**Algorithms**

**1 . Searching**

1. **Linear Search**

* It is also known as sequential search

Approach :-

* Start from the leftmost element of the array and one by one compare the key(to find) with each element of the array .
* If the key matches with any element return the index
* In any other case of not finding return -1

*Algorithm linearSearch(A , key , n)****(iterative)***

*Input : A – the array with number of n elements*

*key – the element to search*

**Improve Linear Search Worst-Case Complexity**

1. if element Found at last  O(n) to O(1)
2. if element Not found O(n) to O(n/2)

**Head to GFG and for better Time-complexity explanation head to the class notes**

*1 . for i <- 0 to n do*

*2 . if A[i] = key then*

*3 . return i*

*4 . end if*

*5 . return -1*

***Time-comlexity = O(n) (Worst-case and Average Case)***

***Recursive approach***:-

- We will pass 4 arguments to the recursive function from the main , array , 1st index , last index and the key

- Compare x in 1st and the last element of the array and if found return the indices .

- Else call the recursive function by passing the decreamented last by 1 index and increamented first by 1 index .

- we will call the recursive function till r >= l , else will return -1 .

**int linearSearchRecursive(int \* a , int l ,int r ,int x)**

{

if(r<l)

return -1 ;

if(a[l]==x)

return l ;

if(a[r]==x)

return r ;

return linearSearchRecursive(a , l+1 , r-1 , x) ;

}

1. **Binary Search**

Approach :-

- get the middle index(mid) of the whole array first

- Compare item with the middle element , if it matches return the middle index

- else if item is greater than middle element , it means item lies on the right half subarray , so

we reset lower-bound(l) to mid + 1 and begin the search again

- else if , item < mid element , it must be at left half sub-array , and we will reset upper-bound(r)

to mid-1.

- If element is not found return -1 .

**int binarySearch(int \*a , int l , int r ,int item)**

{

int mid ;

while(l<=r)

{

mid = (l+r)/2 ;

if(item == a[mid])

return mid ;

if(item < a[mid])

r = mid-1 ;

else

l = mid+1 ;

}

return -1 ;

}

***Average Time-complexity : O(log2n)***

**int binarySearchRecursive(int \*a , int l , int r ,int item)**

{

int mid ;

if(l<=r)

{

mid = (l+r)/2 ;

if(a[mid] == item)

return mid ;

if(a[mid] < item)

return binarySearchRecursive(a , mid+1 , r , item);

return binarySearchRecursive(a , l , mid-1 , item);

}

return -1 ;

}

**2 . Sorting**

1. **. Selection Sort**

**-** This algorithm sorts an array , by repeatedly finding the minimum element from unsorted part an putting it at the beginning

- The algorithm maintains two subarrays in a given array

1 . The array which is already sorted

2 . Remaining array which is unsorted

Approach:-

- Find the minimum element in the unsorted array and swap it with element at beginning of that unsorted array OR

- In every iteration of selection sort , the minimum element from the unsorted subarray is picked and moved to the sorted subarray

***void selectionSort(int \* array , int n)*** *// n is the length of the array*

*{*

*int i , j , temp ;*

*for(i=0 ; i<n-1 ; i++)*

*{*

*for(j= i + 1 ; j < n ; j++)*

*{*

*if(a[j]<a[i])*

**Divide and Conquer Paradigm**

Steps involved to solve a particular

1 . Divide : Break the given problem into subproblems of same type

2 . Conquer : Recursively solve these subproblems

3 . Combine : Appropriately combine the answers

*{*

*temp = array[i] ;*

*array[i] = array[j] ;*

*array[j] = temp ;*

*}*

*}*

*}*

*print(array)*

*}*

***Time-complexity = O(n2)***

**4) . Merge Sort**

**-** It uses Divide and Conquer technique

- array will be recursively divide into two halves till the size becomes 1

- then , the merge action takes place and start merging arrays back , till the complete array is merged

Approach:-

- get the middle point index (m) to divide the array into two subarrays

- call mergeSort() for the first half (from l to m)

- call mergeSort() for the second half(from m+1 to r)

- merge the two halves in sorted manner by calling merge()

**void mergeSort(int \*a , int l , int r)**

{

int m ;

if(l<r)

{

m = (l+r)/2 ;

mergeSort(a , l , m);

mergeSort(a , m+1 , r);

merge(a , l , m , r) ;

}

return ;

}

**void merge(int \*a , int l , int m , int r)**

{

int n1 , n2 , i , j , k ;

n1 = m – l + 1 ;

n2 = r – m ;

int left[n1] ;

int right[n2] ;

for(i=0 ; i<n1 ; i++)

{

left[i] = a[l+i];

}

for(i=0 ; i<n2 ; i++)

{

right[i] = a[m+1+i] ;

}

i = 0 ; j = 0 ; k = l ;

while(i<n1 && j<n2)

{

if(left[i]<right[j])

{

a[k] = left[i] ;

k++ ; i++ ;

}

else

{

a[k] = right[j] ;

k++ ; j++ ;

}

}

while(i<n1)

{

a[k] = left[i] ;

k++ ; i++ ;

}

while(j<n2)

{

a[k] = right[j];

k++ ; j++ ;

}

return ;

}

***Time-complexity mergeSort() : O(nlog2n)***

**5) . Quick Sort**