9th July 2024

*1. Divide and Conquer

Lucky | Page No.: | Date: / /

- What is algorithm?

An algorithm is a finite set of instructions that if followed accomplish a particular task. All algorithms must satisfy the following criteria:

1) Input - zero or more quantities are externally supplied.
2) Output - atleast one quantity is produced.
3) Definiteness - each instruction is clear and unambiguous.
4) Finiteness - The algorithm should terminate after a finite number of steps.
5) Effectiveness - Every instruction must be very basic, so it can be carry out using pencil and paper.

- Algorithm validation:

Once algorithm is deviced it is necessary to show that it computes correct answer for all possible inputs. Once validity is shown, program can be written which is second phase that is program proving of program validation.

- Analysis of algorithm or performance analysis

As an algorithm is executed it uses the computer's CPU to perform operations and its memory to hold data and program

analysis of algorithm of performance analysis refers to the task of determining how much computing time and storage an algorithm requires.

Testing consist of two phases:

1) Debugging:
Debugging is the process of executing program on sample data set to determine whether faulty results occur and if so to correct them.

2) Profiling:
Profiling is a process of executing a correct program on data set and measuring time and space it takes to complete the result.

- Algorithm specifications:

Pseudo code conversions:

1) Comments begins with // and continue till end of line.
2) Blocks are indicated with matching braces
3) Statements are ~~iden~~ delimited by ;
An identifier begins with a letter, the datatype of a variable are not explicitely declared. The type will be clear from the context.
4) Assignments of the values is done by using the assignment statements.

5) There are two boolean values: true and false, in order to produce this, the logical operators (and, or, not) and relational operators ($<, \leq, =, \neq, >, \geq$) are provided.
6) Elements of multidimensional arrays are accessed using [ ] if 'A' is a two dimensional array i,j element of the array is denoted as A[i,j]
7) The looping statements:

```
1. for(initialization; condition; increment/decre.)
   {

   }
2. while {

   }
3. do{

   }
   while(con);
```

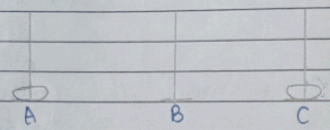
8) Conditional statements:

```
1.  if
    {

    }
2. if(condition)
   {

   }
   else{

   }
```

* Recursive algorithm:

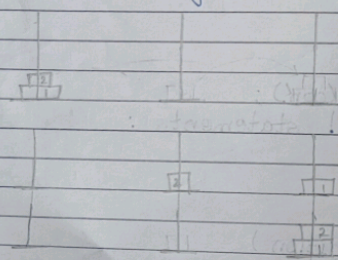
A recursive function is a function that is defined in terms of itself. Algorithm is said to be recursive if the same algorithm is invoked in the body. Algorithm that calls itself is direct recursive.
Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

1)

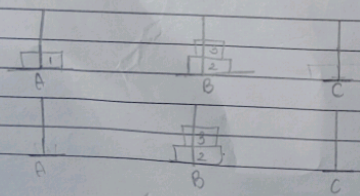


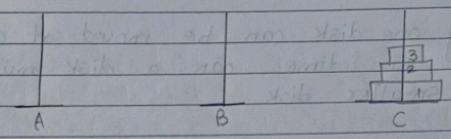
from using to

2)



3)





* Move two disk from A to B using C.
Move a disk from A to C.
Move two disk from B to C using A.

Generalize * Move n-1 disk from A to B using C.
Move a disk from A to C.
Move n-1 disk from B to C using A.

* Tower of Hanoi (n, A, B, C)
{
if (n>0)
{
Tower of Hanoi (n-1, A, C, B); A to B using C
write ("Move a disc from A to C");
Tower of Hanoi (n-1, B, A, C); B to C using A
}
}

* Recurrance relation:

An equation which de

* Conditions for Tower of Hanoi

1) There are three towers in which we have to move disk from one tower to another using intermediate tower for intermediate storage.

2) Only one disk can be moved at a time.
3) At no time can a disk must top on a smaller disk.

* Recurrance Relation:

A equation which defines a sequence recursively that equation is called recurrance equation. The recurrance equation have following generalize form:

$T(n) = T(n-1) + n$ for $n > 0$

$T(0) = 0$

* Forward substitution:

$T(n) = T(n-1) + n \quad \ldots (1)$

$T(0) = 0 \quad \ldots (2)$

Put $n = 1$

$T(1) = T(1-1) + 1$

$= T(0) + 1$

$= 0 + 1$

$= 1 \quad \ldots (3)$

Put $n = 2$

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

$= T(1) + 2$

$= 1 + 2$

$= 3$

Put $n = 3$

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

$= T(2) + 3$

$= 3 + 3$

$= 6$

Put $n = 4$

$T(4) = T(4-1) + 4$

$= T(3) + 4$

$= 6 + 4$

$= 10$

Now we generate generalize formula according to observation $T(n) = \frac{n^2+n}{2}$

* Backward substitution:

$T(n) = T(n-1) + n \quad \ldots (1)$

$T(0) = 0 \quad \ldots (2)$

$T(n-1) = T(n-1-1) + (n-1) \quad \ldots (3)$

Substitute (3) in (1)

$T(n) = T(n-2) + (n-1) + n \quad \ldots (4)$

$T(n-2) = T(n-2-1) + (n-2) \quad \ldots (5)$

substitute (5) in (4)

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

$\vdots$

$T(n) = T(n-x) + (n-x+1) + (n-x+2) + \ldots + n$

when $x=n$ we rewrite

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

$= 0 + 1 + 2 + \ldots\ldots + n$

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

* Performance Analysis:

There are many criteria upon which we can judge a algorithm. For instance, does it
1) Does it do what we want it to do?
2) Does it work correctly according to the original specifications of the task
3) Is their documentation that describes how to use it and how it works?
4) Are procedures created in such a way that they perform logical sub-functions?
5) Is the code readable?

Space complexity:

Space complexity of an algorithm is the amount of memory it needs to run to completion.

Time complexity:

Time complexity of an algorithm is the amount of computer time it needs to run to completion.

Performance evaluation is divided into ~~two measure~~ major phases:
1) Prior estimates - Performance Analysis
2) Posterior testing - Performance Measurement

The space needed by each of the algorithm is seen to be some of the following components:
1) A fix part - that is independent of the characteristic of an input and output. This part typically includes the instructions space that is space for the code, space for simple variables and fixed size component variable also called aggregate and space for constants.
2) A variable part that consist of the space needed by component variable whose size is depend on the particular problem instance being solved, the space needed by reference variable and the recursion stack space. The space requirement, $S(P)$ of any algorithm P may therefore be written as $S(P) = C + S_p$ where S_p is instant characteristic and C is constant for any given problem we need first to determine which instant characteristic to use to measure the space requirement.

* Time complexity:

The Time $T(P)$ taken by a program P is the sum of the compile time and the run time. We may assume that a compiled program will be run several times without recompilation. The run-time is denoted by t_p where $T(P) = C + t_p$, C = Compile time, t_p =

run time.

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

In the above algorithm it always takes three values for addition so it is fix values.

```
Algorithm Sum(a,n)
{
    S=0;
    for i=1 to n do
    {
        S = S+a[i];
    }
    returns;
}
```

Here, the value of n always changes according to user.

We can determine the no. of steps needed by a program to solve a particular problem which is called the step count.

(S/e) → Steps per execution.

S/e of the statement is the amount by which the count & changes as a result of a execution of that statement.

```
Algorithm Add(a,b,c,m,n)                    freq./step count
{                                           0   -
    for i=1 to m do                         0   -
        for j=1 to n do                     1   m+1
            c[i,j] = a[i,j] + b[i,j];       1   m(n+1)
}                                           1   mn
                                            0   -
                                                2mn+2m+1
```

```
Add(a,b,c,m,n)                              0   -
{                                           0   -
    for i=1 to m do                         1   m+1
    {                                       0   -
        count = count+2;                    1   m
        for j = 1 to n do                   1   m(n+1)
            count = count+1;                1   m.n
    }                                       0   -
}                                           0   -
                                                2mn+3m+2
                                                step count
```

```
Algorithm Fibonacci(n)
{
    if (n ≤ 1) then write(n);               =1
    else
    {
        fnm2=0; fnm1=1;                     -1
        for i=2 to n do                     n
        {
            fn = fnm1+fnm2;                 n-1
            fnm2 = fnm1;                    n-1
            fnm1 = fn;                      n-1
        }
        write f(n);                         +1
    }
}
                                            4n+1
```

* Asymptotic Notation:

1) Big O Notation:

The function $f(n) = Og(n)$ read as f of n $f(n)$ is O $g(n)$. If there exist positive constant c, n_0 such that $f(n) \leq c * g(n)$ for all n where $n \geq n_0$

e.g. $3n+2$

$f(n) = 3n+2 \leq c.n$

$c=4$

$3n+2 \leq c.n$

$3(1)+2 \leq 1.1 \quad n=1$

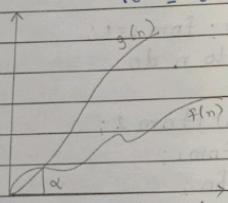
$5 \leq 1$ ×

$c=4, n=2$

$3n+2 \leq c.n$

$6+2 \leq 4 \times 2$

$8 \leq 8$ ✓



O notation / upper bound.

O complexity is also called the upper bound or worst case complexity. Worst case complexity is the maximum time taken by algorithm.

2) Ω notation:

The function $f(n) = \Omega g(n)$ if and only if there exist positive constant c and n_0 such that $f(n) \geq c * g(n)$ for all n, where $n \geq n_0$

e.g. $3n+2 \geq c.n \quad c=3, n=1$

$3+2 \geq 3$

$5 \geq 3$

Omega (Ω) is the best case of lower bound complexity it is the minimum time the algorithm will take to execute.

3) Θ notation: average case or tight bound

The function $f(n) = \Theta g(n)$ ~~such th~~ if and only if there exist positive constant c_1, c_2 and n such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$

e.g. $c_1 n \leq 2n+3 \leq c_2 n$

$c_1 = 1, n=1 \quad c_2 = 5$ or $c_1=1, n=1, c_2=6$

$1(1) \leq 2(1)+3 \leq 5(1) \quad 1 \leq 5 \leq 6$

$1 \leq 5 \leq 5$

	Θ complexity
Algorithm Add (a,b,c,m,n)	0
{	0
for i=1 to m do	Θ(m)
for j=1 to n do	Θ(m.n)
add c[i,j] = a[i,j]+b[i,j]	Θ(m.n)
}	0

* Randomize algorithm:

A randomize algorithm is one that makes use of randomizer, some of the decisions made in the algorithm depends upon output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run the output of the randomize algorithm also changes for the same input.

The execution time of randomize algorithm could also vary from run to run for same input. There are two types of randomize algorithm:

1) Algorithm that always produce same output on same inputs
2) Algorithm whose output might differ for the same input.

Example: Tossing a coin, rolling a die, playing a lottery, gambling and so on. Each possible outcome of an experiment is called a sample coin and a set of all possible outcomes is called samplespace.

Consider a example of throwing three coins the eight possibilities are:

S = { {H,H,H}, {T,H,H}, {H,T,H}, {T,T,H}, {H,T,T}, {T,T,T}, {H,H,T}, {H,T,H} }

* Divide and Conquer:

Algorithm D and C(P)
{
    if small(P) then return S(P);
    else {
    divide P into smaller instances $P_1, P_2, \ldots P_K$, $K \geq 1$;
    Apply D and C to each subproblems;
    return combine {D and C(P_1), D and C(P_2), ... D&C (P_K));
    }
}

Given a function to compute on n inputs the divide & conquer strategy suggest splitting the inputs into K distinct subsets. $K \geq 1$ and $K \leq n$ or $1 < K \leq n$

1) Divide:
Divide the problem P into number of sub-problems. These sub-problems are smaller instance of original problem such that $P_1, P_2 \ldots$ to P_K.

2) Conquer:
Solve the sub-problems by recursively if the size of sub-problem is small enough solve it directly.

3) Combine:
Combine the solution of sub-problem into solution of the original problem.

Date : / /

low mid-1 mid mid+1 high

* Binary 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

x = 151

low	mid	high	mid
1	7	14	$= \frac{14+1}{2}$

Search in second part

low	mid	high
8	11	14

low	mid	high
12	13	14

low	mid	high	
14	14	14	value found

- Algorithm BinSearch (a, n, x)

```
// An array a[1:n] of elements in
//    increasing order, n ≥ 0
// determine x is present 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 (x > a[mid]) then
            low = mid + 1
        else return mid;
    }
    else return 0;
}
```

Lucky Page No.:
Date : / /

- Algorithm RBinSearch (l, h, x)

```
{
    if(l==h)
    {
        if(A(l)==x)
        return l;
        else return 0;
    }
    else
        mid = (l+h) / 2,
    if (x == A[mid])
        return mid;
    if (x < A[mid])
    return RBinSearch (l, mid-1, x);
    else
    return RBinSearch (mid+1, h, x);
}
```

* Complexity of Binary Search :

Recurrance Relation is

$T_n = T_{n/2} + 1$

$T_1 = 0$

$T_n = (T_{n/4} + 1) + 1$

$T_n = T_{n/2^2} + 2$

$T_n = \left(T_{n/2^3} + 1\right) + 2$

$= T_{n/2^3} + 3$

∴ At k^{th} step

$T_n = T_{n/2^k} + k$

Assume $n/2^k = 1$

$n = 2^k$

$k = \log n$

$T_n = T_{n/2^k} + k$

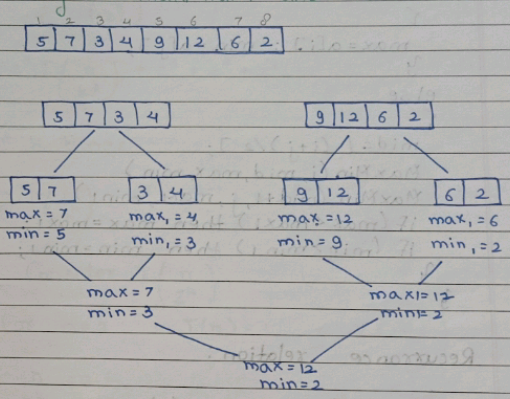
$T_n = T_1 + \log n$

$T_n = \log n$

worst case

$O(\log n)$

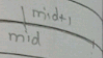
* Finding the maximum and minimum :



- Algorithm MaxMin (i, j, max, min)

```
a[1:n] global array
1 ≤ i ≤ j ≤ n
//set min and max to smallest & largest value
if(i=j)                          //only one element
max = min = a[i]
else (i=j-1)                     // two elements
{
   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;
}
}
```



Recurrance relation :

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

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

$T(n/2^2) = 2T\left(\frac{n}{2^3}\right) + 2$ .... (3)

Substitute eq.(2) in (1)

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

$2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2$ .... (4)

Substitute eq.(3) in (4)

$2^2\left[2T\left(\frac{n}{2^3}\right) + 2\right] + 2^2 + 2$

$2^3 T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2^1$

K times

$2^k T\left(\frac{n}{2^k}\right) + 2^k + 2^{k-1} + 2^{k-2} + \ldots + 2^2 + 2^1$ .... (5)

Consider $T\left(\frac{n}{2^k}\right)$

For $n = 2$, $T(n) = 1$

$\frac{n}{2^k} = 2$

$n = 2^k \cdot 2$

$n = 2^{k+1}$

$T\left(\frac{2^{k+1}}{2^k}\right)$

$T\left(\frac{2^k \cdot 2}{2^k}\right)$

$T(2) = 1$

So now equation (5) becomes,

Date : / /

$2^{k-1} + 2^k + 2^{k-1} + \dots + 2^2 + 2$

$2^k + (2^k + 2^{k-1} + \dots + 2^2 + 2)$ Geometric Progression

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

$\frac{n}{2^k} = 2 \quad \therefore \frac{n}{2} = 2^k$

$\therefore \frac{n}{2} + 2\left(\frac{n}{2} - 1\right)$

$\frac{n}{2} + n - 2$

$\frac{3n}{2} - 2$ .... Total no. of comparison

∴ Complexity is $\Theta(n)$

Divide and conquer algorithm for minmax proceeds as follow

Let $P = (n, a[i] \dots a[j])$ denote an arbitary instance of the problem. Here n is no. of elements in the list and we are interested in finding the maximum & minimum of the list.

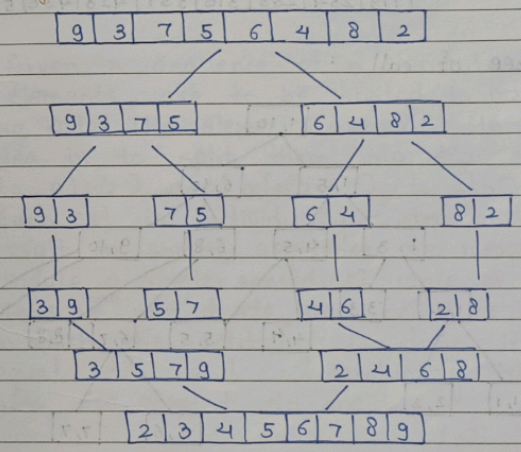
Let Small(P) be true when $n \leq 2$ in this case the maximum and minimum

Lucky Page No.:
Date : / /

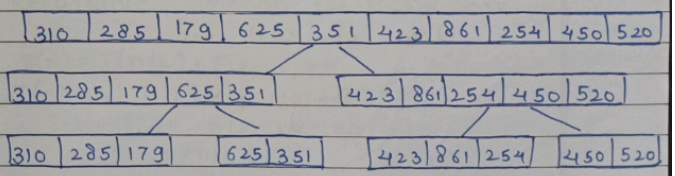
are a[i], If n=1.
If n=2 the problem can be solved by making one comparison.
If the list has more than two elements P is divided into smaller instant.
$P_1 = (\lceil n/2 \rceil, a[1] \dots a[n/2])$
$P_2 = (n - \lceil n/2 \rceil, a[\lceil n/2 \rceil + 1], \dots a[n])$

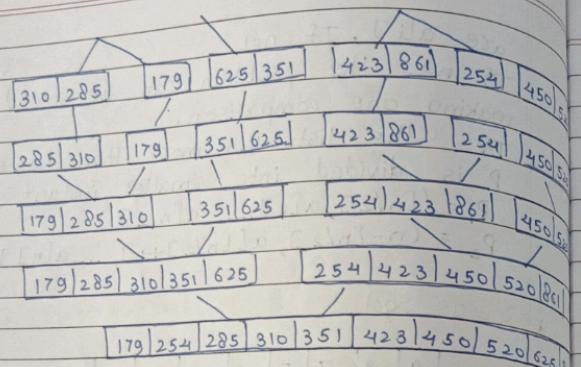
* Merge Sort :

1)

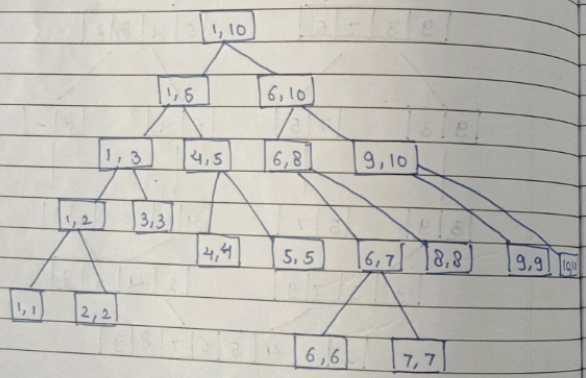


2)





• Tree of call :

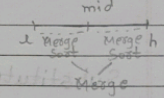


Tree of call for Merge Sort

Lucky | Page No.: | Date : / /

Algorithm MergeSort(l,h)

```
{
  if(l<h)
  [
    mid=(l+h)/2
    Merge Sort (l,mid)
    Merge Sort(mid+1,h)
    Merge (l,mid,h)
  ]
}
```



Given a sequence of n elements, the elements are to be sorted in non-decreasing order. The elements are a[1] to a[n]. The idea is to split them into two sets a[1] to a[n/2] and a[n/2+1] to a[n]. ~~is sa~~ Each set is individually sorted and the result in sorted sequence are merged to produce a single sorted sequence of n elements. Thus, ~~be~~ we use divide and conquer strategy in which the splitting is done in two equal sites and in combining two sorted sets are merged into one.

Recurrance relation for merge sort :

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

$T(n) = 2T(n/2) + n \quad \ldots\ldots (1)$

Substitute n/2 in eq. (1)

$T(n/2) = 2T(n/4) + \frac{n}{2} \quad \ldots\ldots (2)$

Substitute eq(2) in eq(1)

$T(n) = 2\{2T(n/4) + (n/2)\} + n$

$T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2n \quad \ldots\ldots (3)$

Substitute $\frac{n}{4}$ in eq(1)

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4} \quad \ldots\ldots (4)$

Substitute eq(4) in eq(3)

$T(n) = 2^2\left\{2T\left(\frac{n}{8}\right) + \frac{n}{4}\right\} + 2n$

$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n$

After substituting in eq(1)

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

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn \quad \ldots\ldots (5)$

$T(1) = 1$

let $\frac{n}{2^k} = 1$

$n = 2^k$

Apply $\log_2$ on both side to get k

$\log_2 n = \log_2 2^k$

$= k \log_2 2$

$= k$

$\therefore k = \log_2 n$

Substituting

$n = 2^k$

$\frac{n}{2^k} = 1$

$k = \log_2 n$ in eq(5)

$T(n) = nT(1) + n\log_2 n$

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

$\therefore$ Time Complexity $= O(n \log n)$

* Quick sort

	0									
A :	10	16	8	12	15	6	3	9	5	∞

B: 10 16 8 12 15 6 3 9 5 ∞
i j
swap

C: 10 5 8 12 15 6 3 9 16
i j
swap

D: 10 5 8 9 15 6 3 12 16
i j
swap

E: 10 5 8 9 3 6 15 12 16
i
i
j
swap
pivot

F: 6 5 8 9 3 10 15 12 16

G: 6 5 8 9 3 15 12 16
i j
6 5 8 9 3 swap i
i i j 12 15 16
swap
6 5 3 9 8
i
j
swap
3 5 6 9 8
swap
3 5 6 8 9 10 12 15 16

i will search element greater than pivot
j will search element smaller than pivot.
i will stop at the most at ∞
j will stop at pivot element.

increment i till you find value greater than pivot.
decrement j till you find value smaller than pivot.

In quick sort the elements are rearranged in a[1:n] such that a[i] ≤ a[j] for all i between 1 to n and all j between m+1 and n. Thus the element in a[1:m] and a[m+1:n] can be independently sorted. The rearrangement of the element is accomplish by picking some element same t = a[s] & then reordering other elements. So that, all elements appearing before t in a[1:n] are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as partioning.

```
Partition (l,h)
{
  pivot = A[l];
  i=l, j=h;
  {
    do
```

```
{
    i++;
while(A[i] <= pivot)
do
{
    j--;
while(A[j] > pivot);
if(i<j)
swap(A[i], A[j]);
}
swap(A[l], A[j]);
return j;
}

QuickSort(l,h)
{
  if(l<h)
  {
    j=partition(l,h)
    Quicksort(l,j)
    QuickSort(j+1,h);
  }
}
```

Best case:

$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \ldots\ldots (1)$

$T(1) = 1$

we write in place of n as n/2

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

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

Derive $T\left(\frac{n}{4}\right)$ using $T(n)$

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

$= 8T\left(\frac{n}{8}\right) + 3n$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$

$\vdots$

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

let $2^k = n$

$\log_2 2^k = \log n$

$k \log 2 = \log n$

$k = \log n$

Substitute in eq.(2) value of k

$T(n) = n.T\left(\frac{n}{n}\right) + n\log n$

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

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

∴ Complexity is $O(n \log n)$

Worst case

$T(n) = T(n-1) + n \quad \ldots\ldots(1)$

Substitute $(n-1)$ in place of n in eq.(1)

$\therefore T(n-1) = T(n-2) + (n-1) \quad \ldots\ldots(2)$

Now substitute eq(2) in (1)

$\therefore T(n) = T(n-2) + (n-1) + n \quad \ldots(3)$

Derive $T(n-2)$ from eq(1)

$T(n-2) = T(n-3) + (n-2) \quad \ldots\ldots(4)$

Substitute eq(4) in eq(3)

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

$\vdots$

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

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

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

$= O(n^2)$

* Selection Sort : complexity: $O(n^2)$

We are given n elements a[1:n] and are required to determine the k^{th} smallest element. If the partitioning element v is position at a[j] then j-1 elements are less than or equal to a[j] and n-j elements are greater than a[j]. Hence,

1Y if $k < j$ then the k^{th} smallest element is in a[1:j-1]

2Y if $k = j$ then a[j] is the k^{th} smallest element.

3Y if $k > j$ then the k^{th} smallest element is the k-j smallest element in a[j+1:n]

The k^{th} smallest element is position at a[k] and the remaining elements are partition so that $a[i] \le a[k]$ for $1 \le i < k$

$a[i] > a[k]$ for $k < i \le n$

Select(a,n,k)

Select the k^{th} smallest element in a[1:n] and place it in the k^{th} position of a[ ]. The remaining elements are rearrange such that

$a[m] \le a[k]$ for $1 \le m < k$

$a[m] \ge a[k]$ for $k < m \le n$

low = 1; up = n+1;

```
a[n+1] = ∞
repeat
{
  j = partition(a, low, up)

if (k=j) then return;
else if (k<j) then up = j;
else low = j+1;
}
until (false);
}
```

1st August 2024

* 2. Greedy Method *

Lucky Page No.: Date: / /

- Greedy Method is a straight forward Technique If a problem has n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution.
- We need to find a feasible solution that either maximize or minimize a given objective function. A feasible solution that does this is called an optimal solution. The greedy method suggest that one can device an algorithm that works in stages considering one input at a time. At each stage a decision is made regarding whether a particular input is in an optimal solution.

```
Greedy(a,n)
{
  Solution = ϕ
  for i:=1 to n do
  {
    x = Select(a);
    if feasible(Solution, x) then
      solution = Union(Solution, x);
  }
return solution;
}
```

IMP* 5 to 7 Marks — Knapsack problem

Q $n=7 \quad m=15$

object	1	2	3	4	5	6	7
profit	10	5	15	7	6	18	3
weight	2	3	5	7	1	4	1
P/W	5	1.6	3	1	6	4.5	3

maximum profit

$2-2=0$	object 2
$3-1=02$	object 7
$8-5=3$	object 3
$12-4=8$	object 6
$14-2=12$	object 1
$15-1=14$	object 5

$x = \{1, \frac{2}{3}, 1, 0, 1, 1, 1\}$

15 Profit

weight nusar ghyach

$$\sum x_i w_i = 1\times2 + \frac{2}{3}\times3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1$$
$$= 2 + 2 + 5 + 0 + 1 + 4 + 1$$
$$= 15$$

$$\sum x_i p_i = 1\times10 + \frac{2}{3}\times5 + 1\times15 + 0\times7 + 1\times6 + 1\times18 + 1\times3$$
$$= 10 + \frac{10}{3} + 15 + 0 + 6 + 18 + 3$$
$$= 55.33$$

- Greedy Knapsack(m,n)
P[1:n] and w[1:n] are profit, weight respectively n objects such that

$$\frac{P[i]}{w[i]} \geq \frac{P[i+1]}{w[i+1]}$$

m is knapsack size

```
{
  for i=1 to n do x[i]=0;
    U=m;
  for i=1 to n do
  {
    if (w[i] > U) then break;
      x[i]=1;
      U = U - w[i];
    }
    if (i<=n) then x[i] = U/w[i];
  }
```

We are given n objects and a knapsack, object i has a weight w_i and knapsack has capacity m. If a fraction x_i of object i is placed into the knapsack then profit of $p_i x_i$ is earned. The object is to obtain a filling of the knapsack that maximize total profit earned. Since, the knapsack capacity is m we require the total weight of all chosen objects to the atmost m.

$$\text{maximize} \sum_{1\leq i\leq n} p_i x_i$$

$$\text{Subject to} \sum_{1\leq i\leq n} w_i x_i \leq m$$

and

$$0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

Q Find an optimal solution to the knapsack instance $n=7, m=15$

object	1	2	3	4	5	6	7
profit	15	10	7	18	3	4	2
weight	2	3	5	7	1	4	1
$\frac{P}{\omega}$	7.5	3.3	1.4	2.5	3	1	2

$\frac{1}{5}$	1 - 1 = 0	object 3
	2 - 1 = 1	object 7
	9 - 7 = 2	object 4
	10 - 1 = 9	object 5
	13 - 3 = 10	object 2
	15 - 2 = 13	object 1

15

$x = \{1, 1, \frac{1}{5}, 1, 1, 0, 1\}$

$\sum x_i w_i = 1\times2 + 1\times3 + \frac{1}{5}\times5 + 1\times7 + 1\times1 + 0\times4 + 1\times1$
$= 2+3+1+7+1+0+1$
$= 15$

$\sum x_i p_i = 1\times15 + 1\times10 + \frac{1}{5}\times7 + 1\times18 + 1\times3 + 0\times4 + 1\times2$
$= 15+10+1.4+18+3+0+2$
$= 49.4$

* Job sequencing with deadlines :

```
GreedyJob(d, J, n)
{
    J = {1};
    for i = 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J = J ∪ {i};
    }
}
```

We are given a set of n jobs associated with job; is an integer deadline $d_i \geq 0$ and $p_i > 0$ ~~asso~~ for any job; the profit p_i is earned if and only if the job is completed by its deadline. To complete a job one has to process the job on a machine for one unit of time. Only one machine is available for processing job. A feasible solution for this problem is a subset; of jobs such that each job in the subset can be completed by its deadline. The value of a feasible solution j is the sum of profits of jobs in j. An optimal solution is a feasible solution with maximum library.

Q Profit $(J_1, J_2, J_3, J_4, J_5)$ $(20, 15, 10, 5, 1)$ deadline $(J_1, J_2, J_3, J_4, J_5)$ $(2, 2, 1, 3, 3)$ $0 \frac{J_2}{1} \frac{J_1}{2} \frac{J_4}{3}$

Job consider	Slot assign	Solution	Profit
—	—	—	0
J_1	[1, 2]	J_1	20

J_2	[0,1][1,2]	$J_1 J_2$	20+15
J_3	[0,1][1,2]	$J_1 J_2$	20+15
J_4	[0,1][1,2][2,3]	$J_1 J_2 J_4$	20+15+5
J_5	[0,1][1,2][2,3]	$J_1 J_2 J_4$	40

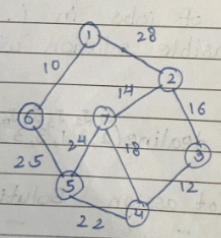
Q.

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

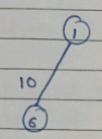
0 — J_6,J_4 — 1 — J_3 — 2 — J_1 — 3 — J_2 — 4

Job consider	Slot assign	Solution	Profit
–	–	–	0
J_1	[2,3]	J_1	35
J_2	[2,3][3,4]	$J_1 J_2$	35+30
J_3	[1,2][2,3][3,4]	$J_1 J_2 J_3$	35+30+25
J_4	[0,1][1,2][2,3][3,4]	$J_1 J_2 J_3 J_4$	35+30+25+20
J_5	[0,1][1,2][2,3][3,4]	$J_1 J_2 J_3 J_4$	35+30+25+20
J_6	[0,1][1,2][2,3][3,4]	$J_1 J_2 J_3 J_4$	35+30+25+20
J_7	[0,1][1,2][2,3][3,4]	$J_1 J_2 J_3 J_4$	110

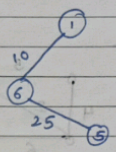
* Minimum Cost Spanning Tree :



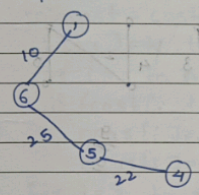
Step I :



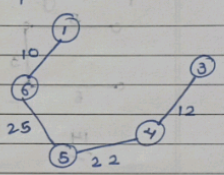
Step II :



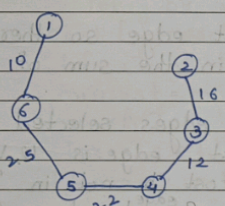
Step III :



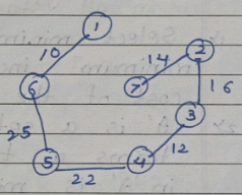
Step IV :



Step V :



Step VI :



Prim's Algorithm

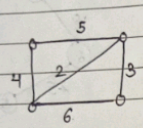
Let $G = \{V, E\}$ be undirected connected graph. Subgraph $t = V, E'$ of Graph G is a spanning tree of graph G if and only if 't' is a tree. In spanning tree no. of vertices are same and connected with fewer number of edges.

n vertices at least n-1 edges and

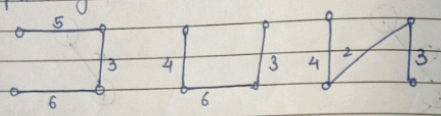
Date : / /

connected graph is a tree i.e no cycle

e.g.

1) 

→ Spanning tree :



14 13 9

* Prims Algorithm: Note: The graph should be tree in all stages.

1) Select minimum cost edge so there is minimum increase in the sum of the cost of the edges.

2) 'A' is a set of edges selected. 'A' forms a tree, next edge is (U, V) included in 'A' is minimum cost edge & not in 'A'. Set 'A' should always be tree. Select edge(i,j) such that vertex i is already included in tree and j is not yet included cost of (i,j) is minimum among all edges (a,l) such that k is in the tree and vertex l is not in the tree.

3) To determine this edge(i,j) with each vertex j not get included in the tree. A value near[j]

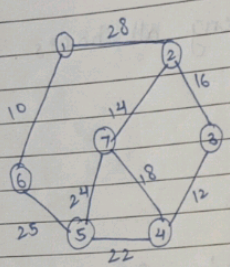
Lucky Page No.:
Date : / /

is minimum among all choices.

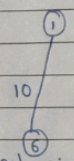
Algorithm:

```
Prim(E,cost,n,t)
E is set of Edges
{
  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]
      near[i] = l else
      near[i] = k;
  for i = 2 to n-1 do
    {
      let j be an index such that near[j] ≠ 0
    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;
  }
```

* KrusKals Algorithm:

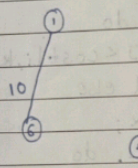


Step 1:

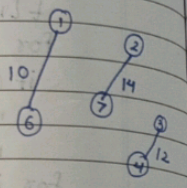


Select minimum cost edge.

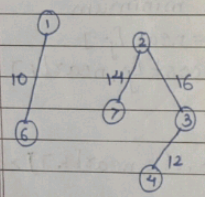
Step 2:



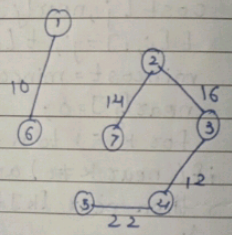
Step 3:



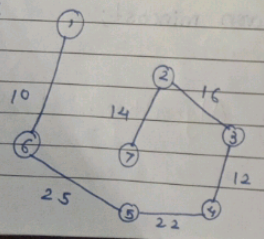
Step 4:



Step 5:



Step 6:

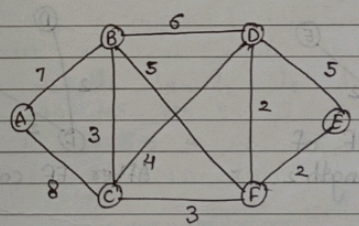


Date : / /

$t = \phi$;
while(t has less than n-1 edges) and ($E \neq 0$)) do
{
choose an edge (v,w) from E of lowest cost;
Delete (v,w) from E;
if (v,w) does not create a cycle in t then
add (v,w) to t;
else discard (v,w);
}

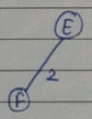
The set 't' of edges so far selected for the spanning tree be such that it is possible to complete 't' into a tree may not be tree in all stages in the algorithm. It only be a forest, since the set of edges 't' can be completed into tree if and only if there are no cycle in tree.

1)

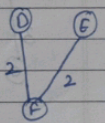


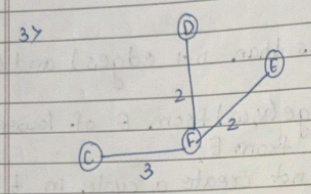
→ a) By Prims algorithm:

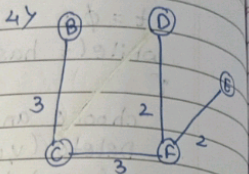
1)

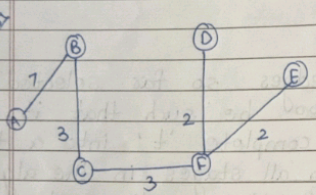


2)



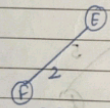






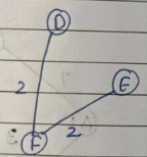
2)

1)



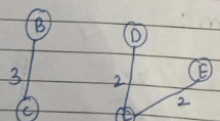
Consider Ef of minimum length 2.

2)

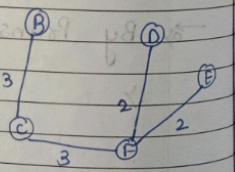


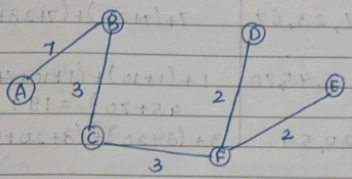
After Ef consider Df

3)



4)





* Optimal Storage on tape :

1) $n = 3$

$(l_1, l_2, l_3) = (5, 10, 3)$

$(1,2,3) = (5) + (5+10) + (5+10+3) = 38$
$(1,3,2) = 5 + (5+3) + (5+3+10) = 31$
$(2,1,3) = 10 + (10+5) + (10+5+3) = 43$
$(2,3,1) = 10 + (10+3) + (10+3+5) = 41$
$(3,1,2) = 3 + (3+5) + (3+5+10) = 29$
$(3,2,1) = 3 + (3+10) + (3+10+5) = 34$

2) $n = 10$

	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9	l_{10}
	10	20	45	70	1	3	7	54	23	67
index	1	2	3	4	5	6	7	8	9	10
Ascending order	1	3	7	10	20	23	45	54	67	70
mod	1	2	0	1	2	0	1	2	0	1

index mod no. of tapes

e.g. mod 3

Date : / /

tape 0	7, 23, 67	7+(7+23)+(7+23+67) = 134
tape 1	1, 10, 45, 70	1+(1+10)+(1+10+45)+(1+10+45+70) = 194
tape 2	3, 20, 54	3+(3+20)+(3+20+54) = 103

Mean retrival time (MRT) = $\frac{134+194+103}{3}$

$= \frac{431}{3}$

$= 143.6$

There are n programs that are to be stored on a computer tape of length l associated with each program i is a length l_i. We assume that whenever a program is to be retrieved from the step the tape is initially position at the front. Hence if the programs are stored in the order $I = i_1, i_2, \ldots, i_n$ the time t_j needed to retrieve program i_j is proportional to $\sum_{1 \le k \le j} l_{ik}$ if all programs are retrived equally often then the expected mean retrieval time (MRT) is $(\frac{1}{n}) \sum_{1 \le j \le n} t_j$. In the optimal storage on tape problem we are required to find a permutation for n programs so that when they are stored on the tape in this order the MRT is minimized. Minimizing the MRT is equivalent to minimizing d(I) = $\sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{ik}$

Lucky Page No.:
Date : / /

Algorithm :

```
Store(n,m)        Assigning program to tapes
//n is no. of programs
//m is no. of tapes
{
   j=0; //next tape to store on
   For i=1 to n do
   {
   write("append program", i, "to permutation for
                                tape", j);
   j=(j+1) mod m;
   }
}
```

Q.1 Find an optimal placement for 13 programs on 3 tapes t_0, t_1 & t_2 whotheree programs of length

	12	5	8	32	7	5	18	26	4	8	11	10	6
index	1	2	3	4	5	6	7	8	9	10	11	12	13
Ascending	3	4	5	5	6	7	8	10	11	12	18	26	32
mod	1	2	0	1	2	0	1	2	0	1	2	0	1

tape 0 : 5, 7, 11, 26

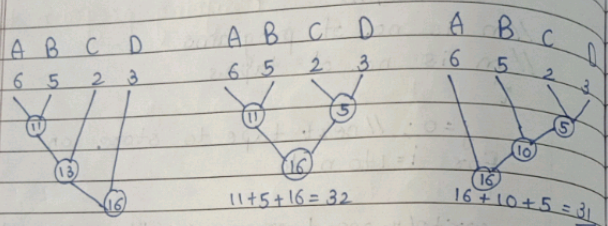
tape 1 : 3, 5, 8, 12, 32

tape 2 : 4, 6, 10, 18

Date : / /

* Optimal Merge pattern :

1) A B C D
6 5 2 3



11+13+16=40

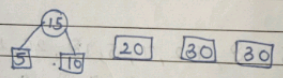
11+5+16=32

16+10+5=31

2) list x_1 x_2 x_3 x_4 x_5
Sizes 20 30 10 5 30

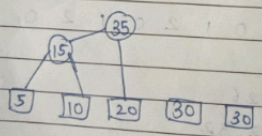
Asc 5 10 20 30 30
x_4 x_3 x_1 x_2 x_5

Step I :



First, merge 5 and 10.

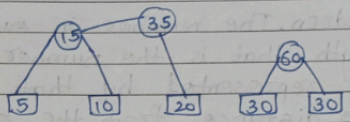
Step II :



Then merge 15 and 20

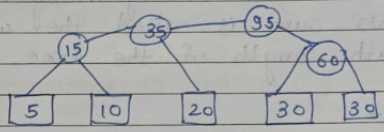
Lucky Page No.:
Date : / /

Step III :



Then merge 30 and 30.

Step IV :



Last, merge 35 and 60.

15+35+95+60= 205

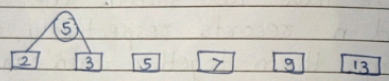
We have two sorted files containing n and n records respectively. we have to merge them together when more than two sorted files are to be merge together a merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if files x_1, x_2, x_3 and x_4 are to be merged ~~be~~ we could 1st merge x_1 and x_2 to get file y_1 then we could merge y_1 and x_3 to get y_2 so on. The leaf nodes are drawn as squares and represent the given files. The nodes are called external nodes. The remaining nodes are drawn as circles 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 that is the number of records of the file represented by that node. If d_i is the distance from the root to the external node for file x_i and q_i the length of x_i is total number of records move from its binary merge tree is $\sum_{i=1}^{n} d_i q_i$. This sum is called the weighted external path length of the tree.

3) list

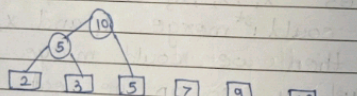
2 3 5 7 9 13

Step I:



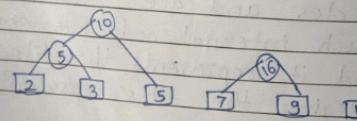
First merge 2 and 3.

Step II:



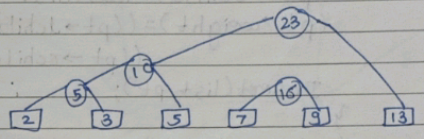
Then merge 5 and 5.

Step III:



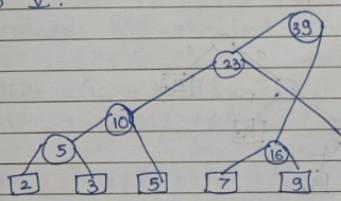
Then merge 7 and 9.

Step IV:

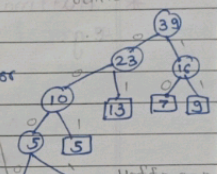


Then merge 10 and 13

Step V:



or



Lchild = 0
rchild = 1

Huffman's code

2 = 0 000
5 = 001
3 = 0001
13 = 01
7 = 10
9 = 11

Then merge 16 and 23.

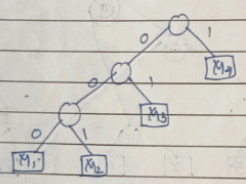
Algorithm:

```
treenode = record {
  treenode *lchild;
  treenode *rchild;
  integer weight;
};

Algorithm Tree(n)
{
  for i=1 to n-1 do
```

```
pt = new treenode;
(pt→lchild) = least(list);
(pt→rchild) = least(list);
(pt→weight) = ((pt→lchild)→weight) +
              ((pt→rchild)→weight);
Insert(list, pt);
}
return least(list);
}
```

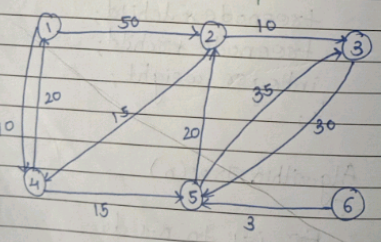
* HuffmanisCode:

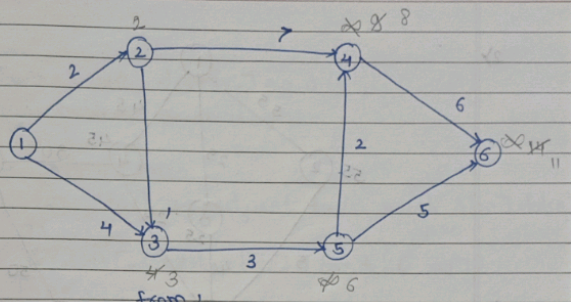
e.g.



M1 = 000 M3 = 01
M2 = 001 M4 = 1

* Single Source Shortest path:



1) 

1) if Direct edge ↑ then assign weight else assign ∞

2) Calculate ~~all~~ shortest path from all vertices

$d(w) + c(u,w) < d[w]$
$d[w] = d(u) + c(u,w)$

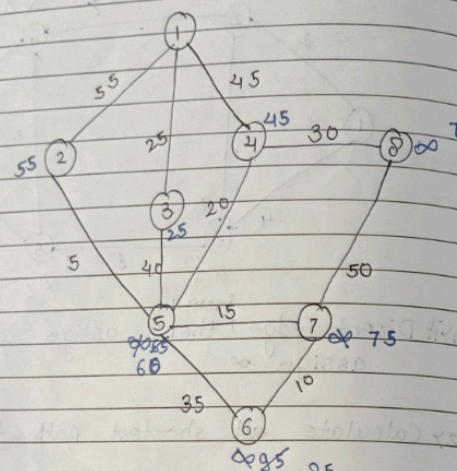
i) $d(2) + c(2,3) < d(3)$ $\quad$ $d(2) + c(2,4) < d(4)$
$2 + 1 < 4$ $\quad$ $2 + 7 < \infty$
$3 < 4$ $\quad$ $9 < \infty$

ii) $d(3) + c(3,5) < d(5)$
$3 + 3 < \infty$
$6 < \infty$

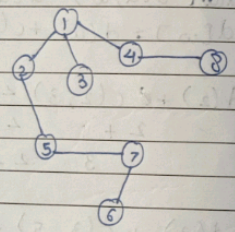
iii) $d(5) + c(5,4) < d(4)$
$6 + 2 < 9$
$8 < 9$

v) $d(5) + c(5,6) < d(6)$
$6 + 5 < \infty$
$11 < \infty$

iv) $d(4) + c(4,6) < d(6)$
$8 + 6 < \infty$
$14 < \infty$



1-2-5-7




```
shortestpaths(v, cost, dist, n)
dist[j] - set to length of shortest path
  from v to j
  dis[v]=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
  {
    s[v]=true
    for (each w adjust to v with s[w]=false)
    do
    if(dist[w]>dist[v]+cost[v,w])
    then
    d[w]=dist[u]+cost[u,w];
  }
}
```

i) $d(1,2) + d(2,5) < d(5)$
$55 + 5 < \infty$
$60 < \infty$

ii) $d(3)+c(3,5)<d(5)$
$25 + 40 < 60$
$65 \not< 60$

iii) $d(4)+c(4,5)<d(5)$
$45 + 20 < 60$
$65 \not< 60$

iv) $d(5)+c(5,6)<d(6)$
$60 + 35 < \infty$
$95 < \infty$

v) $d(5)+d(5,7)<d(7)$
$60 + 15 < \infty$
$75 < \infty$

vi) $d(7)+c(7,6)<d(6)$
$75 + 10 < 95$
$85 < 95$

vii) $d(4)+d(4,8)<d(8)$
$45 + 30 < \infty$
$75 < \infty$

viii) $d(7)+c(7,8)<d(8)$
$75 + 50 < 75$
$120 \not< 75$

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing highway. A motorist wishes to drive from city A to B would be interested in answers to following question:

1. Is there a path from A to B?
2. Is there more than one path from A to B? Which is the smallest path?

The problem define is called the shortest path problem. The length of the path is defined by sum of the weights of the edges on the path.

The starting vertex is referred as source and last vertex is referred as destination. The graphs are digraphs in the problem we consider we are given a directed $G = (V, E)$ a weighting function cost for the $h(G)$ and source vertex is 0. The problem is to determine the shortest path from ~~(or)~~ V_0 ~~to v(not)~~ to all the remaining vertices of graph. It is assume that all the weights are positive. The shortest path between V_0 and some other path v is ordering among the subset of edges.

23 August 2021

* 3. Backtracking *

Backtracking represents one of the most general technique many problems ~~with~~ idea! with searching ~~perspet~~ set of solution and which ask for an optimal solution satisfying some constraints can be solved using backtracking formulation.

In many application of backtrack method the desire solution is expressed as n-tuple x_1 to x_n where x_i are chosen for some finite set S_i. The problem to be solved falls for finding one vector that maximize or minimize a criterion function $P(x_1, \ldots x_n)$. Sometimes it seeks all vectors that ~~st~~ satisfies P.

Suppose M_i is the size of set S_i then there are $M = m_1, m_2, \ldots m_n$ n tuple that are possible candidates for satisfying function P. The root force approach could be form n tuples evaluate each one with P and save those which n optimum. The backtrack algorithm has the ability to answer with far fewer than n trials. The basic idea is to built solution vector one component at a time and to use modifying criterion function called bounding function to test whether the vectors so form has any chance of success.

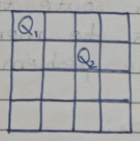
Algorithm Backtrack(k)
// x[1], x[2], ... x[k-1] of solution vector
{

```
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 path to
  answer node)
    then write x[1=k];
  if (k<n)
  then
  Backtrack(k+1)
  }
}
}
```

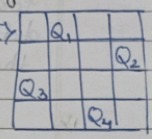
Let $T(x[1] \ldots x[i])$ the set of all possible values for $x[i+1]$ such that $x[1], x[2] \ldots x[i+1]$ is also a path to problem state. $T(x[1], x[2], \ldots x[n]) = \text{null}$ we assume the existence of bounding function P_{i+1} such that if P_{i+1} such that $P_{i+1}(x[1], x[2] \ldots x[i+1])$ is false for a path $(x[1], x[2], \ldots x[i+1])$ from root node to a problem state. Then the path cannot be extended to reach an answer node. Thus, the candidates for position $i+1$ of the solution vector $x[1]$ to $x[n]$ are those values which are generated by T and satisfy $P[i+1]$.

* N queens problem:

Q_1 Q_2 Q_3 Q_4 Not in same row, column, diagonal

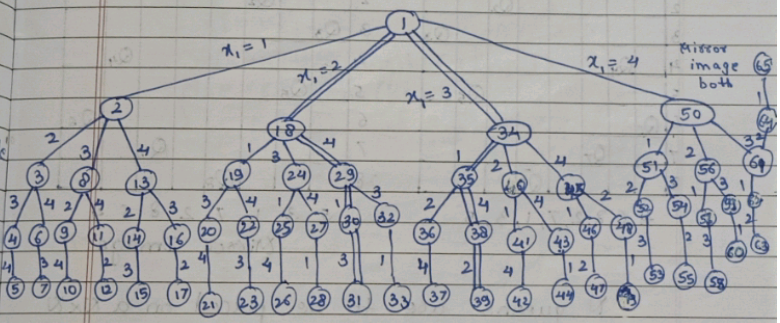
I> 

II> 

III> 

2 4 1 3
3 1 4 2

State space tree (6 marks)

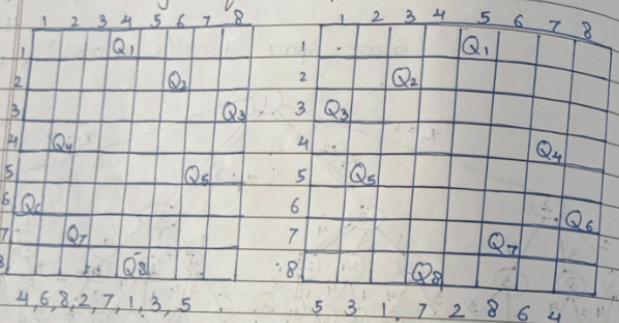


Backtracking algorithm determines problem solution by systematically searching the solution space for the given problem instant this search is faciliated by using a tree organization for the solution space. Each node in this tree determines a problem state all paths from the route to other nodes define the state space of the problem.

Solutions state are those problem states 's' for which the path from the route to 's'

defines a tuple in the solution space. Answer states are those solution states for which the path from root to 's' defines a tuple that is the member of the set of solution of the problem.

1) The Eight queens problem:



N queens are to be placed on a N×N chessboard such that no two queens are attacking each other i.e no two queens are on the same row, column or diagonal. the chessboard squares been numbered as the indices of two-dimensional array a[1:n, 1:n] then we observe that every element on the same diagonal that runs from the upper left to the lower rights has the same row-column values. Suppose two queens are to be placed on position i, j and k, l then by the above they are on the same diagonal only

if $i-j = k-l$ or $i+j = k+l$.

The first equation, implies $j-l = i-k$ and the second equation implies $j-l = k-i$

```
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);
    }
  }
}
```

* Sum of subset

n = 6 , m = 30

w[1:6] = {5, 10, 12, 13, 15, 18} {1, 0, 1, 0, 0, 1}

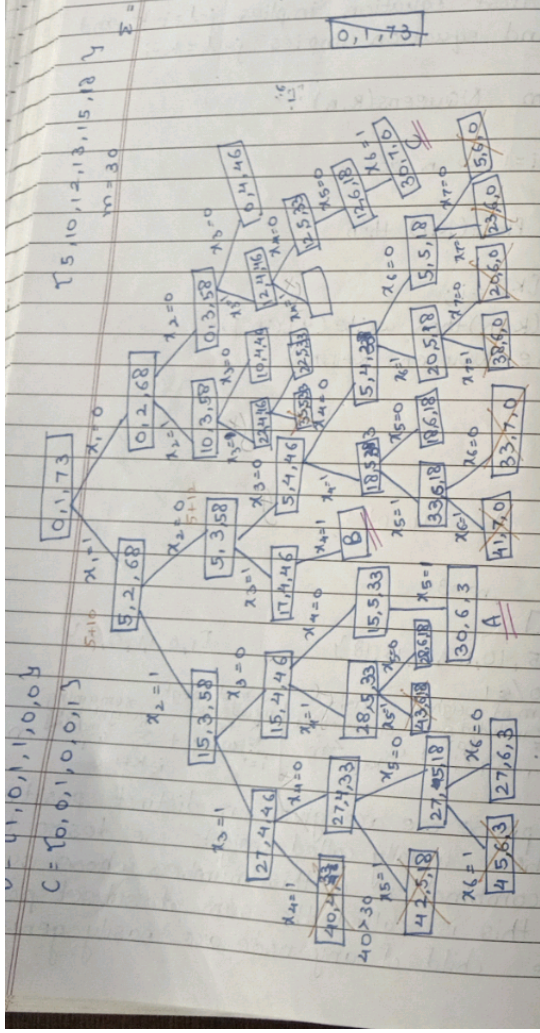
$x_i = 0/1$

Sum of weights included so far: $\sum_{i=1}^{k} w_i x_i$ + weight of next object $w_{k+1} \le m$

Sum of weight included so far: $\sum_{i=1}^{k} w_i x_i$ + remaining weight to be included: $\sum_{i=k+1}^{n} w_i \ge m$

Suppose we are given n distinct positive numbers usually called weights we desire to find all combinations of this numbers whose sums are 'm' this is called the sum of subset problem. The child of any node are easily generated

State space tree:



for a node at level i, the left child corresponds to $x_i = 1$ and the right child $x_i = 0$. The simple choice for bounding function is true if $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$.

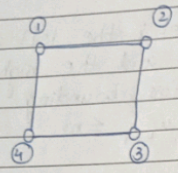
Algorithm:

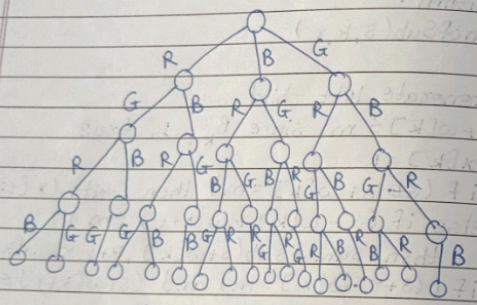
```
SumofSub(s, k, r)
{
  Generate left child
  s+w[k] ≤ m since B_{k-1} is true
  x[k] = 1
  if (s + w[k] = m) then write(x[1:k]);
  else if (s + w[k] + w[k+1] ≤ m)
    then sumofsub(s+w[k], k+1, r-w[k]);
  if ((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
  {
    x[k] = 0;
    sumofsub(s, k+1, r-w[k]);
  }
}
```

* Graph coloring:

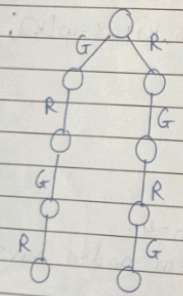
m = 3 (colors), n = 4

- No two adjacent nodes have same color (optimization)
- There are two type of questions:
  1) minimum colors required
  2) Can you color nodes with m colors.





three colors



two colors

Algorithm

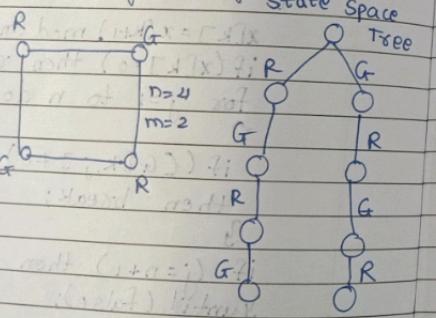
Lucky Page No.:
Date : / /

The graph is represented by boolean adjacency matrix G[1:n, 1:n]

```
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);
}

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);
}
```

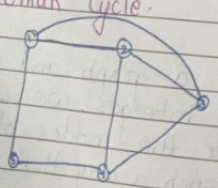
Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be coloured in such a way that no two adja

nt nodes have the same colour, yet only m colors are used. This is term mcolorability decision problem. If D is a degree of the Graph it can be coloured with d+1 colors. The mcolorability optimization problem ask for smallest integer m for which the graph G can be coloured. This integer is referred as chromatic number of the graph. Suppose we represent a graph by its adjacency matrix G[1:n, 1:n] where G[i,j] is equal to one if [i,j] is an edge of graph. Else G[i,j]=0. The colors are represented by the integers 1, 2, ... n and the solution is given by n-tuple.



State Space Tree

* Hamiltonian Cycle:

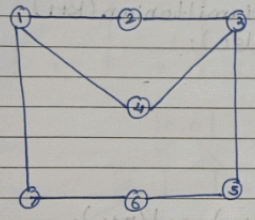


1, 2, 3, 4, 5, 1

1, 3, 2, 4, 5, 1

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

If edge present → 1 if not → 0



Let G = {V, E} be a connected graph with n vertices. A Hamiltonian Cycle is a round trip path along n edges of graph that visits every vertice only once and return to its starting point. In other words, if a Hamiltonian cycle begins at some vertex v_1 and the vertices of graph are visited in the order v_1, v_2 to $v_{(n+1)}$ then the edges v_i, v_{i+1} are in E (edges) and v_i are distinct except v_1 and v_{n+1} which are equal.

Algorithm Hamiltonian(k)
G[1:n, 1:n] is a adjacency matrix of graph G
All cycle begin at node 1

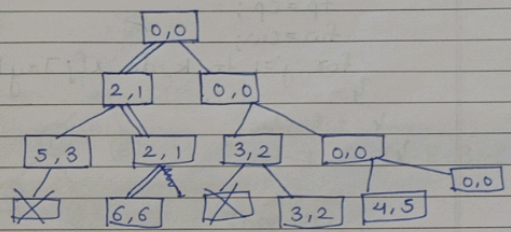
```
{
  repeat
  {
    Nextvalue(k);
    if(x[k]=0) return;
    if (k=n) then write(x[1:n]);
    else Hamiltonian(k+1);
  } until(false);
}
```

```
Nextvalue(k)
repeat
{
  x[k]=(x[k]+1) mod(n+1);
  if(x[k]=0) 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;
  }
  tot until(false);
}
```

* Knapsack problem:

n positive weights w_i and n positive profits p_i and a positive number m that is knapsack capacity the problems call for choosing a subset of weights such that $\sum_{1\le i\le n} w_i x_i \le m$ and $\sum_{1\le i\le n} p_i x_i$ is maximized $i \le i \le n$

Weight	2	3	4
Profit	1	2	5




```
Algorithm BKnap(k,cp,cw)
{
  Generate left child
  if(cw+w[k] ≤ m) then
  {
    y[k]=1
  if(k<n) then BKnap(k+1, cp+p[k], cw+w[k]);
  if((cp+p[k] > fp) and (k=n)) then
  {
    fp = cp+p[k];
    fw = cw+w[k];
    for j=1 to k do
```

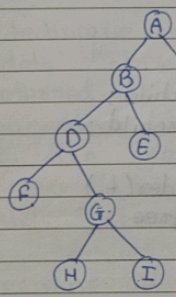
```
        x[j] = y[j];
    }
// Generate right child
if (Bound(cp, cw, k) > fp) then
{
    y[k] = 0;
    if (k < n) then BKnap(k+1, cp, cw)
    if (cp > fp) and (k = n) then
    {
        fp = cp;
        fw = cw;
        for j := 1 to k do x[j] = y[j];
    }
}
}
```

25th September 2024

* 4. Basic Traversal & Search Techniques *

* Techniques for Binary Tree :

1. Inorder (L-D-R)
2. Preorder (D-L-R)
3. Postorder (L-R-D)



Postorder:
F - H - I - G - D - E - B - C - A

Inorder:
F - D - H - G - I - B - E - A - C

Preorder:
A - B - D - F - G - H - I - E - C

- Algorithm for InOrder :

```
tree_node = record
{
    Type data;
    treenode * lchild;
    treenode * rchild;
}
Algorithm InOrder(t)
    t is binary tree
{
```

```
if t≠0 then
  InOrder(t→lchild);
  visit(t);
  InOrder(t→rchild);
}
}
```

- Algorithm for preOrder:

```
treenode = record
{
  Type data;
  treenode *lchild;
  treenode *rchild;
}
Algorithm PreOrder(t)
t is binary tree
{
  if t≠0 then
    visit(t);
    Preorder(t→lchild);
    Preorder(t→rchild);
}
```

- Algorithm for postOrder:

```
treenode = record
{
  Type data;
  treenode *lchild;
  treenode *rchild;
}
Algorithm PostOrder(t)
t is binary tree
{
  if t≠0 then
    Postorder(t→lchild);
    PostOrder(t→rchild);
    visit(t);
}
```

The technique in which every node in the given data object instance is examine is refered as traversal method. The technique that does not examine all vertices are refferred as search method.

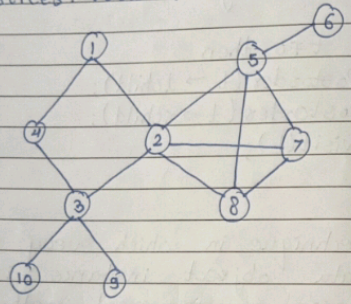
* Reachability problem:

To determine if there exist a path in given Graph G={V, E} such that this path starts at vertex v and ends at vertex u.

* Breadth first search

In breadth first search we start at a vertex v and mark it as having v visited. The vertex v at the first time is said to be unexplored. A vertex is said to have been explored when all vertices adjacent to it are visited.

All unvisited vertices adjacent from v are visited next these are new unexplored vertices. Vertex V has now been explored.



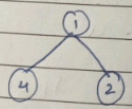
1) Start by exploring vertex 1, now add it to the queue.

1				

(1)

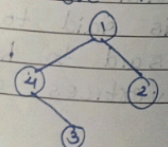
2) Now explore its adjacent vertices and add it to the queue and remove one from the queue, as all its adjacent vertices are visited.

~~1~~	4	2		



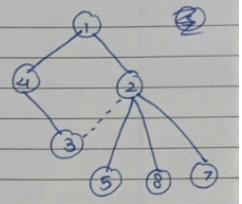
3) After all adjacent vertices are explored select next vertex for exploration from queue that is four. Explore adjacent vertices to 4 & remove 4 from the queue.

~~1~~	~~4~~	2	3	



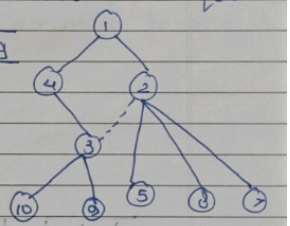
4) After all adjacent vertices are explored select next vertex for exploration from queue that is two. Explore adjacent vertices to 2 and remove 2 from the queue

~~1~~	~~4~~	~~2~~	3	5	7	8	



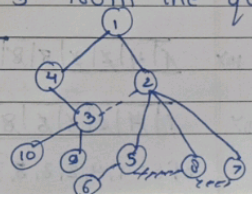
5) After all adjacent vertices are explore select next vertex for exploration from queue that is 3. Explore adjacent vertices to 3 and remove 3 from the queue.

~~1~~	~~4~~	~~2~~	~~3~~	5	8	7	10	9

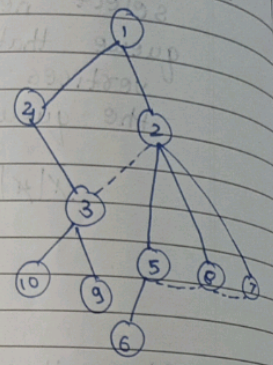


6) After all adjacent vertices are explore select next vertex for exploration from queue that is 5. Explore adjacent vertices to 5 and remove 5 from the queue.

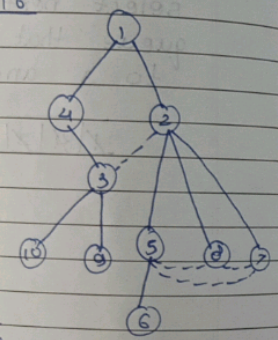
~~1~~	~~4~~	~~2~~	~~3~~	~~5~~	8	7	10	9	6



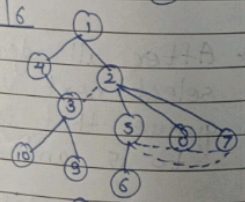
7) 1 4 2 8 8 8 7 10 9 6



8) 1 4 2 8 8 8 7 10 9 6

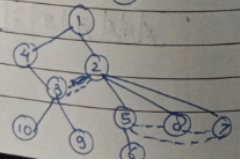


9) 1 4 2 8 8 8 7 10 9 6



10) 1 4 2 3 8 8 7 10 8 6

11) 1 4 2 3 5 8 7 10 9 8



BFS: 1, 4, 2, 3, 5, 8, 7, 10, 9, 6

Algorithm BFS(v)
{
u = v;
visited[v] = 1;
repeat
{
for all vertices w adjacent from u do
{
if (visited[w] = 0) then
{
Add w to q;
visited[w] = 1;
}
}
if q is empty then return;
Delete the next element u from q
} until(false);
}

Algorithm BFT(G, n)
{
for i = 1 to n do
visited[i] = 0;
for i = 1 to n do
if (visited[i] = 0)
then
BFS(i);
}

Note:

1) In BFS you can start at any vertex
2) While exploring adjacent vertex you can

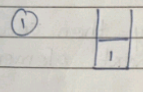
explore in any order.

3) The main rule when you select any vertex for exploration you must visit all its adjacent vertices then, only go to next vertex for exploration.

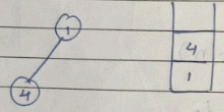
4) Select next vertex for exploration from the queue only.

* Dept first search: Same graph as in BFS.

1) Start at vertex 1 explore its adjacent that is 4, as 4 has adjacent node 4 is suspended and added to the stack.

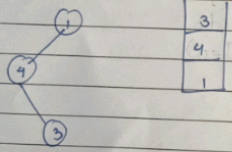


2)



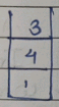
Now suspend 1 and go to 4.

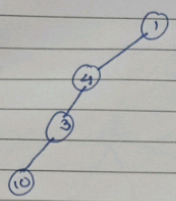
3)



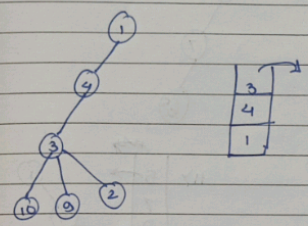
Now suspend 4 and go to 3.

4) Now suspend 3 and goto 10.



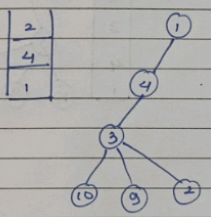


5)

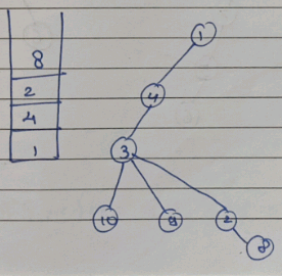


Now pop 3.

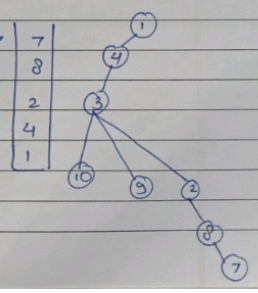
6)

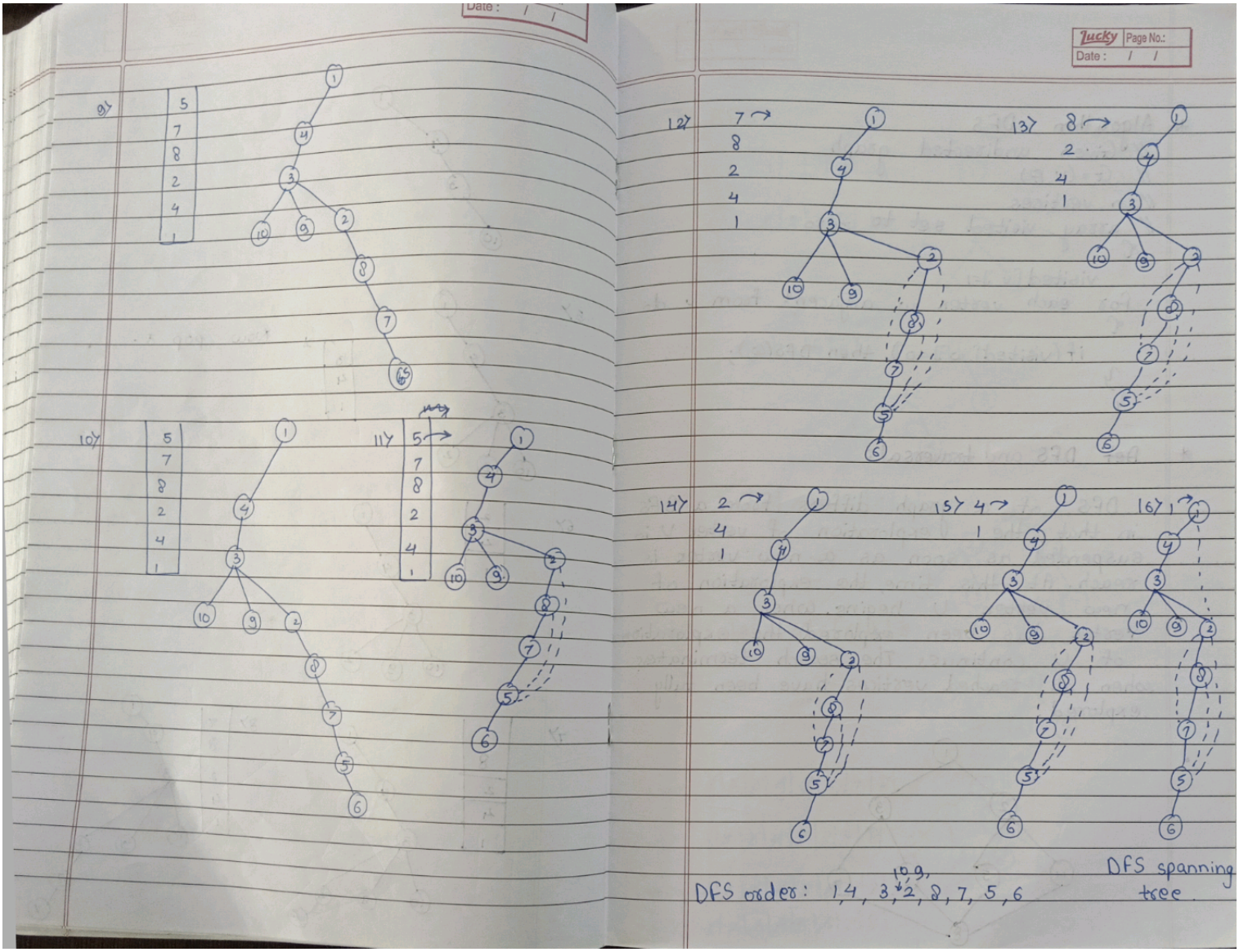


7)



8)

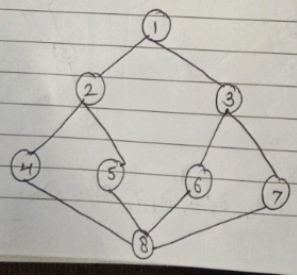



9)
5
7
8
2
4
1
10)
11)
12)
13)
14)
15)
16)
DFS order: 1, 4, 3, 10, 9, 2, 8, 7, 5, 6
DFS spanning tree

* Algorithm DFS
// Given undirected graph
// G=(V,E)
// n vertices
// array visited set to zero
{
visited[v]=1
for each vertex w adjacent from v do
{
if(visited[w]=0) then DFS(w);
}
}

* Def DFS and traversal:

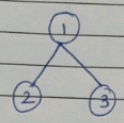
DFS of a graph differs from a BFS in that the exploration of vertex V is suspended as soon as a new vertex is reach. At this time, the exploration of new vertex U begins. When a new vertex has been explored the exploration of U continues. The search terminates when all reached vertices have been fully explored.



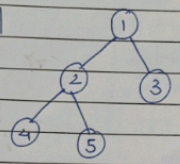
i) BFS

i) 1 | | | |

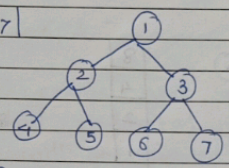
ii) ✗ | 2 | 3 | | |



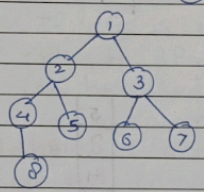
iii) ✗ | ✗ | 3 | 4 | 5 |



iv) ✗ | ✗ | ✗ | 4 | 5 | 6 | 7 |



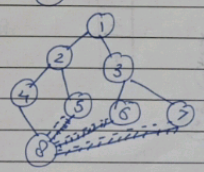
v) ✗ | ✗ | ✗ | ✗ | 5 | 6 | 7 | 8



vi) ✗ | ✗ | ✗ | ✗ | ✗ | 6 | 7 | 8 |

vii) ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | 7 | 8 |

viii) ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | 8 |



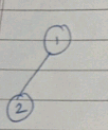
ix) ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

BFS: 1,2,3,4,5,6,7,8

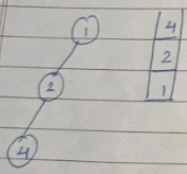
2) DFS

i)

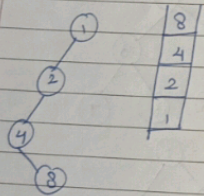
ii)



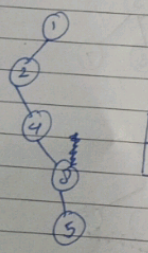
3)



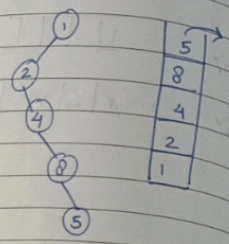
4)



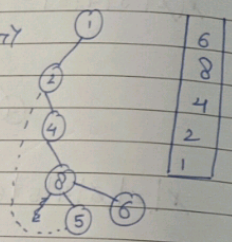
5)



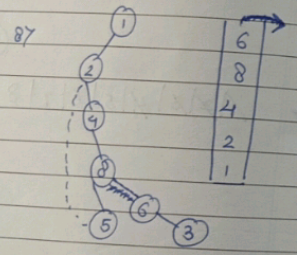
6)



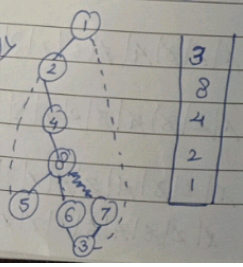
7)



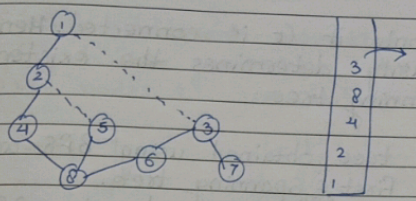
8)



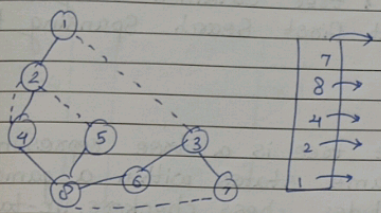
9)



10)



11)



DFS: 1, 2, 4, 8, 5, 6, 3, 7

* Connected components and spanning trees:

If G is connected undirected graph then all vertices of G will get visited on the first call to BFS. If G is not connected then at least two calls to BFS will be needed. Hence, BFS can be used to determine whether G is connected or not. A newly visited vertex on a call to BFS and from BFT represent the vertices in a connected component of G hence, the connected components of Graph can be obtained using BFT. If adjacency list are used BFS Traversal will obtain connected component in time O(n+e)

The graph G has a spanning tree

if and only if G is connected. Hence, DFS easily determines the existence of the spanning tree.

Spanning tree obtained using BFS are called Breadth first spanning tree.
A Spanning tree obtained using DFS are called Depth first Search Spanning tree.

* Game tree:

A game tree is a tree representing all possible game states within a game. Such game includes chess, checkers, tic-tac-toe, etc. This can be used to measure the complexity of a game as it represents all possible ways a game can pan out. A game tree is a directed graph whose nodes are position in a game and edges are moves. A complete game tree for a game starting at initial position and contains all possible moves for each position. The diagram shows the game tree for tic-tac-toe the rotations and reflections of positions are equivalent. So first player has three choices of moves in center at edge or in corner. The second player has two choices for the reply, if center otherwise the 1st player insert the and five choices & so on. The no. of leaf nodes in the complete game tree is the no. of possible different ways the game can be played. The game tree for tic-tac-toe has 255 nodes & 168 leaf nodes.

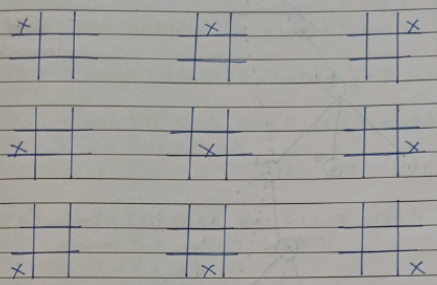
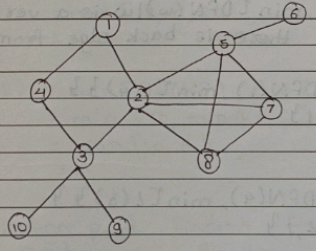
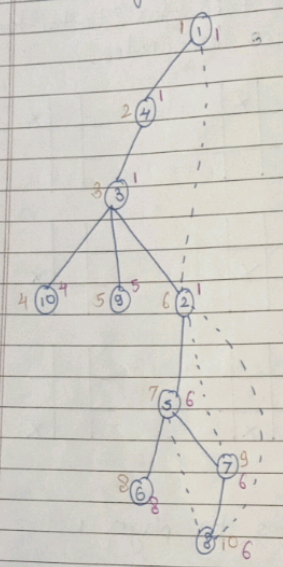


fig. Possible moves for player 1

* Bi-connected components and DFS:



DFT of given graph :-



- DFN
- lowest

$L(u) = \min\{DFN(u), \min\{L(w) \mid w \text{ is child of } u\}, \min\{DFN(w) \mid w \text{ is a vertex to which there is back edge from } u\}\}$

1) $L(1) = \min\{DFN(1), \min\{L(4)\}\}$
$= \min\{1, 1\}$
$= 1$

2) $L(4) = \min\{DFN(4), \min\{L(3)\}\}$
$= \min\{2, 1\}$
$= 1$

3) $L(3) = \min\{DFN(3), \min\{L(10), L(9), L(2)\}\}$
$= \min\{3, \cancel{4, 5,} 1\} = \min\{3, 1\}$
$= 1$

4) $L(10) = \min\{DFN(10)\} = \min\{4\} = 4.$

5) $L(9) = \min\{DFN(9)\} = \min\{5\} = 5$

6) $L(2) = \min\{DFN(2), \min\{L(5)\}, \min\{DFN(1)\}\}$
$= \min\{6, 6, 1\}$
$= 1$

7) $L(5) = \min\{DFN(5), \min\{L(6), L(7)\}\}$
$= \min\{7, \min\{8, 6\}\}.$
$= \min\{7, 6\}$
$= 6.$

8) $L(6) = \min\{DFN(6)\} = \min\{8\} = 8$

9) $L(7) = \min\{DFN(7), \min\{L(8)\}, \min\{DFN(2)\}\}$
$= \min\{9, 6, 6\}$
$= 6$

10) $L(8) = \min\{DFN(8), \min\{DFN(5), DFN(2)\}\}$
$= \min\{10, \min\{7, 6\}\}$
$= \min\{10, 6\}$
$= 6$

Articulation point finding formula :

Child		Parent	if true articulation point
$L[v]$	$\geq$	$dfn[u]$	

e.g. $L[3] \geq dfn[4]$
$\times \quad 1 \geq 2$

Not apply to root node, apply if root node have 2 children

1) $L[4] \geq dfn[1]$ ✓ $1 \geq 1$ ~~$L[3] \geq dfn[4]$~~ ✗ $1 \geq 2$

2) $L[10] \geq dfn[3]$ ✓ $4 \geq 3$ $L[9] \geq dfn[3]$ ✓ $5 \geq 3$

$L[2] \geq dfn[3]$ ✗ $1 \geq 3$ Yes

3) $L[5] \geq dfn[2]$ ✓ $6 \geq 6$ Yes

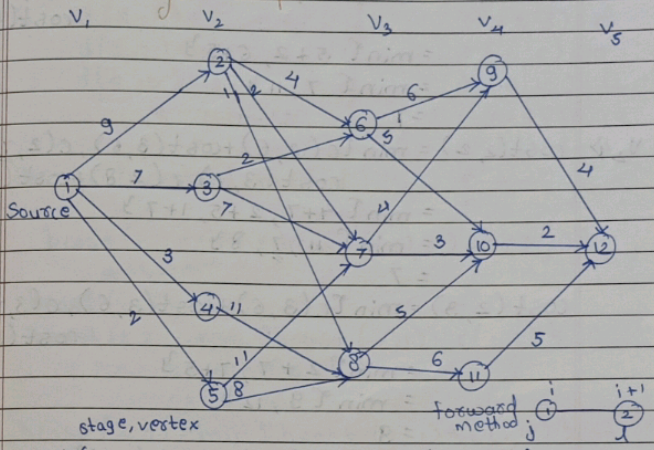
4) $L[6] \geq dfn[5]$ ✓ $8 \geq 7$ $L[7] \geq dfn[5]$ ✗ $6 \geq 7$ Yes

5) $L[8] \geq dfn[7]$ ✗ $6 \geq 9$

8 October 2024

* 5. Dynamic Programming

* Multistage Graph :



$cost(i,j) = \min\{c(j,d) + cost(i+1,d)\}$

- Using forward method :

V_5 1) $cost(5,12) = 0$

V_4 2) $cost(4,9) = 4$

$cost(4,10) = 2$

$cost(4,11) = 5$

V_3 3) $cost(3,6) = \min\{c(6,9) + cost(4,9), c(6,10) + cost(4,10)\}$

$= \min\{6+4, 5+2\}$

$= \min\{10, 7\}$

$= 7$

$cost(3,7) = \min\{c(7,9) + cost(4,9), c(7,10) + cost(4,10)\}$

$= \min\{4+4, 3+2\}$

$= \min\{8, 5\}$

$= 5$

Date : / /

$cost(3,8) = \min\{c(8,10) + cost(4,10), c(8,11) + cost(4,11)\}$

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

$= \min\{7, 11\}$

$= 7$

V_2) $cost(2,2) = \min\{c(2,6) + cost(3,6), c(2,7) + cost(3,7), c(2,8) + cost(3,8)\}$

$= \min\{4+7, 2+5, 1+7\}$

$= \min\{11, 7, 8\}$

$= 7$

$cost(2,3) = \min\{c(3,6) + cost(3,6), c(3,7) + cost(3,7)\}$

$= \min\{2+7, 7+5\}$

$= \min\{9, 12\}$

$= 9$

$cost(2,4) = \min\{c(4,8) + cost(3,8)\}$

$= \min\{11+7\}$

$= 18$

$cost(2,5) = \min\{c(5,7) + cost(3,7), c(5,8) + cost(3,8)\}$

$= \min\{11+5, 8+7\}$

$= \min\{16, 15\}$

$= 15$

V_1) $cost(1,1) = \min\{c(1,2) + cost(2,2), c(1,3) + cost(2,3), c(1,4) + cost(2,4), c(1,5) + cost(2,5)\}$

$= \min\{9+7, 7+9, 3+18, 2+15\}$

$= \min\{16, 16, 21, 17\}$

Shortest path value = 16 (I, II)

Lucky Page No.: Date : / /

I) stage, vertex

$d(1,1) = 2$

$d(2,2) = 7$

$d(3,7) = 10$

$d(4,10) = 12$

1 - 2 - 7 - 10 - 12

II) stage, vertex

$d(1,1) = 3$

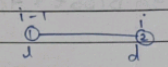
$d(2,3) = 6$

$d(3,6) = 10$

$d(4,10) = 12$

1 - 3 - 6 - 10 - 12

- Using backward method :



$bcost(i,j) = \min\{bcost(i-1, l) + c(l,j)\}$ (stage, vertex)

V_1) $bcost(1,1) = 0$

V_2) $bcost(2,2) = \min\{bcost(1,1) + c(1,2)\}$

$= \min\{0+9\}$

$= 9$

3) $bcost(2,3) = \min\{bcost(1,1) + c(1,3)\}$

$= \min\{0+7\}$

$= 7$

4) $bcost(2,4) = \min\{bcost(1,1) + c(1,4)\}$

$= \min\{0+3\}$

$= 3$

5) $bcost(2,5) = \min\{bcost(1,1) + c(1,5)\}$

$= \min\{0+2\}$

$= 2$

V_3 6) $bcost(3,6) = \min\{bcost(2,2) + c(2,6), bcost(2,3) + c(3,6)\}$

$= \min\{9+4, 7+2\}$

$= \min\{13, 9\}$

$= 9$

7) $bcost(3,7) = \min\{bcost(2,2) + c(2,7), bcost(2,3) + c(3,7), bcost(2,5) + c(5,7)\}$

$= \min\{9+2, 7+7, 2+11\}$

$= \min\{11, 14, 13\}$

$= 11$

8) $bcost(3,8) = \min\{bcost(3,2)+c(2,8), bcost(3,4)+c(4,8), bcost(3,5)+c(5,8)\}$
$= \min\{9+1, 3+11, 2+8\}$
$= \min\{10, 14, 10\}$
$= 10$

v_4 9) $bcost(4,9) = \min\{bcost(3,6)+c(6,9), bcost(3,7)+c(7,9)\}$
$= \min\{9+6, 11+4\}$
$= \min\{15, 15\}$
$= 15$

10) $bcost(4,10) = \min\{bcost(3,6)+c(6,10), bcost(3,7)+c(7,10), bcost(3,8)+c(8,10)\}$
$= \min\{9+5, 11+3, 10+5\}$
$= \min\{14, 14, 15\}$
$= 14$

11) $bcost(4,11) = \min\{bcost(3,8)+c(8,11)\}$
$= \min\{10+6\}$
$= 16$

v_5 12) $bcost(5,12) = \min\{bcost(4,9)+c(9,12), bcost(4,10)+c(10,12), bcost(4,11)+c(11,12)\}$
$= \min\{15+4, 14+2, 16+5\}$
$= \min\{19, 16, 21\}$

Shortest Path = 16

stage, vertex	min value
$d(5,12) = 10$	$d(5,2) = 10$
$d(4,10) = 6$	$d(4,10) = 7$
$d(3,6) = 3$	$d(3,7) = 2$
$d(2,3) = 1$	$d(2,2) = 1$
1-3-6-10-12	1-2-7-10-12

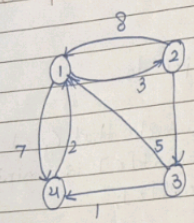
- Algorithm FGraph(G, K, n, p)

```
{
  cost[n] = 0.0;
  for j = n-1 to 1 step-1 do
  {
    let r be a vertex such that (j,r) is a edge
    of G and c[j,r] + cost[r] is minimum.
    cost[j] = c[j,r] + cost[r];
    d[j] = r;
  }
  p[1] = 1;
  p[k] = n
  for j = 2 to k-1 do
    p[j] = d[p[j-1]];
}
```

- Algorith BGraph(G, K, n, p)

```
{
  bcost[1] = 0.0;
  for j = 2 to n
  {
    let r be a vertex such that (j,r) is a
    edge of G and c[j,r] + cost[r] is minimum
    cost[j] = c[j,r] + bcost[r];
    d[j] = r;
  }
  p[1] = 1;
  p[k] = n;
  for j = k-1 to 2 do
  p[j] = d[p[j+1]];
}
```

* All pair shortest path:



$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array}$$

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

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & \infty & 0 \end{array}$$

$A^1[2,3] = \min\{A^0[2,3], A^0[2,1] + A^0[1,3]\}$
$= \min\{2, 8+\infty\}$
$= 2$

$A^1[2,4] = \min\{A^0[2,4], A^0[2,1] + A^0[1,4]\}$
$= \min\{\infty, 8+7\}$
$= 15$

$A^1[3,2] = \min\{A^0[3,2], A^0[3,1] + A^0[1,2]\}$
$= \min\{\infty, 5+3\}$
$= 8$

$A^1[3,4] = \min\{A^0[3,4], A^0[3,1] + A^0[1,4]\}$
$= \min\{1, 5+7\}$
$= \min\{1, 12\}$
$= 1$

$A^1[4,2] = \min\{A^0[4,2], A^0[4,1] + A^0[1,2]\}$
$= \min\{\infty, 2+3\}$
$= 5$

$A^1[4,3] = \min\{A^0[4,3], A^0[4,1] + A^0[1,3]\}$
$= \min\{\infty, 2+\infty\}$
$= \infty$

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 5 & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

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

$A^2[1,4] = \min\{A^1[1,4], A^1[1,2] + A^1[2,4]\}$
$= \min\{7, 3+15\}$
$= 7$

$A^2[3,1] = \min\{A^1[3,1], A^1[3,2] + A^1[2,1]\}$
$= \min\{5, 8+8\}$
$= 5$

$A^2[3,3] = \min\{A^1[3,3], A^1[3,2] + A^1[2,3]\}$
$= \min\{0, 8+2\}$
$= 0$

$A^2[3,4] = \min\{A^1[3,4], A^1[3,2] + A^1[2,4]\}$
$= \min\{1, 8+15\}$
$= 1$

$A^2[4,1] = \min\{A^1[4,1], A^1[4,2] + A^1[2,1]\}$
$= \min\{2, 5+8\}$
$= 2$

$A^2[4,3] = \min\{A^1[4,3], A^1[4,2] + A^1[2,3]\}$
$= \min\{\infty, 5+2\}$
$= 7$

$A^3 =$

	1	2	3	4
1	0	3	5	6
2	7	0	2	3
3	5	8	0	1
4	2	5	7	0

$A^3[1,2] = \min\{A^2[1,2], A^2[1,3] + A^2[3,2]\}$
$= \min\{3, 5+8\}$
$= 3$

$A^3[1,4] = \min\{A^2[1,4], A^2[1,3] + A^2[3,4]\}$
$= \min\{7, 5+1\}$
$= 6$

$A^3[2,1] = \min\{A^2[2,1], A^2[2,3] + A^2[3,1]\}$
$= \min\{8, 2+5\}$
$= 7$

$A^3[2,4] = \min\{A^2[2,4], A^2[2,3] + A^2[3,4]\}$
$= \min\{15, 2+1\}$
$= 3$

$A^3[4,1] = \min\{A^2[4,1], A^2[4,3] + A^2[3,1]\}$
$= \min\{2, 7+5\}$
$= 2$

$A^3[4,2] = \min\{A^2[4,2], A^2[4,3] + A^2[3,2]\}$
$= \min\{5, 7+8\}$
$= 5$

$A^4 =$

	1	2	3	4
1	0	3	5	6
2	5	0	2	3
3	3	8	0	1
4	2	5	7	0

$A^4[1,2] = \min\{A^3[1,2], A^3[1,4] + A^3[4,2]\}$
$= \min\{3, 5+8\}$
$= 3$

$A^4[1,3] = \min\{A^3[1,3], A^3[1,4] + A^3[4,3]\}$
$= \min\{5, 6+7\}$
$= 5$

$A^4[2,1] = \min\{A^3[2,1], A^3[2,4] + A^3[4,1]\}$
$= \min\{7, 3+2\}$
$= 5$

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

$A^4[3,1] = \min\{A^3[3,1], A^3[3,4] + A^3[4,1]\}$
$= \min\{5, 1+2\}$
$= 3$

$A^4[3,2] = \min\{A^3[3,2], A^3[3,4] + A^3[4,2]\}$
$= \min\{8, 1+8\}$

= 86

Algorithm Allpath(cost, A, n)
// cost[1:n, 1:n] is cost adjacency matrix
// cost[i, j] = 0.0
{
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])

* 0/1 Knapsack:

A solution to the knapsack problem can be obtained by making a sequence of decision on the variables $x_1, x_2 \ldots x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assign to it. let us assume that decision on the x_i are made in the order x_n, x_{n-1} to x_1. Following a decision on x_n be may be in one of the two possible states. i) The capacity remaining in the knapsack is m and no profit has occurred.

ii) The capacity remaining is ~~m~~ $m - w_n$ and profit of p_n has occurred. It is clear that the remaining decisions x_{n-1}, x_1 must be optimal with respect to the problem state resulting from the decisions on x_n otherwise x_n to x_1 will not be optimal.

Q.1 $n=3$ $(w_1, w_2, w_3) = (2, 3, 4)$
$m=6$ $(p_1, p_2, p_3) = (1, 2, 5)$

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

$S_1^1 = \{(2,3), (3,5)\}$
$S^2 = S^1 \cup S_1^1 = \{(0,0), (1,2), (2,3), (3,5)\}$ m=6 only 1 is remained.

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

$S^3 = S^2 \cup S_1^2$
$= \{(0,0), (1,2), (2,3), (5,4), (6,6)\}$

$x_3 = 1$ S^1 S^2 S^3 (6,6)
$(6-5, 6-4)$ 3rd object (6,6) − 5, 4
$= (1, 2)$

$x_2 = 0$ → S^1 S^2 S^3 (1,2)
$x_1 = 1$

Q.2 $n=4$ $(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$
$m=8$ $(P_1, P_2, P_3, P_4) = (1, 2, 5, 6)$

→ $S^0 = \{(0,0)\}$

$S_0^1 = \{(1,2)\}$

$S^1 = S^0 \cup S_0^1$

$= \{(0,0), (1,2)\}$

$S_1^1 = \{(2,3), (3,5)\}$

$S_2^2 = S^1 \cup S_1^1$

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

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

कमी weight ज्यादा profit

$S^3 = S^2 \cup S_1^2$

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

$S_{+1}^3 = \{(6,5), (7,7), (8,8), (9,10), (11,14), (12,11), (13,12), (14,14)\}$

$S^4 = S^3 \cup S_1^3$

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

$x^4 = 1$

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

$x^3 = 0$

$x^2 = 1 \qquad (2,3) - (2,3) = (0,0)$

$x^1 = 0$

Solution: $(0,1,0,1)$

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

but $(8,8) \notin S^3 \quad \therefore x_4 = 1$

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

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

but $(2,3) \in S^2 \quad x_3 = 0$

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

but $(2,3) \notin S^1 \quad \therefore x_2 = 1$

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

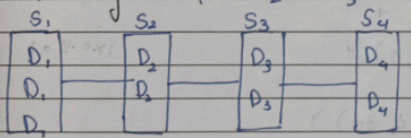
$(0,0) \in S^1$ and $\therefore x_1 = 0$

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

* Reliability Design :-

Reliability design problem is to design a system that compose of several devices connected in series. let r_i be the reliability of device d_i. Hence That is r_i is the probability that the device i will function properly.

Device	D_1	D_2	D_3	D_4
Cost	C_1	C_2	C_3	C_4
Reliability	r_1	r_2	r_3	r_4



In each stage one copy is working others are for backup. Probability of working is 0.9. Therefore $r = 0.9$. Probability of not working is $1-r$ i.e. $= 0.1$. Suppose we have 3 copies. Probability of not working will be $(1-r)^3 = (1-0.9)^3 = (0.1)^3 = 0.001$.

Probability of 3 copies of working will be $1-(1-r)^3$.

$C = 105$ → upper bound on no. of copies.

Q.

D_i	C_i	r_i	v_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

→

$C = 105$

$\sum C_i = 30 + 15 + 20 = 65$

remaining

$105 - \sum C_i$

$105 - 65 = 40$ → at least one copy is mandatory.

$C - \sum C_i$: $40 + 1 = 1 + 1 = 2$

C_i : 30

$40 + 1 = 2 + 1 = 3$

15

S_i → Device

Copy : $\frac{40}{20} + 1 = 3$

$S^0 = \{(1, 0)\}$ $\quad 1 \times 0.9 = 1$

$S_1' = \{(0.9, 30)\}$

$S_2' = \{(0.99, 60)\}$ $\quad 1-(1-r)^2$, $1-(1-0.9)^2 = 0.99$

$S' = S_1' \cup S_2' = \{(0.9, 30), (0.99, 60)\}$ $\quad 30 + 30 = 60$, $0 + 60 = 60$

S_1 = Consider D_2

$C = 15$, $r = 0.8$

$S_1^2 = \{(0.72, 45), (0.792, 75)\}$ $\quad +15$, $+15$

$S_2^2 = \{(0.864, 60), (0.9504, 90)\}$ $\quad +15$

$S_3^2 = \{(0.8928, 75), (0.9821, 105)\}$ $\quad 1-(1-r)^3$, $1-(1-0.8)^3 = 0.992$, $C = 30$, $15+15=30$

$S^2 = S_1^2 \cup S_2^2 \cup S_3^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75)\}$ $\quad 1-(1-0.9)^2 = 0.99$

$S_1^3 = \{(0.36, 65), (0.4320, 80), (0.4464, 95)\}$ $\quad \times 0.5$, $+20$; $1-(1-0.5)^2 = 0.75$

$S_2^3 = \{(0.27, 0.54, 85), (0.648, 100), (0.8096, 115)\}$ $\quad 1-(1-0.5)^3 = 0.875$

$S_3^3 = \{(0.63, 105), (0.756, 120), (0.7812, 135)\}$

$S^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95), (0.54, 85), (0.648, 100), (0.63, 105)\}$

Device copies:

$D_1 = 1$

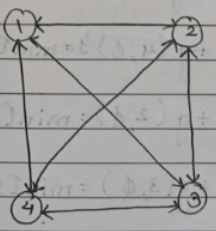
$D_2 = 2$

$D_3 = 2$

0.64 > 0.63

0.72 × 0.5 = 0.360

* Travelling Salesperson problem:



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Probability of 3 copies of working will be $1-(1-\gamma)^3$.

$C = 105$ → upper bound on no. of copies.

Q.

D_i	C_i	γ_i	v_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

→

$C = 105$

$\sum C_i = 30 + 15 + 20 = 65$

remaining

$105 - \sum C_i$

$105 - 65 = 40$ (at least one copy is mandatory)

$C - \sum C_i$: $\frac{40}{30} + 1 = 1 + 1 = 2$

$\frac{40}{15} + 1 = 2 + 1 = 3$

S_1' → Device

Copy: $\frac{40}{20} + 1 = 3$

$S^0 = \{(1, 0)\}$ (×, +) $1 \times 0.9 = 1$

$S_1' = \{(0.9, 30)\}$

$S_2' = \{(0.99, 60)\}$ $1-(1-\gamma)^2$, $1-(1-0.9)^2 = 0.99$

$30 + 30 = 60$, $0 + 60 = 60$

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

S_1 = Consider D_2

$C = 15$, $\gamma = 0.8$

$S_1^2 = \{(0.72, 45), (0.792, 75)\}$ (+15, +15)

$S_2^2 = \{(0.864, 60), (0.9504, 90)\}$ (+15)

$S_3^2 = \{(0.8928, 75), (0.9801, 105)\}$

$1-(1-0.8)^2 = 0.96$, $C = 30$, $15 + 15 = 30$

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

$S^2 = S_1^2 \cup S_2^2 \cup S_3^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75)\}$

$S_1^3 = \{(0.36, 65), (0.4320, 80), (0.4464, 95)\}$ (×0.5, +20)

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

$S_2^3 = \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$

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

$S_3^3 = \{(0.63, 105), (0.756, 120), (0.7812, 135)\}$

$S^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95), (0.54, 85), (0.648, 100), (0.63, 105)\}$

Device copies:

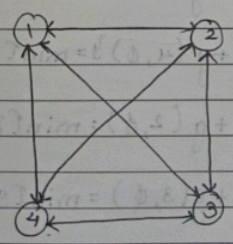
$D_1 = 1$

$D_2 = 2$

$D_3 = 2$

0.64 > 0.63

* Travelling Salesperson problem:



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$g(i, V-\{i\}) = \min_{2 \le k \le n} \{C_{ik} + g(j, S-\{j\})\}$

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

$g(i, \phi) = C_{i1}$

$g(i, \phi) = C_{i1}$

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

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

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

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

$|S| = 1 \quad i \neq 1 \quad 1 \in S : i \in S$

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

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

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

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

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

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

Now we compute for

$|S| = 2 \quad i \neq 1 \quad 1 \notin S \quad i \notin S$

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

$= \min\{9+20, 10+15\}$

$= \min\{29, 25\}$

$= 25$

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

$= \min\{13+18, 12+13\}$

$= \min\{31, 25\}$

$= 25$

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

$= \min\{18+15, 9+18\}$

$= \min\{28, 27\}$

$= 23$

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

$= \min\{35, 40, 43\}$

$= 35.$