DATA STRUCTURES

Abstract Data Types. Data Structures. Complexities.

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University, Faculty of Mathematics and Computer Science

Overview

- Course organization
- Abstract Data Types and Data Structures
- Pseudocode conventions
- Complexities

Course Organization

- **Guiding teachers**
  - Lect. PhD. Diana-Lucia Miholca
  - Assist. PhD. Anamaria Briciu
  - PhD. Student Alexandra Albu
- **Activities**
  - **Lecture:** 2 hours / week
  - **Seminar:** 2 hours / 2 weeks
  - **Lab:** 2 hours / 2 weeks

Evaluation. Grading

- **Grading**
  - **Written exam** (**W**) - in the last week of the semester
  - **Lab grade** (**LG**) - weighted average of grades received for the lab assignments (more details in the *Lab description and rules.pdf* document)
  - **Seminar bonus** (**B**) - maximum 0.5 points bonus, received for the seminar activity.
  - The **final grade (G)** is computed as:

$$G = 0.7 * W + 0.3 * LG + B$$

Platform used during the semester

- **MS Teams**
  - Team code: **jgwk6d9**
  - All the materials (lecture slides, lab requirements, etc.) will be uploaded in the *General* channel under the *Files* section.
  - You can use MS Teams private chat for communicating with teachers. Alternatively, you can communicate via e-mail:
    - ✉ diana.miholca@ubbcluj.ro
    - ✉ anamaria.briciu@ubbcluj.ro
    - ✉ alexandra.albu@ubbcluj.ro

Rules. Attendance

- Attendance is compulsory for the laboratory and the seminar activity. You need at least:
  - 6 out of 7 attendances for the laboratory
  - 5 out of 7 attendances for the seminar

Unless you have the required minimum number of attendances, you cannot participate in the written exam, neither in the regular nor in the retake session.

- In MS Teams, under *Files* $\rightarrow$ *Class materials*, there is a document (*Attendance Sheet Link.pdf*) with a link to a sheet where you can check your attendance situation, lab assignments, lab grades and seminar activity, based on you AcademicInfo ID.

Rules. Attending with other group

- You have to come to the seminar with your group and to the lab with your semi-group.
- If you want to *permanently* switch from one (semi-)group to another, you have to find a person in the other (semi-)group who is willing to switch with you and announce your lab/seminar teacher about the switch in the first two weeks of the semester.
- One seminar can be recovered with another group, within the two weeks allocated for the seminar, with the explicit agreement of the seminar teacher.

Rules. Absences in case of illness

- 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 / sent by e-mail before the first seminar/lab after the absence.

Rules. Retake session.

- In the retake session only the written exam can be repeated.
- The lab grade, **LG**, and the seminar bonus, **B**, correspond to activities carried out during the semester and, therefore, they cannot be increased.

Course objectives

- The study of the concept of **Abstract Data Types** and of 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 of the **complexity** of their operations.

Bibliography

- T. CORMEN, C. LEISERSON, R. RIVEST, C. STEIN, *Introduction to algorithms*, Third Edition, The MIT Press, 2009
- N. KARUMANCHI, *Data structures and algorithms made easy*, CareerMonk Publications, 2016
- A. CLIFFORD, A. SHAFFER, *A practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010
- R. SEDGEWICK, *Algorithms*, Fourth Editions, Addison-Wesley Publishing Company, 2011
- S. SKIENA, *The algorithm design manual*, Second Edition, Springer, 2008
- M. A. WEISS *Data structures and problem solving using C++*, Second Edition, Pearson, 2003

Abstract Data Types. Data Structures. Algorithms.

Representing real world objects in a computer program entails:

- **Modeling**:
  -  How we model the real word objects as abstract mathematical entities and basic data types?
- **Operations**:
  -  Which are the operations we can use to store, access, and manipulate these entities?
- **Representation**:
  -  How can we concretely represent these entities in a computer's memory?
- **Algorithms**:
  -  Which are the algorithms that are used to perform these operations?

 The first two items encapsulate the essence of an **Abstract Data Type**. The third item refers to **Data Structures**, while the last one deals with **algorithms**.

Abstract Data Types

🄳 An **Abstract Data Type** (ADT) is an abstract mathematical entity consisting of a set of objects together with a set of operations and having the following two properties:

① the objects from the domain of the ADT are specified independently of their representation

② the operations of the ADT are specified independently of their implementation

Abstract Data Types - Domain

 The **domain** of an ADT is the set of all the objects of that 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

 The set of all operations of an ADT is called its **interface**.

 The interface of an ADT contains:

- the **signature** of the operations,
- the **preconditions** of the operations,
- the **postconditions** of the operations,
- **no** details regarding the implementation.

 When describing an ADT we focus on the **WHAT** (it is, it does), not on the **HOW** (it is represented, it is implemented).

ADT - Example

Consider the **Date** ADT.

① **Define ADT's domain**: describe the set of all valid dates

② **Define the ADT's interface**:

One possible operation could be: difference(Date d1, Date d2)

- **Description:** 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 and / or d2 are not valid or d1 is after d2

ADT - Example

 The definition of the domain and the definition of the interface give us everything we need to know in order to use the Date ADT, even if we know nothing about how it is represented in a computer's memory or how the operation *difference* is implemented.

 **Other examples** of ADTs: Stack, Queue, Set, Bag, List, Map (all containers), etc.

Container ADT

 A **container** is a collection of elements of the same other type.

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 single elements or (key, value) pairs?

Container ADT

Container-specific 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
- providing **access to the elements** from the container

Container ADT

 During the semester we are going to talk about many different containers.

 There are two containers that you are already familiar with from Python: ***list*** and ***dict***.

- myList = ["data", "structures", "algorithms"]
- myDict Dict = {1: "data", 2: "structures", 3: "algorithms"}

 Moreover, you know them and you have used them as ADT.

 Do you know how a *list* or a *dict* is represented?

 Do you know how their operations are implemented?

 Did not knowing these stop you from using them?

Container ADT

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.

Advantages of working with ADTs

 **Abstraction** - there is separation between the specification of an ADT (its domain and interface) and its representation and implementation.

 **Encapsulation** - the details (regarding how the data type will be implemented) are hidden from the user of the ADT.

Advantages of working with ADTs

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

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

Advantages of working with ADTs

Consider our example with the **Date** ADT. Assume that we:

- Have decided to represent the Date by using 3 integer numbers: one for the year, one for the month and one for the day.
- Have defined a lot of operations for Date: to compute the difference between two dates, to check if a date is a weekend day, to add a given number of days to a date, etc.
- Have also defined an ADT to represent a **DateInterval**, which contains two Dates: the start and the end of the interval.
- Have implemented a list of operations for **DateInterval**: check if two intervals overlap, check if an interval includes a specific day, create an interval starting from a date and a number of days, etc.
- Have implemented a complex application for scheduling holidays (DateInterval) and setting up deadlines for tasks (Date) and all sort of similar functionalities.

Advantages of working with ADTs

 What happens if now we want to change the representation of the **Date**?

 Instead of storing 3 numbers, we choose to store a date as a number with 8 digits, first 4 digits representing the year, the next corresponding to the month and the last two indicating the day.

 How much of our code do we need to change?

Advantages of working with ADTs

 The code implementing the ADT Date (including all the operations defined for it) needs to be rewritten.

 If we worked with Date as an ADT, even if the representation and the implementation of the operations change, they still have to respect the interface. Consequently, the rest of the application is not affected by the changes.

Data Structures

D A data structure is a specific way of representing and organizing data in a computer's memory.

E Examples: Arrays, Linked Lists, Trees, etc.

Data Structures in relation to Abstract Data Types

 For every container ADT we will discuss several possible data structures that can be used for its implementation.

- For every possibility, we will discuss the advantages and disadvantages.
- We will see that, in general, we cannot say that there is one single *best* data structure for a given container.

Why implementing ADTs?

 Why do we need to implement our own ADTs if they are already implemented in most programming languages?

 Implementing these ADT will help us better understand how they work (we cannot use them, if we do not know what they are doing)

 To learn to design, implement and use ADTs for situations when:

- we work in a programming language where they are not already implemented.
- we need an ADT which is not part of the standard ones, but might be similar to one of them.

Pseudocode conventions

🤝 The aim of this course if to give a general description, one that does not depend on any programming language, so we will use **pseudocode**.

🍴 Our algorithms written in pseudocode will consist of two types of instructions:

- standard instructions
  - assignments, conditional/repetitive instructions, etc.
- non-standard instructions
  - written in plain English and will start with @.

Pseudocode conventions

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

Pseudocode conventions

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

The conditional statement:

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

Pseudocode conventions

 The *for* loop will be written in the following way:

The *for* loop:

for i ← init, final, step **execute**
    @instructions
end-for

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

Pseudocode conventions

🤝 The *while* loop will be written in the following way:

The *while* loop:

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

Pseudocode conventions

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

Defining subalgorithms:

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

🤝 The subalgorithm can be called as:

Calling subalgorithms:

name(actual parameter list)

Pseudocode conventions

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

Defining functions:

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

🤝 The function can be called as:

Calling functions:

result ← name(actual parameter list)

Pseudocode conventions

🤝 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: $a : T[Nr]$ - indexing is done from 1 to Nr
- If we do not know the size of the array, we will use: $a : T[]$ - indexing will start from 1

Pseudocode conventions

 A struct (record) will be defined as:

Declaring a struct (record):

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 *v* of type Array, we can access its fields using . (dot):

- v.n
- v.elems
- v.elems[i] - the i-th element from the array

Pseudocode conventions

🤝 For denoting pointers 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

Conventions for specifications

 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 have to throw an exception)

 When using the name of a parameter in the specification we actually mean its value.

 Having a parameter i of an ADT X having the domain T, we will denote by $i \in T$ the condition that the value of variable i is of type X.

Conventions for specifications

🤝 Since the value of a parameter can be changed during the execution of an operation, to refer to the value of the parameter after the execution, we will use the ’ (apostrophe).

🇪 The specification of an operation *decrement*, that just decrements the value of the parameter *x* (*x* : *Integer*):

- **pre:** x: Integer
- **post:** x’ = x - 1

Generic Data Types

 We will consider that the elements of a container 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 testing ($e_1 = e_2$)
  - **pre:** $e_1, e_2 \in TElem$
  - **post:**

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

Generic Data Types

 We will consider that the elements of a sorted container ADT are of a generic comparable type: *TComp*, whose values can be compared or ordered based on a relation.

 Besides the operations from TElem, TComp has an extra operation that compares two elements:

- compare(e_1, e_2)
  - **pre:** $e_1, e_2 \in TComp$
  - **post:**

$$compare(e_1, e_2) \leftarrow \begin{cases} true & \text{if } e_1 <= e_2 \\ false & \text{if } e_1 > e_2 \end{cases}$$

 For simplicity, we will often use the notations $e_1 \leq e_2$, $e_1 \geq e_2$ instead of calling the *compare* function

Complexities

How do we define the efficiency of an algorithm?

The efficiency of an algorithm is given by the quantity of resources it requires, in terms of:

- **time**
- memory

How do we measure the efficiency of an algorithm?

- empirical (experimental) analysis
- **asymptotic** (mathematical) analysis

The RAM model

 To analyze algorithms mathematically, we need a hypothetical computer model, called **RAM** (random-access machine).

 In the RAM model:

- Each simple operation (+, -, *, /, =) takes one time unit.
- Loops and subprograms are *not* simple operations
- Every memory access takes one time unit.

The RAM model

The RAM model is a very simplified model of how computers work, but in practice it is a good enough for understanding 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. The number of steps is usually a function depending on the size of the input.

The RAM model

Example subalgorithm:

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

How many steps does the above subalgorithm take?

The RAM model

Example subalgorithm:

```
subalgorithm something(n) is: //n is an Integer number
    result ← 0
    for i ← 1, n execute
        sum ← 0
        for j ← 1, n execute
            sum ← sum + j
        end-for
        result ← result + sum
    end-for
    print result
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 but in *order of growth* of the complexity function.

 We will consider only the leading term of the formula, the other terms being relatively insignificant for large values of n.

O-notation

O-notation

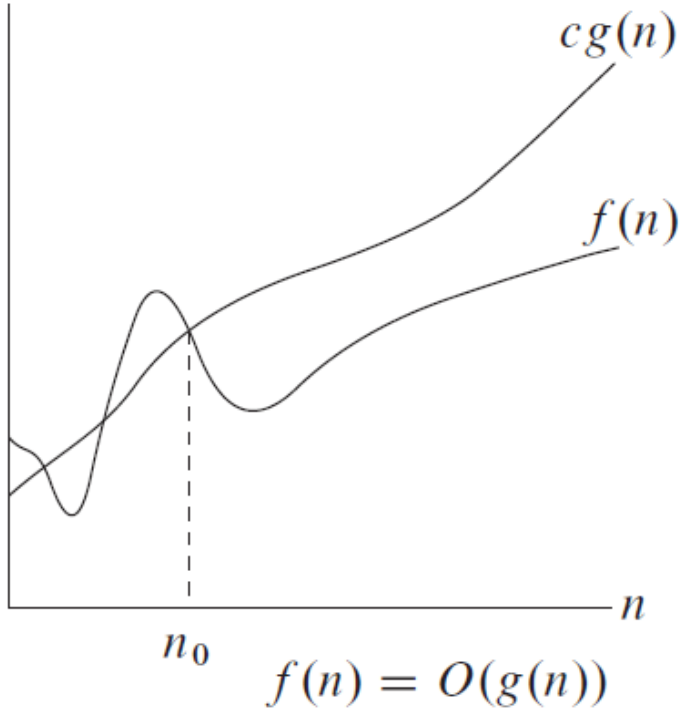
$f(n) \in O(g(n)) \Leftrightarrow \exists$ two positive constants $c \in \mathbb{R}, c > 0,$ and $n_0 \in \mathbb{N}$ s. t.:

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

The O-notation provides an *asymptotic upper bound* for a function: for all values of $n \geq n_0$ the value of the function $f(n)$ is less than or equal to $c \cdot g(n)$.

O-notation

- Graphical representation:



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

O-notation

Alternative definition

$f(n) \in O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is a finite constant.

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

O-notation

Alternative definition

$f(n) \in O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is a finite constant.

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

- $T(n) \in O(n^2)$ because $T(n) \leq c * n^2$ for c = 5 and n $\geq$ 1
- $T(n) \in O(n^3)$ because

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

Ω-notation

Ω-notation

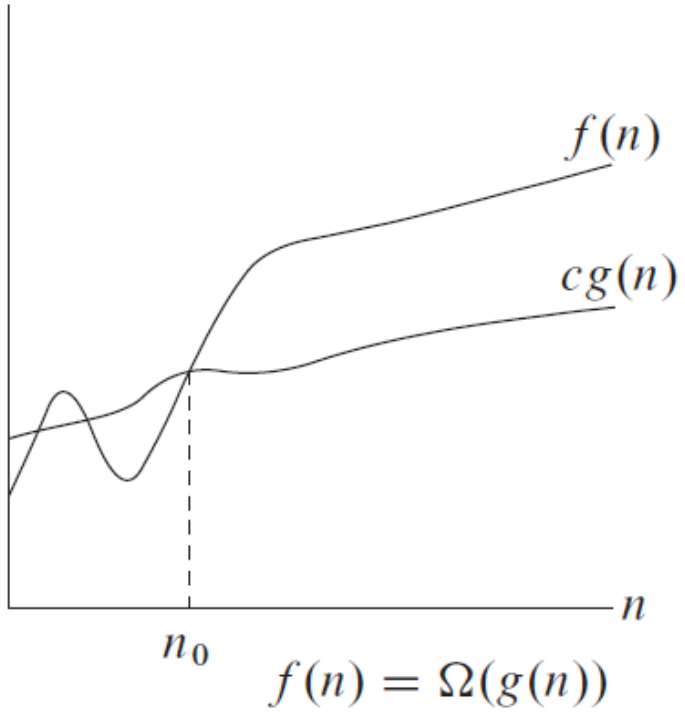
$f(n) \in \Omega(g(n)) \Leftrightarrow \exists$ two positive constants $c \in \mathbb{R}, c > 0,$ and $n_0 \in \mathbb{N}$ s. t.:

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

The Ω-notation provides an *asymptotic lower bound* for a function: for all values of $n \geq n_0$ the value of the function $f(n)$ is greater than or equal to $c \cdot g(n)$.

Ω-notation

- Graphical representation:



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

Ω-notation

Alternative definition

$f(n) \in \Omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is ∞ or a nonzero constant.

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

Ω-notation

Alternative definition

$f(n) \in \Omega(g(n))$ 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 $\geq$ 1
- $T(n) = \Omega(n)$ because

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

Θ-notation

Θ-notation

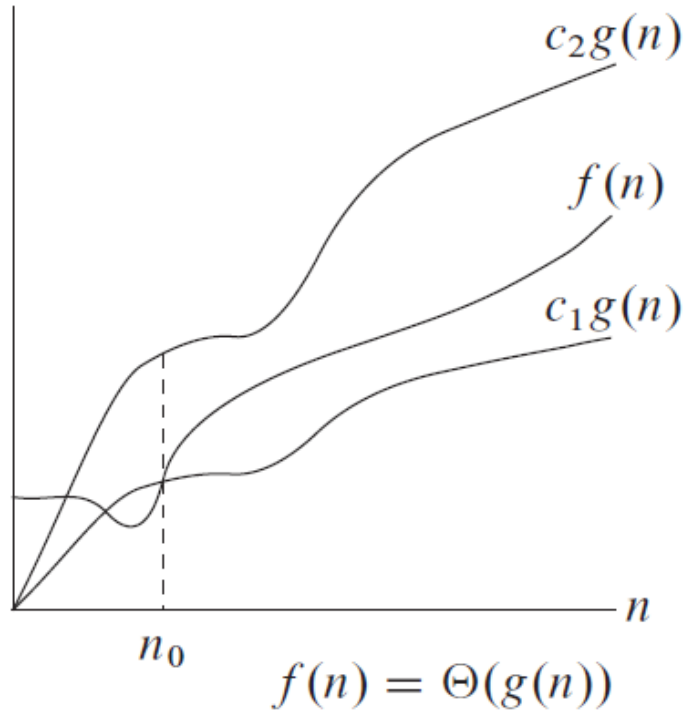
$f(n) \in \Theta(g(n)) \Leftrightarrow \exists$ three positive constants $c_1, c_2 \in \mathbb{R}, c > 0,$ and $n_0 \in \mathbb{N}$ s. t.:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for any } n \geq n_0$$

The Θ-notation provides an *asymptotically tight bound* for a function: for all values of $n \geq n_0$ the value of the function $f(n)$ is between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$.

Θ-notation

- Graphical representation:



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

Θ-notation

Alternative definition

$f(n) \in \Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is a nonzero finite constant.

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

Θ-notation

Alternative definition

$f(n) \in \Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is a nonzero finite constant.

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 $c_1 = 0.5$, $c_2 = 5$ and n $\geq$ 1.
- $T(n) = \Theta(n^2)$ 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?

 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?

Best Case, Worst Case, Average Case

For the second problem the number of steps taken by the algorithm does not depend on the length of the array exclusively, but also on the exact values from the array.

For an array of fixed length n, the execution can stop after:

- verifying the first number - if it is the one we are looking for
- verifying the first two numbers - if the first is not the one we are looking for, but the second is
- ...
- verifying all n numbers - first $n - 1$ are not the one we are looking for, but the last one is, or none of the numbers is the one we are looking for

Best Case, Worst Case, Average Case

For such algorithms we will consider three cases:

- **Best - Case** - the best possible case, where the number of steps is minimum
- **Worst - Case** - the worst possible case, where the number of steps is maximum
- **Average - Case** - the average of all possible cases

Best Case, Worst Case, Average Case

 Best and worst case complexities are usually computed by inspecting the code. For our example, we have:

- Best case: $\Theta(1)$ - just the first number is checked, regardless of the length of the array.
- 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
  - I is one input data
  - $P(I)$ is the probability of having I as an input
  - $E(I)$ is the number of operations performed for input I

Best Case, Worst Case, Average Case

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 the one we are looking for
  - One I represents all the arrays where the first number is not the one we are looking for, but the second is
  - ...
  - One I represents all the arrays where the first $n-1$ elements are not the one we are looking for but the last one is
  - One I represents all the arrays which do not contain the element we are looking for
- If the distribution of the possible input data is unknown, P(I) is considered equal for every I. In our case: $P(I) = \frac{1}{n+1}$

Best Case, Worst Case, Average Case

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 the one we are looking for
  - One I represents all the arrays where the first number is not the one we are looking for, but the second is
  - ...
  - One I represents all the arrays where the first $n-1$ elements are not the one we are looking for but the last one is
  - One I represents all the arrays which do not contain the element we are looking for
- If the distribution of the possible input data is unknown, P(I) is considered equal for every I. In our case: $P(I) = \frac{1}{n+1}$

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

Best Case, Worst Case, Average Case

 When we have different best-case and worst-case complexities, we will report the total complexity using the O-notation and the complexity function for the worst-case.

 For our example:

- Best case: $\Theta(1)$
- Worst case: $\Theta(n)$
- Average case: $\Theta(n)$
- Total (overall) complexity: $O(n)$

 **Obs**: For the best, worst and average cases we will always use the Θ notation.

Algorithm Analysis for Recursive Algorithms

 How can we compute the time complexity of a recursive algorithm?

Recursive Binary Search

Recursive function for binary search

```
function BinarySearchR (array, elem, start, end) is:
//array - an ordered array of integer numbers
//elem - the element we are searching for
//start - the beginning of the interval in which we search (inclusive)
//end - the end of the interval in which we search (inclusive)
    if start > end then
        BinarySearchR ← False
    end-if
    middle ← (start + end) / 2
    if array[middle] = elem then
        BinarySeachR ← True
    else if elem < array[middle] then
        BinarySearchR ← BinarySearchR(array, elem, start, middle-1)
    else
        BinarySearchR ← BinarySearchR(array, elem, middle+1, end)
    end-if
end-function
```

Recursive Binary Search

 Initial call to the *BinarySearchR* algorithm for an ordered array of *nr* elements:

Initial call:

```
BinarySearchR(array, elem, 1, n)
```

 How do we compute the complexity of the BinarySearchR algorithm?

Recursive Binary Search

- We will denote the length of the sequence that we are checking at every iteration by n (so $n = end - start$)

- We need to write the recursive formula of the complexity

Recursive Binary Search

- We will denote the length of the sequence that we are checking at every iteration by n (so $n = end - start$)

- We need to write the recursive formula of the complexity

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

Time complexity for BinarySearchR

- We suppose that $n = 2^k$ and rewrite the second branch of the recursive formula:

$$T(2^k) = T(2^{k-1}) + 1$$

- Now, we write what the value of $T(2^{k-1})$ is (based on the recursive formula)

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

- Next, we add what the value of $T(2^{k-2})$ is (based on the recursive formula)

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

Time complexity for BinarySearchR

- The last value that can be written is the value of $T(2^1)$

$T(2^1) = T(2^0) + 1$

Time complexity for BinarySearchR

Now, we write all these equations together and add them (and we will see that many terms can be simplified, because they appear on the left hand side of an equation and the right hand side of another equation):

$$T(2^k) = T(2^{k-1}) + 1$$
$$T(2^{k-1}) = T(2^{k-2}) + 1$$
$$T(2^{k-2}) = T(2^{k-3}) + 1$$
$$...$$
$$T(2^1) = T(2^0) + 1$$
$$\text{---------- } +$$
$$T(2^k) = T(2^0) + 1 + 1 + 1 + ... + 1 = 1 + k$$

Obs: For non-recursive functions adding a +1 or not does not influence the result, but in the case of recursive functions it does.

Time complexity for BinarySearchR

We started from the notation $n = 2^k$. So, we go back to the notation that uses n. If $n = 2^k \Rightarrow k = log_2 n$

$T(2^k) = 1 + k$
$T(n) = 1 + log_2 n \in \Theta(log_2 n)$

Actually, if we look at the code of *BinarySearchR,* we can observe that it has a best case (when the element is found in the first step, if it is exactly in the middle of the array), so the final complexity is $O(log_2 n)$.

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you