Exercise Session 02

Solve the following exercises. The exercises that are more involved are marked with a star. If you need some guidance for solving the exercise, place your trash bin in front of your group room's door and the first available teaching assistant will come to help you out.

Exercise 1.

Solve exercises CLRS 3.1–1, and 3.1–4

Remark: A function $f(n)$ is asymptotically nonnegative if there exists n_0 such that $f(n) \geq 0$ for all $n \geq n_0$.

Solution 1.

CLRS 3.1–1 Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

We have to prove that there exists $c_1, c_2 > 0$, and n_0 such that $0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$ for all $n \geq n_0$. We know that $f(n)$ and $g(n)$ are asymptotically nonnegative, then we can choose $n_0 > 0$ such that $f(n) \geq 0$ and $g(n) \geq 0$ for all $n \geq n_0$. Let $c_1 = 1/2$, $c_2 = 1$, and $n \geq n_0$. On the one hand, we have that $\max(f(n), g(n)) \leq f(n) + g(n)$ because the max of two nonnegative numbers is always smaller than their sum. On the other hand, we have $1/2(f(n) + g(n)) \leq \max(f(n), g(n))$ because the average of two numbers is always less than or equal to the maximum of the two.

CLRS 3.1–4 Is $2^{n+1} = O(2^n)$? Yes, because $2^{n+1} = 2 \cdot 2^n = O(2^n)$.

Is $2^{2n} = O(2^n)$? We show that $2^{2n} \neq O(2^n)$. Towards a contradiction, assume that $2^{2n} = O(2^n)$. By definition of big-O notation, there exist $c > 0$ and $n_0 > 0$ such that $0 \leq 2^{2n} \leq c2^n$ for all $n \geq n_0$. Since $2^{2n} = 2^n \cdot 2^n$ we must have $2^n \leq c$ for all $n > n_0$. This cannot be true because c is a constant and 2^n diverges for n that goes to infinity. The correct asymptotic class for 2^{2n} is $2^{2n} = (2^2)^n = O(4^n)$.

Exercise 2.

Consider the algorithm SumUpTo that takes as input a natural number $n \in \mathbb{N}$.

```
SumUpTo(n)
1  s = 0
2  for i = 1 to n
3        s = s + i
4  return s
```

Prove that given $n \in \mathbb{N}$, SumUpTo terminates and returns $\frac{n(n+1)}{2}$.

Solution 2.

We prove the following loop invariant: Before each iteration of the outer loop $s = \frac{(i-1)i}{2}$.

Initialisation: Before the first iteration $i = 1$ and $s = 0$. Substituting i with 1 we obtain $\frac{(i-1)i}{2} = 0$.

Maintenance: Before executing line 3 we have that $s = \frac{(i-1)i}{2}$, thus after the assignment we have

$$s = \frac{(i-1)i}{2} + i = \frac{i^2 - i + 2i}{2} = \frac{i^2 + i}{2} = \frac{i(i+1)}{2}.$$

Therefore, incrementing i for the next iteration preserves the invariant.

Termination: The loop terminates when $i > n$, that is when $i = n + 1$. By substituting i with $n + 1$ in the invariant, we obtain $s = \frac{n(n+1)}{2}$.

Therefore the function SumUpTo returns $\frac{n(n+1)}{2}$. Please note what we have just rephrased the classic proof of the arithmetic series $\sum_{i=0}^{n} i = n(n+1)/2$ as the proof of correctness of SumUpTo(n). This is to highlight the close relation between loop invariants and proofs by induction.

Exercise 3.

By getting rid of the asymptotically insignificant parts on the expressions, give a simplified asymptotic tight bounds (big-theta notation) for the following functions in n. Here, $k \geq 1$, $e > 0$ and $c > 1$ are constants.

(a) $0.001n^2 + 70000n$

(b) $2^n + n^{10000}$

(c) $n^k + c^n$

(d) $\log^k n + n^e$

(e) $2^n + 2^{n/2}$

(f) $n^{\log c} + c^{\log n}$ (hint: look at some properties of the logarithm at page 56 in CLRS)

Solution 3.

(a) $0.001n^2 + 70000n = \Theta(n^2)$

(b) $2^n + n^{10000} = \Theta(2^n)$

(c) $n^k + c^n = \Theta(c^n)$, because any exponential function dominates any polynomial functions.

(d) $\log^k n + n^e = \Theta(n^e)$ Look at page 57 of CLRS for the comparison of polylogarithmic and polynomial functions.

(e) $2^n + 2^{n/2} = \Theta(2^n)$. Here note that $2^{n/2} = (\sqrt{2})^n$ and $2 > \sqrt{2} \cong 1.4$ therefore the first term dominates over the second.

(f) $n^{\log c} + c^{\log n} = 2 \cdot n^{\log c} = \Theta(n^{\log c})$ because $n^{\log c} = c^{\log n}$. For the same reason we also have $n^{\log c} + c^{\log n} = 2 \cdot c^{\log n} = \Theta(c^{\log n})$.

★ **Exercise 4.**

Consider the following algorithm that takes an array $A[1\,..\,n]$ and rearrange its elements in nondecreasing order.

Sort(A)

```
1  for i = 1 to A.length
2      for j = i + 1 to A.length
3          if A[i] > A[j]
4              key = A[i]
5              A[i] = A[j]
6              A[j] = key
```

(a) Try Sort(A) on the the instance $A = [4, 2, 8, 7, 1]$. Explain in your words how the algorithms works in general;

(b) Prove that Sort solves the sorting problem (hint: determine suitable invariants for both loops);

(c) Determine the asymptotic worst-case running time using the Θ notation.

Solution 4.

(a) Informally, the algorithm may be explained using the analogy of sorting a hand of playing cards. We start with an empty left hand and the other cards face up on the table. Then we select the smallest card in the table an insert it as the rightmost card in the left hand. To find the smallest card in the table we select the leftmost card in the table and we proceed to the right. Every time we encounter a card which is smaller than the first card in the table, we swap the two and continue this way until all cards in the table have been checked. The sub-array $A[1\,..\,i-1]$ constitutes that sorted hand, the sub-array $A[i\,..\,n]$ corresponds to the pile of cards still on the table.

(b) Invariant of the inner for-loop: $A[i]$ is the smallest element of the sub-array $A[i\,..\,j-1]$ and $A[i\,..\,j-1]$ is a reordering of the elements that where originally in $A[i\,..\,j-1]$ before executing the loop.

Initialisation: We show that the loop invariant holds true before the first loop iteration. Before running the inner loop $j = i+1$, therefore $A[i\,..\,j-1] = A[i]$ which is in fact the smallest element and it is its original position before executing the loop.

Maintenance: By hypothesis we know that $A[i]$ is the smallest element of the sub-array $A[i\,..\,j-1]$. If $A[j] < A[i]$ the two elements are swapped. At this point $A[i]$ contains the smallest element in the sub-array $A[i\,..\,j]$, and $A[i\,..\,j]$ is a reordering of the elements that were originally in $A[i\,..\,j-1]$ before executing the loop. Incrementing j for the next iteration preserves the loop invariant.

Termination: The for-loop terminates when $j > A.length$, that is $j = n+1$. By substituting $n+1$ for j in the invariant we obtain that $A[i]$ is the smallest element of the sub-array $A[i\,..\,n]$ and $A[i\,..\,n]$ is a reordering of the elements that where originally in $A[i\,..\,n]$ before executing the loop.

Now that we have proved the above invariant for the inner loop, we can use it to prove the next invariant relative to the outer loop. Before each iteration of the outer loop the sub-array $A[1\,..\,i-1]$ is sorted, all the elements of the sub-array $A[i\,..\,n]$ are greater than or equal to any element of $A[1\,..\,i-1]$, and A is a reordering of the elements that were originally in the array A.

Initialisation: Before the first outer loop iteration $i = 1$, therefore $A[i\,..\,i-1] = [\,]$ an empty array and $A[i\,..\,n] = A[1\,..\,n]$. Therefore $A[i\,..\,i-1]$ is trivially sorted and the fact that for all element in $a \in A[i\,..\,i-1]$ and all element in $b \in A[i\,..\,n]$, $a \leq b$ trivially holds because there is no element a to compare. Moreover, A at this point has never been modified.

Maintenance: By hypothesis we know that $A[1\,..\,i-1]$ is sorted and all the elements in $A[i\,..\,n]$ are greater than or equal to any other element of $A[1\,..\,i-1]$. As proven above, at the termination of the inner loop $A[i]$ is the smallest element of the sub-array $A[i\,..\,n]$ and $A[i\,..\,n]$ is a reordering of the elements that where originally in $A[i\,..\,n]$ before executing the loop. Hence, incrementing i for the next iteration preserves the invariant.

Termination: The outer for loop terminates when $i > A.length$, that is $i = n+1$ and the invariant reads as follows. The sub-array $A[1\,..\,n]$ is sorted and it is a reordering of the elements that were originally in the array A.

This concludes the proof of correctness for the procedure SORT.

(c) The worst-case running time occurs when the body of the if-then inside the inner for loop is always executed. This occurs when the array $A[1\,..\,n]$ is such that $A[1] > A[2] > \cdots > A[n]$.

Let c_1 and c_2 be the time it takes for executing line 1 and 2 respectively, and $c_{3\text{-}6}$ the time it takes for executing the entire if construct (lines 3–6). Note that the number of iterations of the inner for-loop during the i-th iteration of the outer for loop is $n-(i+1)$ (as usual the condition of the for-loop in line 2 is executed one more time i.e., $n-i$).

Then the exact worst-case running time of SORT is

$$\begin{aligned}
T(n) &= c_1(n+1) + c_2\sum_{i=1}^{n}(n-i) + c_{3\text{-}6}\sum_{i=1}^{n}(n-i-1) \\
&= c_1(n+1) + c_2(n^2 - \sum_{i=1}^{n} i) + c_{3\text{-}6}(n^2 - n - \sum_{i=1}^{n} i) \\
&= c_1(n+1) + c_2\frac{n^2-n}{2} + c_{3\text{-}6}\frac{n^2-3n}{2} && \text{(arithmetic series)} \\
&= \frac{c_2 + c_{3\text{-}6}}{2}n^2 + \frac{2c_1 - c_2 - 3c_{3\text{-}6}}{2}n + c_1
\end{aligned}$$

It is clear from the above equation that $T(n) = \Theta(n^2)$.

A simpler way to perform the asymptotic worst-case runtime analysis can be done as follows. Note that both for loops iterate a number of time which is linear in the length of A. If $A[1\,..\,n]$ if satisfies $A[1] > A[2] > \cdots > A[n]$ then lines 3–6 are executed every iteration of both for loops. Lines 3–6 take constant time. Therefore we have that $T(n) = \Theta(n) \cdot \Theta(n) \cdot \Theta(1) = \Theta(n^2)$.