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 $f_1(k) \leq i_k \leq f_2(k)$.

   (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 $k_1 \in \mathbb{N}$ that makes $f_1(k_1) \geq x$. Since $i_{k_1} \geq f_1(k_1)$, the loop must stop after at most k_1 iterations.
   - For a *lower bound* on the number of iterations: find the smallest value $k_2 \in \mathbb{N}$ that makes $f_2(k_2) \geq x$. 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.[1]

   (iv) Conclude simple Big-O and Omega bounds on the running time. If you find the same expression for Big-O 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.

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

[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.

(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)

```
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
```

(c)

```
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
```

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

```
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 $\mathcal{O}(n)$, 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 $\mathcal{O}(n)$.

(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.