Introduktion til diskret matematik og algoritmer: Problem Set 1

Due: Wednesday February 14 at 9:59 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. 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) In the following snippet of code `A` is an array indexed from 1 to n containing elements that can be compared using the operator $=$.

```
for (i := 1 upto n)
    B[i] := 0
for (i := 1 upto n)
    for (j := 1 upto n)
        if (i != j and A[i] == A[j])
            B[i] := B[i] + 1
return B
```

1a (30 p) Explain in plain language what the algorithm above does. What is the meaning of the entries in the array `B` that the algorithm returns?

Solution: The algorithm first initializes all entries in the the array B to zero. Then it goes over all entries i in A. For every entry $A[i]$ the algorithm checks for how many other entries $A[j]$, $i \neq j$, it holds that $A[i] = A[j]$, and $B[i]$ is incremented every time this happens. Thus, at the end of the algorithm for every $i \in [n]$ $B[i]$ counts for how many $A[j]$, $i \neq j$, it holds that $A[i] = A[j]$.

1b (10 p) Provide an asymptotic analysis of the running time as a function of the array size n. (That is, state how the worst-case running time scales with n, focusing only on the highest-order term, and ignoring the constant factor in front of this term.)

Solution: All lines except for the for loops take a constant amount of time, so in order to determine the time complexity we just need to count how many times the lines in the for loops are executed.

The first for loop runs for i from 1 to n for a total of n steps. In the following nested for loop, the outer loop runs for n steps, and in every iteration of the outer loop the inner loop also runs for n steps. This means that the innermost lines in the if statement are executed n^2, and so the time complexity of the algorithm is $\mathrm{O}(n^2)$ (or even $\Theta(n^2)$ if you wish to be more precise).

1c (20 p) Suppose that we are guaranteed that all elements in the array `A` are integers between 1 and n. Can you improve the code to run faster while retaining the same functionality? How much faster can you get the algorithm to run? Analyse the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

Solution: If we know that all elements in the array A are guaranteed to be between 1 and n, then we can count how many copies we have of each element directly in an auxiliary array C, and then copy the numbers from C to B like so:

```
for (i := 1 upto n)
    C[i] := 0
for (i := 1 upto n)
    C[A[i]] := C[A[i]] + 1
for (i := 1 upto n)
    B[i] := C[A[i]] - 1
return B
```

Here $C[j]$ will count how many copies of the number j can be found in the array. If $A[i] = j$, then according to the description above this is the number that should be stored in $B[i]$, except that we should subtract 1 to compensate for $A[i]$ itself.

Again, all lines take a constant amount of time to execute, so we just need to analyse how many times the for loops are run, but this is clearly n times. The time complexity is hence $\mathrm{O}(n)$ (or $\Theta(n)$ to be precise). No algorithm could compute B asymptotically faster, since B itself has size n, and since we need to read through all of A, which also has size n, in order to compute the entries in B.

2 (70 p) Consider the snippet of code

```
check (A, lo, hi)
    mid := floor ((lo + hi) / 2)
    success := TRUE
    i := lo
    while (i < mid and success)
        if (A[i] > A[mid])
            success := FALSE
        i := i + 1
    i := mid + 1
    while (i <= hi and success)
        if (A[i] < A[mid])
            success := FALSE
        i := i + 1
    if (lo < mid - 1 and success)
```

```
        success := check (A, lo, mid - 1)
    if (mid + 1 < hi and success)
        success := check (A, mid + 1, hi)
    return success
```

where A is an array indexed from 1 to A.size that contains elements that can be compared, and the function floor rounds down to the nearest integer.

2a (30 p) Explain in plain language what the result is of an algorithm call check (A, 1, A.size) (i.e., do not just rewrite the pseudocode word by word, so that your text is just a more verbose version of the pseudocode, but interpret what the code is doing and why). In particular, what holds when the algorithm returns TRUE?

Solution: This is a recursive algorithm. It first computes an index mid so that $A[\mathsf{mid}]$ is the midpoint of the array lo and hi. Then follow two while loops. In the first while loop all elements $A[i]$ for $\mathsf{lo} \leq i < \mathsf{mid}$ are compared to $A[\mathsf{mid}]$, and if it ever holds that $A[i] > A[\mathsf{mid}]$, then the Boolean flag success is set to FALSE (which will cause the algorithm to return FALSE at the end). The second while loop checks in the same way for all indices i such that $\mathsf{mid} < i \leq \mathsf{hi}$ that $A[i] \geq A[\mathsf{mid}]$, or else success is set to FALSE.

After the two while loops, it follows from the discussion above that if the Boolean flag success is still TRUE, then $A[\mathsf{mid}]$ is greater than or equal to all elements preceding it in the array, and smaller than or equal to all elements succeeding it. That is, although we might not know much about the rest of the array, we know that in the array between $A[\mathsf{lo}]$ and $A[\mathsf{hi}]$ the element at $A[\mathsf{mid}]$ is in the right position if we would wish to sort the array in increasing order.

If there is more than one element between indices lo and $\mathsf{mid} - 1$ (inclusive) and if success is TRUE, then a recursive call is made on the subarray between these two positions, which will check in the same way if the middle element in the left subarray is in the right position with respect to sortedness. If no recursive call is made but success is TRUE, then we know from the discussion above that all elements between indices lo and mid are sorted. In the same way, if there are at least two elements between indices $\mathsf{mid}+1$ and hi (inclusive), a recursive call is made, and otherwise we know that all elements between indices mid and hi are sorted (assuming that success is TRUE), After all recursive calls (if any) have been made, we have that success is TRUE if all recursive checks yielded a positive result, meaning that all elements in the whole array were in the correct position with respect to sortedness in increasing order. Thus, the algorithm call check (A, 1, A.size) returns TRUE if and only if the whole array is sorted.

2b (20 p) Provide an asymptotic analysis of the running time as a function of the size of the array A.

Solution: To analyze the time complexity of the algorithm, we try to understand how much work is done per call and how many recursive calls are made.

Starting with the latter, we can see that every call of the algorithm generates at most two subcalls, and that these calls are made on arrays of (slightly less than) half the size. This means that if we let $n =$ A.size, then there will be a at most $\lceil \log n \rceil$ levels of recursion.

If we can analyse the time needed for each recursive call, then we can sum up to get the total time complexity. All lines inside the check algorithms run in constant time except for the while loops (and the recursive calls, but that time is accounted for in the analysis of each recursive call). All statements inside the while loops are also constant time. The while loops together iterate over all elements in the array (except mid) and so the total time required for these iterations, and hence for the full call, is linear in hi − lo, scaling like $K(\mathsf{hi} - \mathsf{lo})$ for some (large enough) constant K.

Summing up, we get:

- At top level, we have one call taking time at most $K \cdot n$,
- At recursion level 1, we have at most two calls taking time at most $K \cdot n/2$ each, for a total cost of $(K \cdot n/2) + (K \cdot n/2) = K \cdot n$ for all calls at level 1.
- At recursion level 2, we have at most four calls taking time at most $K \cdot n/4$ each, summing up to a cost of at most $K \cdot n$ for all calls at level 2.
- Et cetera, all the way down to recursion level $\lceil \log n \rceil$, where we will get at most a linear number of calls, each taking constant time.

We see that we have $O(\log n)$ levels of recursion, and that the total cost of the recursive calls at each level is $O(n)$. Hence, the time complexity of the algorithm is $O(n \log n)$ (and it is not hard to see that this is also a lower bound, so that the time complexity is in fact $\Theta(n \log n)$, but this is not needed for a full score).

2c (20 p) Can you improve the algorithm (i.e., change the pseudocode) to run asymptotically faster while retaining the same functionality? In case you can design an asymptotically faster algorithm, analyse the time complexity of your new algorithm. For the best algorithm that you have—either your new one, or the old one—can you prove that the running time of this algorithm is asymptotically optimal?

Solution: Since by analyzing the algorithm we have determined that it is simply checking whether the array is sorted in increasing order, it is not hard to write a more efficient algorithm performing this check. For instance, the following pseudocode should do the job:

```
check (A, lo, hi)
    success := TRUE
    i := lo
    while (i < hi and success)
        if (A[i] > A[i+1])
            success := FALSE
        i := i + 1
    return success
```

This check runs in linear time, which is clearly asymptotically optimal since we have to look at the full array to know for sure whether it is sorted or not.

3 (70+ p) In the following snippet of code `A` is an array indexed from 1 to n containing integers.

```
found := FALSE
i := 1
while (i <= n and not(found))
    j := 1
    while (j <= n and not(found))
        k := 1
        while (k <= n and not(found))
            if (i != j and j != k and i != k and A[i]+A[j]+A[k] == 0)
```

```
                found := TRUE
            else
                k := k+1
        if (not(found))
            j := j+1
    if (not(found))
        i := i+1
if (found)
    return (i, j, k)
else
    return "failed"
```

3a (20 p) Explain in plain language what the algorithm above does. What is the triple of numbers returned when the algorithm does not "fail"?

Solution: The algorithm tries to fine three distinct indices i, j, k such that $A[i]+A[j]+A[k] = 0$. If such a triple (i, j, k) exists, then this is what the algorithm returns, and otherwise `"failed"` is returned.

3b (20 p) Provide an asymptotic analysis of the running time as a function of the array size.

Solution: We have three nested while loops, each iterating from 1 to n. Except for the while loops, all lines in the code take a constant amount of time, and so we have to calculate the number of times the innermost lines in the while loops are executed. In the worst case (for instance, when there is no solution), our algorithm will make $n \cdot n \cdot n$ iterations of the innermost loop, which gives an asymptotic running time of $\mathrm{O}(n^3)$ (or $\Theta(n^3)$ to be precise).

3c (30 p) Improve the code to run faster while retaining the same functionality. How much faster can you get the algorithm to run? Analyse the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

Solution: A first, small, optimization, would be to let j range from $i + 1$ to n and k range from $j + 1$ to n. It is important to note that this does not yield any asymptotic improvement in running time, so this is not what we are after in this problem, but such a change would of course save a significant constant factor in running time in practice.

It is possible to get a further improvement to $\mathrm{O}(n^2)$, however. First we sort the array in increasing order, which can be done in time $\mathrm{O}(n \log n)$. Then we iterate through the array with i ranging from 1 to n. For a fixed i, set $j := i + 1$ and $k := n$ and compute the sum $S = A[i] + A[j] + A[k]$. If $S = 0$, we terminate. If $S > 0$, we need to decrease the sum, so we set $k := k - 1$. Otherwise $S < 0$, and we set $j := j + 1$ to increase the sum. Note that in every step we either increase j or decrease k, and so after a linear number of steps the two indices meet, and if there is any solution with the smallest element in the sum being $A[i]$, then the above loop will find it. Running this loop for i ranging from 1 to n is a linear number of iterations, so that total running time for the two nested loops is $\mathrm{O}(n^2)$. Since the initial sorting step takes time $\mathrm{O}(n \log n)$, the total running time is $\mathrm{O}(n^2)$.

3d *Bonus problem (worth 40 p extra):* This is in fact a well-known research problem. What information can you find about this on the internet? What are the best known upper and lower bounds for algorithms solving this problem? Explain how you dug up the information, and give references to where it can be found. (Note that for this problem, in contrast to most other problems on the problem sets, you are expected and encouraged to search on the Internet for answers.)

Solution: There is no obviously correct and complete answer to this literature search problem, but searching on the internet, it is not too hard to discover that this is actually an extremely well-studied research problem known as the 3-Sum problem. The current best known algorithm for this problem has running time something like $\mathrm{O}\big(n^2(\log\log n)^k/\log^2 n\big)$ for some constant k. It is fairly widely believed that it is impossible to solve this problem significantly faster than $\mathrm{O}\big(n^2\big)$ in the worst case, by which we mean that there is not expected to exist any algorithm that runs in time $\mathrm{O}\big(n^{2-\varepsilon}\big)$ for any constant $\varepsilon > 0$. Digging up this information is sufficient to get a full score on this subproblem.