Algorithms & Data Structures

Lecture 01
Algorithms, Correctness, and Efficiency

Giovanni Bacci
giovbacci@cs.aau.dk

Outline

- Algorithms & pseudocode
- Efficiency & order of growth
- Correctness of an algorithm
- Case Study: find an element in a sequence of numbers

Intended Learning Goals

KNOWLEDGE

- Mathematical reasoning on concepts such as recursion, induction, concrete and abstract computational complexity
- Data structures, algorithm principles e.g., search trees, hash tables, dynamic programming, divide-and-conquer
- Graphs and graph algorithms e.g., graph exploration, shortest path, strongly connected components.

SKILLS

- Determine abstract complexity for specific algorithms
- Perform complexity and correctness analysis for simple algorithms
- Select and apply appropriate algorithms for standard tasks

COMPETENCES

- Ability to face a non-standard programming assignment
- Develop algorithms and data structures for solving specific tasks
- Analyse developed algorithms

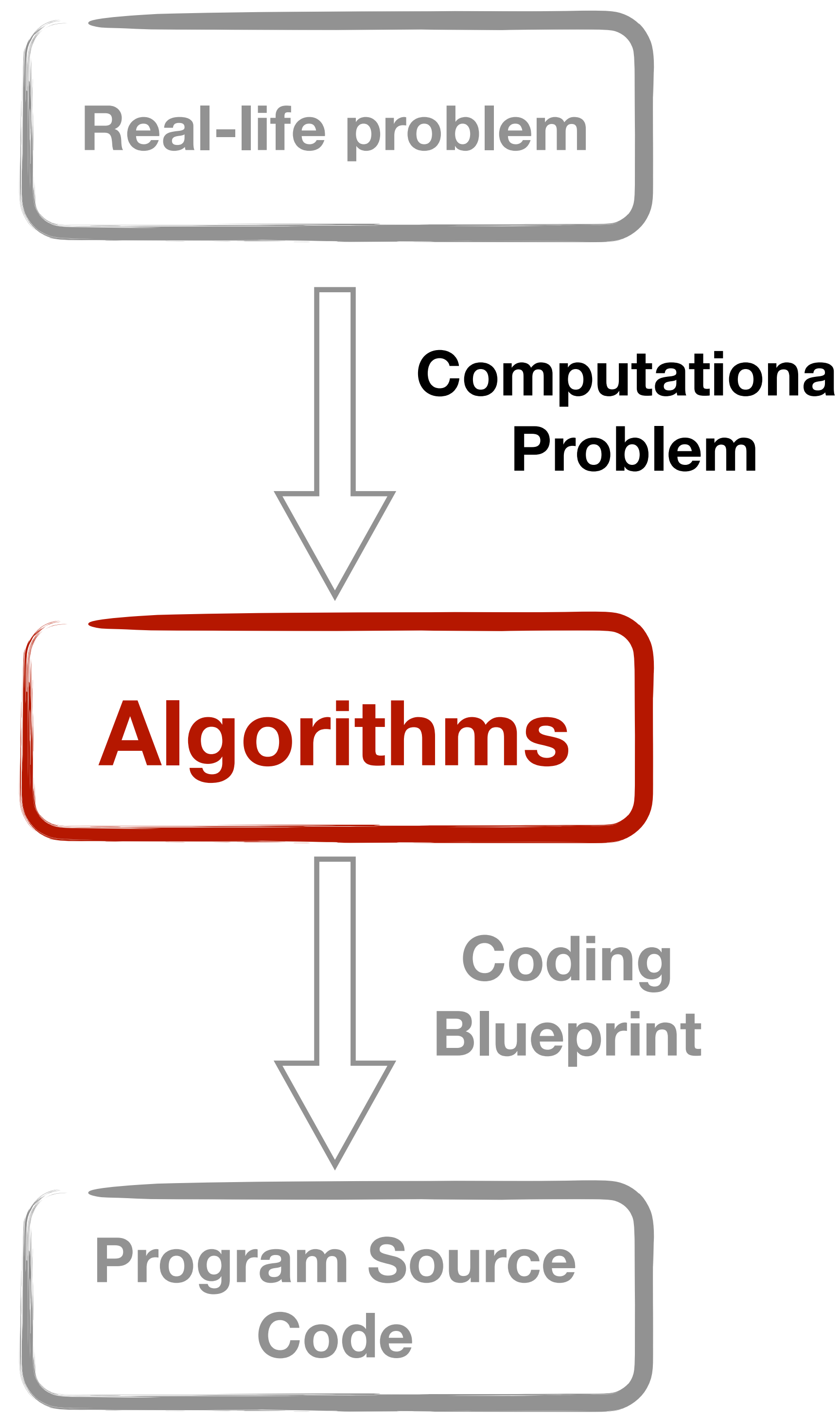
Problem Solving Procedure

Covered in this course

- Analysis
  - Specification
  - Correctness
  - Efficiency
- Design
  - Computational Thinking
  - Algorithm patterns
  - Data Structures

NOT about

- Specific programming languages
- Computer architectures
- Software architectures
- Software design and development principles
- Software testing and verification



What are algorithms?

- Informally, an algorithm is any well-defined computational procedure that takes some values, as **input** and produces some values, as **output**

- An algorithm is a **sequence of computational steps** that transforms the input into the output.

- Is a tool for solving a well-specified **computational problem**. The statement of the problem specifies the desired I/O relationship.

Computational Problems

The sorting problem:
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ of the input such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

The element search problem:
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ and a number a
Output: An index $1 \leq i \leq n$ such that $a = a_i$ if there exists such an index, 0 otherwise.

Characteristics of a computational problem:
- The the format of the input and the output are specified
- The desired input/output relationship is not ambiguous

Computational Problems

The sorting problem:
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: A permutation (reordering) $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ of the input such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

Example:

- An input sequence $\langle 32,5,8,9,20 \rangle$ is a valid **instance** of the sorting problem.
- Its associated output sequence is $\langle 5,8,9,20,32 \rangle$.

Definition: An *instance* of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Quiz

The element search problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ and a number a

Output: An index $1 \leq i \leq n$ such that $a = a_i$ if there exists such an index, 0 otherwise.

- Which of the following are valid instances of the element search problem?
  - A. $\langle 1,2,4,20,0,-5 \rangle, 2$
  - B. $\langle 1,2,4,20,f,-5 \rangle, f$
  - C. $\langle 1,7,30,5 \rangle$
  - D. $\langle \rangle, 4$

Quiz

The element search problem:
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ and a number a
Output: An index $1 \leq i \leq n$ such that $a = a_i$ if there exists such an index, 0 otherwise.

- Which of the following are correct I/O pairs for the 'element search problem'?
  A. $\langle 1,2,4,20,0,-5 \rangle, 2 \mapsto 0$
  B. $\langle 1,2,4,20,2,-5 \rangle, 2 \mapsto 2$
  C. $\langle 1,2,4,20,2,-5 \rangle, 2 \mapsto 5$
  D. $\langle 1,7,30,5 \rangle, 20 \mapsto 0$
  E. $\langle \rangle, 4 \mapsto 0$

- How many I/O pairs do we need to test to establish that an algorithms solves the 'element search problem'?

From Problems to Algorithms

- A problem can be solved in several different ways, therefore there are **many correct algorithms solving the same problem**
- Which algorithm is best for a given application depends on — among other factors —
  - the **size of the input**,
  - the actual configuration of the input,
  - the **data structures used**,
  - the computer architecture, and
  - the **algorithmic design**

Definition: An algorithm is said to be **correct** if, for every input instance, it halts with the correct output. We say that a correct algorithm **solves** the given computational problem.

An incorrect algorithm *might not halt on some input instances*, or it might halt with an incorrect answer.

Describing Algorithms

- We will typically describe algorithms as programs written in a **pseudocode** (similar in many respects to C, C++, Java, Python, or Pascal)

- In pseudocode we employ whatever expressive method is most clear and concise (sometimes just English)

- We are **NOT concerned with issues of software engineering** such as
  - Data abstraction
  - Modularity
  - Error handling

Finding an element

Here is presented the **pseudocode** for solving the element search problem as a **function** called FIND-ELEMENT, which takes as a parameter an array $A[1..n]$ representing a sequence of numbers, and a number a representing the number to find.

Sketched Idea: scan the array $A[1..n]$ element by element and compare the current element $A[i]$ with a. If the two are equal, store the current index.

```
FIND-ELEMENT(A, a)
  j = 0
  for i = 1 to A.length
      if A[i] = a
          j = i
  return j
```

Finding an element

There are **many alternative approaches for solving the same problem**. The following pseudocode describes one possible alternative algorithm for solving the element search problem.

Sketched Idea: scan the array $A[1..n]$ from the last element and compare the current element $A[i]$ with a.

If the two are equal, terminate returning the index.

```
FIND-ELEMENTV2(A, a)
  i = A.length
  while i > 0 and A[i] ≠ a
      i = i − 1
  return i
```

Quiz

- Consider the pseudocode aside. What will be the value of the variables i and j after execution on the following instances?
  - A. $\langle 1,2,4,20,0,-5 \rangle, 2$
  - B. $\langle 1,2,4,2,2,3 \rangle, 2$
  - C. $\langle 1,7,30,5 \rangle, 3$
  - D. $\langle \rangle, 4$

FIND-ELEMENT(A, a)

$j = 0$
for $i = 1$ **to** $A.length$
  if $A[i] = a$
    $j = i$
return j

- Does FIND-ELEMENT solve the 'element search' problem? How do you justify your answer?
- How does the value of i at the end of the execution relate with the length of the given array A?

Efficiency

- Computer may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not for free.
- When designing an algorithm one should use these resources wisely, and algorithms that are **efficient in terms of time and space** will help you do so.
- Different algorithms devised to solve the same problem often differ dramatically in their **efficiency**

Efficiency Analysis

- Analysing the efficiency of algorithms deals with **predicting the resources that the algorithm requires**.

- Most often is **computational time** that we want to measure

- The time taken by an algorithm depends on the **actual input** (e.g., different array size, or different array configurations)

- It is traditional to describe the **running time of an algorithm as a function of the size of its input**.

Input Size

The best notion of input size depends on the problem being solved

- A natural notion is the **number of items in the input** (e.g., array size n for sorting or element search)
- For many other problems it is more appropriate to use the **total number of bits** needed to represent the input (e.g., for multiplying two integers)
- Sometimes, it more appropriate to **use two or more numbers** rather than one (e.g., the size of a graph is described by the number of its vertices and edges)

Running Time

- The running time of an algorithm on a particular input is **the number of primitive operations or ‘steps’ executed**
- We will use a notion of ‘step’ so that it is as **machine-independent** as possible
- The book uses as computational model a generic one-processor, **random-access machine (RAM)** (page 23-24)
- For now, we assume a **constant amount of time for each line of pseudocode**
  - Different lines may take different time, yet constant

Example

The running time of Find-Element on the instance $(A[1..n], a)$ is the sum of the running times for each statement executed.

Find-Element(A, a)	cost	times
$j = 0$	c_1	1
for $i = 1$ **to** $A.length$	c_2	$n + 1$
if $A[i] = a$	c_3	n
$j = i$	c_4	n_a
return j	c_5	1

One more time than the loop body!

Where n is the number of elements of A, and n_a is the number of occurrences of a in A.

$$T(n) = c_1 + c_2(n+1) + c_3 n + n_a c_4 + c_5$$

Example

Consider the **exact running time** of FIND-ELEMENT

$$T(n) = c_1 + c_2(n+1) + c_3 n + n_a c_4 + c_5$$

Even for input of a the same size, the running time may depend on which specific array of that size is given.

- The **best case** occurs if the number of occurrences of a is minimal, that is, when $n_a = 0$

$$T(n) = (c_2 + c_3)n + c_1 + c_2 + c_5$$

- The **worst case** occurs if the number of occurrences of a is maximal, that is $n_a = n$

$$T(n) = (c_2 + c_3 + c_4)n + c_1 + c_2 + c_5$$

Example

Consider the **exact running time** of Find-Element

$$T(n) = c_1 + c_2(n+1) + c_3 n + n_a c_4 + c_5$$

Even for input of a the same size, the running depend on which specific array of that size is given.

linear functions of n.
Of the form $an + b$ for some constants a and b

- The **best case** occurs if the number of occ is, when $n_a = 0$

$$T(n) = (c_2 + c_3)n + c_1 + c_2 + c_5$$

- The **worst case** occurs if the number of occurrences of a is maximal, that is $n_a = n$

$$T(n) = (c_2 + c_3 + c_4)n + c_1 + c_2 + c_5$$

Worst-case analysis

- We will mainly, concentrate on analysing the worst-case running time
- It is the longest running time for any input of size n
- Knowing it **provides a guarantee that the algorithm will never take longer than that to return the output**.
- For some algorithms the worst case occurs fairly often

Order of growth

- We already used some abstraction to describe $T(n)$
  1. We ignored actual cost of statements by using generic constants c_i
  2. Then, we expressed the worst-case running time as $an + b$ for some $a, b > 0$ (that depend on c_i's)
- We can abstract even further by only looking at the **rate of growth** (or order of growth) of $T(n)$
  - We consider **only the leading term** (i.e., an) and we also **ignore its constant coefficient** for large values of n
  - FIND-ELEMENT has worst case running time of $\Theta(n)$

Quiz

Comparison of running times. For each function $f(n)$ and time t in the following table, determine the largest size n of an instance that can be solved in time t, assuming the algorithms for solving the problem takes $f(n)$ microsecond (1 microsecond = 10^{-6} sec)

	1 **sec**	1 **min**	1 **hr**
$\log n$			
n			
n^2			
2^n			

Learned Today

- Formal concepts of
  - algorithm
  - computational problem
  - problem instance
  - correctness of an algorithm
- Using pseudocode to describe algorithms
- Algorithm efficiency
  - Input size
  - ‘Exact’ running time
  - Worst case running time
  - Rate of growth