**Sorting Algorithms**

Sorting is the process of arranging the elements of collection of data so that they can be placed either in ascending or descending order.

In java sorting is applicable to two classes: -

1. Array class (there are two type of array in array class.)

Array of primitives (int, char, …)

Array of objects (String, integer, Student, ….)

1. Collection class (collection are there only for non-primitive)

Which collection allow sorting?

* The collection which are list interface implementing classes like (arraylist, linkedlist, vector)
* Arrays.sort in java

The arrays.sort is used to sort the normal array.

Import java.util.Array;

* Collection.sort in java

The collection.sort is used to the collection.

**Stability in sorting algorithm**

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set.

***Which sorting algorithms are unstable?***

Quick Sort, Heap Sort etc., can be made stable by also taking the position of the elements into consideration. This change may be done in a way that does not compromise a lot on the performance and takes some extra space, possibly theta(n).

Example of stable sorts: -

Bubble sort, insertion sort, merge sort.

Example of unstable sort: -

Selection, Quick sort, Heap sort

***Can we make any sorting algorithm stable?***

Any given sorting algorithm which is not stable can be modified to be stable. There can be algorithm-specific ways to make it stable, but in general, any comparison-based sorting algorithm which is not stable by nature can be modified to be stable by changing the key comparison operation so that the comparison of two keys considers position as a factor for objects with equal keys.

* For sorting the primitive Dual pivot quicksort is used.
* For sorting non-primitive or for the collection(String, Integer etc), based on mergesort adaptation of timSort.

1. **Bubble sort**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

***Working of bubble sort***

In bubble sort repeatedly swap the adjacent element. It has multiple passes. In its first pass we move the largest element to the last index, second to the second last index and so. We swap the element until the array will not be sorted.

Let’s understand with the help of an example.

Let the elements of array are -

Bubble sort Algorithm

This array is not sorted.

First Pass : -

The sorting will start from the initial two elements. We will compare the element of 0th index with element of 1th index.

Here 13 is compared with 32.

13 < 32

32 is greater than 13, so it is already sorted.

So, we will compare 32 with 26.

32 > 26, so the swapping is required. After the swapping array will be

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | 26 | 32 | 35 | 10 |

Now, next elements 32 and 35 will compare. Here, it is already sorted, so we will move to the next element.

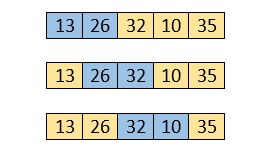
Now we will compare 35 and 10. Here 10 is less then 35, so it will swap 10 and 35.

At last the array will be.

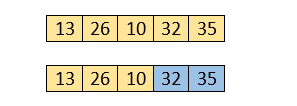
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | 26 | 32 | 10 | 35 |

Second pass: -

The same process will be followed in the second iteration.



Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be



*Third pass: -*

The same process is followed for the third pass also. It compares the each element.

After the third pass, array will be.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 13 | 10 | 26 | 32 | 35 |

*Fourth pass: -*

Similarly, after the fourth iteration, the array will be -

Bubble sort Algorithm

Approach: -

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

        {

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

            {

                if(arr[j] > arr[j+1])

                {

                    int temp = arr[j];

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

                    arr[j+1] = temp;

                }

            }

        }

**Bubble sort complexity**

1. ***Time complexity***

Now, 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.

|  |  |
| --- | --- |
| **Case** | **Time complexity** |
| Best case | O(n) |
| Average case | O(n^2) |
| Worst case | O(n^2) |

**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 must 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).

1. ***Space complexity***

* 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.

|  |  |
| --- | --- |
| Space complexity | O(1) |
| Stable | Yes |

**Optimized Bubble sort Algorithm**

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable swapped. It is set to true if swapping requires; otherwise, it is set to false.

It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable swapped will be false. It means that the elements are already sorted, and no further iterations are required.

This method will reduce the execution time and also optimizes the bubble sort.

Algorithm for optimized bubble sort

bubbleSort(array)

n = length(array)

repeat

swapped = false

for i = 1 to n - 1

if array[i - 1] > array[i], then

swap(array[i - 1], array[i])

swapped = true

end if

end for

n = n - 1

until not swapped

end bubbleSort.

1. **Selection sort**

Selection sort is a basic sorting algorithm. It is comparison-based algorithms and has theta N Square time in all cases. It does less memory write as compared to other algorithms. If you compare it with other popular algorithms like quicksort, marge sort, insertion sort and bubble sort, you will notice that this algorithm is going to do less memory write as compared to these algorithms. However, this is not the optimal algorithm in terms of memory write. There is another algorithm called cycle sort, which is optimal in terms of memory rights. Memory write can be a costly operation in situations like EEP ROM. In EEP ROM if we do more writes, age of this memory is reduces. So, in this type of situation, we prefer selection sort.

It is a basic idea for heapsort. The heapsort is based on selection sort only.

Selection sort is not stable.

It is a In-Place algorithm, it does not require extra memory for sorting.

**Definition**

The selection sort algorithm sorts an array by repeatedly finding the minimum element form unsort part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

* The subarray which is already sorted.
* Remaining subarray which is unsorted

**Working of selection sort:-**

* It finds the smallest element by traversing whole array and swape the element with element at first index.
* For the second pass, it will search for second smallest element and replay with second element.

**Approach:**

* Initialize minimum value(min\_idx) to location 0.
* Traverse the array to find the minimum element in the array.
* While traversing if any element smaller than min\_idx is found then swap both the values.
* Then, increment min\_idx to point to next element.
* Repeat until array is sorted.

**Code**

public static void selectionSort(int arr[], int n)

    {

        for(int i=0; i<n; i++)

        {

            int min = i;

            for(int j=i+1; j<n; j++)

            {

                if(arr[j] < arr[min])

                {

                    min = j;

                }

            }

            int temp = arr[i];

            arr[i] = arr[min];

            arr[min] = temp;

        }

**Time complexity: -**

* Time Complexity: O(n\*n), where n is the number of elements in the input array.
* Space Complexity: O(1).

1. **Insertion sort: -**

In insertion sort the array is virtually divided into a sorted and an unsorted part. Values form the unsorted part are picked and placed at the correct position in the sorted part.

* It is efficient for small data values.
* Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.
* Insertion is In-place and stable.

An algorithm is said to be in-place, if it is not required any extra auxiliary space.

* It is used for small arrays.
* Time complexity is O(n) in the best case.

The best case is happened when the array is already is sorted.

**Working of insertion sort:-**

Let’s given array is: -

Arr[] = {12, 11, 13, 5, 6};

First pass.

Initially, the first two element of an array are compared.

Here 12 is greater than 11. Thus, swap 11 and 12.

Second pass.

Now move to the next element and move the element of third place to virtual sort sub-array and then it will check the sub-array is sorted or not. If the sub-array is not sorted, then it will sort and move to the next element.

For third pass.

Now fourth element will move to sorted sub array and it will check if array is sorted or not. If array is sorted, then sort the array and move to the element.

Time complexity

* For best case: - O(n)
* For worst case: - O(n^2)
* In general : - O(n^2)

Code: -

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 key process that assumes that arr[l….m] and arr[m+1…m] are sorted and merges the two sorted sub-arrays into one in a sorted manner.

1. **Merge sort : -**

Merge sort is 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.

* It is stable algorithm. It maintains the original order of equal data.
* The merge() function is used for merging two halves.
* It well suited for linked List. Work in O(1) auxiliary space.
* Used in external sorting.
* The time complexity of merge sort is O(nlogn) and auxiliary space is O(n).
  1. **Merge two sorted arrays: -**

**Nive solution: -**

Time: O((m+n) \* log(m+n))

Aux: - theta(m+n)

Approach

* Let’s the two arrays are arr1 and arr2.
* Create a new array with the size is equal to the sum of size of both array(arr1 and arr2).
* Store the elements of both array in new array.
* Then sort the array using Arrays.sort() function.

static void fun(int arr1[], int arr2[])

    {

        //creating a new array of size m+n

        int m=arr1.length;

        int n=arr2.length;

        int arr[] = new int[m+n];

        //copy the element of first array.

        for(int i=0; i<m; i++)

        {

            arr[i] = arr1[i];

        }

        //copy the element of second element

        for(int j=0; j<n; j++)

        {

            arr[m+j] = arr2[j];

        }

        //sort the final array.

        Arrays.sort(arr);

        for (int i : arr) {

            System.out.print(i+ " ");

        }

    }

llllllnn