Project 2

1. Optimal binary search tree algorithm.​ Implement the optimal binary search tree algorithm discussed in class. (No credit will be given if any other algorithm is implemented.) Your program should take an input variable n > 0 followed by n float numbers that represent the probabilities for accessing the n keys (see example below).

For this problem, the c or cpp file name should be obst.c or obst.cpp. ​Your makefile

will generate “obst.out​”.

Your command to run the optimal binary search program should be:

./obst.out <value of n> <n probabilities>

Example: ./obst.out 4 0.1 0.2 0.3 0.4

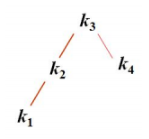
/\* In this example, you have four keys: 1, 2, 3, 4 and each of them has probability of

access 0.1, 0.2, 0.3, and 0.4, respectively. \*/

If your optimal binary search tree looks like the following image, your program

should display the result as follows (only textual output is required):

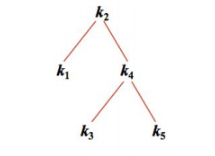
depth 0: 3, depth 1: 2, 4, ​depth 2​: 1, 0, 0, 0



As another example, if the optimal binary search is the following image, your program

should print:

depth 0: 2, depth 1: 1, 4, depth 2: 0, 0, 3, 5

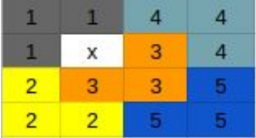


2. Implement the tromino tiling algorithm. Your program should take an input positive

integer k and the position of the hole as the Linux command line and generate a 2k\* 2k board.

For example if your input is 4 then your code should generate 16 x16 board. Show you result

as the following image:



Here, “digits” represent the regular tile cell and “X” represents a hole.

Make sure your program correctly implements the tromino tiling algorithm with O(n2)

time complexity (the divide and conquer algorithm described in class); that is, there must

be only one hole on the board and each of all the remaining 2k\* 2k-1 squares on the

board must be covered by exactly one square of one tromino tile. No credit will be given

if the algorithm is incorrectly implemented, the time complexity of your program is

higher than O(n2), or your program only works for specific k values.

There should be a function named trominoTile()​, which will be called from your main()

function. main() should be able to read the user input. To generate a board and print it,

you can create additional function calls. Make sure you put appropriate one or two line

comments to each of your function definitions. Inappropriate function declaration and

unnecessary function calls will cost your points.

For the tromino problem, the c or cpp file name should be tromino.c or tromino.cpp.

Your ​makefile should generate “tromino.out” file. Make sure your makefile will not

run/execute the file.

To run tromino program your command should be:

./tromino.out <value of k> <hole position row number> <hole position column number>

example: ./tromino.out 3 4 5

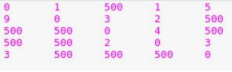
3. Implement Floyd’s algorithm to find the shortest path for each pair of node in the

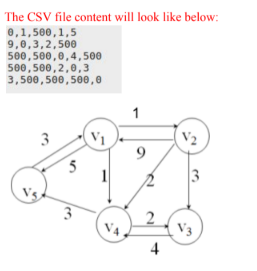
following graph. (Use 500 instead of infinity assuming that the weight of an edge is not bigger

than 499.) Print shortest paths for all pairs of nodes. Input adjacency matrix should be stored in a

csv file and will be passed to run your code. See the following for a sample adjacency matrix and

the corresponding graph.





Make sure your program correctly implements Floyd’s algorithm. Make sure you put appropriate

one or two line comments to each of your function definitions. Inappropriate function declaration

and unnecessary function calls will cost your points.

For the Floyd’s problem, the c or cpp file name should be floyd.c or floyd.cpp. ​Your ​makefile

should generate “floyd.out​” file. Make sure your makefile will not run/execute the file.

To run the floyd ​program, your command should be:

./floyd.out <input file name>