Sorting Algorithms

Sorting

- It’s a way to arrange the unordered collection in some order like ascending or descending

Types of sorting algorithms –

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort
7. Radix Sort
8. Shell Sort etc.

Bubble Sort

- Move from left to right end
- Compare the two elements and swap them if needed

Bubble Sort Illustration

Iteration 1:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	
7	7	5	13	2	3	9	3	1	0	**No Swapping**
7	7	5	13	2	3	9	3	1	0	**Swapping**
7	5	7	13	2	3	9	3	1	0	**No Swapping**
7	5	7	13	2	3	9	3	1	0	**Swapping**
7	5	7	2	13	3	9	3	1	0	**Swapping**
7	5	7	2	3	13	9	3	1	0	**Swapping**
7	5	7	2	3	9	13	3	1	0	**Swapping**

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	
7	5	7	2	3	9	3	13	1	0	**Swapping**
7	5	7	2	3	9	3	1	13	0	**Swapping**
7	5	7	2	3	9	3	1	0	13	**Swapping**

Final Position

Iteration 2:

5	7	2	3	7	3	1	0	9	13

Final Position

Iteration 3:

5	2	3	7	3	1	0	7	9	13

Iteration 4:

2	3	5	3	1	0	7	7	9	13

Iteration 5:

2	3	3	1	0	5	7	7	9	13

Iteration 6:

2	3	1	0	3	5	7	7	9	13

Iteration 7:

2	1	0	3	3	5	7	7	9	13

Iteration 8:

1	0	2	3	3	5	7	7	9	13

Iteration 9:

0	1	2	3	3	5	7	7	9	13

Bubble sort

Algorithm Bubble_sort(a,n): This algorithm sort the elements in ascending order. a is linear array which contains n elements. Variable temp is used to facilitate the swapping of two values. I and J are used as loop control variables.

```
1.  For I = 1 to n-1                                        C1
2.       For J = 0 to (n-I-1)                               C2
3.            If a[J] > a[J+1] then,                        C3
4.                 Set temp = a[J]                          C4
5.                 Set a[J] = a[J+1]                        C5
6.                 Set a[J+1] = temp                        C6
```

Analysis of Bubble Sort (version 1)

For Step 2 :

I =1 J= 0 to (n-2) i.e. total (n-1) and 1 for false. Hence n times

I =2 J =0 to (n-3) i.e. total (n-2) and 1 for false. Hence n-1 times

…….

I=n-1 J=0 to (n-1-n+1) i.e. total 1 and 1 for false. Hence 2 times.

Time Complexity $= n * C1 + \{n(n+1)/2 – 1\}* C2 + \{n(n+1)/2 – 2\}* (C3+C4+C5+C6)$

$= n + (n^2 + n - 2) /2 + 2*(n^2 + n - 4)$

$= O(n^2)$

Analysis of Bubble Sort (version 2)

From the above illustration, we observe following points –

In (n-1) iterations or passes array will become sorted.

Iteration 1: no. of comparisons (n-1)

Iteration 2: no. of comparisons (n-2)

Iteration 3: no. of comparisons (n-3)

………………..

Iteration k: no. of comparisons (n-k)

………………..

Iteration last: no. of comparisons 1

Time Complexity = Total Comparisons

$$= (n-1) + (n-2) + (n-3) + \ldots + (n-k) + \ldots + 3 + 2 + 1$$

$$= n(n-1)/2$$

$$= O(n^2)$$

<u>Property:</u> Once there is no swapping of elements in a particular pass then there will be further swapping in subsequent passes

Selection Sort

- Move from left to right end
- Each time least element gets its final position i.e. we select least element and put it at it’s final position

Selection Sort Illustration

Iteration 1:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	
7	7	5	13	2	3	9	3	1	0	**No Swapping**
7	7	5	13	2	3	9	3	1	0	**Swapping**
5	7	7	13	2	3	9	3	1	0	**No Swapping**
5	7	7	13	2	3	9	3	1	0	**Swapping**
2	7	7	13	5	3	9	3	1	0	**No Swapping**
2	7	7	13	5	3	9	3	1	0	**No Swapping**
2	7	7	13	5	3	9	3	1	0	**No Swapping**

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
2	7	7	13	5	3	9	3	1	0

Swapping

1	7	7	13	5	3	9	3	2	0

Swapping

0	7	7	13	5	3	9	3	2	1

Final Position

Iteration 2:

0	1	7	13	7	5	9	3	3	2

Final Position

Iteration 3:

0	1	2	13	7	7	9	5	3	3

Iteration 4:

0	1	2	3	13	7	9	7	5	3

Iteration 5:

0	1	2	3	3	13	9	7	7	5

Iteration 6:

0	1	2	3	3	5	13	9	7	7

Iteration 7:

0	1	2	3	3	5	7	13	9	7

Iteration 8:

0	1	2	3	3	5	7	7	13	9

Iteration 9:

0	1	2	3	3	5	7	7	9	13

Selection sort

Algorithm Select_sort(a,n): This algorithm sort the elements in ascending order. a is linear array which contains n elements. Variable temp is used to facilitate the swapping of two values. I and J are used as loop control variables.

```
1.  For I = 0 to n-2
2.          For J = I+1 to (n-1)
3.                  If a[I] > a[J] then,
4.                          Set temp = a[I]
5.                          Set a[I] = a[J]
6.                          Set a[J] = temp
```

Analysis of Selection Sort

From the above illustration, we observe following points –

In (n-1) iterations or passes array will become sorted.

Iteration 1:	no. of comparisons (n-1)
Iteration 2:	no. of comparisons (n-2)
Iteration 3:	no. of comparisons (n-3)
………………..	
Iteration k:	no. of comparisons (n-k)
………………..	
Iteration last:	no. of comparisons 1

Time Complexity = Total Comparisons

= (n-1) + (n-2) + (n-3) + ….+ (n-k) + …. + 3 + 2 + 1

= n(n-1)/2

= $O(n^2)$

Insertion Sort

- Find the element smaller than previous elements
- Create a space by shifting or moving the elements to next position
- Insert the element at empty space

Insertion Sort Illustration

Pass 1:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
7	7	5	13	2	3	9	3	1	0

Pass 2:

7	7	5	13	2	3	9	3	1	0
5	7	7	13	2	3	9	3	1	0

Pass 3:

5	7	7	13	2	3	9	3	1	0

Pass 4:

5	7	7	13	2	3	9	3	1	0

2	5	7	7	13	3	9	3	1	0

2	5	7	7	13	3	9	3	1	0

Pass 5:

2	5	7	7	13	3	9	3	1	0

2	3	5	7	7	13	9	3	1	0

2	3	5	7	7	13	9	3	1	0

Pass 6:

2	3	5	7	7	13	9	3	1	0

2	3	5	7	7	9	13	3	1	0

2	3	5	7	7	9	13	3	1	0

Pass 7:

2	3	5	7	7	9	13	3	1	0

2	3	3	5	7	7	9	13	1	0

2	3	3	5	7	7	9	13	1	0

Pass 8:

2	3	3	5	7	7	9	13	1	0

1	2	3	3	5	7	7	9	13	0

1	2	3	3	5	7	7	9	13	0

Pass 9:

1	2	3	3	5	7	7	9	13	0

0	1	2	3	3	5	7	7	9	13

Insertion sort

Algorithm Insert_sort(a,n): This algorithm sort the elements in ascending order. a is linear array which contains n elements. Variable temp is used to facilitate the swapping of two values. I ,J and K are used as loop control variables.

```
1.  For I = 1 to n-1
2.        Key = a[I]
3.        J = I - 1
4.        While J>=0 and a[J] > Key
5.              Set a[J+1] = a[J]
6.              J = J - 1
7.        a[J+1] = Key
```

Analysis of Insertion Sort

From the above illustration, we observe following points –

In (n-1) iterations or passes array will become sorted.

Iteration 1:	no. of comparisons 1
Iteration 2:	no. of comparisons 2
Iteration 3:	no. of comparisons 3
…………….	
Iteration last:	no. of comparisons n-1

Time Complexity = Total Comparisons
= 1 + 2 + 3 + …….. + (n-2) + (n-1)
= $n(n-1)/2$
= $O(n^2)$

Radix Sort

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

Radix sort uses counting sort as a subroutine to sort.

Radix Sort Illustration

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

[*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u>

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

8<u>0</u>2, 2, <u>2</u>4, <u>4</u>5, <u>6</u>6, 1<u>7</u>0, <u>7</u>5, <u>9</u>0

Sorting by the most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, <u>1</u>70, <u>8</u>02

Radix Sort Algorithm

Step 1: Find the maximum number in ARR as Max

Step 2: Calculate the Number of digits in Max and SET NOS = number of digit

Step 3: Repeat Step 4 to 8 for PASS = 1; PASS <= NOS

Step 4: Repeat Step 5 to 7 for I=0 to I < Size of ARR

Step 5: SET DIGIT = Arr[I]

Step 6: Insert element Arr[I] to the bucket at index DIGIT

Step 7: Do Increment in bucket count for index DIGIT
[END OF FOR Loop]

Step 8: Pick elements from the bucket starting from index 0 and put in ARR
[END OF For LOOP]

Step 9: END

Analysis of Radix Sort

What is the running time of Radix Sort?

Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for the decimal system, b is 10. What is the value of d?

If k is the maximum possible value, then d would be $O(\log_b(k))$.

So overall time complexity is $O((n+b) * \log_b(k))$. Which looks more than the time complexity of comparison-based sorting algorithms for a large k. Let us first limit k. Let $k <= n^c$ where c is a constant. In that case, the complexity becomes $O(n\text{Log}_b(n))$. But it still doesn't beat comparison-based sorting algorithms.

What if we make the value of b larger?. What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n).

In other words, we can sort an array of integers with a range from 1 to n^c if the numbers are represented in base n (or every digit takes $\log_2(n)$ bits).