Data structures and Algorithms

SORTING ALGORITHMS

(Part I)

Nguyễn Ngọc Thảo
nnthao@fit.hcmus.edu.vn

Outline

- Selection sort
- Insertion sort
- Bubble sort
- Interchange sort

The sorting problem

- Sorting is a process that organizes a data collection based on a predefined order.

- There are tasks that need manipulations on sorted data.
    - E.g., a list of students is arranged following their names or scores, letters in the alphabet, words in a dictionary, etc.
- Sort can serve as an initialization step for certain algorithms.
    - E.g., binary search must run on sorted data.

The sorting algorithms

$O(n^2)$	Selection sort, Insertion sort, Interchange sort, Bubble sort
$O(n\log_2 n)$	Heap sort, Quick sort*, Merge sort
$O(n)$	Counting sort**, Radix sort**

* Quicksort is $O(n^2)$ in the worst case.

** Radix sort and counting sort are non-comparison sorting algorithm.

Selection sort

Selection sort: Idea

- Let the list be divided into two sublists, *sorted* and *unsorted*, by an imaginary wall.
- Find the smallest element from the unsorted part and **swap** it with the element at the beginning of the unsorted data
  - After each selection and swapping, the size of the sorted region grows by 1 and the size of the unsorted region shrinks by 1.
- A list of n elements requires $n-1$ passes to rearrange the data completely.
  - Each time we move one element from the unsorted sublist to the sorted sublist, we have completed a sort pass.

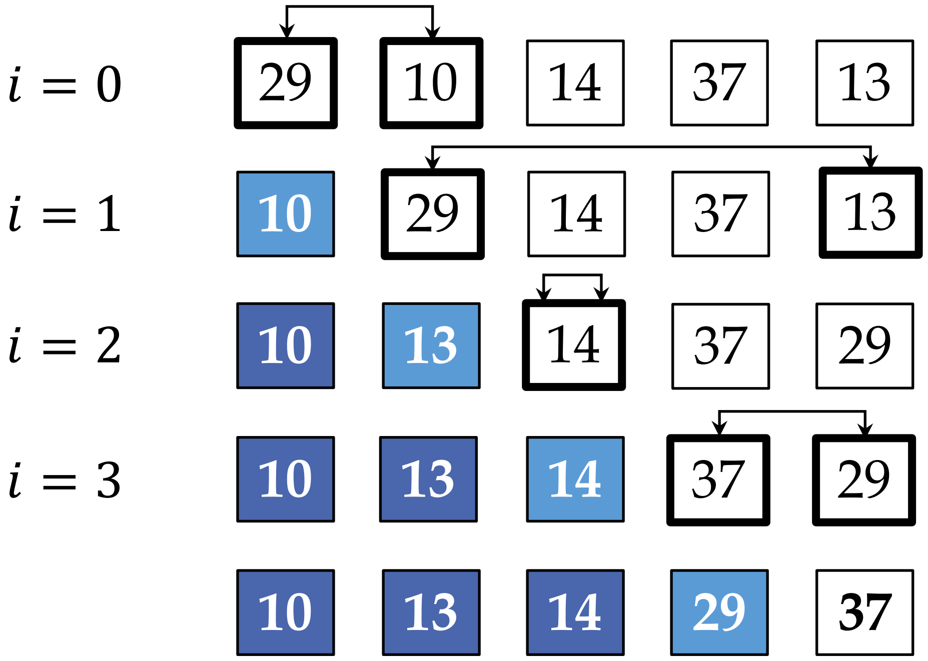
Selection sort: Algorithm

- Consider the array of n elements, $a[1..n]$.

- Step 1. Set the increment variable $i = 1$
- Step 2. Find the smallest element in $a[i..n]$, and then swap it with $a[i]$.
- Increase i by 1 and go to **Step 3**
- Step 3. Check whether the end of the array is reached by comparing i with n.
  - If $i < n$ then go to **Step 2** (The first i elements are in place.)
  - Otherwise, **stop the algorithm**

Example: Selection sort on an array of integers

Sort the following array of integers, **{29, 10, 14, 37, 13}**



Selection sort: Implementation

```
void selectionSort(int arr[], int n){
  for (int i = 0; i < n - 1; ++i){
    // At this point, arr[0..i-1] is sorted, and its
    // entries are smaller than those in arr[i..n-1].
    int minIdx = i;
    // Select the smallest entry in arr[i..n-1]
    for (int j = i + 1; j < n; ++j)
      if (arr[j] < arr[minIdx])
        minIdx = j;
    // Swap the smallest entry, arr[minIdx], with arr[i]
    swap(arr[minIdx], arr[i]);
  }
}
```

Selection sort: An analysis

- The number of comparisons: $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$
  - The inner loop executes the size of the unsorted part minus 1, and in each iteration, there is one key comparison.
- The number of assignments: $3(n-1)$
  - The outer loop runs $n-1$ times and calls swapping once at each iteration.
- Together, the number of key operations that a selection sort of n elements requires is

$$\frac{n(n-1)}{2}+3(n-1)=\frac{n^2}{2}+\frac{5n}{2}-3$$

- Thus, selection sort is $\mathbf{O}(\boldsymbol{n}^{\mathbf{2}})$ in all cases.

Selection sort: An analysis

- Selection sort is independent of the distribution of input data.
- It is appropriate only for small n since $O(n^2)$ grows rapidly.
- It could be a good choice over other sorting methods when data moves are costly, but comparisons are not.
  - $O(n^2)$ comparisons and $O(n)$ data moves
  - That is when each data item is lengthy but the sort key is short.

Checkpoint 01: Selection sort on an array

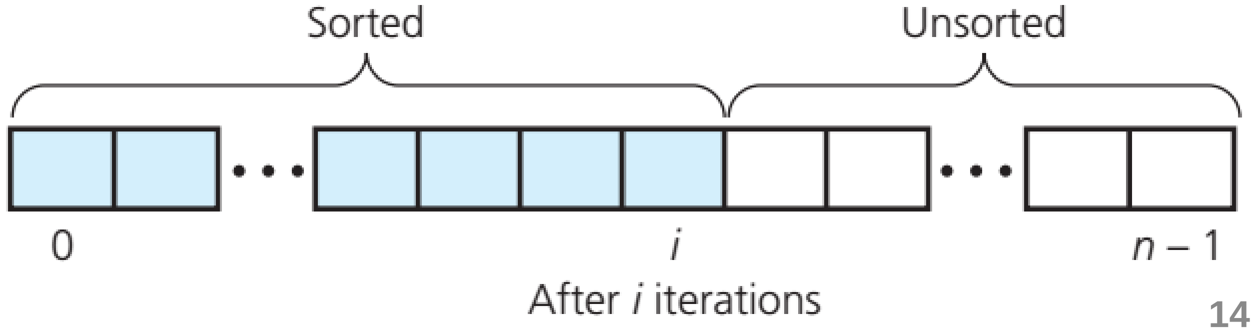
Trace the **selection sort** as it sorts the following array into **ascending order**, **{20, 80, 40, 25, 60, 30}**.

Insertion sort

Insertion sort: Idea

- Let the list be divided into two sublists, *sorted* and *unsorted*, by an imaginary wall.
- Take the first element of the unsorted region and **place** it into its correct position in the sorted region
  - After each placement, the size of the sorted region grows by 1 and the size of the unsorted region shrinks by 1.
- A list of n elements requires $n-1$ passes to rearrange the data completely.

An insertion sort partitions the array into two regions



Insertion sort: Algorithm

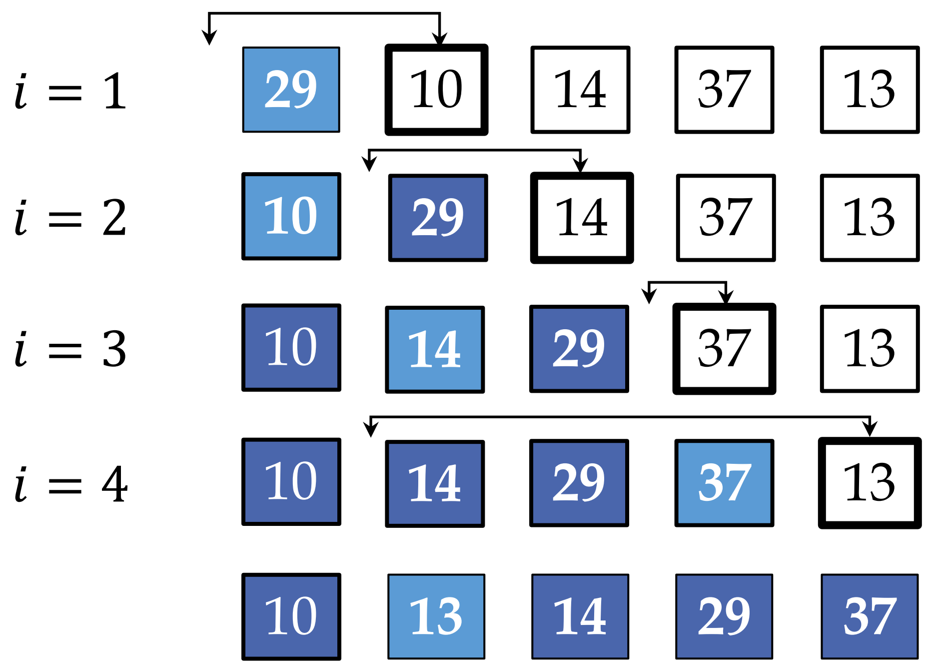
- Consider the array of n elements, $a[1..n]$.
- Step 1. Set the increment variable $i = 2$.
- Step 2. Find the correct position pos in $a[1..i-1]$ to insert $a[i]$, i.e., where $a[pos-1] \leq a[i] \leq a[pos]$
  - Set $x = a[i]$ $\rightarrow$ move forward $a[pos..i-1]$ one element $\rightarrow$ set $a[pos] = x$
- Increase i by 1 and go to **Step 3**.
- Step 3. Check whether the end of the array is reached by comparing i with n.
  - If $i \leq n$ then go to **Step 2**. Otherwise, **stop the algorithm**.

Insertion sort implementation

```
void insertionSort(int arr[], int n){
   for (int i = 1; i < n; ++i){
      // Find the right position in the sorted region arr[0..i-1]
      // for arr[i]; shift, if necessary, to make room
      int key = arr[i];
      int j = i-1;
      while (j >= 0 && arr[j] > key){
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
```

Example: Insertion sort on an array of integers

Sort the following array of integers, **{29, 10, 14, 37, 13}**



Insertion sort: An analysis

- The number of comparisons: $1 + 2 + \cdots + (n-1) = \dfrac{n(n-1)}{2}$
  - The inner loop executes the size of the sorted part, and in each iteration, there is one key comparison.
- The number of assignments: $\dfrac{n(n-1)}{2} + 2(n-1)$
  - The inner loop moves data items at most the same number of times for comparisons. The outer loop moves data items twice per iteration.
- Together, the number of key operations that an insertion sort of n elements requires in the worst case is

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

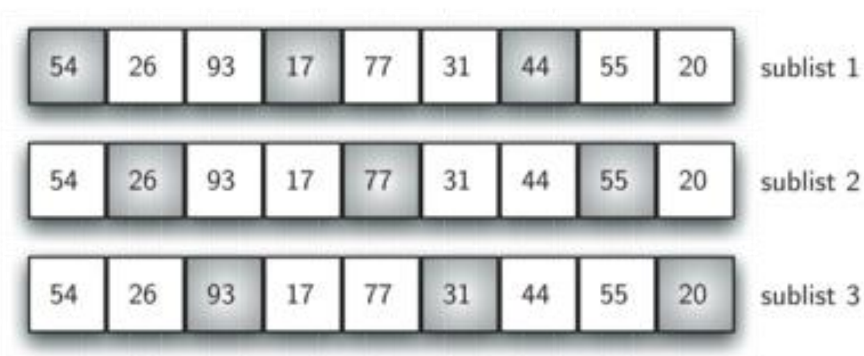
Insertion sort: An analysis

Best case	Worst case	Average case
$\mathbf{O}(\boldsymbol{n})$	$\mathbf{O}(\boldsymbol{n}^2)$	$\mathbf{O}(\boldsymbol{n}^2)$

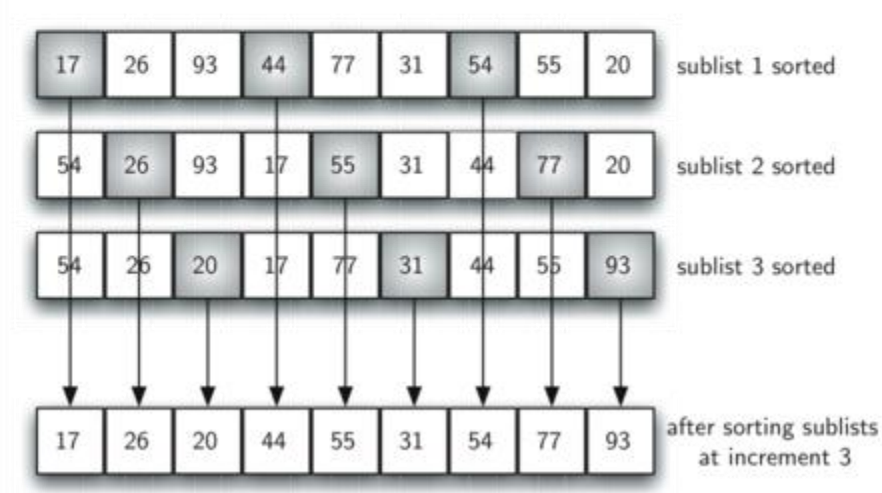
- The time complexity is affected by not only the size but also the distribution of the input data.
- It can also be useful when input array is almost sorted.

Insertion sort: Improvements

- Binary insertion sort: Use binary search to find the correct position for insertion.
  - The search cost may reduce, yet the cost of data moving remains.
- Shell sort: Exchange items that are far apart h steps in the array
  - Every h^{th} item forms a sorted subarray in a decreasing sequence of values. Ultimately, if h is 1, the entire array will be sorted.



Initial sublists with an increment of three



After sorting the sublists

Image credit: pythonds3

Binary insertion sort: Implementation

```
void binaryInsertionSort(int arr[], int n){
  for (int i = 1; i < n; ++i){
    int key = arr[i];
    int first = 0, last = i - 1;
    while (first <= last) {
      int m = (first + last) / 2;
      if (key < arr[m]) last = m - 1;
      else first = m + 1;
    }
    for (int j = i - 1; j >= first; --j)
      arr[j + 1] = arr[j];
    arr[first] = key;
  }
}
```

Checkpoint 02: Insertion sort on an array

Trace the **insertion sort** as it sorts the following array into **ascending order**, **{20, 80, 40, 25, 60, 30}**.

Bubble sort

Bubble sort: Idea

- Let the list be divided into two sublists, *sorted* and *unsorted*, by an imaginary wall.
- Compare adjacent elements in the unsorted region and **exchange** them if they are out of order.
    - Ordering successive pairs of elements causes the extreme element “bubbles“ to either of the two ends of the array.
- A list of n elements requires $n - 1$ passes to rearrange the data completely.

The bubble sort algorithm

- Consider the array of n elements, $a[1..n]$.
- Step 1. Set the increment variable $i = 1$.
- Step 2. Swap any pair of adjacent elements in $a[1..n-i+1]$ if they are in wrong order.
  - Set the increment variable $j = 1$.
  - If $a[j] > a[j+1]$ then swap $a[j]$ with $a[j+1]$
  - Increase j by 1 and repeat **Step 2** until the end of the unsorted region.
- Increase i by 1 and go to **Step 3**
- Step 3. Check whether the data is sorted by comparing i with n
  - If $i < n$ then go to **Step 2** (The last i elements are in place)
  - Otherwise, **stop the algorithm**.

Bubble sort: Implementation

```
void bubbleSort(int arr[], int n){
   for (int pass = 1; pass < n; ++pass)
      for (int j = 0; j < n - pass; ++j){
         if (arr[j] > arr[j + 1])
            swap(arr[j], arr[j + 1]);
      // Last pass elements are already in place
 }
```

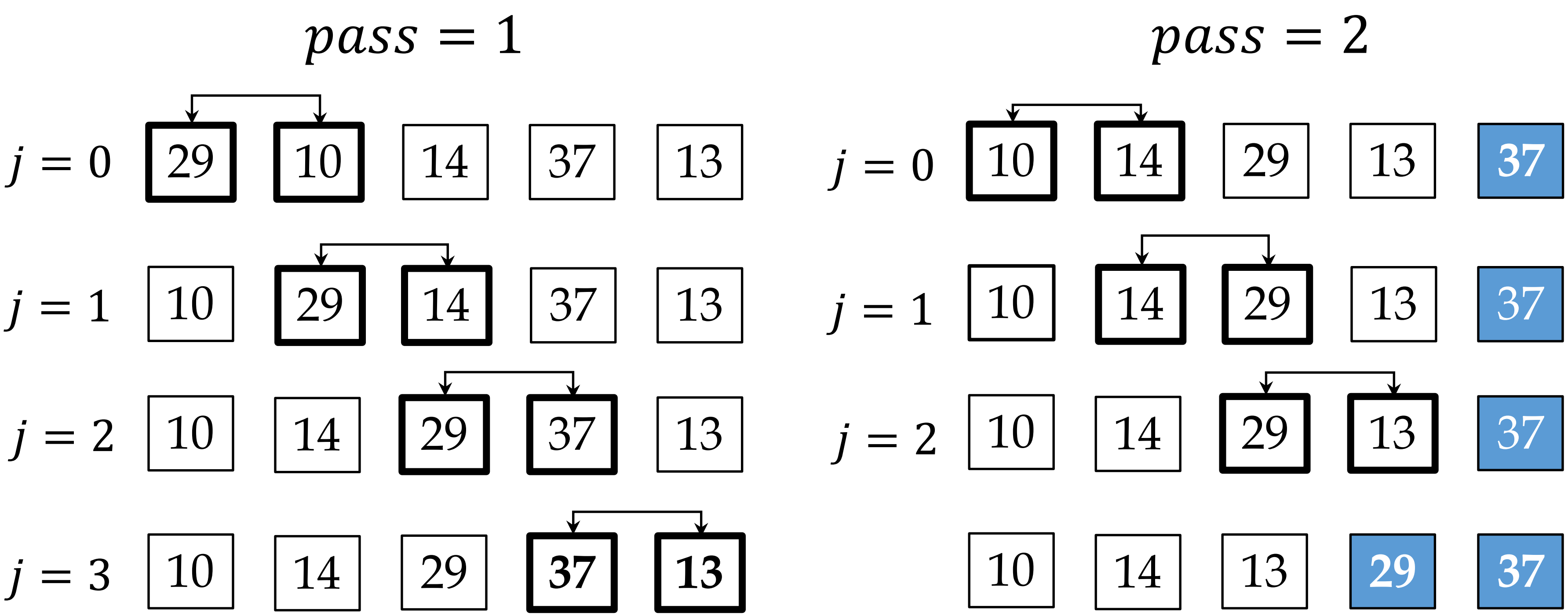
The largest item bubbles to the end of the array

The smallest item bubbles to the top of the array

```
void bubbleSort(int arr[], int n){
   for (int pass = 1; pass < n; ++pass)
      for (int j = n - 1; j >= pass; --j)
         if (arr[j] < arr[j - 1])
            swap(arr[j], arr[j - 1]);
      // Last pass elements are already in place
 }
```

Example: Bubble sort on an array of integers

Sort the following array of integers, **{29, 10, 14, 37, 13}**



…………………………………

An analysis of Bubble sort

- The number of comparisons: $(n-1)+(n-2)+\cdots+1=\dfrac{n(n-1)}{2}$
  - The inner loop executes the size of the unsorted part minus 1, and in each iteration, there is one key comparison.
- The number of exchanges: the same as above
  - Each exchange requires three assignments.
- Together, the number of key operations that a bubble sort of n elements requires in the worst case is

$$2n(n-1)=2n^2-2n$$

- Thus, vanilla bubble sort is $\mathbf{O}(\boldsymbol{n}^{\mathbf{2}})$ in all cases.

Checkpoint 03: Bubble sort on an array

Trace the **bubble sort** as it sorts the following array into **ascending order**, **{20, 80, 40, 25, 60, 30}**.

Bubble sort: Improvements

- The process stops if no exchanges occur during any pass.
- A Boolean variable can signal when an exchange occurs in a pass.

- The best case of bubble sort becomes $\mathbf{O}(\boldsymbol{n})$.

```
void bubbleSort(int arr[], int n){
    bool unsorted = true;
    int pass = 0;
    while (unsorted){
        unsorted = false;
        pass++;
        for (int j = 0; j < n - pass; ++j)
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                unsorted = true;
            }
    }
}
```

```
void shakerSort(int arr[], int n){
    int left = 1, right = n-1, k = n-1;
    do {
       for (int j = right; j >= left; --j)
          if (arr[j - 1] > arr[j]) {
             swap(arr[j - 1], arr[j]);
             k = j;
          } // Smaller elements to the top
       left = k + 1;
       for (int j = left; j <= right; ++j)
          if (arr[j - 1] > arr[j]) {
             swap(arr[j - 1], arr[j]);
             k = j;
          } // Larger elements to the end
       right = k - 1;
    } while (left <= right);
}
```

Shaker sort

- Remember whether any exchange had taken place during a pass
- Remember the position of the last exchange
- Alternate the direction of consecutive passes

Interchange sort

Interchange sort: Idea

- Let the list be divided into two sublists, *sorted* and *unsorted*, by an imaginary wall.
- Compare the element at the top of the unsorted region with every other subsequent element and **exchange** them if they are out of order
- A list of n elements requires $n-1$ passes to rearrange the data completely.

Interchange sort: Algorithm

- Consider the array of n elements, $a[1..n]$.
- Step 1. Set the increment variable $i = 1$.
- Step 2. Swap any element in $a[i+1..n]$ with $a[i]$ if they are in wrong order
  - Set the increment variable $j = 1$.
  - If $a[i] > a[j]$ then swap $a[j]$ with $a[i]$
  - Increase j by 1 and repeat **Step 2** until the end of the unsorted region is reached.
- Increase i by 1 and go to **Step 3**
- Step 3. Check whether the data is sorted by comparing i with n
  - If $i < n$ then go to **Step 2** (The first i elements are in place)
  - Otherwise, **stop the algorithm**

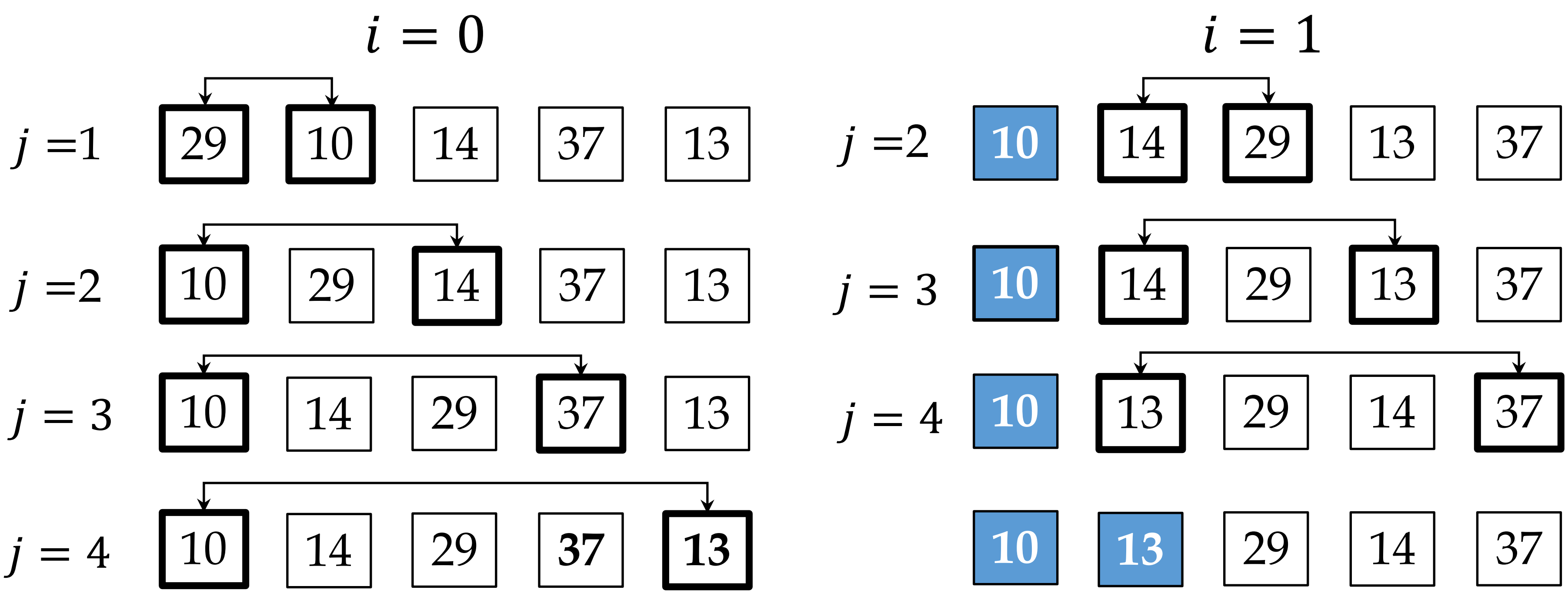
Implementation and Analysis

```
void interchangeSort(int arr[], int n){
  for (int i = 0; i < n - 1; ++i)
    for (int j = i + 1; j < n; ++j)
      if (arr[i] > arr[j])
        swap(arr[i], arr[j]);
}
```

- Similar to bubble sort, the number of key operations that an interchange sort of n elements requires in the worst case is
$$2n(n-1) = 2n^2 - 2n$$
- The interchange sort is $\mathbf{O}(\boldsymbol{n}^2)$ in all cases.

Example: Interchange sort on an array of integers

Sort the following array of integers, **{29, 10, 14, 37, 13}**



…………………………………

Checkpoint 04: Interchange sort on an array

Trace the **interchange sort** as it sorts the following array into **ascending order**: {20, 80, 40, 25, 60, 30}.

Acknowledgements

This part of the lecture is adapted from the following materials.

[1] Pr. Nguyen Thanh Phuong (2020) "*Lecture notes of CS163 – Data structures*" University of Science - Vietnam National University HCMC.

[2] Pr. Van Chi Nam (2019) "*Lecture notes of CSC14004 – Data structures and algorithms*" University of Science - Vietnam National University HCMC.

[3] Frank M. Carrano, Robert Veroff, Paul Helman (2014) "*Data Abstraction and Problem Solving with C++: Walls and Mirrors*" Sixth Edition, Addion-Wesley. **Chapter 10.**

[4] Anany Levitin (2012) "*Introduction to the Design and Analysis of Algorithms*" Third Edition, Pearson.

Exercises

01. Sorting algorithms on an array

- Consider the following array of integers, {26, 48, 12, 92, 28, 6, 33}.
- Apply each of the following sorting algorithms to arrange the elements in the given array in ascending order.
  - Selection sort
  - Insertion sort
  - Bubble sort
  - Interchange sort

02. Erroneous bubble sort

- The following pseudo-code fragment implements bubble sort in ascending order. Is the code valid? If no, suggest how to fix the errors.

```
for (i = 1; i < n; ++i)
   for (j = n – 1; j <= i; --j)
      if (a[j] > a[j - 1])
         a[j] = a[j - 1]);
         a[j-1] = a[j];
```

03. Parsimony in sorting algorithms

- A sorting algorithm is parsimonious if it never compares the same pair of input value twice (Assuming that all input values are distinct).
- Which of the following sorting algorithms is parsimonious?
  - Selection sort
  - Insertion sort
  - Bubble sort
  - Interchange sort
- Give an example or counter-example for each of the above algorithms.