, when $l_{ij}^{(1)} = \omega_{ij}$ and so $L^{(1)} = \omega$.

The following procedure given matrices $L^{(m-1)}$ and ω, returns the matrix $L^{(m)}$. That is, it extends the shortest paths computed so far by one more edge.

EXTEND-SHORTEST-PATHS (L, W)

1. $n \leftarrow$ row [L]
2. let $L' = \left(l'_{ij}\right)$ be an $n \times n$ matrix
3. For $i \leftarrow 1$ to n
4. do for $j \leftarrow 1$ to n
5. do $l_{ij} \leftarrow \infty$
6. for $k \leftarrow 1$ to n
7. do $l'_{ij} \leftarrow \min\left(l'_{ij}, l_{ik} + w_{kj}\right)$
8. return L′.

Its running time is $\theta(n^3)$.

Now suppose we want to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. Then, for $i, j = 1, 2, \ldots\ldots, n$ we compute

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

MATRIX-MULTIPLY (A, B)

1. $n \leftarrow$ rows [A]
2. c be an $n \times n$ matrix
3. for $i \leftarrow 1$ to n
4. do for $j \leftarrow 1$ to n
5. do $c_{ij} \leftarrow 0$
6. for $k \leftarrow 1$ to n
7. do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
8. return c.

We calculate the shortest path weights by extending shortest paths edge by edge. $A \cdot B$ denote the matrix product returned by EXTEND-SHORTEST-PATHS (A, B). We calculate the sequence of $(n-1)$ matrices.

$$L^{(1)} = L^{(0)} \cdot W = W,$$
$$L^{(2)} = L^{(1)} \cdot W = W^2,$$
$$L^{(3)} = L^{(2)} \cdot W = W^3,$$
$$L^{(n-1)} = L^{(n-2)}, W = W^{n-1}.$$

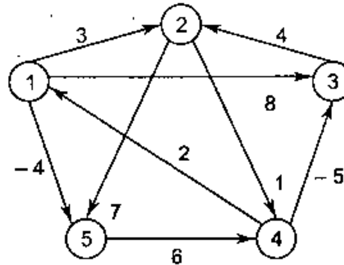
The following procedure calculate this sequence in $\theta(n^4)$ time.

NOTES

SLOW-ALL-PAIRS-SHORTEST-PATHS (W)

1. $n \leftarrow$ rows [W]
2. $L^{(1)} \leftarrow W$
3. For $m \leftarrow 2$ to $n - 1$
4. do $L^{(m)} \leftarrow$ EXTEND-SHORTEST-PATHS $(L^{(m+1)}, W)$
5. return $L^{(n-1)}$

Example. *Given a directed graph G = (V, E) and find out the sequence of matrices $L^{(m)}$ computed by SLOW-ALL-PAIRS-SHORTEST-PATHS.*



Sol. Using SLOW-ALL-PAIRS-SHORTEST-PATHS (W), we compute W.

$$W = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}$$

Step 1. $n \leftarrow$ rows [W] *i.e.*, $n \leftarrow 5$

Step 2. $L^{(1)} \leftarrow W$ *i.e.*,

$$L^{(1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}$$

Step 3. For $m \leftarrow 2$ to $n - 1$, *i.e.*, $m \leftarrow 2$ to $5 - 1$

i.e., $m \leftarrow 2$ to 4, Now $m = 2$.

Step 4. do $L^{(m)} \leftarrow$ EXTEND-SHORTEST-PATHS $(L^{(m-1)}, W)$

i.e., $L^{(2)} \leftarrow$ EXTEND-SHORTEST-PATHS $(L^{(1)}, W)$

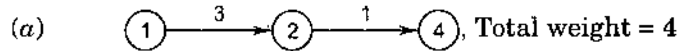
i.e., $L^{(2)} = W \cdot W$.

Note. When we calculate $L^{(1)}$, then we use only one edge to reach one vertex to another the vertex. Now we want to find $L^{(2)}$, then we will use only two edge to reach one vertex to another the vertex. When $L^{(3)}$, then only 3 edge. When $L^{(4)}$, we will use only 4 edge. Then after we choose only smallest weight. In case of $L^{(2)}$,

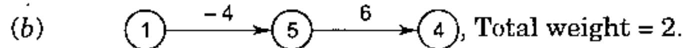
NOTES

$$L^{(2)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 3 & 8 & 2 & -4 \\ 2 & 3 & 0 & -4 & 1 & 7 \\ 3 & \infty & 4 & 0 & 5 & 11 \\ 4 & 2 & -1 & 5 & 0 & -2 \\ 5 & 8 & \infty & 1 & 6 & 0 \end{array}$$

We calculate the value of 1 vertex to 4 vertex, we have two ways:

(*a*) 

, Total weight = 4

(*b*) 

, Total weight = 2.

We choose the case (*b*).

Now $m = 3$, *i.e.*, $L^{(3)} \leftarrow$ EXTEND-SHORTEST-PATHS ($L^{(2)}$, W)

i.e.,

$$L^{(3)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 3 & -3 & 2 & -4 \\ 2 & 3 & 0 & -4 & 1 & -1 \\ 3 & 7 & 4 & 0 & 5 & 11 \\ 4 & 2 & -1 & -5 & 0 & -2 \\ 5 & 8 & 5 & 1 & 6 & 0 \end{array}$$

Similarly,

$$L^{4} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & -3 & 2 & -4 \\ 2 & 3 & 0 & -4 & 1 & -1 \\ 3 & 7 & 4 & 0 & 5 & 3 \\ 4 & 2 & -1 & -5 & 0 & -2 \\ 5 & 8 & 5 & 1 & 6 & 0 \end{array}.$$

FASTER-ALL-PAIRS-SHORTEST-PATHS (W)

1. $n \leftarrow$ rows [W]
2. $L^{(1)} \leftarrow W$
3. $m \leftarrow 1$
4. While $m < n - 1$
5. do $L^{(2m)} \leftarrow$ EXTEND-SHORTEST-PATHS ($L^{(m)}$, $L^{(m)}$)
6. $m \leftarrow 2m$
7. return $L^{(m)}$.

The running time of FASTER-ALL-PAIRS-SHORTEST-PATHS is $\theta(n^3)$ because each of the $\lceil \lg(n-1) \rceil$ matrix products takes $\theta(n^3)$ time.

4.9 THE FLOYD-WARSHALL ALGORITHM

The Floyd-Warshall algorithm is sometimes known as the **Roy-Floyd** algorithm or WFI algorithm, since **Bernard Roy** described this algorithm in 1959 is a graph analysis

algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest path between all pairs of vertices. The Floyd-Warshall algorithm is an example of the dynamic programming.

The Floyd-Warshall algorithm takes the dynamic programming approach. This essentially means that independent sub-problems are solved and the results are stored for later use. The algorithm allows negative edges, but no negative cycles, as per usual with such shortest path problems.

The basic concept is simple: The algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $P = (V_1, V_2, V_3, \ldots\ldots, V_m)$ is any vertex of P other than v_1 to v_m, that is, any vertex in the set $\{V_2, V_3, \ldots\ldots, V_{m-1}\}$.

Let the vertices of G be $V = \{1, 2, \ldots\ldots, n\}$ and consider a subset $\{1, 2, \ldots.., k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots\ldots, k\}$ and let P be a minimum-weight path from among them. (Path P is simple, since we assume that G contains no negative-weight cycles). The floyd-Warshall algorithm exploits a relationship between P and shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots.., k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path P.

IF k is not an intermediate vertex of path P, then all intermediate vertices of path P are in the set $\{1, 2, \ldots.., k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots.., k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots.., k\}$.

If k is an intermediate vertex of path P, then we break p down into $i \xrightarrow{p_1} k \xrightarrow{p_2} j$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots.., k\}$.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & ; \text{ if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & ; \text{ if } k \geq 1 \end{cases}$$

FLOYD-WARSHALL (W)

1. $n \in$ rows [W]
2. $D^{(0)} \leftarrow W$
3. For $k \leftarrow 1$ to n
4. do for $i \leftarrow 1$ to n
5. do for $j \leftarrow 1$ to n
6. do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7. return $D^{(n)}$.

The running time of the Floyd-Warshall algorithm is determined by the three times nested for loops of lines 3-6. Because each execution of line 6 takes O (t) time. The algorithm runs in time θ (n^3).

NOTES

4.10 TRANSITIVE CLOSURE

Transitive closure of a graph G is defined as $G^* = (V, E^*)$, where

$$E^* = \{(i, j) \text{ there is a path from vertex } i \text{ to vertex } j \text{ in } G\}$$

For $i, j, k = 1, 2, ..., n$ we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, ..., k\}$ and 0 otherwise we construct the transitive closure $G^* = (V, E^*)$ by putting edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$.

The recursive definition of $t_{ij}^{(k)}$ is

$$t_{ij}^{(0)} = \begin{cases} 0 \ ; & \text{if } i \neq j \text{ and } (i, j) \notin E \\ 1 \ ; & \text{if } i = j \text{ or } (i, j) \in E \end{cases}$$

and for $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ij}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$$

TRANSITIVE-CLOSURE (G)

1. $n \leftarrow |\ V[G]\ |$
2. for $i \leftarrow 1$ to n
3. do for $j \leftarrow 1$ to n
4. do if $i = j$ or $(i, j) \in E\ [G]$
5. then $t_{ij}^{(0)} \leftarrow 1$
6. else $t_{ij}^{(0)} \leftarrow 0$
7. for $k \leftarrow 1$ to n
8. do for $i \leftarrow 1$ to n
9. do for $j \leftarrow 1$ to n
10. do $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$

Example. *Give an O (VE) – time algorithm for computing the transitive closure of a directed graph G = (V, E).*

Sol.

1. To construct a new graph $G^* = (V, E^*)$
2. E^* is initially empty.
3. For each vertex v traverse the graph G adding edges for every node encountered in E^*.

   It takes O (VE) time.

4.11 JOHNSON'S ALGORITHM

Johnson's algorithm is a way to find shortest path between all pairs of vertices in a sparse directed graph. It allows some of the edge weights to be negative number, but no negative-weight cycles may exist.

Note:

Sparse graph: A graph in which the number of edges is much less than the possible number of edges.

Dense graph: A graph in which the number of edges is close to the possible number of edges.

Johnson's algorithm uses the technique of reweighting, which works as follows: if all edge weight W in a graph G = (V, E) are non-negative, we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex ; with the fibonacci heap min-priority queue, the running time of this all-pairs algorithm is O (V^2 lg V + VE). If G has negative-weight edges, but no negative weight cycles, we simply compute a new set of non-negative edge weights that allows us to use the same method. The new set of edge weights $\hat{W}$ must satisfy two important properties:

(*a*) For all pairs of vertices $u, v \in V$, a path P is a shortest path from u to v using weight function w if and only if P is also a shortest path from u to v using weight function $\hat{W}$.

(*b*) For all edges (u, v), the new weight $\hat{W}(u, v)$ is non-negative.

Suppose given a weighted, directed graph G = (V, E) with weight function W : E $\to$ R, let $h : v \to R$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\hat{W}(u, v) = W(u, v) + h(u) - h(v)$$

Where $h(u)$ – label of u

$h(v)$ – label of v

In other words, Johnson's algorithm consists of the following steps:

1. First, a new node q is added to the graph, connected by zero-weight edge to each other node.
2. Second, the Bellman-Ford algorithm is used, starting from the new vertex q, to find for each vertex V the least weight $h(v)$ of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.
3. Next the edges of the original graph are reweighted using the values computed by the Bellman-Ford algorithm: an edge from u to v, having length W (u, v) is given the new length $W(u, v) + h(u) - h(v)$.
4. Finally, for each nodes, Dijkstra's algorithm is used to find the shortest paths from s to each other vertex in the reweighted graph.

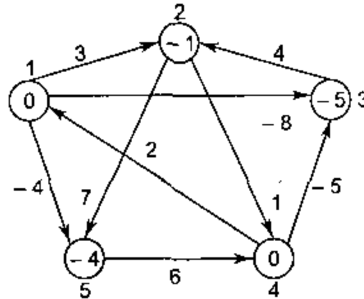
JOHNSON (G)

1. Compute G′, where V [G′] = V [G] $\cup$ {s}
   E [G′] = E [G] $\cup$ [(s, v) ; $v \in$ V [G]], and
   W (s, v) = 0 for all $v \in$ V [G]
2. If BELLMAN-FORD (G′, W, s) = FALSE
3. Then print u in the input graph contains a negative-weight cycle.
4. else for each vertex $v \in$ V [G′]
5. do set $h(v)$ to the value of δ [s, v] computed by Bellman-ford algorithm.
6. For each edge $(u, v) \in$ E [G′]
7. do $\hat{W}(u, v) \leftarrow W(u, v) + h(u) - h(v)$
8. For each vertex $u \leftarrow$ V [G]
9. do run DIJKSTRA (G, $\hat{W}$, u) to compute $\hat{\delta}(u, v)$ for all $v \in$ V [G]
10. For each vertex $v \in$ V [G]
11. do $d_{uv} \leftarrow \hat{\delta}(u, v) + h(v) - h(u)$
12. return D.

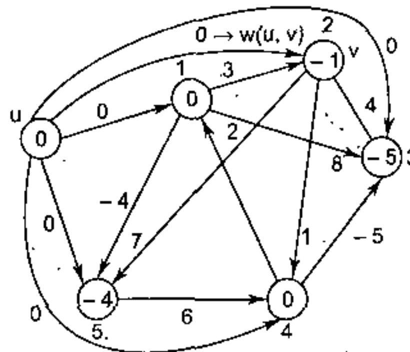
NOTES

Analysis: The time complexity of this algorithm, using Fibonacci heap in the implementation of Dijkstra's algorithm is O (V^2 lg V + VE): the algorithm uses O (VE) time for the Bellman-ford stage of the algorithm and O (V lg V + E) for each of V instantiations of Dijkstra's algorithm. Thus, when the graph is sparse, the total time can be faster than the floyd-Warshall algorithm, which solves the same problem in O (V^3) time.

Example. *Use Johnson's algorithm to find the shortest path between all pairs of vertices in the graph.*



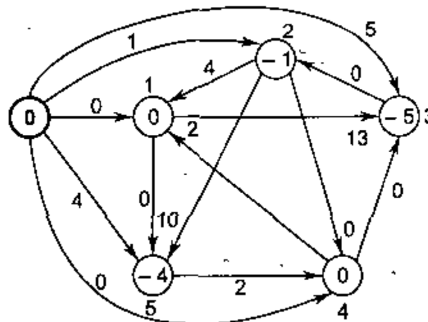
Sol. We compute G′. It means add a new vertex, connected by zero-weight edge to each other node.



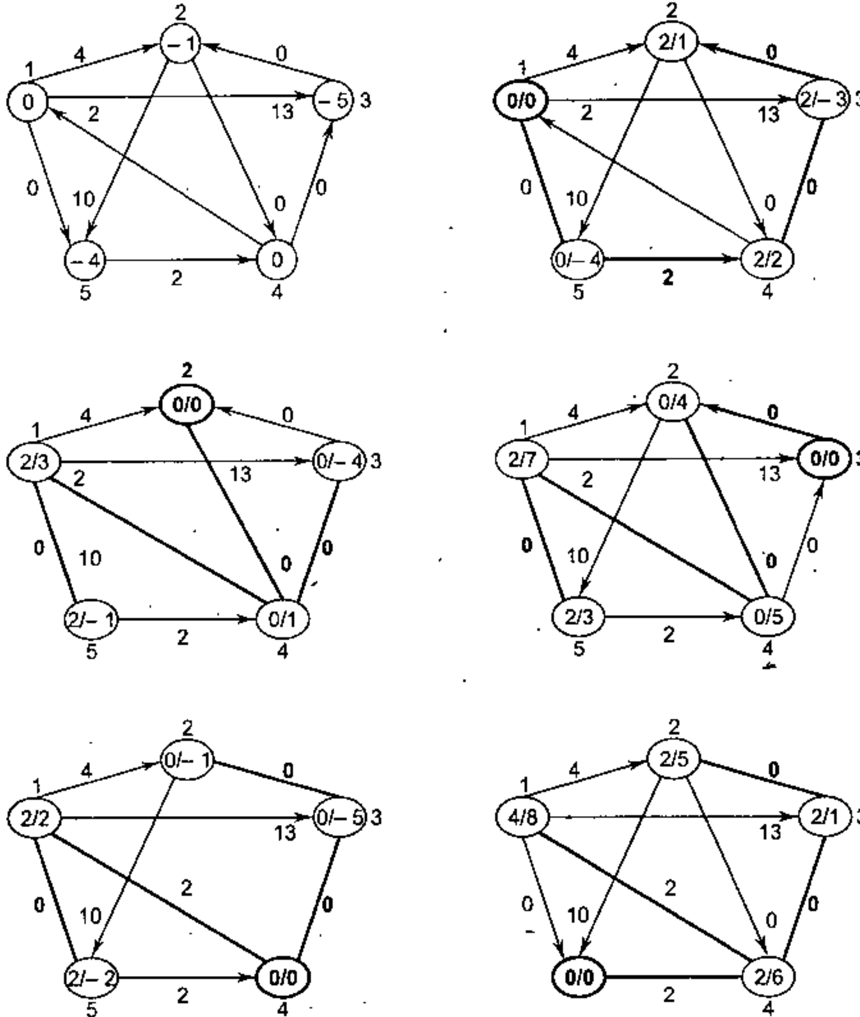
Note:

$$h(u) = 0$$
$$h(v) = -1$$
$$W(u, v) = 0$$
$$\hat{W}(u, v) = 0 + 0 + 1 = 1.$$

In this step, each edge (u, v) is reweighted with weight function $\hat{W}(u, v) = W(u, v) + h(u) - h(v)$.



In next step, we use Dijkstra's algorithm on each vertex of G using weight function $\hat{W}$. The source vertex u is black, and non-black edges are in the shortest paths tree computed by the algorithm. Within each vertex v are the values $\hat{\delta}(u, v)$ and $\delta(u, v)$, separated by slash. The value $\delta_{uv} = \delta(u, v)$ is equal to $\hat{\delta}(u, v) + h(v) - h(u)$. We remove source and also remove edge which are connected by source.



4.12 MAXIMUM FLOW

The **maximum flow problem** is to find a feasible flow through a single-source, single sink flow network that is maximum. Sometimes it is defined as finding the value of such a flow. The maximum flow problem can be seen as special case of more complex network flow problems, as the circulation problem. The maximum s-t (source-to-sink) flow in a network is equal to the minimum s-t cut in the network, as stated in the max-flow min-cut theorem.

Given a directed graph G = (V, E), where each edge u, v has a capacity C (u, v) we want to find a maximal flow f from the source s to the sink t, subject to certain constraints. There are many ways of solving this problem:

NOTES

Method	Complexity	Description
Linear Programming		Constraints given by the definition of a legal flow. Optimize $\sum_{v \in V} f(s, v)$
Ford-Fulkerson algorithm	O (E-max-flow)	As long as there is an open path through the residual graph, send the minimum of the residual capacities on the path.
Edmonds-Karp algorithm	O (VE^2)	A specialization of Ford-Fulkerson, Finding augmenting paths with breadth-first search.

4.13 FLOW NETWORKS AND FLOWS

A flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge can not exceed the capacity of the edge.

Definition

Suppose G (V, E) is a finite directed graph in which every edge $(u, v) \in E$ has a non-negative, real-valued capacity C (u, v). If $(u, v) \notin E$, we assume that C $(u, v) = 0$. We distinguish two vetices: a source s and a sink t. A flow network is a real function $f : V \times V \to R$ with the following three properties for all nodes u and v.

Capacity Constraints: $f(u, v) \le C(u, v)$. The flow along an edge can not exceed its capacity.

Skew Symmetry: $f(u, v) = -f(v, u)$. The net flow from u to v must be the opposite of the net flow from v to u.

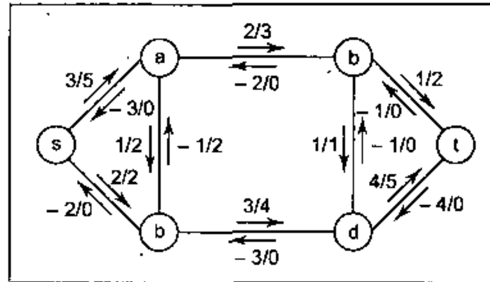
Flow Conservation: $\sum_{w \in V} f(v, w) = 0$, unless $u = s$ or $u = t$. The net flow to a node is zero, except for the source, which produces flow, and the sink, which consumes flow.

Notice that $f(u, v)$ is the net flow from u to v. If the graph represents a physical network, and if there is a real flow from u to v.

The **residual capacity** of an edge is $C_f(u, v) = C(u, v) - f(u, v)$. This defines a residual network denoted G_f (V, C_f) giving the amount of available capacity. An augmenting path is a path $(u_1, u_2, \ldots, u_k)$ in the residual network, where $u_1 = s$, $u_k = t$ and $C_f(u_i, u_{i+1}) > 0$. A network is at maximum flow if and only if there is no augmenting path in the residual network.

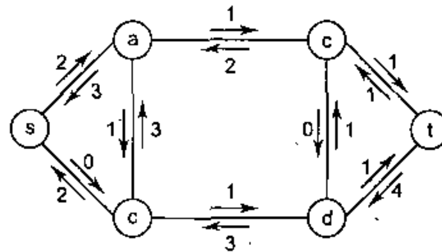
Example. To the right you see a flow network with source labeled s, sink t, and four additional nodes. The flow and capacity is denoted $\frac{f}{C}$. Notice that how the network upholds skew symmetry, capacity constraints and flow conservation. The total amount of flow from s to t is 5, which can be easily seen from the fact that the total outgoing flow from s is 5 which is also the incoming flow to t. We know that no flow appears or disappears in any of the other nodes.

NOTES



A flow network showing flow and capacity

Below you see the residual network for the given flow. Notice how there is positive residual capacity on some edges where the original capacity is zero, for example for the edge (d, c). This flow is not a maximum flow. There is available capacity along the paths $s \to a \to c \to t$, $s \to a \to b \to d \to t$ and $s \to a \to b \to d \to c \to t$, which are then the augmenting paths. The residual capacity of the first path is min (C $(s, a) - f(s, a)$, C $(a, c) - f(a, c)$, C $(c, t) - f(c, t)$ = min (5 – 3, 3 – 2, 2 – 1) = min (2, 1, 1) = 1.



Residual network for the above flow network, showing residual capacities.

Notice that augmenting path (s, a, b, d, c, t) does not exist in the original network, but you can send flow along it, and still get a legal flow.

If this is a real network, there hight actually be a flow of 2 from a to b, and a flow of 1 from b to a, but we only maintain the net flow.

4.14 MAX-FLOW MIN-CUT THEOREM

The **max-flow min-cut theorem** is a statement in optimization theory about maximum flows in flow networks. It drives from **Menger's theorem.** It states that:

"The maximum amount of flow is equal to the capacity of a minimal cut."

In other words, the theorem states that the maximum flow in a network is dictated by its bottleneck. Between any two nodes, the quality of material flowing from one to the other cannot be greater than the weakest set of links somewhere between the two nodes.

Definition

Suppose G = (V, E) is a finite directed graph and every edge (u, v) has a capacity C (u, v) (a non-negative real number). Further assume two vertices, the source s and the sink t, have been distinguished.

A cut is a split of the nodes into two sets S and T, such that s is in S and t is in T. Hence there are $2^{|V|-2}$ possible cuts in a graph. The capacity of a cut (S, T) is

$$C(S,T) = \sum_{u \in S, v \in T / (u,v) \in E} C(u,v)$$

NOTES

the sum of the capacity of all the edges crossing the cut, from the region S to the region T.

The following three conditions are equivalent:

1. F is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $| f | = C(S, T)$ for same cut (S, T).

Proof: If there is an augmenting path, we can send flow along it, and get a greater flow, hence it can not be maximal, and vice versa. If there is no augmenting path, divide the graph into S, the nodes reachable from S in the residual network, and T, those not reachable. Then C (S, T) must be 0. If it is not, there is an edge (u, v) with $C(u, v) > 0$. But then v is reachable from S, and can not be in T.

In particular this proves that max flow >= min cut, because a minimal cut is smaller or equal to the cut corresponding to our max flow.

Then we have min cut >= max flow. If we have a flow f for a given graph G_f removing an edge (u, v) of capacity C changes f in at least f-C, because no more than a capacity of C can be used by the flow f. But if we remove all edges cut by a given minimal cut C_0, we get a flow 0, whatever the flow f we have at first. So $f - C_0 <= 0$ for any flow f, in particular if f is a max-flow which shows $f <= C_0$.

4.15 THE FORD-FULKERSON METHOD

The **Ford-Fulkerson algorithm** (named for L.R. and Jr. and D.R. Fulkerson) computes the maximum flow in a flow network. It was published in 1956. The name "Ford-Fulkerson" is often also used for Edmonds-Karp algorithm, which is a specialisation of *Ford-Fulkerson.*

The idea behind the algorithm is very simple: As long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with avoidable capacity is called an **augmenting path**.

The Ford-Fulkerson method is iterative. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an "augmenting path", which we can think of simply as a path from the source S to the *sink t along* which we can push more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

FORD-FULKERSON-METHOD (G, *s*, *t*)

1. initialize flow f to 0
2. while there exists an augmenting path P
3. do augment flow f along P
4. return f.

The **FORD-FULKERSON** algorithm simply expands on the FORD-FULKERSON-METHOD pseudo code given above. Lines 1-3 initialize the flow f to 0. The while loop of lines 4-8 repeatedly finds an augmenting path P in G_f and augments flow f along P by the residual capacity C_f (P). When no augmenting paths exist, the flow f is a maximum flow.

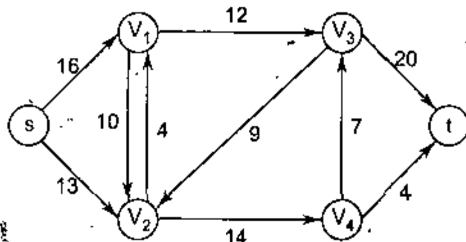
NOTES

FORD-FULKERSON (G, s, t)

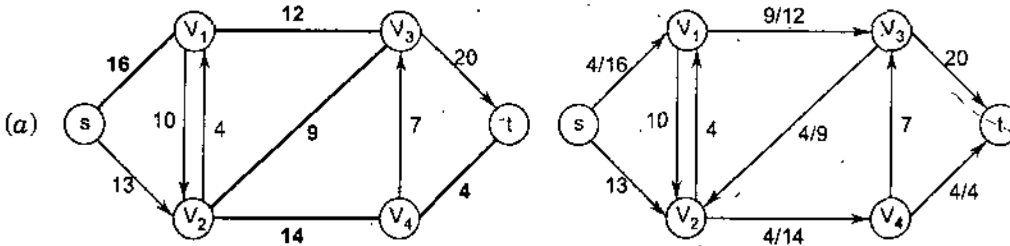
1. For each edge $(u, v) \in E[G]$
2. do $f[u, v] \leftarrow 0$
3. $f[v, u] \leftarrow 0$
4. while there exists a path P from s to t in the residual network G_f
5. do $C_f(P) \leftarrow \min\{C_f(u, v) : (u, v) \text{ is in } P\}$
6. for each edge (u, v) in P
7. do $f[u, v] \leftarrow f[u, v] + C_f(P)$
8. $f[v, u] \leftarrow -f[u, v]$

Analysis: The running time of FORD FULKERSON depends on how the augmenting path P in line 4 is determined. If it is chosen poorly, the algorithm might not even terminate the value of the flow will increase with successive augmentations, but it need not even coverage to the maximum flow value. If the augmenting path is chosen by using a breadth-first search, the algorithm run in polynominal time.

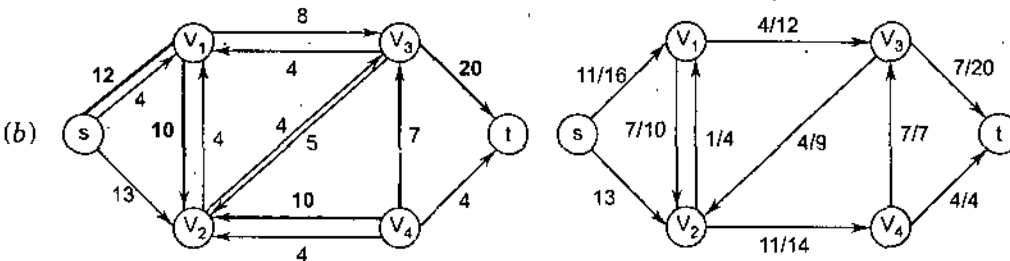
Example. *We are given a flow f in the following flow network. On each edge, the label C represents the capacity C on the edge:*



Solution. The left side of each part shows the residual network G_f with a shaded augmenting path P. The right side of each part shows the new flow f that results from adding f_P to f.

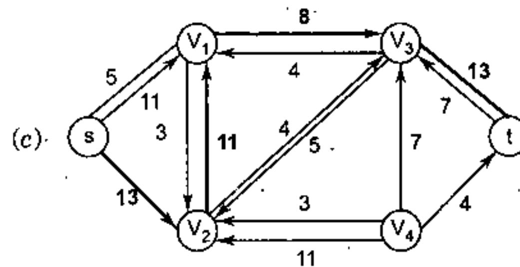


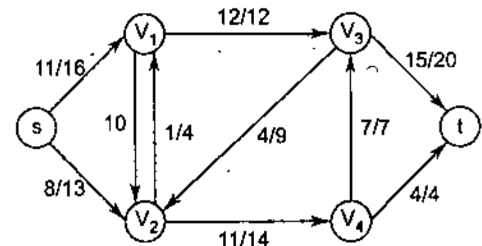
Path: $s \rightarrow V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow t$



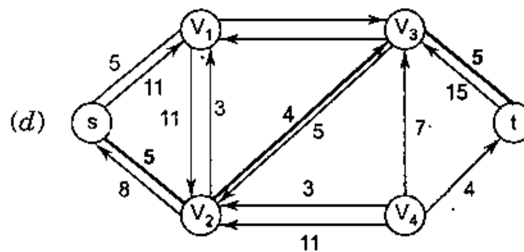
Path: $s \rightarrow V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_3 \rightarrow t$

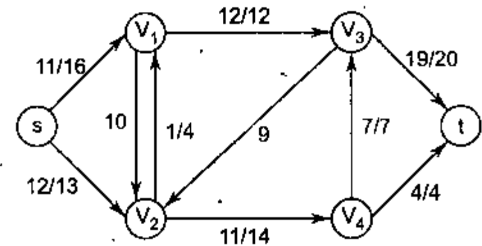
NOTES

(c) 

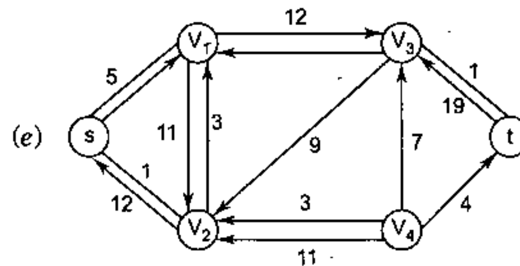


Path: $s \to V_2 \to V_1 \to V_3 \to t$

(d) 



Path: $s \to V_2 \to V_3 \to t$

(e) 

In part (*e*), it has no augmenting paths. The flow f in a with value $|f| = 19$ is a maximum flow.

Example. *Using the definition of a flow, prove that if $(u, v) \notin E$ and $(v, u) \notin E$, then*

$$f(u, v) = f(v, u) = 0$$

Sol. We assume that $(u, v) \in E$ and $(v, u) \notin E$, then by capacity constraints $f(u, v) \leq 0$ and $f(v, u) \leq 0$. By skew symmetry $f(u, v) = f(v, u) = 0$.

4.16 THE EDMONDS-KARP ALGORITHM

The bound on FORD-FULKERSON can be improved if we implement the computation of the augmenting path P in line 4 with a breadth-first search, that is, if the augmenting path is a shortest path from s to t in the residual network, where each edge has unit distance (weight). We call the FORD-FULKERSON method so implemented the Edmons-Karp algorithm. This algorithm runs in O (VE^2) time.

STUDENT ACTIVITY

1. Write Pseudo codes for Kruskal algorithm.

2. Discuss Kruskal and prim algorithm. Derive its complexity.

SUMMARY

NOTES

1. Unordered pair of vertices is called undirected edge and graph G (V, E) is called undirected graph. If, graph possess undirected edge.
2. Directed graph is 'oneway' graph where cost of one node to another are not necessary to equal to *vice versa.*
3. A graph G (V, E) with an edge which is associated an ordered pair of V × V is called a directed graph edge and such graph G (V, E) which consist directed edge is called directed graph.
4. Hamiltonian path is a subgraph of Hamiltonian circuit. So we conclude that Hamiltonian circuit always consist Hamiltonian path.
5. Hamiltonian circuit is a closed walk in which each vertex is traversed only once except the starting state it is similar to travelling sales person problem.
6. A complete bipartite graph is a bipartite graph in which every vertex of G_1 is adjacent to every vertex of G_2.
7. The Floyd-Warshall algorithm is sometimes known as the **Roy-Floyd** algorithm or WFI algorithm, since **Bernard Roy** described this algorithm in 1959 is a graph analysis algorithm for finding shortest paths in a weighted, directed graph.
8. Johnson's algorithm is a way to find shortest path between all pairs of vertices in a sparse directed graph. It allows some of the edge weights to be negative number, but no negative-weight cycles may exist.
9. The **max-flow min-cut theorem** is a statement in optimization theory about maximum flows in flow networks.

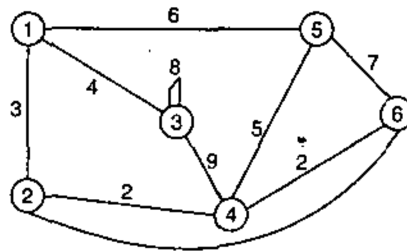
GLOSSARY

- *Sparse Graph:* A graph in which the number of edges is much less than the possible number of edges.
- *Dense Graph:* A graph in which the number of edges is close to the possible number of edges.
- *Maximum Flow:* The maximum flow problem is to find a feasible flow through a single-source, single sink flow network that is maximum.
- *Breadth First Search (BFS):* Breadth First search is elementary searching technique of a graph G(V, E) in this technique we visit or search the vertex and a function visit-L enable in respect of that node.
- *Flow Network:* A flow network is a directed graph where each edge has a capacity and each edge receives a flow.

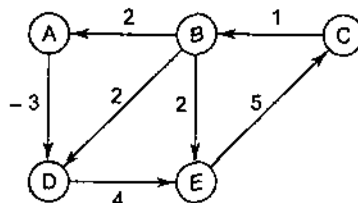
REVIEW QUESTIONS

1. Given an adjacency—list representation of a directed graph, how long does it take to compute the out degree of every vertex? How long does it take to compute the in-degrees?

2. Give an algorithm that determines whether or not a given undirected graph G = (V, E) contains a cycle. Your algorithm should run in O(V) time, independent of | E |.
3. Give an O (V + E) time algorithm to compute the component graph of a directed graph G = (V, E). Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.
4. Write pseudo codes for Kruskal algorithm.
5. Suppose that the graph G = (V, E) is represented as an adjacency matrix. Give a simple implementation of prim's algorithm for this case that runs in O (V^2) time.
6. Find out the minimum spanning tree (MST) of the following graph.



7. Discuss Kruskal and prim algorithm. Derive its complexity.
8. Why do we require that $W_{ii} = 0$ for all $1 \le i \le n$?
9. Write any algorithm to find all-pairs shortest path. Derive its complexity.
10. Modify the Floyd-Warshall algorithm to find the negative weight cycle.
11. Apply Floyd-Warshall algorithm for constructing shortest path.



12. How can the output of the Floyd-Warshall algorithm be used to detect the presence of a negative weight cycle?

FURTHER READINGS

- Gyanendra Kumar Dwivedi, '*Analysis and Design of Algorithm*', University Science Press.
- Hari Mohan Pandey, '*Design Analysis and Algorithm*', University Science Press.

UNIT

5

SELECTED TOPIC

STRUCTURE

5.0 Objectives
5.1 Randomized Algorithm
5.2 String Matching
5.3 NP-Hard and NP-Complete Problems
5.4 Approximation Algorithm
5.5 Sorting Networks
5.6 Comparison Networks
5.7 The Zero-one Principle
5.8 A Bitonic Sorting Network
5.9 A Merging Network
5.10 A Sorting Network
5.11 Matrix Operations
5.12 Properties of Matrices
5.13 Operations on Matrices
5.14 Strassen's Algorithm for Matrix Multiplication
5.15 Polynomials and the FFT
5.16 Representation of Polynomials
5.17 The DFT and FFT
5.18 Number-Theoretic Algorithms
5.19 Elementary Number Theoretic Notions
5.20 Greatest Common Divisor
5.21 Modular Arithmetic
5.22 Solving Modular Linear Equations
5.23 Computational Geometry
5.24 Line Segment Properties
5.25 Finding the Convex Hull
5.26 Finding the Closest Pair of Points
- *Summary*
- *Glossary*
- *Review Questions*
- *Further Readings*

5.0 OBJECTIVES

After going through this unit, you will be able to:

- explain about categories of Randomized Algorithm.
- describe types of string Matching.

- discuss about NP-Hard and NP-completeness.
- define approximation and Number Theoretic Algorithms.
- explain about sorting network and matrix.
- understand the Concepts of Computational geometry.

NOTES

5.1 RANDOMIZED ALGORITHM

A randomized algorithm is one that makes use of a randomizer (such as a random number generator). Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomizer algorithm could also differ from run to run for the same input. The execution time of a randomized algorithm could also vary from run to run for the same input.

Randomized algorithm can be categorised into two ways or classes they are called:

1. Las Vegas Algorithm
2. Monte Carlo Algorithm.

The first algorithm that always produced same (correct) output for the same input these are called Las Vegas algorithm. The execution time of a Las Vegas algorithm depends on the output of the randomizer and algorithm terminate fast and if not, it might run for a longer period of time and in general the execution time of a Las Vegas algorithm is characterized as a random variable.

Second algorithm whose output might differ from run to run for same input. These are called Monte Carlo algorithm.

For a problem such as hiring problem, in which it is helpful to assume that all permutations of the input are likely equal a probabilistic analysis will guide the development of a randomized algorithm instead of assuming a distribution of inputs, we impose the distribution.

We now explore the distinction between probabilistic analysis and randomized algorithm, assuming that the candidates are presented in a random order, the expected number of times we hire a new office assistant is about log n. The algorithm here is deterministic; for any particular input, the number of times a new office assistant is hired will always be the same and the number of times we hired a new office assistant differs for different inputs and it depends on the ranks of the various candidates. Since this number depends only on the rank of its candidates *i.e.*, {rank (1), rank (2) ... rank (n)}.

Given rank list $A_1 = \{1, 2, 3, 4, ...n\}$ new office assistant will always be hired n times and link executes in each iteration of the algorithm.

Given list of rank $A_2 = \{n, n-1, ... 3, 2, 1\}$ a new office assistant will be hired only one times. Given a list of ranks.

$A_3 = \left\{\frac{n}{2}, n-(n-2), 1, n-(n-8) \;......\frac{n}{2}+1\right\}$ a new office assistant will be hired three times upon interviewing the candidates with rank $\frac{n}{2}, \frac{n}{2}+3$, and n, recalling that the cost of our algorithm is dependent on how many times we hire a new office assistant. We see that three are expensive inputs, such as A, inexpensive inputs, such as A_2 and moderately expensive inputs, such as A_1.

For hiring problem, the only change needed in the code is to randomly permute the array.

Randomized – Hire – Assistant (n)

- randomly permute the list
- best ← 0 // candidate 0 is least-qualified dummy candidate
- for i ← 1 to n
- do interview candidate i
- if candidate i is better than candidate best
- then best ← i
- hire candidate i

With this simple change, we have created a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

Note 1. Assuming that the candidates are presented in a random order, algorithm HIRE-ASSISTANT has a total hiring cost of $0(C_n \log n)$.

Theorem 1. *The expected hiring cost of the procedure Randomized-Hire-Assistant is* $0(C_n \log n)$.

Proof. When we permute the input array, we have achieved a situation identical to that of the probabilistic analysis of Hire-Assistant.

The comparison between Note i and this theorem capture the difference between probabilistic analysis and randomized algorithms. In the Note 1, we make an assumption about the input and by this theorem, we make no such assumption, although randomizing the input takes some additional time.

In the remainder of this section. We discuss some issues involved in randomly permuting inputs.

Randomly permuting arrays. Many randomized algorithms randomize the inputs by permuting the given input array. We shall discuss two methods for doing so. We assume that we are given an array A which, without loss of generality, contains the elements 1 through n and goal is to produce a random permutation of the array.

One common method is to assign each element A [i] of the array a random priority P[i], and then sort the elements of A according to these priorities.

i.e., If our initial array A = {1, 2, 3, 4} and we choose random priorities P = {36, 3, 97, 19}, we would produce an array B = {2, 4, 1, 3}. Since the second priority is the smallest, followed by the fourth, then the first, and finally the third we call this procedure Permute-By-Sorting.

Permute-By-Sorting (A)

- n ← length (A)
- for i ← 1 to n
- do P [i] = Random (1, n^3)
- Sort A, using P as sort key.
- return A

Line 3 choose random number in between 1 and n^3. We use a range of 1 to n^3 to make it likely that all the priorities in P unique. Let us assume all the priority are equal.

The time consuming step is sorting step 4. If we use comparison sort, sorting takes Ω (n log n) time. We can achieve this lower bound, since we have seen that merge sort that takes θ (n log n) time after sorting if P [i] is the jth smallest priority, then A [i] will be in position j of the output. In this manner we obtain a permutation it remains to prove that the procedure produced a uniform random permutation. That is, that every permutation of the number 1 through n is equally likely to be produced.

Theorem 2. *Procedure Permute-By-Sorting produces a uniform random permutation of the input, assuming that all priorities are distinct.*

Proof. We start by considering the particular permutation in which each element A [i] receives the ith smallest priority. We shall show that this permutation occurs with probability exactly $1/n$!

For $i = 1, 2, 3 \ldots\ldots n-1, n$ Let X_i be the event that element A [i] receives the ith smallest priority. Then we wish to compute the probability that for all i, event X_i occurs which is

$$P_r\,[X_1 \cap X_2 \cap X_3 \cap \ldots \cap X_{n-1} \cap X_n]$$

and above probability is equal to

$$P_r\,[X_1] \,.\, P_r\,[X_2] \,.\, P_r\,[X_3] \mid X_2 \cap X_1 \mid .\, P_2 \mid X_4 \mid$$
$$\mid X_3 \cap X_2 \cap X_1 \mid \ldots P_r\,[x_i]\, X_{i-1} \cap X_{i-2} \cap \ldots \cap X_1]$$
$$P_r \mid X_n \mid X_{n-1} \ldots \cap \ldots \cap X_i]$$

We have that $P_r\{X_1\} = 1/n$ because it is the probability that one priority chosen randomly out of a set of n is the smallest. Next we observe that $P_r\,\{X_2 \mid X_1\} = 1/(n-1)$ because given that element A [1] has the smallest priority, each of the remaining $(n-1)$ elements has an equal chance of having the second smallest priority.

In general, For $i = 2, 3, \ldots\ldots n$ we have that $P_r\,\{X_i \mid X_{i-1} \cap X_{i-2} \cap \ldots \cap X_1\} = 1/(n-i+1)$.

Since given that elements A [1] through A [$i-1$] have the $i-1$ smallest.

Priority (in order), each of the remaining $n-(i-1)$ elements has an equal chance of having the ith smallest priority. Thus we have

$$P_r\,\{X_1 \cap X_2 \cap X_3 \ldots \cap X_{n-1} \cap X_n\} = \left(\frac{1}{n}\right) \cdot \left(\frac{1}{n-1}\right) \cdot \left(\frac{1}{2}\right) \cdot \left(\frac{1}{1}\right) = \frac{1}{n\,!}$$

and we have shown that the probability of obtaining the identical permutations is $\frac{1}{n\,!}$.

We can extend this proof to work for any permutation of priorities consider any fixed permutation $\sigma = \{\sigma(1), \sigma(2) \ldots \sigma(n)\}$ of the set $\{1, 2, \ldots n\}$. Let us denote by r_i the rank of the priority assigned to element A[i], where the element with the jth smallest priority has rank j. If we define X_i as the event in which element A [i] receives. The $\sigma(i)^{th}$ smallest priority, or $r_i = \sigma(i)$ the same proof still applies. Therefore, if we calculate the probability of obtaining any particular permutation the calculation is identical to the one above, so that the probability of obtaining this permutation is also $1/n$!.

One might think that to prove that a permutation is a uniform random permutation it suffices to show that, for each element A [i], the probability that it winds up in position j is $1/n$.

A better method for generating a random permutation is to permute the given array in place. The procedure randomize-in-place does so in $0\,(n)$ time.

In iteration i, the element A (i) is chosen randomly from among elements A[i] though A [n]. Subsequent to iteration i. A [i] is never altered.

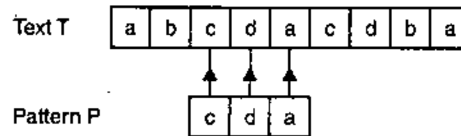
Randomize-In-Place (A)

- $n \leftarrow$ length [A]
- for $i \rightarrow 1$ to n
- do swap A [i] $\leftrightarrow$ A [Random (i, n)]

We will use a loop invarient to show that procedure Randomize-In-Place produces a uniform random permutation. Given a set of n elements a k-permutation is a sequence containing k of the n-elements. There are $n!/(n-k)$! such possible k-permutation.

5.2 STRING MATCHING

In this topic we will study various technique of pattern matching in given text or problem. For example



Length of pattern S = 3.

"String Matching problem is related to locate all or some occurrence of given pattern string with in a given text string pattern".

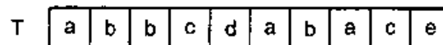
Example. *Text String*

$$T = < a, b, b, c, d, a, b, a, c, e >$$

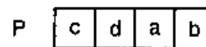
and a string of pattern P is

$P = < c, d, a, b >$. *Find whether P is present in T or not.*

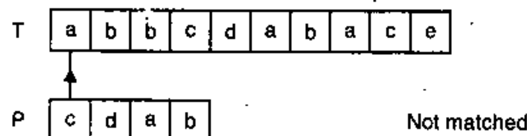
Sol. Given text T is



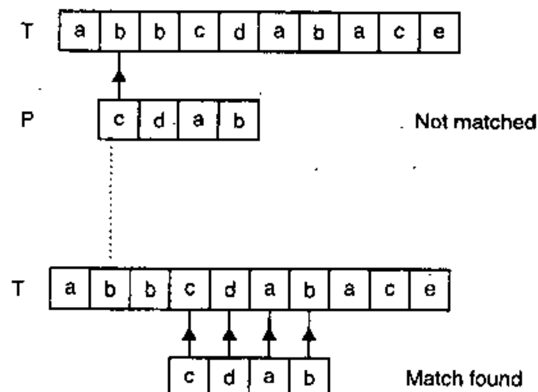
and pattern P is



then



Then pattern is shifted one position right and this will continue to match is not found or pattern last element located to last element of text T.



Hence given pattern matched to given text.

It on whole procedure match not found then we simply state pattern does not matched to given text (T).

Definition. So a text string T of length n over an alphabet Σ^*, and a pattern string (P) of length (m), the problem is locate all (or some) occurrence of P in text T.

NOTES

There is following four method of pattern string (P) matching technique they are:

- Brute Force or Naive algorithm
- Robin-krap algorithm
- Finite Automation
- Knuth-Morris-Pratt algorithm.

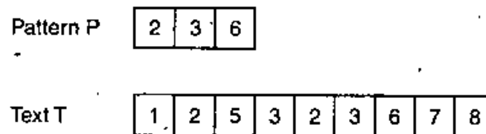
They have following preprocessing and matching time:

Algorithm	*Preprocessing Time*	*Matching Time*
Brute force or Naive	0	$0((n - m + 1)\, m)$
Robin-krap	$\theta(m)$	$0((n - m + 1)\, m)$
Finite automation	$0(m \mid \Sigma \mid)$	$\theta(n)$
Knuth-morris-pratt	$\theta(m)$	$\theta(n)$

Brute Force or Naive Algorithm

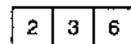
In Brute Force Naive algorithm, pattern "P" are compared character by character in the given text T. If pattern are not matched than it shifted one position to right side and comparison is repeated until match is found or the end of the text is reached.

Example. *Pattern P and text T is given below:*

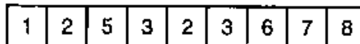


Find occurrence of pattern 'P' in given text 'T'.

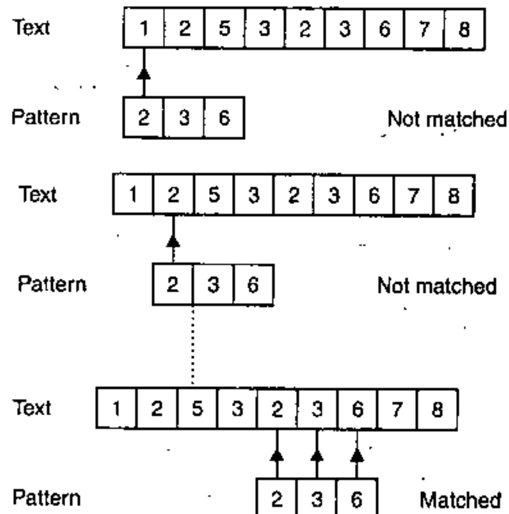
Sol. Pattern 'P' is



Text 'T' is



Then



Hence Pattern 'P' occurs or found in text 'T'.

Brute Force or Naive Algorithm

Step 1. Initializing, loop

set $a \leftarrow$ length [T]

set $b \leftarrow$ length [P]

for $i \leftarrow 0$ to $a - b$

set $j \leftarrow 0$

Step 2. while ($j < b$ and P[j] = T[$i + j$])

set $j \leftarrow j + 1$

if ($j = b$) then

return i

Step 3. Return value at call point return – 1

This algorithm finds the position of the first character of the pattern P in T if the pattern is present in T, ortherwise it returns the – 1 value for mis-match.

In Brute force algorithm pattern scan the whole text from beginning, starting to end. And search the exact pattern with in given text string in worst case run time is proportional to $a \times b$. Where a and b are two string length of text and pattern respectively. Thus the worst, case time is $\theta((a - b + 1) . b)$ which is $\theta(a^2)$ if $b = [a/2]$.

Robin-Krap Algorithm

Robin-krap have proposed a string matching algorithm that performs well in practice and that also generalize to other algorithm for related problems, such as two-dimensional pattern matching, the Robin-krap algorithm uses $\theta(m)$ processing time and for worst case running time is $\theta((n - m + 1)m)$. Based on certain assumptions, however, its average-case running time is better.

$\theta(m) \equiv$ Processing time for the pattern

$\theta(n) \equiv$ Processing time for the text study

The basic idea behind Robin-krap algorithm is hashing here we are using a function Cal which is similar to hashing function.

This function removes most of the strings from consideration on given text (T) and pattern (P). We have calculate Cal (P), then for each sub string S $\in$ T whose length is P. We have calculate Cal (S) and following point be considered they are:

- It can observed that since P = S, thus it implies Cal (P) = Cal (S) or Cal (P) $\neq$ Cal (S) then P $\neq$ S.
- String is ignored if Cal (P) $\neq$ Cal (S). Here we will present the procedure. Which will determine the sum of digits in given pattern string.

Example. *Match the given pattern 'P' in the text 'T' where pattern 'P' is:*

P = 3 2 4 5 and text T is

1	3	4	6	5	8	2	3	2	4	5	6	2

Sol. Pattern 'P' = 3 2 4 5

Cal (P) = 3 + 2 + 4 + 5 = 14

Pattern 'P' length is four so:

Text T	1	3	4	6	5	8	2	3	2	4	5	6	2

Pattern P	3	2	4	5	Cal (P) = 14

Text T	1	3	4	6	5	8	2	3	2	4	5	6	2
Cal (S)	–	–	–	14	18	23	21	18	15	11	14	17	17

NOTES

Where Cal (S) = Cal (P) we proceed the matching before 3 places due to that length of pattern is 4 and previous 3 and present one position is equal to length of pattern.

On above prove Cal (S) = 14 proceed two places first place of Cal (S) = 14 does not match but latter Cal (S) = 14 matches with the pattern.

Text T	1	3	4	6	5	8	2	3	2	4	5	6	2
Cal (S)	–	–	–	14	18	23	21	18	15	11	14	17	17

Pattern P	3	2	4	5

Pattern 'P' match with the text.

On calculating function calls as follows:

$$S = S_0 + S_1 + S_2 + S_3$$

Starting S have sum from S_0 to S_3.

Latter for next sub string.

$$S = S - S_0 + S_4$$

Similarly we calculate rest Cal (S).

Robin-krap – match (T, P, *d*, *q*)

- Set $n \leftarrow$ length [T] // find length of text in 'n'
- Set $m \leftarrow$ length [P] // find length of pattern in 'm'
- Set $h \leftarrow d^{m-1} \bmod q$.
- Set P $\leftarrow$ 0, to $\leftarrow q$
- loop for $i \leftarrow$ L to m
- Set P $\leftarrow (d_p + P[i]) \bmod q$
- Set to $\leftarrow (dt_0 + T[i]) \bmod q$
- // match of pattern
- For $s \rightarrow 0$ to $n - m$
- if $(P = t_s)$ then
- if (P [1, 2 .... m] $\rightarrow$ T [S + 1, ... S + m]) then
- display: "pattern match with place"
- if $(S < n - m)$ then
- Set $t_s + 1 \leftarrow (d(t_s - T[S+1]\, h) + T[S+m+1]) \bmod q$
- return.

Where T is given text, P is pattern for match and d is variable which show the radix of number and 'q' is same prime number.

In Robin-krap algorithm the modulo arithmetic operation is used to obtain the next sequence of text string to which the pattern string is to be matched. This algorithm takes $\theta(m)$ time for preprocessing, and $q((n - m + 1)\, m)$ time in the worst case for

NOTES

matching. The algorithm explicitly checks every valid shift. If P = a^m, and T = a^n, then the checking time is $q((n - m + 1)\, m)$, as each of the $(n - m + 1)$ possible shift is valid.

It can found that the probability of t_s = P (mod) q is $1/q$, so the expected number of match is $(0\ (n - m + 1)\ m/q)$. If we select $q \geq m$, then be expected running time of matching is $0(n - m + 1) = 0\ (n)$ because $n \geq m$.

String Matching Using Finite Automata Machine

The string matching problem can be solved by finite machine *i.e.*, using deterministic finite automata machine.

A D.F.A. machine can be expressed using 5 tuples they are:

M (Q, Σ, δ, q_0, F) where

Q = Finite set of states which will be non-empty.

Σ = Input alphabets

δ = Transition function which can be mapped like.

$$\delta\,.\; Q \times \Sigma \rightarrow R.$$

q_0 = Initial state

F = Find state

For to check the given pattern P match for automata machine we construct a D.F.A. machine of $(n + 1)$ states. Where n is number of alphabets or length of alphabets in pattern and last state belongs to final state.

Example. *Check given pattern "aabab" being accepted by finite automata machine are not.*

Sol. Given pattern is 'P' = '*aabab*' and automata machine m consist following S tuple they are:

$$M = (Q, \Sigma, \delta, q_0, F)$$

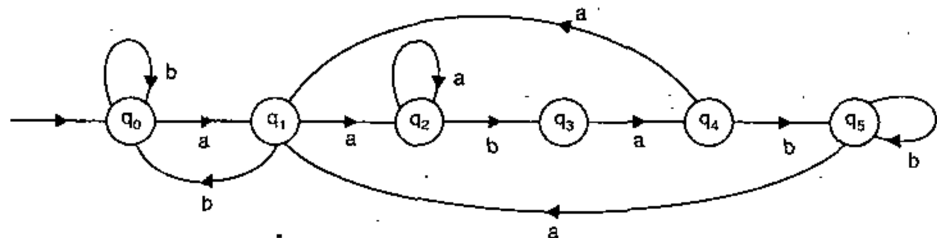
where Q = $\{q_0, q_1, q_2, q_3, q_4, q_5\}$ // set of state

Σ = $\{a, b\}$ // Input alphabets

δ: Q × Σ → Q.

$q_0 = \{q_0\}$ // initial state

F = $\{q_5\}$ // final state



For given pattern we construct D.F.A. machine which accept the pattern 'P'. *after that we include the direct edge for mismatch* and effort of mismatch can be calculated as:

- δ (i, a) = max $\{k \leq i \mid$ P $[i\,.\,k]$ is suffix of P [1, 2, ... i] if maximum exits.
- δ (i, a) = 0 if no suffix is obtained.

And search time for match is $0(n)$ time.

Algorithm for finite automata match

Finite–Automata–Match (T, δ, M)

- $n \leftarrow$ Length (T)
- $q \leftarrow 0$
- For $i \leftarrow 1$ to n.
- do $q \leftarrow \delta(q, T[i])$
- if $q = m$
- then print pattern occur at position $i - m$.

5.3 NP-HARD AND NP-COMPLETE PROBLEMS

Non-deterministic Algorithms

Algorithm that result different output at each every run, but lies under the defined set *i.e.*, output is not uniquely defined. Such type of algorithm is known as non-deterministic algorithm. These algorithm agree with the execution program on machine (computer). Theoretically we remove the restriction at each and every operation. It has functions like:

Choice (S): arbitrarily chooses one of the elements of the set S.

Failure ( ): signals an unsuccessful case that will be mostly – 1.

Success ( ): signals a successful completion. It will be some positive value.

Example. (Searching for an element x in a given set of elements A $[1 : n]$ $n \geq 1$.)

Non deterministic algorithm:

1. J : choice (1, n)
2. If A $[j] = x$ then {write (j); success;}
3. Write (0); failure ( );

Number 0 can be a output if and only if there is no j

such that A $[j] = x$

Algorithm is of nondeterministic complexity O(1)

However any deterministic search algorithm on unordered data is of $\Omega(n)$

Whenever there is set of choices that leads to a successful completion then one such set of choices is always made and the algorithms terminates successfully.

Whenever successful termination is possible, a nondeterministic machine makes a sequence of choices that is a shortest sequence leading to a successful termination. (remember machine is fictitious) A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to success signal.

```
Algorithm Non-deter-sort (A, n)
// Sort n positive integers
{
for i:=1 to n do B[i] : = 0;
for i:= 1 to n do
```

NOTES

```
{j:= choice (1, n)
If B[j] 0 then failure( );
B[j]:= A[i];
}
For i:= 1 to n – 1 do
if B[i] > B[i+1] then failure ( );
Write(B([1:n]);
Success ( );
}
```

It is $O(n)$ while all deterministic sorting algorithms have a complexity $\Omega(n \log n)$

Time of Non-deterministic Algorithms

The time required by a nondeterministic algorithm performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion.

In case successful completion is not possible then the time required is $O(1)$. A non-deterministic algorithm is of complexity $(f(n))$ if for all inputs of size $n \geq n_0$ that results is a successful completion the time required is at most $cf(n)$ for some constant c and n_0.

- Definition of P:
  - — Set of all decision problems solvable in polynomial time by a deterministic algorithm (Turing machine)

Examples:

— MULTIPLE: Is the integer y a multiple of x?

- YES : $(x, y) = (7, 21)$.

— RELPRIME: Are the integers x and y relatively prime?

- YES: $(x, y) = (56, 89)$.

— MEDIAN: Given integers $x_1, \ldots\ldots x_n$, is the median value < M?

- YES: $(M, x_1 \ldots\ldots x_2, x_3, x_4, x_5) = (17, 12, 15, 17, 22, 104)$

P is the set of all decision problems solvable in polynomial time on **REAL** computers.

Example. Ordered searching $O(n)$

Evaluation of polynomial $O(n)$ sorting $O(n \log n)$

There is another group of problems whose best known algorithms are non polynomial.

Example. Travelling salesman problem, Knapsack problem.

Deterministic Algorithm

Algorithm that is using the property yield the result uniquely defined same input *i.e.*, algorithm which result the same output for same input for each and every execution. This property of algorithm is know as deterministic property and algorithm is known as deterministic algorithm. Such type of algorithm can be run in machine (computer). While nondeterministic algorithm yield different-different output when executed in

machine but they are limited with in an internal *i.e.*, output contains within a domain $[m, n]$ where $m < n$.

NOTES

Decision problem. Any problem for which the answer is either zero or one is called decision problem and algorithm is known as decision algorithm.

- *Any problem for which the answer is yes or no, is called a decision problem.*

Example. Knapsack problem:

Optimization problem. Find the largest total profit of any subset of objects that fits in the knapsack, it also identify optimal value of a given cost.

Decision problem. Given k, is there a subset of objects that fits in the knapsack and has total profit at least k? Is there a subset of the objects whose sizes add up to exactly C?

- The decision problem can be solved in polynomial time if and only if the corresponding optimization problem can.
- If the decision problem cannot be solved in polynomial time then the optimization problem cannot either.

Example. Subset sum problem:

Input a positive integer C and n objects whose sizes are positive integers $S_1, S_2 \ldots S_n$. **Optimization problem**. Among subsets of the objects with sum at most C what is largest subset sum.

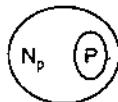
Decision Problem. Is there a subset of the objects whose sizes add up to exactly C ?

- Definition of NP:
  - Set of all decision problems solvable in polynomial time by a NONDETERMINISTIC Algorithm (Turing machine).
  - Definition is important because it links many fundamental problems.
- Useful alternative definition:
  - Set of all decision problems with efficient verification algorithms.
- efficient = polynomial number of steps on deterministic TM:
  - Verifier: algorithm for decision problem with extra input.

The class NP consists of those problems that are verifiable in polynomial time.

If any NP complete problem can be solved in polynomial time, then every NP complete problem has a polynomial time algorithm.

Relation between P and NP

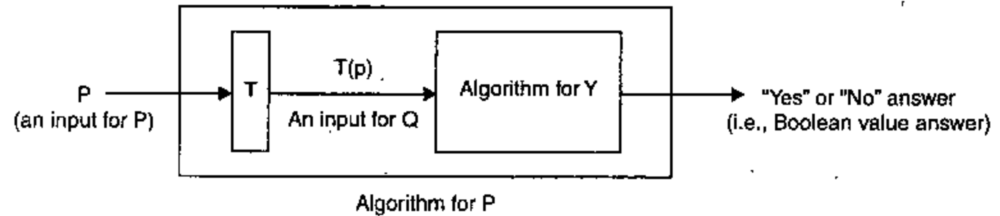


NP = set of decision problem with efficient verification algorithms

- Why doesn't this imply that all problems in NP can be solved efficiently?
  - BIG PROBLEM: need to know certificate ahead of time:
- real computers can simulate by guessing all possible certificates and verifying.
- naïve simulation takes exponential time unless you get "lucky".

Reducibility

Let L_1 and L_2 be two problem. Problem L_1 reduces to problem L_2 (written as $L_1 \alpha L_2$) if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time.



Algorithm for P

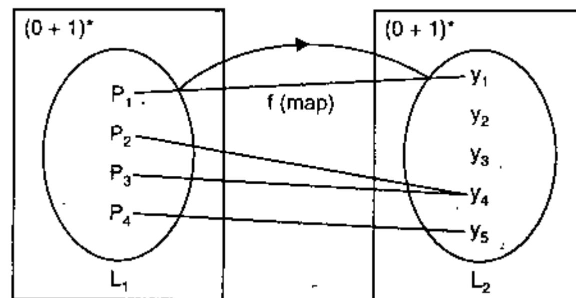
A problem P_1 can be reduced to P_2 as follows:

Provide a mapping so that any instance of P_1 can be transformed to an instance of P_2.

Solve P_2 then map the answer back to the original.

For P_1 to be polynomially reducible to P_2. All the work associated with the transformations must be performed in polynomial time.

Example. Numbers are entered into pocket calculator in decimal. The decimal numbers are converted in binary; All calculations are done in binary. Then the final answer is converted back to decimal for display.



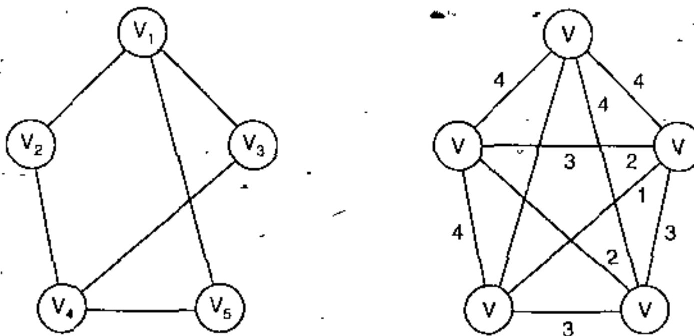
Here a set L_1 problem $\{P_1, P_2, P_3, P_4\}$ are reduciable to set L_2 of problem $\{y_1, y_4, y_5\}$. Hence set L_1 is reduciable to L_2. (L_1 reduciable to L_2).

Example of reduction:

Travelling salesman decision problem is NP complete? Suppose we already know that Hamiltonian cycle problem is NP complete. "Given a complete graph G = (V, E) with edge costs and an integer K, does there exists a simple cycle that visits all vertices and has total cost <= K".

This problem is different than Hamiltonian cycle problem because all possible edges are present in G and graph is weighted. It is easy to see that a solution can be checked in polynomial time so it is certainly in NP. To show that it is NP complete we polynomials reduce the Hamiltonian cycle problem to it. Construct a new graph G′ as:

G′ has the same set of vertices as G. For G′ each edge (v, w) has a weight of 1 if (v, w) is in G and 2 otherwise. Choose K = $|V|$. It is easy to verify that G has a Hamiltonian cycle if and only if G′ has travelling salesman tour of total weight $|V|$



Cook's theorem. *Satisfiability is in P if and only if P = NP*

So if a polynomial time algorithm can be obtained for satisfiability then every problem of NP can also be solves by a polynomial time algorithm.

So P = NP.

Proof. Go through only result proof is out of course.

Satisfiability Problem. The satisfiability problem is to determine whether a boolean formula is true for some assignment of truth values to the variables. (boolean formula is an expression that can be constructed using literals and operators "and", "or". CNF-satisfiability is the satisfiability problem for CNF (conjunctive) formula.

Nondeterministic algorithm for satisfiability problem

Algorithm Eval (E, n)

{

For i:= 1 to n do

x_i := choice (false, ture);

If E(x_1, x_2 ... x_n) then success ( );

Else failure ( );

}

Time of the nondeterministic algorithm O(n) time is required to choose the value of (x_1, x_2,... x_n) + Time needed to evaluate E for that assignment (time is proportional to the length of E).

Circuit Satisfiability Problem (CSP)

Circuit satisfiability problem is boolean combinational circuit composed of AND, OR and NOT gates, CSP one or more boolean combinational elements interconnected by wires. A wire can connect is series *i.e.*, output of one work as input of another and a single wire may have no more than one combinational element output connected to a wire, the wire is called **Fan-out** of the wire. If no output element is connected to a wire, the wire is a **circuit input** and take input from external source, if no element input is connected to a wire, the wire is a circuit output.

The circuit satisfiability problem can be explained mathematically as:

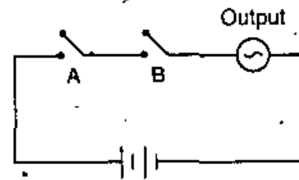
Circuit satisfiability problem = {(n) : n is a satisfiable boolean combinational circuit}.

Boolean circuit of AND, OR, NOT, NAND, NOR, Ex-OR circuit is:

AND. This boolean circuit work like multiplication of input. Where A and B are input 0 is output

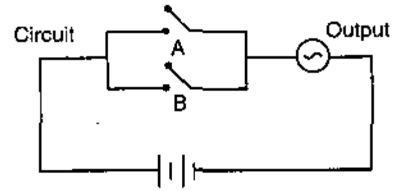
NOTES

A	B	A ∧ B
0	0	0
0	1	0
1	0	0
1	1	1



OR. It like addition of boolean input.

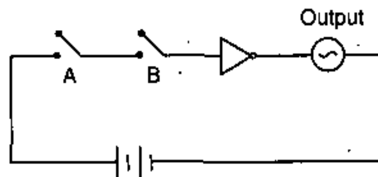
A	B	A ∨ B
0	0	0
0	1	1
1	0	1
1	1	1



NOT. It work like invertor or complement

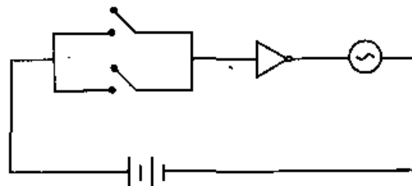
A	output
0	1
1	0

NAND. AND + NOT = NAND



A	B	$\overline{A}$	$\overline{B}$	$\overline{A \wedge B}$	$A \wedge B$
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	0	1

NOR. OR + NOT = NOR

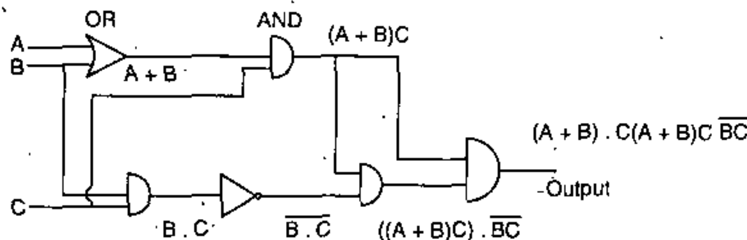


A	B	$\overline{A}$	$\overline{B}$	$A \vee B$	$\overline{A \vee B}$
0	0	1	1	0	1
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	1	0

Ex-OR. For same input (0, 0 or 1, 1) output is 0,

A	B	$\overline{A}$	$\overline{B}$	$A \theta B$	$(A'B + AB')$
0	0	1	1	0	0
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

Example of circuit satisfiable problem:



Output = (A + B) C (A + B) C $\overline{BC}$

$= (A + B) . C ((A + B) C) (\overline{B} + \overline{C})$ using property $\overline{BC} = \overline{B} + \overline{C}$

Important Results

1. Circuit satisfiable problem belongs to the class NP.
2. The circuit satisfiable problem is NP Hard.
3. The circuit satisfiable problem is NP complete.
4. Satisfiability of boolean formulas in 3-conjcetive normal form is NP complete.
5. Clique problem is NP complete.
6. Vertex cover problem is NP complete.

Remark. A problem Q is NP-hard does not mean **"in NP and hard"** It means **"at least as hard as any problem in NP".**

NP Hard Problem

A problem L is NP hard if and only if satisfiability reduces to L.

Hard ness of the NP hard problem can be thought of as. If one can generate polynomial time algorithm for any NP hard problem then by the definition of reducibility it means that satisfiability problem can also be solved by a polynomial time algorithm, which we know how important it is **(remember 'Cook's theorem").**

NP Complete Problem

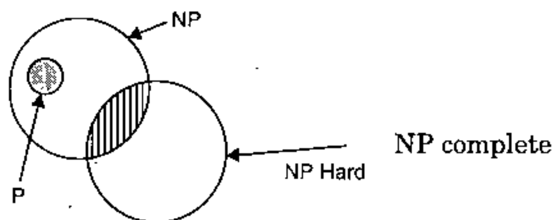
A problem is NP complete if and only if L is NP hard and L is in NP.

These are commonly know NP Complete problems.

Vertex cover problem. Optimization problem: Given an undirected graph G find a vertex cover for G with as few vertices as possible.

Decision problem. Given an undirected graph G and an integer *k*, does G have a vertex cover consisting of *k* vertices.

Graph coloring, Hamiltonian cycle, Hamiltonian path job scheduling with penalities, bin packing, the subset sum, the knapsack problem.



Relationship between P, NP, NP Hard and NP Completeness

$P \neq NP$ question has been one of the deepest, most perplexing open research problems in theoretical computer science. In other words no problem has been found for which it can be proved that it is in NP but not in P.

NP complete problems are hardest among NP problems. A problem that is NP complete can essentially be used as a subroutine for any problem in NP, with only a polynomial amount of overhead.

Thus if a problem has a polynomial time algorithm then every problem in NP must also have.

Prove that problem Q in NP is NP complete. If it is known that problem Q is NP. Then some other NP complete problem is to shown reducible to Q. Since reducibility is transitive satisfiability is reducible to Q. So Q is NP complete.

Polynomially equivalent problems. Two problems L_1 and L_2 are said to be polynomially equivalent if and only if $L_1 \propto L_2 \propto L_1$.

Example of an NP hard problem that is not NP complete:

The Halting problem. For an arbitrary deterministic algorithm A and input I whether algorithm A with input I ever terminates (or enters an infinite loop). In simple words halting problem is to determine whether an arbitrary given algorithm (or computer program) will eventually halt (rather than say, get into infinite loop) while working on a given input. It is well known that it is undecidable so there exists no algorithm of any complexity to solve it.

So it can not be in NP. Is it possible to have a compiler having extra feature that not only detects syntax errors but also all infinite loops?

Clique Decision problem (CDP). The conjective normal form (CNF) α clique decision problem and using transtivity of reducibility and above result we conclude that satisfiability reduces to conjective normal form-satisfiability. Hence satisfiability reduces to clique decision problem and clique decision problem is NP hard. Since clique decision problem belongs to NP, so clique decision problem is also NP complete.

Max clique problem. This problem is an optimization problem which determine the largest size of graph G (V, E).

It provided a sequence of edges and an integer value n each edge in E(a) is a pair of number (m, m + 1) or (m, l). The size for each edge (m, l) is

$$= \log_2 m + \log_2 l + 2$$

And any instance size of input

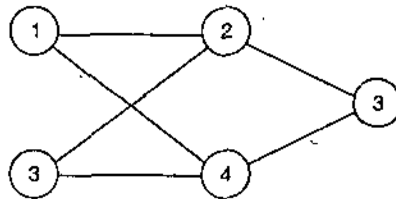
$$a = \sum_{\substack{(m,l) \in E(a) \\ m < l}} [(\log_2 m) + [\log_2 l] + 2 + (\log_2 n) + 1]$$

Node cover decision problem (NCDP). A set N ⊆ V is node cover for a given graph G (V, E) {where V is vertex and E is an edge of (i, j)} iff all edges in E are incident to at least are vertex in N.

The size | N | of the curve is the number of vertices in S.

Example. *For given graph G (V, E). N = {2, 4} is node cover of size | N | = 2, N = {1, 3, 5} is a node cover of size 3. i.e., | N | = 3.*

Sol.



Where node {2, 4} covers whole graph and similarly another set N = {1, 3, 5} covers whole graph G (V, E), but optimal solution is {2, 4} and size | N | = 2.

5.4 APPROXIMATION ALGORITHM

Introduction

A feasible solution with valve close to the value of optimal solution is called an approximate solution. An approximation algorithm for P is an algorithm that generates approximate solution for P.

We can approximate the solution of NP complete problems, by approximating NP complete problems we mean to obtain a solution for the NP-complete problem rather than to obtain optimal solution. For example.

In travelling sales person problem (TSP), the optimization problem is to find the shortest cycle and approximation problem find short cycle, that is short cycle may be shortest cycle but need not be compulsory.

And for vertex cover problem we need to find minimum vertex to cover whole graph G (V, E), but it should not necessary that vertex cover problem is optimal solution provided algorithm.

For measure of error to computed approximation assume N > 0 is an approximation to the exact value N* > 0 then ratio bound N of P is:

$$\text{Max} \left\{ \frac{N}{N^*}, \frac{N^*}{N} \right\} \leq P$$

Definition. A feasible solution with the value close to the value of an optimal solution is called an approximation solution and algorithm is known as *approximation algorithm.*

NOTES

An approximation algorithm for P is an algorithm that generate approximate solution for P.

A *approximate algorithm for* a problem is an absolute approximation for problem P iff very instance I of P is:

$$|A^*(I) - \hat{A}(I)| \leq n$$

where n is same constant.

where $A^*(I) \equiv$ value of optimal solution

$\hat{A}(I) \equiv$ value of feasible solution.

Note. If $A(n)$ is an approximate algorithm, it happens when following condition satisfy.

$$\frac{|A^*(I) - \hat{A}(I)|}{A^*(I)} \leq A(n)$$

For $A^*(I) > 0$

For size n.

Note. 1. An ε-approximation algorithm is an $A(n)$-approximation algorithm for which $A(n) \leq \varepsilon$ for some constant ε.

2. For Maximization problem $\frac{|A^*(I) - \hat{A}(I)|}{A^*(I)} \leq L$.

For every feasible solution of I. Hence for maximization problems we normally required $\varepsilon < 1$. For an algorithm to judge ε-approximation.

3. $A(\varepsilon)$ is an approximation scheme iff for every for every given $\varepsilon > 0$ and problem instance I, $A(\varepsilon)$ generates the feasible solution such that $\frac{|A^*(I) - \hat{A}(I)|}{A^*(I)} \leq \varepsilon$ and assume $A^*(I) > 0$.

4. An approximation scheme is polynomial time approximation scheme iff (if and only if) for every fixed $\varepsilon > 0$, it has a computing time that is polynomial in problem size.

5. An approximation scheme whose computing time is polynomial both in the problem size and in $\frac{1}{\varepsilon}$ is a fully polynomial time approximation.

Vertex Cover Problem

A vertex cover of an undirected graph $G(V, E)$ is a subset $V' \subseteq C$. Such that if $(u, v) \in E$, then $u \in v'$ or $v \in v'$ (or both), where each vertex "covers" the incident edges, and a vertex is the number of vertices in it.

Algorithm for vertex cover problem

Vertex cover ( )

Step 1. Initialization

Set $V_n = \phi$

// Initialize an empty vertex cover

Step 2. Loop

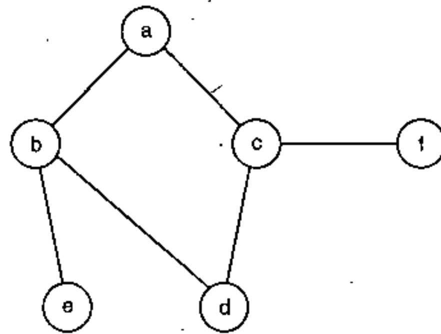
for all $(u, v) \in E$

if $(u \notin V_n)$ & & $v \notin V_n)$ then

Set $V_n = V_n \cup \{u, v\}$

Step 3. Return value at the point of call return V_n.

Example.



Graph G (V, E) has vertex cover {b, c} of (| V_N | = 2) length 2.

- The vertex cover problem is to find a vertex cover of minimum size in graph G (V, E).

  Vertex cover (V.C) = {(G, V_N)}

  V.C. = {G, 2}

  Vertex cover = {(G, N) : where graph G (V, E) has vertex cover of size N}
- Vertex cover problem is NP complete.

ε-Approximation

Scheduling task. Scheduling task works on scheduling rule, which generate a finished time, that is close to the optimal schedule. An instance I of scheduling problem is defined by a set of n task times, t_i, $1 \le i \le n$ and $m \ge 2$, m be the number of processor.

Scheduling rule is known as longest processing time problem (L P T-problem) rule.

Longest processing time. The longest processing schedule is one that is the result of an algorithm that, whenever a processor become free, assign to the processor a task whose time is the largest of those tasks not yet assigned. Ties are broken in arbitrary manner *i.e.*, when a short time process in queue it may be possible that executing process aborted for some time and wait for signal to execute its rest task up to which it does not get signal. For execution its stage remain wait(s) position.

Example. *Let $m = 3$, and number of process $n = 6$ from (p_1, p_2, ...p_6) and time for execution are respectively (t_1, t_2, t_3, t_4, t_5, t_6) = (8, 7, 6, 5, 6, ,4) in an longest processing time schedule. The task 1, 2 and 3 are assigned to processor 1, 2 and 3 respectively.*

Sol. Finished time $= \dfrac{\Sigma t_i}{\text{Numbers of processor}} = \dfrac{\Sigma t_i}{m}$

$$= \frac{8+7+6+5+6+4}{3}$$

$$= \frac{36}{3} = 12.$$

P_1	1	6
P_2	2	4
P_3	3	5

where process one and six are assigned to processor one P_1 process two and four are assigned to processor two P_2. Process three and five are assigned to processor three P_3.

NOTES

Planar Coloring Graph

Planar coloring graph problem is that by which to determine the minimum number of colors needed to a planar graph G (V, E) to color and every planar graph is four colorable.

We can easily determine whether a graph G (V, E) is zero, one or two colorable.

Zero colorable iff $v = \phi$ (vertex is zero *i.e.*, no vertex)

One colorable iff $E = \phi$ (*i.e.*, there is no any edge)

Two colorable iff bipartion of graph and $V \neq 0$ and $E \neq 0$

Algorithm for planar coloring graph

- A color (V, E)
- // Determine the approximation to the minimum numbers of colors.
- {
- if V = 0 return 0,
- else if E = ϕ then return (1).
- else if G is bipartite then return 2;
- else return 4.
- }

Travelling Sales Person Problem

If complete undirected graph G (V, E) has a non-negative integer cost $C(u, v)$ associated with each edge $(u, v) \in E$, and hamiltonian cycle exist with minimum cost an extension of our notation, let cost (A) denote the total cost of the edges in the subset $A \subseteq E$.

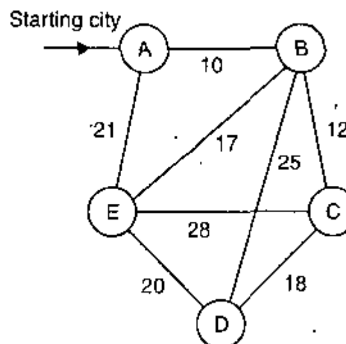
$$\text{cost (A)} = \sum_{(u,\, v) \in A} \text{cost}(u, v)$$

In triangle in equality if i, j, k vertices belongs to V. Then

$$\text{cost}(i, k) \leq \text{cost}(i, j) + \text{cost}(j, k)$$

"Travelling sales person problem, is problem where an salesmen visit each and every city exactly ones no one city visited twice and sales person started from which city came back that city after visiting each and every city exactly visited once.

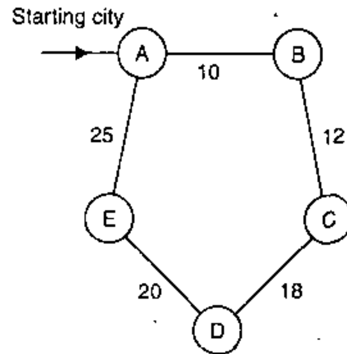
Example. *If a sales person visited following*



Then convert above problem is TSP problem.

NOTES

Sol.



5.5 SORTING NETWORKS

In sorting network, we investigate sorting algorithm based on a comparison-network model of computation, in which many comparison operations can be performed simultaneously. Comparison networks differ from RAM's in two important respects. These are the following:

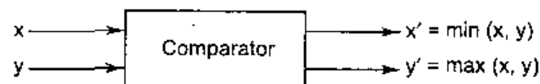
- First, they can only perform comparisons. It means that an algorithm such as counting sort can not be implemented on a comparison network.
- Second, unlike the RAM model, in which operations occur serially – that is, one after another operations in a comparison network may occur at the same time, or in parallel.

5.6 COMPARISON NETWORKS

Sorting networks are comparison networks that always sort their inputs. A **comparison network** is a combination of wires and comparators. A **comparator** is a device with two inputs, x and y, and two outputs, x' and y', that performs the following function:

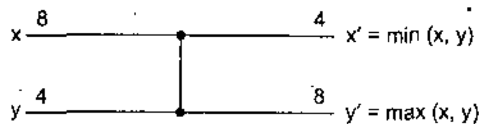
$$x' = \min(x, y)$$
$$y' = \max(x, y).$$

Example. *The following figure shows a comparator with inputs x and y and output x' and y'.*



The above figure drawn as a single vertical line.

Inputs $x = 8, y = 4$ and outputs $x' = 4, y' = 8$ are shown. *i.e.*, Inputs appear on the left and outputs on the right, with the smaller input value appearing on the top output and the larger input value appearing on the bottom output.



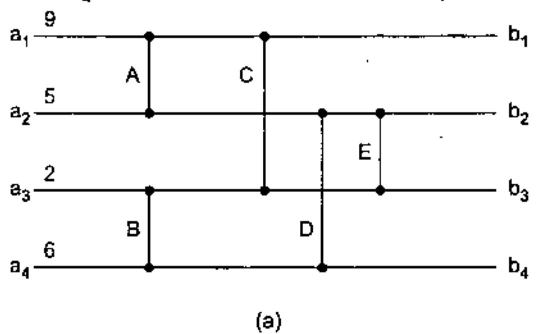
NOTES

We assume that each comparator operates in O (1) time. In other words, the time between the appearance of the input values x and y and the production of the output values x' and y' is a constant.

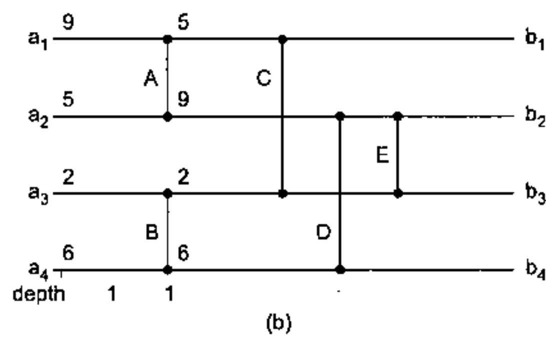
A wire transmits a value from place to place. Wires can connect the output of one comparator to the input of another, but they are either network input wires or network output wires. Suppose a comparison network contains n input wires $a_1, a_2, ..., a_n$ through which the values to be sorted enter the network, and n output wires $b_1, b_2, ..., b_n$, which produce the result computed by the network.

Comparison network is a collection of comparators interconnected by wires. We draw a comparison network on n inputs as a collection of n horizontal lines with comparators stretched vertically. A line does not represent a single wire, but rather a sequence of distinct wires connecting various comparators. We assume that each comparator takes unit time, we can define the "running time" of a comparison network, *i.e.*, the time it takes for all the output wires to receive their values once the input wires receive theirs. We can define the depth of a wire as follows. An input wire of a comparison network has depth 0. If a comparator has the input wires with depths dx and dy, then its output wires have depth max $(dx, dy) + 1$. Because there are no cycles of comparators in a comparison network.

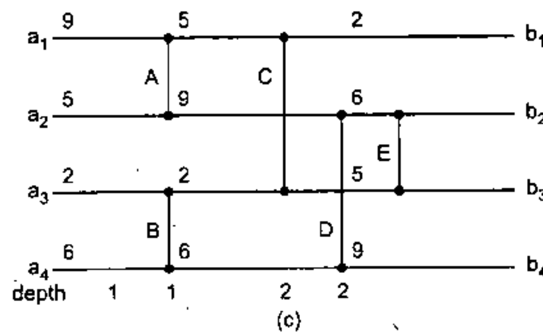
In the following figure, (*a*) A 4-input and 4-output comparison network which is a sorting network. Suppose that the sequence {9, 5, 2, 6} appears on the input wires at time 0. At time 0, only comparators A and B have all their input values available. Assuming that each comparator requires one time unit to compute its output values, comparators A and B produce their outputs at time 1. In figure (*b*), the comparators A and B produce their values at the same time, or "in parallel. At time 1, comparators C and D, but not E, have all their input values available. In figure (*c*), at time 2, they produce their outputs at depth 2. Comparators C and D operate in parallel as well. The top output of comparator C and the bottom output of comparator D connect to output wires b_1 and b_4, respectively, of the comparison network, and these network output wires carry their final values at time 2. At time 2, comparator E has its input available. In figure (*d*), at time 3, it produces its output values. Then values are carried on network output wires b_2 and b_3 and the output sequence {2, 5, 6, 9} is now complete.



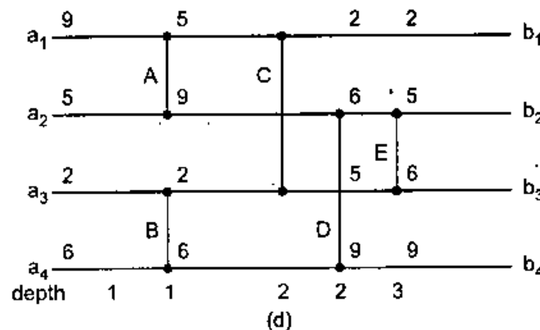
(a)



(b)



(c)



(d)

A **sorting network** is a comparison network for which the output sequence is monotonically increasing (*i.e.*, $b_1 \le b_2 \le ..... \le b_n$) for every input sequence. But every comparison network is not a sorting network.

A comparison network is like a procedure in that it specifies how comparisons are to occur, but it is unlike a procedure in that its size – the number of comparators that it contains – depends on the number of inputs and outputs.

5.7 THE ZERO-ONE PRINCIPLE

In zero-one principle, if a sorting network works correctly when each input is drawn from the set {0, 1}, then it works correctly on arbitrary input numbers. Note that the number can be integers, real, or, in general, any set of values from any linearly ordered set.

Lemma

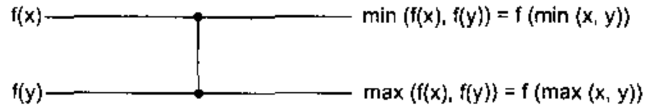
In a comparison network transforms the input sequence $a = \{a_1, a_2, ..., a_n\}$ into the output sequence $b = \{b_1, b_2, ..., b_n\}$ then for any monotonically increasing function f, the network transforms the input sequence $f(a) = \{f(a_1), f(a_2), ..., f(a_n)\}$ into the output sequence $f(b)$ [$\{f(b_1), f(b_2), ..., f(b_n)\}$

Proof: First of all we prove the claim that if f is a monotonically increasing function, then a single comparator with inputs $f(x)$ and $f(y)$ produces outputs $f(\min(x, y))$ and $f(\max(x, y))$.

To prove this claim, we consider a comparator whose input values are x and y. The upper output of comparator is min (x, y) and the lower output is max (x, y). Now we apply $f(x)$ and $f(y)$ to the inputs of the comparator. The opration of the comparator yields the value min $\{f(x) \cdot f(y)\}$ on the upper output and the value max $\{f(x), f(y)\}$ on the lower output. Since f is monotonically increasing, $x \le y$ implies $f(x) \le f(y)$. Then, we have the identifies:

$$\min\{f(x), f(y)\} = f(\min(x, y))$$
$$\max\{f(x), f(y)\} = f\{\max(x, y)\}$$

Thus, the comparator-produces the values f (min (x, y)) and f (max (x, y)) when $f(x)$ and $f(y)$ are its input, which completes the proof of claim.



Theorem. If a comparison network with n inputs sorts all 2^n possible sequences of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof: Suppose for the purpose of contradiction, the network sorts all zero-one sequences, but there exists a sequence of arbitrary numbers that the network does not correctly sort. That is, there exists an input sequence $(a_1, a_2, \ldots\ldots, a_n)$ containing elements a_i and a_j such that $a_i < a_j$, but the network places a_j before a_i in the output sequence. We define a monotonically increasing function f as

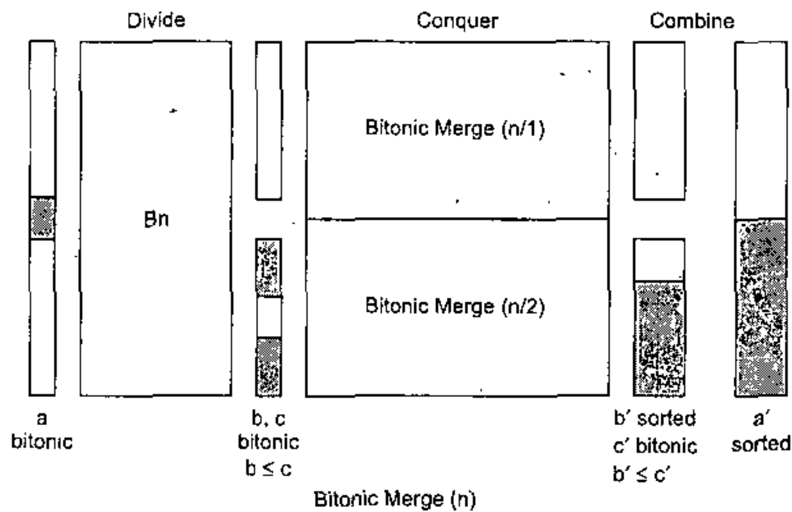
$$f(n) = \begin{cases} 0 & ; \text{ if } x \le a_i \\ 1 & ; \text{ if } x > a_i \end{cases}$$

Since the network places a_j before a_i in the output sequence when $(a_1, a_2, \ldots\ldots, a_n)$ is input, it places $f(a_j)$ before $f(a_i)$ in the output sequence when $(f(a_1), f(a_2), \ldots\ldots, f(a_n))$ is input. But $f(a_j) = 1$ and $f(a_i) = 0$, we obtain the contradiction that the network fails to sort the zero-one sequence $(f(a_1), f(a_2), \ldots\ldots, f(a_n))$ correctly.

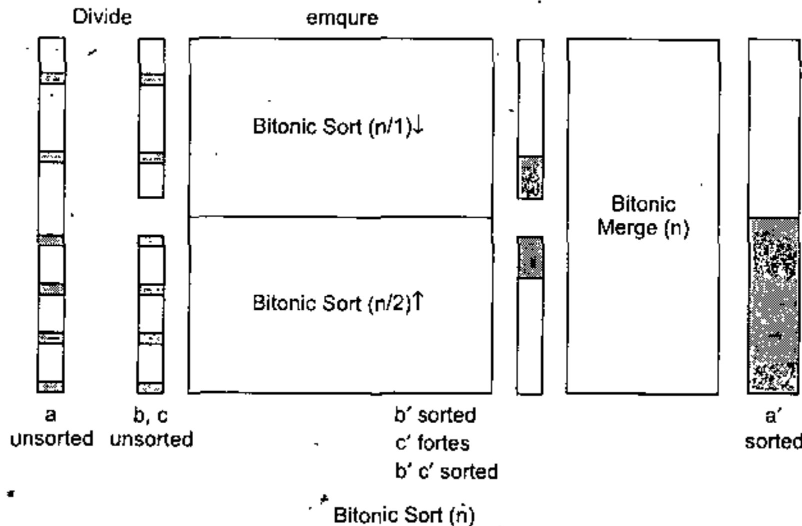
5.8 A BITONIC SORTING NETWORK

The building blocks of the sorting network Bitonic sort are comparator networks B_k with a different k, where k is a power of 2. By using the divide-and-conquer strategy, networks Bitonic Merge and Bitonic sort are formed.

First, a comparator network Bitonic Merge is built that sorts a bitonic sequence. Due to the theorm, B_n produces two bitonic subsequences, where all elements of the first are smaller or equal than those of the second. Therefore, Bitonic Merge can be built recursively as shown in the following figure.



The bitonic sequence necessary as input for Bitonic Merge is compared of two solid subsequences, where the first is in ascending and the other in descending order. The subsequences themselves and sorted by recursive, application of Bitonic sort.



Bitonic Sort (n)

Analysis: In order to form a sorted sequence of length n from two sorted sequences of length $\frac{n}{2}$, there are log (n) comparator stages required (e.g., the 3 = log (B) comparator stages to form sequence i from d and d'). The number of comparator stages T (n) of the entire sorting network is given by:

$$T(n) = T\left(\frac{n}{2}\right) + \log(n)$$

The solution of this recurrence is

$$T(n) = \log(n) + \log(n) - 1 + \log(n) - 2 + \ldots\ldots + 1 = \log(n) \cdot \frac{[\log(n)+1]}{2}$$

Each stage of the sorting network consists of $\frac{n}{2}$ comparators on the whole, there are $\theta(n \cdot \log(n)^2)$ comparators.

5.9 A MERGING NETWORK

Our sorting network will be constructed from merging networks, which are networks that can merge two sorted input sequences into one sorted output sequence. We modify BITONIC-SORTER (n) to create the merging network MERGER (n).

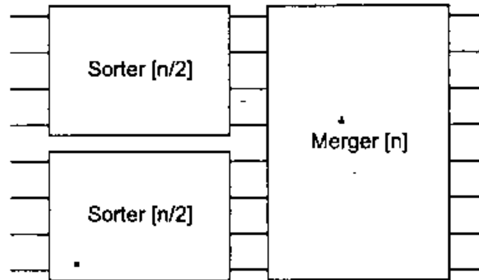
The merging network is based on the following intuition. Given two sorted sequences, if we reverse the order of the second sequence and then concatenate the two sequence, the result sequence is bitonic. For example, given the sorted zero-one sequences X = 00000111 and Y = 00001111, we reverse Y to get Y^R = 11110000. Concatinating X and Y^R yields 0000011111110000, which is bitonic. Thus, to merge the two input sequences X and Y, it suffices to perform a bitonic sort on X concatinated with Y^R.

5.10 A SORTING NETWORK

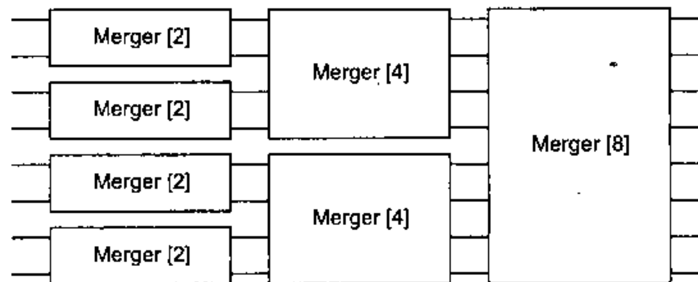
The sorting network SORTER (n) uses the merging network to implement a parallel version of merge sort. The construction and operation of the sorting network are the following:

NOTES

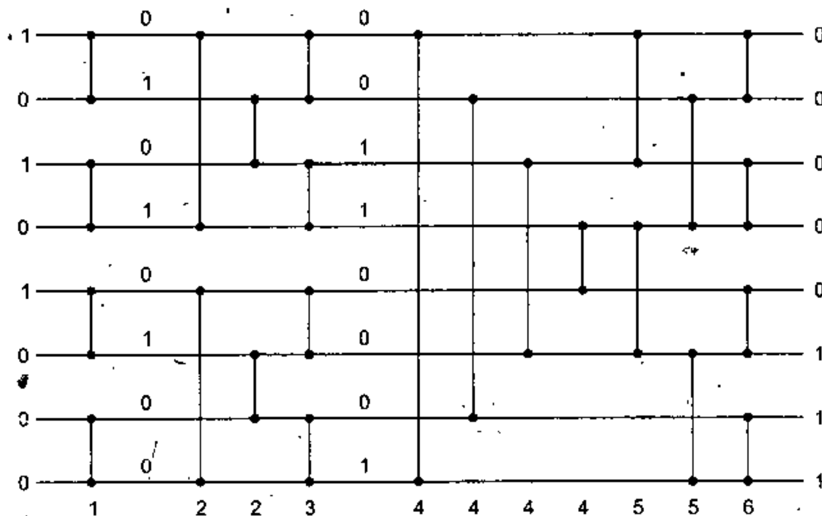
(a) The recursive construction: The n input elements are sorted by using two copies of SORTER $\left[\frac{n}{2}\right]$ recursively to sort (in parallel) two subsequences of length $\frac{n}{2}$ each. The two resulting sequences are then merged by MERGER $[n]$. The boundary case for the recursion is when $n = 1$, in which can we can use a wire to the sort the 1-element sequence, since a 1-element sequence is already sorted. The following figure shows the recursive construction of SORTER $[n]$.



(b) Unrolling the recursion: The following figure shows the result of unrolling the recursion.



(c) In the following figure, to replace the MERGER boxes with the actual merging networks. The depth of each comparator is indicated, and simple zero-one values are shown on the wires.



5.11 MATRIX OPERATIONS

NOTES

Operations on matrices are at the heart of scientific computing. Efficient algorithms for working with matrices are therefore of considerable practical interest. This chapter provides a brief introduction to matrix theory and matrix operations, emphasizing the problems of multiplying matrices and solving sets of simultaneous linear equations.

5.12 PROPERTIES OF MATRICES

Matrices and Vectors

A **matrix** is a rectangular array of numbers. For example,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \end{bmatrix}$$

is a 2×3 matrix $A = [a_{ij}]$, where for $i = 1, 2$ and $j = 1, 2, 3$, the element of the matrix in row i and column j is a_{ij}. We use uppercase letters to denote matrices and corresponding subscripted lowercase letters to denote their elements. The set of all $m \times n$ matrices with real-valued entries is denoted $\mathbf{R}^{m \times n}$. In general, the set of $m \times n$ matrices with entries drawn from a set S is denoted $\mathbf{S}^{m \times n}$.

Transpose of a Matrix

The **transpose** of a matrix A is the matrix A^T obtained by exchanging the rows and columns of A. For the above matrix A,

$$A^T = \begin{bmatrix} 1 & 7 \\ 3 & 9 \\ 5 & 11 \end{bmatrix}$$

In a **vector**, it is a one-dimensional array of numbers. For example,

$$x = \begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix}$$

is a vector of size 3. Lower case letters are used to denote vectors. For a vector x of size-n, i^{th} element is denoted by x_i, for $i = 1, 2, ..., n$. Standard form of a vector is **column vector** which is equivalent to an $n \times 1$ matrix. **Row vector** is transpose of column vector. e.g.,

$$x^T = [2 \quad 5 \quad 8]$$

The **unit vector** e_i is the vector whose i^{th} element is 1 and all of whose other elements are 0. So the size of a unit vector is clear from the context.

A matrix having all the entry as 0 known as **zero vector**. Such a matrix is often denoted 0, since the ambiguity between the number 0 and a matrix of 0's is usually resolved from context.

A matrix having equal number of rows as number of columns, known as **square matrix**.

For example, a $n \times n$ matrix is square matrix. Square matrices have some special cases:

NOTES

1. A **diagonal matrix** contains all the elements except diagonal elements with value 0.

   i.e., $a_{ij} = 0$, whenever $i \neq j$.

   So diagonal matix can be specified by listing the elements along the diagonal:

$$\text{diag.} \quad (a_{11}, a_{22}, \ldots\ldots, a_{nn}) = \begin{bmatrix} a_{11} & 0 & \cdots\cdots & 0 \\ 0 & a_{22} & \cdots\cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots\cdots & a_{nn} \end{bmatrix}$$

2. The $n \times n$ **identity matrix In** can be seen as a special case of diagonal matrix with 1's along the diagonal:

$$\text{In} = \text{diag. } (1, 1, \ldots\ldots, 1)$$

$$= \begin{bmatrix} 1 & 0 & \cdots\cdots & 0 \\ 0 & 1 & \cdots\cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots\cdots & 1 \end{bmatrix}$$

   i^{th} column of an identity matrix is the unit vector e_i.

3. A **tridiagonal matrix** T is one for which $t_{ij} = 0$ if $|i - j| > 1$. Non-zero entries appear only on the main diagonal, immediately above the main diagonal ($t_{i,\, i+1}$ for $i = 1, 2, \ldots\ldots, n - 1$), or immediately below the main diagonal ($t_{i+1,\, i}$ for $i = 1, 2, \ldots\ldots, n - 1$)

$$T = \begin{bmatrix} t_{11} & t_{12} & 0 & 0 & \cdots\cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots\cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots\cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & & 0 & t_{n,n-1} & t_{n,n} \end{bmatrix}$$

4. An **upper-triangular matrix** U is one for which $U_{ij} = 0$ if $i > j$. *i.e.*, all entries below the diagonal are zero.

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots\cdots & u_{1n} \\ 0 & u_{22} & \cdots\cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots\cdots & u_{nn} \end{bmatrix}$$

   An upper-triangular matrix is **unit upper-triangular if it has all 1's** along the diagonal.

5. A **lower-triangular matrix L** is one for which $l_{ij} = 0$ if $i < j$. *i.e.*, all entries above the diagonal are zero:

$$L = \begin{bmatrix} l_{11} & 0 & \cdots\cdots & 0 \\ l_{21} & l_{22} & \cdots\cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots\cdots & l_{nn} \end{bmatrix}$$

A lower-triangular matrix is **unit lower-triangular** if it has all 1's along the diagonal.

6. A matrix is known as **Permutation matrix P**, if it has exactly one 1 in each row or column and 0's elsewhere. An example of a permutation matrix is

NOTES

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

7. A **symmetric matrix A** satisfies the condition $A = A^T$. For example,

$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$ is a symmetric matrix.

Note: A matrix $A = a_{ij}$ for $i, j = 1, 2, \ldots\ldots, n$, is symmetric matrix if $a_{ij} = a_{ji}$

5.13 OPERATIONS ON MATRICES

The order of operations with matrices is the following:

1. Parenthesis
2. Exponents
3. Multiplication
4. Addition / Subtraction.

Procedure for addition of two matrices:

When adding matrices individual components are added.

$$\text{Matrix A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots\cdots \\ a_{21} & a_{22} & a_{23} & \cdots\cdots \\ a_{31} & a_{32} & a_{33} & \cdots\cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

$$\text{Matrix B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots \\ b_{21} & b_{22} & b_{23} & \cdots \\ b_{31} & b_{32} & b_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

$$\text{Matrix A + B} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + b_{13} & \cdots \\ a_{21} + a_{21} & a_{22} + b_{22} & a_{23} + b_{23} & \cdots \\ a_{31} + b_{31} & a_{32} + b_{32} & a_{33} + b_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

An example of adding a matrix is the following:

$$\text{Matrix A} = \begin{bmatrix} 3 & -2 & 5 \\ 4 & 1 & 6 \\ 3 & 0 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & -5 & 0 \\ 0 & 2 & -1 \\ 5 & 0 & 0 \end{bmatrix}$$

$$A + B = \begin{bmatrix} 3+1=4 & -2+(-5)=-7 & 5+0=5 \\ 4+0=4 & 1+2=3 & 6+(-1)=5 \\ 3+5=8 & 0+0=0 & 2+0=2 \end{bmatrix} = \begin{bmatrix} 4 & -7 & 5 \\ 4 & 3 & 5 \\ 8 & 0 & 2 \end{bmatrix}$$

NOTES

Procedure for the subtraction of two matrices:

When subtracting matrices individual components are subtracted

$$\text{Matrix A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots\cdots \\ a_{21} & a_{22} & a_{23} & \cdots\cdots \\ a_{31} & a_{32} & a_{33} & \cdots\cdots \\ \cdots & \cdots & \cdots & \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots \\ b_{21} & b_{22} & b_{23} & \cdots \\ b_{31} & b_{32} & b_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

$$\text{Matrix A} - \text{B} = \begin{bmatrix} a_{11} - b_{11} & a_{12} - b_{12} & a_{13} - b_{13} & \cdots\cdots \\ a_{21} - b_{21} & a_{22} - b_{22} & a_{23} - b_{23} & \cdots\cdots \\ a_{31} - b_{31} & a_{32} - b_{32} & a_{33} - b_{33} & \cdots\cdots \\ \cdots\cdots & \cdots\cdots & \cdots\cdots & \end{bmatrix}$$

An example of subtracting a matrix is the following:

$$\text{Matrix A} = \begin{bmatrix} 3 & -2 & 5 \\ 4 & 1 & 6 \\ 3 & 0 & 2 \end{bmatrix} \text{ and B} = \begin{bmatrix} 1 & -5 & 0 \\ 0 & 2 & -1 \\ 5 & 0 & 0 \end{bmatrix}$$

$$\text{Matrix A} - \text{B} = \begin{bmatrix} 3-1=2 & -2-(-5)=3 & 5-0=5 \\ 4-0=4 & 1-2=-1 & 6-(-1)=7 \\ 3-5=-2 & 0-0=0 & 2-0=2 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 5 \\ 4 & -1 & 7 \\ -2 & 0 & 2 \end{bmatrix}$$

Procedure for the multiplication of a matrix by a constant:

Multiply the constant through to each term in the matrix

$$\text{Matrix A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots\cdots \\ a_{21} & a_{22} & a_{23} & \cdots\cdots \\ a_{31} & a_{32} & a_{33} & \cdots\cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

Let $c = a$ constant, therefore the final matrix is

$$\begin{bmatrix} ca_{11} & ca_{12} & ca_{13} & \cdots\cdots \\ ca_{21} & ca_{22} & ca_{23} & \cdots\cdots \\ ca_{31} & ca_{32} & ca_{33} & \cdots\cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

An example of multiplication of a matrix by a constant:

$$\text{Matrix A} = \begin{bmatrix} 3 & -2 & 5 \\ 4 & 1 & 6 \\ 3 & 0 & 2 \end{bmatrix}. \text{ Find matrix 2A.}$$

i.e.,

$$2\begin{bmatrix} 3 & -2 & 5 \\ 4 & 1 & 6 \\ 3 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 2\times 3=6 & 2\times -2=-4 & 2\times 5=10 \\ 2\times 4=8 & 2\times 1=2 & 2\times 6=12 \\ 2\times 3=6 & 2\times 0=0 & 2\times 2=4 \end{bmatrix} = \begin{bmatrix} 6 & -4 & 10 \\ 8 & 2 & 12 \\ 6 & 0 & 4 \end{bmatrix}$$

Procedure for the multiplication of two matrices:

When multiply matrices the following procedure is used.

$$\text{Matrix A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ a_{21} & a_{22} & a_{23} & \cdots \\ a_{31} & a_{32} & a_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix} \text{ and B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots \\ b_{21} & b_{22} & b_{23} & \cdots \\ b_{31} & b_{32} & b_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

Step 1. Find the dimensions of the individual matrices.

For Matrix A* Matrix B (AB)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ a_{21} & a_{22} & a_{23} & \cdots \\ a_{31} & a_{32} & a_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots \\ b_{21} & b_{22} & b_{23} & \cdots \\ b_{31} & b_{32} & b_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix} = \text{Resultant Matrix}$$

Dimension = row versus column; Dimension = row versus column; Dimension = (row of 1st matrix versus column of 2nd matrix).

Step 2. Multiply the row of the first matrix versus the column of the second matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ a_{21} & a_{22} & a_{23} & \cdots \\ a_{31} & a_{32} & a_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots \\ b_{21} & b_{22} & b_{23} & \cdots \\ b_{31} & b_{32} & b_{33} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}*b_{11}+a_{12}*b_{21}+a_{13}*b_{31}\cdots & a_{11}*b_{12}+a_{12}*b_{22}+a_{13}*b_{31}\cdots & a_{11}*b_{13}+a_{12}*b_{23}+a_{13}*b_{33}\cdots \\ a_{21}*b_{11}+a_{22}*b_{21}+a_{23}*b_{31}\cdots & a_{21}*b_{22}+a_{22}*b_{22}+a_{23}*b_{32}\cdots & a_{21}*b_{13}+a_{22}*b_{23}+a_{23}*b_{33}\cdots \\ a_{31}*b_{11}+a_{32}*b_{21}+a_{33}*b_{31}\cdots & a_{31}*b_{32}+a_{32}*b_{22}+a_{33}*b_{33}\cdots & a_{31}*b_{13}+a_{32}*b_{23}+a_{33}*b_{33}\cdots \end{bmatrix}$$

Step 3. Sum up each of the individual components in the answer. (This will be your final answer).

An example of multiplication of a matrix by another matrix:

$$\text{Matrix A} = \begin{bmatrix} 3 & -2 & 5 \\ 4 & 1 & 6 \\ 3 & 0 & 2 \end{bmatrix} \text{ and B} = \begin{bmatrix} 1 & -5 & 0 \\ 0 & 2 & -1 \\ 5 & 0 & 0 \end{bmatrix}$$

Step 1. Find the dimension of the individual matrices.

For Matrix A × Matrix B (AB)

$$\begin{bmatrix} 3 & -2 & 5 \\ 4 & 1 & 6 \\ 3 & 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & -5 & 0 \\ 0 & 2 & -1 \\ 5 & 0 & 0 \end{bmatrix} = \text{Resultant Matrix.}$$

3 rows by 3 columns; 3 rows by 3 columns; therefore the resultant matrix will be 3 by 3.

Step 2. Multiply the row of the first matrix versus the column of the second matrix.

$$\begin{bmatrix} 3 & -2 & 5 \\ 4 & 1 & 6 \\ 3 & 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & -5 & 0 \\ 0 & 2 & -1 \\ 5 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 3(1)+(-2)(0)+5(5) & 3(-5)+2(2)+5(0) & 3(0)+2(-1)+5(0) \\ 4(1)+1(0)+6(5) & 4(-5)+1(2)+6(0) & 4(0)+1(-1)+6(0) \\ 5(1)+0(0)+2(5) & 3(-5)+0(2)+2(0) & 3(0)+0(-1)+2(0) \end{bmatrix}$$

Step 3. Sum up each of the individual component in the answer $\begin{bmatrix} 28 & -19 & 2 \\ 34 & -18 & -1 \\ 13 & -15 & 0 \end{bmatrix}$ by 3 columns.

5.14 STRASSEN'S ALGORITHM FOR MATRIX MULTIPLICATION

Strassen's algorithm can be viewed as an application of a familiar design technique: **divide and conquer.** Suppose we want to calculate the product C = AB, which each of A, B and C are $n \times n$ matrices. Assuming that n is an exact power of 2, we divide each of A, B, and C into four $\frac{n}{2} \times \frac{n}{2}$ matrices, now rewrite the equation C = AB as follows:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

There are four equation corresponding to the above, which are the following:

$$r = ae + bg$$
$$s = af + bh$$
$$t = ce + dg$$
$$u = cf + dh.$$

Each of these four equations specifies two multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and addition of their $\frac{n}{2} \times \frac{n}{2}$ products. Using these equations to define a straight forward divide and conquer strategy, we find the recurrence for the time T (n) to multiply two $n \times n$ matrices:

$$T(n) = 8T\left(\frac{n}{2}\right) + \theta(n^2)$$

Using master method, we know

$$a = 8, b = 2, f(n) = n^2$$

$$n^{\lg b^a} \doteq n^{\lg_2^8} = n^{\lg_2^{2^3}} = n^3$$

Note $\lg_2^2 = 1$.

Since $\quad f(n) = O\left(n^{\lg_2^{8-\epsilon}}\right)$, where $\epsilon = 1$.

We can apply case 1 of master method and recurrence has the solution $T(n) = \theta(n^3)$, and thus this method is no faster than the ordinary one.

Strassen discovered a different recursive approach that requires only 7 recursive multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and $\theta(n^2)$ scalar additions and subtractions, the recurrence is

$$T(n) = 7T\left(\frac{n}{2}\right) + \theta\left(n^2\right)$$

Apply master method,

$$a = 7, b = 2\ f(n) = n^2$$
$$n^{\lg_b^a} = n^{\lg_2^7} = n^{2.81}$$
$$T(n) = \theta(n^{\lg 7})$$
$$T(n) = O(n^{2.81}).$$

NOTES

Strassen's method has four steps:

1. Divide the input matrices A and B into $\frac{n}{2} \times \frac{n}{2}$ submatrices.
2. Using $\theta(n^2)$ scalar additions and subtractions, compute the matrices A_1, B_1, A_2, B_2, ... A_7, B_7, each of which is $\frac{n}{2} \times \frac{n}{2}$.
3. Recursively compute the seven matrix products
$$P_i = A_i B_i \text{ for } i = 1, 2, ..., 7.$$
4. Compute the desired submatrices r, s, t, u, of the result matrix C by adding and / or subtracting various combinations of the P_i matrices, using only $\theta(n^2)$ scalar addition and subtractions.

Suppose, $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$ and $C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$

We know, $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$

In strassen method, we calculate the 7 $\frac{n}{2} \times \frac{n}{2}$ matrices. We get,

$$P = (a_{11} + a_{22})(b_{11} + b_{22}) \qquad Q = (a_{21} + a_{22})\, b_{11}$$
$$R = a_{11}(b_{12} - b_{22}) \qquad S = a_{22}(b_{21} - b_{11})$$
$$T = (a_{11} + a_{12})\, b_{22} \qquad U = (a_{21} - a_{11})(b_{11} + b_{12})$$
$$V = (a_{12} - a_{22})(b_{21} + b_{22})$$

Then, we find c_{ij}'s using the formula

$$c_{11} = P + S - T + V \qquad c_{12} = R + T$$
$$c_{21} = Q + S \qquad c_{22} = P + R - Q + U$$

Example. *Use strassen's algorithm to compute the matrix product* $\begin{pmatrix} 1 & 8 \\ 3 & 5 \end{pmatrix}\begin{pmatrix} 7 & 4 \\ 6 & 2 \end{pmatrix}$. *Show your work.*

Sol.

$$P = (1 + 8)(7 + 2) = 81 \qquad Q = (3 + 8)\, 7 = 77$$
$$R = 1\,(7 - 2) = 5 \qquad S = 8\,(6 - 7) = -8$$
$$T = (1 + 5)\, 2 = 12 \qquad U = (3 - 1)(7 + 4) = 22$$
$$V = (5 - 8)(6 + 2) = -24.$$

Now we find C,

$$c_{11} = P + S - T + V = 81 + (-8) - 12 - 24 = 37$$
$$c_{12} = R + T = 5 + 12 = 17$$
$$c_{21} = Q + S = 77 - 8 = 69$$
$$c_{22} = P + R - Q + U = 81 + 5 - 77 + 22 = 31$$

Then, $\begin{pmatrix} 1 & 5 \\ 3 & 8 \end{pmatrix}\begin{pmatrix} 7 & 4 \\ 6 & 2 \end{pmatrix} = \begin{pmatrix} 37 & 17 \\ 69 & 31 \end{pmatrix}$

NOTES

5.15 POLYNOMIALS AND THE FFT

The straight forward method of adding two polynomials of degree n takes $\theta(n)$ time, but the straight forward method of multiplying them takes $\theta(n^2)$ time, while as Fast Fourier Transformation or FFT, can reduce the time to multiply polynomials to $\theta(n \lg n)$.

Polynomials

A polynomial in the variable x over field F is a representation of a function A (x) as a formal sum:

$$A(x) = \sum_{j=0}^{n-1} a_j x_j$$

Where n is the degree-bound $a_0, a_1, \ldots, a_{n-1}$ are the coefficients of the polynomial, *i.e.*, $a_j = 0$ for $j > k$. A polynomial with degree bound n can have a degree k, $0 < k \le n-1$. A polynomial with degree k has degree-bound n for all $n > k$.

Sum: *Consider* A (x) *and* B (k), *two polynomials for degree bound* n.

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \quad B(x) = \sum_{j=0}^{n-1} b_j x^j$$

C (x) *is the sum of* A (x) *and* B (x), *it has degree bound* n, *and* $C(x) = A(x) + B(x)$ for all $x \in$ F. We can write

$$C(x) = \sum_{j=0}^{n-1} c_j x^j \text{, where } c_j = a_j + b_j$$

Products: C (x) is the product of A (x) and B (x) if $C(x) = A(x)\,B(x)$ for all $x \in$ F and is of degree-bound $n - 1$. Multiplication is **"school book"** style:

$$A(x) = 3x^3 + 2x + 1$$
$$B(x) = 4x^2 + 3x - 1$$

$$\begin{array}{r} 3x^3 + 2x + 1 \\ 4x^2 + 3x - 1 \\ \hline -3x^3 - 2x - 1 \\ 9x^4 + 6x^2 + 3x \\ 12x^5 + 8x^3 + 4x^2 \\ \hline 12x^5 + 9x^4 + 5x^3 + 10x^2 + x - 1 \end{array}$$

Another way to express the product C (x) is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \text{, where } c_j = \sum_{k=0}^{j} a_k b_{j-k}.$$

Note that degree (C) = degree (A) + degree (B)

degree – bound (C) = degree – bound (A) + degree – bound (B) – 1

≤ degree – bound (A) + degree – bound (B).

We will study a fast, recursive, algorithms to compute polynomial products using the discrete Fourier transform (DFT) and the Fast Fourier Transform (FFT).

5.16 REPRESENTATION OF POLYNOMIALS

NOTES

Polynomial $\qquad A(x) = \sum_{j=0}^{n-1} a_j x^j$

yields the coefficient representation of A (x) with degree-bound n is a vector of coefficients $a = (a_0, a_1, ..., a_{n-1})$. Consider, given $a_0, a_1, ..., a_{n-1}$ as a representation of A (x) evaluation A (x_0). We use **Homer's rule** to do so in $\theta(n)$:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + x_0(a_3 + ... + x_0(a_{n-2} + x_0 a_{n-1})) .....))$$

This can easily be written as an iterative algorithm with a single loop of length n (0 to $n - 1$). Adding A (x) and B (x) is again $\theta(n)$. $(c_0, c_1, ..., c_{n-1})$ with $c_j = a_j + b_j$. What about the multiplication of two degree-bound n polynomials A (x) and B (x)? Polynomial multiplication via the "school book algorithm is $\theta(n^2)$, and $c = (c_0, c_1, ..., c_{n-2})$ is obtained through convolution:

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

C = A ⊗ B (means convolution). We study fast algorithm for convolution ! A point-value representation of A (x) a polynomial of degree-bound n point-value pairs:

$$\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$$

with all the x's distinct and $y_j = A(x_j)$. Note that any set of n distinct x's can be used, so there are many point-value representations of A (x). By using Homer's rule, one can evoluate A (x) at a point in $\theta(n)$ time, so conversion from coefficient point-value takes $\theta(n^2)$ time using way to obtain a $\theta(n \lg n)$ algorithm. Given the point-value representation and computing the coefficient representation of A (x) is called interpolation.

Theorem. For any set $\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ of n point-value pairs there is unique degree-bound n polynomial, A (x) such that $y_i = A(x_i)$ for $i = 0, ..., n - 1$.

Proof: Write out $\qquad A(x_0) = y_0, A(x_1) = y_1, ..., A(x_{n-1}) = y_{n-1}$ as

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

The matrix $v(x_0, x_1, ..., x_{n-1})$ is called a vandemonde matrix and this matrix has determinant

$$\prod_{0 \le j < k \le n-1} (x_k - x_j)$$

determinant $(x_0, x_1, ..., x_{n-1}) \neq 0$ since all the x's are distinct. So the system has a solution:

$$a = V(x_0, x_1, ..., x_{n-1})^{-1} y$$

so the a's can be uniquely found. Solving such a linear system takes $\theta(n^3)$ time, but we can interpolate faster.

LaGrange Formula: Given point-value,

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$

can show that A (x) is a degree-bound n polynomial and A $(x_i) = y_i$. This takes $\theta(n^2)$ time.

$$A(x) \rightarrow \{(x_0, y_0), (x_1, y_1) \ldots\ldots, (x_{n-1}, y_{n-1})\}$$

$$B(x) \rightarrow \{(x_0, y'_0), (x_1, y'_1), \cdots\cdots, (x_{n-1}, y'_{n-1})\}$$

Consider $C(x) = A(x) + B(x)$ so

$$C(x) \rightarrow \{(x_0 + y_0 + y'_0), (x_1 + y_1 + y'_1), \cdots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$$

and conversion takes $\theta(n)$ time. What about multiplication of A (x) and B (x) both degree-bound n polynomials?

C (x) = A (x) B (x) is a degree-bound $2n - 1$, so we must augment the number of point-value pairs used to represent A (x) and B (x).

$$A(x) \rightarrow \{(x_0, y_0), (x_1, y_1), \ldots\ldots, (x_{2n-2}, y_{2n-2})\}$$

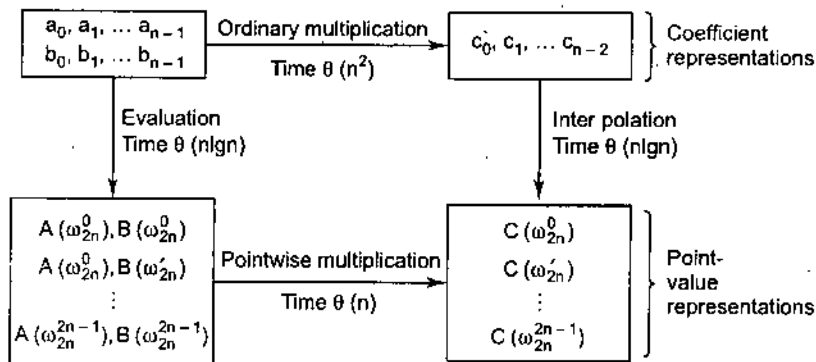
$$B(x) \rightarrow \{(x_0, y'_0), (x_1, y'_1), \cdots\cdots, (x_{2n-2}, y'_{2n-2})\}$$

$$C(x) \rightarrow \{(x_0, y_0\, y'_0), (x_1, y_1\, y'_1), \cdots\cdots, (x_{2n-2}, y_{2n-2}\, y'_{2n-2})\}$$

is $\theta(n)$ time. If we can find a fast way to convert coefficients of point-value pairs, use the $\theta(n)$ multiplication of point-value pairs, and convert back, we will have a very fast algorithm. We do so by choosing the points for the point-value pairs very carefully. These points are complex root of unity.

To multiply A (x) B (x) we:

1. Use the coefficient representation, but up to degree bound $2n$ by
2. Compute the point-value of A (x) and B (x) at the $2n$ roots of unity via the FFT.
3. Do print wise multiplication
4. Interpolate back from the 2nd roots of unity to a coefficient representation via inverse DFT.



This figure shows a graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, while those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)^{th}$ root of unity.

5.17 THE DFT AND FFT

A **complex roots of unity** (n^{th} root) is a complex number w such that:

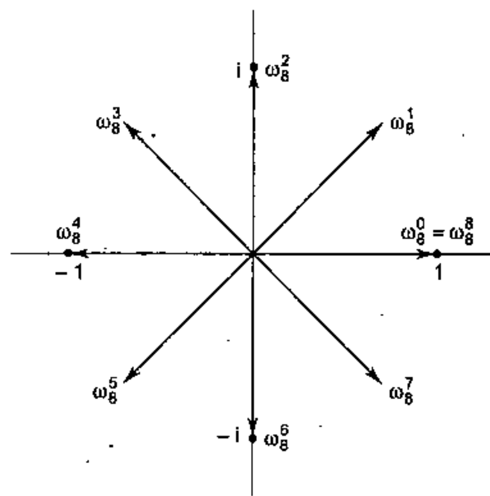
$$\omega^n = 1.$$

There are n, n^{th} roots of unity.

$e^{2\pi ik/n}$, for $k = 0, 1, \ldots\ldots, n-1$.

recall $e^{iu} = \cos(u) + i\sin(u)$ is a point on the complex unit circle.

NOTES



This figure shows the value of $\omega_8^0, \omega_8^1, \cdots\cdots \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principle 8th root of unity.

$$\omega_n = e^{2\pi i/n}$$

is the principal n^{th} root of unity. $\omega_n^0, \omega_n^1, \omega_n^2, \cdots\cdots \omega_n^{n-1}$ are the n, n^{th} roots of unity. They form a group under multiplication, as the additive group $(z_{n_1} +)$ modulo n.

Since $\quad \omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{(j+k) \bmod n}$

Similarly, $\quad \omega_n^{-1} = \omega_n^{n-1}$.

Cancellation Lemma:

For all integers $n, k \geq 0$, $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k$$

Proof:

$$\omega_{dn}^{dk} = \left(e^{2\pi i/dn}\right)^{dk}$$

$$= \left(e^{2\pi i/n}\right)^k = \omega_n^k$$

Corollarly for $n > 0$, even integer $\omega_n^{n/2} = \omega_2 = -1$.

Halving Lemma:

If $n > 0$ is even, then the squares of the n, n^{th} roots of unity, are the $\frac{n}{2}, \frac{n}{2}$ th root of unity.

$$\left(\omega_n^k\right)^2 = \left(e^{2\pi i/n}\right)^{2k} = \left(e^{\frac{2\pi i}{n/2}}\right)^k = \omega_{n/2}^k$$

We get them twice are

$$\left(\omega_n^{k+n/2}\right)^2 = \omega_n^{2k+n} = \omega_n^{2k}\,\omega_n^n = \omega_n^{2k} = \left(\omega_n^k\right)^2$$

NOTES

So $\left(\omega_n^{k+n/2}\right)^2 = \left(\omega_n^k\right)^2$ and $\omega_n^{k+n/2} = \omega_n^k\,\omega_n^{n/2} = -\,\omega_n^k$

Summation Lemma:

For all $n \geq 1$, $k > 0$ not divisible by n we have

$$\sum_{j=0}^{n-1}\left(\omega_n^k\right)^j = 0$$

Proof: Note that this is a geometric series with $z = \omega_n^k$, then recall

$$\sum_{j=0}^{n-1} z^j = \frac{\left(z^n - 1\right)}{(z-1)}$$

here $z = \omega_n^k$ so sum equals $\dfrac{\left(\left(\omega_n^k\right)^n - 1\right)}{\left(\left(\omega_n^k\right) - 1\right)}$. Since k does not divide

$n\left(\omega_n^k\right) \neq 1$, but $\left(\omega_n^k\right)^n = \left(\omega_n^k\right)^n = \left(\omega_n^n\right)^k = 1^k = 1$. So,

$$\sum_{j=0}^{n-1}\left(\omega_n^k\right)^j = 0$$

The Discrete Fourier Transform (DFT)

We want to evaluate

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

at the nth root of unity. $\omega_n^0, \omega_n^1, \omega_n^2, \cdots\cdots, \omega_n^{n-1}$. Consider n as a power-of-two, n is the degree bound, and can be achieved with zero padding. Using the coefficient representation of A, $a = (a_0, a_1, \ldots\ldots, a_{n-1})$, we want

$$y_k = A\left(\omega_n^k\right) = \sum_{j=0}^{n-1} a_j\,\omega_n^{k-j}, \text{ for } k = 0, 1, \ldots\ldots, n-1.$$

The vector $y = (y_0, y_1, \ldots\ldots, y_{n-1})$ is called the **Discrete Fourier Transform (DFT)** of the vector a. We also write,

$$y = \text{DFT}_n(a)$$

where n is the size of the vector.

The Fast Fourier Transform

Use a divide and conquer strategy, since n is a power of 2, we can split the A (x) polynomial into a polynomial of even and old powers:

Even coefficients: $A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots\ldots + a_{n-2}\, x^{\frac{n}{2}-1}$

Odd coefficients: $A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots\ldots + a_{n-1}\, x^{\frac{n}{2}-1}$

We can get A (x) back as follows:

$$A(x) = A^{[0]}(x^2) + x\,A^{[1]}(x^2).$$

NOTES

So to evaluate A (x) at $\omega_n^0, \omega_n^1, \cdots \omega_n^{n-1}$ reduce to evaluating A$^{[0]}$ (x) at $\left\{\left(\omega_n^0\right)^2, \left(\omega_n^1\right)^2, \cdots \left(\omega_n^{n-1}\right)^2\right\}$ and A$^{[1]}$ (x) at the same points, and then combine using. Note by the having lemma $\left\{\left(\omega_n^0\right)^2, \left(\omega_n^1\right)^2, \cdots \left(\omega_n^{n-1}\right)^2\right\}$ are the $\frac{n}{2}, \frac{n}{2}$ th roots of unity twice. So A (x) is evaluated at n points A$^{[0]}$ (x) and A$^{[1]}$ (x) are evaluated at $\frac{n}{2}$ points, but we can recursively continue, dividing by 2 the whole way.

Recurisve - FFT (a)

1. $n \leftarrow$ length $[a]$ ‖ n is a power of 2.
2. if $n = 1$
3. then return a
4. $\omega_n \leftarrow e^{2\pi i/n}$
5. $w \leftarrow 1$
6. $a^{[0]} \leftarrow \{a_0, a_2, \ldots\ldots, a_{n-2}\}$
7. $a^{[1]} \leftarrow \{a_1, a_3, \ldots\ldots, a_{n-1}\}$
8. $y^{[0]} \leftarrow$ Recursive-FFT $(a^{[0]})$
9. $y^{[1]} \leftarrow$ Recursive-FFT $(a^{[1]})$
10. for $k \leftarrow 0$ to $\frac{n}{2} - 1$
11. do $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
12. $y_{k+\frac{n}{2}} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
13. $\omega \leftarrow \omega\, \omega_n$
14. return y ‖ y is assumed to be a column vector.

This is a divide-and-conquer having the size of the FFT each time. Note that the DFT of a single element is $y_0 = a_0\, \omega_i = a_0$, identity. Lines 6 and 7 setup the A$^{[0]}$ and A$^{[1]}$ coefficients, lines 8 and 9 are the recursion, lines 10 – 13 do the recombination, and lines 4 and 5 make sure the right n^{th} root of unity is used. Line 11 does the first half, 0 to $\frac{n}{2} - 1$, line 12 does the second half, $\frac{n}{2}$ to $n - 1$. Note that ω used in this loop is ω_n^k but $\omega_n^{k+n/2} = \omega_n^k\, \omega_n^{n/2} = -\,\omega_n^k$.

The running time satisfies: $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$, so $T(n) = \theta(n \lg n)$ where the 2T is the 2FFT's and the $\theta(n)$ is loops 10 – 13.

Interpolation

The DFT can be written as

$$y = V_n\left(\omega_n^0, \omega_n^1, \omega_n^2, \cdots\cdots, \omega_n^n\right) a$$

the (k, j) entry of V_n is $\quad V_n\big|_{(k,j)} = \omega_n^{kj}$ we can interpolate by inverting:

$$a = V_n^{-1}\left(\omega_n^0, \omega_n^1, \omega_n^2, \cdots\cdots, \omega_n^n\right) y$$

$$= \text{DFT}_n^{-1}(y)$$

NOTES

But it is easy to compute V_n^{-1}.

Theorem: For j, $k = 0, 1, ..., n-1$, $V_n^{-1}\Big|_{(k,j)} = \dfrac{\left(\omega_n^{-kj}\right)}{n}$.

Proof: We show that with this definition $V_n^{-1} V_n = I$, the identity matrix, or $V_n^{-1} V_n \Big|_{(j,j')} = \delta(j, j') = \{1$ if $j = j'$ or 0 otherwise

$$= \sum_{k=0}^{n-1} \left(\omega_n^{-kj/n}\right)\left(\omega_n^{kj'}\right)$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{k(j'-j)}$$

if $j < > j'$ then

$$\sum_{k=0}^{n-1} \omega_n^{k(j'-j)} = \sum_{k=0}^{n-1} \left(\omega_n^{(j'-j)}\right)^k = 0$$

By the summation lemma, if $j' = j$, then $\omega_n^{k(j'-j)} = \omega_n^0 = 1$ so,

$$\frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{k(j'-j)} \ \frac{1}{n}\sum_{k=0}^{n-1} 1 = \frac{n}{n} = 1$$

thus $a_j = \dfrac{1}{n}\displaystyle\sum_{k=0}^{n-1} y_k \left(\omega_n^{-kj}\right)$, which is like a DFT except:

1. The role of a and y are reversed.
2. ω_n is replaced by ω_n^{-1}.
3. The result is divided by n.

Thus, we can perform this in $\theta(n \lg n)$ with a suitably modified Recursive-FFT. Call this

$$a = \mathrm{DFT}_n^{-1}(y).$$

5.18 NUMBER-THEORETIC ALGORITHMS

When analyzing number theoretic algorithms, the input is often one or a few integer Thus, the size of integer is important. And, since many algorithms are based on bits operations, we usually

- measure the integer n's size in bits: $\lg n$
- consider elementary operations in terms of bit operations β-bit numbers.
- multiplication $\theta(\beta^2)$
- division $\theta(\beta^2)$
- addition $\theta(\beta)$.

5.19 ELEMENTARY NUMBER THEORETIC NOTIONS

NOTES

- Z = {0, ± 1, ± 2 , ...} is set of integers
  N = {0, 1, 2, ...} natural or counting numbers.
- **Divisibility**
  $\frac{d}{a}$ " d divides a" means
  $\exists k \in Z$ such that $a = kd$
- All integers $\frac{d}{0}$
  if $\frac{d}{b}$ then b is a multiple of d.
  If d does not divide b, then we have d and b
  If $\frac{d}{b}$ and $d \geq 0$, then d is a divisor of b $\frac{d}{b} \Leftrightarrow -\frac{d}{b}$.
  So why not choose divisors as positive.
  - If d is a divisor of b, then
    $1 \leq d \leq | b |$, *e.g.*, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12 and 24.
  - Every integer b is divisible by 1 and b, the trivial divisors. Non-trivial divisors of b are called factors of b.
    Factors of 24 are 2, 3, 4, 6, 8 and 12.
  - **Definition.** An integer P > 1 with only trivial divisors is a prime number, or prime.
    Small primes: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...
    There are infinitely many primes. Any positive integers that is not prime is **composite**.
    1 is the "unit" for multiplication and is neither prime nor composite.
  - **Division Theorem.** For any integer b, and positive integer n, $\exists$ integers q and r such that

$$0 \leq r < n \text{ and } b = qn + r$$

$$q = \left\lfloor \frac{b}{n} \right\rfloor \text{ is called the } \textbf{quotient}.$$

$$r = b \text{ (mod } n) \text{ is the } \textbf{remainder}.$$

  Note: $\frac{n}{b} \Leftrightarrow b \text{ (mod } n) \equiv 0 \text{ or } r = 0$

- $$b = \left\lfloor \frac{b}{n} \right\rfloor n + [b \text{ (mod } n)]$$

$$[b \text{ (mod } n)] = b - \left\lfloor \frac{b}{n} \right\rfloor n$$

  if $a \text{ (mod } n) = b \text{ (mod } n)$ we write

$$a \equiv b \text{ (mod } n)$$

  "a is equivalent to b modulo n".

NOTES

Thus, a and b have the same remainder, w.r.t. n and so

$$a = q_a n + r$$
$$b = q_b n + r$$

$$b - a = (q_a - q_b)\, n + r - r = (q_a - q_b)\, n \text{ so } \frac{n}{(b-a)}$$

- The equivalence class module n and b:

$$[b]_n = \{b + kn : k \in Z\}, \text{ e.g.,}$$
$$[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$$
$$[-4]_7 = ...$$
$$[17]_7 = ...$$

- $Z_n = \{[a]_n : 0 \le a \le n - 1\}$ and we often write
  $Z_n = \{0, 1, 2, 3, ..., n - 1\}$

associated a with $[a]_n$

$-1 \in [n-1]_n$ since $n - 1 \pmod{n} = -1$

Common Divisors and Greatest Common Divisors

If $\frac{d}{a}$ and $\frac{d}{b}$, then d is a common divisor of a and b.

Lemma 1. If $\frac{d}{a}$ and $\frac{d}{b}$, then $\frac{d}{(a+b)}$ and $\frac{d}{(a-b)}$ and also $\frac{d}{(ax+by)}$ for any $x, y \in z$ if $\frac{a}{b}$ then either

$$|\, a \,| \le |\, b \,| \text{ or } b = 0$$

This means that $\frac{a}{b}$ and $\frac{b}{a} \Rightarrow a = \pm b$

The greatest common divisor of a and b, not both zero, is the largest common divisor:

$gcd\ (24, 30) = 6$

$gcd\ (5, 7) = 1$

$gcd\ (0, 9) = 9$ (all integers divide 0)

$1 \le gcd\ (a, b) \le \min\ (\,|\, a \,|, |\, b \,|\,)$

$gcd\ (0, 0) = 0$ by definition.

Some elementary gcd properties are the following:

1. $gcd\ (a, b) = gcd\ (b, a)$
2. $gcd\ (a, b) = gcd\ (-a, b)$
3. $gcd\ (a, b) = gcd\ (|\, a \,|, |\, b \,|)$
4. $gcd\ (a, 0) = |\, a \,|$
5. $gcd\ (a, ka) = |\, a \,|$ for any $k \in Z$.

Theorem. If a and b are integers and not both zero, then $gcd\ (a, b)$ is the smallest positive element of $\{ax + by : x, y \in Z\}$.

Proof: Let S be the smallest positive element of $\{ax + by : x, y \in Z\}$, with $q = \left[\frac{a}{S}\right]$, we have

$$a \text{ (mod S)} = a - q\text{S}$$
$$= a - q\,(ax + by)$$
$$= a\,(1 - qx) + b\,(-qy)$$

Now a (mod S) < S and is a linear combination of a and b, so a (mod S) = 0, thus

NOTES

$\frac{\text{S}}{a}$ and replacing a with b, $\frac{\text{S}}{a}$. Thus, S is a common factor of a and b so

$$gcd\,(a, b) \geq \text{S} \quad \text{by lemma 1}$$

We have gcd $\frac{(a,b)}{\text{S}_1}$ but since S > 0 means

$$gcd\,(a, b) \leq \text{S} \Rightarrow gcd\,(a, b) = \text{S}.$$

Corollary: $\forall\, a, b \in \text{Z}$ and $n \in \text{N}$

$$gcd\,(an, bn) = n\, gcd\,(a, b).$$

Proof: If $n = 0$, $gcd\,(0, 0) = 0$ is the definition.

If $n > 0$ then $gcd\,(an, bn)$ is the smallest of the $\{anx + bny\}$ numbers and so this is n times the smallest $\{ax + by\}$ which is $gcd\,(a, b)$.

Relatively Prime Integers

Two integers a, b are said to be **relatively prime** if their only common divisor is 1, that is, if $gcd\,(a, b) = 1$. For example, 8 and 15 are relatively prime, since the divisors of 8 are 1, 2, 4 and 8, while the divisors of 15 are 1, 3, 5 and 15.

Theorem. For any integers a, b and P, if both $gcd\,(a, \text{P}) = 1$ and $gcd\,(b, \text{P}) = 1$, then $gcd\,(ab, \text{P}) = 1$.

Proof: There exist integers x, y, x' and y' such that

$$ax + \text{P}y = 1$$
$$bx' + \text{P}y' = 1, \text{ then}$$
$$(ax + \text{P}y)\,(bx' + \text{P}y') = ab\,(xx') + \text{P}\,(ybx' + y'ax + \text{P}yy') = 1.$$

This is a positive linear combination of ab and P, since it is 1, the minimum value, it must be $gcd\,(ab, \text{P}) = 1$.

We say $n_1, \ldots\ldots, n_k$ are "pair wise relatively primes" if whenever $i \neq j$ $gcd\,(n_i, n_j) = 1$.

Unique Factorization

Theorem. For all prime P_1 $\forall\, a, b \in \text{Z}$, then $\frac{\text{P}}{a}$ or $\frac{\text{P}}{b}$.

Proof: Assume $\frac{\text{P}}{ab}$, but P ∦ a and P ∥ b.

Thus $gcd\,(a, \text{P}) = gcd\,(b, \text{P}) = -1$, thus $gcd\,(ab, \text{P}) = 1$, which contradicts $\frac{\text{P}}{ab}$ since

$$\frac{\text{P}}{ab} \Rightarrow gcd\,(ab, \text{P}) = \text{P}$$

This is contradiction.

5.20 GREATEST COMMON DIVISOR

Since $gcd\,(a, b) = gcd\,(|a|, |b|)$, we consider $gcd\,(a, b)$ with $a, b \in \text{N}$. By prime factorization

$$a = \Pi P_i^{l_{ai}}$$

$$b = \Pi P_i^{l_{bi}}$$

NOTES

Then $gcd\ (a, b) = \Pi P_i^{\min(l_{ai}, l_{bi})}$

However, factoring integers is **HARD**, so this is impractical. Instead we will derive a "fast" algorithm. Euclid's algorithm (recall an old algorithm).

Theorem (GCD recursion theorem): For any non-negative integer a and any positive integer b,

$$gcd\ (a, b) = gcd\ (b, (a \bmod b)).$$

Proof: We will show

1. $\dfrac{\gcd(a,b)}{\gcd(b, a \pmod b)}$

2. $\dfrac{\gcd\{b, a \pmod b\}}{\gcd(a,b)} \Rightarrow gcd\ (a, b) = \pm\, gcd\ (b), a \pmod b)$ but since both are non-negative, they are equal.

3. Let $d = gcd\ (a, b)$, then $\dfrac{d}{a}$ and $\dfrac{d}{b}$, now $a \pmod b = a - \left\lfloor \dfrac{a}{b} \right\rfloor b$. So it is integer combination of a and b, and so $\dfrac{d}{a} \pmod b$ and $\dfrac{d}{\gcd\{b, a \pmod b\}}$.

4. Let $d = gcd\ (b, a \pmod b))$, then $\dfrac{d}{b}$ and $\dfrac{d}{a} \pmod{(b)}$.

   Since,

   $a = a \pmod b + \left\lfloor \dfrac{a}{b} \right\rfloor b$ is an integer combination of b and $a \pmod b$),

   $\dfrac{d}{a}$ so $\dfrac{d}{\gcd(a,b)}$.

Euclid's Algorithm

This algorithm was described by Euclidin his "Elements" written about 300 B.C.

It is based on the previous theorem.

```
EUCLID (a, b)
 1. if b = 0
 2. then return a
 3. else return EUCLID (b, a mod b)
```

Example, consider the computation of gcd (30, 21)

EUCLID (30, 21) = EUCLID (21, 9)
= EUCLID (9, 3)
= EUCLID (3, 0)
= 3.

This algorithm computes $gcd(a, b)$ by transforming the gcd into equivalent gcd with progressively smaller arguments.

NOTES

The Running Time of EUCLID's Algorithm

What is the worst case for EUCLID's algorithm?

We can assume $a > b \geq 0$, since it not, the first pass of EUCLID fixes this. Also, if $b = a > 0$, then it returns b after one call. So given this what are the worst a, b pairs to put into this. Successive Fibonacci numbers !.

Lemma: If $a > b \geq 0$ and EUCLID (a, b) performs $k \geq 1$ recursive calls, then

$$a \geq F_{k+2} \text{ and}$$
$$b \geq F_{k+1}$$

Proof: We prove by induction on k.

$$k = 1 : \text{if } b = 0, \text{ then } k = 0, \text{ so } b \geq 1 = F_2.$$

Now $a > b$, so $a \geq b + 1 \geq 2 = F_3$.

Assume true for $k - 1$. Assume that a, b are such that EUCLID (a, b) takes k recursive calls.

The first of there is EUCLID $(b, a \pmod b)$. By assumption, this terminates in $k - 1$ calls so that

$$b \geq F_{k-1+2} = F_{k+1}$$
$$a \pmod b \geq F_{k+1-1} = F_k.$$

We want to prove that $a \geq F_{k+2}$ as well.

$$b + a \pmod b = b + a - \left\lfloor \frac{a}{b} \right\rfloor b$$

$$= a + b\left(1 - \frac{a}{b}\right)$$

Since $a > b > 0$, we know $\left\lfloor \frac{a}{b} \right\rfloor \geq 1$

$$= a \quad \text{if } \left\lfloor \frac{a}{b} \right\rfloor = 1$$

$$= a - b \quad \text{if } \left\lfloor \frac{a}{b} \right\rfloor = 2$$

$$\leq a \text{ in all cases.}$$

So
$$a \geq b + a \pmod b$$
$$\geq F_{k+1} + F_k = F_{k+2}.$$

Theorem (Lame's Theorem)

For any integer $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, then the call EUCLID (a, b) makes fewer than k recursive calls.

Proof: The upper bound on Lame's theorem is the best possible as seen with:

$$gcd(F_{k+2}, F_{k+1}) = gcd(F_{k+1}, F_{k+2} \pmod{F_{k+1}})$$

But
$$F_{k+2} = F_{k+1} + F_k \text{ so}$$
$$F_{k+2} \pmod{F_{k+1}} = F_k$$
$$= gcd(F_{k+1}, F_k)$$
$$\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots$$
$$= gcd(F_1, F_0), gcd(1, 0) = 1.$$

This takes exactly $k + 1$ recursive calls. This is what the lemma states as

$$b = F_{k+1} < F_{k+2}$$

Recall F_k is approximately $\frac{\phi k}{\sqrt{5}}$, where ϕ is the golden ratio $\frac{(1+\sqrt{5})}{2}$. Since there are k calls it, $b < F_{k+1}$

$$b \approx \frac{\phi k}{\sqrt{5}} \text{ or } k = 0\ (lg\ b).$$

The Extended Euclidean Algorithm

Since $gcd\ (a, b) = \min \{ax + by > 0 : x, y \in Z\}$

One can try to find x, y so that

$$gcd\ (a, b) = ax + by.$$

The extended Euclidean algorithm does then *by* "carrying along" valid x and y values until the gcd is computed.

EXTENDED-EUCLID (a, b)

1. if $b = 0$
2. then $(a, 1, 0)$
3. $(d', x', y') \rightarrow$ EXTENDED-EUCLID $(b, a \text{ (mod } b))$
4. $(d, x, y) \rightarrow \left(d', y', x' - \left\lfloor \frac{a}{b} \right\rfloor y'\right)$
5. return (d, x, y)

The following example shows how this works on gcd (99, 78).

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	– 11	14
78	21	3	3	3	– 11
21	15	1	3	– 2	3
15	6	2	3	1	– 2
6	3	2	3	0	1
3	0	–	3	1	0

Each line shows one level of the recursion: the value of the inputs a and b, the computed values $\left\lfloor \frac{a}{b} \right\rfloor$, and the value d, x and y returned. The triple (d, x, y) returned becomes the triple (d', x', y') used in the computation at the next higher level of recursion. The call EXTENDED-EUCLID (99, 78) returns (3, – 11, 14), so *gcd* (99, 78) = 3 and *gcd* (99, 78) = 3 = 99. (– 11) + 78 = 14.

Note:

$$d = gcd\ (a, b) = ax + by$$

$$d' = gcd\ (b, a \text{ (mod } b)) = bx' + a \text{ (mod } b)\, y'$$

they are equal, so

$$d = bx' + a \text{ (mod } b)\, y'$$

$$= bx' + \left(a - \left\lfloor \frac{a}{b} \right\rfloor b\right) y'$$

$$= ay' + b\left(x' - \left\lfloor \frac{a}{b} \right\rfloor y'\right)$$

Thus if $$x = y' \text{ and } y = x' - \left\lfloor \frac{a}{b} \right\rfloor y'$$

We get consistency, and this proves why it works.

NOTES

5.21 MODULAR ARITHMETIC

Working with integers modulo n, or with equivalence classes of integers.

Finite Group

A Group (S, ⊕) is a set S, with a binary operation, ⊕, with

1. Closure: $\forall\ a, b \in S, a \oplus b \in S$
2. Identity: $\exists\ e \in S$ such that
$$e \oplus a = a \oplus e = a\ \forall\ a \in S$$
3. Associativity: $\forall\ a, b, c \in S, (a \oplus b) \oplus c = a \oplus (b \oplus c)$
4. Inverse: $\forall\ a \in S, \exists\,!\ b \in S$ such that $a \oplus b = b \oplus a = e$.

Example. *Consider the familiar group (Z, +) of integers Z under the operation of addition: 0 is the identity, and the inverse of a is – a. If a group (S, ⊕) satisfies the commutative law $a \oplus b = b \oplus a$ for all, a, $b \in S$, then it is an abelian group. If a group (S, ⊕) satisfies $|S| < \infty$, then it is a finite group.*

Consider defining a group on $S = Z_n$ (the integers module n]. We need to define ⊕. Consider

$$[a]_n +_n [b]_n = [a + b]_n$$
$$a + b \text{ is } a + b \pmod n$$
$$[a]_n \cdot_n [b_n] = [ab]_n$$
$$a \cdot b \text{ is } a \cdot b \pmod n.$$

Example. *Suppose there are two finite groups. Equivalence classes are denoted by their representative elements.*

Group $(Z_6, +_6)$

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

Group $\left(Z^*_{15}, \cdot_{15}\right)$

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

Theorem. The system $(Z_n, +_n)$ is a finite abelian group, called the additive group modulo n.

Proof: Associativity and commutativity follows from +, $e = 0$, the additive inverse of a is $-a$, or $n - a \pmod n = -a$.

The multiplication group module n is denoted as $\left(Z^*_n, \cdot_n\right)$

$$Z^*_n = \{(a)_n \in Z_n : gcd\ (a, n) = 1\}$$

NOTES

Since $15 = 5 \cdot 3$ and $Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ as 3, 5, 6, 9, 10 and 12 have 3 or 5 as factors $\cdot_n$ is multiplication modulo n.

$|Z_n^*|$ is not necessarily n, since not all integers are relatively prime to n, so we define

$|Z_n^*| = \phi(n)$ is Euler's phi, or totient function, where

$$\phi(n) = n \prod_{\frac{p}{n}} \left(1 - \frac{1}{P}\right)$$

This is a product over all primes dividing n.

$\phi(45) = ?$ 3, 5 are the primes dividing 45, so

$$\phi(45) = 45 \left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right)$$

$$= 45 \left(\frac{2}{3}\right)\left(\frac{4}{5}\right)$$

$$= 24.$$

If P is prime, then $\phi(P) = P\left(1 - \frac{1}{P}\right) = P - 1$, and if n is composite, $\phi(n) < n - 1$.

5.22 SOLVING MODULAR LINEAR EQUATIONS

Let's solve $ax \equiv b \pmod{n}$ for x, with a, b and n given integers. Consider $< a >$ in Z_n, if $b \in < a > = \{ax \pmod{n} : x > 0\}$, then the equation has a solution.

Theorem: For any positive integers a and n, if $d = gcd(a, n)$, then $<a> = <d>$ $= \left\{0, d, 2d, \cdots, \left(\frac{n}{d} - 1\right)d\right\}$ and thus $| < a > | = \frac{n}{d}$.

Proof: Since $d = gcd(a, n) = ax' + ny'$, and $ax' = d \pmod{n}$ so $d \in < a >$, and $kd \in < a >$ for $\forall\, k$, because a multiple of a is just a multiple of a, so $< d > \leq < a >$.

If $m \in < a >$, then $m = ax \pmod{n}$ form some x, so $m = ax + ny$ for some y, but since $\frac{d}{a}$ and $\frac{d}{n}$ and $\frac{d}{m}$ and so $m \in < d >$ and so $< a > \leq < d >$ so $< a > = < d >$.

Since exactly $\frac{n}{d}$ multiples of d lie in 0, ..., $n - 1$ so $| < a > | = | < d > | = \frac{n}{d}$.

Corollary: $ax \equiv b \pmod{n}$ is soluble for $x \Leftrightarrow gcd\ \frac{(n, a)}{b}$.

Corollary: If $ax \equiv b \pmod{n}$ has either $d = gcd(a, n)$ distinct solutions or none.

Proof: If $ax \equiv b \pmod{n}$ has a solution, then $b \in < a >$, $a_i \pmod{n}$ is periodic with period $| < a > | = \frac{n}{d}$. If $b \in < a >$, then it appears exactly d times in $a_i \pmod{n}$, for

$i = 0, 1, ..., n - 1$. Since $\frac{n}{d}$ long block of values for < a > is repeated d times. The indices of these are the solutions.

NOTES

Theorem. Suppose $d = gcd(a, n) = ax' + by'$ for some $x', y' \in Z$. If $\frac{d}{b}$, then $ax \equiv b \pmod{n}$ has x_0 as a solution, with $x_0 = x'\left(\frac{b}{d}\right) \pmod{n}$.

Proof: $ax' \equiv d \pmod{n}$, so we have

$$ax_0 = ax'\left(\frac{b}{d}\right) \pmod{n}$$

$$= d\left(\left(\frac{b}{d}\right) \pmod{n}\right)$$

$$= b \pmod{n}.$$

Theorem: If x_o is a solution to $a_x \equiv b \pmod{n}$, then this equation has exactly $d = gcd(a, n)$ distinct solutions $\pmod{n}$, $x_i = x_0 + i\left(\frac{n}{d}\right)$, for $i = 0, 1, 2, ..., d - 1$.

Proof: $\frac{n}{d} > 0$ and $0 \le i\left(\frac{n}{d}\right) < n$. For $i = 0, 1, 2, ..., d - 1$, so the solutions are distinct, and are all, clearly solutions and by the above corollary, there are all the solutions.

MODULAR-LINEAR-EQUATION-SOLVE (a, b, n)

1. $(d, x', y') \leftarrow$ EXTEND-EUCLID (a, n)
2. if $\frac{d}{b}$
3. then $x_0 \leftarrow x'\frac{b}{d} \pmod{n}$
4. For $i \leftarrow 0$ to $d - 1$
5. do print $\left(x_0 + i\left(\frac{b}{d}\right)\right) \pmod{n}$
6. else print "No solutions".

Example. *Consider $14x = 30 \pmod{100}$, where $a = 14$, $b = 30$, $n = 100$.*

$(d, x', y') = (2, -7, 1)$ and since $\frac{2}{30}$, there are $Z = d$

Solution. $x_0 = -7\left(\frac{30}{2}\right) = -105 = 95 \pmod{100}$.

So $x_0 = 95$

$$x_1 = 95 + \frac{100}{2} \pmod{100}$$

$$= 145 \pmod{100} = 45$$

The running time is $\theta(\lg n + gcd(a, n))$

$\theta(\lg n)$ is the cost of computing $gcd(a, n)$ and we then have this many solutions to print.

5.23 COMPUTATIONAL GEOMETRY

NOTES

Computational Geometry *is a branch of computer science devoted to the study of* algorithms which can be stated in terms of geometry. Some purely geometrical problems arise out of the study of computational geometric algorithms, and such problems are also considered to be part of computational geometry. The main impetus for the development of computational geometry as a discipline was progress in computer graphics, computer aided design and manufacturing (CAD/CAM), but many problems in computational geometry are classical in nature. Other important applications of computational geometry include robotics, Geographic information system (GIS) and Integrated circuit design.

The primary goal of research of combinational computational geometry is to develop efficient algorithms and data structures for solving problems stated in terms of basic geometrical objects: points, line segments, polygons, polyhedra, etc. Some of these problems seem so simple that they were not regarded as problems at all until the advent of computers. Consider, for example, the closest pair problem.

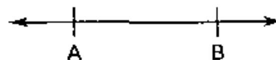
→ Given n points in the plane, find the two with the smallest distance from each other.

One could compute the distances between all the pairs of points of which there are $\frac{n(n-1)}{2}$, then pick the pair with the smallest distances. This brute-force algorithm takes $O(n^2)$ time; *i.e.*, its execution time is proportional to the square of the number of points. A classic result in computational geometry was the formulation of an algorithm that takes $O(n \lg n)$. Randomized algorithm that takes $O(n)$ expected time, as well as a deterministic algorithm that takes $O(n \lg \lg n)$ time, have also been discovered.

Computational geometry focuses heavily on computational complexity since the algorithms are meant to be used in very large data sets containing tens and hundreds of million points. For large data sets, the difference between $O(n^2)$ and $O(n \lg n)$ can be the difference between days and second of computations.

5.24 LINE SEGMENT PROPERTIES

In computational geometry, a line segment is a part of a line that is bounded by two distinct end points, and contains every point on the line between its end points. Example of the line segments include the sides of a triangle or square move generally, when the end points are both vertices of a polygon, the line segment is either an edge (of that polygon) if they are adjacent vertices, or otherwise a diagonal. When the end points both lie on a curve such as a circle, a line segment is called a chord (of that curve).



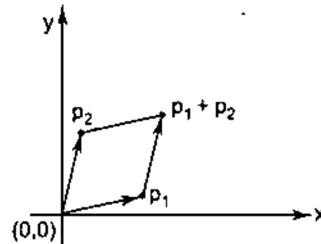
$$\overline{AB} \cap \overline{BA} = \overline{AB}$$

There are the following properties:

1. A line segment is a connected, non-empty set.
2. If V is a topological vector space, then a closed line segment is a closed set in V, however, an open line segment is an open set in V if and only if V is one dimensional.
3. More generally than above, the concept of a line segment can be in an ordered geometry.

Cross Products

Consider, there are two vectors P_1 and P_2. The **cross product** $P_1 \times P_2$ can be interpreted as the signed area of the parallelogram formed by the points (0, 0), P_1, P_2 and $P_1 + P_2 = (x_1 + x_2, y_1 + y_2)$. Actually, the cross product is a three-dimensional concept. It is a vector that is perpendicular to both P_1 and P_2 according to the "right hand rule" and whose magnitude is $|x_1 y_2 - x_2 y_1|$. The following figure shows the cross product of vectors P_1 and P_2 is the signed area of the parallelogram.



By definition, the cross product as the determinant of a matrix:

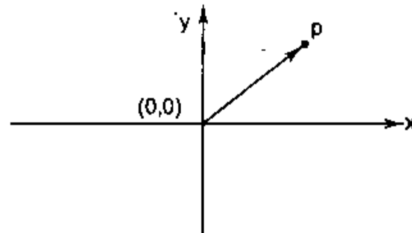
$$P_1 \times P_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$

$$= x_1 y_2 - x_2 y_1$$

$$= -P_2 \times P_1.$$

If $P_1 \times P_2$ is positive, then P_1 is clockwise from P_2 with respect to the origin (0, 0); if the cross product is negative, then P_1 is counter clockwise from P_2.

The following figure shows the clockwise and counter clockwise regions relative to a vector P. A boundary condition occurs if the cross product is 0; in this case the vectors are collinear, pointing in either the same or opposite directions.



To determine whether a directed segment $\overrightarrow{P_0 P_1}$ is clockwise from a directed segment $\overrightarrow{P_0 P_2}$ with respect to their common endpoint P_0, simply we translate to use P_0 as the origin. That is, $P_1 - P_0$ denote the vector $P_1' = (x_1', y_1')$, where $x_1' = x_1 - x_0$ and $y_1' = y_1 - y_0$ and we define $P_2 - P_0$ similarly. We compute the cross product

$$(P_1 - P_0) \times (P_2 - P_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

If the cross product is positive, then $\overrightarrow{P_0 P_1}$ is clockwise from $\overrightarrow{P_0 P_2}$; if negative; it is counter-clockwise.

Determining whether two line segments intersect

To determine whether two line segments intersect, we check each segment straddles the line containing the other. A segment $\overrightarrow{P_1 P_2}$ straddles a line if point P_1 lies on one side of the line and point P_2 lies on the other side. A boundary case occurs if P_1 or P_2 lies directly on the line. Two line segments intersect if and only if either (or both) of the following conditions holds:

1. Each segment straddles the line containing the other.
2. An endpoint of one segment lies on the other segment.

The following procedures implement this concepts.

SEGMENTS-INTERSECT (P_1, P_2, P_3, P_4)

1. $d_1 \leftarrow$ DIRECTION (P_3, P_4, P_1)
2. $d_2 \leftarrow$ DIRECTION (P_3, P_4, P_2)
3. $d_3 \leftarrow$ DIRECTION (P_1, P_2, P_3)
4. $d_4 \leftarrow$ DIRECTION (P_1, P_2, P_4)
5. If (($d_1 > 0$ and $d_2 < 0$) or ($d_1 < 0$ and $d_2 > 0$)) and (($d_3 > 0$ and $d_4 < 0$) or ($d_3 < 0$ and $d_4 > 0$)).
6. then return TRUE
7. else if $d_1 = 0$ and ON-SEGMENT (P_3, P_4, P_1)
8. then return TRUE
9. else if $d_2 = 0$ and ON-SEGMENT (P_3, P_4, P_2)
10. then return true
11. else if $d_3 = 0$ and ON-SEGMENT (P_1, P_2, P_3)
12. then return TRUE.
13. else if $d_4 = 0$ and ON-SEGMENT (P_1, P_2, P_4)
14. then return TRUE
15. else return FALSE

DIRECTION (P_i, P_j, P_k)

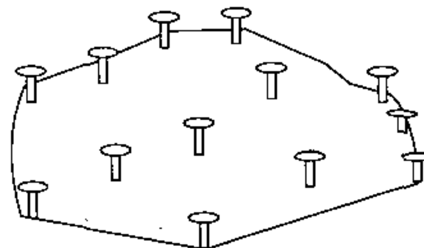
1. return $(P_k - P_i) \times (P_j - P_i)$

ON-SEGMENT (P_i, P_j, P_k)

1. if $\min(x_i, x_j) \le x_k \le \max(x_i, x_j)$ and $\min(y_i, y_j) \le y_k \le \max(y_i, y_j)$
2. then return TRUE
3. else return FALSE

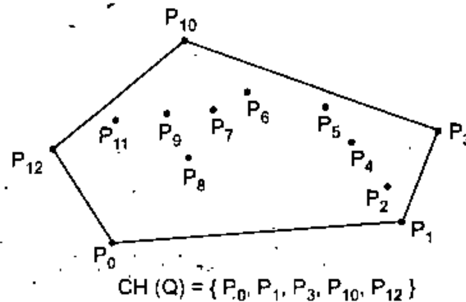
5.25 FINDING THE CONVEX HULL

Intuitively, if we think of each point in given set as being a nail sticking out from a board, the convex hull is the shape formed by a tight rubberband that surrounds all the nails.



Definition of Convex Hull

The convex hull of a set Q of points is the smallest convex polygon P for which each point in set Q is either on the boundary of P or in its iterior. We denote convex hull of Q as CH (Q). The following figure shows a set of point and its convex hull.



Lower Bound on Convex Hull

In order to established the lower bound for common hull, we reduce sorting problem into convex hull problem in the sense that convex hull algorithm can be used to solve sorting problem with little additional work. In other words, if we solve hull problem quickly, we can solve sorting problem quickly.

Suppose, we have an unsorted list of numbers to be sorted $< x_1, x_2, \ldots\ldots, x_n >$, $x_i \geq 0$ for all i. Also, suppose we have an algorithm HULL that constructs the convex hull of n points in T (n) time our task is to use HULL to solve SORTING in time T (n) + O (n) where the O (n) represents additional time to convert the solution of HULL to the solution of SORTING.

Form the set of two-dimensional points $\left(x_i, x_i^2\right)$. These points lie on the parabola $y = x^2$. Run algorithm HULL to construct the hull. Clearly, every point is on the hull. Identify the lowest point a on the hull in 0 (n) time. This corresponds to the smallest x_i. The order in which the points occur on the hull counter clockwise from 'a' is their sorted order. Thus, we can use HULL algorithm to sort.

What we have showed that if we had a fast algorithm for the Hull, we could sort faster than O $(n \lg n)$; but this is known to be impossible. This implies that the lower bound of convex hull is same as that of sorting and that is Ω $(n \lg n)$.

There are the following methods that compute convex hulls in O $(n \lg n)$ time.

1. **In the incremental method,** the points are sorted from left to right yielding a sequence $\{P_1, P_2, \ldots\ldots, P_n\}$. At the ith stage, the convex hull of the $i - 1$ left most points, CH $[(P_1, P_2, \ldots\ldots, P_{i-1})]$, is updated according to the ith point from the left; thus forming CH $(\{P_1, P_2, \ldots\ldots, P_i)\}$. This method can be implemented to take a total of O $(n \lg n)$ time.
2. **In the divide-and-conquer method,** in θ (n) time the set of n points is divided into two subsets, one containing the left most $\left\lceil \frac{n}{2} \right\rceil$ points and one containing the right most $\left\lceil \frac{n}{2} \right\rceil$ points.

   The convex hull of the subsets are computed recursively and then a clever method is used to combine the hulls in $O(n)$ time. The running time is described by the familiar recurrence T (n) = 2T $\left(\frac{n}{2}\right) + O(n)$, and so the divide-and-conquer method runs in $O(n \lg n)$ time.
3. **The Prune-search method** is similar to the worst case linear time median algorithm. It finds the upper portion of the convex hull by repeatedly

throwing out a constant fraction of the remaining points until only the upper chain of the convex hull remains. It then does the same for the lower chain. This method is asymptotically the fastest; if the convex hull contains h vertices, it runs in only O(n lg n) time.

5.26 FINDING THE CLOSEST PAIR OF POINTS

We consider the problem of finding the closest pair of points in a set Q of $n \geq 2$ points. "Closest" refers to the usual euclidean distance: the distance between points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Two points in set Q may be coincident, in which case the distance between them is zero.

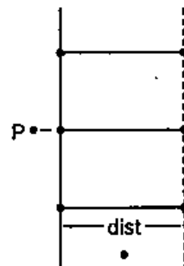
Brute-Force algorithm. A brute-force closest-pair algorithm simply looks at all the $\binom{n}{2} = \theta\ (n^2)$ pairs of points.

1. min dist = infinity
2. For each p in P
3. do for each q in P
4. if $p \neq q$ and dist $(p, q) <$ min dist
5. min dist = dist (p, q)
6. closest pair = (p, q)
7. return closest pair.

Planar Case

The problem can be solved in 0 (n lg n) time using the **recurisve divide and conquer** approach, *e.g.*, as follows:

1. Sort points along the π-coordinate.
2. Split the set of points into two equal-sized subsets by a vertical line $x = x_{mid}$.
3. Solve the problem recursively in the left and right subsets. This will give the left side and right side minimal distances $d_{L_{min}}$ and $d_{R_{min}}$ respectively.
4. Find the minimal distance $d_{LR_{min}}$ among the pair of points in which one point lies on the left of the dividing vertical and the second point lies to the right.



Divide and conquer
sparse box observation

5. The final answer is the minimum among $d_{L_{min}}$, $d_{R_{min}}$ and $d_{LR_{min}}$.

STUDENT ACTIVITY

1. What do you understand by Randomized Algorithm?

2. How many zero-one bitonic sequences of length n are there?

SUMMARY

NOTES

1. A randomized algorithm is one that makes use of a randomizer (such as a random number generator). Some of the decisions made in the algorithm depend on the output of the randomizer.
2. Robin-krap have proposed a string matching algorithm that performs well in practice and that also generalize to other algorithm for related problems, such as two-dimensional pattern matching, the Robin-krap algorithm uses θ (m) processing time and for worst case running time is $\theta((n - m + 1)m)$.
3. The satisfiability problem is to determine whether a boolean formula is true for some assignment of truth values to the variables.
4. Circuit satisfiability problem is boolean combinational circuit composed of AND, OR and NOT gates, CSP one or more boolean combinational elements interconnected by wires.
5. An approximation algorithm for P is an algorithm that generate approximate solution for P.
6. A vertex cover of an undirected graph G (V, E) is a subset $V' \subseteq C$. Such that if $(u, v) \in E$, then $u \in v'$ or $v \in v'$ (or both), where each vertex "covers" the incident edges, and a vertex is the number of vertices in it.
7. Planar coloring graph problem is that by which to determine the minimum number of colors needed to a planar graph G (V, E) to color and every planar graph is four colorable.
8. A **comparison network** is a combination of wires and comparators.
9. **Comparison network** is a collection of comparators interconnected by wires. We draw a comparison network on n inputs as a collection of n horizontal lines with comparators stretched vertically.
10. A **symmetric matrix A** satisfies the condition $A = A^T$. For example, $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$ is a symmetric matrix.
11. A polynomial in the variable x over field **F** is a representation of a function A (x) as a formal sum

$$A(x) = \sum_{j=0}^{n-1} a_j x_j$$

Where n is the degree-bound $a_0, a_1, \ldots\ldots, a_{n-1}$ are the coefficients of the polynomial, *i.e.*, $a_j = 0$ for $j > k$.

12. The convex hull of a set Q of points is the smallest convex polygon P for which each point in set Q is either on the boundary of P or in its iterior. We denote convex hull of Q as CH (Q).

GLOSSARY

- *String Matching:* String matching problem is related to locate all or some occurrence of given pattern string with in a given text string pattern.

- *Approximation Algorithm:* A feasible solution with the value close to the value of an optimal solution is called an approximation solution and algorithm is known as approximation algorithm.
- *Scheduling Task:* Scheduling task works on scheduling rule, which generate a finished time, that is close to the optimal schedule.
- *Sorting Network:* A sorting Network is a comparison network for which the output sequence is monotonically increasing (*i.e.*, $b_1 \le b_2 \le ...... \le b_n$) for every input sequence.
- *Relatively Prime:* Two integers a, b are said to be relatively prime if their only common divisor is 1, that is, if gcd $(a, b) = 1$.
- *Computational Geometry:* It is a branch of computer science devoted to the study of algorithms which can be stated in terms of geometry.

REVIEW QUESTIONS

1. Describe the Randomized Algorithm.
2. Write short notes on:

   (*a*) Las vegas Algorithm

   (*b*) Monte carlo Algorithm.
3. Prove that applying a monotonically increasing function to a sorted sequence produces a sorted sequence.
4. State and prove an analog of the zero are principle for a decision-tree model.
5. How many zero-one bitonic sequences of length n are there?
6. Show that BITONIC-SORTER (n), where n is an exact power of 2 contain $\theta(n \lg n)$ components.
7. How many comparators are there in SORTER [n]?
8. Show that the depth of SORTER [n] is exactly $\dfrac{\lg n\,(\lg n + 1)}{2}$.
9. Prove that matrix inverse are unique, that is, if B and C are inverses of A, then B = C.
10. How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\theta(n \lg 7)$.
11. Use Strassen's algorithm to compute the matrix product $\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix}\begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}$.

    Show your work.
12. Solve the equation: $\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 14 \\ -7 \end{bmatrix}$

    by using forward substitution.
13. Write pseudocode to compute DFT_n^{-1} is $\theta(n \lg n)$ time.

NOTES

14. Compute the DFT of the vector (0, 1, 2, 3).
15. What is difference between DFT and FFT?
16. Describe the generalization of FFT procedure to the care in which n is a power of 3. Give a recurrence for the running time and solve the recurrence.
17. Prove that if $\frac{a}{b}$ and $\frac{b}{c}$, then $\frac{a}{c}$.
18. Prove that if P is prime and $0 < k < P$, then P $\binom{p}{k}$. Conclude that for all integers a, b and primes p.

    $(a + b)^p \equiv ap + b^p \pmod p$
19. Prove that if p is prime and $0 < k < p$, then $gcd\ (k, p) = 1$.
20. Prove that for all integers 0, k and n.

    $gcd\ (a, n) = gcd\ (a + kn, n)$.
21. What does EXTENDED-EUCLID (F_{k+1}, F_k) return? Prove your answer correct.
22. Compute the values (d, x, y) the call EXTENDED-EUCLID (899, 493) return.
23. Show that if p is prime and e is positive integer, then $\phi\ (p^e) = p^{e-1}\ (p - 1)$
24. Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are collinear.
25. Argue that ANY-SEGMENTS-INTERSECT works correctly, even if three or more segments intersect at the same point.
26. Prove that in the procedure GRAHAM-SCAN, points P_1 and P_m must be vertices of CH (Q).
27. Show how to implement the incremental method for computing the convex hull of n points so that it runs in $O(n \lg n)$ time.
28. Suggest a change to the closest-pair algorithm that avoids presenting the Y array but leaves the running time as $O(n \lg n)$.

FURTHER READINGS

- Sachin Dev Goyal, *'Design and Analysis of Algorithm'*, University Science Press.
- Hari Mohan Pandey, *'Design Analysis and Algorithm'*, University Science Press.
- Gyanendra Kumar Dwivedi, *'Analysis and Design of Algorithm'*, University Science Press.