`tag` — Version 1.0 ***due:*** *mon, sep 17 @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

This assignment is meant to help you practice implementing, analyzing, and proving the correctness of a divide and conquer algorithm, and familiarize you with the sequence `scan` operation. There are two separate algorithms to code in this assignment; the The Pittsburgh Skyline problem will work towards all of those stated goals, while the Binary Bignum problem will primarily test your command of the `scan` operation.

We will go over `scan` in more detail next week in lecture.

You will likely find this assignment more conceptually challenging than the first assignment. Expect it to take significantly longer than the first assignment to complete.

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>/assn2/
```

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

```
/afs/andrew.cmu.edu/course/15/210/bin/check/02/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 `hw02.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. This file should be included at the top of your `.tex` file with the command `\input{/<file>/<path>/defs.tex}`.

For the programming part, the only files you're handing in are

```
skyline.sml
skyline-test.sml
bignum.sml
bignum-test.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 below ask you to organized your solutions in a number of modules written from scratch. 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`.

2 Function specifications

A style guideline that worked well last semester: We're going to ask that you give every function (and other complex values, at your discretion) something like a *contract* (from 122) or a *purpose* (from 150).

We ask that you annotate each function with its *type*, any *requirements* of its arguments (if it's a partial function), and any *promises* about its return values. A familiar example might look like this:

```
(* match : paren seq -> bool
   [match S] returns true if S is a well-formed parenthesis sequence
    and false otherwise. It does so by invoking the match' helper
    function below
*)
fun match s =
    let
        (* [match' s : paren seq -> (int * int)
           As we showed in recitation, every paren sequence
           can be reduced (by repeatedly deleting the substring "()")
           to a string of the form ")^i(^j".
           [match s] uses a divide and conquer algorithm to return (i,j)
           for s.
        *)
        fun match' s =
            case (showt s) of
                EMPTY => (0,0)
            | ELT OPAREN => (0,1)
            | ELT CPAREN => (1,0)
            | NODE (L,R) =>
                let
                    val ((i,j),(k,l)) = par (fn () => match' L, fn () => match' R)
                in
                    (* s = LR = )^i(^j)^k(^l
                        so we cancel the pairs in the middle
                    *)
                    case Int.compare(j,k)
```

```
                    of GREATER => (i, l + j - k)
                    | _ => (i + k - j, l)
    end
in
    case (match' s)
    (* a sequence is well-formed iff it reduces to (0,0) *)
    of (0,0) => true
    | _ => false
end
```

If the purpose or correctness if a piece of code is not obvious, you should add a comment as well (for example, the code above makes note of the argument that every paren sequence can, for the purposes of `match`, be reduced to "`)^i(^j`"). Ideally, your comments would convince a reader that your code is correct.

3 Writing 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 that you should prove that your algorithm is correct, not your code. But your proof should highlight the critical steps and describe your algorithm in sufficient detail that your algorithm maps straightforwardly onto your code. Consider Theorem 3.2 of lecture 3, as an example of the level of detail we expect.

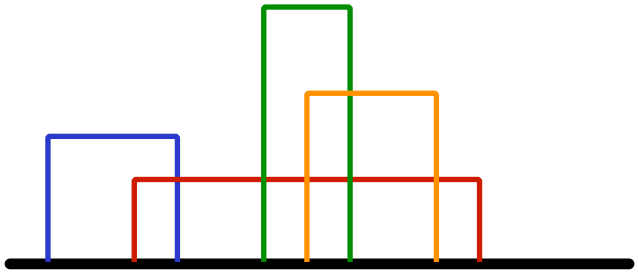
4 Warm up Recurrences

Task 4.1 (17%). Determine the complexity of the following recurrences. Give tight Θ-bounds, and justify your steps to argue that your bound is correct. Recall that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $g \in O(f)$. You may whatever method you choose to show your bound is correct, except you must use the substitution method for problem 3.

1. $T(n) = 3T(n/4) + \Theta(n)$
2. $T(n) = 21T(\frac{n-3}{23}) + \Theta(n)$
3. $T(n) = 2T(n/2) + \Theta(\sqrt{n})$ (Prove by substitution.)
4. $T(n) = \sqrt{n}T(\sqrt{n}) + \Theta(n^2)$.

5 Pittsburgh Skyline

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 tall 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; this happened last semester, when most of you were too busy with 15-251 to notice).

Definition 5.1 (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. An example of a skyline instance is given below.

5.1 Logistics

For this problem you will hand in file `skyline.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 the signature `SEQUENCE`, and implement the function `skyline:(int*int*int) seq -> (int*int) seq`.

5.2 Implementation Instructions

There are many algorithmic approaches to this problem, including brute force, sweep line, and divide and conquer. As you may have guessed, you will implement a work-optimal, low-span, divide-and-conquer

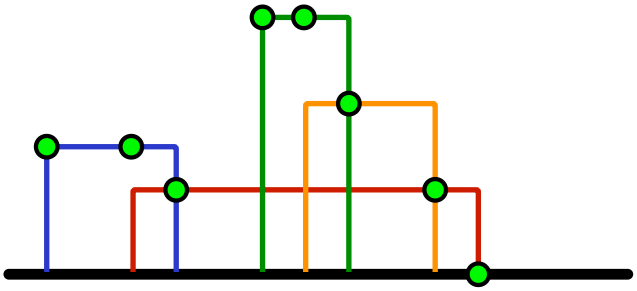
Input sequence:

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:



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.

To simplify your solution, you may assume that no two buildings have the same left or right x-coordinate and that the heights and x-coordinates are positive 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.**

The work of your divide-and-conquer steps must satisfy the recurrence

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

where $W_{combine}(n)$ denotes the work for the combine step.

Task 5.1 (30%).

Implement a divide and conquer solution to the Pittsburgh Skyline Problem. The starter code which you will pull from the Git handout system contains the file `skyline.sml` in which to put your `Skyline` functor. Be sure to remove redundant points from the sequence you return, as specified above.

For full credit, **we expect your solution to have $O(n \log n)$ work and $O(\log^2 n)$ span**, where n is the number of input buildings. Carefully explain why your divide-and-conquer steps satisfy the specified recurrence and prove a closed-form solution to the recurrence.

The `copy_scan`, which will be discussed next week in lecture, may be helpful in your combine step.

Task 5.2 (4%). Implement the function `all` in the functor `SkylineTest` in `skyline-test.sml`. `all` should thoroughly and carefully test your implementation of the `SKYLINE` signature, returning true if and only if your tests succeed. Tests should include both edge cases and more general test cases on specific sequences.

At submission time there should not be any testing code in `skyline.sml`, as it can make it difficult for us to read and test your code when you turn it in.

Task 5.3 (15%).

Prove the correctness of your divide-and-conquer algorithm by induction. Be sure to carefully state the theorem that you're proving and to note all the algorithm steps in your proof.

You can use (strong) mathematical induction on the length of the input sequence. That is,

> Let $P(\cdot)$ be a predicate. To prove that P holds for every $n \geq 0$, we prove:
>
> 1. $P(0)$ holds, and
> 2. For all $n \geq 0$, if $P(n')$ holds for all $n' < n$, then $P(n)$.

Or, you can 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, and
> 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.

6 Binary Bignum

Native hardware integer representations are typically limited to 32 or 64 bits, which can be insufficient for computations which result in very large numbers. Some cryptography algorithms, for example, utilize large primes that require over 500 bits to represent. This motivates the implementation of an arbitrary-precision integer types and the operations to support them.

In this problem, you will implement addition and subtraction for arbitrary-precision non-negative integers represented as sequences of bits. This is trivial to achieve with a sequential algorithm. Needless to say, in 15-210 you must find a lower span solution to this problem.

6.1 Implementation Instructions

We represent an integer with the type `bignum` which is defined as a `bit seq`, where

```
datatype bit = ZERO | ONE
```

We adopt the convention that `bignums` begin with their least-significant bit. Furthermore, `bignums` never contain trailing zeros—The `bignum` representing 0 is an empty sequence, and all other `bignums` must end in a `ONE`. For example, the value represented by the `bignum` $\langle 0, 1, 1 \rangle$ is 6, and the `bignum` $\langle 1, 0, 0 \rangle$ is invalid (it would represent 1 if the trailing zeros were removed). **You must follow this convention for your solutions.**

Our `bignum` implementation will support addition and restricted subtraction (where the result is undefined if the second operand is larger than the first). You will complete the functor `BigNum` in `bignum.sml`, which ascribes to the signature `BIGNUM`. To help you get started, the starter code already has the `bignum` type declared and the infix operators `++` and `--` defined for you.

6.1.1 Addition

Task 6.1 (20%). Implement the addition function

```
++ :  bignum * bignum -> bignum
```

in the functor `BigNum` in `bignum.sml`. For full credit, on input with m and n bits, your solution must have $O(m+n)$ work and $O(\lg(m+n))$ span. For reference, our solution has 40 lines with comments.

The main challenge in meeting the cost bound lies in propagating the carry bits. For example, try adding 1 to $(11101111111111)_2$ and you will see a "ripple effect." You should use `scan` to get around this, but you need to come up with an associative binary operator. As a hint, we have also provided you with an additional datatype which you may find useful:

```
datatype carry = GEN | PROP | STOP
```

where `GEN` stands for generate, `PROP` for propagation, and `STOP` for stop. You might want to work out a few small examples to understand what is happening. Do you see a pattern in the following example?

```
1000100011
1001101001 +
```

For more inspiration, you should refer to lecture notes on the Sequence `scan` operation.

6.1.2 Subtraction

Task 6.2 (10%). Implement the subtraction function

```
-- :  bignum * bignum -> bignum
```

in the functor `BigNum` in `bignum.sml`, where `x -- y` computes the number obtained by subtracting y from x. You may assume that $x \geq y$. You may also assume for this problem that ++ has been implemented as specified above, even if your ++ is incorrect or does not meet cost bounds. For full credit, if x has n bits, your solution must have $O(n)$ work and $O(\lg n)$ span. Our solution has 20 lines with comments.

Perhaps the easiest way to implement subtraction is to use *two's complement* representation for negation, which you should recall from 15-122 or 15-213. As a quick review: two's complement uses k bits to encode the integers -2^{k-1} to $2^{k-1}-1$. When the most significant bit, or "sign bit", of a two's complement value is zero, the remaining $k-1$ bits are directly interpreted as binary, and thus have a value from 0 to $2^{k-1}-1$. When the sign bit is one, the value represented is the binary interpretation of the $k-1$ bits minus 2^{k-1}, which is a value from -2^{k-1} to -1. A two's complement number can be negated by flipping all the bits and adding 1.

6.1.3 Testing

The `BIGNUM` signature exports `add` and `sub`; we've also provided you with utility functions to convert between `bignum` and SML's `IntInf` (the standard arbitrary precision integer type) which you should use in your tests.

Task 6.3 (4%). Implement the function `all` in the functor `BigNumTest` in `bignum-test.sml`. `all` should thoroughly and carefully test your implementation of the `BIGNUM` signature, returning true if and only if your tests succeed. Tests should include both edge cases and more general test cases on specific sequences.

Although not required, it would be good practice to implement random testing of your functions against `IntInf`'s add and subtract. You can use the `Random210` structure in the 210 library or SML-NJ's `Random` to perform random testing. Be aware that the SML-NJ's `Random` functions are not pure functions.

At submission time there should not be any testing code in `bignum.sml`, as it can make it difficult for us to read and test your code when you turn it in.