1. Given a graph $G=(V,E)$ a vertex cover of $G$ is a subset $S \subseteq V$ of vertices such that for every edge $(u,v) \in E$, $u$ or $v$ is in $S$. The goal in the minimum vertex cover problem is to find a vertex cover $S$ of smallest size. In some cases vertices may have non-negative weights $w : V \to \mathbb{Z}_+$ and the goal is to find a vertex cover of minimum weight. You can find some examples and discussion at the following Wikipedia link https://en.wikipedia.org/wiki/Vertex_cover. Describe a *recursive* algorithm that given a graph $G=(V,E)$ and weights $w(v), v \in V$ outputs a vertex cover of $G$ with minimum weight. Do not worry about the running time.

**Solution:** For a graph $G=(V,E)$, let $MWVC(G)$ be the function returning the weight of a minimum weight vertex cover (MWVC) of $G$. Minor modifications to the implementation described below will allow us to instead return an actual MWVC.

(a) The base case is when there are no edges, in which case $\varnothing$ is (vacuously) a MWVC with weight 0.

(b) In general case, pick an arbitrary vertex $v$, and let $E_v$ be the set of edges incident to $v$. There are two cases.

   i. $v$ belongs to a MWVC. Including $v$ in the MWVC means the edges in $E_v$ are covered by $v$, and therefore we can safely remove $v$ and the edges in $E_v$ from $G$ and recursively compute a MWVC for the remaining vertices and edges. In other words, we can create a graph $G' = (V \setminus \{v\}, E \setminus E_v)$ (i.e., $G$ with $v$ and its incident edges removed) and recursively compute a MWVC of $G'$. In this case $MWVC(G) = w(v) + MWVC(G')$.

   ii. $v$ does not belong to a MWVC. In this case, the edges in $E_v$ must be covered by $v$'s neighbors, which we will denote by $N_v$, and therefore all of $v$'s neighbors must be part of the MWVC. The vertices in $N_v$ will cover all the edges that they are incident to, which we will write as $EN_v$ (notice that $E_v \subseteq EN_v$), leaving us to recurse on the remaining graph $G'' = (V \setminus (\{v\} \cup N_v), E \setminus EN_v)$. The result is $MWVC(G) = \big(\sum_{u \in N_v} w(u)\big) + MWVC(G'')$.

   However, we do not know which of the two cases is correct. Following the recursive backtracking paradigm, we will try both options and then take the smaller of the two cases as the correct one.

The analysis above leads us to the following recursive formulation for $MWVC(G)$, where to simplify the presentation we take in $G$ explicitly as a pair $(V,E)$.

$$MWVC(V,E) = \begin{cases} 0 & \text{if } E = \varnothing \\ \min \left\{ \begin{array}{c} w(v) + MWVC(V \setminus \{v\}, E \setminus E_v), \\ \big(\sum_{u \in N_v} w(u)\big) + MWVC(V \setminus (\{v\} \cup N_v), E \setminus EN_v) \end{array} \right\} & \text{otherwise} \end{cases}$$

For people who prefer pseudocode:

```
MWVC($V, E$):
    if $E$ is empty
        return 0
    $v \leftarrow$ an arbitrary vertex in $V$
    ⟨⟨$v$ belongs to a MWVC⟩⟩
    $mwvc\_with\_v \leftarrow w(v) + \text{MWVC}(V \setminus \{v\}, E \setminus E_v)$
    ⟨⟨$v$ does not belong to a MWVC⟩⟩
    $sum \leftarrow 0$
    for $u$ in $N_v$
        $sum \leftarrow sum + w(u)$
    $mwvc\_without\_v \leftarrow sum + \text{MWVC}(V \setminus (\{v\} \cup N_v), E \setminus EN_v)$
    return min$\{mwvc\_with\_v, mwvc\_without\_v\}$
```

Let $T(n)$ be the time taken by the algorithm on a graph with $n$ vertices, and $S(n)$ be the extra time taken inside the body of the MWVC function (i.e., excluding the recursive calls). Then $T(n) = T(n-1) + T(n-k-1) + S(n)$, where $k = |N_v|$ is the number of $v$'s neighbors. Since $k \geq 0$, $T(n) \leq 2T(n-1) + S(n)$. Outside of the recursive calls, the function has to enumerate at most $n$ neighbor vertices and at most $n^2$ edges, which can be done in $O(n^2)$ time via brute force, so $S(n) = O(n^2)$, which means that $T(n) = O(n^2 2^n)$. ■

**Rubric:** 10 points: Modified version of standard Dynamic Programming rubric.

- + 1 points for a clear **English** description of the function you are trying to evaluate. This means describing what the output of the function means with respect to the inputs, *not* regurgitating the execution of your algorithm.
- + 2 for base case. −1 for one *minor* bug, like a typo or an off-by-one error.
- + 6 for recursive case(s). −2 for one *minor* bug, like a typo or an off-by-one error.
- + 1 point for time analysis.

2. Let $\Sigma$ be a finite alphabet and let $L_1$ and $L_2$ be two languages over $\Sigma$. Assume you have access to two routines IsStringIn$L_1(u)$ and IsStringIn$L_2(u)$. The former routine decides whether a given string $u$ is in $L_1$ and the latter whether $u$ is in $L_2$. Using these routines as black boxes describe an efficient algorithm that given an arbitrary string $w \in \Sigma^*$ decides whether $w \in (L_1 + L_2)^*$. To evaluate the running time of your solution you can assume that calls to IsStringIn$L_1()$ and IsStringIn$L_2()$ take constant time.

**Solution:** Given an input string $w[1..n]$, let $InStar(i)$ be the function that returns TRUE if and only if the suffix $w[i..n]$ is in $(L_1 + L_2)^*$.

By the definition of Kleene star, $w$ is in $(L_1 + L_2)^*$ if and only if $w$ is either empty, 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 for $InStar(i)$:

$$InStar(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \Big( \big(\text{IsStringIn}L_1(w[i..j]) \vee \text{IsStringIn}L_2(w[i..j])\big) \wedge InStar(j+1) \Big) & \text{otherwise} \end{cases}$$

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?($w$):
  $InStar[n+1] \leftarrow$ TRUE
  for $i \leftarrow n$ to 1
      $InStar[i] \leftarrow$ FALSE
      for $j \leftarrow i$ to $n$
          if IsStringIn$L_1(w[i..j])$ or IsStringIn$L_2(w[i..j])$
              if $InStar[j+1]$
                  $InStar[i] \leftarrow$ TRUE
  return $InStar[1]$
```

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

**Solution (Clever):** We will define a function IsStringIn$L_1$or$L_2()$ as follows:

```
IsStringIn$L_1$or$L_2(w)$:
  return IsStringIn$L_1(w)$ or IsStringIn$L_2(w)$
```

Given a string $w$, IsStringIn$L_1$or$L_2(w)$ returns whether $w \in L_1 \cup L_2$ in constant time.

Thus giving IsStringIn$L_1$or$L_2()$ as a black box to the algorithm for Text Segmentation in lecture/Jeff's book results an algorithm for determining if an input string $w[1..n]$ is in $(L_1 + L_2)^*$, with running time $\boldsymbol{O(n^2)}$. ■

**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.

3. Recall that a *palindrome* is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

   Any string can be decomposed into a sequence of palindrome substrings. For example, the string `BUBBASEESABANANA` ("Bubba sees a banana.") can be broken into palindromes in the following ways (among many others):

   `BUB • BASEESAB • ANANA`
   `B • U • BB • A • SEES • ABA • NAN • A`
   `B • U • BB • A • SEES • A • B • ANANA`
   `B • U • B • B • A • S • E • E • S • A • B • ANA • N • A`

   Since a given string $w$ can always be decomposed to palindromes we may want to find a decomposition that optimizes some objective. Here are two objectives. The first objective is to decompose $w$ into the smallest number of palindromes. A second objective is to find a decomposition such that each palindrome in the decomposition has length at least $k$ where $k$ is some given input parameter. Both of these can be cast as special cases of an abstract problem. Suppose we are given a function called $cost()$ that takes a positive integer $h$ as input and outputs an integer cost($h$). Given a decomposition of $w$ into $u_1, u_2, \dots, u_r$ (that is, $w = u_1 u_2 \dots u_r$) we can define the cost of the decomposition as $\sum_{i=1}^{r} \text{cost}(|u_i|)$.

   For example if we define $\text{cost}(h) = 1$ for all $h \geq 1$ then finding a minimum cost palindromic decomposition of a given string $w$ is the same as finding a decomposition of $w$ with as few palindromes as possible. Suppose we define cost() as follows: $\text{cost}(h) = 1$ for $h < k$ and $\text{cost}(h) = 0$ for $h \geq k$. Then finding a minimum cost palindromic decomposition would enable us to decide whether there is a decomposition in which all palindromes are of length at least $k$; it is possible iff the minimum cost is 0.

   Describe an efficient algorithm that given black box access to a function cost(), and a string $w$, outputs the value of a minimum cost palindromic decomposition of $w$.

**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)**. Let *IsPal*($k, \ell$) be a function that returns whether or not $A[k\,..\,\ell]$ is a palindrome. Naively, *IsPal*($k, \ell$) can 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 $\ell$, the remainder must be the *optimal* decomposition for the remaining characters $A[\ell + 1\,..\,n]$. The following recurrence considers all possible values of $\ell$.

$$\mathit{MinPals}(k) = \begin{cases} 0 & \text{if } k > n \\ \min \left\{ \begin{array}{r} \text{cost}(\ell + 1 - k) \\ + \mathit{MinPals}(\ell + 1) \end{array} \middle| \begin{array}{c} k \leq \ell \leq n \text{ and} \\ \mathit{IsPal}(k, \ell) \end{array} \right\} & \text{otherwise} \end{cases}$$

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 $\mathrm{cost}(n+1-k) + \mathit{MinPals}(n+1)$. Since the cost of the decomposition of $A[k\,..\,n]$ into one palindrome should just be $\mathrm{cost}(n+1-k)$, we want to $\mathit{MinPals}(n+1) = 0$.

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

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

The outer for loop requires $n$ iterations and the inner for loop requires at most $n$ iterations. Furthermore, given $k$ and $\ell$, $\mathit{IsPal}(k,\ell)$ can be computed in $O(n)$, 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 $O(n^2)$ iterations of the inner for loop, each requiring $O(n)$ time, for an overall running time of $O(n^3)$.

However, one can precompute *IsPal* in $O(n^2)$ time. This would reduce the running time to compute $\mathit{MinPals}(A[1\,..\,n])$ to $\boldsymbol{O(n^2)}$. Observe that we only ever need to compute $\mathit{IsPal}(i,j)$ for $i \leq j$. 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. Thus *IsPal* satisfies the following recurrence:

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

We memoize the *IsPal* function into a two-dimensional array $\mathit{IsPal}[1\,..\,n, 1\,..\,n]$. For $i, j$ where $i+2 \leq j$, $\mathit{IsPal}[i,j]$ depends on $\mathit{IsPal}[i+1, j-1]$, so we can fill this array from row $n$ down to 1 in the outer loop, and from $i$ to $n$ for each row in the inner loop.

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

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).