built on 2023/11/06 at 12:59:09 ***due:*** *mon nov 20 @ 11:59pm*

This assignment aims to give you practice implementing nontrivial logic in Rust, as well as introducing you to shared-memory parallel programming and work-span cost analysis. We're providing some starter files; please *download the starter code from the course website.*

Don't use an AI tool to write code for you; it defeats the purpose of learning to think for yourself in that language.

You will zip all your work in one zip file and upload it to Canvas before the due date. **Make sure to `cargo clean` before creating your zip. Rust builds are bloated and make your TAs unhappy.**

- You can define as many helper functions as necessary.
- You are going to be graded on style as well as on correctness.
- Test your code! Add more tests to what we supply.

Cargo Tips

Running Tests. Use `cargo test` to run all the unit tests. Check out provided tests.

Cleaning Up Garbage. Use `cargo clean` to clean up files resulting from your various builds.

Code Formatting. Use `cargo fmt` to auto-format all your source files.

Code Linting. Use `cargo clippy` for suggestions on how to write more idiomatic Rust code.

Task 1: Perfect Numbers (8 points)

For this task, save your code in `perfect/mod.rs`

In ancient Greek, the aliquot sum of n, denoted $s(n)$, is the sum of all divisors of n, excluding n itself. For example, $s(6) = 1+2+3 = 6$ because all the numbers that divide 6 aside from 6 itself are 1, 2, and 3. As some more examples, $s(1) = 0$ and $s(8) = 1+2+4 = 7$. This notion has led to many insights.

Greek mathematician and music theorist Nicomachus created a classification of numbers based on their aliquot sums. Specifically, a number n is said to be

- *perfect* if $s(n) = n$,
- *deficient* if $s(n) < n$, and
- *excessive* if $s(n) > n$.

Based on this, the starter code has defined an `enum` called `Classification`. Your tasks are as follows:

(i) Implement `pub fn classify_perfect(n: u64) -> Classification` that classifies a number `n` into one of the three kinds above. For example:

```
classify_perfect(1) == Deficient
classify_perfect(6) == Perfect
classify_perfect(12) == Excessive
classify_perfect(28) == Perfect
```

(*Hint:* The divisor sum is a one-liner in Rust.)

(ii) Implement

```
pub fn select_perfect(range: Range<u64>, kind: Classification) -> Vec<u64>
```

that returns a vector containing all number in the given `range` that has the given `kind`. For example:

```
select_perfect(1..10_000, Perfect) == vec![6, 28, 496, 8128]
select_perfect(1..50, Excessive) == vec![12, 18, 20, 24, 30, 36, 40, 42, 48]
select_perfect(1..11, Deficient) == vec![1, 2, 3, 4, 5, 7, 8, 9, 10]
```

To test your code, add plenty tests to the simple test battery we supply. You will hand in your additional tests, which will be looked at.

Task 2: Pangrindrome (8 points)

For this task, save your code in `pangrindrome/mod.rs`

This task involves playing with pangrams and palindromes. This will give you more practice working with strings, iteration, and iterator idioms in Rust. Your task is to implement the following functions:

(i) Write a function

```
pub fn is_palindrome(s: &str) -> bool
```

that takes a string slice as input and returns a `bool` indicating whether `s` is a palindrome. Remember that a palindrome is a string where reading it left to right is the same as reading it right to left. (*Hint:* `s.chars()` gives an iterator over the characters of `s`.)

(ii) Write a function

```
pub fn is_pangram(s: &str) -> bool
```

that takes a string slice as input and returns a `bool` indicating whether `s` is a pangram. For the purpose of this task, a *pangram* is a string where each of `a` through `z` (uppercase or lowercase is fine) is present in the string.
(*Hint:* You may wish to keep a map or a set but it's not at all necessary. Also, `('a'..='z')` is a valid range.)

To test your code, add plenty tests to the simple test battery we supply. You will hand in your additional tests, which will be looked at.

Task 3: Roman Numerals (8 points)

For this task, save your code in `roman/mod.rs`

This problem involves writing code to convert to and from Roman numerals. Refer to Assignment 1 for how the numeral system is set up. You will implement the following two functions:

(i) Write `pub fn to_roman(n: u16) -> String` that takes in a number `n` between 1 and 3,999 and returns a string representing this number in Roman numeral. For example:

```
to_roman(7) == "VII"
to_roman(14) == "XIV"
to_roman(1954) == "MCMLIV"
```

(ii) Write `pub fn parse_roman(roman_number: &str) -> u16` that takes in a string slice representing a Roman number and returns a number that is equal to the Roman number input. For example:

```
parse_roman("XIV") == 14
parse_roman("XLIX") == 49
parse_roman("MCMLIV") == 1954
```

To test your code, add plenty tests to the simple test battery we supply. You will hand in your additional tests, which will be looked at.

Task 4: Character Frequencies (4 points)

For this task, save your code in `charfreq/mod.rs`

Implement a function

```
pub fn par_char_freq(chars: &[u8]) -> HashMap<u8, u32>
```

that takes in an array slice of unsigned 8-bit numbers (i.e., bytes) and returns a hash map where each unsigned 8-bit number key is mapped to its frequency (i.e., the number of times this particular key shows up in the input `chars`).

This implementation is expected to be parallel. Your code will take advantage of Rayon's parallel iterators. If you feel like it, you can also use the concurrent hash map included in the `chashmap` crate—but you aren't required to.

Examples:

```
par_char_freq("banana".as_bytes()) // => {110: 2, 98: 1, 97: 3}
par_char_freq("mississippi".as_bytes()) // => {115: 4, 109: 1, 112: 2, 105: 4}
```

Hints: If you're using the concurrent hash map, the `.upsert` function can be especially useful. The idea of "upsert" is to insert an entry if no entry of the given key is present and to update the value (atomically) if the key is existing. As an example, a histogram `histo` can be updated as

```
// look at c: if c isn't present insert c with value 1;
// otherwise, add 1 to the value.
histo.upsert(c, || 1, |v| *v += 1)
```

Task 5: Work/Span and Parallel Linear Search (8 points)

For this task, save your code in `linsearch/mod.rs`

This problem contains a written portion, as well as a coding portion. Your written answer to this problem will also go inside the same file as a comment block.

In a typical set up, linear search takes as input a sequence of items `xs` and a target key `k`, and returns the *smallest* index into `xs` where `k` is found. This is often implemented using a simple loop over the array, looking for a matching item, for example:

```
fn lin_search<T: Eq + Send>(xs: &[T], k: &T) -> Option<usize> {
    for (index, elt) in xs.iter().enumerate() {
        if *elt == *k {
            return Some(index);
        }
    }
    None
}
```

This problem involves implementing this logic in parallel. Linear search can be implemented using a divide-and-conquer strategy as follows:

> Split the array at midpoint. Recursively find the target key on the left half and on the right half. To combine the answers, prefer an index found on the left.

This means that the two recursive calls can be invoked in parallel.

Your tasks are as follows:

(i) You will implement a function

```
pub fn par_lin_search<T: Eq + Send>(xs: &[T], k: &T) -> Option<usize>
```

that uses fork-join parallelism in Rayon to implement the above parallel linear search idea. Your code will *not* split the array for real but will rather take advantage of array slices.

(ii) As a comment block above your implementation, analyze the work and span of your implementation. If you write a recurrence, point out what it solves to. Explain your reasoning.

To test your code, add plenty tests to the simple test battery we supply. You will hand in your additional tests, which will be looked at.

Task 6: Number of Primes (8 points)

For this task, save your code in `numprimes/mod.rs`

This problem involves making `is_prime` parallel and using it to develop a function that counts the number of primes up to n in parallel.

(i) Write a function

```
pub fn par_is_prime(n: u64) -> bool
```

that determines whether or not `n` is prime. Remember that a number n is prime if $n \geq 2$ and the only divisors of n are 1 and n. Your code will implement the $\sqrt{n}$ optimization, i.e., it'll only test numbers between 2 and $\lfloor \sqrt{n} \rfloor$ (inclusive).

You will use Rayon's parallel iterator to try the potential divisors in parallel.

(ii) Write a function

```
pub fn par_count_primes(n: u32) -> usize
```

that returns the number of primes between 1 and `n` (inclusive). This function will go through the numbers in this range in parallel and will make use of your implementation of `par_is_prime` above.

You will use Rayon's parallel iterator to call the function above and count the number of primes.

To test your code, add plenty tests to the simple test battery we supply. You will hand in your additional tests, which will be looked at.

Task 7: Parallel Base64 Encoder/Decoder (8 points)

For this task, save your code in `base64/mod.rs`

Base64 is a binary-to-text encoding scheme designed to represent binary data[1] (any sequence of 8-bit characters) as a human-readable, plaintext string. The encoding is widely used on the web, for example, as a means to embed images inside `html` or `css` files. Learn the details of Base64 encoding at https://en.wikipedia.org/wiki/Base64. Feel free to also browse other sites for alternative descriptions. For the super detailed, authoritative source, look no further than Section 4 of https://datatracker.ietf.org/doc/html/rfc4648.html#page-5.

This problem is concerned with writing two parallel functions—an encoder that transforms an `&[u8]` into a base64-encoded `String`, and a decoder that transforms a base64-encoded `&str` into the original binary data.

(i) Write a function

```
pub fn par_encode_base64(bytes: &[u8]) -> String
```

[1] The full range of binary data contains both human readable and "weirdo" characters. The weirdos aren't always friendly to systems/programs that only reliably accept plaintext data.

that takes as input a sequence of 8-bit symbols (i.e., binary data) and returns a `String` that is the base64-encoded version of the input.

Use of Rayon's parallel iterator is recommended.

(ii) Write a function

```
pub fn par_decode_base64(code: &str) -> Option<Vec<u8>>
```

that takes as input a string-slice `&str` representing a base64-encoded piece of data and returns the decoded original data as a vector of 8-bit symbols. The return type is an `Option`, so the function can return `None` when the input coded string is invalid and can't be decoded.

Ground Rules and Hints:

- Your code must work directly with the characters, building everything from the ground up. Do *not* use/import a library for base64 encoding.
- Remember that 6 bits × 4 = 24 bits = 8 bits × 3. Therefore, a block of 3 original symbols corresponds to a block of 4 base64 symbols. The blocks can be worked on simultaneously in parallel. However, be careful with padding.
- Your functions must be parallel, i.e., the span must be asymptotically less than n.