Assignment 1

Carlo Theunissen (1353926) ()
c.f.theunissen@student.tue.nl

September 29, 2018

Exercise 5

We use Induction to prove for every $n \geq 1 \implies \sum_{i=1}^{n}(3i-2)(i-1) = n^3 - n^2$

Base case:

$$\begin{aligned} \sum_{i=1}^{n}(3i-2)(i-1) &= n^3 - n^2 \\ \sum_{i=1}^{1}(3i-2)(i-1) &= 1^3 - 1^2 \\ (3-2)(1-1) &= 1-1 \\ (1)(0) &= 1-1 \\ 0 &= 0 \end{aligned} \tag{1}$$

Inductive step:
Let n ≥ 1 and assume the claim holds for n. The IH is:
$\sum_{i=1}^{n}(3i-2)(i-1) = n^3 - n^2$

We prove $n+1$

$$\begin{aligned} \sum_{i=1}^{n+1}(3i-2)(i-1) &= (\sum_{i=1}^{n}(3i-2)(i-1)) + (3(n+1)-2)((n+1)-1) \\ &\stackrel{\text{IH}}{=} n^3 - n^2 + (3n+1)(n) \\ &= n^3 - n^2 + 3n^2 + n \\ &= n^3 + 2n^2 + n \\ &= n(n+1)^2 \\ &= (n+1)^3 - (n+1)^2 \end{aligned} \tag{2}$$

With both the base case and inductive step you can see that claim holds

Exercise 6

(a) We use The Master theorem to calculate the upper- and lower bounds for:

$$T(n) = 4T(n/2) + n \log n$$

$$a = 4, b = 2, f(n) = n \log n$$

$\log_b a = log_2 4 = 2$

The theorem gives us with case 1: $O(n^{2-\epsilon})$ for some constant $\epsilon > 0$. Therefore we have $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

(b) We use The Master theorem to calculate the upper- and lower bounds for:

$$T(n) = 2T(n/4) + n$$

$$a = 2, b = 4, f(n) = n$$

$\log_b a = log_4 2 = \frac{log_2 2}{log_2 4} = \frac{1}{2}$

Case 3 gives us $\Theta(n)$ when the *regularity condition* holds.
Take $c = \frac{1}{4}$ and $n_0 = 1$
$af(\frac{n}{b}) = 2(\frac{n}{4}) = \frac{n}{2} \leq \frac{1}{4}n = cf(n)$
for $n \geq n_0$
Hereby we can conclude $T(n) = \Theta(n)$

(c) We use The Master theorem to calculate the upper- and lower bounds for:

$$T(n) = 8T(n/4) + n\sqrt{n}$$

$$a = 8, b = 4, f(n) = n\sqrt{n}$$

$\log_b a = log_4 8 = \frac{log_2 8}{log_2 4} = \frac{3}{2}$

Case 2 gives us $\Theta(n^{\log_b a} \log n) = \Theta(n^{\frac{3}{2}} \log n)$

(d)

Exercise 7

Loop Invariant: At the start of i-th iteration of the loop. $A[n-i+1, n]$ is a copy of $A[1, i]$ in reverse order.

Initialization: At the start of the 1-st loop the loop invariant states: 'At the start of the 1-st iteration of the loop, $A[n, n]$ is a copy of $A[1, 1]$. However, $A[n, n]$ and $A[1, 1]$ are both empty arrays so the *Loop Invariant* holds.

Maintenance: Assume the *Loop Invariant* holds for $A[n-i+1, n]$. In the body we add the $A[i]$ element at the $A[n+1-i]$ place. Thus at the start of iteration i+1, the *Loop Invariant* holds. Which is what we needed to prove

Termination: When the loop terminates at the $n/2$-th iteration, $A[n/2+1, n]$ holds a copy of $A[1, n/2]$ in reverse order. This is indeed what we expected. The algorithm is correct

Exercise 8

(a) Prove or disprove $n^3 - 3n^2 - n - 1 = \Theta(n^3)$
claim: $n^3 - 3n^2 - n - 1 = \Theta(n^3)$

Proof. By definition of Θ: There exist a c_1, c_2, n_0 with $c_1, c_2 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

$$c_1 = \frac{1}{2}, c_2 = 1, n_0 = 10$$

$$\begin{aligned} c_1 n^3 &= \frac{1}{2} n^3 \\ &= n(\frac{1}{2} n^2) \\ &\leq n(n^2 - 3n - 1) \\ &\leq n(n^2) \\ &= n^3 \\ &= c_2 n^3 \end{aligned} \tag{3}$$

Please note: $n(\frac{1}{2}n^2) \leq n(n^2 - 3n - 1)$ with $n \geq 10$ □

(b) Prove or disprove $n + \log n = \Omega(n \log n)$
claim: $n + \log n = \Omega(n \log n)$

Proof. Lets assume the claim holds.
By definition of Ω states "There exists a c with $c \geq 0$ such that $cg(n) \leq f(n)$"

We that know that $n \log n$ grows faster than every linear function. $n + \log n$ is almost linear. Therefore we can find a c and n for which $cn \log n \geq n + \log n$.
We have reached a contradiction and hence our assumption, and thereby the claim, must be false.

□

(c) Prove or disprove $n^2 \log n = O(n^2)$
Claim: $n^2 \log n = O(n^2)$

Proof. Lets assume the claim holds.
By definition of O we have a: c, n_0 for which $c > 0$ and $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

We know that $n \log n \geq n$, with a simple calculation we can also conclude $n^2 \log n \geq n^2$ for $n_0 \geq 0$:

$$\begin{aligned} n \log n &\geq n \\ n(n \log n) &\geq n(n) \\ n^2 \log n &\geq n^2 \end{aligned} \tag{4}$$

We can find a c and n_0 for which $cn^2 \log n \geq n^2$ therefore we have found a contradiction.
The only conclusion we can make is that the assumption is false. Hence, the original claim must also be false. □

(d) Prove or disprove $n\sqrt{n} = \Omega(n \log^2 n)$

Proof. By definition of Ω we know there exist a c, n_0 for which $cg(n) \leq f(n)$ for all $n \geq n_0$
let $c = 1$ and $n = 1$

$$\begin{aligned} cn \log^2 n &= n \log^2 n \\ &= \log^2(n^{\sqrt{n}}) \\ &\leq n^{\frac{3}{2}} \\ &= n\sqrt{n} \end{aligned} \tag{5}$$

Because any logarithmic function is smaller than any exponential function we know $\log^2(n^{\sqrt{n}}) \leq n\sqrt{n}$ for $n_0 \geq 0$
There exist a constant $c = 1$ for which the statement holds true. Hence, the claim is true by definition □

(e) Prove or disprove $(f(n) = O(g(n)) \wedge g(n) = O(h(n))) \implies (f(n) = O(h(n)))$.

Proof. Assume $f(n) = O(g(n)) \wedge g(n) = O(h(n))$ holds.
Then by definition of O we have a c_1, c_2, n for which $f(n) \leq c_1 g(n)$ and $g(n) \leq c_2 h(n)$ holds for all $n \geq n_0$

$$\begin{aligned} g(n) &\leq c_2 h(n) \\ c_1 g(n) &\leq c_1 c_2 h(n) \end{aligned} \tag{6}$$

Because of transitivity of $f(n) \leq c_1 g(n)$ and $c_1 g(n) \leq c_1 c_2 h(n)$ we can conclude $f(n) \leq c_1 c_2 h(n)$

let $c_3 = c_1 c_2$
Thereby we have: $f(n) \leq c_3 h(n)$ for all $n \geq n_0$

Thus, there exist a constant, namely c_3, which makes the proposition true. Hence, the statement is true by definition of O □

Exercise 9

(a)

```
   PrintDivisibleNumbers(A, p)
   Input: an array A of n natural numbers and a number p
1  zeroRemainer = []
2  oneRemainer = []
3  twoRemainer = []
4  for i ← 1 to A.length do
5      if A[i] % 3 == 0 then
6          zeroRemainer.length = zeroRemainer.length + 1;
7          zeroRemainer[zeroRemainer.length] = A[i];
8      if A[i] % 3 == 1 then
9          oneRemainer.length = oneRemainer.length + 1;
10         oneRemainer[oneRemainer.length] = A[i];
11     if A[i] % 3 == 2 then
12         twoRemainer.length = twoRemainer.length + 1;
13         twoRemainer[twoRemainer.length] = A[i];
14 Build-Min-Heap(zeroRemainer)
15 Build-Min-Heap(oneRemainer)
16 Build-Min-Heap(twoRemainer)
17 for j ← 1 to k do
18     print(zeroRemainer.Extract-Min());
19     print(zeroRemainer.Extract-Min());
20     print(twoRemainer.Extract-Min());
```

(b) **Line 1-3:** each line gives O(1) because creating an Array cost O(n+1) and n is in this situation is 0.
Line 4-13: $\sum_{i=1}^{n} O(5) = O(5n)$ because every natural number divided by three will give as the remainder: 0, 1 or 2. Therefore only 5 lines in the for loop are being executed.
Line 14-16: Each line gives $O(n)$ by definition of Build-Min-Heap
Line 17-20 $\sum_{i=1}^{k} O(3\log n) = O(3k\log n)$ because every iteration 3 lines with O($\log n$) are being executed.

If you add these together you will get: $3 + 5n + 3n + 3k\log n$

Claim: $3 + 5n + 3n + 3k\log n = O(n + k\log(n))$

Proof. By definition of O we know there exist a c, n_0 for which $0 \leq$f(n) $\leq cg(n)$ for all $n \geq n_0$

$$c = 9, n_0 = 3, f(n) = 3 + 5n + 3n + 3k, g(n) = n + k\log(n)$$

$$\begin{aligned} f(n) &= 3 + 5n + 3n + 3k\log n \\ &= 3 + 8n + 3k\log n \\ &\leq 3 + 8(n + k\log n) \\ &\leq 9(n + k\log n) \\ &= c(n + k\log n) \\ &= cg(n) \end{aligned} \tag{7}$$

□

(c) Loop Invariant: At the start of i-th iteration
zeroRemainer contains all numbers from A[1 : i] which divided by three give 0
oneRemainer contains all numbers from A[1 : i] which divided by three give 1
twoRemainer contains all numbers from A[1 : i] which divided by three give 2

Initialization: At the start of the 1-st loop the loop invariant states: 'At the start of the 1-st iteration of the loop,
zeroRemainer contains all numbers from A[1 : 1] which divided by three give 0
oneRemainer contains all numbers from A[1 : 1] which divided by three give 1
twoRemainer contains all numbers from A[1 : 1] which divided by three give 2
A[1:1] is an empty array, and *zeroRemainer*, *oneRemainer* and *twoRemainer* are all empty, hence our loop invarient holds.

Maintenance: Assume the *Loop Invariant* holds for at the start of i-th iteration. In the body we have three cases 1) A[i] divided by three gives 0. Then it is being added to the *zeroRemainer* array. 2) A[i] divided by three gives 1. Then it is being added to the *oneRemainer* array and 3) A[i] divided by three gives 2. Then it is being added to the *twoRemainer* array. Because every natural number gives when divided by three the remainer: 1,2 or 0. By case distinction the *Loop Invariant* holds.

Termination: When the loop terminates at i = A.length + 1. *zeroRemainer*, *oneRemainer* and *twoRemainer* contain the appropriate numbers for A[1, A.length] this is indeed expected, hence our algoritme is correct.

Exercise 10

(a) A A data structure that holds a collection of task can be an array. We use Min-heapify to sort the elements accordingly to their time estimate
B We add the element at the back of the array and swap it with his parent until the data structure is restored
C We extract the first element and place the last element at the front. Then we use Min-Heapify to restore the data structure
D Because we're using a Min-Heap to store the items. The first element in the array will always be the lowest one.

INSERT(T, T) will take $O(\log n)$ because our data structure looks really like a heap
NEXTTASK(T, T) will also take $O(\log n)$ because our data structure looks like a heap
MINTASK(T) the first element is the lowest one, returning the first element will cost $O(1)$

(b)

```
PRINTTOTALTASKS(A, k)
Input: an array A of 5 ∗ n tuples and a number k
1  minheaps ← new Array(5);
2  totalTasksDone ← 0;
3  for i ← 1 to 5 do
4      minheaps[i] = Build-Min-Heap(A[i]);
5  for i ← 1 to 5 do
6      totalTime ← 0;
7      for j ← 1 to minheaps[i].length do
8          item ← minheaps[i].NextTask();
9          if item.time + totalTime > k then
10             if i < 5 then
11                 minheaps[i + 1].Insert(item);
12         if item.time + totalTime ≤ k then
13             totalTime = totalTime + item.time;
14             totalTasksDone = totalTasksDone + 1;
15 print(totalTasksDone);
```

(c) **Line 1:** Creating an array with 5 elements will take O(6)
Line 2: Takes O(1)
Line 3-4: $\sum_{i=1}^{5} O(n) = O(5n)$ because creating a heap will cost n time. We are doing that 5 times and therefore the total cost is $O(5n)$
Line 5-14: $\sum_{i=1}^{5} O(1 + n + 4n + n\log n) = O(5 + 25n + 5n\log n)$
Line 6: takes O(1)
Line 7-14: $\sum_{j=1}^{n} O(4 + logn) = O(5n + n\log n)$ because in any case at most 4 lines with O(1) are executed
Line 11: takes $O(\log n)$ by definition of Insert.
Line 15: takes O(1)

The total running time is: $O(6 + 1 + 5n + 5 + 15n + 5n\log n + 1) = O(13 + 20n + 5n\log n)$

Claim: $13 + 20n + 5n\log n = O(n\log n)$

Proof. By definition of O we know there exist a c, n_0 for which $0 \leq$f(n)$\leq cg(n)$ for all $n \geq n_0$

$$c = 6, n_0 = 32, f(n) = 13 + 20n + 5n\log n, g(n) = n\log n$$

$$\begin{aligned} f(n) &= 13 + 20n + 5n\log n \\ &= 13 + n(20 + 5\log n) \\ &\leq n(21 + 5\log n) \\ &\leq 6n\log n) \\ &= c(n\log n) \\ &= cg(n) \end{aligned} \tag{8}$$

□

(d) **Loop Invariant:** At the start of i-th iteration of the loop *totalTasksDone* contains the sum of tasks that were able to be completed in the previous iterations. And the tasks that we were not able to complete were moved over to the current iteration

Initialization: At the start of the 1-st loop the loop invariant states: 'At the start of the 1-st iteration of the loop, *totalTasksDone* contains the sum of tasks that were able to be completed in the previous iterations. However, there weren't any previous iterations and therefore *totalTasksDone* is 0. Also, there weren't any tasks moved over, so the *Loop Invariant* holds

Maintenance: Assume the *Loop Invariant* holds for $i-1$. In the body we calculate the amount of tasks that can be done within the time k and add that to *totalTasksDone*. All remainder tasks are moved over to the next iteration. Thus at the start of iteration i+1, the *Loop Invariant* holds. Which is what we needed to prove

Termination: When the loop terminates all tasks that could be done were counted in *totalTasksDone*. ∎
This is indeed what we expected. The algorithm is correct