KØBENHAVNS UNIVERSITET

Introduktion til diskret matematik og algoritmer: Problem Set 1

Due: Wednesday February 12 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) In the following snippet of code `A` and `B` are arrays indexed from 1 to n that contain numbers.

```
for i := 1 upto n {
    B[i] := 1
    for j := 1 upto i {
        B[i] := B[i] * A[j]
    }
}
```

1a (30 p) Explain in plain language what the algorithm above does. In particular, what is the meaning of the entries `B[i]` after the algorithm has terminated?

Solution: At termination we will have $B[i] = \prod_{j=1}^{i} A[i]$, i.e., $B[i]$ is the product of all entries $A[1]$, $A[2]$, …, $A[i-1]$, $A[i]$.

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: The algorithm has two nested for loops. The outer loop runs for i from 1 to n and the inner loop runs from 1 to i. Inside the innermost loop a constant number of operations

are performed. The asymptotic time complexity is therefore determined by the total number of times the inner loop is executed.

It follows from what is written above that the inner loop will run a total of $\sum_{i=1}^{n} i$ times, which is $\Theta(n^2)$. If we want to prove this from first principles, then we can observe that $\sum_{i=1}^{n} i \leq \sum_{i=1}^{n} n = n^2$, which shows that the running time is $\mathrm{O}(n^2)$, and also that $\sum_{i=1}^{n} i \geq \sum_{i=n/2}^{n} n/2 = n^2/4$, which shows that the running time is $\Omega(n^2)$.

In general, for this introductory course we do not care so much about the distinction between big-oh and big-theta, so correct answers with only big-oh bounds will also be acceptable unless stated otherwise (and as long as these bounds are tight).

1c (20 p) 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? (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: From our analysis of the algorithm above, it should be clear that the snippet of code below has the same functionality.

```
B[1] := A[1]
for i := 2 upto n {
    B[i] := A[i] * B[i-1]
}
```

It goes through the array only once, thus having a running time of $\mathrm{O}(n)$. Since we need to store n values in the array B, it is clear that linear time is optimal.

2 (60 p) In the following snippet of code `A` is an array indexed from 1 to n that contain elements that can be compared

```
j    := n
good := TRUE
while (j > 1 and good)
    i := j - 1
    while (i >= 1 and good)
        if (A[i] > A[j])
            good := FALSE
        i := i - 1
    j := j - 1
if (good)
    return "success"
else
    return "failure"
```

2a (30 p) Explain in plain language what the algorithm above does. In particular, what do we know about the array `A` when `"success"` or `"failure"` is returned, respectively?

Solution: The two nested while loops in the algorithm will compare all elements $A[i]$ and $A[j]$ for $1 \leq i < j \leq n$, and if it ever happens that $A[i] > A[j]$ the algorithm will set the Boolean flag `good` to false, exit the while loops, and return failure. Otherwise it will return success. That is, success is returned if and only if the elements in the array are sorted in increasing order $A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[n]$.

2b (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: The algorithm has two nested while loops. The outer loop runs for j from n down to 2 and the inner loop runs from $j-1$ to 1 in the worst case (which is when the array is in fact sorted). Inside the loops a constant number of operations are performed. The asymptotic time complexity is therefore determined by the total number of times the inner loop is executed.

It follows from what is written above that the inner loop will run a total of $\sum_{i=1}^{n-1} i$ times, which is $\Theta\big(n^2\big)$. Again, for this introductory course we do not care so much about the distinction between big-oh and big-theta, so correct answers with only big-oh bounds will also be acceptable unless stated otherwise (and as long as these bounds are tight).

2c (20 p) 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? (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: From our analysis of the algorithm above, it should be clear that the snippet of code below checks that the array is sorted in increasing order, and so has the same functionality.

```
for i := 1 upto n-1 {
    if (A[i] > A[i+1])
        return "failure"
}
return "success"
```

This algorithm runs in linear time, which is optimal since we have to look at all elements in an array to be able to determine whether the array is sorted or not.

3 (80 p) In the following snippet of code `A` is an array indexed from 1 to n that contains integers, and `B` is an auxiliary array, also indexed from 1 to n, that is meant to contain Boolean values.

```
for i := 1 upto n {
    if (A[i] < 1 or A[i] > n)
        return "failure"
}
i     := 1
found := -1
while (i <= n and found < 0) {
    for j := 1 upto n {
        B[j] := false
    }
    j := i
    while (B[j] == false) {
        B[j] := true
        j := A[j]
    }
    if (A[A[j]] == j)
```

```
        found := j
    i := i + 1
}
return found
```

3a (40 p) Explain in plain language what the algorithm above does. In particular, when does it return a positive value, and, if it does, what is the meaning of this value?

Solution: First, the algorithm checks that all array entries $A[i]$ for $1 \leq i \leq n$ are between 1 and n. It then does the following, starting from all positions $i = 1, 2, \ldots, n$:

- Visit all positions $j = i$, $A[i]$, $A[A[i]]$, $A[A[A[i]]]$, ... in the array, stopping as soon as we see a number we have seen before (which we keep track of using the array B). Note that we will never get problems with array indices being out of bounds, thanks to the checks in the first for loop.
- As soon as $A[j] = k$ for some k already seen before, check if $A[k] = j$, i.e., if the two entries $A[j]$ and $A[k]$ "point at each other".
- If this is the case, then terminate and return j, otherwise increment i and continue.

If a positive value j is returned, then this is an array index such that $A[A[j]] = j$.

3b (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 initial for loop takes linear time. In the main part of the algorithm we have a while loop that will require n iterations in the worst case, and a nested for loop (resetting the array B) that requires n steps. The nested while loop checking entries $B[j]$ will also run for at most n iterations (and every iteration involves a constant amount of work), since every iteration sets a new entry $B[j]$ to true and there are n entries all in all. Therefore, the total time inside the outermost while loop is $\mathrm{O}(n)$, which means that the time complexity of the whole algorithm is $\mathrm{O}\big(n^2\big)$. (In fact, it is not hard to argue that the above analysis is tight, so that the bound is $\Theta\big(n^2\big)$, but correct bounds stated in big-oh notation is sufficient for a full score on this course unless explicitly stated otherwise.)

3c (20 p) 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: From our analysis of the algorithm above, it is clear that the code does the following two things:

1. It first checks that all array entries $A[i]$ for $1 \leq i \leq n$ are between 1 and n.
2. It then tries to find an i for which $A[A[i]] = i$.

We get exactly the same functionality by instead coding up an algorithm that uses the following snippet of code:

```
for i := 1 upto n {
    if (A[i] < 1 or A[i] > n)
        return "failure"
}
i     := 1
found := -1
while (i <= n and found < 0) {
    if (A[A[i]] == i)
        found := i
    i := i + 1
}
return found
```

This algorithm runs in linear time, which is optimal — if, for instance, we have $A[i] = i + 1$ for $1 \leq i \leq n - 1$ and $A[n] = 1$, then the algorithm will have to look at the whole array in order to determine that there is no solution.

A slightly shorter solution that might be worth discussing is as follows:

```
i     := 1
found := -1
while (i <= n and found < 0) {
    j := A[i]
    if (j < 1 or j > n)
        return "failure"
    else if (A[j] == i)
        found := i
    i := i + 1
}
return found
```

Note that this solution is in fact not fully equivalent, since it could be that we find and return an i such that $A[A[i]] = i$ before detecting that there are out-of-bounds entries in the array. However, for an introductory course like this we will give a full score also to solutions that do not give exactly the same behaviour for corner cases like the `"failure"` case in this algorithm, which are anyway somewhat tangential to the main point of the problem at hand.

4 (80 p) In the following snippet of code `A` is an array indexed from 1 to $n \geq 2$ containing integers.

```
search (A, lo, hi)
    if (A[lo] >= A[hi])
        return "failure"
    else if (lo + 1 == hi)
        return lo
    else
        mid = floor ((lo + hi) / 2))
        if (A[mid] > A[lo])
            search (A, lo, mid)
        else
            search (A, mid, hi)
```

The first call to the algorithm is `search (A, 1, n)`, where n is whatever size (at least 2) the array has.

4a (40 p) Explain in plain language what the algorithm above does. If the algorithm returns something other than `"failure"`, then what is the meaning of the value returned?

Solution: The algorithm first checks that the left endpoint of the array `A[lo]` is strictly smaller than the right endpoint `A[hi]`. Once we know this, it will be the case for any other array entry `A[i]` that `A[i]` $>$ `A[lo]` or `A[i]` $<$ `A[hi]` (or both), and since `mid` is chosen so as to maintain the invariant that the left endpoint of the array is strictly smaller than the right endpoint, recursive calls can never return `"failure"`. This answers part of Problem 4c below.

Given that we do not have a failure for the first recursive call, the algorithm will maintain the invariant that `A[lo]` $<$ `A[hi]` and the difference `hi`$-$`lo` will decrease as long as `hi`$-$`lo` ≥ 2. When `lo` $=$ `hi` $+1$ the value `lo` will be returned, and this is an array index such that `A[lo]` $<$ `A[lo+1]`.

4b (20 p) Provide an asymptotic analysis of the running time as a function of the array size n.

Solution: The amount of work per recursive call of the `search` algorithm is constant, so the running time is determined by the number of recursive calls.

If the size of the array m before a recursive call is even $m = 2k$, then the recursive call will be made either on an array or size k or on one of size $k + 1$. If the size odd $m = 2k + 1$ is odd, then the recursive call will be made on an array of size $k + 1$. We see that (apart from an annoying additive 1) the array size will halve in every recursive call, and this means that after a logarithmic number of recursive calls the algorithm will terminate. The time complexity of the algorithm is therefore $\mathrm{O}(\log n)$ (or $\Theta(\log n)$, if we wish to be fully precise).

Just to be clear, the above argument is a bit handwavy, but it will be sufficient for a full score on an introductory course like this. If we wanted to be more formal, then we could argue, e.g., along the following lines:

- For arrays of size less than 12, the algorithm will take constant time for some constant K (just take the max of the worst-case running times for all arrays of sizes 2, 3, 4, ..., 11).
- For arrays of size at least 12, the array size will shrink by at least a factor 3/2 for each recursive call.

This means that the running time will be proportional to $\log_{3/2} n + K$, which is $\mathrm{O}(\log n)$.

4c (20 p) Could it be the case that recursive calls of the algorithm also return `"failure"`, or would it be sufficient to check just once before making the first recursive call? If we get the additional guarantee that all elements in the array are distinct, could we remove the `"failure"` check completely, since we would be guaranteed to never have this answer returned anyway? What about if we get the additional guarantee that the array is sorted in increasing order? What if both of these extra guarantees apply?

Solution: The answer `"failure"` can only be returned for the very first recursive call, as explained in the solution to Problem 4a. In order for this never to happen, we need to know that `A[1]` $<$ `A[n]`. This, in turn, is certainly guaranteed if the array is sorted in increasing order and all elements are distinct, but it does not hold if we only have one of the guarantees in the problem statement above.