UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in	INF2220 — Algorithms and data structures
Day of examination:	14. December 2012
Examination hours:	14:30 – 18:30
This problem set consists of 18 pages.	
Appendices:	None
Permitted aids:	All printed and written

Please make sure that your copy of the problem set is complete before you attempt to answer anything.

Contents

Some general **advice**:

- Make sure your handwriting is **easy to read** (unreadable answers are always wrong . . .)
- Remember to **justify** your answers.
- Keep your **comments** short and concise. If you use well-known data-structures (list, set, map) there is no need to explain how they work or behave. In general: when using basic abstract data types from the library, you can use them without explaining what they do.
- The **weight** of a problem indicates how we estimate its difficulty. You may take that into account as you organize your time.

Good luck!
Dino Karabeg & Martin Steffen

(Continued on page 2.)

Hints for correcting: *In general, the programming language was Java. For the exam, we intend to take it more relaxed in that pure Java typos (like missing semicolons and similar things the compiler would be able to find) etc should not count negatively. The "pseudo-ness" of the code should be within reasonable bounds. If "implementation" is required, we should give say 25% if no code is provided but an algorithmic solution is described sufficently clear such that a (correct) algorithm is identifyably described in the text. Correct means, however, that for instance the "recursion case"* and *the base cases are described. Things like "one should go recursively through the tree and calculate the maximum", i.e., wishy-washy fishing in the dark like that does not give points (even if recursion would be employed in a standard solution).*
In general, we do not care whether a "algorithm" is implemented in as "method" like `n.m(arg1, arg2)` *(where* `n` *is the node of a tree), or as a "procedure"* `m(n,arg1,arg2)`*. Solutions like* `n.m(n,arg1,arg2)` *(i.e., the node is use as callee* and *method argument) leads to 30% less points for coding, since this shows weakness in mastering programming properly.*

Problem 1 Miscellaneous questions (weight 12%)

1a Time complexity (weight 4%)

What is the *worst-case* complexity of the following piece of code, in terms of the input parameter n?

```
int z = 0;
for (int i = n; i >= 1; i = i/2) {
  for (int j = 1; j <= n ; j++) {
      z = z + i + j ;
  }
}
```

Solution: $n \log n$

Hints for correcting: *All or nothing.*

1b SCCs in graphs (weight 4%)

Given the directed graph G with nodes $a,\ldots,f$ of Figure 1.

1. Which are the strongly connected components in G (just list them).
2. Illustrate, how they can be determined algorithmically by giving the stages of the algorithm *step by step*. One step should correspond to one "graph traversal step" (i.e., following an edge/visiting a node in the graph) not more fine-grained.
3. Assume an *un*directed graph. Describe (no actual code required) a way to determine SCCs for undirected graphs which is simpler compared to the way

(Continued on page 3.)

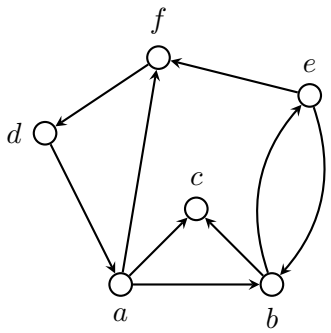


Figure 1: Directed graph

it's done for directed ones?[1] Does this simplification leads to an improvement in the worst-case time complexity? Explain your claim in a few words.

Solution:

1. $$\{\{d, f, a, e, b\}, \{c\}\}$$

2. *The final outcome here must of course be the SCCs given above. Since an animation kind of solution is requested (similar to what we did in the lecture). Different possible runs are of course possible, depending on non-deterministic choices. Basically, there can be two different choices for the first phase: starting at* c *or starting at another node (the solition does the latter). There will be either two dfs's (starting at* c*) or 1, starting anyplace else.*

   One possible first fase is given in 2, I don't give an animation by showing it step by step, but collapse it into one pic where the pair of numbers are: the first one is the visiting time *the second one the* finishing time. *This the numbers shown define uniquely one particular run of an (iterated) DFS, starting at* d *(at "time" 1) and ending at* d *as well (finishing "time"* 12*). If course it's a coincidence that the first DFS finds all nodes (and this the smallest overall starting time is the largest overall finishing time). I show also the DFS-tree, but it's not required.*

   Based on the first phase, the second one must *start at* d *and deliver the first scc* $\{a, b, d, e, f\}$ *before the second scc* $\{c\}$*.*

Hints for correcting:

1. *It's all or nothing. Explanations what SCCs are and how to find etc (without giving the requested answer) don't count anything. How the "list" of SCCs is given does not matter.*

2. *We expect a "animation" kind of solution in the style we had in the lecture. To get full points there, the given pictures must show that the student has understood the core of the algorithm. I.e., the following points must be visible*

   - *the first have must be an (iterated) DFS, the* order *of fininalizing a node ("finishing time") must be clearly visible*

[1] As a side remark: as you remember the notion of *strongly connected components* and just *connected components* coincide for undirected graphs.

(Continued on page 4.)

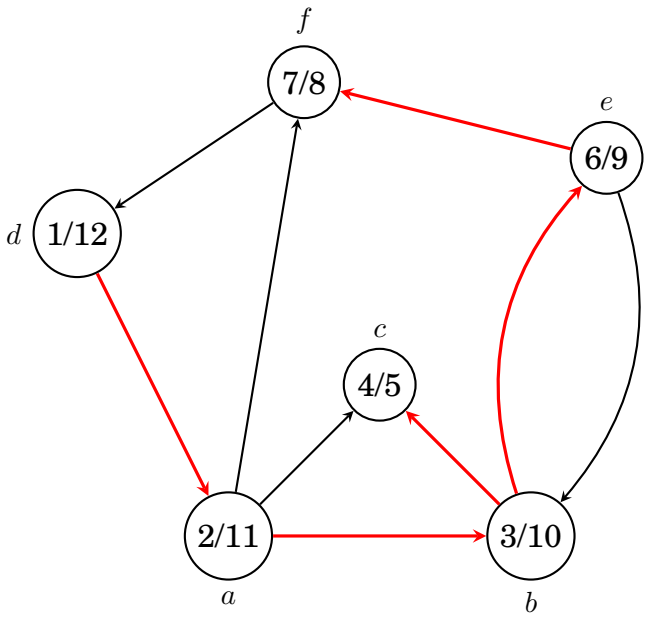


Figure 2: Possible first phase

- *it must be clear that the second (iterated) DFS runs in the graph with "reversed" edges.*
- *the second (iterated) DFS must clearly take the finishing times of the first phase into acount, and the SCCs must be build up during this second phase.*

Unlike in the first sub-task: it's not all or nothing. It's not required that the students depict the dfs-trees *graphically, some may, because that's what we did in the lecture. Giving* only *the DFS-tree without the finishing time is not good enough. It's also not required that the* visiting *times are shown, relevant is only the* order *of the finishing times. A representation as the one given here (not a step-by-step series of pics) is fine too, provided it's accompagnied with a explanation what the numbers mean and how the first phase influences the second.*

1c Boyer Moore (weight 4%)

Show the **good suffix shift** for the *needle* (called "nål" on the Norwegian slides) n the form of a table:

ANANASBANAN

Hints for solving: *That one should be simple and routinely. Write an array which is as long as the needle. Here, the needed is of length* 11*, so the array will go from* 0 *to* 10*, following standard operational procedure. It seems that this word contains all complications that can happen in GSC.*

Solution: *The solution could be written in a table like that. The mis-match column is not required, but this resembles the format as was done in the lecture, so some people*

(Continued on page 5.)

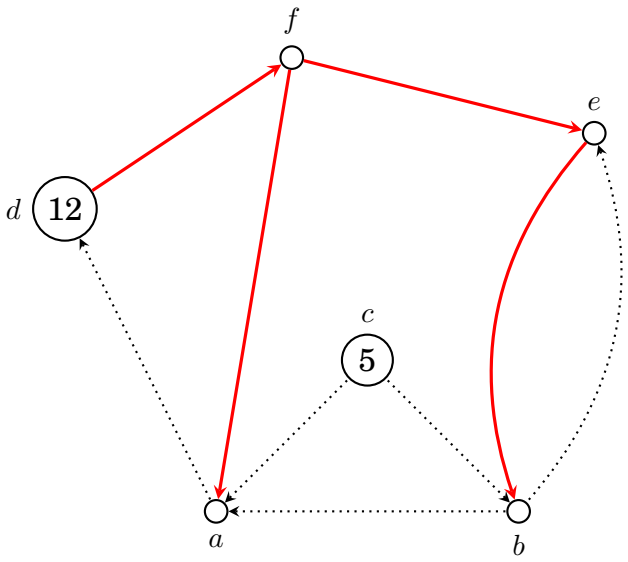


Figure 3: Second phase (based on the first)

will do probably a similar solution.

index	*mismatch*	*shift*	*array*
0	*N*	*1*	
1	*AN*	*11*	
2	*NAN*	*2*	
3	*ANAN*	*11*	
4	*BANAN*	*7*	
5	*SBANAN*	*7*	
6	*ASBANAN*	*7*	
7	*NASBANAN*	*7*	
8	*ANASBANAN*	*7*	
9	*NANASBANAN*	*7*	
10	*ANANASBANAN*	*7*	

Hints for solving: *The needle is of length 11, the array must be of length 11, so the index is from 0 to 10*

Hints for correcting: *Depending on whether the subcases (7, 2, 10 etc) are found the points can be scaled.*

Problem 2 Huffman coding (weight 12%)

2a Huffman tree (weight 2%)

Assume that an input file has given you the following frequency table:

(Continued on page 6.)

letter	frequency
A	2
B	6
C	10
D	7
E	1
F	1
G	3

Draw a Huffman tree based on the frequency table.

Solution: *One solution is given in Figure 4. It's not the only solution. The first 4 letters, when building the tree, are determined (E, F, A, G), so the corresponding sub-tree must occur in all solutions. After that point one has a choice (two trees with weight 7).*

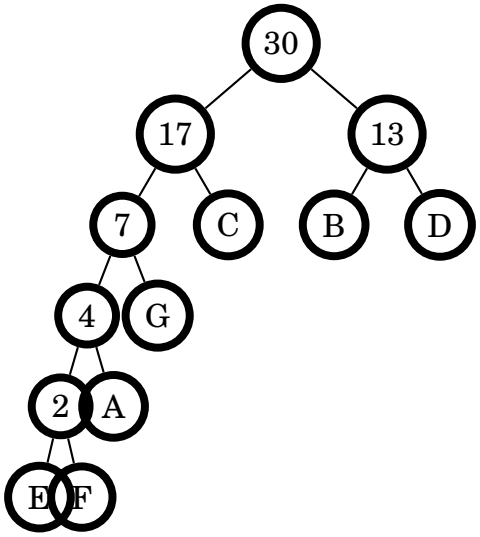


Figure 4: Huffman (one solution)

2b Huffman coding (weight 5%)

Assume an alphabet (= reservoir of characters) of size 16 (say a, b, . . . , p), and a text of length 100 characters. Assume you have a Huffman encoding where the letter a is encoded by a single bit, say 0. What conclusion can one draw about the *frequency* of the letter in the given text?

Solution: *To have a code of length 1 means: in the run of huffman, the letter a must be treated* only *in the last step, i.e., not combined with any other letters until the last combination. In the step before the last step there are 3 "trees" (one of thema) and no tree is allowed to be smaller than* a.
Assuming first that not all letters are actually used. Then the reasoning goes like this: If a letter has frequency 50, he has necessarily a code length 1. If 49, then it's not guaranteed, there could be two letters 49 and 2, and then 1 letter with 49 will not be coded with length 0. So no letter can be must be longer
it must never be that the other (100-a)/2 larger than the frequency of a.

(Continued on page 7.)

Hints for correcting: *Full points for statement over frequency and short explanation. We do not except a full formula, because we have not specified how exactly we mean. A sure conclusion is: to be possible, the* `a` *must be taken in the last step. Since there are potentially different trees, the argument must be to think: what can* prevent *that from happening. The following forumula states: if you divide the rest of the alphabet into two halves, then* a *with one bit is possible, if in the second-to last step (with 3 "trees") it's possible to take the other two:*

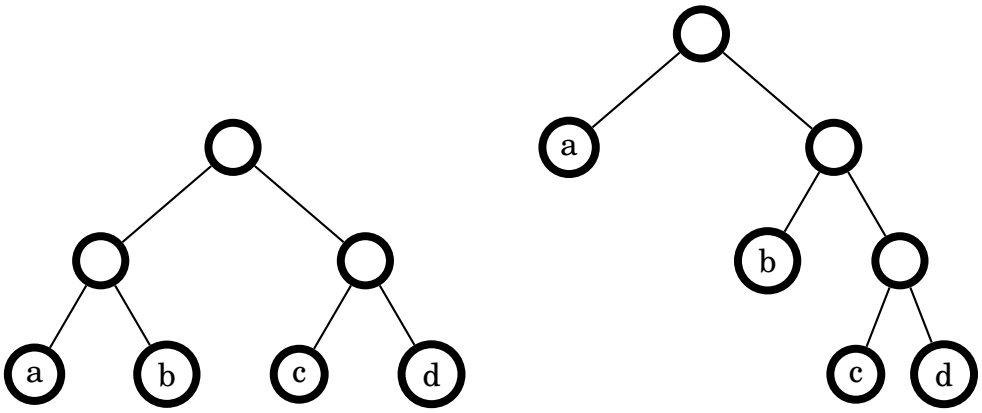
$$\sum_{x \in A_1} x \leq a \quad \textit{and} \quad \sum_{x \in A_2} \leq a \quad \textit{and} \quad A_1 \cup A_2 \cup \{a\} = \{a, \ldots, p\}$$

Hints for solving: *from the TA: The f(A) >= f(X) and f(A) >= f(Y) where X and Y are the niece/nephew sub-trees of A in the Huffman tree (and the f(A) >= ceil(f(T)/3) given by this relation where T is the whole tree), may not be so easy to see right away. Should be given a higher weight than 2A. I think this is a good problem, because it is not automatic points, and you may have to think and draw a little bit to see the solution.*

2c Huffman coding (weight 5%)

In general, a given text may have more than one Huffman encoding, all achieving optimality as defined in the lecture. Assume some text using an alphabet with 4 letters `a`, `b`, `c`, and `d`.

> *Question:* Is it possible, that the text has 2 different Huffman encodings corresponding to the left resp. the right binary tree?



In case of *Yes*, show it by giving an example (i.e., a frequency table for the 4 letters) that this is possible. Or, in case of *No*, provide an argument that this is not possible.

Solution: *The answer is yes. An easy frequency table is the following;*

`a`	*2*
`b`	*2*
`c`	*1*
`d`	*1*

(Continued on page 8.)

Hints for correcting: *It's all or nothing. A correct guess without explanation gives zero point.*

Hints for solving: *One TA found that one easy.*
For me (ms), the task is not 100% easy. At first sight, it seems impossible: an optimal code, having a balanced tree, then one should not have an unbalanced one as well. Let's start with c *and* d*. In both codings they are the siblings, and they are in tendency also the ones furthest away, so in tendency they should have the lowest frequencies, and not much generality is lost assuming they are both equal, say both have frequency* 1*. Both together have frequency* 2 *which is they weight of their common father (in both trees, as said, they are immediate siblings). Now this is the of the trees which is identical in* both *encodings. Since, the placement for this combined node is* different *comparing both trees. In the unbalanced one, the* cd *node of weight* 2 *is a minimal one. I.e., the weights of the remaining two must be* ≥ 2*. (Actually, since* b *is "favored" over* a *in the unbalanced tree we know* $a \geq b$*. If it were* properly $<$*, the unbalanced tree would be impossible. At that point whe have already a solution:* $a = b = 2$ *and* $b = d = 1$*. The size of the encoding is.*

$$2 \times 2 + 2 \times 2 + 1 \times 2 + 1 \times 2 = 12$$

For the unbalanced tree it's

$$2 \times 1 + 2 \times 2 + 1 \times 3 + 1 \times 3 = 12$$

So that's fine then.

Problem 3 Topological sorting (weight 12%)

3a Checking for compatible sorting (weight 9%)

Assume a *directed, acyclic graph* (DAG) $G = (V, E)$. *Furthermore*, you have given the nodes (all of them) of V in a fixed sequence. Implement an efficient algorithm which *checks* whether the given sequence is a valid *topological sorting* of the given DAG.

Solution: *There are two obvious solutions to that.*

Direct implementation: *It's an easy adaptation of top-sort. The top-sort is based on the following idea: calculate the "in-degree" of all the nodes, and then do the following*

1. *"find" a node with in-degree 0*
2. *pick it as next in the sorting*
3. *decrease the in-degrees of it's neighbhors by one (since the picked node is considered done)*

That's the standard one. This needs to be adapted in such a way, that while doing the above steps *it's* checked *at the same time that a node with indegree* 0 *is the next in "given order". We did not specify how the order is actually given. Students should come up with a decent solution, such as an array or a list.*

(Continued on page 9.)

Smartass solution: *Another way to do it is as follows: realize that the "given order" is a DAG in itself. So we have 2 DAGs with the same nodes! One can add the "given order" as additional edges to the given DAG and then check for asyclicity i.e., try whether this combined graph has a topological sorting. If it has, the answer is yes.*

Hints for correcting:

Smartass solution: *since this is an implementation task, this solution needs to program how the two DAGs are combined, i.e., how the list/array (or separate graph) are actually merged. Words decscibing that this "should be done" first is not a (full) solution.*

3b Complexity (weight 3%)

What is the time complexity of your solution?

Problem 4 Optimal money change (weight 12%)

Design an algorithm which calculates *money change* in an optimal way in the sense of not giving unnecessarily many pieces. What is *fixed* is the currency system, i.e. the selection of available coins, and available are the following six *kinds* of coins (you may think "cent coins"):

1, 2, 5, 10, 20, 50.

The input and output of the algorithm are as follows:

Input: The *input* is a non-negative, integer *amount of money*.

Output: The output is an integer corresponding to minimal number of pieces of coins required to sum up to the required amount of money.

For illustration: Given the amount of money *72*, the following two selections of coins sum up to the correct amount, but the one in the second line uses less pieces of coins:

$$\begin{aligned} 72 &= 1+1+10+10+10+10+10+10+10 && \Rightarrow 9 \text{ pieces} \\ 72 &= 20+5+20+5+20+2 && \Rightarrow 6 \text{ pieces} \end{aligned}$$

Your algorithm should give the minimal number of coins for all inputs. The available kinds of coins are *fixed* as given above.

Solution: *The obvious solution uses a* greedy *approach. I would go for a loop (recursion is possible too, but I guess most opt for a loop). In the program, adapted in the loop there will be three data items/structures.*

1. *the largest "still sensible" coin*
2. *the remaining amount that still needs to be changed*

(Continued on page 10.)

3. *"list" of coins already changed appropriately. Actually, one does not need the list, since only the* number *of coins is requested, not which coins are actually used. So this data structure can be an integer. I use the list here, the loop invariant is clearer. The number of coins is abviously just the length of the list*

Initially *: the the largest sensible coin = 50, the remaining amount to be changed = input, the list of coins already changed = empty.*

Loop step: *Check the largest sensible coin against the money still to be changed, if it fits: largest sensible coin unchanged, list of money to change decreased, largest sensible coin added to the list of if changed money. If it does* not *fit: take the next smaller coin in the currency system as largest sensible. The other 2 data structures are unchanged.*

Exit condition: *money to be changed still = 0.*

Loop invariants: *Sum of the money already changed (in the list) plus the money still to change = original input. The list of money already changed contains coins in decreasing order (oldest entries largest), since the algo is greedy. The smallest entry in this list/all entries in this list (when not empty) are* $\geq$ *the current "still sensible" coin*

Hints for correcting: *The task does not require to state complexity or argue that the algo works (it depends on the currency system, so it's not 100% trivial). I would expect that the above solution (greedy) is the only reasonable one. If one comes up with a non-greedy one (especially exhausive combinatorial search), even if that works it will not give full points. Even if we have not requested a good or given time complexity, abstruse solutions should not give full credit.*

Hints for solving: *The solution would be based on a greedy approach. On the one hand that it rather obvious. First of all, the exercise is called* optimization *and greedyness was discussed in the context of that parts. Second, it may correspond to everyday's experience how to handle cash. Finally, the task in itself is not hard anyway. Therefore, I would not give the hint "go for a greedy approach" it should be part of the task.*
What is less good is: on closer inspection it's actually not obvious *that the greedy approach actually works. With a different "currency system", greedy would not succeed (or even their may be amounts of money that cannot be represented at all). Greedy would also fail sometimes, if one does not have an unliminted supply of coins, but a limited one, like a purse. Since in the task there is lurking something complicated (which only for the given currency system does not apply), it may be confusing. Also we should not ask for an argument that the programmed solution is optimal if the student finds the greedy-approach.*

> *TA: It seems this exercise should have its complexity analyzed as it makes a point out of the set of coin types is fixed, but it could also imply the way the code should look (many sequential loops, a single nested loop with reverse sorted coin types, or many if-else-if blocks in a loop). The first and second solutions are slightly better (smaller amplitude linear algorithms), and the second has shorter code and as such is better, the last one doesn't seem great (high amplitude linear), but may seem implied by telling them the set is fixed.*

(Continued on page 11.)

Problem 5 Binary trees and binary search trees (weight 20%)

This task checks whether a binary tree is actually a binary *search* tree. That will be done in steps, which can be solved independently. For instance, you can solve problem **5b** *assuming* you have solved problem **5a**, even if you have not (yet) implemented that part, etc. Following the indicated order of sub-tasks may help, though.

Common to all sub-tasks: You are given a *binary* tree whose nodes carry non-negative integer values (the keys). These values, however, do *not* necessarily satisfy the binary search-tree condition.

Hints for solving: *The challenge here is to get a decent performance. First one has to remember that the binary trees are defined recursively, and in an easy manner. However, the search-tree condition is not defined in the same direct way, which would give rise to a really simple algorithm.*
The following "definition" of a BST is, of course, a "recusive mis-conception":

> *A BST is a binary tree where (if not empty in the base case) the key of the parent is $\geq$ than the node of the left child and $\leq$ than the key of the right child (if they exist) and where* recursively *both children (if exist) are BSTs themselves.*

With this definition, one would get a straightforward and efficient ($\log n$) check, but wrong of course. The definition is more subtle, since the key of a node must be $\geq$ for all *nodes in the left sub-tree for instance. So that's a* non-local *condition (unlike the binary-ness, and like "being balanced" or like being a RB-tree).*
The challenge therefore is to get a decent efficiency in the face of this non-locality.

5a Minimum & maximum key (weight 5%)

Now:

> Program a method `min_key` which calculates the *minimum* of all keys in the tree. Same for the *maximum* (call it `max_key`). Your solution should have a time complexity of $\mathcal{O}(n)$ (= linear in the number of nodes).

Hint: If you want (or your solution has to) invoke `min_key` on an *empty* tree without nodes (and thus without keys), you can use as result in this case `Integer.MAX_VALUE` (the largest integer available). Analogously, the `max_key` of an empty tree can be assumed to be 0 (the smallest non-negative integer).

Solution: *Simple recursion. I made a functional solution for simplicity. The `Leaf` in that solution corresponds to what had been given as a hint for "empty trees", namely one can use `MAXINT` as return value. In the functional setting that's represented by `Leaf`-constructure. In Java, it would depend on concretely how the student would implement the task, either as a procedure or a method of the nodes, in the latter case, probably the empty tree would be represented by a null pointer (which would have to be checked, to protect the "base cases", either in the method itself, or by the caller, such that the method is never applied to null-pointers.*

(Continued on page 12.)

Listing 1: determine min

```
let rec minkey_bt (t: btree)  =
  match t with
    Leaf -> maxint                              (* assumed: maximal *)
  | Node (tl, (k,e), tr) -> min k  (min (minkey_bt tl) (minkey_bt(tl)));;
```

Hints for correcting: *I can't think of any other solution than a recursive one as the one given above. Missing the base cases, unchecked null pointers gives 50% off. If null-pointers are checked, in the sense that the recursion is never applied to null, however, the* outside *user could apply it to a null pointer leading to an exception, then that gives 1 point off (unless the student explains that the the function is never used that way).*

Hints for solving: *TA:*
Using Math.min and Math.max would simplify this a lot, but you would have to change almost everything to change `min_key` *to* max_key *(only a few keywords and variable names stay the same, and it is easier to write the whole function again, rather than describe the changes):*

```
  if (n == null) {
    return Integer.MAX_VALUE;
  }

  return Math.min(min_key(n.left), Math.min(n.key, min_key(n.right)));
```

5b Checking for search-tree condition (weight 5%)

Again, you are given a binary tree as in the previous task.

> Write a recursive algorithm `is_bst` which checks that the given binary tree is a binary *search* tree, *using* two procedures `max_key` and `min_key`,

which determine the maximum, resp. the minimum key in a tree.

Solution: *This one use the only sensible solution. It's a* direct *translation of the definition of BSTs.*

Listing 2: check for BST-ness

```
let rec isbst_slow (t: btree) : bool =
 match t with
   Leaf -> true
| Node (tl, (k,_), tr) ->
 (maxkey_bt(tl) <= k) & (minkey_bt(tr) >= k) &
 isbst_slow(tl) &
 isbst_slow(tr);;
```

(Continued on page 13.)

A wrong solution (which I expect some people to do) is: check that the node is larger (or equal) than the left child and smaller or equal than the right child. That's not *the definition of BST. Careless memory of the condition (or confusion with the Heap condition where that kind of local checking works may lead to that wrong solution.*
Another solution (less obvious) is to do a in-order traversal and make one counter as the minimal and then check wether it never goes down.

Hints for correcting: *The wrong sketched local solution gives not more than 1 point. Getting the base cases uncovered/wrong gives 50% off.*

5c Complexity (weight 5%)

Assume a "completely unbalanced" binary tree, i.e., one that resembles a linear list, for instance: there is no left-child in the whole tree. Under this assumption:

> State the time complexity of your solution (in the number of nodes of the tree) and explain shortly your result.

You can *assume* that the two procedures `max_key` and `min_key` have complexity $\mathcal{O}(n)$ in the number of nodes of the tree.

Solution: *quadratic. there are two "loops" (recursions), the outer one and the inner one (for max resp. min)*[2]*. So the complexity is like*

```
                    // n = length of list = height of tree
  for i = 0 to n    //  recursion for the outer procedure
    for j = 0 to i  //  inner recursion (max/min)
      ....
```

Hints for solving: *This one is (perhaps) challenging. The danger lies in the fact that the student may not get the obvious solution in the previous exercise. If he gets the "intended" solution, the complexity is, unfortunately $\mathcal{O}(n^2)$. If we consider this danger as too big that the student miss the obvious solution or perhaps even get the real "good one" maybe not really relying on the max/min stuff, we show them the ideal solution, and not ask for it.*

5d Improved run-time (weight 5%)

Improve the implementation to achieve a *better* run-time complexity and which *does not use* `max_key` and `min_key`. Use a recursive procedure `is_bst_help` as helper function which takes two additional integer values as argument, say `low` and `high`. Again: `Integer.MAX_VALUE` is the largest available integer. Your solution should have a complexity of $\mathcal{O}(n)$.

```
boolean public is_bst () {

  return this.is_bst_help(Integer.MAX_VALUE,0);
```

[2]Since the tree is unbablance, only max resp. only min will actually be used, depending on whether the tree is left-unbalanced or right-unbalanced

(Continued on page 14.)

```
  }

boolean public is_bst_help (int low, int high) {

     <    ... fill out ..... >
  }
```

Solution: *We are looking for an accumulator solution (similar to the one illustrated by the fibonacci in the lecture) which may look as follows. The (improvement of) fibonacci was not recursive, though.*

Listing 3: check for BST-ness

```
let rec isbst_accu ((t , low, high) : btree * int * int) : bool =
 match t with
 Leaf -> true
 | Node (tl, (k,_), tr) ->
 (low <= k) & (k <= high) &
 isbst_accu (tl, low, k) &
 isbst_accu (tr, k, high);;
```

Hints for correcting: *Since we pretty much force the solution into one particular direction, I would expect any "creative altertives" (can't imagine any, actually) will give little points if any. Difficult to say, though.*
As usual, leaving the base cases unproteced, uncovered: 50% off. Mixing up max and min/low and high in the induction case: 50% off.

Problem 6 B-Trees (weight 12%)

Hints for solving:

> *TA: Only a single order is given for the trees. The students learn B-Trees with two orders: M (internal nodes) and L (leaf nodes). L should probably be specified to be equal to M.*

6a Insert (weight 3%)

Assume that the above B-tree of Figure 5 has *order* $M = L = 3$. Now: draw the B-tree that results when the value 53 is inserted.

6b Deletion (weight 3%)

Draw the B-tree that results when the value 5 is *deleted* from the tree shown in the above figure.

(Continued on page 15.)

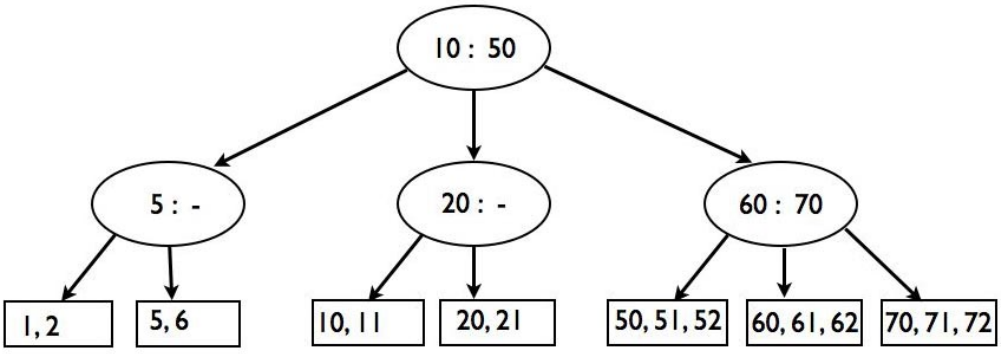


Figure 5: B-tree

6c Complexity (weight 3%)

Consider now a general case of a B-tree of order M that stores n data elements (for some M and n). Assuming that each internal node and each leaf is stored in a separate memory block, estimate the number of disk accesses that are required for inserting a value in the worst case, as a function of M and n.

Solution: $\mathcal{O}(\log_M(n))$

6d Amortized cost (weight 3%)

Generally, node splitting, however, occurs not too often: once a node has been split, it will take several insertions before the same node needs to be split again. The concept of *amortized cost* takes this observation into account.
The *amortized cost* of the insert operation is computed by dividing the number of disk accesses required by a sequence of inserts, by the number of those inserts-by considering the worst-case sequence of inserts. Estimate the amortized cost of the insert operation under the same assumptions as in *task 6c* above, as a function of M and n. Discuss the difference between the efficiency of inserts in 6c and 6d.

Solution: *D. "O(log base M of n / M)- becomes constant time for M>= n" _ is a good - enough answer, get max-1 points. Full credit is given for an attempt to take into account that the internal nodes don't split each time a leaf is split._

Problem 7 Sorting (weight 20%)

We abstractly describe a "pancake sorting" problem. You are given a pile of n pancakes of different sizes. You want to sort the pancakes so that, in the end, smaller pancakes are on top of larger pancakes. The only operation to change the pile is insert a spatula[3] under the top k pancakes, for some integer, and "flip them all over together".

Hints for solving:

[3] A small, thin instrument used in kitchens for instance to flip burgers or, here, pancakes.

(Continued on page 16.)

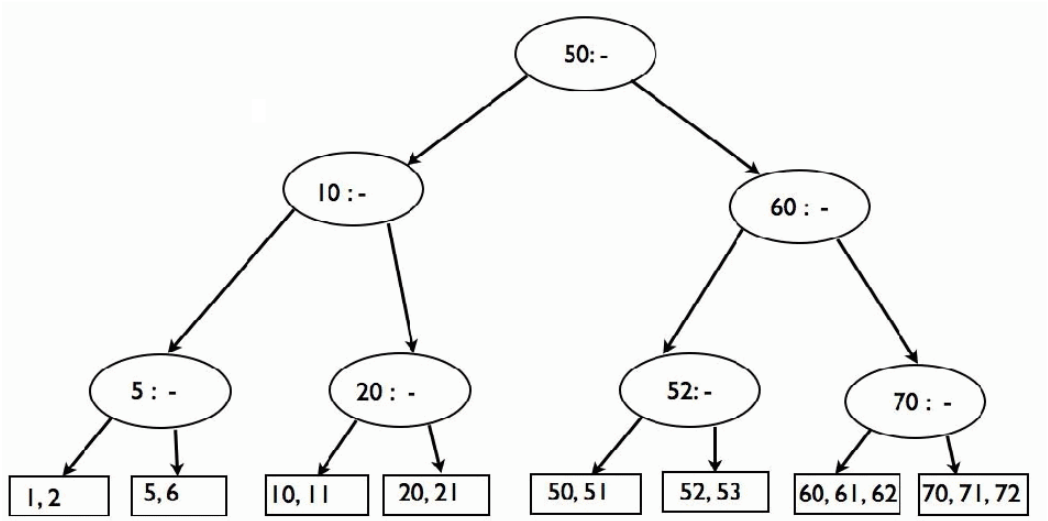


Figure 6: B-tree

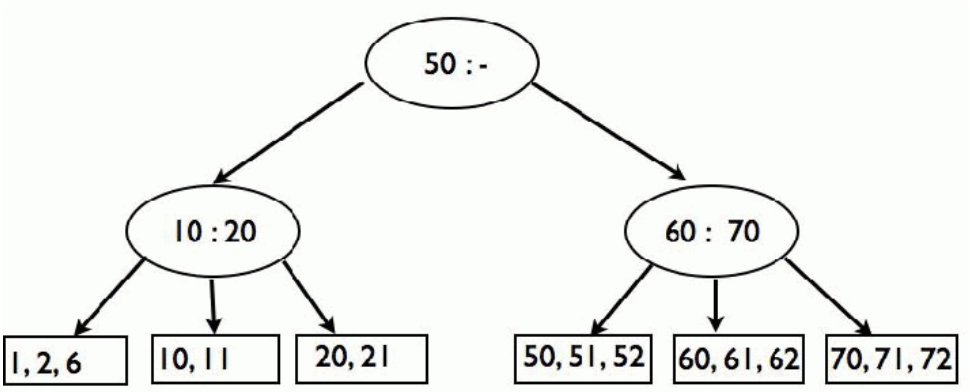


Figure 7: B-tree

TA: I would try to keep the top of the stack reverse sorted at all times when increasing the number of elements we look at. If the new element is not smaller than the previously bottom one, we flip the whole stack. Now find the largest K where this element is the largest element (start with K = L / 2 where L is the size of the stack we are trying to sort right now (there may be stuff below this)). Then move K halfway in one or the other direction until you find the right K. Then flip this K sized stack, then flip K-1 pancakes, then flip all the elements in the sorted stack and increase the amount of elements we look at. Not sure if there is a better solution, but I imagine keeping track of whether the stack is sorted in descending or ascending order at any flip may allow some optimization.

7a Pancake sorting (weight 12%)

Implement "pancake sorting". Assume that the data to sort is stored in an array `p`, where index `0` refers to the bottom of the "pile of pancakes" and the top of the

(Continued on page 17.)

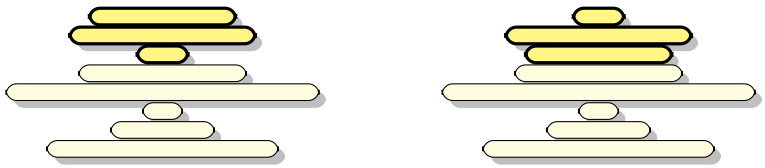


Figure 8: Flipping the top 3 pancakes (from left to right)

pile corresponds to the highest index.
Assume you have 3 methods or procedures at your disposal:

`flip(k)`, `max(i,j)`, and `min(i,j)`.

The first operation is equivalent to the operation described above, the second one allows to determine the position of a maximum value in the array ("the largest pancake") in the elements from `p[i]`, ...`p[j-1]`. The `min(i,j)` for minimal elements. Note that you have no other operations than `flip`, `max`, and `min` to access the pan-cake array (note: you are not *obliged* to use all operations). In particular, you cannot directly use `p[i]` for reading and writing to the array.

Solution: *There seems only one sensible solution (based on* `max` *and* flip*. I would go for a loop (not recursion, but works as well). The loop goes through the array* `p`*, starting from the* bottom *(the other way does not work). Invariant: the lowest* k *pannekaker are sorted. Initially, there are* 0 *pancakes sorted. each iteration:* $k \to k+1$*; since the descripion put the "bottom of the pile" at the lower index, the loop will indeed* count up *if (as it's natural) the* k *corresponds (+/- 1) to the index in the array, depending on the exact choice of the loop. So each iteration will have to put* the (or a) largest pancake *in the unsorted part on top of the already sorted one. That requires: finding the largest (that's done with* `max`*. Moving the thusly found max-cake to the bottom of the unsorted part requires* 2 flips*: one to flip it up to the overall top, and from there down to the appropriate bottom.*

Hints for correcting: *If the idea is correct and the code otherwise ok, however, the "borders" are "one off" that gives 4 points off. Note that we are giving exactly that* `max(i,j)` *finds the position of from* i *to* $j-1$*, we expect that people must just that exactly not just "roughly." If the people misunderstand max or min — it gives the* position *not the value— then the task is not solvable. If they then cheat and somehow access the array in a forbidden way to solve it anyway, then that cannot give more than 3 points.*

7b Complexity (weight 4%)

Assume that the mentioned operations `flip`, `max`, and `min` are of *constant time* complexity i.e., $\mathcal{O}(1)$ (for instance due to special hardware like a spatula and the human eye estimating pancake sizes). What's the worst-case complexity of your solution.

Solution: *Linear.*

Hints for correcting: *If the program a different solution (can't imagine any, though) from the official one and analysze that correctly, it's fine.*

(Continued on page 18.)

7c Complexity (weight 4%)

Make now, unlike in problem **7b** the (reasonable) assumption that the operations `flip`, `max`, and `min` are *linear-time* in their input; for `flip`, linear in the number of pancakes to be flipped, for max/min, linear in the difference between the two input parameters. What's the complexity of your pancake sorting now?

Solution: *quadratic*

Hints for correcting: *Same remark as the previous one.*