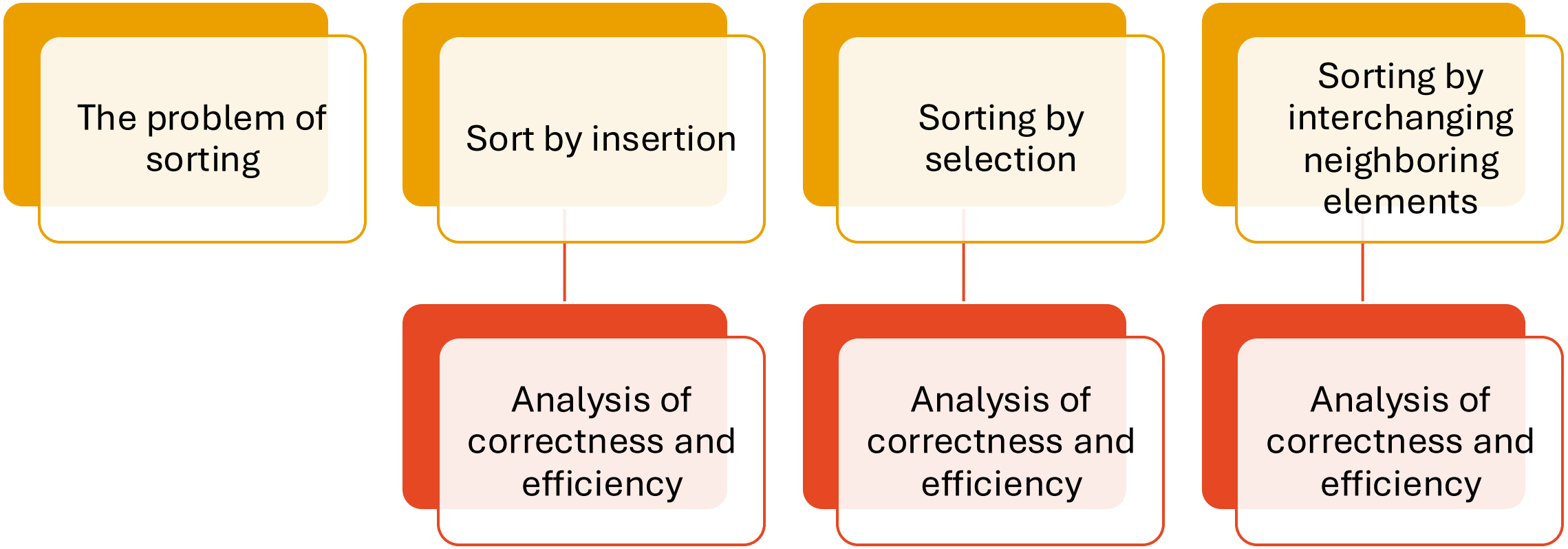
# ALGORITHMS AND DATA STRUCTURES I

*Course 4*

# Previous Course

- Analysis of algorithms
- Basic notations
- Steps in algorithm correctness checking
- Rules in algorithm correctness checking
- Examples

# Today course



# The problem of sorting

Consider:

- A sequence of "entities" (numbers, structures containing more information, etc.)
- Each entity possesses a characteristic called a sort criteria (key). The possible values of the sort key belong to a set on which there is a total order relationship
- Sequence sorting = arranging items so that sort key values are in ascending (or descending) order

# The problem of sorting

# Examples

1. Number sequence
   - Sequence (5,8,3,1,6)
   - The sort criteria is the sequence is the item itself
   - Ascending ordered of sequence elements: (1,3,5,6,8)

2. The sequence of entities (called records or items) containing two fields name and mark
   - Sequence: ((Paul,9), (Joan, 10), (Victor,8), (Ana,9))
   - In this case there are two possible sorting criteria: name and mark
   - Ascending ordered of sequence elements having name like order criteria: ((Ana,9),(Joan,10),(Paul,9),(Victor,8))
   - Ascending ordered of sequence elements having mark like order criteria: ((Joan,10),(Paul,9),(Anna,9),(Victor,8))

# The problem of sorting
# Something more formal…

Ordering (ascending) the sequence $(x_1,x_2,...,x_n)$ is equivalent to finding a permutation $(p(1),p(2),...,p(n))$ of the indices so that:

$$key(x_{p(1)}) <= key(x_{p(2)}) <= \ldots <= key(x_{p(n)})$$

In the following we will consider the sort key as represented by the entire element (the sequence consists of values belonging to set on which there is an order relationship)

In this hypothesis the sequence is considered ascending order if it satisfies:

$$x_{p(1)} <= x_{p(2)} <= \ldots <= x_{p(n)}$$

# The problem of sorting
# Other assumptions

We assume that the sequence is stored in the form of an array, so quick access to any of its elements is ensured. This means that it is internal sorting;

External sorting corresponds to the case where the entire sequence cannot be accessed simultaneously (for example, the sequence is very large, so it cannot be fully loaded into the computer's internal memory), which requires specific sorting methods;

We will only analyze methods that transform the current array without building another array (the additional memory space has the size of order O(1)).

# Properties of sorting methods

- Efficiency
  - The sorting method should require a reasonable amount of resources (lead time)
- Stability
  - A sorting method is stable if it preserves the relative order of items with the same sort key value
- Simplicity
  - The sorting method should be simple to understand and implement
- Naturalness
  - A sorting method is considered natural if the number of operations required is proportional to the degree of 'disorder' of the original sequence (which can be measured by the number of inversions in the permutation corresponding to the sorted sequence)

# Properties of sorting methods
# Example stability

- Initial configuration
  - ((Anna,9), (Joan, 10), (Paul,9), (Victor,8))

- Stable sorting
  - ((Joan,10),(Anna,9),(Paul,9),(Victor,8))
  - If the sorting criteria is name (all different)

- Unstable sorting
  - ((Joan,10), (Paul,9),(Anna,9), (Victor,8))
  - If the sorting criteria is mark (Paul and Anna have the same mark)

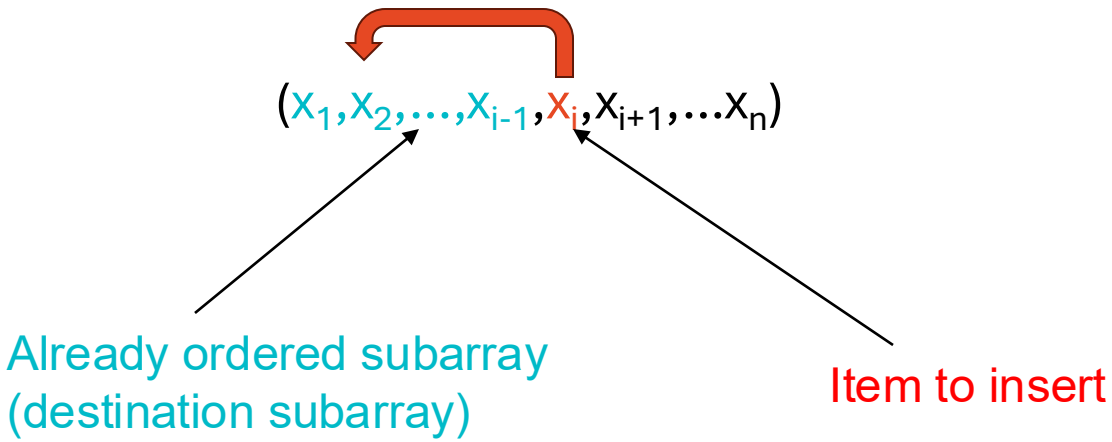
# Basic sorting methods

- They are simple, intuitive but not very efficient…
- They are, however, starting points for more advanced methods.
- Examples:
  - Insertion sort
  - Selection sort
  - Swapping neighboring elements sort

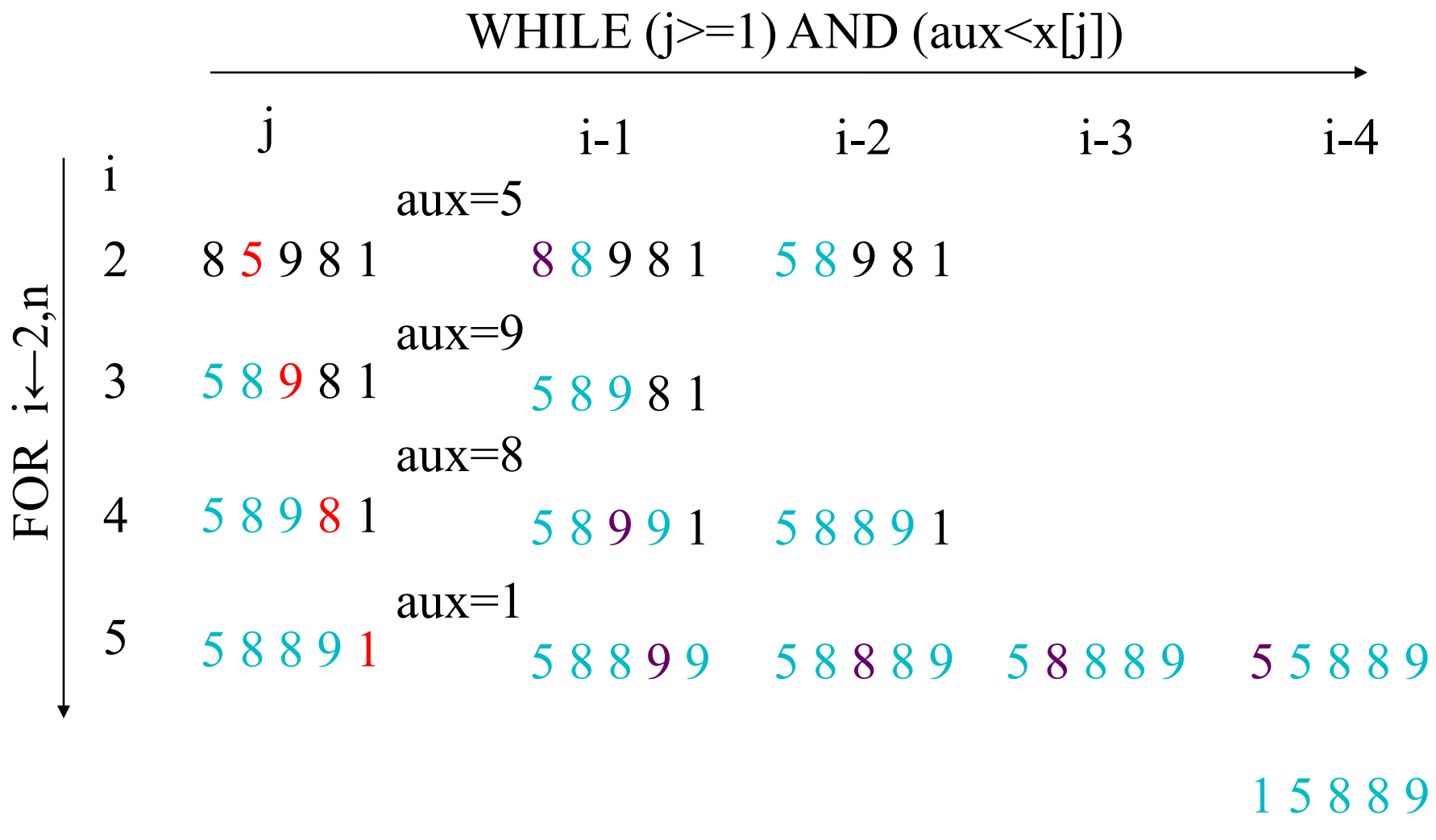
# Insertion sort

Basic idea

- Each element of the array, beginning with the second, is inserted into the sub-table preceding it, so that it remains orderded:



# Insertion sort



# Insertion sort Algorithm

## General structure

FOR i ← 2,n DO

<insert x[i] in subarray x[1..i-1]

such that remains x[1..i] sorted>

ENDFOR

## Algorithm

```
Insertion(x[1..n])
FOR i ← 2,n DO
    aux ← x[i]
    j ← i-1
    WHILE (j>=1) AND (aux<x[j]) DO
        x[j+1] ← x[j]
        j ← j-1
    ENDWHILE
    x[j+1] ← aux
ENDFOR
 RETURN x[1..n]
```

# Insertion sort

# Algorithm - variant

Algorithm

```
Insertion(x[1..n])
FOR i ← 2,n DO
    aux ← x[i]
    j ← i-1
    WHILE (j>=1) AND (aux<x[j]) DO
        x[j+1] ← x[j]
        j ← j-1
    ENDWHILE
    x[j+1] ← aux
ENDFOR
 RETURN x[1..n]
```

X[0] used like an auxiliary space →

Algorithm

```
Insertion(x[0..n])
FOR i ← 2,n DO
    x[0] ← x[i]
    j ← i-1
    WHILE (j>=1) AND (x[0]<x[j]) DO
        x[j+1] ← x[j]
        j ← j-1
    ENDWHILE
    x[j+1] ← x[0]
ENDFOR
 RETURN x[1..n]
```

# Insertion sort
# Correctness check

```
Insertion(x[1..n])
i←2                                  {x[1..i-1] It's sorted}
WHILE i<=n
    aux ← x[i]
    j ← i-1                          {x[1..j] It's sorted,  aux<=x[j+1]<=...<=x[i]}
    WHILE (j>=1) AND (aux<x[j]) DO
        x[j+1] ← x[j]                {x[1..j-1] It's sorted, aux<x[j]=x[j+1]<= … <=x[i]}
        j ← j-1                      {x[1..j] It's sorted, aux<x[j+1]=x[j+2]<= … <=x[i]}
    ENDWHILE
        This satisfies {aux>=x[j]  and x[1]<=…x[j]<=aux<x[j+1]=x[j+2]<=…<=x[i]} or {(j=0)  and  aux<x[1]=x[2]<=…<=x[i]}
    x[j+1] ← aux                     {x[1]<=x[2]<=…x[j+1]<=x[j+2]<=…<=x[i]} (x[1..i] It's sorted)
    i ← i+1                          {x[1..i-1] It's sorted}
ENDWHILE
RETURN x[1..n]
```

# Insertion sort
# Correctness check

So we got that...

Invariant for the outer cycle can be considered: **{x[1..i-1] Sorted}**

Invariant for inner cycle: **{x[1..j] It's sorted, aux<=x[j+1]=x[j+2]<=..<=x[i]}**

Both cycles are finite:

- Termination function for outer cycle: $t(p)=n+1-i_p$
- Termination function for inner cycle:

$$t(p)=\begin{cases} j_p & \text{if } aux < x[j_p] \\ 0 & \text{if } aux >= x[j_p] \text{ sau } j_p=0 \end{cases}$$

# Insertion sort
# Efficiency check

Algorithm

```
Insertion(x[1..n])
FOR i ← 2,n DO
    aux ← x[i]
    j ← i-1
    WHILE (j>=1) AND (aux<x[j]) DO
        x[j+1] ← x[j]
        j ← j-1
    ENDWHILE
    x[j+1] ← aux
ENDFOR
 RETURN x[1..n]
```

Problem size: n

Dominant operations:

- Comparison ($T_C(n)$)

For each i (2 <=i <= n):

1 <=$T_C(n)$ <= ii

For all values of i:

n-1 <=$T_C(n)$ <=n(n+1)2-1

- Items moves ($T_M(n)$)

For each i (2 <=i <= n):

0+2 <=$T_M(n)$ <=(i-1)+2=i+1

For all values of i:

2 (n-1) <=$T_M(n)$ <=(n+1)(n+2)/2-3

# Insertion sort
# Efficiency check

**Comparisons:**

$(n-1) <= T_C(n) <= (n+2)(n-1)/2$

**Moves:**

$2\,(n-1) <= T_M(n) <= (n+1)(n+2)/2-3$

**Total:**

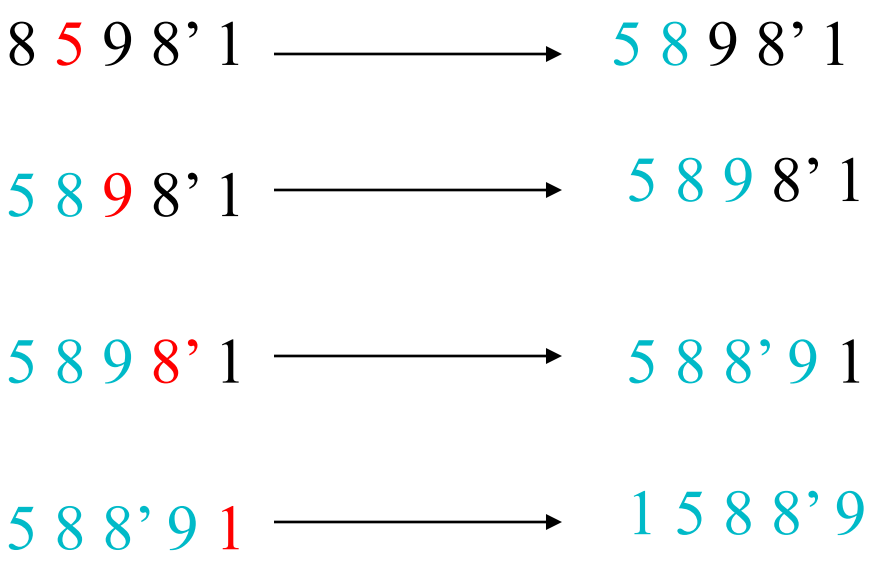
**$3(n-1) <= T(n) <= n^2+2n-3$**

Complexity classes (efficiency):

$T(n)$ in $\Omega(n)$

$T(n)$ in $O(n^2)$

# Insertion sort
# Stability

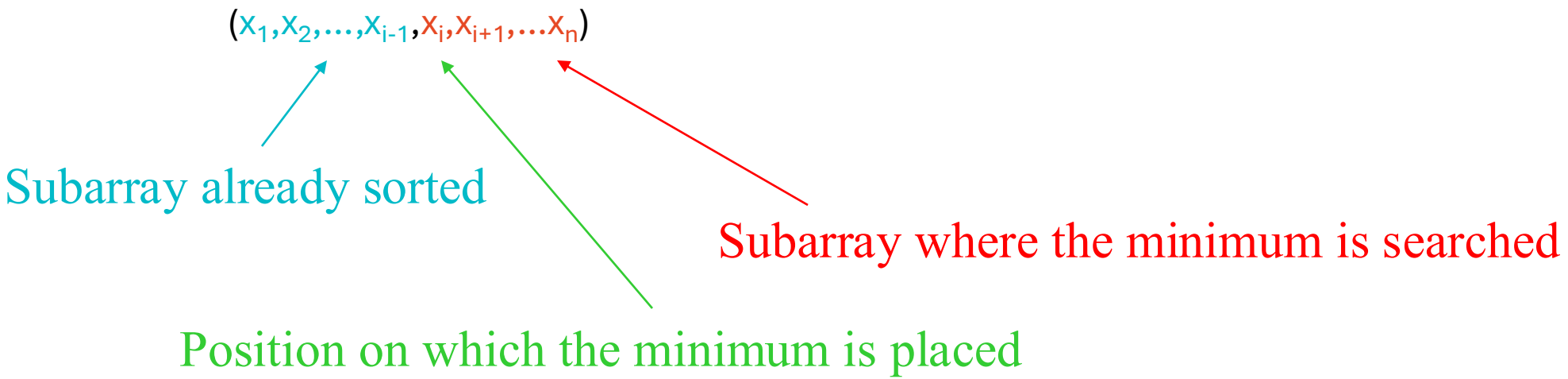



Sorting by insert is stable

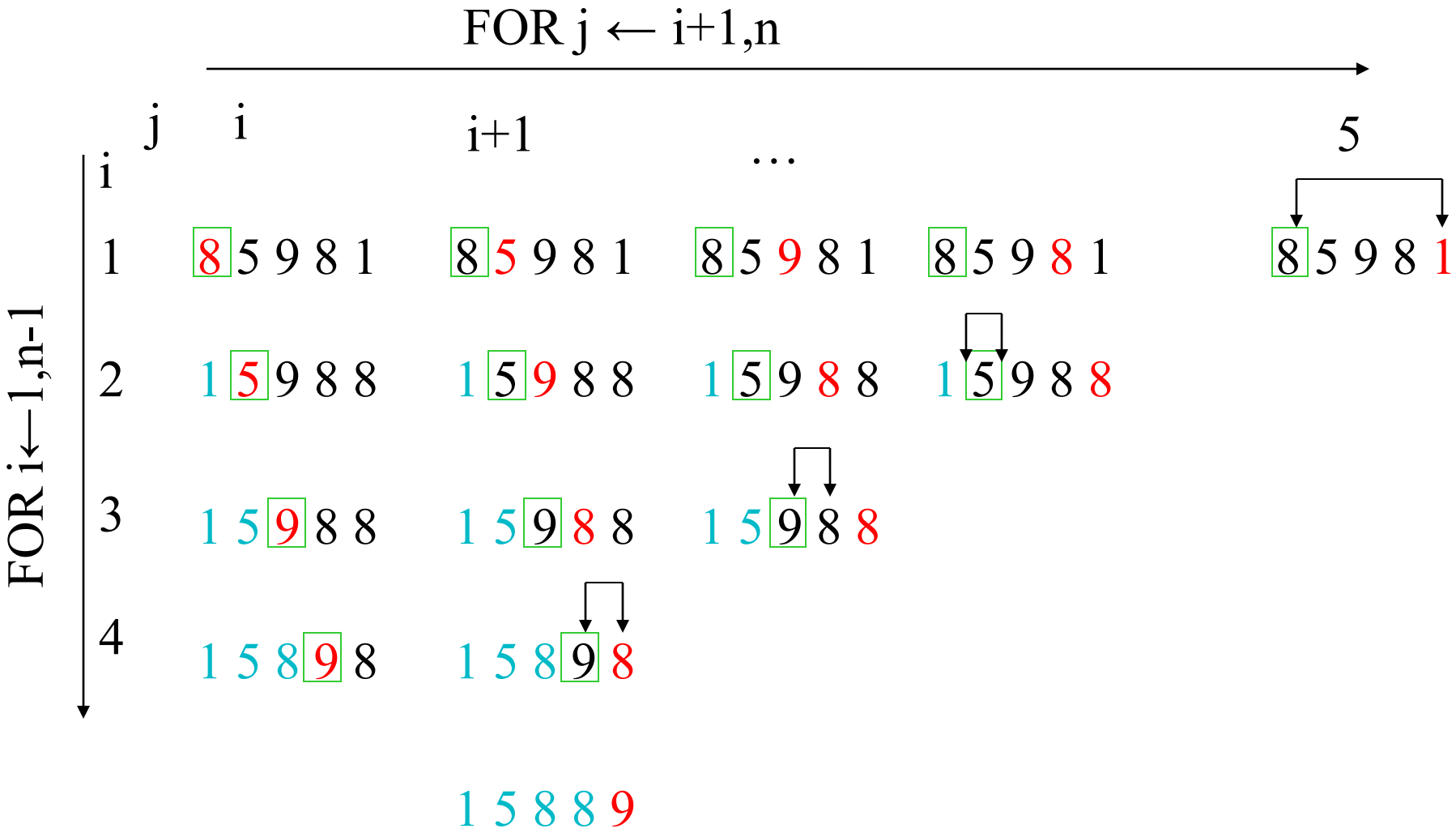
# Selection sort

Basic idea

- For each position , i (starting with the first) look for the minimum in subtable x[i.. n] and interchange the found minimum with the element in position i:



# Selection sort



# Selection sort

## General structure

```
FOR i ← 1,n-1 DO
        <search the minimum in x[i..n]
                and interchange it with x[i]>
ENDFOR
```

## Algorithm

```
selection (x[1..n])
FOR i←1,n-1 DO
   k ← i;
   FOR j ← i+1,n DO
      IF x[k]>x[j] THEN k ← j ENDIF
  ENDFOR
   IF k<>i
      THEN  x[i]<->x[k]
   ENDIF
ENDFOR
RETURN x[1..n]
```

# Selection sort

# Correctness check

Algoritm

```
Selection (x[1..n])
i ← 1                                  {x[1..i-1] sorted and x[i-1]<=x[j], j=i..n}
WHILE i<=n-1 DO
    k ← i
    j ← i+1                            {x[k]<=x[r] for every r=i..j-1}
    WHILE j<=n  DO
       IF x[k]>x[j] THEN k ← j ENDIF
       j ← j+1
    ENDWHILE                           {x[k]<=x[r] for every r=i..n}
    IF k<>i THEN x[i]<->x[k] ENDIF     {x[1..i] sorted and x[i]<=x[j], j=i..n}
    i ← i+1                            {x[1..i-1] sorted and x[i-1]<=x[j], j=i..n}
ENDWHILE
RETURN x[1..n]
```

# Selection sort
# Correctness check

So we got that...

Invariant for the outer cycle can be considered:

**{x[1..i-1] sorted and x[i-1]<=x[j], j=i..n}**

Invariant for the inner cycle can be considered:

**{x[k]<=x[r] for every r=i..j-1}**

Both cycles are finite:

- termination function for outer cycle: $t(p)=n-i_p$
- termination function for inner cycle: $t(p)=n+1-j_p$

# Selection sort

# Efficiency analysis

Algoritm

```
Selection (x[1..n])
i ← 1}
WHILE i<=n-1 DO
    k ← i
    j ← i+1
    WHILE j<=n  DO
       IF x[k]>x[j] THEN k ← j ENDIF
       j ← j+1
    ENDWHILE
    IF k<>i THEN x[i]<->x[k] ENDIF
    i ← i+1
ENDWHILE
RETURN x[1..n]
```

Problem size: n

Operations:

- Comparison ($T_C(n)$)

For each i  (1 <= i <= n-1):

$T_C(n,i) = n-i$

For all values of i:

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

- Items moves ($T_M(n)$)

For each:

$0 <= T_M(n) <= 3$

For all values of i:

$0 <= T_M(n) <= 3(n-1)$

# Selection sort
# Efficiency analysis

Comparisons:

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

Moves:

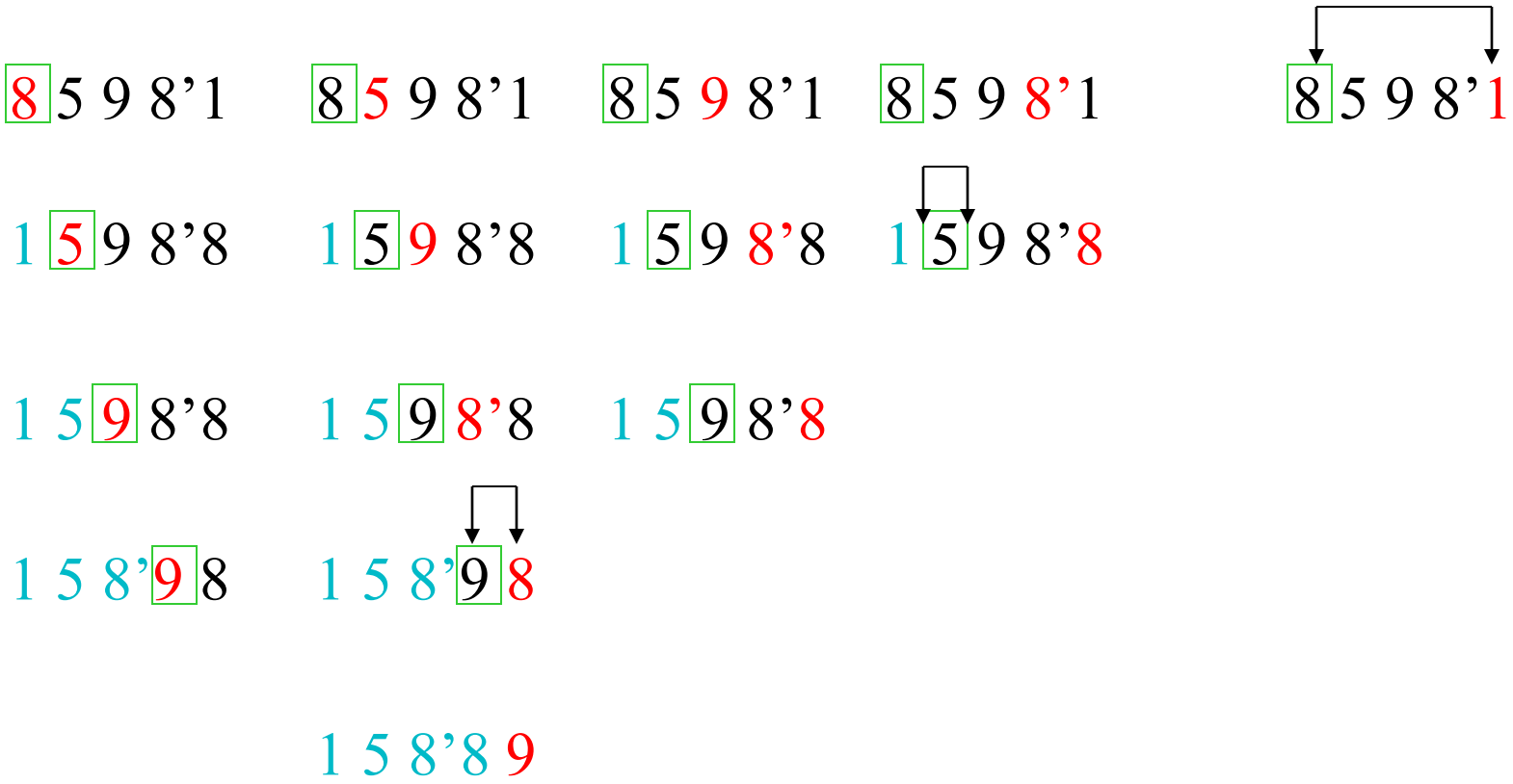
$0 <= T_M(n) <= 3(n-1)$

Total:

$n(n-1)/2 <= T(n) <= n(n-1)/2+3(n-1)$

Efficiency class (complexity):T(n) is $O(n^2)$

# Selection sort
# Stability

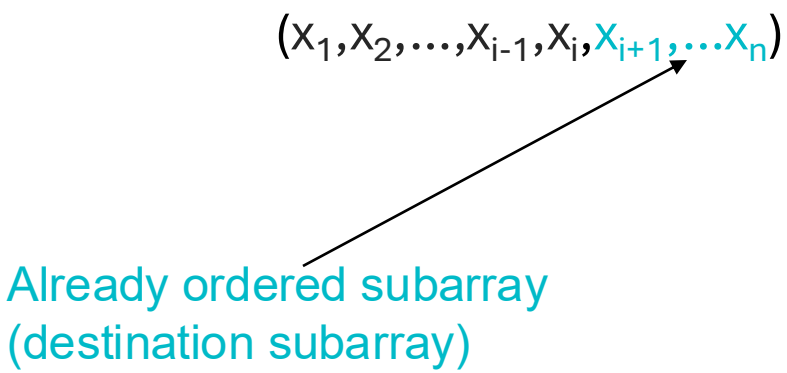



Selection sort is not stable

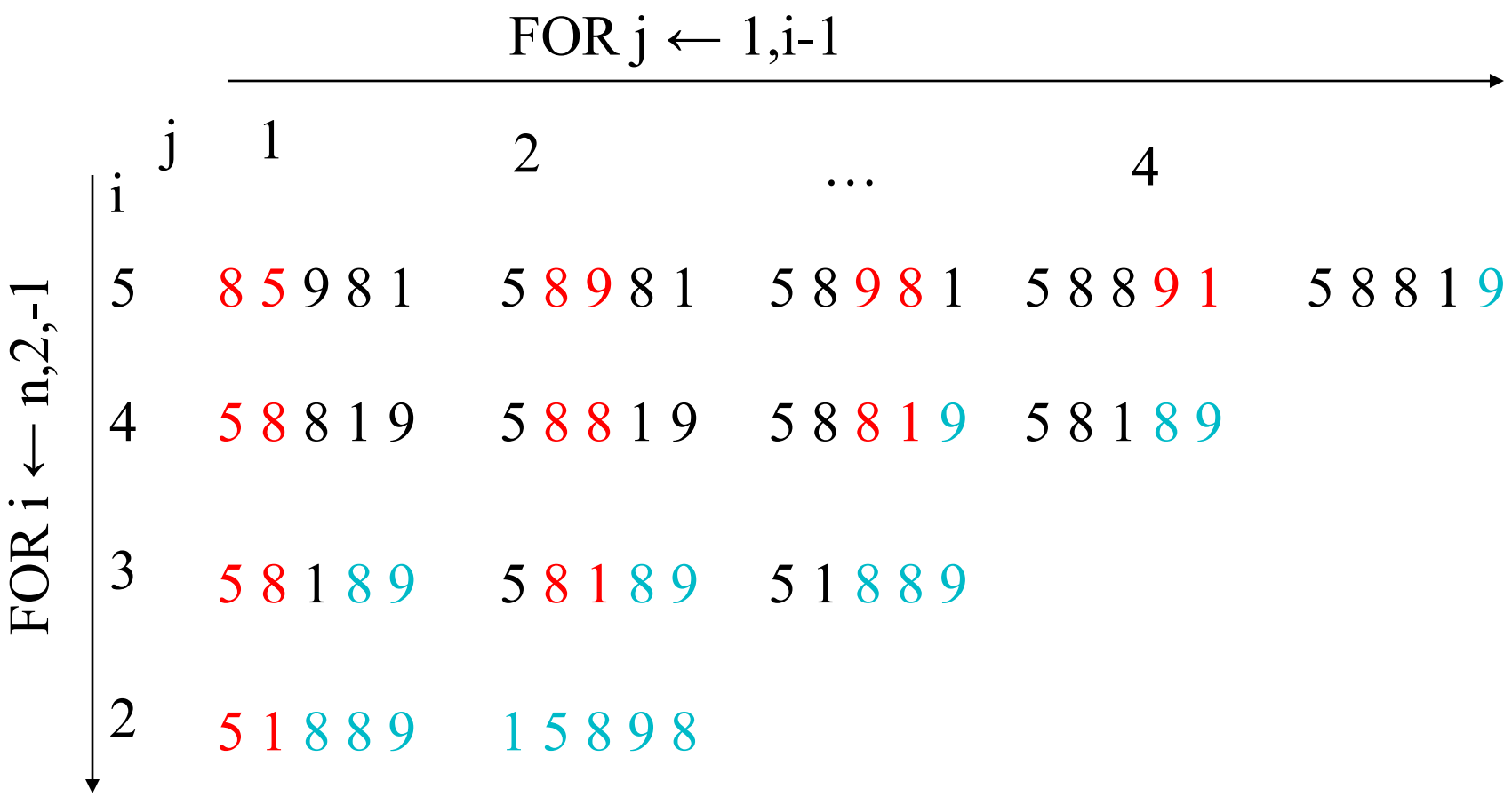
# Sort by swapping neighboring elements

Basic idea

- The array is iterated from left to right and adjacent elements are compared. If they are not in the desired order, then they interchange. The process is repeated until the table is sorted



# Sort by swapping neighboring elements



# Sort by swapping neighboring elements

## General structure

FOR i ← n,2,-1 DO

< In x[1..i-1], compare the neighboring elements and interchange elements if necessary >

ENDFOR

## Algorithm

```
Bubblesort(x[1..n])
FOR i ← n,2,-1 DO
  FOR j ← 1,i-1 DO
    IF x[j]>x[j+1]
        THEN x[j]<->x[j+1]
    ENDIF
  ENDFOR
ENDFOR
RETURN x[1..n]
```

# Sort by swapping neighboring elements Correctness check

Algorithm

```
Bubblesort(x[1..n])
i ← n                          {x[i+1..n] is sorted and x[i+1]>=x[j], j=1..i}
WHILE i>=2 DO
  j ← 1                                                {x[j]>=x[k], k=1..j-1}
  WHILE j<=i-1 DO
      IF x[j]>x[j+1]   THEN x[j]<->x[j+1]      {x[j+1]>=x[k], k=1..j}
      j ← j+1                                          {x[j]>=x[k], k=1..j-1}
   ENDWHILE                                            {x[i-1]>=x[j], j=1..i-1}
                          {x[i..n] is sorted and x[i]>=x[j], j=1..i-1}
  i ← i-1
ENDWHILE                   {x[i+1..n] is sorted and x[i+1]>=x[j], j=1..i}
RETURN x[1..n]
```

# Sort by swapping neighboring elements Correctness check

So we got that ...

Invariant for the outer cycle may refer to:

**{x[i+1..n] is sorted, and x[i+1]>=x[j], j=1..i}**

An invariant for the inner cycle can be:

**{x[j]>=x[k], k=1..j-1}**

Both cycles are finite

- Termination function for outer cycle: $t(p)=i_p-1$
- Termination function for inner cycle: $t(p)=i_p-j_p$

# Sort by swapping neighboring elements Efficiency check

## Algorithm

```
Bubblesort(x[1..n])
FOR i ← n,2,-1 DO
   FOR j ← 1,i-1 DO
     IF x[j]>x[j+1]
           THEN x[j]<->x[j+1]
     ENDIF
  ENDFOR
ENDFOR
RETURN x[1..n]
```

Problem size: n

Operations:

- Comparisons ($T_C(n)$)

For each i (1 <=i <= n-1):

$T_C(n,i) = i-1$

For all values of i:

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

- Element moves ($T_M(n)$)

For each i (2 <= i <= n):

$0 <= T_M(n,i) <= 3(i-1)$

For all values of i:

$0 <= T_M(n) <= 3n(n-1)/2$

# Sort by swapping neighboring elements Efficiency check

Comparisons:

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

Moves

$0 <= T_M(n) <= 3n(n-1)/2$

Total:

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

Efficiency class (complexity): T(n) is $O(n^2)$

Remark. This variant of implementation is the least effective. Better variants avoid executing (n-1) times the outer loop, stopping processing when the array is already sorted (no element interchange is done in the inner loop)

# Sort by swapping neighboring elements Variants

Algorithm

```
Bubblesort(x[1..n])
sw ← true
WHILE sw = true DO
   sw ← flase
   FOR j ← 1,n-1 DO
     IF x[j]>x[j+1] THEN
          x[j]<->x[j+1]; sw ← True
      ENDIF
  ENDFOR
ENDFOR
RETURN x[1..n]
```

The array is iterated again only if at the previous step an element interchange was done

$n-1<=T_c(n)<=n(n-1)$

Algorithm

```
Bubblesort(x[1..n])
t ← n
WHILE t > 1 DO
   m ← t; t ← 0
   FOR j ← 1,m-1 DO
     IF x[j]>x[j+1] THEN
          x[j]<->x[j+1]; t ← m
     ENDIF
  ENDFOR
ENDFOR
RETURN x[1..n]
```

The array is iterated again until interchange position

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

# Sort by swapping neighboring elements Stability

8 5 9 8’ 1  5 8 9 8’ 1  5 8 9 8’ 1  5 8 8’9 1  5 8 8’ 1 9

5 8 8’1 9  5 8 8’1 9  5 8 8’1 9  5 8 1 8’ 9

5 8 1 8’ 9  5 8 1 8’ 9  5 1 8 8’ 9

5 1 8 8’ 9  1 5 8 9 8’

Bubble sort is stable
(provided strict inequality is used: x[j]>x[j+1])

# Summary

Classical sorting techniques (insertion, selection, swapping of neighboring elements) have quadratic complexity - practical only for arrays with a small number of elements (on the order of hundreds).

- Insertion: advantageous if the array is “nearly sorted”; stable
- Selection: useful when partial sorting is required (e.g., searching for the first k < n elements in ascending order; unstable
- Bubblesort: to be avoided (especially the first variant)

For n=1000000 and a duration of one nanosecond/operation, a sorting algorithm in $O(n^2)$ would require about 17 minutes. If T(n) were in O(nlogn) then for the same value of n 0.019 seconds would be sufficient

# Sorting ...

## *By comparison*

**Insertion:** From a given list of items, one item is considered at a time. The item chosen is then inserted into an appropriate position relative to the previously sorted items. The item can be inserted into the same list or to a different list.

e.g.: Insertion sort

**Selection:** First the smallest (or largest) item is located and it is separated from the rest; then the next smallest (or next largest) is selected and so on until all item are separated.

e.g.: Selection sort, Heap sort

**Exchange:** If two items are found to be out of order, they are interchanged. The process is repeated until no more exchange is required.

e.g.: Bubble sort, Shell Sort, Quick Sort

**Enumeration:** Two or more input lists are merged into an output list and while merging the items, an input list is chosen following the required sorting order.

e.g.: Merge sort

## *By distribution*

No key comparison takes place

All items under sorting are distributed over an auxiliary storage space based on the constituent element in each and then grouped them together to get the sorted list.

Distributions of items based on the following choices

- **Radix** - An item is placed in a space decided by the bases (or radix) of its components with which it is composed of.
- **Counting** - Items are sorted based on their relative counts.
- **Hashing** - Items are hashed, that is, dispersed into a list based on a hash function

# Useful resources

- http://www.sorting-algorithms.com/
- http://www.softpanorama.org/Algorithms/sorting.shtml
- http://www.youtube.com/watch?v=INHF_5RIxTE

# Next course

Correctness verification and efficiency analysis of sorting algorithms

- Insertion sort
- Selection sort
- Neighbor swap sort

# Q&A