COMP 3804 — Winter 2026
Solutions Problem Set 1

Some useful facts:

1. $1 + 2 + 3 + \cdots + n = n(n+1)/2$.
2. for any real number $x > 0$, $x = 2^{\log x}$.
3. For any real number $x \neq 1$ and any integer $k \geq 1$,
$$1 + x + x^2 + \cdots + x^{k-1} = \frac{x^k - 1}{x - 1}.$$
4. For any real number $0 < \alpha < 1$,
$$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

Master Theorem:

1. Let $a \geq 1$, $b > 1$, $d \geq 0$, and
$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ a \cdot T(n/b) + \Theta(n^d) & \text{if } n \geq 2. \end{cases}$$
2. If $d > \log_b a$, then $T(n) = \Theta(n^d)$.
3. If $d = \log_b a$, then $T(n) = \Theta(n^d \log n)$.
4. If $d < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.

Question 1: After having attended the first lecture of COMP 3804, Justin Bieber is intrigued by the recursive algorithm $\text{Fib}(n)$ that computes the n-th Fibonacci number in exponential time. He is convinced that a simple modification should run much faster. Here is Justin's algorithm.

> **Algorithm** $\text{FibBieber}(n)$:
> **comment:** $n \geq 0$ is an integer
> initialize an array $f(0 \ldots n)$;
> **for** $i = 0, 1, \ldots n$ **do** $f(i) = -1$
> **endfor**;
> $\text{Bieber}(n)$;
> return $f(n)$

> **Algorithm** $\text{Bieber}(m)$:
> **comment:** $0 \leq m \leq n$, this algorithm has access to the array $f(0 \ldots n)$
> **if** $m = 0$
> **then** $f(0) = 0$
> **endif**;
> **if** $m = 1$
> **then** $f(0) = 0$; $f(1) = 1$
> **endif**;
> **if** $m \geq 2$
> **then if** $f(m-2) = -1$
>   **then** $\text{Bieber}(m-2)$
>   **endif**;
> $x = f(m-2)$;
>   **if** $f(m-1) = -1$
>   **then** $\text{Bieber}(m-1)$
>   **endif**;
> $y = f(m-1)$;
> $f(m) = x + y$
> **endif**

- Is algorithm FibBieber correct? That is, is it true that for every integer $n \geq 0$, the output of algorithm $\text{FibBieber}(n)$ is the n-th Fibonacci number? As always, justify your answer.
- What is the running time of algorithm $\text{FibBieber}(n)$? You may assume that two integers can be added in constant time. As always, justify your answer.

Solution: As in class, we denote the Fibonacci numbers by $F_0, F_1, F_2, \ldots$

This question is more difficult than I thought.

We will show that algorithm FibBieber is correct. Recall that $\text{FibBieber}(n)$ does the following:

- Set $f(0) = f(1) = f(2) = \cdots = f(n) = -1$.
- Run BIEBER(n). This makes recursive calls to BIEBER with smaller and smaller arguments.
- Return $f(n)$.
- We have to show that $f(n) = F_n$.

Claim: For every integer $n \geq 0$, the following hold during the execution of BIEBER(n):

- C1: At any moment, for every $k = 0, 1, \ldots, n$:

$$f(k) = -1 \text{ or } f(k) = F_k.$$

- C2: For every $k = 0, 1, \ldots, n$: At the moment BIEBER(k) has terminated:

$$f(0) = F_0, f(1) = F_1, \ldots, f(k) = F_k.$$

Before we prove this claim: If it is correct, then it follows that algorithm FIBBIEBER(n) returns F_n.

Proof: If $n = 0$ or $n = 1$, C1 and C2 follow from the pseudocode.

Let $n \geq 2$ and assume that C1 and C2 are true for every n' with $0 \leq n' < n$.

Note that C1 is true just before BIEBER(n) is called. Also note that C1 and C2 are true at the moment when BIEBER(0) and BIEBER(1) have terminated.

Let m be such that $2 \leq m < n$. What happens when we run BIEBER(m)? We go through the pseudocode:

- Since $m \geq 2$, the algorithm checks if $f(m-2) = -1$.
  - If this is the case, then we run BIEBER($m-2$). By induction, at termination, we have
  $$f(0) = F_0, f(1) = F_1, \ldots, f(m-2) = F_{m-2}.$$
  - If this is not the case then, by C1, $f(m-2) = F_{m-2}$.
  - Because of the previous two items, we always have $x = f(m-2) = F_{m-2}$.
- Next, the algorithm checks if $f(m-1) = -1$.
  - If this is the case, then we run BIEBER($m-1$). By induction, at termination, we have
  $$f(0) = F_0, f(1) = F_1, \ldots, f(m-1) = F_{m-1}.$$
  - If this is not the case then, by C1, $f(m-1) = F_{m-1}$.
  - Because of the previous two items, we always have $y = f(m-1) = F_{m-1}$.

- At the end of $\textsc{Bieber}(m)$, we set

$$f(m) = x + y = F_{m-2} + F_{m-1} = F_m.$$

- This proves the claim.

Next we estimate the running time of algorithm $\textsc{FibBieber}(n)$. This running time is $O(n)$ (to initialize the array f), plus the running time of $\textsc{Bieber}(n)$, plus $O(1)$ (to return $f(n)$). If we can show that $\textsc{Bieber}(n)$ takes $O(n)$ time, then the total running time of $\textsc{FibBieber}(n)$ is $O(n)$.

Let $T(n)$ be the running time of $\textsc{Bieber}(n)$. For $n \in \{0, 1\}$, $T(n)$ is bounded from above by some constant.

Let $n \geq 2$. Let us see what $\textsc{Bieber}(n)$ does:

- The algorithm takes time $O(1)$ plus the time of the recursive call $\textsc{Bieber}(n-2)$, which is $O(1) + T(n-2)$.
- There may be a call to $\textsc{Bieber}(n-1)$. If this is the case, then:
  - There is no recursive call to $\textsc{Bieber}(n-3)$; this follows from C1 and C2, because $\textsc{Bieber}(n-2)$ has terminated.
  - There is no recursive call to $\textsc{Bieber}(n-2)$; this follows from C1 and C2, because $\textsc{Bieber}(n-2)$ has terminated.
  - Thus, the call to $\textsc{Bieber}(n-1)$ takes $O(1)$ time.
- It follows that
$$T(n) = O(1) + T(n-2).$$
Straightforward unfolding implies that $T(n) = O(n)$.

Question 2: Taylor Swift is not impressed by Justin's algorithm in the previous question. Taylor is convinced that there is a much simpler algorithm. Here is Taylor's algorithm:

> **Algorithm** FIBSWIFT(n):
> **comment:** $n \geq 0$ is an integer
> initialize an array $f(0 \dots n)$;
> **for** $i = 0, 1, \dots n$ **do** $f(i) = -1$
> **endfor**;
> SWIFT(n);
> return $f(n)$
>
> **Algorithm** SWIFT(m):
> **comment:** $0 \leq m \leq n$, this algorithm has access to the array $f(0 \dots n)$
> **if** $m = 0$
> **then** $f(0) = 0$
> **endif**;
> **if** $m = 1$
> **then** $f(0) = 0$; $f(1) = 1$
> **endif**;
> **if** $m \geq 2$
> **then** SWIFT($m - 1$);
> $f(m) = f(m-1) + f(m-2)$;
> **endif**

- Is algorithm FIBSWIFT correct? That is, is it true that for every integer $n \geq 0$, the output of algorithm FIBSWIFT(n) is the n-th Fibonacci number? As always, justify your answer.
- What is the running time of algorithm FIBSWIFT(n)? You may assume that two integers can be added in constant time. As always, justify your answer.

Solution: Swifties will not be surprised that algorithm FIBSWIFT is correct. Recall that FIBSWIFT(n) does the following:

- Set $f(0) = f(1) = f(2) = \cdots = f(n) = -1$.
- Run SWIFT(n). This makes recursive calls to SWIFT with smaller and smaller arguments.
- Return $f(n)$.
- We have to show that $f(n) = F_n$.

Claim: For every integer $m = 0, 1, \ldots, n$, at the moment when $\textsc{Swift}(m)$ terminates:

$$f(0) = F_0, f(1) = F_1, \ldots, f(m) = F_m.$$

Before we prove this claim: If it is correct, then it follows that algorithm $\textsc{FibSwift}(n)$ returns F_n.

Proof: For $m \in \{0, 1\}$, the claim follows from the pseudocode.

Let $m \geq 2$, and assume that the claim is true for all m' with $0 \leq m' < m$. What does algorithm $\textsc{Swift}(m)$ do:

- It runs $\textsc{Swift}(m-1)$. By induction, at termination of this recursive call, we have

$$f(0) = F_0, f(1) = F_1, \ldots, f(m-2) = F_{m-2}, f(m-1) = F_{m-1}.$$

- It sets $f(m) = f(m-1) + f(m-2)$, which is $F_{m-1} + F_{m-2}$, which is F_m.
- Thus, at termination of $\textsc{Swift}(m)$, we have

$$f(0) = F_0, f(1) = F_1, \ldots, f(m) = F_m.$$

Next we estimate the running time of algorithm $\textsc{FibSwift}(n)$. This running time is $O(n)$ (to initialize the array f), plus the running time of $\textsc{Swift}(n)$, plus $O(1)$ (to return $f(n)$). If we can show that $\textsc{Swift}(n)$ takes $O(n)$ time, then the total running time of $\textsc{FibSwift}(n)$ is $O(n)$.

Let $T(n)$ be the running time of $\textsc{Swift}(n)$. For $n \in \{0, 1\}$, $T(n)$ is bounded from above by some constant.

Let $n \geq 2$. The time for $\textsc{Swift}(n)$ is equal to $O(1)$ plus the time for $\textsc{Swift}(n-1)$. Thus,

$$T(n) = O(1) + T(n-1).$$

Straightforward unfolding implies that $T(n) = O(n)$.

Question 3: Consider the following recurrence, where n is a power of 7:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ n^3 + 12 \cdot T(n/7) & \text{if } n \geq 7. \end{cases}$$

- Solve this recurrence using the *unfolding method*. Give the final answer using Big-O notation.
- Solve this recurrence using the *Master Theorem*.

Solution: We write $n = 7^k$. Unfolding gives

$$\begin{aligned}
T(n) &= n^3 + 12 \cdot T(n/7) \\
&= n^3 + 12\left((n/7)^3 + 12 \cdot T(n/7^2)\right) \\
&= \left(1 + 12/7^3\right) n^3 + 12^2 \cdot T(n/7^2) \\
&= \left(1 + 12/7^3\right) n^3 + 12^2\left((n/7^2)^3 + 12 \cdot T(n/7^3)\right) \\
&= \left(1 + 12/7^3 + (12/7^3)^2\right) n^3 + 12^3 \cdot T(n/7^3) \\
&= \left(1 + 12/7^3 + (12/7^3)^2\right) n^3 + 12^3\left((n/7^3)^3 + 12 \cdot T(n/7^4)\right) \\
&= \left(1 + 12/7^3 + (12/7^3)^2 + (12/7^3)^3\right) n^3 + 12^4 \cdot T(n/7^4) \\
&\vdots \\
&= \left(1 + 12/7^3 + (12/7^3)^2 + \cdots + (12/7^3)^{k-1}\right) n^3 + 12^k \cdot T(n/7^k) \\
&= \left(1 + 12/7^3 + (12/7^3)^2 + \cdots + (12/7^3)^{k-1}\right) n^3 + 12^k \cdot 1.
\end{aligned}$$

Let $x = 12/7^3$. Then

$$T(n) = \frac{x^k - 1}{x - 1} \cdot n^3 + 12^k.$$

Since $0 < x < 1$, we write this as

$$T(n) = \frac{1 - x^k}{1 - x} \cdot n^3 + 12^k.$$

Since $1 - x^k \leq 1$ and $1/(1-x)$ is a constant, we have

$$\frac{1 - x^k}{1 - x} \cdot n^3 = O(n^3).$$

Since $n = 7^k$, we have $\log n = k \log 7$, and

$$\begin{aligned}
12^k &= \left(12^{\log n}\right)^{1/\log 7} \\
&= \left(n^{\log 12}\right)^{1/\log 7} \\
&= n^{\log 12/\log 7} \\
&= n^{\log_7 12} \\
&= n^{1.2769}.
\end{aligned}$$

We conclude that

$$T(n) = O(n^3) + n^{1.2769} = O(n^3).$$

This was fun, eh!

Using the Master Theorem: We have $a = 12$, $b = 7$, and $d = 3$. Since

$$\log_b a = \log_7 12 = 1.2769 < d,$$

the Master Theorem tells us that $T(n) = O(n^d) = O(n^3)$.

Question 4: You are given an array $A(1 \dots n)$ of n distinct numbers. This array has the following property: There is an index i with $1 \leq i \leq n$, such that

1. the subarray $A(1 \dots i)$ is sorted in increasing order, and
2. the subarray $A(i \dots n)$ is sorted in decreasing order.

Describe a recursive algorithm that returns, in $O(\log n)$ time, the largest number in the array A. (At the start of the algorithm, you do not know the above index i.)

You may describe your algorithm in plain English or in pseudocode. Justify the correctness of your algorithm and explain why the running time is $O(\log n)$. You may use any result that was proven in class.

Solution: Because of the word "recursive" and a running time of $O(\log n)$, it makes sense to use binary search.

Invariant: ℓ and r are indices with $1 \leq \ell < r \leq n$. The largest number in the entire array is in the subarray $A(\ell \dots r)$.

Initially, we set $\ell = 1$ and $r = n$. Note that the invariant holds.

While $r - \ell + 1$ (which is the length of the subarray $A(\ell \dots r)$) is large, say more than 5, do the following:

- Let $k = \lfloor (r - \ell + 1)/2 \rfloor$.
- If $A(k) < A(k + 1)$, then we set $\ell = k + 1$. Why: because the largest number is in the subarray $A(k + 1 \dots r)$. In this way, the invariant still holds.
- If $A(k) > A(k + 1)$, then we set $r = k$. Why: because the largest number is in the subarray $A(\ell \dots k)$. In this way, the invariant still holds.

Repeat the above as long as $r - \ell + 1 > 5$. Since in each iteration, the value of $r - \ell + 1$ gets smaller, at some point it is at most 5. At this moment, we scan the subarray $A(\ell \dots r)$ and find the largest number.

What is the running time: Initially, we have $r - \ell + 1 = n$. In each iteration, the value of $r - \ell + 1$ is divided by (roughly) 2. It follows that the number of iterations is $O(\log n)$. Since each iteration takes $O(1)$ time, the total running time is $O(\log n)$.

Question 5: You are given a sequence $S = (a_1, a_2, \dots, a_n)$ of n distinct numbers. A pair (a_i, a_j) is called *Out-of-Order*, if $i < j$ and $a_i > a_j$; in words, a_i is to the left of a_j and a_i is larger than a_j.

If the sequence S is sorted then the number of Out-of-Order pairs is zero. On the other hand, if S is sorted in decreasing order, then there are $\binom{n}{2}$ Out-of-Order pairs.

Describe a comparison-based divide-and-conquer algorithm that returns, in $O(n \log n)$ time, the number of Out-of-Order pairs in the sequence S.

You may describe your algorithm in plain English or in pseudocode. Justify the correctness of your algorithm and explain why the running time is $O(n \log n)$. You may use any result that was proven in class.
Hint: Think of Merge-Sort.

Solution: We are going to add some steps to Merge-Sort. After the algorithm has terminated, the sequence will be sorted and we have counted the OoO-pairs.

We assume for simplicity that n is a power of two. Here is the algorithm:

Initialization: Set $OoO = 0$.

If $n = 1$**:** Return OoO.

If $n \geq 2$**:**

- Let $m = n/2$. Split S into $A = (a_1, \ldots, a_m)$ and $B = (a_{m+1}, \ldots, a_n)$.
- Observation: Every OoO-pair (a_i, a_j) in S for which $i \leq m$ and $j > m$, is still an OoO-pair if we permute the sequences A and B.
- We recursively run the algorithm on A. After termination, A is sorted and we know the number OoO_A of OoO-pairs in A.
- We recursively run the algorithm on B. After termination, B is sorted and we know the number OoO_B of OoO-pairs in B.
- Set $OoO = OoO + OoO_A + OoO_B$.
- It remains to merge A and B into one sorted sequence, and count the OoO-pairs with one number in A and the other number in B.
- Initialize an empty sequence C. (This sequence will contain the numbers in $S = A \cup B$ in sorted order.)
- While both A and B are non-empty:
  - Let x be the first element in A.
  - Let y be the first element in B.
  - If $x < y$: Delete x from A and add it at the end of C. (Explanation: x does not form an OoO-pair with any number in B.)
  - If $x > y$: Delete y from B, add it at the end of C, and set $OoO = OoO + |A|$. (Explanation: Every number in A forms an OoO-pair with y.)
- If A is empty: Add A at the end of C.
- If B is empty: Add B at the end of C.

- Return OoO.

Let $T(n)$ denote the running time on a sequence of length n. It follows from the algorithm that, for $n \geq 2$,

$$T(n) = O(n) + 2 \cdot T(n/2).$$

This is the Merge-Sort recurrence and solves to $T(n) = O(n \log n)$.