Topic 2

Divide & Conquer

Sorting and Selecting

Aim

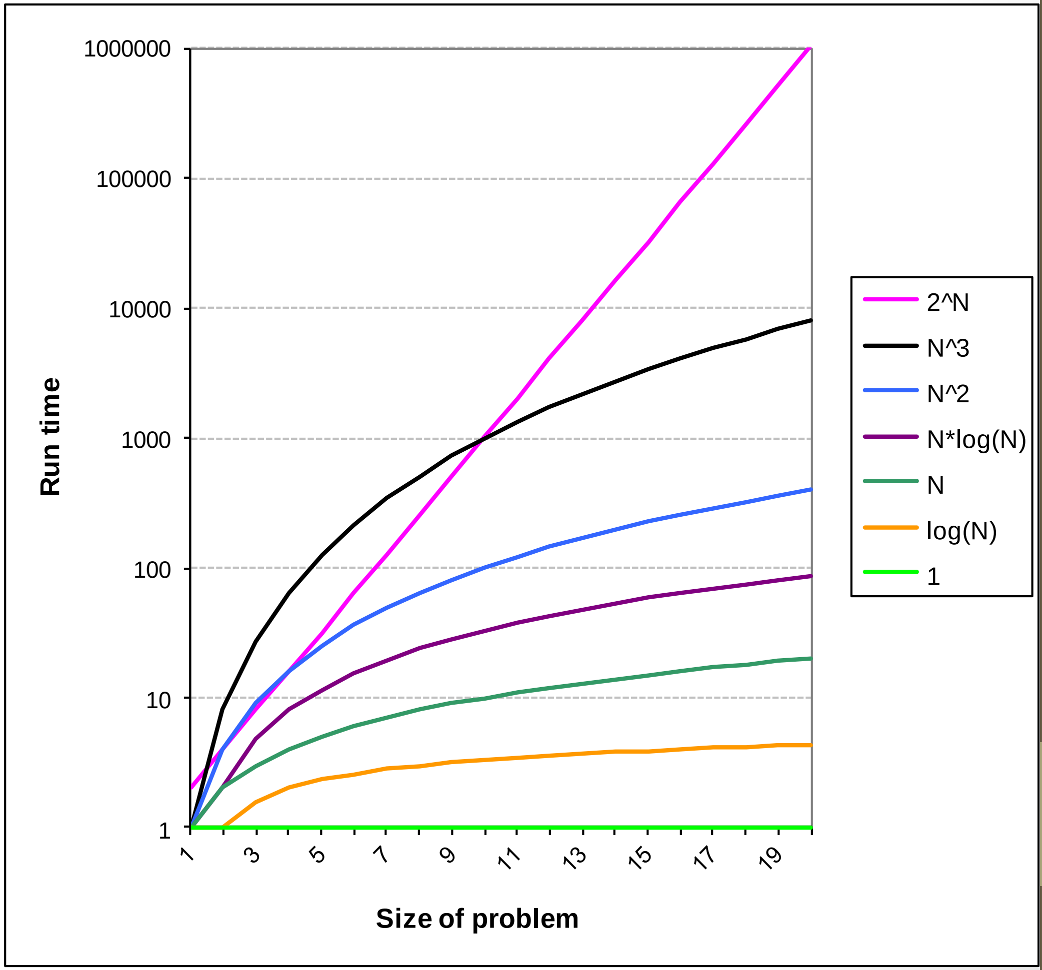
- The general aim is to present recursivity as a design tool, alternative to the iterative approach:
  - To study time complexity of the recursive methods via recurrence relations
  - To introduce the recursive strategy *Divide & Conquer* (D&C) and its application in methods such as *MergeSort*, *QuickSort* and *QuickSelect*.

Contents (4 sessions approx.)

1. Analysis of complexity
   1.1. Complexity of a recursive method: recurrence relations
2. Divide & Conquer
   2.1. Generic schema
   2.2. MergeSort
   2.3. QuickSort
   2.4. QuickSelect

1. Analysis of complexity
Asymptotical bounds

Name	Asymptotical notation
exponential	$\Theta(2^{size})$
cubic	$\Theta(size^3)$
quadratic	$\Theta(size^2)$
linear	$\Theta(size)$
logarithmic	$\Theta(\log size)$
constant	$\Theta(1)$



1. Analysis of complexity

1.1. Complexity of a recursive method

```
static int factorial(int N) {
  if (N < 1) return 1;                    // Base case
  else return N * factorial(N-1);         // General case
}
```

$T_{factorial}(N = 0) = k$

$T_{factorial}(N > 0) = k + T_{factorial}(N - 1) = k + k + T_{factorial}(N - 2) = \ldots$

$= k + k + \ldots + k + T_{factorial}(0) = k * N + k$

$\Rightarrow T_{factorial}(N) \in \Theta(N)$

- What is its space complexity? And the one of the iterative version? Therefore, what is the most efficient one?

1. Analysis of complexity
1.1. Recurrence relations (1/3)

- The complexity of a recursive method depends on:
  - The number of recursive invocations
  - The way the size of the problems decreases
  - The complexity of the calculations in each invocation
- ***Recurrence relations*** allow to obtain the time complexity of a method on the basis of these three parameters

1. Analysis of complexity

1.1. Recurrence relations (2/3)

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

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

Example:

```
private static <T> void reverse(T v[], int theBegin, int theEnd)
{
  if (theBegin < theEnd{
       T tmp = v[theBegin];
       v[theBegin] = v[theEnd];
       v[theEnd] = tmp;
       reverse(v, theBegin + 1, theEnd - 1);
  }
}
```

$a = 1, c = 2 \Rightarrow T_{reverse}(x) \in \Theta(x)$

1. Analysis of complexity
1.1. Recurrence relations (3/3)

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

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

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

- If $a = 1$, $T_{recMethod}(x) \in \Theta(\log_c x)$
- If $a > 1$, $T_{recMethod}(x) \in \Theta(x^{\log_c a})$

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

- If $a < c$, $T_{recMethod}(x) \in \Theta(x)$
- If $a = c$, $T_{recMethod}(x) \in \Theta(x \cdot \log_c x)$
- If $a > c$, $T_{recMethod}(x) \in \Theta(x^{\log_c a})$

2. Divide & Conquer

2.1. Introduction

- Complexity of multiple recursivity is greater than of linear one but is size is decreased in a geometric way in each invocation, it could be very efficient.
  - Divide & Conquer (D&C) technique is based on on this idea
- D&C technique is based on the following steps:
  - <u>DIVIDE</u>: a problem of size x is divided in $N > 1$ disjoint subproblems, with the size of the subproblems the most similar as possible
  - <u>CONQUER</u>: solve recursively each subproblem
  - <u>COMBINE</u>: combine the solutions of the subproblems in order to obtain the solution of the original problem

2. Divide & Conquer

2.1. Generic schema

```
public static TypeResult conquer( TypeData x ) {
    TypeResult resMethod, resInvoke_1,..., resInvoke_a;
  if ( baseCase(x) ) resMethod = solutionBase(x);
  else {
       int c = divide(x);
       resInvoke_1 = conquer(x / c);
       ...
       resInvoke_a = conquer(x / c);
       resMethod = combine(x, resInvoke_1,...resInvoke_a);
  }
  return resMethod;
}
```

- Recurrence relation:

$$T_{conquer}(x > x_{base}) = \underset{\uparrow \text{ number of recursive invocations}}{a} * T_{conquer}(\underset{\uparrow \text{ decreased size}}{x/c}) + \underbrace{T_{divide}(x) + T_{combine}(x)}_{\text{overloading in each invocation}}$$

Complexity as function of:

2. Divide & Conquer
2.1. Sorting an array

- The easiest sorting algorithms (*InsertionSort, SelectionSort* and *bubbleSort*) have a quadratic complexity
- The methods *QuickSort* y *MergeSort* employ the D&C strategy in order to improve the efficiency:
  - The original problem is divided into subproblems (a=2) whose size is approximately the half of the original one (c=2)
  - Divide and combine has a linear complexity
  - The complexity of both algorithms is $\Theta(x*\log_2 x)$

2. Divide & Conquer

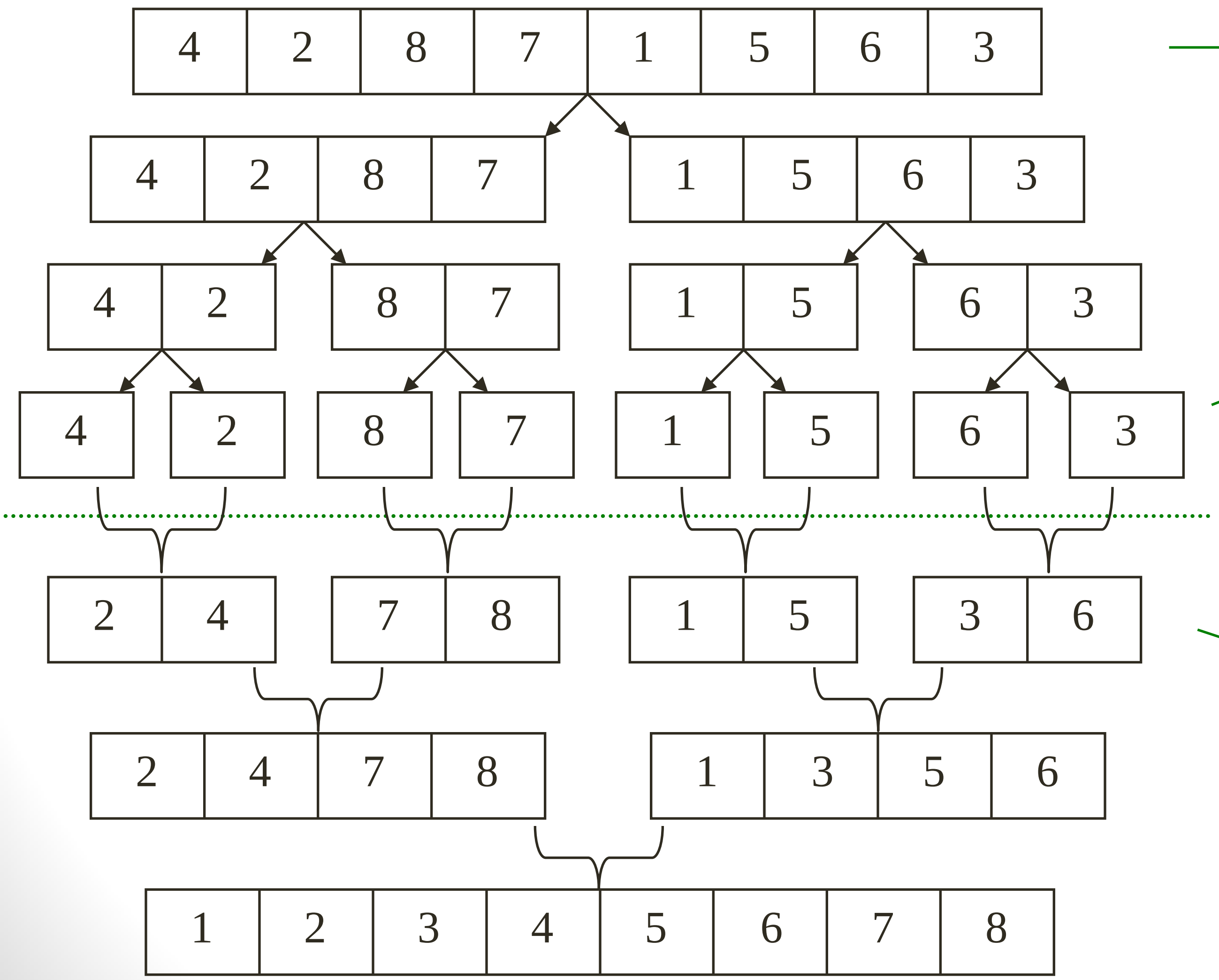
2.2. MergeSort

- **Merge**:
  - Given two arrays in ascending order (`a` and `b`)
  - Merge returns a new array, also in ascending order, that contains the elements of `a` and `b`

```
public static <T extends Comparable<T>>
  T[] merge(T[] a, T[] b) {
  T[] res = (T[]) new Comparable[a.length + b.length];
  int i = 0, j = 0, k = 0;
  while (i < a.length && j < b.length) {
    if (a[i].compareTo(b[j]) < 0) res[k++] = a[i++];
    else res[k++] = b[j++];
  }
  for (int r = i; r < a.length; r++) res[k++] = a[r];
  for (int r = j; r < b.length; r++) res[k++] = b[r];
  return res;
}
```

2. Divide & Conquer

2.2 MergeSort



The array is divided into subarrays of equal size

Subarrays of one element are already sorted

The sorted subarrays are combined (e.g. merged) with the method **merge**

2. Divide & Conquer

2.2. MergeSort

```
private static <T extends Comparable <T>>
    void mergeSort(T[] v, int left, int right) {
    if (left < right) {
        int middle = (left + right) / 2;              // DIVIDE
        mergeSort(v, left, middle);                   // CONQUER
        mergeSort(v, middle + 1, right);              // CONQUER
        Merge(v, left, middle + 1, right);            // COMBINE
    }
}
```

The method is modified in order to receive one array and not two

- The complexity of a method D&C is:

$$T_{conquer}(x > x_{base}) = \underset{a=2}{a} * T_{conquer}(x/\underset{c=2}{c}) + \underbrace{T_{divide}(x) + T_{combine}(x)}_{\Theta(x)}$$

2. Divide & Conquer

2.3. QuickSort

- Given an array `v`:

4	2	8	7	1	5	6	3

- <u>Step 1</u>: an element of the array is chosen (**pivot**)
  - E.g.: 4
- <u>Step 2:</u> given the pivot, the elements of the array are organised in a way that the elements on its left are smaller and those on its right are greater:

2	1	3	4	8	7	5	6

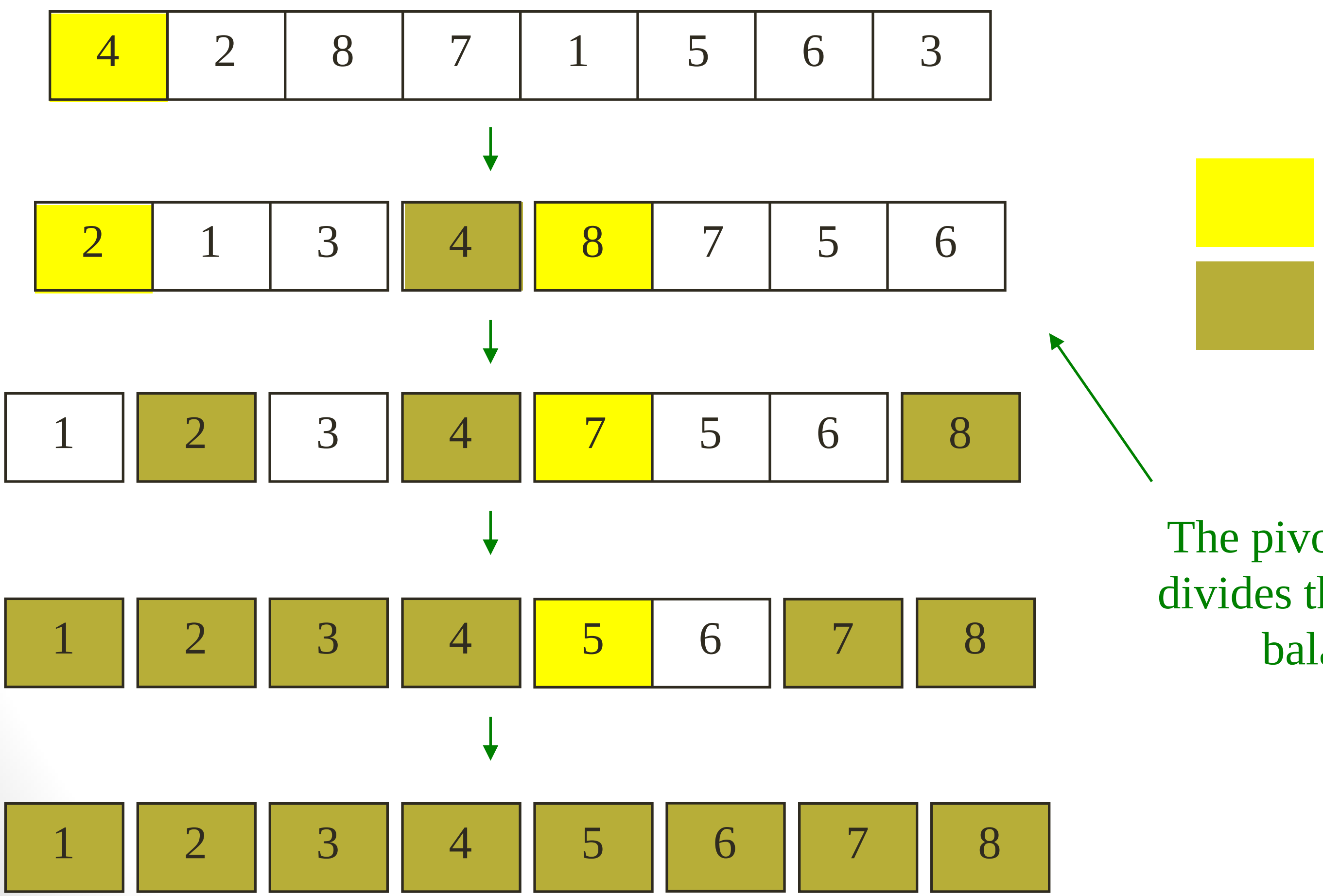
The pivot is already in its final position

- <u>Step 3:</u> we do the same with the subarrays on its left and right

2. Divide & Conquer

2.3. QuickSort

E.g. pivot is the leftmost element



The pivot of the left part divides the array into two balanced parts

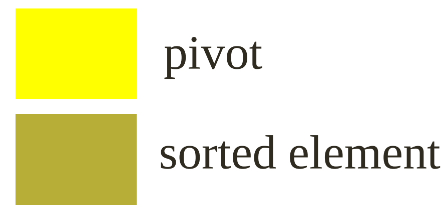
2. Divide & Conquer
2.3. QuickSort

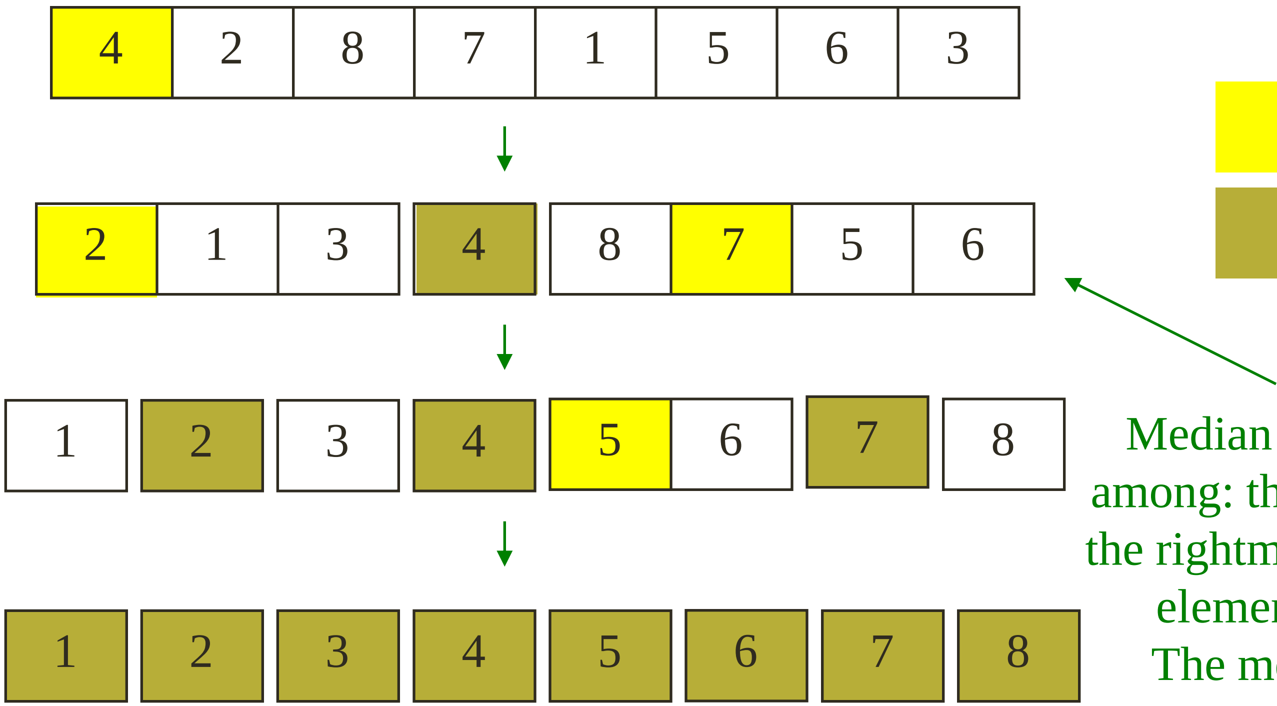
- A not properly chosen pivot produces imbalanced partitions and higher complexity
- A good pivot divides the array in two subarray of equal size, that is, it has to be the **median** of the array
- To calculate the median has a high complexity. Therefore, as approximation the ***median of three*** is employed (as the leftmost element, the rightmost element and the element in the middle)

2. Divide & Conquer

2.3. QuickSort

Pivot: median of three





Median of three is calculated among: the leftmost element (8), the rightmost element (6) and the element in the middle (7).
The median is 7: 6 < 7 < 8

It took less time to sort the array now in comparison to when we selected as pivot the leftmost element

2. Divide & Conquer

2.3. QuickSort

- **Partition**: in the following code the elements smaller than the pivot are on the left and those greater than the pivot on the right

```
int posPivot = selectPivot(v, left, right);
T pivot = v[posPivot];
swap(v, posPivot, right);
int i = left, j = right - 1;
do {
    while (v[i].compareTo(pivot) < 0 ) i++;
    while (v[j].compareTo(pivot) > 0 ) j--;
    if (i < j) {
            swap(v, i, j); i++; j--;
    }
} while (i <= j);
swap(v, i, right);
```

- The complexity of this algorithm is linear

2. Divide & Conquer

2.3. QuickSort

```
private static <T extends Comparable <T>>
    void quickSort(T[] v, int left, int right) {
    if (left < right) {
        int indexP = partition(v, left, right);        // DIVIDE
        quickSort(v, left, indexP - 1);                // CONQUER
        quickSort(v, indexP + 1, right);               // CONQUER
    }                                                  // COMBINE
}
```

- The complexity of *QuickSort* depends on the method `partition`:
  - Best case: `partition` divides the array into two balanced halves
  - Worst case: `partition` divides it into completely imbalanced two parts: a part with all the elements and the other one with none

2. Divide & Conquer

2.3. QuickSort

- If `partition` divides the array into two balanced halves:

$$T_{quickSort}^{M}(x) = 2 * T_{quickSort}^{M}(x/2) + \underbrace{k * x}_{\textit{complexity of partition}} \Rightarrow$$

$$\Rightarrow T_{quickSort}^{M}(x) \in \Theta(x*\log_2 x)$$

- If `partition` divides it in a completely imbalanced way:

$$T_{quickSort}^{P}(x) = T_{quickSort}^{P}(x-1) + k * x \Rightarrow$$

$$\Rightarrow T_{quickSort}^{P}(x) \in \Theta(x^2)$$

- *QuickSort* nearly always is faster than *MergeSort*:
  - Although the complexity of both of them is $\Theta(x*\log_2 x)$, the process of *Partition* is more efficient than *Merge*

2. Divide & Conquer

2.4. QuickSelect

- To find the k-th smallest element of an array
- If we use *QuickSort* the problem is soved with a complexity $\Theta(x*\log_2 x)$
- With *InsertionSort*: $\Theta(k*x)$
- The method *QuickSelect* allows to solve it with linear complexity

2. Divide & Conquer

2.4. QuickSelect

```
static <T extends Comparable <T>>
  void QuickSelect(T[] v, int k, int left, int right) {
  if (left + LIMIT > right) InsertionSort(v, left, right);
  else {
    int indexP = partition(v, left, right);
    if (k-1 < indexP)
      QuickSelect(v, k, left, indexP-1);
    else if (k-1 > indexP)
      QuickSelect(v, k, indexP+1, right);
  }
}
```

threshold for the selection of the sorting method

2. Divide & Conquer

2.4. QuickSelect

```
public static <T extends Comparable<T>>
  T select(T v[], int k) {
      return select(v, 0, v.length - 1, k - 1);
  }

private static <T extends Comparable <T>>
  T select(T[] v, int left, int right, int k) {
      if (left == right) return v[k];
      else {
        int indexP = partition(v, left, right);
        if (k <= indexP) return select(v, left, indexP, k);
        else return select(v, indexP + 1, right, k);
      }
}
```