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.
- Understand and express properties about the minimum and maximum of a set of numbers.

1. Varying loop increments. In lecture, we saw one (complicated) 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. For each of the following functions, do the following:

   (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. (E.g., “\( i_k \geq k \)” and “\( i_k \leq 2^k \)”.)

   (iii) Use these formulas and the loop condition to bound the exact number of loop iterations that will occur:

   - Use the lower bound on the loop variable (e.g., \( i_k \geq k \)) to find a value for \( k \) that guarantees that the loop condition is false. This gives you an upper bound on the possible number of iterations.
   - Use the upper bound on the loop variable (e.g., \( i_k \leq 2^k \)) to find a value of \( k \) that guarantees that the loop condition is true. This gives you a lower bound on the possible number of iterations.

   (iv) Use your upper and lower bounds on the number of iterations to find Big-Oh and Omega bounds on the running time of the function. Note that if you have the same expression for Big-Oh and Omega, then you can also conclude a Theta bound.

   ```python
   (a) def varying1(n: int) -> None:
        i = 0
        while i < n:
            if i % 3 == 0:
                i = i + 1
            elif i % 2 == 1:
                i = i + 3
            else:
                i = i + 6
   ```

---

1 This works whenever the loop condition is of the form \( i < \_\_\_ \), where \( i \) is the loop variable.
(b) `def varying2(n: int) -> None:
    i = 1
    while i < n:
        if n % i <= i/2:
            i = 2 * i
        else:
            i = 3 * i`

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 `helper1` and `helper2`, which each take in a positive integer as input. Moreover, assume that `helper1`’s running time is \( \Theta(n) \) and `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 `helper1(k)` as \( k \) steps, and `helper2(2*n)` as \( 4n^2 \) steps.

(a) `def f1(n: int) -> None:
    helper1(n)
    helper2(n)```
(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
```
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\(^2\). 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.

(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.)

\(^2\)We phrase this as a conditional because it might be the case that the loop stops after fewer than two iterations.