Introduction to Computer Science
HW #4
Due: 2016/05/11

Homework Rules:

Hand-written homework can be handed in **before lecture starts**. Otherwise, you may contact the TA in advance and then bring the hardcopy to the TA in BL-603 (please send e-mail in advance).

As for the programming part, you need to upload it to CEIBA before the deadline. The file you upload must be a **.zip** file that contains the following files:

README.txt

HW01_b04901XXX (a folder that contains all .cpp & .h as required),

1. Do not submit executable files (.exe) or objective files (.o, .obj). Files with names in wrong format will not be graded. You must **remove any system calls**, such as system ("pause"), in your code if any.
2. In README.txt, you need to describe which compiler you used in this homework and how to compile it (if it is in a "project" form).
3. In your .cpp files, we suggest you write comments as detailed as you can. If your code does not work properly, code with comments earns you more partial credits.

Chapter 6 Review Problems (5% each):

21, 41, 42, 47, 58.

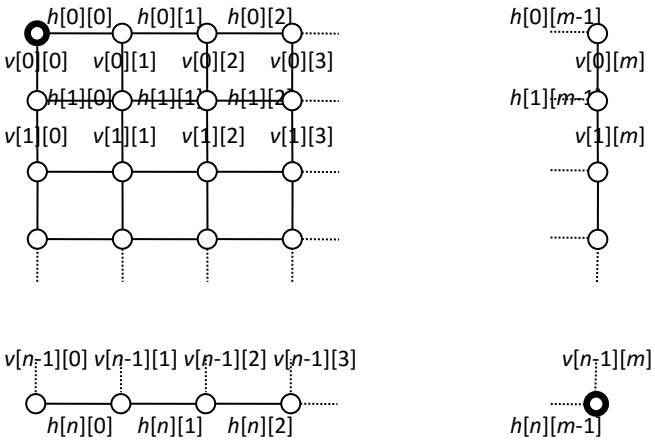
Programming Problem I (50%): Shortest Path

You may use either python or C/C++ for this homework;

however, you need to deal with the "read" by yourself.

Consider a graph of *m* by *n* grids:

Introduction to Computer Science
HW #4
Due: 2016/05/11



Use dynamic programming to find the shortest path from the upper-left node to the lower-right node with **right and down moves only**. Costs of vertical edges are stored in an 2D array *v*[0..*n*-1][0..*m*]; costs of horizontal edges are stored in another 2D array *h*[0..*n*][0..*m*-1]. These costs are **positive real-values**.

You can use readParameters() to read all parameters (*m*, *n*, *v*[][], and *h*[][]) from **input**. Remember to call release() when done. Check out **hw4.cpp** for more information.

Your program should print out 2 lines (on screen). The 1st line is the total cost of the shortest right-down path. The 2nd line is a string of (*m*+*n*) characters of 'v' or 'h', standing for vertical (down) or horizontal (right) respectively.

Programming Problem II (25%):

Write a **prolog** program to compute $\sqrt{a}$ by the following iteration:

$$x_m \leftarrow \frac{x_{m-1} \cdot x_{m-1} + a}{2 \cdot x_{m-1}}$$

You need to write the function (save your code in **"sroot.pl"**):

- ***sroot*(*X*,*M*,*SX*):** compute the square root (*SX*) of *X* up to *m* iterations. Initial guess is 1. In other words, sroot(*X*, 1, 1) is true for any *X*.

Some more examples are listed below:

sroot(2,1,X)	X=1
sroot(2,2,X)	X=1.5
sroot(2,3,X)	X=1.4166666666666667

sroot(2,4,X)	X=1.4142156862745099
sroot(2,5,X)	X=1.4142135623746899

Bonus (5%)

You are facing *n* monsters lining up. Thanks to your combat goggle; you can see the attack power of each monster (the damage that the monster is going to inflict on you). Your only weapon is a shotgun which takes down three nearby monsters with one shot. Your agility is roughly the same as monsters. Therefore, you shoot once, and then all remaining monsters attacks you. You need to find a way to take down all monster while takes only minimum damages.

Index	0	1	2	3	4	5	6	7
Attack Power	3	3	6	6	9	6	4	9

33***649 ➔ 25
33****** ➔ 6

If you shoot at the 3rd, then 6, 6, 9 are down. You then need to take 3+3+6+4+9=25 damages. If you then shoot at the 6th, then 6, 4, 9 are down. You take 3+3=6 damages then. You can shoot at 0th or 1st (same effect, but your program needs to give the minimum index), then all monsters are down. This is actually the best way. Your program should output two lines of numbers. The 1st line is the shooting sequence: "3 6 0", and the 2nd line is the total damages taken: "31". The input is a text file "input.txt" consisting of two lines: the 1st is n (8 in the above example), and the 2nd is monsters' attack powers, all in integers (3 3 6 6 9 6 4 9 in the above example).

Note that greedy does not always give the correct answer. For example, 10, 10, 10, 9, 10, 10, 10, 9, 10, 10, 10, 9, ….

HINT: In the optimum solution, the damages of your shots are non-increasing.

If you accomplished the bonus, save your code in **"bonus.cpp" (or bonus.py)** to hand in.