Sorting Algorithms

Why are sorting algorithms required?

Sorting finds application in following operations:

- Searching
- Minimum
- Maximum
- Duplicate deletion
- Frequency counting
- Uniqueness testing

Sorting can be accomplished using variety of techniques:

- Comparison
- Iterative
- Recursive
- Divide and conquer
- Randomized algorithms

Iterative sorting algorithms (comparison based)

- Bubble sort
- Insertion sort
- Heap sort

Recursive sorting algorithms (Comparison based/ Divide-and-conquer)

- Merge sort
- Quick sort

Bubble Sort

Concept – If we are given an array of *n* items, we can implement bubble sort as follows:

1. Compare pair of adjacent items
2. Swap if the items are out of order (non-ascending).
3. Repeat until the end of array. Largest item will '*bubbled*' to the end of last position.
4. Reduce *n* by 1 and go to Step # 1.

Pass # 1

31	12	16	39	15
12	31	16	39	15
12	16	31	39	15
12	16	31	39	15
12	16	31	15	39

Pass # 2

12	16	31	15	39
12	16	31	15	39
12	16	31	15	39
12	16	15	31	39

```
void BubbleSort (int array[] , int n) {
        for (int i = 0; i < n - 1; i++) {
                for (int j = 0; j < n-i-1; j++) {
                        if (a[j] > a[j+1])
                                swap(a[j] , a[j + 1]);
                        }}}
```

Analysis:

- Two nested loops.
- At iteration i = 0, inner loop runs n – 1 – 0 = n – 1 times. At iteration i = 1, inner loop runs n – 1 – 1 times.
- At iteration i = n – 1, inner loop runs n – 1 – (n – 1) = 0 times. Total number of runs = $\sum_{i=0}^{n-1}(n-i-1)$
- $\sum_{i=0}^{n-1}(n-i-1) = n^2 - n - \sum_{i=0}^{n-1} i$ $\sum_{i=1}^{m} j = \frac{m(m+1)}{2}$ substitute $m = n-1 \rightarrow \frac{(n-1)(n)}{2}$
- $\sum_{i=0}^{n-1}(n-i-1) = n^2 - n - \frac{(n-1)(n)}{2} = \frac{n^2}{2} - \frac{n}{2} = \frac{n(n-1)}{2} = \boldsymbol{O(n^2)}$ – Worst-case with input in descending order.

```
void BubbleSort (int array[] , int n) {
        for (int i = 0; i < n; i++) {
                bool is_sorted = true;
                for (int j = 0; j < n-i-1; j++) {
                        if (a[j] > a[j+1])
                                swap(a[j] , a[j + 1]);
                                is_sorted = false;}
                        if (is_sorted) return;}}
```

Analysis:

- Assume the array is sorted before inner loop
- Any swapping will invalidate the assumption
- If the flag remains true at the end of inner loop, array is already sorted.
- Only n iterations were required $\boldsymbol{O(n)}$ – Best-case with input in ascending order.

Sorting Algorithms

STUDY FOR FE

Insertion Sort

Concept – Similar to arranging a deck of card.

- Consider you have 5 cards (sorted already) out of a deck of cards.
- You are given another card and asked to insert it at the right spot.
- This will require going through all your cards.
- Once you find the right spot, you will insert the new card at correct spot.
- Process will have to be repeated for every new card.

Insertion sort works like this:

1. Start with second array element and make it **key**.
2. Compare **key** with the previous element.
3. Swap if previous element is larger than **key**.
4. Now make third array element as the **key**.
5. Continue until entire array is sorted.

0	1	2	3	4
6	5	3	1	8

key = 5

0	1	2	3	4
6	5	3	1	8

key < 6? Yes, so swap

0	1	2	3	4
5	6	3	1	8

0	1	2	3	4
5	6	3	1	8

key = 3

0	1	2	3	4
5	6	3	1	8

key < 6? Yes, so swap

0	1	2	3	4
5	3	6	1	8

key < 5? Yes, so swap

0	1	2	3	4
3	5	6	1	8

0	1	2	3	4
3	5	6	1	8

key = 1

0	1	2	3	4
3	5	6	1	8

key < 6? Yes, so swap

0	1	2	3	4
3	5	1	6	8

key < 5? Yes, so swap

0	1	2	3	4
3	1	5	6	8

key < 3? Yes, so swap

0	1	2	3	4
1	3	5	6	8

0	1	2	3	4
1	3	5	6	8

key = 8

0	1	2	3	4
1	3	5	6	8

key < 6? No, all values to left are smaller - sorted!

1	3	5	6	8

```
void insertionSort (int array[] , int n) {
        for (int i = 1; i < n; i++) {
                int key = a[i];
                for (int j = i-1; j >= 0 && a[j] > key; j--) {
                        a[j+1] = a[j]; }
                                a[j+1] = key;
                        }}
```

Analysis:

- Two nested loops.
- Outer-loop executes n-1 times.
- Best-case: array is already sorted i.e. a[j] > key is always false. $\boldsymbol{O(n)}$
- Worst-case: array is reversely sorted i.e. a[j] > key is always true. $\boldsymbol{O(n^2)}$