HW2-2022

Homework 2

Due: 11:59pm, Thursday, Sep 15, 2022

100 points

For all Racket assignments in this course, set the DrRacket Language to `Advanced Student` (under `How to Design Programs`). Your assignment will be graded using the specified language level. If you use a different language level, your code may not work when it is graded.

For all Racket assignments in this course, set the DrRacket Language to **Intermediate Student with lambda** (under **How to Design Programs**). Your assignment will be graded using the specified language level. If you use a different language level, your code may not work when it is graded.

- Carefully follow the **Sample Solution to a Programming Problem** in the **Racket HW Guide**. Only half of the credit for each programming problem is based on the the correctness of your code as determined by our test cases. Much of the grade is based on how well you follow the *design recipe*. For a crisp example of what an ideal solution looks like look at the **Sample Solution to a Programming Problem** that sorts **list-of-number** at the end of **Sample Solution to a Programming Problem** in the the **Racket HW Guide**. This process may appear painful but it shows "in the small" how program design should be done "in the large". It is not difficult. If you carefully inspect the sample program, it shows the level of detail in describing your program design (and its derivation!) that we want.

Do the following programming problems using only the primitives mentioned in Lectures 2-5. Do not use the functions in the Racket library if they are not mentioned in Lectures 2-5.

1. [30 pts] Section 14.2 in the HTDP First Edition book describes what it calls Binary Search Trees. The terminology in this section of the book is non-standard because the Binary Search Trees contain both keys and values in each node and hence represents a finite mapping from keys to nodes. The stub file contains describes a simple Racket programming problem (with a solution consisting of only a few lines of executable Racket code) based on essentially the same inductive data definition as Binary Search Trees but the type of the **`value`** field is parametric (**`alpha`**) which must be instantiated to **`symbol`** to match the explication of Binary Search Trees in the book. Your task is to
   - Give some examples of the **`symbol-BSTM`** type.
   - Devise a set of test cases (input output pairs expressed using check-expect) for the **`getBSTM`** function.
   - Write a Template Instantiation for **`getBSTM`** (based on the general template for functions that process **symbol-BSTM**s)
   - Develop the code for the function **`getBSTM`** that satisfies the contract given in the stub file.
   - Briefly compare the asymptotic worst case running time of searching a **`symbol-BSTM`** that is well balanced (maximum depth is proportional to the log N where N is the number of keys in the **`symbol-BSTM`**) and function searches an ordered list of **`(key value)`** pairs represented as two element **list**s (as in Problem 2 below).

   Each of these five subtasks, except for devising the collection of test cases, takes only a few lines. A good set of test cases might take as many as 10 lines.

2. [30 pts] The stub file HW02.rkt provides a detailed description of how to develop the function **`cross`** (and supporting function **`cross-help`**) that consumes a **`number-list`** and a **`symbol-list`** and produces a **`number-symbol-pair-list`** where a **`number-symbol-pair`** is represented by a two element **`list`** containing a **`number`** and a **`symbol`**.

3. [30 pts] The stub file HW02.rkt provides a detailed description of how to develop the function **`merge`** (and supporting function **`merge-help`**) that consumes two ascending (technically non-descending} **`number-list`**s and merges them to form an ascending **`number-list`**.

4. [10pts] The ubiquitous Fibonacci function defined by the trivial **`fib`** program given in the stub tile HW02.rkt is interminably slow (exponential running time) for large inputs. Develop a Racket function **`fastFib`** that consumes a natural number input **`n`**, produces the same answer as the **`fib`** function defined in the stub file, and runs in linear time (assuming that the primitive addition operation runs in constant time, which fails for very large **`n`**). Hint: write a help function **`fastFibHelp`** that accumulates the result in an accumulator argument performing essentially the same computation as an imperative program relying on a loop that maintains **`fib(k-1)`** and **`fib(k-2)`** in mutable variables as k increases from 2 to n. The poor efficiency the trivial functional program for `fib` is due to the fact that it repeatedly computes the Fibonacci function for small k exponentially many times.

   a. Show Type Contracts, Purpose Statements, Examples, and Template Instantiations for **`fastFibHelp`** and **`fastFib`**. (The answers for the Template Instantiations can vary; only the salient features (primarily recursive calls) are matter.)
   b. As usual testing comes for free given that you provided input-output examples. Make sure that after you run your program that no source code text (definitions of **`fastFib`** and **`fastFibHelp`**) is shaded in the DrRacket definitions panel. Such shading indicates a failure to evaluate the shaded expressions in any test cases.

- **Optional problem for extra credit:** [50 pts (a challenging problem)]
  The Fibonacci function *`fib`* is defined in the stub for Problem 4 in HW02.rkt. The naive program for computing *`fib`* coded in the file HW02.rkt runs in exponential time, *i.e.* the running time for **`(fib n)`** is proportional to $C*b^n$ for some proportionality constant C and base b > 1. It is straightforward to write a program that computes **`(fib n)`** in time proportional to n as assigned in Problem 4. Your challenge is to write a program that computes **`(fib n)`** in *`log`* n time assuming that all multiplications and additions take constant time (which is unrealistic for large n). More precisely, your program should compute **`(fib n)`** using only `O(`*`log`* `n)` addition and multiplication operations (less than C*`log` n operations for some constant C).

  Hints:

- Derive a recurrence for *fib(2*m)* in terms of *fib(m)* and *fib(m-1)*. Derive a similar recurrence for *fib(2*m+1)*. To produce an algorithm that runs in log operations you need to reduce computing the pair (*fib(2*m),fib(2*m-1))* to computing (*fib(m),fib(m-1))* using a bounded number of arithmetic operations and tests.
- Initially write a program that works when n is a power of 2. Then refine this prototype to a program that works for all n based on determining whether n is even or odd.
- This is a challenging problem. Make sure that you have thoroughly completed the regular homework problems before attempting it.
- In my solution, I used "dotted pairs" to reduce overhead. The "dotted pair" representation of a pair (a,b) is (cons a b) which is illegal in all of the HTDP dialects when b is not a list, It is supported in the "other language" called "Pretty Big". This design choice is a bit of a stunt to minimize obvious overhead. Of course you can define pairs using `(define-struct pair (left right))` without compromising the goal of a solution that only requires `O(log n)` operations. My intuition was that such pairs have more overhead than dotted pairs but I did not perform any benchmark comparisons. If you decide to use a language other than **Intermediate student with lambda**, please put your solution to the challenge problem in a separate file called Chal02.txt and put a comment in your regular solution file HW02.rkt for problem 5 to that effect.