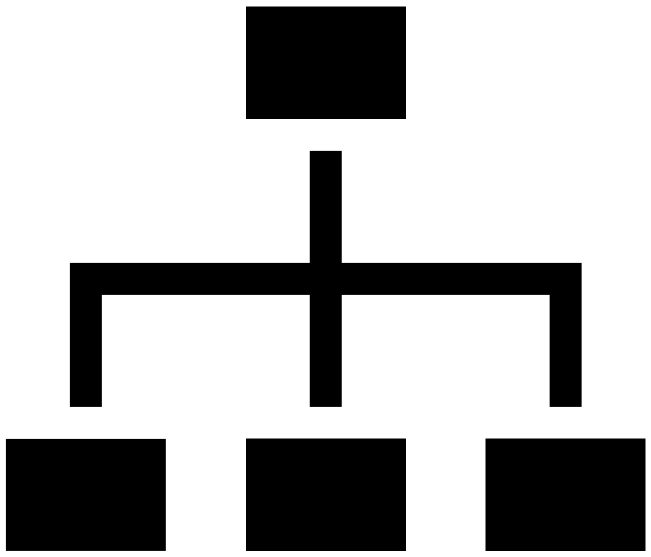
DYNAMIC Programming

Daniel Alvarez Sil

Contents.

"Dynamic Programming (DP) is a technique used in mathematics and computer science to solve complex problems by breaking them down into simpler subproblems."



1. Arrayland's Challenge.

1.1. Exercise.

In the digital realm of Arrayland, where data structures shape the foundations of society, the wise council has posed a challenging task to sharpen the minds of its citizens. The challenge involves understanding the subtle differences within the ordered structures, specifically within subarrays.

You are given an array of integers A of size N. Accompanying the array, you receive Q queries. Each query specifies a subarray defined by two indices L and R (where $1 \leq L < R \leq N$), and you are tasked with finding the smallest absolute difference between any two distinct elements within this subarray.

For each query, determine the minimum absolute difference between any two distinct elements contained in the subarray from index L to R.

Input

The first line of input contains an integer $N (2 \leq N \leq 10^4)$, the size of the array. The second line contains N integers a_i separated by spaces ($1 \leq a_i \leq 10^6$), the elements in the array.

The third line contains an integer Q ($1 \leq Q \leq 10^5$), the number of queries to answer.

Each of the next Q lines contain two integers L, and R ($1 \leq L < R \leq N$) representing the indices of the subarray for the query.

Output

Print Q lines, where the i-th line contains a single integer representing the answers for to the i-th query.

Examples:

Input	Input

10
4 5 1 3 2 1 4 6 7 8
3
1 10
3 6
7 10
	8
	3 45 1 2 3 4 4 4
	2
	1 8
	1 2

Output	**Output**
0 0 1	0 42

1.2. Solution.

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e4+10;

int a[N];
int dp[N][N];

void resolver(){
  int n,q,l,r;
  cin >> n;

  for (int f = 1; f <= n; f++){
    cin >> a[f];
  }

  for (int f = 1; f <= n; f++){
    for (int j = f + 1; j <= n; j++){
      dp[f][j] = abs(a[f] - a[j]);
    }
  }

  for (int f = 3; f <= n; f++){
    for (int j = f; j <= n; j++){
      int i = j - f + 1;
      dp[i][j] = min(dp[i][j], min(dp[i][j-1], dp[i+1][j]));
    }
  }

  cin >> q;
  while (q--){
    cin >> l >> r;
    cout << dp[l][r] << '\n';
  }
}

int main(){
  ios::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);
  resolver();
  return 0;
}
```

1.3. Explanation.

In order to understand this algorithm, we will use an example of 5 integers.

$$4\ 5\ 1\ 3\ 10$$

The algorithm allocates a bidimensional array, dp, of $[10^4 + 10]$ $[10^4 + 10]$ to account for the worst case, which constitutes the scenario in which you are given as input an array of 10^4 elements (the remaining elements are added as a safety measure). This $2D$ array will be used to house all the possible absolute differences between any two given elements of the original array, which is accomplished through the following snippet.

Absolute Difference Between any 2 Indexes of the Array.

```
for (int f = 1; f <= n; f++){
  for (int j = f + 1; j <= n; j++){
    dp[f][j] = abs(a[f] - a[j]);
  }
}
```

For example, the element $dp[1][2]$ stores the absolute difference between the elements 1 and 2 of the original array, which is 1 ($|4 - 5| = 1$).

Thus, we arrive at the following bidimensional array.

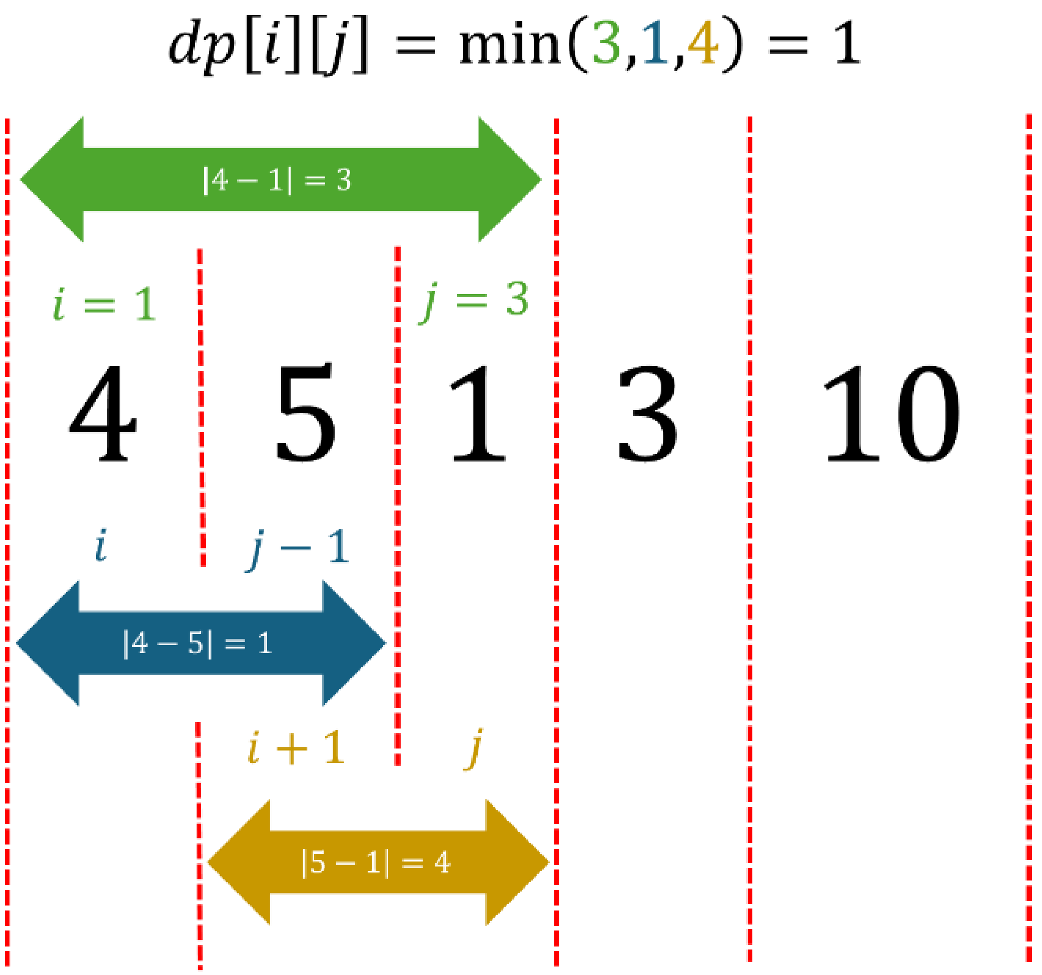
$dp[1][2] = 1$			
$dp[1][3] = 3$	$dp[2][3] = 4$		
$dp[1][4] = 1$	$dp[2][4] = 2$	$dp[3][4] = 2$	
$dp[1][5] = 6$	$dp[2][5] = 5$	$dp[3][5] = 9$	$dp[4][5] = 7$

Having formed this bidimensional array, we iterate as follows. Through the execution of this nested iteration, we update the 2D array so that for every index i, j such that $j > i$, then the value stored in $dp[i][j]$ is the minimum absolute difference between indexes i and j (assuming the first element is index 1).

Minimum Absolute Difference Between any 2 Indexes of the Array.

```
for (int f = 3; f <= n; f++){
    for (int j = f; j <= n; j++){
      int i = j - f + 1;
      dp[i][j] = min(dp[i][j], min(dp[i][j-1], dp[i+1][j]));
    }
}
```

In essence, this snippet updates every range of numbers that contains more than 2 numbers. For every range of numbers, we update the value of $dp[i][j]$ set to the minimum value between itself, $dp[i][j-1]$ and $dp[i+1][j]$. After the execution of this code snippet, we can confirm that every value for $dp[i][j]$ contains the smallest absolute difference between the values contained between the indexes i and j. This is due to the fact that this algorithm is implemented beginning with the smaller ranges, and it increments with each outermost iteration. The following image shows a visual demonstration of the functionality discussed in this paragraph, which corresponds to the first iteration.



We iterate from range 3 to n because the smallest ranges we are interested in updating are those equal or bigger than 3. This is because the elements in dp that cover 2 elements in terms of range already possess the smallest absolute difference. Next, we will cover each iteration of the outermost loop.

Before doing a step-by-step execution of the last snippet shown, we will show one last visualization of every step of the algorithm's logic.

Step 1:

$dp[1][2]$			
$\min\begin{pmatrix} dp[1][3], \\ dp[1][2], \\ dp[2][3] \end{pmatrix}$	$dp[2][3]$		
$dp[1][4]$	$\min\begin{pmatrix} dp[2][4], \\ dp[2][3], \\ dp[3][4] \end{pmatrix}$	$dp[3][4]$	
$dp[1][5]$	$dp[2][5]$	$\min\begin{pmatrix} dp[3][5], \\ dp[3][4], \\ dp[4][5] \end{pmatrix}$	$dp[4][5]$

Step 2:

$dp[1][2]$			
$dp[1][3]$	$dp[2][3]$		
$\min\begin{pmatrix} dp[1][4], \\ dp[1][3], \\ dp[2][4] \end{pmatrix}$	$dp[2][4]$	$dp[3][4]$	
$dp[1][5]$	$\min\begin{pmatrix} dp[2][5], \\ dp[2][4], \\ dp[3][5] \end{pmatrix}$	$dp[3][5]$	$dp[4][5]$

Step 3:

$dp[1][2]$			
$dp[1][3]$	$dp[2][3]$		
$dp[1][4]$	$dp[2][4]$	$dp[3][4]$	
$\min\begin{pmatrix} dp[1][5], \\ dp[1][4], \\ dp[2][5] \end{pmatrix}$	$dp[2][5]$	$dp[3][5]$	$dp[4][5]$

Next, we will conduct a step-by-step see-through but using the values of our array.

First Iteration:

f	j	$i = j - f + 1$	$dp[i][j]$	$dp[i][j-1]$	$dp[i+1][j]$	$\min(range)$
3	3	1	$[1][3] = 3$	$[1][2] = 1$	$[2][3] = 4$	$[1][3] = 1$
3	4	2	$[2][4] = 2$	$[2][3] = 4$	$[3][4] = 2$	$[2][4] = 2$
3	5	3	$[3][5] = 9$	$[3][4] = 2$	$[4][5] = 7$	$[3][5] = 2$

The bidimensional array is updated as following:

$dp[1][2] = 1$			
$dp[1][3] = 1$	$dp[2][3] = 4$		
$dp[1][4] = 1$	$dp[2][4] = 2$	$dp[3][4] = 2$	
$dp[1][5] = 6$	$dp[2][5] = 5$	$dp[3][5] = 2$	$dp[4][5] = 7$

Second Iteration:

f	j	$i = j - f + 1$	$dp[i][j]$	$dp[i][j-1]$	$dp[i+1][j]$	$\min(range)$
4	4	1	$[1][4] = 1$	$[1][3] = 1$	$[2][4] = 2$	$[1][4] = 1$
4	5	2	$[2][5] = 5$	$[2][4] = 2$	$[3][5] = 2$	$[2][5] = 2$

The bidimensional array is updated as following:

$dp[1][2] = 1$			
$dp[1][3] = 1$	$dp[2][3] = 4$		
$dp[1][4] = 1$	$dp[2][4] = 2$	$dp[3][4] = 2$	
$dp[1][5] = 6$	$dp[2][5] = 2$	$dp[3][5] = 2$	$dp[4][5] = 7$

Third Iteration:

f	j	$i = j - f + 1$	$dp[i][j]$	$dp[i][j-1]$	$dp[i+1][j]$	$\min(range)$
5	5	1	$[1][5] = 6$	$[1][4] = 1$	$[2][5] = 2$	$[1][5] = 1$

Finally, the bidimensional array is updated as following:

$dp[1][2] = 1$			
$dp[1][3] = 1$	$dp[2][3] = 4$		
$dp[1][4] = 1$	$dp[2][4] = 2$	$dp[3][4] = 2$	
$dp[1][5] = 1$	$dp[2][5] = 2$	$dp[3][5] = 9$	$dp[4][5] = 7$

As it is now obvious, this 2D array holds all the possible smallest absolute differences between any two indexes i and j. Thus, answering any query can be done in constant, $O(1)$, time.

The time complexity of this whole algorithm is $O(N^2 + M)$ because of the following reasons.

- Calculating every possible absolute difference has a complexity of $O(N^2)$.
- Calculating every smallest possible absolute difference between any two indexes has a complexity of $O(N^2)$, as well.
- Then, there occurs an iteration of all the queries given, so it has a time complexity of $O(M)$.
- Within every query iteration, there occurs an extraction of the smallest possible difference which has already been calculated, so it takes constant time, as was already mentioned.

$$O(N^2 + N^2 + M + 1) = O(2N^2 + M + 1) = O(N^2 + M)$$

2. Journey To Stringland.

2.1. Exercise.

In the mystical kingdom of Stringland, where characters and sequences hold the secrets to ancient magic, there exists a revered challenge that tests the wisdom and skill of the realm's scribes. This challenge, known as "The Quest for the Palindromic Subsequence," involves transforming ordinary sequences of letters into powerful palindromic symbols.

You, a scribe of Stringland, are given a sequence of characters S and must undertake a quest to create a palindromic subsequence of length K. Palindromic sequences are believed to hold magical properties as they read the same forward and backward. To aid in your quest, you are allowed to change any character in S to any other character. The fewer changes you make, the stronger the resulting magic.

Determine the minimum number of character changes needed for S to contain at least one palindromic subsequence of length K.

Input

The first line contains two integers N and K $(1 \leq K \leq N \leq 500)$, representing the length of the string S and the size of the palindromic subsequence. The second line contains the string S, composed of lowercase English letters.

Output

Output one line with an integer, the minimum number of character changes needed for S to contain at least one palindromic subsequence of length K.

Examples:

Input	Input	Input
3 3 abc	10 4 abcdcaefgj	5 2 abcde
Output	**Output**	**Output**
1	0	1

Note:

A subsequence of a given sequence is a sequence that can be derived from the original sequence by deleting some or none of the elements without changing the order of the remaining elements. For example, if the original sequence is ($A = \{a, b, c, d, e\}$), then ($\{a, b, d\}$) and ($\{c, e\}$) are subsequences of (A), but ($\{b, a\}$) is not a subsequence because it changes the order of the elements.

2.2. Solution.

```
#include <bits/stdc++.h>
using namespace std;

static const int INF = 10000; // A large enough value

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  int N, K;
  cin >> N >> K;
  string S;
  cin >> S;

  static int dp[501][501][501];
  for (int i = 0; i < N; i++) {
    for (int j = i; j < N; j++) {
      for (int l = 0; l <= K; l++) {
        dp[i][j][l] = INF;
      }
    }
  }

  // Base case: empty subsequence
  for (int i = 0; i < N; i++) {
    for (int j = i; j < N; j++) {
      dp[i][j][0] = 0;
    }
  }

```

```
    // Base case: subsequence of length 1 is always 0 changes
    for (int i = 0; i < N; i++) {
      dp[i][i][1] = 0;
    }

    // Fill DP
    for (int length = 2; length <= N; length++) {
      for (int i = 0; i + length - 1 < N; i++) {
        int j = i + length - 1;
        for (int l = 1; l <= min(K, length); l++) {
          // Skip the first character
          dp[i][j][l] = min(dp[i][j][l], dp[i+1][j][l]);

          // Skip the last character
         dp[i][j][l] = min(dp[i][j][l], dp[i][j-1][l]);

          // Use both ends if (l == 2)
          if (l == 2){
            int cost = (S[i] == S[j]) ? 0 : 1;
            dp[i][j][l] = min(dp[i][j][l], cost);
          }

          // Use both ends and data contained (if l >= 2)
          else if (l > 2) {
            int cost = (S[i] == S[j]) ? 0 : 1;
            dp[i][j][l] = min(dp[i][j][l], dp[i+1][j-1][l-2] + cost);
          }

        }
      }
    }

    cout << dp[0][N-1][K] << "\n";

    return 0;
}
```

2.3. Explanation.

This code builds an array of 3 dimensions to house the **minimum changes required to obtain a palindrome subsequence of length l between indexes i and j**, thus, the need of 3 dimensions (i, j and l). The general idea is to obtain the answer by building up to it. Firstly, we obtain the minimum changes to obtain palindromic subsequences for ranges of 2 numbers, then, we use that information to obtain the minimum changes for ranges of 3 numbers, and so on.

To illustrate the solution, we will use the following input.

$$n = 5; k = 4; s = "awasa"$$

With this input, we are required to output the minimum number of character changes needed to obtain a palindromic subsequence of size 4 out of the string "$awasa$", which has a size of 5. Under this scenario, the answer should be 1 because, changing either 'w' or 's' to an 'a', results in the string containing the subsequence of $\{a, a, a, a\}$, which is effectively a palindrome of size 4.

To achieve this, firstly, each value of the three-dimensional array will be set to a large value, to simulate infinity. This is done to facilitate future operations, in which the minimum value is calculated between the current value and other value, effectively replacing the value being currently examined (see lines 42, 45, 50 and 56).

Initialization of Every Value of the 3D Array with an Infinity Value.

```
16.  for (int i = 0; i < N; i++) {
17.    for (int j = i; j < N; j++) {
18.      for (int l = 0; l <= K; l++) {
19.        dp[i][j][l] = INF;
20.      }
21.    }
22.  }
```

NOTE: For this exercise, the first index will be regarded as **0, not 1**.

Value	a	w	a	s	a
Index	0	1	2	3	4

The tridimensional array is updated as shown in the following table. As a reminder, the value stored in $dp[i][j]$ for a certain l represents the **minimum number of character changes needed to make a palindromic subsequence of size l within the indexes i and j (inclusive, $[i, j]$,** which means indexes i and j are included in the interval).

	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
$l = 0$	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = \infty$
	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
$l = 1$	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = \infty$

$l = 2$	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = \infty$
$l = 3$	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = \infty$
$l = 4$	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = \infty$

We end the storage of values up until $l = k$, which is 4 in this case. This is due to the fact that we are not interested in obtaining the minimum number of changes for palindromic subsequences of size larger than k.

Next, we update the tridimensional array to follow these two simple rules:

- If the size of the palindromic subsequence to achieve is 0, there is nothing to change. Thus, the number of changes is 0.
- If the size of the subsequence is 1 and the size of the palindromic subsequence currently sought for is 1 as well, then there is nothing to be changed because a single character is already a palindrome. Thus, the number of changes is 0.

Update the 3D Array According to the Two Base Rules.

```
// Base case: empty subsequence
for (int i = 0; i < N; i++) {
  for (int j = i; j < N; j++) {
    dp[i][j][0] = 0;
  }
}

// Base case: subsequence of length 1 is always 0 changes
for (int i = 0; i < N; i++) {
  dp[i][i][1] = 0;
}
```

$l = 0$	$dp[0][0] = 0$				
	$dp[0][1] = 0$	$dp[1][1] = 0$			
	$dp[0][2] = 0$	$dp[1][2] = 0$	$dp[2][2] = 0$		
	$dp[0][3] = 0$	$dp[1][3] = 0$	$dp[2][3] = 0$	$dp[3][3] = 0$	
	$dp[0][4] = 0$	$dp[1][4] = 0$	$dp[2][4] = 0$	$dp[3][4] = 0$	$dp[4][4] = 0$

$l = 1$	$dp[0][0] = 0$				
	$dp[0][1] = \infty$	$dp[1][1] = 0$			
	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = 0$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = 0$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = 0$

With this last update to the 3D array, we can begin to recursively construct our solutions. To do so, we will visit every possible range of numbers that contains more than 1 number, up to N. Then, for each range of numbers, we will visit and update the values of $dp[i][j][l]$ for every l such that $1 \leq l \leq \min(k, r)$, where r is the number of elements contained in the range of numbers found between indexes i and j (inclusive).

Construction of Every Solution.

```
// Fill DP
for (int length = 2; length <= N; length++) {
  for (int i = 0; i + length - 1 < N; i++) {
    int j = i + length - 1;
    for (int l = 1; l <= min(K, length); l++) {
      // Skip the first character
      dp[i][j][l] = min(dp[i][j][l], dp[i+1][j][l]);

      // Skip the last character
     dp[i][j][l] = min(dp[i][j][l], dp[i][j-1][l]);

      // Use both ends if (l == 2)
      if (l == 2){
        int cost = (S[i] == S[j]) ? 0 : 1;
        dp[i][j][l] = min(dp[i][j][l], cost);
      }

      // Use both ends and data contained (if l >= 2)
      else if (l > 2) {
        int cost = (S[i] == S[j]) ? 0 : 1;
        dp[i][j][l] = min(dp[i][j][l], dp[i+1][j-1][l-2] + cost);
      }

    }
  }
}
```

In order to update each value of the tridimensional array, we will use the following rules.

1. Exclude the first character of the subsequence and check the minimum changes required if l remains the same. If said value is less than the current value, we assign it to the current value.

```
// Skip the first character
dp[i][j][l] = min(dp[i][j][l], dp[i+1][j][l]);
```

2. Exclude the last character of the subsequence and check the minimum changes required if l remains the same. If said value is less than the current value, we assign it to the current value.

```
// Skip the last character
dp[i][j][l] = min(dp[i][j][l], dp[i][j-1][l]);
```

3. If l is equal to 2, then we check the ends of the range. If they are equal, then the ends already conform a palindromic subsequence of size 2, else, there is 1 change to be made. If said number of changes is less than the current value, we assign it to the current value.

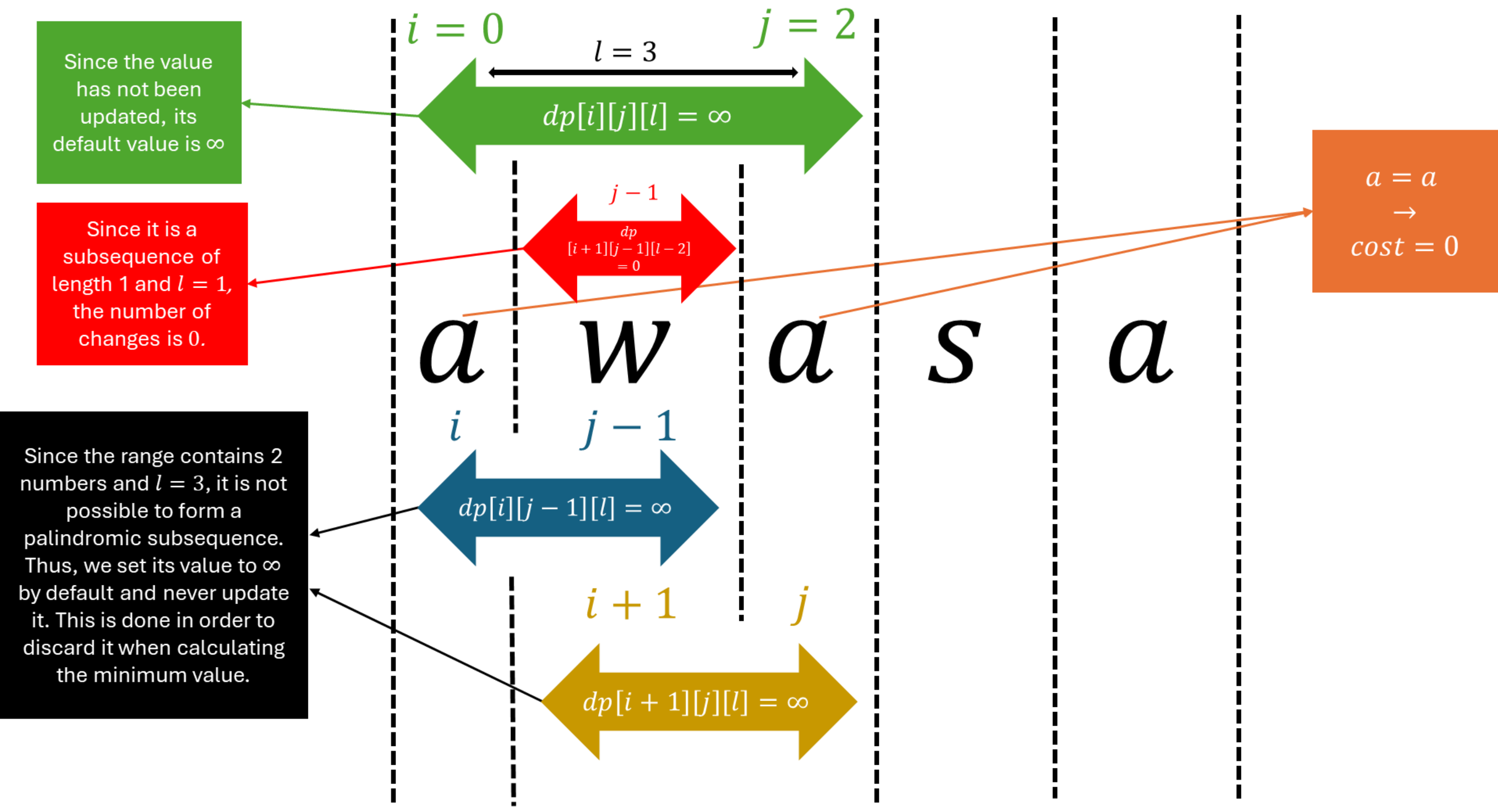
```
// Use both ends if (l == 2)
if (l == 2){
  int cost = (S[i] == S[j]) ? 0 : 1;
  dp[i][j][l] = min(dp[i][j][l], cost);
}
```

4. If l is larger than 2, we also check the ends of the range. If they are equal, then the number of changes is equal to the ones required for a palindromic subsequence of size $l-2$ if the ends are disregarded (due to the nature of palindromes). Else, there is an addition of 1 to be made to the number of changes required for palindromic subsequence of size $l-2$ if the ends are disregarded. If the newly calculated number of changes is less than the current value, we assign it to the current value.

```
// Use both ends and data contained (if l >= 2)
else if (l > 2) {
  int cost = (S[i] == S[j]) ? 0 : 1;
  dp[i][j][l] = min(dp[i][j][l], dp[i+1][j-1][l-2] + cost);
}
```

To further strengthen the logic behind the algorithm, the following image helps visualize the way in which $dp[0][2][3]$ is calculated for the present string “awasa”.

$$UPDATE \rightarrow dp[i][j][l] = \min(\infty, 0 + 0, \infty, \infty) = 0$$


Since the value has not been updated, its default value is ∞
Since it is a subsequence of length 1 and $l = 1$, the number of changes is 0.
Since the range contains 2 numbers and $l = 3$, it is not possible to form a palindromic subsequence. Thus, we set its value to ∞ by default and never update it. This is done in order to discard it when calculating the minimum value.
$i = 0$
$j = 2$
$l = 3$
$dp[i][j][l] = \infty$
$j - 1$
$dp[i+1][j-1][l-2] = 0$
$a = a \rightarrow cost = 0$
a w a s a
i
$j - 1$
$dp[i][j-1][l] = \infty$
$i + 1$
j
$dp[i+1][j][l] = \infty$

On a more abstract fashion, in a general sense, each cell of our 3D array is updated as follows.

$l = x - 2$			
	$dp[i][j-1]$	$dp[i+1][j-1]$	
	$dp[i][j]$	$dp[i+1][j]$	

$l = x$			
	$dp[i][j-1]$		
	$dp[i][j]$	$dp[i+1][j]$	

Assuming the string variable is named s, if $s[i] = s[j]$, then $cost = 0$, else $cost = 1$.

$$dp[i][j][x] = \min(dp[i+1][j-1][x-2] + cost, dp[i][j-1][x], dp[i+1][j][x])$$

Now, without further ado, let's build the solution to the problem step by step.

Step 1. Length of Range, $length = 2$.

$l = 1$	$dp[0][0] = 0$				
	$dp[0][1] = 0$	$dp[1][1] = 0$			
	$dp[0][2] = \infty$	$dp[1][2] = 0$	$dp[2][2] = 0$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = 0$	$dp[3][3] = 0$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = 0$	$dp[4][4] = 0$
$l = 2$	$dp[0][0] = \infty$				
	$dp[0][1] = 1$	$dp[1][1] = \infty$			
	$dp[0][2] = \infty$	$dp[1][2] = 1$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = \infty$	$dp[2][3] = 1$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = \infty$	$dp[3][4] = 1$	$dp[4][4] = \infty$

Step 2. Length of Range, $length = 3$.

$l = 1$	$dp[0][0] = 0$				
	$dp[0][1] = 0$	$dp[1][1] = 0$			
	$dp[0][2] = 0$	$dp[1][2] = 0$	$dp[2][2] = 0$		
	$dp[0][3] = \infty$	$dp[1][3] = 0$	$dp[2][3] = 0$	$dp[3][3] = 0$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = 0$	$dp[3][4] = 0$	$dp[4][4] = 0$
$l = 2$	$dp[0][0] = \infty$				
	$dp[0][1] = 1$	$dp[1][1] = \infty$			
	$dp[0][2] = 0$	$dp[1][2] = 1$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = 1$	$dp[2][3] = 1$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = 0$	$dp[3][4] = 1$	$dp[4][4] = \infty$
$l = 3$	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
	$dp[0][2] = 0$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = \infty$	$dp[1][3] = 1$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = \infty$	$dp[2][4] = 0$	$dp[3][4] = \infty$	$dp[4][4] = \infty$

Step 3. Length of Range, $length = 4 = k$.

$l = 1$	$dp[0][0] = 0$				
	$dp[0][1] = 0$	$dp[1][1] = 0$			
	$dp[0][2] = 0$	$dp[1][2] = 0$	$dp[2][2] = 0$		
	$dp[0][3] = 0$	$dp[1][3] = 0$	$dp[2][3] = 0$	$dp[3][3] = 0$	
	$dp[0][4] = \infty$	$dp[1][4] = 0$	$dp[2][4] = 0$	$dp[3][4] = 0$	$dp[4][4] = 0$
$l = 2$	$dp[0][0] = \infty$				
	$dp[0][1] = 1$	$dp[1][1] = \infty$			
	$dp[0][2] = 0$	$dp[1][2] = 1$	$dp[2][2] = \infty$		
	$dp[0][3] = 0$	$dp[1][3] = 1$	$dp[2][3] = 1$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = 0$	$dp[2][4] = 0$	$dp[3][4] = 1$	$dp[4][4] = \infty$

	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
$l = 3$	$dp[0][2] = 0$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = 0$	$dp[1][3] = 1$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = 0$	$dp[2][4] = 0$	$dp[3][4] = \infty$	$dp[4][4] = \infty$
	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
$l = 4$	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = 2$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = \infty$	$dp[1][4] = 2$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = \infty$

Step 4. Length of Range, $length = 5 = N$, but we do not surpass $l = 4 = k$. It is important to remember that $length$ refers to the number of characters contained in the range (inclusive), whereas l refers to the length of the palindromic subsequence we are looking for.

	$dp[0][0] = 0$				
	$dp[0][1] = 0$	$dp[1][1] = 0$			
$l = 1$	$dp[0][2] = 0$	$dp[1][2] = 0$	$dp[2][2] = 0$		
	$dp[0][3] = 0$	$dp[1][3] = 0$	$dp[2][3] = 0$	$dp[3][3] = 0$	
	$dp[0][4] = 0$	$dp[1][4] = 0$	$dp[2][4] = 0$	$dp[3][4] = 0$	$dp[4][4] = 0$
	$dp[0][0] = \infty$				
	$dp[0][1] = 1$	$dp[1][1] = \infty$			
$l = 2$	$dp[0][2] = 0$	$dp[1][2] = 1$	$dp[2][2] = \infty$		
	$dp[0][3] = 0$	$dp[1][3] = 1$	$dp[2][3] = 1$	$dp[3][3] = \infty$	
	$dp[0][4] = 0$	$dp[1][4] = 0$	$dp[2][4] = 0$	$dp[3][4] = 1$	$dp[4][4] = \infty$
	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
$l = 3$	$dp[0][2] = 0$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = 0$	$dp[1][3] = 1$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$dp[0][4] = 0$	$dp[1][4] = 0$	$dp[2][4] = 0$	$dp[3][4] = \infty$	$dp[4][4] = \infty$
	$dp[0][0] = \infty$				
	$dp[0][1] = \infty$	$dp[1][1] = \infty$			
$l = 4$	$dp[0][2] = \infty$	$dp[1][2] = \infty$	$dp[2][2] = \infty$		
	$dp[0][3] = 2$	$dp[1][3] = \infty$	$dp[2][3] = \infty$	$dp[3][3] = \infty$	
	$\boldsymbol{dp[0][4] = 1}$	$dp[1][4] = 2$	$dp[2][4] = \infty$	$dp[3][4] = \infty$	$dp[4][4] = \infty$

Finally, we arrive at the solution to the question: “Given the string $\{a, w, a, s, a\}$ what is the minimum number of character changes required to obtain a palindromic subsequence of 4 characters?” The answer is $\mathbf{1}$.

The time complexity of this algorithm is $O(N^2 \cdot K)$ because of the following reasons.

- Initializing the three-dimensional array takes $O(N^2 \cdot K)$ time.
- Initializing empty subsequences has a complexity of $O(N^2)$.
- Initializing subsequences of length 1 takes $O(N)$ time.
- Next, the construction of the solution has again a complexity of $O(N^2 \cdot K)$.
- Finally, the output of the solution has constant time, $O(1)$.

$$O(N^2 \cdot K + N^2 + N + N^2 \cdot K + 1) = O\big(2 \cdot N^2 \cdot K + N(N+1)\big)$$

$$= O(N^2 \cdot K + N^2) = O\big(N^2(K+1)\big)$$

$$= O(N^2 \cdot K)$$