1. Describe and analyze an algorithm to figure out what Mr. Fox says.

Solution: Let *Chickens*(i,r,d) denote the maximum number of chickens that Mr. Fox can earn starting at the ith booth, assuming he just said "Ring!" r times in a row, or he just said "Ding!" d times in a row. Notice that we must have either $r = 0$ or $b = 0$, and Boggis, Bunce, and Bean's rules imply that $r \le 3$ and $b \le 3$. We need to compute *Chickens*$(1,0,0)$.

The *Chickens* function satisfies the following recurrence:

$$
\mathit{Chickens}(i,r,d) = \begin{cases} 0 & \text{if } i > n \\ \mathit{Chickens}(i+1,1,0) + A[i] & \text{if } i \le n \text{ and } d = 3 \\ \mathit{Chickens}(i+1,0,1) - A[i] & \text{if } i \le n \text{ and } r = 3 \\ \max\left\{ \begin{array}{l} \mathit{Chickens}(i+1,r+1,0) + A[i] \\ \mathit{Chickens}(i+1,0,d+1) - A[i] \end{array} \right\} & \text{otherwise} \end{cases}
$$

We can memoize this function into a three-dimensional array *Chickens*$[1..n, 0..3, 0..3]$. Because each entry *Chickens*$[i,r,d]$ in this array depends only on entries of the form *Chickens*$[i+1,\cdot,\cdot]$, we can fill this array in order of decreasing i, filling in the 6 interesting entries *Chickens*$[i,r,d]$ for each i in arbitrary order. The resulting algorithm runs in ***$O(n)$ time***. ■

Rubric: Standard dynamic programming rubric. This is not the only correct linear-time solution.

2. (a) Describe an algorithm to compute the length of the shortest palindrome supersequence of a given string.

Solution: Let $A[1..n]$ be the input string. For any indices i and j, let $SPS(i,j)$ denote the length of the shortest palindrome supersequence of the substring $A[i..j]$. This function obeys the following recurrence:

$$SPS(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ 2 + \min\left\{\begin{array}{c} SPS(i+1,j) \\ SPS(i,j-1) \end{array}\right\} & \text{if } i < j \text{ and } A[i] \neq A[j] \\ 2 + \min\left\{\begin{array}{c} SPS(i+1,j-1) \\ SPS(i+1,j) \\ SPS(i,j-1) \end{array}\right\} & \text{otherwise} \end{cases}$$

We need to compute $SPS(1,n)$.

We can memoize the function *LPS* into a two-dimensional array. Each entry depends on the $LPS[i,j]$ depends on (at most) three entries $LPS[i+1,j]$, $LPS[i,j-1]$, and $LPS[i+1,j-1]$ immediately below and/or to the left. Thus, we can fill the array from bottom to top in the outer loop, and from left to right in inner loop, as follows:

```
SPS(A[1..n]):
  for i ← n down to 1
      SPS[i, i−1] ← 0
      SPS[i, i] ← 1
      for j ← i+1 to n
          SPS[i, j] ← min {2 + SPS[i+1, j], 2 + SPS[i, j−1]}
          if A[i] = A[j]
              SPS[i, j] ← min {SPS[i, j], 2 + SPS[i+1, j−1]}
  return SPS[1, n]
```

The resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Rubric: 5 points: standard dynamic programming rubric (scaled). This is not the only correct evaluation order. This is not the only correct algorithm with this running time.

(b) Describe an algorithm to compute the length of the shortest dromedrome supersequence of a given string.

Solution: We actually solve a more general problem. A ***shortest common supersequence*** of two strings w and x is a string z of smallest possible length such that w is a subsequence of z and x is a subsequence of z. For example, `ITRONMYSTANRK` is a shortest common supersequence of `IRONMAN` and `TONYSTARK`. Every dromedrome supersequence of $A[1..n]$ is a string of the form ww, where w is a common supersequence of some prefix $A[1..j-1]$ and the corresponding suffix $A[j..n]$.

For any indices i, j, k, let $SCS(i, j, k)$ denote the length of the shortest common supersequence of the prefix $A[1..i]$ and the substring $A[j..k]$. The length of the shortest dromedrome supersequence of A is exactly $2 \cdot \max_{1 \le j \le n} SCS(j-1, j, n)$. The SCS function obeys the following recurrence:

$$SCS(i,j,k) = \begin{cases} i & \text{if } k < j \\ k - j + 1 & \text{if } i < 1 \\ \min \begin{Bmatrix} 1 + SCS(i,j,k-1) \\ 1 + SCS(i-1,j,k) \end{Bmatrix} & \text{if } A[i] \neq A[k] \\ \min \begin{Bmatrix} 1 + SCS(i,j,k-1) \\ 1 + SCS(i-1,j,k) \\ 1 + SCS(i-1,j,k-1) \end{Bmatrix} & \text{if } A[i] = A[k] \end{cases}$$

We can memoize this function into a 3-dimensional array $SCS[0..n, 0..n, 0..m]$. Each entry $SCS[i, j, k]$ depends only on entries $SCS[i', j, k']$ where $i' < i$ or $k' < k$ (or both); in particular, the second index j is unchanged. Thus, we can fill the array with three nested for-loops, considering j in arbitrary order in the outer loop, increasing i in the middle loop, and increasing k in the inner loop. We only need to consider entries $SCS[i, j, k]$ where $0 \le i \le j - 1 \le k \le n$. The resulting algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

```
MinDromeDrome(A[1..n]):
  mindds ← 2n    ⟨⟨easy upper bound⟩⟩
  for j ← 1 to n
      for k ← j − 1 to n
          SCS[0, j, k] ← 0
      for i ← 1 to j − 1
          SCS[i, j, j − 1] ← 0
          for k ← j to n
              SCS[i, j, k] ← min{1 + SCS[i, j, k − 1], 1 + SCS[i − 1, j, k]}
              if A[i] = A[k]
                  SCS[i, j, k] ← min{SCS[i, j, k], 1 + SCS[i − 1, j, k − 1]}
      mindds ← min{mindds, 2 · SCS[j − 1, j, n]}
  return mindds
```

■

Rubric: 5 points: standard dynamic programming rubric (scaled). This is not the only correct evaluation order. We can simplify this algorithm *very* slightly by using a two-dimensional memoization array, indexed only by i and k.

3. Suppose you are given a DFA M with k states for Jeff's favorite regular language $L \subseteq (0+1)^*$.

(a) Describe and analyze an algorithm that decides whether a given bit-string belongs to the language L^*.

Solution (8/10): Use the text-segmentation algorithm in the textbook, implementing IsWord(i, j) by tracing the substring $w[i\,..\,j]$ through the DFA in constant time per character. The segmentation algorithm makes $O(n^2)$ calls to IsWord, and each call to IsWord runs in $O(n)$ time, so the overall algorithm runs in ***$O(n^3)$ time***. ■

Solution (10/10): We use the text-segmentation algorithm in the textbook, implementing the function IsWord using the following dynamic programming algorithm. Assume that the DFA M is represented by a pair of arrays $\delta[1\,..\,k, 0\,..\,1]$ and $A[1\,..\,k]$, as shown on page 2 of the DFA lecture notes, and that the start state s is state 1. Let $w[1\,..\,n]$ denote the input string.

For any indices i and j, let $state(i, j) = \delta^*(s, w[i\,..\,j])$; more explicitly, $state(i, j)$ is the state that the substring $w[i\,..\,j]$ reaches in our given DFA. This function obeys the following recurrence (directly mirroring the recursive definition of δ^*):

$$state(i, j) = \begin{cases} 1 & \text{if } j < i \\ \delta[state(i, j-1), w[j]] & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $state[1\,..\,n, 0\,..\,n]$. Because each entry $state[i, j]$ depends only on the previous entry $state[i, j-i]$ in the same row, we can fill this array in standard row-major order in $O(n^2)$ time.

After we've precomputed the $state$ array, we can implement the helper function IsWord as follows:

```
IsWord(i, j):
  return A[state[i, j]]
```

The text segmentation makes $O(n^2)$ calls to IsWord, and IsWord runs in $O(1)$ time, so the overall algorithm runs in ***$O(n^2)$ time***. ■

(b) Describe and analyze an algorithm that partitions a given bit-string into as many substrings as possible, such that L contains every substring in the partition. Your algorithm should return only the number of substrings, not their actual positions.

Solution: We modify the text segmentation algorithm in the textbook as follows.

For any index i, let $\mathit{MinWords}(i)$ denote the minimum size of any partition of the suffix $w[i\,..\,n]$ into words, where "words" are defined by the boolean function IsWord. This function obeys the following recurrence:

$$\mathit{MinWords}(i) = \begin{cases} 0 & \text{if } i > n \\ \min\left\{1 + \mathit{MinWords}(j+1) \,\middle|\, \textsc{IsWord}(i,j)\right\} & \text{otherwise} \end{cases}$$

(To handle the case where the suffix $w[i\,..\,n]$ cannot be split into words, we define $\min\varnothing = \infty$.) Like the function *Splittable*, this function can be memoized into a one-dimensional array $\mathit{MinWords}[1\,..\,n+1]$, which we can fill in *decreasing* index order. The resulting algorithm makes $O(n^2)$ calls to IsWord.

```
MinWords(w[1..n]):
  MinWords[n+1] ← 0
  for i ← n down to 1
      MinWords[i] ← ∞
      for j ← i to n
          if IsWord(i, j)
              MinWords[i] ← min { MinWords[i], 1 + MinWords[j+1] }
  return MinWords[1]
```

Now we proceed exactly as in part (a). If we implement each call to IsWord by simulating the DFA, the resulting algorithm runs in $O(n^3)$ time. However, if we precompute all values of the *state* function, then each call to IsWord can be evaluated in $O(1)$ time, and thus the overall algorithm runs in $\boldsymbol{O(n^2)}$ ***time.*** ■