DATA STRUCTURES AND ALGORITHMS
LECTURE 1

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

Overview

- Course organization
- Abstract data types and Data structures
- Pseudocode
- Algorithm analysis

Course Organization I

- Guiding teachers
  - Lecturer PhD. Oneț-Marian Zsuzsanna
  - Lecturer PhD. Lupsa Dana
  - Lecturer PhD. Pop Andreea
  - PhD Student Mihai Andrei
- Activities
  - **Lecture:** 2 hours / week
  - **Seminar:** 1 hour / week
  - **Laboratory:** 1 hour / week

Course Organization II

- **Course page:** www.cs.ubbcluj.ro/˜marianzsu/DSA.html
- **Email:** marianzsu@cs.ubbcluj.ro
  - Please use your *scs.ubbcluj* email address for communication (and make sure the email contains your name!).

Course Organization III

- Grading
    - Written exam (**W**)
    - Lab grade (**L**)
    - Seminar grade (**S**)
        - Partial paper (**P**)
    - The final grade is computed as:

$$G = 0.6 * W + 0.2 * L + 0.2 * S$$

- **To pass the exam W, L and G has to be $\geq$ 5 (no rounding)!**

- Attendance is compulsory for the laboratory and the seminar activity. You need at least
  - 6 attendances from the 7 labs
  - 5 attendances from the 7 seminars.

- **Unless you have the required number of attendances, you cannot participate in the written exam, neither in the regular nor in the retake session!**

- The course page contains the link to the Google Sheets document where you can check your attendance situation.

Rules II

- You have to come to the lab and the seminar with your (semi)group (we will consider the official student lists for the groups from the faculty's web page).
- If you want to *permanently* switch from one group to another, you have to find a person in the other group who is willing to switch with you and announce your teacher about the switch in the first two weeks of the semester.
- One seminar and one laboratory attendance can be recovered with another group, within the two weeks allocated for the seminar/laboratory, with the explicit agreement of the seminar/laboratory teacher.

Rules III

- In case of illness, absences will be motivated by the lab/seminar teacher, based on a medical certificate. Medical certificates have to be presented in the first seminar/lab after the absence, certificates presented later than that will not be accepted.

Rules IV

- You will have a partial exam in the 5^{th} seminar. More details about this exam will be given in the 4^{th} seminar (and in Lecture 8).
- In the retake session only the written exam can be repeated, and grade G will be computed in the same way as in the regular session.
- If the lab grade L is not at least 5, you cannot participate at the written exam in the regular exam session. In the retake session you can participate at the written exam and you will have to turn in all the laboratory assignments, for a maximum grade of 5.

Course Objectives

- The study of the concept of *abstract data types* and the most frequently used abstract data types.
- The study of different *data structures* that can be used to implement these abstract data types and the complexity of their operations.

- What you should learn from this course:
    - to design and implement different applications starting from the use of abstract data types.
    - to process data stored in different data structures.
    - to choose the abstract data type and data structure best suited for a given application.

Bibliography I

- T. CORMEN, C. LEISERSON, R. RIVEST, C. STEIN, *Introduction to algorithms*, Third Edition, The MIT Press, 2009
- S. SKIENA, *The algorithm design manual*, Second Edition, Springer, 2008
- N. KARUMANCHI, *Data structures and algorithms made easy*, CareerMonk Publications, 2016
- M. A WEISS, *Data structures and algorithm analysis in Java*, Third Edition, Pearson, 2012

Bibliography II

- R. SEDGEWICK, *Algorithms*, Fourth Editions, Addison-Wesley Publishing Company, 2011
- R. LAFORE, *Data Structures & Algorithms in Java*, Second Edition, Sams Publishing, 2003
- M. A. WEISS *Data structures and problem solving using C++*, Second Edition, Pearson, 2003

Abstract Data Types

- What is a data type?

Abstract Data Types

- What is a data type?
- A *data type* is a set of values (domain of the data type) and a set of operations on those values (interface). For example:
  - int
    - set of values are integer numbers from a given interval
    - possible operations: add, subtract, multiply, etc.
  - boolean
    - set of values: true, false
    - possible operations: negation, and, or, xor, etc.
  - String
    - "abc", "text", etc.
    - possible operations: get the length, get a character from a position, get a substring, etc.
  - etc.

- An Abstract Data Type (ADT) is a *data type* having the following two properties:
  - the objects from the domain of the ADT are specified independently of their representation
  - the operations of the ADT are specified independently of their implementation

Abstract Data Types - Domain

- The domain of an ADT describes what elements belong to this ADT.
- If the domain is finite, we can simply enumerate them.
- If the domain is not finite, we will use a rule that describes the elements belonging to the ADT.

Abstract Data Types - Interface

- After specifying the domain of an ADT, we need to specify its operations.
- The set of all operations for an ADT is called its *interface*.
- The interface of an ADT contains the *signature* of the operations, together with their input data, results, preconditions and postconditions (but no detail regarding the implementation of the method).
- When talking about ADT we focus on the **WHAT** (it is, it does), not on the **HOW**.

ADT - Example I

- Consider the Date ADT (representing a date: year, month and day)
    - *Domain:* all the valid dates (year is positive, maybe less than 3000, month is between 1 and 12, day is between 1 and maximum number of possible days for the month)

ADT - Example II

- One possible operation for the Data ADT could be: difference(Date d1, Date d2)
  - **Descr:** computes the difference in number of days between two dates
  - **Pre:** d1 and d2 have to be valid Dates, d1 $\leq$ d2
  - **Post:** difference $\leftarrow$ d, d is a natural number and represents the number of days between d1 and d2
  - **Throws:** and exception if d1 is greater than d2

ADT - Example III

- Another possible operation could be: dayOfWeek(Date d)
  - **Descr:** returns as a string the day of the week of a given date
  - **Pre:** d is a Date
  - **Post:** dayOfWeek $\leftarrow$ *str*, where *str* is *"Monday"*, *"Tuesday"*, *"Wednesday"*, *"Thursday"*, *"Friday"*, *"Saturday"*, *"Sunday"* corresponding to the day of the week represented by the date d

- The above specifications contain everything we need to use the Date ADT, even if we know nothing about how it is represented or how the operation *difference* is implemented.

Container ADT I

- A *container* is a collection of data, in which we can add new elements and from which we can remove elements.
- Different containers are defined based on different properties:
  - do the elements need to be unique?
  - do the elements have positions assigned?
  - can any element be accessed or just some specific ones?
  - do we store simple elements or key - value pairs?

Container ADT II

- A container should provide at least the following operations:
  - *creating* an empty container
  - *adding* a new element to the container
  - *removing* an element from the container
  - returning the *number of elements* in the container
  - provide *access to the elements* from the container (usually using an *iterator*)

Container vs. Collection

- Python - Collections
- C++ - Containers from STL
- Java - Collections framework and the Apache Collections library
- .Net - System.Collections framework
- In the following, in this course we will use the term **container**.

Why *abstract* data types?

- There are several different Container Abstract Data Types, so choosing the most suitable one is an important step during application design.
- When choosing the suitable ADT we are not interested in the implementation details of the ADT (yet).
- Most high-level programming languages usually provide implementations for different Abstract Data Types.
    - In order to be able to use the right ADT for a specific problem, we need to know their domain and interface.

Example

- Assume that you have to write an application to handle Roman Numerals (maybe transform them into Arabic numerals, or transform Arabic numerals into Roman ones, for example CXXI into 121 or 529 into DXXIX).
- Maybe you want to define some new "numerals" as well, for example, W to represent the number 200.
- In order to implement this application you will often need to find the number corresponding to a given letter and vice-versa.
- What container would you use?

Advantages of working with ADTs I

- *Abstraction* is defined as the separation between the specification of an object (its domain and interface) and its implementation.

- *Encapsulation* - abstraction provides a promise that any implementation of an ADT will belong to its domain and will respect its interface. And this is all that is needed to use an ADT.

Advantages of working with ADTs II

- *Localization of change* - any code that uses an ADT is still valid if the ADT changes (because no matter how it changes, it still has to respect the domain and interface).

- *Flexibility* - an ADT can be implemented in different ways, but all these implementation have the same interface. Switching from one implementation to another can be done with minimal changes in the code.

Advantages of working with ADTs III

- Consider again our example with the Date ADT:
  - Assume you used variables of type Date and the *difference* operation in your code.
  - If the implementation of the Date changes, these changes will not affect your code, because even if the representation changes and even if the implementation of the operations changes, they still have to respect the domain (it still has to represent a date made by year, month, day) and interface (the signature and behaviour of the *difference* operation is still the same).
  - If a new implementation for the Date will appear and you want to switch to that implementation, it can be done with minimal modifications in your code

Data Structures I

- The domain of data structures studies how we can store and access data.
  - When we want to implement a container, we need to decide where the elements are going to be stored in the memory and how we can work with them $\Rightarrow$ we need a data structure
- A data structure can be:
  - Static: the size of the data structure is fixed. Such data structures are suitable if it is known that a fixed number of elements need to be stored.
  - Dynamic: the size of the data structure can grow or shrink as needed by the number of elements.

- For every ADT we will discuss several possible data structures that can be used for the implementation. For every possibility we will discuss the advantages and disadvantages of using the given data structure. We will see that, in general, we cannot say that there is one single *best* data structure.

- Why do we need to implement our own Abstract Data Types if they are readily implemented in most programming languages?

- Why do we need to implement our own Abstract Data Types if they are readily implemented in most programming languages?
  - Implementing these ADT will help us understand better how they work (we cannot use them if we do not know what they are doing)
  - To learn to create, implement and use ADT for situations when:
    - we work in a programming language where they are not readily implemented.
    - we need an ADT which is not part of the standard ones, but might be similar to them.

Pseudocode I

- The aim of this course if to give a general description of data structures, one that does not depend on any programming language - so we will use the *pseudocode* language to describe the algorithms.
- Our algorithms written in pseudocode will consist of two types of instructions:
  - standard instructions (assignment, conditional, repetitive, etc.)
  - non-standard instructions (written in plain English to describe parts of the algorithm that are not developed yet). These non-standard instructions will start with @.

Pseudocode II

- One line comments in the code will be denoted by //
- For reading data we will use the standard instruction **read**
- For printing data we will use the standard instruction **print**
- For assignment we will use $\leftarrow$
- For testing the equality of two variables we will use $=$

Pseudocode III

- Conditional instruction will be written in the following way (the *else* part can be missing):

if condition **then**
  @instructions
else
  @instructions
end-if

Pseudocode IV

- The *for* loop (loop with a known number of steps) will be written in the following way:

for i $\leftarrow$ init, final, step **execute**
  @instructions
end-for

- *init* - represents the initial value for variable i
- *final* - represents the final value for variable i
- *step* - is the value added to i at the end of each iteration. *step* can be missing, in this case it is considered to be 1.

- The *while* loop (loop with an unknown number of steps) will be written in the following way:

while condition **execute**
  @instructions
end-while

Pseudocode VI

- Subalgorithms (subprograms that do not return a value) will be written in the following way:

subalgorithm name(formal parameter list) **is:**
  @instructions - subalgorithm body
end-subalgorithm

- The subalgorithm can be called as:

name (actual parameter list)

Pseudocode VII

- Functions (subprograms that return a value) will be written in the following way:

function name (formal parameter list) **is:**
  @instructions - function body
  name $\leftarrow$ v *//syntax used to return the value v*
end-function

- The function can be called as:

result $\leftarrow$ name (actual parameter list)

Pseudocode VIII

- If we want to define a variable i of type Integer, we will write: $i : Integer$
- If we want to define an array a, having elements of type T, we will write: $a : T[]$
  - If we know the size of the array, we will use: $a : T[Nr]$
  - If we do not know the size of the array, we will use: $a : T[]$
  - **Obs:** In pseudocode indexing in an array in general starts from 1.

Pseudocode IX

- A struct (record) will be defined as:

Array:
    n: Integer
    elems: T[]

- The above struct consists of 2 fields: *n* of type Integer and an array of elements of type T called *elems*
- Having a variable *var* of type Array, we can access the fields using . (dot):
    - var.n
    - var.elems
    - var.elems[i] - the i-th element from the array

Pseudocode X

- For denoting pointers (variables whose value is a memory address) we will use ↑:
  - p: ↑ Integer - *p* is a variable whose value is the address of a memory location where an Integer value is stored.
  - The value from the address denoted by *p* is accessed using [*p*]
- Allocation and de-allocation operations will be denoted by:
  - allocate(p)
  - free(p)
- We will use the special value NIL to denote an invalid address

- An operation will be specified in the following way:
  - **pre:** - the preconditions of the operation
  - **post:** - the postconditions of the operation
  - **throws:** - exceptions thrown (optional - not every operation throws an exception)
- When using the name of a parameter in the specification we actually mean its value.
- Having a parameter i of type T, we will denote by $i \in T$ the condition that the value of variable i belongs to the domain of type T.

Specifications II

- The value of a parameter can be changed during the execution of a function/subalgorithm. To denote the difference between the value before and after execution, we will use the ' (apostrophe).

- For example, the specification of an operation *decrement*, that decrements the value of a parameter x (x : *Integer*) will be:
  - **pre:** x $\in$ Integer
  - **post:** x' = x - 1

Generic Data Types I

- We will consider that the elements of an ADT are of a generic type: *TElem*
- The interface of the *TElem* contains the following operations:
  - assignment ($e_1 \leftarrow e_2$)
    - **pre:** $e_1, e_2 \in TElem$
    - **post:** $e_1' = e_2$
  - equality test ($e_1 = e_2$)
    - **pre:** $e_1, e_2 \in TElem$
    - **post:**

$$equal = \begin{cases} True, & \text{if } e_1 \text{ } equals \text{ } e_2 \\ False, & \text{otherwise} \end{cases}$$

- Some of the containers are *sorted*: the elements have to be in a specific order. For such cases we will use either:
  - TComp, where, besides the operations for the TElem, we have $e_1 < e_2$, $e_1 \leq e_2$, $e_1 = e_2$, $e_1 > e_2$, $e_1 \geq e_2$
  - TElem/TComp + a relation
    - compare(e_1, e_2)
    - **pre:** $e_1, e_2 \in TElem$
    - **post:**

$$compare = \begin{cases} -1, & \text{if } e_1 < e_2 \\ 0, & \text{if } e_1 = e_2 \\ 1 & \text{if } e_1 > e_2 \end{cases}$$

The RAM model I

- Analyzing an algorithm usually means predicting the resources (time, memory) the algorithm requires. In order to do so, we need a hypothetical computer model, called *RAM* (random-access machine) model.
- In the RAM model:
  - Each simple operation ($+$, -, *, $/$, $=$, if, function call) takes one time step/unit.
  - We have fixed-size integers and floating point data types.
  - Loops and subprograms are *not* simple operations and we do not have special operations (ex. sorting in one instruction).
  - Every memory access takes one time step and we have an infinite amount of memory.

The RAM model II

- The RAM is a very simplified model of how computers work, but in practice it is a good model to understand how an algorithm will perform on a real computer.

- Under the RAM model we measure the run time of an algorithm by counting the number of steps the algorithm takes on a given input instance. The number of steps is usually a function that depends on the size of the input data.

```
subalgorithm something(n) is:
//n is an Integer number
   rez ← 0
   for i ← 1, n execute
      sum ← 0
      for j ← 1, n execute
         sum ← sum + j
      end-for
      rez ← rez + sum
   end-for
   print rez
end-subalgorithm
```

- How many steps does the above subalgorithm take?

```
subalgorithm something(n) is:
//n is an Integer number
   rez ← 0
   for i ← 1, n execute
      sum ← 0
      for j ← 1, n execute
         sum ← sum + j
      end-for
      rez ← rez + sum
   end-for
   print rez
end-subalgorithm
```

- How many steps does the above subalgorithm take?
- T(n) = 1 + n * (1 + n + 1) + 1 = $n^2 + 2n + 2$

Order of growth

- We are not interested in the exact number of steps for a given algorithm, we are interested in its *order of growth* (i.e., how does the number of steps change if the value of n increases)
- We will consider only the leading term of the formula (for example n^2), because the other terms are relatively insignificant for large values of n.

O-notation I

O-notation

For a given function $g(n)$ we denote by $O(g(n))$ the set of functions:
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 s. t.
$0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0\}$

- The O-notation provides an *asymptotic upper bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is on or below $c \cdot g(n)$.
- We will use the notation $f(n) = O(g(n))$ or $f(n) \in O(g(n))$.

- Graphical representation:

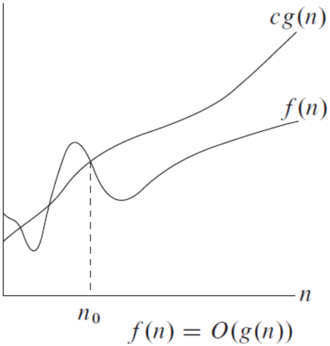

Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

O-notation III

Alternative definition

$$f(n) \in O(g(n)) \text{ if } \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

is either 0 or a constant (but not ∞).

- Consider, for example, $T(n) = n^2 + 2n + 2$:
  - $T(n) = O(n^2)$ because $T(n) \leq c \cdot n^2$ for c = 2 and n $\geq$ 3
  - $T(n) = O(n^3)$ because

$$\lim_{n \to \infty} \frac{T(n)}{n^3} = 0$$

Ω-notation I

Ω-notation

For a given function $g(n)$ we denote by $\Omega(g(n))$ the set of functions:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 s. t.

$$0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0\}$$

- The Ω-notation provides an *asymptotic lower bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is on or above $c \cdot g(n)$.
- We will use the notation $f(n) = \Omega(g(n))$ or $f(n) \in \Omega(g(n))$.

Ω-notation II

- Graphical representation:

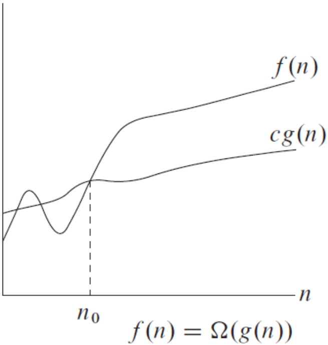

Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

Ω-notation III

Alternative definition

$$f(n) \in \Omega(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

is ∞ or a nonzero constant.

- Consider, for example, $T(n) = n^2 + 2n + 2$:
  - $T(n) = \Omega(n^2)$ because $T(n) \geq c \cdot n^2$ for c $=$ 0.5 and n $\geq$ 1
  - $T(n) = \Omega(n)$ because

$$\lim_{n\to\infty} \frac{T(n)}{n} = \infty$$

Θ-notation

For a given function $g(n)$ we denote by $\Theta(g(n))$ the set of functions:
$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 s. t.
$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0\}$

- The Θ-notation provides an *asymptotically tight bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$.
- We will use the notation $f(n) = \Theta(g(n))$ or $f(n) \in \Theta(g(n))$.

Θ-notation II

- Graphical representation:

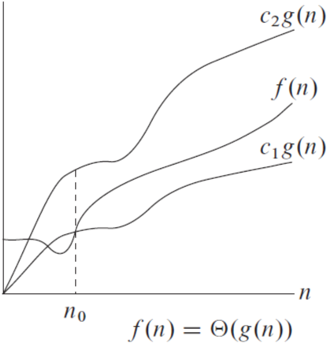

Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

Θ-notation III

Alternative definition

$$f(n) \in \Theta(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

is a nonzero constant (and not ∞).

- Consider, for example, $T(n) = n^2 + 2n + 2$:
  - $T(n) = \Theta(n^2)$ because $c_1 \cdot n^2 \leq T(n) \leq c_2 \cdot n^2$ for $c_1 = 0.5$, $c_2 = 2$ and n ≥ 3.
  - $T(n) = \Theta(n^2)$ because

$$\lim_{n\to\infty} \frac{T(n)}{n^2} = 1$$

- Think about an algorithm that finds the sum of all even numbers in an array. How many steps does the algorithm take for an array of length n?

- Think about an algorithm that finds the sum of all even numbers in an array. How many steps does the algorithm take for an array of length n?
- Think about an algorithm that searches for a given element, e, in an array. How many steps does the algorithm take for an array of length n?

- For the second problem the number of steps taken by the algorithm does not depend just on the length of the array, it depends on the exact values from the array as well.
- For an array of fixed length n, the execution of the algorithm can stop:
  - after verifying the first number - if it is equal to e
  - after verifying the first two numbers - if the first is not e, but the second is equal to e
  - after verifying the first 3 numbers - if the first two are not equal to e and the third is e
  - ...
  - after verifying all n numbers - first $n - 1$ are not e and the last is equal to e, or none of the numbers is e

Best Case, Worst Case, Average Case II

- For such algorithms we will consider three cases:
  - Best - Case - the best possible case, where the number of steps taken by the algorithm is the minimum that is possible
  - Worst - Case - the worst possible case, where the number of steps taken by the algorithm is the maximum that is possible
  - Average - Case - the average of all possible cases.
- Best and Worst case complexity is usually computed by inspecting the code. For our example we have:
  - Best case: $\Theta(1)$ - just the first number is checked, no matter how large the array is.
  - Worst case: $\Theta(n)$ - we have to check all the numbers

- For computing the average case complexity we have a formula:

$$\sum_{I \in D} P(I) \cdot E(I)$$

- where:
  - D is the domain of the problem, the set of every possible input that can be given to the algorithm.
  - I is one input data
  - $P(I)$ is the probability that we will have I as an input
  - $E(I)$ is the number of operations performed by the algorithm for input I

Best Case, Worst Case, Average Case IV

- For our example D would be the set of all possible arrays with length n
- Every I would represent a subset of D:
  - One I represents all the arrays where the first number is e
  - One I represents all the arrays where the first number is is not e, but the second is e
  - ...
  - One I represents all the arrays where the first $n-1$ elements are not e, but the last is equal to e
  - One I represents all the arrays which do not contain e
- P(I) is usually considered equal for every I, in our case $\frac{1}{n+1}$

$$T(n) = \frac{1}{n+1}\sum_{i=1}^{n} i + \frac{n}{n+1} = \frac{n \cdot (n+1)}{2 \cdot (n+1)} + \frac{n}{n+1} \in \Theta(n)$$

Best Case, Worst Case, Average Case V

- When we have best case, worst case and average case complexity, we will report the maximum one (which is the worst case), but if the three values are different, the total complexity is reported with the O-notation.

- For our example we have:
  - Best case: $\Theta(1)$
  - Worst case: $\Theta(n)$
  - Average case: $\Theta(n)$
  - Total (overall) complexity: $O(n)$