* What are different sorting alogorithm avaliable in java?

1. Selection Sort
2. **class** SelectionSort
3. {
4. **void** sort(**int** arr[])
5. {
6. **int** n = arr.length;
8. // One by one move boundary of unsorted subarray
9. **for** (**int** i = 0; i < n-1; i++)
10. {
11. // Find the minimum element in unsorted array
12. **int** min\_idx = i;
13. **for** (**int** j = i+1; j < n; j++)
14. **if** (arr[j] < arr[min\_idx])
15. min\_idx = j;
17. // Swap the found minimum element with the first
18. // element
19. **int** temp = arr[min\_idx];
20. arr[min\_idx] = arr[i];
21. arr[i] = temp;
22. }
23. }
25. // Prints the array
26. **void** printArray(**int** arr[])
27. {
28. **int** n = arr.length;
29. **for** (**int** i=0; i<n; ++i)
30. System.out.print(arr[i]+" ");
31. System.out.println();
32. }
34. // Driver code to test above
35. **public** **static** **void** main(String args[])
36. {
37. SelectionSort ob = **new** SelectionSort();
38. **int** arr[] = {64,25,12,22,11};
39. ob.sort(arr);
40. System.out.println("Sorted array");
41. ob.printArray(arr);
42. }
43. }

**Time Complexity**: O(n2) as there are two nested loops.

**Auxiliary Space**: O(1) : The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

2. Bubble Sort

1. **class** BubbleSort
2. {
3. **void** bubbleSort(**int** arr[])
4. {
5. **int** n = arr.length;
6. **for** (**int** i = 0; i < n-1; i++)
7. **for** (**int** j = 0; j < n-i-1; j++)
8. **if** (arr[j] > arr[j+1])
9. {
10. // swap arr[j+1] and arr[i]
11. **int** temp = arr[j];
12. arr[j] = arr[j+1];
13. arr[j+1] = temp;
14. }
15. }
17. /\* Prints the array \*/
18. **void** printArray(**int** arr[])
19. {
20. **int** n = arr.length;
21. **for** (**int** i=0; i<n; ++i)
22. System.out.print(arr[i] + " ");
23. System.out.println();
24. }
26. // Driver method to test above
27. **public** **static** **void** main(String args[])
28. {
29. BubbleSort ob = **new** BubbleSort();
30. **int** arr[] = {64, 34, 25, 12, 22, 11, 90};
31. ob.bubbleSort(arr);
32. System.out.println("Sorted array");
33. ob.printArray(arr);
34. }
35. }

Optimized Implementation:

The above function always runs O(n^2) time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn’t cause any swap.

1. **class** GFG
2. {
3. // An optimized version of Bubble Sort
4. **static** **void** bubbleSort(**int** arr[], **int** n)
5. {
6. **int** i, j, temp;
7. **boolean** swapped;
8. **for** (i = 0; i < n - 1; i++)
9. {
10. swapped = **false**;
11. **for** (j = 0; j < n - i - 1; j++)
12. {
13. **if** (arr[j] > arr[j + 1])
14. {
15. // swap arr[j] and arr[j+1]
16. temp = arr[j];
17. arr[j] = arr[j + 1];
18. arr[j + 1] = temp;
19. swapped = **true**;
20. }
21. }
23. // IF no two elements were
24. // swapped by inner loop, then break
25. **if** (swapped == **false**)
26. **break**;
27. }
28. }
30. // Function to print an array
31. **static** **void** printArray(**int** arr[], **int** size)
32. {
33. **int** i;
34. **for** (i = 0; i < size; i++)
35. System.out.print(arr[i] + " ");
36. System.out.println();
37. }
39. // Driver program
40. **public** **static** **void** main(String args[])
41. {
42. **int** arr[] = { 64, 34, 25, 12, 22, 11, 90 };
43. **int** n = arr.length;
44. bubbleSort(arr, n);
45. System.out.println("Sorted array: ");
46. printArray(arr, n);
47. }
48. }

Worst and Average Case Time Complexity: O(n\*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Bubble sort using recursion:

1. **public** **class** GFG
2. {
3. // A function to implement bubble sort
4. **static** **void** bubbleSort(**int** arr[], **int** n)
5. {
6. // Base case
7. **if** (n == 1)
8. **return**;
10. // One pass of bubble sort. After
11. // this pass, the largest element
12. // is moved (or bubbled) to end.
13. **for** (**int** i=0; i<n-1; i++)
14. **if** (arr[i] > arr[i+1])
15. {
16. // swap arr[i], arr[i+1]
17. **int** temp = arr[i];
18. arr[i] = arr[i+1];
19. arr[i+1] = temp;
20. }
22. // Largest element is fixed,
23. // recur for remaining array
24. bubbleSort(arr, n-1);
25. }
27. // Driver Method
28. **public** **static** **void** main(String[] args)
29. {
30. **int** arr[] = {64, 34, 25, 12, 22, 11, 90};
32. bubbleSort(arr, arr.length);
34. System.out.println("Sorted array : ");
35. System.out.println(Arrays.toString(arr));
36. }
37. }

3. **Insersion Sort**

1. **class** InsertionSort {
2. /\*Function to sort array using insertion sort\*/
3. **void** sort(**int** arr[])
4. {
5. **int** n = arr.length;
6. **for** (**int** i = 1; i < n; ++i) {
7. **int** key = arr[i];
8. **int** j = i - 1;
10. /\* Move elements of arr[0..i-1], that are
11. greater than key, to one position ahead
12. of their current position \*/
13. **while** (j >= 0 && arr[j] > key) {
14. arr[j + 1] = arr[j];
15. j = j - 1;
16. }
17. arr[j + 1] = key;
18. }
19. }
21. /\* A utility function to print array of size n\*/
22. **static** **void** printArray(**int** arr[])
23. {
24. **int** n = arr.length;
25. **for** (**int** i = 0; i < n; ++i)
26. System.out.print(arr[i] + " ");
28. System.out.println();
29. }
31. // Driver method
32. **public** **static** **void** main(String args[])
33. {
34. **int** arr[] = { 12, 11, 13, 5, 6 };
36. InsertionSort ob = **new** InsertionSort();
37. ob.sort(arr);
39. printArray(arr);
40. }
41. }

Time Complexity: O(n\*2)

Auxiliary Space: O(1)

Using recursion:

1. **public** **class** GFG
2. {
3. // Recursive function to sort an array using
4. // insertion sort
5. **static** **void** insertionSortRecursive(**int** arr[], **int** n)
6. {
7. // Base case
8. **if** (n <= 1)
9. **return**;
11. // Sort first n-1 elements
12. insertionSortRecursive( arr, n-1 );
14. // Insert last element at its correct position
15. // in sorted array.
16. **int** last = arr[n-1];
17. **int** j = n-2;
19. /\* Move elements of arr[0..i-1], that are
20. greater than key, to one position ahead
21. of their current position \*/
22. **while** (j >= 0 && arr[j] > last)
23. {
24. arr[j+1] = arr[j];
25. j--;
26. }
27. arr[j+1] = last;
28. }
30. // Driver Method
31. **public** **static** **void** main(String[] args)
32. {
33. **int** arr[] = {12, 11, 13, 5, 6};
35. insertionSortRecursive(arr, arr.length);
37. System.out.println(Arrays.toString(arr));
38. }
39. }

4. Quick Sort

1. **class** QuickSort
2. {
3. /\* This function takes last element as pivot,
4. places the pivot element at its correct
5. position in sorted array, and places all
6. smaller (smaller than pivot) to left of
7. pivot and all greater elements to right
8. of pivot \*/
9. **int** partition(**int** arr[], **int** low, **int** high)
10. {
11. **int** pivot = arr[high];
12. **int** i = (low-1); // index of smaller element
13. **for** (**int** j=low; j<high; j++)
14. {
15. // If current element is smaller than the pivot
16. **if** (arr[j] < pivot)
17. {
18. i++;
20. // swap arr[i] and arr[j]
21. **int** temp = arr[i];
22. arr[i] = arr[j];
23. arr[j] = temp;
24. }
25. }
27. // swap arr[i+1] and arr[high] (or pivot)
28. **int** temp = arr[i+1];
29. arr[i+1] = arr[high];
30. arr[high] = temp;
32. **return** i+1;
33. }

36. /\* The main function that implements QuickSort()
37. arr[] --> Array to be sorted,
38. low  --> Starting index,
39. high  --> Ending index \*/
40. **void** sort(**int** arr[], **int** low, **int** high)
41. {
42. **if** (low < high)
43. {
44. /\* pi is partitioning index, arr[pi] is
45. now at right place \*/
46. **int** pi = partition(arr, low, high);
48. // Recursively sort elements before
49. // partition and after partition
50. sort(arr, low, pi-1);
51. sort(arr, pi+1, high);
52. }
53. }
55. /\* A utility function to print array of size n \*/
56. **static** **void** printArray(**int** arr[])
57. {
58. **int** n = arr.length;
59. **for** (**int** i=0; i<n; ++i)
60. System.out.print(arr[i]+" ");
61. System.out.println();
62. }
64. // Driver program
65. **public** **static** **void** main(String args[])
66. {
67. **int** arr[] = {10, 7, 8, 9, 1, 5};
68. **int** n = arr.length;
70. QuickSort ob = **new** QuickSort();
71. ob.sort(arr, 0, n-1);
73. System.out.println("sorted array");
74. printArray(arr);
75. }
76. }

5. Merge Sort:

1. **class** MergeSort
2. {
3. // Merges two subarrays of arr[].
4. // First subarray is arr[l..m]
5. // Second subarray is arr[m+1..r]
6. **void** merge(**int** arr[], **int** l, **int** m, **int** r)
7. {
8. // Find sizes of two subarrays to be merged
9. **int** n1 = m - l + 1;
10. **int** n2 = r - m;
12. /\* Create temp arrays \*/
13. **int** L[] = **new** **int** [n1];
14. **int** R[] = **new** **int** [n2];
16. /\*Copy data to temp arrays\*/
17. **for** (**int** i=0; i<n1; ++i)
18. L[i] = arr[l + i];
19. **for** (**int** j=0; j<n2; ++j)
20. R[j] = arr[m + 1+ j];

23. /\* Merge the temp arrays \*/
25. // Initial indexes of first and second subarrays
26. **int** i = 0, j = 0;
28. // Initial index of merged subarry array
29. **int** k = l;
30. **while** (i < n1 && j < n2)
31. {
32. **if** (L[i] <= R[j])
33. {
34. arr[k] = L[i];
35. i++;
36. }
37. **else**
38. {
39. arr[k] = R[j];
40. j++;
41. }
42. k++;
43. }
45. /\* Copy remaining elements of L[] if any \*/
46. **while** (i < n1)
47. {
48. arr[k] = L[i];
49. i++;
50. k++;
51. }
53. /\* Copy remaining elements of R[] if any \*/
54. **while** (j < n2)
55. {
56. arr[k] = R[j];
57. j++;
58. k++;
59. }
60. }
62. // Main function that sorts arr[l..r] using
63. // merge()
64. **void** sort(**int** arr[], **int** l, **int** r)
65. {
66. **if** (l < r)
67. {
68. // Find the middle point
69. **int** m = (l+r)/2;
71. // Sort first and second halves
72. sort(arr, l, m);
73. sort(arr , m+1, r);
75. // Merge the sorted halves
76. merge(arr, l, m, r);
77. }
78. }
80. /\* A utility function to print array of size n \*/
81. **static** **void** printArray(**int** arr[])
82. {
83. **int** n = arr.length;
84. **for** (**int** i=0; i<n; ++i)
85. System.out.print(arr[i] + " ");
86. System.out.println();
87. }
89. // Driver method
90. **public** **static** **void** main(String args[])
91. {
92. **int** arr[] = {12, 11, 13, 5, 6, 7};
94. System.out.println("Given Array");
95. printArray(arr);
97. MergeSort ob = **new** MergeSort();
98. ob.sort(arr, 0, arr.length-1);
100. System.out.println("\nSorted array");
101. printArray(arr);
102. }
103. }

Note:

Q. Why Quick Sort is preferred over MergeSort for sorting Arrays?

Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Q. Why MergeSort is preferred over QuickSort for Linked Lists?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

6. Heap Sort:

1. **public** **class** HeapSort
2. {
3. **public** **void** sort(**int** arr[])
4. {
5. **int** n = arr.length;
7. // Build heap (rearrange array)
8. **for** (**int** i = n / 2 - 1; i >= 0; i--)
9. heapify(arr, n, i);
11. // One by one extract an element from heap
12. **for** (**int** i=n-1; i>=0; i--)
13. {
14. // Move current root to end
15. **int** temp = arr[0];
16. arr[0] = arr[i];
17. arr[i] = temp;
19. // call max heapify on the reduced heap
20. heapify(arr, i, 0);
21. }
22. }
24. // To heapify a subtree rooted with node i which is
25. // an index in arr[]. n is size of heap
26. **void** heapify(**int** arr[], **int** n, **int** i)
27. {
28. **int** largest = i; // Initialize largest as root
29. **int** l = 2\*i + 1; // left = 2\*i + 1
30. **int** r = 2\*i + 2; // right = 2\*i + 2
32. // If left child is larger than root
33. **if** (l < n && arr[l] > arr[largest])
34. largest = l;
36. // If right child is larger than largest so far
37. **if** (r < n && arr[r] > arr[largest])
38. largest = r;
40. // If largest is not root
41. **if** (largest != i)
42. {
43. **int** swap = arr[i];
44. arr[i] = arr[largest];
45. arr[largest] = swap;
47. // Recursively heapify the affected sub-tree
48. heapify(arr, n, largest);
49. }
50. }
52. /\* A utility function to print array of size n \*/
53. **static** **void** printArray(**int** arr[])
54. {
55. **int** n = arr.length;
56. **for** (**int** i=0; i<n; ++i)
57. System.out.print(arr[i]+" ");
58. System.out.println();
59. }
61. // Driver program
62. **public** **static** **void** main(String args[])
63. {
64. **int** arr[] = {12, 11, 13, 5, 6, 7};
65. **int** n = arr.length;
67. HeapSort ob = **new** HeapSort();
68. ob.sort(arr);
70. System.out.println("Sorted array is");
71. printArray(arr);
72. }
73. }

Heap sort is an in-place algorithm.

Its typical implementation is not stable, but can be made stable (See this)

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).