Problem 1. Consider the following algorithm.

**Algorithm 1:** $\text{DFS}(T = (L, R))$

<table>
<thead>
<tr>
<th>Input:</th>
<th>Binary tree $T$, where $L$ is the left (possibly empty) subtree and $R$ is the right (possibly empty) subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $T$ is empty then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return 0</td>
</tr>
</tbody>
</table>
| return 1 + max(DFS(L),DFS(R))

a) What does $\text{DFS}(T)$ compute?

b) Prove your answer to the previous problem is correct using induction on the number of vertices.

(Hint: choose your inductive hypothesis carefully; subtrees vary wildly in size)

c) Give a recurrence relation $R(n)$ for the runtime of $\text{DFS}(T)$ in terms of $n = |T|$ (the number of vertices of $T$).

(Hint: you will want to introduce a utility variable $m = |L|$)

d) Give a good upper bound on $R(n)$ via big-O notation and justify your choice.

**Solution 1.**

a) $\text{DFS}(T)$ computes the height of $T$.

b) The base case is when $T$ has no vertices. This only occurs when $T$ is empty. Thus $\text{DFS}$ correctly returns a height of 0 on any tree with no vertices.

Now assume that $\text{DFS}(T)$ returns the height of $T$ for any tree with at most $n$ vertices.

Let $T = (L, R)$ be a tree with $n+1$ vertices. Then the number of vertices in both $L$ and $R$ is at most $n$. So by assumption, $\text{DFS}(L)$ returns the height of $L$. Also by assumption, $\text{DFS}(R)$ returns the height of $R$.

The height of $T$ is one more than the greatest height of its children by definition. But $\text{DFS}$ returns exactly this, so $\text{DFS}(T)$ correctly returns the height of $T$. \(\square\)

c) $R(n) = 1 + R(n - 1 - m) + R(m)$
d) We know that \( R(n) \in O(n) \). This is because the sum of the input size across all recursive calls of the same depth always decreases by the number of recursive calls. As such, we can make at most \( n \) recursive calls, and each call does \( O(1) \) work. Thus the total work is \( O(1 \cdot n) = O(n) \)

**Problem 2.** Consider the following algorithm.

**Algorithm 2: SS\( \text{Sum}(n) \)**

\[
\text{Input: } n \in \mathbb{N} \\
\text{if } n = 0 \text{ then} \\
\quad \text{return 0} \\
\text{return } n \cdot n + \text{SS\( \text{um}(n-1) \)}
\]

a) What does SS\( \text{Sum}(n) \) compute in terms of \( n \)?

b) Recall that the input size here is \( m = \lfloor \log n \rfloor \) since a number is an array of bits. As a consequence, arithmetic is not considered a constant time operation. You may assume that multiplication takes \( m^2 \) time and addition/subtraction takes \( m \) time.

Give a recurrence relation \( T(n) \) for the runtime of SS\( \text{Sum}(n) \) in terms of \( n \).

c) Give a good upper bound on \( T(n) \) via big-O notation in terms of \( n \) and justify your choice.

(Hint: overestimate \( n \) to one less than the next higher power of 2 [e.g. 6 \( \rightarrow \) 7, 44 \( \rightarrow \) 63]; feel free to use wolfram alpha to number crunch the resulting sum)

d) What is the runtime of SS\( \text{Sum}(n) \) in terms of the input size \( m \)?

**Solution 2.**

a) SS\( \text{Sum}(n) \) computes the sum of the first \( n \) squares.

b) \( T(n) = [\log n]^2 + 2[\log n] + T(n - 1) \)

c) We’ll assume the worst case for an input of size \( m \), that is that \( n \) is one less than a power of two, and overestimate the true runtime of \( n \).

\[
T(n) \leq \sum_{k=1}^{\log n + 1} 2^k(k^2 + k)
\]
\[
\leq \sum_{k=1}^{\log n + 1} 2^k(k^2 + k^2)
\]
\[
= 2 \sum_{k=1}^{\log n + 1} 2^k k^2
\]
\[
\approx 2(nm^2 - 2nm + 3n - 3)
\]

This gives us that \( T(n) \in O(n \log^2 n) \).

d) \( O(m^2 2^m) \)