TIME COMPLEXITY AND ASYMPTOTIC NOTATION

$O \rightarrow f(n) \rightarrow$ upper bound

$\Omega \rightarrow f(n) \rightarrow$ lower bound

$\theta \rightarrow f(n) \rightarrow$ upper + lower bound

Big $O \rightarrow$ If $f(n)$ and $g(n)$ are two functions, then $f(n) = O(g(n))$, If there exists constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Example —

$f(n) = 2n^2$

- $3n^2$ → Least upper bound.
- n^3
- n^4
- ⋮
- n^n

$f(n)$ ka least upper bound, $g(n)$ ka kya hoga Big O.

* If $f(n) = 4n+3$. Is $f(n) = O(g(n))$?

$\Rightarrow f(n) \leq c \cdot g(n)$

$\Rightarrow 4n+3 \leq 5n$

c , $g(n)$

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

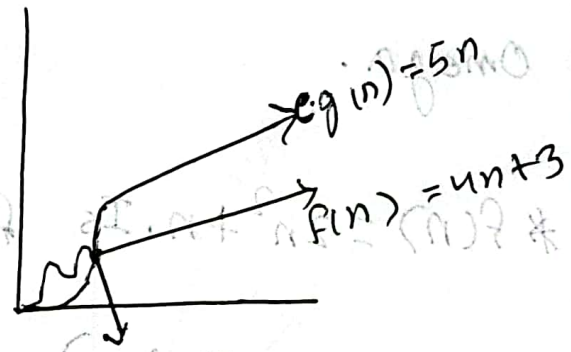
Now,

$4n+3 \leq 5n$

$\Rightarrow 3 \leq 5n - 4n$

$\Rightarrow 3 \leq n$

n_0



[n_0 point এর পর থেকে $c \cdot g(n) > f(n)$]

* $f(n) = 2n^2 + n$. Find $O(g(n))$.

$\Rightarrow f(n) \leq c \cdot g(n)$

$\Rightarrow 2n^2 + n \leq 3n^2$

c , $g(n)$.

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

Omega (Ω) — If $f(n)$ and $g(n)$ are two functions then $f(n) = \Omega(g(n))$, if there exists constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Example -

$f(n) = 2n^2$ → greatest lower bound $g(n)$

$f(n)$ এর greatest lower bound, $g(n)$ হলে তাকে Omega.

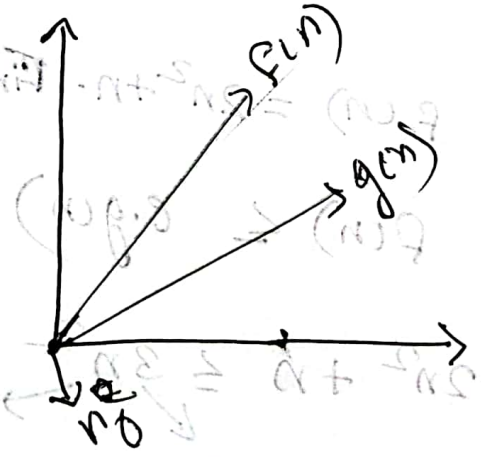
* $f(n) = 2n^2 + n$. Is $f(n) \geq \Omega(g(n))$?

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

$\Rightarrow 2n^2 + n \geq 2n^2$

c $g(n)$

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



Now,

$2n^2 + n \geq 2n^2$

$\Rightarrow n \geq 0$ → n_0

Theta (θ) $\Rightarrow$ If $f(n)$ and $g(n)$ are two functions then $f(n) = \theta(g(n))$, if there exists constants c and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

* $f(n) = 2n^2 + n$, $f(n) = \theta(g(n)) = ?$

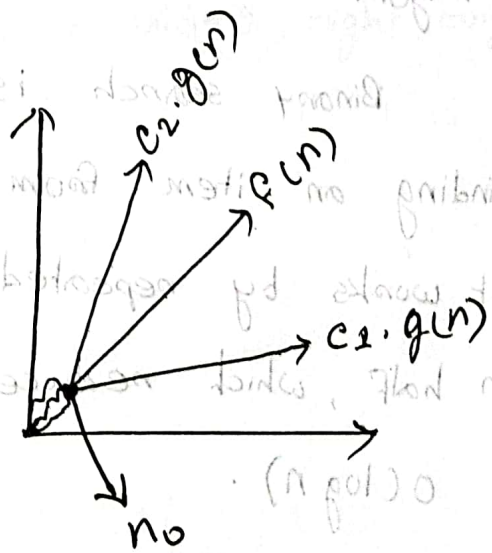
$$c_1 \cdot g(n) \leq f(n) \leq c_2 g(n)$$

$$2n^2 \leq 2n^2 + n \leq 3n^2$$

c_1, $2n^2$ $\rightarrow g(n)$; c_2, $3n^2$ $\rightarrow g(n)$

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

$= \theta(n^2)$



Divide and Conquer

3 steps of D&C approach —

(1) Divide → Creating subproblems

(2) Conquer → Solving subproblems

(3) Combine → Combining solutions to get the final result

BINARY SEARCH ALGORITHM —

*Algorithm —

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half, which reduces the search time complexity to $O(\log n)$.

```
int BinarySearch (int arr[], int left, int right, int target)
{
    if (left <= right)
    {
        int mid = left + (right - left)/2;
        if (arr[mid] == target)
            return mid;
        if (arr[mid] > target)
            return binarySearch (arr, left, mid-1, target);
        else
            return binarySearch (arr, mid+1, right, target);
    }
    return -1;
}
```

Simulation —

1	2	3	4	5	6	7
1	8	10	15	18	20	30

left = 0 , right = 7, mid = 3, target = 15

Here,

arr[mid] = 10 < target.

Now, left = mid + 1 = 3 + 1 = 4

1	2	3	4	5	6	7
1	8	10	15	18	20	30

left = 4 , right = 7 , mid = 5, target = 15

arr[mid] = 18 > target.

Now, right = mid - 1 = 5 - 1 = 4

1	2	3	4	5	6	7
1	8	10	15	18	20	30

left = 4, right = 4, mid = 4 , target = 15

arr[mid] = 15 = target.

So, target found at position 4.

B

Time Complexity –

In each iteration, the search range is divided in half.

Itr = 1, search range size = $\frac{n}{2}$.

Itr = 2, " " " = $\frac{n}{4} = \frac{n}{2^2}$

Itr = 3, " " " = $\frac{n}{8} = \frac{n}{2^3}$

Itr = k, " " " = $\frac{n}{2^k}$

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

$\Rightarrow n = 2^k$

$\Rightarrow \log_2^n = \log_2^{2^k}$

$\Rightarrow k \log_2^2 = \log_2^n$

$\therefore k = \log_2^n$

Best case: element at middle index. Complexity = $\Theta(1)$.

Average " : " found at the middle of search.
Complexity $\Rightarrow \Theta(\log_2^n)$

Worst " : element found at end of the search or not found. complexity $\Rightarrow \Theta(\log_2^n)$.

Ternary Search Algorithm –

The ternary search algorithm is a divide and conquer search technique which is used to find the position of a target element in a sorted array by ~~divide~~ repeated dividing the search space into three equal parts and then narrowing the search to one of these three regions.

Algorithm —

```
int TernarySearch (int arr[], int left, int right, int target)
{
  if (left <= right) {
    int mid1 = left + (right - left)/3;
    int mid2 = right - (right - left)/3;
    if (arr[mid1] == target) return mid1;
    if (arr[mid2] == target) return mid2;
    if (target < arr[mid1])
      return TernarySearch(arr, left, mid1-1, target);
    else if (target > arr[mid2])
      return ternarysearch(arr, mid2+1, right, target);
    else return ternarysearch(arr, mid1+1, mid2-1, target);
  }
  return -1;
}
```

Simulation —

0	1	2	3	4	5	6	7	8
6	8	14	16	18	20	23	32	45

left = 0, mid1 = 2, mid2 = 6, right = 8, target = 45

Here,

arr[mid1] = 14 < arr[mid2] = 23 < 45

As target > arr[mid2], then left = mid2 + 1 = 6 + 1 = 7

0	1	2	3	4	5	6	7	8
6	8	14	16	18	20	23	32	45

left = 7, mid1 = 7, mid2 = 8, right = 8, target = 45

Now, arr[mid2] = target.

So, target found at position 8.

Time Complexity:

Itr = 1, search range size = $\frac{n}{3}$

Itr = 2, " " = $\frac{n}{9} = \frac{n}{3^2}$

Itr = 3, " " = $\frac{n}{27} = \frac{n}{3^3}$

Itr = k, " " = $\frac{n}{3^k}$

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

$\Rightarrow n = 3^k \quad \therefore k = \log_3 n$

Best case : element at the middle index.

$\Theta(1)$.

Average case: element found in the middle of the search. $\Theta(\log_3^n)$

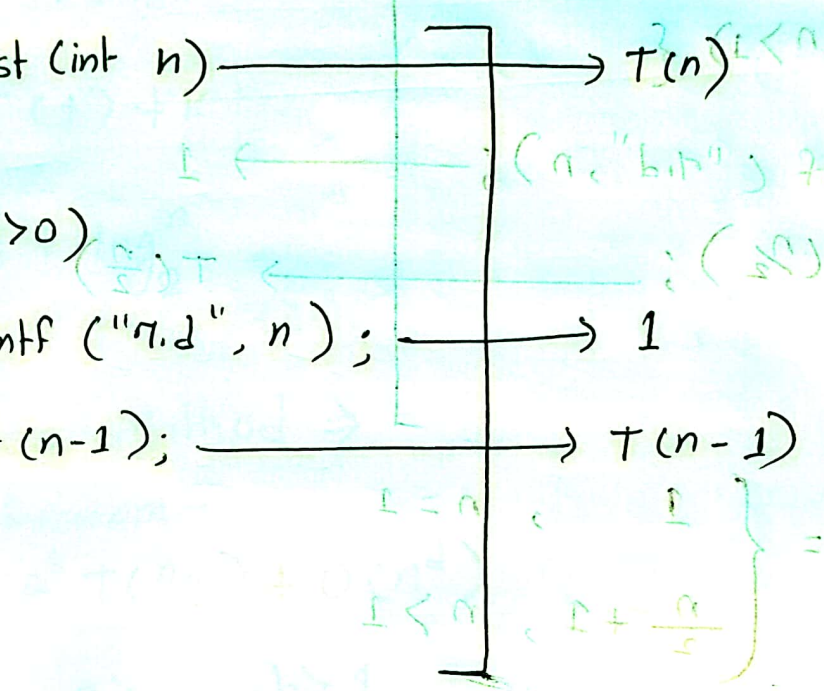
Worst case: element found at either end of the search or not found.

$\Theta(\log_3^n)$.

Recurrance Relation -

* Substitution Method -

⇒
```
void Test (int n)
{
    if(n>0)
    { printf ("%d", n);
      Test (n-1);
    }
}
```



$$T(n) = \begin{cases} 1 & , n = 0 \\ T(n-1) + 1, & n > 0 \end{cases}$$

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

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

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

$= T(n-3) + 3$

$= T(n-k) + k$

If $k = n$

$= T(n-n) + n$

$= T(0) + n$

$= 1 + n$

∴ Time complexity = $O(n)$

⇒ void Test (int n) → T(n)

{ if (n > 1) {

printf ("%d", n); → 1

Test (n/2); → T(n/2)

}

}

$$T(n) = \begin{cases} 1 & , \ n = 1 \\ \frac{n}{2} + 1 & , \ n > 1 \end{cases}$$

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

$$= \left[T\left(\frac{n}{4}\right) + 1\right] + 1$$

$$= \left[\left[T\left(\frac{n}{8}\right) + 1\right] + 1\right] + 1$$

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

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

when,

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

$$\Rightarrow 2^k = n$$

$$\Rightarrow \log_2^{2^k} = \log_2^n$$

$$\Rightarrow k \log_2^2 = \log_2^n$$

$\therefore k = \log_2^n$

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

$= 1 + \log_2^n$

Master Method ⇒

⇒ Master theorem –

$T(n) = a * T(n/b) + O(n^d)$

Here $a \geq 1$, $b > 1$

Here,

$T(n)$ = Time complexity of the algorithm on an input of size n.

a = number of recursive subproblems in each division steps.

$\frac{n}{b}$ = size of each subproblem.

$O(n^d)$ = The time complexity of combining the results of all subproblems and any additional work done outside the recursive calls.

⇒ Time complexity (T(n)) of the algorithm can be determined based on three cases —

(1) $O(n^d)$ if $d > \log_b^a$

(2) $O(n^d \log_b^n)$ if $d = \log_b^a$

(3) $O(n \log_b^a)$ if $d < \log_b^a$.

For binary search algorithm —

$$T(n) = 1 * T(n/2) + O(1)$$

Here, $a = 1$ [যেহেতু একটি array কে 2 ভাগ করে এক ভাগ নিয়ে কাজ করা হয়]

$\frac{n}{b} = \frac{n}{2}$; $b = 2$.

$n^d = 1$, So, $d = 0$

Now, $\log_b^a = \log_2^1 = 0 = d$

$\therefore T(n) = O(\log_2^n)$ [if $d = \log_b^a$, $T(n) = O(n^d \log_b^n)$]

For Ternary search algorithm –

$T(n) = 1 * T(\frac{n}{3}) + O(1)$

Here, $a = 1$ [1 বারা নিয়ে কাজ করে তাই]

$b = 3$, $n^d = 1$, So, $d = 0$

Now, $\log_b^a = \log_3^1 = 0 = d$

$\therefore T(n) = O(\log^n_3)$

For merge sort algorithm –

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

Here, $a = 2$ [2 ভাগ নিয়েই কাজ করে]

$b = 2$

$n^d = n$, so, $d = 1$

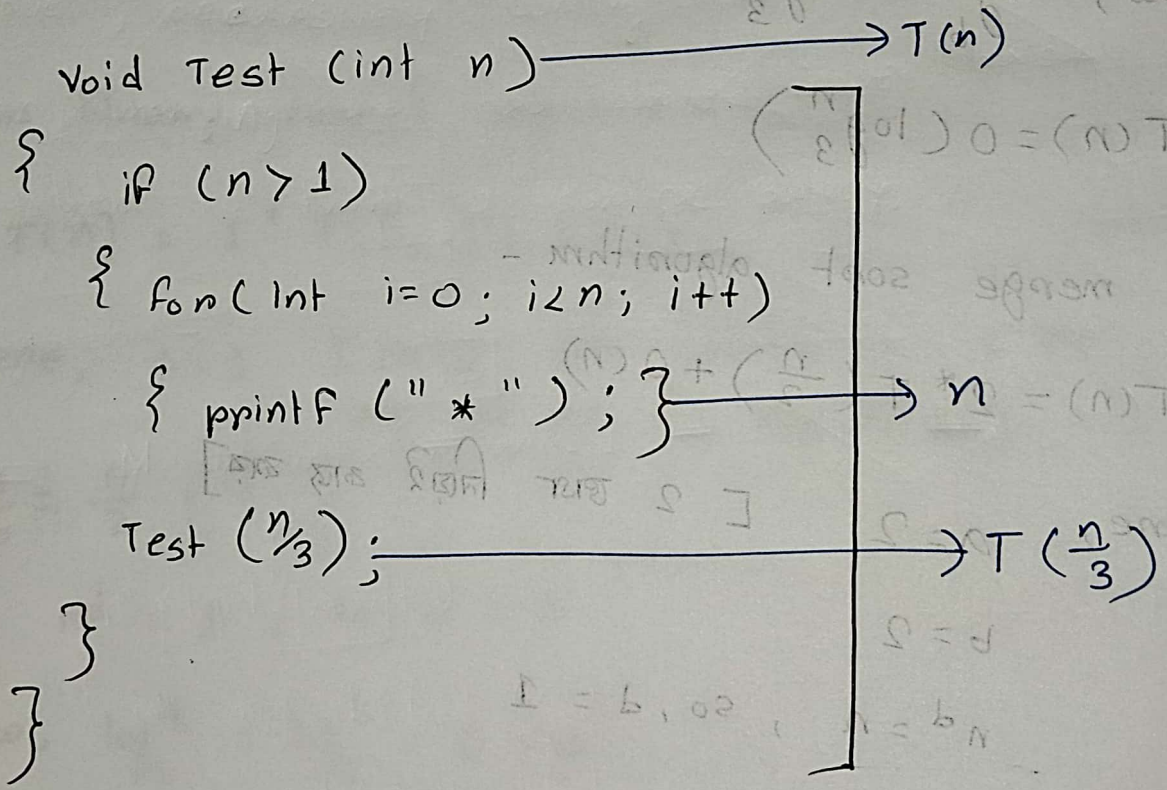
Now, $\log_b^a = \log_2^2 = 1 = d$

$T(n) = O(n \log^n_2)$

CT QUESTION

⇒ Derive recursive relation or equation from the following function and find the time complexity using master method if possible.

```
Void Test (int n)
{
  if (n > 1)
  {
    for(int i=0; i<n; i++)
    { printf ("*"); }
    Test (n/3);
  }
}
```



Recursive relation ⇒

$$T(n) = \begin{cases} 1, & \text{when } n \leq 1 \\ T\left(\frac{n}{3}\right) + n, & \text{when } n > 1 \end{cases}$$

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

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

$a = 1$, $b = 3$, $n^d = n$, so, $d = 1$.

Now,

$$\log_b^a = \log_3^1 = 0 < d.$$

Here,

$$d > \log_b^a.$$

So, $T(n) = O(n^d) = O(n)$.

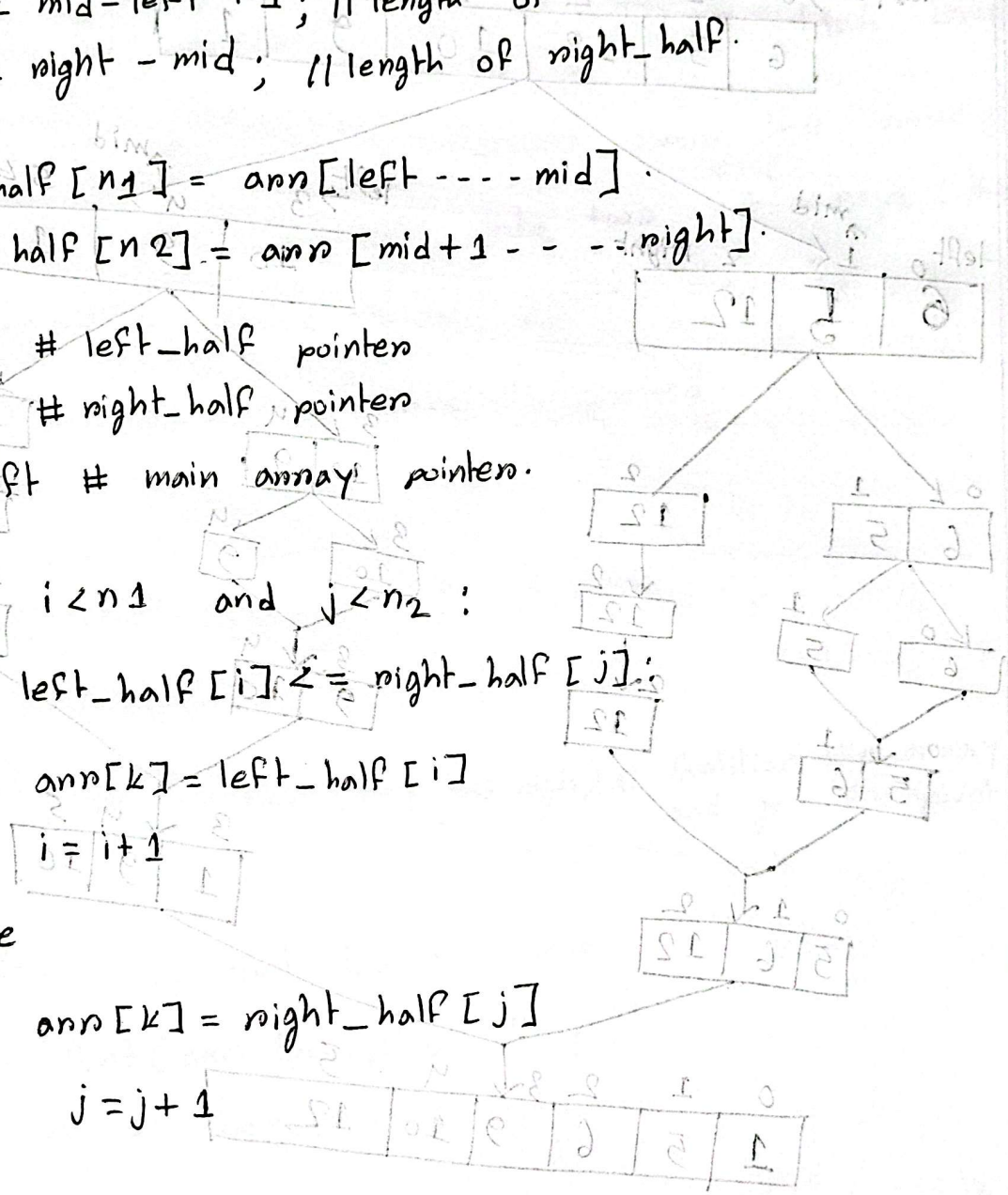
Merge Sort Algorithm :

Merge sort is an efficient, stable and comparison-based sorting algorithm that follows the divide and conquer technique. The algorithm works by dividing an input array into two halves, recursively sorting each half, and then merging the two sorted halves back together to form a sorted array.

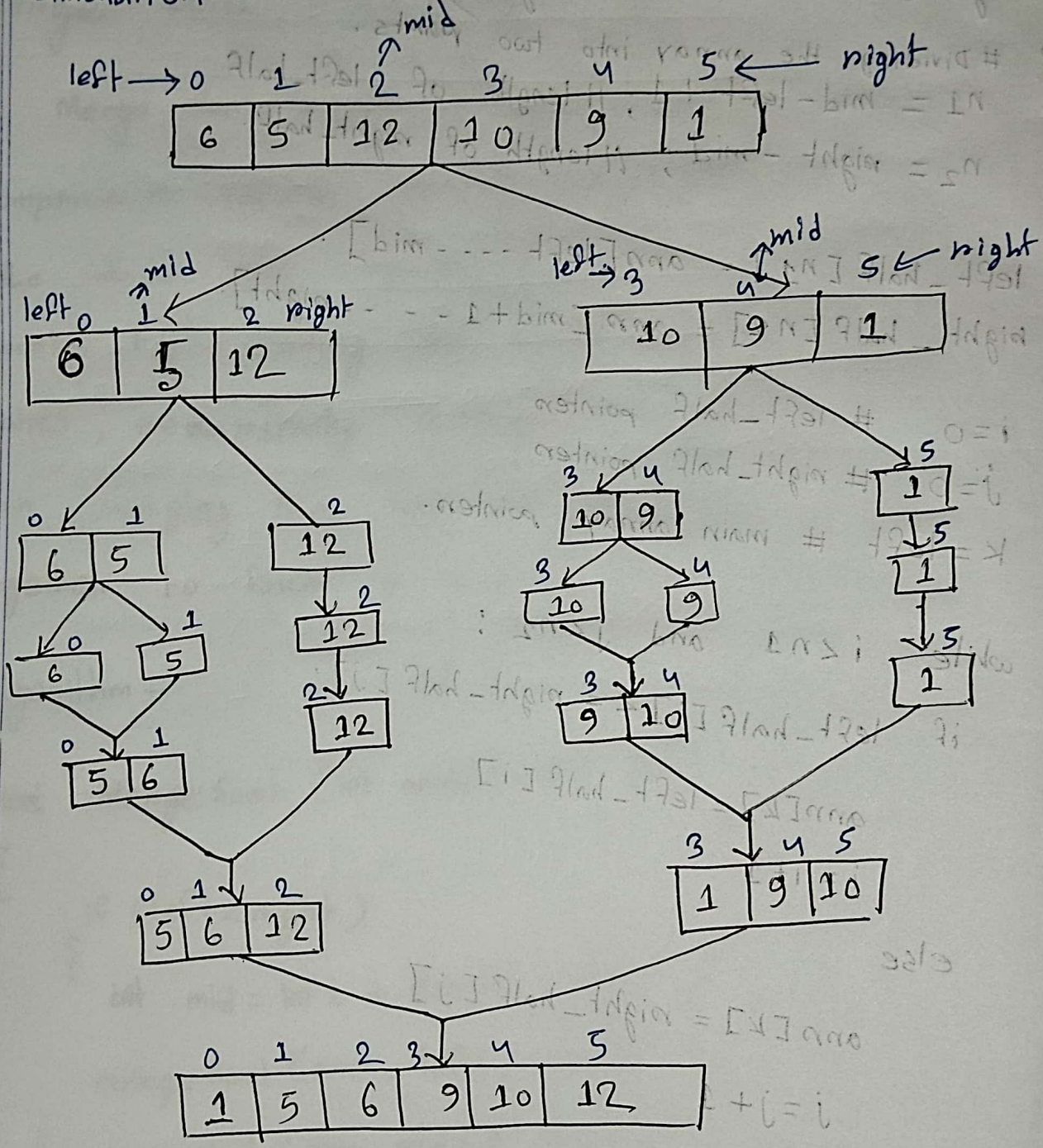
Algorithm -

```
void MergeSort (int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        mergeSort (arr, left, mid);
        mergeSort (arr, mid+1, right);
        merge (arr, left, mid, right);
    }
}
```

```
Merge (arr[], left, mid, right)
{  # Dividing the array into two parts.
   n1 = mid - left + 1 ; // length of left_half
   n2 = right - mid ;  // length of right_half.

   left_half [n1] = arr [left ---- mid] .
   right_half [n2] = arr [mid+1 - - - - right].

   i = 0    # left_half pointer
   j = 0    # right_half pointer
   k = left # main array pointer.

   while  i < n1  and  j < n2 :
      if  left_half [i] <= right_half [j]:
          arr[k] = left_half [i]
          i = i + 1
      else
          arr [k] = right_half [j]
          j = j + 1

      k = k + 1 .

   while  i < n1 :
       arr [k] = left_half [i] .
       i = i + 1
       k = k + 1
   while j < n2 ;
         arr[k] = right_half [j] .
         j = j+1 . k = k+1
}
```



Simulation -



Time Complexity, $T(n) = O(n \log_2 n)$.

Quick Sort Algorithm :

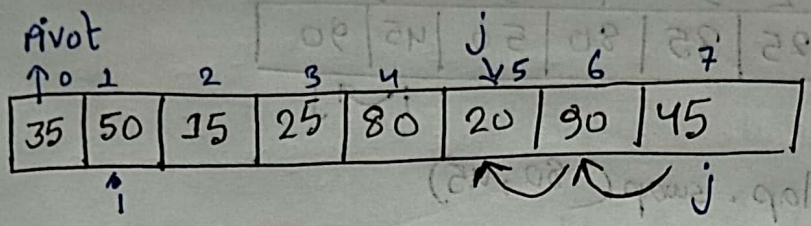
Quick sort is another efficient, comparison-based divide and conquer sorting algorithm which works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays : those less than pivot and those greater than the pivot. The sub-arrays are then recursively sorted.

Algorithm :

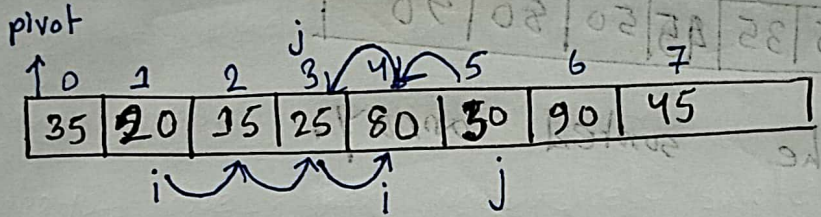
```
QuickSort (arr[], low, high) {
    if (low < high)
    pivot = Partition (arr, low, high) # Partition the array
                                         and get the pivot index.

    #Recursively sort the two halves
    QuickSort (arr, low, pivot - 1);
    QuickSort (arr, pivot+1, high)
}
```

```
Partition (arr[], low, high):
{   pivot = arr[low]      # Select pivot as 1st element
    i = low + 1           # initialize i to start after pivot
    j = high              # initialize j to the end of the array

    while i <= j :
# Find element > ~~on left~~ pivot on the left
    While i <= high and arr[i] <= pivot :
        i = i + 1
# Find element <
    While j >= low and arr[j] > pivot :
        j = j - 1

    If i < j :
        swap (arr[i], arr[j])
        i = i + 1
        j = j - 1

# swap pivot with arr[j]
    swap (arr[low], arr[j])

    return j <-  # return pivot's final position.
```

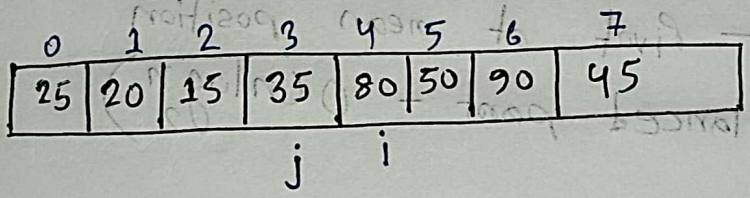
Simulation :



Now, $i < j$, so swap [50, 20]

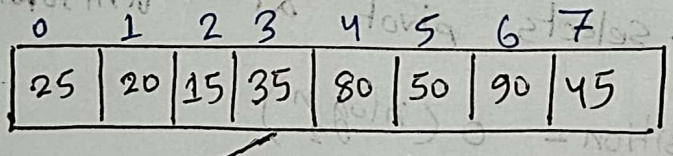


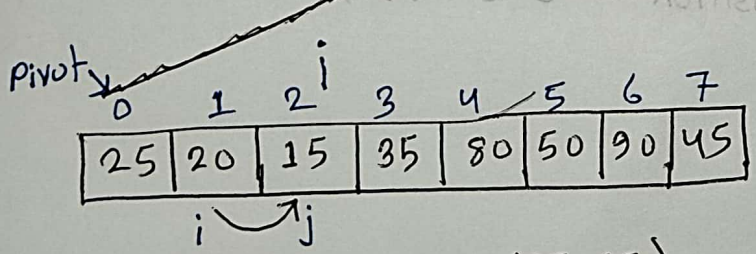
$i > j$ [overlap], ~~swap (pivot~~ swap [35, 25)



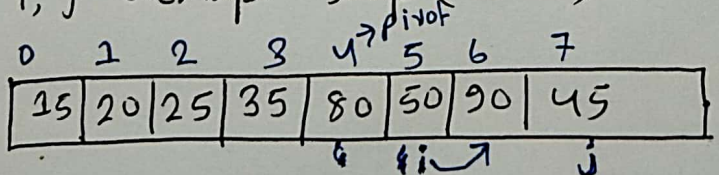
Now, Partition function will return $j = 3$ to quick sort function.

Then -

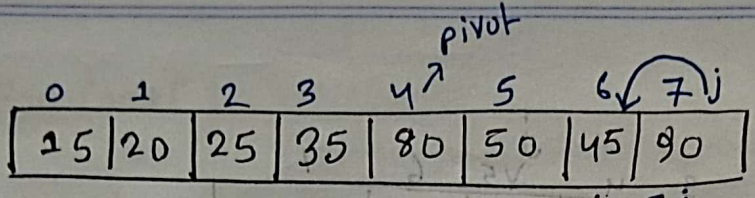




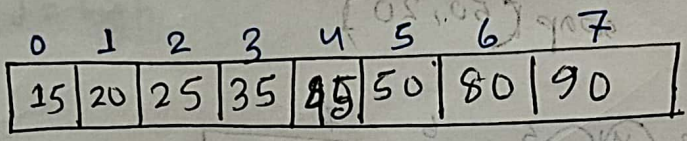
i, j overlap. swap (25, 15)



$i < j$, swap (90, 45).



i, j overlap. swap (80, 45)



which is the sorted array.

Time Complexity —

Best case — Pivot at mean position and two balanced part - $O(n \log_2 n)$

Worst case - pivot is the smallest or largest element - $O(n^2)$.

Average case - selects pivot at uniformly at random position - $O(n \log_2 n)$.

DYNAMIC PROGRAMMING

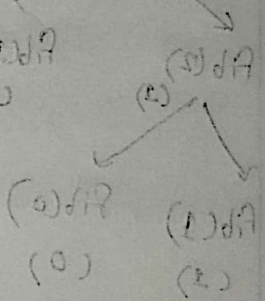
Divide and Conquer — Disjoint subproblems.

Dynamic Programming — Overlapping subproblems.

Fibonacci Series —

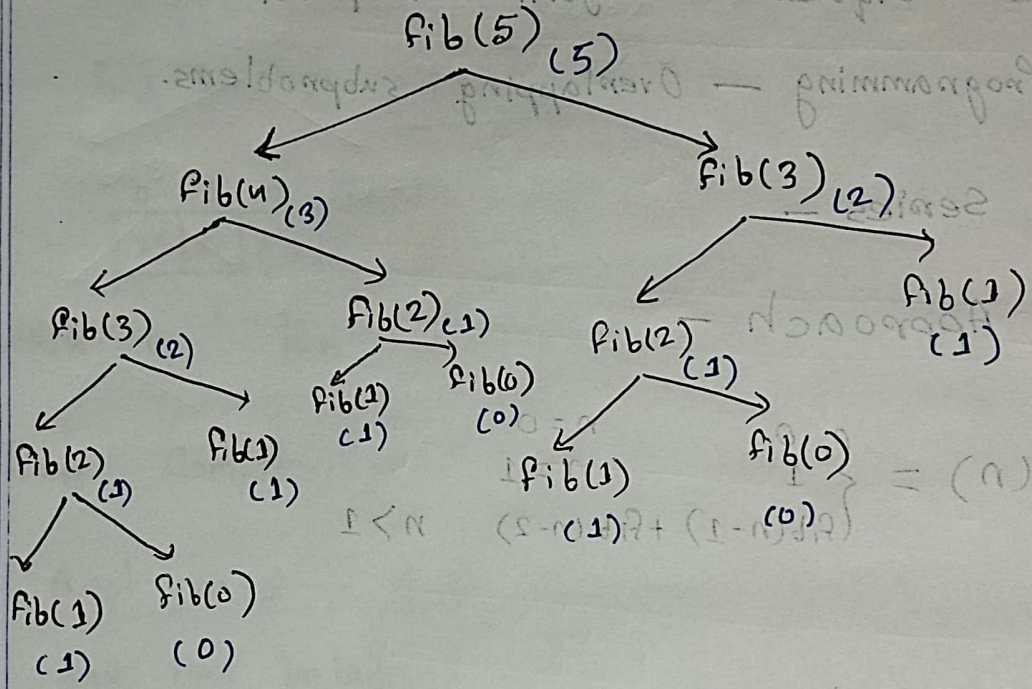
Recursive Approach —

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$



```
int fib (int n) {
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Simulation —



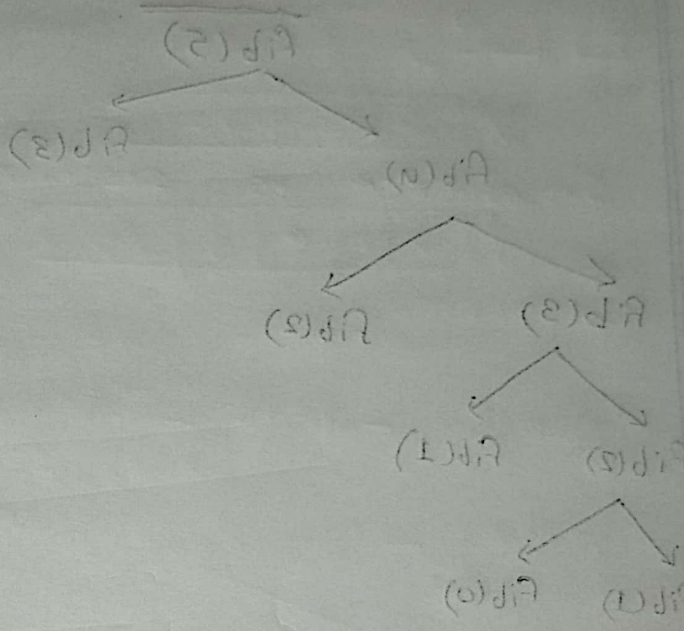
$$T(n) = T(n-1) + T(n-2) + 1$$

$$\simeq 2T(n-1) + 1$$

$a = 2$, $b = 1$; $a \geq 1$, $b \neq 1$.

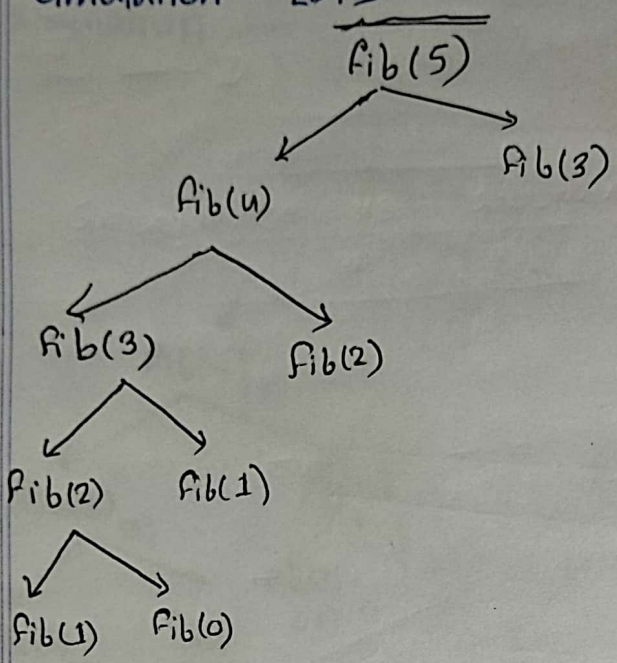
$T(n) = O(2^n)$ → Exponential complexity.

DP Approach –



```
int dp[n+1];
int fibonacci(int n)
{
   if (n == 0)
    { return 0;
    }
  else if (n == 1)
  { return 1;
  }
  else if ( dp[n] == 0)
  {
      dp[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  return dp[n];
}
```

Simulation – [DP]



Factorial —

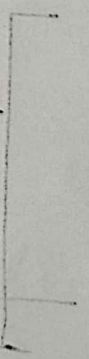
Recursive approach —

```
Factorial(n):
{
  If n==0  or  n == 1
      return 1;
  return n* Factorial(n-1).
}
```

DP Approach —

dp[n+1] # An array of size n+1 to store the results

```
Factorial(n)
{
    If n==0 or n==1
        return 1

    Else If dp[n] == 0
        dp[n] = n * Factorial(n-1)

    return dp[n].
}
```



LCS - Longest Common Subsequence

Subsequence vs substring.

"programming"

gramm - both subsequence and substring

gamng - subsequence, not substring.

* Substring এ main sequence এর same sequence এ element থাকবে।

Now,

S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

E = {2, 4, 6, 8, 10}

X_1 = {2, 4}

X_2 = {2, 4, 6}

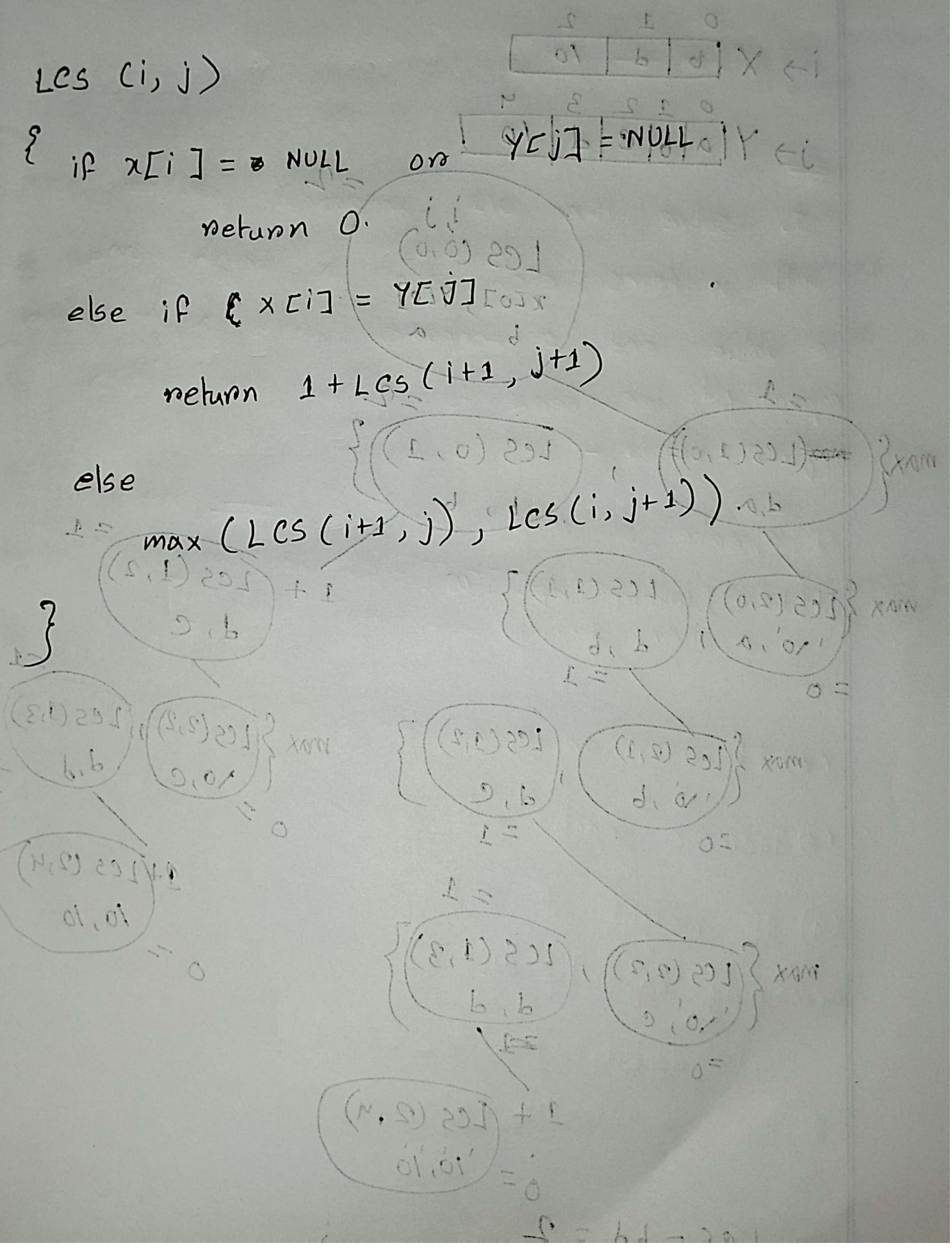
X_3 = {2, 4, 6, 8, 10}

→ Common subsequence of S and E

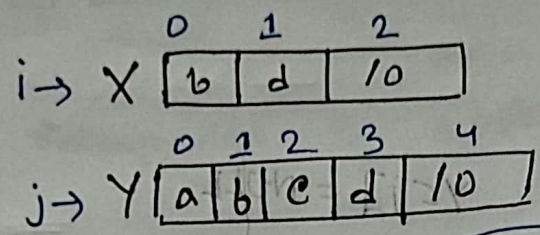
→ Longest.

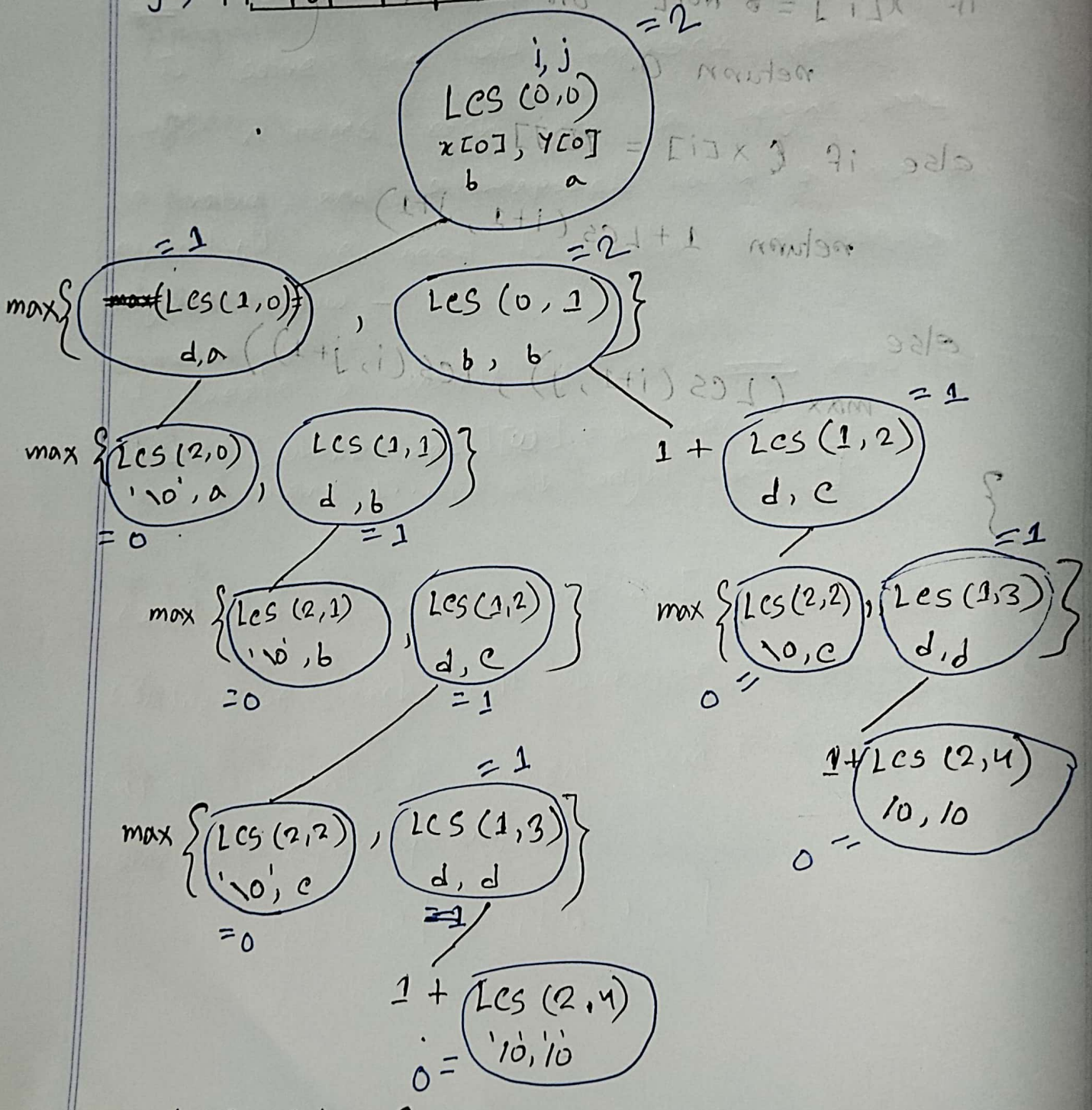
LCS - Recursive Approach -

```
LCS (i, j)
{
    if x[i] = NULL or Y[j] = NULL
        return 0.
    else if (x[i] = Y[j])
        return 1 + LCS(i+1, j+1)
    else
        max (LCS(i+1, j), LCS(i, j+1))
}
```



Simulation –





LCS = bd = 2

LCS [DP Approach] –

~~LCS (i, j)~~

To find the ~~set~~ LCS of 2 sequence, we need to follow the steps given below –

- Create a matrix dp of $m+1$ by $n+1$ size where m and n are sizes of the given sequences.
  [dp [i, j] = LCS of x_i and y_j]
- Traverse each cell row-wise or column-wise.
- If $i=0$ or $j=0$, then set $dp[i,0] = dp[0,j] = 0$
- If $x_i = y_j$, set $dp[i,j] = dp[i-1, j-1] + 1$
- If $x_i \neq y_j$, set $dp[i,j] = \max(dp[i-1,j], dp[i,j-1])$

Knapsack —

Algorithm —

To solve the 0/1 knapsack problem with n items, each with a weight and a value, and a knapsack with a maximum capacity, W, we need to follow the steps given below —

(1) Create a matrix dp of ~~size~~ n+1 by w+1. [where dp[i][w] = the maximum value of items 1 ... i with weight limit w].

(2) If i=0 or w=0, set dp[i,0] = dp[0,w] = 0.

(3) Traverse each cell row-wise or column-wise.

(4) If weight of current item i exceeds the remaining capacity w,

set dp[i, w] = dp[i_0 - 1, w].

[skip the current item and take value of previous row].

(5) If the current item can fit (weight [i-1] <= w):

~~set dp[i, w] = max(dp[i-1, w], dp[i-1, w-weight[i]]+~~

set $dp[i, w] = \max(dp[i-1, w], dp[i-1, w - w[i]] + v(i))$

[Choose the maximum between skipping the item or taking it)

(6) The value at $dp[n, w]$ will be the maximum value ~~sho~~ that can be achieved with the given capacity.