Homework 9

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

tag — Version 1.5 **due:** *wed, may 2 @ 11:59pm*

Disclaimer:
We will not grade non-compiling code.

1 Introduction

This is the last homework this semester! You will do exercises involving dynamic programming.

1.1 Submission

This assignment is distributed in a number of files in our `git` repository. There are both code and written portions.

Submit your solutions by placing you solution files in your handin directory, located at

```
/afs/andrew.cmu.edu/course/15/210/handin/<yourandrewid>/assn9/
```

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

```
/afs/andrew.cmu.edu/course/15/210/bin/check/09/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 `hw09.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, you're handing in a total of 2 files:

```
TreeIS.sml    TestTreeIS.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 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.

2 Dynamic Programming

Here is a model solution to the "breaking a string into palindromes" problem from last recitation.

Throughout this assignment, we write $S[a..b]$ to denote the substring of S between position i and position j. For example, if $S =$ `"ABC"`, then $S[1..2]$ is the string `"BC"`.

Question: Given a string S of length n, describe an $O(n^2)$ time dynamic programming problem to compute the minimum number of palindromes S can be broken into.

Answer: The subproblems are:
`DP[i]` is the minimum number of palindromes the last n characters of S
`IS_PAL[i,j]` is true if $S[i..j]$ is a palindrome

The recurrences are:

$$\texttt{DP}[i] = 1 + \min_{j<i \text{ and } \texttt{IS\_PAL}[j+1,i]} \texttt{DP}[j]$$
$$\texttt{IS\_PAL}[i,j] = S[i] == S[j] \text{ andalso } \texttt{IS\_PAL}[i+1,j-1]$$

The base case is `DP[0] = 0` and `IS_PAL[i][j] = true` if $i+1 \leq j$
The final answer is `DP`$[n]$

In calculating `DP`, there are n subproblems, each of which has $O(n)$ non-recursive work (looping over $O(n)$ choices of j doing constant non-recursive work for each j), so the amount of work in calculating `DP` this step is $O(n^2)$.
In calculating `IS_PAL`, there are $O(n^2)$ subproblems, each of which has $O(1)$ non-recursive work, so the total work in calculating `IS_PAL` is $O(n^2)$.
So the total amount of work is $O(n^2)$.

Writeups should include: The subproblems you are working with, how to get the final answer, any base cases, and recurrences for computing subproblems in terms of other subproblems—and runtime analysis.

Writeups should *not* include: Correctness proofs.

Writeups should be short (excessively long or unclear answers will lose points even if they are correct).

2.1 Independent Sets in Trees

This is the only coding question in this assignment. You have a rooted tree on n vertices, where each vertex has an integer weight. Your goal is to choose an independent set A of vertices such that the sum of the weights of the vertices in your set is maximized. (Recall that A is an independent set if no two vertices in A are connected by an edge).

Details: We represent a tree with the following data type:

```
datatype tree = Node of vertex * weight * tree seq
```

where both `vertex` and `weight` are integers. Each node is a tuple containing the vertex label, weight, and a sequence of children.

You will implement the function `bestSet: tree * int -> vertex seq`, where `bestSet (t, n)` returns a sequence listing vertices in the independent set that maximizes the sum of the weights. If there are multiple answers, return any of them. Your answer does not need to be ordered in any particular way. There will be n vertices labeled from 0 to $n-1$.

Assumptions: Vertex weights may be negative. There may be duplicate weights. No subset of vertices will have a sum-of-weights overflowing an `int`. You may *not* assume anything about the structure of the tree. The empty set has a sum of 0.

Before you start coding, keep in mind that you are to produce an actual set of vertices, not just the best achievable sum-of-weights. However, you should start with computing the best possible sum, and then figure out how to reconstruct the set with this information.

Task 2.1 (15%). Describe an $O(n)$ work dynamic programming solution to this problem.

Task 2.2 (20%). Implement `bestSet` in `TreeIS.sml`

Task 2.3 (5%). Write tests for `bestSet` in `TestTreeIS.sml`

2.2 Tiling a Grid

How many ways are there to tile a $4 \times n$ *grid with* 2×1 *dominos?* Dominos are not distinguishable. Rotations and reflections of a tiling count as separate tilings. Formally, we say that two tilings are *different* if some pair of squares is covered by a single domino in one tiling but not the other.

A TA confused this problem with the $2 \times n$ variant, which is trivial. Don't make the same mistake!

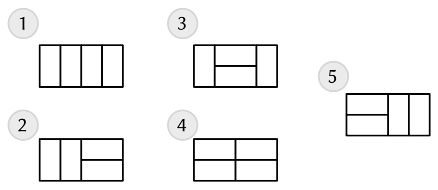

Figure 1: All possible tilings for $n = 2$.

Task 2.4 (15%). Describe an $O(n)$ work dynamic programming solution. (Hint: 4 is small. What are your states?)

2.3 Building a String

You have a set of $k \geq 1$ alphabets $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ (k is not a constant). You have m rules of the form

> "*replace a single character x with yz*" (which we write $x \to yz$)

where $x, y, z \in \Sigma$ are characters from the alphabet set. At each timestep, for each character in your string thus far, you can either leave it alone or apply a rule to it. Several rules can be applied in the same timestep, and each rule can be applied multiple times.

For example, suppose your alphabet were $\Sigma = \{\sigma_1 = A, \sigma_2 = B, \sigma_3 = C\}$ and your rules were:

- *Rule 1*: $A \to BC$;
- *Rule 2*: $B \to AC$; and
- *Rule 3*: $C \to AB$

This means that if we have the string *ACC*, we could apply Rule 3 on both occurrences of C in the same round, yielding *AABAB*. We could also get *BCABAB* by applying Rule 1 on A and Rule 3 on the two occurrences of C.

Now consider deriving the string *ACAB* from A. One possibility is to apply Rule 1 to get $A \to BC$—then, apply Rule 2 on B and Rule 3 on C to get *ACAB*. Alternatively, we could start with A and use Rule 1 to get *BC*; then, apply Rule 2 on B and leave C alone, so we get *ACC*. After that, apply Rule 3 on the last character gives us *ACAB*. In this case, the first path is shorter, hence a better path.

Task 2.5 (20%). Describe an $O(n^3mk)$ work, dynamic programming solution to compute the *minimum* number of steps it takes to make a string S of length n from the string "σ_1". If you cannot make S from "σ_1" using the provided rules, your algorithm should detect this. For concreteness, the alphabets are numbers 1 through k, the rules are given as a sequence of triples (the rule $x \to yz$ is represented as (x, y, z)), and the "target" string S is given as an `int seq`.

2.4 Slicing up an Array

You have an array A of non-negative integers. You will make k cuts between adjacent elements of the array; once you are finished making your cuts, you will be left with several "segments" between cuts. The score of these cuts is the sum of the scores of the segments; the score of the segment which starts at position i and ends at position j is

$$\sum_{k=i}^{j} \sum_{\ell=k+1}^{j} A_k A_\ell$$

What is the minimum achievable score?

For example, if the array is $4, 5, 1, 2$ and you have 1 cut, you could cut between 5 and 1 to get the segments $[4, 5], [1, 2]$ for a total score of $4 * 5 + 1 * 2 = 20$. But it is better to cut between 4 and 5 to get the segments $[4], [5, 1, 2]$ for a total score of $5 * 1 + 5 * 2 + 1 * 2 = 17$ (convention: a segment of length 1 has score 0).

Task 2.6 (25%). Describe an $O(n^2k)$-work dynamic programming solution. You will receive partial credit for giving a $O(n^3k)$-work algorithm.