CS2009
Design and Analysis of
Algorithm 5

Task

1) $2n^2 = \Omega(n^3)$ *True or False?*

2) If f(n) = O(g(n)) and g(n) = O(f(n)) then f(n) = g(n) True or false?

3) If f(n) = O(g(n)) and g(n) = O(h(n)), then h(n) = Ω(f(n)) True or false?

4) n/100= Ω(n) *True or False?*

5) Let $f(n) = \sum_{i=1}^{n} i$ and $g(n) = n^2$. $f(n) = \Theta(g(n))$.

Solution

1-false

2-false n n+1

3-true a<b; b<c ; a<C **Transitive relation**

4-true c>(1/100)

5-true

Sorting techniques and their analysis (I): n^2:

- Bubble sort
- Insertion sort
- Selection sort

Introduction

- 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, …. r_n with key values k_1, k_1, …. k_n, respectively we must produce the same records in an order $r^i{}_1$, $r^i{}_2$, …. $r^i{}_n$ such that the keys are in the corresponding non-decreasing order.

$\text{key}(r^i_1) \leq \text{key}(r^i_2) \leq \text{key}(r^i_3) \leq \text{key}(r^i_4) \leq \text{key}(r^i_5) \leq \text{key}(r^i_6)$

$\rightarrow k^i_1 \leq k^i_2 \leq k^i_3 \leq k^i_4 \leq k^i_5 \leq k^i_6$

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

- **Different criteria to evaluate the running time, as follows:**
    1. Number of algorithm steps.
    2. Number of comparisons between the keys (for expensive comparisons).
    3. The number of times a record is moved (for large records).

Bubble Sort

Bubble Sort

- One of the simplest sorting methods.
- The basic idea is the “weight” of the record.
- The records are kept in an array.

- “heavy” records bubbling up to the end.

- We make repeated passes over the array and sort the values •

If two adjacent elements are out of order we reverse the order.

Bubble Sort

- The overall effect, is that after the first pass the “heavy” record will bubble all the way to the end.

- On the second top pass, the second highest value goes to the second position, and so on.

- Reduce 1 element in each iteration

- The bubble sort algorithm is a reliable sorting algorithm.
- This algorithm has a worst-case time complexity of O(n2).
- The bubble sort has a space complexity of O(1).
- The number of swaps in bubble sort equals the number of inversion pairs in the given array.
- When the array elements are few and the array is nearly sorted, bubble sort is effective and efficient.

Bubble Sort

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


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 outer for
```

Complexity ?

$O(N^2)$

```
// 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

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;
```

```
if (swapped == 0)  break;
} //end inner for
```

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

```
// 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;
}

}//end if
} //end inner for  n = n-1;
```

→

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

Complexity ?

$O(N^2)$

Bubble Sort Example (First Pass)

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

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

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

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

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

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

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

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

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])

swap(*A*[5] , *A*[6])

break; }//End outer for

Bubble Sort Example (Second Pass)

i = 1

j = 1

j = 2

j = 3

j = 4

j = 5

j = 6

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

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

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)

$i = 2$

$j = 5$

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

$j = 1$

$j = 2$

$j = 3$

$j = 4$

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

j = 1

j = 2

j = 3

j = 4

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

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]) {
 swapped(j-1 , j); swapped = 1;
}//end if
n = n-1;
if (swapped == 0)
break; }//End outer for
```

Bubble Sort Example (Fifth Pass)

```
swap(j-1 , j); swapped = 1;
}//end if
n = n-1;
if (swapped == 0)
break; }//End outer for
```

i = 4

j = 1

j = 2

j = 3

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

Bubble Sort Example (Sixth Pass)

i = 5

j = 1

j = 2

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

***j* = 1**

6 (0) 10 (1) 44 (2) 45 (3) 55 (4) 58 (5) 62 (6) 90 (7) 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]) {
 swapped(j-1 , j); swapped = 1;
 }//end if
 n = n-1;
 if (swapped == 0)
 break; }//End outer for
```


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

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
```

Complexity ?

$O(N^2)$

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


```
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; // Optimization: reduce the number of
comparisons
```

if (swapped == 0)
break; // No swaps means the array is sorted } //
end outer for

1.It is the simplest sorting approach
2.Best case complexity is of O(N) [for optimized approach] while the array is sorted.
3.Using optimized approach, it can detect already sorted array in first pass with time complexity of O(N).
4.Stable sort: does not change the relative order of elements with equal keys.
5.In-Place sort.

Time Complexity:

- Best Case Sorted array as input. Or almost all elements are in proper place. [O(N)]. O(1) swaps.
- Worst Case: Reversely sorted / Very few elements are in

proper place. [$O(N^2)$] . $O(N^2)$ swaps.

- Average Case: [$O(N^2)$] . $O(N^2)$ swaps.
- <u>Space Complexity</u>: A temporary variable is used in swapping [a~~uxiliary, O(1)]. He~~nce it is In-Place sort.

Simplest Sorting Approach:

Bubble Sort is indeed one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Best Case Complexity of ��(��):

For the optimized version of Bubble Sort (which uses a swapped flag), the best case occurs when the input array is already sorted. In this case, only one pass is required through the array, resulting in a time complexity of ��(��).

Detect Already Sorted Array in First Pass with ��(��) Complexity: The optimized Bubble Sort can detect a sorted array in the first pass itself if no swaps are needed, leading to a time complexity of ��(��).

- **Stable Sort:**

  Bubble Sort is a stable sorting algorithm because it does not change the relative order of elements with equal keys. When two elements are equal, no swapping is performed, maintaining their relative order.

- **In-Place Sort:**

  Bubble Sort is an in-place sorting algorithm, meaning it requires only a constant amount ��(1) of additional memory space for sorting.

- **Time Complexity:**
  - ❖ Best Case: ��(��) for an already sorted array, where there are ��(1)swaps.
  - ❖ Worst Case: ��(��^{2}) for a reversely sorted array or when very few elements are in their proper places. The number of swaps in the worst case is also ��(��^{2}).

❖ Average Case: The time complexity in the average case is also ��(��^{2}), with ��(��^{2})swaps.

▪ **Space Complexity:**

The space complexity of Bubble Sort is ��(1) since it only requires a single temporary variable for swapping, making it an in-place sort.

Insertion Sort

Insertion Sort

- On the *i*th pass we "insert" the *i*th element *A*[i] into its rightful place among *A*[1],*A*[2],...*A*[i-1] which were placed in sorted order.

- After this insertion *A*[1],*A*[2],...*A*[i] are in sorted order. •

Insertion Sort

```
for (int i =1, i < n, i++){
 temp = A[i];
        for (int j = i, j>0 && A[j-1] > temp, j--)
        A[j] = A[j-1];
              A[j] = temp;
 }// end outer for
```

Complexity ?

$O(N^2)$


```
for (int i = 1; i < n; i++) {
 int temp = A[i];
 int j = i - 1;
 // Shift elements of A[0..i-1], that are greater than temp, to one position ahead
```

```
while (j >= 0 && A[j] > temp) {
 A[j + 1] = A[j];
 j--;
 }
 A[j + 1] = temp;
} // end outer for
```


Insertion Sort

INSERTION-SORT(A)		cost	times
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4	$i = j - 1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

Insertion Sort Example (First Pass)

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

***i* = 1**

***j* = 1** 62 (0) 62 (1) 55 (2) 10 (3) 45 (4) 44 (5) 6 (6) 90 (7) → $j > 0$ && A [j -1] > *temp*
90 (7) 62 > 58

***A*[1] = *A*[0]**

***j* = 0** 58 (0) 62 (1) 55 (2) 10 (3) 45 (4) 44 (5) 6 (6)

}// end outer for

A[0] = temp

= 58;

→ $A[1] = A[0]$

→ $j = 0$ → **exit**

$A[j]$ = **temp (= 58)**

for (int i =1, $i < n$, i++){

temp = A[i];

for (int j = i, $j > 0$ && $A[j\text{-}1] > temp$,

j--) $A[j] = A[j\text{-}1]$

$A[j]$ = temp;

Insertion Sort Example (Second Pass)

(7) **58 > 55**

***temp* = 55 *A*[2] = *A*[1] *A*[1] = *A*[0]**

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

***j* = 2** 58 (0) 62 (1) 62 (2) 10 (3) 45 (4) 44 (5) 6 (6) 90

$j = 1$

58 (0)	58 (1)	62 (2)	10

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

$A[0] = temp = 55;$

```
for (int i =1, i < n, i++){
temp = A[i];
        for (int j = i, j > 0 && A[j-1] > temp, j--)
        A[j] = A[j-1]
A[j] = temp;
}// end outer for
```


Insertion Sort Example (Third Pass)

(6) 90 (7)

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

***temp* = 10 *A*[3] = *A*[2] *A*[2] = *A*[1]**

= 3 ***j* = 3** 55 (0) 58 (1) 62 (2) 62 (3) 45 (4) 44 (5) 6

***j* = 2**

55 (0)	58 (1)	58 (2)	62 (

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

90 (7) ***j* = 0** 10 (0) 55 (1) 58 (2) 62 (3) 45 (4) 44 ***A*[1] = *A*[0]**

A*[*0*] = *temp
= 55;

for (int i =1, $i < n$, i++){

temp = A[i];

for (int j = i, $j > 0$ && $A[j\text{-}1] > temp$, j--)

$A[j] = A[j\text{-}1]$

$A[j]$ = temp;

}// end outer for

Insertion Sort Example (Fourth Pass)

(6) 90 (7)

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

***temp* = 45 *A*[4] = *A*[3] *A*[3] = *A*[2]**

= 4 *j* = 4 10 (0) 55 (1) 58 (2) 62 (3) 62 (4) 44 (5) 6

***j* = 3**

10 (0)	55 (1)	58 (2)	58 (

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

90 (7) <u>10 > 45</u>

***A*[2] = *A*[1]**

***j* = 1** 10 (0) 45 (1) 55 (2) 58 (3) 62 (4) 44 (5) 6 (6)

A*[1] = *temp
= 45;

for (int i =1, $i < n$, i++){

temp = A[i];

for (int j = i, j > 0 && A[j-1] > $temp$, j--)

A[j] = A[j-1]

A[j] = temp;

33

}// end outer for

Insertion Sort Example (Fifth Pass)

(6) 90 (7)

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

***temp* = 44 *A*[5] = *A*[4] *A*[4] = *A*[3]**

j **= 4**

10 (0)	45 (1)	55 (2)	58

j **= 3** 10 (0) 45 (1) 55 (2) 55 (3) 58 (4) 62 (5) 6 (6) **A[3] = A[2]**
90 (7)

j **= 2** 10 (0) 45 (1) 45 (2) 55 (3) 58 (4) 62 (5) 6 (6) 90 (7) **A[2] = A[1]**

j **= 1** 10 (0) 44 (1) 45 (2) 55 (3) 58 (4) 62 (5) 6 (6) 90 (7) ***A*[1] = *temp***

= 44;

```
for (int i =1, i < n, i++){
temp = A[i];
        for (int j = i, j > 0 && A[j-1] > temp, j--)
         A[j] = A[j-1]
A[j] = temp;
}// end outer for
```


Insertion Sort Example (Sixth Pass)

90 (7)

10 (0) 44 (1) 45 (2) 55 (3) 58 (4) 62 (5) 6(6) 90 (7) ***i* = 6** ***j* = 6** ***temp* = 6 *A*[6] = *A*[5] *A*[5] = *A*[4]**

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

***j* = 5**

10 (0)	44 (1)	45 (2)	55

***j* = 4** 10 (0) 44 (1) 45 (2) 55 (3) 55 (4) 58 (5) 62(6) ***A*[4] = *A*[3]**

90 (7)

***j* = 3** 10 (0) 44 (1) 45 (2) 45 (3) 55 (4) 58 (5) 62(6) 90 (7) ***A*[3] = *A*[2]**

j = 2 *j* = 1

10 (0)	10 (1)	45 (2

A[2] = *A*[1] *A*[1] = *A*[0]

10 (0)	44 (1)	45 (2

j = 0 6 (0) 10 (1) 45 (2) 45 (3) 55 (4) 58 (5) 62(6) 90 (7) *A*[0] = *temp* = 6;

for (int i =1, $i < n$, i++){

temp = A[i];

for (int j = i, $j > 0$ && $A[j\text{-}1] > temp$,

j--) $A[j] = A[j\text{-}1]$

$A[j]$ = temp;

}// end outer for

Insertion Sort Example (Sixth Pass)

6(0) 10 (1) 45 (2) 45 (3) 55 (4) 58 (5) 62(6) 90 (7) *i* = 7 *temp* = 90

A[*j*-1] > *temp*?
No
Quit

for (int i =1, $i < n$, i++){

temp = A[i];

for (int j = i, $j > 0$ && $A[j-1] > temp$, j--)

$A[j] = A[j-1]$

$A[j]$ = temp;

}// end outer for

Insertion Sort Example (Sixth Pass)

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

***i* = 7**

***temp* = 90**

***A*[*j*-1] > *temp*?**

No

Quit

```
for (int i =1, i < n, i++){
temp = A[i];
    for (int j = i, j > 0 && A[j-1] > temp, j--)
    A[j] = A[j-1]
A[j] = temp;
}// end outer for
```


Analysis of Insertion Sort

- Because of the nested loops, each of which can take ***n*** iterations, insertion sort is $O(n^2)$.

- Furthermore, this bound is tight, because input in reverse order can actually achieve this bound.
- A precise calculation shows that the test at line 3 can be executed at most ***i*** times for each value of ***i***. Summing over all ***i*** gives a total of

$$\sum_{i=1}^{n-1} i = 1 + 2 + \ldots + (n-1) = \Theta(n^2)$$

- If the input is presorted, the running time is ***O*(*n*)** •

because the test in the inner *for* loop always fails immediately •

The average running time also $O(n^2)$

- Less number of swaps

1.It can be easily computed.

2.Best case complexity is of **O(N)** while the array is already sorted.

3.Number of swaps reduced than bubble sort. 4.For smaller values of N, insertion sort performs efficiently like other quadratic sorting algorithms.

5.Stable sort.

6.Adaptive: total number of steps is reduced for partially sorted array.

7.In-Place sort.

Selection Sort

Selection Sort

- Find the minimum value in the list
- Swap it with the value in the first position •Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Selection Sort

```
for (int i =0, i < n, i++){
min = i;
    for (int j = i+1, j < n , j++){
        if ( A[j] < A[min]){
        min = j
 } // end if
 }// end inner for
  swap(i, min)
}// end outer for
```

Complexity ?

$O(N^2)$

A[n] is a globally declared array

```
swap(i, min) {
int temp = A[i];
A[i] = A[min];
A[min] = temp;
}
```


// Swap function assumes that //

Selection Sort - Example

- The Selection Sort might swap an array element

[redacted] with itself--this is harmless.

2 7 8 5 4

}// end inner for
swap(i, min)
}// end outer for

2 4 8 5 7 2 4 5 8 7 2 4 5 7 8

for (int i =0, $i < n$, i++){
min = i;
for (int j = i+1, $j < n$, j++){

if ($A[j] < A$[min]){
min = j
} // end if

Selection Sort Example

i = 0 min = 6

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

A[min] = 6

A[min] = 62 A[min] = 58A[min] = 55A[min] = 10 A[min] = 10A[min] = 10A[min] = 6A[min] = 6

6 (0) 58 (1) 55 (2) 10 (3) 45 (4) 44 (5) 62 (6) 90 (7) i = 1 min = 3 A[min] = 10

A[min] = 58A[min] = 55A[min] = 10 A[min] = 10A[min] = 10A[min] = 10A[min] = 10

6 $^{(0)}$ 10 $^{(1)}$ 55 $^{(2)}$ 58 $^{(3)}$ 45 $^{(4)}$ 44 $^{(5)}$ 62 $^{(6)}$ 90 $^{(7)}$ ***i* = 2** **min = 5** ***A*[min] = 44**

A[min] = 55A[min] = 58 A[min] = 45A[min] = 44A[min] = 44A[min] = 44

6 $^{(0)}$ 10 $^{(1)}$ 44 $^{(2)}$ 58 $^{(3)}$ 45 $^{(4)}$ 55 $^{(5)}$ 62 $^{(6)}$ 90 $^{(7)}$ ***i* = 3** **min = 4** ***A*[min] = 45**

A[min] = 58 A[min] = 45A[min] = 45A[min] = 45A[min] = 45

```
for (int i =0, i < n, i++){
 min = i;
        for (int j = i+1, j < n ,j++){
              if ( A[j] < A[min]){
              min = j
 } // end if
 }// end inner for
        swap(i, min)

 }// end outer for
```


Selection Sort Example - continued ***i* = 0** **min = 6**

62 $^{(0)}$	58 $^{(1)}$	55 $^{(2)}$	10 $^{(3)}$	45

6 $^{(0)}$	58 $^{(1)}$	55 $^{(2)}$	10 $^{(3)}$	45 $^{(4)}$	44 $^{(5)}$
6 $^{(0)}$	10 $^{(1)}$	55 $^{(2)}$	58 $^{(3)}$	45 $^{(4)}$	44 $^{(5)}$

A[min] = 6

$i = 1$ min = 3

A[min] = 10

$i = 2$ min = 5

A[min] = 44

6 (0) 10 (1) 44 (2) 58 (3) 45 (4) 55 (5) 62 (6) 90 (7) $i = 3$ min = 4 A[min] = 45

6 (0) 10 (1) 44 (2) 45 (3) 58 (4) 55 (5) 62 (6) 90 (7) $i = 4$ min = 55 A[min] = 45

A[min] = 58A[min]= 55A[min] = 55A[min] = 55

6 (0) 10 (1) 44 (2) 45 (3) 55 (4) 58 (5) 62 (6) 90 (7) $i = 5$ min = 5 A[min] = 58

6 (0) 10 (1) 44 (2) 45 (3) 55 (4) 58 (5) 62 (6) 90 (7) $i = 6$ min = 6 A[min] = 62

6 (0) 10 (1) 44 (2) 45 (3) 55 (4) 58 (5) 62 (6) 90 (7) $i = 7$ min = 6 A[min] = 62

```
for (int i =0, i < n, i++){
 min = i;
          for (int j = i+1, j < n ,j++){
                    if ( A[j] < A[min]){
                    min = j
 } // end if
 }// end inner for
          swap(i, min)

 }// end outer for
```


1.It can also be used on list structures that make add and remove efficient, such as a linked list. Just remove the smallest element of unsorted part and end at the end of sorted part.

2.The number of swaps reduced. **O(N)** swaps in all cases.

3.In-Place sort.

4.Time complexity in all cases is **O(N^2)**, no best case scenario. Space Complexity: **O(1).** In-Place sort.(When elements are shifted instead ~~of being swapped~~ (i.e. temp=a[min], then shifting elements from ar[i] to ar[min-1] one place up and then putting a[i]=temp). If swapping is opted for, the algorithm is not In-place.)

Time Complexity:

•**Best Case** [**O(N^2)**]. And **O(1)** swaps.

•**Worst Case**: Reversely sorted, and when the inner loop makes a maximum comparison. [**O(N^2)**] . Also, **O(N)** swaps.

•**Average Case**: [**O(N^2)**] . Also **O(N)** swaps.

Bubble sort performs sorting by checking the neighboring data elements and swapping them if they are in wrong order

Insertion sort performs sorting by transferring one element to a partially sorted array at a time

Selection Sort vs Insertion Sort vs Bubble sort

- Selection sort's advantage is that

- While insertion sort typically makes fewer comparisons than selection sort,
- In bubble sort more swaps then insertion sort.
- Bubble sort is slower then insertion sort.
- But bubble is simple while insertin is complex.
- Insertion sort requires more writes than the selection sort because the inner loop of the insertion sort can require shifting large sections of the sorted portion of the array.
  - In general, insertion sort will write to the array O(n^2) times
  - Whereas selection sort will write/swap only O(n) times
- For this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading,
  - such as with EPROM or flash memory

Comparisons of different sorting

algorithms

$\Theta(n^2)$ comparisons	$\Theta(n^2)$ comparisons	$\Theta(n^2)$ comparisons
$\Theta(n^2)$ swaps	$\Theta(n^2)$ writes	$\Theta(n)$ swaps
Adaptive: O(n) running time when nearly sorted (Best case running time)	Adaptive: O(n) running time when nearly sorted (Best case running time)	Not adaptive $\Theta(n^2)$ running time when nearly sorted (Best case running time)

https://www.toptal.com/developers/sorting-algorithms

Thank you