UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in	INF2220 — Algorithms and data structures
Day of examination:	15. desember 2010
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.
- Some questions contain sentences like "Assume a collection (or set . . .) of this-and-that . . . ". The word "collection" or "set" in this context *does not* mean an implementation of the Java `Collection` or `Set` interface, it is just an English word. When referring to concrete Java classes/interfaces, we use capital letters, i.e., `Collection`.

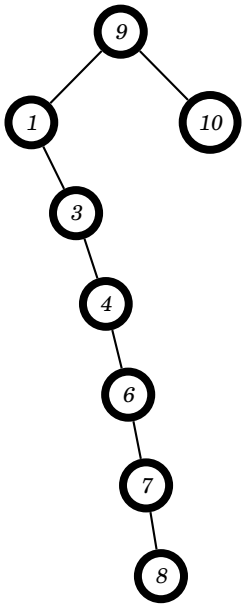
(Continued on page 2.)

Problem 1 Binary Search Tree (weight 15%)

1a Insert Nodes (weight 5%)

Start with an empty binary search tree and insert these values (in **this** order) 9, 1, 3, 10, 4, 6, 7 and 8. Draw the resulting tree.

Solution:



Hints for solving: *That should go very fast. Note that all keys are different. There is only one solution. The question does not ask to show the series of trees in the individual steps, so one does not have to show that.*

1b Logarithmic time (weight 5%)

Is it possible to locate elements in the binary tree in logarithmic (log_2) time? Justify your answer.

Solution: *No it's not possible since the tree is out of balance, it contains 8 elements,* $log_2(8) = 3$*, the tree is higher than that.*

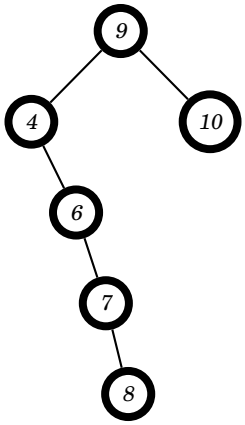
Hints for solving: *The question is kind of tricky, actually. Questions like logarthimic time make sense, in my eyes, only asymptocically. This question refers (not very clear also) to one given and quite unbalanced tree. Since the tree is fixed, the question strictly speaking makes no sense. Especially no one has ever said that one level in a tree is "time one". That makes no sense since in particular we discussed time complexity up-to a constant.*
The proposed solution (and I guess many people "got the answer") is: logarithmic time complexity is for balanced trees (or for the average). This one is pretty unbalanced, so the answer is no. *But the question is not really good.*

(Continued on page 3.)

1c Remove nodes (weight 5%)

Remove the values 1 and 3 from your binary tree. Draw the resulting tree.

Solution:



Hints for solving: ***Note*** *As long as the binary search tree, still is a binary search tree after the nodes have been removed, this assignment should be ok. Different removal schemes may lead to different trees (? is that true).*
The way that we have learned deletion is that there are 3 different situations. Leaves, one child nodes, two child nodes. The cases here are not the hardest, we remove only nodes with one child. In the lecture we said, we single child then just "jumps over" the one deleted. That's in my eyes the only sensible (but perhaps not logically imaginable) way of dealing with it, so there is actually only one possible result.
One (not too smart) alternative could be to use the procedure for 2-children nodes also in cases here. However, that does not lead to a different result: it works like this: one replaces the deleted node with the node with the smallest *key in the left subtree. In this particular tree, where the subtree is just a list, that gives the same result here. One also sees a weeknees if one uses the more complex 2-children-deletion process in the simpler case, because there might be the situation, that there* is no *right subtree, so one has to have a special case distinction, which again treats the one-child case in a specially manner.*
However, if *the sub-tree would not be linear, using the 2-children-approach for just one child would give a different result. However, as said, not in this particular task, so the result, as far as I see it, is unique.*

Problem 2 Sorting (weight 20%)

Look at the method `sorted` given in Figure 1 and determine the following.

2a Classification (weight 5%)

Is this a **value** or **comparison** based sorting algorithm? Justify your answer.

Solution: *It is value based, no comparison of the elements are ever done, we just allocate buckets for all possible values and update how many elements that fall into each bucket.*

(Continued on page 4.)

Hints for solving: *That should again be simple, if one remembers the sorting stuff. It's a typical* value *based algorithm. It's easy to see in that the elements are* not compared. *Of course there are some "comparisons" in the algorithm, but that's indices in loops. Even if the understanding of the algorithm may take some time, the general feeling that this is not a comparison-based algothim should be fast.*

2b Time Complexity (weight 5%)

What is the worst case time complexity of this method, given as $O(n, m)$ where:

1. n = `input.length`
2. m = max value of `input` array

Solution:

2c Time Complexity (weight 5%)

The first loop: $O(n, m) = n$

```
for( int i = 0; i < input.length; i++ ){
    if(input[i] > max){
        max = input[i];
    }
}
```

The second loop: $O(n, m) = n$

```
for( int i = 0; i < input.length; i++ ){
    b[ input[i] ]++;
}
```

The third loop: $O(n, m) = m + n$

```
for( int i = 0; i < b.length; i++ ){
    while( b[i] > 0 ){
        sort[counter] = i;
        b[i]--;
        counter++;
    }
}
```

NOTE *The inner* `while` *loop can only be executed* n *times, since we can fill at most* n *buckets.*
Total time complexity:
$O(n, m) = n + n + m + n = 3n + m$ *(linear time complexity:* n*).*

As bigO, they could state $O(n, m) = O(n + m)$ *with constants and so on removed, and with* $n, m \to \infty$ *this leads to* $O(2n) \to O(n)$*.*

(Continued on page 5.)

Hints for solving: *Again, in principle one should be able to answer that question without really understanding what the algo does, just by applying "complexity thinking". A first observation is: the algorithm is* not *recursive. That typically makes it easier. The complexity comes thus from the* loops *only. We have to check therefore* all *the loops, and in particular find out how they depend on the specified input (as given by the question), how they potentially depend on each other, in particular whether they are nested or sequentially composed. In this case we have 4 loops, 2 are nested. We also keep an eye on the "worst-case" part of the question. Let's go through the 3 sequentially composed loops in the body of the* `sorted` *function.*

1. $\mathcal{O}(n)$. *That's obvious, the upper bound is directly given as one of the parameters. For that loop, there is no difference between worst/best and thus average case. Note in particular that it does not matter wether the internal condition is true or false: it does not make a difference, complexity-wise. Of course, if the conditional is always false ("best case") or to be expected false 50% of the time, the loop will be faster, but it's only a constant factor, so it does not matter.*

2. *same for the second loop, here it's even more obvious that the worst/best case are identical.*

3. *here now we have a* nested *loop, and the exit condition(s) depend not as obvious as before on the "input" parameters. The other loop is the length of the array which is the given input parameter* $m \Rightarrow \quad \mathcal{O}(m)$.

   Now the inner loop is more tricky, and one has to now understand the algorithm, in particular, one has to understand what the second loop has done. What is unusual in this task is the following: typically, when encountering nested *loops, there will be a* multiplication *of their complexity. If one would do that here, that would lead to a too imprecise approximation. The point is that the different nested while-loops* depend on each other. *Here's the naive approach, considering them independent. The inner while loop is determined by the value of* `b[i]`. *Now that is determined by the 2nd loop, and that's where one has to understand the algorithm (or at least it helps). Without deep understanding we see in the second loop that the array* b *(the "buckets") are increased exactly* n*-times. Depending on the input, of course, different buckets are increased. In the worst case, it's only one particular bucket which is increased (so everything "goes into the same bucket"). So the maximal value there is* n. *If one stops thinking now, that leads to the following worst-case estimation*

$$\mathcal{O}(n \times m)$$

   What has been ignored is that even if it's correct that the highest number a bucket may contain is n *and that this is the worst-case complexity of the inner loop, they different inner loops are not independent since the different buckets are not independent. The second loop increase the* whole *bucket array* n*-times, typically spreading it over different individual buckets or slots. Therefore, the inner while-loops should not be considered independent*

(Continued on page 6.)

but are altogether *executed n times, and again that gives $\mathcal{O}(n)$. Counting now* only *how many times the inner commands (inside the inner loop) are executed would be to simple, however. It ignores the input parameter m, and the task explicitly request to take that into account. If the maximal value is very large, but only very few numbers, then the 3rd one still executes m times, and even if many of the inner while-loops are "empty" each iteration still costs some (constant) time. So that gives $\mathcal{O}(n+m)$*

2d Average case (weight 5%)

Does the average case time complexity differ from the worst case complexity? Justify your answer.

Solution: *The average case time complexity is identical to the worst case complexity, no matter what values our input array holds, the exact same steps are preformed.*

2e Weakness (weight 5%)

That it can't handle negative values is obviously a weakness, can you name another weaknesses with this approach?

Solution: *The major weakness with this type of bucket sort is that large values requires you to allocate a very large array, i.e., the bucket array. If you want to sort an array of 2 elements, and one of the elements have a value of 1000000, you need to allocate an* `int` *array with 1000000 elements.*

Problem 3 Huffman Coding (weight 10%)

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

- `a`: 9
- `b`: 2
- `c`: 4
- `d`: 2
- `e`: 1
- `f`: 1

3a Huffman tree (weight 10%)

Draw a Huffman tree based on the frequency table.

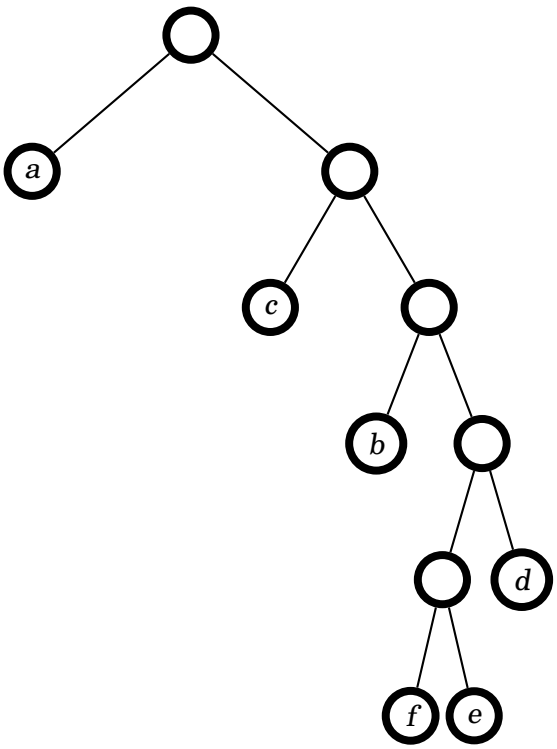
(Continued on page 7.)

```
// assume positive input values

int[] sorted(int[] input){

    int[] sort = new int[input.length];
    int max = 0;

    for( int i = 0; i < input.length; i++ ){
        if(input[i] > max){
            max = input[i];
        }
    }

    int[] b = new int[max + 1]; // all values are 0

    for( int i = 0; i < input.length; i++ ){
        b[ input[i] ]++;
    }

    int counter = 0;

    for( int i = 0; i < b.length; i++ ){
        while( b[i] > 0 ){
            sort[counter] = i;
            b[i]--;
            counter++;
        }
    }

    return sort;
}
```

Figure 1: `sorted`

Solution:

(Continued on page 8.)



***NOTE** There are more than one legal Huffman tree, but **a** and **c** should get the shortest paths, and **e** and **f** should get the longest paths.*

Hints for solving:

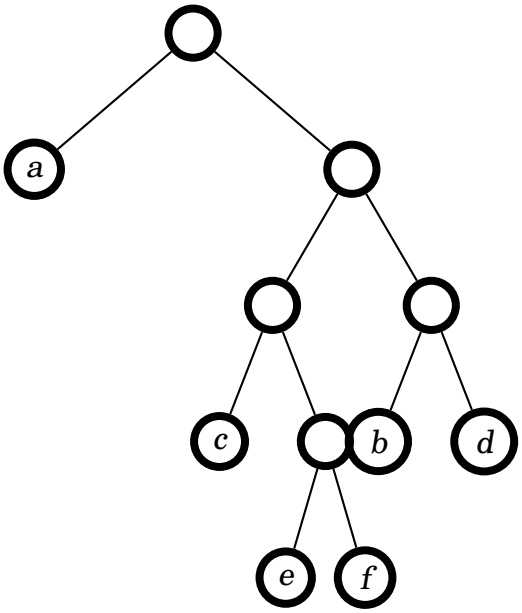
That is a task which again should go without much thinking, if one remembered the Huffman stuff. Note also the (obvious) fact that unlike for heaps and also search trees, one does not have a choice where to put the "larger values". For heap, there are max heaps and min-heaps, for BST I might have the choice for sorting increasingly or decreasingling. For Huffman encoding, assuming that one wants a minimal size of the file, one has no such choice. One has *to start with the smallest ones and proceeds in a greedy manner (one could use a p-queue for that). If one has forgotten whether the greedy algo should always take the minimal ones or the maximals? That should be easy as well. First, remember that* higher *frequencies means that the letter is more often, and this it should be higher up the in the tree and the ones with the lowest frequencies further down. However, due to the prefix condition, of course all letters are at the leaves, i.e., further down means further away from the root. Since the tree builds up by taking too trees/nodes and joining them together, intuitively the later a node is added, the more probably it's close to the root and the ones added first will be further way. Therefore, we must start with the* minimal *ones, and greedy takes always the 2 smallest.*

The task does not seem to state one should make intermediate stages. Anyway, in our example it's

```
e f:   2
[e f] d : 4
b [[e f] d] : 6
c [b [[e f] d]] : 10
a  [c [b [[e f] d]]]: 19
```

(Continued on page 9.)

In the stages, there are points where one can choose to continue different ways. Note also: if one does it on paper, one is like "to grow a tree", at least in this example one can do that. That's however, not how the algorithm works and there are other examples where that does not work. What is slightly misleading when doing it like this is to assume that the "growing tree" part is always one of the two minimal elements to choose from. That seems to be the case in the example, so that could work here.
One could do also a different choices. For instance the following



Note: the code of `d` *(and* `d`*) in these two concoding is 4 bit long or 5 bit long, in other words the trees are differently balanced! Nonetheless, they are both optimal as far as the given text is concerned.*

Problem 4 Parsed Expression (weight 25%)

Assume that a parser for binary calculations generates a tree, where each leaf-node corresponds to a number, all other nodes corresponds to a binary operator ($+, -, \times, /$). In Figure 2 you can see two examples. Each node holds a token (`String`), which is either the string representation of a number, or the string representation of a binary operator. The expressions are represented by the class `BinOp`, shown in Figure 3.

4a String representation (weight 5%)

Binary operators cannot be applied in any order, so we must make sure the order of application is clear from our `String` representation. I.e., the expressions **a** from Figure 2 should not be represented as "$4 \times 4 + 7$" since it's unclear if we mean: $(4 \times 4) + 7 = 23$ or $4 \times (4 + 7) = 44$. Implement the `str` method inside `Node`, and group operations together with parenthesis, to create an unambiguous `String` representation of the expression represented by a `BinOp` object. Example from Figure 2 `a.toString()`: "$((4 \times 4) + 7)$"

Solution: *Pretty simple. It's just printing parentheses + recursion.*

(Continued on page 10.)

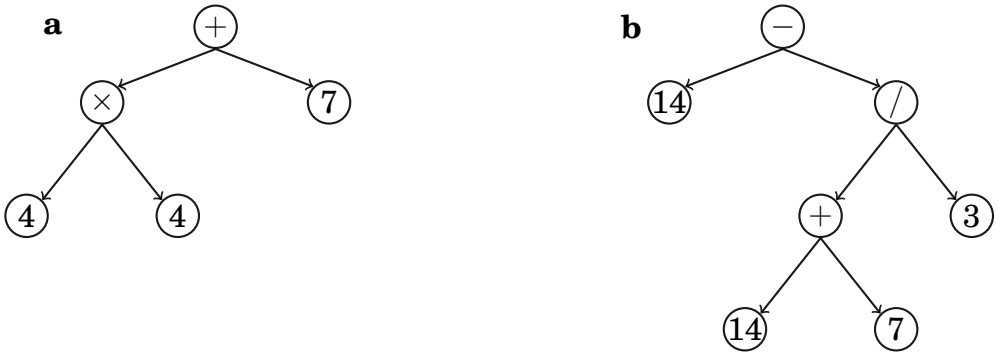


Figure 2: Parsed Expression

```
public class BinOp{

    Node root;

    public String toString(){ return root.str(); }
    public double eval(){  return root.eval();   }
    public String polish(){ return root.polish(); }

    class Node{

        String token;
        Node left, right;

        String str(){ /* TODO */ }
        double eval(){ /* TODO */ }
        String polish(){ /* TODO */ }

        boolean isLeaf(){ // utility function
            return left == null && right == null;
        }
    }
}
```

Figure 3: BinOp

```
String str(){
    if( isLeaf() ){
        return token;
    }else{
        return String.format("(%s %s %s)",
                    left.str(), token, right.str());
    }
}
```

(Continued on page 11.)

4b Evaluate (weight 5%)

Implement the `eval` method inside the `Node` class which returns the value of the expression as a **double**.
Note: The elements inside the tree which represent numbers can be converted with the function `Double.parseDouble(String s)`.

Solution:

```
double eval(){
    if( isLeaf() ){
        return Double.parseDouble(token);
    }else{
        if(token.equals("+")){
            return left.eval() + right.eval();
        }else if(token.equals("-")){
            return left.eval() - right.eval();
        }else if(token.equals("*")){
            return left.eval() * right.eval();
        }else{ //  token.equals("/"))
            return left.eval() / right.eval();
        }
    }
}
```

Hints for solving: *Also this one is equivlantly complex. It's again just a "homomorphism", calculating by "structural induction" over the expression, which means in terms of the tree, but recursion and bottom-up. The only additional "complexity" is the case-switch.*

4c Polish Notation (weight 5%)

To use parentheses to group binary operations is not desirable in all circumstances, so we want to be able to represent our expressions in a unambiguous way without parentheses. Polish notation[1] achieves this, by giving the operator prior to its operands/numbers.

- **a**: $+ \times 4\,4\,7$
- **b**: $- 14 / + 14\,7\,3$

Implement the `polish` method inside the `Node` class, which returns the polish notation of the expression.

Solution:

```
String polish(){
    if(isLeaf()){
        return token;
```

[1] Invented by Polish logician Jan Lukasiewicz

(Continued on page 12.)

```
    }else{
        return String.format("%s %s %s",token,left.polish(),right.polish());
    }
}
```

Hints for solving: *Note much more difficult (except perhaps for understanding the task). It's actually the same as for the infix notation, excect that the paretheses are not needed (well, not wanted actually as specified by the task) and that the order of the printout is changed, of course.*

4d Polish Calculator (weight 10%)

Below you see a calculation of the expression **b** from Figure 2 in polish notation.

- $-\ 14\ /\ +\ 14\ 7\ 3$
- $-\ 14\ /\ (+\ 14\ 7)\ 3$
- $-\ 14\ /\ 21\ 3$
- $-\ 14\ (/\ 21\ 3)$
- $-\ 14\ 7$
- $(-\ 14\ 7)$
- 7

From left to right apply an operator as soon as you find two operands/numbers. Assume that a `String` expression given in polish notation, can be tokenized by a call to the `String.split` method, such that:

```
String[] tokens = polishExpression.split(" ");
```

gives you an array of tokens which are either `String` represented numbers, or operators. Assume you have the following utility methods.

```
boolean isOp(String token){
    /* implementation left out */
}

String strApply(String op, String v1, String v2){
    /* implementation left out */
}

/* ------------------ Example usage ------------------ */

isOp("+") == true
isOp("*") == true
isOp("5") == false
```

(Continued on page 13.)

```
strApply("+", "40", "1").equals("41") == true
strApply("*", "35", "2").equals("70") == true
strApply("/", "5", "2").equals("2.5") == true

/* ——————————— Example usage ——————————— */
```

Choose an appropriate data structure and implement the method with the signature below, which takes the `String` representation of an expression in polish notation, and computes the result.

```
double polishEval( String polishExpression ){
    /* TODO */
}
```

Solution:

4e Polish Calculator (weight 10%)

```
public double polishEval(String expression){

    String[] tokens = expression.split(" ");

    // made a little stack implementation here
    // but java.util.Stack<String> should do :-)

    StrStack estack = new StrStack(tokens.length+1);
    StrStack unused = new StrStack(tokens.length+1);

    for(int i = tokens.length - 1; i >= 0; i--){
        unused.push(tokens[i]);
    }

    String v1, v2, op;

    while( unused.hasMore() ){

        estack.push(unused.pop());

        while( estack.size() > 2 ){
            v1 = estack.pop();
            v2 = estack.pop();
            if( notOp(v1) && notOp(v2) ){
                op = estack.pop();
                estack.push( strOp( op, v1, v2 ) );
            }else{
                estack.push( v2 );
                estack.push( v1 );
                break;
            }
        }
    }
```

(Continued on page 14.)

```
    return Double.parseDouble( estack.pop() );
}
```

Problem 5 Miscellancious (weight 15%)

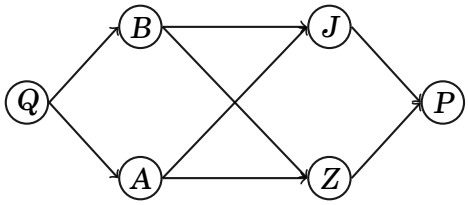
5a Topological sort (weight 7.5%)

Assume that you have a dependency graph with the following nodes: {Q, B, J, P, A, Z}. There are **only** 4 legal topological orderings of the graph, which are listed below.

- Q, A, B, J, Z, P
- Q, B, A, J, Z, P
- Q, A, B, Z, J, P
- Q, B, A, Z, J, P

Draw a graph that meet the criteria.

Solution:



5b Boyer Moore (weight 7.5%)

Calculate the **good-suffix-shift** for the needle: **skjeskj**

Solution:

```
goodshift[0] :        !j    1
goodshift[1] :       !kj    7
goodshift[2] :      !skj    7
goodshift[3] :     !eskj    4
goodshift[4] :    !jeskj    4
goodshift[5] :   !kjeskj    4
goodshift[6] :  !skjeskj    4
```

(Continued on page 15.)

Problem 6 Common Subset (weight 15%)

Assume that you have two arrays `char[] a1` and `char[] a2` both containing 1-byte[2] characters where `a1.length == N` and `a2.length == M`.

6a Implementation (weight 10%)

Implement a method with the signature found in Figure 4 that returns an array of characters that are present in both `a1` and `a2`. The characters in the returned array can come in arbitrary order.

```
// returns characters which are contained in both a1 and a2

char[] both(char[] a1, char[] a2);
```

Figure 4: Signature of `both` function

Solution:

```
    static char[] both(char[] a1, char[] a2){

        final int CHAR_MAX = 256;
        int count = 0;
        int[] cache = new int[CHAR_MAX]; // assume 1-byte initialized to 0

        // loop 1
        for(int i = 0; i < a1.length; i++){
            if( cache[(int) a1[i] ] == 0 ){ cache[(int) a1[i] ]++; }
        }

        // loop 2
        for(int i = 0; i < a2.length; i++){
            if( cache[(int) a2[i] ] > 0 ){
                count++;
                cache[(int) a2[i] ] = -1;
            }
        }

        int tmp = 0;
        char[] result = new char[count];

        // loop 3
        for(int i = 0; i < CHAR_MAX; i++){
            if( cache[i] == -1 ){
                result[tmp++] = (char) i;
            }
        }

        return result;
    }
```

[2] 1-byte characters means that their **int** values are in the range 0...255

(Continued on page 16.)

Hints for solving: *One just has to think about it a bit. The challenge is the* complexity *and not to do the immediate thing, a* nested *loop. The solution as shown has of course 3 loops but they are not nested. The naiv approach would make 2 loops, each one through the two arrays: Starting (e.g.) with* `a1`*: for all elements in* `a1`*, go (= loop) through all elements in* `a2`*; if you find a "match", store it into the result array (or set) (and exit the loop for optimization). This double-loop can be optimized in that one remembers if one has seen in the first array a letter already, then one does not need to check it again in the second one.*
The overall complexity would be of course $n_1 \times n_2$*, optimized or not.*
The algo here is smarter. If one looks at the previous one, certain things seem to be done more *than once, and that's always a source of optimization. Now, what is done to often? The outermost loop not: certainly we* need *to go through the first array fully (actually through both arrays, of course).*[3]
Now the problem is the inner loop: for all entries in the first array we go through the inner array over and over again. The trick must be to avoid that; we want to go through the second array only once, as well. Instead of 2 nested loops we want 2 separate one. That's the difference between multiplication *and* addition.
How can we achieve that? One way of thinking is: if we don't want to nest the loops, we have to remember *relevant information when doing the first loop so that it's available when going to the second one, because the second one is not completely afterwards the first. Now the question is: what to remember. It may be clear already by now, but let's think about it systematically. If the loops are nested, the index of the outer loop, say* i*, represents (the occurency of a) letter in the first array, namely* $a[i]$*. With the inner-loop solution, we go through the second array, and check whether we see* $a[i]$ *again, and if so remember it for the final result. There we don't need to* remember $a[i]$ *of the first array, as soon as the index* i *goes to* $i++$*, we have treated that one and we can forget it. Now, as said, we cannot forget it, the question is, what do we have to remember? Also that is clear. It's unimportant that* position i*, the letter* $a[i]$ *was found, what matters is the* fact *that letter* $a[i]$ *was found? How do we remember that?*
One way of doing it is: to make an array for all letters of the alphabet. That's done in the shown solution. The disadvantage is: the alphabet of available characters must be known statically in advance. Anyway, the algorithm put's a flag (integer 1)in the first loop if it sees a letter. The second loop goes through the second array and checks (using the `cache`*) whether the corresponding letter had occured in the first array, if so they set it to a third value (-1). One could of course also work with booleans, a bit more logical, but anyway. The second loop also keeps track on the* number *of common matches (in the counter* `count`*) which is needed to allocated the* `result` *array which is filled by the third loop.*

6b Complexity (weight 5)

What is the worst case time complexity of your implementation. Justify your answer.

[3] The only situation which would allow *not* to look at the whole array is: *all* letters of the (previously known) alphabet have already been treated. In that case one may exit the loop. That would, however, not affect the average/worst case complexity.

(Continued on page 17.)

Solution:

- *loop 1: N*
- *loop 2: M*
- *loop 3: 256 (constant)*
- $O(N, M) = N + M + 256$
- $O(N, M) = N + M$ *(Ignoring constants)*
- *As* $N, M \to \infty$ *we get* $O(N, M) \to O(2N) \to O(N)$
- *Linear time complexity*

Good luck!

(Continued on page 18.)

Method Interfaces

Here is a list of methods that you may find useful during the exam, they are all taken from the standard library of Java.

```
java.util.Set<E>

* boolean add(E e)                  // add element e
* boolean remove(Object o)          // remove Object o
* boolean contains(Object o)        // is Object o an element?
* int size()                        // number of elements


java.util.List<E>

* boolean add(E e)                  // add element e to list
* void add(E e, int index)          // add element e to index
* boolean remove(Object o)          // remove Object o
* E remove(int index)               // remove element at index
* E get(int index)                  // retrieve element at index
* int indexOf(Object o)             // get index of Object o
* boolean contains(Object o)        // is Object o contained in list
* int size()                        // number of elements


java.util.Map<K,V>

* boolean containsKey(Object o)    // is Object o a key?
* boolean containsValue(Object o) // is Object o a value?
* V put(K k, V v)                  // add key-value pair
* V get(Object key)                // fetch value based on key
* V remove(Object key)             // remove key-value pair
* int size()                       // number of key-value pairs
* Set<K> keySet()                  // set of keys in Map


java.util.PriorityQueue<E>

* boolean add(E e)                  // add element to que
* E peek()                          // fetch element with highest priority
* E poll()                          // same as peek + delete element from que
* boolean contains(Object o)        // is Object o an element in the que
* int size()                        // number of elements
```