- Slides to support lectures
- Notes
- Slides from 2020-2021

Lecture 1

- Introduction
- Abstract data types and Data structures
- Collection / containers

➢ Pseudocode
- Algorithm analysis

Lect. PhD. Lupsa Dana
Babes - Bolyai University
Computer Science and Mathematics Faculty
2021 - 2022

Many slides borrowed from: Lect. PhD. Onet-Marian Zsuzsanna

Course Objectives

- The study of data types
  - The study of the concept of **abstract data types**

  and the most frequently used container abstract data types.
  - The study of different **data structures** that can be used to implement these abstract data types and their properties (including 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 choose the abstract data type and data structure best suited for a given application.
- to work with data stored in different data structures.

Why we should try to create our own implementations for some ADTs if some languages already give us implementations of ADT when we need it?

- To learn how to implement ADTs

  and understand what's behind the scene;
- to learn the behaviour of some of the most used DS - ADT pairs

Bibliography

- 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
- 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

Data Type & Abstract Data Type

A data type (DT) is a set of values (domain) and a set of operations on those values. For example:

- built-in data types
- user defined data types.

Examples ?

An Abstract Data Type (ADT) :

- the objects from the domain
  - specified independently of their representation
- the operations
  - procedural abstraction → operation specification
  - specified independently of their implementation

Example

Consider the Date ADT

(made of three numbers: year, month, day)

- Elements belonging to this ADT are all the valid dates, as follows:

  year is positive, less than 3000, month is between 1 and 12, day is between 1 and maximum number of possible days for the month

One possible operation for the Date ADT could be:

difference(d1, d2)

- **Descr**: compute the difference in number of days between two dates
- **Pre**: d1 and d2 have to be valid Dates, d1 <= d2
- **Post**: difference = a natural number and represents the number of days between d1 and d2.
- **Throws**: and exception if d1 is greater than d2

Working with ADTs …

- Abstraction

  Separation between the specification of an object (its domain and interface) and its implementation
- Encapsulation

  … implementation ?
- Localization of change
- Flexibility

- We have code that uses an ADT. Is it still valid if the ADT changes ?
- An ADT can be implemented in different ways. How hard is it to switch from one implementation to another ?

Container

A container is a collection of elements.

- There we can add new elements and/or 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?

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
- 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.

You are already familiar with: list and dict from Python … as ADT!

- Did you know that in Python the following instruction has a different complexity depending on whether cont is a list or a dict?

```
if elem in cont:
    print("Found")
```

- Implementing these ADT will help us understand better how they work

Container: ADT , DS , pseudocode

- For every container 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 for a given container.
- 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.

See: Lecture01_pseudocode.pdf

ADT, DS and levels of abstraction

different levels of abstraction between domain & representation

ADT implementation

Example:

ADT domain	container with elements	
refinement 0	list of elements	<- ADT List
refinement 1	singly linked list of elements	
refinement 2	singly linked list with dynamically allocated nodes	
DS (representation)	…	

Remarks

1. Refinements involves (permits) some levels of abstraction. For example, we can study the: “single linked list” regardless of the fact that it is dynamically allocated or it has a semi-static representation.
2. The fundamental building blocks for data structures are provided by languages.

(arrays, records, pointers/references)

Analyzing algorithm

- Algorithms, like computer hardware, are ***technology***.
  Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

- Predict the resources that the algorithm requires
  - computational running time
  - memory
  - communication bandwidth **…**

➢ Algorithm efficiency

→ **time**

→ **space**

Many sorting algorithms - example

- bubble sort - before 1960
- insertion sort - before 1960
- variant of insertion sort:
  shell sort - Donald Shell, 1959
- QuickSort – Hoare, 1960
- HeapSort
  smoothsort a variation of heapsort
  developed by Dijkstra in 1981
- …
- QiSort - Matthew Copeland, 2007

Algorithm analysis

- Algorithm efficiency → **time**

  how we measure
  - asymptotic analysis ***running time complexity***

    mathematical analysis of the algorithms

    (~ the limit)
  - empirical analysis ***exact running time***

    determine exact running time
    - fixed implementation
    - fixed input

Algorithm analysis

We must assume a model of the implementation to use.

The RAM (random-access machine) model

- It is a hypothetical computer model, a very simplified model
- 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.

In the RAM model:

- Each simple operation (+, -, *, /, =, if, function call) takes one time step/unit.
- 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.
- There are fixed-size integers and floating point data types.

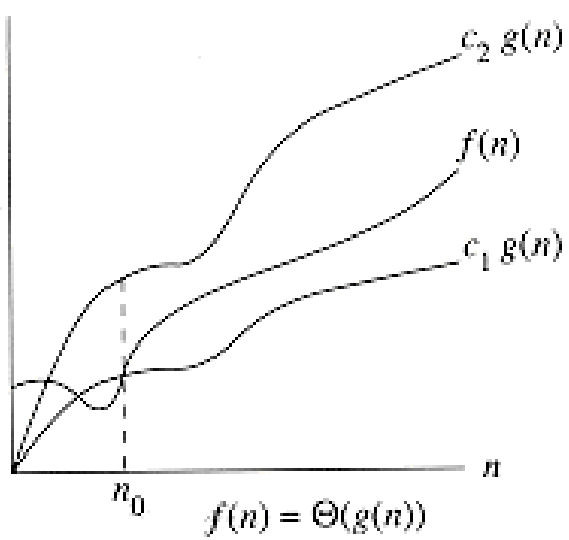
Algorithm analysis. Example

```
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
```

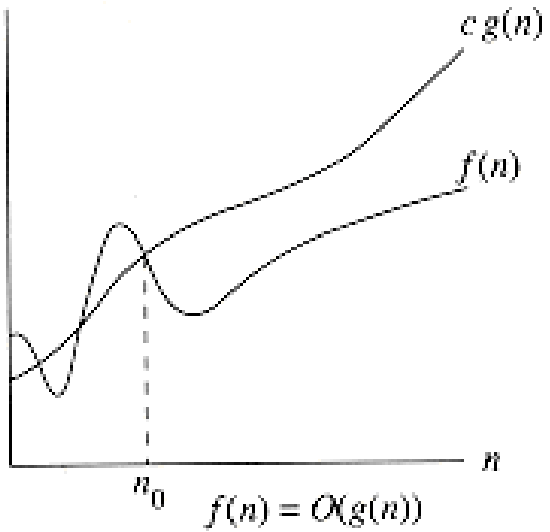
- T(n) = ?
- order of growth ?
  - use leading term of T(n)

- How many steps does the above subalgorithm take?

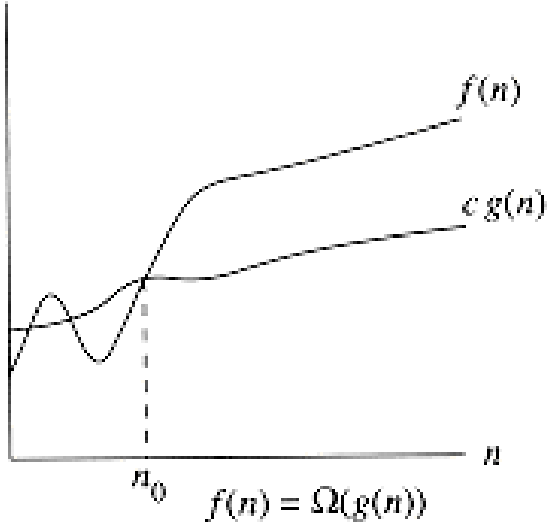
Asymptotic analysis



(a)



(b)



(c)

O-notation

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 $n \geq n_0\}$

Graphical representation:

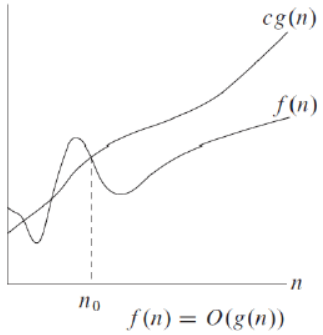


Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

- We will use the notation $f(n) = O(g(n))$ or $f(n) \in O(g(n))$.

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*g(n).

Alternative definition

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

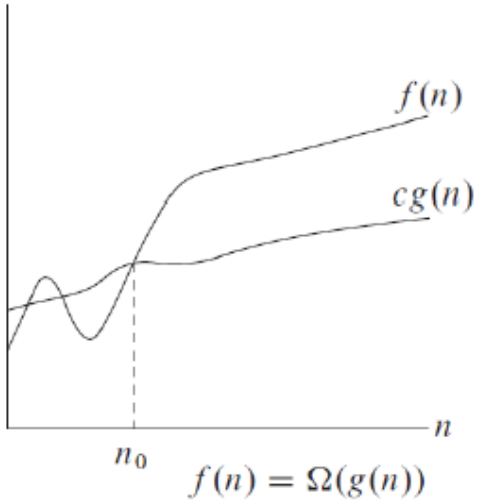
is either 0 or a constant (but not ∞).

Consider, for example, $T(n) = n^2 + 2n + 2$:

- $T(n) = O(n^2)$ because $T(n) \leq c * n^2$ 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



(source: Cormen)

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. $0 \leq c \cdot g(n) \leq f(n)$ 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 $f(n) \in \Omega(g(n))$.

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 $T(n) \geq c * n^2$ for c = 0.5 and n ≥ 1
- $T(n) = \Omega(n)$ because

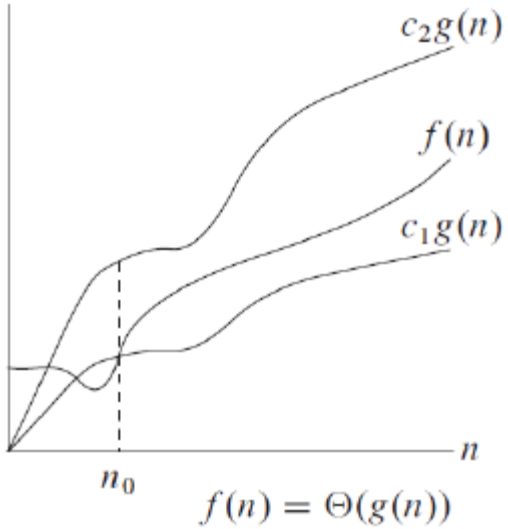
$$\lim_{n\to\infty} \frac{T(n)}{n} = \infty$$

Θ-notation

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 $n \geq n_0\}$



(source: Cormen)

- 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 $f(n) \in \Theta(g(n))$.

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 $c_1 * n^2 \leq T(n) \leq c_2 * n^2$ for $c_1 = 0.5$, $c_2 = 2$ and n ≥ 3.
  - $T(n) = \Theta(n^2)$ because

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

Best Case, Worst Case, Average Case

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?

- … an algorithm that finds the first occurrence of a number in an array.

  How many steps does the algorithm take for an array of length n?

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.

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

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

Average Case. Example

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

Think about:

- … an algorithm that finds the first occurrence of a number in an array.

How many steps does the algorithm take for an array of length n?

D would be the set of all possible arrays with length n ; I = ?

- Case 1:
  - all the arrays where the first number is the one we are looking for
- Case 2:
  - all the arrays where the first number is not the one we are looking for, but the second is
- ...
- Case n:
  - all the arrays where the first n − 1 elements are not the one we are looking for, but the last one is
- Case n+1:
  - all the arrays which do not contain the element we are looking for
- P(I) is usually considered equal for every I

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

How to compute complexity in recursion. Example

- In case of a recursive algorithm, complexity computation starts from the recursive formula of the algorithm
- Example:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2 * T(\frac{n}{2}) + n, & \text{otherwise} \end{cases}$$

Let $n = 2^k$

Ignoring the parameter x we rewrite the recursive branch:

$$T(2^k) = 2 * T(2^{k-1}) + 2^k$$
$$2 * T(2^{k-1}) = 2^2 * T(2^{k-2}) + 2^k$$
$$2^2 * T(2^{k-2}) = 2^3 T(2^{k-3}) + 2^k$$
$$\dots$$
$$2^{k-1} * T(2) = 2^k * T(1) + 2^k$$

$$T(2^k) = 2^k * T(1) + k * 2^k$$

$T(1) = 1$ (base case from the recursive formula)

$$T(2^k) = 2^k + k * 2^k$$

Let's go back to the notation with n.

If $n = 2^k \Rightarrow k = log_2 n$

$$T(n) = n + n * log_2 n \in \Theta(n log_2 n)$$

How to compute complexity in recursion. Example

Consider the next algorithm.

Decide the recursion formula for it : T(n)

- Assume that crossMiddle(x,a,b) has T(x, a, b) = b - a

```
function fromInterval(x, left, right) is:
//x is an array of integer numbers
//left and right are the boundaries of the subsequence
   if left = right then
      fromInterval ← x[left]
   end-if
   middle ← (left + right) / 2
   justLeft ← fromInterval(x, left, middle)
   justRight ← fromInterval(x, middle+1, right)
   across ← crossMiddle(x, left, right)
   fromInterval ← @maximum of justLeft, justRight, across
end-function
```

Please See Lecture01__extraReading.pdf