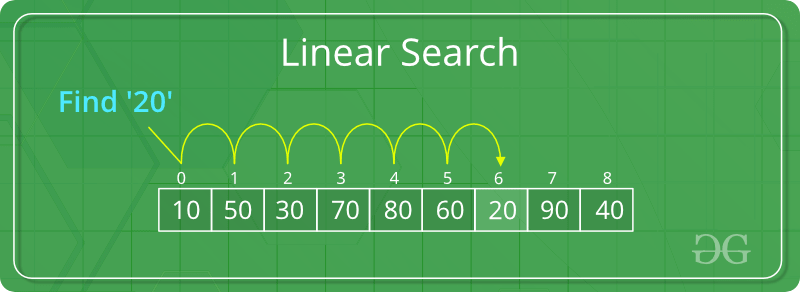
**Topic 1: Linear Search**

**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 the end of the array is reached.  
  
  
  
**Examples**:

**Input** : arr[] = {10, 20, 80, 30, 60, 50,   
 110, 100, 130, 170}  
 key = 110;  
**Output** : 6  
Element 110 is present at index 6  
  
**Input** : arr[] = {10, 20, 80, 30, 60, 50,   
 110, 100, 130, 170}  
 key = 175;  
**Output** : -1  
Element 175 is not present in arr[].

**Problem**: Given an array **arr[]** of **N** elements, write a function to search a given element **X** in arr[].  
  
**Algorithm**:

* Start from the leftmost element of arr[] and one by one compare X with each element of arr[].
* If X matches with an element, return the index.
* If X doesn’t match with any of the elements and end of the array is reached, return -1.

**Function**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

// Function to linearly search the element X in the

// array arr[] of N elements

int search(int arr[], int N, int X)

{

// Pointer to traverse the array

int i;

// Start traversing the array

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

{

// If a successful match is found,

// return the index

if (arr[i] == X)

return i;

}

// If the element is not found,

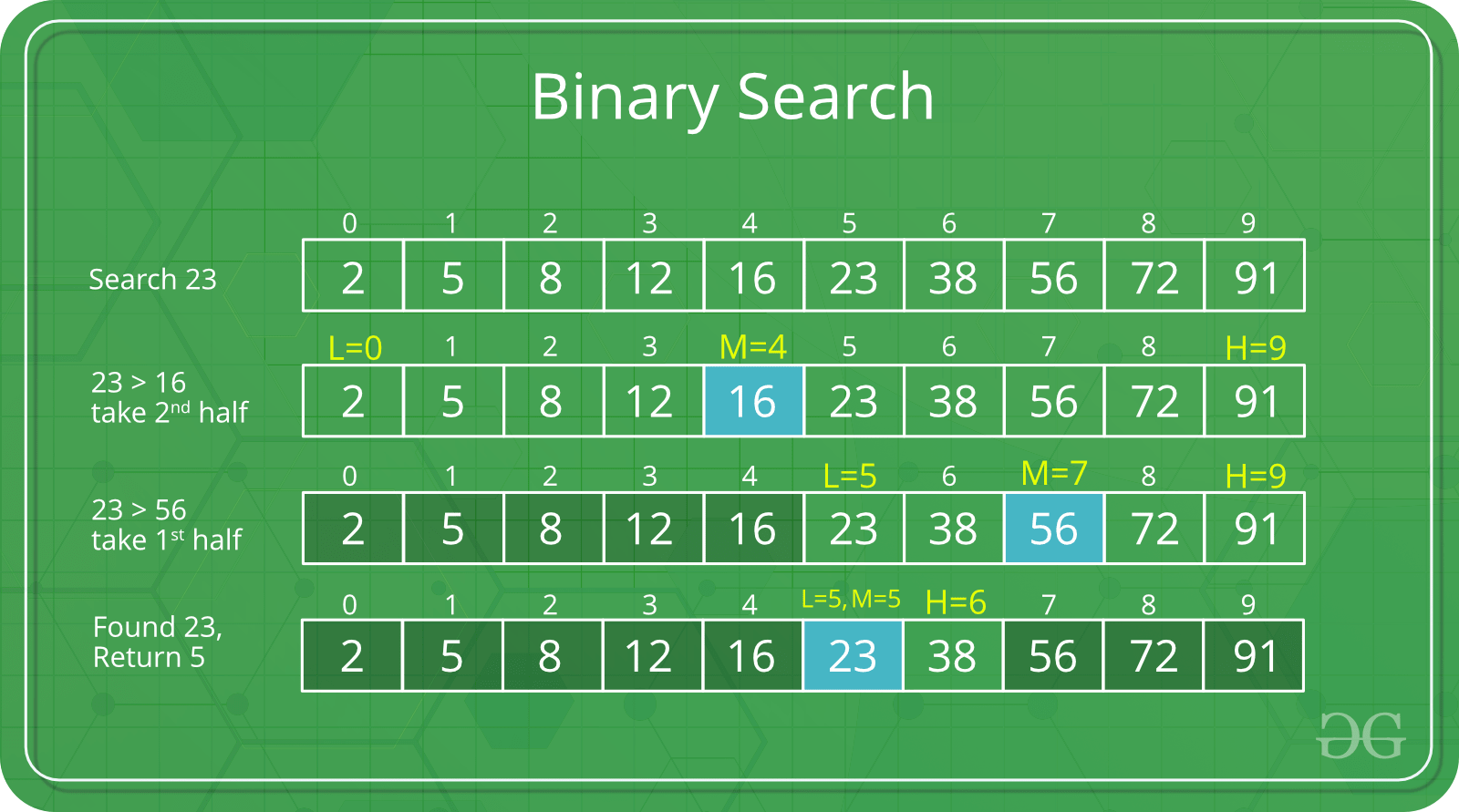
// and end of array is reached

return -1;

}

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

**Topic 2: Binary Search**

**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.  
  


**Problem**: Given a sorted array arr[] of N elements, write a function to search a given element X in arr[] using the Binary *Search Algorithm*.  
  
**Algorithm**: We basically ignore half of the elements just after one comparison.

* Compare X with the middle element of the array.
* If X matches with the middle element, we return the mid index.
* Else If X is greater than the mid element, then X can only lie in the right half subarray after the mid element. So we will now look for X in only the right half ignoring the complete left half.
* Else if X is smaller, search for X in the left half ignoring the right half.

**Implementation**: The Binary Search algorithm can be implemented both recursively and iteratively.

* **Recursive Function**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A recursive binary search function. It returns

// location of x in given array arr[l..r] if present,

// otherwise -1

// Initially,

// l = 0, first index of arr[].

// r = N-1, last index of arr[].

int binarySearch(int arr[], int l, int r, int x)

{

if (r >= l) {

int mid = l + (r - l) / 2;

// If the element is present at the middle

// itself

if (arr[mid] == x)

return mid;

// If element is smaller than mid, then

// it can only be present in left subarray

if (arr[mid] > x)

return binarySearch(arr, l, mid - 1, x);

// Else the element can only be present

// in right subarray

return binarySearch(arr, mid + 1, r, x);

}

// We reach here when element is not

// present in array

return -1;

* **Iterative Function:**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A iterative binary search function. It returns

// location of x in given array arr[l..r] if present,

// otherwise -1

// Initially,

// l = 0, first index of arr[].

// r = N-1, last index of arr[].

int binarySearch(int arr[], int l, int r, int x)

{

while (l <= r) {

int m = l + (r - l) / 2;

// Check if x is present at mid

if (arr[m] == x)

return m;

// If x greater, ignore left half

if (arr[m] < x)

l = m + 1;

// If x is smaller, ignore right half

else

r = m - 1;

}

// if we reach here, then element was

// not present

return -1;

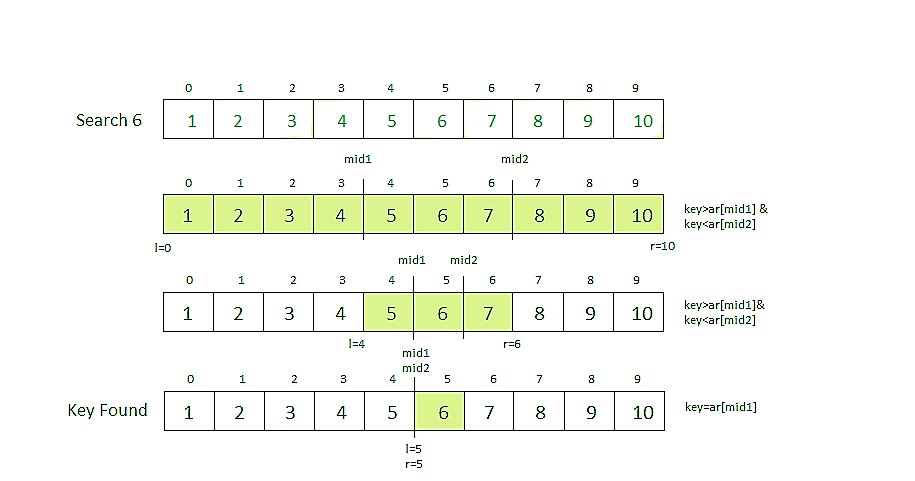
}

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

**Topic 3: Ternary Search**

**Ternary Search** is a Divide and Conquer Algorithm used to perform search operations in a **sorted array**. This algorithm is similar to the Binary Search algorithm but rather than dividing the array into two parts, it divides the array into three equal parts.  
  
In this algorithm, the given array is divided into three parts and the key (element to be searched) is compared to find the part in which it lies and that part is further divided into three parts.  
  
We can divide the array into three parts by taking mid1 and mid2 which can be calculated as shown below. Initially, l and r will be equal to 0 and N-1 respectively, where N is the length of the array.

mid1 = l + (r-l)/3  
mid2 = r – (r-l)/3

  
  
**Note**: The array must be sorted in order to perform the Binary Search or Ternary Search operation.  
  
**Steps to perform Ternary Search:**

* First, we compare the key with the element at mid1. If found equal, we return mid1.
* If not, then we compare the key with the element at mid2. If found equal, we return mid2.
* If not, then we check whether the key is less than the element at mid1. If yes, then recur to the first part.
* If not, then we check whether the key is greater than the element at mid2. If yes, then recur to the third part.
* If not, then we recur to the second (middle) part.

**Implementation**: The Ternary Search Algorithm can be implemented in both recursive and iterative manner. Below is the implementation of both recursive and iterative functions to perform Ternary Search on an array *arr[]* of size *N*to search an element *key*.

* **Recursive Function**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Recursive Function to perform Ternary Search

// Initially,

// l = 0, starting index of array.

// r = N-1, ending index of array.

int ternarySearch(int l, int r, int key, int ar[])

{

if (r >= l)

{

// Find mid1 and mid2

int mid1 = l + (r - l) / 3;

int mid2 = r - (r - l) / 3;

// Check if key is present at any mid

if (ar[mid1] == key)

{

return mid1;

}

if (ar[mid2] == key)

{

return mid2;

}

// Since key is not present at mid,

// check in which region it is present

// then repeat the Search operation

// in that region

if (key < ar[mid1])

{

// The key lies in between l and mid1

* **Iterative Function**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Iterative Function to perform Ternary Search

// Initially,

// l = 0, starting index of array.

// r = N-1, ending index of array.

int ternarySearch(int l, int r, int key, int ar[])

{

while (r >= l) {

// Find mid1 and mid2

int mid1 = l + (r - l) / 3;

int mid2 = r - (r - l) / 3;

// Check if key is present at any mid

if (ar[mid1] == key) {

return mid1;

}

if (ar[mid2] == key) {

return mid2;

}

// Since key is not present at mid,

// check in which region it is present

// then repeat the Search operation

// in that region

if (key < ar[mid1]) {

// The key lies in between l and mid1

r = mid1 - 1;

}

**Time Complexity**: O(Log3N), where N is the number of elements in the array.

Topic 4: Binary Search Functions in C++ STL

So far, we have discussed the Binary Search algorithm and its implementation by writing a function. The C++ standard template library has some built-in functions that can be used to perform Binary Search operation directly on a sequential list or container.  
  
Some of these functions are:

* **binary\_search()**
* **upper\_bound()**
* **lower\_bound()**

binary\_search()

This function only checks if a particular element is present in a sorted container or not. It accepts the starting iterator, ending iterator and the element to be checked as parameters and returns True if the element is present otherwise False.  
  
**Syntax**:

binary\_search(start\_ptr, end\_ptr, ele)

Below program illustrate the working of binary\_search() function with both Arrays and Vectors:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ code to demonstrate the working

// of binary\_search()

#include<bits/stdc++.h>

using namespace std;

int main()

{

/\*\*\* USING binary\_search() ON VECTOR \*\*\*/

// initializing vector of integers

vector<int> vec = {10, 15, 20, 25, 30, 35};

// using binary\_search to check if 15 exists

if (binary\_search(vec.begin(), vec.end(), 15))

cout << "15 exists in vector";

else

cout << "15 does not exist";

cout << endl;

/\*\*\* USING binary\_search() ON ARRAYS \*\*\*/

int arr[] = {10, 15, 20, 25, 30, 35};

int n = sizeof(arr)/sizeof(arr[0]);

// using binary\_search to check if 20 exists

if (binary\_search(arr, arr + n, 20))

cout << "20 exists in Array";

else

cout << "20 does not exist";

Run

**Output**:

15 exists in vector  
20 exists in Array

**Note**: This function only checks if the element is present or not, it does not give any information about the location of the element if it exists.

upper\_bound()

The upper\_bound() function is used to find the upper bound of an element present in a container. That is it finds the location of an element just greater than a given element in a container. This function accepts the start iterator, end iterator and the element to be checked as parameters and returns the iterator pointing to the element just greater than the element passed as the parameter. If there does not exist any such element then the function returns an iterator pointing to the last element.  
  
**Syntax**:

upper\_bound(first\_itr, last\_itr, ele)

**Return Value**: Returns an iterator pointing to the element just greater than *ele*.  
  
**Examples**:

**Input** : 10 20 30 30 40 50  
**Output** : upper\_bound for element 30 will return   
 an iterator pointing to the element 40.  
  
**Input** : 10 20 30 40 50  
**Output** : upper\_bound for element 45 will return   
 an iterator pointing to the element 50.  
  
**Input** : 10 20 30 40 50  
**Output** : upper\_bound for element 60 will   
 return end iterator.

**Note**: We can calculate the exact index position of the elements by subtracting the beginning iterator from the returned iterator.  
  
Below program illustrate the working of upper\_bound() function with both Vectors and Arrays:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// CPP program to illustrate using upper\_bound()

// with vectors and arrays

#include <bits/stdc++.h>

using namespace std;

// Driver code

int main()

{

/\*\*\* USING upper\_bound() WITH VECTOR \*\*\*/

vector<int> v{ 10, 20, 30, 40, 50 };

// Print vector

cout << "Vector contains :";

for (int i = 0; i < v.size(); i++)

cout << " " << v[i];

cout << "\n";

vector<int>::iterator upper1, upper2;

// std :: upper\_bound

upper1 = upper\_bound(v.begin(), v.end(), 35);

upper2 = upper\_bound(v.begin(), v.end(), 45);

cout << "\nUpper\_bound for element 35 is at position : "

<< (upper1 - v.begin());

cout << "\nUpper\_bound for element 45 is at position : "

<< (upper2 - v.begin())<<"\n\n";

Run

**Output**:

Vector contains : 10 20 30 40 50  
  
Upper\_bound for element 35 is at position : 3  
Upper\_bound for element 45 is at position : 4  
  
Array contains : 10 20 30 40 50  
  
upper\_bound for element 35 is at position : 3  
upper\_bound for element 45 is at position : 4

lower\_bound()

The lower\_bound() function is used to find the lower bound of an element present in a container. That is it finds the location of an element just smaller than a given element in a container. This function accepts the start iterator, end iterator and the element to be checked as parameters and returns the iterator pointing to the lower bound of the element passed as the parameter. If all elements of the container are smaller and less than the element passed, then it returns the last iterator.  
  
**Syntax**:

lower\_bound(first\_itr, last\_itr, ele)

**Return Value**: Returns an iterator pointing to the lower bound of the element *ele*. That is if *ele*exists in the container, it returns an iterator pointing to *ele*otherwise it returns an iterator pointing to the element just greater than ele.  
  
Below program illustrate the working of lower\_bound() function with both Vectors and Arrays:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// CPP program to illustrate lower\_bound()

// for both vectors and array

#include <bits/stdc++.h>

using namespace std;

// Driver code

int main()

{

/\*\*\* USING lower\_bound() ON VECTORS \*\*\*/

vector<int> v{ 10, 20, 30, 40, 50 };

// Print vector

cout << "Vector contains :";

for (int i = 0; i < v.size(); i++)

cout << " " << v[i];

cout << "\n";

vector<int>::iterator low1, low2;

// using lower\_bound()

low1 = lower\_bound(v.begin(), v.end(), 20);

low2 = lower\_bound(v.begin(), v.end(), 55);

cout << "\nlower\_bound for element 20 at position : "

<< (low1 - v.begin());

cout << "\nlower\_bound for element 55 at position : "

<< (low2 - v.begin());

Run

**Output**:

Vector contains : 10 20 30 40 50  
  
lower\_bound for element 20 at position : 1  
lower\_bound for element 55 at position : 5  
  
Array contains : 10 20 30 40 50  
  
lower\_bound for element 20 is at position : 1  
lower\_bound for element 55 is at position : 5

**Topic 5: BinarySearch using Built-in methods in Java**

If you are a Java programmer, you must have used built-in methods in Java at some point for basic operations like sorting, reversing etc. Java also provides us methods to perform Binary Search on both Arrays and Collection classes. The most commonly used methods in Java to perform Binary Search are:

* **Arrays.binarySearch()**
* **Collections.binarySearch()**

*Let's look at each of the above two methods in details*:

Arrays.binarySearch()

**Arrays.binarySearch()** is the simplest and most efficient method to find an element in a sorted array in Java  
  
**Declaration:**

public static int binarySearch(data\_type arr, data\_type key )

Where **data\_type** can be any of the primitive data types: *byte*, *char*, *double*, *int*, *float*, *short*, *long* and *Object* as well.  
  
**Description:** This method searches the specified array of the given data type for the specified value using the binary search algorithm. The array must be sorted prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.  
  
**Parameters:**

* arr - the array to be searched
* key - the value to be searched for

**Return Value:** It returns the index of the search key, if it is contained in the array; otherwise, (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or a.length if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.  
  
**Examples:**

Searching for 35 in byteArr[] = {10,20,15,22,35}  
will give result as 4 as it is the index of 35  
  
Searching for 35 in charArr[] = {'g','p','q','c','i'}  
will give result as 1 as it is the index of 'g'  
  
Searching for 22 in intArr[] = {10,20,15,22,35};  
will give result as 3 as it is the index of 22  
  
Searching for 1.5 in doubleArr[] = {10.2,15.1,2.2,3.5}  
will give result as -1 as it is the insertion point of 1.5  
  
Searching for 35.0 in floatArr[] = {10.2f,15.1f,2.2f,3.5f}  
will give result as -5 as it is the insertion point of 35.0  
  
Searching for 5 in shortArr[] = {10,20,15,22,35}  
will give result as -1 as it is the insertion point of 5

**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Java program to demonstrate working of Arrays.

// binarySearch() in a sorted array

import java.util.Arrays;

public class GFG

{

public static void main(String[] args)

{

byte byteArr[] = {10,20,15,22,35};

char charArr[] = {'g','p','q','c','i'};

int intArr[] = {10,20,15,22,35};

double doubleArr[] = {10.2,15.1,2.2,3.5};

float floatArr[] = {10.2f,15.1f,2.2f,3.5f};

short shortArr[] = {10,20,15,22,35};

Arrays.sort(byteArr);

Arrays.sort(charArr);

Arrays.sort(intArr);

Arrays.sort(doubleArr);

Arrays.sort(floatArr);

Arrays.sort(shortArr);

byte byteKey = 35;

char charKey = 'g';

int intKey = 22;

double doubleKey = 1.5;

float floatKey = 35;

short shortKey = 5;

Run

**Output:**

35 found at index = 4  
g found at index = 1  
22 found at index = 3  
1.5 found at index = -1  
35.0 found at index = -5  
5 found at index = -1

**Important Points:**

* If input list is not sorted, the results are undefined.
* If there are duplicates, there is no guarantee which one will be found.

Collections.binarySearch()

The Collections.binarySearch() method is a Collections class method in Java that returns the position of an object in a sorted list.  
  
**Declaration**:

// Returns index of key in sorted list sorted in  
// ascending order  
public static int binarySearch(List slist, T key)  
  
// Returns index of key in sorted list sorted in  
// order defined by Comparator c.  
public static int binarySearch(List slist, T key, Comparator c)  
  
If the key is not present, the it returns "(-(insertion point) - 1)".   
The insertion point is defined as the point at which the key   
would be inserted into the list.

The method throws **ClassCastException** if elements of the list are not comparable using the specified comparator, or the search key is not comparable with the elements.  
  
**Searching an int key in a list sorted in ascending order:**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

// Java program to demonstrate working of Collections.

// binarySearch()

import java.util.List;

import java.util.ArrayList;

import java.util.Collections;

public class GFG

{

public static void main(String[] args)

{

List al = new ArrayList();

al.add(1);

al.add(2);

al.add(3);

al.add(10);

al.add(20);

// 10 is present at index 3.

int index = Collections.binarySearch(al, 10);

System.out.println(index);

// 13 is not present. 13 would have been inserted

// at position 4. So the function returns (-4-1)

// which is -5.

index = Collections.binarySearch(al, 15);

System.out.println(index);

}

}

Run

**Output**:

3  
-5

**Searching an int key in a list sorted in descending order.**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

// Java program to demonstrate working of Collections.

// binarySearch() in an array sorted in descending order.

import java.util.List;

import java.util.ArrayList;

import java.util.Collections;

public class GFG

{

public static void main(String[] args)

{

List al = new ArrayList();

al.add(100);

al.add(50);

al.add(30);

al.add(10);

al.add(2);

// The last parameter specifies the comparator method

// used for sorting.

int index = Collections.binarySearch(al, 50,

Collections.reverseOrder());

System.out.println("Found at index " + index);

}

}

Run

**Output**:

Found at index 1

**Note**: Arrays.binarysearch() works for arrays which can be of primitive data type also. Collections.binarysearch() works for objects Collections like ArrayList and LinkedList.

**Topic 6 : Sample problems on Searching**

**Problem #1 : Missing and Repeating Number**

**Description:**Given an unsorted array of size **n**. Array elements are in the range from **1 to n**. One number from set {1, 2, …n} is missing and one number occurs twice in the array. Our task is to find these two numbers.

**Input**  
[2, 3, 2, 1, 5]  
**Output**  
4 2

1. **Solution : Use Sorting** Follow the given steps-

1) Sort the input array.  
 2) Traverse the array and check for missing and repeating.

**Time Complexity :** O(nLogn)  
**Auxiliary Space:**O(1)

1. **Solution : Make two equations using Sum and Product**
   1. Let x be the missing and y be the repeating element.
   2. Get the the sum of Array using formula **S = n(n+1)/2 - x + y**
   3. Get product of Array using formula **P = 1\*2\*3\*...\*n \* y / x**
   4. The above two steps give us two equations, we can solve the equations and get the values of x and y.

**Example Array** : [2, 3, 2, 1, 5]  
S = 13  
(n \*(n+1))/2 = 15  
13 = 15 - x + y  
x - y = 2 ....  **1**  
P = 60  
1\*2\*3..n = 120  
60 = (120\*y)/x  
x = 2y .... **2**  
Solving Equation 1 and 2 --   
x = 4 and y = 2

**Time Complexity:** O(n)  
**Auxiliary Space :** O(1)  
  
**Note:** This method can cause arithmetic overflow as we calculate the product and sum of all array elements. Can you avoid this?

1. **Solution : Use Hashing** We can create a auxiliary array to count the elements in the Array. We traverse the auxiliary array for finding missing and repeating numbers in the array. Can we optimize the space ?  
   **Pseudo Code**

//n : size of array  
void repeating\_missing(arr, n)  
{  
 count[n+1] = {0}  
 for (i=0 to n-1 )  
 count[a[i]]++  
  
 for (i=1 to n) {  
 if (count[i] == 0 )  
 missing = i  
 if (count[i] == 2 )  
 repeating = i  
 }  
 print(repeating, missing)  
}

**Time Complexity:** O(n)  
**Auxiliary Space :** O(n)

1. **Solution : Use Negative Indexing** Traverse the array. While traversing, use the absolute value of every element as an index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.  
   **Pseudo Code**

//n : size of array  
void repeating\_missing(arr, n)  
{  
 for ( i=0 to n-1 ) {  
 temp = arr[abs(arr[i])- 1]  
 if (temp < 0 ) {  
 repeating = abs(arr[i])  
 break  
 }  
 arr[abs(arr[i])- 1] = -arr[abs[arr(i)]- 1]  
 }  
  
 for (i=0 to n-1) {  
 if (arr[i] > 0 )  
 missing = i+1  
 }  
 print(repeating, missing)  
}

**Time Complexity:** O(n)  
**Auxiliary Space :** O(1)

**Problem #2 : Count number of Occurrences in Sorted Array**

**Description -** Given a sorted array **arr[ ]** and a number **x**, We have to count the occurrences of **x**in arr[ ].

Input : [1, 1, 2, 2, 2, 2, 3] , x = 2  
Output : 4

**Solution : Linear Search** We can traverse the array and count the number of occurrences of x in the given input array.  
**Time Complexity :** O(n)  
Since Array is sorted, can we optimize the solution using binary search.  
  
**Solution: Binary Search** We can solve this problem using binary search by reducing the effective search space in each step. We will be using these steps -

1. Use Binary search to get the index of the first occurrence of x in arr[ ]. Let the index of the first occurrence be i.
2. Use Binary search to get the index of the last occurrence of x in arr[ ]. Let the index of the last occurrence be j.
3. Return the count as difference between first and last indices (j – i + 1);

**Pseudo Code**

int first\_index(arr, low, high, x, n)  
{  
  
 if(high >= low)   
 {  
 mid = (low + high)/2 /\*low + (high - low)/2\*/  
 if( ( mid == 0 || x > arr[mid-1]) && arr[mid] == x) :  
 return mid  
 else if(x > arr[mid]) :  
 return first\_index(arr, (mid + 1), high, x, n)  
 else :  
 return first\_index(arr, low, (mid -1), x, n)  
 }  
}  
  
int last\_index(arr, low, high, x, n):  
{  
 if (high >= low)   
 {  
 int mid = (low + high)/2 /\*low + (high - low)/2\*/  
 if( ( mid == n-1 || x < arr[mid+1]) && arr[mid] == x )   
 return mid  
 else if(x < arr[mid]) :  
 return last\_index(arr, low, (mid -1), x, n)  
 else :  
 return last\_index(arr, (mid + 1), high, x, n)  
 }  
}  
  
int count\_occurences(arr, n, x)   
{  
 i = first\_index(arr, 0, n-1, x, n)  
 j = last\_index(arr, 0, n-1, x, n)  
 count = j-i + 1  
 return count  
}

**Time Complexity :** O(Log(n))

**Problem #3 : Find the index of first 1 in a sorted array of 0’s and 1’s**

**Description -** We are given an sorted boolean array, We have to find out the index of first 1 in the Array

Input : arr[] = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1]  
Output : 6  
The index of first 1 in the array is 6.

**Solution -** One simple solution can traverse the Array and find out the first index of 1. Since the array is sorted, we can optimize the solution using binary search by reducing the effective search space in each step.  
**Pseudo Code**

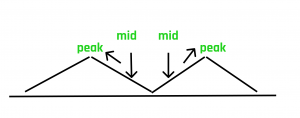
int indexOfFirstOne(arr[], low, high)   
{  
 while (low <= high)   
 {  
 int mid = (low + high) / 2  
  
 if (arr[mid] == 1 && (mid == 0 || arr[mid - 1] == 0)) :  
 return mid  
 else if (arr[mid] == 1) :  
 high = mid - 1  
 else :  
 low = mid + 1  
 }  
}

**Time Complexity :** O(Log(n))

**Problem #4 : Find Peak element in Sorted Array**

**Description -** We are given an array of distinct integers. We have to find the peak element ( The element which is greater than both the neighbours ). There can be many such elements we need to return any of them.

Input : [10, 20, 15, 2, 23, 90, 67]  
Output : 20 or 90

**Solution :**A simple solution is to traverse the array and as soon as we find a peak element, we return it. The worst case time complexity of this method would be O(n).Can we find a peak element in worst time complexity better than O(n)?  
We can use the Divide and Conquer. The idea is Binary Search-based, we compare the middle element with its neighbors. If the middle element is not smaller than any of its neighbors, then we return it. If the middle element is smaller than its left neighbor, then there is always a peak in the left half. If the middle element is smaller than its right neighbor, then there is always a peak in the right half (due to the same reason as the left half).  
  
**Pseudo Code**

int findPeak(arr[], low, high, n)   
{  
 int mid = low + (high - low)/2 /\* (low + high)/2 \*/  
   
 if ((mid == 0 || arr[mid-1] <= arr[mid]) &&   
 (mid == n-1 || arr[mid+1] <= arr[mid])) :  
 return arr[mid]  
   
 // If middle element is not peak and its left neighbour is greater   
 // than it, then left half must have a peak element   
 else if (mid > 0 && arr[mid-1] > arr[mid]) :  
 return findPeak(arr, low, (mid -1), n)  
   
 // If middle element is not peak and its right neighbour is greater   
 // than it, then right half must have a peak element   
 else :  
 return findPeak(arr, (mid + 1), high, n)  
}

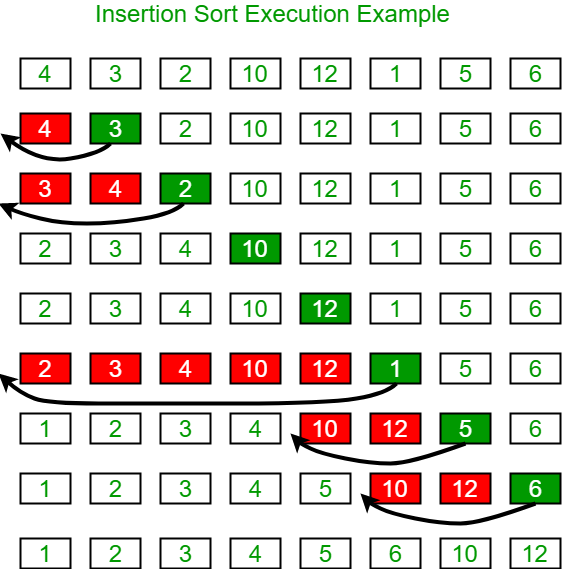
**Time Complexity :** O(Log(n))

Topic 7:Introduction to Sorting

**Sorting**any sequence means to arrange the elements of that sequence according to some specific criterion.  
  
For Example, the array arr[] = {5, 4, 2, 1, 3} after *sorting in increasing order* will be: arr[] = {1, 2, 3, 4, 5}. The same array after *sorting in descending order* will be: arr[] = {5, 4, 3, 2, 1}.  
  
**In-Place Sorting**: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.  
  
In this tutorial, we will see three of such in-place sorting algorithms, namely:

* Insertion Sort
* Selection Sort
* Bubble Sort

Insertion Sort

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.  
  
The idea is to start iterating from the second element of the array till the last element and for every element insert at its correct position in the subarray before it.  
  
In the below image you can see how the array [4, 3, 2, 10, 12, 1, 5, 6] is being sorted in increasing order following the insertion sort algorithm.  
  
  
**Algorithm**:

Step 1: If the current element is 1st element of array,   
 it is already sorted.  
Step 2: Pick next element  
Step 3: Compare the current element will all elements   
 in the sorted subarray before it.  
Step 4: Shift all of the elements in the sub-array before   
 the current element which are greater than the current   
 element by one place and insert the current element   
 at the new empty space.  
Step 5: Repeat step 2-3 until the entire array is sorted.

**Another Example**:  
arr[] = {**12**, 11, 13, 5, 6}  
  
Let us loop for i = 1 (second element of the array) to 4 (Size of input array - 1).

* *i = 1*, Since 11 is smaller than 12, move 12 and insert 11 before 12.  
  **11, 12,** 13, 5, 6
* *i = 2*, 13 will remain at its position as all elements in A[0..I-1] are smaller than 13  
  **11, 12, 13,** 5, 6
* *i = 3*, 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.  
  **5, 11, 12, 13,** 6
* *i = 4*, 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.  
  **5, 6, 11, 12, 13**

**Function Implementation:**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

/\* Function to sort an array using insertion sort\*/

void insertionSort(int arr[], int n)

{

int i, key, j;

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

{

key = arr[i];

j = i-1;

/\* Move elements of arr[0..i-1], that are

greater than key, to one position ahead

of their current position \*/

while (j >= 0 && arr[j] > key)

{

arr[j+1] = arr[j];

j = j-1;

}

arr[j+1] = key;

}

}

**Time Complexity**: O(N2), where N is the size of the array.

Bubble Sort

Bubble Sort is also an in-place sorting algorithm. This is the simplest sorting algorithm and it works on the principle that:

In one iteration if we swap all adjacent elements of an array such that after swap the first element is less than the second element then at the end of the iteration, the first element of the array will be the minimum element.

Bubble-Sort algorithm simply repeats the above steps N-1 times, where N is the size of the array.  
  
**Example:** Consider the array, arr[] = {5, 1, 4, 2, 8}.

* **First Pass:** ( **5** **1** 4 2 8 ) --> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
  ( 1 **5** **4** 2 8 ) -->  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
  ( 1 4 **5** **2** 8 ) -->  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
  ( 1 4 2 **5** **8** ) --> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
* **Second Pass:** ( **1** **4** 2 5 8 ) --> ( **1** **4** 2 5 8 )  
  ( 1 **4** **2** 5 8 ) --> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
  ( 1 2 **4** **5** 8 ) --> ( 1 2 **4** **5** 8 )  
  ( 1 2 4 **5** **8** ) -->  ( 1 2 4 **5** **8** )  
  Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
* **Third Pass:** ( **1** **2** 4 5 8 ) --> ( **1** **2** 4 5 8 )  
  ( 1 **2** **4** 5 8 ) --> ( 1 **2** **4** 5 8 )  
  ( 1 2 **4** **5** 8 ) --> ( 1 2 **4** **5** 8 )  
  ( 1 2 4 **5** **8** ) --> ( 1 2 4 **5** **8** )

**Function Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

int i, j;

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

// Last i elements are already in place

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

if (arr[j] > arr[j+1])

swap(&arr[j], &arr[j+1]);

}

**Note**: The above solution can be further optimized by keeping a flag to check if the array is already sorted in the first pass itself and to stop any further iteration.  
  
**Time Complexity**: O(N2)

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.  
  
Following example explains the above steps:

arr[] = 64 25 12 22 11.  
  
// Find the minimum element in arr[0...4]  
// and place it at beginning  
**11** 25 12 22 64  
  
// Find the minimum element in arr[1...4]  
// and place it at beginning of arr[1...4]  
11 **12** 25 22 64  
  
// Find the minimum element in arr[2...4]  
// and place it at beginning of arr[2...4]  
11 12 **22** 25 64  
  
// Find the minimum element in arr[3...4]  
// and place it at beginning of arr[3...4]  
11 12 22 **25** 64

**Function Implementation:**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

void selectionSort(int arr[], int n)

{

int i, j, min\_idx;

// One by one move boundary of unsorted subarray

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

{

// Find the minimum element in unsorted array

min\_idx = i;

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

if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first element

swap(&arr[min\_idx], &arr[i]);

}

}

**Time Complexity**: O(N2)

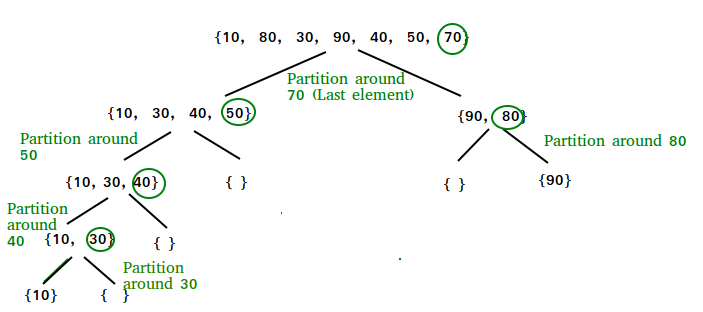
**Topic 8 : Quick Sort**

**QuickSort**is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.  
  
  
  
**Pseudo Code for recursive QuickSort function :**

/\* low --> Starting index, high --> Ending index \*/  
quickSort(arr[], low, high)  
{  
 if (low < high)  
 {  
 /\* pi is partitioning index, arr[pi] is now  
 at right place \*/  
 pi = partition(arr, low, high);  
  
 quickSort(arr, low, pi - 1); // Before pi  
 quickSort(arr, pi + 1, high); // After pi  
 }  
}

  
**Partition Algorithm**:  
There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current elements with arr[i]. Otherwise we ignore current elements.

/\* low --> Starting index, high --> Ending index \*/  
quickSort(arr[], low, high)  
{  
 if (low < high)  
 {  
 /\* pi is partitioning index, arr[pi] is now  
 at right place \*/  
 pi = partition(arr, low, high);  
  
 quickSort(arr, low, pi - 1); // Before pi  
 quickSort(arr, pi + 1, high); // After pi  
 }  
}

**Pseudo code for partition()**

/\* This function takes last element as pivot, places  
 the pivot element at its correct position in sorted  
 array, and places all smaller (smaller than pivot)  
 to left of pivot and all greater elements to right  
 of pivot \*/  
partition (arr[], low, high)  
{  
 // pivot (Element to be placed at right position)  
 pivot = arr[high];   
   
 i = (low - 1) // Index of smaller element  
  
 for (j = low; j <= high- 1; j++)  
 {  
 // If current element is smaller than or  
 // equal to pivot  
 if (arr[j] <= pivot)  
 {  
 i++; // increment index of smaller element  
 swap arr[i] and arr[j]  
 }  
 }  
  
 swap arr[i + 1] and arr[high])  
 return (i + 1)  
}

**Illustration of partition() :**

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

**j = 0** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 0**

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j

// are same

**j = 1** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 1**

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

**j = 3** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 4** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 2**

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

**j = 5** : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

**i = 3**

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

**Finally we place pivot at correct position by swapping**

**arr[i+1] and arr[high] (or pivot)**

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

it.

**Implementation:**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high- 1; j++)

{

// If current element is smaller than or

// equal to pivot

if (arr[j] <= pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

/\* The main function that implements QuickSort

arr[] --> Array to be sorted,

low --> Starting index,

high --> Ending index \*/

void quickSort(int arr[], int low, int high)

Run

**Analysis of QuickSort**

Time taken by QuickSort in general can be written as follows.

T(n) = T(k) + T(n-k-1) + Θ(n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.  
  
***Worst Case:*** The worst case occurs when the partition process always picks the greatest or smallest element as pivot. If we consider the above partition strategy where the last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + Θ(n)  
which is equivalent to   
 T(n) = T(n-1) + Θ(n)

The solution of the above recurrence is Θ(n2).  
  
***Best Case:*** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + Θ(n)

The solution of above recurrence is **Θ(nLogn)**. It can be solved using case 2 of Master Theorem.  
  
***Average Case:*** To do average case analysis, we need to consider all possible permutations of the array and calculate time taken by every permutation which doesn't look easy.  
We can get an idea of the average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in another set. Following is a recurrence for this case.

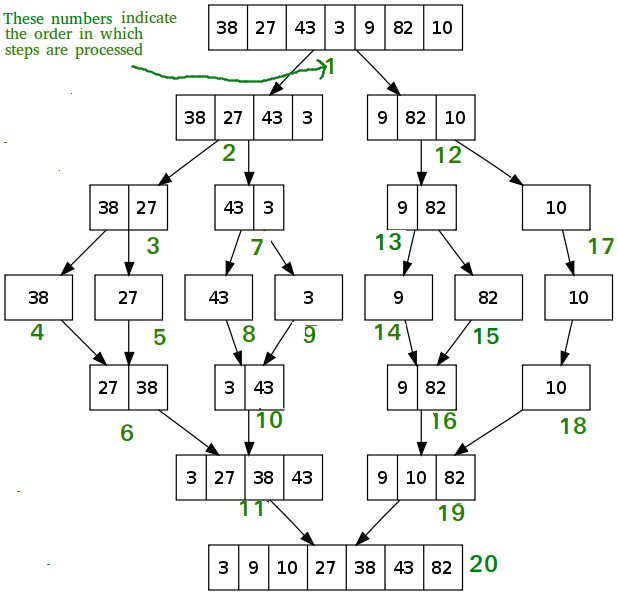
T(n) = T(n/9) + T(9n/10) + Θ(n)

Solution of above recurrence is also O(nLogn)  
  
Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like **Merge Sort** and **Heap Sort**, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**Topic 9: Merge Sort**

**Merge Sort** is a Divide and Conquer algorithm. It divides the input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one in a sorted manner. See following implementation for details:

**MergeSort(arr[], l, r)**  
If r > l  
 **1.** Find the middle point to divide the array into two halves:   
 middle m = (l+r)/2  
  **2.** Call mergeSort for first half:   
 Call mergeSort(arr, l, m)  
 **3.** Call mergeSort for second half:  
 Call mergeSort(arr, m+1, r)  
 **4.** Merge the two halves sorted in step 2 and 3:  
 Call merge(arr, l, m, r)

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge process comes into action and starts merging arrays back till the complete array is merged.  
  
  
  
**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

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

L[i] = arr[l + i];

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

R[j] = arr[m + 1+ j];

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as the following recurrence relation.  
T(n) = 2T(n/2) + Θ(n)  
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is Θ(nLogn).  
  
Time complexity of Merge Sort is **Θ(nLogn)** in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.  
  
**Auxiliary Space:** O(n)

**Topic 10 : Counting Sort**

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.  
  
Let us understand it with the help of an example.

For simplicity, consider the data in the range 0 to 9.   
Input data: 1, 4, 1, 2, 7, 5, 2  
 1) Take a count array to store the count of each unique object.  
 Index: 0 1 2 3 4 5 6 7 8 9  
 Count: 0 2 2 0 1 1 0 1 0 0  
  
 2) Modify the count array such that each element at each index   
 stores the sum of previous counts.   
 Index: 0 1 2 3 4 5 6 7 8 9  
 Count: 0 2 4 4 5 6 6 7 7 7  
  
The modified count array indicates the position of each object in   
the output sequence.  
   
 3) Output each object from the input sequence followed by   
 decreasing its count by 1.  
 Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.  
 Put data 1 at index 2 in output. Decrease count by 1 to place   
 next data 1 at an index 1 smaller than this index.

**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// The main function that sort the given string arr[] in

// alphabetical order

void countSort(char arr[])

{

// The output character array that will have sorted arr

char output[strlen(arr)];

// Create a count array to store count of inidividul

// characters and initialize count array as 0

int count[RANGE + 1], i;

memset(count, 0, sizeof(count));

// Store count of each character

for(i = 0; arr[i]; ++i)

++count[arr[i]];

// Change count[i] so that count[i] now contains actual

// position of this character in output array

for (i = 1; i <= RANGE; ++i)

count[i] += count[i-1];

// Build the output character array

for (i = 0; arr[i]; ++i)

{

output[count[arr[i]]-1] = arr[i];

--count[arr[i]];

}

// Copy the output array to arr, so that arr now

// contains sorted characters

**Time Complexity:** O(N + K) where N is the number of elements in the input array and K is the range of input.  
**Auxiliary Space:** O(N + K)  
  
The problem with the previous counting sort was that it could not sort the elements if we have negative numbers in the array because there are no negative array indices. So what we can do is, we can find the minimum element and store the count of that minimum element at zero index.  
  
**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

void countSort(vector <int>& arr)

{

int max = \*max\_element(arr.begin(), arr.end());

int min = \*min\_element(arr.begin(), arr.end());

int range = max - min + 1;

vector<int> count(range), output(arr.size());

for(int i = 0; i < arr.size(); i++)

count[arr[i]-min]++;

for(int i = 1; i < count.size(); i++)

count[i] += count[i-1];

for(int i = arr.size()-1; i >= 0; i--)

{

output[ count[arr[i]-min] -1 ] = arr[i];

count[arr[i]-min]--;

}

for(int i=0; i < arr.size(); i++)

arr[i] = output[i];

}

**Important Points:**

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. It's running time complexity is O(n) with space proportional to the range of data.
3. It is often used as a subroutine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).
5. Counting sort can be extended to work for negative inputs also.

**Topic 11: Heap Sort**

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining elements.

**What is Binary Heap?**

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.  
  
A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called max heap and the latter is called min heap. The heap can be represented by a binary tree or array.  
  
**Array based representation for Binary Heap**: Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 \* I + 1 and right child by 2 \* I + 2 (assuming the indexing starts at 0).  
  
**Heap Sort Algorithm for sorting an array in increasing order:**

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat the above steps while the size of the heap is greater than 1.

**How to build the heap?**

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.  
  
Lets understand with the help of an example:

**Input data**: [4, 10, 3, 5, 1]  
 4(0)  
 / \  
 10(1) 3(2)  
 / \  
 5(3) 1(4)  
  
The numbers in bracket represent the indices in the array   
representation of data.  
  
**Applying heapify procedure to index 1**:  
 4(0)  
 / \  
 10(1) 3(2)  
 / \  
5(3) 1(4)  
  
**Applying heapify procedure to index 0**:  
 10(0)  
 / \  
 5(1) 3(2)  
 / \  
 4(3) 1(4)  
  
**The heapify procedure calls itself recursively to build heap  
in top down manner.**

**Implementation**:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

void heapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int l = 2\*i + 1; // left = 2\*i + 1

int r = 2\*i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i)

{

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// Main function for heap sort

void heapSort(int arr[], int n)

{

// Build heap (rearrange array)

**Important Notes:**

* Heap sort is an in-place algorithm.
* Its typical implementation is not stable, but can be made stable.

**Time Complexity:**Time complexity of heapify is O(N\*LogN). Time complexity of createAndBuildHeap() is O(N) and overall time complexity of Heap Sort is **O(N\*LogN)** where N is the number of elements in the list or array.  
  
Heap sort algorithm has limited use because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

**Topic 12 : Sort() Function in C++ STL**

C++ STL provides a built-in function sort() that sorts a vector or array (items with random access).  
  
**Syntax to sort an Array**:

**sort(arr, arr+n);**  
  
Here, *arr* is the name or base address of the array  
and, *n* is the size of the array.

**Syntax to sort a Vector**:

**sort(vec.begin(), vec.end());**  
  
Here, *vec* is the name of the vector.

Below program illustrate the sort function:



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

// C++ program to demonstrate default behaviour of

// sort() in STL.

#include <bits/stdc++.h>

using namespace std;

int main()

{

// Sorting Array

int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};

int n = sizeof(arr)/sizeof(arr[0]);

sort(arr, arr+n);

cout << "Array after sorting is : \n";

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

// Sorting Vector

vector<int> vec = {1,2,4,5,3};

sort(vec.begin(), vec.end());

cout << "\nVector after sorting is : \n";

for (int i = 0; i < vec.size(); ++i)

cout << vec[i] << " ";

return 0;

}

Run

**Output**:

Array after sorting is :   
0 1 2 3 4 5 6 7 8 9   
Vector after sorting is :   
1 2 3 4 5

So by default, sort() function sorts an array in ascending order.

**How to sort in descending order?**

The sort() function takes a third parameter that is used to specify the order in which elements are to be sorted. We can pass the "greater()" function to sort in descending order. This function does comparison in a way that puts a greater element before.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

// C++ program to demonstrate descending order

// sort using greater<>().

#include <bits/stdc++.h>

using namespace std;

int main()

{

int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};

int n = sizeof(arr)/sizeof(arr[0]);

sort(arr, arr+n, greater<int>());

cout << "Array after sorting : \n";

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

return 0;

}

Run

**Output**:

Array after sorting :

9 8 7 6 5 4 3 2 1 0

**How to sort in particular order?**

We can also write our own comparator function and pass it as a third parameter.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// A C++ program to demonstrate STL sort() using

// our own comparator

#include<bits/stdc++.h>

using namespace std;

// An interval has start time and end time

struct Interval

{

int start, end;

};

// Compares two intervals according to staring times.

bool compareInterval(Interval i1, Interval i2)

{

return (i1.start < i2.start);

}

int main()

{

Interval arr[] = { {6,8}, {1,9}, {2,4}, {4,7} };

int n = sizeof(arr)/sizeof(arr[0]);

// sort the intervals in increasing order of

// start time

sort(arr, arr+n, compareInterval);

cout << "Intervals sorted by start time : \n";

for (int i=0; i<n; i++)

cout << "[" << arr[i].start << "," << arr[i].end

<< "] ";

Run

**Output**:

Intervals sorted by start time :

[1,9] [2,4] [4,7] [6,8]

**Topic 13: Sorting using Built – in Methods in Java**

Arrays.sort()

The Arrays.sort() is a built-in method in Java of Arrays class which is used to sort an array in ascending or descending or any other order specified by the user.  
  
**Syntax:**

public static void **sort**(int[] arr, int from\_Index, int to\_Index)  
  
**arr** - The array to be sorted.  
**from\_Index** - The index of the first element, inclusive, to be sorted.  
**to\_Index** - The index of the last element, exclusive, to be sorted.

Below are different ways of using the sort() method of Arrays class in Java to sort arrays differently.

* **A Java program to sort an array of integers in ascending order**.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

// A sample Java program to sort an array of integers

// using Arrays.sort(). It by default sorts in

// ascending order

import java.util.Arrays;

public class SortExample

{

public static void main(String[] args)

{

// Our arr contains 8 elements

int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

Arrays.sort(arr);

System.out.printf("Modified arr[] : %s",

Arrays.toString(arr));

}

}

Run

**Output:**

Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]

* **We can also use sort() to sort a subarray of arr[]**.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

// A sample Java program to sort a subarray

// using Arrays.sort().

import java.util.Arrays;

public class SortExample

{

public static void main(String[] args)

{

// Our arr contains 8 elements

int[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

// Sort subarray from index 1 to 4, i.e.,

// only sort subarray {7, 6, 45, 21} and

// keep other elements as it is.

Arrays.sort(arr, 1, 5);

System.out.printf("Modified arr[] : %s",

Arrays.toString(arr));

}

}

Run

**Output:**

Modified arr[] : [13, 6, 7, 21, 45, 9, 2, 100]

* **We can also sort in descending order.**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

// A sample Java program to sort a subarray

// in descending order using Arrays.sort().

import java.util.Arrays;

import java.util.Collections;

public class SortExample

{

public static void main(String[] args)

{

// Note that we have Integer here instead of

// int[] as Collections.reverseOrder doesn't

// work for primitive types.

Integer[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

// Sorts arr[] in descending order

Arrays.sort(arr, Collections.reverseOrder());

System.out.printf("Modified arr[] : %s",

Arrays.toString(arr));

}

}

Run

**Output:**

Modified arr[] : [100, 45, 21, 13, 9, 7, 6, 2]

* **We can also sort strings in alphabetical order**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

// A sample Java program to sort an array of strings

// in ascending and descending orders using Arrays.sort().

import java.util.Arrays;

import java.util.Collections;

public class SortExample

{

public static void main(String[] args)

{

String arr[] = {"practice.acm",

"quiz.acm",

"code.acm"

};

// Sorts arr[] in ascending order

Arrays.sort(arr);

System.out.printf("Modified arr[] : \n%s\n\n",

Arrays.toString(arr));

// Sorts arr[] in descending order

Arrays.sort(arr, Collections.reverseOrder());

System.out.printf("Modified arr[] : \n%s\n\n",

Arrays.toString(arr));

}

}

Run

**Output:**

Modified arr[] :   
[code 1="practice.acm," 2="quiz.acm" language=".acm,"][/code]  
  
Modified arr[] :   
[quiz.acm, practice.acm, code.acm]

* **We can also sort an array according to user defined criteria**: We use Comparator interface for this purpose. Below is an example.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Java program to demonstrate working of Comparator

// interface

import java.util.\*;

// A class to represent a student.

class Point

{

int x, y;

Point(int i, int j) {x = i; y = j;}

}

class MySort implements Comparator<Point>

{

// Used for sorting in ascending order of

// roll number

public int compare(Point a, Point b)

{

return a.x - b.x;

}

}

// Driver class

class Main

{

public static void main (String[] args)

{

Point [] arr = {new Point(10, 20), new Point(3, 12), new Point(5, 7)};

Arrays.sort(arr, new MySort());

for (int i=0; i<arr.length; i++)

System.out.println(arr[i].x + " " + arr[i].y);

Run

**Output:**

3 12  
5 7  
10 20

Collections.sort()

The **Collections.sort()** method is present in the Collections class. It is used to sort the elements present in the specified list of Collection in ascending order.  
  
It works similar to the Arrays.sort() method but it is better as it can sort the elements of Array as well as any collection interfaces like a linked list, queue and many more.  
  
**Syntax**:

public static void sort(List myList)  
  
myList : A List type object we want to sort.  
  
This method doesn't return anything

**Example**:

Let us suppose that our list contains  
{"acmiitism", "Friends", "Dear", "Is", "Superb"}  
  
After using Collection.sort(), we obtain a sorted list as  
{"acmiitism", "Dear", "Friends", "Is", "Superb"}

Below are some ways of using the Collections.sort() method in Java:

* **Sorting an ArrayList in ascending order**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

// Java program to demonstrate working of Collections.sort()

import java.util.\*;

public class Collectionsorting

{

public static void main(String[] args)

{

// Create a list of strings

ArrayList<String> al = new ArrayList<String>();

al.add("acmiitism");

al.add("Friends");

al.add("Dear");

al.add("Is");

al.add("Superb");

/\* Collections.sort method is sorting the

elements of ArrayList in ascending order. \*/

Collections.sort(al);

// Let us print the sorted list

System.out.println("List after the use of" +

" Collection.sort() :\n" + al);

}

}

Run

**Output**:

List after the use of Collection.sort() :

[acmiitism, Dear, Friends, Is, Superb]

* **Sorting an ArrayList in descending order**



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

// Java program to demonstrate working of Collections.sort()

// to descending order.

import java.util.\*;

public class Collectionsorting

{

public static void main(String[] args)

{

// Create a list of strings

ArrayList<String> al = new ArrayList<String>();

al.add("acmiitism");

al.add("Friends");

al.add("Dear");

al.add("Is");

al.add("Superb");

/\* Collections.sort method is sorting the

elements of ArrayList in ascending order. \*/

Collections.sort(al, Collections.reverseOrder());

// Let us print the sorted list

System.out.println("List after the use of" +

" Collection.sort() :\n" + al);

}

}

Run

**Output**:

List after the use of Collection.sort() :

[Superb, Is, Friends, Dear, acmiitism]

* **Sorting an ArrayList according to user defined criteria**: We can use Comparator Interface for this purpose.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// Java program to demonstrate working of Comparator

// interface and Collections.sort() to sort according

// to user defined criteria.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

// A class to represent a student.

class Student

{

int rollno;

String name, address;

// Constructor

public Student(int rollno, String name,

String address)

{

this.rollno = rollno;

this.name = name;

this.address = address;

}

// Used to print student details in main()

public String toString()

{

return this.rollno + " " + this.name +

" " + this.address;

}

}

Run

**Output** :

Unsorted

111 bbbb london

131 aaaa nyc

121 cccc jaipur

Sorted by rollno

111 bbbb london

121 cccc jaipur

131 aaaa nyc