Dynamic programming

Coin exchange problems

E-OLYMP 1283. Banking reform According to unconfirmed rumors the banking reform is approaching, and you decide to make for it some proposals – and what if they suddenly will be accepted, and even you will be payed for these ideas?

Your idea is to change the denominations in coin circulation. By your opinion, it should be coins 1, 5, 10, 25 and 50 and how to call the petty cash – let the central bank decides.

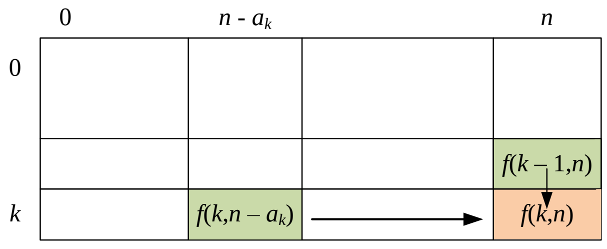
However, the Central Bank has immediately demanded you to provide information about how many ways are there to represent in cash the given amount of money up to 7489 inclusive. Why up to this amount? And how the bank will name them? But who knows: the bankers have their quirks, we also name them for simplicity, just coins.

For example, an amount of 11 units can be represented in the form 10 * 1 coin + 1 * 1 coin, or 5 * 2 coins + 1 * 1 coin, or 5 * 1 coin + 1 * 6 coins or 11 * 1 coin, i.e. total in four ways.

Your task is to write a program to count the number of ways – to quickly respond to any request from the bankers.

► Let f(k, n) be the number of ways to make up the sum n using the first k denominations of coins. It equals to the number of ways to make up the sum of n using the first (k – 1) denominations only (that is, without using the k-th denomination) plus the number of ways to make up the sum (n – a_k) using k denominations. We have the relation:

$$f(k, n) = f(k - 1, n) + f(k, n - a_k)$$



Initially set f(0, 0) = 1, f(0, n) = 0, n > 0.

The sum s = 11 can be represented in 4 ways:
1) 10 + 1
2) 5 + 5 + 1
3) 5 + 1 + 1 + 1 + 1 + 1 + 1
4) 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

	$k \setminus n$	0	1	2	3	4	5	6	7	8	9	10	11
start	0	1	0	0	0	0	0	0	0	0	0	0	0
c = 1	1	1	1	1	1	1	1	1	1	1	1	1	1
c = 5	2	1	1	1	1	1	2	2	2	2	2	3	3
c = 10	3	1	1	1	1	1	2	2	2	2	2	4	4

The denominations of 25 and 50 cents will not affect the result for the amount of 11 cents.

For example, consider the equality f (2, 10) = f (1, 10) + f (2, 5). It means that the number of ways to represent number 10 using two denominations of coins 1 and 5 includes:

- f(1, 10) = 1, number of ways to represent 10 with one denomination 1 (10 = 1 + 1 + 1 + … + 1).
- f(2, 5) = 2, number of ways to represent 5 with two denominations 1 and 5 (5 = 1 + 1 + 1 + 1 + 1 and 5 = 5).

Declare the arrays.

```
#define MAX 7500
long long mas[MAX];
int coins[11] = {1,5,10,25,50};
```

The main part of the program.

```
memset(mas,0,sizeof(mas)); mas[0] = 1;
for(i = 0; i < 5; i++)
{
  for(j = coins[i]; j < MAX; j++)
    mas[j] += mas[j - coins[i]];
}
```

Read the input data and print the result.

```
while(scanf("%d",&n) == 1)
  printf("%d\n",mas[n]);
```

Recursion with memoization.

```
#include <stdio.h>
#include <string.h>

int i, j, n;
int dp[6][7500];
int a[6] = { 0,1,5,10,25,50 };

int f(int k, int n)
{
  if (k == 0 && n == 0) return 1;
  if (k == 0 || n < 0) return 0;
  if (dp[k][n] != -1) return dp[k][n];
  return dp[k][n] = f(k - 1, n) + f(k, n - a[k]);
}

int main(void)
{
  memset(dp, -1, sizeof(dp));
  while (scanf("%d", &n) == 1)
```

```
    printf("%d\n", f(5, n));
  return 0;
}
```

E-OLYMP [**9614. Coin change problem**]() You are working at the cash counter at a fun-fair, and you have different types of coins available to you in infinite quantities. The value of each coin is already given. Can you determine the number of ways of making change for a particular number of units using the given types of coins?

For example, if you have 4 types of coins, and the value of each type is given as 8, 3, 1, 2 respectively, you can make change for 3 units in three ways: {1, 1, 1}, {1, 2}, and {3}.

► Use the idea of solution from the previous problem.

E-OLYMP [**9615. Frobenius coin problem**]() Given two coins of denominations x and y respectively. Find the largest amount S that cannot be obtained using these two coins (assuming infinite supply of coins) and the total number T of such non obtainable amounts. If no such value exists print “NA”.

► If GCD(x, y) > 1, then there are an infinite number of sums that cannot be paid with two denominations. In this case, print “NA”.

The largest sum S that cannot be paid with denominations x and y is $x * y – x – y$.

The total number T of sums that are not representable by the available coins is $(x – 1) * (y – 1) / 2$.

Consider the sample $x = 2$, $y = 5$. In ths case

$$S = 2 * 5 – 2 – 5 = 3,$$

$$T = 1 * 4 / 2 = 2$$

The non-representable sumss are 1 and 3 (two sums).

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.

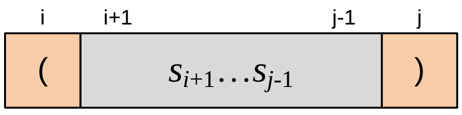
As examples, "([{}])", "" and "(){}[]" are well formed strings and "([}]", "([)]" and "{" are malformed strings.

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:

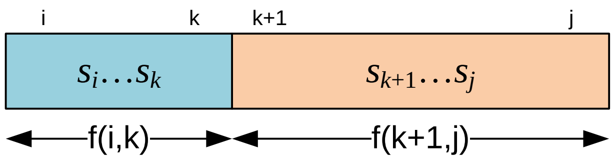
1. f(i, j) = 0 for $i > j$, since in this case the substring will be empty.

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 $s_{i+1} \ldots s_{j-1}$.

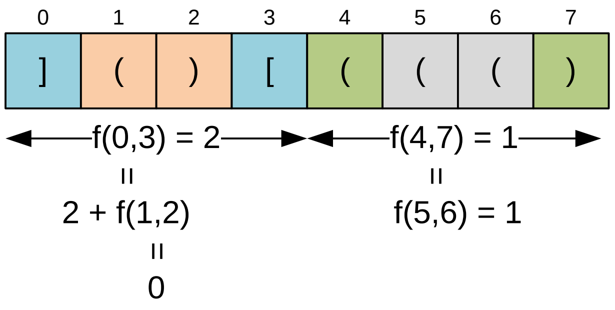


Function enc(s_i ,s_j) returns:
a) 0, if the characters s_i and s_j are matching opening and closing parentheses;
б) 2, if s_i is a closing parenthesis and s_j is an opening parenthesis;
в) 1 otherwise. In this case, it is enough to change one of the symbols s_i or s_j so that they form the correct pair 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.

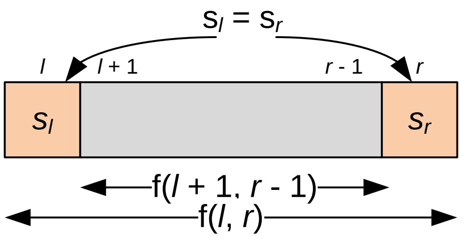
```
  res = f(0, s.size() - 1);
  cout << res << endl;
}
```

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;

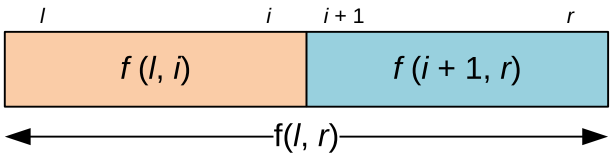
- f(l, r) = f(l + 1, r – 1), if s[l] = s[r]. If the leftmost and the rightmost characters are the same, then the inner part should be deleted, after which these characters become adjacent and they can be deleted by applying the operation;



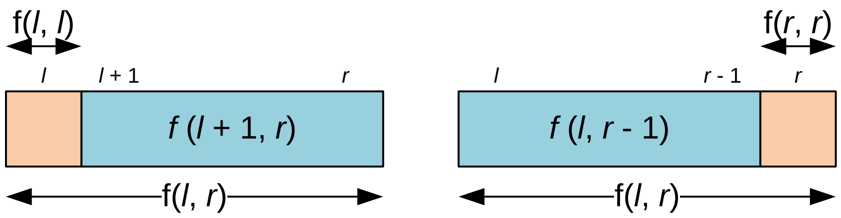
- f(l, r) = 1 + f(l + 1, r), if we remove the first character;
- f(l, r) = 1 + f(l, r – 1), if we remove the last character;

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] ($1 \le i < r$) and take the smallest result:

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



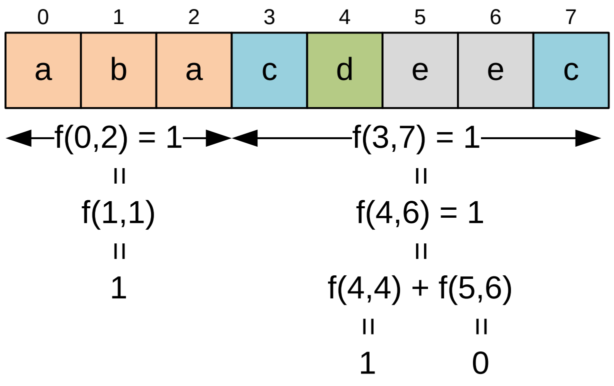
For example, the case of removing the first character from the string is equivalent to i = l (then f(l, l) = 1), and the case of removing the last character is equivalent to i = r – 1 (f (r, r) = 1).



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));
```

E-OLYMP [5205. Parentheses](#) The pattern is given. It consists of parentheses and question marks.

Find in how many ways one can replace the question marks with parentheses to obtain the correct bracket sequence.

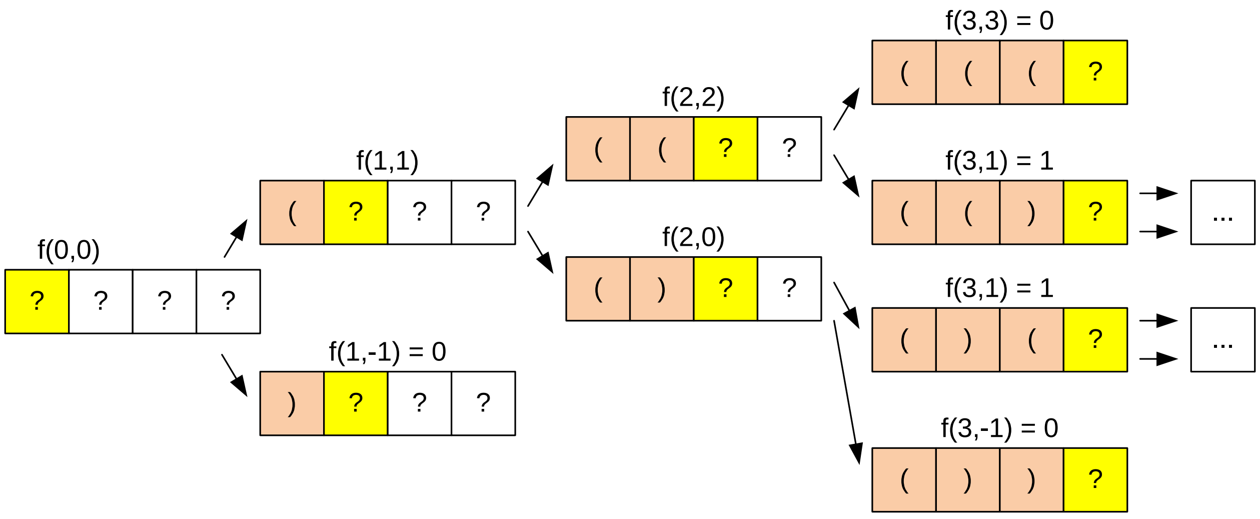
► Let f(*n*, *k*) be the number of correct parentheses starting at position *n*, if *k* open parentheses have already been encountered. Then the answer to the problem will be the value f(0, 0). Store the value of f(*n*, *k*) in dp[*n*][*k*].

Let *s* be the input string. Then:

- if s[*n*] = ‘(‘, then f(*n*, *k*) = f(*n* + 1, *k* + 1). There will be *k* open parentheses before position *n*, if there will be *k* + 1 open parenthesis before position *n* + 1;
- if s[*n*] = ‘)‘, then f(*n*, *k*) = f(*n* + 1, *k* – 1) . There will be *k* open parentheses before position *n*, if there will be *k* – 1 open parenthesis before position *n* + 1;
- if s[*n*] = ‘?‘, then at the *n*-th place one can put both opening and closing parentheses. Therefore f(*n*, *k*) = (f(*n* + 1, *k* + 1) + f(*n* + 1, *k* – 1)) % 301907;

It remains to write down the conditions for the base case of dynamic:

- if at some stage becomes $k < 0$, then the number of closed brackets turns out to be more than the number of open ones, so we exit this branch of calculations, return 0;
- if we have reached the end of the string $s_0s_1 \dots s_{n-1}$ (index numbering starts from zero), then dp[n][k] = 1 if only $k = 0$ (when the number of open and closed brackets is the same). For $k > 0$, set dp[n][k] = 0.



Declare the arrays.

```
#define MAX 2010
char s[MAX];
int dp[MAX][MAX];
int len;
```

Implement the function f(n, k).

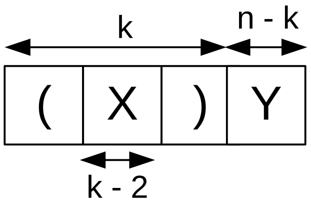
```
int f(int n, int k)
{
  if(k < 0) return 0;
  if(n == len) return (k == 0);
  if(dp[n][k] != -1) return dp[n][k];

  if(s[n] == '(') return dp[n][k] = f(n+1, k+1);
  if(s[n] == ')') return dp[n][k] = f(n+1, k-1);
  return dp[n][k] = (f(n+1, k-1) + f(n+1, k+1)) % 301907;
}
```

The main part of the program. Read the line and print the number of correct bracket sequences.

```
gets(s); len = strlen(s);
memset(dp,-1,sizeof(dp));
printf("%d\n",f(0,0));
```

E-OLYMP 401. Sharik's letter from Prostokvashino Find the correctly formed bracket expression of length n and depth d.

► Any non-empty bracket expression A of length n and depth no greater than d can be expressed as (X)Y, where X and Y some expressions, possibly empty. Let the length of expression (X) equals to k. Then the length of X is $k - 2$, and the length of Y is $n - k$. Obviously that $2 \le k \le n$ and k can take only even values. When $k = 2$ the expression X is empty, and when $k = n$ the expression Y is empty. Note also, since the depth of expression A is not greater than d, then the depth of X is not greater than $d - 1$, and the depth of Y is not greater than d.



Denote $f(n, d)$ the number of ways to get a well-formed bracket expression of length n and depth not greater than d. Then we have $f(k - 2, d - 1)$ ways to represent the expression X and $f(n - k, d)$ ways to represent the expression Y. Using the multiplication rule we can say that for a fixed k the expression (X)Y can be represented in $f(k - 2, d - 1) * f(n - k, d)$ ways. So

$$f(n, d) = \sum_{\substack{2 \le k \le n \\ \text{k is even}}} f(k-2, d-1) \cdot f(n-k, d)$$

Since the problem requires to find the number of ways to get a well-formed bracket expression of length n and depth exactly d, then the answer will $g(n, d) = f(n, d) - f(n, d - 1)$.

Separately you must handle the following cases:

- If $d > n / 2$, then $g(n, d) = 0$;
- If $d = n / 2$, then we have a unique bracket expression (((…))), so $g(n, n / 2) = 1$;
- If $d = 1$, then we have a unique bracket expression ()()…()(), so $g(n, 1) = 1$;

$f(2, 2) = \sum_{k=2} f(k-2, 1) \cdot f(2-k, 2) = f(0, 1) * f(0, 2) = 1 * 1 = 1$. If one can represent the bracket expression of length 2 and depth not greater than 2 as (X)Y, then the factor $f(0, 1)$ corresponds to the number of ways to represent X, and the factor $f(0, 2)$ corresponds to the number of ways to represent Y. These factors equal to one, and X and Y corresponds an empty expression. So for $f(2, 2)$ corresponds only one expression ().

$f(4, 2) = \sum_{k=2,4} f(k-2, 1) \cdot f(4-k, 2) =$

$$f(0, 1) * f(2, 2) + f(2, 1) * f(0, 2) = 1 * 1 + 1 * 1 = 2$$

The summand $f(0, 1) * f(2, 2)$ corresponds to expression ()(), and the summand $f(2, 1) * f(0, 2)$ corresponds to expression (()).

$f(6, 2) = \sum_{k=2,4,6} f(k-2, 1) \cdot f(6-k, 2) =$

$$f(0, 1) * f(4, 2) + f(2, 1) * f(2, 2) + f(4, 1) * f(0, 2) = 1 * 2 + 1 * 1 + 1 * 1 = 4$$

Summand	**The corresponding bracket expressions**
$f(0, 1) * f(4, 2)$	()()(), ()(())
$f(2, 1) * f(2, 2)$	(())()

$f(4, 1) * f(0, 2)$	(()())

The number of correctly formed bracket expressions of length 6 and of depth exactly 2 equals to

$$g(6, 2) = f(6, 2) – f(6, 1) = 4 – 1 = 3$$

They are: ()(()),(())(),(()()).

The value f(*n*, *d*) we shall keep in array of long numbers ff.

```
BigInteger ff[301][151];
```

The function f computes the value f(*n*, *d*). Separately handle the cases when $n < 0$ or $d < 0$ (the function f returns 0). If $n = 0$, then f(0, *d*) is assumed to be equal to 1 for any *d* (this is the case when either X or Y is empty).

```
BigInteger f(long long n, long long d)
{
  long long k;
  BigInteger &s = ff[n][d];
  if ((n < 0) || (d < 0)) return 0;
  if (!n) return ff[n][d] = BigInteger(1);
  if (ff[n][d] >= 0) return ff[n][d];
  for(s = 0, k = 2; k <= n; k += 2)
    s = s + f(k - 2,d - 1) * f(n - k,d);
  return s;
}
```

The main part of the program. First handle the special cases.

```
memset(ff,-1,sizeof(ff));
while(scanf("%lld %lld",&n,&d) == 2)
{
   if (d > n / 2) res = BigInteger(0); else
   if ((d == n / 2) || (d == 1)) res = BigInteger(1); else
   res = f(n,d) - f(n,d-1);
   res.print();printf("\n");
}
```

Java realization

```
import java.util.*;
import java.math.*;

public class Main
{
  static BigInteger dp[][];

  static BigInteger f(int n, int d)
  {
    BigInteger s = BigInteger.ZERO;
    if ((n < 0) || (d < 0)) return BigInteger.ZERO;
    if (n == 0) return dp[n][d] = BigInteger.ONE;
    if (dp[n][d].compareTo(BigInteger.ZERO) >= 0) return dp[n][d];
    for(int k = 2; k <= n; k += 2)
```

```
        s = s.add(f(k - 2,d - 1).multiply(f(n - k,d)));
      return dp[n][d] = s;
    }

    public static void main(String[] args)
    {
      Scanner con = new Scanner(System.in);
      while(con.hasNextInt())
      {
        int n = con.nextInt();
        int d = con.nextInt();
        dp = new BigInteger[n+1][d+1];

        for(int i = 0; i <= n; i++)
        for(int j = 0; j <= d; j++)
          dp[i][j] = new BigInteger("-1");

        BigInteger res = new BigInteger("0");

        if (d > n / 2) res = BigInteger.ZERO; else
        if ((d == n / 2) || (d == 1)) res = BigInteger.ONE; else
        res = f(n,d).subtract(f(n,d-1));

        System.out.println(res);
      }
      con.close();
    }
  }
```

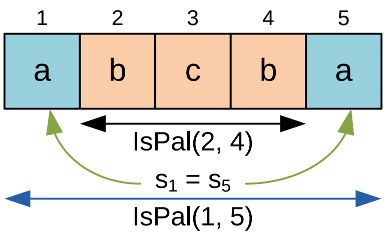
Problems with palindromes

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.

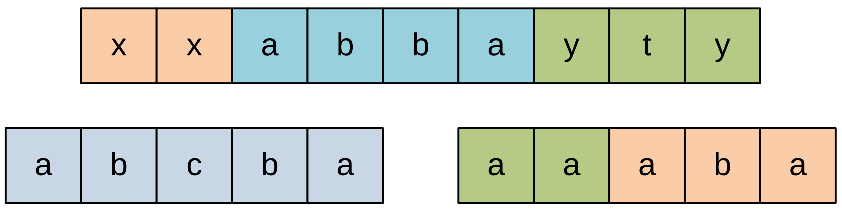
► Let *s* be the input string. The substring $s_i \dots s_j$ is a palindrome, if:

- $i \geq j$ (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*].

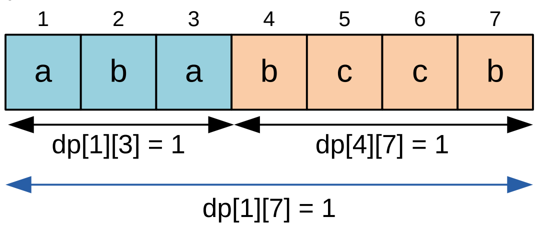


A string is a super palindrome if it can be represented as the concatenation of one or more palindromes. For example, the following strings are superpalindromes:



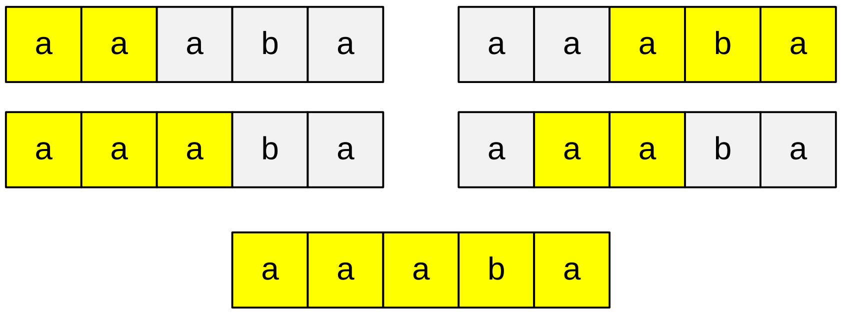
Let dp[*i*][*j*] = 1, if substring $s_i \ldots s_j$ ($i < j$) is a super palindrome and dp[*i*][*j*] = 0 otherwise. Iterate over the pairs (i, j) for $0 \le i < j < n$ and if the substring $s_i \ldots s_j$ is a palindrome, then it is a super palindrome, set dp[*i*][*j*] = 1. Note that dp[*i*][*j*] = 0 for $i \ge j$. 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 $s_i \ldots s_k$ and $s_{k+1} \ldots s_j$ are super palindromes (they consist of more than one symbol due to the restriction on k), then $s_i \ldots s_j$ 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 arrays.

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

Implement the recursive function ***IsPal***(i, j), which returns 1 if $s_i \ldots s_j$ is a palindrome. Otherwise, the function returns 0. The substring $s_i \ldots s_j$ is a palindrome if $s_i = s_j$ and $s_{i+1} \ldots s_{j-1}$ 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);
}
```

The main part of the program. Read the input string.

```
gets(s); n = strlen(s);
memset(dp,0,sizeof(dp));
memset(pal,-1,sizeof(pal));
```

Iterate over the pairs (i, $i + len$) in ascending order of interval lengths.

```
for(len = 1; len < n; len++)
for(i = 0; i + len < n; i++)
{
  j = i + len;
```

Substring $s_i \dots s_j$ contains more than one character. If it is a palindrome, then it is also a super palindrome.

```
  if (IsPal(i,j))
  {
    dp[i][j] = 1;
    continue;
  }
```

If $s_i \dots s_k$ and $s_{k+1} \dots s_j$ are super palindromes, then $s_i \dots s_j$ is a super palindrome.

```
  for(k = i + 1; k < j - 1; k++)
    if(dp[i][k] && dp[k + 1][j])
    {
      dp[i][j] = 1;
      break;
    }
}
```

Count the number of super palindromes.

```
res = 0;
for(i = 0; i < n; i++)
for(j = i+1; j < n; j++)
  res += dp[i][j];
```

Print the answer.

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

Algorithm realization – recursive

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 $s_i \ldots s_j$ is a palindrome. Otherwise, the function returns 0. The substring $s_i \ldots s_j$ is a palindrome if $s_i = s_j$ and $s_{i+1} \ldots s_{j-1}$ 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 $s_i \ldots s_j$ 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 $s_i \ldots s_j$ ($i < j$) is a palindrome, then it is also a super palindrome.

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

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

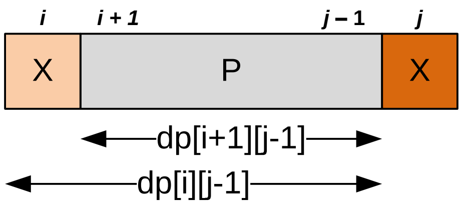
E-OLYMP **873. Palindromes** Non-empty string containing a certain word is called palindrome if it reads the same from left to right and from right to left.

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.

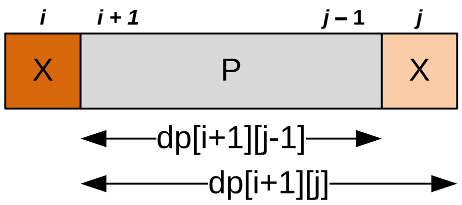
► Let dp[*i*][*j*] stores the number of palindromes that can be obtained from the substring $s_i \ldots s_j$ by deleting letters. Then dp[*i*][*i*] = 1, since a word of one character is a palindrome.

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:

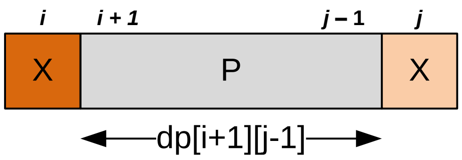
- include the left X and do not include the right X. The number of palindromes equals to the number of palindromes in string XP minus the number of palindromes in P, that equals to dp[*i*][*j* – 1] – dp[*i* + 1][*j* – 1];



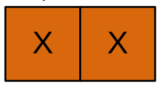
- include the right X and do not include the left X. The number of palindromes equals to the number of palindromes in string PX minus the number of palindromes in P, that equals to dp[*i* + 1][*j*] – dp[*i* + 1][*j* – 1];



- 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).

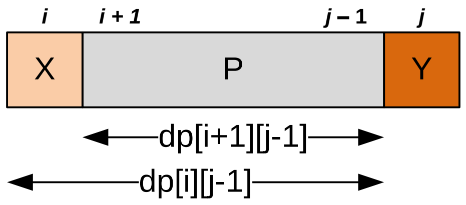


The total number of palindromes for the case $s_i = s_j$ is

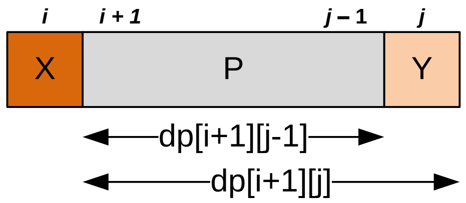
$$(\mathrm{dp}[i][j-1] - \mathrm{dp}[i+1][j-1]) +$$
$$(\mathrm{dp}[i+1][j] - \mathrm{dp}[i+1][j-1]) +$$
$$2 * \mathrm{dp}[i+1][j-1] +$$
$$1$$
$= \mathrm{dp}[i][j-1] + \mathrm{dp}[i+1][j] + 1$.

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:

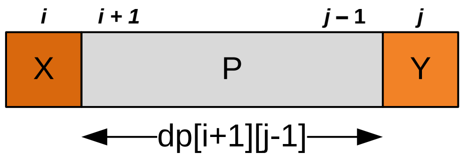
- include the symbol X and do not include the symbol Y. The number of palindromes equals to the number of palindromes in string XP minus the number of palindromes in P, that equals to $\mathrm{dp}[i][j-1] - \mathrm{dp}[i+1][j-1]$;



- 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 $\mathrm{dp}[i+1][j] - \mathrm{dp}[i+1][j-1]$;

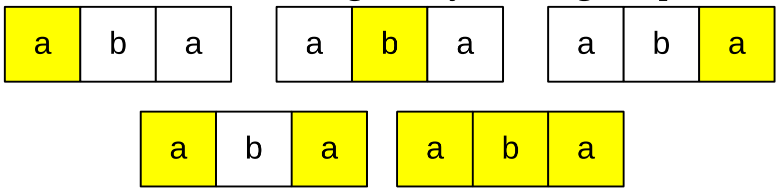


- palindromes of the string P. Their number is $\mathrm{dp}[i+1][j-1]$.



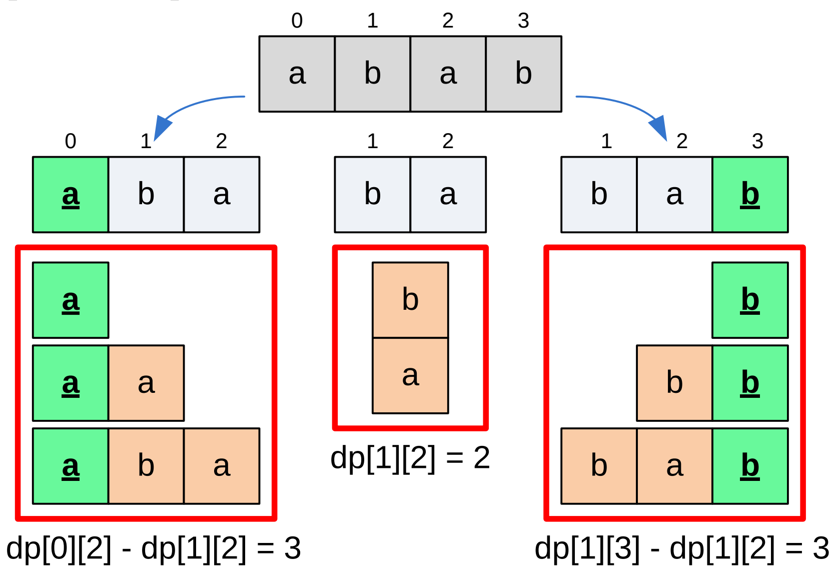
The total number of palindromes for the case $s_i \neq s_j$ is
$$(\mathrm{dp}[i][j-1] - \mathrm{dp}[i+1][j-1]) +$$
$$(\mathrm{dp}[i+1][j] - \mathrm{dp}[i+1][j-1]) +$$
$$\mathrm{dp}[i+1][j-1]$$
$= \mathrm{dp}[i][j-1] + \mathrm{dp}[i+1][j] - \mathrm{dp}[i+1][j-1]$.

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



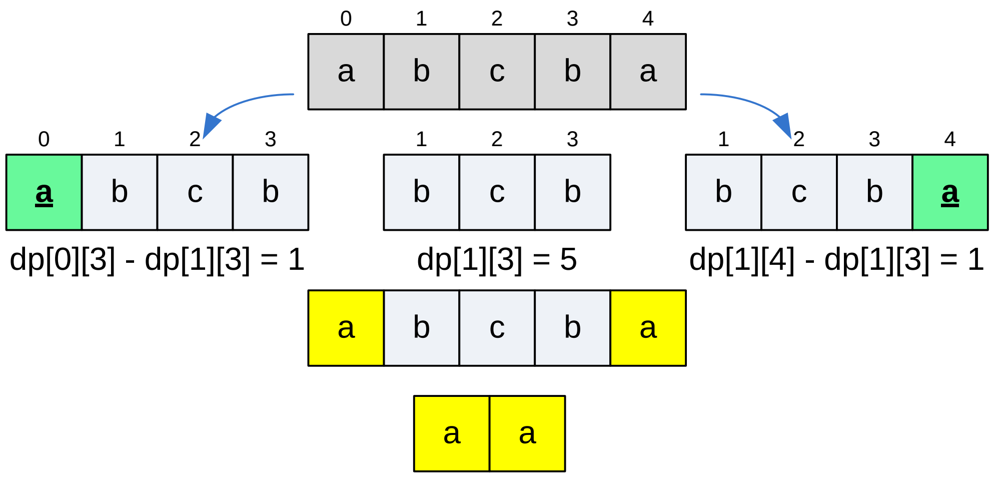
Consider the string *abab* = $s_0s_1s_2s_3$. Since $s_0 \neq s_3$, the substring $s_0 \ldots s_3$ has the form XPY. Hence dp[0][3] =
$$(\mathrm{dp}[0][2] - \mathrm{dp}[1][2]) +$$

(dp[1][3] – dp[1][2]) +
dp[1][2] =
= dp[0][2] + dp[1][3] – dp[1][2] = 5 + 5 – 2 = 8.



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

(dp[0][3] – dp[1][3]) +
(dp[1][4] – dp[1][3]) +
2 * dp[1][3] +
1 =
= dp[0][3] + dp[1][4] + 1 = 6 + 6 + 1 = 13.



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

```
#define MAX 65
char s[MAX];
long long dp[MAX][MAX];
```

Read the input string *s*. Fill dp[*i*][*i*] = 1.

```
gets(s); n = strlen(s);
```

```
memset(dp,0,sizeof(dp));
for(i = 0; i < n; i++) dp[i][i] = 1;
```

Iterate over the lengths of the substrings *len* and their starting positions *i*.

```
for(len = 1; len < n; len++)
for(i = 0; i + len < n; i++)
{
  j = i + len;
```

For each such substring $s_i \ldots s_j$ compute the value dp[*i*][*j*], the number of palindromes that can be obtained from it by removing characters. Since the substrings $s_i \ldots s_j$ are iterated in increasing order of their lengths, the dp values for all subsegments of shorter length have already been calculated.

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

Print the answer.

```
printf("%lld\n",dp[0][n-1]);
```

Algorithm realization – recursion

```
#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;
}
```