

Name: Harlee Ramos

Due Date: 06/07/2024

1. Consider the following approach to the matrix-chain multiplication problem: Whenever there are at least 3 matrices $(A_i \dots A_j, j - i \geq 2)$ remaining, always split the chain at $k: (A_i \dots A_k)(A_{k+1} \dots A_j)$, such that p_k is the smallest number among $\{p_i, \dots, p_{j-1}\}$. Prove that this algorithm doesn't always give the optimal way to calculate the product.

According to the book's algorithm, the matrix-chain multiplication problem involves choosing the most efficient way to multiply a given sequence of matrices. The objective is to reduce the number of scalar multiplications. The suggested technique separates the chain at the matrix with the least dimension in terms of p_k , resulting in overlapping subproblems that are addressed several times in larger problems.

MATRIX-CHAIN-ORDER(p)

```

1   $n = p.length - 1$ 
2  let  $m[1..n, 1..n]$  and  $s[1..n - 1, 2..n]$  be new tables
3  for  $i = 1$  to  $n$ 
4       $m[i, i] = 0$ 
5  for  $l = 2$  to  $n$  //  $l$  is the chain length
6      for  $i = 1$  to  $n - l + 1$ 
7           $j = i + l - 1$ 
8           $m[i, j] = \infty$ 
9          for  $k = i$  to  $j - 1$ 
10              $q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
11             if  $q < m[i, j]$ 
12                  $m[i, j] = q$ 
13                  $s[i, j] = k$ 
14  return  $m$  and  $s$ 
```

This algorithm involves overlapping subproblems. That is, the same subproblems are solved multiple times in solving larger problems. Dynamic programming avoids redundant computations by storing the results of these subproblems in a table (memorization) and reusing them when needed.

Example

A_1 55 x 100	A_2 100 x 60	A_3 60 x 40	A_4 40 x 200
$p_0 = 55$	$p_1 = 100$	$p_2 = 60$	$p_3 = 40$
			$p_4 = 200$

Split at k where p_k is the smallest among $\{p_0, p_1, p_2, p_3, p_4\}$, Optimal split at k where $p_k = 2$, middle value of all p_k 's in this case $p_3 = 40$ is the smallest, then the splits happen at $k = 3$.

$A_3 \times A_4$ Cost: $p_2 \cdot p_3 \cdot p_4 = 60 \cdot 40 \cdot 200 = 480000$ Result matrix dimension 60×200	$A_2 \times A_3$ Cost: $p_1 \cdot p_2 \cdot p_3 = 100 \cdot 60 \cdot 40 = 240000$ Result matrix dimension 100×40
$A_2 \times (A_3 \times A_4)$ Cost: $p_1 \cdot p_2 \cdot p_4 = 100 \cdot 60 \cdot 200 = 1200000$ Result matrix dimension 100×200	$A_1 \times (A_2 \times A_3)$ Cost: $p_0 \cdot p_1 \cdot p_3 = 55 \cdot 100 \cdot 40 = 220000$ Result matrix dimension 55×40
$A_1 \times (A_2 \times (A_3 \times A_4))$ Cost: $p_0 \cdot p_1 \cdot p_4 = 55 \cdot 100 \cdot 200 = 1100000$ Total cost: $480000 + 1200000 + 1100000 = 2780000$	$(A_1 \times (A_2 \times A_3)) \times A_4$ Cost: $p_1 \cdot p_2 \cdot p_4 = 55 \cdot 40 \cdot 200 = 440000$ Total cost: $240000 + 220000 + 440000 = 900000$

The following example demonstrates that constantly separating the chain at the least p_k value does not yield an optimal solution, as the total cost of the first approach is higher than the total cost of choosing an optimal p_k . Therefore, we need a more advanced approach like dynamic programming to determine the optimal solution for the matrix-chain multiplication problem.

In this case, dynamic programming is required for the matrix-chain multiplication problem to ensure that we find the optimal solution that minimizes the total number of scalar multiplications. This is because it avoids redundant computations by storing the results of these subproblems in a table memorization and reusing them when needed

2. Given a set of k positive integers $\{a_1, a_2, \dots, a_k\}$ that add up to N . Note that, there might exist $1 \leq i \neq j \leq k$ such that $a_i = a_j$ (in other words, these k integers might not be unique). Is there a subset $B \subseteq \{1, 2, \dots, k\}$ such that the follow equation is satisfied?

$$\sum_{i \in B} a_i = \sum_{i \in \{1, 2, \dots, k\} \setminus B} a_i$$

a) If N is an odd number, what can we say about the problem?

This problem is a subset-sum scenario. It consists of a set of k positive integers $\{a_1, a_2, \dots, a_k\}$ that add up to N , which is the target total. The task is to identify whether any subset of the provided integers sums up to the target the sum.

Now if we consider the N as an odd number, then $\frac{N}{2}$, is not an integer value. Since the subset sums must be integers (because they are sums of integers), it is impossible to divide the set into two subsets with equal sums.

Therefore, there cannot exist such a subset B when N is odd. On the other hand, if we consider the N as an even number, then $\frac{N}{2}$, is an integer. In this case, we need to determine if there exists a subset B such that the sum of the elements in B is equal to $\frac{N}{2}$.

Example, we have the set $\{2, 1, 5, 3, 4\}$, and two target sums N .

Even number

$N = 6$

$N = 2+1+3 = 6$

$$\frac{N}{2} = \frac{6}{2} = 3$$

$$\sum_{i \in B} a_i = 7$$

Finding a subset for B

$$\sum_{i \in B} a_i = 2 + 1 + 3 = 7 \quad \text{Subset } B\{1, 2, 3\}$$

$$\sum_{i \in \{1, 2, \dots, k\} \setminus B} a_i = 1 + 2 + 3 = 6$$

This shows that when N is even, it is possible to find such subsets.

Odd number

$N = 7$

$N = 2+5 = 7$

$$\frac{N}{2} = \frac{7}{2} = 3.5$$

$$\sum_{i \in B} a_i = 7.5$$

Since 7.5 is not an integer, it's impossible to find a subset B such that the summation is equal to 3.5. Hence, no solution exists.

When N is an odd number, we can conclude that the problem is unsolvable. This is because the total of the integers in any subset cannot equal the sum of the integers in the complementary subset, as one sum would be half of an odd number, which is impossible because integer sums are always whole numbers.

b) From now on, we assume that N is an even number. Try to create a recursive function $b(i, j)$ that represents the Boolean value whether there is a subset in $\{a_1, a_2, \dots, a_i\}$ that add up to j .

$b(i, j)$: Boolean

if $j == 0$: return True // Empty subset sums to 0

if $i == 0$: return False // No elements to form sum j

return $b(i-1, j)$ OR $b(i-1, j-a_i)$ // Include or exclude the i th element

In the line return $b(i-1, j)$ OR $b(i-1, j-a_i)$, the function will check if there exists a subset in $\{a_1, a_2, \dots, a_i\}$ that adds up to j , by considering two cases:

- Exclude the i th element: Use $b(i-1, j)$ to see if the sum j can be achieved without the i th element.
- Include the i th element: Use $b(i-1, j-a_i)$ to see if the sum $(j - a_i)$ can be achieved with the remaining elements and then add a_i to reach j .

c) Create an algorithm using dynamic programming to solve this problem. You may use pseudo-code, or you may answer the following questions:

This algorithm SUBSET-SUM will initialize a dp table of size $(k+1) \times (sum+1)$, then it will fill the dp table iteratively to determine if a subset of the given elements can sum up to the total sum.

SUBSET-SUM (a, k, sum)

```
1. let dp[0..k, 0..sum] be a new table          // Create a DP table of size (k+1) x (sum+1)
2. for i = 0 to k
3.   dp[i, 0] = TRUE                             // Initialize the first column as TRUE (sum of 0 can always be achieved)
4.   for j = 1 to sum
5.     dp[0, j] = FALSE                          // Initialize the first row as FALSE (no elements to achieve positive sums)
6.   for i = 1 to k
7.     for j = 1 to sum
8.       dp[i, j] = dp[i-1, j]                  // Copy the value from the previous row (excluding the current element)
9.       if j >= a[i-1]                          // Check if the current element can be included
10.        dp[i, j] = dp[i, j] OR dp[i-1, j - a[i-1]] // Include the current element if possible
11. return dp[k, sum]                          // Return the value indicating if the target sum can be achieved
```

a. How to create a memo to record the value of each $b(i, j)$?

To create a memo to record the value, a dp 2D boolean array of size $(k+1) \times (sum+1)$, is required to store the precomputed results of $b(i, j)$. $dp[i][j]$ represents whether a subset in $\{a_1, a_2, \dots, a_i\}$ sums to j .

To better explain the memo's creation, I will use the set to achieve a target sum of 6. The following memo includes the possible values in each column based on the elements available in each k-row.

Then, we will perform the sum on each spot and check to see if it equals the target value. If the value is equal, we will store 'T' for "TRUE"; otherwise, we will store 'F' for "FALSE".

$k = 5$

$sum = 6$

Dimensions of memo matrix $(k+1) \times (sum+1) = (5+1) \times (6+1) = 6 \times 7$

At the first stage we can fill T on the 1st column, because this condition is meet, **for $i = 0$ to k . Then, $dp[i, 0] = TRUE$**

At the 1st row we can fill F, because this condition is meet **for $j = 1$ to S . Then, $dp[0, j] = FALSE$**

sum \ k	0	1	2	3	4	5	6
0	T↓	F	F	F	F	F	F
1	T↓						
2	T↓						
3	T↓						
4	T↓						
5	T						

Starting from the 2nd stage we must consider the following conditions:

for $i = 1$ to k for $j = 1$ to S
 $dp[i, j] = dp[i-1, j]$

Examples

Cell [1,1]

$dp[i, j] = dp[i-1, j] \rightarrow dp[1, 1] = dp[1-1, 1] = dp[0, 1] = T$

Cell [1,2]

$dp[i, j] = dp[i-1, j] \rightarrow dp[1, 2] = dp[1-1, 2] = dp[0, 2] = F$

sum \ k	0	1	2	3	4	5	6
0	↓T↓	F	F↓	F↓	F↓	F↓	F↓
1	T↓	T	F	F	F	F	F
2	T↓						
3	T↓						
4	T↓						
5	T						

From this stage we must consider all conditions of the pseudocode.

Examples

Cell [2,2]

$= dp[2-1, 2] = [1, 2]$ ***At this point $j > a[i-1]$**

$dp[i, j] = dp[i, j]$ OR $dp[i-1, j - a[i-1]]$

$= [1, 1]$ OR $[1, 2-(2)]$

$= [1, 1]$ OR $[1, 0] = T$ OR $F = T$

Cell [2,4]

$= dp[2-1, 4] = [1, 4]$ ***At this point $j > a[i-1]$**

$dp[i, j] = dp[i, j]$ OR $dp[i-1, j - a[i-1]]$

$= [1, 4]$ OR $[1, 4-(2)]$

$= [1, 4]$ OR $[1, 2] = F$ OR $F = F$

sum \ k	0	1	2	3	4	5	6
0	↓T↓	F	F↓	F↓	F↓	F↓	F↓
1	T↓	↓T↓	F↓	F↓	F↓	F↓	F↓
2	T↓	T	T OR F T	F OR T T	F OR F F	F	F
3	T↓						
4	T↓						
5	T						

Examples

Cell [3,3]

```
= dp[3-1, 3]=[2,3]    *At this point j>=a[i-1]
dp[i, j] = dp[i, j] OR dp[i-1, j - a[i-1]]
=[2,3] OR [2,3-(3)]
=[2,3] OR [2,0]
= F OR T = T
```

Cell [3,6]

```
= dp[3-1, 6]=[2,6]    *At this point j>=a[i-1]
dp[i, j] = dp[i, j] OR dp[i-1, j - a[i-1]]
=[2,6] OR [2,6-(3)]
=[2,6] OR [2,3]
= F OR T = T
```

sum k \	0	1	2	3	4	5	6
0	↓T↓	F	F↓	F↓	F↓	F↓	F↓
1	T↓	↓T↓	F↓	F↓	F↓	F↓	F↓
2	T↓	↓T	↓T	↓T	F	F	F
3	T↓	T	T	F OR T T	F OR T T	F OR T T	F OR T T
4	T↓						
5	T						

Examples

Cell [4,4]

```
= dp[4-1, 4]=[3,4]    *At this point j>=a[i-1]
dp[i, j] = dp[i, j] OR dp[i-1, j - a[i-1]]
=[3,4] OR [3,4-(4)]
=[3,4] OR [3,0]
= T OR T = T
```

Cell [4,6]

```
= dp[4-1, 6]=[3,6]    *At this point j>=a[i-1]
dp[i, j] = dp[i, j] OR dp[i-1, j - a[i-1]]
=[3,6] OR [3,6-(4)]
=[3,6] OR [3,2]
= F OR T = T
```

sum k \	0	1	2	3	4	5	6
0	↓T↓	F	F↓	F↓	F↓	F↓	F↓
1	T↓	↓T↓	F↓	F↓	F↓	F↓	F↓
2	T↓	↓T	↓T	↓T	F	F	F↓
3	T↓	↓T	↓T	↓T	↓T	↓T	↓T
4	T↓	T	T	T	T	F OR T T	F OR T T
5	T						

Examples

Cell [5,5]

```
= dp[5-1, 5]=[4,5]    *At this point j>=a[i-1]
dp[i, j] = dp[i, j] OR dp[i-1, j - a[i-1]]
=[4,5] OR [4,5-(5)]
=[4,5] OR [4,0]
= T OR T = T
```

Cell [5,6]

```
= dp[5-1, 6]=[4,6]    *At this point j>=a[i-1]
dp[i, j] = dp[i, j] OR dp[i-1, j - a[i-1]]
=[4,6] OR [4,6-(5)]
=[4,6] OR [4,1]
= T OR T = T
```

sum k \	0	1	2	3	4	5	6
0	↓T↓	F	F↓	F↓	F↓	F↓	F↓
1	T↓	↓T↓	F↓	F↓	F↓	F↓	F↓
2	T↓	↓T	↓T	↓T	F	F	F↓
3	T↓	↓T	↓T	↓T	↓T	↓T	↓T
4	T↓	↓T	↓T	↓T	↓T	↓T	↓T
5	T	T	T	T	T	T	T

b. How to use the created memo to find the solution to the problem?

After filling the table, check $dp[k][sum]$, if that cell is True, there exists a subset that partitions the set. To find the actual subset that sums to the target, it is needed to backtrack through the table.

$dp[5][6] \rightarrow$ then $dp[5][6] == dp[4][5]$

From $dp[4][5] \rightarrow$ then $dp[4][5] == dp[3][4]$

From $dp[3][4] \rightarrow$ then $dp[3][4] == dp[3][3]$ *At this point Add 3 to the subset: subset = [3].

From $dp[3][3] \rightarrow$ then $dp[3][3] == dp[2][2]$ *At this point Add 2 to the subset: subset = [3,2].

From $dp[2][2] \rightarrow$ then $dp[2][2] == dp[1][1]$ *At this point Add 1 to the subset: subset = [3,2,1].

From $dp[1][1] \rightarrow$ then $dp[1][1] == dp[0][0]$

Then the final subset: [3, 2, 1] sums to 6.

sum k \	0	1	2	3	4	5	6
0	↓T↓	F	F↓	F↓	F↓	F↓	F↓
1	T↓	↓T↓	F↓	F↓	F↓	F↓	F↓
2	T↓	↓T	↓T	↓T	F	F	F↓
3	T↓	↓T	↓T	↓T	↓T	↓T	↓T
4	T↓	↓T	↓T	↓T	↓T	↓T	↓T
5	T	T	T	T	T	T	T

d) What is the time complexity of your dynamic programming algorithm?

The dynamic programming approach to the subset-sum algorithm has a time complexity of $O(k \cdot sum)$, because the initialization of the dp table takes $O(k \cdot sum)$. The complexity comes from the necessity to fill a table of size $(k+1) \times (sum+1)$ using nested loops that run over every single element and possible sums. The initialization time complexity is $O(k) + O(sum)$, whereas filling the dp matrix is $O(k \cdot sum)$. Because $O(k \cdot sum)$ surpasses $O(k + sum)$, the algorithm's overall time complexity is $O(k \cdot sum)$. This time complexity is efficient for moderate values of k and sum , but it may become unsuitable for very large inputs.

3. A subsequence is a *palindrome* if it is the same when read left to right and right to left. A subsequence does not have to be contiguous. How to find the longest subsequence which is a palindrome in the given string $A = a_1a_2 \dots a_n$? For example, the string *abcab* contains four palindromes of length 3 as subsequence: *aba*, *aca*, *bcb*, and *bab*, but no palindrome of length 4; thus, any palindrome of length 3 is an optimal solution.

a) Create a recursive function $L(i, j)$ that represents the length of the longest subsequence which is a palindrome in substring $a_i \dots a_j$

In the following pseudocode of the function $L(i, j)$, some considerations were taken.

- The base case is the line 1, If $i = j$, then $L(i, j) = 1$, because a single character is always a palindrome of length 1, otherwise if $i > j$, then it will return 0.
- On line 5, we have the situation of if $A[i] == A[j]$, that is when there are two characters and both are the same, they will form a palindrome of length 2. $L(i + 1, j - 1) + 2$. This is because the characters a_i and a_j form a palindromic pair and it is required to add 2 to the length of the longest palindromic subsequence in the substring $a_{i+1} \dots a_{j-1}$.
- When the case of line 5 is not met, then the characters do not match $a_i \neq a_j$, then we must take the maximum length from either excluding the current start character or the current end character $\max(L(i + 1, j), L(i, j - 1))$.

```

L(i, j)
1  if i == j
2    return 1
3  if i > j
4    return 0
5  if A[i] == A[j]
6    return L(i + 1, j - 1) + 2
7  else
8    return max(L(i + 1, j), L(i, j - 1))

```

b) Create an algorithm using dynamic programming to solve this problem. You may use pseudocode, or you may answer the following questions:

The pseudocode below of LPS, which stands for Longest Palindrome Subsequence implements dynamic programming in the following way:

- Lines 1-2 set up a 2D array dp with size $n \times n$. Lines 3-4 set each diagonal element $dp[i][i]$ to 1 as a single character is a palindrome of length 1.
- Lines 5-7 iterate over the substring lengths ranging from 2 to n , and for each substring length l , we iterate over the starting index i and determine the ending index j .
- From lines 8 to 9, if $A[i] == A[j]$, the letters form a palindromic pair, and we add 2 to the length of the largest palindromic subsequence found in the substring $A[i+1..j-1]$.
- Finally, on lines 10-12, if $A[i] \neq A[j]$, we calculate the maximum length by excluding either the current start or end character.

```

LPS(A)
1  n = length(A)
2  let dp[1..n][1..n] be a new table
3  for i = 1 to n
4    dp[i][i] = 1
5  for l = 2 to n // l is the length of the substring
6    for i = 1 to n - l + 1
7      j = i + l - 1
8      if A[i] == A[j]
9        dp[i][j] = dp[i + 1][j - 1] + 2
10     else
11       dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])
12  return dp[1][n]

```

a. How to create a memo to record the value of each $L(i, j)$?

To create the memo, I will use the following example with the string 'abcab'.

Steps:

- Initialization: Initialize a 2D array dp of size $n \times n$.
- Fill the diagonal with 1s because each single character is a palindrome of length 1.
- Moving forward, continue filling the table diagonally using recursive relations.
- The table must be filled with 0 in the cells when the comparison of the strings requires it to go backwards.

Steps					dp Matrix																																																						
Array A					<div>dp</div> <table><tr><th></th><th></th><th>a</th><th>b</th><th>c</th><th>a</th><th>b</th></tr><tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th></th></tr><tr><th>a</th><th>0</th><td>1</td><td></td><td></td><td></td><td></td></tr><tr><th>b</th><th>1</th><td>0</td><td>1</td><td></td><td></td><td></td></tr><tr><th>c</th><th>2</th><td>0</td><td>0</td><td>1</td><td></td><td></td></tr><tr><th>a</th><th>3</th><td>0</td><td>0</td><td>0</td><td>1</td><td></td></tr><tr><th>b</th><th>4</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>								a	b	c	a	b		0	1	2	3	4		a	0	1					b	1	0	1				c	2	0	0	1			a	3	0	0	0	1		b	4	0	0	0	0	1
		a	b	c	a	b																																																					
	0	1	2	3	4																																																						
a	0	1																																																									
b	1	0	1																																																								
c	2	0	0	1																																																							
a	3	0	0	0	1																																																						
b	4	0	0	0	0	1																																																					
The LPS for 1 character will be 1, thus $L(i,j)=1$.																																																											
From this step we can consider. $j = i + 2 - 1$																																																											
Example Cell[0][1] A[0] = 'a', A[1] = 'b' A[0] \neq A[1] *At this point condition is meet Else $dp[i][j] = \max(dp[i + 1][j], dp[i][j - 1])$ $dp[0][1]= \max(dp[1][1], dp[0][0])$ $dp[0][1]= (1 , 1) \rightarrow 1$					Example Cell[3][4] A[3] = 'a', A[4] = 'b' A[4] \neq A[1] *At this point condition is meet Else $dp[i][j] = \max(dp[i + 1][j], dp[i][j - 1])$ $dp[3][4]= \max(dp[4][4], dp[3][3])$ $dp[3][4]= \max(1 , 1) \rightarrow 1$					<div>dp</div> <table><tr><th></th><th></th><th>a</th><th>b</th><th>c</th><th>a</th><th>b</th></tr><tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th></th></tr><tr><th>a</th><th>0</th><td>1</td><td>1</td><td></td><td></td><td></td></tr><tr><th>b</th><th>1</th><td>0</td><td>1</td><td>1</td><td></td><td></td></tr><tr><th>c</th><th>2</th><td>0</td><td>0</td><td>1</td><td>1</td><td></td></tr><tr><th>a</th><th>3</th><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><th>b</th><th>4</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>			a	b	c	a	b		0	1	2	3	4		a	0	1	1				b	1	0	1	1			c	2	0	0	1	1		a	3	0	0	0	1	1	b	4	0	0	0	0	1
		a	b	c	a	b																																																					
	0	1	2	3	4																																																						
a	0	1	1																																																								
b	1	0	1	1																																																							
c	2	0	0	1	1																																																						
a	3	0	0	0	1	1																																																					
b	4	0	0	0	0	1																																																					
From this step we can consider. $j = i + 3 - 1$																																																											
Example Cell[0][2] A[0] = 'a', A[2] = 'c' A[0] \neq A[2] *At this point condition is meet Else $dp[i][j] = \max(dp[i + 1][j], dp[i][j - 1])$ $dp[0][2]= \max(dp[1][2], dp[0][1])$ $dp[0][2]= \max(1 , 1) \rightarrow 1$					Example Cell[2][4] A[2] = 'c', A[4] = 'b' A[2] \neq A[4] *At this point condition is meet Else $dp[i][j] = \max(dp[i + 1][j], dp[i][j - 1])$ $dp[2][4]= \max(dp[3][4], dp[2][3])$ $dp[2][4]= \max(1 , 1) \rightarrow 1$					<div>dp</div> <table><tr><th></th><th></th><th>a</th><th>b</th><th>c</th><th>a</th><th>b</th></tr><tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th></th></tr><tr><th>a</th><th>0</th><td>1</td><td>1</td><td>1</td><td></td><td></td></tr><tr><th>b</th><th>1</th><td>0</td><td>1</td><td>1</td><td>1</td><td></td></tr><tr><th>c</th><th>2</th><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr><tr><th>a</th><th>3</th><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><th>b</th><th>4</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>			a	b	c	a	b		0	1	2	3	4		a	0	1	1	1			b	1	0	1	1	1		c	2	0	0	1	1	1	a	3	0	0	0	1	1	b	4	0	0	0	0	1
		a	b	c	a	b																																																					
	0	1	2	3	4																																																						
a	0	1	1	1																																																							
b	1	0	1	1	1																																																						
c	2	0	0	1	1	1																																																					
a	3	0	0	0	1	1																																																					
b	4	0	0	0	0	1																																																					
From this step we can consider. $j = i + 4 - 1$																																																											
Cell[0][3] A[0] = 'a', A[3] = 'a' A[0] == A[3] *At this point condition is meet if $A[i] == A[j]$ $dp[i][j] = dp[i + 1][j - 1] + 2$ $dp[0][3] = dp[1][2] + 2$ $dp[0][3] = 1+2=3$					Cell[1][4] A[1] = 'b', A[4] = 'b' A[1] == A[4] *At this point condition is meet if $A[i] == A[j]$ $dp[i][j] = dp[i + 1][j - 1] + 2$ $dp[1][4] = dp[2][3] + 2$ $dp[0][3] = 1+2=3$					<div>dp</div> <table><tr><th></th><th></th><th>a</th><th>b</th><th>c</th><th>a</th><th>b</th></tr><tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th></th></tr><tr><th>a</th><th>0</th><td>1</td><td>1</td><td>1</td><td>3</td><td></td></tr><tr><th>b</th><th>1</th><td>0</td><td>1</td><td>1</td><td>1</td><td>3</td></tr><tr><th>c</th><th>2</th><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr><tr><th>a</th><th>3</th><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><th>b</th><th>4</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>			a	b	c	a	b		0	1	2	3	4		a	0	1	1	1	3		b	1	0	1	1	1	3	c	2	0	0	1	1	1	a	3	0	0	0	1	1	b	4	0	0	0	0	1
		a	b	c	a	b																																																					
	0	1	2	3	4																																																						
a	0	1	1	1	3																																																						
b	1	0	1	1	1	3																																																					
c	2	0	0	1	1	1																																																					
a	3	0	0	0	1	1																																																					
b	4	0	0	0	0	1																																																					
From this step we can consider. $j = i + 5 - 1$																																																											
Example Cell[0][4] A[0] = 'a', A[4] = 'b' A[0] \neq A[4] *At this point condition is meet Else $dp[i][j] = \max(dp[i + 1][j], dp[i][j - 1])$ $dp[0][4]= \max(dp[1][4], dp[0][3])$ $dp[0][4]= (3 , 3) \rightarrow 3$										<div>dp</div> <table><tr><th></th><th></th><th>a</th><th>b</th><th>c</th><th>a</th><th>b</th></tr><tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th></th></tr><tr><th>a</th><th>0</th><td>1</td><td>1</td><td>1</td><td>3</td><td>3</td></tr><tr><th>b</th><th>1</th><td>0</td><td>1</td><td>1</td><td>1</td><td>3</td></tr><tr><th>c</th><th>2</th><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr><tr><th>a</th><th>3</th><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><th>b</th><th>4</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>			a	b	c	a	b		0	1	2	3	4		a	0	1	1	1	3	3	b	1	0	1	1	1	3	c	2	0	0	1	1	1	a	3	0	0	0	1	1	b	4	0	0	0	0	1
		a	b	c	a	b																																																					
	0	1	2	3	4																																																						
a	0	1	1	1	3	3																																																					
b	1	0	1	1	1	3																																																					
c	2	0	0	1	1	1																																																					
a	3	0	0	0	1	1																																																					
b	4	0	0	0	0	1																																																					

b. So that one can find the longest subsequence, which is a palindrome in A , what extra info is needed to be recorded in the memo?

- Record the choices made during the computation. In another table $s[i][j]$ that records the direction of the optimal subproblem.
- Create an algorithm CONSTRUCT-LPS for reveal the path of the subsequence, in which we can take the following considerations:
 - if $A[i] == A[j]$ $s[i][j] = (i+1, j-1)$
 - if $dp[i+1][j] \geq dp[i][j-1]$ $s[i][j] = (i+1, j)$
 - if $dp[i][j-1] > dp[i+1][j]$ $s[i][j] = (i, j-1)$

c. How to use the created memo to find an optimal solution to the problem?

The pseudocode CONSTRUCT-LPS takes as arguments array A , matrixes dp and s to constructs the LPS. It initializes indices i and j to the length of the string A , sets up pointers to start from both ends, and initializes an empty list lps to store the characters of the LPS.

The loop runs while i is less than or equal to j , checking for identical characters at indices i and j . If $A[i] == A[j]$, append $A[i]$ to the lps list, incrementing i to move the left pointer to the right and diminishing j to move the right pointer to the left.

The decision-making process begins at line 8, using the stored decision stored in $s[i][j]$. If $s[i][j] == (i+1, j)$, increment i , decrement j , and add $A[i]$ to the LPS list. Finally, line 12 combines the lps list with its reverse to form the final palindromic sequence, as the lps list currently contains only one half of the LPS.

dp matrix of LPS algorithm

dp		a	b	c	a	b
		0	1	2	3	4
a	0	1	1	1	3	3
b	1	0	1	1	1	3
c	2	0	0	1	1	1
a	3	0	0	0	1	1
b	4	0	0	0	0	1

CONSTRUCT-LPS(A , dp , s)

```

1  i = 1, j = length(A)
2  lps = empty list
3  while i <= j
4    if A[i] == A[j]
5      append A[i] to lps
6      i = i + 1
7      j = j - 1
8    else if s[i][j] == (i+1, j)
9      i = i + 1
10   else
11     j = j - 1
12   return lps + reverse(lps)

```

CONSTRUCT-LPS. Applying the backward algorithm to find the lps of length 3.

Iter	<div>LPS 'bcb'</div> <table><tr><td></td><td></td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td></tr><tr><td>s</td><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>a</td><td>0</td><td>-</td><td>-</td><td>-</td><td>-</td><td>↓</td></tr><tr><td>b</td><td>1</td><td>-</td><td>-</td><td>-</td><td>-</td><td>↘</td></tr><tr><td>c</td><td>2</td><td>-</td><td>-</td><td>↙</td><td>←</td><td>-</td></tr><tr><td>a</td><td>3</td><td>-</td><td>*</td><td>-</td><td>-</td><td>-</td></tr><tr><td>b</td><td>4</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr></table>			a	b	c	a	b	s		0	1	2	3	4	a	0	-	-	-	-	↓	b	1	-	-	-	-	↘	c	2	-	-	↙	←	-	a	3	-	*	-	-	-	b	4	-	-	-	-	-	<div>LPS 'aba'</div> <table><tr><td></td><td></td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td></tr><tr><td>s</td><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>a</td><td>0</td><td>-</td><td>-</td><td>-</td><td>↙</td><td>←</td></tr><tr><td>b</td><td>1</td><td>-</td><td>*</td><td>←</td><td>-</td><td>-</td></tr><tr><td>c</td><td>2</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr><tr><td>a</td><td>3</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr><tr><td>b</td><td>4</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr></table>			a	b	c	a	b	s		0	1	2	3	4	a	0	-	-	-	↙	←	b	1	-	*	←	-	-	c	2	-	-	-	-	-	a	3	-	-	-	-	-	b	4	-	-	-	-	-	<div>LPS 'aca'</div> <table><tr><td></td><td></td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td></tr><tr><td>s</td><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>a</td><td>0</td><td>-</td><td>-</td><td>-</td><td>↙</td><td>-</td></tr><tr><td>b</td><td>1</td><td>-</td><td>-</td><td>↓</td><td>-</td><td>-</td></tr><tr><td>c</td><td>2</td><td>-</td><td>-</td><td>*</td><td>-</td><td>-</td></tr><tr><td>a</td><td>3</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr><tr><td>b</td><td>4</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr></table>			a	b	c	a	b	s		0	1	2	3	4	a	0	-	-	-	↙	-	b	1	-	-	↓	-	-	c	2	-	-	*	-	-	a	3	-	-	-	-	-	b	4	-	-	-	-	-	<div>LPS 'bab'</div> <table><tr><td></td><td></td><td>a</td><td>b</td><td>c</td><td>a</td><td>b</td></tr><tr><td>s</td><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>a</td><td>0</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr><tr><td>b</td><td>1</td><td>-</td><td>-</td><td>-</td><td>-</td><td>↘</td></tr><tr><td>c</td><td>2</td><td>-</td><td>-</td><td>-</td><td>↓</td><td>-</td></tr><tr><td>a</td><td>3</td><td>-</td><td>-</td><td>-</td><td>*</td><td>-</td></tr><tr><td>b</td><td>4</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr></table>			a	b	c	a	b	s		0	1	2	3	4	a	0	-	-	-	-	-	b	1	-	-	-	-	↘	c	2	-	-	-	↓	-	a	3	-	-	-	*	-	b	4	-	-	-	-	-
		a	b	c	a	b																																																																																																																																																																																																		
s		0	1	2	3	4																																																																																																																																																																																																		
a	0	-	-	-	-	↓																																																																																																																																																																																																		
b	1	-	-	-	-	↘																																																																																																																																																																																																		
c	2	-	-	↙	←	-																																																																																																																																																																																																		
a	3	-	*	-	-	-																																																																																																																																																																																																		
b	4	-	-	-	-	-																																																																																																																																																																																																		
		a	b	c	a	b																																																																																																																																																																																																		
s		0	1	2	3	4																																																																																																																																																																																																		
a	0	-	-	-	↙	←																																																																																																																																																																																																		
b	1	-	*	←	-	-																																																																																																																																																																																																		
c	2	-	-	-	-	-																																																																																																																																																																																																		
a	3	-	-	-	-	-																																																																																																																																																																																																		
b	4	-	-	-	-	-																																																																																																																																																																																																		
		a	b	c	a	b																																																																																																																																																																																																		
s		0	1	2	3	4																																																																																																																																																																																																		
a	0	-	-	-	↙	-																																																																																																																																																																																																		
b	1	-	-	↓	-	-																																																																																																																																																																																																		
c	2	-	-	*	-	-																																																																																																																																																																																																		
a	3	-	-	-	-	-																																																																																																																																																																																																		
b	4	-	-	-	-	-																																																																																																																																																																																																		
		a	b	c	a	b																																																																																																																																																																																																		
s		0	1	2	3	4																																																																																																																																																																																																		
a	0	-	-	-	-	-																																																																																																																																																																																																		
b	1	-	-	-	-	↘																																																																																																																																																																																																		
c	2	-	-	-	↓	-																																																																																																																																																																																																		
a	3	-	-	-	*	-																																																																																																																																																																																																		
b	4	-	-	-	-	-																																																																																																																																																																																																		
1	<p>Start at i=0, j=4, lps[]</p> <p>A[i]='a' A[j]='b' A[i] ≠ A[j] *At this point use condition else if s[i][j] == (i+1, j) i = i + 1</p> <p>s[i][j]=(1,4)</p>	<p>Start at i=0, j=4, lps[]</p> <p>A[i]='a' A[j]='b' A[i] ≠ A[j] *At this point use condition else j = j - 1</p> <p>s[i][j]=(0,3)</p>	<p>Start at i=0, j=3, lps[]</p> <p>A[i]='a' A[j]='a' A[i] = A[j] *At this point condition is meet if A[i] == A[j] i = i + 1 j = j - 1 append A[i] to lps → lps['a']</p> <p>s[i][j]=(1,2)</p>	<p>Start at i=1, j=4, lps[]</p> <p>A[i]='b' A[j]='b' A[i] = A[j] *At this point condition is meet if A[i] == A[j] i = i + 1 j = j - 1 append A[i] to lps → lps['b']</p> <p>s[i][j]=(2,3)</p>																																																																																																																																																																																																				
2	<p>i=1, j=4 A[i]='b' A[j]='b' A[i] = A[j] *At this point condition is meet if A[i] == A[j] i = i + 1 j = j - 1 append A[i] to lps → lps['b']</p> <p>s[i][j]=(2,3)</p>	<p>i=0, j=3 A[i]='a' A[j]='a' A[i] = A[j] *At this point condition is meet if A[i] == A[j] i = i + 1 j = j - 1 append A[i] to lps → lps['a']</p> <p>s[i][j]=(1,2)</p>	<p>i=1, j=2 A[i]='b' A[j]='c' A[i] ≠ A[j] *At this point use condition else if s[i][j] == (i+1, j) i = i + 1</p> <p>s[i][j]=(2,2) append A[i] to lps → lps['c']</p>	<p>i=2, j=3 A[i]='c' A[j]='a' A[i] ≠ A[j] *At this point use condition else if s[i][j] == (i+1, j) i = i + 1</p> <p>s[i][j]=(3,3) append A[i] to lps → lps['a']</p>																																																																																																																																																																																																				
3	<p>i=2, j=3 A[i]='c' A[j]='c' A[i] = A[j] *At this point use condition else j = j - 1</p> <p>lps['b'] s[i][j]=(2,2)</p>	<p>i=1, j=2 A[i]='b' A[j]='c' A[i] ≠ A[j] *At this point use condition else j = j - 1</p> <p>lps['a','b'] s[i][j]=(1,1)</p>	<p>i=2, j=2 *At this point the loops end</p> <p>Reverse the lps list and append lps + reverse(lps) = ['a', 'c'] + ['c', 'a'] = lps['a', 'c', 'a']</p>	<p>i=3, j=3 *At this point the loops end</p> <p>Reverse the lps list and append lps + reverse(lps) = ['b', 'a'] + ['a', 'b'] = lps['b', 'a', 'b']</p>																																																																																																																																																																																																				
4	<p>i=2, j=2 A[i]='c' A[j]='c' A[i] = A[j] *At this point condition is meet if A[i] == A[j] append A[i] to lps → lps['b','c']</p> <p>s[i][j]=(3,1)</p>	<p>i=1, j=1 *At this point the loops end</p> <p>Reverse the lps list and append lps + reverse(lps) = ['a', 'b'] + ['b', 'a'] = lps['a', 'b', 'a']</p>																																																																																																																																																																																																						
5	<p>i=3, j=1 *At this point the loops end i>j</p> <p>Reverse the lps list and append lps + reverse(lps) = ['b', 'c'] + ['c', 'b'] = lps['b', 'c', 'b']</p>																																																																																																																																																																																																							

- Palindromes with the subsequence of length 1 (the shortest subsequence): a, b, c, a, b.
- Palindromes with the subsequence of length 2: aa, bb.
- Palindromes with the subsequence of length 3 (longest subsequence for this case): bcb, others of length 3 aba, aca, bab.

c) What is the time complexity of your dynamic programming algorithm?

The LPS's dynamic programming algorithm has a time complexity of $O(n^2)$, where n is the length of the string. Filling the $n \times n$ dp table and computing each cell takes constant time.

The time complexity of the CONSTRUCT-LPS is $O(n)$, which includes $O(1)$ for initialization, $O(n)$ for the while loop, $O(n)$ for appending between lines, and $O(k)$ for moving in reverse order. In the worst-case situation moving reverse in a long dp matrix of LPS will cost $O(n)$.

a) Design a greedy algorithm for finding a placement of guards that uses the minimum number of guards to protect all the paintings with positions in X . A guard can be placed at any real number position in the hallway.

- A set $X = \{x_1, x_2, \dots, x_n\}$ of positions of paintings along a straight line L .
- A guard can protect all paintings within distance 1 on both sides of their position.
- Objective: Minimize the number of guards needed to protect all paintings.
- Input: X positions of the paintings.
- Output: $G[1..k]$ a list of locations for k guards such that for each $P[i]$ there exists a $G[j]$ such that $|P[i] - G[j]| \leq 1$, and k is minimum.

MERGE and MERGE_SORT use A as an array of painting positions, with p as the beginning index, q as the midway index, and r as the end index.

Finally, the `MIN_GUARDS` function sorts the paintings' positions using `MERGE_SORT` before applying a greedy approach to calculate the minimal number of guards required to protect all the paintings. Initially, the pseudocode imposed the constraint "within a distance of at most 1 of his position on both sides." If this restriction is not specified, the function must consider that distance as an argument.

```

1 n1 = q - p + 1
2 n2 = r - q
3 let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
4 for i = 1 to n1
5     L[i] = A[p + i - 1]
6 for j = 1 to n2
7     R[j] = A[q + j]
8 L[n1 + 1] = ∞
9 R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1

```

```

1 if p < r
2   q = (p + r) / 2
3   MERGE-SORT(A, p, q)
4   MERGE-SORT(A, q + 1, r)
5   MERGE(A, p, q, r)

```

[illegible]

Pseudocode without restriction

//Take as argument the painting's position and d coverage distance of the guards of both sides

MIN_GUARDS(X, d)

```

1  MERGE_SORT(X, 1, length(X))
2  G ← {}
3  last_guard ← -∞
4  for i ← 1 to length(X) do
5      if X[i] > last_guard + d then
6          last_guard ← X[i] + d
7          add last_guard to G
8  return G

```

In terms of time complexity, as we know the merge sort implementation ensures that the array is sorted in $O(n \log n)$ time, and the greedy algorithm then runs in linear time $O(n)$, resulting in an overall time complexity of $O(n \log n)$ for the entire algorithm.

b) Show the correctness of your algorithm (aka, show greedy choice property).

Validity and optimality of the problem:

Validity: all pieces have a guard within at most 1 foot.

Optimality: We use the minimum number of guards.

Scenarios

- Naïve approach. Place a guard that covers at most pieces as possible. Place guards such that each guard covers exactly two paintings if possible. This would not necessarily minimize the number of guards.

Painting														
Guard														
Distance position	0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Optimal guard placement. Optimal guard placement. This scenario is taking the greedy algorithm MIN_GUARDS without sorting the paintings. Without applying the merge sort the pseudocode ensures the minimum number of guards, however it doesn't guarantee to cover all the paintings.

Painting														
Guard														
Distance position	0	1	2	3	4	5	6	7	8	9	10	11	12	13

- Guard swapping for optimal location. Before placing a guard, check if swapping the position slightly improves the coverage without increasing the number of guards. This is implicitly handled by placing the guard at $x_i + 1$ ensuring maximum coverage. For the Optimal Guard Placement and Guard Swapping for Optimal Location, we can implement the pseudocode,

MIN_GUARDS(X, d)

```

1  MERGE_SORT(X, 1, length(X))
2  G ← {}
3  last_guard ← -∞
4  for i ← 1 to length(X) do
5      if X[i] > last_guard + d then
6          last_guard ← X[i] + d
7          add last_guard to G
8  return G

```

In the pseudocode, we start the process of determining the quantity of guards at line 2 by initializing an empty set $G \leftarrow \{\}$ to store the positions of the guards.

The line 3, $\text{last_guard} \leftarrow -\infty$, means that it will start from the leftmost unguarded painting and move to the right, ensuring that each guard placed covers as many paintings as possible.

Line 4, For $i \leftarrow 1$ to $\text{length}(X)$ do. The loop iterates over the paintings' sorted positions from left to right. This for loop will process the paintings; the difference is that for the optimal guard placement scenario, we

don't have the paintings in sorted order. Next, sorting is important because it ensures that we cover each painting with the fewest number of guards.

Line 5 indicates the point at which the pseudocode starts optimally placing the guards. If $X[i] > \text{last_guard} + d$, then it will check if the current painting is outside the coverage range of the last placed guard. If this is the case, a new guard will take over.















We will place a guard at the position $X[i] + d$ on line 6. This is the farthest position that can still cover the current painting $X[i]$ and potentially cover future paintings as well.

On line 7, add `last_guard` to `G`, and then this line adds the position of the guard to the set of guards in `G`.

Once we return the `G`, we continue the process until we have secured all the paintings.

Example unsorted paintings
















- Unsorted positions of paintings: $X=\{7,1,10,2,8,4,5\}$
- Guard coverage distance: $d=1$

Painting												
Guard												
Distance position	0	1	2	3	4	5	6	7	8	9	10	11

Step	i	X[i]	last_guard	Guard Placed?	New last_guard	G
1	1	7	$-\infty$	Yes	8	{8}
2	2	1	8	No	8	{8}
3	3	10	8	Yes	11	{8, 11}
4	4	2	11	No	11	{8, 11}
5	5	8	11	No	11	{8, 11}
6	6	4	11	No	11	{8, 11}
7	7	5	11	No	11	{8, 11}

Example sorted paintings

- Sorted positions of paintings: $X=\{1,2,4,5,7,8,10\}$
- Guard coverage distance: $d=1$

Painting												
Guard												
Distance position	0	1	2	3	4	5	6	7	8	9	10	11

Step	i	X[i]	last_guard	Guard Placed?	New last_guard	G
1	1	1	$-\infty$	Yes	2	{2}
2	2	2	2	No	2	{2}
3	3	4	2	Yes	5	{2, 5}
4	4	5	5	No	5	{2, 5}
5	5	7	5	Yes	8	{2, 5, 8}
6	6	8	8	No	8	{2, 5, 8}
7	7	10	8	Yes	11	{2, 5, 8, 11}

We can see in both examples that not sorting the paintings' positions leads to suboptimal guard placement. In the unsorted case, the algorithm fails to cover all paintings with the minimum number of guards because it does not process the positions in a sequential and ordered manner. Even though we place more guards on the sorted case, the algorithm conveys our objective, "Minimize the number of guards needed to protect all paintings." In this scenario, guards are placed optimally to cover all paintings with the minimum number of guards. For this reason, sorting the positions is crucial for the greedy algorithm to work correctly and ensure that all paintings are covered optimally.