Warmup

For each of the following shortest path problems, choose which algorithm you will use.

1. All-Pairs Shortest Path on a graph with (possibly negative) edge weights
2. All-Pairs Shortest Path on a graph with non-negative edge weights
3. Single Source Shortest Path on a graph with non-negative edges weights
4. Single Source Shortest Path on a graph with (possibly negative) edge weights

(1) Floyd-Warshall, (2) Dijkstra, (3) Dijkstra, (4) Bellman-Ford

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. They have an unlimited supply of this apple strudel, but each serving will still cost you $1.

The restaurant also has 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 cost 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 money as possible throughout the first \( k \) courses so that you have more money to spend on the unlimited dessert. Design an algorithm to find the smallest amount of money you can spend on the first \( k \) courses 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 complimentary leaf of spinach that costs 0 dollars).

Notice that this problem can be modeled as a shortest-path graph problem with costs on vertices rather than edges. However, we can transform this graph fairly easily into a directed weighted graph.

First we construct a directed graph in which each node is a dish and edges exist from \( v_i \) to \( v_j \) if you must finish dish \( i \) in order to order dish \( j \).
Next, we augment the edges with weights as follows: for all \( j \), for all \((v_i, v_j)\), the weight of \((v_i, v_j)\) is the price of \( v_j \).

Finally, we run Dijkstra’s algorithm as we saw in class to find the shortest (least cost) path from any dish in the first course to the strudel! An equivalent formulation of this algorithm is to skip the edge-weighting step and instead use a modified version of Dijkstra’s algorithm in which path lengths are given by the sum of the costs of the vertices along that path.

Aside: note that without the dependencies caveat, a naive greedy algorithm solves this problem.

Currency Exchange

Suppose the various economies of the world use a set of currencies \( C_1, \ldots, C_n \) — think of these as dollars, pounds, bitcoins, etc. Your bank allows you to trade each currency \( C_i \) for any other currency \( C_j \) at an exchange rate \( r_{ij} \), that is, you can exchange each unit of \( C_i \) for \( r_{ij} > 0 \) units of \( C_j \). Due to fluctuations in the markets, it is occasionally possible to find a sequence of exchanges that lets you start with currency A, change into currencies, B, C, D, etc., and then end up changing back to currency A in such a way that you end up with more money than you started with. That is, there are currencies \( C_{i_1}, \ldots, C_{i_k} \) such that

\[
r_{i_1 i_2} \times r_{i_2 i_3} \times \cdots \times r_{i_{k-1} i_k} \times r_{i_k i_1} > 1.
\]

This is called an arbitrage opportunity, but to take advantage of it you need to be able to identify it quickly (before other investors leverage it and the exchange rates balance out again)! Devise an efficient algorithm to determine whether an arbitrage opportunity exists. Justify the correctness of your algorithm and its runtime.

Build the complete directed graph with the currencies \( C_i \) as the vertices, and assign the edge \((C_i, C_j)\) weight \(-\log r_{ij}\). We run Floyd-Warshall on this graph, and conclude there is an arbitrage opportunity if and only if Floyd-Warshall detects a negative cycle.

**Correctness:** An arbitrage opportunity is equivalent to a negative cycle in this graph:

\[
\log r_{i_1 i_2} + \log r_{i_2 i_3} + \cdots + \log r_{i_{k-1} i_k} + \log r_{i_k i_1} > 0
\]

Thus Floyd-Warshall identifies a negative cycle if and only if there exists an arbitrage opportunity.

**Runtime:** Floyd-Warshall runs in time \( O(n^3) \) total. Creating the graph takes \( O(n^2) \) for the edges, making the total runtime \( O(n^3) \).

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 \( k \).

To solve this problem we are going to exploit the following overlapping sub-problems. If we let \( f(k) \), be the maximum product possible for a rod of length \( k \), then we have
Another way to think of this is that we are going to try cutting the rod of length \( k \) into two rods of length \( c \) and \( k-c \) and try all possible values of \( c \), taking the one which produces the maximum product. Note that not cutting the rod at all is another option which we can take. Also notice that we do not need to consider cutting off a length of 1 since that will never yield the optimal product, and also do not need to try cuts any larger than \( \lfloor k/2 \rfloor \) since those will already have been explored due to the symmetry of the cutting. The running time for this algorithm is \( O(k^2) \) since for each value of \( k \) we loop through \( O(k) \) values to get the answer for that \( k \).

```python
def max_rod_cut(k):
    # max_prods[i] := largest product for cutting rod of length i
    max_prods = [0 for _ in range(k + 1)]
    max_prods[1] = 1  # base case. length 1 cannot be cut more

    for i in range(2, k + 1):
        best_prod = i  # compare against not cutting at all
        for cut in range(2, i // 2 + 1):
            remaining = i - cut  # the length remaining
            p = max_prods[cut] * max_prods[remaining]
            best_prod = max(best_prod, p)

        max_prods[i] = best_prod

    return max_prods[k]
```

\[
f(k) = \max_{c \in \{2, \ldots, k-1\}} (k, c \cdot f(k-c))
\]