COMP 212 Spring 2025
Lab 4

The goal for this lab is to practice asymptotic analysis.

1 Big-O

Let f and g be two mathematical functions from the natural numbers to the natural numbers. We define f **is** $O(g)$ to mean

> there exist natural numbers k and l such that for all natural numbers x,
> $f(x) \leq (k \times g(x)) + l$

It is common to notate the functions $f(n)$ and $g(n)$ by just writing their bodies in terms of n. For example, $O(n^2)$ means $O(g)$ where $g(n) = n^2$.

For each of the following, decide whether it is true or false. If it is true, give k and l for which the inequality holds. If it is false, convince yourself that there are no such k and l.

- $3n$ is $O(4n)$
- $3n$ is $O(n)$
- $3n$ is $O(1)$
- $3n + 4$ is $O(n)$
- $2n^2 + 17n$ is $O(n^2)$
- $2n^2$ is $O(n)$
- $8n^2$ is $O(2^n)$

Have the course staff check your work before proceeding.

2 Analysis

2.1 Simple Fibonacci

The *Fibonacci sequence* is a sequence of numbers defined by the rule that the next number in the sequence is the sum of the previous two:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

This can be implemented by a recursive function with two recursive calls:

```
fun fib (n : int) : int =
    case n of
       ~1 => 0
      | 0 => 1
      | 1 => 1
      | _ => fib (n - 1) + fib (n - 2)
```

We're going to add the harmless but slightly strange base case defining the negative first element of the sequence to the definition as well; you'll see why later in lab, but just go with it for now.

Like `evenP` in lab last week, this function has *three* useful cases—zero one and $2 + n$ (we won't count the negative one case). The new thing about this function is that it makes recursive calls not just on $n - 2$ but also on $n - 1$.

Because of these two recursive calls, the recurrence for the work looks like this (using our convention that we can write any constant number of steps as 1):

$$\begin{aligned}
&W_{\texttt{fib}}(0) = 1\\
&W_{\texttt{fib}}(1) = 1\\
&W_{\texttt{fib}}(n) = 1 + W_{\texttt{fib}}(n-1) + W_{\texttt{fib}}(n-2) \text{ for non-zero } n
\end{aligned}$$

This is not so helpful, since it says that the time to compute the n^{th} Fibonacci n is the n^{th} Fibonnaci number!

However, if we can get an *upper bound* for this recurrence as follows:

$$\begin{aligned}
&W_{\texttt{fib}}(0) = 1\\
&W_{\texttt{fib}}(1) = 1\\
&W_{\texttt{fib}}(n) \leq 1 + 2W_{\texttt{fib}}(n-1) \text{for non-zero } n
\end{aligned}$$

Because $W_{\texttt{fib}}(n)$ is *monotonically increasing* (it's never smaller on bigger inputs), we can pretend that it's two recursive calls on $n - 1$.

This recurrence is $O(2^n)$, or exponential time, because at each level of the recursion you double the number of steps done in the previous level.

2.2 Fast Fibonacci

In this problem, you will show that you can compute Fibonacci more efficiently. The key insight is that one of the recursive calls can be reused each time:

```
To compute     We need
fib n          fib (n-1) and fib (n-2)
fib (n-1)      fib (n-2) and fib (n-3)
fib (n-2)      fib (n-3) and fib (n-4)
```

So we really don't need two recursive calls, if we reuse the same computaton of `fib n` the two times we use it. To implement this, you must *generalize* the problem so that we compute both `fib n` and `fib (n-1)`.

2.2.1 Programming

Task 2.1 Implement a function

```
fastfib : int -> int * int
```

such that for all nats n, `fastfib` $n \cong (\texttt{fib}(n-1), \texttt{fib}\ n)$

Have the course staff check your code for `fastfib` before proceeding.

2.2.2 Analysis

Task 2.2 Write a recurrence for the work of `fastfib`,
$W_{\texttt{fastfib}}$.

Task 2.3 Compute the closed form for your recurrence.

Task 2.4 Give a tight big-O bound for this closed form.

2.2.3 Proof

Task 2.5 Prove that your code meets the specification, which is to say:

Theorem 1. *For all natural numbers `n`, `fastfib n` $\cong$ `(fib (n - 1) , fib n)`*

Use the template on the following page. Have the course staff check your work once you finish.

Theorem 2. *For all natural numbers* `n`*,* `fastfib n` $\cong$ `(fib (n - 1) , fib n)`

The proof is by induction on m.

- **Case for** 0

  To show:

  Proof:

- **Case for** $1 + k$

  Inductive hypothesis:

  To show:

  Proof:

Have the course staff check your code, analysis, and proof for `fastfib` before proceeding.

3 Merge

Task 3.1 Write a function

```
merge : int list * int list -> int list
```

that merges two sorted lists into one sorted list. You should assume that your input lists are sorted in increasing order, and the list you return should also be in increasing order.

Task 3.2 Write a recurrence relation for the work of `merge`, in terms of the lengths of `l1` and `l2`. What is the O of this recurrence?

Task 3.3 Prove the following correctness theorem about `merge`:

Theorem 3. *For all lists of integers `l1` and `l2`, if `l1` and `l2` are both sorted in increasing order, then `merge (l1, l2)` is sorted in increasing order.*

Hint: In your proof, you will need a lemma about how the contents of `merge(l1,l2)` relates to `l1` and `l2`. You should state this lemma, and convince yourself it is true, but you don't need to prove it formally.