ASSIGNMENT 2

DFS LAB

Deadline :15^{th} September, 2017

P0. Make use of the code discussed in class to create a generic stack library supporting the standard operations. Note that you will NOT be permitted to implement stack operations while solving any other problem in this (or subsequent) assignments; instead, when needed, you must henceforth use the library functions that you implement while solving this problem.

P1. Create a similar library of functions for operations on generic queues.

P2. A *stack permutation* is a permutation of objects in the given input queue which is done by transferring elements from input queue to the output queue with the help of a stack and the built-in push and pop functions.

The well defined rules are:

- Only dequeue from the input queue.
- Use inbuilt push, pop functions in the single stack.
- Stack and input queue must be empty at the end.
- Only enqueue to the output queue.

Given two arrays, both of unique elements. One represents the input queue and the other represents the output queue. The task is to check if the given output is possible through stack permutation.

Input : First array: $1, 2, 3$

Second array: $2, 1, 3$

Output : Yes

Procedure:
push 1 from input to stack
push 2 from input to stack
pop 2 from stack to output
pop 1 from stack to output
push 3 from input to stack
pop 3 from stack to output

Input : First array: $1, 2, 3$

Second array: $3, 1, 2$

Output : Not Possible.

P3. *A game of stacks.* Alexa has two stacks of non-negative integers, stack $A = [a_0, a_1, \ldots, a_{n-1}]$ and stack $B = [b_0, b_1, \ldots, b_{m-1}]$ where index 0 denotes the top of the stack. Alexa challenges Nick to play the following game:

- In each move, Nick can remove one integer from the top of either stack A or stack B.
- Nick keeps a running sum of the integers he removes from the two stacks.
- Nick is disqualified from the game if, at any point, his running sum becomes greater than some integer given at the beginning of the game.
- Nick's final score is the total number of integers he has removed from the two stacks.

Given A, B and x for g games, find the maximum possible score Nick can achieve (i.e., the maximum number of integers he can remove without being disqualified) during each game and print it on a new line.

Input Format:

The first line contains an integer, g (the number of games). The $3.g$ subsequent lines describe each game in the following format:

1. The first line contains three space-separated integers describing the respective values of n (the number of integers in stack A), m (the number of integers in stack B), and x (the number that the sum of the integers removed from the two stacks cannot exceed).
2. The second line contains n space-separated integers describing the respective values of $a_0, a_1, \ldots, a_{n-1}$. The third line contains m space-separated integers describing the respective values of $b_0, b_1, \ldots, b_{m-1}$.

Output Format:

For each of the g games, print an integer on a new line denoting the maximum possible score Nick can achieve without being disqualified.

P4. *Hide-and-Seek.* Recall how a group of children playing hide-and-seek choose IT (the "chor"). They stand in a circle, and the child (called the *counter*, say) recites some kind of a rhyme consisting of m words, starting with himself / herself and moving on to the next child after reciting each word in a circular fashion. The child at whom the rhyme ends is eliminated from subsequent rounds of counting. Counting for the next round starts from the child following the eliminated child (in circular order). The process continues until one child remains, who becomes IT.

Write a program to simulate the above activity. The children, numbered $1 \ldots N$, should be represented by nodes in a circular linked list . The circular linked list should shrink as children are eliminated from counting. Your program should print the order in which children are eliminated.

Input format:

Input will be given via stdin. An integer (say n) specifying the number of test cases, followed by n lines, each of which corresponds to a test case. These lines will each consist of 3 integers, corresponding to the number of children, the serial number of the child chosen as the counter, and the length of the rhyme, respectively.

Your program should print the serial numbers of the children in the order in which they are eliminated, and finish with the number of the child who becomes the "chor".

For each test case, your program should print the output in a single line.

For example, for the following input:

2
4 1 10
5 2 6

Your output should be:
2 3 1 4
2 4 3 1 5

P5. A *priority queue* is an abstract data type which is like a regular queue data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority (lower value) is served before an element with low priority (higher value). Now consider the following variation of the priority queue. Let the range of the priorities be $[0, 50)$. Divide the priorities into 5 groups, $[0, 10)$, $[10, 20)$, $[20, 30)$, $[30, 40)$, and $[40, 50)$. An element with priority "a" is placed in group $[x, y)$ provided $x \leq a < y$. If two elements are in the same priority group, their values are compared and the lowest valued element is processed.

Consider the following example.

element	priority
45	6
85	15
23	32
15	13

Then, the resulting priority queue will be:

$45, 15, 85, 23$

If a new element comes, say : 65 13.

The resulting priority queue will be changed to:

$45, 15, 65, 85, 23$

The deletion operation will remove 45.

Again deletion operation will remove 15 and so on.

Write a program that will take a file input which will consist of 3 operations : Insert k, Delete and Display and perform the operations as explained above.

Input format :

i x y : i stands for insert; x stands for element; y stands for priority

p: p stands for display the queue

d: d stands for delete an element from the queue

Input :

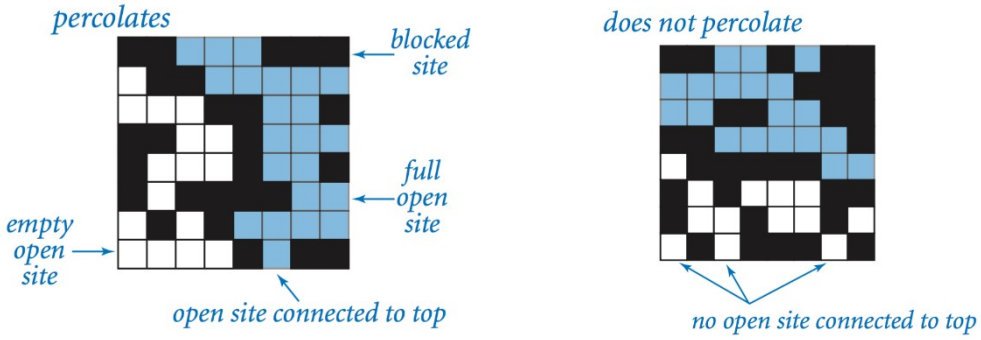
i 100 5
i 200 16
i 300 41
i 400 25
i 500 11
i 50 9
p
d
p
i 700 3
p

Output :

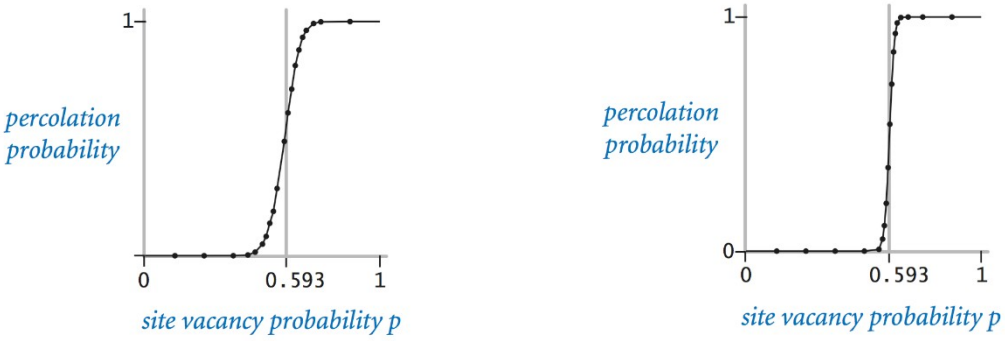
50 100 200 500 400 300
100 200 500 400 300
100 700 200 500 400 300

P6. *Union Find*. This assignment asks you to write a program to estimate the value of the percolation threshold for a porous substance below. Hopefully, by now, you have a library for the union find implementation, with realizations of all the different algorithms.

The model. We model a porous substance using an *n*-by-*n* grid of *cells*. Each cell is either *open* or *blocked*. A *full* cell is an open cell that can be connected to an open cell in the top row via a chain of neighbouring (left, right, up, down) open cells. We say a substance percolates if there is a full cell in the bottom row. In other words, a system percolates if we fill all open sites connected to the top row and that process fills some open site on the bottom row. *In other words, a system that percolates has a path from top to bottom, with full sites percolating.* For the porous substance, the open sites correspond to empty space through which water might flow, so that a system that percolates lets water fill open sites, flowing from top to bottom.



The problem. In a famous scientific problem, researchers are interested in the following question: if sites are independently set to be open with probability *p* (and therefore blocked with probability 1 − *p*), what is the probability that the system percolates? When *p* equals 0, the system does not percolate; when *p* equals 1, the system percolates. The plots below show the site vacancy probability *p* versus the percolation probability for a 20-by-20 random grid (left) and 100-by-100 random grid (right).



Assignment: When *n* is sufficiently large, there is a *threshold* value *p** such that when *p* < *p**, a random *n*-by-*n* grid almost never percolates, and when *p* > *p**, a random *n*-by-*n* grid almost always percolates. Your task is to write a computer program to estimate *p**.

Hint: *You do not have to adhere to the description below, it is perfectly fine to use your own.*

To model a percolation system, we create an appropriate data type with the following supported functions:

```
Percolation (int n)              // create n-by-n grid, with all sites blocked
void open(int row, int col)      // open site (row, col) if not open already
int  isOpen(int row, int col)    // is site (row, col) open?
int  isFull(int row, int col)    // is site (row, col) full?
int numberOfOpenSites()          // number of open sites
```

int checkIfPercolates() // does the system percolate?

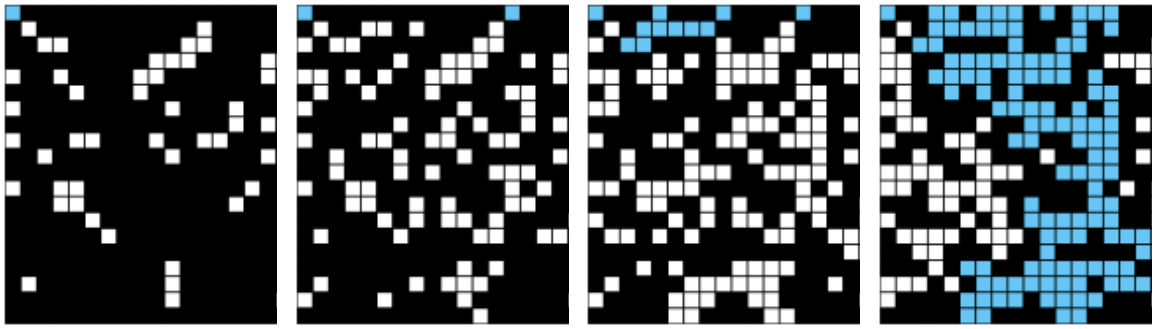
int main(int argc, char **argv) // test client (optional)

Corner cases. By convention, the row and column indices are integers between 1 and *n*, where (1, 1) is the upper-left site. Issue appropriate errors if any argument to open(), isOpen(), or isFull() is outside its prescribed range, or if $n \le 0$. If you are using g++ to compile the code, you can use **bool** as the return type of the functions **isOpen, isFull, checkIfPercolates** functions.

Monte Carlo simulation. To estimate the percolation threshold, consider the following computational experiment:

- Initialize all sites to be blocked.
- Repeat the following until the system percolates:
  - Choose a site uniformly at random among all blocked sites.
  - Open the site.
- The fraction of sites that are opened when the system percolates provides an estimate of the percolation threshold.

For example, if sites are opened in a 20-by-20 lattice according to the snapshots below, then our estimate of the percolation threshold is 204/400 = 0.51 because the system percolates when the 204th site is opened.



50 open sites *100 open sites* *150 open sites* *204 open sites*

By repeating this computation experiment *T* times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let x_t be the fraction of open sites in computational experiment *t*. The sample mean x^* provides an estimate of the percolation threshold; the sample standard deviation *s* measures the sharpness of the threshold.

$$x^* = (x_1 + x_2 + \ldots + x_T) / T, \; s^2 = [(x_1 - x^*)^2 + (x_1 - x^*)^2 + \ldots + (x_T - x^*)^2] / (T - 1)$$

Assuming *T* is sufficiently large (say, at least 30), the following provides a 95% confidence interval for the percolation threshold:

$$[x^* - 1.96s / \sqrt{T}, \; x^* + 1.96s / \sqrt{T}]$$

To perform a series of computational experiments, create a data type PercolationStats with the following functions.

void PercolationStats(int n, int trials) // perform trials independent experiments on an n-by-n grid
double mean() // sample mean of percolation threshold
double stddev() // sample standard deviation of percolation threshold

double confidenceLo() // low endpoint of 95% confidence interval
double confidenceHi() // high endpoint of 95% confidence interval

int main(int argc, char ** argv) // test client (described below)

You should throw an error if either $n \le 0$ or $trials \le 0$.

The main() method takes two *command-line arguments* n and T, performs T independent computational experiments (discussed above) on an n-by-n grid, and prints the sample mean, sample standard deviation, and the *95% confidence interval* for the percolation threshold.

Example Runs (assume PercolationStats is the name of the executable)

% **./PercolationStats 200 100**
mean = 0.5929934999999997
stddev = 0.00876990421552567
95% confidence interval = [0.5912745987737567, 0.5947124012262428]

% **./PercolationStats 200 100**
mean = 0.592877
stddev = 0.009990523717073799
95% confidence interval = [0.5909188573514536, 0.5948351426485464]

% **./PercolationStats 2 10000**
mean = 0.666925
stddev = 0.11776536521033558
95% confidence interval = [0.6646167988418774, 0.6692332011581226]

Analysis of running time and memory usage

Implement the Percolation data type using the *quick find* algorithm.

- Measure the total running time of PercolationStats for various values of n and T. How does doubling n affect the total running time? How does doubling T affect the total running time?

- Now, implement the Percolation data type using the *weighted quick union* algorithm

Acknowledgement: This assignment was developed with help from Bob Sedgewick and Kevin Wayne.