UNIT-I

DESIGN and ANALYSIS of ALGORITHMS

TOPIC-1: INTRODUCTION

Presented By:

D. PHANI KUMAR

Assistant Professor

Department of CSE

GIET-[A]

Lecture Details:

Subject : Design and Analysis of Algorithms

Topic : Introduction

Branch/Year : CSE, III-B.Tech I-Semester

UNIT-1	TOPIC	DESCRIPTION
Introduction	TOPIC-1	➢ **Algorithm & Pseudo code** ➢ **Performance Analysis** ➢ **Asymptotic Notations**

Course Outcome

Student able to:

- ✓ Relate various algorithm design situations
- ✓ Identify Space complexity, Time complexity (Performance Analysis)
- ✓ Categorize Asymptotic notations.

Prerequisite(s)

Basic knowledge on:

- ✓ Algorithm Design
- ✓ Programming Language concepts

Introduction

- **Design Algorithm** refers to a method or a mathematical process for problem-solving and engineering **algorithms**.
  The **design** of **algorithms** is part of many solution theories of operation research, such as dynamic programming and divide-and-conquer.

- **Design Analysis** is essentially a decision-making process in which analytical tools derived from basic sciences, mathematics, statistics, and engineering fundamentals are utilized for the purpose of developing a product model that is convertible into an actual product.

Text/Reference Books

- **Introduction to Algorithms**, second edition,
  T.H. Cormen, C.E.Leiserson, R.L.Rivest and C.Stein, PHI Pvt. Ltd

- **Fundamentals of Computer Algorithms**,
  Ellis Horowitz, Satraj Sahni and Rajasekharam, Universities Press.

- **Design and Analysis of algorithms**,
  Aho, Ullman and Hopcroft,Pearson education.

Algorithm

An **Algorithm** is a step-by-step procedure, with a set of operations designed to perform a specific task.

- An **Algorithm** is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.
- Algorithms are always unambiguous and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks.

An **Algorithm** is a sequence of step to solve a specific problem.

Algorithm

Characteristics of an algorithm:

- ✓ Must take **input**
- ✓ Must give some **output**
- ✓ **Definiteness**: Each instruction is clear and unambiguous.
- ✓ **Finiteness**: Algorithm terminates after a finite number of steps.
- ✓ **Effectiveness**: Every instruction must be basic i.e. simple instruction.
- ✓ **Correctness**: Correct, Produce an incorrect answer & approximation algorithm
- ✓ **Less resource**: Use less resources (time and space).

Pseudo Code

Pseudo code simply an implementation of an algorithm in the form of annotations and informative text written in plain English.

- **Pseudo code** is an artificial and informal language that helps programmers develop algorithms. Pseudo code is a "text-based" detail (algorithmic) design tool.
- The rules of Pseudo code are reasonably straightforward. These include while, do, for, if, switch. Examples below will illustrate this notion.

Advantages of Pseudo code:

- ✓ Improves the readability of any approach. It's one of the best approaches to start implementation of an algorithm.
- ✓ Acts as a bridge between the program and the algorithm or flowchart. Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out.
- ✓ The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.

Algorithm

Algorithm to check whether the given number is even or odd:

START

Step 1 → Take integer variable A

Step 2 → Assign value to the variable

Step 3 → Perform A modulo 2 and check result if output is 0

Step 4 → If true print *A is even*

Step 5 → If false print *A is odd*

STOP

Pseudo Code

Pseudo code to check whether the given number is even or odd:

1. **START**
2. **DISPLAY** "Enter the Number - "
3. **READ** number
4. **IF** number **MOD** 2 = 0 **THEN**
   DISPLAY "Number is Even"
   ELSE
   DISPLAY "Number is Odd"
   END IF
5. **STOP**

DoIt::

Chat box:

In simple terms differentiate the **Algorithm** and **Pseudo code**

Time complexity: The amount of time required to run an algorithm in terms of the size of the input.

"Time" can mean

- ✓ the number of memory accesses performed,
- ✓ the number of comparisons between integers,
- ✓ the number of times some inner loop is executed, or
- ✓ some other natural unit related to the amount of real time the algorithm will take.

Space complexity: The amount of space or memory taken by an algorithm to run as a function of the length of the input.

The actual running time depends on a variety of backgrounds:

- ✓ the speed of the Computer,
- ✓ the language in which the algorithm is implemented,
- ✓ the compiler/interpreter,
- ✓ skill of the programmers etc.

- Suppose given an array **A** and an integer **key** and have to find if **key** exists in array **A**.
- Simple solution to this problem is traverse the whole array **A** and check if the any element is equal to **key**.

```
for i : 1 to length of A
    if A[i] is equal to key
        return TRUE
return FALSE
```

The number of lines of code executed is actually depends on the value of **key**.

- Each of the operation in computer take approximately constant time (**c**).

- In the **worst case**, the **if** condition will run **N** times where **N** is the length of the array **A**.
- Total execution time will be **(N * c + c)**

  N * c for the **if** condition and

  c for the **return** statement
- The total time depends on the length of the array **A**.
- If the length of the array will increase the time of execution will also increase.

- **Order of growth** is how the time of execution depends on the length of the input.
- In the previous example, the time of execution is linearly depends on the length of the array.
- Order of growth will help to compute the running time with ease.
- Different notation are there to describe limiting behavior of a function.

Space:

S(P) = c+ Sp(Instance characteristics)

Where, c is constant.

Example:

```
Algorithm abc(a,b,c)
{
   return a+b++*c+(a+b-c)/(a+b) +4.0;
}
```

In this algorithm sp=0;

Let assume each variable occupies one word, then the space occupied by above algorithm is **>=3**.

S(P)>=3

Example:

```
Algorithm sum(a,n)
{
    s=0.0;
    for i=1 to n do
    s= s+a[i];
    return s;
}
```

In the above algorithm **n**, **s** occupies one word each and array "**a**" occupies **n** number of words and **i** occupies **n+1** number of words so

S(P)>=2n+3

Example:

```
Algorithm RSum (a, n)
{
    if(n<=0) then return 0.0;
    else
    return RSum(a,n-1)+a[n];
}
```

In the above recursion algorithm, the space need for the values of **n**, **return address** and **pointer to array**. The above recursive algorithm depth is **(n+1)**. To each recursive call we require space for values of **n**, **return address** and **pointer to array**. So the total space occupied by the above algorithm is **S(P) >= 3(n+1)**

Performance Analysis

```
sum(a,n)
{
      s= 0.0;          // count = count+1;
      for i=1 to n do // count = count+1;
      {
            s=s+a[i];  // count = count+1;
      }
      return s;     // count = count+1;
}
```

n+1 time's

n time's

Each time a statement in the program is executes, **count** is incremented by the step of that statement

total of **2n+3** steps

Performance Analysis

Statement	Steps per execution	Frequency	Total Steps
sum(a,n)	0	-	0
{	0	-	0
s= 0.0;	1	1	1
for i=1 to n do	1	n+1	n+1
{	0	-	0
s=s+a[i];	1	n	n
}	0	-	0
return s;	1	1	1
}	0	-	0
			2n+3

- In above example, if we analyze carefully frequency of

  "**for count = 1 to n do**"

  it is '**n+1**' this is because the statement will be executed one time more die to condition.

- Once the total steps are calculated they will resemble the instance characteristics in time complexity of algorithm.

- The repeated compile time of an algorithm will also be constant every time. we compile the same set of instructions so we can consider this time as constant '**C**'.

- Therefore the time complexity can be expressed as:

Time(Sum) = C + (2n +3)

- **Asymptotic analysis** of an algorithm refers to defining the mathematical boundation/framing of its run-time performance.
- Using asymptotic analysis, we can very well conclude the
  - ✓ **Best case**: Minimum time required for program execution
  - ✓ **Average case**: Average time required for program execution
  - ✓ **Worst case**: Maximum time required for program execution
  - ✓ **Amortized**: A sequence of operations applied to the input of size a averaged over time

Asymptotic Notations

The commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- → **O** Notation (Big O Notation)
- → **Ω** Notation (Omega Notation)
- → **θ** Notation (Theta Notation)
- → **o** Notation (Little o Notation)

O Notation (Big O Notation)

'**O**' (**Big Oh**) is the most commonly used notation. It is used to find the upper bound time of an algorithm , that means the maximum time taken by the algorithm.

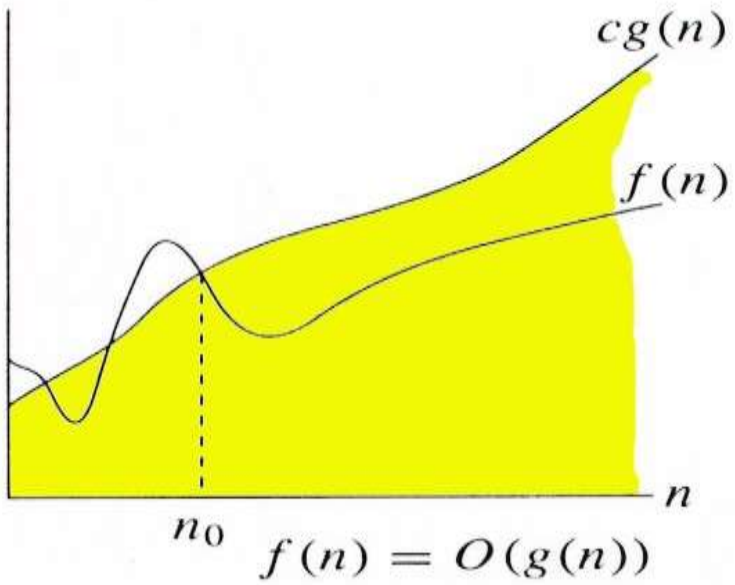
Definition : Let f(n), g(n) are two non-negative functions. If there exists two positive constants **c** ,$\mathbf{n_0}$. such that **c>0** and for all n>=$\mathbf{n_0}$ if

$$f(n) \leq c.g(n)$$

then we say that

f(n)=O(g(n))

Hence, function $g(n)$ is an upper bound for function $f(n)$**, as** $g(n)$ grows faster than $f(n)$**.**



Example:

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$.

Considering $g(n)= n^3$,

$f(n) \leq 5.g(n)$ for all the values of $n>2$.

Hence, the complexity of $f(n)$ can be represented as $O(g(n))$, **i.e.** $O(n^3)$.

Ω Notation (Omega Notation)

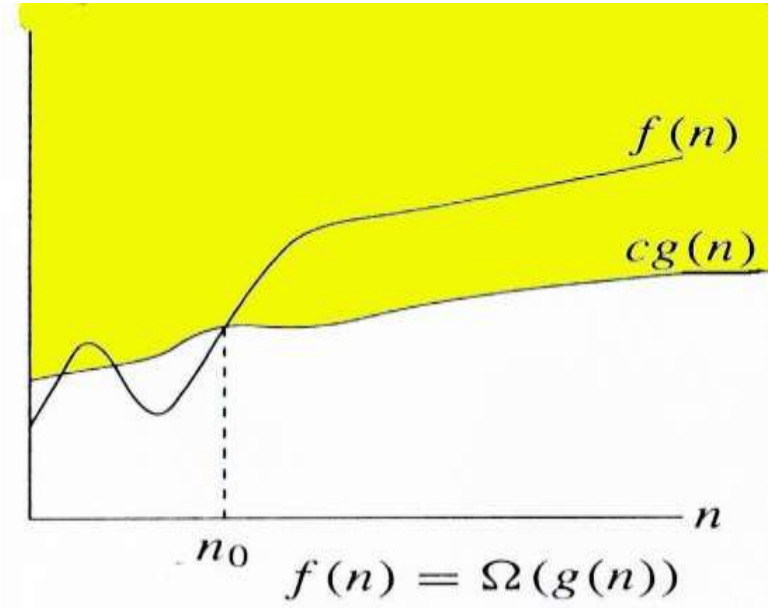
It is denoted by '**Ω**' (**Omega**), used to find the lower bound time of an algorithm, that means the minimum time taken by an algorithm.

Definition : Let f(n), g(n) are two non-negative functions. If there exists two positive constants **c**, $\mathbf{n_0}$. such that **c>0** and for all **n>=n_0**, if

f(n)>=c*g(n)

then we say that

f(n)=Ω(g(n))



Ω Notation (Omega Notation)

Example:

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$.

Considering $g(n)= n^3$, $f(n)\geq 4.g(n)$ for all the values of $n>0$.

Hence, the complexity of $f(n)$ can be represented as $\Omega(g(n))$, i.e. $\Omega(n^3)$.

θ Notation (Theta Notation)

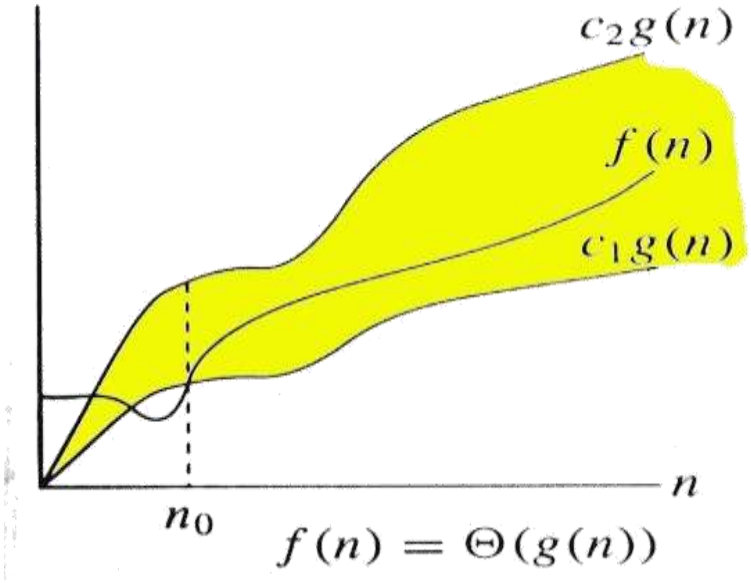
It is denoted by 'Θ' (Theta), used to find the time in-between lower bound time and upper bound time of an algorithm.

Definition : Let f(n), g(n) are two non-negative functions. If there exists positive constants c_1, c_2 and n_0. such that $c_1>0, c_2>0$ and for all $n>=n_0$, if

$$c_1*g(n)<=f(n)<=c_2*g(n)$$

then we say that

$$f(n)=\Theta(g(n))$$



θ Notation (Theta Notation)

Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$.

Considering $g(n)= n^3$, $4.g(n) \leq f(n) \leq 5.g(n)$ for all the large values of **n.**

Hence, the complexity of $f(n)$ can be represented as **Θ(g(n)), i.e.** $\Theta(n^3)$.

o Notation (Little o Notation)

We formally define $o(g(n))$ as the set $f(n)=o(g(n))$ for any positive constant $c>0$ and there exists a value $n_0>0$, such that $0 \leq f(n) \leq c.g(n)$.

Intuitively, in the **o-notation,** the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

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

o Notation (Little o Notation)

Example

Let us consider the same function, $f(n)=4.n^3+10.n^2+5.n+1$.

Considering $g(n)=n^4$,

$$\lim_{n\to\infty}\left(\frac{4\,n\text{^}3\,+\,10\,n\text{^}2\,+\,5\,n\,+\,1}{n\text{^}4}\right)=0$$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))$, i.e. $o(n^4)$.

ω Notation (Little-Omega Notation)

We use **ω-notation** to denote a lower bound that is not asymptotically tight. Formally, however, we define $\omega(g(n))$ as the set $f(n)=\omega(g(n))$ for any positive constant $c>0$ and there exists a value $n_0>0$, such that

$$0 \leq c.g(n) < f(n).$$

For example, $n^2/2=\omega(n)$, but $n^2/2 \neq \omega(n2)$. The relation $f(n)= \omega(g(n))$ implies that the following limit exists

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

ie., $f(n)$ becomes arbitrarily large relative to $g(n)$ as **n** approaches infinity.

ω Notation (Little-Omega Notation)

Example

Let us consider same function, $f(n)=4.n^3+10.n^2+5.n+1$.

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

$$\lim_{n\to\infty}\left(\frac{4\,n^3+\,10\,n^2+\,5\,n\,+\,1}{n^2}\right)=\ \infty$$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))$**, i.e.** $\omega(n^2)$**.**

Amortized analysis is generally used for certain algorithms where a sequence of similar operations are performed.

- Amortized analysis provides a bound on the actual cost of the entire sequence, instead of bounding the cost of sequence of operations separately.
- Amortized analysis differs from average-case analysis; probability is not involved in amortized analysis. Amortized analysis guarantees the average performance of each operation in the worst case.

$$\sum_{1 \le i \le n} amortized(i) \ge \sum_{1 \le i \le n} actual(i)$$

Aggregate Method

The aggregate method gives a global view of a problem. In this method, if **n** operations takes worst-case time $T(n)$ in total. Then the amortized cost of each operation is $T(n)/n$. Though different operations may take different time, in this method varying cost is neglected.

Accounting Method

In this method, different charges are assigned to different operations according to their actual cost. If the amortized cost of an operation exceeds its actual cost, the difference is assigned to the object as credit. This credit helps to pay for later operations for which the amortized cost less than actual cost.

If the actual cost and the amortized cost of i^{th} operation are c_i and $\hat{c}$, then

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

$$p(n)\text{-}p(0) = \sum_{1 \leq i \leq n} (amortized(i) - actual(i))$$

$$p(n)\text{-}p(0) \geq 0$$

Potential Method

This method represents the prepaid work as potential energy, instead of considering prepaid work as credit. This energy can be released to pay for future operations.

If we perform n operations starting with an initial data structure D_0. Let us consider, c_i as the actual cost and D_i as data structure of i^{th} operation. The potential function ϕ maps to a real number $\phi(D_i)$, the associated potential of D_i. The amortized cost $\hat{c}$ can be defined by

$$\hat{c} = c_i + \phi(D_i) - \phi(D_{i-1})$$

Hence, the total amortized cost is

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

The **aggregate method**, where the total running time for a sequence of operations is analyzed.

The **accounting (or banker's) method**, where we impose an ***extra charge*** on inexpensive operations and use it to pay for expensive operations later on.

The **potential (or physicist's) method**, in which we derive a ***potential function*** characterizing the amount of extra work we can do in each step. This potential either increases or decreases with each successive operation, but cannot be negative.

Probabilistic Analysis

P**robabilistic analysis**, analyze the algorithm for finding efficiency. The efficiency of algorithm is also depend upon distribution of inputs. This can be done by analyzing algorithm by the concept of probability.

Example: A company wants to recruiting **k** persons from the **n** persons. To do this the company assigns ranking to all **n** persons depend upon their performance. The rankings of **n** persons from $\mathbf{r_1}$ to $\mathbf{r_n}$. To **n** persons we get **n!** permutations out of **n!** permutations the company selects any one combination that is from $\mathbf{r_1}$ to $\mathbf{r_k}$**.**

Assignment-I

1. Define an algorithm. What are the different criteria that satisfy the algorithm and Explain different techniques to represent an algorithm?

2. Define the terms “Time complexity" and “Space complexity" of algorithms.

3. Write an algorithm to find the 1 to n prime numbers. Determine the frequency counts for all the statements in the algorithm.

Note:

1. *Complete the Assignment-I (**Hand Written**).*
2. ***Scan** the work, convert into **pdf** and **upload.***
3. *Assignment **Posted date 31-Sept-2020***
4. ***Last date** of submission **02-Sept-2020**.*

UNIT-II

DESIGN and ANALYSIS of ALGORITHMS

TOPIC: GREEDY METHOD

Presented By:

D. PHANIKUMAR

Assistant Professor

Department of CSE

GIET-[A]

Lecture Details:

Subject : Design and Analysis of Algorithms

Topic : GREEDY METHOD

Branch/Year : CSE, III-B.Tech I-Semester

UNIT-2	TOPIC	DESCRIPTION
Greedy Method	➢ Greedy Method ➢ Applications	➢ Greedy Method ✓ General Method ➢ Applications ✓ Job Sequencing with Deadlines ✓ Knapsack Problem ✓ Spanning Tree ✓ Minimum Cost Spanning Tree ✓ Single Source Shortest Path Problem

Outcome

Student ableto:

- ✓ Categorize the Greedy Method with various Algorithms
- ✓ Identify and solve the major graph algorithms problems with their analyses
- ✓ Relate various algorithm design situations

Greedy Method

The **simplest** and **straightforward** approach is the **Greedy method**. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

- Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.
- This approach never reconsiders the choices taken previously.
- This approach is mainly used to solve optimization problems.
- In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Components of Greedy Algorithm:

Candidate Set: A solution is created from this set.

Selection Function: Used to choose the best candidate to be added to the solution.

Feasibility Function: Used to determine whether a candidate can be used to contribute to the solution.

Objective Function: Used to assign a value to a solution or a partial solution.

Solution Function: Used to indicate whether a complete solution has been reached.

Job Sequencing with Deadlines

The objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

- Consider set of jobs.
- Each job has a defined deadline and some profit associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs.
- Processor takes one unit of time to complete a job.

Job Sequencing with Deadlines

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

Step-1:

- Sort all the given jobs in decreasing order of their profit.

Step-2:

- Check the value of maximum deadline.
- Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-3:

- Pick up the jobs one by one.
- Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Job Sequencing with Deadlines

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Step-1: Sort all the given jobs in decreasing order of their profit

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Job Sequencing with Deadlines

Step-2:

- ✓ Value of maximum deadline = 5.
- ✓ So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown

Gantt chart

Job Sequencing with Deadlines

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Gantt chart

Find the Job Sequence for the following 2 examples

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	2	1	3
Profit	100	19	27	25	15

Knapsack Problem

Given a set of **items**, each with a **weight** and a **value**, determine a subset of items to include in a collection so that the total weight is **less than or equal to** a given **limit** and the **total value** is as **large** as possible.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack (0/1 Knapsack)

Knapsack Problem

Knapsack is basically means bag. A bag of given capacity.

We want to pack **n** items in your luggage.

- ✓ The **i^{th}** item is worth **v_i** dollars and weight **w_i** pounds.
- ✓ Take as valuable a load as possible, but cannot exceed **W** pounds.
- ✓ **v_i w_i W** are integers.

W ≤ capacity

Value ← Max

Knapsack Problem

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output:

- ✓ To maximize profit and minimize weight in capacity.

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence fractions of items can select. According to the problem statement,

There are **n** items in the store

Weight of **i^{th}** item **$w_i > 0$**

Profit for **i^{th}** item **$p_i > 0$** and

Capacity of the Knapsack is **W**

Items can be broken into smaller pieces, So that only fraction x_i of **i^{th}** item may take.

$$0 \leq x_i \leq 1$$

The **i^{th}** item contributes the weight $x_i.w_i$ to the total weight in the knapsack and profit $x_i.p_i$ to the total profit.

Fractional Knapsack

The objective of this algorithm is to

$$maximize \sum_{n=1}^{n} (x_i . pi)$$

subject to constraint

$$\sum_{n=1}^{n} (x_i . wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n} (x_i . wi) = W$$

Fractional Knapsack

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n do
    x[i] = 0
weight = 0
for i = 1 to n
    if weight + w[i] ≤ W then
            x[i] = 1
            weight = weight + w[i]
    else
            x[i] = (W - weight) / w[i]
            weight = W
            break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of p_i/w_i,

then the loop takes a time in ***O(n)***;

Therefore, the total time including the sort is in ***O(n logn)***.

Fractional Knapsack

Consider that the capacity of the knapsack $W = 60$

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
P_i/w_i	7	10	6	5

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
P_i/w_i	10	7	6	5

Fractional Knapsack

After sorting all the items according to p_i/w_i. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

Fractional Knapsack

Consider that the capacity of the knapsack $W = 60$

Item	A	B	C	D	E
Profit	30	40	45	77	90
Weight	5	10	15	22	25
P_i/w_i	6	4	3	3.5	3.6

Fractional Knapsack

Consider that the capacity of the knapsack $W = 60$

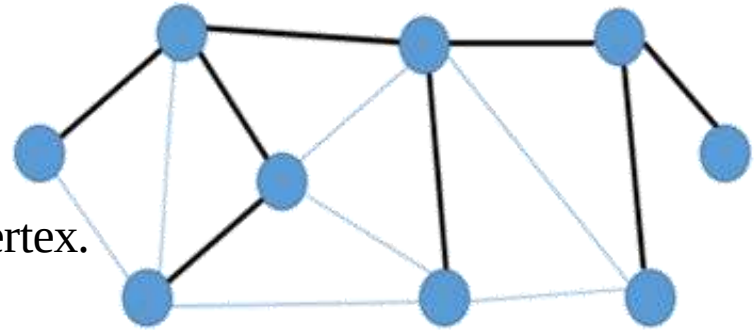
Item	A	B	C	D	E
Profit	30	40	45	77	90
Weight	5	10	15	22	25
P_i/w_i	6	4	3	3.5	3.6

Item	A	B	E	D	C
Profit	30	40	90	77	45
Weight	5	10	25	22	15
P_i/w_i	6	4	3.6	3.5	3

Spanning Tree

A **Spanning Tree** is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.

If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.



Properties:

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight.

Minimum Spanning Tree make use of **Prim's** algorithm or **Kruskal's** algorithm.

A **graph** may have more than one spanning tree. If there are **n** number of vertices, the spanning tree should have ***n*-1** number of edges.

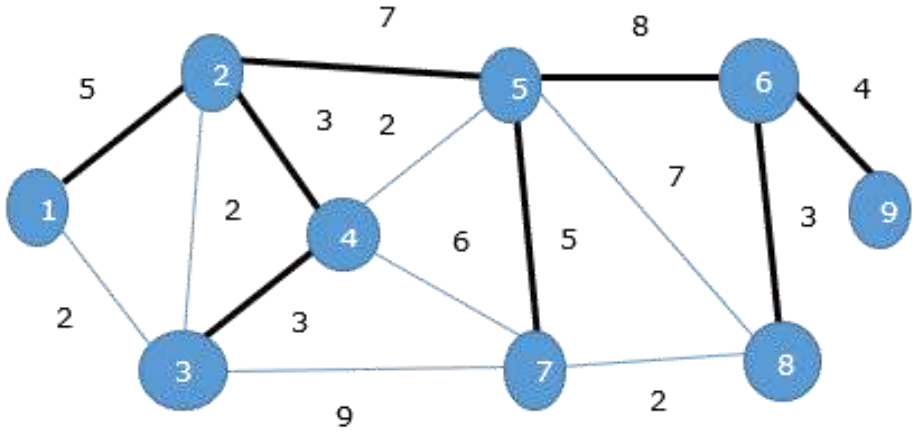
Each edge of the graph is associated with a **weight** and there exists more than one spanning tree, we need to find the **minimum** spanning tree of the graph.

Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.

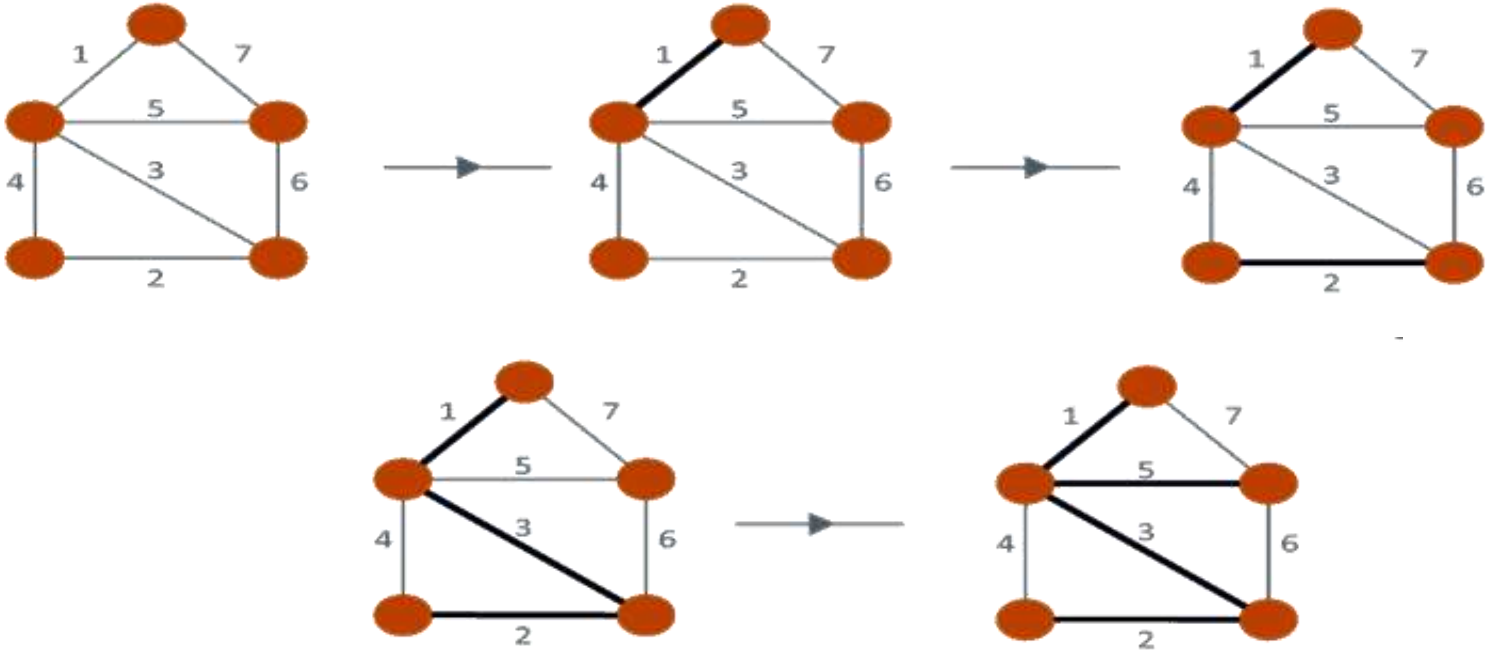
Minimum Spanning Tree

In the graph, thick edges shown a spanning tree though it's not the minimum spanning tree.

The cost of this spanning tree is (5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38.



Minimum Spanning Tree



Kruskal's algorithm

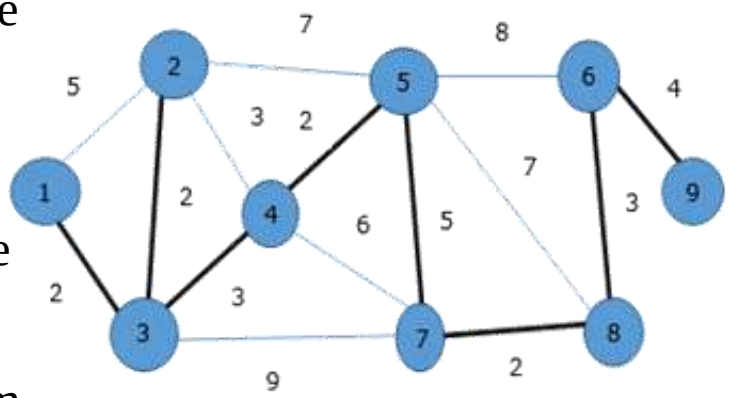
Minimum Spanning Tree

Prim's algorithm to find the minimum spanning tree

Randomly start from any vertex, let us start from vertex **1**.

Vertex **3** is connected to vertex **1** with minimum edge cost, hence edge **(1, 2)** is added to the spanning tree.

Next, edge **(2, 3)** is considered as this is the minimum among edges **{(1, 2), (2, 3), (3, 4), (3, 7)}**.



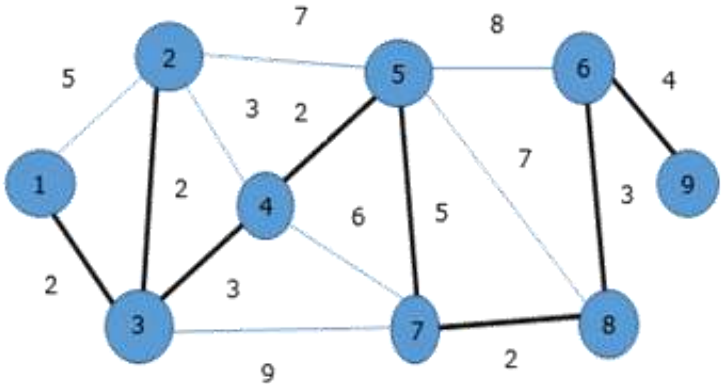
Minimum Spanning Tree

In the next step, we get edge **(3,4)** and **(2,4)** with minimum cost. Edge **(3,4)** is selected at random.

In a similar way, edges **(4,5), (5,7), (7,8), (6,8)** and **(6,9)** are selected. As all the vertices are visited, now the algorithm stops.

The cost of the spanning tree is
(2 + 2 + 3 + 2 + 5 + 2 + 3 + 4) = 23.

There is no more spanning tree in this graph with cost less than **23**.

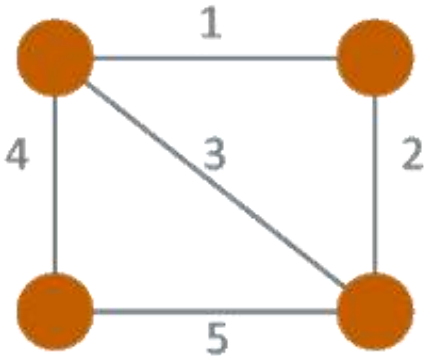


Time Complexity:

Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be **O(ElogV)**, which is the overall Time Complexity of the algorithm.

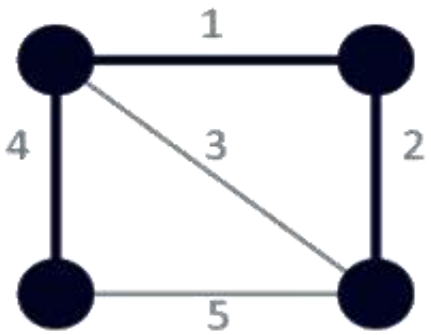
Prim's Algorithm, the time complexity will be **O((V+E)logV)** because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

Minimum Spanning Tree



Undirected Graph

Find the Minimum Spanning Tree ?



Minimum Spanning Tree
Cost = 7(=4+1+2)

Single Source Shortest Path Problem

Definition

The shortest path problem can be a path requires that consecutive vertices be connected by an appropriate directed edge.

It defined for graphs whether undirected, directed, or mixed.

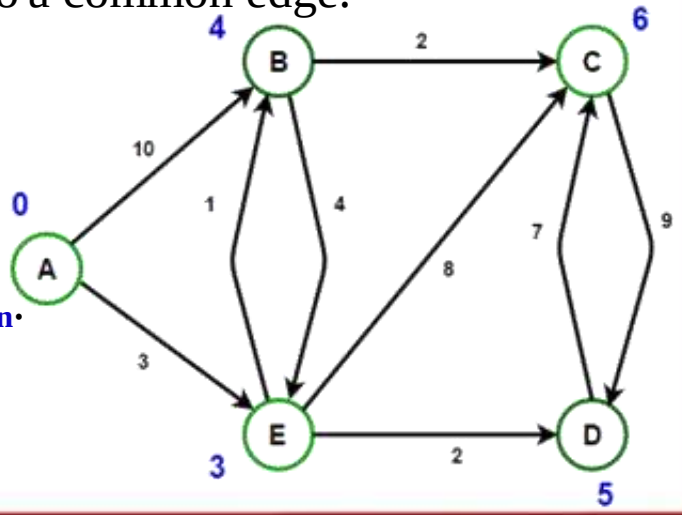
Two vertices are adjacent when they are both incident to a common edge.

A path in an undirected graph is a sequence of vertices

$$P=(v_1,v_2,\ldots,v_n) \in V*V*\ldots\ldots*V$$

Such that v_i is adjacent to v_{i+1} for $1 \leq i < n$.

Such a path **P** is called a path of length **n-1** from v_i to v_n.

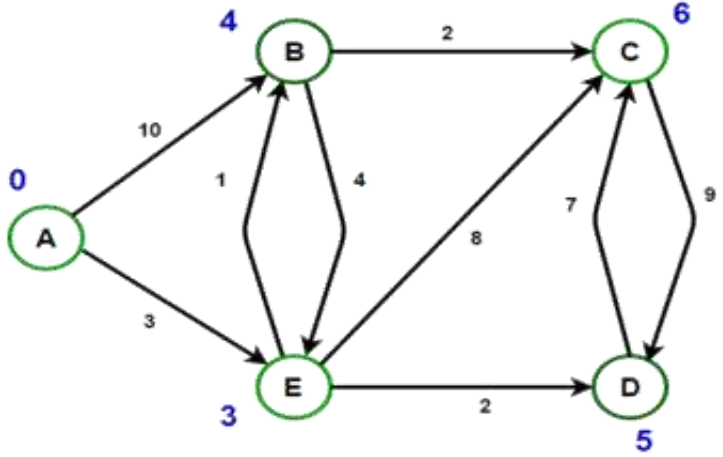


Single Source Shortest Path Problem

Relaxation

if (d[u] + c(u,v) < d[v])

d[v]= d[u] + c(u,v)



Vertex	Minimum Cost	Route
A -> B	4	A -> E -> B
A -> C	6	A -> E -> B -> C
A -> D	5	A -> E -> D
A -> E	3	A -> E

Single Source Shortest Path Problem

GIET INSTITUTIONS
ANDHRA PRADESH, INDIA

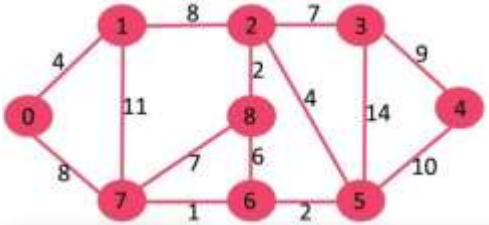
Dijkstra's Algorithm

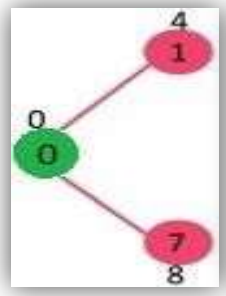
- An algorithm for finding the shortest paths between nodes in a graph.
- The algorithm finds the shortest path between that node and every other node.
- It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined

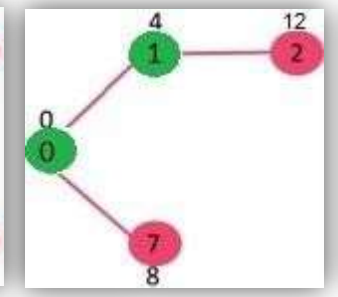
Dijkstra's Algorithm

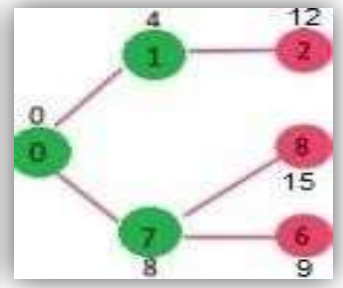
- Dijkstra Algorithm is based on the principle of Relaxation, in which an approximation for the correct distance is gradually replaced by more accurate values until shortest distance is reached.
- The approximate distance to each vertex is always an overestimate of the true distance, and it is replaced by the minimum of its old value with the length of newly found path.
- It uses a priority queue to greedily select the closest vertex that has not yet been processed and performs this relaxation process on all of its out going edges.

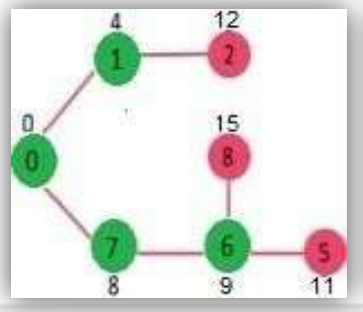
Single Source Shortest Path Problem using Dijkstra's Algorithm

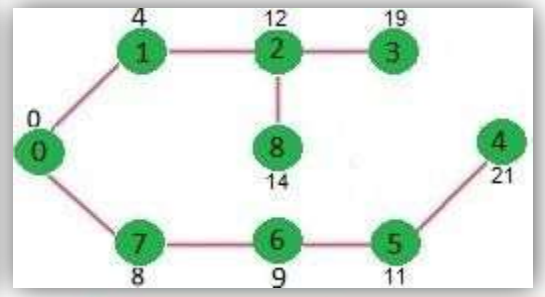










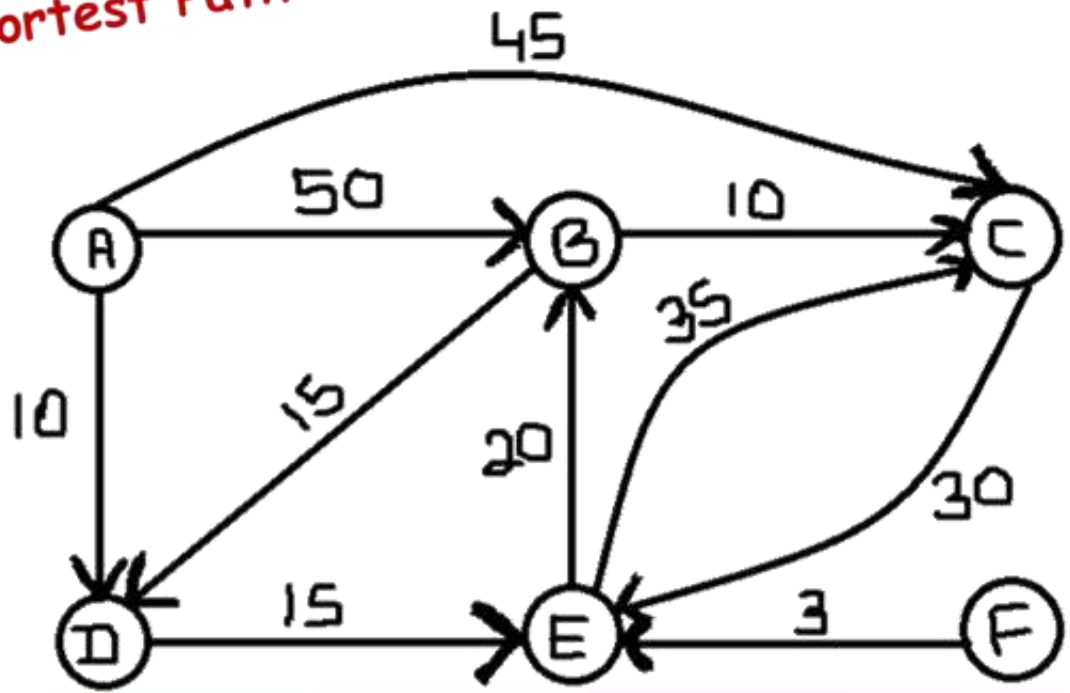


Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Single Source Shortest Path Problem

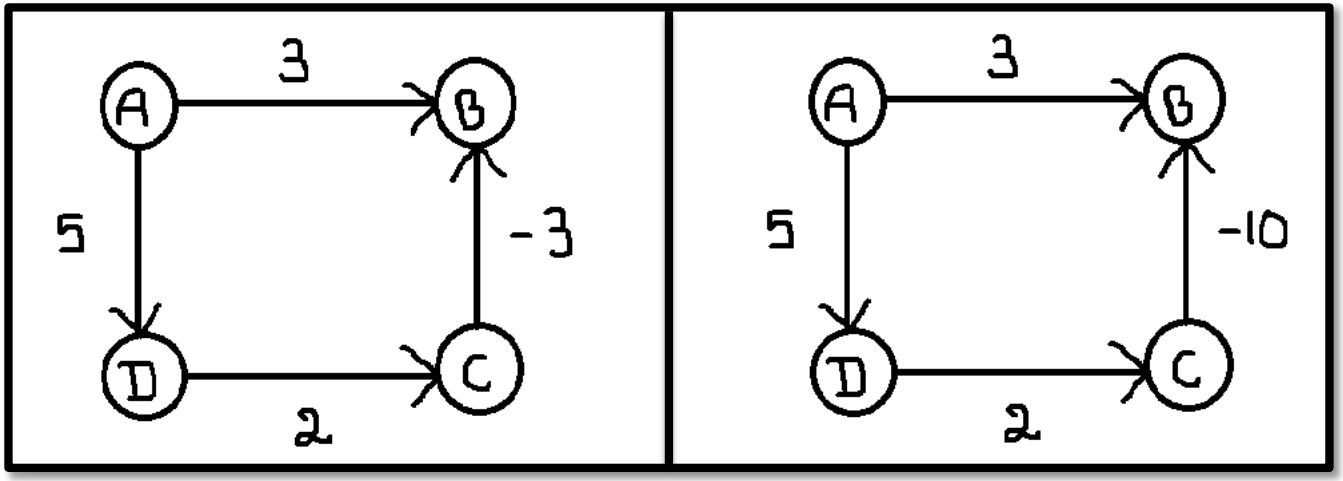
Find the

Single Source Shortest Path Problem ?



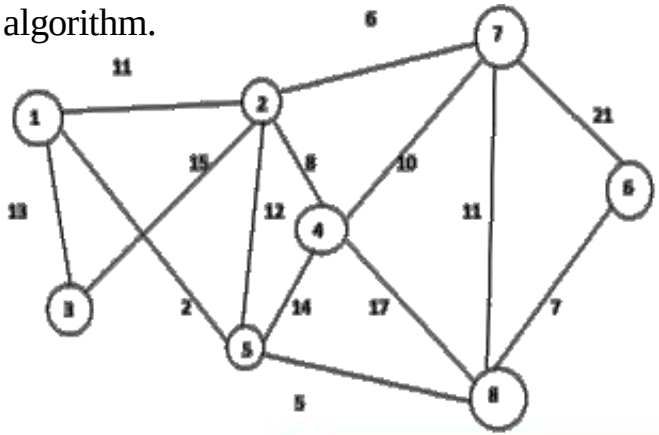
Find the

Single Source Shortest Path Problem ?



Design and Analysis of Algorithms
Assignment-II.1

1. Write a greedy algorithm to the Job sequencing with deadlines. Find the optimal solution using greedy Algorithm for given problem. Let **n=5**, **(p1, p2, p3, p4, p5)=(40,33,30,14,10)** and **(d1, d2, d3, d4, d5)=(2, 1, 2, 3, 3)**. Find the optimal solution using greedy Algorithm.
2. What is knapsack problem? State knapsack problem formally. Obtain the solution to knapsack problem where **n=6**, **(p1, p2, p3, p4, p5, p6) = (100, 50, 60, 20, 70, 30)**, **(w1, w2, w3, w4, w5, w6) = (20,10,15,5,25,10)** and **m=60**.
3. Define spanning tree. Compute a minimum cost spanning tree for the graph of figure using Kruskal's algorithm.

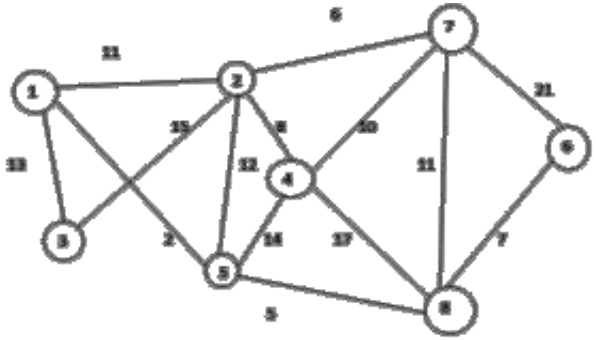


Note:

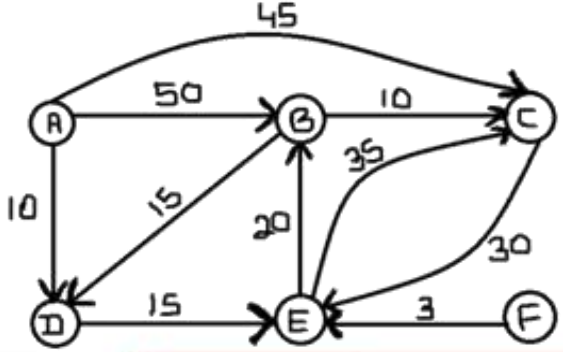
1. *Complete the Assignment-II.1 (**Hand Written**).*
2. ***Scan the work, convert into pdf and upload.***
3. *Assignment **Posted date 01-Oct-2020***
4. ***Last date of submission 03-Oct-2020.***

Design and Analysis of Algorithms
Assignment-II.2

1. Compute a minimum cost spanning tree for the graph of figure using Prims's algorithm.



2. What is Single Source Shortest Path Problem. Find the shortest path for the given directed graph



Note:

1. *Complete the Assignment-II.2 (**Hand Written**).*
2. ***Scan the work, convert into pdf and upload.***
3. *Assignment **Posted date 01-Oct-2020***
4. ***Last date of submission 05-Oct-2020.***

UNIT-I

DESIGN and ANALYSIS of ALGORITHMS

TOPIC-2: DIVIDE AND CONQUER

Presented By:

D. PHANI KUMAR

Assistant Professor

Department of CSE

GIET-[A]

Lecture Details:

Subject : Design and Analysis of Algorithms

Topic : DIVIDE AND CONQUER

Branch/Year : CSE, III-B.Tech I-Semester

UNIT-1	TOPIC	DESCRIPTION
Divide and Conquer	TOPIC-2	➢ **Divide and Conquer** ➢ **Applications of Divide and Conquer** ✓ **Binary Search** ✓ **Quick Sort** ✓ **Merge Sort**

Divide and Conquer:

This technique can be divided into the following three parts:

Divide: This involves dividing the problem into some sub problem.

Conquer: Sub problem by calling recursively until sub problem solved.

Combine: The Sub problem Solved so that we will get find problem solution.

Applications

Binary Search is a searching algorithm. In each step, the algorithm compares the input element **x** with the value of the **middle** element in array. If the values match, return the **index** of the middle. Otherwise, if **x** is **less than** the **middle** element, then the algorithm recurs for **left side** of **middle** element, else recurs for the **right side** of the **middle** element.

Quick Sort is a sorting algorithm. The algorithm picks a **pivot** element, rearranges the array elements in such a way that all elements smaller than the picked **pivot** element move to **left** side of **pivot**, and all greater elements move to **right** side. Finally, the algorithm recursively sorts the subarrays on **left** and **right** of **pivot** element.

Merge Sort is also a sorting algorithm. The algorithm **divides** the array in **two halves**, recursively sorts them and finally **merges** the two sorted halves.

Divide and Conquer Algorithm

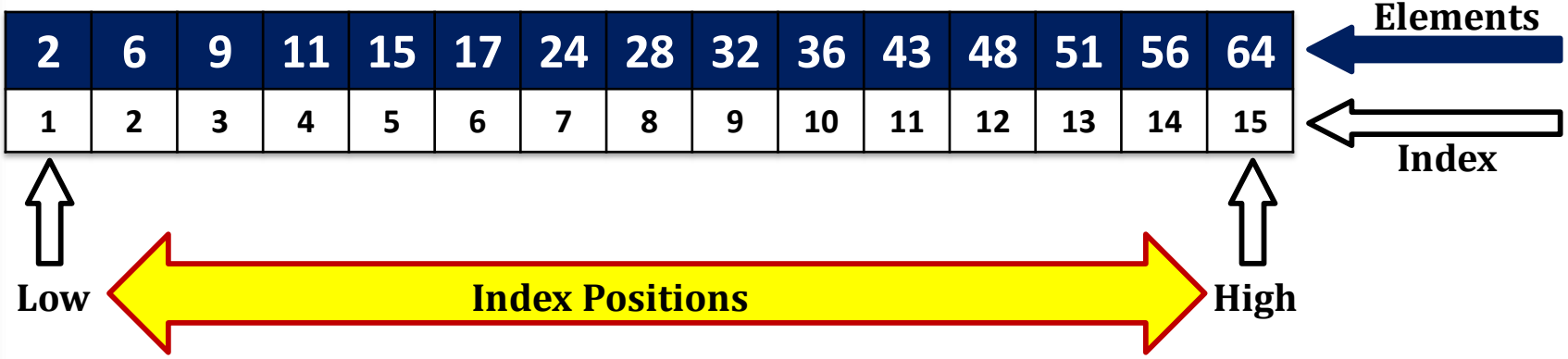
```
DAC(a, i, j)
{
    if(small(a, i, j))
        return(Solution(a, i, j))
    else m = divide(a, i, j)        // f1(n)
        b = DAC(a, i, mid)          // T(n/2)
        c = DAC(a, mid+1, j)        // T(n/2)
        d = combine(b, c)           // f2(n)
    return(d)
}
```

Recurrence Relation

O(1) if **n** is small

T(n) = f1(n) + 2T(n/2) + f2(n)

Binary Search



Binary Search

Key=43 (Searching Element)

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Low Mid High

Mid = (Low + High) / 2
= (1 + 15) / 2
= 8

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Binary Search

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Low Mid High

Here **Key** > **Mid**

- ✓ Right half partition is going to search
- ✓ **Low** = **Mid** + 1

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Low High

Binary Search

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Low Mid High

Again **Key** < **Mid**

- ✓ Right half partition is going to search
- ✓ **High** = **Mid** - 1

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Low High

Binary Search

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Low Mid High

Again **Key** > **Mid**

- ✓ Right half partition is going to search
- ✓ **Low** = **Mid** + 1

Mid

2	6	9	11	15	17	24	28	32	36	43	48	51	56	64
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Low High

Binary Search

```
low :=1 ; high :=n ;
while (low ≤ high) do
{
   mid :=|(low + high)/2|
   if (x < a [mid]) then
       high:=mid – 1;
   else if (x > a [mid]) then
       low:= mid + 1
   else
       return mid;
}
return 0;
```

Successful searches

O(1),	O(log n),	O(log n)
Best	average	worst

un-successful searches

O(log n)

best, average and worst

Merge Sort

- ✓ **Merge sort** algorithm is a classic example of **divide and conquer**.
- ✓ To sort an array, recursively, sort its left and right halves separately and then merge them.
- ✓ The time complexity of merge mort in the ***best case, worst case*** and ***average case*** is **O(n log n)** and the number of comparisons used is nearly optimal.

Merge Sort

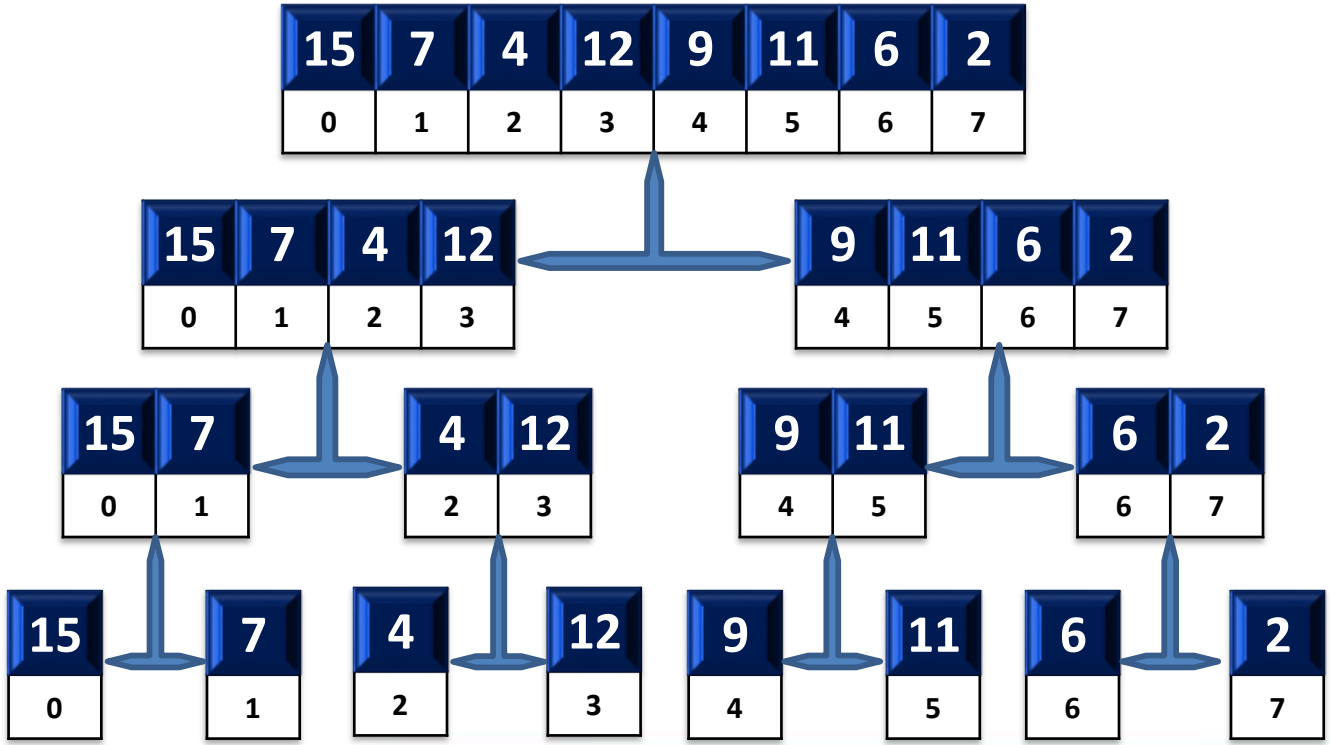
MergeSort (a[], l, r)

if **r > l**

1. Find the middle point to divide the array into two halves:
   middle m = (l + r) / 2
2. Call mergeSort for first half:
   Call mergeSort (a, l, m)
3. Call mergeSort for second half:
   Call mergeSort (a, m + 1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge (a, l, m, r)

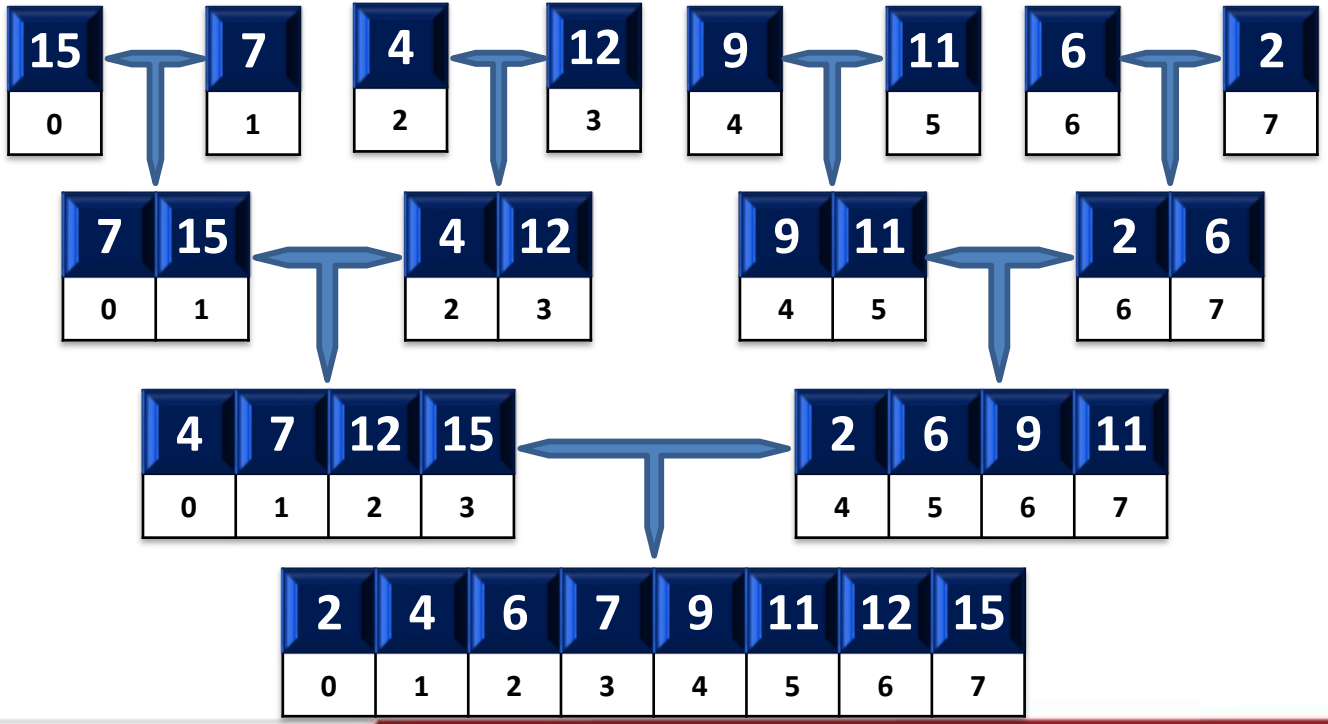
Merge Sort

✓ **Divide** the unsorted list into n sublists, each comprising 1 element.



Merge Sort

- ✓ Repeatedly **merge** sublists to produce newly sorted sublists until there is only 1 sublist remaining.

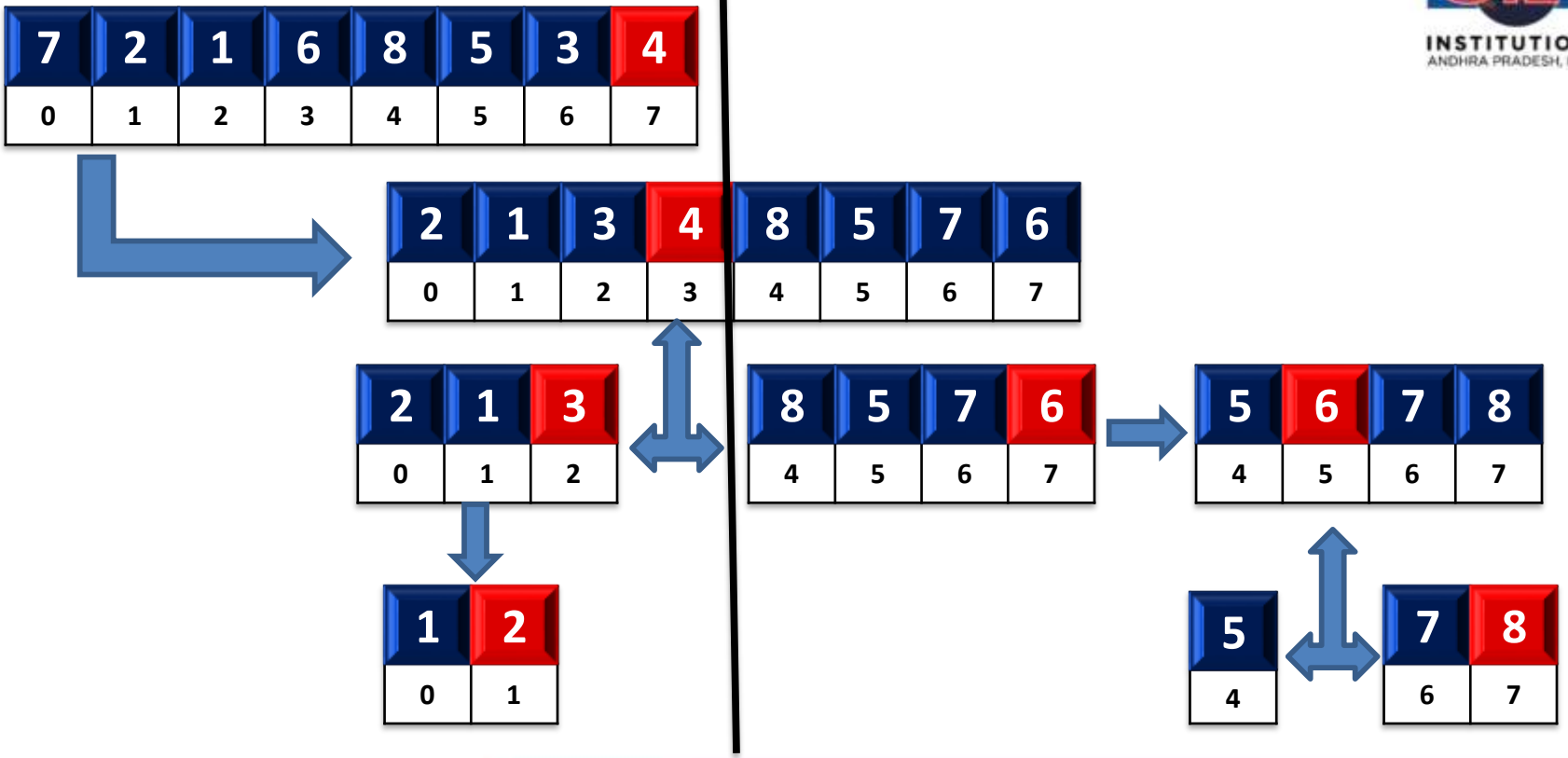


Quick Sort

Pick an element as **pivot** and partitions the given array around the picked pivot. There are different ways to pick pivot.

- ✓ Always pick first element as pivot.
- ✓ Always pick last element as pivot (implemented below)
- ✓ Pick a random element as pivot.
- ✓ Pick median as pivot.

Quick Sort



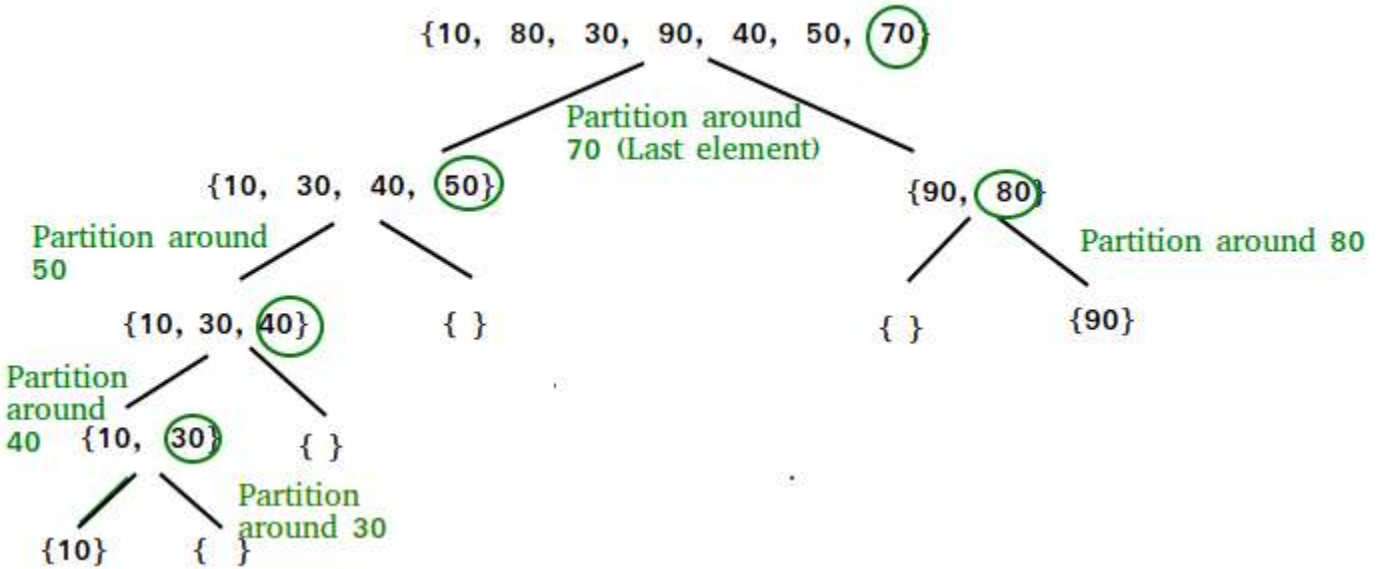
Quick Sort

```
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now at right place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Quick Sort

```
partition (arr[], low, high)
{
    pivot = arr[high];
    i = (low - 1)      // Index of smaller element
    for (j = low; j <= high- 1; j++)
    {
      if (arr[j] < pivot) // If current element is smaller than the pivot
      {
          i++; // increment index of smaller element
          swap (arr[i], arr[j])
      }
    }
    swap (arr[i + 1], arr[high])
    return (i + 1)
}
```

Quick Sort


{10, 80, 30, 90, 40, 50, 70}
Partition around 70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around 50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition around 40
{10, 30}
{ }
Partition around 30
{10}
{ }

Quick Sort

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0 arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j, as are same

j = 1 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing // No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Quick Sort

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.

Following is recurrence for worst case.

$$T(n) = T(0) + T(n-1) + O(n)$$

which is equivalent to $T(n) = T(n-1) + O(n)$

The solution of above recurrence is **$O(n^2)$**.

Quick Sort

Best Case: The best case occurs when the partition process always picks the middle element as pivot.
Following is recurrence for best case.

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

The solution of above recurrence is **O(nlog(n))**.

Average Case: Consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set.
Following is recurrence for this case.

$$T(n) = T(n/9) + T(9n/10) + (n)$$

Solution of above recurrence is also **O(nlog(n))**

Design and Analysis of Algorithms
Assignment-I.2

GIET INSTITUTIONS
ANDHRA PRADESH, INDIA

1. Give the general procedure of divide and conquer method. Briefly explain about quick sort algorithm with neat example and derive it's time complexity.
2. Explain in detail merge sort . Sort the following set of elements using merge sort 12,24,8,71,4,23,6,89,56
3. Write recursive binary search algorithm with an example and analyze time complexity.

Note:

1. *Complete the Assignment-I-2 (**Hand Written**).*
2. ***Scan** the work, convert into **pdf** and **upload.***
3. *Assignment **Posted date 09-Sept-2020***
4. ***Last date** of submission **12-Sept-2020**.*

GIET INSTITUTIONS
ANDHRA PRADESH, INDIA

UNIT-IV
DESIGN and ANALYSIS of ALGORITHMS

TOPIC: BACKTRACKING

Presented By:
D. PHANI KUMAR
Assistant Professor
Department of CSE
GIET-[A]

Lecture Details:
Subject : Design and Analysis of Algorithms
Topic : BACKTRACKING
Branch/Year : CSE, II-B.Tech II-Semester

UNIT-4	TOPIC	DESCRIPTION
BACKTRACKING	➢ General Method ➢ Applications	➢ Backtracking ✓ General Method ➢ Applications ✓ n-Queen problem ✓ Sum of Subsets problem ✓ Graph Colouring ✓ Hamiltonian Cycles

Outcome

Student able to:

- ✓ Categorize the Backtracking with various Algorithms
- ✓ Identify and solve the major graph algorithms problems with their analyses
- ✓ Relate various algorithm design situations

Backtracking

Backtracking is a technique based on algorithm to solve problem. It uses **recursive** calling to find the solution by building a solution step by step **increasing values** with **time**. It removes the solutions that doesn't give rise to the solution of the problem based on the **constraints** given to solve the problem.

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time

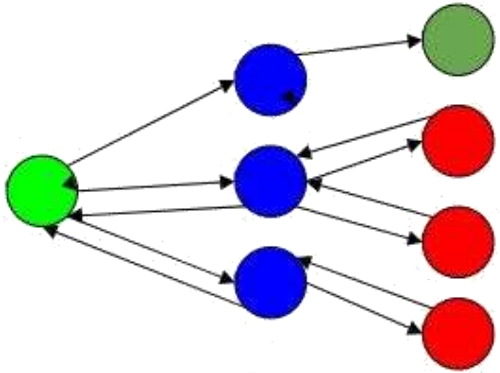
Backtracking

Backtracking algorithm is applied to some specific types of problems:

- ✓ **Decision problem** used to find a **feasible solution** of the problem.
- ✓ **Optimization problem** used to find the **best solution** that can be applied.
- ✓ **Enumeration problem** used to find the **set of all feasible solutions** of the problem.

Backtracking

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints, and the problem can backtrack if no feasible solution is found for the problem.

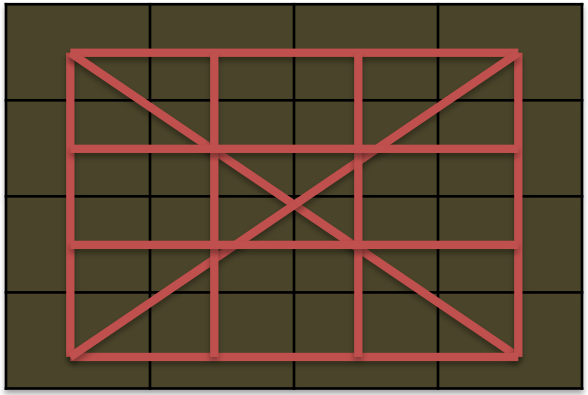


Algorithm:

Step 1 : if **current_position** is goal, return success
Step 2 : else,
Step 3 : if **current_position** is an end point, return failed.
Step 4 : else, if **current_position** is not end point, explore and repeat above steps.

n-Queen Problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.



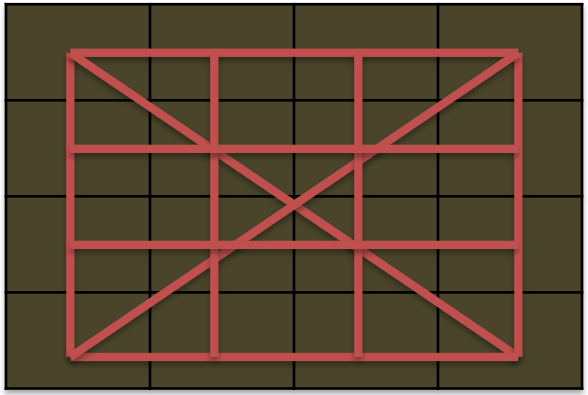
n-Queen Problem

1) Start in the leftmost column
2) If all queens are placed
   return true
3) Try all rows in the current column.
   Do following for every tried row.
   a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   b) If placing queen in [row, column] leads to a solution then return true.
   c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4) If all rows have been tried and nothing worked, return false to trigger backtracking.

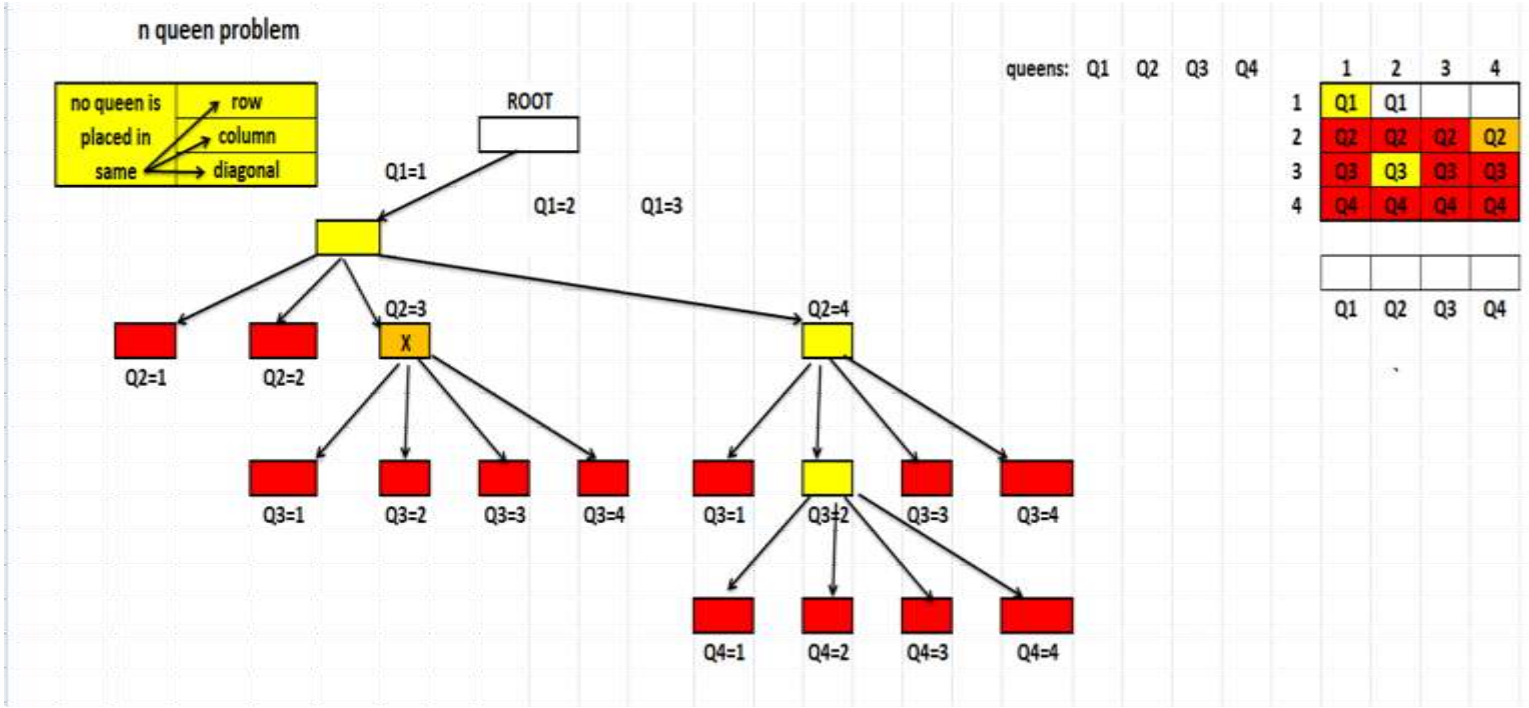
n-Queen Problem

Bounding Function:

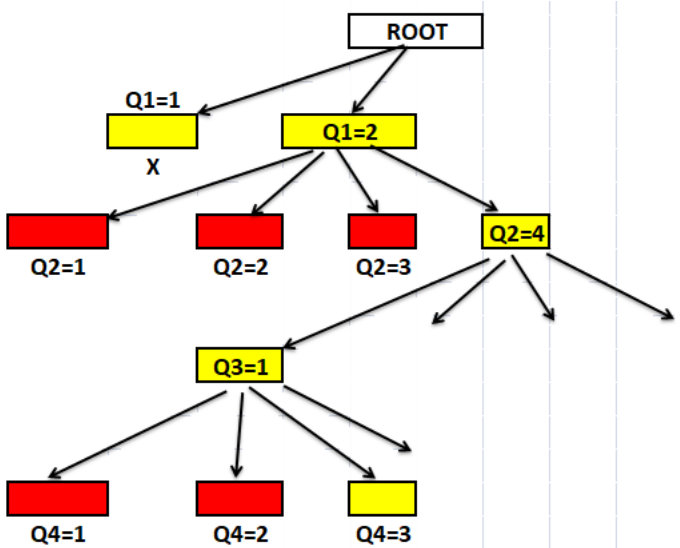
No Queen is placed in same Row, Column and Diagonal



n-Queen Problem



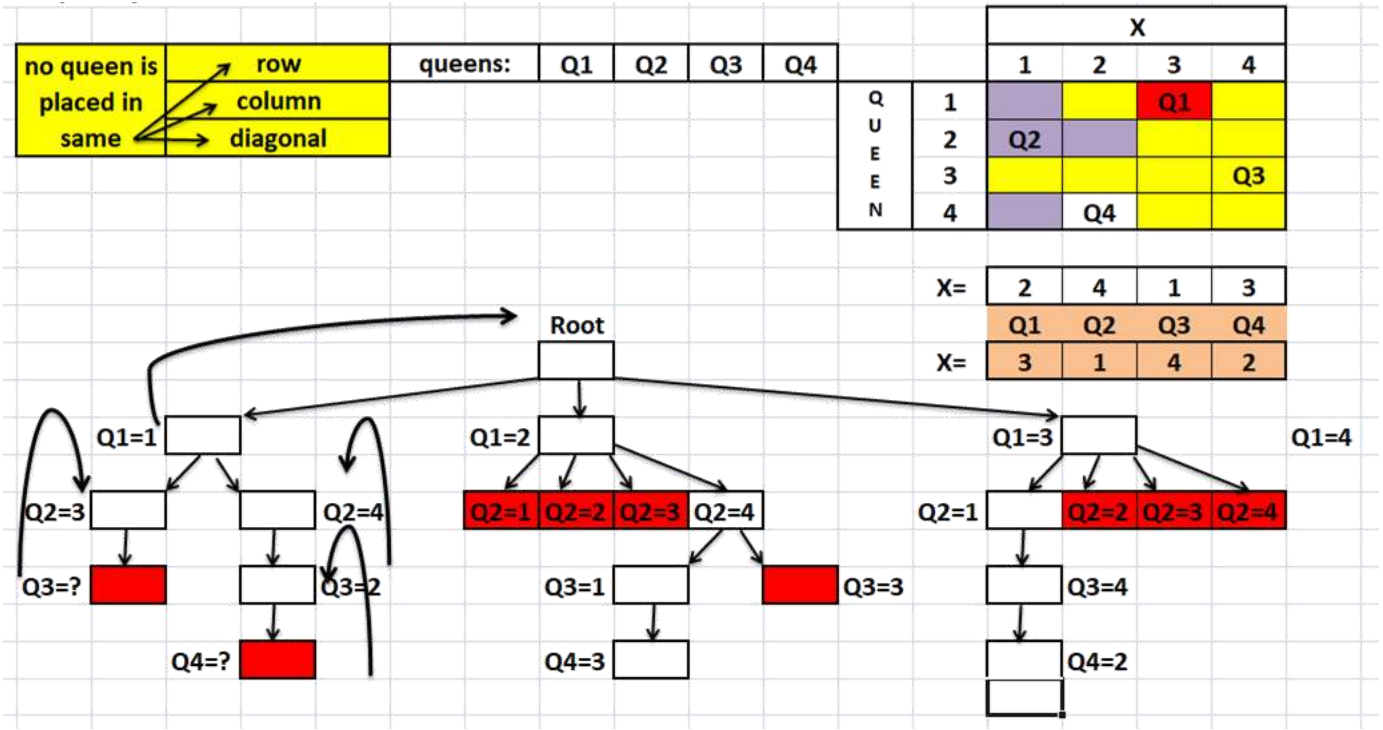
n-Queen Problem



	1	2	3	4
1		Q1		
2	Q2	Q2	Q2	Q2
3	Q3			
4	Q4	Q4	Q4	

Q1	Q2	Q3	Q4
2	4	1	3
3	1	4	2

n-Queen Problem



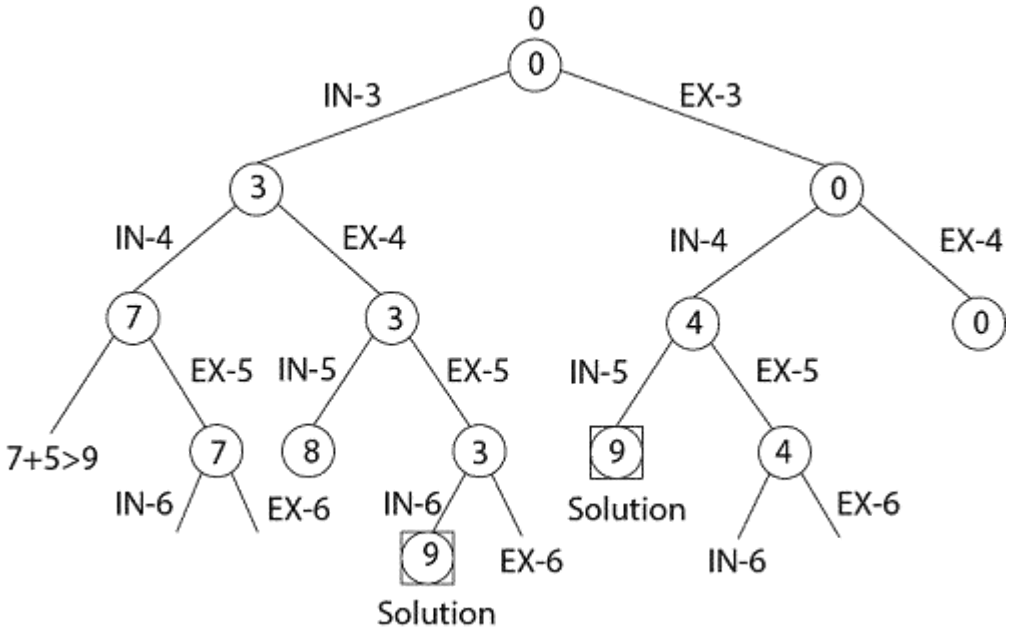
sum of subset problem

Subset sum problem is to find subset of elements that are **selected** from a given set whose **sum** adds **up to a given number K**. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

Input: This algorithm takes a set of numbers, and a sum value.
The Set: {10, 7, 5, 18, 12, 20, 15}
The sum Value: 35

Output: All possible subsets of the given set, where sum of each element for every subsets is same as the given sum value.
{10, 7, 18} {10, 5, 20} {5, 18, 12} {20, 15}

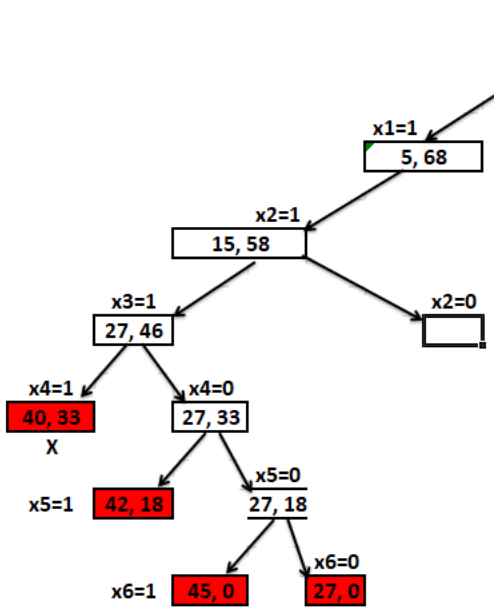
sum of subset problem



sum of subset problem

	1	2	3	4	5	6		
w={	5	10	12	13	15	18	}	73

n=6 k=30



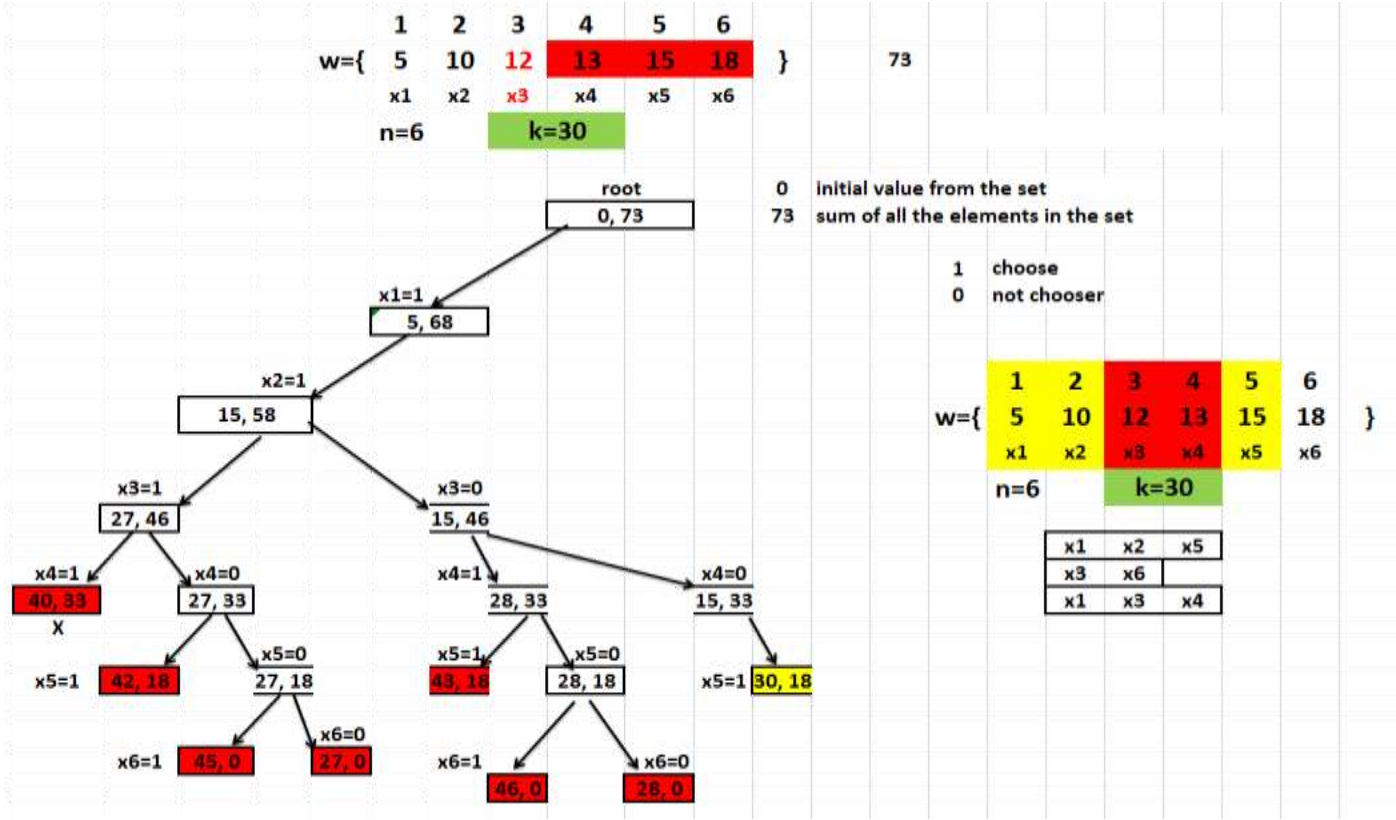
0 initial value from the set

73 sum of all the elements in the set

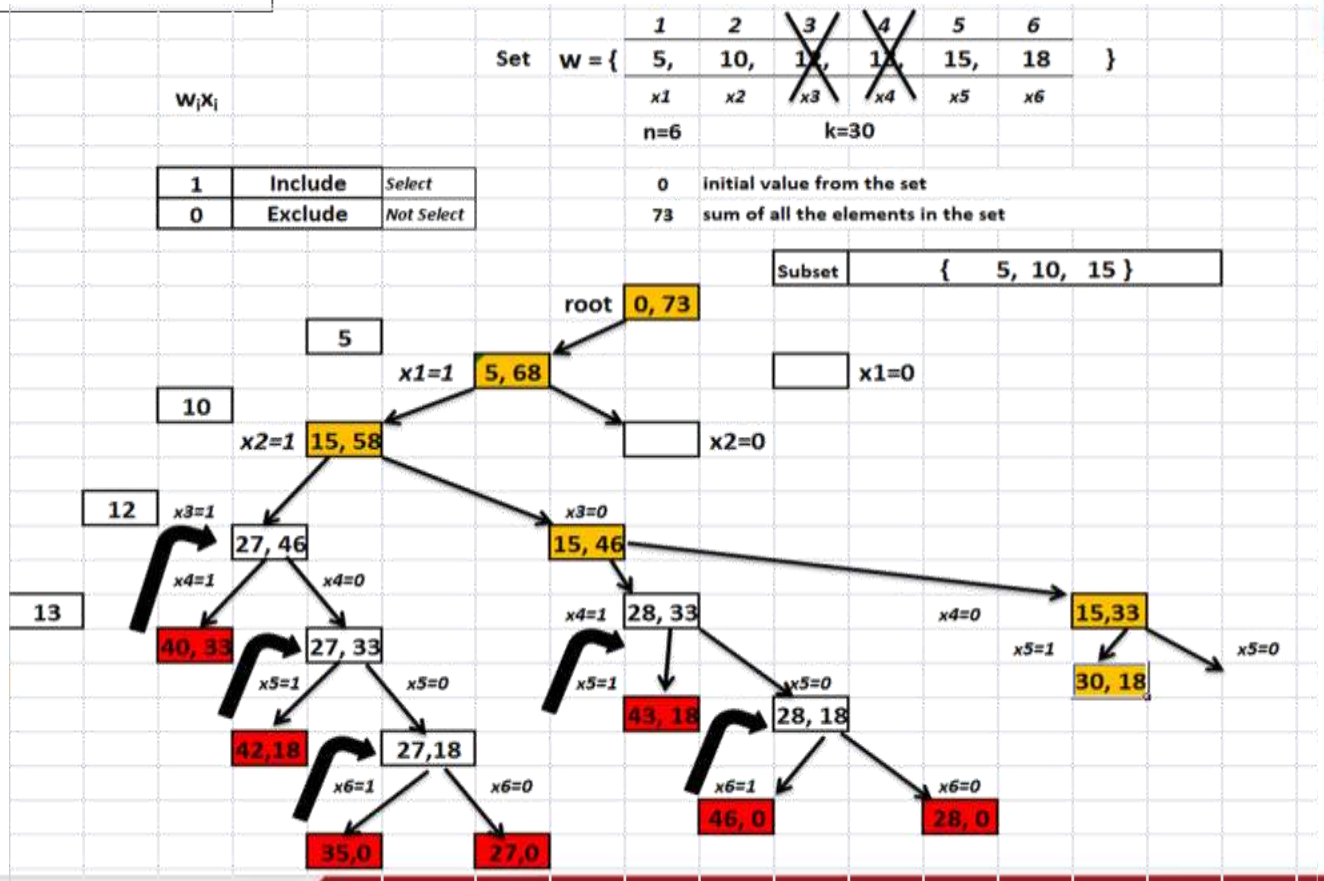
1 choose

0 not chooser

sum of subset problem

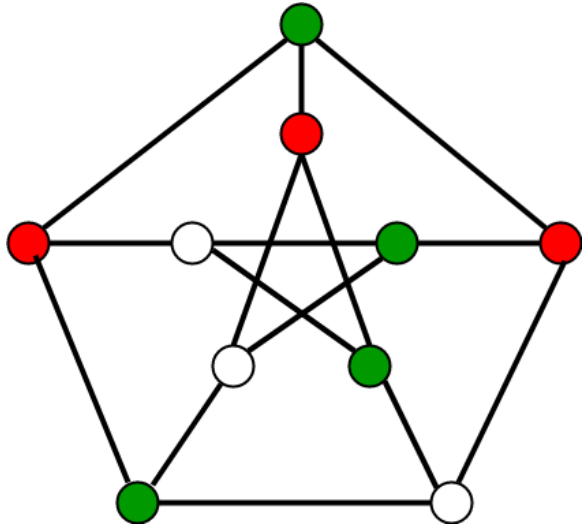


sum of subset problem



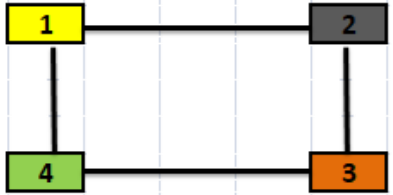
Graph Colouring

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.



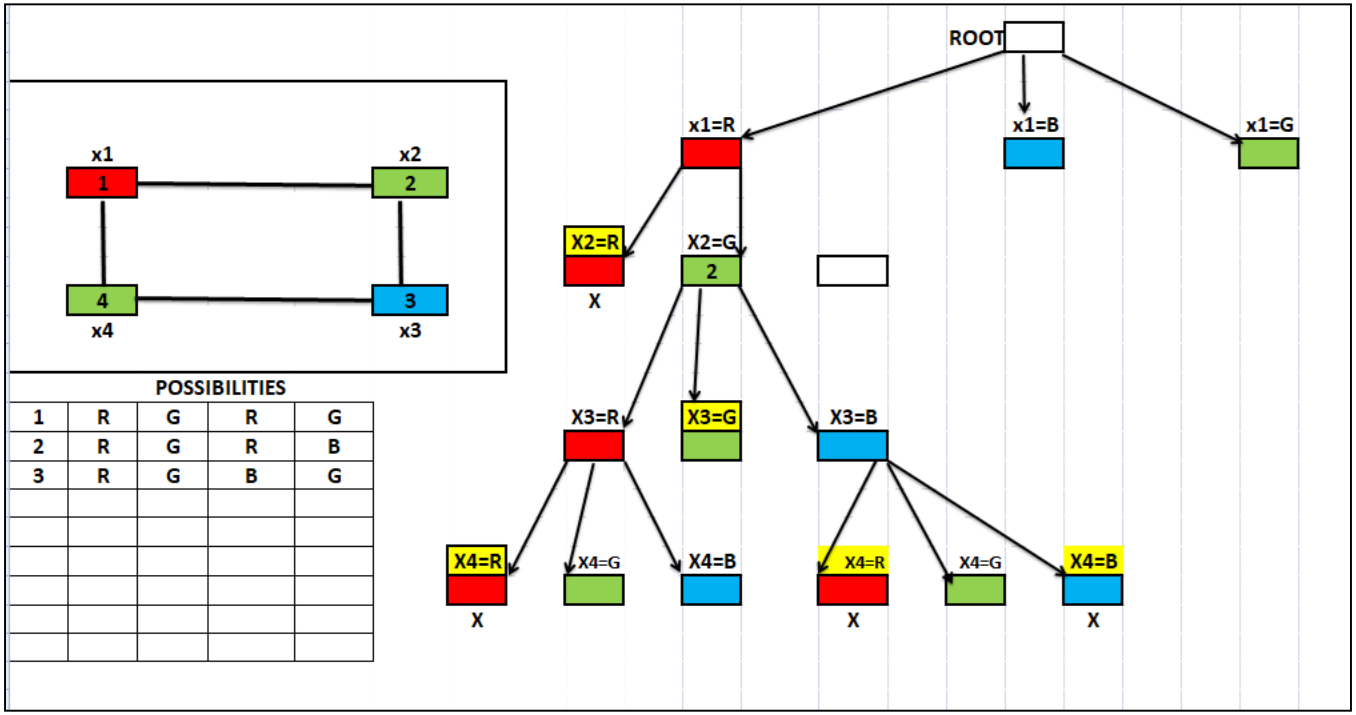
Graph Colouring

mcoloring Decision Problem mcoloring Optimization Problem	Chromatic Number



mcoloring Decision Problem	mcoloring Optimization Problem
Whether given colors are enough or not	How many colors

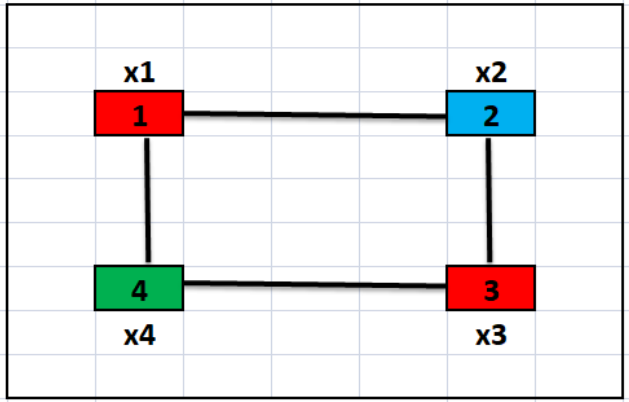
Graph Colouring



POSSIBILITIES

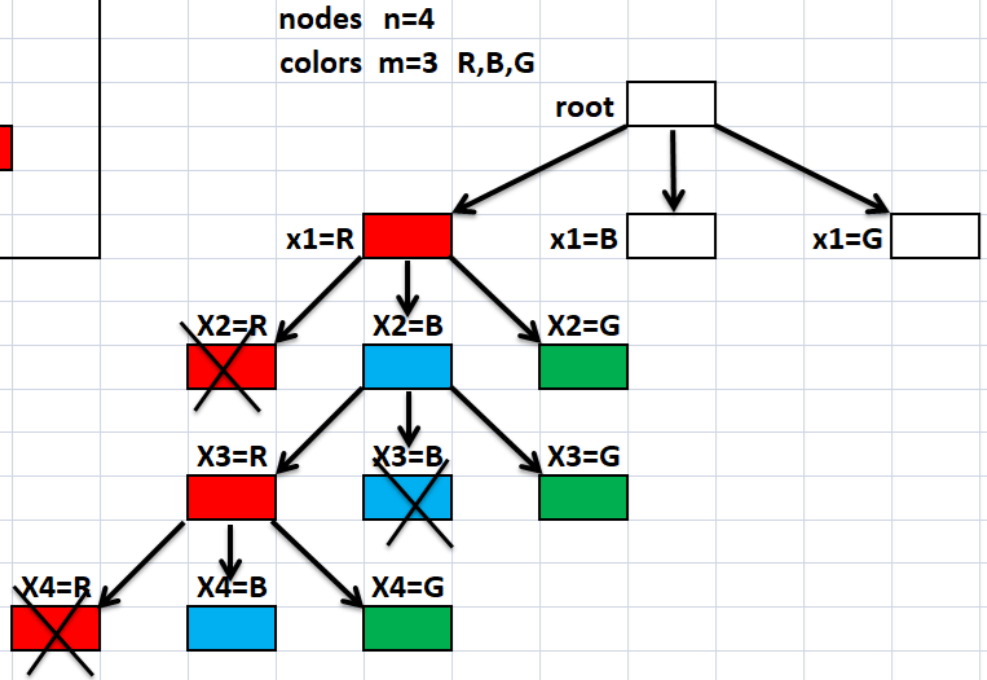
1	R	G	R	G
2	R	G	R	B
3	R	G	B	G

Graph Colouring



POSSIBILITIES

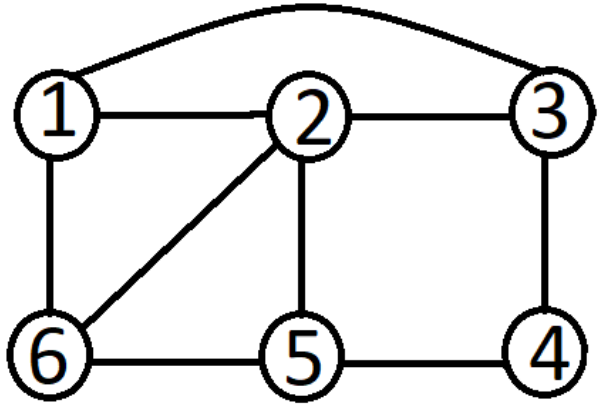
	X1	X2	X3	X4
1	R	B	R	B
2	R	B	G	B
3	R	G	R	G
	R	G	B	G
	G	B	G	B
	G	R	G	R
	G	B	G	R
	G	R	G	B



Hamiltonian Cycle

- ✓ Hamiltonian Path is an undirected graph where each vertex visits exactly once.
- ✓ A Hamiltonian cycle is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex.

Determine whether a given graph contains Hamiltonian Cycle or not ?



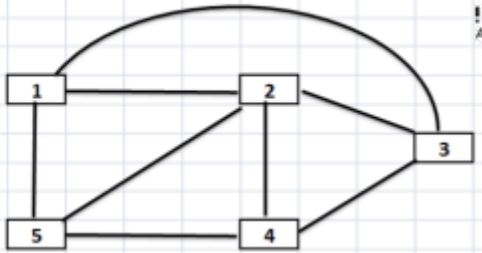
Hamiltonian Cycle

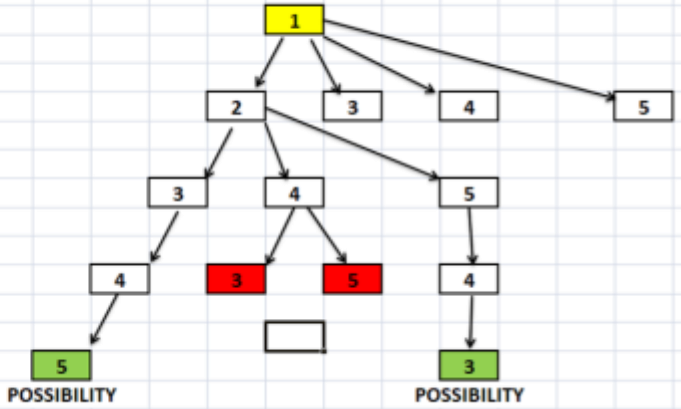
Hamiltonian Cycle

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

X	1	2	3	4	5	1
	1	2	5	4	3	1

1. NO DUPLICATES VERTICES
2. PATH MUST EXIST BETWEEN CURRENT AND LAST VERTEX
3. PATH MUST EXIST BETWEEN LAST AND INITIAL VERTEX





Assignment-4

- Explain the following Algorithms with examples:
  1. Sum of Subsets
  2. n Queen Problem
  3. Graph Coloring
  4. Hamiltonian Cycle

UNIT-V

DESIGN and ANALYSIS of ALGORITHMS

TOPIC: BRANCH AND BOUND

Presented By:

D. PHANI KUMAR

Assistant Professor

Department of CSE

GIET-[A]

Lecture Details:

Subject : Design and Analysis of Algorithms

Topic : BRANCH AND BOUND

Branch/Year : CSE, II-B.Tech II-Semester

UNIT-5	TOPIC	DESCRIPTION
BRANCH AND BOUND	➢ General Method ➢ Applications	➢ Branch and Bound ✓ General Method ➢ Applications ✓ Travelling sales person problem ✓ 0/1 knapsack problem

Outcome

Student able to:

- ✓ Categorize the Branch and Bound with various Algorithms
- ✓ Identify and solve the major graph algorithms problems with their analyses
- ✓ Relate various algorithm design situations

Branch and Bound

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.

The principle consists in partitioning the solution space into disjoint subsets, which are represented by the nodes of the branching tree. Then, the algorithm explores the branches of the tree according to a route strategy. To avoid exploring the entire tree, before creating a new node in the tree, the algorithm evaluates the node by comparing the value of the best possible solution, which could be found in the corresponding subtree, with the best current solution. If a better solution cannot belong to the subtree rooted at the considered node, the subtree is discarded.

Branch and Bound

- ✓ Branch and Bound is a Systematic method for solving the optimization problem.
- ✓ Branch and Bound technique applied when Greedy Method and Dynamic Programming method may fails.
- ✓ Branch and Bound is much slower. Often leads to exponential time complexities in the worst case.
- ✓ Branch and Bound can lead to run reasonably fast on an average cases when applied carefully.

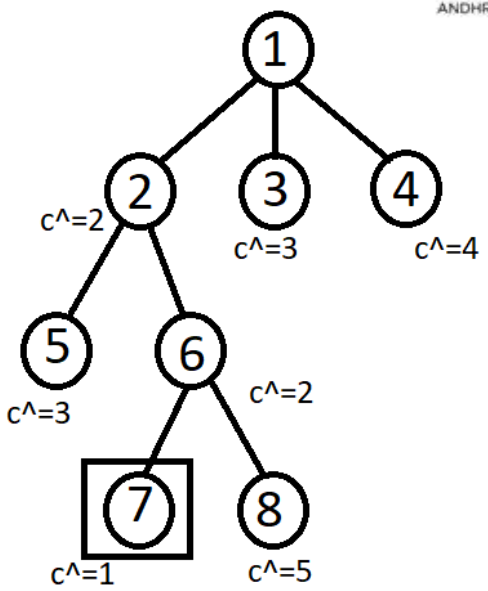
- ✓ Branch and Bound refers to all the state space search method in which all the children of an e-node are generated before any other live node can become the e-node.

Branch and Bound

E-node - Expanded node or E node is the node which is been expanded. As we know a tree can be expanded using both BFS(Breadth First Search) and DFS(Depth First Search),all the expanded nodes are known as **E-nodes**.

Live-node - A node which has been generated and all of whose children are not yet been expanded is called a live-node.

Dead-node - If a node can't be expanded further, it's known as a dead-node.



Branch and Bound

The word, Branch and Bound refers to all the state space search methods in which we generate the children of all the expanded nodes, before making any live node as an expanded one. In this method, we find the most promising node and expand it. The term **promising node** means, choosing a node that can expand and give us an optimal solution. We start from the root and expand the tree until unless we approach an optimal (minimum cost in case of this problem) solution

Branch and Bound

- Searching Techniques used in Branch and Bound
  - ✓ BFS
  - ✓ DFS
  - ✓ Least Count Search
    - Lower Bound
    - Upper Bound

How to get the cost for each node in the state space tree ?

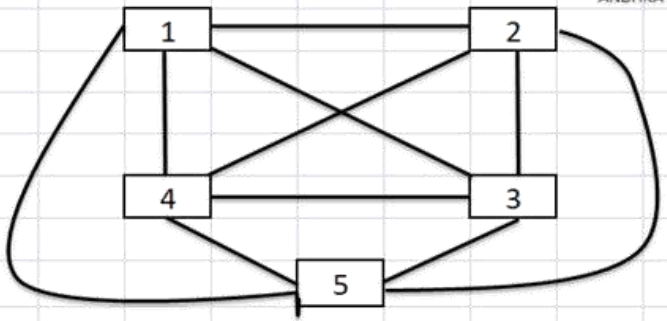
To get further in branch and bound, we need to find the cost at the nodes at first. The cost is found by using **cost matrix reduction**, in accordance with two accompanying steps **row reduction** & **column reduction**.

In general to get the optimal(lower bound in this problem) cost starting from the node, we reduce each row and column in such a way that there must be atleast one **0** in each row and column. For doing this, we just need to reduce the minimum value from each row and column.

- **Applications**
  - ✓ Travelling sales person problem
  - ✓ 0/1 knapsack problem

Travelling sales person problem

	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞



c(i,j)+r+r^

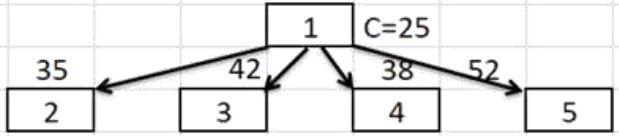
Reduction Matrix

	1	2	3	4	5	
1	∞	10	17	0	1	10
2	12	∞	11	2	0	2
3	0	3	∞	0	2	2
4	15	3	12	∞	0	3
5	11	0	0	12	∞	4
	1	0	3	0	0	

21 Row wise-sum of reduced values

4 Column wise-sum of reduced values

r= 25



Travelling sales person problem

V[1,2]

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	12	∞	11	2	0	0
3	0	∞	∞	0	2	0
4	15	∞	12	∞	0	0
5	11	∞	0	12	∞	0
	0		0	0	0	

r^= 0

c(i,j)+r+r^

10+25+0=35

V[1,3]

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	12	∞	∞	2	0	0
3	0	3	∞	0	2	0
4	15	3	∞	∞	0	0
5	11	0	∞	12	∞	0
	0	0		0	0	

r^=0

c(i,j)+r+r^

17+25+0=42

V[1,4]

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

V[1,5]

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

V[2,3]

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

V[2,4]

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

V[2,5]

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

0/1 Knapsack Problem

Profit	10	10	12	18
Weight	2	4	6	9

m=15

n=4

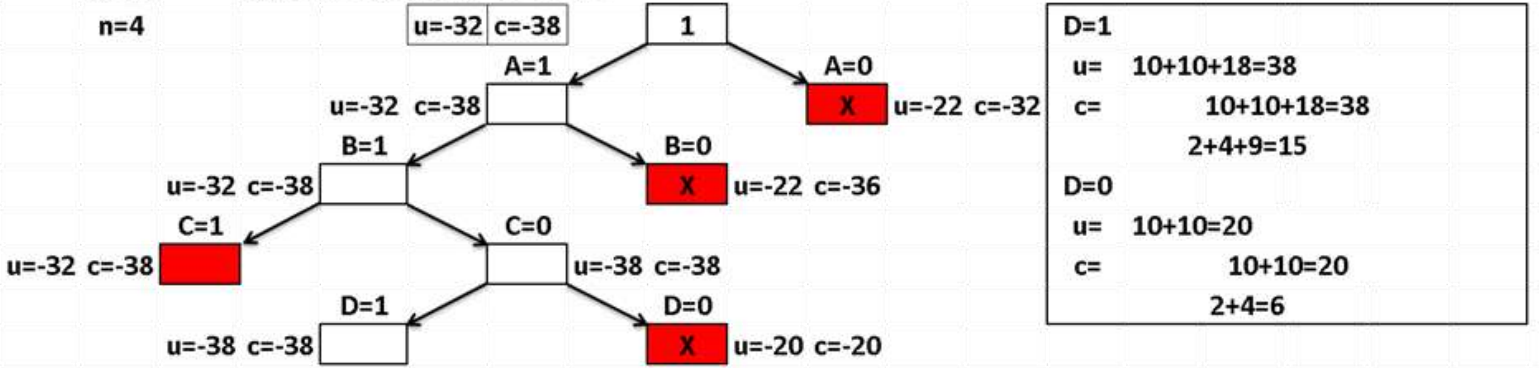
0/1 Knapsack Problem

0/1 Knapsack Problem

A	B	C	D	ITEMS
10	10	12	18	PROFIT
2	4	6	9	WEIGHTS

M=15
n=4

Least Count upper bound= ~~00~~ ~~-32~~ -38



Items= { A, B, C, D}
X= { 1, 1, 0, 1}

NP-Completeness

Polynomial Time

Linear Search	: n
Binary Search	: log*n*
Insertion Sort	: n^2
Merge Sort	: *n*log*n*
Matrix Chain Multiplication	: n^3

Exponential Time

0/1 Knapsack	: 2^n
Travelling Salesman	: 2^n
Sum of Subsets	: 2^n
Graph Colouring	: 2^n
Hamiltonian Cycle	: 2^n

NP-Completeness

NP-Completeness: Complexity Classes P, NP, NP-hard and NP-complete, Clique decision problem, Node cover decision problem.

A problem is in the class NPC if it is in NP and is as hard as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.

If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language B is NP-complete if it satisfies two conditions

- ✓ B is in NP
- ✓ Every A in NP is polynomial time reducible to B.

If a language satisfies the second property, but not necessarily the first one, the language B is known as NP-Hard. Informally, a search problem B is NP-Hard if there exists some NP-Complete problem A that Turing reduces to B.

The problem in NP-Hard cannot be solved in polynomial time, until P=NP. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

Clique Problem

Definition: In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.

Clique Cover: Given a graph G and an integer k, can we find k subsets of verticesV1, V2...VK, such that
UiVi = V, and that each Vi is a clique of G.