Functional & Parallel Programming — Test II (T. II/22–23)

Directions:

- This examination consists of *three* problems. You have 30 hours to complete it. The examination is due back on Canvas at 11.59PM on Saturday April 1, 2023. In the same zip file as this handout, you will also find 3 starter files, one per problem.
- This is an online exam with an open-everything policy; you can look at your notes, previous assignments, search the Internet, etc. However, you must cite your sources. Also, you are not allowed to talk to/collaborate with another person (or AI). **Asking a question on or using answers from online study-help sites (e.g., Chegg) is considered cheating**. But using stackoverflow and similar professional sites in *read-only mode* is okay.
- To upload your work, zip up the files and upload it to Canvas. There are only 3 files to hand in (one per problem). Don't include any other files.
- The goal is for the code to be concise, efficient, and readable. Every subtask in Q1 and Q2 should not need more than 10–15 lines of code. Unless stated otherwise, use built-in functions when possible.
- Use `#funpar-text` for questions/clarification/etc. For private questions, direct message or email me.
- Good luck!

(Q1) **Two-Dimensional Arrays [10 points]** *Save your work in `twodee.rs`.* You will write a few Rust functions and test them thoroughly. Every function must be as parallel as possible.

A two-dimensional n-by-n array is often kept as a contiguous one-dimensional array. Specifically, the first row appears first, then the second row, then the third row, and so on. Hence, the following two-dimensional array

```
1, 3, 2, 5
4, 1, 7, 9
8, 6, 4, 3
7, 9, 2, 0
```

is stored as `[1, 3, 2, 5, 4, 1, 7, 9, 8, 6, 4, 3, 7, 9, 2, 0]`.

(a) **[5 points]** Write a function

```
pub fn puff(one_d : &[i32], n: usize) -> Vec<Vec<i32>>
```

that takes as input `one_d`, a one-dimensional representation of an `n`-by-`n` array, and returns a vector of vectors storing the array in the two-dimensional form. Hence, the output will be a vector of `n` vectors, each of length `n`. See some examples in the starter code.

(b) **[5 points]** Write a function

```
pub fn reflect(one_d: &[i32], n: usize) -> Vec<i32>
```

that takes an 2-d array encoded as 1-d and returns a vector representing the reflection of the 2-d array. In particular, the 1st row will become the 1st column, the 2nd row will become the 2nd column, and so on. For example:

```
1, 3, 2, 5              1, 4, 8, 7
4, 1, 7, 9    ====>     3, 1, 6, 9
8, 6, 4, 3              2, 7, 4, 2
7, 9, 2, 0              5, 9, 3, 0
```

More concretely, calling `reflect` on `[1, 3, 2, 5, 4, 1, 7, 9, 8, 6, 4, 3, 7, 9, 2, 0]` will result in `vec![1, 4, 8, 7, 3, 1, 6, 9, 2, 7, 4, 2, 5, 9, 3, 0]`. More examples are provided in the starter code.

(Q2) **Rising Runs [10 points]** *Save your work in* `risingruns.rs`. You will write a few Rust functions and test them thoroughly. Every function must be as parallel as possible and must run in $O(n)$ work.

In a sequence of numbers, a *rising run* is a contiguous part of the sequence that is only increasing. For example, in the sequence $3, 1, 4, 5, 2, 4$, we can find 3 rising runs: $[3]$, $[1, 4, 5]$, $[2, 4]$. Notice that, in fact, $[1]$, $[1, 4]$, $[1, 4, 5]$, $[4, 5]$ are all rising runs, but for this problem, we will be greedy and always look for the widest scope, hence only acknowledging $[1, 4, 5]$. For those who're mathematically inclined, these are known as maximal rising runs.

Curiously, every sequence can be broken down fully into one or more (maximal) rising runs. Let's look at a few examples:

```
[3, 1, 4, 5, 2, 4] ==> [[3], [1, 4, 5], [2, 4]]
[1, 2, 3, 4, 5] ==> [[1, 2, 3, 4, 5]]
[5, 4, 3, 2, 1] ==> [[5], [4], [3], [2], [1]]
[7, 11, 7, 11] ==> [[7, 11], [7, 11]]
[7, 11, 11, 7, 11] ==> [[7, 11], [11], [7, 11]]
[7, 11, 11, 15, 19] ==> [[7, 11], [11, 15, 19]]
[9, 9, 9] ==> [[9], [9], [9]]
```

(a) **[5 points]** Write a function

```
pub fn num_runs(xs: &[i32]) -> usize
```

that takes in a slice and returns the number of maximal rising runs found in the input slice. Examples are provided in the starter file.

(b) **[5 points]** Write a function

```
pub fn longest_run(xs: &[i32]) -> Option<i32>
```

that takes in a slice and returns the length of the longest maximal rising runs found in the input slice. Examples are provided in the starter file.

(Q3) **Go Home [10 points]** *Save your work in* `gohome.rs`. You will write a few Rust functions and test them thoroughly. Every function must be as parallel as possible.

In this question, we will solve a maze and return the number of minimum steps the path from `start` will take to reach the `goal` using the Breadth-First Search (BFS) algorithm. On a high level, the BFS algorithm will iteratively expand the search space starting from your initial position until it reaches the goal.

Our input maze will consist of the text-based board, where the top-left corner represents coordinate $(0, 0)$. The symbol `x` on the board is a wall, while a blank space is a walkable path. The code you write should be able to return **the minimum distance** from the starting coordinate `m` to the home coordinate `h`.

Below is an example of a board that returns 11 as the minimum distance back home.

```
xxxxxxxxxxx
x hx   x mx
x    x    x
xxxxxxxxxxx
```

To do so, your algorithm should maintain a HashSet that keeps track of the current coordinates that can be marked as the "frontier" and another HashSet that marks locations that you have "seen". Originally, the frontier will consist of the starting position. Then, the search algorithm will collect all the possible neighbor positions and insert these positions into the "frontier" and mark the searched location as seen.

(a) **[2 points]** Write a function

```
pub fn locate(board: &[&[u8]], target_ch: u8) -> Option<(usize, usize)>
```

that takes in a board and a target character and returns the coordinate for that symbol on the board. For example, when I call locate(board, 'h') using the board above, the function should return `Some((1,2))` because home is at row 1 column 2. **Hint:** `find_map_any` might be useful.

(b) **[3 points]** Write a function

```
fn nbrs(board: &[&[u8]], row: usize, col: usize) -> Vec<(usize,usize)>
```

that is a helper function inside to generate the viable neighbor coordinates given the board, a row, and a column. The returning vector should only return locations that can be visited. Do not include coordinate with a wall (`x`) on the board.

(c) **[5 points]** Finish the function

```
pub fn dist_home(board: &[&[u8]]) -> Option<usize>
```

that takes in a 2D board and returns the **minimum** distance from the starting location to the home coordinate.

To help you in this question, we have included the function `to_board` that allows you to process the input string and return a 2-dimensional board.

Hint: HashSet has several parallel APIs that will be useful here that includes `par_iter()` that returns a parallel iterator for items in the set and `par_extend` that takes an iterator (can be parallel iterator) and performs parallel insertion to the HashSet by extending the set.