Full Name: ______________________________

Andrew ID: ______________ **Section:** ______________

15–210: Parallel and Sequential Data Structures and Algorithms

Exam I (Solutions)

February 2013

- There are 16 pages in this examination, comprising 6 questions worth a total of 120 points. The last 2 pages are an appendix with costs of sequence, set and table operations.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question. Clearly indicate your answers.
- You may refer to your one double-sided $8\frac{1}{2} \times 11$in sheet of paper with notes, but to no other person or source, during the examination.
- Your answers for this exam must be written in blue or black ink.
- **When writing code, you may use the following mathematical primitives:**
  - `val min : int * int -> int`
  - `val max : int * int -> int`
  - `val isEven : int -> bool`
  - `val isOdd : int -> bool`

	Sections	
A	10:30am-11:20am	Laxman Dhulipala, Bill Duff
B	2:30pm-3:20pm	Aakash Rathi, Shannon Williams
C	12:30pm-1:20pm	Chris Powell, Naman Bharadwaj
D	1:30pm-2:20pm	Julian Shun, Susan Wang
E	3:30pm-4:20pm	Vincent Siao

Question	Points	Score
Recurrences	15	
Unique Sequences	10	
Parentheses Revisited	20	
SkylineLab Reloaded	20	
Higher Order Costs	20	
Dynamic Shortest Paths	35	
Total:	120	

Question 1: Recurrences (15 points)

Recall that $f(n)$ is $\Theta(g(n))$ if $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$. Give a closed-form solution in terms of Θ for the following recurrences, where $f(1) = \Theta(1)$.

You do not have to show your work, but it might help you get partial credit.

(a) (3 points) $f(n) = f(n/4) + \Theta(n)$.

Solution: $\Theta(n)$.

(b) (3 points) $f(n) = 4f(n/4) + \Theta(n)$.

Solution: $\Theta(n \lg n)$.

(c) (3 points) $f(n) = 8f(n/2) + \Theta(n^2)$.

Solution: $\Theta(n^3)$

(d) (3 points) $f(n) = f(n/4) + \Theta(\lg^2 n)$.

Solution: $\Theta(\lg^3 n)$

(e) (3 points) (This might be hard.) $f(n) = 2f(\sqrt{n}) + \Theta(1)$.

Solution: $\Theta(\lg n)$

Question 2: Unique Sequences (10 points)

Write the function `uniquify` to eliminate duplicate integers in a sequence by using the sequence library only (no sets and tables). You may not assume anything about the order of the input, but you may find `Int.compare : int * int -> order` useful.

```
fun uniquify (s : int seq) =
```

Solution:

```
let
  val pairs = map (fn x => (x, ())) S
  val grouped = collect Int.compare pairs
in
  map (fn (x, _) => x) grouped
end
```

State the work and span of `uniquify` in terms of the length, n, of the input sequence.

$W(n) =$

Solution: $W(n) = O(n \lg n)$

$S(n) =$

Solution: $S(n) = O(\lg^2 n)$

Question 3: Parentheses Revisited (20 points)

A parenthesis expression is called *immediately paired* if it consists of a sequence of open-close parentheses — that is, of the form "()()()() … ()".

(a) (10 points) **Longest immediately paired subsequence (LIPS) problem.** Given a (not necessarily matched) parenthesis sequence s, the longest immediately paired subsequence problem requires finding a (possibly non-contiguous) longest subsequence of s that is immediately paired. For example, the LIPS of "(((((((()()()))))()(((()(())" is "()()()()()()" as highlighted in the original sequence.

Write a function that computes the *length* of a LIPS for a given sequence. Your function should have $O(n)$ work and $O(\lg n)$ span.

(**Hint:** Try to find a property that simplifies computing LIPS. This problem might be difficult to solve otherwise.)

```
fun findLIPS (s: paren seq) : int = (* Work = O(n), Span = O(lg n) *)
```

Solution: The algorithm simply extracts immediately paired parentheses and counts them. We prove below why this is sufficient.

```
fun findLIPS (s: paren seq) =
    let
      fun isIP i =
          case (nth s i, nth s (i+1))
            of (LPAREN, RPAREN) => 1
             | _ => 0
      val nIPs = reduce op+ 0 (tabulate isIP (length s - 2))
    in
      2 * nIPs
    end
```

(b) (10 points) Prove succintly that your algorithm correctly computes LIPS.

Solution: Consider any parenthesis expression and let () be an immediately paired parenthesis in the result. Let i and j be the positions of the parenthesis in the original sequence. Note that $i < j$. Let k be the leftmost RPAREN and note that $i < k \leq j$ and the parenthesis at $k-1$ and k are immediately paired. In other words, there exists one immediately paired parentheses in the contiguous subsequence defined by i and j, e.g., "(....()....)", "(....()", "()...)". It thus suffices to count the immediately paired parenthesis in the input.

Question 4: SkylineLab Reloaded (20 points)

Given a sequence s of integers, an element at position i is a *local minimum* if $s_i < s_{i-1}$ and $s_i < s_{i+1}$. Similarly, an element at position i is a *local maximum* if $s_i > s_{i-1}$ and $s_i > s_{i+1}$. The first and last elements are local minima or maxima if they are less or greater than their only neighbor (respectively).

For example, in the sequence, the local maxima are marked with a plus and local minima are marked with a minus sign $\langle 5^+, 4, 3^-, 4, 5, 6, 7^+, 6, 5, 2^-, 3, 4^+ \rangle$.

For this question, assume no two adjacent elements are equal.

(a) (10 points) **Local minima and maxima.**

Write a function `extrema` that computes local minima and maxima of a sequence of integers in $O(n)$ work and $O(1)$ span. `extrema` should map an element k to `SOME k` if k is a local minimum or maximum, and to `NONE` otherwise. For example, `extrema` $\langle 5^+, 4, 3^-, 4, 5, 6, 7^+, 6, 5, 2^-, 3, 4^+ \rangle$ should evaluate to

$\langle$ *SOME(5), NONE, SOME(3), NONE, NONE, NONE,*
SOME(7), NONE, NONE, SOME(2), NONE, SOME(4)$\rangle$.

Complete the following implementation of `extrema`.

```
fun extrema (s : int seq)  =  (* Work = O(n), Span = O(1) *)
    let
      fun cmp f (i,j) =
          (j < 0) orelse (j > length s - 1)   (* first/last true *)
          orelse f (nth s i, nth s j)

      fun isLocalMin i = (cmp Int.< (i, i-1)) andalso (cmp Int.< (i, i+1))
      fun isLocalMax i = (cmp Int.> (i, i-1)) andalso (cmp Int.> (i, i+1))
    in
      (* Your solution here *)
```

Solution:

```
tabulate (fn i => if (localMin i) orelse (localMax i) then
                        (SOME (Seq.nth s i))
                      else
                        NONE) (length s)
```

```
    end
```

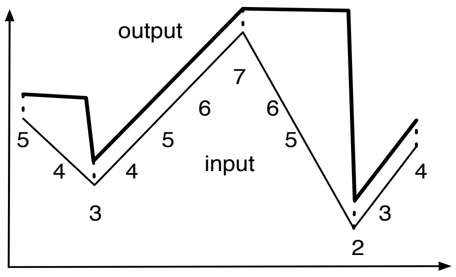

Figure 1: An example. The output is offset slightly for readability.

(b) (10 points) **Sentimental Markets.** In a sequence, the *trail-max* of a number v at a position is the largest number that is at the end of a monotonically increasing sequence of numbers starting (inclusively) at v and continuing to the **left**.

Write a function that maps each element v either 1) to v if v is a local mimima or maxima 2) to the trail-max of v otherwise. Your function should have $O(n)$ work and $O(\lg n)$ span. The figure above illustrates the following example:

S:	$\langle\, 5^+,$	4,	$3^-,$	4,	5,	6,	$7^+,$	6,	5,	$2^-,$	3,	$4^+\rangle$
Output:	$\langle\, 5,$	5,	3,	4,	5,	6,	7,	7,	7,	2,	3,	$4\ \rangle$

This function can model an optimistic trader that only corrects himself or herself at local minima (bust).

```
fun boomBust (s : int seq) = (* Work = O(n), Span = O(lg n) *)
```

Solution:

```
let
  fun copy(a, b) =
      case b of
        SOME _ => b
      | NONE => a

  fun pickMax (v, vo) =
      case vo of NONE => v
               | SOME v' => Int.max (v,v')

  val e = extrema s
  val (ec, _) = Seq.scani copy NONE e
in
  Seq.map2 pickMax s ec
end
```

Question 5: Higher Order Costs (20 points)

For full credit, show your work.

(a) (10 points) Give closed forms in terms of Θ for the work and span of the function f assuming the sequence s contains n sequences of m elements each and b contains m elements.

```
fun zipPlus (s1: int seq, s2: int seq) = map2 op+ s1 s2
fun f (b : int seq) (s : int seq seq) = reduce zipPlus b s
```

$W_f(n, m) =$

Solution: The work for `zipPlus` is linear in the number of elements. The work at the leaves of the reduction tree is $\frac{n}{2}\Theta(m)$. At each level the work decreases by a factor of 2. Thus, $W_f(n, m) = \Theta(nm)$. Alternatively, using a divide and conquer view of reduce $W_f(n, m) = 2W_f(n/2, m) + \Theta(m) = \Theta(nm)$, since the work is leaf dominated with $n/2$ pairs of leaves, each with $\Theta(m)$ work.

$S_f(n, m) =$

Solution: The span for `zipPlus` is $\Theta(1)$. Since the reduction tree has depth $\lg n$, $S_f(n, m) = \Theta(\lg n)$

(b) (10 points) Give closed forms in terms of Θ for the work and span of the following function g assuming the sequence s contains n sets of m elements each. Assume that all elements are unique accoss all sets and that element comparison is $O(1)$.

```
fun g (s : Set.set seq) = reduce Set.union (Set.empty ()) s
```

$W_g(n, m) =$

Solution: Since each union is with sets of the same size, the work is linear in the number of elements. The size of the union set can double in size and the number of applications of union halves at each level, the total work at each level is $\Theta(nm)$. Again, there are $\lg n$ levels. Thus $W_g(n, m) = \Theta(nm \lg n)$.

$S_g(n, m) =$

Solution: The span of union is the logarithm of the number of elements. Since the number of elements doubles each level above the leaves and for $\lg n$ levels, the total span $\sum_{i=1}^{\lg n} \lg(2^i m)$, which yields $S_g(n, m) = \Theta(\lg n \lg(nm))$.

Question 6: Dynamic Shortest Paths (35 points)

In class, you have learned about how to compute shortest paths on **unweighted, directed** graphs that do not change over time. However, graphs in the real world often change as the objects that they model change naturally. In this question, you will update single-source shortest path lengths as new edges are inserted (between existing vertices) into the graph.

First, you will define a data structure (of type `sssp`). The function `preProcess`, given a source vertex s, will initialize `sssp`. The `sssp` data structure can then be used to answer the shortest path length from the source to any vertex v in the graph efficiently.

You will then implement a function called `insertEdge` that will update the query structure `sssp` when an edge is inserted. When implementing `insertEdge` you will also find the `sssp` useful: It will help you to avoid doing redundant work when computing the new shortest paths.

For bounds assume that the graph has n vertices, m edges, and that its diameter is d. The signature for dynamic shortest paths on unweighted graphs is given below.

```
signature DSP =
sig
  structure Table : TABLE
  structure Set : SET = Table.Set

  type vertex = Table.Key.t
  type edge = vertex * vertex

  (* vertex set tables *)
  type graph = Set.set Table.table
  type sssp

  val preProcess: graph * vertex -> sssp
  val query: sssp -> vertex -> int
  val insertEdge : graph * sssp * edge -> graph * sssp
end
```

(a) (6 points) Describe (i) a data structure for `sssp`, (ii) the type of your `sssp`, and (iii) how it can be used to answer a shortest path length query in $O(\lg n)$ work and span. You can use the types defined in the signature.

Solution: i) A table mapping vertices to their shortest distances.
ii) type sssp = int Table.table
iii) A query is a simple table lookup.

(b) (5 points) Describe how you would implement `preProcess` in no more than two sentences.

Solution: By performing a BFS with the given source to return a table with the shortest distances to each vertex.

(c) (4 points) Give tight bounds for the work and the span of your implementation of `preProcess` for a graph with n vertices and m edges?

$W =$

$S =$

Solution: $W = O(m \lg n)$
$S = O(D \lg^2 n)$ where D is the diameter of the graph.

(d) (15 points) Describe your implementation of `insertEdge` given an edge (u, v). Your implementation should update the graph and update the query data structure `sssp` — avoid performing redundant work as much as possible.

- **Update the graph; describe in no more than 2 sentences:**

- **Update the query structure; describe in no more than 6 short sentences:**

Solution:

- `Update graph:` Find the vertex u in the table representing the graph, insert v into its neighbors set, and reinsert u and its updated neighbors into the table, replacing the old value.
- `Update the query structure:` Perform a BFS starting with u on the frontier and a starting level that is equal to the distance of u from the source (available in `sssp`). Remove from the frontier any vertex w whose distance stored in the `sssp` is no more than the current level. At this point, if the frontier is empty, return the updated `sssp`. Otherwise, for each remaining vertex in the frontier, update its distance stored in `sssp` with the current level. Recursively apply BFS to the neighbors of the frontier with the level incremented by 1.

(e) (5 points) **Analysis:** Give the work and span of `insertEdge`.

- **Worst Case:**

  $W =$

  $S =$

- **Best Case:**

  $W =$

  $S =$

- **"Expected" Case:** In general, when do you expect your algorithm to be more efficient re-computing all shortest paths?

Solution:

- **Worst Case:** Same as that of `preProcess`.
- **Best Case:**
  $W = O(\lg n)$
  $S = O(\lg n)$
- **"Expected" Case:** When the inserted edge is not on a relatively small number of paths.

Scratch Work:

Appendix: Library Functions

```
signature SEQUENCE =
sig
  type 'a seq
  type 'a ord = 'a * 'a -> order
  datatype 'a listview = NIL | CONS of 'a * 'a seq
  datatype 'a treeview = EMPTY | ELT of 'a | NODE of 'a seq * 'a seq

  exception Range
  exception Size

  val nth : 'a seq -> int -> 'a
  val length : 'a seq -> int
  val toList : 'a seq -> 'a list
  val toString : ('a -> string) -> 'a seq -> string
  val equal : ('a * 'a -> bool) -> 'a seq * 'a seq -> bool

  val empty : unit -> 'a seq
  val singleton : 'a -> 'a seq
  val tabulate : (int -> 'a) -> int -> 'a seq
  val fromList : 'a list -> 'a seq

  val rev : 'a seq -> 'a seq
  val append : 'a seq * 'a seq -> 'a seq
  val flatten : 'a seq seq -> 'a seq

  val filter : ('a -> bool) -> 'a seq -> 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val map2 : ('a * 'b -> 'c) -> 'a seq -> 'b seq -> 'c seq
  val zip : 'a seq -> 'b seq -> ('a * 'b) seq

  val enum : 'a seq -> (int * 'a) seq
  val inject : (int * 'a) seq -> 'a seq -> 'a seq

  val subseq : 'a seq -> int * int -> 'a seq
  val take : 'a seq * int -> 'a seq
  val drop : 'a seq * int -> 'a seq
  val showl : 'a seq -> 'a listview
  val showt : 'a seq -> 'a treeview

  val iter : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b
  val iterh : ('b * 'a -> 'b) -> 'b -> 'a seq -> 'b seq * 'b
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val scan : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a seq * 'a
  val scani : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a seq

  val sort : 'a ord -> 'a seq -> 'a seq
  val merge : 'a ord -> 'a seq -> 'a seq -> 'a seq
  val collect : 'a ord -> ('a * 'b) seq -> ('a * 'b seq) seq
  val collate : 'a ord -> 'a seq ord

end
```

ArraySequence	Work	Span

`empty ()`
`singleton a`
`length s`
`nth s i`
`showt s`
if $\|s\| = n$
	$O(1)$	$O(1)$

`tabulate f n`
if `f i` has W_i work and S_i span
`map f s`
if `f` s_i has W_i work and S_i span, and $\|s\| = n$
`map2 f s t`
if `f` (s_i, t_i) has W_i work and S_i span, and $\|s\| = n$
	$O\left(\sum_{i=0}^{n-1} W_i\right)$	$O\left(\max_{i=0}^{n-1} S_i\right)$

`reduce f b s`
if `f` does constant work and $\|s\| = n$
`scan f b s`
if `f` does constant work and $\|s\| = n$
`filter p s`
if `p` does constant work and $\|s\| = n$
	$O(n)$	$O(\lg n)$

`sort cmp s`
if `cmp` does constant work and $\|s\| = n$
`collect cmp s`
if `cmp` does constant work and $\|s\| = n$
	$O(n \lg n)$	$O(\lg^2 n)$

`merge cmp (s,t)`
if `cmp` does constant work, $\|s\| = n$, and $\|t\| = m$
`flatten s`
if if $s = \langle s_1, s_2, \ldots, s_k \rangle$ and $m + n = \sum_i \|s_i\|$
	$O(m + n)$	$O(\lg(m + n))$

`append (s,t)`
if $\|s\| = n$, and $\|t\| = m$
	$O(m + n)$	$O(1)$

Table/Set Operations	*Work*	*Span*
`size`(T) `singleton`(k,v)	$O(1)$	$O(1)$
`filter` f T	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\lg\lvert T\rvert + \max_{(k,v)\in T} S(f(v))\Big)$
`map` f T	$O\Big(\sum_{(k,v)\in T} W(f(v))\Big)$	$O\Big(\lg\lvert T\rvert + \max_{(k,v)\in T} S(f(v))\Big)$
`tabulate` f S	$O\Big(\sum_{k\in S} W(f(k))\Big)$	$O\Big(\max_{k\in S} S(f(k))\Big)$
`find` T k `insert` f (k,v) T `delete` k T	$O(\lg\lvert T\rvert)$	$O(\lg\lvert T\rvert)$
`extract` (T_1,T_2) `merge` f (T_1,T_2) `erase` (T_1,T_2)	$O\big(m\lg(1+\frac{n}{m})\big)$	$O\big(\lg(n+m)\big)$
`domain` T `range` T `toSeq` T	$O(\lvert T\rvert)$	$O(\lg\lvert T\rvert)$
`collect` S `fromSeq` S	$O(\lvert S\rvert\lg\lvert S\rvert)$	$O(\lg^2\lvert S\rvert)$
`intersection` (S_1,S_2) `union` (S_1,S_2) `difference` (S_1,S_2)	$O\big(m\lg(1+\frac{n}{m})\big)$	$O\big(\lg(n+m)\big)$

where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$. For `reduce` you can assume the cost is the same as `Seq.reduce f init (range(T))`. In particular `Seq.reduce` defines a balanced tree over the sequence, and `Table.reduce` will also use a balanced tree. For `merge` and `insert` the bounds assume the merging function has constant work.