**Searching and Sorting**

1. **Linear Search**

**Brute-Force Searching**  
Brute-force searching is also known as exhaustive searching, and it simply means to check all possible configurations for a given problem. It is easy to implement and would most definitely find the solution, although it consumes a lot of time.

Example:

If you have a problem set in acountable space (chess moves are countable, passwords are countable, continuous stuff is uncountable), then the brute force approach will explore this space considering all solutions equally. In the chess example, if you want to checkmate your opponent, then this is done through a series of moves, which are countable. Brute force will go through all the sequences of moves, however unlikely they may be. The word unlikely is important because it means that if you have knowledge of your problem (you know what is unlikely to be a solution, for example, sacrificing your queen), then you can do much better than applying brute force.

For more details, please visit the link given below:

<https://stackoverflow.com/questions/8103050/what-exactly-is-the-brute-force-algorithm>

#### Q1. Brute-Force Searching

Choose the correct statement from the options given below.

Ans: Linear search can be said to be a special case of the brute-force method.

**✓ Correct**

**Feedback:**

The brute-force method is a broader category, in which each option available is checked for the best possible answer. Although linear search also compares the key element with every element in the series, it would stop as soon as the element being searched is found and would not further check whether the element is present in the series again later.

# Java Implementation of Linear Search

#### Q1: Analysis of Linear Search

In an array of ‘n’ elements, the average number of key comparisons that are done for a successful linear search are:

(**Hint:** In an array of 'n' elements, you have to consider all the comparisons from the first to the nth element.)

**Ans:** (n+1)/2

**✓ Correct**

**Feedback:**

If the key that you are searching is found at the first position, then the number of comparison is only 1. If the key is found at the second position, then the number of of comparisons done are 2. Similarly, if the key is found at the 'nth' position, then the number of comparisons done is 'n'. Therefore, if we add the comparisons for each case, then we would draw the following conclusion:

          1+2+3+4+...+(n-1)+n  = n(n+1)/2

We have a total of 'n' elements and, therefore, the average case would be:

**n(n+1)/2**n  =

                                                        n+12

# Binary Search

# **So, for binary search, you need a sorted array**. You start at the middle index and compare the element that you are searching for with the element at the middle index. If the element at this index does not match the element that you are searching for, then you check whether it is greater or lesser than the element at the index. If it is greater than the element at the middle index, then you discard everything to the left and move to the right. Then you find the middle of this array, which is to the right, compare the required element with its middle and continue doing this until you find the required element.

#### Q1: Linear Search vs Binary Search

Suppose you have an unsorted array, and you are told to search for a particular element in it.

Choose whether the following statement is True or False:

Since you know that binary search is more efficient than linear search, and since binary search can be applied only on a sorted array, the most efficient way to approach this task is to first sort the array and then apply binary search.

Ans: False

**✓ Correct**

**Feedback:**

In such a case, we should calculate the efficiency of the entire process. Sorting an unsorted array is an O(nlogn) process, as you will learn in the upcoming sessions. Once sorted, applying binary search on the array is an O(logn) process. So, the entire process would take steps in the order of nlogn + logn, which is more than O(n) as taken by linear search. So, if we have an unsorted array and there is no need to sort it, then using linear search is more efficient than using binary search.

#### Q2: Binary Search

In the binary search algorithm, after every search operation, the search gets reduced by \_\_\_\_\_\_?  
 Ans: 1/2   

**✓ Correct**

**Feedback:**

In the binary search algorithm, after each search operation, the search gets reduced by 1/2. For example, if we are given the sequence {0,1,3,4,5,7,8,9 }, and we are searching for 1, then according to binary search, we would first find the middle element, which would be at 0+7/2 =3.5~3, the third position, i.e., '4', in this case.  Next, we would compare our key '1' to '4' . Since they are not equal, we would now check whether our key is greater than or less than 4. We conclude that it is less than 4, and now the search zone is narrowed down to '0' to 3-1 = '2' (3 was the position of our current middlemost element)

Next, we have to search for '1' with elements at positions {0,1,2}. Before this step, the total number of elements in the sequence that we had was '7'. Searching sequentially, we should have compared it to all the 7 elements, but with binary search, we are comparing it with a maximum of 3 elements only, that is, almost half of the total number of elements. Hence, after every step, the search zone is narrowed down to ½.

1. **Code for Binary Search**:

#### **Q1**: What Can be Done?

Based on what you have learnt so far, can you think of an approach that will make this search operation as fast as possible?

Ans: Suppose you have 1,000 entries in a database. If you use linear search to search for an email ID, then in the worst case, it would take you 1,000 steps. Instead, if you use binary search, then it would take just 10 steps in the worst case (log2 1,000). So, you can use binary search to search for the email ID and then fetch the score corresponding to it. However, for binary search, you need a sorted array. Email IDs that are strings can be sorted in an alphabetical order. In the remainder of the video, we will see how it can be done.

#### Q2: Size of Array

Suppose you have a large array (of the order of a million entries) of unknown size. The array is sorted in increasing order. You are told to find an element k using binary search on the array. You cannot use the function array.length to find the number of elements. However, the system will show you the message 'Index Out of Bounds' if you try to read an element in the array using an index that is larger than the array length. So, how will you approach this problem? You can write your response in the box given below.

Ans: In this question, you are dealing with a sorted array. As discussed in the videos as well, when you encounter an element, say, j, that is greater than the element k, you can surely say index of k < index of j. The index of this element j then becomes your final index, and you can apply binary search on the array from index 0 to the index of j. So, what you need to do is find the first element that is greater than k and use its index up to which you can apply binary search. To achieve this,comparing k with each element would make it an O(N) process. You need to make this an O(logN) process. So, if you start with index 1, then you can compare on index 2, next on index 4, then on index 8, next on index 16, and so on. The moment you find an element greater than the element k, you stop and apply binary search till that index only. Since we are doubling our index every time, defining this range also becomes an O(logN) process.

#### Q3: Binary Search

Compute the average number of comparisons done for each element in a given sequence of 12 sorted consecutive terms {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} using the binary search method.

[**Hint:** Count the number of comparisons after which a particular element will be found. For example, to find ‘3’ in the series, the number of comparison required would be ‘3’. Consider three cases: Equality check with middle element, comparison check (greater/smaller) with the middle element and equality check with the new middle element (now matched)].

Ans: The given sequence is {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}.

The average number of comparison required for searching each term in the sequence would be different for terms at different positions.

For 6, i.e., the middle element, the number of comparisons required is only 1.

For 3 and 9, the number of comparisons required are 3 for each, because first 3 will be checked if it equal to 6, then it will be compared if it is greater than or less than 6 and accordingly will be moved to left and right and the last comparison for checking its equality with 3 , which is true. Similar computation goes for 9.

For 1, 4, 7 and 11, the number of comparisons is 5 each.

For 2, 5, 8, 10 and 12, the number of comparisons is 7 each.

Therefore,

Average comparisons =  1+2∗3+4∗5+5∗712  =  6212= 5.16 = 5.

**Sorting**

#### Q1: Best-Case Time Complexity

What is the best-case time complexity of Bubble sort?

[**Hint:** With a Bubble sort algorithm, you will be able to see that the largest element (assuming we are sorting from smallest to largest) will 'bubble' up to the top of the list or the array. Now, think what would the best case be for a bubble sort that works by swapping the adjacent elements repeatedly if they are in the incorrect order.]

Ans: O(n)

**✓ Correct**

**Feedback:**

The best case occurs when the elements are already sorted. The algorithm would use only one iteration, which would include (n-1) comparisons only and no swap. Hence, the best case time complexity would be O(n).

#### Q2: Time Complexity of Bubble Sort

Suppose we have an array of 10 elements sorted in increasing order. We want to sort this array in decreasing order using Bubble sort. Let X represent the Big O time complexity to sort this array.

Now, let's add two more elements to the array (assume the array is still in increasing order), such that the new array with the added elements is still in increasing order.

So, now we have an array of 12 elements in increasing order, which is required to be sorted in decreasing order. Let Y represent the Big O time complexity to sort this array in decreasing order.

Then Y - X should approximately be equal to:

Ans: 44

Explaination: -Bubble sort follows O(N\*N) time complexity in the worst case. This means that if N elements are added to the array, then the number of steps required to sort the array increases by N\*N. Since the array is in increasing order and it needs to be sorted in decreasing order, this becomes the worst-case scenario. Since we are adding two elements to this array, the number of steps would increase by approximately 144 -100, i.e., 44 steps.

# Array Sorting Using Bubble Sort

Now, We understand how the bubble sort works. Now, let's see its practical implementation by sorting the given array.

Let's assume you are having an array of 5 elements

[**18**,**3**,**17**,**19**,**1**]

Code Snippet of Bubble Sort

**for** (**int** i = **0**; i < numbers.length; i++) {

**for** (**int** j = **1**; j < (numbers.length - i); j++) {

**if** (numbers[j - **1**] > numbers[j]) {

//swap elements

swap(j - **1**, j, numbers);

}

}

}

Now, let's see what happens to our array when it went through the Bubble Sort Algorithm

After first pass :

**3** **18** **17** **19** **1**

**3** **17** **18** **19** **1**

**3** **17** **18** **19** **1**

**3** **17** **18** **1** **19**

After second pass:

**3** **17** **18** **1** **19**

**3** **17** **18** **1** **19**

**3** **17** **1** **18** **19**

After third pass:

**3** **17** **1** **18** **19**

**3** **1** **17** **18** **19**

After fourth pass:

**1** **3** **17** **18** **19**

So, you can see our unsorted array [18,3,17,19,1] becomes sorted array [1,3,17,18,19] after going through the Bubble Sort Algorithm.

#### Q3: Analysis of Selection Sort

According to what you have learnt so far, choose the correct answer to the question given below:

Why is Selection sort more efficient than Bubble sort?

Ans: It has fewer swaps than Bubble sort.

**✓ Correct**

**Feedback:**

Remember that in Bubble sort, the number of swaps has a time complexity of O(N2 ), whereas in Selection sort, it is O(N). This is because in Bubble sort, two consecutive elements are compared, and every time they are not in sequence, you swap them and continue doing so until the end of the iteration. On the other hand, in Selection sort, you need to find the ith largest element, and in just one swap, put that element at the (n-i)th position using a ‘swap’ function. Even though the number of comparisons they perform are the same, the difference lies in the number of swaps.

For more information, please visit the following links:  
<https://cs.stackexchange.com/questions/13106/why-is-selection-sort-faster-than-bubble-sort>  
<https://stackoverflow.com/questions/4561587/how-does-bubble-sort-compare-to-selection-sort>

#### Q4: Selection Sort vs Bubble Sort

Select the incorrect statement from the options given below:

Ans: If there are N items, then Bubble sort performs exactly N\*N comparisons.

**✓ Correct**

**Feedback:**

Since you have learnt the Bubble sort algorithm, after the first iteration, the largest element is shifted to the end of the array and so, there is no need to compare the numbers with the last element as it already in its correct position. Therefore, the number of comparisons would be N-1. Similarly, for the second iteration, the number of comparisons would be N-2, and so on. In this way, there are maximum N(N-1)/2 comparisons, which is not exactly equal to N\*N but they are of the order of N\*N.

# Array Sorting Using Selection Sort

Now, We understand how the Selection Sort works. Now, let's see its practical implementation by sorting the given array.

Let's assume you are having an array of 5 elements

[**54**,**15**,**25**,-**40**,**4**]

Code Snippet of Selection Sort

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

{

**int** min = i;

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

**if** (arr[j] < arr[min])

min = j;

**int** temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

Now, let's see what happens to our array when it went through the Selection Sort Algorithm

After first pass:

[-**40** **15** **25** **54** **4**]

After second pass:

[-**40** **4** **25** **54** **15**]

After third pass:

[-**40** **4** **15** **54** **25**]

After fourth pass:

[-**40** **4** **15** **25** **54**]

So, you can see our unsorted array [54,15,25,-40,4]  becomes sorted array [-40,4,15,25,54] after going through the Selection Sort Algorithm.

# Selection Sort - Real Life Usage

In the last few segments, you understood the selection sort algorithm, learnt to implement it, and finally went through its analysis. In this segment, you will understand its major uses in real life.

Selection sort, like the bubble sort algorithm, is useful to expose students to different sorting algorithms. However, the selection sort algorithm is preferred in a situation where you would want to minimise the swaps. A standard selection sort algorithm assures that you make exactly 'n - 1' swaps per sort.

Finally, here is a Gypsy folk dance demonstrating the selection sort [algorithm](https://www.youtube.com/watch?v=Ns4TPTC8whw):

#### Q5: In-Place Sorting Algorithms

An 'In-place' sorting algorithm is the one that does not use any extra array or memory (auxiliary memory) for sorting. It sorts the sequence within the given memory itself.

From the options given below, select the sorting algorithm that can be considered an 'In-place' algorithm.

Ans: All of the above

**✓ Correct**

**Feedback:**

All of the algorithms given are 'In-place' algorithms, since none of them use extra array or memory and sorts within the given data structure. Recall that we did not create any new array or any other data structure to sort the elements using Bubble sort, Selection sort or Insertion sort. We were performing the swaps or comparisons within the original array itself. Hence, all of them are in-place algorithms.

# Array Sorting Using Insertion Sort

Now, We understand how the Insertion Sort works. Now, let's see its practical implementation by sorting the given array.

Let's assume you are having an array of 5 elements

[**87**,-**74**,**0**,**5**,**55**]

Code Snippet of Insertion Sort

**for**( **int** i =**1** ;i<arr.length;i++){

**int** v = arr[i];

**int** j = i;

**while**(j>=**1** && arr[j-**1**]>v){

arr[j]=arr[j-**1**];

j--;

}

arr[j]=v;

}

Now, let's see what happens to our array when it went through the Insertion Sort Algorithm

After first pass :

[-**74** **87** **0** **5** **55**]

After second pass :

[-**74** **0** **87** **5** **55** ]

After third pass :

[-**74** **0** **5** **87** **55**]

After fourth pass :

[-**74** **0** **5** **55** **87**]

So, you can see our unsorted array [87,-74,0,5,55] becomes sorted array [-74,0,5,55,87] after going through the Insertion Sort Algorithm.

#### Q6: Insertion Sort

What is the best-case run-time for Insertion sort?

Ans: O(n)​

**✓ Correct**

**Feedback:**

The best case occurs when the data is already sorted. It would take only one iteration, where it would perform (n-1) comparisons and no swap. This leads to a time complexity of O(n).

#### Q7: Bubble Sort

What would be the output after **three iterations** of Bubble sort on the input given below? Consider the first element as the starting point.

The input elements are:  
12, 8, 9, 10, 15, 4, 3, 6, 7

Ans: 8 9 4 3 6 7 10 12 15

**✓ Correct**

**Feedback:**

In the first iteration, 12 will be swapped with 8, 9 and 10, one after another. Upon encountering 15, which is greater than 12, it would stop. So, 12 will not be swapped with 15. After that, the process of swapping would repeat, with 15 being swapped with 4, 3, 6 and 7. Therefore, after the first call, the array will be: 8,9,10,12,4,3,6,7,15. In the second iteration, the elements 8, 9, 10 and 12 would not be swapped. However, 12 will be swapped with 4, 3, 6 and 7, one after another. So, after the second iteration, the array would be: 8,9,10,4,3,6,7,12,15. Similarly, in the third iteration, 10 would be swapped with 4, 3, 6, and 7, one element after another. The array after the third iteration would be: 8,9,4,3,6,7,10,12,15.

#### Q8: Insertion Sort

 The number of steps required to sort the following array in ascending order using insertion sort would be:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Ans:   
Closer to N

**✓ Correct**

**Feedback:**

If you closely observe the array, then you will notice that most of the elements in the array are sorted. In a sorted array, Insertion sort has O(N) time complexity. Although the array is not completely sorted, the number of steps required would still be closer to N than to N​22​. So, it would be closer to 10 (i.e., <50).

#### Q9: Comparison of Algorithms

Suppose you have an array whose elements are arranged in descending order, and your aim is to sort this array in ascending order.

Which of the following sorting algorithms would help you perform this job in the best possible manner?

(**Note:** Assume that you are allowed to use only one of the algorithms, i.e., Insertion, Selection and Bubble sort, and consider their worst- and best-case time complexities to answer the question.)

Ans: All three algorithms will perform equally and will run at the worst-case time complexity.

**✓ Correct**

**Feedback:**

All three algorithms will reach their worst case if the array is sorted in descending order. This is because the program would compare all the elements and perform the maximum number of swaps.

Merge Sort:

MergeSort(A)

n=length(A)

if (n<2) return

mid=n/2

L=A[0...mid-1]

R=A[mid....n-1]

for 0<= l <=mid-1

L[l]=A[l]

for mid<= r <=n-1

R[r-mid]=A[r]

MergeSort(L)

MergeSort(R)

Merge(L,R,A)