Date.....

Design and Analysis of Algorithm (BCS503)

Unit-1:- Introduction:- Algorithm, Ala Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurments, Sorting and Order statistics,- Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparisson of Sorting Algorithms, Sorting in Linear Time.

Unit-2:- Advanced Data Structures:- Red-Black Trees, B-Trees, Binomial Heaps, Fibonacci Heaps, Tries, Skip List.

Unit-3:- Divide and Conquer:- with Examples Such as Sorting, Matrix Multiplication, Convex Hull and Searching.

Greedy Methods:- with Examples Such as Optimal Reliability Allocation, Knapsack, Minimum Spanning Trees- Prim's and Kruskal's Algorithm, Single Source Shortest Paths - Dijkstra's and Bellman Ford Algorithms.

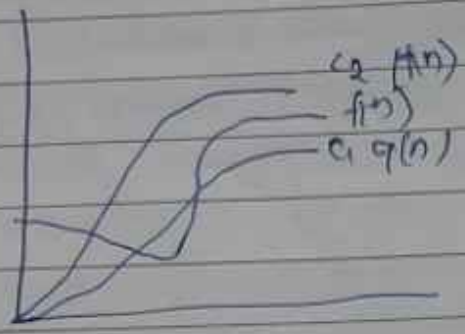
Unit-4:- Dynamic Programming with Examples Such as Knapsack. All pair Shortest Paths - Warshal's and Floyd's Algorithms, Resource Allocation Problem. Backtracking, Branch and Bound with Examples Such as Travelling Salesman Problem, Graph Colouring, n-Queen Problem, Hamiltonian Cycles and Sum of Subsets.

Teacher's Sign.....

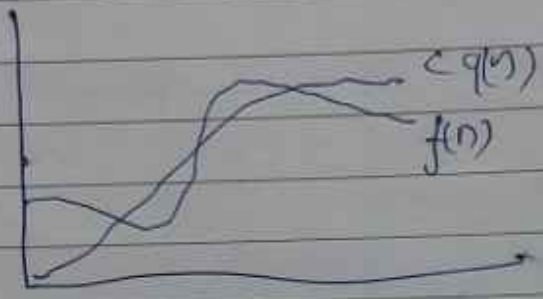
Date.....

Unit-5 :- Selected topics : Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NP-Completeness, Approximation Algorithms and Randomized Algorithms.

Θ notation :

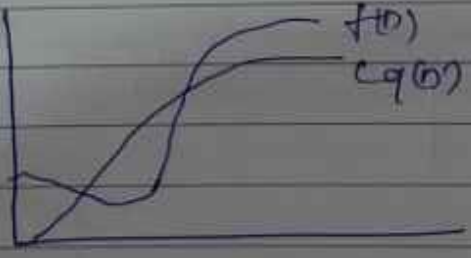


Θ notation :



$0 \leq f(n) \leq (g(n)$

Omega



$0 \leq c_1 g(n) \leq f(n)$

13/08/18

Algorithm

An algorithm is a finite set of instructions that if followed accomplishes a particular task.

Algorithms are the ideas behind computer programs.

All algorithm must satisfies the following criteria

1) Input
2) Output
3) Definiteness
4) finiteness
5) Effectiveness

1) **Input** :- To check the performance of the algorithm first we supply the input to the designed algorithm. Input should be small or large it depends on application

2) **Output** :- At least one quantity is produced.

3) **Definiteness** :- Each instruction is clear and unambiguous.

4) **finiteness** :- If we trace out the instructions of an algorithm then for all cases the algorithm terminates after a finite no. of steps

5) Effectiveness :- Every instruction must be a very basic so that it can be carried out and any person can be easily understand the designed algorithm.

Analysing Algorithms.

For analysing algorithms we assume a RAM model { Random access machine } and algorithm will be implimented as computer programs.

In this instructions are executed one after another with no concurrent operations.

Analysing an algorithm is concerned of space and time complexity.

Space complexity

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

Time complexity

Time complexity is amount of time it needs to run to completion

Space complexity

The space complexity can be defined as amount of memory required by an algorithm to run.

To compute the space complexity we use two factors constant and ~~instant~~ Instance characteristics.

The space S(P) requirement can be given as

$$S(P) = C + S_p$$

where,

C is constant i.e fixed part and it denotes the space of inputs & outputs.

This space is an amount of space taken by instruction, variables.

S_p is a space dependent upon instance characteristics.

This is a variable part whose space requirement depends on particular problem instance.

Time complexity.

The time complexity of an algorithm is

the amount of computer time required by an algorithm to run to completion.

It is difficult to compute the time complexity in terms of physically clocked time. For instance in multi-user system executing time depends on many factors such as 1) system load
2) No. of other programs running
3) Instruction set used.
4) Speed of underlin hardware.

Frequency count is a count denoting no. of times of execution of statement

Obtain the frequency count for the following code.

```
void fun()
{
  int a;
  a=0;                    ---> 1
  for(i=0; i<n; i++)      ---> n+1   {Because one condition false also}
  {
    a = a+i;              ---> n
  }
  printf("%d", a)         ---> 1
}
```

frequency count = 1+n+1+n+1

= 2n+3

frequency count = 2n+3.

Growth of functions

Measuring the performance of an algorithm in relation with the input size n is called order of growth.

Ex.

The order of growth for varying input size of n is as given below

n	log n	n log n	n^2	2^n
1	0	0	01	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4294967296

Fig: Order of Growth

From the above table it is clear that the logarithmic function is the slowest

growing function and the exponential function 2^n is fastest and grows rapidly with varying input size n.

The exponential function gives huge value even for small input n for instance for the value of $n=16$ we get 65536

Ques. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than the algorithm whose running time is 2^n on the same machine

Ans. As $n=15$ then 2^n exceeds $100n^2$ hence $n=15$ is the smallest value satisfying the given condition

$n=15$

$100(15)^2 < 2^{15}$

$22500 < 32768$

Asymptotic notation.

To choose the best algorithm we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm.

Asymptotic is a shorthand way to represent the time complexity. Using Asymptotic notation we can give time complexity "fastest possible, (slowest possible) or average time"

Various notation such as Ω, Θ, O omega, theta and Big O notation are called as asymptotic notation.

Big Oh notation.

The Big oh notation is denoted by 'O' It is a method of representing the upper bound of algorithms running time. Using Big Oh notation we can give longest amount of time taken by an algorithm to complete.

Definition :- Let $f(n)$ and $g(n)$ be two non-negative functions. Let n_0 and constant c are two Integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is

some constant such that $c > 0$.

We can write.

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

Then $f(n)$ is big oh of $g(n)$. It is also denoted as $f(n) \in O(g(n))$

In other words $f(n)$ is less than $g(n)$ if $g(n)$ is multiple of some constant c.

Consider function $f(n) = 2n+2$ and $g(n) = n^2$ Then we have to find some constant c so that $f(n) \leq c * g(n)$

for $n = 1$

$f(n) = 4$

$g(n) = 1$

$f(n) > g(n)$

for $n = 2$

$f(2) = 2 \times 2 + 2$

$= 6$

$g(2) = 2^2$

$g(2) = 4$

$f(2) > g(2)$

for $n = 3$

$f(3) = 6 + 2$

$= 8$

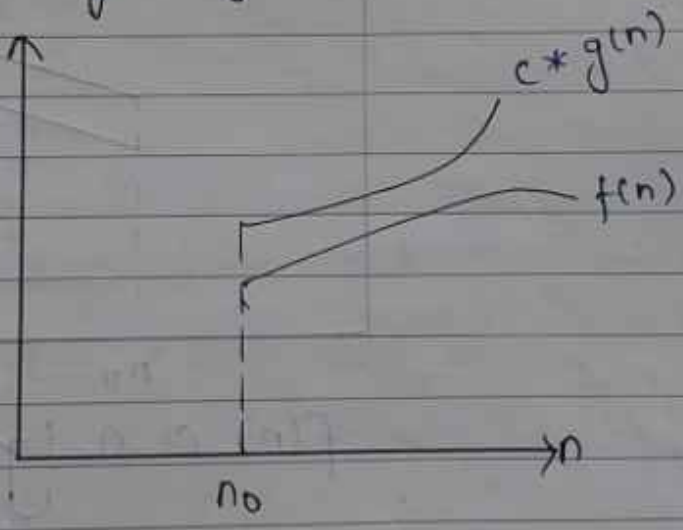
$g(3) = 3^2$

$= 9$

$\therefore f(n) < g(n)$ True

Hence we can conclude that for $n > 2$ we obtain $f(n) < g(n)$

Thus always upper bound of existing time is obtained by big Oh Notation



$f(n) \in O(g(n))$

2) Omega (Ω) Notation

Omega Notation is denoted by Ω. This Notation is used to represent lower bound of algorithms running time. Using omega notation we can denot shortest

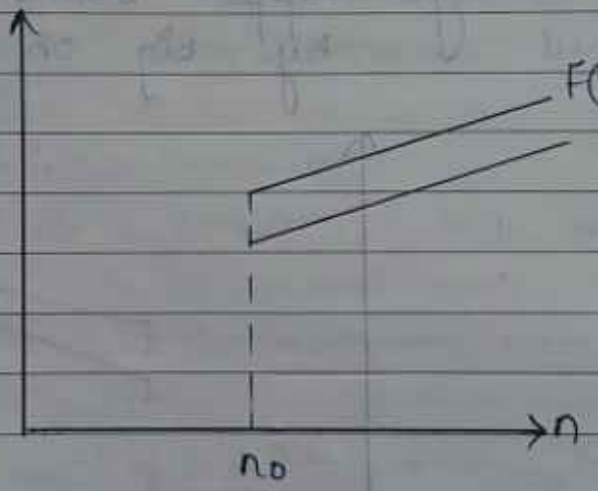
amount of time taken by algorithms

Definition

A function $f(n)$ is said to be in omega $\Omega(g(n))$ if $f(n)$ is bounded below by some positive constant multiple of $g(n)$ such that

$$f(n) \geq c * g(n) \quad \text{for all } n \geq n_0$$

It is denoted as $f(n) \in \Omega(g(n))$



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

Consider $f(n) = 2n^2+5$ and $g(n) = 7n$

for $n=0$

$f(0) = 2\times0+5$

$f(0) = 5$

$g(0) = 0$

Here $F(n) > g(n)$

for n=1
F(1) = 7
g(1) = 7
for n=2
F(2) = 13
g(2) = 14

for n=3
F(3) = 23
g(3) = 21
F(n) > g(n)

It can be represented as $2n^2+5 \in \Omega(n)$ and $n^3 \in \Omega(n)^2$

3) Theta Notation The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

Definition

Let f(n) and g(n) be two non-negative functions. There are two positive constants C_1 and C_2 such that

$$C_1 \leq g(n) \leq C_2 g(n)$$

Then we can say that

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

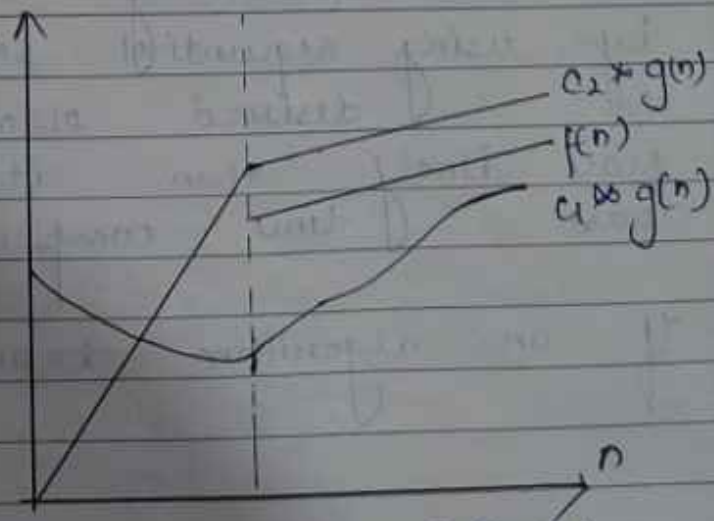


Fig: Theta notation $f(n) \in \Theta g(n)$

Example If $f(n) = 2n + 8$ and $g(n) = 7n$ when $n \geq 2$

Sol $f(n) = 2n + 8$.

for $n = 2$.

$f(2) = 2 \times 2 + 8$

$f(2) = 12$

$g(2) = 14$.

* Best case, Worst case and average case analysis

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case time complexity

Eg While searching a particular element by using sequential search we get the desired element and first play itself then it is called best case time complexity.

If an algorithm takes maximum amount

of time to run to completion for a specific set of input then it is called worst case time complexity.

Ex

while searching an element by using linear searching method if desired element is placed at the end of the list then it is called worst case time complexity

The time complexity that we get for certain set of inputs is same then for corresponding input such complexity is called average case time complexity.

* Amortized Analysis.

Amortized time cannot be computed within one execution of an algorithm rather over a sequence of operations we compute the time complexity there may be a situation that a single operation is expensive but the time taken by sequence of n operations can be better than worst case time..

4 Amortized analysis means finding the average running time per operation over a worst case sequence of operation

Recurrence relation.

The recurrence equation is an equation that defines a sequence recursively it is normally in the following form—

$$T(n) = T(n-1) + n \quad \text{for } n > 0 \quad \text{—(1)}$$

$$T(0) = 0 \quad \text{—(2)}$$

Here eqn (1) is called recurrence relation & eqn (2) is called initial condition.

The recurrence eqn can have infinite (∞) no. of sequences.

The general solution to the recursive function specifies some formula.

Ex.
Consider a recurrence relation

$$f(n) = 2f(n-1) + 1 \quad \text{for } n > 1$$

$$f(1) = 1 \quad \text{—}$$

then by solving these recurrence relation we get $f(n) = 2^n - 1$ when $n = 1, 2, 3, 4$

The recurrence relation can be solved using three methods -

1) Substitution method.
2) Recurrence tree method.
3) Master's theorem or Master's method

1) Substitution method

The substitution method is a kind of method in which a guess for the solution is made.

There are two types of substitution

1. Forward substitution
2. Backward substitution.

1) Forward substitution

This method makes use of an initial condition in the initial term and value for the next term is generated.

This process is continued until some formula

is guessed. Thus in this kind of substitution method we use recurrence eqn to generate few terms.

Ex. Consider a recurrence relation
$T(n) = T(n-1) + n$ with initial condition $T(0) = 0$

Sol

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

If $n = 1$

$T(1) = T(1-1) + 1$
$= T(0) + 1$
$= 0 + 1$
$T(1) = 1$

If $n = 2$.

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

If $n = 3$.

$T(3) = T(3-1) + 3$
$= T(2) + 3$
$= 3 + 3$
$T(3) = 6$. , By observing above generating eqns, we can derive a formula,

$$\boxed{T(n) = \frac{n(n+1)}{2}} \text{ or } \boxed{T(n) = \frac{n^2}{2} + \frac{n}{2}}$$

2) Backward Substitution.

In this method backward values are substituted recursively in order to derive some formula.

Ex.

Consider a recurrence relation

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

with initial condition $T(0) = 0$.

Sol

If $n = n-1$

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

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

Put eqn (2) in (1)

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

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

Now for $n = n-2$

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

Put eqn (4) in eqn (3)

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

$$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \dots + n$$

If $k = n$ then

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

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

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

* Solve the following recurrence relation

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

with $T(0) = 0$ as initial condition by using Backward Substitution.

Sol

If $n = n-1$

$$T(n-1) = T(n-1-1) + 1$$
$$T(n-1) = T(n-2) + 1 \quad \text{---(2)}$$

for $n = n-2$

$$T(n-2) = T(n-2-1) + 1$$
$$T(n-2) = T(n-3) + 1 \quad \text{---(3)}$$

Put eq^n (2) in eq^n (1)

$$T(n) = T(n-2) + 1 + 1 \quad \text{---(4)}$$

Put (3) in (4)

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

$$T(n) = T(n-k) + k$$

for $k = n$, then

$$T(n) = T(n-n) + n$$
$$= T(0) + n$$

$$\boxed{T(n) = n}$$

Ques. Solve the following recurrence

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

$T(1) = 1$

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

$T(1) = 1$

Sol 1) $T(n) = 2T\left(\frac{n}{2}\right) + c$ as $T(n) = 2T\left(\frac{n}{2}\right) + c$.

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

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

$$= 4T\left(\frac{n}{4}\right) + 3c$$

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

$$T(n) = 8T\left(\frac{n}{8}\right) + 7c$$

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

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)\,c$$

If $2^k = n$ Then

$$T(n) = nT\left(\frac{n}{n}\right) + (n-1)\,c$$

$$T(n) = nT(1) + (n-1)\,c \quad \text{as } T(1) = 1$$
$$T(n) = n + (n-1)\,c$$

Sol 2. as $T(n) = T\left(\frac{n}{3}\right) + c$

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

$$T(n) = T\left(\frac{n}{9}\right) + 2c$$

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

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

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

$3^K = n$.

$K = \log_3 n$.

$$T(n) = T\left(\frac{n}{n}\right) + \log_3 n\, C$$

$$= T(1) + \log_3 n\, C$$

as $T(1) = 1$

$$T(n) = C\log_3 n + 1$$

Ques. Solve the following recurrence exactly.

$$t_n = \begin{cases} n & \text{if } n = 0, 1, 2 \text{ or } 3 \\ t_{n-1} + t_{n-3} + t_{n-4} & \text{otherwise} \end{cases}$$

Express your answer using the Θ notation.

Sol Let $t(n) = t(n-1) + t(n-3) - t(n-4)$ be a relation.

for $n = 1$

$t(1) = t(0) + t(-2) - t(-3)$

$= t(0) + t(2) + t(3)$

$= 0 - 2 + 3$

$= 1$

if $n = 2$

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

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

$= 1 - 1 + 2$

$= 2$

If $n = 3$

$t(3) = t(2) + t(0) - t(-1)$

$= t(2) + t(0) + t(1)$

$= 2 + 0 + 1$

$= 3$

if $n = 4$, $t(4) = t(3) + t(1) - t(0)$

$= 3 + 1 - 0$

$= 4.$

$t(k) = t(k-1) + t(k-3) - t(k-4)$

$t(k) = k$

if $k = n$

$\boxed{t(n) = n}$

Recurrence Tree Method.

The recurrence relation can also be solved using tree method. In this method a recursion tree is built in which each node represents the cost of a single subproblem in the form of recursive function invocations then we sum up the cost at each level to determine the overall

cost. (The recursion tree method helps us to make a good guess of the time complexity)

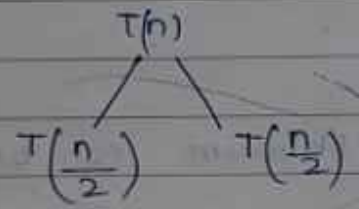
Ques. $T(n) = 2T\left(\frac{n}{2}\right) + n$

$T(1) = O(1)$

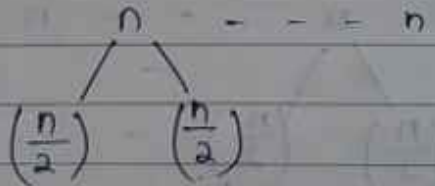
Soln

Step 1. $T(n)$

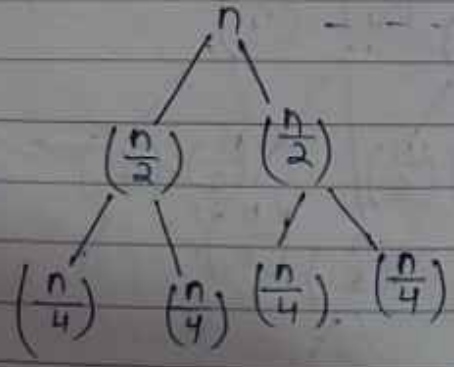
Step 2.



Step 3



Step 4.



Master's Theorem

We can solve recurrence relation using a formula denoted by master's method.

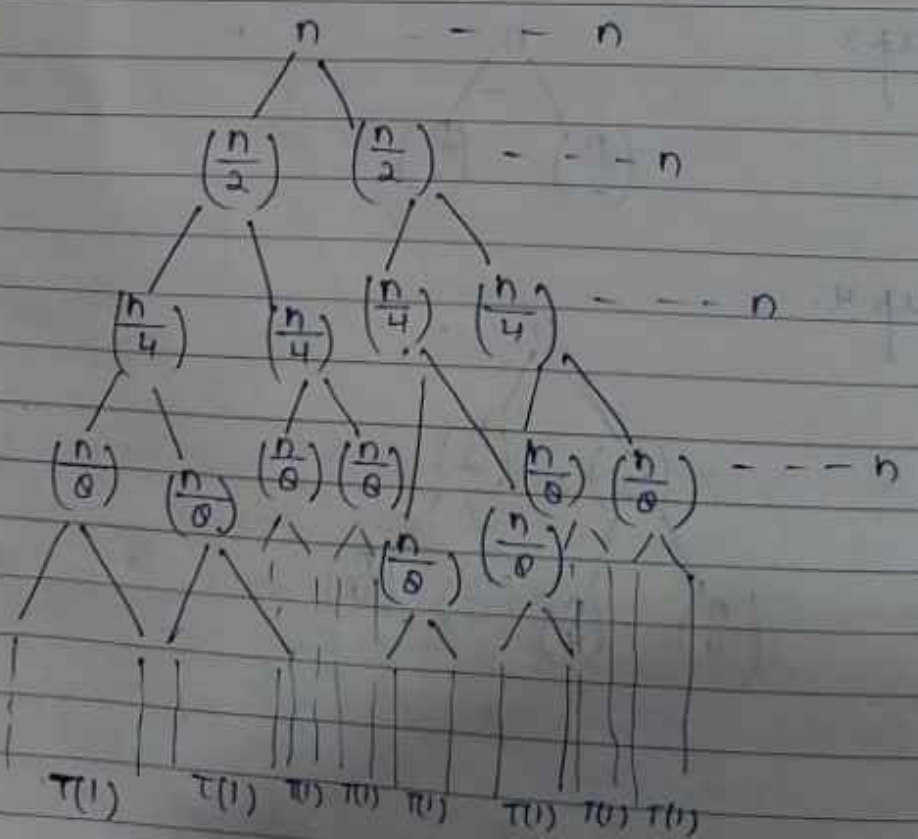
$$T(n) = aT\left(\frac{n}{b}\right) + F(n)$$

where, $n \geq d$ and d is some constant.

Some steps are.

Then the master theorem can be.

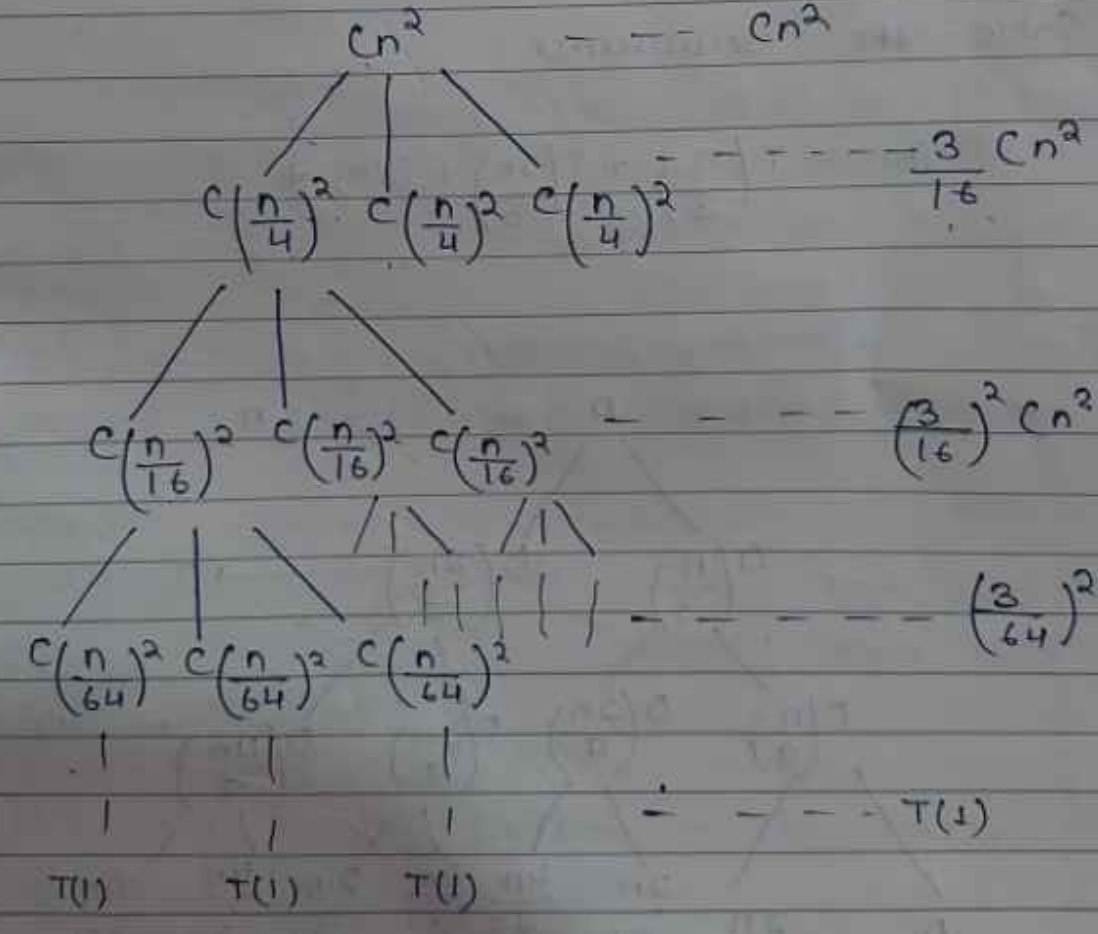
Step 5.



The depth of the tree is $\log n$. Hence we can guess that total cost $= n \log n\, T(1)$. Hence we can get the overall cost as $O(n \log n)$

Solve the recurrence relation

$$T(n) = 3T\left(\frac{n}{4}\right) + Cn^2$$ using tree method.



Thus the size for a node at depth i is $\frac{n}{4^i}$

If $n = 1$ then $\frac{n}{4^i} = 1$ i.e, $n = 4^i$ Hence $\log_4 n = 4^i$ where $i = (0, 1, 2, \ldots \log_4 n + 1)$

and cost of each node is $c\left(\frac{n}{4^i}\right)^2$

$\therefore$ $3^i c\left(\frac{n}{4^i}\right)^2$ is overall cost

$= \frac{3}{16} c(n^2)$

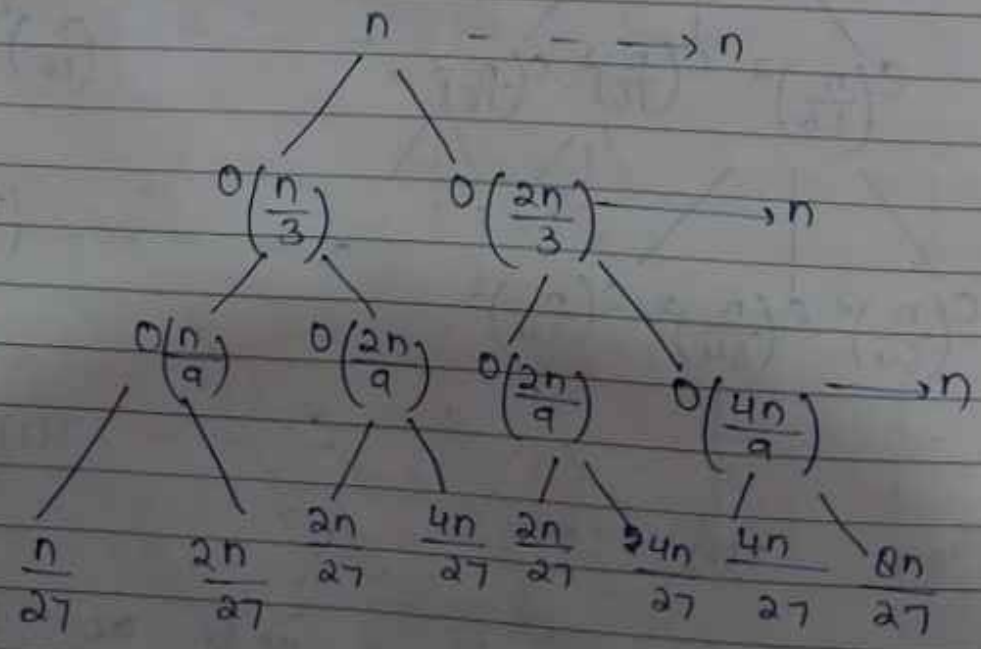
The last level at depth $\log_4 n$ has

$$3^{\log_4 n} = n^{\log_4 3} = \Theta\left(n^{\log_4 3}\right)$$

Solve the recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n) + \text{-----}$$

Sol



The longest path from root to a leaf is

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

i.e, $\left(\frac{2}{3}\right)^K n = 1$

If we assume $K = \log_{3/2} n$. Then the height of the tree is $\log_{3/2} n$.

The soln to recurrence = No. of levels * cost of each level

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

Master's Theorem.

We can solve recurrence relation using a formula denoted by masters method:

$$T(n) = aT\left(\frac{n}{b}\right) + F(n)$$

where $n \geq d$ and d is some constant.

Then the Master theorem can be stated for efficiency analysis as -

If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ in the recurrence relation, then -

Case 1:

$$T(n) = \Theta(n^d) \text{ if } a < b^d$$

Case 2:

$$T(n) = \Theta(n^d \log n) \text{ if } a = b$$

Case 3.

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

SORTING

Insertion Sort (A, N)

1. Set A[0] = -∞
2. Repeat step 3 to 5 for K = 2, 3, 4 - - - N
3. Set TEMP = A[K] and PTR = K-1
4. Repeat while TEMP < A[PTR]
   a) Set A[PTR+1] = A[PTR]
   b) PTR = PTR-1
5. Set A[PTR+1] = TEMP
6. RETURN

Bubble Sort — Sort (DATA, N)

1. Repeat step 2 and 3 for K-1 to N-1
2. Set PTR = 1
3. Repeat while PTR ≤ N-K
   a) If DATA[PTR] > DATA[PTR+1] then Interchange DATA[PTR] and DATA[PTR+1]
   b) Set PTR = PTR+1
4. Exit

Selection Sort (A, N)

1. Repeat step 2 and 3 for K = 1, 2 - - - N-1
2. Call MIN(A, K, N, LOC)
3. Set TEMP = A[K] , A[K] = A[LOC] and A[LOC] = TEMP
4. EXIT

MIN(A, K, N, LOC)

1. Set MIN = A[K] and LOC = K
2. Repeat for T = K+1, K+2 - - - - N

   If MIN > A[J] then set MIN = A[J] and LOC = J
3. Return

~~Heap Sort~~

Quick Sort.

1. If $n \leq 1$ then return
2. Pick any element v in A[] this is called pivot rearrange the elements $x_i > v$ to the right of v and all elements in $x_i < v$ is to left of v. If the place of v

after rearrangement is j then all the elements with value less than v appears in A[0], A[1], A[2] - - - A[j-1] and all values greater than v appears in A[j+1], A[j+2] - - - A[n-1].

3. Apply quick sort recursively to A[0] A[1] - - - [j-1] and to other half of j means A[j+1], A[j+2] - - A[n-1]

Entire array will be thus sorted by selecting an element v

a) Partion the array around v

b) Recursively sort left portion

c) Recursively sort right portion.

Sudocode.

Heap Sort

```
Heapsort(A) {
BuildHeap(A)
for i <- length(A) down to 2 {
    exchange A[1] <-> A[i]
    heapsize <- heapsize - 1
    Heapify(A, 1)
}
```

```
BuildHeap(A) {
heapsize ← length(A)
for i ← floor(length/2) down to 1
  Heapify(A, i)
}

Heapify(A, i) {
 le ← left(i)
 ri ← right(i)
 if (le <= heapsize) and (A[le] > A[i])
   largest ← le
 else
     largest ← i
 if (ri <= heapsize) and (A[ri] > A[largest])
   largest ← ri
 if (largest != i) {
 exchange  A[i] <-> A[largest]
  Heapify(A, largest)
  }
}
```

MERGE - SORT (A, p, r)

1. If p < r
2. q = [(p+q)/2]
3. MERGE - SORT (A, p, q)
4. MERGE - SORT (A, q+1, r)

5. MERGE(A, p, q, r)

MERGE(A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. let $L[1..n_1+1]$ and $L[1...n_2+1]$ be new arrays.
4. for $i = 1$ to n_1
5. $L[i] = A[p+i-1]$
6. for $j = 1$ to n_2
7. $R[j] = A[q+j]$
8. $L[n_1+1] = \infty$
9. $R[n_2+1] = \infty$
10. $i = 1$
11. $j = 1$
12. for $k = p$ to r
13. if $L[i] \leq R[j]$
14. $A[k] = L[i]$
15. $i = i + 1$
16. else $A[k] = R[j]$
17. $j = j + 1$

Quick Sort —

```
Quicksort(A, p, r) {
  if (p < r) {
    q <- Partition(A, p, r)
    Quicksort(A, p, q)
    Quicksort(A, q+1, r)
  }
```

5. MERGE(A, p, q, r)

MERGE(A, p, q, r)

```
1.  n1 = q - p + 1
2.  n2 = r - q
3.  let L[1..n1 + 1] and L[1...n2 + 1] be new arrays.
4.  for i = 1 to n1
5.      L[i] = A[p + i - 1]
6.  for j = 1 to n2
7.      R[j] = A[q + j]
8.  L[n1 + 1] = ∞
9.  R[n2 + 1] = ∞
10.     i = 1
11.     j = 1
12.     for k = p to r
13. if L[i] ≤ R[j]
14.     A[k] = L[i]
15.     i = i + 1
16. else A[k] = R[j]
17.     j = j + 1
```

Quick Sort

```
Quicksort(A, p, r) {
  if (p < r) {
    q <- Partition(A, p, r)
    Quicksort(A, p, q)
    Quicksort(A, q + 1, r)
  }
}
```

```
Partition (A, p, r)
  x <- A[p]
  i <- p-1
  j <- r+1
  while (true) {
  repeat
      j <- j-1
  until (A[j] <= x)
  repeat
      i <- i+1
  until (A[i] >= x)
  if (i <-= " " > A[j])
  else
      return(j)
  }
}.
```

Solve the following recurrence relation

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

We know that

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

On comparing, $a=4$, $b=2$, $f(n)=n$

$\Rightarrow n^1$ $\quad d=1$

Case 1. if $a < b^d \Rightarrow 4 < 2^1$; false

Case III If $a > b^d \Rightarrow 4 > 2$.

$$T(n) = \Theta(n^{\log_b a})$$

$$= \Theta(n^{\log_2 4})$$

$$\boxed{T(n) = \Theta(n^2)}$$

* Solve the following recurrence relation.

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

On comparing with.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a=2,\ b=2,\ f(n) = n\log n$

Case II $\because \quad a=b \quad , \ T(n) = \Theta(n^{\log}$

Cases : $T(n) = aT\left(\frac{n}{b}\right) + f(n)$: if $n \geq d$.

Case I.

If $f(n)$ is $O(n^{\log_b a - \epsilon})$

then $T(n) = \Theta(n^{\log_b a})$

Case II If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$ then

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

Case III

If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ then

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

Here $\log_2 2 = 1 = k$

Case II

$$T(n) = \Theta(n^{\log_2 2} \log^{1+1} n)$$

$$\boxed{T(n) = \Theta(n \log^2 n)}$$

Ques 1) $T(n) = 9T(n/3) + n$

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

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

Sol 1) $T(n) = aT\left(\frac{n}{b}\right) + F(n)$

on comparing with above formula

$a = 9$

$b = 3$

$F(n) = n$

$d = 1.$

Case III $a > b^d \Rightarrow 9 > 3^1$

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

$T(n) = \Theta(n^{\log_b a})$

$= \Theta(n^{\log_3 9})$

$= \Theta(n^{2\log_3 3})$

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

(2)

$a = 1$

$b = 3/2$

$F(n) = 1$

Case I. $a < b^d$

$1 < \frac{3}{2}^1$

Masters theorem

Let $a \geq 1$ and $b > 1$ be constants. Let $F(n)$ be a function and Let $T(n)$ be defined on the non-negative integers by the recurrence.

$$T(n) = aT\left(\frac{n}{b}\right) + F(n)$$

Where we interpret $\left(\frac{n}{b}\right)$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ then $T(n)$ can be bounded asymptotically as follows

Case I

If $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0$

then $T(n) = \Theta\left(n^{\log_b a}\right)$

Case II

If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$

Case III

If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

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

In each of the three cases we are comparing the function $f(n)$ with the function $n^{\log_b a}$

Intitively, the soln to the recurrence is determined by the larger of the two functions if as in Case I The function $n^{\log_b a}$ is the larger then the solution is

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

If as in case III the function $f(n)$ is

larger than the Sol^n is

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

If as in case II the functions are the same size we multiply by a logarithmic factor of the Sol^n is

$$T(n) = \Theta\left(n^{\log_b a} \log(n)\right)$$

Beyond this inductive thesis are some technicalities that must be understood In the first case not only must $f(n)$ be smaller than $n^{\log_b a}$ it must be polynomially smaller i.e. $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of (n^{ϵ}) for some constant $\epsilon > 0$

In the case III not only must $f(n)$ be larger than $n^{\log_b a}$ it must be polynomially larger and in addition satisfy the regularity condition

$$af(n/b) \leq cf(n)$$

This condition is satisfied by most of the polynomially bounded functions that we shall encounter.

Solve the recurrence relation by Masters method.

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

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

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

On comparing with —

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a = 8$, $b = 2$, $f(n) = n^2$, $d = 2$

Case III $\quad 8 > 2^2$, $a > b^d$

$$T(n) = \Theta(n^{\log_2 8})$$

$$\boxed{T(n) = \Theta(n^3)}$$

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

On comparing with

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a = 9$, $b = 3$, $f(n) = n^3$

Case I :- $\quad 9 < 3^3$ i.e. $a < b^d$

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

$$\boxed{T(n) = \Theta(n^3)}$$

Shell sort

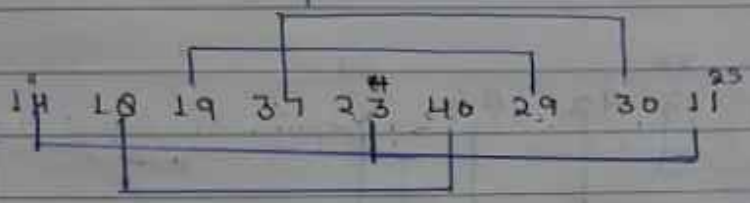
14 18 19 37 23 40 29 30 11

Pass 1 N = 9

gap = Floor $\left(\frac{N}{2}\right)$

$= \frac{9}{2}$

$= 4.5 \Rightarrow$ gap = 4



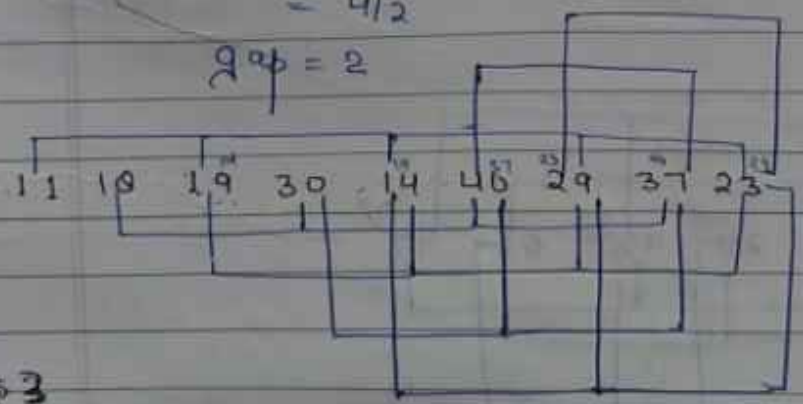
11 18 19 30 14 40 29 30 23

Pass 2

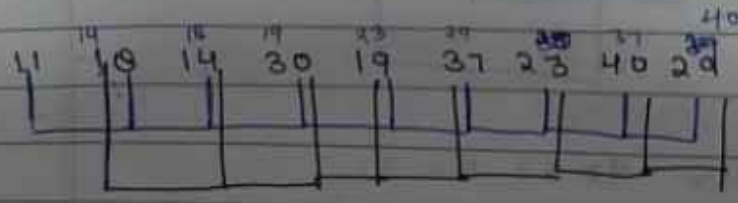
gap = Floor (gap/2)

$= 4/2$

gap = 2



Pass 3



Sorted array

11 14 18 19 23 29 30 37 40

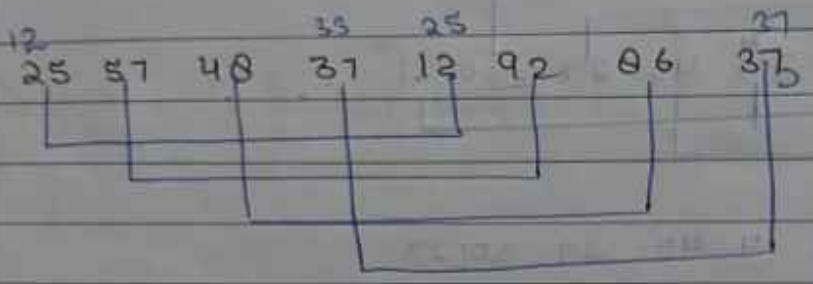
(2) 25 57 48 37 12 92 86 33

Pass 1.

$$gap = Floor\left(\frac{N}{2}\right)$$

$$= \frac{8}{2}$$

$$\boxed{gap = 4}$$

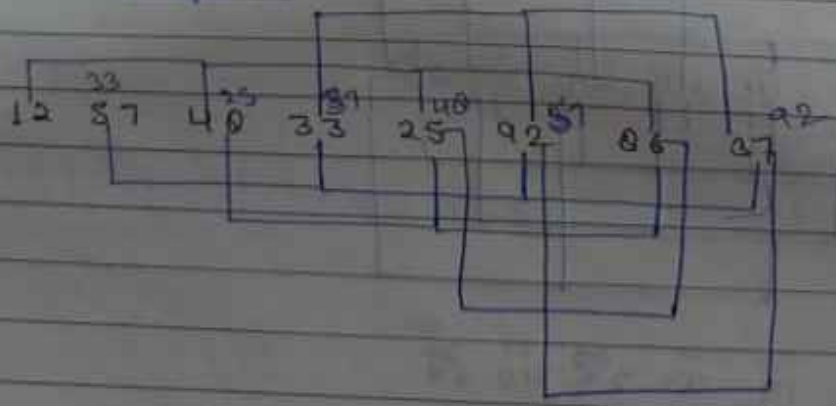


12 57 48 33 25 92 86 37

Pass 2.

$$gap = 4/2$$

$$\boxed{gap = 2}$$



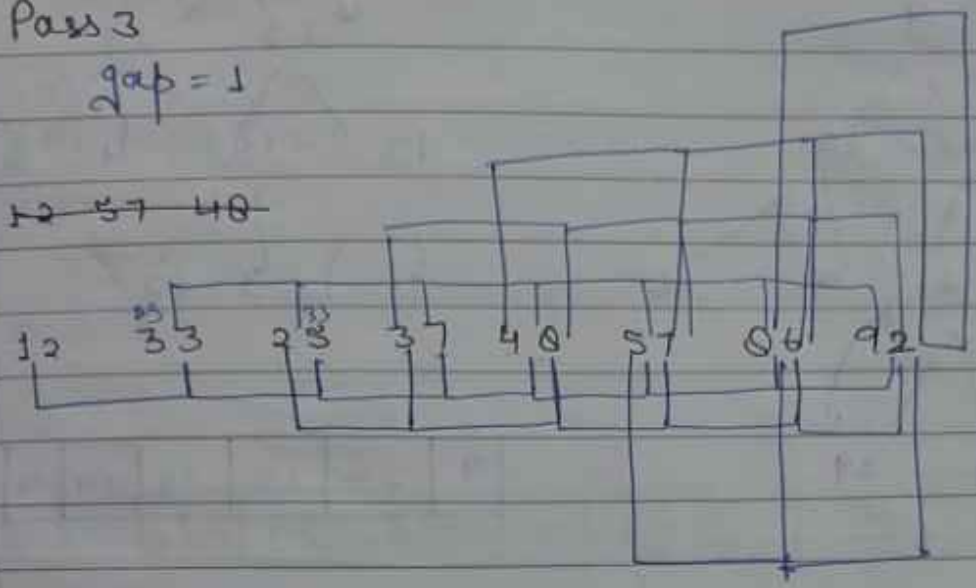
12 33 25 37 48 57 86 92

~~12 57 48~~

Pass 3

gap = 1

~~12 57 40~~

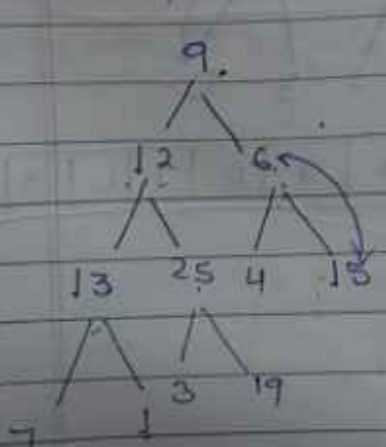


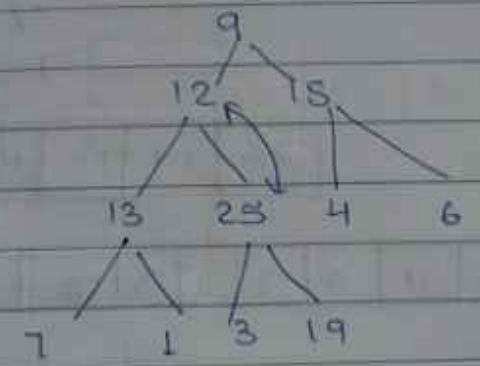
Sorted array

12 25 33 37 48 57 06 92

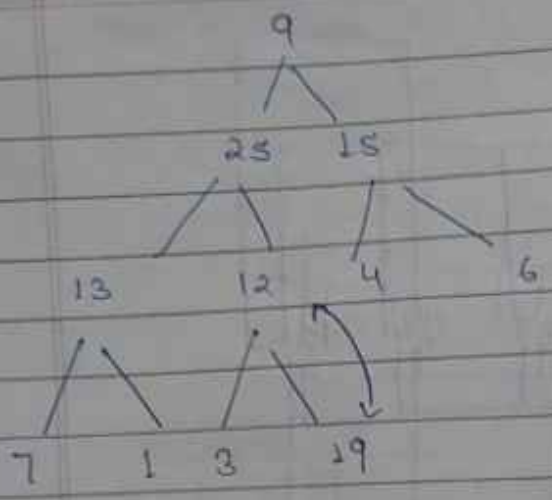
Heap Sort

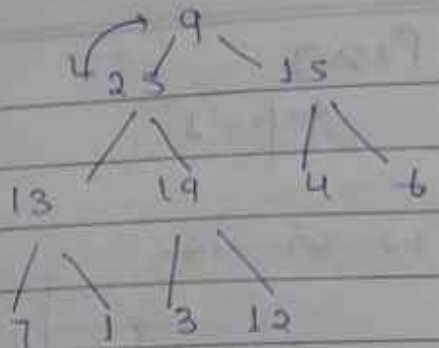
9 12 6 13 25 4 15 7 1 3 19





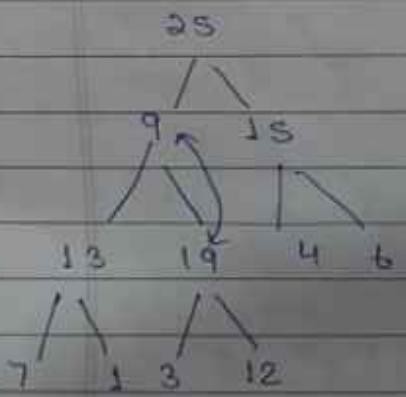
9	12	15	13	25	4	6	7	1	3	19

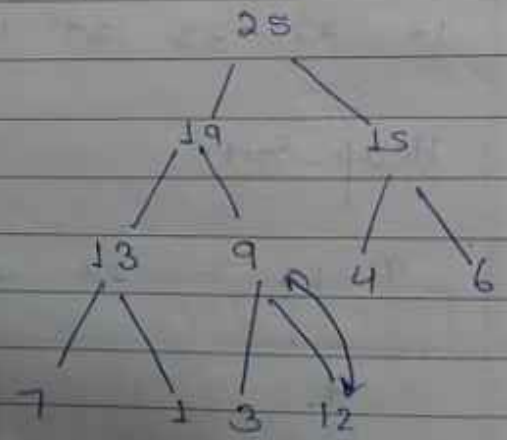




9	25	15	13	19	4	6	7	1	3	12

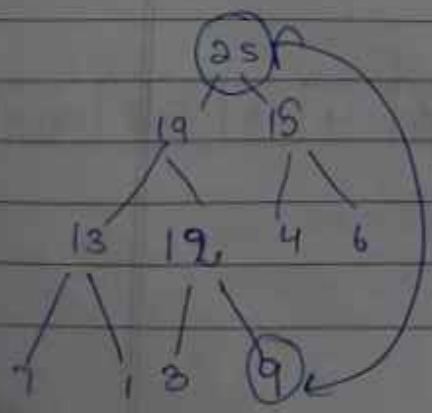
9	25	15	13	12	4	6	7	1	3	19



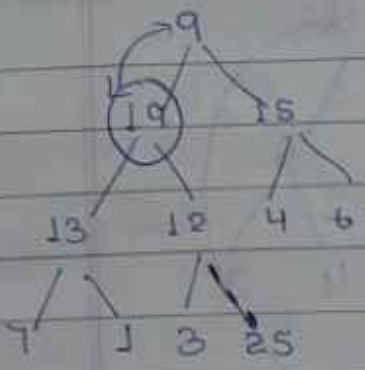


25	19	15	13	9	4	6	7	1	3	12

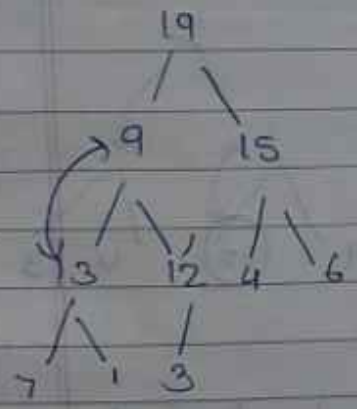
25	9	15	13	19	4	6	7	1	3	12

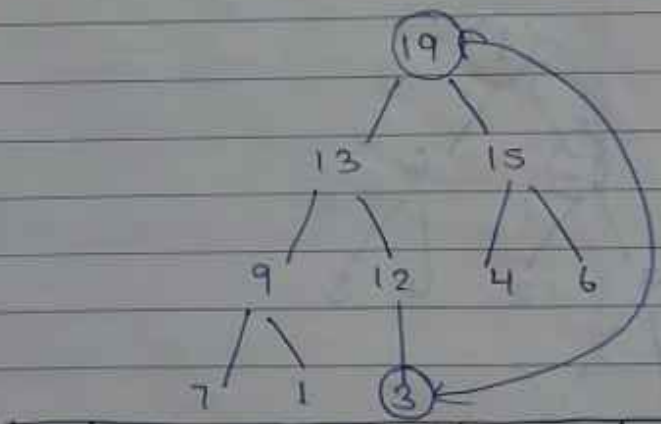


25	19	15	13	12	4	6	7	1	3	9



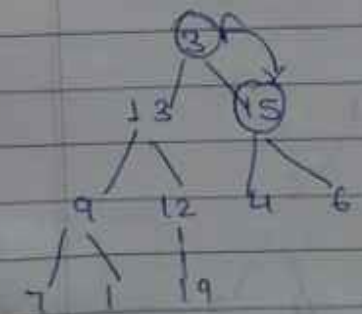
9	19	15	13	12	4	6	7	1	3	25



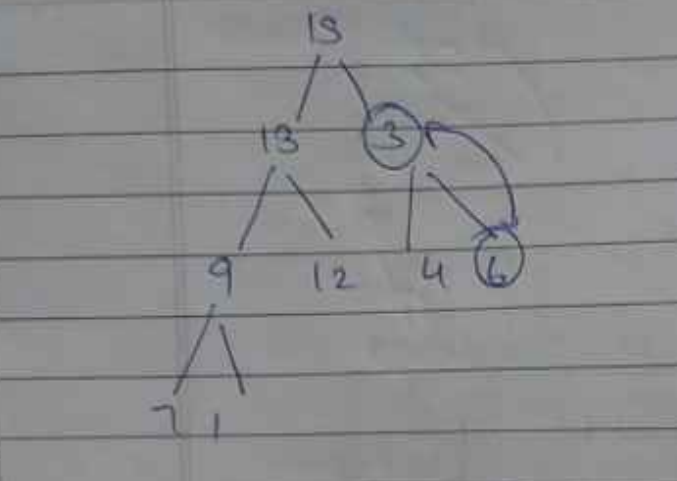


19	13	15	9	12	4	6	7	1	3

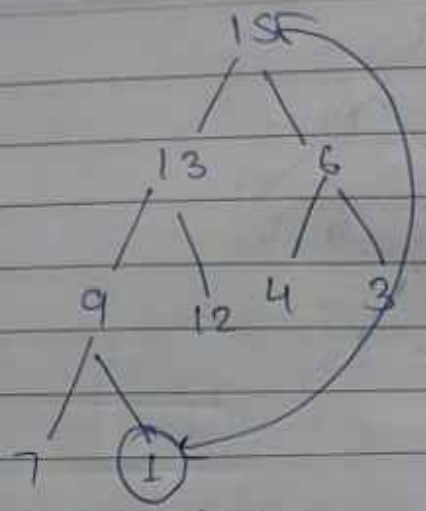
19	9	15	13	12	4	6	7	1	3



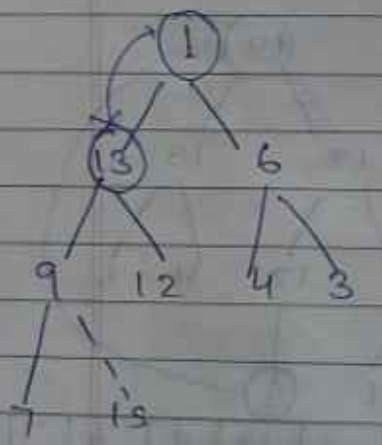
3	13	15	9	12	4	6	7	1	19	25

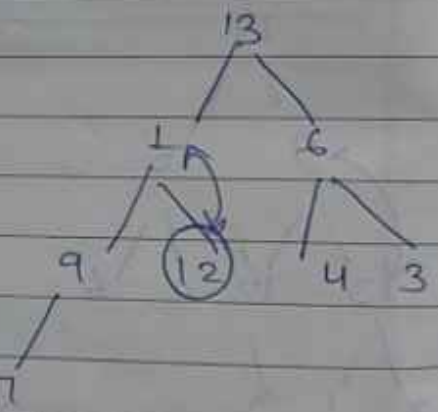


15	13	3	9	12	4	6	7	1



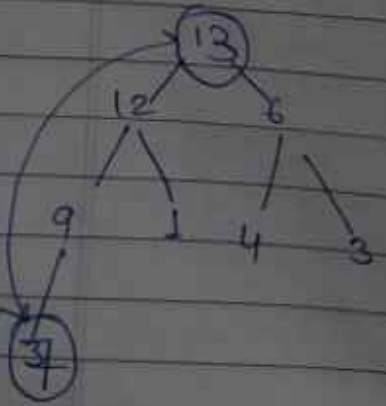
15	13	6	9	12	4	3	7	1

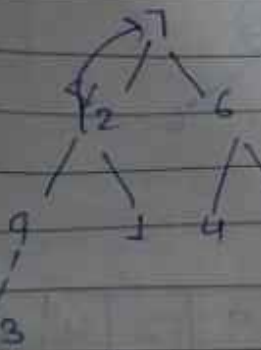




13	1	6	9	12	4	3	7

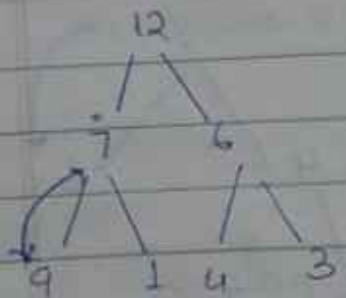
1	13	6	9	12	4	3	7	15	19	25

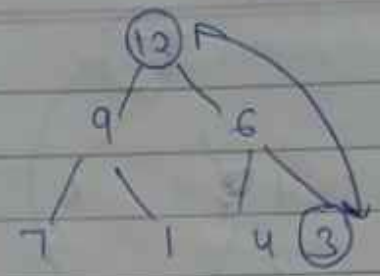




13	12	6	9	1	4	3	7

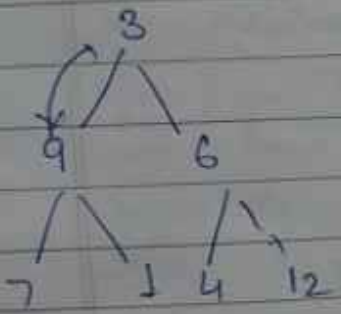
7	12	6	9	1	4	3	13	15	19	25

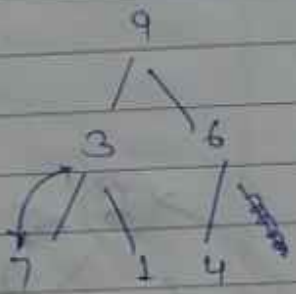




12	7	6	9	1	4	3

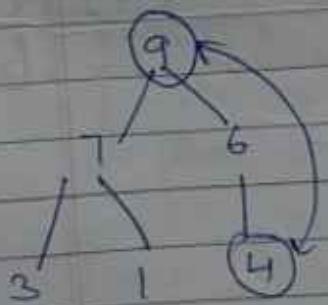
12	9	6	7	1	4	3

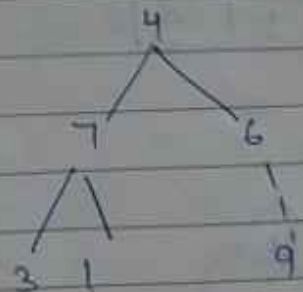




9	3	6	7	1	4

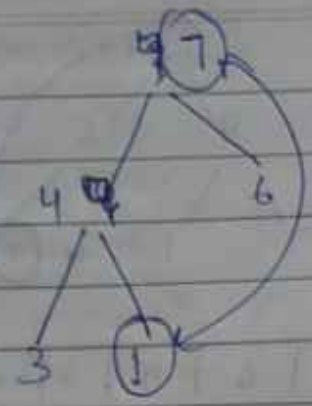
3	9	6	7	1	4	12	13	15	19	25

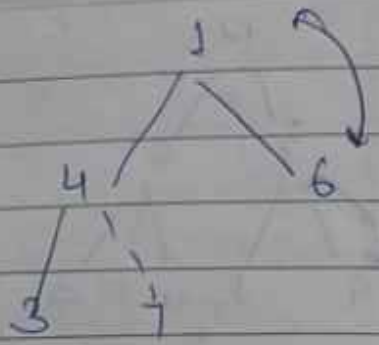




9	7	6	3	1	4

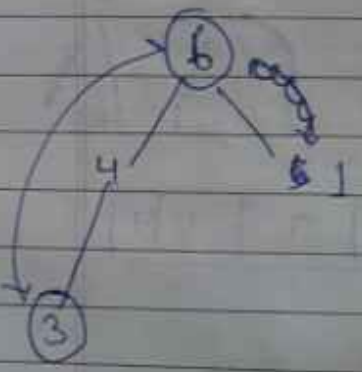
4	7	6	3	1	9	12	13	15	19	25

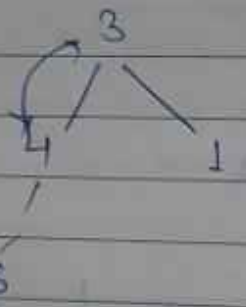




7	4	6	3	1

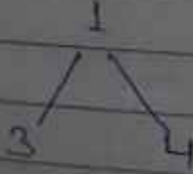
1	4	6	3	7	9	12	13	15	19	25





6	4	1	3	1

3	4	1	6	7	9	12	13	15	19	25



4	3	1

Sorted array

1	3	4	6	7	9	12	13	15	19	25

~~KRUSKAL's Algorithm~~

SPANNING TREE

A spanning tree of a graph is just a subgraph that contain all the vertex and edge is a tree. a graph may have many spanning tree.

A minimum spanning tree of minimum weight spanning tree is a spanning tree which have weight less than or equal to the weight of every other spanning tree.

Kruskal's Algorithm

Kruskal algorithm is an algorithm that find a minimum spanning tree for a connected weighted graph.

It finds a same edge to add to the growing forest by finding of all the edges that connected any two tree in the forest and a edge (u, v) of least weight.

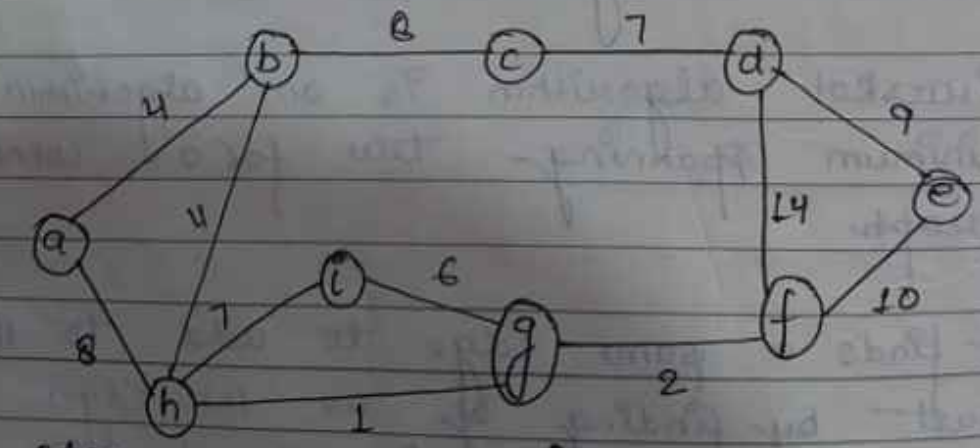
If graph is connected that it find a minimum spanning tree for each components

Algorithm

MST (G, W)

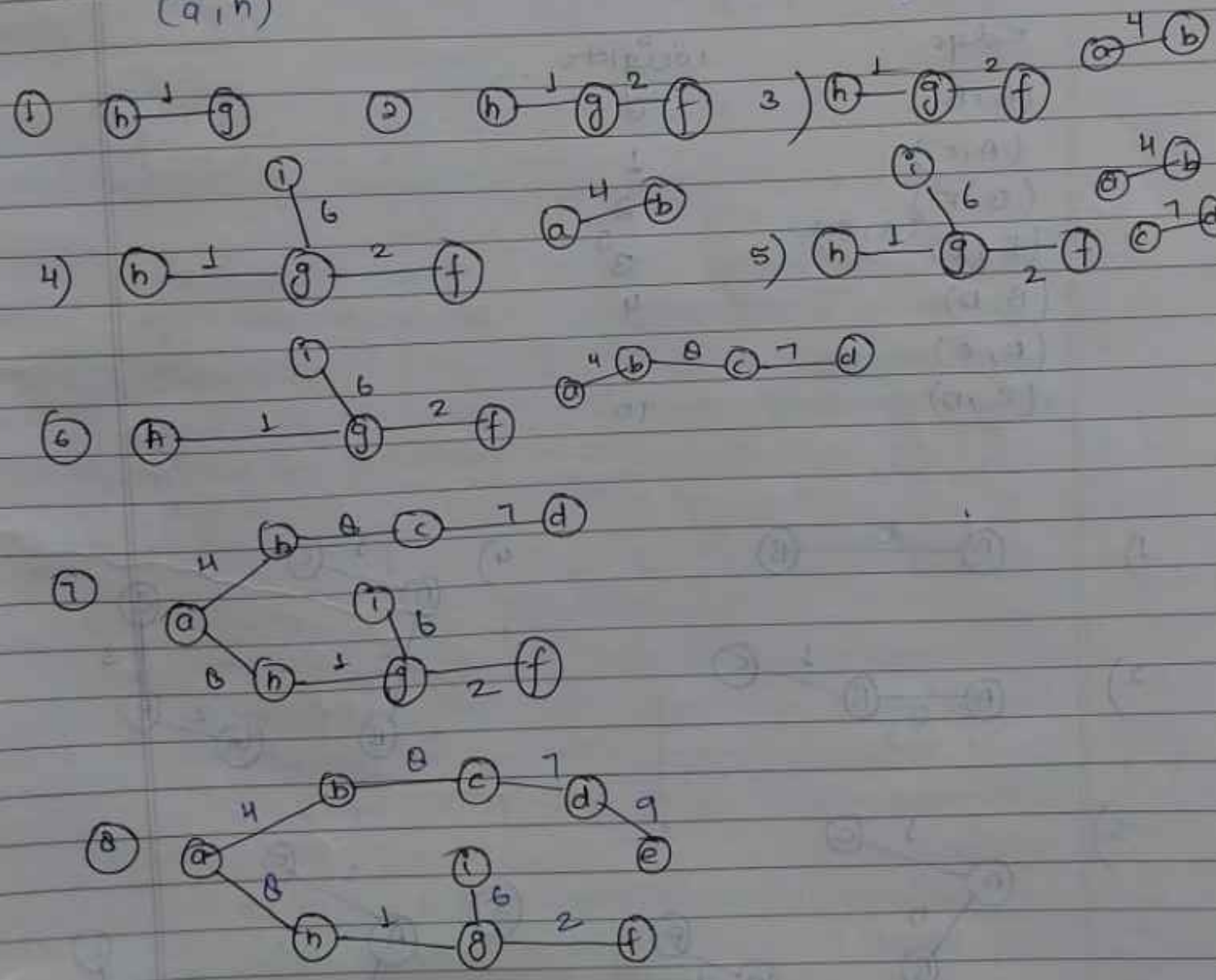
1) A ← ϕ
2) For each vertex V ∈ V(G)
3) do make - SET(V)
4) Sort the edge of E into non-decreasing order by weight W
5) For each edge (u, v) ∈ E taken in non-decreasing order by weight (w)
6) do if Find set(V) ≠ Find - set(V)
7) Then A ← A ∪ {(u, v)}
8) UNION (U, V)
9) Return A

Ans



edge	weight
(h, g)	1
(g, f)	2
(a, b)	4
(i, g)	6
(a, h)	7
(c, d)	7

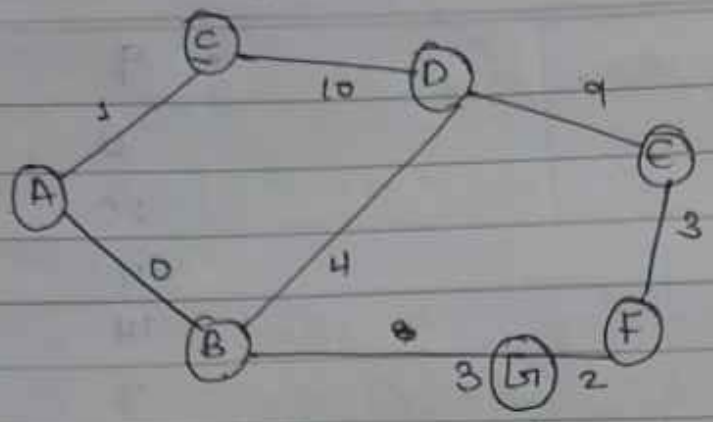
(e, d)	9
(b, c)	8
(e, f)	10
(b, h)	11
(d, f)	14
(h, i)	7
(g, h)	8



Total cost = 8+4+8+7+9+6+2+1

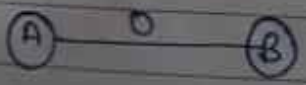
= 45

Q

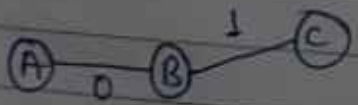


edge	weights
(A, B)	0
(A, C)	1
(G, F)	2
(F, E) (B, G)	3
(B, D)	4
(D, E)	9
(C, D)	10

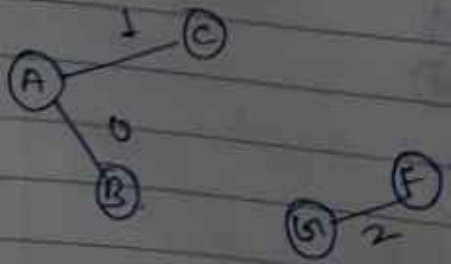
1)



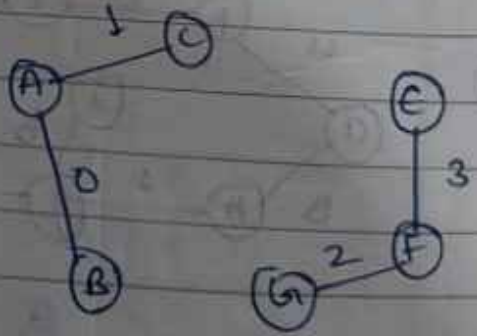
2)



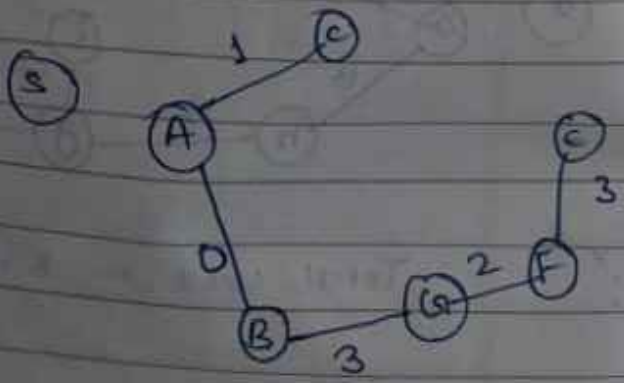
3)

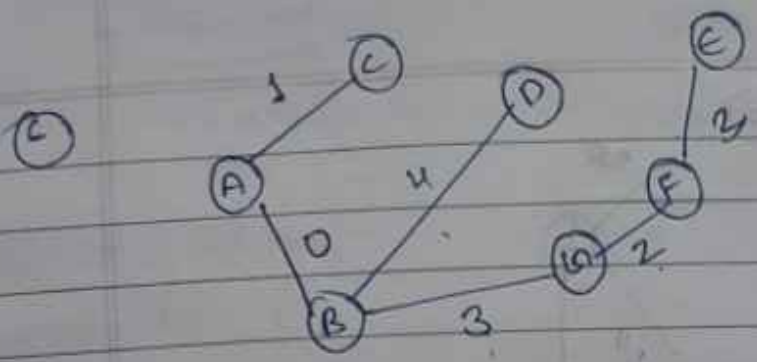


4)



5)





Total cost = 0+1+2+3+3+4 = 13

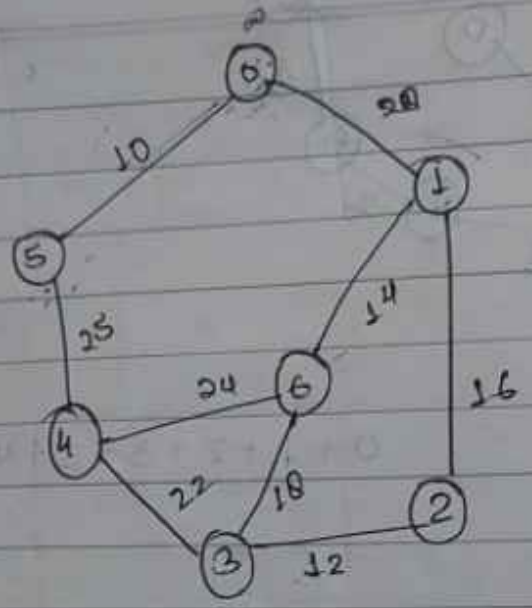
Prim's algorithm

The main idea prims algorithm is similar to the Kruskal algorithm. Finding shortest path in a given graph and also find out minimum spanning tree

Algorithm :

```
For each u ∈ V[G]
do key[u] ← ∞
   π[u] ← NIL
   key[s] ← 0
   Q ← V[G]
while Q ≠ ϕ
do u ← Extract-MIN(ϕ)
   for each v ∈ Adj[v]
do if v ∈ ϕ and w(u,v) < key(v)
      key[v] ← w(u,v)
```

Q



Step 1.

u, v

$$\begin{bmatrix} (0,5) \rightarrow 10 \\ (0,1) \rightarrow 28 \end{bmatrix} - 10 \quad (5,0)$$

Step 2

$$\begin{bmatrix} (5,0) \rightarrow 10 \\ (5,4) \rightarrow 25 \end{bmatrix} 25$$

3

$$\begin{bmatrix} (4,5) \rightarrow 25 \\ (4,6) \rightarrow 24 \\ (4,3) \rightarrow 22 \end{bmatrix} - 22$$

4

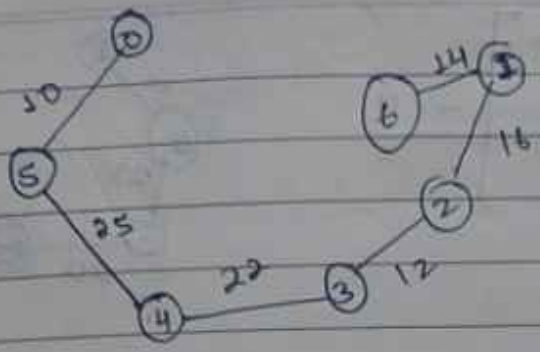
$$\begin{bmatrix} (3,6) \rightarrow 18 \\ (3,4) \rightarrow 22 \\ (3,2) \rightarrow 12 \end{bmatrix} - 12$$

5

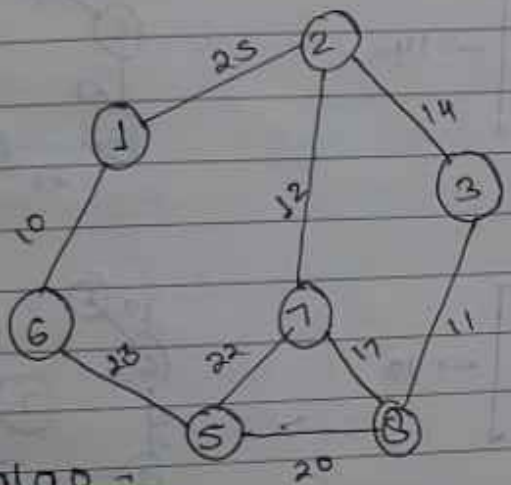
$$\begin{bmatrix} (2,1) \rightarrow 16 \\ (2,3) \rightarrow 12 \end{bmatrix} - 16$$

6

$$\begin{bmatrix} (1,0) \rightarrow 28 \\ (1,6) \rightarrow 14 \\ (1,2) \rightarrow 16 \end{bmatrix} - 14$$



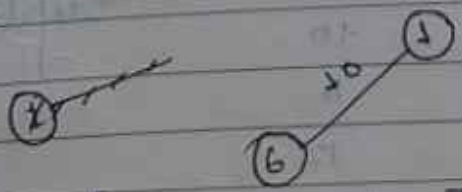
Ques.



Prims algo -

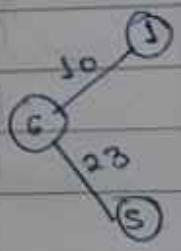
Sol Step 1.

$$\begin{bmatrix} (1,2) \longrightarrow 25 \\ (1,6) \longrightarrow 10 \end{bmatrix} \rightarrow 10$$



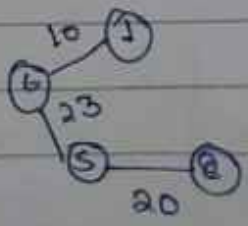
Step 2

$$\begin{bmatrix} (1,6) \rightarrow 10 \\ (6,5) \rightarrow 23 \end{bmatrix} \rightarrow 23$$



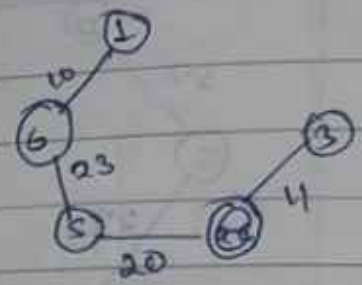
Step 3

$$\begin{bmatrix} (6,5) \rightarrow 23 \\ (5,7) \rightarrow 22 \\ (5,8) \rightarrow 20 \end{bmatrix} \rightarrow 20$$



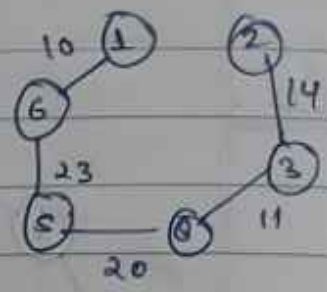
Step 4

$$\begin{bmatrix} (5,8) \rightarrow 20 \\ (7,8) \rightarrow 17 \\ (8,3) \rightarrow 11 \end{bmatrix} \rightarrow 11$$



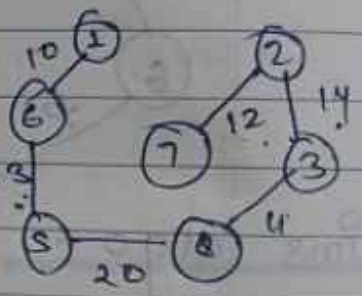
Step 5

$$\begin{bmatrix} (8,3) \rightarrow 11 \\ (2,3) \rightarrow 14 \end{bmatrix} \rightarrow 14$$



Step 6

$$\begin{bmatrix} (1,2) \rightarrow 25 \\ (2,7) \rightarrow 12 \\ (2,3) \rightarrow 14 \end{bmatrix} \rightarrow 12$$

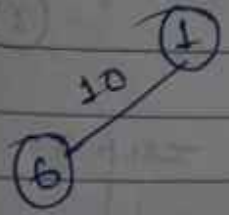


Total cost = 10+12+14+11+23+20 = 90

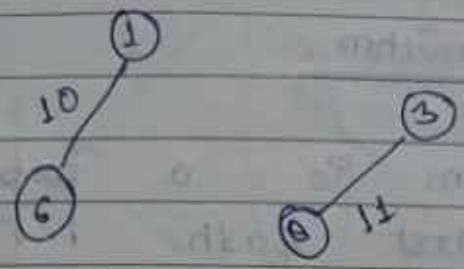
Kruskal Algorithm

edge	weights
(1,6)	10
(8,3)	4
(2,7)	12
(2,3)	14
(7,8)	17
(5,8)	20
(7,5)	22
(6,5)	23
(1,2)	25

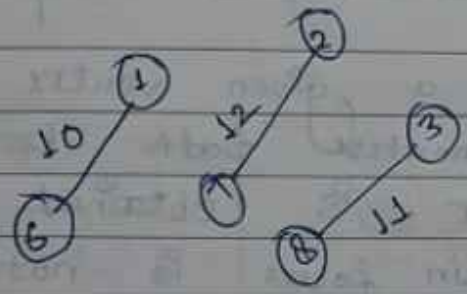
Step 1



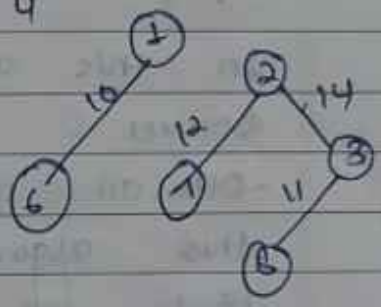
Step 2



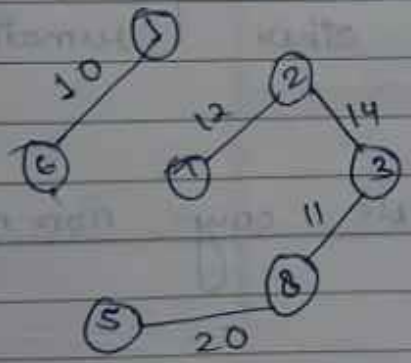
Step 3

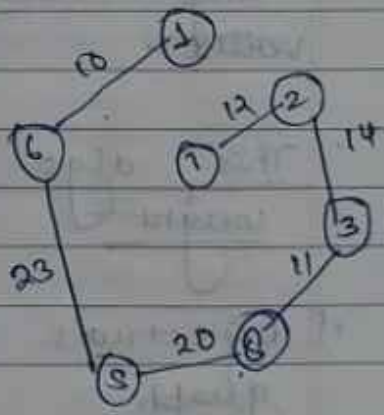


Step 4



Step 5





Total cost = 10+12+14+11+23+20 = 90

Dijkstra's Algorithm

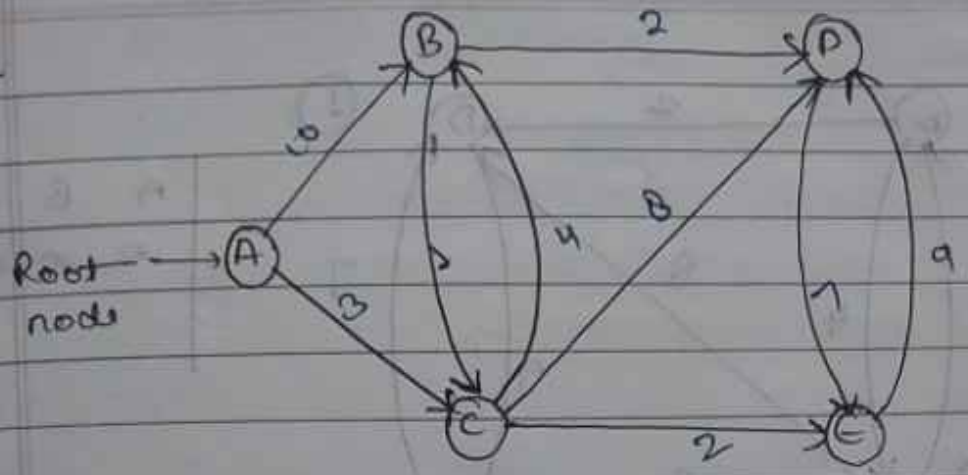
Dijkstra's algorithm is a popular algorithm for finding shortest path. This algo called single source shortest path algo.

In this algo. for a given vertex called source. The shortest path to all other vertex is obtained. In this algo. the main focus is not to find only one single path but to find the shortest path from any vertex to all other remaining vertex.

This algo. is applicable only non-negative weight

1) Dijkstra's algo. find shortest path to graph vertex in order of their distance from given source.
2) First it find the shortest path from the source to a vertex nearest to it.
3) The second nearest and so on.

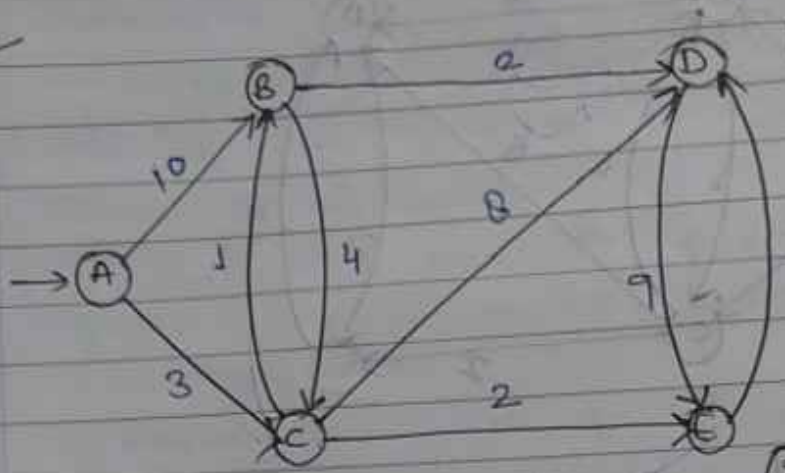
Ex



Visited node	A	B	C	D	E
	0	∞	∞	∞	∞
A	0	10	3	∞	∞
C	0	7	3	11	5
E	0	7	3	11	5
B	0	7	3	9	5

A → C → E → B → D

Ex



(1)

	A	B	C	D	E
	∞	∞	∞	∞	∞

(2)

	A	B	C	D	E
A	0	∞	∞	∞	∞

(3)

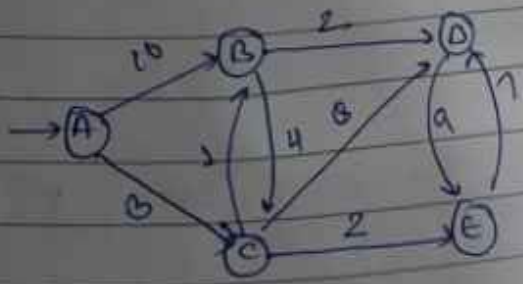
	A	B	C	D	E
A	0	10	3	∞	∞

(4)

	A	B	C	D	E
A	0	∞	∞	∞	∞
	0	10	3		
	0	4	3	11	5

(5)

	A	B	C	D	E
	∞	∞	∞	∞	∞
A	0	∞	∞	∞	∞
C	0	10	3	∞	∞
B	0	4	3	11	5
D	0	4	3	6	5



Algorithm

1) Initialize single source (G, S)
2) $S \leftarrow \phi$
3) $\phi \leftarrow V[G]$
4) while $\phi \neq \phi$
5) do $u \leftarrow$ Extract - Min(ϕ)
6) $S \leftarrow S \cup \{u\}$
7) For each vertex $\in$ Adj $\{u\}$
8) do relax (u, v, w)

Bellman Ford Algorithm.

The graph G(V, E) contain a negative weight cycle, then some shortest path may not exist. Bellman ford algorithm find all shortest path length from source to all edge or determine that a negative weight cycle single source shortest path problem in the more general case in which edge width can be negative.

Algorithm.

Bellman Ford (G, W, S)

1) Initialize single source (G, S)
2) For $i \leftarrow 1$ to $[V(G) - 1]$

3 do for each edge $(u,v) \in E[G]$

4. do relax (u, v, w)

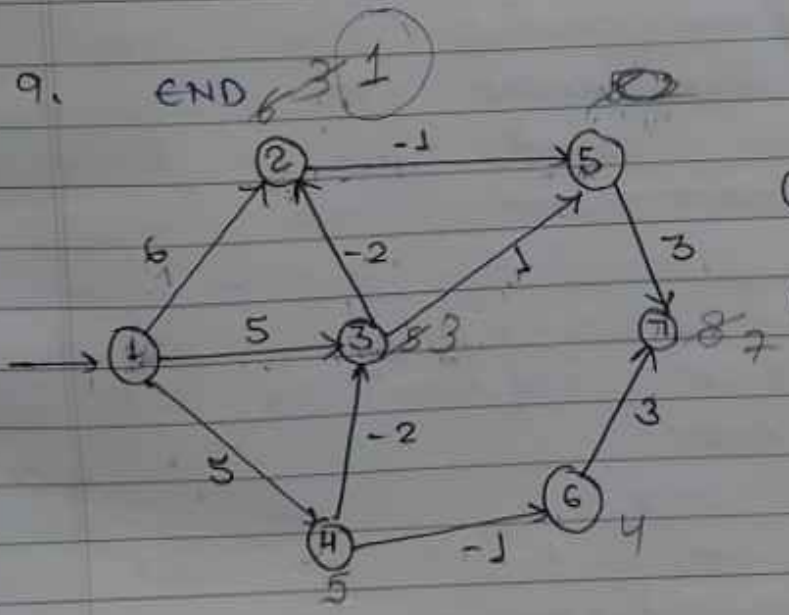
5. For each edge $(u,v) \in E[G]$

6. do if $d[v] > d[u] + w(u,v)$

7. then return false.

8. Return TRUE

9. END



① No negative cycle exist.

② Negative weight hold

③ Source node acc. to you.

Step 1 edge of list → (1,2), (1,3), (1,4), (2,5), (3,2) (3,5) (4,3) (4,6), (5,7) (6,7)

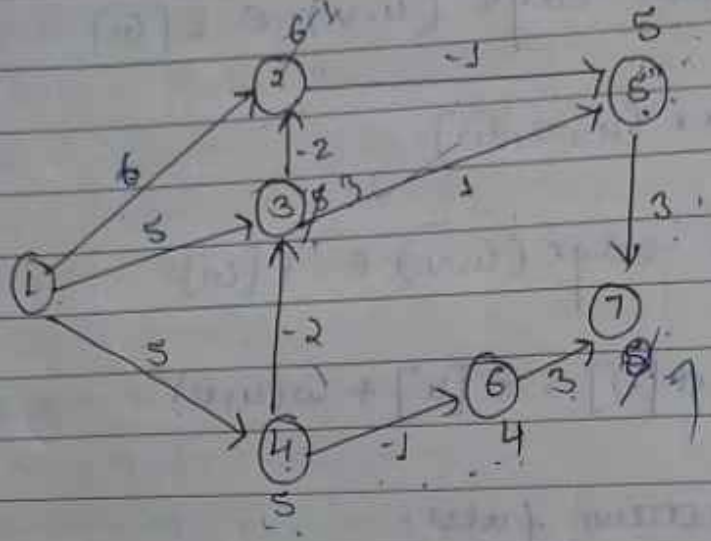
N = 7 (no. of vertex)

n = N-1

= 7-1

n = 6 Relaxation

Step 1 :



Greedy methods.

The greedy method is straight forward method. This method is popular for obtaining the optimize soln. In Greedy technique the solution is constructed through a sequence of step, Each step expanding a particular constructed soln obtained

"Greedy algorithm always make the choice that look best at that moment."

1. First we select some soln from input domain.

2. Then we check whether the soln is feasible ~~physical~~ or not

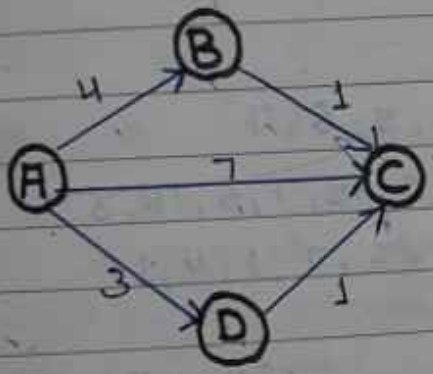
3. From the set of feasible soln for particular soln that satisfy the object of the fun^n. Such a soln is called optimal soln.

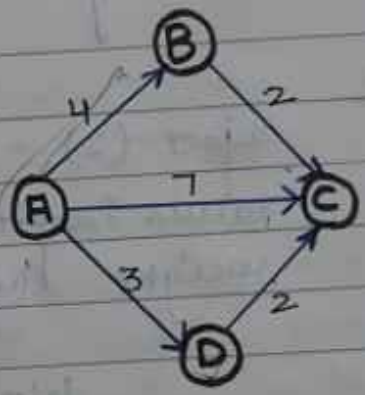
* Greedy algorithm property

Greedy choice property - It make local optimal choice in the hope that

choice lead to global optimal soln.

Optimal substructure -
Optimal soln contain -





A → D ⇒ 3
A → B → 4
] 3 local optimal

A → B → C → 6
A → D → C → 5
] global optimal
]-5

* Application of greedy method.

a) Activity selection problem.
b) Knapsack problem.
c) Minimum spanning tree.
d) Single source shortest path problem

Knapsack problem

If there are n item with the weight $w_1, w_2 \cdots w_n$ and values (profit) $v_1, v_2 \cdots v_n$ and capacity Knapsack to be W then find the most valuable subset of item that fit in the Knapsack.

Ex.

object (I) = 1, 2, 3, 9, 5, 6, 7

values Profit (P) = 10, 5, 15, 7, 6, 18, 3

weight (w) = 2, 3, 5, 7, 1, 4, 1

W = 15 kg

$x\{x_1, x_2, x_3 \cdots x_n\}$

$1 \leq x \leq 0$

There are two type of Knapsack problem

1. **Fractional Knapsack problem**

Here the thief can take fractional of item, meaning that the item can be broken into small pieces so that thief may decide to carry only a fraction of x_i of item I

Fractional knapsack problem can be solve by greedy choice property.

2. 0/1 Knapsack problem (container)

→ Here the item not be broken into smaller pieces, so they may decide either to take an item or to leave it. But may not take a fraction of an item.

0/1 Knapsack problem do not solve by greedy algorithm it only follow dynamic programming algorithm.

* Algorithm.

Fractional Knapsack

1. For i ← 1 to n
2. do x(i) ← 0
3. weight ← 0
4. while weight < w (capacity)
5. do i ← best remaining item.
6. if weight + w[i] ≤ w
7. then x[i] = 1
8. weight ← weight + w[i]
9. else
10. x[i] ← (w - weight) / w[i]
11. weight ← w
12. Return x

Ques Consider the following instance of knapsack problem.

$W = 50$

$(V_1, V_2, V_3) = (60, 100, 120)$

weight $= (10, 20, 30)$

(w_1, w_2, w_3)

$n = 3$

Sol For $i \leftarrow$ item.

For $i \leftarrow 1$ to 3

do $x[i] = 0$] initially
weight $= 0$

while weight ≤ 50

$0 < 50$

do $i \leftarrow$ best remaining item

if weight $+ w[i] \leq 50$

$0 + 10 \leq 50$

$10 \leq 50$

then $x[i] = 1$

$x[i] = 1$ (weight 10 taken)

weight $\leftarrow$ weight $+ w[i]$

$0 + 10 \leftarrow 10$

Therefore weight is equal to 10 (leaving 40 pounds)

while weight < 50

while $10 < 50$

do $i \leftarrow$ best remaining item (40)

$i \leftarrow 2$

if weight $+ w[i] \leq 50$

Ques

$10 + 20 \leq 50$

$30 \leq 50$

then $x[i] = 1$

$x[2] = 1$ (weight 20 taken)

weight ← weight + w[i]

← 10+20 ← 30

therefore weight = 30 (leaving 10 pounds)

$x[1] = 0$ (not taken)

$x[2] = 1$ (taken)

$x[3] = 1$ (taken)

$[0\ 1\ 1]$

Ques Five Item belong their respective weight and value.

$I = [I_1, I_2, I_3, I_4, I_5]$

$W = [5, 10, 20, 30, 40]$

$V = [30, 20, 100, 90, 160]$

Capacity $W = 60$

ing 40 pounds)

Item	Wi	Vi
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Item	w_i	v_i	$P_i = v_i/w_i$
I_1	5	30	6
I_2	10	20	2
I_3	20	100	5
I_4	30	90	3
I_5	40	160	4

Arrange P_i in descending order.

Item	w_i	v_i	P_i
I_1	5	30	6
I_3	20	100	5
I_5	40	160	4
I_4	30	90	3
I_2	10	20	2

35
20
5

60kg

$(60 - 25 = 35)$

$\frac{160}{40} \times 35 = 140$

Thus optimal value

$= 30 + 100 + 140$

$= 270$

Activity Selection problem

Activity selection problem is given a resource such as CPU or lecture hall and small n set of activity such as task or lecture that

want to utilize the resource. The activity have starting and finishing time which can be denoted by s_i and f_i Then activity selection is to find maximum size subset 'A' of mutual compatible activity.

Activity share a resource which can be used by only one activity at a time

$$s_i \leq f_i$$

The algorithm of activity selection problem

1. n ← length
2. A ← {1}
3. j ← 1
4. for i ← 2 to n
5. do if $s_i \leq F_i$
6. A ← A ∪ {i}
7. j ← i
8. Return

Eg. Ten activity along with there start and finishing time

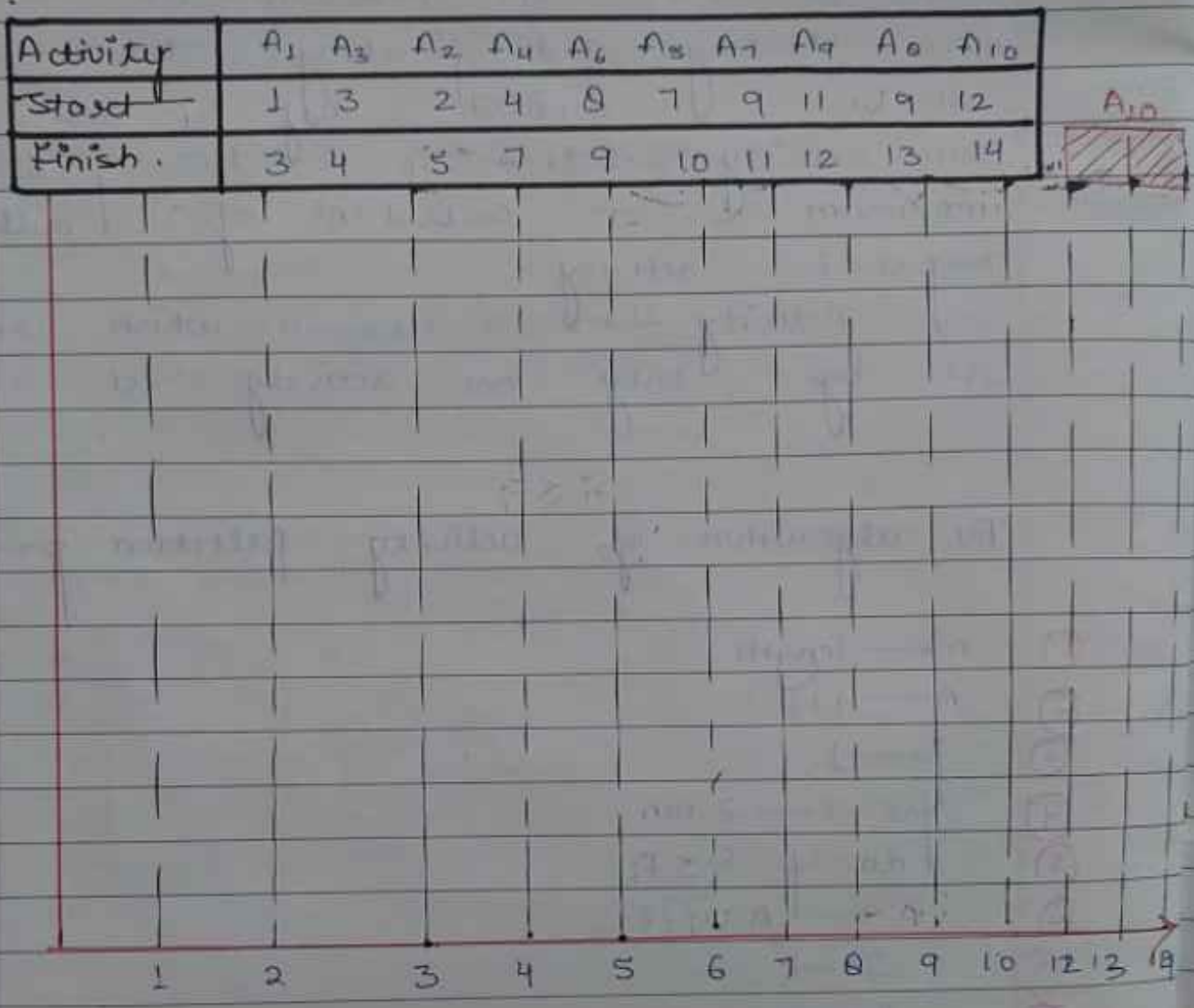
$S = \langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10} \rangle$

$s_i = \langle 1, 2, 3, 4, 7, 8, 9, 9, 11, 12 \rangle$

$f_i = \langle 3, 5, 4, 7, 10, 9, 11, 13, 12, 14 \rangle$

Step 1.

Activity	A_1	A_3	A_2	A_4	A_6	A_8	A_7	A_9	A_5	A_{10}
Stard	1	3	2	4	8	7	9	11	9	12
Finish.	3	4	5	7	9	10	11	12	13	14



matrix chain multiplication

In this there are sequence $(A_1, A_2 - - An)$ of n matrix to be multiplied and we wish to compute the product $(P_1, P_2, P_3 - - P_n)$. In order to evaluate the expression we had to bracket. Matrix multiplication is associated and so {fill the same product}

A

Longest common Subsequence (LCS)

A subsequence of a given sequence is just the sequence with some elements left out. Given two sequence X and Y we say that a sequence Z is a common sequence of X and Y. If Z is a subsequence of both X and Y. In the longest common subsequence problem we are given two sequence X and Y and with to find maximum length common subsequence of X and Y.

LCS problem can be solved using dynamic programming.

LCS Algorithm

```
1.  LCS length (x, y)
2.  m ← length (x)
3.  n ← length (y)
4.  For i ← 1 to m
5.  do c[i, 0] ← 0
6.  for j ← 1 to n
7.  do c[0, j] ← 0
8.  For i ← 1 to m
9.  do for j ← 1 to n
10. do if xi = xj
    then c[i, j] ← [c(i-1, j-1)+1]
```

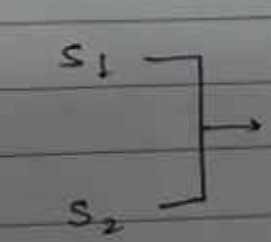
11. $b[i,j] \leftarrow \nwarrow$
12. else if $c[i-1,j] \geq c[i,j-1]$
13. then $c[i,j] \leftarrow c[i-1,j]$
14. $b[i,j] \leftarrow$ "↑"
15. else $c[i,j] \leftarrow c[i,j-1]$
16. $b[i,j] \leftarrow$ '←'
17. return c and b

Formula.

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ (c[i-1,j-1]+1) & \text{if } i,j>0 \text{ and } x_i=x_j \\ \max(c(i,j-1), c(i-1,j)) & \text{if } i,j>0 \text{ and } x_i \neq x_j \end{cases}$$

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ (c[i-1,j-1]+1) & \text{if } i,j>0 \text{ and } x_i=x_j \\ \max(c(i,j-1), c(i-1,j)) & \text{if } i,j>0 \text{ and } x_i \neq x_j \end{cases}$$

Ex.



Diagonal

~~when math~~ max

		L	O	N	G	E	S	T
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
S 1	0	0	0	0	0	0	1	1
T 2	0	0	0	0	0	0	1	2
O 3	0	0	1	1	1	1	1	2
N 4	0	0	1	2	2	2	2	2
E 5	0	0	1	2	2	3	3	3

O N E

Ex

x = A B C B D A B (0 1 2 3 4 5 6 7)

y = B D C A B A (0 1 2 3 4 5 6)

find out LCS ?

Step 1 m×n = 7×6

		A	B	C	B	D	A	B
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
B 1	0	0	1	1	1	1	1	1
D 2	0	0	1	1	1	2	2	2
C 3	0	0	1	2	2	2	2	2
A 4	0	1	1	2	2	2	3	3
B 5	0	1	2	2	3	3	3	4
A 6	0	1	2	2	3	3	4	4

A B C B A

जिधर max होगा उधर जाना बराबर होगा ↑

UNIT-4.

ALL PAIR SHORTEST PATH

The all pair shortest path can be considered mother of all routing problem.

Given a directed graph G(V, E) where V is vertex and E is Edge where edge is non-negative cost

In all pair shortest path algorithm we want minimum route b/w all the pair of vertex of G. It is dynamic programming approach. This algorithm also allow negative edge but no negative cycle.

The basic concept behind all pair shortest path is if for a path (U, V) and its length optimal.

Shortest path for D[U][V] if we can take, for W then —

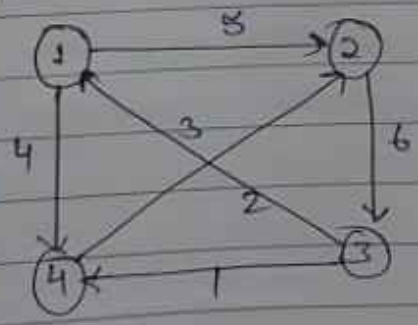
$$D[U][V] = \min[D[U][V], D[U][W] + D[V][W]]$$

Algorithm

Time complexity $= O(n^3)$

1) $n \leftarrow$ row[W]
2) $D^0 \leftarrow W$
3) for $k \leftarrow 1$ to n
4) do for $i \leftarrow 1$ to n
5) do for $j \leftarrow 1$ to n
6) do $d_{ij}^{k} \leftarrow \min\left[d_{ij}^{k-1}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right]$
7) Return $(D)^n$

* Apply Flyod warshal algorithm to find out all pair shortest path in this graph



$$D^0 = \begin{bmatrix} 0 & 5 & 5 & 4 \\ 0 & 0 & 6 & 12 \\ 2 & 7 & 0 & 6 \\ 3 & 8 & 1 & 0 \end{bmatrix}$$

{Branch and bound}

TSP {Travelling Sales Man Problem}

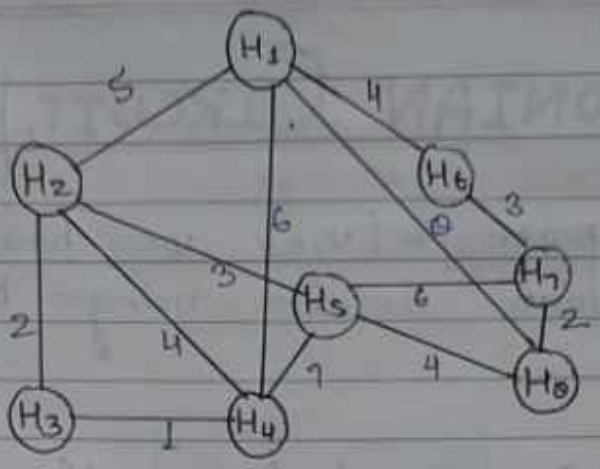
OR

In the optimization version of this problem, we are given a graph G with a cost function defined for each edge in G, and we ward to find the smallest cost that visit every vertex in G and returning back its starting vertex.

We can design an algorithm for TSP by computing for each edge (u, v). The minimum cost path that begin at u and end at v while visiting all the vertex in G along the path. To find such a path we apply TSP for given graph

Ex.

* A newspaper agent daily drop the newspaper to the area assign in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.



Step 1.

	H1	H2	H3	H4	H5	H6	H7	H8
H1	0	5	0	6	0	4	0	7
H2	5	0	2	4	3	0	0	0
H3	0	2	0	1	0	0	0	0
H4	6	4	1	0	7	0	0	0
H5	0	3	0	7	0	0	6	4
H6	4	0	0	0	0	0	3	0
H7	0	0	0	0	6	3	0	2
H8	7	0	0	0	4	0	2	0

Once we visit one vertex then we take another minimum value from other vertex.

Step 2

	H1	H2	H3	H4	H5	H6	H7	H8
→ H1	0	5	0	6	0	(4)	0	7
→ H2	5	0	(2)	4	3	0	0	0
→ H3	0	2	0	(1)	0	0	0	0
→ H4	(6)	4	1	0	7	0	0	0
→ H5	0	(3)	0	7	0	0	6	4
→ H6	4	0	0	0	0	0	(3)	0
→ H7	0	0	0	0	6	3	0	(2)
→ H8	7	0	0	0	(4)	0	2	0

$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5 \xrightarrow{3} H_2 \xrightarrow{2} H_3 \xrightarrow{1} H_4 \xrightarrow{6} H_1$

$= 25$

HAMILTONIAN CIRCUIT. {Back-tracking}

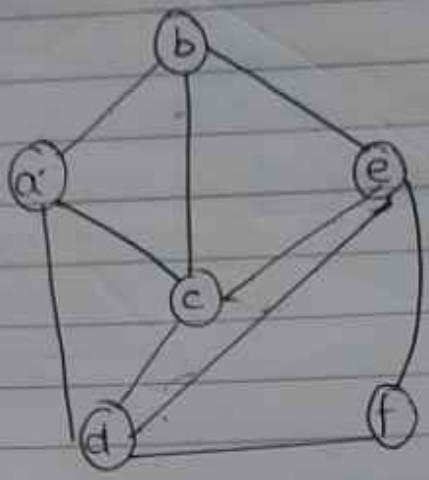
Given a graph G(V,e) we have to find the hamiltonian circuit using back tracking approach

We start our search from any vertex say 'd'. This vertex d becomes the loop of our implicity tree. The first element of our partial soln is the first intermediate vertex of the hamiltonian cycle i.e to be constructed. The next adjacent vertex is selected. On the basis of alphabetical order is selected. If any stage any vertex make cycle with other vertex then we say that dead end is reach.

In this case we back-tracking one step and again search began by selecting another vertex and backtrack the element from the partial soln must be removed.

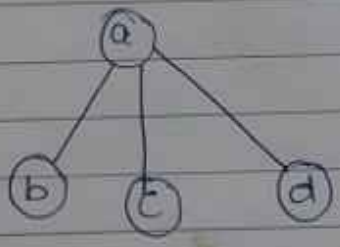
The search using backtracking is successful if a hamiltonian cycle is obtained

Ex

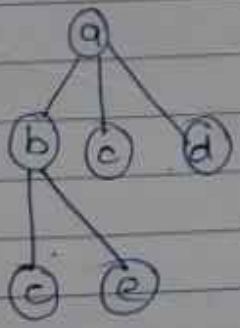


Step 1.

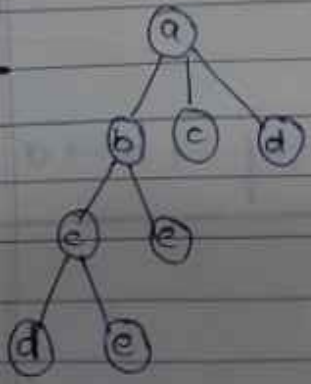
a ← root



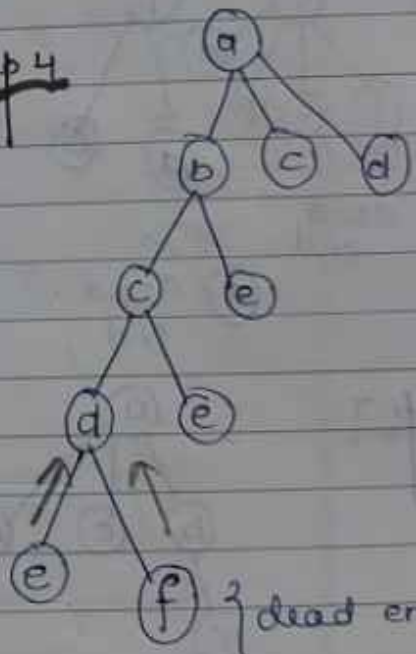
Step 2



Step 3



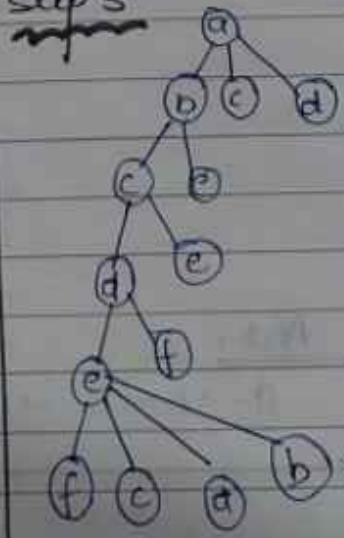
Step 4



} dead end }

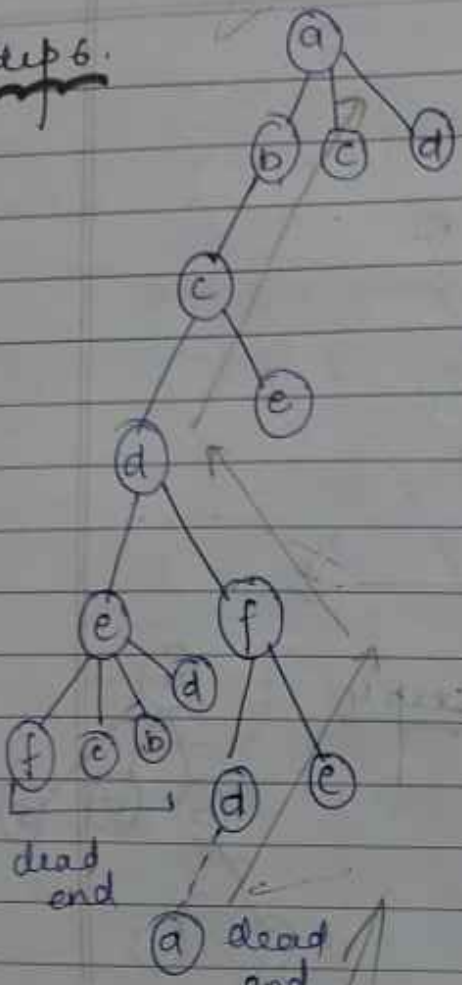
back-tracking

Step 5

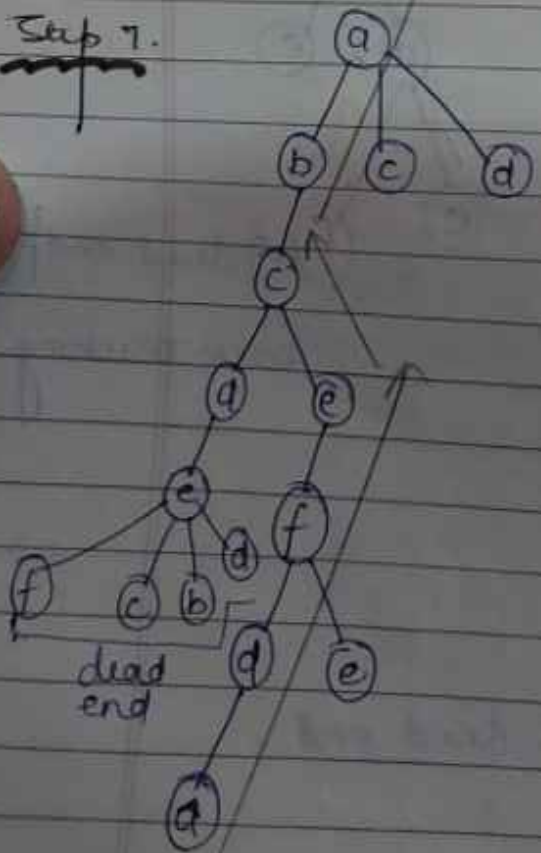


← dead end.

Step 6.

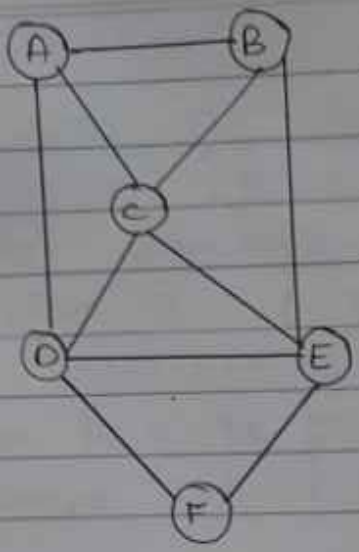


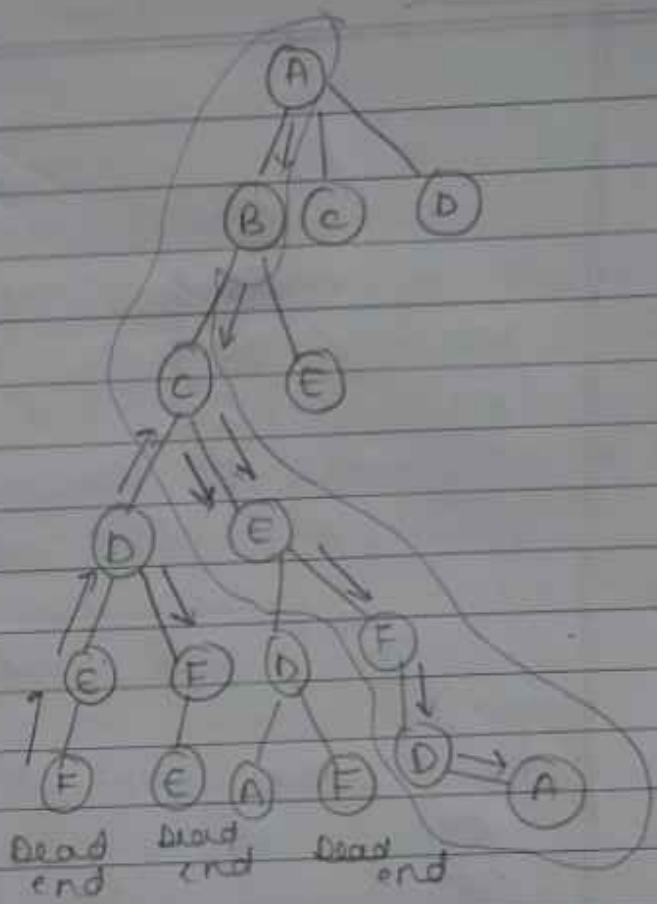
Step 7.



Ans.

a → b → c → e → f → d → a





Sum of Subset

The Sum of subset problem is defined as "Given a set S of n object with width $(w_1 - - - w_n)$ and value M (capacity) we have to find subset of element of S whose total weight equal to M"

The use of notation is -

$S_k = (True, False)$

OR

$S_k = (1, 0)$

Example

Let set S = (1, 2, 3, 4) i.e., here n = 4 and weight vector {10, 25, 5, 10} and M = 25

Solution

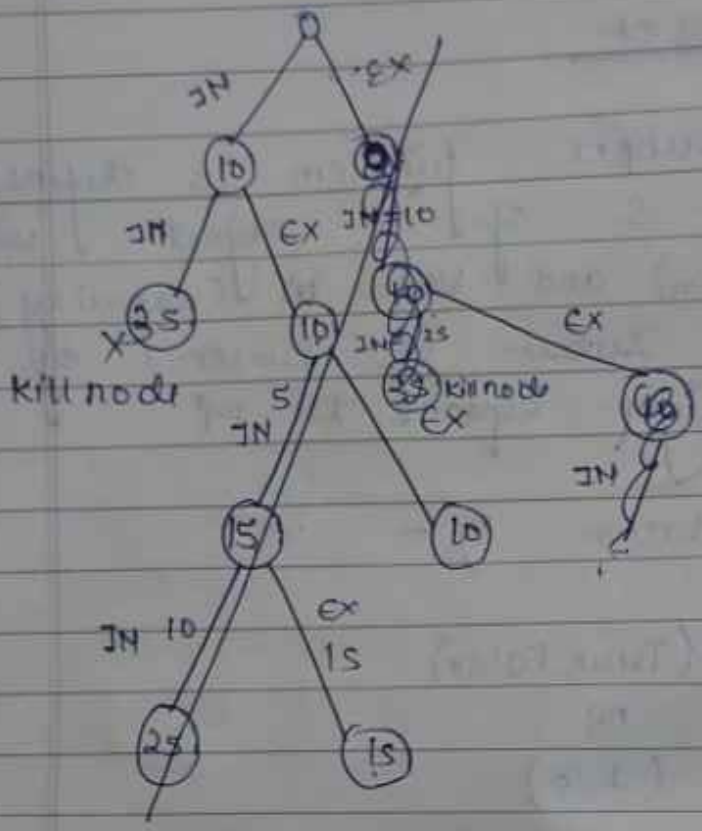
According to our assumption for backtracking algorithm we put 1 if element is included otherwise we put zero. Since the value of M = 25 is our goal is to find {25} by using element of set S.

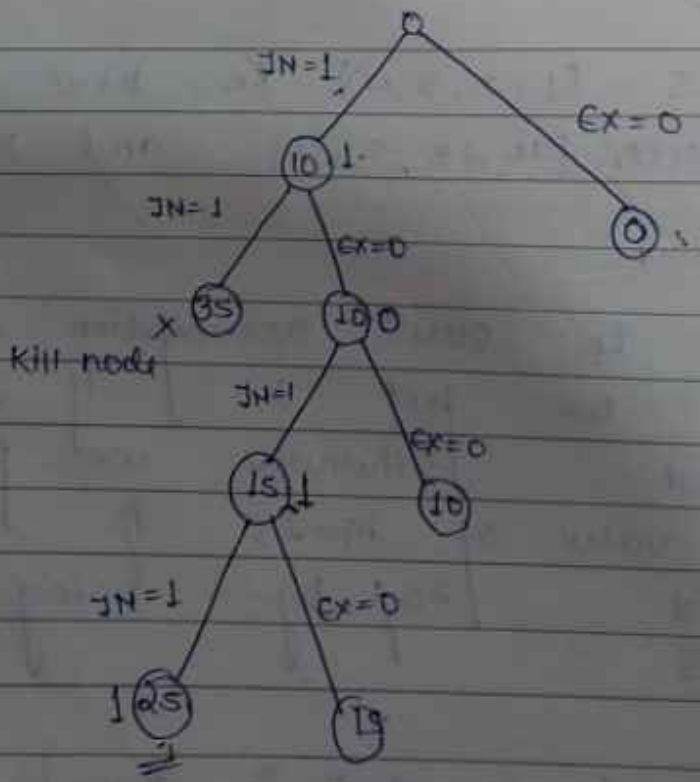
The fix length soln is ~~zero~~ {0, 1, 0, 0} or {1, 0, 1, 1}.

$W = \{10, 25, 5, 10\}$

(1 0 1 1)

(0 1 0 0)





Ques

$S = \{3, 4, 5, 6\}$

and

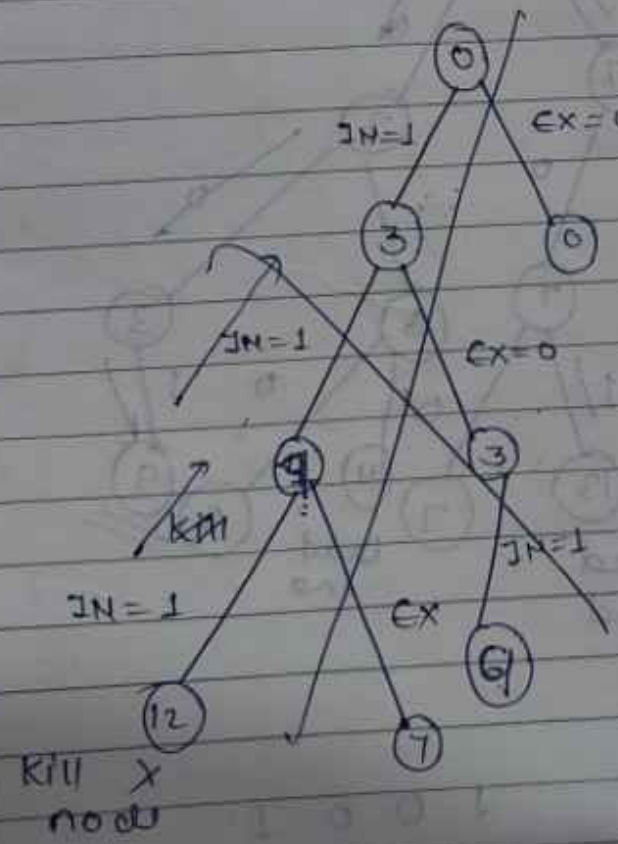
$X = \{9\}$

$\boxed{X = 9}$

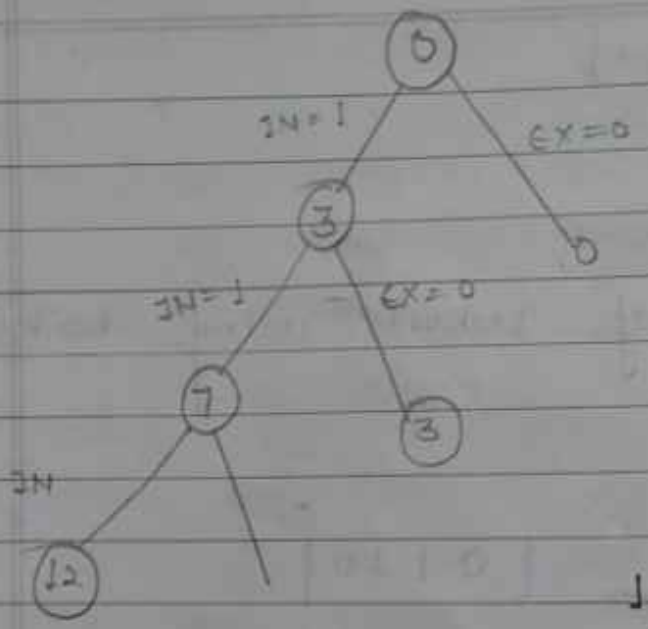
obtain sum of subset using backtracking algo.

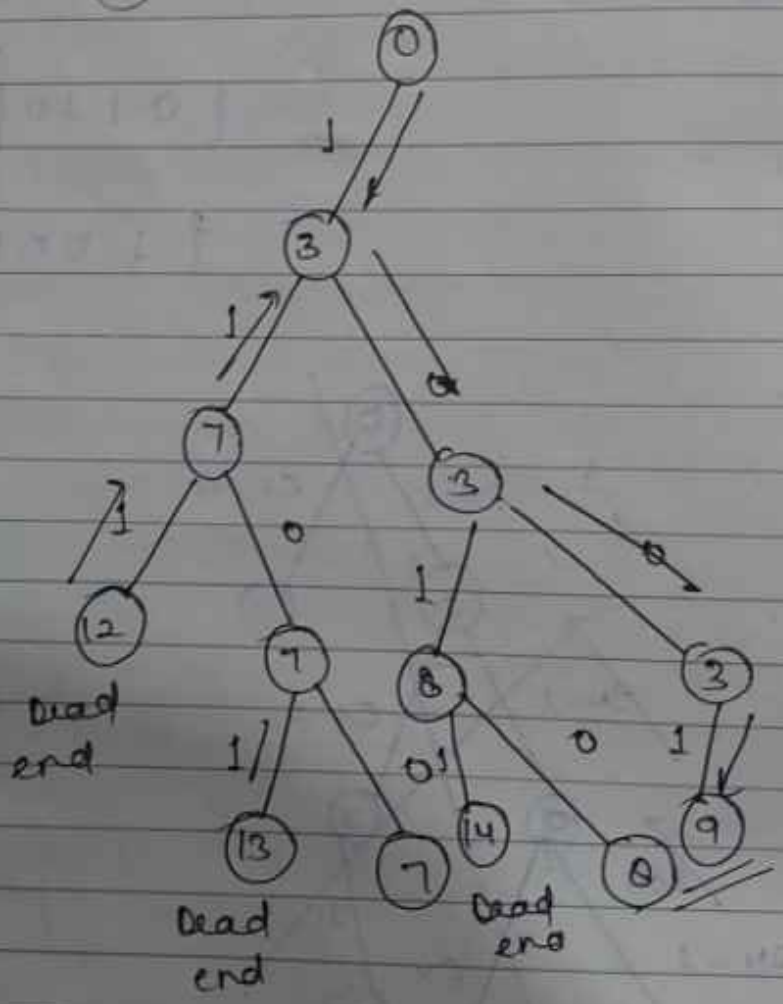
{ 0 1 1 0 }

{ 1 0 0 1 }



(3, 9, 5, 1)



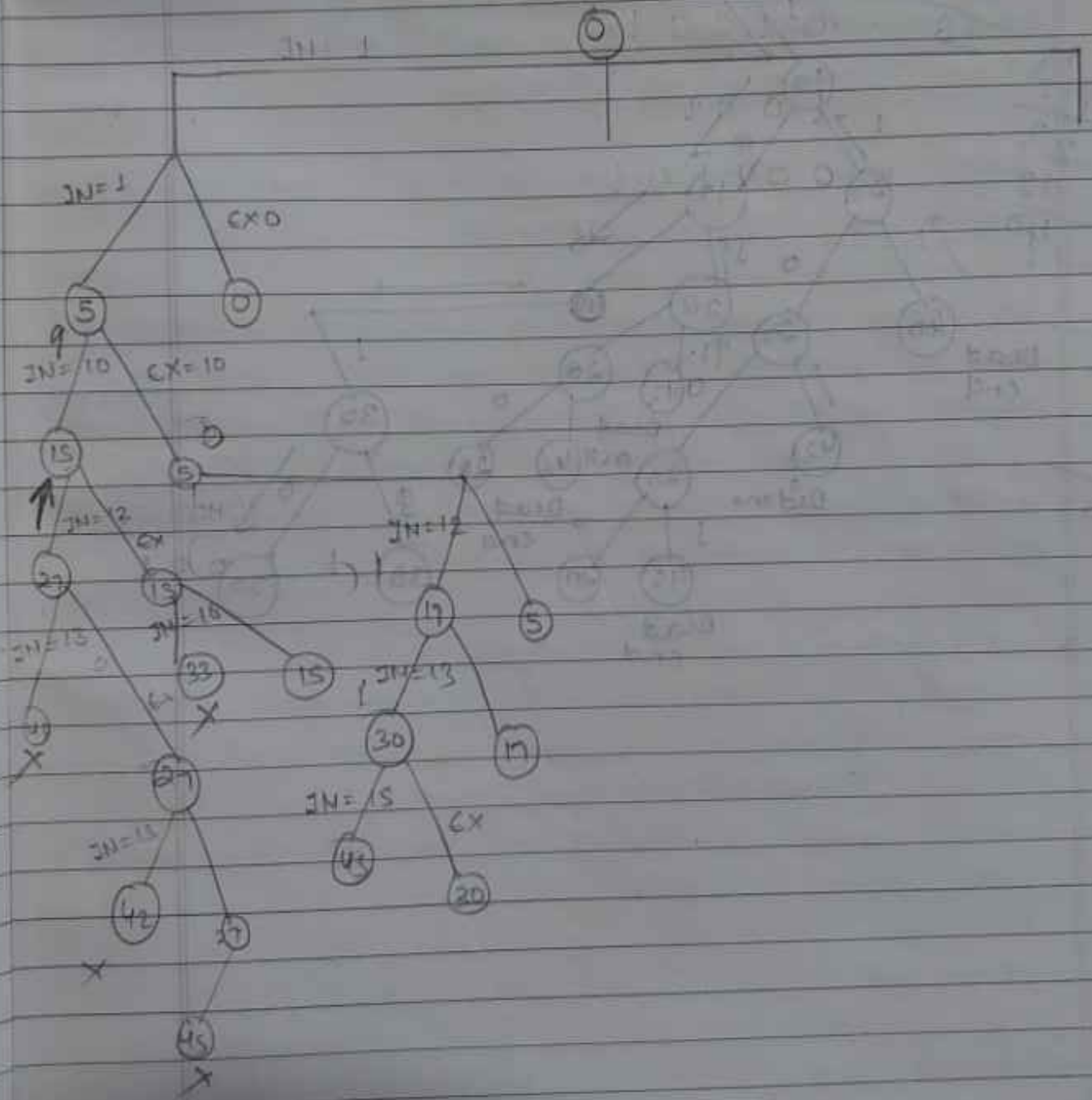


1 0 0 1

Ques

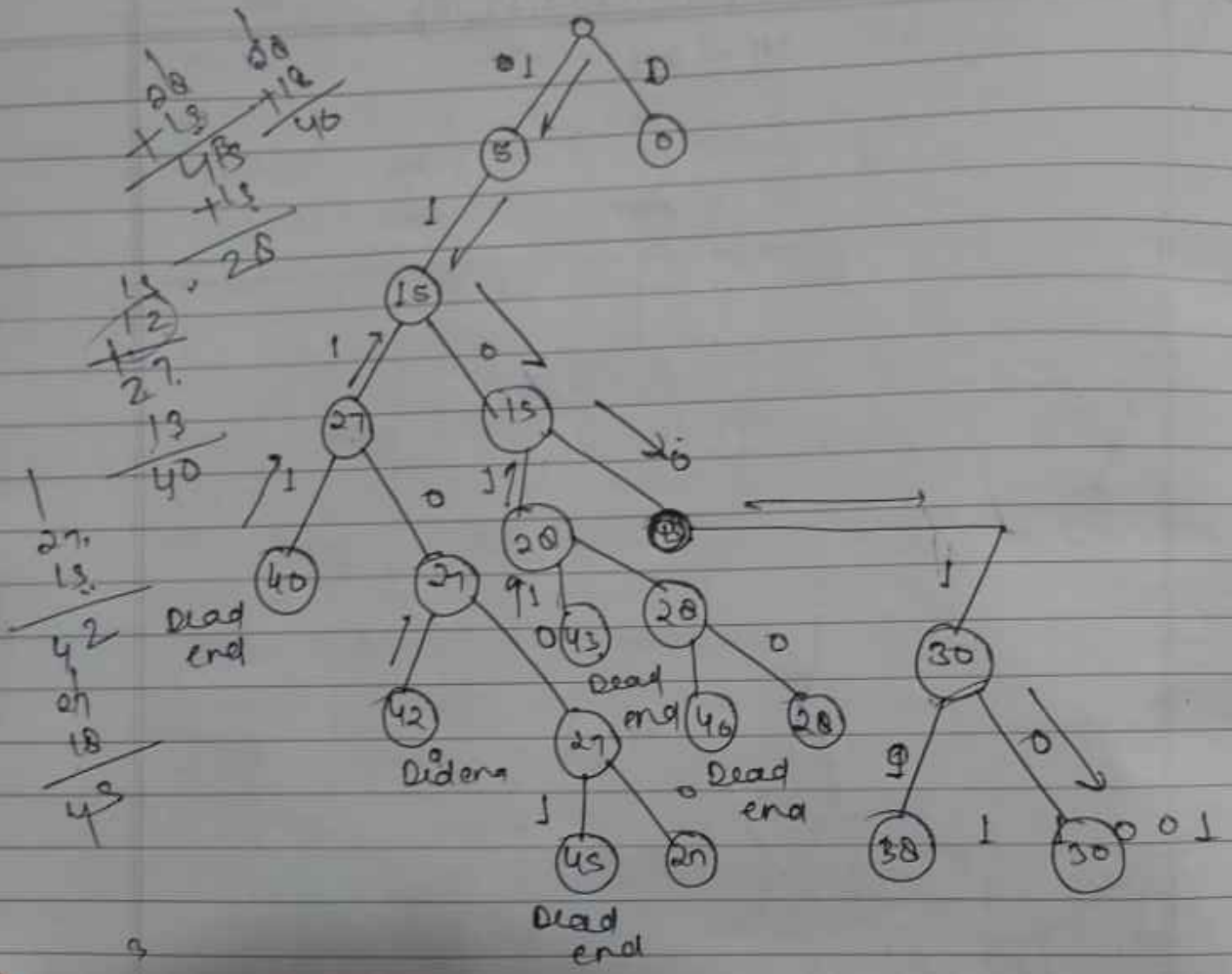
$W = \{5, 10, 12, 13, 15, 18\}$

$M = 30$



$W = \{5, 10, 12, 13, 15, 18\}$

$M = 30$



N-Queen Problem

N-queen problem is to play N-queen in such a manner on n*n chess board that know two queen attack.

↳ each other by began in the same row, column or diagonal.

It can be seen that for n=1, the problem has a trivial solution & no solution exist for n=2 and n=3, So first we will consider the four queen problem and then generate it to n queen problem.

Backtracking is a systematic way together through all possible consideration of each space. we assume our soln is a vector V, $V=\{a_1, a_2, a_3 \ldots a_n\}$ where each element a_i is selected form finite set order.

Algorithm.

1) For i = 1 to n

2) do if place(k,i)

3) then x[k] = i

4) if K=n then

5) Print x[1...n]

6) else N-Queen[K+1,n]

4-Queen Problem

4-Queen problem based on backtracking, we can also define a 4-Queen problem on the same function as 8-Queen problem. We are given 4-Queen to be place on 4x4 matrix (chess board) so that no two queen are on the same row, column or diagonal.

Here the Sol^n to the problem. in four tuple where 3 are placed with the above constrain.

Step 1.

Place first queen in the first column

	1	2	3	4
1	q_1	x	x	x
2	x	x	x	x
3	x	x	x	x
4	x	x	x	x

Step 2.

After placing first queen in the first

column we cannot place second queen in the first or second column or diagonal so we place second queen in the third column

	1	2	3	4
1	q_1	x	x	x
2	x	x	q_2	x
3	x	x	x	x
4	x	x	x	x

Step 3. After placing the first and second queen we cannot place q_3 anywhere. So we apply another placement for q_2.

	1	2	3	4
1	q_1	x	x	x
2	x	x	x	q_2
3	x	x	x	x
4	x	x	x	x

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

Step 4. Now we have option to place q_3 in column 2.

	1	2	3	4
1	q_1	x	x	x
2	x	x	x	q_2
3	x	q_3	x	x
4	x	x	x	x

move left to right

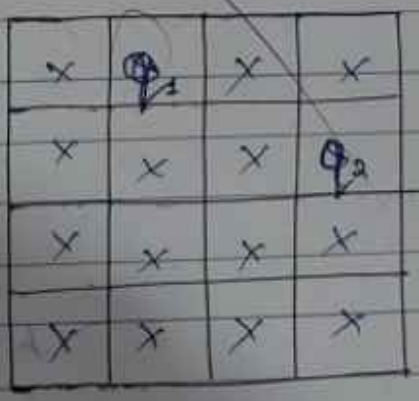
Step 4 After applying 4 q_1 in column 1, q_2 in column and q_3 in column 2 again we have no option for q_4.

Step 5. From step 4 it is notice that we have no any option to change the position of q_1, q_2, q_3 for the placement of q_4.

Hence we ~~hav~~ now have to make backtracking & examine whether there any option for placement of queen.

we now start the replacement of queen(q_3) on the column 2.

Step 6 Having place the queen q_3 in column 2 so we can place queen - 2 in the column four(4).



Step 7 Now we place q_3 in column-1

×	q_1	×	×
×	×	×	q_2
q_3	×	×	×
×	×	×	×

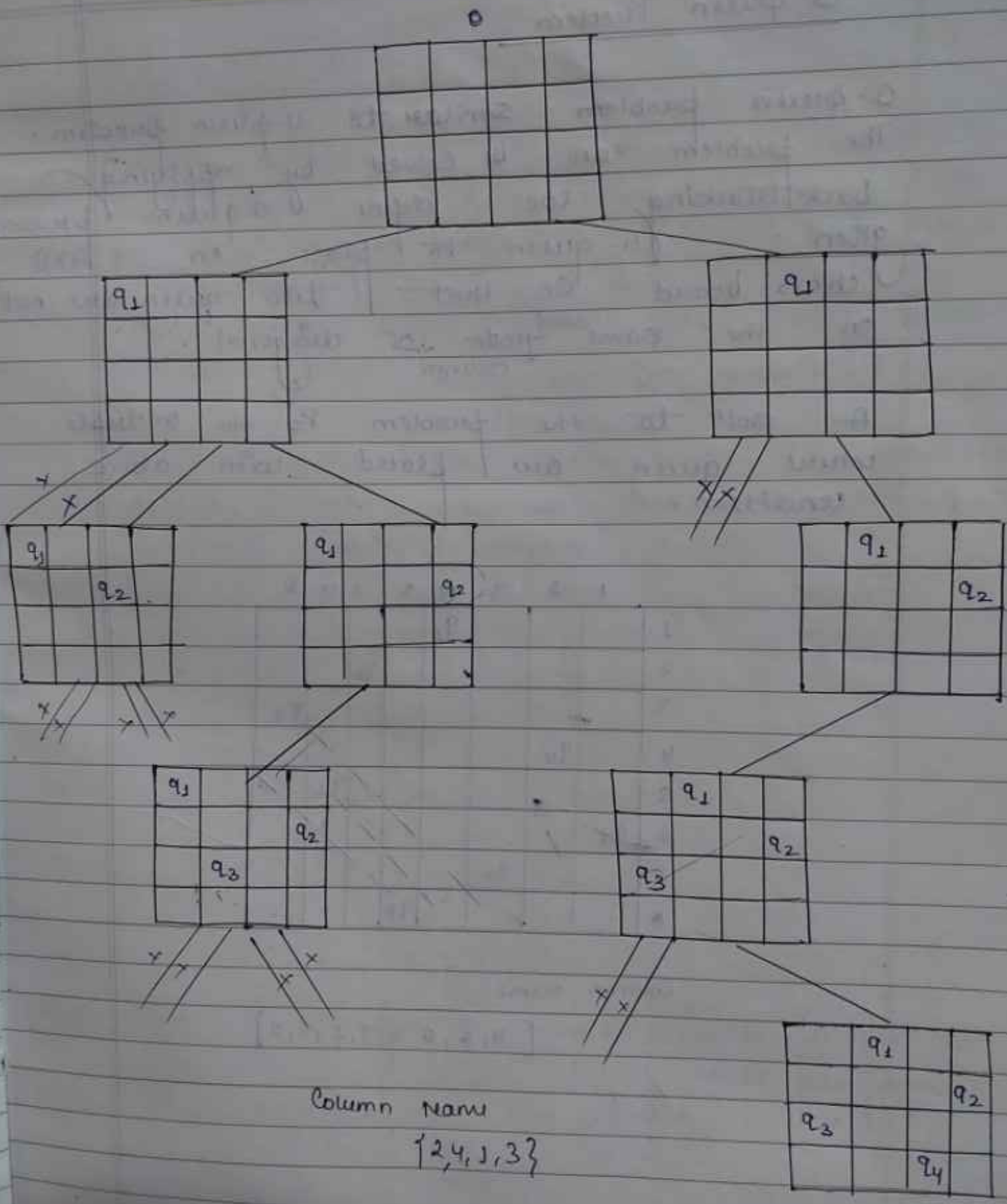
Step 8 Now after placing q_1 in column-2 & q_2 in column-4, q_3 in column-1 we can place q_4 in column-3.

	1	2	3	4
1	×	q_1	×	×
2	×	×	×	q_2
3	q_3	×	×	×
4	×	×	q_4	×

Thus we find solution

$\langle 2, 4, 1, 3 \rangle$

Shoort method for N-Queen Problem



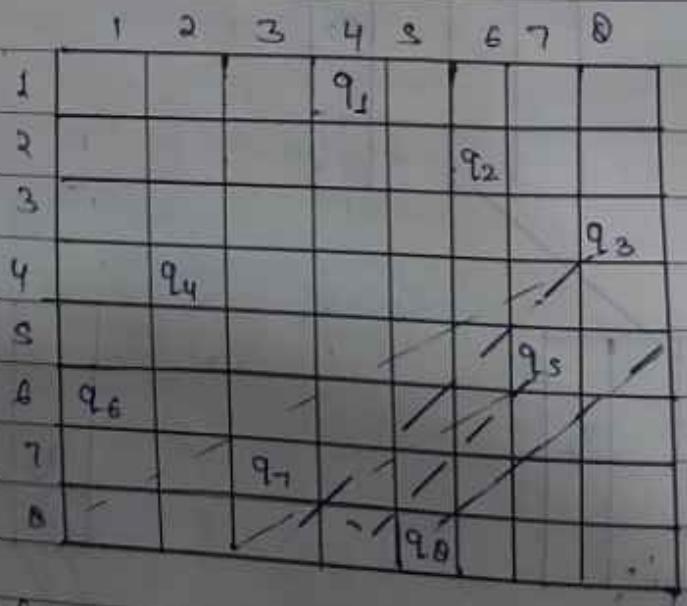
Column Name

{2, 4, 1, 3}

8-Queen Problem

8-queen problem similar to 4-queen problem. The problem can be solved by applying backtracking we define 8-queen problem, given 8-queen to place on 8×8 chess board so that two queen are not on the same row/column or diagonal.

A soln to the problem is the 8-tuple where queen are placed with above condition.



Column Name

[4, 6, 8, 2, 7, 1, 3, 5]

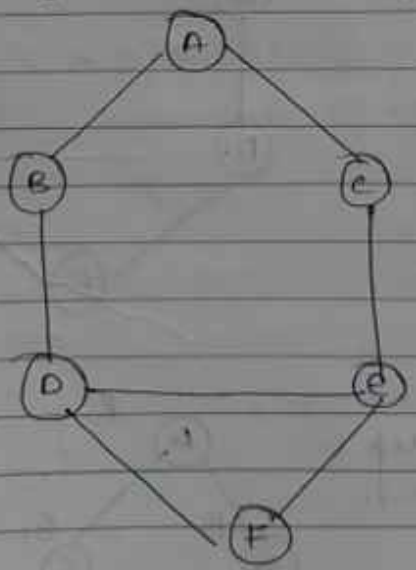
Graph Coloring

Graph coloring is a problem of coloring each vertex in a way such that no two adjacent vertex have same color.

This problem is also called as m-coloring problem.

A given figure we require three color to color the graph, Here we use backtracking technique to solve graph coloring problem.

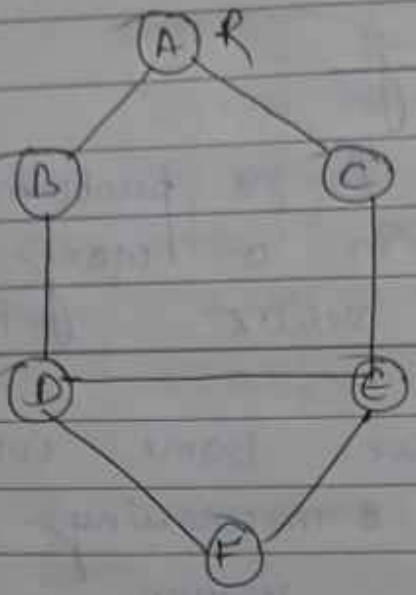
Ex



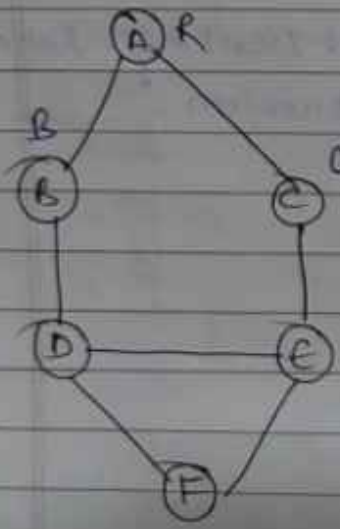
Step1 Here a graph G1 consist vertex A to F & here three color red, blue, green to apply on this graph.

Red,
Blue
Green

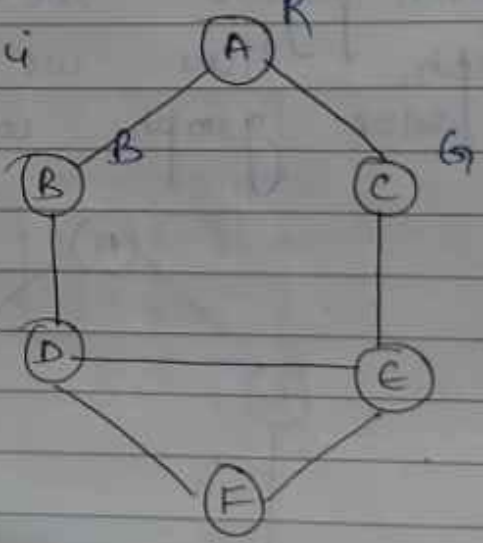
Step 2



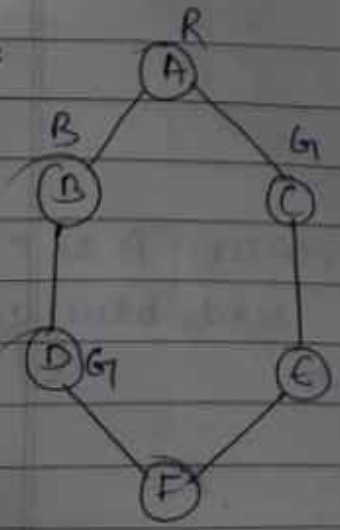
Step 3



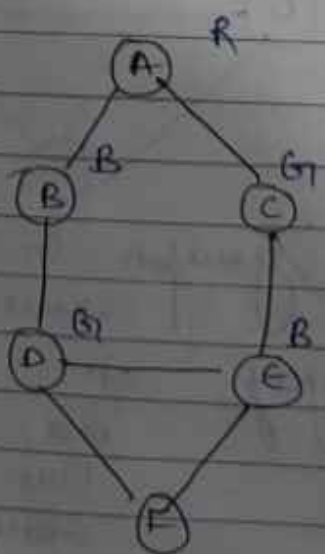
Step 4



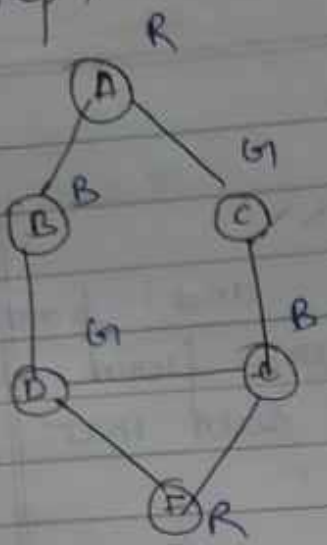
Step 5



Step 6



step 7



UNIT-5:

String Matching

Given a text array 1 to n and pattern array 1 to m of characters from alphabet Σ of finding all shift such that $T[s+1 \ldots n] = P[1 \ldots m]$.

Application of string matching

a) Text matching
b) Pattern matching

Algorithm for string matching

There are no. of algorithm for string matching-

a) Naive String Matching
b) Robin Karp string Matching
c) Boyer Moore String Matching
d) KHP string matching

KHP [Kuth-Morris Pattern]

a) Naive string Matching

Naive string matching (T, P)

1) $n \leftarrow$ length[T]
2) $m \leftarrow$ length[P]
3) for $s = 0$ to $n - m$

4) If $P[1 \ldots m] == T[S+1 \ldots S+m]$

5) Then print "Pattern occur with shift".

Ex.

T: A B C A B A B C D D

Step 1.

P: B A B

S = 0

I = 1

T: A B C A B A B C D D

P: B A B

Step 2

S = 1

I = 2

T: A B C A B A B C D D

P: B A B

Step 3.

S = 2

I = 3

T: A B C A B A B C D D

P: B A B

Step 4

S = 3

I = 4

T: A B C A B A B C D D

P: B A B

Not matching

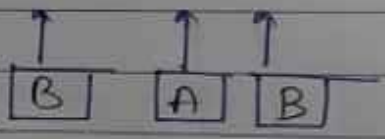
Step 5

S=4

I=5

T: A B C A B A B C D D

P:



Matching

Step 6

S=5

I=6

T: A B C A B A B C D D

P: B A B

Not matching

Step 7 S = 6
I = 7

T: A B C A B A B C D D

P: B A B

Not matching

Step 8. S = 7
I = 8.

T: A B C A B A B C D D

P: B A B

Not matching

Total iteration I = 8 and S = 5 found matching

Ques T(n) = a a c c a a b c a a b a c c a b a c c

P(m) = a a b a c ¢ $ b

Step 1.

S = 0

J = 1

T(n): a a c c a a b c a a b a c c a b a c c

P(m): a a b a c c b b

Not matching

Step 2.

S = 1

J = 2

T(n): a a c c a a b c a a b a c c a b a c c

P(m): a a b a c c b b

Not matching

Step 3.

S = 2

J = 3

T(n): a a c c a a b c a a b a c c a b a c c

P(m): a a b a c c b b

Not matching

Step 4. S = 3

I = 4

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) : a a b a c c b

Not matching

Step 5. S = 4

I = 5

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) : a a b a c c b

Not matching

Step 6. S = 5

I = 6

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) : a a b a c c b

Not matching

Step 7 S = 6

I = 7

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) : a a b a c c a b

Not matching

Step 8. S = 7
J = 8

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) : a a b a c c a b

Not matching

Step 9 S = 8
J = 9

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) : a a b a c c a b

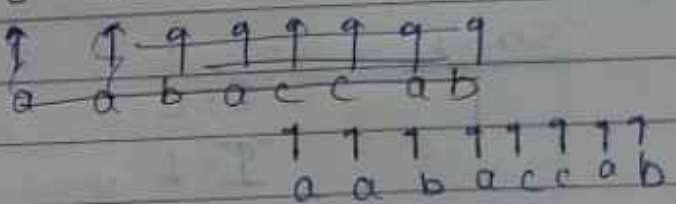
matching

Step 10 $S = 9$

$I = 10$

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) :



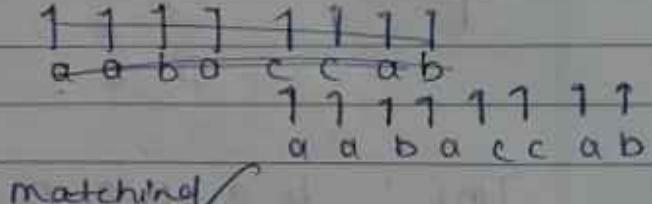
Not matching

Step 11 $S = 10$

$I = 11$

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) :



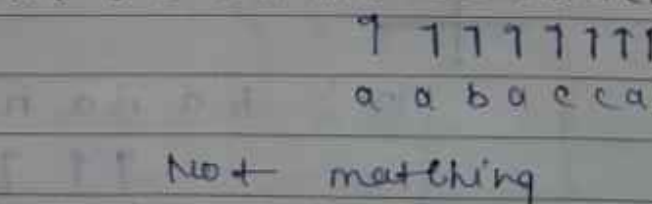
Not matching

Step 12 $S = 11$

$I = 12$

T(n) : a a c c a a b c a a b a c c a b a c c

P(m) :



Not matching

Total Iteration are 12 and $S = 9$ found matching

Q4

T(n) : b a n a n o b a b n o

P(m) : n a n o

Step 1

S = 0

I = 1 .

T(n) : b a n a n o b a b n o

P(m) : n a n o

Not matching

Step 2.

S = 1

I = 2 .

T(n) : b a n a n o b a b n o

P(m) : n a n o

Not matching

Step 3.

S = 2, I = 3

T(n) : b a n a n o b a b n o

P(m) : n a n o

Matching

S = 3 , I = 4

Step 4

T(n) : b a n a n o b a b n o

P(m) : n a n o

Not matching

Step 5

S = 4 , I = 5

T(n) : b a n a n o b a b n o

P(m) : n a n o

Not matching

S = 5 , I = 6

Step 6

T(n) : b a n a n o b a b n o

P(m) : n a n o

Not matching

Step 7

S = 6 I = 7

T(n) : b a n a n o b a b n o

P(m) n a n o

Not matching

Step 8. $s = 7$, $I = 0$

T(n) : b a n a n o b a b n a

P(n) : n a n a

Not matching

Total iteration are $I = 8$ and $s = 2$ found matching

Given 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

, q = 13

m = 5

n = 19

Step (1) Evaluate P mod q

$x = p \bmod q$

$x = [5] \bmod 13$

$x = 31415 \bmod 13$

Hash value $x = 7$

Step (2) Now in step 8 we examine other string for hash value calculate modulo q of text size m which

is match with pattern (even if does not match)

now $h = T \bmod q$

$h = 23590 \bmod 13$

$h = 0$

$7 \neq 8$ not matching

Let $d = 10$ (always)

$s = 1$ Now $t_{s+1} = [d(t_s - T(s+1)n) + T(s+m+1)]$

$t_{0+1} = [10(t_0 - T(0+1)n) + T(0+5+1)]$

$t_{+1} = [10(23590 - 10000 \times 2) + T(6)]$

$= [10(23590 - 20000) + 2]$

$t_1 = 35902 \bmod 13$

$= 9$

$\boxed{9 \neq 7}$ not matching

$S = 2$ Hash value of $T = 59023 \bmod 13$

$T = 3$

$3 \neq 7$ not matching

$S = 3$ Hash value of $T = 90231 \bmod 13$

$T = 11$

$7 \neq 11$ not matching

$S = 4$ Hash value of $T = 02314 \bmod 13$

$= 1$

$1 \neq 7$ not matching

S=5 Hash value of T = 23141 mod 13
= 12
7 ≠ 12 not matching

S=6 Hash value of T = 31415 mod 13
7 ≠ 1 not matching

S=7 Hash value of T = 14152 mod 5 = 5
7 ≠ 5 not matching

Knuth - Morries - Pratt Algorithm.

KMP or KHP algorithm keep the information name approach wasted matter during the scan of the text by avoiding.

this waste of information it achieves a running time Order of O(n+m) which acheive by using π table. The function π compute the shifting of pattern match itself. It can be observed that if we know that how the pattern match shift again itself. Then we can slide the pattern more character toward then just one character. The basic idea is to slide the pattern toward the right along the string so that the longest prefix of p that we have match the longest suffix of T that we have already match.

If the longest prefix of p that match of suffix of T is nothing then the we slide whole toward right.

Algorithm

~~n ← long~~

1) n ← length[T]
2) m ← length[P]
3) π ← Compute Prefix function (P)
4) q ← 0

5) for i ← 1 to n
6) do while q > 0 and p(q+1) ≠ T[i]
7) do q ← π[q]
8) if p[q+1] = T[i]
9) then q ← q+1
10) if q = m
11) then Print "Pattern occurs with shift" i-m
12) q ← π[q]

Ex

s = a b a b c a b c a b a b a b d

p = a b a b d

By Navie String Matching

S=0 T = a a c c a a b c a a b a c c a b a c c

I=1

P = a a b c a a b

Not matching

S=1 T = a a c c a a b c a a b a c c a b a c c

I=2

P = a a b c a a b

Not matching

S=2 T = a a c c a a b c a a b a c c a b a c c

I=3

P = a a b c a a b

Not matching

S=3 T = a a c c a a b c a a b a c c a b a c c

I=4

P = a a b c a a b

Not matching

S=4 T = a a c c a a b c a a b a c c a b a c c

I=5 P = a a b c a a b

Matching

S = 5
I = 6
T = a a c c a a b c a a b a c c a b a c c
P = a a b c a a b

Not matching

S = 6
I = 7
T = a a c c a a b c a a b a c c a b a c c
P = a a b c a a b

Not matching

S = 7
I = 8
T = a a c c a a b c a a b a c c a b a c c
P =

Not matching

S = 8
I = 9
T = a a c c a a b c a a b a c c a b a c c
P =

Not matching

S = 9
I = 10
T = a a c c a a b c a a b a c c a b a c c
P =

Not matching

S=10 T= a a c c a a b c a a b a c c a b a c c

I=11 P= a a b c a a b

Not matching

S=11 T= a a c c a a b c a a b a c c a b a c c

I=12 P a a b c a a b

Not matching

S=12 T= a a c c a a b c a a b a c c a b a c c

I=13 P= a a b c a a b

Not matching

Shift = 4

String is matching on iteration 5 ∩ Total iteration are 13 and shift = 12

* Kutt maries method

S= a a c c a a b c a a

P=

Robin Karp method

T = a a c c a a b c a a b a c c a b a c c

P = a a b c a a b

1+1+2+3+1+1+2

Hash value = 11.

S=0 T = a a c c a a b c a a b a c c a b a c c

1+1+3+3+1+1+2 = 12 Not matching

S=1 T = a a c c a a b c a a b a c c a b a c c

1+3+3+1+1+2+3 = 14 Not matching

S=2 T = a a c c a a b c a a b a c c a b a c c

3+3+1+1+2+3+1 = 14 Not matching

S=3 T = a a c c a a b c a a b a c c a b a c c

3+1+1+2+3+1+1 = 12 Not matching

S=4 T = a a c c a a b c a a b a c c a b a c c

1+1+2+3+1+1+2 = 11

Here matching found.

S=5 T = a a c c a a b c a a b a c c a b a c c

1+2+3+1+1+2+1 = 11

This is Spurious hit because hash value ps matching but string ps not matching.

S=6 T = aa cc aa bc aa ba cc ab acc

2+3+1+1+2+1+3 = 13

Not matching

S=7 T = aa cc aa bc aa ba cc ab acc

3+1+1+2+1+3+3 = 14

Not matching

S=8 T = aa cc aa bc aa ba cc ab acc

1+1+2+1+3+3+1

= 12

Not matching

S=9 T = aa cc aa bc aa ba cc ab acc

1+2+1+3+3+1+2 = 13

Not matching

S=10 T = aa cc aa bc aa ba cc ab acc

2+1+3+3+1+2 = 12

Not matching

S=11 T = aa cc aa bc aa ba cc ab acc

1+3+3+1+2+1+3 = 14

Not matching

S=12 T = aa cc aa bc aa ba cc ab acc

3+3+1+2+1+3+3 = 16

Not matching

Randomized Algorithm

Algorithm where some of action depends on chance are generally called probability or randomized algorithm

Algo. run faster than non-deterministic algo.

The result of randomized algo may not 100% correct

By some probability by running the algorithm for longer time increases the correctness of result

Numerical Algo

For some problem finding exact soln require a long time randomness can be used to find approximate numerical soln for this algorithm, produce approxim-

at result & quality of soln. If we run algorithm for long time.

Ex

$\sqrt{3} = 1.732$

Monte carlo algorithm

There are some problem (decision problem) for which we need to find exact answer (Yes/No) to approximate answer work here. Monte carlo algorithm always give result but the result may or may not be true/correct

Las Vegas Algorithm.

It is ~~always~~ may or may not be find answer but whenever we find answer, answer must be correct.

Shrewood Algorithm

The randomness reduce the difference b/w time requirement for good instance & bad instance. The Shrewood algo. not only reduce that difference

Ex:- randomized algo.
quick sort
N-queen

worst case behaviour & average case behaviour.
In Shurwood algo. always give an answer & answer is always correct.

Min-cut randomized algorithm

~~Min cut~~
Let $G_1 = \{V, E\}$ be a connected undirected loop free multigraph on n vertex. In a multigraph there may be more than one edge b/w any pair of vertex. A cut in a multigraph is a partition of a vertex set of V into in two disjoint non-empty subset $\{V_1, V_2\}$ such that

$V_1 \cap V_2 \neq \phi$

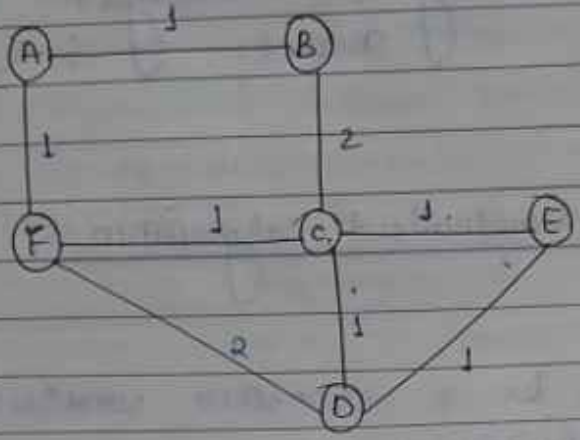
$V_1 \cup V_2 = V$

The size of cut is the no. of edge crossing the cut

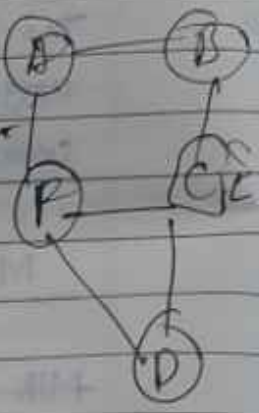
This algo does not ~~produce~~ ~~return~~ ~~the~~ produce the correct size of minimum cut.

Min-cut(G_1)
1) While $|V| > 2$
2) Do select an edge(x,y) belonging to E uniformely at random.
3) Contract the edge(x,y)
4) Return (|E|)

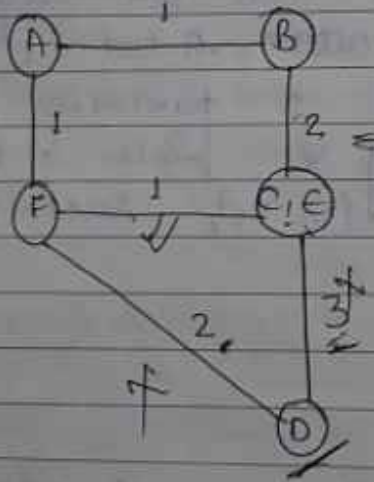
Ex.



(C, E) contract →

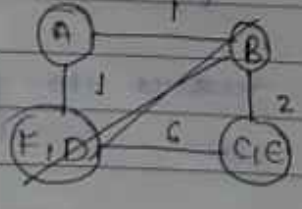


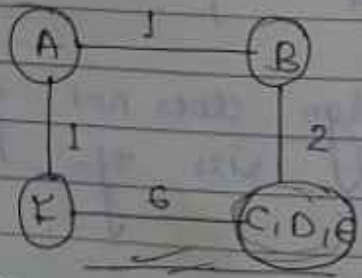
1

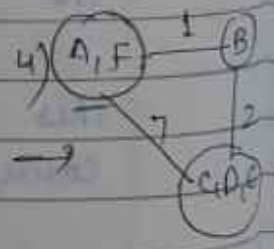


27 ✓

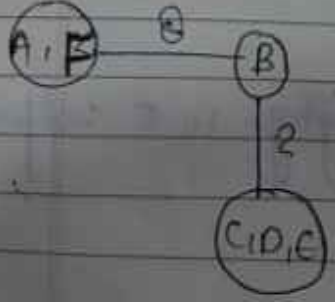
2



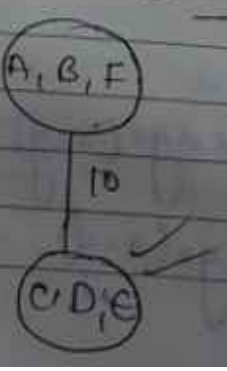


4) → 

3



4



Application of randomized algorithm

Quick sort

Minimum cost spanning tree

N-queen Problem

Travelling Salesman Problem.

Advantages of randomized algorithm

It is very simple.

High efficiency

Better complexity bound.

Random selection of good and bad choice.

Cost efficient.

Approximation Algorithm.

This algorithm is procedure which give some kind of soln for a given problem though it may fail to find optimal soln.

We can find the ratio b/w optimal soln & soln by approximation algorithm.

NP- Completness

There are many problem for which non polynomial time algorithm is "Know". Some of their problem are travelling sales man problem, optimal graph colouring, the knapsack problem, Hamiltonian cycle, finding the shortest path in a Graph. There are many examples of non-polynomial.

1) These problems belong to an Interesting (Intersecting) class of problem called the NP complete problem whose status is known

2) NP complete problem are Traceable.

3) Require a simple polynomial time. The reason is that a polynomial time algorithm to solve any one of the NP complete problem would automatically provide us with a polynomial time algorithm for all of them.

Classification of problem.

The subject of computational complexity theory is dedicated to classifying problem by how hard.

- P (stand for Polynomial time) (Deterministic)

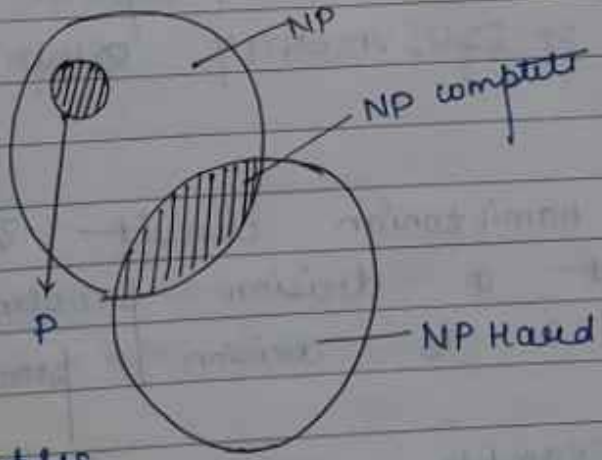
Problem that can be solve in polynomial time most specific, they are the problem that can be solve in $O(n^k)$ for some constant k

where n is the size of input to the problem

NP {NP stand for non-deterministic polynomial time} where non-deterministic is just funny way of talking about question of soln.

A problem is in NP if you can quickly ~~test~~ whether a soln is correct.

Problem in NP are still relative easily if only we could guess the right soln we could then quickly ~~test~~ it.



Examples

P → Shortest path

NPC → Vertex Colouring

NPH → Turing Machine.

Relation b/w P, NPC and NPH.

From above figure it is easy to see that NP hard problem that are not NP complete. only a decision problem can be NP complete. However an optimization problem may be hard.

Decision problem

This is example of NP complete. For this problem the answer is either YES or NO or EQUIVALENTLY. either TRUE OR FALSE.

Ex

Find hamiltonian circuit in a graph G1 is not a decision problem but its graph G1 is decision problem

Example.

Vertex colouring Problem.

NP hard problem. (Non deterministic polynomial time) :-

In computational complexity theory is the defining property of the class of problem that are informally atleast

as hard as the hardest problem in NP.

A problem H is NPH when every problem L in NP can be reduced in polynomial time H.

where L = Language

H = Problem

Application

- Datamining
- Cryptography
- Scheduling

* Each NPC problem has to be NP

NPH ≠ NPC

Boyer's Moore Algorithm

It is more efficient when pattern is long & alphabet is sumation is large.

Its matches from right to left.

It utilize to heuristics

Algorithm.

1) n ← length[T]
2) m ← length[P]
3) λ = compute last occurance
4) γ = compute good suffix
5) s = 0
6) while s ≤ n-m
7) do j = m
8) while j > 0 and P(J) = T[s+j]
9) do j = j-1
10) if j = 0
11) then print "Pattern occur with shift"
12) s = s+γ[0]
else
s = s+max[γ[j]; j→λ[T[s+j]]

T(n) = aaccaab caabacc
P(m) = aabcaab

S=0 T(n) = a a cc aabcaabacc
P(m) = a a b c a a b
x

S=1 T(n) = a a cc a a b c a a b a c c
P(m) = a a b c a a b

γ = b a a c b a a
λ = a a b c a a b

Qw Apply Boyer's moore on this string

a) T(n) = 00101100101001101

P(m) = 1100101

* Navie string method.

S=0 T(n) = 00101100101001101

J=1 x

P(m) = 1100101

not matching

S=1 T(n) = 00101100101001101

J=2 x

P(m) = 1100101

not matching

S=2 T(n) = 00101100101001101

J=3 x

P(m) = 1100101

not matching

S=3 T(n) = 00101100101001101

J=4 x

P(m) = 1100101

not matching

S=4
I=5

T(n) = 00 101 100 101 00 111 01
↑ ↑ ↑ ↑ ↑ ↑ ↑
P(m) = 1 1 00 101

matching

S=5
I=6

T(n) = 00 101 100 101 00 111 01
× ↑
P(m) = 1 1 00 101

not matching

S=6
I=7

T(n) = 00 101 100 101 00 111 01
× ↑
P(m) = 1 1 00 101

not matching

S=7
I=8

T(n) = 00 101 100 101 00 111 01
× ↑
P(m) = 1 1 00 101

not matching

S=8
I=9

T(n) = 00 101 100 101 00 111 01
× ↑
P(m) = 1 1 00 101

not matching

S=9 T(n) 0 0 1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 1

J=10 P(m) 1 1 0 0 1 0 1

not matching

S=10 T(n) = 0 0 1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 1

J=11 P(m) = 1 1 0 0 1 0 1

not matching

S=11 T(n) = 0 0 1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 1

J=12 1 1 0 0 1 0 1

not matching

String is matching on iteration
J=5 & shift=4

* Boyeurs moore

S=0 T(n) = 0 0 1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 1

P(m) = 1 1 0 0 1 0 1

S=1 T(n) = 0 0 1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 1

P(m) = 1 1 0 0 1 0 1

Y = Suffix
λ = Substring

Y = 1 0 1 0 0 1 1

λ = 1 1 0 0 1 0 1

(b) T(n) = a a c c a a b c a a b a c c

P(m) = a a b c a a b

* Navie String method.

S=0, I=1

T(n) = a a c c a a b c a a b a c c

P(m) = a a b c a a b

not matching

S=1, I=2

T(n) = a a c c a a b c a a b a c c

P(m) = a a b c a a b

not matching

S=2, I=3

T(n) = a a c c a a b c a a b a c c

P(m) = a a b c a a b

not matching

S=3, I=4

T(n) = a a c c a a b c a a b a c c

P(m) = a a b c a a b

not matching

S=4, I=5

T(n) = a a c c a a b c a a b a c c

P(m) = a a b c a a b

matching

S=5 T(n) = a a c c a a b c a a b a c c

J=6 P(m) = a a b c a a b

not matching

S=6 T(n) = a a c c a a b c a a b a c c

J=7 P(m) = a a b c a a b

not matching

S=7 T(n) = a a c c a a b c a a b a c c

J=0 P(m) a a b c a a b

not matching

String is matching on iteration 5 & shift 4.

* Boyer's moore

S=0 T(n) = a a c c a a b c a a b a c c

P(m) = a a b c a a b

not matching

S=1 T(n) = a a c c a a b c a a b a c c

P(m) λ- a a b c a a b

γ = b a a c b a a

λ = a a b c a a b

c) T(n) = 00101001 1010

P(m) = 10011

* Navie method.

S=0 T(n) = 00101001 1010
I=1 ↑
P(m) = 10011
not matching

S=1 T(n) = 00101001 1010
I=2 ↑
P(m) = 10011 not matching

S=2 T(n) = 00101001 1010
I=3 ↑
P(m) = 10011 not matching

S=3 T(n) = 00101001 1010
I=4 ↑
P(m) = 10011 not matching

S=4 T(n) = 00101001 1010
I=5 ↑↑↑↑↑
P(m) = 10011
matching

S=5 T(n) = 0 0 1 0 1 0 0 1 1 0 1 0
J=0 P(m) = 1 0 0 1 1
not matching

S=6
J=7 T(n) = 0 0 1 0 1 0 0 1 1 0 1 0
P(m) 1 0 0 1 1
not matching

S=7 T(n) = 0 0 1 0 1 0 0 1 1 0 1 0
J=8 P(m) = 1 0 0 1 1
not matching

String is matching on iteration 5 and shift 4.

* Boyer's moose.

S=0 T(n) 0 0 1 0 1 0 0 1 1 0 1 0
P(m) = 1 0 0 1 1
not matching

S=1 T(n) 0 0 1 0 1 0 0 1 1 0 1 0
P(m) 1 0 0 1 1

Y= 1 1 0 0 1
λ= 1 0 0 1 1

T(n) = a a a b a b a a b a a b a a b

(a) P(m) = a a b a b

Navie method.

S=0 T(n) = a a a b a b a a b a a b a a b

J=1 ↑

P(m) = a a b a b

not matching

S=1 T(n) a a a b a b a a b a a b a a b

J=2 ↑ ↑ ↑ ↑ ↑

P(m) a a b a b

matching found.

S=2 T(n) a a a b a b a a b a a b a a b

J=3 ↑ ↑

P(m) a a b a b

not matching

S=3 T(n) a a a b a b a a b a a b a a b

J=4. ↑

P(m) a a b a b

not matching

S=4 T(n) a a a b a b a a b a a b a a b

J=5 ↑

P(m) a a b a b

not matching

S=5 T(n) a a a b a b a a b a a b a a b

J=6

P(m) a a b a b

not matching

S=6 T(n) a a a b a b a a b a a b a a b

J=7

P(m) a a b a b

not matching

S=7 T(n) a a a b a b a a b a a b a a b

J=8

P(m) a a b a b

not matching

S=8 T(n) a a a b a b a a b a a b a a b

J=9

P(m) a a b a b

not matching

S=9

J=10 T(n) a a a b a b a a b a a b a a b

P(m) a a b a b

not matching

S=10 T(n) a a a b a b a a b a a b a a b

J=11

a a b a b

not matching

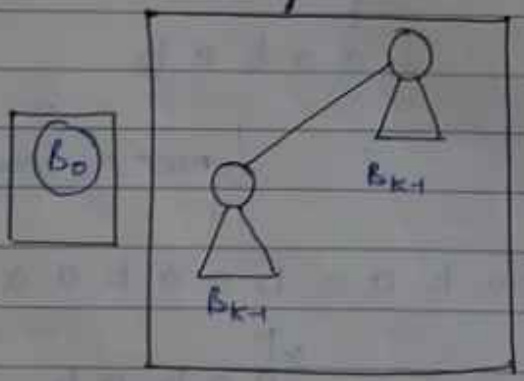
String is matching on iteration 2 and shift 1

BINOMIAL HEAP

A binomial heap is a collection of binomial tree that is a binomial heap is different size collection of binomial tree.

1) Binomial tree B_0 consist of single node.

2) For $K \geq 1$ then binomial tree B_k consist of two binomial tree $K-1$ that are link together the root of one is the left most child of the root



Properties

Each binomial tree H is heap order the key of node is $\geq$ of its parent node.

There is at most one binomial heap in H whose root has a given degree.

The first property ensures that the root of each binomial tree contain the smallest key in the tree, which apply to the entire heap.

The second property implies that a binomial heap with n elements consist of at most $\lfloor \log n \rfloor + 1$ binomial tree with n element consist n uniquely determine the no. of order of these tree

Each node contain -

A filled key for its key

A filled degree for the no. of children.

The pointer child which point to the left most child.

A binomial tree of order K has a root node whose children are root of binomial tree of order K-1, K-2, K-3 ..... K

$k = 0, 1, \ldots n$

2^0 : no. of node = 1, depth = 0, order = 0

2^1 : no. of node = 2, depth = 1, order = 1

2^2 : no. of node = 4, depth = 2, order = 2

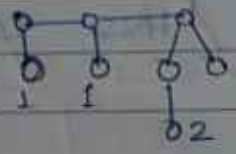
2^3 : no. of node = 8, depth = 3, order = 3

Properties:

1) $2^K \rightarrow 2^0 (B_0)$
   $2^1 = 2 (B_1)$

2) Maximum no. of nodes at depth
   $KC_I = \frac{K!}{I!(K-I)!}$

3) Height = K (depth)



4) if nod = 13

8	4	2	1
↓	↓	↓	↓
1	1	0	1
↓	↓	↓	↓
B_3	B_2	B_1	B_0

Binomial heap →

a) The degree of root ~~are~~ increases as ~~traverse~~ the root list

Union of two binomial heap

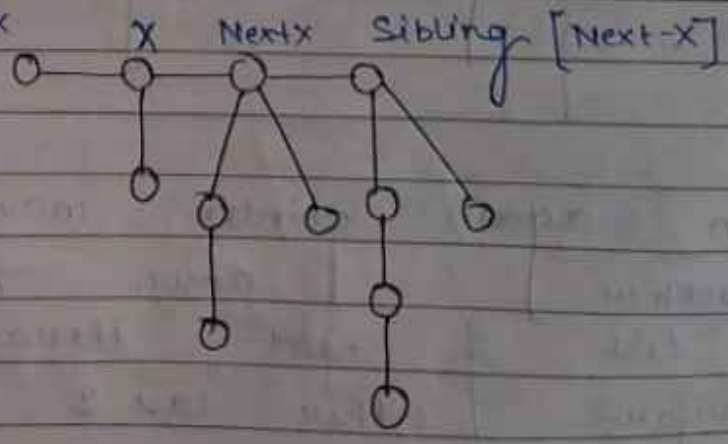
If we have two binomial heap H_1 and H_2 then by using BINOMIAL-HEAP-UNION (H_1, H_2). One can create a new heap containing all the nodes of H_1 & H_2 before implementing the algorithm for union of two binomial heap. Let us make following assumptions.

1. X.p point to the root i.e., currently being checked
2. Prev x point to the root preceding x on the root list.

Sibling [Prev-X] = X

3. Next x point to the root following x on the root list,

Sibling [x] = Next x

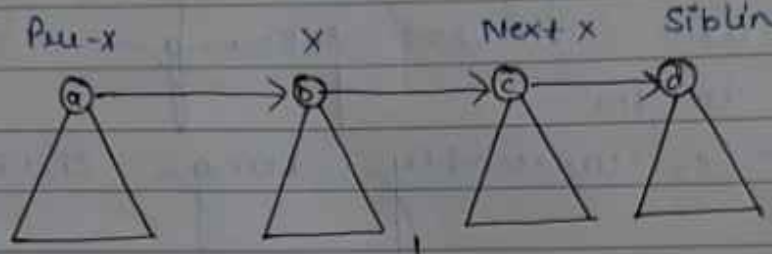


There are four cases in binomial heap

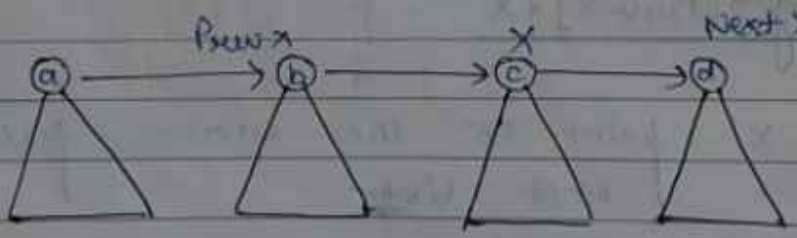
Case I

If degree of [x] is not = degree of [next-x]

then pointer move one position further down to the root list



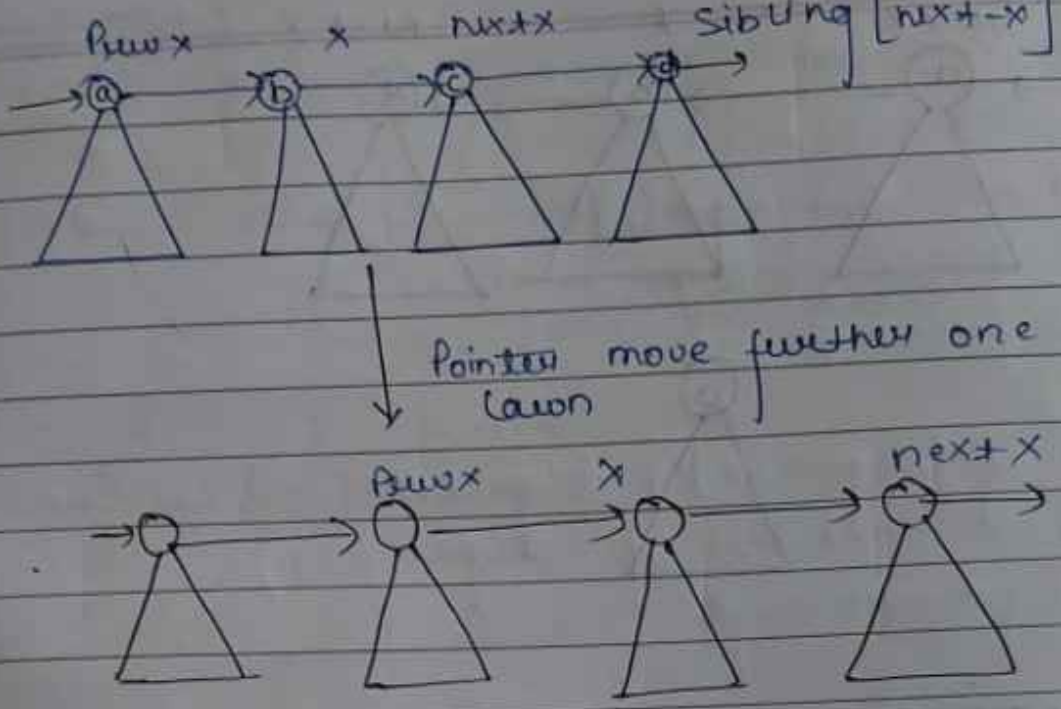
CASE 1 - Pointer move one position down



Case II

If degree [x] = degree of [next-x] = degree of [sibling [next-x]]

then again pointer move one position further down to the root list & next situation may perform either case 3 or case 4



Case III

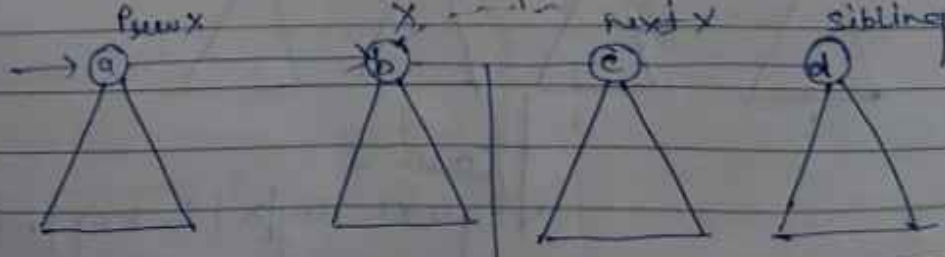
if

degree of [x] = degree [next-x] ≠ degree [sibling [next-x]]

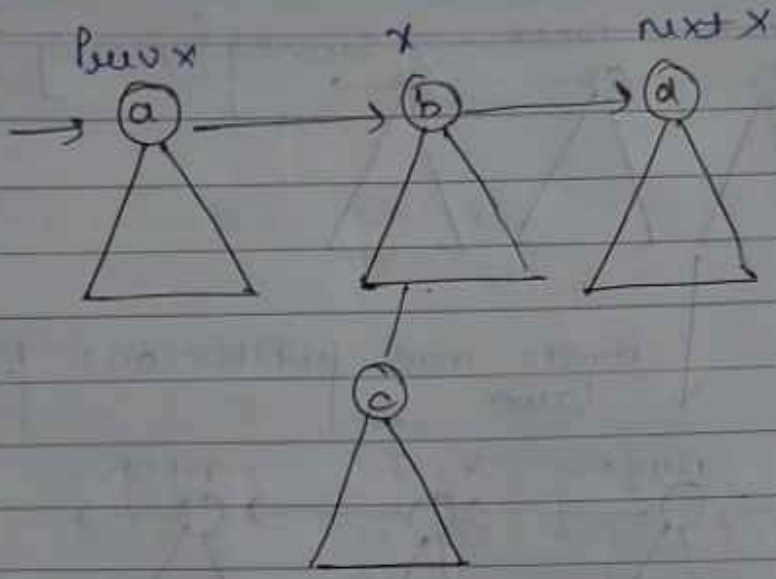
and key [x] ≤ key [next-x]

then remove next-x from the root list & link it to x creating tree

Case-3 equal key [x] ≤ key [next-x]



remove next x from root list & links to x

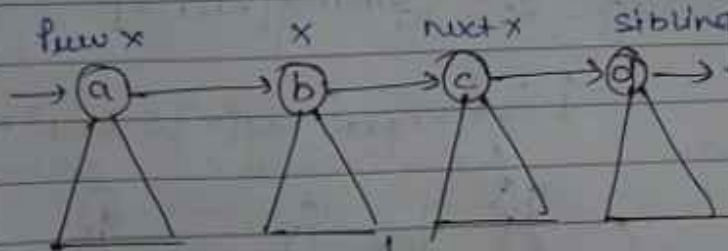


Case IV

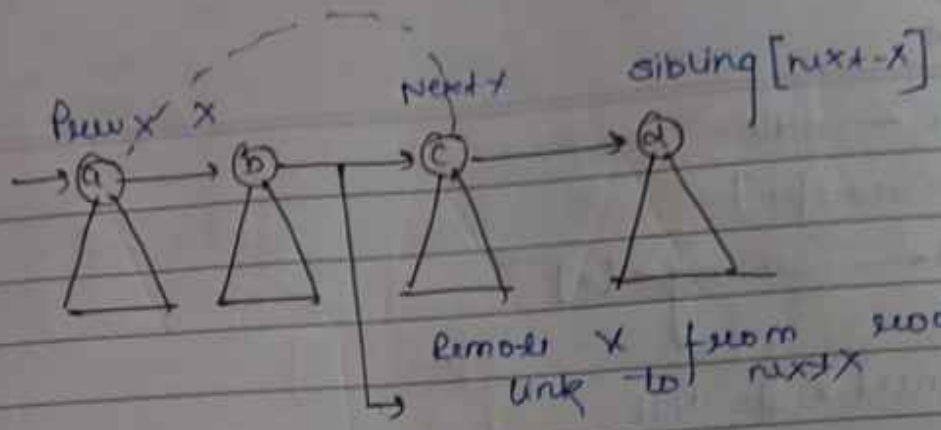
If degree of [x] = degree [next-x] ≠ degree [sibling [next-x]]

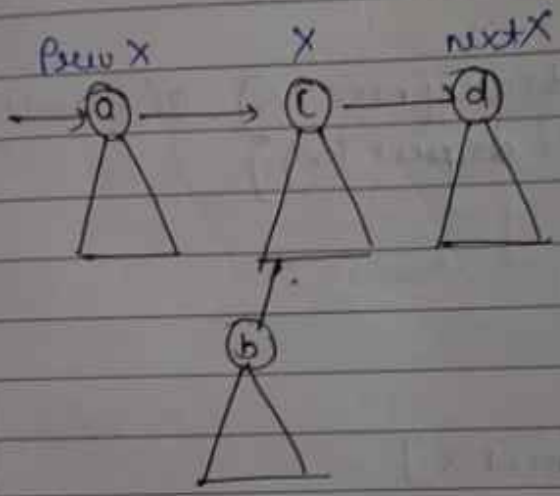
& Key [x] ≥ Key [next-x]

then remove x from the root list & link to next x. Again creating tree



Case-4
equal Key [x] ≥ Key [next-x]





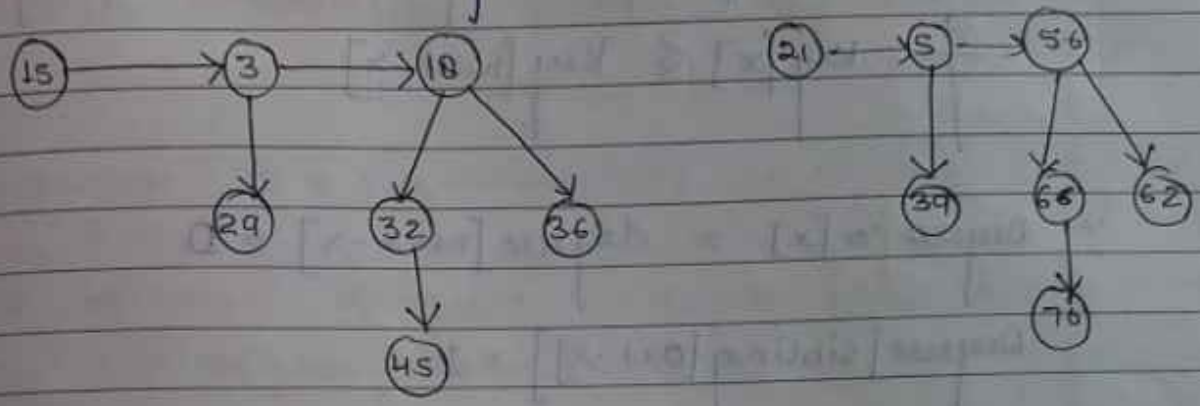
Algorithm.

If ---

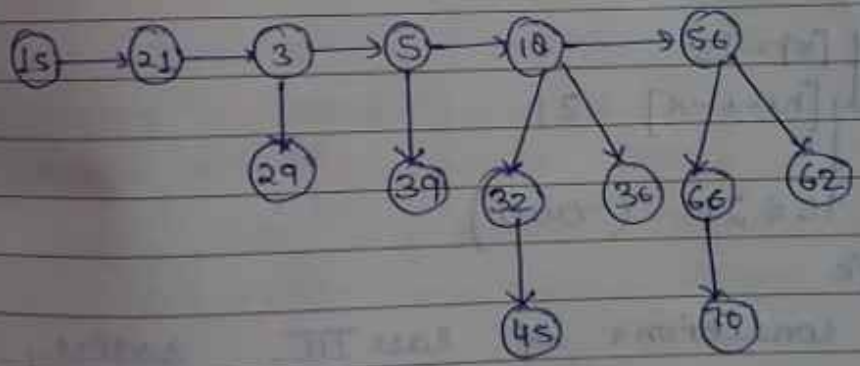
1. H ← CREATE_BINOMIAL HEAP[x]
2. Head[H] ← Binomial_heap_merge (H_1, H_2)
3. Free the object H_1 and H_2 but not the list they point to
4. If Head[H] = NIL
5. Then Return H

```
6.  Prev x ← NIL
7.  x ← head[H]
8.  Next x ← sibling [x]
9.  while next x ≠ NIL
10. do if (degree(x] ≠ degree(next-x) or (sibling(next-x]
          ≠ NIL & degree(x))
11  then Prev x ← x
12  x ← next x
13  else if key[x] ≤ key[next x]
14  then sibling [x] ← sibling [next-x]
15  else if Prev-x = NIL
16  Then head[H] ← next x
17  else sibling [Prev x] ← next [x]
18  BINOMIAL-LINK[x, next x]
19  x ← next x
20  Next x ← sibling [x]
21  Return
```

Two heap H_1 and H_2 - find the union of these two heap.



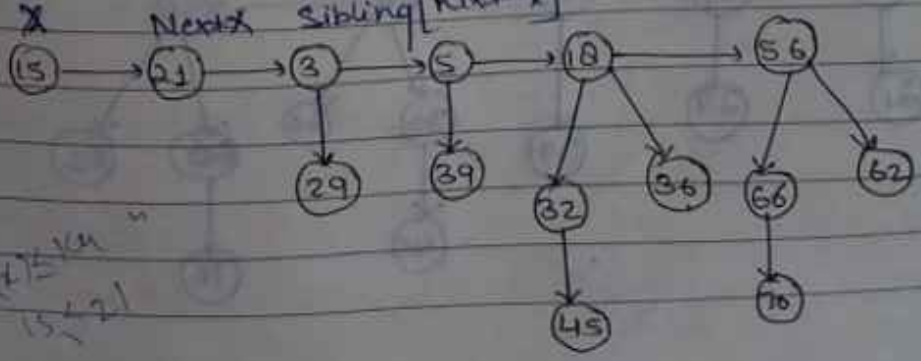
Step1. Apply head[H] ← Binomial heap merge (H_1, H_2)



Step2 If head[H] = nill

but here head[H] ≠ Nil so, Now, set x, Prev x, next x

& sibling [next-x]



Step 3 Since from binomial tree it is clear that degree of x = degree [next-x] ≠ degree [sibling [next-x]] & key[x] ≤ key[next-x]

* Degree [x] = degree [next-x] = D

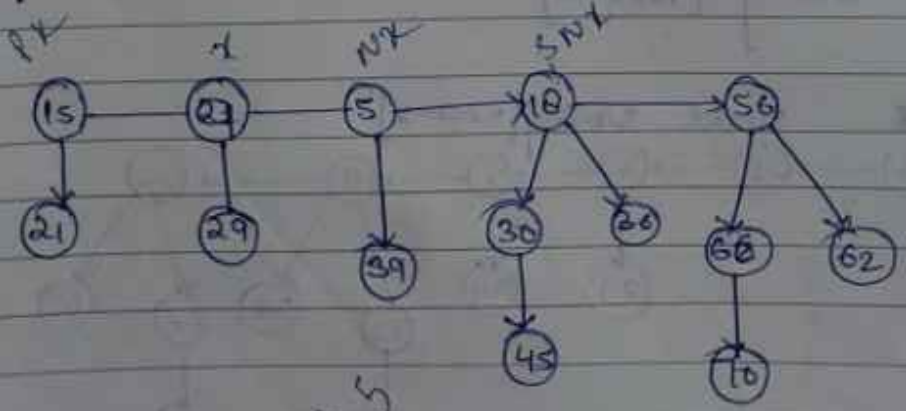
Degree [sibling [next-x]] = 1

Case III satisfy then check key[x]

key [x] = 15
key [next-x] = 21

15 ≤ 21 (true)

Therefore All conditions of case III satisfy then according to case III remove [next-x] from the root list & link it to x, creating tree.



Step 4.

Now we find that degree[x] ≠ degree[next-x]

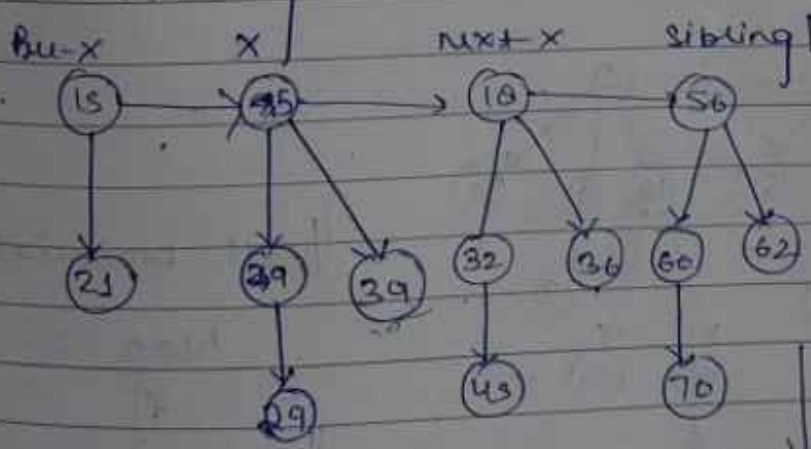
degree[x] ≠ degree[next-x] then case I apply

but in this diagram degree[x] = degree[next-x] ≠ degree[sibling[next-x]]

& also key[x] ≥ key[next-x] i.e.,

9 ≥ 5 (true)

i.e., case IV is satisfy then remove [x] from root list & link it to next-x & make tree



remove [x]

Now again degree[x] ≠ degree[next-x] ≠ degree[sibling[next-x]]

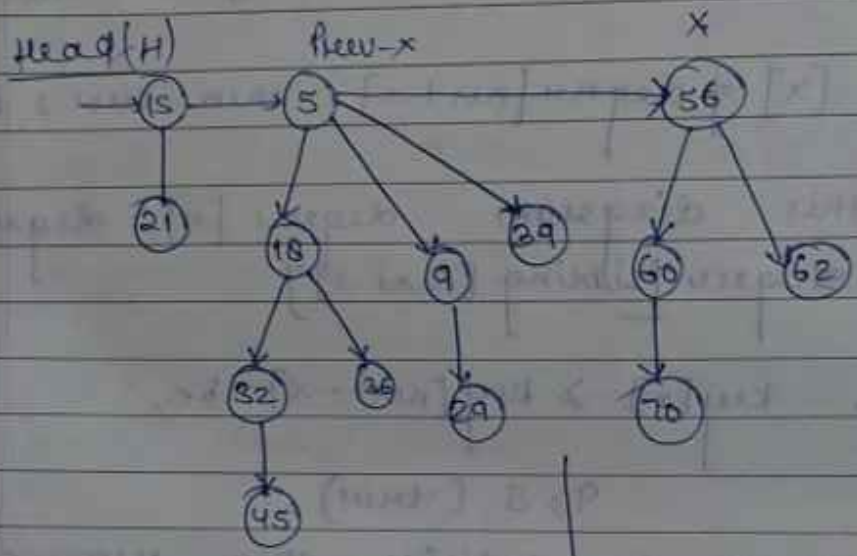
& also key[x] ≤ key[next-x]

key[x] = 5

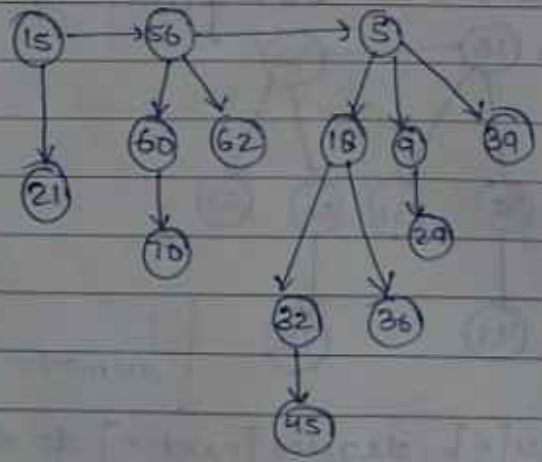
key[next-x] = 18 i.e., 5 ≤ 18

~~So the~~

Step 5 Satisfy the above condition so remove [Next x] from the root list & link it to x



Orders change



Final binomial heap.

3 Insertion of node into a {minimum key} binomial heap

The procedure binomial heap insert [H, x] inserts a new node x into given binomial heap H. Here we assume that the node x already allocated & key x already being filled.

"Inserting a new element to a heap can be done by simply creating new heap containing this element & merging only it with the original heap"

Algorithm.

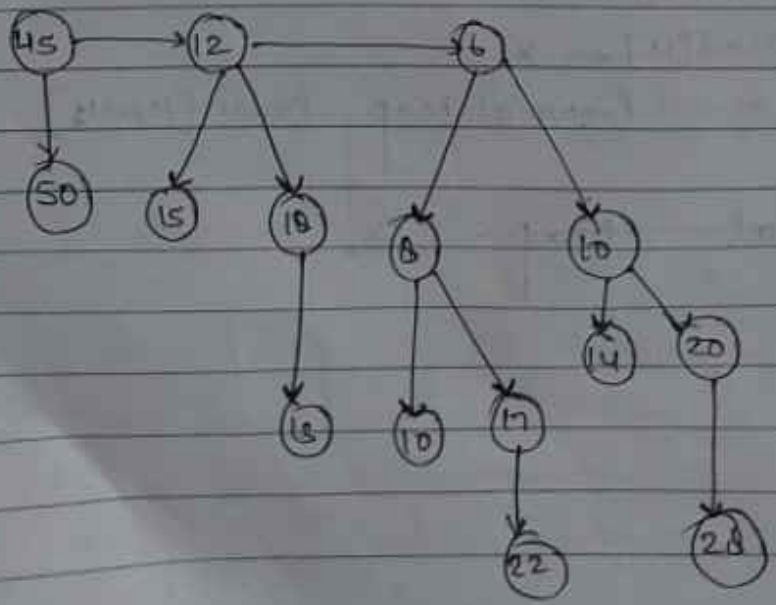
1. H_1 ← CREATE BINOMIAL HEAP
2. P[x] ← NIL
3. Child[x] ← NIL
4. degree [x] ← 0
5. head[H] ← x
6. H ← Binomial heap Union [H_1, H_2]

. ~~Binomial heap~~ ex

* BINOMIAL HEAP EXTRACT MINIMUM

1. Find the root x with minimum key in the root list of H & remove x from the root list of H.
2. H_1 ← CREATE BINOMIAL HEAP
3. Remove the order of the link list x child & set H to point to the head of the resulting list.
4. H ← Binomial heap union $[H, H_1]$
5. Return x

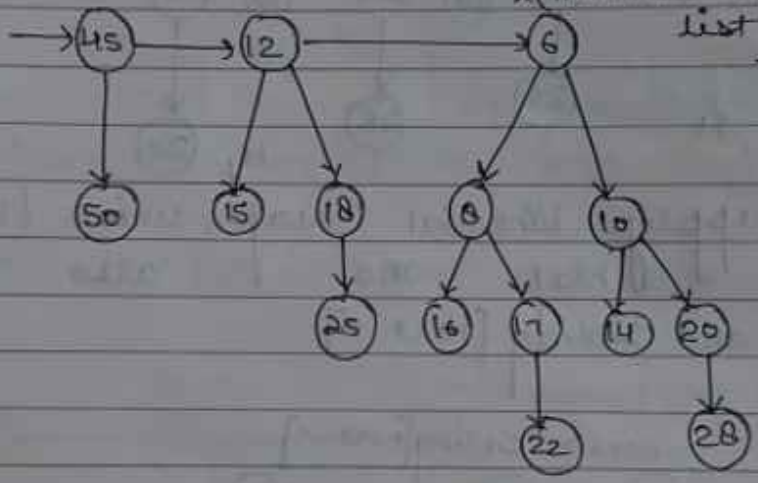
Apply the binomial heap extract minimum on the following given binomial heap



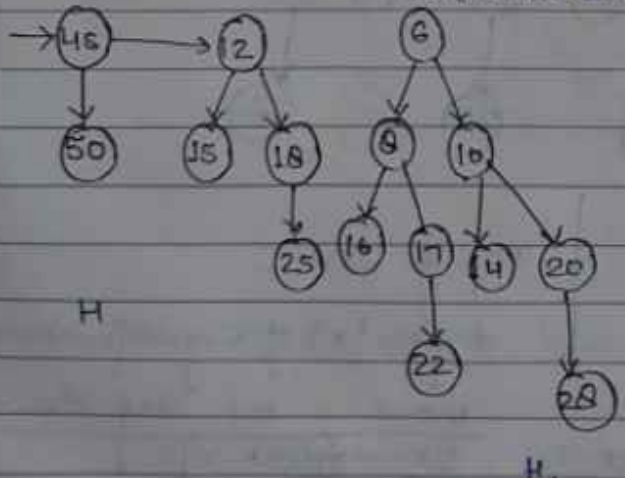
L → Small
R → Large

Step 1

Finding node x having minimum Key in the root list of edge



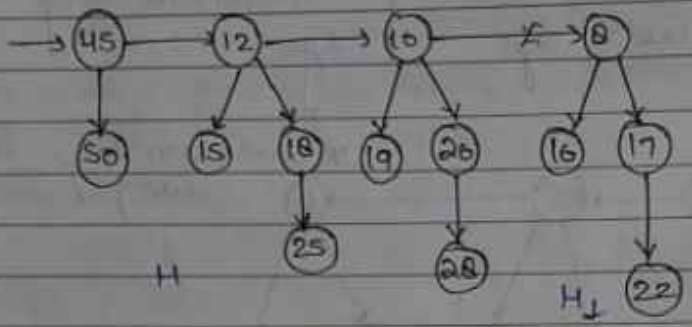
Step 2. Create a binomial heap H1 before deleting node x.



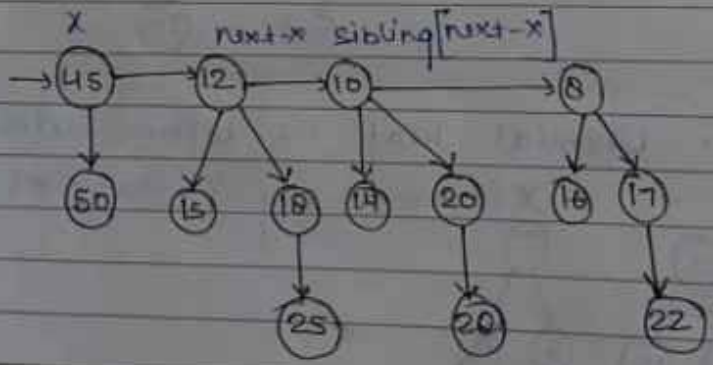
H

H_1

Step 3. Remove the node x & reverse the Order of link list of x children & set head H_1 to point to the head of resulting list



Step 4. Now apply binomial heap union (H, H_1) for binomial heap and also set x, Prev-x and Sibling [next-x]

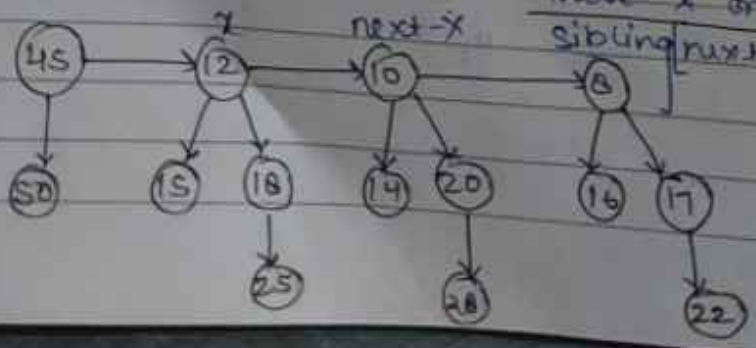


apply case I

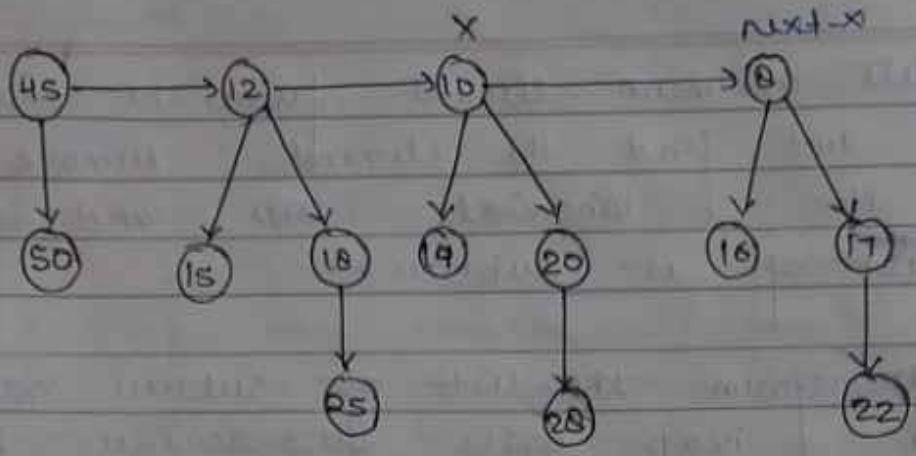
degree[x] ≠ Degree[next-x]

move x one step down

sibling[next-x]

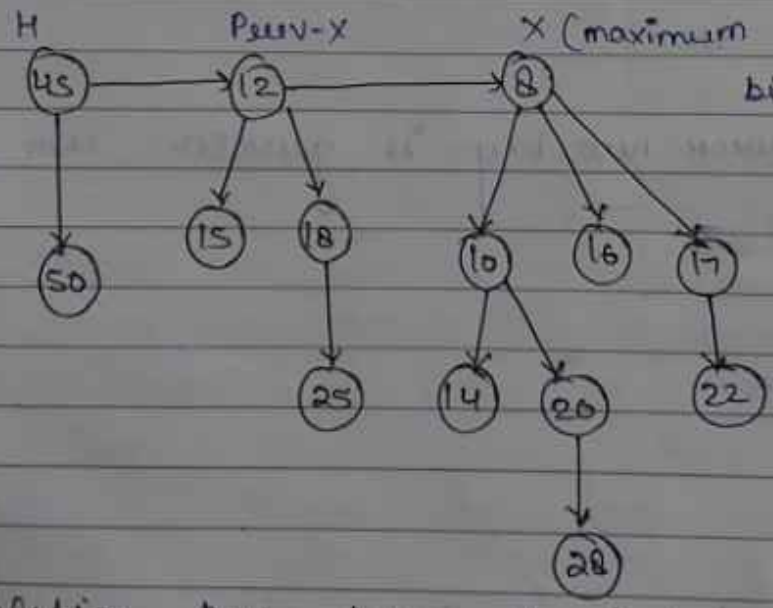


case 2 degree[x] = degree[next-x]



Satisfy case 4 then we compare Key[x] and Key[next-x], key[x] = 10 and Key[next-x] = 8

Remove the x in root node & link it [next=x]



Deletion key from binomial heap.

The operation in binomial heap decreases (H, x, k) assigned a new key to node x in a binomial heap

[next[x]]

To delete minimum element from the heap first and find the element, remove it from its binomial tree and obtain a list of it subtree.

Then transform the list of subtree into binomial heap while reordering from smallest to biggest order & merge this heap to original heap

Algorithm

BINOMIAL HEAP_DECREASE KEY(H,X,K)

if K > Key[x] then

Message "error new key is greater than current key"

Key[x] ← K

y ← x

z ← p(y)

while (z ≠ NIL) and Key[y] < Key[z]

do exchange Key[y] ⟷ Key[z]

y ← z

$z \longleftarrow p[y]$

BINOMIAL HEAP DECREASE [H, x]

1) BINOMIAL Heap - decrease Key [H, x, ∞]

2) Bnomial heap extract MIN[H]

∩

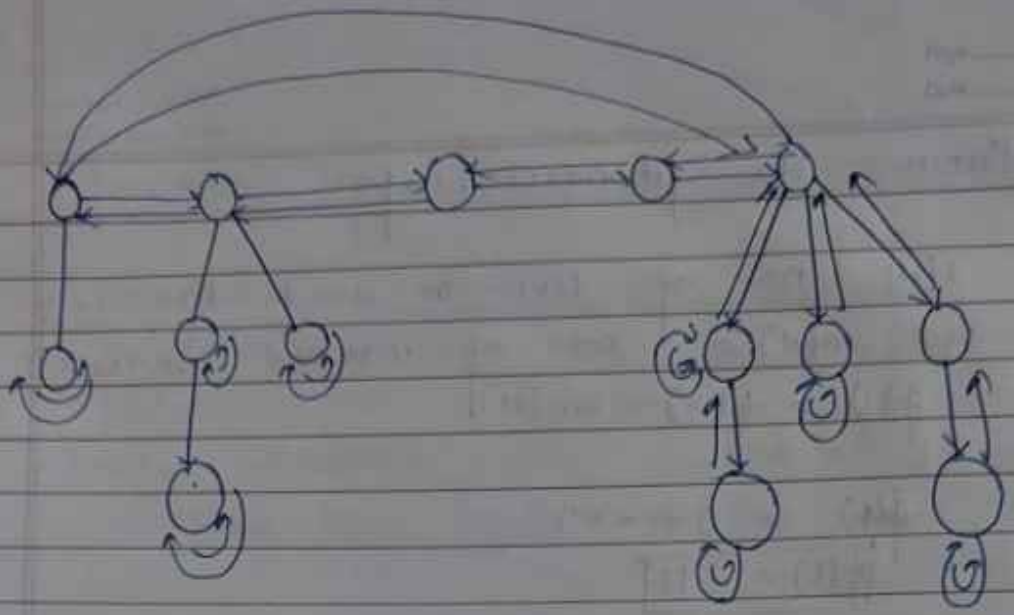
Fibonacci heap

Fibonacci heap is a collection of tree find out minimum order 1 amortized time. Operation insertion, decrease key and merge work in constant amortized key.

Fibonacci heap the heap order mean each node is smaller than each of its children unlike binomial heap tree may have many tree. In fabonacci heap are not necessary a binomial tree & also a rooted but not ordered

Advantage of circular double linked list

- We can remove a node from circular, double linked list in order on time
- We can concate to list into circular, double linked list in order one time
- The fibonacci heap are unordered binomial tree



Circular doubly structure of fibonacci heap

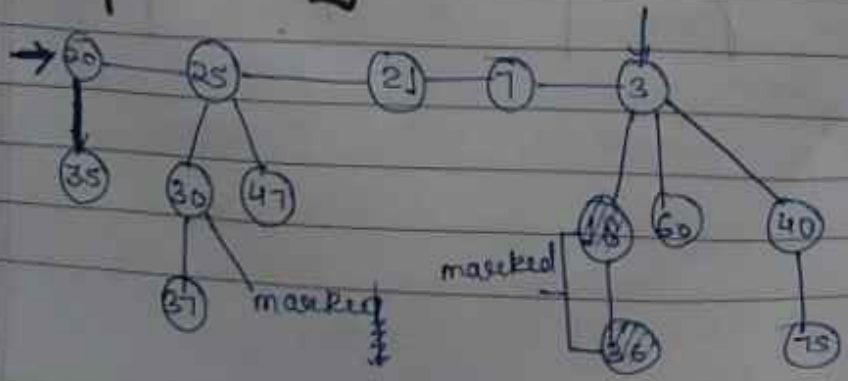
Potential function

Fibonacci heap view as amortized data structure If H is any fibonacci heap t(H) represent the no. of tree in the root list of H and if m(H) represent the no. of marked node in H. The potential function $\phi(H)$ is define as -

$$\phi(H) = t[H] + 2m[H]$$

Ex.

Find the potential fun^n $\phi[H]$ for the fibonacci heap given below.



From the fibonacci heap.

t(H) = no. of list in root list H = 5

and m(H) = no. of marked nodes = 3

$\phi(H) = t(H) + 2m(H)$

$\phi(t) = 5 + 2 \times 3$

$\boxed{\phi(t) = 11}$

Operation on fibonacci heap.

1. Creation of an empty fibonacci heap
2. Inserting node in fibonacci heap.
3. Finding the node with minimum key.
4. Union of two fibonacci heap.
5. Extracting the minimum key node from fibonacci heap
6. Decreases key operation. from fibonacci heap.

Creation of an empty fibonacci heap

The procedure CREATE FIBONACCI HEAP creates an empty fibonacci heap. It allocates & returns the procedure fibonacci heap object H where n[H] = 0 (no any node present) & min[H] = NIL

This show that there are no any tree exist in fibonacci heap H. Therefore

t(H) = number of tree in root list

$$t(H) = 0$$

Potential function $\Phi(H) = t(H) + 2m(H)$

$$\Phi(H) = 0$$

The amortized cost of creating fibonnaci heap is equal to its order + actual cost

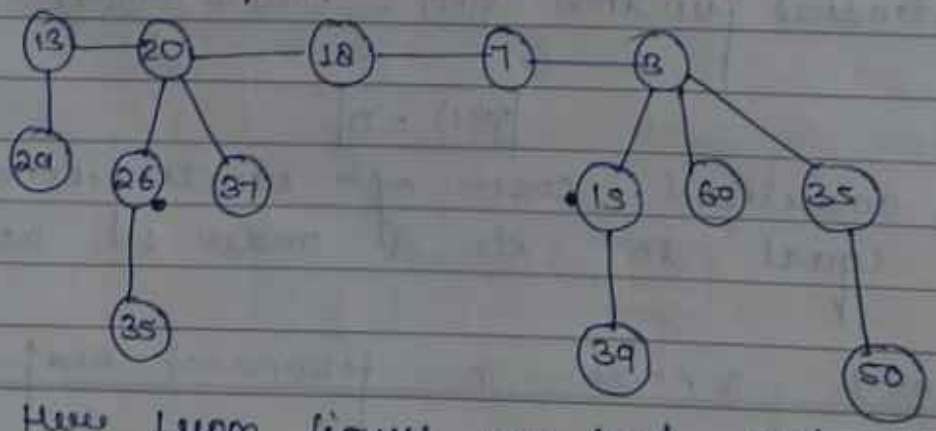
Insert a node in fibonacci heap

The procedure (Fibonacci heap) insert (FIBONACCI HEAP INSERT (H, x)) is used to insert node x in fibonacci heap H. Here we assume that the node is allocate & contain the desire key values

ALGORITHM

1. degree[x] ← 0
2. P[x] ← NIL
3. child[x] ← NIL
4. left[x] ← right[x] ← x
5. mark[x] ← FALSE
6. Concatinate the root list containing x with root list H.
7. if [min[H] = NIL OR key[x] < key[min[H]]] then
8. min[H] ← x
9. n[H] ← n[H] + 1

* Consider the fibonacci heap in this diagram we have to insert a new node as in this heap



Step 1. Here from figure we find out that degree[x]=0

P[x] = NIL

child[x] = NIL

left[x] ← right[x] ← x

mark[x] ← FALSE

$X = 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1$ (1 2 3 4 5 6 7 8)

$Y = 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0$ (1 2 3 4 5 6 7 8 9)

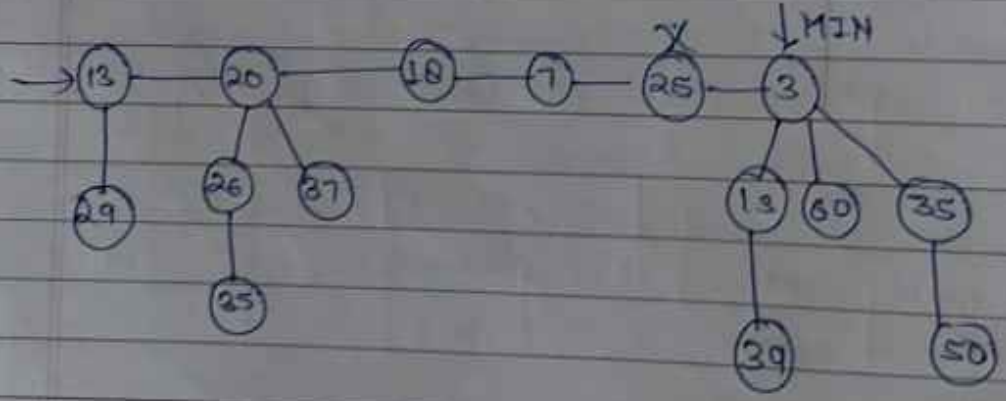
		0	1	0	1	1	0	1	1	0
		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	1	1	1	1	1	1	1
0	2	0	1	1	2	2	2	2	2	2	2
0	3	0	1	1	2	2	2	3	3	3	3
1	4	0	1	2	2	3	3	3	4	4	4
0	5	0	1	2	3	3	3	4	4	4	5
1	6	0	1	2	3	4	4	4	5	5	5
0	7	0	1	2	3	4	4	5	5	5	6
1	8	0	1	2	3	4	5	5	6	6	6

0 0 1 0 1 0

Now concat the root list containing x with root list H

Now examine the conditions:

If min(x) = NIL or Key[x] < Key[Min(H)]
if (3 ≠ NIL or 25 < 3)



Step 3. Now here we implement the value of n[H]

$$n[H] \leftarrow n[H] + 1$$

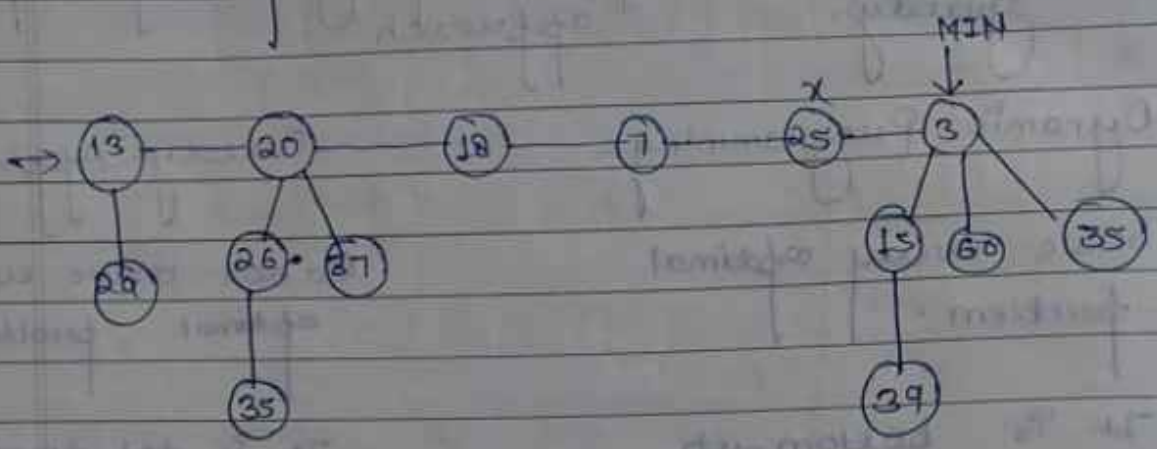
Initially n[H] = no of nodes present in Fibonacci heap = 14

$$n[H] \leftarrow n[H] + 1$$
$$\leftarrow 14 + 1$$
$$\boxed{n[H] \leftarrow 15}$$

Difference b/w dynamic programming & greedy approach

Dynamic Programming	Greedy Approach.
Solve every optimal sub-problem.	Do not solve every optimal problem.
It is bottom-up approach.	It is top-down approach.
Dynamic programming works when a problem has following properties.	Greedy algorithm works when a problem has following properties
a) optimal sub-structure b) Overlapping sub-problems.	a) Greedy choice property b) optimal substructure.

now stop the execution.



* Union of two fibonacci heap

Suppose H_1 and H_2 are two fibonacci heap & one want to find the union of H_1 and H_2.

Suppose H be the union of H_1 and H_2 i.e.,

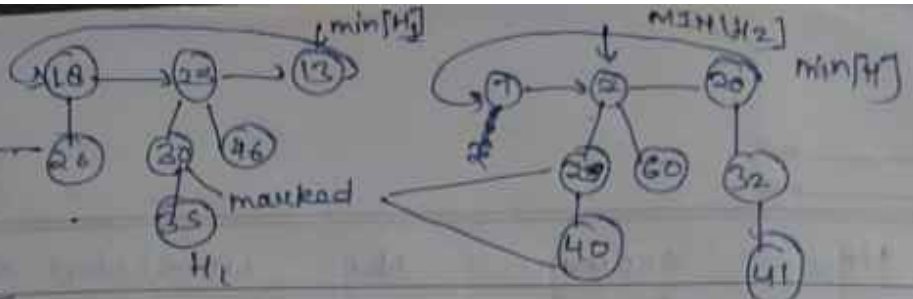
$$H = H_1 \cup H_2$$

Then for finding H the procedure FIBONACCI HEAP UNION (H_1, H_2) is used.

The procedure unit fibonacci heap H_1 and H_2 destroying H_1 and H_2 in the process.

It simply concate the root list of H_1 and H_2 & then determine the new minimum node that is determine new min[H] for fibonacci heap H

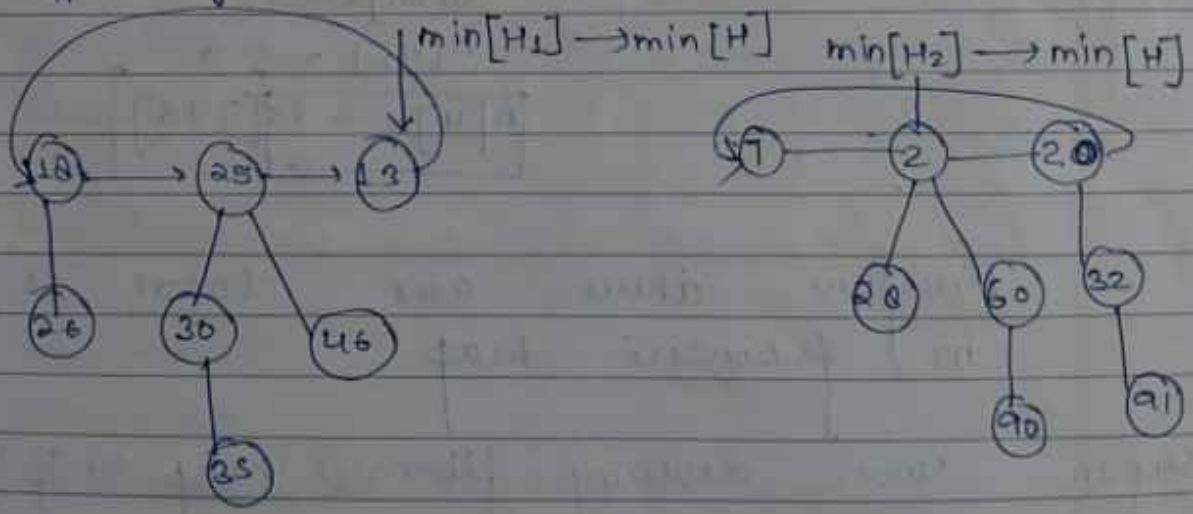
Ex



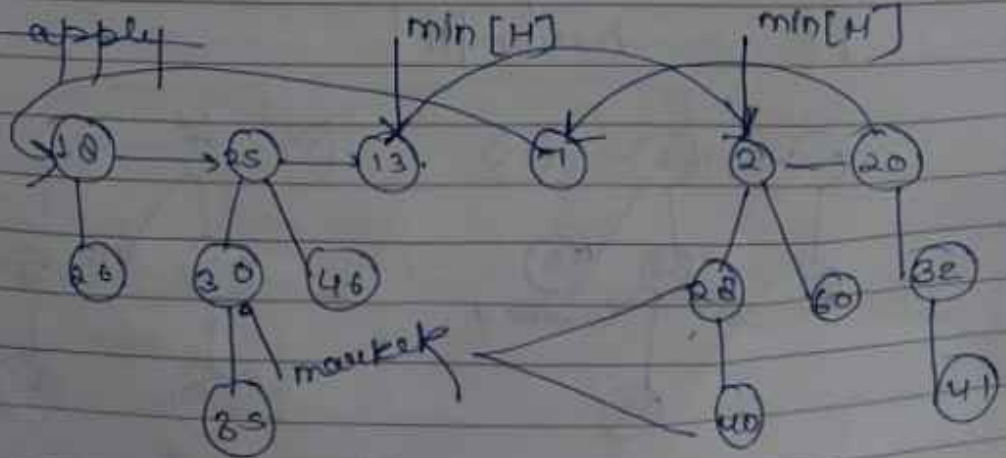
ALGO

1) H ← CREATE - FIBONACCI - HEAP()
2) MinH ← min[H1]
3) Concatenate the root list of H2 with the root list H.
4) If (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5) then min[H] ← min[H2]
6) n[H] ← n[H1] + n[H2]
7) Free the object H1 and H2
8) return [H]

Step-1 First of all create a new fibonacci heap H



Now apply



H = H1 ∪ H2

Now examine the condition as if
$min[H_1]$ = nill or $min[H_2] \neq$ NIL &
$min[H_2] < min[H_1]$

if 130 ≠ NILL OR 2 ≠ NILL &

2 < 13 (true) i.e., condition examine in if statment is true therefore increment the values

Step 3 Now apply increment the no. of node $n[H] \leftarrow n[H_1] + n[H_2]$

$n[H_1] \leftarrow 7$

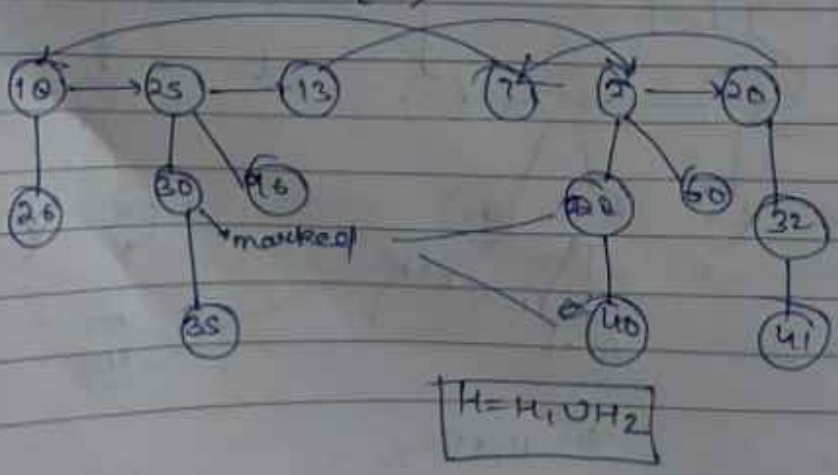
$n[H_2] \leftarrow 8$

$n[H] \leftarrow 15\{7+8\}$

Therefore there are total 15 node in fibonacci heap.

Step 4. Now destroy fibonacci heap H_1 & H_2

Step 5 return [H]



$H = H_1 \cup H_2$

Extracting the minimum key node from fibonacci heap

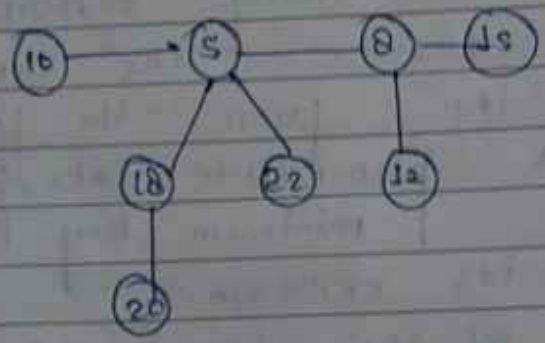
By procedure fibonacci heap extract min[H] is used to extract the minimum key node from the from the fibonacci heap H this procedure take place by deleting minimum key node & then concate its children to the root list of the fibonacci heap & then consolidate tree so that no two roots have the same degree.

Algorithm

1) z ← min[H]
2) If z ≠ NIL
3) Then for each child x of z
4) do add x to the root list of H
5) P[x] ← NIL
6) Remove z from the root list of H
7) if z = right[z]
8) The min[H] ← NIL
9) else min[H] ← Right[z]
10) Consolidate [H]
11) n[H] ← n[H] + 1
12) Return

Example.

Consider the fibonacci heap given figure & find out minimum extract the node with minimum key.



Step 1. $z \leftarrow min[H]$

$z \leftarrow 5$

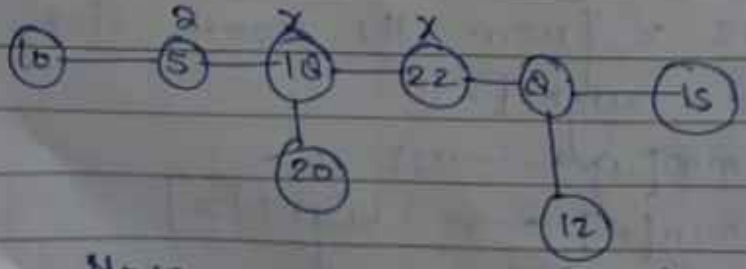
If $z \neq NIL$

Then for each child x of z

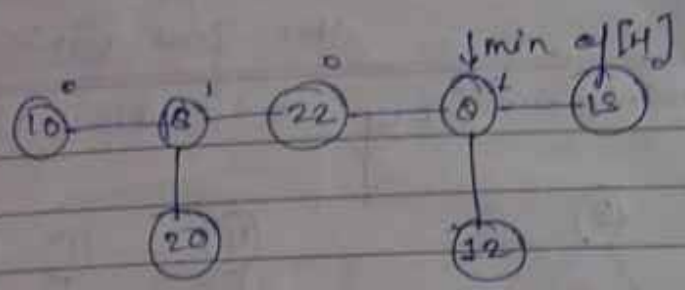
do add x to the root list of H

$P[x] \leftarrow NIL$

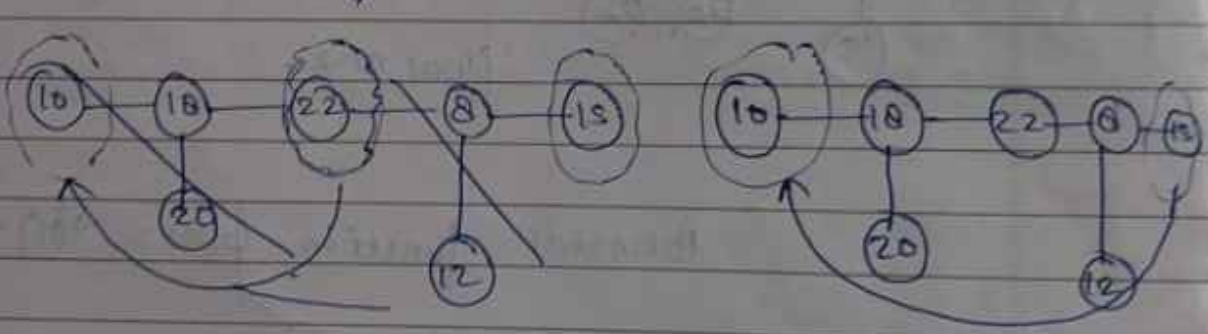
Since here $z \neq NIL$ the condition given is satisfy so each child x of z is added to the root list of H & set

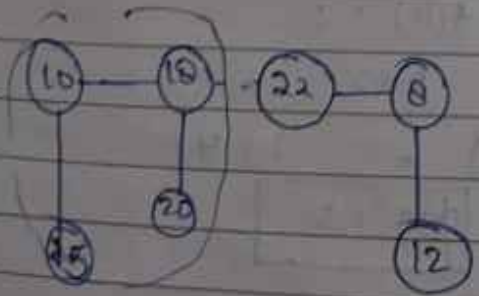


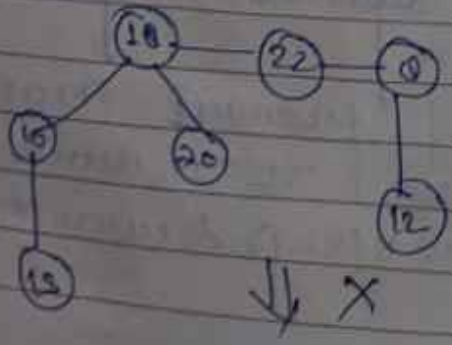
Now remove z from the root list



Step 2 — for consolidate implementing H procedure maintaining a array which contain a degree of tree in this stage temporary root node point to corresponding position.

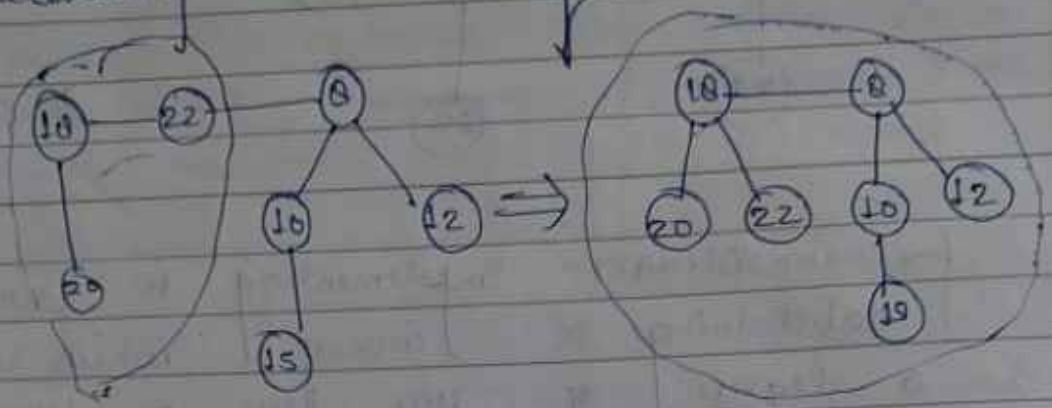


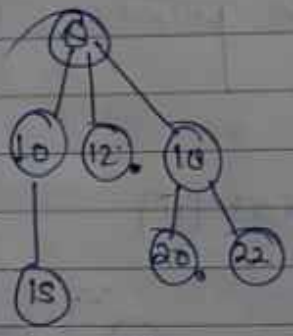




when degree is not same then find minimum node.

According minimum key add the all node.





final tree.

Potential function $\phi(H) = t(H) + 2m[H]$

$t(H) = 1$

2×2

$\phi(H) = 1 + 4$

$\boxed{\phi(H) = 5}$

* Decrease key operation from fibonacci heap.

The procedure fibonacci heap decrease key (H, x, k) is used to decrease key[x] & assign a new k(key) ≤ current key.

FIBONACCI HEAP DECREASE KEY [H, X, K]

Algorithm

1 if K > Key[x]

2. then error new key is greater than current key

3. Key[x] ← K

4. y ← P[x]

5. if (y to NIL & Key[x] < Key[y])

6. then cut (H, x, y)

7. CASCADING CUT (H, y)

8) if Key[x] < Key[min[H]]

9) then min[H] ← H

Now here we given Algo [CUT [H, X, y]

CUT [H, X, K]

1) Remove x from the child list y, decrementing degree[y]

2) add x to the root list [H]

3)

4) P(x) ← NIL

Move[x] ← FALSE

Algo for CASCADING CUT

1) z ← P[y]

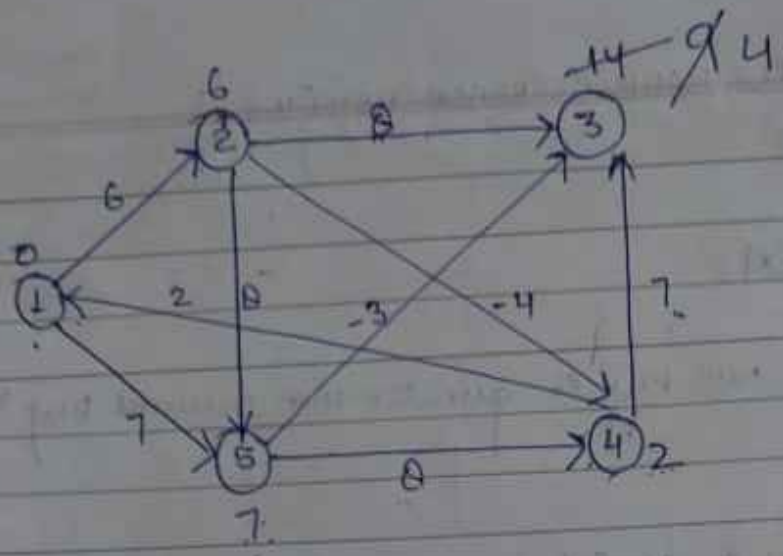
2) if (z ≠ NIL) then

3) if mark (y) = FALSE

4) then mark[y] ← TRUE

5) else CUT (x, y, z)

6) CASCADING CUT [x, z]



1 → 0

2 → 6

3 → 4

4 → 2

5 → 7