Week #4

| Complexity

Algorithmic complexity is concerned about how fast or slow particular algorithms perform.

1- Time Complexity:

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of input.

2- Space Complexity:

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the size of input.

O-Notation [`O()`] :

Big O notation is used in Computer Science to describe the complexity of an algorithm. Big O specifically describes the **worst-case** scenario, and can be used to describe the execution time required or the space used.

Let's see some examples to understand it better.

- O(1) *describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data.*

- Arithmetic Operations.

```
int x = 2, y = 3, z;
z = x + y;
```

Time Complexity = O(1)
Space Complexity = O(1)

- Accessing any array element.

```
const int N = 3;
int arr[N] = {2, 4, 6}, x;
x = arr[1];
```

Time Complexity = O(1)
Space Complexity = O(N)

- O(N) *describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data.*

- Iterating over 1D arrays.

```
const int N = 4;
int arr[N] = {1, 2, 3, 4}, sum = 0;
for(int i = 0; i < N; i++)
{
   Sum += arr[i];
}
```

Time Complexity = O(N)
Space Complexity = O(N)

- O(N^2) *represents an algorithm whose performance is directly proportional to the square of the size of the input data . This is common with algorithms that involve nested iterations over the data . Deeper nested iterations will result in O(N^3), O(N^4) etc.*

- Iterating over nested loops.

```
int arr[N][N], sum = 0;
for(int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
       sum += arr[i][j];
   }
}
```

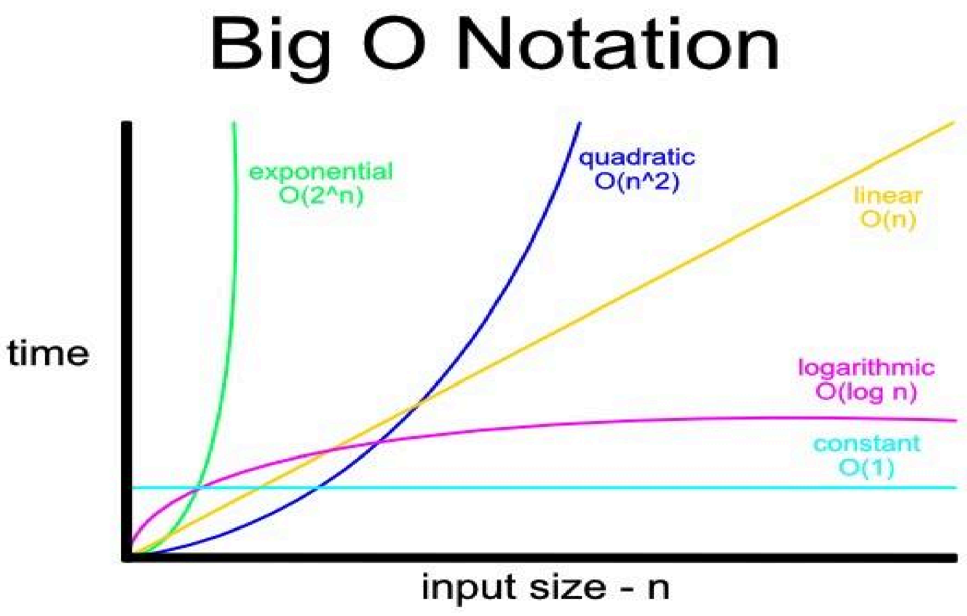
Time Complexity = O(N * N)
Space Complexity = O(N * N)

- O(log(N)) *Finding an item in a sorted array with a binary search or a balanced search tree.*
Which means splitting the input into 2 equal halves.(decreases the search space by half in each iteration)

```
int N, cnt = 0;
cin >> N;
while(N > 1)
{
   cnt++;
   N = N / 2;
}
```

Time Complexity = O(Log (N))
Space Complexity = O(1)

This graph illustrates the relation between input and it’s time complexity:

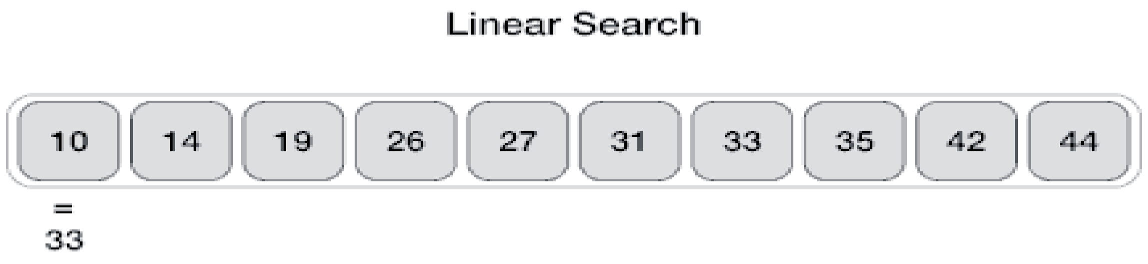


- As the size of the input increases its time complexity increases.

| Binary Search

If you were given an array or list of numbers and you want to search for an integer X , The linear approach is to iterate over all numbers one by one from the beginning.

Example : We are searching for number 33 in this array.



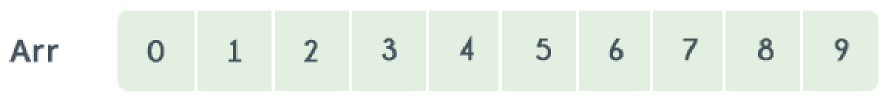
In the worst case it takes N (the total num of elements) steps to reach the answer.

But, What if the N was more than or equal 10^9!!

Definitely your program will Explode 💥!

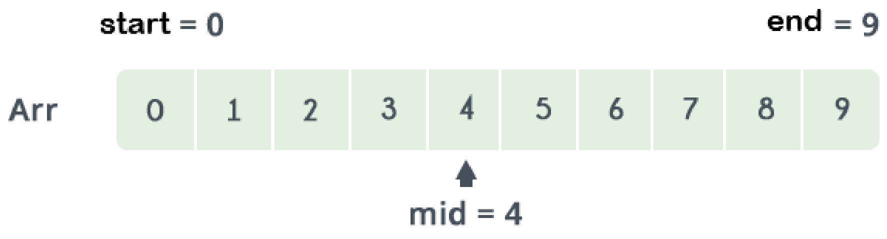
So a better searching technique is used which is easier and faster called **Binary search**, it works only on a **Sorted** set of elements. To use binary search on a collection, the collection must first be sorted.

Let us consider the following array:



By using linear search, the position of element 8 will be determined in the 9th iteration.

Let's see how the number of iterations can be reduced by using binary search. Before we start the search, we need to know the start and end of the range. Let us call them Start and End.



Start is marked at the beginning of the array at **Arr[0].**
End is marked at the end of the array at **Arr[N - 1].**

```
int Start = 0;
int End = N - 1;
```

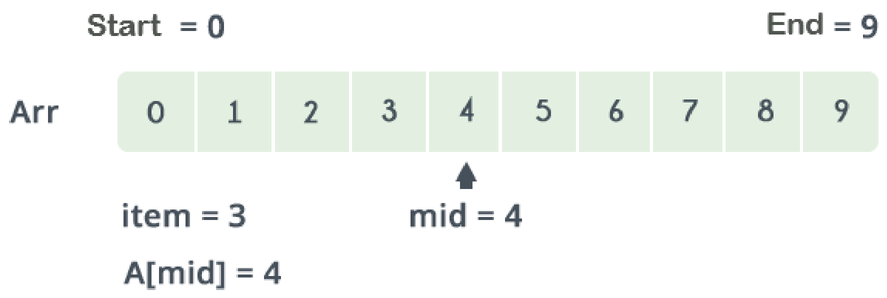
Now, we get the Middle element between our two bounds the Start and the End.
and we compare the target value (**X**) to the middle element of the interval.

```
int Middle = (Start + End) / 2;
```

If the target value **X** is greater than the Middle element, increase the lower bound ”Start”, else decrease the upper bound ”End”.

```
if (arr[Middle] < X)
{
   Start = Middle + 1;
}
else
{
   End = Middle - 1;
}
```

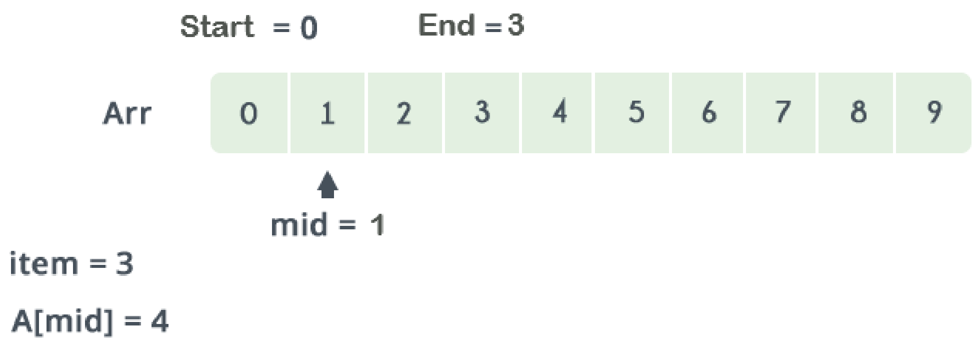
Let us Explain further more on this example:



Referring to the image above, we are looking for the third item which is = **2,**
The Start is 0 and the End is 9.
The Middle of the lower and upper bounds is (Start + End) / **2** = **4**.
Here **Arr[4]** = **4**. The value **4 > 2**.

Therefore, we do not need to conduct a search on any element that comes after 4 as the elements beyond it will obviously be greater than 2.

So,the End is shifted just before the middle element.



This procedure repeats until the Start > End. If at any iteration, We get
Arr [Middle]= X, we return value of Middle. This is the position of X in the array. If X is not present in the array, we return −1 or false.

Implementation:

```
int BinarySearch(int Start, int End, int X)
{
   while(Start <= End)
   {
       int Middle = (Start + End) / 2;
       if (Arr[Middle] < X)
       {
           Start = Middle + 1;
       }
       else if (Arr[Middle] > X)
       {
           End = Middle - 1;
       }
       else
       {
           return Middle;
       }
   }
   return -1;
}                //X not found
```

Built in binary search:

In C++ we have two built in functions similar to binary search: upper-bound and lower_bound, both functions take three parameters (first, last, value).

upper_bound: Returns an iterator pointing to the first element greater than to the value we are looking for.
Possible implementation for the upper_bound function:

```
const int SIZE = 5;
int arr[SIZE] = {1 , 4 , 4 , 5 , 7};
int Start = 0, End = SIZE - 1, mid, Index = -1, value;
while (Start <= End)
{
    mid = (Start + End) / 2;
    if (arr[mid] <= value)
    {
        Index = mid;
        Start = mid + 1;
    }
    else
        End = mid-1
}
cout << Index + 1 << endl;
cout << upper_bound(arr, arr + SIZE, value) - arr << endl;
```

For the given example if the value is 4 the output will be 3 while if the value is 0 the output will be 0.

lower_bound: Returns an iterator pointing to the first element smaller than or equal to the value we are looking for.
Possible implementation for the lower_bound function:

```
const int SIZE = 5;
int arr[SIZE] = {1 , 4 , 4 , 5 , 7};
int Start = 0, End = SIZE - 1, mid, Index = -1, value;
while(Start <= End)
{
    mid = (Start + End) / 2;
    if (arr[mid] < value)
    {
        Start = mid + 1;
    }
    else
    {
        Index = mid;
        End = mid - 1;
    }
}
cout << Index + 1 << endl;
cout << lower_bound(arr, arr + SIZE, value) - arr << endl;
```

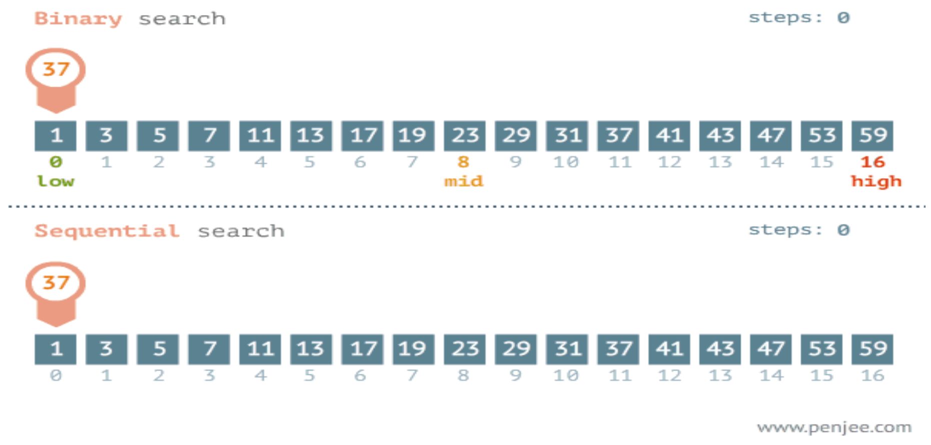
For the given example if the value is 4 the output will be 1 while if the value is 0 the output will be 0.

***So similar functions Could be written in different complexities.**

example:

If we are searching for a number through a list of elements ,
Complexity of binary search is better than linear search ;
O(Log(N)) < O(N).

This illustrates the difference complexities of Binary search and linear search.



The better the time complexity of an algorithm is , the faster the algorithm will carry out his work in practice. Apart from time complexity, its space complexity is also important. This is essentially the number of memory cells which an algorithm needs A good algorithm keeps this number as small as possible.

| Built-in Sorting

In order to use some algorithms i.e binary search you need to sort the data structure you are working on. There are several sorting algorithms, but we will deal with a predefined one.

There are some functions defined before in the libraries of the compiler called built in functions. One of them is the function sort. It sorts the elements in ascending order. Sort function has different forms. They are defined in a library called “algorithm” which you have to include before calling the function. This function doesn’t return anything so it’s void function.

As we mentioned it has several forms. One of them has two parameters and is declared in this format:

```
      void sort(startAdress, endAdress);
```

Example to show how it works:

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

int main()
{
    int n = 10; //size of arr
    int arr[n] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};

    sort(arr, arr + n);

    cout << "Array after sorting using built-in sort is :" << endl;
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```

Output :

```
Array after sorting using default sort is:
0 1 2 3 4 5 6 7 8 9
```

What if we need to sort in descending order ! What if we need to sort according to another concept ex. sorting even then odd numbers all in ascending order !
Should we write the whole algorithm from scratch ?!!

The solution of all of these issues is to use another form of the built-in sort function. This form has three parameters and is declared in this format:

```
      void sort(startAdress, endAdress, compare function);
```

Compare function:

It's a function that returns boolean and takes two parameters:

```
      bool cmp(int i, int j);
```

If the function returns true then i will be on the left of j. Otherwise i will be on the right of j.

So if we want to sort according to this concept: if the number is even it will come first, and if two numbers are odds or evens so the smaller comes first.

Code:

```
bool cmp(int i, int j)
{
   //even then odd all in Ascending order
   if (i % 2 == 0 && j % 2 == 0)
   {
       if (i < j)
           return true;
       else
           return false;
   }
   if(i % 2 == 1 && j % 2 == 1)
   {
       if (i < j)
          return true;
       else
          return false;
   }
   if(i % 2 == 0 && j % 2 == 1)
   {
       return true;
   }
   else
```

```
        return false;
}
```

Then we pass only the function name (cmp) as a third parameter to the sort function. If we applied this code on our last example the output will be:

```
Array after sorting using default sort is:
0 2 4 6 8 1 3 5 7 9
```

| Ternary Search

Like linear search and binary search, ternary search is a searching technique that can be used to determine the position of a specific value in an array. In binary search, the sorted array is divided into two parts while in ternary search, it is divided into **3** parts and then you determine in which part the element exists.

Ternary search, like binary search, is a divide-and-conquer algorithm. It is mandatory for the array (in which you will search for an element) to be sorted before you begin the search. In this search, after each iteration it neglects ⅓ part of the array and repeats the same operations on the remaining ⅔.

Let us consider the following example to understand :

Let the sorted array be **Arr[9] = {2, 3, 5, 6, 8, 9, 12, 13, 14}** with indices from 0 to 8. You are required to find the position of x = 13 in this array.

Divide the sorted array into 3 parts by evaluating the values of **Mid1** & **Mid2**:
- {2, 3, 5}
- {6, 8, 9}
- {12, 13, 14}

Here **Arr[Mid1] = 5** and **Arr[Mid2] = 12**.

As **13** is not equal to **Arr[Mid1]** and **Arr[Mid2]** and it is also not smaller than **Arr[Mid1]**, you can safely assume that it lies in the 3rd part of the array as it is greater than **Arr[Mid2]**.

Run the ternary search again with L=7 and R=8.
Now,**Arr[Mid1]** = **Arr[7]** = **13,** and **Arr[Mid2]** = **Arr[8]** = **14**.
As **Arr[Mid1]** = x, Mid1 is the required answer.
If the value is not in the array, it returns −1 as the answer.

Let's implement it together :

```
int Arr[n], target, ans = -1;
int left = 0, right = n;  // set your range
for(int i = 0; i < 9; i++)
    cin >> Arr[i];
cin >> target;
while (right - left > 3)  // We need 4 different positions
{
    int Mid1 = left + (right - left) / 3;
    int Mid2 = left + 2 * (right - left) / 3;

    If (Arr[Mid1] == target)
    {
        ans = Mid1;
        break;
    }
    else if (Arr[Mid2] == target)
    {
        ans = Mid2;
        break;
    }
    else if (target > Arr[Mid2])
        left = Mid2 + 1;
    else if (target < Arr[Mid1])
        right = Mid1 - 1;
    else
    {
        left = Mid1 + 1;
        right = Mid2 - 1;
    }
}
for (int i = left; i <= right; ++i)// iterate on the remaining
    if(target == Arr[i])
    {
        Ans = i;
    }
cout << Ans;
```