# Solutions to Homework 4

This homework has 8 questions, for a total of 100 points and 5 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on Piazza, and we do not guarantee anything about the difficulty of these questions.
We strongly encourage you to typeset your solutions in LaTeX.
If you collaborated with someone, you must state their name(s). You must *write your own solution* for all problems and *may not use any other student's write-up*.

(0 pts) 0. **Before you start; before you submit.**

(a) If you haven't already, carefully read Handout 3 about "giving an algorithm," and follow it in your solutions.

(b) If you need a quick refresher on *graph terminologies*, check out Handout 4.

(c) If applicable, state the name(s) and uniqname(s) of your collaborator(s).

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't do last week's homework, choose a problem from it that looks challenging to you, and in a few sentences, explain the key ideas behind its solution in your own words.

**Solution:** Both my answers for 2d and 2e lost points because my proof for correctness and runtime analysis were not clear enough. My solution for 2e is as follows:

Using this 2x2 matrix, the algorithm from part (d) without any modulus operations would run in $O(\log n)$ time and compute the kth Fibonacci number. As you multiply the matrix by itself k times, the resultant matrix has the kth Fibonacci number in the first row, second column, or the first column, second row.

A more concrete solution would have included correctness and runtime analysis like how it is included on the solution sheet. It would have also benefitted me to show mathematically what the matrices look like as shown on the solution sheet.

2. **Seems Familiar?**

In this course, pattern matching is a crucial skill. Many problems you will see on homeworks and exams are, at their cores, similar to those you have previously solved in lecture or discussion. Noticing these similarities allows you to use familiar strategies and techniques to solve new problems efficiently.

In this problem, you are given 3 dynamic programming problems below. For each of the following parts, you are given another problem that is the same as one of aforementioned problems and can be solved with same recurrence.

For each part, state which problem it is the same as, and briefly justify why the two problems are reformulations of the same problem. Explain how the constraints of the problem and the value(s) to be optimized match.

- **0-1 Knapsack Problem:** Given lists of the values, $V = [v_1, \ldots, v_n]$, and weights of items, $W = [w_1, \ldots, w_n]$, determine the maximum possible value of unique items you can select with a total weight less than $C$.
- **Knights Walk:** Given an $n$-by-$n$ chess board, and a knight that starts in the top left square, $(1, 1)$, find the number of unique ways can the knight reach the bottom right square, $(n, n)$. Note that the knight must move in a way such that its distance to $(n, n)$ decreases every move. For a knight at position $(i, j)$, its distance to $(n,\ n)$ is $d = (n - i, n - j)$.
- **Coin Change Problem:** Given a finite list of coins $c = [c_1, \ldots, c_k]$, where $c_i$ is the value of coin $i$, and a target amount $n$, find the number of unique collections of coins that have a total value of $n$.

(5 pts) (a) Given a multiset[1] of elements, $U$, find the number of different submultisets of $U$ sum up to a given target $t$.

> **Solution:** This is the same as the coin change problem. In this problem, each element in $U$ is a coin, and the target sum $t$ is the same as the target value $n$ in the coin change problem. Each element in $U$ can be used only as many times as it appears in $U$, which is the same as the coin change problem where each coin can only be used as many times as it appears in the list of coins. Both problems are tallying the number of ways to sum to a target value using a specific set of elements.

(5 pts) (b) Given a pipe of length $L$, and a list of pipe lengths with associated prices, find the maximum revenue obtainable by cutting the rod. You may only sell each length of a maximum of one time.

> **Solution:** This is the same as the 0-1 knapsack problem. In this problem, the prices are the values, the pipe lengths are the weights, and the rod length $L$ is the knapsack capacity. The goal is to maximize value by selecting the pipe lengths such that the total length is less than or equal to $L$, while the goal in the 0-1 knapsack problem is to maximize value by selecting items such that the total weight is less than or equal to $C$. In both problems, each available element (pipe length or item) can be used only once, and the total value cannot be greater than a certain limit.

[1] A *multiset* $U$ is just like a set, but it can have duplicate elements. A submultiset $I \subseteq U$ also can have duplicate elements, as long as it does not have more copies of an particular element than $U$ does.

3. **Oops!...ilon needs your help again.**

Upsilon ($\Upsilon$) is fascinated by the idea of dynamic programming (DP) that you learned this week! They are excited to apply this technique, but their recurrence relations[2] for the DP formulation often raise questions. Your task is to help them fix their questionable recurrence relation in each of the following problem.

(8 pts) (a) Given a set $S = \{s_1, \ldots, s_n\}$ of $n$ positive integers, we aim to partition $S$ into two subsets such that the sums of elements in both subsets are equal. Formally, we want to determine if there is a partition $(S', S \setminus S')$ of $S$ such that

$$\sum_{x \in S'} x = \sum_{y \in S \setminus S'} y.$$

$\Upsilon$ makes the following clever observation: Having two disjoint subsets **that partition the set and have equal sum** is equivalent to having one subset of $S$ that sums to $H = (\sum_{s \in S} s)/2$ (computed once at the beginning of the problem). In light of that observation, $\Upsilon$ proposes the following recurrence relation: let $P(i)$ represent whether there is a subset of $\{s_1, \ldots, s_i\}$ that sums to $H$. The recurrence is

$$P(i) = \begin{cases} \text{false} & \text{if } i = 1 \text{ and } s_i \neq H \\ \text{true} & \text{if } \sum_{k=1}^{i} s_k = H \\ P(i-1) & \text{otherwise.} \end{cases}$$

$\Upsilon$'s intuition is as follows: if there is a $j \leq i$ such that $\{s_1, \ldots, s_j\}$ sums to $H$, then $P(i)$ is automatically true because adding more elements from $\{s_{j+1}, \ldots, s_i\}$ cannot invalidate the existence of that subset. The answer to whether it's possible to partition $S = \{s_1, \ldots, s_n\}$ into two subsets of equal sum is given by $P(n)$.

(i) Explain the flaws in $\Upsilon$'s recurrence relation. You may use an example to support your reasoning.

(ii) It turns out that we need a 2-dimensional recurrence relation for this problem. Propose, with justification, a recurrence relation (including base cases) that is suitable for a dynamic programming solution to the problem.

*Hint*: Some problems from HW3 may provide inspiration (so read the solution!). You may cite the results without re-proving them.

[2] We stress that a recurrence relation is the key ingredient in a DP formulation. Always make sure you get it right before implementing the algorithm!

**Solution:**

(i)
- Upsilon's recurrence assumes that if $\sum_{k=1}^{i} s_k$ sums to $H$, the problem is solved. This is incorrect, because valid subsets do not necessarily have to be contiguous. Consider $S = \{1, 2, 4, 5\}$, where $H = 6$. The subset $\{1, 5\}$ sums to 6, but no contiguous subset sums to 6.
- Upsilon's recurrence only checks the sum of the full subset up to $s_i$ instead of deciding whether (or not) to include $s_i$ in forming $H$.
- Upsilon's recurrence also does not handle the standard base case of the empty set summing to 0.

(ii) Let $P(i, j)$ represent whether there exists a subset of $\{s_1, s_2, \ldots, s_i\}$ that sums to $j$. The recurrence can be modeled as follows:

$$P(i, j) = \text{true if there exists a subset of } \{s_1, s_2, \ldots, s_i\} \text{ that sums to } j.$$

Base cases:

$$P(0, 0) = \text{true}, \quad P(0, j) = \text{false for all } j > 0.$$

Recurrence:

$$P(i, j) = \begin{cases} P(i-1, j) \vee P(i-1, j-s_i) & \text{if } j \geq s_i, \\ P(i-1, j) & \text{if } j < s_i, \\ true & \text{if } i = 0 \text{ and } j = 0, \\ false & \text{if } i = 0 \text{ and } j \neq 0. \end{cases}$$

If there is a subset of $\{s_1, s_2, \ldots, s_{i-1}\}$ that sums to $j$ ($P(i-1, j)$ is true), then $P(i, j)$ is also true because we can exclude $s_i$ from the subset. Similarly, if $j < s_i$, then $s_i$ cannot be included in the subset, so $P(i, j)$ is the same as $P(i-1, j)$. If there is a subset of $\{s_1, s_2, \ldots, s_{i-1}\}$ that sums to $j - s_i$ ($P(i-1, j-s_i)$ is true), then $P(i, j)$ is also true because we can then include $s_i$ in the subset.

(10 pts) (b) Given an array $A[1, \ldots, n]$ of $n \geq 1$ *positive real numbers* (represented as floating-point numbers that take constant time to operate on), we are interested in finding the smallest *product* of any **contiguous** non-empty subarray of $A$, i.e.,

$$\min\{A[i]A[i+1] \cdots A[j] : i \leq j \text{ are indices of } A\}.$$

Given what happened in the previous problem, $\Upsilon$ is now thinking of a 2-dimensional solution to this problem. They propose the following recurrence relation: let $S(i, j)$ be the smallest subarray product in the subarray $A[i, \ldots, j]$. The recurrence is

$$S(i, j) = \begin{cases} A[i] & \text{if } j \leq i \\ \min\{A[i], A[j]\} & \text{if } S(i+1, j) \geq 1 \text{ or } S(i, j-1) \geq 1 \\ \min\{A[i] \cdot S(i+1, j), A[j] \cdot S(i, j-1)\} & \text{otherwise.} \end{cases}$$

$\Upsilon$'s intuition is as follows: if $S(i+1, j) \geq 1$ or $S(i, j-1) \geq 1$, it's better to just start the subarray at either $A[i]$ or $A[j]$, depending on which is smaller. Otherwise, the smallest subarray product in $A[i, \ldots, j]$ is achievable by either (1) appending $A[i]$ to the subarray considered in $A[i+1, j]$ or (2) appending $A[j]$ to the subarray considered in $A[i, j-1]$. The smallest subarray product in $A[1, ..., n]$ is given by $S(1, n)$.

(i) Explain the flaws in $\Upsilon$'s recurrence relation. You may use an example to support your reasoning.

(ii) It turns out that a 1-dimensional solution suffices for this problem. Propose, with justification, a recurrence relation (including base cases) that is suitable for a dynamic programming solution to the problem.

**Solution:**

(i)
- As stated, a one-dimensional approach is appropriate here. The problem only depends on the previous contiguous subarray ending at index $j$, not on the subarray starting at index $i$.
- The condition $S(i+1, j) \geq 1$ or $S(i, j-1) \geq 1$ fails to consider whether extending the previous minimum product subarray provides a smaller result than starting a new subarray.

(ii) Let the smallest subarray product ending at index $j$ be $F(j)$. The recurrence can be modeled as follows:

$$F(j) = \min\{A[i] \cdot A[i+1] \cdots A[j] : 1 \leq i \leq j\}.$$

Base case:

$$F(1) = A[1].$$

Recurrence:

$$F(j) = \min\{A[j], A[j] \cdot F(j-1)\}.$$

4. **Competing for rubber ducks.**

Lambda and Waquackquack are competing in a strategic game to win rubber debugging ducks for their friends. The ducks are laid out in a *row* of separate *piles*, which have various numbers of ducks in them. The two players alternate turns, taking (and removing) one entire pile of ducks *from either end of the remaining row* per turn. The game concludes once all piles are taken. The goal is to collect the maximum total number of rubber ducks.

Without loss of generality, suppose Lambda is the first player and *both* players play *optimally*. We model the problem as follows. There are initially $n$ piles of ducks, and there is an array $D[1, \ldots, n]$ where $D[i]$ is the number of ducks in the $i$th pile (from the left). We wish to compute the maximum number of ducks that Lambda can be *guaranteed* to collect in the game, regardless of how Waquackquack plays.

(4 pts) (a) Suppose $D = [4, 9, 1, 3]$. Determine the maximum number of ducks Lambda can be guaranteed to collect. Briefly explain Lambda's strategy.

> **Solution: 12.** Lambda's strategy is to first take the 3 ducks from the right end. Then, no matter what Waquackquack does, Lambda can take the 9 ducks from the left or right end.

(8 pts) (b) Give, with justification, a recurrence relation (including base cases) that is suitable for an $O(n^2)$ dynamic-programming solution to the problem. Start your solution by defining the recurrence relation in English. You will give and analyze the algorithm in Part c.

> **Solution:** Let P(i,j) be the maximum number of ducks Lambda can be guaranteed to collect from piles $i$ to $j$.
> Base case: P(i,i) = D[i].
> When there is more than one pile, a player has two options, either take the leftmost pile or the rightmost pile:
>
> - **leftmost:** The player collects $D[i]$. Their net lead from this choice is $D[i] - P(i+1, j)$, as the other player will take the maximum number of ducks from the remaining piles.
> - **rightmost:** The player collects $D[j]$. Their net lead from this choice is $D[j] - P(i, j-1)$.
>
> Thus, $P(i, j)$ can be modeled as:
>
> $$P(i, j) = max(D[i] - P(i+1, j), D[j] - P(i, j-1)).$$
>
> Note that the first player is guaranteed to collect $\frac{T+P(1,n)}{2}$ ducks, where $T$ is the total number of ducks.

(8 pts) (c) Give a (bottom-up) dynamic-programming algorithm that computes the maximum number of ducks that Lambda can be *guaranteed* to collect in the game, regardless of how Waquackquack plays.

**Solution:**

**Input:** Array $D[1 \dots n]$ where $D[i]$ is the number of ducks in the $i$th pile.
**Output:** Maximum number of ducks Lambda can guarantee.

**function** ComputeLambdaDucks($D[1 \dots n]$)
    Let $memo[1 \dots n][1 \dots n]$ be an $n$ x $n$ 2D array table for memoization.
    **for** $i = 1$ to $n$ **do**
        $memo[i][i] = D[i]$ // base case
    **for** length $= 2$ to $n$ **do**
        **for** $i = 1$ to $n - \text{length} + 1$ **do**
            $j = i + \text{length} - 1$
            $memo[i][j] = \max\{\, D[i] - memo[i+1][j],\ D[j] - memo[i][j-1] \,\}$
    $total = D[1] + \cdots + D[n]$
    **return** $\dfrac{total + memo[1][n]}{2}$

5. **Dynamic programming for dynamic shortest paths.**

Graphs that arise in real applications are often not fixed, but can change over time; such graphs are called "dynamic." (This is a totally different use of the word "dynamic" than in "dynamic programming". Computer scientists sometimes name things in confusing ways!) For example, roads and intersections can be added to or deleted from the road network, computers can be added to or removed from the Internet, and people can (un)follow each other on social networks. For many problems of interest, there are algorithms for dynamic graphs that are faster than just re-computing answers "from scratch" whenever the graph changes. In this problem you will give such algorithms for shortest-path problems.

Let $G = (V, E)$ be a weighted directed graph with $n = |V|$ vertices, where the weight of each edge $(u, v)$ is denoted $w(u, v)$. (There can be negative-weight edges, but there is no negative-weight cycle in $G$.) Suppose that we have already computed the all-pairs distance table $D$ of $G$. That is, for each vertex pair $u, v \in V$, entry $D(u, v)$ stores the distance (i.e., the length of a shortest path) from $u$ to $v$ in $G$.

Now, a new vertex $v_{new}$ is added to the graph, together with its incident edges. Let $G_{new} = (V \cup \{v_{new}\}, E \cup E_{new})$ denote the updated graph, where $E_{new}$ consists of all the incoming and outgoing edges for $v_{new}$. (There is no negative-weight cycle in $G_{new}$.) We aim to compute the updated distance table $D_{new}$ of $G_{new}$.

A straightforward approach is to compute $D_{new}$ from scratch using the Floyd-Warshall algorithm, in $\Theta(n^3)$ time. But we want to do better by exploiting the fact that we already have $D$. In this problem, you will obtain a faster $O(n^2)$-time algorithm.

(4 pts) (a) Consider a weighted directed graph $G$ with four vertices $a$, $b$, $c$, and $d$. The table below shows the shortest distance from vertex $u$ to $v$ computed using the Floyd-Warshall algorithm for all pairs of vertices in the graph.

| $u$ \ $v$ | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|
| $a$ | 0 | 3 | 7 | 5 |
| $b$ | 2 | 0 | 6 | 4 |
| $c$ | 3 | 1 | 0 | 5 |
| $d$ | 5 | 3 | 2 | 0 |

Now suppose a new vertex $v_{new}$ is added to the graph, along with four new edges with the following weights:

| Edge | Weight |
|---|---|
| $(v_{new}, a)$ | 1 |
| $(v_{new}, b)$ | 4 |
| $(v_{new}, c)$ | 2 |
| $(d, v_{new})$ | 1 |

Determine the shortest distance from $v_{new}$ to vertices $a$, $b$, $c$, and $d$. Show your calculation.

**Solution:** The distance between $v_{new}$ and each vertex is the minimum of the possible paths from $v_{new}$ to the vertex.

$$\begin{aligned} D_{new}(v_{new}, a) &= \min\{1, 2+3, 4+2\} = \min\{1, 5, 6\} = 1 \\ D_{new}(v_{new}, b) &= \min\{4, 1+3, 2+1\} = \min\{4, 4, 3\} = 3 \\ D_{new}(v_{new}, c) &= \min\{2, 1+3, 4+1\} = \min\{2, 4, 5\} = 2 \\ D_{new}(v_{new}, d) &= \min\{1+5, 4+4, 2+5\} = \min\{6, 8, 7\} = 6 \end{aligned}$$

(10 pts) (b) Write expressions for $D_{new}(v_{new}, u)$ and $D_{new}(u, v_{new})$ that hold for all $u \in V$, where the expressions are in terms of the already-known quantities $D(y, z)$ for $y, z \in V$, and $w(y, z)$ for $(y, z) \in E \cup E_{new}$. Evaluating your expressions should take $O(n)$ time for each vertex $u$, for a total of $O(n^2)$ time. Justify the correctness of your expressions and the evaluation time.

*Hint*: Consider why the Bellman-Ford algorithm is correct.

**Solution:**

- Any path from $v_{new}$ to $u$ must leave $v_{new}$ through an outgoing edge:

$$(v_{new}, y), y \in V$$

- Any path from $u$ to $v_{new}$ must enter $v_{new}$ through an incoming edge:

$$(z, v_{new}), z \in V$$

Thus,

$$D_{new}(v_{new}, u) = \min_{x \in V}\{\, w(v_{new}, y) + D(y, u)\}$$

and

$$D_{new}(u, v_{new}) = \min_{x \in V}\{\, D(u, z) + w(z, v_{new})\}$$

**Correctness:** Every path from $v_{new}$ to $u$ **must** begin with a new edge, and every path from $u$ to $v_{new}$ **must** end with a new edge. Then, each path continues along a distance already correctly given by $D$.
**Runtime:** For each vertex $u$, we need to consider all possible paths from $v_{new}$ to $u$ and $u$ to $v_{new}$, which takes $O(n)$ time for each vertex. Thus, evaluating the expressions for all vertices takes $O(n^2)$ time.

(6 pts) (c) Write an expression for $D_{new}(u, v)$ that holds for all $u, v \in V$, where the expression is in terms of the already-known quantities listed in the previous part, as well as the quantities you computed in Part b. Evaluating your expression for all $u, v \in V$ should take a total of $O(n^2)$ time. Justify the correctness of your expression and the evaluation time.

Observe that by combining the two parts, we have computed the entire new distance table $D_{new}$ of $G_{new}$ in $O(n^2)$ time!

*Hint*: While not required, you may want to compute $D_{new}$ for the example in Part a to get some intuition. Also consider why the Floyd-Warshall algorithm is correct.

**Solution:** Any possible shortest path in $G_{new}$ from $u$ to $v$ for $u, v \in V$ either:

- does not use $v_{new}$, so is the same as $G$, or $D(u, v)$,
- enters $v_{new}$ through an incoming edge $(y, v_{new})$ and continues along a path from $v_{new}$ to $u$, with distance given by $D_{new}(u, v_{new}) + D_{new}(v_{new}, v)$, both quantities calculated in part (b).

Thus, for graph $G_{new}$, the shortest distance from $u$ to $v$ is given by:

$$D_{new}(u, v) = \min\{D(u, v), D_{new}(u, v_{new}) + D_{new}(v_{new}, v)\}$$

**Correctness:** Every path from $v$ to $u$ either avoids $v_{new}$ and has length given by $D(u, v)$, or includes $v_{new}$ and has length given by $D_{new}(u, v_{new}) + D_{new}(v_{new}, v)$. Both outcomes give the shortest distance incorporating $v_{new}$
**Runtime:** The minimum of two numbers ($O(1)$) is computed for each pair of vertices ($O(n^2)$), so the total runtime is $O(n^2)$.

6. **Maximizing money made in Manhattan.**

To manage the surge in New Year's Eve tourism, the New York City government has implemented traffic control measures: all roads have been made one-way toward Times Square; to prevent accumulating traffic, all intersections are reachable and there are no loops or dead ends (except at Times Square); and all tourists must use rideshares. As the midnight countdown approaches, tourists at various intersections are "bidding" on rides to Times Square to witness the iconic Ball Drop, by posting the prices they are willing to pay for rides.

Vikram, a limousine driver, will drive through the city to Times Square, picking up all the tourists at each intersection along the way. (His limousine has unlimited passenger capacity and fuel.) Vikram's goal is to find a route to Times Square that maximizes the total money he earns.

We model this problem mathematically as follows. The city's road network is given by a directed acyclic graph (DAG) $G = (V, E)$ with $n$ vertices given in topological order[3] as $V = \{1, \ldots, n\}$ (representing intersections), and $m$ directed edges $(i, j) \in E$ where $i < j$ (representing road segments). Every vertex $i < n$ has at least one outgoing edge (there is only one dead end), and every vertex $j > 1$ has at least one incoming edge (all intersections are reachable). For each vertex $i$ there is an associated price $p_i \geq 0$ that the tourists at intersection $i$ are willing to pay (in total). Vikram wishes to find a route from vertex 1 (his starting location) to vertex $n$ (Times Square) that maximizes the total price along the route.

(4 pts) (a) Briefly but clearly describe a brute-force algorithm for this problem, which may run in exponential time in the number of vertices $n$. You do not need to give pseudocode, justify correctness, or analyze the running time.

> **Solution:** A brute-force algorithm for this problem could start with a depth-first search starting at vertex 1. Every time the search reaches vertex $n$, compute the total money earned by summing the prices on the vertices along that route. Once this is done, return the maximum total money computed over all such routes.

(8 pts) (b) Give, with justification and base case(s), a recurrence relation that is suitable for a dynamic programming solution to this problem.

*Hint:* A recurrence relation is a *mathematical object*, which should be about the "value version" of the problem. Start your solution with a clear English description of what the recurrence relation represents.

[3] A *topological ordering* for a DAG $G = (V, E)$ is a linear ordering of vertices such that for every directed edge $(u, v)$, vertex $u$ comes before $v$ in the ordering.

**Solution:** Let $M(j)$ be the maximum total money that can be earned on a route from vertex 1 to vertex $j$.
To compute $M(j)$ add the price $p_j$ for vertex $j$ to the maximum over all routes ending at an immediate predecessor $i$ (where $(i, j) \in E$). Since the vertices are given in a topological order, every $M(i)$ for $i < j$ is already computed.
Base case:

$$M(1) = p_1,$$

For every vertex $j > 1$:

$$M(j) = p_j + \max_{(i,j)\in E} M(i).$$

(8 pts) (c) Give a (bottom-up) dynamic programming algorithm, including pseudocode, that solves the "value version" of this problem in $O(n + m)$ time. (The "value version" is to find the maximum money that Vikram can earn, not necessarily a route that obtains it.) You may assume that for each $j \in V$, you are given a list of all its incoming edges $(i, j) \in E$.

**Solution:**

**Input:** A directed, acyclic graph $G = (V, E)$ with vertices in topological order, and an array $p[1 \dots n]$ where $p[i]$ is the price at vertex $i$.
**Output:** The maximum Vikram can earn from vertex 1 to vertex $n$.

```
1:  function MaximizeMoney(G = (V, E), p[1...n])
2:      Let M[1...n] be the maximum income at each vertex.
3:      for j = 1 to n do
4:          M[j] = −∞
5:      M[1] = p[1] // base case
6:      for j = 2 to n do
7:          max_val = −∞
8:          for each (i, j) ∈ E do
9:              if M[i] > max_val then
10:                 max_val = M[i]
11:         M[j] = p[j] + max_val
12:     return M[n]
```

(2 pts) (d) Briefly describe in words (no pseudocode or correctness/runtime analysis needed) how to extend the above algorithm to output an optimum route.

**Solution:** For each vertex $j$, store a pointer $prev[j]$ indicating the vertex $i$ that achieved the maximum value at $j$. After all vertices are resolved, start from vertex $n$ (Times Square) and repeatedly follow the $prev$ pointers to backtrack to vertex 1. This sequence in reverse order gives the optimum route from vertex 1 to Times Square (vertex $n$) to maximize the money earned.

7. **Rick's genetic mashup (optional extra credit).**

*Even though this is an extra-credit problem that will be graded with high standards and not much partial credit, we highly encourage you to at least brainstorm about Part a.*

Rick has just stumbled upon two ancient DNA fragments from a long-extinct alien species[4]. To bring it back to life (for *science*, not for some messed-up interdimensional scheme—probably), he needs to reconstruct its full genome. However, DNA synthesis is expensive, and Morty keeps whining about how they're low on budget.

To minimize costs, Rick wants the shortest possible DNA strand that still contains both fragments as subsequences *in order*. For example, to merge "**376cse**" and "370eecs", "**37**0**6**ee**cse**" is allowed but "**37**0**6cse**e" is not. In this problem, you will use dynamic programming to help them with this task.

(2 EC pts) (a) Suppose there are two fragments of DNA sequences of length $m$ and $n$. Either give, with justification and base case(s), a recurrence relation, that is suitable for an $O(mn)$ dynamic programming solution to merge the two fragments as the shortest sequence possible, or explain how a recurrence relation from lecture, discussion, or HW3 can be adapted for this problem. You will give and analyze the algorithm in Part b.

**Solution:**

(3 EC pts) (b) Morty, still complaining about the cost, points out that even running a fancy algorithm on their computer is expensive. Frustrated but somewhat agreeing, Rick challenges him to optimize the approach so that they can run it on a *bootleg Meeseeks Box wired to a sentient butter-passing robot* with extremely limited RAM. Instead of using $O(mn)$ space[5], help Morty to design an $O(mn)$ (bottom-up) dynamic programming algorithm that computes the *length* of the shortest merged sequence using only $O(\max\{m, n\})$ space, where $m$ and $n$ are the lengths of the two DNA fragments.

Then, either briefly describe how you can extend your algorithm to output an actual shortest merged sequence (if there are multiple, output *any* one of them), or briefly explain why it's impossible to do so under the space constraint.

**Solution:**

[4]The genomes of this species are not limited to A, C, T, and G like humans; instead, it can consist of any symbols Rick chooses to use.

[5]Space complexity refers to the total memory required by an algorithm or program. In the case of a bottom-up dynamic programming algorithm, this typically includes the size of the table and some additional overhead. This is the only problem on this assignment where you have to talk about space complexity.