Lecture 7 :Sorting Algorithms

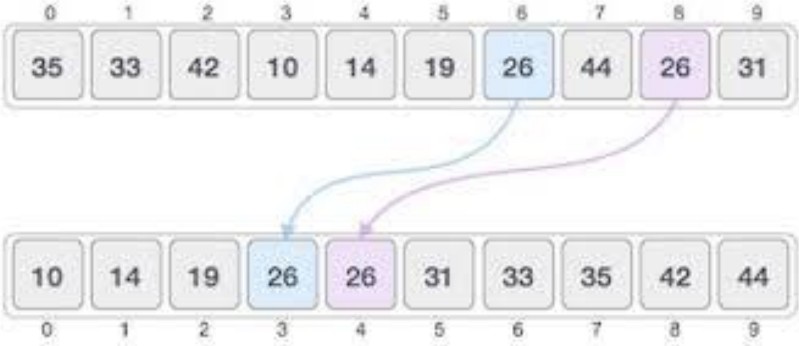
Sorting

- Arranging elements of set into order.

- Let A be a list of *n* elements A1, A2, ……. An in memory. Sorting of list A refers to the operation of rearranging the contents of A so that they are in increasing (or decreasing) order (numerically or lexicographically); A1 < A2 < A3 < …… < An.

- Examples of Sorting:

  - Words in a dictionary are sorted.
  - Files in a directory are often listed in sorted order.
  - The index of a book is sorted.



Sorting Algorithms

- There are many, many different types of sorting algorithms, but the primary ones are:

  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Shell Sort
  - Radix Sort
  - Swap Sort
  - Heap Sort

Bubble sort

- In bubble sort:
  - Each element is compared with its adjacent element.
  - If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed.
  - Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Bubble sort (steps)

- ***Step 1:***

  - Compare A[1] and A[2] and arrange them in the (or desired) ascending order, so that A[1] < A[2].

  - if A[1] is greater than A[2] then interchange the position of data by swap = A[1]; A[1] = A[2]; A[2] = swap.

  - Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Continue the process until we compare A[N – 1] with A[N].

Bubble sort (steps)

- ***Step 2:***
  - Repeat step 1 with one less comparisons that is, now stop comparison at A [n – 1] and possibly rearrange A[N – 2] and A[N – 1] and so on.
- **Step *n* – 1:**
  - Compare A[1]with A[2] and arrange them so that A[1] < A[2].

Bubble sort example

- **First Pass:**
  - (**5 1** 4 2 8) (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  - (1 **5 4** 2 8) (1 **4 5** 2 8), Swap since 5 > 4
  - (1 4 **5 2** 8) (1 4 **2 5** 8), Swap since 5 > 2
  - (1 4 2 **5 8**) (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Bubble sort example

- **Second Pass:**
  - (**1 4** 2 5 8) (**1 4** 2 5 8)
  - (1 **4 2** 5 8) (1 **2 4** 5 8), Swap since 4 > 2
  - (1 2 **4 5** 8) (1 2 **4 5** 8)
  - (1 2 4 **5 8**) (1 2 4 **5 8**)
  - Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Bubble sort example

- **Third Pass:**
  - (**1** **2** 4 5 8) (**1** **2** 4 5 8)
  - (1 **2** **4** 5 8) (1 **2** **4** 5 8)
  - (1 2 **4** **5** 8) (1 2 **4** **5** 8)
  - (1 2 4 **5** **8**) (1 2 4 **5** **8**)

Bubble sort example

i = 1	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 2	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 3	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 4	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			

Bubble sort example

i = 5	0	1	2	3	4	5
	1	1	2	3	4	
	2	1	2	3	4	
i = 6	0	1	2	3	4	
	1	1	2	3		
i = 7	0	1	2	3		
		1	2			

Bubble sort algorithm

- Let A be a linear array of n numbers. Swap is a temporary variable for swapping (or interchange) the position of the numbers.

1. Input n numbers of an array A
2. Initialize $i = 0$ and repeat through step 4 if $(i < n)$
3. Initialize $j = 0$ and repeat through step 4 if $(j < n - i - 1)$
4. If $(A[j] > A[j + 1])$

(a) Swap = A[j]

(b) $A[j] = A[j + 1]$

(c) A[j + 1] = Swap

5. Display the sorted numbers of array A
6. Exit.

bubble()

```
void bubbleSort (int a[ ] , int size)
{
   int i, j, temp;
   for ( i = 0; i < size; i++ )    /* controls passes through the list */
   {
            for ( j = 0; j < size - 1; j++ )   /* performs adjacent comparisons */
            {
                        if ( a[ j ] > a[ j+1 ] )   /* determines if a swap should occur */
                        {
                                    temp = a[ j ];      /* swap is performed */
                                    a[ j ] = a[ j + 1 ];
                                    a[ j+1 ] = temp;
                        }
            }
    }
}
```

Bubble Sort – Analysis

- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - The number of moves: 0 ➔ O(1)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **O(n^2)**
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 ➔ O(n^2)
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ O(n^2)
- Average-case: ➔ **O(n^2)**
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is O(n^2)**

Insertion sort

- Insertion sort algorithm:
  - It sorts a set of values by inserting values into an existing sorted file.
  - Compare the second element with first, if the first element is greater than second, place it before the first one.
  - Otherwise place is just after the first one.
  - Compare the third value with second and then with first and so on.

Insertion sort: steps

- **Step 1:**
  - As the single element A [1] by itself is sorted array.
- **Step 2:**
  - A [2] is inserted either before or after A [1] by comparing it so that A[1], A[2] is sorted array.
- **Step 3:**

  A [3] is inserted into the proper place in A [1], A [2], that is A [3] will be compared with A [1] and A [2] and placed before A [1], between A [1] and A[2], or after A [2] so that A [1], A [2], A [3] is a sorted array.

- **Step n:**

  A [n] is inserted into its proper place in an array A [1], A [2], A [3], ... A [n –1] so that A [1], A [2], A [3], ... ,A [n] is a sorted array.

Play Cards Example

Values for 4 Play Cards are :

5 9 3 1

Step 1: Compare first two cards 5 with 9 and get the highest number 9.

Step 2 : Compare the highest number from step1 with another play card 9 and 3 and get the highest number 9.

Step 3: Compare the highest number from step 2 with another play card, 9 and 1 and get the highest number 9.

Remaining Cards are:

5 3 1

Step 1: Compare first two play cards 5 and 3 and get the highest number 5.

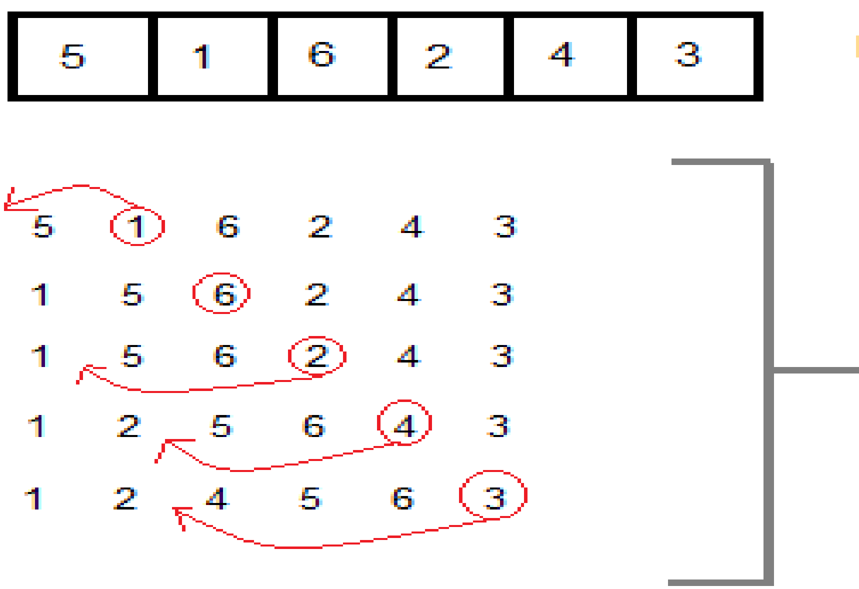
Step 2 : compare highest number from step 1 with another play card 5,1 and get highest number 5.

Remaining Cards are: 3 1

Step 1: Compare the remaining cards are 3 and 1 and get the highest no 3.

Finally, we have total 6 no. of comparisons.

Example



Lets take this Array.

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

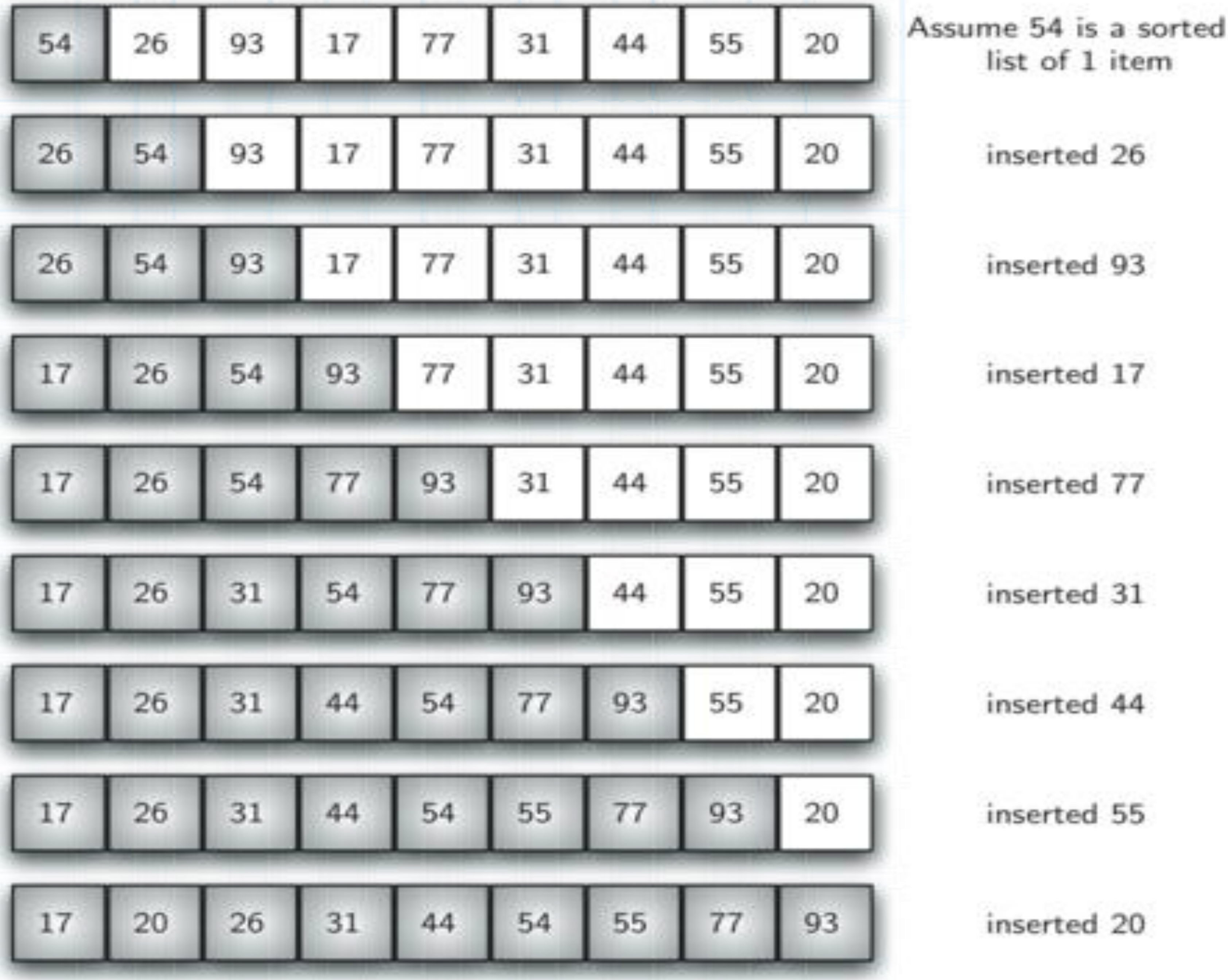
1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

(Always we start with the second element as key.)


54 26 93 17 77 31 44 55 20
Assume 54 is a sorted list of 1 item
26 54 93 17 77 31 44 55 20
inserted 26
26 54 93 17 77 31 44 55 20
inserted 93
17 26 54 93 77 31 44 55 20
inserted 17
17 26 54 77 93 31 44 55 20
inserted 77
17 26 31 54 77 93 44 55 20
inserted 31
17 26 31 44 54 77 93 55 20
inserted 44
17 26 31 44 54 55 77 93 20
inserted 55
17 20 26 31 44 54 55 77 93
inserted 20

Algorithm

- Let A be a linear array of n numbers A [1], A [2], A [3], ... ,A [n]......Swap be a temporary variable to interchange the two values. Pos is the control variable to hold the position of each pass.

1. Input an array A of *n numbers*
2. Initialize *i = 1 and repeat through steps 4 by incrementing i by one.*
   (*a*) *If (i < = n – 1)*
   (*b*) *Swap = A [i],*
   (*c*) *Pos = i – 1*
3. Repeat the step 3 if (Swap < A[Pos] and (Pos >= 0))
   (*a*) *A [Pos+1] = A [Pos]*
   (*b*) *Pos = Pos-1*
4. A [Pos +1] = Swap
5. Exit

Selection sort

- **Selection sort :**
  - It finds the smallest element of the array and interchanges it with the element in the first position of the array.
  - Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

Selection sort: steps

- Let A be a linear array of ‘*n*’ numbers, A [1], A [2], A [3],...... A [*n*].

- ***Step* 1:**

  - Find the smallest element in the array of *n* numbers A[1], A[2], ... A[*n*].

  - Let LOC is the location of the smallest number in the array.

  - Then interchange A[LOC] and A[1] by swap = A[LOC]; A[LOC] = A[1]; A[1] = Swap.

Selection sort: steps

- ***Step* 2:**
  - Find the second smallest number in the sub list of $n - 1$ elements A [2] A [3]…… A [$n - 1$] (first element is already sorted).
  - Now we concentrate on the rest of the elements in the array.
  - Again A [LOC] is the smallest element in the remaining array and LOC the corresponding location then interchange A [LOC] and A [2].
  - Now A [1] and A [2] is sorted, since A [1] less than or equal to A [2].

Selection sort: steps

- ***Step n – 1:***
  - Find the n – 1 smallest number in the sub array of 2 elements (*i.e.*, A(n–1), A (n)).
  - Consider A [LOC] is the smallest element and LOC is its corresponding position.
  - Then interchange A [LOC] and A(n – 1).
  - Now the array A [1], A [2], A [3], A [4],……….A [n] will be a sorted array.

Selection sort: algorithm

- Let A be a linear array of *n* numbers A [1], A [2], A [3], ……… A [k], A [k+1], …….. A [n]. *Swap* be a temporary variable for swapping (or interchanging) the position of the numbers. *Min* is the variable to store smallest number and *Loc* is the location of the smallest element.

1. Input *n* numbers of an array A
2. Initialize $i = 0$ and repeat through step5 if $(i < n - 1)$
   - (a) min = $a[i]$
   - (b) loc = I
3. Initialize $j = i + 1$ and repeat through step 4 if $(j < n - 1)$
4. if (a[j] < min)
   - (a) min = a[j]
   - (b) loc = j
5. if (loc ! = i)
   - (a) swap = $a[i]$
   - (b) $a[i] = a$[loc]
   - (c) a[loc] = swap
6. display "the sorted numbers of array A"
7. Exit

Selection Sort: Example (ascending)

- 70 75 89 61 37
  - Smallest is 37
  - Swap with index 0
- 37 75 89 61 70
  - Smallest is 61
  - Swap with index 1
- 37 61 89 75 70
  - Smallest is 70
  - Swap with index 2
- 37 61 70 75 89
  - Smallest is 75
  - Swap with index 3
    - Swap with itself
- 37 61 70 75 89
  - Don't need to do last element because there's only one left
- 37 61 70 75 89

Selection Sort: Example

[0]	[1]	[2]	[3]	[4]	
5	1	3	7	2	find min
1	5	3	7	2	swap to index 0
1	5	3	7	2	find min
1	2	3	7	5	swap to index 1
1	2	3	7	5	find min
1	2	3	7	5	swap to index 2
1	2	3	7	5	find min
1	2	3	5	7	swap to index 3

Selection Sort: Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

  So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.

  - Ignoring other operations does not affect our final result.

- In selectionSort function, the outer for loop executes n-1 times.
- We invoke swap function once at each iteration.

  Total Swaps: n-1

  Total Moves: 3*(n-1)   (Each swap has three moves)

Selection Sort: Analysis

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.
  - # of key comparisons = 1+2+...+n-1 = n*(n-1)/2
  - **So, Selection sort is O(n^2)**
- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **O(n^2)**
  - This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
  - Since O(n^2) grows so rapidly, the selection sort algorithm is appropriate only for small n.
  - Although the selection sort algorithm requires O(n^2) key comparisons, it only requires O(n) moves.
  - A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).