due: *tue, feb 7 @ 11:59pm*

1 Introduction

In this homework, you will solve and analyze two problems (one about geometry and the other about sequences) to explore divide-and-conquer style algorithms. Then, you will do some exercises involving recurrences. You will likely find this assignment more conceptually challenging than the first assignment. ***Start now!***

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.

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

`closestpair.sml` and `countswaps.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`.

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, in your solution to the closest-pair problem, you should comment on how the lemma we ask you to prove is used). 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. 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 describe your algorithm in sufficient detail that your algorithm maps straightforwardly onto your code). As an example, here is a proof of the algorithm above:

Lemma 3.1. *If* `match' s = (a,b)` *then s can be reduced to* $)^a(^b$ *by deleting the substring "()" repeatedly*

Proof. Proceed by induction on $|s|$. The base cases are trivial. Otherwise, $S = LR$. Compute `(i,j) = match' L` and `(k,l) = match' R`. By IH, L can be reduced to $)^i(^j$ and R can be reduced to $)^k(^l$. So S can be reduced to $)^i(^j)^k(^l$, and we delete $|k - j|$ matched pairs in the center, to get an answer of the desired form. □

Lemma 3.2. `s` *is well-formed* $\iff$ `match' s = (0,0)`

Proof. If `match' s = (0,0)`, then by the lemma above, `s` can be reduced to the empty string by deleting "()" repeatedly. The empty string is well-formed, so by Observation 3.1 from Recitation 1, `s` is well-formed.

If `match' s <> (0,0)`, then `s` can be reduced to a string with either opening close parenthesis or trailing open parentheses. Obviously these reduced forms are not well-formed. By observation 3.1, `s` was not well-formed either. □

4 Closest Pair

Suppose we are given a set of n two-dimensional points. How fast can we find the distance between the closest pair of points? The most basic algorithm has to examine all $\binom{n}{2}$ pairs and find the minimum, but, perhaps, we don't have to look at every pair of points because these are points in a 2-d plane, which has a lot of structure.

For inspiration, the problem in one dimension can be solved in $O(n \log n)$ work and $O(\log n)$ span via sorting (Do you see how?). Alternatively, there is an equally efficient divide-and-conquer algorithm, which works roughly as follows. On input a sequence S of n numbers,

1. Find a median[1] m and split S into $S_1 = \{x \in S : x < m\}$ and $S_2 = \{x :\in S : x \geq m\}$.
2. Recursively find the closest pair distance δ_i on S_i, for $i = 1, 2$; and
3. Derive the closest pair distance on S by taking the smaller of $\min(\delta_1, \delta_2)$ and the distance between the rightmost point of S_1 and the leftmost point of S_2.

[1] This can be computed directly in $O(n)$ work and $O(\log^2 n)$ span using divide and conquer; however, we won't sketch this solution here. Instead, we will presort the sequence S in $O(n \log n)$ work and $O(\log^2 n)$ span, so median finding takes constant work and span.

In this problem, you will design and analyze a work-optimal divide-and-conquer solution to the closest pair problem.

Problem 4.1 (The Closest Pair Problem)**.** Given a sequence of points $S = \langle p_1, \ldots, p_n \rangle$, where $p_i = (x_i, y_i)$, the *closest pair problem* is to report the distance between the closest pair of points, that is,

$$\min\{d(p_i, p_j) : p_i, p_j \in S \wedge i \neq j\},$$

where $d(\cdot, \cdot)$ is the Euclidean distance measure.

Remember that the distance $d(p_i, p_j)$ between $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$ is given by

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

4.1 Logistics

The only file you're handing in for this problem is `closestpair.sml`. You will implement a function `closestPair: point seq -> real`. inside `helper-modules.sml`. The type `point` is defined in the structure `Point2D`. Your program will return `Real.posInf` if $|S| < 2$ and return δ if the closest pair distance is δ. Your implementation must be inside `closestpair.sml` in a functor called `ClosestPair` that ascribes to the signature `CLOSEST_PAIR`. Your `ClosestPair` functor should take exactly one argument structure ascribing to `CP_PACKAGE`.

4.2 Detailed Specification

The algorithm you design will follow the same outline as the divide-and-conquer algorithm for one dimension. That is to say, it involves the following steps: (1) **divide** that splits the input sequence in approximately half (according to some criteria); (2) **recurse** that recursively solves the two halves of the problem; (3) **combine** that finds the solution to your input, presumably with help of the recursive solutions you have found.

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

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

where $W_{combine}(n)$ denotes the work needed in the combine step. In your analysis, you'll assume that the sequence implementation is the `ArraySequence` implementation. Section 7 shows cost specifications of `ArraySequence` functions.

Task 4.1 (5%).

As a warm-up, give a proof to the following lemma:[2]

Lemma 4.2. *Let $\delta > 0$. There is a constant C_P, independent of δ, such that if R is a $\delta \times \delta$ square and the closest pair distance in R is at least δ, then R contains at most C_P points.*

[2] *Hint:* Pigeonhole principle. Or, how many circles of radius $\frac{r}{2}$ can you pack into an $r \times r$ square such that none of the circles overlap?

Task 4.2 (25%). Implement the algorithm `closestPair` in a functor called `ClosestPair` in the file `closestpair.sml`. Your functor will take a "package" structure that ascribes to the signature `CP_PACKAGE`. Make sure that your functor ascribes to the signature `CLOSEST_PAIR`. The starter code you pull from our Git handout system already contains the file `closestpair.sml`, which has some starter code.

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 points. To achieve this, you may need a preprocessing step that runs before the divide-recurse-combine main body. Carefully explain why your divide-and-conquer steps satisfy (1) and give a closed-form solution to the recurrence. (Later in the semester, you will be able to show that this $O(n \log n)$ work bound is the best possible.)

Significant partial credit will be given to $O(n \log^2 n)$-work $O(\log^2 n)$-span solutions.

Task 4.3 (10%).

Prove the correctness of your divide-and-conquer solution by induction. Be sure to carefully state the theorem that you're proving and to note all the 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. $P(\texttt{hidet EMPTY})$
> 2. $\forall \texttt{x} : \alpha . P(\texttt{hidet (ELT x)})$
> 3. $\forall \texttt{s1} : \alpha\ \texttt{seq}.\ \forall \texttt{s2} : \alpha\ \texttt{seq}.\ (P(s_1) \wedge P(s_2)) \implies P(\texttt{hidet (NODE(s1, s2))})$

5 How costly is insertion sort?

The insertion sort algorithm from 15-122 should be familiar. Here is a standard implementation of insertion sort in a C-like language (for this problem, we will think of the input S as an `int seq`).

```
void insertion_sort(int[] S) {
    for(int i=0; i<S.length; i++) {
        int j = i;
        while(j > 0 && S[j] < S[j-1]) {
            swap(S,j,j-1);
            j = j-1;
        }
    }
}
```

```
void swap(int[] S, int i, int j) {
    int T = S[i];
    S[i] = S[j];
    S[j] = T;
}
```

We want you to perform a careful analysis of the above algorithm. In particular, for a given input S, how many times will the `swap` function be called?

One way to compute this quantity would be to modify the swap function to increment a counter every time it is called, run `insertion_sort()` on the input sequence, and read off the answer. Unfortunately, on a list sorted in descending order, this approach is $O(|S|^2)$ work. But we can do better.

First, we need to get a handle on the problem: it would be better if we had a characterization of the cost in terms of a property of the sequence S, rather than the insertion sort algorithm.

Task 5.1 (5%). Prove the following lemma:

Lemma 5.1. *Let* $I(S) = \{(i,j) : 0 \leq i < j < |S| \text{ and } S_i > S_j\}$. *Then, the number of times the function* `swap` *is called during the execution of* `insertion_sort(S)` *is* $|I(S)|$.

Using this characterization, we can define a formal problem as follows:

Problem 5.2 (The Swap Counting Problem)**.** Given a integer sequence S of length n which is a permutation of $\{1,\ldots,n\}$, the *swap counting problem* is to report the size of the set $I(S)$, where

$$I(S) \;=\; \{(i,j) : 0 \leq i < j < |S| \text{ and } S_i > S_j\}. \tag{2}$$

$I(\langle 1,3,2\rangle)$ has size 1, $I(\langle 3,2,1\rangle)$ has size 3, and $I(\langle 1,2,3,4,5\rangle)$ has size 0.

5.1 Detailed Specification

You will write the function

```
count_swaps:  int seq -> int
```

which solves the swap counting problem (i.e., `count_swaps S` computes $|I(S)|$). You are encouraged to take advantage of the library functions in our sequence implementation.

You will hand in the file `countswaps.sml`, which should contain a function `count_swaps: int seq -> int`. You solution will be a functor name `CountSwaps` ascribing to the signature `COUNTSWAPS` defined in `COUNTSWAPS.sig`. Your functor will take as a parameter a structure ascribing the signature `SEQUENCE`.

You will implement a low-span, divide-and-conquer solution to this problem. To simplify your solution, you may assume that S is a permutation of $1,\ldots,|S|$ (that is, each number from 1 to $|S|$ appears exactly once in S).

You will use the `ArraySequence` implementation of `SEQUENCE` and the cost specifications given in Section 7. For full credit, the overall work of your solution must be $O(|S|\log|S|)$, and the overall span of your solution must be $O(\log^2|S|)$.

Task 5.2 (30%). Implement a divide and conquer solution to the Count Swaps problem. The starter code you will pull from the Git handout system contains the file `countswaps.sml` in which to put your `CountSwaps` functor.

Task 5.3 (5%). Explain your algorithm and prove that it is correct.

Task 5.4 (3%). Argue that your algorithm runs in $O(|S| \log |S|)$ work.

Task 5.5 (3%). Argue that your algorithm runs in $O(\log^2 |S|)$ span.

6 Recurrences

Task 6.1 (14%). In this problem, you will determine the complexity of the following recurrences. Please give tight Θ-bounds. Recall that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $g \in O(f)$.

1. $f(n) = f(\sqrt{n}) + \Theta(\log n)$.
2. $f(n) = 4f(n/4) + \Theta(\sqrt{n})$
3. $f(n) = \sqrt{n} f(\sqrt{n}) + \Theta(\sqrt{n} \log n)$.

Please justify your steps and be sure to argue why your bound is tight.

7 Cost Specifications

ArraySequence	Work	Span

`empty ()`
`singleton a`
`length s`
`nth s i`
	$O(1)$	$O(1)$

`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 $\|s\| = n$
`showt s`
if $\|s\| = n$
`hidet tv`
if the combined length of the sequences is n
	$O(n)$	$O(\log n)$

`sort cmp s` *if* `cmp` does constant work and $\|s\| = n$	$O(n \log n)$	$O(\log^2 n)$
`merge cmp s t` *if* `cmp` does constant work, $\|s\| = n$, and $\|t\| = m$	$O(m+n)$	$O(\log(m+n))$