Data Structures and Algorithms
(ES221)

Sorting

Dr. Zubair Ahmad

- **Attendance?**

  - Active Attendance
  - **Dead Bodies.**
  - **Active Minds**
  - Mobiles in hands -> Mark as absent
  - 80% mandatory

Sorting

Sorting is the process of ordering a list of objects, according to some linear order, such as

$$a_1 \leq a_2 \leq a_3 \leq a_4 \leq a_5 \leq a_6$$

- **Internal** sorting
  - When all the data is stored in the main memory
  - The random access nature of the memory can be utilized
- **External** sorting
  - When the size of data too large
  - The main memory cannot accommodate all the data

3

Sorting

- **External** sorting
  - The bottleneck is usually the data movement between the secondary storage and the main memory.
  - Data movement is efficient if it is moved in the form of large blocks.
  - However, moving large blocks of data is efficient only if it is physically located in contiguous locations.

4
4

Sorting

- The simplest algorithms usually take $O(n^2)$ time to sort n objects, and suited for sorting short lists.

- One of the most popular algorithms is Quick-Sort takes $O(n\log n)$ time on average.

- Quick-Sort works for most common applications, although in worst case it can take $O(n^2)$ time.

5

Sorting

- There are other sorting techniques, such as Merge-Sort and Heap-Sort that take time *O(n*log*n)* in worst case.

- Merge-Sort however, is well suited for external sorting.

- There are other algorithms such as “bucket” and “radix” sort when the keys are integers of known range. They take time O(*n*).

6

Sorting

- The sorting problem is to arrange a sequence of records so that the values of their key fields form a non-decreasing sequence.

- Given records $r_1, r_1, \cdots. r_n$ with key values $k_1, k_1, \cdots. k_n$, respectively we must produce the same records in an order $r_{i_1}, r_{i_2}, \cdots. r_{i_n}$ such that the keys are in the corresponding non-decreasing order.

$$key(r_{i_1}) \leq key(r_{i_2}) \leq key(r_{i_3}) \leq key(r_{i_4}) \leq key(r_{i_5}) \leq key(r_{i_6})$$

$$\rightarrow \quad k_{i_1} \leq k_{i_2} \leq k_{i_3} \leq k_{i_4} \leq k_{i_5} \leq k_{i_6}$$

- The records may NOT have distinct values, and can appear in any order.

7

Bubble Sort

One of the simplest sorting methods.

The basic idea is the "weight" of the record.

The records are kept in an array held vertically.

"light" records bubbling up to the top.

We make repeated passes over the array from bottom to top.

If two adjacent elements are out of order i.e. "lighter" one is below, we reverse the order.

8

Bubble Sort

The overall effect, is that after the first pass the “lightest” record will bubble all the way to the top.

On the second top pass, the second lowest rises to the second position, and so on.

On second pass we need not try bubbling to the top position, because we know that the lightest record is already there.

9

Bubble Sort

```
int n =N; // N is the size of the array;
  for (int i = 0; i < N; i++){
    for (int j = 1; j <  n; j++) {
          if (A[j] < A[j-1]) {
              swap(j-1 , j);
           }//end if
     } //end inner for
} //end inner for
```

Complexity ?

$O(N^2)$

➔

Algorithm does not exit until all the data is checked

```
// Swap function assumes that
// A[n] is a globally declared array
swap ( x , y) {
 int temp = A[x];
A[x] = A[y];
A[y] = temp;
}
```


10

Bubble Sort

```
int n =N; // N is the size of the array;
  for (int i = 0; i < N; i++){
      int swapped = 0;
      for (int j = 1; j <  n; j++) {
          if (A[j] < A[j-1]) {
              swap(j-1 , j);
              swapped = 1;
           }//end if
      } //end inner for
     // n = n-1;
      if (swapped == 0)
          break;
  } //end inner for
```

```
// Swap function assumes that
// A[n] is a globally declared array
swap ( x , y) {
 int temp = A[x];
A[x] = A[y];
A[y] = temp;
}
```

➔ Algorithm exits if no swap done in previous (outer loop) step

Complexity ?

$O(N^2)$

Bubble Sort

```
int n =N; // N is the size of the array;
 for (int i = 0; i < N; i++){
    int swapped = 0;
     for (int j = 1; j <  n; j++) {
         if (A[j] < A[j-1]) {
              swap(j-1 , j);
              swapped = 1;
          }//end if
    } //end inner for
    n = n-1;
    if (swapped == 0)
        break;
 } //end inner for
```

```
// Swap function assumes that
// A[n] is a globally declared array
swap ( x , y) {
 int temp = A[x];
A[x] = A[y];
A[y] = temp;
}
```

No bubbling to the top position, because the lightest record is already there.

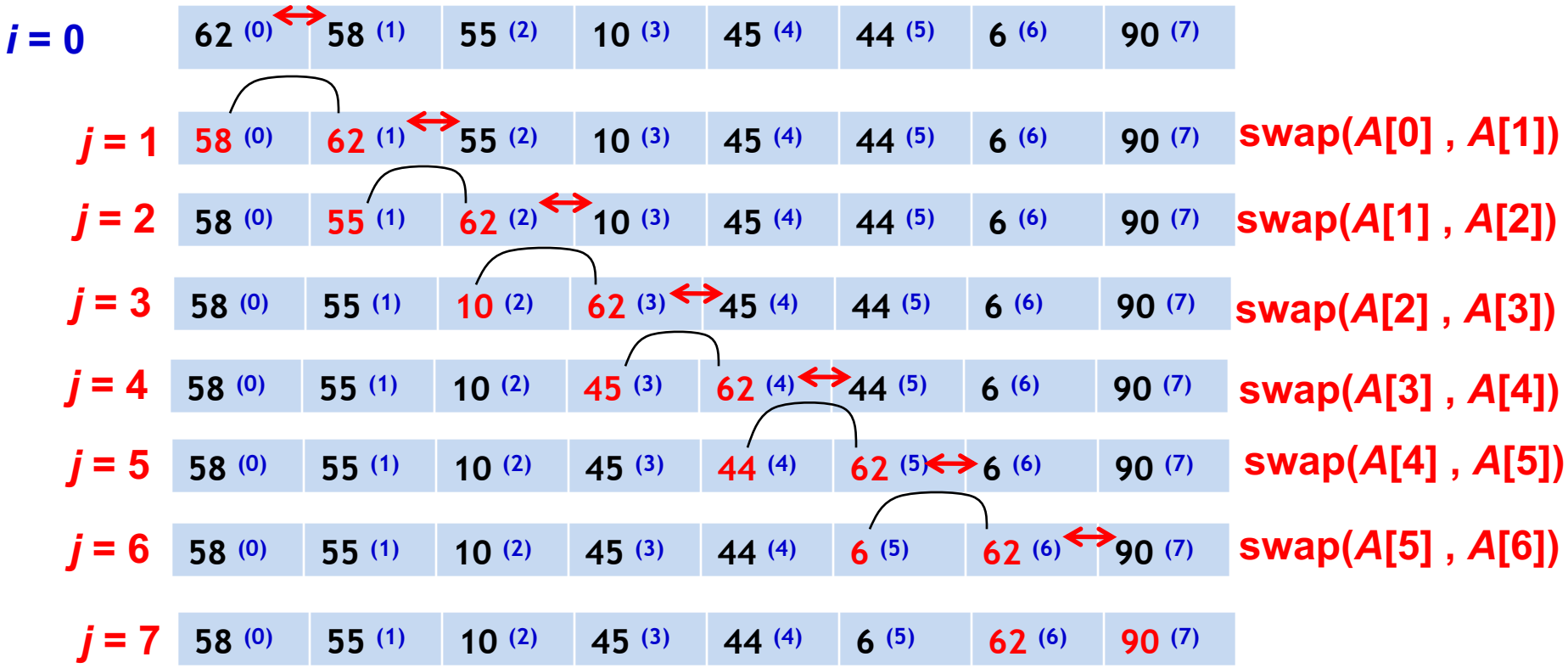
➔

Algorithm exits if no swap done in previous (outer loop) step

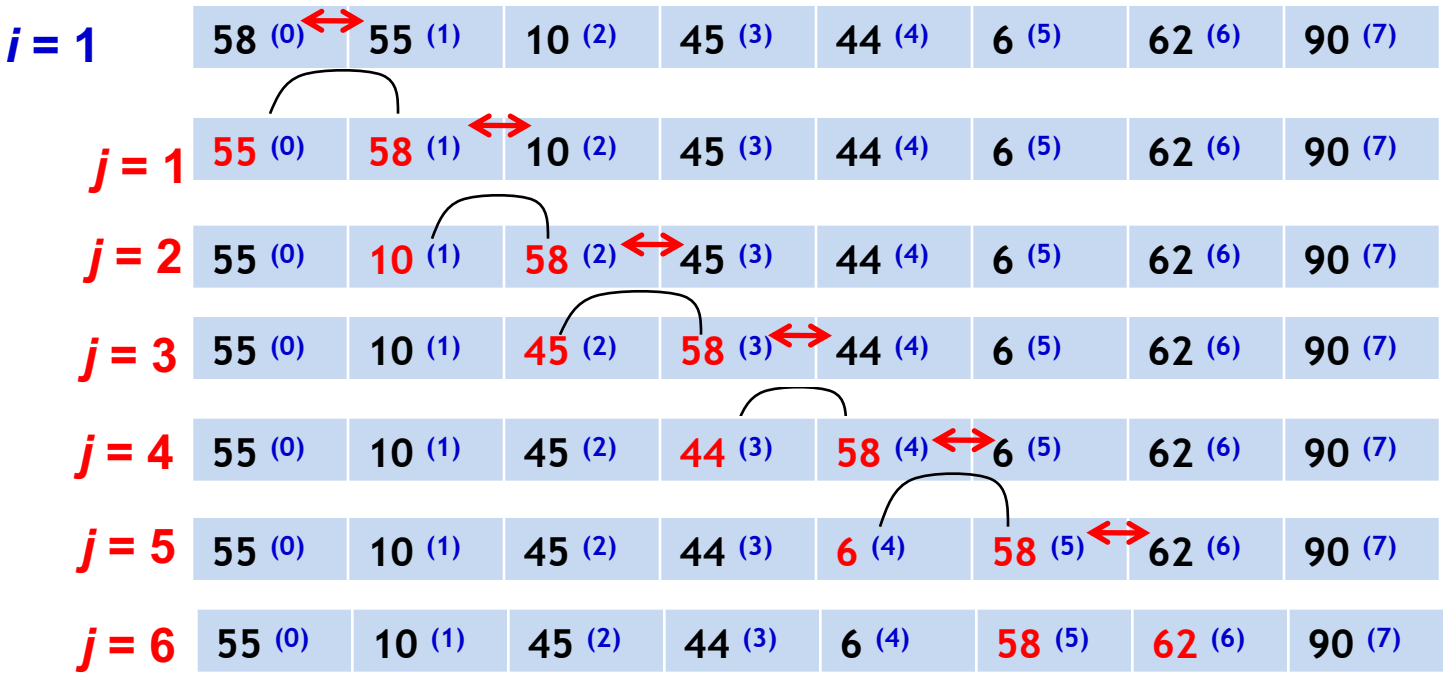
Complexity ?

$O(N^2)$

Bubble Sort Example (First Pass)



Bubble Sort Example (Second Pass)



swap(*A*[0] , *A*[1])

swap(*A*[1] , *A*[2])

swap(*A*[2] , *A*[3])

swap(*A*[3] , *A*[4])

swap(*A*[4] , *A*[5])

Bubble Sort Example (Third Pass)

i = 2

55 (0)	10 (1)	45 (2)	44 (3)	6 (4)	58 (5)	62 (6)	90 (7)

j = 1

j = 2

j = 3

j = 4

j = 5

10 (0)	45 (1)	44 (2)	6 (3)	55 (4)	58 (5)	62 (6)	90 (7)

```
int n =N; // N is the size of the array;
 for (int i = 0; i < N; i++){
    int swapped = 0;
    for (int j = 1; j <  n; j++)
            if (A[j] < A[j-1]) {
                swap(j-1 , j); swapped = 1;
             }//end if
     n = n-1;
    if (swapped == 0)
            break;            }//End outer for
```

Bubble Sort Example (Fourth Pass)

i = 3

10 (0)	45 (1)	44 (2)	6 (3)	55 (4)	58 (5)	62 (6)	90 (7)

j = 1

j = 2

j = 3

j = 4

10 (0)	44 (1)	6 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

```
int n =N; // N is the size of the array;
 for (int i = 0; i < N; i++){
    int swapped = 0;
    for (int j = 1; j <  n; j++)
            if (A[j] < A[j-1]) {
                swap(j-1 , j); swapped = 1;
             }//end if
     n = n-1;
    if (swapped == 0)
            break;              }//End outer for
```

Bubble Sort Example (Fifth Pass)

i = 4

10 (0)	44 (1)	6 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

j = 1

j = 2

j = 3

10 (0)	6 (1)	44 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

```
int n =N; // N is the size of the array;
 for (int i = 0; i < N; i++){
    int swapped = 0;
    for (int j = 1; j <  n; j++)
            if (A[j] < A[j-1]) {
                swap(j-1 , j); swapped = 1;
             }//end if
     n = n-1;
    if (swapped == 0)
            break;              }//End outer for
```

Bubble Sort Example (Sixth Pass)

i = 5

10 (0)	6 (1)	44 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

j = 1

j = 2

6 (0)	10 (1)	44 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

```
int n =N; // N is the size of the array;
 for (int i = 0; i < N; i++){
    int swapped = 0;
    for (int j = 1; j <  n; j++)
            if (A[j] < A[j-1]) {
                swap(j-1 , j); swapped = 1;
             }//end if
     n = n-1;
    if (swapped == 0)
            break;              }//End outer for
```

Bubble Sort Example (Seventh Pass)

***i* = 6**

6 (0)	10 (1)	44 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

***j* = 1**

6 (0)	10 (1)	44 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

Bubble Sort

```
int n =N; // N is the size of the array;
  for (int i = 0; i < N; i++){
    for (int j = 1; j <  n; j++) {
          if (A[j] < A[j-1]) {
              swap(j-1 , j);
           }//end if
      } //end inner for
} //end inner for
```

➔ Algorithm does not exit until all the data is checked

Complexity ?

$O(N^2)$

Bubble Sort

```
int n =N; // N is the size of the array;
  for (int i = 0; i < N; i++){
       int swapped = 0;
       for (int j = 1; j <  n; j++) {
            if (A[j] < A[j-1]) {
                 swap(j-1 , j);
                 swapped = 1;
             }//end if
       } //end inner for
       n = n-1;
       if (swapped == 0)
            break;
  } //end inner for
```

➔ Algorithm exits if no swap done in previous (outer loop) step

Complexity ?

$O(N^2)$

Bubble Sort (Best Case)

The **best case** scenario occurs when the input array is **already sorted in ascending order**.

In this case, Bubble Sort will still go through the array to check if any swaps are needed, but it will quickly determine that the list is already sorted.

Bubble Sort (Best Case)

Let's take an already sorted array:
[1, 2, 3, 4, 5]

Step-by-Step Execution
Pass 1:

Compare **1** and **2** → No swap
Compare **2** and **3** → No swap
Compare **3** and **4** → No swap
Compare **4** and **5** → No swap

Since no swaps were made in this pass, the algorithm can stop early, concluding that the array is already sorted.

Bubble Sort (Best Case)

Bubble Sort includes an **optimization** where if no swaps occur in a full pass, the algorithm **stops early** instead of continuing through unnecessary passes.

In the best case, only **one pass** is needed to verify that the list is sorted.

This requires exactly **n−1 comparisons** (checking each pair once).

Since the number of operations grows **linearly** with the size of n, the time complexity is **O(n)**

Bubble Sort (Worst Case)

The worst-case scenario occurs when the input array is in **reverse order** (completely unsorted in descending order).

This means that **every element needs to be swapped** in each pass to move to its correct position.

Bubble Sort (Worst Case)

For example, let's take the array:
[5, 4, 3, 2, 1] (completely reversed)

Pass 1:
Compare **5** and **4** → Swap → **[4, 5, 3, 2, 1]**
Compare **5** and **3** → Swap → **[4, 3, 5, 2, 1]**
Compare **5** and **2** → Swap → **[4, 3, 2, 5, 1]**
Compare **5** and **1** → Swap → **[4, 3, 2, 1, 5]**

Pass 2:
Compare **4** and **3** → Swap → **[3, 4, 2, 1, 5]**
Compare **4** and **2** → Swap → **[3, 2, 4, 1, 5]**
Compare **4** and **1** → Swap → **[3, 2, 1, 4, 5]**

Bubble Sort (Worst Case)

For example, let's take the array:
[5, 4, 3, 2, 1] (completely reversed)

Pass 3:
Compare **3** and **2** → Swap → **[2, 3, 1, 4, 5]**
Compare **3** and **1** → Swap → **[2, 1, 3, 4, 5]**

Pass 4:
Compare **2** and **1** → Swap → **[1, 2, 3, 4, 5]**
(Sorted!)

In the worst case, Bubble Sort **performs the maximum number of swaps** and comparisons.

Bubble Sort (Worst Case)

In the first pass, **(n - 1)** comparisons are made.

In the second pass, **(n - 2)** comparisons are made.

This continues until the last pass with **1** comparison.

Bubble Sort (Worst Case)

$$(n-1)+(n-2)+(n-3)+\cdots+2+1 = \frac{n(n-1)}{2}$$

The highest order term in $\frac{n(n-1)}{2}$ is **n2**, so we write the time complexity as **O(n2)**.

Bubble Sort (Average Case)

Time Complexity?

Questions?

zahmaad.github.io