15-451 Algorithms, Fall 2010

Homework # 3 **Due: October 5, 2010**

Please hand in each problem on a separate sheet and put your **name** your **andrew id** and your **recitation** (time or letter) at the top of each page. You will be handing each problem into a separate box in lecture, and we will then give homeworks back in recitation.
Remember: Homework is due at the begining of class. If you hand in your homework after class has started you are late. Please cite any sources you use (google, course notes, etc) as well any collaborators. As always each solution must be written in your own words. You must fully understand any solution you write down. You are required to typeset your homework.

Problems:

(25 pts) 1. **Potentially Hopping Pennies.** You are given an infinite 2 dimensional grid G. If there is a penny on cell (i, j) then we write $G[i][j] = 1$, otherwise $G[i][j] = 0$ (Notice that since the grid is infinite i and/or j may be negative!) Bill Gates places and infinite number of pennies on the grid such that initially $G[i][j] = 1$ whenever $j \leq 0$. Pennies move by jumping (horizontally or vertically) over one other penny onto an empty square. When this happens the other penny is removed. Formally, you are allowed to make four types of moves:

(a) Horizontal Jump Right: If $G[i][j] = G[i+1][j] = 1$ and $G[i+2][j] = 0$ then we the penny at (i, j) can jump the penny at $(i+1, j)$ and move to $(i+2, j)$, when we do this we also remove the penny at $(i+1, j)$. The final configuration is $G[i][j] = G[i+1][j] = 0$ and $G[i+2][j] = 1$

(b) Horizontal Jump Left: If $G[i][j] = G[i-1][j] = 1$ and $G[i-2][j] = 0$ then the penny at (i, j) can jump the penny at $(i-1, j)$ and move to $(i-2, j)$, when we do this we also remove the penny at $(i-1, j)$. The final configuration is $G[i][j] = G[i-1][j] = 0$ and $G[i-2][j] = 1$

(c) Vertical Jump Up: If $G[i][j] = G[i][j+1] = 1$ and $G[i][j+2] = 0$ then the penny at (i, j) can jump the penny at $(i, j+1)$ and move to $(i, j+2)$, when we do this we also remove the penny at $(i, j+1)$. The final configuration is $G[i][j] = G[i][j+1] = 0$ and $G[i][j+2] = 1$

(d) Vertical Jump Down: If $G[i][j] = G[i][j-1] = 1$ and $G[i][j-2] = 0$ then the penny at (i, j) can jump the penny at $(i, j-1)$ and move to $(i, j-2)$, when we do this we also remove the penny at $(i-1, j)$. The final configuration is $G[i][j] = G[i][j-1] = 0$ and $G[i][j-2] = 1$

These moves are perhaps most easily understood with pictures:

```
(Vertical Jump Up)          (Vertical Jump Down)
.........       .........    .........       .........
...000...       ...010...    ...010...       ...000...
...010...  =>   ...000...    ...010...  =>   ...000...
...010...       ...000...    ...000...       ...010...
.........       .........    .........       .........
```

```
(Horizontal Jump Right)      (Horizontal Jump Left)
.........       .........       .........       .........
...000...       ...000...       ...000...       ...000...
...110...  =>   ...001...       ...011...  =>   ...100...
...000...       ...000...       ...000...       ...000...
.........       .........       .........       .........

Examples of Illegal Moves

(No Diagonal Jumps)         (Must Jump into Empty Spot)
.........       .........       .........       .........
...000...       ...100...       ...0000..       ...0000..
...010...  =>   ...000...       ...0111..  =>   ...0100..
...001...       ...000...       ...0000..       ...0000..
.........       .........       .........       .........
(Can't Jump Over Empty Square)    (Can't Jump Over Two Squares)
.........       .........           .........       .........
...000...       ...010...           ...0000..       ...0000..
...000...  =>   ...000...           ...0111..  =>   ...1000..
...010...       ...000...           ...0000..       ...0000..
.........       .........           .........       .........
```

If a penny is at cell (i, j) we say that j is the height of the penny. Your goal will be to provide upper and lower bound on the maximum height a penny can ever reach after a finite number of moves.

(a) (20 pts) Using potential functions provide an upper bound on the maximum height h that any penny can reach.

(b) (5 pts) Provide a matching lower bound (ie. a sequence of jumps resulting in a penny at some cell of height h). Hint: Once you have the right upper bound it should not be too hard to find a matching lower bound.

(25 pts) 2. **Doing more with search trees.** Often, by adding extra information to the nodes of a binary search tree, it is possible to perform other operations you might be interested in. In particular, suppose we want to be able to do the following operations efficiently:

- Given k, output the kth smallest element (let's assume all keys are distinct).
- Given a key x, do a version of lookup(x) that tells us the rank of x (how many keys are smaller than x).

Note that if we had a fixed set of data, this would be easy: we just make a sorted array. If we are asked to produce the kth smallest element, we just output $A[k]$. To get the rank of x, we just do binary search to find it, and then output which location it's in. How could we do this with binary search trees, if we want to do inserts too? (Obviously, all basic BST properties have to be perserved.) In particular, describe a piece of additional information you could store at each node of the binary search tree such that:

(1) it allows us to perform the above operations in time O(depth of tree).
(2) the values can be maintained efficiently (time proportional to the depth of the tree) when a new node is inserted.

For example, a *bad* way of solving the problem would be to have each node store the rank of its key. This is bad because if you insert a new key that is smaller than everything else, you may have to update *everyone's* rank. So we don't have property (2). For this problem you should:

(a) Describe the extra information you will store at each node of the tree.

(b) Describe how you can use this to find the kth smallest element efficiently, and how you can use this to find the rank of a given key efficiently.

(c) Describe how the information can be updated efficiently when a new node is inserted into the tree. (You can assume for this that we are doing simple binary search tree insertion, though if you like you can also describe how it is updated when rotations are performed).

(25 pts) 3. **Super Thief**

You are an amazing super thief. You have scoped out a circle of n, houses that you would like to rob tonight. However, as the super thief that you are, you've done your homework and realized that you want to avoid robbing two houses that are adjacent to one another because this will minimize the chance that you get caught. You've also scouted out the hood, so for every house i, you know that the net worth you will gain from robbing that house is v_i.

You want to try to maximize the amount of money that you can rob. Remembering your former life as a student at Carnegie Mellon taking 15-451, you decide to develop an algorithm that will output the maximum amount of money you can steal.

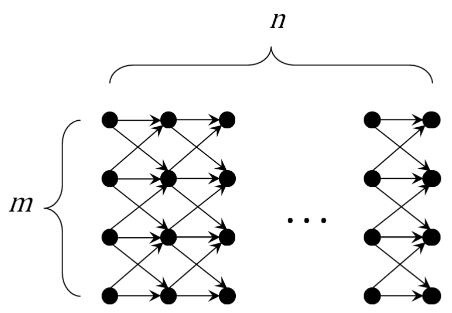
(a) (3 pts) Because you were a great student in 15451, you suspect that this problem can be solved with dynamic programming. Explain why this problem is a good candidate for a DP algorithm.

(b) (15 pts) Give a dynamic programming algorithm that to find the maximum amount of money that you can steal given that you never rob two houses that are adjacent and state whether your are using the bottom up approach or the top down approach.

(c) (7 pts) Modify your algorithm so that it also returns the best set of houses to rob instead of the maximum value you can steal.

(25 pts) 4. **Let's start playing with Graphs!**

(a) (13 pts) Suppose you have a directed graph that has mn nodes laid out in an m (rows) $\times$ n (columns) grid (as shown below). There is a directed edge from node i to node j if $col(i) = col(j) - 1$ and either $row(i) = row(j) - 1$, $row(i) = row(j)$, or $row(i) = row(j) + 1$. Nodes on the top and bottom edges of the grid will have only two outgoing/incoming edges each. Nodes in the left-most column will have only outgoing edges and nodes in the right-most column will have only incoming edges. Every edge has a weight in $\mathbb{R}^+$ (real numbers greater than 0). Describe an optimal algorithm (the worst-case asymptotic lower bound of the problem should match the worst-case asymptotic upper bound of the algorithm) that finds the maximum product of all the edges along any path from any node in the left-most column to any node in the right-most column. Also, analyze the complexity (asymptotic analysis is sufficient) of your algorithm and prove that it is optimal (arguments on asymptotic complexities are sufficient).



(b) (5 pts) Now suppose that, in the above part, you were asked to find the minimum product (instead of the maximum). How would your algorithm change? How would this affect the runtime?

(c) (7 pts) Suppose you now have an *un*directed graph that has n nodes and has edge weights in $\mathbb{N}^+$ (the set of natural numbers, other than 0). Given a source node, s, and a target node, t, describe an efficient algorithm to find a path from s to t which has the minimum product of all the edges. (Hint: Look at the Bellman-Ford Algorithm in Section 12.4 of the lecture notes.)