Dynamic programming

Problems with parentheses

E-OLYMP 1551. Correcting parenthesization Given a string of parentheses, we must turn it into a well formed string by changing as few characters as possible (we cannot delete or insert characters).

There are three kinds of parentheses: regular (), brackets [] and curly brackets {}. Each pair has an opening ('(', '[' and '{' respectively) and a closing (')', ']' and '}') character.

A well formed string of parentheses is defined by the following rules:

- The empty string is well formed.
- If *s* is a well formed string, (*s*), [*s*] and {*s*} are well formed strings.
- If *s* and *t* are well formed strings, the concatenation *st* is a well formed string.

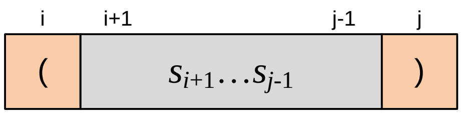
As examples, "([{}])", "" and "(){}[]" are well formed strings and "([}]", "([)]" and "{" are malformed strings.

For the given string of parentheses find the minimum number of characters that need to be changed to make it into a well formed string.

► Let f(i, j) be the smallest number of characters that can be changed so that the substring $s_i s_{i+1} \ldots s_{j-1} s_j$ becomes correct. Then the following statements hold:

1. f(i, j) = 0 for $i > j$, since in this case the substring will be empty.

2. f(i, j) = f(i + 1, j – 1) + enc(s_i ,s_j). Make s_i to be the opening parenthesis and s_j to be the corresponding closing parenthesis. Further, recursively consider the substring $s_{i+1} \ldots s_{j-1}$.



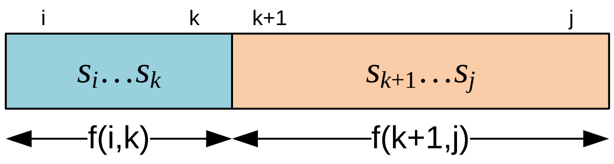
Function enc(s_i ,s_j) returns:

a) 0, if the characters s_i and s_j are matching opening and closing parentheses;

б) 2, if s_i is a closing parenthesis and s_j is an opening parenthesis;

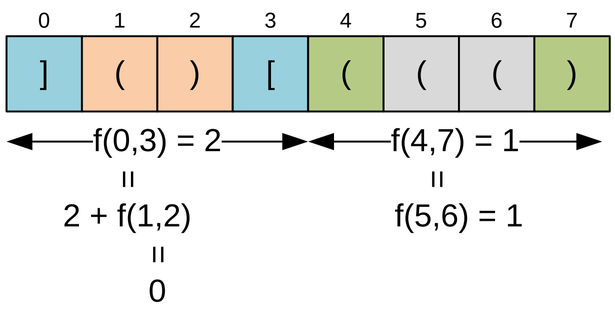
в) 1 otherwise. In this case, it is enough to change one of the symbols s_i or s_j so that they form the correct pair of parenthesis;

3. f(i, j) = $\min\limits_{i<k<j}$(f(i, k) + f(k + 1, j)). Consider the sequence $s_i s_{i+1} \ldots s_{j-1} s_j$ as a sequence of two regular parenthesis structures: $s_i \ldots s_k$ и $s_{k+1} \ldots s_j$. The length of a substring $s_i \ldots s_k$ must be ***even***, hence k takes the values i + 1, i + 3, …, j – 2.



Consider the first line from the sample.

$$f(0, 7) = f(0, 3) + f(4, 7) = (2 + f(1, 2)) + (0 + f(5, 6)) = (2 + 0) + (0 + 1) = 3$$



Store the values of f(i, j) in m[i][j]. Read the input string into s.

```
int m[51][51], res;
string s;
```

Implementation of the function ***enc***(c, d).

```
int enc(char c, char d)
{
  string p = "([{)]}";
```

Function returns 2 if c is a closing parenthesis and d is an opening parenthesis.

```
  if (p.find(c) / 3 > 0 && p.find(d) / 3 < 1) return 2;
```

Function returns 0, if c and d are the corresponding parentheses. If they are not in the correct order, the function will return the value 2 above.

```
  if (p.find(c) % 3 == p.find(d) % 3 && c != d) return 0;
```

In all other cases, return 1.

```
  return 1;
}
```

Function ***f***(i, j) returns the smallest number of characters that can be changed so that the substring $s_i s_{i+1} \ldots s_{j-1} s_j$ becomes valid.

```
int f(int i, int j)
{
  if (i > j) return 0;
  if (m[i][j] != -1) return m[i][j];

  int r = f(i+1,j-1) + enc(s[i],s[j]);
  for(int k = i + 1; k < j; k += 2)
    r = min(r,f(i,k) + f(k+1,j));

  return m[i][j] = r;
}
```

The main part of the program. Process multiple test cases.

```
while(cin >> s)
{
  memset(m,-1,sizeof(m));
```

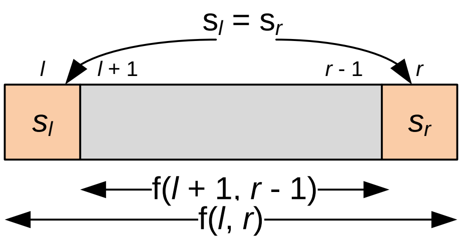
The answer to the problem is the value f(0, |*s*| – 1), where *s* is the input string.

```
  res = f(0, s.size() - 1);
  cout << res << endl;
}
```

E-OLYMP 7447. Cut a string You are given a string *s*. It is allowed to take any two same neighbor symbols of this string and delete them. This operation you can do while possible. At the beginning you can choose any symbols from string and delete them. Determine the minimum number of symbols you can delete at the beginning, so that you get the empty string after performing allowed operations.

► Let dp[*l*][*r*] = f(*l*, *r*) be the minimum number of characters that should be removed from the substring $s_l..s_r$, so that later, as a result of applying operations (removing identical adjacent characters), an empty string can be obtained. Then:

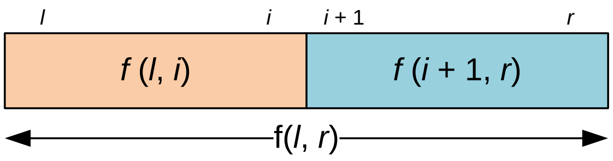
- f(*l*, *r*) = 0, if *l* > *r*;
- f(*l*, *l*) = 1, single character should be removed at the beginning;
- f(*l*, *r*) = f(*l* + 1, *r* – 1), if s[*l*] = s[*r*]. If the leftmost and the rightmost characters are the same, then the inner part should be deleted, after which these characters become adjacent and they can be deleted by applying the operation;



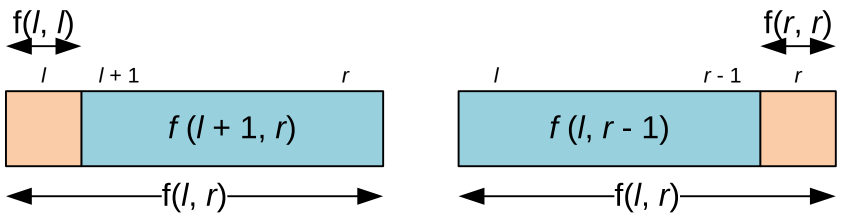
- f(*l*, *r*) = 1 + f(*l* + 1, *r*), if we remove the first character;
- f(*l*, *r*) = 1 + f(*l*, *r* – 1), if we remove the last character;

However, the last two conditions can be included in the following: to solve the problem on the segment [*l*; *r*] let’s solve the problem separately on segments [*l*; *i*] and [*i* + 1; *r*] (1 ≤ *i* < *r*) and take the smallest result:

$$f(l, r) = \min_{1 \le i < r} (f(l, i) + f(i+1, r))$$



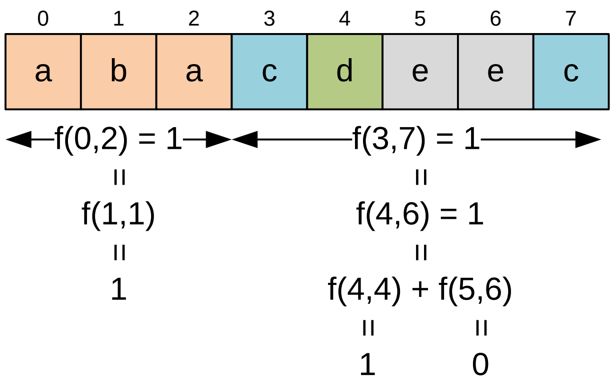
For example, the case of removing the first character from the string is equivalent to *i* = *l* (then f(*l*, *l*) = 1), and the case of removing the last character is equivalent to *i* = *r* – 1 (f (*r*, *r*) = 1).



The answer to the problem is dp[0][n – 1] = f(0, n – 1), where n is the length of the input string.

Consider the sample given in the problem statement. We have:

$$f(0, 7) = f(0, 2) + f(3, 7) = 1 + 1 = 2$$



Declare the arrays.

```
#define MAX 101
#define INF 0x3F3F3F3F
int dp[MAX][MAX];
string s;
```

Let f(l, r) be the solution of the problem on the segment [l; r].

```
int f(int l, int r)
{
  if (l > r) return 0;
  if (l == r) return 1;
  if (dp[l][r] != INF) return dp[l][r];
  int &res = dp[l][r];

  if (s[l] == s[r])
    res = min(res, f(l + 1, r - 1));

  for (int i = l; i < r; i++)
    res = min(res, f(l, i) + f(i + 1, r));
  return res;
}
```

The main part of the program. Read the line.

```
cin >> s;
memset(dp,0x3F,sizeof(dp));
```

Compute and print the answer f(0, n – 1), where n is the length of string s.

```
    printf("%d\n",f(0, s.size() - 1));
```

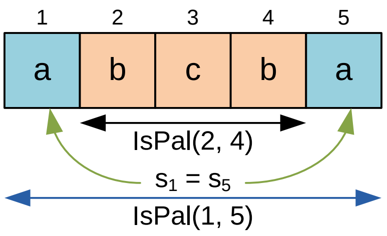
Problems with palindromes

E-OLYMP **470. Super palindromes** The palindrome is a string longer than one character, that reads the same right to left and left to right. The *super palindrome* is a string that can be represented as a concatenation of one or more palindromes. Given the string *s*. Find the number of substrings in *s* that are super palindromes.

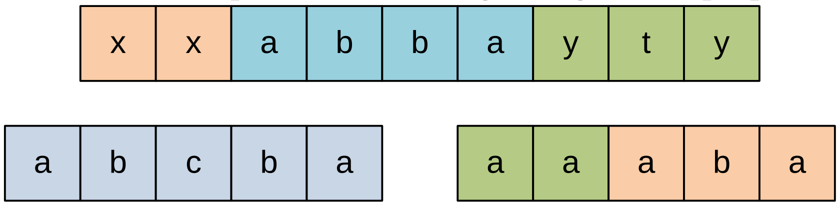
► Let *s* be the input string. The substring $s_i \dots s_j$ is a palindrome, if:

- $i \geq j$ (substring is empty or consists of one character);
- $s_i = s_j$ and $s_{i+1} \dots s_{j-1}$ is a palindrome;

Let function IsPal(i, j) returns 1, if $s_i \dots s_j$ is a palindrome, and 0 otherwise. Memoize the values of IsPal(i, j) in pal[i][j].

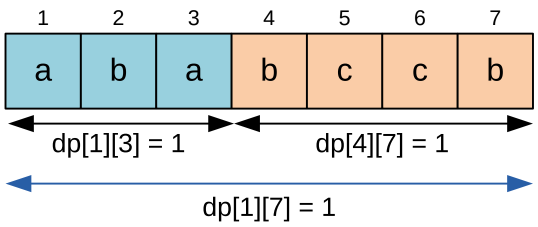


A string is a super palindrome if it can be represented as the concatenation of one or more palindromes. For example, the following strings are superpalindromes:



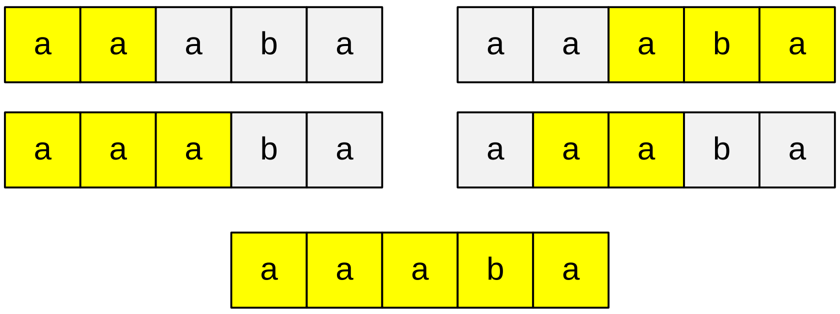
Let dp[i][j] = 1, if substring $s_i \dots s_j$ ($i < j$) is a super palindrome and dp[i][j] = 0 otherwise. Iterate over the pairs (i, j) for $0 \leq i < j < n$ and if the substring $s_i \dots s_j$ is a palindrome, then it is a super palindrome, set dp[i][j] = 1. Note that dp[i][j] = 0 for $i \geq j$. A word of one letter is not considered a palindrome, therefore, as a special case, we have dp[i][i] = 0.

For each pair (i, j), iterate over all possible values of k ($i < k < j - 1$) and if $s_i \dots s_k$ and $s_{k+1} \dots s_j$ are super palindromes (they consist of more than one symbol due to the restriction on k), then $s_i \dots s_j$ is a super palindrome.



It remains to compute the number of pairs (i, j) for which $i < j$ and dp[i][j] = 1. This number is the answer.

There are 5 substrings for *aaaba*, which are super palindromes.



Declare the input string *s* and arrays.

```
#define MAX 1010
char s[MAX];
int dp[MAX][MAX], pal[MAX][MAX];
```

Implement the recursive function ***IsPal***(*i*, *j*), which returns 1 if $s_i \dots s_j$ is a palindrome. Otherwise, the function returns 0. The substring $s_i \dots s_j$ is a palindrome if $s_i = s_j$ and $s_{i+1} \dots s_{j-1}$ is a palindrome. Store the values IsPal(*i*, *j*) in pal[*i*][*j*].

```
int IsPal(int i, int j)
{
  if (i >= j) return pal[i][j] = 1;
  if (pal[i][j] != -1) return pal[i][j];
  return pal[i][j] = (s[i] == s[j]) && IsPal(i+1,j-1);
}
```

Function ***f***(*i*, *j*) returns 1, if $s_i \dots s_j$ is a super palindrome.

```
int f(int i, int j)
{
```

Super palindrome must contain more than one symbol.

```
  if (i == j) return dp[i][j] = 0;
```

If ***f***(*i*, *j*) is already computed, return its value.

```
  if (dp[i][j] != -1) return dp[i][j];
```

If a substring $s_i \dots s_j$ ($i < j$) is a palindrome, then it is also a super palindrome.

```
  if (IsPal(i,j)) return dp[i][j] = 1;
```

If $s_i \dots s_k$ ($i < k$) is a palindrome, and $s_{k+1} \dots s_j$ ($k + 1 < j$) is a super palindrome, then $s_i \dots s_j$ is a super palindrome.

```
  for(int k = i + 1; k < j - 1; k++)
    if(IsPal(i,k) && f(k + 1,j)) return dp[i][j] = 1;
```

If none of the above conditions is satisfied, then $s_i \dots s_j$ is not a super palindrome.

```
  return dp[i][j] = 0;
}
```

The main part of the program. Read the input string *s*. Initialize the *dp* and *pal* arrays.

```
gets(s); n = strlen(s);
memset(dp,-1,sizeof(dp));
memset(pal,-1,sizeof(pal));
```

In the variable *res*, count the number of super palindromes.

```
res = 0;
for(i = 0; i < n; i++)
for(j = i + 1; j < n; j++)
  res += f(i,j);
```

Print the answer.

```
printf("%d\n",res);
```

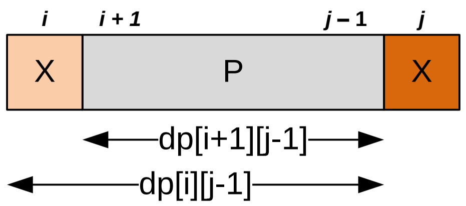
E-OLYMP **873. Palindromes** Non-empty string containing a certain word is called palindrome if it reads the same from left to right and from right to left.

Let we are given a word *s*, consisting of *n* uppercase letters of Latin alphabet. Deleting from the word a certain set of characters, you can get the palindrome string. Find the number of ways to delete from the word some (possibly empty) set of symbols so that the resulting string is a palindrome. Ways in different order of deleting characters are considered equal.

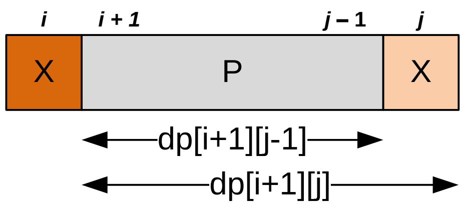
► Let dp[i][j] stores the number of palindromes that can be obtained from the substring $s_i \ldots s_j$ by deleting letters. Then dp[i][i] = 1, since a word of one character is a palindrome.

Let $s_i = s_j$ = X, substring $s_i \ldots s_j$ has the form XPX. Here P denotes the substring $s_{i+1} \ldots s_{j-1}$. Split the palindromes of the string XPX into non-overlapping groups:

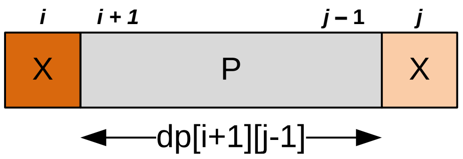
- include the left X and do not include the right X. The number of palindromes equals to the number of palindromes in string XP minus the number of palindromes in P, that equals to dp[i][j – 1] – dp[i + 1][j – 1];



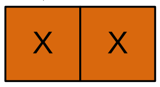
- include the right X and do not include the left X. The number of palindromes equals to the number of palindromes in string PX minus the number of palindromes in P, that equals to dp[i + 1][j] – dp[i + 1][j – 1];



- the palindromes of the string P. Their number equals to dp[i + 1][j – 1]. However, with each palindrome Q of string P, we can construct the XQX palindrome. The number of palindromes of the form XQX will also be dp[i + 1][j – 1].



- palindrome XX (one palindrome).



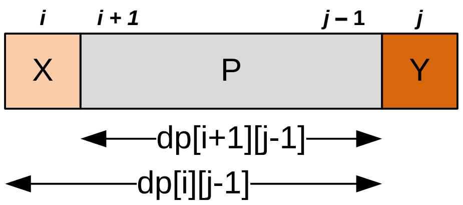
The total number of palindromes for the case $s_i = s_j$ is

$$(\text{dp}[i][j-1] - \text{dp}[i+1][j-1]) +$$
$$(\text{dp}[i+1][j] - \text{dp}[i+1][j-1]) +$$
$$2 * \text{dp}[i+1][j-1] +$$
$$1$$

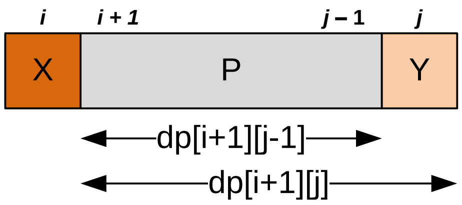
= dp[i][j – 1] + dp[i + 1][j] + 1.

Let $s_i \neq s_j$, substring $s_i \ldots s_j$ has the form XPY (s_i = X, s_j = Y). Split the palindromes of the string XPY into non-overlapping groups:

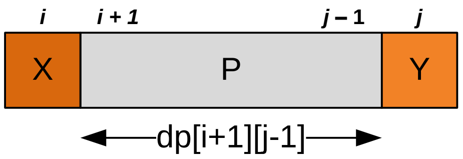
- include the symbol X and do not include the symbol Y. The number of palindromes equals to the number of palindromes in string XP minus the number of palindromes in P, that equals to dp[i][j – 1] – dp[i + 1][j – 1];



- include the symbol Y and do not include symbol X. The number of palindromes equals to the number of palindromes in string PY minus the number of palindromes in P, that equals to dp[i + 1][j] – dp[i + 1][j – 1];



- palindromes of the string P. Their number is dp[i + 1][j – 1].

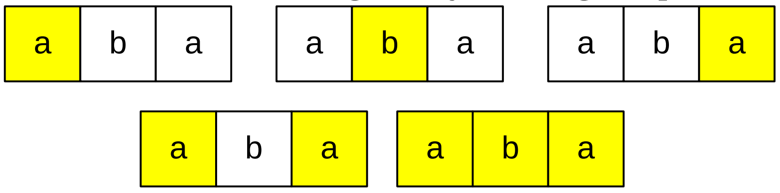


The total number of palindromes for the case $s_i \neq s_j$ is

$$(\text{dp}[i][j-1] - \text{dp}[i+1][j-1]) +$$

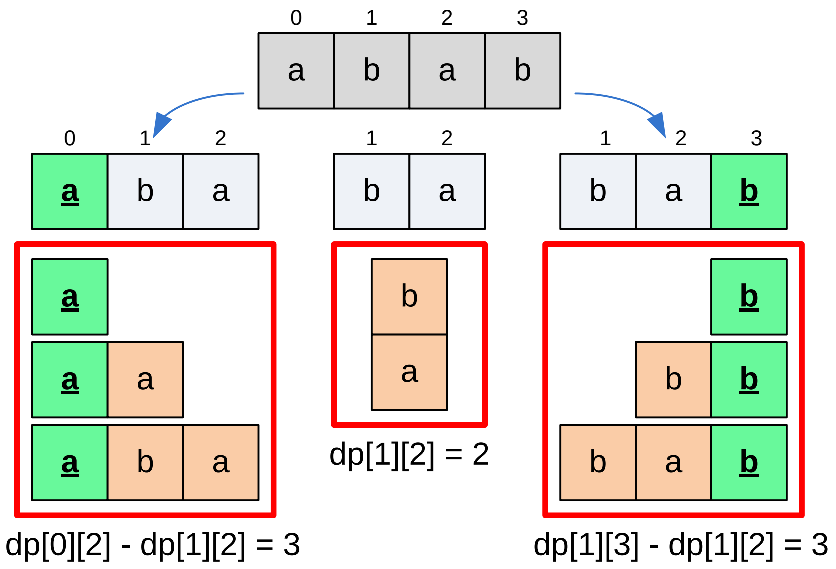
$$(\text{dp}[i + 1][j] - \text{dp}[i + 1][j - 1]) +$$
$$\text{dp}[i + 1][j - 1]$$
$= \text{dp}[i][j - 1] + \text{dp}[i + 1][j] - \text{dp}[i + 1][j - 1]$.

By deleting the letters from the string *aba*, you can get 5 palindromes.



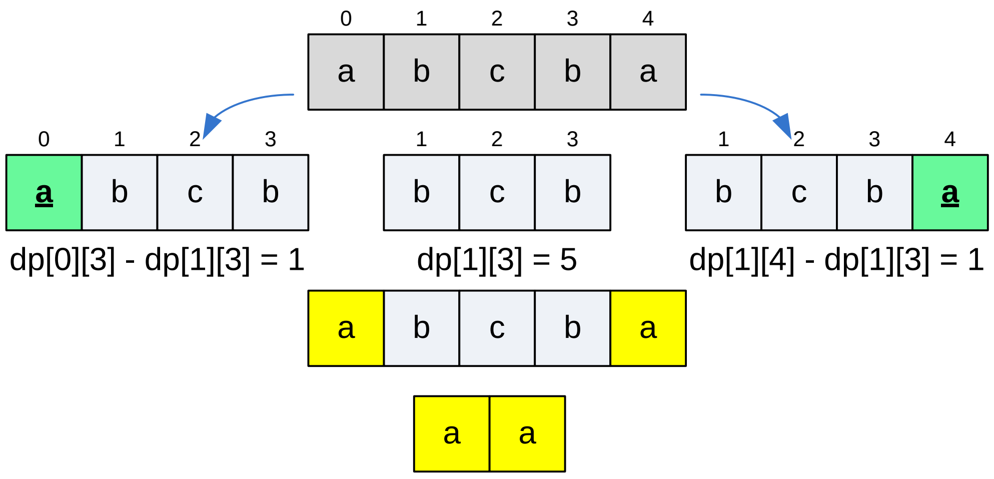
Consider the string *abab* = $s_0s_1s_2s_3$. Since $s_0 \neq s_3$, the substring $s_0 \ldots s_3$ has the form XPY. Hence dp[0][3] =

$$(\text{dp}[0][2] - \text{dp}[1][2]) +$$
$$(\text{dp}[1][3] - \text{dp}[1][2]) +$$
$$\text{dp}[1][2] =$$
$= \text{dp}[0][2] + \text{dp}[1][3] - \text{dp}[1][2] = 5 + 5 - 2 = 8$.



Consider the string *abcba* = $s_0s_1s_2s_3s_4$. Since $s_0 = s_4$, the substring $s_0 \ldots s_4$ has the form XPX. Hence dp[0][4] =

$$(\text{dp}[0][3] - \text{dp}[1][3]) +$$
$$(\text{dp}[1][4] - \text{dp}[1][3]) +$$
$$2 * \text{dp}[1][3] +$$
$$1 =$$
$= \text{dp}[0][3] + \text{dp}[1][4] + 1 = 6 + 6 + 1 = 13$.



```
#include <stdio.h>
#include <string.h>
#define MAX 61
```

Store the input string in the array *s*. Declare an array dp.

```
char s[MAX];
long long dp[MAX][MAX];
int i, j, len, n;

long long f(int i, int j)
{
```

If *i* > *j*, there is no palindromes.

```
  if (i > j) return 0;
```

A word of one character is a palindrome, set dp[*i*][*i*] = 1.

```
  if (i == j) return dp[i][j] = 1;
```

If the value of dp[*i*][*j*] is already computed, return it.

```
  if (dp[i][j] != -1) return dp[i][j];
```

Compute the value of dp[*i*][*j*] depending on whether the symbols s_i and s_j are the same.

```
  if (s[i] == s[j])
    dp[i][j] = f(i + 1, j) + f(i, j - 1) + 1;
  else
    dp[i][j] = f(i + 1, j) + f(i, j - 1) - f(i + 1, j - 1);
  return dp[i][j];
}

int main(void)
{
```

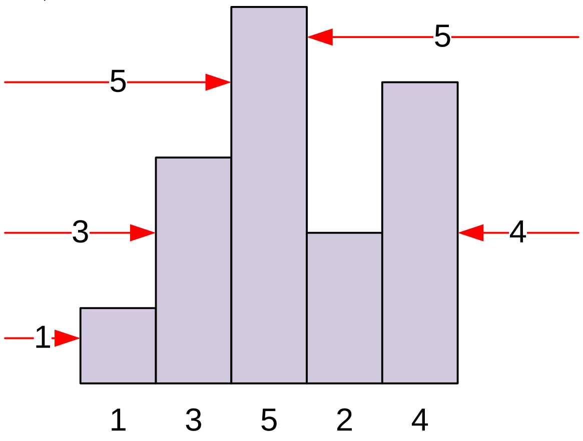
The main part of the program. Read the input string *s*. Initialize array dp.

```
  gets(s); n = strlen(s);
  memset(dp, -1, sizeof(dp));
```

Print the answer.

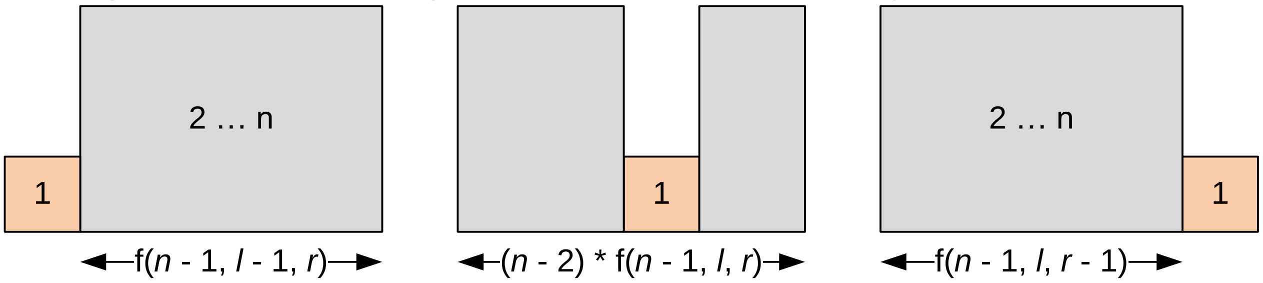
```
  printf("%lld\n", f(0, n - 1));
  return 0;
}
```

E-OLYMP **1535. Skyscrapers** The skyline of the city has *n* buildings all in a straight line; each building has a distinct height between 1 and *n*, inclusive. The building at index *i* is considered visible from the left if there is no building with a smaller index that is taller. Similarly, a building is visible from the right if there is no taller building with a higher index. For example, if the buildings in order are {1, 3, 5, 2, 4}, then three buildings are visible from the left (1, 3, 5), but only two are visible from the right (4 and 5).



You will be given the total number of buildings *n*, *l* buildings visible from the left, and *r* buildings visible from the right. Find the number of permutations of the buildings that are consistent with these values.

► Suppose it remains to arrange *n* houses, *l* of which should be visible from the left, and *r* from the right. Let its possible to do in f(*n*, *l*, *r*) ways. Consider the house with the smallest height. If you put it on the left, then it will always be visible, and the remaining houses can be arranged in f(*n* – 1, *l* – 1, *r*) ways. If the house with the smallest height is placed on the right, then it will always remain visible on the right, the rest of the houses can be arranged in f(*n* – 1, *l*, *r* – 1) ways. The smallest house can be placed between the other houses in *n* – 2 ways. In this case it will not be visible, so the remaining houses can be arranged in (*n* – 2) * f(*n* – 1, *l*, *r*) ways.

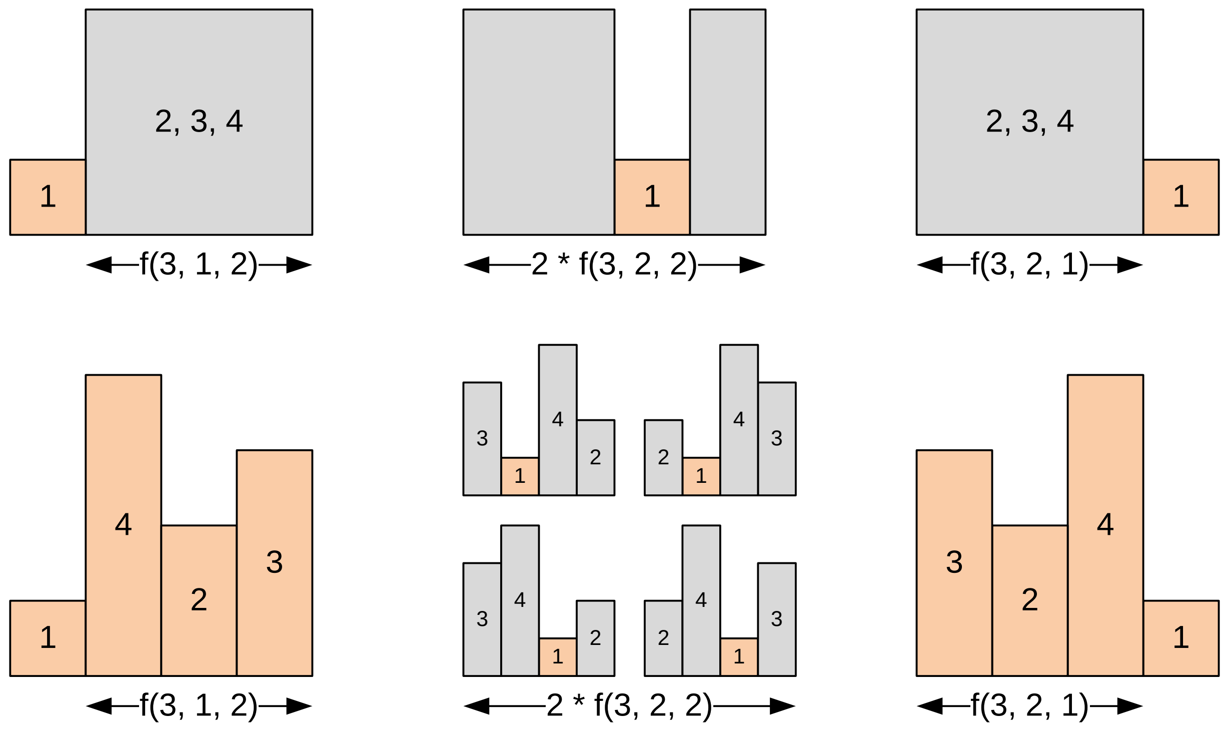


Note that for *n* = 1 the only possible arrangement will be only for *l* = *r* = 1. We obtain the recurrence relation:

$$f(n, l, r) = f(n-1, l-1, r) + f(n-1, l, r-1) + (n-2) * f(n-1, l, r),$$
$$f(1, 1, 1) = 1,$$
$$f(1, x, y) = 0, \text{ when } x \text{ and } y \text{ are not simultaneously equal to } 1$$

Construct all permutations of houses for $n = 4$, $l = 2$, $r = 2$.

$$f(4, 2, 2) = f(3, 1, 2) + 2 * f(3, 2, 2) + f(3, 2, 1) = 1 + 2 * 2 + 1 = 6$$



Declare a three-dimensional array dp, where dp[i][j][k] will store the value f(i, j, k).

```
#define MAX 101
int dp[MAX][MAX][MAX];
```

Function ***f*** returns the number of ways that n houses can be arranged so that l can be visible from the left and r houses can be visible from the right.

```
long long f(int n, int l, int r)
{
  if (n == 1) return (l == 1 && r == 1) ? 1 : 0;
  if ((l < 1) || (r < 1)) return 0;
  if (dp[n][l][r] != -1) return dp[n][l][r];
  dp[n][l][r] = (f(n - 1, l - 1, r) + f(n - 1, l, r - 1) + (n - 2)*f(n
- 1, l, r)) % 1000000007;
  return dp[n][l][r];
}
```

The main part of the program. Read the input data. Compute and print the answer.

```
while (scanf("%d %d %d", &n, &l, &r) == 3)
{
  memset(dp, -1, sizeof(dp));
  res = f(n, l, r);
  printf("%d\n", res);
}
```