Algorithmics

Sorting

Vicente García Díaz – garciavicente@uniovi.es
University of Oviedo, 2016

Table of contents

Basic concepts

Introduction

- Given a set of $\mathtt{n}$ elements $\mathtt{a_1,\ a_2,\ a_3,\ \ldots,\ a_n}$ and an order relation (≤) the problem of sorting **is to sort these elements increasingly**
- What can influence the sorting?
  - The type
  - The size
  - The device on which they are
- **Integers stored in a vector** (simplification)

Sorting integers

- In practice, the problems tend to be much more complex
  - However they can be reduced to the same problem that the integers (using registry keys, indexes, etc.)

- In essence it is the same to sort numbers or listed streets, houses, cars or any numerically quantifiable property

Classification criteria for algorithms

1. Total number of steps
   - Complexity
2. Number of comparisons performed
3. Number of interchanges performed
   - Much more expensive than the comparison
4. Stability
5. Type of memory used
   - Internal algorithms
   - External algorithms

Preliminary considerations (I)

- We will sort arrays of integers using the Java programming language `int[] vector;`
- We will make use of utility methods to facilitate the work and make the code more readable

```
/**
 * Interchange element i and element j
 * @param elements
 * @param i
 * @param j
 */
public static void interchange(int[] elements, int i, int j) {
    int temp = elements[i];
    elements[i] = elements[j];
    elements[j] = temp;
}
```

Preliminary considerations (II)

```java
/**
 * Find the position of the smallest element
 * @param elements
 * @param firstElement
 * @return position of the element
 */
public static int findPosMin(int[] elements, int firstElement) {
    int value = Integer.MAX_VALUE;
    int pos = Integer.MAX_VALUE;
    for (int i = firstElement; i < elements.length; i++){
        if (elements[i] < value){
            value = elements[i];
            pos = i;
        }
    }
    return pos;
}
```

```java
/**
 * Find the position of the biggest element
 * @param elements
 * @param firstElement
 * @return position of the element
 */
public static int findPosMax(int[] elements, int firstElement) {
    int value = Integer.MIN_VALUE;
    int pos = Integer.MIN_VALUE;
    for (int i = firstElement; i < elements.length; i++){
        if (elements[i] > value){
            value = elements[i];
            pos = i;
        }
    }
    return pos;
}
```

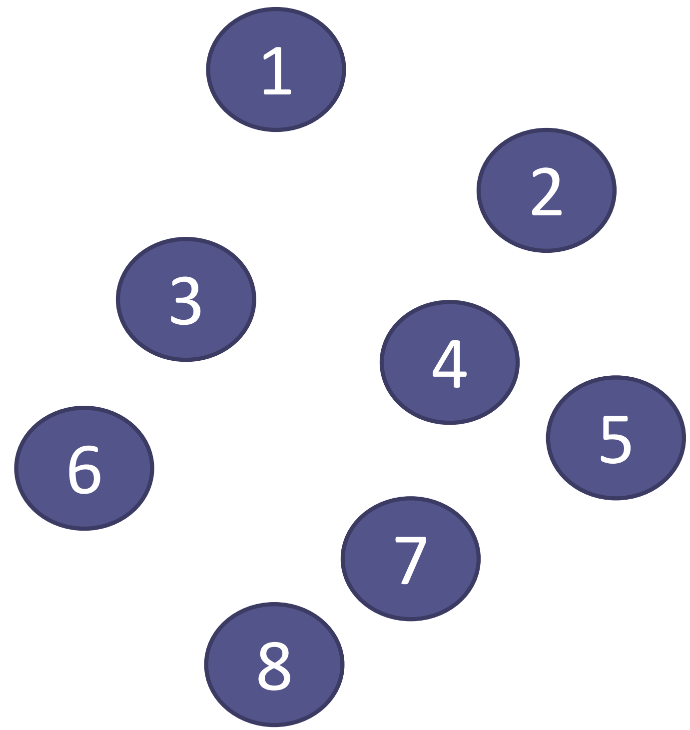
Sorting algorithms

Simple algorithms
Sophisticated algorithms
Constrained algorithms
External algorithms

Sorting by direct exchange
Sorting by insertion
Sorting by selection

Bubble

- Description
  - Based on the successive comparison and exchange of adjacent elements
  - At each step, each item is compared with the previous one and in case of being disordered, they are exchanged
  - In the first step, the smallest element will be placed in the left most position, and so on…



Bubble

- Initial vector: 4 5 6 1 3 2 7 8

1	1	4	5	6	2	3	7	8
2	1	2	4	5	6	3	7	8
3	1	2	3	4	5	6	7	8
4	1	2	3	4	5	6	7	8
5	1	2	3	4	5	6	7	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Bubble

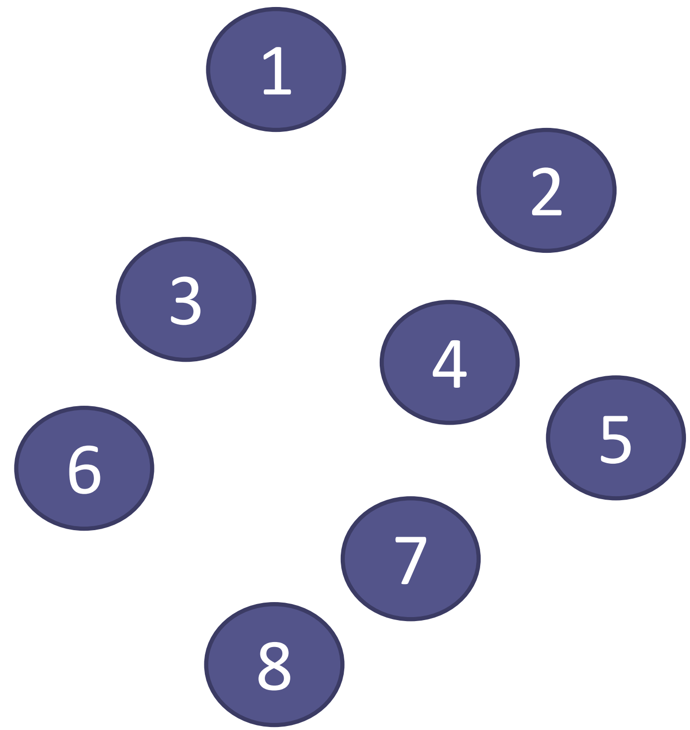
```
public void sort(int[] elements) {
    for (int i = 1; i < elements.length; i++) {
        for (int j = elements.length - 1; j >= i; j--) {
            if (elements[j-1] > elements[j]){
                Util.interchange(elements, j-1, j);
            }
        }
    }
}
```

- High number of exchanges
- High number of comparisons

Best case: $O(n^2)$
Worst case: $O(n^2)$
Average case: $O(n^2)$

Bubble with sentinel

- Description
  - Observing the previous result you can see that the last steps are repeated unnecessarily
  - To avoid this you can enter a stop condition when the vector is sorted
    - If you make a pass and you do not make any exchange it means that it is already sorted



Bubble with sentinel

- Initial vector:

4	5	6	1	3	2	7	8

1	1	4	5	6	2	3	7	8
2	1	2	4	5	6	3	7	8
3	1	2	3	4	5	6	7	8
4	1	2	3	4	5	6	7	8

Bubble with sentinel

```
public void sort(int[] elements) {
    int i = 1;
    boolean hasChange = true;

    while (hasChange && (i < elements.length)){
        hasChange = false;
        for (int j = elements.length - 1; j >= i; j--){
            if (elements[j-1] > elements[j]){
                Util.interchange(elements, j-1, j);
                hasChange = true;
            }
        }
        i++;
    }
}
```

- Complexity whether the input would be 8 1 2 3 4 5 6 7?

Best case: $O(n)$
Worst case: $O(n^2)$
Average case: $O(n^2)$

Bubble with sentinel

- Initial vector:

8	1	2	3	4	5	6	7

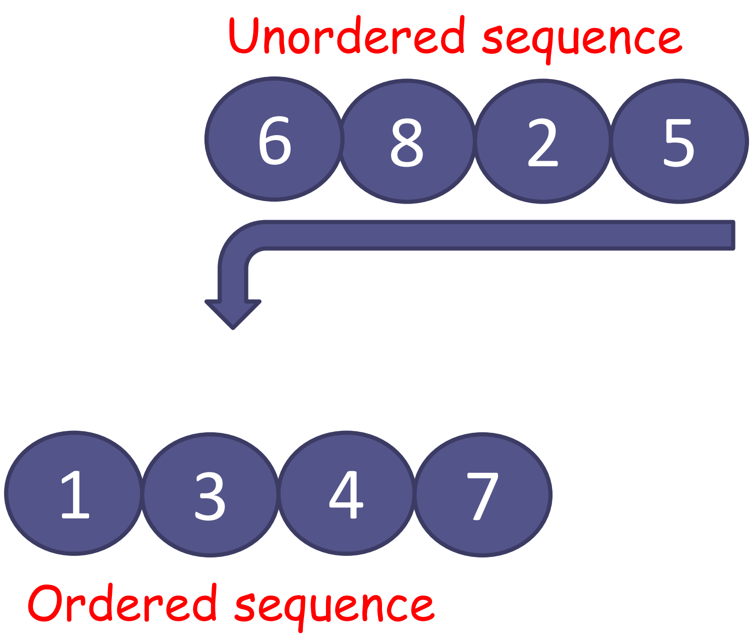
1	1	8	2	3	4	5	6	7
2	1	2	8	3	4	5	6	7
3	1	2	3	8	4	5	6	7
4	1	2	3	4	8	5	6	7
5	1	2	3	4	5	8	6	7
6	1	2	3	4	5	6	8	7
7	1	2	3	4	5	6	7	8

Simple algorithms
Sophisticated algorithms
Constrained algorithms
External algorithms

Sorting by direct exchange
Sorting by insertion
Sorting by selection

Direct insertion

- Description
  - The vector is divided into two "virtual" parts: an ordered and an unordered sequence
  - At each step we take the first element of the unordered sequence and **insert it into the corresponding place** of the ordered sequence



Direct insertion

- Initial vector:

4	5	6	1	3	2	7	8

Ordered vector

1	4	5						
2	4	5	6					
3	1	4	5	6				
4	1	3	4	5	6			
5	1	2	3	4	5	6		
6	1	2	3	4	5	6	7	
7	1	2	3	4	5	6	7	8

Unordered vector

6	1	3	2	7	8		
1	3	2	7	8			
3	2	7	8				
2	7	8					
7	8						
8							

Direct insertion

- Initial vector:

4	5	6	1	3	2	7	8

1	4	5	6	1	3	2	7	8
2	4	5	6	1	3	2	7	8
3	1	4	5	6	3	2	7	8
4	1	3	4	5	6	2	7	8
5	1	2	3	4	5	6	7	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Direct insertion

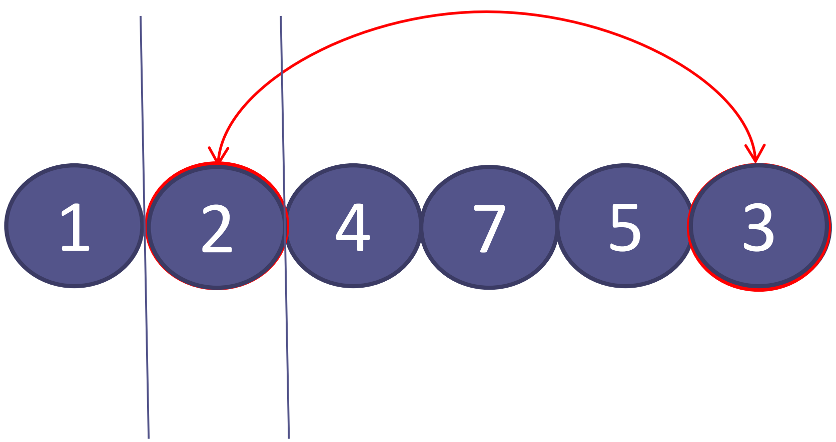
```
public void sort(int[] elements) {
    int j;
    int pivot;

    for (int i =  1; i < elements.length; i++) {
        pivot = elements[i];
        j = i-1;

        while (j >= 0 && pivot < elements[j]) {
            elements[j+1] = elements[j];
            j--;
        }

        elements[j+1] = pivot;
    }
}
```

- High number of exchanges and comparisons
- Less comparisons and exchanges than the bubble method

Best case: $O(n)$
Worst case: $O(n^2)$
Average case: $O(n^2)$

Direct selection

- Description
  - It consists on selecting the smallest element and exchange it with the first element
  - Repeat the process with the remaining elements until there is only one element (the greatest of all)



Direct selection

- Initial vector:

4	5	6	1	3	2	7	8

1	1	5	6	4	3	2	7	8
2	1	2	6	4	3	5	7	8
3	1	2	3	4	6	5	7	8
4	1	2	3	4	6	5	7	8
5	1	2	3	4	5	6	7	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Direct selection

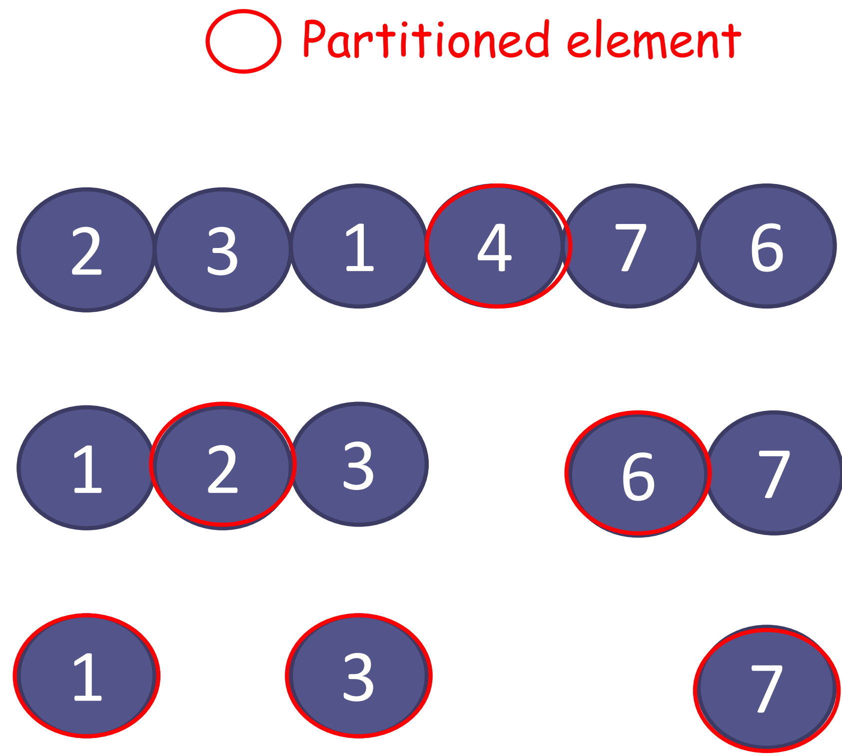
```
public void sort(int[] elements) {
    int posMin;

    for (int i =  0; i < elements.length-1; i++) {
        posMin = Util.findPosMin(elements, i); //O(n)
        Util.interchange(elements, i, posMin);
    }
}
```

- The number of exchanges is very small
- It is predictable: the number of exchanges and comparisons depends on `n`
- The number of comparisons is very high

Best case: $O(n^2)$
Worst case: $O(n^2)$
Average case: $O(n^2)$

QuickSort

- Description
  - It is based on partitioning
  - When you partition an item, that item will be in its corresponding position
    - Then, the idea is to partition all the elements
  - You have to choose a good element to partition (**pivot**) so that it is the median and it creates a tree (if it would be on one corner it will create a list)
  - The implementation is recursive



QuickSort

1	3	8	7	2	4	6	5

- Criteria for choosing a good pivot
  - The median
    - It is the ideal solution but it is very expensive to compute
  - The first element
    - It is "cheap" but it is a bad choice
  - The last element
    - Occurs as with the first element
  - A random element
    - Statistically it is not the worst choice, but has computational cost
  - The central element
    - Statistically it is not a bad choice
  - A compromise solution (median-of-3)
    - We obtain a sample of 3 elements and calculate the median
    - It does not guarantee anything but it can be a good indicator
    - We choose the first element, the last element and the central element
    - We order the elements, and we assume that the median is the element which is in the center

QuickSort

- Idea of the algorithm

REPEAT UNTIL ALL THE ELEMENTS ARE SORTED → O(log n) …O(n)

CHOOSE A PIVOT → Using median-of-3 is O(1) First part

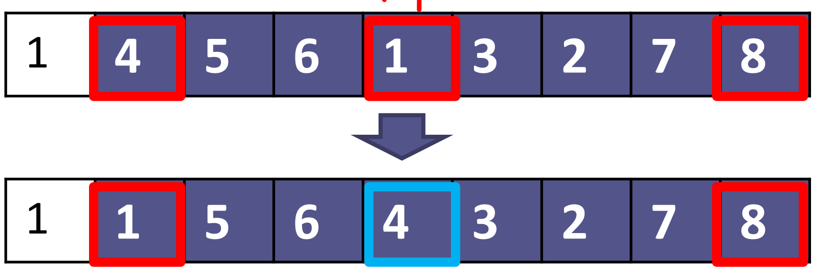
PARTITIONING THE PIVOT THROUGH A PARTITIONING STRATEGY → Typical case O(n) Second part

QuickSort

- Initial vector:

4	5	6	1	3	2	7	8

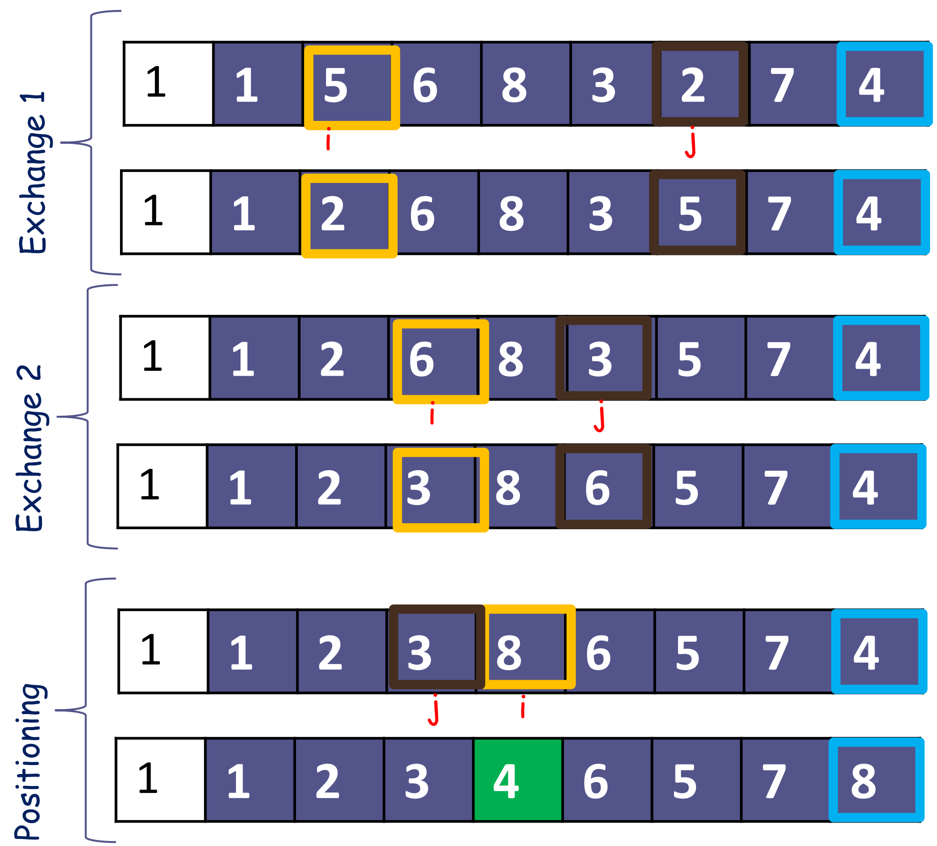
Choice of pivot



Partitioning strategy

Search for an `element[i] > pivot` and an `element[j] < pivot` and interchange them

`i` is moved to the right and `j` is moved to the left displacing elements `element[j] < pivot` to the left and elements `element[i] > pivot` to the right (Ends when `j` and `i` se are crossed, determining `i` the final position)



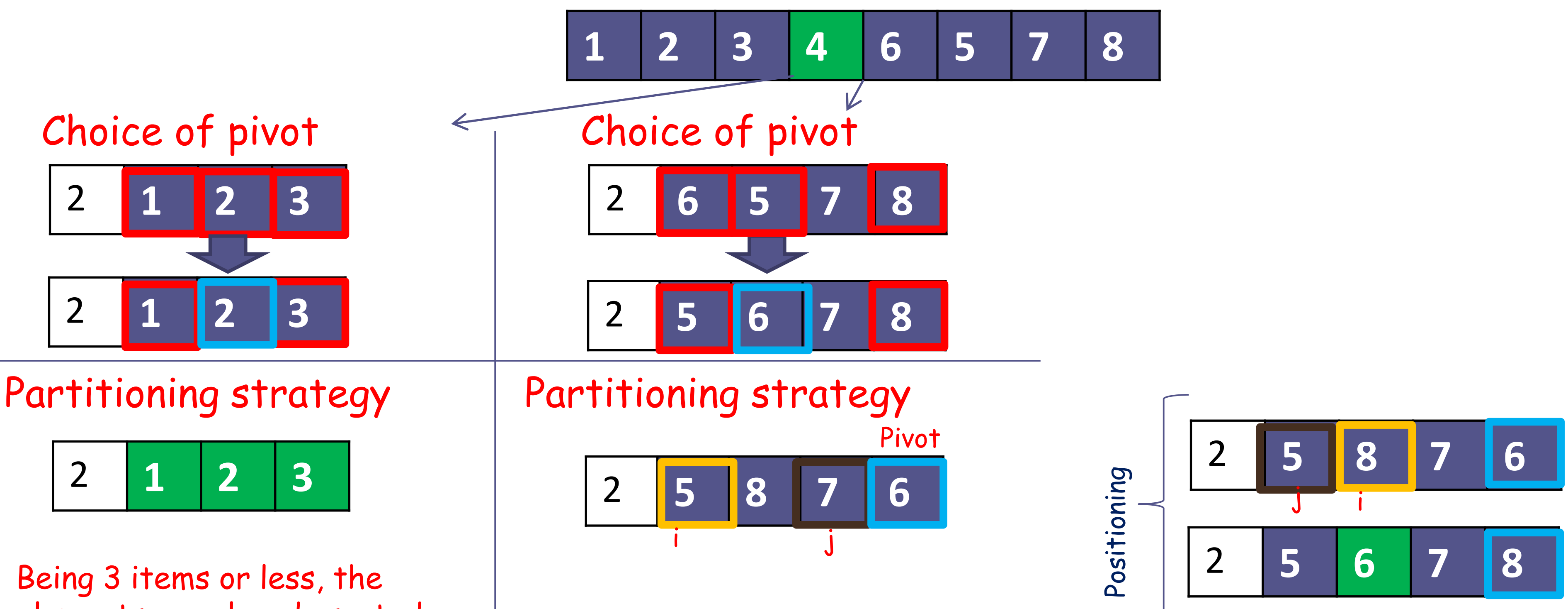
Simple algorithms
Sophisticated algorithms
Constrained algorithms
External algorithms

Sorting by direct exchange
Sorting by insertion
Sorting by selection

QuickSort

- Initial vector:

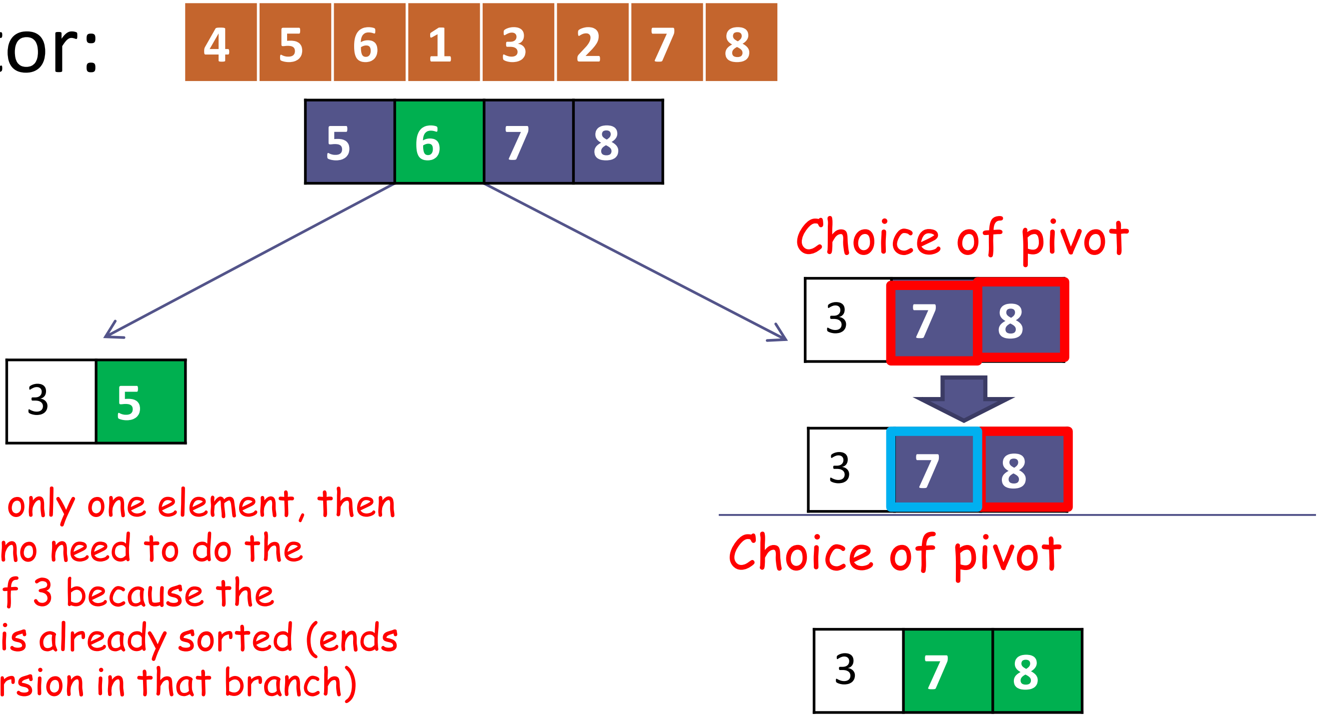
4	5	6	1	3	2	7	8



Being 3 items or less, the elements are already sorted when searching the median of 3 (ends the recursion in that branch)

QuickSort

- Initial vector:



There is only one element, then there is no need to do the median of 3 because the element is already sorted (ends the recursion in that branch)

Being 3 items or less, the elements are already sorted when searching the median of 3 (ends the recursion in that branch)

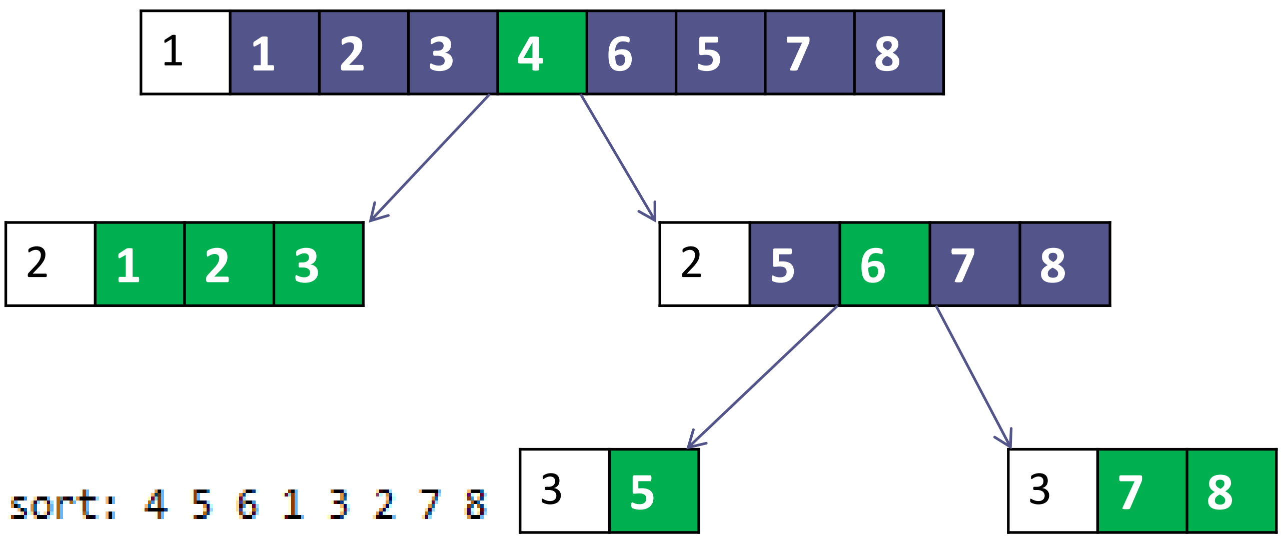
Simple algorithms
Sophisticated algorithms
Constrained algorithms
External algorithms

Sorting by direct exchange
Sorting by insertion
Sorting by selection

QuickSort

- Recursive calls performed

4	5	6	1	3	2	7	8




```
Values before the sort: 4 5 6 1 3 2 7 8
Level: 1 - 1 2 3 4 6 5 7 8
Level: 2 - 1 2 3 4 6 5 7 8
Level: 2 - 1 2 3 4 5 6 7 8
Level: 3 - 1 2 3 4 5 6 7 8
Level: 3 - 1 2 3 4 5 6 7 8
Values after the sort: 1 2 3 4 5 6 7 8
```

Sorting algorithms

Simple algorithms
Sophisticated algorithms
Constrained algorithms
External algorithms

Sorting by direct exchange
Sorting by insertion
Sorting by selection

QuickSort

```java
public class Quicksort implements ISortingAlgorithm{
    @Override
    public void sort(int[] elements) {
        quickSort(elements, 0, elements.length-1, 1);
    }

    /*get the position of the median of the three (left, right and
     the element which position is in the center between them, and
     move the elements to order them */
    private int median_of_three(int elements[], int left, int right){
        int center = (left + right) / 2;
        if (elements[left] > elements[center])
            Util.interchange(elements, left, center);
        if (elements[left] > elements[right])
            Util.interchange(elements, left, right);
        if (elements[center] > elements[right])
            Util.interchange(elements, center, right);
        return center;
    }
```

```java
private static void quickSort(int elements[], int left, int right){
    int i = left;
    int j = right - 1;
    int pivot;

    if (left < right){ //if there is one element it is not necessary
        int center = median_of_three(elements, left, right);
        //if there are less than or equal to 3 elements, there are just ordered
        if ((right - left) >= 3){
            pivot = elements[center]; //choose the pivot
            Util.interchange(elements, center, right); //hide the pivot

            do {
                while (elements[i] <= pivot && i < right) i++; //first element > pivot
                while (elements[j] >= pivot && j > left) j--; //first element < pivot
                if (i < j) Util.interchange(elements, i, j);
            } while (i < j);   //end while

            //we set the position of the pivot
            Util.interchange(elements, i, right);
            quickSort(elements, left, i-1);
            quickSort(elements, i+1, right);
        } //if
    } //if
}
```

- Very fast for values of n > 20
- Optimizable distributing the processing of each partition in different threads in parallel

Best case: O(n logn)
Worst case: $O(n^2)$
Average case: O(n logn)

Comparison of algorithms

n = 256	Sorted	Random	Inverse
Bubble	540	1026	1492
Bubble with sent.	5	1104	1645
Bidirect. bubble	5	961	1619
Direct insertion	12	366	704
Binary insertion	56	373	662
Direct selection	489	509	695
ShellSort	58	127	157
HeapSort	116	110	104
QuickSort	31	60	37

n = 512	Sorted	Random	Inverse
Bubble	2165	4054	5931
Bubble with sent.	8	4270	6542
Bidirect. bubble	9	3642	6520
Direct insertion	23	1444	2836
Binary insertion	125	1327	2490
Direct selection	1907	1956	2675
ShellSort	116	349	492
HeapSort	253	241	226
QuickSort	69	146	79

External algorithms

- They are based on the principle of DIVIDE AND CONQUER
- They use external memory
  - The slowest of them is like Quicksort
- They use the same basic principles that internal sorting algorithms
- Different algorithms:
  - Direct mixture
  - Natural mixture
  - Balanced mixture
  - Polyphasic mixture