Learning Objectives

By the end of this worksheet, you will:

- Analyse the running time of loops whose loop counter changes differently in different iterations.
- Analyse the running time of functions that call helper functions.

1. Varying loop increments. In lecture, we saw one example of a loop where the change to the loop variable was not the same on each iteration. In this question, you’ll get some practice analyzing such loops yourself using a general technique, which works when the loop condition is of the form $i < x$, where $i$ is the loop variable and $x$ is some expression (e.g., $n$ or $n^2$). There are four steps:

   (i) Identify the minimum and maximum possible change for the loop variable in a single iteration.
   
   (ii) Use this to determine formula for an exact lower bound and upper bound on the value of the loop variable after $k$ iterations, say $f_1(k) ≤ i_k ≤ f_2(k)$.
   
   (iii) Use these formulas and the loop condition to bound the exact number of loop iterations that will occur:
   
   - For an upper bound on the number of iterations: find the smallest value $k_1 ∈ \mathbb{N}$ that makes $f_1(k_1) ≥ x$.
     Since $i_{k_1} ≥ f_1(k_1)$, the loop must stop after at most $k_1$ iterations.
   
   - For a lower bound on the number of iterations: find the smallest value $k_2 ∈ \mathbb{N}$ that makes $f_2(k_2) ≥ x$.
     Then at least $k_2$ iterations must occur before the loop stops.
   
   One subtlety with this technique is that a lower bound on $i_k$ determines an upper bound on the number of iterations, and an upper bound on $i_k$ determines a lower bound on the number of iterations.

   (iv) Conclude simple Big-Oh and Omega bounds on the running time. If you find the same expression for Big-Oh and Omega, then you can also conclude a Theta bound.

(a) Note: we’ve omitted the conditions in the if, but assume they are constant time checks.

```python
def varying1(n: int) -> None:
    i = 0
    while i < n:
        if ...:
            i = i + 1
        elif ...:
            i = i + 3
        else:
            i = i + 6
```

Solution

The minimum change in the loop is that $i$ increases by 1; the maximum change is that $i$ increases by 6. Let $i_k$ be the value of $i$ after $k$ iterations. The previous observation tells us that $k ≤ i_k ≤ 6k$ (these are the lower and upper bounds on $i_k$).

Part 1 (upper bound on running time).

We want to determine a good upper bound (“at most _____”) on the number of iterations that could occur before the loop stops. Since the loop terminates when $i ≥ n$, we want to find the smallest value of $k$ such that $i_k ≥ n$.

To do this, we use the lower bound on $i_k$: since we know that $i_k ≥ k$, if we pick $k = n$ then this implies $i_k ≥ n$. This means that the loop can run at most $n$ iterations. Since each iteration takes 1 step, this loop takes at most $n$ steps.

Counting 1 step for the line $i = 0$, the total number of steps is at most $n + 1$, which is $O(n)$.

1 Similarly, suppose we want to run 10km. A lower bound on our speed determines an upper bound on the time it will take to cover the distance, and vice versa.
Part 2 (lower bound on runtime).
Now we want to determine a good lower bound (“at least ______”) on the number of iterations that must occur before the loop stops. We find the smallest value of \( k \) that makes \( 6k \geq n \).
We can isolate \( k \) to obtain \( k \geq \frac{n}{6} \), and since \( k \) is an integer, we can conclude \( k \geq \left\lceil \frac{n}{6} \right\rceil \). So at least \( \left\lceil \frac{n}{6} \right\rceil \) iterations must occur.
Using a similar analysis as above, in total at least \( \left\lceil \frac{n}{6} \right\rceil + 1 \) steps occur, which leads to an asymptotic lower bound \( \Omega(n) \).

Note: since the asymptotic upper bound and lower bounds are the same, we can conclude that the overall running time of this function is \( \Theta(n) \).

(b)
```python
def varying2(n: int) -> None:
    i = 1
    while i < n:
        if n % i <= i/2:
            i = 2 * i
        else:
            i = 3 * i
```

Solution
The argument is the same as the previous one, except now \( i \) increases by at least a multiplicative factor of 2, and at most a factor of 3. This means that \( 2^k \leq i_k \leq 3^k \), and so the number of loop iterations is at most \( \lceil \log_2 n \rceil \) and at least \( \lceil \log_3 n \rceil \) [using the same reasoning as in part (a)].
That is, the upper bound on the running time here is \( O(\log_2 n) \), and the lower bound is \( \Omega(\log_3 n) \). Since we know that \( \log_3 n \in \Theta(\log_2 n) \), we can conclude that the tight bound on the running time is \( \Theta(\log n) \).

2. Helper functions. We have mainly analysed loops as the mechanism for writing functions whose running time depends on the size of the function’s input. Another source of non-constant running times that you often encounter are other functions that are used as helpers in an algorithm.
For this exercise, consider having two functions \( \text{helper1} \) and \( \text{helper2} \), which each take in a positive integer as input. Moreover, assume that \( \text{helper1} \)'s running time is \( \Theta(n) \) and \( \text{helper2} \) is \( \Theta(n^2) \), where \( n \) is the value of the input to these two functions.
Your goal is to analyse the running time of each of the following functions, which make use of one or both of these helper functions. When you count costs for these function calls, simply substitute the value of the argument of the call into the function \( f(x) = x \) or \( f(x) = x^2 \) (depending on the helper). For example, count the cost of calling \( \text{helper1}(k) \) as \( k \) steps, and \( \text{helper2}(2n) \) as \( 4n^2 \) steps.

(a)
```python
def f1(n: int) -> None:
    helper1(n)
    helper2(n)
```

Solution
The call to \( \text{helper1} \) takes \( n \) steps, and the call to \( \text{helper2} \) takes \( n^2 \) steps, for a total of \( n^2 + n \) steps. This is \( \Theta(n^2) \).

(b)
```python
def f2(n: int) -> None:
    i = 0
    while i < n:
        helper1(n)
        i = i + 2
    j = 0
    while j < 10:
        helper2(n)
        j = j + 1
```

**Solution**

The first loop takes $\lceil \frac{n}{2} \rceil$ iterations, and each iteration requires $n$ steps for the call to `helper1`. As with nested loops, we ignore the lower-order cost of the loop counter increment $i = i + 2$. So the total cost of this loop is $\lceil \frac{n}{2} \rceil \cdot n$.

The second loop runs for 10 iterations, and each iteration requires $n^2$ steps for the call to `helper2`. So we count the cost for this loop as $10n^2$.

The total cost is $\lceil \frac{n}{2} \rceil \cdot n + 10n^2$, which is $\Theta(n^2)$.

(c)

```python
def f3(n: int) -> None:
    i = 0
    while i < n:
        helper1(i)
        i = i + 1
    j = 0
    while j < 10:
        helper2(j)
        j = j + 1
```

**Solution**

The first loop takes $n$ iterations, but now the cost of the call to `helper1` changes at each iteration. For a fixed iteration of this loop, the cost of calling `helper1(i)` is $i$, and so the total cost over all iterations of this loop is $\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$ (note that this is the same as when we analysed one of the nested loop examples from lecture).

Similarly, the cost of the second loop is $\sum_{j=0}^{9} j^2$; this is one, however, is a constant cost with respect to $n$.

So the first loop has a running time of $\Theta(n^2)$, and the second has a running time of $\Theta(1)$. The overall running time is the sum of these two, which is $\Theta(n^2)$. 

Page 3/5
3. **A more careful analysis.** Recall this function from lecture:

```python
def f(n: int) -> None:
    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 \( x_0 \) be the starting value of \( x \), and \( x_1, x_2, \) and \( x_3 \) be the value of \( x \) after 1, 2, and 3 loop iterations, respectively. We want to prove that \( x_3 \leq \frac{1}{2} x_0 \). There are many ways of dividing this proof into cases based on whether these values are even/odd. One simple approach is to look at the remainders of \( x_0 \) when dividing by 8; this has a lot of cases, but the calculation required for each case is pretty straightforward. Here’s one as an example.

**Case ??:** assume \( x \) has remainder 5 when divided by 8, i.e., that there exists \( k \in \mathbb{Z} \) such that \( x_0 = 8k + 5 \). In this case, \( x_0 \) is odd, and so Line 7 executes in the first loop iteration, making

\[
x_1 = 2x_0 - 2 = 16k + 8 = 2(8k + 4)
\]

So then \( x_1 \) is even, and at the second loop iteration Line 5 executes, making

\[
x_2 = \frac{1}{2} x_1 = 8k + 4 = 2(4k + 2)
\]

So then \( x_2 \) is even, and at the third loop iteration Line 5 executes again, making

\[
x_3 = \frac{1}{2} x_2 = 4k + 2 = \frac{1}{2} x_0 - \frac{1}{2}
\]

Therefore \( x_3 \leq \frac{1}{2} x_0 \).

(b) For every \( k \in \mathbb{N} \), let \( x_k \) be the value of the variable \( x \) after \( 3k \) loop iterations, in the case when \( 3k \) 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 \( O(n) \).

(Note: you might need to write your analysis on a separate sheet of paper.)

**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 \( 3k \) 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 \( x_k \leq \frac{n}{2^k} \).

The loop terminates when \( x \leq 1 \), and this occurs when \( 2^k \geq n \), i.e., \( k \geq \lceil \log n \rceil \). So then the loop will run

\(^2\)We phrase this as a conditional because it might be the case that the loop stops after fewer than two iterations.
for at most $3 \cdot \lceil \log n \rceil$ iterations; since each iteration takes constant time, the total runtime is $O(\log n)$. 