Exam IN2010 fall 2023

15. December 2023

About the exam

- The exam consists of a tiny warm-up, followed by two main parts.
- The first part consists of small exercises, which are automatically corrected. No distinction is made between leaving a question unanswered and providing an incorrect answer; this means there is no reason to leave a question unanswered.
- The second part consists of more comprehensive exercises, where you to a greater extent have to provide pseudocode and written explanations.
- No aids are allowed.
- An English translation of the full set of exercises is attached as a PDF.

Comments and tips

- *Read the exercise very carefully*.
- Make sure you answer precisely what the exercise asks of you.
- Explain any assumptions you make.
- Make sure that your delivery is clear, precise, and easy to understand, both in terms of structure and content.
- If you get stuck on a problem, then you might want to proceed to another problem and return later.
- All exercises that require an implementation should be answered in *pseudo-code*. The important thing is that the pseudo-code is *easy to comprehend, unambiguous, and precise*.
- An *easy to comprehend, unambiguous, and precise* natural language explanation, may yield more points than pseudocode that is difficult to comprehend, ambiguous or imprecise.
- A *easily understandable, unambiguous and precise* explanation in natural language, may be worth more points than pseudocode that is difficult to comprehend, ambiguous or imprecise.
- In implementation tasks, lower runtime complexity is preferred.
- You can assume that you have algorithms and data structures known from the curriculum available, unless otherwise specified.

Warm-up 2 poeng

(a) What is an algorithm? Keep your answer brief (maximum four sentences).

(b) What is a data structure? Keep your answer brief (maximum four sentences).

Here, we are not after a “textbook answer”. We are interested in seeing your understanding of the terms. Any reasonable answer gives a full score.

Insertion in heap — 10 poeng

(a) You are given an empty array of size 7.

0	1	2	3	4	5	6

Insert the following numbers using min-heap insertion:

$$3, 5, 10, 8, 4, 2, 7$$

Fill out the table as it looks after all the numbers have been inserted. Write the answer as a comma-separated list.

(b) Briefly explain how the algorithm for insertion in a min-heap works, and sketch the algorithm with pseudocode. You can assume that the array in the input is not full.

```
Input: An array A that represents a heap with n elements, and an element x
Output: An array that represents a heap, which contains x
1 Procedure HeapInsert(A, x)
  | // ...
```

(c) Account for the runtime complexity of the algorithm.

Find duplicate 12 poeng

You are given an array `A` with comparable elements. You are told that `A` contains exactly one duplicate. Thus, all elements in `A` are unique, except for *one* element, which occurs exactly twice.

(a) Assume that you are given a *sorted* array `A`, and you are told that x is the duplicate in `A`. Write an efficient procedure that prints the two positions that contains x. State the runtime complexity of the algorithm.

Input: A *sorted* array `A` with n comparable elements, and an element x which occurs exactly twice
Output: Prints the two positions $0 \leq i < j < n$ where `A[i]` $= x$ and `A[j]` $= x$ occur
Procedure `FindSortedIndicesOfDuplicate(A,` x`)`

```
// ···
```

(b) Assume that you are given a *sorted* array `A` (and now you are not given the element that is duplicated). Write an efficient procedure that prints the two positions of the element that occurs twice. State the runtime complexity of the algorithm.

Input: A *sorted* array `A` with n comparable elements
Output: Prints the two positions $0 \leq i < j < n$ where `A[i]` $=$ `A[j]`
Procedure `FindSortedDuplicateIndices(A)`

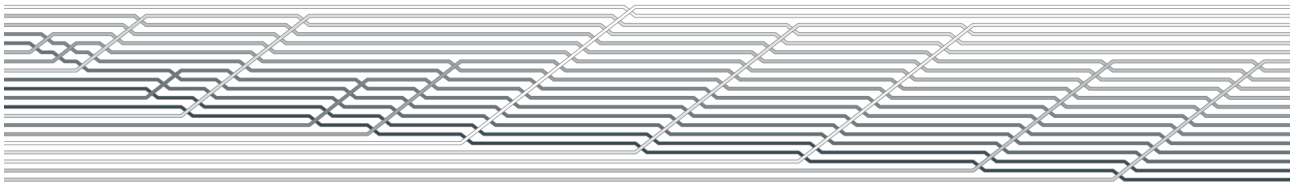
```
// ···
```

(c) Assume that you are given an array `A` (now you cannot assume that the array is sorted). Write an efficient procedure that prints the two positions of the element that is duplicated. State the runtime complexity of the algorithm.

Input: An array `A` with comparable n elements
Output: Prints the two positions $0 \leq i < j < n$ where `A[i]` $=$ `A[j]`
Procedure `FindDuplicateIndices(A)`

```
// ···
```

Gnome Sort 10 poeng



Gnome Sort is a simple sorting algorithm that is *not* known from the curriculum. The illustration above is generated from a run of Gnome Sort. The pseudocode for the algorithm is provided below.

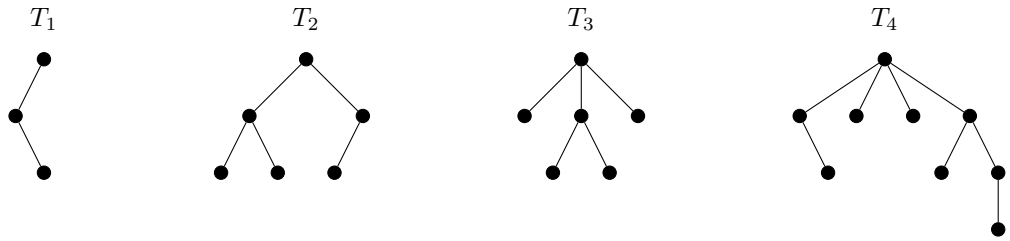
```
Input: An array A with n elements
Output: A sorted array with the same n elements
1 Procedure GnomeSort(A)
2     i ← 0
3     while i < n do
4         if i > 0 and A[i − 1] > A[i] then
5             A[i − 1], A[i] ← A[i], A[i − 1] // swap A[i] and A[i − 1]
6             i ← i − 1
7         else
8             i ← i + 1
```

In this task, you should discuss the advantages and disadvantages of Gnome Sort compared to other sorting algorithms known from the curriculum. Your text should include:

- a brief explanation of how Gnome Sort works (using natural language).,
- the runtime complexity of Gnome Sort (along with a brief justification),
- a comparison of Gnome Sort with sorting algorithms from the curriculum, and
- a brief account of whether Gnome Sort is stable and/or in-place.

Diameter of a Tree 12 poeng

We define the *diameter* of a tree as the length of the longest path between two nodes.



In the examples above, T_1 has a diameter of 2, T_2 a diameter of 4, and T_3 a diameter of 3.

(a) What is the diameter of T_4?

In the next subproblems, we limit ourselves to *binary* trees. If v is a node, then:

- v.`left` the left child of v
- v.`right` the right child of v

(b) Will the longest path in a binary tree always go through the root node? Justify your answer.

(c) Write a procedure that finds the diameter of a given *binary tree*.

```
Input: The root node v of a binary tree
Output: Returns the diameter of the tree
1 Procedure Diameter(v)
  |  // ···
```

A lower run-time complexity gives a higher score.

(d) State the runtime complexity of your algorithm for a binary tree with n nodes.

The Blindern Problem 10 poeng

The Blindern problem is a challenge known to many students at the University of Oslo. This challenge arises when students must move from one lecture to another on the opposite side of campus in just under fifteen minutes. You have been asked to consult on UiO's new project where a tunnel system is to be developed connecting all buildings belonging to the campus.

It has already been mapped out how much it will cost to dig tunnels between each pair of buildings (and assume that none of the outlined tunnels will collide). Your task is to find the most cost-effective way to construct a tunnel system, based on the outlined tunnels, such that one can move between all buildings solely by using the tunnel system.

(a) Briefly explain how this problem could be expressed as a graph problem. Your answer should include:
- What type of graph (directed/undirected and weighted/unweighted) is suitable here?
- What do the nodes represent?
- What do the edges represent?

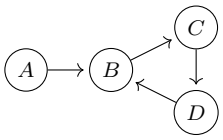
(b) Provide a suitable algorithm, known from the curriculum, which can be used to find the most cost-effective way to excavate a tunnel system so that one can move between all buildings solely by using the tunnel system. For the algorithm, state:
- the most central data structures it uses,
- a *short* explanation of how the algorithm works, and
- the runtime complexity of the algorithm.

(c) The project is completed in record time and is ready for use. Describe, in natural language, an efficient algorithm for finding the shortest path from one building to another using only the tunnel system. You can use algorithms known from the curriculum *without* having to explain them. State the runtime complexity of the algorithm.

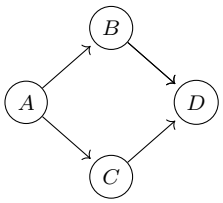
Dependency hell 12 poeng

In modern software development, it is common for an application to depend on many *libraries*, which in turn may depend on other libraries. A library can be seen as a collection of code that is designed for reuse. The libraries used to build an application are called the application's *dependencies*. The dependencies between libraries are given as a directed graph $G = (V, E)$ where a node represents a library, and where there is an edge from u to v if u uses v as a library.

The examples below are situations often referred to as *dependency hell*.



Circular dependency



Diamond dependency

Cyclic dependencies are generally unwanted because an update of one of the libraries in a cycle could affect all the other libraries in the cycle. *Diamond dependency* is also generally unwanted because an update can lead to being dependent on different versions of the same library simultaneously.

You must develop algorithms that can help identify such unwanted situations.

(a) More precisely, we say that the graph contains a *cyclic dependency* if there is a path from a node v to the same node v which consists of at least two nodes.

Write a procedure that, given a graph $G = (V, E)$, returns **`true`** if the graph contains *cyclic dependencies*, and **`false`** otherwise. If your procedure uses algorithms known from the curriculum, you must briefly explain how they work.

(b) In this sub-problem, you may assume that the graph does not contain cyclic dependencies.

More precisely, we say that the graph contains a *diamond dependency* if there are two distinct paths from u to v. Remember that two paths are distinct if they do not share any nodes or edges other than u and v.

Write a procedure that, given a directed acyclic graph $G = (V, E)$, returns **`true`** if the graph contains *diamond dependencies*, and **`false`** otherwise.