Searching

Linear Search means to sequentially traverse a given list or array and check if an element is present in the respective array or list. The idea is to start traversing the array and compare elements of the array one by one starting from the first element with the given element until a match is found or end of the array is reached.

Time Complexity: O(N). Since we are traversing the complete array, so in worst case when the element X does not exists in the array, number of comparisons will be N.
Therefore, *worst case time complexity of the linear search algorithm is O(N)*.

Binary Search is a searching algorithm for searching an element in a sorted list or array. Binary Search is efficient than Linear Search algorithm and performs the search operation in logarithmic time complexity for sorted arrays or lists.

Binary Search performs the search operation by repeatedly dividing the search interval in half. The idea is to begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Time Complexity: O(Log N), where N is the number of elements in the array.

Binary Search Iterative

```cpp
#include<iostream>
using namespace std;

int bSearch(int arr[], int n , int x)
{
    int low=0, high=n-1;

    while (low<=high)
    {
        int mid=(low+high)/2;

        if(arr[mid]==x)
            return mid;
        else if(arr[mid]>x)
            high=mid-1;
        else
            low=mid+1;
    }
    return -1;

}

int main()
{
    int arr[]={10,20,30,40,50,60},n=6;
    int x=25;
    cout<<bSearch(arr,n,x);
}
```

OUTPUT:

```
-1
```

Binary Search Recursive

```cpp
#include <iostream>
using namespace std;

int bSearch(int arr[],int low,int high, int x)
{
    if(low>high)
        return -1;

    int mid=(low+high)/2;
    if(arr[mid]==x)
        return mid;

    else if(arr[mid]>x)
        return bSearch(arr, low, mid-1, x);

    else
        return bSearch(arr,mid+1,high,x);
}

int main() {

     int arr[] = {10, 20, 30, 40, 50, 60, 70}, n = 7;

    int x = 50;

    cout<<bSearch(arr, 0, n - 1, x);
    return 0;
}
```

OUTPUT:

```
4
```

Index Of First Occurrence in sorted

Naïve:

```
#include<iostream>
using namespace std;

int firstOccurrence(int arr[], int n, int x)
{
    for(int i=0;i<n;i++)
        if(arr[i]==x)
            return i;

    return -1;
}

int main() {

     int arr[] = {5, 10, 10, 15, 15}, n = 5;

    int x = 15;

    cout<<firstOccurrence(arr, n, x);
    return 0;
}
```

OUTPUT:

```
3
```

Efficient iterative solution of Index Of First Occurrence in Sorted

```cpp
#include<iostream>
using namespace std;

int firstOccruance(int arr[],int n,int x)
{
    int low=0,high=n-1;


    while(low<=high)
    {
        int mid=(low+high)/2;

        if(arr[mid]>x)
            high=mid-1;
        else if(arr[mid]<x)
            low=mid+1;
        else
        {
            if(mid==0 || arr[mid-1]!=arr[mid])
                return mid;

            else
                high=mid-1;
        }
    }
    return -1;
}

int main() {

    int arr[] = {5, 10, 10, 15, 20, 20, 20}, n = 7;

    int x = 25;

    cout<<firstOccruance(arr, n , x);
    return 0;
}
```

OUTPUT:

```
-1
```

Efficient recursive solution of Index Of First Occurrence in Sorted

```cpp
#include<iostream>
using namespace std;

int firstOccurrence(int arr[], int low, int high, int x)
{
    if(low>high)
        return -1;

    int mid=(low+high)/2;
    if(arr[mid]>x)
        return firstOccurrence(arr,low,mid-1,x);
    else if(arr[mid]<x)
        return firstOccurrence(arr,mid+1,high,x);

    else
    {
        if(mid==0 || arr[mid-1]!=arr[mid])
            return mid;
        else
            return firstOccurrence(arr,low,mid-1,x);
    }
}

int main() {

    int arr[] = {5, 10, 10, 15, 20, 20, 20}, n = 7;

    int x = 20;

    cout<<firstOccurrence(arr, 0, n - 1, x);
    return 0;
}
```

OUTPUT:

```
4
```

Index Of Last Occurrence in sorted

Efficient Iterative:

```cpp
#include<iostream>
using namespace std;

int lastOccurance(int arr[],int n, int x)
{
    int low=0, high=n-1;

    while(low<=high)
    {
        int mid=(low+high)/2;
        if(arr[mid]>x)
            high=mid-1;
        else if(arr[mid]<x)
            low=mid+1;
        else
        {
            if(mid==n-1 || arr[mid+1]!=arr[mid])
                return mid;
            else
                low=mid+1;
        }
    }
    return -1;
}

int main() {

   int arr[] = {5, 10, 10, 10, 10, 20, 20}, n = 7;

    int x = 20;

    cout << lastOccurance(arr, n, x);
    return 0;
}
```

OUTPUT:

```
6
```

Efficient recursive solution of Index Of last Occurrence in Sorted

```
#include<iostream>
using namespace std;

int lastOccurance(int arr[],int low,int high, int x,int n)
{
    if(low>high)
        return -1;
    int mid=(low+high)/2;

    if(arr[mid]>x)
        return lastOccurance(arr,low,mid-1,x,n);
    else if(arr[mid]<x)
        return lastOccurance(arr,mid+1,high,x,n);
    else
    {
        if(mid==n-1 || arr[mid+1]!=arr[mid])
            return mid;
        else
            return lastOccurance(arr,mid+1,high,x,n);
    }
    return -1;
}

int main() {

   int arr[] = {5, 10, 10, 10, 10, 20, 20}, n = 7;

    int x = 25;

    cout << lastOccurance(arr, 0, n - 1, x,n);
    return 0;
}
```

OUTPUT:

```
-1
```

Count Occurrence in sorted Array

```cpp
#include<iostream>
using namespace std;

int firstOccurance(int arr[], int n , int x)
{
    int low=0,high=n-1;
    while(low<=high)
    {
        int mid=(low+high)/2;

        if(x<arr[mid])
            high=mid-1;
        else if(x>arr[mid])
            low=mid+1;
        else
        {
            if(mid==0 || arr[mid-1]!=arr[mid])
                return mid;
            else
                high=mid-1;
        }
    }
    return -1;
}

int lastOccurance(int arr[], int n, int x)
{
    int low=0, high=n-1;
    while (low<=high)
    {
        int mid=(low+high)/2;

        if(x>arr[mid])
            low=mid+1;
        else if(x<arr[mid])
            high=mid-1;
        else
        {
            if(mid==n-1 || arr[mid+1]!=arr[mid])
                return mid;
            else
                low=mid+1;
        }
    }
    return -1;
}
```

```
int countOccrunace(int arr[], int n , int x)
{
    int first=firstOccurance(arr,n,x);

    if(first==-1)
        return 0;
    else
        return lastOccurance(arr,n,x)-first+1;
}

int main() {

    int arr[] = {10, 20, 20, 20, 40, 40}, n = 6;

    int x = 20;

    cout<<countOccrunace(arr, n,x);

    return 0;
}
```

OUTPUT:

```
3
```

Count 1s in Sorted Binary Array

```cpp
#include<iostream>
using namespace std;

int countOnes(int arr[], int n)
{
    int low=0,high=n-1;

    while ((low<=high))
    {
        int mid=(low+high)/2;

        if(arr[mid]==0)
            low=mid+1;
        else
        {
            if(mid==0 || arr[mid-1]==0)
                return(n-mid);
            else
                high=mid-1;
        }
    }

    return 0;
}

int main() {

   int arr[] = {0, 0, 1, 1, 1, 1}, n = 6;

   cout << countOnes(arr, n);

    return 0;
}
```

OUTPUT:

```
4
```

Sqaure Root

Naïve:

```cpp
#include<iostream>
using namespace std;

int sqRootFloor(int x)
{
    int i=1;
    while (i*i<=x)
         i++;

    return i-1;
}

int main()
{
    cout<<sqRootFloor(8);

    return 0;
}
```

OUTPUT: 2

Efficient solution for square root

```cpp
#include<iostream>
using namespace std;

// here we use last occurance concept find finding last occurance
int sqRoot(int x)
{
    int low=0,high=x, ans=-1;

    while (low<=high)
    {
        int mid=(low+high)/2;

        int mSq=mid*mid;
        if(mSq==x)
            return mid;
        else if(mSq>x)
            high=mid-1;
        else
        {
            low=mid+1;
            ans=mid;
        }
    }
    return ans;
}

int main()
{
    cout<<sqRoot(24);

    return 0;
}
```

OUTPUT:

```
4
```

Search in Infinite Size Array

Naïve:

```cpp
#include<iostream>
using namespace std;

int search(int arr[], int x)
{
    int i=0;
    while (true)
    {
        if(arr[i]==x)
            return i;
        if(arr[i]>x)
            return -1;
        i++;
    }
}

int main() {


   int arr[] = {1, 2, 3, 5, 5};

    int x = 5;

    cout << search(arr, x);

    return 0;
}
```

OUTPUT:

```
3
```

Efficient solution for Search in Infinite Sized Array

```
#include<iostream>
using namespace std;

int bSearch(int arr[], int low, int high,int x)
{
    if(low>high )
        return -1;

    int mid=(low+high)/2;
    if(arr[mid]==x)
        return mid;
    else if(arr[mid]>x)
        return bSearch(arr,low,mid-1,x);
    else
        return bSearch(arr,mid+1,high,x);
}

int search(int arr[], int x)
{
    if(arr[0]==x)
        return 0;

    int i=1;
    while (arr[i]<x)
        i*=2;

    if(arr[i]==x)
        return i;

    return bSearch(arr,i/2 +1,i-1,x);

}

int main() {


   int arr[] = {1, 2, 3, 5, 5};

    int x = 5;

    cout << search(arr, x);

    return 0;
}
```

OUTPUT: 4

Search in Sorted Rotated Array

Naïve:

```cpp
#include<iostream>
using namespace std;

int search(int arr[],int n, int x)
{
    for(int i=0;i<n;i++)
        if(arr[i]==x)
            return i;
    return -1;
}

int main() {


    int arr[] = {100, 200, 400, 1000, 10, 20}, n = 6;

    int x = 10;

    cout << search(arr, n, x);

    return 0;
}
```

OUTPUT:

```
4
```

Efficient solution for Search in sorted rotated array

```cpp
#include<iostream>
using namespace std;

//in sorted rotated array half array is always sorted
int search(int arr[], int n , int x)
{
    int low=0,high=n-1;

    while (low<=high)
    {
        int mid=(low+high)/2;

        if(arr[mid]==x)
            return mid;

        if(arr[low]<arr[mid])
        {
            if(x>=arr[low] && x<arr[mid])
                high=mid-1;
            else
                low=mid+1;
        }
        else
        {
            if(x>arr[mid] && x<=arr[high])
                low=mid+1;
            else
                high=mid-1;
        }
    }

    return -1;
}

int main() {


  int arr[] = {10, 20, 40, 60, 5, 8}, n = 6;

  int x = 5;

    cout << search(arr, n, x);

    return 0;
}
```

OUTPUT: 4

Find Peak Element

Naïve:

```cpp
#include<iostream>
using namespace std;

// peak element means number is not smaller than neighbor
int getPeak(int arr[],int n)
{
    if(n==1)
        return arr[0];
    if(arr[0]>=arr[1])
        return arr[0];
    if(arr[n-1]>=arr[n-2])
        return arr[n-1];

    for(int i=1;i<n-1;i++)
    {
        if(arr[i]>=arr[i-1] && arr[i]>=arr[i+1])
            return arr[i];
    }
    return -1;

}

int main() {


 int arr[] = { 10, 7, 8, 20, 12}, n = 5;

 cout << getPeak(arr, n);

    return 0;
}
```

OUTPUT:

```
10
```

Efficient solution for Find Peak Element

```cpp
#include<iostream>
using namespace std;

// peak element means number is not smaller than neighbor
// there is always peak element in array if array arranged any manner

int getPeak(int arr[], int n)
{
    int low=0, high=n-1;

    while (low<=high)
    {
        int mid=(low+high)/2;

        if((mid==0 || arr[mid-1]<=arr[mid]) &&
        (mid==n-1 || arr[mid+1]<=arr[mid]))
            return mid;                               //arr[mid] for peak element

        if(mid>0 &&  arr[mid-1]>=arr[mid])
            high=mid-1;
        else
            low=mid+1;
    }

    return -1;
}

int main() {


 int arr[] = {5, 10, 15, 30, 40, 50, 10}, n = 7;

 cout << getPeak(arr, n);

    return 0;
}
```

OUTPUT:

```
5
```

Two Pointer Approach

Find pair in unsorted array which gives sum X

```
// C++ program to check if given array
// has 2 elements whose sum is equal
// to the given value
#include <bits/stdc++.h>

using namespace std;

void printPairs(int arr[], int arr_size, int sum)
{
    unordered_set<int> s;
    for (int i = 0; i < arr_size; i++) {
        int temp = sum - arr[i];

        if (s.find(temp) != s.end())
            cout << "Pair with given sum " << sum << " is (" << arr[i] << ", "
 << temp << ")" << endl;

        s.insert(arr[i]);
    }
}

/* Driver program to test above function */
int main()
{
    int A[] = { 1, 4, 45, 6, 10, 8 };
    int n = 16;
    int arr_size = sizeof(A) / sizeof(A[0]);

    // Function calling
    printPairs(A, arr_size, n);

    return 0;
}
```

OUTPUT:

```
Pair with given sum 16 is (10, 6)
```

Find pair in sorted array which gives sum X

```
# #include<iostream>

using namespace std;

// Find pair in sorted array which gives sum X if array is sorted

bool isPresent(int arr[], int n, int sum)
{
    int low=0,high=n-1;
    while (low<high)
    {
        if(arr[low]+arr[high]==sum)
            return true;
        else if(arr[low]+arr[high]>sum)
            high--;
        else
            low++;
    }

    return false;
}

int main()
{
    int arr[] = {2, 3, 7, 8, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    int sum = 14;

    cout << isPresent(arr, n, sum);
}
```

OUTPUT:

```
1
```

Find triplet in an array which gives sum X

```
#include<bits/stdc++.h>
using namespace std;

// returns true if there is triplet with sum equal
// to 'sum' present in A[]. Also, prints the triplet

bool find3Numbers(int A[], int arr_size, int sum)
{
    int l,r;

    // sort the element
    sort(A,A+arr_size);

    /* Now fix the first element one by one and find the
    other two elements */

    for(int i=0;i<arr_size-2;i++)
    {
        // To find the other two elements, start two index
        // variables from two corners of the array and move
        // them toward each other
        l = i + 1; // index of the first element in the
        // remaining elements

        r = arr_size - 1; // index of the last element

        while (l<r)
        {
            if(A[i]+A[l]+A[r]==sum){
                cout<<"Triplet is :"<<A[i]<<" "<<A[l]<<" "<<A[r];
                return true;
            }

            else if(A[i]+A[l]+A[r]<sum)
                l++;
            else   // A[i] + A[l] + A[r] > sum
                r--;

        }

    }

    // If we reach here, then no triplet was found
    return false;
}

/* Driver program to test above function */
```

```
int main()
{
    int A[] = { 1, 4, 45, 6, 10, 8 };
    int sum = 22;
    int arr_size = sizeof(A) / sizeof(A[0]);

    find3Numbers(A, arr_size, sum);

    return 0;
}
```

OUTPUT:

```
Triplet is :4 8 10
```

Median of Two Sorted Array

```cpp
#include<bits/stdc++.h>
using namespace std;

double getMed(int a1[],int a2[],int n1, int n2)
{
    int begin1=0, end1=n1;

    while(begin1<end1)
    {
        int i1=(begin1+end1)/2;
        int i2=(n1+n2+1)/2 -i1;

        int min1=(i1==n1)?INT32_MAX:a1[i1];
        int max1=(i1==0)?INT32_MIN:a1[i1-1];

        int min2=(i2==n2)?INT32_MAX:a2[i2];
        int max2=(i2==0)?INT16_MIN:a2[i2-1];

        if(max1<=min2 && max2<=min1)
        {
            if((n1+n2)%2==0)
                return ((double) max(max1,max2)+ min(min1,min2))/2;

            else
                return (double)max(max1,max2);
        }
        else{
            if(max1>min2)
                end1=i1-1;
            else
                begin1=i1+1;
        }
    }

    return 0;
}

int main(){
    int a1[]={10,20,30,40,50}, n1=5, a2[]={5,15,25,35,45}, n2=5;

    cout<<getMed(a1,a2,n1,n2);

    return 0;
}
```

OUTPUT : 27.5

Repeating Element Part 1

```cpp
#include<bits/stdc++.h>
using namespace std;

int repeat(int arr[], int n)
{
    bool visit[n];

    memset(visit, false,sizeof(visit));

    for(int i=0;i<n;i++)
    {
        if(visit[arr[i]])
            return arr[i];
        visit[arr[i]]=true;
    }
    return -1;
}

int main()
{
    int arr[]={0,2,1,3,2,2};
    int n=sizeof(arr)/sizeof(arr[0]);

    cout<<repeat(arr,n);

    return 0;
}
```

OUTPUT:

```
2
```

Repeating Element Part 2

```
#include<iostream>
using namespace std;

int repeat(int arr[], int n)
{
    int slow=arr[0],fast=arr[0];

    do{
        slow=arr[slow];
        fast=arr[arr[fast]];
    }while(slow!=fast);

    slow=arr[0];

    while(slow!=fast)
    {
        slow=arr[slow];
        fast=arr[fast];
    }

    return slow;
}

int main() {


 int arr[] = {1, 3, 2, 4, 6, 5, 7, 3}, n= 8;

 cout << repeat(arr, n);

    return 0;
}
```

OUTPUT:

```
3
```

Allocate Minimum Pages Naïve Method

Naïve

```cpp
//minimize the maximum number of pages read by the student
//book read by student are in contigeous in array

#include <iostream>
using namespace std;

int sum(int arr[], int b, int e)
{
    int s=0;
    for(int i=b;i<=e;i++)
        s+=arr[i];
    return s;
}

int minPages(int arr[],int n, int k)
{
    if(k==1)
        return sum(arr,0,n-1);
    if(n==1)
        return arr[0];
    int res=INT16_MAX;
    for(int i=1;i<n;i++)
    {
        res=min(res,max(minPages(arr,i,k-1),sum(arr,i,n-1)));
    }
    return res;
}

int main()
{
    int arr[]={10,20,10,30};
    int n=sizeof(arr)/sizeof(arr[0]);
    int k=2;

    cout<<minPages(arr,n,k);
}
```

OUTPUT:

40

Efficient solution Allocate minimum pages Binary Search

```
//minimize the maximum number of pages read by the student
//book read by student are in contigeous in array

#include<iostream>
using namespace std;

bool isFeasible(int arr[],int n , int k,  int ans)
{
    int req=1, sum=0;
    for(int i=0;i<n;i++)
    {
        if(sum+arr[i]>ans)
        {
            req++;
            sum=arr[i];
        }
        else
            sum+=arr[i];
    }
    return (req<=k);
}

int minPages(int arr[], int n, int k)
{
    int sum=0, mx=0;
    for(int i=0;i<n;i++)
    {
        sum+=arr[i];
        mx=max(mx,arr[i]);
    }

    int low=mx, high=sum,res=0;

    while(low<=high)
    {
        int mid=(low+high)/2;
        if(isFeasible(arr,n,k,mid))
        {
            res=mid;
            high=mid-1;
        }
        else
            low=mid+1;
    }

    return res;
}
```

```cpp
int main()
{
    int arr[]={10,20,10,30};
    int n=sizeof(arr)/sizeof(arr[0]);
    int k=2;

    cout<<minPages(arr,n,k);
}
```

OUTPUT:

40