KØBENHAVNS
UNIVERSITET

Diskret Matematik og Formelle Sprog: Problem Set 1

Due: Monday February 15 at 23:59 CET.
Submission: Please submit your solutions via *Absalon* as 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. *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 in the course information 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 pseudo-code. Also, no such material may be downloaded from the internet and/or used verbatim. 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 on the problems. Good luck!

1 In the following snippet of code `A` is an array indexed from 1 to n containing elements that can be compared using the operator $<$.

```
stop := FALSE
i := 1
while (i <= n and not(stop))
    j := 1
    found := FALSE
    fail  := FALSE
    while (j <= n and not(fail))
        if (A[i] < A[j])
            if (found)
                fail := TRUE
            else
                found := TRUE
        j := j+1
    if (found and not(fail))
        stop := TRUE
    else
        i := i+1
if (stop)
    return A[i]
else
    return "failed"
```

1a (30 p) Explain in plain language what the algorithm above does. What is the number that the algorithm returns when it does not return "failed", and when does the algorithm declare failure?

Solution: The algorithm tries to find the element indexed with i such that there exists exactly one element that has higher value than $A[i]$. If such an element exists, the algorithm returns this element, and otherwise it returns `failed`.

1b (20 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: The algorithm has two nested `while` loops. For each element of the array `A` (first while loop) the algorithm compares it with every other element of the array (including itself) to see if certain conditions are satisfied (second while loop). The number of comparisons that is done in the worst case is equal to $n \cdot n$ (this happens, for instance, for an array of distinct elements sorted in increasing order, except that the order of the last two elements is flipped), and the amount of additional work per comparison is easily seen to be at most a constant. Thus, the running time of the algorithm is $\mathrm{O}\big(n^2\big)$ (or, if we want to be more precise, $\Theta\big(n^2\big)$, but just answering with big-oh is perfectly fine).

1c (20 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? (That is, that no algorithm solving this problem can run faster except possibly for a constant factor in the highest-order term or improvements in lower-order terms.)

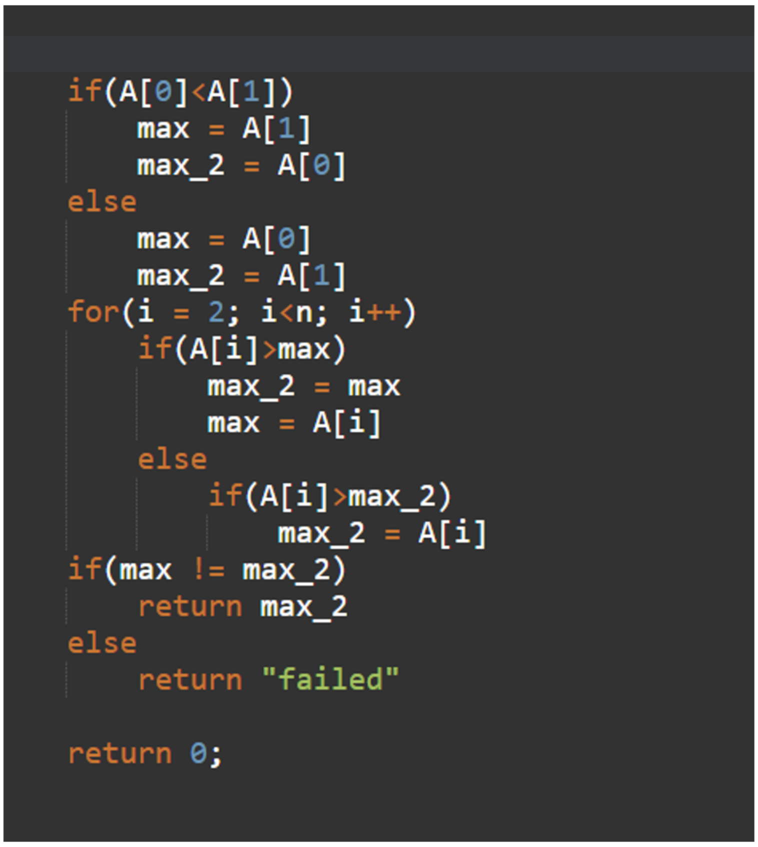
Solution: The snippet of code in Figure 1 runs faster while retaining the same functionality. It goes through the array only once, thus having a running time of $O(n)$. Since we want to find the second largest element of the array, we need to go through the array at least one time, meaning that this solution is asymptotically optimal. (This is a general claim that holds in most settings, at least in introductory algorithms classes: In order to be correct, the algorithm has to be given a chance to read the data, and so any linear-time algorithm is asymptotically optimal.)

In the improved version of the algorithm, we instantiate the largest (`max`) and the second largest element (`max_2`) from the first two elements of the array. In the `for` loop we compare the current element of the array with the `max` and `max_2` changing the values of the `max` and `max_2` if necessary. Finally, if `max` and `max_2` are different values we print the `max_2`.

A slightly suboptimal solution would be to instead sort the whole array in time $\mathrm{O}(n \log n)$ and then output the second largest element if if is distinct from the largest element, but this is not asymptotically optimal.

2 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
```



```
if(A[0]<A[1])
    max = A[1]
    max_2 = A[0]
else
    max = A[0]
    max_2 = A[1]
for(i = 2; i<n; i++)
    if(A[i]>max)
        max_2 = max
        max = A[i]
    else
        if(A[i]>max_2)
            max_2 = A[i]
if(max != max_2)
    return max_2
else
    return "failed"

return 0;
```

Figure 1: Pseudo-code for more efficient version of algorithm in Problem 1.

2a (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: Very, very briefly, `B[i]` will contain the number of other positions $j \neq i$ for which `A[j]`$=$`A[i]`.

2b (20 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: Very, very briefly, two nested for loops, so $\Theta(n^2)$.

2c (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: Very, very briefly, yes, the code can be improved. Create a new array `C` initialized to all zeros, and whenever `A[i]`$= j$ just increment `C[j]`. Then copy the required information to `B` (adjusting by an additive one). This will run in linear time, which is clearly optimal since we have to read the input and this takes linear time.

3 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 find three distinct indices i, j, k such that $A[i]+A[j]+A[k] = 0$. If such i, j, k exist, the triple (i, j, k) is returned as result, and otherwise the algorithm returns `"failed"`.

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. The total amount of work is at most a constant times the number of iterations in the innermost loop. 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, thus giving us a running time of $\mathrm{O}(n^3)$.

3c (40 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 in the loops. This does not save asymptotically, but would be a signficant constant factor gain in practice.

We can further improve our algorithm to run in $\mathrm{O}(n^2)$ time. First we sort the array, which can be done in time $O(n \log n)$. Then we iterate through the array with i ranging from 1 to n. For each index i, set $j = i + 1$ and $k = n - 1$ and sum elements $A[i] + A[j] + A[k]$. If the sum is equal to zero, we terminate, otherwise we move either index j or index k based on the result (if the sum is grater than zero we set $k = k - 1$, otherwise we set $j = j + 1$). For each i we go through the whole array, thus our running time is $O(n^2)$. The total running time then is $O(n \log n) + O(n^2) = O(n^2)$.

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.

Solution: There is no obvious "correct 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 problem known as the 3-SUM problem. According to Wikipedia, the current best known running time is $\mathrm{O}\bigl(n^2(\log\log n)^{O(1)}/\log^2 n\bigr)$. It is believed that it is impossible to solve this problem in time $\mathrm{O}(n^{2-\epsilon})$ for $\epsilon > 0$ in the worst case. Providing this information is sufficient for a full score on this subproblem.