Mock In-class Test
COMS10007 Algorithms 2018/2019

Throughout this paper log() denotes the binary logarithm, i.e, $\log(n) = \log_2(n)$, and ln() denotes the logarithm to base e, i.e., $\ln(n) = \log_e(n)$.

1 O-notation

1. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a function. Define the set $\Theta(f(n))$.

Proof.

$$\begin{aligned}\Theta(f(n)) &= \{g(n) \ : \text{ There exist positive constants } c_1, c_2 \text{ and } n_0 \\ &\quad \text{s.t. } 0 \le c_1 f(n) \le g(n) \le c_2 f(n) \text{ for all } n \ge n_0\}\end{aligned}$$

□

2. Give a formal proof of the statement:

$$10\sqrt{n} \in O(n) \ .$$

Proof. We need to show that there are positive constants c, n_0 such that $10\sqrt{n} \le c \cdot n$, for every $n \ge n_0$. The previous inequality is equivalent to $\frac{10}{c} \le \sqrt{n}$, which in turn gives $\frac{100}{c^2} \le n$. Hence, we can pick $c = 1$ and $n_0 = \frac{100}{1^2} = 100$. □

3. Use the racetrack principle to prove the following statement:

$$n \in O(2^n) \ .$$

Hint: The following facts can be useful:

- The derivative of 2^n is $\ln(2)2^n$.
- $\frac{1}{2} \le \ln(2) \le 1$ holds.

Proof. We need to show that there are positive constants c, n_0 such that $n \le c \cdot 2^n$. We pick $c = 1$ and $n_0 = 1$. Observe that $n \le 2^n$ holds for $n = n_0 (= 1)$. It remains to show that $n \le 2^n$ also holds for every $n \ge n_0$. To show this, we use the racetrack principle. Observe that the derivative of n is 1 and the derivative of 2^n is $\ln(2)2^n$. Hence, by the racetrack principle it is enough to show that $1 \le \ln(2)2^n$ holds for every $n \ge n_0$, or $\log(\frac{1}{\ln 2}) \le n$. Since $\frac{1}{2} \le \ln(2) \le 1$, we have $1 \le \frac{1}{\ln(2)} \le 2$ and $0 \le \log(\frac{1}{\ln(2)}) \le 1$. Hence, $\log(\frac{1}{\ln 2}) \le n$ holds for every $n \ge 1 (= n_0)$, which thus proves $n \le 2^n$ for every $n \ge n_0$. □

2 Sorting

1. Why is Mergesort not an in-place sorting algorithm?

   Proof. This is a bookwork question. An in-place sorting algorithm is only allowed to use $O(1)$ memory in addition to the array that is to be sorted. In the combine step of Mergesort, it merges the sorted left and right halves of the input array. To this end, it first copies the left half to a new array B and the right half to a new array C. Arrays B and C are both of size $\Theta(n)$ in the initial call of Mergesort, which is not in $O(1)$. □

2. A divide-and-conquer algorithm consists of three parts: The divide, the conquer, and the combine phase. Compare Mergesort and Quicksort with regards to these three phases.

   Proof. This is a bookwork question. In Quicksort, the combine phase is trivial, i.e., there is nothing to do. In Mergesort, in the combine phase the sorted left and right halves need to be merged, which takes time $O(n)$. In mergesort, the divide phase is trivial and there is nothing to do. In Quicksort, the divide phase partitions the input elements around a pivot which takes time $O(n)$. The conquer phase is similar in both algorithms, i.e., we recursively sort a subarray. However, in merge-sort, subproblems are always balanced while in Quicksort, depending on the chosen pivot, we may have highly unbalanced subproblems. . . . □

3. What is the runtime (in Big-O notation) of Insertionsort when executed on the following arrays of lengths n: (no justification needed)

   (a) $1, 2, 3, 4, \dots, n-1, n$

   Proof. This is $\Theta(n)$. No explanation is required here.
   The reason for this is (and this is not a formal proof) that the inner loop always runs in time $O(1)$. Recall that in Insertionsort, the current element (determined by the outer loop) is placed at the correct position within the already sorted prefix array. On this input, the correct position however is the element's initial position, and the current element is therefore not moved at all. The loop thus stops immediately. □

   (b) $n, n-1, n-2, \dots, 2, 1$

   Proof. This is $\Theta(n^2)$. No explanation is required here.
   The reason for this is (and this is not a formal proof) that the current element (determined by the outer loop) is always placed at the leftmost position within the already sorted prefix array. For example, every element of the second half of the input is therefore moved at least $n/2$ steps to the left. Hence, for at least $n/2$ elements in the input, the elements move at least a distance of $n/2$. The runtime is therefore at least $n^2/4$ or $\Theta(n^2)$. □

3 Loop-Invariant

Consider the following algorithm:

Algorithm 1

```
Require: integer $n \geq 1$
1: $x \leftarrow 1$
2: for $i \leftarrow 1, \ldots, n-1$ do
3:     $x \leftarrow 2 \cdot x + 1$
4: end for
5: return $x$
```

The goal of this exercise is to show that this algorithm computes the value $2^n - 1$ on input n. Let x_i be the value of x at the beginning of iteration i (i.e., after i is updated in Line 2 and before Line 3 is executed). Consider the following loop invariant:

$$x_i = 2^i - 1$$

1. *Initialization:* Argue that at the beginning of the first iteration, i.e. when $i = 1$, the loop-invariant holds.

   Proof. When $i = 1$, the loop invariant gives us $x_1 = 2^1 - 1 = 1$. Observe that x is initialized before the loop with the value 1. The loop invariant thus holds at the beginning of the first iteration. □

2. *Maintenance:* Suppose that the loop invariant holds at the beginning of iteration i. Argue that the loop-invariant then also holds at the beginning of iteration $i + 1$.

   Proof. Suppose that the loop invariant holds at the beginning of iteration i. Observe that this means that the current value of x is $x_i = 2^i - 1$. In Line 3, we calculate $x = 2 \cdot x + 1$, and hence

   $$x_{i+1} = 2 \cdot x_i + 1 = 2 \cdot (2^i - 1) + 1 = 2^{i+1} - 1 \ .$$

   The loop invariant thus also holds at the beginning of iteration $i + 1$. □

3. *Termination:* Use the loop invariant to conclude that the algorithm indeed computes the value $2^n - 1$ on input n.

   Proof. At the end of the last iteration (i.e., when $i = n-1$), or before the nth iteration which is never executed, the loop invariant states that $x_n = 2^n - 1$. The algorithm thus outputs the value $2^n - 1$. □

4. What are the worst-case and best-case runtimes of the algorithm?

   Proof. The algorithm always runs in time $\Theta(n)$. The best-case and worst-case runtime is thus $\Theta(n)$. □