1. It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the list of n songs that the judges will play during the contest, in chronological order.

   For each integer k, if you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+Wait[k]$). Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

Solution: Without loss of generality, we can assume that $0 \le Wait[k] \le n-k$ for every index k, because we can't skip songs that don't exist or that are already past.

For any integer k, let $MaxScore(k)$ be the maximum score you can achieve only from songs k through n, or equivalently, after ignoring songs 1 through $k-1$. We need to compute $MaxScore(1)$.

For each index k, we either dance to the kth song or we don't. Thus, the $MaxScore$ function obeys the following recurrence:

$$MaxScore(k) = \begin{cases} 0 & \text{if } k > n \\ \max\left\{\begin{array}{c} Score[k] + MaxScore(k + Wait[k] + 1) \\ MaxScore(k+1) \end{array}\right\} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $MaxScore[1..n+1]$, which we can fill in reverse order in ***O(n) time***.

```
VoteForPedro(Score[1..n], Wait[1..n]):
  MaxScore[n+1] ← 0
  for k ← n down to 1
      Wait[k] ← max{0, min{Wait[k], n−k}}
      yes ← Score[k] + MaxScore[k + Wait[k] + 1]
      no ← MaxScore[k+1]
      MaxScore[k] ← max{yes, no}
  return MaxScore[1]
```

■

Solution: Without loss of generality, we can assume that $0 \le Wait[k] \le n-k$ for every index k, because we can't skip songs that don't exist or that are already past.

For each integer k, we define two functions:

- $MaxFirst(k)$ is the maximum score we can achieve if song k is the first song we dance to.
- $MaxAfter(k)$ is the maximum score we can achieve if we do not dance to any of the first k songs.

We need to compute $MaxAfter(0)$. These two functions satisfy the following mutual

recurrence. To compute $MaxScore(k)$, we need to choose the *second* song to dance to, if any, if we first dance to song k.

$$MaxFirst(k) = Score[k] + MaxAfter(k + Wait[k])$$

On the other hand, after skipping the first k songs, we either dance to song $k+1$ or we don't (unless song $k+1$ doesn't exist).

$$MaxAfter(k) = \begin{cases} 0 & \text{if } k \geq n \\ \max\left\{MaxFirst(k+1),\ MaxAfter(k+1)\right\} & \text{otherwise} \end{cases}$$

We can memoize these two functions into arrays $MaxFirst[1..n]$ and $MaxAfter[0..n]$, which we can fill in reverse order in ***O(n) time*** as follows:

```
DeliciousBass(Score[1..n], Wait[1..n]):
  MaxAfter[n] ← 0
  for k ← n down to 1
      Wait[k] ← max{0, min{Wait[k], n − k}}
      MaxFirst[k] ← Score[k] + MaxAfter[k + Wait[k]]
      MaxAfter[k − 1] ← max{MaxFirst[k], MaxAfter[k]}
  return MaxAfter[0]
```

■

Rubric: 10 points: standard dynamic programming rubric

2. Suppose you are given a NFA $M = (\{0,1\}, Q, s, A, \delta)$ and a binary string $w \in \{0,1\}^*$. Describe and analyze an algorithm to determine whether M accepts w. Report the running time of your algorithm as a function of k (the number of states in M) and n (the length of the input string w).

Solution: For any state p and any index i, let $\mathit{Accepts?}(p,i)$ indicate whether the NFA would accept the suffix $w[i..n]$ if it started in state p, or equivalently, if the set $\delta^*(p, w[i..n])$ contains an accepting state. We need to compute $\mathit{Accepts?}(1,1)$.

This function obeys the following recurrence:

$$\mathit{Accepts?}(p,i) = \begin{cases} \textsc{True} & \text{if } i > n \text{ and } p \in A \\ \textsc{False} & \text{if } i > n \text{ and } p \notin A \\ \displaystyle\bigvee_{q=1}^{k} \big(q \in \delta(p, w[i]) \wedge \mathit{Accepts?}(q, i+1)\big) & \text{otherwise} \end{cases}$$

Rewriting this recurrence in terms of our input representation gives us

$$\mathit{Accepts?}(p,i) = \begin{cases} \mathit{Acc}[p] & \text{if } i > n \\ \displaystyle\bigvee_{q=1}^{k} \big(\mathit{inDelta}[p, w[i], q] \wedge \mathit{Accepts?}(q, i+1)\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $\mathit{Accepts?}[1..k, 1..n]$, which we can fill column by column from right to left, filling each column in arbitrary order, in $\boldsymbol{O(k^2 n)}$ ***time***.

```
NFA-ACCEPTS(k, Acc[1..k], inDelta[1..k, 0..1, 1..k]):
  for p ← 1 to k
      Accepts?[p, n+1] ← Acc[p]
  for i ← n down to 1
      for p ← 1 to k
          Accepts?[p, i] ← FALSE
          for q ← 1 to k
              if (inDelta[p, w[i], q] ∧ Accepts?[q, i+1])
                  Accepts?[p, i] ← TRUE
  return Accepts?[1, 1]
```

■

Rubric: 10 points: standard dynamic programming rubric

3. (a) Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm should return the integer 3.

Solution: Let $A[1..n]$ be the input string. We define two functions:

- $IsPal?(i,j)$ is TRUE if the substring $A[i..j]$ is a palindrome, and FALSE otherwise.
- $MinPals(k)$ is the minimum number of palindromes that make up the suffix $A[k..n]$.

The problem asks for an algorithm to compute $MinPals(1)$.

Every string with length at most 1 is a palindrome. A string of length 2 or more is a palindrome if and only if its first and last characters are equal *and* the rest of the string is a palindrome. Thus:

$$IsPal?(i,j) = \begin{cases} \text{TRUE} & \text{if } i \geq j \\ (A[i] = A[j]) \wedge IsPal?(i+1, j-1) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $IsPal?[1..n, 0..n]$. Each entry $IsPal[i,j]$ depends only on $IsPal[i+1, j-1]$. Thus, we can fill this array row-by-row, from the bottom row upward.

```
COMPUTEISPAL?(A[1..n]):
  for i ← n down to 1
      IsPal?[i, i−1] ← TRUE
      IsPal?[i, i] ← TRUE
      for j ← i + 1 to n
          if A[i] = A[j]
              IsPal?[i, j] ← IsPal?[i + 1, j − 1]
          else
              IsPal?[i, j] ← FALSE
```

This algorithm clearly runs in $O(n^2)$ time. Notice that we aren't returning anything; we're just memoizing the function for use by the main algorithm.

The empty string can be partitioned into zero palindromes. Otherwise, the best palindrome decomposition has at least one palindrome. If the first palindrome in the *optimal* decomposition of $A[1..n]$ ends at index ℓ, the remainder must be the *optimal* decomposition for the remaining characters $A[\ell+1..n]$. The following recurrence considers all possible values of ℓ.

$$MinPals(k) = \begin{cases} 0 & \text{if } k > n \\ 1 + \min\left\{ MinPals(\ell+1) \;\middle|\; k \leq \ell \leq n \text{ and } IsPal?(k,\ell) \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $MinPals[1..n]$. Each entry $MinPals[k]$ depends only on entries $MinPals[\ell+1]$ with $\ell \geq k$. Thus, we can fill this array from right to left.

```
MinPals(A[1..n]):
  ComputeIsPal?(A[1..n])
  MinPals[n+1] ← 0
  for k ← n down to 1
      MinPals[k] ← ∞
      for ℓ ← k to n
          if IsPal?[k, ℓ]
              MinPals[k] ← min{MinPals[k], 1 + MinPals[ℓ+1]}
  return MinPals[1]
```

The subroutine ComputeIsPal? runs in $O(n^2)$ time, and the rest of the algorithm also clearly runs in $O(n^2)$ time. So the total running time is $\boldsymbol{O(n^2)}$.

If we had used the obvious iterative algorithm to test whether each substring is a palindrome, instead of precomputing the array *IsPal?*, our algorithm would have run in $O(n^3)$ time. ■

Rubric: 5 points, standard dynamic programming rubric. This is not the only correct solution. A correct algorithm that runs in $O(n^3)$ time (the same *MinPals* recurrence, but checking for palindromes by brute force) is worth at most 4 points; scale partial credit.

(b) Describe and analyze an efficient algorithm to find the length of the shortest metapalindrome for a given string. For example, given the string BUBBALASEESABANANA, your algorithm should return the integer 7.

Solution: Again, let $A[1..n]$ be the input string. For any index i, let $ShMeta(i)$ denote the length of the shortest metapalindrome of the substring $A[i+1..n-i]$; this substring is the result of removing the first i and last i characters of A. We need to compute $ShMeta(0)$. This function satisfies the following recurrence:

$$ShMeta(i) = \begin{cases} 0 & \text{if } i > n/2 \\ 1 & \text{if } i \le n/2 \text{ and } IsPal?(i+1, n-i) \\ 2 + \min\left\{ ShMeta(j) \;\middle|\; \begin{array}{ll} i < j \le n/2 & \text{and} \\ IsPal?(i+1, j) & \text{and} \\ IsPal?(n-j+1, n-i) & \end{array} \right\} & \text{otherwise} \end{cases}$$

Here we are using the same *IsPal?* function from part (a). We can memoize this function into an array $ShMeta[0..\lfloor n/2 \rfloor + 1]$, which we can fill from right to left in $\boldsymbol{O(n^2)}$ ***time*** after memoizing all values of *IsPal?*.

```
SHORTESTMETAPALINDROME(A[1..n]):
  COMPUTEISPAL?(A[1..n])
  m ← ⌊n/2⌋                          ⟨⟨middle index⟩⟩
  ShMeta[m+1] ← 0
  for i ← m−1 down to 0
      if IsPal?[i+1, n−i]
          ShMeta[i] ← 1
      else
          ShMeta[i] ← ∞
          for j ← i+1 to m
              if IsPal?[i+1, j] and IsPal?[n−j+1, n−i]
                  ShMeta[i] ← min{ShMeta[i], 2 + ShMeta[j]}
  return ShMeta[0]
```

■

Rubric: 5 points, standard dynamic programming rubric. This is not the only correct solution. A correct algorithm that runs in $O(n^3)$ time (the same main recurrence, but checking for palindromes by brute force) is worth at most 4 points; scale partial credit.

Standard dynamic programming rubric. For problems worth 10 poins:

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear English description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Deadly Sin: Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound. No extra credit for saving space.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative pseudocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

As usual, for problems worth less than 10 points, partial credit is scaled and rounded up to the nearest half-integer.