ASSIGNMENT 3

DUE: Sunday, March 4, before midnight. DO NOT COPY. ACKNOWLEDGE YOUR SOURCES. For each problem, please justify its correctness and analyze its time complexity. Please read http://www.student.cs.uwaterloo.ca/~cs341 for general instructions and policies.

1. Chocolate breaking problem (10 marks)

Consider an \( m \times n \) bar of chocolate. For simplicity we will refer to the vertical line between column \( i \) and column \( i + 1 \) bars as \( C_i \) and the horizontal line between row \( j \) and row \( j + 1 \) as \( R_j \). Therefore an \( m \times n \) chocolate has vertical lines \( C_1, ..., C_{n-1} \) and horizontal lines \( R_1, ..., R_{m-1} \). As an example, the leftmost figure in Figure 1 shows a 3 by 2 bar with \( C_1 \) vertical and \( R_1 \) and \( R_2 \) horizontal lines. Your goal is to design an algorithm that breaks the chocolate into \( m \times n \) many 1 by 1 bars. The only operation your algorithm can use is to break the chocolate along an entire vertical or horizontal line. For example, one way to break the 3 by 2 bar is to break along \( C_1 \) first, then \( R_1 \) and finally \( R_2 \). This sequence of operations is shown in Figure 1. Note that you cannot take one partial \( k \) by \( t \) bar that you generate along the way, where \( k < m \) and/or \( t < n \), and only break that that partial bar. For example, after operation 1 in Figure 1, you cannot take the left 3 by 1 bar and only break that along \( R_1 \), while leaving the right 3 by 1 bar intact. If one can only use this operation, notice that any algorithm has to do exactly \( n - 1 + m - 1 = n + m - 2 \) operations to break the chocolate into \( m \times n \) many 1 by 1 bars: each algorithm has to eventually pick each \( C_i \) and \( R_j \) in some order and break it. Therefore each algorithm can be thought of as some permutation of \( \{C_1, ..., C_{n-1}, R_1, ..., R_{m-1}\} \).

![Figure 1: An example of a sequence of 3 operations to break a 3 by 2 bar into 6 1x1 bars.](image)

The cost of an operation is given as follows. Let \( wc_i \) be the “cost factor” of breaking vertical line \( C_i \) and \( wr_j \) be the “cost factor” of breaking horizontal line \( R_j \). Let \( a_i \) be the number of partial bars that is broken when breaking along a vertical line \( C_i \). Then the cost of breaking along \( C_i \) is \( a_i \times wc_i \). Similarly, if breaking along a horizontal line \( R_j \) breaks \( a_j \) partial bars, then the cost this operation is \( a_j \times wr_j \). For example the cost of the algorithm in Figure 1 is: \((1 \times wc_1) + (2 \times wr_1) + (2 \times wr_2)\). That’s because breaking \( C_1 \) breaks one (partial) bar, whereas breaking \( R_1 \) and \( R_2 \) both break 2 partial bars.

Design a greedy algorithm that runs in \( O((n + m) \log(n + m)) \) time and outputs the sequence of breaks with the minimum cost. You can assume for simplicity that all \( wc_i \) and \( wr_j \) are distinct.
Hint: Consider renaming the set \( \{wc_1, \ldots, wc_{n-1}, wr_1, \ldots, wr_{m-1}\} \) as \( W = \{w_1, \ldots, w_{n+m-2}\} \). You can assume each element in \( W \) is a pair \((w_l, C_i)\) or \((w_l, R_j)\), so \( w_l \) has with it which horizontal or vertical line's weight it corresponds to. Then greedily break the lines using \( W \). There are multiple ways to argue the correctness of your algorithm but consider an argument that’s similar in spirit to the exchange argument we used to prove that the job scheduling algorithm that greedily picks the highest length/weight job is optimal.

2. **Shopping Addict** (10 marks) A dollar store is on sale. There are \( N \) items, the \( i \)-th item has a price tag \( p(i) \) dollars, discounted from the original \( p'(i) \geq p(i) \) dollars. Thus, if you buy the \( i \)-th item, you save \( p'(i) - p(i) \) dollars. Design a dynamic programming algorithm that takes as input \( N, M, p(i), p'(i) \), for \( i = 1 \ldots N \), spends \( M \) dollars on these \( N \) items and achieves the maximum possible saving. Your algorithm only needs to output the maximum saving. It is ok to repeatedly buy one item and you do not need to buy everything.

3. **Parsing Chinese** (10 marks) It is well-known that a Chinese sentence, unlike English, is a sequence of consecutive words without spaces separating them. Segmenting such a sentence is a major task for Chinese language processing. In this problem, we will use English to mimic this situation. Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words, by dynamic programming. That is, the output of your algorithm should be true, if the input can be segmented into a space-separated sequence, and false otherwise. For example, if the input string is: “ilikecs341” and with a dictionary \{i, like, cs341, other, words\}, you output “yes”; if the input string is: “ihatecs341” with the same dictionary, you output “no”.

4. **Coin Game** (10 marks) Consider a row of \( n \) coins of values \( v_1, \ldots, v_n \). We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Suppose player 1 starts the game first. Determine, by dynamic programming, the maximum possible amount of money player 1 can definitely win. (You only need to output the maximum amount. No need to output the choices.) That is, we are asking for the maximum amount of money the player with the first move can guarantee to win, no matter what player 2 does in the rest of the game. Specifically, in each move, player 2 can take the first or the last coin, and if your algorithm returns \( X \), no matter what player 2 does in each move, the amount player 1 wins has to be guaranteed to be greater or equal to \( X \). As a simple example consider the following set of coins: 2, 4, 5, 1. In this example, player 1 has two options: 1) take the first coin with value 2 or; 2) take the last coin with value 1. Let’s consider each case:

- **Taking 2** In his first move, player 2 can take 1 or 4. In either case, player 1, can take 5 in her second move and win a total of 2+5=7. Therefore, this move guarantees winning 7.
- **Taking 1** In his first move, player 2 can take 2 or 5. If player 2 takes 2, player 1 can take 5 and win a total of 1+5=6. If player 2 takes 5, player 1 can take 4 and win 1+4=5. Therefore, this move only guarantees winning 5.

The answer in this example is therefore 7 (and the move to guarantee winning 7 is taking the first coin 2).
Hint 1: Let the initial set of coins be stored in an array $C[1...n]$. Notice that at any point in the game, a substring of $C[i...j]$ is left in the game, where $i \leq j$. Consider constructing your subproblems in terms of $C[i...j]$.

Hint 2: There are multiple ways to formulate this DP. You do not have to use this hint but you can find the following fact useful. At any point of game, (The maximum possible remaining gain of the first mover) = (the total remaining coin value) - (maximum possible remaining gain of the other player after the first mover picks her coin). You don’t have to prove this fact.

5. **Sequence alignment** (10 marks)

Implement the sequence alignment problem by dynamic programming (as we have discussed in class), in C++ or Python.

The input format is:

- First line: DNA sequence $X$;
- Second line: DNA sequence $Y$;
- Third line: gap penalty $\delta$ as a positive integer, for insertions and deletions;
- Fourth line: mismatch penalty $p$, as a positive integer, meaning $\alpha(x_i, y_j) = p$ if $x_i \neq y_j$, and 0 if $x_i = y_j$. Note that we are assuming all mismatches (e.g. C matching G, or A matching C) have the same penalty.

Please use a dash to represent a gap. The output format is an alignment of the two input sequences minimizing the total penalty using the penalty parameters given in the input. In the case that there are multiple optimal alignments, your program may just output any one of them. Your program will read from the standard input and write to the standard output.

For example, for the following input:

```
ACTTTTACGGGGATCCCAAACACA
CTTACTGGGGCCTTCACA
1
1
```

Your program should output the following (or another optimal alignment in the same format):

```
ACTTTTACGGGGATCCCAAACACA
|| |||| |||| || ||||
-CT--TACTGGGG--CCTTCACA
```

Here, the character used in the second line of the output is the vertical bar (not capital i or lower case L). The submission instructions will be given on piazza.