DAA

Page No.
Date / /

* Algorithm :- It is a set of operation to solve a problem performing calculation, data processing, and automated reasoning task.

⇒ Types of Algorithm :-

1. Divide & Conquer Approach
2. Greedy Technique
3. Brute Force Algorithm
4. Recursive Algorithm
5. Dynamic Programming Algorithm
6. Back Tracking Algorithm.
7. Randomized Algorithm.

(1) Divide & Conquer Approach :- It is a top-down approach. The algorithms which follows the divide & conquer techniques involves three steps :

1. Divide the original problem into a set of sub problems.
2. Solve every subproblems individually, recursively.
3. Combine the solution of subproblems into a solution of whole original problem.

⇒ Some common problem that is solved using Divide & Conquers Algorithms are Binary Search, Merge Sort, Quick sort, etc.

(2) Greedy Algorithm :- Greedy method is used to solve optimization problem.

→ An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized.

→ Greedy algorithm always makes choice looks best at the moment, to optimize a given objective.

(3) Brute Force Algorithm :- This is the basic and simplest type of algorithms.

→ The Brute force Algorithm is the straight forward approach to a problem, i.e, the first approach that comes to our mind on seeing the problem.

→ It is just like iterating every possibility ~~to~~ available to solve a problem.

(4) Recursive Algorithm :- This type of algorithm is based on the recursion.

In recursion, a problem is solved by breaking it into subproblems of the same type and calling own self again and again until the problem is solved with the help of the base condition.

→ Some common problems that is solved using recursive algorithms are Factorial of Number, Fibonacci Series, Tower of Hanoi, DFS for Graph, etc.

(5) Dynamic Programming :- It involves sequence of 4 steps:

1. Characterize the structure of optimal solution.
2. Recursively define the value of optimal solution.
3. Compute the value of optimal solution in a bottom-up minimum.
4. Construct an optimal solution from computed Information.

→ It is non-recursive.
→ It solves sub problems only once and then stores into the table.
→ It is a bottom-up approach.
→ In this, sub-problems are independent.
→ For ex., Matrix Multiplication.

⇒ Optimal Solution :- A feasible solution that minimize the objective functions.

(6) Backtracking Algorithm :- It is an algorithmic-technique for solving problems recursively by trying to build a solutions incrementally, one at a time, removing those solutions that fails to satisfy the constraints of a problem at any point of time.

→ Some common problems that is solved using Backtracking are Hamiltonian cycle, M-coloring problem, N-Queen problem, Rat in Maze problem.

→ 3 Types of problem in Backtracking :-

1) Decision Problem :- In this, we search for a feasible solution.
2) Optimization Problem :- In this, we search for a best solution.
3) Enumeration Problem :- In this, we find all feasible solutions.

(7) Randomized Algorithm :- A randomized algorithm uses a random number at least once during the computation make a decision.

→ Ex :- In Quick sort, using a random number to choose a pivot.

* Complexity of Algorithm :- It represent how many steps are required by an algorithm to solve a given problem.

→ O(f) notation represent the complexity of an algorithm, which is also termed as Asymptotic Notation or "Big Oh" notation.

Page No.
Date / /

Typical Complexity of Algorithm :-

1) Constant Complexity
2) Logarithmic "
3) Linear "
4) Quadratic Complexity
5) Cubic "
6) Exponential "

(1) Constant :- It imposes a complexity of $O(1)$.
→ It undergoes an execution of constant no. of steps like 1, 5, 10, etc. for solving a given problem.

(2) Logarithmic :- It imposes a complexity of $O(\log(N))$
→ It undergoes the execution of the order of $\log(N)$ steps.

(3) Linear :- It imposes a complexity of $O(N)$.

(4) Quadratic :- It imposes a complexity of $O(n^2)$.
→ For N input data size, It undergoes the order of N^2 count of operations on N number of elements for solving a given problem.

(5) Cubic :- It imposes a complexity of $O(n^3)$.
→ For N input data size, It execute the order of N^3 elements on N elements for solving a given problem.

(6) Exponential :- It imposes a complexity of $O(2^n)$, $O(N!)$, $O(n^k)$, ...

* Asymptotic Notation :- They are used to represent the complexity of an algorithm.

⇒ Types :-

(1) O - Big Oh
(2) Ω - Big Omega
(3) Θ - Big Theta
(4) o - Little Oh
(5) ω - Little Omega.

(1) O-Big Oh :- It specially describe worst-case scenario. It represent the upper bound running time complexity of an algorithm.

(2) Ω :- It specially describe best case scenario. It represent lower bound running time complexity of an algorithm.

(3) Θ :- It define exact asymptotic behavior in the real case scenario. It represent the average complexity.

(4) o :- we use o notation to denote upper bound that is not asymptotically tight.

(5) ω :- we use ω notation to denote lower bound that is not asymptotically tight.

Page No.
Date / /

* Space Complexity :- Space complexity of an algorithm or computer program is amount of memory space required by to solve an instance of computational problem.
→ It is a memory required by an algorithm to execute a program and produce the result.

* Time Complexity :- The time complexity is the computational complexity that describe the amount of time it takes to run an algorithm.
→ The complexity is commonly estimated by counting the no. of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.
→ Thus, the amount of time taken and no. of elementary operations performed by algorithm are taken to differ by at most a constant factor.

→ O(f) notation represent the time complexity of an algorithm, and it is also termed as Asymptotic notation or big O notation.

* Feasible Solution :- A non-negative vector of variables that satisfies the constraints of (P) is called feasible solution to a linear programming problem.

Page No.
Date / /

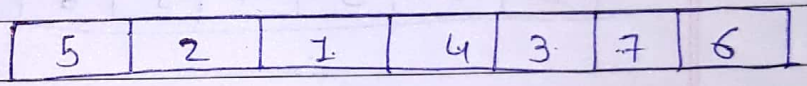
* **Bubble Sort :-** It is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. when no exchanges are required the file is sorted.

⇒ **Algorithm :-**

```
for i ← 1 to length [A] do
    for j ← length [A] down-to i+1 do
        if A [A] < A [j-1] then
            Exchange A[j] ⟷ A[j-1].
```

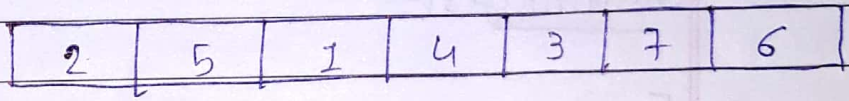
⇒ **example :-**

Unsorted list :-

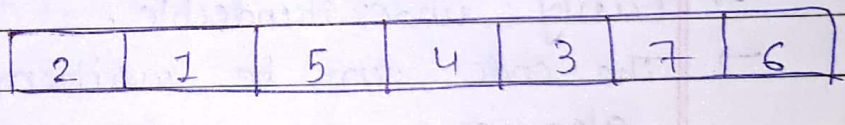


1st iteration

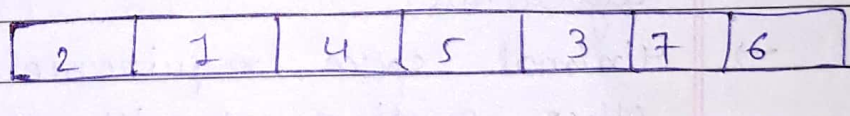
5 > 2 swap



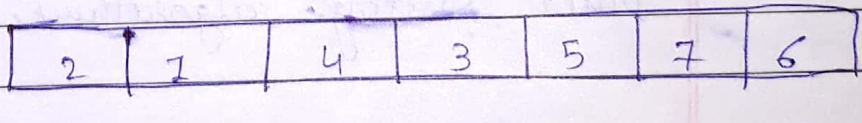
5 > 1 swap



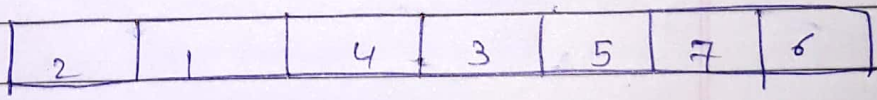
5 > 4 swap



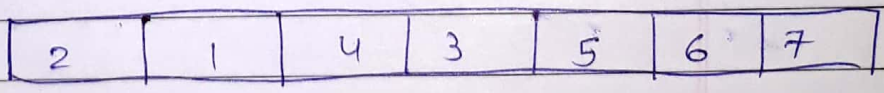
5 > 3 swap

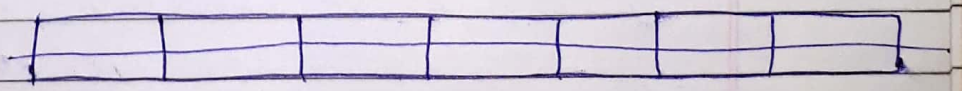


5 ~~&~~ 7 no-swap



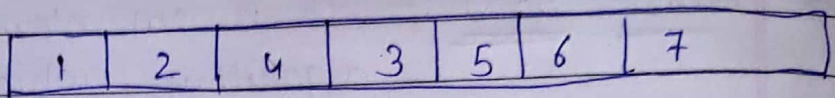
7 > 6 swap



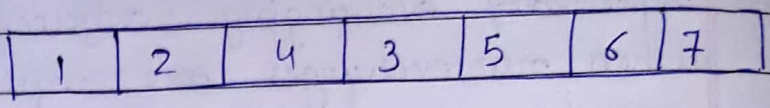


→ 2nd iteration :-

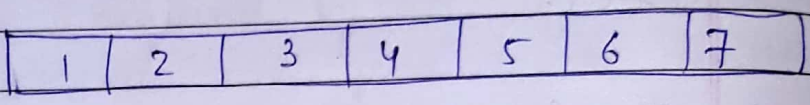
2 > 1 swap



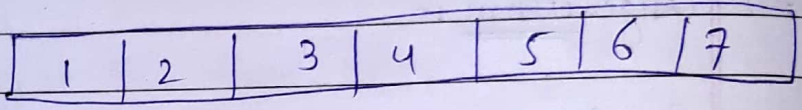
2 < 4 no swap



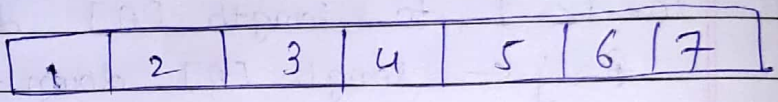
4 > 3 swap



4 < 5 no swap

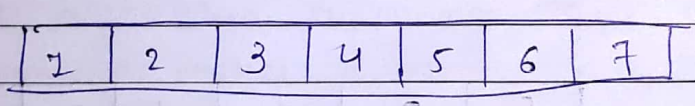


5 < 6 no swap



→ There is no change in 3rd, 4th & 5th iteration.

→ Finally, the sorted list is



⇒ Advantages :-

→ Easily understandable.

→ The code can be written easily for this algorithm.

→ Minimal space requirements than that of other sorting algorithms.

Page No.
Date / /

* Merge Sort :- It closely works as a divide and conquer paradigm.

→ It works as follows :

1. Divide :- Divide ~~the~~ the unsorted lists into two sub-~~lists~~ lists of about half the size.

2. Conquer :- Sort ~~the~~ each of the two sublists recursively until we have lists sizes of length 1, in which case list items are returned.

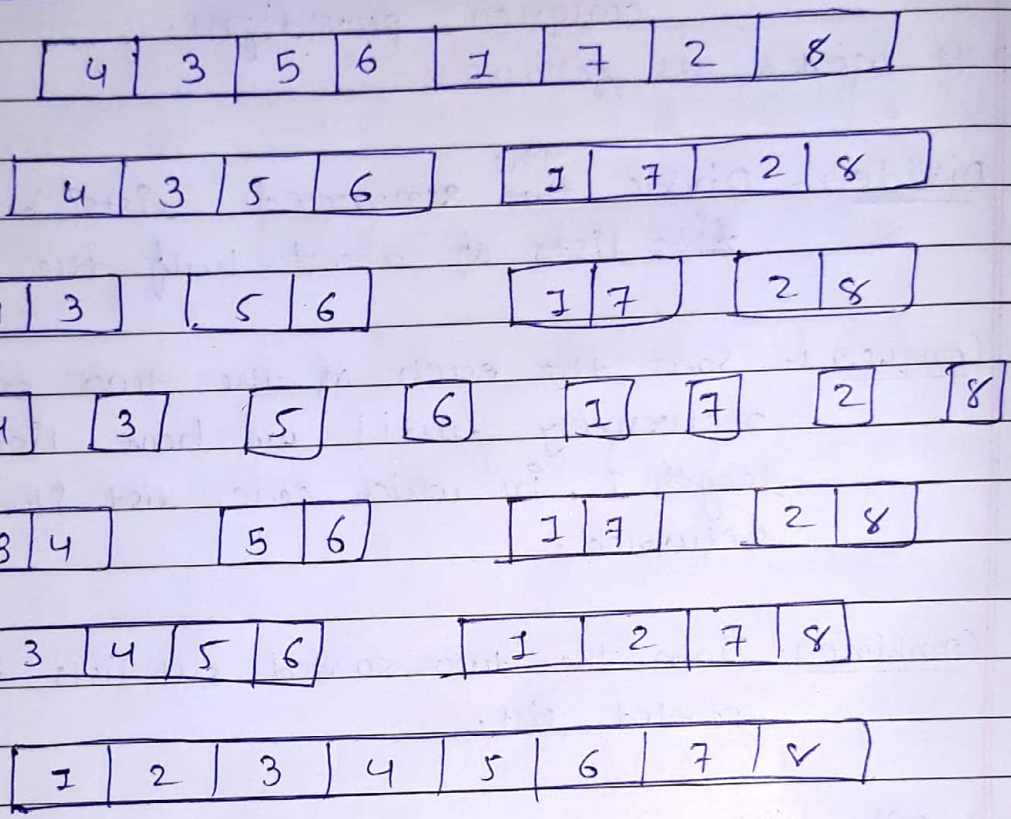
3. Combine :- Join the two sorted sub lists into one sorted list.

→ Algorithm :-

1. If $p < r$
2. Then $q \leftarrow (p+r)/2$
3. MERGE-SORT (A, p, q)
4. MERGE-SORT $(P, q+1, r)$
5. MERGE (A, p, q, r)

Page No.
Date / /

⇒ Example:-



* Insertion Sort :- It is a very simple method to sort numbers in ascending or descending order. This method follows the incremental method.
→ It can be compared with the technique how cards are sorted at the time of playing a game.
→ The numbers, which are needed to be sorted, are known as keys.
⇒ Advantage:- It is simple to implement.
→ It is efficient on small data sets.
→ It is stable.
→ It is in-place.

Page No.
Date / /

→ Algorithm :- Insertion-Sort (A)

```
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key.
```

→ Example :-

Unsorted list :-

2	13	5	18	14

- 1st iteration :

  key = a[2] = 13

  a[1] = 2 < 13

  swap, no swap

2	13	5	18	14

- 2nd iteration :

  key = a[3] = 5

  a[2] = 13 > 5

  swap 5 & 13

2	5	13	18	14

  Next, a[1] = 2 < 13

  swap, no swap

2	5	13	18	14

- 3rd iteration :

  key = a[4] = 18

  a[3] = 13 < 18

  a[2] = 5 < 18

  a[1] = 2 < 18

swap, no swap

2	5	13	18	14

→ 4th iteration :

key = a[5] = 14

a[4] = 18 > 14

swap 18 and 14

2	5	13	14	18

Next, a[3] = 13 < 14

a[2] = 5 < 14

a[1] = 2 < 14

So, no swap

2	5	13	14	18

→ finally the sorted list is

2	5	13	14	18

* Selection sort :- This type of sorting is called selection sort as it works by repeatedly sorting an elements.

→ It works as follows:

first find the smallest in the array and exchange it with the first element in the first position, then find the second element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Page No.
Date / /

⇒ Algorithm :-

```
for i ← 1 to n-1 do
    min j ← i;
    min x ← A[i]
    for j ← i+1 to n do
        if A[j] < min x then
        min j ← j
        min x ← A[j]
    A[min j] ← A[i]
    A[i] ← min x
```

⇒ example :-

- Unsorted list :-

5	2	1	4	3

- 1st iteration :-

smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1 ~~< 1~~, smallest = 1

3 > 1, smallest = 1

swap 5 and 1

1	2	5	4	3

- 2nd iteration :-

smallest = 2

2 < 5, smallest = 2

2 < 4, "

2 < 3, "

No swap

1	2	5	4	3

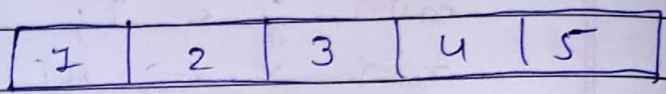
→ 3rd iteration :-

smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

swap 5 and 3

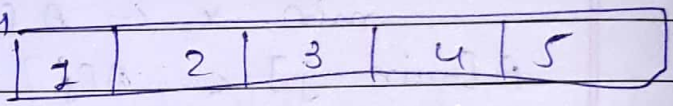


→ 4th iteration :-

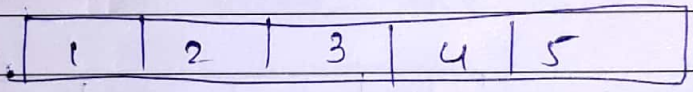
smallest = 4

4 < 5, smallest = 4

no swap



→ finally, the sorted list is



* **Quick Sort:-** It is an algorithm of Divide and Conquer Type.

- **Divide :-** Rearrange the elements ~~into two sub-arrays~~ and split arrays into two sub-arrays and an element in b/w search that each element in left sub array is less than or equal to the average element and each element in the right sub-array is larger than the middle element.
- **Conquer :** Recursively, sort two sub-arrays.
- **Combine :** Combine the already sorted array.

Page No.
Date / /

⇒ Algorithm :-

if p < r then
    q Partition (A, p, r)
    Quick-Sort (A, p, q)
    Quick-Sort (A, q+1, r)

⇒ Advantages :-

→ It is in-place
→ It require n(log n) time to sort n item.
→ It has an extremely short inner loop.

⇒ Example :-

44 33 11 55 77 90 40 60 99 22 88

→ let 44 be the pivot.
→ 22 is smaller than 44 so swap them.

22 33 11 55 77 90 40 60 99 44 88

→ 55 are greater then 44 so swap it.

22 33 11 44 77 90 40 60 99 55 88

→ Recursively, repeating steps 1 and 2 until we get two lists, one left from the pivot element 44 and one right from the pivot element.

22 33 11 40 77 90 44 60 99 55 88

→ Swap with 77:

22 33 11 40 44 90 77 60 99 55 88

→ Now elements on the right side and left side are greater than & smaller than 44 respectively.

→ Now we get two sorted lists:

22 33 11 40 44 90 77 60 99 55 88

sublist 1 sublist 2.

→ These two sorted sublist side by side.

22 33 11 40 44 90 77 60 99 55 88

11 33 22 40 44 88 77 60 99 55 90

11 22 33 40 44 88 77 60 90 55 99

1st sorted list 88 77 60 55 90 99

sublist 3 sublist 4

55 77 60 88 90 99

55 60 77 sorted

sorted

→ Merging sublists:-

11 22 33 40 44 55 60 77 88 90 99.

sorted list

Page No.
Date / /

* Steps to solve any problem with computer:-

1. Problem Definition.
2. Development of a Model.
3. Specification of an Algorithm.
4. Designing of an Algorithm.
5. Checking the correctness of an Algorithm.
6. Analysis of an algorithm.
7. Implementation of an Algorithm.
8. Program Testing.
9. Documentation.

* Qualities of an Algorithm :-

(1) Time :- To execute a program, the computer system takes some amount of time. The lesser is the time required, the better is the algorithm.

(2) Memory :- To execute a program, the computer system takes some amount of memory space. The lesser the memory required, the better is the algorithm.

(3) Accuracy :- Multiple algorithms may provide suitable or correct solution to a given problem. Some of these may provide more accurate result than others, and such algorithms may be suitable.

	BFS	DFS
1 →	Breadth First search.	Depth First Search.
2 →	BFS finds the shortest path to the destination.	DFS goes to the bottom of subtree, then backtracks.
3 →	BFS traverses according to tree level.	DFS traverses according to tree depth.
4 →	BFS is implemented using FIFO List.	It is implemented using LIFO list.
5 →	It require more memory.	It require less memory.
6 →	It gives the shallowest path solutions.	It doesn't guarantee shallowest path solutions.
7 →	There is no need of backtracking in Bfs.	There is a need of backtracking in DFS.
8 →	You can never be trapped into finite loops.	You can be trapped into infinite loops.

* Prim's algorithm :- It is a greedy approach used to find a minimum spanning Tree.

→ In this algorithm, to form an MST we can start from an arbitrary Vertex.

Page No.
Date / /

⇒ Algorithm: MST - Prim's (G, w, r)
for each u ∈ G.V
u.key = ∞
u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
u = Extract - Min (Q)
for each v ∈ G.adj[u]
if each v ∈ Q and w(u,v) < v.key
v.π = u
v.key = w(u,v)

* Dijkstra's Algorithm :- Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph G = (V, E), where all the edges are non-negative.

⇒ Algorithm: Dijkstra's-Algorithm (G, w, s)
for each vertex v ∈ G.V
v.d := ∞
v.π := NIL
s.d := 0
S := ∅
Q := G.V
while Q ≠ ∅
u := Extract-Min (Q)
S := S ∪ {u}
for each vertex v ∈ G.adj[u]

Page No.
Date / /

if $v.d > u.d + w(u,v)$
$v.d := u.d + w(u,v)$
$v.\pi := u$

* Kruskal's Algorithm :- An algorithm to construct a minimum spanning Tree for connected weighted graph. It is a greedy algorithm.

→ The greedy choice is to put the smallest weight edge that does not bcz the cycle in the MST constructed so far.

→ If a graph is not linked, then it finds a Minimum Spanning Tree.

⟹ Steps for finding MST using Kruskal's Algo :-

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G, and proceeding sequentially add edge which does not result in a cycle, until (n-1) edges are used.
3. EXIT.

⟹ MST-KRUSKAL (G, w)
1. $A \leftarrow \emptyset$
2. for each vertex $v \in V[G]$
3. do MAKE-SET (v)
4. Sort edges of E into non-decreasing order by weight w.
5. for each edge $(u,v) \in E$, taken in non-decreasing order by weight.

Page No.
Date / /

6. do if FIND-SET(u) ≠ if FIND-SET(v)
7. then A ← A ∪ {(u,v)}
8. UNION(u,v)
9. Return A

* Rod Cutting :- A rod is given of length n. Another table is also provided, which contains different size and price for each size. Determine the maximum price by cutting rod & selling them in the market.

→ To get the best price by making a cut at different positions and comparing the prices after cutting the rod.

→ Let the f(n) will return the max possible price after cutting a row with length n.

→ we can simply write the function f(n) like this f(n) := maximum value from price[i] + f(n-i-1), where i is in range 0 to (n-1).

⇒ Input and Output :-

→ Input: The price of different lengths, and the length of rod. Here the length is 8.

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

→ output :- Maximum profit after selling is 22.
- cut the rod in length 2 and 6.
- The profit is 5 + 17 = 22.

Page No.
Date / /

⇒ Algorithm :-

rodCutting (price, n)

- Input :- Price list, number of different prices on list.
- output :- Maximum profit by cutting rods.

* Closest pair of Points Problem :-

→ In this problem, a set of n points are given on 2D plane.
→ In this problem, we have to find the pair of points, whose distance is minimum.
→ To solve this problem, we have to divide the points into two halves, after that smallest distance b/w two points is calculated in a recursive way.
→ Using distances from a middle line, the points are separated into some steips.
→ we will find the smallest distances from the stripe array.
→ At first two lists are created with data points, one list will hold the points, which are sorted on x values, and another will hold data points, sorted on y values.
→ The time complexity of this algorithm will be O(n log n).

Page No.
Date / /

⇒ Algorithm :-

→ findMinDist (pointsList, n)
- Input :- Given point list & no. of points in the list.
- output :- Finds minimum distance from two points.

→ stripClose (strips, size, dist)

- Input :- Different points in the strip, no. of points, distance from the midline.
- output :- closest distance from the two points in the strip.

→ findClosest (xSorted, ySorted, n)

- Input :- points sorted on x values, and points sorted on y values, no. of points.
- output :- Find minimum distance from a total set of points.

* Binary search :- Binary search is performed on a sorted array.
→ In this approach, the index of element x is determined if the element belongs to the list of elements.
→ If the array is unsorted, linear search is used to determine the position.

→ In this algorithm, we want to find whether element x belongs to the set of numbers stored in an array numbers[].

→ where l and r represent the left and right index of a sub-array in which searching operation should be performed.

Algorithm

Algorithm : Binary-Search (numbers[], x, l, r)
if l = r
    return l
else
    m := ⌊(l+r)/2⌋
    if x ≤ numbers[m] then
        return Binary-Search (numbers[], x, l, m)
    else
        return Binary-Search (numbers[], x, m+1, r)

⇒ Binary search produces the result in O(log n) times.

⇒ Let T(n) be the no. of comparisions in worst-case in an array of n elements.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

- Using this recurrence relation T(n) = log n.
- Therefore, binary search uses O(log n) time.