Introduction to Algorithms	Problem Set 7
CS 4820, Spring 2019	Due 11:59pm Thursday, March 21

Hand in your solutions electronically using CMS. Each solution should be submitted as a separate file. Collaboration is encouraged while solving the problems, but:

1. **list the names of those with whom you collaborated;**
2. **you must write up the solutions in your own words;**
3. **you must write your own code.**

Remember that when a problem asks you to design an algorithm, you must also prove the algorithm's correctness and analyze its running time. The running time must be bounded by a polynomial function of the input size.

(1) *(10 points)*

(1a) *(3 points)* Let $G = (A \cup B, E)$ be an undirected p-regular bipartite graph, i.e., every vertex in $A \cup B$ has degree exactly p. Prove that G has a perfect matching.

(1b) *(7 points)* Latin rectangles are well studied objects in combinatorics, and various popular puzzles are based on Latin rectangles. For integers $m \leq n$, any $m \times n$ matrix, with each entry in the set $\{1, \ldots, n\}$, such that each integer appears at most once in each row and at most once in each column is called an (m, n)-Latin rectangle. An (n, n)-Latin rectangle is called an n-Latin square. Prove that any (m, n)-Latin rectangle can be extended to an n-Latin square. Given as input an (m, n)-Latin rectangle, design an efficient algorithm that extends the Latin rectangle to an n-Latin square.

(2) *(10 points)* You are organizing a carnival for the town of Ithaca. There are various artists, arriving from all over the globe, to perform in various shows as part of the carnival. A total of n artists are participating from a total of m countries. Further, there are a total of r shows. The following are the constraints on organizing the carnival:

1. Artist ℓ, subject to her skills, is only capable of participating in a subset of the shows given by a list L_ℓ.
2. The i'th show requires exactly n_i artists to perform.
3. Each show can have at most one artist from each country.
4. So as to be fair to all artists, each artist can participate in at most r' shows.

As part of the input, you will receive integers $n, m, r, r', n_1, \ldots, n_r$ and the capability list $L_1, \ldots, L_n$ of the artists. Your task is to design an efficient algorithm to determine if you can organize the carnival subject to the above restrictions. If indeed it is possible, you have to output the allocation of artists to shows.

(3) Extra Credit Question *(5 points)* Give an example of a bipartite graph, with countably infinite nodes on each side, which satisfies Hall's condition but does not have a perfect matching.

(4) *(15 points)* In this problem, you will implement a basic version of image segmentation, as described in section 7.10 of the textbook. You should read this before getting started. At a high level, your algorithm will take in images as input, highlight the foreground, and shade the background.

In addition to test cases, we provide you with two files: `Main.java`, and `ImageHelper.java`. `ImageHelper.java` is used to preprocess images into easy-to-parse text files, and to convert the program's output back into a displayable image. **You should not change this file, and it is included only for your entertainment** (so you can observe the highlighting sections of a real image). `Main.java` is the only file you must change. You can change it however you wish, but note that we have already filled in code to parse input and produce output in the right format.

Recall the image segmentation problem from 7.10. In this problem, you are given an 2D matrix of pixels M with $\mathsf{height} \in \mathbb{N}, \mathsf{width} \in \mathbb{N}$. Each pixel $(i, j) \in [\mathsf{height}] \times [\mathsf{width}]$ has a *reward* for assigning it to the foreground, $f_{i,j}$, and a reward for assigning it to the background, $b_{i,j}$. We assume that the pixels are laid out in a grid graph, as in figure 7.18(a) in the book. For every pair of pixels $i, x \in [\mathsf{height}], j, y \in [\mathsf{width}]$ that are adjacent – i.e. $|i - x| + |j - y| = 1$ – there is a *separation penalty* $p_{\alpha,\beta}$ incurred if pixels $\alpha = (i, j), \beta = (x, y)$ are not both assigned to the foreground or the background respectively. The goal then is to find a set $F \subseteq [\mathsf{height}] \times [\mathsf{width}]$ that maximizes

$$\phi(F) = \sum_{(i,j)\in F} f_{i,j} + \sum_{(i,j)\in \overline{F}} b_{i,j} - \sum_{(\alpha,\beta)\in \mathcal{S}} p_{\alpha,\beta},$$

where $\mathcal{S} := \{(\alpha = (i, j), \beta = (x, y)) : |i - x| + |j - y| = 1, \{\alpha, \beta\} \not\subseteq F, \{\alpha, \beta\} \not\subseteq \overline{F}\}$.

Given the loaded input to `Main.java`, you should aim for a time complexity of $O((\mathsf{height} \cdot \mathsf{width})^2)$. We will impose a time bound of **5 seconds** on your code. Unfortunately, for interesting images (i.e., those of dimensions much greater than 50×50), an algorithm of the aforementioned asymptotic complexity will be too slow.

So to truly appreciate this application, we must introduce one small twist. We instead ask you to solve the `BlockImageSegmentation` problem, which involves tiling the image with equally sized blocks, and assigning each block to the foreground or background so as to maximize a similar function. Essentially, you will be solving the same problem for a scaled-down version of the image, with the each pixel of the new image representing the average of a block in the original one:

- The reward for putting the block (I, J) into the foreground (background) should be the **floor** (greatest integer no bigger than) of the **average** of all rewards of the pixels in the block, i.e.

$$\left\lfloor \frac{\sum_{y=0}^{\mathsf{blockHeight}-1} \sum_{x=0}^{\mathsf{blockWidth}-1} \mathsf{fReward}[I \cdot \mathsf{blockHeight} + y][J \cdot \mathsf{blockWidth} + x]}{\mathsf{blockHeight} \cdot \mathsf{blockWidth}} \right\rfloor.$$

- The separation penalty between two adjacent blocks (I, J) and (I', J') should likewise be the floor of the average of separation penalties between all pairs of adjacent pixels **crossing the edge between the blocks**, i.e. where one pixel is in each of the blocks.

When `Main.java` is done reading, your input will be provided in the following variables:

- $1 \le \mathsf{height} \le 10^7$: the height of the image.
- $1 \le \mathsf{width} \le 10^7$: the width of the image.
- $1 \le \mathsf{blockHeight} \le 10^7$: the height of a block, i.e. the vertical scaling factor. $\mathsf{height}/\mathsf{blockHeight} \le 50$ is an integer.
- $1 \le \mathsf{blockWidth} \le 10^7$: the width of a block, i.e. the horizontal scaling factor. $\mathsf{width}/\mathsf{blockWidth} \le 50$ is an integer.
- For $0 \le i < \mathsf{height}$, $0 \le j < \mathsf{width}$:
  - $0 \le \mathsf{fReward}[i][j] \le 100$: reward for putting pixel (i, j) in the foreground.

- $0 \leq \mathsf{bReward}[i][j] \leq 100$: reward for putting pixel (i, j) in the background.
- $0 \leq \mathsf{pBtwCols}[i][j] \leq 100$: separation penalty between pixels $(i, j), (i, j+1)$.
- $0 \leq \mathsf{pBtwRows}[i][j] \leq 100$: separation penalty between pixels $(i, j), (i+1, j)$.

You should **write your output to the following variables**:

- For $0 \leq i < \mathsf{height}$, $0 \leq j < \mathsf{width}$, $\mathsf{foreground}[i][j]$ should be **true** if the pixel (i, j) *belongs to a block that is in the foreground*, and **false** if it is in the background.

That's it! Again, we take care of all image processing, loading input, and printing output. We will provide you with some public test cases to try out your code. A public test will consist of a few things:

- An image file, of the form `image.png`
- A more structured and parsable version of this image, of the form `image_testin.txt`
- A structured and parsable version of the solution to this instance, of the form `image_testout_guide_soln.txt`

To test your code for correctness (after filling in `Main.java`), simply execute the following commands:

- $ `javac Main.java`
- $ `java Main < image_testin.txt > image_testout_my_soln.txt`
- $ `cmp image_testout_guide_soln.txt image_testout_my_soln.txt`

If the last command has no output, that means you're good to go! For final grading, as always, you should upload `Main.java` to the autograder at `https://cs4820.cs.cornell.edu` to check its correctness. This will run your code on a small number of public test cases, but you won't be able to see the actual visualization of your algorithm's output on the original image (which is the most fun part)! To do this, simply execute the commands:

- $ `javac ImageHelper.java`
- $ `java ImageHelper 0 image_testout_my_soln.txt image.png filtered_image.png`

and open up `filtered_image.png` with an image viewer of your choice to check it out!