**COL351: Analysis and Design of Algorithms** **Fall 2024**

# Tutorial Sheet 9

Announced on: Sept 26 (Thurs)

- This tutorial sheet contains problems in dynamic programming (DP). When presenting your solutions, please define the DP table clearly. Don't forget to write the base case. The correctness and running time arguments can be brief (1-2 sentences).
- Problems marked with ($\star$) will not be asked in the tutorial quiz.

**Note:** For some of the problems, we've only provided the main ideas and not the full solutions. We expect you to fill the gaps by yourself. If there are any doubts, you can contact your tutorial TA, head TA or the instructor.

1. The SUBSET SUM problem is defined as follows: We are given as input $n$ nonnegative integers $a_1, \dots, a_n$ and a target integer $T$. The goal is to decide whether there exists a subset $S \subseteq \{a_1, a_2, \dots, a_n\}$ such that the entries of $S$ add up to exactly $T$. If the answer is YES, your algorithm must return the set $S$. Design an $\mathcal{O}(nT)$ time and $\mathcal{O}(T)$ space algorithm for this problem. Briefly justify your algorithm's correctness, running time, and space requirements.

**Informal Idea:** We use a dynamic programming (DP) approach to solve the SUBSET SUM problem. The goal is to find a subset of given nonnegative integers that sums to an exact target $T$. We define a boolean DP array $dp$ where $dp[t]$ indicates whether a subset sum of $t$ is achievable. Additionally, we use a *parent* array to track the first number that led to each new sum, facilitating an easy backtracking for subset reconstruction.
**Definitions:** Let $dp[t]$ represent whether it's possible to achieve a subset sum of $t$ using the first $j$ elements.

- **Base Case:** $dp[0] =$ True since a sum of zero is achievable with an empty subset.
- **Transition:** For each number $a_j$ in the input, update the DP array from right to left:

$$\text{If } dp[t-a_j] \text{ is True and } dp[t] \text{ is False, then:}$$

$$dp[t] = \text{True} \quad \text{and} \quad parent[t] = a_j$$

  if $t \geqslant a_j$.

If $dp[T]$ is True, there exists a subset with sum $T$. We then backtrack using the *parent* array to retrieve this subset.
**Algorithm:**

**ALGORITHM 1:** Subset Sum with Backtracking

```
Input: $n$ nonnegative integers $a_1, \dots, a_n$ and target integer $T$
Output: Subset $S$ such that $\sum_{x \in S} x = T$ or "No solution"
Declare: $dp$ array of size $T+1$ initialized to False, except $dp[0] = $ True
Declare: $parent$ array of size $T+1$ initialized to $-1$ for backtracking
for $j \leftarrow 1$ to $n$ do
    for $t \leftarrow T$ down to $a_j$ do
        if $dp[t - a_j]$ and not $dp[t]$ then
            $dp[t] \leftarrow$ True
            $parent[t] \leftarrow a_j$
if $dp[T] = True$ then
    Declare: empty set $S$
    $t \leftarrow T$
    while $t > 0$ do
        $S \leftarrow S \cup \{parent[t]\}$
        $t \leftarrow t - parent[t]$
    return $S$
else
    return "No solution"
```

**Correctness:** By only recording the first number that achieves each new sum, we create a unique path in the *parent* array for any achievable sum. This ensures the algorithm finds one valid subset summing to $T$, if it exists, and allows easy backtracking to construct the subset.

**Time Complexity:** The algorithm runs in $O(nT)$ time, as each number updates the DP array at most once for each sum up to $T$.

**Space Complexity:** The algorithm requires $O(T)$ space for the $dp$ and $parent$ arrays.

2. We are given a checkerboard that has four rows and $n$ columns and has an integer written in each square. We are also given a set of $2n$ pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) to maximize the sum of the integers in the squares that are covered by pebbles. For a placement of pebbles to be valid, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is allowed). Give an $\mathcal{O}(n)$ time algorithm to find an optimal placement of the pebbles.

**Informal Idea:** We will try to find the maximum sum of every valid pebble placement configuration for every column. **Note** There are only $2^4$ pebble placement configurations.
We define $dp[j][mask]$ as the maximum sum up to column $j$ where the pebble placement in column $j$ is defined by the bitmask *mask*. Each bit of *mask* represents whether a

pebble is placed on a specific row in column $j$ (1 if placed, 0 if not). The valid transitions depend on ensuring that no two pebbles are placed in adjacent squares (vertically or horizontally). We compute the sum for each valid configuration and transition from column $j-1$ to column $j$. The solution will be the maximum value of $dp[n-1][mask]$ for the last column. The recurrence relation will be as follows:

$$dp[j][mask] = \max(dp[j-1][mask'] + value(j, mask)) \text{ if } valid(mask, mask')$$

In the above recurrence $value(j, mask)$ function calculates the sum of integers in column $j$ for the rows where the corresponding bit in $mask$ is set to 1.
The $valid(mask, prev\_mask)$ function checks for conflicts between the current and previous column configurations, ensuring there are no adjacent pebbles.

**ALGORITHM 2:** Maximize Pebble Placement

```
Input: A 4-row by $n$-column checkerboard with integers $grid[4][n]$
Output: Maximum sum of integers with valid pebble placement
Declare: $dp$ array of size $(n+1) \times 16$, $placement$ array of size $(n+1) \times 16$ initialized to $-1$
Initialize: $dp[0][\text{mask}] = 0$ for all mask
for $j \leftarrow 1$ to $n$ do
    for $mask \leftarrow 0$ to 15 do
        Calculate $Value(j, \text{mask})$ by summing $grid[i][j]$ for rows $i$ where $\text{mask}[i] = 1$
        for $prev\_mask \leftarrow 0$ to 15 do
            if $valid(mask, prev\_mask)$ and $dp[j][mask] < dp[j-1][prev\_mask] + Value(j, mask)$ then
                $dp[j][\text{mask}] = dp[j-1][\text{prev\_mask}] + Value(j, \text{mask})$
                $placement[j][\text{mask}] = \text{prev\_mask}$
$max\_sum \leftarrow 0$, $best\_mask \leftarrow 0$
for $mask \leftarrow 0$ to 15 do
    if $dp[n][mask] > max\_sum$ then
        $max\_sum \leftarrow dp[n][\text{mask}]$
        $best\_mask \leftarrow \text{mask}$
Declare: $optimal\_placement$ array of size $n$
$current\_mask \leftarrow best\_mask$
for $j \leftarrow n$ down to 1 do
    $optimal\_placement[j] = current\_mask$
    $current\_mask \leftarrow placement[j][current\_mask]$
return $max\_sum$ and $optimal\_placement$
Function value(j, mask):
sum $\leftarrow 0$
for $i \leftarrow 0$ to 3 do
    if $mask \; \& \; (1 \ll i) \neq 0$ then
        sum $\leftarrow sum + grid[i][j]$
return sum
Function valid(mask, prev_mask):
if $(mask \; \& \; prev\_mask) \neq 0$ or $(mask \; \& \; (mask \ll 1)) \neq 0$ or $(mask \; \& \; (mask \gg 1)) \neq 0$ then
    return False
return True
```

**Time Complexity:** To calculate every $dp[j][mask]$ we require $O(16)$ time. Hence the total time required to calculate whole $dp$ table is $O(16 \cdot \text{sizeof}(dp)) = O(16^2 \cdot n) = O(n)$.

- Written "I do not know how to approach this problem" - 0.6 points
- 
  - Identifying that there are constant ways of placing pebbles in a column - 0.75 points
  - Correct Base Case - 0.25 points
  - Correct recurrence relation - 1 point
  - Brief justification of the recurrence relation - 0.25 points
  - Mentioning the correct order of filling the DP table - 0.25 points
  - Outputting the optimal placement (not just the optimal value) - 0.25 points
  - Brief justification of the time complexity - 0.25 points

3. Let $A$ be an $n \times n$ bit matrix, that is, every entry $A[i,j]$ is either 0 (white) or 1 (black). A submatrix $A[i_1 : i_2, j_1 : j_2]$ is a *black square* of size $s$ if (a) $s = i_2 - i_1 + 1 = j_2 - j_1 + 1$ and (b) $A[i,j] = 1$ for all $i_1 \leqslant i \leqslant i_2$ and $j_1 \leqslant j \leqslant j_2$. Design an $\mathcal{O}(n^2)$ time algorithm to find the largest black square submatrix of $A$.

**Informal Idea**: Let $dp[i][j]$ denote the maximum side length of a square with cell $(i,j)$ as the bottom-right corner if $A[i][j] = 1$. Let $col[i][j]$ be the maximum length of consecutive 1's in column $j$ ending at cell $(i,j)$, and $row[i][j]$ be the maximum length of consecutive 1's in row $i$ ending at cell $(i,j)$. We can use the following recurrence relation to calculate $dp[i][j]$:

$$dp[i][j] = \begin{cases} 1 & \text{if } (i = 0 \vee j = 0) \wedge A[i][j] = 1 \\ \min(\text{col}[i][j], \text{row}[i][j], dp[i-1][j-1] + 1) & \text{if } A[i][j] = 1 \\ 0 & \text{otherwise} \end{cases}$$

The size of the largest square submatrix would then be $\max(dp[i][j])$.

**ALGORITHM 3:** Find Largest Black Square Submatrix

**Input:** An $n \times n$ binary matrix $A$

**Output:** Top-left and bottom-right coordinates of the largest black square submatrix

```
Initialize: $dp$, $row$, and $col$ as $n \times n$ tables filled with 0;
Initialize: $max\_size = 0$;
Initialize: $top\_left = (-1, -1)$, $bottom\_right = (-1, -1)$;
for $i \leftarrow 0$ to $n - 1$ do
    for $j \leftarrow 0$ to $n - 1$ do
        if $A[i][j] = 1$ then
            if $j = 0$ then
                $row[i][j] = 1$;
            else
                $row[i][j] = row[i][j-1] + 1$;
            if $i = 0$ then
                $col[i][j] = 1$;
            else
                $col[i][j] = col[i-1][j] + 1$;
            if $i > 0$ and $j > 0$ then
                $dp[i][j] = \min(dp[i-1][j-1] + 1, col[i][j], row[i][j])$;
            else
                $dp[i][j] = 1$;
            if $dp[i][j] > max\_size$ then
                $max\_size = dp[i][j]$;
                $top\_left = (i - dp[i][j] + 1, j - dp[i][j] + 1)$;
                $bottom\_right = (i, j)$;
        else
            $dp[i][j] = 0$;
            $row[i][j] = 0$;
            $col[i][j] = 0$;
return $top\_left, bottom\_right$;
```

**Time Complexity:** To calculate every $dp[i][j]$ we require $O(1)$ time. Hence the total time required to calculate whole $dp$ table is $O(n^2)$.

- Written "I do not know how to approach this problem" - 0.6 points
- 
  - Correct base case for the recurrence - 0.25 points
  - Correct recurrence relation - 1.25 points
  - Brief justification of the recurrence relation - 0.5 points
  - Mentioning the correct order of filling the DP table - 0.25 points
  - Outputting the optimal submatrix (not just the optimal size) - 0.25 points

– Brief justification of the time complexity - 0.5 points

**Note:** 1 mark will be deducted from total points for an $O(n^3)$ time algorithm.

4. Consider the following inventory problem: You are running a store that sells some large product (let us assume you sell trucks), and predictions tell you the number of sales to expect over the next $n$ months. Let $d_i$ denote the number of sales you expect in month $i$. We will assume that all sales happen at the beginning of the month, and trucks that are not sold are stored until the beginning of the next month. You can store at most $S$ trucks, and it costs $C$ to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of $K$ each time you place an order (regardless of the number of trucks you order). You start with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $d_i$ and minimize the costs. In summary:

   - There are two parts to the cost. First, storage: It costs $C$ for every truck on hand that is not needed that month. Second, ordering fees: every order placed costs $K$.
   - In each month, you need enough trucks to satisfy the demand $d_i$, but the amount left over after satisfying the demand for the month should not exceed the inventory limit $S$.

   Give an algorithm that solves this problem in time that is polynomial in $n$ and $S$.

Let $dp[i][j]$ denote the minimum cost to meet demands from month 1 up to month $i$ with $j$ trucks in stock after selling expected trucks in month $i$. For each month $i$, given that we start with $j$ trucks in stock, we need $d[i]$ trucks to meet the demand. Let's assume we place an order for $q$ (where $\max(0, j + d[i] - S) \leqslant q \leqslant j + d[i]$) trucks such that the remaining trucks after satisfying demand are within the inventory limit $S$. The transition for each order size $q$ depends on storage cost of the trucks in stock and whether we placed order for $q > 0$ trucks. Then the base case and recurrence relation is as follows:
**Base Case:**

$$dp[0][0] = 0 \tag{1}$$
$$dp[0][j] = \infty \quad \text{for } j > 0 \tag{2}$$

**Recurrence Relation**:

$$dp[i][j] = \min_{\max(0,j+d[i]-S)\leqslant q\leqslant j+d[i]} \left(dp[i-1][j-q+d[i]] + K \cdot 1_{q>0} + C \cdot (j - q + d[i])\right) \tag{3}$$

In the above recurrence relation $1_{q>0}$ represents indicator function.

**ALGORITHM 4:** Inventory Cost Minimization

**Input:** Demands $d[1 \dots n]$, storage capacity $S$, storage cost per truck $C$, ordering cost $K$
**Output:** Minimum total cost to meet demands

```
Declare a $dp$ array of size $(n+1) \times (n+1)$
Initialize $dp[0][0] = 0$
for $j \leftarrow 1$ to $S$ do
    $dp[0][j] \leftarrow \infty$ // Impossible to start with any initial inventory
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 0$ to $S$ do
        $dp[i][j] \leftarrow \infty$
        for $q \leftarrow \max(0, j + d[i] - S)$ to $j + d[i]$ do
            $cost \leftarrow dp[i-1][j-q+d_i] + K \cdot 1_{q>0} + C \cdot (j - q + d[i])$
            $dp[i][j] \leftarrow \min(dp[i][j], cost)$
return $\min(dp[n][j]$ for $j = 0$ to $S)$ // Minimum cost for satisfying all demands
```

**Time Complexity:** To calculate every $dp[i][j]$ value we require $O(S)$ time. Hence the total time required to calculate whole $dp$ table is $O(nS^2)$.

5. This problem describes four generalizations of the knapsack problem. In each, the input consists of item values $v_1, v_2, \dots, v_n$, item sizes $s_1, s_2, \dots, s_n$, and additional problem-specific data (all positive integers). Which of these generalizations can be solved by dynamic programming in time polynomial in the number $n$ of items and the largest number $M$ that appears in the input? Mention all options that apply and provide a brief justification for your selection.

   a) Given a positive integer capacity $C$, compute a subset of items with the maximum possible total value subject to having total size *exactly* $C$. (If no such set exists, the algorithm should correctly detect that fact.)

   Yes, it can be solved in polynomial time using dynamic programming approach. Let $dp[s]$ represent the maximum total value achievable with a total size exactly $s$, where $s \in [0, C]$.

   **Base case:** $dp[0] = 0$ and $dp[s] = -\infty \; \forall s > 0$
   **Recurrence Relation:** For each item $i$, update:
   $dp[s + s_i] = max(dp[s + s_i], dp[s] + v_i) \; \forall s \; s.t. \; s + s_i \leqslant C$

   **Time complexity:** $O(nC)$, which is polynomial in $n$ and $C$

   **Do it yourself:** Compute the subset having the optimal value, and not just the optimal value. (For all parts)

   b) Given a positive integer capacity $C$ and an item budget $k \in \{1, 2, \dots, n\}$, compute a subset of items with the maximum possible total value subject to having total size at most $C$ and *at most $k$ items*.

Yes, it can be solved in polynomial time using dynamic programming approach. Let $dp[i][s][k']$ represent the maximum total value using the first $i$ items with total size $s$ and using at most $k'$ items.

**Recurrence Relation:** We describe the following cases:

i. If item $i$ is excluded: $dp[i][s][k'] = dp[i-1][s][k']$

ii. If item $i$ is included: If $s + s_i \leqslant C$ and $k' + 1 \leqslant k$:
$dp[i][s+s_i][k'+1] = max(dp[i][s+s_i][k'+1], dp[i-1][s][k'] + v_i)$

**Time complexity:** $O(nCk)$, which is polynomial in $n$ and $C$

c) Given capacities $C_1$ and $C_2$ of two knapsacks, compute disjoint subsets $S_1, S_2$ of items with the maximum possible total value $\sum_{i \in S_1} v_i + \sum_{i \in S_2} v_i$ subject to the knapsack capacities $\sum_{i \in S_1} s_i \leqslant C_1$ and $\sum_{i \in S_2} s_i \leqslant C_2$.

Yes, it can be solved in polynomial time using dynamic programming approach. Let $dp[i][s_1][s_2]$ represent the maximum total value using the first $i$ items, with total $s_1$ and $s_2$ used in knapsacks 1 and 2 respectively.

**Recurrence relation**: We describe the following cases:

i. If item $i$ is excluded: $dp[i][s_1][s_2] = dp[i-1][s_1][s_2]$

ii. Include item $i$ in knapsack 1: If $s_1 + s_i \leqslant C_1$:
$dp[i][s_1+s_i][s_2] = max(dp[i][s_1+s_i][s_2], dp[i-1][s_1][s_2] + v_i)$

iii. Include item $i$ in knapsack 2: If $s_2 + s_i \leqslant C_2$:
$dp[i][s_1][s_2+s_i] = max(dp[i][s_1][s_2+s_i], dp[i-1][s_1][s_2] + v_i)$

**Time complexity:** $O(nC_1C_2)$, which is polynomial in $n$, $C_1$ and $C_2$

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct base case for the recurrence - 0.25 points
  - Correct recurrence relation - 1.25 points
  - Brief justification of the recurrence relation - 0.5 points
  - Mentioning the correct order of filling the DP table - 0.25 points
  - Outputting the optimal subsets (not just the optimal value) - 0.25 points
  - Brief justification of the time complexity - 0.5 points

d) Given capacities $C_1, C_2, \ldots, C_m$ of $m$ knapsacks, where $m$ could be as large as $n$, compute disjoint subsets $S_1, S_2, \ldots, S_m$ of items with the maximum possible total value $\sum_{i \in S_1} v_i + \sum_{i \in S_2} v_i + \cdots + \sum_{i \in S_m} v_i$, subject to knapsack capacities $\sum_{i \in S_1} s_i \leqslant C_1$, $\sum_{i \in S_2} s_i \leqslant C_2, \ldots,$ $\sum_{i \in S_m} s_i \leqslant C_m$.

No, it cannot be solved in polynomial time.
With $m$ knapsacks potentially as large as $n$, the state space becomes impractically larger of the exponential order.
A naive DP approach is to define $dp[i][s_1][s_2]...[s_m]$ (an extension to the previous approach).
**Time complexity:** $O(nC_1C_2...C_m) = O(nM^m)$ making it exponential.

6. The following problems all take as input two strings $X$ and $Y$, with lengths $m$ and $n$, over some alphabet $\Sigma$. Which of them can be solved in $\mathcal{O}(mn)$ time? Mention all options that apply and provide a brief justification for your selection.

a) Consider the variation of sequence alignment in which, instead of a single gap penalty $\alpha_{\text{gap}}$, you are given two positive numbers $a$ and $b$. The penalty for inserting $k \geqslant 1$ gaps in a row is now defined as $ak + b$, rather than $k \cdot \alpha_{\text{gap}}$. The other penalties (for mismatching two symbols) are defined as before. The goal is to compute the minimum possible penalty of an alignment under this new cost model.

**Yes**, we can solve this variation of sequence alignment in $\mathcal{O}(mn)$ time. Let us recall the dp solution for the original sequence alignment problem.
$X_i :=$ first $i$ letters of $X$
$Y_j :=$ first $j$ letters of $Y$
$P_{i,j} :=$ optimal solution for $X_i$ and $Y_j$
**Recurrence:**

$$P_{i,j} = \min \begin{cases} \alpha_{x_i,y_j} + P_{i-1,j-1} \\ \alpha_{\text{gap}} + P_{i,j-1} \\ \alpha_{\text{gap}} + P_{i-1,j} \end{cases} \tag{4}$$

However this solution doesn't work for the modified version of the problem as last two cases we don't know if we should add $a$ or $b$ (as we don't store any information about how many gaps we have used up until now in the dp $P$). To fix this we can maintain two dps:
$Q_i :=$ optimal solution for $X_i$ and $Y_i$ without using any gaps.
$P_{i,j} :=$ optimal solution for $X_i$ and $Y_j$ using at least one gap.
**Base cases:**
$Q_0 = P_{0,0} = 0$
$P_{k,0} = P_{0,k} = ak + b$ for $k \geqslant 1$
**Recurrence:**

$$Q_i = \alpha_{x_i,y_j} + Q_{i-1}$$

$$P_{i,j} = \min \begin{cases} \alpha_{x_i,y_j} + P_{i-1,j-1} \\ a + P_{i,j-1} \\ a + P_{i-1,j} \\ a + b + Q_i \text{ (if } i = j-1) \\ a + b + Q_j \text{ (if } j = i-1) \end{cases} \tag{5}$$

Now if $n = m$ then the solution would be $\min(Q_n, P_{n,m})$, otherwise it is just $P_{n,m}$ (because we would have to use at least one gap).

b) Compute the length of the longest common subsequence of $X$ and $Y$. (A *subsequence* need not comprise consecutive symbols. For example, the longest common subsequence of "abcdef" and "afebcd" is "abcd".)

**Yes**, we construct a DP as follows:
$X_i$ := first $i$ letters of $X$
$Y_j$ := first $j$ letters of $Y$
$P_{i,j}$ := longest common subsequence for $X_i$ and $Y_j$.
**Base case**
$P_{i,0} = P_{0,j} = 0$
**Recurrence:**

$$P_{i,j} = \max \begin{cases} P_{i-1,j} \\ P_{i,j-1} \\ 1 + P_{i-1,j-1} \text{ (if } x_i = y_i) \end{cases} \tag{6}$$

The answer is simply $P_{n,m}$.

c) Assume that $X$ and $Y$ have the same length $n$. Determine whether there exists a permutation $f$, mapping each $i \in \{1, 2, \ldots, n\}$ to a distinct value $f(i) \in \{1, 2, \ldots, n\}$, such that $X_i = Y_{f(i)}$ for every $i \in \{1, 2, \ldots, n\}$.

**Yes.** We can solve this problem by maintaining the following DPs:
**Notation**

- $X_i$ : The multi-set of the first $i$ letters of $X$
- $Y_j$ : The multi-set of the first $j$ letters of $Y$
- $R_{i,\sigma}$ : smallest $j \geqslant i$ such that $y_j = \sigma$ (here $\sigma \in \Sigma$). If such an index doesn't exist then $R_{i,\sigma} = \perp$
- $Q_i$ : smallest $j$ such that $X_i \subseteq Y_j$, if no such $j$ exists then $Q_i = \perp$
- $P_i$ : $Y_{Q_i} \setminus X_i$. If $Q_i = \perp$ then $P_i = \phi$.

To construct $R$ we can run the following algorithm:

**ALGORITHM 5:** Construction of $R$

```
Input: $Y$
Output: $R$
$R_{i,\sigma} := \bot$ forall $i \in [n], \sigma \in \Sigma$
for $i \in [n]$ do
    $j := i$
    while $j \geqslant 0 \wedge y_j \neq y_i$ do
        $R_{j,y_i} := i$
        $j := j - 1$
```

**Base case**
$Q_0 = 0$
$P_0 = \phi$
**Recurrence:**

$$Q_i = \begin{cases} Q_{i-1} \text{ if } x_i \in P_{i-1} \\ R_{(Q_{i-1},x_i)} \text{ otherwise} \end{cases} \tag{7}$$

$$P_i = (P_{i-1} \cup (Y_{Q_i} \setminus Y_{Q_i - 1})) \setminus \{x_i\} \tag{8}$$

The answer is false if $Q_n = \bot$, otherwise true
**Time Complexity:** If we use lists as the data structures used to maintain $Q$ and $P$ then the algorithm will have a complexity of $\mathcal{O}(n^2)$. Bonus: what changes can you make to the algorithm to make it run in $\mathcal{O}(n)$?

d) Compute the length of the longest common substring of $X$ and $Y$. (A *substring* is a subsequence comprising consecutive symbols. For example, "bcd" is a substring of "abcdef", while "bdf" is not.)

**Yes**, we construct a DP as follows:
$P_{i,j}$ := longest common substring ending at $x_i$ and $y_j$
**Base case**
$P_{i,0} = P_{0,j} = 0$
**Recurrence:**

$$P_{i,j} = \begin{cases} 0 \text{ (if } x_i \neq y_j) \\ 1 + P_{i-1,j-1} \text{ (if } x_i = y_j) \end{cases} \tag{9}$$

The answer is max $P_{i,j}$ for $0 \leqslant i \leqslant n, 0 \leqslant j \leqslant m$.

7. ($\star$) We are given a set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ in the two-dimensional Euclidean plane, with $x_1 < x_2 < \cdots < x_n$. We will use $p_i$ to denote the point $(x_i, y_i)$.

Given a line $L$ defined by the equation $y = ax + b$, we say that the error of $L$ with respect to $P$

is the sum of its squared "distances" to the points in $P$, i.e.,

$$\texttt{error}(L, P) = \sum_{i=1}^{n} (ax_i + b - y_i)^2.$$

Using calculus, it can be shown that the line of best fit is given by:

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \text{ and } b = \frac{\sum_i y_i - a \sum_i x_i}{n}.$$

Our goal in this problem is twofold: First, we want to partition $P$ into some number of *segments*. A segment is a subset of $P$ that represents a contiguous set of $x$-coordinates; that is, it is a subset of the form $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for some indices $i \leqslant j$. Second, for each segment $S$ in our partition of $P$, we compute the line minimizing the error with respect to the points in $S$, according to the formulas above.

The penalty of a partition is defined to be a sum of the following terms.

a) The number of segments into which we partition $P$, times a fixed, given multiplier $C > 0$.

b) For each segment, the error value of the optimal line through that segment.

Design an efficient algorithm to find a partition of minimum penalty.

**Solution Sketch:** Let $dp[i]$ denote the penalty of optimal partition of the points $p_1, p_2, \ldots, p_i$. Also, let $cost(j, i)$ $(j \leqslant i)$ denote the error value of the optimal line through the segment $p_j, p_{j+1}, p_{j+2}, \ldots p_i$.

Then, the following relation holds:

$$dp[1] = C \text{ (No error if just one point)}$$

$$dp[i] = \min \begin{cases} \min\limits_{2 \leqslant j \leqslant i} (C + cost(j, i) + dp[j-1]) \\ C + cost(1, i) \end{cases}$$

The above recurrence relation holds since we take minimum over all possible segments of $i$. Note that $cost[j, i]$ can be computed in polynomial time and hence, the penalty of optimal partition in polynomial time.

**Do it Yourself:** Find the optimal partition and not just the penalty of optimal partition. Also, try to see the time complexity of the above algorithm. Assume multiplication, division to be $O(1)$ time operations.