CSC165H1: Problem Set 4 Sample Solutions

Due Thursday March 28, 2019 before 4pm

Note: solutions may be incomplete, and meant to be used as guidelines only. We encourage you to ask follow-up questions on the course forum or during office hours.

1. [9 marks] Printing multiples. Consider the following algorithm:

```python
def print_multiples(n: int) -> None:
    """Precondition: n > 0."""
    for d in range(1, n + 1):  # Loop 1
        for multiple in range(0, n, d):  # Loop 2 (also, see Python Note below)
            print(multiple)
```

Python Note: `range(0, n, d)` consists of the multiples of \( d \geq 0 \) and \( < n: 0, d, \ldots, d([n/d] - 1) \).

(a) [3 marks] Find, with proof, a Theta bound for \( \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \) in terms of elementary functions (e.g., \( n \), \( \log n \), \( 2^n \), etc.). You may not use any properties of Big-Oh/Omega/Theta; instead, only use their definitions, as well as the following Facts (clearly state where you use them):

\[
\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n) \quad \text{(Fact 1)}
\]

\[\forall x \in \mathbb{R}, \ x \leq \lfloor x \rfloor < x + 1 \quad \text{(Fact 2)}\]

Solution

Claim: \( \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \in \Theta(n \log n) \).

Proof. We want to prove that there exist \( c_1, c_2, n_0 \in \mathbb{R}^+ \) such that for all \( n \in \mathbb{N} \), if \( n \geq n_0 \) then \( c_1 n \log n \leq \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \leq c_2 n \log n \).

Before picking these variables, we use Fact 1, which tells us that there exist \( c_3, c_4, n_1 \in \mathbb{R}^+ \) such that for all \( n \in \mathbb{N} \), if \( n \geq n_1 \) then \( c_3 \log n \leq \sum_{i=1}^{n} \frac{1}{i} \leq c_4 \log n \).

Let \( c_1 = c_3 \), \( c_2 = c_4 + 1 \), and \( n_0 = n_1 + 2 \). Let \( n \in \mathbb{N} \) and assume \( n \geq n_0 \). We’ll prove each of the two inequalities separately.
Part 1 \((\geq c_1 n \log n)\)

\[
\sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \geq \sum_{i=1}^{n} \frac{n}{i} \tag{first part of Fact 2}
\]

\[
= n \times \sum_{i=1}^{n} \frac{1}{i}
\]

\[
\geq n \times (c_3 \log n) \tag{Fact 1}
\]

\[
= c_3 n \log n
\]

Part 2 \((\leq c_2 n \log n)\)

\[
\sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor \leq \sum_{i=1}^{n} \left( \frac{n}{i} + 1 \right) \tag{second part of Fact 2}
\]

\[
= \left( \sum_{i=1}^{n} \frac{n}{i} \right) + \left( \sum_{i=1}^{n} 1 \right)
\]

\[
= n \times \left( \sum_{i=1}^{n} \frac{1}{i} \right) + n
\]

\[
\leq (n \times c_4 \log n) + n \tag{Fact 1}
\]

\[
\leq c_4 n \log n + n \log n \tag{since } n \geq 2, \log n \geq 1
\]

\[
= (c_4 + 1)n \log n
\]

\[
= c_2 n \log n
\]

\(\square\)

(b) [3 marks] Using your answer to part (a), analyse the running time of `print_multiples` to determine a Theta bound on the running time.

Solution

We first analyse the inner loop (Loop 2) for a fixed iteration of the outer loop:

- Each iteration takes constant time (1 step).
- The number of iterations is \(\lfloor n/d \rfloor\) (this follows directly from the Python Note).

For the outer loop (Loop 1):

- This loop iterates for \(d = 1, 2, \ldots, n\).
- For iteration \(d\), the running time is \(\lfloor n/d \rfloor\) (the cost of running the inner loop).
- So the total running time is

\[
\sum_{d=1}^{n} \left\lfloor \frac{n}{d} \right\rfloor
\]

By part (a), this expression is \(\Theta(n \log n)\), and so the overall running time is \(\Theta(n \log n)\).

(c) [3 marks] Now consider the following variation on the above algorithm.
```python
def print_multiples2(n: int) -> None:
    """Precondition: n > 0."""
    for d in range(1, n + 1):  # Loop 1
        for multiple in range(0, n, d):  # Loop 2
            print(multiple)
        if d % 5 == 0:
            for i in range(d):  # Loop 3
                print(i)
```

Analyse the running time of `print_multiples2` to determine a Theta bound on the running time. To simplify your analysis, you can focus on just counting the steps taken by Lines 5 and 9.

You may use your work from previous parts of this question, and the summation formula $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

**Solution**

The analysis for Loop 2 is the same as part (b).

For Loop 3 (if it runs): each iteration takes constant time (1 step), and it runs for $d$ iterations, for a total running time of $d$ steps.

For Loop 1:

- Again, $d$ goes from 1 to $n$.
- The cost of iteration $d$ is now $\lceil n/d \rceil + d$ if $d$ is a multiple of 5, and $\lceil n/d \rceil$ otherwise.
- Since we are adding up the total steps over all the iterations, we can do the totals separately for Loop 2 and Loop 3.

The total cost of Loop 2 across all iterations of Loop 1 is the same as part (b): $\sum_{i=1}^{n} \lceil n/d \rceil$.

The total cost of Loop 3 across all iterations of Loop 1 is the sum of “$d$ steps” for every $d$ between 1 and $n$, inclusive, that’s a multiple of 5. We can write $d = 5k$, where $1 \leq k \leq \lfloor n/5 \rfloor$.

$$
\sum_{k=1}^{\lfloor n/5 \rfloor} 5k = 5 \times \sum_{k=1}^{\lfloor n/5 \rfloor} k
= 5 \times \left( \frac{n}{5} \right) \left( \left\lfloor \frac{n}{5} \right\rfloor + 1 \right)
$$

So the total running time of this algorithm is

$$
\left( \sum_{i=1}^{n} \lceil n/d \rceil \right) + \frac{5}{2} \left( \left\lfloor \frac{n}{5} \right\rfloor \left( \left\lfloor \frac{n}{5} \right\rfloor + 1 \right) \right)
$$

In that expression, the first term is $\Theta(n \log n)$, and the second term is $\Theta(n^2)$, so the overall running time is $\Theta(n^2)$. 
2. [9 marks] Varying running times, input families, and worst-case analysis.

```python
def alg(lst: List[int]) -> None:
    n = len(lst)
    i = 1
    j = 1
    while i < n:  # Loop 1
        if lst[i] % 2 == 0:
            i = i + 1
            j = j * 2
        else:
            i = i * 2
    for k in range(j):  # Loop 2
        print(k)
```

To simplify your analyses for this question, ignore the costs of Lines 2–4 and Line 10. (Of course, you’ll have to understand what these lines do to analyse the running time, but you don’t need to count them as “steps.”)

(a) [2 marks] Find, with proof, an input family for `alg` whose running time is $\Theta(2^n)$.

Solution
Let $n \in \mathbb{N}$, and let $\text{lst} = [0, \ldots, 0, 1]$, with $n - 1$ 0’s followed by a single 1.∗

Part 1.
The first $n - 2$ iterations of the `while` loop execute the first branch of the `if` statement (because $\text{lst}[i] \% 2 == 0$ for $i = 1, \ldots, n - 2$). The body of the `while` loop takes constant time, so for these $n - 2$ iterations, $n - 2$ steps occur.

Also, note that starting from $j = 1$, the value of $j$ doubles at each iteration. So by the end of iteration $n - 2$, $j = 2^{n-2}$.

Part 2. During iteration number $n - 1$ (when $i = n - 1$), $\text{lst}[n-1] \% 2 != 0$, so the `else` branch executes. The `for` loop (Loop 2) executes based on the current value of $j = 2^{n-2}$.

- The inner loop body takes constant time (1 step).
- The inner loop iterates exactly $j = 2^{n-2}$ times.
- So the `for` loop executes $2^{n-2}$ steps in total.

Putting it together.
Overall, the running time is $n - 2 + 2^{n-2}$ (adding up the cost of each of the above parts). This gives us a running time of $\Theta(2^n)$. (Note: $2^{n-2} = \frac{1}{4} \cdot 2^n$.)

∗Technically here we assume $n \geq 1$. This is okay because we’re only looking for a $\Theta$ expression at the end.

(b) [3 marks] Find, with proof, an input family for `alg` whose running time is $\Theta(\log n \times 2^{\sqrt{n}})$, where the `else` branch of Loop 1 runs $\Theta(\log n)$ times.

You can use the fact that $\log n - \log(\log n) \in \Theta(\log n)$ in your proof.
Let \( n \in \mathbb{N} \), and let \( \text{lst} \) be the list with \( \left\lceil \sqrt{n} \right\rceil \) 0’s followed by \( n - \left\lceil \sqrt{n} \right\rceil \) 1’s.

**Part 1.**

The first \( \left\lceil \sqrt{n} \right\rceil - 1 \) iterations of the \textbf{while} loop execute the first branch of the \textbf{if} statement (because \( \text{lst}[i] \% 2 == 0 \) for \( i = 1, \ldots, \left\lceil \sqrt{n} \right\rceil - 1 \)). The body of the \textbf{while} loop takes constant time, so for these \( \left\lceil \sqrt{n} \right\rceil - 1 \) iterations, \( \left\lceil \sqrt{n} \right\rceil - 1 \) steps occur.

Starting from \( j = 1 \), the value of \( j \) doubles at each iteration. So by the end of iteration \( \left\lceil \sqrt{n} \right\rceil \), \( j = 2^{\left\lceil \sqrt{n} \right\rceil - 1} \).

**Part 2.**

After these iterations, the \textbf{else} branch executes at each subsequent iteration of Loop 1. Since at this point \( i = \left\lceil \sqrt{n} \right\rceil \) and its value doubles at each iteration, after \( k \) executions of the \textbf{else} branch its value is \( \left\lceil \sqrt{n} \right\rceil \cdot 2^k \).

- Since Loop 1 stops when \( i \geq n \), the number of times the \textbf{else} branch executes is \( k = \left\lceil \log(n/\left\lfloor \sqrt{n} \right\rfloor) \right\rceil \).
- Inside the \textbf{else} branch, Loop 2 iterates \( j = 2^{\left\lceil \sqrt{n} \right\rceil} \) times, with each iteration taking a single step. So each time the \textbf{else} branch runs, there are \( 2^{\left\lceil \sqrt{n} \right\rceil} \) iterations.

**Putting it together.**

So then overall, the total running time is \( \left\lceil \sqrt{n} \right\rceil - 1 \) steps for the iterations of Loop 1 in which the \textbf{if} branch runs, plus \( \left\lceil \log(n/\left\lfloor \sqrt{n} \right\rfloor) \right\rceil \cdot 2^{\left\lceil \sqrt{n} \right\rceil} \) steps for the iterations of Loop 1 in which the \textbf{else} branch runs.

We know that \( \left\lceil \sqrt{n} \right\rceil - 1 \in \Theta(\sqrt{n}) \). Also, \( \log(n/\left\lfloor \sqrt{n} \right\rfloor) \in \Theta(\log n) \) and \( 2^{\left\lceil \sqrt{n} \right\rceil - 1} \in \Theta(2^{\sqrt{n}}) \), and so \( \left\lceil \log(n/\left\lfloor \sqrt{n} \right\rfloor) \right\rceil \cdot 2^{\left\lceil \sqrt{n} \right\rceil - 1} \in \Theta(\log n \times 2^{\sqrt{n}}) \).

Finally, since \( \sqrt{n} \in \mathcal{O}(\log n \times 2^{\sqrt{n}}) \), we can conclude that the total running time for this input family is \( \Theta(\log n \times 2^{\sqrt{n}}) \).

(c) \[ 4 \text{ marks} \]

Prove that the worst-case running time of \( \text{alg} \) is \( \mathcal{O}(2^n) \) (note that this matches the worst-case lower bound we can derive from part (a)).

HINT: unlike other examples we’ve looked at in class, there are no “early returns” here. Instead, use what you’ve learned about analysing loops with non-standard loop variable changes, and remember that you don’t need \textit{exactly} \( 2^n \) steps, just an \textit{at most} \( \mathcal{O}(2^n) \).

**Solution**

Let \( n \in \mathbb{N} \), and let \( \text{lst} \) be an arbitrary list of integers of length \( n \).

The key idea for this proof is that the only way for the \textbf{else} branch to run approximately \( 2^n \) times is for both \( i \) and \( j \) to be large. First, we formalize this idea with the following claim.

**Claim.** After every loop iteration, \( j \leq 2^i \).

When the loop starts, \( j = 1 \) and \( i = 0 \), so \( j = 2^i \). Whenever the \textbf{if} branch executes, \( i \) increases by 1 and \( j \) doubles in value, so the inequality still holds. Whenever the \textbf{else} branch executes, \( i \) increases (by a factor of 2) and \( j \) stays the same, so \( 2^i \) only gets bigger.

**Case 1:** assume that the \textbf{else} branch \textit{never} executes when \( i \geq \frac{n}{2} \).
In this case, from the previous claim we know that \( j \leq 2^n \), and so each time Loop 2 runs, it takes at most \( 2^n \) steps in total.

We can overestimate the total running time of Loop 1 as follows:

- The variable \( i \) starts at 1 and increases by at least 1 each iteration. Since Loop 1 stops when \( i \geq n \), there are at most \( n \) iterations (in fact, at most \( n - 1 \) iterations since \( i \) starts at 1, but we can overestimate to \( n \)).
- Each iteration takes either 1 step if the if branch runs, and at most \( 2^n \) steps if the else branch runs.
- So the total cost of Loop 1 is at most \( n \cdot 2^n \).

So the total running time in this case is at most \( n \cdot 2^n \in O(2^n) \).

**Case 2:** assume that the else branch executes at least once when \( i > \frac{n}{2} \).

We first note that the value of \( i \) doubles when the else branch executes. So in this case, the else branch executes at most once with the value of \( i \) being \( > \frac{n}{2} \).

Let \( k \) be the iteration number at which the else branch executes and \( k > \frac{n}{2} \). To add up the total running time of Loop 1 in this case, we divide its iterations before \( k \), and iteration \( k \).

- For the same reason as Case 1 above, for the iterations before \( k \), the value of \( j \) is \( \leq \frac{n}{2} \), and so each of these iterations takes at most \( 2^n \) steps.
- Since \( i \) increases by at least 1 per iteration, at most \( \frac{n}{2} \) iterations occur until \( i > \frac{n}{2} \), and so there are at most \( \frac{n}{2} \) iterations before \( k \) (i.e., \( k \leq \frac{n}{2} \)).
- So the total running time of the iterations before \( k \) is \( \frac{n}{2} \cdot 2^n \).
- Finally for iteration \( k \), since \( i < n \) (this is the Loop 1 condition), we know from the previous claim that \( j < 2^n \), and so the else branch takes at most \( 2^n \) steps.

So the total running time of Loop 1 in this case is at most \( \frac{n}{2} \cdot 2^n + 2^n \) steps, which is \( O(2^n) \).

We define the best-case running time of an algorithm func as follows:

\[ BC_{\text{func}}(n) = \min\{\text{running time of executing } \text{func}(x) \mid x \in \mathcal{I}_n\} \]

Note that this is analogous to the definition of worst-case running time, except we use min instead of max.

(a) [4 marks] Review the definitions of what it means for a function \( f : \mathbb{N} \to \mathbb{R}_{\geq 0} \) to be an upper bound or lower bound on the worst-case running time of an algorithm (Definitions 5.11 and 5.12 in the Course Notes).

Then, let \( \mathcal{I}_n \) denote the set of all inputs of size \( n \) for rearrange, and write two symbolic definitions:

(i) What it means for a function \( f : \mathbb{N} \to \mathbb{R}_{\geq 0} \) to be an upper bound on the best-case running time of an algorithm.

(ii) What it means for a function \( f : \mathbb{N} \to \mathbb{R}_{\geq 0} \) to be a lower bound on the best-case running time of an algorithm.

Clearly state which definition is which in your answer. Your definitions should be very similar to the definitions we mentioned above from the Course Notes.

**Hint:** it is easier to first translate the simpler statements “\( M \) is an upper bound on the minimum of set \( S \)” and “\( M \) is a lower bound on the minimum of set \( S \)” to make sure you have the right idea.

**Solution**

Let \( \mathcal{I}_{\text{func},n} \) denote the set of all inputs of size \( n \) for the program func.

Upper bound:

\[
\forall n \in \mathbb{N}, \ BC_{\text{func}}(n) \leq f(n) \iff \forall n \in \mathbb{N}, \ \min\{\text{running time of } \text{func}(x) \mid x \in \mathcal{I}_{\text{func},n}\} \leq f(n) \iff \forall n \in \mathbb{N}, \ \exists x \in \mathcal{I}_{\text{func},n}, \ \text{running time of } \text{func}(x) \leq f(n)
\]

Lower bound:

\[
\forall n \in \mathbb{N}, \ BC(n) \geq f(n) \iff \forall n \in \mathbb{N}, \ \min\{\text{running time of } \text{func}(x) \mid x \in \mathcal{I}_{\text{func},n}\} \geq f(n) \iff \forall n \in \mathbb{N}, \ \forall x \in \mathcal{I}_{\text{func},n}, \ \text{running time of } \text{func}(x) \geq f(n)
\]

(b) [4 marks] Consider the following Python function.

```python
def rearrange(lst: List[int]) -> None:
    for i in range(2, len(lst)):
        if i % 2 == 0:
            j = i - 2
            while j >= 0 and lst[j+2] < lst[j]:
                lst[j+2], lst[j] = lst[j], lst[j+2]  # Swap lst[j] and lst[j+2]
                j = j - 2
        else:
            j = i - 2
            while j >= 0 and lst[j+2] > lst[j]:
                lst[j+2], lst[j] = lst[j], lst[j+2]  # Swap lst[j] and lst[j+2]
                j = j - 2
```
Analyse the best-case running time of `rearrange` to find a Theta bound for it. Your analysis should consist of two parts, proving matching upper and lower bounds on the best-case running time separately, and then using them to conclude a Theta bound.

To simplify your analysis, you only need to count the total number of steps taken inside Loops 2 and 3, and not the other operations (e.g., Lines 4 and 9).

**Solution**

**Lower bound** Let \( n \in \mathbb{N} \), and let \( \text{lst} \) be an arbitrary list of integers of length \( n \).

The outer loop (Loop 1) iterates exactly \( \max(n-2, 0) \) times, since \( i \) goes from 2 to \( n-1 \). Each iteration takes at least one step (this includes the possibility that Loops 2 and 3 don’t iterate at all!).

So the total running time is at least \( \max(n-2, 0) \) steps, which is \( \Omega(n) \).

**Upper bound**

Let \( n \in \mathbb{N} \), and let \( \text{lst} \) be the list of length \( n \) that contains only 0’s. In this case, the conditions of Loops 2 and 3 will always be false, because \( \text{lst}[j+2] == \text{lst}[j] \) for every \( j \). So in this case, Loops 2 and 3 run for zero iterations for every iteration of Loop 1.

To analyse the overall running time of Loop 1 for this choice of \( \text{lst} \):

- Loop 1 runs exactly \( \max(n-2, 0) \) iterations.
- Each iteration takes 1 step.

So the total cost is \( \max(n-2, 0) \) steps, which is \( \Theta(n) \), leading to an upper bound on the best-case running time of \( O(n) \).
4. [0 marks] An average-case analysis.

Updated March 21: because of the lecture scheduling change on March 18, we have decided not to grade this question, and instead will only be counting the first three questions on this problem set. We encourage you to complete this question as extra practice and preparation for the final exam, but please do not include it in your problem set submission.

Note: we’ll introduce this type of analysis with an example similar to this problem on Monday, March 18. We recommend prioritizing the other problems on this problem set until then.

Consider the following algorithm.

```python
def find_one_or_two(lst: List[int]) -> Optional[int]:
    """Return the index of the first 1 or 2 in lst, or None if neither of them appear."""
    for i in range(len(lst)):
        if lst[i] == 1 or lst[i] == 2:
            return i
    return None
```

We consider the following inputs for this function: for every \( n \in \mathbb{N} \), we define the set \( \mathcal{I}_n \) to be the set of lists of length \( n \) that are permutations of the numbers \( \{1, 2, \ldots, n\} \). We know that \( |\mathcal{I}_n| = n! \) (where \( n! = n \times (n-1) \times \cdots \times 2 \times 1 \)).

Note that if \( n \geq 1 \), then every list in \( \mathcal{I}_n \) contains a 1 or 2, and so the algorithm `find_one_or_two` will always return an index.

For this question, you can use your answer for each part in all subsequent parts.

(a) [0 marks] Let \( n \in \mathbb{N} \), and assume \( n \geq 2 \). Let \( i, j \in \mathbb{N} \), and assume \( i \neq j \), and that both are between 0 and \( n - 1 \), inclusive.

Find a formula, in terms of \( n, i, \) and/or \( j \), for the exact number of lists \( \texttt{lst} \) in \( \mathcal{I}_n \) where \( \texttt{lst}[i] \) equals 1 and \( \texttt{lst}[j] \) equals 2. Prove that your formula is correct.

**Solution**

Fix \( \texttt{lst}[i] = 1 \) and \( \texttt{lst}[j] = 2 \). We need to count the ways to put the \( n - 2 \) numbers \( \{3, 4, \ldots, n\} \) into \( \texttt{lst} \).

There are \( n - 2 \) possible values for the first list element other than \( i \) or \( j \), \( n - 3 \) possible values for the second list element other than \( i \) or \( j \), etc. In total, there are \( (n-2) \times (n-3) \times \cdots \times 1 = (n-2)! \) lists with \( \texttt{lst}[i] = 1 \) and \( \texttt{lst}[j] = 2 \).

(b) [0 marks] Let \( n \in \mathbb{N} \), and assume \( n \geq 2 \). Let \( i \in \mathbb{N} \), and assume \( i \) is between 0 and \( n - 1 \), inclusive.

Find a formula, in terms of \( n \) and/or \( i \), for the exact number of lists \( \texttt{lst} \) in \( \mathcal{I}_n \) where \( \texttt{lst}[i] \) equals 1, and 2 appears in \( \texttt{lst} \) at an index greater than \( i \).

**Solution**

Fix \( \texttt{lst}[i] = 1 \). There are \( (n - 1 - i) \) indices \( j > i \). By part (a), for each such index there are \( (n-2)! \) lists with \( \texttt{lst}[i] = 1 \) and \( \texttt{lst}[j] = 2 \). In total, there are \( (n - 1 - i) \cdot (n-2)! \) lists with \( \texttt{lst}[i] = 1 \).

(c) [0 marks] Let \( n \in \mathbb{N} \), and assume \( n \geq 2 \). Find an exact expression for the average running time of `find_one_or_two` on \( \mathcal{I}_n \).
Count each iteration of the loop as just 1 step, and include the last partial iteration (the one that executes `return i`) in your count. Note that for this set of inputs, the loop will always return early, so the final `return None` will never execute.

You may use the following formulas in your calculation, valid for all \( m \in \mathbb{N} \):

\[
\sum_{i=1}^{m} i = \frac{m(m + 1)}{2}
\]
\[
\sum_{i=1}^{m} i^2 = \frac{m(m + 1)(2m + 1)}{6}
\]

**HINT:** you can use your work from part (b), but make sure in your analysis here to consider the cases where 2 appears before 1 in the list.

---

**Solution**

We group the input lists based on the index of the *first* occurrence of 1 or 2 in the list. By the same reasoning as in the previous part, there are \((n - 1 - i) \cdot (n - 2)!\) different lists where \( \text{lst}[i] = 2 \) and 1 appears at an index greater than \( i \). So for each value of \( i \) between 0 and \( n - 1 \), there are \( 2 \cdot (n - 1 - i) \cdot (n - 2)! \) lists for which \( i \) is the first index where a 1 or 2 appears in the list. For these lists, Loop 1 iterates \( i + 1 \) times (including the loop iteration that terminates the loop by executing `return i`), with 1 step per iteration. For each \( i \), we let \( S_i \subset I_n \) be the set of lists whose first occurrence of 1 or 2 is at index \( i \).
So the average running time is equal to

\[
\frac{\sum_{\text{lst} \in I_n} \text{running time of } \text{find_one_or_two(lst)}}{|I_n|} = \frac{\sum_{\text{lst} \in I_n} \text{running time of } \text{find_one_or_two(lst)}}{n!}
\]

\[
= \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{\text{lst} \in S_i} \text{running time of } \text{find_one_or_two(lst)}
\]

\[
= \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{\text{lst} \in S_i} (i + 1)
\]

\[
= \frac{1}{n!} \sum_{i=0}^{n-1} (i + 1)(2 \cdot (n - 1 - i)(n - 2) !)
\]

\[
= \frac{2}{n(n - 1)} \sum_{i=0}^{n-1} (i + 1)(n - i - 1)
\]

\[
= \frac{2}{n(n - 1)} \sum_{i=0}^{n-1} (n(i + 1) - (i + 1)^2)
\]

\[
= \left( \frac{2}{n - 1} \sum_{i=0}^{n-1} (i + 1) \right) - \frac{2}{n(n - 1)} \sum_{i=0}^{n-1} (i + 1)^2
\]

\[
= \left( \frac{2}{n - 1} \sum_{i' = 1}^{n} i' \right) - \frac{2}{n(n - 1)} \sum_{i' = 1}^{n} i'^2
\]

\[
= \frac{2}{n - 1} \cdot \frac{n(n + 1)}{2} - \frac{2}{n(n - 1)} \cdot \frac{n(n + 1)(2n + 1)}{6}
\]

\[
= \frac{n(n + 1)}{n - 1} - \frac{(n + 1)(2n + 1)}{3(n - 1)}
\]

\[
= \frac{(n + 1)(3n - 2n - 1)}{3(n - 1)}
\]

\[
= \frac{n + 1}{3}
\]