SIMATS

School of Engineering

# Design and Analysis of Algorithms

Computer Science and Engineering

Saveetha Institute of Medical And Technical Sciences.Chennai.

INDEX - CSA 06 / DESIGN AND ANALYSIS OF ALGORITHMS

# DESIGN AND ANAYSIS OF ALGORITHMS

## INTRODUCTION

01> Algorithm Basics
02> Fundamendals of the Analysis of Algorithm Efficieny
03> Asymptotic Notations
04> Mathematical Analysis of Recursive and Non-Recursive Algorithms

## CLASSIFICATION OF ALGORITHMS BY DESIGN

### Divide and conquer Technique.

- Recursive Equation
- Binary Search
- Finding Maximum and Minimum Values
- Merge Sort
- complexity Analysis
- Strassen's Matrix Multiplication

### Greedy Technique

- Container loading Problem
- Knapsack Problem
- Minimum Cost Spanning tree.

### Dynamic Programming

- Optimal Binary Search tree
- Knapsack and Memory Functions
- Travelling Salesman Problem
- Warshall's and Floyid's Algorithms
- Computing Binomial Co-efficients

### Backtracking

- N' Queen Problem
- Hamiltonian Cirluit Problem
- Sum of subset Problem.
- Graph Colouring Problem.

### Branch and Bound

- Assignment Problem
- Knapsack Problem
- Travelling Sales man Problem.

### class And Approximation Algorithms

- NP-Complete and NP-Hard Problem
- P and NP Problem
- Travelling Salesman Problem
- Knapsack Problem
- Minimum Spanning tree

## ALGORITHM:

→ Sequence of unambiguous instructions for solving a problem.

→ Finite set of instructions.

Problem to be solved

↓

Algorithm created for performing particular task

↓

Input → Computes

Output → correct o/p ; error if any

ALGORITHM NAME ( $P_1, P_2, P_3 \ldots P_n$ )

- name of algorithm
- write Parameter (if any)

### CHARACTERISTICS:

Input : Output

Definiteness : Instruction is clear.

Finiteness : Proper sequence.

Efficiency : runs in short time with less memory

Ex:→ Sum of 'n' numbers

```
Algorithm sum (1, n)
// Input : 1 to n numbers
// Output : Summation of numbers
result ← 0
for i ← 1 to n do i ← i+1
    result ← result + 1
return result
```

### STEPS IN ALGORITHM SOLVING

Understand the problem

Decision making
- → Capabilities of computational device
- → Select | extract method
- → Data structures
- → Algorithmic strategies.

Specification of algorithm

Design of algorithm.

Verification

analysis

Coding

### DESIGN STEPS

Specification: Algorithm →
- Using natural language
- Pseudocode
- Flow chart.

ANALYSIS OF ALGORITHM: Specify

| | |
|---|---|
| → Time Efficiency | → Simplicity. |
| → Space Efficiency | → Generality of algorithm. |
| → Range of input | |

### ALGORITHMIC VERIFICATION:

CHECKING CORRECTNESS → Gives correct output in finite amount of time [for a valid input]

by use → mathematical induction

### TIME COMPLEXITY ESTIMATION:

SINGLE LOOP: Ex: Maximum Value.

ALGORITHM : Input : array A[0 ... (n-1)]

Output : Return single maximum value.

```
Max_value ← A[0]
for i ← 1 to (n-1) do
begin
    if (A[i] > max_value) then
        Max_value ← A[i]
end
```

### MATHEMATICAL ANALYSIS

n → no. of elements in array.

c(n) → no. of times comparison & executed hence $\boxed{i = 1 \text{ to } (n-1)}$ times

Sum is $c(n) = \sum_{i=1}^{(n-1)} (1)$

$$\boxed{c(n) = (n-1) \in O(n)}$$

MULTIPLE LOOP:

∴ Example: Elements in a set distinct or not.

Algorithm : Unique Element [A(0 ... (n-1)]

Input : A[0 ... (n-1)]

Output : Return [Elements are not distinct] False

Return [Elements are distinct] True.

ALGORITHM: Unique Element

```
for i ← 0 to n-2 do
begin
    for j ← i+1 to (n-1) do
    begin
        if (A[i] = = A[j]) then
            return false
    end.
end
Return True
```

### MATHEMATICAL ANALYSIS:

$C_{worst}(n)$ = outer loop * inner loop.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (1)$$

$$\left[ \sum_{j=i+1}^{n-1} (1) = (n-1) - (i+1) + 1 \quad \boxed{= n-1-i} \right]$$ Sub in 2nd loop [outer loop]

$$\rightarrow \sum_{i=0}^{n-2} (n-1-i)$$

$$\rightarrow \sum_{i=0}^{n-2} (n-1) + \sum_{i=0}^{n-2} i$$

$$\rightarrow \sum_{i=0}^{n-2} (n-1) - \left[\frac{(n-2)(n-1)}{2}\right]$$

$$\Rightarrow (n-1) \sum_{i=0}^{n-2} (1) = (n-1)[n-2-0+1]$$

$$= (n-1)(n-1) \quad \text{sub in } (n-1)$$

$$(n-1)(n-1) - \left[\frac{(n-2)(n-1)}{2}\right]$$

$$\Rightarrow \frac{(n^2-n)}{2}$$

$$\Rightarrow \frac{n^2}{2} \Rightarrow \boxed{\frac{1}{2} n^2 \in O(n^2)}$$

## Asymptotic Notations:

Asymptotic notations is a short way to represent the time complexity.

Efficiency can be measured by computing time complexity of each algorithm.

Asymptotic notations can give time complexity as fastest possible, shortest possible or average time.

Various notations such as $\Omega, \theta, O$ used are called asymptation.

## Big oh Notation:

The Big oh notation is denoted by "O". It is a method of representing the upper bound of algorithmic running time.

→ Longest amount of time taken by the Algorithm.

**Definition:**

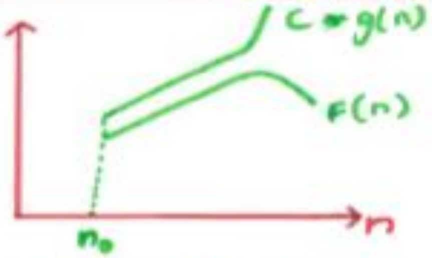
Let $F(n)$ and $g(n)$ be two non-negative functions.

Let ~~no~~ $n_0$ and constant $C$ are two integers such that $n_0$ denotes som value of i/p & $n > n_0$ similarly $C$ is constant $C > 0$

$$F(n) \le C * g(n)$$

Then $F(n)$ is big oh of $g(n)$

$$F(n) \in O(g(n))$$




$$F(n) \in O(g(n))$$

Consider $F(n) = 2n+2$ and $g(n) = n^2$ we have to find $C$, $F(n) \le C * g(n)$

$n = 1$ then
$F(n) = 2n+2$
$= 2(1)+2$
$= 4$
$g(n) = n^2$
$= 1$
$F(n) > g(n)$

if $n = 2$
$F(n) = 2n+2$
$= 2(2)+2$
$= 6$
$g(n) = n^2$
$= 4$
$F(n) > g(n)$

if $n = 3$
$F(n) = 2n+2$
$= 2(3)+2$
$= 8$
$g(n) = n^2$
$= 9$
$F(n) < g(n)$ ✓

Upper bound of existing time is obtained by big oh notation.

## Omega Notation:

→ Denoted by "$\Omega$"

→ Represent the lower bound of algorithm's running time

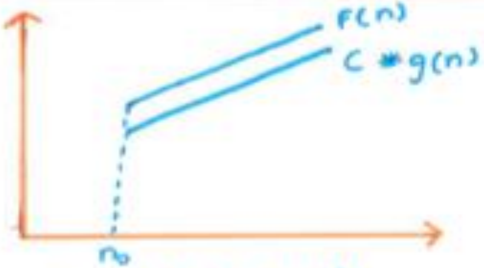
→ Shortest amount of time taken by Algorithm.

**Definition:**

A function $F(n)$ is said to be $\Omega(g(n))$ if $F(n)$ is bounded below some positive contant multiple of $g(n)$ such that

$$F(n) \ge C * g(n) \;\; \forall \; n \ge n_0$$

$$F(n) \in \Omega(g(n))$$




$$F(n) \in \Omega(g(n))$$

Consider $F(n) = 2n^2+5$ and $g(n) = 7n$

Then if $n = 0$
$F(n) = 2(0)^2+5$
$= 5$
$g(n) = 7(0)$
$= 0$
$F(n) > g(n)$

if $n = 1$
$= 7$
$= 7$
$F(n) = g(n)$

if $n = 3$
$= 23$
$= 21$
$F(n) > g(n)$

for $n > 3$ get $F(n) > C * g(n)$

$2n^2+5 \in \Omega(n)$

any $n^3 \in \Omega(n^2)$

## Theta Notation:

→ denoted by "$\theta$".

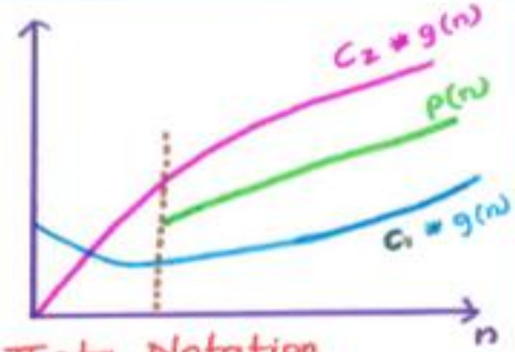
→ Running time between upper bound and lower bound.

**Definition:**

$F(n)$ and $g(n)$ be two non-negative functions. Two Positive constants $C_1$ & $C_2$

$$C_1 \le g(n) \le C_2\, g(n)$$

$$F(n) \in \theta(g(n))$$




**Theta Notation**

$$F(n) \in \theta(g(n))$$

For Example:

If $F(n) = 2n+8$ & $g(n) = 7n$ where $n \ge 2$

Similarly $F(n) = 2n+8$
$g(n) = 7n$

$5n < 2n+8 < 7n$ for $n \ge 2$

Here $C_1 = 5$ and $C_2 = 7$ with $n_0 = 2$

Theta Notation is more precise with both big oh and Omega notation.

## Properties:

1. If $F_1(n)$ is order of $g_1(n)$ & $F_2(n)$ is order of $g_2(n)$, then $F_1(n) + F_2(n) \in O(\max(g_1(n), g_2(n)))$.
2. Polynomials of degree $m \in \theta(n^m)$. That means max.degree is considered from the polynomial

# FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

Important Problem Types:
- Sorting
- Searching
- String Processing
- Graph problems
- Combinational problems
- Geometric Problems
- Numerical Problems

**Sorting:** Rearrange the items of a given list in ascending order.

**Searching:** Deals with finding a given value called a search key in a given set.

**String Processing:** String matching problem searching for a given word in a text.

**Graph Problems:** - $G = (V, E)$ (V → vertices, E → edges)
- Graph traversal
- Shortest path
- Graph colouring
- Topological sort

**Combinational Problems:** To find a combinational object such as permutation, combination, or subset that satisfies certain constraints and has some desired property.

**Geometric Problem:** (To find a combinational object), to deal with geometric objects such as points, lines, polygons
- Closest pair problem
- Convex hull problem

**Numerical Problem:** Mathematical objects of conforming nature, computing definite integrals.

## Fundamentals of the Analysis of Algorithm Efficiency:

Analysis of algorithms is the process of investigation of an algorithm efficiency with respect to the aspects.

Running-time & m/y space

**Analysis Framework:**

Time efficiency or time complexity indicates how fast an algorithm runs.

Space efficiency or space complexity is the amount of m/y units required by the algorithm including the m/y needed for the i/p & o/p

**Measuring an Input's size:**

The efficiency of an algorithm is directly proportional to the input size or range.

Eg: Multiplying two matrices, the efficiency depends on the no. of multiplication of order of matrix.

$b = \text{floor}(\log_2 n + 1)$

Units for measuring Running time:
→ Speed of particular computer
→ Quality of the program
→ Compiler used
→ Difficulty of clocking

The time $T(n)$ for the no. of items ($C(n)$ the basic

Operation ($C_{op}$) is given by

$T(n) \approx C_{op}\, C(n)$

- $T(n)$ → running time
- $C_{op}$ → basic operation
- $C(n)$ → no. of times the operation need to be executed.

**Orders of Growth:**

Logarithmic function grows slow even for high range of inputs whereas the exponential function grows fast for a small increment in the no. of inputs.

| $n$ | $\log_2 n$ | $n$ | $n\log_2 n$ | $n^2$ | $n^3$ |
|---|---|---|---|---|---|
| 10 | 3.3 | $10^1$ | $3.3 \cdot 10^1$ | $10^2$ | $10^3$ |
| $10^2$ | 6.6 | $10^2$ | $6.6 \cdot 10^2$ | $10^4$ | $10^6$ |

**Time - Space Trade of**

A way of solving a problem is less time by using more storage space or by solving a problem in very little space by spending a long time.

Time Complexity:
- no. of steps required by algorithm

Compilation:
- Check Syntax & Semantic

Runtime:
- No. of instructions present in the algorithm

Consider
↳ Limit of executing instruction

**Example:**

**Addition of two numbers:**

```
Sum()
{
1  integer x,y,z ;   declaration
2  Read x,y;
3  z = x+y;
4  Print " The sum of x & y is", z
}
```

3 units for executing the above program.

Complexity
- Worst case $T(n) = O(f(n))$
- Average case
- Best case $T(n) = \Omega(f(n))$

**Space Complexity:**

2 types of m/y
- fixed amount of m/y
- available amount of m/y

Sample Problem:

```
void fun()
{
 int a, b, c, s;
 s = a+b;
 print "Sum: ", s
}
```

Space req:
a = 2
b = 2
c = 2
s = 2
8 units

Space required by the algorithm in 8 units of m/y

# MATHEMATICAL ANALYSIS OF RECURSIVE AND NON-RECURSIVE ALGORITHM



## Recurrence Equation:

Recurrence Equation is an Equation that depends and defines a sequence recursively

$T(n) = T(n-1) + n$ for $n > 0$ ...①

$T(0) = 0$ ...②

Eq.① is called recurrence relation

Eq.② is called initial condition.

Solving Recurrence Equations.

→ Substitution method is a kind of method in which a guess for the solution is made.

* Forward Substitution
* Backward Substitution

## Forward Substitution:

→ Use of an initial condition in the initial term and value for the next term is generated, continued until some formulae is guessed.

$T(n) = T(n-1) + n$ —①

$T(0) = 0$

if $n=1$ $T(1) = T(0) + 1$

$T(1) = 1$ —②

if $n=2$ $T(2) = T(1) + 2$

$T(2) = 1 + 2 = 3$ —③

if $n=3$ $T(3) = 6$

By observing above generation equations. $T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + n/2$

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

## Backward Substitution:

→ Backward values are substituted recursively in order to derive some formulae.

Consider a recurrence relation.

$T(n) = T(n-1) + n$ —①

Initial Condition $T(0) = 0$

$T(n-1) = T(n-1-1) + (n-1)$ —②

Putting Eq② in Eq①, we get

$T(n) = T(n-2) + (n-1) + n$ —③

Let

$T(n-2) = T(n-2-1) + (n-2)$ —④

putting Eq.④ in Eq.③ we get

$T(n) = T(n-3) + (n-2) + (n-1) + n$

⋮

$= T(n-k) + n(n-k+1) + (n-k+2) + \dots + n$

if $k = n$ then

$T(n) = T(0) + 1 + 2 + \dots n$

$T(n) = 0 + 1 + 2 + \dots + n$ $\because T(0) = 0$

$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + n/2$

$T(n) \in O(n^2)$

## Mathematical Analysis of Non-Recursive Algorithm.

General Plan for Analysing Efficiency of Non-Recursive Algorithms.

1. Decide the input size based on parameter n
2. Identify algorithms basic operation:
3. How many times the basic operation is executed. find execution of basic operation depends upon the input size n. Determine worst case, avg & best case.

## Finding the element with maximum value in a given array:

```
Algorithm Max.Element(A[0...n-1])
// Problem description: finding maximum
// Input: array A[0.....n-1]
// Output: Returns the largest element
   from array.
Max.value ← A[0]
for i ← 1 to n-1 do
{
  if (A[i] > Max-value) then
     max-value ← A[i]
}
return max-value.
```

## Mathematical Analysis:

Step.1: Input size n

Step.2: Basic operation is comparison in loop

Step.3: Executing in loop no need to find WC, AC, BC

Step.4: $C(n)$ be the no. of times the comparison is executed.

for i=1 to n-1

$C(n)$ = one comparison made for each value of i.

Step.5: Simplify the sum.

$C(n) = \sum_{i=1}^{n-1} 1$ $\left[\text{using rule } \sum_{i=1}^{n} 1 = n \in \Theta(n)\right]$

$C(n) = n - 1 \in \Theta(n)$

The efficiency of above alg. $\Theta(n)$

## Mathematical Analyses of Recursive alg.

General plan for analysis efficiency of Recursive alg.

1. Decide input size.
2. Identify basic operations
3. Check how many times executing
4. Setup recursive relation with some initial condition as expressing the basic operation.
5. Solve the recurrence or atleast determine the order of growth, use forward / backward subst method

## Factorial of some no. n:

```
Algorithm Factorial(n)
// problem description: compute n!
// Input: A non-negative Integer n
// Output: return the fact value
   if (n=0)
      return 1
   else
      return Factorial(n-1)*n
```

Analysis is: $M(n) = M(n-1) + 1$

Time complexity: $\Theta(n)$

# Divide And Conquer

## Divide and conquer technology

### Steps

* Divided into smaller sub problem → Divide
* Sub problems – solved independently → conquer
* combines all the solution of subproblems of the whole } → combine

### Recurrence eqution is

$$T(n) = \begin{cases} g(n) \\ T(n_1) + T(n_2) + \cdots T(n_r) + f(n) \end{cases}$$

$T(n) \rightarrow$ Time for divide & conquer

$g(n) \rightarrow$ compute time to solve smale inputs

problem of size 'n'

sub problem of size n/2 | sub problem of size n/2

solution to sub problem m-1 | solution to sub problem m-2

solution to original problem

Obtaining time for size n is:

$$T(n) = \underset{\downarrow}{a}\, \underbrace{T(n/b)}_{\text{Time for size } n/b} + f(n)$$

Time for size 'n' — No of subinstances — Time required for dividing the problem into subproblem

$(a_0 + a_1 + \cdots a_n)$

$(a_0 + \cdots a[n/2]\ (2))$ | $a[n/2] + \cdots a_{n-1})$

solution (1) | solution (2)

solution to $a_0 + a_1 + \cdots a_{n-1}$

### Recurrence equation to obtaing time

$$\boxed{\text{for size 'n' is} \quad T(n) = a^T(a/n) + f(n)}$$

Time for size 'n' — no. of sub instances — time required for divides the problem into sub problem

### The order is at

$$T(n) = a^k \left[ T(1) + \sum_{i=1}^{k} \frac{f(b^i)}{a^i} \right]$$

$$T(n) = n^{\log_b a} \left[ T(1) + \sum_{j=1}^{\log_b n} \frac{f(b^i)}{a^i} \right]$$

order of growth of T(n) depends the values of constants a

### Binary search

It is an efficient searching method and all the elements in the array should be shorted

### Three conditions

i) needs to be tested

if key = A[m] → then the desired element present in a list

if key < A[m] → then search the left sublist

if key & A(m) then search the right sublist

It can be represent as

$A(0) \cdots A(m)$   $A(m+1) + \cdots A(n-1)$

search here if key < A(m) — key — search here if key > A(m)

**Example**

consider a list of element

$a \rightarrow \{10, 20, 30, 40, 30, 60, 70\}$

key element, '60' → {element to be search}

m = (low+high) / 2

m = (0+6) / 2

m = 3

A(3) = 40   40 < 60

right list = (40, 60, 70)

mid-element (60 - find)

★ The comparison is also called a three way comparison because algorithm makes the comparison to determine whether KEY is smaller, equal to or greater than A[m].

## Finding maximum & minimum

The list of elements is divided at the mid in order to obtain two sublists. From both the sublist maximum & minimum elements are closer

Algorithm max-min (i, j, max, min)
(i, j, max, min)
// Problem description finding min, max recursively
// input: i, j variables nsw as index to the array

```
if (i==j) then
{
 max ← A[i]
 min ← A[j]
}
else if (i=j-1) then
{
 if A[i]<A[j]) then
 {
  max ← A[j]
  min ← A[i]
 }
 else
 {
  max ← A[i]
  min ← A[j]
 }
}
else
{
 mid ← (i+j)/2
 max-min-val (i, mid, max, min)
 max-min-val (mid+1, j, max_new, min_new)
 if (max<max_new) then
   max ← max_new // combine
 if (min > min_new) then
   min ← min_new // combine
}
```

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 50 | 40 | -5 | -9 | 45 | 90 | 65 | 25 | 75 |

| | | | | |
|---|---|---|---|---|
| 50 | 40 | -5 | -9 | 45 |

| | | | |
|---|---|---|---|
| 90 | 65 | 25 | 75 |

min = -9 max = 90.

### Analysis

Two recursive calls made in this algorithm, for each half divided sub lists.

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 2$ when $n>2$

$T(n) = 1$ when $n=2$

$T(n) = 0$ single element

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

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

$= 2(2(2T(n/8)+2]+2)+2$

$= 8T(n/8)+10$

if we put $n = 2^k$

$T(n) = 2^{k-1}T(2) + \sum_{i=1}^{k-1} 2^i$

$= 2^{k-1} + 2^k - 2$

$T(n) = 3n/2 - 2.$

Neglecting the order of magnitude Time complexity is $O(n)$

## Merge sort

Merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out.

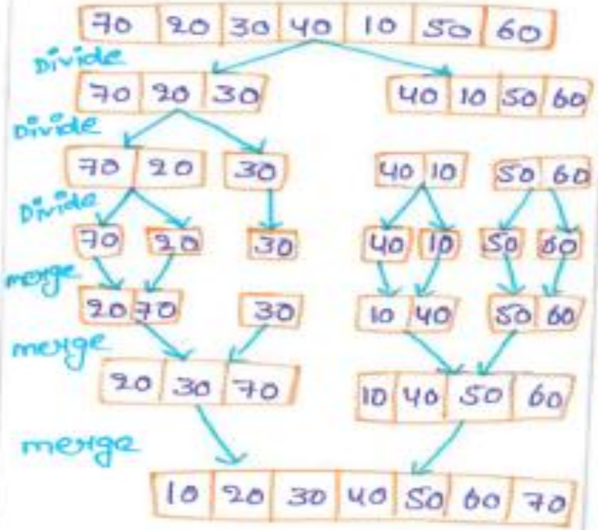
**3 steps**

Divide: Position array into 2 sublist $S_1$, $S_2$ with $n/2$ elements

Conquer: sort sublist $S_1$ & $S_2$

Combine: merge $S_1$ & $S_2$ into a unique sorting group

Example.




### Analysis

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

average and worst case $O(n \log_2 n)$

Algorithm merge sort (A[0....n-1], low, high)

```
{
 if (low<high) then
 {
  mid ← (low+high)/2
  merge sort (A, low, mid)
  merge sort (A, mid+1, high)
  combine (A, low, mid, high)
 }
```

Algorithm combine (A[0...n-1], low, mid, high)

```
{
 k ← low  i ← low  j ← mid+1
 while (i<=mid and j<=high) do
 {
  if (A[i]<=A[j]) then
  {
   Temp[k] ← A[i]
    i ← i+1
    k ← k+1
  }
  else
  {
   temp[k] ← A[j]
    j ← j+1
    k ← k+1
  }
 }
 while (i<=mid) do
 { temp[k] ← A[i]
    i ← i+1
    k ← k+1
 } while (j<=high) do
 { temp[k] ← A[j]
    j ← j+1
    k ← k+1
 }
```

Best case

$O(n \log_2 n)$

# BINARY SEARCH - INDUCTION - Merge Sort

## Binary Search:

Time Complexity Analysis

Basic → Key element is operation competed with all array elements

Efficiency → To count the no. of times the search key is compared with the array elements

Comparision array is divided each time n/2 sublists

$C_{worst} = C_{worst}(\lfloor n/2 \rfloor) + 1$

(time required to compare left sublist or rightsublist)

∀ (n>1) one comparision is made.

Also,

$C_{worst}(1) = 1$

Then the Recurrence eq'n

$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1$ for $n>1$

$C_{worst}(1) = 1$

Assume $n = 2^k$

$C_{worst}(2^k) = C_{worst}(2^k/2) + 1$

$C_{worst}(2^{k-1}) + 1$ —①

Then substitute

$C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1$ —②

Sub ② in ①;

$C_{worst}(2^k) = [C_{worst}(2^{k-2}) + 1] + 1$

$= C_{worst}(2^{k-2}) + 2$

then,

$C_{worst}(2^k) = [C_{worst}(2^{k-k}) + k]$

$\Rightarrow C_{worst}\ 2^0 + k$

$C_{worst}(2^k) = C_{worst} + k$

Use recurrence Equation:

$C_{worst}(1) = 1$

$C_{worst}(2^k) = 1 + k$

$C_{worst}(n) = 1 + \log_2 n$

$C_{worst}(n) \approx \log_2 n \quad [n>1]$

Time Complexity is $\theta(\log_2 n)$

Best case $\theta(1)$ | Avg case $\theta(\log_2 n)$ | Worst case $\theta(\log_2 n)$

Time Complexity Difference:

| Linear Search | Binary Search |
|---|---|
| Best Complexity is $\theta(1)$ where the element is found at first index | Best Complexity is $\theta(1)$ where the element is found at middle |
| more no. of comparision is taken | Less no of comparision is taken |

## Merge Sort

Time Complexity Analysis In Merge Sort, two recursive calls are made.

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

Time taken by left sublist | Right sublist | time taken

where $n>1$ $T(1) = 0$

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

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

$T(1) = 0$

Assume $n = 2^k$

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

$T(n) = 2T\left(\frac{2^k}{2}\right) + c.2^k$

$T(2^k) = 2T(2^{k-1}) + c.2^k$

If we put $k = k-1$ then,

$T(2^k) = 2T(2^{k-1}) + c.2^k$

$= 2[2T(2^{k-2}) + c.2^{k-1}] + c.2^k$

$= 2^2\{T(2^{k-2}) + 2.c.2^{k-1} + c.2^k$

$= 2^2 T(2^{k-2}) + 2.c.\frac{2^k}{2} + c.2^k$

$= 2^2 T(2^{k-2}) + c.2^k + c.2^k$

$T(2^k) = 2^2 T(2^{k-2}) . 2c.2^k$

Similarly; we can write;

$T(2^k) = 2^3 T(2^{k-3}) + 3c.2^k$

$= 2^4 T(2^{k-4}) + 4c.2^k$

....

....

$= 2^k T(2^{k-k}) + k.c.2^k$

$= 2^k T(2^0) + k.c.2^k$

$T(2^k) = 2^k T(1) + k.c.2^k$

As per eq. $T(1) = 0$,

then,

$T(2^k) = 2^k.0 + k.c.2^k$

$T(2^k) = k.c.2^k$

But we assumed $n = 2^k$,

By taking log on both sides

i.e.,

$\log_2 n = k$

$\therefore T(n) = \log_2 n . cn$

$\therefore T(n) = \theta(n.\log_2 n)$

Hence the average and worst case time complexity of merge sort is $\theta(n \log_2 n)$.

Time Complexity of Merge Sort

Best Case $\theta(n \log_2 n)$

Worst case $\theta(n \log_2 n)$

Average Case $\theta(n \log_2 n)$

Divide & conquer approach can reduce the no. of one digit multiplications in multiplying two inputs. The principal insight of the algorithm lies in the discovery that we can find the product c of two 2 by 2 matrices A & B

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ b_{10} & b_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{bmatrix}$$

$m_1 = (a_{00}+a_{11}) * (b_{00}+b_{11})$

$m_2 = (a_{10}+a_{11}) * b_{00}$

$m_3 = a_{00} * (b_{01}-b_{11})$

$m_4 = a_{11} * (b_{10}-b_{00})$

$m_5 = (a_{00}+a_{01}) * b_{11}$

$m_6 = (a_{10}-a_{00}) * (b_{00}+b_{01})$

$m_7 = (a_{01}-a_{11}) * (b_{10}+b_{11})$

Strassen's Algorithm in $O(n \log_2 7)$

The greedy method is a straight forward method. This method. This method is popular for obtaining the optimization solutions.

→ The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained, until a complete solution to the problem is acheived

General method

```
Algorithm Greedy (D,n)
solution <- 0
for i <- 1 to n do
{
  s <- select (D)
  if (feasible (solution,s)) then
    solution <- union(solution,s);
}
return solution.
```

**Container loading**

→ The strip is loaded with containers, At each stage each containers is loaded.

→ The total weight of all the containers must be less than or equal to the capacity

**for example**

n=8 be total no. of containers having weights $(w_1, w_2, w_3 \ldots w_8)$ = [50, 100, 30, 80, 90, 200, 150, 20]. e = 400

Stage 1: select the container with mint weight 20

remaining wt 400-20 = 380

Solution set = [0,0,0,0,0,0,0,1]

Here 1 in the array $x_8$ container loader

Stage 2: next min wt 30

remaining wt 380-30 = 350

set = [0,0,1,0,0,0,0,1]

Stage 3: next min wt 50.

remaining wt 350-50 = 300

set = [1,0,1,0,0,0,0,1]

Stage 4: next 80 remaining 300-80 = 220

set = [1,0,1,1,0,0,0,1]

Stage 5: next 90 remaining 220-90 = 130

set = [1,0,1,1,1,0,0,1]

Stage 6: next 100 remaing 130-100 = 30

set = [1,1,1,1,1,0,0,1]

Stage 7: next wt 150 exceute the value

$[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$

= [1,1,1,1,1,0,0,1]

```
Algorithm container load (int container int w, int num, int sol[])
{
  Heap - sort (container, num);
  for (i <- 1 to num) do
  {
    solution [i] <- 0;
  //select the container with min wt
  for (i <- 1 to num AND container [i]. wt <= w) do
  {
    solution [container [i]. id] = 1
    w = w - container [i]. wt;
  }
}
```

**Analysis**

Algorithm takes o(n logn) time complexity because heap sort takes o(n log n) time and remaing time of Algorithm taks o(n) time where n is the no. of containers.

knapsack problem can be stated that

n objects from $i=1,\dots n$

each object $i$ has some positive weight $w_i$ as some profit value is associated with each object which is denoted as $p_i$ at most weight $w$.

1. choose only those objects should be $\le w$
2. total weight of selected objects should be $\le w$

$\boxed{\text{maximized } \sum_n p_i x_i \text{ subject } \sum_n w_i x_i \le w}$

where the knapsack can carry the fraction $x_i$ of an object $i$ such that $0 \le x_i \le 1$ as $1 \le i \le n$

consider 3 item, weight & profit value of each item & given

| i | $w_i$ | $p_i$ |
|---|---|---|
| 1 | 18 | 30 |
| 2 | 15 | 21 |
| 3 | 10 | 18 |

$w = 20$

sol: feasible solution

| $x_1$ | $x_2$ | $x_3$ |
|---|---|---|
| $1/2$ | $1/3$ | $1/4$ |
| 1 | 2/15 | 0 |
| 0 | 2/3 | 1 |
| 0 | 1 | 1/2 |

let us compute $\sum w_i x_i$

1. $1/2 * 18 + 1/3 * 15 + 1/4 * 10$
   $= 16.5$
2. $1 * 18 + 2/15 * 15 + 0 * 8$
   $= 20.$
3. $0 * 18 + 2/3 * 15 + 10$
   $= 20$
4. $0 * 18 + 1 * 15 + 1/2 * 10$
   $= 20$

let us compute $\sum p_i x_i$

1. $1/2 * 30 + 1/3 * 21 + 1/4 * 18$
   $= 26.5$
2. $1 * 30 + 2/15 * 21 + 0 * 18$
   $= 32.8$
3. $0 * 30 + 2/3 * 21 + 18$
   $= 32$
4. $0 * 30 + 1 * 21 + 1/2 * 18$
   $= 30$

solution 2 gives the maximum profit and hence it terms out to be optimal solution

Algorithm knapsack-greedy(w,n)

{

// P[i] → profits W[i] → wt

// X[i] → solution vector

for i := 1 to n do

{ if (w[i] < w) then

{ x[i] = 1.0;

w = w - w[i]

}

} if (i <= n) then

x [i] := w/w[i];

}

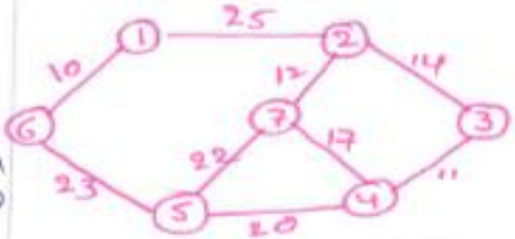
Time complexity O(n)

## minimum cost spanning Tree

spanning tree of a graph G is a subgraph which is basically a tree as it contains all the vertices of G containing no circuit.

minimum spanning tree of a weight. connected graph G is spanning tree with minimum or smallest wt

consider the graph




using kruskal's Algorithm

step1

total wt = 0

step 2

total wt = 10

step 3:

total wt = 21

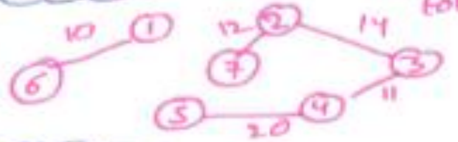
step 4

total wt = 33

step 5

total wt = 47

step 6



total wt = 67

step 7

total wt = 90

efficiency of kruskal's Algorithm is $O(|E| \log |E|)$

$E \to$ total no. of edges in the graph

# DYNAMIC PROGRAMMING - OPTIMAL BINARY SEARCH TREE

## DYNAMIC PROGRAMMING

* For Solving Overlapping of Sub Problems.

### GENERAL METHOD

Applied to optimization Problem.

Ex: Find Min & Max Value in a list

| DIVIDE & CONQUER | DYNAMIC PROGRAMMING |
|---|---|
| 1) Divide - Solve Problem - Combine to get feasible Soln. | Many Seq. Decisions are generated |
| 2) Duplicate Sol. may be obtained | Duplicate sol. totally avoided |
| 3) Top-Down Approach | Bottom up approach |

### Steps of dynamic Programming

* Characterize the Structure of optimal Solution.
* Recursively define → Value of an optimal sol.
* Develop a recurrence relation solution to the subproblem.

### Principles of Optimality

* Optimal sequences of decision or choice
* Subsequence must be obtained

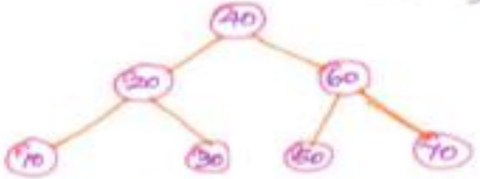
### Application

* Multistage graph
* Finding shortest path
* Optimized binary Search tree
* 0/1 Knapsack Problem
* Travelling Salesman problem.

### Optimal Binary Search tree. (OBST)

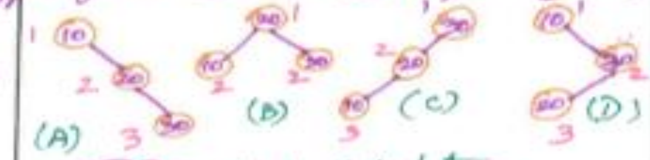
* Using Dynamic Programming

Example:
Key - 10, 20, 30, 40, 50, 60, 70 (Sorted Order)




* It is Balanced Search Tree
* Takes O(log n) time to search the tree.
* As the no. of input elements increases, the no. of operations same. i.e Same time O(log n)
* Inputs → Search keys → Sorted
  → Frequences → Each key

EXAMPLE:
Keys[ ] = {10, 20, 30} freq[ ] = {3, 2, 5}





Cost Calculation

(A) 1×3 + 2×3 + 3×5 = 22
(B) 1×2 + 2×3 + 2×5 = 18
(C) 1×5 + 2×2 + 3×3 = 18
(D) 1×3 + 5×2 + 2×3 = 19
(E) 1×5 + 2×3 + 3×2 = 17

BEST Search tree

Though it is not balanced with respect to freq. the search cost is less.

**No. of Possible ways to Construct BST**

$$C_n = \frac{(2n)!}{(n+1)!\,n!}$$

Where n is the total no. of keys.

**Example to Construct OBST**

| | | | | |
|---|---|---|---|---|
| Index → | 0 | 1 | 2 | 3 |
| Keys → | 10 | 12 | 16 | 21 |
| Freq. → | 4 | 2 | 6 | 3 |

Formula for computing each seq

$$C_{i,j} = W_{i,j} + \min_{i<k\leq j}\{C_{i,k-1} + C_{k,j}\}$$

i=j

Cost[0,0] = 4    Cost[2,2] = 6
Cost[1,1] = 2.    Cost[3,3] = 3

### OBST Calculation:

| i \ j | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 4 | 8 (0) | 20 (1) | 26 (2) |
| 1 | | 2 | 10 (1) | 16 (1) |
| 2 | | | 6 | 12 |
| 3 | | | | 3 |

0 1 2 3 / 10 12 16 21 / 4 2 6 3 → 6 + min{2 - 0, 4 - 1} = 6 + 2 = 8

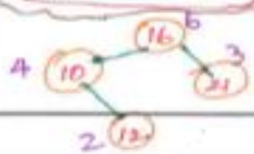
0 1 2 3 / 10 12 16 21 / 4 2 6 3 → 8 + min{2 - 1, 6 - 2} = 8 + 2 = 10

0 1 2 3 / 10 12 16 21 / 4 2 6 3 → 9 + min{3 - 3, 6 - 2} = 9 + 3 = 12

0 1 2 3 / 10 12 16 21 / 4 2 6 3 → 12 + min{10 - C[1,2], 4+6 - 0,2, 8 - C[0,1]} = 12 + 8 = 20

0 1 2 3 / 10 12 16 21 / 4 2 6 3 → = 11 + min{12 - C[2,3], 2+3 - 1,3, 10 - C[1,2]} = 11 + 5 = 16

0 1 2 3 / 10 12 16 21 / 4 2 6 3 → 15 + min{16 - C[1,3], 4+2 - 1, 8+3 - 2, 20 - C[0,2]} = 15 + 11 = 26




Optimal Binary Search tree

6×1 + 4×2 + 3×2 + 2×3 = 26

# KNAPSACK, BINOMIAL CO-EFFICIENT AND MEMORY FUNCTION

## Dynamic Programming

* Mathematical Optimization Method
* A computer programming
* By Richard Bellman in 1950
* Application
  - → Computer networks
  - → Routing
  - → Graph problem
  - → Computer vision
  - → AI & ML applications
* Store result of Subproblems
* Avoid re-compute
* Less Time complexities
* Reduce complexity from exponential to polynomial
  - → Optimal Soln. = { Sub problem Soln. and Current problem.
  - → Best soln. from all possible cases to provid guaranteed optimal Soln.

## 0/1 Knapsack Problem

Want to carry essential item from these in one bag?

Either Pick the item

Don't pick.

Can't split the item

| W | V | i \ w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 4 | 2 | 0 | 1 | [illegible] | 4 | 5 | 5 | 5 | 5 |
| 4 | 5 | 3 | 0 | 1 | [illegible] | 4 | 5 | 6 | 6 | 9 |
| 5 | 7 | 4 | 0 | 1 | [illegible] | 4 | 5 | 7 | 8 | 9 |

← weight (W)

Value (V)

$V[i,w] = \max\{V[i-1,w],\ V[i-1,\ w-w[i]] + Value[i]\}$

$i \rightarrow$ row $w \rightarrow$ column.

$V[4,1] = \max\{[V(3,1),\ V(3,1-5)] + Value[4]\} = 1$

$V[4,2] = \max\{[V(3,2),\ V(3,2-5)] + Value[4]\} = 1$

$V[4,3] = \max\{[V(3,3),\ V(3,3-5)] + Value[4]\} = 4$

$V[4,4] = \max\{[V(3,4),\ V[3,4-5]] + Value[4]\} = 5$

$V[4,5] = \max\{[V(3,5),\ V[3,5-5]] + Value[4]\} = \max(6, 0+7) = 7$

$V[4,6] = \max\{[V(3,6),\ V(3,6-5)] + Value[4]\} = \max(6, 1+7) = 8$

$V[4,7] = \max\{[V(3,7),\ V(3,7-5)] + Value[4]\} = \max(9, 1+7) = 9$

Time Complexity – $O(nW)$

Max value = 9

$\therefore$ Value[2] + Value[3]

$= 4 + 5 = 9$.

$\therefore$ These two values are Pick into the Sack.

## Memory Function

* Sup problem → Solving more than once
* Makes insufficient of Solving a problem
* deal with Overlapping of Subproblem
* Computing the soln. → Subproblem Stack is in a table.
* make use of recursive calls

## Binomial co-efficient

Binomial formula

$(a+b)^n = {}^nC_0 a^n b^0 + {}^nC_1 a^{n-1} b^1 + {}^nC_2 a^{n-2} b^2 + \dots + {}^nC_n a^0 b^n$

→ ${}^nC_0, {}^nC_1, \dots {}^nC_n$ are binomial co-efficient

Binomial co-efficient → $C(n,k)$

$$C(n,k) = \frac{n!}{k!\,(n-k)!}$$

→ No. of ways in disregarding order

→ K objects chosen from 'n' objects

→ More formally the no. of K-element subset of n-element set.

$${}^nC_k = \begin{cases} 1 & \text{if } k=0 \text{ (or) } n=k \\ {}^{n-1}C_{k-1} + {}^{n-1}C_k, & \text{for } n>k>0 \end{cases}$$

(or) Recurrence relation can also be written as:

$${}^nC_k = \begin{cases} 1 & \text{if } k=0 \text{ or } n=k \\ C(n-1,k-1) + C(n-1,k) & \text{for } n>k>0 \end{cases}$$

For Example

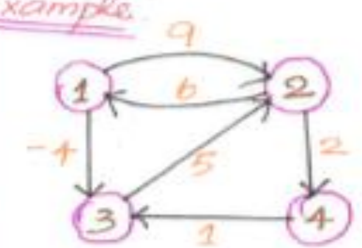
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | | |
| 1 | 1 | 1 | | | | | | |
| 2 | 1 | 2 | 1 | | | | | |
| 3 | 1 | 3 | 3 | 1 | | | | |
| 4 | 1 | 4 | 6 | 4 | 1 | | | |
| 5 | 1 | 5 | 10 | 10 | 5 | 1 | | |
| 6 | 1 | 6 | 15 | 20 | 15 | 6 | 1 | |
| 7 | 1 | 7 | 21 | 35 | 35 | 21 | 7 | 1 |
| K | | | | | | | | |

## FLOYDS - WARSHALL ALGORITHMS

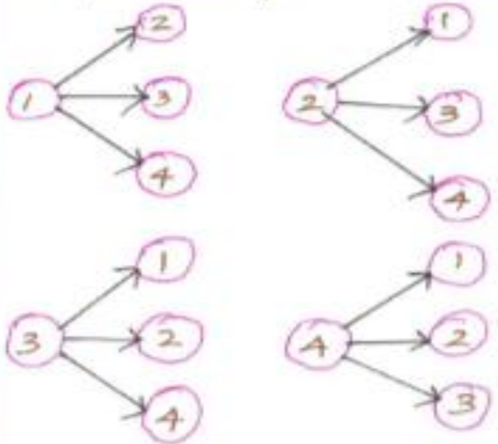
→ All pairs shortest Path

Suitable for Dense Graph and Graph with negative weights.

Example




All possible pairs of nodes.



### ALGORITHM:

STEP 1: Construct $D^0$

If $i == j$, $w_{ij}$ = "—"

else if

$i \to j = \{c\}$

else "$\infty$"

STEP 2: Construct $D^K$

$$w_{ij}^K = \min\{w_{ij}^{K-1}, w_{ik}^{K-1} + w_{kj}^{K-1}\}$$

$D^0$

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 9 | -4 | ∞ |
| 2 | 6 | 0 | ∞ | 2 |
| 3 | ∞ | 5 | 0 | ∞ |
| 4 | ∞ | ∞ | 1 | 0 |

$D^1$

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 9 | -4 | ∞ |
| 2 | 6 | 0 | 2 | 2 |
| 3 | ∞ | 5 | 0 | ∞ |
| 4 | ∞ | ∞ | 1 | ∞ |

$D^1[2,3] = \min\{D^0[2,3], D^0[2,1] + D^0[1,3]\}$

$D^1[2,3] = \min\{\infty, 6+(-4)\} = \boxed{2}$

$D^1[2,4] = \min\{D^0[2,4], D^0[2,1] + D^0[1,4]\}$

$D^1[2,4] = \min\{2, 6+\infty\} = \boxed{2}$

$D^1[3,2] = \min\{D^0[3,2], D^0[3,1] + D^0[1,2]\}$

$D^1[3,2] = \min\{5, \infty+9\} = \boxed{5}$

$D^1[3,4] = \min\{D^0[3,4], D^0[3,1] + D^0[1,4]\}$

$D^1[3,4] = \min\{\infty, \infty+\infty\} = \boxed{\infty}$

$D^1[4,2] = \min\{D^0[4,2], D^0[4,1] + D^0[1,2]\}$

$D^1[4,2] = \min\{\infty, \infty+9\} = \boxed{\infty}$

$D^1[4,3] = \min[D^0[4,3], D^0[4,1] + D^0[1,3]]$

$D^1[4,3] = \min\{1, \infty+4\} = \boxed{1}$

$D^2$

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 9 | -4 | 11 |
| 2 | 6 | 0 | 2 | 2 |
| 3 | 11 | 5 | 0 | 7 |
| 4 | ∞ | ∞ | 1 | 0 |

$D^2[1,3] = \min\{-4, 9+2\} = \boxed{-4}$

$D^2[1,4] = \min\{\infty, 9+2\} = \boxed{11}$

$D^2[3,1] = \min\{\infty, 6+5\} = \boxed{11}$

$D^2[3,4] = \min\{\infty, 5+2\} = \boxed{7}$

$D^2[4,1] = \min\{\infty, \infty+6\} = \boxed{\infty}$

$D^2[4,3] = \min\{1, \infty+2\} = \boxed{1}$

$D^3$

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | -4 | 3 |
| 2 | 6 | 0 | 2 | 2 |
| 3 | 11 | 5 | 0 | 7 |
| 4 | 12 | 6 | 1 | 0 |

$D^3[1,2] = \min\{9, 5-4\} = \boxed{1}$

$D^3[1,4] = \min\{11, 7-4\} = \boxed{3}$

$D^3[2,1] = \min\{6, 11+2\} = \boxed{6}$

$D^3[2,4] = \min\{2, 7+2\} = \boxed{2}$

$D^3[4,1] = \min\{\infty, 11+1\} = \boxed{12}$

$D^3[4,2] = \min\{\infty, 5+1\} = \boxed{6}$

$D^4$

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | -4 | 3 |
| 2 | 6 | 0 | 2 | 2 |
| 3 | 11 | 5 | 0 | 7 |
| 4 | 12 | 6 | 1 | 0 |

$D^4[1,2] = \min\{1, 3+6\} = \boxed{1}$

$D^4[1,3] = \min\{-4, 3+1\} = \boxed{-4}$

$D^4[2,1] = \min\{6, 12+2\} = \boxed{6}$

$D^4[2,3] = \min\{2, 1+2\} = \boxed{2}$

$D^4[3,1] = \min\{11, 12+7\} = \boxed{11}$

$D^4[3,2] = \min\{5, 6+7\} = \boxed{5}$

# Travelling Salesman Problem

## Travelling Salesman Problem.

**Problem:**

Person wants to visit all the 'town' exactly once.

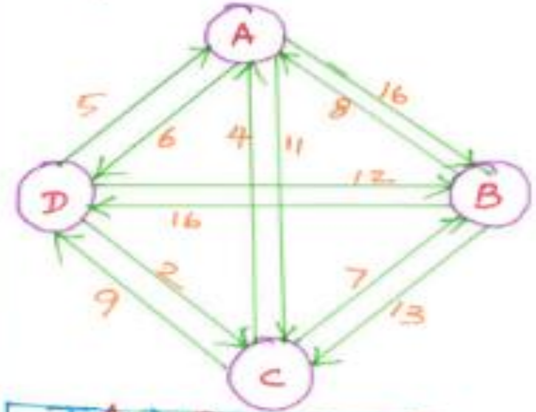
$G(V,E)$. V – Set of Vertices
E – Set of Edges.

Edges with cost $c_{ij} > 0$

$c_{ij} = \infty$ (No of edges between i & j)

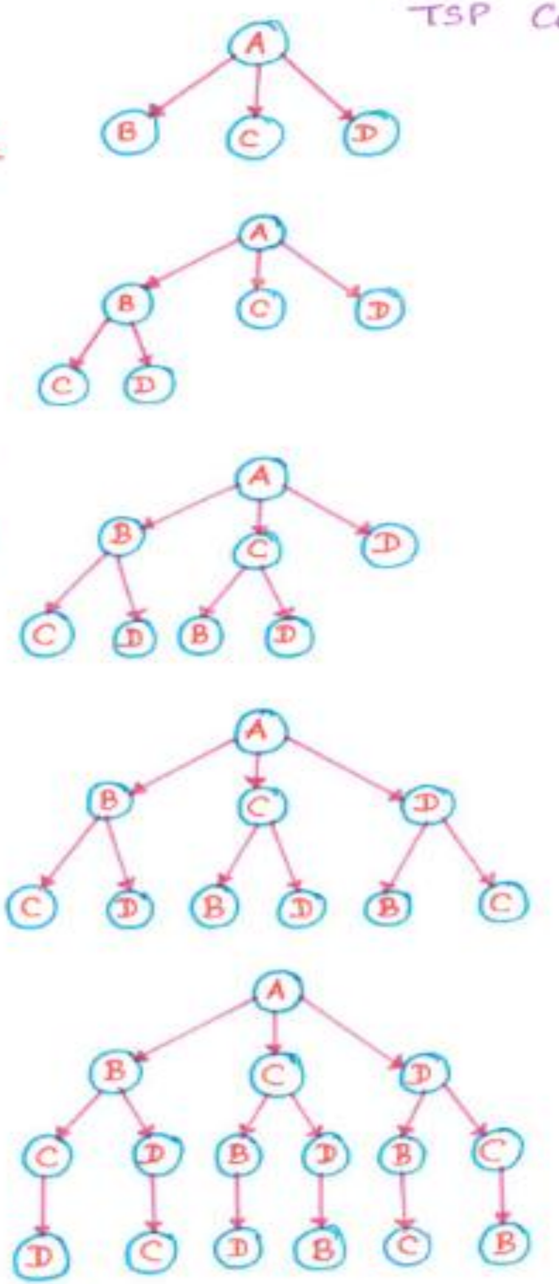
Soln. To find minimum Cost

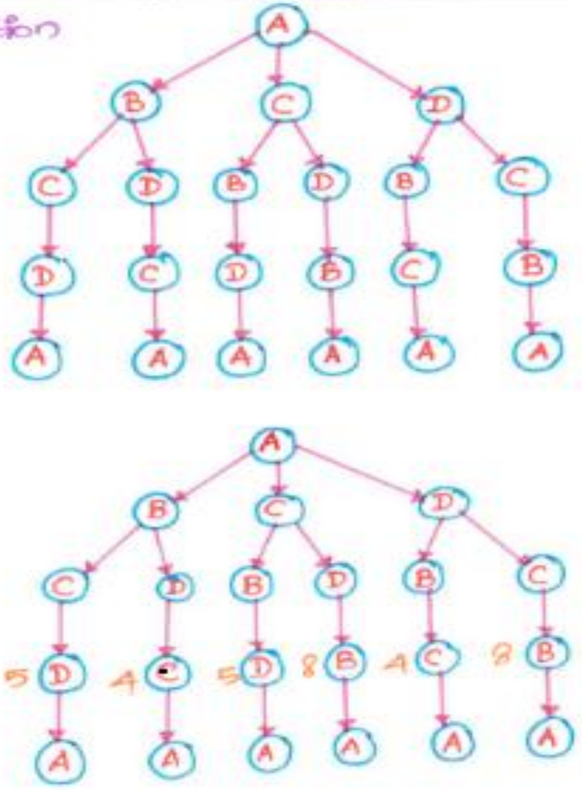
Example:

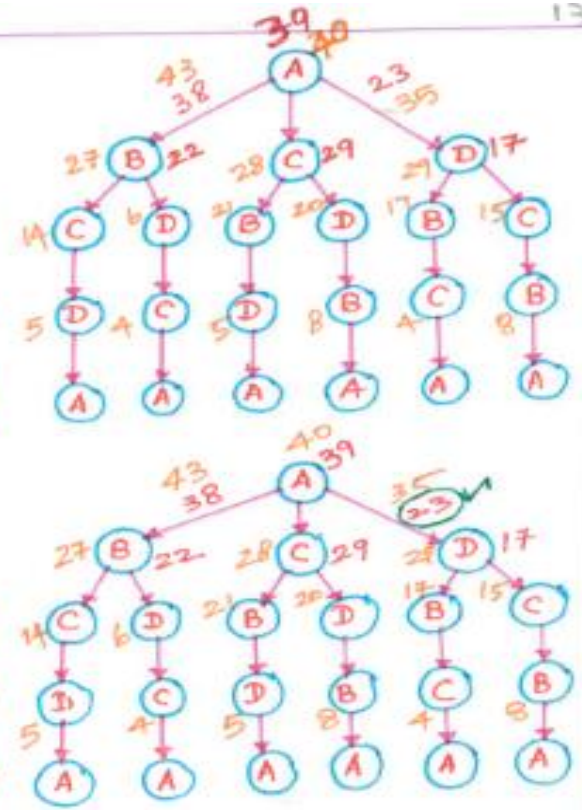


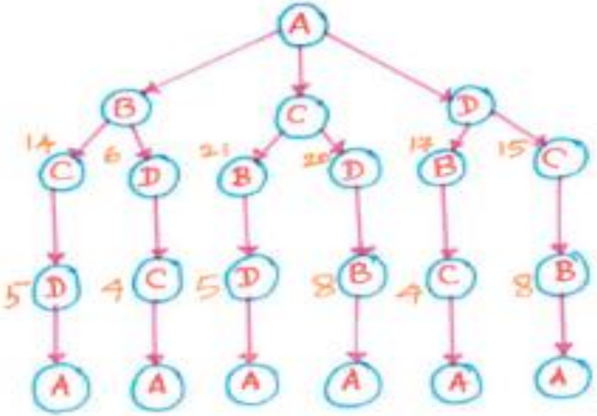
| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 16 | 11 | 6 |
| B | 8 | 0 | 13 | 16 |
| C | 4 | 7 | 0 | 9 |
| D | 5 | 12 | 2 | 0 |

TSP Calculation










| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 16 | 11 | 6 |
| B | 8 | 0 | 13 | 16 |
| C | 4 | 7 | 0 | 9 |
| D | 5 | 12 | 2 | 0 |


$$g(i,s) = \min\{ w(i,j) + g(j, (s-j))$$

# BACK TRACKING AND 'N' QUEEN

## Back tracking :-

Variation of exhaustive search Applications such as "n" queen Problem, sub of subsets, graphs coloring.

**General method :-**

Desired solution :- exp - "n" tuples $(x_1, x_2, \dots x_n)$ $x_i$ (chosen from set).

Objective → maximize (or) minimize (or) satisfy the criteria.

**"N" queen Problem :-**

* Consider a chess board can have "n" queen.

Condition → No queen attack each other in diagonal horizontal (or) vertical.

**To solve 4x4 queen :-**

* To place a queen in 1st Position.
* Then place a queen (2) using unsuccessful places (1,2), (2,1), (2,2) + (2,3).
* Back track all the way upto queen 1 and then move to (1,2).
* Now place a queen at (1,2) "2" at (2,4), "3" at (3,1) and "4" at (4,3).

**Algorithm queen :-**

Input → total no. of queen "n" for column < 1 to "n" begin.

if place → queen (row, col). if (row<n) then

Print _ board (n)

**Back tracking - advantages :-**

Partial vector generalised does not lead to an optimal solution — uses depth — (shift search —) with some bounding function.

Constraint → Implict → Explict

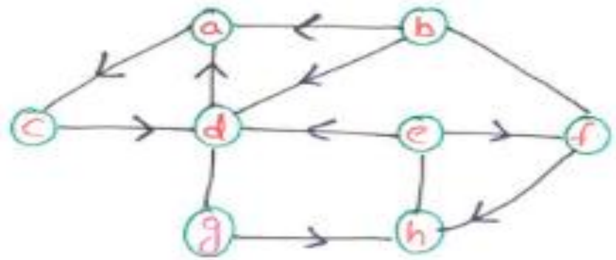
tuples in the solution space

Each vector element to be choosen from the set.

**Hamiltonian circuit Program :-**

Consider a undirected connected graph — with two nodes x + y.

objective → to find a path from x to y visiting each node in a graph exactly.



a → c → d → g → h → e → f → b → a

Sick space is generated to find all the hamiltonian cycles. A → B → B → E → C → f → A

**Algorithm H-cycle :-**

```
{
  Repeat
    begin
      next_ vertex [k]
  if(x[k] = 0) then write (x[1 : n])
  else
    cycle (x+1)
}
```

**Applications :-**

computer graphics, electronic circuit design, mapping genomes and operational research.

**Hamilton cycle :-**

* A cycle that use → every vertex exactly onces.

**Hamilton Path :-**

* Path that uses every vertex in a graph exactly onces.

## Assignment problems

there 'n' people to when 'n' jobs to be assigned conditions.

The total cost of assignment → small ass possible

Idea.

Each element in a row (each) to be selected

→ no two selected elements are in same column.

**Problem**

| | $J_1$ | $J_2$ | $J_3$ | $J_4$ |
|---|---|---|---|---|
| $P_1$ | 10 | 2 | 7 | 8 |
| $P_2$ | 6 | 4 | 3 | 7 |
| $P_3$ | 5 | 8 | 1 | 8 |
| $P_4$ | 7 | 6 | 10 | 4 |

four persons alloted four '4' jobs. there '24' possible ways to predict all the values.

⇓

| | | | | | |
|---|---|---|---|---|---|
| 10 | 2 | 7 | 8 | $P_1$ | $P_1 \to 2$ ($J_2$) |
| 6 | 4 | 3 | 7 | $P_2$ | $P_2 \to 6$ ($J_1$) |
| 5 | 8 | 1 | 8 | $P_3$ | $P_3 \to 1$ ($J_3$) |
| 7 | 6 | 10 | 4 | $P_4$ | $P_4 \to 4$ ($J_4$) |

**Back tracking**

Algorithm for capt -uring some or all solutions to given computional issues, especially for constraint satisfaction

**Branch of bond**

An Algorithm to find the optimal solution to many optimization problems, especially in discreate and combinatrol optimization.

### knapsack problem

To find the most valuable subset of the items → that are fit in the knapsack each

(Aim)

↳ Select object having the same point

Arrange the weighed-value pairs $v/w_1 \geq v_2/w_2 \geq v_3/v_3 \cdot v_n/w_n$.

The state space tree is

$v_b = v_f\ (W - w_i)\ (v_{i+1}/w_{i+1})$

v → profit value earned  W → knapsack Capacity

$v_b$ – Upper bond  w → weighted object to be placed

Problem statement

'n' objects of 'm' capacity of knapsack To make maximation object to a solution is

Minimize profit $- \sum_{i=1}^{n} p_i x_i$

Sub to $\sum_{i=1}^{n} w_i x_i$

Such that $\sum w_i x_i \leq m$ and $x_i = 0$ (or) 1

where $1 \leq i \leq n$

problem

| Item | weight | value |
|---|---|---|
| 1 | 4 | $40 |
| 2 | 7 | $42 |
| 3 | 5 | $25 |
| 4 | 3 | $12 |

| Item | weight | val | v/w |
|---|---|---|---|
| 1 | 4 | $40 | 10 |
| 2 | 7 | $42 | 6 |
| 3 | 5 | $25 | 5 |
| 4 | 3 | $12 | 4 |

knapsack weight → to

Sample calculation

$v = 0,\ w = 0,\ i = 0$

$v_{i+1}/w_{i+1} \Rightarrow v_i/w_i$

$v_b \Rightarrow v + (W - w_i)/(v_{i+1}/w_{i+1})$

$\Rightarrow 0 + (10 - 0)/(40/4)$

$\Rightarrow 10 \times 10$

$= 100$

Computation at node I

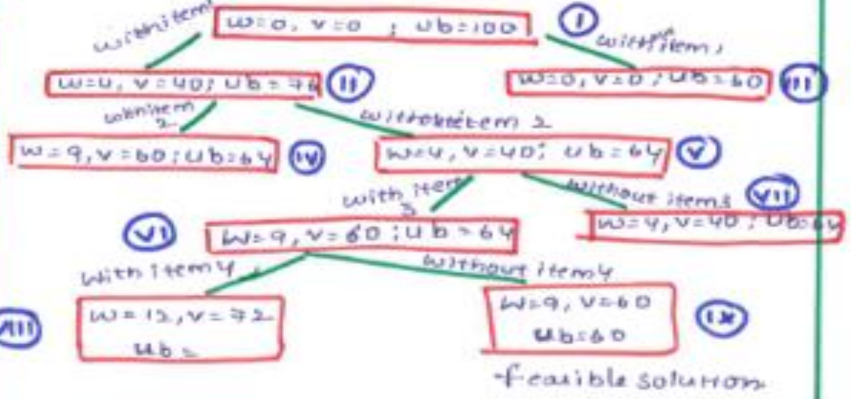
i.e.. root of state space tree

Initally, $w=0$, $v=0$ and $v_{i+1}/w_{i+1} - v_1/w_1 = 40/4 = 10$

The capacity $W = 10$

$\therefore\ ub = v + (W - w)\ v_{i+1}/w_{i+1}$

$= 0 + (10-10)(10) \Rightarrow ub = 100.$



## Sum of subset problems

Let $s = \{s_0, s_1, s_2, \ldots s_n\}$ be a set of 'n' +ve integers solution whose sum is equal to +ve integers (d)

### Problem statement

* First arrangement in ascending order → 's' → set of elements
  'd' → expeded sum of subsets

### Steps

1. start with empty set.
2. Add subset - next set from the list.
3. subset have 'sum' - (d) set the solution
4. If the subset is not feasable → reached the end of subset
5. If the subset is feasible → repeat step 2
6. visited all the elements - f(x) to find a suitable subset.

Problem - the given set
(S) = {6, 2, 8, 1, 5}
sum show be 9

| | | |
|---|---|---|
| (1) | 1 | |
| (1,2) | 3<9 | Add next element |
| (1,2,5) | 8<9 | Add next element |
| (1,2,5,6) | 14 | sum exceed |
| (1,3,5,8) | 16 | check constaint |
| (1,2,6) | 9=9 | solution is found |

These are 's' distinct no's combination of that numbers whose sum = 9 ⇒ set = {(1,2,6), (1,8)}

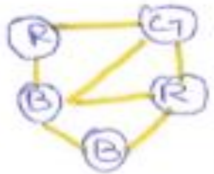
## Graph coloring

Graph coloring → procedure of assignment of colors

Objective:-
No adjacent vertex have the same colour

### chromatic number

minimun 'no' of color → to color of all the nodes in a graph (G)




R - Red
B - Blue
G - Green
Chromatic no - 3

### Application

- M colouring problem
- Bi-connected graph
- Graph databases

## Branch and Band

'state space' of all possible solution is generated

### Condition

'Bounded value' of the same node is not better then the best solution
→ then corresponding node is not Expanded
node → defined as non-promising node.

### Search

Live node in First in first search

| LIFO | FIFO |
|---|---|
| the branch is extended that every first child discovered | Always the oldest node in the queue is (First in first out search) |

### General method

* For Exploring new nodes either BFS or D-search technique can be used.
* BFS-like state space search will be called FIFO
* D search like state space search will be called LIFO

### Selection of answer Node

The partitioning has done at each node of the tree. We compete lower bound and upper bound of the tree. This computation lead to selection of answer node

# TRAVELLING SALESMAN PROBLEM

Travelling Salesman Problem (TSP):

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns back to starting point.

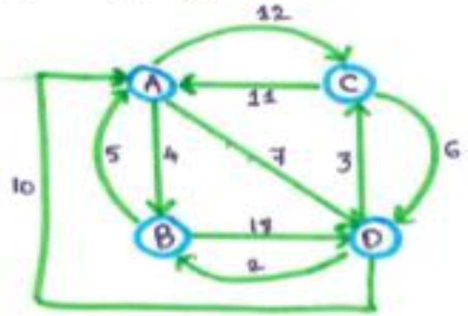
Time Complexity: O(N!), As for the first node there are N possibilities and for the second node there are n-1 possibilities.

[for N nodes] $N^*(N-1)^* \dots 1 = O(N!)$

Auxilary Space: O(N)

## Problem:

Solve Travelling Salesman Problem using Branch Algorithm in the following graph-




**Step-1:**

Write the initial cost matrix & reduce.

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 4 & 12 & 7 \\ B & 5 & \infty & \infty & 18 \\ C & 11 & \infty & \infty & 6 \\ D & 10 & 2 & 3 & \infty \end{array}$$

Row Reduction:

If; row contain '0' – no need to reduce

row doesn't contain '0' – reduce that row

- select the least value element
- Subtract that element from each element
- this will create any entry '0' in that row, so reduce that row.

Row reduced Matrix:

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 0 & 8 & 3 \\ B & 0 & \infty & \infty & 13 \\ C & 5 & \infty & \infty & 0 \\ D & 8 & 0 & 1 & \infty \end{array}$$

**Column Reduction:**

If, column contain 'o' – no need to reduce

column doesn't contain 'o' – reduce that column.

- Select the least value element
- subtract the element from each element
- This will create the entry 'o' to that column, so reduce that column.

**Column Reduced Matrix:**

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 0 & 7 & 3 \\ B & 0 & \infty & \infty & 13 \\ C & 5 & \infty & \infty & 0 \\ D & 8 & 0 & 0 & \infty \end{array}$$

Cost of node-1 by adding the reduced elements

Cost (1) = sum of all reduction elements

= 4 + 5 + 6 + 2 + 1 = 18

**Step-2:**

- we consider all other vertices by one by one.
- select the best vertex where we can land upon to minimize the tour cost.

* Choosing to Go to Vertex-C : Node 3 (A→C)

from reduced matrix, M[A,C] = 7

Set row-A & column-C to ∞

Set M[C,A] = ∞

**Resulting Cost Matrix:**

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & 13 \\ C & \infty & \infty & \infty & 0 \\ D & 8 & 0 & \infty & \infty \end{array}$$

**Step-3:** explore vertices B & D from n-3.

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & 13 \\ C & \infty & \infty & \infty & 0 \\ D & 8 & 0 & \infty & \infty \end{array}$$

cost(3) = 25

Choosing to go to vertex-B : Node 5 (A→C→B)

- From reduced matrix, M[C,B] = ∞
- row C & column-B to ∞
- set M[B,A] = ∞

resulting matrix:

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & 13 \\ C & \infty & \infty & \infty & \infty \\ D & 8 & \infty & \infty & \infty \end{array}$$

**Step-4:**

- We explore vertex-B from node-6.
- Start with the cost matrix at node – 6:

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & 0 & \infty & \infty & \infty \\ C & \infty & \infty & \infty & \infty \\ D & \infty & 0 & \infty & \infty \end{array}$$

Cost(6) = 25

Choosing to go to vertex-B:

Node-7 (Path A→C→D→B)

* From reduced matrix of step-3, M[D,B] = 0
* Set row-D & column-B to ∞
* Set M[B,A] = ∞

**Resulting Matrix:**

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & \infty & \infty & \infty \\ B & \infty & \infty & \infty & \infty \\ C & \infty & \infty & \infty & \infty \\ D & \infty & \infty & \infty & \infty \end{array}$$

- we reduce this matrix
- then, we find out the cost of n-7

cost(7):

= cost(6) + Sum of reduction elements + M[D,B]

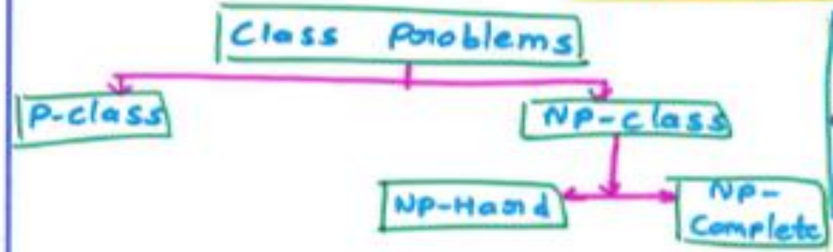
= 25 + 0 + 0 = 25

Thus,

→ Optimal path is : A → C → D → B → A

→ Cost of optimal path = 25 units.

# TRACTABILITY - CLASS PROBLEMS




P-class → Polynomial problems
→ can be solved in polynomial time.
→ Tractable problems.
→ Time complexity
- O(1) - constant time
- O($\log_2 n$) - logarithmic time
- O(n) - linear time
- O($n^2$) → Quadratic time
- O($n^k$) → polynomial time
→ (n>k)
→ Easy to solve
→ Easy to verify.

Examples
- → Sorting
- → Searching
- → Basic operations addition, subtraction, multiplication, division.
- → Matrix multiplication.
- → Floyd's algorithm

NP-class → Non-Deterministic polynomial
→ Cannot be solved on polynomial time
→ But can be verified in polynomial time
→ Intractable / hard problems.
→ Exponential time problems.
→ Time complexity
- → O($k^n$) - Exponential time.
- → O(n!) - Factorial time.

Examples
- → Travelling sales man problems
- → knap sack problem
- → Hamiltonian cycle
- → Su-Do-ku

## P-class
Easy to solve
Easy to verify
Polynomial time
Tractable

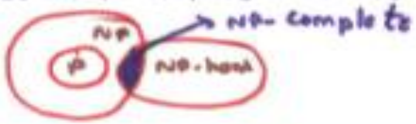

$\boxed{P \subseteq NP}$

## NP-class
Hard to solve
Easy to verify
Exponential time
Intractable.

## NP-Complete problems
- → Quick to verify
- → Slow to solve
- → can be reduced to another NP-complete problem.
- → A problem is in NP-Hard if all problems in NP are polynomial time reducible to it.
- → A problem is in NP-Complete if the problem is both in NP-Hard & NP.
- → NP-Complete are decision problem Reduction (∝)


→ problem A reduces to problem B iff there is a way to solve A by deterministic algorithm that solve B in polynomial time.

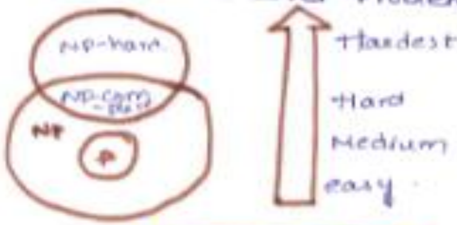
Properties.
1. If A is reducible to B and B is in P, then A in P.
2. A is not in P implies B is not in P.

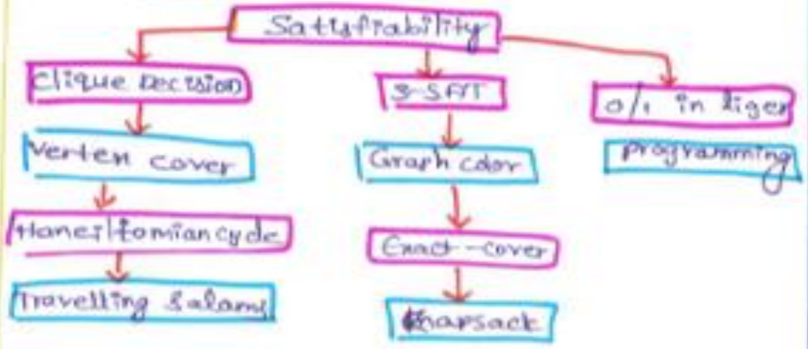



- → All NP-Complete problems are NP-Hard but all NP-Hard problems are not NP-Complete.
- → Example for NP-Complete
- → Circuit SAT problem (Satisfiability)

## NP-HARD PROBLEMS
- → Hard to solve
- → Hard to verify.
- → Not decidable
- → optimization problems.
- → can be reduced to another NP problem.

Examples:
- → k-means clustering.
- → Travelling Salesman problem.
- → Graph colouring
- → Set cover problem
- → Vertex cover problem.



# P, NP Problems

Solution of Algorithms
- Polynomial time
- Non polynomial time

## Polynomial time problems

Solved in polynomial time using deterministic algorithms

## Exponential problems

Solved in non-determinism algorithms

Polynomial time problems
- → searching o(logn)
- → All pair shortest o($n^3$)
- → Matrix multiplication (o($n^3$))

Exponential time
- → TSP o($2^n$)
- → knapsack o($2^n$)
- → graph colouring o($nm^n$)

**Deterministic Algorithms** - can solve the problem in polynomial in time

$q_0 \xrightarrow{a} q_1 \rightarrow$

**Non deterministic Algorithms** - can possibilities for every solution

$q_0 \xrightarrow{a} q_1$, $q_0 \xrightarrow{a} q_2$, $q_0 \xrightarrow{a} q_3$

# Tractable & Intractable Problems

| ↓ | ↓ |
|---|---|
| Easy problems | hard problems |
| Ex:- Merge sort, matrix | Ex:- TSP, 0/1 knapsack |

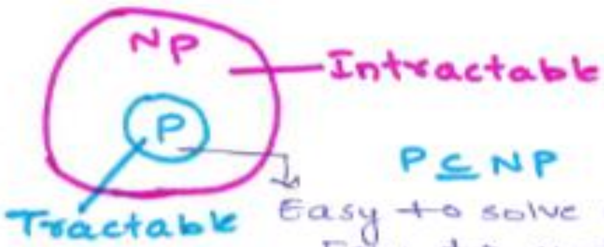
## P class Problem

A problem which can be solved on polynomial time

Ex:- All sorting & searching algorithms

## NP class problem

A problem which cannot be solved on polynomial time but is verified in polynomial time is known as Non determinism or NP class problem.




$P \subseteq NP$

Easy to solve & Easy to verify polynomial time

$A \xrightarrow[\text{polynomial time}]{\text{reduce}} B$

Let A & B are two problems, then A reduces to problem B iff then there is a way to solve A by deterministic algorith that solve B in polynomial time

## Properties:-

1. If A is reducable to B, and B in P the A in P
2. A is not in P implies B is not in P.

## compitational complexity problem

- P class
- NP class
  - NP hard — A problem is NP hard if every problem in NP can be polynomial reduced to it.
  - NP complete — A problem is NP-complete if it is in NP as it it NP-hard

$A \propto B$

## NP Complete

→ SAT problem (statisfiability) problem determines if then exist a set of boolean variables.

## Approximation Algorithms

- Exact Algorithm — Exact solution
- Approximation Algorithm — Near optimal solution

# NEAREST - NEIGHBOUR ALGORITHM

## Approximation Algorithms for NP-hard Problems

How to handle difficult problems of combinational optimization, such as travelling salesman problem and the knapsack problem, these problems are NP-complete.

→ The optimization versions of such difficult combinational problems fall in the clam of NP-Hard problems that are at least as hard as NP complete problems

**Polynomial - time approximation**

algorithm is a c-approximational. its performance ratio is at most c.

$$f(sa) \le c f(s^*)$$

↓ approximation solution. ↓ Exact solution

$$\boxed{\text{Accuracy ratio } r(sa) = \frac{f(sa)}{f(s^*)}}$$

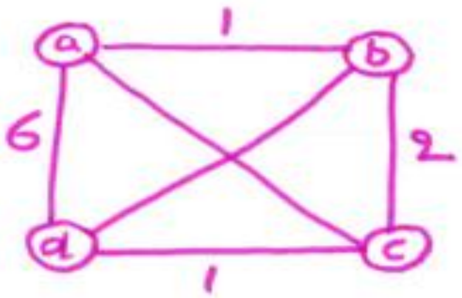
**Nearest-neighbour algorithm**

* The following simple greedy algorithm is based on the nearest neighbour heuristic. :- * the idea of always going to the nearest unvisited city next.

**Step-1** :- choose as the arbitary city as the start

**Step-2** :- Repeat the following operation until all the cities have been visited go the unvisited city nearest the one visited last.

**Step 3** :- Return to the starting city.




Instance of the travelling salesman problem for illustrating the nearest neighbour algorithm for the above diagram as a starting vertex the nearest neighbour algorithm yeilds to the tour (Hamiltonian circuit)

Sa: a-b-c-d-a of length 10.

The optimal solution, can be easily checked by exhaustive search, is the tour.

$s^*$: a-b-d-c-a of length 8

The accuracy ratio of this approximation is

$$\boxed{\begin{aligned} r(sa) &= \frac{f(sa)}{f(s^*)} \\ &= \frac{10}{8} \\ &= 1.25 \end{aligned}}$$

Tour Sa is 25.1 longer than the optimal tour $s^*$.

An algorithm that return near optimal solution is called Approximation Algorithm

Given an optimization problem P, an algorithm A is said to be an approximation algorithm for P, if for any given instance I. it returns an approximate solution that is feasible solution

**Approximation ratio P(n)**

let cost of the optimal solution $= c^*$

let cost of the solution produced by the approximation algorithm is $c$

$$\boxed{e(n) \ge \max\left(\frac{c}{c^*}, \frac{c^*}{c}\right)}$$

# Approximation Algorithm

## Minimum Spanning Tree

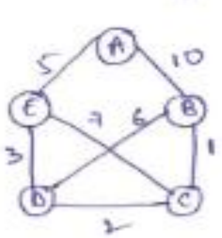
A spanning tree of a graph G is a subgraph which is basically a tree and it contains all the vertices of G containg no circuit

## Minimum Spanning tree

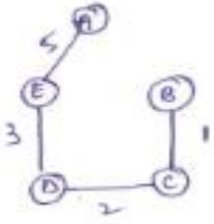
A minimum spanning tree of a weighted connected graph G is a spanning tree with minimum or smallest weight.

## Weight of the Tree

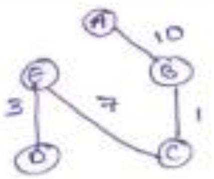
A weight of tree is defined as the sum of weights of all its edges



Graph G



$w(T_1) = 11$

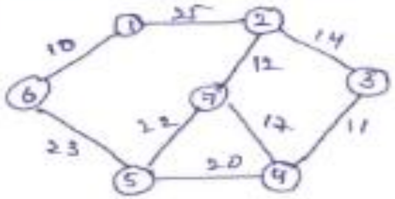


$w(T_2) = 21$

## Applications of spanning trees

1. Spanning trees are very important designing efficient routing algorithm
2. It is used for N/w design.

## Prim's Algorithm



→ Select an edge with minimum weight. The algorithm proceeds by selecting adjacent edges with minimum weight

→ No circuit

Step 1:-

TW = 0

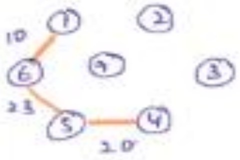
step 2:-

TW=10

Step 3:-

TW=33

Step 4:-

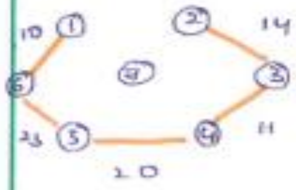


TW=53

Step 5:-

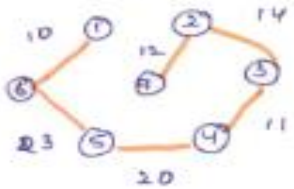
TW=64

Step 6:-



TW=78

step 7:-



Total weight=90

## Prim's Algorithm

```
Prim [G[0 ... size-1, 0 ... size-1] nodes)
  for i ← 0 to nodes-1 do
    tree[i] ← 0
    tree[0] = 1
  for k ← 1 to nodes do
  {
  min dist ← ∞
  for i ← 0 to nodes -1 do
  {
  for j ← 0 to nodes -1 do
  {
  if (G[i,j] and ((tree[i] and !
    tree[j]) or
  (!tree[i] and tree[j]))) then
  {
  if (G[i,j] < min-dist) then
  {
   min-dist ← G[i,j])
     v1 ← i
     v2 ← j
   }
   }
   }
   }
write (v1, v2, min-dist);
tree[v1] ← tree[v2] ← 1
total ← total + min-dist
}
write (" Total path length is", total" }.
```

## TRAVELLING SALESMAN PROBLEM

**GIVEN**

Set of cities along with the cost of travel.

**TO FIND**

The cheapest route visiting all cities and returning to starting point.

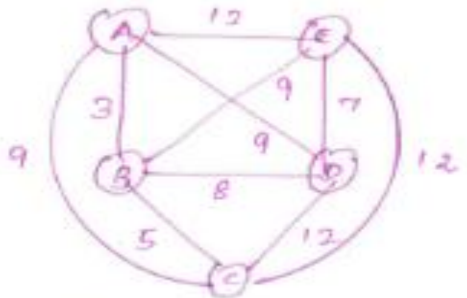
ALGORITHM 1 (TWICE AROUND-THE-TREE)

STEP 1: Compute minimum spanning tree for the given graph

STEP 2: Start at any arbitrary city and walk around the tree and record nodes visited

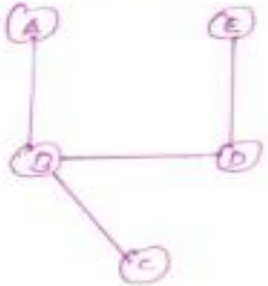
STEP 3: Eliminate duplicates from the generated node list

EXAMPLE




STEP 1: Obtain MST

STEP 2: Start from A and have




DFS walk

STEP 3: Record visited nodes

A – B – C – B – D – E – D – B – A. Eliminate duplicates A – B – C – D – E – A, which is Hamiltonian circuit.

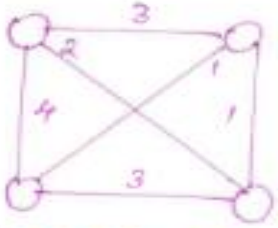
Not a optimal tour.

ALGORITHM 2 (NEAREST NEIGHBOUR)

STEP 1: Start at any city

STEP 2: Repeat until all the nodes are visited: Go to nearest city (the unvisited) each time.

STEP 3: Return to starting city.




2+4+1+1 = 8

## KNAPSACK PROBLEM

STEP 1: Compute value/weight ratio

STEP 2: Sort the items in non-increasing order of $v_i/w_i$

STEP 3: Repeat until no item is left

a. If current item fits in use it

b. Otherwise take its largest fraction to fill the knapsack to its full capacity.

| Item | Weight | Value |
|---|---|---|
| 1 | 7 | $49 |
| 2 | 3 | $12 |
| 3 | 4 | $42 |
| 4 | 5 | $30 |

capacity $W = 10$

Optimal Soln:

| item | Weight | Value | Value to weight |
|---|---|---|---|
| 3 | 4 | $42 | 10.5 |
| 1 | 7 | $49 | 7 |
| 4 | 5 | $30 | 6 |
| 2 | 3 | $12 | 4 |

This is the optimal solution.

SIMATS

SCHOOL OF ENGINEERING

Engineer to Excel

Approved by AICTE | IET-UK Accreditation

Saveetha Nagar, Thandalam, Chennai - 602 105, TamilNadu, India