Homework 3

15-210: Parallel and Sequential Data Structures and Algorithms (Spring'12)

`tag` — Version 1.2 ***due:*** *thu, feb 16 @ 11:59pm* **(NO LATE DAYS)**

1 Introduction

In this homework, you will implement a number of operations on an arbitrary-precision representation of integers. You will also explore implementing work-optimal `SEQUENCE merge` with low span. You will use the Karatsuba multiplication algorithm, as well as `scan` with different associative binary operators.

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>/assn3/
```

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

```
/afs/andrew.cmu.edu/course/15/210/bin/check/03/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 `hw03.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, the only files you're handing in are

`karatsuba-bignum.sml`, `merge.sml`, and `lcis.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 BigNum Arithmetic

In this problem, you will implement functions to support *bignum*, or arbitrary-precision arithmetic. Native hardware integer representations are typically limited to 32 or 64 bits, which is sometimes insufficient for computations which result in very large numbers. Some cryptography algorithms, for example, require the use of large primes which require over 500 bits to represent in binary. This motivates the implementation of an arbitrary-precision representation which can support such operations.

2.1 Logistics

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 if x is a `bignum`, then x is non-negative, and x_0 is the least-significant bit. Furthermore, if x represents the number 0, x is an empty sequence—and if $x > 0$, the right-most bit of x must be `ONE` (that is to say, there cannot be any trailing zeros following the most significant `ONE` bit). **You must follow this convention for your solutions.**

Our `bignum` implementation will support addition, subtraction (assuming the number never goes negative), and multiplication. You will complete the functor `KaratsubaBigNum` in `karatsuba-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 `**`, `--`, `++` defined for you.

2.2 Specification

2.2.1 Addition

Task 2.1 (30%). Implement the addition function

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

in the functor `KaratsubaBigNum` in `karatsuba-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. 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 look at the `copy_scan` implementation in the notes for Lecture 5.

2.2.2 Subtraction

Task 2.2 (10%). Implement the subtraction function

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

in the functor `KaratsubaBigNum` in `karatsuba-bignum.sml`, where `x -- y` computes the number obtained by subtracting y from x. We will assume that $x \geq y$; that is, the resulting number will always be non-negative. You should also assume for this problem that ++ has been implemented correctly. 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. For a quick review: we can represent positive numbers in k bits from 0 to $2^{k-1}-1$, reserving the most significant bit as the "sign bit." For any integer x representable in k bits in two's complement, $-x$ is simply the number y such that $x+y=2^k$. Then, we can negate x by simply flipping all the bits and adding 1.

2.2.3 Multiplication

Task 2.3 (20%). Implement the function

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

in the functor `KaratsubaBigNum` in `karatsuba-bignum.sml`. For full credit, if the larger number has n bits, your solution must have $O(n^{\log_2 3})$ work and $O(\lg^2 n)$ span. You should assume for this problem that `++` and `--` have been implemented correctly. Our solution has 40 lines with comments.

Recall from lecture the divide-and-conquer solution to n-bit multiplication in $O(n^{\log_2 3})$ work, which you should use to implement `**`. This is the Karatsuba algorithm, invented in 1960 by Anatolii Alexeevitch Karatsuba. There is no guarantee that the pseudocode in the lecture notes is at all correct—use it with caution. We have provided you with the function

```
par3 :  (unit -> 'a) * (unit -> 'b) * (unit -> 'c) -> 'a * 'b * 'c
```

to indicate three-way parallelism in your implementation of Karatsuba.

2.2.4 Testing

The `BIGNUM` signature exports `add`, `sub`, and `mult`; we've also provided you with utility functions to convert between `bignum` and SML's `IntInf`, so you can test your implement at the REPL. Again, you will be expected to have tests for your code, but it is up to you how to write these tests.

3 Sequence Merge

Some of you have noticed that the ArraySequence library implementation of `merge` currently uses `append` followed by a `sort`. This has $O(N \log N)$ work and $O(\log N)$ span, which obviously does not fulfill our library cost specifications. Here N is the length of the combined sequence.

In this problem, you will patch our library with a parallel implementation of merge which *does* meet the cost bounds of $O(N)$ work and $O(\log^2 N)$ span, where N is the length of the combined sequence. Our solution (45 lines) actually meets the $O(\log N)$ span bound but we would like to give you some buffer.

You will use the `ArraySequence` implementation of `SEQUENCE` and the cost specifications given on the Resources page.

Task 3.1 (30%). Implement the function

```
merge :  'a ord -> 'a seq -> 'a seq -> 'a seq
```

where `merge cmp s t` takes two arrays sorted according to `cmp` and returns the sequence resulting from combining `s` and `t`, sorted by the comparison function `cmp`. The two input sequences may *not* have the same length.

There are many ways to go about solving this problem.We'll try to gear you towards a simple solution. Keep in mind the following when you attempt this problem:

(1) You may want to use divide and conquer, but be careful! Remember that increasing the branching factor can make the call tree shallower.

(2) Binary search is your friend.

(3) Look up the cost specifications of the library functions *before* you start thinking about your algorithm. Specifically, you may want to take a look at `append`, especially if you're planning to use it in a recursive divide and conquer algorithm. Hint: `append` has linear work and `flatten` is a cheaper alternative to concatenate multiple sequences together.

(4) How fast can you merge small sequences, say each of size $O(\log N)$, using a sequential merge?

Task 3.2 (10%). Prove that your algorithm has $O(N)$ work and $O(\log^2 N)$ span.

You do not need to step through every instruction of your code, but be sure to take into account the costs of any library functions you use. Cost specifications for them are available on the course website. You may state without proof that binary search on a sequence of size n takes $O(\log n)$ work and span, but be sure to implement it correctly—remember that splitting and concatenating an array sequence has linear work!

4 BONUS: Longest Contiguous Increasing Subsequence

Given a sequence of numbers, the *longest contiguous increasing subsequence* problem is to find the largest number of contiguous increases in a sequence of numbers. In this problem, you will find the number of increases in the longest contiguous increasing subsequence of a given sequence. For example,

$$\texttt{lcis}(\langle 7,2,3,4,1,8\rangle)$$

will return 2 since there are 2 increases in a row in the contiguous subsequence $\langle 2,3,4\rangle$.

Task 4.1 (0%). Implement the function

```
lcis :  int Seq.seq -> int
```

which returns the largest number of contiguous increases in the input sequence. For credit, your solution must use `scan` and have $O(n)$ work and $O(\lg n)$ span.