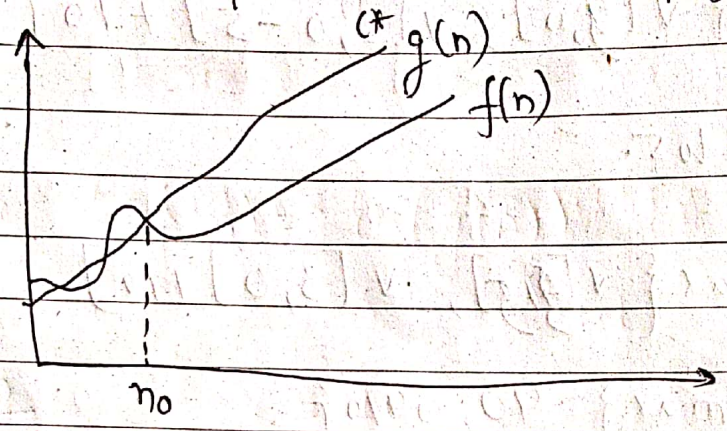
(1) Big-OH (O) : → (upper Bound)

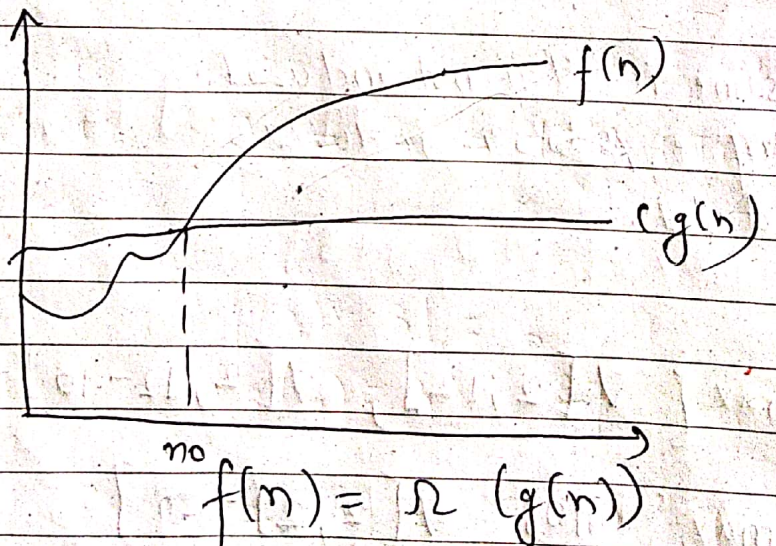
$f(n) = O(g(n))$ says that the growth rate of $f(n)$ is less than or equal to ($\leq$) that of $g(n)$



$f(n) = O(g(n))$

(2) Big-Omega (Ω) (lower Bound) :

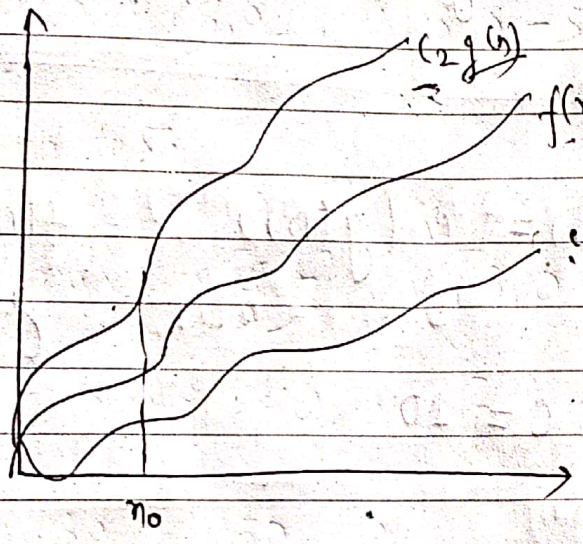
$f(n) = \Omega(g(n))$ greater or equal than $g(n)$



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

(3) Big Theta : (Same order):

$f(n) = \Theta(g(n))$ says that the growth rate of $f(n)$ equals ($=$) the growth rate of $g(n)$. [if $f(n) = O(g(n))$ and $T(n) = \Omega(g(n))$]



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

(Q) Prove: $(n!) \equiv O(n^n)$

$f(n) = n!$

$g(n) = n^n$

To Prove: $f(n) = O(g(n))$

let us choose c.

$0 \leq f(n) <= c * g(n)$, $n >= n_0$

Big O: $0 \leq f(n) \leq c * g(n) \quad n \geq n_0$

Big Ω: $0 \leq c * g(n) \leq f(n) \quad n \geq n_0$

Big θ: $c_1 * g(n) <= f(n) <= c_2 * g(n) \quad n \geq$

To prove: $f(n) = O(g(n))$ $\quad f(n) = n!$, $g(n) = n^n$

let $n_0 = 2$ $\quad c = 10$

Let; $n = 2$ $\quad f(2) = 2! = 2$

$g(2) = 2^2 = 4$

Check: $0 \leq 2 \leq 4 * 10$ (True)

$n = 3$ $\quad f(3) = 3! = 6$

$g(3) = 3^3 = 27$

$0 \leq 6 \leq 27 * 10$ (True)

Randomized Algorithms:

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called randomized algorithm.

For example; in randomized quick sort, we use random numbers to pick the next pivot. Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

Two main types of randomized algorithms:

(i) Las Vegas algorithms
(ii) Monte - Carlo algorithms

Las Vegas Algorithms: Eg: Randomized Quick Sort

→ Algorithm may use randomness to speed up the computation, but algorithm must always return correct answer to input.

Monte Carlo Algorithms: Eg: finding ~~radius~~ value of π

→ They are allowed to give wrong return values.

However, returning a wrong return value must have a small probability, otherwise that Monte-Carlo algorithm would not be of any use.

Step Table Method:

We calculate the cost of executing each step.

```
int sum ( int a[], int n) {
    int sum = 0;                          ——— line ①
    for (int i = 0; i < n; i++) {         ——— ②
        sum += a[i];                      ——— ③
    }
    return sum;                           ——— ④
}
```

COST	Freq	Total Cost
c_1	1	$c_1 * 1 = c_1$
c_2	$n+1$	$c_2 * (n+1) = c_2(n+1)$
c_3	n	$c_3 * n$
c_4	1	c_4

$$T(n) = c_1 + (n+1)c_2 + c_3 * n + c_4$$

$$= c_2 n + c_3 n + c_1 + c_4 + c_2 \Rightarrow T(n) = (\ldots)$$

(Q) ~~float sum~~

~~Recurrence~~

Recurrence Relation:

A recurrence is an inequality that describes a problem in terms of itself

For example:

Recursive algorithm for finding factorial:

$$T(n) = 1 \quad \text{when } n = 1$$
$$T(n) = T(n-1) + O(1) \quad \text{when } n > 1.$$

Technique for solving recurrences:

(1) Substitution Method
(2) Iterative Method
(3) Recursion Tree Method
(4) Master Method

(1) Substitution Method:

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

(i) Guess the solution

(ii) Use mathematical induction to solve boundary condition and show the guess are correct.

$$T(n) = 2T(n/2) + n \quad \text{——(1)} \qquad T(1) = 1 \quad (i=1)$$

Replace $n \rightarrow n/2$, $i = 1$

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

So eqn. (1) becomes.

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

$$T(n) = 4T\left(\frac{n}{4}\right) + 2n \quad \text{——(3)} \qquad (i=2)$$

$n \longrightarrow n/4$

$$T(n/4) = 2T(n/8) + n/4 \quad \text{——— (4)}$$

Putting (4) in (3):

$$T(n) = 4 * \left[2T(n/8) + n/4 \right] + 2n$$

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

$$= 8T(n/8) + 3n \quad \text{——— (5)} \quad (i=3)$$

So; General eqn. is:

$$\boxed{T(n) = 2^i T\left(\frac{n}{2^i}\right) + in}$$

let; $\frac{n}{2^i} = 1$

$$n = 2^i$$

$$\log_2 n = i$$

So; new general eqn. becomes;

$$T(n) = n \cdot T(1) + n \log_2 n = n + n \log_2 n.$$

(2) Iterative method:

Unfold recurrence until you see the pattern.

$T(n) = 8\,T(n/2) + n^2$ & $T(1) = 1$

$$T(n) = 8\,T(n/2) + n^2$$

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

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

$$= 8^2\left[8 * T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2\right] + 2n^2 + n^2$$

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

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

$$= 2^0 n^2 + 2^1 n^2 + 2^2 n^2 + \cdots + 2^{\log_2 n - 1}\, n^2 + 8^{\log_2 n}\,T(1)$$

$$= 2^0 n^2 + 2^1 n^2 + 2^2 n^2 + \cdots + 2^{\log_2 n - 1}\, n^2 + 8^{\log_2 n}$$

Date
Page

General nth term = $\frac{n}{2^i}$

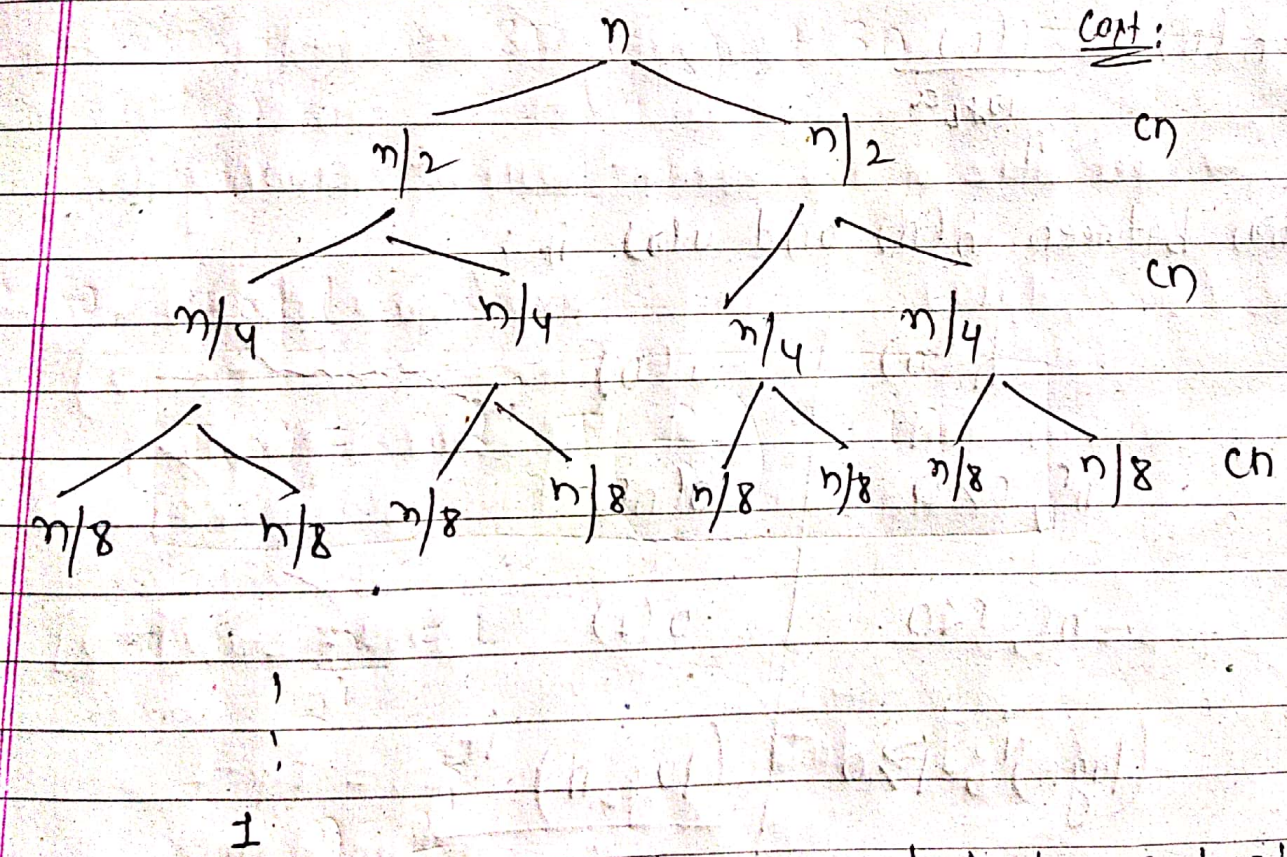
at Base condition; $\frac{n}{2^i} = 1$

$\Rightarrow n = 2^i$

$\Rightarrow \log_2 n = i$

(3) Recursion Tree:

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



Since, will go up to log n height, cost will be,

$= cn + cn + \dots$ up to $\log n$

$= cn \log n$

$O(n \log n)$

Recursion Tree:

① $T(n) = 2T(n/2) + cn$

→ Height of tree $\log_2 n$

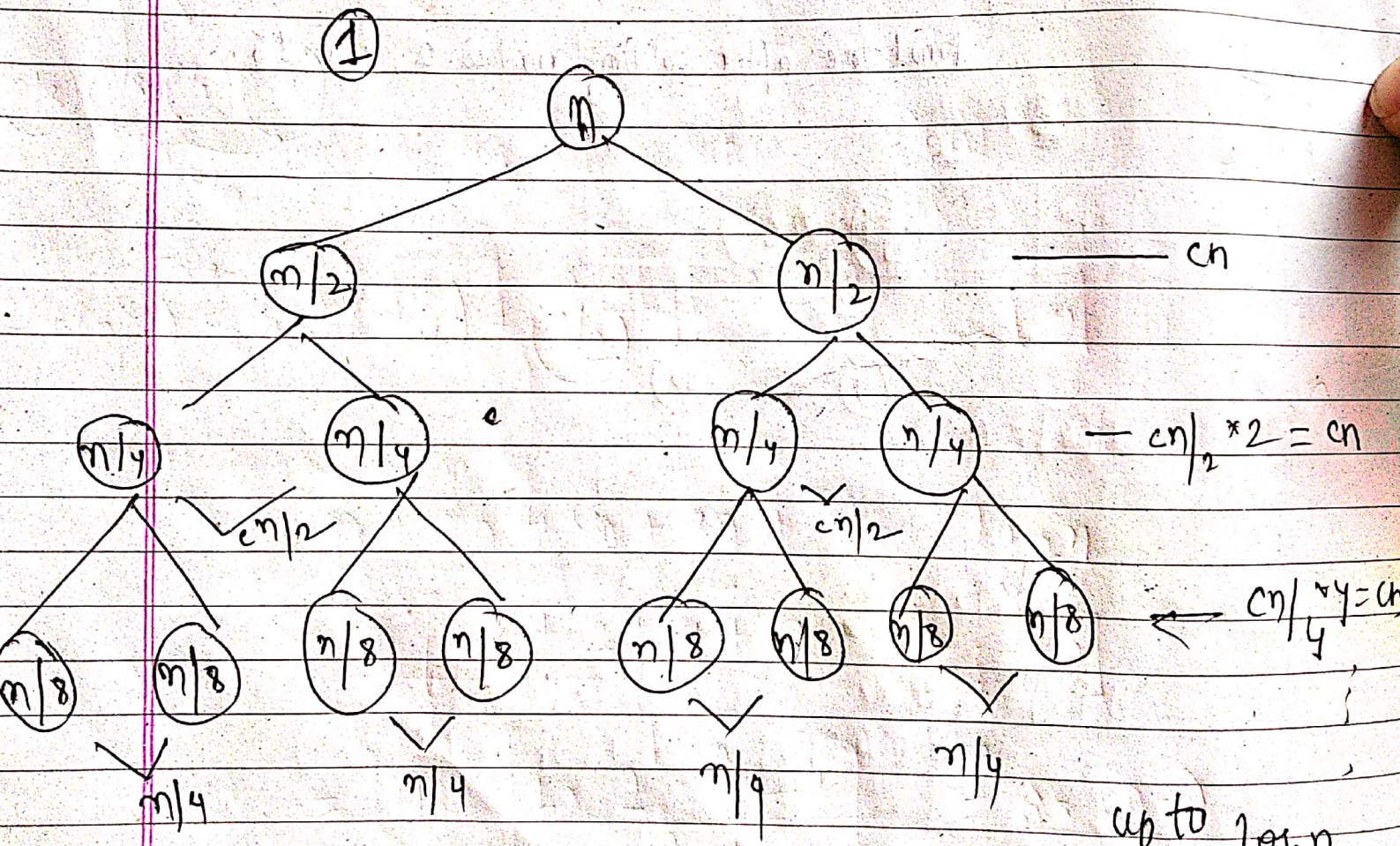
A problem is broken into 2 parts with cost cn

② $T(n) = 3T(n/4) + cn^2$

Height of tree $= \log_4 n$

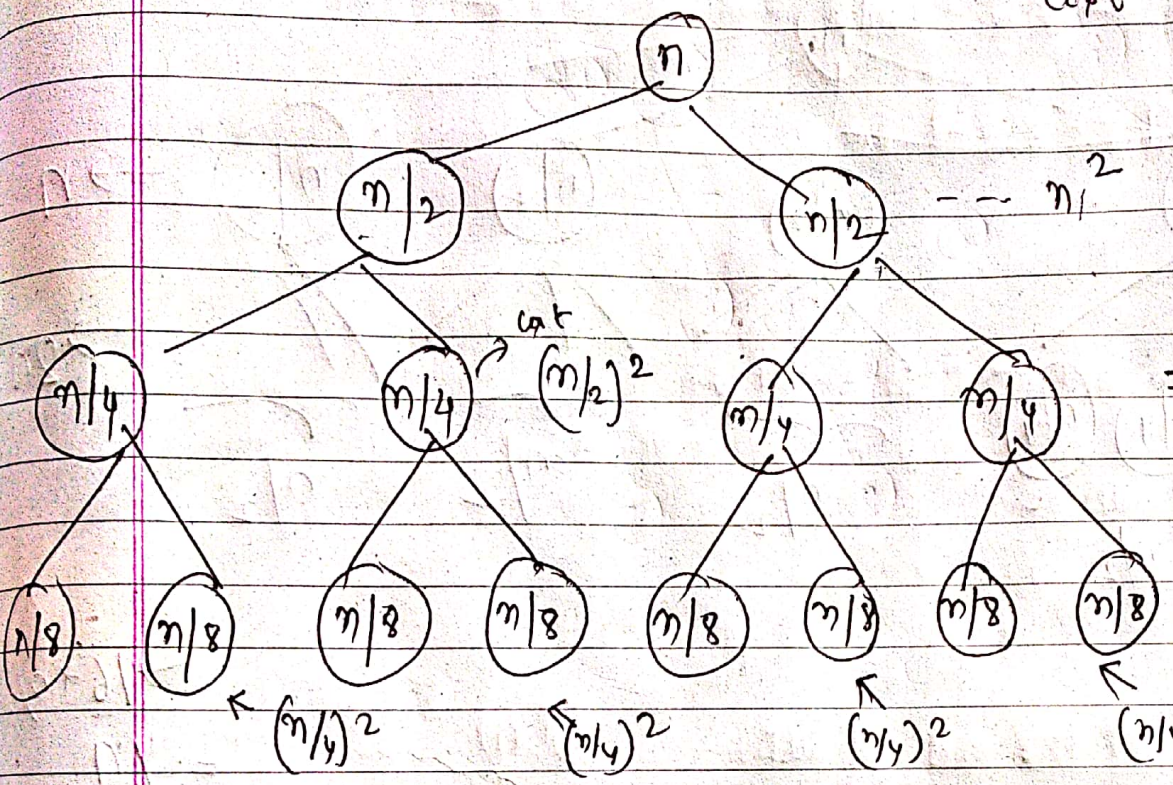
A problem is broken into 3 parts with cost cn^2

①



$= n \log n$

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

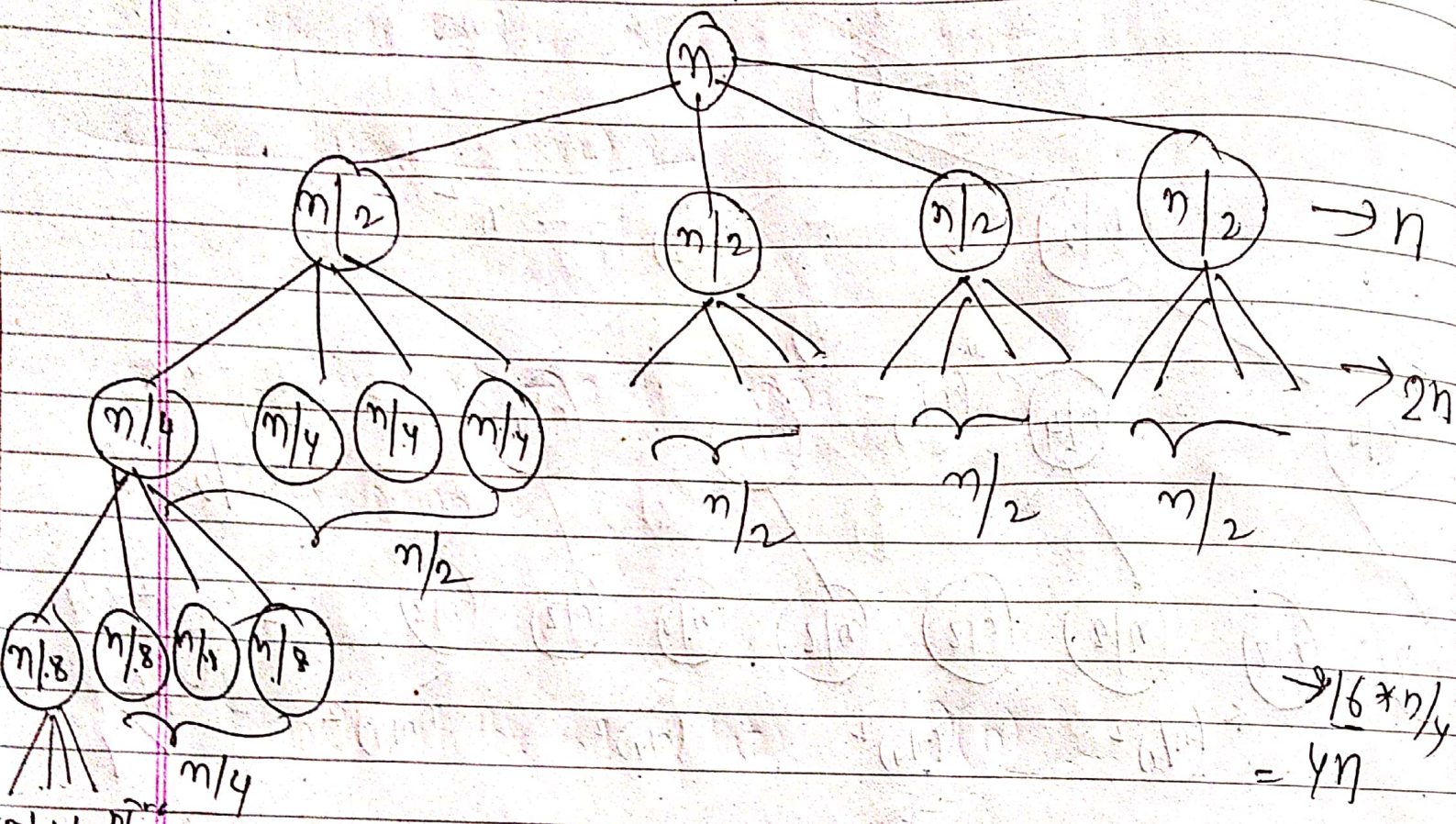


$n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \cdots$ upto $\log_2 n$ times.

$= n^2 \left[1 + 1/2 + 1/2^2 + \cdots \text{upto } \infty \right]$

$= n^2 * \left(\frac{1}{1 - 1/2} \right) = 2n^2.$

③ $T(n) = 4\,T(n/2) + n$



$T(n/2) = 4\,T(n/4) + n/2$

$n + 2n + 4n + 8n + \cdots \log_2 n$

$= n[1 + 2 + 4 + 8 + \cdots \log_2 n]$

$= n\left[\frac{1 * 2^{\log_2 n} - 1}{2-1}\right]$

$= n * (n-1)$

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

$y = 2^{\log_2 n}$

Master Theorem for Dividing Functions

$$T(n) = a\,T(n/b) + f(n)$$

① $\log_b^a$

② k

$a \geq 1$, $b > 1$, $f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b^a > k$ then $\Theta(n^{\log_b^a})$

Case 2: if $\log_b^a = k$

- if $p > -1$ $\Theta(n^k \log^{p+1} n)$
- if $p = -1$ $\Theta(n^k \log\log n)$
- if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b^a < k$

- if $p \geq 0$ $\Theta(n^k \log^p n)$
- if $p < 0$ $\Theta(n^k)$

Chapter 2: Review abstract data types:

(1) Stack:

→ FILO / LIFO:

→ linear data structure.

Basic operations:

(i) Push:
(ii) Pop
(iii) isEmpty
(iv) Isfull
(v) Peek → Get value of top element without removing it.

```
Push (stack, data)

if stack is full
        return null

top = top + 1

stack[top] = data
```

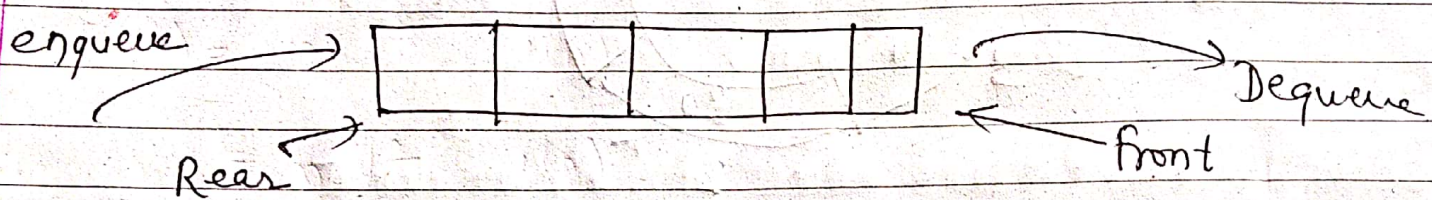
```
Pop (stack, data)

if stack is empty
        return null

data = stack[top]

top = top - 1
return data
```

Push & Pop (→ O(1)).

(2) Queue:

↳ Both ended

↳ linear ; FIFO



isFull():

if (rear == maxsize − 1) return true

isEmpty(): if (front > rear || front == −1)

enqueue():

if queue is full
return overflow

rear = rear + 1
queue[rear] = data

dequeue()

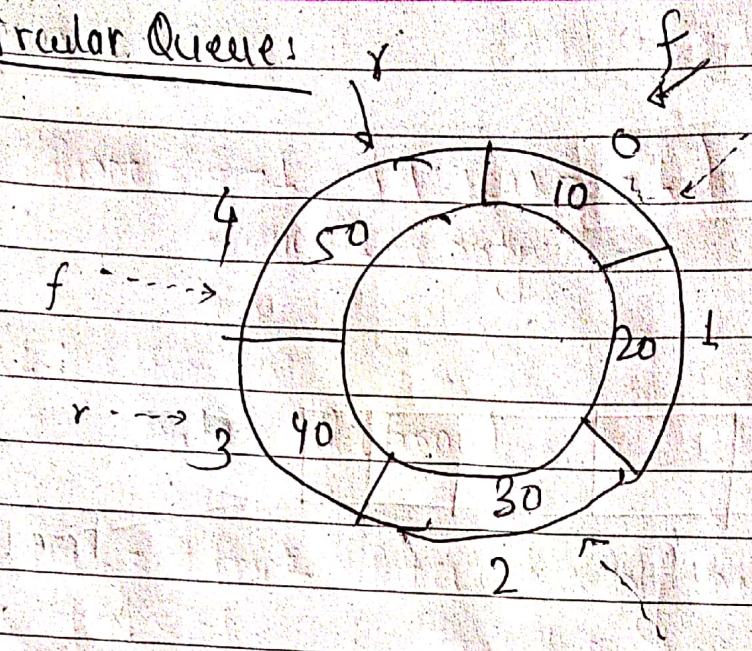
if queue is empty
return underflow

data = queue(front)
front = front + 1

return data.

Enqueue & Dequeue is O(1).

Circular Queue:



Enqueue

In the linear queue, we insert elements until queue becomes full. But once the queue becomes full, we can not insert the next element even if there is a space in front of queue.

isfull():

condition: (rear == size - 1 && front == 0)
|| (rear == front - 1)

isEmpty: front == -1

enqueue (value):

(1) check whether the queue is full.

(2) If it is full then display queue is full.

(3) If queue not full then check if (rear == size-1 && front != 0) if it is true then set rear=0 and insert element.

(1) If (front == 0 && rear == size-1) OR front = rear+1
Display eque is full

(2) If Front == -1 && rear == -1 (Inserting first time)
Front = 0
rear = 0

(3) ~~else~~
if Rear == size-1 // Rear is at last position
Rear = 0

Else
Rear = (Rear+1) % size

(4) Q[Rear] = data.

dequeue:

1. If (front == -1 && rear == -1)
   Queue is empty
2. Else:
   Data = Q[front]
3. If rear == front. Queue has only one element
   front = -1
   rear = -1
4. Else:
   front = (front + 1) % size
5. return data.

Time complexity:

Enqueue & dequeue is O(1) as there is no loop in the algorithm.

(3) Binary Trees:

Binary search Tree:

Binary search tree is a binary tree data structure:

(1) left subtree of node contains nodes with keys lesser than node's key

(2) Right subtree of node contains nodes with keys greater than node's key

(3) left and right subtree each must also be a binary search tree.

1. ~~Insertion~~ Search:

Algorithm:

(1) Compare the key with the value of root node.

(2) If the key is present at root node, return root node

(3) If key is greater than root node value, then search in right subtree.

(4) If the key is smaller than root node value, then search in left subtree

Complexity: $O(\log_2 n)$

Insertion:

(1) Tree must be searched to determine where the node is to be inserted.

(2) Then node is inserted into the tree.

Step 1:
1. If data < root → data
   Proceed to left child of root
2. If data > root → data
   Proceed to right child of root

Step 2: Repeat step 1 until we meet an empty subtree where we can insert data in place of empty subtree by creating a new node.

step3: Exit

Algorithm:

1. Input Data to be inserted; ROOT node of tree
2. NewNode = Create a new node
3. If (ROOT == NULL) {

ROOT = NewNode
}

4. ELSE {
Temp = Root
while(1) {
If (DATA < Temp->data) {
Temp = temp -> left
}
Else {
Temp = 1

while(1) {
If (Data < Temp -> data) {
If (Temp -> left != NULL) {
Temp = Temp->left
}
elx {
temp -> left = newNode
}
}
else if (Data > Temp -> data) {
if (temp -> right != NULL) {
Temp = Temp -> right;
}else {
temp -> right = newnode
}

Time complexity = $O(h) = O(\log_2 n)$

Deletion:

- First search and locate the node to be deleted. Then any one of the condition arises:

(1) The node to be deleted has no children
(2) The node has exactly one child
(3) The node has two children.

Suppose: (1) Node to be deleted has no children;

Delete the node and place its parent node by NULL.

(2) Node has exactly one child:

(i) Check whether it is right or left child

(ii) If it is right child, then find smallest element from right subtree and replace node to be deleted with it.

(iii) If it is left child, then find largest element from left subtree and replace node to be deleted with it.

(3) Node has two children:

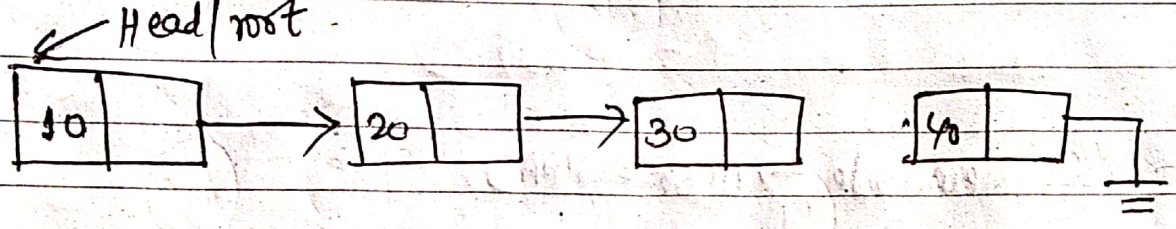
(1) Randomly select a child (left or right)

(2) Similarly as above.

Complexity: $O(h) = O(\log_2 n)$

Linked List:

Linear data structure in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.



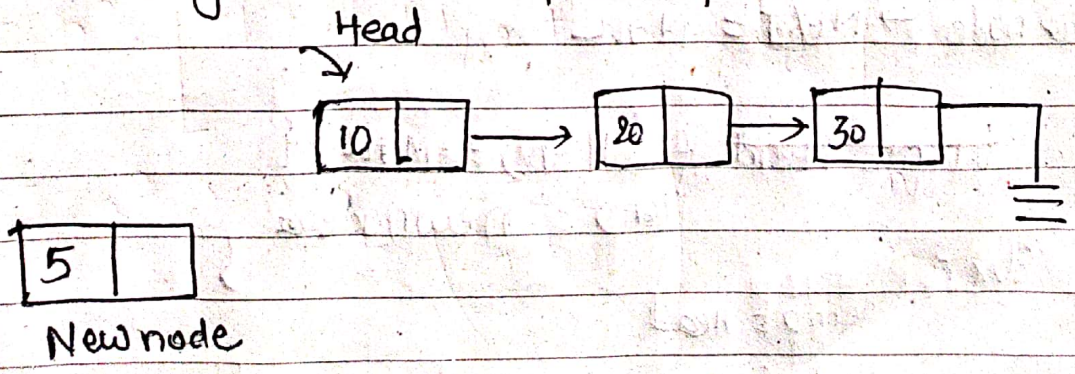
Linked list consists of nodes where each node has data field and a reference/link to next node.

Insertion: O(1)

A new node can be inserted in 3 ways.

1. At front of linked list
2. After a given node
3. At the end of linked list

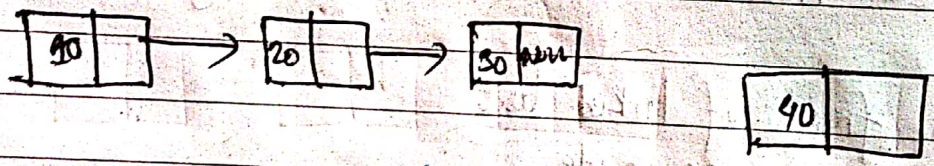
1. Adding node at the front of linked list:



1. Input Data to be inserted
2. Create a newnode.
3. Newnode → Data = Data
4. ~~If (Head == NULL) {~~

5. Newnode → next = HEAD
6. HEAD = Newnode

(B) Newnode at the end of list O(n)



(1) Create a newnode
(2) Newnode → Data = Data
(3) Newnode → NEXT = NULL

(4) ~~temp = Head~~ if (HEAD == NULL) {
HEAD = newnode
}

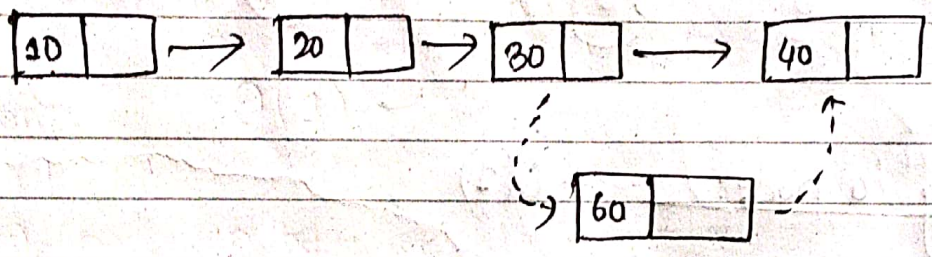
(5) Else {
temp = Head
while (temp → next != NULL) {
temp = temp → next
}

temp → next = NewNode.

(c) Insert node at specified position. O(n)

1. INPUT Data and Pos to be inserted.

2. If (HEAD == NULL) {
   Head = Newnode RETURN
   }



3. i=0 , temp = Head

4. while (i < pos) {
   temp = temp → next
   i = i+1
   }

5. NewNode → next = temp → next

6. temp → next = Newnode

(B) Displaying:

(1) If Head == NULL: Empty Exit

(2) temp = Head

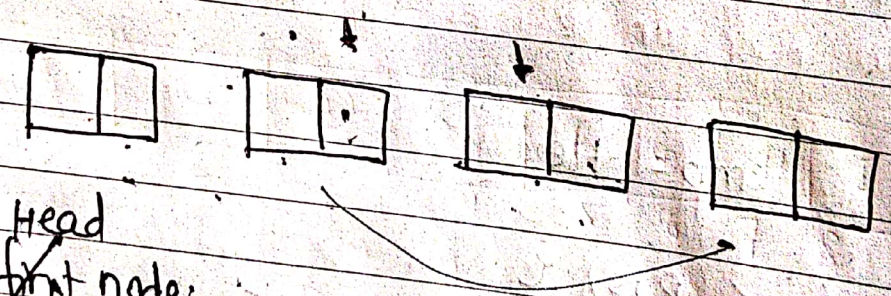
(3) while (temp != NULL) {

display temp → data

temp → temp → next

}

(C) Deleting a node: O(1)



Deleting ~~first~~ node:

1. Temp = Head
2. Head = Head → next
3. Free temp

Deleting Last node: O(n)

1. Temp = Head

2. while(temp -> next != NULL){
   temp = temp ->next
   }

3. ~~Del = Deleted = temp~~ Del = temp -> next

4. temp -> next = NULL

5. free Del

n = 3

Deleting nth node: O(n)

1. i = 1 ; temp = Head

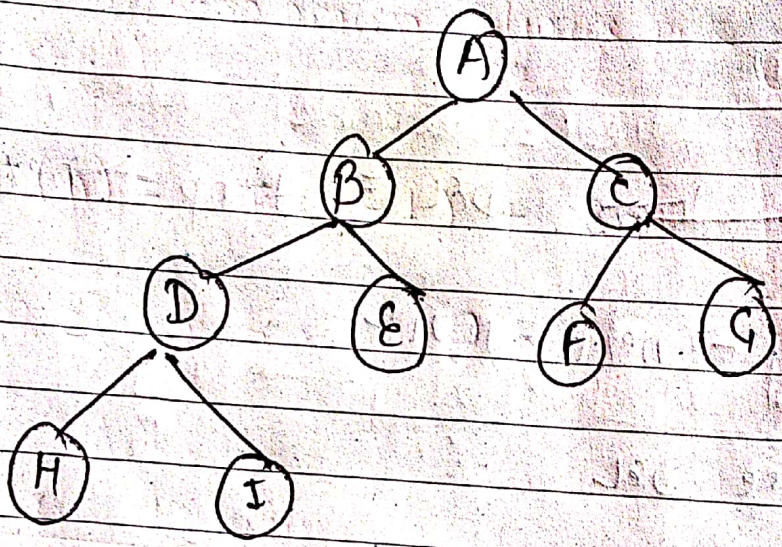
2. while (i < n) {
   temp = temp ->next
   i++;
   }

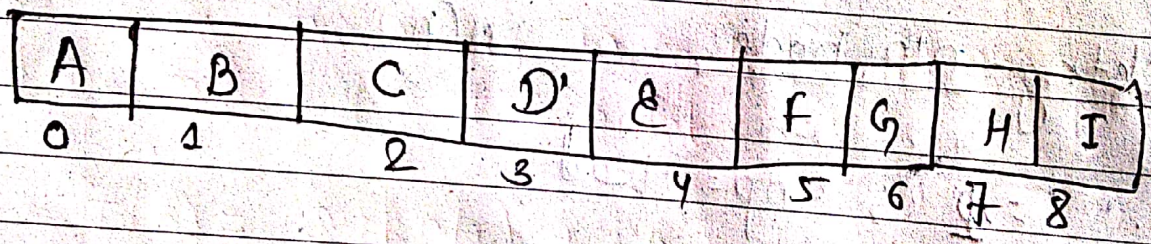
3. temp ->next = temp -> next ->next .

Binary Search Tree (BST) using Array:

→ 1. Initialize an array of size n (n = no. of nodes).

2. Fill each node level wise into the array; in each level fill nodes from left to right.





If a node is at ith index:

→ left child will be at: $(2 * i + 1)$
→ Right child will be at: $(2 * i + 2)$
→ Parent would be: $\left\lfloor \frac{(i-1)}{2} \right\rfloor$

For E, i = 4

Right child of E = 2 * 4 + 2 = 10 (no child)
Left child = 2 * 4 + 1 = 9 (NO)
Parent = $\left[\frac{4-1}{2}\right] = \left[\frac{3}{2}\right] = 1$ (B).

Amortized Analysis of Algorithm

Set, Priority Queue, Dictionary.

Unit -3 Sorting, Selecting and Sequencing

Min-max problem:

(1) Iterative Method
(2) Divide and Conquer Method

Divide and conquer Method:

Algorithm:

minmax (L, r):

```
if (L == r) {
        max = min = A[L];
}
else if (L == r-1) {
        if (A[L] < A[r]) {
                max = A[r];
                min = A[L]
        }
        else {
                max = A[L];
                min = A[r];
        }
}
else {  mid = (L+R)/2      // Divide the problem
```

{min, max} = Minmax(l, mid)

{min1, max1} = minmax(mid+1, r)

If (max1 > max) max = max1

If (min1 < min) min = min1

}

}

Analysis:

The number of steps in Divide and Conquer Method are less than the iterative method to find the minimum and maximum value but we can represent worst case complexity as O(n).

Solution of Recurrence Equation:

Methods to solve recurrence eqn.

1. Substitution Method
2. Iterative Method
3. Recursion Tree Method
4. Master method.

Chapter-3 Sorting, Selection, Sequencing

Sorting:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Heap Sort

1. Bubble Sort:

→ Compares adjacent elements
→ If first is greater than second, swap them.
→ At end of one pass, greatest element finds its position.

```
for (pass = 0; pass < len-1; pass++) {
    for (j=0; j< len - pass - 1; j++) {
        if (arr[j] > arr[j+1]) {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
}
```

Time complexity: $O(n^2)$
~~Space complexity: $O(n)$~~

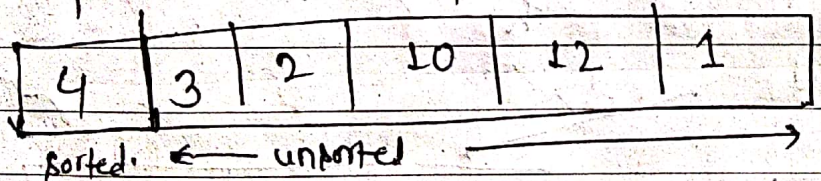
(2) Selection Sort:

→ Selection sort algorithms sorts array by repeatedly finding the ~~maximum~~ minimum element from unsorted part and putting it at the beginning.

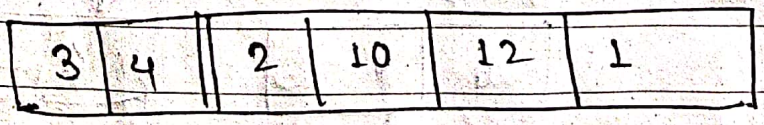
The algorithm maintains two sub-arrays in a given array.
time: $O(n^2)$ ~~space: $O(n)$~~

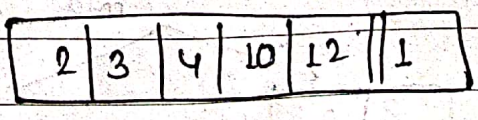
(3) Insertion Sort:

Values from unsorted part are picked and placed at correct position in sorted part.

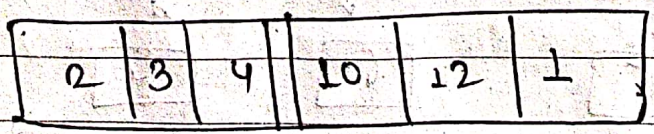


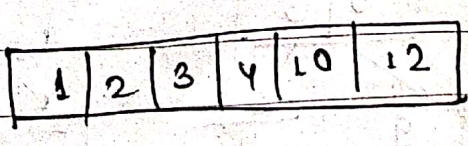
Pick 3: and place at correct position in sorted part.

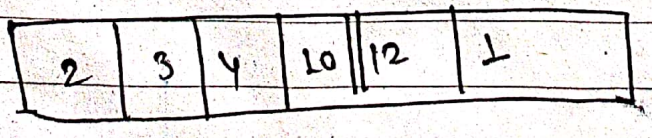




Pick 2:





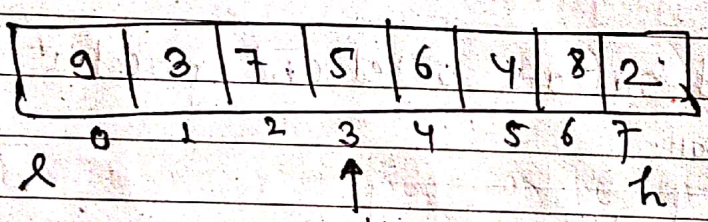


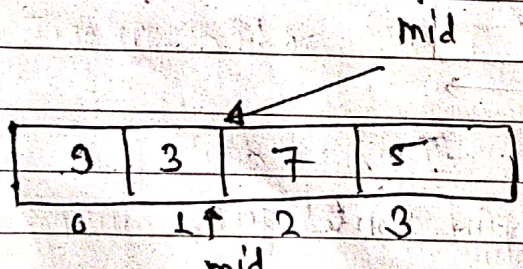
Time: $O(n^2)$
~~Space: $O(n)$~~

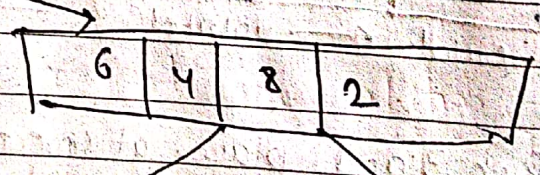
(4) Merge sort:

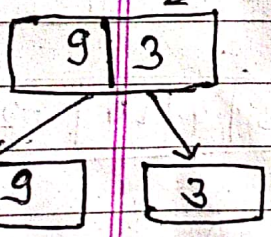
Divide & conquer

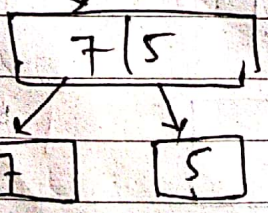
```
mergesort (l, h)
  if (l < h) {
      mid = (l+h)/2 ;
      mergesort (l, mid) ;
      mergesort (mid+h) ;
       merge (l, mid, h)
    }
}
```

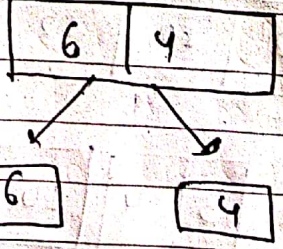


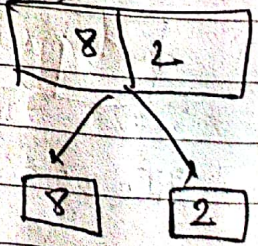


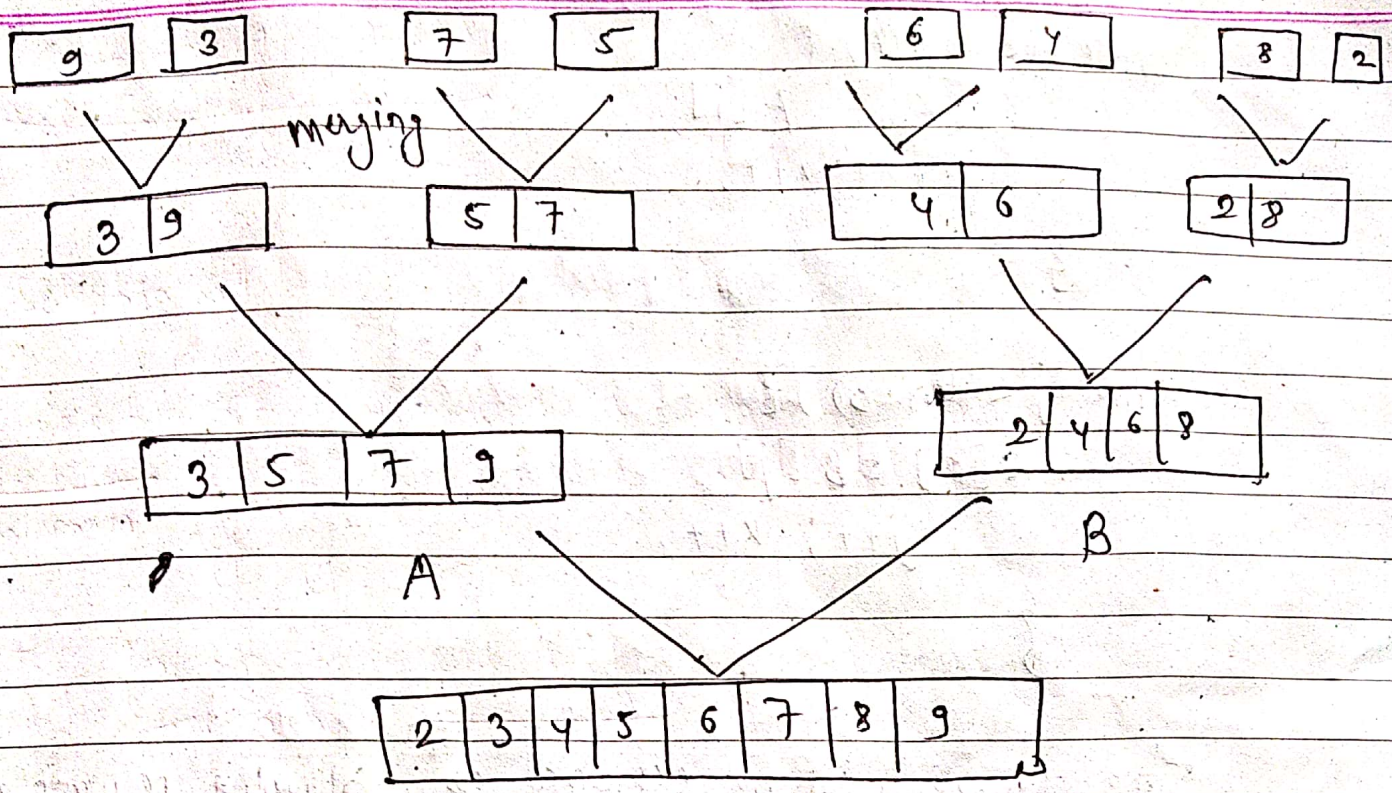












Algorithm for merging two subarrays A & B into C: [l, mid, h]

```
int i=0, j=0, k=l
while ( i < len(A) && j < len(B) ) {
    if ( A[i] <= B[j]) {
        C[k] = A[i];
        k++;
        i++;
    } else {
        C[k] = B[j];
        j++;
        k++;
    }
//copy remaining elements if any
```

```
while (i < len(A) {
        C [k] = A[i];
            i++; k++;
    }

while (j < len(B)) {
        C[k] = B [j];
            j++; k++;
    }
```

Time complexity can be expressed as following recurrence relation:

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

The above recurrence can be solved using Recurrence Tree method or Master method.

It falls in case II of master method and solution of recurrence is $\theta(n \log n)$.

So; Time complexity of merge sort is $\theta(n \log n)$ in all 3 cases (worst, average, best) as merge sort always divide the array into two halves and takes linear time to merge two halves.

Drawbacks:

↳ Slower comparative to other sort algorithm for smaller tasks.

↳ merge sort algorithm requires an additional memory space of O(n) for temporary array.

↳ It goes through whole process even if the array is already sorted.

(5) Quick Sort: [Divide & conquer]

An element is in its sorted position if all elements on left side are smaller and elements on right side are greater. Process is partition.

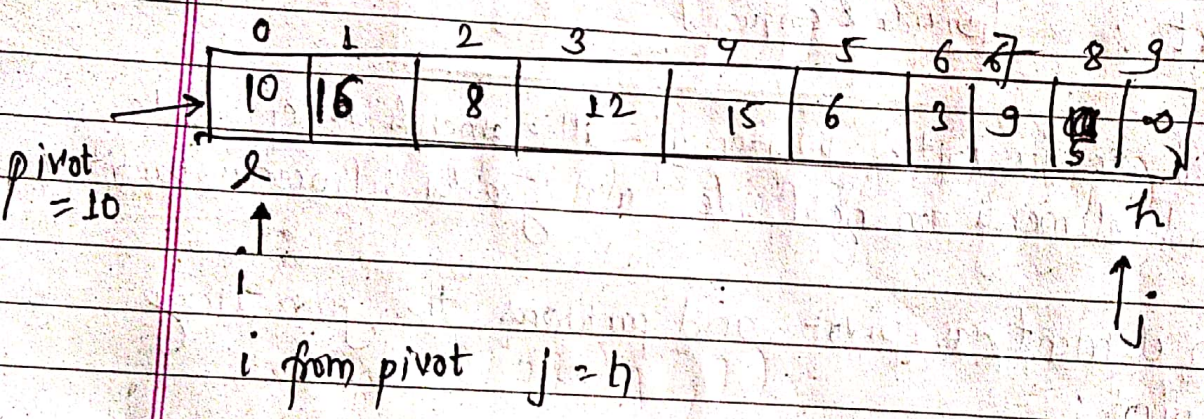
It picks an element as pivot and partitions the given array around the pivot.

Different ways of picking pivot: (1) First element
(2) last element
(3) Random element
(4) Median element.

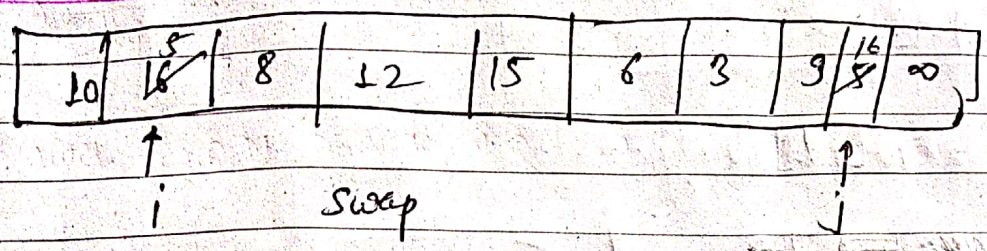
The key process of quick sort is partition.

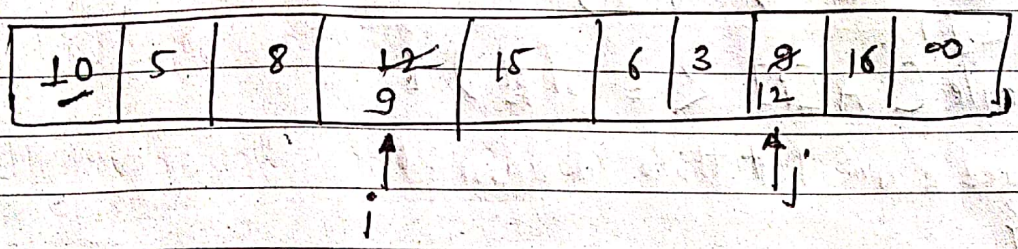
Partition: Given an array and an element x as pivot, put x in its sorted position.

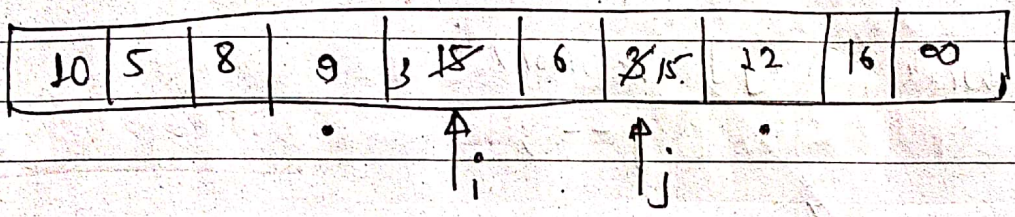
```
Quicksort ( arr, l, h) {
    if ( l < h) {
        p = partition (a, l, h)
        Quicksort ( arr, l, p-1)
        Quicksort (arr, p+1, h)
    }
}
```

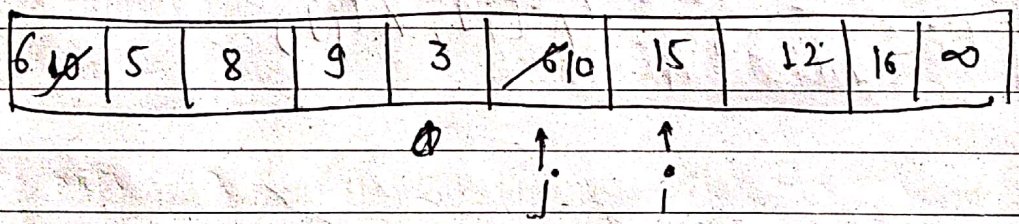


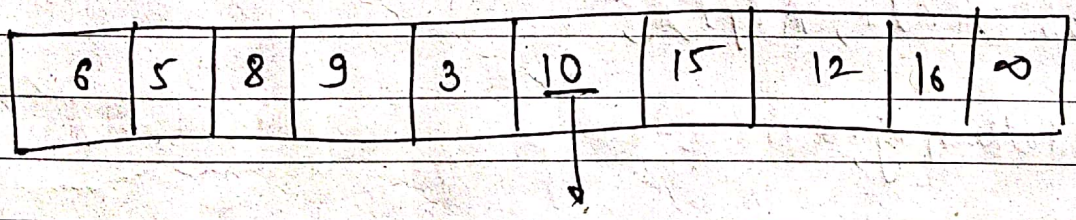
(i) increase i until element greater than pivot is found.
(ii) decrease j until element smaller than or equal to pivot is found
(iii) If i < j, swap arr[i] & arr[j]
(iv) If i > j, position of pivot is j.











10 is in sorted place.

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

Time taken by Quicksort is;

$$T(n) = T(K) + T(n-K-1) + \theta(n).$$

↳ First two terms are for two recursive calls.

↳ Last term is for partition process.

K is the no. of elements smaller than pivot.

Quick Sort Analysis:

(A) Worst Case Analysis:

The pivot is the smallest/greatest element, all the time.

When partition process always pick greatest/smallest element as pivot.

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

whose solution is $\theta(n^2)$.

(B) Best Case Analysis:

The best case occurs when the partition process always picks the middle element as pivot.

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

Solving above equation, we get $O(n \log n)$

ⓒ Average Case:

To do average case analysis, we need to consider all the possible permutation of array and calculate the time taken by every permutation.

We can get an idea of average case by considering the case when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ in another set.

$$T(n) = T(n/9) + T(9n/10) + cn.$$

Solution of above recurrence is $O(n \log n)$

Note:

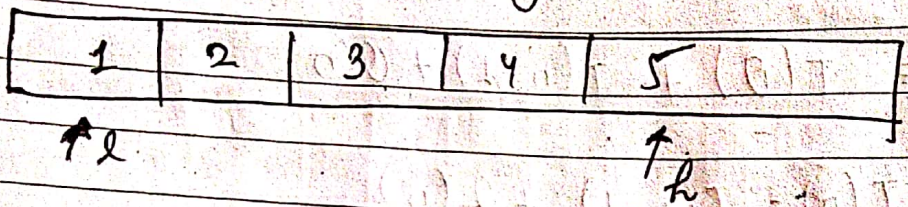
Although worst case time complexity is more than merge/heap sort, quick sort is faster in practice because its inner loop can be be effectively implemented on most architectures, and in most real world data.

Quicksort can be implemented in different ways by changing choice of pivot so that worst case rarely occurs for a given type of data.

HOWEVER, mergesort is generally considered better when data is huge and stored in external storage.

Randomized QuickSort:

Pivot is picked randomly.



↳ To make sure that data is unsorted.

```
RandomizedQS (A, l, h) {
    if (l < h) {
        i = random (l, h)
        swap (A[i], A
```

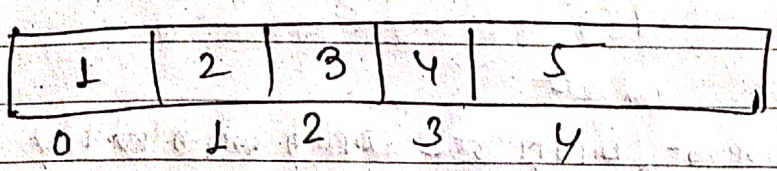
```
RandomizedQS (A, l, h) {
    if (l < h) {
        p = random-partition (A, l, h)
        RandomizedQS (A, l, p-1)
        RandomizedQS (A, p+1, h)
    }
}
```

random-partition (A, l, h) {

r = random number from l to h.
swap (A[r], A[l])

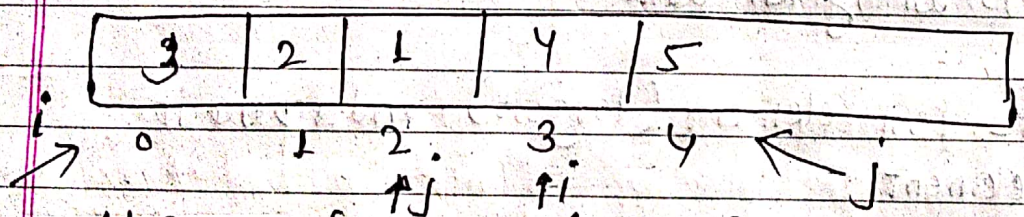
return partition (A, l, h)

↳ normal partition.



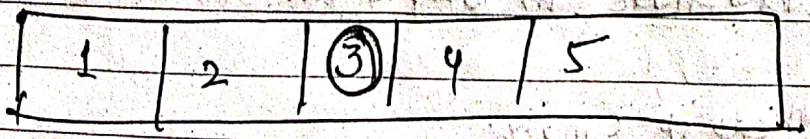
Suppose, random index is 2 and element is 3.

swap (A[2], A[0])



Now perform normal partition

j is the pivot position, swap



(6) Heap Sort:

↳ Heapify O(log n)
↳ Build heap O(n)

So Heapsort: O(n log n)

Algorithm:

Heap is a special type of complete binary tree in which which root is maximum (Max Heap) or minimum (min Heap) than left & right subtree.

(1) Build a heap with given data
(2) Remove the root element (largest) and replace it with last element
(3) Repeapify the complete binary tree.
(4) Place the deleted node in output list
(5) Repeat until heap is empty.

Matrix Multiplication:

Given two square matrices A and B of size n*n each. Find their multiplication matrix.

(I) Naive Method:

```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        C[i][j] = 0;
        for (K=0; K<n; K++) {
            C[i][j] += A[i][K] * B[K][j]
        }
    }
}
```

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

(II) Divide and conquer.

↳ We divide the matrix up to the smallest i.e. 2*2.

↳ [Matrix should be in power of 2; if not make by adding extra zeros]

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

A X B = C

(II) In above method, we do 8 multiplications for matrices of size $N/2 \times N/2$ and 4 additions.

So; $T(n) = 8T(n/2) + \Theta(n^2)$ for addition

From master's theorem: Time complexity is $O(n^3)$. which is same as above naive method.

(III) Strassen's Matrix multiplication.

In above method, main reason for high time complexity is 8 recursive calls. The idea of Strassen's method is to reduce number of recursive calls to 7.

Strassen's method is similar to above method as it also divides matrix into $N/2 * N/2$ but four sub-matrices are calculated using following formulae.

$p1 = a \cdot (f-h)$
$p2 = (a+b) \cdot h$
$p3 = (c+d) \cdot e$
$p4 = d(g-e)$
$p5 = (a+d) \cdot (e+h)$
$p6 = (b-d)(g+h)$
$p7 = (a-c)(e+f)$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p_5+p_4-p_2+p_6 & p_1+p_2 \\ p_3+p_4 & p_1+p_5-p_3-p_7 \end{bmatrix}$$

Time complexity is now given as;

$$T(n) = 7T(n/2) + \Theta(n^2)$$

By Master's theorem; solution is $O(N^{\log 7})$ which is appr. $O(N^{2.8074})$

Generally strassen's method is not preferred for practical applications as:

(1) The constants used in strassen's method are high and for a typical application Naive method works better.

(2) for sparse matrices, there are better methods especially designed for them.

(3) The submatrices in recursion takes extra space.

Date ____
Page ____

Tip to remember equation:

- ahed
- Diagonal
- last column - last Row
- first column - first Row

(ahed, Diagonal) > (P_1, P_2, P_3, P_4)

$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

[Row +, Column −]   from A → Rows, form B → Columns

$P_1 = a * (f - h)$
$P_2 = h * (a + b)$
$P_3 = e * (c + d)$
$P_4 = d * (g - e)$

[Corresponding pos लाई छोड्दिने]

$P_5 = (a + d) * (e + h)$

$P_6 = (b - d)(g + h)$

$P_7 = (a - c)(e + f)$

Scanned with CamScanner

* Job Sequencing with Deadline: [Greedy] $O(n^2)$

Problem:

Given an array of jobs where every job has a deadline & associated profit if job is finished before deadline. It is also given that every job takes a single unit of time, so minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Job	a	b	c	d
Deadline	4	1	1	1
Profit	20	10	40	30

Greedy Algorithm:

1. Sort all jobs in order of profit (descending)

2. Iterate on jobs and for each job:
   (i) Find a time slot i, such that slot is empty and $i <$ deadline and i is the greatest.
   (जति सक्न पछि सार्न सकिन्छ, सार्ने)

   (ii) Put job in this slot and mark this slot filled.

   (iii) If no such slot exists, ignore the job.

Scanned with CamScanner

(2)

Job	a	b	c	d	e
Deadline	2	1	2	1	3
Profit	100	19	27	25	15

Sort:

Job	a	c	d	b	e
Deadline	2	2	1	1	3
Profit	100	27	25	19	15

[Each job is done in unit time]

0 — c — 1 — a — 2 — e — 3

Pick a: a has to be done upto 2nd hour so do this in second slot

Pick c: c has to be done upto 2nd hour, second slot is booked so do this in slot 1.

Pick d: d has to be done in 1st hour, discarded

Pick b: 1st hour discarded

Pick e: upto 3rd hour so in last slot

Value = 60 + 100 + $\frac{20}{30}$ × $\frac{4}{120}$

= 60 + 100 + 80

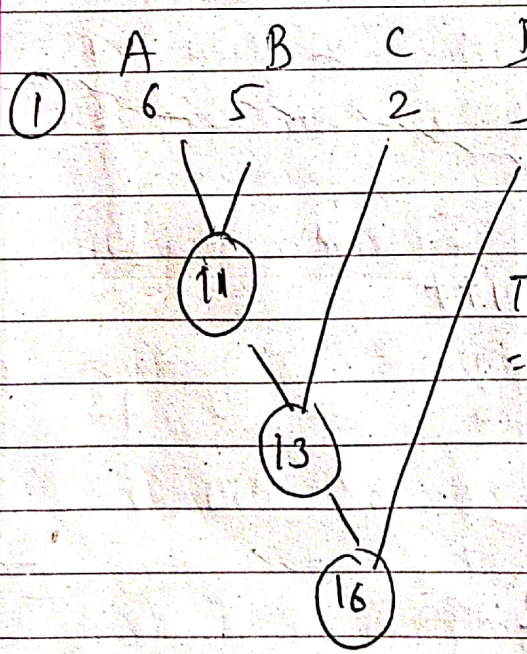
= 240.

Optimal Merge Pattern [Greedy method] O(nlogn)

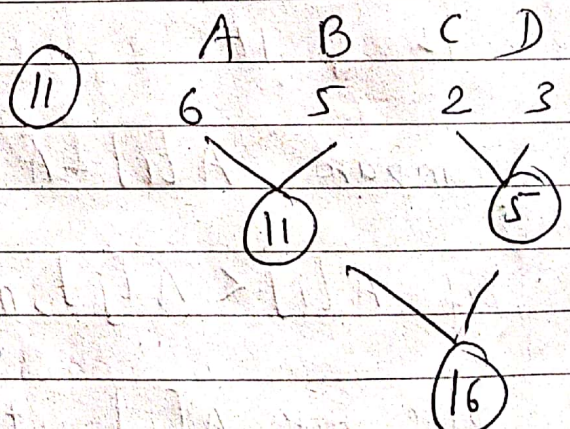
→ While merging we should always merge two small size list to get best result.

List: A B C D

Size: 6 5 2 3



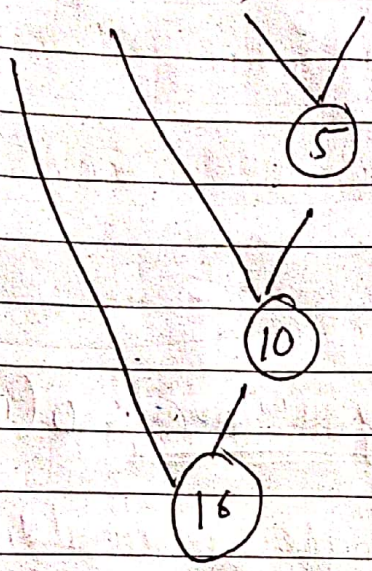
Total merge
= 11 + 13 + 16
= 40



Total merge = 11 + 5 + 16
= 32

(iii)

A	B	C	D
6	5	2	3



Total merge = 16 + 10 + 5
= 31 ✓

Merging two list:

A		B	
i →	3	j →	5
	8		9
	12		11

Compare A[i] & A[j]

if A[i] < A[j], note A[i] & i++

else note A[j], j++

Merged list: 3, 5, 8, 9, 11, 12.

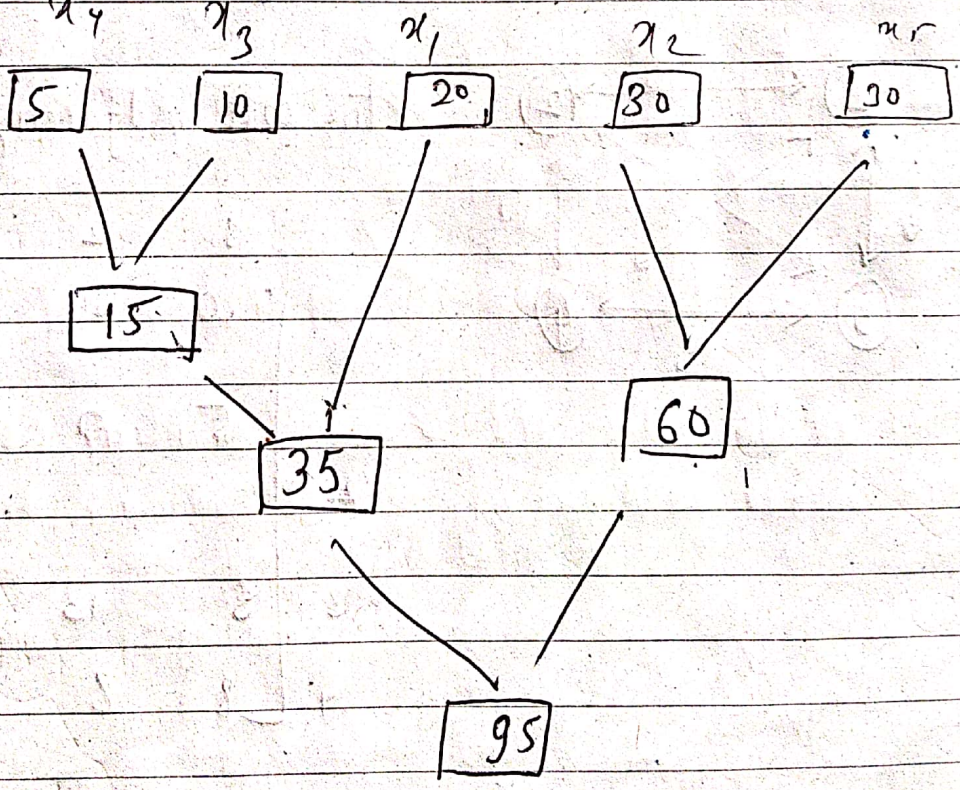
(Q) Find pattern such that merging cost is minimum.

lists:	x_1	x_2	x_3	x_4	x_5
sizes:	20	30	10	5	30

Let's sort:

lists	x_4	x_3	x_1	x_2	x_5
sizes:	5	10	20	30	30

merge two smallest:



Total cost = 15 + 35 + 60 + 95 = 205.

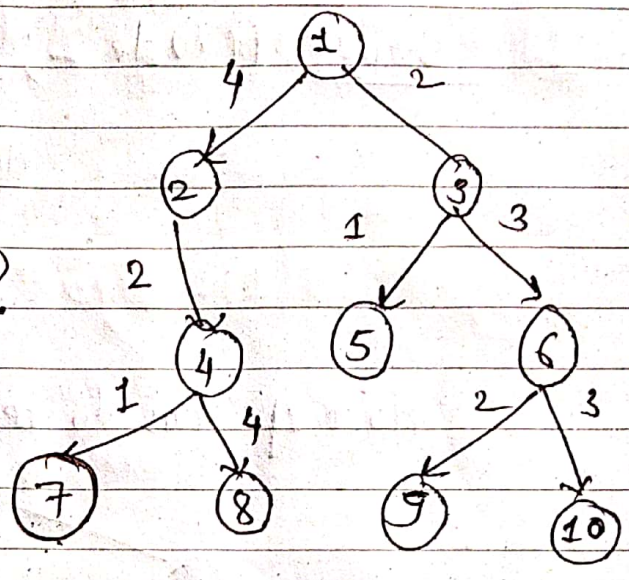
↳ = sum of sizes of internal node

Tree Vertex Splitting:

Any node → u

C(u) → set of children of u.

$$d(u) = \max_{v \in C(u)} \{ d(v) + w(u,v) \}$$



leaf node → u

$d(u) = 0$

If u has a parent v such that

$$d[v] = \boxed{d(u) + w(v,u) > \delta}$$

then u is splitted and d(u) is set to zero.

Step 1:

leaf nodes, $d(u) = 0$ [7, 8, 9, 10, 5]

$d(7) = d[8] = d[9] = d[10] = d[5] = 0.$

Node 4: children of 4 are 7, 8 $c(4) = \{7, 8\}$

$$d[v] = d[4] = \max_{u \in \{7,8\}} \{ d(7) + w(4,7), \; d(8) + w(4,8 \}$$

$$= \{ 0+1, \; 0+4 \} = 4.$$

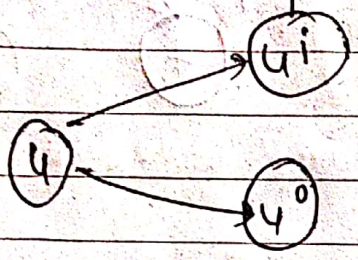
$d[4] = 4 < 5$ (No splitting)

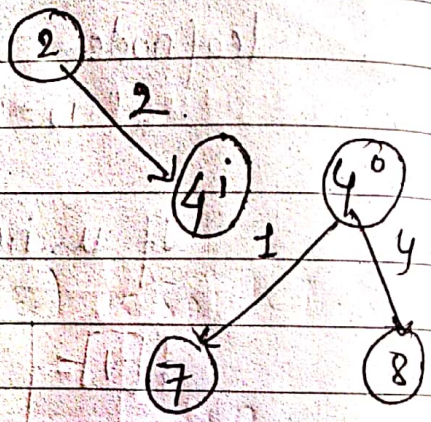
Step 2: For Node 2;

$$d[2] = \min\{d[4] + c(2,4)\}$$

$$= 4 + 2 = 6 \not< 5$$ (Splitting)

Node 4 has to be splitted.





$d[4] = 0$

Step 3: Node 6: $d[6] = \max_{u \in \{9,10\}} \begin{bmatrix} d[9] + c(6,9), \\ d[10] + c(6,10) \end{bmatrix}$

$$= \max\begin{Bmatrix} 0+2, \\ 0+3 \end{Bmatrix} = 3$$

$d[6] = 3 < 5$

Step 4: Node 3:

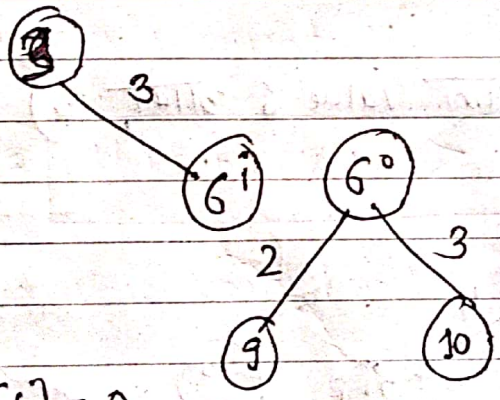
$$d[3] = \max \begin{Bmatrix} d[5] + c(3,5), \\ d[6] + c(3,5) \end{Bmatrix}$$

$$= \max \begin{Bmatrix} 0+1, \\ 3+3 \end{Bmatrix}$$

$$= 6 > 5$$

(child)

So splitting: 6 must be splitted.



$d[6] = 0$

Step 5: Node 1:

$$d[1] = \max_{u \in \{2,3\}} \begin{Bmatrix} d[2] + c[1,2], \\ d[3] + c[1,3] \end{Bmatrix}$$

$d[2]$ has changed as 4 is splitted.

$d[2] = d(4) + \omega(2,4) = 0 + 2 = 2$

$$d[3] = \max_{\{5,6\}} \begin{Bmatrix} d[5] + c(3,5) \\ d[6] + w(3,6) \end{Bmatrix}$$

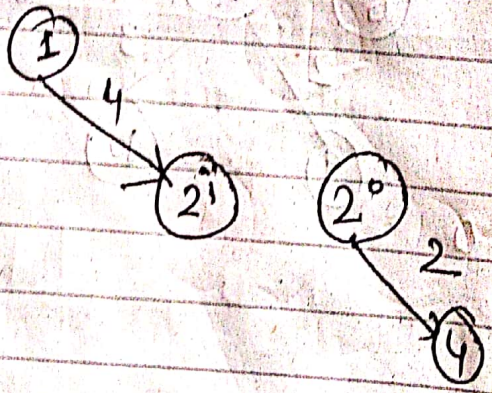
$$= \begin{Bmatrix} 0+1 \\ 0+3 \end{Bmatrix} = 3$$

$$d(1) = \max \begin{Bmatrix} d[2] + c(1,2), \\ d[3] + w(1,3 \end{Bmatrix}$$

$$= \max \begin{Bmatrix} 2+4, \\ 3+2 \end{Bmatrix} \rightarrow$$

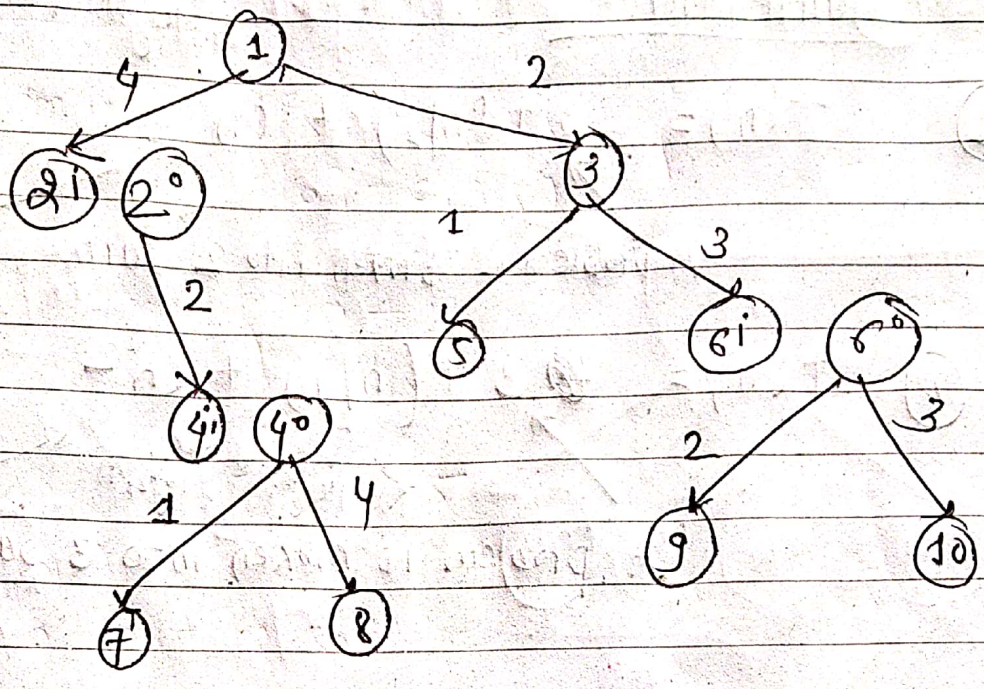
$$= 6, 5$$

So; node 2 (जसको value 6 आको छ) is splitted.



$d(2) = 0.$

Final tree:



Final tree after splitting nodes 2, 4, and 6.

$O(V+E)$ $O(V+E)$

Date
Page

BFS and DFS:

BFS:

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes. You then move towards next level neighbour nodes.

(1) First move horizontally and visit all nodes of current layer.

(2) Move to next layer.

BFS finds nodes level by level. Here we first check all the immediate successors of the start state.

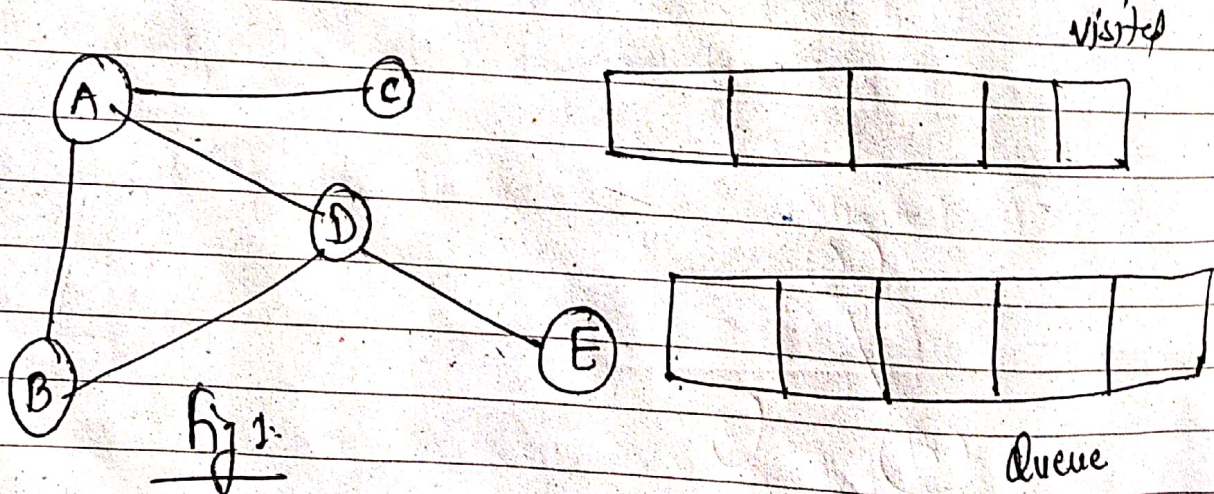
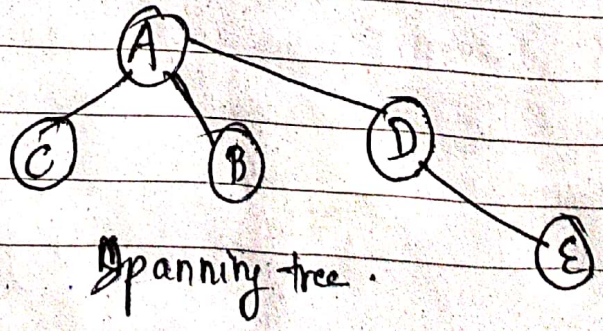


fig 1.



Spanning tree.

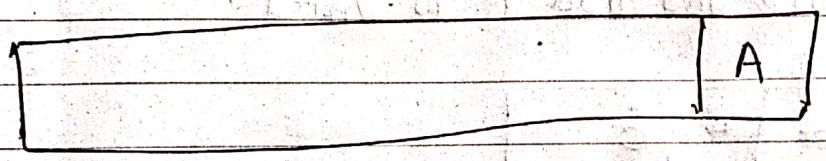
(1) ~~Start~~ By putting any one of the graph's vertices at the back of queue.

(2) Take the front item of the queue and add it to the visited list.

(3) Create a list of vertex's adjacent nodes. Add the ones which are not in the visited list to the back of the queue.

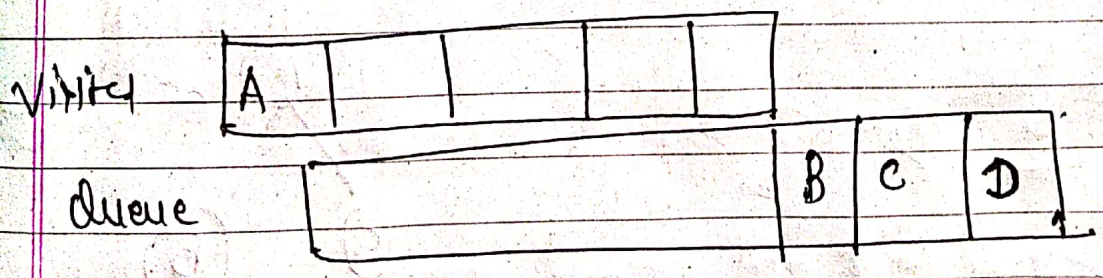
(4) Keep ~~representing~~ repeating steps 2 and 3 until the queue is empty.

Eg:- Let's start from node A. Keep node A in the queue.

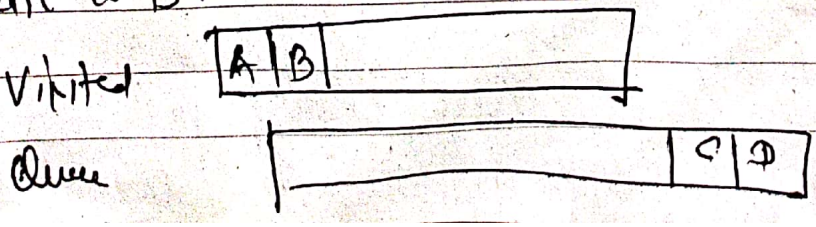


Queue.

Pick A from queue and add it to visited. The ~~adja~~ vertices C, B and D which are adjacent to A are kept in the queue.



Pick B from Queue and add it to visited. The vertices D is adjacent to B.



Pick C and add it to visited.

A	B	C

Pick D from queue and add it to visited. The node E is adjacent to D so add it to Queue.

A	B	C	D

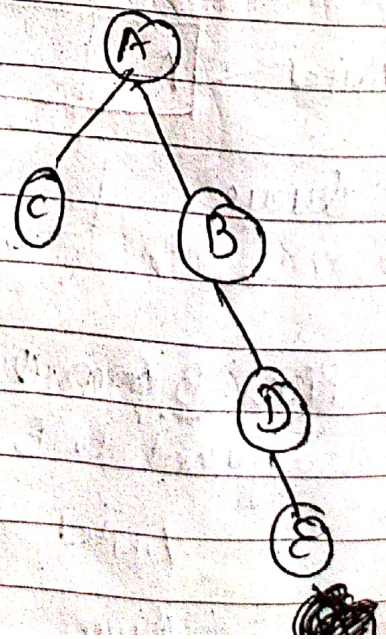
visited

E

Queue

Pick E from queue and add it to visited.

A	B	C	D	E



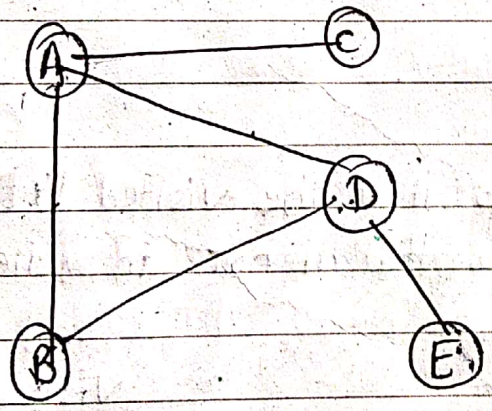
Depth first Search (DFS):

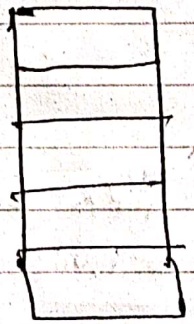
(1) Start by putting anyone of the graph's vertex to top of the stack.

(2) Take the top item from the stack and add it to the visited list.

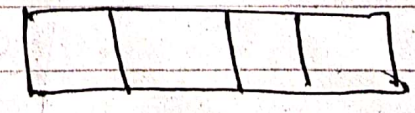
(3) ~~Create a list~~ Add the adjacent nodes which are n't in the visited list on top of stack.

(4) Keep repeating (2) and (3) until stack is empty.

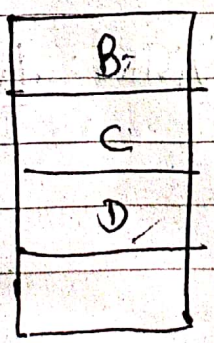




Stack

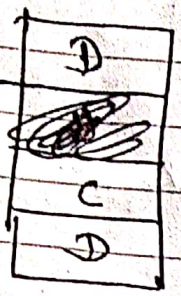
Visited 

Pick A. Put it in the visited list. Nodes adjacent to A ie. B, C, and D are kept in the stack.



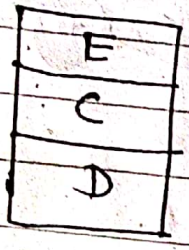
visited : A

Pick B from stack and keep it in the visited list. The nodes adjacent to B i.e. D is kept in the stack.



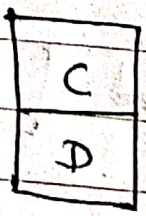
Visited: A, B

Pick D from stack and keep it in the visited list. The node E is adjacent to D. so keep it in the stack.



Visited: A, B, D

Pick E from stack and keep it in the visited list. The node has no adjacents nodes that are not in visited.



Visited: A, B, D, E

Pick C from stack and keep it in visited list. D is already visited.

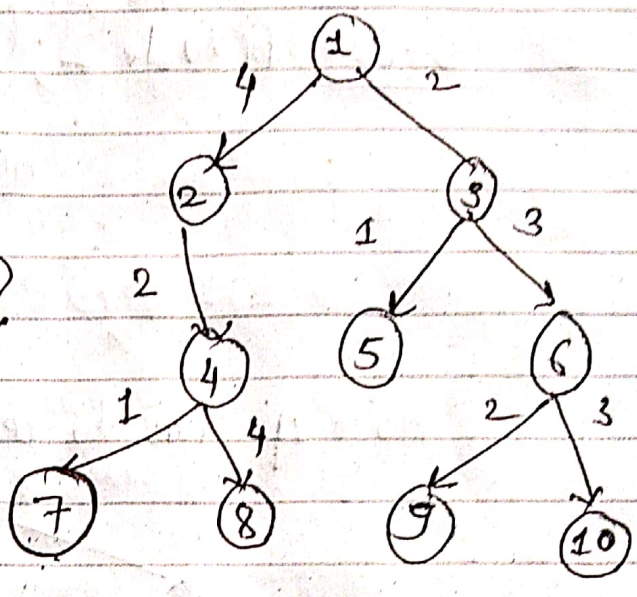
Visited: A, B, D, E, C

Tree Vertex Splitting:

Any node → u

$c(u)$ → set of children of u.

$$d(u) = \max_{v \in c(u)} \{ d(v) + w(u,v) \}$$



leaf node → u

$d(u) = 0$

If u has a parent v such that

$$d[v] = \boxed{d(u) + w(v,u) > \delta}$$

then u is splitted and $d(u)$ is set to zero.

Step 1:

leaf nodes, $d(u) = 0$ $\quad [7, 8, 9, 10, 5]$

$$d(7) = d[8] = d[9] = d[10] = d[5] = 0.$$

Node 4: children of 4 are 7, 8 $\quad c(4) = \{7, 8\}$

$$d[v] = d[4] = \max_{u \in \{7,8\}} \{ d(7) + w(4,7), \; d(8) + w(4,8 \}$$

$$= \{ 0+1, \; 0+4 \} = 4.$$

$d[4] = 4 < 5$ (No splitting)

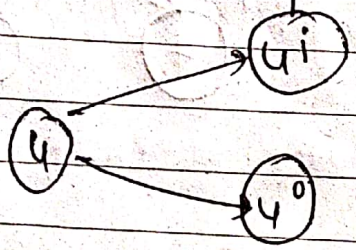
Step 2: For Node 2;

$$d[2] = \min\{d[4] + c(2,4)\}$$

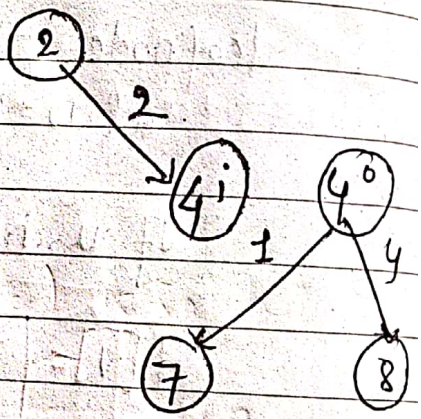
$$= 4 + 2 = 6 \not< 5$$

(Splitting)

Node 4 has to be splitted.



$d[4] = 0$



Step 3: Node 6: $d[6] = \max_{u \in \{9,10\}} \begin{bmatrix} d[9] + c(6,9), \\ d[10] + c(6,10) \end{bmatrix}$

$$= \max\begin{Bmatrix} 0+2, \\ 0+3 \end{Bmatrix} = 3$$

$d[6] = 3 < 5$

Step 4: Node 3:

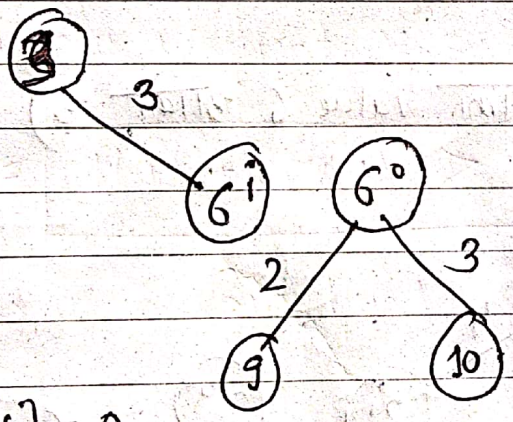
$$d[3] = \max_{u \in \{5,6\}} \{ d[5] + c(3,5), \; d[6] + c(3,5) \}$$

$$= \max \{ 0+1, \; 3+3 \}$$

$$= 6 > 5$$

(child)

So splitting: 6 must be splitted.



$d[6] = 0$

Step 5: Node 1:

$$d[1] = \max_{u \in \{2,3\}} \{ d[2] + c[1,2], \; d[3] + c[1,3] \}$$

$d[2]$ has changed as 4 is splitted.

$d[2] = d(4) + c(2,4) = 0 + 2 = 2$

$$d[3] = \max_{\{5,6\}} \begin{Bmatrix} d[5] + c(3,5) \\ d[6] + w(3,6) \end{Bmatrix}$$

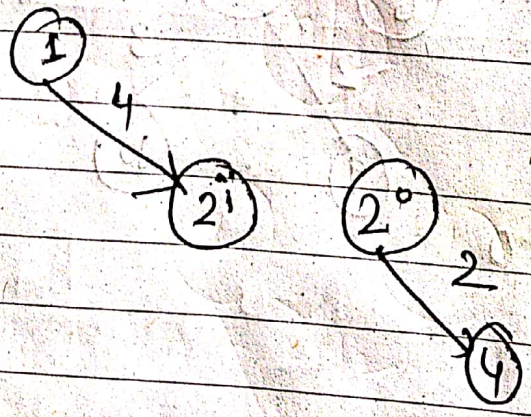
$$= \begin{Bmatrix} 0+1 \\ 0+3 \end{Bmatrix} = 3$$

$$d(1) = \max \begin{Bmatrix} d[2] + w(1,2), \\ d[3] + w(1,3 \end{Bmatrix}$$

$$= \max \begin{Bmatrix} 2+4, \\ 3+2 \end{Bmatrix} \rightarrow$$

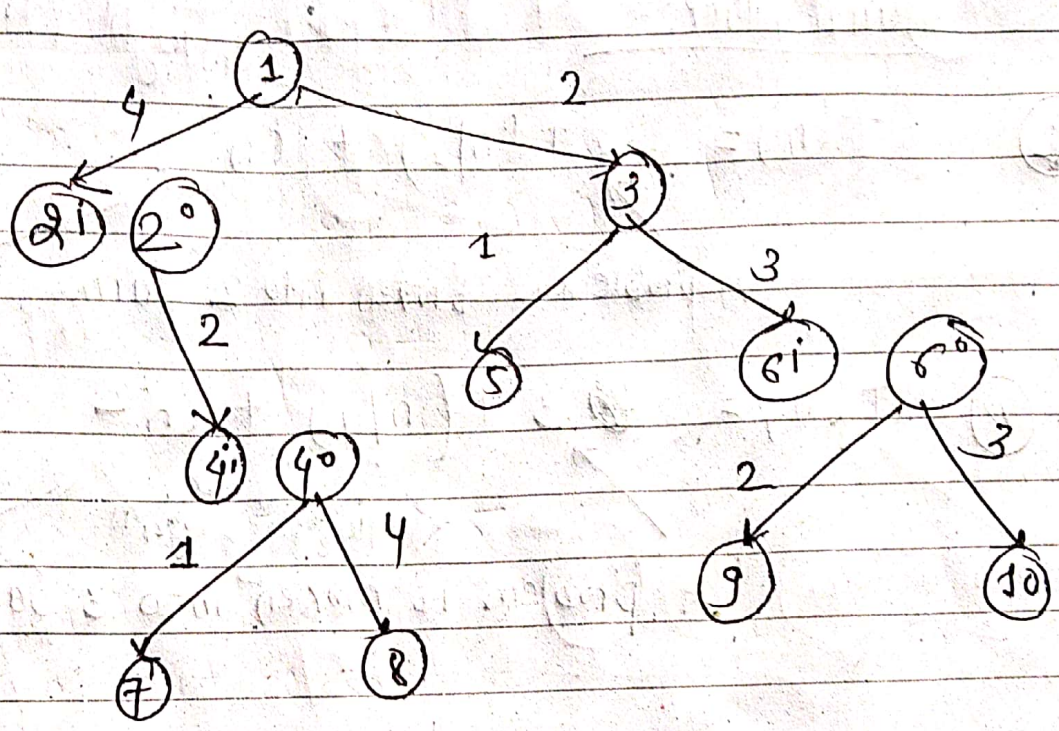
$$= 6 > 5$$

So, node 2 (पहिलो value 6 आको छ) is splitted.



$d(2) = 0.$

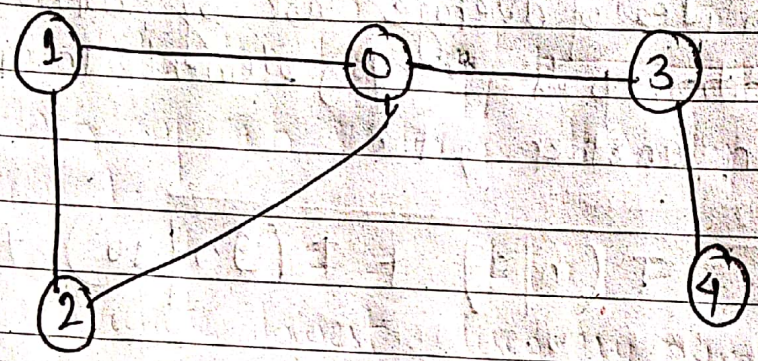
Final tree:



Final tree after splitting nodes 2, 4, and 6.

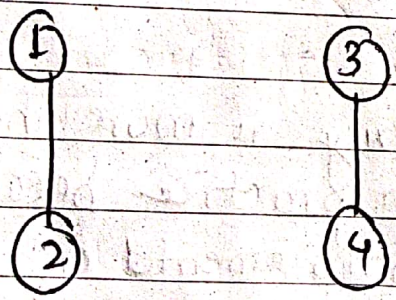
Articulation Point :

A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph | breaks the graph into two or more pieces.

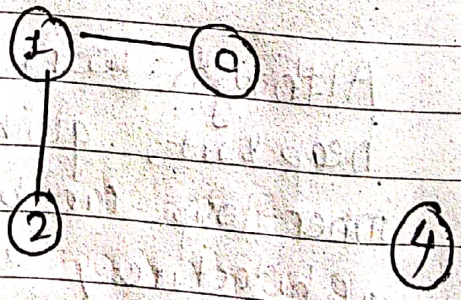


Articulation points are 0 and 3.

Removing 0:



Removing 3:



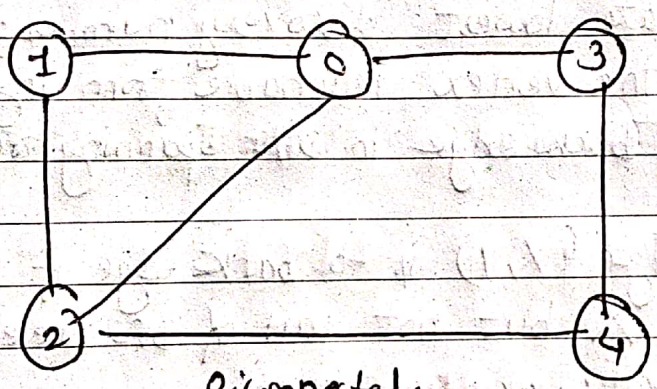
Biconnected graph:

An undirected graph is called biconnected if there are no any articulation points and possible to reach every vertex from every other vertex.

(i) It is possible to reach every vertex from every other vertex

(ii) There are no articulation points (no such vertices removing which breaks the graph into pieces)

Eg:



Biconnected.

[If only 1 condition is fulfilled → Connected Graph]

Algorithm to find articulation point:

~~Determining the arti~~

The articulation point can be found using DFS.

(1) Construct a DFS spanning tree.

(2) Calculate the depth ~~of discovery time~~ of selected vertex. Depth of each vertex is the order in which they are visited.

(3) Calculate lowest discovery number, which is equal to the lowest depth of the vertex reachable from any vertex by considering one back edge in DFS spanning tree.

An edge (X, Y) is a back edge if Y is an ancestor of edge X but not part of DFS tree. But (X, Y) is a part of original graph.

(4) Take a pair of vertices and check whether its an articulation point or not. Let's take two vertices u and v, v is parent and u is child.

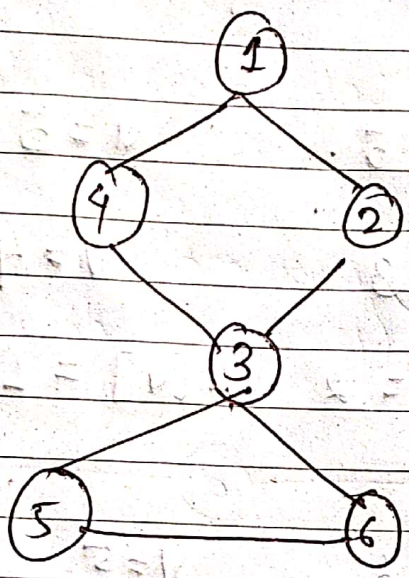
if $L[u] \geq d[v]$, then v is articulation point. (parent)

$L[u]$ = lowest discovery number of u (child)

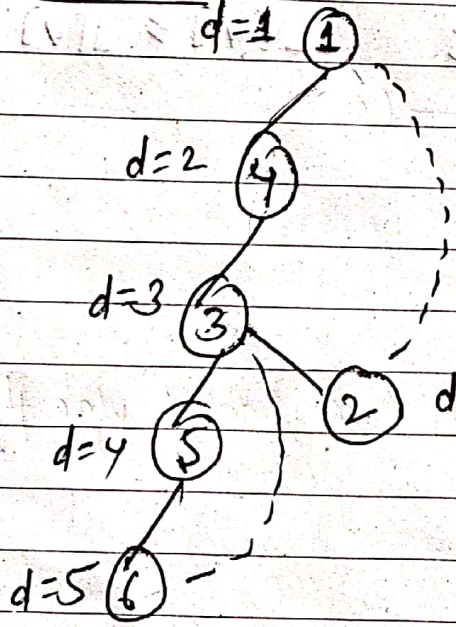
$d[v]$ = depth of v. (parent)

(5) For root to be an articulation point if and only if it has more than one child in the spanning tree.

Example:



Dfs tree:



Vertex	1	2	3	4	5	6
depth	1	6	3	2	4	5
lowest dis num	1	1	1	1	3	3

Consider 4 & 3:

$L[3] = 1 \quad d[4] = 2$

Consider 3 and 5:

$L[5] \geq d[3]$ The $L[5] = 3 \quad d[3] = 3$

So 3 is articulation point.

Vertex 1: Root, has only one child so it is not articulation point

Vertex 2: Consider the child of 2. Vertex 2 is leaf node so no articulation.

Vertex 3: Child is 5. $u = 5$ $V = 3$

Vertex 4: child is 3 $V = 4$ $u = 3$

$d[V] = 2$ $L[u] = 1$ $L[4] \geq d[V]$ (F)

So 4 is not articulation point.

Vertex 5: child is 6. $V = 5$ $4 = 6$

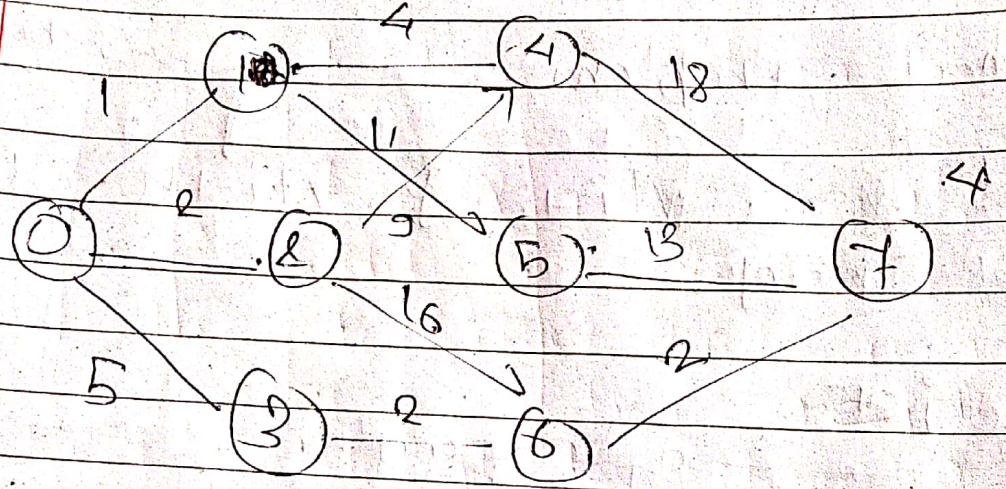
$d[V] = 4$ $L[u] = 3$ $L[4] \geq d[V]$ (false)

Vertex 6: Leaf. no articulation

PAGE NO.:
DATE: /

Multistage Graph.

$v[4, 8] = \max \{$



$$cost(i, s) = \min_{j \in \text{adj. vertices of } i} \{ c(i, j) + c(j, s+1) \}$$

M

All.

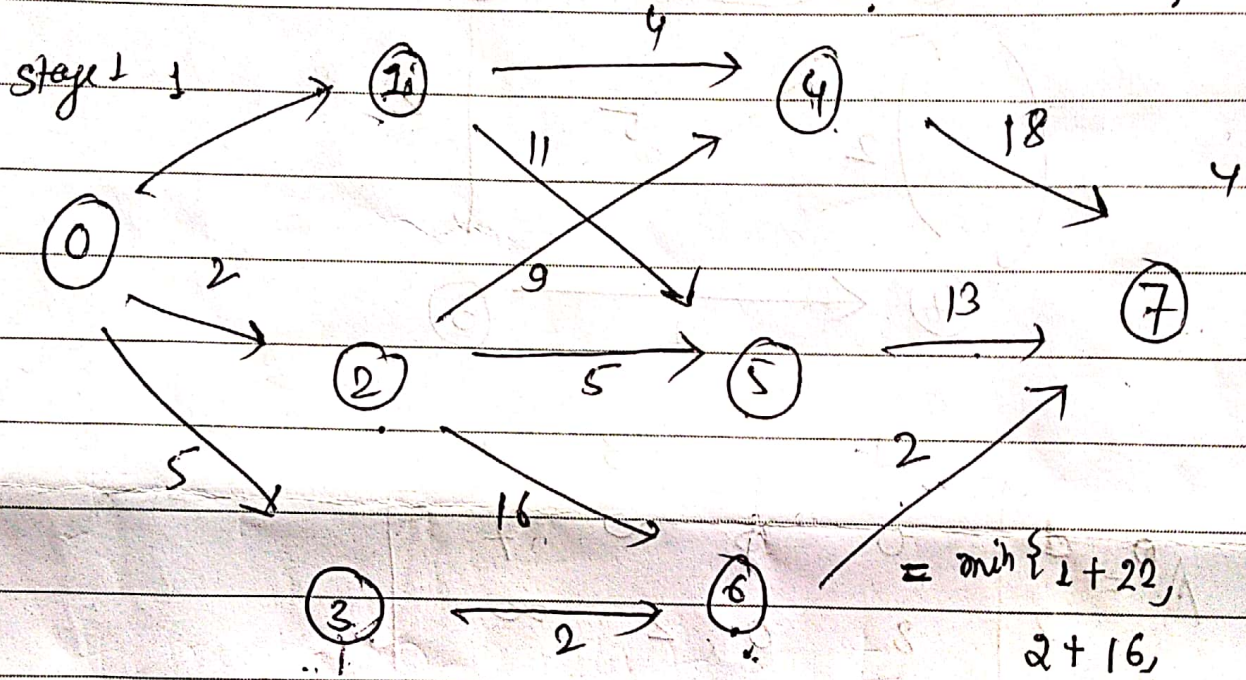
$c(3,2) = \min\{ c(3,6) + c(6,3) \}$

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

$= 4$

Multistage Graph:

→ A directed graph



v	0	1	2	3	4	5	6	7
cost	5	22	16	4	18	13	2	0
d	3	4	6	6	7	7	7	7

$c(7,4) = 0$

$c(4,3) = 18$

$c(5,3) = 13$

$c(6,3) = 2$

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

$= \min\{ 4+18, 11+13 \}$

$= 22$

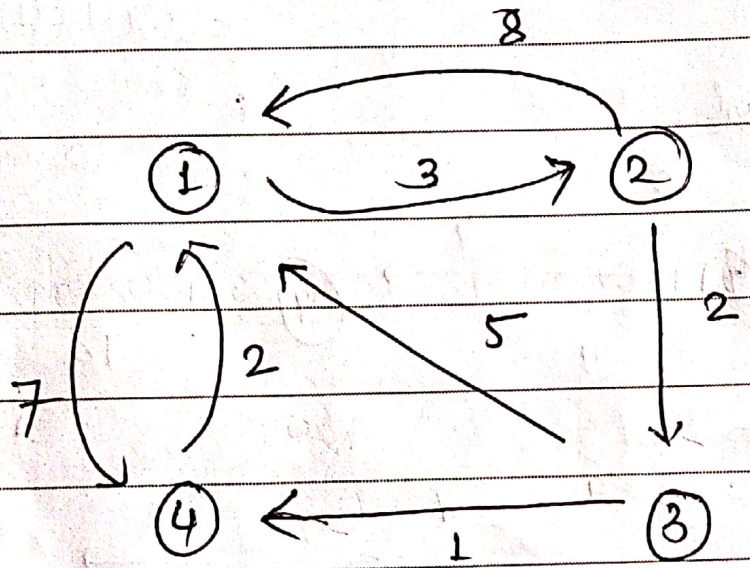
$c(2,2) = \min\{ c(2,4) + c(4,3), c(2,5) + c(5,3), c(2,6) + c(6,3) \}$

$9+18, 5+13, 16+2$

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

$= \min\{ 1+22, 2+16, 5+4 \}$

All Pairs shortest Path: (floyd-warshall)



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

Path between vertices through intermediate vertex

A^1 (path between vertices with 1 as intermediate vertex)

1 → 1st row/col uncharged

$A^{(1)} =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

$A^1[2,3]$ ⇒ path from 2 to 3 through vertex 1.

$A^0[2,3] =$ 2 Now let 1 vertex in between

i.e. 2 ⟶ 1 ⟶ 3

$A^0[2,1] = 8$ $A^0[1,3] = \infty$

i.e.

$A^0[2,3]$ $A^0[2,1]$ $A^0[1,3]$

2 8+∞ [2 (lesser)]

$A^0[2,4]$ $A^0[2,1]$ $A^0[1,4]$

∞ 8+7

$A^0[3,2]$ $A^0[3,1] + A^0[1,2]$

∞ 5+3

$A^0[3,4]$ $A^0[3,1] + A^0[1,4]$

1 5+7

1 < 12

$A^0[4,2]$ $A^0[4,1] + A^0[1,2]$

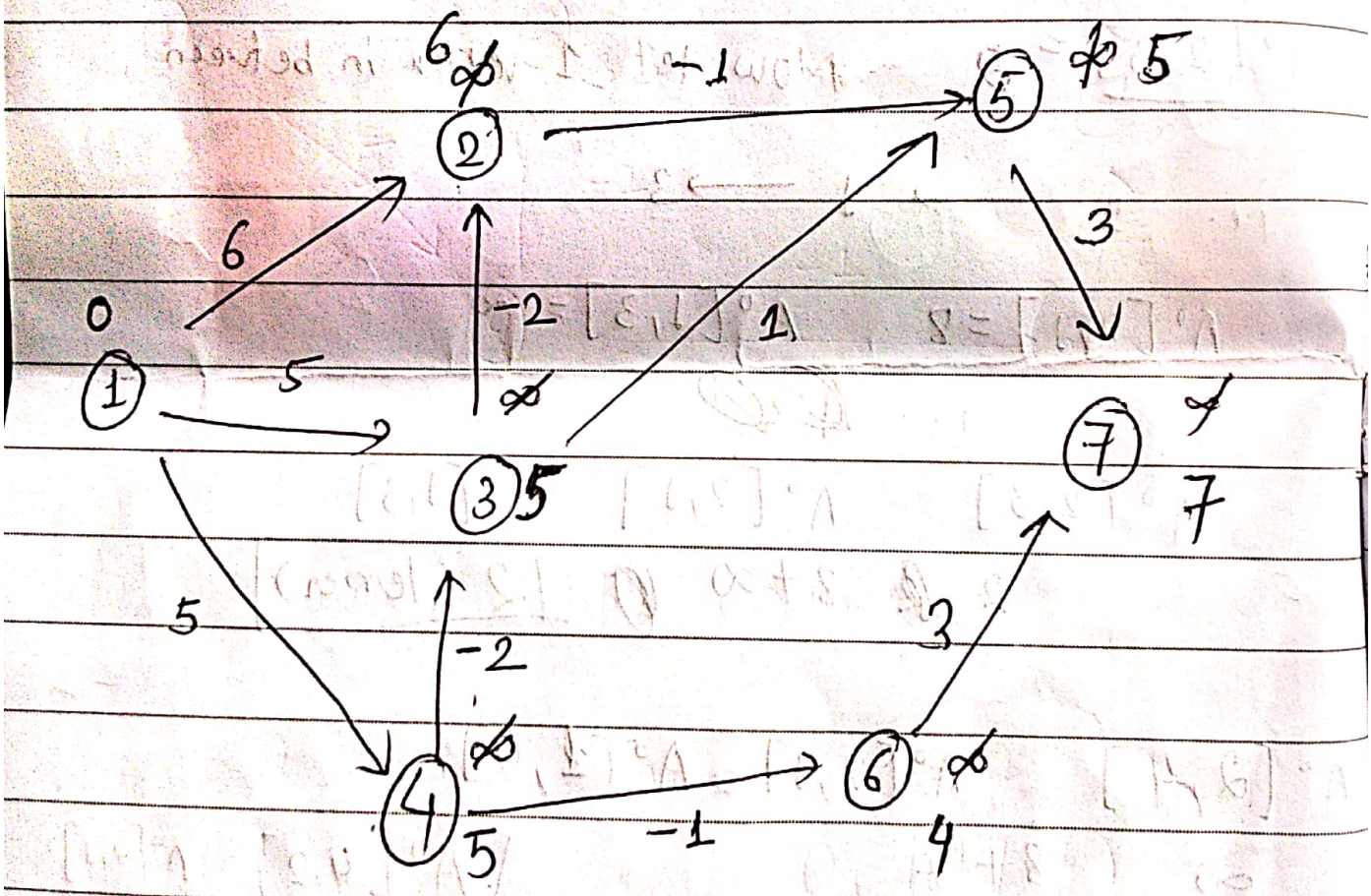
∞ 2+3

$A^0[4,3] = A^0[4,1] + A^0[1,3]$

∞ 2+∞

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

Single Source shortest Path: (Bellman-ford)



Step 1: Assign source vertex 0 and other ∞

Relax (V-1) times = (7-1) = 6 times

Dijkstra's Algorithm – Single Source Shortest Path:

Assumption: weight of all edges is non-negative.

Algorithm:

1. Initializes the distance of source vertex to zero and remaining vertices to ∞.

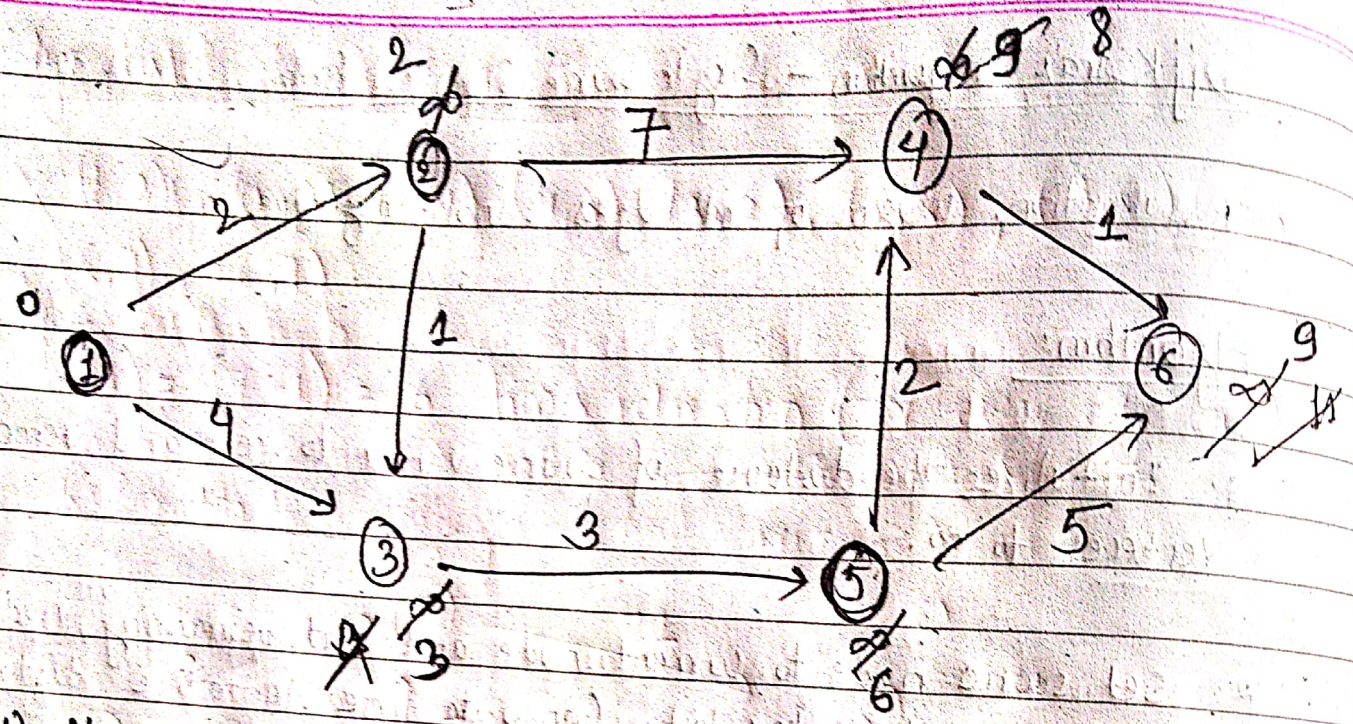
2. Set source node to current node and put remaining all nodes in list of unvisited vertex list. Compute the tentative distance of all immediate neighbour vertex of current node.

3. If the newly computed value is smaller than old value, update it. If 'u' be the source vertex and 'v' be destination vertex and 'c(u,v)' be cost from u to v, then;

```
if ( d[u] + c[u,v] < d[v] ){
        d[v] = d[u] + c[u,v]
}
```

4. When all the neighbours of a current node are explored, mark it as visited. Remove it from unvisited vertex list. Mark the vertex from unvisited list with minimum distance and repeat the process.

5. Stop when the destination node is tested or when unvisited vertex list becomes empty.



(1) Mark source vertex as 0 and all other vertices as ∞.

(2) Set source node as current node and put all remaining nodes in unvisited list. Compute the tentative distance of all immediate neighbour vertex.

(3) If new distance is less than old, relax.

(4) When all neighbouring nodes are explored, mark current node as visited. Set current node to node with minimum distance from unvisited list and repeat the process.

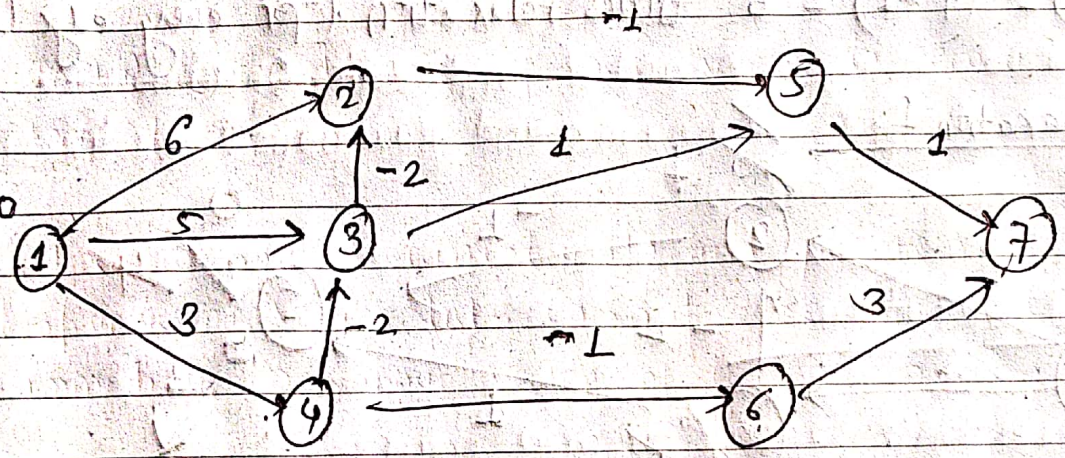
So, the shortest path is:

$$1 \xrightarrow{2} 2 \xrightarrow{1} 3 \xrightarrow{3} 5 \xrightarrow{2} 4 \xrightarrow{1} 6$$

Single Source Shortest Path - Bellman Ford (Dynamic Programming)

Dijkstra Algorithm cannot be used if there are negative weights.

→ Relaxation for all edges.



[Relax (V-1) times]

~~Edge list:~~

The algorithm:

→ Initialize distance to the source 0 and all others infinity.

→ for all the edges, if distance to destination can be shortened by taking edges, distance is updated to new lower value.

Since, longest possible path without a cycle can be (V-1) edges, the edges must be scanned (V-1) times to ensure shortest path has been found for all nodes. so complexity is: n^2

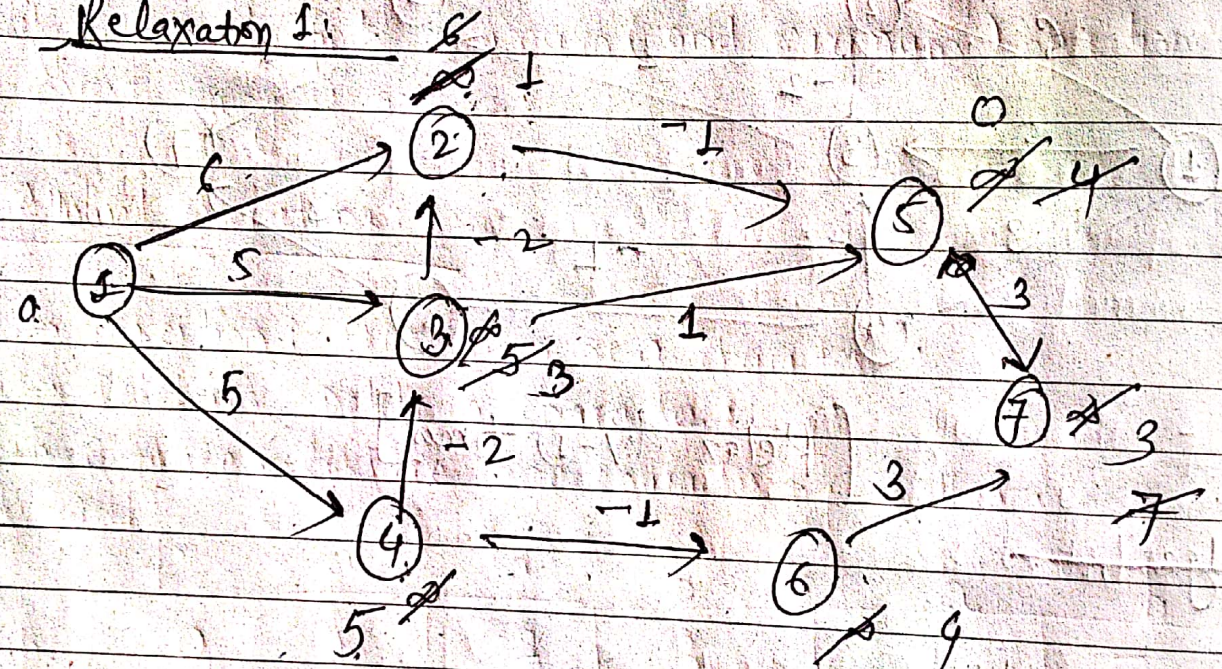
$$O(|E| \cdot |V-1|)$$

Edge list:

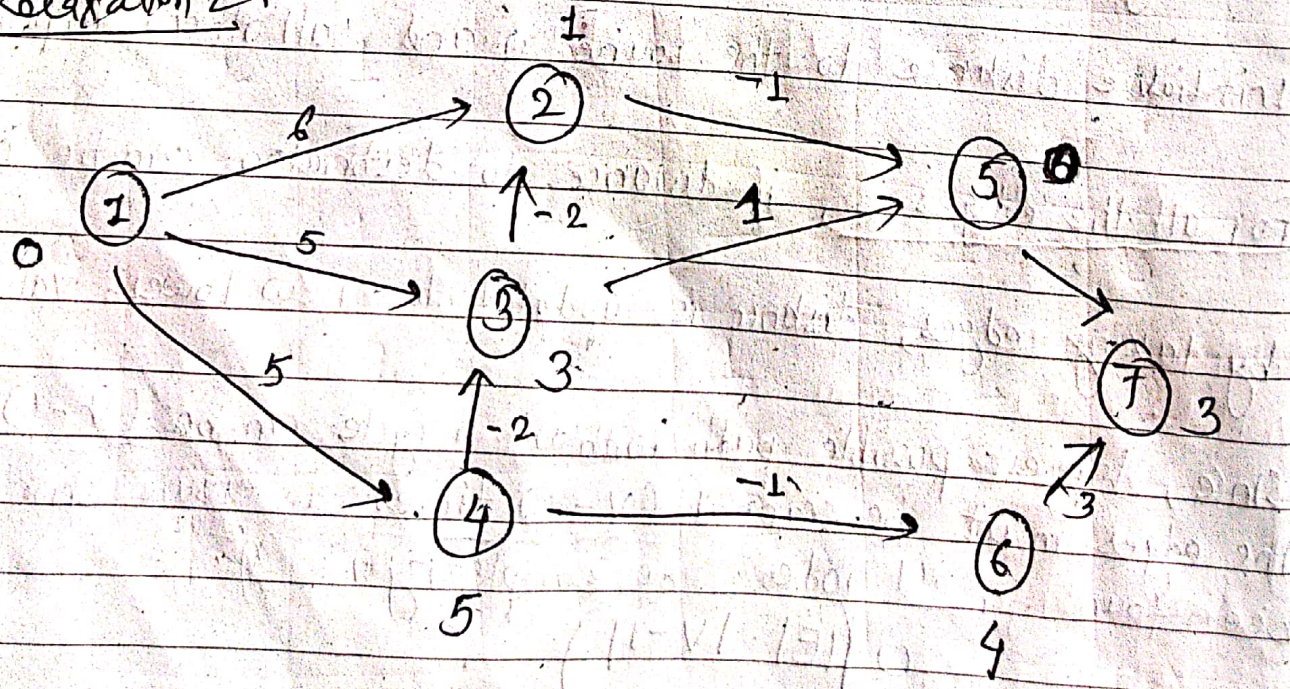
(1,2), (1,3), (1,4), (4,3), (3,2), (2,5), (3,5), (4,6), (5,7), (6,7).

(V-1) = (7-1) = 6 times relaxation (for every edge list)

Relaxation 1:



Relaxation 2:



String Editing Problem:

X = a, a, b, a, b

Y = b, a, b, b

Given two strings X and Y ~~and edit Operations~~. Write an algorithm to find minimum (and maximum) operations required to convert string s_1 and s_2.

Allowed Operations:

→ Insertion: Insert a new character.
→ Deletion: Delete a character.
→ Replace: Replace one character by another.

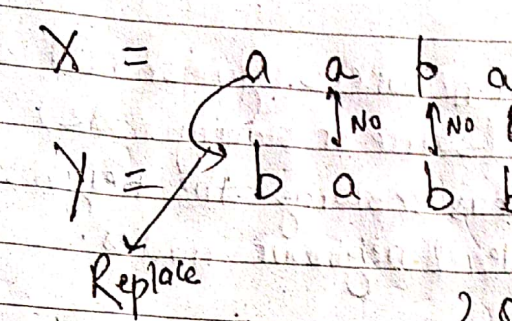
(Source) X →

(Destination) ↓

	ϕ	a	a^2	b	a	b
ϕ	0	1	2	3	4	5
b	1	1	2	2	3	4
a	2	1	1	2	2	3
b	3	2	2	1	2	2
b	4	3	3	2	2	2

(i,j)

for internal cells: if X[i] check surrounding value. select the smallest & add 1. If same, copy diagonal value.



2 operation (Replace + Remove)

~~for~~: T[i,j][0] = i_1 & T[0][i_2] = i_2 (i_1<n1; i++ & i_2<n2, i_2

For internal cells.

if str1[i] == str2[j]
then T[i][j] = T[i-1][j-1] // diagonal element

else
T[i][j] = min (T[i-1][j-1], T[i-1][j], T[i][j-1]) + 1

Q1 Knapsack Problem:

$m = 8$ ← Total capacity, $n = 4$ ← 4 objects

$P = \{1, 2, 5, 6\}$ Prices

$W = \{2, 3, 4, 5\}$ weights

Tabular method:

	P_i	w_i
1st	1	2
2nd	2	3
3rd	5	4
4th	6	5

V \ Bag capacity (W) →	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

* for 1st item, $w_i = 2$, $P_i = 1$; it can be filled only if capacity of Bag is 2 and that time its prize will be 1. For bag greather than 2, same is the case. For bag less than 2, it cannot be put into it so prize is 0.

for i^{th} item, we consider items upto $(i-1)$.

for 2nd item, w = 3 so it can be filled if bag capacity is 3 / greater

if Bag capacity is 1, neither item 1 / item 2 can be filled.
if Bag capacity is 2, item 1 can be filled so p = 1
if Bag capacity is 3, item 2 can be filled so p = 2
if Bag capacity is 4, only of items 1 or 2 can be filled. We choose item with max prize i.e. item 2.

if Bag capacity is 5 or above, both 1 and 2 can be filled.

for item 3 : w = 4, p = 5 - we should consider item 1, item 2, item 3,

when Bag capacity

= 1, none
= 2, item 1
= 3, item 2
= 4, item 3 (max prize)
= 5, item 3 (Because item 1 + item 2 < item 3
$p_1 + p_2 < p_3$)
= 6 item 3 + item 1
= 7 item 3 + item 2
= 7 item 3 + item 1

$V[i, w] = \max\{ V[i-1, w],\ V[i-1, w - w[i]] + p[i] \}$

(w of bag; w of object; $p[i]$ = price of obj)

for 4th item,

$V[4,1] = \max\{ V[3,1],\ V[3, 1-5] + 6 \}$

$= \max\{ V[3,1],\ V[3,-4] + 6 \}$

index is (-)ve upto 4 so from 0 to 4, write previous values.

$V[4,5] = \max\{ V[3,5],\ V[3, 5-5] + 6 \}$ ($V[3,0]$)

$= \max\{ 5,\ 0+6 \}$

$= 6.$

$V[4,6] = \max\{ V[3,6],\ V[3, 6-5] + 6 \}$

$= \max\{ 6,\ 0+6 \}$

$= 6$

$V[4,7] = \max\{ V[3,7],\ V[3, 7-5] + 6 \}$

$= \max\{ 7,\ 1+6 \} = 7$

$V[4,8] = \max\{ V[3,8],\ V[3, 8-5] + 6 \}$

$= \max\{ 7,\ 2+6 \} = 8$

(Q)

i	P_i	W_i	P_i/W_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

n = 4, w = 16

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
0	0	40	40	40	40	40	70	70	70	70	70	70	70	70	70	70
0	0	40	40	40	40	40	70	70	70	70	70	90	90	90	90	90
0	0	40	40	40	40	40	70	70	70	70	70	90	90	90	90	90

For item 1;

When Bag weight = 1

$$V[i, w] = \max\{V[i-1, w], V[i-1, w-w[i]] + p[i]\}$$

V

$i = 3$

for item 4,

$V[i,w] = \max\{V[i-1,w], V[i-1, w-w[i]] + p[i]\}$

$V[4,0] = \max\{V[3,0], V[3,0-5]+10\}$

negative up to 5.

$V[4,5] = \max\{V[3,5], V[3,0]+10\}$

$= \max\{40, 0+10\}$

	40	30	50	10
Now;	x_1	x_2	x_3	x_4
	1	0	1	0

weight = 2 + 10 = 12

price = 90.

max price is 90.

Row 3 and 4 is same ie. 4 is not included.

Row 2 and 3 are diff ie. 3 is included.

2 is also included.

$V[3,12] = \max\{V[2,12], V[2,12-10]+50\}$

$= \max\{70, V[2,2]+50\}$

$= \max\{70, 40+50\} = 90$

Dynamic Programming:

Traveling salesman problem:

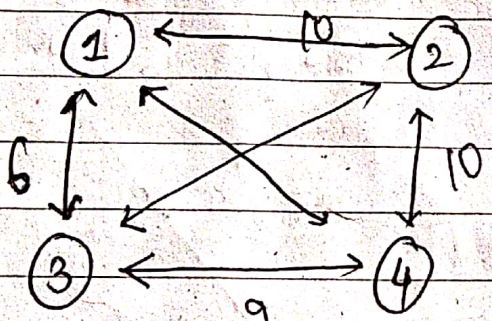
→ Problem statement:

Recurrence Eqn:

$$g(i,s) = \min \{ C_{ij} + g(j, s-\{j\}) \}$$

where $j \in s$

Example:



The cost Adjacency matrix is:

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

Let tour starts from vertex ①:

Assumption :

$g(i, \phi) = c_{i1}$, $1 \le i \le n$

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

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

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

Now, using recurrence eq^n is,

$g(1, \{2, 3, 4\}) =$

The minimum cost of the travelling salesperson problem in given example is 35.

The required path is : $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

(A) N-Queens Problem:

Problem:

We have to place N queens on a N X N chessboard so that no two queens are under 'attack' i.e. two of them are on the same row, column or diagonal.

All the solutions of N queen problem can be represented as N-tuples $(x_1, x_2, x_3, \ldots, x_N)$ where x_i is the column of i^{th} row where Q_i is placed.

The bounding function must check whether two queens are in the same row, column or diagonal.

Suppose two queens are placed at positions (i, j) and (k, l). Then;

(i) Column conflicts:

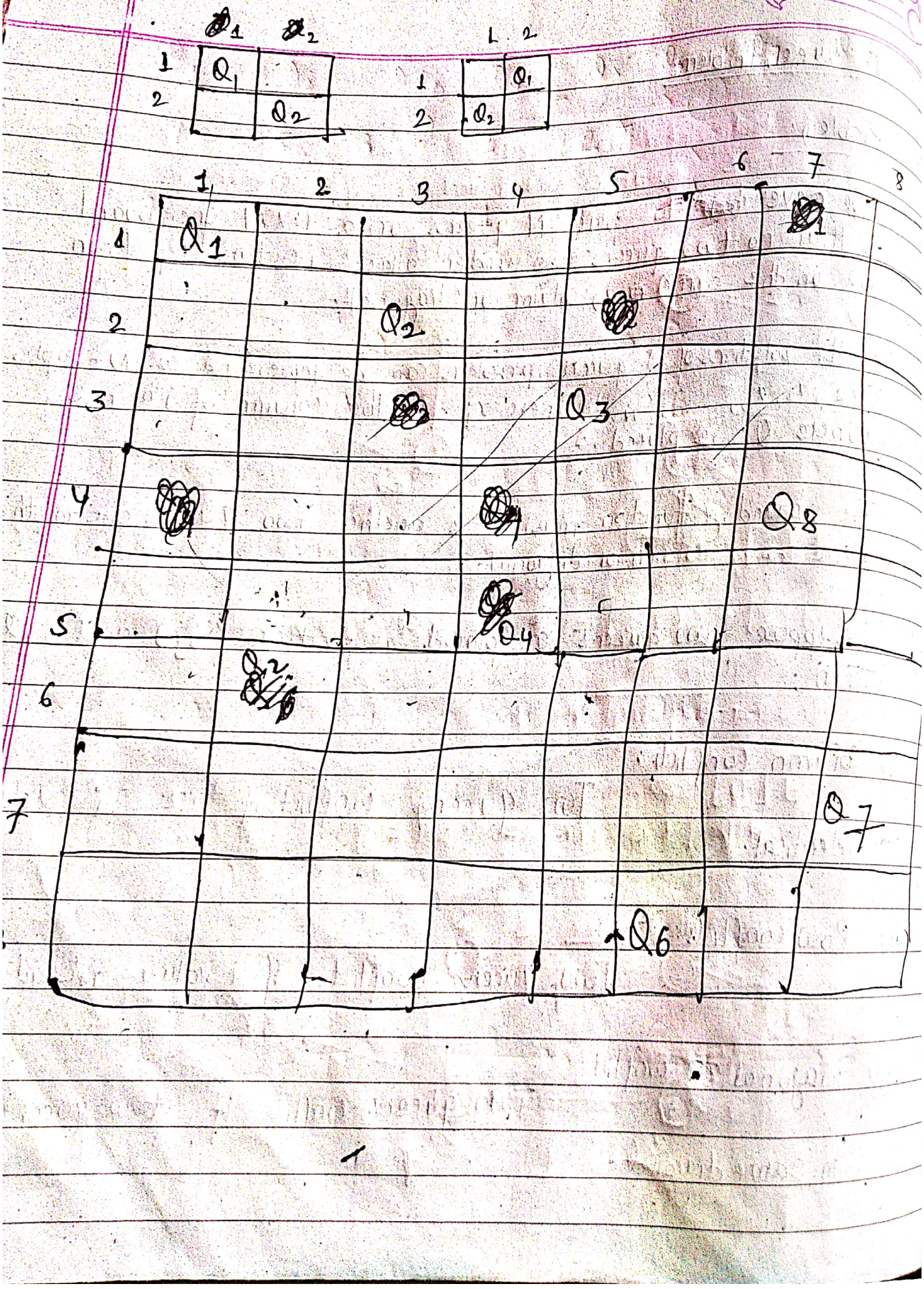
Two queens conflict if their x_i values are identical.

(ii) Row conflicts:

Two queens conflict if i values are identical.

(iii) Diagonal conflict:

Two queens conflict if two queens lie in same diagonal.


Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8

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

Step 1:

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

Step 2:

If this new sequence is feasible, and has length 8, stop with solution. If the new sequence is feasible and has length less than 8, repeat step 1.

Step 3:

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

String Editing problem:

Given two strings string1, string2 and some edit operations. Write an algorithm to find the minimum number of operations required to convert string1 and string2.

The allowed operations are:

- ↳ Insertion: Insert a new character
- ↳ Deletion: Delete a character
- ↳ Substitution: Replace one character by another.

If ~~element~~ f[row] == ~~element~~ f[col], just copy the diagonal value. If not, take minimum of

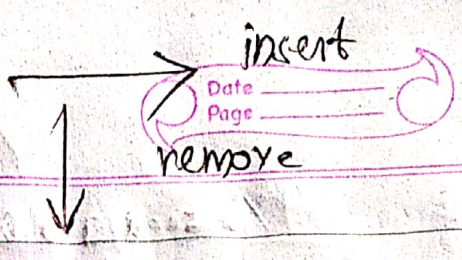
$$\left(f(i-1,j),\ f(i,j-1),\ f[i-1,j-1] \right) + 1.$$

If str1[i] == str2[i]

$$T[i][j] = T[i-1][j-1]$$

Else:

$$T[i][j] = \min\{ T[i-1][j],\ T[i][j-1],\ T[i-1][j-1] \} + 1.$$



Example:

X = aabab
Y = babb

a a b a b → replace

	ϕ	a	a	b	a	b
ϕ	0	1	2	3	4	5
b	1	1	2	2	3	4
a	2	1	1	2	2	3
b	3	2	2	1	2	2
b	4	3	3	2	2	2

b
a b a b.
b a b b

Keeping source at Y-axis:

	ϕ	b	a	b	b
ϕ	0	1	2	3	4
a	1	1	1	2	3
a	2	2	1	2	3
b	3	2	2	1	2
a	4	3	2	2	2
b	5	4	3	2	2

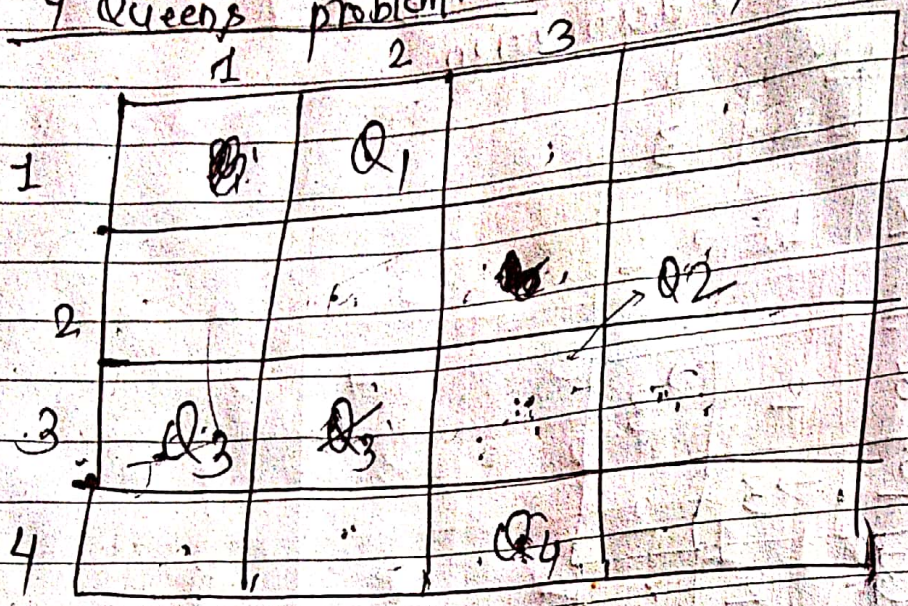
$X = aaba$ $\quad\quad$ $Y = bab$

$\longrightarrow$ destination

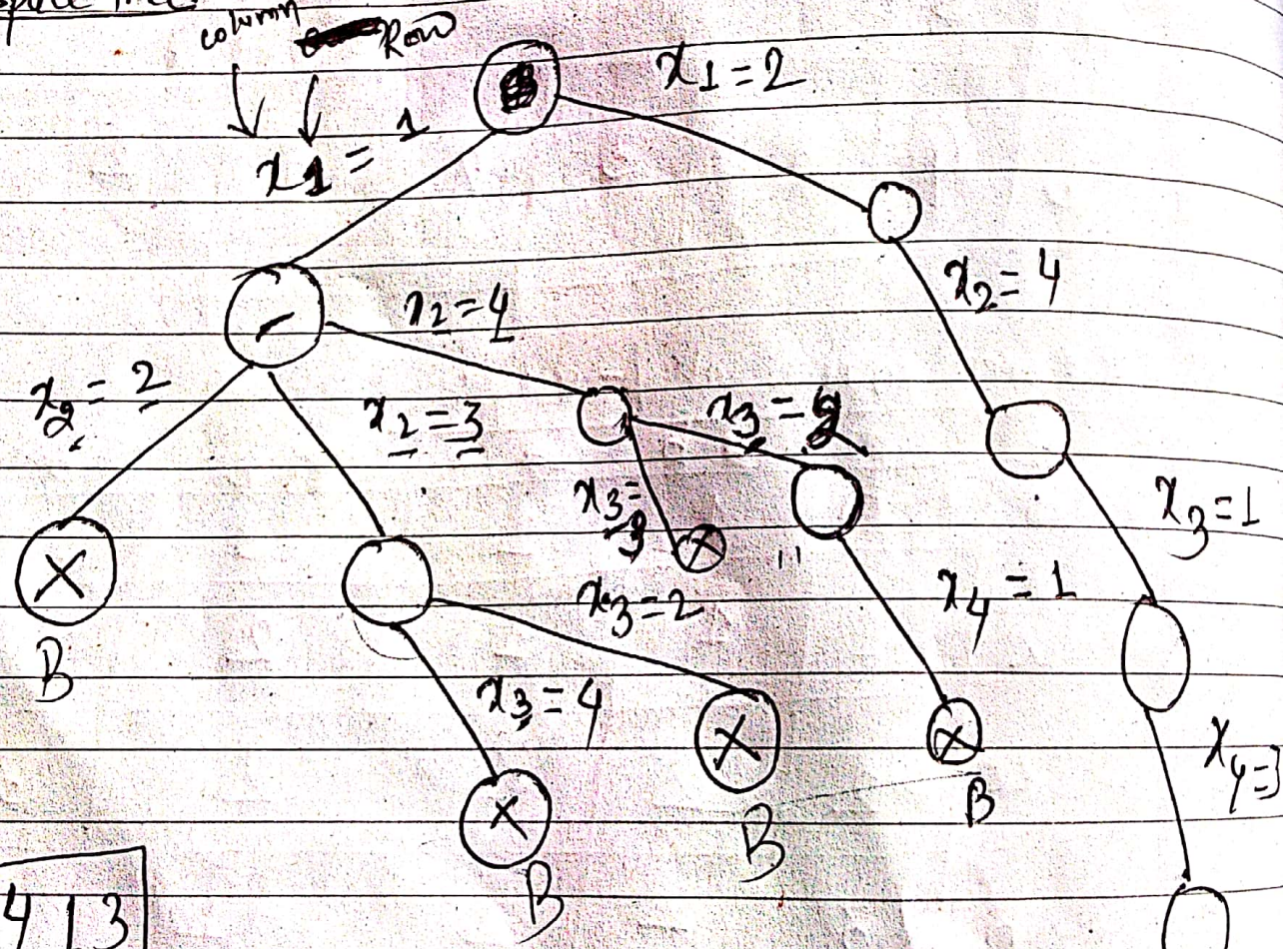
	ϕ	b	a	b
ϕ	0	1	2	3
a	1	2	1	2
a	2	3	2	3
b	3	2	3	2
a	4	3	2	3

urie

4 Queens problem:

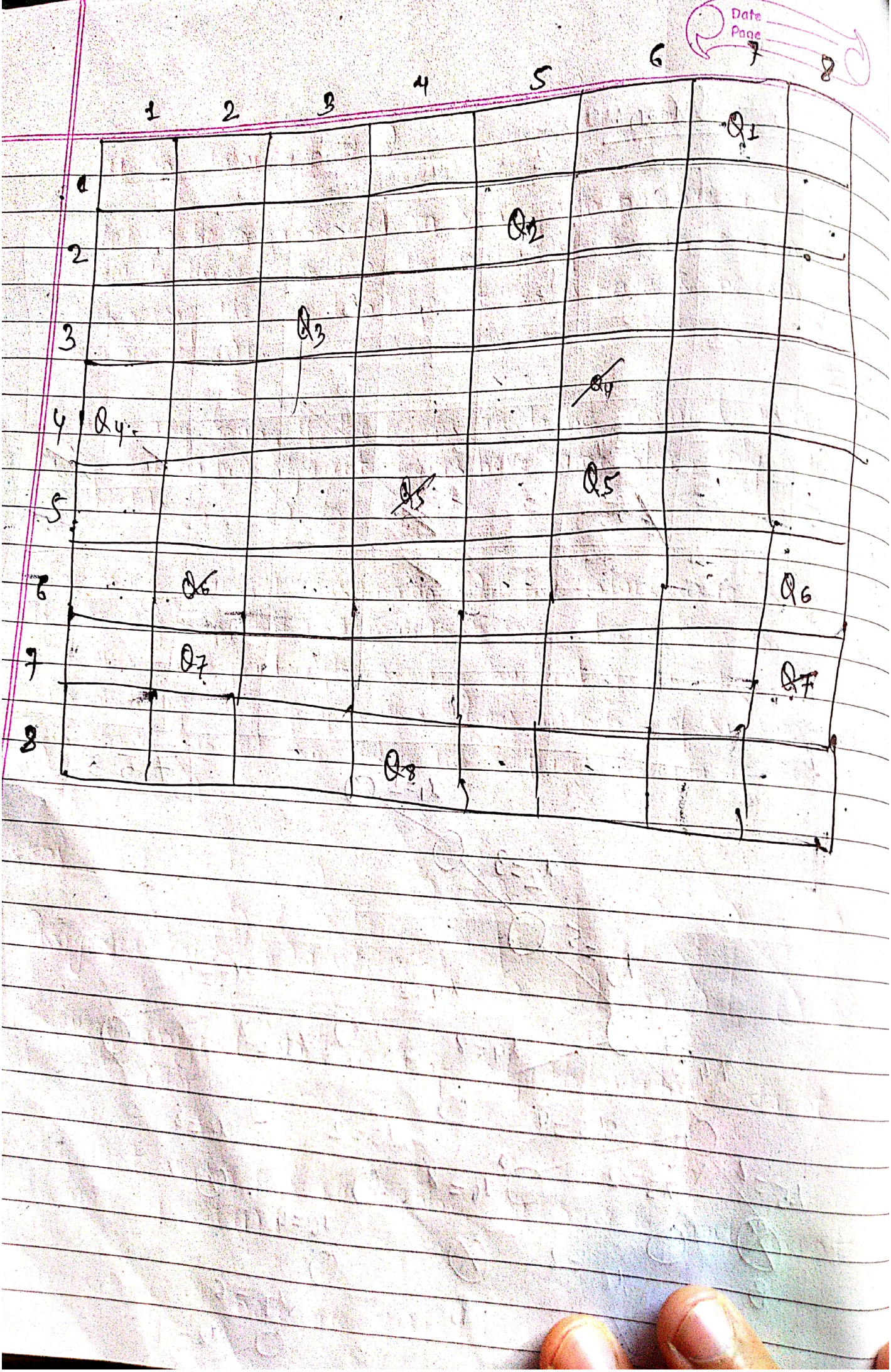


State space tree:



2 4 1 3

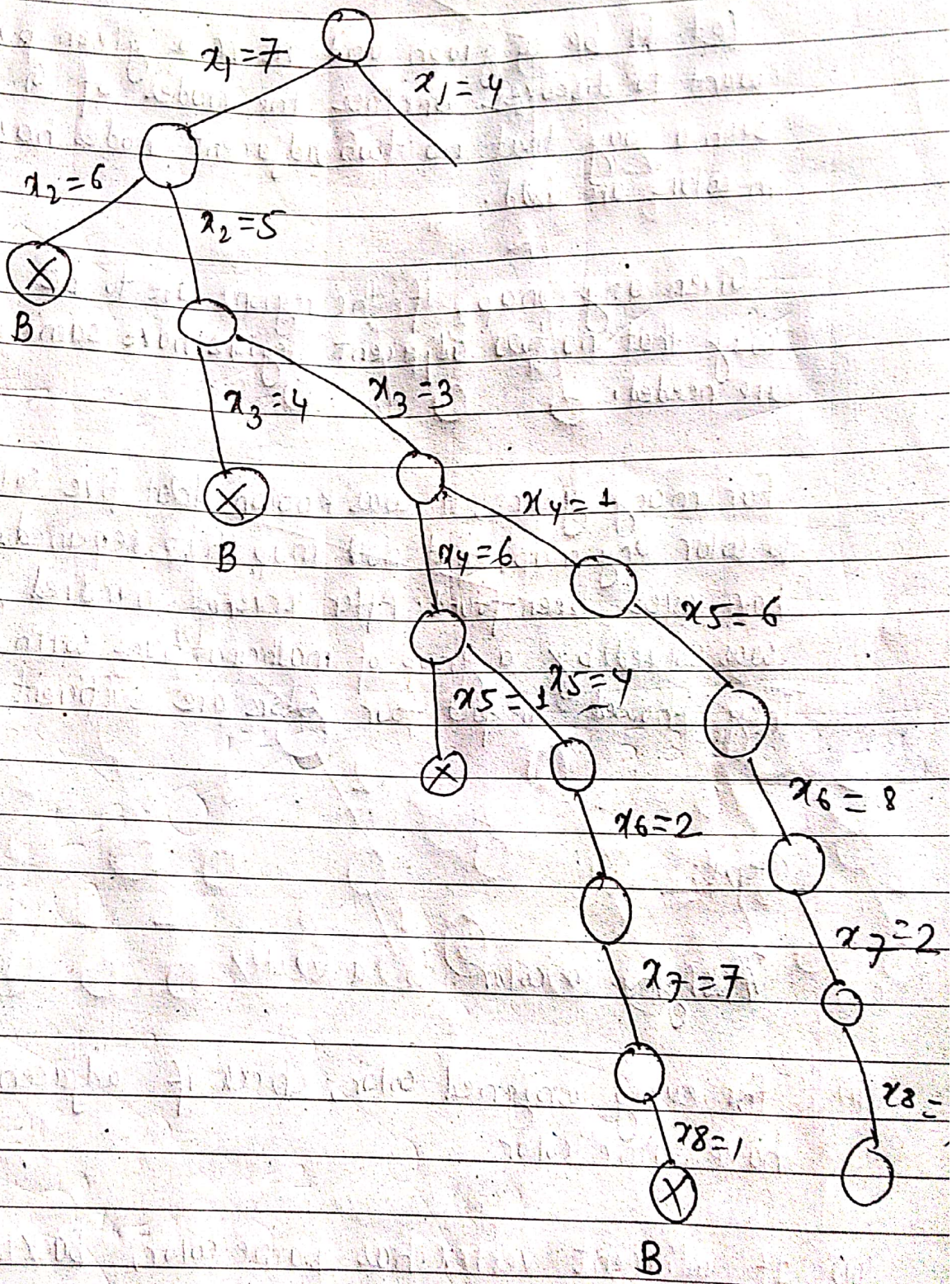
7 5 3 1 6 8 2 4



75316824

46897 L35

let keep 1st Queen at 7th col.



Graph Coloring problem:

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have same color yet only m colors are used.

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

For many years, it was known that five colors were sufficient to color any map, but not map that required more than four colors have ever been found. After several hundred years, this problem was solved by a group of mathematicians with help of a computer. They showed in fact four colors are sufficient for planar graphs.

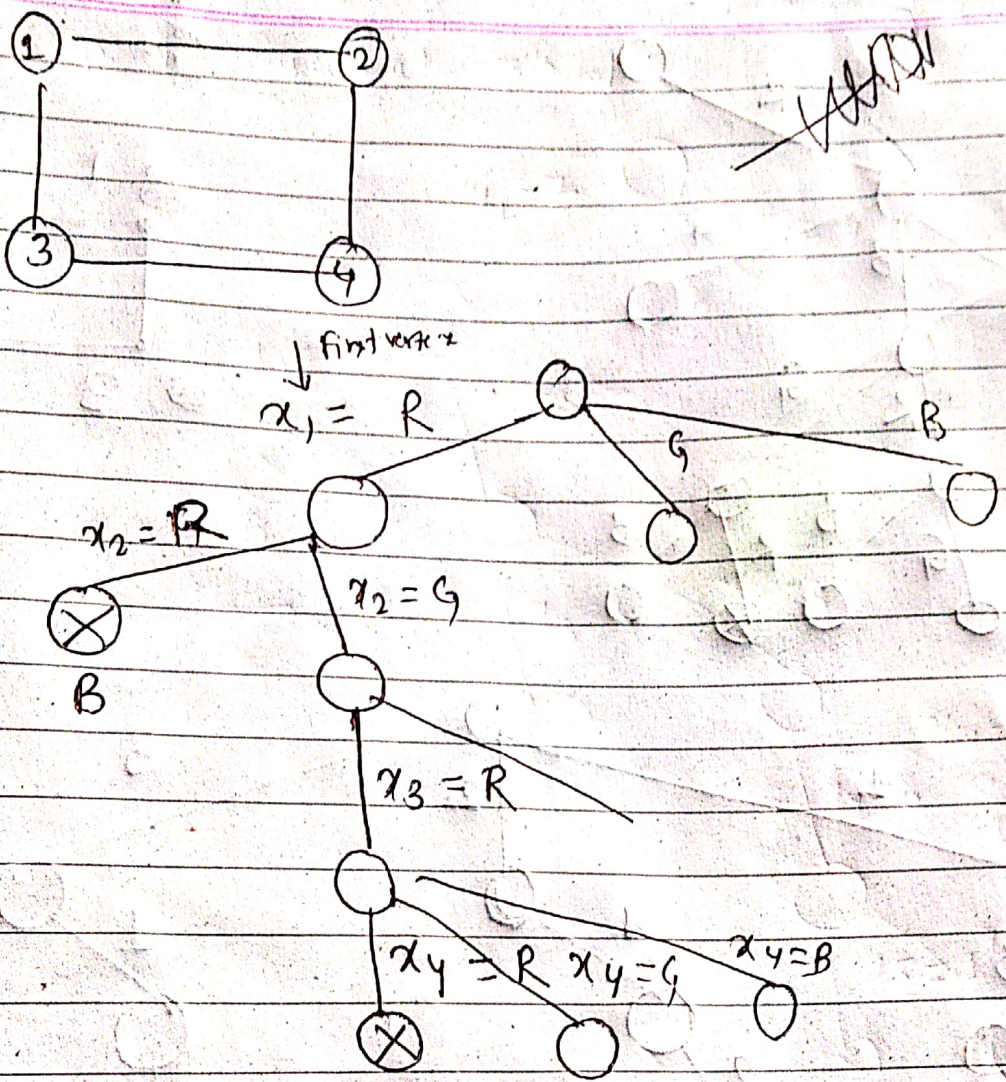
Steps:

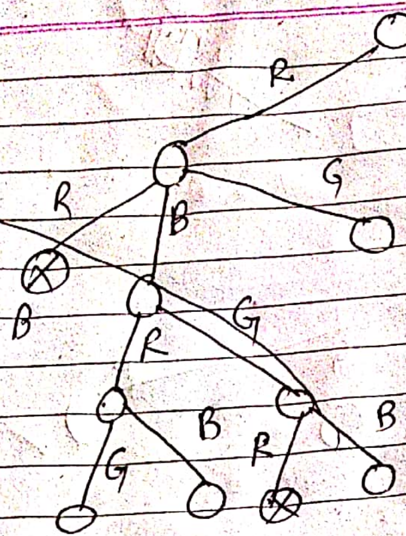
(i) Assign a color to a vertex

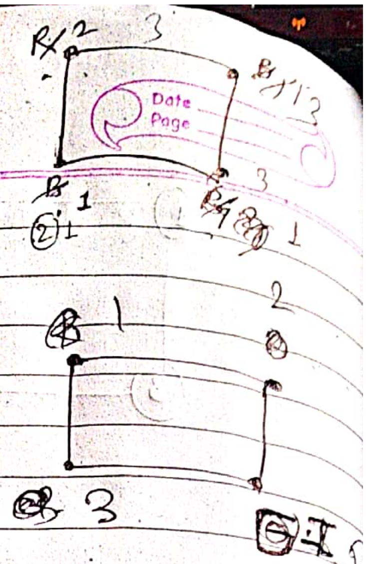
(ii) for every assigned color, check if adjacent vertices donot have same color

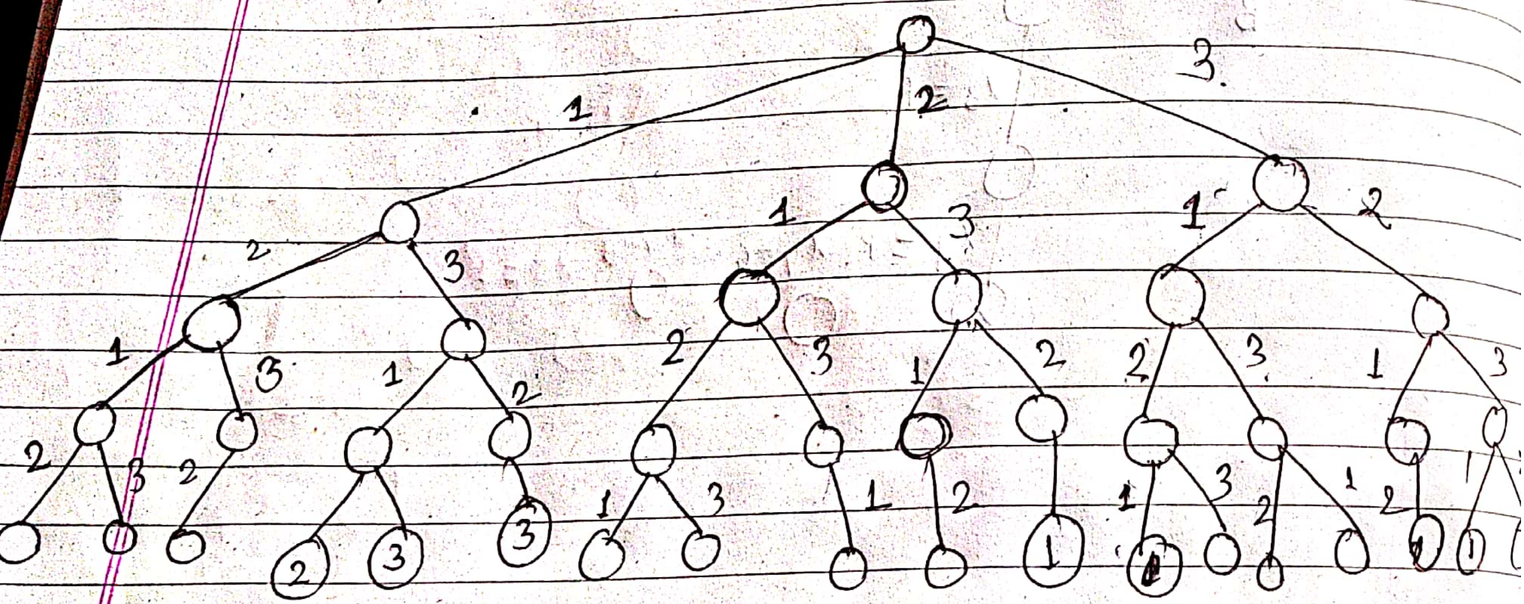
(iii) If adjacent vertex has same color, backtrack to most recent node and repeat (i)

(iv) Terminate if all nodes are visited.









Hamilton cycles:

For a graph G, there should be a path that visits all the nodes and return back to original node.