Introduction to Algorithms	Problem Set 3
CS 4820, Spring 2019	Due 11:59am Tuesday, February 19

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)* In American politics, *gerrymandering* refers to the process of subdividing a region into electoral districts to maximize a particular political party's advantage. This problem explores a simplified model of gerrymandering in which the region is modeled as one-dimensional. Assume that there are $n > 0$ *precincts* represented by the vertices $v_1, v_2, \ldots, v_n$ of an undirected path. A *district* is defined to be a contiguous interval of precincts; in other words a district is specified by its endpoints $i \leq j$, and it consists of precincts $v_i, v_{i+1}, v_{i+2}, \ldots, v_j$. We will refer to such a district as $[i, j]$.

Assume there are two parties A and B competing in the election, and for every $1 \leq i \leq j \leq n$, $P[i, j]$ denotes the probability that A wins in the district $[i, j]$. The probability matrix P is given as part of the input. Assume that the law requires the precincts to be partitioned into exactly k disjoint districts, each containing at least $s_{\min}$ and at most $s_{\max}$ nodes. You may also assume the parameters $n, k, s_{\min}, s_{\max}$ are chosen such that there is at least one way to partition the precincts into k districts meeting the specified size constraints. Your task is to find an efficient algorithm to gerrymander the precincts into k districts satisfying the size constraints, so as to maximize the expected number of districts that A wins.

(Extra Credit)** *(10 points) [Attempt at your own risk. We do not know how to solve this, or even if a solution exists. The deadline for this part is* ***May 7, 2019****]* The input is the same as above. Your task now is to design an efficient algorithm to gerrymander the precincts into exactly k districts to maximize the probability that A wins more than $k/2$ districts.

(2) *(15 points)* In the sport of American football, teams move a ball toward the opposing team's goal line in a sequence of plays called an offensive drive. Subject to some simplifications, the game has the following rules.

1. An offensive drive starts n yards away from the other team's goal line. In American football, n is often but not always equal to 80.

2. To retain possession of the ball, the team must move forward at least k yards in its first d plays, called "downs". In American football, $k = 10$ and $d = 4$.

3. More generally, certain plays in an offensive drive are called "first downs". The initial play in the drive is a first down. A subsequent play is called a first down if and only if the total yardage accumulated since the preceding first down is greater than or equal to k.

4. There are three ways that a drive could end.

    - If the total yardage accumulated in the drive is greater than or equal to n, the team scores a touchdown.

- If the team has completed d plays since the last first down, and the total yardage accumulated in those d plays is less than k, the team loses possession.
- Any play could result in a "turnover". If this happens, the team loses possession immediately.

5. On any given play of the drive, the team needs to decide whether to try a passing play or a running play. The probability of a turnover, and the distribution of the random number of yards gained in the event that the play is not a turnover, depend on whether the team chooses passing or running. We will assume that the number of yards gained is always a non-negative integer.[1]

A *football strategy* is a strategy for choosing between a running or passing play in every possible situation that may arise during an offensive drive. A strategy is *optimal* if it maximizes the probability of scoring a touchdown.

For an integer y in the range $0 \leq y \leq n$, let p_y denote the probability of gaining y yards on a passing play, and let r_y denote the probability of gaining y yards on a running play. Let p_{n+1} and r_{n+1} denote the probability of a turnover on a passing or running play, respectively. Assume these are non-negative rational numbers and that $\sum_{y=0}^{n+1} p_y = \sum_{y=0}^{n+1} r_y = 1$. Design an algorithm which is given the parameters n, k, d and the probabilities p_y, r_y for $y = 0, 1, \ldots, n+1$, and which determines whether the optimal football strategy chooses to run or to pass at the start of an offensive drive, n yards away from the opponent's goal line. Your algorithm should work for *any* values of n, k, d, not only for $n = 80, k = 10, d = 4$.

(3) *(10 points, with option for 5 additional bonus points)* In this problem you are asked to implement an algorithm that solves *optimum sequence alignment*, as described in sections 6.6 and 6.7 in the textbook. You should read at least 6.6 before getting started. Your implementation should run $O(nm)$ time, where n and m are the lengths of the two strings to be aligned. Note that any such algorithm will also use $O(nm)$ space (can you see why?). **As an extra challenge, you can get up to 5 bonus points for implementing an algorithm that runs in $O(n+m)$ space, as described in section 6.7.**

Implement the algorithm in Java. The only libraries you are allowed to `import` are the ones in `java.util.*, java.io.*`. There is no `Framework.java` provided for this assignment. You will need to implement your Java program from scratch. **Warning**: be aware of the space and time complexity of any data structures / methods you use from a built-in Java class, as these will count towards the complexity of your code.

Your Java program should take input from `stdin` and write output to `stdout`. We recommend using the `BufferedReader` and `BufferedWriter` classes, as these have much better performance than `Scanner` and `System.out.println`. We will use an online autograder (URL to be released shortly on Piazza) that will allow you to upload your code and test it on a small number of public test cases. When we grade the assignment, we will run it on a larger number of more complex private test cases.

Your algorithm should read data from standard input in the following format:

- The first line contains two positive integers $m, n \in [1, 10^4]$, separated by a space, representing the number of symbols in the first and second string to be compared, respectively.
- The second line is a string $X = x_1 x_2 \cdots x_m$, with each x_i equal to one of the 26 symbols in the English alphabet.
- The third line is a string $Y = y_1 y_2 \cdots y_n$, with each y_i equal to one of the 26 symbols in the English alphabet.

[1] In the actual sport of football, it's possible for a play to gain a negative number of yards. We are deliberately ignoring this possibility, for the sake of simplicity.

- The fourth line contains a positive integer, $\delta \in [1, 30]$, representing the *gap* penalty: each symbol in X that is not matched to a symbol in Y, and each symbol in Y that is not matched to a symbol in X, will incur a penalty of δ.
- The next 26^2 lines each contain, separated by spaces, two (*not necessarily distinct*) English letters ϕ, ψ and a positive integer $\alpha_{\phi,\psi} \in [1, 30]$ representing the penalty of matching x_i with y_j when $\{x_i, y_j\} = \{\phi, \psi\}$. You may assume these lines appear in the natural lexicographic order (aa, ab, ..., az, ba, ..., bz, ..., za, ..., zz) and that costs are symmetric (i.e., $\alpha_{\phi,\psi} = \alpha_{\psi,\phi}$).

Your algorithm should output data in the following format:

- The first line should contain an integer indicating the cost of the optimum (minimum-cost) alignment.
- The second line should contain an integer p indicating the number of matches made in your alignment between symbols of X and symbols of Y.
- The next p lines should each contain two integers $i \in [m], j \in [n]$, separated by a space, indicating a match made in your alignment between x_i and y_j. **These p lines should list the matches in order of increasing i.**

Example:

- Input:

```
4 4           # 4 symbols in the first word, 4 in the second
pear          # first string
desk          # second string
3             # gap penalty of 3
a a 0         # penalty of 0 for matching identical letters
a b 7         # penalty of 7 for mismatching vowel with consonant
...
a e 2         # penalty of 2 for mismatching vowel with vowel
...
b c 2         # penalty of 2 for mismatching consonant with consonant
...
z z 0
```

- Output:

```
10            # cost of optimum alignment
3             # number of matches in optimum alignment
1 1           # p matched with d
2 2           # e matched with e
4 3           # r matched with s
```

A test case may yield more than one optimum alignment. This is OK: we will accept any minimum cost alignment. We simply require that your output cost is minimum and that your outputted matches are valid (i.e., no "crossing" pairs) and can generate that cost. Partial credit will be given if the minimum cost is correct but is not generated by your outputted matches.

Remember that your code must run in $O(nm)$ time, and that you may earn up to 5 bonus points for implementing an algorithm that uses $O(n + m)$ space. We will impose a runtime limit of **5 seconds** on each instance. You should make sure you pass the public test cases within this time. There will be no public test cases to test the extra credit. We will use Java 8 to compile and test your program.