Warm-up

Problem 1. Sort the following functions in increasing order of asymptotic growth

$$n, n^3, n \log n, n^n, \frac{3^n}{n^2}, n!, \sqrt{n}, 2^n$$

Solution 1.

$$\sqrt{n}, n, n \log n, n^3, 2^n, \frac{3^n}{n^2}, n!, n^n$$

Problem 2. Sort the following functions in increasing order of asymptotic growth

$$\log \log n, \log n!, 2^{\log \log n}, n^{\frac{1}{\log n}}$$

Solution 2.

$$n^{\frac{1}{\log n}}, \log \log n, 2^{\log \log n}, \log n!$$

Problem 3. Consider the following pseudocode fragment.

```
1: function printFive(n)
2:     for i ← 1; i ≤ 5; i++ do
3:         Print a single character 'n'
```

Using the O-notation, upperbound the running time of printFive.

Solution 3. Printing a single character takes constant time. The for-loop is always executed exactly five times, so the running time is therefore

$$\sum_{i=1}^{5} O(1) = 5 \cdot O(1) = O(1).$$

Problem 4. Consider the following pseudocode fragment.

```
1: function stars(n)
2:     for i ← 1; i ≤ n; i++ do
3:         Print '*' i times
```

a) Using the O-notation, upperbound the running time of stars.

b) Using the Ω-notation, lowerbound the running time of stars to show that your upperbound is in fact asymptotically tight.

Solution 4.

a) The first iteration prints 1 star, second prints two, third prints three and so on. The total number of stars is $1 + 2 + \cdots + n$, namely,

$$\sum_{j=1}^{n} j \leq \sum_{j=1}^{n} n = n^2 = O(n^2).$$

b) Assume for simplicity that n is even. To lowerbound the running time, we consider only the number of stars printed during the last $\frac{n}{2}$ iterations. Since this is part of the full execution, analyzing only this part gives a lower bound on the total running time. The main observation we need is that for each of the considered iterations, we print at least $n/2$ stars, allowing us to lower bound the total number of stars printed:

$$\sum_{j=1}^{n} j \geq \sum_{j=n/2+1}^{n} \frac{n}{2} = \frac{n^2}{4} = \Omega(n^2).$$

Problem 5. Recall the problem we covered in lecture: Given an array A with n entries, find $0 \leq i \leq j < n$ maximizing $A[i] + \cdots + A[j]$.

Prove that the following algorithm is incorrect: Compute the array B as described in the lectures. Find i minimizing $B[i]$, find j maximizing $B[j+1]$, return (i, j).

Come up with the smallest example possible where the proposed algorithm fails.

Solution 5. Let $A = [1, -2]$, so $B = [0, 1, -1]$. The algorithm return $i = 2$ and $j = 0$. Which does not even obey $i \leq j$.

Problem solving

Problem 6. Given an array A consisting of n integers, we want to compute the upper triangle matrix C where

$$C[i][j] = \frac{A[i] + A[i+1] + \cdots + A[j]}{j - i + 1}$$

for $0 \leq i \leq j < n$. Consider the following algorithm for computing C:

```
1: function SUMMING_UP(A)
2:     C ← new matrix of size(A) by size(A)
3:     for i ← 0; i < n; i++ do
4:         for j ← i; j < n; j++ do
5:             Compute average of entries A[i : j]
6:             Store result in C[i, j]
7:     return C
```

a) Using the O-notation, upperbound the running time of SUMMING-UP.

b) Using the Ω-notation, lowerbound the running time of SUMMING-UP.

Solution 6.

a) The number of iterations is $n + n - 1 + \cdots + 1 = \binom{n+1}{2}$, which is bounded by n^2. In the iteration corresponding to indices (i, j) we need to scan $j - i + 1$ entries from A, so it takes $O(j - i + 1) = O(n)$. Thus, the overall time is $O(n^3)$.

b) In an implementation of this algorithm, Line 5 would be computed with a for loop; when $i < \frac{1}{4}n$ and $j > \frac{3}{4}n$, this loop would iterate least $n/2$ times, which takes $\Omega(n)$ time. There are $n^2/16$ pairs (i, j) of this kind, which is $\Omega(n^2)$. Thus, the overall time is $\Omega(n^3)$.

Problem 7. Come up with a more efficient algorithm for computing the above matrix $C[i][j] = \frac{A[i]+A[i+1]+\cdots+A[j]}{j-i+1}$ for $0 \leq i \leq j < n$. Your algorithm should run in $O(n^2)$ time.

Solution 7. The idea is very simple, suppose we had at our disposal an array B whose ith entry is the sum of the entries 0 through $i - 1$ of A; in other words,

$$B[i] = \sum_{k=0}^{i-1} A[k].$$

Then $C[i][j]$ is simply $\frac{B[j+1]-B[i]}{j-i+1}$. If we can compute B in $O(n^2)$ time we are done. In fact, we can compute B is just $O(n)$ time.

```
1: function SUMMING_UP_FAST(A)
2:     B[0] ← 0
3:     for i ← 1; i < n; i++ do
4:         B[i] ← B[i − 1] + A[i − 1]
5:     for i ← 0; i < n; i++ do
6:         for j ← i; i < n; j++ do
7:             C[i][j] ← (B[j + 1] − B[i])/(j − i + 1)
8:     return C
```

The correctness of the algorithm is clear since

$$C[i][j] = \frac{B[j+1] - B[i]}{j - i + 1} = \frac{\sum_{k=0}^{j} A[k] - \sum_{k=0}^{i-1} A[k]}{j - i + 1} = \frac{\sum_{k=i}^{j} A[k]}{j - i + 1}.$$

as desired.

For the time complexity, we note that the for loop in Line 3 runs in $O(n)$ time and the nested for loops starting in Line 5 runs in $O(n^2)$ time, yielding the desired overall complexity.

Problem 8. Give a formal proof of the transitivity of the O-notation. That is, for function f, g, and h show that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.

Solution 8. Since $f = O(g)$, it follows that there exists $n_0 > 0$ and $c > 0$ such that $f(n) \leq cg(n)$ for all $n > n_0$. Since $g = O(h)$, it follows that there exists $n_0' > 0$ and $c' > 0$ such that $g(n) \leq c'h(n)$ for all $n > n_0$.

It follows that for all $n > \max(n_0, n_0')$ we have

$$f(n) \leq c\, g(n) \leq c\, c'\, h(n).$$

Thus, if we define $n_0'' = \max(n_0, n_0')$ and $c'' = c\, c'$, the above inequality means $f = O(h)$.

Problem 9. Using O-notation, show that the follow snippet of code runs in $O(n^2)$ time. As an extra challenge, it can be shown that it actually runs in $O(n)$ time.

```
1: function ALGORITHM(n)
2:     j ← 0
3:     for i ← 0; i < n; i++ do
4:         while j ≥ 1 and [condition that can be checked in O(1) time] do
5:             j ← j − 1
6:         j ← j + 1
7:     return j
```

Solution 9. Showing $O(n^2)$ time: Line 2 and 5-7 take $O(1)$ time each, as they contain only assignments and basic mathematical operations. Checking the condition of the while-loop on line 4 takes $O(1)$ time as well. Since the while-loop checks for j being positive and the body of the loop decrements j, the loop can be executed at most j times. Since j starts out at 0 and is incremented once in the for-loop, we get that $j \leq n$. Hence, the while-loop on line 4 takes at most $O(n)$ time. As the while loop is inside the for-loop on line 3, which is executed n times, the total time of $O(n^2)$ follows.

Showing $O(n)$ time: To improve the analysis to $O(n)$ time, we need to observe that while j can indeed be decremented a large number of times (depending on the exact condition checked in the while-loop), it can still be decremented at most n times over the entire execution of the algorithm. This is because in every iteration of the for-loop j is incremented once and since the while-loop check for j being positive the while-loop can't execute more than n times over the entire execution of the algorithm. Hence, our running time is actually $O(n)$ time for line 4-5 plus $O(n)$ time for line 3 and 6, instead of the multiplication we used above.

Why does the while-loop contain an arbitrary condition that can be checked in constant time? This algorithm is one of the key components of the Knuth-Morris-Pratt string-matching algorithm, the first linear-time string matching algorithm, which you'll see in COMP2022/2922. That algorithm uses an extra condition that can be checked in constant time, as it needs to check more conditions, but those conditions can checked in constant time.

Problem 10. Given a string (which you can see as an array of characters) of length n, determine whether the string contains k identical consecutive characters. You can assume that $2 \leq k \leq n$. For example, when we take $k = 3$, the string "abaaab" contains three consecutive a's and thus we should return true, while the string "abaaba" doesn't contain three consecutive characters that are the same.

Your task is to design an algorithm that always correctly returns whether the input string contains k identical consecutive characters. Your solution should run in $O(n)$ time. In particular, k isn't a constant and your running time shouldn't depend on it.

Solution 10. We will loop through the string while maintaining two variables: *last* stores the last character we've seen so far and *count* stores how many consecutive characters before and including *last* are the same as *last*. Initially, we set *last* to the first character of the string and *count* to 1. In every iteration of the loop, we check if the next character is the same as *last*. If so, we increment *count* and once *count* equals k, we return true. If not, we update *last* to be the character that we just processed and reset *count* to 1. If we manage to fully scan the string without outputting true along the way, we return false.

In pseudocode this algorithm could look something like this.

```
1:  function TEST_THREE_CONSECUTIVE(S, k)
2:      last ← S[0]
3:      count ← 1
4:      for i ← 1; i < n; i++ do
5:          if S[i] = last then
6:              count ← count + 1
7:              if count = k then
8:                  return true
9:          else
10:             last ← S[i]
11:             count ← 1
12:     return false
```

To show that this algorithm is correct, we focus on showing that *last* and *count* correctly store the last processed character and the number of identical consecutive characters before and including *last*. This is the invariant of the algorithm. We start by showing that the invariant holds initially: by setting *last* to $S[0]$ and *count* to 1, we ensure that after processing the first character, *last* is indeed set to the last (and only) processed character and *count* stores the number of identical consecutive characters before and including *last*: there are no characters before *last*, so there's only one consecutive character which is equal to *last* itself.

Next we show that the invariant is maintained after processing the next character. In order to do so, we assume that the invariant holds before we process this character (making this essentially an inductive argument). Now let's see what happens when we process the next character. We'll first consider the case where $S[i]$ doesn't match *last*. In this case we failed to find k consecutive characters that are identical and since $S[i]$ differs from last, the next substring that we should consider starts from $S[i]$ and thus currently we have seen 1 consecutive identical characters. Hence, by setting *last* to $S[i]$ and *count* to 1, we maintain the invariants. If *last* and $S[i]$ are equal, we can extend our current substring of identical characters using $S[i]$ and hence its length increases by 1. Hence, we don't have to update *last*, but we do need to increment *count*.

How does this help us to prove the correctness of the algorithm? Since we maintain this invariant, we know that at any point in the algorithm we have the correct length of the longest substring of identical characters that includes the last character. Hence, when we check if this length is k, we know whether we have found a substring of sufficient length. Hence, when we return true, we know we indeed found a substring satisfying the condition. On the other hand, when we return false, we know that we scanned the entire string and at no point did we have a substring of identical characters of length k (since we check for this every time we extend our substring by one character). Hence, when we return false there indeed was no substring satisfying our condition. Together these two statements imply the correctness of our algorithm.

Analyzing the running time of this algorithm is fairly straightforward: line 2-3 and 5-12 all consist of (a constant number of) assignments, simple comparisons, and/or simple math operations and the return statements return booleans, all of which takes $O(1)$ time. The loop on lines 4-11 loops through (at most) the remaining $n-1$ characters of the string, performing $O(1)$ time operations each time. Hence, the loop takes at most $(n-1) \cdot O(1)$ or $O(n)$ time. Since we saw that the operations outside the loop all take constant time, the loop dominates the running time, which comes down to $O(n)$ time in total, as required.

Problem 11. Given an array with n integer values, we would like to know if there are any duplicates in the array. Design an algorithm for this task and analyze its time complexity.

Solution 11. The straightforward solution is to do a double for loop over the entries of the array returning "found duplicates" right away when we find a pair of identical elements, and "found no duplicates" at the end if we exit the for loops.

This algorithm is correct, since we compare every pair of elements, thus if there exists a duplicate pair, we consider this pair at some point in the execution. Similarly, if no duplicates exist, we can safely return this after checking all pairs.

The complexity of this solution is $O(n^2)$, since there are $O(n^2)$ pairs of elements to consider and performing a single such comparison takes $O(1)$ time.

A better solution is to sort the elements and then do a linear time scan testing adjacent positions. The correctness follows from the fact that in a sorted array, duplicate integers have to occur in consecutive indices. As we shall see later in class one can sort an array of length n in $O(n \log n)$ time, which also dominates the running time of this algorithm, as scanning through n elements and performing a single comparison for consecutive integers takes $O(1)$ time per pair and $O(n)$ time in total.