

# Sorting Algorithms (Part I)

# Outline

- Definition of sorting algorithms
- Attributes of sorting algorithms
  - Stability
  - Loop invariant
  - In-place vs out-of-place
  - Adaptability
- Sorting algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
  - Quick sort

# Sorting Algorithm

- **Input:** A sequence of  $n$  numbers  $\{a_1, a_2, \dots, a_n\}$
- **Output:** A permutation (reordering)  $\{a'_1, a'_2, \dots, a'_n\}$  of the input sequence such that
  - $a'_1 \leq a'_2 \leq \dots \leq a'_n$  (increasing order) or
  - $a'_1 \geq a'_2 \geq \dots \geq a'_n$  (decreasing order)
- For example,
  - Input:  $\{31, 41, 59, 26, 41, 58\}$
  - Output:  $\{26, 31, 41, 41, 58, 59\}$  (increasing order)

# Applications of Sorting Algorithms

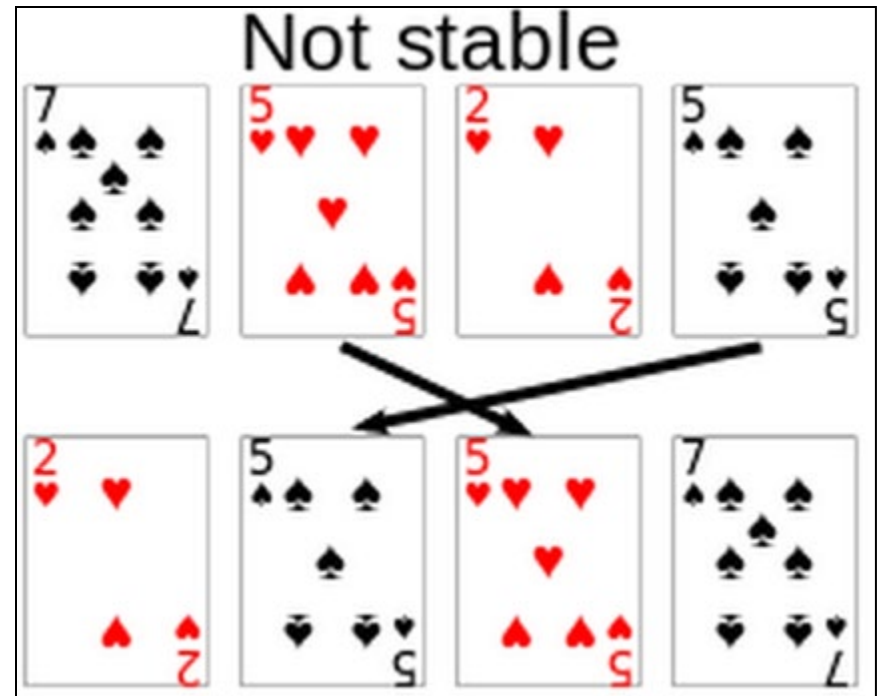
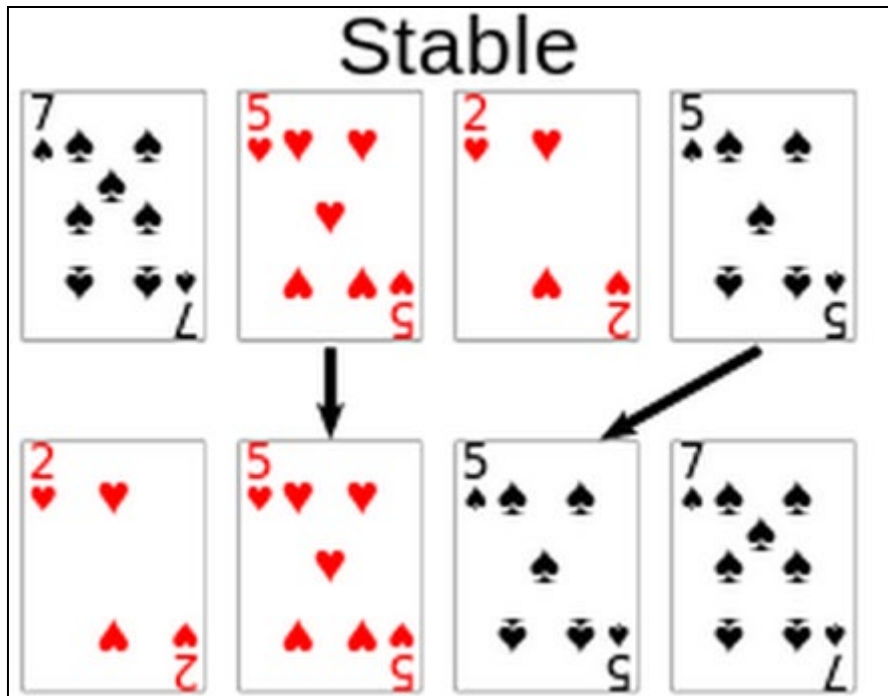
- Databases
- Operating Systems
- Search Engine
- E-Commerce Recommendation System

# Attributes of Sorting Algorithms

- Stability
- Loop invariant
- In-place and Out-of-place
- Adaptability

# Stability

- A sorting algorithm is said to be **stable** if it preserves the relative order of items with equal values.



# Stability

- This stability attribute is crucial in some scenarios, e.g., sorting based on multiple criteria.
- Example: Sort by age first (in ascending order) and then by name (maintain their original order based on their names).

Original Data	
Name	Age
Alice	25
Bob	30
Charlie	25
David	22
Eva	30

Sorted Data (Stable)	
Name	Age
David	22
Alice	25
Charlie	25
Bob	30
Eva	30

Sorted Data (Unstable)	
Name	Age
David	22
Charlie	25
Alice	25
Eva	30
Bob	30

# Loop Invariant

- A loop invariant is a condition that is true
  - before the loop starts,
  - at the end of each iteration,
  - after the loop's termination.



# Loop Invariant

```
// find the maximum value of an n-element array a
1.int max(const int a[], int n) {
2.    int m = a[0];
3.    int i = 1;
4.    // m equals the maximum value in a[0],...,a[i-1], i=1 → Before the loop
5.    while (i != n) {
6.        if (a[i]>m)
7.            m = a[i];
8.        ++i;
9.        // m equals the maximum value in a[0],...,a[i-1], → At the end of
10.           each iteration
11.    }
12.    // m equals the maximum value in a[0],...,a[i-1], and i=n → After the loop
13.    return m;
14.}
```

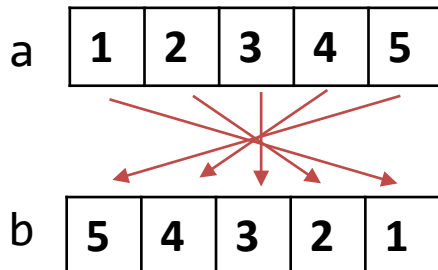
# In-Place and Out-of-Place

- An **in-place** algorithm is an algorithm which transforms input without using any extra memory or with only a small and constant amount of extra memory.
  - Extra memory space, if any, does not depend on the input size.
- An algorithm that is not in-place is called **out-of-place** algorithm.
  - Extra memory space depends on the input size.

# Example: Reverse an array

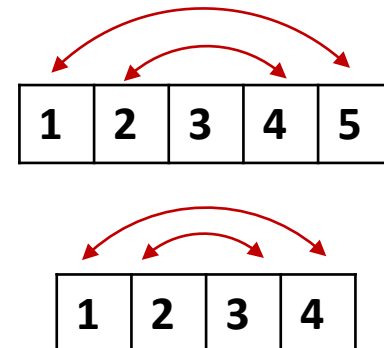
## Out-of-place

```
void f1(int a[],int n){  
    int* b = new int[n];  
    int i;  
    for(i=0;i<n;++i)  
        b[n-i-1] = a[i];  
    for(i=0;i<n;++i)  
        a[i]=b[i];  
    delete [] b;  
}
```



## In-place

```
void f2(int a[], int n){  
    int i;  
    int m=n/2;  
    for(i=0;i<m;++i)  
        Swap(a[i],a[n-i-1]);  
}
```



# In-Place and Out-of-Place

- In-place sorting algorithms are more memory efficient.
- They are often preferred when memory is a constraint, e.g., for large dataset.

# Adaptability

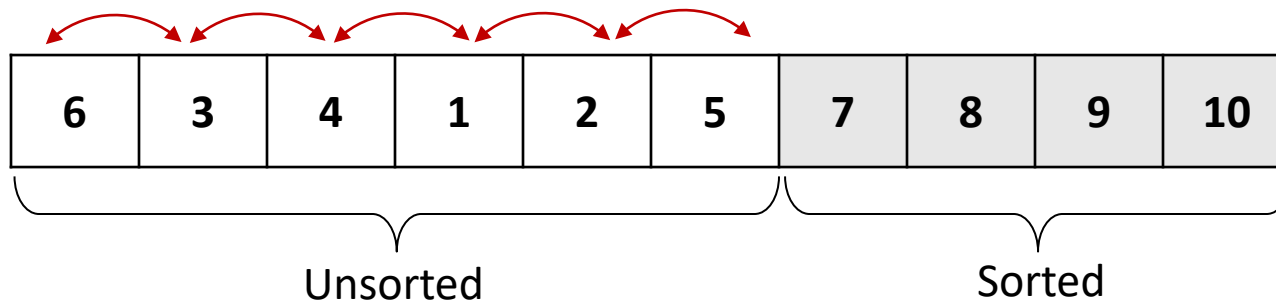
- An adaptive sorting algorithm takes advantage of existing order in its input.
- It becomes more efficient when dealing with partially sorted data.
- Adaptability is an attractive feature for a sorting algorithm because nearly sorted sequences are common in practice.

# Sorting Algorithms

- Bubble sort
- Selection sort
- Insertion sort

# Bubble Sort

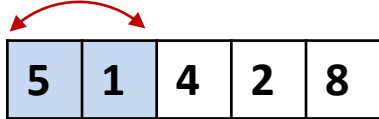
- Main idea: in each pass
  - The array is divided into unsorted and sorted parts.
  - It repeatedly swaps adjacent elements in the unsorted part if they are out of order.
  - The largest element will "bubble up" to its correct position at the end of the unsorted part.



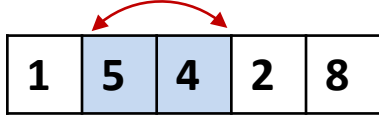
# Example: Bubble Sort

## 1<sup>st</sup> pass

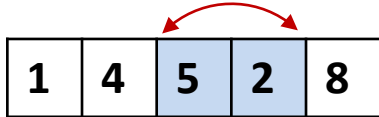
Swapping



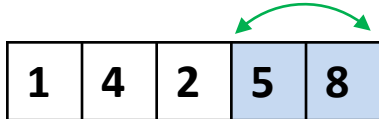
Swapping



Swapping



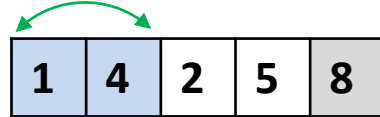
No Swapping



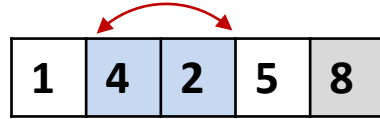
The largest element will be placed at the last position.

## 2<sup>nd</sup> pass

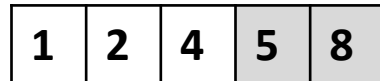
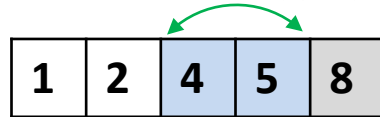
No Swapping



Swapping



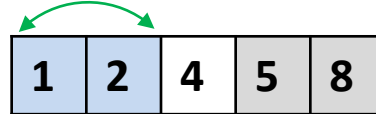
No Swapping



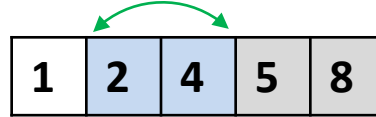
The second largest element will be placed at the second last position.

## 3<sup>rd</sup> pass

No Swapping



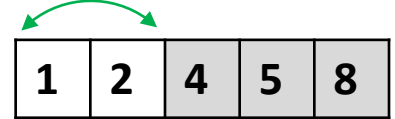
No Swapping



The third largest element will be placed at the third last position.

## 4<sup>th</sup> pass

No Swapping



All are placed at their correct position.



# Bubble Sort

```
void BubbleSort(int a[], int N){
    for (int i = 0; i < N - 1; ++i) //pass
        for (int j = 0; j < N - i - 1; ++j)
            if (a[j] > a[j + 1])
                Swap(a[j], a[j + 1]);
}

void Swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
```

## Attributes

- Stable:
- Loop invariant:
- In-place:
- Adaptive:

1	2	4	5	8
---	---	---	---	---

**Question:** How can we improve the algorithm when the array is already sorted?

# Bubble Sort

Attributes	Yes/No	Explanation
Stable	Yes	Elements of equal value are not swapped. The original order will be retained.
Loop invariant	Yes	The loop invariant condition is that at the end of $i$ -th pass, the right most $i$ elements are sorted and in their correct place.
In-place	Yes	Sorting operations are performed directly on the input array. Extra memory space required is constant and does not depend on the input size.
Adaptive	No	The complexity is the same (same number of passes and comparisons) for sorted arrays.

# Adaptive Bubble Sort

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

## Time Complexity:

- Best case:

1	2	4	5	8
---	---	---	---	---

- Worst case:

8	5	4	2	1
---	---	---	---	---

The algorithm stops when there is no swap.

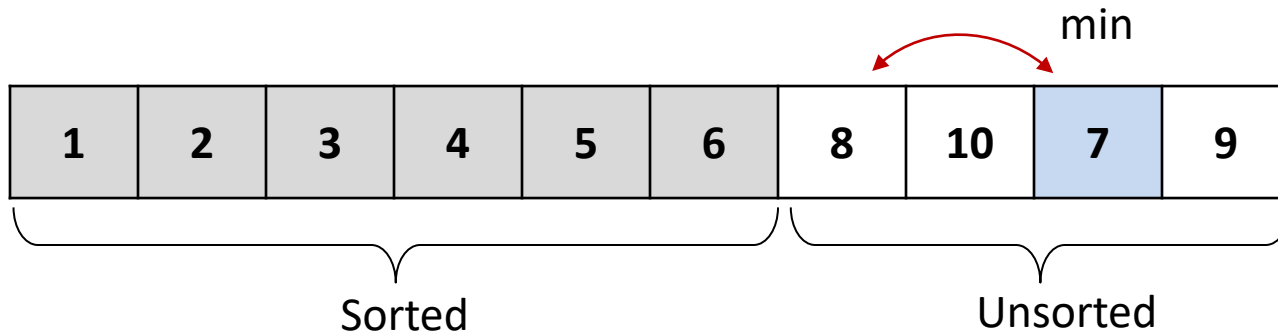
# Adaptive Bubble Sort

Complexity	Yes/No	Explanation												
Best case	$O(N)$	When the array is already sorted. Only go through 1 pass with $N - 1$ comparisons.												
Worst case	$O(N^2)$	<ul style="list-style-type: none"> <li>When the array is in reverse order.</li> <li>Every comparison will lead to a swap.</li> <li>Number of comparisons and swaps: <table border="1"> <thead> <tr> <th><math>i</math></th><th><math>j</math></th><th>No. Comparisons &amp; Swaps</th></tr> </thead> <tbody> <tr> <td>0</td><td><math>0, \dots, N - 2</math></td><td><math>N - 1</math></td></tr> <tr> <td>1</td><td><math>0, \dots, N - 3</math></td><td><math>N - 2</math></td></tr> <tr> <td><math>N - 2</math></td><td>0</td><td>1</td></tr> </tbody> </table> <ul style="list-style-type: none"> <li>Total: <math>(N - 1) + (N - 2) + \dots + 2 + 1 = \frac{N(N-1)}{2}</math></li> <li>Complexity: <math>O\left(\frac{N(N-1)}{2}\right) = O\left(\frac{N^2}{2} - \frac{N}{2}\right) = O(N^2)</math></li> </ul> </li> </ul>	$i$	$j$	No. Comparisons & Swaps	0	$0, \dots, N - 2$	$N - 1$	1	$0, \dots, N - 3$	$N - 2$	$N - 2$	0	1
$i$	$j$	No. Comparisons & Swaps												
0	$0, \dots, N - 2$	$N - 1$												
1	$0, \dots, N - 3$	$N - 2$												
$N - 2$	0	1												

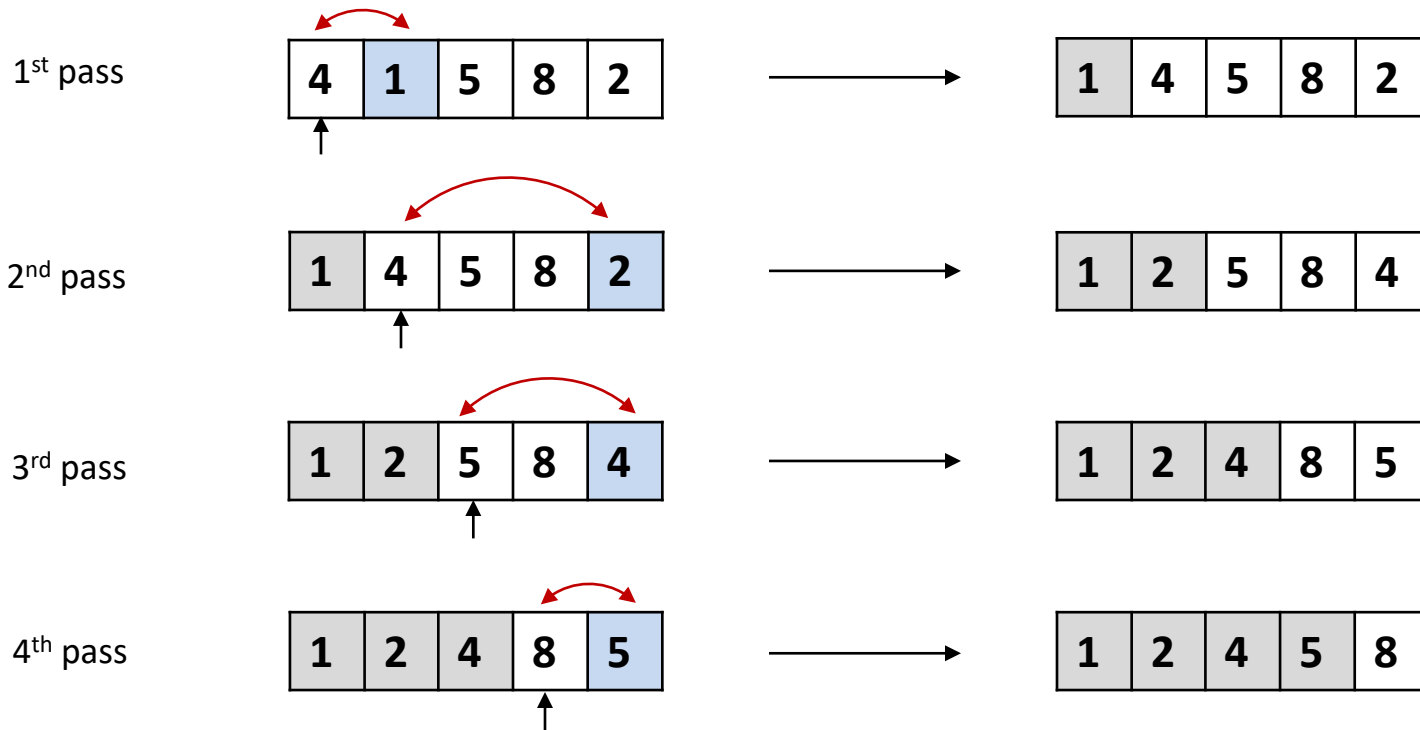
# Selection Sort

- Main idea: Increase the sorted sequence by repeatedly selecting the smallest element from the unsorted.
- 1. Select the smallest element from the unsorted side.
- 2. Append it at the end of the sorted side (or the first position of the unsorted).

Find the Smallest, then the second smallest from the rest...



# Example: Selection Sort




# Selection Sort

```
void SelectionSort(int a[], int N){  
    for (int i = 0; i < N-1; ++i){  
        int min = i;  
  
        for (int j = i + 1; j < N; ++j)  
            if (a[j] < a[min])  
                min = j;  
  
        Swap(a[min], a[i]);  
    }  
}
```

## Attributes

- Stable:
- Loop invariant:
- In-place:
- Adaptive:

# Selection Sort

Attributes	Yes/No	Explanation
Stable	No	The relative order of equal elements may change. For example:  <b>3</b> , 2, 3, <b>1</b> becomes <b>1</b> , 2, 3, <b>3</b> after 1 <sup>st</sup> pass.
Loop invariant	Yes	The loop invariant condition is that at the end of $i$ -th pass, the left most $i$ elements are sorted and in their correct position.
In-place	Yes	Sorting operations are performed directly on the input array. Extra memory space required is constant and does not depend on the input size.
Adaptive	No	The complexity is the same (same number of passes and comparisons) for sorted arrays.



# Selection Sort

```
void SelectionSort(int a[], int
N){
    for (int i = 0; i < N-1; ++i){
        int min = i;
        int j;

        for (j = i + 1; j < N; ++j)
            if (a[j] < a[min])
                min = j;

        Swap(a[min], a[i]);
    }
}
```

## Time Complexity:

- Best case:
- Worst case:

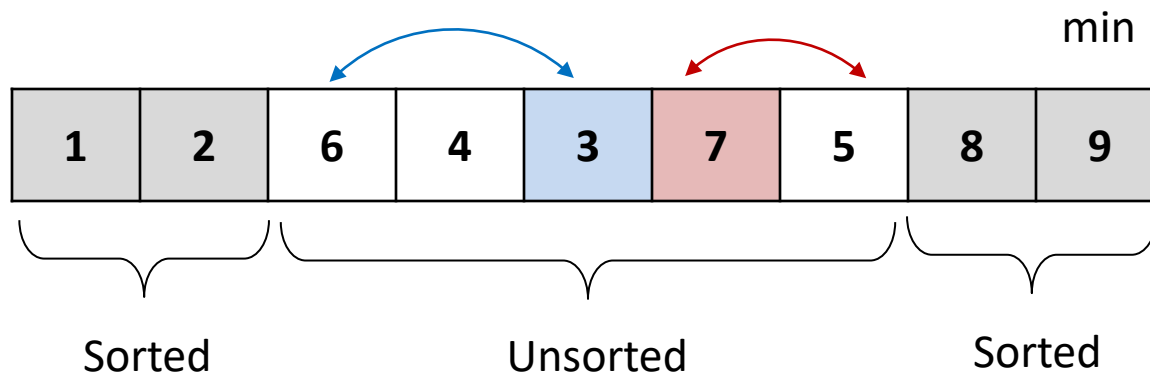
**Question:** How could you modify this algorithm to sort the array in half the number of passes?

# Selection Sort

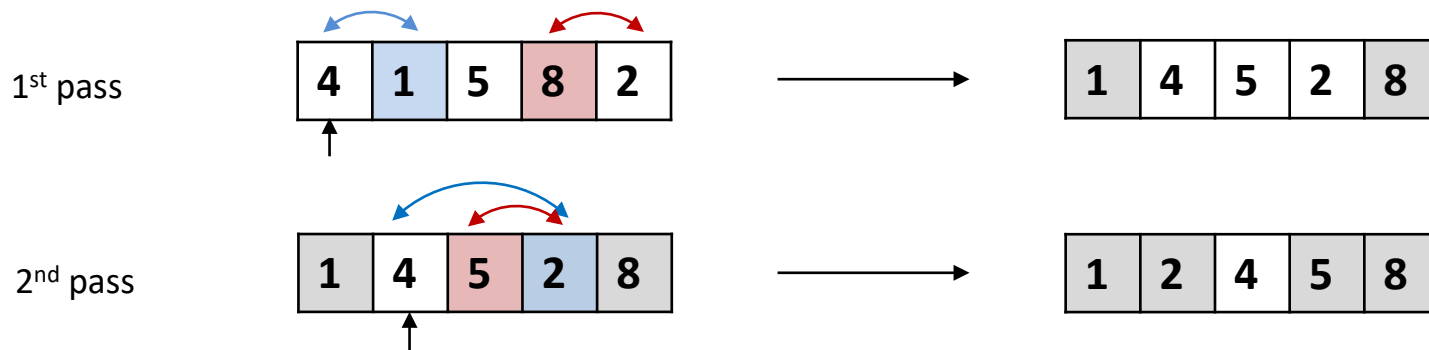
Complexity	Yes/No	Explanation												
Best case	$O(N^2)$	Selection sort performs the same number of comparisons regardless of the order of the input. The best-case time complexity is the same as the worst case.												
Worst case	$O(N^2)$	<ul style="list-style-type: none"> <li>Number of comparisons: <table border="1"> <thead> <tr> <th><math>i</math></th><th><math>j</math></th><th>No. Comparisons</th></tr> </thead> <tbody> <tr> <td>0</td><td><math>1, \dots, N - 1</math></td><td><math>N - 1</math></td></tr> <tr> <td>1</td><td><math>2, \dots, N - 1</math></td><td><math>N - 2</math></td></tr> <tr> <td><math>N - 2</math></td><td><math>N - 1</math></td><td>1</td></tr> </tbody> </table> <ul style="list-style-type: none"> <li>Total: <math>(N - 1) + (N - 2) + \dots + 2 + 1 = \frac{N(N-1)}{2}</math></li> <li>Complexity: <math>O\left(\frac{N(N-1)}{2}\right) = O\left(\frac{N^2}{2} - \frac{N}{2}\right) = O(N^2)</math></li> </ul> </li> </ul>	$i$	$j$	No. Comparisons	0	$1, \dots, N - 1$	$N - 1$	1	$2, \dots, N - 1$	$N - 2$	$N - 2$	$N - 1$	1
$i$	$j$	No. Comparisons												
0	$1, \dots, N - 1$	$N - 1$												
1	$2, \dots, N - 1$	$N - 2$												
$N - 2$	$N - 1$	1												

# Bidirectional Selection Sort

- Find both the minimum and maximum in each pass and place them at their correct position.

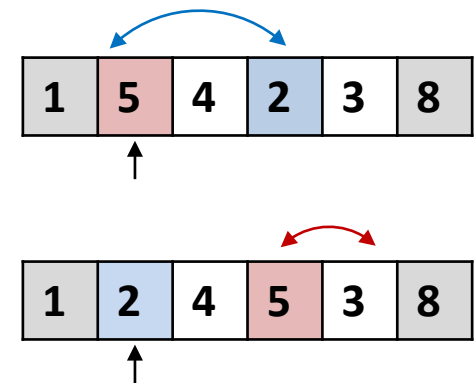


# Example: Bidirectional Selection Sort

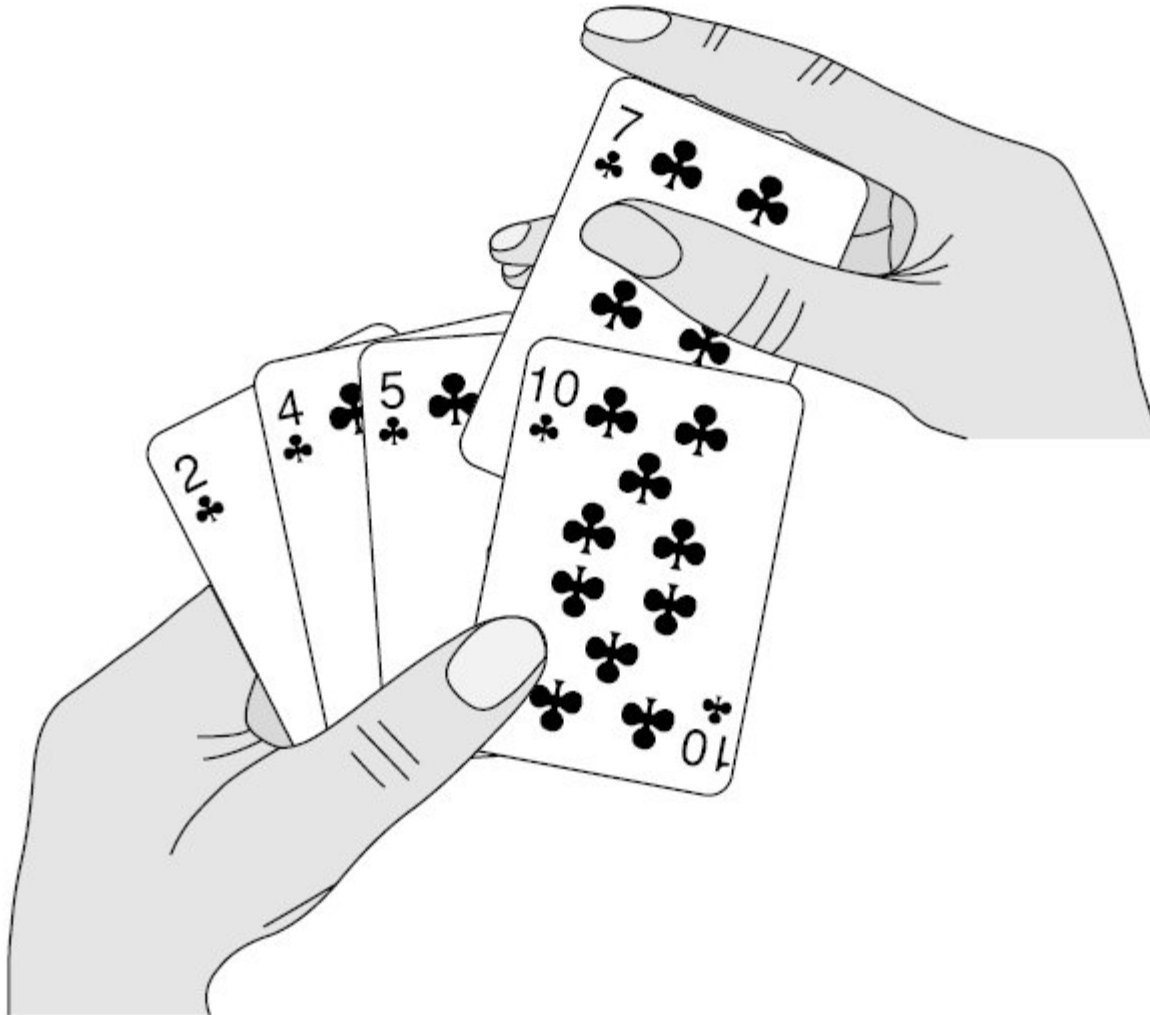


# Bidirectional Selection Sort

```
void BiSelectionSort(int a[], int N){  
    for (int i = 0; i < N/2; ++i){  
        int min = i;  
        int max = N-i-1;  
        int j;  
        for (j = i; j < N-i; ++j){  
            if (a[j] < a[min])  
                min = j;  
            if (a[j] > a[max])  
                max = j;  
        }  
        Swap(a[min], a[i]);  
        if (i==max)  
            Swap(a[min], a[N-i-1]);  
        else  
            Swap(a[max], a[N-i-1]);  
    }  
}
```

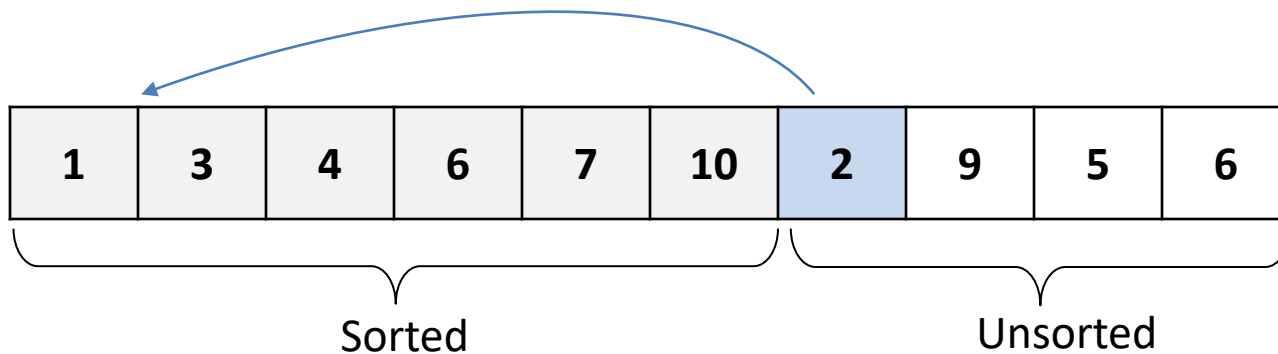


# Insertion Sort

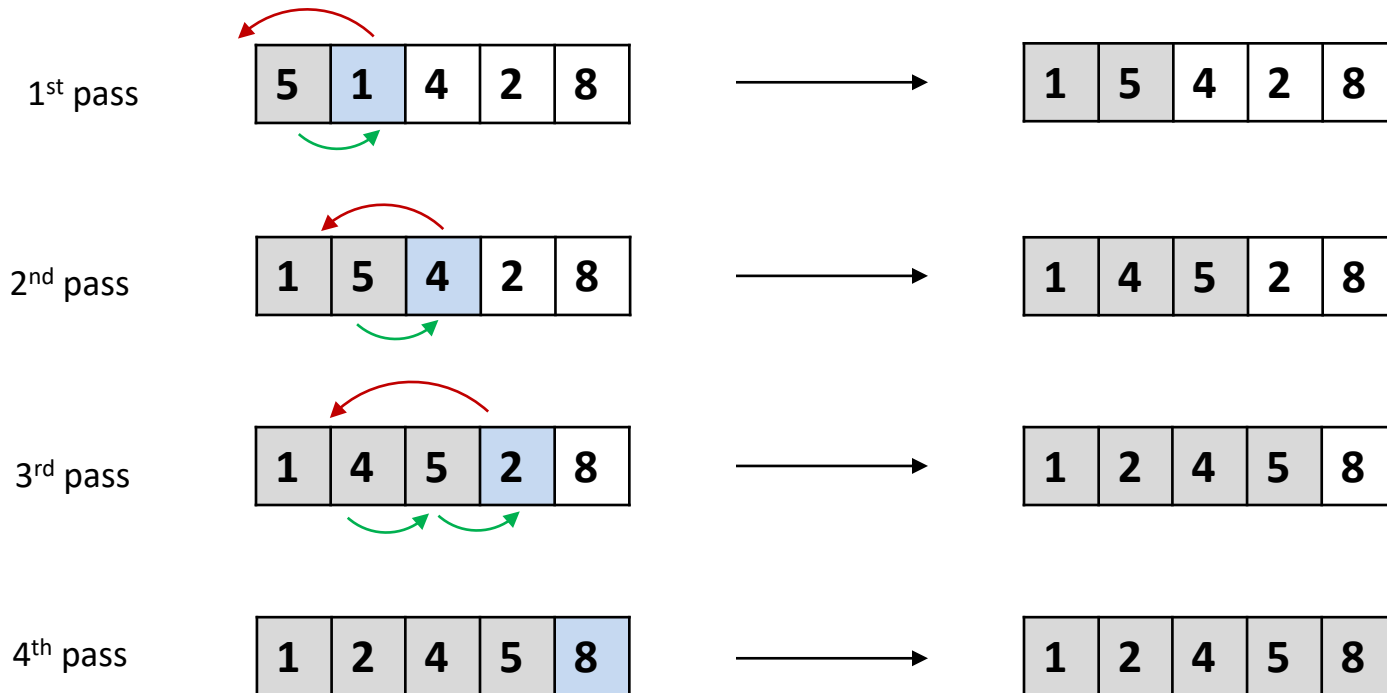


# Insertion Sort

- Main idea: Keep inserting the next element into the sorted sequence.
  1. Get the 1<sup>st</sup> element from the unsorted side
  2. Find the correct position in the sorted side
  3. Shift elements to make room for it.



# Example: Insertion Sort





# Insertion Sort

```
void InsertionSort(int a[], int N){
    for (int i = 1; i < N; ++i){
        int j = i;
        int current = a[i]; // item to be inserted

        while ((j > 0) && (a[j-1] > current)){
            a[j] = a[j-1];
            --j;
        } // find the position for insertion: j

        a[j] = current; // insert the item
    }
}
```

## Attributes

- Stable:
- Loop invariant:
- In-place:
- Adaptive:

1	2	3	4	5
---	---	---	---	---

# Insertion Sort

Attributes	Yes/No	Explanation
Stable	Yes	Elements of equal value are not swapped. The original order will be retained. For example: Initial: 3 1 2 4 <b>3</b> After 4 passes: 1 2 3 4 <b>3</b> <b>3</b> will be inserted after 3.
Loop invariant	Yes	The loop invariant condition is that at the end of $i$ -th pass, the left most $i$ elements are sorted.
In-place	Yes	Sorting operations are performed directly on the input array. Extra memory space required is constant and not dependent on the input size.
Adaptive	Yes	When the array is already sorted, the inner loop will not execute in each pass, and thus avoiding unnecessary comparisons and swaps.

# Insertion Sort

```
void InsertionSort(int a[], int N){
    for (int i = 1; i < N; ++i){
        int j = i;
        int current = a[i]; // item to be
                             inserted

        while ((j > 0) && (a[j-1] > current)){
            a[j] = a[j-1];
            --j;
        } // find the position for insertion: j

        a[j] = current; // insert the item
    }
}
```

## Time Complexity:

- Best case:

1	2	3	4	5
---	---	---	---	---

- Worst case:

5	4	3	2	1
---	---	---	---	---

# Insertion Sort

Complexity	Yes/No	Explanation												
Best case	$O(N)$	When the array is already sorted, the inner loop will not execute in each pass. The outer loop will execute $N - 1$ times.												
Worst case	$O(N^2)$	<p>When the array is in reverse order, the inner loop traverses the entire sorted portion.</p> <ul style="list-style-type: none"> <li>Number of comparisons and swaps: <table border="1"> <thead> <tr> <th><math>i</math></th><th><math>j</math></th><th>No. Comparisons and swaps</th></tr> </thead> <tbody> <tr> <td>1</td><td>1</td><td>1</td></tr> <tr> <td>2</td><td>1, 2</td><td>2</td></tr> <tr> <td><math>N - 1</math></td><td><math>1, \dots, N - 1</math></td><td><math>N - 1</math></td></tr> </tbody> </table> <ul style="list-style-type: none"> <li>Total: <math>1 + 2 + \dots + (N - 1) = \frac{N(N-1)}{2}</math></li> <li>Complexity: <math>O\left(\frac{N(N-1)}{2}\right) = O\left(\frac{N^2}{2} - \frac{N}{2}\right) = O(N^2)</math></li> </ul> </li> </ul>	$i$	$j$	No. Comparisons and swaps	1	1	1	2	1, 2	2	$N - 1$	$1, \dots, N - 1$	$N - 1$
$i$	$j$	No. Comparisons and swaps												
1	1	1												
2	1, 2	2												
$N - 1$	$1, \dots, N - 1$	$N - 1$												

# Comparison

Sorting Algorithm	Stable	Loop Invariant	In-place	Adaptive	Best-case Complexity	Worst-case Complexity
Bubble sort	Yes	Yes	Yes	No	$O(N^2)$	$O(N^2)$
Adaptive bubble sort	Yes	Yes	Yes	Yes	$O(N)$	$O(N^2)$
Selection sort	No	Yes	Yes	No	$O(N^2)$	$O(N^2)$
Insertion sort	Yes	Yes	Yes	Yes	$O(N)$	$O(N^2)$

# Summary

- Definition of sorting algorithms
- Attributes of sorting algorithms
  - Stability
  - Loop invariant
  - In-place vs out-of-place
  - Adaptability
- Sorting Algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort