Algorithms

1) Sorting

Sorting :-

Insertion Sort

* Incremental Approach.
* in place sorting Algorithm.
* stable

When to use:
if number of elements is small or the array is almost sorted. (only few elements are misplaced)

Worst case for Insertion sort :-
if you are trying to sort in ascending order but input is given as descending order.

Best	Average	Worst
$O(n)$	$O(n^2)$	$O(n^2)$

$n = A.length$

```
Insertion_Sort(A)
    for j = 1 to n
        key = A[j]
        // insert A[j] into A[0 ... j-1]
        i = j-1
        while(i > 0 and A[j] > key)
            A[i+1] = A[i]
            i = i-1
        A[i+1] = key
```

Selection Sort

* default implementation is **not stable** however it can be made stable.
* in place.
* Selection sort makes $O(n)$ swaps which is minimum.

⇒

Best	Average	Worst
$O(n^2)$	$O(n^2)$	$O(n^2)$

* Selection sort is not very efficient algorithm when data sets are large.

Merge Sort

```
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
```

Recurrance :-

$$T(n) = 2T(n/2) + \theta(n)$$

Time Complexity

$O(n \log n)$

Space Complexity

$O(n)$

merge operation of linked list takes $O(1)$ auxiliary space.

* not inplace sorting
* stable sorting
* Divide & conquer

Drawbacks of Merge Sort:

- Slower compared to the other sort algorithms for smaller tasks.
- The merge sort algorithm requires an additional memory space of 0(n) for the temporary array.
- It goes through the whole process even if the array is sorted.

Quick Sort

```
/* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot */

int partition (int arr[], int low, int high)

{

   int pivot = arr[high]; // pivot

   int i = (low - 1); // Index of smaller element and indicates the right position of
pivot found so far

   for (int j = low; j <= high - 1; j++)

   {

      // If current element is smaller than the pivot

      if (arr[j] < pivot)

      {

         i++; // increment index of smaller element
         swap(&arr[i], &arr[j]);

      }

   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);

}
```

```
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Analysis:-

in general:

$T(n) = T(k) + T(n-k-1) + \theta(n)$

worst case

$T(n) = T(0) + T(n-1) + \theta(n)$

$= T(n-1) + \theta(n)$

↳ $\theta(n^2)$

occurs when the partition algorithm takes the greatest or smallest element as pivot.

Best Case

$T(n) = 2T(n/2) + \theta(n)$

↳ $O(n \log n)$

occurs when middle element is taken as pivot.

Average case

$T(n) = T(n/9) + T(9n/10) + \theta(n)$

$\theta(n^2)$

* default implementation is not stable.
* inplace
* Good cache locality (faster in virtual memory environment)

Heap Sort

- Comparision based sorting algorithm based on binary heap. (similar to selection sort)
- inplace.
- Typical implementation is **not stable**.
- 2-3 times slower than well implemented quick sort. (lack of Locality of reference)

Time Complexity: $O(n \log n)$

Space Complexity: $O(1)$

Radix Sort

⇒ digit by digit sort starting from the least significant digit to the most significant digit. It uses counting sort as a subroutine to sort.

⇒ non comparision based

⇒ not inplace sorting

Time Complexity

$O((n+b)*d)$

- n → size of input
- b → base
- d → no of digits in input numbers

if k is max possible value then

$d = \log_b k$

$O((n+b)*\log_b k)$