KØBENHAVNS UNIVERSITET

Introduktion til diskret matematik og algoritmer: Problem Set 1

Due: Monday February 16 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 (50 p) This problem is about different representations of integers.

1a (10 p) Write the binary number $(110)_2$ in decimal notation.

Solution: We have $(110)_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6$.

1b (20 p) Write the decimal number 110 in binary notation.

Solution: Dividing by 2 and outputting the remainders will give us the bits in the binary expansion in reverse order. We obtain

$$\begin{aligned} 110 &= 55 \cdot 2 + 0 \\ 55 &= 27 \cdot 2 + 1 \\ 27 &= 13 \cdot 2 + 1 \\ 13 &= 6 \cdot 2 + 1 \\ 6 &= 3 \cdot 2 + 0 \\ 3 &= 1 \cdot 2 + 1 \\ 1 &= 0 \cdot 2 + 1 \end{aligned}$$

from which we see that $(110)_{10} = (1101110)_2$. Just to verify that we have not made any mistake, we can check our answer by computing $(1101110)_2 = 1{\cdot}2^6{+}1{\cdot}2^5{+}0{\cdot}2^4{+}1{\cdot}2^3{+}1{\cdot}2^2{+}1{\cdot}2^1{+}0{\cdot}2^0 = 64 + 32 + 8 + 4 + 2 = 110$.

1c (20 p) Write the octal number $(2025)_8$ in decimal notation.

Solution: We have $(2025)_8 = 2 \cdot 8^3 + 0 \cdot 8^2 + 2 \cdot 8^1 + 5 \cdot 8^0 = 2 \cdot 512 + 2 \cdot 8 + 5 = 1045$.

2 (50 p) Use the algorithm we have learned for determining $d = \gcd(m, n)$ for the numbers below, showing details of all function calls made, and then express d as a linear combination of m and n.

2a (20 p) $m = 38$ and $n = 14$.

Solution: We use the Euclidean algorithm, which is based on the observation that $\gcd(m, n) = \gcd(n, m \bmod n)$.

For $m = 38$ and $n = 14$ we obtain

$$\begin{aligned} 38 &= 2 \cdot 14 + 10 \\ 14 &= 1 \cdot 10 + 4 \\ 10 &= 2 \cdot 4 + 2 \\ 4 &= 2 \cdot 2 + 0 \end{aligned}$$

from which we see that $\gcd(38, 14) = 2$. By considering the above equalities in reverse order we can write the greatest common divisor

$$\begin{aligned} 2 &= 10 - 2 \cdot 4 \\ &= 10 - 2 \cdot (14 - 1 \cdot 10) \\ &= 3 \cdot 10 - 2 \cdot 14 \\ &= 3 \cdot (38 - 2 \cdot 14) - 2 \cdot 14 \\ &= 3 \cdot 38 - 8 \cdot 14 \end{aligned}$$

as a linear combination of 38 and 14 as desired.

2b (30 p) $m = 117$ and $n = 69$.

Solution: For $m = 117$ and $n = 69$ we obtain

$$\begin{aligned} 117 &= 1 \cdot 69 + 48 \\ 69 &= 1 \cdot 48 + 21 \\ 48 &= 2 \cdot 21 + 6 \\ 21 &= 3 \cdot 6 + 3 \\ 6 &= 2 \cdot 3 + 0 \end{aligned}$$

from which we see that $\gcd(117, 69) = 3$. Processing these equalities in reverse order we obtain

$$\begin{aligned} 3 &= 21 - 3 \cdot 6 \\ &= 21 - 3 \cdot (48 - 2 \cdot 21) \\ &= 7 \cdot 21 - 3 \cdot 48 \\ &= 7 \cdot (69 - 1 \cdot 48) - 3 \cdot 48 \\ &= 7 \cdot 69 - 10 \cdot 48 \\ &= 7 \cdot 69 - 10 \cdot (117 - 1 \cdot 69) \\ &= 17 \cdot 69 - 10 \cdot 117 \end{aligned}$$

as a linear combination of 117 and 69 as desired.

3 (60 p) In the following snippet of code `A` is an array indexed from 1 to n that contains numbers.

```
j := 1
while (j <= n) {
    A[j] := 0
    for i := j downto 1 {
        A[j] := A[j] + i * i
    }
    j := j + 1
}
```

3a (30 p) Explain in plain language what the algorithm above does. In particular, what are the numbers that are computed and stored in the array `A`?

Solution: We have an outer while loop where j goes from i to n. In every iteration, the inner loop will sum up all squares $j^2, (j-1)^2, (j-2)^2, \ldots 2^2, 1^2$ (in decreasing order) and store this sum in $A[j]$. That is, when the algorithm terminates the jth element in the array will contain the sum $\sum_{i=1}^{j} i^2$ of the j first squares for $j \in [n]$.

3b (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 1 to n and the inner loop runs from j down to 1. 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 $\sum_{j=1}^{n} j$ times, which is $\Theta(n^2)$. If we would want to prove this from first principles, then we can observe that $\sum_{j=1}^{n} j \leq \sum_{j=1}^{n} n = n^2$, which shows that the running time is $\mathrm{O}(n^2)$, and also that $\sum_{j=1}^{n} j \geq \sum_{j=n/2}^{n} n/2 = n^2/4$, which shows that the running time is $\Omega(n^2)$. Since the running time is both $\mathrm{O}(n^2)$ and $\Omega(n^2)$, this means that we have a asymptotically precise bound $\Theta(n^2)$.

Instead of doing the above calculations, we can appeal to that we have learned in class that $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$, and it is perfectly fine to use this as a known fact without proof (as long as there is an explanation that this is what is happening).

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).

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

Solution: From our analysis of the algorithm above, we know that $A[j] = \sum_{i=1}^{j} i^2$. But this means that we can compute $A[j]$ as

$$A[j] = \sum_{i=1}^{j-1} i^2 + j^2 = A[j-1] + j^2 \ . \tag{1}$$

Therefore, the snippet of code

```
A[1] := 1
j    := 2
while (j <= n) {
    A[j] := A[j-1] + j * j
    j := j + 1
}
```

will compute the same result. This code goes through the array only once and does a constant amount of work per array element $A[j]$ (assuming that integer arithmetic is constant-time, which is a simplifying assumption that we make in this course). Therefore, the total running time is $O(n)$. Since we need to store n values in the array A, it is clear that linear time is optimal.

4 (70 p) In the following snippet of code `A` and `B` are arrays indexed from 1 to n that contain numbers.

```
j    := n
good := TRUE
while (j >= 1 and good) {
    s := A[j]
    for i := j - 1 downto 1 {
        s := s + A[i]
    }
    s := s / j
    if (s > B[j]) {
        good := FALSE
    }
    else
    {
        j := j - 1
    }
}
return good
```

4a (30 p) Explain in plain language what the algorithm above does. In particular, when does the algorithm return `TRUE` or `FALSE` and why?

Solution: Cutting straight to the chase, this algorithm checks if the average (i.e., arithmetic mean) of the j first numbers in A is less than or equal to the number stored in $B[j]$. If this is the case for all $j \in [n]$, then `TRUE` is returned, and otherwise the algorithm returns `FALSE`.

In a bit more detail, we have an outer while loop in which j goes from n down to 1, except that the outer loop is terminated if `good`, which is initialized to `TRUE`, is ever assigned `FALSE`.

For every value of j in the outer loop, we have an inner for loop that computes $s = \sum_{i=1}^{j} A[i]$ (but in decreasing order of the index i). After the end of the for loop this value is divided by j, so that the final result is $s = \frac{1}{j}\sum_{i=1}^{j} A[i]$, i.e., the arithmetic mean of the numbers $A[1], A[2], \ldots, A[j]$. If it holds that $s > B[j]$, then `good` is set to `FALSE`, which will make the while loop terminate so that the algorithm returns `FALSE`. Otherwise j will be decremented, and the next while loop will then compare $\frac{1}{j-1}\sum_{i=1}^{j-1} A[i]$ to $B[j-1]$, and so on. If for every $j \in [n]$ it holds that $\frac{1}{j}\sum_{i=1}^{j-1} A[i] \leq B[j]$, then `good` will keep the value `TRUE`, which is what the algorithm will finally return.

4b (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 loops. The outer while loop runs for j from n down to 1 in the worst case (namely, when all the checks are successful). The inner for loop runs from j down to 1. Before the while loop starts, and outside of the inner for loop, there is a constant number of operations. The asymptotic time complexity is therefore determined by the total number of times the inner for loop is executed.

It follows from what is written above that the inner loop will run a total of $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \Theta\big(n^2\big)$ times in the worst case, and so the the time complexity of the algorithm is $\Theta\big(n^2\big)$. This analysis is very similar to that in Problem 3.

Just as for Problem 3, it is the case that 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).

4c (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 except for improvements in lower-order terms.)

Solution: From our analysis of the algorithm above, it follows that we can just sum up the elements in the array A as we go and for every step compare this sum to $c \cdot B[j]$ (since multiplying both sides of the comparison by j does not change the outcome). Therefore, the snippet of code

```
j    := 1
sum  := 0
good := TRUE
while (j <= n and good) {
    sum := sum + A[j]
    if (sum > j * B[j]) {
        good := FALSE
    }
    else
    {
        j := j + 1
    }
}
return good
```

will give exactly the same result. This piece of code goes through the array only once, thus having a running time of $\mathrm{O}(n)$. Since we need to check n values, namely for all numbers $B[1], B[2], \ldots, B[n]$, it is not possible to run faster than in linear time, so this is asymptotically optimal.