1 Edsger’s Apfelstrudel

You are eating at a cozy little restaurant which serves a *prix fixe* menu of \( k + 1 \) courses, with several available choices for each course. Each dish belongs to exactly one course (e.g., risotto can only be ordered as an appetizer, not a main), and you are effectively indifferent between most of the items on the menu (because they are all so tasty), but the main draw of this particular restaurant is that they serve a delicious bottomless dessert: their world-famous Viennese-style apple strudel. However, there are also a few interesting rules:

1. You must finish your current dish before ordering another.

2. Each dish after the first course depends on what you ordered in the previous course, e.g., you can only order salmon for your main if you ordered a Caesar salad or chicken noodle soup for the previous course. You are told on the menu exactly what these restrictions are before you order anything.

3. Most importantly, you are not allowed to have their unlimited dessert unless you finish one dish from each of the first \( k \) courses!

You are told the caloric value of each item in each course on the menu, and you plan your meal with a twofold goal: to be able to order the strudel, but also to save as much room in your stomach as possible for the unlimited dessert. Assuming that there are no celery dishes (i.e., every dish on the menu has at least 0 Calories), design an algorithm to find the smallest number of Calories you can consume and still order the bottomless strudel. If you would like, you may assume the very first course has exactly one choice (e.g., a single leaf of spinach worth 0 Calories).

2 Rod Cutting

Suppose we have a rod of length \( k \), where \( k \) is a positive integer. We would like to cut the rod into integer-length segments such that we maximize the *product* of the resulting segments’ lengths. Multiple cuts may be made. For example, if \( k = 8 \), the maximum product is 18 from cutting the rod into three pieces of length 3, 3, and 2. Write an algorithm to determine the maximum product for a rod of length \( k \).

3 Cheeseville

The streets of Cheeseville form an \( k \times k \) grid, and one of \( k^2 \) mice lives at the corner of each intersection. You’re about to place a big slice of cheese at the South-West corner of Cheeseville, after which all \( k^2 \) mice will immediately smell it and run to the cheese as fast as they can. There is enough cheese for all the mice, but not all cheese is the same: the mice that arrive first get to eat the best, stinkiest parts of the cheese. The problem: if two mice arrive at exactly the same time, they will fight over who gets to eat next best part of the cheese.

All mice run at exactly the same speed. All mice always take a shortest (minimum number of blocks) route, i.e. they never run North or East. But there are some small delays on each block, so paths with the same
number of blocks can take slightly different time - which you hope will help as tie-breakers between the mice racing for the cheese.

Your input is a pair of matrices: an \((k - 1) \times k\) matrix with delays going West from street-corner \((i, j)\) to street-corner \((i - 1, j)\), and an \(k \times (k - 1)\) matrix with delays going South from \((i, j)\) to \((i, j - 1)\). Design an algorithm that predicts ties, i.e. it finds a pair of mice that is expected to arrive at \((0, 0)\) at exactly the same time, or notifies that such a pair doesn’t exist.

**Input:** An \((k - 1) \times k\) matrix \(W\) with cost of going from \((i, j)\) to \((i - 1, j)\) and an \(k \times (k - 1)\) matrix \(S\) with cost of going from \((i, j)\) to \((i, j - 1)\).

**Output:** Either a pair \((i_1, j_1), (i_2, j_2)\) of mice that will arrive at exactly the same time, or \(\emptyset\) if no ties will occur.

**Example.**

**Input:**

\[
W = \begin{bmatrix}
101 & 105 \\
105 & 102 \\
101 & 105 \\
\end{bmatrix}
\]

\[
S = \begin{bmatrix}
103 & 105 & 105 \\
102 & 103 & 105 \\
\end{bmatrix}
\]

**Output:**

\[
[(1 , 2), (2 , 1)]
\]

In the diagram above, North, East, South, and West are up, right, down, and left respectively. The mice living at \((1, 2)\) and \((2, 1)\) will arrive at \((0, 0)\) at the same time (the delays they encounter are bolded). The algorithm will output \([(1, 2), (2, 1)]\)