Learning Objectives

By the end of this worksheet, you will:

- Analyse the running time of functions containing loops with varying increments.
- Analyse the worst-case and best-case running time of functions.

**Note:** we’ve only written out questions about worst-case running time here; however, both of the algorithms given on this worksheet are very good exercises in analysing best-case running time as well.

1. **A more careful analysis.** Recall this function from lecture:

```python
def f(n):
    x = n
    while x > 1:
        if x % 2 == 0:
            x = x / 2
        else:
            x = 2*x - 2
```

We argued that for any positive integer value for `x`, if two loop iterations occur then `x` decreases by at least one. This led to an upper bound on the running time of $O(n)$, but it turns out that we can do better.

(a) First, prove that for any positive integer value of `x`, if three loop iterations occur then `x` decreases by at least a factor of 2. Note: this is an exercise in covering all possible cases; it’s up to you to determine exactly what those cases are in your proof.

**Solution**

**Proof.** Let $x \in \mathbb{Z}^+$. We’ll divide up the proof into four different cases, depending on which powers of 2 divide $x$.

**Case 1:** assume $x$ is odd.
In this case, the values of $x$ set by three iterations are:

$$x \to 2x - 2 \to x - 1 \to \frac{x}{2} - 1$$

In the third iteration, we note that because $x$ is odd, $x - 1$ is even, and so this quantity is divided by 2.

**Case 2:** assume $2 \mid x$ and $4 \nmid x$.
In this case, the values of $x$ set by three iterations are:

$$x \to \frac{x}{2} \to x - 2 \to \frac{x}{2} - 1$$

In the second iteration, since $4 \nmid x$, $\frac{x}{2}$ is odd, and so this iteration executes the `else` branch. Then, because $x$ is even, $x - 2$ is also even, and so in the third iteration $x - 2$ is divided by 2.

**Case 3:** assume $4 \mid x$ and $8 \nmid x$.
In this case, the values of $x$ set by three iterations are:

$$x \to \frac{x}{2} \to \frac{x}{4} \to \frac{x}{2} - 2$$

Now $\frac{x}{2}$ is even, so we divided by 2 again. Then because we assumed $8 \nmid x$, we know that $\frac{x}{4}$ is odd, so the `else` branch occurs.

---

1 We phrase this as a conditional because it might be the case that the loop stops after fewer than 2 iterations.
Case 4: assume $8 \mid x$.

In this case, the values of $x$ set by three iterations are:

$$x \rightarrow \frac{x}{2} \rightarrow \frac{x}{4} \rightarrow \frac{x}{8}$$

In all four cases, the resulting value after three iterations is $\leq \frac{x}{2}$; that is, in all four cases, the value of the variable $x$ has decreased by a factor of 2.

(b) For every $k \in \mathbb{N}$, let $x_k$ be the value of the variable $x$ after $3^k$ loop iterations, in the case when $3^k$ iterations occur. Using part (a), find an upper bound on $x_k$, and hence on the total number of loop iterations that will occur (in terms of $n$). Finally, use this to determine a better asymptotic upper bound on the runtime of $f$ than $O(n)$.

Solution

We showed in part (a) that after 3 iterations, the current value of $x$ decreases by at least a factor of 2, or the loop has terminated. So then for any $k$, either the loop terminates within $3^k$ iterations, or the value of $x$ has decreased by at least a factor of $2^k$. Since $x$ is initialized to $n$, we know that $i_k \leq \frac{n}{2^k}$.

The loop terminates when $i \leq 1$, and this occurs when $2^k \geq n$, i.e., $k \geq \lfloor \log n \rfloor$. So then the loop will run for at most $3 \cdot \lfloor \log n \rfloor$ iterations; since each iteration takes constant time, the total runtime is $O(\log n)$.

2. Worst-case analysis. Let $L$ be a list of numbers. Consider the following function, which takes in a list of numbers and determines whether the list contains any duplicates.

```python
def has_duplicate(L):
    n = len(L)
    for i in range(n):  # i goes from 0 to n-1
        for j in range(i + 1, n):  # j goes from i+1 to n-1
            if L[i] == L[j]:
                return True
    return False
```

(a) Find a good upper bound on the worst-case running time of this function.

Solution

For a fixed iteration of the outer loop, the inner loop runs at most $n - 1 - i$ iterations (if it doesn’t stop early), with each iteration taking constant time (1 step). The outer loop runs at most $n$ iterations, for $i$ going from 0 to $n - 1$. The total cost is

$$\sum_{i=0}^{n-1}(n - 1 - i) = n(n - 1) - \sum_{i=0}^{n-1}i = \frac{n(n - 1)}{2}$$

So the total number of steps of this algorithm is at most $\frac{n(n - 1)}{2}$ (ignoring the +1 for the cost of the final return), which is $O(n^2)$.

(b) Prove a matching lower bound on the worst-case running time of this function, by finding an input family whose asymptotic runtime matches the bound you found in the previous part. For an extra challenge, find an input family for which this function does return early (i.e., the return on line 6 executes), but the runtime is still Theta of the upper bound you found in the previous part.

Solution

There’s actually several possibilities here! One is that the input list contains no duplicates, except that the last two elements ($L[n-2]$ and $L[n-1]$) are equal. In this case, all iterations of the outer loop occur except the last one: when $i = n - 2$ and $j = n - 1$, the inner loop returns early. The running time in this
case is

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

This runtime is \( \Theta(n^2) \), matching the bound from part (a).

(c) Find an input family whose running time is \( \Theta(n) \), where \( n \) is the length of the input list, and analyse the running time of \texttt{has_duplicate} on this input family. [Note that \( \Theta(n) \) is neither the worst-case nor best-case running time!]
3. **String matching.** Here is an algorithm which is given two strings, and determines whether the first string is a substring of the second. (In Python, this would correspond to the `in` operation, e.g. "oof" in "proofs are fun"). You may assume here that the length of the second string is equal to the square of the length of the first string, and both strings are non-empty.

```python
def substring(s1, s2):
    """Precondition: len(s2) = len(s1) * len(s1)."""
    i = 0
    while i < len(s2) - len(s1):
        # Check whether s1 == s2[i..i+len(s1)-1]
        match = True
        for j in range(len(s1)):
            # If the current corresponding characters don't match,
            # stop the inner loop.
            if s1[j] != s2[i + j]:
                match = False
                break

        # If a match has been found, stop and return True.
        if match:
            return True
        i = i + 1
    return False
```

(a) Let \( n \) represent the length of \( s_1 \) (and so the length of \( s_2 \) is \( n^2 \)). Find a good asymptotic upper bound on the worst-case running time of this function in terms of \( n \).

**Solution**

Let’s analyze the running time of this function assuming there are no early returns. In this case, for a fixed iteration of the outer loop, the inner loop takes \( n \) iterations, and hence this many steps (since each iteration takes constant time). The outer loop runs for \( n^2 - n \) iterations, for a total cost of \( n(n^2 - n) \), which is \( O(n^3) \).

(b) Now find an input family whose running time matches the upper bound you found in part (a). While you may be able to describe the input family in the space below, you’ll certainly need extra paper to perform the analysis correctly.

**Hint:** you can pick \( s_1 \) to be a string of length \( n \) that just repeats the same character \( n \) times.

**Solution**

This input family is rather tricky to describe and analyse properly. Let \( n \in \mathbb{Z}^+ \), and let \( s_1 \) be the string of length \( n \) that only contains the character ‘a’, and let \( s_2 \) be the string of length \( n \) defined as:

\[
s_2[i] = \begin{cases} 
  b, & \text{if } n \mid i + 1 \\
  a, & \text{otherwise}
\end{cases}
\]

For example, when \( n = 4 \), we have

\[
s_1 = aaaa \text{ and } s_2 = aaabaabaab
\]

Intuitively, since \( s_1 \) and \( s_2 \) are so similar, the inner loop has to run for many iterations until it finds a mismatch.

We leave the analysis of the running time of `substring` on this input family as an exercise, with one hint: the outer loop will run \( n^2 - n \) times in total; rather than trying to sum up over all of these iterations, break it up into \( n - 1 \) groups of \( n \) consecutive iterations. You should find that the running time of the
first \( n \) iterations (from \( i = 0 \) to \( n - 1 \)) is more straightforward to analyse, and each subsequent group of \( n \) iterations has the same cost.