Computer Science 1

Sorting Problems

Thomas Grenier

Summary

I. Introduction

II. First example : sorting algorithms

III. Bases of C++ syntax

IV. Second example: sorting with linked list

V. C++ Classes and UML

VI. Useful data structures

VII. Exercises/project

- Algorithms design and C++

Sorting algorithms

- Definition of the sorting problem
  - Sort a sequence of numbers into non-decreasing order
    - ***Input***: sequence of ***n*** numbers $\{a_1, a_2, \ldots a_n\}$
    - ***Output***: permutation (reordering) $\{a'_1, a'_2, \ldots a'_n\}$ of the input sequence such as $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
  - The input sequence is usually a *n*-element array
  - Numbers to be sorted are rarely isolated values
    - Part of data collection called a ***record***
    - Each record contains a ***key***, which is the value to be sorted
    - The remainder of the record consists of ***satellite data***

Sorting algorithms

- Why sorting?
  - Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms
    - Easy to understand
    - Inherent in many applications
    - Used as subroutine
      - (graphical layered objects render, …)
    - There is a wide variety of sorting algorithms
      - Large set of techniques are used (memory, data structure, recurrence)
      - Application of correctness and efficiency demonstrations

List of sorting algorithms

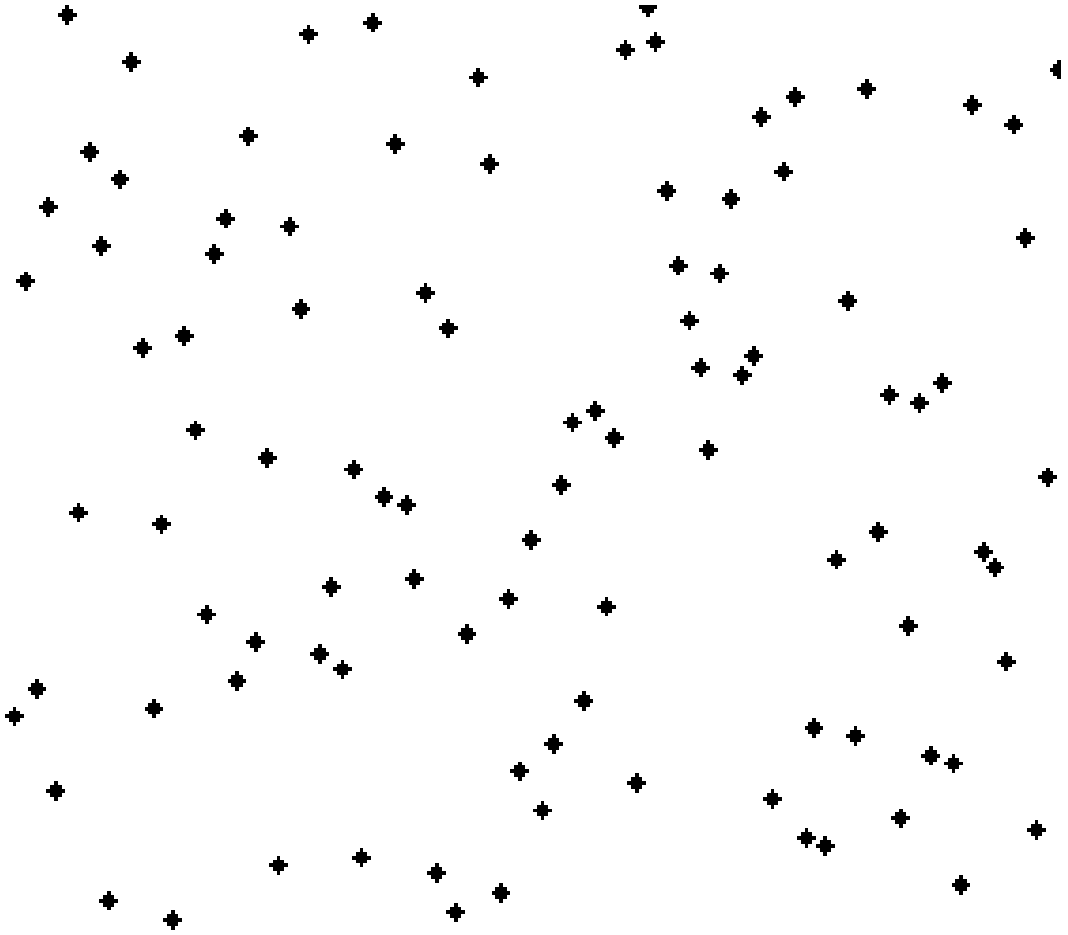
- More than 20 good sorting algorithms
  - Wikipedia → list of sorting algorithm
- Popular sorting algorithms
  - ***Bubble-Sort***
  - ***Insertion-Sort, Selection-Sort***
  - Shell-Sort
  - ***Merge-Sort, Quick-Sort***
  - Heapsort
  - ***Counting-Sort***, BucketSort,

} IntroSort

Bubble-Sort algorithm

- Method
  - The algorithm works as follows:
    - The algorithm starts at the beginning of the data set.
    - It compares the first two elements, and if the first one is greater than the second one, it swaps them.
    - It continues doing this for each pair of adjacent elements to the end of the data set.
    - And then it starts again with the first two elements, repeating until no swaps have occurred on the last pass.
- Write the Bubble-Sort algorithm
- *Show* the correctness of the Bubble-Sort algorithm
- Give the time complexity (worst and best case)

Bubble-Sort algorithm, *animation*



Bubble-Sort algorithm *While*

```
WHILE-BUBBLE-SORT(A)
1  swap ← TRUE
2  while swap = TRUE
3        do swap ← FALSE
4           for i ← 1 to length[A] − 1
5                 do if A[i] > A[i + 1]
6                       then exchange A[i] ↔ A[i + 1]
7                            swap ← TRUE
```

➔Correctness :

- ➔ the ***for*** inner loop improve the sorting sequence of A
  - ➔The j^{th} highest value of A moves to j^{th} index at the j^{th} execution of this for loop
- ➔The process ends when no permutation occurs :
  - ➔each A[i] is less or equal than A[i+1]

➔ Best case: already sorted array $T(n) = O(n)$

➔Worst case: invert sorted array $T(n) = O(n^2)$

- ➔Last element has to move to the first index

➔ Memory: (defined later) $M(n) = O(1)$

Bubble-Sort algorithm, *For*

- Another version of *Bubblesort*

```
BUBBLESORT(A)
1  for i ← 1 to length[A]
2       do for j ← length[A] downto i + 1
3            do if A[j] < A[j − 1]
4                 then exchange A[j] ↔ A[j − 1]
```

➔ Give the time complexity (worst and best case)

➔ Prove the correctness of the *Bubblesort* algorithm

Insertion-Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

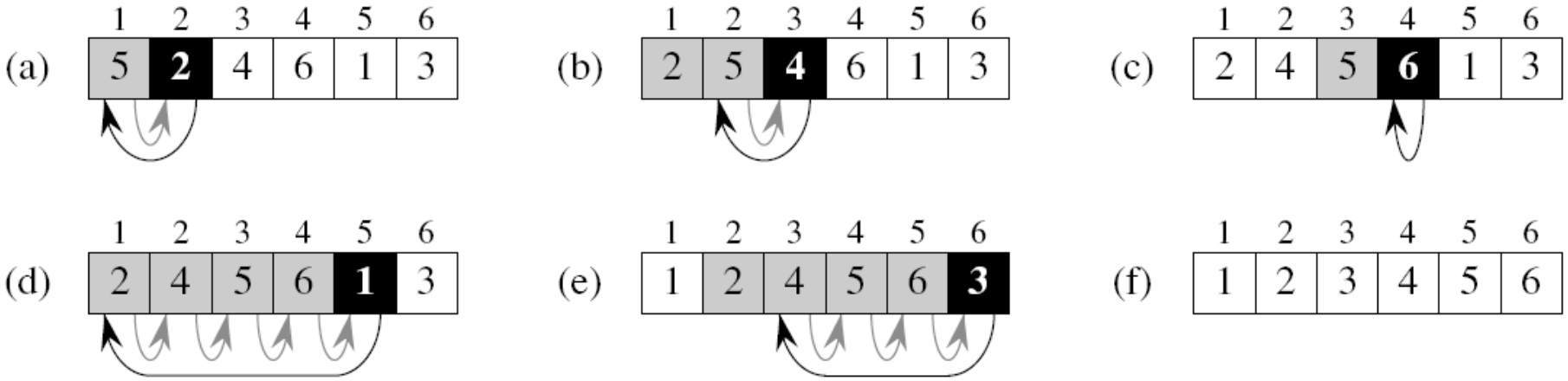
➔ Time complexity $T(n) = O(n^2)$

➔ Memory $M(n) = O(1)$

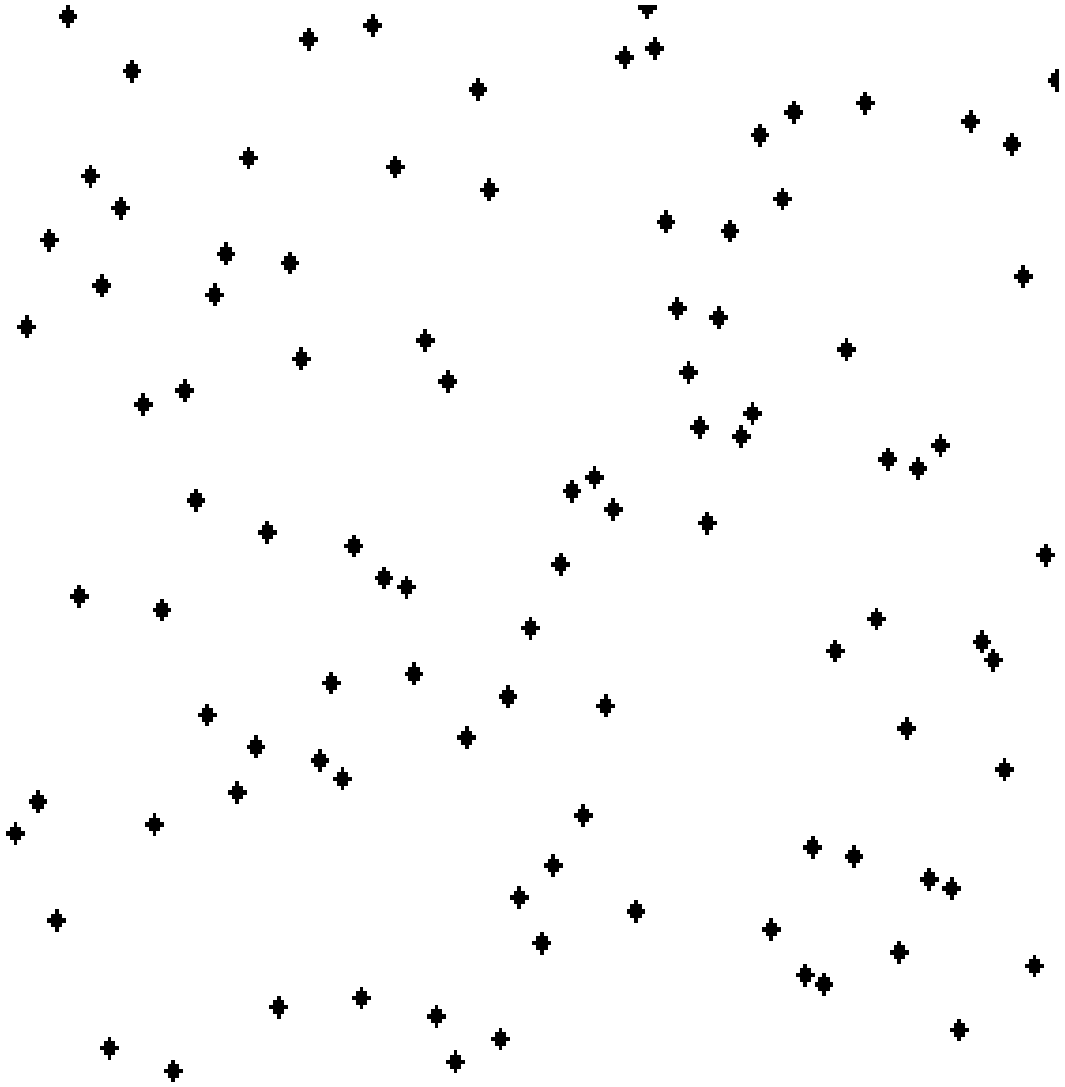
"Memory" denotes the amount of auxiliary storage needed beyond that used by the array itself

Insertion-Sort

- How this algorithm works



Insertion-Sort, *animation*



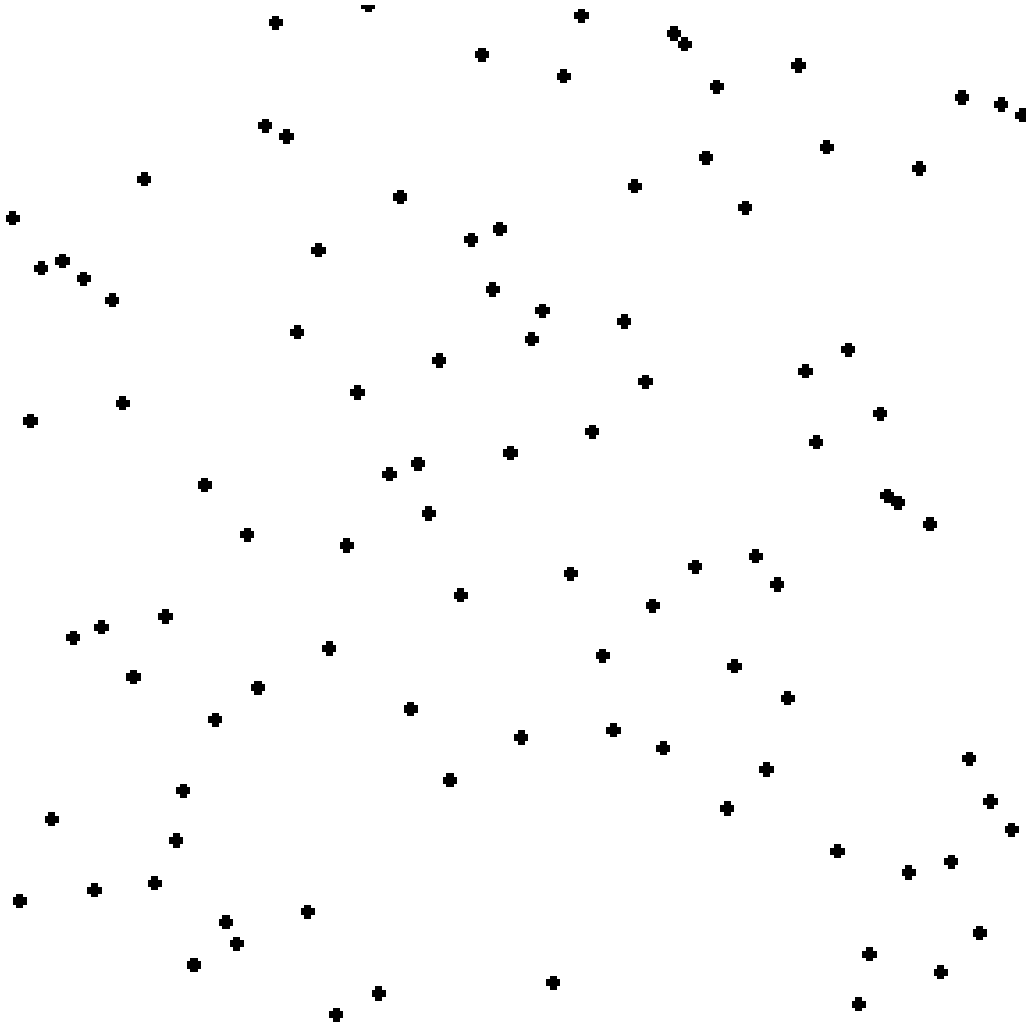
Selection-Sort algorithm

- Method
  - The algorithm works as follows:
    - Find the minimum value in the array (index!)
    - Swap it with the value in the first position
    - Repeat the steps above for remainder of the array (starting at the next position)

➔ Write the Selection-Sort algorithm

➔ Give the time and memory complexities of this algorithm

Selection-Sort algorithm, *animation*



Not-In-Place Selection-Sort algorithm

- Give a sorting algorithm using 2 different arrays based on Selection-Sort algorithm
- Give the time complexity
- Give the memory complexity
- Compare these values to values of the In-place version

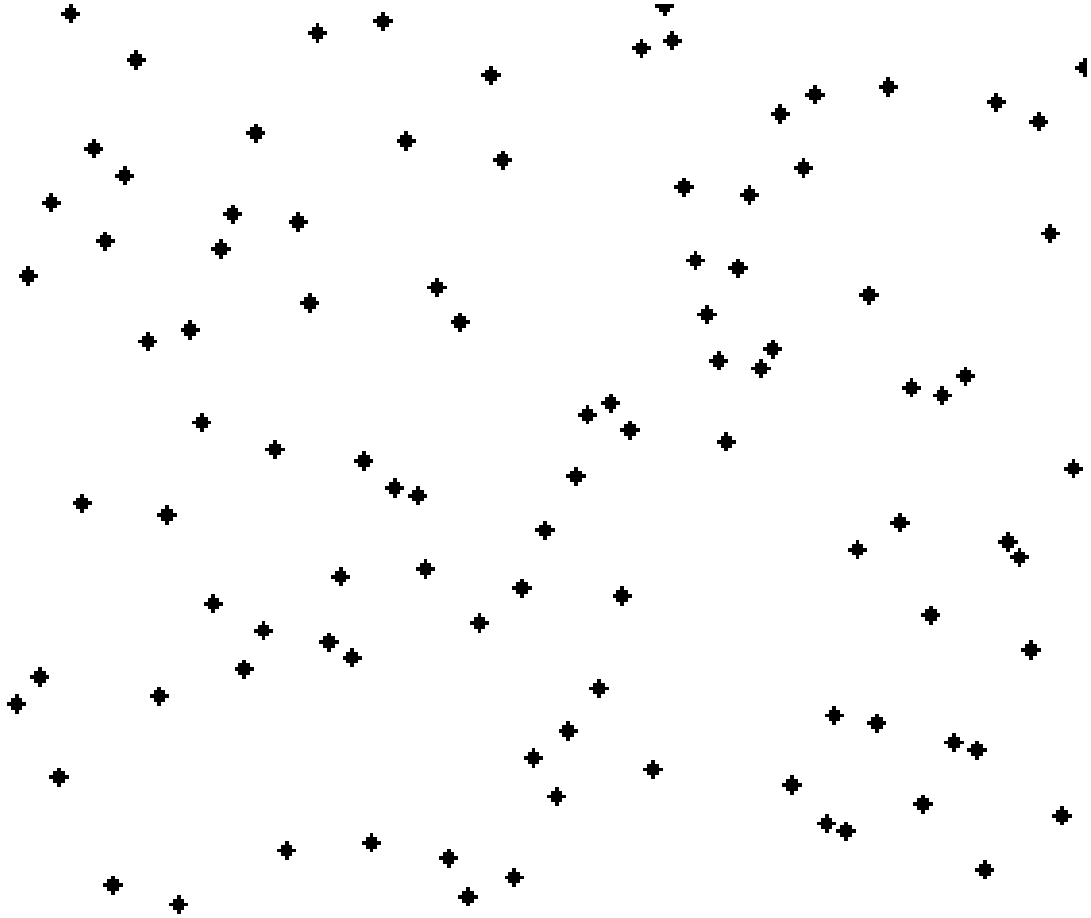
Merge-Sort algorithm, *recursive*

- Recursive algorithms
  - To solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems
  - These algorithms typically follow a ***Divide and conquer*** approach
    - ***Divide*** the problem into a number of subproblems
    - ***Conquer*** the subproblems by solving them recursively. If the subproblem sizes are small enough, just solve the subproblems in a straightforward manner
    - ***Combine*** the solutions of the subproblems into the solution of the original problem

Merge-Sort algorithm, *recursive*

- For Merge-Sort
  - **Divide** the n-element sequence to be sorted into ***two*** subsequences of n/2 elements
  - **Conquer**: Sort the two subsequences recursively using Merge-Sort
  - **Combine**: Merge the two sorted subsequences to produce the sorted answer
  - ➔Write the Merge(A,p,q,r) algorithm which merges two sorted sequences (A[p..q] and A[q+1..r]).

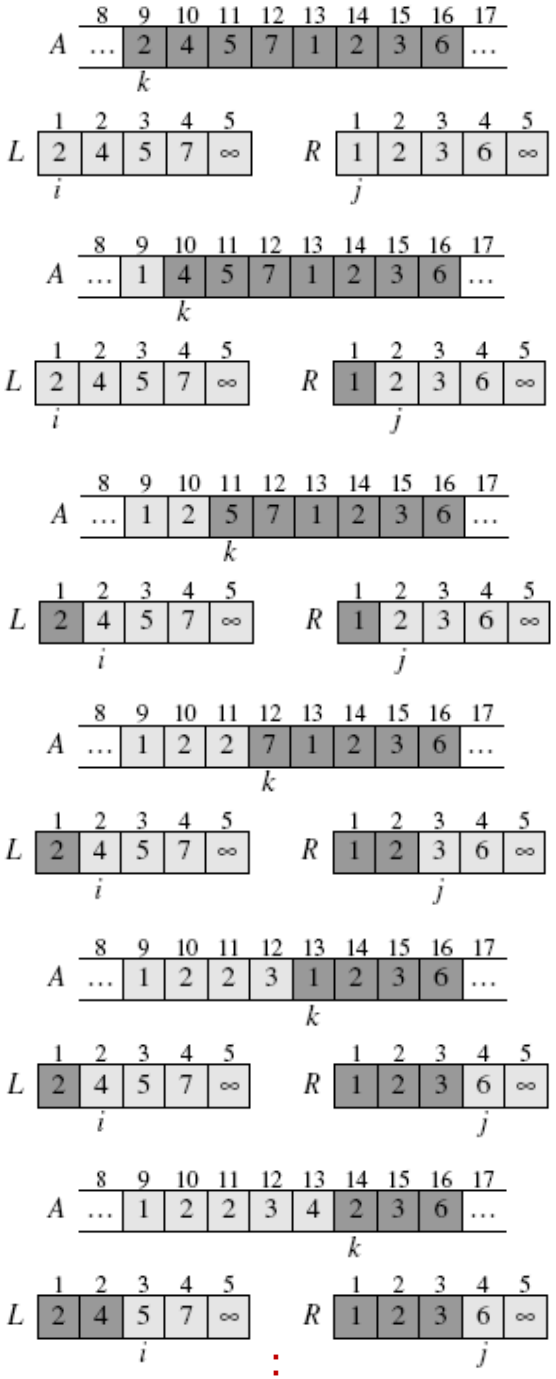
Merge-Sort algorithm, *animation*



Merge-Sort algorithm, *recursive*

- Write the Merge(A,p,q,r) algorithm which merges two sorted subarrays (A[p..q] and A[q+1..r]),and then forms a single sorted subarray that replaces the current subarray (A[p..r])
  - Hints : create 2 subarrays

```
MERGE(A, p, q, r)
 1  create arrays L[1..q − p + 1] and R[1..r − q]
    ▷ First : copy the A array in the subarrays L and R
 2  for Li ← 1 to Length[L]
 3        do L[Li] ← A[Li + p − 1]
 4  for Ri ← 1 to Length[R]
 5        do R[Ri] ← A[Ri + q]
    ▷ Second : merge L and R to A (sorted!)
 6  Li ← 1
 7  Ri ← 1
 8  for Ai ← p to r
 9        do if Li ≤ Length[L] and Ri ≤ Length[R]
10              then if L[Li] ≤ R[Ri]
11                      then A[Ai] ← L[Li]
12                           Li ← Li + 1
13                      else A[Ai] ← R[Ri]
14                           Ri ← Ri + 1
15              else
                        ▷ End of subarrays
16                      if Li ≤ Length[L]
17                         then A[Ai] ← L[Li]
18                              Li ← Li + 1
19                      if Ri ≤ Length[R]
20                         then A[Ai] ← R[Ri]
21                              Ri ← Ri + 1
```



Merge-Sort algorithm, *recursive*

- Write the Merge-Sort(*A*,*p*,*r*) algorithm which sort recursively the array *A[p..r]*
  - If p ≥ r the subarray contains at most one element and then it is already sorted
  - Otherwise, the divide step computes an index q that partition A[p..r] into 2 subarrays
    - A[p..q] containing $\lceil n/2 \rceil$ elements
    - A[q+1..r] containing $\lfloor n/2 \rfloor$ elements

Merge-Sort algorithm, *recursive*

- Running time analysis
  - Intuitively
    - Merge procedure takes O(n)
    - Merge-Sort procedure calls Merge procedure $\log_2(n)$ times

    ➔ Merge-Sort algorithms running time is $O(n.\log_2(n))$
  - **Master theorem** approach

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + C(n) + D(n) & \text{otherwise} \end{cases}$$

Straightforward solution

Number of subproblems

Subproblems depending value

Combine time

Divide time

Merge-Sort algorithm, *recursive*

- Master theorem

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + C(n) + D(n) & \text{otherwise} \end{cases}$$

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Merge-Sort algorithm, *recursive*

- Master theorem applied to Merge-Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(1) & \text{if } n > 1 \end{cases}$$

thus

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + c.n & \text{if } n > 1 \end{cases}$$

then

$$\Theta(n^{\log_2(2)}) = \Theta(n) = c.n \quad \Rightarrow \quad T(n) = \Theta(n \log_2(n))$$

The constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps

Merge-Sort algorithm, *recursive*

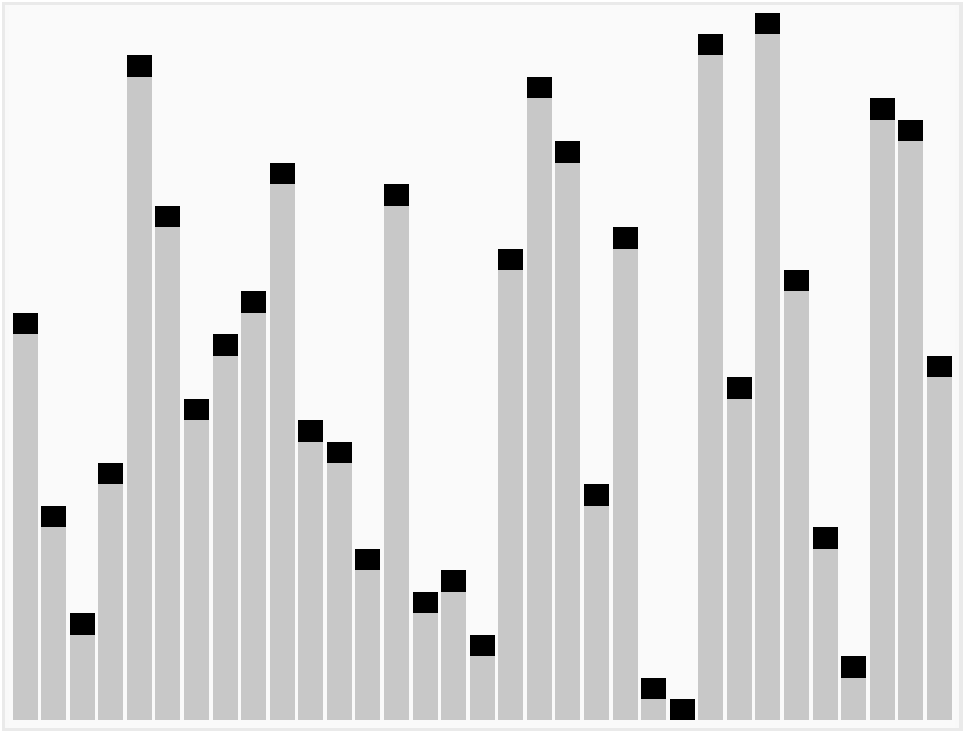
- Memory needed by the Merge-Sort algorithm?
  - The A array (normal…)
  - At the last step the summed size of subarrays is *n*

  ➔ M(n)=O(n)

Quicksort

- Method
  - Quicksort sorts by using a divide and conquer strategy to divide an array into two sub-arrays
  - The steps are:
    - Pick an element, called a ***pivot***, from the array
    - Reorder the array so that all elements which are smaller than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go whatever the way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
    - Recursively sort the sub-array of smaller elements and the sub-list of greater elements
  - The base cases of the recursion are either array of size zero or of size one, which are always sorted

Quicksort, *animation*



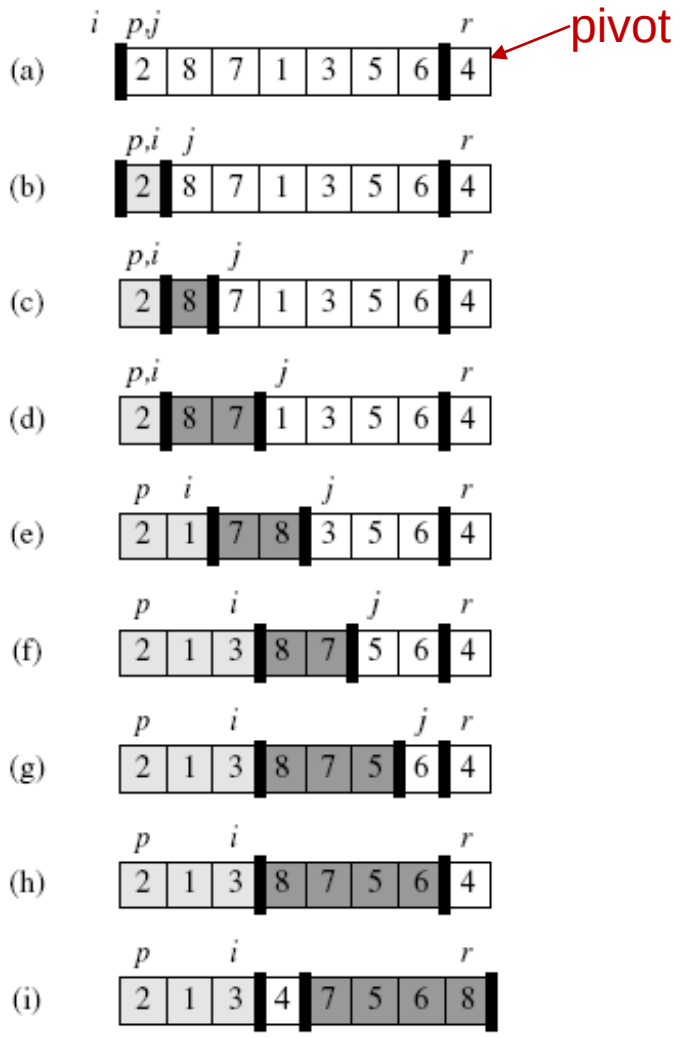
Quicksort

- Write the quicksort algorithm
- Compare time complexity of quicksort with merge-sort
- Compare memory used by quicksort to memory used by merge-sort

Quicksort algorithm

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```



Partition procedure

Sorting in O(n)

- Game…
  - Students vs. teacher