KØBENHAVNS UNIVERSITET

Diskret Matematik og Formelle Sprog: Problem Set 1

Due: Monday February 13 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. 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 In the following snippet of code `A` is an array indexed from 1 to `A.size` that contains elements that can be compared.

```
n      := A.size
failed := FALSE
i      := 1
while (i <= n and not failed)
    j := i + 1
    while (j <= n and not failed)
        k := j + 1
        while (k <= n and not failed)
            if (A[i] == A[j] or A[i] == A[k] or A[j] == A[k])
                failed := TRUE
            k := k + 1
        j := j + 1
    i := i + 1
if (failed)
    return "fail"
else
    return "success"
```

1a (30 p) Explain in plain language what the algorithm above does (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, explain what properties of the array `A` will make the algorithm return "success" or "fail" and why.

Solution: The algorithm has three nested while loops that loop over all triples of elements $A[i]$, $A[j]$, $A[k]$, for $1 \leq i < j < k \leq n$. If in any such triple two array elements are found to be equal, the algorithm terminates and returns "fail"; otherwise it returns "success". In slightly less convoluted terms, this means that if any two elements in the array are equal, then the algorithm declares failure, but the execution is deemed a success if all elements in the array A are distinct.

There is a slightly annoying special case here in that if the array is of size 2, then the algorithm will return success even if the two elements in the array are identical. For this particular problem, which was meant to be a fairly easy warm-up problem, we have given a full score even if you did not catch this case.

1b (20 p) Provide an asymptotic analysis of the running time as a function of the array size n. (That is, state how the running time scales with n, focusing only on the highest-order term, and ignoring the constant factor in front of this term.)

Solution: There are no recursive function calls in this code, so we just need to analyse the number of steps from start to finish. The first three lines and the last four lines take constant time, so to analyse the time complexity we need to understand the while loops. All assignment and comparison operations inside the while loops also take constant time, so the running time of the nested while loops will scale like the number of times the innermost while loop is run. This happens for all i, $1 \leq i \leq n$, for all j, $i < j \leq n$, and for all k, $j < k \leq n$, or, as we wrote above, for all triples (i, j, k) such that $1 \leq i < j < k \leq n$. There are $\Theta\big(n^3\big)$ such triples. We note that if all elements in the array are distinct, then the algorithm will run through all triples before terminating, and so the worst-case running time is $\Theta\big(n^3\big)$. (Answering $\mathrm{O}\big(n^3\big)$ is also sufficient for a full score.)

1c (20 p) Suppose that in the array `A` we are guaranteed that all array entries are positive integers between 1 and `A.size` (inclusive). 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: One obvious improvement of the code is to just loop over pairs, since we only compare pairs anyway. This yields something like the following (if we also take care of the special case of arrays of size 2 as explained above):

```
n      := A.size
failed := FALSE
i      := 1
while (i <= n and not failed)
    j := i + 1
    while (j <= n and not failed)
        if (A[i] == A[j])
            failed := TRUE
        j := j + 1
```

```
    i := i + 1
if (not failed or n == 2)
    return "success"
else
    return "fail"
```

This algorithm runs in time $\Theta(n^2)$, which is already a significant improvement.

If we have a bound $O(n)$ on the size of the integers in the array, we can create an auxiliary array B to keep track of whether we have seen any element in the array more than once. We initialize all elements in B to `FALSE`, mark $B[k]$ to be `TRUE` whenever we see an element k in A, and terminate whenever we discover an element in B that has already been marked. In this subproblem we have the guarantee that all elements should be between 1 and n, where n is the size of the array, which yields pseudocode along the following lines (again taking care of the special case of size 2):

```
n := A.size
B := new array of size n
for (i := 1 upto n)
    B[i] := FALSE
failed := FALSE
i      := 1
while (i <= n and not failed)
    if (A[i] <= 0 or A[i] > n)
        error ("array element value out of bounds")
        exit
    if (B[A[i]])
        failed := TRUE
    B[A[i]] := TRUE
    i := i + 1
if (not failed or n == 2)
    return "success"
else
    return "fail"
```

(The error checking would not be needed to get full score for a solution, but has been inserted just to provide an example of good programming practice.)

In the pseudocode above, the first line takes constant time and the second line might take time $O(n)$ (since we need to reserve that much space. The for loop runs for n steps and each step takes constant time (it is just an assignment). Then we have two more lines of constant-time assignments before we reach the while loop. Clearly, the while loop runs for at most n steps and all instructions inside the loop are constant-time operations. The final lines are also constant-time operations, so the running time of the whole algorithm is $\Theta(n)$. This is clearly asymptotically optimal, since in order to decide whether all elements in an array are optimal we need to look at all elements in the worst case, and this takes linear time.

2 In the following snippet of code `A` and `B` are arrays indexed from 1 to `A.size` and `B.size`, respectively, that contain elements that can be compared.

```
m := A.size
n := B.size
```

```
failed := FALSE
for (i := 1 upto m)
    for (j := 1 upto n)
        if (A[i] < B[j])
            failed := TRUE
if (failed)
    return "fail"
else
    return "success"
```

2a (30 p) Explain in plain language what the algorithm above does (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, explain what properties of the arrays `A` and `B` will make the algorithm return "success" or "fail".

Solution: The algorithm compares all array elements $A[i]$ and $B[j]$ for $1 \leq i, j \leq n$. If it every finds a pair (i, j) such that $A[i] < B[j]$, then it declares failure, but otherwise it returns success. That is, the algorithm execution is considered successful if for all elements $A[i]$ and $B[j]$ it holds that $A[i] \geq B[j]$, or, in other words, if the smallest element in A is at least as large as the largest element in B.

2b (20 p) Assume that both arrays `A` and `B` have size n. Provide an asymptotic analysis of the running time as a function of n.

Solution: There are no recursive function calls in this code, so we just need to analyse the number of steps from start to finish. The first three lines and the last four lines take constant time, so to analyse the time complexity we need to understand the nested for loops, where in accordance with the problem statement we assume that $m = n$. There is a constant number of assignment and comparison operations inside the for loops, which all take constant time. Hence, the running time of the nested loops will scale like the number of times the innermost loop is run, which clearly is $m \cdot n = n^2$ times (and the algorithm will always run for this number of nested loop iterations). Thus, the running time (worst-case and best-case) is $\Theta(n^2)$. (Answering $\mathrm{O}(n^2)$ is also sufficient for a full score.)

2c (20 p) Can you improve the algorithm 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 (where, to simplify matters, we again assume that both arrays `A` and `B` have the same size n). 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: Using our analysis in Problem 2b, we see that it is sufficient to compute the smallest element in A and the largest element in B, and then compare them. This can be done as follows:

```
m    := A.size
n    := B.size
minA := A[1]
for (i := 2 upto m)
    if (minA > A[i])
        minA := A[i]
```

```
maxB := B[1]
for (i := 2 upto n)
    if (maxB < B[i])
        maxB := B[i]
if (minA >= maxB)
    return "success"
else
    return "fail"
```

The first three lines and the last four lines again run in constant time. There is a constant number of operations inside the for loops, which all take constant time. The total number of iterations of the two for loops is equal to the total number of elements in the arrays A and B, and so the running time of our optimized algorithm is linear in the size of the input. This is clearly asymptotically optimal, since we have to look at all of the array A in order to find the minimal element and all of the array B in order to find the maximal element.

3 In the following snippet of code `A` is an array indexed from 1 to `A.size` that contains elements that can be compared, and `B` is intended to be an auxiliary array of the same type and size. The functions `floor` and `ceiling` round down and up, respectively, to the nearest integer.

```
m := A.size
for (i := 1 upto m)
    B[i] := A[i]
while (m > 1)
    for (i := 1 upto floor(m / 2))
        if (B[2 * i - 1] >= B[2 * i])
            B[i] := B[2 * i - 1]
        else
            B[i] := B[2 * i]
    if (m mod 2 == 1)
        B[ceiling(m / 2)] := B[m]
    m := ceiling(m / 2)
return B[1]
```

3a (30 p) Explain in plain language what the algorithm above does (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, how can you describe the element `B[1]` that is returned at the end of the algorithm?

Solution: Let us write M to denote the value of m at the start of the algorithm, i.e., the size of the array A. The algorithm starts by copying all of the elements in the array A to the array B.

In the first iteration of the while loop, each pair of elements $B[2i-1]$ and $B[2i]$ are compared, and the largest of the two is copied to $B[i]$. If the size of the array $m = M$ is odd, the final element $B[m]$ is copied to $B[\lceil m/2 \rceil]$. Finally, the size of the array is adjusted to $m := \lceil M/2 \rceil$ and a new iteration of the while loop is initiated.

As long as there are two or more elements left in $B[1], \ldots, B[m]$ i.e., as long as $m > 1$ each pair of elements $B[2i-1]$ and $B[2i]$ for $i = 1, \ldots, \lfloor m/2 \rfloor$ are compared and the largest one is kept by being copied to $B[i]$, and any final uncompared element is also copied as in the first iteration. After the size of the array has been adjusted to $m := \lceil m/2 \rceil$ at the end of an iteration,

it holds for every element in $B[m+1], \ldots, B[M]$ that it is smaller than or equal to than some element in $B[1], \ldots, B[m]$. If $m > 1$, a new iteration of the while loop is initiated.

When the while loop ends we have $m = 1$, and it follows from the description above that $B[1]$ is a largest element in the array (since every element $B[2], \ldots, B[M]$ is smaller than or equal to $B[1]$). That is, the code snippet finds and returns a largest element in the array A.

3b (30 p) Provide an asymptotic analysis of the running time as a function of the array size m.

Solution: As before, let us write M to denote the size of the input, i.e., the array A. The initialization code

```
m := A.size
for i := 1 upto m
    B[i] := A[i]
```

clearly runs in time $\mathrm{O}(M)$.

Inside the while loop, there is a for loop

```
for (i := 1 upto floor(m / 2))
    if (B[2 * i - 1] >= B[2 * i])
        B[i] := B[2 * i - 1]
    else
        B[i] := B[2 * i]
```

that does a constant amount of work per element $B[1]$, $B[2]$, ..., $B[\,2 \cdot \lfloor m/2 \rfloor\,]$, namely, it compares the elements pairwise and copies the largest one in each pair, and the final lines of code inside the while loop

```
if (m mod 2 == 1)
    B[ceiling(m / 2)] := B[m]
m := ceiling(m / 2)
```

then just copy the final element if m is odd, after which m is divided by 2 and rounded up. All of this work in an iteration of the while loop can be upper-bounded by $K \cdot m$ for some (small) constant K. The final return call after the while loop is just a constant amount of work and can be ignored.

The while loop is run for values $m = M, \lceil M/2 \rceil, \lceil \lceil M/2 \rceil / 2 \rceil, \ldots$ all the way down to $m = 2$. If we round up M to the nearest power of 2, which cannot decrease our estimate of the running time, we can think of the while loop being run for $m = 2^i$ for $i = \lceil \log M \rceil, \lceil \log M \rceil - 1, \ldots, 1$, and we get that the total amount of work in all iterations of the while loop is

$$\sum_{i=1}^{\lceil \log M \rceil} K \cdot 2^i \;\leq\; K \cdot \sum_{i=0}^{\lceil \log M \rceil} 2^i \;=\; K \cdot \frac{2^{\lceil \log M \rceil} - 1}{2 - 1} \;\leq\; K \cdot 2M \;, \tag{1}$$

which is linear in M, i.e., $\mathrm{O}(M)$.

Switching back to denoting the array size by m as in the problem statement, we see that the algorithm runs in time $\mathrm{O}(m)$, i.e., in time linear in the size of the array. It is not hard to verify that all of the estimates above also have lower bounds matching up to constant factors, and so if we want to be really precise we can state the running time as $\Theta(m)$, but this is not needed for a full score.

3c (20 p) Can you improve the code 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: In order to find a largest element in an array we have to go over the whole array, and so linear time is asymptotically optimal. Hence, the answer to the first question in Problem 3c is that no, we cannot solve the same problem asymptotically faster.

However, it is certainly possible to construct a simpler algorithm that would probably also have a better constant factor in the linear running time, for instance, something like this:

```
max := A[1]
for i := 2 upto A.size
    if ([A[i] > max)
        max := A[i]
return max
```

In case you did not get a tight analysis in Problem 3b, you could get a full score on Problem 3c by presenting a linear time algorithm and noting why this is optimal. Otherwise, an explanation why the original algorithm is already asymptotically optimal is also sufficient for a full score on Problem 3c.