Exam IN2010 fall 2020

10. December 2020

About the exam

- The exam consists of a mixture of multiple choice, text, and coding questions. All your solutions must be provided in Inspera, and it is not possible to upload handwritten documents.
- In multiple choice problems, correct answers are given full score, incorrect answers gives 0 points (you cannot get a negative score).
- The answers must be written by you alone.
- All aids are allowed (textbook, online resources, notes, etc.).
- You are not allowed to collaborate or communicate with other people about the exam or to share their work with others.
- You can find contacts for questions regarding the execution of the exam at user support page for home exams,
- All information at the website on final examinations at the MN Faculty autumn 2020 applies.

Comments

- *Read the exercise very carefully*.
- Make sure you answer precisely what the exercise asks of you.
- 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. The lecture material are good examples of pseudo-code, but you are also free to use a Java- or Python-like syntax.

On grading

- The exam will be assessed with a letter grade. We place emphasis on clear answers and good explanations.
- After the exam, you might be be selected for an interview to check ownership of your solutions. This conversation will not affect your grade, but may lead to the department issuing a «suspicion-of-cheating case». This is described on UiO's website on control interviews and website on routines for handling suspicion of cheating and attempted cheating.

Warm-up 2 poeng

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

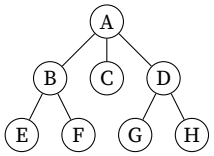
(b) What is a data structure? Keep your answer brief (maximum four sentences).

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.

Binary search trees — 4 poeng

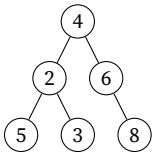
(a) — 1 poeng

The tree on the right is *not* a binary tree. Which node has to be removed in order for it to become a binary tree?



(b) — 1 poeng

The tree on the right is *not* a binary search tree. Which node has to be removed in order for it to become a binary search tree?



(c) — 1 poeng

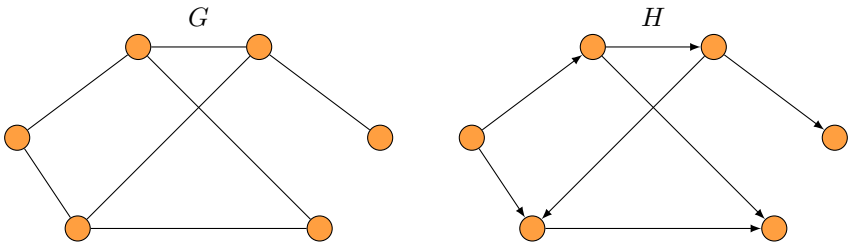
Is every binary search tree an AVL-tree?

(d) — 1 poeng

Is every AVL-tree a binary search tree?

Properties of graphs 8 poeng

We are given an undirected graph G, and a directed graph H that looks like this:



- Do we need to add or remove edges for G to be connected?
- Do we need to add or remove edges for G *not* to be connected?
- Do we need to add or remove edges for G to be a tree?
- Do we need to add or remove edges for G *not* to be a tree?
- Do we need to add or remove edges for H to be a DAG?
- Do we need to add or remove edges for H *not* to be a DAG?
- Do we need to add or remove edges for H to be strongly connected?
- Do we need to add or remove edges for H *not* to be strongly connected?

Linear probing 2 poeng

We begin with an initially empty array of size 10

0	1	2	3	4	5	6	7	8	9

The hash function you will apply is $h(k, N) = k \mod N$, which for this example is the same as $h(k) = k \mod 10$. In other words: a number is hashed to its last digit.

Use *linear probing* to insert the following numbers in the given order:

$$93, 48, 74, 99, 29, 13, 45$$

Fill in the table to reflect its state after all the numbers has been inserted with linear probing.

Search 12 poeng

Assume we are given three procedures:

- `Search` – which performs a straight forward search
- `BinarySearch` – which performs a binary search
- `HeapSort` – which sorts using heapsort

The procedures below take an integer array `A` of size n and an array `B` of size k as input. All of the procedures should print the numbers which are contained in both `A` and `B`.

```
1 Procedure Intersection1(A, B)
2 |  for x in B do
3 |  |  if Search(A, x) then
4 |  |  |  print(x)
```

```
1 Procedure Intersection2(A, B)
2 |  for x in B do
3 |  |  if BinarySearch(A, x) then
4 |  |  |  print(x)
```

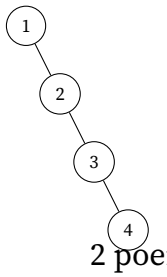
```
1 Procedure Intersection3(A, B)
2 |  HeapSort(A)
3 |  for x in B do
4 |  |  if BinarySearch(A, x) then
5 |  |  |  print(x)
```

```
1 Procedure Intersection4(A, B)
2 |  for x in B do
3 |  |  HeapSort(A)
4 |  |  if BinarySearch(A, x) then
5 |  |  |  print(x)
```

Three of these procedures always give the correct result, whereas one may give an incorrect result. We want to know which variant to use depending on the size of k (i.e. the length of `B`) in relation to n (i.e. the length of `A`).

AVL

poeng



We usually perform AVL-insertion on AVL-trees. However, we can apply the same procedure on ordinary binary search trees (given that each node stores the height of its subtree).

To the right, you see a binary search tree which is *not* and AVL-tree. Insert 5 into the tree using AVL-insertion.

(a) 2 poeng

What will the tree look like after the first rotation during the insertion?

(b) 4 poeng

Note that a double rotation counts as two single rotations.
How many single rotations are performed during the insertion?

(c) 1 poeng

Is the resulting tree an AVL-tree?

(d) 2 poeng

What is the value stored in the root node after the insertion is completed?

Subanagram poeng

We wish to make a website for helping sneaky Scrabble players. A Scrabble player has a set of letters, with which they try to form words from a given dictionary. In this exercise, we can ignore all other Scrabble-rules, and only consider what we call *subanagrams*.

Let us assume that the player's letters are given as a string S, and that a word W is also given as a string. We say that W is a *subanagram* of S if W can be written using only letters from S.

Note that we do not allow the reuse of letters from S. For example, `"hole"` is a subanagram of `"ehlo"`. However, `"hello"` is *not* a subanagram of `"ehlo"`, because `"ehlo"` does not contain two `"l"`'s. Both `"hole"` and `"hello"` are subanagrams of `"hheelloo"`.

Here are some examples of subanagrams of `"aghilmort"`:

- `"algorithm"`
- `"logarithm"`
- `"alright"`
- `"right"`
- `"math"`
- `"git"`

In the following exercises, we consider different ways of finding subanagrams for a given dictionary.

(a) 4 poeng

ALGORITHM: IS W A SUBANAGRAM OF S?

Input: A string W and a string S

Output: Returns **true** if W is a subanagram of S, **false** otherwise

```
1 Procedure IsSubanagramOf1(W, S)
2     r ← copy of W
3     for c in S do
4         if c is in r then
5             remove an occurrence of c from r
6     return r.length = 0
```

The algorithm above checks whether a word W can be written using letters from S.

- Assume that a string works just like an array of letters.
- Copying an array is done in linear time.
- Checking whether an element is in an array is done in linear time.
- Removing an element from an array is done in linear time.
- Finding the length of an array is done in constant time.
- We let w specify the size of W and s specify the size of S.

What is the runtime complexity of the algorithm?

(b) 7 poeng

You will now implement an alternative to `IsSubanagramOf1` using *frequency tables*. Suppose you have a procedure `FreqTable` available, which builds a map from letters to the number of occurrences in linear time.

If F is a frequency table for the string `"abbcccdddd"`, then F.`get("a")` returns 1 and F.`get("d")` returns 4. You can assume that F.`get`(x) returns 0 for all letters that are not in the word the frequency table was built from. You can update the value in the frequency table by use F.`put`, just as with a regular map. Both `get` and `put` are in $O(1)$.

Complete the implementation of `IsSubanagramOf2` so that it has runtime complexity $O(w + s)$, where w is the size of W and s is the size of S.

ALGORITHM: IS W A SUBANAGRAM OF S?

Input: A string W and a string S

Output: Return **true** if W is a subanagram of S, **false** otherwise

```
1 Procedure IsSubanagramOf2(W, S)
2 |   F ← FreqTable(W)
  |   // Fill in
```

(c) 5 poeng

Strategy 1

```
1  Procedure SubanagramsOf1(D, S)
2      r ← empty list
3      for i ← 0 to d − 1 do
4          W ← D[i]
5          if IsSubanagramOf2(W, S) then
6              add W to r
7      return r
```

In the pseudocode above, we are given a dictionary `D` of length d, which is an array of strings. It uses `IsSubanagramOf2` from the previous exercise to find all words in the dictionary which is a subanagram of S. Finding subanagrams using this technique runs in $O(d \cdot (w + s))$, where d is the size of the dictionary, w is the size of the largest word in the dictionary, and s is the size of S.

Strategy 2

```
1  Procedure Build(D)
2      M ← empty map
3      for i ← 0 to d − 1 do
4          sortedword ← Sort(D[i])
5          if M.get(sortedword) is empty then
6              M.put(sortedword, empty list)
7          add D[i] to M.get(ws)
8      return M
9  Procedure SubanagramsOf2(M, S)
10     r ← empty list
11     for x in sorted substrings of S do
12         append M.get(x) to r
13     return r
```

In the pseudocode above, we create a hashmap where each key k being a string of letters sorted alphabetically, and the value is a list of anagrams of k found in the dictionary. To find all subanagrams in the dictionary, we must look up all sorted substrings of S in the hashmap. To find subanagrams using this technique runs in $0(2^s)$, i.e. exponential time with respect to the size of S.

Discuss advantages/disadvantages of strategy 1 and strategy 2 with respect to runtime.

(d) 6 poeng

Now that our website can find subanagrams of a string S, all that remains is to present the results to the user. Assume the subanagrams returned by `SubanagramsOf2` are given as an array, sorted *alphabetically*. We want the subanagrams to be sorted by *length*, and that words of the same length are sorted alphabetically, as in the following example:

Input	**Output**
`"algorithm"`	`"algorithm"`
`"alright"`	`"logarithm"`
`"git"`	`"alright"`
`"logarithm"`	`"right"`
`"math"`	`"math"`
`"right"`	`"git"`

Fill in the pseudocode below. Your procedure should have as low runtime complexity as possible.

ALGORITHM: SORTING OF SUBANAGRAMS

Input: An array A of size n containing strings

Output: An array with the same n strings sorted by length where words of the same length are alphabetized

```
1 Procedure SortSubanagrams(A)
  |  // Fill in
```

Four kinds of graphs poeng

You want to improve the communications network in a small town. Historically, all communication between households in the city has taken place with the help of homing pigeons. In this manner, everyone could reach each other's houses directly. But homing pigeons are impractical, so you wish to find better ways for households to communicate with one another. You have explored the following possibilities:

- The new startup HamiltonBikes has begun building new bike routes and will operate a letter delivery service between all houses. They guarantee that you can take a bike ride around the whole city without having to pass the same house twice.
- Houses can be connected with network cables. In order to mitigate cost, redundant connections will be avoided.
- Some households are so close that neighbors can shout for quick, one-to-one communication.

You realize that each form of communication can be formalized as a graph where the nodes are the households and the edges are the connections between them. You identify four types of graphs, each of which corresponds to a form of communication:

Type 1 is an undirected complete graph (homing pigeons)
Type 2 is an undirected graph with a hamiltonian cycle (bike route)
Type 3 is an undirected tree (network cables)
Type 4 is an undirected non-connected graph (shouting)

All graphs in this exercise has more than two nodes.

There is no need to memorize the graph types, they are repeated where needed.

(a) 7 poeng

Your first task is to check whether HamiltonBikes really delivers what they promise. They offer to send you a map of the bike paths and a suggested bike route which passes each house precisely once. You translate the problem into a decision problem:

Type2	
Instance:	A graph G
Question:	Does G contain a hamiltonian cycle?

You decide to create an efficient algorithm that verifies **Type2**. The algorithm takes a graph and a certificate as input, and answers YES if the certificate confirms that the graph is of type 2, NO otherwise. The algorithm should run in polynomial time.

HamiltonBikes delivers the certificate as an array C of nodes in G, so that each node is included exactly once. Write pseudocode for a verifier `Type2Verifier`.

ALGORITHM: VERIFIER FOR **TYPE2**

Input: A graph G and an array of nodes C
Output: Returns YES if C shows that G is of type 2, NO otherwise

```
1 Procedure Type2Verifier(G, C)
    // Fill in
```

(b) 3 poeng

Explain briefly (maximum 4 sentences) why the answer to the previous exercise shows that the decision problem **Type2** is in NP. Do not include a detailed runtime analysis of your implementation of `Type2Verifier`.

If your answer to the previous exercise has shortcomings, you can make necessary assumptions to answer this exercise.

(c) 3 poeng

Type 1 is an undirected complete graph (homing pigeons)
Type 2 is an undirected graph with a hamiltonian cycle (bike route)
Type 3 is an undirected tree (network cables)
Type 4 is an undirected non-connected graph (shouting)

All graphs in this exercise has more than two nodes.

One advantage of the HamiltonBikes bike paths is redundancy. For example, if all the bike paths around a house have to be closed due to construction work, it is still possible to cycle around the city.

Explain briefly (maximum 4 sentences) why type 2 graphs are biconnected.

(d) 8 poeng

Type 1 is an undirected complete graph (homing pigeons)
Type 2 is an undirected graph with a hamiltonian cycle (bike route)
Type 3 is an undirected tree (network cables)
Type 4 is an undirected non-connected graph (shouting)

All graphs in this exercise has more than two nodes.

The city council does not understand graph theory, so they want you to demonstrate how one can get around the city if construction work is being done. You decide you to write a program that prints two distinct paths from a given house to another.

You start with a graph $G = (V, E)$ of type 2, as well as two nodes s and t in the graph.

Write pseudocode for a procedure `TwoPaths` that prints two distinct paths from s to t.

You have been given a certificate C as described in exercise **??**.

Do not worry about the print format; the order in which nodes are printed is most important.

ALGORITHM: PRINTS TWO DISTINCT PATHS FROM s TO t

Input: A graph G, two nodes s and t and a certificate C
Output: Prints two distinct paths from s to t

```
1 Procedure TwoPaths(G, C, s, t)
  | // Fill in
```

(e) 3 poeng

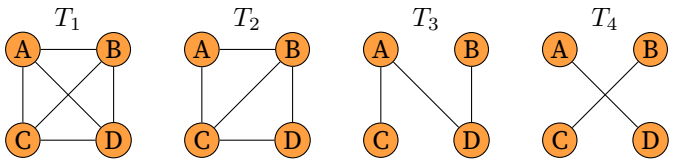
Type 1 is an undirected complete graph (homing pigeons)
Type 2 is an undirected graph with a hamiltonian cycle (bike route)
Type 3 is an undirected tree (network cables)
Type 4 is an undirected non-connected graph (shouting)

All graphs in this exercise has more than two nodes.

We wish to get an overview of the methods of communication in the city. Therefore, we assign weights to the different edge types based on cost. Then we merge these into a graph S. We only keep the edge with the lowest weight if there are parallel edges.

Here are four sample graphs, one of each type, where

- edges in T_1 have weight 10
- edges in T_2 have weight 8
- edges in T_3 have weight 3
- edges in T_4 have weight 1



Fill out the weights of the edges in the graph S.

(f) 4 poeng

Type 1 is an undirected complete graph (homing pigeons)
Type 2 is an undirected graph with a hamiltonian cycle (bike route)
Type 3 is an undirected tree (network cables)
Type 4 is an undirected non-connected graph (shouting)

All graphs in this exercise has more than two nodes.

You construct the merged graph G in the same manner as in the previous exercise, given the following graphs:

- G_1 of type 1, where every edge has weight 10,
- G_2 of type 2, where every edge has weight 8,
- G_3 of type 3, where every edge has weight 3,
- G_4 of type 4, where every edge has weight 1.

You now wish to to find the cheapest way to connect the houses in the city. Therefore, you have decided to use Kruskal's algorithm on the merged graph G. Answer the questions below and briefly justify your answer (maximum 4 sentences).

- Is there exactly one cheapest way to connect the houses in the city?
- Is it possible for everyone in the city to communicate only by shouting? In other words, could we have built a spanning tree for G, with only edges from G_4?
- Your colleague says that if G_4 has no edges, then the minimum spanning tree for G is exactly G_3. Is this correct?
- What is the weight of the *last* edge Kruskal's algorithm chooses when building the spanning tree?

(g) 2 poeng

Type 1 is an undirected complete graph (homing pigeons)
Type 2 is an undirected graph with a hamiltonian cycle (bike route)
Type 3 is an undirected tree (network cables)
Type 4 is an undirected non-connected graph (shouting)

All graphs in this exercise has more than two nodes.

Let $G = (V, E)$ be a graph and let s and t be nodes in V. We will now find the length of the shortest path from s to t.

- Which algorithm solves the problem most effectively if you know that G is of type 1 and all the edges have a weight of 10?
- Which algorithm solves the problem most effectively if you know that G is of type 2 and all the edges have a weight of 8?

(h) 8 poeng

You wish to find the shortest route of communication from the post office (A) to all houses in the city, formalized as nodes in the set V. You have represented all types of communication in the following sets of edges:

- E_1 contains an edge between every pair of nodes in V.
  All edges in E_1 has weight 10.
- $E_2 = \{\{A, B\}, \{B, C\}, \{C, D\}, \{D, E\}, \{E, F\}, \{F, G\}, \{G, H\}, \{E, H\}, \{F, H\}, \{F, A\}\}$
  All edges in E_2 has weight 8.
- $E_3 = \{\{A, C\}, \{B, C\}, \{C, D\}, \{D, E\}, \{E, F\}, \{F, G\}, \{G, H\}\}$
  All edges in E_3 has weight 3.
- $E_4 = \{\{A, D\}, \{C, D\}, \{B, D\}, \{F, G\}\}$
  All edges in E_4 has weight 1.
- $V = \{A, B, C, D, E, F, G, H\}$

Find the shortest paths from A to all other nodes. You can use edges from all four sets of edges. Remember that there can be several paths of communication between houses, i.e., there are parallel edges between the nodes.

Fill out the table with the correct minimal distance from A for each node in V.

HTML poeng

You have taken on a job at a company that has an older website written in pure HTML. They discovered recently that the website works very poorly. They themselves have given up, since they lack the necessary expertise. Their final hope is that the young and promising computer scientist will find out what in the world is going on. The website consists of thousands of handwritten HTML pages; any talk of “making something new and fresh” is met with a puzzled look.

It does not take you very long to find a recurring error in many of the faulty pages: many of the handwritten HTML files are incorrectly formatted! You notice that in many places a `<div>` tag is opened, but is never closed with a `</div>`. Other places a closing `</div>` tag occurs without ever having been opened. Here is an example of one incorrectly formatted HTML file:

```
<div>
  <div>
    Velkommen til nettsiden vår!
  <div>
   Se våre nyeste saker!
    <div>
    Vi har ansatt en ung person! Kanskje blir nettsiden oppdatert nå?
    FØLG MED PÅ UTVIKLINGEN. Hilsen Kåre.
    </div>
</div>
```

(a) 4 poeng

You realize that the first thing you need to do is to find all the files that contain errors. After a bit of Unix magic, you are left with files where all text except `<div>` and `</div>` have been removed. Everything runs smoothly until you have to solve a small puzzle:

How to check that each tag that is opened with `<div>` is also closed with `</div>`, and that no tag is closed before it is opened?

Your task is to write a procedure `GoodDivs` that checks this property, i.e., that `<div>` tags are well-formatted. It takes an array `A` as an argument where each element is either the string `"<div>"` or the string `"</div>"`.

ALGORITHM: DECIDE IF `<div>`-TAGS ARE WELL-FORMATTED

Input: An array of `"<div>"` and `"</div>"` of length n

Output: Returns **`true`** if the tags are well-formatted, **`false`** otherwise

```
1 Procedure GoodDivs(A)
    // Fill in
```

(b) 8 poeng

Assume that the method `GoodDivs` works properly and finds many files that contain incorrectly formatted HTML. Unfortunately, it does not detect all the files reported erroneous. You notice that they contain the same error, however with other types of tags! So it is not just the `<div>`-tags causing problems, but other tags like `<head>`, `<body>`, `<p>` and several others as well. You realize that the procedure `GoodDivs` must be generalized to a procedure `GoodTags`.

How to check that each tag that is opened is also closed with a tag of the same type, and that no tag is closed before it is opened?

You already have a simple procedure `isOpen`. It takes a string and returns **`true`** if it is an open tag and **`false`** if it is a closing tag, i.e., `isOpen("<tag>")` returns **`true`** and `isOpen("</tag>")` returns **`false`**. In addition, you have a procedure that returns a tag's type, e.g., `TagType("<div>")` gives `"div"`, and `TagType("</p>")` gives `"p"`.

ALGORITHM: DECIDE IF THE TAGS ARE WELL-FORMATTED

Input: An array A of tags of length n
Output: Returns **`true`** if the tags are well-formatted, **`false`** otherwise

```
1 Procedure GoodTags(A)
    // Fill in
```

Hint: A clever choice of data structure may be very helpful here.