PROGRAMMING AND ALGORITHMS II

GABRIEL VALIENTE

ALGORITHMS, BIOINFORMATICS, COMPLEXITY AND FORMAL METHODS
RESEARCH GROUP, TECHNICAL UNIVERSITY OF CATALONIA

2023–2024

Schedule 2023–2024

Week	Mon		Thu	
01	18 Sep	Theory 1	21 Sep	Lab 1
02	25 Sep	—	28 Sep	Lab 2
03	02 Oct	Theory 2	05 Oct	Lab 3
04	09 Oct	Theory 3	12 Oct	—
05	16 Oct	Partial exam 1	19 Oct	Exam review
06	23 Oct	Theory 4	26 Oct	Lab 4
07	30 Oct	Theory 5	02 Nov	Lab 5
08	06 Nov	Theory 6	09 Nov	Lab 6
09	13 Nov	Partial exam 2	16 Nov	Exam review
10	20 Nov	Lab 7	23 Nov	Lab 8

25%	Partial exam 1	16 Oct
25%	Partial exam 2	13 Nov
50%	Final exam	15 Dec

100%	Recovery exam	09 Jan

Theory 1	Lab 1
• Description of algorithms using structured pseudocode • Review of basic data structures • Review of elementary algorithms	• Flash cards 1 • Problems 1–6

Theory 2	Lab 2
• Parameter passing • References versus pointers • Python versus C++	• Flash cards 2 • Problems 7–12

Theory 3	Lab 3
• Analysis of iterative algorithms • Examples	• Flash cards 3 • Problems 13–18

Theory 4	Lab 4
• Linear recursion • Examples	• Flash cards 4 • Problems 19–24

Theory 5	Lab 5
• Tail recursion • Multiple recursion • Examples	• Flash cards 5 • Problems 25–30

Theory 6	Lab 6
• Analysis of recursive algorithms • Recurrence relations and their solution methods • Examples	• Flash cards 6 • Problems 31–36

DESCRIPTION OF ALGORITHMS USING STRUCTURED PSEUDOCODE

Some of the basic data structures needed for the description of algorithms are illustrated next using a fragment of pseudocode.

Pseudocode conventions follow modern programming guidelines, such as avoiding global side effects (with the only exception of object attributes, which are hidden behind dot-notation) and the unconditional transfer of control by way of goto or gosub statements.

- Assignment of value a to variable x is denoted by $x = a$.
- Comparison of equality between the values of two variables x and y is denoted by $x = y$, comparison of strict inequality is denoted by either $x \neq y$, $x < y$, or $x > y$, and comparison of non-strict inequality is denoted by either $x \leqslant y$ or $x \geqslant y$.
- Logical true and false are denoted by **true** and **false**, respectively.
- Logical negation, conjunction, and disjunction are denoted by **not**, **and**, and **or**, respectively.
- Non-existence is denoted by **nil**.

- Mathematical notation is preferred over programming notation. For example, the cardinality of a set S is denoted by $|S|$, membership of an element x in a set S is denoted by $x \in S$, insertion of an element x into a set S is denoted by $S = S \cup \{x\}$, and deletion of an element x from a set S is denoted by $S = S \setminus \{x\}$.
- Control structures use the following reserved words: **all**, **break**, **do**, **else**, **for**, **if**, **repeat**, **return**, **then**, **to**, **until**, **while**.
- Blocks of statements are shown by means of indention.

```
if ... then
    ...
else
    ...
```

```
for all ... do
    ...
    ...
return ...
```

```
while ... do
    ...
    if ... then
        ...
```

```
repeat
    ...
    ...
until ...
```

REVIEW OF BASIC DATA STRUCTURES

The collection of abstract operations on arrays, matrices, lists, stacks, queues, priority queues, sets, and dictionaries are presented next by way of examples.

An array is a one-dimensional array indexed by non-negative integers. The $[i]$ operation returns the i-th element of the array, assuming there is such an element.

> let $A[1..n]$ be a new array
> **for** $i = 1$ **to** n **do**
> $A[i] =$ **false**

A matrix is a two-dimensional array indexed by non-negative integers. The $[i, j]$ operation returns the element in the i-th row and the j-th column of the matrix, assuming there is such an element.

> let $M[1..m][1..n]$ be a new matrix
> **for** $i = 1$ **to** m **do**
>   **for** $j = 1$ **to** n **do**
> $A[i, j] = 0$

A list is just a sequence of elements. The **front** operation returns the first element and the **back** operation returns the last element in the list, assuming the list is not empty. The **prev** operation returns the element before a given element in the list, assuming the given element is not at the front of the list. The **next** operation returns the element after a given element in the list, assuming the given element is not at the back of the list. The **append** operation inserts an element at the rear of the list. The **concatenate** operation deletes the elements of another list and inserts them at the rear of the list.

let L be an empty list
append x to L
let L' be an empty list
append x to L'
concatenate L' to L

let x be the element at the front of L
while $x \neq nil$ **do**
    output x
    $x = L.next(x)$
let x be the element at the back of L
while $x \neq nil$ **do**
    output x
    $x = L.prev(x)$

A stack is a sequence of elements which are inserted and deleted at the same end (the top) of the sequence. The **top** operation returns the top element in the stack, assuming the stack is not empty. The **pop** operation deletes and returns the top element in the stack, also assuming the stack is not empty. The **push** operation inserts an element at the top of the stack.

let S be an empty stack
push x onto S
let x be the element at the top of S
output x
while S is not empty **do**
    pop from S the top element x
    output x

A queue is a sequence of elements which are inserted at one end (the rear) and deleted at the other end (the front) of the sequence. The **front** operation returns the front element in the queue, assuming the queue is not empty. The **dequeue** operation deletes and returns the front element in the queue, assuming the queue is not empty. The **enqueue** operation inserts an element at the rear end of the queue.

let Q be an empty queue
enqueue x into Q
let x be the element at the front of Q
output x
repeat
    dequeue from Q the front element x
    output x
until Q is empty

A priority queue is a queue of elements with both an information and a priority associated with each element, where there is a linear order defined on the priorities. The **front** operation returns an element with the minimum priority, assuming the priority queue is not empty. The **dequeue** operation deletes and returns an element with the minimum priority in the queue, assuming the priority queue is not empty. The **enqueue** operation inserts an element with a given priority in the priority queue.

```
let Q be an empty priority queue
enqueue (x, y) into Q
let (x, y) be an element x
    with the minimum priority y in Q
output (x, y)
repeat
    dequeue from Q an element x
        with the minimum priority y
    output (x, y)
until Q is empty
```

A set is just a set of elements. The **insert** operation inserts an element in the set. The **delete** operation deletes an element from the set. The **member** operation returns true if an element belongs to the set and false otherwise.

let S be an empty set
$S = S \cup \{x\}$
for all $x \in S$ **do**
    output x
    delete x from S

A dictionary is an associative container, consisting of a set of elements with both an information and a unique key associated with each element, where there is a linear order defined on the keys and the information associated with an element is retrieved on the basis of its key. The **member** operation returns true if there is an element with a given key in the dictionary, and false otherwise. The **lookup** operation returns the element with a given key in the dictionary, or *nil* if there is no such element. The **insert** operation inserts and returns an element with a given key and a given information in the dictionary, replacing the element (if any) with the given key. The **delete** operation deletes the element with a given key from the dictionary, if there is such an element.

let D be an empty dictionary
$D[x] = y$
for all $x \in D$ **do**
    $y = D[x]$
    output (x, y)
    delete x from D

REVIEW OF ELEMENTARY ALGORITHMS

The procedure LINEAR-SEARCH takes an array $A[1 : n]$ and a value x.

```
function LINEAR-SEARCH(A, x)
    i = 1
    while i ⩽ n and x ≠ A[i] do
        i = i + 1
    if i > n then
        return nil
    return i
```

The procedure BINARY-SEARCH takes a sorted array $A[1:n]$ and a value x.

```
function BINARY-SEARCH(A, x)
    return BINARY-SEARCH(A, x, 1, n)
```

The procedure BINARY-SEARCH takes a sorted array A, a value x, and a range $[low : high]$ of the array, in which we search for the value x.

```
function BINARY-SEARCH(A, x, low, high)
    while low ⩽ high do
        mid = ⌊(low + high)/2⌋
        if x = A[mid] then
            return mid
        else if x > A[mid] then
            low = mid + 1
        else
            high = mid − 1
    return nil
```

The procedure INSERTION-SORT sorts array $A[1:n]$.

```
procedure INSERTION-SORT(A)
    for i = 2 to n do
        key = A[i]
                                insert A[i] into the sorted subarray A[1 : i − 1]
        j = i − 1
        while j > 0 and A[j] > key do
            A[j + 1] = A[j]
            j = j − 1
        A[j + 1] = key
```

The procedure SELECTION-SORT sorts array $A[1 : n]$.

```
procedure SELECTION-SORT(A)
    for i = 1 to n − 1 do
        smallest = i
        for j = i + 1 to n do
            if A[j] < A[smallest] then
                smallest = i
        exchange A[i] with A[smallest]
```

The procedure BUBBLE-SORT sorts array $A[1:n]$.

```
procedure BUBBLE-SORT(A)
    for i = 1 to n − 1 do
        for j = n downto i + 1 do
            if A[j] < A[j − 1] then
                exchange A[j] with A[j − 1]
```