DATA STRUCTURES AND ALGORITHMS
LECTURE 1

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

Overview

Guiding Teachers

- Lecturer Dr. Marian Zsuzsanna
- Lecturer Dr. Lupșa Dana
- Assistant Dr. Mircea Gabriel

Activities

- **Lecture:** 2 hours / week
- **Seminar:** 1 hour / week
- **Course page:** www.cs.ubbcluj.ro/~marianzsu/DSA.html
- **Email:** marianzsu@cs.ubbcluj.ro
  - Please use your *scs.ubbcluj.ro* email address for communication.

Grading

- Written exam (**W**)
- Project (**P**)
- Seminar grade (**S**)
  - Partial paper (**P**)
  - Seminar activity (**A**)
  - Project stage (**ST**)
  - Seminar grade $S = 0.5 * P + 0.4 * A + 0.1 * ST$
- The final grade is computed as:

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

- **To pass the exam W and P has to be $\geq$ 5 (no rounding)**

Rules I

- Attendance is compulsory for the seminar activity. You need at least 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!**

Rules II

- You have to come to the seminar with your 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 seminar teacher about the switch during the first two weeks of the semester.
- Seminar attendance can be recovered with another group, within the two weeks allocated for the seminar, with the explicit agreement of the seminar teacher.

Rules III

- There is a project that you will have to realize by the end of the semester. Project topics will be allocated in the 5^{th} seminar, where more information about the requirements of the project and its grading will be given.

- The grade for the project has to be ≥ 5 in order to be able to pass the exam (no matter what the value of G is).

- Projects with a grade < 5 have to be redone in the retake session. For these projects the maximum possible grade is 5.

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.

- The partial exam and the seminar activity cannot be redone in the retake session. The project cannot be redone in the retake session (unless the grade for the project is < 5, but in this case it *has to be* redone).

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

- M. D. MOUNT, *Data Structures*, University of Maryland, 1993. PDF version available at: *http://www.cs.ubbcluj.ro/~gabis/sda/Docs/David%20Mount/*
- S. SALTENIS, *Algorithms and data structures*, 2002, Lecture notes available at: *http://www.cs.ubbcluj.ro/~gabis/sda/Docs/Simonas%20Saltenis/*

Abstract Data Types

- 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).

Why do we need so much abstraction?

- There are several different 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 (the STL library from C++, Collections in Java, containers and collections in Python, System.Collections in .Net, etc.).

- 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 don't 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.

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.

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 only 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
- .Net - System.Collections framework
- In the following, in this course we will use the term **container**.

Data Structures I

- The domain of data structures studies how we can store and access data.
- A data structure can be:
  - Static: the size of the data structure is fixed. Such data structures are suitable if a known 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.

Data Structures II

- 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.

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 type 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 us 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.

Pseudocode V

- 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 ← v *//syntax used to return the value v*
end-function

- The function can be called as:

result ← 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: $a : T[Nr]$ - indexing is done from 1 to Nr
  - If we want to specify both indexes for the array we will use: $a : T[Min...Max]$ - indexing is done from Min to Max

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 $\uparrow$:
  - p: $\uparrow$ 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

Specifications I

- 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 can throw 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 *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:**

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

Generic Data Types II

- When the values of a data type can be compared and ordered based on a relation, we will use the generic type: *TComp*.
- 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:**

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

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

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, 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 model is an overly 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

```
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
```

- The number of steps taken by the above subalgorithm is: $T(n) = 1 + n * (1 + n + 1) + 1 = n^2 + 2n + 2$

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*
- 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:

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

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

O-notation II

- Graphical representation:

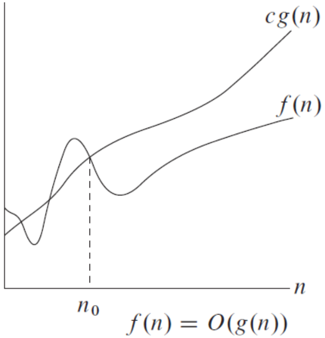


Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

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 $T(n) \leq c * n^2$ 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:

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s. t.} \\ 0 \le c \cdot g(n) \le f(n) \text{ for all } n \ge n_0\}$$

- 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:

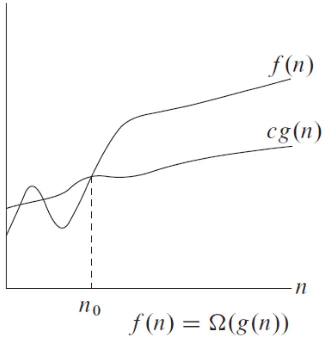


Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

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 $T(n) \geq c * n^2$ for c $=$ 0.5 and n $\geq$ 1
  - $T(n) = \Omega(n)$ because $\lim_{n\to\infty} \frac{T(n)}{n} = \infty$

Θ-notation I

Θ-notation

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

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

- 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:

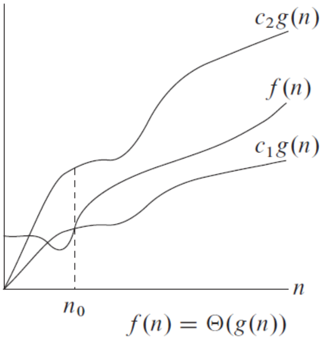


Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

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 $c_1 * n^2 \leq T(n) \leq c_2 * n^2$ 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 I

? Think about an algorithm that finds the first even number in an array. How many steps does the algorithm take for an array of length n?

Best Case, Worst Case, Average Case II

- For this 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, execution of the algorithm can stop:
  - after verifying the first number - if it is even
  - after verifying the first two numbers - if the first is odd and the second is even
  - after verifying the first 3 numbers - if the first two are odd and the third is even
  - ...
  - after verifying all n numbers - first $n-1$ are odd and the last is even, or all numbers are odd

Best Case, Worst Case, Average Case III

- 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

Best Case, Worst Case, Average Case IV

- 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 V

- 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 even
  - One I represents all the arrays where the first number is odd, the second is even
  - ...
  - One I represents all the arrays where the first $n-1$ elements are odd and the last is even
  - One I represents all the arrays with no even number
- 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 VI

- 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: $O(n)$

Example

- In order to see empirically how much the number of steps taken by an algorithm can influence its running time, we will consider 4 different implementations for the same problem:
- *Given an array of positive and negative values, find the maximum sum that can be computed for a subsequence. If a sequence contains only negative elements its maximum subsequence sum is considered to be 0.*
- For the sequence [-2, 11, -4, 13, -5, -2] the answer is 20 (11 - 4 + 13)
- For the sequence [4, -3, 5, -2, -1, 2, 6, -2] the answer is 11 (4 - 3 + 5 - 2 - 1 + 2 + 6)

First algorithm

First algorithm

- The first algorithm will simply compute the sum of elements between any pair of valid positions in the array.

```
function first (x, n) is:
//x is an array of integer numbers, n is the length of x
   maxSum ← 0
   for i ← 1, n execute
      for j ← i, n execute
      //compute the sum of elements between i and j
         currentSum ← 0
         for k ← i, j execute
            currentSum ← currentSum + x[k]
         end-for
         if currentSum > maxSum then
            maxSum ← currentSum
         end-if
      end-for
   end-for
   first ← maxSum
end-function
```

Complexity of the algorithm:

$$T(x,n)=\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1=\ldots=\Theta(n^3)$$

Second algorithm

Second algorithm

- We can eliminate the third (innermost) loop by observing the following:
  - If we have the sum of numbers between indexes i and j we can compute the sum of numbers between indexes i and $j+1$ by simply adding the element $x[j+1]$. We don't need to recompute the whole sum.

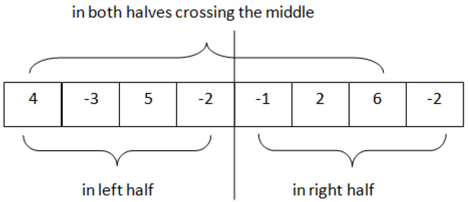
```
function second (x, n) is:
//x is an array of integer numbers, n is the length of x
   maxSum ← 0
   for i ← 1, n execute
      currentSum ← 0
      for j ← i, n execute
         currentSum ← currentSum + x[j]
         if currentSum > maxSum then
            maxSum ← currentSum
         end-if
      end-for
   end-for
   second ← maxSum
end-function
```

Complexity of the algorithm:

$$T(x, n) = \sum_{i=1}^{n} \sum_{j=i}^{n} 1 = ... = \Theta(n^2)$$

Third algorithm I

Third algorithm II



- The maximum subsequence sum for the two halves can be computed recursively.
- How do we compute the maximum subsequence sum that crosses the middle?

Third algorithm III

- We will compute the maximum sum on the left (for a subsequence that ends with the middle element)
  - For the example above the possible subsequence sums are:
    - -2 (indexes 4 to 4)
    - 3 (indexes 3 to 4)
    - 0 (indexes 2 to 4)
    - 4 (indexes 1 to 4)
  - We will take the maximum (which is 4)

Third algorithm IV

- We will compute the maximum sum on the right (for a subsequence that starts immediately after the middle element)
    - For the example above the possible subsequence sums are:
        - -1 (indexes 5 to 5)
        - 1 (indexes 5 to 6)
        - 7 (indexes 5 to 7)
        - 5 (indexes 5 to 8)
    - We will take the maximum (which is 7)
- We will add the two maximums (11)

Third algorithm V

- When we have the three values (maximum subsequence sum for the left half, maximum subsequence sum for the right half, maximum subsequence sum crossing the middle) we simply pick the maximum.

Third algorithm VI

- We divide the implementation of the third algorithm in three separate algorithms:
  - One that computes the maximum subsequence sum crossing the middle - *crossMiddle*
  - One that computes the maximum subsequence sum between position [left, right] - *fromInterval*
  - The main one, that calls *fromInterval* for the whole sequence - *third*

```
function crossMiddle(x, left, right) is:
//x is an array of integer numbers
//left and right are the boundaries of the subsequence
  middle ← (left + right) / 2
  leftSum ← 0
  maxLeftSum ← 0
  for i ← middle, left, -1 execute
    leftSum ← leftSum + x[i]
    if leftSum > maxLeftSum then
      maxLeftSum ← leftSum
    end-if
  end-for
//continued on the next slide...
```

```
    //we do similarly for the right side
    rightSum ← 0
    maxRightSum ← 0
    for i ← middle+1, right execute
        rightSum ← rightSum + x[i]
        if rightSum > maxRightSum then
            maxRightSum ← rightSum
        end-if
    end-for
    crossMiddle ← maxLeftSum + maxRightSum
end-function
```

```
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
```

function third (x, n) **is:**
//x is an array of integer numbers, n is the length of x
  third ← fromInterval(x, 1, n)
end-function

Complexity of the solution (fromInterval is the main function):

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

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

Let $n = 2^k$

Ignoring the parameter x we rewrite the recursive branch:

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

$$\overline{\hspace{10em}} +$$

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

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

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

Let's go back to the notation with n.

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

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

Fourth algorithm

Fourth algorithm

- Actually, it is enough to go through the sequence only once, if we observe the following:
  - The subsequence with the maximum sum will never begin with a negative number (if the first element is negative, by dropping it, the sum will be bigger)
  - The subsequence with the maximum sum will never start with a subsequence with total negative sum (if the first k elements have a negative sum, by dropping all of them, the sum will be bigger)
  - We can just start adding the numbers, but when the sum gets negative, drop it, and start over from 0.

```
function fourth (x, n) is:
//x is an array of integer numbers, n is the length of x
   maxSum ← 0
   currentSum ← 0
   for i ← 1, n execute
      currentSum ← currentSum + x[i]
      if currentSum > maxSum then
         maxSum ← currentSum
      end-if
      if currentSum < 0 then
         currentSum ← 0
      end-if
   end-for
   fourth ← maxSum
end-function
```

Complexity of the algorithm:

$$T(x, n) = \sum_{i=1}^{n} 1 = ... = \Theta(n)$$

Comparison of actual running times

Input size	First $\Theta(n^3)$	Second $\Theta(n^2)$	Third $\Theta(nlogn)$	Fourth $\Theta(n)$
10	0.00005	0.00001	0.00002	0.00000
100	0.01700	0.00054	0.00023	0.00002
1,000	16.09249	0.05921	0.00259	0.00013
10,000	-	6.23230	0.03582	0.00137
100,000	-	743.66702	0.37982	0.01511
1,000,000	-	-	4.51991	0.16043
10,000,000	-	-	48.91452	1.66028

Table: Comparison of running times measured with Python's default_timer()

Comparison of actual running times

- From the previous table we can see that complexity and running time are indeed related:
- When the input is 10 times bigger:
  - The first algorithm needs $\approx$ 1000 times more time
  - The second algorithm needs $\approx$ 100 times more time
  - The third algorithm needs $\approx$ 11-13 times more time
  - The fourth algorithm needs $\approx$ 10 times more time

Think about it

? How would the implementation of the discussed 4 algorithms change if we could not have 0 as maximum sum when all the elements are negative?