KØBENHAVNS
UNIVERSITET

Introduktion til diskret matematik og algoritmer: Problem Set 2

Due: Wednesday February 25 at 12:59 CET.
Submission: Please submit your solutions via *Absalon* as a PDF file. State your name and e-mail address close to the top of the first page. Solutions should be written in LaTeX or some other math-aware typesetting system with reasonable margins on all sides (at least 2.5 cm). Please try to be precise and to the point in your solutions and refrain from vague statements. Never, ever just state the answer, but always make sure to explain your reasoning. *Write so that a fellow student of yours can read, understand, and verify your solutions.* In addition to what is stated below, the general rules for problem sets stated on *Absalon* always apply.
Collaboration: Discussions of ideas in groups of two to three people are allowed—and indeed, encouraged—but you should always write up your solutions completely on your own, from start to finish, and you should understand all aspects of them fully. It is not allowed to compose draft solutions together and then continue editing individually, or to share any text, formulas, or pseudocode. Also, no such material may be downloaded from or generated via the internet to be used in draft or final solutions. Submitted solutions will be checked for plagiarism.
Grading: A score of 120 points is guaranteed to be enough to pass this problem set.
Questions: Please do not hesitate to ask the instructor or TAs if any problem statement is unclear, but please make sure to send private messages—sometimes specific enough questions could give away the solution to your fellow students, and we want all of you to benefit from working on, and learning from, the problems. Good luck!

1 (60 p) Provide formal proofs of the following claims using proof techniques that we have learned during the course.

1a (30 p) Define $a_1 = 1$ and $a_n = 2a_{n-1} + 1$ for $n \geq 2$. Prove that for all positive integers $n \in \mathbb{N}^+$ it holds that $a_n = 2^n - 1$.

Solution: We prove this equality by induction over n.

Base case $(n = 1)$***:*** We have $a_1 = 1 = 2^1 - 1$ by definition.

Induction step: Assume that the equality holds for $n - 1$ and consider n. We have

$$\begin{aligned} a_n &= 2a_{n-1} + 1 \\ &= 2 \cdot (2^{n-1} - 1) + 1 && \text{[by the induction hypothesis]} \\ &= 2^n - 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

which is the desired equality.

The claim now follows by the induction principle.

1b (30 p) Prove that for all non-negative integers $n \in \mathbb{N}$ it holds that $3 \mid 4^n + 5$.

Solution: We prove this by induction over n.

Base case ***(**$n = 0$**)**:* For $n = 0$ we have that $4^0 + 5 = 6$ is divisible by 3.

Induction step: Suppose that $3 \mid 4^n + 5$, which is the same as saying that $4^n + 5 = 3 \cdot M$ for some integer M. Fixing this M, and working on the expression for $n + 1$, we get

$$\begin{aligned}
4^{n+1} + 5 &= 3 \cdot 4^n + 4^n + 5 && [\text{since } 4^{n+1} = (3+1) \cdot 4^n] \\
&= 3 \cdot 4^n + 3 \cdot M && [\text{by the induction hypothesis}] \\
&= 3 \cdot (4^n + M)
\end{aligned}$$

which shows that this expression is divisible by 3.

The claim follows by the induction principle.

2 (60 p) For each of the propositional logic formulas below, determine whether it is a tautology or not. If the formula is not a tautology, show how to add a single connective to make it into a tautology. Please make sure to justify your answers (e.g., by presenting truth tables, or by using rules for rewriting logic formulas that we have learned in class).

2a (30 p) $\neg\big((p \to q) \lor r\big) \to \big((\neg q \land \neg r) \land p\big)$

Solution: This formula is a tautology, i.e., it is always true. To see this, note first that we know that

$$p \to q \;\equiv\; \neg p \lor q \tag{1}$$

since the only way the implication $p \to q$ can be false is that p is true and q is false, and this is one of the basic equivalences we have learned in the course. Furthermore, from De Morgan's laws it is easy to derive that

$$\neg(a \lor b \lor c) \;\equiv\; \neg a \land \neg b \land \neg c \tag{2}$$

(i.e., the only way a disjunction can be false is that all its disjuncts are false).

By combining (1) and (2), and using that conjunction is commutative, we see that the premise $\neg\big((p \to q) \lor r\big)$ is in fact equivalent to the conclusion $(\neg q \land \neg r) \land p$, and so the implication in Problem 2a will always evaluate to true.

2b (30 p) $\big((p \land q) \to r\big) \leftrightarrow \big((q \lor r) \lor \neg p\big)$

Solution: This is not a tautology. If we set p and q to true but r to false, we get that $(p \land q) \to r$ evaluates to false but $(q \lor r) \lor \neg p$ evaluates to true, and so the whole formula is false.

If we change q on the right-hand side to $\neg q$, we obtain the formula

$$\big((p \land q) \to r\big) \leftrightarrow \big((\neg q \lor r) \lor \neg p\big) \;, \tag{3}$$

which is a tautology. To see this, we can rewrite the left-hand side of (3) as

$$\begin{aligned}
(p \land q) \to r \;&\equiv\; \neg(p \land q) \lor r && \text{(4a)} \\
&\equiv\; (\neg p \lor \neg q) \lor r && \text{(4b)} \\
&\equiv\; (\neg q \lor r) \lor \neg p && \text{(4c)}
\end{aligned}$$

by using the property (1) of implication, De Morgan's laws, and commutativity of disjunction. This show that the left-hand and right-hand sides of of the formula (3) are equivalent, and so the formula will always evaluate to true.

3 (80 p) One slightly annoying feature of the *merge sort* algorithm is that it ignores long runs of elements that are already sorted. Jakob has therefore devised an algorithm that will make use of already sorted runs, and your task in this problem is to help him analyse this algorithm.

The *merge* part of the algorithm would be essentially the same as before:

```
merge (L, R)
    m := length (L)
    n := length (R)
    M := new array of length m + n
    i := 1
    j := 1
    for k := 1 upto m + n {
        if (i <= m) {
            if (j <= n and R[j] <= L[i]) {
                M[k] := R[j]
                j    := j + 1
            }
            else {
                M[k] := L[i]
                i    := i + 1
            }
        else {
            M[k] := L[j]
            j    := j + 1
        }
    }
    return M
```

For the main recursive sorting method, however, the array should be split only when we detect the first element that is not in sorted order, which Jakob is trying to achieve as follows:

```
mergerunssort (A)
    n := length (A)
    i := 1
    while (i < n and A[i] <= A[i + 1] {
        i := i + 1
    }
    if (i < n) {
        L := new array of length i
        R := new array of length n - i
        for j := 1 upto i {
            L[j] := A[j]
        }
        for j := 1 upto n - i {
            R[j] := A[i + j]
        }
        R := mergerunssort (R)
        A := merge (L, R)
    }
    return A
```

3a (30 p) Is the pseudocode algorithm above correct in that for any input array `A` it will be the case that `mergerunssort(A)` returns the same array but sorted in increasing order?

Solution: Let us first argue about the correctness of the *merge* method that merges two sorted arrays into a larger sorted array. This is essentially the same algorithm that was covered in class, except we have removed the (perhaps confusing) infinitely large elements at the end of the arrays, but let us nevertheless argue correctness from first principles. (To be clear, though, if it is explained convincingly in a solution why this algorithm has already been covered in class, then this is sufficient).

The assumption for *merge* is that the input arrays L and R are already sorted. The main for loop fills in the elements from L and R in M in sorted order. We have the following invariants at the top of the for loop:

1. The elements $L[1], L[2], \ldots, L[i-1]$ and $R[1], R[2], \ldots, R[j-1]$ have been copied to the output array positions $M[1], M[2], \ldots, M[k-1]$ sorted in increasing order.

2. The elements $L[i]$ and $R[j]$ are greater than or equal to all elements inserted in M in previous iterations.

3. The element $L[i]$ is less than or equal to all elements $L[i+1], L[i+2], \ldots, L[m]$ and $R[j]$ is less then or equal to all elements $R[j+1], R[j+2], \ldots, R[n]$.

Invariants 1 and 2 are vacuously true the first time we enter the for loop (since nothing has been copied to M). Invariant 3 remains true throughout the algorithm since L and R are assumed to be sorted, so we do not need to argue about how to maintain it. If we can show that the invariant 1 and 2 are maintained after every iteration of the loop, then this implies that right after the final iteration of the foor loop (where we can mentally think of k as being $k = m+n+1$) the array M contains all elements in L and R correctly sorted.

A special case to handle is that when we get to a point where the check $i \leq m$ fails (or $j \leq n$ fails), then we know that we have emptied the whole array L (or R, respectively). By invariants 2 and 3, this means that it is correct to just copy the remaining elements from the other array. This is exactly what will happen after the first time the check $i \leq m$ (or $j \leq n$) fails inside the for loop, since the same check will keep failing in all future iterations. We can also note that it can never happen that both checks fail simultaneously, since the for loop only runs for $m+n$ steps and the arrays L and R together contain that many elements, so they cannot both be emptied before the for loop has run all iterations.

If both $i \leq m$ or $j \leq n$ hold, so that we get to line 10 in the algorithm, then the smallest element of $L[i]$ and $R[j]$ is inserted next in M in position k. By invariants 2 and 3 this must be the kth smallest element. Suppose first that this smallest element is $L[i]$. Since $L[i] \leq R[j]$ and both arrays L and R are sorted in increasing order (invariant 3), this means that the smallest remaining element was copied to M but that this element was also greater than or equal to all previously copied elements (invariant 2). After i has been incremented to $i+1$, this means that invariants 1 and 2 have been restored. The other case, when the smallest element in $R[j]$, is symmetric.

Let us now discuss the main *merge runs sort* method. We will show by induction over the length of the array that this is a correct sorting algorithm. Arrays of length 1 are covered by the more general base case of arrays of any positive length n that are already sorted in increasing order. For such an input array A, the while loop will scan through A until $i = n$ (since for all $i < n$ it holds that $A[i] \leq A[i+1]$), after which it will return the input array A unchanged. This is clearly correct.

For the induction step, we adopt as our inductive hypothesis that *merge runs sort* sorts arrays of length less than n correctly. Let A be an array of length n that is not already sorted in increasing order. After the end of the first while loop, the elements $A[1], A[2], \ldots, A[i]$ are sorted in increasing order, since this is what the while loop checks, and these elements are copied to L, which will thus be an array sorted in increasing order. The array R is not necessarily sorted, but it has length strictly smaller than n, and so by the induction hypothesis *merge run sort* will sort R correctly in the fourth-to-last line of the algorithm. This means that on the next line *merge* will be run on two sorted arrays, and so the output A will be correctly sorted.

The correctness of the *merge runs sort* algorithm now follows by the principle of mathematical induction.

3b (30 p) Regardless of whether the sorting algorithm is correct or not, does it always terminate (assuming that the input is an array of elements that can be compared with `<=`) and, if so, what is the worst-case time complexity?

Solution: The algorithm will always terminate, since all recursive calls are made on arrays of strictly smaller size, and if an array has size 1, then no recursive call will be made.

It follows from the careful proof of correctness for *merge* in our solution to Problem 3a (or from what has been said in class) that this subroutine runs in time linear in the sums of the array lengths.

For *merge runs sort*, let the worst-case running time for arrays of length n be $T(n)$. Except for the calls to `merge` and `mergerunssort`, the running time is linear in n, dominated by the cost of copying the array A to L and R. As just discussed, `merge` also runs in linear time. For `mergerunssort`, in the worst case the size of the array R could be $n - 1$, which leads to a worst-case running time estimate

$$T(n) = Kn + T(n-1) \tag{5}$$

for some constant K. If we pick this constant K so that $K \geq T(1)$ (which we can always do if we like), and use the equality (5) to substitute $T(n-1)$ on the right, and then continue to substitute for $T(n-2)$, et cetera, then we get

$$\begin{aligned}
T(n) &= Kn + T(n-1) && \text{(6a)}\\
&= Kn + K(n-1) + T(n-2) && \text{(6b)}\\
&= Kn + K(n-1) + K(n-2) + T(n-3) && \text{(6c)}\\
&= Kn + K(n-1) + K(n-2) + K(n-3) + T(n-4) && \text{(6d)}\\
&= \cdots && \text{(6e)}\\
&= \textstyle\sum_{i=2}^{n} Ki \ + T(1) && \text{(6f)}\\
&= K \textstyle\sum_{i=1}^{n} i && \text{(6g)}\\
&= K\frac{n(n+1)}{2} && \text{(6h)}\\
&= \mathrm{O}\big(n^2\big) \ , && \text{(6i)}
\end{aligned}$$

resulting in a worst-case asymptotic running time $\mathrm{O}\big(n^2\big)$. As we will see in Problem 3c, unfortunately there are worst-case inputs triggering this quadratic running time, so this analysis is tight.

Remark: The general idea of making use of already sorted runs is very good, however, so if you want an additional exercise, you can think about how to take this idea and turn it into a version of *merge sort* that will always run in time $\mathrm{O}(n \log n)$ but will be faster on sorted or almost sorted arrays.

3c (20 p) Can you give any example of a family of input arrays of growing size for which Jakob's *merge runs sort* algorithm will output a correctly sorted array and be asymptotically faster than the *merge sort* algorithm that we have covered in class? Can you give any example of a family of input arrays of growing size for which *merge runs sort* will be asymptotically slower than standard *merge sort*?

Solution: If the input is an array $A = \{1, 2, 3, \ldots, n-2, n-1, n\}$ that is already sorted in order, then (as already discussed above in our solution to Problem 3a) *merge runs sort* will just scan through the array once until $i = n$ (since for all $i < n$ it will always be the case that $A[i] \leq A[i+1]$), after which it will return the input array A unchanged. This is correct, since A is already sorted, and will only take linear time $\mathrm{O}(n)$. The standard *merge sort* algorithm that we have covered in the course, however, will always take time $\Omega(n \log n)$, since there will be $\Omega(\log n)$ levels of recursive calls, and at each level the total time required for all *merge* operations is linear even if everything is already perfectly sorted. Hence, *merge runs sort* is asymptotically faster for an already sorted array.

If the input is instead an array $A = \{n, n-1, n-2, \ldots, 3, 2, 1\}$ sorted in decreasing order, then *merge runs sort* will be called recursively on all subarrays $A[2, n]$, $A[3, n]$, $A[4, n]$, … (where we use the notation $A[j, k]$ for the subarray of A between positions j and k inclusive), since in each recursive call the very first check $A[i] \leq A[i+1]$ will fail. For every array of size $n - i$ for $i = 1, 2, \ldots$, the ensuing *merge* call will then take linear time (as it always does). This means that the total running time will scale like $\sum_{i=1}^{n}(n-i) = \frac{(n-1)n}{2} = \Theta\bigl(n^2\bigr)$ (matching our analysis of Equation (5) in Problem 3b), so *merge runs sort* will run in quadratic time whereas *merge sort* always runs in time $\mathrm{O}(n \log n)$.

4 (60 p) When constructing this problem set, Jakob ran into some very unfortunate problems. When he wanted to add a couple of more examples illustrating the power of inductive proofs, the proofs turned out to be a little bit too powerful. Specifically, Jakob was able to use mathematical induction to show

1. that all swans have the same colour (presumably white, so that there are no black swans after all), and
2. that all positive integers are in fact equal.

Both of these claims are fairly disturbing from a mathematical point of view. Please help Jakob by pointing out clearly what goes wrong in his induction proofs below.

4a **Theorem.** All swans have the same colour.

Proof. We prove by induction over n that any set of n swans have the same colour.

For the base case, if we have a set of $n = 1$ swan, then this swan vacuously has the same colour as itself.

For the induction step, assume as our induction hypothesis that all sets of n swans have the same colour, and consider a set of $n + 1$ swans. Fix some swan S_1 in this set. If we remove S_1, then we have n swans left, and by the induction hypothesis they all have the same colour. Let C be this colour.

Now consider another swan S_2 in the set. If we remove S_2 from our set of $n + 1$ swans instead of S_1, then we again have n swans left, and they all have the same colour by the induction hypothesis. Since S_1 is one of the swans in this set, it must have the same colour C as all the others. Hence, all $n + 1$ swans have the same colour.

It follows by the principle of mathematical induction that any set of n swans must always have the same colour.

Solution: The base case is correct.

The induction step is also correct for all $n \geq 3$. It is true that if I have a set of $n \geq 3$ objects with the property that any subset of $n - 1$ objects must all have the same colour, then the only way this can happen is that all n objects have the same colour.

However, for $n = 2$ there is a gap in the argument, and this is why we can "prove" a false theorem. If we remove swan S_2 from the set, then there is only swan S_1 left. This swan S_1 could have a different colour, contrary to what the induction step argument claims.

4b **Theorem.** All positive integers are equal.

Proof. By induction over n.

For the base case, the positive integer 1 is equal to itself.

For the induction step, assume as our induction hypothesis that $n - 1 = n$. Adding 1 to both sides of this equality, we derive that $n = n + 1$.

It follows from this by the principle of mathematical induction that for all integers n the equality $n = n + 1$ holds. But then by transitivity we obtain that all integers are equal, and the claim in the theorem statement has thus been established.

Solution: The induction step in this proof is just fine, but the base case is wrong.

Since the induction hypothesis is that $n - 1 = n$, for the base case we would need to prove that $0 = 1$ or $1 = 2$ or something. But in the base case in the "proof" above, we are not proving a base case of the form $n - 1 = n$, but rather $n = n$. This is of course true, but this does not match what we need when we want to use the base case as induction hypothesis for our first induction step.