Unit 2

The Divide-and-Conquer Strategy (D&C)

1. **The D&C approach**: Why (reasons)? What (definition and general scheme)? When is it appropriate to use it ("cost cookbook")?
2. **D&C solutions to Fast-Sorting and Fast-Selection**: **Which are the best D&C solutions to the Sorting problem?** What about the Selection problem?
   - **Lab. 2** Effciency's empirical analysis of two sorting D&C algorithms
3. **Reduce-and-Conquer Exercises** : Why not D&C exercises?

Bibliography

- Weiss, M.A. *Data Structures and Problem Solving Using Java, 4th Edition.* Adisson-Wesley, 2010. Chapter 8, sections 5, 6 and 7

- Galiano I. and Prieto N. *Notes from the "Estructuras de Datos y Algoritmos" course*. Previous Computer Science Curriculum. Available in PoliformaT
  - Apuntes - Diseño recursivo y eficiente: soluciones Divide y Vencerás para la Ordenacioón y la Selección

1. The D&C approach

***Why** (reasons)? Recursion: the bad, the good and the fair*

Recursion is a powerful problem-solving approach that yields compact but readable, elegant and very efficient algorithms

The Sorting Problem:

Recursion ($O(x \log x)$) **VS** Iteration ($O(x^2)$)

Divide and Conquer

Too much recursion can be dangerous!

The Fibonacci numbers:
Recursion ($O(2^x)$) **VS** Iteration ($O(x)$)

MORAL

Use recursion ONLY to solve problems complex enough to deserve it!

1. The D&C approach

What? *Definition*

The D&C Strategy involves three steps at each level of the recursion

- DIVIDE the problem of size **x** into a number of subproblems **a** that are smaller instances of the same problem (**a > 1**)

  WARNING: at least two **DISJOINT** subproblems

  TIPS: The size of the subproblems **should** reduce the size of the original problem …

  - **GEOMETRICALLY**, or by the same constant factor **c**: **x / c**
  - In the most **BALANCED** way possible: **a = c**

- CONQUER the subproblems by solving them recursively, **except, of course, the base cases**
- COMBINE the solutions to the subproblems into the solution for the original problem

1. The D&C approach

What? General scheme and its Recurrence equation

```
public static ResultType conquer(DataType x) {
    ResultType method_res, call_1_res, ..., call_a _res;
    if (x == x_base) { method_res = baseCaseSolution(x); }
    else {
        int c = divide(x);
        call_1_res = conquer(x / c);
        ...
        call_a _res = conquer(x / c);

        method_res = combine(x, call_1_res, ..., call_a _res);
    }
    return method_res;
}
```

But... Can a Strategy Recurrence Equation be solved? How?

Recurrence Equation for the general (recursive) case

$$T_{conquer}(x > x_{base}) = \underset{\substack{\uparrow \\ \text{Number of} \\ \text{recursive calls}}}{a} * T_{conquer}(x \underset{\substack{\uparrow \\ \text{x decreases} \\ \text{geometrically}}}{/} c) + \underbrace{T_{divide}(x) + T_{combine}(x)}_{\text{Call overhead}}$$

1. Divide & Conquer

When is it appropriate to use it ? "Cost Cookbook"

Theorem 1: $T_{recursiveMethod}(x) = a \cdot T_{recursiveMethod}(x-c) + b, \; b \geq 1$

- If a=1, $T_{recursiveMethod}(x) \in \Theta(x)$
- If a>1, $T_{recursiveMethod}(x) \in \Theta(a^{x/c})$

Theorem 2: $T_{recursiveMethod}(x) = a \cdot T_{recursiveMethod}(x-c) + b \cdot x + d$, b and d≥1

- If a=1, $T_{recursiveMethod}(x) \in \Theta(x^2)$
- If a>1, $T_{recursiveMethod}(x) \in \Theta(a^{x/c})$

Theorem 3: $T_{recursiveMethod}(x) = a \cdot T_{recursiveMethod}(x/c) + b, \; b \geq 1$

- If a=1, $T_{recursiveMethod}(x) \in \Theta(\log_c x)$ **Reduce & Conquer** (Binary Search)
- If a>1, $T_{recursiveMethod}(x) \in \Theta(x^{\log_c a})$

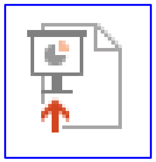
Theorem 4: $T_{recursiveMethod}(x) = a \cdot T_{recursiveMethod}(x/c) + b \cdot x + d$, b and d≥1

- If a<c, $T_{recursiveMethod}(x) \in \Theta(x)$
- If a=c, $T_{recursiveMethod}(x) \in \Theta(x \cdot \log_c x)$ **Divide & Conquer** (Fast Sorting)
- If a>c, $T_{recursiveMethod}(x) \in \Theta(x^{\log_c a})$

$$T_{conquer}(x > x_{base}) = a * T_{conquer}(x / c) + T_{divide}(x) + T_{combine}(x)$$

1. Divide & Conquer

How to use the "Cost Cookbook"? *Examples*



To review what you learned last year about the analysis of recursive methods and, at the same time, to know how to use the "cost cookbook", the slide-show at left shows some examples

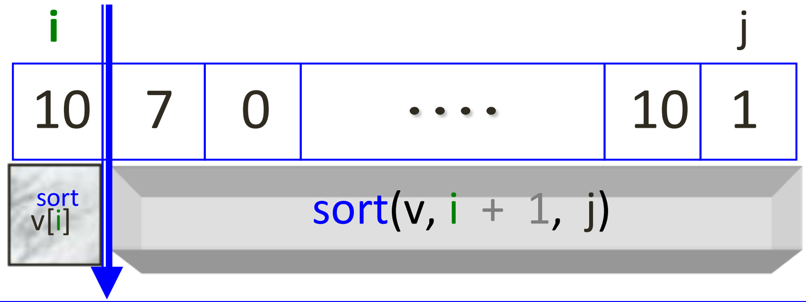
2. D&C solutions to Fast-Sorting

Two recursive approaches to Sorting

Restriction: sorting one element out of the total takes linear time in both the worst and the average cases

"Conservative" approach

sort(v, i, j) =

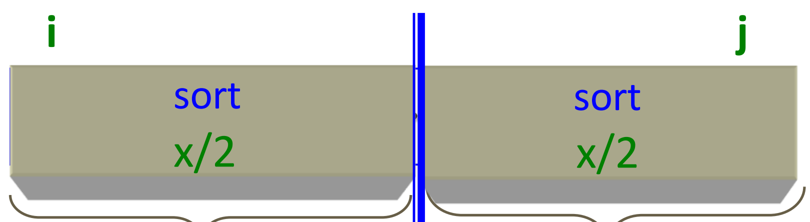


Average cost: by **T2** (Linear Overhead) with c = a = 1, $T^{\mu}_{sort}(x) \in \Theta(x^2)$

D&C approach?

IFF $T_{Divide}(x) + T_{Combine}(x) = k \cdot x$ **&&** sp1 size ≈ sp2 size (c = a)

Divide "properly" the original problem of size x (sort on x)



Conquer sp1 on x/2 && sp2 on x/2

Combine "properly" the sp1 & sp2 solutions into the solution to sort on x

Average cost: by **T4** (Linear Overhead) with c = a (= 2), $T^{\mu}_{sort}(x) \in \Theta(x \cdot \log x)$

2. D&C solutions to Fast-Sorting

Merge Sort: D&C approach and its analysis

IFF $T_{Divide}(x) + T_{Combine}(x) = k \cdot x$ **&&** sp1 size ≈ sp2 size ($c = a$)

By **T4** (LinearOverhead) with $c = a = 2$, $T^{\mu}_{sort}(x) \in \Theta(x \cdot \log x)$

Divide "properly" the original problem of size x

$m = (i + j) / 2$; $T_{Divide}(x) \in \Theta(1)$

i j

mergeSort	mergeSort
v[i, m]	v[m + 1, j]

Conquer sp1 on x/2 Conquer sp2 on x/2

v[i, m] y v[m+1, j] sorted

Combine "properly" the sp1 & sp2 solutions into the solution to sort on x

Sort v[i, j] by ***merging*** the already sorted v[i, m] and v[m+1, j]: $T_{Combine}(x) \in \Theta(x)$

```
public static <T extends Comparable<T>> void merge(T[] v, int i, int f, int m) {
    int a = i, b = m + 1, k = 0; T[] aux = (T[]) new Comparable[f - i + 1];
     while (a <= m && b <= f) {
         if (v[a].compareTo(v[b]) < 0) { aux[k++] = v[a++]; }
         else                          { aux[k++] = v[b++]; }
     }
     while (a <= m) { aux[k++] = v[a++]; }
     while (b <= f) { aux[k++] = v[b++]; }
     for (a = i, k = 0; a <= f; a++, k++) { v[a] = aux[k]; }
}
```

2. D&C solutions to Fast-Sorting

Merge Sort: D&C approach and its analysis

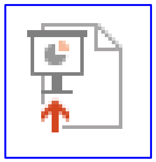
```
private static <T extends Comparable <T>> void mergeSort(T[] v, int i, int j) {
    if (i < j) {
        int m = (i + j) / 2;            // DIVIDE
        mergeSort(v, i, m);             // CONQUER
        mergeSort(v, m + 1, j);         // CONQUER
        merge(v, i, j, m);              // COMBINE
    }
}

public static <T extends Comparable<T>> void mergeSort(T v[]) {
    mergeSort(v, 0, v.length - 1);
}
```

As expected, $T_{\texttt{mergeSort}}(x) \in \Theta(x \cdot \log x)$ by T4:

$$T_{\texttt{conquer}}(x > 1) = \underset{\substack{\downarrow \\ a = 2}}{a} * T_{\texttt{conquer}}(x/\underset{\substack{\downarrow \\ c = 2}}{c}) + \underbrace{T_{\texttt{divide}}(x) + T_{\texttt{combine}}(x)}_{\Theta(x)}$$

2. D&C solutions to Fast-Sorting

How does Merge Sort work? *Unrolling the recursion*



To understand `mergeSort` method, it is worthwhile to consider carefully the dynamics of the method calls. The slide-show at left shows this dynamics “at the click of a mouse” -by tracing the call to its driver method with the array v = {5, 2, 4, 6, 1, 3, 8, 7} as argument

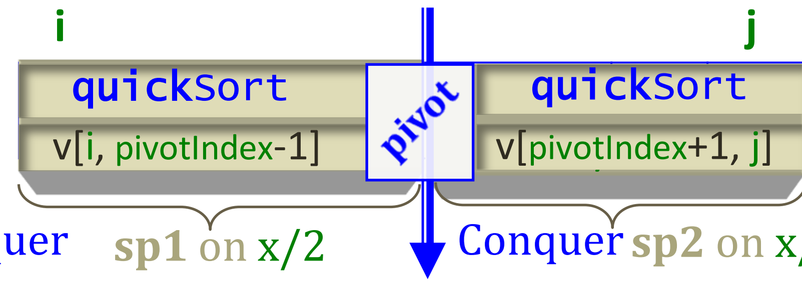
2. D&C solutions to Fast-Sorting

Quick Sort: D&C approach

IFF $T_{Divide}(x) + T_{Combine}(x) = k \cdot x$ **&&** sp1 size ≈ sp2 size (c = a)

By **T4** (LinearOverhead) with c = a = 2, $T^{\mu}_{sort}(x) \in \Theta(x \cdot \log x)$

Divide "properly" the original problem of size x



Conquer sp1 on x/2 Conquer sp2 on x/2

Combine "properly" the sp1 & sp2 solutions into the solution to sort on x

Choose 1 element in v[i, j] **"well"** –the **pivot**- and sort it **by Exchange?** $T_{Divide}(x) \in \Theta(x)$

v[i, pivotIndex - 1] &
v[pivotIndex + 1, j] sort**ed**

As v[pivotIndex] is sort**ed** at Divide step, v[i, j] is ALREADY sort**ed** and **NO Combine** step is required: $T_{Combine}(x) \in \Theta(1)$

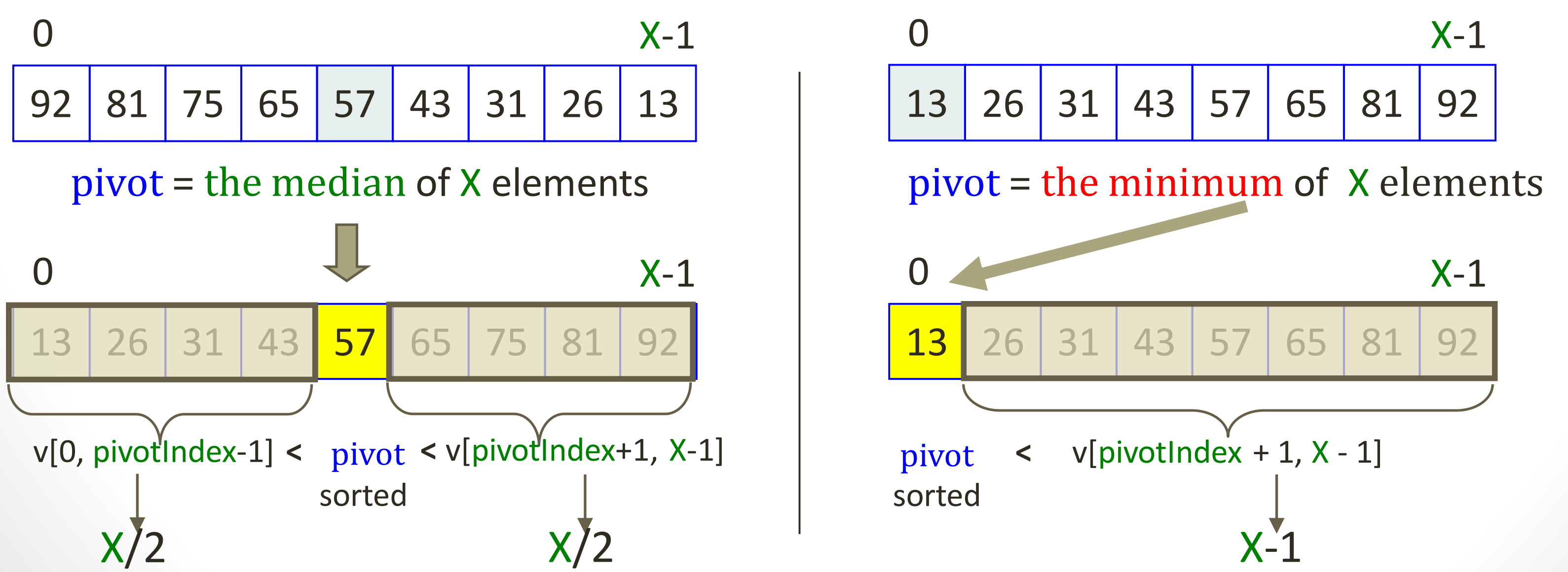
2. D&C solutions to Fast-Sorting

*Quick Sort: **pros & cons** of sorting an element by Exchange (as in Bubble Sort!)*

Sorting 1 element (pivot) of an array of size X by Exchange ...

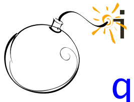
- **PARTITIONS** (rearranges) **the array into 2 subarrays** (Divide) **SUCH THAT** all elements in the *left* subarray (v[0, pivotIndex–1]) are less than or equal to the pivot **AND** all elements in the *right* subarray (v[pivotIndex+1, X-1]) are greater than or equal to the pivot
- **Takes $\Theta(X)$ time** ($T_{Divide} \in \Theta(X)$)**:** Each element in the array is compared once with the pivot (X comparisons) and it may or may not be exchanged (at most, X/2 exchanges)

The pivot index depends on the relative ordering of the *values* of the elements to be sorted, hence it **determines HOW WELL the partitioning divides the array**



2. D&C solutions to Fast-Sorting

Quick Sort: D&C approach and its analysis



```
private static <T extends Comparable <T>> void quickSort(T[] v, int i, int d) {
    if (i < d) {
        int pivotIndex = particion(v, i, d);   // DIVIDE in Θ(x) time
        quickSort(v, i, pivotIndex - 1);       // CONQUER
        quickSort(v, pivotIndex + 1, d);       // CONQEER
        // NO COMBINE step is required!
    }
}
public static <T extends Comparable<T>> void quickSort(T[] v) {
    quickSort(v, 0, v.length - 1);
}
```

$$T_{conquer}(x > 1) = a * T_{conquer}(x/c) + T_{Divide}(x) + T_{Combine}(x)$$

$a = 2$ $c = 2?$ $\Theta(x)$

Depending on the pivot index, the partitioning can be …

- **Perfectly balanced** (pivot = median): **c=2** ➔ By T4, $T_{quickSort}(x) \in \Omega(x \cdot \log x)$
- **Perfectly Unbalanced** (pivot = minimum): **c=1** ➔ By T2, $T_{quickSort}(x) \in O(x^2)$**!**

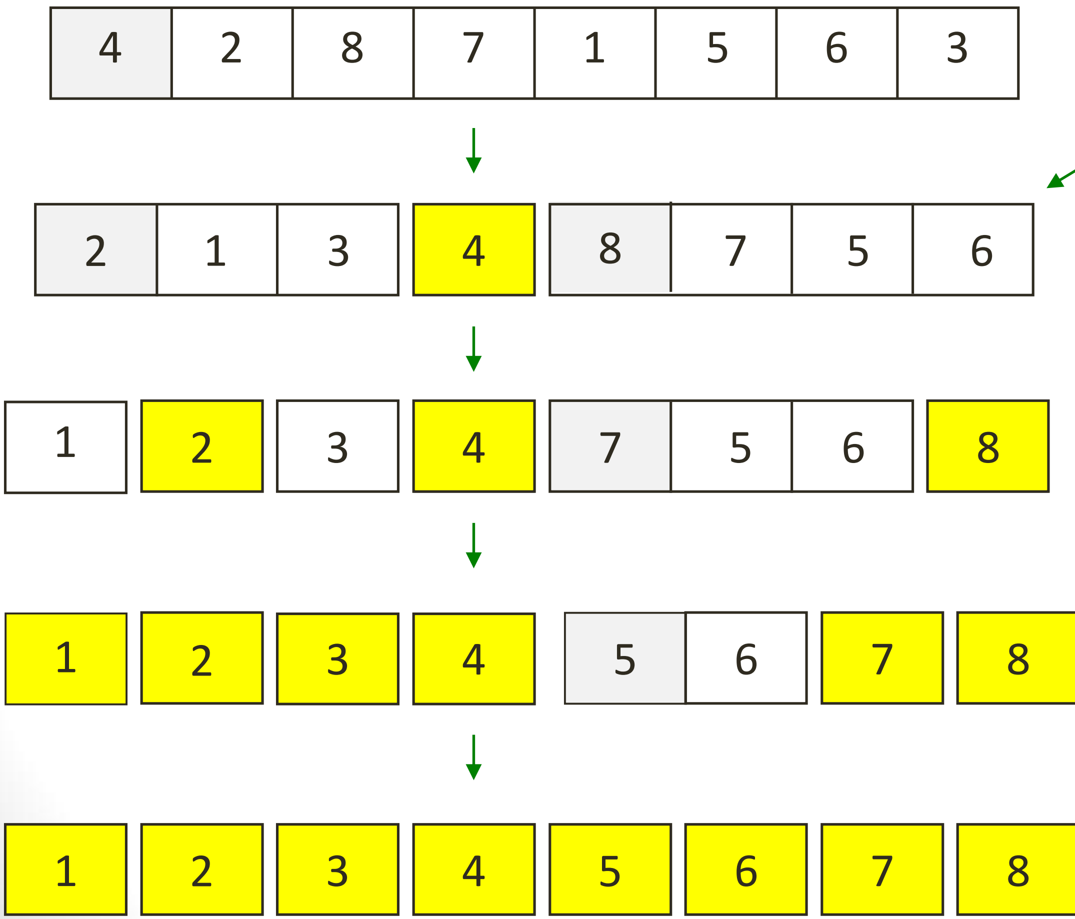
GOOD NEWS: it can be proved, that if we assume the uniform distribution of all the possible input permutations, $T_{quickSort}^{\mu}(x) \in O(x \cdot \log x)$**!!**

2. D&C solutions to Fast-Sorting

How does Quick Sort work? *A bird's-eye view*

chosen pivot = 1st subarray element

sorted pivot



As shown in this trace, the left subarray is sorted faster than the right one, because it ALWAYS divides the array into two equal-size subarrays, whereas the rigth one ALWAYS does exactly the opposite

2. D&C solutions to Fast-Sorting

How does Quick Sort work? A bird's-eye view

chosen pivot = median **of 3** (1st, middle & last)

sorted pivot

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

↓

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

↓

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

↓

1	2	3	4	5	6	7	8

The trace shows now that both the left and right subarrays are sorted at the same speed, thanks to the current pivot choice

Wouldn't it be better to choose the median of each subarray as the pivot?

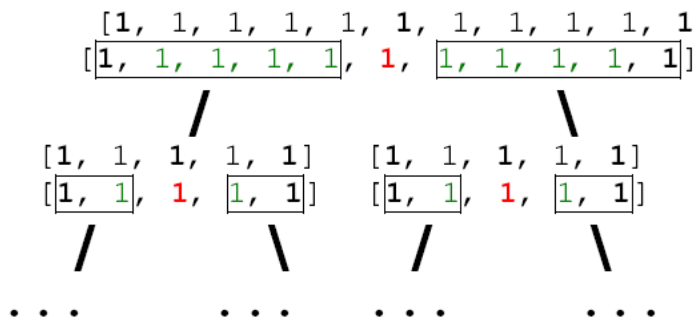
NO, it wouldn't: Calculating the median would **SLOW** the sorting **SUBSTANTIALLY!**

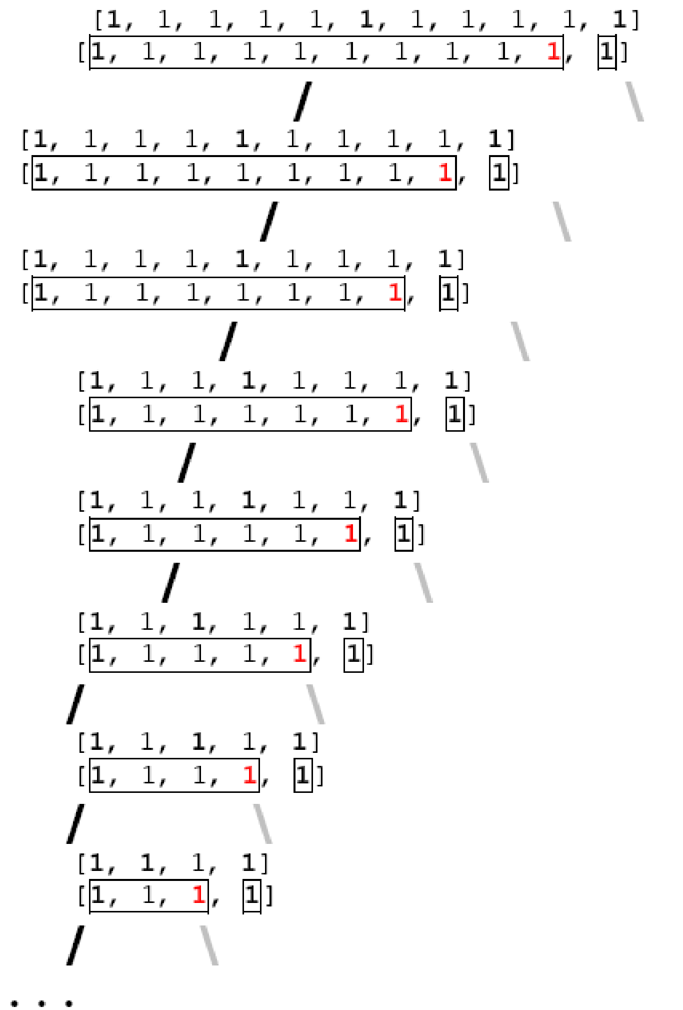
Instead, the median of three gives a **CHEAP** and good-enough estimate of the median

2. D&C solutions to Fast-Sorting

How does Quick Sort work? *What do we do if we see an element that is equal to the pivot?*

TO STOP **OR** NOT TO STOP





If you want to sort an array of 1.000.000 elements (instead of 11), 3.000 out of which are identical (instead of 11), it will not be a miracle that, sooner than later, a recursive call happens on only these 3.000 identical elements…

And if that is the case, you need to ensure that the partition is still perfectly balanced

2. D&C solutions to Fast-Sorting

How does Quick Sort work? An optimized version of the partitioning procedure $(T^{\mu}_{quickSort}(x) \in (x \cdot \log x))$

```
private static <T extends Comparable <T>> void quickSort(T[] v, int i, int d) {
    if (i < d) {
     // DIVIDE, or PARTITION, v[i, d]: sort the pivot by Exchange
        T pivot = medianaDe3(v, i, d) // pIndex = (i + d) / 2; v[i] & v[d] sorted
        intercambiar(v, (i + d) / 2, d - 1); // "hides" the pivot at d - 1
        int pIndex = i, j = d - 1;
        for (; pIndex < j;) {
            while (v[++pIndex].compareTo(pivot) < 0) { ; }
            while (v[--j].compareTo(pivot)      > 0) { ; }
            intercambiar(v, pIndex, j);
        }
        intercambiar(v, pIndex, j); // undoes the last exchange performed
        intercambiar(v, pIndex, d - 1); // restores the pivot to its proper position
        // CONQUER sp1:
        quickSort(v, i, pIndex - 1);
        // CONQUER sp2:
        quickSort(v, pIndex + 1, d);
    }
}
```

2. D&C solutions to Fast-Selection

Quick Select: D&C approach and its implementation

A problem closely related to sorting is **selection**, or finding the *k*th smallest element in an array of x elements. Obviously, we can sort the elements …

- Using selectionSort: $T_{kthSmallest} \in \Theta(k \cdot x)$ and $T_{kthSmallest} \in O(x^2)$
- Using quickSort or mergeSort: $T^{\mu}_{kthSmallest} \in O(x \cdot \log x)$ time

Can a D&C strategy be devised to solve this problem more efficiently?

```
/** Places the kth smallest element in v[k-1]
 *  Uses the method particion of quickSort */
private static <T extends Comparable <T>> void seleccionRapida(T[] v, int k, int i, int d) {
    if (i < d) {
        int pIndex = particion(v, i, d);
        if (k - 1 < pIndex ) { seleccionRapida(v, k, i, pIndex - 1); }
        else if (k - 1 > pIndex) { seleccionRapida(v, k, pIndex + 1, d); }
        // else, if pIndex = k - 1 the pivot is the kth smallest element!
    }
}
/** Returns the kth smallest element (v[k-1]) in the array v */
public static <T extends Comparable<T>> T seleccionRapida(T[] v, int k) {
    seleccionRapida(v, k, 0, v.length - 1);
    return v[k - 1];
}
```

2. D&C solutions to Fast-Sorting

How does Quick Select work? A bird's-eye view

Tracing the first call to `seleccionRapida` with k = 1 and v = {51, 77, 15, 0, 86, 82, 51, 23, 34, 38, 8} as arguments

```
[51, 77, 15, 0, 86, 82, 51, 23, 34, 38,  8]
[8 , 77, 15, 0, 86, 51, 51, 23, 34, 38, 82]
[8 , 77, 15, 0, 86, 38, 51, 23, 34, 51, 82]
[8 , 34, 15, 0, 23, 38, 51, 86, 77, 51, 82]

[8 , 34, 15, 0, 23, 38 | 51, 86, 77, 51, 82]
[8 , 34, 15, 0, 23, 38, 51, 86, 77, 51, 82]
[8 , 34, 23, 0, 15, 38, 51, 86, 77, 51, 82]
[8 ,  0, 15,34, 23, 38, 51, 86, 77, 51, 82]

[8 ,  0 | 15, 0, 23, 38, 51, 86, 77, 51, 82]
[0 ,  8, 15, 0, 23, 38, 51, 86, 77, 51, 82]
[0 ,  8, 15, 0, 23, 38, 51, 86, 77, 51, 82]
```

Hands-On Exercise 1: Analyze the `seleccionRapida` method

3. Reduce-and-Conquer (R&C) Exercises

Why not D&C exercises?

IFF $T_{Divide}(x) + T_{Combine}(x) = k$ **&&** $a = 1$ (Linear Recursion)

By **T3** (ConstantOverhead) with $a = 1$, $T^{\mu}_{conquer}(x) \in \Theta(\log_c x)$

The Iconic Example of R&C: Binary Search (1D)

A Binary Search Sequel: 2D Binary Search

Hands-On Exercise 2: Cross Point of a monotonically increasing function (two examples in the below figure)

0	1	2	3	4	5
-3	-2	-1	1	4	5

−3	−2	−1	0	1	4	5