Learning Objectives

By the end of this worksheet, you will:

- Determine the exact number of iterations of loops with a variety of loop counter behaviours.
- Find the asymptotic running time of programs containing loops.

---

1. **Loop variations**. Each of the following functions takes as input a *non-negative integer* and performs at least one loop. For each loop, determine the *exact* number of iterations that will occur (in terms of the function's input n), and then use this to determine the simplest Theta expression[1] for the running time of each function. You do *not* need to prove any "$g \in \Theta(f)$" statements here.

   Note: each loop body runs in $\Theta(1)$ time in this question. While this won't always be the case, such examples allow you to focus on just counting loop iterations here.

```python
def f1(n: int) -> None:
    i = 0
    while i < n:
        print(i)
        i = i + 5
```

ceiling (n/ 5) in Theta(n)

```python
def f2(n: int) -> None:
    i = 4
    while i < n:
        print(i)
        i = i + 1
```

[1] By "simplest," we mean ignoring constants and slower-growth terms. For example, write $\Theta(n)$ instead of $\Theta(2n + 0.3)$.

```python
def f3(n: int) -> None:
    """Precondition: n > 0."""
    i = 0
    while i < n:
        print(i)
        i = i + (n / 10)
```

n >= 10

i = i + floor(n/10)

10 in Theta(1)

Lemma:
i = a
while i < b:
    i = i +c
ceiling {(b-a)/c}

after 10 loops,
i = 10floor(n/10)
using Lemma:
ceiling{(n - 10* floor(n/10))/(floor(n/10))} + 10

```python
def f4(n: int) -> None:
    i = 20
    while i < n * n:
        print(i)
        i = i + 3
```

max(0, ceiling{(n^2 -20)/3}) in Theta(n^2)

```python
def f5(n: int) -> None:
    i = 20
    while i < n * n:
        print(i)
        i = i + 3

    j = 0
    while j < n:
        print(j)
        j = j + 0.01
```

max(0, ceiling{(n^2 -20)/3})

100n

max(0, ceiling{(n^2 -20)/3}) + 100n in Theta(n^2)

2. **Multiplicative increments**. Consider the following function:

```
def f(n: int) -> None:
    """Precondition: n > 0."""
    i = 1
    while i < n:
        print(i)
        i = i * 2
```

Even though this looks similar to previous examples, the fact that the loop variable `i` changes by a multiplicative rather than additive factor requires a more principled approach in determining the number of loop iterations.

(a) Let i_0 be the value of variable `i` when 0 loop iterations have occurred, i_1 be the value of `i` immediately after 1 loop iteration has occurred, and in general i_k be the value of `i` immediately after k loop iterations have occurred. For example, $i_0 = 1$ (the initial value of `i`), $i_1 = 2$, and $i_2 = 4$.
Determine the values of i_3, i_4, and a general formula for i_k.[2]

i3 = 8, i4 = 16, ik = 2^k

(b) Use your formula from part (a) to determine the exact number of loop iterations that occur, in terms of n.
HINT: Fnd the *smallest* value of k that makes the loop condition false.

(c) Determine the Theta running time for the function `f`.

(d) Why did we not initialize `i = 0` in this function?

[2]Of course, if n is small then not a lot of loop iterations occur. More formally, i_k represents the value of `i` after k loop iterations, *if* k iterations occur.

3. **A more unusual increment**. Consider the following function:

```python
def f(n: int) -> None:
    """Precondition: n >= 2."""
    i = 2
    while i < n:
        print(i)
        i = i * i
```

Analyse the running time of this function using the same technique as the previous question.

i0 =2
i1 = 2*2
i2 =2*2*2*2
i3 = 2*2*2*2*2*2*2*2
…
ik = 2^(2^k)

2^(2^k) >= n
2^k >= log(n)
k >= log(log(n))

running times: k = ceiling{log(log(n))}

in Theta(log(log(n)))