EECS 376: Foundations of Computer Science
University of Michigan, Winter 2024

Solutions to Homework 4

This homework has 8 questions, for a total of 100 points and 8 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.**

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

Solution:

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

Carefully read and understand the posted solutions to the previous homework; you may also find the video "walkthroughs" on Canvas helpful. 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.)

Solution:

2. **Keeping things fresh with potpourri.**

(3 pts) (a) Consider the following function, which takes as input non-negative integers a and b that are powers of two.

```
1: function ALG($a$,$b$)
2:     if $a = 1$ or $b = 1$ or $a = b$ then return 0
3:     if $a > b$ then return ALG($a/2, 2b$)
4:     if $b > a$ then return ALG($2a, b/2$)
```

Either find a valid potential function to prove that ALG halts on all valid inputs a, b, or show that no such function exists by giving an input on which ALG runs forever.

Solution: The algorithm does not terminate on input $(4, 8)$, nor on many other inputs. So, no such potential function exists (because its existence would imply that the algorithm halts on all valid inputs).

(3 pts) (b) Let $T(n)$ be the running time of the following function on an array of n entries. Write a recurrence for $T(n)$, and *briefly* justify your answer.

```
function Foo($A[1, \dots, n]$)
    if $n = 1$ then return $A[1] + 1$
    $x = 2 \cdot$ Foo($A[1, \dots, n/2]$)
    $y = 3 \cdot$ Foo($A[n/2 + 1, \dots, n]$)
    $tmp = 0$
    for $i = 1, \dots, n$ do
        for $j = 1, \dots, n$ do
            $tmp \leftarrow tmp + A[i] + A[j] - x + y$
    return $tmp$
```

Solution: The recurrence is $T(n) = 2 \cdot T(n/2) + \Theta(n^2)$.
The function is called twice, each time on an array of size $n/2$, on lines 3 and 4. And the nested loop takes $\Theta(n^2)$ time to run.

(4 pts) (c) Suppose there are algorithms A and B, where $T_A(n)$ and $T_B(n)$ are respectively the worst-case running times of A and B on inputs of size n. Suppose that T_A satisfies $T_A(n) = 2T_A(n/2) + T_B(n)$, and $T_B(n) = O(n^2)$ but $T_B(n) \neq O(1)$. Select all bounds for $T_A(n)$ that *could possibly hold* (for some choice of $T_B(n)$ satisfying the above bounds):

A. $\Theta(n^3)$
B. $\Theta(n^2)$
C. $\Theta(n \log n)$
D. $\Theta(n)$
E. $\Theta(1)$

Choose one answer you selected as possible, and give a specific choice of $T_B(n)$ that would yield that result. You do not need to provide any justifications, other than why your choice of $T_B(n)$ yields the claimed solution for $T_A(n)$.

Solution: We have $k = 2, b = 2$, and $d = 2$ (because $T_B(n) = O(n^2)$), so by the Master Theorem, $T_A(n) = O(n^2)$. Note that this is just an upper bound, and may not be tight. However, it implies that option A is not possible, because a function cannot be bounded by both $\Omega(n^3)$ and $O(n^2)$. Option E is also not possible, because if $T_A(n) = O(1)$, then because $T_B(n) \leq T_A(n)$, we would also have $T_B(n) = O(1)$, which is disallowed.
Options B, C, or D are possible.
If $T_B(n) = \Theta(n^2)$, then $k/b^d < 1$, so $T_A(n) = \Theta(n^2)$ by the Master Theorem.
If $T_B(n) = \Theta(n)$, then $k/b^d = 1$, so $T_A(n) = \Theta(n \log n)$.
If $T_B(n) = \Theta(\log n)$, the $k/b^d > 1$, so $T_A(n) = \Theta(n)$.

3. **Counting certain cupcakes.**

Lily has started a new job at Bakehouse 46 in Downtown Ann Arbor! Her responsibilities in opening up the store everyday include placing the store's cupcakes in the display case. The cafe makes several different flavors of cupcakes, and various numbers of each flavor. The manager requires that for each flavor, either *all* or *none* of the cupcakes of that flavor are displayed. Lily wants to determine whether it is possible to *completely* fill the display case with cupcakes while following this rule.

We formalize the problem as follows. We are given an array of non-negative integers $S[1, \dots, n]$, where each $S[i]$ represents the number of cupcakes of the ith flavor, and a non-negative integer K representing the total number of cupcakes the display case can hold. We wish to determine whether there is a set of flavors that have exactly K cupcakes in total.

Given the array S, for suitable integers i, m (in ranges for you to determine) define $D(i, m)$ to be the Boolean value indicating whether there a subset of the first i flavors having exactly m cupcakes.

(10 pts) (a) Derive a recurrence relation, including base case(s), for $D(i, m)$ that is suitable for a dynamic-programming solution to this problem. Briefly justify its correctness.

Solution: The base cases are:

- For all $i \geq 0$, $D(i, 0)$ is true: we can take the empty set of cupcake flavors, which has exactly zero cupcakes.
- For all $m \neq 0$ (including negative values), $D(0, m)$ is false: there are zero cupcakes available among the first $i = 0$ flavors, so we cannot get a nonzero number of cupcakes.
- For all $i \geq 0$ and $m < 0$, $D(i, m)$ is false: we cannot get a negative number of cupcakes, no matter which flavors of cupcakes we are allowed to use.

We now derive a recurrence for $D(i, m)$. There are only two possible ways there can be a subset of the first i flavors that has exactly m cupcakes: either flavor i is used, and there is a subset of the first $i - 1$ flavors that has exactly $m - S[i]$ cupcakes; or flavor i is not used, and there is a subset of the first $i - 1$ flavors that has exactly m cupcakes. So, we have

$$D(i, m) = D(i - 1, m - S[i]) \vee D(i - 1, m).$$

Note that $m - S[i]$ might be negative, but this is covered by the final base case above.

(5 pts) (b) Give pseudocode for a (bottom-up) dynamic programming algorithm that solves this problem, and analyze its running time.

Solution: We now will use the recurrence from the previous part to implement a bottom-up solution in pseudocode. Note that we compute $D(i, m)$ for all values of $0 \leq i \leq n$ and $0 \leq m \leq K$ using a nested for loop and logical or operators. Each cell in the table represents whether we can fill a display of size m with the first i flavors. It is also important to note that in line 7 when we get a negative value for $j - S[i]$, we implicitly yield false for that term in the or statement.

```
Input: array $S[1,\dots,n]$ of non-negative integers, and non-negative integer $K$
1: function CupcakeDisplay($S[1,\dots,n], K$)
2:     allocate $D[0,\dots,n][0\dots,K]$
3:     $D[0,\dots,n][0] \leftarrow true$
4:     $D[0][1,\dots,K] \leftarrow false$
5:     for $i = 1,\dots,n$ do
6:         for $j = 1,\dots,K$ do
7:             $D[i][j] \leftarrow D[i-1][j-S[i]] \lor D[i-1][j]$
8:     return $D[n][K]$
```

This algorithm has a running time of $O(nK)$. Allocating D takes $O(nK)$ operations. The pair of for loops take $O(nK)$ time, because each innermost iteration takes constant time. All other operations are constant time.

4. **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 $\ell(u, v)$. (There is no negative-weight cycle in G.) Suppose that we have already computed the all-pairs distance table D_G of G. That is, for each vertex pair $u, v \in V$, $D_G(u, v)$ 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 $D_{G_{new}}$ of G_{new}.

A straightforward approach is to compute $D_{G_{new}}$ 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_G. In this problem, you will obtain a faster $O(n^2)$-time algorithm.

(20 pts) (a) Write expressions for $D_{G_{new}}(v_{new}, u)$ and $D_{G_{new}}(u, v_{new})$ that hold for all $u \in V$, where the expressions are in terms of the already-known quantities $D_G(y, z)$ for $y, z \in V$, and $\ell(y, z)$ for $y, z \in V \cup \{v_{new}\}$. 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: To compute $D_{G_{new}}(v_{new}, u)$ for all $u \in V$, the algorithm is as follows. For each $u \in V$, set

$$D_{G_{new}}(v_{new}, u) \leftarrow \min_{(v_{new}, w) \in E_{new}} \ell(v_{new}, w) + D_G(w, u).$$

The running time is $O(n^2)$ because we loop over all vertices $u \in V$, and each loop minimizes over at most n edges.

For correctness, recall that any shortest path P from v_{new} to u in G_{new} must be a concatenation of some edge $(v_{new}, w) \in E_{new}$ followed by a shortest path P' from w to u in G_{new}. We claim that P' is actually a shortest path from w to u in the original graph G. This is because there is no negative-weight cycle, so without loss of generality, P' will not visit v_{new}. So P' uses only original edges in G and, hence, is indeed a shortest path in G. Since P must start with one of the edges $(v_{new}, w) \in E_{new}$, we have

$$D_{G_{new}}(v_{new}, u) = \min_{(v_{new}, w) \in E_{new}} \ell(v_{new}, w) + D_G(w, u),$$

which is what the above algorithm uses.

To compute $D_{G_{new}}(u, v_{new})$ for all $u \in V$, the algorithm is symmetric: for each $u \in V$, set

$$D_{G_{new}}(u, v_{new}) \leftarrow \min_{(w, v_{new}) \in E_{new}} D_G(u, w) + \ell(w, v_{new}).$$

The running time and correctness analysis are symmetric too.

(10 pts) (b) Write an expression for $D_{G_{new}}(u, v)$ 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 the previous part. 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 $D_{G_{new}}$ of G_{new} in $O(n^2)$ time.

Hint: Consider why the Floyd-Warshall algorithm is correct.

Solution: The algorithm is as follows: For all $u, v \in V$, set

$$D_{G_{new}}(u, v) \leftarrow \min\{D_G(u, v), D_{G_{new}}(u, v_{new}) + D_{G_{new}}(v_{new}, v)\}.$$

The running time is $O(n^2)$ because we loop over all $O(n^2)$ pairs of vertices and each loop takes $O(1)$ time. Observe that this corresponds to just one (more) iteration of the "outer loop" of Floyd-Warshall, where we are newly allowing v_{new} as an intermediate vertex on paths.

The correctness proof is also the same as that of the Floyd-Warshall algorithm. We know that a shortest path from u to v in G_{new} either contains v_{new} as an intermediate vertex, or not. If so, their distance is $D_{G_{new}}(u, v_{new}) + D_{G_{new}}(v_{new}, v)$. If not, the distance is $D_G(u, v)$ because G is exactly the graph G_{new} after ignoring v_{new}. Since a best one of the two cases must hold, we have

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

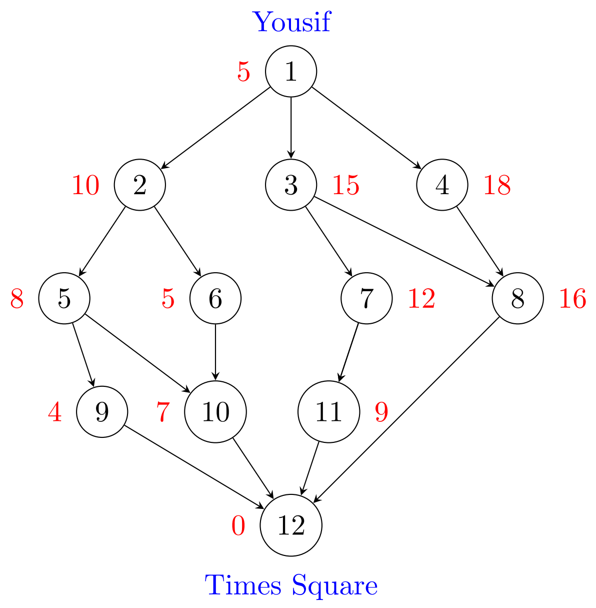
which is what the algorithm uses.

5. **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; 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 ride-shares. 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.

Yousif, 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.) Yousif's goal is to find a route to Times Square that maximizes the total money he earns.

We model this problem mathematically as follows (see the graph below for an example). The city's road network is given by a directed acyclic graph (DAG) $G = (V, E)$ with n vertices given in topological order 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). Yousif wishes to find a route from vertex 1 (his starting location) to vertex n (Times Square) that maximizes the total price along the route.



(2 pts) (a) Consider the following "greedy" algorithm for this problem: at each step, move to an adjacent vertex that has the maximum price, until arriving at Times Square. What total price will this algorithm yield for the above graph? (The price for each vertex is shown next to it, in red.) Is it optimal, and why or why not?

Solution: The "greedy" algorithm would produce the path $\textcircled{1} - \textcircled{4} - \textcircled{8} - \textcircled{12}$, yielding a total fare of 39. This is not optimal; we could instead follow $\textcircled{1} - \textcircled{3} - \textcircled{7} - \textcircled{11} - \textcircled{12}$ to get a total fare of 41.

(4 pts) (b) 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: Consider each subset $S \subset \{2, \ldots, n-1\}$ of vertices excluding 1 and n. There is at most one valid path from 1 to n that uses all the vertices in S, and no others, as internal nodes, because $i < j$ for every edge $(i, j) \in E$. So, for each such subset we check whether the corresponding path exists (by checking for all its edges), calculate how much money it earns, and return one of the best such paths. The running time is 2^{n-2} times the polynomial cost of checking each path; overall, this is exponential in n.

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

Solution: Let $P[j]$ be the maximum money Yousif can earn if his path ends at vertex j. The base case is $P[1] = p_1$, since Yousif picks up only the tourists at the starting vertex 1, which is the end of his path.

For any vertex $j > 1$ we have the guarantee that there exists at least one incoming edge. Consider any path p ending at vertex j that maximizes Yousif's earnings. Let (i, j) be the final edge of that path; then the subpath excluding this last edge must maximize Yousif's earnings if he were to end at vertex i. (For if not, Yousif could take a higher-earning path to i then the edge (i, j), which would earn more overall than p, thus contradicting the optimality of p.) Since the last edge must be one of the incoming edges to j, the optimum earnings must be given by

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

Alternative solution: Instead, we could define $P[i]$ to be the maximum money Yousif can earn starting at vertex i and ending at vertex n. The analysis proceeds symmetrically to the case above, with a base case of $P[n] = p_n$, and

$$P[i] = \max_{(i,j)\in E} (p_i + P[j]) = p_i + \max_{(i,j)\in E} P[j] \,.$$

Note that for this recurrence, the bottom-up pseudocode in the next part would instead iterate from $i = n$ down to 1, and would use lists of *outgoing* edges from each vertex. (These kinds of adjacency lists are standard graph representation.)

(7 pts) (d) 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 Yousif 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: Algorithm:

Input: DAG ($V = \{1, \ldots, n\}, E$) as above, and prices $p_i \geq 0$ for $1 \leq i \leq n$.
Output: Maximum total price over all routes from 1 to n

```
function MaxMoney($G$)
    Initialize $P[1, \ldots, n]$
    $P[1] = p_1$
    for $j = 2, \ldots, n$ do                          ▷ Topological order
        $P[j] = p_j + \max_{(i,j) \in E} P[i]$
    return $P[n]$
```

Correctness: Since the input graph is topologically sorted, every edge $(i, j) \in E$ has $i < j$. Since we iterate over the vertices from 2 to n, we can evaluate the recurrence by just looking up the values which would already be computed. Hence above pseudo-code implements the recurrence correctly.

Since vertex n corresponds to Times Square, $P[n]$ is the most money that Yousif can collect when his path ends at Times Square, which is the desired output value.

Running Time: to fill the entries of the array P, we iterate over each vertex, and over each incoming edge of that vertex, so in total we consider each vertex and edge exactly once. Because we do constant work for each, the total running time is $O(n + m)$.

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

Solution: For every node $j > 1$, when we compute and store the value of $P[j] = p_j + \max_{(i,j) \in E}$, we also store the value of an in-neighbor $i < j$ that yields this maximum value. At the end, we "backtrack" through these values, reconstructing a path (in reverse) starting from vertex n and ending at 1.

6. **Fixed-pattern paths.**

Let $G = (V, E)$ be a graph with n vertices and m edges. Also, suppose that each edge is colored either black or white. We say that a path $P = (e_1, \ldots, e_k)$ of edges has *color pattern* $(c_1, \ldots, c_k)$, where each $c_i \in \{\text{black}, \text{white}\}$, if each e_i has color c_i. In this problem, you will design an algorithm that, given two vertices s, t and a desired color pattern $(c_1, \ldots, c_k)$, determines in $O(mk)$ time whether there exists a path from s to t with that color pattern. Note that paths in this problem need not be *simple*; they can visit vertices or edges more than once.

(10 pts) (a) Derive a recurrence relation, including base case(s), for this problem. Briefly justify its correctness.

Solution: Define $\text{path}^{(i)}(v)$ to denote whether there exists a path from s to v in G with pattern $(c_1, \ldots, c_i)$. So our goal is to compute $\text{path}^{(k)}(t)$. Observe that we have the following recurrence for $\text{path}^{(i)}(v)$:

$$\text{path}^{(i)}(v) = \bigvee_{(u,v)\in E:\ \text{color}(u,v)=c_i} \text{path}^{(i-1)}(u)$$

with the base case

$$\text{path}^{(0)}(v) = \begin{cases} \text{True} & \text{if } v = s \\ \text{False} & \text{if } v \neq s. \end{cases}$$

This is because there is a path from s to v with pattern $(c_1, \ldots, c_i)$ if and only if, for some incoming neighbor u of v, there is path from s to u with pattern $(c_1, \ldots, c_{i-1})$ and the color of (u, v) is c_i. The base case is straightforward: since $i = 0$, only the empty path (of no edges) has an empty (length-0) pattern, so the base case is true if and only if the starting and ending vertices are equal.

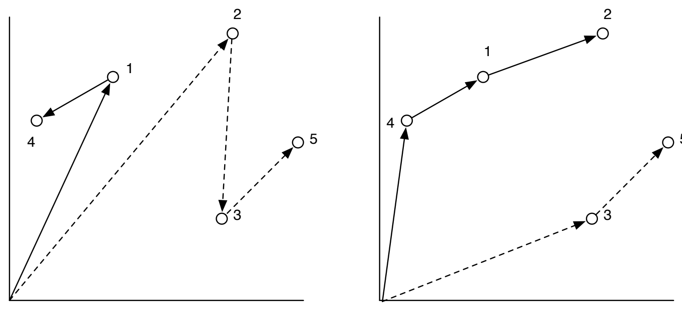
(5 pts) (b) Give a bottom-up dynamic programming algorithm, including pseudocode, for this problem that has running time $O(k(n+m))$ (or just $O(km)$ under the assumption that $m \geq n$).

Solution: The recurrence directly suggests the following bottom-up dynamic programming algorithm. It iterates from $i = 1$ to k, so every cell that is read from the table has already been computed when it is needed.

```
function PatternPath($G = (V, E), s, t, (c_1, \ldots, c_k)$)
    $\forall v \in V$ table$(0, v) \leftarrow \begin{cases} \text{True} & \text{if } v = s \\ \text{False} & \text{if } v \neq s. \end{cases}$
    for $i = 1$ to $k$ do
        for each vertex $v \in V$ do
            table$(i, v) = \bigvee_{(u,v)\in E:\text{color}(u,v)=c_i}$ table$(i-1, u)$
    return table$(k, t)$.
```

The running time is $O(k(n + m))$ because there are k iterations of the outer loop, and each iteration looks at each vertex exactly once and each edge exactly twice (once for each endpoint of the edge, since the graph is undirected), doing constant work for each.

7. **Optional extra credit: Wheel deliver.**



A legal (but not necessarily optimal) solution

An illegal solution because one rider visits 4 and then 1, which is out of order!

Figure 1: Two examples of routes for the two riders. The origin denotes the starting point p_0. One rider's route is shown with solid lines and the other with dashed lines. The route on the left is fair because points are visited by increasing indices. The route on the right is not fair because, for example, point p_4 is visited before point p_1.

Julie and Daphne have started a new unicycle-based meal delivery business called "Wheel Deliver" that works as follows. Residents of Ann Arbor submit their requests for a meal each day before noon using the Wheel Deliver App. In the afternoon, Julie and Daphne cook up a delicious meal in their kitchen and deliver the food on their unicycles.

Let n be the number of customers on a given day and $P = (p_1, \ldots, p_n)$ be the list of locations of these customers as points in the 2D plane, sorted in the order in which the customers submitted their requests: p_1 is the location of the first person to place an order, and p_n is the location of the last person to place an order. Let p_0 denote the location of the the kitchen from which Julie and Daphne depart. Let $d(p_i, p_j)$ denote the distance between p_i and p_j.

Julie and Daphne wish to split up the list of locations P, not necessarily evenly, so that each delivery is made by exactly one of them. In addition, "Wheel Deliver" has a fairness rule that goes like this: If customers at p_i and p_j for $i < j$ are assigned *to the same unicycle rider*, then the delivery to the customer at p_i must occur before the delivery to the customer at p_j—even if that makes the trip longer. (If p_i and p_j are assigned to different riders, then we don't care which delivery occurs first.) Subject to these constraints, the objective is to minimize the total distance travelled by the two unicycle riders.

Julie and Daphne have hired you to design an efficient algorithm that takes as input an array of points $P = (p_1, p_2, \ldots, p_n)$, sorted by the times in which their orders arrived, and outputs the sequences of delivery points for the two riders.

(4 EC pts) (a) Derive a recurrence relation, including base case(s), that is suitable for an efficient dynamic-programming solution for this problem. Briefly justify its correctness.

Solution: For $i < j$, define $\mathrm{Len}(i, j)$ to be the optimal total length of the two delivery routes assuming that the first j points receive delivery, and the riders end up at points i and j at the end. The base case is $i = 0, j = 1$; here the optimum solution is simply $\mathrm{Len}(0, 1) = d(p_0, p_1)$. (One could also have a base case of $i = j = 0$ with $\mathrm{Len}(0, 0) = 0$, though this deviates from the convention that $i < j$.) For the recursive case, we claim that

$$\mathrm{Len}(i,j) = \begin{cases} \mathrm{Len}(i,j-1) + d(p_{j-1},p_j) & \text{if } i < j-1 \\ \min\{\mathrm{Len}(i-1,j) + d(p_{i-1},p_i), \mathrm{Len}(i-1,i) + d(p_{i-1},p_j)\} & \text{if } i = j-1. \end{cases}$$

The justification for correctness is as follows. Consider an optimum assignment OPT of points to the riders, where they end at points $i < j$, respectively. If $i < j-1$, we know that the rider that ends at point j must visit point $j-1$ immediately prior, because the rider that ends at point $i < j-1$ cannot have visited point $j-1$ due to the fairness rule. Also, the riders must have used an optimum assignment leading up to points i and $j-1$, for if there were a better assignment with smaller total distance, we could replace it in OPT to reduce its distance, thus contradicting the assumed optimality of OPT. This establishes the first case of the recurrence.

Because $i < j$, the only remaining case is $i = j-1$. Here, *one* of the two riders visits point $i-1$ immediately prior, due to the fairness constraint. Whichever rider does so, the riders must have an optimum assignment up to that point, or else it could be replaced in OPT to reduce its distance, a contradiction. Thus, the optimum distance is the minimum of the optima for the cases "the rider who ends at point i comes from point $i-1$", and "the rider who ends at point j comes from point $i-1$." This minimum is what the second line of the recurrence captures, where note that for the second case, we have reversed the order of the arguments $i, i-1$ to Len, to put the smaller one first.

An alternative recurrence is:

$$\mathrm{Len}(i,j) = \begin{cases} \mathrm{Len}(i,j-1) + d(p_{j-1},p_j) & \text{if } i < j-1 \\ \min_{k<i}\{\mathrm{Len}(k,i) + d(p_k,p_j)\} & \text{otherwise.} \end{cases}$$

There are other valid recurrences that can work as well.

(4 EC pts) (b) Give a (bottom-up) dynamic programming algorithm, including pseudocode, for this problem that has running time $O(n^2)$.

Solution:
Assume function $\mathrm{d}(p_x, p_y)$ returns the distance between points x and y.

```
1: function TRIP($P = (p_0, \dots, p_n)$)                         ▷ $p_0$ is the kitchen
2:     allocate Len$[0, \dots, n][0, \dots, n]$                   ▷ only using Len$[i][j]$ where $i < j$
3:     Len$[0][1] \leftarrow$ d$(p_0, p_1)$
4:     for $j = 2$ to $n$ do
5:         for $i = 0$ to $j - 1$ do
6:             if $i < j - 1$ then
7:                 Len$[i][j]$ = Len$[i][j]$ + d$(p_{j-1}, p_j)$
8:             else                                                 ▷ $i = j - 1$
9:                 Len$[i][j]$ = min(Len$(i-1, j)$ + d$(p_{i-1}, p_i)$, Len$(i-1, i)$ + d$(p_{i-1}, p_j)$)
       return min(Len$[k][n]$ for $k$ between 0 and $n - 1$)
```

The pseudocode closely mirrors the recurrence. The two loops on lines 4 and 5 have $O(n^2)$ iterations each taking $O(1)$ time. Thus the runtime is $O(n^2)$.