**A:** Yes.

- **Q:** May I use some additional library that would require changing the Makefile or tags file?
  **A:** No.
- **Q:** May I use loops?
  **A:** No.
- **Q:** May I use imperative features?
  **A:** No. See the Imperative Features Prohibitions page linked from the Canvas front page.
- **Q:** Why am I getting "Unbound module" errors in VS Code?
  **A:** Make sure you have run `make build` and you have the `.merlin` file from the release code in your source directory.

## Getting Started



Download the release code from CMS. Watch the A0 GIST in Canvas for a demo of how to get the code into Ugclinux by dragging and dropping it into the Explorer pane in VS Code. Or, if you're using a local installation, just put the code somewhere on your own local drive.



Run `make check` from the folder containing the release code. You should get output that includes the following (other lines might be interspersed):



```
============================================================
Your OCaml environment looks good to me.  Congratulations!
============================================================
Your function names and types look good to me.
Congratulations!
============================================================
```

If instead you get warnings about your OCaml environment, seek help immediately from a consultant. Assuming your environment is good, you should not get warnings about about your function names and types. After completing each function, you should re-run `make check` to ensure that you haven't changed any required names or types.

There are a few other targets provided in the release code:

- Running `make` will open the toplevel and automatically load your code. That should be useful for any interactive testing you want to do.
- Running `make test` will build and non-interactively run your OUnit test suite.
- Running `make clean` will delete files generated from your source code.

- Running `make finalcheck` will do the same as `make check` as well as two other things, which are described under the submission instructions at the end of this handout.

## Academic Integrity

Please review and complete the Academic Integrity statement provided in a comment at the top of the release code file `warmup.ml`.

## Problem 1: I No Longer Know What Day It Is

Define a function `day_of_week : int -> int -> int -> int` that takes integers `year`, `month`, and `day`, in that order, as input; and outputs the day of the week on which that date falls.

- Use the ISO 8601 weekday number standard to represent the day of the week: Monday is 1, Tuesday is 2 ... Sunday is 7.
- Also per ISO 8601, assume that the proleptic Gregorian calendar is in use. That is, don't bother thinking about whether the Gregorian calendar was actually in use on the input date—which anyway would depend in part on the location in the world. Just assume that the Gregorian calendar always has and always will be in use, everywhere.
- The behavior of the function is *unspecified* if the input date is not a valid date in the proleptic Gregorian calendar. Valid dates have a year that is strictly greater than 0, a month in the range 1..12, representing Jan..Dec; and a day in the range 1..{28,29,30,31}, depending on what the number of days is in the year and month. Since the behavior is unspecified, your implementation is free to do anything you like on an invalid input—for example, raise an exception with a helpful error message, return a bogus output, or even set the computer on fire. 🔥🔥🔥 Though, truth be told, the graders would prefer you didn't set their laptops on fire. Note that "asserting the precondition" is helpful **but not required.** See section 2.4.2 of the textbook for details.
- Use Zeller's congruence to solve this problem. Be careful about coding the computions correctly, especially the modulo operations.

**Specification:** [UPDATED 09/09/20] Before writing any code, read the provided specification comment for the function. Please do not change it. Note the precondition: it requires validity. Also, Python programmers take note: there is no precondition about the types of values; section 2.3.8 of the textbook explains why should a precondition would be unidiomatic in statically-typed languages (e.g., Java and OCaml).

**Testing:** File `warmup_test.ml` has two OUnit test cases to get you started. Make sure your solution passes those, without any modification to them. Add eight more test cases, for a total of ten.

- Section 3.1.3 of the textbook covers the mechanics of unit testing with OUnit.
- If you want to test for an exception being raised, section 3.2.5 covers that, but it's getting a little ahead of where we are so far in the course. There's no real need to test for exceptions on this assignment if you require validity of the date as a precondition.
- Later in the semester we'll learn about more principled approaches to testing; for now, this "big round number of 10" is just to get practice with writing OUnit test cases.
- If you're having trouble inventing test cases for dates far away from today, consider cross-validating against another language and library you already know. For example, both Python and Java have libraries for computing this same function. With a little research, you should be able to find them.



One of the big differences between CS 2110 and 3110 is that, starting now, development of test suites becomes your responsibility. It was a luxury that they were provided for you in 2110. So please take this part of the assignment seriously.



**Code quality:** After you finish testing, take extra time to make your code as readable and elegant as you can. See the code quality rubric at the end of this handout for suggestions. For this problem, concentrate especially on (i) using whitespace and indentation well, (ii) using `let` and `if` expressions to organize the computation, and (iii) using parentheses only where necessary. If you're not sure whether parentheses are necessary, try removing them and re-running your test suite.



## Problem 2: How Do I Live Without Loops?

*Sigma notation*, credited to Leonhard Euler, compactly represents long arithmetic expressions. For example,

$$\sum_{n=1}^{10} n^2$$

represents

$$1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2.$$

To an imperative programmer, sigma notation immediately suggests a loop. For example, a Java

programmer could implement

$$\sum_{n=a}^{b} f(n)$$

as

```
int sum= 0;
int n= a;
while (n <= b) {
  sum+= f(n);
  n++;
}
```

But to a functional programmer, sigma notation suggests a recursive definition:

$$\sum_{n=a}^{b} f(n) = f(a) + \sum_{n=a+1}^{b} f(n) \quad \text{if } a \leq b$$
$$= 0 \quad \text{otherwise}$$



**Problem 2.1:** Implement a function `sigma : (int -> int) -> int -> int -> int` that reduces an integer summation to its final numeric value, such that `sigma f a b` is $\sum_{n=a}^{b} f(n)$.



■

*Pi notation* is like sigma notation, but it represents a product rather than a sum. For example, a Java programmer could implement

$$\prod_{n=a}^{b} f(n)$$

as

```
int prod= 1;
int n= a;
while (n <= b) {
  prod*= f(n);
  n++;
}
```

Note that the product over an empty *range* is 1, as suggested by the Java implementation above.

The range is the set of values $a..b$. In mathematics, any repeated operation like $\Sigma$ or $\Pi$ based on a binary operator is defined to be the *identity element* of that operator when taken over an empty range. The identity element is the value that causes the operator to leave its other operand unchanged. So the identity element of $+$ is 0, and the identity element of $\times$ is 1.

**Problem 2.2:** Implement a function `pi : (int -> int) -> int -> int -> int` that reduces an integer product to its final numeric value, such that `pi f a b` is $\prod_{n=a}^{b} f(n)$.

■

Your implementations of `sigma` and `pi` should be nearly identical. Let's factor out their commonality into a single function `reduce`:

```
let rec reduce inc test op base f x =
  if test x then base
  else op (f x) (reduce inc test op base f (inc x))
```

You may not change the provided implementation of `reduce`.



**Problem 2.3:** Write a specification comment for `reduce`. Imagine that you were trying to explain the function to a friend. What would you say? Write that down in the comment. Then, re-implement `sigma` and `pi` as `sigma_r` and `pi_r`, with the same types, using just a single application of `reduce` in each. You should not use the `rec` keyword in the definitions of `sigma_r` and `pi_r`; all the recursion should be in `reduce`. Your two new function bodies should contain just a single line of code each. *Hint: make use of the OCaml syntax for treating infix operators as functions, e.g.,* `( + )`.





■

The implementation of `reduce` that we gave above is not tail recursive. That means your implementation of `sigma_r` is going to struggle with sums over large ranges:

```
# sigma_r (fun x -> x) 0 1_000_000;;
Stack overflow during evaluation (looping recursion?).
```

**Problem 2.4:** Re-implement `reduce` as a tail-recursive function `reduce_tr`, with the same type. Re-implement `sigma_r` and `pi_r` as `sigma_tr` and `pi_tr`; the only change should be replacing identifier `reduce` with `reduce_tr`.

The tail-recursive version might end up applying `op` in a different order. For example, computing

$$\sum_{n=1}^{2} n$$

could cause `sigma_r` to evaluate `1 + (2 + 0)`, whereas `sigma_tr` might instead evaluate `2 + (1 + 0)`. This is fine. You may assume that `op` is commutative and associative, and that `base` is an identity element w.r.t. `op`.

You will need a helper function to implement `reduce_tr` with the correct type. Although we generally don't care in this course whether you use `let` expressions to nest the definitions of helper functions, in this case we do want you to nest: it will be good practice, and more importantly it will enable you to write an elegant helper function that does not take any unnecessary arguments. *Hint: it needs only two.*

■

Now you will be able to compute sums over larger ranges:

```
# sigma_tr (fun x -> x) 0 1_000_000;;
- : int = 500000500000
```



Of course, sums and products of large numbers will still be problematic because of the limited range of integers, but that's not your fault.

**Specification:** The only specification comment we will grade in this problem is the one you wrote for `reduce`. Your comment needs to explain each input, as well as the output.

**Testing:** Include at least two test cases each for `sigma` and `pi`, testing the base case and the recursive case for each. Include at least one test case each for `sigma_r`, `pi_r`, `sigma_tr`, and `pi_tr`. For the tail-recursive functions, those test cases need to test on large ranges that would overflow the stack on the non-tail-recursive implementations—e.g., a million elements. In total, that's at least eight test cases.

**Code quality:** Again, after you finish testing, take extra time to make your code as readable and elegant as you can. Focus on the same aspects that are listed above in the first problem.

## Problem 3: So Simple A Child Could Do It

Martha and Mike (M&M) are picking teams of Managing Vice Provosts (MVPs) for the annual Administrivia Tournament. MVPs are rated by their Bureaucratic Aptitude for Endowment Raising (BEAR). Because M&M are so fair-minded, they want to form teams whose total BEAR scores are as equal as possible. How to do it?

Martha, being a computer scientist, recognizes this task as the *partition problem*. (You might too,

Martha, being a computer scientist, recognizes this task as the partition problem. (You might too, if you took CS 2110 in Spring 2020. It was covered as an example of a recursive backtracking algorithm.) Solving the problem perfectly is computationally expensive, seemingly requiring an exponential-time algorithm. But there is a simple polynomial-time algorithm that is easy to describe and implement, even though it does not produce perfect solutions. Children around the world know this algorithm. For that, among other even more compelling reasons, the partition problem has been called "the easiest hard problem."

We'll state the partition problem more carefully as follows. You are given a bag $b$ of strictly positive integers.

> *A bag (aka multiset) is like a set, in that order does not matter. But unlike a set, bags may contain multiple copies of an element. So, {1, 1, 2, 3} is a bag but not a set, and it is considered to be the same bag as {3, 2, 1, 1}.*

You want to partition $b$ into $n$ non-empty bags $b_1, \ldots, b_n$ whose sums are close to one another. In particular, let $s_i$ be the sum of the integers in $b_i$, let $M$ be largest element of $\{s_i \mid i \in 1..n\}$, and let $m$ be the smallest. You want to make $M - m$, which is called the *discrepancy*, as small as possible. For example, if $n$ is 2, you could partition {8, 7, 6, 4} into



- two bags {8, 7} {6, 4} with a discrepancy of 5, or



- two bags {8, 4} {7, 6} with a discrepancy of 1 (which is the best achievable for the given bag),
- and so forth.



Or if $n$ is 3, you could partition it into three bags {8} {7} {6, 4} with a discrepancy of 3, and so forth.

The polynomial-time algorithm is simple. Start with $n$ empty bags. Sort the input bag $b$ in descending order. Remove the largest element and place it in the bag with the smallest sum (or any such bag, if there is a tie for the smallest sum). Keep doing that until the input bag is empty. If $n$ is 2 and the input bag is {4, 8, 7, 6}, the algorithm would:

- initialize two empty bags
- sort {4, 8, 7, 6} into the sequence 8, 7, 6, 4.
- put 8 in the first bag
- put 7 in the second bag
- put 6 in the second bag
- put 4 in the first bag

That yields output bags {4, 8} and {6, 7}, with a discrepancy of 1. A discrepancy of 0 isn't

achievable, because the sum of the input bag is odd.

This algorithm is an example of a *greedy algorithm*, which is a class of algorithms that make locally-optimal choices at the risk of not finding a globally-optimal solution. Dijkstra's algorithm, which you might recall from CS 2110, is a greedy algorithm that always finds a globally-optimal solution. But the algorithm we just gave does not necessarily find globally-optimal solutions. For example, when asked to partition input {8, 7, 6, 5, 4} into two bags, the algorithm outputs {8, 5, 4} {7, 6} or {8, 5} {7, 6, 4} with a discrepancy of 4. But there is a perfect solution with a discrepancy of 0, which is {8, 7} {6, 5, 4}.

Imperfect though it may be, the greedy algorithm for the partition problem is nonetheless easier to understand than more optimal algorithms. And on a slight variant of the problem, the greedy algorithm is provably not far away from optimal [Graham 1969, Theorem 2].

**Problem 3.1:** Implement a function `partition : int list -> int list * int list` that partitions its input bag into two output bags using the greedy algorithm above. Despite the simplicity of the algorithm, implementing it is not child's play.



- We use lists here to represent bags, and a pair to represent the two output bags. Of course, since order matters in a list and in a pair, that complicates testing. In grading of your solution, we will ignore the order of elements in lists and the order of components in the pair. That makes the course staff's job harder; but yours, easier.
- If there's a tie in which bag you should put an element into, we'll accept either as a correct output.
- As a precondition, you may require that the input bag has at least 2 elements.
- Use `List.sort` to handle the sorting of the input bag. Code your own comparison function to sort in descending order.
- A few points in the Excellent Scope will be allocated to testing that your solution can handle inputs of up to 1 million elements in fewer than 10 seconds without a stack overflow. We've provided an operator `(--)` for generating large lists. Concrete execution time bounds are admittedly somewhat meaningless, because some people have slower or faster machines. But 10 seconds is more than generous here; 2 seconds is probably more accurate. If in doubt about your own machine's speed, try running your code on ugclinux. If your code isn't fast enough, think carefully about whether it is needlessly re-computing any values (e.g.,the sums of the output bags) at each recursive call.
- If it's obvious to you how to solve the next subproblem, `multi_partition`, you could implement `partition` completely in terms of it by reversing the order of their definitions in `warmup.ml`. But in general we recommend solving `partition` first to get some insight

and practice, then moving on to `multi_partition` afterwards. It's fine for your solutions to the two problems be separate pieces of code: you don't need to re-implement `partition` in terms of `multi_partition`.

■

**Problem 3.2:** Implement a function `multi_partition : int -> int list -> int list list`, such that `multi_partition n b` partitions input bag `b` into `n` output bags. The length of the output list should therefore be `n`. Each element of that output list is itself a bag.

- Really we should think of the output list as itself being a bag, because order does not matter in it. So in grading your solution we will again ignore order there.
- As preconditions, you may require that `n` is at least 1, and that the input bag contains at least `n` elements.
- For performance-testing purposes, you may assume that `n` will be 1000 or less. We could use a heap data structure (or something similar) to eliminate that assumption, but we haven't covered enough OCaml yet.



■



**Specification:** We will not grade any specification comments in this problem.



**Testing:** Include at least five test cases each for `partition` and `multi_partition`. Make sure to test `multi_partition` on some `n` that are not 2. The "Easiest Hard Problem" paper mentioned above has some examples you could use as test cases if you like.

We won't grade whether *you* test on some really large input bags, though our grading suite will include a few such tests for the Excellent Scope. But if you're interested… Testing on really large input bags could be tedious, because you would seemingly need to hard-code the correct outputs. Instead, try testing whether the output is a partition of the input. That can be done with much less code. This is an example of what's called *property-based testing*: checking whether the output has some property, rather than what the exact output is.

**Code quality:** As always, take extra time to make your code as readable and elegant as you can. Focus on the same aspects that are listed above in the first problem. Now you will have `match` expressions to beautify as well. Although the compiler will sometimes permit the omission of parentheses around pairs, you should leave them in: stylistically, they are not considered unnecessary except in a couple very specific circumstances. See the OCaml Community guidelines under the heading "How to write pairs" for details.

# Rubric

- 25 points: submitted and compiles
- 25 points: satisfactory scope
- 25 points: good scope
- 5 points: excellent scope
- 10 points: testing
- 10 points: code quality (including specifications)

**Scope Rubric.** Your solution earns points by passing the grading suite's OUnit test cases on the following functions:

- Satisfactory: `day_of_week`, `sigma`, `pi`.
- Good: `sigma_r`, `pi_r`, `reduce_tr`, `sigma_tr`, `pi_tr`, `partition`
- Excellent: `multi_partition`, performance of `partition` and `multi_partition` on large inputs

Each test case is worth some number of points, so it's definitely possible to get partial credit on individual functions that way. Sorry, but we can't provide a more detailed breakdown in advance.



**Testing Rubric.** Later in the semester we will focus much more on developing test suites. For this first assignment, though, you just need to provide the test cases required above. We will grade your test cases independently of whether you've finished a level of scope, which is to say you can get (or lose) points for testing even if you don't finish the functions they test. For example, if you don't write `multi_partition` or test cases for it, you will lose points both from the excellent scope autograder and from the manual grading of your test cases. But you could provide those test cases and get points for them even if you don't write the function itself. Part of the reason we grade this way is to incentivize you to write test cases before writing (or finishing) a function.

**Code Quality Rubric.** There are many aspects of quality that make code easier to read. We will focus on just a few for this first assignment. The graders will assess the following aspects of your source code:

- The programmer's name and NetID are supplied at top of file.
- The specification comments described above are provided.
- White space (the space character and empty lines) is used effectively.
- Indentation of `let`, `if`, and `match` follows the OCaml Community Guidelines.
- Identifier names are clear and consistent. They use `snake_case`, not `camelCase`. See "How to choose identifiers" and following paragraphs in the Community Guidelines.
- Parentheses are used only where necessary. See "when to use parentheses within an expression" in the Community Guidelines for some hints.
- No line is wider than 80 columns. Not only does the OCaml community agree on that, so does Google.