Part A: Orders of growth (40 points)

This section will feature problems relating to estimating time and space complexity in OCaml functions. As usual, write essay-type answers as OCaml comments.

0. In-class exercise [**no collaboration**]

[5 points]

What are the time and space complexities of the following function?

```
let rec is_power_of_two n =
  if n = 1 then
    true
  else if n mod 2 <> 0 then
    false
  else
    is_power_of_two (n / 2)
```

Explain your reasoning. (A sentence or two should be enough.)

1. Fibonacci again

[5 points]

Consider the tree-recursive fibonacci function discussed in class:

```
let rec fib n =
  if n < 2
     then n
     else fib (n - 1) + fib (n - 2)
```

You know that the time complexity for this function is $O(2^n)$.

> **Note**
>
> More specifically, it is $\Theta(g^n)$, where g is the golden ratio ($1.618...$). See SICP section 1.2.2 and exercise 1.13 for more on this.

If we assume that OCaml is using applicative-order evaluation (the normal OCaml evaluation rule), then what is its *space* complexity? Explain why this is different from the time complexity.

Hint: consider what the largest number of pending operations would have to be when evaluating `fib 7`. You may assume that once an expression is fully evaluated, all the memory used in evaluating that expression is returned to the system.

> **Note**
>
> You are not required to state what the effect of normal-order evaluation is on the space requirements of the `fib` function, though it is interesting! 😄

2. (SICP exercise 1.15)

[5 points]

The sine of an angle (specified in radians) can be computed by making use of the approximation:

$$sin\ x = x$$

(if x is "sufficiently small"), and the trigonometric identity:

$$sin\ x = 3\ sin(x/3) - 4\ sin^3(x/3)$$

to reduce the size of the argument of sin if x is *not* sufficiently small. For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is less than 0.1 radians.

These ideas are incorporated in the following functions (using floating-point arithmetic throughout):

```
let cube x = x *. x *. x
let p x = 3.0 *. x -. 4.0 *. cube x
let rec sine angle =
```

```
  if abs_float angle < 0.1
     then angle
     else p (sine (angle /. 3.0))
```

1. How many times is the function `p` applied when `sine 12.15` is evaluated?
2. What is the order of growth in space and number of steps used by the process generated by the `sine` function when `sine a` is evaluated (as a function of `a`)?

By "growth in number of steps", we mean the asymptotic time complexity of the `sine` function as a function of the size of the input. Explain the reason for the space/time complexities; don't just state an answer.

3. (SICP exercise 1.16)

[5 points]

SICP describes a non-iterative function called `fast_expt` that does exponentiation using successive squaring (when possible). Translated into OCaml, that function looks like this:

```
let rec fast_expt b n =
  let is_even m = m mod 2 = 0 in
  let square m = m * m in
    if n = 0 then 1
    else if is_even n then square (fast_expt b (n / 2))
    else b * fast_expt b (n - 1)
```

Note that `mod` is a predefined infix operator in OCaml (not a function!) which computes remainders; you use it like this: `5 mod 2` (which will return `1`).

a.

This function uses nested `if` / `then` / `else` forms, which are a bit ugly and error-prone (since OCaml doesn't actually have an `else if` syntax). Rewrite the function using pattern matching on the `n` argument (a `match` expression); use `when` clauses in pattern matches when you need to test for non-structural conditions (and *only* then). Your function should not have any `if` expressions.

Here is a skeleton version of the function you should write:

```
let rec fast_expt b n =
  let is_even m = m mod 2 = 0 in
  let square m = m * m in
```

```
    match n with
      (* fill in the rest here *)
```

N.B. the "wildcard" pattern `_` may be useful to you. Any compiler warnings will be considered to be errors.

b.

Write a function called `ifast_expt` that evolves an *iterative* exponentiation process that uses successive squaring and uses a logarithmic number of steps.

> **Hint**
>
> Using the observation that $b^n = (b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.

Use integer arithmetic for this problem. You may assume that all the (integer) arguments to your function are non-negative.

You will need some helper functions in the implementation of your `ifast_expt` function. You should make these internal to your `ifast_expt` function, as was done with the `fast_expt` function above. One of these will need to be recursive; call it `iter`. Only use `let rec` with that function; use `let` for all other internal definitions and for the `ifast_expr` function as a whole.

Use pattern matching instead of nested `if` / `then` / `else` forms as you did for the `fast_expt` function.

4. (SICP exercise 1.17)

[5 points]

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. The simplest such function is this:

```
let rec expt a b =
  if b = 0
```

```
    then 1
    else a * expt a (b - 1)
```

In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication function (in which it is assumed that our language can only add, not multiply), is analogous to the `expt` function:

```
let rec mult a b =
  if b = 0
     then 0
     else a + mult a (b - 1)
```

This algorithm takes a number of steps that is linear in `b`. Now suppose we include, together with addition, the operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication function analogous to `fast_expt` that uses a logarithmic number of steps. Call this function `fast_mult`.

> **Note**
>
> Don't do any multiplications or divisions in `fast_mult` except indirectly by using `double` or `halve`. (`double` and `halve` themselves can do multiplication or division, of course.) You can also use addition, subtraction, and the `mod` operator to test for evenness.

Use integer arithmetic for this problem. The multiplication function you write should generate a recursive process.

For this problem, write all helper functions (including `double` and `halve`) inside the `fast_mult` function, and again use pattern matching on the `n` argument instead of nested `if`/`then`/`else` expressions.

5. (SICP exercise 1.18)

[5 points]

Using the results of the previous exercises, devise a function called `ifast_mult` that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

This multiplication function should generate an iterative process. If you are multiplying integer b by integer n, you will need another state variable a such that the invariant is $a + bn$, and n will

decrease to zero, at which point a will be the answer.

Again, use pattern matching instead of nested `if` / `then` / `else` expressions. And again, only use `let rec` where it's absolutely necessary.

6. A mysterious function

[5 points]

Consider the following (higher-order) function:

```
let rec foo f n =
  if n <= 1
     then f 0
     else foo f (n / 2) + foo f (n / 2)
```

Note that function calls have the highest precedence in OCaml, so the last expression is the same as `(foo f (n / 2)) + (foo f (n / 2))`.

If we assume that the function `f` can compute its result in constant time and constant space, what are the (worst-case) time and space complexities of the function `foo`? Justify your answer. (It doesn't have to be a full mathematical proof, but it should be a convincing argument.) Assume that the integer input `n` is always non-negative, and assume the usual applicative-order evaluation rule.

> **Note**
>
> This problem shows that tree recursion doesn't *necessarily* give rise to an inefficient function.

7. Fibonacci yet again

[5 points]

Consider this function to compute fibonacci numbers:

```
let fib n =
  let rec last_two n =
    if n < 1
      then (0, 1)
      else
```

```
        let (p0, p1) = last_two (n - 1) in
          (p1, p0 + p1)
  in
    fst (last_two n)
```

A couple of OCaml notes:

- It's legal to assign more than one value at a time in a `let` expression as shown above. (Effectively, you are doing a pattern match that cannot fail.)
- `fst` is a built-in function which extracts the first value of a two-tuple.

Please answer the following two questions in OCaml comments:

1. What kind of process does this function represent (linear recursive, linear iterative, tree recursive *etc.*) and why?
2. What is the space and time complexity of this function with respect to its argument `n`?