Name : Dilini Chamika Silva
Batch : GDSE – 71

Content

01. Linear Search

The linear search algorithm is another alternative name for the sequential search algorithm. It is the most straightforward type of search algorithm. In the linear search method we just cross the list completely and check for every element in the list and compare it with the data for which location is to be found. If it's found then it returns the location of the item , else it results in a 'NULL' return by the algorithm.

It's widely used to search an element from unordered list , like the list in which items are not in any proper order.

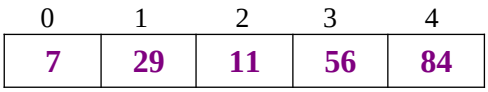
The implementation steps :

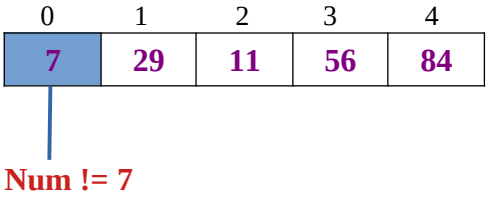
First we have to cross the array elements

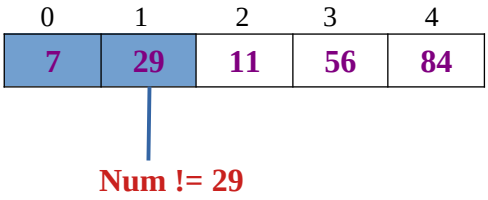
Inside every iteration of the loop compare the search element against the current array element and either,

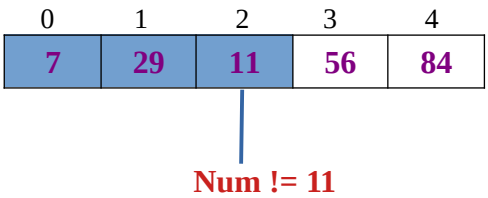
- If match found , return the index of that corresponding array element.
- If not matching , then do the comparison against the next element.
- If no match is found , or the element to be searched isn't the specified array , return -1.

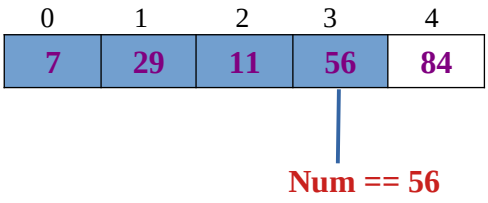
Let the element to be searched in **num = 56**











Now, the element to be searched is found. So algorithm will return the index of the element matched.

```java
import java.util.*;

class Main{
    static Scanner input = new Scanner(System.in);

    public static int linearSearchMethod(int[] array){
        System.out.print(s:"Enter a number (7 , 29 , 11 , 56 , 84) : ");
        int num =  input.nextInt();

        for (int i = 0; i < array.length; i++) {
            if(array[i] == num){
               return i;
            }

        }
        return -1;
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {7 , 29 , 11 , 56 , 84};

        int index =  linearSearchMethod(array);

        if(index != -1){
            System.out.println("The target number found  at index " + index);
        }
        else{
            System.out.println(x:"The number is not found!");
        }
    }
}
```

OUTPUT : The target number found at index 3

Let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

- **Best Case Complexity** - In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1)**.

- **Average Case Complexity** - The average case time complexity of linear search is **O(n)**.

- **Worst Case Complexity** - In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n)**.

The **time complexity** of linear search is **O(n)** because every element in the array is compared only once.

The **space complexity** of linear search is **O(1)**.

Use Case - Finding a specific item in an unsorted list or array.

➔ **EX:-**

Looking up a specific contact in a phone book or searching for a file in a directory where items are not sorted.

➢ **Advantages**

- *Simplicity* - Linear search can be easily comprehended and implemented. It's a simple algorithm with very simple logic and simple loops, making it very fitting for anyone just starting in the field of algorithms about searching.

- *No Pre-Processing Required* - Linear search can be applied to data structures that are unsorted or have no specific order as a prerequisite.

- *Versatility* - This algorithm is applied to all data structures, whether arrays, linked lists, or even collections when random access isn't possible.

- *Low Memory Usage* - Linear search has a constant space complexity of O(1) since it requires only a small number of variables to maintain the current position and the target.

- *Works with Small Dataset* - Sometimes, the simplicity of linear search can overcome the additional benefits that are brought in by the complex algorithms of search, especially when dealing with small datasets.

- *No Assumptions* - Another merit of linear search is that it does not make any assumptions about the distribution or properties of the dataset.

- **Disadvantages**

  - *Inefficiency for Large Datasets* - Linear search has a time complexity of O(n), where n is the number of elements in the dataset to be searched. As such, it can prove inefficient over large datasets because it may end up examining every element.

  - *Poor Performance* - The performance of linear search is poor compared to other advanced search algorithms like binary search, especially when the size of the dataset becomes large.

  - *Not Suitable for Sorted Data* - Linear search does not use the fact that the dataset is sorted, unlike binary search, which is much faster on sorted data.

  - *Searches Are Inefficient if Done Frequently* - As a result of the linearity in its time complexity, linear search can turn into a bottleneck in case many search operations are needed.

  - *High Time Complexity* - In each of the search operations, it probably goes through all the elements; this could be time-consuming, especially where there are large numbers of elements to be dealt with in the dataset.

  - *Poor Scalability* - Linear search has very poor performance for large data sizes; it is not appropriate for applications requiring fast queries on big datasets.

02. Binary Search

Binary search is a searching technique that works efficiently on ordered lists. Therefore, we need to ensure the list is sorted to perform the binary search for an element in some list.

Binary search is a divide-and-conquer approach where the list is divided into two halves and the required element compared with its middle element. In case of a match, it returns the location of the middle element; otherwise, it searches into either half depending upon the result produced through the match.

Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

There are two methods to implement the binary search algorithm

- ✔ Iterative method
- ✔ Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the element to be searched in **num = 74**

0	1	2	3	4	5	6	7	8
11	**85**	**36**	**74**	**5**	**88**	**51**	**13**	**3**

This is not arranged in a sorted manner. First we should sorted elements.

0	1	2	3	4	5	6	7	8
3	**5**	**11**	**13**	**36**	**49**	**51**	**74**	**85**

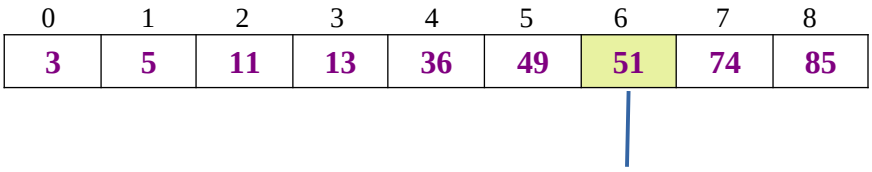
We have to use the below formula to calculate the mid of the array

mid = (start + end) / 2

start = 0
end = 8
mid = (0+8)/2 = 4

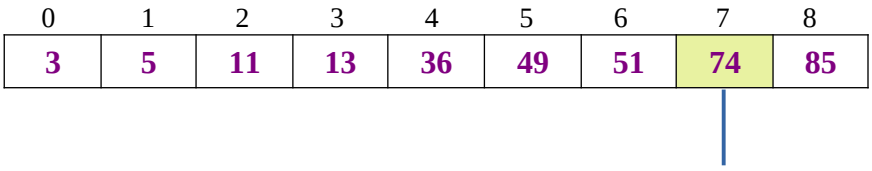
0	1	2	3	4	5	6	7	8
3	**5**	**11**	**13**	**36**	**49**	**51**	**74**	**85**

array[mid] = 36
array[mid] < num => **start = mid +1**
end = 8
mid = ((4 + 1) + 8) / 2 = 13/2 = 6



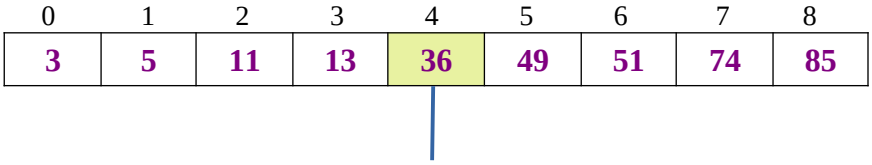
array[mid] = 51
array[mid] < num =>

start = mid +1
end = 8
mid = ((6 + 1) + 8) / 2
= 15/2 = 7



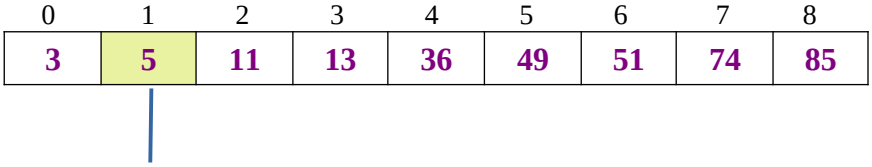
array[mid] = 74
array[mid] == num

Let the element to be searched in **num = 5**



array[mid] = 36
array[mid] > num =>

start = 0
end = mid - 1
mid = ((0 + (4-1)) / 2 = 3/2 = 1



array[mid] = 5
array[mid] == num

```java
import java.util.*;

class Main{
    static Scanner input = new Scanner(System.in);

    public static int binarySearchMethod(int[] array){
        System.out.print(s:"Enter a number (3 , 5 , 11 , 13 , 36 , 49 , 51 , 74 , 85) : ");
        int num =  input.nextInt();

        int start = 0;
        int end = array.length -1;

        while(start <= end){
            int mid = start + (end - start) / 2;

            if(array[mid] == num){
                return mid;
            }
            if(array[mid] < num){
                start = mid +1;
            }
            else{
                end = mid - 1;
            }
        }
        return -1;
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {3 , 5 , 11 , 13 , 36 , 49 , 51 , 74 , 85};

        int index =  binarySearchMethod(array);

        if(index != -1){
            System.out.println("The target number found  at index " + index);
        }
        else{
            System.out.println(x:"The number is not found!");
        }
    }
}
```

OUTPUT : The target number found at index 7
OUTPUT : The target number found at index 1

Let's see the time complexity of binary search in the best case, average case, and worst case. We will also see the space complexity of binary search.

- **Best Case Complexity** - In Binary search, best case occurs when the element to search is found in first comparison, when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1)**.

- **Average Case Complexity** - The average case time complexity of Binary search is **O(logn)**.

- **Worst Case Complexity** - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn)**.

The **space complexity** of binary search is **O(1)**.

Use Case - Efficiently finding an item in a sorted list or array.

➔ **EX:-**
Searching for a name in a sorted list of names or finding a specific value in a sorted database index.

➢ **Advantages**

- *Efficiency* - Time Complexity: Binary search runs in O(log n), making it much faster compared to linear search's O(n) for large data sets. It manages this by reducing the size of the search space by half with each step.

- *Performance on Large Datasets* - Binary Search, due to its logarithmic time complexity, is viable for very large data sets on which a linear search would be impractically slow.

- *Predictable Performance* - One of the important properties of binary search is that it is predictable its performance does not vary with the size of data beyond its logarithmic growth. Predictability is a desirable property in a number of performance-sensitive applications.

- *Fewer Comparisons* - Binary search has fewer comparisons compared to linear search; thus, it brings about performance benefits, mostly where comparisons are dear operations.

- *Immutable Data* - Binary Search is very good at searching datasets that do not change frequently, since the overhead of maintaining sorted data is justified by its fast search times.

- **Disadvantages :**

  - *Requires Sorted Data* - The biggest disadvantage with binary search is that it requires the data to be sorted previously. The process of sorting can be fairly computationally expensive O(nlogn)), which is counterproductive when applying binary search for frequently changing datasets.

  - *Implementation Complexity* - The implementation of binary search in its recursive form, in particular, is complex due to careful handling of index calculations and boundary conditions when compared to linear search.

  - *Inefficient over Small Dataset* - For small datasets, binary search does not bring much benefit in performance over linear search because added overhead from sorting and extra logic associated with binary search may not be justifiable.

  - *Not Suitable for Linked Lists* - Binary search is also inappropriate for data structures where random access is forbidden, like linked lists, because elements in these structures cannot be efficiently accessed by their index.

  - *Overhead of Maintaining Sorted Data* - In the case of frequently updated dynamic data, maintaining the list in a sorted fashion may be too expensive to gain the advantages of binary search on search.

  - *Non-Trivial Index Calculations* - Care should be taken that no overflow occurs during the midpoint calculation. This is especially true for languages that do not handle integer overflows gracefully.

03. Bubble Sort

The mechanism of bubble sort works on the basis of repeated swapping of the adjacent elements until they are not in the intended order. It is so named because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface similarly, the array elements in bubble sort move to the end in each iteration.

Though it is easy to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is O(n2), where n is a number of items.

Bubble short is majorly used where

- complexity does not matter.
- Simple and short code is preferred.

To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the **complexity** of bubble sort is **O(n2)**.

Let the elements of array are ,

0	1	2	3	4
49	**23**	**5**	**27**	**1**

First Pass :

01) Sorting will begin from the first two elements. Let us compare to check which one is bigger.

0	1	2	3	4
49	**23**	**5**	**27**	**1**

Here, 49 is greater than 23 (49 > 23), So, swapping is required. After swapping new array will look like ,

0	1	2	3	4
23	**49**	**5**	**27**	**1**

02) Now, compare 49 and 5.

0	1	2	3	4
23	**49**	**5**	**27**	**1**

Here, 49 is greater than 5 (49 > 5), So, swapping is required. After swapping new array will look like ,

0	1	2	3	4
23	**5**	**49**	**27**	**1**

03) Now, compare 49 and 27.

0	1	2	3	4
23	**5**	**49**	**27**	**1**

Here, 49 is greater than 27 (49 > 27), So, swapping is required. After swapping new array will look like ,

0	1	2	3	4
23	**5**	**27**	**49**	**1**

04) Now, compare 49 and 1.

0	1	2	3	4
23	**5**	**27**	**49**	**1**

Here, 49 is greater than 1 (49 > 1), So, swapping is required. After swapping new array will look like ,

0	1	2	3	4
23	**5**	**27**	**1**	**49**

Now, move to the second iteration.

Second Pass :

The same process will be followed for second iteration.

0	1	2	3	4
23	**5**	**27**	**1**	**49**

01) compare 23 and 5

0	1	2	3	4
23	**5**	**27**	**1**	**49**

Here, 23 is greater than 5 (23 > 5), So, swapping is required. After swapping new array will look like ,

0	1	2	3	4
5	**23**	**27**	**1**	**49**

02) Now, compare 23 and 27. No swapping required .

0	1	2	3	4
5	**23**	**27**	**1**	**49**

03) Now, compare 27 and 1.

0	1	2	3	4
5	**23**	**27**	**1**	**49**

Here, 27 is greater than 1 (27 > 1), So, swapping is required. After swapping new array will look like ,

0	1	2	3	4
5	**23**	**1**	**27**	**49**

04) Now, compare 27 and 49. No swapping required.

Third Pass :

0	1	2	3	4
5	**23**	**1**	**27**	**49**

0	1	2	3	4
5	**23**	**1**	**27**	**49**

Swapping required ,

0	1	2	3	4
5	**1**	**23**	**27**	**49**

Fourth Pass :

0	1	2	3	4
5	**1**	**23**	**27**	**49**

Swapping required ,

0	1	2	3	4
1	**5**	**23**	**27**	**49**

Hence, there is no swapping required, so the array is completely sorted.

```java
class Main{
    static Scanner input = new Scanner(System.in);

    public static void bubbleSortMethod(int[] array){
        int n = array.length;
        int temp = 0;

        for (int i = 0; i < array.length; i++) {
            for (int j = 1; j < (n-i); j++) {
                if(array[j-1] > array[j]){
                    temp = array[j-1];
                    array[j-1] = array[j];
                    array[j] = temp;

                }

            }
        }
        System.out.println(Arrays.toString(array));
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {49 , 23 , 5 , 27 , 1};

        bubbleSortMethod(array);


    }
}
```

OUTPUT : [1, 5, 23, 27, 49]

Let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n)**.

- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **O(n2)**.

- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **O(n2)**.

The **space complexity** of bubble sort is **O(1)**. It is because, in bubble sort, an extra variable is required for swapping.
The **space complexity of optimized bubble sort** is **O(2)**. It is because two extra variables are required in optimized bubble sort..

Use Case - Simple sorting tasks, mainly for educational purposes or small datasets.

➔ **EX:-**
Sorting a small list of student names or grades where simplicity is preferred over efficiency.

➢ **Advantages :**

- *Simplicity* - It is easy to understand, thus making it a very good algorithm for educational purposes and for beginners in algorithms. The logic behind the algorithm includes only simple comparisons and swaps, which are pretty easy to implement and trace.

- *In-place Sorting* - Bubble sort is an in-place sorting algorithm that means it requires only a small constant amount of additional memory space for its execution. There is no need for additional space for any extra data structures like arrays or lists to hold the intermediate data.

- *Stability* - Another important property of Bubble Sort is that it is a stable sorting algorithm; that is, it produces the same result as any other stable sorting algorithm. This means relative order between equal elements is maintained in the sorted array. This point often becomes quite significant when the key to be used for sorting is a subset of the data.

- *Early Termination* - This algorithm can be optimized by adding a condition to stop early if no swaps are made in a pass through the array, which means it is already sorted. This optimization will enhance the performance for nearly sorted arrays.

- **Disadvantages :**

  - *Inefficiency* - Among the algorithms for sorting huge data sets, the Bubble sort is highly inefficient. This will be with an average and worst-case time complexity of O(n2)O(n2). Therefore, it is unfit to sort a huge list or array.

  - *Performance* - Even in the best case, when the array is already sorted, the amount of comparisons that Bubble Sort does is (n−1)(n−1). Hence, in any case, for the best-case time complexity, it is O(n)O(n). Under these circumstances, a more efficient algorithm will be able to do better.

  - *Non-Essential Operations* - The algorithm wastes comparisons and swaps, even when the elements are in order, thus inserting additional operations that lower performance.

  - *Low Practical Value* - Due to its inefficiency, bubble sort is rarely used in real applications, except perhaps in education or in very small dataset sorting.

04. Selection Sort

In selection sort, on every pass, the smallest value is selected from the unsorted elements of the array and inserted into its proper position in the array. It is also one of the easiest algorithms. It is an internal comparison sorting algorithm. This algorithm divides the array into two parts: one is a sorted part, and another is the unsorted part. First of all, the sorted part is empty and the unsorted part is the array given. The part that is sorted is placed on the left, while the one that is not is on the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. Then, the second smallest element is selected and placed in the second position. The process continues till the array is completely sorted.

The average and worst-case complexity of selection sort is O(n2), where 'n' is the number of items. Because of this, it's not suitable for large data sets.

Selection sort is generally used when ,

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

Let the elements of array are ,

0	1	2	3	4	5	6	7	8
11	**85**	**36**	**74**	**5**	**88**	**51**	**13**	**3**

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

01) At present, 11 is stored at the first position, after searching the entire array, it is found that 3 is the smallest value.

0	1	2	3	4	5	6	7	8
11	**85**	**36**	**74**	**5**	**88**	**51**	**13**	**3**

So, swap 11 with 3. After the first iteration, 3 will appear at the first position in the sorted array.

0	1	2	3	4	5	6	7	8
3	**85**	**36**	**74**	**5**	**88**	**51**	**13**	**11**

02) For the second position, where 85 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 5 is the second lowest element in the array that should be appeared at second position.

0	1	2	3	4	5	6	7	8
3	**85**	**36**	**74**	**5**	**88**	**51**	**13**	**11**

Now, swap 85 with 5. After the second iteration, 5 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

0	1	2	3	4	5	6	7	8
3	**5**	**36**	**74**	**85**	**88**	**51**	**13**	**11**

03) For the third position, where 36 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 11 is the third lowest element in the array that should be appeared at third position.

0	1	2	3	4	5	6	7	8
3	**5**	**36**	**74**	**85**	**88**	**51**	**13**	**11**

Now, swap 26 with 11. After the third iteration, 11 will appear at the third position in the sorted array. So, after three iterations, the three smallest values are placed at the beginning in a sorted way.

0	1	2	3	4	5	6	7	8
3	**5**	**11**	**74**	**85**	**88**	**51**	**13**	**36**

04) For the fourth position, where 74 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 13 is the fourth lowest element in the array that should be appeared at fourth position.

0	1	2	3	4	5	6	7	8
3	**5**	**11**	**74**	**85**	**88**	**51**	**13**	**36**

Now, swap 74 with 13. After the fourth iteration, 13 will appear at the fourth position in the sorted array. So, after four iterations, the four smallest values are placed at the beginning in a sorted way.

0	1	2	3	4	5	6	7	8
3	**5**	**11**	**13**	**85**	**88**	**51**	**74**	**36**

05) For the fifth position, where 85 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 36 is the fifth lowest element in the array that should be appeared at fifth position.

0	1	2	3	4	5	6	7	8
3	**5**	**11**	**13**	**85**	**88**	**51**	**74**	**36**

Now, swap 85 with 36. After the fifth iteration, 36 will appear at the fifth position in the sorted array. So, after five iterations, the five smallest values are placed at the beginning in a sorted way.

0	1	2	3	4	5	6	7	8
3	**5**	**11**	**13**	**36**	**88**	**51**	**74**	**85**

06) For the sixth position, where 88 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 51 is the sixth lowest element in the array that should be appeared at sixth position.

0	1	2	3	4	5	6	7	8
3	5	11	13	36	88	51	74	85

Now, swap 88 with 51. After the sixth iteration, 51 will appear at the sixth position in the sorted array. So, after six iterations, the six smallest values are placed at the beginning in a sorted way.

0	1	2	3	4	5	6	7	8
3	5	11	13	36	51	88	74	85

07) For the seventh position, where 88 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 74 is the seventh lowest element in the array that should be appeared at seventh position.

0	1	2	3	4	5	6	7	8
3	5	11	13	36	51	88	74	85

Now, swap 88 with 74. After the seventh iteration, 74 will appear at the seventh position in the sorted array. So, after seven iterations, the seven smallest values are placed at the beginning in a sorted way.

0	1	2	3	4	5	6	7	8
3	5	11	13	36	51	74	88	85

08) For the eighth position, where 88 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 85 is the eighth lowest element in the array that should be appeared at eighth position.

0	1	2	3	4	5	6	7	8
3	5	11	13	36	51	74	88	85

Now, swap 88 with 85. After the eighth iteration, 85 will appear at the eighth position in the sorted array. So, after eight iterations, the eight smallest values are placed at the beginning in a sorted way.

0	1	2	3	4	5	6	7	8
3	5	11	13	36	51	74	85	88

Now, the array is completely sorted.

```java
class Main{

    public static void selectionSortMethod(int[] array){
        int n = array.length;

        for (int i = 0; i < n-1; i++) {
            int minIndex = i;
            for (int j = i+1; j < n; j++) {
                if(array[j] < array[minIndex]){
                    minIndex = j;
                }
            }
            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
        System.out.println(Arrays.toString(array));
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {11 , 85 , 36 , 74 , 5 , 88 , 51 , 13 , 3};

        selectionSortMethod(array);
    }
}
```

OUTPUT : [3, 5, 11, 13, 36, 51, 74, 85, 88]

Let's see the time complexity of selection sort in the best case, average case, and worst case. We will also see the space complexity of selection sort.

- **Best Case Complexity** - It occurs when there is no sorting required, the array is already sorted. The best-case time complexity of selection sort is **O(n2)**.

- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n2)**.

- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **O(n2)**.

The **space complexity** of selection sort is **O(1)**. It is because, in selection sort, an extra variable is required for swapping.

Use Case - Sorting tasks where simplicity is more important than efficiency, or where memory usage is a concern.

➔ **EX:-**
Sorting a small number of items, such as arranging playing cards in hand, or when memory space is limited and only simple sorting is required.

➢ **Advantages :**

- *Simplicity* - Selection sort algorithm is straightforward, and it is easy to learn and implement because it derives from a straightforward idea: selecting elements in an increasing or decreasing manner.

- *In-place Sorting* - This sorting algorithm is done in-place, meaning that it uses only a constant amount of additional memory space, O(1)O(1), beyond the input array itself.

- *Good Performance on Small Lists* - For small arrays or lists, simplicity and low overhead can be the significant advantages of Selection Sort.

- *Predictable Pattern* - Number of swaps performed by this algorithm is always exactly n−1n−1 which may be useful if writing to memory is significantly more expensive than comparisons (although the same number of swaps can be achieved by better algorithms like Heap Sort).

- **Disadvantages :**

  - *Inefficiency* - The average and worst-case time complexity of Selection Sort is O(n2)O(n2), where nn denotes the number of elements in the array. This makes it inefficient for large datasets compared to more advanced sorting algorithms like Quick Sort, Merge Sort, or Heap Sort.

  - *Unstable* - The fact that Selection Sort is not a stable sorting algorithm means that it can change the relative order of equal elements, which might be considered as one of the negative sides when sorting data structures with complex fields.

  - *Bad Performance on Already Sorted Arrays* - Unlike other simple sorting algorithms, including Bubble Sort with its early termination optimization, Selection Sort does not benefit from partial ordering of the input array; it makes the same number of comparisons regardless of the initial order of the array.

  - *Redundant Comparisons* - This algorithm makes a large number of redundant comparisons, since it keeps on comparing the elements with each other even when the minimum element is found at the early part in a pass through the array.

05. Insertion Sort

Insertion sort works just like the Sorting of playing cards in hands. We assume the first card is already sorted in the card game, then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put at their exact place.

The same concept is applied in insertion sort. Now, the concept of insertion sort is this: Take one element, process it in the sorted array. Though it is easy to use, it's not appropriate for large datasets as the average and worst-case complexity of insertion sort is O(n2), where n is the number of items. Insertion sort is less effective as compared to other sorting algorithms such as heap sort, quick sort, merge sort, etc.

There are a number of advantages of insertion sort, which include ,

- Easy implementation
- Efficient for small datasets.
- Adaptive, meaning that it is suitable for data sets already significantly sorted.

Let the elements of array are ,

0	1	2	3	4
18	**35**	**78**	**3**	**24**

01) Initially, the first two elements are compared in insertion sort.

0	1	2	3	4
18	**35**	**78**	**3**	**24**

Here, 35 is greater than 18. That means both elements are already in ascending order. So, for now, 18 is stored in a sorted sub-array.

0	1	2	3	4
18	**35**	**78**	**3**	**24**

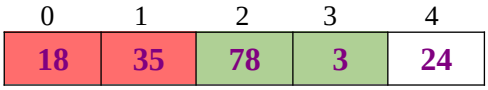
02) Now, move to the next two elements and compare them.

0	1	2	3	4
18	**35**	**78**	**3**	**24**

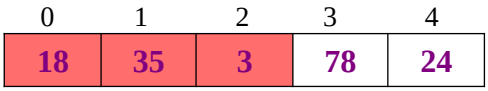
Here, 35 is smaller than 78. That means both elements are already in ascending order.

0	1	2	3	4
18	**35**	**78**	**3**	**24**

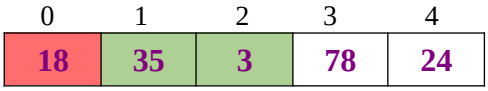
03) Now, two elements in the sorted array are 18 and 35. Move forward to the next elements that are 78 and 3.



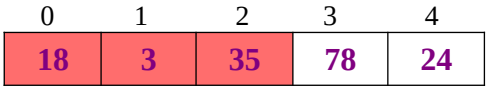
Here, 3 is smaller than 78. So, 3 is not at correct position. Now, swap 3 with 78.



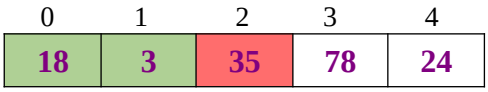
After swapping, elements 35 and 3 are unsorted.



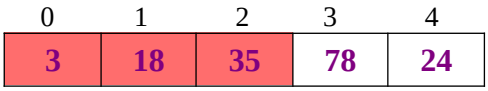
So, swap them.



Now , elements 18 and 3 are unsorted.

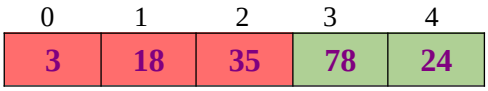


So , swap them.

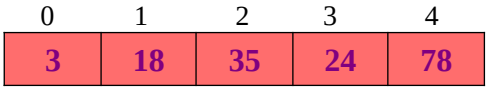


Now, the sorted array has three items that are 3, 18 and 35.

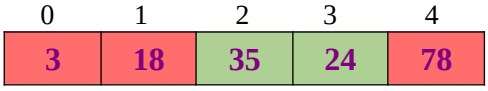
04) Move to the next items that are 78 and 24.



Here, 24 is smaller than 78. So, 24 is not at correct position. Now, swap 24 with 78.



After swapping, elements 35 and 24 are unsorted.



Now , swap them.

0	1	2	3	4
3	18	24	35	78

Now, the array is completely sorted.

```java
class Main{

    public static void insertionSortMethod(int[] array){
        int n = array.length;

        for (int i = 1; i < n; i++) {
            int currentElement = array[i];
            int j = i-1;

            while(j >= 0 && array[j] > currentElement){
             array[j+1] = array[j];
             j = j-1;
            }
            array[j+1] = currentElement;
        }

        System.out.println(Arrays.toString(array));


    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {18 , 35 , 78 , 3 , 24};

       insertionSortMethod(array);
    }
}
```

OUTPUT : [3, 18, 24, 35, 78]

Let's see the time complexity of insertion sort in the best case, average case, and worst case. We will also see the space complexity of insertion sort.

- **Best Case Complexity** - It occurs when there is no sorting required, the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.

- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **O(n2)**.

- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **O(n2)**.

The **space complexity** of insertion sort is **O(1)**. It is because, in insertion sort, an extra variable is required for swapping.

Use Case - Sorting small datasets or datasets that are already partially sorted.

➔ **EX:-**

Sorting a list of to-do items or small arrays in applications where frequent insertions and removals occur, like maintaining a sorted list of real-time data.

➢ **Advantages :**

- *Simplicity* - Insertion sort is simple and straightforward. Thus, it is an ideal introductory example to be used for illustrating the concept of sorting algorithms for teaching purposes.

- *In-place Sorting* - The algorithm performs the sort in-place; hence, it has only a constant amount of extra memorization space, O(1) apart from the input array itself.

- *Stability* - It is a stable sorting algorithm that maintains the relative order of records having equal keys. This can be important when records are sorted with multiple fields.

- *Adaptive* - Insertion sort is adaptive; hence, it performs well for nearly sorted arrays. Its best time complexity is O(n)O(n) when the array is already sorted, making it faster than O(n2)O(n2) algorithms under such conditions.

- *Efficient for Small Datasets* - This algorithm performs well on small datasets; hence, it's quite appropriate for small arrays/lists where the extra overhead of the more complex algorithms is not warranted.

- *Online Algorithm* - Insertion sort can sort its data as it receives it. This makes it useful for real-time applications where data is continuously coming in.

- **Disadvantages :**

  - *Inefficiency for Large Datasets* - The average and worst-case time complexity of Insertion sort is O(n2)O(n2), making the algorithm pretty inefficient for large datasets compared to more advanced algorithms like Quick Sort, Merge Sort, or Heap Sort.

  - *Poor Performance with Random Data* - This algorithm performs very poorly on randomly ordered data. It does so because, to insert the current element in its correct position, it has to shift those elements several times.

  - *High Number of Comparisons* - Insertion Sort requires a high number of comparisons. This is true in particular when the array is in reverse order. It can degrade performance for large datasets.

  - *Not Suitable for Linked Lists* - While this algorithm is efficient for arrays due to the contiguous memory layout, it is less efficient for linked lists since insertion in the middle of a linked list requires traversal of the list.

06.2-Sum Algorithm

Finding two numbers in the list whose summation is equal to the given target sum. There is a variety of how to do this problem, each of them putting a lower and higher time and space complexity measure.

Here are a few common methods that can apply to solve the below problem.

Problem Definition :

Given an array of integers nums and an integer target, return the indices of two numbers such that they add up to the target.

2-SUM Problem Approaches to solve it:

Brute Force Approach Algorithm

Use 2 nested loops to check every pair of numbers and see if they sum up to the target.

- Time Complexity Worst-case: **O(n2)**
- Space Complexity: **O(1)**

Use Case - Finding pairs of numbers in an array that sum up to a specific target.

➔ **EX:-**

In financial applications, finding two transactions that together match a specific amount, or in gaming, identifying two items that together cost a certain amount.

```java
import java.util.*;

class Main{
    static Scanner input = new Scanner(System.in);

    public static int[] twoSumAlgoMethod(int[] array){
        System.out.print(s:"Enter a number : ");
        int num = input.nextInt();

        int n = array.length;

        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                if(array[i] + array[j] == num){
                    return new int[]{i,j};
                }
            }
        }
        return null;
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {5 , 17 , 6 , 10 , 2 , 3};

        int[] result = twoSumAlgoMethod(array);

        if(result != null){
            System.out.println("Indicies : [" + result[0] + " , " + result[1] + "]");
        }
        else{
            System.out.println(x:"Not found");
        }
    }
}
```

- **Advantages :**
  - *Simplicity* - Easy to understand and implement, the algorithm has been the subject of much attention.
  - *No extra space* - It does not need any additional data structure other than the input array itself.
- **Disadvantages :**
  - *Inefficiency* - This has a time complexity of O(n2)O(n2) , which is pretty slow, especially for big arrays.
  - *Scalability* - It will not work on large data sets because of its quadratic time complexity.

07.Reverse array without temp array

An array in Java can be reversed without the use of a temporary array by using the two-pointer technique. Actually, the algorithm realizes swapping elements from two ends, moving towards the center.

- **Time Complexity** - **O(n)**, where n is the number of elements in an array. This is because every element gets swapped exactly once.
- **Space complexity** - **O(1)** , no extra space proportional to the array size is used.

Use Case - Reversing elements in an array or list with minimal extra space.

➔ **EX:-**

Reversing a list of items in a queue or stack where memory efficiency is crucial, like in low-memory embedded systems.

Let the elements of array are ,

0	1	2	3	4
5	17	6	10	23

01) First swap the first index and last index.

0	1	2	3	4
5	17	6	10	23

0	1	2	3	4
23	17	6	10	5

02) Now swap the second left index and second right index.

0	1	2	3	4
23	17	6	10	5

0	1	2	3	4
23	10	6	17	5

Now all elements are sorted.

```java
class Main{

    public static void reversed(int[] array){
        int left = 0;
        int right = array.length -1;

        while(left <= right){
            int temp = array[left];
            array[left] = array[right];
            array[right] = temp;

            left++;
            right--;

        }
        System.out.println(Arrays.toString(array));
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {5 , 17 , 6 , 10 , 2 , 3};

        reversed(array);


    }
}
```

08. Find Duplicate Numbers in an array

If it is necessary to find the presence of duplicate numbers in the array, it is possible to first sort the array and then find the duplicate numbers by checking adjacent elements.
In this approach, the sorting technique is used to bring duplicate elements next to each other in order to permit simple scanning to find duplicates.

This algorithm has a **time complexity** of **O(n log n)** due to the sorting step, with an additional O(n) for the scanning step.
Space Complexity **O(1)** extra space if in-place sorting is done.

Use Case - Identifying duplicate entries in datasets.

➔ **EX:-**
Detecting duplicate records in a database, finding repeated values in user input, or ensuring unique entries in a user registration system.

```java
import java.util.Arrays;

class Main{

    public static void findDuplicates(int[] array){
        Arrays.sort(array);
       boolean hasDuplicates = false;

        for (int i = 1; i < array.length; i++) {
            if(array[i] == array[i-1]){
                System.out.println("Duplicate found : " + array[i]);
                hasDuplicates = true;
            }
        }
        if(!hasDuplicates){
            System.out.println(x:"No duplicates found!!!");
        }
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {5 , 17 , 5 , 10 , 10 , 18 , 13 , 3 , 9 , 3 };

        findDuplicates(array);


    }
}
```

09.Find Index

Finding the index of a specific element in an array is a common task. You can accomplish this in Java using a simple loop to iterate through the array and compare each element with the target value.

- **Time Complexity** - **O(n)**, where n is the number of elements in the array, as you may need to check each element in the worst case.
- **Space Complexity** - **O(1)**, as no additional space is required other than a few variables.

Use Case - Locating the position of a specific element within an array or list.

➔ **EX:-**

Retrieving the index of a product in an inventory system or finding the position of a keyword in a text document.

```java
import java.util.*;;

class Main{
    static Scanner input = new Scanner(System.in);

    public static void findIndex(int[] array){
        System.out.print(s:"Enter a number : ");
        int num = input.nextInt();

        boolean isFound = false;

        for (int i = 0; i < array.length; i++) {
            if(array[i] == num){
                isFound = true;
                System.out.println("Element " + num + " found at index : " + i);
            }

        }
        if(!isFound){
            System.out.println(x:"Not found!!!");
        }
    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {5 , 17 , 23 , 57 , 48 };

        findIndex(array);


    }
}
```

10. Add Insertions Algorithm

The Add Insertions Algorithm is used to insert elements into a sorted array while maintaining the sorted order. This is useful in scenarios where you need to continuously add elements to a sorted list.

Time Complexity:

- **Worst-case** - **$O(n^2)$**, where n is the number of elements, occurs when the array is sorted in reverse order.
- **Best-case** - **O(n),** occurs when the array is already sorted.
- **Average-case** - **$O(n^2)$**, as it generally involves comparing each element with all other elements before it.

Space Complexity - **O(1)**, as it sorts the array in-place without needing extra storage.

Use Case - Inserting new elements into a data structure efficiently.

➔ **EX:-**

Inserting new data into a sorted list while maintaining order, like adding new entries into a priority queue or updating a real-time sorted leaderboard.

```
class Main{

    public static void insertionSortMethod(int[] array){
        int n = array.length;

        for (int i = 1; i < n; i++) {
           int currentElement = array[i];
           int j = i-1;

           while(j >= 0 && array[j] > currentElement){
            array[j+1] = array[j];
            j = j-1;
           }
           array[j+1] = currentElement;
        }

        System.out.println(Arrays.toString(array));


    }
    Run | Debug
    public static void main(String[] args) {
        int[] array = {18 , 35 , 78 , 3 , 24};

       insertionSortMethod(array);
    }
}
```