Introduction to Algorithms	Problem Set 3
CS 4820 Spring 2016	Due 11:59pm Tuesday, February 23

Your homework submissions need to be typeset (hand-drawn figures are OK). See the course web page for suggestions on typing formulas. Solution to each question needs to be uploaded to CMS as a separate pdf file. (Questions with multiple parts need to be uploaded as a single file.)

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.

We will accept late homeworks (using late days) as usual, but may not be able to grade late homeworks before the prelim!

Prelim 1 is scheduled for Tuesday March 1st, starting at 7:30 p.m. The prelim will roughly cover the material from the first 3 problem sets (Chapters 1-4 and 6). In each chapter you are only required to know sections we covered. For Chapter 6, this includes all of Sections 6.1-6.4 6.6, and 6.8, some of which we will cover while you are working on this homework. **Please contact TA Ronan LeBras (lebras@cs.cornell.edu) ASAP if you have a conflict or need any special accommodation for the test.**

Some Comments on Dynamic Programming Problems. The following problems can all be solved by a dynamic programming approach. For problem 1 all you have to submit is your code, and we'll test it by running it on some examples. In writing up dynamic programming algorithms for problems 2 and 3, all the guidelines from earlier in the course apply; but there are some additional things worth keeping in mind as well. If your solution consists of a dynamic programming algorithm, you must clearly specify the set of sub-problems you are using, and the recurrence you are using — describing what they mean in English as well as any notation you define. You must also explain why your recurrence leads to the correct solution of the sub-problems — this is the heart of a correctness proof for a dynamic programming algorithm. Finally, you should describe the complete algorithm that makes use of the recurrence and sub-problems.

A description consisting of a piece of pseudo-code without these explanations is not a valid answer for problems 2 and 3.

(1) Consider the following problem where Alice is collecting colored numbers (which represents the amount of rewards) in a line. There are n cells in the line starting from 0(leftmost) to $n-1$(rightmost). Alice is originally at cell 0 with color red, and can only go right in every step. There is a colored number in each cell (0 to $n-1$), where the color could be red (0), blue (1) or yellow (2). When Alice is in cell i, she could decide to either "pick that colored number and become that color", or "skip it and do not change her color". Alice can pick colored number in cell 0 as well since she was originally in that cell. So if Yellow Alice collects a blue "5", then she becomes Blue Alice after leaving that cell; but if Yellow Alice skips the blue "5", she is still Yellow Alice. If Alice collects a colored number, and she was in the same color when entering the cell, she gets double score for that number; otherwise she only gets score equals to that number. Give an algorithm that helps Alice find the best strategy to maximize her score.

Examples:

- If $n = 4$, and the colored numbers are: red 5, blue 10, yellow 6, red 7. Then Alice will get $5 * 2 + 10 + 6 + 7 = 33$ points, if she collects all the numbers. And she will become Blue Alice after leaving cell 1, Yellow Alice after leaving cell 2, and Red Alice finally.
- If $n = 3$, and the colored numbers are: red 1, blue 2, red 3. Then Alice will get $1 * 2 + 2 + 3 = 7$ points, if she collects all the numbers. However, she should skip the blue number and keep color red, and get $1 * 2 + 3 * 2 = 8$, which is the optimal solution.

Your algorithm needs to use the dynamic programming framework with the following subproblems: **score[c][i]** equals to the maximum score when Alice is starting at cell i with color c and goes to cell $n-1$. So the maximum reward Alice can collect will be **score[0][0]**, and we claim that **score[c][i]** can be determined using a dynamic programming recurrence using **score[c'][j]** for $j > i, c' = 0, 1, 2$.

In addition, your algorithm should populate the Boolean array **picked**, where *picked*[i] is `true` if Alice should pick the colored number in cell i, and `false` otherwise.

This problem is asking you to design and implement this algorithm in $O(n)$ time, outputting both the value that Alice can collect (i.e. the value *score*[0][0]), as well as the sequence of steps she should take (i.e. the array *picked*). Algorithms that take more than $O(n)$ time will only get partial credit.

For the code, you must use the framework code (*Framework.java*) we provide on CMS. We will put the code you have submitted in the part of the framework that we have specified by the commented line
//YOUR CODE GOES HERE
and run the code.

Starting this homework we require you to **submit both *Fragment.txt* and *Framework.java***. We will first test your program using *Fragment.txt*, and if it fails to compile, we will use your *Framework.java* file (which is a whole java program that **includes** your *Fragment.txt*, and works well on your own computer) to test instead. However, you will get at most 5 out of 10 points if your *Fragment.txt* file fails to compile, even if your *Framework.java* file passes all the test data.

Please read the provided framework carefully before starting to write your code. You can test your code with the test cases provided on CMS. *Framework.java* will take two command line arguments. The first one is the name of the input file, and the second is the name of the output file. The input file should be in the same folder in which your compiled java code is. After you compile and run your code, the output file will also be in the same folder. In order to test your code with the provided test cases, copy the test cases in the folder in which you have compiled your code, and set the name of the input file to be the name of one of the sample inputs (*SampleTest_i.txt* in which $0 \leq i \leq 5$). You can compare your output with the provided sample outputs (*SampleOutput_i.txt* in which $0 \leq i \leq 5$).

The format of the input file is the following:

- The first line has one number, n, which is the number of cells.
- In the i-th line of the next n lines, there are two numbers, c, r, which represent the color c and reward r for the colored number in cell $i-1$.

In the output file, you will need to output two lines. The first line is the best reward that Alice could get, and the second line has n numbers of either 1 or 0, where the i-th number is 1 if Alice will pick the colored number in cell $i-1$, and 0 if Alice will skip it. Our framework handles reading the input from the input file and writing the output in the output file, so your job is only to implement the algorithm. More specifically, we read the data into the array *rewards* and the array *colors*, and your task is to compute the array *score* and *picked*.

(2) In class we considered the sequence alignment problem. One common mistake in mistyping words is swapping two neighboring letters, so for example, typing "peice" instead of "piece". To also account for this kind of mistake, we will have the usual parameters from sequence alignment, in addition to this new swapping cost.

- parameter $\delta > 0$ that defines a *gap penalty.*
- for each pair of letters p, q in our alphabet, there is a *mismatch cost* of α_{pq} for lining up p with q.
- for each pair of letters p, q in our alphabet, there is a *swapping cost* of β_{pq} for having pq in one string, and lining up these two letters with the letters qp in this order in the other string.

Suppose we are given two strings X and Y, where X consists of the sequence of symbols $x_1 x_2 \cdots x_m$ and Y consists of the sequence of symbols $y_1 y_2 \cdots y_n$, and the penalties δ, α, β described above, give an $O(mn)$ algorithm to determine the cost of the best alignment between the two strings. Notice that in this problem we are not asking you to output the alignment though that is also possible to do in the $O(mn)$ time.

(3) Word processors aim to print paragraphs with right margin of all lines except the last line as aligned as possible. This is called "pretty-printing". Suppose you have a text for printing, consisting of a sequence of *words*, $W = \{w_1, w_2, \ldots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L. We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of a partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word except the last; and so if $w_j, w_{j+1}, \ldots, w_k$ are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \leq L.$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line—that is, the number of spaces left at the right margin. Note that for simplicity of the problem, we include the slack of the last line also in this computation.

Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the *squares* of the slacks of all lines including the last line is minimized.[1]

[1] A more typical version of pretty-printing doesn't contain the squared slack of the last line, i.e., it is OK if the last line is very short. The solution of the problem you solve here can also be used to solve this slightly harder problem.