Data structure **Sem II Batch A** **By . Rupali Jadhav.**
Academic year 2016-17

Q. What is Data Structure?

- Structure – set of rules that hold the data together.
- If we take a combination of data and fit them into a structure such that we can define its relating rules, we have made a data structure.

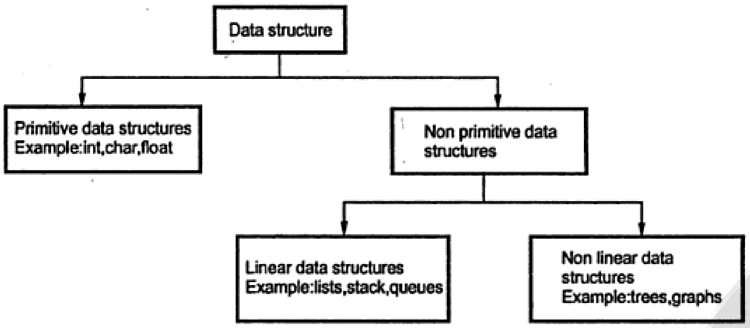


Fig. Types of data structure

- **Primitive data structure:** theses are the basic data types such as integer , float , character .
- **Non Primitive data structure:** these are the data structure which are basically derived from primitive data structure.
  - **Linear data structure:** The data structure in which data are arranged in list or in straight sequence. For eg; array and linked list ,queue, stack.
  - **Non linear data structure:** The data structure in which data are arranged in hierarchical manner. For eg; tree and graph.

Q. What is algorithm?

- An algorithm is a well defined, finite step-by-step procedure to achieve a required result. It has following properties:-

1. Input
   - Zero or more values
2. Output
   - At least one value
3. Definiteness
   - Each instruction is precise and unambiguous.
4. Finiteness
   - Should terminate after finite number of steps
5. Effectiveness
   - Every instruction should be basic enough

Q. Algorithmic efficiency:

- More than one algorithms for solving one problem.
- One algorithm will be efficient than others.
- If function is linear, efficiency depends on number of instructions.
  efficiency = f(n) where n = number of instructions
- **Linear loops**

```
for(i=0;i<1000;i++)
{
        code
}
```

  - n = Loop factor =1000
  - Number of iterations are directly proportional to loop factor
  - f(n) = n

- **Linear loops**

```
for(i=0;i<1000;i=i+2)
{
        code
}
```

  - n = Loop factor =1000
  - Number of iterations are directly proportional to half the loop factor
  - f(n) = n/2

- **Nested loops**

  Iterations = outer loop iterations * inner loop iterations

- **Quadratic loop**

```
for(i=1;  i <= n;  i++)
{
                for(j=1;  j <= n;  j++)
                {
                        code
                }
}
```

  $f(n) = n^2$

- **Dependent Quadratic loop**

```
for(i=1;  i <= n;  i++)
{
        for(j=1;  j <= i;  j++)
        {
                code
        }
}
```

- f(n) = n (n+1) / 2

Space complexity:

- Amount of memory needed by program upto completion of execution.
- Space needed by program
  - Instruction space
  - Data space
    - Space needed by constants,
    - variables,
    - fixed sized structured variables,
    - dynamically allocated space

- Environmental stack space
    - Return address
    - Values of local variables

Time Complexity:

- Amount of time program needs to run upto the completion.
- Time complexity varies system to system.

Steps

- Count all sort of operations performed in algorithm.
- Know the time required for each operation.
- Compute the time required for execution of algorithm.

Execution time physically clocked
Count no. of operations

```
algo sum()
  {   s=0              --------- 1
      for i= 1 to n     ---------  n + 1
         s=s+a[i]      ----------  n
      return s         ----------  1
  }                         2n + 3
```

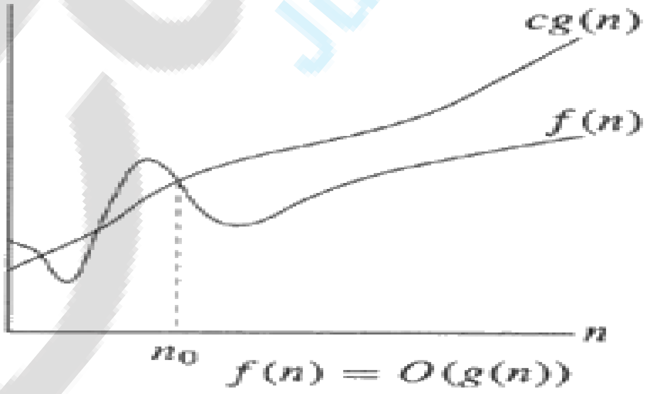
Q. What is analysis of algorithm? Explain various notations used while analyzing an algorithm. (Big O, omega, theta notation)

- Asymptotic analysis is how we study the behavior of algorithms as their input approaches infinite amounts.
- In particular, we are looking at how the input size affects the running time of our algorithms by analyzing their time complexity.

O - Big Oh
Ω - Omega
Θ - Theta

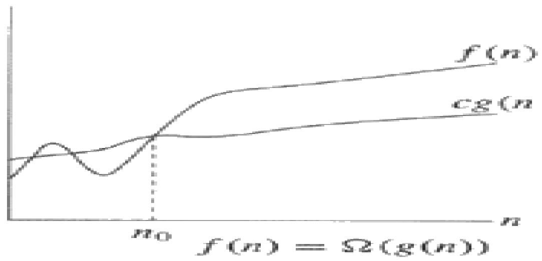
O-Notation (Upper Bound)

- ✓ Def - Given functions f(n) and g(n), we say that f(n) is O(g(n)) if and only if there are positive constants c and n_0 such that f(n)≤ c g(n) for n ≥ n_0.
- ✓ g(n) should be as small as possible.



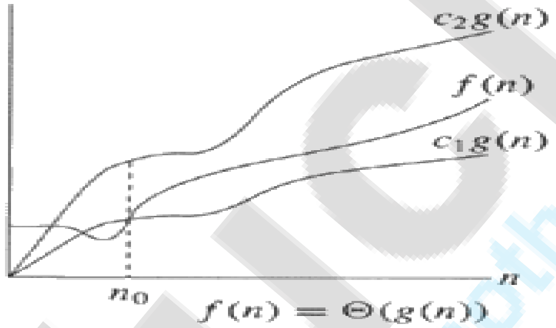
Ω-Notation (Lower Bound)

- Def - Given functions $f(n)$ and g(n), we say that $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$; that is, there exists positive constants c and n_0 such that $f(n) \geq c\, g(n)$ for $n \geq n_0$.
- g(n) should be as large as possible. Gives best case running time



Θ-Notation (Same order)

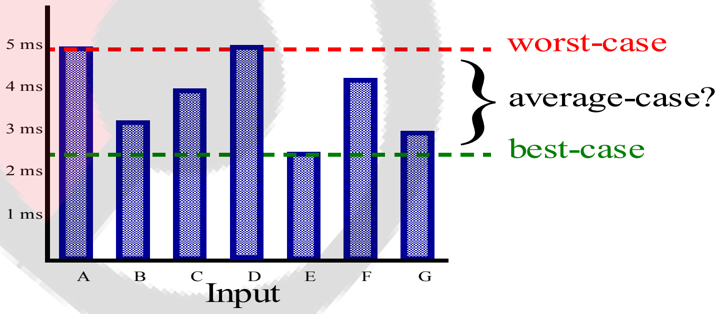
- We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive i.e. g(n) is both upper and lower bound of f(n).



Q. Explain Best , average,worst case time complexity with algorithm?

The type of input also affects the running time:

- **Best-case analysis**:- based on “ideal” input
- **Worst-case analysis**:- based on worst possible input
- **Average-case analysis**:- based on the average outcome of running an algorithm many times over random input



- Best case input
  - With this input algorithm takes shortest time to execute.
  - E.g. for searching algorithm, number we search is found at the first place itself
- Worst case input
  - With this input algorithm will take most time to execute.
  - E.g. for searching algorithm, number we search is found at the last place itself
- Average case input
  - With this input algorithm delivers average performance.
  - A(n) = ∑ Pi ti for i=1 to m
  - n = size of input
  - m = number of groups
  - pi = probability that input will be from group i
  - ti = time that algorithm takes for input from group i

Sorting

Algorithm:

- Bubble sort: Sort by comparing each adjacent pair of items in a list in turn, swapping the items if not in order, and repeating the pass through the list until no swaps are done.

```
Algorithm bubble (a, n)

Pre: Unsorted array a  of length n.

Post: Sorted array in ascending order of length n

    for i = 1 to (n – 1) do                    // n-1 passes
      for j = 1 to (n- i ) do
           if ( a[j] > a[j+1] )
                1. temp=a[j]          //swapping of numbers
                2. a[j]=a[j+1]
                3. a[j+1]=temp
```

- Q. Write an algorithm to sort the element using bubble sort. Also sort the following Sort the following numbers using bubble sort.

  25 14 62 35 69 12

  Pass 1:

---

25 **14** 62 35 69 12

14 **25** **62** 35 69 12

14 25 **62** **35** 69 12

14 25 35 **62** **69** 12

14 25 35 62 **69** **12**

14 25 35 62 12 **69**

Number of comparisons = 5

Pass 2

14 **25** 35 62 12 69

14 **25** **35** 62 12 69

14 25 **35** **62** 12 69

14 25 35 **62** **12** 69

14 25 35 12 **62** **69**

Number of comparisons = 4

Pass 3

14 **25** 35 12 62 69

14 **25** **35** 12 62 69

14 25 **35** **12** 62 69

14 25 12 **35** **62** **69**

Number of comparisons = 3

Pass 4

14 **25** 12 35 62 69

14 **25** **12** 35 62 69

14 12 **25** **35** **62** **69**

Number of comparisons = 2

Pass 5

14 **12** 25 35 62 69

12 **14** **25** **35** **62** **69**

Number of comparisons = 1

Number of elements = 6

Number of pass = 5

Number of comparison in any pass

= n – pass number

Bubble Sort – Optimized:

```
Algorithm bubble (a, n)

Pre: Unsorted array a  of length n.

Post: Sorted array in ascending order of length n

1.  for i = 1 to (n – 1) do                // n-1 passes
        1.  test = 0
        2.  for j = 1 to (n– i ) do
                1.  if ( a[j] > a[j+1] )
                        1.  temp=a[j]
                        2.  a[j]=a[j+1]
                        3.  a[j+1]=temp
                        4.  test = 1            // exchange happened
2.  if (test = 0)   //no exchange - list is now sorted
        return
```

Time Complexity:

During 2nd pass we perform n-2 comparisons

Comparisons = (n-1)+(n-2)

Continuing like this we get

Comparisons = (n-1)+(n-2)+ …………….. + 1

$= n(n-1)/2$

$= n^2/2 - n/2$

$= O(n^2)$ since <u>Largest power is 2</u>

Therefore Worst case complexity = $O(n^2)$

If list is already sorted then no. of pass = 1 and no. of comparisons = n-1

Best case complexity = O(n)

Q.sort the following element bubble sort. 7 8 26 44 13 23 98 57

Q.Sort following elements using bubble sort method.20 3 17 19 25 35 9 42 16 27

Insertion Sort:

Sort by repeatedly taking the next item and inserting it into the final array in its proper order with respect to items already inserted.

Suppose array A with n elements A[1], A[2], A[3],…..,A[n]

Pass 1: A[1] is already sorted.

Pass 2: A[2] is inserted before or after A[1] such that A[1],A[2] is sorted array.

Pass 3: A[3] is inserted in A[1], A[2] in such a way that A[1],A[2],A[3] is sorted array.

Pass N: A[N] is inserted in A[1],A[2],A[3]…A[n-1] in such a way that A[1],A[2],A[3]…A[n-1], A[n] is sorted array.

This algorithm inserts A[k] into its proper position in the previously sorted sub array A[1], A[2],…,A[k-1]

Algorithm:

Algorithm insertion (a, n)

Pre: Unsorted list a of length n.

Post: Sorted list a in ascending order of length n

```
1. for i = 1 to (n -1)  do                // n-1 passes
       1. temp = a[i]                     //value to be inserted
       2. ptr = i – 1                     //pointer to move downward
       3. while ( temp < a[ptr]   and   ptr >= 0)
              1. a[ptr + 1] = a[ptr]
              2. ptr = ptr - 1
       4. a[ptr +1] = temp
```

Time complexity:

- Best Case: - O (n)

List is already sorted. In each iteration, first element of unsorted list compared with last element of sorted list, thus (n-1) comparisons.

- Worst Case: - O(n^2)

List sorted in reverse order. First element of unsorted list compared with one element of sorted list, second compared with 2 elements. Last element to be inserted compared with all the n-1 elements.

1 + 2 + 3 + ………………… (n-2) + (n-1)

= (n (n-1))/2

= O (n^2)

- Average Case: - O(n^2)

Q.Sort the following array using insertion sort.252 10 5 8 7

25 2 10 5 8 7

25 First value is considered as sorted.

Pass 1:

2 25

Insert next value 2

2 is less than 25

25 slides over

Pass 2:

2 10 25

Insert next value 10

10 is less than 25

25 slides over

Pass 3:

2 5 10 25

Insert next value 5

5 is less than 10, 25

10, 25 slide over

Pass 4:

2 5 8 10 25

Insert next value 8

8 is less than 10, 25

10, 25 slide over

Pass 5:

2 5 7 8 10 25

Insert next value 7

7 is less than 8,10, 25

8,10, 25 slide over

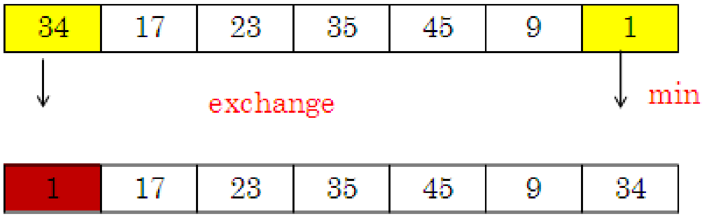
Q.Sort the following element using insertion sort 24 13 9 64 7 23 34

Q. Sort following elements using insertion sort algorithm.78 1234 98 22 65 11

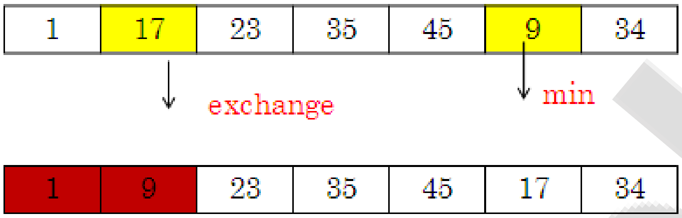
Selection sort:

- Find the first smallest element in the list and place it at the first position.
- Find next smallest number and place it at the second position.
- And so on…

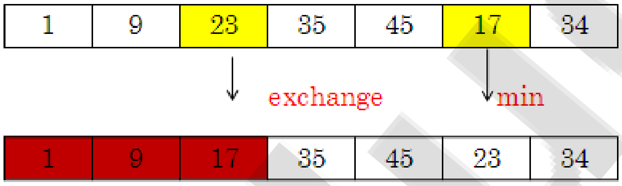
Pass 1:



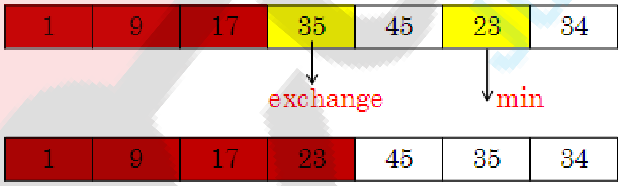
Pass 2:



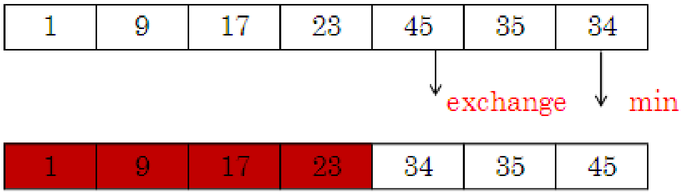
Pass 3:



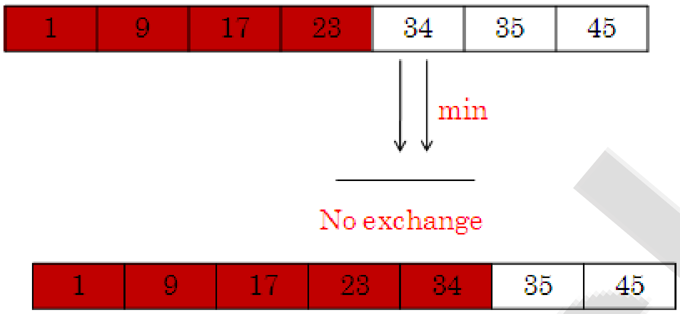
Pass 4:



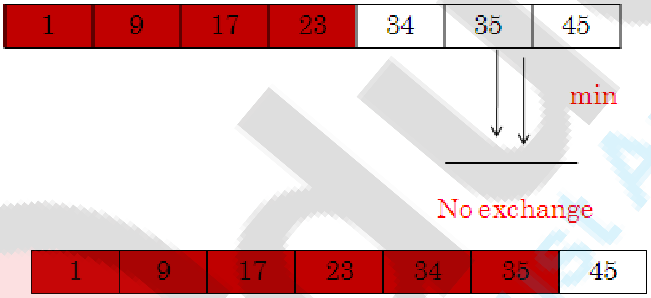
Pass 5:



Pass 6:



Pass 7:



Number of elements = n
Number of pass = n - 1

Algorithm:

```
Algorithm selection (a, n)
Pre: Unsorted array a of length n.
Post: Sorted list in ascending order of length n
    1.   for i = 0 to (n -2)  do                    // n-1 passes
```

```
1.  min_index=i
2.  for j = (i+1) to (n -1) do
        1. if ( a[min_index] > a[j] )
            1. min_index = j
3.  if (min_index != i) //place smallest element at ith place
        1. temp= a[i]
        2. a[i]=a[min_index]
        3. a[min_index]=temp
```

Complexity of algorithm:

Worst case and best case complexity:

- No. of comparisons in 1st pass = N – 1
- No. of comparisons in 2nd pass = N – 2
- No. of comparisons in 3rd pass = N – 3
- No. of comparisons in (N – 1) pass = 1

$f(n) = (n – 1) + (n – 2)+ \ldots.+ 1$

$= n(n-1)/2 = O(n^2)$

Q. Sort following elements using selection sort method of sorting

29 83 26 74 95 28

Shell sorting:

- ✓ It is improvement over simple insertion sort.
- ✓ This method sorts separate subfiles of the original file.
- ✓ This subfile contain every kth element of the original file.
- ✓ The value of k is called increment or span.

E.g. if k = 5 then following subfiles are created.

Subfile 1: x[0] x[5] x[10]

Subfile 2: x[1] x[6] x[11]

Subfile 3: x[2] x[7] x[12]

Subfile 4: x[3] x[8] x[13]

Subfile 5: x[4] x[9] x[14]

All subfiles are sorted files.

Algorithm shell (a, n, inc, n_inc)

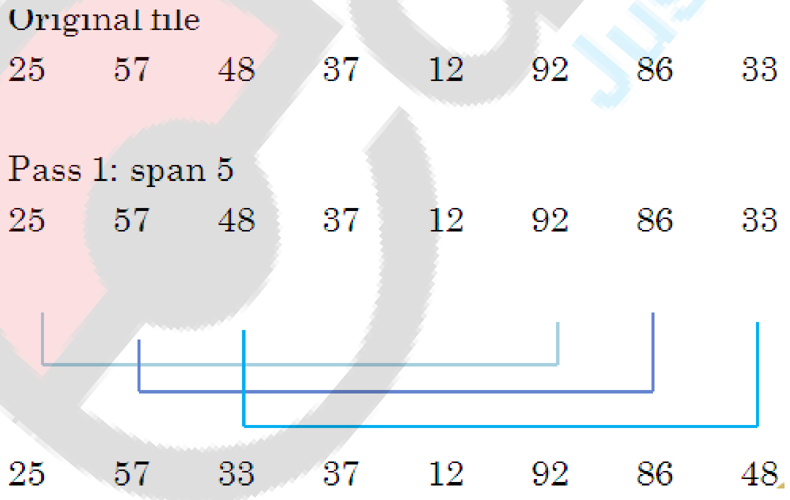
// a – unsorted array, n – size of array, inc – array storing increment values, n_inc - size of array increments

Pre: Unsorted list of length n.

Post: Sorted list in ascending order of length n

```
1. for(increment=0;  increment < n_inc;  increment++)
        //span is the size of increment
        1. span = inc[increment]
        2. for(j = span; j<n ; j++)
                //inserts element a[j] into its proper position within
                subfile
                // sorting
                1. y = a[j]
                2. for(k = j-span; (k>=0 && y<a[k]); k= k-span)
                        1. a[k+span] = a[k]
                3. a[k+span] = y;
```

Example:



Original file

25 57 48 37 12 92 86 33

Pass 1: span 5

25 57 48 37 12 92 86 33

25 57 33 37 12 92 86 48

25 57 33 37 12 92 86 48

Pass 2: span 3

25 57 33 37 12 92 86 48

25 12 33 37 48 92 86 57

---

25 12 33 37 48 92 86 57

Pass 3: span 1

25 12 33 37 48 92 86 57

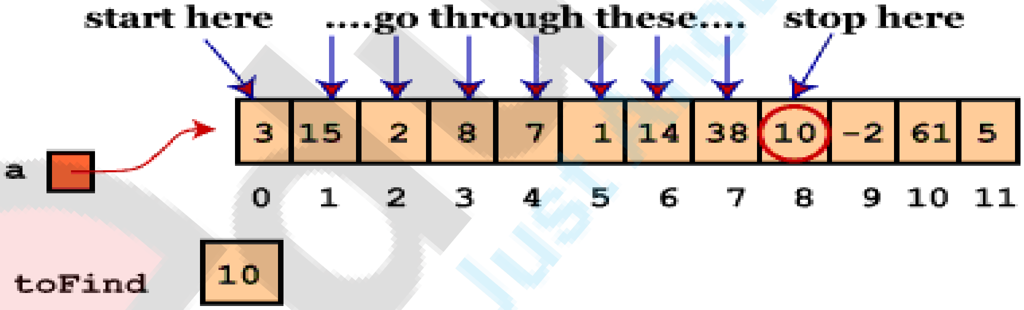
12 25 33 37 48 57 86 92

- **Complexity : O($n^{1.5}$) empirically proved.**

Searching Techniques

Q. Explain linear search with example. Also write algorithm for linear search?

- It is process of checking and finding an element from list of elements . The algorithm searches key by comparing it with each element in turn.



Algorithm:

```
Algorithm linear (a, n, key)
// key is data to be searched in array a of size length
Pre:    Unsorted list of length n.
Post:   If found, return position of key in array a. If key not present in list, return negative value
    1.  for i = 0 to (n - 1)  do
          if  (key == a[i])
                return i
    2.  return -1
```

Complexity:

- Best Case: - O (1)

  - Item found at first position.
- Worst Case: - O(n)
  - Item found at last position.
- Average Case: - O(n)
  - On an average (n+1)/2 comparisons required

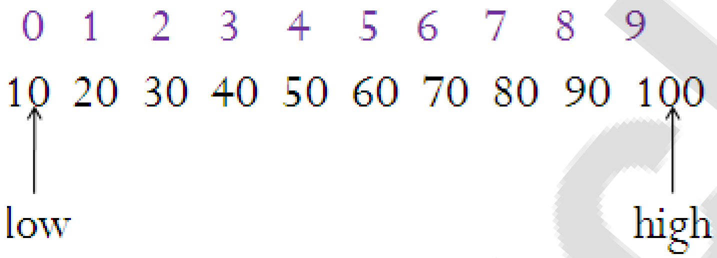
Q. **Explain Binary search with example. Also write algorithm for linear search**?

Working:

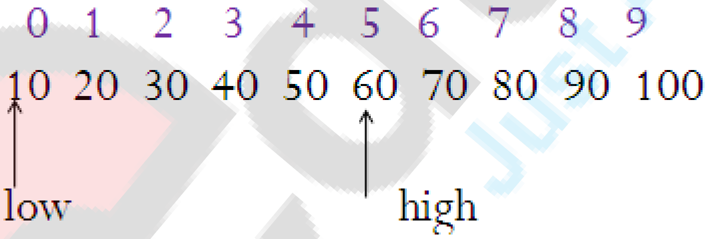
- When array is not sorted, sequential search is only the option for sorting.
- But if array is sorted binary search is more efficient algorithm.
- It starts with the testing of data at the middle of an array.
- Target may be in first half or second half of the array.
- To find middle of the list, beginning of the list and end of the list are used.

Example:

- Search key = 80 in given array. 10 20 30 40 50 60 70 80 90 100



low = 0
high = 9
mid = (low + high) / 2 = 4
a[mid] = 50
80>50, low = mid + 1 = 4+1 = 5



low = 5
high = 9
mid = (low + high) / 2 = 7
a[mid] = 80 //found

algorithm:

Algorithm binary_search (a, n, key)
// key - data to be searched in array a of size n
Pre: Sorted list of length n.
Post: If present, return position of key in array a; Else return -1

```
1.  low =0
2.  high = n-1
3.  while (low <= high)
            1.   mid = (low + high)/2
            2.   if (key  == a[mid])
                     return mid
            3.  if ( key  <  a[mid])
                     high = mid -1
            4. else
                     low = mid + 1
4.  return -1
```

complexity:

- Complexity: O ($\log_2 n$)
- Each comparison in binary search reduces the number of possible elements by factor of 2.
- Say array size $n = 2^m$, then number of passes = m
- When $n = 2^m$, thus $m = \log_2 n$. (e.g. if n=64, m = 6)
- During each pass, maximum only 2 operations will be required (one to check equality & if that fails another to check in which part of list is the key to be further searched)
- Thus 2m comparisons i.e. $2 \log_2 n$. Thus O($\log_2 n$)

Q. Given a target value, perform Binary Search on an array of numbers
11 22 30 33 40 44 55 60 66 77 80 88 99

A) Search key = 40
B) Search key = 85

Q. write recursive binary search algorithm.

```
BinarySearch(a,  key,  low,  high)
    1.   if (low > high)
         return -1                        // not found
    1.  mid = (low + high) / 2
    2.  if (key < a[mid] )
         return BinarySearch ( a, key, low,  mid-1)
    1.  else if (key > a[mid])
         return BinarySearch ( a, key, mid+1,  high)
    1.  else
         return mid                       // found
```

Q. **Difference between linear and Binary search**

	Linear search	**Binary Search**
1	Data may be any order	Data must be in sorted order

2	Time complexity O(n)	O(log n)
3	Access is slower	Access is faster
4	Search works by looking each element in list until either it finds the target or reaches the end.	As input data is sorted we can leverage this information to decrease the number of item we need to look at to find our target. We know that if we look random item in the data and that item is greater than our target , then all items to the right of that item will also be greater than our target . This means that only need to look at left part of the data. Each time we search for the target and eliminate half of the remaining item