1. **Solution:** Let $G = (V, E)$. For a vertex u, let $N(u)$ be its neighbors (those that have an edge to u). To obtain a recursive backtracking algorithm to find the minimum weighted vertex cover (MWVC), we make the following observations.

   Let $\mathscr{S}$ be the set of all vertex covers in G. Fix some arbitrary vertex v in G. We partition $\mathscr{S}$ into $\mathscr{S}_1$ and $\mathscr{S}_2$ where $\mathscr{S}_1$ is the set of vertex covers that contain v, and $\mathscr{S}_2$ is the set of vertex covers that do not contain v. We can recursively find the minimum weight vertex cover from $\mathscr{S}$ as follows. If we include v in the vertex cover, then all edges incident to v are covered by it and hence we can remove v and its incident edges and find the minimum weight vertex cover in the graph $G_1 = G - v$ (removing a vertex removes all its incident edges too); this gives the min weight vertex cover from $\mathscr{S}_1$. If we do not include v then we are forced to include all of $N(v)$ into the vertex cover since the only way to cover an edge (v, u) without including v is to include u. Thus we include $N(v)$ into the vertex cover and recurse on $G_2 = G - v - N(v)$. This allows us to compute minimum weight vertex cover from $\mathscr{S}_2$. Note that $N(v)$ may be empty if there are no edge incident to v but even in this case G_2 is smaller since we remove v. Taking the smallest weight one among the two gives us an optimum solution. Letting $OPT(G)$ denote the weight of a minimum weight vertex cover of G we can summarize the preceding discussion via the following recursive relation.

$$OPT(G) = \begin{cases} 0 \text{ if } V = \emptyset \\ \min\{w(v) + OPT(G - v), w(N(v)) + OPT(G - v - N(v))\} \text{ otherwise} \end{cases}$$

   We describe the recursive algorithm below which returns both both the weight of a minimum weight vertex cover and also a vertex cover that achieves that weight.

```
MWVC($G$):
  if ($|V| = 0$))
      return $(0, \emptyset)$
  Let $v$ be an arbitrary vertex
  $G_1 = G - v$
  $(a, A) = $ MWVC($G_1$)
  $a = a + w(v)$
  $A = A \cup \{v\}$
  $G_2 = G - v - N(v)$
  $(b, B) = $ MWVC($G_2$)
  $b = b + \sum_{u \in N(v)} w(u)$
  $B = B \cup N(v)$
  if ($a \leq b$)
      return $(a, A)$
  else
      return $(b, B)$
```

   To analyze the running time of the algorithm we can write a recurrence. Let $T(n)$ be the running time on the graph with n nodes. Then, following the pseudocode,

$$T(n) \leq T(n-1) + T(n-k) + f(n)$$

   where $k = 1 + |N(v)|$ and $f(n)$ is the time the algorithm spends in the procedure outside the recursive calls. We can loosely upper bound $f(n)$ by $O(n^2)$ which suffices to create the graphs G_1 and G_2 from G, and to process other basic steps. Since we can only guarantee that $k \geq 1$ we have

$$T(n) \leq 2T(n-1) + O(n^2)$$

with base case $T(0) = O(1)$.

If you consider the recursion tree for the above recurrence it will have at most 2^n nodes and each node requires at most n^2 work so the total work is upper bounded by $O(n^2 2^n)$. One can potentially improve this bound slightly but we have not optimized it. ■

Rubric: 10 points.

- + 6 for a correct recurrence
  - + 2 for a clear English description of the function you are trying to evaluate.
  - + 1 for base case(s) —½ for one minor bug
  - + 3 for recursive case(s) —1 for each minor. No credit for the rest of the problem if the recursive case(s) are incorrect.
- + 4 for details for algorithm
  - + 3 for correct algorithm
  - + 1 for runtime analysis

2. **Solution:** Let $w \in \Sigma^*$ be the input, and let n be its length. We define a boolean function $InStar(i)$, which is TRUE if and only if the substring $w_i w_{i+1} \ldots w_{n-1} w_n$ is in $(L_1 + L_2)^*$. (We let $InStar(n+1) =$ TRUE, because the corresponding substring is the empty string)

By the definition of Kleene star, w is in $(L_1 + L_2)^*$ if and only if w is either ε, or can be expressed as $w = w_1 w_2$ where $w_1 \neq \varepsilon$ is in $L_1 \cup L_2$ and $w_2 \in (L_1 + L_2)^*$. This gives us the following recursive definition:

$$InStar(i) = \begin{cases} \textsc{True} & \text{if } i = n+1 \\ \textsc{True} & \text{if } \exists\, j \in \{1,2\}, i \leq k \leq n \text{ st. IsStringIn}L_j(w_i w_{i+1} \ldots w_k) \wedge InStar(k+1) \\ \textsc{False} & \text{Otherwise} \end{cases}$$

We need to compute $InStar(1)$.

We can memoize all function values into a one-dimensional array $InStar[1..n+1]$. Each array entry $InStar[i]$ depends only on the entries to the right. Thus, we can fill the array from right to left. The recurrence gives us the following pseudocode:

```
InStar?($A[1,\dots,n]$):
    $InStar[n+1] \gets$ True
    for $i \gets n$ down to 1
        $InStar[i] \gets$ False
        for $j \gets i$ to $n$
            if $InStar[j+1]$
                if IsStringIn$L_1$($A[i,\dots,j]$)
                    $InStar[i] \gets$ True
                    break
                else if IsStringIn$L_2$($A[i,\dots,j]$)
                    $InStar[i] \gets$ True
                    break

    return $InStar[1]$
```

The array assignments are constant time, as are the calls to IsStringInL_1 and IsStringInL_2 (as stated in the problem). The outer loop runs n times, and the inner loop runs at most n times per iteration of the outer loop (since we're just iterating over some subset of the integers from 1 to n). Since the work inside of the loops is constant time, this means that the total time for the algorithm is a constant, times n for the inner loop, times n for the outer loop, so the algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Rubric: Standard Dynamic Programming Rubric (Max 8 points for a slower solution, scale partial credit accordingly)

3. **Solution:** Let $A[1\,..\,n]$ be the input string. We define a function $MinPals(k)$ to be the minimum cost palindromic decomposition of $A[k\,..\,n]$. The problem asks for an algorithm to compute $MinPals(1)$. For now, suppose that we have a function $IsPal(k,\ell)$ that is 1 if $A[k\,..\,\ell]$ is a palindrome and ∞ otherwise. Note that $IsPal(k,\ell)$ can easily be computed in $O(n)$ time by scanning from both sides of $A[k\,..\,\ell]$ and checking for equality at each step.

We give a recurrence for *MinPals*. We interpret the cost of the empty string to be 0, which we give our reasons for below. This would imply that $MinPals(k) = 0$ for $k > n$. Otherwise, the best palindromic 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.

$$MinPals(k) = \begin{cases} 0 & \text{if } k > n \\ \min\left\{\text{cost}(\ell+1-k)\cdot IsPal(k,\ell) + MinPals(\ell+1) \mid k \le \ell \le n\right\} & \text{otherwise} \end{cases}$$

As cost may be 0, we will assume that $0\cdot\infty = \infty$.

We can now explain the decision to interpret the cost of the empty string to be 0. In determining $MinPals(k)$, one possible decomposition is the decomposition of $A[k\,..\,n]$ into one palindrome. The recurrence computes this cost as $\text{cost}(n+1-k)+MinPals(n+1)$. Since the cost of the decomposition of $A[k\,..\,n]$ into one palindrome should just be $\text{cost}(n+1-k)$, we want to $MinPals(n+1) = 0$.

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

```
MinPals($A[1..n]$):
  $MinPals[n+1] \leftarrow 0$
  for $k \leftarrow n$ down to 1
      $MinPals[k] \leftarrow \infty$
      for $\ell \leftarrow k$ to $n$
          $MinPals[k] \leftarrow \min\{MinPals[k],\ \mathrm{cost}(\ell+1-k) \cdot IsPal(k,\ell) + MinPals[\ell+1]\}$
  return $MinPals[1]$
```

The outer for loop requires n iterations and the inner for loop requires at most n iterations. Furthermore, the amount of time needed to compute $IsPal(k,\ell)$ is $O(n)$ and computing the minimum in the inner for loop takes $O(1)$ time, assuming that $\mathrm{cost}(i)$ can be computed in $O(1)$ time. Thus, there are a total of $O(n^2)$ iterations of the inner for loop and each iteration requires $O(n)$ time, implying an overall running time of $O(n^3)$.

Note that one can precompute the values of $IsPal(k,\ell)$ for all k and ℓ in $O(n^2)$ time. This would have reduced the running time to compute $MinPals(A[1..n])$ to $O(n^2)$ as the inner for loop would now only require $O(1)$ time. Since precomputing $IsPal$ required $O(n^2)$ time, this would lead to an algorithm with an overall running time of $O(n^2)$.

For the sake of completeness, we show how to compute $IsPal$ in $O(n^2)$ time. Observe that we only ever need to compute $IsPal(i,j)$ for $i \le j$, so we will only concern ourselves with the situation where $i \le j$. We use the following two observations. First, every string of length 1 is a palindrome and strings of length 2 are palindromes if the characters are the same. Second, every string of length 3 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. Let f be a function from boolean values to $\{\infty, 1\}$ where $f(\textsc{True}) = 1$ and $f(\textsc{False}) = \infty$. Combining the two observation, we have

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

We can memoize the $IsPal$ function into a two-dimensional array $IsPal[1..n, 1..n]$. For all i and j where $i+2 \le j$, each entry $IsPal[i,j]$ depends only on $IsPal[i+1, j-1]$. Thus, we can fill this array row-by-row, from row n to row 1.

```
ComputeIsPal($A[1..n]$):
  for $i \leftarrow n$ down to 1
      for $j \leftarrow i$ to $n$
          if $i = j$ or $i+1 = j$
              $IsPal[i,j] \leftarrow f(A[i] = A[j])$
          else
              $IsPal[i,j] \leftarrow f((A[i] = A[j]) \wedge IsPal[i+1, j-1])$
```

Again, as we have a nested for loop where both loops require at most n iterations and each iteration requires $O(1)$ time, the total running time of the algorithm is $O(n^2)$. Notice that we aren't returning anything; we're just memoizing the function for use by the main algorithm.

■

Rubric: Standard dynamic programming rubric (given on last page). The problem is worth 10 points. Max 8 points for a slower, polynomial-time algorithm; scale partial credit accordingly.

Rubric: Standard dynamic programming rubric

For problems worth 10 points:

- 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.) **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.
- 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).