

Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Carnival

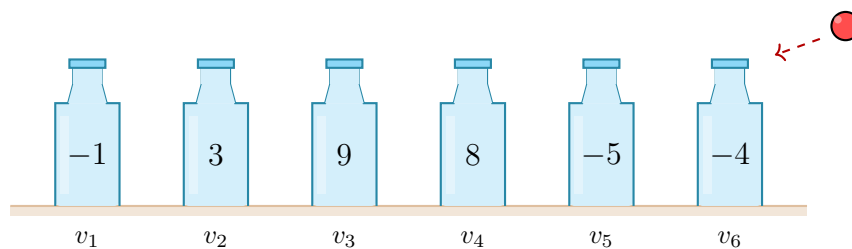
Playground: <https://gemini.google.com/share/d7225194933d>

You are throwing balls at a row of n bottles. Each bottle i has a value v_i written on it. On each throw, you have two options:

- Option 1 (Direct Hit): Hit bottle i directly. It falls over, and you earn v_i points.
- Option 2 (Gap Shot): Hit the gap between two adjacent bottles i and $i + 1$. Both bottles fall over, and you earn $v_i \times v_{i+1}$ points.

Each bottle can participate in at most one throw. You may also choose to leave bottles standing. Your goal is to maximize your total score.

Example: Consider the following row of 6 bottles:



Come up with a dynamic programming algorithm to compute the maximum possible score. Define your subproblems, write the recurrence, specify the base cases, and analyze the runtime.

Solution: Playground: <https://gemini.google.com/share/ad13a8cd3194>

Subproblems. Let $DP(i)$ be the maximum score achievable using only bottles $1, 2, \dots, i$.

Recurrence. For each bottle i , we have three choices: skip it, hit it directly, or pair it with bottle $i - 1$ via a gap shot. For $i \geq 2$:

$$DP(i) = \max \begin{cases} DP(i-1) & \text{(skip bottle } i) \\ DP(i-1) + v_i & \text{(hit bottle } i \text{ directly)} \\ DP(i-2) + v_{i-1} \cdot v_i & \text{(gap shot between } i-1 \text{ and } i) \end{cases}$$

Base case.

$$DP(0) = 0$$

Answer. $DP(n)$.

Runtime. $O(n)$ — there are $n + 1$ subproblems, each computed in $O(1)$ time.

Example trace for $[v_1, \dots, v_6] = [-1, 3, 9, 8, -5, -4]$:

i	0	1	2	3	4	5	6
v_i	—	-1	3	9	8	-5	-4
DP(i)	0	0	3	27	75	75	95

Optimal strategy: Skip $v_1 = -1$. Hit $v_2 = 3$ directly (+3). Gap shot $v_3 \times v_4 = 9 \times 8$ (+72). Gap shot $v_5 \times v_6 = (-5) \times (-4)$ (+20). Total: $3 + 72 + 20 = \mathbf{95}$.

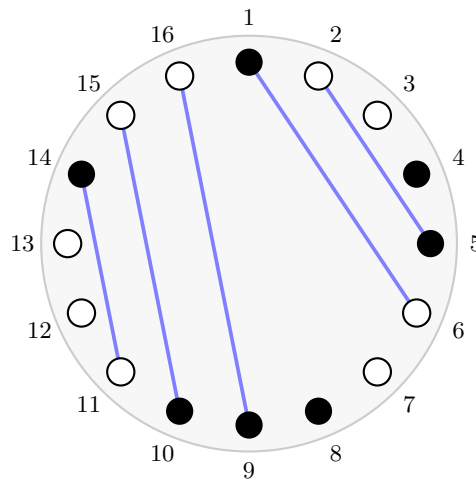
Note: the last gap shot illustrates why greedy approaches fail — pairing two negative values yields a positive product!

2 Circuit Design

A start-up is working on a new electronic circuit design for highly-parallel computing. Evenly spaced along the perimeter of a circular wafer sit n ports, each holding either a power source or a computing unit. Each computing unit needs energy from a power source, transferred between ports via a wire etched into the top surface of the wafer. However, if a computing unit is connected to a power source that is too close, the power can overload and destroy the circuit. Further, no two etched wires may cross each other.

The input to your algorithm is a boolean array $A[1..n]$ where $A[i]$ indicates whether port i is a power source or a computing unit. Describe an $O(n^3)$ -time dynamic programming algorithm to match computing units to power sources by etching non-crossing wires between them onto the surface of the wafer, in order to maximize the number of powered computing units, where wires may not connect two adjacent ports along the perimeter.

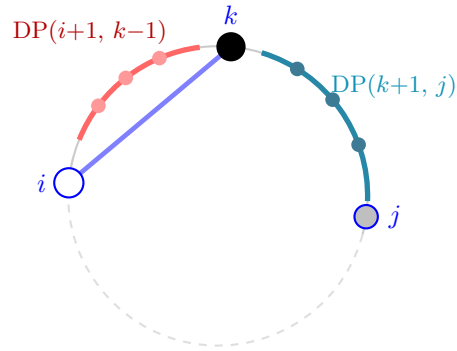
Below is an example wafer with $n = 16$ ports. Non-crossing wires connect computing units (\circ) to power sources (\bullet), powering 5 of the 9 computing units:



Solution: Subproblems. Since the circular arrangement has no distinguished starting point, we can label any port as port 1 and number the rest $2, \dots, n$ clockwise, flattening the circle into the array $A[1..n]$. The only artifact of the circular structure is that ports 1 and n remain adjacent.

For $1 \leq i \leq j \leq n$, let $DP(i, j)$ be the maximum number of matched pairs (non-crossing wires between opposite-type, non-adjacent ports) using only ports in the clockwise arc from port i to port j .

The key observation is that non-crossing wires on a circle behave like nested intervals: if we match port i to some port k in the arc, the wire from i to k splits the remaining ports into two independent sub-arcs.



Recurrence. Consider the first port i in the arc $[i, j]$. Either we leave port i unmatched, or we match it to some valid port k . For $j - i \geq 1$:

$$DP(i, j) = \max \left(DP(i+1, j), \max_{\substack{k: i+2 \leq k \leq j \\ \text{type}(i) \neq \text{type}(k) \\ \neg \text{adj}(i, k)}}} \{1 + DP(i+1, k-1) + DP(k+1, j)\} \right)$$

where $\text{adj}(a, b)$ is true iff $|a - b| = 1$ or $\{a, b\} = \{1, n\}$ (i.e., ports a and b are neighbors on the circular perimeter). The constraint $k \geq i + 2$ ensures we skip the immediately adjacent port, and the adj check additionally handles the wrap-around case where ports 1 and n are adjacent.

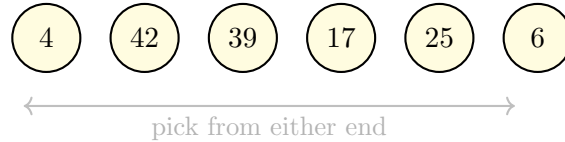
Base cases. $DP(i, j) = 0$ whenever $j - i + 1 \leq 1$ (arcs with fewer than two ports cannot form a matched pair). This includes $j < i$ (empty arc) and $j = i$ (single port).

Answer. $DP(1, n)$.

Runtime. There are $O(n^2)$ subproblems (one for each pair $i \leq j$). Each subproblem iterates over at most $O(n)$ choices for k , doing $O(1)$ work per choice. Total: $O(n^3)$.

3 Alternating Coin Game

A row of n coins with values V_1, V_2, \dots, V_n is laid out on a table, where n is even. Two players take turns. In each turn, the current player selects either the first or the last coin from the row, removes it permanently, and receives the value of that coin. Player 1 goes first.



- (a) Try playing the game on the row of coins shown above: $[4, 42, 39, 17, 25, 6]$. What is the best score any player can achieve? You can use: <https://gemini.google.com/share/1694f6ba260e> to play the game.
- (b) Design a dynamic programming algorithm that computes the maximum total value Player 1 can guarantee, regardless of how Player 2 plays.

Solution:

Part (a): Actually try to play the game!

Part (b):

Observation: Player 1 can always guarantee at least half the total.

Label the coins V_1, V_2, \dots, V_n . Player 1 can compute:

$$S_{\text{odd}} = V_1 + V_3 + V_5 + \dots + V_{n-1}$$

$$S_{\text{even}} = V_2 + V_4 + V_6 + \dots + V_n$$

Player 1 picks whichever sum is larger and commits to taking only coins from that parity. This is always possible: if Player 1 wants the odd-indexed coins, they start by taking V_1 (the first coin). No matter what Player 2 takes from either end, both exposed coins will be even-indexed, so after Player 2's turn the two exposed coins are again both odd-indexed. Player 1 can always pick from their chosen subset.

This guarantees Player 1 at least $\max(S_{\text{odd}}, S_{\text{even}}) \geq \frac{1}{2} \sum V_i$. But this parity strategy is not always optimal. Maximizing the payout requires a more sophisticated approach.

Optimal strategy via DP.

Subproblems. Let $DP(i, j)$ be the maximum total value the current player can guarantee when only coins V_i, V_{i+1}, \dots, V_j remain and it is their turn.

Recurrence. The current player picks either V_i (left end) or V_j (right end). After picking, the opponent plays optimally on the remaining coins. Since the opponent also plays optimally, they will leave us with the *worst case* for us on their turn:

- Pick V_i : the range becomes $(i+1, j)$ with the opponent to move. The opponent picks optimally, so we are left with the minimum of the two sub-ranges they could leave us:

$$\min(DP(i+2, j), DP(i+1, j-1)) + V_i$$

where $DP(i+2, j)$ results from the opponent taking V_{i+1} , and $DP(i+1, j-1)$ from the opponent taking V_j .

- Pick V_j : similarly, we get:

$$\min(DP(i+1, j-1), DP(i, j-2)) + V_j$$

where $DP(i+1, j-1)$ results from the opponent taking V_i , and $DP(i, j-2)$ from the opponent taking V_{j-1} .

We take the maximum over both choices:

$$DP(i, j) = \max \left\{ \min \left\{ \begin{array}{l} DP(i+1, j-1), \\ DP(i+2, j) \end{array} \right\} + V_i, \min \left\{ \begin{array}{l} DP(i, j-2), \\ DP(i+1, j-1) \end{array} \right\} + V_j \right\}$$

Base cases.

$$DP(i, i) = V_i \quad (\text{one coin left, just take it})$$

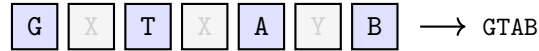
$$DP(i, i+1) = \max(V_i, V_{i+1}) \quad (\text{two coins left, take the larger})$$

Answer. $DP(1, n)$.

Runtime. There are $\Theta(n^2)$ subproblems (one for each pair $i \leq j$). Each takes $\Theta(1)$ time. Total: $\Theta(n^2)$.

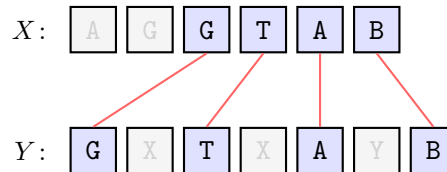
4 Longest Common Subsequence

A *subsequence* of a string is obtained by deleting zero or more characters without changing the order of the remaining characters. Crucially, the selected characters need not be contiguous. For example, GTAB is a subsequence of GXTXAYB:



Given two strings $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$, a *longest common subsequence* (LCS) is the longest string that is a subsequence of both X and Y .

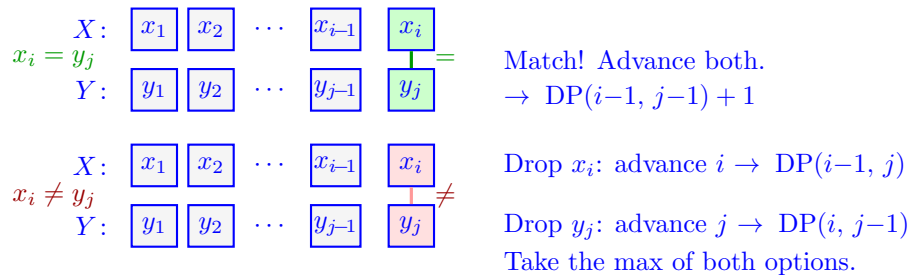
Example: Consider $X = \text{AGGTAB}$ and $Y = \text{GXTXAYB}$. The LCS is GTAB (length 4). Notice how the matched characters are spread across both strings, not necessarily next to each other:



Come up with a dynamic programming algorithm to compute the length of the longest common subsequence. Define your subproblems, write the recurrence, specify the base cases, and analyze the runtime.

Solution: Subproblems. Let $DP(i, j)$ be the length of the LCS of the prefixes $X[1 \dots i]$ and $Y[1 \dots j]$.

Recurrence. Compare the last characters of each prefix. We look at x_i and y_j at the ends of the two prefixes:



$$DP(i, j) = \begin{cases} DP(i-1, j-1) + 1 & \text{if } x_i = y_j \\ \max(DP(i-1, j), DP(i, j-1)) & \text{if } x_i \neq y_j \end{cases}$$

Base cases. $DP(i, 0) = 0$ for all i and $DP(0, j) = 0$ for all j (the empty string shares no characters with anything).

Answer. $DP(m, n)$.

Runtime. There are $O(mn)$ subproblems, each computed in $O(1)$ time. Total: $O(mn)$.