UNIT - I

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 should ensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithm is 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 can be 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' given numbers:

```
1.  Algorithm Max(A,n)
2.  // A is an array of size n
3. {
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 Local variables.

Algorithm:

```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing
order. 3.{
4.       for I:=1 to n do
5.       {
6.              j:=I;
7.              for k:=i+1 to n do
```

```
8.                  if (a[k]<a[j])
9.                  t:=a[I];
10.                 a[I]:=a[j];
11.                 a[j]:=t;
12.       }
13. }
```

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. One is 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 this can be any one of the following:

i. We would like an algorithm that is easy to understand code anddebug.
ii. We would like an algorithm that makes efficient use of the computer's resources, 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 object program.
3. The nature and speed of the instructions on the machine used to execute the program,
4. The time complexity of the algorithm underlying theprogram.

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 to completion. 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 inprogram.
- Space needed by dynamically allocated objects such as arrays and class instances.

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

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

- The compiler used to complete the program into machinecode.
- The compiler options in effect at the time ofcompilation
- 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 do
    s= 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 floating point numbers.
- This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- 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 thealgorithm.

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.

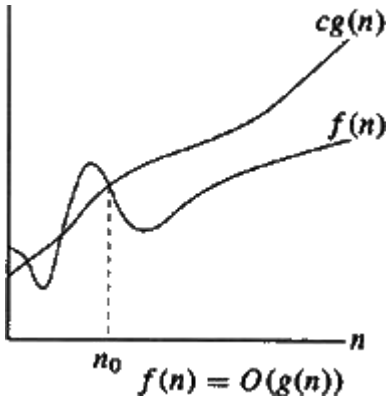
Asymptotic Notations:

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

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

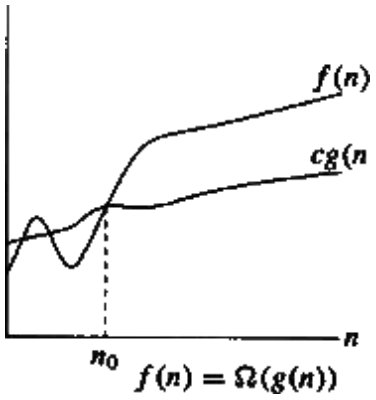
Big–OH O (Upper Bound)

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



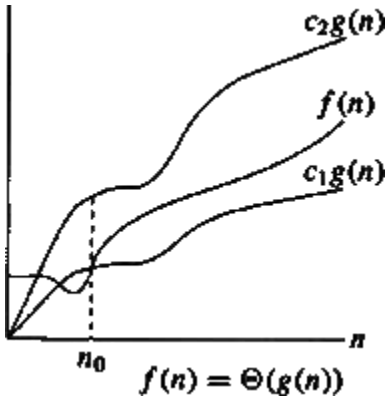
Big–OMEGA Ω (Lower Bound)

f(n) = Ω ***(g(n))*** (pronounced omega), says that the growth rate of f(n) is greater than or equal 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, "f of 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 $0 \leq f(n) < cg(n)$ for all $n \geq k$. 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. Clearly the 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:

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n. \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, 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 of the 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 few times. 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 gets slightly 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 the optimal situation for an algorithm that must process n inputs.

o. **log n** This running time arises for algorithms that solve a problem by breaking it up 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 nested loop) 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 on small 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 timesquares.

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 instructions executed by a 1 gigaflop computer in 5000 years.

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 in time 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 problems are solved recursively.
Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

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 very powerful use of recursion.

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 left undefined. 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 p1, p2, …. Pk, k ≥ 1;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
        }
}
```

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. These sub 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 computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

Binary Search:

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. 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' such that 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 of the 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 complexity of 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.

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then  // If Small(P)
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      { // Reduce P into a smaller subproblem.
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                  return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

Recursive binary search

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid − 1;
11          else if (x > a[mid]) then low := mid + 1;
12              else return mid;
13      }
14      return 0;
15  }
```

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	
			found

Number of comparisons = 4

2. Searching for x = 82

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

Number of comparisons = 3

3. Searching for x = 42

low	high	mid	
1	9	5	
6	9	7	
6	6	6	
7	6		not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid	
1	9	5	
1	4	2	
1	1	1	
2	1		not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine 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 or approximately 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 element comparisons 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 unsuccessful search is Θ(log n).

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

Analysis for worst case

Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

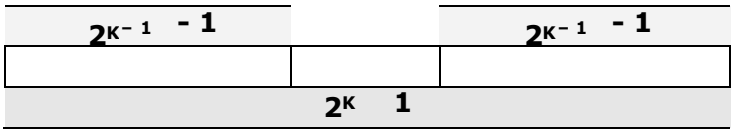
$T(0) = 0$

$T(n) = 1$ 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 always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\lceil \frac{n+1}{2} \rceil = \lceil \frac{2^K - 1 + 1}{2} \rceil = 2^{K-1}$ 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, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

This is now solved by repeated substitution:

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

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

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 seems to be no easy way to merge 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 can be done in one pass through the input, if the output is put in a third list.

Algorithm

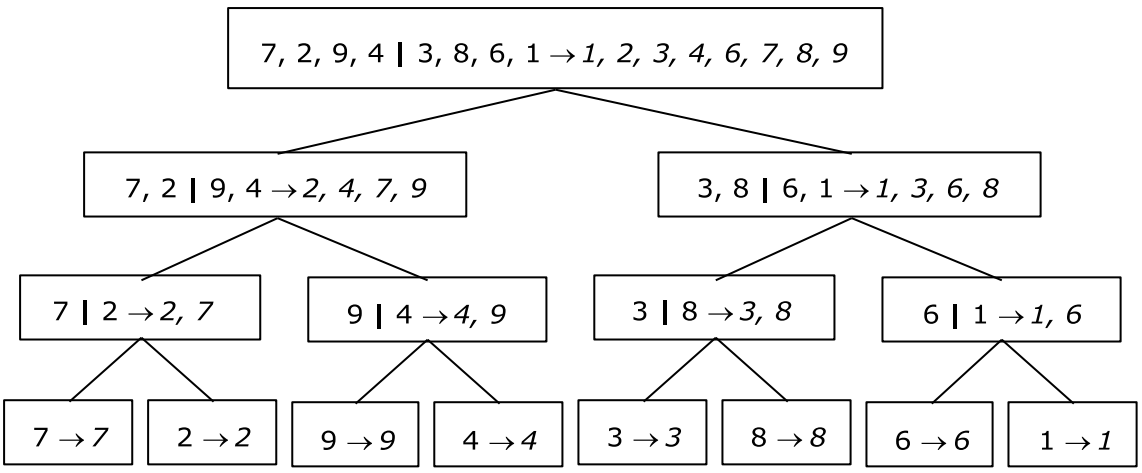
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
    if (low < high)
    {
        mid := ⌊(low + high)/2⌋;   //finds where to split the set
        MERGESORT(low, mid);   //sort one subset
        MERGESORT(mid+1, high); //sort the other subset
        MERGE(low, mid,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

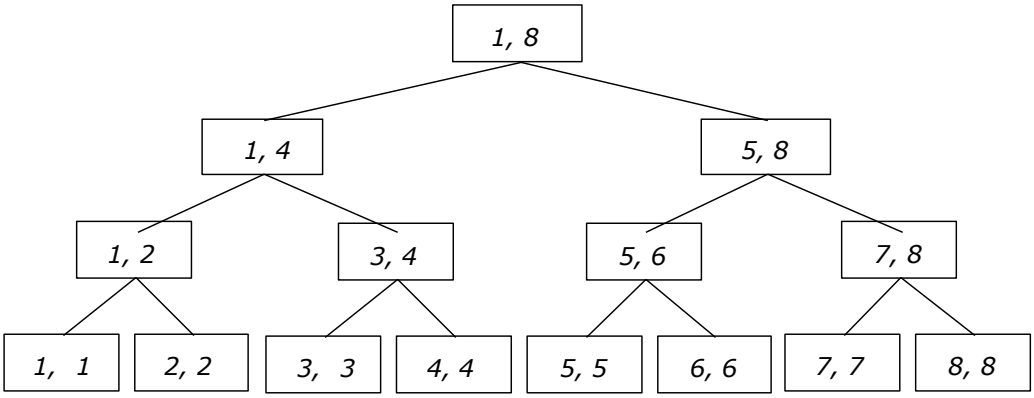
Example

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



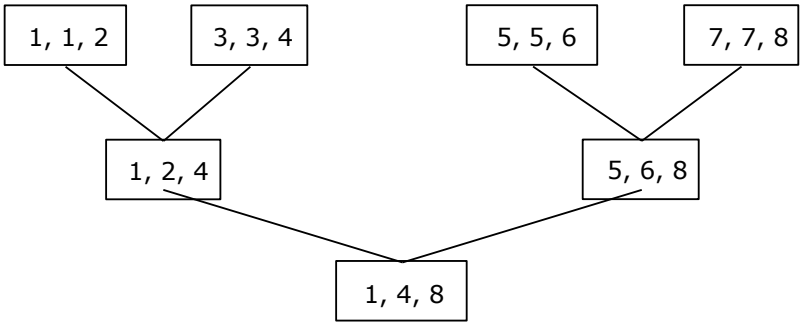
Tree Calls of MERGESORT(1, 8)

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



Tree Calls of MERGE()

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



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we 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 almost identical.

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 log n).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of 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 ofsingle numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

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.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 8\ T(n/2) \end{aligned}$$

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 of place to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ 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 rearranged in 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 sort function sorts all elements 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 completely sorted.
- 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.

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

```
1   Algorithm Partition(a, m, p)
2   // Within a[m], a[m + 1], ..., a[p − 1] the elements are
3   // rearranged in such a manner that if initially t = a[m],
4   // then after completion a[q] = t for some q between m
5   // and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6   // for q < k < p. q is returned. Set a[p] = ∞.
7   {
8       v := a[m]; i := m; j := p;
9       repeat
10      {
11          repeat
12              i := i + 1;
13          until (a[i] ≥ v);

14          repeat
15              j := j − 1;
16          until (a[j] ≤ v);

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

18      } until (i ≥ j);

19      a[m] := a[j]; a[j] := v; return j;
20  }

1   Algorithm Interchange(a, i, j)
2   // Exchange a[i] with a[j].
3   {
4       p := a[i];
5       a[i] := a[j]; a[j] := p;
6   }
```

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. We will 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 time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

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

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, the recurrence is:

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

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

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

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

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)]

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

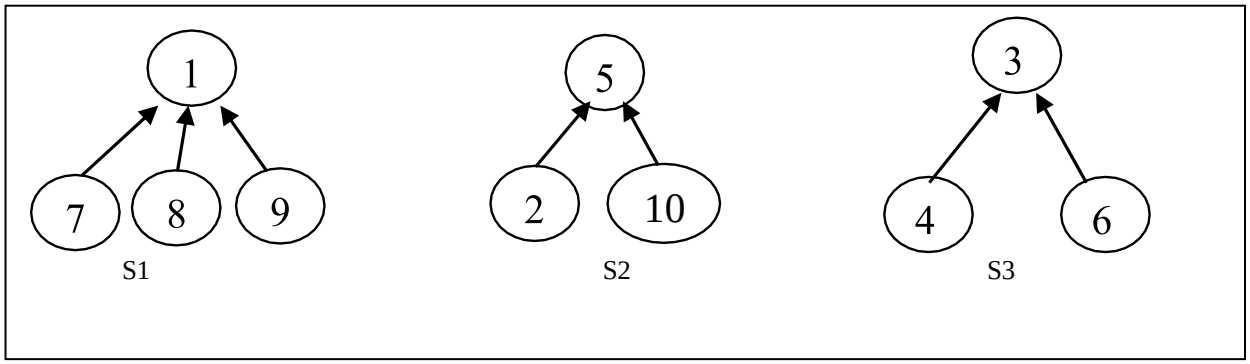
=2(n+1)-[ ]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2
T(n)= O(n log n)

UNIT-2

Disjoint Sets: If S_i and S_j, $i \neq j$ are two sets, then there is no element that is in both S_i and S_j..
For example: n=10 elements can be partitioned into three disjoint sets,

S_1= {1, 7, 8, 9}
S_2= {2, 5, 10}
S_3= {3, 4, 6}

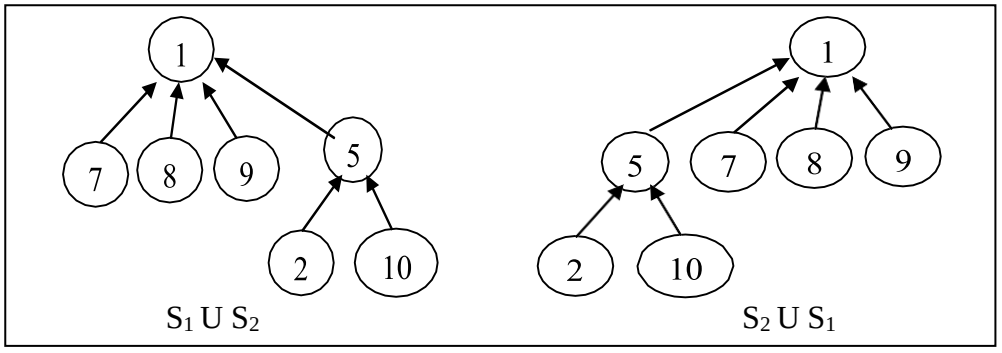
Tree representation of sets:



Disjoint set Operations:

- Disjoint set Union
- Find(i)

Disjoint set Union: Means Combination of two disjoint sets elements. Form above example $S_1 \cup S_2 = \{1,7,8,9,5,2,10\}$
For $S_1 \cup S_2$ tree representation, simply make one of the tree is a subtree of the other.



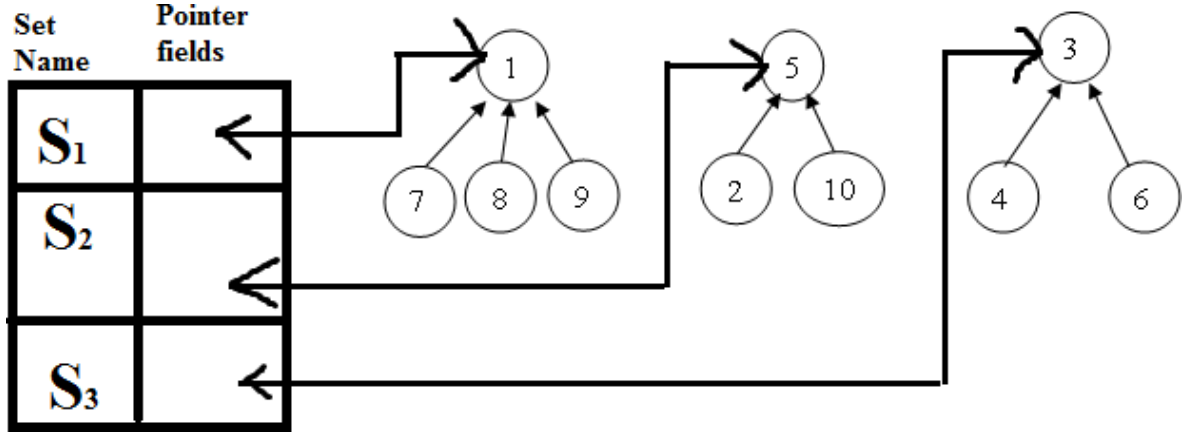
Find: Given element i, find the set containing i.
Form above example:

Find(4)→S_3

Find(1)→S_1
Find(10)→S_2

Data representation of sets:

Tress can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.



For presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them.
For example: if we determine that element 'i' is in a tree with root 'j' has a pointer to entry 'k' in the set name table, then the set name is just **name[k]**

For unite (**adding or combine**) to a particular set we use FindPointer function.
Example: If you wish to unite to S_i and S_j then we wish to unite the tree with roots
FindPointer (S_i) and FindPointer (S_j)
FindPointer→ is a function that takes a set name and determines the root of the tree that represents it.
For determining operations:
Find(i)→ 1^{St} determine the root of the tree and find its pointer to entry in setname table.
Union(i, j)→ Means union of two trees whose roots are i and j.

If set contains numbers 1 through n, we represents tree node
P[1:n].
n→Maximum number of elements.

Each node represent in array

i	1	2	3	4	5	6	7	8	9	10
P	-1	5	-1	3	-1	3	1	1	1	5

Find(i) by following the indices, starting at i until we reach a node with parent value -1.
Example: Find(6) start at 6 and then moves to 6's parent. Since P[3] is negative, we reached the root.

Algorithm for finding Union(i, j):	Algorithm for find(i)

Algorithm Simple union(i, j)
{
P[i]:=j; // Accomplishes the union
}
	Algorithm SimpleFind(i)
	{
	While(P[i]≥0) do i:=P[i];
	return i;
	}

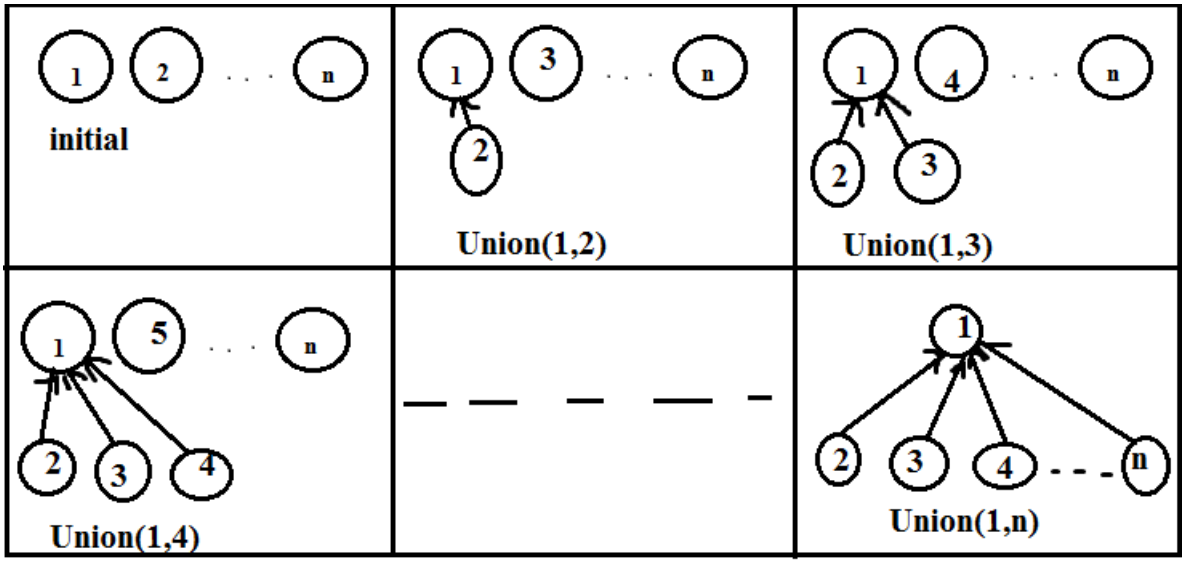
If n numbers of roots are there then the above algorithms are not useful for union and find.
For union of n trees→ Union(1,2), Union(2,3), Union(3,4),…..Union(n-1,n).
For Find i in n trees→ Find(1), Find(2),….Find(n).

Time taken for the union (simple union) is → O(1) (constant).
For the n-1 unions→ O(n).

Time taken for the find for an element at level i of a tree is → O(i).
For n finds → $O(n^2)$.

To improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. For this we use a weighting rule for union(i, j)

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



Tree obtained using the weighting rule

Algorithm for weightedUnion(i, j)

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i≠j
// The weighting 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 or equal nodes.
P[j] := i;
P[i] := temp;
}
}
```

For implementing the weighting rule, we need to know how many nodes there are in every tree.
For this we maintain a count field in the root of every tree.
i→ root node
count[i]→ number of nodes in the tree.
Time required for this above algorithm is O(1) + time for remaining unchanged is determined by using **Lemma.**

Lemma:- Let T be a tree with **m** nodes created as a result of a sequence of unions each performed using WeightedUnion. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.

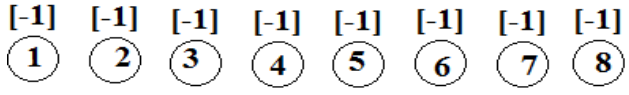
Collapsing rule: If 'j' is a node on the path from 'i' to its root and p[i]≠root[i], then set p[j] to root[i].

Algorithm for Collapsing find.

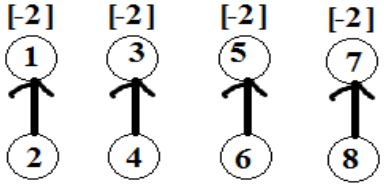
Algorithm CollapsingFind(i)
//Find the root of the tree containing element i.
//collapsing rule to collapse all nodes form i to the root.
{
r;=i;
while(p[r]>0) do r := p[r]; //Find the root.
While(i ≠ r) do // Collapse nodes from i to root r.
{
s:=p[i];
p[i]:=r;
i:=s;
}
return r;
}

Collapsing find algorithm is used to perform find operation on the tree created by WeightedUnion.

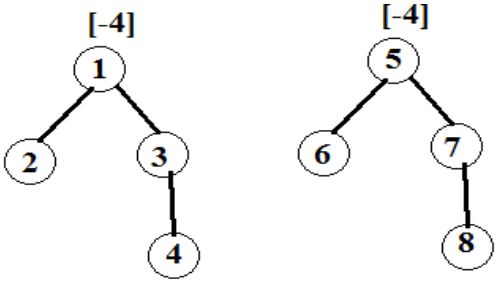
For example: Tree created by using WeightedUnion



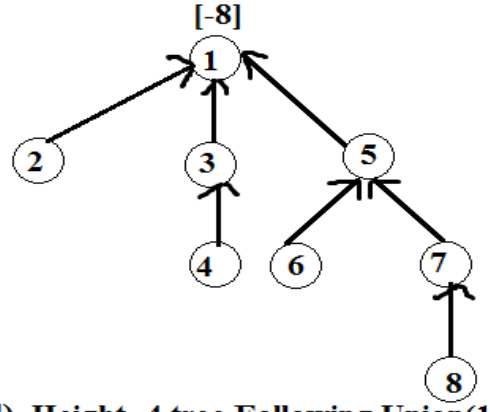
(a) initial height -1 tree



(b) Height -2 trees follwoing Union(1,2),(3,4),(5,6),(7,8)



(c) Height -3 trees following Union (1,3) and (5,7)



(d) Height -4 tree Following Union(1,5)

Now process the following eight finds: Find(8), Find(8),…….....Find(8)
If SimpleFind is used, each Find(8) requires going up three parent link fields for a total of 24 moves to process all eight finds.

W
h
e
n

C
o
l
l
a
p
s
i
n
g
F
i

nd is uised the first Find(8) requires going up three links and thenresetting two links. Total 13 movies requies for process all eight finds.

BACKTRACKING

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, , x_n) where each $x_i \in S$, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x_1, , x_n). Form a solution and check at every step
if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i's must relate to each other.

- For 8-queens problem:

  Explicit constraints using 8-tuple formation, for this problem are S= {1, 2, 3, 4, 5, 6, 7, 8}.

  The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep

track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

State space is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

Terminology:

Problem state is each node in the depth first search tree.

Solution states are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

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

Depth first node generation with bounding functions is called ***backtracking***. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

N-Queens Problem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.
All solutions to the 8-queens problem can be represented as 8-tuples (x_1, ... , x_8), where x_i is the column of the i^{th} row where the i^{th} queen is placed.

The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples.

The implicit constraints for this problem are that no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.
This realization reduces the size of the solution space from 8^8tuples to 8! Tuples.

The promising function must check whether two queens are in the same column or diagonal:
Suppose two queens are placed at positions (i, j) and (k, l) Then:

- Column Conflicts: Two queens conflict if their x_i values are identical.
- Diag 45 conflict: Two queens i and j are on the same 45^0 diagonal if:

  $$i - j = k - l.$$

  This implies, $j - l = i - k$
- Diag 135 conflict:

$$i + j = k + l.$$

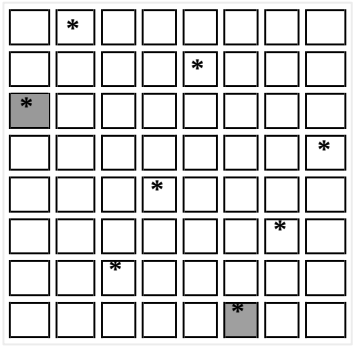
This implies, $j - l = k - i$

Therefore, two queens lie on the same diagonal if and only if:

$$|j - l| = |i - k|$$

Where, j be the column of object in row i for the i^{th} queen and l be the column of object in row 'k' for the k^{th} queen.

To check the diagonal clashes, let us take the following tile configuration:



In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

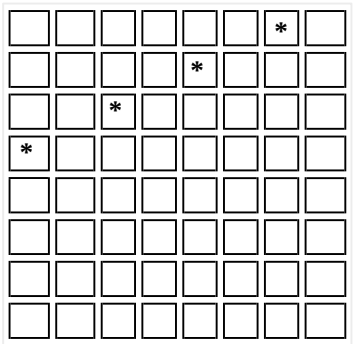
Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$$
$$\Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.



Step 1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.

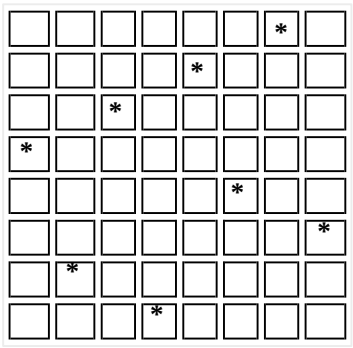
Step 3:

If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

1	2	3	4	5	6	7	8	Remarks
7	5	3	1					
7	5	3	1*	2*				$\|j - l\| = \|1 - 2\| = 1$ $\|i - k\| = \|4 - 5\| = 1$
7	5	3	1	4				
7*	5	3	1	4	2*			$\|j - l\| = \|7 - 2\| = 5$ $\|i - k\| = \|1 - 6\| = 5$
7	5	3*	1	4	6*			$\|j - l\| = \|3 - 6\| = 3$ $\|i - k\| = \|3 - 6\| = 3$
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		$\|j - l\| = \|4 - 2\| = 2$ $\|i - k\| = \|5 - 7\| = 2$
7	5	3	1	4*	8	6*		$\|j - l\| = \|4 - 6\| = 2$ $\|i - k\| = \|5 - 7\| = 2$
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*
7	5	3	1	6				
7*	5	3	1	6	2*			$\|j - l\| = \|1 - 2\| = 1$ $\|i - k\| = \|7 - 6\| = 1$
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	$\|j - l\| = \|3 - 8\| = 5$ $\|i - k\| = \|3 - 8\| = 5$
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

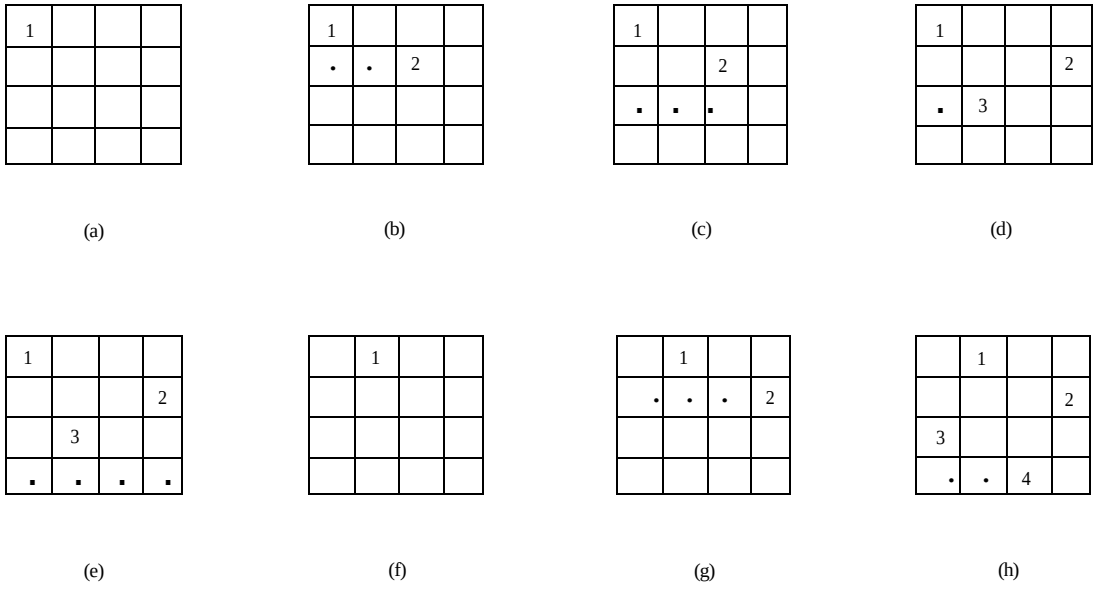
* indicates conflicting queens.

On a chessboard, the **solution** will look like:



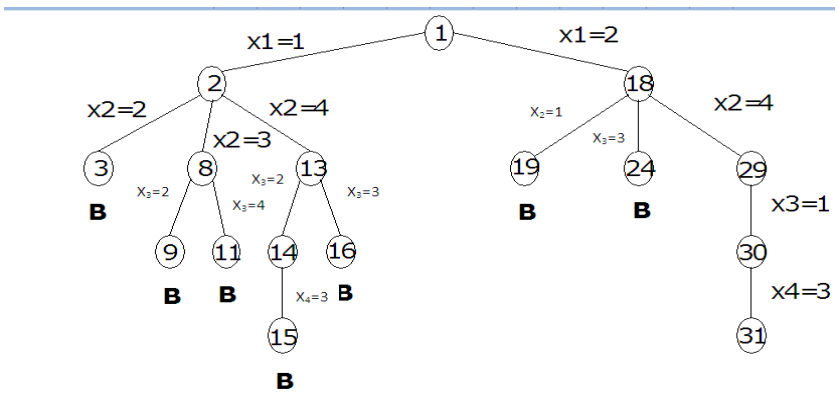
4 – Queens Problem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:



The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9                 or (Abs(x[j] − i) = Abs(j − k)))
10                    // or in the same diagonal
11                then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm 7.5 All solutions to the n-queens problem

Complexity Analysis:

$$1 + n + n^2 + n^3 + \ldots\ldots\ldots\ldots\ldots\ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

For the instance in which n = 8, the state space tree contains: 8^{8+1}

$$\frac{-1}{8-1} = 19,173,961 \text{ nodes}$$

Sum of Subsets:

Given positive numbers wi, $1 \leq i \leq n$, and m, this problem requires finding all subsets of w_i whose sums are 'm'.

All solutions are k-tuples, $1 \leq k \leq n$. Explicit

constraints:

- $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$.

Implicit constraints:

- No two x_i can be the same.
- The sum of the corresponding w_i's be m.
- $x_i < x_{i+1}$, $1 \leq i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

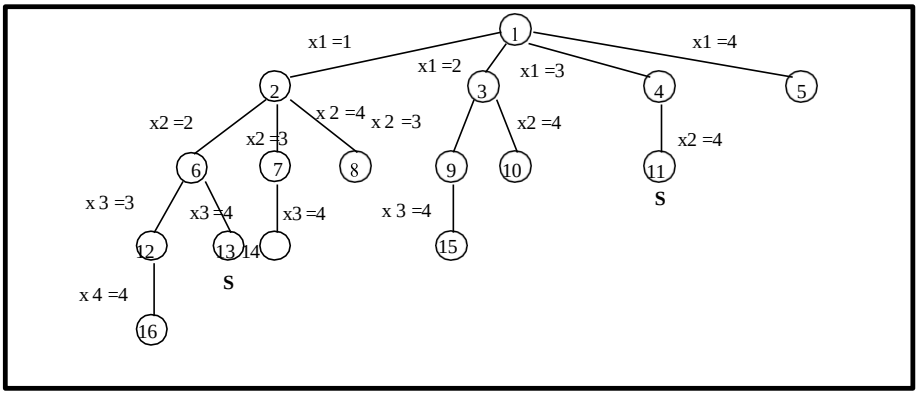
A better formulation of the problem is where the solution subset is represented by an n-tuple $(x_1, \ldots\ldots\ldots\ldots, x_n)$ such that $x_i \in \{0, 1\}$.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1). For both

the above formulations, the solution space is 2^n distincttuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are(11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of



the solution space for the case n = 4.

A possible solution space organisation for the sum of the subset s problem.

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for x_i. At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the left mot sub-tree defines all subsets containing w_1, the next sub-tree defines all subsets containing w_2 but not w_1, and so on.

```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r - w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r - w[k]);
18      }
19  }
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

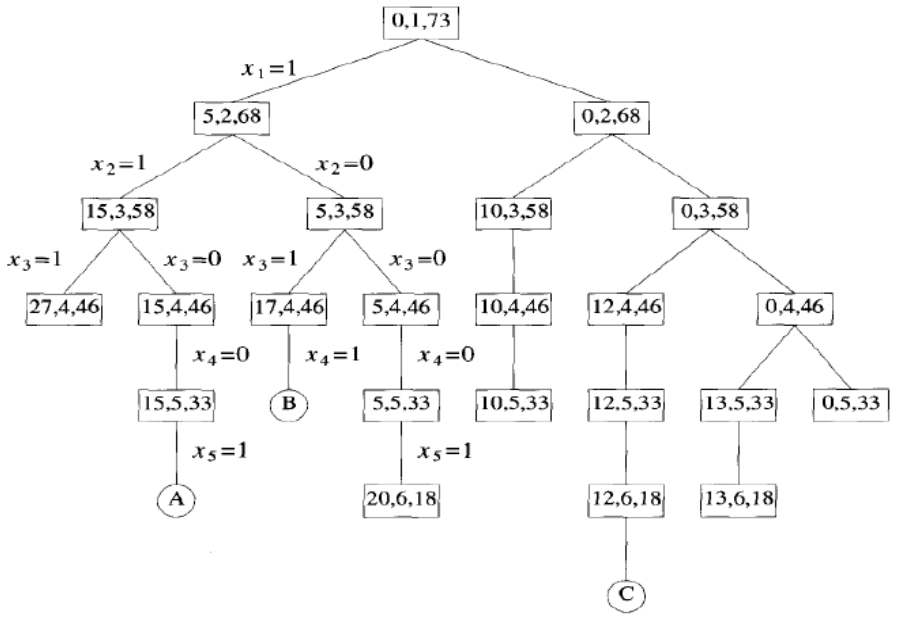


Figure 7.10 Portion of state space tree generated by SumOfSub

Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether 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. This is termed the m-colorabiltiy decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., x_n), where x_i is the color of node i.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

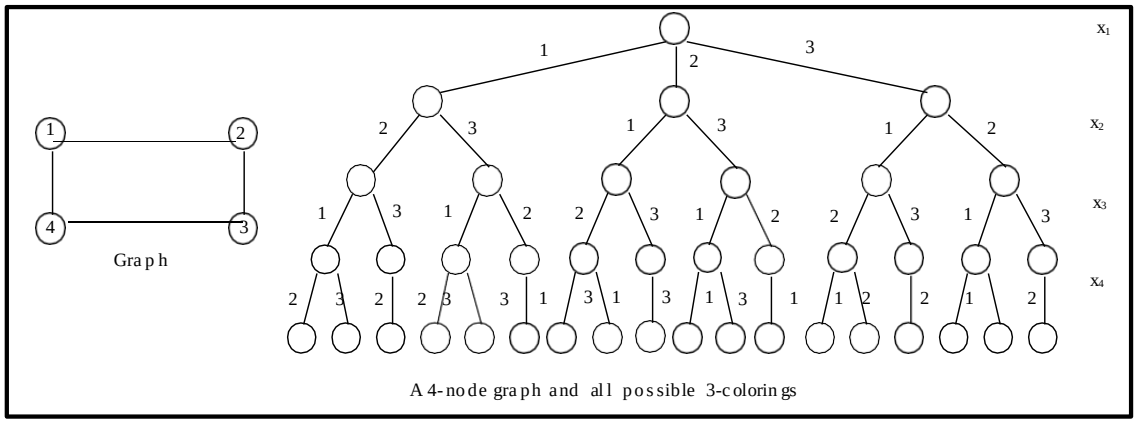
```
Algorithm mcoloring (k)
// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
// 1, 2, ....., m to the vertices of the graph such that adjacent vertices areassigned
// distinct integers are printed. k is the index of the next vertex to color.
{
        repeat
        {                                                   // Generate all legal assignments forx[k].
                NextValue   (k);            // Assign to x [k] a legal color. If (x [k]
                = 0)  then return;          // No new color possible If (k =  n) then
                                            // at most 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] suchthat
// adjacent vertices have distinct integers. A value for x [k] is determined in the range
// [0, m].x[k] is assigned the next highest numbered color while maintaining distinctness
// from 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 colors if ((G [k, j] ≠
                        0) and (x [k] = x [j]))
                        // If (k, j) is and edge and if adj. vertices have the same color. then break;
```

```
            }
            if (j = n+1)  then return;                    // New color found
        } until (false);                                  // Otherwise try to find another color.
}
```

Example:

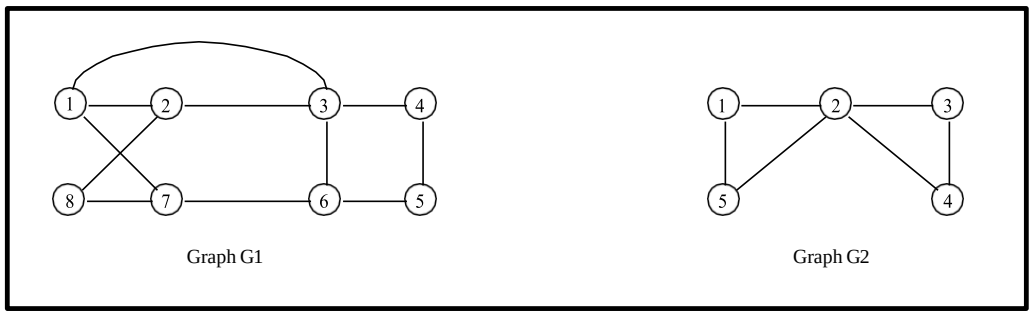
Color the graph given below with minimum number of colors by backtracking using state space tree



A 4-node graph and all possible 3-colorings

Hamiltonian Cycles:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order $v_1, v_2, \ldots\ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct expect for v_1 and v_{n+1}, which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.



Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector $(x_1, \ldots\ldots x_n)$ is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If k = 1, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to k_{x-1}. The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1.

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency

matrix G[1: n, 1: n], then setting x[2: n] to zero and x[1] to 1, and then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

Algorithm NextValue (k)

```
// x [1: k-1] is a path of k – 1 distinct vertices . If x[k] = 0, then no vertex has as yet been
// assigned to x [k]. After execution, x[k] is assigned to the next highest numberedvertex
// which does not already appear in x [1 : k – 1] and is connected by an edge to x [k – 1].
// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].
{
        repeat
        {
                x [k] := (x [k] +1)   mod (n+1);                    // Next vertex. If
                (x [k] = 0) then return;
                If (G [x [k – 1], x [k]] ≠ 0) then
                {                                                   // Is there an edge?
                        for j := 1 to k – 1 do if (x [j] = x [k]) then break;
                                                                    // check for distinctness.
                        If (j =  k) then                            // If true, then the vertex is distinct. If ((k < n)
                        or ((k = n) and G [x [n], x [1]] ≠ 0))
                        then return;
```

```
                }
        } until (false);
}
```

Algorithm Hamiltonian (k)

```
// This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian
// cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles begin
// at node 1.
{
        repeat
        {                                                       // Generate values for x [k].
                NextValue   (k);                        //Assign a legal Next value to x [k]. if (x [k]
        = 0) then return;
                if (k = n) then write (x [1: n]); else
                Hamiltonian (k + 1)
        } until (false);
```

UNIT - I

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 should ensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithm is 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 can be 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' given numbers:

```
1.  Algorithm Max(A,n)
2.  // A is an array of size n
3. {
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 Local variables.

Algorithm:

```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing
order. 3.{
4.       for I:=1 to n do
5.       {
6.              j:=I;
7.              for k:=i+1 to n do
```

```
8.                  if (a[k]<a[j])
9.                  t:=a[I];
10.                 a[I]:=a[j];
11.                 a[j]:=t;
12.       }
13. }
```

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. One is 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 this can be any one of the following:

i. We would like an algorithm that is easy to understand code anddebug.
ii. We would like an algorithm that makes efficient use of the computer's resources, 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 object program.
3. The nature and speed of the instructions on the machine used to execute the program,
4. The time complexity of the algorithm underlying theprogram.

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 to completion. 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 inprogram.
- Space needed by dynamically allocated objects such as arrays and class instances.

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

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

- The compiler used to complete the program into machinecode.
- The compiler options in effect at the time ofcompilation
- 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 do
    s= 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 floating point numbers.
- This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- 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 thealgorithm.

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.

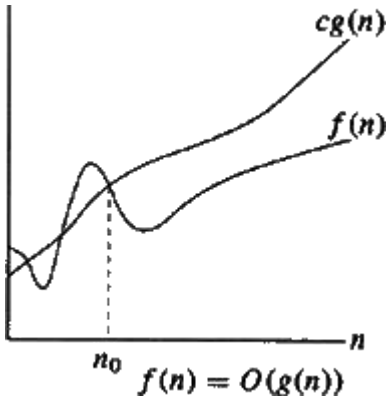
Asymptotic Notations:

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

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

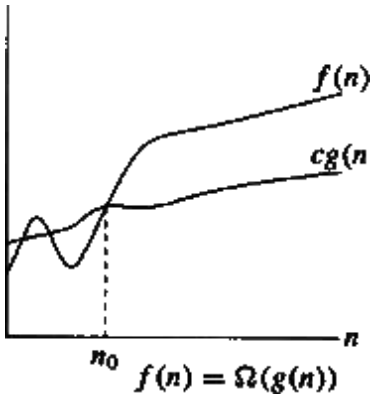
Big–OH O (Upper Bound)

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



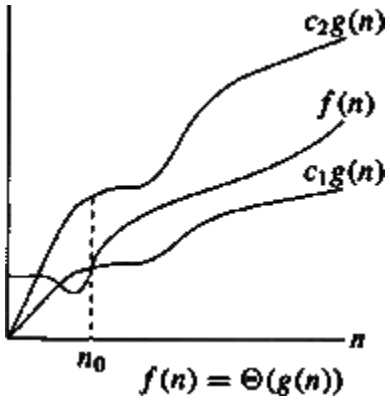
Big–OMEGA Ω (Lower Bound)

f(n) = Ω ***(g(n))*** (pronounced omega), says that the growth rate of f(n) is greater than or equal 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, "f of 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 $0 \leq f(n) < cg(n)$ for all $n \geq k$. 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. Clearly the 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:

$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n. \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, 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 of the 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 few times. 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 gets slightly 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 the optimal situation for an algorithm that must process n inputs.

o. **log n** This running time arises for algorithms that solve a problem by breaking it up 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 nested loop) 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 on small 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 timesquares.

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 instructions executed by a 1 gigaflop computer in 5000 years.

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 in time 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 problems are solved recursively.
Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

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 very powerful use of recursion.

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 left undefined. 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 p1, p2, …. Pk, k ≥ 1;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
        }
}
```

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. These sub 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 computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

Binary Search:

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. 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' such that 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 of the 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 complexity of 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.

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then  // If Small(P)
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      { // Reduce P into a smaller subproblem.
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                  return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

Recursive binary search

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid − 1;
11          else if (x > a[mid]) then low := mid + 1;
12              else return mid;
13      }
14      return 0;
15  }
```

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	
			found

Number of comparisons = 4

2. Searching for x = 82

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

Number of comparisons = 3

3. Searching for x = 42

low	high	mid	
1	9	5	
6	9	7	
6	6	6	
7	6		not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid	
1	9	5	
1	4	2	
1	1	1	
2	1		not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine 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 or approximately 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 element comparisons 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 unsuccessful search is Θ(log n).

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

Analysis for worst case

Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

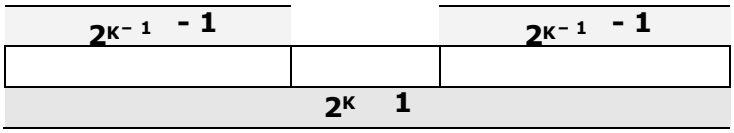
$T(0) = 0$

$T(n) = 1$ 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 always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\lceil \frac{n+1}{2} \rceil = \lceil \frac{2^K - 1 + 1}{2} \rceil = 2^{K-1}$ 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, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

This is now solved by repeated substitution:

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

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

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 seems to be no easy way to merge 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 can be done in one pass through the input, if the output is put in a third list.

Algorithm

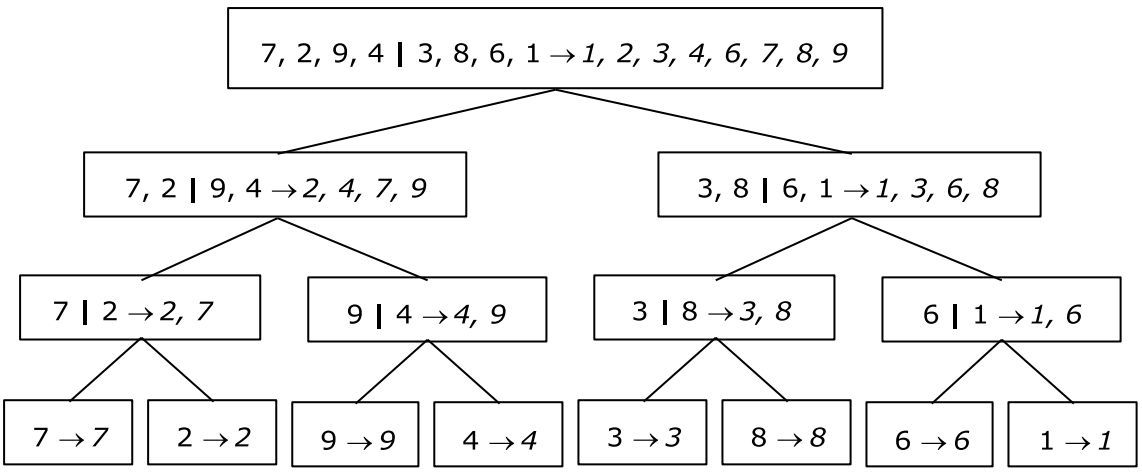
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
    if (low < high)
    {
        mid := ⌊(low + high)/2⌋;   //finds where to split the set
        MERGESORT(low, mid);   //sort one subset
        MERGESORT(mid+1, high); //sort the other subset
        MERGE(low, mid,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

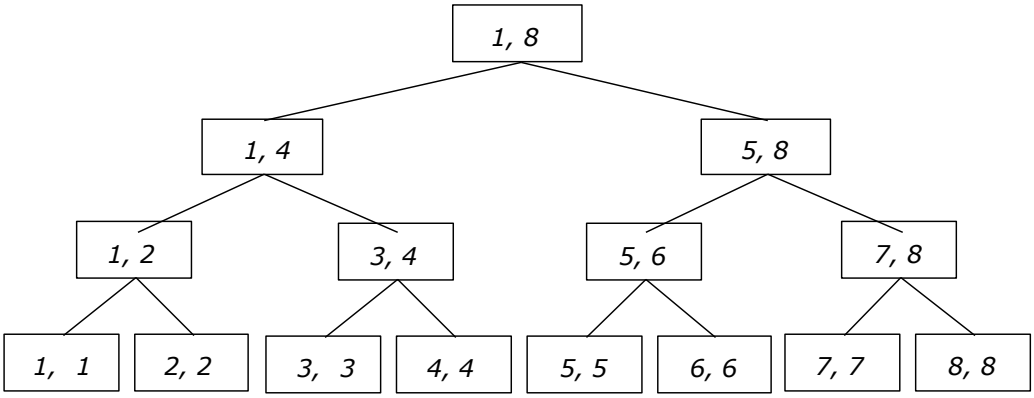
Example

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



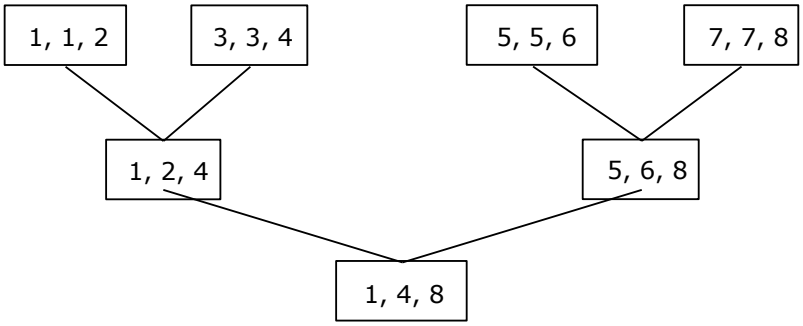
Tree Calls of MERGESORT(1, 8)

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



Tree Calls of MERGE()

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



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we 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 almost identical.

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 log n).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of 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 ofsingle numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

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.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 8\ T(n/2) \end{aligned}$$

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 of place to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ 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 rearranged in 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 sort function sorts all elements 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 completely sorted.
- 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.

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

```
1   Algorithm Partition(a, m, p)
2   // Within a[m], a[m + 1], ..., a[p − 1] the elements are
3   // rearranged in such a manner that if initially t = a[m],
4   // then after completion a[q] = t for some q between m
5   // and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6   // for q < k < p. q is returned. Set a[p] = ∞.
7   {
8       v := a[m]; i := m; j := p;
9       repeat
10      {
11          repeat
12              i := i + 1;
13          until (a[i] ≥ v);

14          repeat
15              j := j − 1;
16          until (a[j] ≤ v);

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

18      } until (i ≥ j);

19      a[m] := a[j]; a[j] := v; return j;
20  }

1   Algorithm Interchange(a, i, j)
2   // Exchange a[i] with a[j].
3   {
4       p := a[i];
5       a[i] := a[j]; a[j] := p;
6   }
```

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. We will 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 time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

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

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, the recurrence is:

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

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

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

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

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)]

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

=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)

UNIT-III

Dynamic programming

General Method :-

Dynamic Programming is a method where a given problem is divided into subproblems. The subproblems are then solved & stored in a table. whenever the situation of solving the same subproblem arises, then the stored solution is fetched from the table instead of recomputing the subproblem again.

The idea behind dynamic programming is "Avoid solving the same problem twice".

- Dynamic programming is based on the principle of 'optimality'. It is typically applied to optimization problems. In such problems there can be many solutions. Each solution has a value & we wise to find a solution with the optimal value.

Major components in Dynamic programming :-

∴ Recursion :- Solve the problems recursively.

∴ Memorization :- To store the pre computation of subproblems to use further.

(DP) Dynamic programming = recursion + Memorization.

- The time complexity using D.P is $O(n)$

In dynamic programming an optimal sequence of decisions are obtained by making explicit appeal to the "principle of optimality".

Scanned with CamScanner

Principle of Optimality :-

In this, the decisions resulting from the first decisions has a property that they are in optimal sequence irrespective of the initial state & decision. It always give the optimal solution from many feasible solutions. Optimal solution will be achieved from the initial state itself.

when it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using dynamic programming approach.

Eg :- finding the shortest path in a given graph.

The difference between the greedy & dynamic programming is that in greedy method only one decision sequence is ever generated.

In dynamic programming many decision sequences may be generated.

Characteristics of dynamic programming :-

1, Simple subproblems :-

we should be able to break the original problem into small subproblems that have the same structure.

2, Optimal structure of Problem :-

The optimal solution to the problem contains within optimal solution to its subproblem.

3, Overlapping subproblems :-

There exists some places where we solve the same subproblem morethan once.

Scanned with CamScanner

Steps for solving problem using DP :-

- Breaks down the complex problem into simpler subproblems.
- Find the optimal solution to these subproblems.
- Store the result of subproblems (Memoization)
- Reuses the above results so that the same sub problem is calculated more than once.
- Finally calculate the result of the complex problem.

Approaches of dynamic Programming :-

1. Top-down approach :- (Memorization)

In this approach the problem is broken into sub problems & the result at each subproblem is remembered when a result at subproblem is need again further, it is first checked in memory & then used.

2. Bottom up Approach :- (table Method)

This approach follows the idea of computing first & moves backward. It is done to avoid the process of recursion. The smaller inputs at each step in the bottomup approach contribute to form larger once.

Applications :-

1. Optimal binary search tree
2. All pairs shortest path problem
3. Travelling sales person problem
4. 0/1 knapsack problem
5. Reliability design
6. Chain matrix multiplication
7. String algorithms
8. Optimized graph algorithms.

Scanned with CamScanner

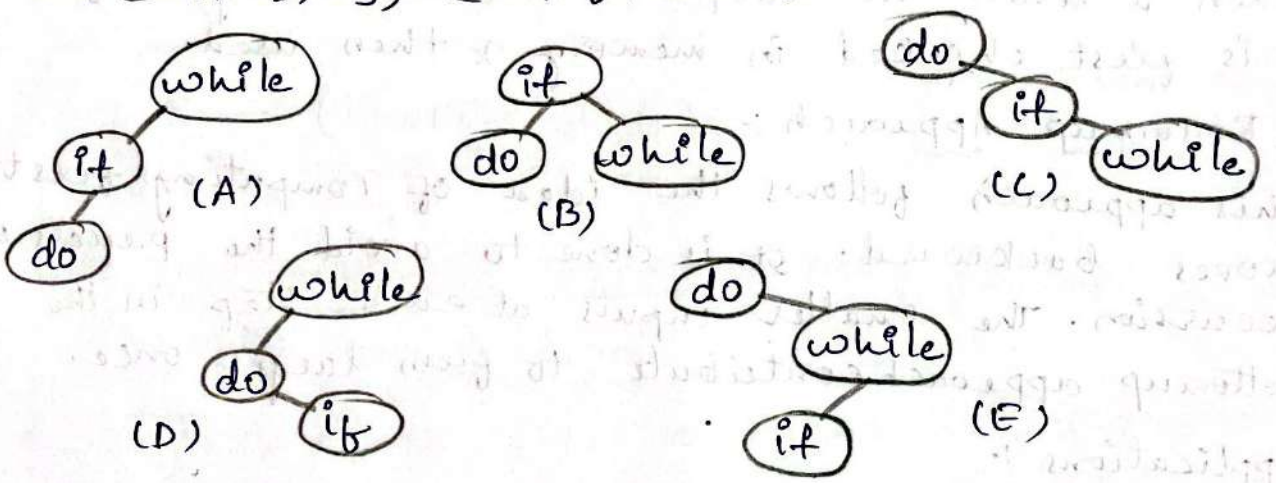
Optimal Binary Search Tree:-

In BST the nodes in the leaf subtree have lesser value than the root node & the nodes in the right subtree have greater value than the root node. The cost of searching a key is very important in various applications.

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.

Ex:-

The possible binary search trees for the identifier set $(a_1, a_2, a_3) = (do, if, while)$.



tree A requires 3 comparisons
tree B requires 2 comparisons
tree C requires 3 comparisons
tree D requires 3 comparisons
tree E requires 3 comparisons.

Tree B requires least no of comparisons & tree having optimum cost is known as optimal binary search tree.

Scanned with CamScanner

Let us assume that the given set of identifiers is $\{a_1, a_2 \ldots a_n\}$ with $a_1 < a_2 < \ldots a_n$. Let $P(i)$ be the probability with which we search for a_i. Let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \le i \le n$

Then $\sum_{0 \le i \le n} q(i)$ is the probability of an unsuccessful search.

$$\sum_{1 \le i \le n} P(i) + \sum_{0 \le i \le n} q(i) = 1.$$

$P(i)$ denote the probability of successful search & $q(i)$ denote the probability of unsuccessful search.
A tree with minimum cost is obtained by adding $P(i)$ and $q(i)$ ie,

$$c(i) = \sum_{i=1}^{n} P(i) + \sum_{i=1}^{n} q(i).$$

In dynamic programming, to get mincost, we use following formulas.

$$c(i,j) = \min_{i < k \le j} \{c(i, k-1) + c(k, j)\} + w(i,j)$$

$$w(i,j) = w(i, j-1) + P(j) + q(j)$$

$$r(i,j) = k.$$

Scanned with CamScanner

Algorithm:-

```
Algorithm OBST(P, q, n)
// Given n distinct identifiers a1 < a2 < ... an and probabilities P[i], 1 ≤ i ≤ n and q[i], 0 ≤ i ≤ n, this algorithm computes the cost c[i,j] of optimal binary search trees tij for identifiers ai+1, ... aj. It also computes r[i,j], the root of tij
w[i,j] is the weight of tij.
{
   for i=0 to n-1 do
   {
   //Initialize.
    w[i,j] = q[i];
    r[i,i] = 0;
    c[i,i] = 0.0;
  // optimal trees with one node
  w[i,i+1] = q[i] + q[i+1] + P[i+1];
  r[i,i+1] = i+1;
  c[i,i+1] = q[i] + q[i+1] + P[i+1];
}
w[n,n] = q[n]; r[n,n] = 0; c[n,n] = 0.0;
for m=2 to n do // Find optimal trees with m nodes.
   for i=0 to n-m do
    {
      j = i+m;
     w[i,j] = w[i,j-1] + P[j] + q[j];
       k = Find[c, r, i, j];
 // A value of l in the range r[i,j-1] ≤ l
//≤ r[i+1,j] that minimizes c[i,l-1] + c[l,j];
 c[i,j] = w[i,j] + c[i,k-1] + c[k,j];
   r[i,j] = k;
}
```

Scanned with CamScanner

```
    write (c[0,n], w[0,n], r[0,n]);
  }
Algorithm find(c, r, i, j)
  {
    min = ∞ ;
  for m = r[i, j-1] to r[i+1, j] do
    if (c[i, m-1] + c[m, j]) < min then
      {
        min = c[i, m-1] + c[m, j] ; l = m;
      }
      return l;
  }
```

Time complexity :-

Each c(i,j) can be computed in time O(m). The total time for all c(i,j) is O(n-m).

Time complexity is $O(m^3)$.

Scanned with CamScanner

Ex :-

Let $n = 4$ and $(a_1, a_2, a_3, a_4) = (do, if, int, while)$. Let $P(1:4) = (3, 3, 1, 1)$ and $q(0:4) = (2, 3, 1, 1, 1)$. Initially, we have $w(i, j) = q(i)$. $c(i, i) = 0$, $r(i, i) = 0$, $0 \le i \le 4$. Construct OBST.

($P_1\ P_2\ P_3\ P_4$: cost; $q_0\ q_1\ q_2\ q_3\ q_4$: root)

Sol :- Identifier set $= (a_1, a_2, a_3, a_4) = (do, if, int, while)$

$P_1 = 3,\ P_2 = 3,\ P_3 = 1,\ P_4 = 1,\ q_0 = 2,\ q_1 = 3,\ q_2 = 1,\ q_3 = 1,\ q_4 = 1$

consider the set who's $\boxed{j - i = 0}$

possible terms are $T_{00}, T_{11}, T_{22}, T_{33}, T_{44}$

computing the value of $w(i, j)$, $R(i, j)$ & $c(i, j)$

Initially, we have $w(i, j) = q(i)$, $c(i, j) = 0$, $r(i, i) = 0$

T_{00} $i = 0, j = 0$

$w(0, 0) = q(0) = 2$

$c(0, 0) = 0$

$r(0, 0) = 0$

T_{11} $i = 1, j = 1$

$w(1, 1) = q(1) = 3$

$c(1, 1) = 0$

$r(1, 1) = 0$

T_{22} $i = 2, j = 2$

$w(2, 2) = q(2) = 1$

$c(2, 2) = 0$

$r(2, 2) = 0$

T_{33} $i = 3, j = 3$

$w(3, 3) = q(3) = 1$

$c(3, 3) = 0$

$r(3, 3) = 0$

T_{44} $i = 4, j = 4$

$w(4, 4) = q(4) = 1$

$c(4, 4) = 0,\ r(4, 4) = 0$

Scanned with CamScanner

Consider the set who's $j-i=1$

$T_{01}, T_{12}, T_{23}, T_{34}$

T_{01}

$$\boxed{w(i,j) = w(i,j-1) + P(j) + q(j)}$$

$w(0,1) = w(0,0) + P(1) + q(1) = 2+3+3 = 8$

$$\boxed{c(i,j) = \min_{i<k\le j} \{c(i,k-1) + c(k,j)\} + w(i,j)}$$

k value in b/w i & j

$c(0,1) = \min_{0<k\le 1} \{c(0,0) + c(1,1)\} + w(0,1) = 0+0+8 = 8$

$k = 1$

$r(0,1) = 1$

$$\boxed{r(i,j) = k}$$

T_{12}

$w(1,2) = w(1,1) + P(2) + q(2) = 3+3+1 = 7$

$c(1,2) = \min_{1<k\le 2} \{c(1,1) + c(2,2)\} + w(1,2) = \min\{0\} + 7 = 7$

$k = 2$

$r(1,2) = 2$

T_{23}

$w(2,3) = w(2,2) + P(3) + q(3) = 1+1+1 = 3$

$c(2,3) = \min_{2<k\le 3} \{c(2,2) + c(3,3)\} + w(2,3) = \min\{0+0\} + 3 = 3$

$k = 3$

$r(2,3) = 3$

T_{34}

$w(3,4) = w(3,3) + P(4) + q(4) = 1+1+1 = 3$

$c(3,4) = \min_{3<k\le 4} \{c(3,3) + c(4,4)\} + w(3,4) = 3$

$k = 4$

$r(3,4) = 4$

Scanned with CamScanner

Consider the set who's $j-i=2$

T_{02}, T_{13}, T_{24}

T_{02} $w(0,2) = w(0,1) + P(2) + q(2) = 8+3+1 = 12$

$$c(0,2) = \min_{\substack{0<k\le 2 \\ k=1,2}} \begin{cases} {}^{k=1}c(0,0) + c(1,2) \\ {}^{k=2}c(0,1) + c(2,2) \end{cases} + w(0,2)$$

Select min number

$$= \min \begin{Bmatrix} 7 \\ 8 \end{Bmatrix} + 12 = 19$$

$r(0,2) = 1$

T_{13}

$w(1,3) = w(1,2) + P(3) + q(3) = 7+1+1 = 9$

$$c(1,3) = \min_{\substack{1<k\le 3 \\ k=2,3}} \begin{cases} c(1,1) + c(2,3) \\ c(1,2) + c(3,3) \end{cases} + w(1,3)$$

$$= \min \begin{Bmatrix} 3 \\ 7 \end{Bmatrix} + 9 = 12$$

$r(1,3) = 2$

T_{24}

$w(2,4) = w(2,3) + P(4) + q(4) = 3+1+1 = 5$

$$c(2,4) = \min_{\substack{2<k\le 4 \\ k=3,4}} \begin{cases} c(2,2) + c(3,4) \\ c(2,3) + c(4,4) \end{cases} + w(2,4)$$

$$= \min \begin{Bmatrix} 3 \\ 3 \end{Bmatrix} + 5 = 8$$

$r(2,4) = 3$

Consider the set who's $j-i=3$

T_{03}

$w(0,3) = w(0,2) + P(3) + q(3) = 12+1+1 = 14$

$$c(0,3) = \min_{\substack{0<k\le 3 \\ k=1,2,3}} \begin{cases} c(0,0) + c(1,3) \\ c(0,1) + c(2,3) \\ c(0,2) + c(3,3) \end{cases} + w(0,3)$$

Scanned with CamScanner

$$= \min \begin{Bmatrix} 12 \\ 11 \\ 19 \end{Bmatrix} + 14 = 25$$

$r(0,3) = 2$

T_{14}

$w(1,4) = w(1,3) + P(4) + q(4) = 9+1+1 = 11$

$$c(1,4) = \min_{\substack{1<k\le 4 \\ k=2,3,4}} \begin{Bmatrix} c(1,1)+c(2,4) \\ c(1,2)+c(3,4) \\ c(1,3)+c(4,4) \end{Bmatrix} + w(1,4)$$

$$= \min \begin{Bmatrix} 8 \\ 10 \\ 12 \end{Bmatrix} + 11 = 19$$

$r(1,4) = 2$

consider the set whose $j-i=4$

T_{04}

$w(0,4) = w(0,3) + P(4) + q(4) = 14+1+1 = 16$

$$c(0,4) = \min_{\substack{0<k\le 4 \\ k=1,2,3,4}} \begin{Bmatrix} c(0,0)+c(1,4) \\ c(0,1)+c(2,4) \\ c(0,2)+c(3,4) \\ c(0,3)+c(4,4) \end{Bmatrix} + w(0,4)$$

$$= \begin{Bmatrix} 19 \\ 16 \\ 23 \\ 25 \end{Bmatrix} + 16 = 32$$

$r(0,4) = 2$

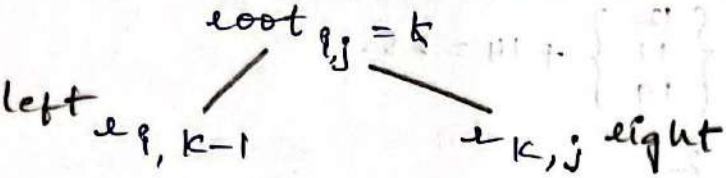
The table containing the values of $w(i,j)$, $c(i,j)$ & $r(i,j)$

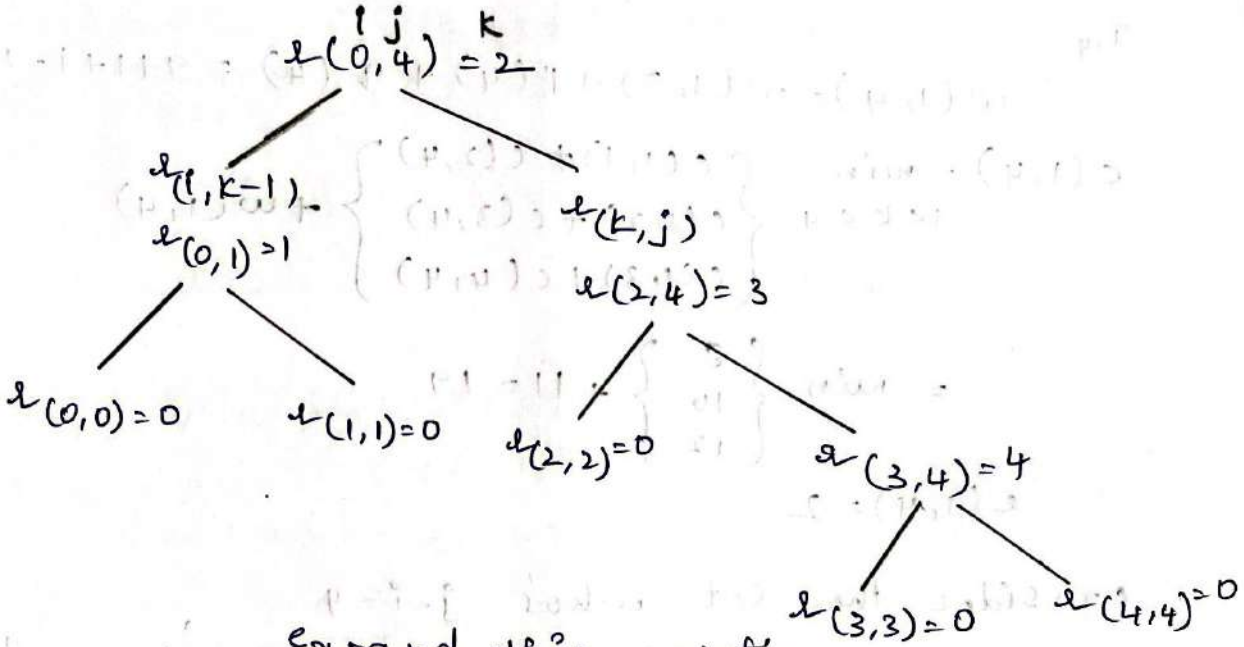
From the table, it can be seen that $c(0,4) = 32$ is the minimum cost of the binary search tree for (a_1, a_2, a_3, a_4)

The tree has root node $r(0,4) = 2$

The procedure for constructing of optimal binary search tree is,

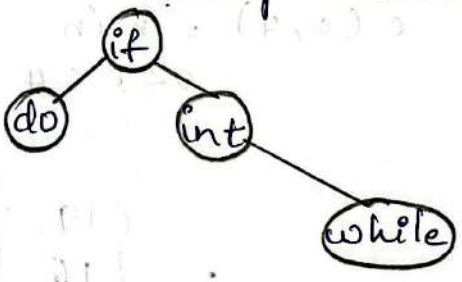
Scanned with CamScanner



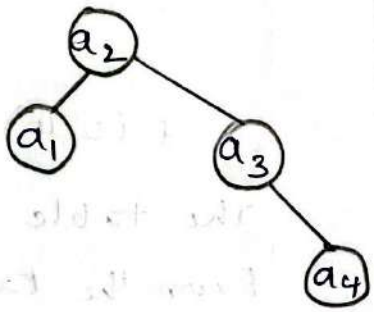


Expand this notation until all the leaf nodes becomes zero.

The optimal binary search tree



T00 w(0,0)=2 c(0,0)=0 r(0,0)=0	T01 w(0,1)=8 c(0,1)=8 r(0,1)=1	T02 w(0,2)=12 c(0,2)=19 r(0,2)=1	T03 w(0,3)=14 c(0,3)=25 r(0,3)=2	T04 w(0,4)=16 c(0,4)=32 r(0,4)=2
	T11 w(1,1)=3 c(1,1)=0 c(1,1)=0	T12 w(1,2)=7 c(1,2)=7 r(1,2)=2	T13 w(1,3)=9 c(1,3)=12 r(1,3)=2	T14 w(1,4)=11 c(1,4)=19 r(1,4)=2
		T22 w(2,2)=1 c(2,2)=0 r(2,2)=0	T23 w(2,3)=3 c(2,3)=3 r(2,3)=3	T24 w(2,4)=5 c(2,4)=8 r(2,4)=3
			T33 w(3,3)=1 c(3,3)=0 r(3,3)=0	T34 w(3,4)=3 c(3,4)=3 r(3,4)=4
				T44 w(4,4)=1 c(4,4)=0 r(4,4)=0



Scanned with CamScanner

0/1 Knapsack Using Dynamic Programming :-

In greedy Method, If the object is placed in the bag or knapsack, it is treated as 1, otherwise 0. and the partial object is also placed in the bag.

But, in dynamic programming partial object placing is not possible.

Here consider some objects & related weights & profits.

- we use the ordered set S^i.
- S^i represent the possible states resulting from the 2^i decision sequences for $x_1, x_2 \ldots x_i$.
- Each member of S^i is a pair (P, W) where P represents profit & W represent total weight of objects included in knapsack.
- Initially $S^0 = \{0, 0\}$
- we compute S^{i+1} from S^i
- To obtain S^{i+1}, we note that the possibilities for x_{i+1} are $x_{i+1} = 0$ or $x_{i+1} = 1$.
- when $x_{i+1} = 0$ the resulting state's are same as for S^i.
- when $x_{i+1} = 1$ the resulting states are obtained by adding (P_{i+1}, W_{i+1}) to each state in S^i.
- call the set of these additional states as S_1^i.
- Now S^{i+1} can be computed by merging the states in S^i and S_1^i together.
- If S^{i+1} contains two pairs (P_j, W_j) & (P_k, W_k) with the property that $P_j \le P_k$ and $W_j \ge W_k$, then the pair (P_j, W_j) can be discarded. This is called 'purging' rule also known as dominance rule.

(don't place the object which is having more weight & less profit)

Scanned with CamScanner

Here dominated tuples can be pueged.
ie (P_K, W_K) dominates (P_j, W_j)

* The worst case time complexity is $O(2^n)$.
The best case time complexity is $O(N*W)$
N- no of items, W- capacity of knapsack.

Ex:- Consider the knapsack instance

$n=4, m=21, \quad P=(2,5,8,1) \quad W=(10,15,6,9)$

Sol :-

Initially $i=0$ and $S^i = \begin{pmatrix} P & W \\ 0 & 0 \end{pmatrix}$

$$\boxed{S^{i+1} = S^i \cup S_1^i}$$ to get next set

Now calculate S_1^0.

For this take 1st object profit & weight. Add to S^0 pair. ie $(0,0)$

$(2,10) \quad S_1^0 = \{(0+2), (0+10)\} = \{2, 10\}$

$$S^{0+1} = S^0 \cup S_1^0 = \{(0,0), (2,10)\}$$

After finding each set S^0 or S^1 or S^2 or S^3 apply pueging rule & discard some pairs by using formula

ie, (P_1, W_1) (P_2, W_2) two pairs.

if $(P_1 \le P_2)$ & $(W_1 \ge W_2)$ then discard first pair.
else keep all the pairs as it is.

calculate : S^2, S^3, S^4

$$S^{i+1} = S^i \cup S_1^i$$

$S_1^1 = \{(0+5, 0+15), (2+5, 10+15)\} = \{(5,15), (7,25)\}$

$S^2 = \{(0,0), (2,10), (5,15), (7,25)\}$

Scanned with CamScanner

∴ Here remove the pair (7,25), why because, the weight of the object is greaterthan the knapsack weight.

$S^2 = \{(0,0), (2,10) (5,15)\}$

$(8,6) S_1^2 = \{(0+8, 0+6), (2+8, 10+6), (5+8, 15+6)\}$

$= \{(8,6), (10,16), (13,21)\}$

$S^3 = S^2 \cup S_1^2 = \{(0,0) (2,10) (5,15) (8,6) (10,16) (13,21)\}$

Here remove the pairs $(2,10)$ $(5,15)$ because of purging rule. The weight of the object is greaterthan the $(8,6)$

$2 < 8, 10 > 6$; $5 < 8, 15 > 6$

$S^3 = \{(0,0), (8,6), (10,16), (13,21)\}$

$(1,9) S_1^3 = \{(0+1, 0+9) (8+1, 6+9) (10+1, 16+9) (13+1, 12+9)\}$

$= \{(1,9) (9,15) (11,25) (14,30)\}$

$S^4 = \{(0,0) (8,6) (10,16) (13,21) (1,9) (9,15) (11,25) (14,36)\}$

. Here remove $(11,25)$ $(14,30)$. Moethan the knapsack weight.

$S^4 = \{(0,0) (8,6) (10,16) (13,21) (1,9) (9,15)\}$

Select the pair which have weight $m = 21$

$(13,21) \in S^4$ Yes

also check $(13,21) \in S^3$ Yes

consider S^3 only.

On solution vector select 3rd object and reject 4th object.

Substract $(13,21)$ from $(8,6)$ 3rd object

$(13-8, 21-6) = (5,15) \in S^2$

$(5,15)$ is present in S^2. Again check $(5,15) \notin S^1$

$x_2 = 1$

Substract $(5,15)$ from 2nd object $(5,15)$

Scanned with CamScanner

$(5-5, 15-15) = (0,0) \in S^1$ Yes

$\in S^0$ Yes

$\therefore x_4 = 0$

Final solution vector is $(x_1, x_2, x_3, x_4) = (0, 1, 1, 0)$

profit $= 5+8 = 13$

Ex 2:-

$M = 8$, $n = 4$, profits $= \{1, 2, 5, 6\}$ weight $= \{2, 3, 4, 5\}$

Sol:-

$$S^{i+1} = S^i \cup S_1^i$$

Initially $S^0 = \{0,0\}$

$S_1^0 = \{(0+1), (0+2)\}$

$S_1^0 = \{1, 2\}$

$S^1 = S^0 \cup S_1^0 = \{(0,0), (1,2)\}$

Apply pueging rule

$0 < 1$, $0 < 2$. No change

$S_1^1 = \{(0+2, 0+3), (1+2)(2+3)\}$

$= \{(2,3), (3,5)\}$

$S^2 = \{(0,0)(1,2)(2,3), (3,5)\}$

No change

$S_1^2 = \{(0+5, 0+4), (1+5, 2+4)(2+5, 3+4)$

$(3+5, 5+4)\}$

$S_1^2 = \{(5,4)(6,6)(7,7)(8,9)\}$

$\times$ Maximum weight exceeded

$S_1^2 = \{(5,4)(6,6)(7,7)\}$

Scanned with CamScanner

$S^3 = \{(0,0)(1,2)(2,3)(3,5)(5,4)(6,6)(7,7)\}$

Apply pueging rule & remove (3,5)

$S_1^3 = \{(6,5)(7,7)(8,8)(9,10)(11,9)(12,10)(13,12)\}$

$= \{(6,5)(7,7)(8,8)\}$

$S^4 = \{(0,0)(1,2)(2,3)(5,4)(6,6)(7,7)(6,5)(7,7)(8,8)\}$

$S^4 = \{(0,0)(1,2)(2,3)(5,4)(7,7)(6,5)(7,7)(8,8)\}$

Solution vector $(x_1, x_2\ x_3\ x_4)$

consider (8,8) as it is max order pair

$(8,8) \in S^4$

$(8,8) \notin S^3$ $\quad \boxed{x_4 = 1}$

4th object (6,5).

$(8-6, 8-5) = (2,3)$

$(2,3) \in S^3$

$(2,3) \in S^2$

$(2,3) \notin S^1$ $\quad \boxed{x_2 = 1}$

2nd object (2,3)

$(2-2, 3-3) = (0,0)$

$(0,0) \in S^3$

$(0,0) \in S^2$

$(0,0) \notin S^1$

$(x_1\ x_2\ x_3\ x_4) = (0,1,0,1)$

Scanned with CamScanner

Algorithm:-

```
Algorithm DKP(p, w, n, m)
{
  S^0 = {(0, 0)};
  for i=1 to n-1 do
  {
    S1^(i-1) = {(P, W) | (P-Pi, W-Wi) ∈ S^(i-1) and W ≤ m};
    S^i = MergePurge(S^(i-1), S1^(i-1));
  }
  (PX, WX) = last pair in S^(n-1);
  (PY, WY) = (P' + Pn, W' + Wn) where W' is the largest W in
             any pair in S^(n-1) such that W + Wn ≤ m;
  // Trace back for xn, x(n-1) --- x1.
  if (PX > PY) then xn = 0;
    else xn = 1;
    TraceBackfor(x(n-1) --- x1);
}
```

Scanned with CamScanner

All pairs Shortest Paths :-

Let $G = \{V, E\}$ be a directed graph with n vertices. Let cost be a cost adjacency matrix for G such that cost $(i, i) = 0$, $1 \le i \le n$.

Then (i, j) is the length of edge (i, j) if $(i, j) \in E(G)$ and cost $(i, j) = \infty$ if $i \neq j$ and $(i, j) \notin E(G)$.

The all pairs shortest path problem is to determine a matrix A such that $A[i, j]$ is the length of shortest path from i to j.

The matrix A can be obtained by solving 'n' single source problems using the algorithm Single Source shortest path in greedy approach.

Since each application of this procedure requires $O(n^2)$ time, the matrix A can be obtained in $O(n^3)$ time.

We obtain an alternate $O(n^3)$ solution to this problem using the principle of optimality.

Principle of Optimality :-

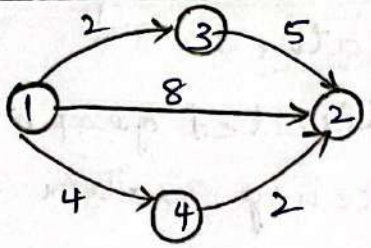
If K is the node on the shortest path from i to j then the path from i to K and K to j must also be shortest.

while considering k^{th} vertex as intermediate vertex, there are two possibilities.

- If k is not part of shortest path from i to j we keep distance $D[i, j]$ as it is.

If K is part of shortest path from i to j, update distance $D[i, j]$ as $D[i, K] + D[K, j]$

Scanned with CamScanner



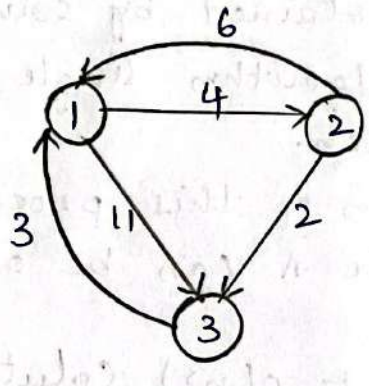
1 - 3 - 2 → 7
1 - 2 → 8
1 - 4 - 2 → 6

1 - 4 - 2 → 6
i K j

Optimal substructure of the problem is given as

$$A^K(i,j) = \min \{ A^{K-1}(i,j), A^{K-1}(i,K) + A^{K-1}(K,j) \}, \quad K \geq 1$$

Ex :-



Sol :-

Construct cost matrix A^0

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

There is no path from vertex V_3 to V_2. Therefore '∞' is placed.

From above cost matrix, we have to find other matrices A^1, A^2, A^3.

(no. of vertices are 3, hence A^1, A^2, A^3.)

$$A^K[i,j] = \min \{ A^{K-1}[i,j], A^{K-1}[i,K] + A^{K-1}[K,j] \}$$

A^K - no of iterations

The no of iterations depends on the no of vertices in the graph.

Scanned with CamScanner

A^1 : In A^1 matrix keep 1st row & 1st column as it is in A^0 and diagnol values also same.

$$A^1 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & - \\ 3 & - & 0 \end{bmatrix}.$$

We need to find $A^1(2,3)$ and $A^1(3,2)$ values using formula.

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

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

$= \min \{ 2,\ 6+11 \} = \min \{ 2, 17 \}$

Here minimum value is '2'. So, select '2'.

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

$$= \min \{ \infty,\ 3+4 \} = \min \{ 7 \}$$

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

From A^1 matrix, we get A^2 matrix. In A^1 keep 2nd row, 2nd column as it is & find remaining values.

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

$A^2(1,3)$ & $A^2(3,1)$:-

Scanned with CamScanner

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

$= \min\{11, 4+2\} = \min\{11, 6\} = \min\{6\}$

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

$= \min\{3, 7+6\} = \min\{3\}$

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

Now, find A^3 matrix from A^2 matrix. on A^2 keep 3rd row & 3rd column as it is & find remaining.

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

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

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

$= \min\{4, 6+7\} = 4$

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

$= \min\{6, 2+3\} = \{5\}.$

Final Matrix is,

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

From this, we verify any path from any vertex to any other vertex, we get shortest path.

Scanned with CamScanner

Algorithm :-

```
Algorithm Allpaths (cost, A, n)
// cost [1:n, 1:n] is the cost adjacency matrix of a
 graph with 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]);
    }
```

Scanned with CamScanner

Teavelling Sales Peeson Peoblem :-

In teavelling salesman peoblem the salesman has to visit eveey one of the cities staeting feom a ceetain city and to eeteen the same city. with minimum cost.

we will get the minimum cost or distance using a formula.

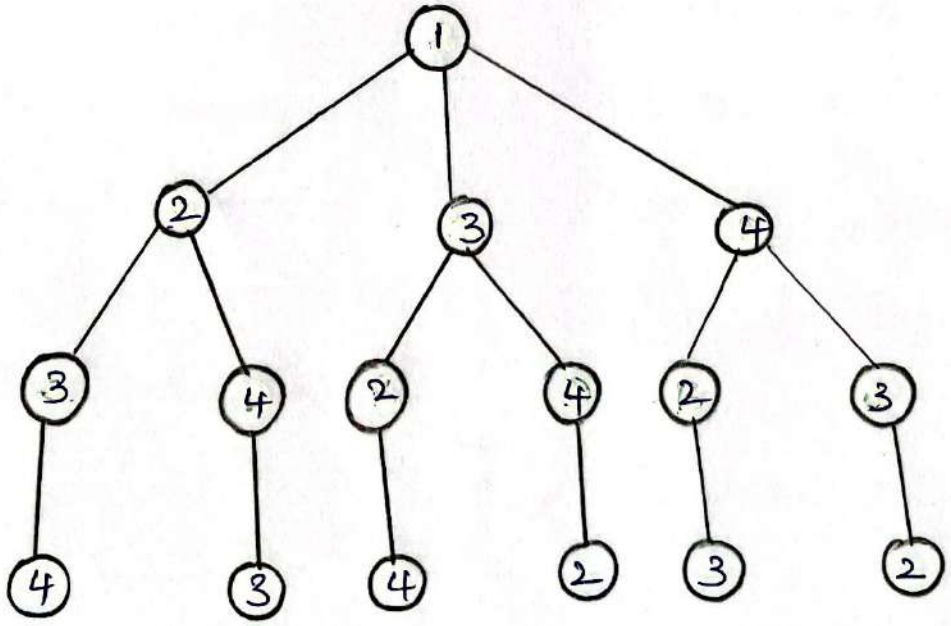
$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S - [j]) \}$$

'i' is a staeting veeten

S is a set of all eemaining veetices

j is a selected feom set S one by one.

In teavelling salesman peoblem staet at veeten A and visit all othee veetices and come back to same veeten A.



In the given geaph, consideo (1) is the soueee veetiex and eemaining veetices aee 2, 3, 4.

Ex :-

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Scanned with CamScanner

|S| represents no of elements

Initially $|S| = 0$ ($S = \phi$)

$g(i, S)$ $\quad i = 1$ to 4

$g(1, \phi) = 0$

$g(2, \phi) = 5$

$g(3, \phi) = 6$

$g(4, \phi) = 8$

$|S| = 1$

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

$i = 2$, $j = S = 3$

$$g(2, \{3\}) = \min \{ c_{23} + g(3, \{3\} - 3) \}$$

$$= \min \{ 9 + g(3, \phi) \} = \min \{ 9 + 6 \} = 15$$

$$g(2, \{4\}) = \min \{ c_{24} + g(4, \{4\} - 4) \}$$

$$= \min \{ 10 + g(4, \phi) \} = \min \{ 10 + 8 \} = 18$$

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

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

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

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

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

$|S| = 2$

$$g(2, \{3, 4\}) = \min \begin{cases} j=3 & c_{23} + g(3, \{3, 4\} - 3) \\ j=4 & c_{24} + g(4, \{3, 4\} - 4) \end{cases}$$

$$= \min \begin{cases} 9 + g(3, \{4\}) \\ 10 + g(4, \{3\}) \end{cases} = \min \begin{cases} 9 + 20 \\ 10 + 15 \end{cases} = \min \{25\}$$

$$g(3, \{2, 4\}) = \min \begin{cases} c_{32} + g(2, \{2, 4\} - 2) \\ c_{34} + g(4, \{2, 4\} - 4) \end{cases} = \min \begin{cases} 13 + g(2, \{4\}) \\ 12 + g(4, \{2\}) \end{cases}$$

$$= \min \begin{cases} 13 + 18 \\ 12 + 13 \end{cases} = \min \{25\}$$

Scanned with CamScanner

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

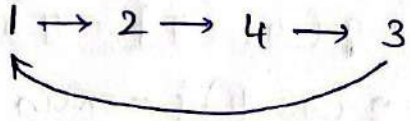
$$= \min \begin{Bmatrix} 8 + g(2, \{3\}) \\ 9 + g(3, \{2\}) \end{Bmatrix} = \min \begin{Bmatrix} 8+15 \\ 9+18 \end{Bmatrix} = \min\{23\}$$

$|S| = 3$

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

$$= \min \begin{Bmatrix} 10 + g(2, \{3,4\} \\ 15 + g(3, \{2,4\} \\ 20 + g(4, \{2,3\} \end{Bmatrix} = \min \begin{Bmatrix} 10+25 \\ 15+25 \\ 20+23 \end{Bmatrix} = \min\{35\}$$

Path is



cost is $10 + 10 + 9 + 6 = 35$

Time complexity is $O(n^2 2^n)$.

Scanned with CamScanner

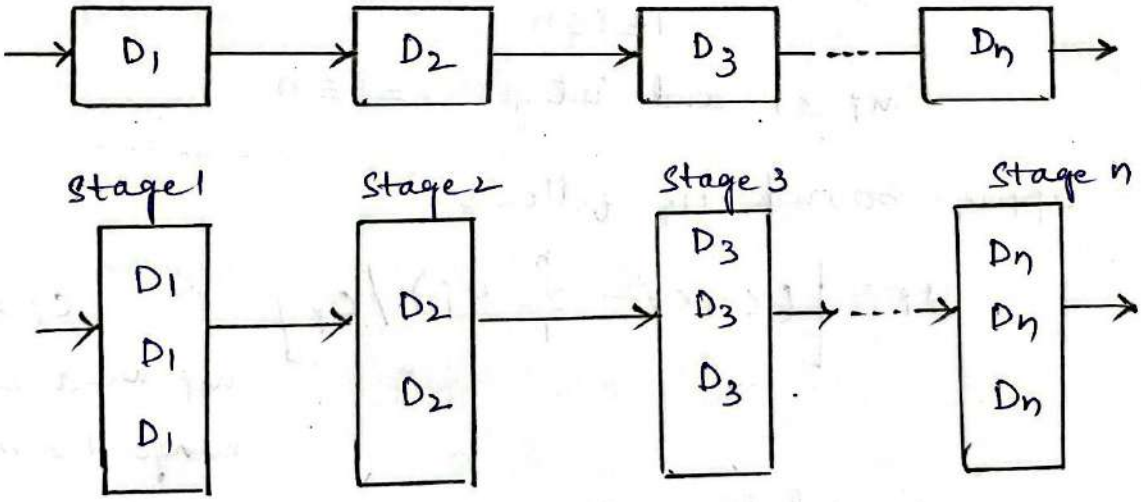
Reliability Design :-

It is used 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 Πr_i.

Even if the individual devices are very reliable, the reliability of the system may not be very good. Hence it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel through the use of switching circuits.

The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.



If Stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1-r_i)^{m_i}$. Hence the reliability of stage i becomes $1-(1-r_i)^{m_i}$.

$$\phi_i(m_i) = 1-(1-r_i)^{m_i}$$

Scanned with CamScanner

In any practical situation, the stage reliability ~~becomes~~ is a little less than $1-(1-r_i)^{m_i}$ because the switching circuits themselves are not fully reliable.
Also, failures of copies of same device may not be fully independent.
Let us assume that reliability of stage i is given by a function $\phi_i(m_i)$, $1 \le n$. The reliability of the system of stages is $\pi_{1 \le i \le n} \phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint.
Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. Maximization problem is,

$$\text{maximize } \pi_{1 \le i \le n} \phi_i(m_i)$$

$$\text{subject to } \sum_{1 \le i \le n} c_i m_i \le c$$

$$m_i \ge 1 \text{ and integer}, 1 \le i \le n$$

upper bound u_i follows

$$u_i = \left[(c + c_i - \sum_{1}^{n} c_j) / c_i \right] \qquad c_i > 0$$

m_i must be in the range $1 \le m_i \le u_i$.

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

Scanned with CamScanner

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

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

The cost of 3 devices are

$$c_1 = 30, \quad c_2 = 15, \quad c_3 = 20$$

reliabilities of 3 devices are

$$r_1 = 0.9, \quad r_2 = 0.8, \quad r_3 = 0.5$$

Upper bound at each stage is

$$u_i = \left[\left(c + c_j - \sum_{j=1}^{n} c_j \right) / c_i \right]$$

upperbound for device 1 is

$i=1$ $\quad u_1 = \left(105 + 30 - \sum_{j=1,2,3} (30+15+20)\right) / 30$

$= (135 - 65) / 30 = 70/30 \simeq 2$ 2 copies of device 1

Upperbound for device 2 is

$i=2$ $\quad u_2 = (105 + 15 - (30+15+20)) / 15 = (120 - 65) / 15$

$\simeq 3$ 3 copies of device 2

Upperbound for device 3 is

$i=3$ $\quad u_3 = (105 + 20 - (30+15+20)) / 20$

$= (125 - 65) / 20 = 60/20 = 3$

3 copies of device 3

Scanned with CamScanner

In dynamic programming, S^i consists of tuples like (r, c) r - reliability, c - cost.

$$r_i(c) = \max_{1 \le m_i \le u_i} \{ \phi_i(m_i)\, r_{i-1}(c - c_i m_i) \}$$

Initially $S^0 = \{1, 0\}$

we can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together.

$S_j^i \Rightarrow$ i represents stage no
j represents no of copies of device.

Using S_j^i to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$. Compute $S^0 \& S^1$

- Device 1 has 2 copies hence find S_1^1, S_2^1

$\underline{S_1^1} \Rightarrow i=1, j=1, m_i = j = 1$

$$\phi_i(m_i) = \phi_1(m_1) = 1 - (1 - r_1)^{m_1} = 1 - (1-0.9)^1 = 0.9$$

$$c_i m_i = c_1 m_1 = 30 \times 1 = 30$$

$$S_1^1 = \{0.9, 30\}$$

$S_2^1 \Rightarrow i=1, j=2, m_1 = 2$

$$\phi_i(m_i) = \phi_1(m_1) = 1 - (1-0.9)^2 = 0.99$$

$$c_1 m_1 = 30 \times 2 = 60$$

$$S_2^1 = \{0.99, 60\}$$

$$S^1 = S_1^1 \cup S_2^1 = \{(0.9, 30)\ (0.99, 60)\}$$

Scanned with CamScanner

Device 2 has 3 copies, hence find

S_1^2, S_2^2, S_3^2

$\underline{S_1^2} \Rightarrow \quad i=2,\ j=1,\ m_2=1$

$\phi_i(m_i) = 1-(1-r_2)^{m_2} = 1-(1-0.8)^1 = 1-0.2 = 0.8$

$c_2 m_2 = 15 \times 1 = 15$

$S_1^2 = (0.8, 15)$

Multiply these values with s' values to get final set S_1^2.

$S_1^2 = \{(0.8 \times 0.9, 30+15), (0.99 \times 0.8, 60+15)\}$

$= \{(0.72, 45), (0.792, 75)\}$

$\underline{S_2^2} \Rightarrow \quad i=2,\ j=2,\ m_2=2$

$\phi_i(m_i) = 1-(1-r_2)^{m_2} = (1-(1-0.8)^2) = 0.96$

$c_2 m_2 = 15 \times 2 = 30.$

$S_2^2 = \{0.96, 30\}$

$S_2^2 = \{(0.9 \times 0.96, 30+30), (0.99 \times 0.96, 60+30)\}$

$= \{(0.864, 60), (0.95, 90)\}$

$\underline{S_3^2} \Rightarrow \quad i=2,\ j=3,\ m_3=3$

$\phi_i(m_i) = 1-(1-r_2)^{m_2} = 1-(1-0.8)^3 = 0.992$

$c_2 m_2 = 15 \times 3 = 45$

$S_3^2 = \{(0.9 \times 0.992, 30+45), (0.99 \times 0.992, 60+45)\}$

$= \{0.8928, 75), (0.98208, 105)\}$

$S^2 = S_1^2 \cup S_2^2 \cup S_3^2$

Scanned with CamScanner

$S^2 = \{(0.72, 45), (0.792, 75)$
$(0.864, 60), (0.95, 90)$
$(0.8928, 75), (0.98208, 105)\}$

By checking dominance rule & cost, eliminate some pairs.

* $(0.792, 75)$ $(0.8928, 75)$ pairs have same cost & remove the pair with low reliability.

$(0.792, 75)$ is removed.

* After removing the pairs have the costs are in the order of 45, 60, 90, 75, 105

The cost should be in descending order, but $90 < 75$ fails. Remove the pair $(0.9504, 90)$

$S^2 = \{(0.72, 45), (0.864, 60)(0.8928, 75)(0.98208, 105)\}$

Compute S^3 :-

S^3 as it has 3 copies, find S_1^3, S_2^3, S_3^3.

$S_1^3 \Rightarrow \quad i = 3, \; j = 1, \; m_j = j \Rightarrow m_3 = 1$

$S_1^3 = \phi_3(m_3) = 0.5 \;, \quad C_3 m_3 = 20 \times 1 = 20 \Rightarrow (0.5, 20)$

$S_1^3 = \{(0.72 \times 0.5, 45+20), (0.864 \times 0.5, 60+20),$
$(0.8928 \times 0.5, 75+20), (0.98208 \times 0.5, 105+20)\}$

$= \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

remove $(0.98208 \times 0.5, 105+20)$ pair, because cost exceeds 105.

$S_2^3 = \phi_i(m_j) \Rightarrow i = 3, \; j = 2, \; m_3 = 2$

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

$C_3 m_3 = 20 \times 2 = 40$

$S_2^3 = (0.75, 40)$

Scanned with CamScanner

$S_2^3 = \{(0.72 \times 0.75, 45+40), (0.864 \times 0.75, 60+40),$
$(0.8928 \times 0.75, 75+40), (0.98208 \times 0.75, 105+40)\}$

Here 2 pairs removed cost > 105

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

$S_3^3 \Rightarrow \quad i = 3, j = 3, m_3 = 3$

$S_3^3 = \phi_i(m_i) = 0.875$

$c_3 m_3 = 20 \times 3 = 60$

$(r, c) = (0.875, 60)$

$S_3^3 = \{(0.72 \times 0.875, 45+60), (0.864 \times 0.875, 60+60),$
$(0.8928 \times 0.875, 60+75), (0.98208 \times 0.875, 105+60)\}$

remove 3 pairs, cost > 105

$S_3^3 = (0.72 \times 0.875, 105) = (0.63, 105)$

$S^3 = S_1^3 \cup S_2^3 \cup S_3^3$

$= \{(0.36, 65), (0.432, 80), (0.4464, 95), (0.54, 85),$
$(0.648, 100), (0.63, 105)\}$

In (0.4464, 95) & (0.54, 85) $\Rightarrow$ the reliability is low & cost is high in (0.4464, 95). Hence remove the pair.

(0.63, 105) is compared with (0.648, 100),
(0.63, 105) pair is low reliable. Then remove the pair

After removing

$S^3 = \{(0.36, 65) (0.432, 80) (0.54, 85) (0.648, 100)\}$

From S^3, we select last pair which has best reliability with cost 100 < 105.

$(0.648, 100) \in S_2^3 \quad \Rightarrow m_3 = 2$

Scanned with CamScanner

Hence 3rd device required 2 copies. ⇒ 20×2=40

On Stage 2, Observe Stage 2 where 60 cost is present ⟩ 100-40=60

$S^2 = (0.864, 60) \in S_2^2 \Rightarrow m_2 = 2$

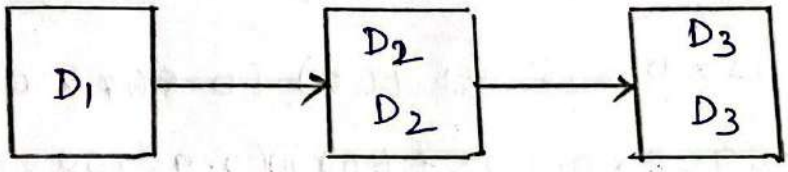
Device 2 also required 2 copies.

Now cost is 60 − stage1 cost(30) = 30 ⇒ 60-30=30

check where cost 30 is present in Stage 1

$S' = (0.9, 30) \in S_1'$

Hence device 1 required only 1 copy.



Scanned with CamScanner

UNIT - IV

Greedy Method

The Greedy method is a most powerful & straight forward technique in designing an algorithm. A wide variety of problems can be solved using this technique.

The drawback of divide & conquer technique, which is rectified in greedy Method. Divide & conquer technique is applicable only for problems, which can be divisible. There exist some problems which can't be divisible.

Problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a 'feasible solution'.
We need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an 'optimal solution'.
Every feasible solution need not be an optimal solution, but every optimal solution is a feasible solution. Every problem will have a unique optimal solution.

The greedy method operates in stages by considering single input at a time. It makes sequences of choices which is selected that appear best at that moment.
If the inclusion of the next input into the partially

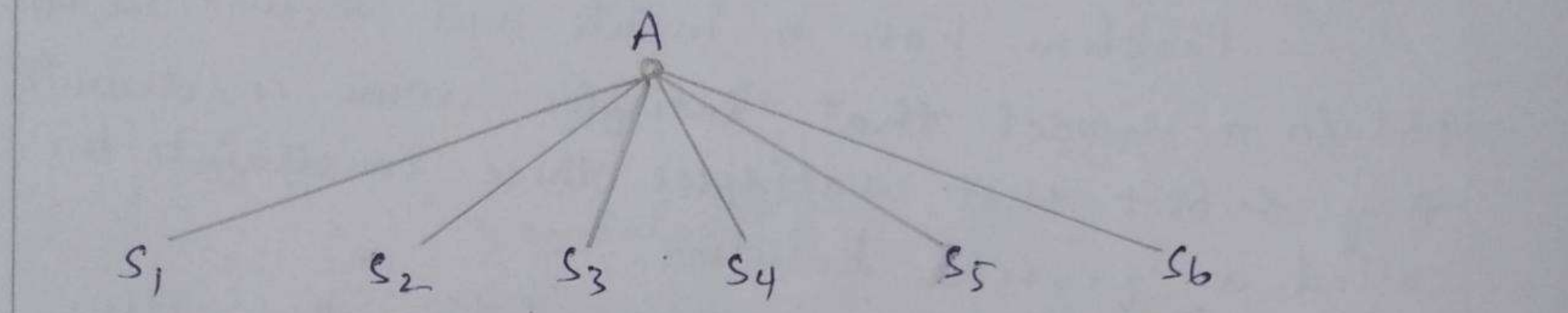
Scanned by TapScanner

constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution. Otherwise, it is added.

- The selection procedure itself is based on some optimization measure. This version of the greedy method is called "subset paradigm".

Example :-

Suppose there is a problem P and that problem is, person 'A' wants to travel location 'X' (Hyd) to 'Y' (Delhi). There may be more solution, Let us say,



But their is a constraint in the problem, that is, covered the journey within 12 hours.

By the constraint, by walk, bus, car, bike not possible to travel within the time, only possible solutions are by train or flight.

For the problem, there are many solutions but these solutions which are satisfying the condition given in the problem. These type of solutions are 'feasible solutions'.

If the person covers the journey with minimum cost (train). Then this is called as optimal solution.

Scanned by TapScanner

For any problem, there can be only one 'optimal solution', there cannot be multiple solutions.

- If the problem requires either minimum or maximum result then we call that type of problem as 'optimization Problem'.

Control Abstraction Algorithm for Greedy Method:-

```
Algorithm Greedy(a, n)
// a[1:n] contains the n inputs
{
    solution = ϕ; // Initialize the solution
    for i=1 to n do
    {
        x = Select(a);
        if Feasible(solution, x) then
            solution = Union(solution, x);
    }
    return solution;
}
```

- The function 'Select' selects an input from a[] removes it. The selected input's value is assigned to x.
- 'Feasible' is a Boolean-valued function that determines whether x can be included into the solution vector.
- The function 'union' combines x with the solution & updates the objective function.
- For problems that don't call for the selection of

Scanned by TapScanner

an optimal subset, for the greedy method, we make decision by considering the inputs in some order.
- Each decision is made using an optimization criterion that can be computed using decisions already made.
- The version of greedy method is called 'ordering paradigm'.

Greedy Method Time complexity :-

The algorithm may be implemented to have complexity $O(n \log n)$.

Applications of Greedy Method :-

1. Job sequence with deadlines
2. 0/1 knapsack problem
3. Minimum cost Spanning Tree
4. Single source shortest path problem.

Scanned by TapScanner

Job Sequence with Deadlines :-

We are given a set of n jobs. Associated with job 'i' is an integer deadline $d_i \geq 0$ and a profit $P_i > 0$. For any job i the profit P_i is earned iff the job is completed by its deadline.

To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.

The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} P_i$.

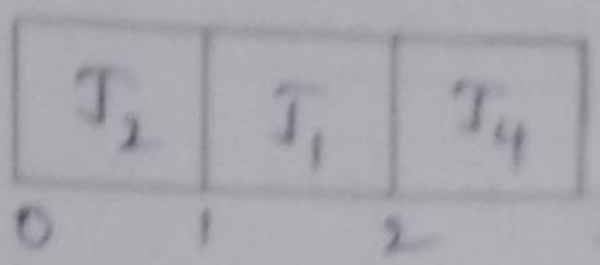
An optimal solution is a feasible solution with maximum value.

Ex :- Let n = 5

	J_1	J_2	J_3	J_4	J_5
profits (P_i)	20	15	10	5	1
Deadlines (D_i)	2	2	1	3	3

Sol :- In the problem maximum deadline is '3'.
Only 3 jobs are completed within the deadline.
- we will select the jobs, in the order of their maximum profit. If the profits are not in the descending order, arrange the profits in descending order.
- J_1 having highest profit & deadline is 2, arrange J_1 in the 1 to 2 slot.

Scanned by TapScanner



- J_2 having next highest profit and deadline is 2. On 1 to 2 time slot, already J_1 is placed. So J_1 is placed in 0 to 1 slot.
- Next J_3 deadline is 1. J_3 not having time slot & it is not feasible solution, hence discard it.
- Now J_4 deadline is 3, arrange J_4 in timeslot 2-3.
- There is no slot available for Job.

Job sequence is J_1 to $J_2 \rightarrow J_4$ or,

$J_2 \rightarrow J_1 \rightarrow J_4$: 20+15+5=40

Profit is - 40.

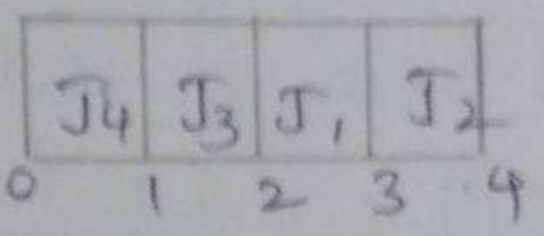
Job	Feasible solution or. Slots	processing Sequence	profit $\sum_{i \in J} P_i$
-	-	ϕ	0
J_1	[1, 2]	J_1	20
J_2	[0, 1] [1, 2]	J_1, J_2	20+15
J_3 (Rejected)	[0, 1] [1, 2]	J_1, J_2	35
J_4	[0, 1] [1, 2] [2, 3]	J_1, J_2, J_4	20+15+5
J_5 (Slot not available)	[0, 1] [1, 2] [2, 3]	J_1, J_2, J_4	40

Scanned by TapScanner

Ex2:- n = 7

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
profits	35	30	25	20	15	12	5
deadline	3	4	4	2	3	1	2

Sol:-



Job	Slot or, Feasible Solution	Solution	profit
J_1	[2, 3]	J_1	35
J_2	[2, 3] [3, 4]	J_1, J_2	35+30
J_3	[1, 2] [2, 3] [3, 4]	J_1, J_2, J_3	35+30+25
J_4	[0, 1] [1, 2] [2, 3] [3, 4]	J_1, J_2, J_3, J_4	35+30+25+20
J_5	[0, 1] [1, 2] [2, 3] [3, 4]	J_1, J_2, J_3, J_4	35+30+25+20
J_6	[0, 1] [1, 2] [2, 3] [3, 4]	J_1, J_2, J_3, J_4	35+30+25+20
J_7	[0, 1] [1, 2] [2, 3] [3, 4]	J_1, J_2, J_3, J_4	110

Job Sequence is $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$.

Scanned by TapScanner

Algorithm :-

Algorithm JS(d, j, n) (d: deadlines, j: Jobs, n: no of jobs)

// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that P[1] ≥ P[2] ≥ ... ≥ P[n].
// J[i] is the ith job in the optimal solution, 1 ≤ i ≤ k.
// At termination d[J[i]] ≤ d[J[i+1]], 1 ≤ i < k

{ d[0] = J[0] = 0; // Initialize (d[0] = deadline array, J[0] = job sequence array)
J[1] = 1; // include job 1

k = 1; no of jobs taken
for i = 2 to n do // 1st job we have taken already
{
// consider jobs in nonincreasing order of P[i].
// Find position for i and check feasibility of
insertion.

r = k; deadlines highest
while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do (check the current job with existing job whether the shifting is required or not)
r = r - 1;
if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then (shift existing job down words. check the feasibility the job can be inserted or not)
// insert i into J[].
for q = k to (r+1) step -1 do J[q+1] = J[q];
time slot J[r+1] = i; → insert the job in sequence
k = k + 1; - increase the job count
}
}
return k;
}

Scanned by TapScanner

	J_1	J_2	J_3	J_4	J_5
P	20	15	10	5	1
d	2	2	1	3	3

Algorithm logic :-

$d[0] = 0,\ J[0] = 0$

$J[1] = 1$

$k = 1,\ r = 2$

deadline $i = k \Rightarrow i = 1$

deadline of J_2

of J_1 $d[J[i]] > d[i]$ and $d[J[i]] \neq i$ — Note: On the strength of 1st condition if false, 2nd condition is not checked. Two conditions must be true in 'and' statement

$2 > 2$ false.

loop will never execute

There is no chance to shift the deadline.

if $(d[J[i]] \leq d[i]$ and $d[i] > i)$

$2 \leq 2$ and $2 > 1$. True.

2nd job can be inserted. check whether the existing job shifted up or down.

for $q = 1$ to $i+1$ step-1

$q = 1$ to 2

$J[i+1] = 2 \Rightarrow J[2] = 2$ insert the job 2 in sequence.

$\rightarrow k = 2$, $i = 3$, $i = k \Rightarrow i = 2$

$d[J[2]] > d[3]$ and $d[J[2]] \neq 2$

$2 > 1$ and $2 \neq 2$ false

loop will not execute.

If $((d[J[2]] \leq d[3]$ and $d[3] > 2)$ then

$2 \leq 1$ condition False.

- Insertion not possible. The job is not feasible.
- comeout of the loop

Scanned by TapScanner

→ i = 4, p = 2, K = 2

while (d[J[2]] > d[4] and d[J[2]] ≠ 2)

2 > 3 false.

if (d[J[2]] ≤ d[4] and d[4] > 2)

2 ≤ 3 and 3 > 2. True

q = 2 to 3 Step - 1

loop cannot be executed. Shifting is not required.

J[2+1] = 4 Job 3 is inserted in 4th slot.

K = 3, i = 5, q = 3

while (d[J[3]] > d[4] and d[J[3]] ≠ 3)

4 > 3 false

if (d[J[3]] ≤ d[5] and d[5] > 3)

3 ≤ 3 and 3 > 3 . False

return K = 3.

Scanned by TapScanner

Knapsack problem :-

In the knapsack problem, we are given n objects & a knapsack or bag. object i has a weight w_i and the knapsack has a capacity m.

If a fraction x_i, $0 \le x_i \le 1$, of object 'i' is placed into the knapsack, then a profit of $P_i x_i$ is earned. The resulting profit has to be maximum.

We require the total weight of all chosen objects to be at most m. The problem can be stated as,

$$\text{maximize}_{\text{profit}} \sum_{1 \le i \le n} P_i x_i$$

$$\text{subject to}_{\text{weight}} \sum_{1 \le i \le n} w_i x_i \le m$$

The profits & weights are positive numbers.

Ex :- Consider the following instance of the knapsack problem, $n = 3$, $m = 20$, $(P_1, P_2, P_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

Sol :-

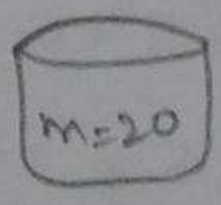
Knapsack problem is to find the most valuable subset of items that fit into the knapsack.

$n = 3$, $m = 20$.

$\frac{P_i}{w_i} = \frac{25}{18}, \frac{24}{15}, \frac{15}{10}$

$= 1.3, 1.6, 1.5$

x_i	0	1	1/2
	not placed		1/2 placed



20 - 15 = 5

5 - 5 = 0

(If 18 kg weight is placed in the bag, we will get only 25 profit. Here we are calculating profit per kg & objects ni fractional gaa divide chesi bag lo place cheyavachu) Ex: veg 1/2, 1/2, 1/2

Scanned by TapScanner

$\sum x_i w_i = 0 \times 18 + 1 \times 15 + \frac{1}{2} \times 10 = 20$

$\sum x_i P_i = 0 \times 25 + 1 \times 24 + \frac{1}{2} \times 15 = 31.5$

Optimal solution knapsack vector is $0 \leq x \leq 1$

- On this method, the list of items are divisible, which means any fraction of the item can be considered

Ex 2 :- $n = 7, \; m = 15$

	1	2	3	4	5	6	7
profit (P_i)	10	5	15	7	6	18	3
weight (w_i)	2	3	5	7	1	4	1

Sol :-

$\frac{P_i}{w_i}$						
$\frac{10}{2}$	$\frac{5}{3}$	$\frac{15}{5}$	$\frac{7}{7}$	$\frac{6}{1}$	$\frac{18}{4}$	$\frac{3}{1}$
5	1.3	3	1	6	4.5	3

Highest profit object is 5, place the object in bag.

- object 1 is placed, next
- object 6, object 3, object 7, placed
  object 2.$\frac{2}{3}$, object 4 is not placed

in the bag.

15 - 1 = 14
14 - 2 = 12
12 - 4 = 8
8 - 5 = 3
3 - 1 = 2
2 - 2 = 0

$\frac{P_i}{w_i}$	5	1.3	3	1	6	4.5	3
	1	$\frac{2}{3}$ (only 2kg)	1	0	1	1	1
	x_1	x_2	x_3	x_4	x_5	x_6	x_7

$\sum x_i w_i = 1 \times 2 + \frac{2}{3} \times 3 + 1 \times 5 + 0 \times 7 + 1 \times 1 + 1 \times 4 + 1 \times 1$

$= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$

$\sum x_i P_i = 1 \times 10 + \frac{2}{3} \times 5 + 1 \times 15 + 1 \times 6 + 1 \times 18 + 1 \times 3$

$= 10 + 2 \times 1.3 + 15 + 6 \times 18 + 3 = 54.6$

Scanned by TapScanner

```
Algorithm GreedyKnapsack(m, n)
// P[1:n] and w[1:n] contain the profits & weights
   respectively of the n objects ordered such that
   P[1]/w[1] ≥ P[1+1]/w[1+1]. m is the knapsack
    size and x[1:n] is the solution vector.
    {
      for i=1 to n do x[1]= 0.0 // Initialize x
      U= m; maximum weight
bag capacity for i=1 to n do
                    1 > 20
          { if (w[i] > U) then break;
              x[i] = 1.0;
               U= U- w[i];
                   20-15= 5
          }
        if (i ≤ n) then x[i]= U/w[1];
    }
```

P_i	25	24	15
w_i	18	15	10
$\frac{P_i}{w_i}$	1.3	1.6	1.5

m=20, n=3

U=20

Initially

x[1]=0.0

x[2]=0.0

x[3]=0.0

i=1

w[1] > 20

15 > 20 false

x[1] =1.0

U= 20-15-5

i= 2

w[2] > 5

10 > 5 True

break & comes out of the loop

2 <= 3

i=2

n=3 $x[2] = \frac{5}{10} = 0.5$

x[1]=1

x[2]=0.5

x[3]=0

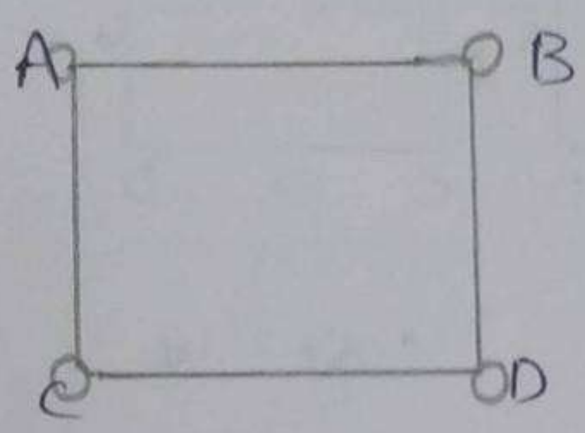
Scanned by TapScanner

Minimum cost Spanning Tree :-

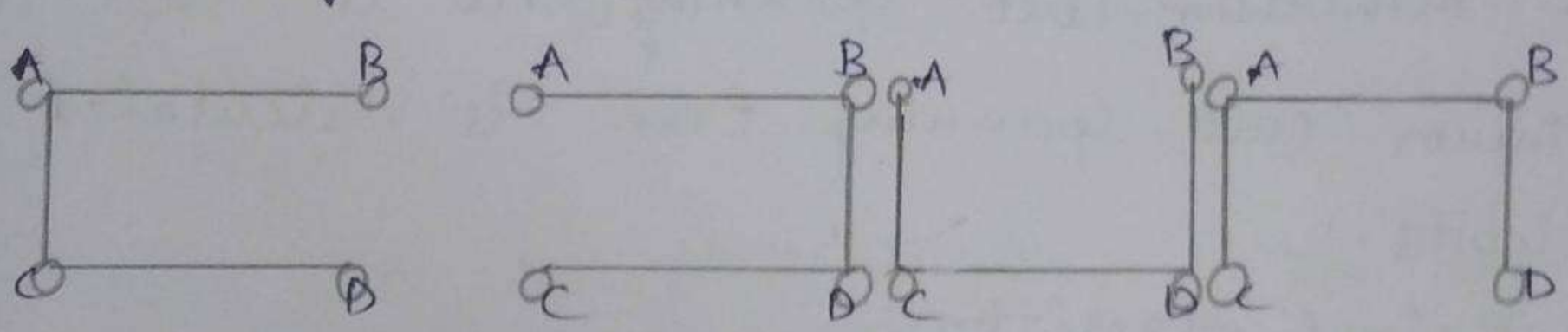
Spanning Tree is a sub graph of the given graph. It must satisfies three properties.

1. It should contain all the vertices of the graph.
   (On the given graph, no of nodes must be equal to spanning tree nodes)
2. If the graph contains 'n' vertices, then the spanning tree should contains (n-1) edges.
3. Spanning tree should not contain any cycle.

Ex :-



4 Spanning trees for above graph.



3 conditions satisfied.

- All spanning trees contain same no of vertices (A, B, C, D)
- Graph contains 4 vertices, spanning tree also contains (4-1=3) edges.
- Spanning trees not contain any cycle

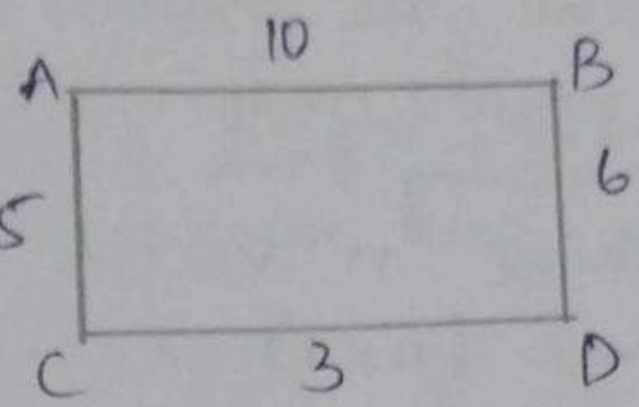
Applications :-

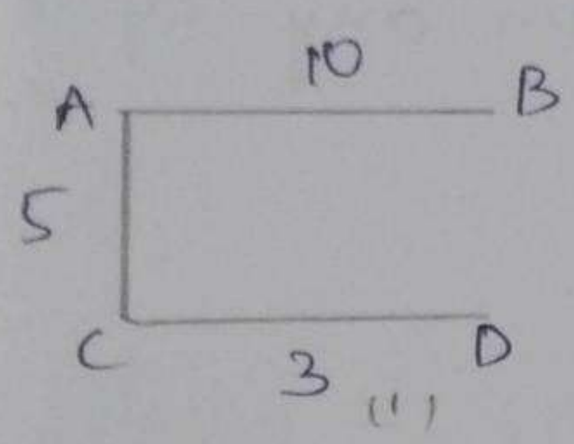
- It is useful in constructing networks
- It is used to find the airlines routes.

Scanned by TapScanner

- A spanning tree in which the cost is minimum then the remaining spanning tree. It is said to be minimum cost spanning tree.

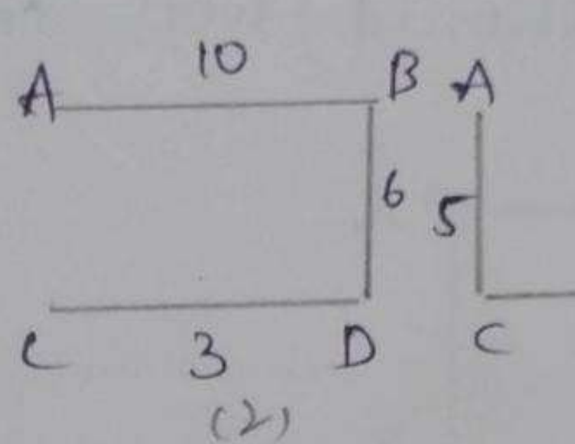
Ex:-





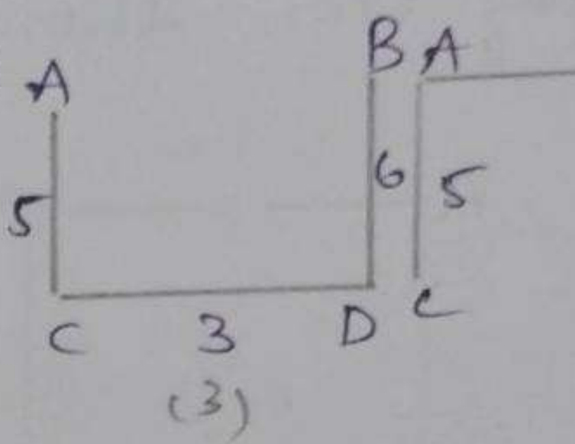
(1)

cost of the spanning tree-18



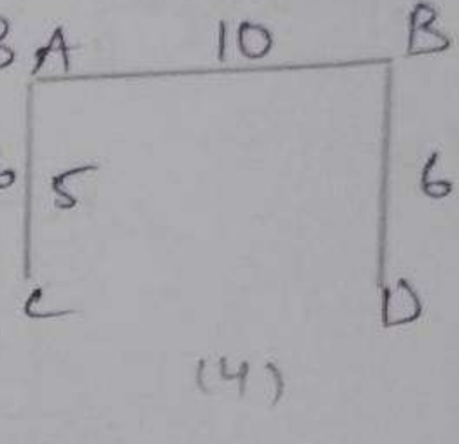
(2)

cost - 19



(3)

cost - 14



(4)

cost - 21.

∴ Minimum cost spanning tree is cost=14. (fig3)

Minimum cost spanning tree is calculated by 2 methods.

1, Prim's Algorithm

2, Kruskal's algorithm.

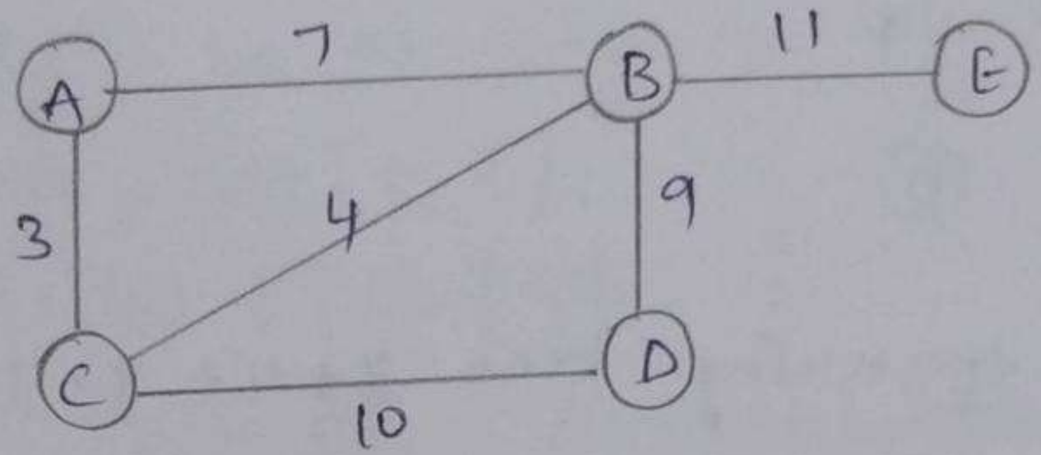
Scanned by TapScanner

Prim's Algorithm :-

This algorithm mainly used to calculate minimum cost spanning tree. Steps,

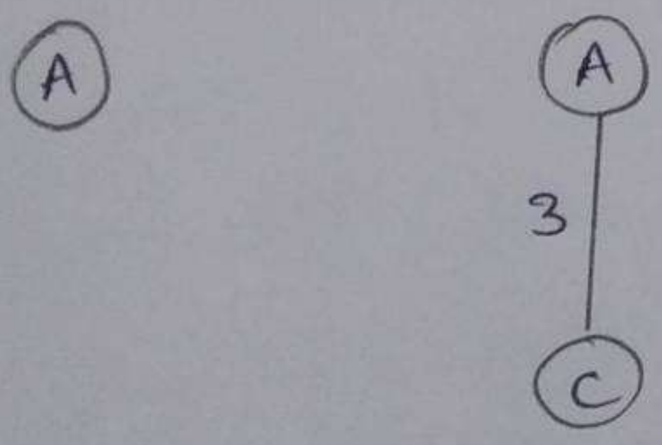
1, Start with any vertex of the graph.
2, Find the edges associated with that vertex and add minimum cost edge to the spanning tree, if it not forming any cycle. It is forming any cycle then discard the edge.
3, Repeat the step 2, untill all the vertices are covered.

Ex :-



Sol :- Step 1 :-

Start with the first vertex A. The edge weights of this vertex are compared ie. 3 is compared with 7. Since, 3 is smaller than 7, the edge (A, C) is selected.

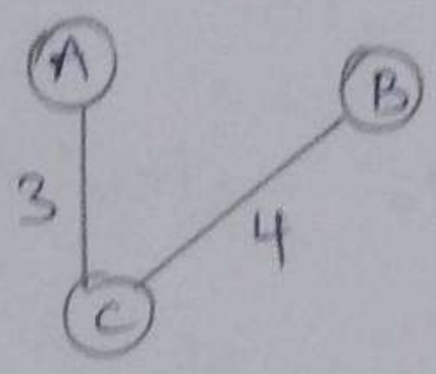


A-B - 7
A-C - 3 ✓

Step 2 :-

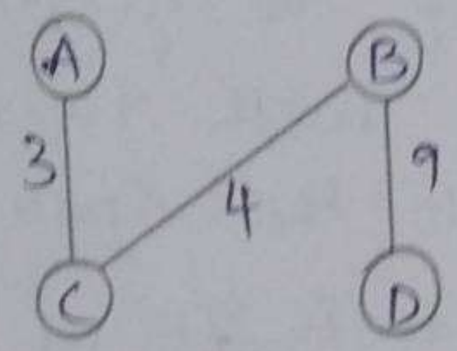
Now, consider the vertex 'C'. The edge weights of this vertex are compared and the edge with the smallest weight is selected. ie, edge (C, B)

Scanned by TapScanner



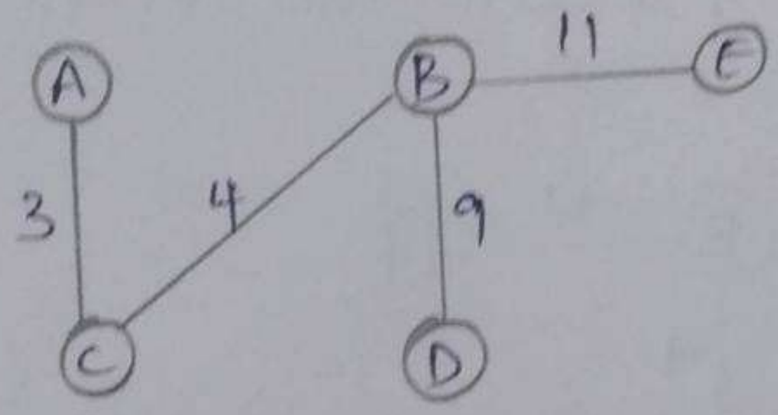
C-A-3 (Already constructed)
C-B-4 ✓
C-D-10

Step 3 :-



B-C-4 (constructed)
B-D-9 ✓
B-E-11
B-A-7 (cycle)

Step 4 :-



cost of the spanning tree - 3+4+9+11 = 27

This spanning tree satisfies all the conditions of the spanning tree.

The resulting graph obtained is a minimum spanning tree in which the comparisons made by the tree algorithm are 4 in total, for n=5 vertices. (n-1)=(5-1) comparisons are done.

Scanned by TapScanner

Algorithm:-

Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1:n, 1:n] is the cost adjacency matrix of an n vertex graph such that cost[i, j] is either a positive real number or 'oo' if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array t[1:n-1, 1:2] (t[i, 1], t[i, 2]) is an edge in the minimum-cost spanning tree. The final cost is returned.

```
{
  Let (k, l) be an edge of minimum cost in E;
  mincost = cost[k, l];
  t[1, 1] = k; t[1, 2] = l;
  for (i = 1 to n do    // Initialize near.
  if (cost[i, l] < cost[i, k]) then near[i] = l;
  else near[i] = k;
  near[k] = near[l] = 0;
  for i = 2 to n-1 do
  { // Find n-2 additional edges for t.
    Let j be an index such that near[j] ≠ 0 and
      cost[j, near[j]] is minimum;
    t[i, 1] = j;
    t[i, 2] = near[j];
    mincost = mincost + cost[j, near[j]];
    near[j] = 0;
    for k = 1 to n do  // Update near[].
      if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
```

Scanned by TapScanner

```
        then near[k]=j;
      }
      return mincost;
    }
```

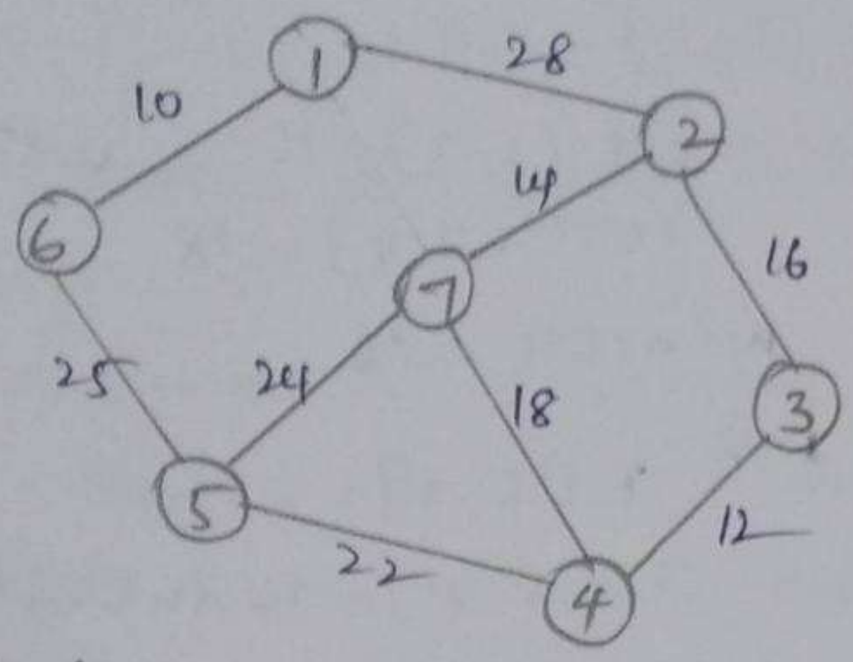
Time complexity :-

PRIM's Algorithm TM is

$O(n^2)$.

Scanned by TapScanner

Prim's Algorithm logic :-



k = l, l = 6

mincost = cost (1, 6) = 10 ✓

T[1, 1] = 1, T[1, 2] = 6

i = 1 to n

if (cost (1, 6) < cost (1, 1) then near [1] = 6

10 < ∞ then near [1] = 6 (True)

i = 2

cost (2, 6) < cost (2, 1) then near [2] = 6

∞ < 28. false

else near [2] = 1

i = 3 l = 6, k = 1

cost (3, 6) < cost (3, 1) then near [3] = 6

∞ < ∞ false

near [3] = 1 near [6] = 1

near [4] = 1 near [7] = 1

near [5] = 6

near [1] = near [6] = 0

for i = 2 to 7 - 1

near [j] ≠ 0 and cost [j, near [j]] is minimum;

j = 2, 3, 4, 5, 7

cost (2, 1) = 28 cost (7, 1) = ∞

cost (3, 1) = ∞

cost (4, 1) = ∞

cost (5, 6) = 25 ✓

minimum cost is (5, 6) = 25

i = 2, j = 5

T[2, 1] = 5, T[2, 2] = near [5] = 6

near [5] = 0

for k = 1 to n

if [near [k] ≠ 0] and (cost [k, near [k]) > cost (k, j))

k = 2, 3, 4, 7

cost (2, 1) > cost (2, 5)

28 > ∞. false

cost (3, 1) > cost (3, 5)

∞ > ∞. false.

cost (4, 1) > cost (5, 4)

∞ > 22. True

near [4] = 5

cost (7, 1) > cost (7, 5)

∞ > 24. True

near [7] = 5

i = 3

near [j] ≠ 0 and cost [j, near [j])

j = 2, 3, 4, 7.

cost (2, 1) = 28

cost (3, 1) = ∞

cost (4, 5) = 22 ✓ minimum cost

cost (7, 5) = 24

Now j = 4

T[3, 1] = 4

T[3, 2] = near [4] = 5

Mincost = mincost + cost [j, near [j]]

= 10 + 25 + 22

Near [4] = 0

k = 2, 3, 7

if (near [2] ≠ 0) and (cost [2, near [2]] > cost [2, 4])

28 > ∞

Scanned by TapScanner

cost [3, near [3] > cost [3,4])
cost (3,1) > cost [3,4]
∞ > 12 . True
near [3] = 4
cost [7, near [7]] > cost [7,4]
cost [7,5] > cost [7,4]
24 > 18. True.
near [7] = 4.

i=4
near [j] ≠ 0 and cost [j, near [j])
j = 2, 3, 7
cost (2,1) = 28
cost (3,4) = 12 ✓ mincost
cost (7,4) = 18
now j = 3
T [4,1] = 3
T [4,2] = near [3] = 4
Mincost = 10+25+22+12
near [3] = 0
for k=1 to n
if (near [k] ≠ 0) and (cost [k, near [k] > cost [k,j])
k = 2, 7
cost (2, near [2] > cost [2,3]
cost (2,1) > cost [2,3]
28 > 16 .. True
near [2] = 3
k = 7
cost (7, near [7]] > cost (7,3))
cost (7,4) > cost
18 > ∞. False

For i = 5
near [j] ≠ 0 and cost [j, near [j])
j = 2, 7
cost (2,3) = 16 ✓
cost (7,4] = 18
mincost = 16
j = 2
T [5,1] = 2
T [5,2] = near [2] = 3
Mincost = 10+25+22+12+16
near [2] = 0
for k=1 to n
k = 7
cost [7, near [7] > cost [7,2]]
cost [7,4] > cost [7,2]
18 > 17 . True.
near [7] = 2
i = 6
i = 2 to 6-1 = 5. True
near [j] ≠ 0 and cost [j, near [j])
j = 7
T [6,1] = 7
T [6,2] = near [7];
Mincost = 10+25+22+12+16+14
near [7] = 0
for k=1 to n do.
near [k] ≠ 0 and [cost [k, near [k]]
> cost [k,j]]
false.
go's to for i=2 to n-1 false.
It returns
min cost = 10+25+22+12+16+
14.

ing tree.
eight.
order
ted
e,
n

Scanned by TapScanner

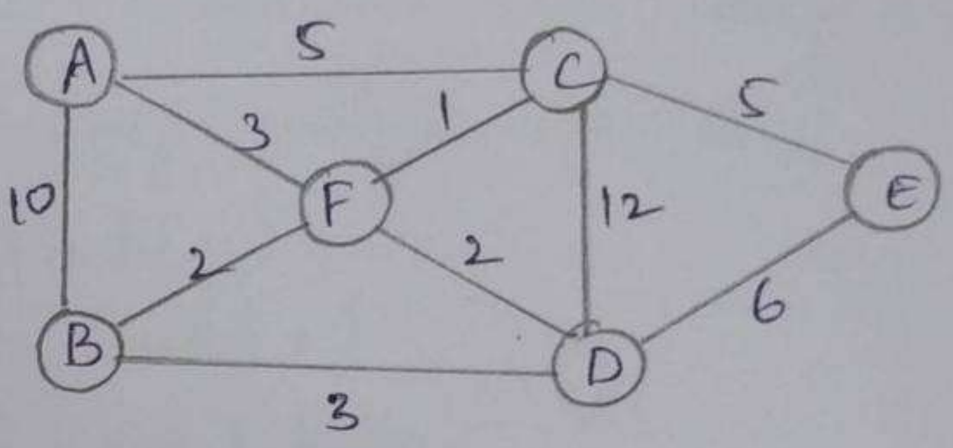
Kruskal's Algorithm :-

It is used to calculate minimum cost spanning tree. It chooses an edge in the geaph with minimum weight.

Steps :-

1. We have to arrange all the nodes in ascending order based on the cost.
2. Select minimum cost edge from the list of sorted edges and add that edge to the spanning tree, If it is not forming anycycle. If any edge form any cycle then discard that edge.
3. Repeat step 2 till all edges covered.

Ex :-



Sol :- Minimum cost edges arranged in ascending order

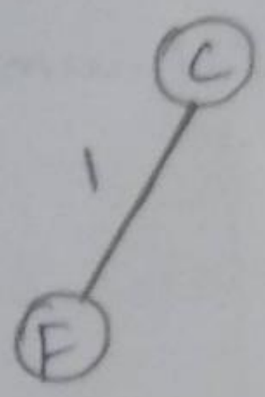
Step 1 :-

	Edge	cost
1.	C - F	1
2.	B - F	2
3.	D - F	2
4.	A - F	3
5.	B - D	3
6.	A - C	5
7.	C - E	5
8.	D - E	6
9.	A - B	10
10	C - D	12

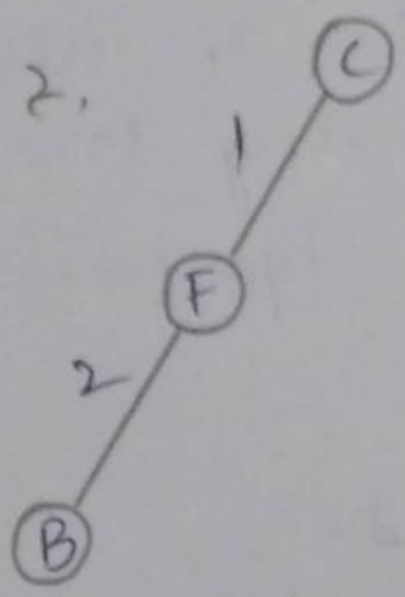
Scanned by TapScanner

Step 2 :- Select minimum cost edge & add that edge to spanning tree.

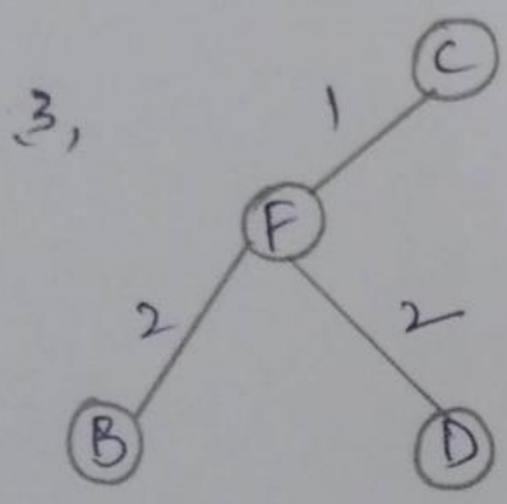
1,



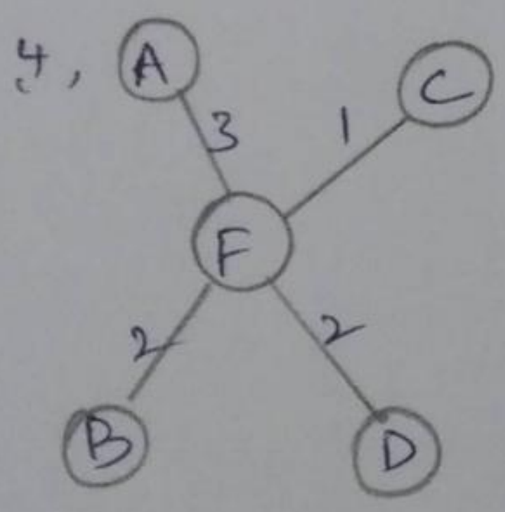
2,



3,



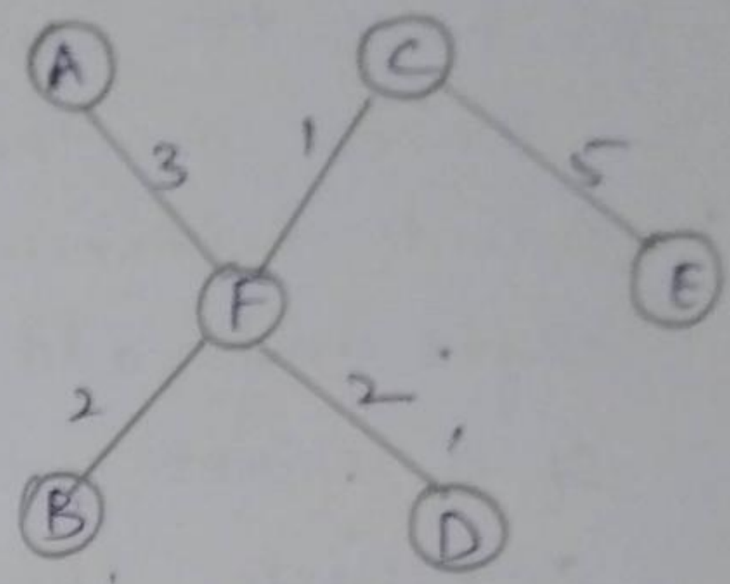
4,



5, If you add B-D edge it will form cycle. So discard the edge.

6, A-C, also formed cycle. discard it

7,



Cost = 13.

Scanned by TapScanner

Algorithm :-

```
Algorithm Kruskal (E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u,v]
   is the cost of edge (u,v). 't' is the set of edges in the
   minimum cost spanning tree. The final cost is returned
{ construct a heap out of the edge costs using Heapify;
   for i=1 to n do parent[i] = -1;
   // Each vertex is in a different set.
     i=0; mincost = 0.0;
while ((i < n-1) and (heap not empty)) do
{
   Delete a minimum cost edge (u,v) from the heap &
      reheapify using Adjust;
   j = Find(u);
   k = find(v);
   if (j ≠ k) then
   { i = i+1;
   t[i, 1] = u;
   t[i, 2] = v;
   mincost = mincost + cost[u,v];
   Union (j, k);
   }
}
if (i ≠ n-1) then write("No spanning tree");
   else return mincost;
}
```

Time complexity :- $O(E \log V)$

Scanned by TapScanner

Prim's Algorithm	Kruskal's Algorithm
1, This method starts with a single vertex of a graph similar to tree & gradually adds the smallest edge to make the tree to grow by one more vertex.	1, It starts with each & every vertex of the graph & adds the smallest edge that joins the two trees as the whole.
2, It is suitable when only a single tree & only a few edges are taken into consideration at a time.	2, Kruskal's algorithm is suitable when all the edges & vertices are simultaneously taken into consideration.
3, considered as more space efficient since only one tree is taken & the edges that connect to the vertices in the tree are examined.	3, considered as time-efficient because ordering of edges as per their weights is possible for traversing them quickly.
4, It consumes more time.	4, It consumes more space.
5, The tree that we are making or growing always remains connected.	5, The tree that we making or growing usually remains disconnected.
6, It is faster for dense graphs.	6, It is faster for sparse graphs.

Scanned by TapScanner

Single Source Shortest Path Problem:-

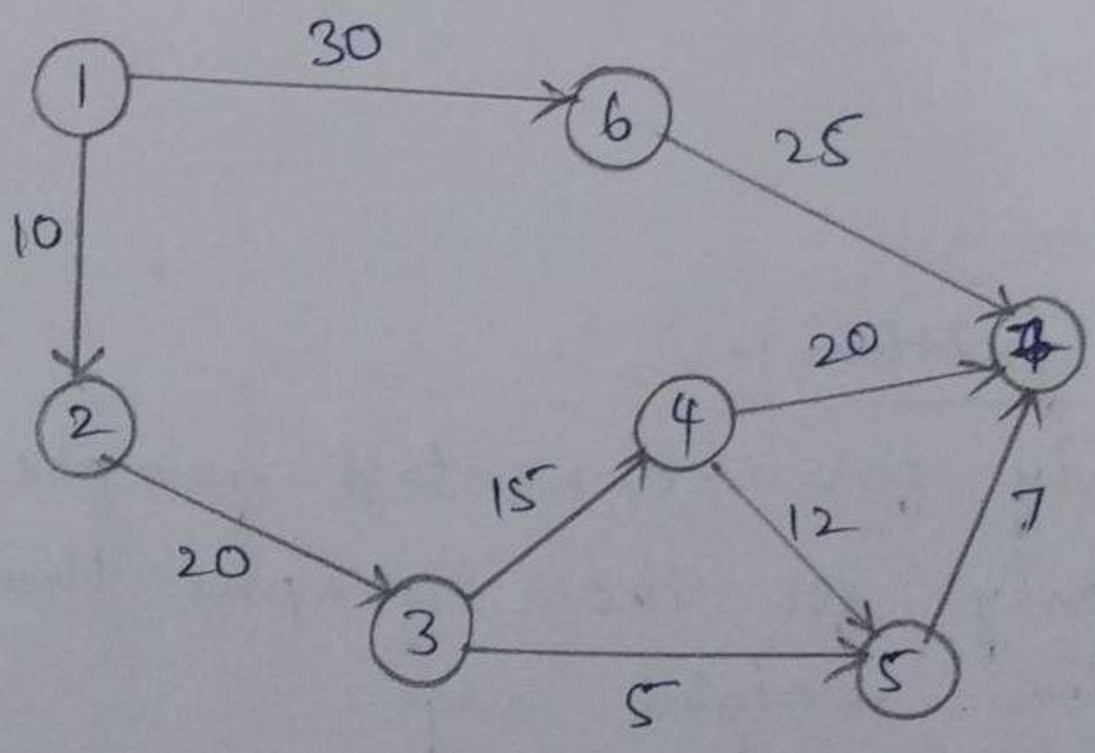
The single source shortest path can be defined as a problem wherein shortest path between source vertex & other vertices is identified.

The most important algorithm for solving this problem,

Dijkstra's algorithm:-

Dijkstra's single source shortest path algorithm finds the shortest path from a given source to all remaining nodes of a digraph.

In each stage of computation, the nodes of the graph associates themselves with labels. Some labels in this are permanent which represents the shortest path from source node to that node. At the experimental level, the source node is usually set as a permanent label & its value is kept zero.



Let V = {1, 2, 3, 4, 5, 6, 7}

Scanned by TapScanner

Selected vertex	Visited set	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]
1	{1}	(10)	∞	∞	∞	30	∞
2	{1,2}	(10)	(30)	∞	∞	30	∞
3	{1,2,3}	(10)	(30)	45	35	(30)	∞
6	{1,2,3,6}	(10)	(30)	45	(35)	(30)	65
5	{1,2,3,6,5}	(10)	(30)	45	(35)	(30)	(42)
7	{1,2,3,6,5,7}	(10)	(30)	45	(35)	(30)	(42)
4	{1,2,3,6,5,7,4}	(10)	(30)	(45)	(35)	(30)	(42)

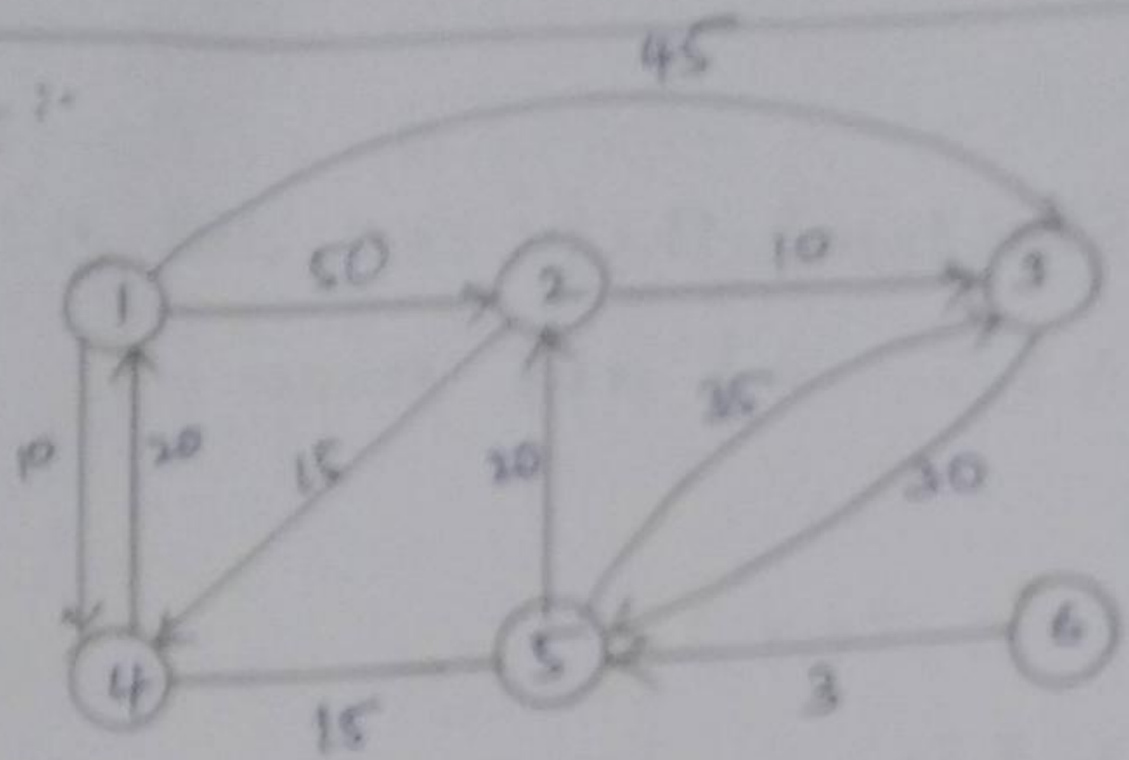
Vertex	Cost
2	10
3	30
4	45
5	35
6	30
7	42

Dijkstra's Algorithm :-

- It works only for connected graphs.
- It works only for those graphs that do not contain any negative weight edge.
- It only provides the value or cost of the shortest paths
- It works for directed as well as undirected graphs.

Scanned by TapScanner

Ex 2 :-



V = {1, 2, 3, 4, 5, 6}

Selected vertex	Visited set	2	3	4	5	6
1	{1}	50	45	(10)	∞	∞
4	{1, 4}	50	45	(10)	(25)	∞
5	{1, 4, 5}	(45)	45	(10)	(25)	∞
2	{1, 4, 5, 2}	(45)	45	(10)	(25)	∞
3	{1, 4, 5, 2, 3}	(45)	45	(10)	(25)	∞
6	{1, 4, 5, 2, 3, 6}	(45)	(45)	(10)	25	∞

Vertex	Cost
2	10
3	30
4	45
5	35
6	30
7	42

Scanned by TapScanner

Algorithm :-

Algorithm ShortestPaths (v, cost, dist, n)

Let V = {1, 2, ... n} and source = 1

S = {1}

for (i = 2; i <= n; i++)

D[i] = c[1, j];

for (i = 1; i <= n-1; i++)

{

Choose a vertex w ∈ V-S such that D[w] is a minimum;

S = S ∪ {w};

for each vertex v ∈ V-S

D[v] = min(D[v], D[w] + c[w, v])

}

Time complexity :-

- Single - source shortest path algorithm takes $O(n^2)$ time.

Scanned by TapScanner

Example 2, Algorithm Teacking:

V = {1, 2, 3, 4, 5, 6}

S = {1}

i = 2;

D[2] = C[1,2] = 50

D[3] = C[1,3] = 10

D[4] = C[1,4] = ∞

D[5] = C[1,5] = 45

D[6] = C[1,6] = ∞

- shortest value

for i=1, ω = 2, 3, 4, 5, 6

ω = 3

S = {1,3}

V = {2, 4, 5, 6}

D[2] = min{D[2], D[3] + C[1,2]} = min{50, 10 + ∞} = 50

D[4] = min{D[4], D[3] + C[3,4]} = min{∞, 10 + 15} = 25

D[5] = min[D[5], D[3] + C[3,5]} = min{45, 10 + ∞} = 45

D[6] = min[D[6], D[3] + C[3,6]} = min{∞, 10 + ∞} = ∞

i = 2, ω = 2, 4, 5, 6

ω = 4

S = {1, 3, 4}

V = {2, 5, 6}, D[2] = min{D[2], D[4] + C[4,2]} = 45

D[5] = min{D[5], D[4] + C[4,5]} = 45

D[6] = min{D[6], D[4] + C[4,6]} = ∞

i = 3, ω = 2, 5, 6

ω = 5, S = {1, 3, 4, 5} ω = 2

Scanned by TapScanner

UNIT-V

Branch & Bound

Branch & Bound is the one of the problem solving strategy. It is similar to the back tracking since it also uses the state space tree. It is used for solving the optimization problem & only minimization, not maximization.

If we have given a maximization problem then we can convert it using the branch & bound. This is best when used for combinatorial problems with exponential time complexity.

Types of solutions :-

For B & B problem, there are two ways in which the solution can be represented,

1. Variable size solution :-

This provides the subset of the given set that gives the optimized solution for the given problem.

$\{A, B, C, D, E\} \rightarrow$ solution set is $\{A, B, E\}$

2. Fixed size solution :-

This solution is a sequence of 0's & 1's, with the digit at i^{th} position denoting whether the i^{th} element should be included or not.

$\{A, B, C, D, E\} \rightarrow$ solution set $\{1, 1, 0, 0, 1\}$

* The main difference between backtracking & Branch & Bound is, backtracking follows the DFS & B&B follows the BFS.

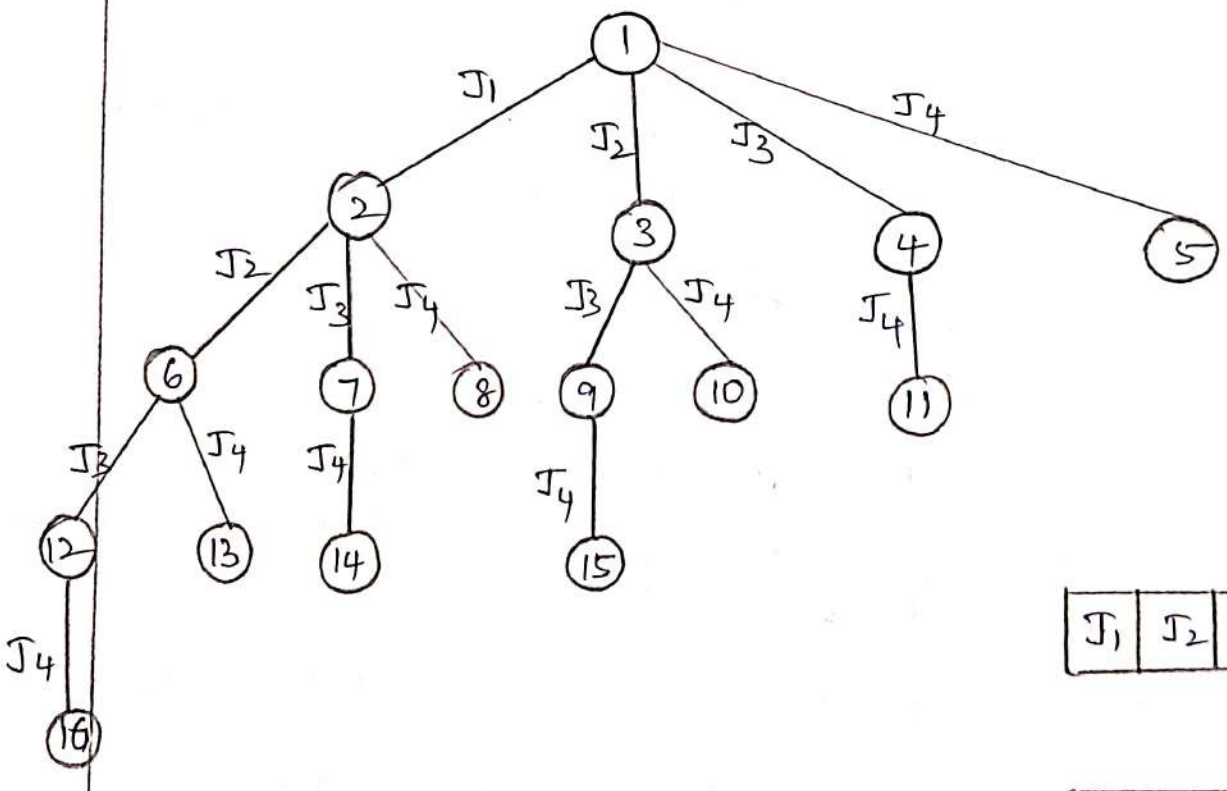
Types of Beanch & Bound :-

There are multiple types of Beanch & Bound methods based on the order in which the State Space tree is to be Seaech.
we will be using the variable solution method to denote the solutions in these methods.

1, FIFO (First In First Out) Beanch & Bound :-

It follows the queue appeoach in ceeating the State Space teee. Here BFS is peefoemed ie, the elements at a paeticulae level aee seaeched & then the elements of next level aee seaeched staeting from the first child of the first node in peevious level.

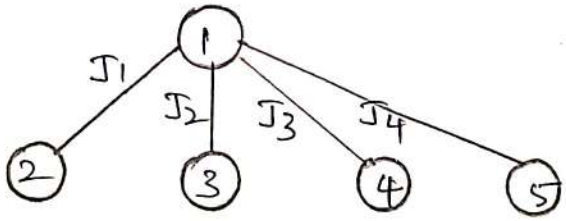
Ex :- Set $\{ J_1, J_2, J_3, J_4 \}$

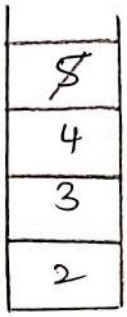


J_1	J_2	J_3	J_4

2, LIFO Beanch & Bound :-

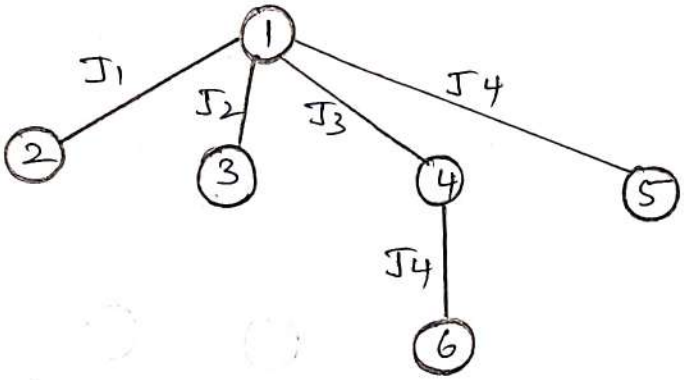
Last In fiest Out beanch & bound follows the stack approach in ceating state space tree.



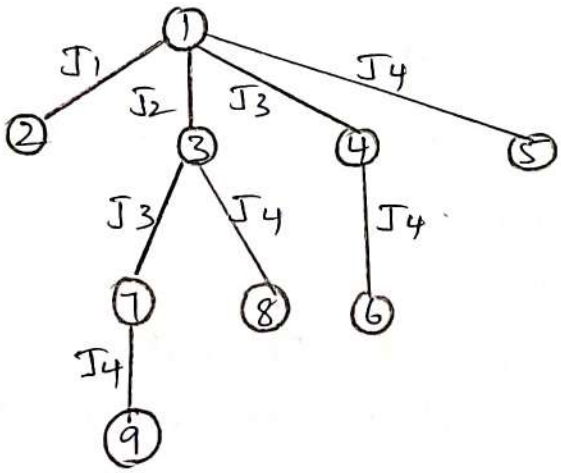


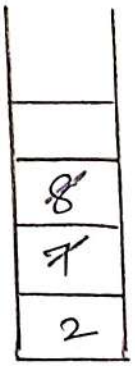
Top of the element is 5, expansion not possible, pop the element. only 4 jobs

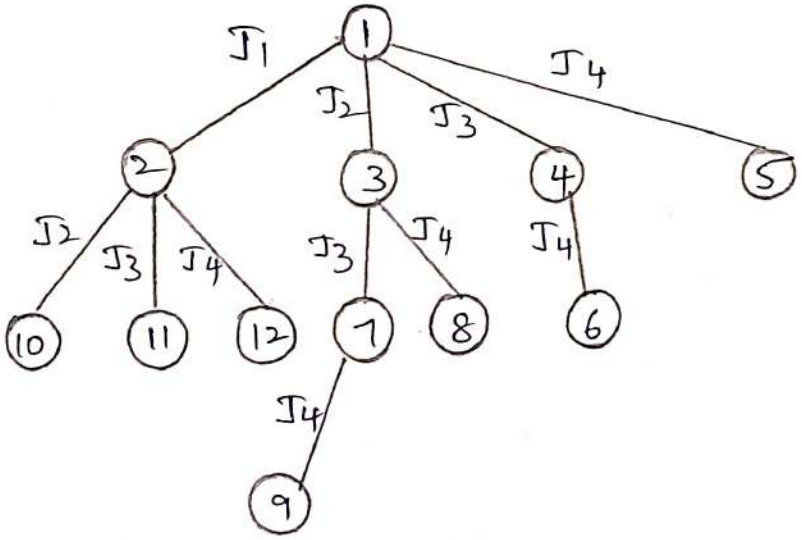
Now, top of the element is 4, expand it.



top of the element is 6. expansion not possible, pop the element. After poping next element is 3. expand the element

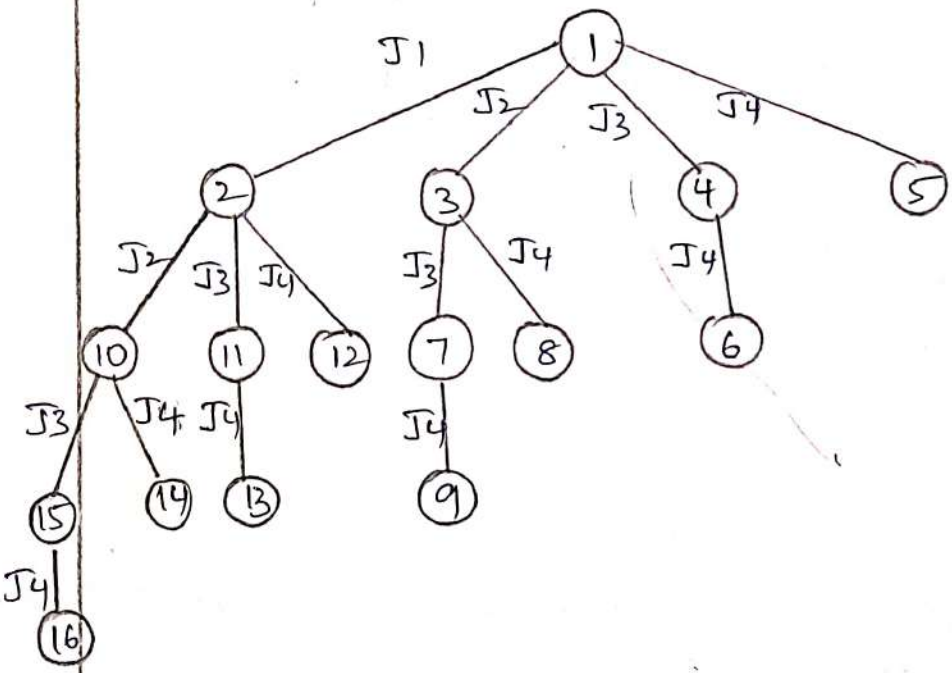






~~12~~
~~11~~
10

~~13~~
~~10~~

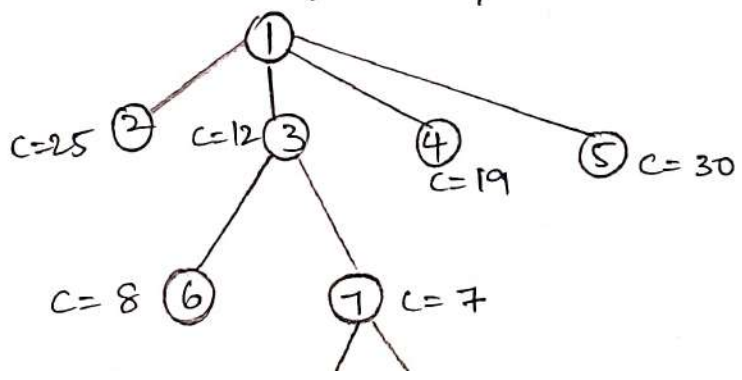


15
14

3. Least Cost Beanch & Bound :-

This method of exploration uses the cost function in order to explore State Space tree.

In this method, after the children of a particular node have been explored the next node to be explored would be that node out of unexplored nodes which has the least cost.



least cost node is explore

Beanch & Bound technique applied when the Greedy method & Dynamic peogeamming methods may fails.
- B&B is much slowes. It often leads to exponential time complexities in the woest case.
- Use of bounding function to limit the seaech. ie, to avoid the geneeation of sub tees that do not contain an answee node.
- Seaech techniques used in B&B : i, BFS ii, DFS iii, LCBB
- Two types of Bounds used in LCBB
  i, Lowe Bound ($\hat{c}$)
  ii, Uppee Bound ($\hat{u}$)

while calculating $\hat{c}$ foe a node in state space tee fractions aee allowed.
while calculating $\hat{u}$ foe a node in state space tee ~~space tee~~, fractions aee not allowed.

Applications :-
1, 0/1 knapsack peoblem
2, Teavelling Sales peeson peoblem
3, Job sequencing.

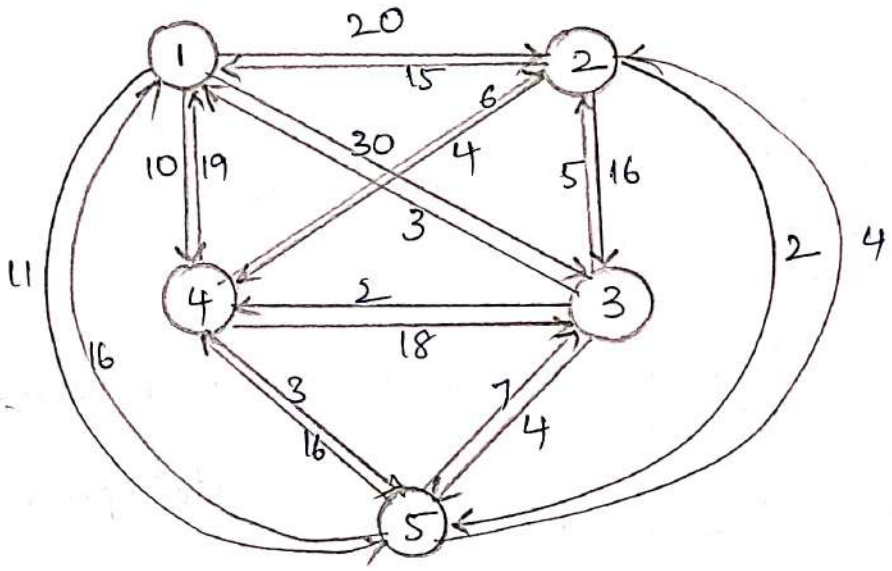
LCSeaech Algoithm :-

```
Algoithm LCSeaech(t)
// Seaech t foe an answee node
{
 if *t is an answee node then output *t and eetuen;
 E=t; // E-node
 Initialize the list of live nodes to be empty;
 epeat
 {
   foe each child x of E do
     { if x is an answee node then output the path fcom x to t
                                                and eetuen;
       Add(x); // x is a new live node.
       (x → paent) := E; // Pointe foe path to oot.
     } if theee aee no moe live nodes then
       { wite ("No answee node"); eetuen;
       } E = Least(); } until (false); }
```

Travelling Salesman Problem Branch & Bound :-

For given set of cities & distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once & returns to the starting point.

Ex :-



Sol :-

From the Graph, Adjansency Matrix is,

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

∞ - no edge

Step 1 :-

Find out the reduced cost matrix from a given cost matrix. This can be obtained through.

1, Row reduction, 2, column reduction.

- Row / column is said to be reduced iff it contains atleast one zero & all remaining entries are non-negative.

1, Row reduction :-

Take minimum element from first row & subtract that element from all the elements of first row (including that

element). Next take minimum element from 2nd row & subtract. Similarly, apply same procedure for all rows. On applying row reduction a resultant matrix is obtained. For this matrix apply colomn reduction.

2. column reduction :-

Take minimum element from first column, the subtract that element from all the elements of first column. Similarly, apply same process for the remaining columns.

Now, find row wise reduction sum & colomn wise reduction sum.

Total reduction cost = row reduction cost + column reduction cost.

- First, Find the minimum value in the row & subtract minimum value from all rows.

row reduction

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} -10 \\ -2 \\ -2 \\ -3 \\ -4 \end{matrix} \quad \overline{21}$$

$\Rightarrow$

colomn reduction

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

$-1 \quad 0 \quad -3 \quad 0 \quad 0 = 4$

If already zero in colomn or row no need to subtract.

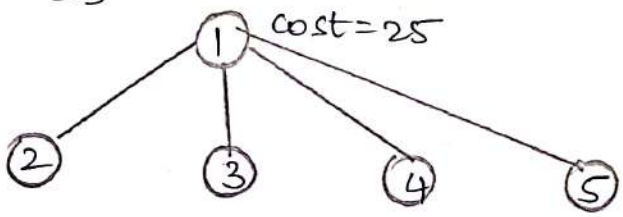
Total = $21 + 4 = 25$

Reducted matrix is

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 0 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Operation on reduced cost matrix:-

The nodes are 1, 2, 3, 4, 5. The cost of node 1 is calculated = 25



Now, we have to calculate the cost of the edges 1 to 2, 1 to 3, 1 to 4, 1 to 5. For this calculation, apply 'branch & bound principle', ie,

- change all enteies of i^{th} row & j^{th} column to ∞.
- we have to change $(j, 1) \rightarrow \infty$
- we have to make reduced cost matrix.

Cost for path (1, 2) :-

(i) we have to check whether it is reduced cost matrix or not. If it is not reduced cost matrix, then we have to convert it into reduced cost matrix. (reduced cost matrix is nothing but at least one row entry or column entry must be zero).

- (i) consider edge (1, 2) & make 1^{st} row & 2^{nd} column as '∞'.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

- (ii) we have to change $(j, 1) \rightarrow \infty$.
  Here $i = 1$, $j = 2$, $\Rightarrow (2, 1) \rightarrow \infty$
- (iii) check whether above matrix is reduced or not.
  If not change it.
  - It is reduced cost matrix.

cost = cost of parent + (1,2) edge cost + reduced cost
↓ From reduced cost matrix

cost = 25 + 10 + 0 = 35

Cost for path (1,3) :-

- Now change 1st row & 3rd column as '∞'.
- change (3,1) → ∞
- reduce the above matrix & find the cost →

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \\ -11 & 0 & & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Total (reduced cost) = row reduction + column reduction = 0 + 11 = 11

Cost = 25 + 17 + 11 = 53

Cost for path (1,4) :-

- change 1st row & 4th column as '∞'
- change (4,1) → ∞
- reduce the matrix & find the cost

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

reduced cost = 0

Cost = 25 + 0 + 0 = 25.

cost for path (1,5) :-

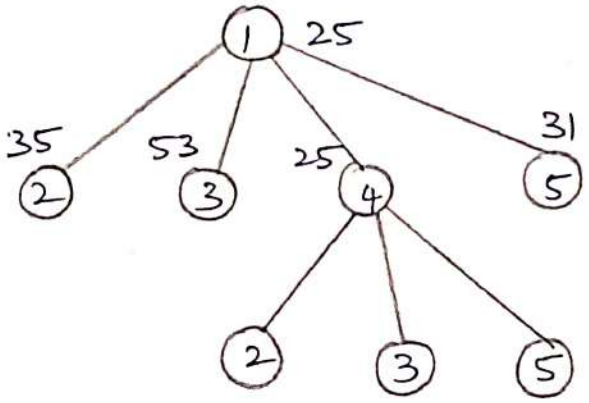
- change 1st row & 5th column as '∞'
- change (5,1) → ∞
- Find reduced matrix

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \begin{matrix} -2 \\ 0 \\ -3 \\ 0 \\ \hline 5 \end{matrix}$$

$$= \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

reduct cost = 5 + 0 = 5

cost = 25 + 1 + 5 = 31



From above Spanning tree select minimum cost node. ie, node 4 with cost-25. and explore it.

This is called Least cost Branch & Bound. (LCBB).

Find the cost-{(4,2), (4,3), (4,5)}

cost for path (4,2) :-

perform operation on 'cost for path (1,4) matrix.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

- Change 4th row & 2nd column as ∞. change $(2,1) \to \infty$.
- Already it is in reduced cost matrix.

reduced cost = 0

cost = 25 + 3 + 0 = 28.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

cost of path (4,3):-

- change 4th row & 3rd column as ∞.
- change $(3,1) \to \infty$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \begin{matrix} \\ \\ -2 \\ \\ \\ \hline 2 \end{matrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ -11 \end{matrix}$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

reduced cost = 2 + 11 = 13

cost = 25 + 12 + 13 = 50

cost for path (4,5):-

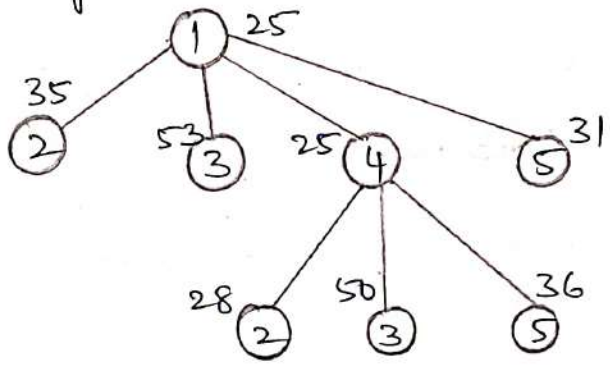
- change 4th row & 5th column as ∞
- change $(5,1) \to \infty$.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} -11 \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

reduced cost = 11 + 0 = 11

cost = 25 + 0 + 11 = 36

Now Spanning tree is



Now select least cost node. ie, node 2 with cost 28. explore it.

Find the cost of the paths (2,3), (2,5) :-

cost for path (2,3) :-

consider the matrix ^cost for path (4,2).

- change 2nd row & 3rd column as ∞
- change (3,1) → ∞.

cost for path (4,2) matrix is,

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} -11$$

$$\Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix} -2 \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

reduced cost = 11 + 2 = 13

cost = 28 + 11 + 13 = 52

Cost for path (2, 5) :-

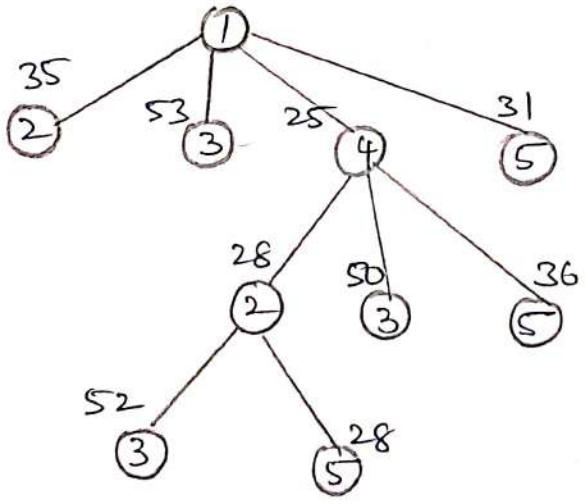
Change 2nd row & 5th column as ∞

Change (5, 1) → ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

reducted cost = 0

cost = 28 + 0 + 0 = 28.



Now select least cost node. ie node 5. with cost 28. explore it.

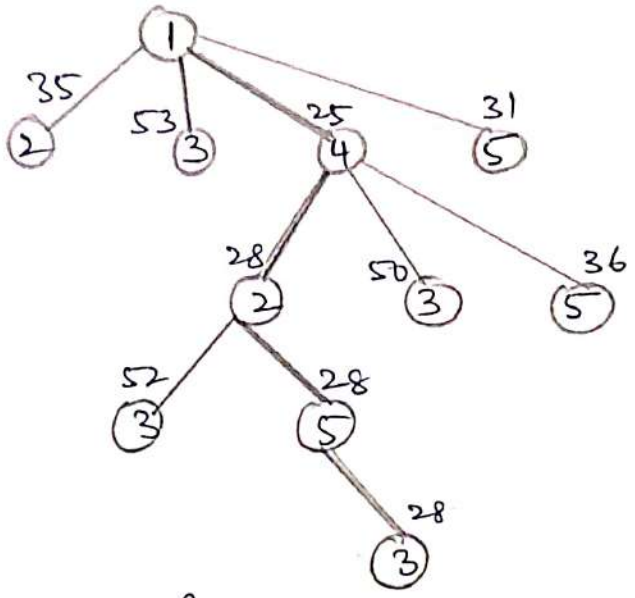
Find the cost of the paths (5, 3) :-

Consider the matrix, cost for path (2, 5).

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

reduced cost = 0

cost = 28 + 0 + 0 = 28.



least cost path is $1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 1$

* The time complexity of travelling salesperson problem is $O(n^2 2^n)$.

0/1 Knapsack Problem using LCBB:-

To use branch & bound technique to solve any problem, it is first necessary to conceive of a state space tree for the problem. Branch & Bound is used for minimization problems.

As 0/1 knapsack is about maximizing the total value. we cannot directly use the LCBB to solve this. we convert this into a minimization problem by taking negative of the given values.

Ex :-

$n = 4, (P_1, P_2, P_3, P_4) = (10, 10, 12, 18),$

$(w_1, w_2, w_3, w_4) = (2, 4, 6, 9), \ m = 15$

Sol :- place the objects in knapsack, so that get maximum profit.

first convert it into maximization problem by taking negative values of profits.

$(P_1, P_2, P_3, P_4) = (-10, -10, -12, -18).$

we use fixed size solution vector (x_1, x_2, x_3, x_4)

calculate lowerbound ($\hat{c}$) & upperbound ($\hat{u}$) for each & every node.

$$\text{lowerbound}(\hat{c}) = \sum_{i=1}^{n} P_i x_i \text{ (with fractions)}$$

$$\text{upperbound}(\hat{u}) = \sum_{i=1}^{n} P_i x_i \text{ (without fractions)}$$

Here, we have to calculate cost of lowerbound & upperbound for each node in state space tree.

Based on the cost, at each level select a node & explore it.

- On lowerbound fractions are allowed, on upperbound fractions are not allowed.

Calculate the lowerbound & upperbound for node 1.

Here, knapsack capacity $m = 15$.

Node 1 :-

lowerbound = (−10 − 10 − 12) + (−6) = −38

(1st object profit, 2nd, 3rd, fraction of 4th object profit)

$\frac{3}{9}$ x −18	
−12	6
−10	4
−10	2

m = 15

To get the lowerbound, take the fraction of the object which was not being accommodated in the knapsack to fill the knapsack completely.

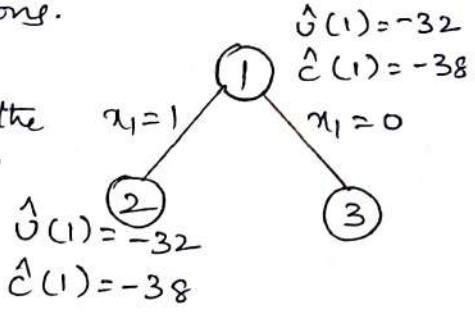
Upperbound $(\hat{u}) = -10 - 10 - 12 = -32$

Here, we cannot include 4th object because knapsack size is 3 and object is 9. Don't take fractions.

Node 2 :- 1st object must be placed in the bag.

lowerbound $(\hat{c}) = -38$

Upperbound $(\hat{u}) = -32$



Node 3 :-

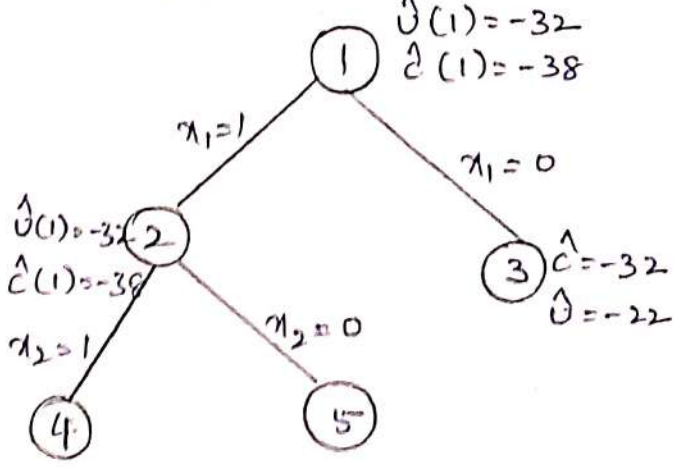
Here we are not including the 1st object. i.e, $x_1 = 0$.

Don't consider size & profit of 1st object.

$\frac{5}{9}$ x −18	
−12	6
−10	4

$LB(\hat{c}) = -10 - 12 - 10 = -32$

$\hat{u} = -10 - 12 = -22$



Take least cost node, i.e, 2.

Now, consider 2nd object.

- Calculate $\hat{c}$ & $\hat{u}$ for node 4. It is similar to node 2, because we are including all objects.

Node 4 :- $\hat{c} = -38$, $\hat{u} = -32$

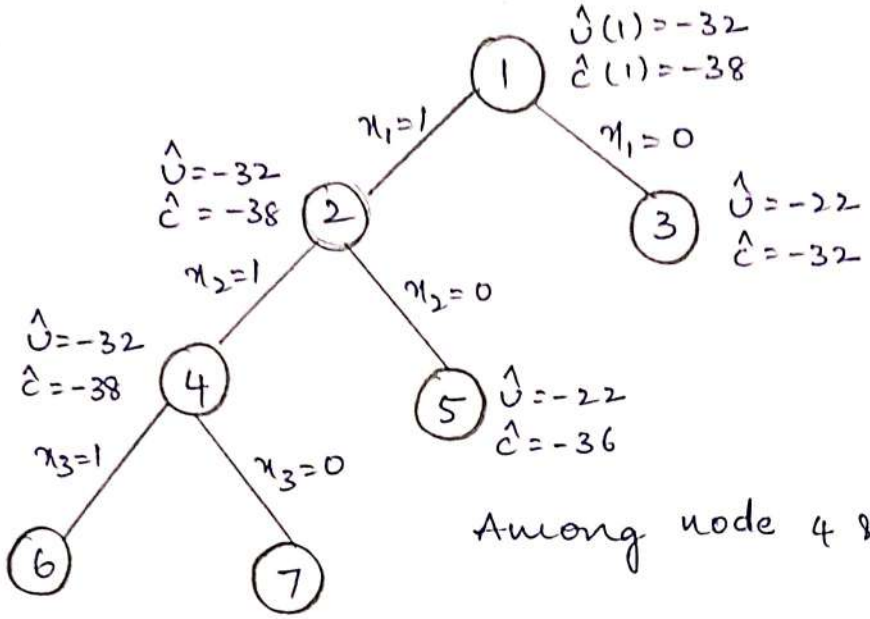
Node 5 :-

Here don't consider 2nd object.

$\hat{c} = -10 - 12 - 14 = -36$

$\hat{v} = -10 - 12 = -22$

-14	$\frac{7}{9} \times -18$
-12	6
-10	2



Among node 4 & 5, least cost node is 4

Now explore node 4. calculate $\hat{v}$ & $\hat{c}$ for node 6.

Node 6 :-

It is similar to node 4, because we are including all objects.

$\hat{c} = -38$

$\hat{v} = -32$

Node 7 :-

Don't consider 3rd object in the bag.

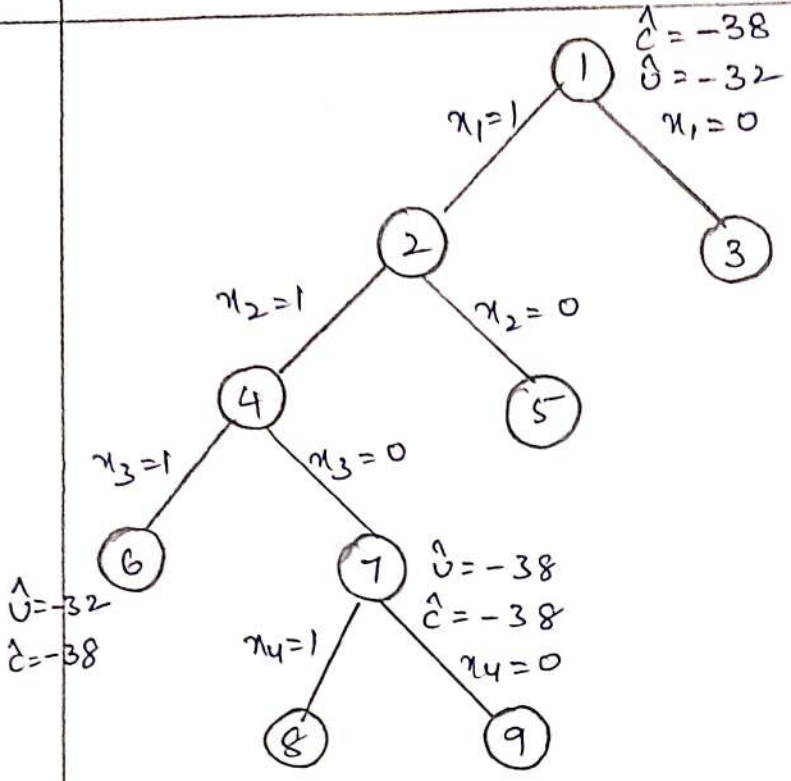
$\hat{c} = -10 - 10 - 18 = -38$

$\hat{v} = -10 - 10 - 18 = -38$.

-18	9
-10	4
-10	2

- compare node 6 & node 7 values.

Here lowerbound values are same. They compare upper bound values of node 6 & 7 and consider min value node. Node 7 has the least cost value. Hence explore it.



-18	9
-10	4
-10	2

Node 8 :-

Don't consider 3rd object.

$\hat{c} = -10 - 10 - 18 = -38$

$\hat{u} = -38$

Node 9 :-

Don't consider 3rd object & 4th object.

$\hat{c} = -10 - 10 = -20$

$\hat{u} = -20$

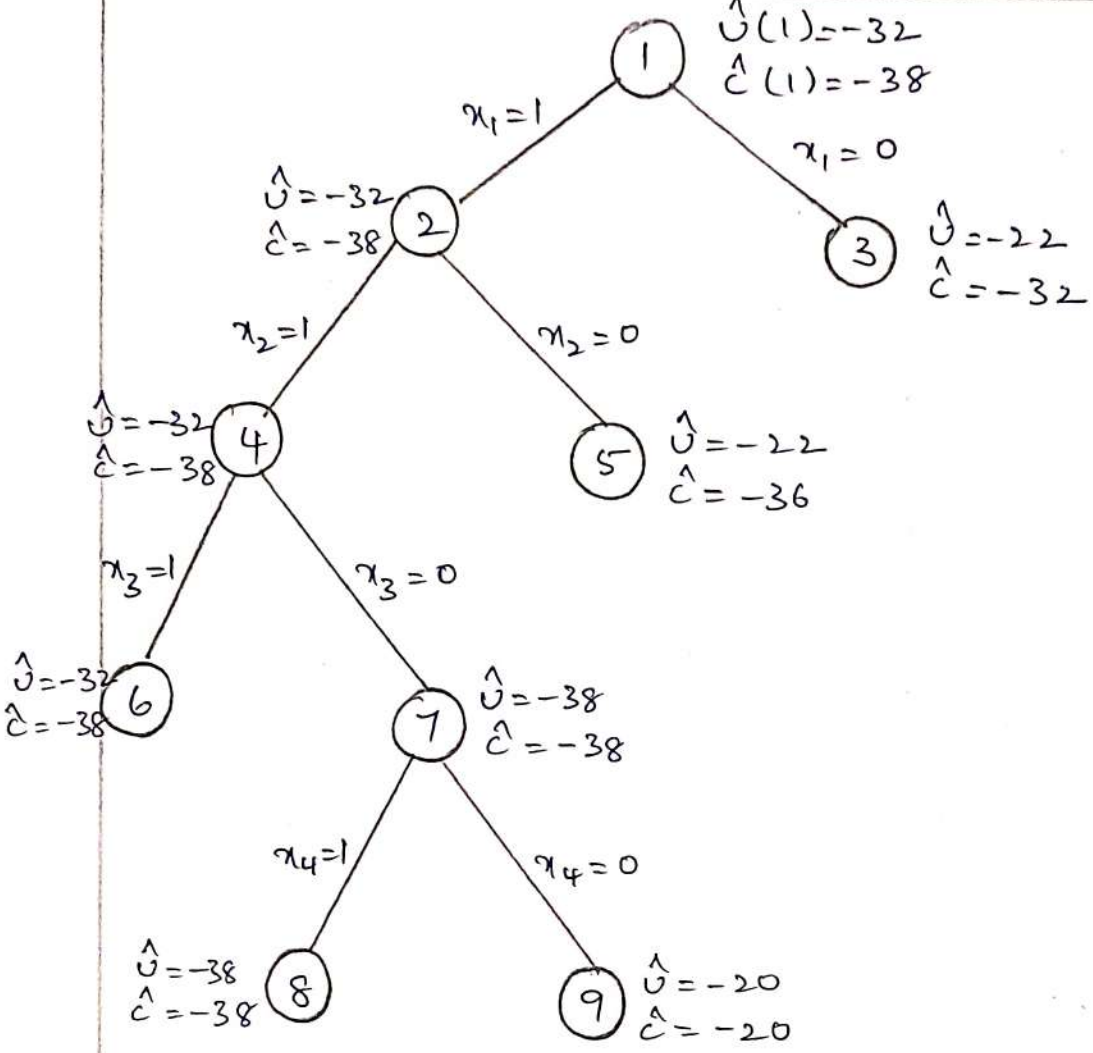
-10	4
-10	2

Among node 8 & 9, least cost is 8th node with minimum upperbound.

Based on the above selection in state space tree, the solution vector is (1, 1, 0, 1)

Profit = $-10 - 10 + 0 - 18 = -38$.

Now convert this minimum value into maximum +38


1
$\hat{v}(1) = -32$
$\hat{c}(1) = -38$
$x_1 = 1$
$x_1 = 0$
2
$\hat{v} = -32$
$\hat{c} = -38$
3
$\hat{v} = -22$
$\hat{c} = -32$
$x_2 = 1$
$x_2 = 0$
4
$\hat{v} = -32$
$\hat{c} = -38$
5
$\hat{v} = -22$
$\hat{c} = -36$
$x_3 = 1$
$x_3 = 0$
6
$\hat{v} = -32$
$\hat{c} = -38$
7
$\hat{v} = -38$
$\hat{c} = -38$
$x_4 = 1$
$x_4 = 0$
8
$\hat{v} = -38$
$\hat{c} = -38$
9
$\hat{v} = -20$
$\hat{c} = -20$

LC Beanch & Bound for 0/1 Knapsack Algorithm:-

Algorithm UBound (cp, cw, k, m) // cp - current profit, cw - current weight

// w[i] & P[i] are respectively the weight & profit of the i^{th} object.

```
{
  b = cp ; c = cw ;
  for i = k+1 to n do
    {
      if (c + w[i] ≤ m) then
        {
          c = c + w[i] ; b = b - P[i] ;
        }
    }
  return b ;
}
```

FIFO- Branch & Bound for 0/1 knapsack :-

FIFO branch & bound constructs state space tree in such a manner that nodes specify the choices used for the solution.

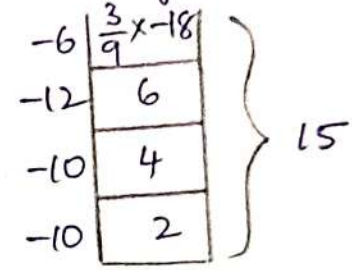
Ex:- $n = 4$, $(P_1, P_2, P_3, P_4) = (-10, -10, -12, -18)$

$(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, $m = 15$.

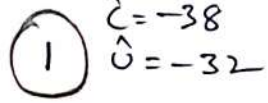
Sol :- Find cost $\hat{c}$ & $\hat{u}$ for node 1, by placing the objects

$\hat{c} = -10 - 10 - 12 - 6 = -38$

$\hat{u} = -10 - 10 - 12 = -32$



($\hat{u}$ - fractions not allowed)

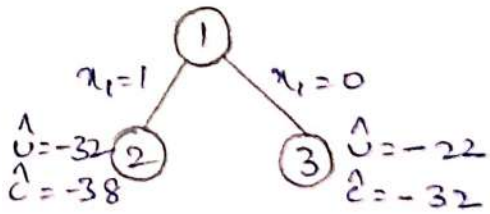


After finding cost ($\hat{c}$) & $\hat{u}$ of node 1, consider Global Upper bound value as upper bound of node 1.

(GUB) Global upperbound = -32

Node 2 :- $\hat{u} = -32$, $\hat{c} = -38$

Node 3 :- $\hat{u} = -22$, $\hat{c} = -32$



compare upperbound of node 2 & node 3 with Global upper bound and it should always minimum

node ② UB = -32 node ③ UB = -22

GUB = -32 GUB = -32

both values are same -32 is minimum

- no need to update GUB in both cases
- Now compare LB (lower Bound) with GUB of 2 & 3 nodes

If LB > GUB then kill the node

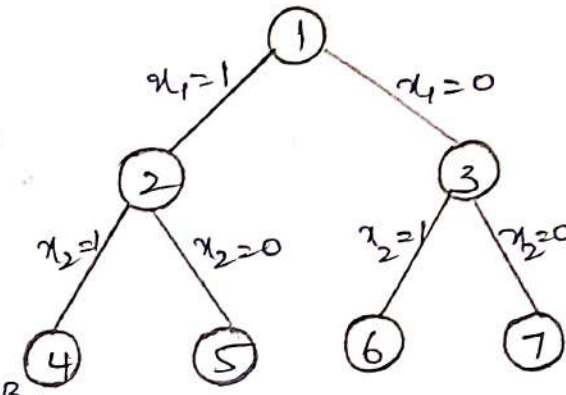
node 2 LB GUB
$-38 > -32$
fail - don't kill

node 3 LB GUB
$-32 > -32$
fail - don't kill

- Explore 2 & 3 nodes.

Calculate LB & UB of node 4 & 5.

node 4 : LB = −38, UB = −32
node 5 : LB = −36, UB = −22



- compare UB of node 4 & 5 with GUB.

node 4
LB GUB
$-32, -32$
↓
Same

node 5
LB GUB
$-22 ; -32$
↓ minimum

No need to update GUB.

compare LB with GUB :-

node 4
$-38 > -32$
Fail

node 5
$-36 > -32$
Fail

Don't kill the nodes.

- Calculate LB & UB of node 6 & node 7

Node 6 :-

−10	$\frac{5}{9}x-18$	4th
−12	6	3rd
−10	4	2nd

$\hat{C} = -10-12-10 = -32$

$\hat{U} = -10-12 = -22$

node 7 :-

$\hat{c} = -12-18 = -30$

$\hat{v} = -12-18 = -30$

-18	9
-12	6

Compare upperbounds of node 6 & node 7 with GUB.

node 6 $\hat{v}$ GUB

-22, -32

min

node 7 $\hat{v}$ GUB

-30 -32

min

No need to update GUB.

Compare LB with GUB.

node 6 : LB

$-32 > -32$

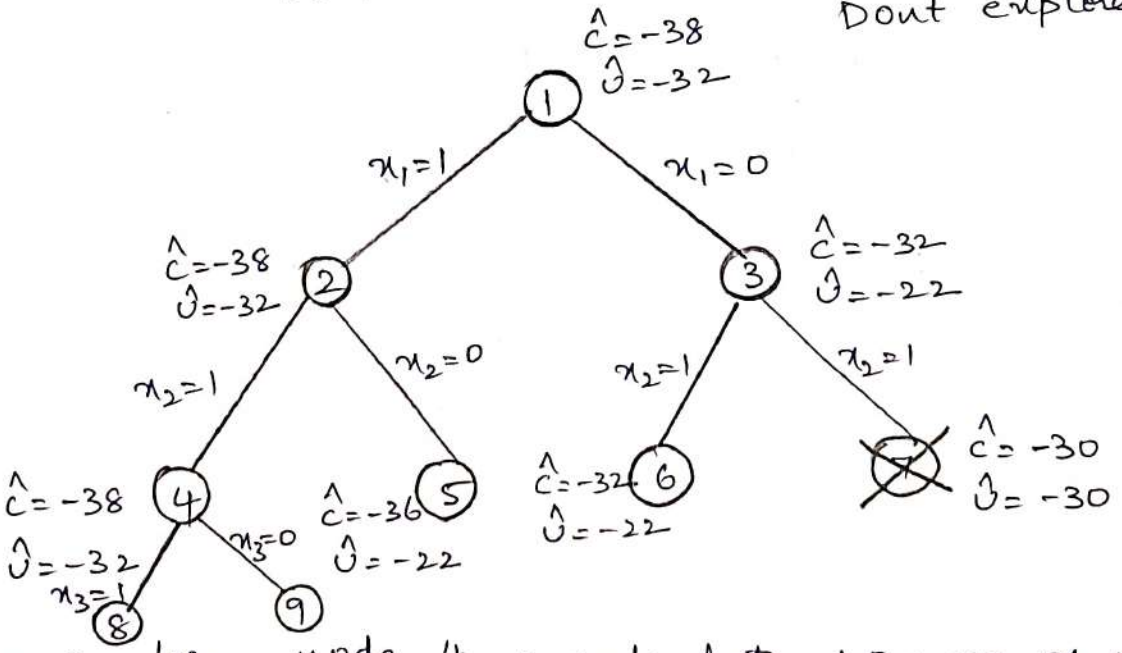
Fail

node 7 : LB

$-30 > -32$

True. Kill the node 7

Don't explore node 7.



- Explore node 4. & calculate LB & UB of node 8 & 9.

Node 8: $\hat{c} = -38$, $\hat{v} = -32$ (similar to node 4)

Node 9: $\hat{c} = -10-10-18 = -38$

$\hat{v} = -38$

-18	9
-10	4
-10	2

Compare UB with GUB for node 8 & 9.

node 8 : $\hat{U} = -32$, $G\hat{U}B = -32$ same, don't update

node 9 : $\hat{U} = -38$, $G\hat{U}B = -32$

Here upperbound value is lessthan GUB. So, update Global upper bound value.

Now $\boxed{GUB = -38}$

Now explore node 5. Initially check UB & LB with GUB

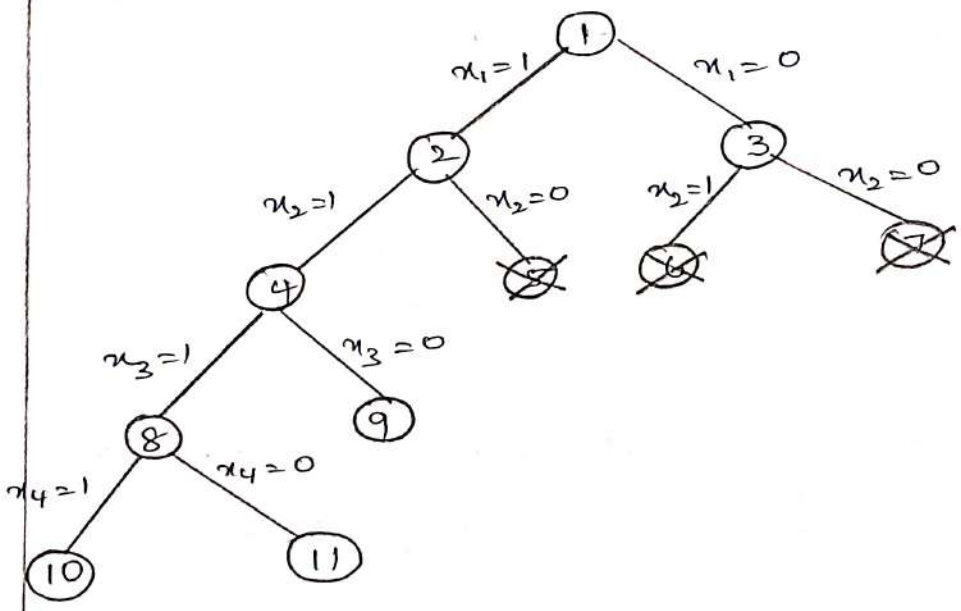
node 5 : UB = −22, GUB = −38 (minimum) - Don't update

node 6 : UB = −22, GUB = −38 (min) Don't update

node 5 : LB > GUB ⇒ −36 > −38. True. Kill the node 5

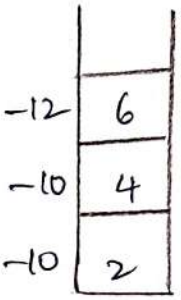
node 6 : LB > GUB ⇒ −32 > −38. True. Kill the node 6.

- Next, explore node 8 & node 9



Infeasible

Node 10, UB & LB is not possible. Because $x_1=1, x_2=1, x_3=1, x_4=1$, we can't place all objects in knapsack. It is not feasible.



Node 11 :- $x_1=1, x_2=1, x_3=1, x_4=0$

$\hat{c} = -10-10-12 = -32$

$\hat{u} = -32$

Compare upperbound with GUB.

$\hat{u} = -32$, GUB $= -32$. same. Don't update.

LB > GUB $\Rightarrow -32 > -38$. True. Kill the node.

Node 12 :- $x_1=1, x_2=1, x_3=0, x_4=1$

$\hat{c} = -10-10-18 = -38$

$\hat{u} = -38$

Compare UB with GUB $\Rightarrow -38, -38$. same

LB > GUB $\Rightarrow -38 > -38$ fail

Node 13 :- $x_1=1, x_2=1, x_3=0, x_4=0$

$\hat{c} = -10-10 = -20$

$\hat{u} = -20$

compare UB with GUB $\Rightarrow -20, -38$. Don't update

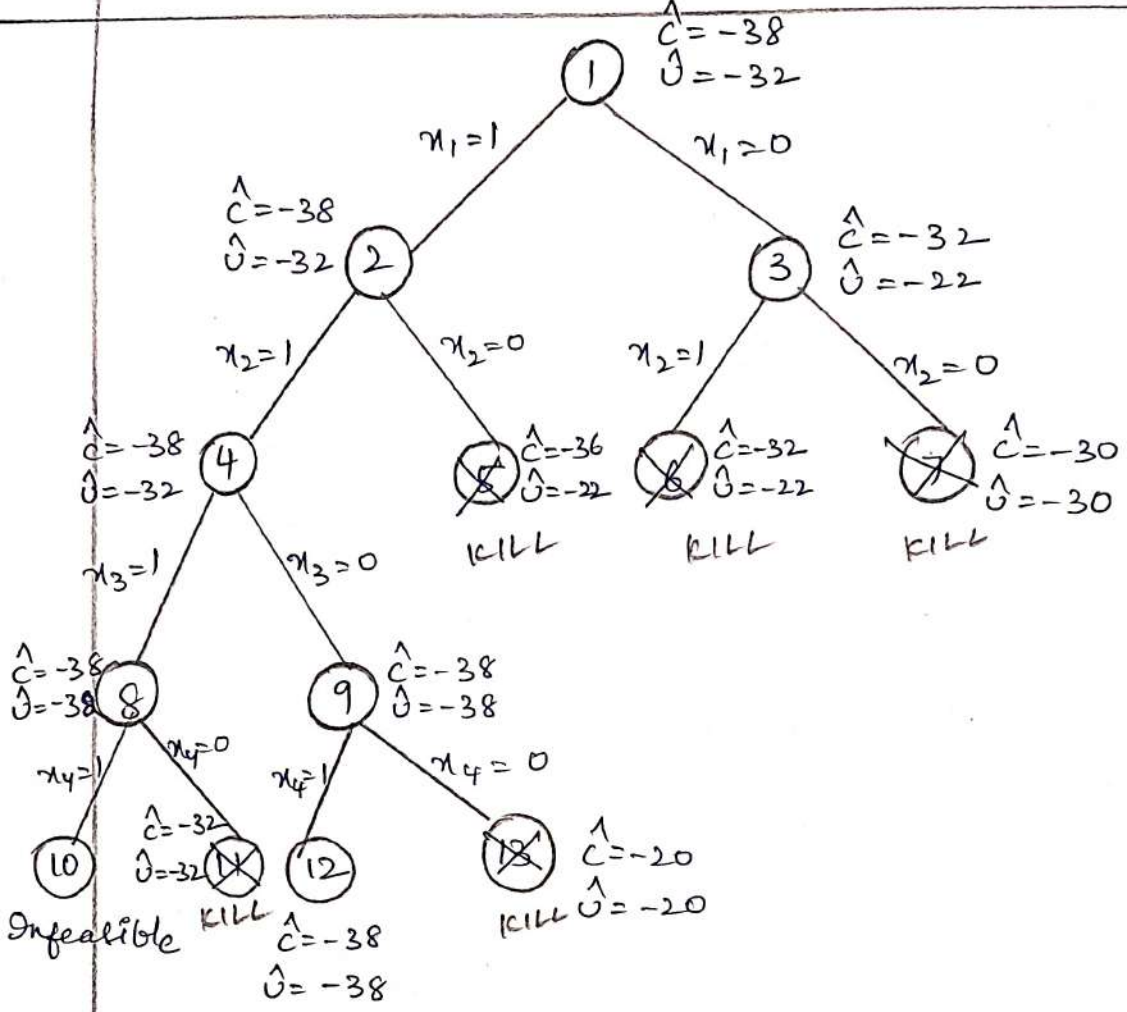
min

LB > GUB $\Rightarrow -20 > -38$ - True. Kill the node 13

Solution vector $= (x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

cost $= -10-10+0+(-18) = -38$.

profit $= 38$.

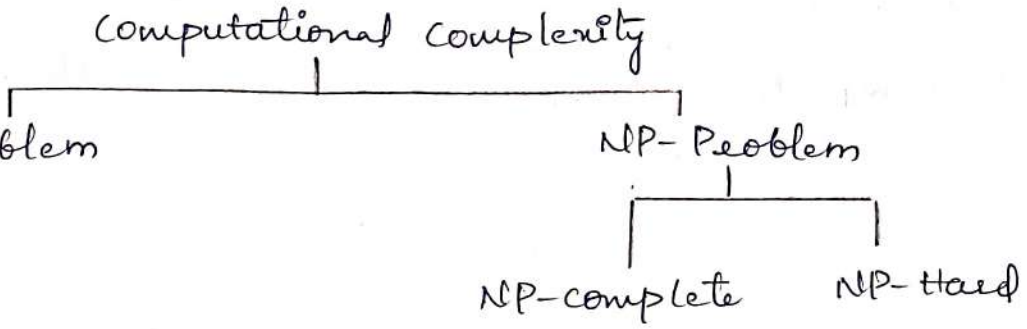


Time complexity - $O(n^2)$

NP- Hard & NP- Complete Problems :-

Basic concepts :-

There are 2 different types of problems.



P- Problem :- Problem that can be solved in polynomial time.

Ex :- Searching, sorting, All pairs shortest path.

Problem can be classified into 2 groups.

1. Polynomial time algorithm
2. Non- Polynomial time algorithm.

Problem whose solution times are bounded by polynomials of small degree are called 'polynomial time algorithm'.

Eg :- Linear search ($O(\log n)$), Quicksort ($O(\log n)$), All pairs shortestpath.

Problem whose solution times are bounded by non-polynomials are called 'non-polynomial time algorithm'.

Eg :- Travelling salesman problem ($O(n^2 2^n)$), knapsack problem ($O(2^{n/2})$).

* Any problem for which the answer is either Yes or no is called Decision Problem. The algorithm for decision problem is called Decision Algorithm.

Eg :- Sum of subsets problem, Maximum clique problem.

* Any problem that involves the identification of an optimal value is called 'optimization Problem'.

Eg :- Knapsack Problem, Travelling Salesperson Problem.

NP-class problem :-

NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Ex:- Travelling salesperson problem, Graph coloring problem.

NP problems cannot be solved efficiently. The relation between the P and NP-Problems is $P \subseteq NP$.

NP-complete Problems :-

A problem is said to be NP-complete if & only if it satisfies the following conditions.

i. Problem should be present in class NP.

ii. Each problem should also be solved in polynomial type.

Ex:- Graph coloring problem, Knapsack problem etc.

NP-Hard Problem :-

A problem is said to be NP-hard if & only if it is solved in polynomial time or reduced to an NP-hard problem. An NP-hard problem is solvable in polynomial time then each NP-complete problem can also be solved in polynomial time.

Ex:- Travelling Salesman problem.

Each NP-complete problem is said to be NP-hard whereas each NP-hard problem cannot said to be NP complete problem.

Non-Deterministic Algorithm :-

Non-deterministic algorithm is an algorithm in which each operation generates one or more result. This algorithm does not produce unique result. It consists of 2 stages.

The algorithm in which every operation is uniquely defined is called deterministic algorithm.

The Non-deterministic algorithm consists of 2 stages.

i, Non-deterministic stage :-

This stage is also known as 'guessing stage'. It can create an arbitrary resultant string which can be considered as a candidate solution.

ii, Deterministic Stage :-

This stage is also known as 'verification stage'. It verifies whether the candidate solution is the actual solution. In order to do this, it takes candidate solution as input & returns output which can be either yes or no.

Non-Deterministic algorithm use the following functions :-

i, choice (S) :-

This function is used to select an element randomly from given sets.

ii, failure () :-

This function is used to donote an unsuccessful completion

iii, Success() :-

It is used to denote a successful completion.

A non-deterministic algorithm for searching an element :-

x is a given set of elements $A[1:n]$, $n \geq 1$. we are required to determine an index j such that $A[j] = x$ or $j = 0$ if x is not in A.

```
Algorithm Search (A, x)
{ j = choice (1, n);
  if A[j] = x then
  { write (j);
```

```
        success();
      }
    else
      {
        write(0);
        failure();
      }
  }
```

A non-deterministic algorithm terminates unsuccessful iff there exist no set of choices leading to success signal. whenever there is a set of choices that leads to successful completion, then one such set of choices is selected & the algorithm terminates successfully.

In case, the successful completion is not possible, then the time complexity is O(1). In case of successful signal completion then the time complexity is O(n), where n is the no of i/p's.

Non-Deterministic algorithm for sorting :-

Let A[1:n] be an unsorted array of positive integers B[1:n] is an auxillary array. The non-deterministic algorithm NSort(A,n) sorts then the no's into non-decreasing order.

```
Algorithm NSort(A,n)
// Sort n positive integers
{
  for i=1 to n do B[i]=0
  for i=1 to n do
    { j= choice(1,n);
      if B[j] ≠ 0 then failure();
        B[j]= A[i];
    }
  for i=1 to n-1 do // verify order
    if B[i] > B[i+1] then failure();
      write (B[1:n]);
        success();
    }
```

Time complexity = O(n).

Decision Problem :-

Any problem for which the answer is either zero or one is called a Decision problem. An algorithm for a decision problem is called 'Decision algorithm.

Any problem that involves the identification of an optimal (either minimum & maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

Maximum clique :-

Let $G = \{V, E\}$ be a undirected graph. A maximal complete subgraph of a graph G is a clique. The size of the clique is the no of vertices in it. The maximum clique problem is an optimization problem that has to determine the size of a largest clique in G.

G has a clique of size at least k for some given k.

Let DClique(G, k) be a deterministic decision algorithm for the clique decision problem. If the no of vertices in G is n, the size of max clique found through DClique. DClique is used one for each k, $k = n, n-1, n-2 \ldots$ until the output from DClique is 1.

If time complexity of DClique is $f(n)$, then the size of a max clique can be found in time $\leq n \cdot f(n)$. Also, if the size of a max clique can be determined in the time $g(n)$, then the decision problem can be solved in time $g(n)$. Hence, the max clique problem can be solved in polynomial time, if the clique decision problem can be solved in polynomial time.

The input to the max clique decision problem can be provided as sequence of edges & an integer k. Each edge in E(G) is a pair of no's (i, j). The input size for each edge (i, j) is

$\log_2 i + \log_2 j$ if a binary representation is assumed.

The input size of any instance is

$$n = \sum_{\substack{(i,j) \in E(G) \\ i<j}} \left([\log_2 i] + [\log_2 j] + 2 \right) + [\log_2 K] + 1$$

Note that if G has only one connected component, then $n \geq |V|$. Thus, if this decision problem cannot be solved by an algorithm of complexity $P(n)$ for some polynomial $P()$, then it cannot be solved by an algorithm of complexity $P(|V|)$

Algorithm DCK is a non-deterministic algorithm for the clique decision problem. Algorithm DCK begins to form set of K distinct vertices. Then it tests to see whether these vertices form a complete subgraph. If G is given by its adjacency matrix & $|V| = n$. The input length m is $n^2 + [\log_2 K] + [\log_2 n] + 2$.

Algorithm for Nondeterministic clique pseudocode.

```
Algorithm DCK(G, n, K)
{
  S = φ; // S is an initially empty set.
  for i=1 to K do
  {
    t = choice(1, n);
    if t ∈ S then Failure();
    S = S ∪ {t} // Add t to set S.
  } // At this point S contains K distinct vertex
    indices. for all pairs (i, j) such that i ∈ S, j ∈ S
    and i ≠ j do if (i, j) is not an edge of G
    then Failure();
  }
```

Time complexity is $O(n + K^2) = O(n^2) = O(m)$.

Satisfiability :-

The satisfiability problem is to determine whether a formula is true for some assignment of truth values to the variables.

Let x_1, x_2 -- denote boolean variables (their value is either true or false). Let $\bar{x}_i$ denote the negation of x_i. A literal is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals & the operations AND and OR.

Ex:- $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ and $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$

The symbol $\vee$ denotes 'or' and $\wedge$ denotes AND.

The satisfiable formulas are :-

1, CNF (conjunctive Normal Form) :-

If & only if it is represented as $\bigwedge_{i=1}^{k} c_i$, where the c_i are represented as $\bigvee l_{ij}$.

Eg :- $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF.

2, DNF (Disjunctive Normal Form) :-

If & only if it is represented as $\bigvee_{i=1}^{k} c_i$ and each clause c_i is represented as $\bigwedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF.

The nondeterministic algorithm that terminates successfully iff given propositional formula $E(x_1, x_2 \ldots x_n)$. is satisfiable.

Algorithm for 'Nondeterministic satisfiability :-

```
Algorithm Eval(E, n)
// Determine whether the propositional formula E is
   satisfiable. The variables are x1, x2 ... xn.
{ for i=1 to n do // choose a truth value assignment
     xi = choice(false, true);
  if E(x1, ... xn) then Success();
    else Failure();
}
```

The time complexity of nondeterminist Satisfiability is $O(n)$. This time is proportional to the length of E.

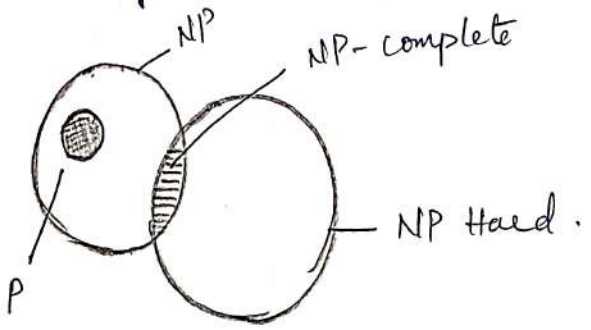
NP-Hard & NP- complete classes :-

An algorithm A is of polynomial complexity if there exists a polynomial $p()$ such that the computing time of A is $O(p(n))$ for every input of size n.

P is the set of all decision problems solvable by deterministic algorithms in polynomial time. NP is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Most famous unsolved problem in computer science, is whether $P = NP$ or $P \neq NP$.

Non-deterministic polynomial time problems can be classified into 2 classes. They are NP-Hard & NP-complete.



NP- Hard :-

A problem L is NP-Hard problem if Satisfiability reduces to L (satisfiability α L)

A problem L is NP-complete if & only if L is NP-Hard & $L \in NP$.

Nondeterministic polynomial time problem is satisfiable & reducable then the problem is said to be NP-Hard.

Eg:- Halting problem, flow-shop scheduling problem.

NP- Complete :-

A problem L is NP-complete if L is NP-Hard and $L \in NP$ (nondeterministic polynomial).

The NP-complete problem has the property that it can be
i) solved in polynomial time (ie NP-Hard problem) &
ii) Problem should be present in NP class.

If an NP-Hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time. All NP-complete problems are NP-Hard. But some NP-Hard problems that are not NP-complete.

Normally the decision problems are NP-complete but an optimization problems are NP-Hard.

Relationship between P, NP, NP-Hard & NP-complete :-

Let P, NP, NPH, NPC are the set of all decision problem that are solvable in polynomial time by using deterministic algorithms, Non deterministic algorithm NP-Hard, NP-complete respectively.

- Both P & NPC are with in $P \cap NPC = \phi$, $NPC = NP \cap NPH$.
- when $NP = NPH$ and $P = NP \cap NPH$ then relationship between P, NP, NPC & NPH.

Cook's Theorem :-

Scientist Stephen cook in 1971 stated that Boolean Satisfiability (SAT).

cook's theorem states that Satisfiability is in P if $P = NP$. we already know that Satisfiability is in NP, hence if $P = NP$, then Satisfiability is in P.

ie, to show that if satisfiability is in P then $P = NP$.

In order to prove that statement, we show how to obtain any polynomial time non deterministic algorithm A & input I, a formula $Q(A, I) \Rightarrow Q\ (Q(A, I) \Rightarrow Q)$ is satisfiable, if input

I has successful termination with 'A'.
- If length of I is in x time complexity of A is P(n) for some polynomial P, then length of Q will be $O(P^3(n)\log n) = O(P^4(n))$
- Time needed to construct Q will also be same.
- The deterministic algorithm to determine outcome of A on any input I may be obtained.
- If we compute Q and uses deterministic algorithm for a satisfiability problem to determine whether or not Q is satisfiable.
- If $O(g(m))$ is time needed to determine a formula of length m is satisfiable then complexity is $O(P^3(n)\log n) + O(P^3(n)\log$

* Difference between NP-Hard & NP-complete.

NP-Hard	NP-complete
- problem P_i is harder than any other problems in NP.	- Problem P_i is most complex of all problems.
- Symbolically, it is represented as $P_j \rightarrow NP$, P_j is polynomial type not equal to P_i.	- Symbollically it is represented as $P_i \rightarrow NP$.
- A decision problem is said to be NP-Hard if every problem in NP-Hard is reduced to P_j in polynomial type.	- A decision problem P_i is said to be NP-complete if it is NP-complete & solvable in polynomial type P_i.

Difference between divide-Conquer & Greedy approach.

Divide & Conquer	Greedy Method
- It is result oriented.	- There are some chances of getting optimal solution to specific problem
- This approach donot depend on constraints, to solve a specific problem.	- This approach cannot make feasible move if subset does donot satisfy specific subset
- This approach is not effective for larger problems.	- This approach is applicable as well as efficient for a wide variety of problems.
- It is not applicable to problems which are not divisible	- This problem is rectified in greedy method.
- Time taken by this algorithm is efficient compared to greedy approach.	- Time taken by this algorithm is not that much efficient when compared to divide & conquer.
- Problems divided into a large no of subproblems space requirement is very high.	- Space requirement is less when compared to D & C.
applications :- Merge sort, binary sort etc	Applications :- For sequence, minimum cost spanning tree.

Dynamic Programming	Greedy Approach
1. Decision sequences are generated.	- Only one decision sequence is generated.
- It considers all possible sequences inorder to obtain optimal solution.	- It is straight forward method for choosing optimal solution, it selects optimal solution without revising previous false solution.
- It is bottom up approach.	- It is top-down approach.

Differences between Back-Tracking & Branch & Bound.

Back Tracking	Branch & Bound
∴ Solution is obtained using BFS, DFS.	∴ Any of search methods FIFO, LIFO, LC search can be used to obtain solution.
- This approach provides solution to decision problem	- This technique is used to solve optimization problem.
- State space tree is not generated completely, instead process of searching terminates as soon as solution is obtained.	- State space tree is generated using branch & bound method is expanded completely, there is a possibility of obtaining optimal solution at any point in state space tree.
- It is applicable for subsets, Hamiltonian cycle, N-Queens, Graph coloring.	- It is applicable for optimization problems like travelling sales person, 0/1 knapsack prob