习题课2

TEACHING COURSEWARE POWPOINT

授课时间：2025.08.14

目录

01

AK计数

TEACHING
COURSEWARE

TEACH

AK计数

AK47 计数

输入文件 ak47.in 输出文件 ak47.out
时间限制 1000ms 空间限制 128MB

题目描述

AK-47 自动步枪于 1947 年定型，1949 年装备苏联部队。除了大规模装备于苏军外，世界上有许多国家也都进行了仿制或授权生产 AK 步枪系列，其中包括德国、捷克斯洛伐克、前南斯拉夫、匈牙利、中国、波兰、罗马尼亚、保加利亚、埃及、古巴、朝鲜等多个国家。AK-47 的设计思路也影响了以色列、芬兰、中国等多个国家的步枪设计构思路线。按不同统计 AK 系列包括了未经许可生产的仿制品，总产量超过 1 亿支，是世界上产量最高、适用范围最广和改进类型最多的枪械。

现在 Bob 有一个字符串。字符串中有许多数字和大小写字母。Bob 的任务是从中计数有几个不同的子序列，可以形成 AK47 这样一个字符串。

输入格式

第一行一个正整数 n，表示接下来给定的字符串长度。
接下来一行一个字符串，字符串中包括大小写字母和数字。

输出格式

计数字符串中的 AK47 子序列的数量并输出。

AK计数

样例 #1

样例输入 #1

8

AAKK4477

样例输出 #1

16

样例 1 解释

上面这个例子中任意选一个 A 一个 K 一个 4 一个 7 即可。总方案有 16 种。

样例 #2

样例输入 #2

12

AAK4QQK477dd

样例输出 #2

12

样例 2 解释

选第一个 K 的方法有 2*2*2=8 种，选第二个 K 的方法有 2*2=4 种，共计 12 种。

AK计数

数据范围

共 10 个测试点，每个测试点 10 分。

测试点编号	字符串长度 len 范围	其他说明
#1~#3	$1 \le len \le 10$	无
#4~#5	$1 \le len \le 1{,}000$	字符串形如$A \ldots AK \ldots K4 \ldots 47 \ldots 7$
#6~#7		无
#8~#10	$2 \le len \le 100{,}000$	无

AK计数

状态含义
dp[i][j] 代表处理到字符串的第 i 个字符时，当前已匹配出 “AK47” 中第 j 位字符的方案总数 。具体来说：
dp[i][1] 是处理到第 i 个字符时，匹配到 “AK47” 第 1 位（即字符 A）的方案数；
dp[i][2] 是处理到第 i 个字符时，匹配到 “AK47” 第 2 位（即字符 K）的方案数；
dp[i][3] 是处理到第 i 个字符时，匹配到 “AK47” 第 3 位（即字符 4）的方案数；
dp[i][4] 是处理到第 i 个字符时，匹配到 “AK47” 第 4 位（即字符 7）的方案数，
最终答案取 dp[n][4]（n 为字符串长度 ）。

AK计数

状态转移方程
状态转移的核心逻辑是基于前一字符的匹配状态，结合当前字符是否符合 “AK47” 对应位的要求，更新当前状态：

对于 dp[i][1]：若第 i 个字符是 A，则新增独立匹配 A 的方案（+1），同时继承处理到 i-1 字符时匹配 A 的方案数（dp[i-1][1]），即 dp[i][1] = dp[i-1][1] +（当前字符 == 'A'）；
对于 dp[i][2]：若第 i 个字符是 K，则需基于已匹配 A 的方案数（dp[i-1][1]）转移，累加得到新方案数，同时继承处理到 i-1 字符时匹配 K 的方案数（dp[i-1][2]），即 dp[i][2] = dp[i-1][2] +（当前字符 == 'K'）* dp[i-1][1]；
对于 dp[i][3]：若第 i 个字符是 4，需基于已匹配 K 的方案数（dp[i-1][2]）转移，累加得到新方案数，同时继承处理到 i-1 字符时匹配 4 的方案数（dp[i-1][3]），即 dp[i][3] = dp[i-1][3] +（当前字符 == '4'）* dp[i-1][2]；
对于 dp[i][4]：若第 i 个字符是 7，需基于已匹配 4 的方案数（dp[i-1][3]）转移，累加得到新方案数，同时继承处理到 i-1 字符时匹配 7 的方案数（dp[i-1][4]），即 dp[i][4] = dp[i-1][4] +（当前字符 == '7'）* dp[i-1][3] 。
简单来说，每一步先 “继承” 前一位置的状态（不选当前字符时的方案数），再根据当前字符是否匹配 “AK47” 对应位，累加 “新增” 的方案数（需基于前一位已匹配的结果转移）。

AK计数


```cpp
#include <bits/stdc++.h>
using namespace std;
long long dp[100005][5] = {0};
int main() {
    int n;
    string s;
    cin >> n >> s;
    // dp[i][j] 表示处理到字符串第 i 个字符（从 1 开始计数）时，匹配到“ AK47”  第 j 位的方案数
    // j = 1 对应 'A', j = 2 对应 'K', j = 3 对应 '4', j = 4 对应 '7'
    for (int i = 1; i <= n; ++i) {
        // 继承前 i - 1 位的状态（不选当前字符时的方案数）
        dp[i][1] = dp[i - 1][1];
        dp[i][2] = dp[i - 1][2];
        dp[i][3] = dp[i - 1][3];
        dp[i][4] = dp[i - 1][4];
```

AK计数


```
        // 继承前 i - 1 位的状态（不选当前字符时的方案数）
        dp[i][1] = dp[i - 1][1];
        dp[i][2] = dp[i - 1][2];
        dp[i][3] = dp[i - 1][3];
        dp[i][4] = dp[i - 1][4];
        char c = s[i - 1];  // 字符串下标从 0 开始，所以用 i - 1 取字符
        if (c == 'A') {
            // 匹配到“ AK47”  的第 1 位，新增方案数
            dp[i][1] += 1;
        } else if (c == 'K') {
            // 匹配到“ AK47”  的第 2 位，需要基于第 1 位的方案数转移
            dp[i][2] += dp[i - 1][1];
        } else if (c == '4') {
            // 匹配到“ AK47”  的第 3 位，需要基于第 2 位的方案数转移
            dp[i][3] += dp[i - 1][2];
        } else if (c == '7') {
            // 匹配到“ AK47”  的第 4 位，需要基于第 3 位的方案数转移
            dp[i][4] += dp[i - 1][3];
        }
    }
    // 最终答案是匹配到“ AK47”  第 4 位的总方案数
    cout << dp[n][4] << endl;
    return 0;
}
```

02

最长括号匹配

TEACHING
COURSEWARE

TEACH

最长括号匹配

题目描述

对一个由 `(`、`)`、`[`、`]` 四种括号组成的字符串，求出其中最长的括号匹配子串。具体来说，满足如下条件的字符串成为括号匹配的字符串：

- `()`、`[]` 是括号匹配的字符串。
- 若 `A` 是括号匹配的串，则 `(A)`、`[A]` 是括号匹配的字符串。

3.若 `A`、`B` 都是括号匹配的字符串，则 `AB` 也是括号匹配的字符串。

例如：`()`、`[]`、`([])`、`()()` 都是括号匹配的字符串，而 `][`、`[(])` 则不是。

字符串 A 的子串是指由 A 中连续若干个字符组成的字符串。

例如，`A`、`B`、`C`、`ABC`、`CAB`、`ABCABC` 都是字符串 `ABCABC` 的子串，而 `D`、`BA`、`ACB` 则不是。空串是任何字符串的子串。

输入格式

输入一行，为一个仅由 `()[]` 组成的非空字符串。

最长括号匹配

输出格式

输出一行，为最长的括号匹配子串。若有相同长度的子串，输出在原字符串内出现位置靠前的子串。

输入输出样例

输入 #1	复制	输出 #1	复制
([()[()]]()		[()]	

输入 #2	复制	输出 #2	复制
())[]		()	

说明/提示

数据范围

记 n 为输入字符串的长度。

对于 20% 的数据，$n \leq 10^2$。

对于 50% 的数据，$n \leq 10^4$。

对于 100% 的数据，$n \leq 10^6$。

最长括号匹配

状态定义

dp[i] 表示以字符串第 i 个字符为结尾的最长有效括号（包括圆括号 () 和方括号 []）序列的长度。若第 i 个字符无法构成有效括号序列的结尾，则 dp[i] 保持默认值 0。

状态转移方程

当 s[i] 是右括号（) 或]）时，检查其左侧对应位置是否存在匹配的左括号：

计算可能匹配的左括号位置：pos = i - dp[i-1] - 1（跳过前 i-1 个字符中已匹配的部分）

若 pos 合法且 s[pos] 与 s[i] 是匹配的括号对（() 或 []），则：

dp[i] = dp[i-1] + 2（当前匹配的一对括号长度 + 前一段已匹配的长度）

若 pos > 0，还需加上 dp[pos-1]（合并更前面的有效括号序列）

当 s[i] 是左括号（(或 [) 时，无法作为有效序列的结尾，dp[i] 保持 0。

pos > 0 时的合并操作，是为了把当前匹配的括号序列和它前面可能存在的独立有效序列连接成更长的有效序列，避免漏掉这种连续有效的情况。

最长括号匹配


```cpp
#include<bits/stdc++.h>
using namespace std;
string s;
int l, dp[1000005], ans, id;
int main()
{
    cin >> s;
    l = s.size();
    for(int i = 1; i < l; ++i)
    {
        if(s[i] == '(' || s[i] == '[')
            continue;                    // 左括号无法作为匹配终点，跳过
```

最长括号匹配


```
        else
        {
            // 检查是否与前面对应的括号匹配
            if( (s[i] == ')' && i - dp[i-1] - 1 >= 0 && s[i - dp[i-1] - 1] == '(') ||
                (s[i] == ']' && i - dp[i-1] - 1 >= 0 && s[i - dp[i-1] - 1] == '[') )
            {
                // 状态转移：当前匹配长度 = 前一段匹配长度 + 2 + 再前一段匹配长度
                dp[i] = dp[i-1] + 2;
                if(i - dp[i-1] - 2 >= 0)
                    dp[i] += dp[i - dp[i-1] - 2];
                // 更新最长匹配长度和结束位置
                if(dp[i] > ans)
                {
                    ans = dp[i];
                    id = i;
                }
            }
        }
    }
```

最长括号匹配


```
    for(int i = id - ans + 1; i <= id; ++i)
    {
        cout << s[i];
    }
    cout << endl;

    return 0;
}
```