**Bubble sort**

* The swap will happen from left to right
* The largest value will be at the end after every pass
* If the array is already sorted then there will be total 1 pass.

Example : 2, 7, 4, 1, 5, 3

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

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

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

Utility.*swap*(arr, j, j+1);

}

}

1st iteration i=0 , j =0 to 4

2,4,1,5,3,7

2nd iterartion i=1, j =0 to 3

2,1,4,3,5,7

3rd iteration i=2, j= 0 to 2

1,2,3,4,5,7

4th iteration i=3, j =0 to 1

1,2,3,4,5,7

5th iteration i=4, j=0

1,2,3,4,5,7

Look at the example the array is already sorted at the mid way. So it is a average case

Average and worst case : O(n^2)

Best case : I will iterate only 1 time, so O(n)

But we need to modify the algorithm to meet the best and average case.

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

boolean flag =false;

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

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

Utility.*swap*(arr, j, j+1);

flag =true;

}

if(!flag)

break;

}

}

**Selection Sort**

* The sorting can be compared with cards. Where we can pick the minimum number one by one from one hand and place them in order in another hand , and we can get a sorted sequence
* If we compare two hands with two arrays, then selection sort will work like same way. But there can be time complexity if we use two arrays for large number of elements.
* We need to fix 0th position first, then find the minimum from rest of the elements and swap with 0th position, then do this for next iteration like 1st, 2nd and 3rd position.

Example :

2, 7, 4, 1, 5, 3

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

**int** minIndex = i;

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

**if**(arr[j]<arr[minIndex]) {

minIndex =j;

}

}

Utility.*swap*(arr, i, minIndex);

}

We are iterating I from 0 to n-2 because, when i=n-1 i.e. at the last position, we do not need to iterate over the array as it is already sorted by the time.

n =6

1st iteration i=0, j= 0 to 5

1,7,4,2,5,3

2nd iteration i=1, j=1 to 5

1,2,4,7,5,3

3rd iteration i=2, j=2 to 5

1,2,3,7,5,4

4th iteration i=3, j=3 to 5

1,2,3,4,5,7

5th iteration i=4 j =4 to 5

1,2,3,4,5,7

For all the cases we need to iterate over outer and the inner loop both, so for all the cases time complexity = O(n^2)

**Insertion Sort**

Go through this video <https://www.youtube.com/watch?v=i-SKeOcBwko>

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

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

**int** hole =i;

**while**(hole>0 && arr[hole-1]>val) {

arr[hole] = arr[hole-1];

hole--;

}

arr[hole] = val;

}

For best case when the array is already sorted arr[hole-1] > val will not satisfy,

so inner while loop never get executed, then the time complexity will be O(n-1) i.e. O(n)

For best and avg case O(n^2)

**Heap Sort**

Formulae:

* If index of a node is i then 2\*1+1 is left node index, 2\*i+2 is right node index.
* If n is the the number of elements in the array then n/2 to n-1 is the index of all leaf nodes.
* maximum heap means the root node is greater than left and right node.
* minimum heap means root node is less than left and right node.
* in both max and min heap there is no relation between two sibling nodes.

Example: 2, 7, 4, 1, 5, 3

Step 1: Build max heap

Step 2: swap the root node with the last leaf node.

step3 : decreament the n by 1 and again heapify the tree.

step 4: at last when n =1 we only left with one element, rest of the element after this element is already sorted.

time complexity is O(n)

<https://www.geeksforgeeks.org/heap-sort/>

**Quick Sort**

* This sort depends on divide and conquer rule, but unlike merge sort this is in place algorithm.

Step 1: Choose the pivot element wisely. it can be first, last or mid element. Let’s explain for first and last pivot

**Last element pivot :**

pivot = arr[end]

partition index = start

* Scan the array from i = start to end-1
* if i finds the smaller or equals to element of the pivot swap the arr[i] and arr[pationIndex]

Because our main motive is to smaller than pivot will be at left side and bigger than pivot will be at the right side. i is canning the array from left to right, so it’s responsibility is to find the larger element than pivot, so that after swapping the element should be at right.

* Now increament the partionIndex by 1.
* Repeat these steps until i < end or i != index of pivot element
* Now the left side of the partion index contains the elements less than pivot and

Right side elements contains larger element than pivot and the pivot itself. We know the responsibility of i is to find the larger element of pivot , so at the end also partion index will hold the larger element than pivot.

* Next step is to swap arr[partionIndex] and pivot i.e. arr[end].
* After swapping partIndex will hold the pivot element. old value of partion index will be at the end index.
* Pivot is now fixed at partIndex position, do not touch it.
* split the array from i= start to partIndex-1 and partIndex+1 to end , if start < end. For only one element we don’t need to split it .

**private** **int** partition(**int**[] arr, **int** start, **int** end){

**int** pivot = arr[end];

**int** partIndex = start;

**for**(**int** i = start ; i< end;i++){

**if**(arr[i]<= pivot){

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

arr[i] = arr[partIndex];

arr[partIndex] = temp;

// after swaping increament partIndex

partIndex++;

}

}

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

arr[end] = arr[partIndex];

arr[partIndex] = temp;

**return** partIndex;

}

**public** **void** sort(**int**[] arr,**int** start, **int** end){

**if**(start < end){

**int** partionIndex = partition(arr,start,end);

sort(arr,start,(partionIndex-1));

sort(arr,(partionIndex+1),end);

}

}

**First element pivot :**

pivot = arr[start]

partIndex = end

* Scan the array from i=end to i=start+1
* if i finds >= pivot element then swap it with partIndex and partIndex -- . because the main motive is same, move larger element of pivot to the right of partIndex.
* Repeat this step unitil i > start or i != the pivot element.
* at last all the smaller element including pivot is at the left side of partIndex and larger is at right side of the partionIndex.
* partIndex holds the value >= pivot as per the first step
* swap arr[start] or pivot with arr[partIndex]
* Now all the elements smaller than pivot is at left side of part index and larger will be at right side of partIndex.

Best case is when the pationindex is in such a place where n/2 elements is at the ;eft and n/2 is at the right side.

O(n \* log (n))

Worst case is when the array is already sorted, every time we split the array like one element at left rest of the element at right side or vice versa. O(n^2)

Averagge case Θ(n \* log(n) )

<https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort>

**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.