UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in	INF2220 — Algorithms and data structures
Day of examination:	13. December 2011
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, binary-tree) there is no need to explain how they work or behave. In general: when using abstract data types from the library, you can use them without explaining what they do.
- The **weight** of a problem indicates how difficult it is estimated to be. You may take that into account as you organize your time.

Good luck!
Ragnhild Kobro Runde and Ingrid Chieh Yu

(Continued on page 2.)

Hints for solving: *The is the English version of the exam 2011. The extra parts called Solution is a very short description of the solution.*
It is* not *meant in all cases that this is the solution that gives the full points! *It is NOT! For instance in the tasks where you are supposed to show the individual steps, the solution that gives you full points involves giving those steps. Here, we show only enough information, to help you check your results when you do it yours (which is actually what we hope that you do, not just enjoying to read solutions from last years, but solving the tasks yourself).* ***Therefore, print the version without solution first.***

(Continued on page 3.)

Problem 1 Priority Queue (weight 10%)

A *max heap* is a binary heap with the heap order property that children are always *smaller or equal* to their parents.

Hints for solving: *Read carefully here, we are talking about a* max *heap. Don't overlook that.*

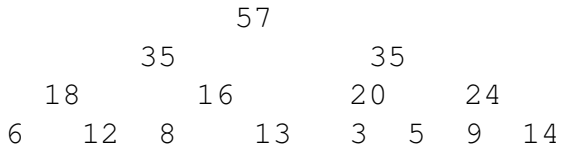
Given the values $13, 24, 5, 6, 18, 57, 9, 12, 35, 8, 16, 3, 20, 35$, and 14

1a Insertion (weight 5%)

Show the array representation you get by inserting the values one at a time into an initially empty max heap.

Solution: *57, 35, 35, 18, 16, 20, 24, 6, 12, 8, 13, 3, 5, 9, 14*

Hints for solving: *The best (fastest) way to solve that is: to do the* tree *first, even if only on a separate sheet of paper, and then copy it into the array. Note that there are 15 numbers, so it fits exactly into the heap array (which has 16 slots). Assuming that the calculation uses the* $n * 2$ *and* $n * 2 + 1$ *and* $n/2$ *calculation, as shown in the lecture, directly on the position in the array (which definitely makes sense looking at speed), then we have to start at position 1 anyway, as we have shown in the lecture.*
As for the solution: As stated at the beginning of the task, we are dealing with a (max) heap (which put's a condition on the tree structure), but besides that, since we are asking here about the array implementation of a binary heap, there is the additional condition that the tree is "totally" balanced and filled from left to right. As a result of the two condition, there is only one reasonable way how to build up the heap, given the sequence *of numbers.*
Btw. because of the strict conditions on the full binary heap trees, the above sequences corresponds to (actually it "is") the following tree:



Note also that one should be able to solve that question without even looking at the code *of the implementation (hopefully even without looking up what a binary heap actually does), because, as mentioned, there is logically only one meaningful way the insertion can be done (independent of how concretely it's coded). To solve this exercise by looking up the code takes too much time.*

(Continued on page 4.)

1b Linear time construction of heap (weight 5%)

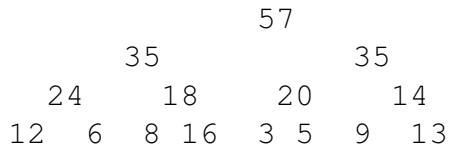
Let B be the array representation of a complete binary tree containing the values given above. Show the array representation of the max heap you get by using the linear time *buildHeap* algorithm on B.

Solution: *57, 35, 35, 24, 18, 20, 14, 12, 6, 8, 16, 3, 5, 9, 13*

Hints for solving: *The* `buildHeap` *is described in [?, page 255–257]. There they also explain the fact that the thing is* linear time. *The core of the algorithm is the* `percolateDown` *(which has been described earlier; the insert-algorithm of the previous task was using the* `percolate-up` *procedure.) So the trick here is* not *to use the* repeated *insert, as this would violate the requirement on the run-time mentioned in the task.*
Percolate-down is used in the procedure `deleteMin`. *The argument of the procedure is not an element, but a node, where the percolate should start. As an invariant, it takes a node which has two (potentially empty) subtrees. Those satisfy the heap condition, so the only potential violatation is of the designated (root) node and it's children.*
Of course one does not need to start with the last node in the arrays, "half" of them (the bottom row) satisfies the heap condition: each node is a tree with just one single node, and this satisfies trivially the heap condition (and nothing can percolate down anyway). If one remembers that the build-heap is to take a heap / array a input and (that's the important part) is built on percolating down*, then again one does not need to study the code. And furthermore: having understood the principles, the outcome of the procedure and the steps are (almost) determined. The root may, of course, be in violation with both children. However, if there is a violation at all, then it's determined whether the root should percolate down the left subtree or the right. In the case of a max-heap, the maximum of the root and the both immediate children must be on top.*
There is only the possibility, that the some of the three nodes under comparison are equal*. In one case (the root is equal to one (or both) children, we could make logically a percolate-down function that also in this case percolates down, but most probably we won't do that (in absense of further conditions) and as far as the outcome of this test is concerned, where only the keys matter, there won't be no difference whatsoever. More critical is the situation when the root is different from the children but the children are equal. In this case, logically, one must replace the root with* one *of the children, but it's not determined logically, by which it should be, that depends on the actual code.*
Concretely in the code, as given in [?, page 254], the line which decides that is line 30*, which compares the keys of the children. The data structure is a* min-heap*, not a max heap. So the procedure compares the two children. But it prioritizes* left *child: only if the right child is really smaller than the left, it (has to) percolate down there.*
In the current task, drawing the tree for the input shows that this situation (equal siblings) may occur. In the original tree the situation is not present, but by running the algo, of course, the tree is transformed, so if the equal numbers occur as siblings, the non-determinism arises. In the algo, the two equal number are 35,

(Continued on page 5.)

and in the end, they will be siblings as the algorithms runs) but it does not result in non-determinism.
Anyway, doing the procedure gives the following end-result, written as tree more clearly.



When doing the algorithm, one should be aware of course, that percolate down is a recursive procedure, so reparing on place and swapping father and child may lead to a further violation etc.

Problem 2 Complexity (weight 10%)

What is the *worst-case* complexity of the following implementations:

2a For loops (weight 3%)

```
for i = 1 to n {
  for j = i to n {
    for k = i to j {
       print (i, j, k);
    }
  }
}
```

Solution: *Solution:* $\mathcal{O}(N^3)$

Hints for solving: *This one should go without blinking. Three loops, thus cubic. The only "complication" is that one of the loops does not take n into account, but j. However, in the way complexity is defined: that does not matter, it's just a constant factor. What is also very easy is: this is not about worst-case or best-case estimation, since the running time is* fixed *based on the input. Thus, best = worst = average case. So maybe the question about worst-case complexity may be a bit misleading. The fact that this gives 3 points only indicates that we think it's easy, and we expect people to be fast on this one.*
Also, without saying, we assume that the complexity of printing is constant. Which it probably not is: if one assumes a standard logarithmic (decimal) representation of numbers, longer numbers of course take longer time to print. So one could argue there's a log factor here as well.

(Continued on page 6.)

2b Recursion (weight 7%)

```
float foo(A) {
  n = A.length;
  if (n==1) {
    return A[0];
  }
  let A1,A1,A3,A4 be arrays of size n/2
  for (i=0; i <= (n/2)-1; i++) {
    for (j=0; j <= (n/2)-1; j++) {
      A1[i] = A[i];
      A2[i] = A[i+j];
      A3[i] = A[n/2+j];
      A4[i] = A[j];
    }
  }
  b1 = foo(A1);
  b2 = foo(A2);
  b3 = foo(A3);
  b4 = foo(A4);
}
```

Solution: $\mathcal{O}(N^2\ logN + N^2) = \mathcal{O}(N^2\ log\ N)$

Hints for solving: *This one is of course more complex, as we have loops and recursive calls. Both recursion and loops may (and do) contribute to the time complexity. Recursion is the more complicated matter. The situation of the function is not* too complicated *either. It's rather straightforward in that the (recursive) procedure contains a loop (or rather 2 loops, but that's not the point), but it's* not the other way around. *So it's not that inside the loops there are recursive calls. So that makes it rather straightforward, actually.*
The the recursision is "outside" *and the loop(s)* "inside" *the recursion, the question is broken down (i.e., analyzed) basically into two questions:*

1. *How often do I call the method?*
2. *What the time complexity of one method body in itself?*

For the second question: The complexity of the method body is "dominated" by the two loops, to be precise, by the two nested *loops*[1] *Now, there are two complications wrt. the loops (or one depending on how you count complications). Ultimately, in tasks like that one is supposed to give the complexity in the size of the* input *in this case the size or length of the input array.*[2] *Let's call that* N. *Now the length of the loops is not from* 0 *to* N, *it's more complex.*

[1] Two loops one after the other gives a different complexity that nested loops, right. The difference is that between $+$ and $\times$ ...

[2] Things like that "complexity in terms of the length of the array" are often in examples like that not made explicit. It's dicated by common sense and so that experience with this class of exercises gives a reliable feeling what the relevant parameter of the input is. If one makes a unreasonable kind of set-up, where one uses "hyper-astronomically large" numbers stored in the array, one could make the argument that the time complexity would be dominated by for instance by *copying* a number from one array slot to the other.

(Continued on page 7.)

1. *i and j don't go up to N but there is a factor of 2 (and there is one -1.*
2. *the inner loop goes up not to a fix upper bound, but the upper bound depends on the outer loop index i, so that may vary.*
3. *finally, upper bound of the outer loop depends on the argument of the function, which means, considering the recursion as a form of "loop" as well, the upper bound of i depends in the even more outer "recursion loop".*

Problem 3 Binary Search Trees (weight 15%)

Given the following class for the nodes in a binary search tree:

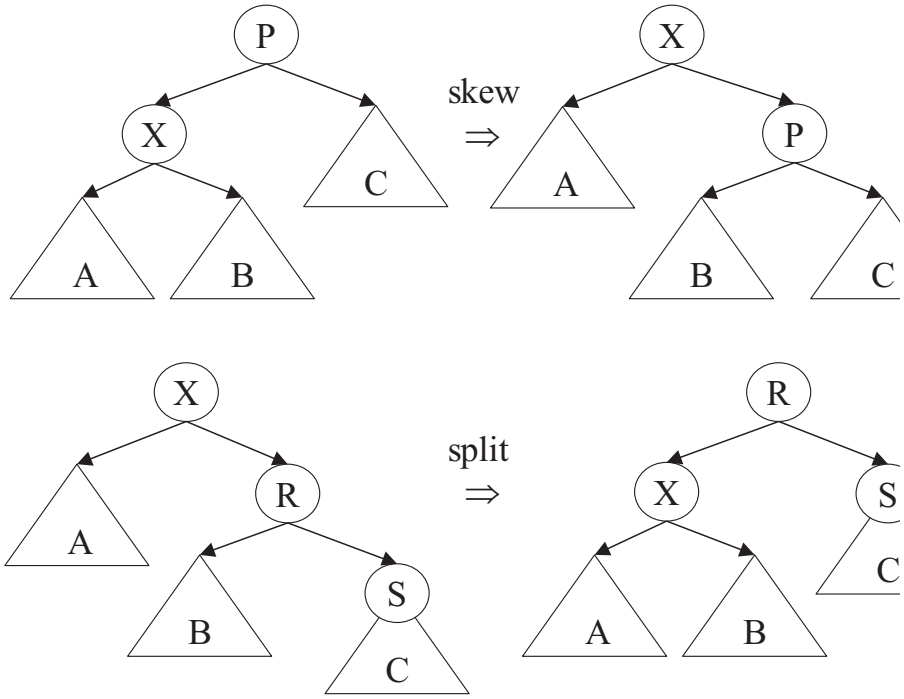
```
class BinNode {
    BinNode left, right;
    int element;
}
```

3a Balancing trees (weight 7.5%)

Two popular methods to re-balance a binary search tree are the `skew` and `split` operations as shown in the following figure:

If one assume that, then one could analysis also the complexity in terms of growing the size of the data stored in the array. But that's boring, unrealisting, contracting common sense. The analysis would show that the compexity of the algorithm is *linear* in the size of the stored in the cells: longer numbers probably take linear time to copy. But it therefore does not say saything about *this* algorithm, since the algorithm does not even mention the size of the data slots. Actually, it does not even mention the nature of things stored in the array (whether it's integers, strings or more complex things. The problem of the size of the stored data will (especially in this example) not influence greatly the run-time. And even if it would, taking that into account would say somthing about the complexity of things `A1[i] = A[i]`, so basically, when analyzing the code, one makes assumptions about the underlying operation, like assigmnents, which are not made explicit. In almost all cases, the complexity of those underlying operations are simply *assumed* to be constant. Technically, they are often not, but assuming that is still ok, because the complexity of the algorithm is *dominated* by other factors. The other ("pragmatic") reason why one assumes typically that the complexity of those operations are constant (which basically means to ignore them in the analysis) is that, as mentioned, their implementation and their code etc. is not given. Therefore, one has either the choice to ignore what is not given (which is what we do), or to make the analysis of, here, for instance `foo` *dependent* on an assumed complexity of the underlying operations. In the simple cases of assingments, that makes not sense, but of course if one wants to for instance evaluate *heap-sort* which is a form of iteration over `insert/buildheap` operations, it's unwise to ignore the auxiliary functions, especially since they *depend* on the length of the array (which is the main parameter of interest), unlike here, where the underlying assignment-operations depend on the size of the stored data only.

(Continued on page 8.)



1. Explain why `skew` and `split` maintain the search tree properties.

2. Implement `skew` and `split` without using auxiliary methods. Both methods shall take the root of the subtree as parameter (in the example, P for `skew`), rotate as shown and return the new root (in the example, X for `skew`). You can assume that the nodes corresponding to X and P, R and S, on the figure exist (i.e., that they are not null).

Solution: *First it is good to state what the binary search tree opererations are. One thing to observe is that both operations are actually very similar, in the sence that* skew *is the inverse of* `split`*, except that what's* P *in the first one is* R *in the second and the the root of tree* C *is not explicityly given as* S *in the first operation. Therefore, if one has the argument for one operation, one should have the argument in the reverse direction too. Actually, the operation* skew *actually just one example of* single rotation *as described on [?, page 146] (only that it's not called skew, so people will not*

(Continued on page 9.)

directly find it in the book, but should be familiar anyhow). Anyway, the conditions for the binary search tree are

1. *it's a binary, rooteed tree*
2. *each node in the tree is $\geq$ than its left child (if exists) and $\leq$ th right child.*[3]

A non-property *of a binary search tree is* balance*. Of course, trees may be balanced (and rotation helps to maintain balance in a binary search tree, for instance in AVL trees or for red-black trees). Even if binary trees sometimes are balanced, adding that as a property of a binary tree especially in the context of this question is an error (and gives less points!) Balance is a wrong condition, because it is* not *maintained by* `skew` *and* `split`*. Already because of that fact, one would get points deduced. Of course, skew, rotation, etc has to do with balanced trees (especially, with balanced bst), but rotations does not* maintain *balance (i.e., it's not an invariant), it turns an* unbalanced *tree into a balanced one.*[4]
1. (Svaret er skrevet som om det ikke finnes like elementer. Det er trivielt å endre svaret i forhold til hvilken løsning som velges for like elementer.)
Pekerne som endres av skew er P.left og X.right. P.left settes til å være subtreet B, som allerede er en del av det venstre subtreet til P og dermed kun inneholder elementer mindre enn P. X.right settes til å være P, og siden X er venstre-barn til P må P være større enn X i utgangspunktet.
Pekerne som endres av split er X.right og R.left. X.right settes til å være subtreet B, som allerede er en del av det høyre subtreet til X. R.left settes til å være X, og siden R er høyre-barn til X må X være mindre enn R i utgangspunktet.
2.

```
BinNode skew(BinNode p) {
    BinNode x = p.left;
    p.left = x.right;
    x.right = p;
    return x;
}

BinNode split(BinNode x) {
    BinNode r = x.right;
    x.right = r.left;
    r.left = x;
    return r;
}
```

3b ABBA trees (weight 7.5%)

An ABBA tree is a red-black tree where only the right child can be red. Each node in the tree also contains information about its level, such that:

a. All leaf nodes have level = 1.

b. All black nodes' level is one less than its parent.

[3] Alternatively the other way around, does not actually matter.

[4] Of course, it's also not that it takes a completenly unbalanced tree and balances it. Depending on the kind of balance condition, certain operations like deleting *destrouys* the balance in a BST "a bit" and it's this "bit" that rotating operations can repair.

(Continued on page 10.)

c. All red nodes' level is equal to its parent.

Extend class `BinNode` with the necessary attributes and write a method that checks whether a red-black tree is a valid ABBA tree according to the given requirements. You can assume the tree is a valid red-black tree. Feel free to use auxiliary methods.

Solution: *Ekstra-attributter: int color (med konstantene RED og BLACK) og int level.*

```
// Kall: ABBAtre(rot);
boolean ABBAtre(BinNode b) {
    if (b == null) return true;
    return ABBAtre(b.left, b.level, true)
           && ABBAtre(b.right, b.level, false);
}

boolean ABBAtre(BinNode b, int forelder, boolean isLeft) {
    if (b == null) return true;
    if (b.left == null && b.right == null) return (b.level == 1);
    if (b.bolor = RED) {
        if (b.isLeft) return false;
        if (b.level != forelder) return false;
    } else {
      if (b.level != forelder-1) return false;
    }
    return ABBAtre(b.left, b.level, true) &&
           ABBAtre(b.right, b.level, false);
}
```

Hints for solving: *One should read the text carefully, it's about to extend the binary tree with attributes, not the red-black tree. Therefore, the color is an attribute and the level, as decribed. Also wrt. the task, it's actually not really a task about RB-trees. The procedure to implement is* not *checking the color condition, it is checking the given conditions. The red-black tree implements in a smart way a balance condition but that one is not checked. When thinking about the solution, one realizes first that it's about trees; trees (red-black, whatever) are, before everything else, recursively (or inductively) given data structures. Therefore, the* natural *way (and typical way in exercises/exams) to solve a problem is that it's some form of recursive method.*[5]
Anyway, let's first analyze the conditions on ABBA trees. There are two groups,

1. *one about color only*
2. *one about the levels (but talking about color as well).*

[5]There is no guaratnee that every problem ever in a tree is a recursive method, or that the recursive one is the natural solution but actually there's a (very) good chance. Therefore, when presented a problem over trees, one should actively look for a recursive solution, probably it's the right direction (but again, no guarantee). Note, however, that exams are not made to "trick" people in that they can come up with an obscure solution. they are constructed to find out whether people can apply (variation of) standard techchies to (variations of) standard data structures. So we hope to see that stanrdard problems are solved in a standard manner (after a bit of thinking).

(Continued on page 11.)

Another way of seeing it is: there is a "horizontal" *condition (saying left childs must be black) and* "vertical" *condition (about the levels). The exercises does not specify that but one might choose to make two methods, each of which checks only one condition.*
Well, before programming, one needs to understand. *As indicated, we expect it to be recursive, therefore we need a recursive understanding of what it means to be horizontally-ok. So a "recursive understanding" of the problem, in general is:*

> *I satisfy the horizontal condition, if I (as a node) "locally I satisfy P" and my 2 children (which are trees) are horizontally ok, too*

It's important to realize that this understanding can be applied to our problem (in tree problems it will (almost) alway be the case, that's why one has trees). Anyway, the question then is:

> *How to formulate P, capturing that, as far as the current node is concerned, the horizontal condition is ok.*

There are basically two ways to interpret that

1. *I am locally ok, if I have no parent or I* am *a* right *child or I* am *a* black left *child.*
2. *I am locally ok, if I do no* have *a black left child*

The rest is recursion. The first formulation could lead to the following pseudo-code.

```
is-horizontal-ok(node) =
    if b = null then true
    else
        is-horizontal-ok(this.left)  /\
        is-horisontal-ok(this.right) /\
        (this.parent == nil \/              // no parent, fine
         this.parent.right == this          // right child: fine
         (this.parent.left  == this /\      // superfluous actually
           this.color == black))            // left child = black!
```

The other formulation leads to the following code.

```
is-horizontal-ok(b) =
  if b =  null then true
  else is-horisontal-ok(b.left)  /\
       is-horisontal-ok(b.right) /\
       (if b.left ≠ null
        then (b.left.color == black))
```

Actually, another solution (which corresponds logically to the first one) realizes that the condition P is different *depending on whether the node* is *a left child or not. The "I am a child" interpretation is, as mentioned the first choice for P. Since there are two versions, one could write two different versions of* `is-horizontal`*. Perhaps more elegant is that the two versions are written as one method but with a boolean argument.*

(Continued on page 12.)

```
is-horizontal-ok(b, isleft) =
    if b = null then true
    else
        is-horizontal-ok(b.left ,true)    /\
        is-horisontal-ok(b.right ,false) /\
          (isleft  → color == black)        // left child = black!
```

Of course, the code can be slightly "optimized" at least it looks slightly optimized. For instance, it looks as if the code evaluates all parts of the conjuction. But actually, that's not true in general. If one part is false, the evaluation stops. Of course that is not so obvious, and perhaps one cannot rely on that. However, in this case, in particular, since the thing is function, without side-effects, it does not matter much. Anyway, they official solution makes more use of cascaded conditional, here I am more "logical".
The other vertical problem can be done in the same way. Again we have to think of P "when am I ok". We choose now directly the approach that corresponds to the first one above, i.e., when I compare "myself" with the one state of my parents. Since we are proceeding (as before) top-down, that has the advantage of "looking back" (instead of looking forward to the values of my childen). This allows to hand-over the relevant property as argument to the procedure or method.

```
is-vertically-ok(b, level) =
    if    b == null then true                                  // as before
    else if == leaf then (b.level == 1)                        // checking fields
    else if is-vertically-ok(b.left ,b.level)    /\
            is-vertically-ok(b.right ,b.level) /\
            (b.level == level-1)                              // left child = black!
```

Problem 4 Binary Tree (weight 10%)

A complete binary tree with N elements is stored in an array, in position 1 to N. How big must the array be for

1. a binary tree with two extra levels?
2. a binary tree with N nodes where the deepest node is at depth $2\ log\ N$?
3. a binary tree with the deepest node at depth $4.1\ log\ N$?

Solution:

1. *Når vi legge til et nivå, dobler vi antall node-plasser. 2 ekstra nivåer krever 4 ganger antall plasser.*
2. *Et komplett binærtre har dypeste noden på $log\ N$, $2^{log\ N} \approx N$. Arrayen blir N^2 stor siden $N * N \approx (2^{log\ N})(2^{log\ N}) = 2^{2\ log\ N}$. (Den eksakte relasjonen er $2^{log\ (N+1)} = N + 1$)*
3. *omtrent $2^{4.1\ log\ N} = N^{4.1}$*

(Continued on page 13.)

Hints for solving: *In all cases we are always dealing with complete binary trees.*

1. *For the first question, it should be clear that the size must depend on N.*
2. *I find the formulation of the task not too clear. In particular, the N in the genral description and the N here is confusion, the formulation is not clear, also the mentioning of "a binary tree with N nodes" is really confusing. So the real forumlation should be, in my eyes: Assume an complete BT where the deepest node is $2 \log n$, how many nodes has the tree (= how big is the array).*

Problem 5 Graphs (weight 13%)

5a Longest path in a graph (weight 5%)

Write a method `longestPath` such that, given a weighted directed acyclic graph $G = (V, E)$, it returns the weight of the longest path in the graph in time $\mathcal{O}(|V| + |E|)$. (The graph may contain edges with negative weights.) Explain why this algorithm satisfies the time requirement.

What must the algorithm do in order to return the path itself?

You can write the method in pseudo code, but it must be detailed. It must be clear what the initialization consists of and what the main loops are doing.

Solution:

```
int longestPath (){

    length_to = array[|V|]
    for <each v in  V> {
        length_to[v] = if v.inngrad = 0 then 0 else −∞
    }

    for <each vertex v in topSort() > {
        for <each edge (v, w) in E> {
            if length_to[w] <= length_to[v] + weight(v,w) then
                length_to[w] = length_to[v] + weight(v,w)
        }
     }
    return max(length_to[v] for v in V)
}
```

$\mathcal{O}(|V| + |E| + |V| + |E| + |V|) = \mathcal{O}(|V| + |E|)$
Løsningen består av å bruke topologisk sortering og relaxation.
To get the path one needs to introduce a predecessor (or successor)-array which is updated whenever the length_to is modified

Hints for solving: *One has, I guess, to read the question carefully. The reason is that perhaps the "most" important condition gets overlooked. Of course all*

(Continued on page 14.)

conditions are somehow equally important, so ignoring that it's a graph kills any smart solution as well. What can put people on the wrong track is that sounds like a shortest path *algo. Of course, it's the longest path, but shortest path and longest paths seem close enough. Two things, however, should trigger some thinking here.*

1. *The graph is assumed to be acyclic*
2. *the required complexity. Taking for instance Dijkstra, if one remembers, that was quadratic in the number of nodes!*

The last one should warn you that better not take Dijkstra or similar. Even if Dijstra might be adapted for maximum, one still needs to argue for the required complexity. The other warning sign should be the acyclicity. Dijkstra etc, the shortest path problem in general as far as we have treated it in the lecture worked for arbitrary *graphs, including loops. Sometimes even negative-weights (but that increased the complexity a lot). Anyway, loops, especially negative weights, seem to increase the toughness of the problem. Since here we give an* acyclic *graph, one should take that into account.*[6]
So we need to take into account the acyclicity. Let's first ignore potentially complications (like negative weights) and even make the thing unweighted. Remember that this was also the route how we came to the from the bread-first shortest path to the relaxation-based version (Disjkstra) which works for weighted ones (which had as important data-structure had a p-queue, to make relaxition do the proper thing). Anyway, as a start observation, a bad idea certainly is to calculate all paths and then find the longest. Another observation or side remark is as follows. Let's call end-nodes *those which have no pre-decessors or no successors. It's not guaranteed that the longest path connects endpoints, so one should not concentrate on that. Basically we need to calculate the "longest-path" for all nodes. The question is then: what does it mean, the longest path "of a node". We can considere the problem "forward" or "backward". I.e., the longest path* starting *from a node or the longest path* ending *in a node. Let's take the second one: the path* to *a node. To get a good solution, one has to find a way to* reuse *information one has somehow already calculated. And that's where the acyclicity comes in. The key to any good solution is to* use *already calculated results to get later ones. Now that we have decided that we want the longest path* to *a node, let's see where to start? Of course, then we start with the a node without incoming edges. There the longest path is* 0. *Now, if we have calculated all pre-decessors of a node, then the longest path to that one is determined as well. A different formulation is:*

> *Assume that I have two nodes* $v \rightarrow w$ *and the maximal path to* v *is known, then the maximal path to* w *is* at least *the maximal path to* v *plus the weight of* $v \rightarrow w$.

Now, that's perhaps half of the core of the idea. The other half is: assuming that one has the maximal path already definitely *determined somehow and one has taken this information into account for the estimation of* w,

[6] Again, exams are not meant to trick people; it's unlikely that we write "assume an acyclic graph" just as a blindspor or trick or "noise", which actually does not help in any solution, but just to "add a little confusion" ...

(Continued on page 15.)

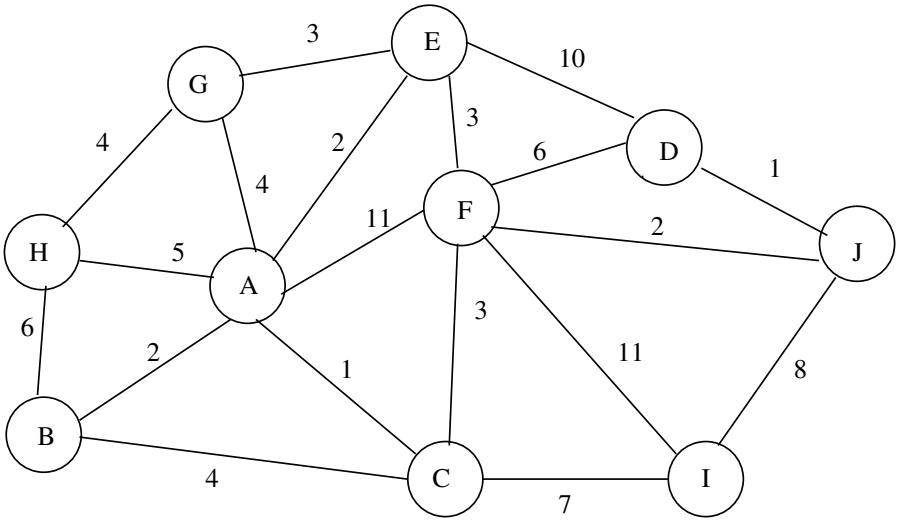
then we can consider v as treated

If of course later the information of v would change (by becoming larger), then also w might be adapted again *(and that smells like a not-efficient solition). But that's where acyclicity comes in: if one has treatead all the ones "smaller" of v then the value of v itself is fixed and therefore it can be indeed considered as done.*
Now considering nodes "earlier" in a DAG before the ones later can be solved by topological sorting. And that's the solution:

topological sorting and relaxation.

5b Kruskal and Prim (weight 5%)

Given the following graph:



1. Find a minimum spanning tree for the graph using *Kruskal*'s and *prime*'s algorithms. Show clearly the result after each step of the algorithm by listing the edges in the order they are selected.
2. Is this tree unique?
3. When can we guarantee that the tree is unique?

Solution: *Solution: There are in general (and here?) more than one solultion. It's actually not* the *MST but a MST. But the question is not so much whether MST is minimal, but whether the algo is unique. One has to be careful here. One could argue that "of course" the answer is unqique because the algo, when presented with different choices, it always make a concrete one. So one* implementation *always gives a unique solution. But that is not meant here.*

1. *Kruskal: (A,C) (D,J) (F,J) (A, E) (A, B) (E, F) (G, E) (H, G) (C,I)*

   Prim: (A, C) (A, E) (A, B) (G, E) (E, F) (F, J) (D, J) (G, H) (C, I)
2. *nei. (A, C) (D, J) (A, E) (F, J) (A, B) (F, C) (G, E) (H, G) (C,I)*
3. *If all the edges have unique weights. Of course that is only a sufficient condition, not a necessary one.*

(Continued on page 16.)

5c Spanning tree (weight 3%)

Given a connected undirected graph G where the edges have unique weights and let S be a minimal spanning tree of G. Is it the case that the edge with the second smallest weight $e \in G$ must be an edge in S? Explain.

Solution: *The answer is yes.*
hvis e ikke er i S er det enten fordi:
1. e ligger parallellt med den minste kanten (multikant) eller

2. e er en loop.

Begge tilfeller er utenfor vår def av grafer.

Hints for solving: *This gives 3 points. We expect that this is fast (depending on of course) the definitions of MST etc need not to be looked up. Note that it's not a question about specific algos and about programming. That makes the answer potentially fast.*
Anyway, a good way of approaching the algo is probably to draw some examples. Of course, one could also make use of the MST algos. We had two, Prim or Kruskal, and since we had both algos in the previous task, one can take then in principle easy for an argument (since we know they do the MST-job). When choosing between them, the one better suited for making an argument here is Kruskal *(growing the forest), because it takes the edges in the order of their appearance. So it's clear that Kruskal takes the edge we are interested in in the second step. With Prim we don't know when it is taken, which makes the argument harder. So for Kruskal, in the second step we take the second smallest edge, but one edge is already there. Now we don't add the edge, if it's not a "forest" by adding that. This is only when it's a loop (in an undirected graph) or multi-edges. So the expected answer is "yes". However, no is also ok* if *one states that this happens only if there is a loop. We want to see that one understands what's going on, less some details which are, to some extent, a matter of choice.*

Problem 6 Treasure Hunt (weight 15%)

The university of Uqbar is organizing a treasure hunt (rebus run) for their new students. There are in total n teams, and all teams must visit the same m stops, where $n \leq m$. The university wants all teams to start simultaneously, and maximum one team at a time on each stop. Any two teams shall also not have the same order on any two stops (if team 1 shall go directly from stop 1 to stop 2, no other teams can do the same at any point in the game).

Assume that all teams use the same amount of time on each stop (and between stops). Write a method (with auxiliary methods if necessary) that takes n and m as input and generates and prints a complete set-up for such a treasure hunt, if possible. If it is not possible to meet all the requirements, the method should return an error message.

(Continued on page 17.)

For example, for $n = m = 4$, a possible result of the method can be:

```
Team 1:  Stop 1 - Stop 2 - Stop 3 - Stop 4
Team 2:  Stop 2 - Stop 4 - Stop 1 - Stop 3
Team 3:  Stop 3 - Stop 1 - Stop 4 - Stop 2
Team 4:  Stop 4 - Stop 3 - Stop 2 - Stop 1
```

```
class RebusLop {
    int N, M;
    int[][] oppsett;

    // Eventuelle andre variable

    RebusLop(int antallLag, int antallPoster) {
        N = antallLag;
        M = antallPoster;
        oppsett = new int[N][M];
        // Eventuell initialisering av andre variable
    }

    void lagOppsett(...) {
        // TODO
    }

    void skrivUt() {
        // Innhold ikke tatt med, skriver ut det ferdige oppsettet.
    }
}
```

Solution: *Ekstra variable:*

```
    // bruktRad[i][j] angir om post j er brukt på rad i eller ikke.
    boolean[][] bruktRad = new boolean[N][M];

    // bruktKolonne[i][j] angir om post j er brukt i kolonne i eller ikke.
    boolean[][] bruktKolonne = new boolean[M][N];

    // etter[i][j] angir at det finnes et lag som går rett fra post i
    // til post j
    boolean[][] etter = new boolean[N][M];
```

```
void lagOppsett(int lag, int post) {
    for (int i = 0; i < antPost; i++) {
        if (!bruktRad[lag][i]) {
            if (!bruktKolonne[post][i]) {
                if (post == 0 || !etter[oppsett[lag][post-1]][i]) {
                    bruktRad[lag][i] = true;
                    bruktKolonne[post][i] = true;
                    if (post != 0) etter[oppsett[lag][post-1]][i] = true;
                    oppsett[lag][post] = i;
                    if (post < antPost-1) {
                        lagOppsett(lag, post+1);
                    } else if (lag < antLag-1) {
```

(Continued on page 18.)

```
                        lagOppsett(lag+1,0);
                    } else {
                        skrivUt();
                    }
                    bruktRad[lag][i] = false;
                    bruktKolonne[post][i] = false;
                    if (post != 0) etter[oppsett[lag][post-1]][i] = false;
                }
            }
        }
    }
}
```

Hints for solving: *på rad: one after the other.*

Problem 7 Text Algorithms (weight 12%)

7a Boyer Moore (weight 4%)

Compute the *good suffix shift* for the *pattern*: `a b c a b c a c a b`

Solution:

index	***Mismatch***	***Shift***	`goodCharShift`
0	`b`	*1*	*goodCharShift[0] == 1*
1	`ab`	*10*	*goodCharShift[0] == 10*
2	`cab`	*8*	*goodCharShift[0] == 8*
3	`acab`	*5*	*goodCharShift[0] == 5*
4	`cacab`	*8*	*goodCharShift[0] == 8*
5	`bcacab`	*8*	*goodCharShift[0] == 8*
6	`abcacab`	*8*	*goodCharShift[0] == 8*
7	`cabcacab`	*8*	*goodCharShift[0] == 8*
8	`bcabcacab`	*8*	*goodCharShift[0] == 8*
9	`abcabcacab`	*8*	*goodCharShift[0] == 8*

7b Boyer Moore (weight 4%)

A mismatch is found during the search for the pattern `a b c a b c a c a b`, as given below.

```
   Text: a a c a b c a a b c a b c a b c a c a b
Pattern: a b c a b c a c a b
```

Where in the text will the *Boyer Moore* algorithm look for the next match? (i.e., how far will Boyer Moore shift the pattern after this mismatch?)

Solution:

```
   Text | ... a a c a b c a a b c a b c a b c a c a b ...
Pattern |         a b c a b c a c a b
```

skift 2 med bad character shift.

7c Huffman coding (weight 4%)

Create a Huffman code for the string `nonconcurrence`. Show the decoding table.

Solution: *Dekodingstreet er ikke unik.*
c = 01
e = 101
n = 11
o = 100
r = 001
u = 000

Problem 8 The Dutch Flag (weight 15%)

The game "The Dutch flag" comprises a number of red, white and blue chips in a row. The goal of the game is to swap pieces so that all the red chips are on the left, all the blue chips on the right, and all the white chips in the middle. The only legal operations are to examine a chip's color and to switch two chips.

Assume that the initial position is represented in a character array where the only possible array values are 'R', 'H', and 'B' (there is no empty slots in the array). Write a method that sorts the array according to the above criteria with as few permutations as possible.

Hint: Think of the array as consisting of four (dynamically sized) parts — sorted red chips, sorted white chips, unsorted, and sorted blue chips. Initially, all pieces will belong to the unsorted section and the other three sections will be empty.

(Continued on page 20.)