UNIT – I

Algorithm Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

Characteristics of an Algorithm:

1. INPUT → Zero or more quantities are externally supplied.
2. OUTPUT → At least one quantity is produced.
3. DEFINITENESS → Each instruction is clear and unambiguous.
4. FINITENESS → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. EFFECTIVENESS → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Differences between algorithm and program:

Algorithm	Program
1. It is a step by step procedure for solving the given problem.	1. A program is nothing but a set of instructions or executable code.
2. An algorithm is designed by designer	2. The program can be implemented by a programmer.
3.An algorithm is done at design phase	3. A program is implemented at implementation phase.
4. An algorithm can be expressed by using English, flowchart and pseudo code.	4. A program can be written by using languages like C, C++, Java etc…
5. After writing the algorithm, we have to analyze it using space and time complexities.	5.After writing a program, we have to test them

Algorithm Specification:
Algorithm can be described in three ways.

1. Natural language like English: When this way is choose 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: The following are set of rules need to be followed while writing algorithms

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces { and }. A compound statement can be represented as a block. The body of a procedure also forms a block. Statements are delimited by **;**.
3. An identifier begins with a letter. The data types of variables are not explicitly declared. Whether a variable is local or global to a procedure will also be evident from the context.
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>
}
```

As long as condition is TRUE, the statements get executed. When condition becomes FALSE, the loop is exited. The value of condition is evaluated at top of the loop. The general form of For loop is

For Loop:

for variable: = value 1 to value 2 **step** step do

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

Here value 1, value 2 and step are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "**step** step" is optional and taken as +1 if it does not occur. Step could either be positive or negative. Variable is tested for termination at the start of each iteration. The repeat-until loop is constructed as follows.

repeat-until:

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

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

The statements are executed as long as condition is false. The value of condition is computed after executing the statements. The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A **return** statement within any of the above also will result in exiting the loops. A return statement results in the exit of the function itself.

8. A conditional statement has the following forms.
   → If <condition> then <statement>
   → If <condition> then <statement-1>
   else <statement-1>

   Here condition is the Boolean expression and statements are arbitrary statements.

   Case statement:

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

Here statement 1, statement 2 etc. could be either simple statement or compound statements. A case statement is interpreted as follows. If condition 1 is true, statement 1 gets executed and case statement is exited. If statement 1 is false, condition 2 is evaluated. If condition 2 is true, statement 2 gets executed and the case statement exited and so on. If none of the conditions are true, statements + 1 is executed and the case statement is exited. The else clause is optional.

9. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the <i,j>th element of an array is denoted as A[i,j].
10. Input and output are done using the instructions read & write.
11. There is only one type of procedure:
    Algorithm, the heading takes the form,

    Algorithm Name (Parameter lists)

    Where Name is the name of the procedure and parameter list is a listing of the procedure parameters. The body has one or more statements enclosed with braces { and }. An algorithm may or may not return values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or record name is treated as a pointer to the respective data type.

    → As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

```
Algorithm Max(A,n)
// A is an array of size n
{
    Result := A[1];
    for I:= 1 to n do
      if A[I] > Result then
          Result :=A[I];
     return Result;
```

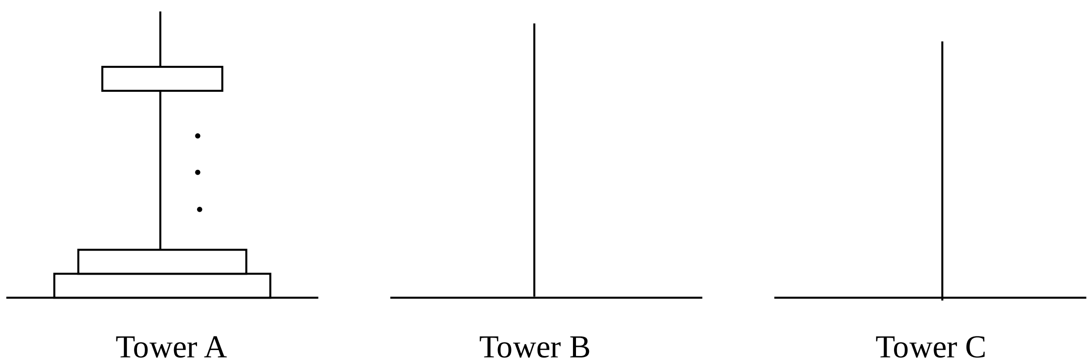
}

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

Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is Direct Recursive.
- Algorithm 'A' is said to be Indirect Recursive if it calls another algorithm which in turns calls 'A'.
- The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
- The following 2 examples show how to develop recursive algorithms.
  → In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1. **Towers of Hanoi:**



- According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
- The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
- Besides these tower there were two other diamond towers(labeled B & C)
- Goal is to move the disks from tower A to tower B using tower C, for intermediate storage.
- As the disks are very heavy, they can be moved only one at a time.
- In addition, at no time can a disk be on top of a smaller disk.
- According to legend, the world will come to an end when the priest have completed this task.
- A very elegant solution results from the use of recursion.
- Assume that the number of disks is 'n'.
- To get the largest disk to the bottom of tower B, we move the remaining 'n-1' disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.

- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

Algorithm:

```
Algorithm TowersofHanoi(n,x,y,z)
//Move the top 'n' disks from tower x to tower y.
{
    if(n>=1) then
    {
    TowersofHanoi(n-1,x,z,y);
    Write("move top disk from tower " X ,"to top of tower " ,Y);
    Towersofhanoi(n-1,z,y,x);
    }
}
```

2. **Recursive algorithm for Factorial Of Given Number:**

```
Algorithm rfactorial(n)
{
    If(n=1) then
            return 1;
    else
            return (n-1) * rfactorial(n);
}
```

3. **Recursive algorithm for GCD of two numbers:**

```
Algorithm rgcd(a,b)
{
    if(a!=b) then
    {
            if(a>b) then
            {
                    a := a – b;
                    rgcd(a,b);
            }
            else
            {
                    b := b- a;
                    rgcd(a,b);
            }
    }
    return a;
}
```

Performance Analysis: The efficiency of an algorithm is declared by measuring the performance of an algorithm. Performance of an algorithm can be computed using Space and Time complexities. Given algorithm can be analyzed in two ways:

1. **Space Complexity:** The space complexity of an algorithm is the amount of space or memory it needs to run to compilation.
2. **Time Complexity: T**he time complexity of an algorithm is the amount of computer time it needs to run to compilation.

1. **Space Complexity:** The Space needed by each of these algorithms is seen to be the sum of the following component.

   a. A fixed part that is independent of the characteristics (eg: number, size) of the inputs and outputs.
   The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

   b. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

   The space requirement s(p) of any algorithm p may therefore be written as,
   $S(P) = c + S_p$ (Instance Variable)
   Where 'c' is a constant variable.

`

Example 1: Compute Space complexity for the following examples:

```
Algorithm abc(a,b,c)
{
return a+b+c;
}
```

Here, the above algorithm contains three fixed part variables (which requires 3 words of memory), and no variable part (hence 0). Hence S(P) = 3

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 type array of floating point numbers.
- This is at least 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So, we obtain S(P) >= (n + 3)

[n for a[],one each for n,I and s]

2. **Time Complexity:** The time T(p) taken by a program P is the sum of the compile time and the run time(execution time). The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics). Time complexity is done by using Frequency count method i.e. the number of times a statement is executed by the compiler.

   → The number of steps any problem statement is assigned depends on the kind of statement.
   For example,

Comments	→ 0 steps.
Assignment statements	→ 1 steps.
Interactive statement such as for, while & repeat-until	→ Control part of the statement.

Time complexity is classified in 5 types based on frequency count method:

- Constant: This statement will be executed by the compiler only once. For example, c:=a+b;
- Linear: This statement will be executed by the compiler n number of times.

```
for i := 1 to n step do   ----------- n+1 times
   Statement; -----------------------n times
```

- Quadratic: This statement will be executed by the compiler n*n times that is n^2 times.

```
for i := 1 to n step do  ----------------- n+1 times
   for j := 1 to  n step do ---------------n(n+1) times
      Statement; -------------------------n^2 times
```

- Cubic: This statement will be executed by the compiler n*n*n times that is n^3 times.

```
for i := 1 to n step do  ----------------- n+1 times
   for j := 1 to  n step do ---------------n(n+1) times
      for k := 1 to n step do------------n^2(n+1) time
      Statement; -------------------------n^3 times
```

- Logarithmic: For each and every time the work area will be sliced to half. In such cases time complexity will be **log n**.

Time complexity can be expressed in three ways: **Best case, Worst case and Average case.**

If an algorithm takes minimum amount of time to complete for a set of specific inputs it is the Best case. For example, 'key' element is found at beginning of an array in linear search.

If an algorithm takes maximum amount of time to complete for a specific set of inputs it is worst case. For example, 'key' element is found at end of an array or element not found.

If an algorithm takes average amount of time to complete for set of specific inputs it is average case. For example, 'key' element found at middle of an array in linear search.

➔ Compute Space and Time complexity to find Sum of individual digits in a number.

Statement	Time Complexity	Space Complexity
Algorithm Sumofindiviual(n)	-	
{	-	
While n > 0 do	**m**	**1 for n**
{	-	**1 for k**
k:= n % 10;	**m-1**	**1 for s**
n:= n / 10;	**m-1**	
s:= s + k;	**m-1**	
}	-	
Return s;	**1**	
}		
Total = ***4m-2 (where m indicates number of digits in the given number)***		**S(P) = 3**

➔ Compute Space and Time complexity to check given number is Palindrome or not.

Statement	Time Complexity	Space Complexity

Algorithm Palindrome(n)
{
m:= n;
while n > 0 do
{
k := n % 10;
n := n / 10;
s := (s * 10) + k;
}
If s = m then
Write "given number is Palindrome"
Else
Write "Given number is not Palindrome"
}
	-
	-
	1
	m
	-
	m-1
	m-1
	m-1
	-
	1
	1
	
	0
		
		1 for m
		1 for n
		1 for k
		1 for s

Total = ***4m (where m indicates number of digits in the given number)***		**S(P) = 4**

➔ Compute Space and Time complexity to check given number is Armstrong or not.

Algorithm	Time Complexity	Space Complexity
Algorithm armstrong(n) **{** **m:=n; ------------------------------------** **sum := 0; ------------------------------------** **while(n>0) do ------------------------------------** **{** **n := n / 10;-----------------------------------** **k := n % 10;-------------------------------------** **sum := sum + (k * k* k);-----------------------** **}** **if(m = sum)--** **write "Given number is Armstrong";----------** **else** **write "Given number is Not Armstrong";----** **}**	 **1** **1** **m** **m-1** **m-1** **m-1** **1** **1** **0**	 **m - - 1** **n - - 1** **sum - 1** **k - - 1**
Total	**4m + 1**	**S(P) = 4 + 0 = 4**

➔ Compute Space and Time complexity to check given number is Strong or not

Algorithm	Time Complexity	Space Complexity

Algorithm strong(n) **{** **sum := 0;--** **f := 0;--** **for i := 1 to n-1 do--------------------------------** **if (n%i=0) then---------------------------------** **f := f + i;--------------------------------------** **if (f = n) then--------------------------------------** **write "Given number is strong number";-----** **else** **write "Given number is not strong number";** **}**	 **1** **1** **n** **n - 1** **n - 1** **1** **1** **0**	 **sum – 1** **f -------1** **i -------1** **n ------1**
Total	**3n + 1**	**S(P) = 4 + 0 = 4**

➔ Compute Space and Time complexity to check given number is prime or not

Algorithm	**Time Complexity**	**Space Complexity**
Algorithm Prime(n) **{** **for i := 1 to n do --------------------------------** **if (n%i=0) then ---------------------------------** **c++; ---------------------------------------** **if c = 2 then ---------------------------------------** **write "Given number is Prime"; -----** **else** **write "Given number is not Prime number";** **}**	 **n + 1** **n** **n** **1** **1** **0**	 **i -------1** **n ------1** **c ------1**
Total	**3n + 3**	**S(P) = 3 + 0 = 3**

➔ Compute Space and Time complexity to find Fibonacci sequence up to given number.

Algorithm	**Time Complexity**	**Space Complexity**
Algorithm fibonacci(n) **{** **a := 0;---** **b := 1;---** **write a, b;--** **c := a + b;--** **while (c<=n) do-------------------------------------** **{** **write c;--** **a := b;--** **b := c;--** **c := a + b;--** **}** **}**	 **1** **1** **1** **1** **n** **n - 1** **n - 1** **n - 1** **n - 1**	 **m - - 1** **n - - 1** **sum - 1** **k - - 1**
Total	**5n**	**S(P) = 4 + 0 = 4**

➔ Compute Space and Time complexity to find GCD of two numbers.

Algorithm	Time Complexity	Space Complexity

Algorithm GCD(a, b)
{
While a != b do
{
If a > b then
a := a – b;
else
b := b – a;
}
Return a;
}
	
	a
	
	a – 1
	a – 1
	
	0
	
	1
		
		1 for a
		1 for b

Total	**3a – 1(Let *a* is largest among two)**	**S(P) = 2 + 0 = 2**

➔ Compute Space and Time complexity to find factorial of a given number

Statement	Time Complexity	Space Complexity
Algorithm factorial(n) **{** **f=1.0;-------------------------------------** **for i=1 to n do----------------------------** **f:=f * i;-----------------------------------** **return f;---------------------------------------** **}**	**-** **-** **1** **n+1** **n** **1** **-**	 **f – 1** **i – 1** **n – 1**
Total	**2n + 2**	**S(P) = 3 + 0**

➔ Compute Space and Time complexity to find sum of elements present in an array

Statement	Time Complexity	Space Complexity
Algorithm Sum(a,n) **{** **S=0.0;-------------------------------------** **for i=1 to n do----------------------------** **s=s+a[i];-----------------------------------** **return s;---------------------------------------** **}**	**-** **-** **1** **n+1** **n** **1** **-**	 **S – 1** **i – 1** **a[] – n**
Total	**2n + 2**	**S(P) = 2 + n**

➔ Compute Space and Time complexity to perform matrix addition

Algorithm	**Time Complexity**	**Space Complexity**
Algorithm matadd(a,b,c,n) **{** **c[i,j] := 0; ----------------------------------** **for i := 1 to n do ---------------------** **for j := 1 to n do -----------------------** **c[i,j] := c[i,j] + (a[i,j] + b[i,j]) ---------** **return c[i,j]; --------------------------------------** **}**	 **1** **(n+1)** **n(n+1)** **n^2** **1**	 **i ---- 1** **j ---- 1** **n ----1** **a - - n^2** **b - - n^2** **c - - n^2**
Total	**$2n^2 + 2n + 3$**	**S(P) = $3 + 3n^2$**

➔ Compute Space and Time complexity to perform matrix multiplication

Algorithm	**Time Complexity**	**Space Complexity**
Algorithm matmul(a,b,c,n) **{** **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]) ----------** **return c[i,j]; --------------------------------------** **}**	 **n + 1** **n(n+1)** **n^2** **$n^2(n+1)$** **n^3** **1**	 **i ---- 1** **j ---- 1** **k --- 1** **n ----1** **a - - n^2** **b - - n^2** **c - - n^2**
Total	**$2n^3+3n^2+2n+2$**	**S(P) = $4 + 3n^2$**

➔ Compute Space and Time complexity to perform transpose of a matrix

Algorithm	**Time Complexity**	**Space Complexity**
Algorithm mattranspose(a,c) **{** **c[i,j] := 0; ----------------------------------** **for i := 1 to n do ---------------------** **for j := 1 to n do -----------------------** **c[i,j] := a[j,i] ----------** **return c[i,j]; --------------------------------------** **}**	 **1** **(n+1)** **n(n+1)** **n^2** **1**	 **i ---- 1** **j ---- 1** **n ----1** **a - - n^2** **c - - n^2**
Total	**$2n^2 + 2n + 3$**	**S(P) = $3 + 2n^2$**

➔ Compute Space and Time complexity to perform Linear Search

Algorithm	Time Complexity	Space Complexity

Algorithm LS(a, key)
{
for i := 1 to n step 1 do
{
If a[i] = key then
Write "successful search"
Else
Write "unsuccessful search"
}
}
	
	n + 1
	
	n
	1
	
	0
		
		1 for i
		1 for key
		n for a[n]

Total	**2n + 2**	**S(P) = 2 + n**

➔ Compute Space and Time complexity to perform Binary Search

Algorithm	Time Complexity	Space Complexity

Algorithm BS(a, key)
{
Low:=1;
High:=n
While low<=high do
{
Mid:=(low+high)/2;
If a[mid] < key then
Low:=mid+1;
Else if a[mid] > key then
High:=mid – 1;
Else
Return mid;
}
Return 0;
}
	
	1
	1
	n
	
	n-1
	n-1
	n-1
	0
	0
	
	1
	
	0
		
		1 for low
		1 for high
		1 for mid
		1 for n
		1 foe key
		n for a[n]

Total	**4n**	**S(P) = 5 + n**

How to validate Algorithms: Algorithm validation consists of two phases**: Debugging and Profiling**.
Debugging is the process of executing programs on sample data sets to check whether faulty results occur, and if so correct them.

In case, verifying correction of output on sample data fails, the following strategy can be used: Let more than one programmer develop programs for the same problem, and compare outputs produced by those programs. If the outputs match, then there is a good chance that they are correct.

Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to complete the results.

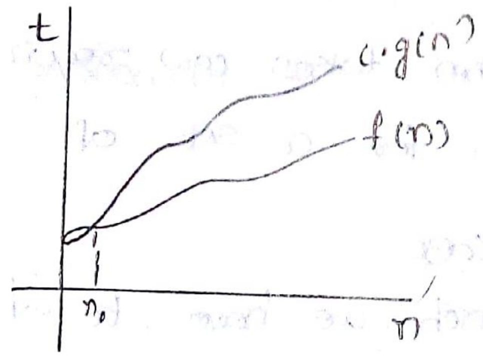
Asymptotic notations: Asymptotic notations are used to express time complexities of algorithms in worst, best and average cases. The following are different types of asymptotic notations which are used.

1. Big – Oh Notation
2. Omega Notation

3. Theta Notation
4. Small Oh Notation
5. Small Omega Notation

1. **Big – Oh Notation: (O)** Big – Oh Notation gives upper bound of an algorithm. This notation describes the Worst case scenario.

   Definition: Let f(n), g(n) are two non-negative functions and there exists positive constants c, n_0 such that f(n) = O(g(n)) iff $f(n) \leq c*g(n)$ for all n, $n \geq n_0$. It is represented as follows.



Examples:

a) Compute Big-Oh notation for f(n) = 3n+2

Ans: Given f(n) = 3n+2

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

$3n+2 \leq 3n + n$ for $n \geq 2$

$3n+2 \leq 4n$ where c = 4, g(n) = n and $n_0=2$

Hence f(n) = O(n)

b) Compute Big-Oh notation for $f(n) = 10n^2+4n+2$

Ans: Given $f(n)= 10n^2+4n+2$

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

$10n^2+4n+2 \leq 10n^2+4n+n$ for $n \geq 2$

$10n^2+4n+2 \leq 10n^2+5n$

$10n^2+4n+2 \leq 10n^2+n^2$ for $n \geq 5$

$10n^2+4n+2 \leq 11n^2$ where c = 11, $g(n) = n^2$ and $n_0=5$

Hence $f(n) = O(n^2)$

c) Compute Big-Oh notation for $f(n) = 1000n^2+100n-6$

Ans: Given $f(n) = 1000n^2+100n-6$

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

$1000n^2+100n-6 \leq 1000\, n^2+100n$ for all values of n

$1000n^2+100n-6 \leq 1000\, n^2+n^2$ for $n \geq 100$

$1000n^2+100n-6 \leq 1001\, n^2$ where c =1001, $g(n) = n^2$ and $n_0=100$

Hence $f(n)=O(n^2)$

d) Compute Big-Oh notation for $f(n) = 6*2^n+n^2$

Ans: Given $f(n) = 6*2^n+n^2$

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

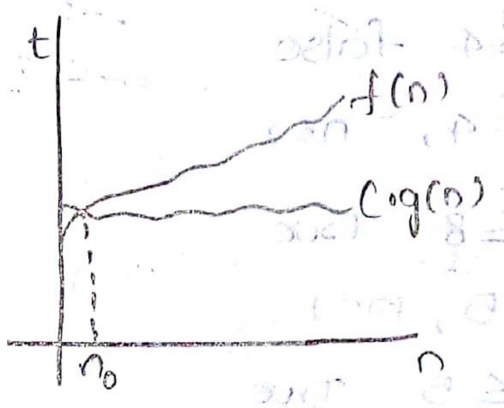
$6*2^n+n^2 \leq 6*2^n+ 2^n$ for $n \geq 4$

$6*2^n+n^2 \leq 7*2^n$ where c = 7, $g(n)=2^n$ and $n_0=4$

Hence $f(n)=O(2^n)$

2. **Omega Notation (Ω):** Omega Notation gives lower bound of an algorithm. This notation describes best case scenario.

Definition: Let f(n), g(n) are two non-negative functions and there exists positive constants c, n_0 such that f(n) = **Ω**(g(n)) iff f(n) ≥c*g(n) for all n, n ≥ n_0. It is represented as follows.



Examples:

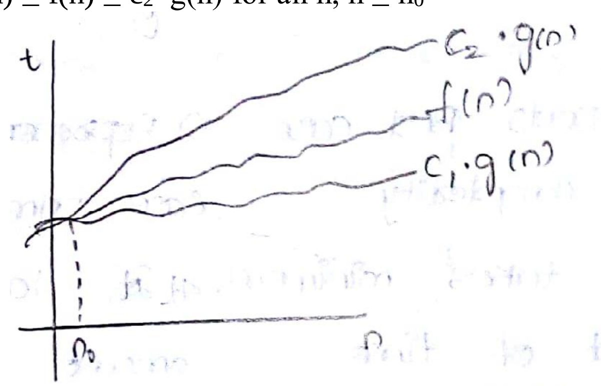
a) Compute omega notation for f(n)=3n+2
   Ans: Given f(n)=3n+2
   f(n) ≥ c * g(n)
   3n+2≥3n   for all values of n (n≥0)
   Where c =3, g(n)=n and n_0=0
   Hence f(n) = **Ω**(n)

b) Compute omega notation for f(n)= $10n^2+4n+2$
   Ans: Given f(n)= $10n^2+4n+2$
   f(n) ≥ c * g(n)
   $10n^2+4n+2 \geq 10n^2$ for all values of n (n≥0)
   Where c=10, g(n)=n^2 and n_0=0
   Hence f(n) = **Ω**(n^2)

c) Compute omega notation for f(n)= $4n^3+2n+3$
   Ans: Given f(n)= $4n^3+2n+3$
   f(n) ≥ c * g(n)
   $4n^3+2n+3 \geq 4n^3$ for all values of n (n≥0)
   Where c=4, g(n)=n^3 and n_0=0

3. **Theta Notation (ɵ):** Theta Notation gives the complexity between lower bound and upper bound. This notation describes the average case scenario.

Definition: Let f(n), g(n) are two non-negative functions and there exists positive constants c_1, c_2, n_0 such that f(n) = **ɵ** (g(n)) iff c_1* g(n) ≤ f(n) ≤ c_2*g(n) for all n, n ≥ n_0



Examples:

a) Compute theta notation for f(n)=3n+2
   Ans: Given f(n)=3n+2
   c_1* g(n) ≤ f(n) ≤ c_2*g(n)

Compute $f(n) \leq c_2 * g(n)$

$3n+2 \leq 3n+n$ for $n \geq 2$

$3n+2 \leq 4n$ where $c_2=2$ and $g(n)=n$

Compute $c_1 * g(n) \leq f(n)$

$3n \leq 3n+2$ for all values of n

Where $c_1=3$, $g(n)=n$

Hence

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

b) Compute theta notation for $f(n)=10n^2+4n+2$

Ans: Given $f(n)=10n^2+4n+2$

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

Compute $f(n) \leq c_2 * g(n)$

$10n^2+4n+2 \leq 10n^2+4n+n$ for $n \geq 2$

$10n^2+4n+2 \leq 10n^2+5n$

$10n^2+4n+2 \leq 10n^2+n^2$ for $n \geq 5$

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

where $c_2=11$ and $g(n)=n^2$

Compute $c_1 * g(n) \leq f(n)$

$10n^2 \leq 10n^2+4n+2$ for all values of n

Where $c_1=10$, $g(n)=n^2$

Hence

$f(n)= \theta(n^2)$

4. **Small Oh Notation (o):** Let f(n), g(n) are two non-negative functions, then we can say that f(n) = o(g(n) iff

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

Examples:

a) Compute theta notation for f(n)=3n+2

Ans: Given f(n)=3n+2

Let g(n)=1

Then $\lim_{n \to \infty}$ **f(n)/g(n) = 3n+2/1=3n+2=** ∞

Let g(n)=n

Then $\lim_{n \to \infty}$ **f(n)/g(n) = 3n+2/n=3+2/n=3**

Let $g(n)=n^2$

Then $\lim_{n \to \infty}$ **f(n)/g(n) =** $\mathbf{3n+2/n^2=3/n+2/n^2=0}$

Hence $f(n) = o(n^2)$

5. **Small Omega Notation (ω):** Let f(n), g(n) are two non-negative functions, then we can say that f(n) = ω(g(n) iff

$$\lim_{n \to \infty} \mathbf{f(n)/g(n) = \infty}$$

Examples:

a) Compute Small Omega notation for f(n)=3n+2

Ans: Given f(n)=3n+2

Let g(n)=1

Then **Lt** **f(n)/g(n) = 3n+2/1=3n+2=** ∞

$\mathbf{n} \to \infty$

Hence, f(n)= $\boldsymbol{\omega(1)}$

Amortized Analysis: Amortized analysis is a method for analyzing a given algorithms complexity. Amortized analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In amortized analysis, sequences of operations are analyzed and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation. If one input is changing the running time of the next set of inputs, use Amortized analysis.

For example, finding n number of K^{th} smallest elements in an array of **n** elements. To solve this, first we have to sort the array of n elements which need **nlogn** time + one second for finding minimum element. So, total amount of time required for first operation is nlogn + 1. The remaining n-1 operations need 1 second each with a total of n-1 seconds. Average amount of time required is given as follows.

Average Time complexity = (nlogn+1+n-1)/n = logn + 1.

The following three different types of techniques are used to compute Amortized complexity.

- **Aggregate method**: Aggregate analysis is a simple method which computes the total cost T(n) for a sequence of n operations, then divide T(n) by the number of n operations to obtain the amortized cost or the average cost in the worst case. i.e. T(n)/n.
- **Accounting method:** In this method, assign different charges to different operations, with some operations charged more or less than they actually cost. The amount we charge on operation is called Amortized cost. The excess charge will be deposited into the data structure called Credit. i.e Credit = Amortized cost – Actual Cost
  This credit can be used later for operations whose amortized cost is less than their actual cost.

Let $\hat{c}_i$ is the amortized cost of ith operation
c_i is the actual cost of ith operation.

then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$, for all n operations.

Total Credit = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$ and credit ≥ 0.

- **Potential Functional Method:** In this method, after performing the operation the change is captured as a data structure Credit. The function that captures the change is known as potential function. If the change in potential is non-negative, then that operation is over charged, the excess potential will be stored at the data structure. If the change in the potential is negative, then that operation is under charged which would be compensated by excess potential available at the data structure.

Let c_i denote the actual cost of ith operation, and $\hat{c}_i$ denote amortized cost of ith operation. if $\hat{c}_i > c_i$, then the ith operation leaves some positive amount of credit, the credits $\hat{c}_i - c_i$ can be used up by future operations. As long as

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \sim (1)$$

the total available credit will always be non-negative, and the sum of amortized costs will be an upper bound on the actual cost.

In the potential method, the amortized cost of operation i is equal to the actual cost plus the increase in potential due to that operation.

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1} \sim (2).$$

from (1) & (2) $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \phi_i - \phi_{i-1})$.

Frequently Asked Questions

1. Define an algorithm. What are the different criteria that satisfy the algorithm?
2. Explain pseudo code conventions for writing an algorithm.
3. Explain how algorithms performance is analyzed? Describe asymptotic notation?
4. What are the different techniques to represent an algorithm? Explain.
5. Explain recursive algorithms with examples.
6. Distinguish between Algorithm and Psuedocode.
7. Give an algorithm to solve the towers of Hanoi problem.
8. Write an algorithm to find the sum of individual digits of a given number
9. Explain the different looping statements used in pseudo code conventions.
10. What is meant by recursion? Explain with example, the direct and indirect recursive algorithms.
11. What is meant by time complexity? What is its need? Explain different time complexity notations. Give examples one for each.
12. Describe the performance analysis in detail
13. Discuss about space complexity in detail.
14. Define Theta notation. Explain the terms involved in it. Give an example
15. Determine the running time of merge sort for
i) Sorted input ii) reverse-ordered input iii) random-ordered input
16. Explain about two methods for calculating time complexity.
17. Show that f(n) = 4n2 - 64n + 288 = **Ω** (n2).

18. Present an algorithm for finding Fibonacci sequence of a given number.
19. Write the non-recursive algorithm for finding the fibonacci sequence and derive its time complexity.
20. Compare the two functions n2 and 2n/4 for various values of n. Determine when the second becomes larger than the first.
21. Determine the frequency counts for all statements in the following algorithms.

```
i) for i:=1 to n do
for i:=1 to i do
for k:=1 to j do
x:= x+1;
ii) i := 1;
while (i<=n) do
{
 x : = x + 1;
i : = i + 1;
}
```

21. Calculate the time complexity for matrix multiplication algorithm.
22. Calculate the time complexity for Armstrong number algorithm
23. Explain about different Asymptotic Notations with two examples
24. Find the time complexity for calculating sum of given array elements.
25. Calculate space and time complexity for matrix multiplication algorithm
26. Write an algorithm for Armstrong number and also calculate space and time complexity?
27. Write an algorithm for strong number and also calculate space and time Complexity?
28. Describe the Algorithm Analysis of Binary Search.
29. Differentiate between Big-oh and omega notation with example.
30. Write short note on amortized analysis.

3. Divide and Conquer

General Method: In Divide and Conquer method, a given problem is

→ Divided into smaller sub problems.
→ these subproblems are solved independently.
→ Combine all the solutions of sub problems into a single solution.
→ If the sub problems are large enough then again Divide and Conquer is applied.

Algorithm for Divide and Conquer: A Control abstraction for Divide and Conquer general method is as shown below.

```
Algorithm 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 subproblems.
     return Combine (DAndC(P1), DAndC(P2).. DAndC(Pk));
   }
}.
```

Here Small(P) is a Boolean valued function which determines whether the input size is small enough. If it is small, then solution is returned.

Otherwise, the problem P is divided into smaller subproblems $P_1, P_2 \cdots P_K$. these are solved by recursive applications of DAndC. Combine is a function that determines the solution 'P' using the solutions to the 'K' subproblems.

*. Recurrence Relation: the Recurrence Relation is an equation that defines a sequence recursively. The Recurrence Relation can be solved by the following methods

1. Substitution Method.
2. Master's Method.

1. Substitution Method: there are two types of substitution methods.

(a) Forward Substitution: This method makes use of initial condition and value for the next term is generated. For example, Consider a recurrence relation.

$$T(n) = T(n-1)+n \quad \text{with } T(0)=0.$$

If $n=1 \Rightarrow T(1) = T(0)+1. = 1. \quad = 1$

$n=2 \Rightarrow T(2) = T(1)+2 \quad = 1+2. \quad = 3$

$n=3 \Rightarrow T(3) = T(2)+3 \quad = 1+2+3 \quad = 6$

$\vdots \qquad \vdots \qquad = 1+2+\dots+n.$

$n=n \Rightarrow T(n) = \dots$

By observing the above generated equation we can derive a formula

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

(b) Backward Substitution: In this method backward values are substituted recursively in order to derive some formula. For example, Conside a recurrence relation

$$T(n) = T(n-1)+n. \quad \text{with } T(0)=0. \sim (1)$$

$$\Rightarrow T(n-1) = T(n-1-1)+n-1 = T(n-2)+(n-1) \sim (2)$$

Substitute $\sim(2)$ in $\sim(1)$

$\Rightarrow T(n) = T(n-2) + (n-1) + n \sim ③.$

Let $T(n-2) = T(n-2-1) + (n-2) \sim ④.$

Sub $\sim ④$ in $\sim ③$

$\Rightarrow T(n) = T(n-3) + (n-2) + (n-1) + n.$

$\vdots$

$T(n) = T(0) + (n-n) + n-(n-1) + \cdots + n.$

$= 0 + 1 + 2 + \cdots + n.$

$T(n) = \frac{n(n+1)}{2} = \underline{O(n^2)}.$

② **Master's Method:** Consider the following recurrence relation:

$$T(n) = aT(n/b) + F(n).$$

then Master's theorem can be stated as -

If $F(n)$ is $O(n^d)$ where $d \geq 0$ in the RR then.

(i) $T(n) = O(n^d)$ if $a < b^d$.

(ii) $T(n) = O(n^d \log n)$ if $a = b$.

(iii) $T(n) = O(n^{\log_b a})$ if $a > b^d$.

For example, $T(n) = 4T(n/2) + n$. Compare with

$T(n) = aT(n/b) + f(n).$

Here $a=4, b=2, f(n) = n^1$. where $d=1$.

$\because a > b^d.$ $T(n) = O(n^{\log_b a})$

$= O(n^{\log_2 4}) = O(n^2).$

Another variation of Master theorem is for

$$T(n) = aT(n/b) + f(n).$$

(i). If $f(n)$ is $O(n^{\log_b a})$ then $T(n) = O(n^{\log_b a})$.

(ii) If $f(n)$ is $O(n^{\log_b a} \log^k n)$ then $T(n) = O(n^{\log_b a} \log^{k+1} n)$.

~~(iii) If f(n) is~~ For example, Consider

$$T(n) = 2T(n/2) + n\log n.$$

Here $f(n) = n\log n$. where $a=2, b=2, k=1$

According to case (ii)

$$T(n) = O(n^{\log_b a} \log^{k+1} n).$$

$$= O(n^{\log_2 2} \log^2 n)$$

$$= O(n\log^2 n).$$

For more examples refer class notes

(*) **Analysis of D and C General Method:** The computing time of Divide and Conquer is given by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small.} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & \text{otherwise.} \end{cases}$$

Where $T(n)$ is the time for DAnd C on any input of size n and $g(n)$ is the time to compute the solution directly for small inputs. The function $f(n)$ is the time for dividing P and combining the solutions to subproblems.

The complexity of many divide and conquer algorithms is given by recurrence relation.

$$T(n) = \begin{cases} T(1) & \text{if } n=1 \\ aT(n/b) + f(n) & \text{if } n>1 \end{cases}$$

Where a and b are constants.

Let $n = b^k$

$\therefore T(b^k) = a\,T(b^k/b) + f(b^k)$

$T(b^k) = a\,T(b^{k-1}) + f(b^k)$. ~ ①.

From ~①

$T(b^{k-1}) = a\,T(b^{k-2}) + f(b^{k-1})$ ~②

Sub stitute ~② in ~① we get

$T(b^k) = a\left[a\,T(b^{k-2}) + f(b^{k-1})\right] + f(b^k)$.

$= a^2 T(b^{k-2}) + a\,f(b^{k-1}) + f(b^k)$. ~③

By substituting $T(b^{k-2}) = a\,T(b^{k-3}) + f(b^{k-2})$ in ~③

$$T(b^k) = a^3 T(b^{k-3}) + a^2 f(b^{k-2}) + a f(b^{k-1}) + f(b^k).$$

$$\vdots$$

$$= a^k T(b^{k-k}) + a^{k-1} f(b^1) + a^{k-2} f(b^2) + \cdots + a^0 f(b^k).$$

$$= a^k T(1) + \frac{a^k}{a} f(b) + \frac{a^k}{a^2} f(b^2) + \cdots + \frac{a^k}{a^k} f(b^k).$$

$$= a^k \left[T(1) + \frac{f(b)}{a} + \frac{f(b^2)}{a^2} + \cdots + \frac{f(b^k)}{a^k} \right]$$

$$= a^k \left[1 + \sum_{j=1}^{k} \frac{f(b^j)}{a^j} \right]. \qquad \because b^k = n. \quad \therefore k = \log_b n.$$

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

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

(*). **Applications of Divide and Conquer:** Various applications of Divide and conquer strategy are

1. Binary Search.
2. Merge Sort
3. Quick Sort
4. Strassen's Matrix Multiplication.

(1) Binary Search: Let a_i, $1 \le i \le n$, be a list of elements that are sorted in ascending order. Consider the problem of determining whether a given element x is present in the list. If x is present, determine a value j such that $a_j = x$. If x is not in the list, then j is set to be zero.

Let $P = (n, a_i, \cdots, a_l, x)$ ~~dento~~ denote an arbitary instance of this search problem, where n is the number of elements in the list, $a_i, \cdots a_l$ is the list of elements, and x is the searching element.

According to Divide and Conquer, if the problem is small $(n=1)$ then take the value i if $x = a_i$; otherwise it will take the value 0.

If P has more than one element, it can be divided into sub problems. Pick an index q in the range $[i, l]$, compare x with a_q. If $x = a_q$, the problem P is solved, If $x < a_q$, then search for x in the sublist $a_i, a_{i+1}, \cdots, a_{q-1}$. If $x > a_q$, then search for x in the sublist $a_{q+1}, \cdots a_l$. Now, the given problem P is divided into the following two sub problems.

$$P = (n, a_i, \cdots, a_l, x)$$

$$\begin{array}{ccc} & P = (n, a_i, \cdots, a_l, x) & \\ \swarrow^{x < a_q} & & \searrow^{x > a_q} \\ P(q-i, a_i, \ldots a_{q-1}, x) & & P(l-q, a_{q+1} \cdots a_l, x) \end{array}$$

To obtain solution, repeatedly apply DAnd C on each subproblems.

⊛ Algorithm for Recursive Binary Search:

```
Algorithm BinSearch(a, i, l, x)
// Given an array a[i:l] of elements in ascending order,
// 1≤i≤l, determine whether x is present, and if so;
// return j such that x = a[j]; else return 0.
{
   if (l=i) then
   {
      if (x= a[i]) then return i;
      else return 0;
   }
   else
   {
      mid := ⌊(i+l)/2⌋;
      if (x= a[mid]) then return mid;
      else if (x< a[mid]) then
            return BinSearch(a, i, mid-1, x);
      else return BinSearch(a, mid+1, l, x);
   }
}//
```

* Algorithm for Iterative & Non-Recursive Binary Search:

```
Algorithm BinSearch (a,n,x)
// Given an array a[i:l] of elements in ascending order,
// n≥0, determine whether x is present, and if so, return
// j such that x = a[j]; else return 0.
{
  low:=1; high:=n;
  while (low ≤ high) do
  {
    mid:= ⌊(low+high)/2⌋;
    if (x< a[mid]) then high:=mid-1;
  else if (x > a[mid]) then low: =mid+1;
    else return mid;
  }
  return 0;
}//.
```

Ex: Apply Divide and Conquer strategy for the following input values for searching 112, −14 by showing the values of low, mid, high for each search.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
−15	−6	0	7	9	23	54	82	101	112	125	131	142	151

Sol: For $x = 112$.

low	high	mid
1	14	7
8	14	11
8	10	9
10	10	10 found.

For $x = -14$.

low	high	mid
1	14	7
1	6	3
1	2	1
		not found.

Ex: ②: Explain the method for searching the element 94 following set of elements using Binary Search. Also draw binary decision tree for the same.

10, 12, 14, 16, 18, 20, 25, 30, 35, 38, 40, 45, 50, 55, 60, 70, 80, 90

Sol: Given that $i=1$, $l=18$, $x=94$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
10	12	14	16	18	20	25	30	35	38	40	45	50	55	60	70	80	90

Now pick the index $q = \left\lfloor \frac{(i+l)}{2} \right\rfloor = \left\lfloor \frac{1+18}{2} \right\rfloor = 9$

Compare x and $a[q]$. Since $x > a[q]$ divide the given array into two sub arrays and continue search in second sub array.

Now $i = q+1 = 10$, $l=18$, $x=94$.

Now pick the index $q = \left\lfloor \frac{(i+l)}{2} \right\rfloor = \left\lfloor \frac{(10+18)}{2} \right\rfloor = 14$.

Compare x and $a[q]$. Since $x > a[q]$ divide the array into two sub arrays and continue search in second sub array.

Now $i = q+1 = 15$, $l=18$, $x=94$.

Now pick the index $q = \left\lfloor \frac{(i+l)}{2} \right\rfloor = \left\lfloor \frac{(15+18)}{2} \right\rfloor = 16$

$\because x > a[q]$. continue search in second sub array.

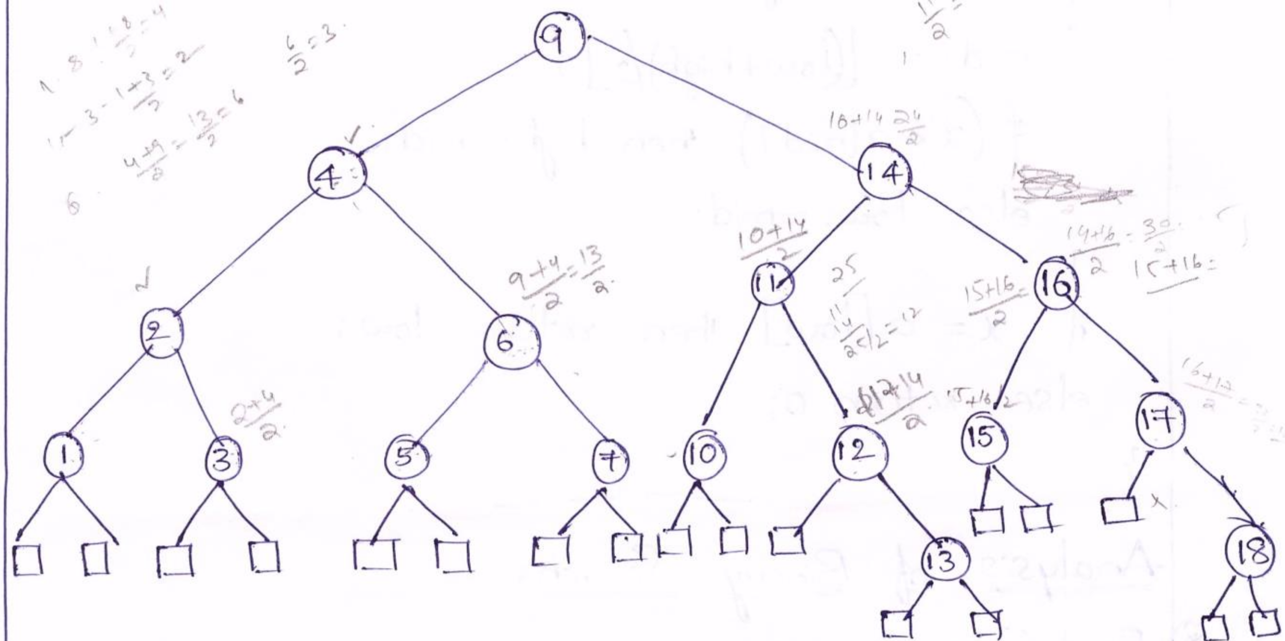
$\therefore i = q+1 = 17$, $l=18$, $x=94$. $\therefore q = \left\lfloor \frac{(17+18)}{2} \right\rfloor = 17$

$\because x > a[q]$ continue search in second sub array.

$\therefore i = q+1 = 18$, $l=18$, $x=94$ $\therefore q = \frac{(18+18)}{2} = 18$.

$\because x > a[q]$. element is not found.

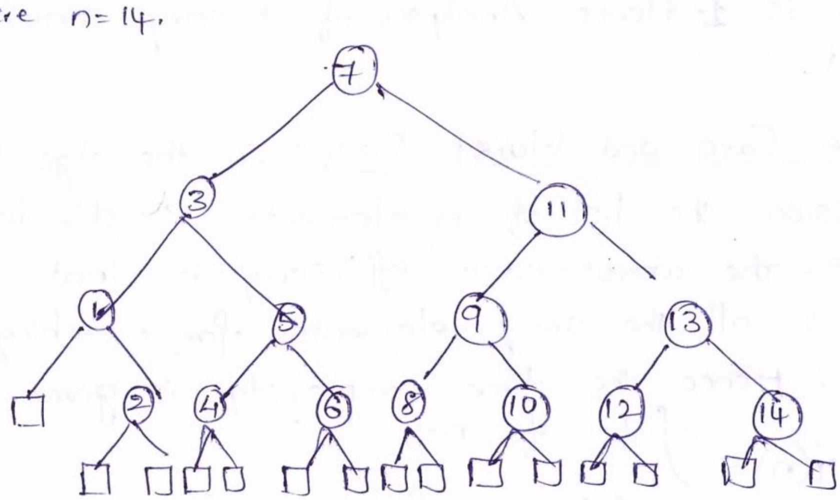
Binary Decision Tree: Binary Decision tree contains: circular nodes and square nodes. Every successful search ends at circalar node and unsuccessful search ends at square node. Binary Decision tree for the above problem n=18 is as shown below.



Ex: Draw Binary Decision Tree for the following elements:

−15, −6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151.

Sol: Here n = 14.



(*) Present an Algorithm for Binary Search using one comparison per cycle.

```
Algorithm BinSearch1(a,n,x)
{
  low:=1; high:=n+1;
  while (low<high-1) do
  {
    mid:= ⌊(low+high)/2⌋;
    if (x < a[mid]) then high:=mid;
    else low:=mid;
  }
  if x= a[low] then return low;
  else return 0;
}
```

Analysis of Binary Search:

(*) **Best Case:** The basic operation in binary search is comparison of search key with array element. If the search key is found at middle of the array, total no: of comparisons required is 1. Hence, Analysis of binary search in best case is $O(1)$.

(*) **Average Case and Worst Case:** In the algorithm after one comparision the list of n elements is divided into $n/2$ sublists. the worst case efficiency is that the algorithm compares all the array elements for searching the desired element. Hence the time complexity is given by

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2)+1 & \text{if } n>1 \end{cases}$$

$T(n) = T(n/2) + 1 \sim ①$

$\Rightarrow T(n/2) = T(n/4) + 1 \sim ②$

Substitute $\sim ②$ in $\sim ①$ we get

$T(n) = T(n/4) + 1 + 1 = T(n/4) + 2 \sim ③$

$T(n/4) = T(n/8) + 1$

By substituting $T(n/4)$ value, we will get

$T(n) = T(n/8) + 1 + 2$

$\vdots$

$T(n) = T(n/16) + 4$ Let $2^4 = n$

$4 = \log_2 n$.

$= T(n/n) + \log_2 n$.

$= 1 + \log_2 n$.

$\therefore$ $O(\log n)$.

(*) **Merge Sort:** the Merge Sort is a sorting algorithm tha uses the divide and conquer strategy. In merge Sort, Given a sequence of n elements $a[1], a[2] \cdots a[n]$ will be split into two sets $a[1], a[2] \cdots a[n/2]$ and $a[n/2+1], \cdots a[n]$. Each set is individually sorted, and the resultin sorted sequences are merged to produce a single sorted sequence of n elements.

Algorithm for Merge Sort:

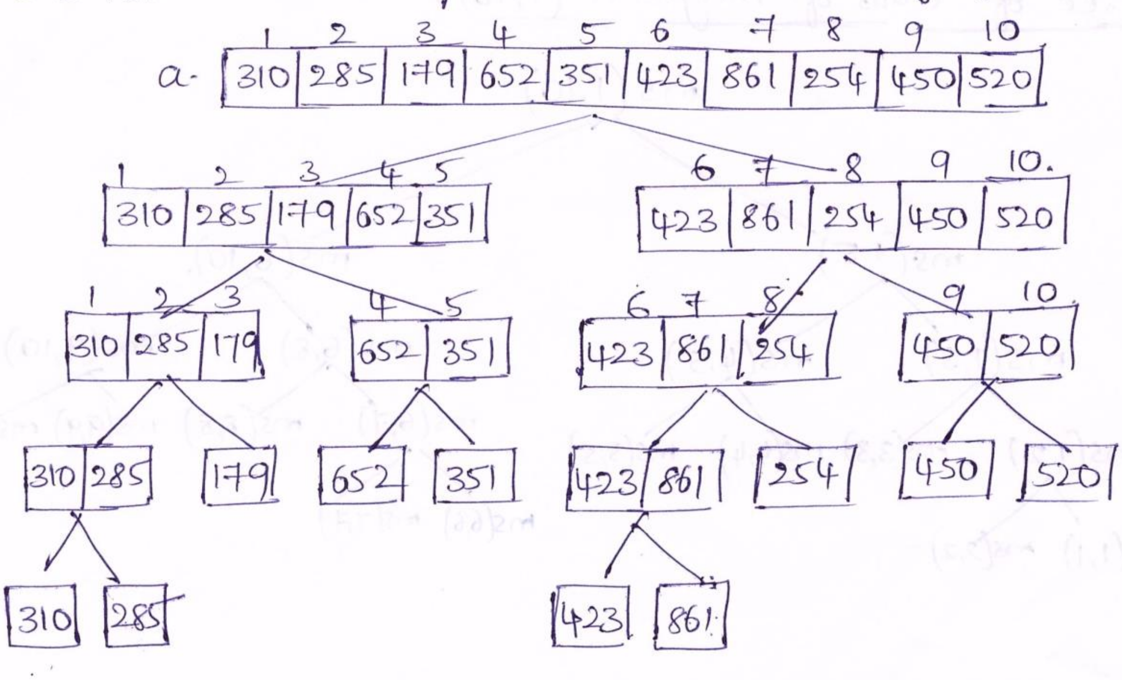
```
Algorithm MergeSort(low, high)
// a[low:high] is a global array to be sorted.
// Small(P) is true if there is only one element to sort.
{
    if (low < high) then
    {
        mid = ⌊(low+high)/2⌋;
        MergeSort (low, mid);
        MergeSort (mid+1, high);
        Merge (low, mid, high);
    }
}
Algorithm Merge(low, mid, high)
{
    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;
        }
```

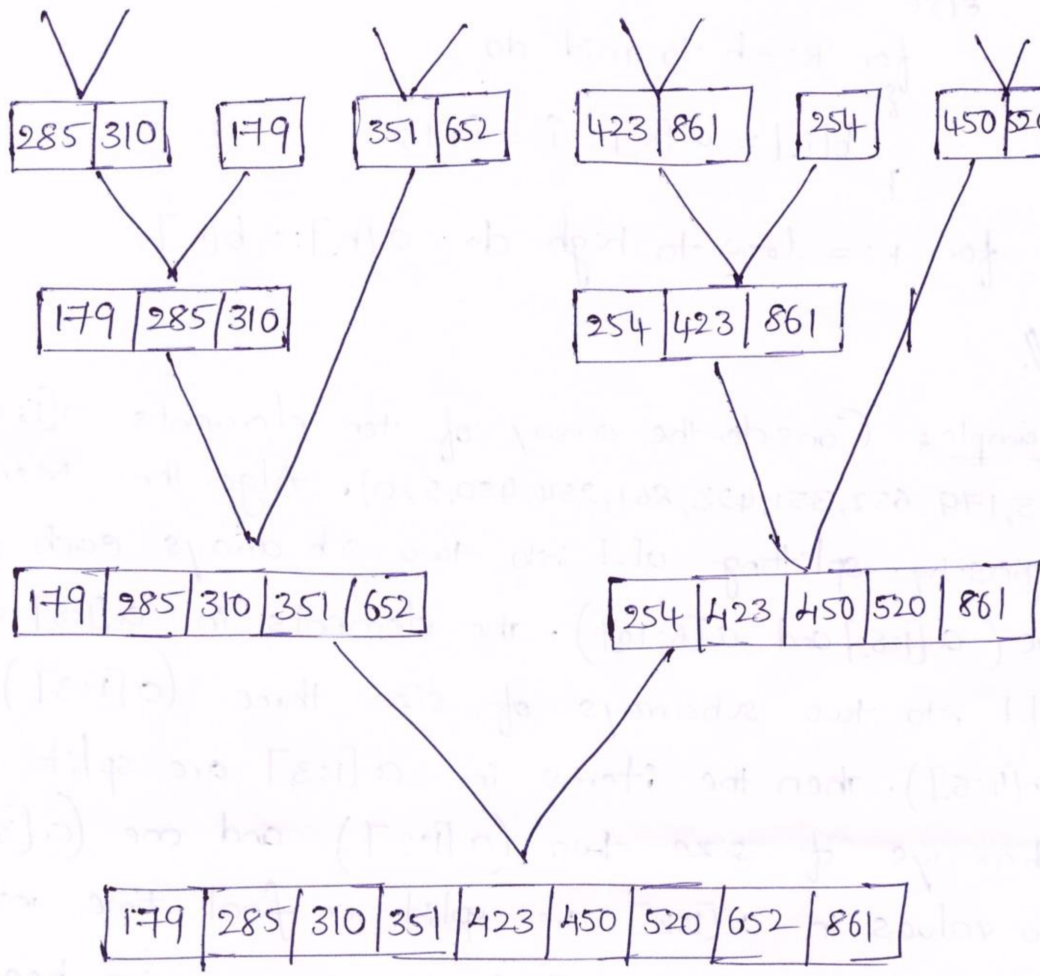
```
else
    for k:=h to mid do
    {
      b[i] = a[k]; i:=i+1;
    }
 for k:= low to high do a[k]:=b[k];
}//.
```

$k=10$ $\frac{1+10}{2}$ $11/2$ $=5$

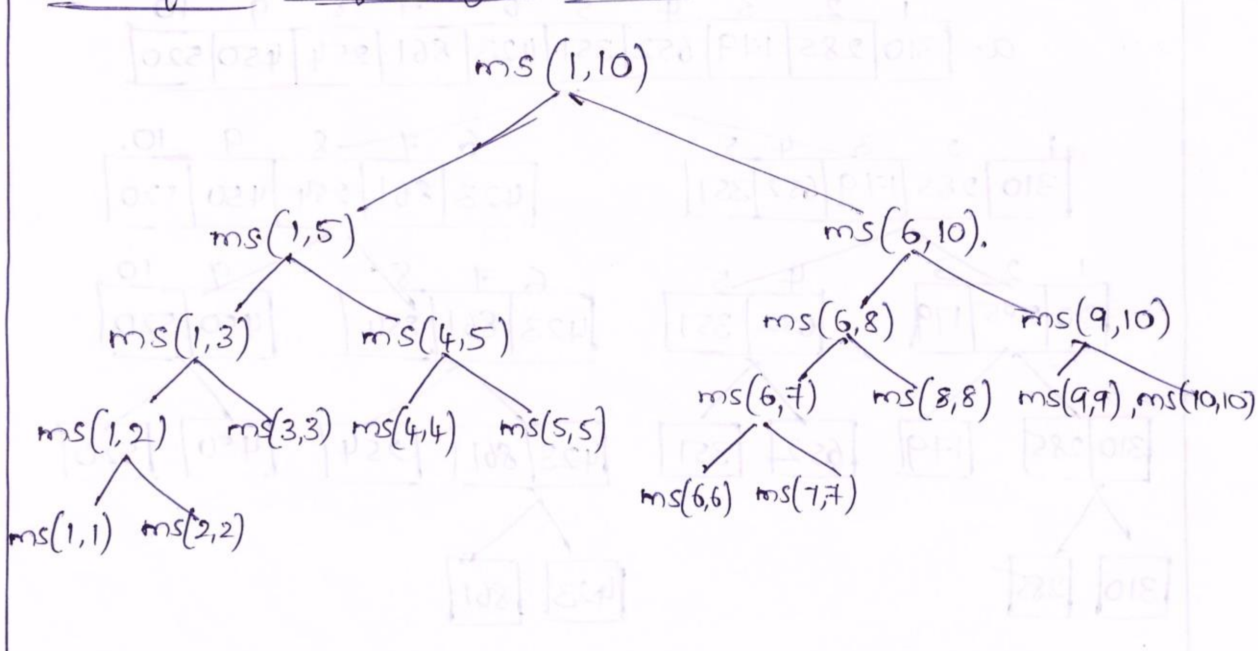
a[1:5] a[6:10] a[6:8] a[9:10] a[1:3] a[4:5] a[6:7] a[8] a[1:2] a[3:3]

Example: Consider the array of ten elements a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520). Algorithm MergeSort begins by splitting a[] into two sub arrays each of size five (a[1:5] and a[6:10]). the elements in a[1:5] are then split into two subarrays of size three (a[1:3]) and two (a[4:5]). then the items in a[1:3] are split into two subarrays of size two (a[1:2]) and one (a[3:3]). the two values in a[1:2] are split a final time into one-elemented sub arrays, and now the merging begins.





Tree of Calls of MergeSort (1, 10):



Tree of calls of Merge (1,10):

merge(1,1,2) merge(6,6,7)

merge(1,2,3) merge(4,4,5) merge(6,7,8) merge(9,9,10)

merge(1,3,5) merge(6,8,10)

merge(1,5,10).

Ex: Draw tree of calls of merge for the following set
35, 25, 15, 10, 45, 75, 85, 65, 55, 5, 20, 18.

Sol:

ms(1,12)

ms(1,6) ms(7,12)

ms(1,3) ms(4,6) ms(7,9) ms(10,12)

ms(1,2) ms(3,3) ms(4,5) ms(6,6) ms(7,8) ms(9,9) ms(10,11) ms(12,12)

ms(1,1) ms(2,2) ms(4,4) ms(5,5) ms(7,7) ms(8,8) ms(10,10) ms(11,11)

merge(1,1,2) merge(4,4,5) merge(7,7,8) merge(10,10,11)

merge(1,2,3) merge(4,5,6) merge(7,8,9) merge(10,11,12)

merge(1,3,6) merge(7,9,12)

merge(1,6,12).

* Analysis of Merge Sort : (Best Case, Average Case, Worst Case).

In merge sort algorithm two recursive calls are used. Each recursive call focuses on $n/2$ elements of the list. One recursive call is used to divide the elements, otherisused to combine the two sublists. Hence we can write recurrence relation as.

$T(n) =$ Time taken to sort left sublist + Time taken to sort right sublist + Time taken for combining two sublists

$\therefore T(n) = T(n/2) + T(n/2) + n.$

$T(n) = 2T(n/2) + n.$ if $n > 1$ ~①.

$T(n). = 1$ if $n = 1$

$T(n/2) = 2T(n/4) + (n/2)$ ~②

Substitute ~② in ~①

$T(n) = 2\left[2T(n/4) + \frac{n}{2}\right] + n = 4T(n/4) + n + n$

$T(n) = 4T(n/4) + 2n$ ~③

By substituting $T(n/4)$ in ~③ we will get

$T(n) = 4\left[2T(n/8) + \frac{n}{4}\right] + 2n.$

$= 8T(n/8) + 3n.$ Let $2^3 = n.$

$\therefore 3 = \log_2 n.$

$= nT(n/n) + \log n * n.$

$= n + n\log n \quad \therefore \quad O(n\log n).$

Note: Is Merge Sort is Stable Sorting?

Ans: A sorting technique is said to be stable if at end of the method identical elements are in same order as they are in original unsorted set. Hence, Merge Sort is stable sort.

* Determine the running time of merge sort for.
 (i) Sorted input (ii) Reverse-Ordeded input (iii) Random-Ordered i/p.

Ans: the Merge Sort is the only algorithm whose time complexity does not get affected by the ordering of the input. Hence.

In Sorted input, Reverse-Ordered input and Random-Ordered input time complexity is $O(n\log n)$.

* Quick Sort: Quick Sort is a sorting algorithm that uses the divide and conquer strategy. the steps for quick sort are as follows:

- Divide: Split the array into two arrays such that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element. the splitting of the array into two sub arrays is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.
- Conquer: Recursively sort the two sub arrays.
- Combine: Combine all the sorted elements in a grap to form a list of sorted elements.

Algorithm for Quick Sort:

```
Algorithm QuickSort(p,q)
{
  if (p<q) then
  {
    j:= Partition (a,p,q+1);
    Quick Sort(p,j-1);
    Quick Sort(j+1, q);
  }
}
Algorithm Partition(a,m,p)
{
  v:=a[m]; i:=m; j:=p;
  repeat
  {
    repeat
      i:=i+1;
    until (a[i]≥v);
    repeat
      j:=j-1;
    until (a[j]≤v);
    if (i<j) then Interchange(a,i,j);
  }until (i≥j);
  a[m]:=a[j]; a[j]:=v; return j;
}
Algorithm Interchange(a,i,j)
{
  p:=a[i];
  a[i]:=a[j];
  a[j]:=p;
}//
```

10	20	30	40	50	70	80	90

this is the sorted list.

* Analysis of Quick Sort:

* **Best Case and Average Case:** If the array is always partitioned at the mid, then it brings the best case efficiency of an algorithm. then the recurrence relation is given by

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

= Time required to sort left sub array +
Time required to sort right sub array +
Time required for partitioning the sub array.

$$\therefore T(n) = \begin{cases} 1 & \text{if } n = 1. \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$T(n) = 2T(n/2) + n \sim ①$$

$$\therefore T(n/2) = 2T(n/4) + (n/2) \sim ②$$

By substituting ~② in ~① we will get

$$T(n) = 2\left[2T(n/4) + (n/2)\right] + n$$

$$T(n) = 4T(n/4) + 2n. \sim ③$$

By substituting $T(n/4)$ in ~③ we will get

$$T(n) = 8T(n/8) + 3n.$$

Let $2^3 = n$

$\therefore 3 = \log_2 n$.

$$T(n) = nT(n/n) + \log_2 n \times n.$$

$T(n) = n + n\log_2 n = \underline{O(n\log_2 n)}.$

<u>Worst Case</u>: the worst case for quick sort occurs when the pivot is a minimum or maximum of all the elements in the list. Hence recurrence relation is given as

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+n & \text{if } n>1. \end{cases}$$

$\therefore T(n) = T(n-1) + n \sim (1)$

$T(n-1) = T(n-2) + (n-1) \sim (2)$

By substituting $\sim(2)$ in $\sim(1)$ we get

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

By substing $T(n-2)$ in $\sim(3)$ we get

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

$\vdots$

$= T(n-n) + n-(n-1) + n-(n-2) + \cdots n$

$= 0 + 1 + 2 + \cdots + n.$

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

$= \frac{n^2+n}{2} = \frac{n^2}{2} + \frac{n}{2}.$

$\therefore O(n^2).$

* **Randomised Quick Sort:** The worst case for quick sort depends upon the selected pivot element. If min or maximum element in the array is choosed as pivot results in worst case. The following are different methods to choose pivot element which improves the performance of quick sort.

→ Use middle element of the array as pivot
→ Use a random element of the array as pivot
→ Take median of first, last and middle elements as a pivot.

Randomized Quick Sort Algorithm:

```
Algorithm RQuickSort(p,q).
{
   if (p<q) then
   {
      if ((q-p) >5) then
         Interchange(a, Random() mod (q-p+1)+p, p);
      j:= Partition(a,p,q+1);
      RQuickSort(p,j-1);
      RQuickSort(j+1,q);
   }
}//.
```

Finding Maximum and Minimum element in an Array:

Let $P = (n, a[i], \ldots a[j])$ denote an arbitary instance of the problem. Here n is the number of elements in the list $a[i] \cdots a[j]$ and we have to find maximum and minimum of this list.

Let small (P) be true when $n \leq 2$. In this case, the maximum and minimum are $a[i]$ if $n=1$. If $n=2$, the problem can be solved by making one comparision.

Divide: If the list has more than two elements, P has to be divided into smaller instances. For example, P might divide into two instances $P1 = \left(n/2, a[i], \ldots a[n/2]\right)$ and $P2 = \left(n/2, a[n/2+1] \cdots a[j]\right)$. After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

Conquer: Let Max (P) and Min (P) be the maximum and minimum elements of P, then Max (P) is the larger of Max (P1) and Max (P2), Min (P) is the smaller of Min (P1) and Min (P2).

Algorithm

```
Algorithm MaxMin (i, j, max, min)
{
  if i=j then max := min := a[i];
  else if i=j-1 then
   {
     if a[i] < a[j] then max := a[j]; min := a[i];
     else
          max := a[i]; min := a[j];
   }
   else
   {
     mid := (i+j)/2;
     MaxMin (i, mid, max, min);
     MaxMin (mid+1, j, max1, min1);
     if max < max1 then max := max1;
     if min > min1 then min := min1;
   }
}
```

Analysis: Computing time of Finding maximum and minimum element in the array is given the following Recurrence Relation.

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ T(n/2)+T(n/2)+2 & \text{if } n>2 \end{cases}$$

$T(n) = 2T(n/2) + 2 \sim ①$

From $\sim ①$ $T(n/2) = 2T(n/4) + 2 \sim ②$

Substitute $\sim ②$ in $\sim ①$

$\Rightarrow T(n) = 2[2T(n/4) + 2] + 2 = 4T(n/4) + 4 + 2 \sim ③$

From $\sim ①$ $T(n/4) = 2T(n/8) + 2$ — ④

Substitute $\sim ④$ in $\sim ③$

$\Rightarrow T(n) = 4[2T(n/8) + 2] + 4 + 2$

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

$\vdots$

$\because 2^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$

$2^1 + 2^2 + \cdots + 2^k = \underline{\underline{2^{k+1} - 2}}.$

$= 2^{k-1} T\left(\frac{2^k}{2^{k-1}}\right) + 2^k - 2$ there are k-1 terms so $= 2^k - 2$.

For simplification Let $n = 2^k$

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

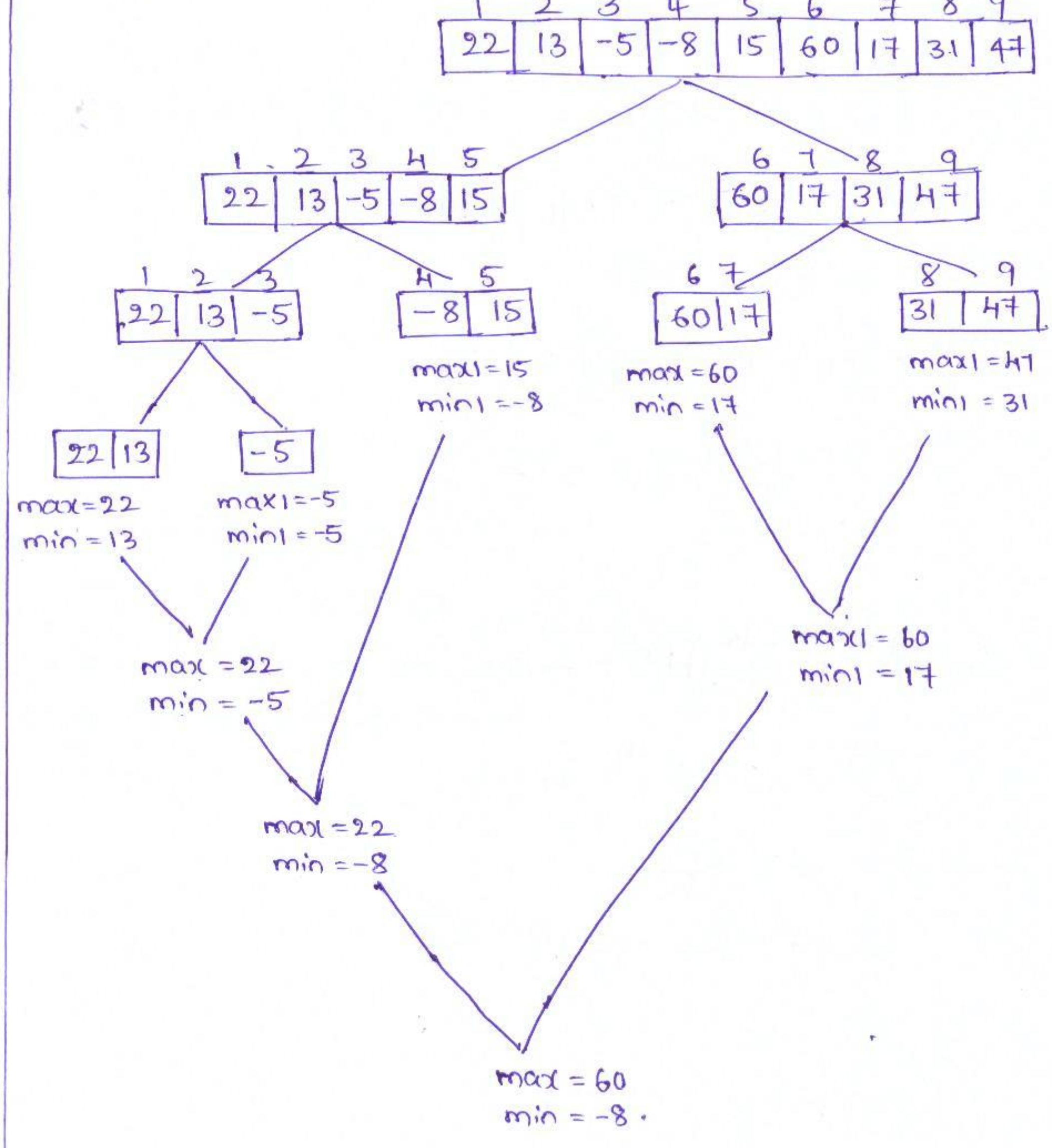
$= 2^{k-1} + 2^k - 2$

$= \frac{n}{2} + n - 2 \quad \because n = 2^k.$

$= \frac{3n}{2} - 2$

$= O(n)$ //.

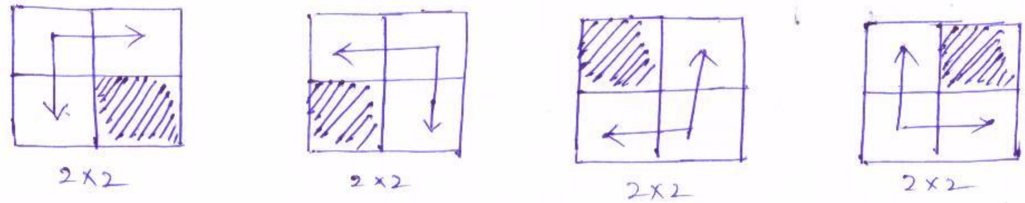
Ex: Find maximum and minimum element in the array using D and C : 22, 13, −5, −8, 15, 60, 17, 31, 47



* **Defective Chessboard:** Consider a $n \times n$ chessboard of the form $n = 2^k, k \geq 1$ with one defective cell. Fill the board using L shaped tiles called triominoes. A L shaped tile is a 2×2 board with one defective cell. A triominoe has the following four orientations.

Our aim is to place $(n^2-1)/3$ triominoes on an $n \times n$ defective chessboard so that all n^2-1 nondefective positions are covered. this problem can be solved using divide and conquer as follows.

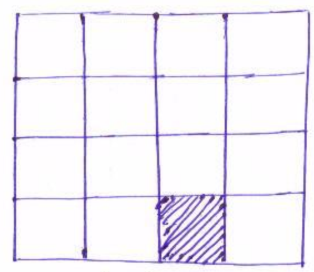
① If $n=2$, A 2×2 square with one missing cell is nothing but problem is small. In this case, missed cell is already covered with a L shaped triominoe as shown below.



② If $n > 2$, i.e for 4×4, 8×8, 16×16 etc.. place a L shaped triminoe at the center such that it does not cover the $n/2 * n/2$ subsquare that has a missing cell. Now all four squares of size $n/2 * n/2$ has a defective cell.

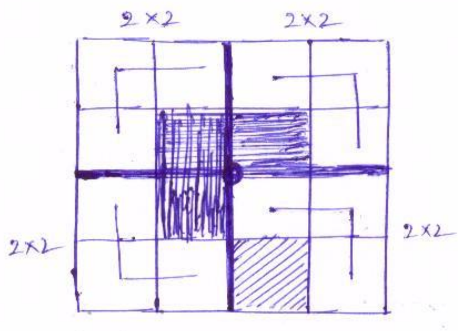
③ Solve the problem recursively for four $n/2 * n/2$ subsquares.

4x4 Defective Chessboard: Consider 4x4 defective chessboard as shown below.



4x4

∵ $n > 2$, divide the 4x4 chessboard into four 2x2 chessboards by placing a L shaped triominoe at the center which doesnot cover defective 2x2 chessboard as shown below.



∵ $n=2$ for each subproblem, in each subsquare defective cell is automatically covered with a L shaped triominoe.

Algorithm for Defective Chessboard:

```
Algorithm TileBoard(tR, tC, dR, dC, size).
{
  if size=1 then return;
  tileToUse = tile++;
  qS = size/2;
  if dR<tR+qS and dC<tC+qS then
        TileBoard(tR, tC, dR, dC, qS);
  else
  {
    board[tR+qS-1][tC+qS-1] = tileToUse;
    TileBoard(tR, tC, tR+qS-1, tC+qS-1, qS);
  }
}//
```

Analysis of Defective chessboard: The recurrence relation for defective chessboard is given as

$$T(k) = \begin{cases} d & \text{if } k=0 \\ 4T(k-1)+c & \text{if } k>0. \end{cases}$$

where $T(k)$ denote the time taken to solve $2^k \times 2^k$ board.

$T(k) = 4T(k-1)+c \sim (1)$

From $\sim(1)$ $T(k-1) = 4T(k-2)+c \sim (2)$

$T(k-2) = 4T(k-3)+c \sim (3)$

Substitute $\sim(2)$ in $\sim(1)$

$\Longrightarrow T(k) = 4[4T(k-2)+c]+c = 4^2 T(k-2)+4c+c \sim (4)$

Substitute ~③ in ~④

$\Rightarrow T(K) = 4^2[4T(K-3)+C] + 4C + C$

$= 4^3T(K-3) + 4^2C + 4C + C.$

$\vdots$

$= 4^K T(K-K) + 4^{K-1}C + 4^{K-2}C + \cdots + C$

$= 4^K \times 0 + 4^{K-1} - 1. \quad \because 4^0 + 4^1 + \cdots 4^K = 4^{K+1} - 1.$

$= \frac{4^K}{4} - 1 \qquad$ Let $4^K = n.$

$= \frac{n}{4} - 1$

Apply Big-Oh Notation.

$\therefore T(n) = O(n)$

$$T(n) = 7^k\, T(n/n) + n^2\left[\left(\frac{7}{4}\right)^0 + \left(\frac{7}{4}\right)^1 + \cdots + \left(\frac{7}{4}\right)^{k-1}\right]$$

$$= 7^k\, T(1) + n^2 \cdot \left(\frac{7}{4}\right)^k$$

$$= 7^{\log_2 n} + n^2 \left(\frac{7}{4}\right)^{\log_2 n}$$

$$= n^{\log_2 7} + n^2 * n^{\log_2 (7/4)}$$

$$= n^{\log_2 7} + n^2\left[n^{\log_2 7 - \log_2 4}\right]$$

$$= n^{\log_2 7} + n^2\left[n^{\log_2 7 - 2}\right]$$

$$= n^{\log_2 7} + \cancel{n^2} * \frac{n^{\log_2 7}}{\cancel{n^2}}$$

$$= 2n^{\log_2 7} = 2n^{2.81}$$

$$\therefore O(T(n)) = O(n^{2.81}).$$

Frequently Asked Questions

1. What is Divide and Conquer strategy? Give its reccurence relation?
2. Write and explain the control abstraction for D and C?
3. Explain the general method of Divide and Conquer?
4. What is Binary Search? How it can be implemented by divide and conquer strategy? Explain with example?
5. Write the iterative algorithm for searching an element by using Binary Search.
6. Present an algorithm for Binary Search using one comparison per cycle.
7. Write and explain Recursive Binary Search algorithm with an example?

(8) Apply divide and Conquer strategy to the following input values for searching 112 and -14 by showing the values of low, mid, high for each search.

−15, −6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151.

(9) Explain the method for searching element 94 from the following set of elements. by using Binary Search.

{10, 12, 14, 16, 18, 20, 25, 30, 35, 38, 40, 45, 50, 55, 60, 70, 80, 90}.

(10) Search for an element −2 from the below set by using Binary Search.

A = {−15, −6, 0, 7, 9, 23, 54, 82, 101, 112}

Also draw Binary decision tree for the above.

(11) Give an example for Binary Search. Draw binary decision tree.

(12) Derive the time complexity of Binary Search?

(13) Draw Binary Decision tree for the following set

(3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 47).

(14) Explain Merge Sort with an example?

(15) Write an algorithm for Merge Sort using Divide & Conquer?

(16) Find the best, average and worst case complexity for Merge Sort?

(17) A sorting method is seaid to be stable if at end of the method, identical elements occur in the same order as in the original unsorted set. Is merge sort a stable sorting method? Prove it.

(18) Apply merge sort and show the file after each splitting and then merging for the following input:

50, 10, 25, 30, 15, 70, 35, 55.

(19) Draw the tree of calls of merge for the following set of elements

(20, 30, 10, 40, 5, 60, 90, 45, 35, 25, 15, 55).

(20) Draw the tree of calls of mergesort for the following set

(35, 25, 15, 10, 45, 75, 85, 65, 55, 5, 20, 18).

(21) Explain the way divide and conquer works for quick sort with example.

(22) Write an algorithm of Quick Sort and explain in detail.

(23) Find the best, average and worst case complexity for Quick sort.

(24) Show how procedure QUICKSORT sorts the following set of keys:

(1, 1, 1, 1, 1, 1, 1) and (5, 5, 8, 3, 4, 3, 2).

(25) Sort the following elements using Quick Sort.

3-3

(i) 5, 1, 7, 3, 4, 9, 8, 2, 6. ~~9~~, 24, 27, 30, 45, 53,

(ii) 20, 30, 80, 50, 40, 70, 60, 90, 10

(iii) 25, 30, 36, 49, 58, 67, 69, 10.

(iv) 25, 20, 16, 49, 28, 17, 9, 10.

(26) Discuss briefly about the randomized Quick Sort?

(27) Explain Strassen's Matrix Multiplication with example?

(28) Write an algorithm for SMM?

4. Greedy Method

* **General Method**: In Greedy Method, the problem is solved based on the information available. The Greedy Method is straight forward method. for obtaining optimal solution.

Optimal Solution: From a set of feasible solutions, a feasible solution that satisfies the objective function is called as Optimal Solution.

Objective Function: A function which is used to determine a better solution is called as Objective Function.

Feasible Solution: The subset of n inputs which satisfies some constraints are called as feasible solutions.

For example, What is the max even number b/w 1 to 50?

Here inputs are 1, 2, 3.... 50.

Constraint is even number

∴ Feasible Solutions are: 2, 4, 6, 8.... 50

Objective Function is max even number

Optimal Solution is 50.

In Greedy method,

1. For every input a solution is obtained.
2. Then Feasibility of solution is performed.
3. For each set of feasible solutions the solution which satisfy the given objective function is obtained.
4. Such solution is called optimal Solution.

* Algorithm for Greedy Method:

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

* Difference between Divide and Conquer and Greedy method.

Divide and Conquer:

1. Divide and Conquer is used to obtain a solution to given problem.
2. In this technique, the problem is divided into small subproblems. these subproblems are solved independently. Finally, all the solutions of subproblems are collected together to get the solution to the given problem.
3. In this method, duplications in subsolutions are neglected.
4. Divide and Conquer is less efficient.
5. Examples are Quick Sort, Binary Search..etc.

Greedy Method:

1. Greedy Method is used to obtain Optimal Solution.
2. In Greedy Method, a set of feasible is generated and optimum solution is obtained.

3. In Greedy Method, the optimal solution is obtained without revising previous solutions.
4. Greedy Method is compagitively efficient.
5. Examples are : 0/1 Knapsack Problem, Minimum Spanning Tree.

Applications of Greedy Method :

1. **0/1 Knapack Problem :** Suppose there are n objects from $i = 1, 2, 3 \cdots n$. Each object i has some positive weight w_i and profit P_i. Consider a Knapsack (or) Bag with capacity m. Place these objects into the knapsack such that weight of all objects in the knapsack must be less than or equal to capacity of the knapsack 'm'.

The objective is to obtain maximized $\sum_{1 \le i \le n} P_i x_i$ subject to $\sum_{1 \le i \le n} w_i x_i \le m$ and $0 \le x_i \le 1, 1 \le i \le n$. this can be solved based on the following three strategies.

1. Greedy about Profit
2. Greedy about Weight
3. Greedy about Profit per Unit Weight

① Greedy about Profit:

- (a) In this strategy, choose the objects having more profit
- (b) Total weight of the objects in the knapsack must be less than or equal to m.

② Greedy about Weight:

- (a) In this strategy, choose the objects having less weig
- (b) Total weight of the objects in the knapsack must be less than or equal to m.

③ Greedy about Profit per Unit Weight:

ⓐ In this strategy, choose the objects having maximum profit per unit weight.

ⓑ Total weight of the objects in the knapsack must be less than or equal to m.

Example: Find an optimal solution to the knapsack instance $n=3, m=20, (P_1, P_2, P_3) = (30, 21, 18), (\omega_1, \omega_2, \omega_3) = (18, 15, 10)$. using Greedy method.

Sol:

	x_1	x_2	x_3	$\sum_{1 \le i \le n} P_i x_i$
case 1 :	1	2/15	0	32.8
case 2 :	0	2/3	1	32
case 3 :	5/9	0	1	34.6.

Case 1 : Greedy about Profit : Choose the objects having more profit. Since first object has more profit, place it into the knapsack (i.e $x_1=1$). Since weight of the ~~knapsack~~ first object is 18, after placing first object into the bag remaining capacity of the bag is $m = 20-18 = 2$.

Now, identify the object having more profit among the remaining objects. Since 2nd object has more profit with weight 15. Since remaining capacity of the bag is 2, place fraction of second object. i.e $x_2 = 2/15$. After placing fraction of 2nd object into the bag remaining capacity of the bag is $m = 2 - 15 \times \frac{2}{15} = 0$.

Hence $x_i = 0$ for all remaining objects

$\therefore x_3 = 0$.

Now compute $\sum_{i=1}^{n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$

$$= 30 \times 1 + 21 \times \frac{2}{15} + 18 \times 0$$

$$= 32.8.$$

Case 2: Greedy about Weight: Choose the object having less weight. Since 3rd object has less weight, place it into the bag i.e ($x_3 = 1$). After placing 3rd object into the knapsack remaining capacity of the knapsack is

$m = 20 - 10 = 10.$

Now, identify the objects having less weight among the remaining objects. Since 2nd object has less weight ~~but~~ place it into the knapsack. Here, weight of the knapsack is 10 and weight of the object is 10. Hence place a fraction of 2nd object i.e $x_2 = 10/15 = 2/3$. After placing fraction of 2nd object into the knapsack, remaining capacity is $m = 10 - 15 \times \frac{10}{15} = 0.$

Hence $x_i = 0$ for all remaining objects i.e $x_1 = 0$.

Now Compute $\sum_{i=1}^{n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$

$$= 30 \times 0 + 21 \times \frac{2}{3} + 18 \times 1$$

$$= 0 + 14 + 18 = 32.$$

Case 3: Greedy about profit per Unit weight: Here, first compute

$$\frac{P_1}{w_1} = \frac{30}{18} = 1.66, \quad \frac{P_2}{w_2} = \frac{21}{15} = 1.4, \quad \frac{P_3}{w_3} = \frac{18}{10} = 1.8.$$

Choose the object having maximum profit per unit weight. Since 3rd object has maximum profit per unit weight, place it into the knapsack i.e $x_3 = 1$. After placing 1st object into the knapsack, remaining capacity of the knapsack is $m = 20 - 10 = 10$.

Now, identify the object having maximum profit per unit weight among the remaining objects. First object has maximum profit per unit weight. Since weight of the 1st object is greater than weight of the knapsack fraction of 1st object is placed into the knapsack i.e $x_1 = \frac{10}{18} = 5/9$. After placing fraction of 1st object into the knapsack, remaining capacity is $m = 10 - 18 \times \frac{10}{18} = 0$.

Hence $x_i = 0$ for all remaining objects

$\therefore x_2 = 0$.

Now, Compute $\sum_{i=1}^{n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$

$$= 30 \times \frac{5}{9} + 21 \times 0 + 18 \times 1$$

$$= 34.6$$

Among these three cases third case gives more profit which is 34.6. Hence optimal solution is

$$(x_1, x_2, x_3) = (5/9, 0, 1)$$

Algorithm for Greedy Knapsack:

```
Algorithm GreedyKnapsack(m,n)
// p[1:n] and w[1:n] contain the profits and weights respectively
// of the n objects. m is knapsack size and x[1:n] is
// solution vector
{
    for i:=1 to n do x[i] = 0.0;
    U:=m;
    for i:= 1 to n do
    {
        if (w[i] > U) then break;
        x[i]:=1.0; U:= U-w[i];
    }
    if (i ≤ n) then x[i] = U/w[i];
}//
```

② **Job Sequencing With Deadlines:** Consider that there are n jobs to be executed from $i = 1, 2, 3 \cdots n$. Each job i has some profit $P_i > 0$ and deadline $d_i \geq 0$. These profits are gained by corresponding jobs. For obtaining feasible solution the job must completed within their given deadlines. The following are the rules to obtain the feasible solution.

① Each job takes one unit of time.

② If job starts before or its deadline, profit is obtained, otherwise no profit

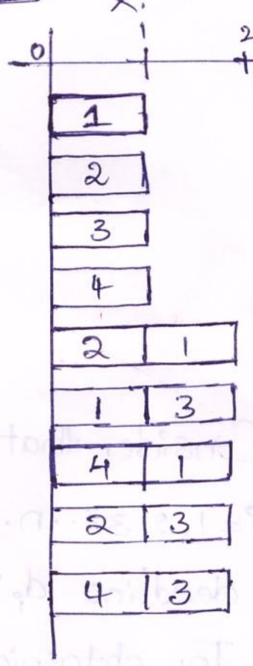
③ Goal is to schedule jobs to maximize total profit

④ Consider all possible schedules and compute the minimum total time in the system.

Example: Find optimal solution for the instance $n=4$, $(P_1, P_2, P_3, P_4) = (70, 12, 18, 35)$; $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$ using JS algorithm.

Sol: The feasible solutions are obtained by various permutations and combinations of jobs. Since max deadline is 2. Hence we can list out the following possibilities

1, 2, 3, 4, (1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (3, 4), (4, 1), (4, 2), (4, 3).



S.NO	feasible solution	processing sequence	profit
1	(1)	1	~~100~~ 70
2	(2)	2	12
3	(3)	3	18
4	(4)	4	35
5	(1,2), (2,1)	2,1	12+70=82.
6	(1,3), (3,1)	(1,3)	70+18 = 88
* 7	(1,4), (4,1)	4,1	35+70 = 105
8	(2,3), (3,2)	2,3	12+18 = 30
9.	(3,4), (4,3)	(4,3)	35+18 = 53.

Here, the feasible solution (2,4) or (4,2) is not allowed bcoz both have deadline 1. If Job 2 is started at 0 it will be completed on 1 but we cannot start job 4 at 1, since deadline of job 4 is 1.

Since the feasible solution (4,1) has more profit 105 it is the optimal solution.

Algorithm for Job Sequencing with Deadlines:

```
Algorithm JS (d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, the jobs are ordered such that
// p[1] ≥ p[2] ≥ ... p[n]. J[i] is the ith job in the optimal solution.
{
  d[0] := J[0] := 0; J[1] := 1; K := 1;
  for i := 2 to n do
  {
    r := K;
    while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r-1;
    if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
    {
        for q := K to (r+1) do
            J[q+1] = J[q];
        J[r+1] = i;
        K := K+1;
    }
  }
  return K;
}//.
```

Ex: What is the solution generated by the function JS when $n=7$, $P(1:7) = (3,5,20,18,1,6,30)$, $d(1:7) = (1,3,4,3,2,1,2)$

Sol: Initially $d[0]=0$, $J[0]=0$, $J[1]=1$

$K=1$

for $i=2$

$r = K = 1$

while $(d[J[1]] > d[2]$ and $d[J[1]] \neq 1)$

$1 > 3$ and $1 \neq 1$

F and T

F

if $(d[J[1]] \le d[2])$ and $(d[2] > 1)$ then.

$1 \le 3$ and $3 > 1$

T and T → T

$q := 1$ to 2 ∴ $q = 1, 2$. then

$J[2] = J[1]$

$J[3] = J[2]$

$J[2] = 2$

$\boxed{K = k+1 = 1+1 = 2.}$

for $i=3$ $r = K = 2$.

while $(d[J[2]] > d[3])$ and $d[J[2]] \ne 2$.

$3 > 4$ and $3 \ne 2$

F and T → F

if $d[J[2]] \le d[3]$ and $(d[3] > 2)$ then.

$3 \le 4$ and $4 > 2$

T and T → T

$q := 2$ to 3 i.e $q = 2, 3$. then

$J[3] = J[2] = 2$

$J[4] = J[3] = 2$

$J[3] = 3$

$\boxed{k = k+1 = 3}$

$i=4$ $r = K = \cancel{3}, 2$.

while $(d[J[3]] > d[4])$ and $(d[J[3]] \ne 3)$ do $r = 3-1 = 2$.

$4 > 3$ and $4 \ne 3$ T and T → T

while $(d[J[2]] > d[4])$ and $(d[J[2]] \ne 2)$

$3 > 3$ and $3 \ne 2$ F and T → F

if $(d[J[2]] \le d(4])$ and $(d[4] > 2)$ then

$3 \le 3$ and $3 > 2$ T and T → T

$q := 3$ to 3 then $q = 3$

$J[4] = J[3] = 3$

$J[3] = 4$ $\boxed{K = K+1 = 4}$

$J[0] = 0$

$J[1] = 1$ ✓

$J[2] = J[1] = 1$ ✗.

$J[3] = J[2] = 1$ ✗.

$J[2] = 2$. ✓

$J[3] = J[2] = 2$ ✗

$J[4] = J[3] = 2$ ✗.

$J[3] = 3$ ✗

$J[4] = J[3] = 3$ ✓

$J[3] = 4$ ✓

i=5 $\quad r = K = 4.$

while $(d[J[4]] > d[5])$ and $(d[J[4]] \neq 4)$

$4 > 2$ and $4 \neq 4$ T and F → F

if $(d[J[4]] \leq d[5])$ and $(d[5] > 4)$

$4 \leq 2.$ and $2 > 4$ F and F → F

i=6 $\quad r = K = 4.$

while $(d[J[4]] > d[6])$ and $(d[J[4]] \neq 4)$

$4 > 1$ and $4 \neq 4$ T and F → F

if $(d[J[4]] \leq d[6])$ and $(d[6] > 4)$

$4 \leq 1$ and $1 > 4$ F and F → F

i=7 $\quad r = K = 4.$

while $(d[J[4]] > d[7])$ and $(d[J[4]] \neq 4)$

$4 > 2$ and $4 \neq 4$ T and F → F

if $(d[J[4]] \leq d[7])$ and $(d[7] > 4)$

$4 \leq 2$ and $2 > 4$ F and F → F

We get $J[1] = 1, J[2] = 2, J[3] = 4, J[4] = 3$

Now arrange the jobs in descending ~~o~~ order based on their profits

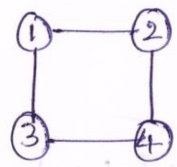
$$\therefore \quad \overset{1}{P_7} \quad \overset{2}{P_3} \quad \overset{3}{P_4} \quad \overset{4}{P_6} \quad \overset{5}{P_2} \quad \overset{6}{P_1} \quad \overset{7}{P_5}$$

$$(P_7 \; P_3 \; P_6 \; P_4.)$$

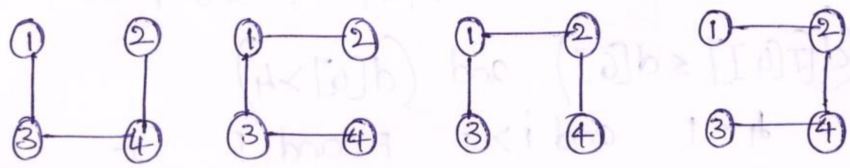
Hence optimal solution is $(6, 7, 4, 3)$ with profit 74.

③ <u>Minimum Cost Spanning Trees</u>: Let $G(V,E)$ be an undirected connected graph with V vertices and E edges. A subgraph $t=(V,e)$ of G is a spanning tree of G if and only if t is a tree.

<u>Ex</u>:



Spanning trees for the above graph are



<u>Applications of Spanning Trees</u>:

① Spanning trees are used to design efficient routing algorithms.

② Spanning trees are used to solve travelling sales person problem.

③ Spanning trees are used to design networks.

④ Spanning trees are used to find airline routes.

<u>Minimum Cost Spanning Trees</u>: Minimum Cost Spanning tree weighted connected graph G is a spanning tree with minimum weight. Minimum Cost Spanning Trees can be obt using the following algorithms.

1. Prim's Algorithm
2. Kruskal's Algorithm.

① Prim's Algorithm: A Greedy method is used to obtain a minimum cost spanning tree, builds this tree edge by edge. The next edge to include is choosen according to some optimization criteria. In prim's Algorithm.

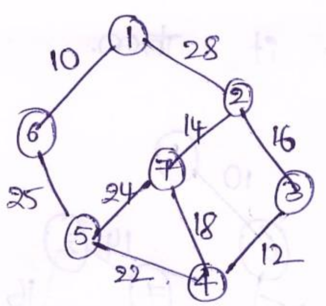
→ Initially choose an edge (u,v). containing minimum weight.

→ Now, among the list of vertices which are adjacent to u, v choose the edge having minimum weight and. make it as visited.

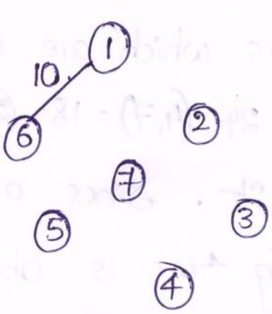
→ Now, among the list of vertices which are adjacent to all visited vertices choose the edge with min weight

→ Continue this process until all the vertices are visited.

Ex: Compute minimum cost spanning tree for the following graph.

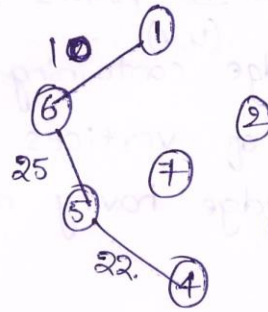


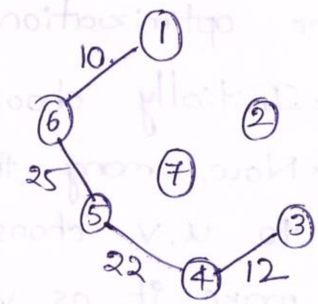
Sol: Since (1, 6) with weight 10 is the minimum cost edge in the given graph, make it as visited. Then.



Next, identify the list of vertices which are adjacent to 1, 6 ((1, 2), (5, 6)). choose (6, 5) with weight 25. then.

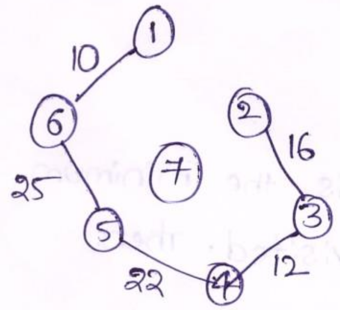
Now, identify the list of vertices which are adjacent to 1, 6, 5. [(1,2) = 28, (5,4) = 22, (5,7) = 24]. Since (5,4) has less weight choose it. then.

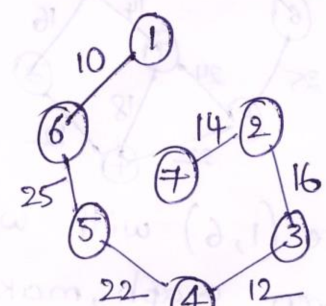




Identify the list of vertices which are adjacent to 1, 6, 5, 4. i.e (1,2) = 28, (5,7) = 24, (4,7) = 18, (4,3) = 12. Since (4,3) less weight choose it.

Identify the list of vertices which are adjacent to 1, 6, 5, 4, 3. i.e (1,2) = 28, (5,7) = 24, (4,7) = 18, (3,2) = 16. Since (3,2) has less weight choose it. then.



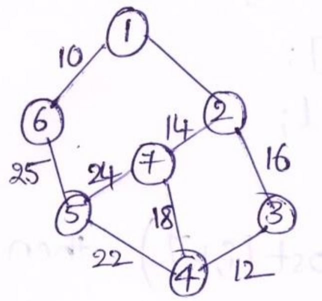


Identify the list of vertices which are adjacent to 1, 6, 5, 4, 3, 2. i.e (1,2) = 28, (5,7) = 24, (4,7) = 18, (2,7) = 14. Since (2,7) has less weight choose it. Since all vertices are visited minimum cost spanning tree is obtained with total cost = 10+25+22+12+16+14 = 99.

* Algorithm for Prim's :

```
Algorithm Prim (E, cost, n, t)
// E is set of edges in G. cost[] is the cost adjacency matrix
// of an n vertex graph.
{
    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
        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
    {
        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
            if (near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j])
                then near[k] := j;
    }
    return mincost;
}//
```

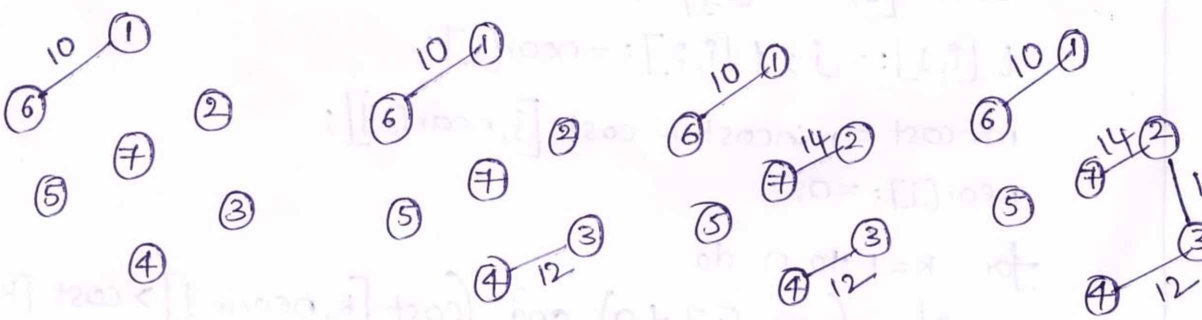
2. Kruskal's Algorithm: In this algorithm, arrange the list of edges in ascending order based on their weights. Choose the edges one by one which does not form any cycle. It is not necessary that selected edge is adjacent. For example, consider the following graph.



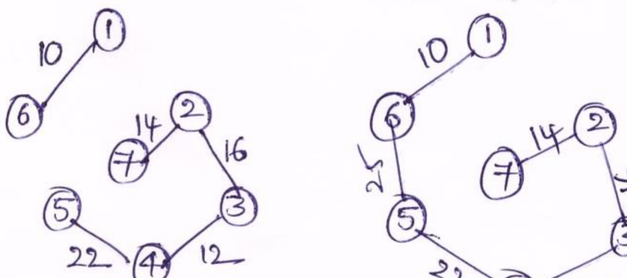
Now arrange the list of edges in ascending order based on their weights. Hence.

(1,6), (4,3), (7,2), (3,2), ~~(4,7)~~, (5,4), (5,7), (6,5).

Next, select the edges one by one until all the vertices are visited exactly once and ignore the edge which forms cycle. Hence.



Since edge (4,7) causes cycle ignore it



Total cost = 10+12+14+16+22+25
= 99.

* Algorithm for Kruskal's:

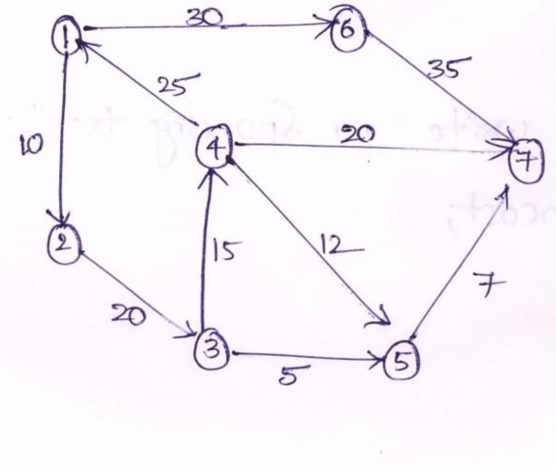
```
Algorithm Kruskals(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost [u,v] is
// the cost of edge (u,v). t is set of edges in MCST.
{
    Construct a heap out of the edge costs using Heapify;
    for i:=1 to n do parent [i]:=-1;
    i:=0; mincost:=0.0;
    while ((i<n-1) and (heap not empty)) do
    {
        delete a minimum cost edge (u,v) from the heap and
        reheapify;
        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;
```

(4) **Single Source Shortest Path Problem:** Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. A motorist wishing to drive from city A to city B would be interested in answers to the following questions:

- Is there a path from A to B ?
- If there is more than one path from A to B, which is the shortest path ?

Let G(V, E) be a directed or undirected graph. In Single Source shortest path, the shortest path from vertex V_0 to all the remaining vertex is determined. The vertex V_0 is called as source vertex and the last vertex is called as destination vertex. The length of the path is defined to be the sum of weights of the edges on that path.

For example, Consider a graph G given below.



Source vertex	Destination vertex	path	Distance
1	2	1→2	10.
1	3	1→2→3	30
1	4	1→2→3→4	45
1	5	1→2→3→5	35
1	6	1→6	30
1	7	1→2→3→5→7	42.

(*) Algorithm for Single Source Shortest Problem:

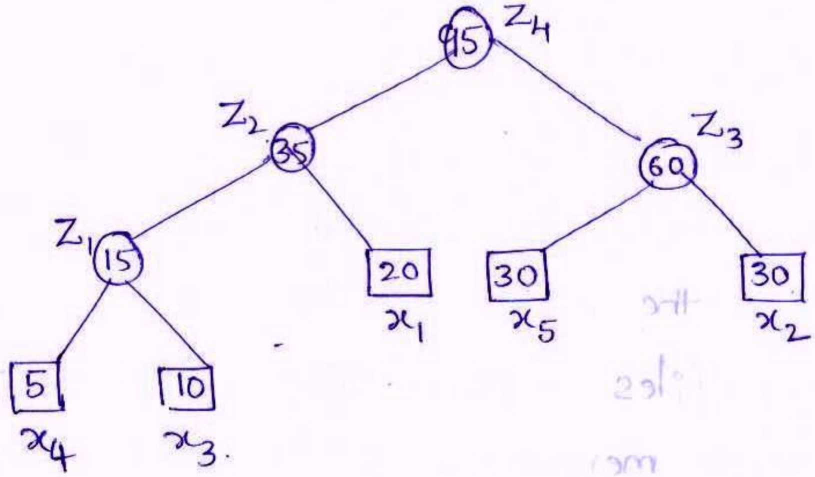
```
Algorithm ShortestPaths (v, dist, cost, n)
// dist[j] is set to the length of the shortest path from
// vertex v to vertex j in a digraph G with n vertices.
// dist[v] is set to 0.
{
   for i := 1 to n do
   {
      s[i] := false; dist[i] := cost [v,i];
   }
   s[v] := true; dist [v]: = 0.0;
   for num: = 2 to n do
   {
      choose u from among those vertices not in S
      such that dist [u] is minimum;
      s [u]: = true;
      for (each w adjacent to u with s[w] = false) do
         if (dist [w] > dist [u] + cost [u,w]) then
               dist [w] : = dist [u] + cost [u,w];
   }
}//.
```

Optimal Merge Patterns: Optimal Merge pattern is a pattern that relates to the merging of two or more sorted files in a single sorted file. this type of merging can be done by two-way merging method. If we have two sorted files containing m and n records then they could be merged together to obtain one sorted file in time $O(m+n)$.

When more than two sorted files are to be merged together the merge can be done by repeatedly merging sorted files in pairs. Thus, if files x_1, x_2, x_3 and x_4 are to be merged we could first merge x_1 and x_2 to get a file y_1. Then merge y_1 and x_3 to get y_2. Finally merge y_2 and x_4 to get desired sorted file. Alternatively we could first merge x_1 and x_2 getting y_1, then merge x_3 and x_4 getting y_2 and finally y_1 and y_2 getting the desired sorted file. Different pairings require different amounts of computing time.

For eample, x_1, x_2 and x_3 are three sorted files of length 30, 20 and 10 records each. Merging x_1 and x_2 need 50 record moves. Merging the result with x_3 need another 60 moves. the total number of record moves required to merge the three files this way is 110. Instead, if we first merge x_2 and x_3 and then x_1, the total record moves made is only 90.

A greedy attempt to obtain optimal merge patt[...] is, at each stage merge the two smallest files together. If we have five files $x_1, x_2 \cdot\cdot x_5$ with sizes 20, 30, 10, 5, [...] our greedy rule would generate the following merge path



Binary Merge Tree Representing a Merge Pattern

Here, first merge x_4 and x_3 to get z_1 ($|z_1|=15$); merge z_1 and x_1 to get z_2 ($|z_2|=35$); merge x_5 and x_2 to get z_3 ($|z_3| = 60$); Finally merge z_2 and z_3 to get Z_4 ($|z_4| = 95$). The total number of record moves is $15+35+60+95 = 205$.

In the above tree leaf nodes are drawn as squares and represent the given 5 files. These nodes are called as external nodes. The remaining nodes are drawn circular and are called internal nodes. Each internal node has exactly two children and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e no: of records of the file represented by that node.

The external node x_4 is at a distance of 3 from the root node Z_4. Hence, the records of file x_4 will be moved three times, once to get Z1, once again to get z_2 and finally one more time to get Z_4. If d_i is the distance from the root to the external node for file x_i and q_i is length of x_i then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^{n} d_i q_i$$

This sum is called the weighted external path lengt of the tree.

Algorithm:

```
Algorithm Tree (L,n)
//L is a list of n single node Binary tree .
{
   for i := 1 to n-1 do
   {
      GETNODE(T);
      LCHILD (T) <- LEAST (L);
      RCHILD (T) <- LEAST (L);
      WEIGHT (T) <- WEIGHT (LCHILD(T)) + WEIGHT (RCHILD(T))
      INSERT (L,T)
   }
   return LEAST(L)
}//.
```

The algorithm has a list of L trees as input. Each node in a tree has three fields, LCHILD, RCHILD and WEIGHT Initially, each tree in L has exactly one node. This node

is an external node and has LCHILD and RCHILD fields ze
while the WEIGHT is the length of one of the n files t
be merged. GETNODE(T) provides a new node for use in
building the tree. LEAST(L) finds a tree in L whose roo
has least WEIGHT. this tree is removed from L. INSER
(L,T) inserts the tree with root T into the list L.

<u>Analysis</u>: The main loop is executed $n-1$ times. If L
is kept in non decreasing order according to the
WEIGHT value in the roots, then the LEAST(L) requir
only $O(1)$ time and INSERT (L,T) can be done in
$O(n)$ time. Hence the total time taken is $O(n^2)$.

In case L is represented as a min heap
where the root values is $\leq$ the values of its childr
then LEAST(L) and INSERT (L,T) can be done in
$O(\log n)$ time. In this case the computime is $O(n\log$

Frequently Asked Questions

1. What is greedy method? Explain with example?
2. Give the control abstraction for Greedy Method?
3. What is a Knapsack Problem? Find an optimal solution to the knapsack instance $n=7, m=18$, $(P_1, P_2 \cdots P_7) = (15, 5, 6, 7, 10, 1)$ and $(w_1, w_2 \cdots w_7) = (7, 4, 8, 2, 1, 4, 1)$.
4. Write an algorithm for Greedy Knapsack?
5. Find an optimal solution to the knapsack instance $n=7$, $m=15$, $(P_1, P_2 \cdots P_7) = (18, 15, 5, 7, 16, 8, 13)$ and $(w_1, w_2 \cdots w_7) = (2, 3, 5, 7, 1, 4, 1)$.
6. Find an optimal solution to the knapsack instance $n=7, m=15$, $(P_1, P_2 \cdots P_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(w_1, w_2 \cdots w_7) = (2, 3, 5, 7, 1, 4, 1)$.
7. Find an optimal solution to the following knapsack problem: $n=3, m=20$, $(P_1, P_2, P_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.
8. Find an optimal solution to the following knapsack instance: $n=7, m=10$, $(P_1, P_2 \cdots P_7) = (7, 18, 6, 7, 5, 3, 4)$, $(w_1, w_2 \cdots w_7) = (1, 4, 3, 2, 3, 6, 7)$
9. Explain in detail about Job Sequencing with Deadlines problem
10. What is the solution generated by the function JS when $n=7$, $P[1:7] = (3, 5, 20, 18, 1, 6, 30)$ and $d[1:7] = (1, 3, 4, 3, 2, 1, 2)$.
11. Let $n=5$, $P[1:5] = (10, 3, 33, 11, 40)$ and $d[1:5] = (3, 1, 1, 2, 2)$. Find the optimal solution using greedy method.
12. Let $n=5$, $P[1:5] = (40, 33, 30, 14, 10)$ and $d[1:5] = (2, 1, 2, 3, 3)$. Find the optimal solution using greedy method.
13. Let $n=5$, $P[1:5] = (20, 13, 10, 4, 1)$ and $d[1:5] = (2, 1, 2, 3, 3)$. Find the optimal solution using greedy method.
14. Present a Greedy algorithm for Sequencing Unit time Jobs with deadlines and profits.

15. Explain, how to find minimum cost spanning trees by using prim's algorithm?
16. Define minimum cost Spanning trees. Explain with suitable examples.
17. Draw a simple, connected, weighted graph with 8 vertices and 16 edges, with each unique edge weight. Apply prim's algorithm to get minimum cost spanning tree. Show all the stages.
18. Explain, how to find minimum cost spanning trees using kruskal's algorithm?
19. Write primè algorithm?
20. Write Kruskal's algorithm?
21. Define the following terms:
    (i) feasible Solution (ii) Optimal Solotion (iii) Object function.
22. What is a single source shortest path problem? Explain with an example?
23. Give the greedy algorithm to generate Single Source Shortest paths?
24.

(*) <u>General Method</u>: Dynamic Programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of the sequence of decisions.

Dynamic Programming is used to find optimal solution to the given problem. In this method, solution to a problem is obtained by making sequence of decisions. In dynamic programming an optimal sequence of decisions is obtained by using principal of optimality.

<u>Principle of Optimality</u>: the principle of optimality states that " in an optimal sequence of decisions or choices, each subsequence must also be optimal."

(*) <u>Differences between Divide and Conquer and Dynamic Programming</u>:

<u>Divide and Conquer</u>:

1. The problem is divided into smaller subproblems. These subproblems are solved independantly. Finally, all the solutions of subproblems are collected together to get the solution to the given problem.

2. In this method duplications in subsolutions are ignored.

3. This method is less efficient.

4. This method uses top down approach of problem solving.

5. This method splits its input at specific points usually in the middle.

Dynamic Programming:

① In this method, many decision sequences are generated and all the overlapping subinstances are considered.

② In this method, duplications are not allowed.

③ This method is efficient than Divide and Conquer

④. In this method, bottom up approach is used.

⑤. Dynamic Programming splits its input at every possible split point. then it determines which split point is optimal.

⊛ Differences between Greedy method and Dynamic Programming

Greedy Method:

① Greedy method is used for obtaining optimum solution.

② In Greedy method, from a set of feasible solutions, a solution which satisfies the constraints is optimal solution.

③ In Greedy method, Optimal solution is obtained without revising previously generated solutions.

④ In Greedy method, there is no guarantee of getting optimal sol.

Dynamic Programming:

①. Dynamic programming is used for obtaining optimal solution.

② There is no special set of feasible solutions in this method.

③ Dynamic Programming considers all possible sequences in order to obtain the optimal solution.

④ It is guaranteed that the dynamic programming will generate optimal solution using principle of optimality.

Applications of Dynamic Programming:

1. **Matrix Chain Multiplication:** Let there are n matrices $A_1, A_2, A_3 \cdots A_n$ of dimensions $P_1 \times P_2, P_2 \times P_3, P_3 \times P_4 \cdots P_n \times P_{n+1}$ need to be multiplied, In what order should $A_1 A_2 A_3 \cdots A_n$ be multiplied so that it would take the minimum nuber of computations to derive the product.

For example, Cosider an example of best way of multiplying 3 matrices: Let A_1 of size 5×4, A_2 of size 4×6 and A_3 of size 6×2.

$$(A_1 A_2) A_3 \text{ takes } (5 \times 4 \times 6) + (5 \times 6 \times 2) = 180$$

$$A_1 (A_2 A_3) \text{ takes } (4 \times 6 \times 2) + (5 \times 4 \times 2) = 88.$$

those, $A_1(A_2A_3)$ is much cheaper to compute than $(A_1 A_2) A_3$. Here optimal cost is 88.

To solve this problem using Dynamic programming perform the following steps.

1. Let M_{ij} denote the cost of multiplying $A_i \cdots A_j$, where the cost is measured in the number of scalar multiplications. Here,

$M_{ij} = 0$ if $i = j$ and

M_{1n} is the optimal solution.

② The sequence of decisions can be build using the principle of optimality.

③. Apply following formala for computing each sequence

$$M_{ij} = \min_{i \le K \le j-1} \left\{ M_{iK} + M_{K+1j} + P_i P_{K+1} P_{j+1} \right\}$$

⊛ Find the minimum number of operations required for the following chain matrix multiplication using dynamic programming.

$$A(30,40) * B(40,5) * C(5,15) * D(15,6)$$

Sol: Given that $P_1 = 30,\ P_2 = 40,\ P_3 = 5,\ P_4 = 15,\ P_5 = 6$

Since 4 matrices are given.

M_{14} is the required solution.

$M_{11}=0$	M_{12}	M_{13}	M_{14}
$\cancel{M_{21}}$	$m_{22}=0$	m_{23}	m_{24}
$\cancel{m_{31}}$	$\cancel{m_{32}}$	$m_{33}=0$	m_{34}
$\cancel{m_{41}}$	$\cancel{m_{42}}$	$\cancel{m_{43}}$	$m_{44}=0$

Initially $m_{ij} = 0$ if $i = j$

$\therefore M_{11} = M_{22} = m_{33} = m_{44} = 0.$

Compute M_{12} for $i=1, j=2 \Rightarrow 1 \le K \le 1 \ \boxed{\therefore K=1}$

$$M_{12} = \min \left\{ M_{11} + m_{22} + P_1 P_2 P_3 \right\} = \min\{0+0+6000\} = 6000$$

for $i=1,\ j=3 \Rightarrow 1 \le K \le 2 \ \boxed{\therefore K = 1,2}$

$$M_{13} = \min \left\{ \begin{array}{l} M_{11} + m_{23} + P_1 P_2 P_4 \\ \underline{m_{12} + m_{33} + P_1 P_3 P_4} \end{array} \right\} = \min \left\{ \begin{array}{l} 0 + m_{23} + P_1 P_2 P_4 \\ 6000 + 0 + P_1 P_3 P_4 \end{array} \right\}$$

$$= \min \left\{ \begin{array}{l} 21000 \\ 8250 \end{array} \right\} = 8250$$

for $i=1, j=4 \Rightarrow 1 \le K \le 3 \quad \therefore K = 1,2,3.$

$$M_{14} = \min \begin{cases} M_{11} + M_{24} + P_1 P_2 P_5 \\ \underline{M_{12} + m_{34} + P_1 P_3 P_5} \\ M_{13} + m_{44} + P_1 P_4 P_5 \end{cases} = \min \begin{cases} 0 + \cancel{1850}1650 + 7200 \\ 6000 + 450 + 900 \\ 8250 + 0 + 2700 \end{cases}$$

$$= 7350.$$

for $i=2, j=3 \Rightarrow 2 \le K \le 2 \; \boxed{\therefore K=2}$

$$M_{23} = \min \{ M_{22} + m_{33} + P_2 P_3 P_4 \} = 0+0+3000 = 3000$$

for $i=2, j=4 \Rightarrow 2 \le K \le 3 \; \boxed{\therefore K = 2,3}$

$$M_{24} = \min \begin{cases} \underline{M_{22} + m_{34} + P_2 P_3 P_5} \\ M_{23} + m_{44} + P_2 P_4 P_5 \end{cases} = \min \begin{cases} 0 + 450 + 1200 \\ 3000 + 0 + 3600 \end{cases}$$

$$= 1650$$

for $i=3, j=4 \Rightarrow 3 \le K \le 3 \; \boxed{\therefore K = 3}$

$$M_{34} = \min \{ m_{33} + m_{44} + P_3 P_4 P_5 \} = 0+0+450 = 450$$

$\therefore \quad ABCD = 7350. \qquad 30 \times 40 \times 5 + 5 \times 15 \times 6 + 30 \times 5 \times 6$

$M_{12} + M_{34} + P_1 P_3 P_5 \qquad (AB)_{30\times5} + 5 \times 15 \times 6 + 30 \times 5 \times 6$

$M_{12} + P_3 P_4 P_5 + P_1 P_3 P_5 \qquad (AB)_{30\times5} + (CD)_{5\times6} + 30 \times 5 \times 6$

$P_1 P_2 P_3 + P_3 P_4 P_5 + P_1 P_3 P_5 \qquad [(AB)(CD)]_{30\times6}$

Algorithm for Matrix Chained Multiplication:

```
Algorithm MCM(P[1,2,3...n])
{
  for i:= 1 to n do M[i,i]:=0;
  for len:= 2 to n do
  {
    for i:=1 to (n-len+1) do
    {
      j:= i+len-1;
      M[i,j]:=∞;
      for k:= i to j-1 do
      {
        q:= M[i,k]+ M[k+1,j] + P[i]*P[k+1]*P[j+1];
        if (q < M[i,j]) then
        {
          M[i,j]:=q; S[i,j]:=k;
        }
      }
    }
  }
  return M[1,n];
}
Algorithm mul(i,j)
{
  if (i=j) then return A[i];
  else
  {
    k= S[i,j]; P:= Mul[i,k]; Q= Mul[k+1,j];
    return P*Q;
  }
```

(2) Optimal Binary Search Tree (OBST): Suppose we are searching for a word from a dictionary, and for every required word, searching in dictionary becomes time consuming process. To perform this lookup more efficiently build the binary search tree of common words as key elements, arrange frequently used words nearer to the root and less frequently used words away from the root this type of Binary Search Tree is called Optimal Binary Search Tree.

Let us assume that the given set of identifiers is $\{a_1, a_2, \cdots a_n\}$ with $a_1 < a_2 < \cdots < a_n$. Let $P(i)$ be the probability with which we search for a_i. Let $q(i)$ be the probability that the identifier x is searched for 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 unsucessful search. clearly,

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

For this given data, we have to construct OBST for $\{a_1, a_2 \cdots a_n\}$. The following notations are used to solve this problem using Dynamic Programming.

(1). Let $T_{ij} = OBST(a_{i+1}, \cdots a_j)$

C_{ij} denotes the cost (T_{ij})

W_{ij} is the weight of each T_{ij}

T_{0n} is the final tree obtained

During the computations the root values are computed and r_{ij} stores the root value of T_{ij}.

② The OBST can be build using the following formula.

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

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

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

(*) Using the function OBST compute $w(i,j)$, $r(i,j)$ and $c(i,j)$, $0 \le i \le j \le 4$ for the identifier set (a_1, a_2, a_3, a_4) = {do, if, int, while} with $p(1:4) = (3,3,1,1)$ and $q(0:4) = (2,3,1,1,1)$. Using the $r(i,j)$'s construct OBST.

__Sol:__ Given that

$p(1) = 3,\ p(2) = 3,\ p(3) = 1,\ p(4) = 1$

$q(0) = 2,\ q(1) = 3,\ q(2) = q(3) = q(4) = 1$

$C_{00} = 0$ $w_{00} = 2$ $r_{00} = 0$	$C_{01} = 8$ $w_{01} = 8$ $r_{01} = 1$	$C_{02} = 19$ $w_{02} = 12$ $r_{02} = 1$	$C_{03} = 25$ $w_{03} = 14$ $r_{03} = 2$	$C_{04} = 32$ $w_{04} = 16$ $r_{04} = 2$
$\cancel{C_{10}}$ $\cancel{w_{10}}$ $\cancel{r_{10}}$	$C_{11} = 0$ $w_{11} = 3$ $r_{11} = 0$	$C_{12} = 7$ $w_{12} = 7$ $r_{12} = 2$	$C_{13} = 12$ $w_{13} = 9$ $r_{13} = 2$	$C_{14} = 19$ $w_{14} = 11$ $r_{14} = 2$
$\cancel{C_{20}}$ $\cancel{w_{20}}$ $\cancel{r_{20}}$	$\cancel{C_{21}}$ $\cancel{w_{21}}$ $\cancel{r_{21}}$	$C_{22} = 0$ $w_{22} = 1$ $r_{22} = 0$	$C_{23} = 3$ $w_{23} = 3$ $r_{23} = 3$	$C_{24} = 8$ $w_{24} = 5$ $r_{24} = 3$
$\cancel{C_{30}}$ $\cancel{w_{30}}$ $\cancel{r_{30}}$	$\cancel{C_{31}}$ $\cancel{w_{31}}$ $\cancel{r_{31}}$	$\cancel{C_{32}}$ $\cancel{w_{32}}$ $\cancel{r_{32}}$	$C_{33} = 0$ $w_{33} = 1$ $r_{33} = 0$	$C_{34} = 3$ $w_{34} = 3$ $r_{34} = 4$
$\cancel{C_{40}}$ $\cancel{w_{40}}$	$\cancel{C_{41}}$ $\cancel{w_{41}}$	$\cancel{C_{42}}$ $\cancel{w_{42}}$	$\cancel{C_{43}}$ $\cancel{w_{43}}$	$C_{44} = 0$ $w_{44} = 1$

Initially if $i=j$

$C(i,j)=0,\ w(i,j)=q(i),\ r(i,j)=0$

$\therefore\ C_{00}=C_{11}=C_{22}=C_{33}=C_{44}=0.$

$r_{00}=r_{11}=r_{22}=r_{33}=r_{44}=0.$

$w_{00}=q(0)=2 \quad w_{11}=q(1)=3 \quad w_{22}=q(2)=1$

$w_{33}=q(3)=1 \quad w_{44}=q(4)=1.$

$w_{01}=P(1)+q(1)+w(0,0)=3+3+2=8$

$w_{12}=P(2)+q(2)+w(1,1)=3+1+3=7$

$w_{23}=P(3)+q(3)+w(2,2)=1+1+1=3$

$w_{34}=P(4)+q(4)+w(3,3)=1+1+1=3$

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

$w_{13}=P(3)+q(3)+w(1,2)=1+1+7=9$

$w_{24}=P(4)+q(4)+w(2,3)=1+1+3=5$

$w_{03}=P(3)+q(3)+w(0,2)=1+1+12=14$

$w_{14}=P(4)+q(4)+w(1,3)=1+1+9=11$

$w_{04}=P(4)+q(4)+w(0,3)=1+1+14=16.$

Compute for $i=0,\ j=1 \Rightarrow 0<k\le 01\ \therefore\ k=01$

$C_{01}=\min\{C_{00}+C_{11}+w(0,1)\}=0+0+8=8.$

$r_{01}=k=1.$

Compute C_{12} for $i=1, j=2 \Rightarrow 1<k\le 2\ \therefore\ k=2.$

$C_{12}=\min\{C_{11}+C_{22}+w(1,2)\}=0+0+7=7,\ r_{12}=2$

Compute C_{23} for $i=2,\ j=3 \Rightarrow 2<K\leq 3 \ \therefore K=3.$

$$C_{23} = \min\left\{C_{22}+C_{33}+w(2,3)\right\} = 0+0+3 = 3,\ r_{23}=3$$

Compute C_{34} for $i=3, j=4 \Rightarrow 3<K\leq 4 \ \therefore K=4,\ r_{34}=4$

$$C_{34} = \min\left\{C_{33}+C_{44}+w(3,4)\right\} = 0+0+3 = 3$$

Compute C_{02} for $i=0, j=2 \Rightarrow 0<K\leq 2 \ \therefore \boxed{K=1,2}$

$$C_{02} = \min\left\{\begin{array}{l} C_{00}+C_{12}+w(0,2) \\ C_{01}+C_{22}+w(0,2) \end{array}\right\} = \min\left\{\begin{array}{l} 0+7+12 \\ 8+0+12 \end{array}\right\} = 19$$

$\therefore r_{02} = 1.$

Compute C_{13} for $i=1,\ j=3 \Rightarrow 1<K\leq 3 \ \therefore K=2,3.$

$$C_{13} = \min\left\{\begin{array}{l} C_{11}+C_{23}+w(1,3) \\ C_{12}+C_{33}+w(1,3) \end{array}\right\} = \min\left\{\begin{array}{l} 0+3+9 \\ 7+0+9 \end{array}\right\} = 12.$$

$\therefore r_{13} = 2$

Compute C_{24} for $i=2,\ j=4 \Rightarrow 2<K\leq 4 \ \therefore K=3,4$

$$C_{24} = \min\left\{\begin{array}{l} C_{22}+C_{34}+w(2,4) \\ C_{23}+C_{44}+w(2,4) \end{array}\right\} = \min\left\{\begin{array}{l} 0+3+5 \\ 3+0+5 \end{array}\right\} = 8$$

$\therefore r_{24} = 3$ or 4.

Compute C_{03} for $i=0, j=3 \Rightarrow 0<K\leq 3 \therefore K=1,2,3$

$$C_{03} = \min\begin{Bmatrix} C_{00}+C_{13}+\omega(0,3) \\ C_{01}+C_{23}+\omega(0,3) \\ C_{02}+C_{33}+\omega(0,3) \end{Bmatrix} = \min\begin{Bmatrix} 0+12+14 \\ \underline{8+3+14} \\ 19+0+14 \end{Bmatrix} = 25$$

$\therefore r_{03} = 2.$

Compute C_{14} for $i=1, j=4 \Rightarrow 1<K\leq 4 \therefore K=2,3,4.$

$$C_{14} = \min\begin{Bmatrix} C_{11}+C_{24}+\omega(1,4) \\ C_{12}+C_{34}+\omega(1,4) \\ C_{13}+C_{44}+\omega(1,4) \end{Bmatrix} = \min\begin{Bmatrix} \underline{0+8+11} \\ 7+3+11 \\ 12+0+11 \end{Bmatrix} = 19$$

$\therefore r_{14} = 2.$

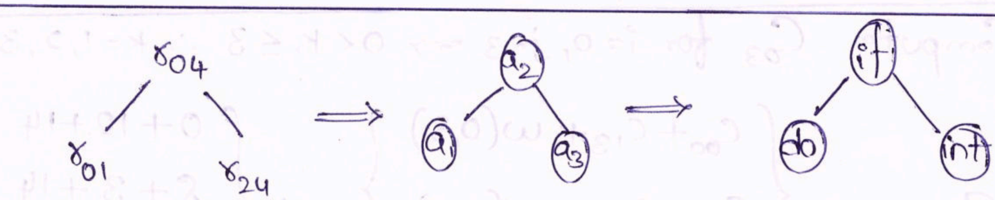
Compute C_{04} for $i=0, j=4 \Rightarrow 0<K\leq 4 \therefore K=1,2,3,4.$

$$C_{04} = \min\begin{Bmatrix} C_{00}+C_{14}+\omega(0,4) \\ C_{01}+C_{24}+\omega(0,4) \\ C_{02}+C_{34}+\omega(0,4) \\ C_{03}+C_{44}+\omega(0,4) \end{Bmatrix} = \min\begin{Bmatrix} 0+19+16 \\ \underline{8+8+16} \\ 19+3+16 \\ 25+0+16 \end{Bmatrix} = 32.$$

$\therefore r_{04} = 2.$

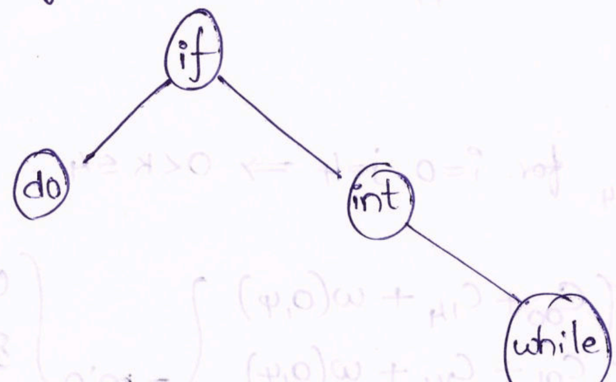
therefor T_{04} has a root r_{04}. The value of r_{04} is 2. From the given identifier set $\underline{a_2}$ becomes the root node. Now r_{ik-1} becomes left child and r_{kj} becomes the right child.

$\therefore r_{01}$ becomes the left child and r_{24} becomes right child of r_{04}. Here $r_{01}=1$, so a_1 becomes left child of a_2 and $r_{24}=3$ so a_3 becomes right child of a_2.



For the node r_{01}, $i=0, j=1, k=1$. does not have any children $\because r_{ik-1} = r_{00} = 0$, $r_{kj} = r_{11} = 0$.

For the node r_{24}, $i=2, j=4, k=3$. left child is $r_{ik-1} = r_{22} = 0$. that means left child of r_{24} is empty. the right child of r_{24} is $r_{kj} = r_{34} = 4$. Hence right child of a_3 is a_4. Hence, Optimal Binary Search tree is



⊛ Algorithm for OBST :

Algorithm OBST(p, q, n)

// Given n distinct identifiers $a_1 < a_2 < \cdots < a_n$ and probabilitie

// $p[i]$, $1 \le i \le n$, and $q[i]$, $0 \le i \le n$, this algorithm computes the

// cost $c[i,j]$ of OBST t_{ij} for identifiers. It also computes

// $r[i,j]$, the root of t_{ij}. $w[i,j]$ is the weight of t_{ij}.

```
{
    for i:=0 to n-1 do
    {
        w[i, i]:= q[i]; r[i,i]:=0; c[i,i]:=0;
```

```
    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
    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);
      c[i,j] := w[i,j] + c[i,k-1] + c[k,j];
      r[i,j] := k;
    }
  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;
}//.
```

(3) 0/1 Knapsack Problem: Consider n no: of objects for $i=1,2\dots n$ having their corresponding profits P_i and weights w_i. Consider a knapsack or bag with 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 objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, total weight of all chosen objects to be at most m. Formally, the problem can be states as

$$\text{maximize} \sum_{1 \le i \le n} P_i x_i \quad \text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and} \quad 0 \le x_i \le 1, \; 1 \le i \le n.$$

A solution to the knapsack can be obtained by making a sequence of decisions on the variables $x_1, x_2 \dots x_n$. To solve this problem using dynamic programming use the following notations.

(1). Compute $S^1, S^2 \dots S^i$. Initially $S^0 = \{(0,0)\}$

(2) Use the following formulae to compute S^i

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

$$\text{Where} \quad S_1^{i-1} = S^{i-1} + (P_i, w_i)$$

(3) After computing every S^i value apply purging rule (or) Dominance rule: If S^i contains (P_j, w_j) and (P_k, w_k) such that $P_j \le P_k$ and $w_j \ge w_k$ then eliminate the tuple (P_j, w_j) from S^i.

* Ex: Solve the following knapsack instance $m=6, n=3$, $(P_1, P_2, P_3) = (1,2,5)$ and $(w_1, w_2, w_3) = (2,3,4)$ using dynamic programming.

Sol: Given that $n=3, m=6$, $(P_1, w_1) = (1,2)$, $(P_2, w_2) = (2,3)$
$(P_3, w_3) = (5,4)$.

Compute S^1, S^2, S^3 $\boxed{\because n=3}$

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

$$S^i = S^{i-1} \cup S_1^{i-1} \text{ where } S_1^{i-1} = S^{i-1} + (P_i, w_i).$$

for $i=1$ $\quad S^1 = S^0 \cup S_1^0$

$$\therefore S_1^0 = S^0 + (P_1, w_1) = \{(0,0)\} + (1,2) = \{(1,2)\}$$

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

After applying purging rule S^1 is remains unchanged.

for $i=2$ $\quad S^2 = S^1 \cup S_1^1$

$$\therefore S_1^1 = S^1 + (P_2, w_2) = \{(0,0),(1,2)\} + (2,3)$$

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

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

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

After applying purging rule S^2 is unchanged.

for $i=3$ $\quad S^3 = S^2 \cup S_1^2$

$$\therefore S_1^2 = S^2 + (P_3, \omega_3)$$

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

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

$$\rightarrow S^3 = \{(0,0),(1,2),(2,3),(3,5)\} \cup \{(5,4),(6,6),(7,7),(8,9)\}$$

$$= \{(0,0),(1,2),(2,3),\underset{P_j,\omega_j}{\underline{(3,5)}},\underset{P_k,\omega_j}{(5,4)},(6,6),\underline{(7,7)},\underline{(8,9)}\}$$

By applying purging rule : $P_j \le P_k$ and $\omega_j \ge \omega_k$. the tuple $(3,5)$ is eliminated and since capacity of the knapsack is 6 eliminate the tuples having weight greater than 6. Hence

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

Since $m=6$, search for the tuple whose weight is 6 or closer. Now the selected tuple (P_i, ω_i) is $(6,6)$.

if $(P_i,\omega_i) \in S^i$ and $(P_i,\omega_i) \notin S^{i-1}$ then $x_i = 1$ else $x_i = 0$.

$(6,6) \in S^3$ and $(6,6) \notin S^2$ then $x_3 = 1$.

Now search for $(P_i - P_3, \omega_i - \omega_3) = (6-5, 6-4) = (1,2)$.

$(1,2) \in S^2$ and $(1,2) \notin S^1$ is false then $x_2 = 0$

$(1,2) \in S^1$ and $(1,2) \notin S^0$ is true then $x_1 = 1$

Now search for $(P_i - P_1, \omega_i - \omega_1) = (1-1, 2-2) = (0,0)$. Hence optimal solution is $(x_1, x_2, x_3) = (1,0,1)$ //

* Algorithm for Dynamic 0/1 Knapsack:

```
Algorithm DKnap(p, w, x, n, m)
{
  b[0] := 1; pair[1].p = pair[1].w = 0.0;
  t := h := 1;
  b[1] := next := 2;
  for i := 1 to n-1 do
  {
    k := t;
    u := Largest(pair, w, t, h, i, m);
    for j := t to u do
    {
      pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
      while ((k ≤ h) and (pair[k].w ≤ ww)) do
      {
        pair[next].p = pair[k].p;
        pair[next].w = pair[k].w;
        next := next+1; k := k+1;
      }
      if ((k ≤ h) and (pair[k].w = ww)) then
      {
        if pp < pair[k].p then pp := pair[k].p;
        k := k+1;
      }
      if pp > pair[next-1].p then
      {
        pair[next].p := pp; pair[next].w = ww;
        next := next+1;
      }
      while ((k ≤ h) and (pair[k].p ≤ pair[next-1].p)) do
          k := k+1;
  }
```

```
while (k≤h) do
{
    pair [next].p:= pair[k].p;
    pair [next].w:= pair[k].w;
    next:= next+1;
    k:= k+1;
}
t:=h+1; h:= next-1; b(i+1) := next;
}
TraceBack(p,w,pair,x,m,n);
}//
```

(4) **All Pairs Shortest Path Problem**: 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 $cost(i,j)$ is the length of edge $<i,j>$ and $cost(i,j)=\infty$, if there is no edge between i and j. Our aim is to find shortest available paths from every vertex to every other vertex.

The following notations are used to solve this problem using dynamic programming.

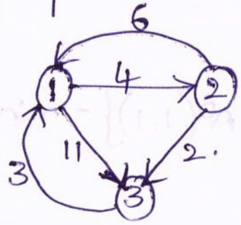
(1). Let $A^k(i,j)$ be the length of shortest path from node i to node j. We have to compute A^k for $k=1,2\ldots n$

(2). The following formulae is used to compute $A^k(i,j)$.

Initially $A^0(i,j) = cost(i,j)$.

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

(*) Ex Compute All Pairs shortest path problem for the following graph.



Sol: for the given graph cost adjacency matrix is

$$A^0(i,j) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$\because$ no: of nodes $n=3$ compute A^1, A^2, A^3.

for K=1

$$A^1(i,j) = \begin{bmatrix} A^1(1,1) & A^1(1,2) & A^1(1,3) \\ A^1(2,1) & A^1(2,2) & A^1(2,3) \\ A^1(3,1) & A^1(3,2) & A^1(3,3) \end{bmatrix}$$

$\therefore$ for $i=1, j=1$

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

$i=1, j=2$

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

$i=1, j=3$

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

$i=2, j=1$

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

$i=2, j=2$

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

$i=2, j=3$

$A^1(2,3) = \min\{A^0(2,3), A^0(2,1)+A^0(1,3)\} = \min\{2, 6+11\} = 2.$

$i=3, j=1$

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

$i=3, j=2.$

$A^1(3,2) = \min\{A^0(3,2), A^0(3,1)+A^0(1,2)\} = \min\{\infty, 3+4\} = 7$

$i=3, j=3$

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

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

Now for $\boxed{K=2}$ Compute $A^2 = \begin{bmatrix} A^2(1,1) & A^2(1,2) & A^2(1,3) \\ A^2(2,1) & A^2(2,2) & A^2(2,3) \\ A^2(3,1) & A^2(3,2) & A^2(3,3) \end{bmatrix}$

$i=1, j=1$

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

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

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

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

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

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

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

$A^2(3,2) = \min\{A^1(3,2), A^1(3,2)+A^1(2,2)\} = \min\{7, 7+0\} = 7$

$A^2(3,3) = \min\{A^1(3,3), A^1(3,2)+A^1(2,3)\} = \min\{0, 7+2\} = 0$ //.

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

for $\boxed{k=3}$ compute $A^3(i,j) = \begin{bmatrix} A^3(1,1) & A^3(1,2) & A^3(1,3) \\ A^3(2,1) & A^3(2,2) & A^3(2,3) \\ A^3(3,1) & A^3(3,2) & A^3(3,3) \end{bmatrix}$

$i=1, j=1$

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

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

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

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

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

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

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

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

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

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

* Algorithm for All Pairs Shortest Path Problem:

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 $1 \le i \le n$.

```
{
  for i:=1 to n do
     for j:=1 to n do
         A[i,j]: = cost [i,j];

  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]);

}//
```

(5) **The Travelling Salesperson Problem:** Let $G(V,E)$ be a directed graph with V vertices and E edges. The edges are given along with their cost C_{ij}. The cost $C_{ij}>0$ for all i and j. and $C_{ij}=\infty$ if there is no edge between i and j.

A Salesperson starts his tour at any vertex and should ends his tour at same vertex. During the tour he should visit all the vertices exactly once. Main objective of travelling salesperson problem is to find the tour of optimum cost. The following notations are used to solve the problem using dynamic programming.

(1). Let the function $g(1, V-\{1\})$ be total length of the tour terminating at vertex 1.

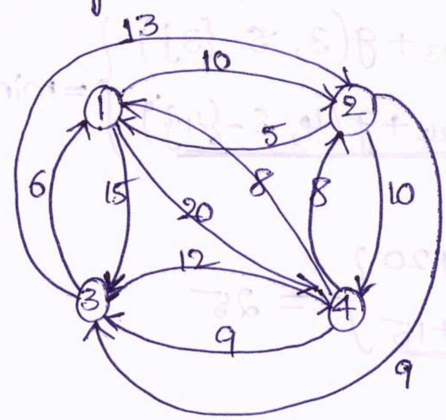
(2) Let $V_1, V_2 \ldots V_n$ be the sequence of vertices followed in optimal tour.

③ The following formula can be used to obtain tour of optimum cost.

$$g(i,S) = \min_{i \notin S,\, j \in S} \left\{ C_{ij} + g(j, S - \{j\}) \right\}$$

$$g(i,\phi) = C_{i1} : \text{Cost of the edge b/w } i \text{ and } 1$$

Ex: Find the shortest tour of a travelling sales person for the following instance using dynamic programming.



Sol: Initially construct cost adjacency matrix. i.e

	1	2	3	4
1	∞	10	15	20
2	5	∞	9	10
3	6	13	∞	12
4	8	8	9	∞

Let us assume vertex ① as source vertex. i.e find $g(1, V-\{1\})$ i.e $g(1, \{2,3,4\})$.

$$\underset{\substack{i,\ \ S \\ j = 2,3,4}}{g(1, \{2,3,4\})} = \min \begin{cases} C_{12} + g(2, S-\{2\}) \\ C_{13} + g(3, S-\{3\}) \\ C_{14} + g(4, S-\{4\}) \end{cases}$$

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

$$= \min \begin{cases} \underline{10 + 25} \\ 15 + 25 \\ 20 + 23 \end{cases} = 35$$

$$\underset{\substack{i,\ S \\ j=3,4}}{g(2, \{3,4\})} = \min \begin{cases} C_{23} + g(3, S-\{3\}) \\ \underline{C_{24} + g(4, S-\{4\})} \end{cases} = \min \begin{cases} C_{23} + g(3, \{4\}) \\ \underline{C_{24} + g(4, \{3\})} \end{cases}$$

$$= \min \begin{cases} 9 + 20 \\ \underline{10 + 15} \end{cases} = 25$$

$$\underset{i,\ S\ \therefore j=4}{g(3, \{4\})} = \min \{C_{34} + g(4, S-\{4\})\} = \min\{C_{34} + g(4, \phi)\}$$

$$= 12 + 8 = 20.$$

$$\cancel{= 9 + 6 = 15}$$

$$\underset{i,\ S}{g(4, \phi)} = C_{i1} = C_{41} = 8.$$

$$\underset{i,\ S\ \therefore j=3}{g(4, \{3\})} = \min \{C_{43} + g(3, S-\{3\})\} = \min \{\underline{C_{43} + g(3, \phi)}\}$$

$$= 9 + 6 = 15$$

$$\underset{i}{g(3, \phi)} = C_{i1} = \underline{C_{31}} = 6$$

$$g(3,\{2,4\}) = \min \begin{Bmatrix} C_{32} + g(2, S-\{2\}) \\ C_{34} + g(4, S-\{4\}) \end{Bmatrix} = \min \begin{Bmatrix} C_{32} + g(2,\{4\}) \\ C_{34} + g(4,\{2\}) \end{Bmatrix}$$

i, S
$\therefore j = 2, 4$

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

$$g(2,\{4\}) = \min \{C_{24} + g(4, S-\{4\})\} = \min \{C_{24} + g(4,\phi)\}$$

$i, S \;\therefore j=4$

$$= 10 + 8 = 18$$

$$g(4,\phi) = C_{i1} = C_{41} = 8.$$

i

$$g(4,\{2\}) = \min \{C_{42} + g(2, S-\{2\})\} = \min \{C_{42} + g(2,\phi)\}$$

$i, S \;\therefore j=2$

$$= 8 + 5 = 13.$$

$$g(2,\phi) = C_{i1} = C_{21} = 5.$$

$$g(4,\{2,3\}) = \min \begin{Bmatrix} C_{42} + g(2, S-\{2\}) \\ C_{43} + g(3, S-\{3\}) \end{Bmatrix} = \min \begin{Bmatrix} C_{42} + g(2,\{3\}) \\ C_{43} + g(3,\{2\}) \end{Bmatrix}$$

$i, S \;\therefore j = 2, 3$

$$= \min \begin{Bmatrix} 8 + 15 \\ 9 + 18 \end{Bmatrix} = 23.$$

$$g(2,\{3\}) = \min \{C_{23} + g(3, S-\{3\})\} = \min \{C_{23} + g(3,\phi)\}$$

$i, S \;\therefore j=3$

$$= 9 + 6 = 15$$

$$g(3,\phi) = C_{i1} = C_{31} = 6$$

$$g(3,\{2\}) = \min (C_{32} + g(2, S-\{2\})) = \min \{C_{32} + g(2,\phi)\}$$

$$= 13 + 5 = 18.$$

$g(2, \phi) = C_{i1} = C_{21} = 5.$

Hence length of the shortest tour is 35. and path is

$1 \to 2 \to 4 \to 3 \to 1.$

$$\underline{C_{12} + g(2, \{3,4\})}$$

$$\underline{C_{24} + g(4, \{3\})}$$

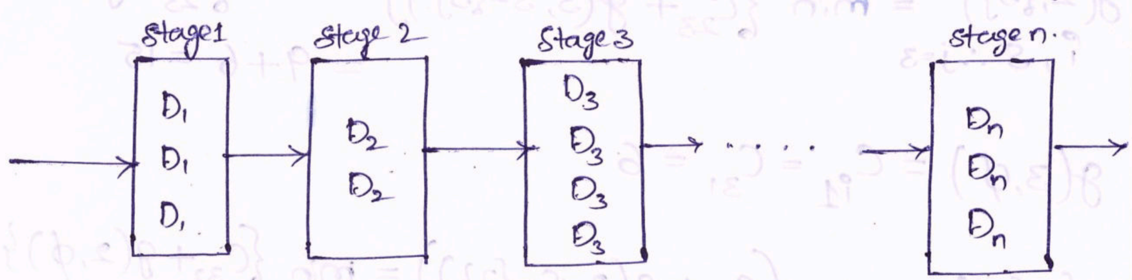
$$\underline{C_{43} + g(3, \phi)}$$

$$C_{31}.$$

1 – 2
2 – 4
4 – 3
3 – 1.

⑥ **Reliability Design**: The problem is to design a system that is composed of several devices connected in series.

Let r_i be the reliability of device D_i i.e, r_i is the probability that device i will function properly. Then, the reliability of the entire system is $\prod r_i$. 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 are connected in parallel at each stage.



If stage i contains m_i copies of device D_i, then the reliability of stage is $1-(1-r_i)^{m_i}$. Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$, then the reliability of the entire system is $\prod_{1 \le i \le n} \phi_i(m_i)$.

Let C_i be the cost of each unit of device i and C be the maximum allowable cost of the system. Here, find maximize $\prod_{1 \le i \le n} \phi_i(m_i)$ subject to $\sum_{1 \le i \le n} C_i m_i \le C$,

$m_i \ge 1$, $1 \le m_i \le u_i$, where $u_i = \left\lfloor \left(C + C_i - \sum_{j=1}^{n} C_j\right) / C_i \right\rfloor$

The dynamic programming solution S^i consist of tuples of the form (f, x) initially $(1,0)$. Compute S^i for $i = 1, 2, 3 \cdots n$.

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

Sol: Given that $C_1 = 30$, $C_2 = 15$, $C_3 = 20$, $C = 105$

$r_1 = 0.9$, $r_2 = 0.8$, $r_3 = 0.5$.

First compute u_1, u_2, u_3 using the following formula.

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

for $i=2$ $u_2 = (105+15-(30+15+20))/15 = 3$.

for $i=3$ $u_3 = (105+20-(30+15+20))/20 = 3$.

Now start computing the subsequences.

Initially $s^0 = \{(1,0)\}$.

Compute $s^1, s^2 \ldots s^i$ $\because i=3$, compute s^1, s^2, s^3

$S^1 = S_1^1 \cup S_2^1$ $\because u_1 = 2$.

$S^2 = S_1^2 \cup S_2^2 \cup S_3^2$ $\because u_2 = 3$

$S^3 = S_1^3 \cup S_2^3 \cup S_3^3$ $\because u_3 = 3$.

Now for S_1^1 $i=1, j=m_i=1$. then

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

$= 0.9$.

$C_i m_i = C_1 \times m_i = 30 \times 1 = 30$.

$\prod_{1 \le i \le n} \phi_i(m_i) = 1 \times 0.9 = 0.9$

$\sum_{1 \le i \le n} C_i m_i = 0+30 = 30$.

$\therefore S_1^1 = \{(0.9, 30)\}$.

For S_2^1 $i=1$; $m_i = j = 2$.

$\phi_1(m_i) = 1-(1-r_i)^{m_i} = 1-(1-r_1)^2 = 1-(1-0.9)^2$

$= 1-(0.1)^2 = 1-0.01 = 0.99$

$C_i m_i = C_1 m_i = 30 \times 2 = 60$.

Hence $S_2' = \{(0.99 \times 1, 60+0)\} = \{(0.99, 60)\}$

$$\therefore S' = \{(0.9, 30)\} \cup \{(0.99, 60)\}$$

$$\boxed{S' = \{(0.9, 30), (0.99, 60)\}}$$

After applying purging rule S' is unchanged.

For S_1^2 $i=2$, $m_i = j = 1$ then

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

$$C_i m_i = C_2 \times m_i = 15 \times 1 = 15$$

$$\therefore S_1^2 = \{(0.9 \times 0.8, 30+15), (0.99 \times 0.8, 60+15)\}$$
$$= \{(0.72, 45), (0.792, 75)\}$$

For S_2^2, $i=2$, $m_i = j = 2$ then

$$\phi_i(m_i) = 1-(1-r_i)^{m_i} = 1-(1-r_2)^2 = 1-(1-0.8)^2$$
$$= 1-(0.2)^2 = 1-0.04 = 0.96.$$

$$C_i m_i = C_2 \times m_i = 15 \times 2 = 30.$$

$$\therefore S_2^2 = \{(0.9 \times 0.96, 30+30), (0.99 \times 0.96, 60+30)\}$$
$$= \{(0.864, 60), (0.9504, 90)\}$$

For S_3^2, $i=2$, $m_i = j = 3$ then

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

$$= 1-(0.2)^3 = 1-0.008 = 0.992$$

$$C_i m_i = C_2 \times m_i = 15 \times 3 = 45.$$

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

$$= \{(0.8928, 75), (0.982, 105)\}$$

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

$$= \{(0.72, 45), (0.792, 75)\} \cup \{(0.864, 60), (0.9504, 90)\}$$
$$\cup \{(0.8928, 75), (0.982, 105)\};$$

$$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75), (0.9504, 90), (0.982, 105)\}$$

After applying purging rule the tuple $(0.792, 75)$ is eliminated from S^2. Hence

$$\boxed{S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75), (0.9504, 90), (0.982, 105)\}}$$

For S_1^3 $i=3, m_i=j=1$ then

$$\phi_i(m_i) = 1-(1-r_i)^{m_i} = 1-(1-r_3)^1 = 1-(1-0.5) = 0.5$$

$$C_i m_i = C_3 * m_i = 20 \times 1 = 20.$$

$\therefore S_1^3 = \{ (0.72\times0.5, 45+20), (0.864 * 0.5, 60+20), (0.8928\times0.5, 75+20), (0.9504\times0.5, 90+20), (0.982\times0.5, 105+20) \}$

$S_1^3 = \{ (0.36, 65), (0.432, 80), (0.4464, 95), (0.4752, 110), (0.491, 125) \}$

for S_2^3 $i=3$, $m_i = j = 2$ then

$\phi_i(m_i) = 1-(1-r_3)^{m_i} = 1-(1-0.5)^2 = 1-(0.5)^2$

$= 1-0.25 = 0.75$

$C_i m_i = 20\times2 = 40.$

$\therefore S_2^3 = \{ (0.72\times0.75, 45+40), (0.864\times0.75, 60+40), (0.8928\times0.75, 75+40), (0.9504\times0.75, 90+40), (0.982\times0.75, 105+40) \}$

$= \{ (0.54, 85), (0.648, 100), (0.6696, 115), (0.7128, 130), (0.7365, 145) \}.$

for S_3^3 $i=3$, $m_i = j = 3$. then

$\phi_i(m_i) = 1-(1-r_i)^{m_i} = 1-(1-r_3)^3 = 1-(1-0.5)^3$

$= 1-(0.5)^3 = 1-0.125$

$= 0.875$

$C_1 m_1 = C_3 m_i = 20 \times 3 = 60.$

$\therefore S_3^3 = \{(0.72 \times 0.875, 60+45), (0.864 \times 0.875, 60+60),$
$(0.8928 \times 0.875, 75+60), (0.9504 \times 0.875, 90+60),$
$(0.982 \times 0.875, 105+60)\}$

$= \{(0.65, 105), (0.756, 120), (0.7767, 135), (0.8316, 150),$
$(0.8592, 165)\}$

$\therefore S^3 = \{ (0.36, 65), (0.432, 80), (0.4464, 95), \cancel{(0.4752, 110)},$
$\cancel{(0.491, 125)}, (0.54, 85), (0.648, 100), (0.65, 105),$
$\cancel{(0.6696, 115)}, \cancel{(0.7128, 130)}, \cancel{(0.7365, 145)}, \cancel{(0.756, 120)},$
$\cancel{(0.7767, 135)}, \cancel{(0.8316, 150)}, \cancel{(0.8592, 165)}\}.$

$\because$ Capacity of the entire system is 105. delete the tuples having cost > 105.

$\therefore S^3 = \{(0.36, 65), (0.432, 80), \underset{P_j, \omega_j}{(0.4464, 95)}^{\times}, \underset{P_k, \omega_k}{(0.54, 85)}, (0.648, 100),$
$(0.65, 105)\}$

$\boxed{\therefore S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100), (0.65, 105)\}}$

Now choose the tuple containing cost approximately equal to 105. i.e $(0.65, 105)$.

Now tuple $(0.65, 105)$ is obtained from S_3^3 where $i=3, j=3$.

At that time $m_i = j$ i.e $\boxed{m_3 = 3}$

$\therefore (0.65, 105)$ is obtained from $(0.72, 45)$.

Now tuple $(0.72, 45)$ is obtained from S_1^2 where $i=2, j=1$

At that time $m_i = j$ i.e $\boxed{m_2 = 1}$

$\therefore (0.72, 45)$ is obtained from $(0.9, 30)$.

Now tuple $(0.9, 30)$ is obtained from S_1^1 where $i=1, j=1$.

At that time $m_i = j$. i.e $\boxed{m_1 = 1}$

Then we get $m_1 = 1, m_2 = 1, m_3 = 3$ as a solution to reliability design.

Frequently Asked Questions

1. Differentiate Divide and Conquer and Dynamic Programming
2. Differentiate Dynamic programming and Greedy method?
3. Explain the General Concept of Dynamic Programming?
4. Explain how to solve Matrix chain multiplication using DP?
5. Find the minimum number of operations required for the following chain matrix multiplication using Dynamic Programming?

   $A(20,30) * B(30,10) * C(10,5) * D(5,15)$

6. Find the minimum number of operations required for the following chain matrix multiplication using DP.

   $A(30,40) * B(40,5) * C(5,15) * D(15,6)$.

(7) Device an algorithm to find the optimal order of multiplying n matrices using dynamic programming technique.

(8) Explain in detail about OBST problem?

(9) Use the function OBST to compute $w(i,j)$, $r(i,j)$ and $c(i,j)$, $0 \le i \le j \le 4$, for the identifier set $(a_1, a_2, a_3, a_4) = $ (do, if, int, while) with $p(1:4) = (3,3,1,1)$ and $q(0:4) = (2,3,1,1,1)$. Using the $r(i,j)$'s construct the OBST.

(10) Using algorithm OBST compute $w(i,j)$, $R(i,j)$ and $c(i,j)$, $0 \le i \le j \le 4$ for the identifier set $(a_1, a_2, a_3, a_4) = $ (end, goto, print, stop) with $P(1) = 1/20$, $P(2) = 1/5$, $P(3) = 1/10$, $P(4) = 1/20$, $Q(0) = 1/5$, $Q(1) = 1/10$, $Q(2) = 1/5$, $Q(3) = 1/20$; $Q(4) = 1/20$. Using the $r(i,j)$'s construct OBST.

(11) Construct the OBST with the identifier set $(a_1, a_2, a_3, a_4) = $ (end, goto, print, stop) with $P(1:4) = (0.5, 0.1, 0.02, 0.5)$, $Q(0:4) = (0.5, 0.15, 0.2, 0.1, 0.2)$. Also compute the cost of the tree.

(12) Write an algorithm for OBST?

(13) Explain how to solve 0/1 Knapsack using Dynamic Program

(14) Find the solution ~~to~~ for the knapsack problem when $n = 3$, $m = 20$, $(P_1, P_2, P_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$ using DP.

(15) Solve the following 0/1 knapsack problem using Dynamic programming $m = 8, n = 3$, $(w_1, w_2, w_3) = (3, 6, 6)$, $(P_1, P_2, P_3) = (2, 3, 4)$.

(16) What is 0/1 Knapsack Problem? Define merging and purging rules of 0/1 Knapsack problem?

(17) Solve the following 0/1 knapsack problem using dynamic programming: $n = 4$, $m = 40$, $P[1:4] = (11, 21, 31, 33)$, $w[1:4] = (2, 11, 22, 15)$.

(18) Solve the following 0/1 Knapsack problem using dynamic programming? $n=3$, $m=6$, $(P_1, P_2, P_3) = (1, 2, 4)$, $(\omega_1, \omega_2, \omega_3) = (2, 3, 3)$.

(19) Solve the following 0/1 knapsack using Dynamic programming $n=6$, $m=165$, $P[1:6] = \omega[1:6] = (100, 50, 20, 10, 7, 3)$.

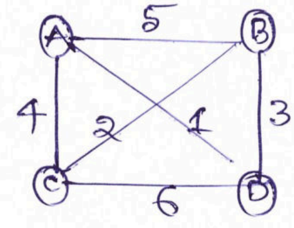
(20) Solve the following 0/1 Knapsack using Dynamic Programming $n=4$, $m=30$, $\omega[1:4] = (10, 15, 6, 9)$ and $P[1:4] = (2, 5, 8, 1)$.

(21) Generate the sets, $0 \le i \le 6$, when $\omega[1:4] = (10, 15, 6, 9)$, $P[1:4] = (2, 5, 8, 1)$.

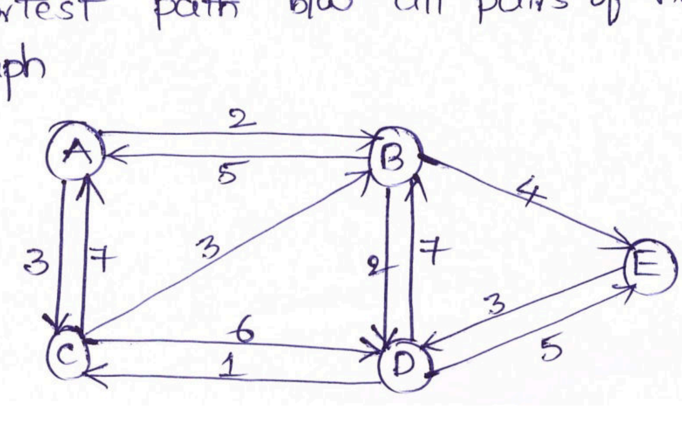
(22) Given $n=3$, weights & profits as $(\omega_1, \omega_2, \omega_3) = (2, 3, 4)$; $(P_1, P_2, P_3) = (1, 2, 5)$ & Knapsack capacity $m=6$ compute the set S^i containing the pair (P_i, ω_i).

(23) Explain the procedure for solving All Pairs shortest path problem using Dynamic programming?

(24) Find the shortest paths between all pairs of nodes in the following graph.



(25) Find the shortest path b/w all pairs of nodes in the following graph



(26) What is Travelling Sales Person Problem? How can it be solved using Dynamic Programming approach?

(27) Given an example of Travelling Sales Person Problem?

(28) Discuss the Dynamic Programming Solution for the problems of Reliability Design?

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

6. Backtracking

* **General Method:** Backtracking represents one of the most general searching technique. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using backtracking.

In many applications of the backtrack method, the desired solution is expressed as an n-tuple $(x_1, x_2 \cdots x_n)$, where the x_i are chosen from some finite set S_i. the solution maximizes or minimizes or satisfies a criterion function $p(x_1, x_2 \cdots x_n)$ is the required solution

the basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success. If the partial vector generated does not lead to an optimal solution, it can be ignored.

Backtracking algorithm determines the solution by systematically searching the solution space tree for the given problem. Backtracking is a depth first search with some bounding function. All solutions using back-tracking are required to satisfy a complex set of constraints. the constraints may be explicit or implicit.

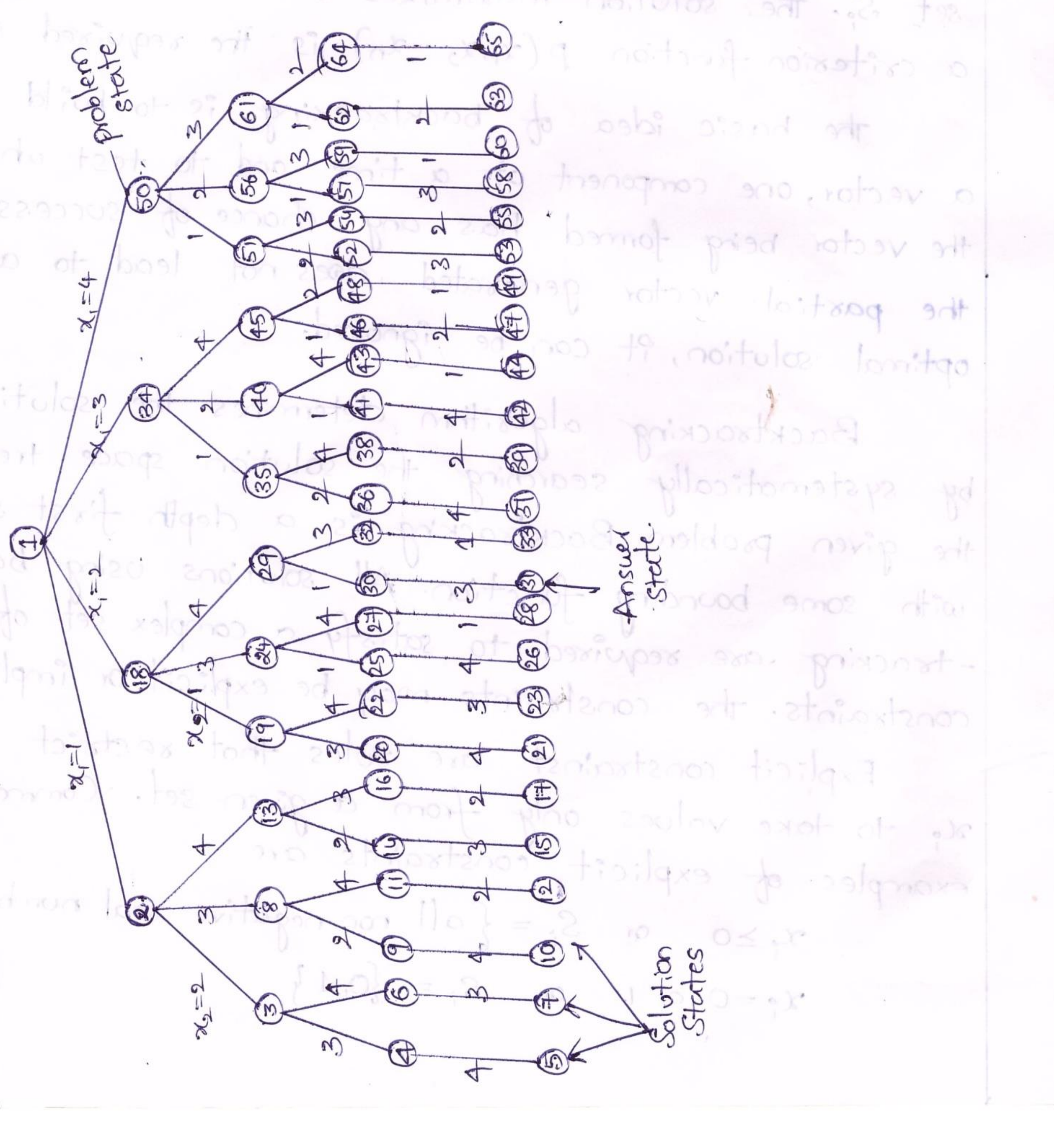
Explicit constrainst are rules that restrict each x_i to take values only from a given set. Common examples of explicit constraints are

$x_i \geq 0$ or $S_i = \{$all non negative real numbers$\}$

$x_i = 0$ or 1 or $S_i = \{0, 1\}$

The implicit constraints are rules that determine which of the tuples in the solution space tree satisfy the criterion function.

For example, Consider a 4-Queen's problem. It could be stated as 'there are 4-Queen's to be placed on 4×4 chessboard such that no two Queens can attack each other. Solution space tree for this problem is drawn as below.



All paths from root to other nodes define the state space of the problem.

Each node in the state space tree is called problem state.

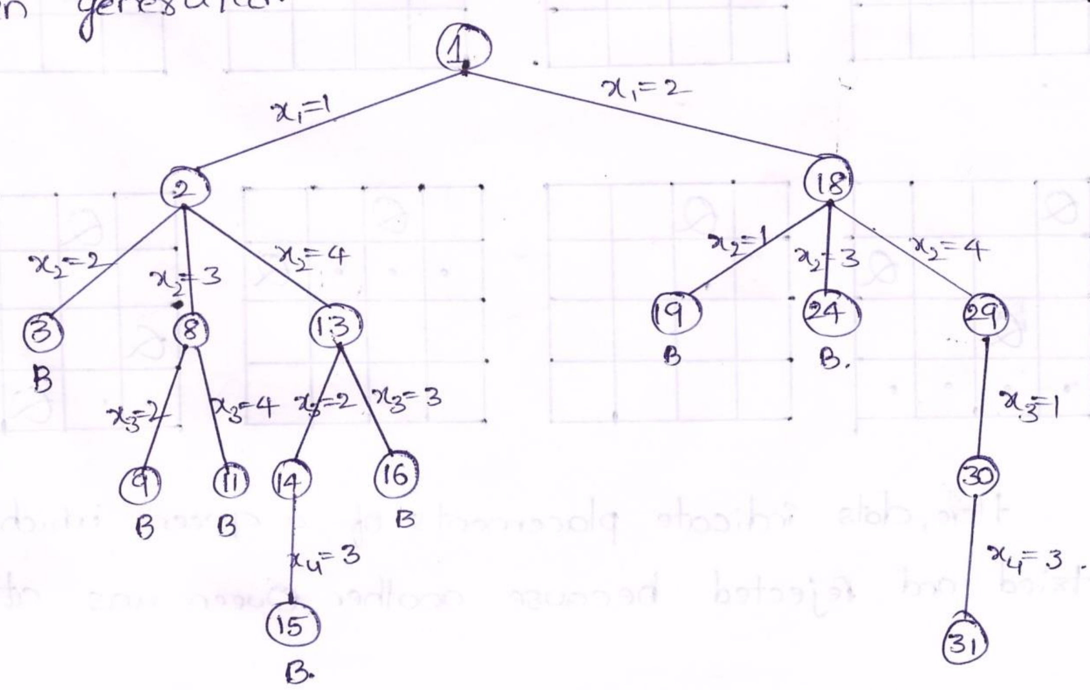
The solution states are the problem states S for which the path from root to S defines a tuple in the solution space.

The leaf nodes which correspond to an element in the set of solutions which satisfy the implicit constraints are called answer states.

A node which is generated and whose children have not yet been generated is called Live node.

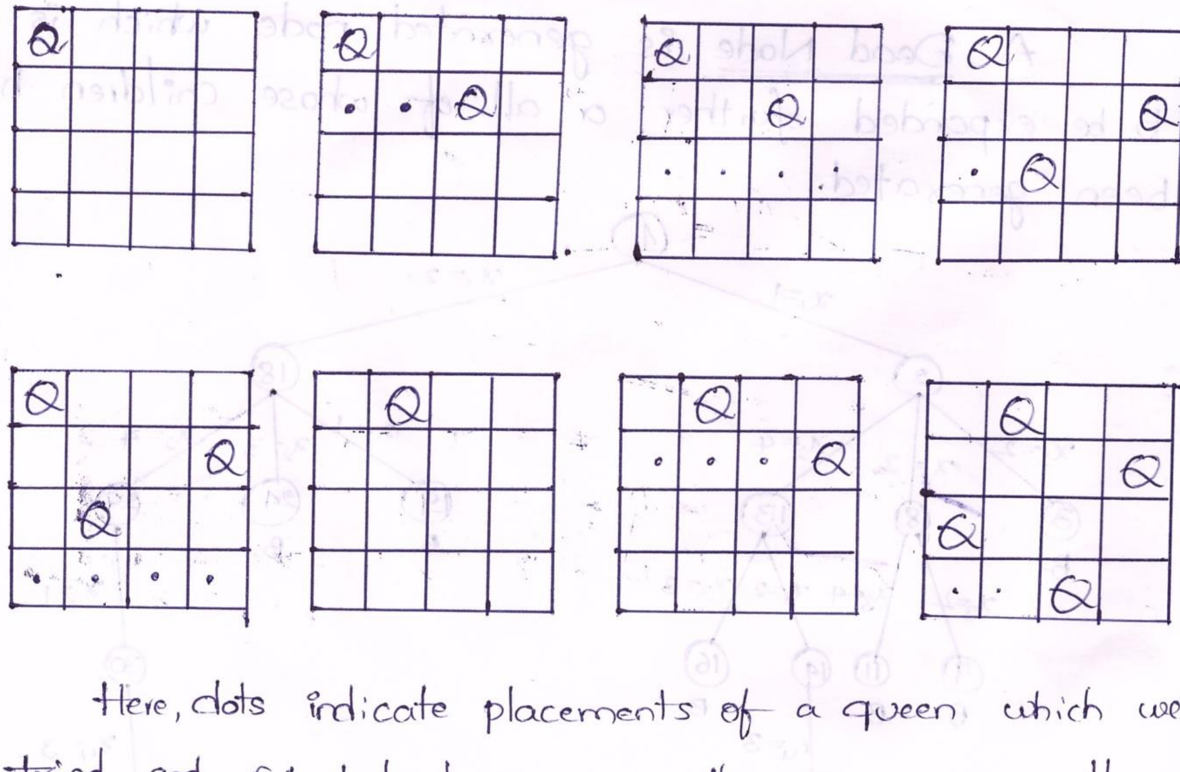
The Live Node whose children are currently being expanded is called E-node.

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



The above tree represents portion of the state space that is generated during the backtracking.

Initially start with the root node as the only live node. This becomes the E-node and its one of the children ② is generated. Node 2 becomes E-node. Node 3 is generated and immediately killed by applying bounding function. The next node generated is 8 and the path becomes (1,3). Node 8 becomes the E-node. However, it gets killed as its children cannot lead to an answer node. Now backtrack to node 2 and generate another child, node 13. The path is now (1,4) and so on. The following diagrams shows the board configurations as backtracking proceeds.



Here, dots indicate placements of a queen which were tried and rejected because another Queen was attacking.

* Recursive Backtracking Algorithm:

```
Algorithm Backtrack(k)
{
  for (each x[k] ∈ T(x[1]... x[k-1])) do
  {
    if (B_k(x[1], x[2]... x[k]) ≠ 0) then
    {
      if (x[1], x[2]... x[k] is a path to answer node)
          then write (x[1:k]);
      if (k<n) then Backtrack(k+1);
    }
  }
}
```

Iterative Backtracking Algorithm:

```
Algorithm IBacktrack(n)
{
   k:= 1;
   while (k≠0) do
   {
     if (there remains an untried x[k] ∈ T(x[1]... x[k-1])
         and B_k(x[1],... x[k]) is true) then
     {
        if (x[1],... x[k] is a path to answer node)
            then write (x[1:k]);
        k: = k+1;
     }
     else   k: = k-1;
   }
}//
```

(*) Applications of Backtracking:

(1) The n-Queen's Problem: [8-Queen's Problem]

Consider a $n \times n$ chessboard on which we have to place n queens so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. Let $(x_1, x_2 \cdots x_n)$ represent a solution in which x_i is the column of the ith row where ith queen is placed. The x_i's will all be distinct since no two ~~columns~~ queens can be placed in the same column.

Imagine that the chessboard squares being numbered as the indices of two-dimensional array a[1:n, 1:n], then observe that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value. For example, consider the queen at a[4,2]. The squares that are diagnal to this queen are a[3,1], a[5,3], a[6,4], a[7,5] and a[8,6]. All these squares have a row-column value of 2. Also, every element on the same diagonal that goes from the upper right to the lower left has the same row+column value. Suppose two queens are placed at positions (i,j) and (k,l). Then they are on the same diagonal only if

$$i-j = k-l \quad \text{or} \quad i+j = k+l \Rightarrow i-k = l-j$$

$$\Rightarrow i-k = j-l$$

$\therefore$ Two queens lie on the same diagonal iff

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

Algorithm for N-Queen's Problem:

```
Algorithm NQueens(k,n)
{
  for i:=1 to n do
  {
    if Place(k,i) then
    {
      x[k]:=i;
      if (k=n) then write (x[1:n]);
      else NQueens(k+1,n);
    }
  }
}
Algorithm Place(k,i)
// Returns true if a queen can be placed in kth row
// and ith column. Otherwise it returns false.
// Abs(r) returns the absolute value of r.
{
  for j:=1 to k-1 do
    if (x[j]:=i) or (Abs(x[j]-i) = Abs(j-k))
        then return false;
  return true;
}
```

(2) **Sum of Subsets Problem:** Let $S = \{S_1, S_2 \cdots S_n\}$ be n distinct positive numbers with $S_1 \le S_2 \le \cdots \le S_n$. Then we have to find all combinations of these numbers whose sums are m. This is called the sum of subsets problem. In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i (S_i) is included or not. The children of any node are easily generated. For a node at level i the left child corresponds to $x_i=1$ and the right to $x_i=0$.

Let S be a set of elements and m is the expected sum of subsets. Then.

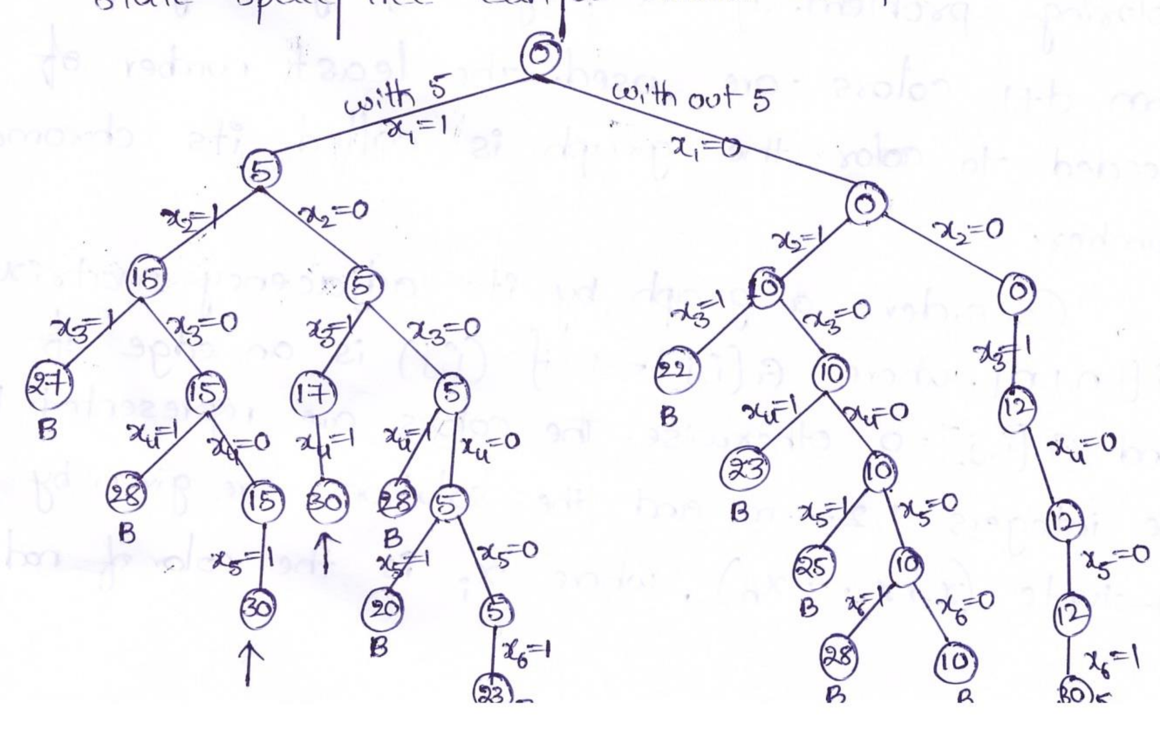
1. Start with an empty set.
2. Add next element from the list to the subset.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if end of the set is reached then backtrack through the subset until finding the optimal solution.
5. If the subset is feasible then repeat step 2.
6. If all the elements are visited without finding a solution and if no backtracking is possible then stop without solution.

* **Ex:** Let $w = \{5, 10, 12, 13, 15, 18\}$ and $m = 30$. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

Sol:

Initially Subset = { }	Sum = 0	
5	5	Add next element
5,10	15 ∵ 15<30	Add next element
5,10,12	27 ∵ 27<30	Add next element
5,10,12,13	40	Sum exceeds 30, hence backtrack.
5,10,12,15	42	Sum exceeds 30, hence backtrack
5,10,12,18	45	Sum exceeds 30, hence backtrack
5,10,13	28	∵ 28<30, add next element
5,10,13,15	43	Sum exceeds 30, hence backtrack.
5,10,13,18	46	Sum exceeds 30, hence backtrack.
5,10,15	30	∵ sum = 30, Solution is obtained.

State space tree can be drawn as follows.



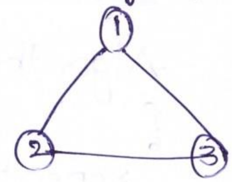
Algorithm for Sum of Subsets:

```
Algorithm SumOfSub(s, k, r)
{
    x[k] := 1;
    if (s+w[k]=m) then write (x[1:k]);
    else if (s+ w[k]+w[k+1] ≤ m)
        then SumOfSub(s+w[k], k+1, r-w[k]);
    if ((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
    {
        x[k] := 0;
        SumOfSub(s, k+1, r-w[k]);
    }
}//
```

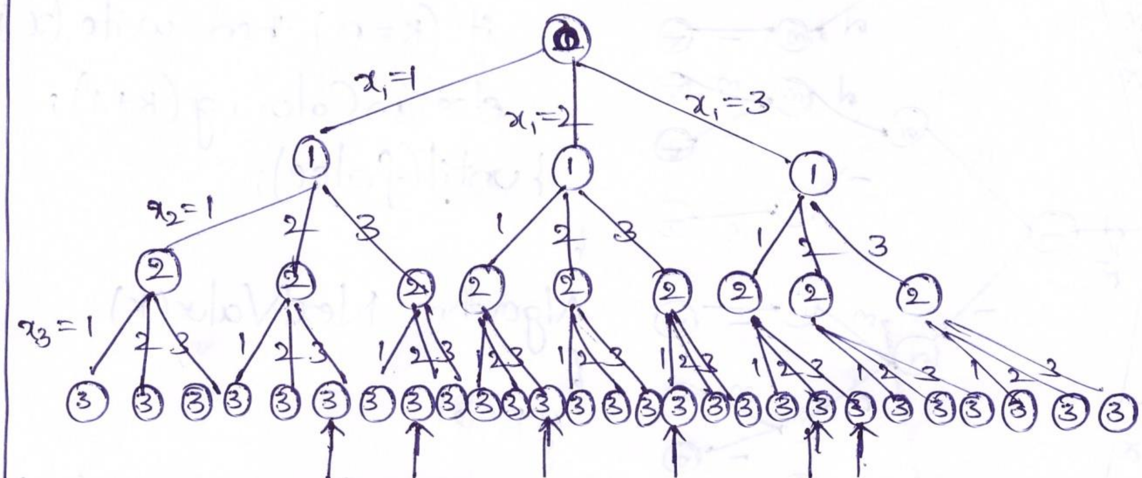
(3) **Graph Coloring Problem:** Graph Coloring is a problem of coloring each vertex in graph G in such away that no two adjacent vertices have same color. and yet m-colors are used. This problem is called m-coloring problem. If the degree of given graph is d then d+1 colors are used. The least number of colors needed to color the graph is called its chromatic number.

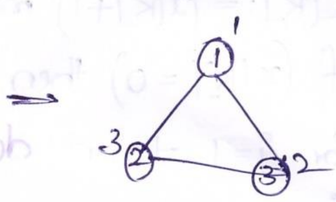
Consider a graph by its adjacency matrix $G[1:n, 1:n]$, where $G[i,j] = 1$ if (i,j) is an edge of G and $G[i,j] = 0$ otherwise. The colors are represented by the integers $1, 2 \ldots m$ and the solutions are given by the n-tuple $(x_1, x_2 \cdots x_n)$, where x_i is the color of node i.

Example: Color the following graph using m-Coloring problem

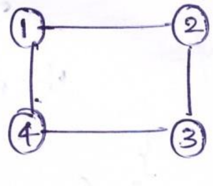


Sol: Since degree of the graph is 2, Hence 3 colors are required. State space tree for mColoring when $n=3$ and $m=3$ is as shown below.

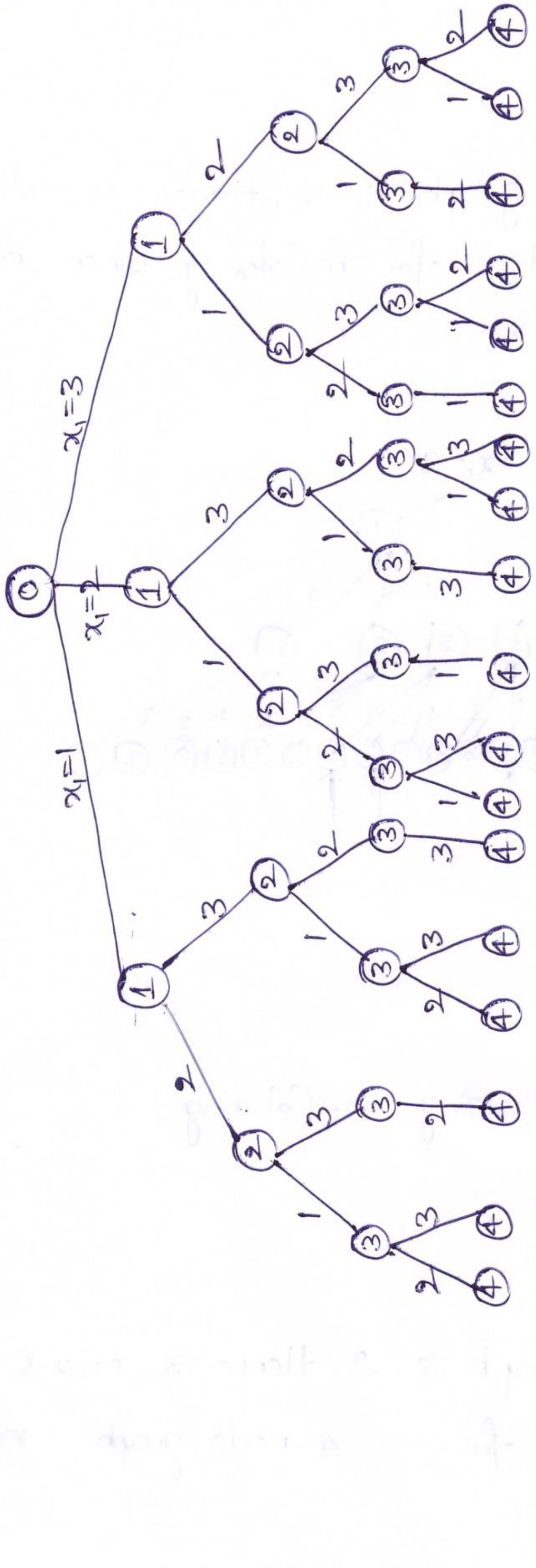




Ex: Color the following graph using m-Coloring



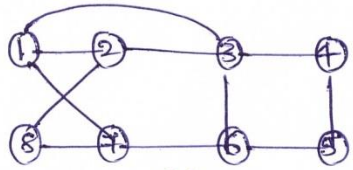
Sol: Since degree of the graph is 2. Hence 3 colors are required. State space tree for a 4-node graph and all possible 3-Colorings.



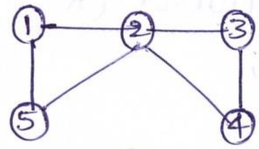
<u>Algorithm for Graph Coloring</u>

```
Algorithm mColoring(k)
{
  repeat
  {
    NextValue(k);
    if (x[k]=0) then return;
    if (k=n) then write (x[1:n]);
    else mColoring(k+1);
  } until(false);
}
Algorithm NextValue(k)
{
  repeat
  {
    x[k]:= (x[k]+1) mod (m+1);
    if (x[k]=0) then return;
    for j=1 to n do
    {
      if (G[k,j] ≠ 0 and
            x[k] = x[j])
        then break;
    }
    if (j=n+1) then return;
  } until(false);
}
```

4 Hamiltonian Cycles: Let G(V,E) be a connected graph with n vertices. A Hamiltonian cycle is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. For example, the following graph G1 contains the Hamiltonian Cycle 1,2,8,7,6,5,4,3,1. The graph G2 contains no Hamiltonian cycle.

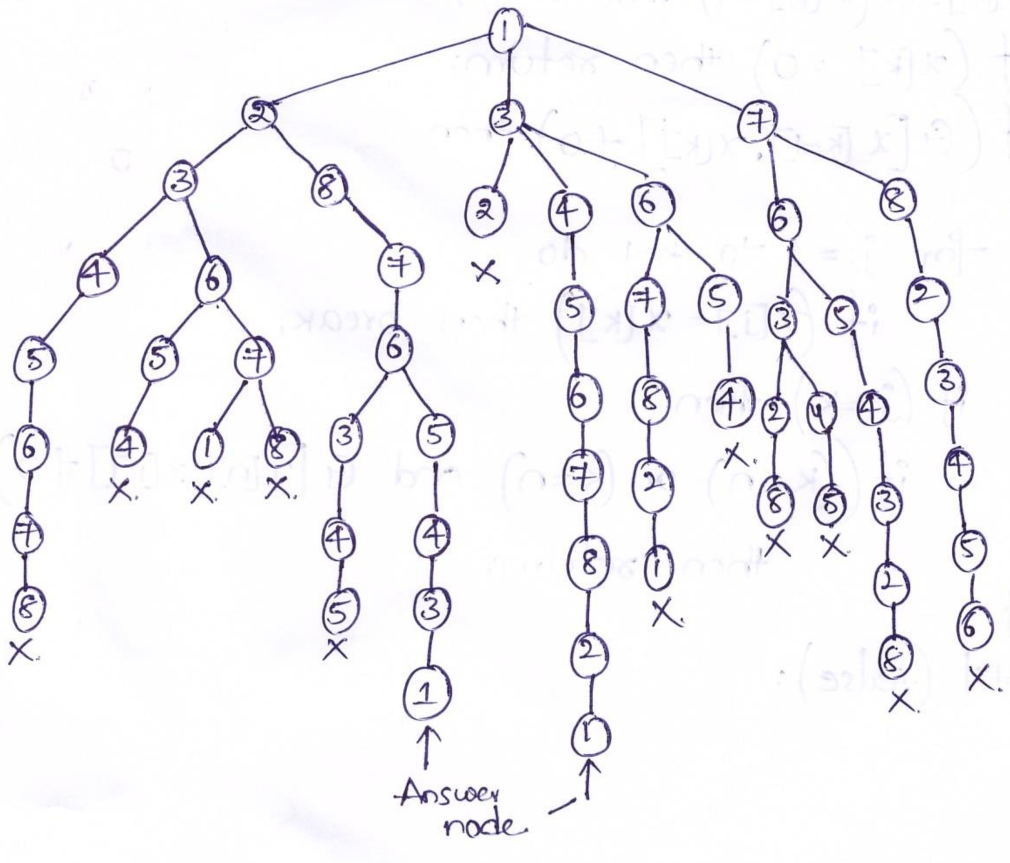


G1



G2

the graph may be directed or undirected, Only distinct cycles are output. State Space tree for G1 is ashown below:

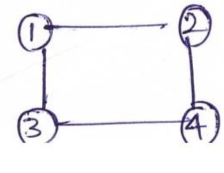


* Algorithm for Hamiltonian Cycle:

```
Algorithm Hamiltonian (k)
{
  repeat
  {
    NextValue (k);
    if (x[k] := 0) then return;
    if (k = n) then write (x[1:n]);
    else Hamiltonian (k+1);
  } until (false);
}
Algorithm NextValue (k)
{
  repeat
  {
    x[k] := (x[k]+1) mod (n+1);
    if (x[k] := 0) then return;
    if (G[x[k-1], x[k]] ≠ 0) then
    {
      for j := 1 to k-1 do
        if (x[j] = x[k]) then break;
      if (j = k) then
        if ((k < n) or (k = n) and G[x[n], x[1]] ≠ 0)
            then return;
    }
  } until (false);
}//
```

Frequently Asked Questions.

1. Explain in detail about Backtracking?
2. Write the control abstraction of Backtracking?
3. Explain the general Backtracking process using recursion?
4. Define the following terms: problem state, solution state, state space tree, answer states.
5. Suggest a solution for 8-Queen's problem?
6. Describe the 4-Queen's problem using Backtracking?
7. Draw and explain the portion of the tree for 4-Queens problem that is generated during Backtracking?
8. Describe the 8-Queen's problem?
9. Write Backtracking algorithm for 8-Queens?
10. Explain about n-Queens problem?
11. Write an algorithm for n-Queens problem?
12. Write a recursive Backtracking algorithm for Sum of Subsets?
13. Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ and $m = 35$. Find all possible subsets of w that sum to m. Do this using Sum of Subset Draw the portion of the state space tree that is generated.
14. Give an example of Sum of Subsets?
15. Explain about the sum of Subsets problem?
16. Explain about Graph Coloring problem and Chromatic Number?
17. Write an algorithm for Graph Coloring problem? (or)
18. Device a backtracking alg'm for m-coloring problem?
19. For the below graph draw the portion of state space tree generated by MCOLORING.

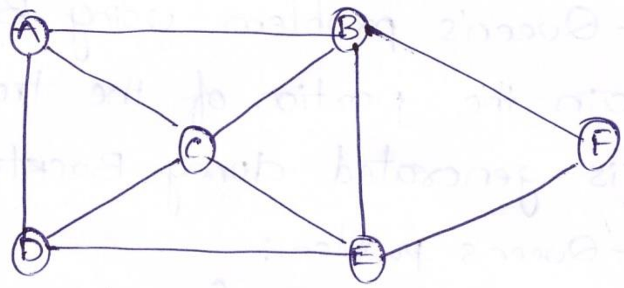


20. Give the state space tree for 3-coloring problem?

21. Explain how the Hamiltonian circuit problem is solved by using the backtracking concept.

22. Write an algorithm for generating all Hamiltonian Cycles.

23. Find the Hamiltonian circuit in the following graph by using Backtracking.

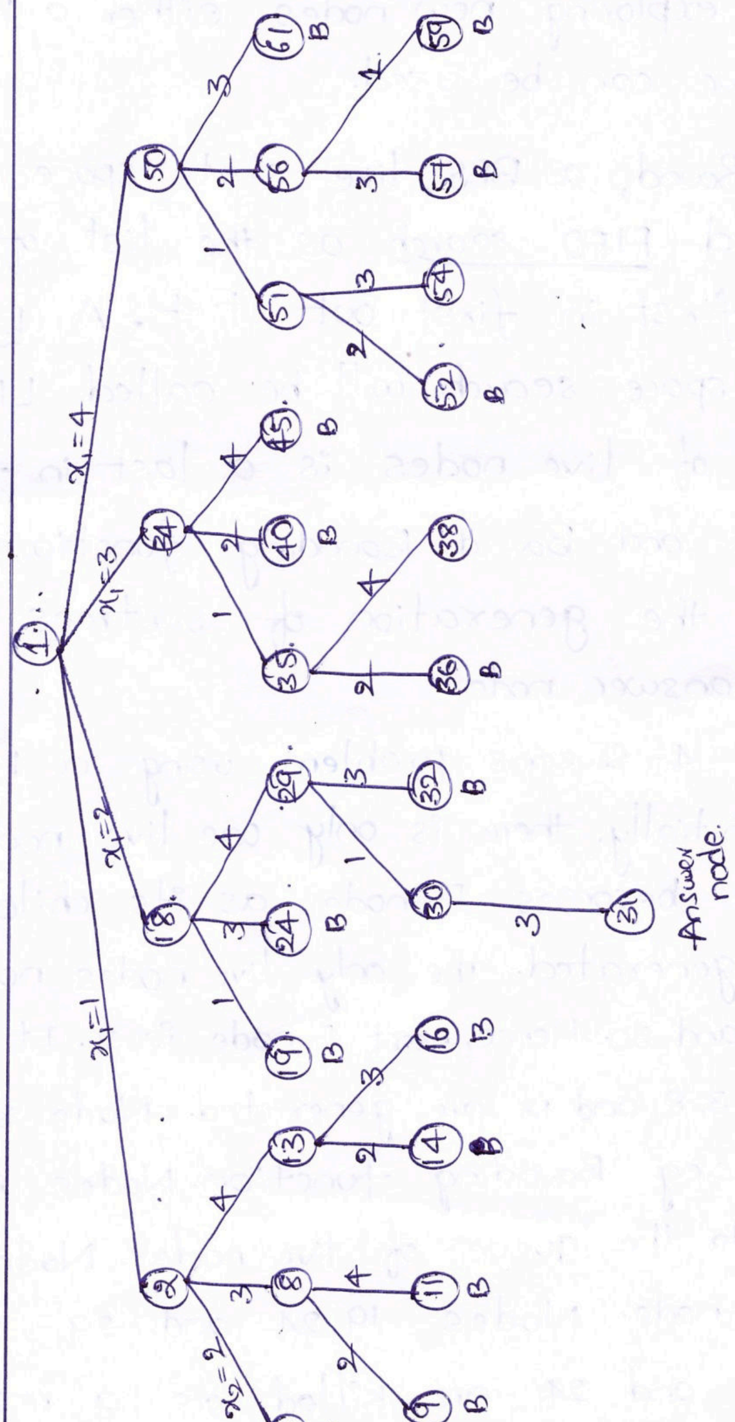


7. Branch and Bound.

General Method: Branch and Bound is general optimizatio technique that applies where the greedy method and dynamic programming fails. In Branch and Bound, a state space tree is built and all the children of E-nodes are generated before any other live node become E-node. For exploring new nodes either a BFS or D-Search technique can be used.

In Branch and Bound, a BFS like state space search will be called FIFO search as the list of live nodes is a first in first out list. A D-search like state space search will be called LIFO search as the list of live nodes is a last in first out list. In Branch and Bound, Bounding functions are used to avoid the generation of subtrees that do not contain an answer node.

Example: Consider the 4-Queens problem using a FIFO Branch and Bound. Initially, there is only one live node, i.e node 1. this node becomes E-node as its childrens 2, 18, 34 and 50 are generated. The only live nodes now are nodes 2, 18, 34 and 50. Hence, next E-node is 2. It is expanded and nodes 3, 8, and 13 are generated. Node 3 is immediately killed using Bounding function. Nodes 8 and 13 are added to the queue of live nodes. Node 18 becomes the next E-node. Nodes 19, 24 and 29 are generated. Nodes 19 and 24 are killed as a result of Bounding functions. Node 29 is added to the queue of live nodes. Now the E-node is 34. the following diagram shows the portion of the tree of 4-Queen's problem that is generated by a FIFO Branch and Band Search.



* **Least Cost Search (LC):** In both FIFO and LIFO branch and bound the selection rule for the next E-node is very complicated and blind. the selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

For speeding up the search process an intelligent ranking function $\hat{c}(\cdot)$ is used for live nodes. The next E-node is selected on the basis of this ranking function. the ideal way to assign ranks would be on the basis of additional computational effort needed to reach an answer node from the live node. For any node x, this cost could be ① the number of nodes in the subtree x that need to be generated before an answer node is generated., ② the number of levels the nearest answer node is from x.

Let $\hat{g}(x)$ be an estimate of additional effort needed to reach an answer node from x. Node x is assigned a rank using function $\hat{c}(\cdot)$ such that

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

Where, $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any non decreasing function.

In LC Search, a cost function $\hat{c}(\cdot)$ can be defined as follows:

① if x is an answer node, then $c(x)$ is the cost of reaching x from the root of the state space tree.

(ii) If x is not an answer node, then $c(x) = \infty$.

(iii) Otherwise, $c(x)$ equals the cost of a minimum cost answer node in the subtree x.

(*) Control abstraction for LC-Search:

```
Algorithm LCSearch (t)
{
  if *t is an answer node then output *t and return;
  E:=t;
  Initialize the list of live nodes to be empty;
  repeat
  {
    for each child x of E do
    {
      if x is an answer node then output the path
            from x to t and return;
      Add(x);
      (x -> parent):=E;
    }
    if there are no more live nodes then
    {
      write ("No answer node"); return;
    }
    E:=Least();
  }until (false);
}//
```

* **Boundings:** the Bounding functions are used to avoid the generation of subtrees that do not contain the answer nodes. In bounding, lower and upper bounds are generated at each node. A cost function $\hat{c}(x)$ is used to provide the lower bounds for any node x. Let upper is an upper bound on cost of minimum cost solution. In that case, all the live nodes with $\hat{c}(x) > \text{upper}$ can be killed.

Initially upper is set to ∞. After generating the children of current E-node, upper can be updated by minimum cost answer node.

* **Applications of Branch and Bound:**

① **0/1 Knapsack Problem:** The 0/1 knapsack problem states that - there are n objects $i = 1, 2, \cdots n$ and capacity of knapsack is m. and every object have its corresponding profits and weights. Then select some objects to fill the knapsack in such a way that it should not exceed the capacity of knapsack and maximum profit can be earned. i.e the knapsack problem is a maximization problem and this ~~Boun~~ Branch and Bound cannot be directly applied to maximization problem. This can be overcome by replacing the objective function $\sum P_i x_i$ by the function $-\sum P_i x_i$. Hence, the modified 0/1 knapsack problem can be stated as,

minimize $-\sum P_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \le m$, $x_i = 0$ or 1,
$1 \le i \le n$.

Algorithm for Computing Upper Band:

```
Algorithm UBand (cp, cw, k, m)
{
  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;
}
```

LC Branch and Band Solution: The following steps are used to solve 0/1 knapsack using LCBB.

(i) Draw state space tree.

(ii) Compute $\hat{c}(\cdot)$ and $u(\cdot)$ for each node.

(iii) If $\hat{c}(x) >$ upper kill node x.

(iv) Otherwise the minimum cost $\hat{c}(x)$ becomes E-node & generate its children.

(v) Repeat step (iii) and (iv) until all nodes get covered.

(vi) The minimum cost $\hat{c}(x)$ becomes the answer node. Trace the path in backward direction to get solution.

Example: Draw the portion of state space tree generated by LCKNAP for knapsack instance: $n=4, m=15, (P_1, P_2, P_3, P_4) = (10,10,12,18)$ and $(w_1, w_2, w_3, w_4) = (2,4,6,9)$.

Sol: Initially state space tree contains

$$① \; \hat{c} = -38, \; u = -32$$

Compute upper bound $u(1)$, using UBOund. for $i = 1,2,3,4$. $\because K=0$

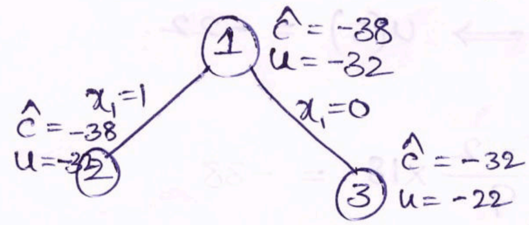
$$b = \cancel{0} \; \cancel{-10} \; \cancel{-20} \; -32 \qquad \Longrightarrow u(1) = -32.$$
$$C = \cancel{0} \; \cancel{2} \; \cancel{6} \; 12$$

Compute $\hat{c}(1)$ using the formula

$$\hat{c}(x) = u(x) - \left[\frac{m - \text{Current total wt}}{\text{Actual wt of remaining object}} \right] \times \text{Actual profit of remaining object}$$

$$\therefore \hat{c}(1) = -32 - \frac{15-12}{9} \times 18 = -38.$$

Now generate the childrens of node ① which are 2,3



Compute $u(2), u(3)$ for $i = 2,3,4$ $\boxed{\because K=1}$

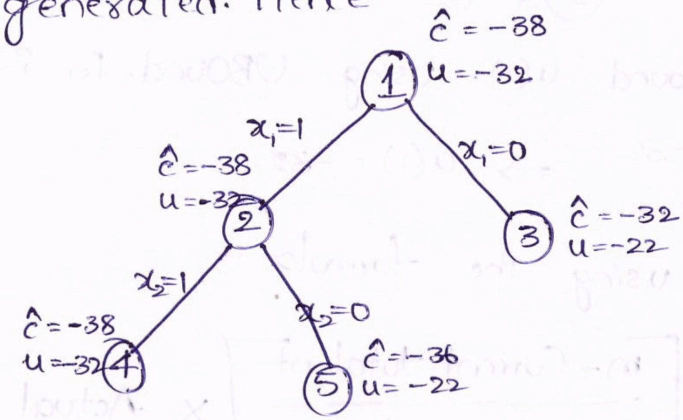
$$b = \cancel{-10} \; \cancel{-20} \; -32 \qquad \Longrightarrow u(2) = -32$$
$$C = \cancel{2} \; \cancel{6} \; 12$$

$$b = \cancel{0} \; \cancel{-10} \; -22 \qquad \Longrightarrow u(3) = -22.$$
$$C = \cancel{0} \; \cancel{4} \; 10$$

$\therefore \hat{c}(2) = u(2) - \frac{15-12}{9} \times 18 = -32 - 6 = -38$

$\therefore \hat{c}(3) = u(3) - \frac{15-10}{9} \times 18 = -22 - 10 = -32$

Since node ② has minimum ranking function, its children are generated. Here



Now compute $u(4)$ and $u(5)$ for $i = 3, 4$ $\boxed{\because k = 2}$

$b = -\cancel{20} -32$
$C = \cancel{6}\ 12$ $\Longrightarrow u(4) = -32.$

$b = -\cancel{10} -22$
$C = \cancel{2}\ 8$ $\Longrightarrow u(5) = -22.$

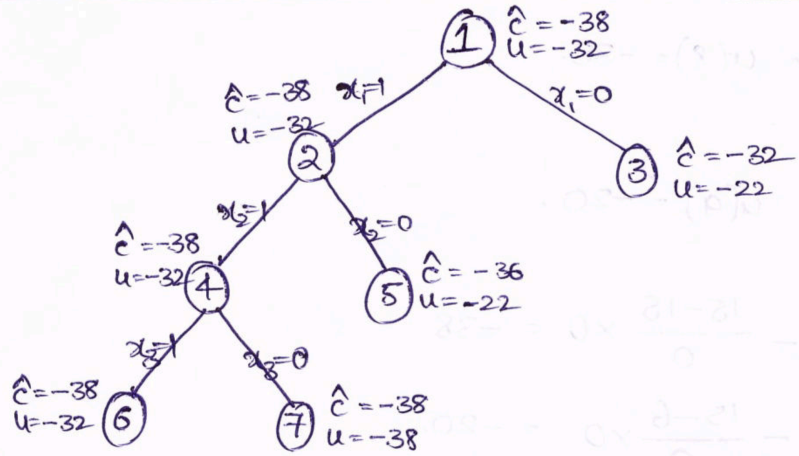
$\therefore \hat{c}(4) = -32 - \frac{15-12}{9} \times 18 = -38$

$\therefore \hat{c}(5) = -22 - \frac{15-8}{9} \times 18 = -36$

Since node ④ has min ranking function, its children are generated.

Compute $u(6), u(7)$ for $i = 4$ $\boxed{\because k = 3}$

$b = -32$
$C = 12$ $\Longrightarrow u(6) = -32.$

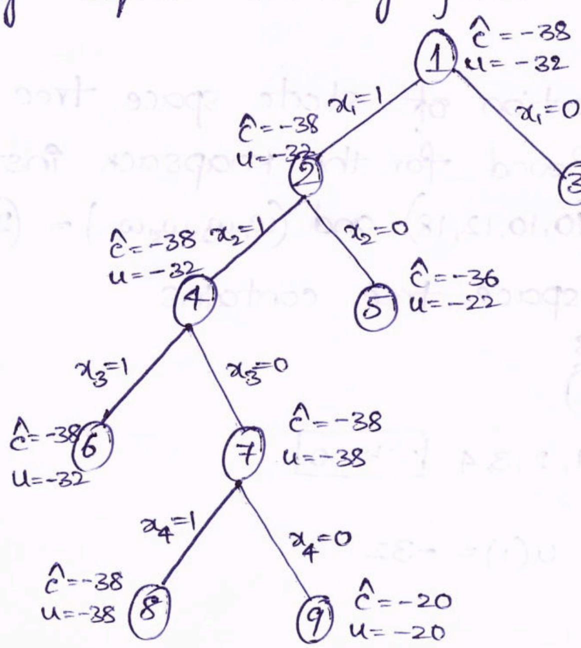


$b = -\cancel{20}\ -38$
$c = \cancel{6}\ 15$ $\Longrightarrow u(7) = -38.$

$$\therefore \hat{c}(6) = -32 - \frac{15-12}{9} \times 18 = -38$$

$$\therefore \hat{c}(7) = -38 - \frac{15-15}{0} \times 0 = -38$$

Generate the childrens of node 6 or 7, Since both having equal ranking function. Hence



Now compute $u(8), u(9)$ for $i=5$ $\boxed{\because K=4}$

$b = -38$
$c = 15$ $\Longrightarrow u(8) = -38.$

$b = -20$
$c = 6$ $\Longrightarrow u(9) = -20.$

$$\therefore \hat{c}(8) = -38 - \frac{15-15}{0} \times 0 = -38$$

$$\hat{c}(9) = -20 - \frac{15-6}{0} \times 0 = -20.$$

$\therefore \hat{c}(9) >$ upper node ⑨ is killed. Since node ⑧ has minimum $\hat{c}(\cdot)$ value it is the answer node. Hence optimal solution is $(x_1, x_2, x_3, x_4) = (1,1,0,1)$//

<u>FIFO Branch and Bound Solution</u>: Similar to LIFOBB, construct state ~~space~~ tree and compute $\hat{c}(\cdot)$ and upper at each node. If $\hat{c}(x) >$ upper kill the node immediately. Upper bound value is updated with minimum upper value automatically.

<u>Example</u>: Draw the portion of state space tree generated by FIFO Branch and Bound for the knapsack instance: $n=4, m=15, (P_1, P_2, P_3, P_4) = (10,10,12,18)$ and $(\omega_1, \omega_2, \omega_3, \omega_4) = (2,4,6,9)$.

<u>Sol</u>: Initially state space tree contains

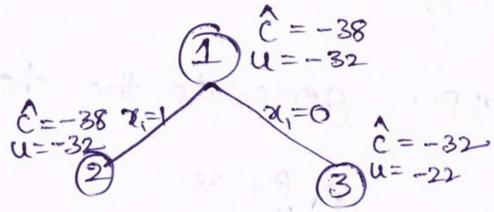
$\hat{c} = -38$
$u = -32$ ①

Compute $u(1)$ for $i = 1,2,3,4$ $\boxed{\because k=0}$

$b = \cancel{0} - \cancel{10} - \cancel{20} - 32$
$c = \cancel{0}\ \cancel{2}\ \cancel{6}\ 12$ $\Longrightarrow u(1) = -32.$

$$\therefore \hat{c}(1) = -32 - \frac{15-12}{9} \times 18 = -38.$$

Now, generate the childrens of node 1, which are 2, 3



Compute $\hat{c}(2)$, $u(3)$ for $i = 2, 3, 4$ $\boxed{\because k = 1}$

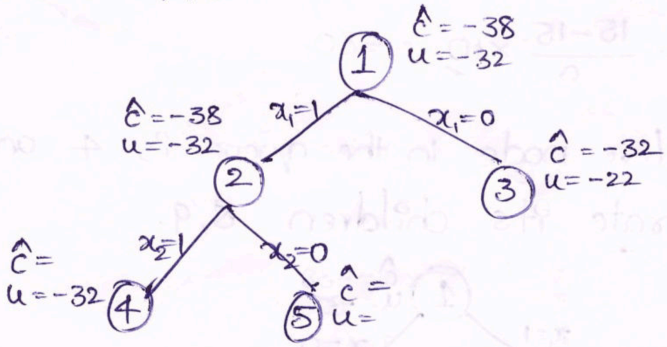
$b = \cancel{-10}\ \cancel{-20}\ -32$
$c = \cancel{7}\ \cancel{6}\ 12$ $\Longrightarrow u(2) = -32.$

$b = \cancel{0}\ \cancel{-10}\ -22.$
$c = \cancel{0}\ \cancel{4}\ 10$ $\Longrightarrow u(3) = -22.$

$\therefore \hat{c}(2) = -32 - \frac{15-12}{9} \times 18 = -38$

$\therefore \hat{c}(3) = -22 - \frac{15-10}{9} \times 18 = -32.$

Since node 2 ~~has minimum ranking function~~ is next node in Queue and $\hat{c}(2) <$ upper, generate its childrens i.e 4, 5.



Compute $u(4)$, $u(5)$ for $i = 3, 4$ $\boxed{\because k = 2}$

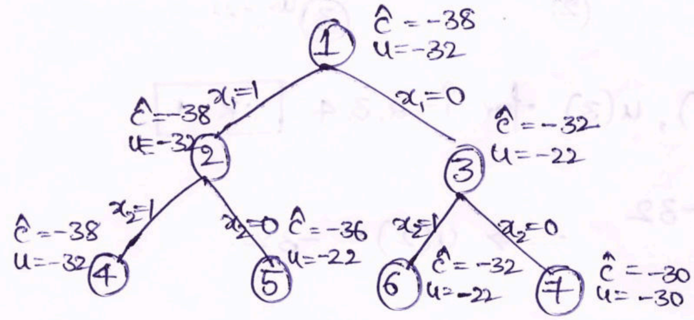
$b = \cancel{-20}\ -32$
$c = \cancel{6}\ 12$ $\Longrightarrow u(4) = -32.$

$b = \cancel{-10}\ -22$
$c = \cancel{7}\ 8$ $\Longrightarrow u(5) = -22$

$\therefore \hat{c}(4) = -32 - \frac{15-12}{9} \times 18 = -38$

$\hat{c}(5) = -22 - \frac{15-8}{9} \times 18 = -36$

Now, $\because \hat{c}(3) <$ upper generate the childrens of 3 i.e 6, 7



Compute $u(6), u(7)$ for $i = 3, 4$ $\boxed{\because k=2}$

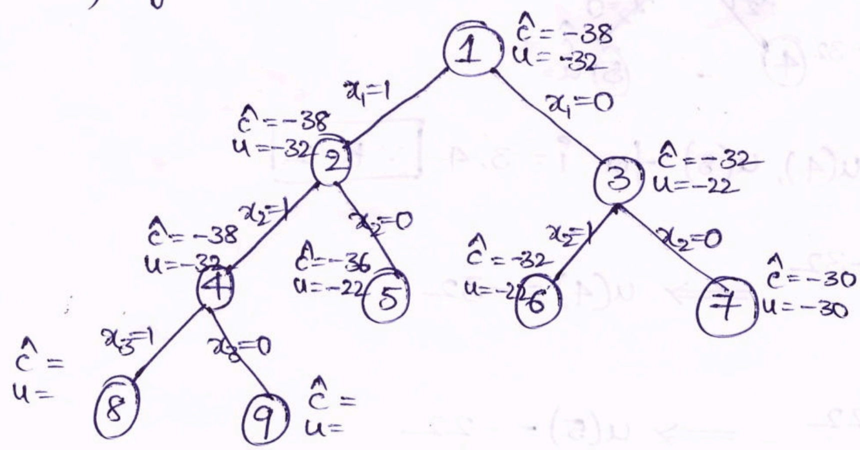
$b = \cancel{-10} - 22$
$c = \cancel{4}\ 10$ $\implies u(6) = -22$

$b = \cancel{0} - \cancel{12} - 30$
$c = \cancel{0}\ \cancel{6}\ 15$ $\implies u(7) = -30$

$\therefore \hat{c}(6) = -22 - \frac{15-10}{9} \times 18 = -32.$

$\therefore \hat{c}(7) = -30 - \frac{15-15}{0} \times 0 = -30.$

Since next live node in the queue is 4 and $\hat{c}(4) <$ upper $(-38 < -32)$, generate its children 8, 9.



Compute $u(8), u(9)$ for $i = 4$ $\because k=3$

$b = -32$
$C = 12$ $\Longrightarrow u(8) = -32.$

$b = -38$
$C = 15$ $\Longrightarrow \underline{u(9) = -38}.$

$\therefore \hat{c}(8) = -32 - \frac{15-12}{9} \times 18 = -38$

$\hat{c}(9) = -38 - \frac{15-15}{0} \times 0 = -38.$

Here upper is updated to -38.

Next live node in the queue is 5 and

$\because \hat{c}(5) > \text{Upper}$ $\;-36 > -38.$ kill the node 5

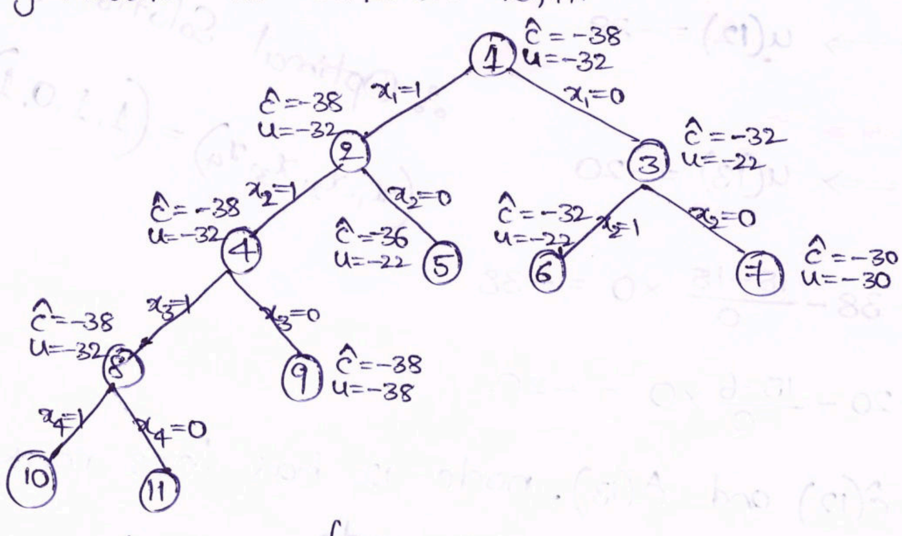
Next live node in the queue is 6 and.

$\because \hat{c}(6) > \text{upper}$ $\;-32 > -38$ kill the node 6.

Next live node in the queue is 7 and

$\because \hat{c}(7) > \text{upper}$ $\;-30 > -38$ kill the node 7.

Next live node in the queue is 8 and $\hat{c}(8) <$ upper$(-38 < -38$
generate its children 10, 11.



Node 10 is not answer node, hence compute $u(11)$

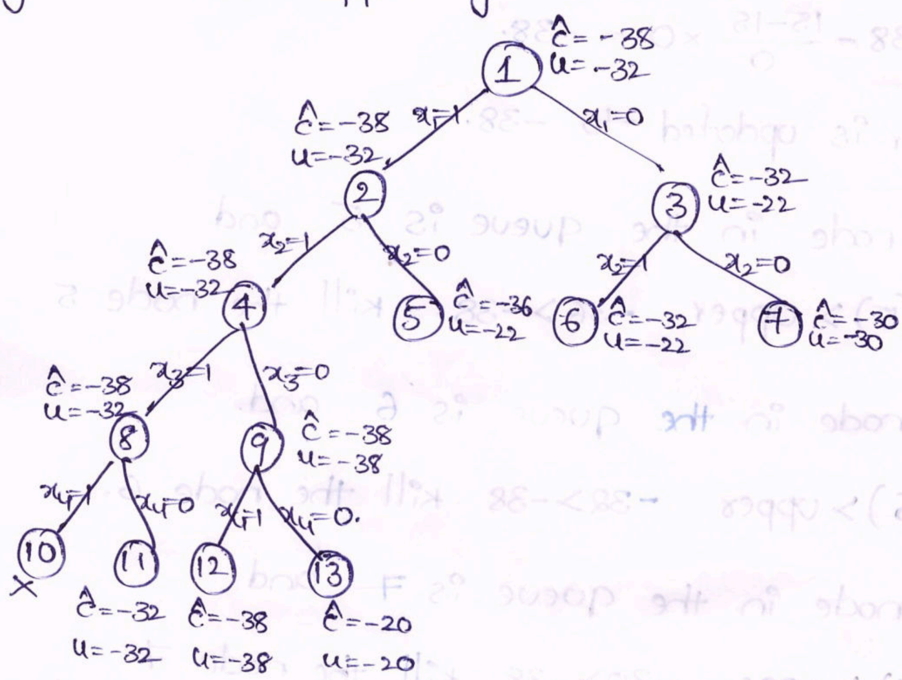
for $i=5$ $\boxed{\because K=4}$

$b = -32$ $\Longrightarrow u(11) = -32.$

$C = 12$

$\therefore \hat{c}(11) = -32 - \frac{15-12}{9} \times 8 = -32.$

Next live node in the queue is 9 and $\hat{c}(9)$ is not greater than upper generate its children. 12, 13.



Compute $u(12), u(13)$ for $i=5$ $\boxed{\because K=4}$

$b = -38$ $\Longrightarrow u(12) = -38.$

$C = 15$

$b = -20$ $\Longrightarrow u(13) = -20$

$C = 6$

$\therefore$ Optimal Solution is $(x_1, x_2, x_3, x_4) = (1, 1, 0, 1).$

$\therefore \hat{c}(12) = -38 - \frac{15-15}{0} \times 0 = -38$

$\hat{c}(13) = -20 - \frac{15-6}{0} \times 0 = -20.$

Among $\hat{c}(12)$ and $\hat{c}(13)$, node 12 has less ranking function which becomes answer node.

(2) <u>Travelling Sales Person Problem</u>: Let $G(V,E)$ be a directed or undirected graph with V vertices and E edges. Let C_{ij} = cost of the edge $<i,j>$, $C_{ij}=\infty$ if there is no edge between i and j.

In Branch and Bound, define a cost function $\hat{C}(\cdot)$ to search the traveling salesperson state space tree. The cost $\hat{C}(\cdot)$ is such that the solution node with least $\hat{C}(\cdot)$ corresponds to a shortest tour in G.

A better $\hat{C}(\cdot)$ can be obtained by using reduced cost matrix corresponding to G. A row or column is said to be reduced iff it contains atleast one zero and all remaining entries are non-negative. A matrix is reduced if every row and column is reduced. For example consider the following matrix with vertices.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{array}{l} \to 10 \\ \to 2 \\ \to 2 \\ \to 3 \\ \to 4 \\ \hline 21. \end{array} \Longrightarrow \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

$$\begin{array}{ccccc} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 0 & 3 & 0 & 0 \end{array} = 4.$$

Reduced Cost matrix is $\Longrightarrow$

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

Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R,S) corresponds to including edge <i, j> in the tour. If S is not a leaf, then the reduced cost matrix for S is obtained as follows:

(i) change all entries in row i and column j of A to ∞.

(ii) Set $A(j,1)$ to ∞.

(iii) Reduce all rows and columns in the resulting matrix except for rows and colums containing only ∞.

then $\hat{c}(S) = \hat{c}(R) + r + A(i,j)$

where r is the reduced cost.

Example: Apply the branch and bound algorithm to solve TSP for the following cost matrix.

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

Sol: Given that

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} -10 \\ -2 \\ -2 \\ -3 \\ -4 \end{matrix} \Longrightarrow \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Row Reduction = 21

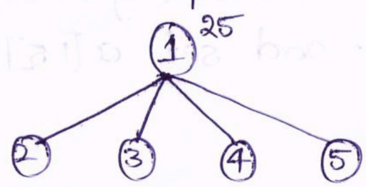
$\quad 1 \quad 0 \quad 3 \quad 0 \quad 0$ = Column Reduction

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

$\therefore$ Optimum Cost = Row Reduction + Column Reduction

= 21+4 = 25

Now construct the space tree as follows



To compute cost matrix for node ② make 1^{st} row and 2^{nd} column as ∞ and set $a[1,2] = a[2,1] = \infty$.

$$\therefore \begin{array}{cc} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix} & \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} \\ \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \end{array} & \end{array} \Longrightarrow \begin{aligned} \hat{c}(2) &= \hat{c}(1) + r + A(1,2) \\ &= 25 + 0 + 10 \\ &= 35. \end{aligned}$$

To compute reduced cost matrix for node ③ make 1^{st} row and 3^{rd} column as ∞ and set $a[1,3] = a[3,1] = \infty$.

$$\therefore \begin{array}{cc} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix} & \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} \\ \begin{array}{ccccc} 11 & 0 & 0 & 0 & 0 \end{array} & \end{array} \Longrightarrow \begin{aligned} \hat{c}(3) &= \hat{c}(1) + r + A(1,3) \\ &= 25 + 11 + 17 \\ &= 53. \end{aligned}$$

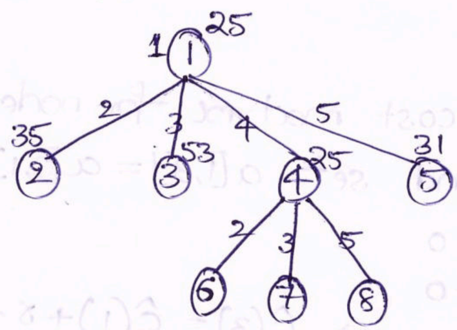
To compute reduced cost matrix for node ④ make 1^{st} row and 4^{th} column as ∞ and set $a[1,4] = a[4,1] = \infty$.

$$\therefore \begin{array}{cc} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix} & \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} \\ \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \end{array} & \end{array} \Longrightarrow \begin{aligned} \hat{c}(4) &= \hat{c}(1) + r + a(1,4) \\ &= 25 + 0 + 0 \\ &= 25. \end{aligned}$$

To compute reduced cost matrix for node ⑤ make 1^{st} row and 5th column as ∞ and set $a[1,5] = a[5,1] = \infty$.

$$\therefore \begin{array}{c} \left[\begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{array}\right] \begin{array}{c} -0 \\ -2 \\ -0 \\ -3 \\ -0 \end{array} \\ \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \end{array} \end{array} \Longrightarrow \begin{aligned} \hat{c}(5) &= \hat{c}(1) + r + A(1,5) \\ &= 25 + 5 + 1 \\ &= \underline{31} \end{aligned}$$

Since, cost of node 4 is optimum, generate its children nodes 6, 7, 8.



To compute reduced cost matrix for node ⑥ make 1^{st} row, 2^{nd} and 4^{th} columns as ∞ and set $a(1,2) = a(1,4) = a(2,1) = a(2,4) = a(4,1) = a(4,2) = \infty$.

$$\therefore \begin{array}{c} \left[\begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{array}\right] \begin{array}{c} -0 \\ -0 \\ -0 \\ -0 \\ -0 \end{array} \\ \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \end{array} \end{array} \Longrightarrow \begin{aligned} \hat{c}(6) &= \hat{c}(4) + r + a(4,2) \\ &= 25 + 0 + 3 \\ &= 28. \end{aligned}$$

To compute reduced cost matrix for node ⑦ make 1^{st} row, 3^{rd} and 4^{th} columns as ∞ and set $a(1,3) = a(1,4) = a(3,1) = a(3,4) = a(4,1) = a(4,3) = \infty$.

$$\therefore \begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} & \begin{array}{c} 0 \\ 0 \\ 2 \\ 0 \\ 0 \end{array} \\ \hline \begin{array}{ccccc} 11 & 0 & 0 & 0 & 0 \end{array} & \end{array} \Longrightarrow \hat{c}(7) = \hat{c}(4) + r + a(4,3)$$

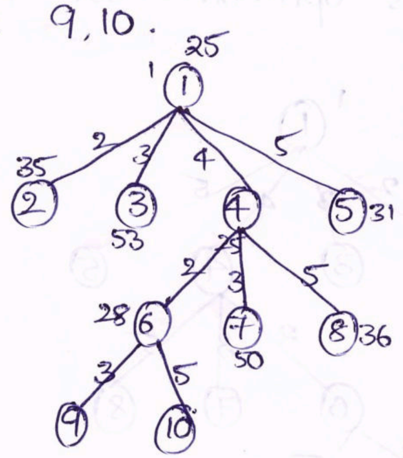
$$= 25 + 13 + 12 = 50.$$

To compute reduced cost matrix for node 8 make 1^{st} row, 4^{th} and 5^{th} columns as ∞ and set $a[1,4] = a[1,5] = a[4,1] = a[4,5] = a[5,1] = a[5,4] = \infty$.

$$\therefore \begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} & \begin{array}{c} 0 \\ 0_{11} \\ 0 \\ 0 \\ 0 \end{array} \\ \hline \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \end{array} & \end{array} \Longrightarrow \hat{c}(8) = \hat{c}(4) + r + a(4,5)$$

$$= 25 + 0 + 11$$
$$= 36.$$

Since, node 6 has optimum cost its childrens are generated i.e 9, 10.



To compute reduced cost matrix for node 9 make 1^{st} row, 2^{nd}, 3^{rd} and 4^{th} column as ∞ and set $a[1,2] = a[1,3] = a[1,4] = a[2,1] = a[2,3] = a[2,4] = a[3,1] = a[3,2] = a[3,4] = a[4,1] = a[4,2] = a[4,3] = \infty$.

$$\therefore \begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 2 \\ 0 \\ 11 \end{matrix} \\ \begin{matrix} 11 & 0 & 0 & 0 & 2 \end{matrix} & \end{array} \Longrightarrow \hat{c}(9) = \hat{c}(6) + r + a(2,3)$$

$$= 28 + 26 + 11$$

$$= 65$$

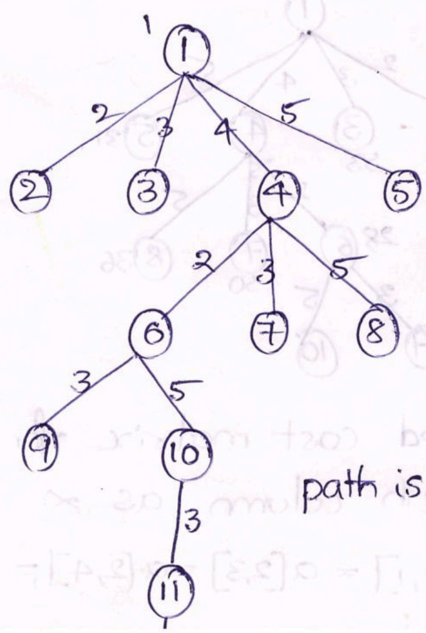
To compute reduced cost matrix for node 10 make 1st row, 2nd, 4th and 5th columns as ∞. and set $a[1,2] = a[1,4] = a[1,5] = a[2,1] = a[2,4] = a[2,5] = a[4,1] = a[4,2] = a[4,5] = a[5,1] = a[5,2] = a[5,4] = \infty$.

$$\therefore \begin{array}{c|c} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 & 0 & 0 \end{matrix} & \end{array} \quad \hat{c}(10) = \hat{c}(6) + r + a(2,5)$$

$$= 28 + 0 + 0$$

$$= 28.$$

Since, node 10 has optimum cost its only children 11 is generated.



path is 1, 4, 2, 5, 3, 1