Lab 2 - *SkylineLab*

Due: *Mon, Feb 3rd, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Spring '14)

1 Introduction

This assignment is meant to help you practice implementing, analyzing, and proving the correctness of a divide and conquer algorithm, and to familiarize you with the sequence `scan` operation. In this assignment, you will code a solution to The Pittsburgh Skyline Problem, which will help you learn all of those stated goals. We will go over `scan` in more detail this week in lecture.

You will likely find this assignment more conceptually challenging than the previous assignments. We recommend starting early so you have time to ask for help.

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf skylinelab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `support/MkReferenceSkyline.sml`
3. * `MkSkyline.sml`
4. * `Tests.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

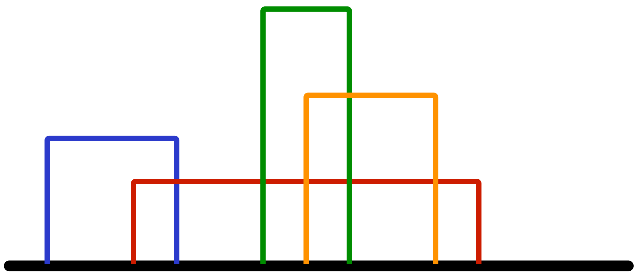
To submit your assignment to Autolab, open a terminal, `cd` to the `skylinelab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "*Handin your work*" link.

4 The Pittsburgh Skyline Problem

Did you know the Pittsburgh skyline was rated the second most beautiful vista in America by USA Weekend in 2003? However, you won't get to go out and see it any time soon, because you're an overworked Carnegie Mellon student. The 15-210 staff felt sorry for you, so we recorded the location and the silhouette of each building in Pittsburgh. Using this data, you can calculate the silhouette of Pittsburgh's skyline.



In this instance of the problem, we will assume that each building silhouette b is a two-dimensional rectangle, represented as the triple (ℓ, h, r). The corners of the rectangle will be at $(\ell, 0)$, (ℓ, h), (r, h), and $(r, 0)$. That is, the base of the building runs along the ground from $x = \ell$ to $x = r$, and the building is h tall (contrary to popular belief, the city's ground is flat).

Definition 4.1 (The Pittsburgh Skyline Problem)**.** Given a non-empty set of buildings $B = \{b_1, b_2, \ldots, b_n\}$, where each $b_i = (\ell_i, h_i, r_i)$, the *Pittsburgh skyline problem* is to find a set of points $S = \{p_1, p_2, ..., p_{2n}\}$, where each $p_j = (x_j, y_j)$ such that

$$S = \left\{ \Big(x, \max(h : (\ell, h, r) \in B \wedge x \in [\ell, r)) \Big) : x \in \bigcup_{(\ell,h,r) \in B} \{\ell, r\} \right\}$$

In other words, the x-coordinates expressed in S are exactly the x-coordinates expressed in B, and the y-coordinate for every element $(x_j, y_j) \in S$ is the height of the tallest building $b_i = (\ell_i, h_i, r_i)$ for which $\ell_i \leq x_j < r_i$. The input set of buildings is represented as an unsorted sequence of tuples, and the output set is represented as a sequence of points sorted by x-coordinate.

4.1 Logistics

For this problem you will hand in file `MkSkyline.sml`, which should contain a functor ascribing to the signature `SKYLINE` defined in `SKYLINE.sml`. Your functor will take as a parameter a structure ascribing to the signature `SEQUENCE`, and implement the function

```
val skyline : (int * int * int) seq -> (int * int) seq
```

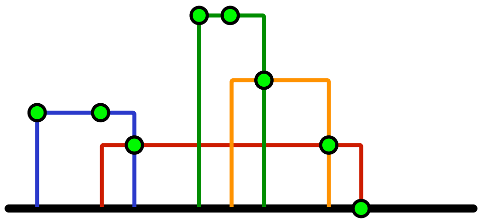
Input sequence:

ℓ_l	h_i	r_i
1	3	4
3	2	11
6	6	8
7	4	10

Output sequence:

x_j	y_j
1	3
3	3
4	2
6	6
7	6
8	4
10	2
11	0

Output points shown on input buildings:



4.2 Implementation Instructions

There are many algorithmic approaches to this problem, including brute force, sweep line, and divide and conquer. you will implement a work-optimal, low-span, divide-and-conquer solution to this problem. Even with divide and conquer, there are several choices on how to divide the problem. For example, you can divide along the x-axis, finding the skylines for $x < x'$ and for $x \geq x'$, or you can divide along the y-axis, finding the skylines for tall buildings and for short buildings. Alternately, you can divide the sequence of buildings into two sequences with a nearly equal number of buildings. You will use this last approach because it is easier to balance the recursive computations.

To simplify your solution, you may assume that all x-coordinates given in the input are unique, and that all heights and x-coordinates are non-negative integers.

You may have noticed that the definition of a skyline output sequence includes redundant points, and that removal of points with the same y-coordinate as the previous point would still result in a fully defined skyline. For example, the two points $(3,3)$ and $(7,6)$ in the example above are redundant. **Omit these points from the sequence returned by your `skyline` function.**

Let $W_{combine}(n)$ denote the work for the combine step of your divide-and-conquer algorithm. Then the work of your entire algorithm must satisfy the recurrence:

$$W_{skyline}(n) = 2W_{skyline}(n/2) + O(1) + W_{combine}(n) \qquad (1)$$

Task 4.1 (50%). Implement a divide and conquer solution to The Pittsburgh Skyline Problem. We have provided you with some skeleton code, but using it is completely optional. For full credit, **your solution must have $O(n \log n)$ work and $O(\log^2 n)$ span**, where n is the number of input buildings.

Note that the `copy_scan` function discussed in lecture may be helpful in your combine step. Also, please use the `par` function for parallel function calls.

Your score for this problem will be determined by some composition of public and private tests, as well as algorithmic correctness, efficiency, and code style. Keep in mind that you may receive **zero credit** for this task if you do any of the following:

- Hard code to the public tests
- Submit a sequential solution
- Submit the reference solution or a similar solution as your own

Task 4.2 (5%). In `Tests.sml`, write and submit test cases for your skyline implementation. To aid with testing we have provided a testing structure, `Tester`, which looks at `Tests.sml`, and greatly simplifies the testing process.

At submission time there must not be any testing code in the same file as your `SKYLINE` implementation, as it can make it very difficult for us to test your code. Additionally, unlike the 15-150 testing methodology, our testing structure does not test your code at compile time. To test your implementation:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
- Tester.testSkyline ();
```

4.3 Proofs

In 15-150, you were encouraged to write correctness proofs by stepping through your code. **Do not do this in 15-210.** For complicated programs, that level of detail quickly becomes tedious for you and your TA. A helpful guideline is:

Prove that your algorithm is correct, not your code.

Your proof should highlight and describe the critical steps of your algorithm, and discuss straightforwardly yet in sufficient detail how it maps onto your code.

Task 4.3 (30%). Prove the correctness of your divide-and-conquer algorithm by **structural induction**. Be sure to carefully state the theorem that you're proving and to note all the algorithmic steps in your proof. You should use the following structural induction principle for abstract sequences:

Let P be a predicate on sequences. To prove that P holds for every sequence, it suffices to show the following:

1. $P(\langle\,\rangle)$ holds
2. For all x, $P(\langle x\rangle)$ holds
3. For all sequences S_1 and S_2, if $P(S_1)$ and $P(S_2)$ hold, then $P(S_1 @ S_2)$ holds

Look at the next page for an example proof of algorithmic correctness.

Task 4.4 (15%). Carefully explain why your divide-and-conquer algorithm satisfies the recurrence (1) and **prove a closed-form solution** to the recurrence.

The following is a sample proof of correctness for the `parenMatch` *function given in recitation. Your proof for your Skyline algorithm should not be more technical than this—again, you are proving your algorithm, not the intricacies of the code.*

Parenthesis matching problem: For any string of parentheses s, determine if s is matched, where s is matched if and only if for every parenthesis p in s, there is a unique matching parenthesis $p' \neq p$ such that p closed $\implies$ p' open and p' before p in s, and p open $\implies$ p' closed and p' after p in s.

Proof. Claim that `parenMatch` solves the parenthesis matching problem on all sequences.
First, we define the **parenthesis match count problem**: for any string of parentheses s, find the number of unmatched closed and open parentheses in s (where matching is defined as above).
The parenthesis matching problem is a special instance of this new problem where $i = j = 0$ so the match count problem is a strengthened version of the original.
Let $\mathcal{P}(s)$ = "`pm s` solves the parenthesis match count problem on s".
We will prove $\mathcal{P}(s)$ by structural induction on s. Showing:

1. $\mathcal{P}(\langle\,\rangle)$.

   The empty string has no unmatched parentheses, so `pm S` must return `(0, 0)`.
   In the `EMPTY` case, `pm S` returns `(0, 0)`, so the empty case holds.

2. For all parentheses p, $\mathcal{P}(\langle p \rangle)$.

   Let p be either `OPAREN` or `CPAREN`.
   If p is `OPAREN`, then in the `ELT` case `pm S` returns `(0, 1)`.
   If p is `CPAREN`, then in the `ELT` case `pm S` returns `(1, 0)`.
   In both cases, `pm S` accurately describes the number of unmatched closed/open parentheses.

3. For all parenthesis sequences S_1, S_2, $P(S_1) \wedge P(S_2) \implies P(S_1 @ S_2)$.

   Let S_1, S_2 be arbitrary and fixed parenthesis sequences.
   Assume by the IH that `pm S1 = (i, j)` and `pm S2 = (k, l)` are the correct numbers of unmatched closed/open parentheses.
   Note that when concatenating S_1 and S_2, only unmatched open parens in S_1 and unmatched closed parens in S_2 can be matched.
   Two cases:

   - $j > k$

     In this case, we have more unmatched open parentheses in S_1 than unmatched closed parentheses in S_2, so we have $j - k$ remaining unmatched open parentheses in S_1 along with l unmatched open parentheses in S_2.

     We still have i unmatched open parentheses, so we have $(i, l + j - k)$ remaining unmatched parentheses. `pm S = (i, l + j - k)` if $j > k$, so this case holds.

   - $j \leq k$

     In this case, we have more unmatched closed parentheses in S_2 than unmatched open parentheses in S_1, so we have $k - j$ remaining unmatched closed parentheses in S_2 along with i unmatched closed parentheses in S_1.

We still have l unmatched open parentheses, so we have $(i + k - j, l)$ remaining unmatched parentheses. `pm S = (i + k - j, l)` if $j \leq k$, so this case holds.

Hence `pm (append(S1, S2))` correctly finds the number of unmatched closed/open parentheses in $S_1@S_2$.

Thus, `pm` solves the parenthesis match count problem on all parenthesis sequences s.

`parenMatch` only returns true if `pm S = (0, 0)`, so by extension `parenMatch` solves the parenthesis matching problem.

□