`tag` — Version 1.2 ***due:*** *tue, apr 24 @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

In this homework, you'll design and implement data structures and algorithms that combine ideas from augmented trees, divide and conquer, and dynamic programming.

1.1 Submission

This assignment is distributed in a number of files in our `git` repository. Instructions on how to access that repository can be found at `http://www.cs.cmu.edu/~15210/resources/git.pdf`. This is how assignments will be distributed in this course.

This assignment requires that you submit both code and written answers.

Submit your solutions by placing you solution files in your handin directory, located at

```
/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn8/
```

Name the files exactly as specified below. You can run the check script located at

```
/afs/andrew.cmu.edu/course/15/210/bin/check/08/check.pl
```

to make sure that you've handed in appropriate files. Do not submit any other files.

Your written answers must be in a file called `hw08.pdf` and must be typeset. You do not have to use LaTeX, but if you do, we have provided the file `defs.tex` with some macros you may find helpful.

For the programming part, you're handing in a total of 6 files:

```
NaiveIM.sml     SmartIM.sml     TestIM.sml
NaiveTycoon.sml    SmartTycoon.sml     TestTycoon.sml
```

These files must contain all the code that you want to have graded for this assignment and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.2 Naming Modules

The questions that follow ask you to organized your solutions in a number of modules. Your modules must be named exactly as stated in the handout. Correct code inside an incorrectly named structure, or a structure that does not ascribe to the specified signatures, *will not receive any credit*.

You may not modify any of the signatures or other library code that we give you. We will test your code against the signatures and libraries that we hand out, so if you modify the signatures your code will not compile and you will not receive credit.

1.3 The SML/NJ Build System

This assignment includes a substantial amount of library code spread over several files. The compilation of this is orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found at on the website at `http://www.cs.cmu.edu/~15210/resources/cm.pdf`.

1.4 Style

As always, you will be expected to write readable and concise code. If in doubt, you should consult the style guide at `http://www.cs.cmu.edu/~15210/resources/style.pdf` or clarify with course staff. In particular:

1. **If the purpose or correctness of a piece of code is not obvious, document it.** Ideally, your comments would convince a reader that your code is correct.
2. **You will be required to write tests for any code that you write.** In general, you should be in the habit of thoroughly testing your code for correctness, even if we do not explicitly tell you to do so.
3. **Clearly indicate parallelism in your code.** Use the provided `par` and `par3` functions in the library structure `Primitives` for this purpose.

2 Interval Minimum

The *interval minimum* (IM) problem is: Given a sequence A and positions L and R, compute the minimum value between index L and index R; i.e., it computes $\min_{L \le i \le R} A[i]$.

2.1 Naive Implementation

Task 2.1 (5%). Implement the function

```
intervalMin :  int seq -> (int * int) -> int
```

in the structure `NaiveIM` in `NaiveIM.sml`. Your solution to this task should run in $O(n)$ work and $O(\log n)$ span, where $n = |S|$ is the length of the sequence S. This function should be straightforward to implement but will be useful in testing your code for the next part.

Of course, there are many ways to correctly implement this function within the given cost bounds. While perhaps not the most straightforward solution, you might consider developing a divide-and-conquer algorithm which will help your thinking for the next part of the problem.

2.2 And Now For Something Smarter

For each sequence A, we are often interested in solving the interval minimum problem on multiple pairs (L,R). In this part, you'll implement a staged function that performs some preprocessing so that subsequent queries can be answered more efficiently.

Task 2.2 (25%). Implement the function

```
intervalMin :  int seq -> (int * int) -> int
```

in the structure `SmartIM` in `SmartIM.sml`. Again, consider the simplest divide-and-conquer algorithm for finding the minimum element of a sequence — how can you use augmented trees? For full credit, You must *stage* `intervalMin` so that the following bounds hold:

```
val S_mins = intervalMin S          O(n) work, O(log n) span
val minLR = S_mins (L,R)            O(log n) work, O(log n) span
```

where n is the length of S. You may wish to examine the solutions to assignment 4 as a reminder on how to properly stage a function.

Task 2.3 (5%). Complete the structure `TestIM` in `TestIM.sml` which tests `SmartIM.intervalMin` against `NaiveIM.intervalMin`.

3 Doing Jon's Homework

Jon was supposed to write this 15-210 assignment, but he's behind on his competition programming practice. Jon decides to kill two birds with one stone; you can do the competition programming for him. Here's the problem: http://acm.sgu.ru/problem.php?contest=0&problem=311 (which, for your convenience, is also reproduced below, with appropriate edits)

> **Ice Cream Tycoon.** You've recently started an ice-cream business at Carnegie Mellon University. During the day you have many suppliers delivering ice-cream to you, and many students buying it from you. You are not allowed to set the prices, as you are told the selling price for each serving of ice-cream by the suppliers.
>
> The day is described with a sequence of queries. Each query can be either `ARRIVE` n c, meaning that a supplier has delivered n servings of ice-cream each costing c DineX, or `BUY` n t, meaning that a student wants to buy n servings of ice-cream with a total of t DineX. The latter is processed as follows: if your n cheapest servings of ice-cream cost no more than t in total, you sell those n cheapest servings to the student; otherwise, the student gets nothing. You start the day with no ice-cream in stock.
>
> For each student, output TARTAN if she gets her ice-cream, and SPARTAN if she doesn't.

Example:

input	output
`ARRIVE 1 1`	`TARTAN`
`ARRIVE 10 200`	`SPARTAN`
`BUY 5 900`	`TARTAN`
`BUY 5 900`	
`BUY 5 1000`	

Note that in the above example, even if the first buy request becomes `BUY 5 1000`, none of the outcomes change. First of all, Jon has revised the spec for the homework with the following signature:

```
signature TYCOON =
  sig
    type t

    val empty : unit -> t
    val arrive : (t * (int * int)) -> t
    val buy : (t * (int * int)) -> (bool * t)
  end
```

The idea is that the type `t` is some (persistent) internal state which your algorithm keeps track of. Then, instead of mutating the state, you can pass it in and return it (this is analogous to how we pass around state for Random). The two functions `arrive` and `buy` correspond to the actions `ARRIVE` and `BUY` in the original problem.

- `arrive(s, (n,c))` means that given the state s, a supplier delivers n pieces of ice cream priced at c a piece. Thus, the function returns the state after this action.
- `buy(s, (n,t))` means that given the state s, a student with budget t seeks to buy n pieces of ice cream. Therefore, the function returns a Boolean value indicating whether the student got the ice cream and the resulting state.

Task 3.1 (5%). Implement `empty`, `arrive` and `buy` in `NaiveTycoon.sml`. They should each run in work and span O(number queries so far).

Task 3.2 (25%). Implement `empty`, `arrive` and `buy` in `SmartTycoon.sml`. They should each run in work and span O(log(number queries so far)).
(*Hint:* `Treap.sml` is a fine upstanding example of a balanced binary search tree, well worth emulating. What information do you need to store to support `arrive` and `buy`?)

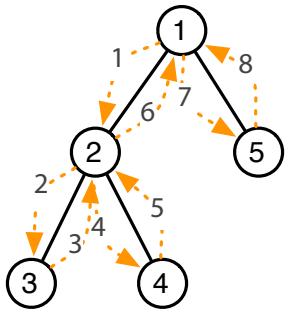
Task 3.3 (5%). Test `SmartTycoon.sml` against `NaiveTycoon.sml`. Put your tests in `TestTycoon.sml`.

4 LCA and LLR

Do not write code for this question. Write clear English.

You may be wondering, "What does any of this have to do with anything we talked about in class?" The other TAs sure were. Jon thought, and thought, and eventually Kanat came up with something:

The *least common ancestor* (LCA) problem is: given a rooted tree on n vertices and vertices x, y in a tree, compute the root of the smallest subtree that contains both x and y (i.e. their least common ancestor). There is a straightforward $O(n)$ work/span, $O(n)$ space algorithm to solve this problem. We can do better.



First, we need a different way to think about our tree: the linked list representation (LLR). The LLR of a tree is a **circular** linked list of all the edges in the tree so that if you walked around all the edges in the order they're listed you would visit every vertex, use each edge exactly once in each direction, and get back to where you started (the root).

For example, the tree shown on the left has an LLR $(1,2)$, $(2,3),(3,2),(2,4),(4,2),(2,1),(1,5),(5,1)$, which then wraps around. An important property of LLRs is that each edge is listed twice, once "forwards" and once "in reverse."

There are many ways to construct an LLR, but the following simple approach is easy to carry out in parallel: Let the neighbors of a vertex v be given as $N_v[0]$, $N_v[1],\ldots,N_v[d(v)-1]$. For each edge $e=(x,y)$, the next edge in the LLR is $(y, N_y[(i+1) \bmod d(v)])$, where i is the index such that $x = N_y[i]$. This computation is highly parallel because we can determine which edge comes after an edge (x,y) in the LLR sequence knowing only the neighbors of x and y, so we can compute the "next" pointer of each edge all in parallel.

Task 4.1 (10%). Prove that if you walk all the edges of the tree according to this LLR, you'll visit every vertex, traverse every edge exactly once in each direction, and return to the starting vertex.

Tree Representation: For the following tasks, we'll assume that the tree T is given as a sequence of integer sequence (`int int seq`), where the integer sequence $T[i]$ lists all the neighbors of i. In this representation, the vertices are numbered between 0 and $n-1$. Furthermore, an edge (x,y) appears in both $T[x]$ and $T[y]$.

4.1 Tree Rooting (Revisited)

As a warm-up exercise for the LCA problem, we'll take a second look at the tree rooting problem. Recall from Midterm II that the tree rooting problem is: given an undirected tree T (in the format above) and a node r, find a tree on the same set of vertices where each of the original edges is directed away from the root r.

The LLR provides a nice way to root the tree. Suppose you had an unrooted tree (edges in both directions) and a root. You could still compute the LLR as above. We saw earlier that each edge appears twice in the LLR: once "forwards" (i.e., away from the root) and once "backwards" (i.e., towards the root). If we break the LLR at the root node r, the tree rooting problem is exactly the problem of picking the forward copy of each edge.

Task 4.2 (5%). How could you use ideas from list ranking to root a tree, given its LLR? (Hint: remember that you can do plus scan on a length-n linked list in $O(n)$ work and $O(\log n)$ span)

Task 4.3 (5%). How could you compute the depth of each vertex using ideas from list ranking, given the LLR?

4.2 It's LCA Time!

Task 4.4 (10%). We're ready. Describe a staged function `LCA` (T, r) (x, y) that computes the LCA of x, y in a tree T which is rooted at r. Your function should meet the following bounds:

`val T_LCA = LCA(T,r)`	$O(n)$ work, $O(\log^2 n)$ span
`val ancestor = T_LCA x y`	$O(\log n)$ work, $O(\log n)$ span

(Hint: Use the LLR and interval minimum question)