15CSE211- Design & Analysis of Algorithms

Goodrich - "Algorithm design and applications" 7th Edition

Coremen - Introduction to Algorithms

Dasgupta - Algorithms

Experimental setup

↳ plot of size vs time

Limitations :-

1. Need to implement
2. Depends on hardware/software
3. Only a certain range of inputs

Analytical framework Components :-

1. A language which is used for implementing the algorithm (High-level description language)
2. A computational model which is used for characterising (execution) of an algorithm
3. A metric which is used for measuring running time
4. (to check) Assumptions which is used for characterizing the running time (recursive also)

Find out the max value of an array.

Algorithm : Max (A, n)

INPUT : An Array containing n integer elements $n \geq 1$

OUTPUT : Maximum element in array

```
Step1 : currMax ← A[0]
    2 : for i ← 1 to n-1 do
            if (currMax < A[i]) then
                currMax ← A[i]
```

3. return currMax

Pseudo code components:-

1. Expression
2. method declaration
3. for loop / repeat until
4. decision making blocks (if, then)
5. Array indexing
6. Object reference
7. return method.

RAM model

Random Access machine model

primitive operations are defined to calculate the actual running time of an algorithm in RAM model. Goal is to count the no of primitive operations.

Primitive operations

1. Calling a method
2. Array indexing
3. Assigning values to a variable
4. Any arthimetic operations
5. Comparision
6. returning a value
7. Object reference.

Algorithm can be analyzed in three ways:-

1. Best case
2. Worst case
3. Average case

Algorithm RecMax(A, n):

Input: The Array having n integers $n \geq 1$

Output: The maximum element of Array

```
if n = 1 then
    return A[n-1]
return Max( recurMax(A, n-1), A[n-1] )
```

The no. of primitive operations is $7(n-1)+3 = 7n-4$

$$T(n) = \begin{cases} 3 & \text{if } n=1 \\ T(n-1)+7 & \forall n \geq 2 \end{cases}$$

Write a procedure for finding sum of natural numbers

```
procedure: SumN(A,n)
Input: Value of n, Array
Output: Sum of n natural numbers
    sum ← 0                                   1+1+n+4(n-1)+1
    for i ← 0 to n                            3+n+4n-4
        sum = sum + A[0]                      6n-1
    return sum
```

Write procedure linear search.

```
(ii) for i = 0 to n do          n+1+3(n-1)      1+n+1+n+1
         3(n-1)+n+1             1+4+4+1
     if i = 3                      +6
         return true               16
```

(iii)

```
for (i=1 to 1000)
    i = i+2
```

$1 + 1000 + \frac{999(2)}{2}$

```
for(i=1 to 1000)
    i=i*2
```

$1 +$

```
for i=n to >1
    i=i/2
```

procedure: LS(A, n, k)

Input: An array with n integers, $n \geq 1$ and key

output: true or false

```
for i ← 0 to n
    if (A[i] == k]
        return true
return false
```

$1 + n + 4(n-1) + 1$

$4n - 4 + 2 + n$

$5n - 2$

$1 + n + 1 + 4n + 1$

$5n + 3$

```
for i ← 0 to n-1 do
for j ← 0 to n-1 do
    sum ← i + j
```

$1 + n(1 + n + 1 + 4(n) + 2n + 1) + 1$

$1 + n(5$

$1 + n + 1 + 2n + n(1 + n + 1 + 4n)$

$3n + 2 + n(5n + 2)$

$3n + 2 + 5n^2 + 2n$

$5n^2 + 5n + 2$

To solve nested problem the inner loop is taken as seperate block. That will be executed, later to the comparisions of outer loop.

Now inner unit should be solved in such a way that what is the task specified inside inner loop along with comparision of inner loop.

Asymptotic Notation

We have two function $f(n)$ & $g(n)$ $f(n) = O(g(n))$ if and only if there are two positive constants

$$|f(n)| \leq |c\,g(n)|$$

where $c > 0$, $n_0 \geq 1$, $n \geq n_0$

As n increases $f(n)$ grows no faster than $g(n)$

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

$f(n) \geq c(g(n))$

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

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

$20n^3 + 10n\log n + 5 \quad O(n^3)$

$20n^3 + 10n\log n + 5 \leq cn^3$

let $c = 200$

$20n^3 + 10n\log n + 5 \leq 200\, n^3$

$\forall\, n \geq n_0 \quad n_0 = 1$

$3\log n + \log\log n$ is $O(\log n)$

2^{100} is $O(1)$

$\frac{5}{n}$ is $O(\frac{1}{n})$

$2^n + n\log n$ is $\Theta(2^n)$

Rules to simplify Asymptotic notation

i) If $d(n)$ is $O(f(n))$ then $a\,d(n)$ is $O(f(n))$

ii) If $d(n)$ is $O(f(n))$ & $e(n)$ is $O(g(n))$ then $a>0$

$$d(n)+e(n) = O(f(n)+g(n))$$

iii) If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$

$$d(n)+e(n) = O(f(n)*g(n))$$

iv) If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$ then $d(n)$ is $O(g(n))$

v) $f(n)$ is a polynomial of degree d then $f(n)$ is $O(n^d)$

vi) n^x is $O(a^n)$ for any fixed $x>0$ and $a>0$

vii) $\log n^x$ is $O(\log n)$ $x \geq 0$

viii) $\log^x n$ is $O(n^y)$ $\forall x \geq 0$ $y \geq 0$

$20n^3 + 10n\log n + 5$ is $O(n^3)$

$O(n^3)$ $O(n)$ $O(n)$ $O(1)$ $\rightarrow O(n^3+n^2+1)$

$O(n^2)$ $= O(n^3)$

When a log term comes don't assume that the complexity is term of $\log(n)$ we have to apply the $\log n$ property and then check for the complexity.

$n\log n$ $\log^2 n$ $\sqrt{n}$ $\log\log n$ $\log n$ 2^n n^3 n^2 n

$\log\log n$ $\log n$ $(\log n)^2$ $\sqrt{n}$ n $n\log n$ n^2 n^3 2^n

$6n\log n$ 2^{100} $\log(\log n)$ $n^{0.01}$ $\sqrt{n}$ $4n^{\frac{3}{2}}$ $2n\log^2 n$ $4^{\log n}$

2^{100} $\log(\log n)$ $n^{0.01}$ $\sqrt{n}$ $6n\log n$ $4n^{\frac{3}{2}}$ $2n\log^2 n$ $4^{\log n}$

n $\sqrt{n}$ $n^{1.5}$ $n\log n$ $n\log\log n$ $\frac{2}{n}$ 2^n

$\frac{2}{n} < \sqrt{n} < n\log\log n < n\log n < n^{1.5} < 2^n$

Iterative substitution method

Recursive tree method

Guess and Test method

Master method.

$$T(n) = \begin{cases} 3 & n=1 \\ 8T(\frac{n}{2}) + n^2 \end{cases}$$

$$T(n) = 8\,T(\tfrac{n}{2}) + n^2$$

$$T(\tfrac{n}{2}) = 8\,T(\tfrac{n}{4}) + \tfrac{n^2}{4}$$

$$8\left[8\,T(\tfrac{n}{4}) + \tfrac{n^2}{4}\right] + n^2$$

8) $T(\frac{n}{4}) = 8\,T(\frac{n}{8}) + \frac{n}{4}$

$T(n) = 8^2\,T(\frac{n}{2^2}) + 2n^2 + n$

$= 8^3\,T(\frac{n}{2^3}) + 21n$

$T(n) = 8^i\,T(\frac{n}{2^i}) + (2^i - 1)\,n^2 \quad n = 2^i$

$8^i \cdot T(\frac{2^i}{2^i}) + (2^i - 1)$

$n^3 + (n-1)n^2$

$= 2n^3 - n^2$

$= O(n^3)$

$T(n) = \begin{cases} b & \text{if } n \le 3 \\ 8T(\frac{n}{3}) + bn \end{cases}$

$T(n) = 4\,T(\frac{n}{2}) + n^3$

$T(\frac{n}{2}) = 4\,T(\frac{n}{4}) + \frac{n^3}{8}$

$T(\frac{n}{4}) = 4\,T(\frac{n}{8}) + \frac{n^3}{64}$

$T(n) = 4\left(4\,T(\frac{n}{4}) + \frac{n^3}{8}\right) + n^3$

$= 16\,T(\frac{n}{4}) + \frac{n^3}{2} + n^3$

$= 16\,T(\frac{n}{4}) + \frac{3n^3}{2}$

$$= 16\left(4T\left(\frac{n}{8}\right) + \frac{n^3}{64}\right) + \frac{3n^3}{2}$$

$$4^3 T\left(\frac{n}{8}\right) + \frac{n^3}{4} + \frac{3n^3}{2}$$

$$4^3 T\left(\frac{n}{8}\right) + \frac{7n^3}{4}$$

$$4^i T\left(\frac{n}{2^i}\right) + \frac{2^i - 1}{2^{i-1}} n^3$$

$$n^2 \times 1 + 2\frac{(n-1)}{n} n^3$$

$$O(n^3)$$

$n = 2^i$

$\frac{2^i}{2}$

$\frac{3}{2}$

$\frac{7}{4}$

$T(n) = 3T\left(\frac{n}{3}\right) + bn$ (3-way merge sort)

$T\left(\frac{n}{3}\right) = 3T\left(\frac{n}{9}\right) + \frac{bn}{3}$

$T\left(\frac{n}{9}\right) = 3T\left(\frac{n}{27}\right) + \frac{bn}{9}$

$$T(n) = 3\left(3T\left(\frac{n}{9}\right) + \frac{bn}{3}\right) + bn$$

$$= 3^2 T\left(\frac{n}{9}\right) + bn + bn$$

$$= 3^2 T\left(\frac{n}{9}\right) + 2bn$$

$$= 3^2\left(3T\left(\frac{n}{27}\right) + \frac{bn}{9}\right) + 2bn$$

$$= 3^3 T\left(\frac{n}{27}\right) + 3bn$$

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

$$= nb + \log_3 n\, bn$$

$n \log n$

$$T(n) = \begin{cases} a & n = 1 \\ 2T(n-1)+b & \text{otherwise} \end{cases}$$

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

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

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

$T(n) = 2(2T(n-2)+b)+b$

$= 2^2T(n-2)+3b$

$= 2^2[2T(n-3)+b)+3b$

$= 2^3T(n-3)+7b$

$= 2^iT(n-i)+2^i-1b$

$i+1 = n$

$2^{n-1}(a) + 2^{n-1}-1\ b$

$= 2(2^{n-1})$

$= 2^n$

$T(n) = 2T(\sqrt{n}) + \log n$

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

$T(n^{\frac{1}{2}}) = 2T(n^{\frac{1}{4}}) + \log n^{\frac{1}{2}}$

$T(n^{\frac{1}{4}}) = 2T(n^{\frac{1}{8}}) + \log n^{\frac{1}{4}}$

$2\left(2T\left(n^{\frac{1}{4}}\right)+\log n\right)+\log n$

$2^2 T\left(n^{\frac{1}{4}}\right)+2\log n$

$2^2\left(2T\left(n^{\frac{1}{8}}\right)+\log n\right)+2\log n$

$2^3 T\left(n^{\frac{1}{8}}\right)+3\log n$

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

$\log n + \log\log n = \log n$

$O(\log\log n \log n)$

$T(n)=T\left(\frac{n}{2}\right)+1$ (P.T $T(n)$ is $O(\log n)$)

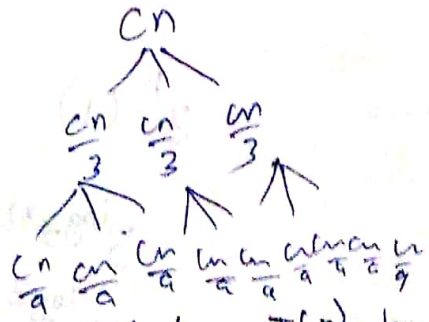
$T(n)=2T\left(\frac{n}{2}\right)+bn\log n$ $O(n(\log n)^2)$

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

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

In the recursive tree method construct a tree/build a tree starting from a single node that single node is considered as $T(n)$ and represent it same in the form of a diagram.

The value of all 3 children nodes + the root node. Also find out each level sum. To evaluate sum the cost of each node of each level.



At h level we assume that $T(n)$ becomes $T(1)$ by dividing $\frac{n}{2^h}=1$

The sub problem sizes are $\frac{n}{2}, \frac{n}{4}, \frac{n}{27} \cdots 1$ equal to $T(1)$ = At h^{th} level sub prob size will become

$$\frac{n}{3^n} = 1$$

$$n = 3^h$$

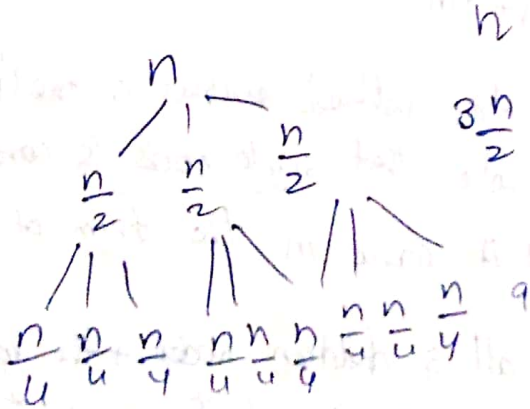
$$h = \log_3 n$$

To evaluate $T(n)$ add all constant of all level along with n^{th} level sum.

To find out efficiency in terms of n add all level sum n^{th} level sum

$$3^h \times 1 + cn\log_3 n$$

$$n + cn\log_3 n$$



h

$3\frac{n}{2}$

$9 \times \frac{n}{4}$ $\frac{9n}{4}$

$2^h = n$

$n = \log_{2^h} 3^h$ n

$3^h(1)$

$3^{\log_2 n}$

$n^{\log_2 3}$

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

3^h

$2\left(\frac{3}{2}\right)^h - 1$

$2 \times \left(\frac{3}{2}\right)^h - 1$

$2 \times \left(\frac{3}{2}\right)^{\log_2 n}$

$\left(2 \times n^{\log_2\left(\frac{3}{2}\right)}\right)$

$2n$

$3^{\log_2 n}$

$r^{\log_2 3}$

$\frac{2 \times 3^h - 1}{2^n}$

$\frac{2 \times n^{1.5}}{n}$

There are three rules in master method.

Rule 1: If there is a small constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$ then $T(n) = \Theta(n^{\log_b a})$

The rule 1 characterize where $f(n)$ is polynomial smaller than special function $n^{\log_b a}$

Rule 2: If there is a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \log^k n)$ then $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$

Rule 2 characterize that $f(n)$ is asymptotically close to the special function $n^{\log_b a}$

Rule 3:- If that is a small constant $\epsilon > 0$ another constant $\delta < 1$ s/t $f(n) = \Omega(n^{\log_a b + \epsilon})$ & $af\left(\frac{n}{b}\right) \leq \delta f(n)$ then $T(n) = \Theta f(n)$

Rule 3 characterization that $f(n)$ is polynomially larger to the special function $n^{\log_b a}$

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

$n^{\log_2 2}$

$n^1 = n$

$f(n) = n \log n$

$k = 1$

$\Theta(n)(n \log^{k+1} n)$

$= \Theta(n(\log^2 n))$

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

$f(n) = n^{\log_3 1}$

$\} = 1$

Polynomially ~~scaled~~ larger

$T(n) = \Theta(n)$

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

$f(n) = n^2$

$n^{\log_2 4} = n^2$

$\Theta(n^2 \log^2 n)$

* $T(n) = 4\,T\left(\frac{n}{2}\right) + n^3$
* $T(n) = 3\,T\left(\frac{n}{2}\right) + n$
* $T(n) = 3\,T\left(\frac{n}{4}\right) + cn^2$

$+ 3\,T\left(\frac{n}{4}\right) + n \log n$

$+ 0.5\,T\left(\frac{n}{2}\right) + \frac{1}{n}$

Sorting

1. bubble sort
2. Ins sort
3. selection sort
4. heapsort
5. Quick sort
6. Merge sort
7. bucket sort
8. Radixsort
9. shell sort

curr = 0
sorted = False
top (curr < n & sorted = False)
    i = n-1
    sorted = true
    loop(i > curr)
        A[i] < A[i-1]
            sorted = False
        swap(A, i, i-1)
    else
        i = i-1
    curr = curr+1

for (i=0 to n-1) do
{ swapped = False
    for (j=0 to n-i-1) do
    if (A[j] > A[j+1])
        swap(A[j], A[j+1])
        swapped = true

if(swapped = False)
    break;
end loop

```
curr←1
loop(curr<n)
    hold←n
    curr←curr-1
    loop(i>=0 ft hold < A[i])
        A[i-1]←A[i]
        i=i-1
    end loop
        A[i+1]←hold
        curr=curr+1
end loop.
```

```
curr←0
loop(curr<n-1)
    smallest
      i=curr+1
    loop(i<n)
      if(A(i)<A(small))
        small←i
      end if
          i←i+1
      end for
      swap(A(curr),A(small))
          curr=curr+1

for( i←0 to. i<=n-1 do)
      reheap(heap,i)
  for(j←n-1 to j>=0 do)
      { exchange(heap,0,j)
      . reheapdown(heap,0,j-1)
```

```
end if
Reheapupheap, newnode)
    if (newnode at zero)
    {   parent = newnode(newnode - 1)/2
        if [heap[newnode] > heap[parent]
          then
              swap(newnode, parent)
              reheap (heap, parent)
          end if

Reheapdown (heap, root, n)
   if ( root*2+1 < n) then
      left ← heap[root * 2 + 1]
        if (root x2 +2 < n) then
          right ← heap[root x2 + 2]
        else right ← -1

   if (left < right) then
        large CI ← root + 2 + 1
   ele
        large CI ← root + 2 + 2
   end if
     if (heap[root] < heap[large CI) then
        swap (root, large, CI)
        heapdown (heap, large(I, n-1))
     end if
   endif
```

heapsort is improved version of selection sort we can either construct max heap or min heap

For a node located at index i the children are found as follows:

$left \leftarrow 2i+1$

$right \leftarrow 2i+2$

For a node located i, its parent is located at $(i-1)/2$

The two operations which takes place is as follows

The reheapup

The reheapdown

In reheapup the broken heap is rearranged by pushing the last element towards up (the large element) until it is in its current location of heap

reheapdown the broken heap is rearraged by pushing the root down the tree until in its correct position in heap.

25 43 21 60 55 4

Quicksort

```
if((right-left)>minsize)
 {median(n,left,right)
    i←left
    j=right
    pivot= a[i]
     while(pivot>=a[i])
       do i++
```

```
while(pivot<=a[j])
  j--
end loop
if(i<j)
  exchage(n,i,j)
  end if
  end loop
exchage(n,left,j)
  if(left<j)
    quicksort(a,left,j-1)
  endif
  if(i<right)
    quicksort(a,i,right)
    end if
    }
  else insertionsort(a,left,right)
  endif

median(n,l,k)
  m<-(l+n/2)
    exchage(a,l,n)
    if(a[i]>a[k])
      exhaye(n,l,k)
    if(a[m]>a[k])
      exchage(a,m,k)
      exchage(n,l,m)
```

B A W
O(nlogn) O(nlogn) $O(n^2)$

```
Mergesort(a,low,high)
if(low<high) then
  mid= L(low+high)/2 ]
  mergsort(a,low,mid);
  mergesort(a,mid+1,high)
  return  merge(a,low,high)

merge (a,low,high)
    h← low
    j← mid+1
while( h<= mid && j<=high)
    if(a[h] <a[j]) then
          b[i]←a[h]
          h←h+1
        else
          b[i]←a[j]
           j←j+1
    endif
          i←i+1
    endwhile
        if(h>mid)then
  for k←j to high do
        b[i]←a[k]
        i←i+1
  endfor
```

else for i ← h to mid do

b[i] ← a[k]

i ← i + 1

end for

end if

for (i ← low to high do)

end for a[i] ← b[i]

B, W, Av

Complexity O(nlogn)

Let s be the no. of elements

in the array to be sorted

for each item x in s do

Let k be the key

place item x in the bucket B[k]

end for

for i ← 0 to d do (denote the no. of buckets)

for each item n B(i) do

place x in array (sorted order)

end for

end for

Radix sort

Complexity

O(s+d) → bucketsort

O(D(s+d))

The input array is sorted vector/array

```
Binary search(s,k, low, high)
    if low = high then
        return No-such-key
    else
        mid = ⌊(low+high)/2⌋
        if k = key(mid) then
            return ele(mid)
        elif if k < key(mid) then
            return BinarySearch(s,k,low,mid-1)
        else
            return Binarysearch(s,k,mid+1,high)
```

	Bis	Lis	Ins	Buc	Sel	Radix	Bub	H	Q	M
B	O(1)	O(1)	O(n)	O(n+N) (n=data, N=buckets)	$O(n^2)$	O(d(n+N))	O(n)	O(nlogn)	O(nlogn)	O(nlogn)
A	O(logn)	O(n)	$O(n^2)$	"	$O(n^2)$	"	$O(n^2)$	O(nlogn)	O(nlogn)	O(nlogn)
W	O(logn)	O(n)	$O(n^2)$	"	$O(n^2)$	"	$O(n^2)$	O(nlogn)	$O(n^2)$	O(nlogn)

Stability & Inplace :-

An algorithm is said to be stable if it preserves order of the occurence of keys

7 12 25 7 13 7

7 12 25 7 13 7

7 12 25 7 13 7

7 7 12 25 13 7

7 7 12 13 25 7

7 7 7 12 13 25

An algorithm if it uses same available space throught the process in place algorithm no extra memory is required.

Heapsort is not inplace not stable

Quicksort is not stable not inplace

Mergesort is not inplace and not stable

15×11

1111 1011

I J

$I = I_h 2^{n/2} + I_L$

$J = J_h 2^{n/2} + J_L$

$I \cdot J = (I_h 2^{n/2} + I_L) \cdot (J_h 2^{n/2} + J_L)$

$= I_h 2^{n/2} J_h 2^{n/2} + I_h J_h 2^{n/2} + I_h 2^{n/2} I_l + I_L J_L$

The complexity of a recursive is as follows

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & n \leq 2 \end{cases}$$

Consider $(I_h - I_L) \times (J_L - J_h) = I_h J_L + I_L J_h - (I_h I_h + I_L J_L)$

$I \cdot J = I_h 2^{n/2} J_h 2^{n/2} + 2^{n/2} \left[I_h J_h + I_L J_L + (I_h - I_L)(J_h - J_L) \right] + J_L I_L$

$O(n^{1.58})$

15 x 11

1111 1011

11 11 10 11

1 1 1 0 1 1

0110
1001
1111 110

1100 * 1000 + 1111 00 + 1001

$I.J = 2^n I_n J_n + 2^{n/2}(I_n J_n + I_1 I_1) + (I_n - I_1)\cdot(J_1 - J_n) + I_1 J_1$

1011 0011 * 1011 1010

00111100 111100
1100000 1001
1001 1100 1000
1001 1100
001

1011 0011 1011 1010

10100101 11100
1001
01000101
1000
01001101
1100
01010001

10 11 00 11 10 11 10 10

1 0 1 1 0 0 1 1 0 1 0 1 0

128 32 5 165

10 10 64 16 1 81

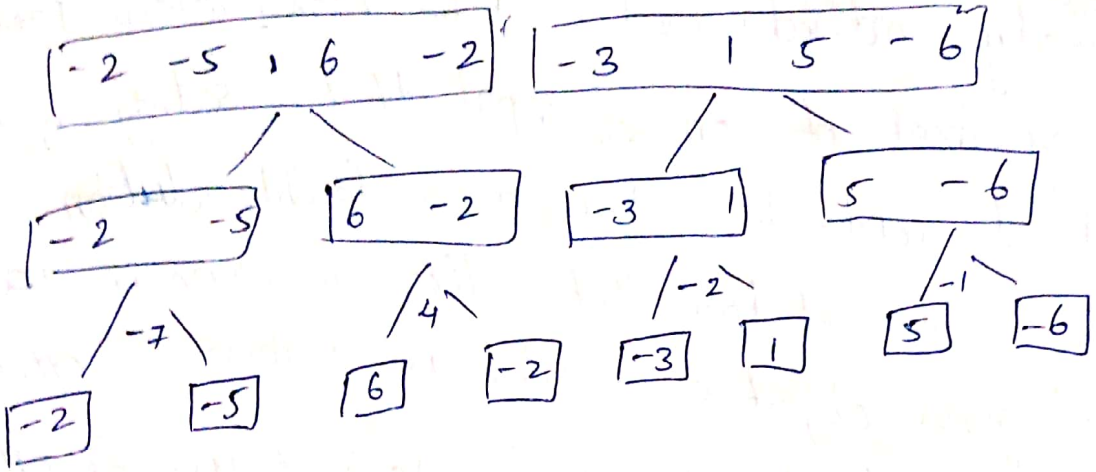
$2^2(1.1) +$ $2(1+0)+1($

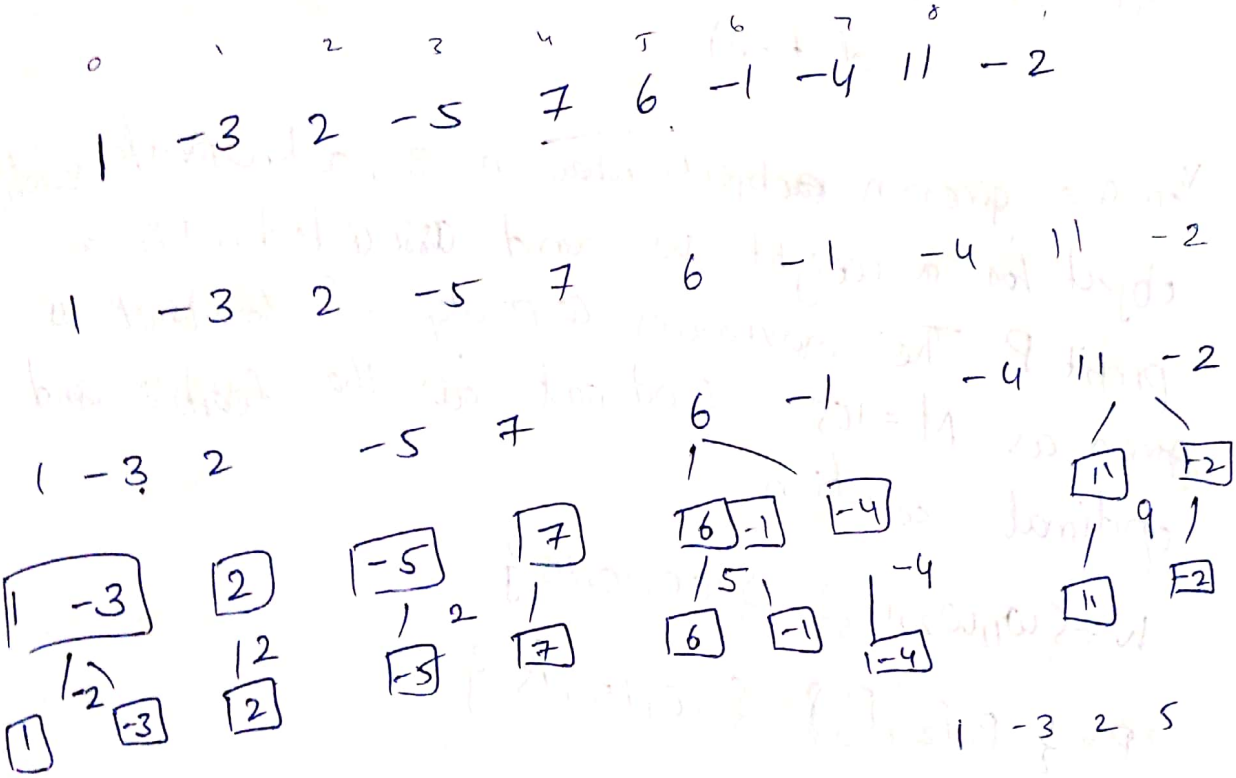
0001
1001
111

Subarray max:-

1. Divide the given array into two halves
2. Return the max of following
   - (a) Maximum sub array sum of left half
   - (b) Maximum subarray sum of right half
   - (c) Maximum subarray sum such that sub passes the mid point

where a, b is recursive call c is finding the maximum sum starting from mid point ending at some point on left side of mid and also find maximum sum of starting from mid+1 and ending at some points on the right of mid+1, finally combine two of results and return back.






```
maxcur = max_global = A[0]
for i from 1 to length(A) - 1
{ max_cur = max(A[i], max_cur + A[i])
  if(maxcur_cur > global)
     maxglobal = max_cur
}
return max_global.
```

* Greedy method is used to solve optimization Problem.
* Any subset of original input that satisfy given set of constraints is known as feasible solution.
* A feasible solution that either minimize or maximize a given objective function is known as optimal solution (Best out of feasible is known as optimal solution)

You are given n objects where n = 3, a knapsack each object has a weight W and associated with a profit P. The maximum capacity of knapsack is given as M = 105. Find out all the feasible and optimal solution.

$W = \{w_1, w_2, w_3\} = \{100, 10, 10\}$

$P = \{P_1, P_2, P_3\} = \{20, 15, 15\}$

$\{1\} \rightarrow 20$

$\{2\} \rightarrow 15$

$\{3\} \rightarrow 15$

$\{1,2\} = X$

$\{1,3\} = X$

$\{2,3\} = 30 \rightarrow$ optimal.

$\{1,2,3\} = X$

~~$P = 0$~~

Task scheduling

① m ← 0

② While (set of tasks(T) ≠ 0) do

③ remove a task i with smallest start time s_i from T

④ If there is a machine 'j' with no task conflicting with Task i then

⑤ Schedule task i on machie i

⑥ else m ← m+1

⑦ schedule disk i an machine M

⑧ end if

⑨ end while.

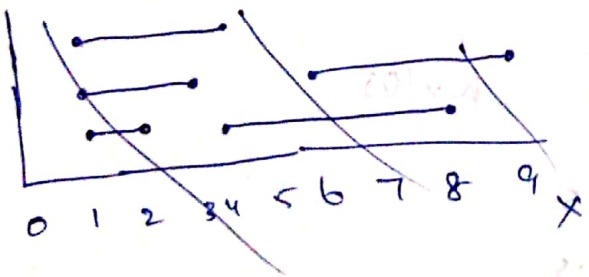
Suppose you are given a set of tasks specified by the starting time and ending time as follows

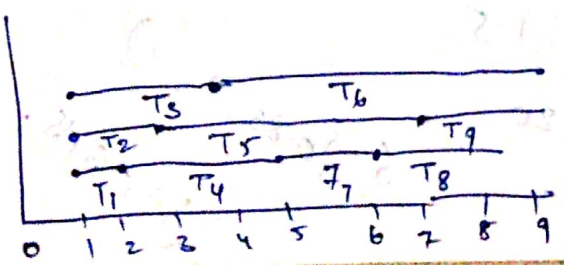
Solve T using greedy approach and find no. of machine used

① $T = \{ \overset{T_1}{(1,2)}\ \overset{T_2}{(1,3)}\ \overset{T_3}{(1,4)}\ \overset{T_4}{(2,5)}\ \overset{T_5}{(3,7)}\ \overset{T_6}{(4,9)}\ \overset{T_7}{(5,6)}\ \overset{T_8}{(6,8)}\ (7,9) \}$

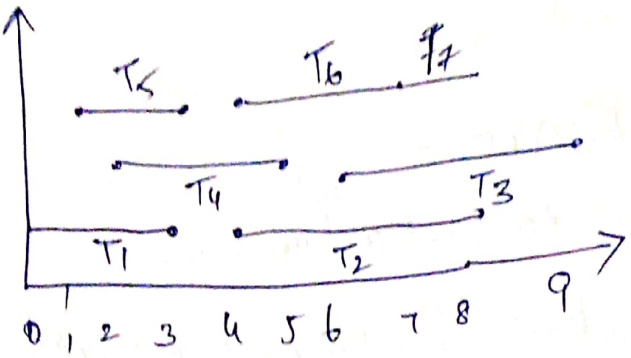
② $T = \{ \overset{T_9}{(0,3)}, (4,8), (6,9), (2,5)(1,3)(4,7)(7,8) \}$

①





(c)



Fractional knapsack:-

I/p set S of items, such that each item i ∈ S has a +ve benefit b_i and +ve weight w_i +ve max total weight W

Algorithm

For each item i ∈ S do

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$

end for

$w \leftarrow 0$

while $w < W$ do

remove from S an item i with highest value in it

$a \leftarrow \min(w_i, W - w)$

$x_i \leftarrow a$

$w \leftarrow w + a$

end while

W = 105

n = 3

	b	v
100 →	20	0.5
10 →	15	1.5
10 →	15	1.5

(85) $\frac{w_2}{15} + \frac{w_3}{15} + \frac{85}{100} + \frac{20}{}$

$S = \{0.85, 1, 1, 1, 1, 1\}$

	w	p	v_i	x_i	W = 20
①	18	25	$\frac{25}{18} = 1.3$	0	
	15	24	$\frac{24}{15} = 1.6$	$\frac{15}{15}$	
	10	15	1.5	$\frac{5}{10}$	

$x_i = \{0, 1, 0.5\}$ $\quad 24 + 7.5 = 31.5$

② $S = \{a, b, c, d, e, f, g\}$ $\quad W = 18$

$w = \{11, 6, 5, 7, 3, 1, 6\}$

$p = \{12, 10, 8, 11, 4, 7, 9\}$

3 1.6 2 1.6 1.5 2 4.6 7 1.5

⑤ $w = \{2, 3, 5, 7, 1, 4, 1\}$ $\quad W = 15$

$p = \{10, 5, 15, 7, 6, 18, 3\}$

Huffman coding :-

```
n = |C|
Q = C
for i = 1 to n-1
allocate a new node z
z.left = x = EXTRACT_MIN(Q)
z.right = y = EXTRACT_MIN(Q)
z.freq = x.freq + y.freq
INSERT(Q, z)
return EXTRACT_MIN(Q)
```

An optimal code for a file is always represented by a full binary tree. Every non-leaf node has two children. For each character 'c' in the alphabet 'C' (code), let the attribute c.frequency denote the frequency c in the file and $d_T(c)$ denote

depth of c's leaf in the tree. The cost of the tree

$$B(T) = \sum_{c \in C} c.\text{frequency} * d_T(c)$$

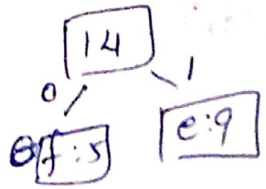
Calculate the total no. of bits used with fixed and variable length codeword.

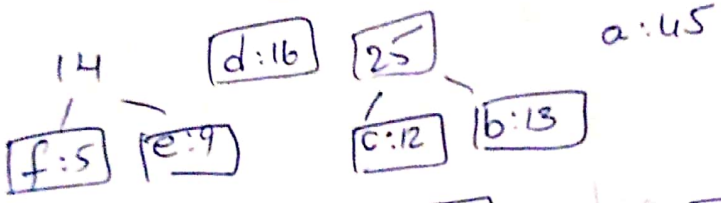
Frequency	a	b	c	d	e	f
Fixlength	45	13	12	16	9	5
	000	001	010	011	100	101

a:45 b:13 c:12 d:16 e:9 f:5

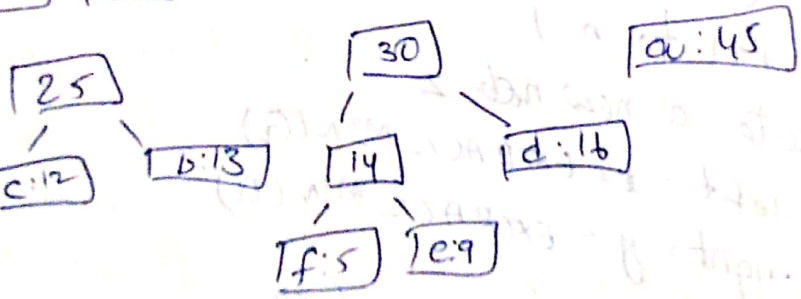
f:5 e:9 c:12 b:13 d:16 a:45

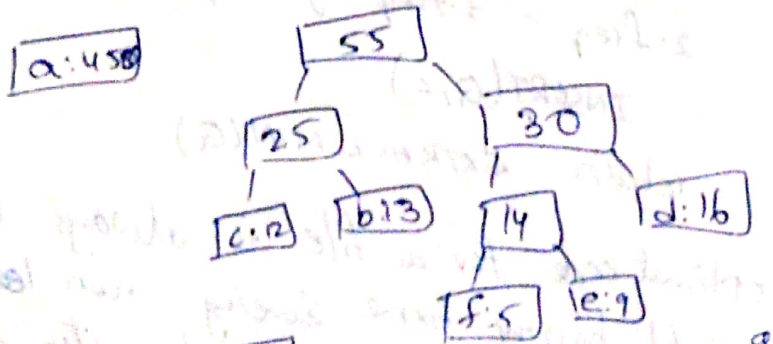
c:12 b:13 d:16 a:45

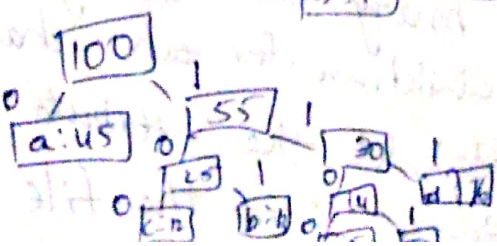




nlogn+n



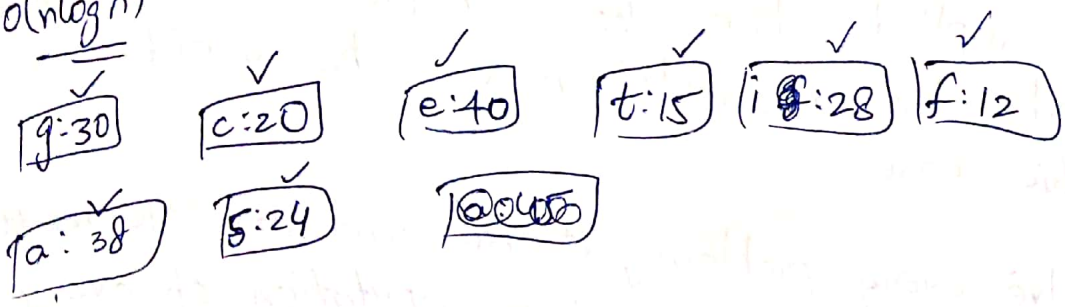


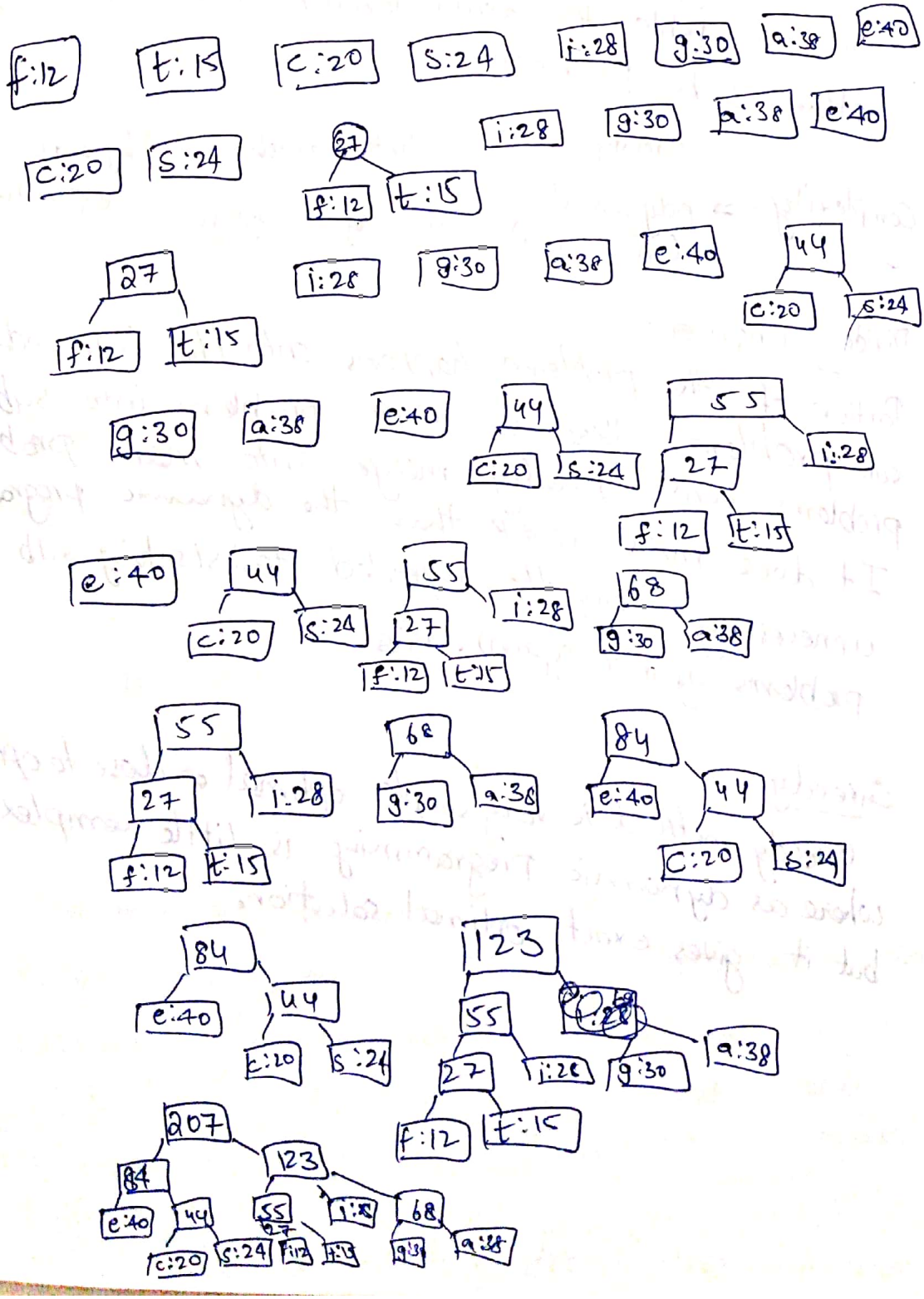


a → 0
c → 100
b → 101

If c is an alphabet from which the tree is drawn then optimal prefix code will have |c| → leaf nodes |c|−1 root nodes.

O(n log n)





Dynamic Programming:-

It is a strategy which is used to solve the problem which leads to exponential complexity. It can be applied when the sub problem are dependent and the recomputation can be avoided in this case.

Solve every problem just once and store the result in a table to avoid recomputation of every other sub problem.

	Greedy	Divide & Conquer	Dynamic
Complexity →	polynomial ($n^2, n, n\log n$)	$\log n, n\log n, n^2\log n$ etc	exponential

Divide & Conquer:-

Portion of the problem happens onto the independent sub problems. Here we divide problem into sub problems and finally merge into main problem. It does more work than the dynamic programming unnecessary doing the repeated tasks (solving sub problems again & again).

Greedy:-

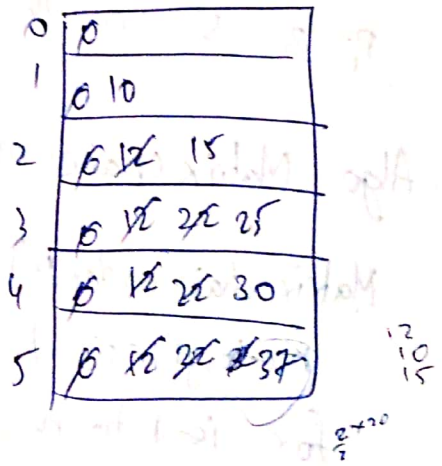
Greedy method is very simple optimal or close to optimal where as dynamic programming is little complex but it gives exact optimal solution.

Dynamic Programming

Algo :- 0/1 knapsack

```
for W←0 to W do
    B[W]←0
  for k←1 to n do
    for w←0 to wk do
      if B[w-wk]+bk>B[w] do
        B[w] ← B[w-wk]+bk

        keep[k,w]=1
        else
            keep[k,w]=0
        end if
    end for
end for
```

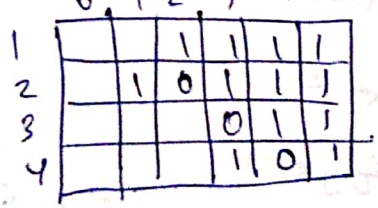


Profit :-

```
k=W
for i←n to 1 decrement[decrement]
  if keep[i,k]==1
  k=k-wi
end for
```

W=5

	2	1	3	2
	12	10	20	15
	6	10	6.67	7.5



[1 1 0 1]

There are four steps in Dynamic programming

1. characterise the steps/structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the optimal solution of all sub problems in a bottom up fashion by filling table
4. Construct optimal solution from the computed

information.

W = 10

n = 4

w_i	5	4	6	3
p_i	10	40	30	50

W = 9

n = 5

w_i	1	3	4	3	2
p_i	3	5	20	18	4

Algo:- Matrix chain multiplication

```
Matrix chain order(p)
    n ← length[P] - 1
   for i ← 1 to n
    do m[i,i] ← 0
   for i ← 2 to n
   do for i ← 1 to n-l+1
    do j ← i+l-1
       m[i,j] ← ∞
   for k ← i to j ← i
    do q ← m[i,k] + m[k+1,j] +
                                  <
       if q ← m[i,j]
        then m[i,j] ← q
             s[i,j] ← k
          return m,s
```

Recursive solution

k lies between i, j

i to k and k+1 to j are 2 different matrices so

$m[i,j] + m[k+1,j] + P_{i-1} P_k P_j$.

so

$$m[i,j] = \begin{cases} 0 & \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] \} + P_{i-1} P_k P_j & \text{if } i < j \end{cases}$$

A → 10×100

B → 100×5

C → 5×50

D → 50×1

10 100
100 5
5 × 1

8000 → ((AB)C)D → 5000+2500+500 (100000)

→ A(B(CD)) 250+500+1000 (1750)

A((BC)D) 25000+5000+1000 = 31000

(AB)(CD)

5000+2500+50

= 5300

Count the number of parenthesis (paranthesization):-

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)\, P(n-k) & \text{if } n \ge 2 \end{cases}$$

n=1 P(1) = 1

n=2 P(1) P(1) = 1

n=3 P(1)·P(2) + P(2) P(1) = 2

n=4 P(1)P(3) + P(2)P(2) + P(3)P(1) = 2 + 1 + 2 = 5

n=5 P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1) = 5 + 2 + 2 + 5 = 14

(i, j)

$A_1 = P_0 \times P_1$ 30 × 35

$A_2 = 35 \times 15\ P_2$

$A_3 = 15 \times 5\ P_3$

$A_4 = 5 \times 10\ P_4$

$A_5 = 10 \times 20\ P_5$

$A_6 = 20 \times 25\ P_6$

Create two tables 'M' table and 's' table where in m table we enter the no. of computations and in s table we enter the minimum Value of k. For all values of k which lies in between i & j we have to find no of computations and Put it into s table.

$P_0 = 30$ $P_1 = 35$ $P_2 = 15$ $P_3 = 5$ $P_4 = 10$ $P_5 = 20$ $P_6 = 25$

M

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

6 5 4 3 2 1

$0 + 3500 + 15 \cdot 0 \cdot 25$

5375

$2500 + 0 + 15 \cdot 20 \cdot 25$

6250

$0 + 5000 + 5 \times 10 \times 25$

$1000 + 0 + 5 \cdot 20 \cdot 25$

$0 + 0 + 15 \cdot 5 \cdot 10$

$750 + 5000 -$

$0 + 1000 + 15 \cdot 5 \cdot$

S

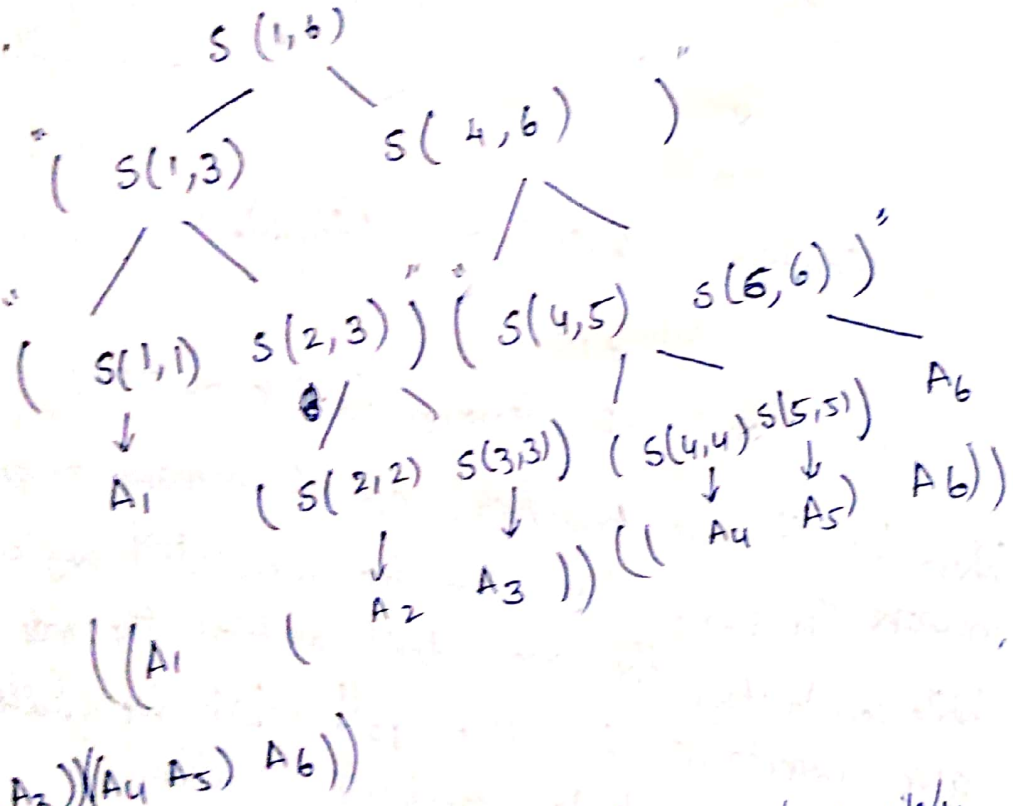
	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

6 5 4 3 2 1

i=3 j=5 k=3,4

i=3 j=4 k=3

```
print optimal parenthesis (s, i, j)
  if i == j then
    print Ai
  else
    print "("
print_optimal (s, i, s[i,j])
print_optimal (s, s[i,j]+1, j)
print ")"
```



Backtracking :-
A given problem a set of constraints and possibly an objective function

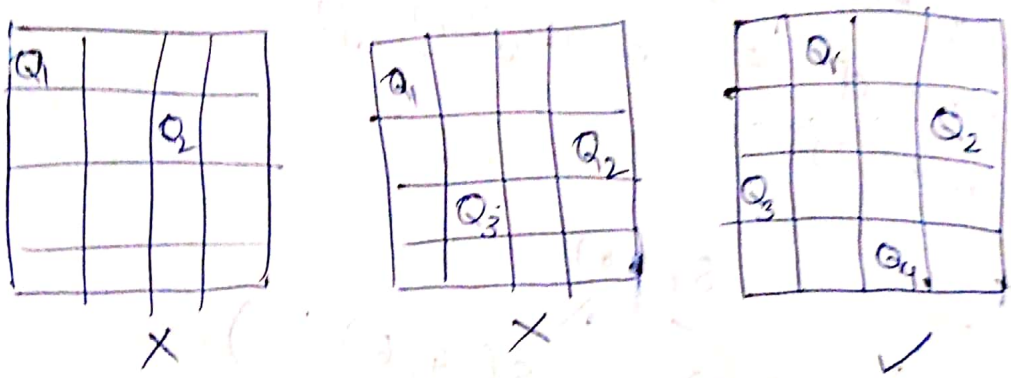
The solution optimizes the objective function. We can represent the solution in the form of tree. The tree is known as state space tree. In this tree a path from the root node to leaf node represents ~~a~~ candidate solution.

Search the state space tree in depth first manner we may endup with a recursion procedure

we may have to use the stack to refer the path from root to current node.

The solutionspace tree is a reference for our analysis but not exactly displayed to the computer.

Eg:- Sudoku, Backtracking, TSP



Solution 2,4,1,3 (feasible)

(or)

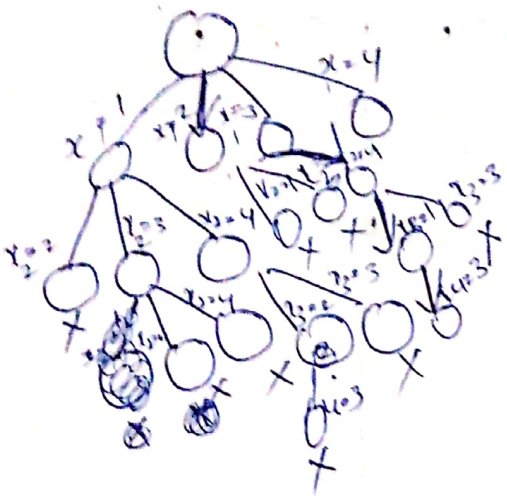
3,1,4,2 (another feasible solution)

When the Queens number ~~of~~ increases, no of solutions increases. The same this can be represented using a tree.

While constructing the tree don't mention the node numbers, give importance for the path which we follow. So that we can reach to candidate solutions.

Fix a root node from the root node we can have four diffrent paths $x_1 = 1, x_1 = 2, x_1 = 3, x_1 = 4$. For the next level to place x_2 find out all possible paths.

Also leave the column which is selected



Here x_2 in 2 is not allow

1

3×4

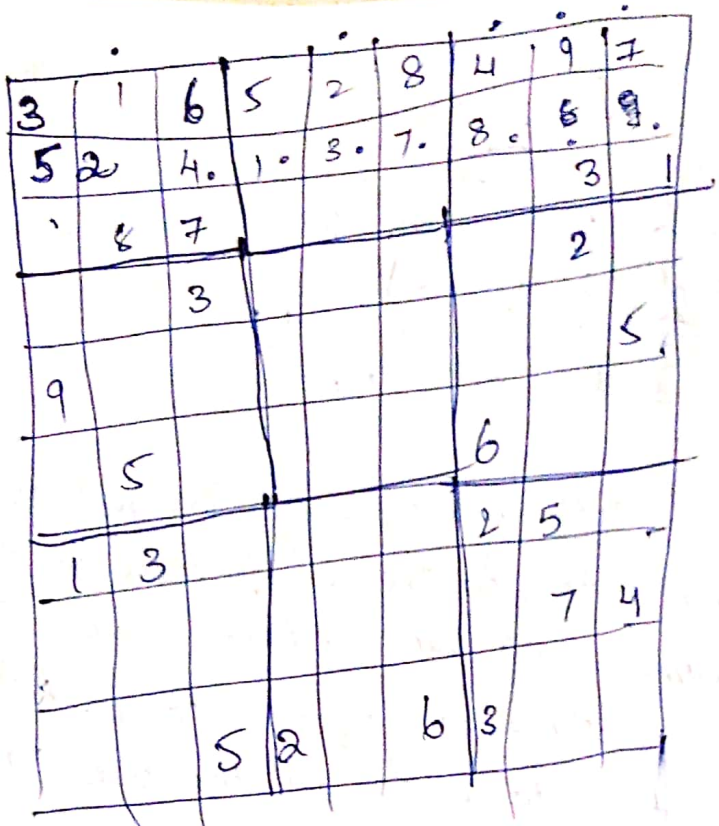
O(n!)

Given a partial 4×4 grid (2D array) The goal is to assign digits from 1 to 4 to the empty cells. so that every row column and sub grid of size 2×2 contains one instance of digit 1 to 4. All possible configuration of 1 to 4 to fill the cells. Try every configuration one by one until you find configuration which is correct that take $O(n^n)$.

When we apply backtracking algorithm the complexity can be reduced that can teams in $O(n!)$ and it is based on clues which is given to you.

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

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



Branch & Bound:-
It is an enhancement of of backtracking. It is used for finding optimal solution for various optimal problems. It consists of systematic enumeration of all candidate solutions. The partially/candidate solution can be discarded based on the upper & lower Bound.
It does not limit any particular way of traversing the tree
It is similar to backtracking in term of tree creation

```
nqueen(n)
    //x[] is a vector
    // k denotes row
     x[1]←1
       k←1
    while(k>=1)
{   while( x[k] <= n && !place (x,k)))
          x[k] ← x[k]+1
      if(x[k]<= n) //place queen
          if(k= n)   Display(x, n)
      else
          k←k+1  //next queen
        x[k]←1  }
```

```
else
  k←k-1   // backtrack

}
place[x[],k)
{ for i←1 to k
    if (x[i]==x[k]||(abs(x[i] - x[k]) == abs(i-k))
      return 0

    return 1
  }
Display(x[],n)
for i←1 to n do
 for j←1 to n do
    chessboard[i][j]='x'
 for i←1 to n do
      chessboard[i][x[i]]='Q'
for i←1 to n do
   for j←1 to n do
   print "chessboad[i][j]

Algorithm Nqueen(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   Nqueen(k+1,N)
      }
  }
```

```
Algorithm place(k,i)
  { for j←1 to k-1 do
      if (x[j]=i)  //same column
        //(Abs x[j]-i)- Abs(j-k) //same digonal
        then return false
    return true
  }
```

The idea behind backtracking w.r.t to nqueens problems to place the queens one by one starting from left most column from left most column. When we place Q in column we check for crashes already. We check for the clashes with already placed queens (In the current column we find a row for which there is no clash we mark this row and column as part of the solution. If we don't find such a row due to clashes, then we have to apply backtracking (or return false/no solution)

Normally, if the permutation tree or the state space tree will be drawn with all possible solutions (nodes) then n^n nodes will have to be generated. So complexity is $O(n^n)$. If we use binding function in the backtracking algorithm no. of nodes generated in the case of n-queens problem will be n so the complexity will be $n!$

1) bool SolveSoduku (g[N][N])
if(! Findunassigned location (g, row, col)
return true
for num ← 1 to 9
{ if (issafe(g, row, col, num)
{ g[row][col] ← num
if (solvesoduku (g))
return true
g[row][col] ← UNASSIGNED //failure, try again
} } return false; //trigger back tracking
}

2) bool Findunassigned location (g[N][N], &row, &col)
for row ← 1 to N
for col ← 1 to N
if (g[row][col] = UNASSIGNED)
return true
return false
}

3) bool usedInRow (g[N][N], row, num)
for col ← 1 to N
if (g[row][col] = num)
return true
return false

4) bool usedIncol (g[N][N], col, num)
for row ← 1 to N
if (g[row][col] ← num)
return true
return false

5) bool UsedInBox(g[N][N], box, startRow, boxstartCol, num)

```
for row←1 to 3
    for col←1 to 3
        if (g[row + boxstartedRow][col + boxstartd] = num)
            return true
        return false
```

boxstartedRow = row - 0
boxstartedCol = col - 0

6) bool issafe(g[N][N], row, col, num)

return (!UsedInRow(g, row, num) && !UsedInCol(g, col, num) && !UsedInBox(g, row - row%3, col - col%3, num))

7. print Grid(g[N][N])

```
for row←1 to N
    for col 1 to N
        printf("%d", g[row][col])
            pf("\n").
```

Rabin-Karp(T, P, d, q)

```
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t0 = 0
for i = 1 to m
    p = (d.p + P[i]) mod q
    t0 = (d.t0 + T[i]) mod q
for s = 0 to n-m
        if p == ts
        if (P[1 ... m] = T[s+1 ... sm])
```

print pattern found

if $s < n-m$

$t_{s+1} = d(t_s - T[s+1]t_1) + T[s+m+1] \bmod q$

Time complexity

$\theta((n-m+1)m + O(m)$

The given text T: 2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1

P: 3 1 4 1 5

Horkipmocllo

$q = 13$ (prime number, if it is not given take any prime number greater than 10)

// $31415 \% 13 = (7)$

$23590 \% 13 = 8$

$35902 \% 13 = 9$

$59023 \% 13 = 3$

$90231 \% 13 = 11$

$02314 \% 13 = 0$

$23141 \% 13 = 1$

$31415 \% 13 = (7)$

$14152 \% 13 = 8$

$41526 \% 13 = 4$

$15267 \% 13 = 5$

$52673 \% 13 = 10$

$26739 \% 13 = 11$

$67392 \% 13 = (7)$

$73992 \% 13 = 9$

$39921 \% 13 = 11$

In sevath step $31415 \% 13$ give value 7 (same as modulo we conclude search is found and pattern is matching)

but when we proceed at thirteen step there is a valid hit which is again 67399 which is not matchig with given pattern. That hit is knowas spurious hit which is not matchig pattern eventhough hit value is same.

Another method

KMP (Kruth Morris Prat Algorithm) :-

Example of Rabin karp

T: 3 4 1 5 9 2 6 5 3 5 8 9 7 9 3

P: 26

9-11

26%11 = 4

34%11 = 1

41%11 = 8

15%11 = 4

59%11 = 4

92%11 = 4

26%11 = 4 ← true

65%11 = 10

53%11 = 9

35%11 = 2

58%11 = 3

89%11 = 1

97%11 = 9

79%11 = 2

93%11 = 5

Graph:-

Graph is a combination of vertices and edges. Where vertices are represented $e_1, e_2 \ldots$. The vertices are connected using an edge. There are two types of directed graph & undirected graph

A graph is said to be a directed graph (u, v), if $E(u, v)$ is the order pair

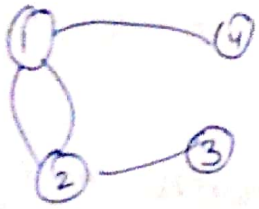
Undirected graph no direction is specified

Mixedgraph- combination of directed & undirected.

Graph is non-linear data structre

Representation of graph :-

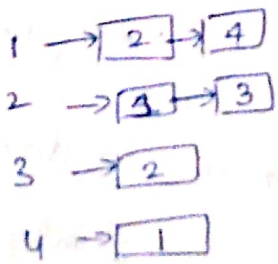
1. Adjacency List
2. Adjacency matrix



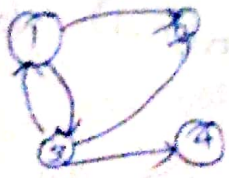
x	1	2	3	4
1	0	2	0	1
2	2	0	1	0
3	0	1	0	0
4	1	0	0	0

indegree, outdegree

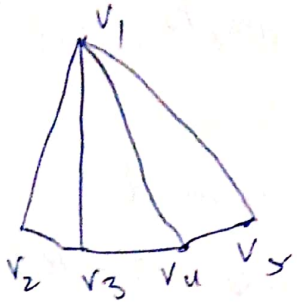
In the case of undirected graph indegree can be mentioned as -1 and outdegree as +1. If both are there both need to be mentioned as an entry to matrix. If no connection as 0



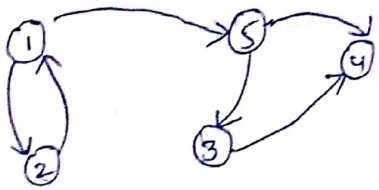
while making/drawing adjacency list take only +1 values to not take -1 values



x	1	2	3	4
1	0	+1	+1	0
2	0	0	0	0
3	1	0	0	0
4	0	0	0	0



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



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

Properties of Graph

1. If G is an undirected graph with m edges then

$$\sum_{v \in G} \deg(v) = 2m$$

2. If G is an directed graph with m edges then

$$\sum_{v \in G} \text{indeg}(v) = \sum_{v \in G} \text{outdegree} = m$$

3. Let G be a simple graph (no parallel edges, selfloops)

$$m \leq \frac{n(n-1)}{2} \quad \text{(undirected)}$$

$$m \leq n(n-1) \quad \text{(directed)}$$

4. Tree and forest

A tree / free tree is an undirected graph connected acyclic.

A forest is an undirected acyclic but not necessarily is connected.

A graph is said to be connected for any two vertices if there is a path existig between them.

Let G be an undirected graph with n vertices & m edges then the followig are true

If G is connected then $m \geq (n-1)$

If G is a tree then $m = n-1$ (acyclic graph)

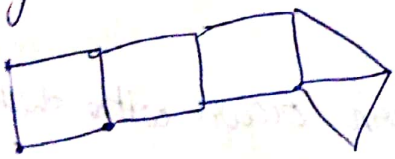
If G is a forest then $m \leq (n-1)$.

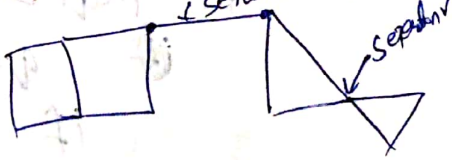
Multiple Graphs :-

If G contains parallel edges then it is known as multiple graphs.

Biconnected Graphs :-

A G with no seperation edges and no seperate vertices. For any vertices u and v there are two disjoint simple paths between u and v.





If V is visited for first time when we traverse through u, v then that edge is known as tree edge or discovery edge.

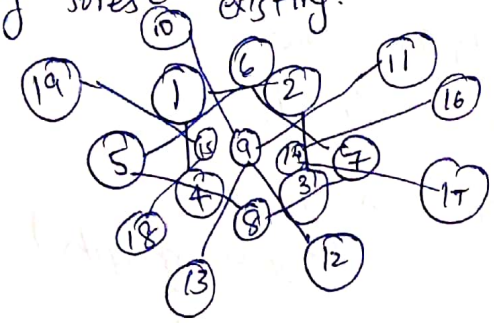
If V has already been visited then the following happen

① v is an ancestor of u (u, v) - back edge

② v is a dscat of u (u, v) - forward edge

(3) v is neither an ancestor nor an descent then (u,v) is an cross edge.

Check whether following graph is connected or not. If yes connected no-of components/trees and find out is there any forest existing.



4 trees
5-connected component
not a forest

```
KMPMATCHER (T,P)
{
    n = T.length
    m = p.length
    Π = computeprefixfunction(p)
    q = 0
    for i=1 to n
     while q>0 and p[q+1] ≠ T[i]
          q = Π[q]
        if p[q+1] == T[i]
            q = q+1
            if q == m
              print("Pattern occurs with shift", i-m)
              q = Π[q]

compute prefixfunction (p)
    m = p.length
    Let Π[1...m] be a new array
        Π[1] = 0
          k = 0
    for q = 2 to m
      while k>0 and p[k+1] ≠ p[q]
```

```
k = π[k]
if p[k+1] == p[q]
    k = k+1
    π[q] = k
return π
```

T: aabababababba

P: babab

Find out prefix function & check whether given pattern if it is present then how many shifts are required

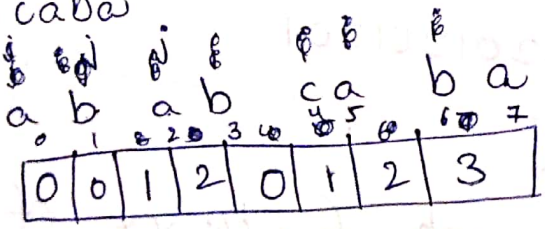
babab

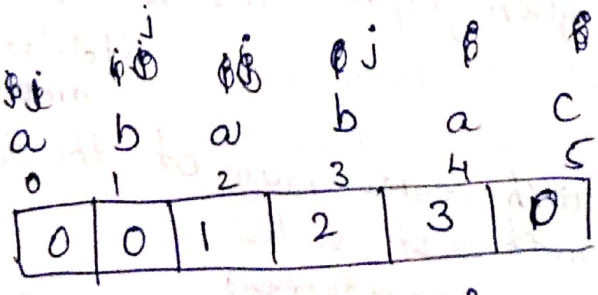
π { 0,0,1,2,3 }

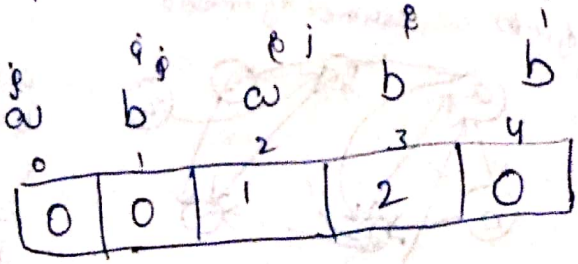
π[0] = 0
π[1] = 0
π[2] = 1
π[3] = 2
π[4] = 3

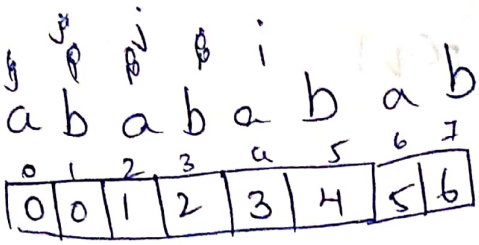
n = 11
m = 5
compute

ababcaba





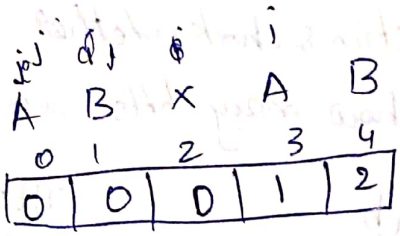


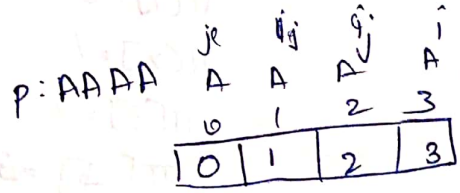


Compute the prefix function for the given pattern

P: ABXAB

ABXAB ABXAB



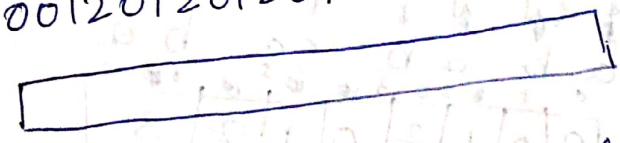


T: AAAABAAABA

Compute prefix function

ABABBABBABBA

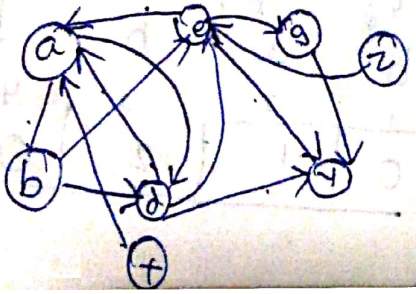
001201201201



~~Adj~~ For the given graph find the size of the graph (total no. of edges) irrespective of dir

degree of each vertex, sum of the degrees, no. of odd vertices, no. of even vertices (sum of deg = even)

6 ← odd
2 ← even

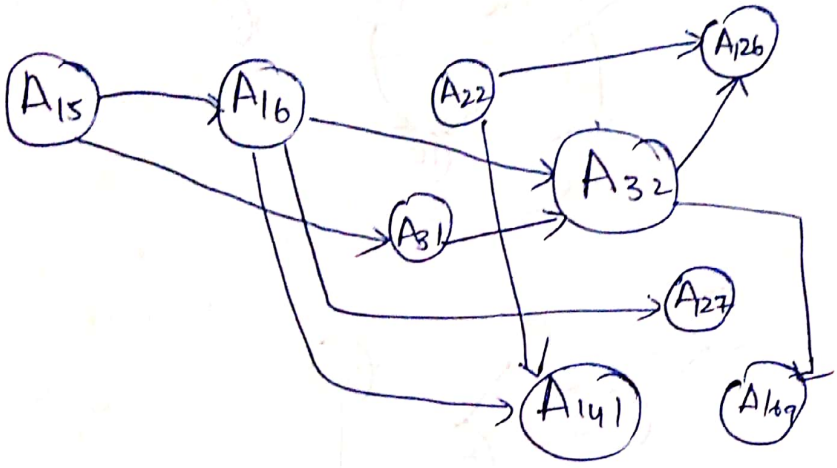


a = 3
b = 1
c = 2
e = 3
g = 1
y = 3
z = 3

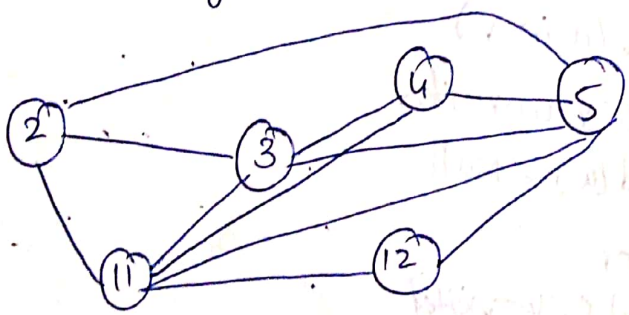
A student A wants to plan his course schedule to take the following 9 language courses they are given as follows:

$A_{15}, A_{16}, A_{22}, A_{31}, A_{32}, A_{126}, A_{141}, A_{127}, A_{169}$

The course prequite



Let G be a graph with $V(G) = \{2,3,4,5,11,12\}$. two vertices s and t are adjacat if and only gcd of s and t is 1 and draw the graph total no. of edges 11

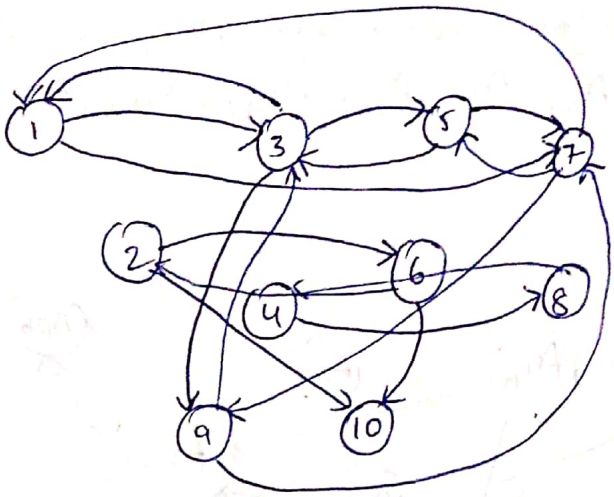


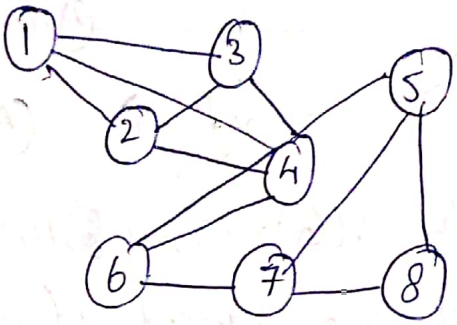
$V(G) = \{1,2,3,4,5,6,7,8,9,10\}$

Two vertices i and j are adjacat $i+j$ is an multiple of 4 (directed)

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

1 : (2,3,4)
2 : (1,3,4)
3 : (1,2,4)
4 : (1,2,3,6)
5 : (6,7,8)
6 : (4,5,7)
7 : (5,6,8)
8 : (5,7)





g G

BFS (branch & bound)

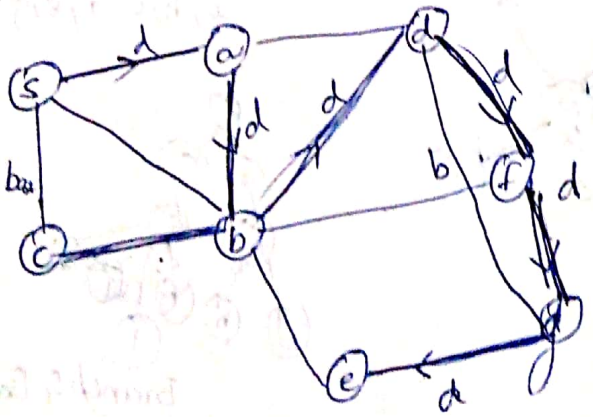
```
BFS(G,s)
for each(u in v)
  mark[u]←unvisited
    pred[u]←Null
end for
  mark[s]←visited
    Q={s}
while(Q is not empty)
      u←deque from head Q
for each v in Adj[u]
      if mark[v]=unvisited
        mark[v]←visited
Pred[v]←u
  included edge(u,v) in the graph
  append v to the tail of Q
end if
```

end for
mark[u] = finished
End while

DFS (backtracking)
dfs(v)
visit(v)
for each neighbour
N of v
if N is unvisited
dfs(N)
add edge vN to tree t
end if
end for

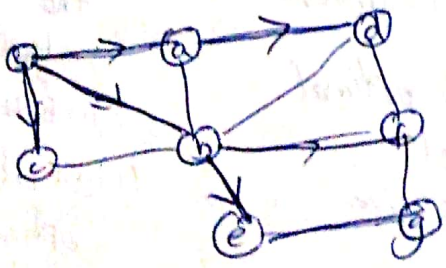
Graph traversal

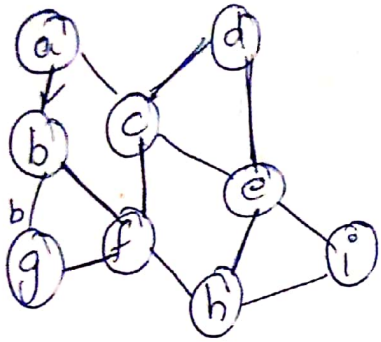
It is a systematic procedure for exploring a graph by examining all its vertices & edges. Search engine the documents are considered to the vertices and links are to be edges.



s, a, b, c, d, f, g, e

s, a, b, c, d, e, f, g



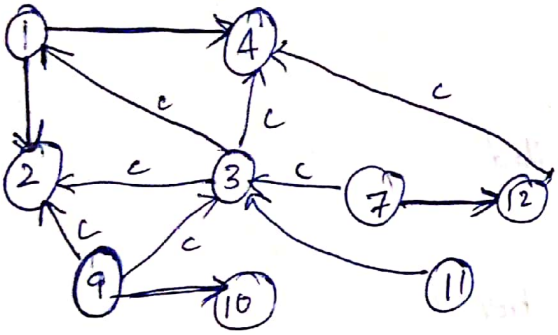


DFS

a, b, f, g, c, d, e, h, i,

BFS

a, b, c, g, f, d, e, h, i



1, 2, 14

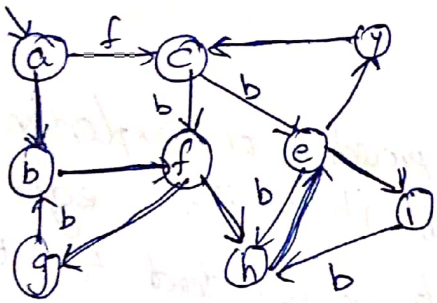
3

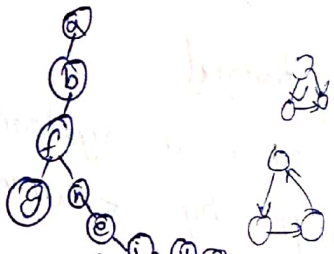
7, 12

9, 10

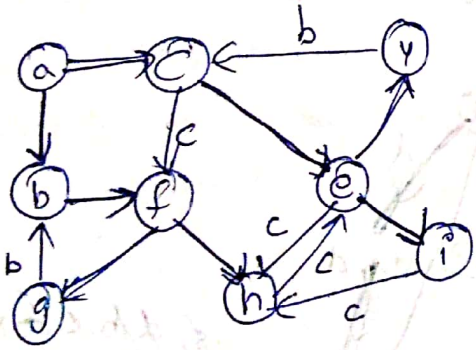
11

BFS & DFS to find whether a graph is forest

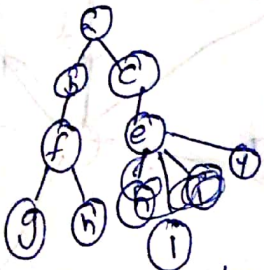




a, b, f, g, h, e, i, y, c



a, b, c, f, e, g, h, i, y



Backtracking

i) It is used to find all possible solution to given problem
Feasible is either optimal or non-optimal

ii) DFS is used

iii) It will not always give optimal solution

Branch & Bound

i) Used to solve only the optimization algorithm.

ii) Both DFS & BFS is used

iii) It always gives you an optimal solution.

DFS

In DFS space complexity is less

Uses stack as data structure

$O(V+E)$

DFS gives multiple trees different feasible solutions

It produce a spanning tree where most of the edges are back edges.

It gives a sub-optimal/feasible solution

Graph traversal
Maze problem.

BFS

In BFS space complexity is more

Uses queue as data struct

$O(V+E)$

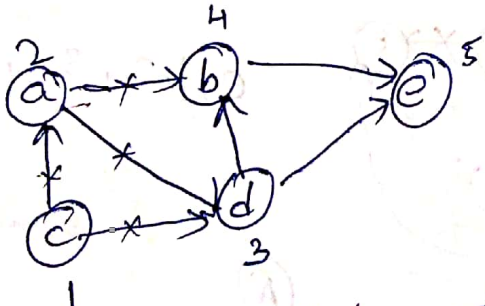
Produces one tree if graph is connected

It produces a spanning tree such that most of the cross edges

It gives an optimal solution

Peer to peer connectivity
Search engine

Topological sorting:-
G should be a directed acyclic graph. Note down the vertices which does not have incoming edges.

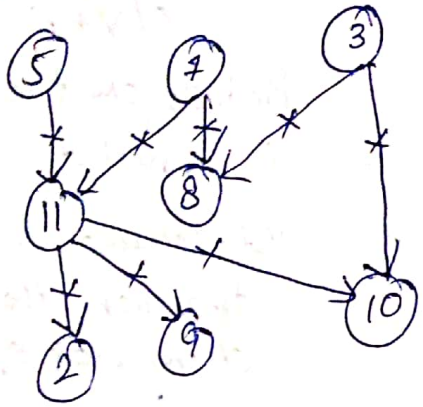


cadbe

L is an empty list that contain sorted elements
S is the set of all nodes with no incoming edge while S is non-empty do remove a node n from S and add n to the tail of L for each node m with an edge e from n to m do remove edge e from graph and if has m to nother edges insert m into S.

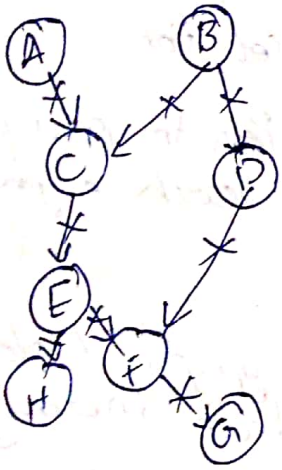
if graph has edges then return error/ graph has atleast one cycle)

else return L

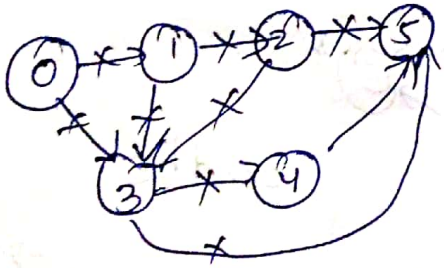


3, 5, 7, 8, 11, 2, 9, 10

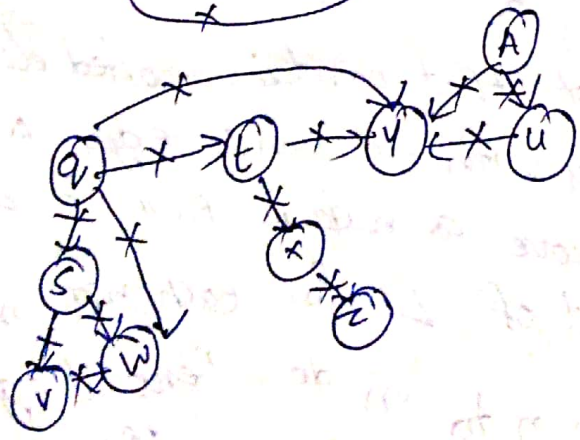
~~6×6×2~~



A. B C D E F G H



0, 1, 2, 3, 4, 5



A q s t u v w x y z

Dijkstra's -Algorithm

Dijkstra's ShortestPath Algorithm(Q, v)

D[v] ← 0

for each vertex u ≠ v of G do

D[u] ← ∞

Let a priority queue Q contains all the vertices of G using D values as key while queue is not empty do

u ← Q.removemin()

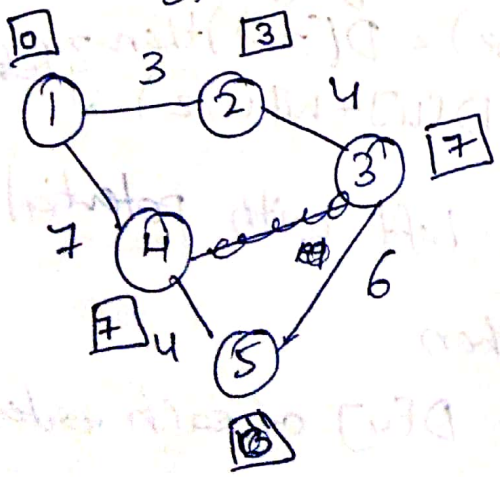
for each vertex z adjacent to u such that z is queue do

if [D[u] + w[u,z) < D(z)]

then

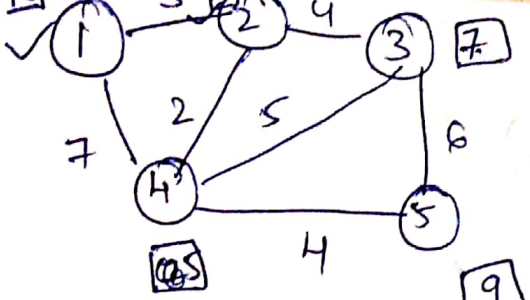
D(z) ← D(u) + w(u,z)

Return the label D[u] of each vertex u

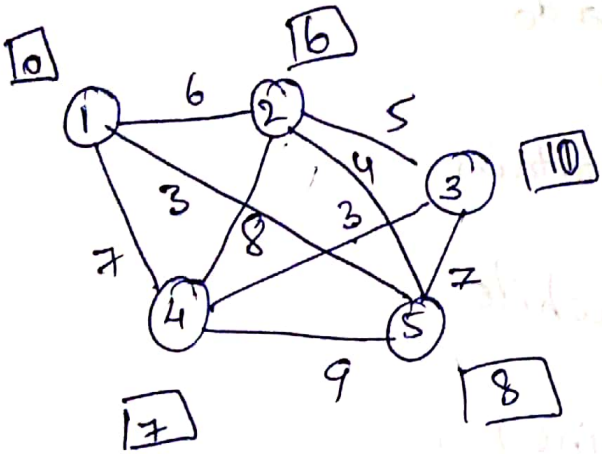


① 0
② 3
③ 7
④ 7
⑤ 11

O(E + log V)



1 0
2 3
3 7
4 5
5 9



1 0
2 6
3 10
4 7
5 3

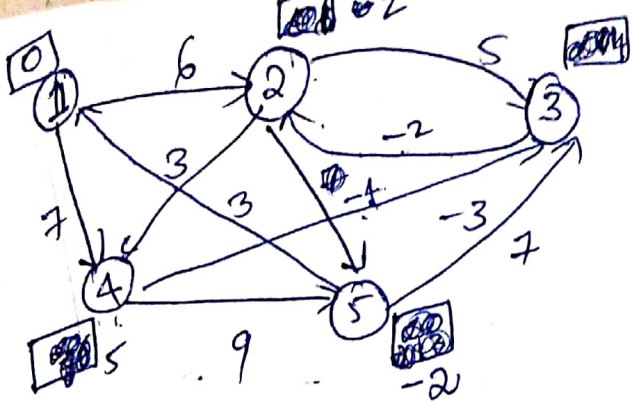
Bellmen ford shortest path Algorithm:- $O(EV)$

```
D[v] = 0
for each vertex u ≠ v of G do
    D[u] ← ∞
for i ← 1 to n-1 do
    for each Edge (u,z) outgoing from u
        if [D[u] + w(u,z) < D[z]] then  { Relaxation
            D[z] ← D[u] + w(u,z) }
```

If there are no edges left with potential relaxation operation then

return label D[u] of each vertex

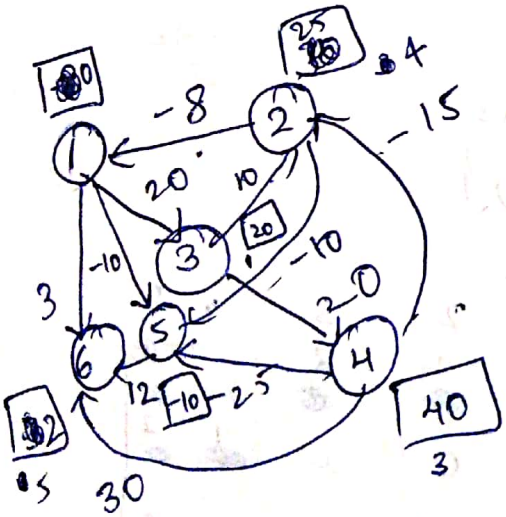
else return G contains a negative weight cycle



change (2,4) → 8

outgoing edges

	II	III	IV
✓(1,2)	✓(1,2)	✓	✓
✓(1,4)	✓(1,4)	✓	✓
✓(2,3)	✓(2,3)	✓	✓
✓(2,4)	✓(2,4)	✓	✓
✓(2,5)	✓(2,5)	✓	✓
✓(3,2)	✓(3,2)	✓	✓
✓(4,3)	✓(4,3)	✓	✓
✓(4,5)	✓(4,5)	✓	✓
✓(5,1)	✓(5,1)	✓	✓
✓(5,3)	✓(5,3)	✓	✓

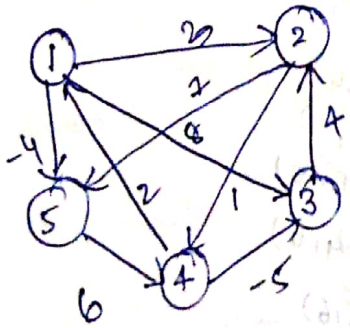


✓(1,3)
✓(1,5)
✓(1,6)
✓(2,1)
✓(2,5)
✓(3,2)
✓(3,4)
✓(4,2)
✓(4,5)
✓(4,6)
✓(5,6)

Floyd Warshall Algorithm:-

```
for i ← 1 to n do
for j ← 1 to n do
  if i=j then
    D[i,j] = 0
if (vi, vj) is an edge in G⃗ then
    D[i,j] = w(vi, vj)
  else
    D[i,j] = ∞
for k ← 1 to n do
for i ← 1 to n do
for j ← 1 to n do
D[i,j] = min{D[i,j], D[i,k] + D[k,j]}
    return matrix D^n
```

$$d_{ij}^{3} = \begin{cases} w_{ij} & k=0 \\ \min\{d_{ij}^{2}, d_{i3}^{2} + d_{3j}^{2}\} & \end{cases}$$



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

D^1

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

D_2

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

D_3

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

k = 3

D_4

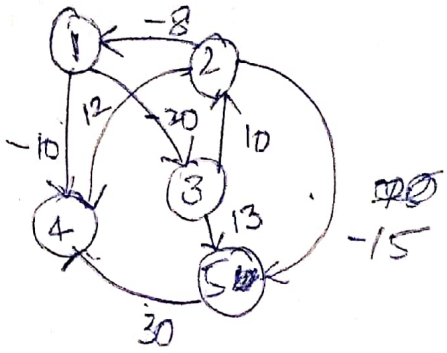
	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

i = 3 j = 5
k = 4

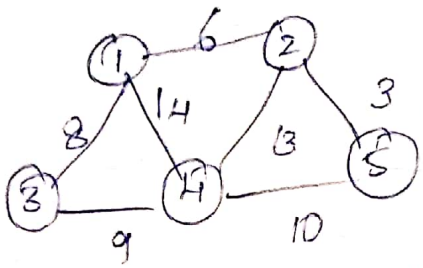
D_5

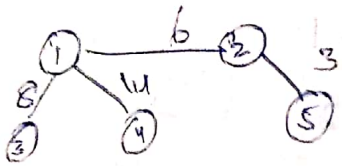
	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$O(V^3)$ complexity

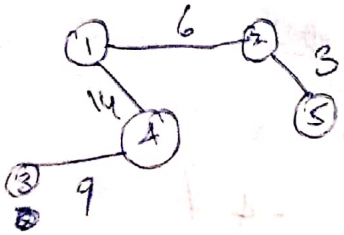


All vertices should be connected without forming the cycle and with minimum cost is known as minimum spanning tree.



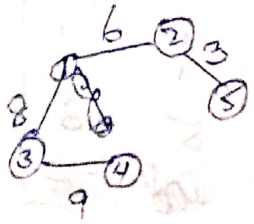


31



32

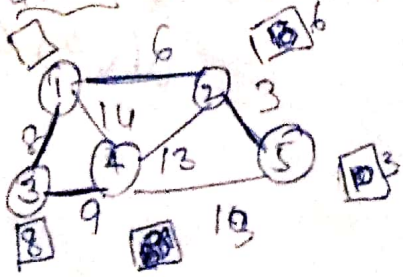
$E \log E + E(V+E)$



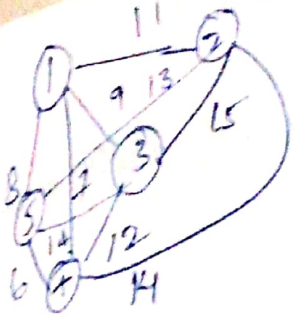
26

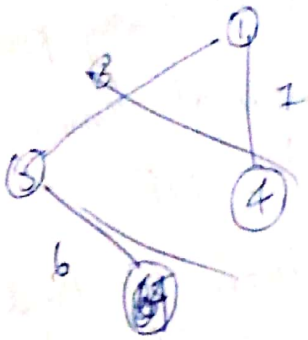
Prim's Algorithm

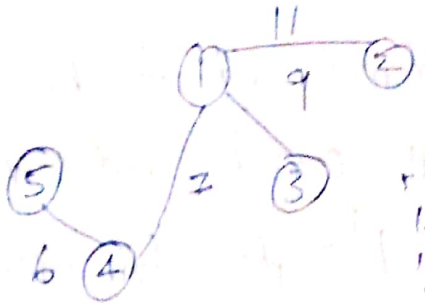
$O(V+E)$



$O(E \log E)$

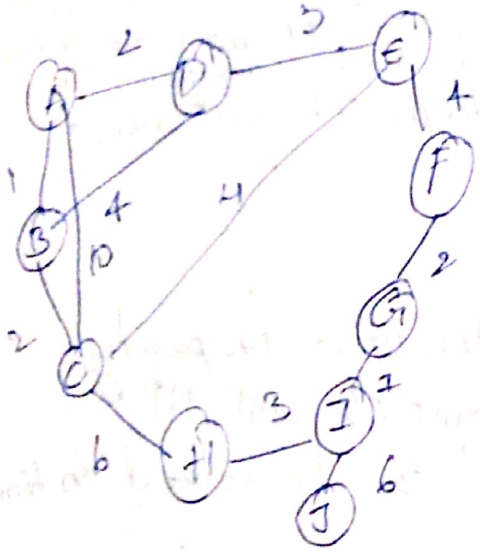


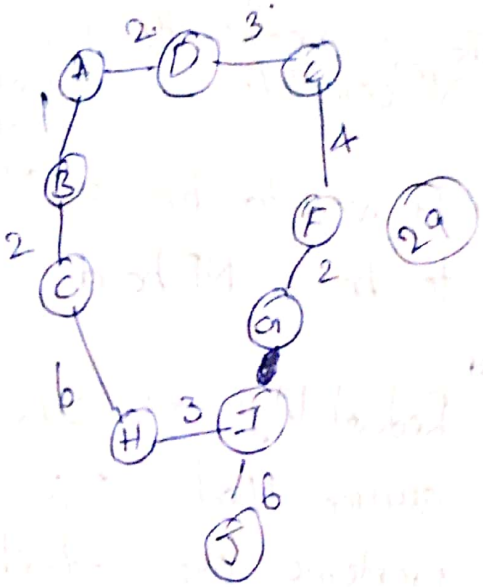




33

13
11
9
33





29

NP problems

Class P problems consists of the problems that are solved in polynomial time. They are the problems which can be solved in n^k, where k is constant. n is the size of input to the problem. (MST problem)

The class NP:-

NP means non-deterministic polynomial. NP means only verified in polynomial time.

Verifiable:

If you are given a certificate of a solution we can verify the legitimacy in polynomial

Time (Verify yes answer in polynomial time)

Problem of NP can be verified using non-deterministic turing machine.

A problem is said to be NP-Hard if every problem of NP can be reduced in polynomial time.

$$L' <=_p L \quad \text{for every} \quad L' \in NP$$

It is easy to show NP rather than NP-Hard

The '<' means the time taken to solve L' is not more that a polynomial factor away from the time taken to solve L. L' is reduced in L time.

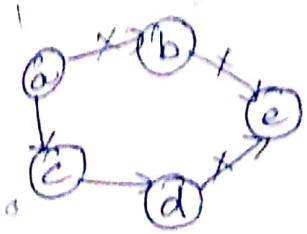
NP Complete :-

It need to be a NP problem. It also need to be a NP-hard

Reducability:- It follows the transitive property assume that P is NP complete. All NP problems are reducible, L' can be solved in time which is less than L.

NP Problems Examples:-

1. TSP (Euler cycle)
2. CPP (Hamiltonian cycle)
3. Chromatic number (Graph colouring)

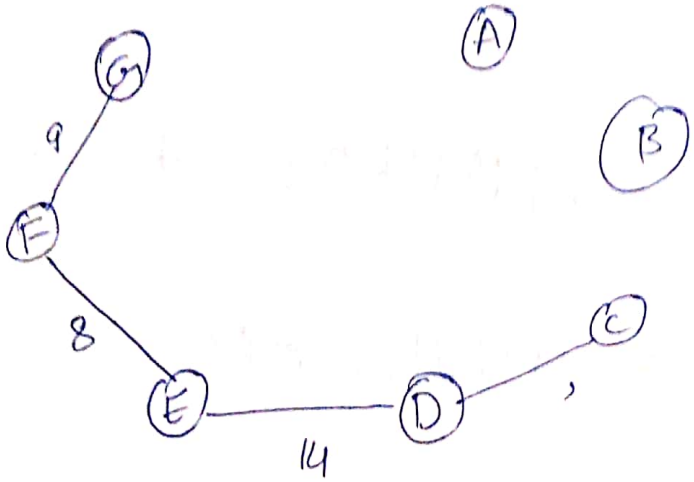


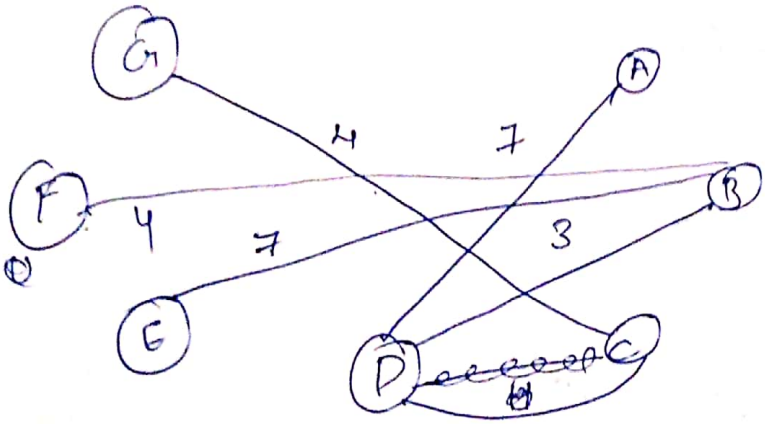
3)
bcd ✓
bdcx
cbd ✓
cdb ✓
dcb x
dbcx

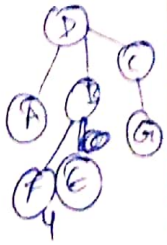
acdbe ✓
acbde ✓
abcde ✓

a - - - - e
b
d

e

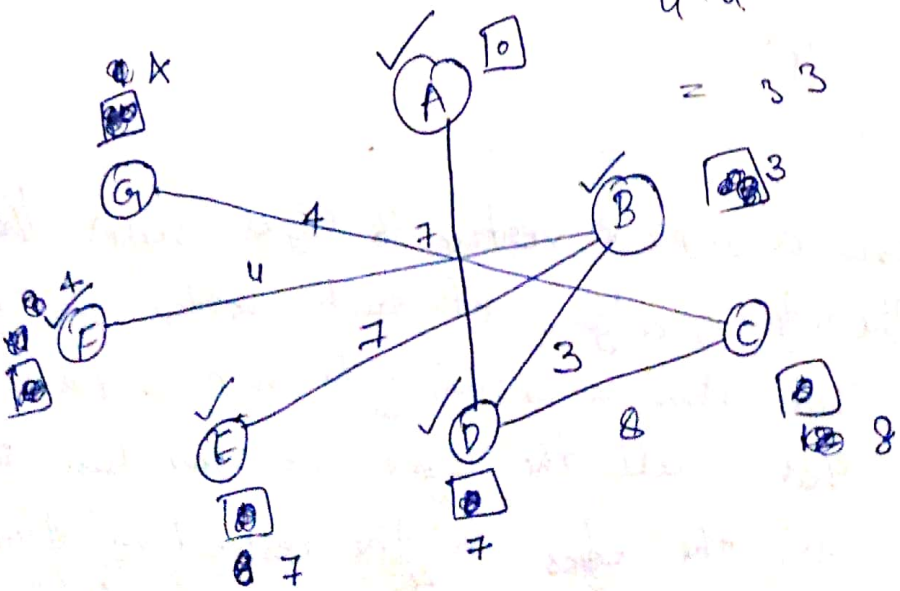


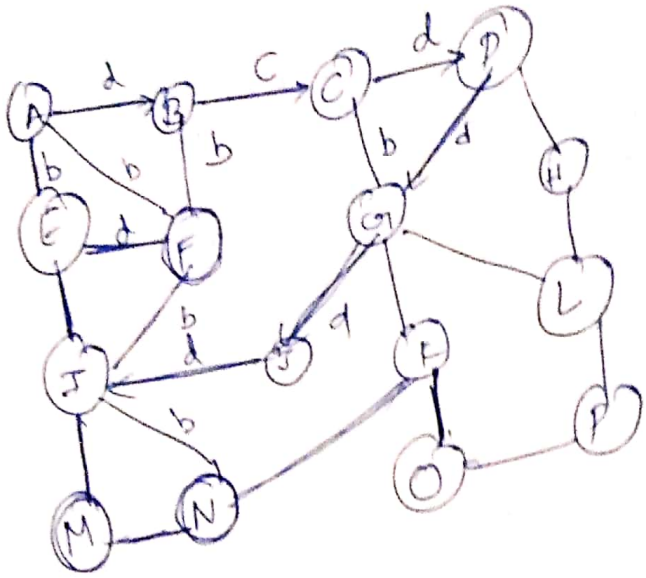




4+4+7+7+3+8

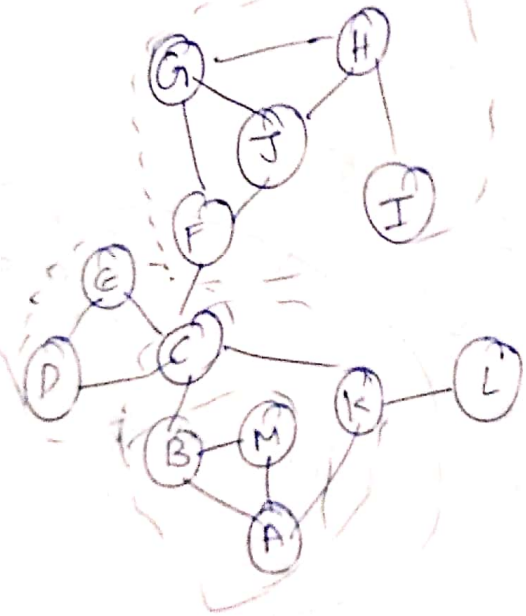
= 33





DFS → A B C D G J I E F M N K O P L H

BFS → A B E F C I D G J M N H K L O P



Draw a simple vertex 16 edges such that The indegree & outdegree of each vertex 2. Sho that there is a simple cycle (non-simple) co that all the edges i.e you can trav all the edges in the respective directio without ever lifting your pencil. such an

cycle is called (Euler II)