CSC165 - Problem Set 4

Junaid Arshad, Sadeed Ahmed, & Frederick Meneses

April 8, 2022

1 Nested Loops

Consider the following algorithm:

```python
def twisty_too(n: int) -> None:
    """Precondition: n > 0."""
    i = n
    while i > 0:            # Loop 1
        s = -1
        j = 0
        while j < i:        # Loop 2
            s = s + 2
            j = j + s
        i = i - 1
        while i % 4 > 0:   # Loop 3
            i = i - 1
```

(a) The lower bound on the number of iterations of Loop 1, as a function of the input n, is shown below:

Proof.

- Lower bound on the number of iterations corresponds to the maximum change for the loop variable i.
- The maximum that the value of the loop variable i can change is by 4. i.e., if i starts off as a number that is divisible by 4 then before going into loop 3, its value will decrease once and then after that loop 3 will iterate an additional 3 times until the value of i is again divisible by 4 and we go back to loop 1. Thus, the value of i will decrease by at most 4 and thus the loop will keep running until the loop condition becomes false.
- Since i starts off as n, the upper bound on the value of i after k iterations (i_k) is $n - 4k$. Moreover, the loop condition becomes false when:

$$i_k \leq 0$$

- Thus the lower bound of the number of iterations of loop 1 is:

$$n - 4k \leq 0$$
$$k \geqslant \frac{n}{4}$$

□

(b) The upper bound on the number of iterations of Loop 1, as a function of the input n, is shown below:

Proof.

- Upper bound on the number of loop iterations corresponds to the minimum possible change in the loop variable i.
- The minimum change in the loop variable i is also the same as the maximum change.
  - The case when i is divisible by 4 , the lower bound on the value of the loop variable i after k iterations is equal to the upper bound found in part (a)
  - When the remainder of i divided by 4 is 1 then in the first iteration of loop 1, i only decreases by 1 and starting in the second iteration of loop 1, it follows the same behaviour as described in part (a) because it's now divisible by 4.
  - When the remainder of i after division by 4 is 2, then after the first iteration of loop 1 its value decreases by 2 but in every subsequent loop after that its value decreases by 4 as i now becomes divisible by 4.
  - Similarly for when remainder of i after division by 4 is 3, in this case the value of i decreases by 3 in the first iteration of loop 1 and then in every subsequent iteration, it decreases by 4.
- Thus, the lower bound on value of the loop variable i after k iterations is also $i_k = n - 4k$ The upper bound on the number of iterations then is:

$$\begin{aligned} i_k &\leq 0 \\ n - 4k &\leq 0 \\ k &\geqslant \frac{n}{4} \end{aligned}$$

□

(c) The Theta bound on the running time function RT(n) for algorithm twisty_too is shown below:

Proof.
Loop 2:

- Loop body takes 1 steps
- Loop stops when the value of j after k iterations, j_k, becomes $\geq i$. Looking at the values of j_k we notice that, $j_0 = 0, j_1 = 1, j_2 = 4, j_3 = 9, j_4 = 16, ..., j_k = k^2$. Since loop stops when $j_k \geq i$, we get that:

$$\begin{aligned} k^2 &\geq i \\ k &\geq \sqrt{i} \end{aligned}$$

- Thus the loop iterates at least $k = \lceil \sqrt{i} \rceil$ times and the total cost for this loop is $= \lceil \sqrt{i} \rceil$

Loop 3:

- Loop body takes: 1 step
- Loop iterations: The loop runs ≤ 3 times so no matter the input, the number of iterations is always a constant
- Therefore, the total cost of loop 3 $\in \Theta(1)$

Loop 1:

- Loop body takes: $\sqrt{i} + 1$ steps. This is because both loop 2 and loop 3 are inside loop 1 and since they are executed in sequence, we just add up the total cost of each loop.
- Loop iterations: From our answer to part (a), we know that the upper and lower bound on the number of loop iterations of loop 1 is $\frac{n}{4}$. Moreover, we also know that the as loop 1 iterates, the value of i goes from n all the way down to 1.
- To obtain the total cost, we have to add up the cost of each iteration of loop 1, as i goes from 1 to n:

$$\sum_{i=1}^{n}(\sqrt{i} + 1)$$

For the upper bound:

- We will show that the expression above has a matching upper and lower bound of $n^{3/2}$

$$\begin{aligned}
\sum_{i=1}^{n}(\sqrt{i} + 1) &\leq \sum_{i=1}^{n}(\sqrt{n} + 1) && \text{(using the fact that } \sqrt{i} \leq \sqrt{n}\text{)} \\
&= \sum_{i'=0}^{n-1}(\sqrt{n} + 1) && \text{(change of indices)} \\
&= (\sqrt{n})(n-1) + 1(n-1) \in \mathcal{O}(n^{3/2})
\end{aligned}$$

- Thus, the upper bound on the expression for the total cost is $n^{3/2}$.

For the lower bound:

$$\sum_{i=1}^{n}(\sqrt{i} + 1) = \sum_{i=1}^{\lceil \frac{n}{2} \rceil}(\sqrt{i} + 1) + \sum_{i=\lceil \frac{n}{2} \rceil + 1}^{n}(\sqrt{i} + 1)$$

- Using the fact that for the second summation: $i \geq \lceil \frac{n}{2} \rceil + 1 \geq \frac{n}{2} \Rightarrow \sqrt{i} \geq \sqrt{\frac{n}{2}}$

$$\begin{aligned}
\sum_{i=1}^{\lceil \frac{n}{2} \rceil} (\sqrt{i}+1) + \sum_{i=\lceil \frac{n}{2} \rceil+1}^{n} (\sqrt{i}+1) &\geq \sum_{i=1}^{\lceil \frac{n}{2} \rceil} (1) + \sum_{i=\lceil \frac{n}{2} \rceil+1}^{n} \left(\sqrt{\frac{n}{2}} + 1 \right) \\
&= \left\lceil \frac{n}{2} \right\rceil + \sum_{i'=0}^{n-(\lceil \frac{n}{2} \rceil+1)} \left(\sqrt{\frac{n}{2}} + 1 \right) \qquad \text{(change of index)} \\
&= \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil \left(n - \left(\left\lceil \frac{n}{2} \right\rceil + 1 \right) \right) \in \Omega(n^{3/2})
\end{aligned}$$

- Therefore, since the total running time function $RT(n)$ for algorithm `twisty_too` is in $\mathcal{O}(n^{3/2})$ and in $\Omega(n^{3/2})$, we can conclude that it is also in $\Theta(n^{3/2})$

$\square$

2 Worst-case and Best-case

Consider the following algorithm:

```
1  import matplotlib.pyplot as plt
2  from math import ceil
3
4
5  def long_prod(lst: list, t: int) -> int:
6      """Return the maximum length of any slice of lst whose product is at most t.
7      Preconditions: t > 0; lst is non-empty; every element of lst is positive.
8      """
9      m = 0  # max length found so far
10     for i in range(1, len(lst) + 1):        # Loop 1
11         j = i - 1
12         p = 1  # product of lst[j+1:i]
13         while j >= 0 and p * lst[j] <= t:  # Loop 2
14             p = p * lst[j]
15             j = j - 1
16         j = j + 1
17         if i - j > m:
18             m = i - j
19     return m
20
21
22 if __name__ == "__main__":
23     xs = list(range(200))
24     lst = []
25     for _ in lst:
26         lst.append(200**(1/3))
27     ys = [long_prod(lst, ceil(x ** ((1/3) ** (x ** (1/3))))) for x in xs]
28     ys2 = [x**(4 / 3) for x in xs]
29
30     plt.plot(xs, ys, label='true')
31     plt.plot(xs, ys2, label='expected')
32     plt.legend()
33     plt.show()
```

(a) Since loop 1 runs exactly n iterations, in order for the running time to be in $\Theta(n^{4/3})$, it must be that loop 2 runs in approximately $\sqrt[3]{n}$ steps.

Proof. Let $n \in \mathbb{N}$ and assume that $n \geq 2$.
Let `lst` be a list of length n where all the elements are equal to $\sqrt[3]{n}$. i.e.,
`lst = [n ** 1/3, n ** 1/3, ..., n ** 1/3]`. Let $t = \left\lceil \sqrt[3]{n}^{\sqrt[3]{n}} \right\rceil$, where $t \in \mathbb{Z}^+$.

`Loop 1` will run for exactly n iterations as the value of the loop variable i goes from 1 to n inclusive, as seen in Line 6.

`Loop 2` has two stopping conditions:

i) `j >= 0`

ii) `p * lst[j] <= t`

The first stopping condition (`j >= 0`) for `Loop 2` is dependent on i since $j_k = i - 1 - k < 0$.

`Loop 2` will run for i iterations if the value of $i < \lceil \sqrt[3]{n} \rceil$. In this case, the second stopping condition, `p * lst[j] <= t` will remain `True` because $\sqrt[3]{n}$ will be multiplied with itself less than $\sqrt[3]{n}$ times so will always remain $\leq t$. Thus, loop 2 will end once the first stopping condition, `j >= 0` becomes `False`.

Thus, after $\lceil \sqrt[3]{n} \rceil + 1$ elements in the list have been parsed, `Loop 2` will always iterate $\sqrt[3]{n}$ times. This is because the first loop condition will remain `True` long enough for the second loop condition to now evaluate to `False`. More explicitly, this is because $\sqrt[3]{n}$ will be multiplied by itself $\sqrt[3]{n}$ times, and since $t = \sqrt[3]{n}^{\sqrt[3]{n}}$, Loop 2 will run for no more than $\sqrt[3]{n}$ iterations.

Mathematically, the total cost of this algorithm for the specified input family can be expressed as:

$$
\begin{aligned}
&= \sum_{i=1}^{\lceil \sqrt[3]{n} \rceil} i + \sum_{i=\lceil \sqrt[3]{n} \rceil + 1}^{n+1} \sqrt[3]{n} \\
&= \frac{\sqrt[3]{n}(\sqrt[3]{n}+1)}{2} + \sum_{i'=0}^{n-\lceil \sqrt[3]{n} \rceil} (\sqrt[3]{n}) && \text{(change of indices)} \\
&= \frac{n^{2/3} + n^{1/3}}{2} + \sqrt[3]{n}(n - \sqrt[3]{n}) \\
&= \frac{n^{2/3} + n^{1/3}}{2} + n^{4/3} - n^{2/3} \in \Theta(n^{4/3})
\end{aligned}
$$

Therefore, the running time of `long_prod` with our specified input family is $\Theta(n^{4/3})$. □

(b) The upper-bound on the worst-case running time of `long_prod`, with proof, is shown below:

Proof. Let $n \in \mathbb{N}$. Let $x =$ `(lst, t)` be an arbitrary input family where the size of the input list `lst` is length n; every element of `lst` is positive; $t \in \mathbb{Z}^+$

Loop 2 runs at most i times for each iteration of Loop 1. Assuming that the second condition in Loop 2 is `True`, the loop keeps running until $j < 0$. Since the value of j starts from $i - 1$ and goes down to 0, the number of times Loop 2 iterates is at most i.

Loop 1 iterates at most $n + 1 - 1 = n$ times. The value of i goes from 1 to n and since the loop body depends on i, the total number of iterations is:

$$
\begin{aligned}
RT_{long\_prod}(x) &= \sum_{i=1}^{n} (i+2) \\
&= \frac{n(n+1)}{2} \in \mathcal{O}(n^2)
\end{aligned}
$$

This result shows $WC_{\texttt{long\_prod}}(\texttt{lst}, t) \in \mathcal{O}(n^2)$ □

(c) The lower-bound on the worst-case running time of `long_prod`, with proof, is shown below:

Proof. Let $n \in \mathbb{N}$. Let `t = 2` and let `lst` be a list of length n where all the elements are equal to 1 i.e., `lst = [1, 1, ..., 1]`. In this case, Loop 2 iterates i times for each iteration of Loop 1. This is because the second loop condition in Loop 2 (`...p * lst[j] <= t`) is always `True` since no matter how many elements of `lst` are multiplied with each other, they are always $\leq t$. Therefore, loop 2 iterates until `j < 0` becomes `True`.
The value of j starts from $i-1$ and goes down by 1 for each iteration of Loop 2. Thus, loop 2 iterates i times.
Moreover, Loop 1 iterates n times i.e., the value of i goes from 1 to n, therefore the total number of iterations in this case is:

$$\begin{aligned} RT_{long\_prod}(\texttt{lst, t}) &= \sum_{i=1}^{n} i \\ &= \frac{n(n+1)}{2} \in \Omega(n^2) \end{aligned}$$

This result shows $WC_{\texttt{long\_prod}}(\texttt{lst}, t) \in \Omega(n^2)$.

Since our upper bound and lower bound are the same expression, this analysis is sufficient to conclude: $WC_{long\_prod}(\texttt{lst, t}) \in \Theta(n^2)$ □

(d) The tight-bound on the best-case running time of `long_prod`, with proof, is shown below:

Proof. **Upper bound on the Best Case Running Time:**

Let $n \in \mathbb{N}$. Let `t = 1`, and let `lst` be a list of length n where all the elements are equal to 2 i.e., `lst = [2, 2, ..., 2]`.

In this case, Loop 2 doesn't iterate at all because the second condition (`...p * lst[j] <= t`) is always `False`. The reason it's always `False` is because since `p = 1`, the only value `p * lst[j]` can have is 2, which is `> t = 1`
Loop 1 iterates at most n times and since the loop body takes constant time (independent of input size) then the total number of iterations in this case is: $n \cdot 1 = n$ iterations.

Thus $BC_{long\_prod}(\texttt{lst, t}) \in \mathcal{O}(n)$

Lower bound on the Best Case Running Time:

Let $n \in \mathbb{N}$. Let `(lst, t)` be any input family where the size of the input list `lst` is n; every element of `lst` is positive; $\texttt{t} \in \mathbb{Z}^+$. For all input lists, Loop 1 runs at least n iterations and the loop body (including Loop 2) takes at least 1 step. Therefore, the total number of iterations is: $n \cdot 1 = n$

Thus $BC_{long\_prod}(\texttt{lst, t}) \in \Omega(n)$

This is sufficient to conclude $BC_{long\_prod}(\texttt{lst, t}) \in \Theta(n)$

□

3 Average-Case Analysis

Consider the following algorithm:

```python
def alpha_min(s: str) -> int:
    """Return the smallest index k such that s[k:len(s)] is sorted.
    Precondition: s is non-empty. """
    i = len(s) - 1
    while i > 0 and s[i-1] <= s[i]:
        i = i - 1
    return i
```

For each $n \in \mathbb{N}$ with $n \geq 2$, let $\mathcal{I}_n$ be the set that contains all strings of length n with 2 b's and $(n-2)a$'s, in any order. (For example, $\mathcal{I}_4 = \{aabb, abab, abba, baab, baba, bbaa\}$.)
Note that $|\mathcal{I}_n| = \binom{n}{2} = \frac{n(n-1)}{2}$ because each element of $\mathcal{I}_n$ is made up of n individual characters, all but two of which are equal to a, and there are exactly $\binom{n}{2}$ many different ways to choose the 2 positions that will contain b.

(a) Let $n \in \mathbb{N}$ with $n \geq 2$, and let k be the value returned by `alpha_min`(s), for some input $s \in \mathcal{I}_n$. An expression for the "exact" number of steps executed by `alpha_min`(s) is:

- Let s be str of length n belonging to $\mathcal{I}_n$. By definition, s has 2 b's and $(n-2)$ a's.
- Value to be returned, k, starts from $n-1$ (last *str* element) and decreases by 1 every time the element before it is $\leq$ element at k i.e, everytime the loop runs.
- If the entire str is sorted then k goes down to 0 and `alpha_min`(s) iterates $n - 0 = n$ - k times.
- If the str is not sorted to begin with i.e., the last 2 elements of the str are '..ba' then the loop doesn't execute at all and in this case `alpha_min`(s) runs $n-(n-1) = n-k$ times where $k = n-1$ since it doesn't decrease at all.
- Thus, the exact number of iterations of `alpha_min`(s) are $n-k$.

(b) The expression for the exact average-case running time of `alpha_min` over the set of inputs $\mathcal{I}_4$ is calculated and shown below, in the form of a simplified, concrete rational number:

Let $s_1, s_2, s_3, s_4, s_5, s_6$ be the strings in the set $\mathcal{I}_4$ i.e.,

- $s_1 = aabb$: Here, str s_1 is sorted after $k = i = 0$ so $RT_{\texttt{alpha\_min}}(s_1) = n - k = 4 - 0 = 4$
- $s_2 = abab$: Here, str s_2 is sorted after $k = i = 2$ so $RT_{\texttt{alpha\_min}}(s_2) = n - k = 4 - 2 = 2$
- $s_3 = abba$: Here, str s_3 is sorted after $k = i = 3$ so $RT_{\texttt{alpha\_min}}(s_3) = n - k = 4 - 3 = 1$
- $s_4 = baab$: Here, str s_4 is sorted after $k = i = 1$ so $RT_{\texttt{alpha\_min}}(s_4) = n - k = 4 - 1 = 3$
- $s_5 = baba$: Here, str s_5 is sorted after $k = i = 3$ so $RT_{\texttt{alpha\_min}}(s_5) = n - k = 4 - 3 = 1$
- $s_6 = bbaa$: Here, str s_6 is sorted after $k = i = 2$ so $RT_{\texttt{alpha\_min}}(s_6) = n - k = 4 - 2 = 2$

- Let $AC(4)$ be the average of the runtimes over all inputs $s \in \mathcal{I}_4$.
- Let $RT(s)$ be the runtime of `alpha_min`(s) on str s.
- Given that $\mathcal{I}_4 = \{aabb, abab, abba, baab, baba, bbaa\}$, we know that $|\mathcal{I}_4| = 6$.

- By definition of average case, we know that:

$$\begin{aligned} AC(4) &= \frac{1}{|\mathcal{I}_4|} \sum_{s \in \mathcal{I}_4} RT(s) \\ &= \frac{RT(s_1) + RT(s_2) + RT(s_3) + RT(s_4) + RT(s_5) + RT(s_6)}{|\mathcal{I}_4|} \\ &= \frac{13}{6} \end{aligned}$$

(c) The exact expression for the number of inputs such that $s \in \mathcal{I}_n$ returns k, i.e., $|\{s \in \mathcal{I}_n |$ `alpha_min`(s) returns $k\}|$, is calculated below:

The value of k returned by `alpha_min`(s) is between 0 and $n-1$. For each $n \in \mathbb{N}$ and each $j \in \{1, 2, \ldots, n-2\}$, let $S_{n,j}$ denote the set of strings s where one of the b's occurs at position $j-1$.

The number of inputs for which $k = 0$, i.e. $|\{s \in \mathcal{I}_n |$ `alpha_min`(s) returns $0\}|$

There is only a single possible string for which the value of k returned is 0, which is the string s in which the last 2 elements are both b and the remaining elements are a.

Let $S_{n,0}$ be the string where one of the b's is at -1 and the other one is at -2. For this string, $k = 0$ since all the other elements are equal to a and the entire string is sorted. (i.e., $|S_{n,0}| = 1$).

The number of inputs for which $1 \leqslant k \leqslant n-2$, i.e., $|\{s \in \mathcal{I}_n |$ `alpha_min`(s) returns $k\}|$

For $n \in \mathbb{N}$ and for $j \in \{1, \ldots, n-2\}$, let $S_{n,j}$ denote the set of strings s where one of the b's occurs at index $j-1$.

- For $k = 1$, the number of possible strings are $|S_{n,1}| = 1$. This is because, with one of the b's at index 0, the only place that the second b can be at while still having the remaining string sorted is at index $n-1$.
- For $k = 2$, the number of possible strings are $|S_{n,2}| = 2$. This is because with one of the b's at index 1, the only places that the second b can be at while still having the remaining string sorted is at index 0 or $n-1$.
- For $k = 3$, the number of possible strings are $|S_{n,3}| = 3$. This is because with one of the b's at index 2, the only places that the second b can be at while still having the remaining string sorted is at index 0, 1, or $n-1$.

  $\vdots$
- For $k = j$, the number of possible strings are $|S_{n,j}| = j$. This is because with one of the b's at index $j-1$, the only places that the second b can be at while still having the remaining string sorted is at index 0, 1, …, $j-2$ or $n-1$.

The number of inputs for which $k = n-1$, i.e., $|\{s \in \mathcal{I}_n |$ `alpha_min`(s) returns $n-1\}|$

There are $n-2$ possible strings in this case. This is because $k = n-1$ is only possible when there's an a at index $n-1$ and a b at index $n-2$. This means the other b can be at index $0, 1, \ldots,$ or $n-3$ (n-2 possible positions). Thus, $n-2$ strings are possible for which $k = n-1$.

Let $S_{n,n-1}$ be the string s where one of the b's is at $n-2$ and $k = n-1$. We know $|S_{n,n-1}| = n-2$ by our observation above.

(d) The average-case analysis of `alpha_min` for the input set $\mathcal{I}_n$ (with an exact expression) is shown below:

$$\begin{aligned}
AC(n) &= \frac{1}{|\mathcal{I}_n|} \sum_{S_{n,j} \in \mathcal{I}_n} RT\left(S_{n,j}\right) \\
&= \frac{1}{\left(\frac{n(n-1)}{2}\right)} \sum_{j=0}^{n-1} |S_{n,j}| \cdot (n-k) \\
&= \frac{2}{n(n-1)} \left(|S_{n,0}| \cdot (n-0) + \sum_{j=1}^{n-2} |S_{n,j}| \cdot (n-k) + |S_{n,n-1}| \cdot (n-k) \right) \\
&= \frac{2}{n(n-1)} \left((1 \times n) + \sum_{j=1}^{n-2} \big(j \cdot \big(n-j\big)\big) + (n-2) \cdot (n-(n-1)) \right) \\
&= \frac{2}{n(n-1)} \left(n + \sum_{j=1}^{n-2} n \cdot j - \sum_{j=1}^{n-2} j^2 + (n-2) \right)
\end{aligned}$$

Using the given hint, we know that:

$$\begin{aligned}
&= \frac{2}{n(n-1)} \left(2n - 2 + n \left(\frac{(n-2)(n-2+1)}{2} \right) - \frac{(n-2)(n-2+1)(2(n-2)+1)}{6} \right) \\
&= \frac{2}{n(n-1)} \left(2(n-1) + n \left(\frac{(n-2)(n-1)}{2} \right) - \frac{(n-2)(n-1)(2n-3)}{6} \right) \\
&= \frac{4}{n} + (n-2) - \left(\frac{(n-2)(2n-3)}{3n} \right)
\end{aligned}$$