Unit 1: Time and Space Complexity

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

- Independent of the machine and its configuration, on which the algorithm is running on.

- Shows a direct correlation with the number of inputs.

- Can distinguish two algorithms clearly without ambiguity.

There are two such methods used, **time complexity** and **space complexity** which are discussed below:

Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

Definition–

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

Example 1: Addition of two scalar variables.

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <- A + B
return C
```

The addition of two scalar numbers requires one addition operation. the time complexity of this algorithm is constant, so T(n) = O(1) .

In order to calculate time complexity on an algorithm, it is assumed that a **constant time c** is taken to execute one operation, and then the total operations for an input length on **N** are calculated.

Example 1: Consider an example to understand the process of calculation: Suppose a problem is to find whether a pair **(X, Y)** exists in an array, A of **N** elements whose sum is **Z**. The simplest idea is to consider every pair and check if it satisfies the given condition or not.

The pseudo-code is as follows:

```
int a[n];
for(int i = 0;i < n;i++)
  cin >> a[i]

for(int i = 0;i < n;i++)
  for(int j = 0;j < n;j++)
    if(i!=j && a[i]+a[j] == z)
       return true
return false
```

Implementation:

```
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;

// Function to find a pair in the given
// array whose sum is equal to z
bool findPair(int a[], int n, int z)
{
    // Iterate through all the pairs
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)

            // Check if the sum of the pair
            // (a[i], a[j]) is equal to z
            if (i != j && a[i] + a[j] == z)
                return true;

    return false;
}

// Driver Code
int main()
{
    // Given Input
    int a[] = { 1, -2, 1, 0, 5 };
    int z = 0;
    int n = sizeof(a) / sizeof(a[0]);

    // Function Call
    if (findPair(a, n, z))
        cout << "True";
    else
        cout << "False";
    return 0;
}
```

Assuming that each of the operations in the computer takes approximately constant time, let it be **c**. The number of lines of code executed actually depends on the value of **Z**. During analyses of the algorithm, mostly the worst-case scenario is considered, i.e., when there is no pair of elements with sum equals **Z**. In the worst case,

- **N*c** operations are required for input.
- The outer loop **i** loop runs **N** times.
- For each **i**, the inner loop **j** loop runs **N** times.

So total execution time is **N*c + N*N*c + c**. Now ignore the lower order terms since the lower order terms are relatively insignificant for large input, therefore only the highest order term is taken (without constant) which is **N*N** in this case. Different notations are used to describe the limiting behavior of a function, but since the worst case is taken so big-O notation will be used to represent the time complexity.

Hence, the time complexity is **O(N^2)** for the above algorithm. Note that the time complexity is solely based on the number of elements in array **A** i.e the input length, so if the length of the array will increase the time of execution will also increase

Order of growth is how the time of execution depends on the length of the input. In the above example, it is clearly evident that the time of execution quadratically depends on the length of the array. Order of growth will help to compute the running time with ease.

Example: Let's calculate the time complexity of the below algorithm:

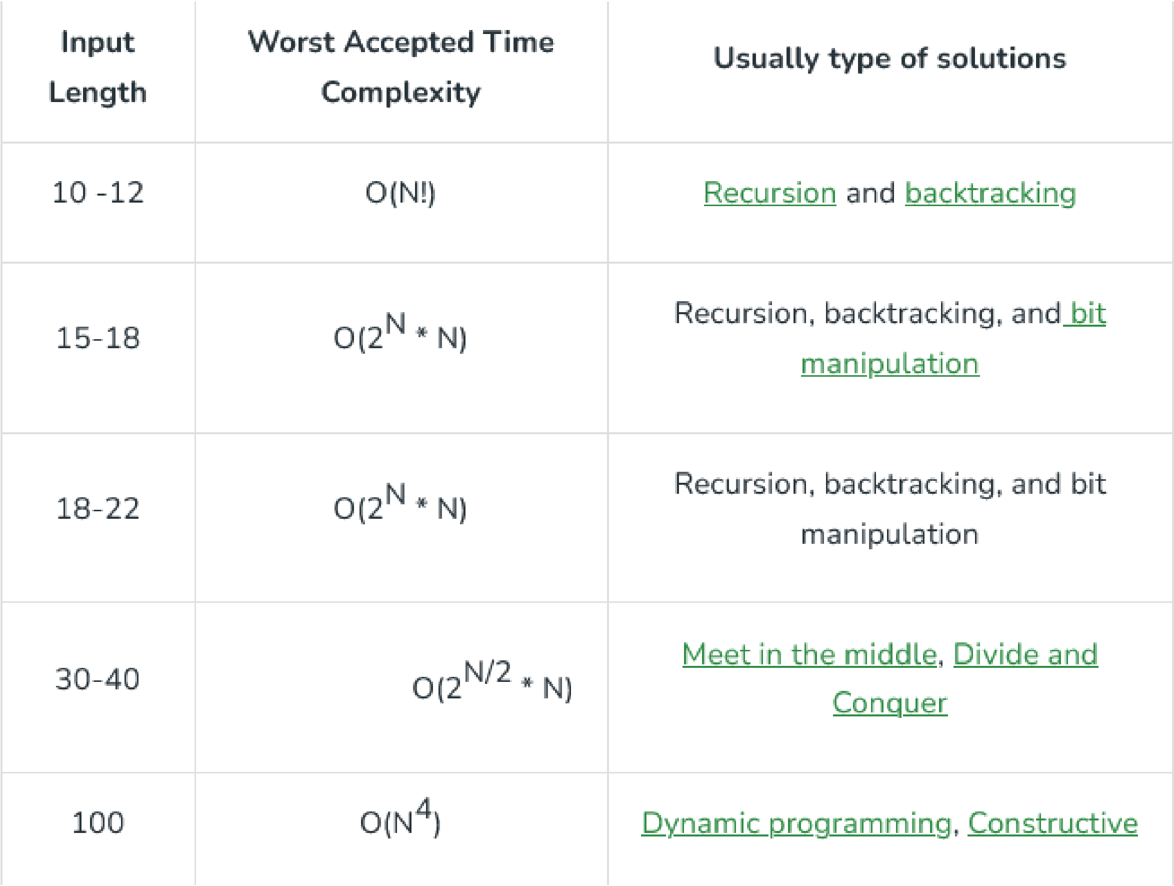
```
count = 0
for (int i = N; i > 0; i /= 2)
  for (int j = 0; j < i; j++)
    count++;
```

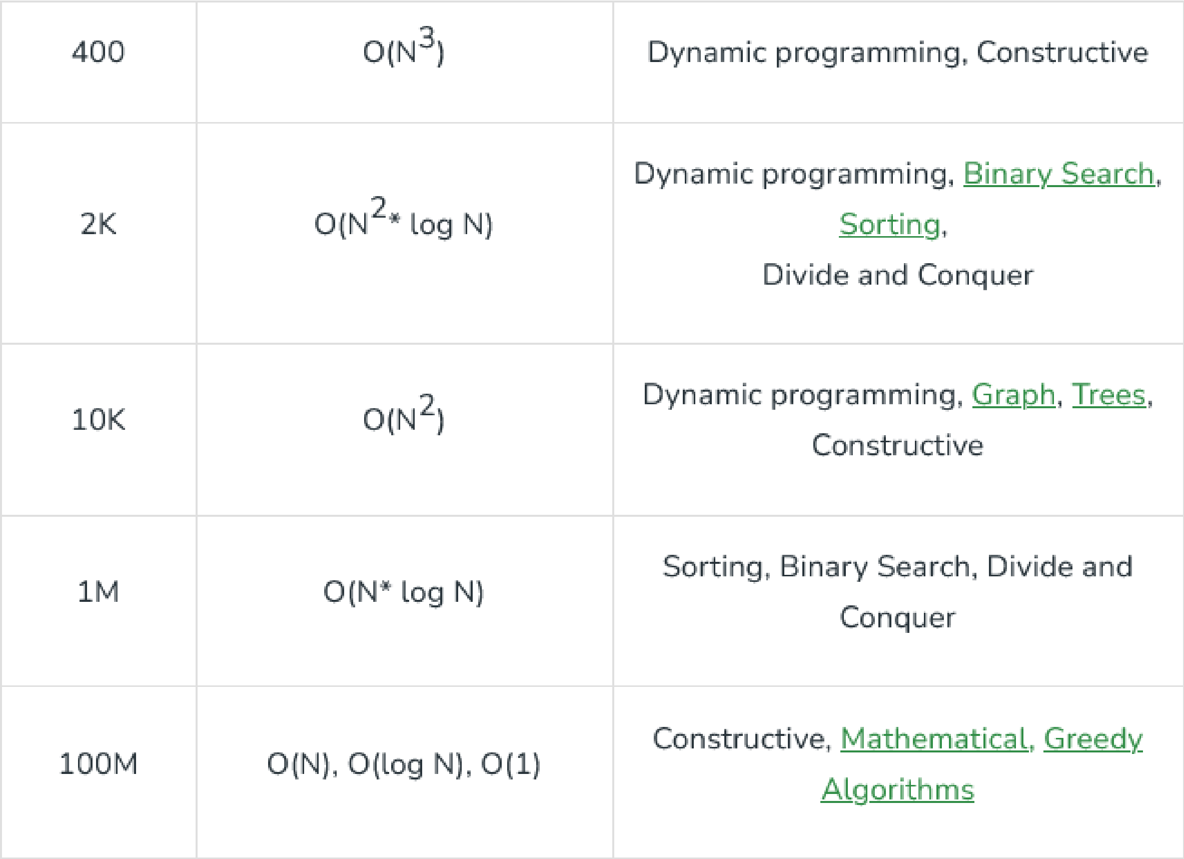
In the first look, it seems like the complexity is **O(N * log N)**. **N** for the **j's** loop and **log(N)** for **i's** loop. But it's wrong. Let's see why.

Think about how many times **count++** will run.

- When **i = N**, it will run **N** times.
- When **i = N / 2**, it will run **N / 2** times.
- When **i = N / 4**, it will run **N / 4** times.
- And so on.

The total number of times **count++** will run is **N + N/2 + N/4+…+1= 2 * N**. So the time complexity will be **O(N)**.Some general time complexities are listed below with the input range for which they are accepted in competitive programming:





Input Length	Worst Accepted Time Complexity	Usually type of solutions
10 -12	O(N!)	Recursion and backtracking
15-18	O(2^{N} * N)	Recursion, backtracking, and bit manipulation
18-22	O(2^{N} * N)	Recursion, backtracking, and bit manipulation
30-40	O($2^{N/2}$ * N)	Meet in the middle, Divide and Conquer
100	O(N^{4})	Dynamic programming, Constructive
400	O(N^{3})	Dynamic programming, Constructive
2K	O(N^{2}* log N)	Dynamic programming, Binary Search, Sorting, Divide and Conquer
10K	O(N^{2})	Dynamic programming, Graph, Trees, Constructive
1M	O(N* log N)	Sorting, Binary Search, Divide and Conquer
100M	O(N), O(log N), O(1)	Constructive, Mathematical, Greedy Algorithms

Space Complexity:

Definition –

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.

It is the amount of memory needed for the completion of an algorithm.

To estimate the memory requirement, we need to focus on two parts:

(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

Example 2: Addition of two scalar variables

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <— A+B
return C
```

The addition of two scalar numbers requires one extra memory location to hold the result. Thus the space complexity of this algorithm is constant, hence S(n) = O(1).

The pseudo-code is as follows:

```
int freq[n];
int a[n];
for(int i = 0; i<n; i++)
{
   cin>>a[i];
   freq[a[i]]++;
}
```

Implementation of the Space Complexity:

```
// C++ program for the above approach
#include <bits/stdc++.h>
using namespace std;
```

```
// Function to count frequencies of array items
void countFreq(int arr[], int n)
{
    unordered_map<int, int> freq;

    // Traverse through array elements and
    // count frequencies
    for (int i = 0; i < n; i++)
        freq[arr[i]]++;

    // Traverse through map and print frequencies
    for (auto x : freq)
        cout << x.first << " " << x.second << endl;
}

// Driver Code
int main()
{
    // Given array
    int arr[] = { 10, 20, 20, 10, 10, 20, 5, 20 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function Call
    countFreq(arr, n);
    return 0;
}
```

Here two arrays of length **N**, and variable **i** are used in the algorithm so, the total space used is **N * c + N * c + 1 * c = 2N * c + c**, where **c** is a unit space taken. For many inputs, constant **c** is insignificant, and it can be said that the space complexity is **O(N)**.

There is also **auxiliary space,** which is different from space complexity. The main difference is where space complexity quantifies the total space used by the algorithm, auxiliary space quantifies the extra space that is used in the algorithm apart from the given input. In the above example, the auxiliary space is the space used by the freq[] array because that is not part of the given input. So total auxiliary space is **N * c + c** which is **O(N)** only.

Unit 2: Data Structures - Divide and Conquer

To understand the divide and conquer design strategy of algorithms, let us use a simple real world example. Consider an instance where we need to brush a type C curly hair and remove all the knots from it. To do that, the first step is to section the hair in smaller strands to make the combing easier than combing the hair altogether. The same technique is applied on algorithms.

Divide and conquer approach breaks down a problem into multiple sub-problems recursively until it cannot be divided further. These sub-problems are solved first and the solutions are merged to form the final solution.

The standard procedure for the divide and conquer design technique is as follows −

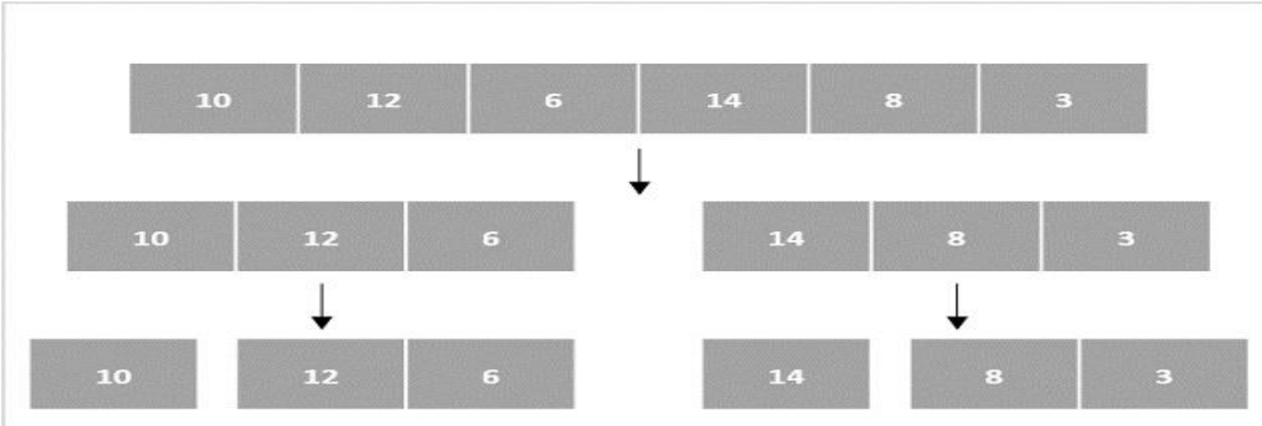
- **Divide** − We divide the original problem into multiple sub-problems until they cannot be divided further.
- **Conquer** − Then these subproblems are solved separately with the help of recursion
- **Combine** − Once solved, all the subproblems are merged/combined together to form the final solution of the original problem.

There are several ways to give input to the divide and conquer algorithm design pattern. Two major data structures used are − **arrays** and **linked lists**. Their usage is explained as

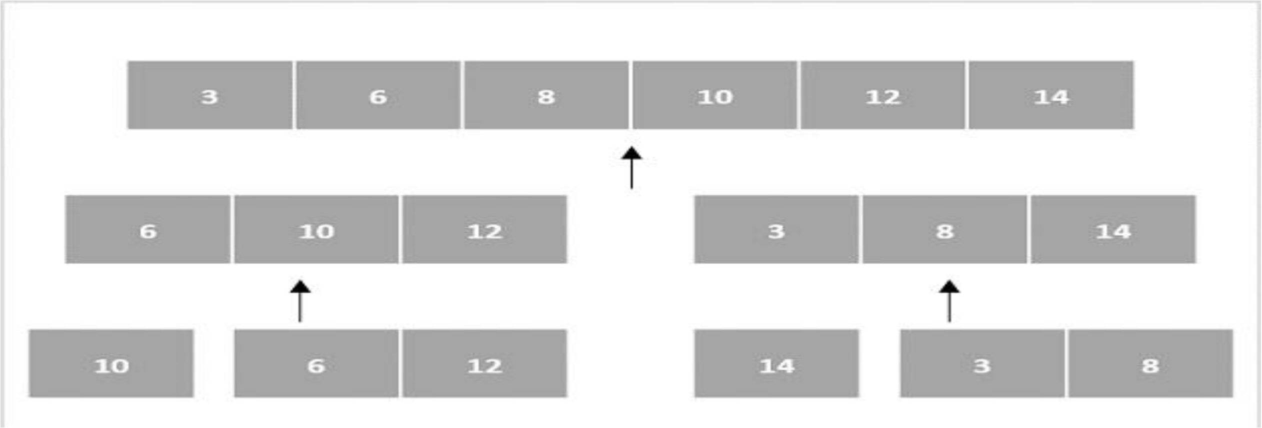
Arrays as Input

There are various ways in which various algorithms can take input such that they can be solved using the divide and conquer technique. Arrays are one of them. In algorithms that require input to be in the form of a list, like various sorting algorithms, array data structures are most commonly used.

In the input for a sorting algorithm below, the array input is divided into subproblems until they cannot be divided further.



Then, the subproblems are sorted (the conquer step) and are merged to form the solution of the original array back (the combine step).

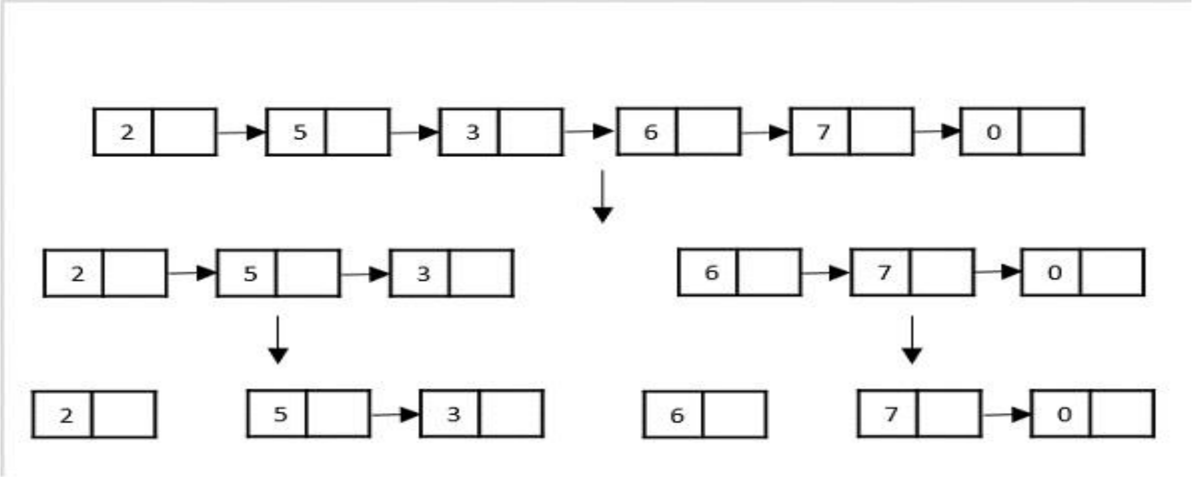


Since arrays are indexed and linear data structures, sorting algorithms most popularly use array data structures to receive input.

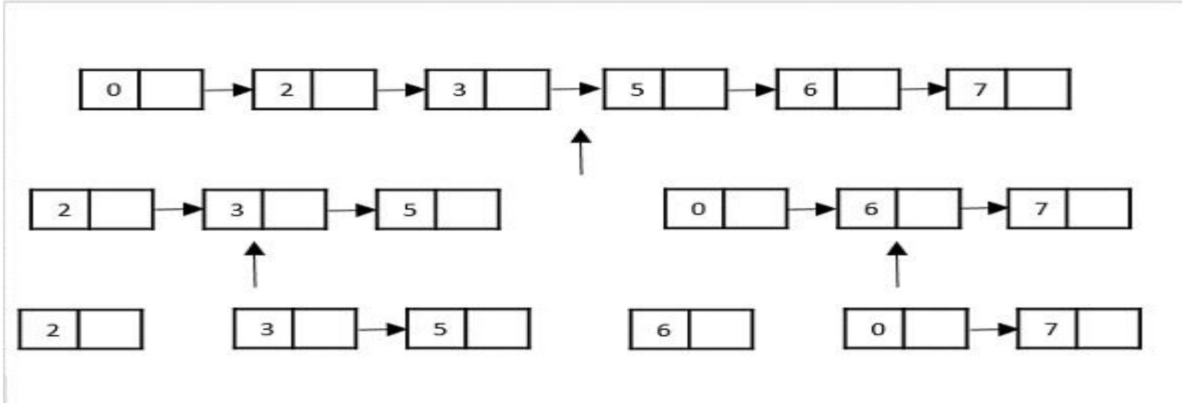
Linked Lists as Input

Another data structure that can be used to take input for divide and conquer algorithms is a linked list (for example, merge sort using linked lists). Like arrays, linked lists are also linear data structures that store data sequentially.

Consider the merge sort algorithm on linked list; following the very popular tortoise and hare algorithm, the list is divided until it cannot be divided further.



Then, the nodes in the list are sorted (conquered). These nodes are then combined (or merged) in recursively until the final solution is achieved.



Various searching algorithms can also be performed on the linked list data structures with a slightly different technique as linked lists are not indexed linear data structures. They must be handled using the pointers available in the nodes of the list.

Examples

The following computer algorithms are based on **divide-and-conquer** programming approach −

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest Pair

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in a sorted form.

Implementation in C

```
#include <stdio.h>

#define MAX 20

// array of items on which linear search will be conducted.
int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {
   int i;

   for(i = 0;i <count-1;i++) {
      printf("=");
   }

   printf("=\n");
```

```
}

int find(int data) {
   int lowerBound = 0;
   int upperBound = MAX -1;
   int midPoint = -1;
   int comparisons = 0;
   int index = -1;

   while(lowerBound <= upperBound) {
      printf("Comparison %d\n" , (comparisons +1) );
      printf("lowerBound : %d, intArray[%d] = %d\n",lowerBound,lowerBound,
         intArray[lowerBound]);
      printf("upperBound : %d, intArray[%d] = %d\n",upperBound,upperBound,
         intArray[upperBound]);
      comparisons++;

      // compute the mid point
      // midPoint = (lowerBound + upperBound) / 2;
      midPoint = lowerBound + (upperBound - lowerBound) / 2;

      // data found
      if(intArray[midPoint] == data) {
         index = midPoint;
         break;
      } else {
         // if data is larger
         if(intArray[midPoint] < data) {
            // data is in upper half
            lowerBound = midPoint + 1;
         }
         // data is smaller
         else {
            // data is in lower half
            upperBound = midPoint -1;
         }
      }
   }
   printf("Total comparisons made: %d" , comparisons);
   return index;
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i<MAX;i++) {
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void main() {
   printf("Input Array: ");
   display();
   printline(50);

   //find location of 1
   int location = find(55);
   // if element was found
   if(location != -1)
      printf("\nElement found at location: %d" ,(location+1));
   else
      printf("\nElement not found.");
}
```

Output

If we compile and run the above program then it would produce following result −

```
Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66 ]
==================================================
Comparison 1
lowerBound : 0, intArray[0] = 1
upperBound : 19, intArray[19] = 66
Comparison 2
lowerBound : 10, intArray[10] = 15
upperBound : 19, intArray[19] = 66
Comparison 3
lowerBound : 15, intArray[15] = 34
upperBound : 19, intArray[19] = 66
Comparison 4
lowerBound : 18, intArray[18] = 55
upperBound : 19, intArray[19] = 66
Total comparisons made: 4
Element found at location: 19
```

Sorting

Sorting is the process of arranging the elements either in ascending (or) descending order.

Types of sorting

C language provides five sorting techniques, which are as follows −

- Bubble sort (or) Exchange Sort
- Selection sort
- Insertion sort (or) Linear sort
- Quick sort (or) Partition exchange sort
- Merge Sort (or) External sort

Quick sort

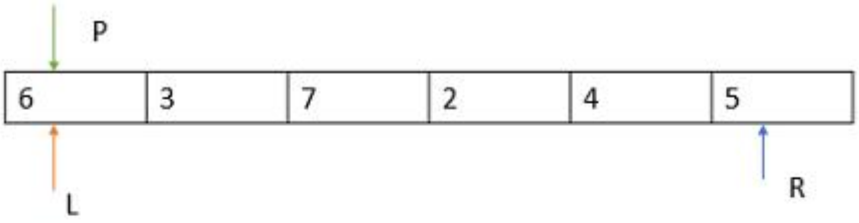
It is a divide and conquer algorithm.

- Step 1 − Pick an element from an array, call it as pivot element.
- Step 2 − Divide an unsorted array element into two arrays.
- Step 3 − If the value less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

Consider an example given below, wherein

- P is the pivot element.
- L is the left pointer.
- R is the right pointer.

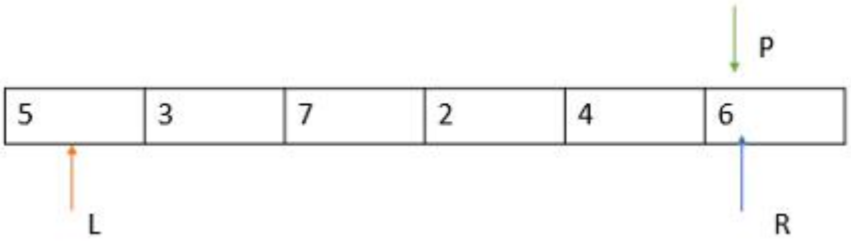
The elements are 6, 3, 7, 2, 4, 5.



6	3	7	2	4	5

Case 1: P=6 {Right side P is greater and Left side of is Leese}

Is P<R => 6<5 {wrong}
So, swap P with R

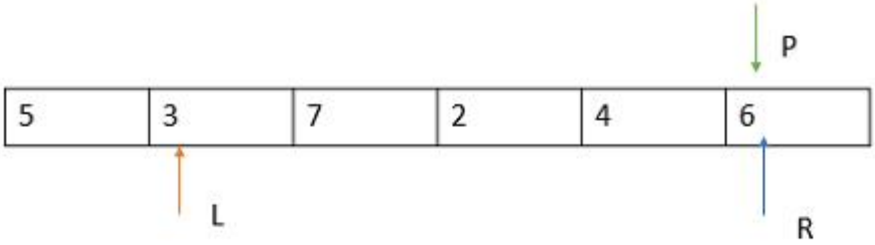


5	3	7	2	4	6

Case 2: P=6 , L=5 {Right side P is greater and Left side of is Lesser}

Is P>L => 6> 5 {right}
Move L towards right

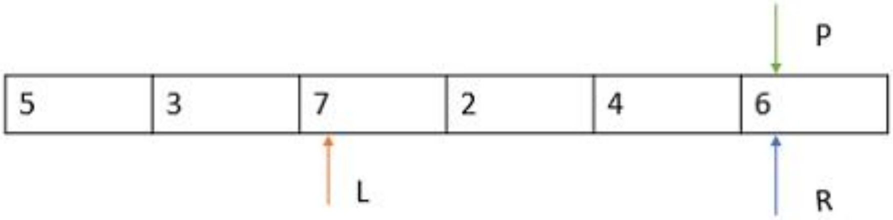
Case 3:



5	3	7	2	4	6

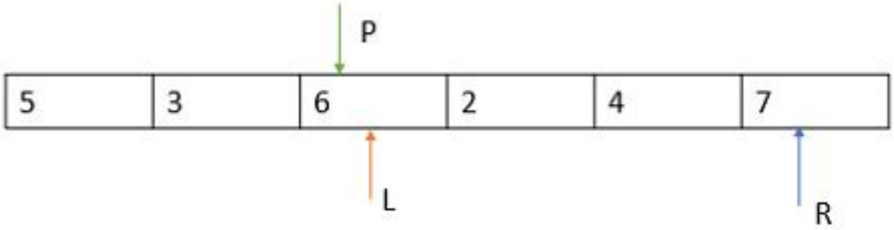
Is P>L, 6>3, Yes
So, move L towards right

Case 4:



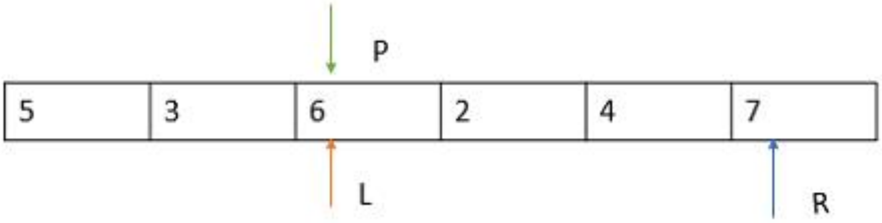
Is P>L => 6> 7 {wrong}
then swap P and L

Case5 :



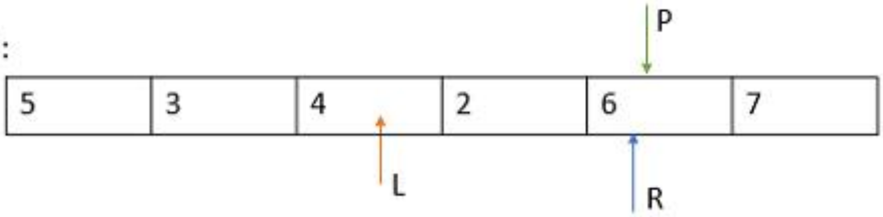
Is P<R => 6<7, => Yes
Decrement R

Case 6:



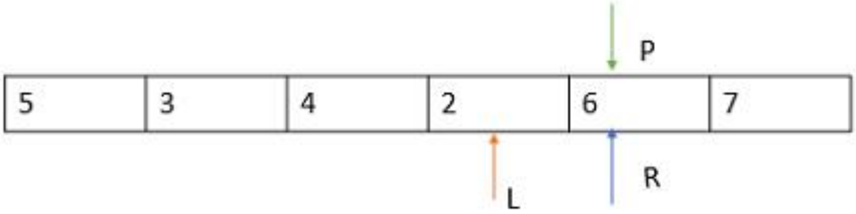
Is P<R => 6<4 {wrong}
then swap

Case 7 :



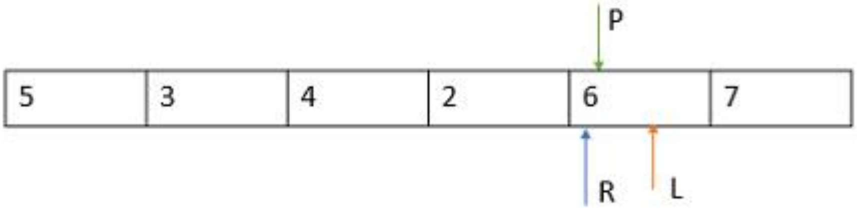
Is P>L => 6>4, => Yes
Move L to right

Case 8:



Is P>L => 6>2 {right}
move L to right

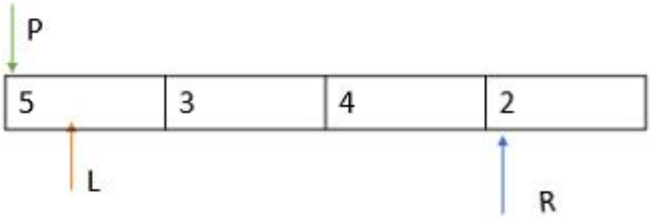
Case 9 :



Now,

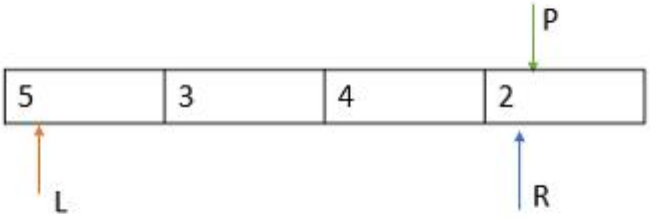
- The pivot is in fixed position.
- All the left elements are less.
- The right elements are greater than pivot.
- Now, divide the array into 2 sub arrays left part and right part.
- Take left partition apply quick sort.

Case 1:

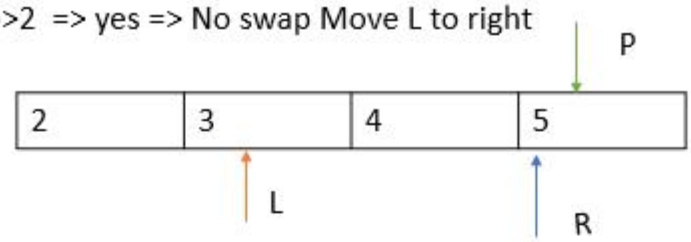


Is P<R => 5<2 {wrong} so, swap

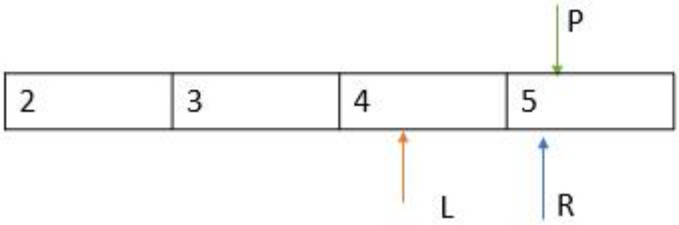
Case 2 :



Case 3: is P>L => 5>2 => yes => No swap Move L to right



Case 4: Is P> L => 5>3 => Yes => No swap => Move L to right



Now,

- The pivot is in fixed position.
- All the left elements are less and sorted
- The right elements are greater and are in sorted order.
- The final sorted list is combining two sub arrays is 2, 3, 4, 5, 6, 7

Example

Following is the C program to sort the elements by using the quick sort technique −

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
   int i, j, pivot, temp;
   if(first<last){
      pivot=first;
      i=first;
      j=last;
      while(i<j){
         while(number[i]<=number[pivot]&&i<last)
         i++;
         while(number[j]>number[pivot])
         j--;
         if(i<j){
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
         }
      }
      temp=number[pivot];
      number[pivot]=number[j];
      number[j]=temp;
      quicksort(number,first,j-1);
      quicksort(number,j+1,last);
   }
}
int main(){
   int i, count, number[25];
   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);
   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
   scanf("%d",&number[i]);
   quicksort(number,0,count-1);
   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
   printf(" %d",number[i]);
   return 0;
}
```

Output

When the above program is executed, it produces the following output −

```
How many elements are u going to enter?: 10
Enter 10 elements: 2 3 5 7 1 9 3 8 0 4
Order of Sorted elements: 0 1 2 3 3 4 5 7 8 9
```

Merge Sort in C++ is a popular sorting algorithm that divides the input array into smaller subarrays, recursively sorts them, and then merges them to produce a sorted output in ascending or descending order.

Merge Sort Algorithm

Here’s the pseudo code for the Merge Sort algorithm in C++:

```
void Merge(int arr[], int left, int mid, int right) {
    // Merge two sorted subarrays.
    // ...
}

void MergeSort(int arr[], int left, int right) {
    if (left >= right) {
        return;
    }

    int mid = left + (right - left) / 2;
    MergeSort(arr, left, mid);
    MergeSort(arr, mid + 1, right);
    Merge(arr, left, mid, right);
}
```

The **MergeSort function** recursively divides the array into smaller parts and then merges them back together using the Merge function. The Merge function combines two sorted subarrays into a single sorted array.

How does Merge Sort work?

Let’s understand how Merge Sort works with a simple example:

Consider an unsorted array: **[38, 27, 43, 3, 9, 82, 10]**

Step 1: Divide

- The array is divided into two halves: [38, 27, 43] and [3, 9, 82, 10]

Step 2: Conquer

- Each individual subarray is sorted:
- [38, 27, 43] -> [27, 38, 43]
- [3, 9, 82, 10] -> [3, 9, 10, 82]

Step 3: Merge

- The sorted subarrays are merged: [27, 38, 43] and [3, 9, 10, 82] are merged into [3, 9, 10, 27, 38, 43, 82]

Step 4: Repeat

- The process is repeated recursively for the merged array until the entire array is sorted.
- The final sorted array is [3, 9, 10, 27, 38, 43, 82].

Merge Sort Implementation in C++

Program Explanation

1. Take input of data.
2. Call **MergeSort()** function.
3. Recursively split the array into two equal parts.
4. Split them until we get at most one element in both half.
5. Combine the result by invoking **Merge()**.
6. It combines the individually sorted data from low to mid and mid+1 to high.
7. Return to main and display the result.
8. Exit.

```
/*
 * C++ Program to Implement Merge Sort
 */

#include <iostream>

using namespace std;
```

```
// A function to merge the two half into a sorted data.
void Merge(int *a, int low, int high, int mid)
{
    // We have low to mid and mid+1 to high already sorted.
    int i, j, k, temp[high-low+1];
    i = low;
    k = 0;
    j = mid + 1;

    // Merge the two parts into temp[].
    while (i <= mid && j <= high)
    {
        if (a[i] < a[j])
        {
            temp[k] = a[i];
            k++;
            i++;
        }
else
        {
            temp[k] = a[j];
            k++;
            j++;
        }
    }

    // Insert all the remaining values from i to mid into temp[].
    while (i <= mid)
    {
        temp[k] = a[i];
        k++;
        i++;
    }

    // Insert all the remaining values from j to high into temp[].
    while (j <= high)
    {
        temp[k] = a[j];
        k++;
        j++;
    }

    // Assign sorted data stored in temp[] to a[].
    for (i = low; i <= high; i++)
{
        a[i] = temp[i-low];
    }
}

// A function to split array into two parts.
void MergeSort(int *a, int low, int high)
{
    int mid;
    if (low < high)
    {
        mid=(low+high)/2;
        // Split the data into two half.
        MergeSort(a, low, mid);
        MergeSort(a, mid+1, high);
        // Merge them to get sorted output.
        Merge(a, low, high, mid);
    }
}

int main()
{
    int n, i;
    cout<<"\nEnter the number of data element to be sorted: ";
    cin>>n;

    int arr[n];
for(i = 0; i < n; i++)
    {
        cout<<"Enter element "<<i+1<<": ";
        cin>>arr[i];
    }

    MergeSort(arr, 0, n-1);

    // Printing the sorted data.
    cout<<"\nSorted Data ";
    for (i = 0; i < n; i++)
    cout<<"->"<<arr[i];

    return 0;
}
```

Array | Order Statistics

What is Order Statistics?

The **i**th order statistics is defined as the value that comes in the **i**th position of the **N** element sequence when the sequence is sorted in increasing order.

In simple terms, the order statistics is the **i**th smallest element in the given array. Below are a few examples to understand the order statistics:

- Minimum = Order statistics **1**
- 2nd minimum = Order statistics **2**
- Median = Order statistics **floor((N+1)/2)**
- 2nd Maximum = Order Statistics **N-1**
- Maximum = Order statistics **N**

Program for Mean and median of an unsorted array

Given an unsorted array **a[]** of size **N**, the task is to find its mean and median.

Mean of an array = (sum of all elements) / (number of elements)

The **median** of a sorted array of size **N** is defined as the middle element when N is odd and average of middle two elements when N is even. Since the array is not sorted here, we sort the array first, then apply above formula.

Input: a[] = {1, 3, 4, 2, 6, 5, 8, 7}
Output: Mean = 4.5, Median = 4.5
Explanation: **S**um of the elements is 1 + 3 + 4 + 2 + 6 + 5 + 8 + 7 = 36, Mean = 36/8 = 4.5
Since number of elements are even, median is average of 4th and 5th largest elements, which means Median = (4 + 5)/2 = 4.5

Input: a[] = {4, 4, 4, 4, 4}
Output: Mean = 4, Median = 4

Approach: To solve the problem follow the below steps:

To find median:

- First, simply sort the array
- Then, check if the number of elements present in the array is even or odd
- If odd, then simply return the mid value of the array
- Else, the median is the average of the two middle values

To find Mean:

- At first, find the sum of all the numbers present in the array.
- Then, simply divide the resulted sum by the size of the array

Below is the code implementation:

```
// C program to find mean and median of
// an array
#include <stdio.h>
#include <stdlib.h>

// Function to compare two integers for qsort
int cmpfunc(const void* a, const void* b)
{
    return (*(int*)a - *(int*)b);
}

// Function for calculating mean
double findMean(int a[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];

    return (double)sum / (double)n;
}

// Function for calculating median
double findMedian(int a[], int n)
{
    // First we sort the array
    qsort(a, n, sizeof(int), cmpfunc);

    // check for even case
    if (n % 2 != 0)
        return (double)a[n / 2];

    return (double)(a[(n - 1) / 2] + a[n / 2]) / 2.0;
}

// Driver code
int main()
{
    int a[] = { 1, 3, 4, 2, 7, 5, 8, 6 };
    int N = sizeof(a) / sizeof(a[0]);

    // Function call
    printf("Mean = %f\n", findMean(a, N));
    printf("Median = %f\n", findMedian(a, N));
    return 0;
}
```

Maximum and minimum of an array using minimum number of comparisons

Given an array of size **N.** The task is to find the maximum and the minimum element of the array using the minimum number of comparisons.

Examples:

Input: arr[] = {3, 5, 4, 1, 9}
Output: Minimum element is: 1
Maximum element is: 9

Input: arr[] = {22, 14, 8, 17, 35, 3}
Output: Minimum element is: 3
Maximum element is: 35

Maximum and minimum of an array using Sorting:

One approach to find the maximum and minimum element in an array is to first sort the array in ascending order. Once the array is sorted, the first element of the array will be the minimum element and the last element of the array will be the maximum element.

Step-by-step approach:

- Initialize an array.
- Sort the array in ascending order.

- The first element of the array will be the minimum element.
- The last element of the array will be the maximum element.
- Print the minimum and maximum element.

Below is the implementation of the above approach:

```
// C++ implementation of the above approach
#include <bits/stdc++.h>
using namespace std;

struct Pair {
    int min;
    int max;
};

Pair getMinMax(int arr[], int n)
{
    Pair minmax;

    sort(arr, arr + n);

    minmax.min = arr[0];
    minmax.max = arr[n - 1];

    return minmax;
}

int main()
{
    int arr[] = { 1000, 11, 445, 1, 330, 3000 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    Pair minmax = getMinMax(arr, arr_size);
    cout << "Minimum element is " << minmax.min << endl;
    cout << "Maximum element is " << minmax.max << endl;

    return 0;
}

// This code is contributed by Tapesh(tapeshdua420)
```

Learn Data Structures & Algorithms with GeeksforGeeks

Output

```
Minimum element is 1
Maximum element is 3000
```

Time complexity: O(n log n), where n is the number of elements in the array, as we are using a sorting algorithm.
Auxilary Space: is O(1), as we are not using any extra space.

Median of two Sorted arrays of equal size

- Find the union of the given two arrays.
- Sort both array 1 and array2 (Using inbuilt sort() function).
- Then the median element will be
  Median = (arr1[n-1]+arr2[0])/2

Time and Space Complexities :

- Time-Complexity :**O(nlogn)**
- Space-Complexity : **O(1)**

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

int getMedian(int ar1[], int ar2[], int n)
{
    int j = 0;
    int i = n - 1;
    while (ar1[i] > ar2[j] && j < n && i > -1)
        swap(ar1[i--], ar2[j++]);

    sort(ar1, ar1 + n);
    sort(ar2, ar2 + n);

    return (ar1[n - 1] + ar2[0]) / 2;
}


```

```



int main()
{
    int n;
    cin>>n;

    int arr1[n], arr2[n];
    for(int i=0; i<n; i++)
        cin>>arr1[i];

    for(int i=0; i<n; i++)
        cin>>arr2[i];

    cout<<getMedian(arr1, arr2, n);

    return 0;
}
```

Practice Questions

- Find Second largest element in an array
- Find the smallest and second smallest elements in an array

- K’th Smallest/Largest Element in Unsorted Array
- Maximum sum such that no two elements are adjacent