1. **Introduction**

We talk about sorting very much and in every part of our study of computer science sorting is studied more than others. The reasons for this are sorting is the most fundamental operation that is done most by the computer, the sorting algorithms are widely studied problem and different varieties of algorithms are known. Most of the ideas for algorithm design and analysis can be obtained from sorting algorithms like divide and conquer, lower bounds etc.

**Applications of Sorting:**

Sorting being the fundamental operation in computer science has many applications.

**Searching:**

Searching speeds up by the use of sorted elements. Binary search is an example that takes sorted sequence of keys to search and the whole searching operation is done in O(log n) time.

**Closest pair:**

Given *n* numbers, we have to find the pair which are closest to each other.Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an O(n) linear scan completes the job.

**Element uniqueness:**

Given n elements, are there any duplicate values for all elements or they are unique. When the elements are sorted we can check every adjacent elements by linear searching. **Frequency Distribution:**

Given n numbers of element, when the elements are sorted we can linearly search the elements to calculate the number of times the elements repeats. This can be applied for determining Huffman codes for each letters.

**Median and selection:**

Given n elements, to find kth largest or smallest element we sort the elements and find it in constant time (array is used).

**In-place Sorting**

A sorting algorithm is said to be in-place if it uses (1) memory—that is, it requires at most the local variables of the sorting function. Some of the sorting algorithms we will see will require a second array of equal size to the list being sorted; that is, O(n) additional memory. We will prefer in-place sorting algorithms. Insertion and quick sort are in-place sorting algorithm.

**What is stable Sorting ?**

A sorting algorithm is called **stable** if it keeps elements with equal keys in the same relative order in the output as they were in the input.

For example, in the following input the two 4's are indistinguishable:

1,4a,3,4b,2

And so the output of a stable sorting algorithm must be:

1. 1,2,3,4a,4b**bble Sort**

The bubble sort algorithm Compare adjacent elements . If the first is greater than the second, swap them. This is done for every pair of adjacent elements starting from first t

Bubble sort, merge sort, counting sort ,insertion sort are stable sorting methods. Most implementations of quicksort are not stable sorts.

1. **Bubble Sort:** two elements to last two elements.here the last element is the greatest one. The whole process is repeated except for the last one so that at each pass the comparison becomes fewer.

**Algorithm:**

*BubbleSort(A, n) {*

*for(i = n - 1; i > 0; i--)*

*for(j = 0; j < i; j++)*

*if(A[j] > A[j+1])*

*{*

*temp = A[j];*

*A[j] = A[j+1];*

*A[j+1] = A;*

*}*

*}*

The above algorithm for any instances of input runs in O(n2) time. This algorithm is similar to insertion sort algorithm studied earlier but operates in reverse order. However there is no best-case linear time complexity for this algorithm as of insertion sort.

[**Video Link**](https://www.youtube.com/watch?v=Jdtq5uKz-w4)

1. **Insertion**

efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

**INSERTION-SORT(*A*)**

1. **for *j*←2 to *length*[*A*]**
2. **do *key* ← *A*[*j*]**
3. **▹ Insert *A*[*j*] into the sorted sequence *A*[1 *j* - 1].**
4. ***i*←*j*-1**
5. **while *i*>0 and *A*[*i*]>*key***
6. **do *A*[*i* + 1] ← *A*[*i*]**
7. ***i*←*i*-1**
8. ***A*[*i*+1]←*key***

Our pseudo code for insertion sort is presented as a procedure called INSERTION-SORT, which takes as a parameter an array A[1 n] containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by length[A].) The input numbers are sorted in place: the numbers are rearranged within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when INSERTION-SORT is finished.

Obviously the time complexity of insertion sort is O(n2) due for inner loop.

[VideoLink](https://www.youtube.com/watch?v=i-SKeOcBwko)

1. **Selection**

In selection sort the all the elements are examined to obtain smallest (greatest) element at one pass then it is placed into its position, this process continues until the whole array is sorted.

**Algorithm:**

*Selection-sort(A) {*

*for( i = 0;i < n ;i++*

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

*if (A[j] < A[i]) swap(A[i],A[j])*

*}*

From the above algorithm it is clear that time complexity is O(n2). We can also see that for every instances of input complexity is O(n2).

[Video Link](https://www.youtube.com/watch?v=6nDMgr0-Yyo)

1. **Quick**

**Divide and conquer**

Divide and Conquer algorithms break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

There are three steps to applying Divide and Conquer algorithm in practice:

* Divide the problem into one or more subproblems.
* Conquer subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
* Combine the solutions to the subproblems into the solution for the original problem

[video Link](https://www.youtube.com/watch?v=COk73cpQbFQ)

**Quick Sort**

Quick sort developed by C.A.R Hoare is an unstable sorting. In practice this is the fastest sorting method. This algorithm is based on the divide and conquer paradigm. The main idea behind this sorting is partitioning of the elements.

Quicksort is based on the three-step process of divide-and-conquer.

To sort the subarray *A*[*p* ..*r*]:

* Divide: Partition *A*[ *p* . . *r* ], into two (possibly empty) subarrays *A*[ *p* . . *q* − 1] and *A*[*q*+1..*r*],such that each element in the first subarray *A*[*p*..*q*−1] is ≤ *A*[*q*] and *A*[*q*] is ≤ each element in the second subarray *A*[*q* + 1 . . *r*].
* Conquer: Sort the two subarrays by recursive calls to QUICKSORT.
* Combine: No work is needed to combine the subarrays, because they are sorted in place.

Perform the divide step by a procedure PARTITION, which returns the index *q* that marks the position separating the subarrays.

QUICKSORT(A, p,r)

if p < r

then q ← PARTITION(A, p,r)

QUICKSORT(A, p,q − 1)

QUICKSORT(A,q +1,r)

Initial call is QUICKSORT(A, 1, n).

Partitioning

Partition subarray A[ p . . r ] by the following procedure:

PARTITION(A, p,r)

x ← A[r] i←p−1

for j ← p to r − 1

do if A[j] ≤ x

then i ← i + 1

exchange A[i] ↔ A[j] exchange A[i + 1] ↔ A[r]

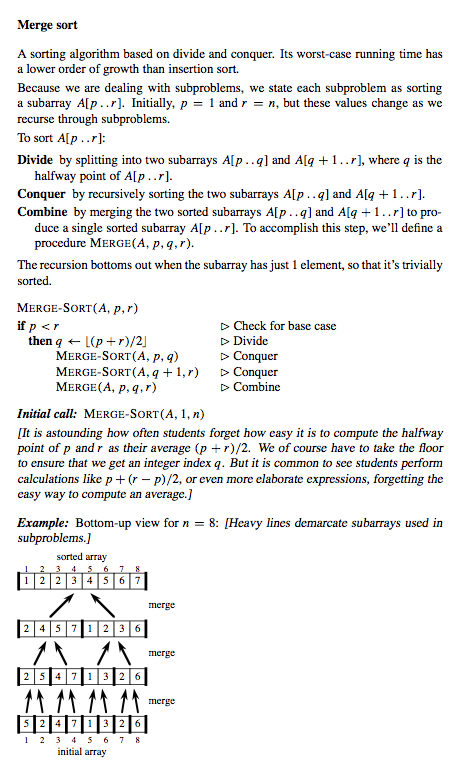
return i + 1

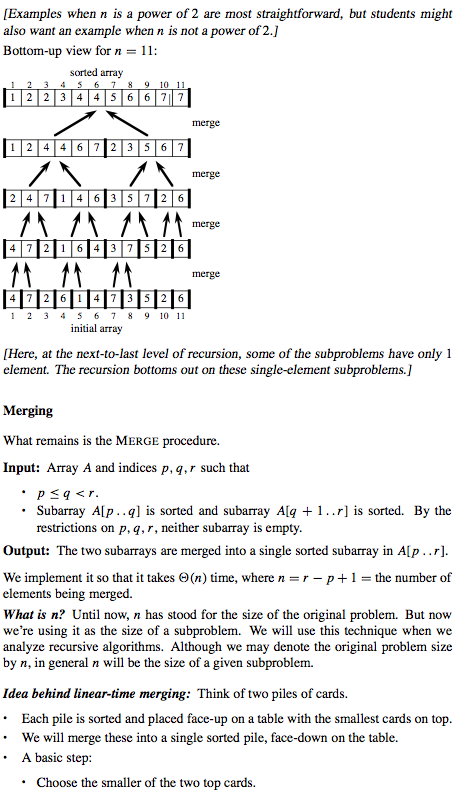
PARTITION always selects the last element A[r] in the subarray A[p ..r] as the pivot to the element around which to partition.

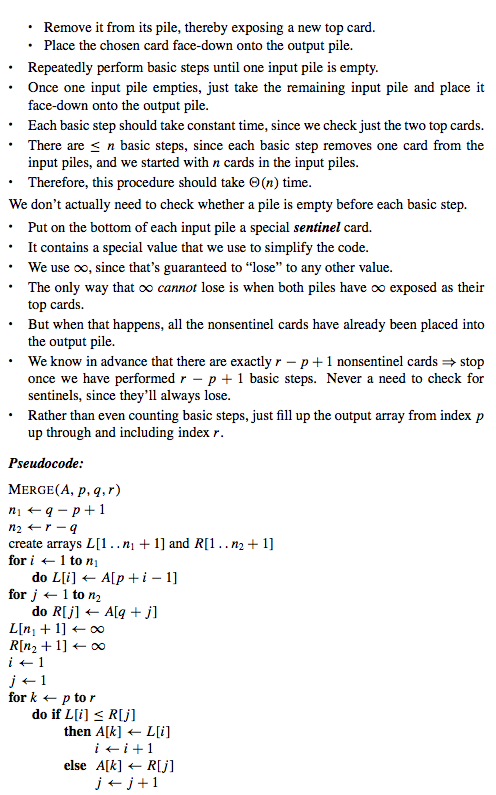
Analysis:

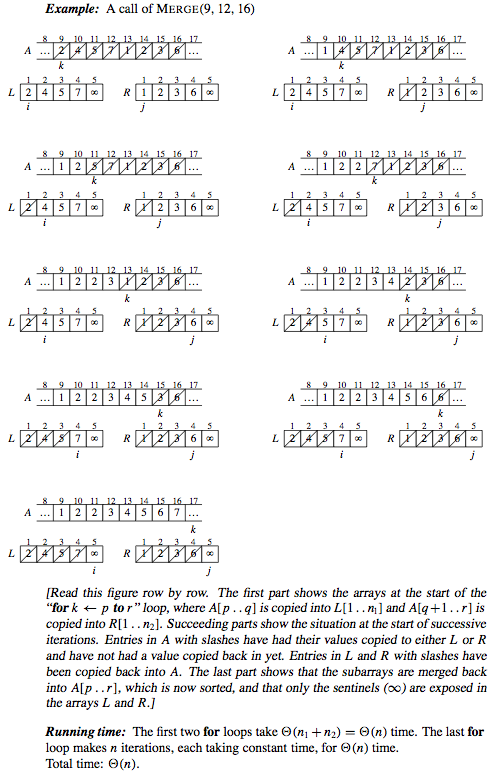
In the above algorithm either left pointer or right pointer moves at a time and scans the array at most 1 time. You can note that at most n swaps are done hence the time complexity for above algorithm is Θ(n).

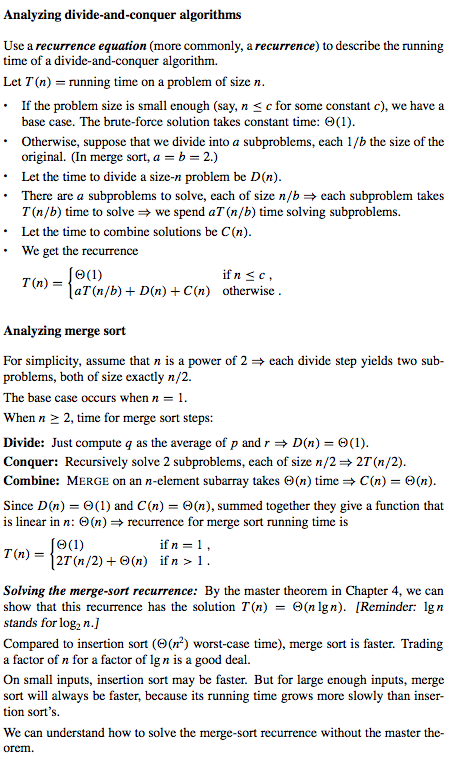
1. **Merge**











1. **Comparison and Efficiency of sorting**

