Chapter 1

SkylineLab

This assignment is designed to give you more practice with divide-and-conquer algorithms as well as provide an introduction to using the sequence function `scan`. You will analyze and solve the *skyline problem*, which is a geometric problem in 2 dimensions. This lab is conceptually difficult, so be sure to get started early!

1.1 Files

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

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

from a terminal window.

Some of the files worth looking at are listed below. The files denoted by * will be handed in by the submission script.

1. `Makefile`
2. * `MkSkyline.sml`
3. `Sandbox.sml`
4. `Tests.sml`

Additionally, you should create a file called `written.pdf` which contains the answers to the written parts of the assignment.

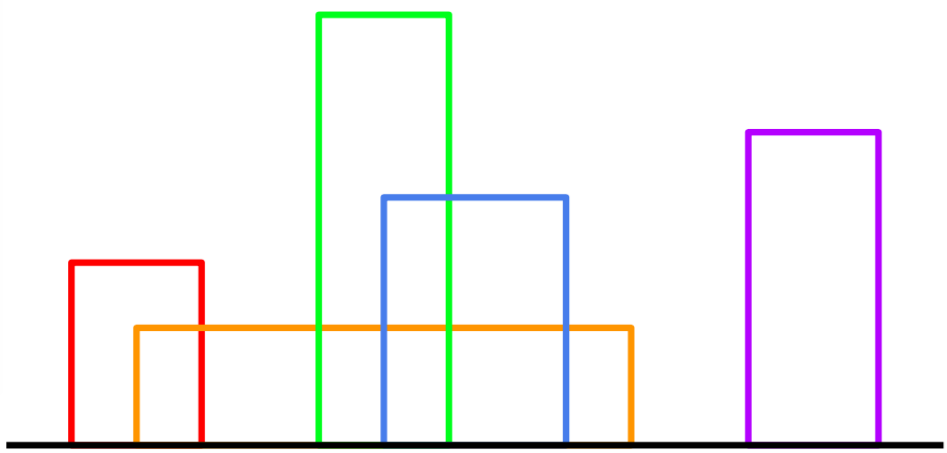
1.2 Submission

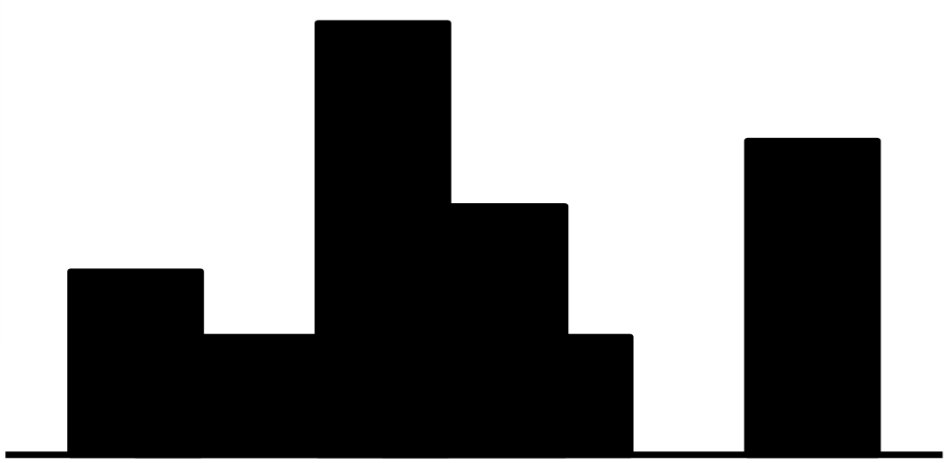
To submit the code portion of the assignment, run `make package` and upload the resulting `handin.tgz` to Diderot. You should ensure your code compiles by checking for a score of 0 under the `Compilation` section on Diderot. Note that Diderot can take up to around 10 minutes to compile code submissions.

Submit the written portion by uploading your `written.pdf` to Gradescope.

1.3 The Skyline Problem

Feast your eyes upon downtown Pittsburgh. From our perspective, most of the buildings are roughly rectangular (the exceptions being those like PPG Place), so let's assume from now on that all buildings are rectangles. We'll also assume that the ground is perfectly flat so that all the rectangles rest on the same line. The *skyline* of these rectangles is their silhouette.





Buildings. Under the assumptions given, each building can be represented by a triple (ℓ, h, r) which describes a rectangle with corners $(\ell, 0)$, (ℓ, h), (r, h), and $(r, 0)$. We will assume $\ell < r$ so that ℓ and r are the left and right sides of the building, and h is the height of the building.

Definition 1.1. The skyline of a set of buildings B is

$$\Big\{(x, H(x)) : x \in X \;\Big|\; x = \min X \;\vee\; H(x) \neq H(\text{prev}(x))\Big\}$$

where X is the set of all x-coordinates in B, $H(x)$ is the max height at x, and $\text{prev}(x)$ is the nearest x' to the left of x, given by

$$\begin{aligned} X &= \bigcup_{(\ell,h,r)\in B} \{\ell, r\} \\ H(x) &= \max\big\{h : (\ell, h, r) \in B \;\big|\; \ell \leq x < r\big\} \\ \text{prev}(x) &= \max\big\{x' \in X \;\big|\; x' < x\big\} \end{aligned}$$

In other words, the x-coordinates in the skyline are exactly those which appear in B, and the y-coordinate for any particular x is the height h of the tallest building (ℓ, h, r) for which $\ell \leq x < r$. We do not include **redundant** points: points whose height is unchanged from the nearest point to the left.

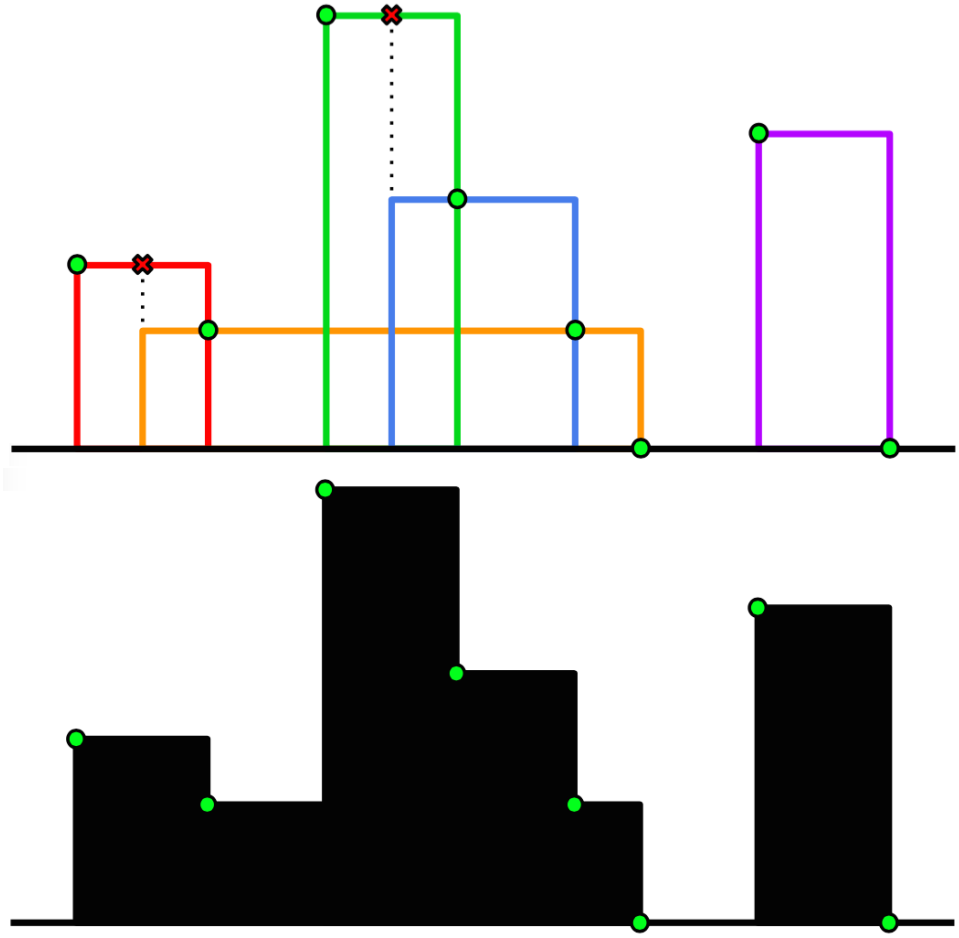
Example 1.1. Buildings:

ℓ	h	r
2	3	4
3	2	11
6	7	8
7	4	10
13	5	15

Skyline:

x	y
2	3
4	2
6	7
8	4
10	2
11	0
13	5
15	0

In the diagrams below, skyline points are marked with  . Redundant points are marked in the first diagram with  .



1.3.1 Implementation

We define `type skyline = (int * int) Seq.t`. We maintain the convention that all skylines are **sorted by x-coordinate**, and **do not contain redundant points**. Throughout, we assume that all x-coordinates are unique and non-negative, and that all heights are

strictly positive.

Given a sequence of buildings B in no particular order, your goal is to write a divide-and-conquer algorithm for the skyline problem of the following form:

```
fun skyline B =
  case Seq.splitMid B of
    Seq.EMPTY => Seq.empty ()
  | Seq.ONE x => singleton x
  | Seq.PAIR (L, R) =>
      combine (Primitives.par (fn () => skyline L,
                               fn () => skyline R))
```

`singleton` has been implemented for you in `MkSkyline.sml` and has constant work and span. All you have to do is implement `combine`.

Note. You may assume all x-coordinates in B are unique.

Remark. The behavior of this code is identical to the following, which uses `reduce` to "automate" the divide-and-conquer process.

```
fun skyline B =
  Seq.reduce combine (Seq.empty ()) (Seq.map singleton B)
```

Task (1). (60 points) In `MkSkyline.sml`, implement the following function.

```
val combine : skyline * skyline -> skyline
```

`combine (S1, S2)` should evaluate to the skyline of all buildings from both S_1 and S_2. Your implementation must have $O\left(|S_1| + |S_2|\right)$ work and $O\left(\log|S_1| + \log|S_2|\right)$ span.

Hint. You will probably find the functions `merge`, `scan`/`scanIncl`, and `filter`/`filterIdx` useful. In particular, take a close look at `scan`. For constant-work functions f, `scan f b s` has $O(|s|)$ work and $O(\log|s|)$ span. The cost bounds of `scanIncl` are identical.

Important. In `scan f b s`, the function f must be *associative*, meaning that for all inputs a, b, c,

$$f(f(a,b),c) = f(a,f(b,c))$$

Note. As usual, assume `Seq` = `ArraySequence` for cost analysis.

Note. Don't forget about removing redundant points!

1.3.2 Testing

It is very important that you thoroughly test your code before you submit. After the deadline, we will grade your code with a private set of test cases.

We provide two ways to test your code with SML/NJ.

1. In `Sandbox.sml`, write whatever testing code you'd like. You can then access the sandbox at the REPL:

   ```
   - CM.make "sandbox.cm";
   ...
   - open Sandbox;
   ```

2. In `Tests.sml`, add test cases according to the instructions given. Then run the autograder:

```
- CM.make "autograder.cm";
...
- Autograder.run ();
```

1.4 Performance Evaluation

Note. All tasks in this section should be included in your `written.pdf` submitted to Gradescope.

How efficient is your parallel code? To answer this question, we might begin by splitting it into two distinct issues:

1. How parallel is your code? That is, how much faster does it get as we use more processors?
2. How much extra work does your code do, in order to be parallel?

1.4.1 Introduction and Definitions

We can tackle these questions by measuring speedup and overhead. In particular, for a fixed input, let T^* be the amount of time it takes to solve that input using a fast sequential program, and let T_P be the amount of time it takes your parallel program on P processors (and similarly T_1 is your program's runtime on 1 processor). We can then compute:

1. The self-speedup on P processors, T_1/T_P.
2. The overhead, T_1/T^*.

Task (2)**.** (5 points) Ideally, what is a lower bound on T_P? Consequently, what is an upper bound on the self-speedup on P processors? Give your answers in terms of P and T_1.

Task (3)**.** (5 points) Is T_1 an upper bound on T^*? Briefly justify your answer.

1.4.2 Experiments

You will now experimentally measure T^*, T_1, and T_P. To facilitate these experiments, we've implemented a sequential skyline solution and a small testing harness in the `mpl/` subdirectory. The sequential solution, in `mpl/FastSequentialSkyline.sml`, will be used for calculating T^*. The parallel solution in `mpl/ParallelSkyline.sml` uses your `combine` function, and it will be used to calculate T_1 and T_P.

Note. You're welcome to read the sequential solution if you so desire, but it's not necessary to complete the assignment.

You can run experiments by `cd`'ing into the `mpl/` subdirectory, running `make skyline` to compile, and then running `./report`. This will run 5 trials of each program, reporting the runtime of each trial as well as the average (of the 5 trials). The test harness will automatically select a number of processors to use which is appropriate for the machine you are on.

Task (4)**.** (7 points) Run the experiments, and write down the reported values of T^*, T_1, and T_P. (For each, use the reported average of the 5 runs.) Also write down the number of processors P that were used for T_P.

Note. For reference, our T_1 is (depending on the machine) between 3 and 7 seconds.

Task (5)**.** (3 points) Calculate the self-speedup and overhead of your code, using the equations given in the 1.4.1 subsection.

Task (6)**.** (5 points) Read the code in `ParallelSkyline.sml` and describe a small tweak/optimization that could be made **in this file** that would improve your overhead. (1-2 sentences)

Note. The granularity threshold has already been tuned and does not need to be adjusted. The optimization you are looking for should be generic, and should work for any student in the class (it shouldn't be specific to your `combine` function).

1.5 Written Questions

1.5.1 Cost Analysis

Consider the functions *skyline*, *combine*, and *singleton* as described for Task 1.3.1. Assume *combine* is implemented correctly and meets the required cost bounds.

Task (7)**.** (5 points) Give an upper bound for $|combine\ (S_1, S_2)|$ in terms of $|S_1|$ and $|S_2|$.

Note. We use $|S|$ to refer to the length of the sequence S.

Task (8)**.** (5 points) Write the work and span recurrences of *skyline* B in terms of $n = |B|$. State the tight Big-O bound for each recurrence (don't show your work of solving them; we just want the bound).

Task (9)**.** (15 points) Suppose that *combine* (S_1, S_2) now has $O\left(|S_1| \log |S_1| + |S_2| \log |S_2|\right)$ work. Write the new work recurrence of *skyline* B in terms of $n = |B|$. Solve it using the **substitution method** and give a tight Big-O bound (show your work this time).

1.5.2 Finding a Lower Bound

Our skyline algorithm has $O\left(n \log n\right)$ work. But is this algorithm the fastest possible, asymptotically? In order to answer this question, we need to find a lower bound.

Remark. Oops, we just gave away the answer for part of an earlier task... oh well!

One technique for showing a *lower bound* on a problem is through a reduction, which lets us reuse known lower bounds for new problems.

It is well known that any comparison-based sorting algorithm requires $\Omega(n \log n)$ work. (If you haven't seen this proof before, a quick Google search should yield a wealth of useful information.) You will now show a reduction from the sorting problem to the skyline problem, thus proving a lower bound of $\Omega\left(n \log n\right)$ for any comparison-based solution to the skyline problem.

Task (10)**.** (15 points) Assume you have a black-box algorithm $skyline$ which solves the skyline problem. Write pseudocode for a function

$$sort \ : \ int \ seq \ \rightarrow \ int \ seq$$

which sorts the input. Note that any function calls to $skyline$ must satisfy any preconditions we assumed when implementing it above. Your solution must have $O\left(n + \mathcal{W}_{skyline}(n)\right)$ work. For the sake of simplicity, you may assume the input contains only non-negative numbers and no duplicates.

Note. Our solution is 4 lines long.

Task (11)**.** (5 points) Briefly explain why this reduction proves that $skyline$ must have $\Omega\left(n \log n\right)$ work.