(1) Shift scheduling.

This is a 1-dimensional DP problem, where we have already given you the recursive subproblem. Your job is to figure out how to populate **total_reward** and **picked**.

How to construct **total_reward**: populate this table from left to right, using the following base case and recurrence. *Base case*: **total_reward[0]** and **total_reward[1]** are just the corresponding entries in **rewards[0]** and **rewards[1]**, respectively.

Recurrence: At shift i, Alice either works or she doesn't. If she works, then the best possible reward for the subproblem is **rewards[i] + total_reward[i-2]**, skipping the shift she can't work. If she does not work shift i, then the best reward is **total_reward[i-1]**. Computing each of these and finding their maximum should take $O(1)$ time (since we are populating the table from left to right, and we are thus simply retrieving and comparing values that already exist).

How to construct **picked**: Fill in the **picked** array from right to left. At each timestep i, check if **total_reward[i]** $>$ **total_reward[i-1]**. If yes, then shift i must have been picked, and we should set **picked[i]** to true and **picked[i-1]** to false, and examine timestep $i-2$ next. If no, then shift i must not have been picked, so we should set **picked[i]** to false and examine timestep $i-1$ next.

Common mistakes: Some students tried to create a larger table than necessary; in particular, note that since we asked for $O(n)$ time, creating a data structure of size $O(n^2)$ (e.g., by misinterpreting that this is a 2d problem instead of a 1d problem) would put you over that time limit. Some students also mistakenly believed a shift i must be picked if $i-1$ is not. Note, however, that sometimes it is optimal to skip two shifts in a row (e.g., like in the example provided in the problem statement). There were also some issues in constructing the **picked** array, even if the optimal value was correctly obtained. Most students correctly understood that if **total_reward[i]** $>$ **total_reward[i-1]**, then i must have been picked. However, the most common mistake was to assume that if the above two values are equal, then $i-1$ must have been picked. To see this is not true, consider the case where there are 3 shifts, with rewards 5, 0, 0.

(2) Parallel MST. Yes, this algorithm works - in fact, it is called Borůvka's algorithm, and it is at its core a parallel version of Prim's algorithm. Interesting fact to note: Boruvka publishes his algorithm in 1926, 30 years before either Prim or Krushkal. Boruvka's paper was about a method of constructing an efficient electricity network for Moravia.

To prove this algorithm works, the students should observe that at each iteration, the edges added are all valid edges to add to an MST, and no cycles are introduced. This proof can be very simple: every edge added this way is the minimum edge across a cut from the tree represented by that processor to all other vertices. By the Cut Property (shown in class and on page 145 of the textbook), if edge costs are all distinct, then any edge added this way must exist in every minimum spanning tree. If we only add edges belonging to every MST, we can never have a cycle.

To provide more intuition for why there are no cycles: consider by way of contradiction an iteration when a cycle was introduced. Suppose during this iteration, out of the t originally disjoint trees, the algorithm forms a cycle that passes through trees $T_1, \ldots, T_k$. Then each T_i is connected to T_{i-1} and T_{i+1} by two new edges in the cycle. Thus there are k new edges that connect the k disjoint trees, meaning each tree must have chosen a different edge. Label the edge that T_i chose as e_i. Since the edge costs are distinct, going around the cycle, we have a chain of inequalities where $e_i < e_{i+1}$ for $i = 1, \ldots, k-1$ and $e_k < e_1$. However, this is a contradiction since then we have $e_1 < e_k$ and $e_k < e_1$. Thus there can be no cycles in any iteration.

At every iteration, if there are k trees currently active, then there are between $k/2$ and k unique edges added to trees (as two trees may simultaneously add the same edge). As each edge reduces the

total number of trees by 1, we note that this implies that the number of trees at each time step must decrease by a factor of *at least* 2 at each iteration, leading to a maximum of $O(\log n)$ total iterations of the algorithm to find a single tree. Each step will require each processor to check $O(m)$ edges, so the combined running time of this algorithm is $O(m \log n)$.

Common Mistakes:

- Providing a counterexample that relies on edges not having distinct edge weights.
- Not noticing that the cut property gives effectively the entire proof: every edge added this way is in the MST, and thus when all components are connected, we have achieved the min-cost MST.
- Disproving the possibility of a cycle linking two components, but not a cycle that goes through three or more components.
- Using a comparison to Prim's or Kruskal's without further justifying the correctness (again, using the MST).
- Using examples instead of full proofs to justify the worst-case running time.
- Claiming that the algorithm will *always* half the total number of trees at each iteration.

(3) Panini contention.

This is a tricky problem: the best subproblem to use is $OPT[i]$ is the optimal solution for time periods $1, \ldots, i$ assuming that we run the panini press at time i. Some students will instead try to use a different (more natural) definition (optimal solution for time periods $1, \ldots, i$), which we don't see how to us.

Even with the right subproblems, the optimal recurrence requires both dealing with the delay that "lateness" produces (the last paninis considered in $OPT[i]$ will not be actually handed to some customers until time $i + m$) and handling the fact that the recurrence is not producing a general optimal solution ($OPT[i]$ assumes the panini press runs at time i). Many students miss one or both of these things, but our goal is to still give partial credit to students who are attempting the correct steps for proving a DP algorithm correct.

There are solutions that work from the front of the schedule OR the back of the schedule; however, it is important to ensure that students can produce some kind of subproblem with a subset of the customers to make their solution work. A student tried a subproblem considering all customers but only if the panini press could run at minutes i or later; it can be very tricky to track the lateness cost produced by this.

Solution A: From Scratch

For students writing a solution from scratch instead of reducing, use the following grading guidelines:

Stating the subproblem: describe the definition of their subproblem, e.g. "$OPT[i]$ is the minimal sum of latenesses that is produced by the orders placed on or before time i assuming the panini press runs at time i."

Base case: Ideally, the base case for the first m minutes assumes the panini press cannot have run before that timestep:

$$\text{totalLateness}[j] = \sum_{i=1}^{j} (j - i + m) \cdot c_i.$$

This is the amount of lateness accumulated from time 0 to time j, assuming we haven't yet run the panini press.

Recurrence:

$$\text{totalLateness}[j] = \min_{1 \le k \le j-m+1} \text{totalLateness}[k-1] + \sum_{i=k}^{j} (j - i + m) \cdot c_i.$$

totalLateness$[k-1]$ gives the amount of wait time accumulated until time $k-1$, then $\sum_{i=k}^{j} (j-i+m) \cdot c_i$ gives the amount of wait time accumulated between time $k-1$ and time j.

Best solution will be stored at the minimum of entries n through $n + 2m - 1$

Rebuilding the schedule: Either keep track of a table of integer "pointers" to which prior subproblem was used to construct $OPT[i]$, or use a traditional "tracing back" technique where they compare the score with the previous value.

Common Mistakes:

- Writing a recurrent step that doesn't correctly compute a useful recurrence for the problem. (The most common case of this comes from not specifying in their subproblem that $OPT[i]$ assumes the panini press ran at time i)
- Not providing the subproblem description. Note that saying that $Opt(i)$ is the optimal subproblem for problem i is not useful. What is problem i?
- not explaining why the recurrence works (not even 1-2 sentences)
- Accidentally allowing the panini press to run twice in a row.
- Not indicating the direction in which to fill out the table
- Not rebuilding the schedule
- Attempting to use greedy for this

Solution B: Reduction to Segmented Least Squares

The algorithm above is effectively the same as Segmented Least Squares as presented in lecture in terms of the recurrence used. The easiest way to do this is to start by describing the method for precomputing the error/loss for a span of paninis from i to j, that is, if the panini press will get started at time j and will do all the panini's from time i (that is, the panini press last worked at time $i - 1$. Note that the error must be set to infinite (or super large) for entries where $j - i < m$ and $i > 1$, as we cannot start the panini press that close together. For $j \geq i + m$ or $i = 1$ we set

$$e_{ij} = \sum_{k=i}^{j} c_k \cdot (j - k + m)$$

and can set $C = 0$ as we don't get extra cost for running the panini press. Now let's use $Opt(i)$ as the minimum total cost (with cost as e_{ij}) to segment the interval $[1, \ldots, i]$. This is the recurrence

$$Opt(i) = \min_{j \leq i}(Opt(j - 1) + e_{ji})$$

with the base case of $Opt(0) = 0$.

Note that, as before, the value $Opt(i)$ is the best solution with the panani press startinga t time i, so once these values are computed, we need to return

$$\min_{n \leq i \leq n+m} Opt(i)$$

The running time via this reduction is $O(n^3)$ to compute all e_{ij} values, and then an additional $O(n^2)$ time to solve the dynamic programming problem. Note that we can reduce this to $O(n^2)$ by observing that we can find $e_{i,j+1}$ in $O(1)$ time using e_{ij} if we also keep the total paninis neeeded in this time period $n_{ij} = \sum_{k=i}^{j} c_k$, as then we get for any $j \geq i + m$ that $e_{ij+1} = e_{ij} + n_{ij} + c_{j+1}m$. Further we can get this down to $O(mn)$ by only offering finite e_{ij} when $j \leq i + 2m$ as longer segments get cheaper by breaking them into two.

Solution C: Reduction to Shortest path

This is very similar to the above problem. The nodes of the graph are the times we start the panini press, so nodes $1, \ldots, n + m - 1$, with two extra nodes s and t for start an finish. (note that $n + m - 1$ is the last reasonable moment to start the panini press if $n - 1$ was the previous time. We add edges (s, i) for all i, edge (i, t) for all $i \geq n$, and edges (i, j) if $j \geq i + m$. The edge costs are as follows

$$
\begin{aligned}
cost_{si} &= \sum_{k=1}^{i} c_k \cdot (i - k + m) \\
cost_{ij} &= \sum_{k=i+1}^{j} c_k \cdot (j - k + m) \\
cost_{it} &= 0
\end{aligned}
$$

As before, the simple way to compute all costs is $O(n^3)$, but can get this down to $O(mn)$ by only doing edges spanning a maximum of $2m$ size intervals, and updating $cost_{ij+1}$ from $cost_{ij}$. On an N node graph with M edges Bellman-Ford runs in $O(MN)$ time. Our graph has $N = m + n \leq 2n$ nodes and $M = mn$ edges, so that is a total running time of $O(n^2 m)$. We could do a bit better by noticing that we can run Dijkstra, as there were no negative edge costs, getting this down to $O(M \log N) = O(mn \log n)$, but this is still slower than the direct dynamic programming solutions above.