1. A grammar G is a way of generating strings of "terminal" characters from a nonterminal symbol S, by applying simple substitution rules, called productions. If $B \to \beta$ is a production, then we can convert a string of the form $\alpha B\gamma$ into the string $\alpha\beta\gamma$. A grammar is in Chomsky normal form if every production is of the form "$A \to BC$" or "$A \to a$," where A, B, and C are nonterminal characters and a is a terminal character. Design an $O(n^3)$-time dynamic programming algorithm for determining if string $x = x_0x_1 \dots x_{n-1}$ can be generated from the start symbol S.

   Hint: The solution to this problem is the very famous CYK-algorithm, but the main point of this exercise is to see if you can summarize this algorithm succinctly (must formulate the solution as a recurrence with english description for each case).

Solution: In this question, we can create a two-dimensional table $T[i][j]$ where each element represents the set of non-terminal symbols that can generate substring $x_ix_{i+1}\cdots x_j$.
This table has following characterisctics:

(a) Any element below diagonal is 0

(b) Base case: any element $T[i][i]$ on diagonal, if $A \to w_i$ exists, $T[i][i] = A$

(c) recursive case: for elements above diagonal, at every possible split point k, where $i \le k < j$. if any production rule $A \to BC$ exists, where B can generate $x_ix_{i+1}\cdots x_k$, C can generate $x_{k+1}\cdots x_j$, this means $T[i][j] = A$.

We can define a function CYKALGO($x[x_0, x_1, \cdots x_{n-1}]$) to check if a string can be generated from symbol S, where $x[x_0, x_1, \cdots x_{n-1}$ is the string need to be checked.

```
CYKALGO($x[x_0, x_1, \cdots x_{n-1}]$):
  ⟨⟨Define a $n \times n$ 2d table $T[i][j]$⟩⟩
  for $i \leftarrow 0$ to $n-1$: ⟨⟨Initialize table T⟩⟩
      for $j \leftarrow 0$ to $n-1$:
          $T[i][j] \leftarrow \{\}$
  for $i \leftarrow 0$ to $n-1$:
      $T[i][j] \leftarrow \{A | A \to w_i \in grammar\}$ ⟨⟨Base case⟩⟩
  for $length \leftarrow 2$ to $n$:
      for $i \leftarrow 0$ to $n - length$:
          $j \leftarrow i + length - 1$
          for $k \leftarrow i$ to $j-1$:
              $T[i][j] \leftarrow T[i][j] \cup \{A | A \to BC, B \in T[i][k], C \in T[k+1][j]\}$
  if $S \in T[0][n-1]$: ⟨⟨Final check⟩⟩
      return True
  else:
      return False
```

Finally, if $S \in T[0][n-1]$, the string can be generated from symbol S. The time complexity is $O(n^3)$ because we have three levels of loops which enumerate rows, columns of the table and length of substring respectively.

■

2. Suppose you have a circular necklace with n jewels, each with some value, v_i. You wish to sell the jewels individually, but unfortunately, removing jewel i from the necklace breaks the neighboring jewels, making them worthless. Design an efficient algorithm to figure out the maximum revenue you can receive from the circular necklace.

Solution: We decide to pick an arbitrary gem and say it's v_0. Then, we say the gems proceed in a clockwise fashion as $v_1, v_2, \ldots, v_{n-1}$. From this lens, the problem is a classic example of an optimization problem which can be solved with DP. The idea is that at each gem, we make a "decision" as to if we are going to sell it or not. The quirky bit is the first gem, as selling it or not affects whether you can sell the last gem. We rectify this by modifying our recurrence to store whether we sold the first gem or not, handling both cases. This can be encapsulated in the following recurrence MaxProfit(k, a) which represents the maximum profit we can make starting at gem k with $a = 0$ meaning we didn't sell the first gem and $a = 1$ meaning we did.

$$\text{MaxProfit}(k,a) = \begin{cases} 0 & k \geq n \text{ or } (k = n-1 \wedge a = 1) \\ \max \begin{cases} \text{MaxProfit}(k+2, a) + v_k \\ \text{MaxProfit}(k+1, a) \end{cases} & \text{otherwise} \end{cases}$$

We memoize this recurrence with a 2-D table with the first row representing $a = 0$ and the second row $a = 1$ with each column represents a gem. We fill in both rows from high k to low k independently. Our return value is the following:

$$\max\{\text{MaxProfit}(1, 0), v_0 + \text{MaxProfit}(2, 1)\}$$

This requires $O(n)$ space and $O(n)$ time complexity since each array entry can be filled in $O(1)$ time. ■

3. We have an $n \times m$ rectangle/array M. Each cell in M has an arbitrary label value, which is a positive integer. We want to cut M into several pieces of rectangles such that all the grid cells in a single rectangle have the same value. A rectangle can be cut only along one of it's horizontal or vertical grid lines, which will break it into two rectangles. Note that you can only cut one rectangle at a time. And the price of cutting the rectangle is the area of the rectangle (aka the number of grid cells in it). Clearly the worst case is to cut M into $n \times m$ pieces. But we want to find if there is any better solution.

   Describe an algorithm that computes the minimum cost of cutting the input grid M into pieces, following the above rules, so that if two grid cells are on the same rectangle, they must have the same label. Your algorithm should be as fast as possible.

Solution: +10pts We would introduce a sub-problem here first. The sub-problem is defined by a rectangle, such a rectangle is determined by two points $p, q \in M = [n] \times [m]$. We first precompute, for each $p, q \in M$, if the rectangle
$r(p,q) = \{(i,j) \in M | x(p) \leq i \leq x(q) \text{ and } y(p) \leq j \leq y(q)\}$
is valid — that is, the labels for all cells in $r(p,q)$ are the same. Namely, this takes $O((nm)^3)$ time. But a simple recursive algorithm can do it in $O((nm)^2)$ using memoization. Introduce the function $label(p,q)$ where it will return the label in the rectangle r, denoted with p as the top left corner and q as the bottom right corner, if all cells in it have the same label; and it will return *BAD* if the labels are different. Then we have the following recurrence:

```
label(p,q):
  if p = q then
    return M[p]
  end if
  if $x(p) < x(q)$ then
    $mm = \lfloor (x(p) + x(q))/2 \rfloor$
    $l_1 = label(p, (mm, y(q)))$
    $l_2 = label((mm+1, y(p)), q)$
  else
    $mm = \lfloor (y(p) + y(q))/2 \rfloor$
    $l_1 = label(p, (x(q), mm))$
    $l_2 = label((x(p), mm+1), q)$
  end if
  if $l_1 \neq l_2$ then
    return BAD
  end if
  return $l_1$
```

Clearly the function label(p,q) contains at most $O((nm)^2)$ different recursive calls. So it takes $O((nm)^2)$ time to fill the dp table, of size $O((nm)^2)$. And the filling order will be starting from the rectangle with size $1, 2, 3, \ldots, nm$. Assume V is the memoization structure we use, $V[p,q]$ will store the value of $label(p,q)$.

Next, let us also define a helper function that runs in $O(1)$ time, to calculate the price of cutting a rectangle.

```
price(p,q):
  if $x(p) > x(q)$ or $y(p) > y(q)$ then
```

```
    return $\infty$
  end if
  return $(x(p)-x(q)+1)(y(q)-y(p)+1)$
```

The solution is now straight-forward. Given a rectangle, we should try all possible cuts, and take the minimum of these choices. The base case is the time when the current rectangle has all same labels, which could be determined by V in $O(1)$ time. Denote the function $best_{cut}$(p,q) that will return the minimum cost to cut the rectangle given by M[p(x),...,q(x)][p(y),...,q(y)], we will have the following recurrence:

```
$best_{cut}$(p,q):
  if $x(p) > x(q)$ or $y(p) > y(q)$ then
    return $\infty$
  end if
  if $V[p,q] \neq BAD$ then
    return 0
  end if
  $\rho \leftarrow \infty$
  for $x = x(p)+1$ to $x(q)$ do
    $c \leftarrow price(p,q) + best_{cut}(p,((x-1),y(q))) + best_{cut}((x,y(p)),q)$
    if $c < \rho$ then
      $\rho \leftarrow c$
    end if
  end for
  for $y = y(p)+1$ to $y(q)$ do
    $c \leftarrow price(p,q) + best_{cut}(p,(x(q),(y-1))) + best_{cut}((x(p),y),q)$
    if $c < \rho$ then
      $\rho \leftarrow c$
    end if
  end for
  return $c$
```

Clearly the function $best_{cut}(p,q)$ contains at most $O((nm)^2)$ different recursive calls. So it takes $O((nm)^2)$ time to fill the dp table, of size $O((nm)^2)$. And the filling order will be starting from the rectangle with size $1,2,3,\ldots,nm$. The initial function call or the return value from the dp structure will be $best_{cut}((0,0),(m,n))$. Since each recursive call takes $O(n+m)$ time, the total runtime will be $O((nm)^2(n+m))$. ■

Solution: Solution that is similar to the above approach but did not have the $V[p,q]$ look-up-table will run in $O((nm)^2(nm)) = O((nm)^3)$. As every time the recursion need to spend $O(nm)$ time to check if all the cells on the board have the same label. Since there are a total of $O((nm)^2)$ function calls, the runtime will be $O((nm)^3)$. Correct solution that runs in $O((nm)^3)$ or other polynomial in the input parameters should be awarded at most 5 pts. ■

4. Assume we have a graph G with n vertices and m equal weight edges. We have two balls B_1, B_2 that will move along their designed path P_1, P_2, where P_1 has length N_1 and P_2 has length N_2. Both balls will start at $P_1[1]$ and $P_2[1]$, and can move only forward along their designed paths. To be more precise, we define a valid move for a ball is either stay in it's current node, or move to the next node on its path. Each ball makes exactly one legal move in each round. A sequence, specifying in each round a valid move for each robot, is a plan.

   The score of a plan is the maximum distance of the two balls from each other at any moment during the execution of the plan.

   Describe an algorithm that computes the minimum score plan for the two balls. Your algorithm should be as fast as possible.

Solution: We will build a look-up-table, denoted as $d_G(u,v)$, that will return the shortest distance between u and v. We can run BFS n times to build the value for the look-up-table. After that, the solution is a straightforward adaptation of the edit-distance DP. Let $l(i,j) = d_G(P_1[i], P_2[j])$, we can build the function minscore(i,j) that will return the minimum score plan for two balls if they are currently on position $P_1[i]$ and $P_2[j]$. And we have the following recurrence:
minscore(i,j) =

$$\begin{cases} l(N_1, N_2) & i = N_1 and j = N_2 \\ max(minscore(N_1, j+1), l(N_1, j)) & i = N_1 \\ max(minscore(i+1, N_2), l(i, N_2)) & j = N_2 \\ \alpha & otherwise \end{cases}$$

where $\alpha = max(min[minscore(i{+}1, j), minscore(i, j{+}1), minscore(i{+}1, j{+}1)], l(i,j))$. Since i and j are bounded by $O(N_1)$ and $O(N_2)$, there are at most $O(N_1 \cdot N_2)$ recursive function calls. We can use a 2-D array with size $O(N_1 \cdot N_2)$ for memoization. The filling order will be in decreasing i and decreasing j. And the value we are interested in is minscore(1,1). Since building the look-up-table l takes $O(nm)$ time, our total runtime will be $O(N_1 N_2 + nm)$. This problem is a variant of the discrete Fréchet distance problem. ∎