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:

```
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.[1] 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.

[1] We phrase this as a conditional because it might be the case that the loop stops after fewer than 2 iterations.

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

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.

```
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.

(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.

(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 `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.

```
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 `s1` (and so the length of `s2` is n^2). Find a good asymptotic upper bound on the worst-case running time of this function in terms of n.

(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 `s1` to be a string of length n that just repeats the same character n times.