Exam IN2010 fall 2021

10. December 2021

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.
- Answers are entered in Inspera, and there is no possibility of uploading handwritten answers.
- No aids are allowed.

Comments and tips

- It is recommended to briefly read through the exam before you start. The full set of exercises is attached as a PDF.
- *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*.
- An *easy to comprehend, unambiguous, and precise* natural language explanation, may yield more points than pseudocode that is difficult to comprehend, ambiguous or imprecise.

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.

Sorting — 10 poeng

In the exercises below, assume that A is an array of size n, and i is an integer $0 \leq i < n$.

Bubble sort

(a) After i iterations of the outer loop of Bubble sort, the i *first* elements are sorted.

(b) After i iterations of the outer loop of Bubble sort, the i *last* elements are sorted.

(c) Bubble sort only swaps adjacent elements.

(d) Bubble sort guarantees a minimal number of swaps.

Selection sort

(e) After i iterations of the outer loop of Selection sort, the i *first* elements are sorted.

(f) After i iterations of the outer loop of Selection sort, the i *last* elements are sorted.

(g) Selection sort only swaps adjacent elements.

(h) Selection sort guarantees a minimal number of swaps.

Insertion sort

(i) After i iterations of the outer loop of Insertion sort, the i *first* elements are sorted.

(j) After i iterations of the outer loop of Insertion sort, the i *last* elements are sorted.

(k) Insertion sort only swaps adjacent elements.

(l) Insertion sort guarantees a minimal number of swaps.

Stability **3 poeng**

Assume that the array A is unsorted and contains person-objects, each of which has a age-field. Furthermore, assume that the person-objects at A[3] and A[42] are both 22 years of age.

The array A is sorted by age. After sorting, you know that:

- The person-object that was at A[3] before sorting, is now at A[9]
- The person-object that was at A[42] before sorting, is now at A[7]

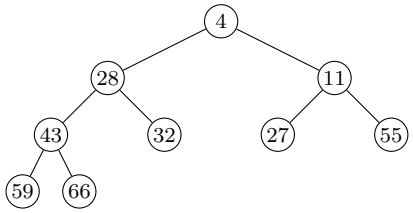
(a) Was the sorting stable?

(b) What is the age of the person-object at A[8] after sorting?

(c) If you know that no person in A is older than 100 years of age, what sorting algorithm should you apply, with regards to run-time efficiency?

Binary heaps — 6 poeng

Take the following binary heap H_1, where both the tree-representation and array-representation are given:



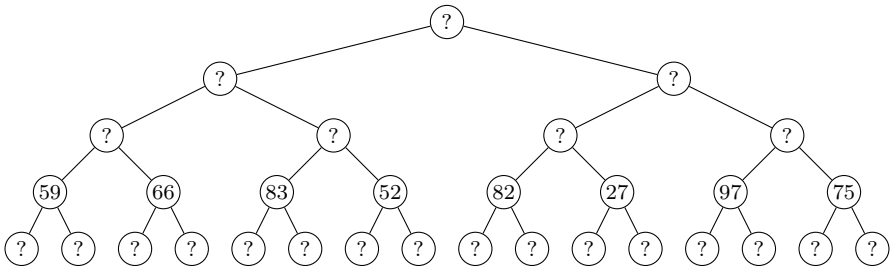
4	28	11	43	32	27	55	59	66
0	1	2	3	4	5	6	7	8

Fill out the tables for each exercise in Inspera. The tables correspond to the array-representation of H_1 after applying a given operation. The questions are independent, meaning that H_1 always refers to the heap as given above, and any mutation to the heap in one exercise does not persist to the next.

(a) What does the heap look like after one call to H_1.`RemoveMin`()?

(b) What does the heap look like after one call to H_1.`Insert`(7)?

In another binary min-heap H_2, you are only provided the values at depth 3:



(c) What is the smallest value that can be located at depth 4 of H_2?

Huffman trees **5 poeng**

We want to compress a string, consisting of 5 distinct symbols a,b,c,d og e. The relative frequencies are given by the following frequency table:

a	b	c	d	e
2	1	12	1	4

(a) What is the longest code-length for a symbol in the corresponding huffman tree?

(b) How many bits does the code for the symbol e contain?

(c) How many bits are used to code the string aaabbcdee?

Now, we will not consider any concrete huffman tree for a given frequency-table, but rather consider huffman trees in general.

(d) If a frequency table consists of 8 distinct symbols, how many nodes are there in the corresponding huffman tree?

(e) In a huffman tree with 8 leaf-nodes, what is the longest possible code-length for a symbol?

Computability and complexity — 8 poeng

For all of the questions below, answer **True** or **False**.

(a) It is proven that $P = NP$.

(b) It is proven that $P \neq NP$.

(c) All NP-complete problems may be reduced to each other in polynomial-time.

(d) All problems in P are also in NP.

(e) All decision problems are either in P or NP.

(f) All problems where YES-instances may be verified in polynomial time, are in NP.

(g) If someone manages to solve the decision problem **Hamiltonsykel**, **Knapsack** or **Sudoku** in polynomial time, then they have proved that $P = NP$.

(h) All decision problems may be solved with a sufficiently fast computer.

Shortest distances in a graph — 8 poeng

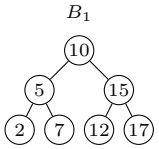
For each type of graph, choose the fastest algorithm which finds the shortest distances from a start vertex to all other vertices in the graph.

	Breadth-first search	Dijkstra	Topological sort	Bellman-Ford
No negative edges				
Weighted DAG				
Unweighted				
No negative cycles				

- Unweighted: an unweighted graph. The graph is either directed or undirected.
- Weighted DAG: a weighted, directed acyclic graph. The edges may have negative weights.
- No negative edges: the graph is weighted, but no edge has negative weights. The graph is either directed or undirected.
- No negative cycles: the graph is weighted, but contains no cycle of negative weight. The graph is either directed or undirected.

Range in a binary search tree — 10 poeng

You are given a binary search tree B and two integers a and b where $a \leq b$. Provide an algorithm that prints the values within the range $[a, b]$ in *sorted* order (ascending). That is, if x is the value of a node, then the node should be printed if and only if $a \leq x \leq b$.



Input: A binary search B and two integers a and b where $a \leq b$
Output: Prints the values within the range $[a, b]$ in order

```
1 Procedure InRange(B, a, b)
  | // ···
```

For the binary search tree B_1 (shown in the example above), `InRange(`B_1`, 2, 7)` should print 2, 5 and 7, and `InRange(`B_1`, 10, 20)` should print 10, 12, 15 and 17.

(a) Provide your algorithm (with pseudo code). You may make reasonable assumptions regarding the representation of B. A more efficient algorithm gives a higher score. You do not need to consider the formatting of the output.

In the following exercises, assume that B has n nodes, is balanced, and does not contain any duplicates.

(b) Assume that a is the smallest value in B and b is the greatest value in B. Give the run-time complexity of your algorithm with respect to n.

(c) Assume $a = b$. Give the run-time complexity of your algorithm with respect to n.

Most frequent — 8 poeng

Strategy 1

Input: An array A of n positive integers
Output: The most frequent number in A

```
1  Procedure MostFrequent1(A)
2      H ← new HashMap()
3      for x in A do
4          H.put(x, 0)
5      m ← A[0]
6      for x in A do
7          f ← H.get(x)
8          H.put(x, f + 1)
9          if H.get(x) > H.get(m) then
10             m ← x
11     return m
```

Strategy 2

Input: An array A of n positive integers
Output: The most frequent number in A

```
1  Procedure MostFrequent2(A)
2      k ← maximum element in A
3      B ← array of size k
4      for i ← 0 to k − 1 do
5          B[i] ← 0
6      m ← A[0]
7      for x in A do
8          B[x] ← B[x] + 1
9          if B[x] > B[m] then
10             m ← x
11     return m
```

Above you see two (correct) algorithms that both take an array A of positive integers as input, and return the number that occurs most frequently. Discuss the advantages and disadvantages of the two different strategies. You should state both the run-time complexity in the *worst* case, and the *expected* run-time complexity, with respect to n and k. Justify your analysis *briefly*. You should also state in which situations **Strategy 1** should be preferred over **Strategy 2**, and vice versa, with regard to concrete running time. Be sure to clarify any assumptions you make.

Find pairs that sum to x — 10 poeng

You are given an array of unique integers A and an integer x. Write a procedure that prints all pairs of integers y and z in the array, where $y + z = x$. The number at a given position in the array may only be printed once. The order in which the pairs (y, z) are printed does not matter, and the order within the pairs does not matter either.

Input: An array A of integers, and an integer x
Output: Prints all pairs $y, z \in$ A such that $y + z = x$

```
1 Procedure FindSummands(A, x)
  | // ···
```

For example, a call to `FindSummands(`$[0, 2, 4, 6, 8, 10]$`, 10)` should print the pairs $(0, 10)$, $(2, 8)$ and $(4, 6)$.

For both exercises: Provide pseudocode for an algorithm that solves the problem **and** provide the run-complexity of the algorithm (no justification is necessary). A lower run-time complexity gives a higher score.

(a) Write a procedure `FindSummands` as described above, with the assumption that A is *sorted in ascending order*.

(b) Write a procedure `FindSummands` as described above, but you may *not* assume that A is sorted. Hint: You may assume $O(1)$ for insertion, lookup and deletion in hash-based data structures.

Two-coloring a graph — 10 poeng

You are given an *undirected and connected* graph $G = (V, E)$. Provide an algorithm that colors the vertices in V with two colors R(red) and B(blue), such that no neighbors $(u, v) \in E$ has the same color; if this *is not possible*, then the algorithm should print that it is impossible.

You may assume that each vertex in V has a field v.color that is initially set to null, and that you are provided with a function `FlipColor`, where `FlipColor(`R`)` $= B$ and `FlipColor(`B`)` $= R$.

Input: A connected graph $G = (V, E)$
Output: A graph where the vertices are colored, if it is possible

```
1 Procedure TwoColor(G)
  | // ···
```

Whops!-settlement — 10 poeng

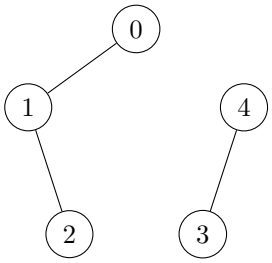
Employees at the Department of Informatics have been to a department seminar where things have spiraled rather out of control with regards to employees lending small amounts of money to each other. Everyone is to settle their expenses by transferring money via the service Whops!. Unfortunately, due to a series of inter-departmental "misunderstandings", many employees have blocked each other on Whops!, making the settlement process difficult. The administration observes the situation with despair, noting that this problem will only continue in the coming years, due to the department's plan to expand dramatically. They have an existing process to determine how much each employee owes or is owed in total. Now, they are asking the students of IN2010 (who are at this point experts in algorithms and data structures) to come up with a general solution to check if a settlement may be realized using Whops! or not, that will scale with the departments planned expansion.

You get to work, formalizing the problem as a graph problem. You let $G = (V, E)$, where every employee is represented with a vertex $v \in V$, and let there be an (undirected) edge $\{u, v\} \in E$ between two employees if they have not blocked each other on Whops!. For each employee $v \in V$, you can lookup in a table T, where $\mathsf{T}[v]$ is an integer representing how much v owes or is owed in total. If $\mathsf{T}[v]$ is negative, then v owes money, but if $\mathsf{T}[v]$ is positive, then v is owed money. You know that the sum of all numbers in T is exactly 0. Note that this means that if everyone was able to exchange money between each other using Whops! (possibly via other employees), a settlement could always be reached.

Provide an algorithm that takes G and T as input and return **true** if a settlement may be reached using Whops!, and **false** if this is not possible.

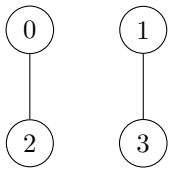
Here are two examples of how G and T may look. In the example of G_1 and T_1, a settlement is reachable through Whops!, but for the example of G_2 and T_2, it is not possible to reach a settlement through Whops!.

G_1



v	$\mathsf{T}_1[v]$
0	500
1	−200
2	−300
3	−50
4	50

G_2



v	$\mathsf{T}_2[v]$
0	300
1	−250
2	0
3	−50

Whops!-logs 10 poeng

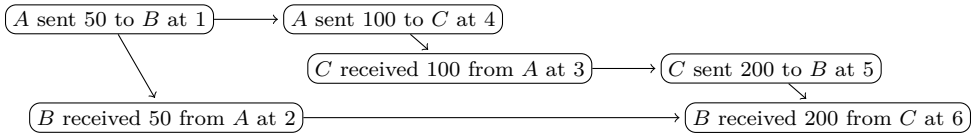
The service Whops! allows users to transfer money between each other using an app. They try to avoid storing information about their users whenever possible, preferring that information is stored on the phones of their users.

Once in a while, they need to get a complete log of transactions within a time period, in order to debug their system. To do this, they fetch the *local* logs from a set of phones in the system. A local log is a list of events, where each event represents a payment being sent or received. Every event contains a timestamp, conveying when the payment was sent or received. For simplicity, the timestamps are represented as positive integers. The following is a small example of three logs from phones A, B and C.

A	B	C
A sent 50 to B at 1	B received 50 from A at 2	C received 100 from A at 3
A sent 100 to C at 4	B received 200 from C at 6	C sent 200 to B at 5

Sadly, the timestamps from the phones are unreliable. They often observe a payment being received *before* the payment was sent, according to the timestamps. In the example above, C receives a payment of 100 *before* A has sent that payment to C.

To approximate a correct log, they construct a directed graph $G = (V, E)$, where each event is represented with a vertex. If two events u and v appear adjacently in a local log, then the graph contains a directed edge (u, v). For every event u that represents a payment being sent, the graph contains a directed edge (u, v), where v is the corresponding reception of the payment. Here is the graph for the example above:



The desired complete log for this example is:

A sent 50 to B at 1
B received 50 from A at 2
A sent 100 to C at 4
C received 100 from A at 3
C sent 200 to B at 5
B received 200 from C at 6

Every vertex v may access the timestamp from the log by $v.\texttt{ts}$. Give an algorithm that takes a directed graph $G = (V, E)$ as input, and prints all vertices in V in an order such that:

- If there is an edge from u to v, then u is printed before v.
- u is printed before v if $u.\texttt{ts} < v.\texttt{ts}$, if this does not violate the first requirement.