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.

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.)

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?

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".

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.

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?

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?

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

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?