



# Theory of algorithms (1st lecture)

Pál Pusztai  
[pusztai@sze.hu](mailto:pusztai@sze.hu)

## Outline

- Foundations
- Efficiency of the algorithms
- Asymptotic notation
- Sorting methods
  - Insertion sort
  - Merge sort
  - Quicksort
- Exercises



## Preface

*„Before there were computers, there were algorithms. But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing.”*

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: **Introduction to Algorithms**



# Foundations

- What is the algorithm?
  - Informally, an **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.
    - An algorithm is thus a sequence of computational steps that transform the input into the output.
    - The algorithm describes a specific computational procedure for achieving that input/output relationship.
  - The **sorting problem** (formally definition)
    - **Input:** A sequence of  $n$  numbers:  $\langle a_1, a_2, \dots, a_n \rangle$ .
    - **Output:** A permutation (reordering),  $\langle a'_1, a'_2, \dots, a'_n \rangle$  of the input sequence such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .
    - For example:  $\langle 31, 41, 59, 26, 41, 58 \rangle \rightarrow \langle 26, 31, 41, 41, 58, 59 \rangle$ .



# Foundations

- „How good” an algorithm is?
  - An algorithm is said to be **correct** if, for every input instance, it halts with the correct output.
    - We say that a correct algorithm **solves** the given computational problem.
  - The measure of **efficiency** usually is **speed**, i.e., how long an algorithm takes to produce its result (running **time**).
    - There are some problems, however, for which no efficient solution is known.
  - Data structure, memory size
    - The (used) **memory size** is inversely proportional to the (running) **time** and vice versa.
  - It is necessary to think about **designing** and **analyzing** algorithms.
    - *„With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more.”*



## Foundations

- Our algorithms will be given with **subroutines** (procedure or function) and with **pseudocode**.
- Advances/properties of **pseudocode**
  - It is **similar** to C, C++, Java, Python, or Pascal languages.
  - The most **expressive** and **clear** method can be employ (to specify a given algorithm).
  - An English phrase or sentence can be embedded in a section of “real” code.
  - It is not typically concerned with issues of software engineering.
    - Issues of data abstraction, modularity, and error handling are often ignored (in order to convey the essence of the algorithm more concisely).



# Foundations

- Pseudocode conventions, notation
  - One line contains one statement.
  - The **indentation** of the pseudocode indicates the **block structure**.
  - **Multiply assignment**
    - $i = j = e$  equivalent  $j = e$  assignment followed by  $i = j$  assignment.
  - The symbol `==` indicates the relation of **equality**.
  - Access an **element** of an **array**
    - $A[i]$  indicates the  $i$ th element of the (one dimensional) array  $A$ .
    - $B[2, j+1]$  indicates the  $j+1$ th element of the 2nd row of the (two dimensional ) array  $B$ .
    - $A[1..j]$  indicates the **subarray** of  $A$  consisting of the  $j$  elements  $A[1], A[2], \dots, A[j]$ .
  - Giving an (one dimensional) array
    - For example:  $A = \langle 5, 3, 8, 1, 9, 7, 2, 6, 4 \rangle$
  - Compound data (treated as objects) are composed of **attributes**
    - For example:  $A.length$  specifies the number of elements in array  $A$ .

# Foundations

- Pseudocode conventions, notations
  - The Boolean operators (**and**, **or**) are **short circuiting**.
  - The symbol `//` indicates that the remainder of the line is a **comment**.
  - The keyword **error**
    - It is occurred when the conditions are wrong for the procedure to have been called.
    - The calling procedure is responsible for handling the error (so we do nothing).
  - The **return** statement
    - It immediately transfers control back to the point of call in the calling procedure.
    - Most **return** statements also take a value to pass back to the caller. We allow multiple values to be returned in a single **return** statement.
  - Variables and parameters
    - Variables (such as  $i$ ,  $j$ ,  $key$ ) are **local** to the given procedure.
    - Simple parameters are passed **by value** and objects are passed **by reference**.
    - Variables are representing an **array** or **object** as a **pointer** to the data.
    - **NIL** indicates if a pointer refers to no object at all. In some figures (if there is no enough space) it will be replaced with `/`.



# Foundations

## ■ Floors and ceilings

- For any real number  $x$ , we denote
  - the greatest integer less than or equal to  $x$  by  $\lfloor x \rfloor$  (“the **floor** of  $x$ ”), and
  - the least integer greater than or equal to  $x$  by  $\lceil x \rceil$  (“the **ceiling** of  $x$ ”).
- For all real  $x$ ,  $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$ .
  - For example:  $x = 3.8$ ,  $\lfloor x \rfloor = 3$ ,  $\lceil x \rceil = 4$ ;  $x = -3.8$ ,  $\lfloor x \rfloor = -4$ ,  $\lceil x \rceil = -3$
- For all integer  $n$ ,  $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$ .
  - For example:  $n = 7$ ,  $\lfloor n/2 \rfloor = 3$ ,  $\lceil n/2 \rceil = 4$ ;  $n = 6$ ,  $\lfloor n/2 \rfloor = 3$ ,  $\lceil n/2 \rceil = 3$



## Elements of structured programming

- Every algorithm can be constructed with **sequence**, **selection** and **iteration**.
  - If an algorithm is given with these elements only, then (we say that) this algorithm is a **structured algorithm**.
- Remarks
  - The program languages (usually) allow us to make **unstructured** subroutine, but a structured subroutine is (usually) more clear and easier to maintain.
    - For example: exit from loop or subroutine, conditional and unconditional control passing.



# Elements of structured programming

## ■ Sequence

$S_1$   
 $S_2$   
 ...  
 $S_n$

## ■ Iteration 1 (testing before executing $S$ )

**while**  $C$   
      $S$

## ■ Iteration 2 (testing after executing $S$ )

**repeat**  
      $S$   
**until**  $C$

## ■ Selection

**if**  $C_1$        $S_1$   
**else if**  $C_2$      $S_2$   
 ...  
**else if**  $C_n$      $S_n$

## ■ Iteration 3 (incremental/for loop)

**for**  $counter = start$  **to|downto**  $end$   
      $S$

- $S_i$ ,  $S$ : Statement(s)
- $C_i$ ,  $C$ : Condition  
 (logical expression)

- The *counter* is incremented (**to**) or decremented (**downto**) automatically by 1.



# Input and output statements

- Input statement
  - **read** *variable-list*
    - For example: **read** *a,b,c*
- Output statement
  - **write** *expression-list*
    - For example: **write** "The result:", *result*
- Remarks
  - Usually these statements are not necessary due to the algorithms are given with **subroutines** where the input and output data can be passed with **parameters**.

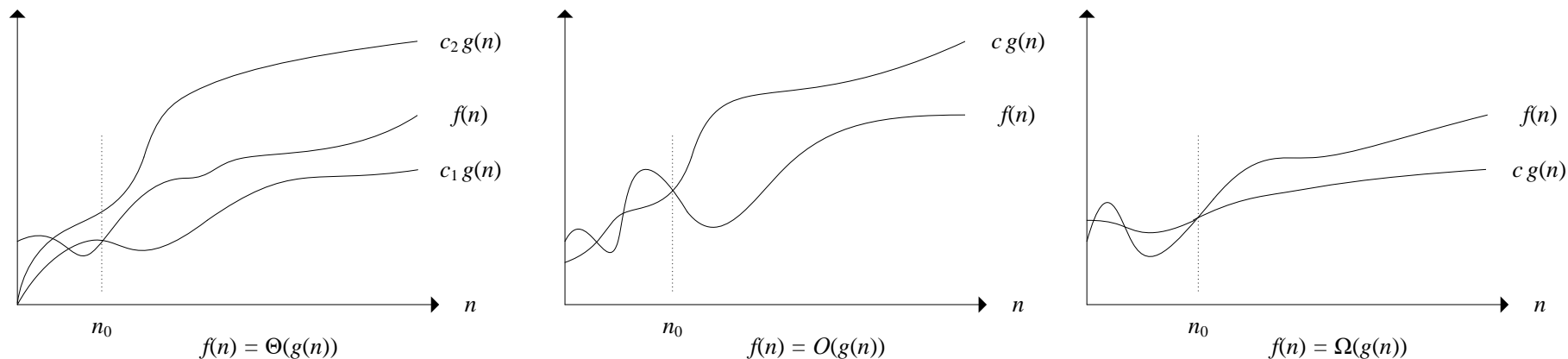


## Exercises

- Give algorithm with pseudocode to output the odd integers in  $[1, 10]$ . Do it with all kind of iterations!



# Growth of functions



## Asymptotic notations

$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$

$O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0\}.$

$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0\}.$

$o(g(n)) = \{f(n): \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < c g(n) \text{ for all } n \geq n_0\}.$

$\omega(g(n)) = \{f(n): \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq c g(n) < f(n) \text{ for all } n \geq n_0\}.$

$\Theta$ : **asymptotically tight** bound;  $o, \omega$ : **not asymptotically tight** bounds.

$O, \Omega$ : asymptotically tight or not asymptotically tight bounds.

$\Theta(1), O(1)$  denotes the constant functions.



# Growth of functions

## ■ Remarks

- The asymptotic notations give **set of functions**, but we use symbol  $=$  instead of symbol  $\in$ .
  - For example:  $2n^2 + n + 100 = \Theta(n^2)$  instead of  $2n^2 + n + 100 \in \Theta(n^2)$
- Polynomials
  - Given a nonnegative integer  $d$ , a **polynomial in  $n$  of degree  $d$**  is a function  $p(n)$  of the form  $p(n) = \sum_{i=0}^d a_i n^i$ , where the constants  $a_0, a_1, \dots, a_d$  are the **coefficients** of the polynomial and  $a_d \neq 0$ .
  - A polynomial is **asymptotically positive** if and only if  $a_d > 0$ . For an asymptotically positive polynomial  $p(n)$  of degree  $d$ , we have  $p(n) = \Theta(n^d)$ .
  - A function  $f(n)$  is **polynomially bounded** if  $f(n) = O(n^k)$  for some constant  $k$ .
- Exponential and polynomial functions
  - For any  $c > 1$  and  $k > 1$  integer we have  $c^n = \omega(n^k)$ ,  $n^k = o(c^n)$ , thus any exponential function with a base strictly greater than 1 grows faster than any polynomial function.
- Logarithm functions
  - $\lg(n!) = \Theta(n \lg n)$
  - $\lg n = \log_2 n$  (**binary logarithm**),  $\ln n = \log_e n$  (**natural logarithm**),  $\log_b a = \frac{\log_c a}{\log_c b}$



## Exercises

### ■ True or false?

- $0.1n^2 - 3n + 1 = \Theta(n^2)$
- $2n = O(n^2)$
- $2n = \Omega(n^2)$
- $2n = o(n^2)$
- $2n = o(n)$
- $2n^2 = \omega(n)$
- $2^{n+1} = O(2^n)$
- $2^{2n} = O(2^n)$
- $n! = \Omega(n^n)$ .
- $n! = \Omega(2^n)$ .
- $f(n) = \Theta(g(n))$  if and only if  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .
- $f(n) = \Theta(g(n))$  if and only if  $g(n) = \Theta(f(n))$ .
- $f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$ .
- $f(n) = o(g(n))$  if and only if  $g(n) = \Omega(f(n))$ .
- $o(f(n)) \cap \omega(f(n)) = \emptyset$





## Insertion sort

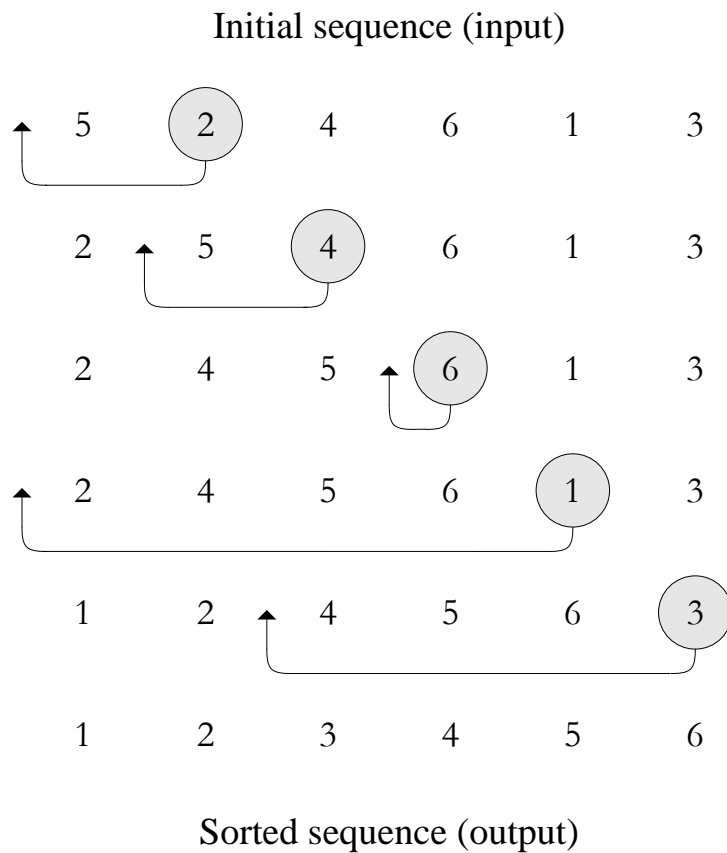
INSERTION-SORT( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ 
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i+1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i+1] = key$ 
```

**Efficiency:** For  $n$  elements (the size of input is  $n=A.length$ ), the running time is  $O(n^2)$ .



# Insertion sort



**The operation of INSERTION-SORT**



## Exercises

- What input is the worst-case and what is the best-case for the INSERTION-SORT?
- Illustrate the operation of INSERTION-SORT with array  $A$ . Give the contents of the array after each element is inserted into its place.
  - $A = \langle 8, 2, 1, 5, 6, 9, 4, 3, 7 \rangle$



# Merge sort

- **Divide-and-conquer** approach:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - **Combine** the solutions to the subproblems into the solution for the original problem.
- The **merge sort** algorithm follows these steps:
  - **Divide:** Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each.
  - **Conquer:** Sort the two subsequences recursively using merge sort.
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer.

MERGE-SORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \lfloor (p+r)/2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q+1, r$ )
5      MERGE( $A, p, q, r$ )
```

**Remark:** To sort entire array  $A$ , the initial call is MERGE-SORT( $A, 1, A.length$ ).

**Efficiency:** For  $n$  elements the running time is  $\Theta(n \lg n)$ .



## Merge sort

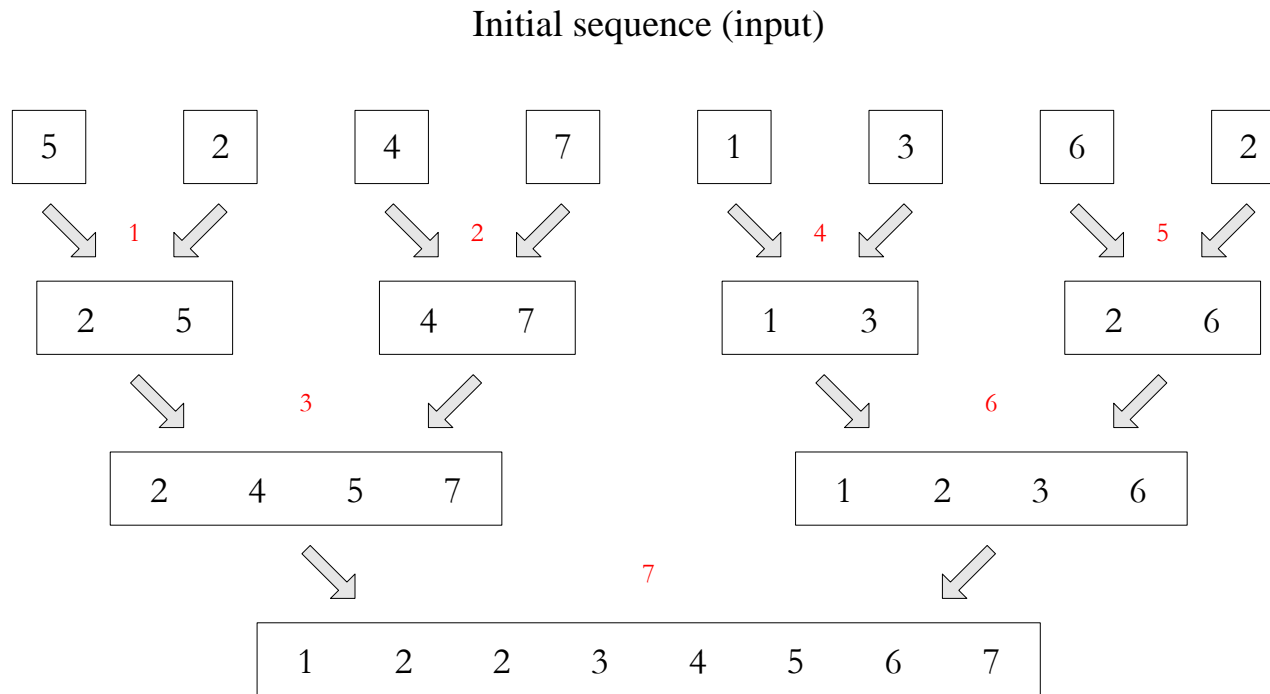
MERGE( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1+1]$  and  $R[1..n_2+1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1+1] = \infty$ 
9   $R[n_2+1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else
17          $A[k] = R[j]$ 
18          $j = j + 1$ 
```

**Efficiency:** For the subarray of  $n$  elements the running time is  $\Theta(n)$ .



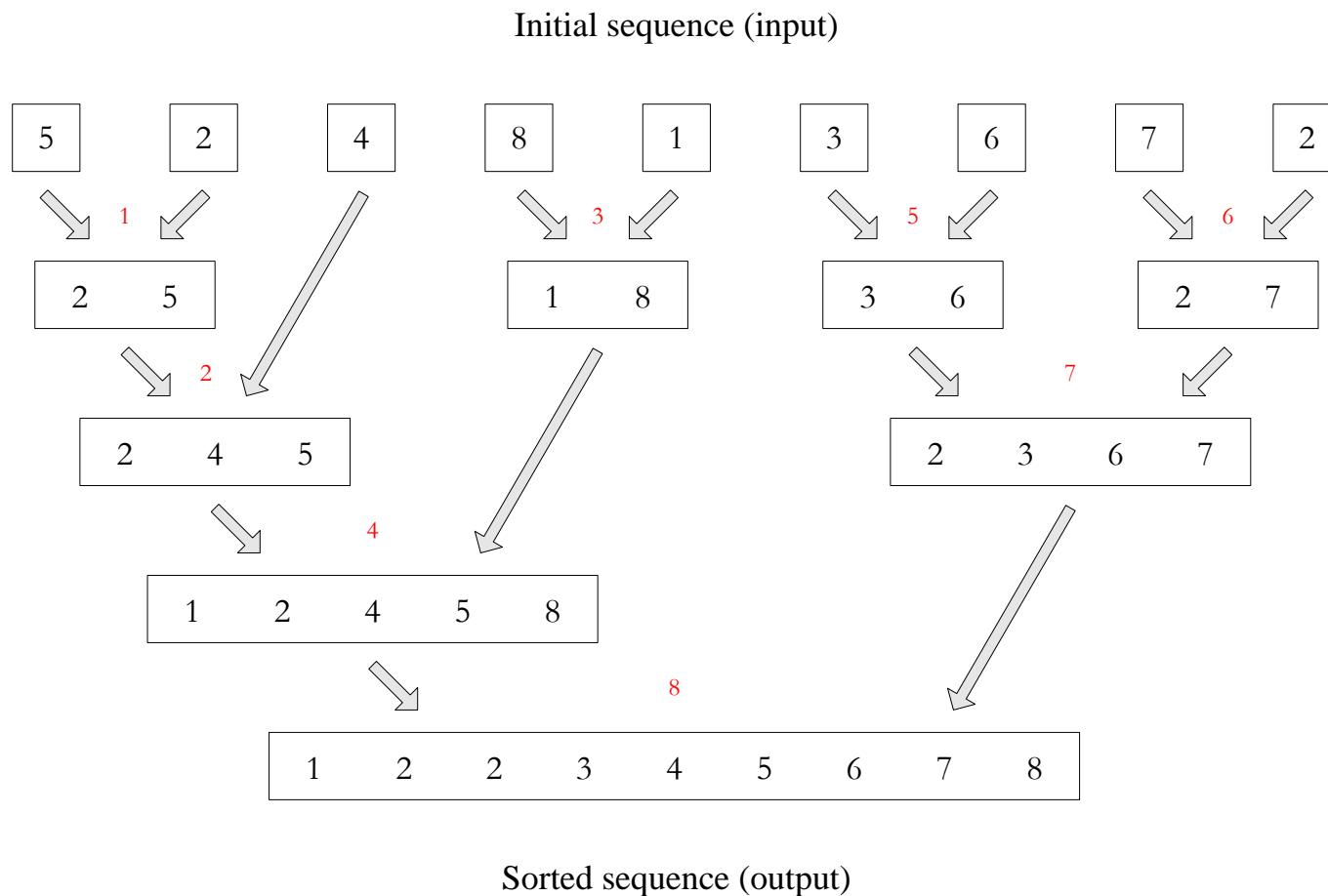
# Merge sort



Sorted sequence (output)

**The operation of MERGE-SORT**

# Merge sort



## The operation of MERGE-SORT



## Exercises

- Illustrate the operation of MERGE-SORT with array  $A$ . Give a figure with arrows and the number of merge steps.
  - $A = \langle 8, 11, 5, 10, 2, 1, 9, 6, 7, 4, 3 \rangle$



# Quicksort

- Three-step divide-and-conquer process for sorting a typical subarray  $A[p..r]$ 
  - **Divide:** Partition (rearrange) the array  $A[p..r]$  into two (possibly empty) subarrays  $A[p..q-1]$  and  $A[q+1..r]$  such that  $x \leq A[q] \leq y$  is satisfied if  $x \in A[p..q-1]$  and  $y \in A[q+1..r]$ . Compute the index  $q$  as part of this partitioning procedure.
  - **Conquer:** Sort the two subarrays  $A[p..q-1]$  and  $A[q+1..r]$  by recursive calls to quicksort.
  - **Combine:** Because the subarrays are already sorted, no work is needed to combine them: the entire array  $A[p..r]$  is now sorted.

QUICKSORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q-1$ )
4      QUICKSORT( $A, q+1, r$ )
```

**Remark:** To sort entire array  $A$ , the initial call is  $\text{QUICKSORT}(A, 1, A.length)$ .

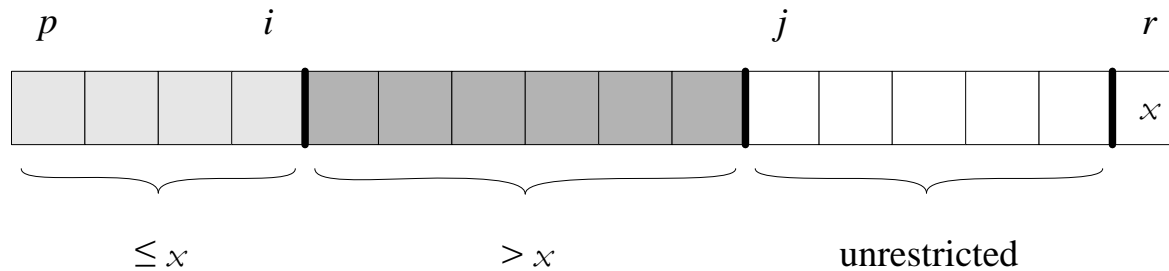
**Efficiency:** For the subarray of  $n$  elements the **average-case** running time is  $O(n \lg n)$ . If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort, if it is unbalanced, it can run asymptotically as slowly as insertion sort.



# Quicksort

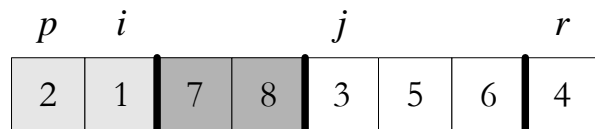
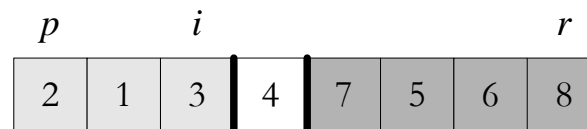
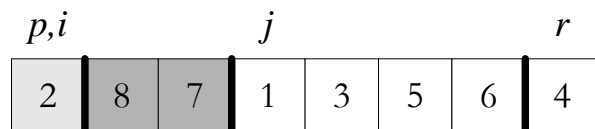
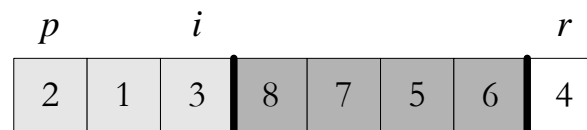
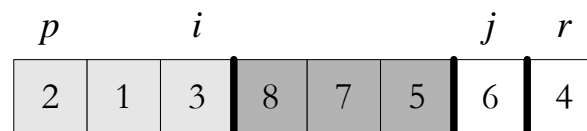
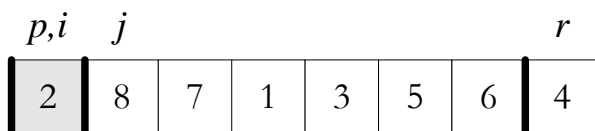
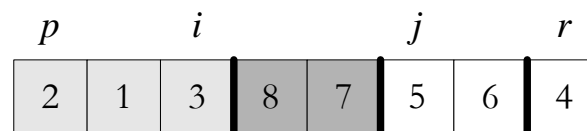
PARTITION( $A, p, r$ )

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```



The four regions maintained by PARTITION

# Quicksort



The operation of PARTITION

# Quicksort

- **Problem:** Bad efficiency ( $O(n^2)$ ) with some input cases.
- **Solution:** Randomly choose the pivot element.

RANDOMIZED-PARTITION( $A, p, r$ )

- 1  $i = \text{RANDOM}(p, r)$
- 2 exchange  $A[r]$  with  $A[i]$
- 3 **return** PARTITION( $A, p, r$ )

RANDOMIZED-QUICKSORT( $A, p, r$ )

- 1 **if**  $p < r$
- 2      $q = \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3     RANDOMIZED-QUICKSORT( $A, p, q-1$ )
- 4     RANDOMIZED-QUICKSORT( $A, q+1, r$ )

**Remark:** The  $\text{RANDOM}(a, b)$  function gives a random integer number in the interval  $[a, b]$ .

## Exercises

- Illustrate the operation of `PARTITION(A, 1, 9)` call with array  $A$ . How many exchanges will be done? What is the result value of the function?
  - $A = \langle 8, 2, 1, 5, 6, 9, 4, 3, 7 \rangle$
- Illustrate the operation of `QUICKSORT(A, 1, 6)` call with array  $A$ . Give the contents of the array after each exchange.
  - $A = \langle 3, 6, 2, 4, 5, 1 \rangle$

