DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE MATERIAL

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
(2022-2023)

NAME OF THE FACULTY: Dr. T. Srikanth & Dr.Y.Madhusekhar

YEAR: III-I

ACADEMIC YEAR 2022-2023

REGULATION R18

CS603PC: DESIGN AND ANALYSIS OF ALGORITHMS

III Year B.Tech. CSE II-Sem **L T P C**
3 1 0 4

Prerequisites:
1. A course on "Computer Programming and Data Structures"
2. A course on "Advanced Data Structures"

Course Objectives:
- Introduces the notations for analysis of the performance of algorithms.
- Introduces the data structure disjoint sets.
- Describes major algorithmic techniques (divide-and-conquer, backtracking, dynamic
- programming, greedy, branch and bound methods) and mention problems for which eachtechnique is appropriate;
- Describes how to evaluate and compare different algorithms using worst-, average-, and best-caseanalysis.
- Explains the difference between tractable and intractable problems, and introduces the problems thatare P, NP and NP complete.

Course Outcomes:

- Ability to analyze the performance of algorithms
- Ability to choose appropriate data structures and algorithm design methods for a specified application
- Ability to understand how the choice of data structures and the algorithm design methods impact theperformance of programs.

UNIT - I
Introduction: Algorithm, Performance Analysis-Space complexity, Time complexity, AsymptoticNotations- Big oh notation, Omega notation, Theta notation and Little oh notation.
Divide and conquer: General method, applications-Binary search, Quick sort, Merge sort, Strassen's matrix multiplication.
UNIT - II
Disjoint Sets: Disjoint set operations, union and find algorithms
Backtracking: General method, applications, n-queen's problem, sum of subsets problem, graphcoloring
UNIT - III
Dynamic Programming: General method, applications- Optimal binary search trees, 0/1 knapsackproblem, All pairs shortest path problem, Traveling sales person problem, Reliability design.
UNIT - IV
Greedy method: General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.
UNIT - V
Branch and Bound: General method, applications - Travelling sales person problem, 0/1 knapsackproblem - LC Branch and Bound solution, FIFO Branch and Bound solution.
NP-Hard and NP-Complete problems: Basic concepts, non-deterministic algorithms, NP - Hard andNP-Complete classes, Cook's theorem.

TEXT BOOK:
1. Fundamentals of Computer Algorithms, Ellis Horowitz, Satraj Sahni and Rajasekharan, University Press

REFERENCE BOOKS:
1. Design and Analysis of algorithms, Aho, Ullman and Hopcroft, Pearson education.
2. Introduction to Algorithms, second edition, T. H. Cormen, C.E. Leiserson, R. L. Rivest, and C.Stein, PHI Pvt. Ltd./ Pearson Education.
3. Algorithm Design: Foundations, Analysis and Internet Examples, M.T. Goodrich and R.Tamassia, John Wiley and sons.

Course outcomes:

1. Ability to analyze the performance of algorithms
2. Describe the data structures of graph coloring, back tracking and disjoint sets.
3. Solves the problems on Knapsack problem, Job sequencing with deadlines, Minimum cost spanningtrees, Single source shortest path problem.
4. Develop algorithms using greedy method
5. Choose Branch and Bound, NP class problems and formulate solutions using standard approaches

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT I:
Introduction- Algorithm definition, Algorithm Specification, Performance Analysis- Space complexity, Time complexity, Randomized Algorithms.
Divide and conquer- General method, applications - Binary search, Merge sort, Quick sort, Strassen's Matrix Multiplication.

Algorithm:

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

- **Input:** there are zero or more quantities, which are externally supplied;
- **Output:** at least one quantity is produced
- **Definiteness:** each instruction must be clear and unambiguous;
- **Finiteness:** if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
- **Effectiveness:** every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Psuedo code for expressing algorithms:

Algorithm Specification: Algorithm can be described in three ways.

1. Natural language like English: When this way is choused care should be taken, we shouldensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithmis small& simple.

3. Pseudo-code Method: In this method, we should typically describe algorithms as program,which resembles language like Pascal & Algol.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,Node. Record

```
{
  data type – 1 data-1;
        .
        .
        .
 data type – n data –
 n;node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record canbe accessed with and period.

5. Assignment of values to variables is done using the assignment statement.

```
<Variable>:= <expression>;
```

6. There are two Boolean values TRUE and FALSE.

  Logical Operators AND, OR, NOT
 Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

For, while and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
                .
                .
                .
        <statement-n>
}
```

For Loop:

```
For variable: = value-1 to value-2 step step do

{
  <statement-1>
        .
        .
        .
<statement-
n>
}
```

repeat-until:

```
repeat <statement-1>
              .
              .
              .
```

```
        <statement-
    n>until<condition>
```

8. A conditional statement has the following forms.

    - If <condition> then <statement>
    - If <condition> then <statement-1>Else <statement-1>

    Case statement:

```
Case
{       : <condition-1> : <statement-1>
                .
                .
                .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>

}
```

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure: Algorithm, the heading takes the form,

Algorithm <Name> (<Parameter lists>)

 As an example, the following algorithm fields & returns the maximum of 'n' givennumbers:

```
1.  Algorithm Max(A,n)
2.  // A is an array of size
n3. {
4.  Result := A[1];
5.  for I:= 2 to n do
6.     if A[I] > Result then
7.          Result :=A[I];
8.    return
Result;9. }
```

In this algorithm (named Max), A & n are procedure parameters. Result & I are Localvariables.

Performance Analysis:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. Oneis analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for thiscan be any one of the following:

i. We would like an algorithm that is easy to understand code and debug.
ii. We would like an algorithm that makes efficient use of the computer'sresources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to theprogram.
2. The quality of code generated by the compiler used to create the objectprogram.
3. The nature and speed of the instructions on the machine used to

execute theprogram,

4. The time complexity of the algorithm underlying the program.

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0

The total time will be 2n+3

Space Complexity:

The space complexity of a program is the amount of memory it needs to run tocompletion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and classinstances.

Environment stack space: The environment stack is used to save informationneeded to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends onfactors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

The space requirement s(p) of any algorithm p may therefore be written as,S(P) = c+ Sp(Instance characteristics)

Where 'c' is a constant.

Example 2:

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

- The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by 'n' is one word, since it is of type integer.
- The space needed by 'a'a is the space needed by variables of tyepe array of floatingpoint numbers.
- This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to

besummed.

- So,we obtain
  Ssum(n)>=(n+s)[n for a[],one each for n,I a& s]

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

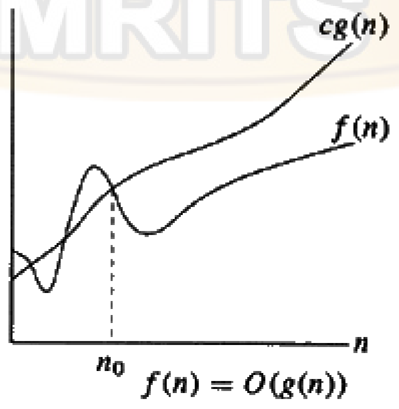
Asymptotic Notations:

The following notations are commonly use notations in performance analysis andused to characterize the complexity of an algorithm:

1. Big–OH (O)
2. Big–OMEGA (Ω),
3. Big–THETA (*Θ*) and
4. Little–OH (o)

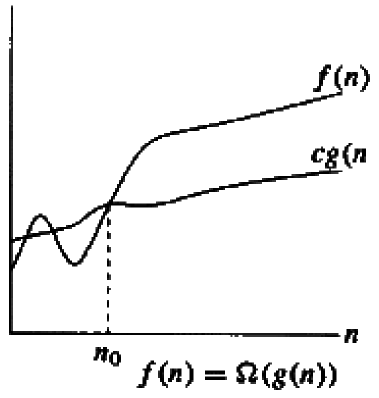
Big–OH O (Upper Bound)

f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is lessthan or equal (≤) that of g(n).

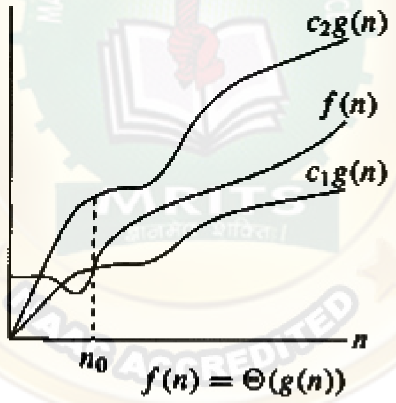


Big–OMEGA Ω (Lower Bound)

f(n) = Ω ***(g(n))*** (pronounced omega), says that the growth rate of f(n) is greater than orequal to (≥) that of g(n).



Big–THETA Θ (Same order)
f(n) = Θ (g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = ***Θ*** (g(n)].



little-o notation

Definition: A theoretical measure of the execution of an *algorithm*, usually the time or memory needed,given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity. The notation is read, "fof n is little oh of g of n".
Formal Definition: f(n) = o(g(n)) means for all c > 0 there exists some k > 0 such that 0 ≤ f(n) < cg(n) forall n ≥ k. The value of k must not depend on n, but may depend on c.

Different time complexities
Suppose ‘M’ is an algorithm, and suppose ‘n’ is the size of the input data. Clearlythe complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

O(1), O(log2 n), O(n), O(n. log2 n), O(n^2), O(n^3), O(2^n), n! and n^n

Classification of Algorithms

If ‘n’ is the number of data items to be processed or degree of polynomial or the size ofthe file to be sorted or searched or the number of nodes in a graph etc.

1	Next instructions of most programs are executed once or at most only a fewtimes. If all the instructions of a program have this property, we say that its running time is a constant.
Log n	When the running time of a program is logarithmic, the program getsslightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.
n	When the running time of a program is linear, it is generally the case that a

small amount of processing is done on each input element. This is theoptimal situation for an algorithm that must process n inputs.

n log n This running time arises for algorithms that solve a problem by breaking itup into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nestedloop) whenever n doubles, the running time increases fourfold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only onsmall problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

n	log2 n	n*log2n	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructionsexecuted by a 1 gigaflop computer in 5000 years.

Randomized algorithm:

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use random number to pick the nextpivot (or we randomly shuffle the array). Quicksort is a familiar, commonly used algorithm in which randomness can be useful. Any deterministic version of this algorithm requires O(n2) time to sort n numbers for some well-defined class of degenerate inputs (such as an already sorted array), with thespecific class of inputs that generate this behavior defined by the protocol for pivot selection. However, if the algorithm selects pivot elements uniformly at random, it has a provably high probability of finishingin O(n log n) time regardless of the characteristics of the input. Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

Divide and Conquer

General Method:

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement intime complexity. For example, from O (n^2) to O (n log n) to sort theelements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide	:	Divide the problem into a number of sub problems. The sub problemsare solvedrecursively.
Conquer	:	The solution to the original problem is then formed from the solutionsto the subproblems (patching together theanswers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a verypowerful use ofrecursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are leftundefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
        if SMALL (P) then return S (p);else
        {
                divide p into smaller instances $p_1$, $p_2$, .... $P_k$, k□
                                                          1;appl
                yDANDC to each of these sub problems;
                return (COMBINE (DANDC ($p_1$) , DANDC ($p_2$),...., DANDC ($p_k$));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. Thesesub problems p_1, p_2, . . . , p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computingtime of DANDC

Where, T (n) is the time for DANDC on ‘n’ inputs
g (n) is the time to complete the answer directly for small inputsandf (n) is the time forDivide and Combine

Binary Search:

If we have ‘n’ records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element ‘x’, binary search is used to find the corresponding element from the list. In case ‘x’ is present, we have to determine a value ‘j’ suchthat a[j] = x (successful search). If ‘x’ is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare ‘x’ with a[mid]. If x = a[mid] then the desired record has been found.If x < a[mid] then ‘x’ must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past ofthe file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of‘x’ with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between ‘x’ and a[mid], and since an array of length ‘n’ can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexityof Binary search is about**$\log_2 n$**

low and *high* are integer variables such that each time through the loop either ‘x’ is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if ‘x’ is not present.

```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    if ($l = i$) then // If Small($P$)
    {
        if ($x = a[i]$) then return $i$;
        else return 0;
    }
    else
    { // Reduce $P$ into a smaller subproblem.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                return BinSrch($a, i, mid - 1, x$);
            else return BinSrch($a, mid + 1, l, x$);
    }
}
```

Recursive binary search

```
Algorithm BinSearch($a, n, x$)
// Given an array $a[1 : n]$ of elements in nondecreasing
// order, $n \geq 0$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    $low := 1$; $high := n$;
    while ($low \leq high$) do
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        if ($x < a[mid]$) then $high := mid - 1$;
        else if ($x > a[mid]$) then $low := mid + 1$;
             else return $mid$;
    }
    return 0;
}
```

Iterative binary search

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9

Number of comparisons = 4

Number of comparisons = 4

2. Searching for x = 82

4. Searching for x = -14

Number of comparisons = 3

Number of comparisons = 3

3. Searching for x = 42

```
                                          5
            found
                                  6     9    7
                                  6     6    6
                                  7     6 not found

  low   high   mi
   1     9     d5
   6     9     7
   8     9     8
              found
                                  low   high   mi
                                   1     9     d5
                                   1     4     2
                                   1     1     1

  low   high   mi                  2     1   not found
   1     9     d
```

Continuing in this manner the number of element comparisons needed to find each ofnine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 orapproximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a[1], a[1] < x < a[2], a[2] < x < a[3], a[5] < x < a[6], a[6] < x < a[7] or a[7] < x < a[8] the algorithm requires 3 element comparisons to determine that 'x'is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of elementcomparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

The time complexity for a successful search is O(log n) and for an unsuccessfulsearch is Θ(log n).

Successful searches			**un-successful searches**
Θ(1), Best	Θ(log n), average	Θ(log n) worst	Θ(log n) best, average and worst

Analysis for worst case

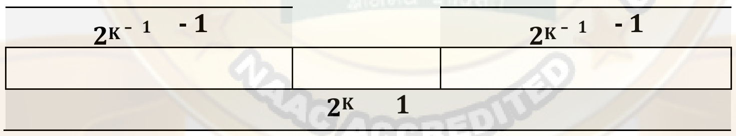
Let T (n) be the time complexity of Binary

searchThealgorithm sets mid to [n+1 / 2]

Therefore,

T(0)	=	0	
T(n)	=	1	if x = a [mid]
	=	1 + T([(n + 1) / 2] – 1)	if x < a [mid]
	=	1 + T(n – [(n + 1)/2])	if x > a [mid]

Let us restrict 'n' to values of the form n = 2^K – 1, where 'k' is a non-negative integer. The array alwaysbreaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{2^K - 1 + 1}{2} \right\rfloor = 2^{K-1}$ for K > 1

Giving,

T(0)	=	0	
T(2^k – 1)	=	1	if x = a [mid]
	=	1 + T(2^{K-1} – 1)	if x < a [mid]
	=	1 + T(2^{k-1} – 1)	if x > a [mid]

In the worst case the test x = a[mid] always fails, sow(0)

= 0w(2^k – 1) = 1 + w(2^{k-1} – 1)

This is now solved by repeated substitution:w(2^k – 1)

= 1 + w(2^{k-1}– 1)

$$= \quad 1 + [1 + w(2^{k-2} - 1)]$$
$$= \quad 1 + [1 + [1 + w(2^{k-3} - 1)]]$$
$$= \quad \ldots\ldots\ldots$$
$$= \quad \ldots\ldots\ldots$$
$$= \quad i + w(2^{k-i} - 1)$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$But as $2^K - 1 = n$,

so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

for $n = 2^K-1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form 2^K-1 weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form 2^K-1.

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.

This strategy is so simple, and so efficient but the problem here is that there seemsto be no easy way tomerge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this canbe done in one pass through the input, if the output is put ina third list.

Algorithm

```
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
        if (low < high)
        {
                mid := ⌊(low  + high)/2⌋              //finds where to split the set
                MERGESORT(low,  mid);                  //sort one subset
                MERGESORT(mid+1, high); //sort the other subset
                MERGE(low,  mid,  high);               // combine the
                results
        }
}
```

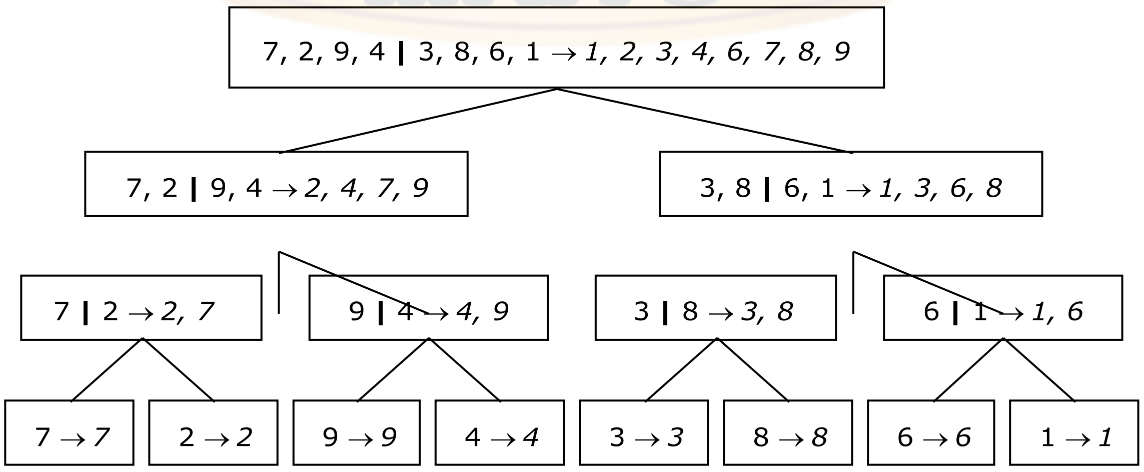
```
Algorithm MERGE (low, mid, high)
// a (low : high) is a global array containing two sorted subsets
// in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.
{
        h :=low; i := low; j:= mid + 1; while ((h
        ≤mid) and (J ≤ high)) do
        {
                if (a[h] ≤ a[j]) then
                {
                        b[i] := a[h]; h := h + 1;
                }
                els
                e
                {       b[i] :=a[j]; j := j + 1;

                }

                i := i + 1;
        }
        if (h > mid) then
                for k := j to high do
                {
                        b[i] := a[k]; i := i + 1;
                }
        else
                for k := h to mid do
                {
                        b[i] := a[K]; i := i + l;
                }
        for k := low to high do
                a[k] := b[k];
}
```

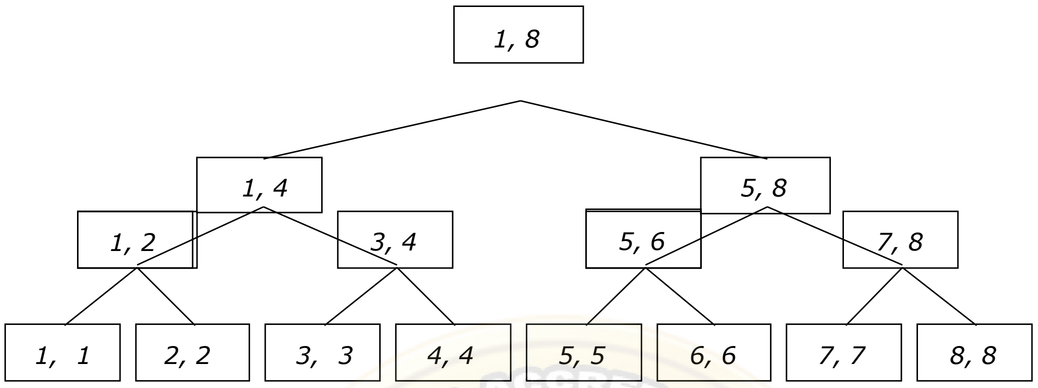
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustratemerge sort algorithm:



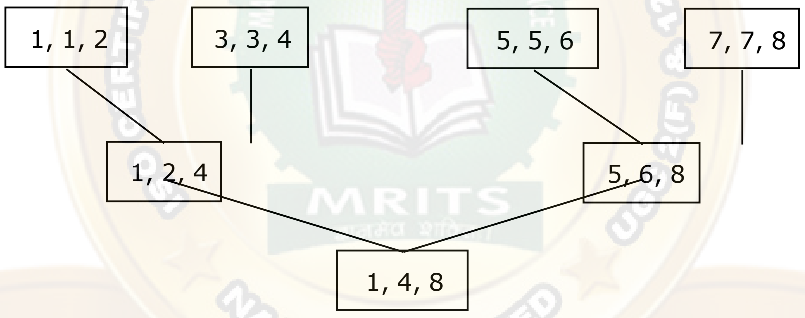
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when itis applied to 8 elements. The values in each node are the valuesof the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is asfollows:



Analysis of Merge Sort

We will assume that ‘n’ is a power of 2, so that we always split into even halves, sowe solve for the case n =2^k.

For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort ‘n’ numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says this exactly:

$$T(1) = 1$$
$$T(n) = 2\,T(n/2) + n$$

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right–hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

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

We have,

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

Again, by substituting n/4 into the main equation, we see that

$$4T(n/4) = 4\,(2T(n/8)) + n/4 = 8\,T(n/8) + n$$

So we have,

$$T(n/4) = 2\,T(n/8) + n$$
$$T(n) = 8\,T(n/8) + 3n$$

Continuing in this manner, we obtain:

$$T(n) = 2^k\,T(n/2^k) + K.n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

$$T(n) = 2^{\log_2 n}\,T\left(\frac{2^k}{2^k}\right) + \log_2 n\,.\,n$$

$= n\,T(1) + n \log n$
$= n \log n + n$ Representing

this in O notation:

$T(n) = \mathbf{O(n \log n)}$

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n'is not a power of 2. The answer turns out to be almostidentical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably.*The Best and worst case time complexity of Merge sort is O(n logn).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic exampleof divide and conquer technique (1969).

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
        for j :=1 to n do
                c[i, j] := 0;
                for K: = 1 to n do
                        c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication ofsinglenumbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers threemultiplication likethis:

⎤A_{11} A_{12} ⎤B_{11} B_{12} ⎤ ⎤C_{11} C_{12} ⎤⎤
⎤⎤
⎡A ⎤ ⎤B B ⎡C C

A
⎤ 21 22⎤ ⎤ 21 22⎤ ⎤ 21 22⎤

Then c_{ij} can be found by the usual matrix multiplication algorithm,$C_{11} = A_{11}\,.\,B_{11}$

$+ A_{12}\,.\,B_{21}$

$C_{12} = A_{11}\,.\,B_{12} + A_{12}\,.\,B_{22}$ $C_{21} =$

$A_{21}\,.\,B_{11} + A_{22}\,.\,B_{21}$ $C_{22} = A_{21}\,.\,B_{12}$

$+ A_{22}\,.\,B_{22}$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$$T(1) = 1$$
$$T(n) = 8\,T(n/2)$$

Which leads to T (n) = O (n^3), where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$P = (A_{11} + A_{22})\ (B_{11} + B_{22})$ $Q =$

$(A_{21} + A_{22})\ B_{11}$

$R = A_{11}\ (B_{12} - B_{22})$ $S = A_{22}$

$(B_{21} - B_{11})$ $T = (A_{11} +$

$A_{12})\ B_{22}$

$U = (A_{21} - A_{11})\ (B_{11} + B_{12})$ $V = (A_{12}$

$- A_{22})\ (B_{21} + B_{22})$ $C_{11} = P + S - T + V$

$C_{12} = R + T$ $C_{21} =$

$Q + S$

$C_{22} = P + R - Q + U.$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$T(1) = 1$

$T(n) = 7\ T(n/2)$

Solving this for the case of $n = 2^k$ is easy:

$T(2^k) = 7\ T(2^{k-1})$

$= 7^2\ T(2^{k-2})$

$=$ - - - - - -

$=$ - - - - - -

$= 7^i\ T(2^{k-i})$

Put i = k

$= 7^k\ T(1)$

$= 7^k$

That is, $T(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$= O(n^{\log_2 7}) = O(^2n^{.81})$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, , w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out ofplace to work its way into its proper position in the sortedsequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves theO(n^2) behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearrangedin this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.

- If j > i, interchange a[j] with a[i]
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sortfunction sorts allelements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completelysorted.
- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ...
  . . . x[j-1] and x[j+1], x[j+2],   x[high].
- It calls itself recursively to sort the left sub-array x[low], x[low+1], ...
  . . x[j-1] between positions low and j-1 (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ...
  . . . x[high] between positions j+1 and high.

```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \dots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j$ := Partition($a, p, q+1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j-1$);
            QuickSort($j+1, q$);
        // There is no need for combining solutions.
    }
}
```

```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}

Algorithm Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j

					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot& j
pivot					j, i								swap pivot&j
(02	08	16	06	04)	**24**								
pivot,j	i												swap pivot&j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot&j
	pivot,j	i											
	(04)	**06**											swap pivot&j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	
							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot&j
							45 pivot, j, i						swap pivot&j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot& j
									57 pivot, j, i				
											(70	79)	
											pivot,j	i	swap pivot& j
											70		
												79 pivot,j, i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. Wewill do the analysis for a quick sort, assuming a random pivot(and no cut off for small files).

We will take T (0) = T (1) = 1, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear timespent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + Cn \qquad - \qquad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1.

Worst Case Analysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, therecurrence is:

$$T(n) = T(n - 1) + Cn \qquad n > 1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly, thus

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

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

- - - - - - - -

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

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

$$= \mathbf{O(n^2)} \qquad - \qquad (3)$$

Best and Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) +---------------------- + T(n-2) +T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) + ------------------------+ T(n-2)] \

Subtracting both sides:

nT(n) –(n-1)T(n-1) = [n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1)nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)
T(n) = 2 + (n+1)T(n-1)/n
The recurrence relation obtained is:
T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Using the method of subsititution:

T(n)/(n+1)	=	2/(n+1) + T(n-1)/n
T(n-1)/n	=	2/n + T(n-2)/(n-1)
T(n-2)/(n-1)	=	2/(n-1) + T(n-3)/(n-2)
T(n-3)/(n-2)	=	2/(n-2) + T(n-4)/(n-3)
.		.
.		.
T(3)/4	=	2/4 + T(2)/3
T(2)/3	=	2/3 + T(1)/2 T(1)/2 = 2/2 + T(0)

Adding both sides:
T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ------------------------------- + T(2)/3 + T(1)/2]
= [T(n-1)/n + T(n-2)/(n-1) + ----------------------------+ T(2)/3 + T(1)/2] + T(0)+
[2/(n+1) + 2/n + 2/(n-1) + ------------------------- +2/4 + 2/3]
Cancelling the common terms:
T(n)/(n+1) = 2[1/2 +1/3 +1/4+ ---------------------------- +1/n+1/(n+1)]

T(n) = (n+1)2[$\square_{2 \le k \le n+1} 1/k$

=2(n+1) [–]

=2(n+1)[log (n+1) – log 2]

=2n log (n+1) + log (n+1)-2n log 2 –log 2

T(n)= O(n log n)

DISJOINT SETS

DISJOINT SET OPERATIONS:

Set:
A set is a collection of distinct elements. The Set can be represented, for examples, asS1={1,2,5,10}.

Disjoint Sets:
The disjoints sets are those do not have any common element.
For example S1= {1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjointsets.

Disjoint Set Operations:
The disjoint set operations are
1. Union
2. Find

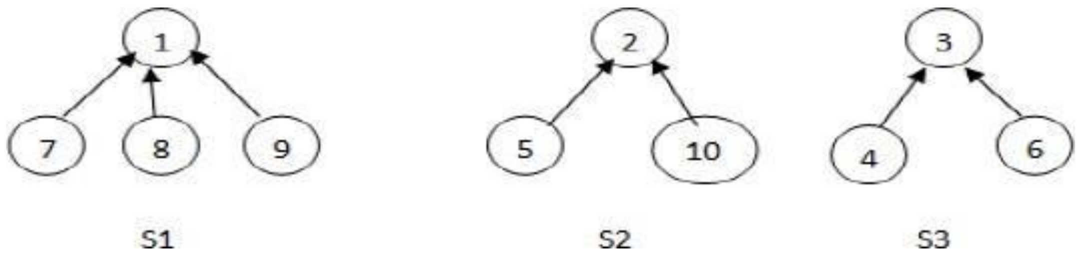
Disjoint set Union:
If Si and Sj are two disjoint sets, then their union Si U Sj consists of all the elements x suchthat x is in Si or Sj.

FIND:
Example:
S1={1,7,8,9} S2={2,5,10}
S1 U S2={1,2,5,7,8,9,10}
Given the element I, find the set containing i.
Example:
S1 = {1,7,8,9}   S2 = {2,5,10}   S3 = {3,4,6}
Then,
Find(4)= S3   Find(5) = S2   Find(7) = S1

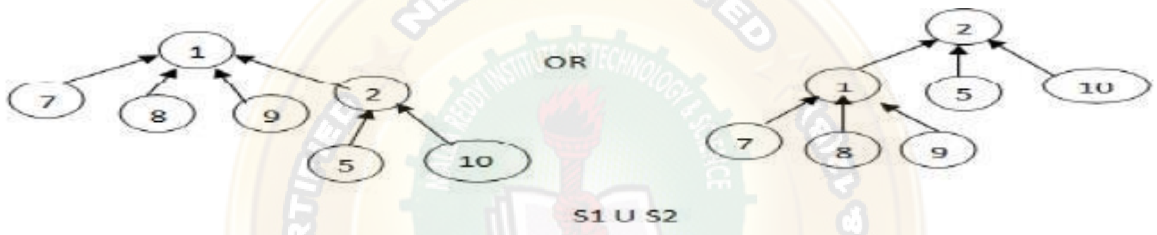
Set Representation:
The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

Example:
S1={1,7,8,9} S2={2,5,10} s3={3,4,6}

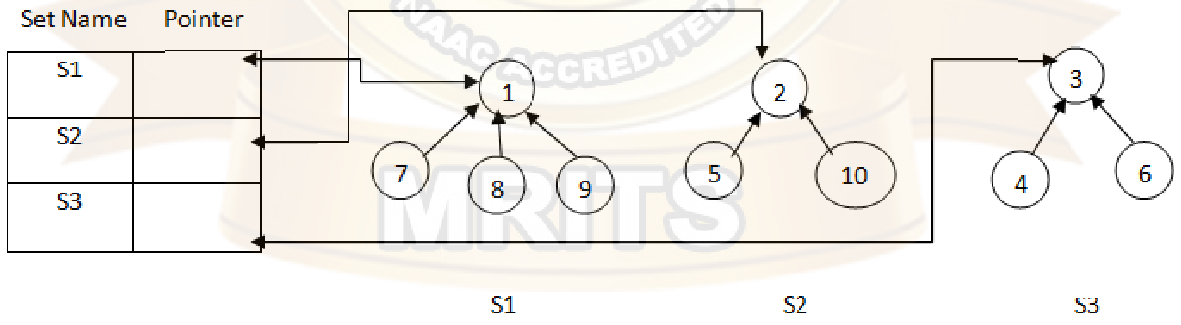
Then these sets can be represented as



Disjoint Union:
To perform disjoint set union between two sets Si and Sj can take any one root andmake it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.



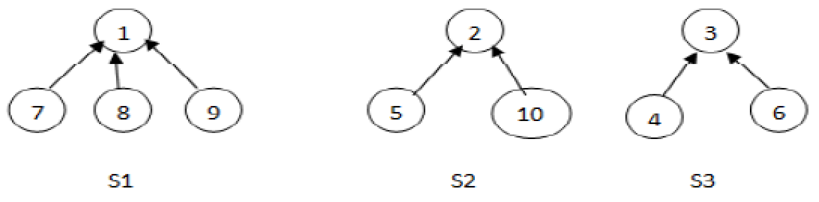
Find:
To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer toroot.



UNION AND FIND ALGORITHMS:

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The indexvalues represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:
For the following sets the array representation is as shown below.



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	-1	-1	3	2	3	1	1	1	2

ALGORITHM FOR UNION OPERATION:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . Andmake the parent of i as j i.e, make the second root as the parent of first root.

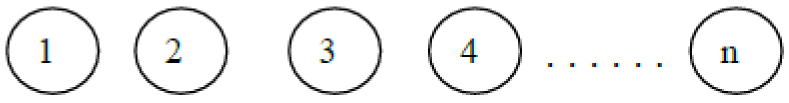
```
Algorithm SimpleUnion(i,j)
{
   P[i]:=j;
}
```

ALGORITHM FOR FIND OPERATION:

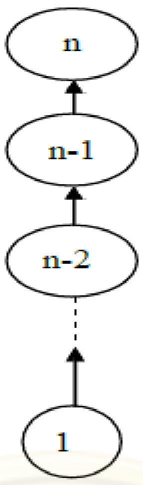
The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at Iuntil it reaches a node with parent value -1.

```
Algorithms SimpleFind(i)
{
   while( P[i]≥0) do i:=P[i]; return i;
}
```

Analysis of SimpleUnion(i,j) and SimpleFind(i):
Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets



Then if we want to perform following sequence of operations Union(1,2) , Union(2,3).......Union(n-1,n) and sequence of Find(1), Find(2)......... Find(n).
The sequence of Union operations results the degenerate tree as below.

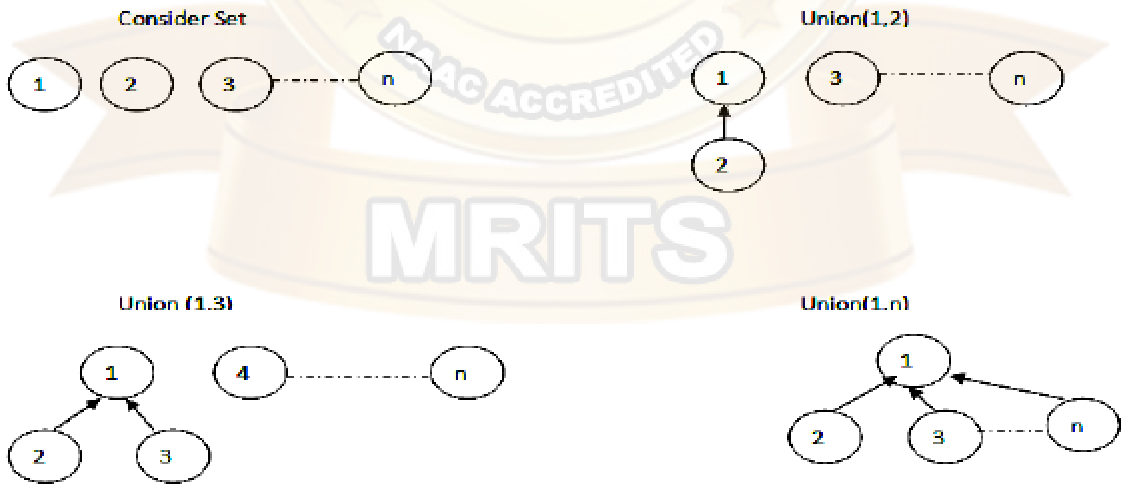


Since, the time taken for a Union is constant, the n-1 sequence of union can be processed intime O(n). And for the sequence of Find operations it will take time complexity of

$O(\sum_{i=1}^{n} i) = O(n^2)$.

We can improve the performance of union and find by avoiding the creation of degeneratetree by applying weighting rule for Union.

Weighting rule for Union:
If the number of nodes in the tree with root I is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.

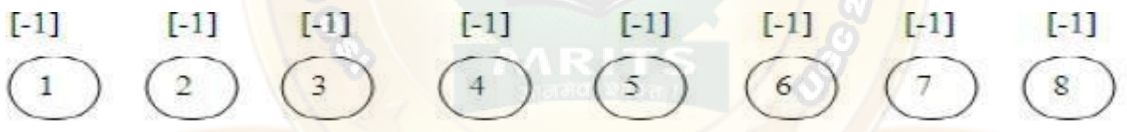


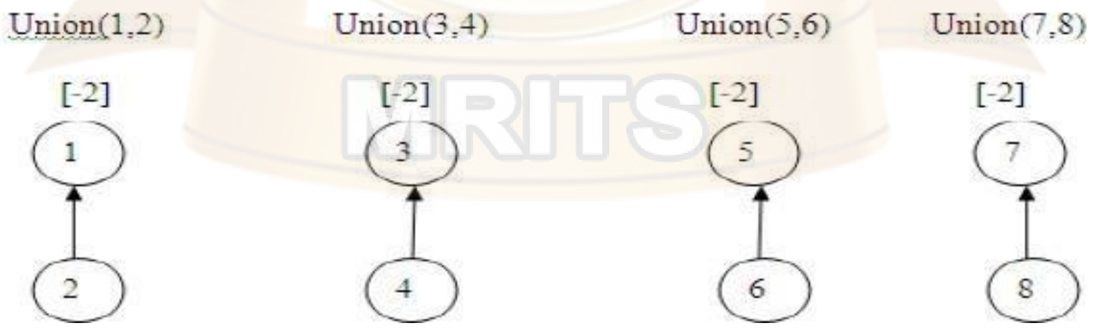
To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then count[i] equals to number of nodes in tree with root i.
Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

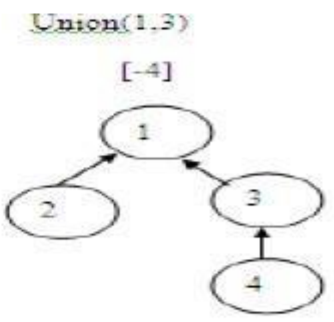
```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j , i≠j using the weighted rule
// P[i]=-count[i] and p[j]=-count[j]
{
     temp:= P[i]+P[j];
     if (P[i]>P[j]) then
    {
       // i has fewer nodes
       P[i]:=j;P[j]:=temp;
    }
    else
    {
       // j has fewer nodes
       P[j]:=i;P[i]:=temp;
    }
}
```

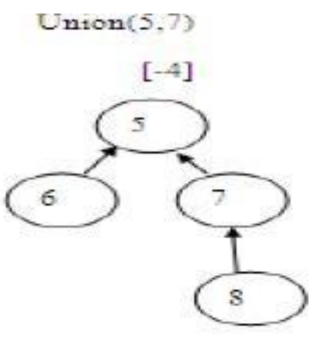
Collapsing rule for find:

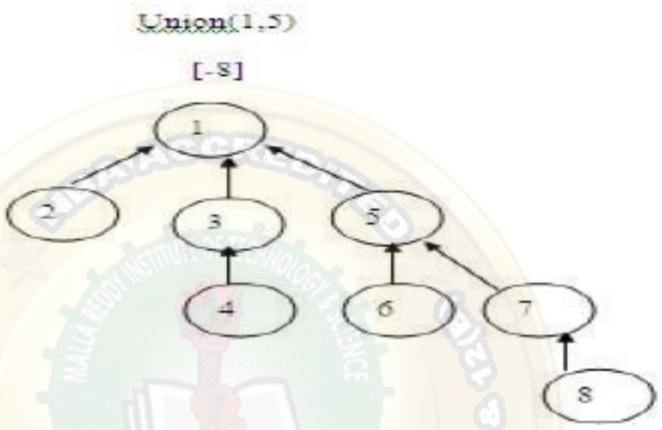
If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i].
Considerthe tree created by WeightedUnion() on the sequence of 1≤i≤8.
Union(1,2), Union(3,4), Union(5,6) and Union(7,8)











Now process the following eight find operations Find(8), Find(8).....................Find(8)
If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24moves .
When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i
// use the collapsing rule to collapse all nodes from i to root.
{
        r:=i;
        while(P[r]>0) do r:=P[r]; //Find root while(i≠r)
        {
            //reset the parent node from element i to the root
           s:=P[i];P[i]:=r;
           i:=s;
        }
}
```

BACKTRACKING

BACKTRACKING

The general method—8 queens problem—Sum of subsets—Graph coloring

BACKTRACKING

- It is one of the most general algorithm design techniques.

- Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.

- To apply backtracking method, tne desired solution must be expressible as an n-tuple (x1…xn) where xi is chosen from some finite set Si.

- The problem is to find a vector, which maximizes or minimizes a criterion function
  P(x1….xn).

- The major advantage of this method is, once we know that a partial vector (x1,…xi)
  will not lead to an optimal solution that (mi+1……. mn) possible test vectors may be
  ignored entirely.

- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

- These constraints are classified as:

  i) Explicit constraints.
  ii) Implicit constraints.

1) Explicit constraints:

Explicit constraints are rules that restrict each Xi to take values only from a given set.Some examples are,
Xi>=0 or Si = {all non-negative real nos.}Xi =0 or 1 or Si={0,1}.
Li <= Xi<=Ui or Si= {a: Li<= a<=Ui}

- All tupules that satisfy the explicit constraint define a possible solution space for I.

2) Implicit constraints:

The implicit constraint determines which of the tuples in the solution space I can actuallysatisfy the criterion functions.

Algorithm:

```
Algorithm IBacktracking (n)
// This schema describes the backtracking procedure .All solutions are generated in X[1:n]
//and printed as soon as they are determined.
{
k=1
;
While (k 0) do
{
   if (there remains all untried
  X[k] belongs to  T (X[1],[2],.....X[k-1]) and Bk (X[1],.....X[k])) is true ) then
  {
     if(X[1],......X[k] )is the path to the answer node)
     Then write(X[1:k]);
     k=k+1; //consider the next step.
  }
  else k=k-1; //consider backtracking to the previous set.
 }
}
```

- All solutions are generated in X[1:n] and printed as soon as they are determined.

- T(X[1].....X[k-1]) is all possible values of X[k] gives that X[1],... X[k-1] have already been chosen.

- Bk(X[1].........X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.

- Certain problems which are solved using backtracking method are,

1. N-Queens problem.
2. Sum of Subsets
3. Graph coloring

N-QUEENS PROBLEM:

This 8 queens problem is to place n-queens in an ‘N*N’ matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector X (X1…Xn) represents a solution in which Xi is the column of theth row where I th queen is placed.

- First, we have to check no two queens are in same row.

- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in it diagonal.
- Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same value.
- Also, every element on the same diagonal that runs from lower right to upper left has the same value.
- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

STEPS TO GENERATE THE SOLUTION:

 Initialize x array to zero and start by placing the first queen in k=1 in the first row.
 To find the column position start from value 1 to n, where ‘n’ is the no. Of columns
or no. Of queens.
 If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have tocheck whether there is any queen in the same column or diagonal.
 For this considers the previous position, which had already, been found out. Checkwhether
 X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.
 If any one of the conditions is true then return false indicating that k th queen can’t
be placed in position X (k).
 For not possible condition increment X (k) value by one and precede d until theposition is found.
 If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have tocheck whether there is any queen in the same column or diagonal.
 For this considers the previous position, which had already, been found out. Checkwhether
 X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.
 If any one of the conditions is true then return false indicating that k th queen can’t
be placed in position X (k).
 For not possible condition increment X (k) value by one and precede d until theposition is found.
 If the position X (k) n and k=n then the solution is generated completely.
 If k<n, then increment the ‘k’ value and find position of the next queen.
 If the position X (k)>n then k th queen cannot be placed as the size of the matrix is
‘N*N’.
 So decrement the ‘k’ value by one i.e. we have to back track and after the position of

the previous queen.

Algorithm:

```
Algorithm place (k,I)
//return true if a queen can be placed in k th row and I th column. otherwise it returns
false.
//X[] is a global array whose first k-1 values have been set. Abs ® returns the absolute
value
//of r.
{
   For j=1 to k-1 do
     If ((X [j]=I) //two in same
    column.Or (abs (X [j]-I)=Abs (j-
    k)))
  Then return
  false;
  Return true;
}
```

Algorithm Nqueen (k,n)

```
//using backtracking it prints all possible positions of n queens in 'n*n' chessboard. So
//that they are non-tracking.
{
   For I=1 to n do
  {
    If place (k,I) then
   {
      X [k]=I;
      If (k=n) then write (X
      [1:n]);Else
      nquenns(k+1,n) ;
   }
  }
}
```

Example: 4 queens.
Two possible solutions are

	Q		
			Q
Q			
		Q	

Solutin-1
(2 4 1 3)

		Q	
Q			
			Q
	Q		

Solution 2
(3 1 4 2)

SUM OF SUBSETS:

- We are given 'n' positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.

- If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.

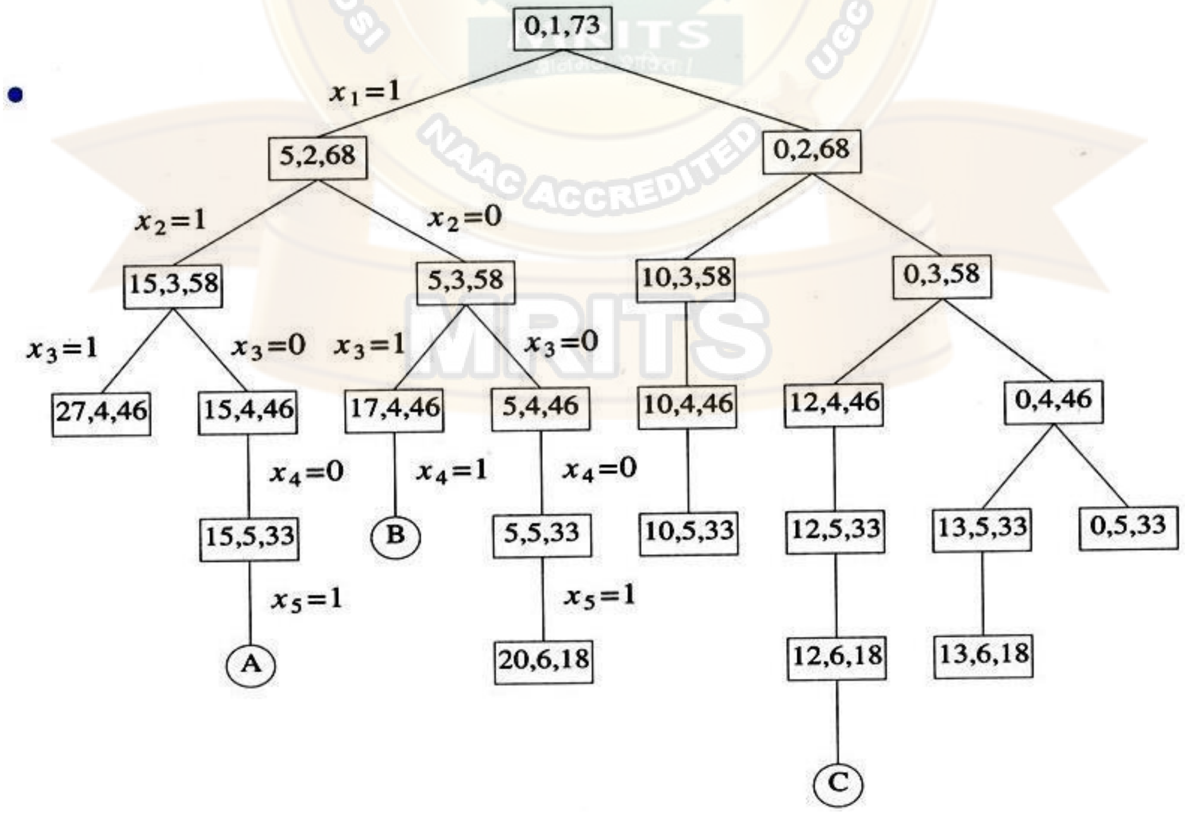
- If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and right to X(i)=0.

Example:

- Given n=6,M=30 and W(1...6)=(5,10,12,13,15,18).We have to generate all possible
  combinations of subsets whose sum is equal to the given value M=30.

- In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets,'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.

- The state space tree for the given problem is,



Ist solution is **A** -> 1 1 0 0 1 0
IInd solution is **B** -> 1 0 1 1 0 0

III rd solution is **C** -> 0 0 1 0 0 1

- In the state space tree, edges from level ‘i’ nodes to ‘i+1’ nodes are labeled with the
  values of Xi, which is either 0 or 1.
- The left sub tree of the root defines all subsets containing Wi.
- The right subtree of the root defines all subsets, which does not include Wi.

Generation of state space tree:

 Maintain an array X to represent all elements in the set.

 The value of Xi indicates whether the weight Wi is included or not.

 Sum is initialized to 0 i.e., s=0.

 We have to check starting from the first node.

 Assign X(k)<- 1.

 If S+X(k)=M then we print the subset because the sum is the required output.

If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to check for the next k.

 After generating the left sub tree we have to generate the right sub tree, for this we have to check S+W(k+1)<=M. Because W(k) is omitted and W(k+1) has to be selected.

 Repeat the process and find all the possible combinations of the subset.

Algorithm:

```
Algorithm sumofsubset(s,k,r)
{
//generate the left child. note s+w(k)<=M since Bk-1 is
true.X{k]=1;
If (S+W[k]=m) then write(X[1:k]); // there is no recursive call here as
W[j]>0,1<=j<=n.Else if (S+W[k]+W[k+1]<=m) then sum of sub (S+W[k], k+1,r-
W[k]);
//generate right child and evaluate Bk.
If ((S+ r- W[k]>=m)and(S+ W[k+1]<=m)) then
{ X{k]=0;
```

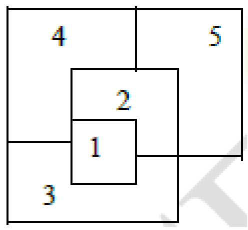
sumofsubset (S, k+1, r- W[k]);

} }

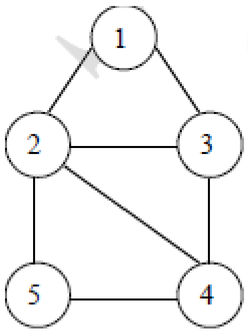
GRAPH COLORING:

- Let 'G' be a graph and 'm' be a given positive integer. If the nodes of 'G' can be colored in such a way that no two adjacent nodes have the same color. Yet only 'M' colors are used. So it's called M-color ability decision problem.
- The graph G can be colored using the smallest integer 'm'. This integer is referred to
  as chromatic number of the graph.
- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.
2 is adjacent to 1, 3, 4, 5
3 is adjacent to 1, 2, 4
4 is adjacent to 1, 2, 3, 5
5 is adjacent to 2, 4



Steps to color the Graph:

- First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.

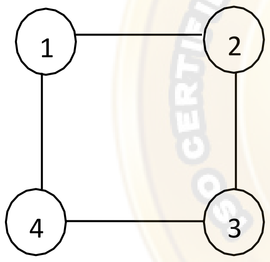
- The Colors will be represented by the integers 1,2,.....m and the solutions will be stored in the array X(1),X(2),... ,X(n) ,X(index) is the color, index is the node.

- Here formula is used to set the color

  is,X(k) = (X(k)+1) % (m+1)
- First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.

- First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then wehave to assign the next value.
- Repeat until all the possible combinations colors are found
- The function which is used to check the adjacent nodes and same color

  is,If((Graph (k,j) == 1) and X(k) = X(j))

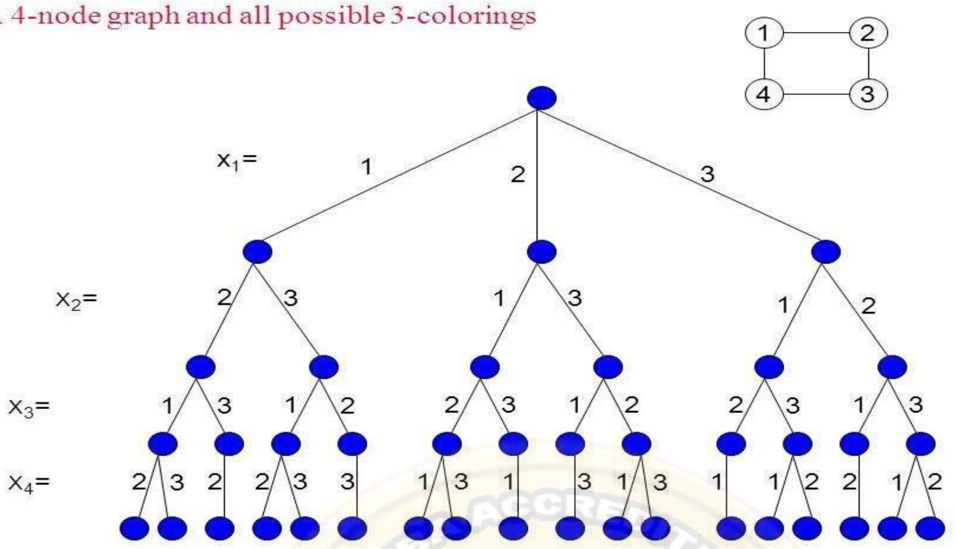


N= 4
M= 3
Adjacency Matrix:

$$\begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix}$$

- The problem is to color the given graph of 4 nodes using 3 colors.
- Node-1 can take the given graph of 4 nodes using 3 colors.
- The state space tree will give all possible colors in that ,the numbers which are insidethe circles are nodes ,and the branch with a number is the colors of the nodes.

A 4-node graph and all possible 3-colorings



Algorithm:

Algorithm mColoring(k)

```
// the graph is represented by its Boolean adjacency matrix G[1:n,1:n] .All assignments
//of
1,2,.......... ,m to the vertices of the graph such that adjacent vertices are assigned
//distinctintegers are printed. 'k' is the index of the next vertex to color.
{
repeat
{
   // generate all legal assignment for X[k].
  Nextvalue(k); // Assign to X[k] a legal color.
     If (X[k]=0) then return; // No new color possible.
     If (k=n) then // Almost 'm' colors have been used to color the 'n' vertices
         Write(x[1:n]);
     Else
     mcoloring(k+1);
}until(false);
}
```

Algorithm Nextvalue(k)

```
// X[1],......X[k-1] have been assigned integer values in the range[1,m] such that
//adjacent values have distinct integers. A value for X[k] is determined in the
//range[0,m].X[k] is assigned the next highest numbers color while maintaining
//distinctness form the adjacent vertices of vertex K. If no such color exists, then X[k] is
0.
{

repeat
{
    X[k] = (X[k]+1)mod(m+1); // next highest color.
    If(X[k]=0) then return; //All colors have been
    used.
       For j=1 to n do

       {
           // Check if this color is distinct from adjacent color.
         If((G[k,j] 0)and(X[k] = X[j]))
           // If (k,j) is an edge and if adjacent vertices have the same
         color.Then break;
       }
     if(j=n+1) then return; //new color found.
   } until(false); //otherwise try to find another color.
 }
```

The time spent by Nextvalue to determine the children is
(mn)Total time is = (m^n n).

UNIT-III

DYNAMIC PROGRAMMING

INTRODUCTION

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations,* that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute anoptimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprisedof optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot beapplied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds

- Set up the dynamic-programming recurrence equations

- Solve the dynamic-programming recurrence equations for the value of theoptimal solution.

- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to

avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Two difficulties may arise in any application of dynamic programming:
1. It may not always be possible to combine the solutions of smaller problems toform the solution of a larger one.

2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does notseen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

Applications of dynamic programming:

1. Optimal binary search trees
2. 0/1 knapsack problem
3. All pairs shortest path problem
4. Traveling sales person problem
5. Reliability design

OPTIMAL BINARY SEARCH TREES

Let us assume that the given set of identifiers is {a1, . . . , an} with a1 < a2 < . . . . < an. Let p (i) be the probability with which we search for ai. Let q (i) be the probability that the identifier x being searched for is such that ai < x < ai+1, 0 <i < n (assume a0 = - ∞and an+1 = +∞). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time. In a binary search tree, the number of comparisons needed to access an element at depth 'd' is d + 1, so if 'ai' is placed at depth 'di', then we want to minimize:

$$\sum_{i=1}^{n} P_i \ (1 + d_i)$$

Let P (i) be the probability with which we shall be searching for 'ai'. Let Q (i) bethe probability of an un-successful search. Every internal node represents a point where a successful search may terminate. Every external node represents apoint where an unsuccessful search may terminate.
The expected cost contribution for the internal node for 'ai' is:

$$P(i) * level\ (a_i) .$$

Unsuccessful search terminate with I = 0 (i.e at an external node). Hence the cost contribution for this node is:

$$Q(i) * \text{level}((E_i) - 1)$$

The expected cost of binary search tree is:

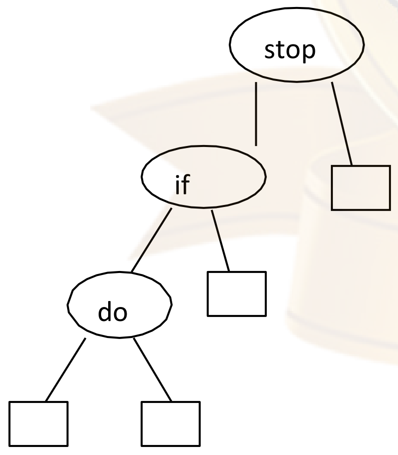
$$\sum_{i=1}^{n} P(i) * \text{level}(a_i) + \sum_{i=0}^{n} Q(i) * \text{level}((E_i) - 1)$$

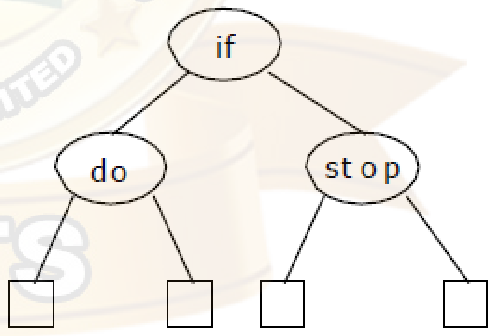
Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

The computation of each of these c(i, j)'s requires us to find the minimum of m quantities. Hence, each such c(i, j) can be computed in time O(m). The total time for all c(i, j)'s with j – i = m is therefore O(nm – m^2).

The total time to evaluate all the c(i, j)'s and r(i, j)'s is therefore:Σnm(nm-m^2) = O(n^3)

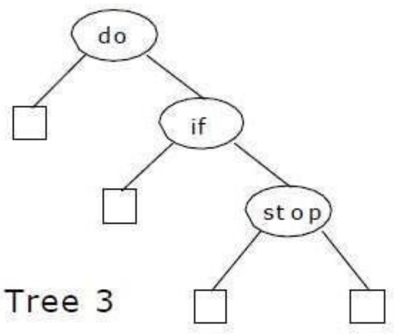
Example 1: The possible binary search trees for the identifier set (a1, a2, a3) = (do, if, stop) are as follows. Given the equal probabilities p (i) = Q (i) = 1/7 for all i, we have:



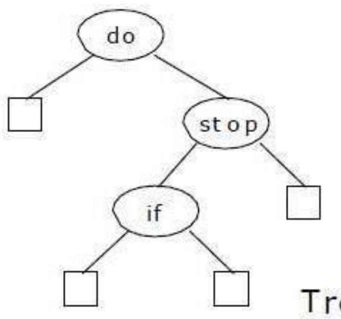


Tree 2

Tree 1



Tree 3



Tree 4

$$\text{Cost (tree \# 1)} = \left(\frac{1}{7} x\, 1 + \frac{1}{7} x\, 2 + \frac{1}{7} x\, 3\right) + \left(\frac{1}{7} x\, 1 + \frac{1}{7} x\, 2 + \frac{1}{7} x\, 3 + \frac{1}{7} x\, 3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

$$\text{Cost (tree \# 2)} = \left(\frac{1}{7} x1 + \frac{1}{7} x2 + \frac{1}{7} x2\right) + \left(\frac{1}{7} x2 + \frac{1}{7} x\, 2 + \frac{1}{7} x\, 2 + \frac{1}{7} x2\right)$$

$$= \frac{1+2+2}{7} + \frac{2+2+2+2}{7} = \frac{5+8}{7} = \frac{13}{7}$$

$$\text{Cost (tree \# 3)} = \left(\frac{1}{7} x\, 1 + \frac{1}{7} x\, 2 + \frac{1}{7} x\, 3\right) + \left(\frac{1}{7} x\, 1 + \frac{1}{7} x\, 2 + \frac{1}{7} x\, 3 + \frac{1}{7} x\, 3\right)$$

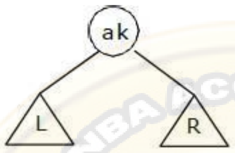
$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

$$\text{Cost (tree \# 4)} = \left(\frac{1}{7} x\, 1 + \frac{1}{7} x\, 2 + \frac{1}{7} x\, 3\right) + \left(\frac{1}{7} x\, 1 + \frac{1}{7} x\, 2 + \frac{1}{7} x\, 3 + \frac{1}{7} x\, 3\right)$$

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

Huffman coding tree solved by a greedy algorithm has a limitation of having thedata only at the leaves and it must not preserve the property that all nodes to the left of the root have keys, which are less etc. Construction of an optimal binary search tree is harder, because the data is not constrained to appear only at the leaves, and also because the tree must satisfy the binary search tree property andit must preserve the property that all nodes to the left of the root have keys, which are less.

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality. A possible approach to this is to make a decision as which of the ai's be arraigned to the root node at 'T'. If we choose 'ak' then is clear that the internal nodes for a1, a2, ... ak-1 as well as the external nodes for the classes Eo, E1, ... Ek-1 will lie in the left sub tree, L, of the root. The remaining nodes will be in the right subtree, R. The structure of an optimal binary search tree is:



$$\text{Cost }(L) = \sum_{i=1}^{K} P(i) * level\ (a_i) + \sum_{i=0}^{K} Q(i) * (level\,(E_i) - 1)$$

$$\text{Cost }(R) = \sum_{i=K}^{n} P(i) * level\ (a_i) + \sum_{i=K}^{n} Q(i) * (level\,(E_i) - 1)$$

The C (i, J) can be computed as:

$$C(i, J) = \min_{i<k\leq J} \{C(i, k-1) + C(k, J) + P(K) + w(i, K-1) + w(K, J)\}$$

$$= \min_{i<k\leq J} \{C(i, K-1) + C(K, J)\} + w(i, J) \quad -- \quad (1)$$

$$\text{Where } W(i, J) = P(J) + Q(J) + w(i, J-1) \quad -- \quad (2)$$

Initially C (i, i) = 0 and w (i, i) = Q (i) for $0 \leq i \leq n$.

Equation (1) may be solved for C (0, n) by first computing all C (i, J) such that J - i = 1
Next, we can compute all C (i, J) such that J - i = 2, Then all C (i, J) with J - i = 3 and so on.

C (i, J) is the cost of the optimal binary search tree 'Tij' during computation we record the root R (i, J) of each tree 'Tij'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation (1).
We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), 0 ≤ i < 4; Knowing W (i, i+2), C (i, i+2) and R (i, i+2), 0 ≤ i < 3 and repeating until W (0, n), C (0, n) and R (0, n) are obtained.
The results are tabulated to recover the actual tree.

Example 1:
Let n = 4, and (a1, a2, a3, a4) = (do, if, need, while) Let P (1: 4) = (3, 3, 1, 1) and Q (0: 4) = (2, 3, 1, 1, 1)
Solution:
Table for recording W (i, j), C (i, j) and R (i, j):

Column Row	0	1	2	3	4
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	14, 25, 2	11, 19, 2			
4	16, 32, 2				

This computation is carried out row-wise from row 0 to row 4. Initially, W (i, i)
= Q
(i) and C (i, i) = 0 and R (i, i) = 0, 0 < i < 4.
Solving for C (0, n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 < i < 4; i = 0, 1, 2 and 3; i < k ≤ J. Start with i = 0; so j = 1; as i < k ≤ j, so the possible value for k = 1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 3 + 3 + 2 = 8
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 8
R (0, 1) = 1 (value of 'K' that is minimum in the above equation). Next with i = 1; so j = 2; as i < k ≤ j, so the possible value for k = 2
W (1, 2) = P (2) + Q (2) + W (1, 1) = 3 + 1 + 3 = 7
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 7
R (1, 2) = 2
Next with i = 2; so j = 3; as i < k ≤ j, so the possible value for k = 3

W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 = 3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] = 3
R (2, 3) = 3
Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k = 4 W (3, 4) = P(4) + Q (4) + W (3, 3) = 1 + 1 + 1 = 3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] = 3
R (3, 4) = 4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 < i < 3; i = 0, 1, 2; i < k ≤ J. Start with i = 0; so j = 2; as i < k ≤ J, so the possible values for k = 1 and 2.
W (0, 2) = P (2) + Q (2) + W (0, 1) = 3 + 1 + 8 = 12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))}
= 12 + min {(0 + 7, 8 + 0)} = 19
R (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.
W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 7 = 9
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3, 3)]}
= W (1, 3) + min {(0 + 3), (7 + 0)} = 9 + 3 = 12
R (1, 3) = 2
Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4.
W(2, 4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 = 5
C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4, 4)]
= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 = 8
R (2, 4) = 3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 < i < 2; i = 0, 1;
i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and 3.
W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 = 14
C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)], [C (0, 2) + C (3, 3)]}
= 14 + min {(0 + 12), (8 + 3), (19 + 0)} = 14 + 11 = 25
R (0, 3) = 2
Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and 4.
W(1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 = 11
C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3, 4)], [C (1, 3) + C (4, 4)]}
= 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 + 8 = 19
R (1, 4) = 2

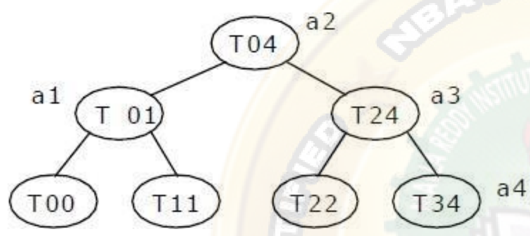
Fourth, Computing all C (i, j) such that j - i = 4; j = i + 4 and as 0 < i < 1; i = 0;i < k ≤ J.
Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and 4.

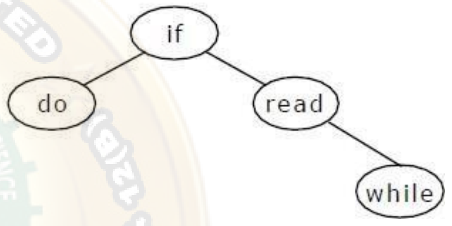
W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 = 16
C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2, 4)], [C (0, 2) + C (3, 4)], [C (0, 3) + C (4, 4)]}
= 16 + min [0 + 19, 8 + 8, 19+3, 25+0] = 16 + 16 = 32
R (0, 4) = 2

From the table we see that C (0, 4) = 32 is the minimum cost of a binary searchtree for (a_1, a_2, a_3, a_4). The root of the tree 'T04' is 'a_2'.

Hence the left sub tree is 'T_{01}' and right sub tree is T_{24}. The root of 'T_{01}' is 'a_1'and the root of 'T_{24}' is a_3.
The left and right sub trees for 'T_{01}' are 'T_{00}' and 'T_{11}' respectively. The root ofT_{01} is 'a_1'
The left and right sub trees for T_{24} are T_{22} and T_{34} respectively. The root of $T_{24}$is 'a_3'.
The root of T_{22} is

nullThe root of T_{34} is





a_4.

Example 2:

Consider four elements a1, a2, a3 and a4 with Q0 = 1/8, Q1 = 3/16, Q2 = Q3 = Q4 = 1/16 and p1 = 1/4, p2 = 1/8, p3 = p4 =1/16. Construct an optimal binary search tree. Solving for C (0, n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 < i < 4; i = 0, 1, 2 and 3; i < k ≤ J. Start with i = 0; so j = 1; as i < k ≤ j, so the possible value for k = 1
W (0, 1) = P (1) + Q (1) + W (0, 0) = 4 + 3 + 2 = 9
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 9 + [(0 + 0)] = 9
R (0, 1) = 1 (value of 'K' that is minimum in the above equation).

Next with i = 1; so j = 2; as i < k ≤ j, so the possible value for k = 2
W (1, 2) = P (2) + Q (2) + W (1, 1) = 2 + 1 + 3 = 6
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 6 + [(0 + 0)] = 6
R (1, 2) = 2
Next with i = 2; so j = 3; as i < k ≤ j, so the possible value for k = 3 W (2, 3) = P(3) + Q (3) + W (2, 2) = 1 + 1 + 1 = 3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] = 3

R (2, 3) = 3
Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k = 4 W (3, 4) =

P(4) + Q (4) + W (3, 3) = 1 + 1 + 1 = 3

C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] = 3
R (3, 4) = 4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 < i < 3; i = 0, 1, 2; i < k ≤ J
Start with i = 0; so j = 2; as i < k ≤ j, so the possible values for k = 1 and 2.
W(0, 2) = P (2) + Q (2) + W (0, 1) = 2 + 1 + 9 = 12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))}
= 12 + min {(0 + 6, 9 + 0)} = 12 + 6 = 18
R (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.
W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 6 = 8
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3, 3)]}
= W (1, 3) + min {(0 + 3), (6 + 0)} = 8 + 3 = 11
R (1, 3) = 2
Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4.
W(2, 4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 = 5
C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4, 4)]
= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 = 8
R (2, 4) = 3

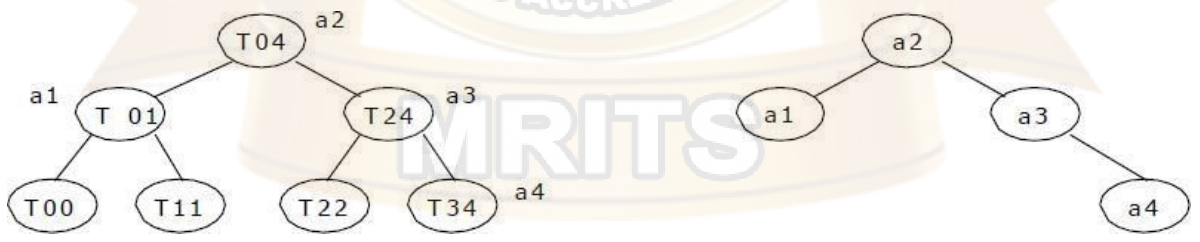
Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 < i < 2; i = 0, 1;
i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and 3.
W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 = 14
C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)],
[C (0, 2) + C (3, 3)]}
= 14 + min {(0 + 11), (9 + 3), (18 + 0)} = 14 + 11 = 25
R (0, 3) = 1
Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and 4.
W(1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 8 = 10
C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3, 4)],
[C (1, 3) + C (4, 4)]}
= 10 + min {(0 + 8), (6 + 3), (11 + 0)} = 10 + 8 = 18
R (1, 4) = 2

Fourth, Computing all C (i, j) such that J - i = 4; j = i + 4 and as 0 < i < 1; i = 0;
i < k ≤ J. Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and 4.
W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 = 16
C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2, 4)],
[C (0, 2) + C (3, 4)], [C (0, 3) + C (4, 4)]}

= 16 + min [0 + 18, 9 + 8, 18 + 3, 25 + 0] = 16 + 17 = 33
R (0, 4) = 2
Table for recording W (i, j), C (i, j) and R (i, j)

Column Row	0	1	2	3	4
0	2, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	9, 9, 1	6, 6, 2	3, 3, 3	3, 3, 4	
2	12, 18, 1	8, 11, 2	5, 8, 3		
3	14, 25, 2	11, 18, 2			
4	16, 33, 2				

From the table we see that C (0, 4) = 33 is the minimum cost of a binary searchtree for (a1, a2, a3, a4)
The root of the tree 'T04' is 'a2'.
Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1'and the root of 'T24' is a3.
The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root ofT01 is 'a1'
The left and right sub trees for T24 are T22 and T34 respectively. The root ofT24 is 'a3'.
The root of T22 is null. The root of T34 is a4.



0/1 KNAPSACK PROBLEM

We are given n objects and a knapsack. Each object i has a positive weight wi and a positive value Vi. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.
A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable xi involves determining which of the values 0 or 1 is to be assigned to it. Let us assume thatdecisions on the x_i are made in the order $x_n, x_{n-1}, \ldots x_1$. Following a decision

on x_n, we may be in one of two possible states: the capacity remaining in m – $w_n$and a profit of p_n has accrued. It is clear that the remaining decisions $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principal of optimality holds.

$$F_n(m) = \max\{f_{n-1}(m),\ f_{n-1}(m - w_n) + p_n\} \quad -- \quad 1$$

For arbitrary $f_i(y)$, $i > 0$, this equation generalizes to:

$$F_i(y) = \max\{f_{i-1}(y),\ f_{i-1}(y - w_i) + p_i\} \quad -- \quad 2$$

Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge fo (y) = 0 for all y and fi (y) = - ∞, y < 0. Then $f_1, f_2, \ldots f_n$ can be successively computed using equation–2.
When the wi's are integer, we need to compute fi (y) for integer y, 0 < y < m. Since fi (y) = - ∞ for y < 0, these function values need not be computed explicitly. Since each fi can be computed from fi - 1 in **Θ** (m) time, it takes **Θ** (m n) time to compute fn. When the wi's are real numbers, fi (y) is needed for real numbers y such that 0 < y < m. So, fi cannot be explicitly computed for ally in this range. Even when the wi's are integer, the explicit **Θ** (m n) computation of fn may not be the most efficient computation. So, we explore **analternative method for both cases.**

The $f_i(y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \ldots < y_k$, such that $f_i(y_1) < f_i(y_2) < \ldots < f_i(y_k)$; $f_i(y) = -\propto$, $y < y_1$; $f_i(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y \leq y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S_i by first computing:

$$S^i_1 = \{(P, W) \mid (P - p_i, W - w_i) \in S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S^i_1 together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Example 1:

Consider the knapsack instance n = 3, (w1, w2, w3) = (2, 3, 4), (P1, P2, P3) =(1, 2, 5) and M = 6.

Solution:

Initially, fo (x) = 0, for all x and fi (x) = - ∞ if x < 0. Fn (M) = max {fn-1 (M), fn-1 (M - wn) + pn}

F3 (6) = max (f2 (6), f2 (6 – 4) + 5} = max {f2 (6), f2 (2) + 5}

F2 (6) = max (f1 (6), f1 (6 – 3) + 2} = max {f1 (6), f1 (3) + 2}

F1 (6) = max (f0 (6), f0 (6 – 2) + 1} = max {0, 0 + 1} = 1
F1 (3) = max (f0 (3), f0 (3 – 2) + 1} = max {0, 0 + 1} = 1
Therefore, F2 (6) = max (1, 1 + 2} = 3
F2 (2) = max (f1 (2), f1 (2 – 3) + 2} = max {f1 (2), -□ + 2}
F1 (2) = max (f0 (2), f0 (2 – 2) + 1} = max {0, 0 + 1} = 1
F2 (2) = max {1, - ∞ + 2} = 1
Finally, f3 (6) = max {3, 1 + 5} = 6

Other Solution:
For the given data we have:
S_0 = {(0, 0)}; $S^0{}_1$ = {(1, 2)}
S_1 = (S_0 U $S^0{}_1$) = {(0, 0), (1, 2)}
X - 2 = 0 => x = 2. y – 3 = 0 => y = 3
X - 2 = 1 => x = 3. y – 3 = 2 => y = 5
$S_1{}^1$ = {(2, 3), (3, 5)}
S^2 = (S^1 U $S^1{}_1$) = {(0, 0), (1, 2), (2, 3), (3, 5)}
X – 5 = 0 => x = 5. y – 4 = 0 => y = 4
X – 5 = 1 => x = 6. y – 4 = 2 => y = 6
X – 5 = 2 => x = 7. y – 4 = 3 => y = 7
X – 5 = 3 => x = 8. y – 4 = 5 => y = 9
S^1 = {(5, 4), (6, 6), (7, 7), (8, 9)}
S^3 = (S^2 U $S^2{}_1$) = {(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)}
By applying Dominance rule,
S^3 = (S^2 U $S^2{}_1$) = {(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)}
From (6, 6) we can infer that the maximum Profit Σ pi xi = 6 and weight Σxiwi = 6

ALL PAIRS SHORTEST PATHS

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i. These two paths are the same when G is undirected.
When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.
The all pairs shortest path problem is to determine a matrix A such that A (i, j)is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n^2) time, the matrix A can be obtained in O (n^3) time.

The dynamic programming solution, called Floyd's algorithm, runs in O (n3) time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

The shortest i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let Ak (i, j) represent the length of a shortest path from i to j going through no vertex ofindex greater than k, we obtain:

$$A^k (i, j) = \{\min \{\min_{1 \le k \le n} \{A^{k-1} (i, k) + A^{k-1} (k, j)\}, c (i, j)\}$$

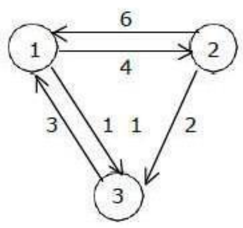
Algorithm:

```
Algorithm All Paths (Cost, A, n)
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for 1 < i < n.
{
for i := 1 to n do
        for j:= 1 to n do
                A [i, j] := cost [i, j]; // copy cost into
A.for k := 1 to n do
        for i := 1 to n do
                for j := 1 to n do
                        A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
}
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n^3).

Example 1:

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



Cost adjacency matrix $(A^0) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$

General formula: $\min\limits_{1 \le k \le n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$

Solve the problem for different values of k = 1, 2 and 3

Step 1: Solving the equation for, k = 1;

$A^1(1, 1) = \min \{(A^o(1, 1) + A^o(1, 1)), c(1, 1)\} = \min \{0 + 0, 0\} = 0$

$A^1(1, 2) = \min \{(A^o(1, 1) + A^o(1, 2)), c(1, 2)\} = \min \{(0 + 4), 4\} = 4$

$A^1(1, 3) = \min \{(A^o(1, 1) + A^o(1, 3)), c(1, 3)\} = \min \{(0 + 11), 11\} = 11$

$A^1(2, 1) = \min \{(A^o(2, 1) + A^o(1, 1)), c(2, 1)\} = \min \{(6 + 0), 6\} = 6$

$A^1(2, 2) = \min \{(A^o(2, 1) + A^o(1, 2)), c(2, 2)\} = \min \{(6 + 4), 0)\} = 0$

$A^1(2, 3) = \min \{(A^o(2, 1) + A^o(1, 3)), c(2, 3)\} = \min \{(6 + 11), 2\} = 2$

$A^1(3, 1) = \min \{(A^o(3, 1) + A^o(1, 1)), c(3, 1)\} = \min \{(3 + 0), 3\} = 3$

$A^1(3, 2) = \min \{(A^o(3, 1) + A^o(1, 2)), c(3, 2)\} = \min \{(3 + 4), \propto\} = 7$

$A^1(3, 3) = \min \{(A^o(3, 1) + A^o(1, 3)), c(3, 3)\} = \min \{(3 + 11), 0\} = 0$

$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$

Step 2: Solving the equation for, K = 2;

$A^2(1, 1) = \min \{(A^1(1, 2) + A^1(2, 1), c(1, 1)\} = \min \{(4 + 6), 0\} = 0$

$A^2(1, 2) = \min \{(A^1(1, 2) + A^1(2, 2), c(1, 2)\} = \min \{(4 + 0), 4\} = 4$

$A^2(1, 3) = \min \{(A^1(1, 2) + A^1(2, 3), c(1, 3)\} = \min \{(4 + 2), 11\} = 6$

$A^2(2, 1) = \min \{(A(2, 2) + A(2, 1), c(2, 1)\} = \min \{(0 + 6), 6\} = 6$

$A^2(2, 2) = \min \{(A(2, 2) + A(2, 2), c(2, 2)\} = \min \{(0 + 0), 0\} = 0$

$A^2(2, 3) = \min \{(A(2, 2) + A(2, 3), c(2, 3)\} = \min \{(0 + 2), 2\} = 2$

$A^2(3, 1) = \min \{(A(3, 2) + A(2, 1), c(3, 1)\} = \min \{(7 + 6), 3\} = 3$

$A^2(3, 2) = \min \{(A(3, 2) + A(2, 2), c(3, 2)\} = \min \{(7 + 0), 7\} = 7$

$A^2(3, 3) = \min \{(A(3, 2) + A(2, 3), c(3, 3)\} = \min \{(7 + 2), 0\} = 0$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 3: Solving the equation for, k = 3;

$A^3(1, 1) = \min\{A^2(1, 3) + A^2(3, 1), c(1, 1)\} = \min\{(6 + 3), 0\} = 0$

$A^3(1, 2) = \min\{A^2(1, 3) + A^2(3, 2), c(1, 2)\} = \min\{(6 + 7), 4\} = 4$

$A^3(1, 3) = \min\{A^2(1, 3) + A^2(3, 3), c(1, 3)\} = \min\{(6 + 0), 6\} = 6$

$A^3(2, 1) = \min\{A^2(2, 3) + A^2(3, 1), c(2, 1)\} = \min\{(2 + 3), 6\} = 5$

$A^3(2, 2) = \min\{A^2(2, 3) + A^2(3, 2), c(2, 2)\} = \min\{(2 + 7), 0\} = 0$

$A^3(2, 3) = \min\{A^2(2, 3) + A^2(3, 3), c(2, 3)\} = \min\{(2 + 0), 2\} = 2$

$A^3(3, 1) = \min\{A^2(3, 3) + A^2(3, 1), c(3, 1)\} = \min\{(0 + 3), 3\} = 3$

$A^3(3, 2) = \min\{A^2(3, 3) + A^2(3, 2), c(3, 2)\} = \min\{(0 + 7), 7\} = 7$

$A^3(3, 3) = \min\{A^2(3, 3) + A^2(3, 3), c(3, 3)\} = \min\{(0 + 0), 0\} = 0$

$$A^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

TRAVELING SALES PERSON PROBLEM

Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \infty$ if < i, j>$\notin$ E. Let |V| = n and assume n
> 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.

Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g (1, V – {1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{--} \quad 1$$

Generalizing equation 1, we obtain (for i $\notin$ S)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(i, S - \{j\})\} \quad \text{--} \quad 2$$

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for all choices of k.

Complexity Analysis:

For each value of |S| there are n – 1 choices for i. The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$.

Hence, the total number of g (i, S)'s to be computed before computing g (1, V – {1}) is:

$$\sum_{k=0}^{n-1} (n-1)\binom{n-2}{k}$$

To calculate this sum, we use the binominal theorem:

$$(n-1)\left[\binom{n-2}{0}+\binom{n-2}{1}+\binom{n-2}{2}+ ---- +\binom{n-2}{n-2}\right]$$

According to the binominal theorem:

$$\left[\binom{n-2}{0}+\binom{n-2}{1}+\binom{n-2}{2}+ ---- +\binom{n-2}{n-2}\right] = 2^{n-2}$$

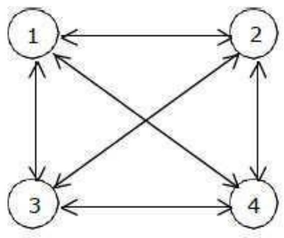
Therefore,

$$\sum_{k=0}^{n-1} (n-1)\binom{n-2}{k} = (n-1)\,2^{n-2}$$

This is Φ (n 2^{n-2}), so there are exponential number of calculate. Calculating oneg (i, S) require finding the minimum of at most n quantities. Therefore, the entire algorithm is Φ (n^2 2^{n-2}). This is better than enumerating all n! different tours to find the best one. So, we have traded on exponential growth for a much smaller exponential growth. The most serious drawback of this dynamic programming solution is the space needed, which is O (n 2^n). This is too large even for modest values of n.

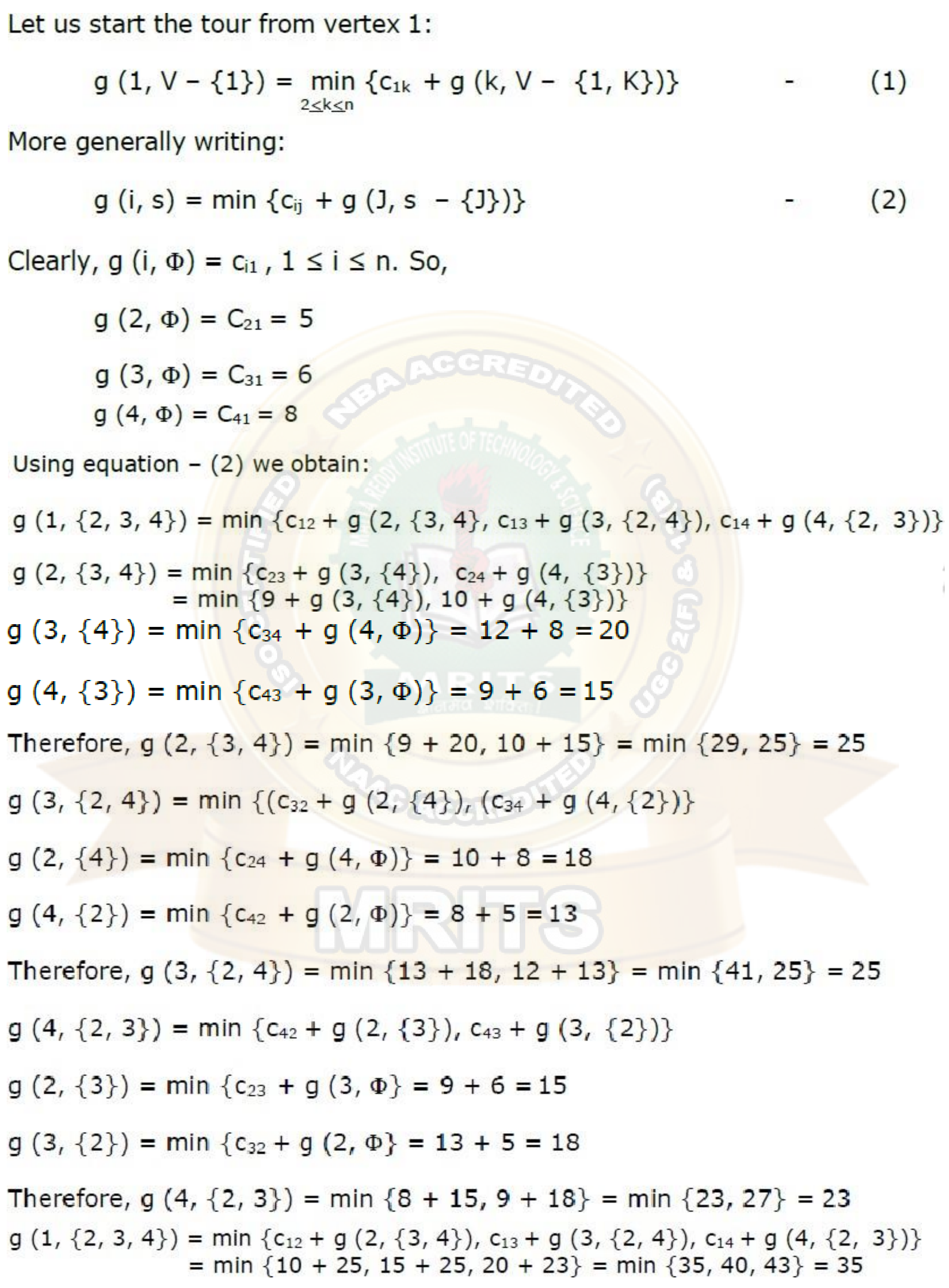
Example 1:

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix = $\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$

Let us start the tour from vertex 1:



Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, K\})\} \qquad - \qquad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\} \qquad - \qquad (2)$$

Clearly, $g(i, \Phi) = c_{i1}$, $1 \le i \le n$. So,

$g(2, \Phi) = C_{21} = 5$

$g(3, \Phi) = C_{31} = 6$

$g(4, \Phi) = C_{41} = 8$

Using equation – (2) we obtain:

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}, c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$

$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$
$= \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$

$g(3, \{4\}) = \min \{c_{34} + g(4, \Phi)\} = 12 + 8 = 20$

$g(4, \{3\}) = \min \{c_{43} + g(3, \Phi)\} = 9 + 6 = 15$

Therefore, $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

$g(3, \{2, 4\}) = \min \{(c_{32} + g(2, \{4\}), (c_{34} + g(4, \{2\})\}$

$g(2, \{4\}) = \min \{c_{24} + g(4, \Phi)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c_{42} + g(2, \Phi)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{41, 25\} = 25$

$g(4, \{2, 3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$g(2, \{3\}) = \min \{c_{23} + g(3, \Phi\} = 9 + 6 = 15$

$g(3, \{2\}) = \min \{c_{32} + g(2, \Phi\} = 13 + 5 = 18$

Therefore, $g(4, \{2, 3\}) = \min \{8 + 15, 9 + 18\} = \min \{23, 27\} = 23$

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$
$= \min \{10 + 25, 15 + 25, 20 + 23\} = \min \{35, 40, 43\} = 35$

The optimal tour for the graph has length =
35The optimal tour is: 1, 2, 4, 3, 1.

RELIABILITY DESIGN

The problem is to design a system that is composed of several devices connected in series. Let r_i be the reliability of device D_i (that is r_i is the probability that device i will function properly) then the reliability of the entire system is $\pi\ r_i$. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. For example, ifn = 10 and ri = 0.99, i <= i <= 10, then $\pi\ r_i$ = .904. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.

If stage i contains m_i copies of device D_i. Then the probability that all m_i have amalfunction is $(1 - r_i)^{mi}$.
Hence the reliability of stage i becomes $1 - (1 - r)^{mi}$.
The reliability of stage 'i' is given by a function ϕ_i i (m_i).
Our problem is to use device duplication. This maximization is to be carried out
under a cost constraint. Let ci be the cost of each unit of device i and let c be themaximum allowable cost of the system being designed.

We wish to solve:

Maximize $\displaystyle\pi_{1 \leq i \leq n} \phi_i\ (m_i)$

Subject to $\displaystyle\sum_{1 \leq i \leq n} C_i\ m_i < C$

$m_i \geq 1$ and interger, $1 \leq i \leq n$

Let f_i (x) represent the maximum value of $\displaystyle\pi_{1 \leq j \leq i} \phi\ (m_j)$

Subject to the constrains:

$\displaystyle\sum_{1 \leq J \leq i} C_J\ m_J \leq X$ and $1 \leq m_j \leq u_J$, $1 \leq j \leq i$

The last decision made requires one to choose m_n from $\{1, 2, 3, \ldots\ldots u_n\}$

Once a value of m_n has been chosen, the remaining decisions must be such as to use the remaining funds $C - C_n\ m_n$ in an optimal way.

The principle of optimality holds on

Let $f_i(x)$ represent the maximum value of $\prod_{1 \le j \le i} \phi(m_j)$

Subject to the constrains:

$$\sum_{1 \le j \le i} C_j m_j \le x \quad \text{and} \quad 1 \le m_j \le u_j,\ 1 \le j \le i$$

The last decision made requires one to choose m_n from $\{1, 2, 3, \ldots\ldots u_n\}$

Once a value of m_n has been chosen, the remaining decisions must be such as to use the remaining funds $C - C_n m_n$ in an optimal way.

The principle of optimality holds on

$$f_n(C) = \max_{1 \le m_n \le u_n} \{\phi_n(m_n)\, f_{n-1}(C - C_n m_n)\}$$

Assume each $C_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$, where

$$u_i = \left\lfloor \left(C + C_i - \sum_1^n C_j \right) \Big/ C_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \ge 1$

An optimal solution $m_1, m_2 \ldots\ldots m_n$ is the result of a sequence of decisions, one decision for each m_i.

Let $f_i(x)$ represent the maximum value of $\prod_{1 \le j \le i} \phi(m_j)$

Subject to the constrains:

$$\sum_{1 \le j \le i} C_j m_j \le x \quad \text{and} \quad 1 \le m_j \le u_j,\ 1 \le j \le i$$

The last decision made requires one to choose m_n from $\{1, 2, 3, \ldots\ldots u_n\}$

Once a value of m_n has been chosen, the remaining decisions must be such as to use the remaining funds $C - C_n m_n$ in an optimal way.

The principle of optimality holds on

$$f_n(C) = \max_{1 \le m_n \le u_n} \{\phi_n(m_n)\, f_{n-1}(C - C_n m_n)\}$$

for any $f_i\ (x_i)$, $i > 1$, this equation generalizes to

$$f_n(x) = \max_{1 \le m_i \le u_i} \{\phi_i(m_i)\, f_{i-1}(x - C_i\, m_i)\}$$

clearly, $f_0(x) = 1$ for all x, $0 \le x \le C$ and $f(x) = -\infty$ for all $x < 0$.

Let S^i consist of tuples of the form (f, x), where $f = f_i(x)$.

There is atmost one tuple for each different 'x', that result from a sequence of decisions on $m_1, m_2, \dots m_n$. The dominance rule (f_1, x_1) dominate (f_2, x_2) if $f_1 \ge f_2$ and $x_1 \le x_2$. Hence, dominated tuples can be discarded from S^i.

Example 1:

Design a three stage system with device types D_1, D_2 and D_3. The costs are \$30, \$15 and \$20 respectively. The Cost of the system is to be no more than \$105. The reliability of each device is 0.9, 0.8 and 0.5 respectively.

Solution:

We assume that if if stage I has mi devices of type i in parallel, then $\phi_i(m_i) = 1 - (1-r_i)^{mi}$

Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$. Where:

$$u_i = \left\lfloor \left(C + C_i - \sum_{1}^{n} C_J \right) \Big/ C_i \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3.

$$u_1 = \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3$$

$$u_3 = \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3$$

We use $S^i_j \rightarrow i{:}\textit{stage number and } J{:}\textit{ no. of devices in stage } i = m_i$

$S^o = \{f_o(x), x\}$ $\quad$ *initially* $f_o(x) = 1$ *and* $x = 0$, *so*, $S^o = \{1, 0\}$

Compute S^1, S^2 and S^3 as follows:

S^1 = depends on u_1 value, as $u_1 = 2$, so

$$S^1 = \{S^1_1, S^1_2\}$$

S^2 = depends on u_2 value, as $u_2 = 3$, so

$$S^2 = \{S^2_1, S^2_2, S^2_3\}$$

S^3 = depends on u_3 value, as u_3 = 3, so

$$S^3 = \{S^3_1, S^3_2, S^3_3\}$$

Now find $S^1_1 = \{(f(x), x)\}$

$f_1(x) = \{\phi_1(1) f_o(), \phi_1(2) f_o()\}$ With devices m_1 = 1 and m_2 = 2

Compute $\phi_1(1)$ and $\phi_1(2)$ using the formula: $\phi_i(mi)) = 1 - (1 - r_i)^{mi}$

$\phi_1(1) = 1 - (1 - r_1)^{m1} = 1 - (1 - 0.9)^1 = 0.9$

$\phi_1(2) = 1 - (1 - 0.9)^2 = 0.99$

$S^1_1 = \{f_1(x), x\} = = (0.9, 30)$

$S^1_2 = \{0.99, 30 + 30\} = (0.99, 60)$

Therefore, $S^1 = \{(0.9, 30), (0.99, 60)\}$

Next find $S^2_1 = \{(f_2(x), x)\}$

$f_2(x) = \{\phi_2(1) * f_1(), \phi_2(2) * f_1(), \phi_2(3) * f_1()\}$

$\phi_2(1) = 1 - (1 - r_1)_{mi} = 1 - (1 - 0.8)_1 = 1 - 0.2 = 0.8$

$\phi_2(2) = 1 - (1 - 0.8)^2 = 0.96$

$\phi_2(3) = 1 - (1 - 0.8)^3 = 0.992$

$S^2_1 = \{(0.8(0.9), 30 + 15), (0.8(0.99), 60 + 15)\} = \{(0.72, 45), (0.792, 75)\}$

$S^2_2 = \{(0.96(0.9), 30 + 15 + 15), (0.96(0.99), 60 + 15 + 15)\}$
$= \{(0.864, 60), (0.9504, 90)\}$

$S^2_3 = \{(0.992(0.9), 30 + 15 + 15 + 15), (0.992(0.99), 60 + 15 + 15 + 15)\}$
$= \{(0.8928, 75), (0.98208, 105)\}$

$$S^2 = \{S^2_1, S^2_2, S^2_3\}$$

By applying Dominance rule to S^2:
Therefore, $S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

Dominance Rule:

If S^i contains two pairs (f_1, x_1) and (f_2, x_2) with the property that $f_1 \geq f_2$ and $x_1 \leq x_2$, then (f_1, x_1) dominates (f_2, x_2), hence by dominance rule (f_2, x_2) can be discarded. Discarding or pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^i and Dominated tuples has to be discarded from S^i.

Case 1: if $f_1 \leq f_2$ and $x_1 > x_2$ then discard (f_1, x_1)

Case 2: if $f_1 \geq f_2$ and $x_1 < x_2$ the discard (f_2, x_2)

Case 3: otherwise simply write (f_1, x_1)

$S_2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

$\phi_3(1) = 1 - (1 - r_I)^{mi} = 1 - (1 - 0.5)^1 = 1 - 0.5 = 0.5$

$\phi_3(2) = 1 - (1 - 0.5)^2 = 0.75$

$\phi_3(3) = 1 - (1 - 0.5)^3 = 0.875$

$S_1^3 = \{(0.5\,(0.72), 45 + 20), (0.5\,(0.864), 60 + 20), (0.5\,(0.8928), 75 + 20)\}$

$S_1^3 = \{(0.36, 65), (0.437, 80), (0.4464, 95)\}$

$S_2^3 = \{(0.75\,(0.72), 45 + 20 + 20), (0.75\,(0.864), 60 + 20 + 20),$
$(0.75\,(0.8928), 75 + 20 + 20)\}$

$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$

$S_3^3 = \{\ (0.875\,(0.72), 45 + 20 + 20 + 20), (0.875\,(0.864), 60 + 20 + 20 + 20),$
$(0.875\,(0.8928), 75 + 20 + 20 + 20)\ \}$

$S_3^3 = \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$

If cost exceeds 105, remove that tuples

$S^3 = \{(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)\}$

The best design has a reliability of 0.648 and a cost of 100. Tracing back for the solution through S^i's we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

Other Solution:

According to the principle of optimality:

$$f_n(C) = \max_{1 \leq m_n \leq u_n} \{\phi_n(m_n).\ f_{n-1}(C - C_n m_n)\} \text{ with } f_o(x) = 1 \text{ and } 0 \leq x \leq C;$$

Since, we can assume each $c_i > 0$, each mi must be in the range $1 \leq m_i \leq u_i$. Where:

$$u_i = \left\lfloor \left(C + C_i - \sum_{i}^{n} C_J \right) / C_i \right\rfloor$$

Using the above equation compute u_1, u_2 and u_3.

$$u_1 = \frac{105 + 30 - (30 + 15 + 20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105 + 15 - (30 + 15 + 20)}{15} = \frac{55}{15} = 3$$

$$u_3 = \frac{105 + 20 - (30 + 15 + 20)}{20} = \frac{60}{20} = 3$$

$$f_3(105) = \max_{1 \le m3 \le u3} \{\phi_3(m_3).\ f_2(105 - 20m_3)\}$$

$$= \max \{\phi_3(1)\ f_2(105 - 20),\ \underline{\phi_3(2)\ f_2(105 - 20x2)},\ \phi_3(3)\ f_2(105 - 20x3)\}$$

$$f_2\ (85) = \max_{1 \le m2 \le u2} \{\phi_2(m_2).\ f_1(85 - 15m_2)\}$$

$$= \max \{\phi_2(1).f_1(85 - 15),\ \phi_2(2).f_1(85 - 15x2),\ \phi_2(3).f_1(85 - 15x3)\}$$

$$= \max \{0.8\ f_1(70),\ 0.96\ f_1(55),\ 0.992\ f_1(40)\}$$

$$= \max \{0.8 \times 0.99,\ 0.96 \times 0.9,\ 0.99 \times 0.9\} = 0.8928$$

$$f_1\ (70) = \max_{1 \le m1 \le u1} \{\phi_1(m_1).\ f_0(70 - 30m_1)\}$$

$$= \max \{\phi_1(1)\ f_0(70 - 30),\ \phi_1(2)\ f_0(70 - 30x2)\}$$

$$= \max \{\phi_1(1) \times 1,\ \phi_1(2) \times 1\} = \max \{0.9,\ 0.99\} = 0.99$$

$$f_1\ (55) = \max_{1 \le m1 \le u1} \{\phi_1(m_1).\ f_0(55 - 30m_1)\}$$

$$= \max \{\phi_1(1)\ f_0(50 - 30),\ \phi_1(2)\ f_0(50 - 30x2)\}$$

$$= \max \{\phi_1(1) \times 1,\ \phi_1(2) \times -\infty\} = \max \{0.9,\ -\infty\} = 0.9$$

$$f_1\ (40) = \max_{1 \le m1 \le u1} \{\phi_1(m_1).\ f_0(40 - 30m_1)\}$$

$$= \max \{\phi_1(1)\ f_0(40 - 30),\ \phi_1(2)\ f_0(40 - 30x2)\}$$

$$= \max \{\phi_1(1) \times 1,\ \phi_1(2) \times -\infty\} = \max\{0.9,\ -\infty\} = 0.9$$

$$f_2\ (65) = \max_{1 \le m2 \le u2} \{\phi_2(m_2).\ f_1(65 - 15m_2)\}$$

$$= \max \{\phi_2(1)\ f_1(65 - 15),\ \underline{\phi_2(2)\ f_1(65 - 15x2)},\ \phi_2(3)\ f_1(65 - 15x3)\}$$

$$= \max \{0.8\ \ f_1(50),\ 0.96\ \ f_1(35),\ 0.992\ \ f_1(20)\}$$

$$= \max \{0.8 \times 0.9,\ 0.96 \times 0.9,\ -\infty\} = 0.864$$

$$f_1\ (50) = \max_{1 \le m1 \le u1} \{\phi_1(m_1).\ f_0(50 - 30m_1)\}$$

$$= \max \{\phi_1(1)\ f_0(50 - 30),\ \phi_1(2)\ f_0(50 - 30x2)\}$$

$$= \max \{\phi_1(1) \times 1,\ \phi_1(2) \times -\infty\} = \max\{0.9, -\infty\} = 0.9$$

$$f_1\ (35) = \max_{1 \le m1 \le u1}\ \phi_1(m_1).\ f_0(35 - 30m_1)\}$$

$$= \max \{\underline{\phi_1(1).f_0(35-30)},\ \phi_1(2).f_0(35-30x2)\}$$

$$= \max \{\phi_1(1) \times 1,\ \phi_1(2) \times -\infty\} = \max\{0.9, -\infty\} = 0.9$$

$$f_1\ (20) = \max_{1 \le m1 \le u1} \{\phi_1(m_1).\ f_0(20 - 30m_1)\}$$

$$= \max \{\phi_1(1)\ f_0(20 - 30),\ \phi_1(2)\ f_0(20 - 30x2)\}$$

$$= \max \{\phi_1(1) \times -\infty,\ \phi_1(2) \times -\infty\} = \max\{-\infty, -\infty\} = -\infty$$

$$f_2\ (45) = \max_{1 \le m2 \le u2} \{\phi_2(m_2).\ f_1(45 - 15m_2)\}$$

$$= \max \{\phi_2(1)\ f_1(45 - 15),\ \phi_2(2)\ f_1(45 - 15x2),\ \phi_2(3)\ f_1(45 - 15x3)\}$$

$$= \max \{0.8\ f_1(30),\ 0.96\ f_1(15),\ 0.992\ f_1(0)\}$$

$$= \max \{0.8 \times 0.9,\ 0.96 \times -\infty,\ 0.99 \times -\infty\} = 0.72$$

$$f_1\ (30) = \max_{1 \le m1 \le u1} \{\phi_1(m_1).\ f_0(30 - 30m_1)\}$$

$$= \max \{\phi_1(1)\ f_0(30 - 30),\ \phi_1(2)\ f_0(30 - 30x2)\}$$

$$= \max \{\phi_1(1) \times 1,\ \phi_1(2) \times -\infty\} = \max\{0.9, -\infty\} = 0.9$$

Similarly, $f_1\,(15) = -\infty$, $f_1\,(0) = -\infty$.

The best design has a reliability = 0.648 and

Cost = 30 x 1 + 15 x 2 + 20 x 2 = 100.

Tracing back for the solution through S^i's we can determine that:

$m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

UNIT-IV
Greedy method: General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

General method:

The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results. Let's understand through some terms.

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Advantages

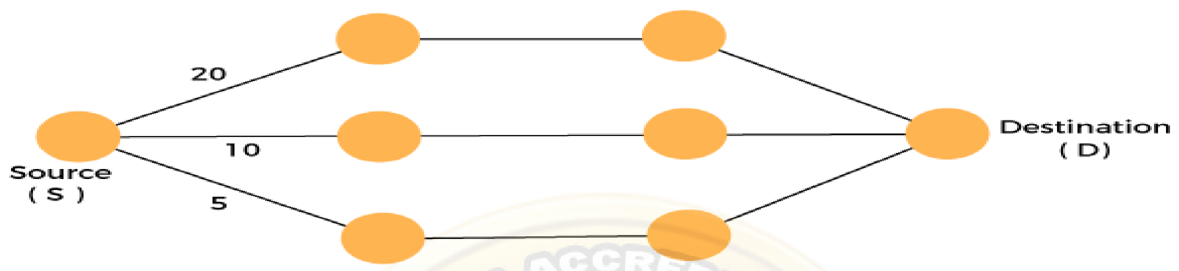
- It is easy to implement.
- Has fewer time complexities.
- Can be used for the purpose of optimization or finding close to optimization in the case of NP-Hard problems.

Disadvantages of using Greedy algorithm

Greedy algorithm makes decisions based on the information available at each phase without considering the broader problem. So, there might be a possibility that the greedy solution does not give the best solution for every problem.

It follows the local optimum choice at each stage with a intend of finding the global optimum. Let's understand through an example.

Consider the graph which is given below:



We have to travel from the source to the destination at the minimum cost. Since we have three feasible solutions having cost paths as 10, 20, and 5. 5 is the minimum cost path so it is the optimal solution. This is the local optimum, and in this way, we find the local optimum at each stage in order to calculate the global optimal solution.

Characteristics of Greedy approach

1. The greedy approach consists of an ordered list of resources (profit, cost, value, etc.)
2. The greedy approach takes the maximum of all the resources (max profit, max value, etc.)
3. For example, in the case of the fractional knapsack problem, the maximum value/weight is taken first based on the available capacity.

Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

Job Sequencing With Deadlines

Here is the process of Job sequencing in brief.

- Firstly, you are given a set of jobs.
- Each job has a set of defined deadlines and some profit associated with it.
- A job is profited only if that job is completed within the given deadline.
- Another point to note is that only one processor will be available for processing all the jobs.
- The processor will take one unit of time in order to complete a job.

The problem states-

Approach to Solution

- A feasible solution is a subset of jobs such that each job of the subset is completed within the given deadline.
- The value of a feasible solution is said to be the sum of the profit of all the jobs contained in that subset.
- An optimal solution to the problem would be a feasible solution that gives the maximum profit.

Greedy Algorithm Approach-

We adopt the greedy algorithm inorder to determine the selection of the next job to get an optimal solution.

Below is the greedy algorithm that is always supposed to give an optimal solution to the job sequencing problem.

Step-01:

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

Step-02:

- Checking the value of the maximum deadline.
- Drawing a Gantt chart such that the maximum time on the Gantt chart is the value of the maximum deadline.

Step-03:

- Picking up the jobs one after the other.

- Adding the jobs on the Gantt chart in such a way that they are as far as possible from 0. This ensures that the job gets completed before the given deadline.

Algorithm for Job Sequencing with Deadline:

Algorithm: Job-Sequencing-With-Deadline (Dead, Job, n, k)

```
Dead(0) := Job(0) := 0
k := 1
Job(1) := 1 // means first job is selected
for i = 2 … n do
  r := k
  while Dead(Job(r)) > Dead(i) and Dead(Job(r)) ≠ r do
    r := r – 1
  if Dead(Job(r)) ≤ Dead(i) and Dead(i) > r then
    for l = k … r + 1 by -1 do
      Job(l + 1) := Job(l)
      Job(r + 1) := i
      k := k + 1
```

PROBLEM BASED ON JOB SEQUENCING WITH DEADLINES-

Problem-

We are given the jobs, their deadlines and associated profits as shown-

Jobs	**J1**	**J2**	**J3**	**J4**	**J5**	**J6**
Deadlines	5	3	3	2	4	2
Profits	201	181	191	301	121	101

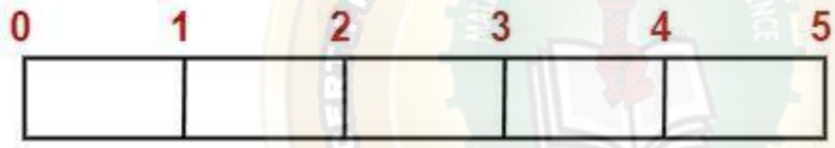
Answer the following questions-

1. Write the optimal schedule that provides us the maximum profit.
2. Can we complete all the jobs in the optimal schedule?
3. What is the maximum earned profit?

Solution:
Step-01:
Firstly, we need to sort all the given jobs in decreasing order of their profit as follows.

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

Step-02:
For each step, we calculate the value of the maximum deadline.
Here, the value of the maximum deadline is 5.
So, we draw a Gantt chart as follows and assign it with a maximum time on the Gantt chart with 5 units as shown below.



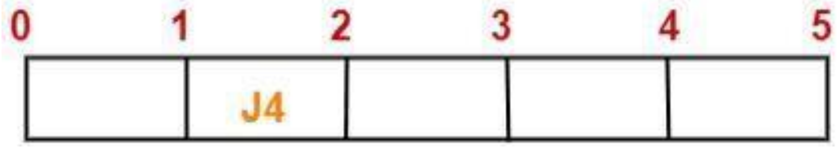
Gantt Chart

Now,

- We will be considering each job one by one in the same order as they appear in the Step-01.
- We are then supposed to place the jobs on the Gantt chart as far as possible from 0.

Step-03:

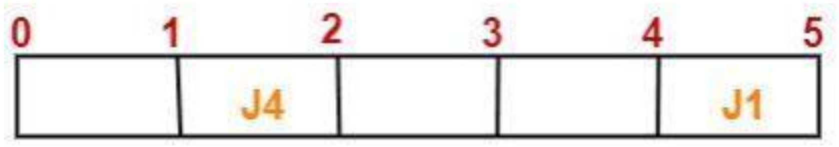
- We now consider job4.
- Since the deadline for job4 is 2, we will be placing it in the first empty cell before deadline 2 as follows.



Step-04:

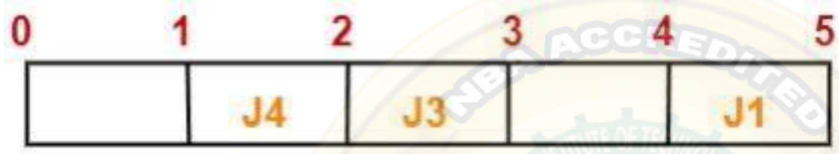
- Now, we go with job1.

- Since the deadline for job1 is 5, we will be placing it in the first empty cell before deadline 5 as shown below.



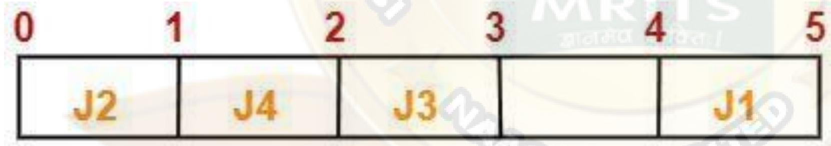
Step-05:

- We now consider job3.
- Since the deadline for job3 is 3, we will be placing it in the first empty cell before deadline 3 as shown in the following figure.



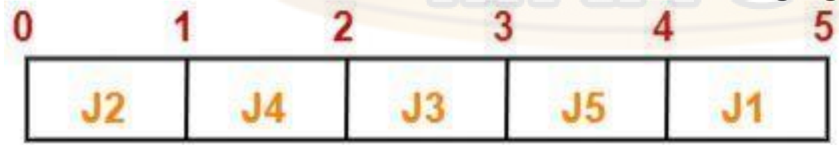
Step-06:

- Next, we go with job2.
- Since the deadline for job2 is 3, we will be placing it in the first empty cell before deadline 3.
- Since the second cell and third cell are already filled, so we place job2 in the first cell as shown below.



Step-07:

- Now, we consider job5.
- Since the deadline for job5 is 4, we will be placing it in the first empty cell before deadline 4 as shown in the following figure.



Now,

- We can observe that the only job left is job6 whose deadline is 2.
- Since all the slots before deadline 2 are already occupied, job6 cannot be completed.

Now, the questions given above can be answered as follows:

Part-01:
The optimal schedule is-
Job2, Job4, Job3, Job5, Job1
In order to obtain the maximum profit this is the required order in which the jobs must be completed.
Part-02:

- As we can observe, all jobs are not completed on the optimal schedule.
- This is because job6 was not completed within the given deadline.

Part-03:
Maximum earned profit = Sum of the profit of all the jobs from the optimal schedule
= Profit of job2 + Profit of job4 + Profit of job3 + Profit of job5 + Profit of job1
= 181 + 301 + 191 + 121 + 201
= 995 units

Analysis of the algorithm:
In the job sequencing with deadlines algorithm, we make use of two loops, one loop within another. Hence, the complexity of this algorithm would be O(n2).

Knapsack Problem

When given a set of items, where each item has a weight and a value, we need to determine a subset of items that are to be included in a collectionin such a way that the total weight aggregates up to be lower than or equalto a given limit and the total value could be as big as possible.

The Knapsack problem is an instance of a Combinatorial Optimization problem. One general approach to crack difficult problems is to identify the most restrictive constraint. For this, we must ignore the others and solve a knapsack problem, and finally, we must somehow fit the solutionto satisfy the constraints that are ignored.

Applications
For multiple cases of resource allocation problems that have some specific constraints, the problem can be solved in a way that is similar to the Knapsack problem. Following are a set of examples.

- Finding the least wasteful way to cut down the basic materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

Consider a problem scenario where a thief is robbing a store and his knapsack (bag) can carry a maximal weight of W. Consider that there aren items in the store and the weight of the ith item is wi and its respectiveprofit is pi.

What are all the items the thief should take?

Here, the main goal/objective of the thief is to maximize the profit anyhow. So, the items should opt-in such a way that the items which are carried bythe thief will fetch the maximum profit.

Based on the nature of the items, Knapsack problems are classified into two categories

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this category, items can be broken into smaller pieces, and the thief can select fractions of items.

According to the problem scenario,

- There are n items in the store
- Weight of ith item
- wi>0
- Profit for ith item
- pi>0 and
- The capacity of the Knapsack is W

As the name suggests, in the Knapsack problem, items can be broken into smaller fragments. So, the thief might only take a fraction or a part of *xi* of ith item.

0⩽xi⩽1

The ith item in the store contributes a weight of xi.wi to the total weight in the knapsack(bag) and profit xi.pi to the Total Profit.

Hence, the main objective of the algorithm is basically to maximize the value of ∑n=1n(xi.pi) with respect to the given constraint,

∑n=1n(xi. wi)⩽W

We already know that a solution that is said to be an optimal solution must fill the knapsack(bag) exactly, if not, we could at least add a smaller fraction of one of the remaining items. This will result in an increase in the overall profit.

Thus, an optimal solution to this problem can be obtained by,

∑n=1n(xi.wi)=W

Now, we have to sort all those items based on their values of piwi, so that

pi+1wi+1 ≤ piwi

Here, *x* is an array that is used to store the fraction of items.

Analysis

Suppose that we are provided with items that have already been sorted in the decreasing order of piwi, then the time taken by the “while” will be O(n). So, the total time including that includes even sorting will be O(n logn).

Example

Let us consider that the capacity of the knapsack(bag) *W* = 60 and the list of items are shown in the following table −

Item	**A**	**B**	**C**	**D**
Profit	281	101	121	121

Weight	40	10	20	24
Ratio (piwi)	7	10	6	5

We can see that the provided items are not sorted based on the valueof piwi, we perform sorting. After sorting, the items are shown in the following table.

Item	**B**	**A**	**C**	**D**
Profit	101	281	121	121
Weight	10	40	20	24
Ratio (piwi)	10	7	6	5

Solution

Once we sort all the items according to the piwi, we choose all of B as the weight of B is less compared to that of the capacity of the knapsack. Further, we choose item A, as the available capacity of the knapsack is greater than the weight of A. Now, we will choose C as the next item. Anyhow, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of the chosen item – C.

Hence, a fraction of C (i.e. $(60 - 50)/20$) is chosen.

Minimum cost spanning trees:

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

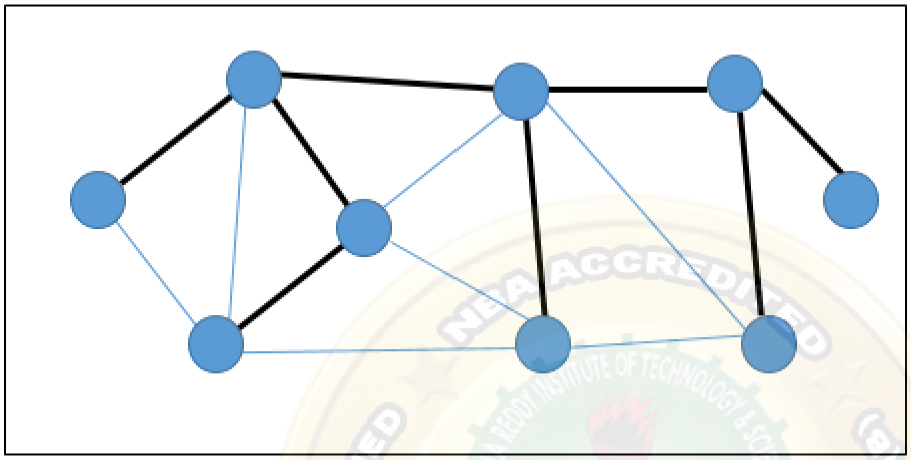
If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

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

Example

In the following graph, the highlighted edges form a spanning tree.

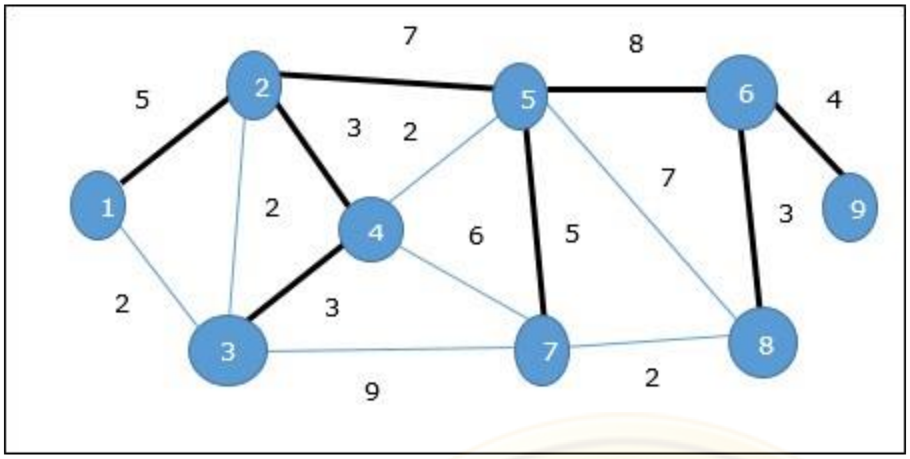


Minimum Spanning Tree

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. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one 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. In this context, if 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.



In the above graph, we have 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.

We will use Prim's algorithm to find the minimum spanning tree.

Prim's Algorithm

Prim's algorithm is a greedy approach to find the minimum spanning tree. In this algorithm, to form a MST we can start from an arbitrary vertex.

```
Algorithm: MST-Prim's (G, w, r)
for each u ϵ G.V
u.key = ∞
u.∏ = NIL
r.key = 0
Q = G.V
while Q ≠ Φ
u = Extract-Min (Q)
for each v ϵ G.adj[u]
if each v ϵ Q and w(u, v) < v.key
v.∏ = u
v.key = w(u, v)
```

The function Extract-Min returns the vertex with minimum edge cost. This function works on min-heap.

Example

Using Prim's algorithm, we can start from any vertex, let us start from vertex **1**.

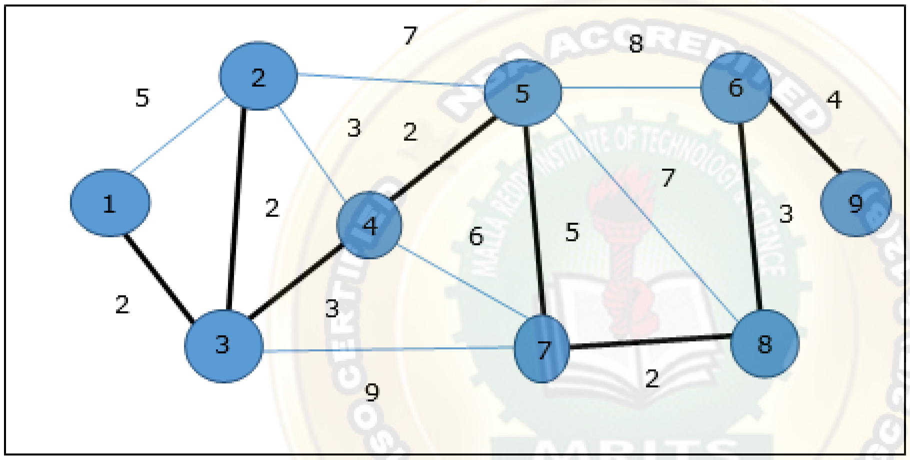
Vertex **3** is connected to vertex **1** with minimum edge cost, hence edge **(1, 2)** is added to the spanning tree.

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

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**.



Some of the properties of the spanning tree are listed below:

- A connected graph can have more than one spanning trees.
- All spanning trees in a graph have the same number of nodes and edges.
- If we remove one edge from the spanning tree, then it will become **minimally connected** and will make the graphdisconnected.
- On the other hand, adding one edge to the spanning tree will make it **maximally acyclic** thereby creating a loop.
- A spanning tree does not have a loop or a cycle.

What Is A Minimum Spanning Tree (MST)

A minimum spanning tree is the one that contains the least weight among all the other spanning trees of a connected weighted graph. There can be more than one minimum spanning tree for a graph.

There are two most popular algorithms that are used to find the minimum spanning tree in a graph.
They include:

- Kruskal’s algorithm
- Prim’s algorithm

Kruskal’s Algorithm

Kruskal’s algorithm is an algorithm to find the MST in a connected graph.

Kruskal’s algorithm finds a subset of a graph G such that:

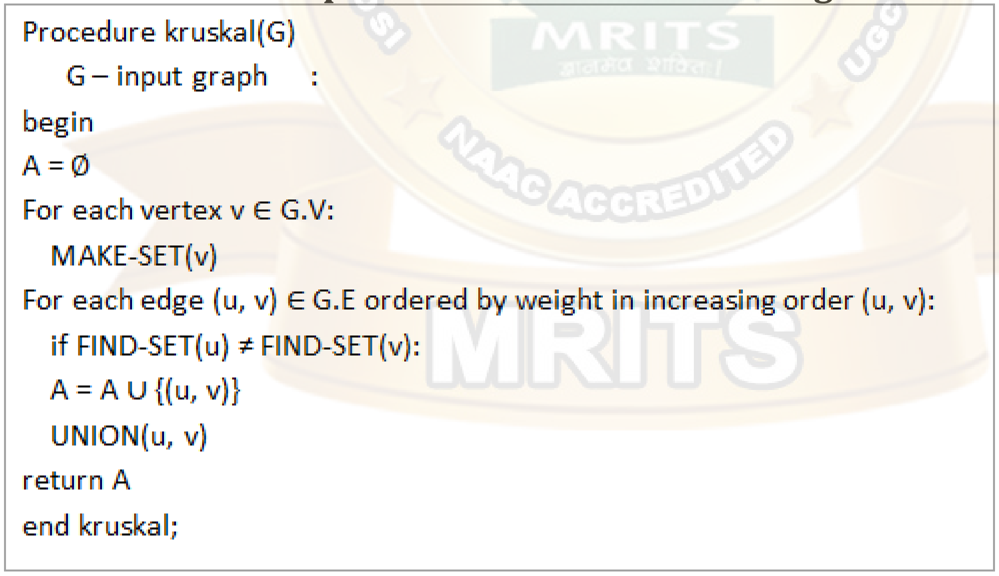
- It forms a tree with every vertex in it.
- The sum of the weights is the minimum among all the spanning trees that can be formed from this graph.

The sequence of steps for Kruskal’s algorithm is given as follows:

1. First sort all the edges from the lowest weight to highest.
2. Take edge with the lowest weight and add it to the spanning tree. If the cycle is created, discard the edge.
3. Keep adding edges like in step 1 until all the vertices are considered.

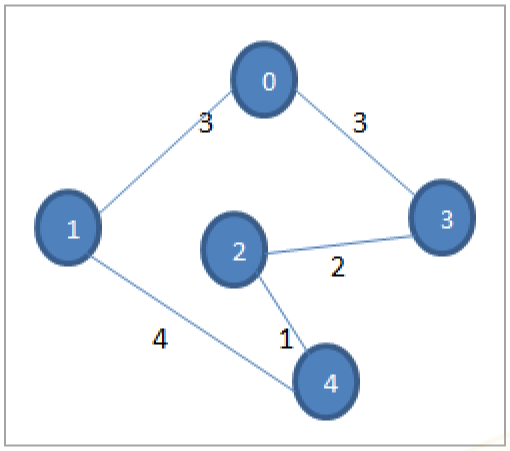
Pseudocode for Kruskal’s Algorithm

Given below is the pseudo-code for Kruskal’s Algorithm

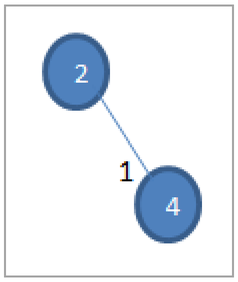


```
Procedure kruskal(G)
    G – input graph    :
begin
A = Ø
For each vertex v ∈ G.V:
   MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by weight in increasing order (u, v):
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A ∪ {(u, v)}
   UNION(u, v)
return A
end kruskal;
```

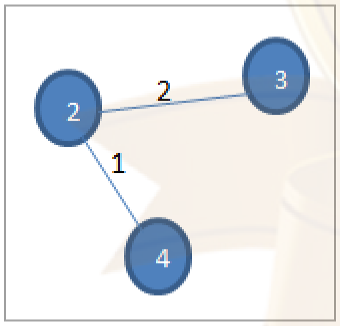
Now let us see the illustration of Kruskal’s algorithm.



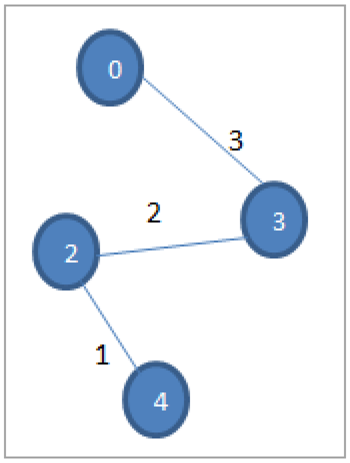
Now we choose the edge with the least weight which is 2-4.



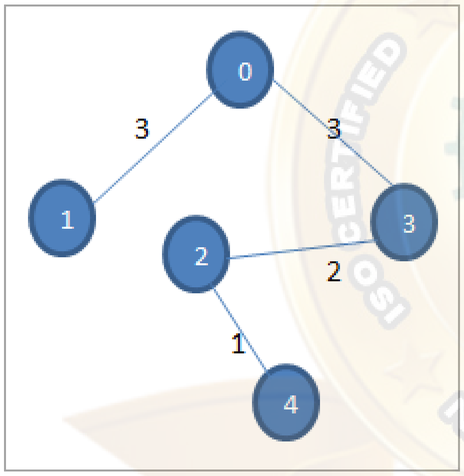
Next, choose the next shortest edge 2-3.



Then we choose next edge with the shortest edge and that does not create a cycle i.e. 0-3



The next step is to choose the shortest edge so that it doesn't form a cycle. This is 0-1.



Prim's Algorithm

Prim's algorithm is yet another algorithm to find the minimum spanning the tree of a graph. In contrast to Kruskal's algorithm that starts with graph edges, Prim's algorithm starts with a vertex. We start with one vertex and keep on adding edges with the least weight till all the vertices are covered.

The sequence of steps for Prim's Algorithm is as follows:

1. Choose a random vertex as starting vertex and initialize a minimum spanning tree.

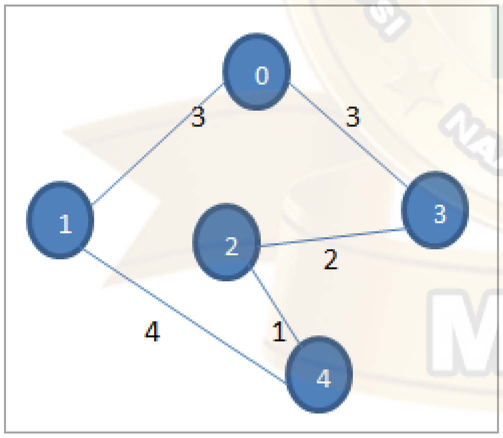
2. Find the edges that connect to other vertices. Find the edge with minimum weight and add it to the spanning tree.
3. Repeat step 2 until the spanning tree is obtained.

Pseudocode for Prim's Algorithm

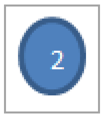
```
Procedure prims
        G – input graph
        U – random vertex
        V – vertices in graph G
begin
        T = Ø;
        U = { 1 };
        while (U ≠ V)
          let (u, v) be the least cost edge such that u ∈ U and v ∈ V - U;
          T = T U {(u, v)}
          U = U U {v}
end procedure
```

Now let us see an illustration for Prim's algorithm.

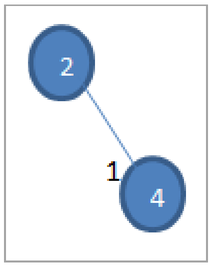
For this, we are using the same example graph that we used in the Illustration of Kruskal's algorithm.



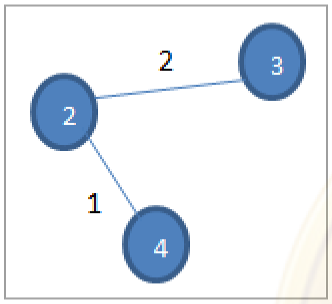
Let us select node 2 as the random vertex.



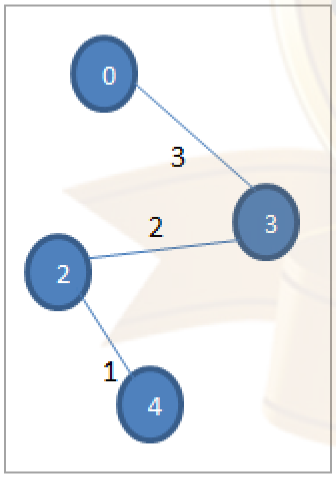
Next, we select the edge with the least weight from 2. We choose edge 2-4.



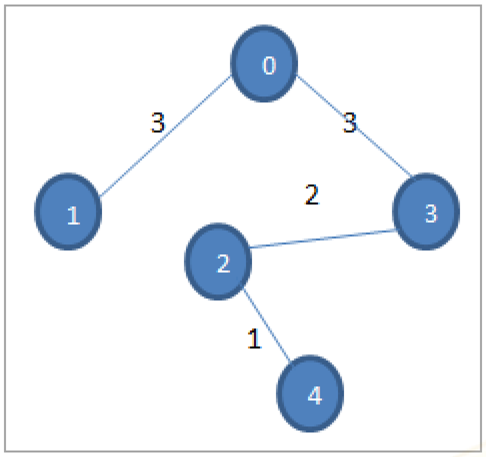
Next, we choose another vertex that is not in the spanning tree yet. We choose the edge 2-3.



Now let us select an edge with least weight from the above vertices. We have edge 3-0 which has the least weight.



Next, we choose an edge with the least weight from vertex 0. This is the edge 0-1.



From the above figure, we see that we have now covered all the vertices in the graph and obtained a complete spanning tree with minimum cost.

Single source shortest path problem:

The single source shortest path algorithm (for arbitrary weight positive or negative) is also known Bellman-Ford algorithm is used to find minimum distance from source vertex to any other vertex. The main difference between this algorithm with Dijkstra's algorithm is, in Dijkstra's algorithm we cannot handle the negative weight, but here we can handle it easily.

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph $G = (V, E)$, where all the edges are non-negative (i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$).

In the following algorithm, we will use one function ***Extract-Min()***, which extracts the node with the smallest key.

Algorithm: Dijkstra's-Algorithm (G, w, s)

```
for each vertex v Є G.V
v.d := ∞
v.∏ := NIL
s.d := 0
S :=  Φ
Q := G.V
while Q ≠ Φ
u := Extract-Min (Q)
S := S U {u}
for each vertex v Є G.adj[u]
```

```
if v.d > u.d + w(u, v)
v.d := u.d + w(u, v)
v.∏ := u
```

Analysis

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is ***$O(V^2 + E)$***.

In this algorithm, if we use min-heap on which ***Extract-Min()*** function works to return the node from ***Q*** with the smallest key, the complexity of this algorithm can be reduced further.

Example

Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

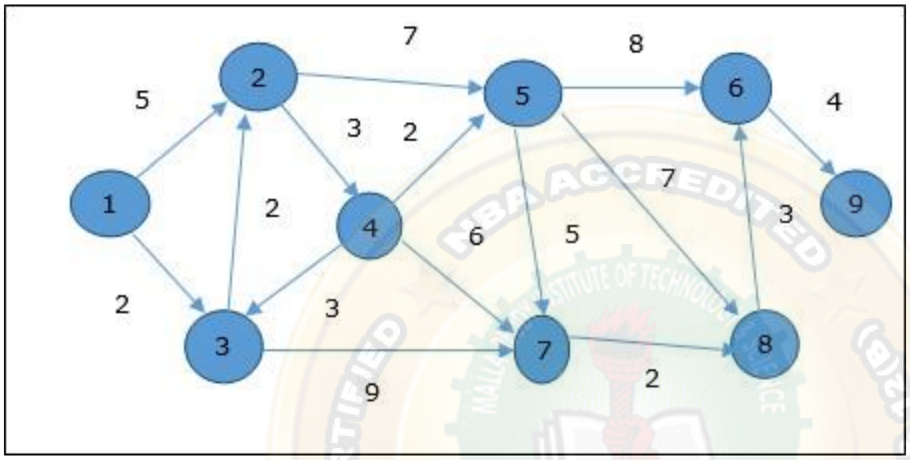
Vertex	**Initial**	**Step1 V_1**	**Step2 V_3**	**Step3 V_2**	**Step4 V_4**	**Step5 V_5**	**Step6 V_7**	**Step7 V_8**	**Step8 V_6**
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13

9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is

1→ 3→ 7→ 8→ 6→ 9

This path is determined based on predecessor information.



Bellman Ford Algorithm

This algorithm solves the single source shortest path problem of a directed graph ***G = (V, E)*** in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.

```
Algorithm: Bellman-Ford-Algorithm (G, w, s)
for each vertex v Є G.V
v.d := ∞
v.∏ := NIL
s.d := 0
for i = 1 to |G.V| - 1
for each edge (u, v) Є G.E
if v.d > u.d + w(u, v)
v.d := u.d +w(u, v)
v.∏ := u
for each edge (u, v) Є G.E
if v.d > u.d + w(u, v)
return FALSE
return TRUE
```

Analysis

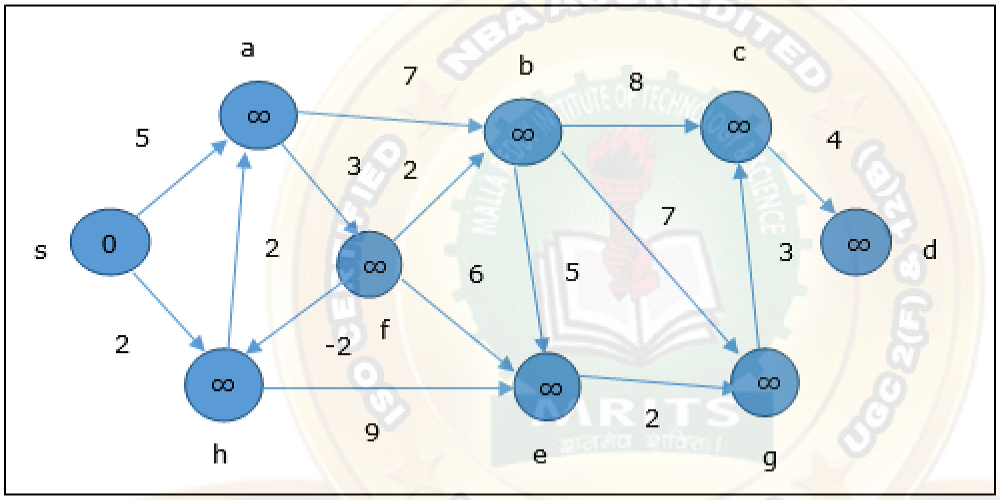
The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs |***V - 1***| passes over the edges, which takes ***O(E)*** times.

Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.

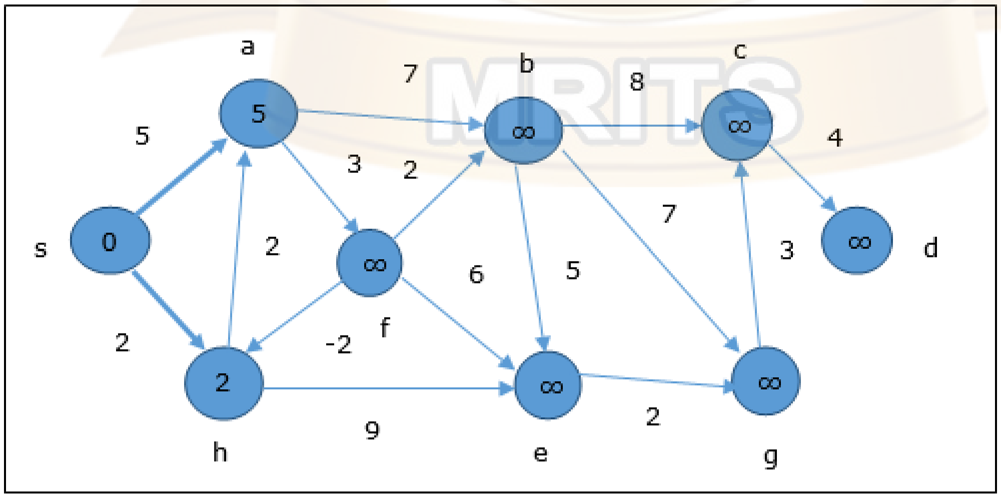
Example

The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.

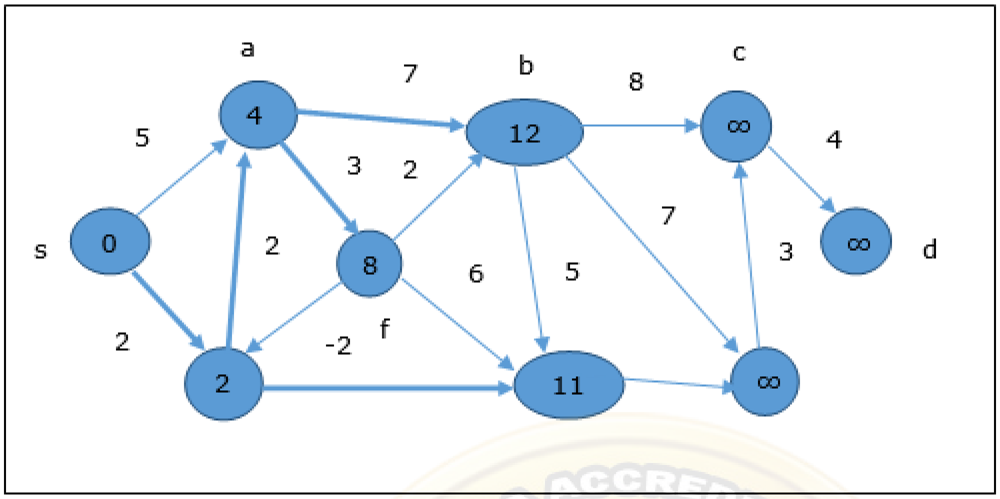
At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.



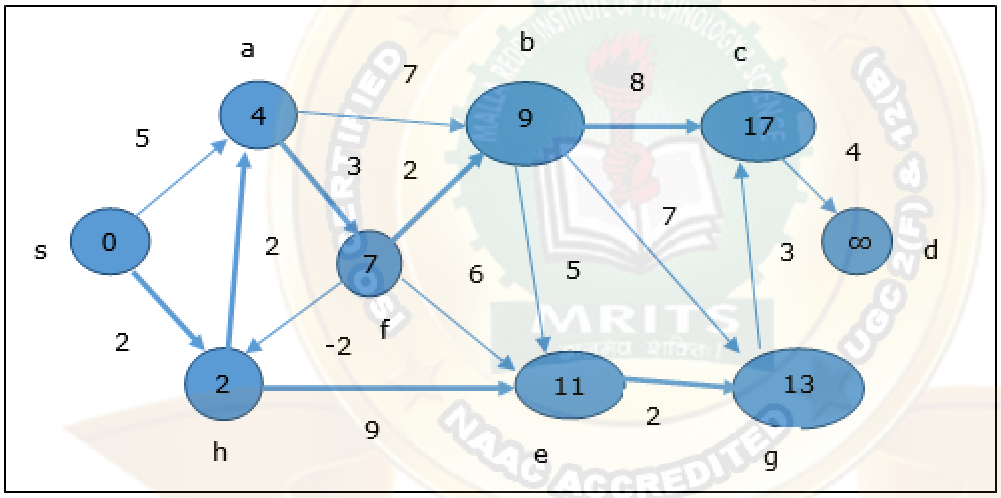
In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated.



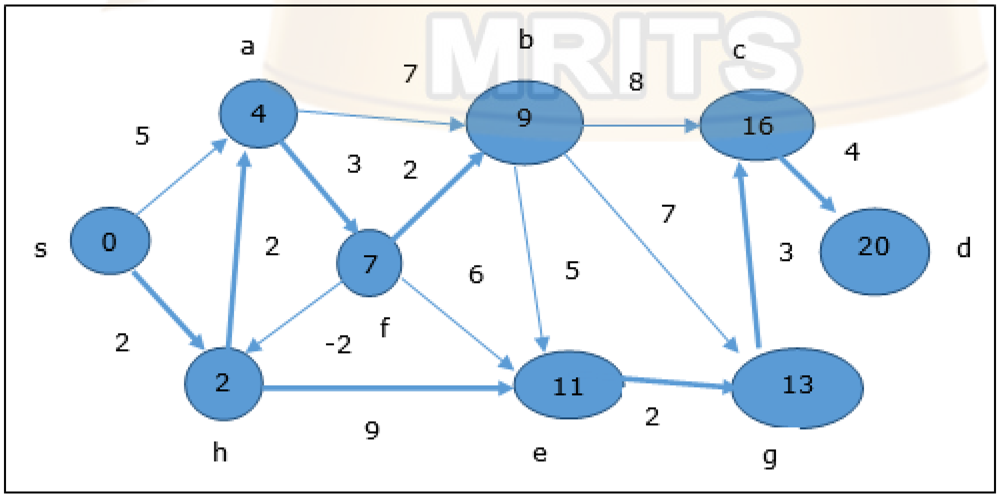
In the next step, vertices ***a, b, f*** and ***e*** are updated.



Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



Here, vertices ***c*** and ***d*** are updated.



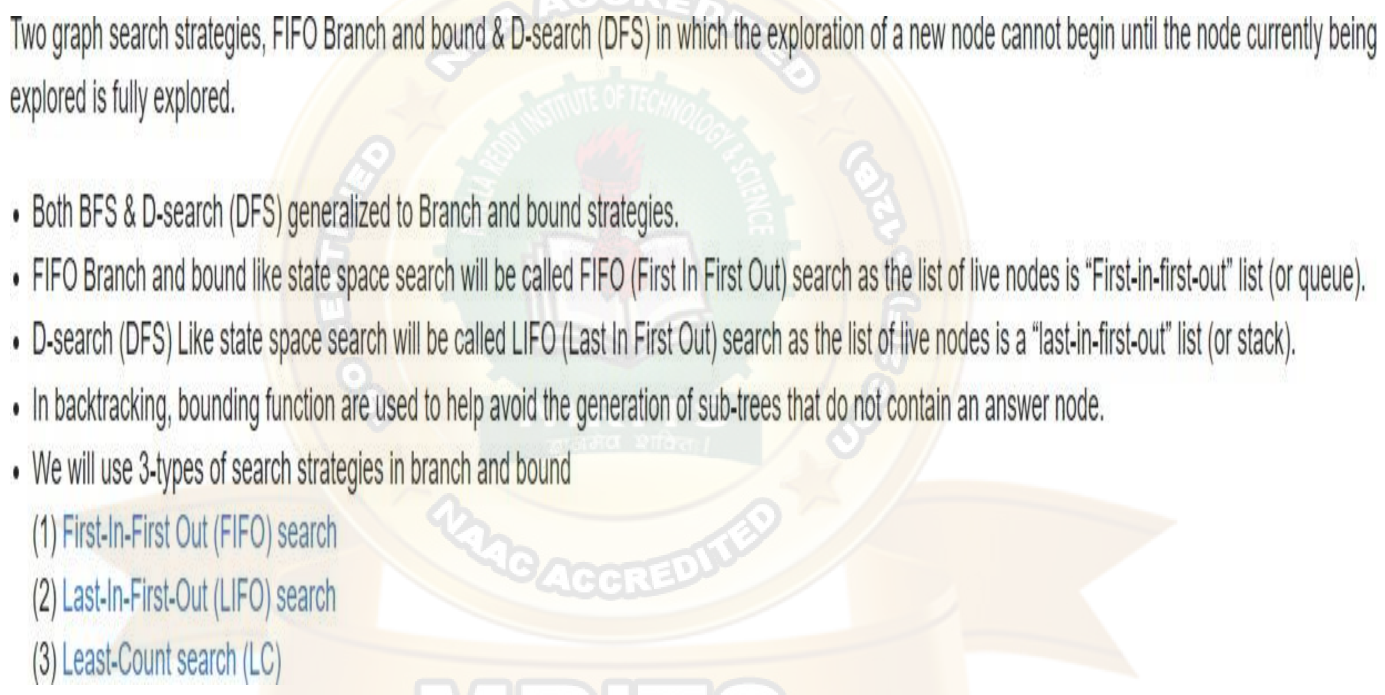
Hence, the minimum distance between vertex **s** and vertex **d** is **20**.

Based on the predecessor information, the path is s→ h→ e→ g→ c→ d

Branch and bound

What is Branch and bound?

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.



Two graph search strategies, FIFO Branch and bound & D-search (DFS) in which the exploration of a new node cannot begin until the node currently being explored is fully explored.

- Both BFS & D-search (DFS) generalized to Branch and bound strategies.
- FIFO Branch and bound like state space search will be called FIFO (First In First Out) search as the list of live nodes is “First-in-first-out” list (or queue).
- D-search (DFS) Like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a “last-in-first-out” list (or stack).
- In backtracking, bounding function are used to help avoid the generation of sub-trees that do not contain an answer node.
- We will use 3-types of search strategies in branch and bound

  (1) First-In-First Out (FIFO) search

  (2) Last-In-First-Out (LIFO) search

  (3) Least-Count search (LC)

Let's understand through an example.

Jobs = {j1, j2, j3, j4}

P = {10, 5, 8, 3}

d = {1, 2, 1, 2}

The above are jobs, problems and problems given. We can write the solutions in two ways which are given below:

Suppose we want to perform the jobs j1 and j2 then the solution can be represented in two ways:

The first way of representing the solutions is the subsets of jobs.

S1 = {j1, j4}

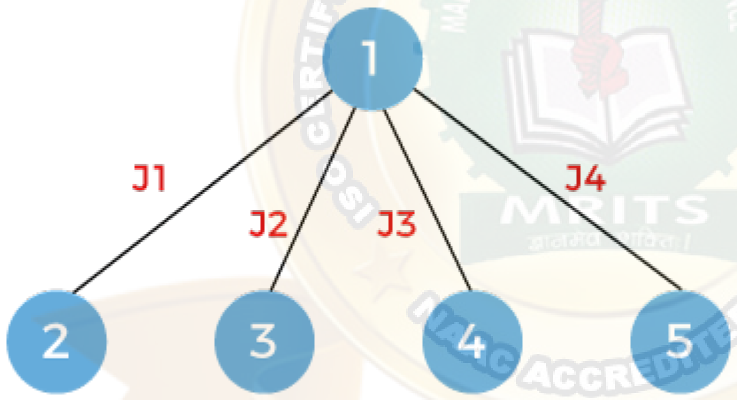
The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.

S2 = {1, 0, 0, 1}

The solution s1 is the variable-size solution while the solution s2 is the fixed-size solution.

First, we will see the subset method where we will see the variable size.

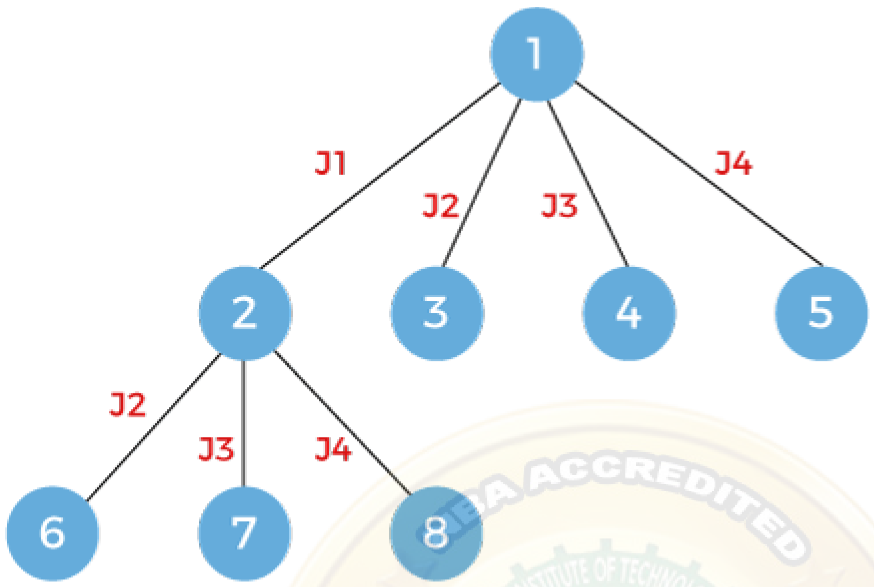
First method: (First in First out (FIFO)Search)



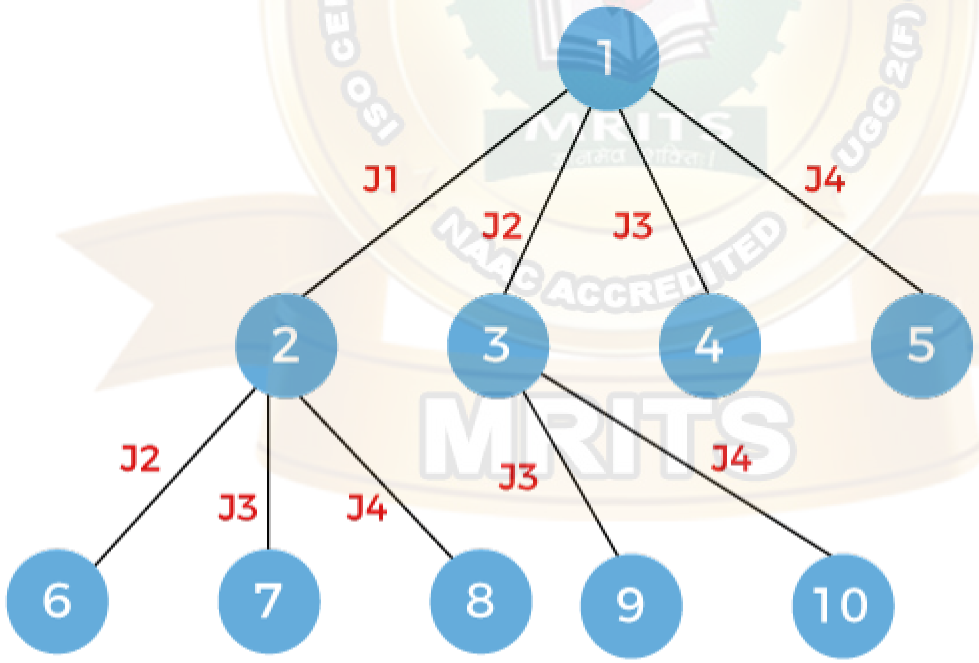
In this case, we first consider the first job, then second job, then third job and finally we consider the last job.

As we can observe in the above figure that the breadth first search is performed but not the depth first search. Here we move breadth wise for exploring the solutions. In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.

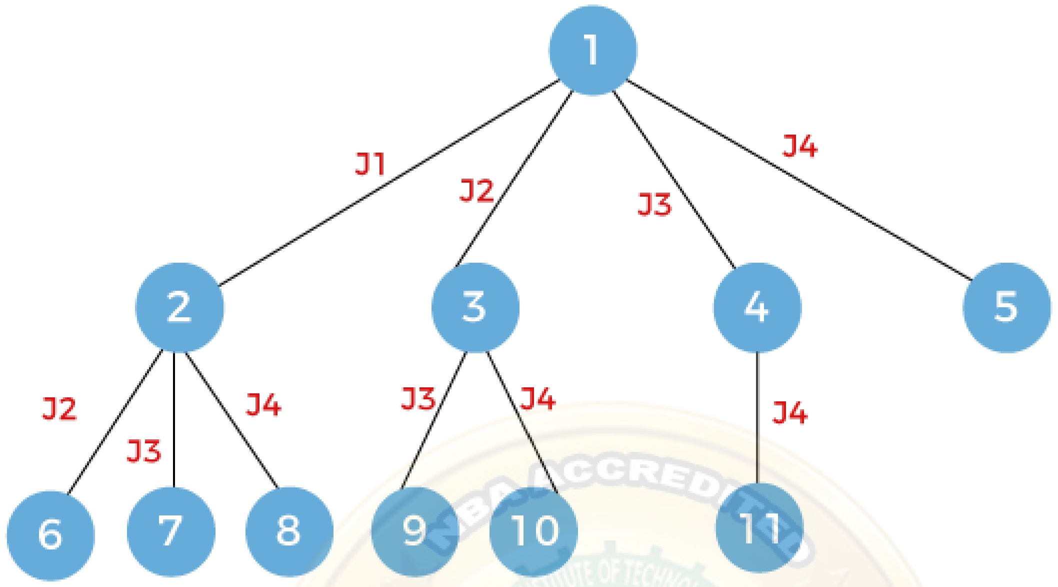
Now one level is completed. Once I take first job, then we can consider either j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.



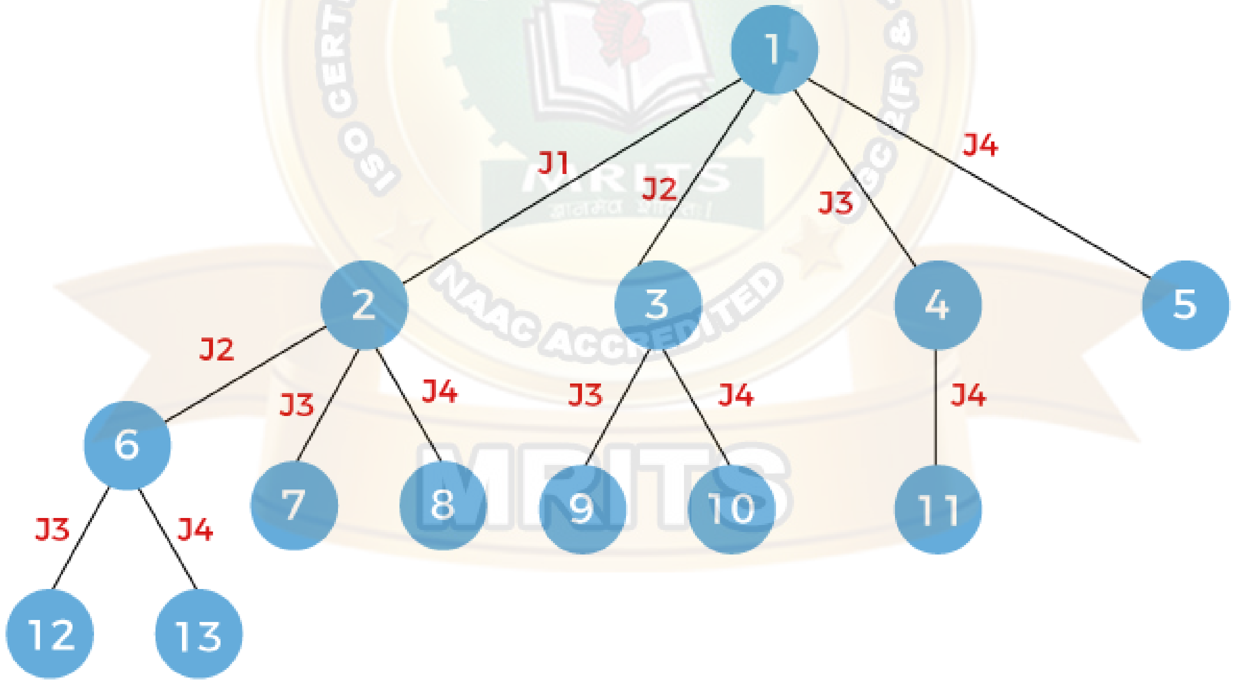
Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.



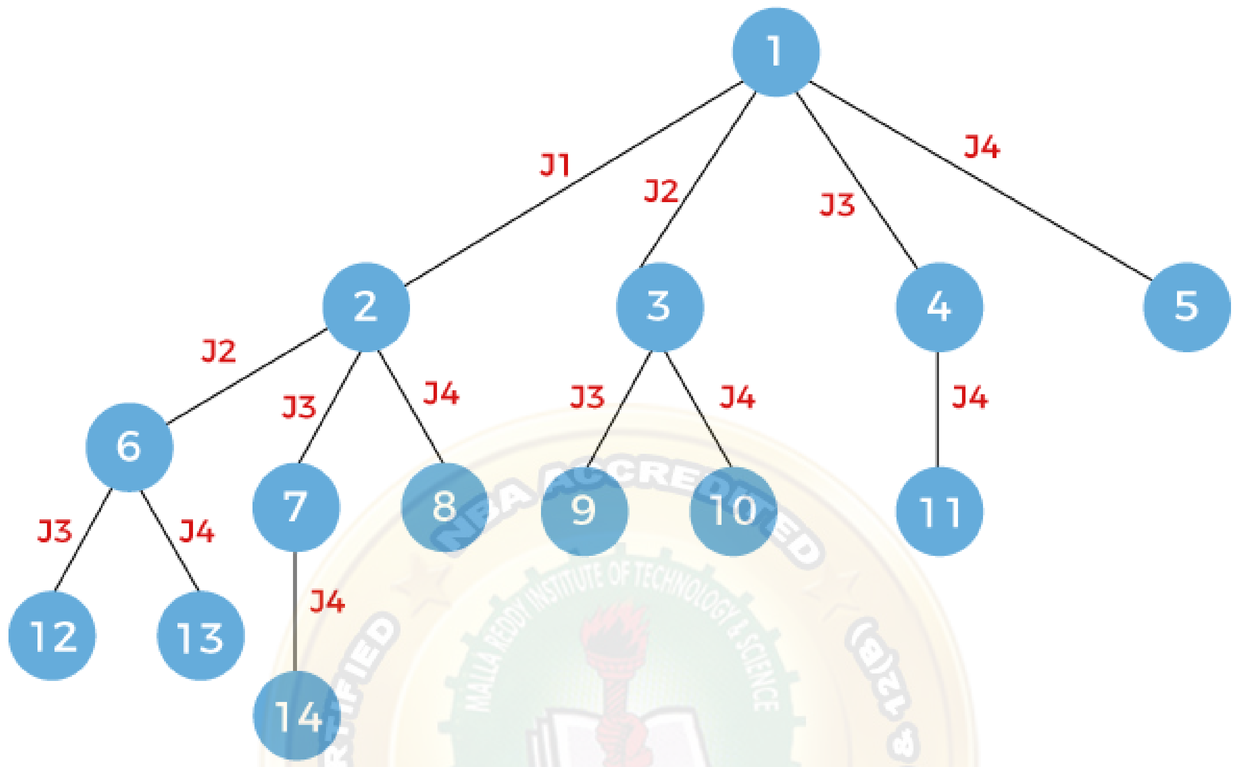
Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.



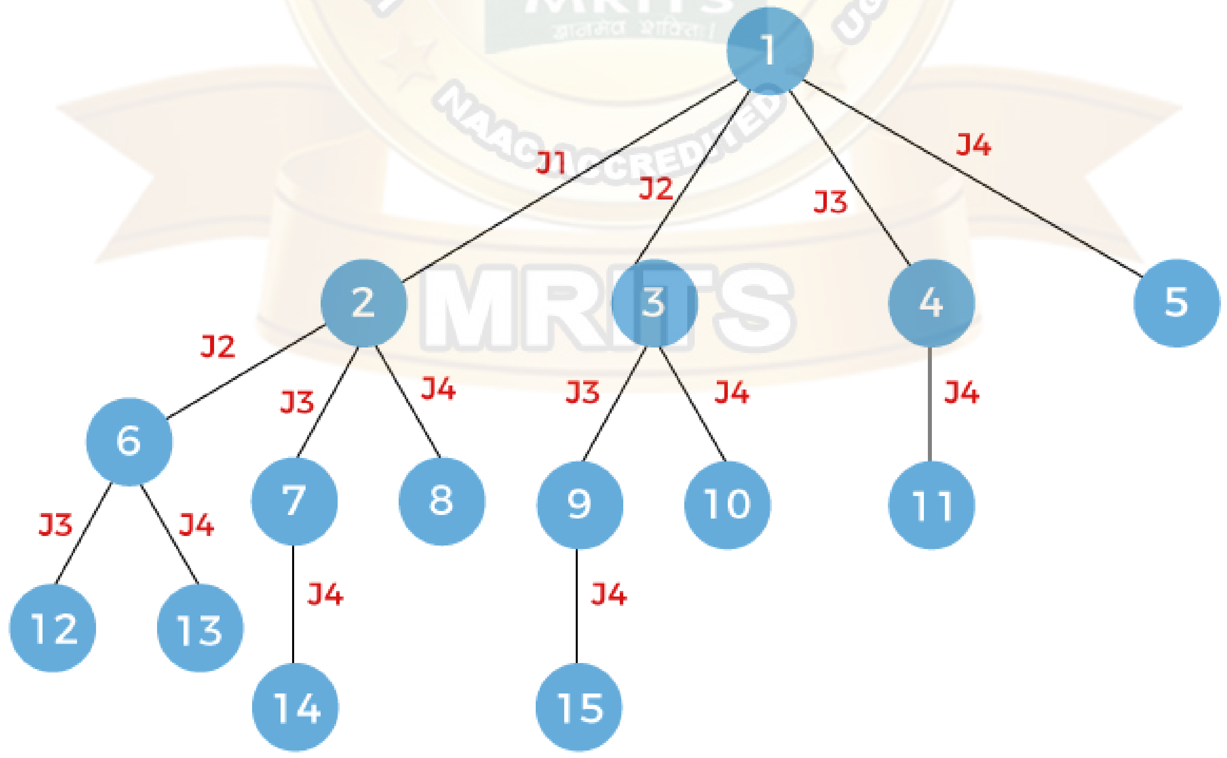
Now we will expand node 6, and here we will consider the jobs j3 and j4.



Now we will expand node 7 and here we will consider job j4.



Now we will expand node 9, and here we will consider job j4.



The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.

The above is the state space tree for the solution s1 = {j1, j4}

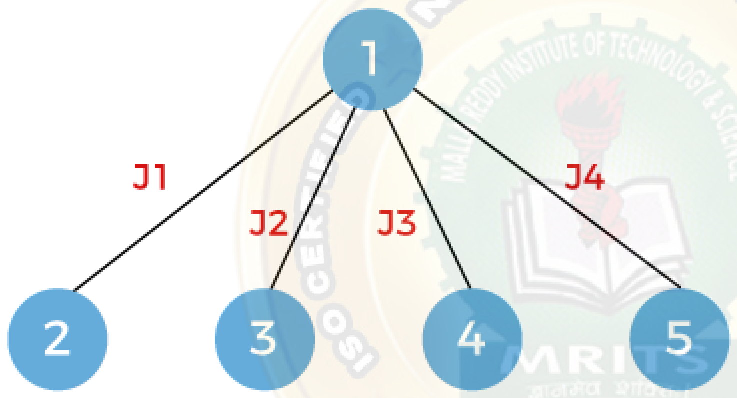
Second method: (Last in First out (LIFO) Search)

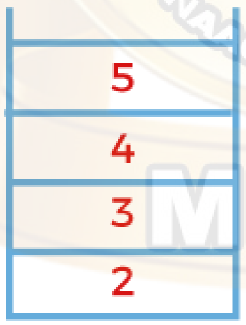
We will see another way to solve the problem to achieve the solution s1.

First, we consider the node 1 shown as below:

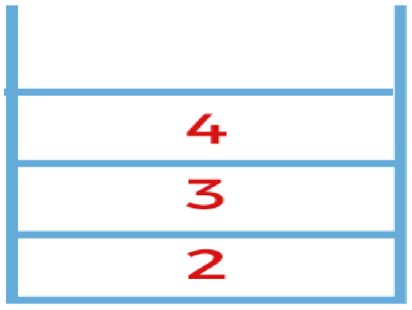
Now, we will expand the node 1. After expansion, the state space tree would be appeared as:

On each expansion, the node will be pushed into the stack shown as below:

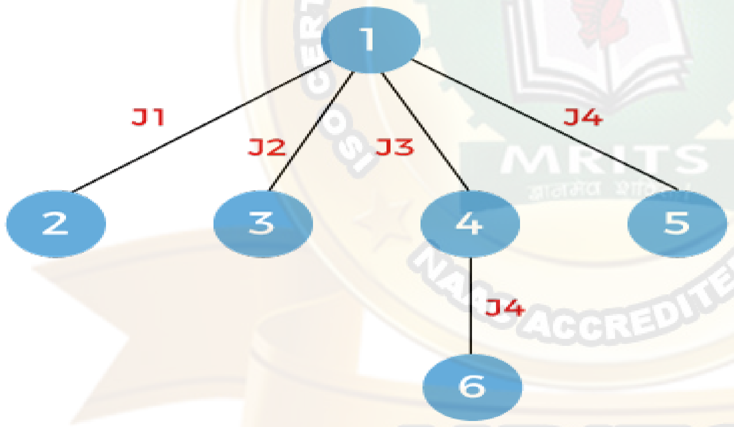


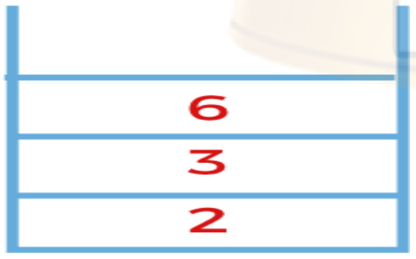


Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.

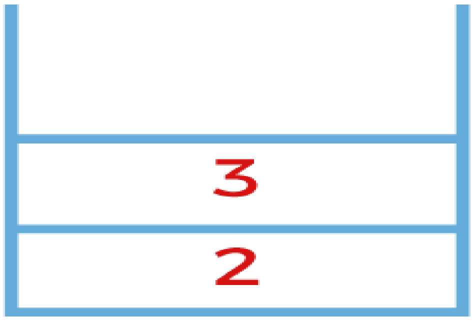


The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:

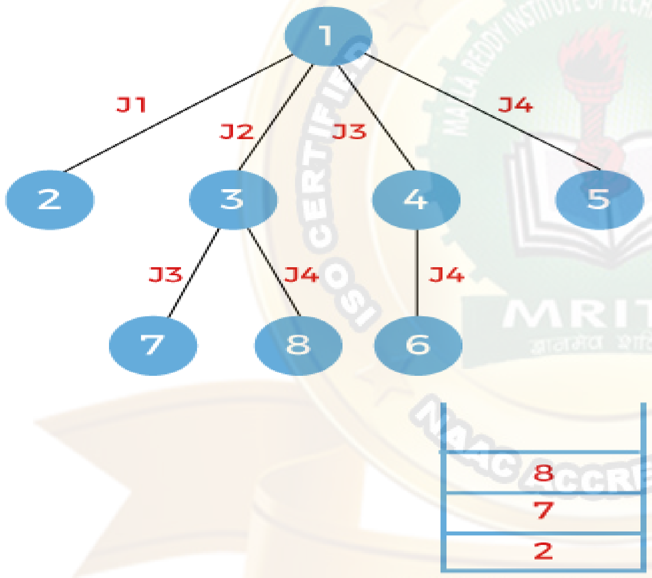




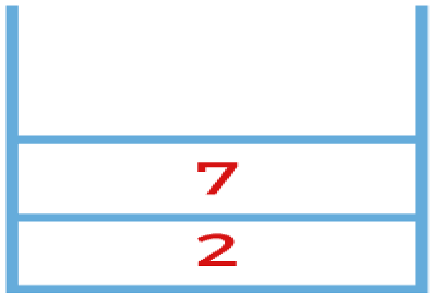
The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j4 so there is no further scope of expansion.



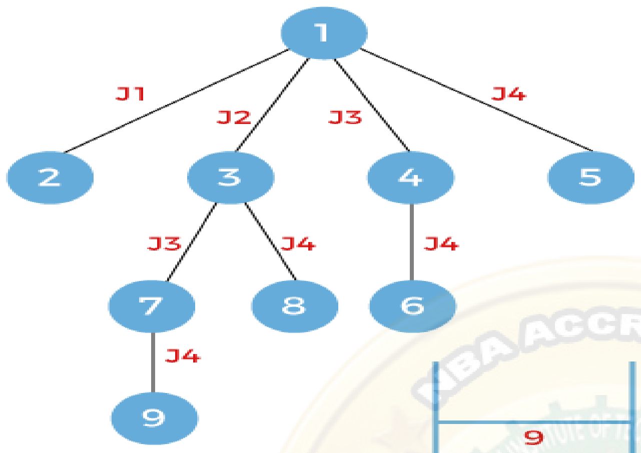
The next node to be expanded is node 3. Since the node 3 works on the job j2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:

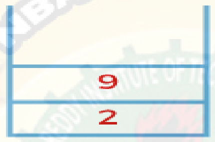


The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j4 so there is no further scope for the expansion.

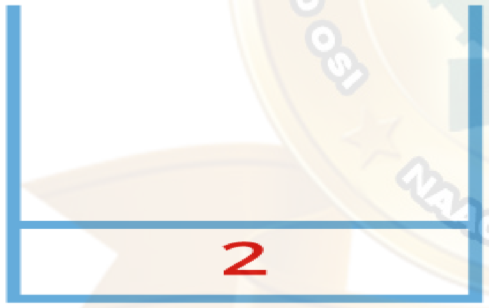


The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j3 so node 7 will be further expanded to node 9 that works on the job j4 as shown as below and the node 9 will be pushed into the stack.

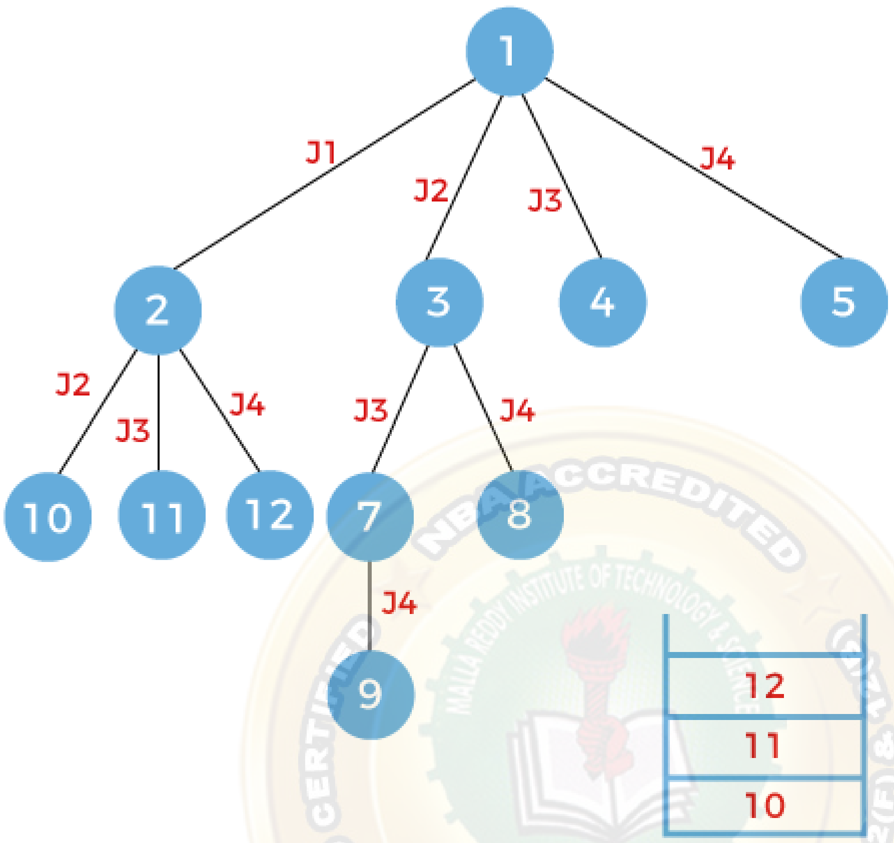




The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j2, j3, and j4 respectively. There newly nodes will be pushed into the stack shown as below:



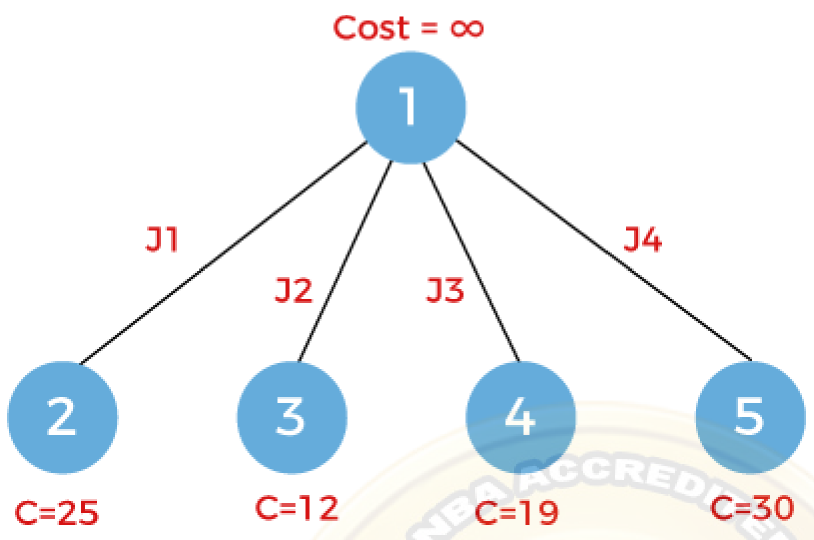
In the above method, we explored all the nodes using the stack that follows the LIFO principle.

Third method (Least -Count Search (LC))

There is one more method that can be used to find the solution and that method is Least cost branch and bound. In this technique, nodes are explored based on the cost of the node. The cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.

Let's first consider the node 1 having cost infinity shown as below:

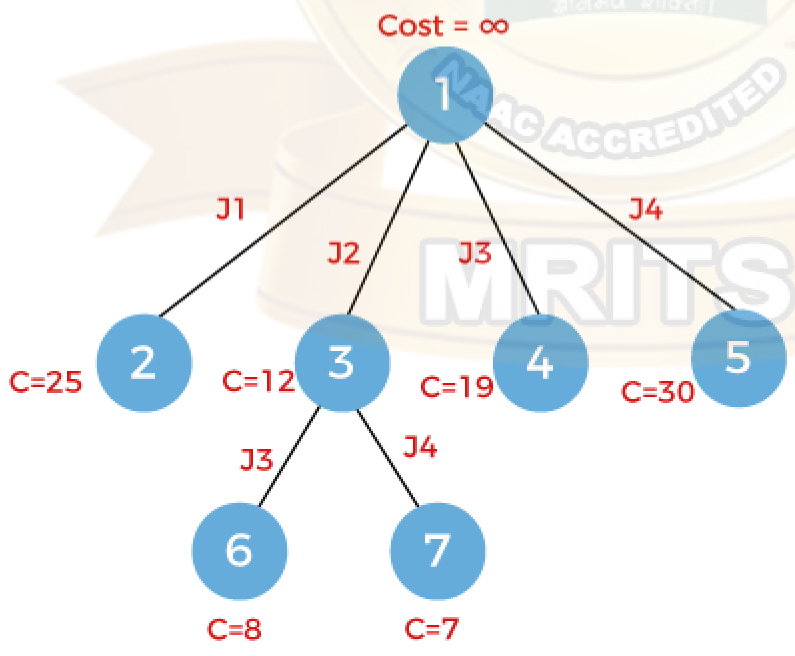
Now we will expand the node 1. The node 1 will be expanded into four nodes named as 2, 3, 4 and 5 shown as below:



Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.

Since it is the least cost branch n bound, so we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 3. So, we will explore the node 3 having cost 12.

Since the node 3 works on the job j2 so it will be expanded into two nodes named as 6 and 7 shown as below:



The node 6 works on job j3 while the node 7 works on job j4. The cost of the node 6 is 8 and the cost of the node 7 is 7. Now we have to select the node which is having the minimum cost.

The node 7 has the minimum cost so we will explore the node 7. Since the node 7 already works on the job j4 so there is no further scope for the expansion.

- Travelling salesman problem (TSP)
- Quadratic assignment problem (QAP)
- Maximum satisfiability problem (MAX-SAT)
- Nearest neighbor search
- Flow shop scheduling
- Parameter estimation
- 0/1 knapsack problem

Travelling Salesman Problem- using Branch and bound

You are given-

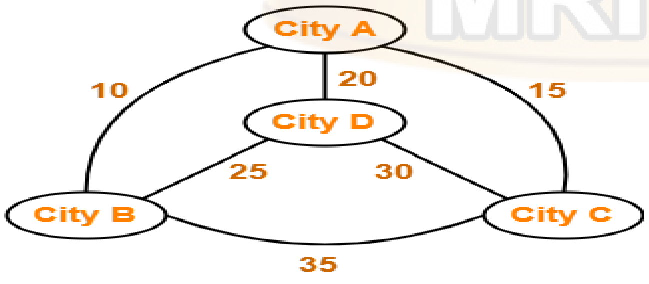
- A set of some cities
- Distance between every pair of cities

Travelling Salesman Problem states-

- A salesman has to visit every city exactly once.
- He has to come back to the city from where he starts his journey.
- What is the shortest possible route that the salesman must follow to complete his tour?

Example-

The following graph shows a set of cities and distance between every pair of cities-



Travelling Salesman Problem

If salesman starting city is A, then a TSP tour in the graph is-

$$A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$$

Cost of the tour

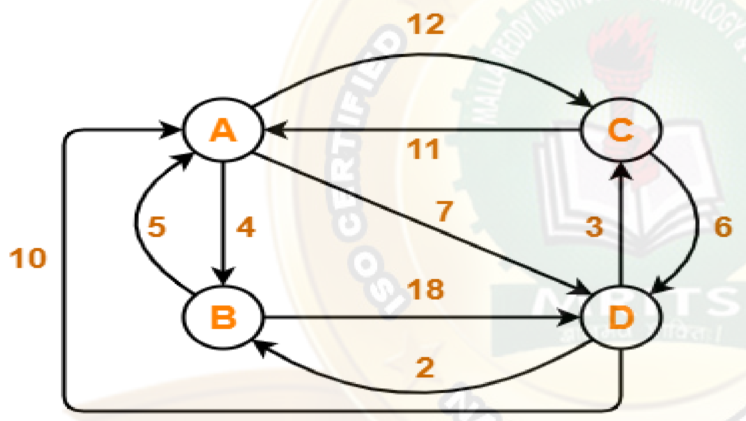
= 10 + 25 + 30 + 15

= **80 units**

PRACTICE PROBLEM BASED ON TRAVELLING SALESMAN PROBLEM USING BRANCH AND BOUND APPROACH-

Problem-

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



Solution-

Step-01:

Write the initial cost matrix and reduce it-

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

Rules

- To reduce a matrix, perform the row reduction and column reduction of the matrix separately.
- A row or a column is said to be reduced if it contains at least one entry '0' in it.

Row Reduction-

Consider the rows of above matrix one by one.

If the row already contains an entry '0', then-

- There is no need to reduce that row.

If the row does not contains an entry '0', then-

- Reduce that particular row.
- Select the least value element from that row.
- Subtract that element from each element of that row.
- This will create an entry '0' in that row, thus reducing that row.

Following this, we have-

- Reduce the elements of row-1 by 4.
- Reduce the elements of row-2 by 5.
- Reduce the elements of row-3 by 6.
- Reduce the elements of row-4 by 2.

Performing this, we obtain the following 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-

Consider the columns of above row-reduced matrix one by one.

If the column already contains an entry ‘0’, then-

- There is no need to reduce that column.

If the column does not contains an entry ‘0’, then-

- Reduce that particular column.
- Select the least value element from that column.
- Subtract that element from each element of that column.
- This will create an entry ‘0’ in that column, thus reducing that column.

Following this, we have-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- Reduce the elements of column-3 by 1.
- There is no need to reduce column-4.

Performing this, we obtain the following column-reduced matrix-

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

Finally, the initial distance matrix is completely reduced.

Now, we calculate the cost of node-1 by adding all the reduction elements.

Cost(1)

= Sum of all reduction elements

= 4 + 5 + 6 + 2 + 1

= 18

Step-02:

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

Choosing To Go To Vertex-B: Node-2 (Path A → B)

- From the reduced matrix of step-01, M[A,B] = 0
- Set row-A and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

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

Now,

- We reduce this matrix.

- Then, we find out the cost of node-02.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- Reduce all the elements of row-2 by 13.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

Performing this, we obtain the following row-reduced matrix-

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

Column Reduction-

- Reduce the elements of column-1 by 5.
- We can not reduce column-2 as all its elements are ∞.
- There is no need to reduce column-3.
- There is no need to reduce column-4.

Performing this, we obtain the following column-reduced matrix-

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

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-2.

Cost(2)

= Cost(1) + Sum of reduction elements + M[A,B]

= 18 + (13 + 5) + 0

= 36

Choosing To Go To Vertex-C: Node-3 (Path A → C)

- From the reduced matrix of step-01, M[A,C] = 7
- Set row-A and column-C to ∞
- Set M[C,A] = ∞

Now, resulting cost matrix is-

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

Now,

- We reduce this matrix.
- Then, we find out the cost of node-03.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

Thus, the matrix is already row-reduced.

Column Reduction-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-3.

Cost(3)

= Cost(1) + Sum of reduction elements + M[A,C]

= 18 + 0 + 7

= 25

Choosing To Go To Vertex-D: Node-4 (Path A → D)

- From the reduced matrix of step-01, M[A,D] = 3
- Set row-A and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-

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

Now,

- We reduce this matrix.
- Then, we find out the cost of node-04.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.

- Reduce all the elements of row-3 by 5.
- There is no need to reduce row-4.

Performing this, we obtain the following row-reduced matrix-

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

Column Reduction-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- There is no need to reduce column-3.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column-reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-4.

Cost(4)

= Cost(1) + Sum of reduction elements + M[A,D]

= 18 + 5 + 3

= 26

Thus, we have-

- Cost(2) = 36 (for Path A → B)
- Cost(3) = 25 (for Path A → C)
- Cost(4) = 26 (for Path A → D)

We choose the node with the lowest cost.

Since cost for node-3 is lowest, so we prefer to visit node-3.

Thus, we choose node-3 i.e. path **A → C**.

Step-03:

We explore the vertices B and D from node-3.

We now start from the cost matrix at node-3 which is-

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

Cost(3) = 25

Choosing To Go To Vertex-B: Node-5 (Path A → C → B)

- From the reduced matrix of step-02, M[C,B] = ∞
- Set row-C and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

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

Now,

- We reduce this matrix.
- Then, we find out the cost of node-5.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- Reduce all the elements of row-2 by 13.
- We can not reduce row-3 as all its elements are ∞.
- Reduce all the elements of row-4 by 8.

Performing this, we obtain the following row-reduced matrix-

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

Column Reduction-

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-5.

Cost(5)

= cost(3) + Sum of reduction elements + M[C,B]

= 25 + (13 + 8) + ∞

= ∞

Choosing To Go To Vertex-D: Node-6 (Path A → C → D)

- From the reduced matrix of step-02, M[C,D] = ∞
- Set row-C and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-

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

Now,

- We reduce this matrix.
- Then, we find out the cost of node-6.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

Thus, the matrix is already row reduced.

Column Reduction-

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-6.

Cost(6)

= cost(3) + Sum of reduction elements + M[C,D]

= 25 + 0 + 0

= 25

Thus, we have-

- Cost(5) = ∞ (for Path A → C → B)
- Cost(6) = 25 (for Path A → C → D)

We choose the node with the lowest cost.

Since cost for node-6 is lowest, so we prefer to visit node-6.

Thus, we choose node-6 i.e. path **C → D**.

Step-04:

We explore vertex B from node-6.

We start with the cost matrix at node-6 which is-

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

Cost(6) = 25

Choosing To Go To Vertex-B: Node-7 (Path A → C → D → B)

- From the reduced matrix of step-03, M[D,B] = 0
- Set row-D and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

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

Now,

- We reduce this matrix.
- Then, we find out the cost of node-7.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- We can not reduce row-2 as all its elements are ∞.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

Column Reduction-

- We can not reduce column-1 as all its elements are ∞.
- We can not reduce column-2 as all its elements are ∞.

- We can not reduce column-3 as all its elements are ∞.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

All the entries have become ∞.

Now, we calculate the cost of node-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**