区间动态规划

河南省实验中学信息技术组

2025 年 07 月 04 日

【例】石子合并

有 n 堆石子排成一排，其中第 i 堆石子的重量为 a_i。每次可以选择其中相邻的两堆石子合并成一堆，形成的新石子堆的重量以及消耗的体力都是两堆石子的重量之和。求把全部 n 堆石子合成一堆最少需要消耗的体力。例如：有 3 堆石子，重量分别为 13,7,8，有两种合并方案：

- 第一种方案：先合并 1,2 号堆，合并后的新石子堆重量为 20，本次合并消耗的体力为 20，再拿新堆与第 3 堆石子合并，合并后的石子重量为 28，本次合并消耗的体力为 28，将 3 堆石子合并到一起的消耗的总体力为第一次合并消耗的体力 20 加上第二次合并消耗的体力 28，即 48；
- 第二种方案：先合并 2,3 号堆，合并后的新石子堆重量为 15，本次合并消耗的体力为 15，再拿新堆与第 1 堆石子合并，合并后的石子重量为 28，本次合并消耗的体力为 28，将 3 堆石子合并到一起的消耗的总体力为第一次合并消耗的体力 15 加上第二次合并消耗的体力 28，即 43；

采用第二种方案可取得最小的体力消耗，值为 43。

【例】石子合并

【输入格式】

第一行一个整数 $n(n \leq 100)$，表示石子的堆数。

第二行 n 个整数，表示每一堆石子的重量。

【输出格式】

一行一个整数，表示将所有石子合并成一堆最少需要消耗的体力。

【样例 1 输入】

```
5
3 7 8 2 5
```

【样例 1 输出】

```
57
```

【样例 2 输入】

```
7
13 7 8 16 21 4 18
```

【样例 2 输出】

```
239
```

【例】石子合并

- 如果 $n = 1$，即只有一堆石子时，不需要合并，所以消耗的体力为 0。
- 如果 $n = 2$，那么将两堆石子合并即可，消耗的体力为两堆石子的重量之和。
- 如果 $n = 3$，那么正如题目描述的最后只有两种方案 ((1,2),3) 和 (1,(2,3))，在这两种方案中取最优解。定义 $f(l, r)$ 表示合并第 l 堆到第 r 堆的最小体力消耗，则

$$f(1,3) = \min\{f(1,2) + f(3,3), f(1,1) + f(2,3)\} + (a_1 + a_2 + a_3)$$

- 对于任意的 n 堆，那么最后一定是由两堆合成的，则根据两堆划分的位置不同，有

$$f(1,n) = \min_{k=1}^{n-1}\{f(1,k) + f(k+1,n)\} + (a_1 + a_2 + \cdots + a_n)$$

【例】石子合并

- 为了求出 $f(1,n)$，则需要求出 $f(1,k)$ 和 $f(k+1,n)$。
- 为了求出 $f(1,k)$，则需要求出 $f(1,i)$ 和 $f(i+1,k)$。
- 为了求出 $f(k+1,n)$，则需要求出 $f(k+1,j)$ 和 $f(j+1,n)$。
- 对于这种大区间信息需要由小区间合并得到的动态规划问题，称为区间动态规划。

【例】石子合并

- 状态：定义 $f(l,r)$ 表示合并第 l 堆到第 r 堆的最小体力消耗。
- 状态转移方程：$f(l,r)=\min\limits_{k=l}^{r-1}\{f(l,k)+f(k+1,r)\}+\sum\limits_{i=l}^{r}a_i$
- 初始状态：$f(i,i)=0(1\le i\le n)$，其余为无穷大。
- 目标状态：$f(1,n)$，表示合并第 1 堆到第 n 堆的最小体力消耗。
- 阶段：区间的长度。

【例】石子合并

例：共有 5 堆石子，它们的重量依次为 3、7、8、2、5。

f	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

表: 初始状态

f	1	2	3	4	5
1	0	10			
2		0	15		
3			0	10	
4				0	7
5					0

表: 结果状态

$$f(2,3) = \min\left\{ f(2,2) + f(3,3) = 0 + 0 = 0 \right\} + (7+8) = 15$$

【例】石子合并

例：共有 5 堆石子，它们的重量依次为 3、7、8、2、5。

f	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

表: 初始状态

f	1	2	3	4	5
1	0	10	28		
2		0	15	27	
3			0	10	22
4				0	7
5					0

表: 结果状态

$$f(1,3) = \min \left\{ \begin{array}{l} f(1,1) + f(2,3) = 0 + 15 = 15 \\ f(1,2) + f(3,3) = 10 + 0 = 10 \end{array} \right\} + (3 + 7 + 8) = 28$$

【例】石子合并

例：共有 5 堆石子，它们的重量依次为 3、7、8、2、5。

f	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

表: 初始状态

f	1	2	3	4	5
1	0	10	28	40	
2		0	15	27	44
3			0	10	22
4				0	7
5					0

表: 结果状态

$$f(1,4) = \min \left\{ \begin{array}{l} f(1,1) + f(2,4) = 0 + 27 = 27 \\ f(1,2) + f(3,4) = 10 + 10 = 20 \\ f(1,3) + f(4,4) = 28 + 0 = 28 \end{array} \right\} + (3 + 7 + 8 + 2) = 40$$

【例】石子合并

例：共有 5 堆石子，它们的重量依次为 3、7、8、2、5。

f	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

表: 初始状态

f	1	2	3	4	5
1	0	10	28	40	57
2		0	15	27	44
3			0	10	22
4				0	7
5					0

表: 结果状态

$$f(1,5)=\min\left\{\begin{array}{l} f(1,1)+f(2,5)=0+44=44 \\ f(1,2)+f(3,5)=10+22=32 \\ f(1,3)+f(4,5)=28+7=35 \\ f(1,4)+f(5,5)=40+0=40 \end{array}\right\}+(3+7+8+2+5)=57$$

【例】石子合并

```
1  memset(f, 0x3f, sizeof(f));
2  for(int i = 1; i <= n; ++i) s[i] = s[i - 1] + a[i];    // 前缀和
3  for(int i = 1; i <= n; ++i) f[i][i] = 0;
4  for(int len = 2; len <= n; ++len)  // 阶段：区间长度
5      for(int l = 1; l <= n - len + 1; ++l)
6      {
7          int r = l + len - 1;        // 状态 f(l,r)
8          for(int k = l; k < r; ++k)
9              f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r]);
10         f[l][r] += s[r] - s[l - 1];  // 加上本次合并消耗的体力
11     }
12 cout << f[1][n];
```

- 时间复杂度：$O(N^3)$；空间复杂度：$O(N^2)$
- 运用四边形不等式规则可以改进状态转移方程，从而可以将时间复杂度降至 $O(N^2)$。
- 对于石子合并问题还有 $O(N \log N)$ 的算法，不过不是动态规划算法。

【例】矩阵连乘

一个 $n \times m$ 矩阵由 n 行 m 列共 $n \times m$ 个数排列而成。

两个矩阵 A 和 B 可以相乘当且仅当 A 的列数等于 B 的行数。

一个 $n \times m$ 的矩阵乘以一个 $m \times p$ 的矩阵等于一个 $n \times p$ 的矩阵，其所需要的乘法运算量为 $n \times m \times p$。矩阵乘法满足结合律，$A \times B \times C$ 可以表示成 $(A \times B) \times C$ 或者是 $A \times (B \times C)$，两者的运算量却不同。例如当 A 是一个 2×3 的矩阵，B 是一个 3×4 的矩阵，C 是一个 4×5 的矩阵时，$(A \times B) \times C$ 所需要的运算量为 64，而 $A \times (B \times C)$ 的运算量为 90，显然第一种顺序节省运算量。

现在给出 N 个矩阵，请你设计一个乘法顺序使得需要的乘法运算量最小。

【例】矩阵连乘

【输入格式】
第一行一个整数 $N(N \leq 100)$，表示矩阵的个数。
第二行 $N+1$ 个整数 A_i，用来表示 N 个矩阵的行数和列数，具体的第 i 个矩阵是一个 A_i 行 A_{i+1} 列的矩阵。
【输出格式】
最优的运算量，所有的数据均保证结果不超过 $2^{31}-1$。

【样例输入】

```
3
2 3 4 5
```

【样例输出】

```
64
```

【样例解释】
三个矩阵 A, B, C 的大小分别为 2×3、3×4、4×5。
最优的计算顺序为 $(A \times B) \times C$，计算 $A \times B$ 需要的运算量为 $2 \times 3 \times 4 = 24$，得到的矩阵 D 的大小为 2×4，计算 $D \times C$ 需要的运算量为 $2 \times 4 \times 5 = 40$，故而需要的总运算量为 64。

【例】矩阵连乘

- 定义第 i 个矩阵的大小为 $b[i] \times c[i]$。
- 定义 $f(l,r)$ 表示第 $l \sim r$ 的矩阵连乘的最小运算量。
- 考虑最后一次连乘，假设相乘的两个矩阵分别由 $l \sim k$ 和 $k+1 \sim r$ 的矩阵连乘得到，它们的大小分别为 $b[l] \times c[k]$ 和 $b[k+1] \times c[r]$ 且 $c[k] = b[k+1]$，那么

$$f(l,r) = \min_{l \le k < r} \{f[l][k] + f[k+1][r] + b[l] \times c[k] \times c[r]\}$$

```
for(int i = 1; i <= n; ++i) b[i] = a[i], c[i] = a[i + 1];
memset(f, 0x3f, sizeof(f));
for(int i = 1; i <= n; ++i) f[i][i] = 0;
for(int len = 2; len <= n; ++len)
    for(int l = 1; l <= n - len + 1; ++l)
        for(int r = l + len - 1, k = l; k < r; ++k)
            f[l][r] = min(f[l][r], f[l][k] + f[k + 1][r] + b[l] * c[k] * c[r]);
cout << f[1][n];
```

- 时间复杂度：$O(N^3)$。

【例】最长括号匹配

我们对“合法的括号序列”进行以下定义：

1. 空序列是一个合法的括号序列；
2. 如果 S 是合法的括号序列，则 (S) 和 [S] 是合法的括号序列；
3. 如果 A 和 B 是合法的括号序列，则 AB 是合法的括号序列；
4. 没有其他序列是合法的括号序列。

例如，()，[]，(())，()[]，()[()] 都是合法的，而 (，]，)(，([)]，([(则不是合法的。

现在给定一个长度为 n 的括号组成的字符序列，需要找到它的一个最长的且符合括号序列定义的子序列。形式化的描述是，需要求出最大的 m，使得对于下标 $i_1, i_2, \cdots, i_m (1 \le i_1 < i_2 < \cdots < i_m \le n)$ 的序列 $s_{i_1} s_{i_2} \cdots s_{i_m}$ 是一个合法的括号序列。

【例】最长括号匹配

【输入格式】
输入包含多组 (不超过 10 组) 数据，每行一个由 (,),[,] 组成的字符串 (长度不超过 100)。最后以 end 表示输入结束。
【输出格式】
对于每一个括号字符串，输出一行一个整数表示最长的合法括号子序列的长度。

【样例输入】

```
(((())))
()()()
([]])
)[)(
([][][)
end
```

【样例输出】

```
6
6
4
0
6
```

【例】最长括号匹配

- 定义 $f(l,r)$ 表示对于字符串 $s[l:r]$ 而言，它的最长的合法括号子序列的长度。
  1. 如果 $s[l]$ 和 $s[r]$ 满足括号匹配，那么 $f(l,r)=f(l+1,r-1)+2$。
  2. 如果不满足括号匹配，以 k 为分界，将 $s[l:r]$ 分成 $s[l:k]$ 和 $s[k+1:r]$，那么

$$f(l,r)=\max_{l\le k<r}\{f(l,k)+f(k+1,r)\}$$

在以上两种情况中取最大值。

```
memset(f, 0, sizeof(f));
for(int len = 2; len <= n; ++len)       // 阶段：区间长度
    for(int l = 1; l <= n - len + 1; ++l)  // 状态：区间左端点
    {
        int r = l + len - 1;            // 状态：区间右端点
        if(s[l] == '(' && s[r] == ')' || s[l] == '[' && s[r] == ']')
            f[l][r] = max(f[l][r], f[l + 1][r - 1] + 2);   // 情况 1
        for(int k = l; k < r; ++k)      // 情况 2
            f[l][r] = max(f[l][r], f[l][k] + f[k + 1][r]);
    }
cout << f[1][n] << "\n";
```

- 时间复杂度：$O(N^3)$。

【例】最便宜的回文

给定一个仅有小写字母构成的字符串，请你通过在字符串任意位置增加或删除小写字母使其变为回文。

例如，对于字符串 abcb，可以通过在末尾增加 a 使其变为 abcba、或者删除字符 a，使其变为bcb、或者在开头增加 bcb，使其变为bcbabcb。

但是现在增加、删除每个字符的成本可能是不同的，现在请你求解将给定字符串转化为回文的最小成本。

【例】最便宜的回文

【输入格式】
第 1 行包含两个整数 $n, m(1 \leq n \leq 26, 1 \leq m \leq 2000)$，分别表示给定成本的字符个数和字符串长度。
第 2 航给定一个字符串。
接下来 n 行，每行包含一个字符和两个整数，分别表示增加和删除该字符的成本 $(0 \sim 10^4)$。
【输出格式】
一行一个整数，表示将原字符串变为回文的最小成本。

【样例输入】

```
3 4
abcb
a 1000 1100
b 350 700
c 200 800
```

【样例输出】

```
900
```

【例】最便宜的回文

- 定义 $f(l,r)$ 表示字符串 $s[l:r]$ 转化为回文的最小成本。
- 如果 $s[l]=s[r]$，则两端的字符不需要花费成本，那么有

$$f(l,r)=f(l+1,r-1)$$

- 如果 $s[l]\neq s[r]$，那么有四种情况：
  1. 删除左侧字符 $s[l]$；
  2. 在最右侧添加字符 $s[l]$；
  3. 删除右侧字符 $s[r]$；
  4. 在最左侧添加字符 $s[r]$。

  那么有

$$f(l,r)=\min\{f(l+1,r)+w_1,f(l+1,r)+w_2,f(l,r-1)+w_3,f(l,r-1)+w_4\}$$

【例】最便宜的回文

```
for(int len = 2; len <= m; ++len)
    for(int l = 1; l <= m - len + 1; ++l)
    {
        int r = l + len - 1;
        if(s[l] == s[r]) f[l][r] = f[l + 1][r - 1];
        else
        {
            f[l][r] = min({f[l + 1][r] + a[s[l] - 'a'],
                f[l + 1][r] + b[s[l] - 'a'],
                f[l][r - 1] + a[s[r] - 'a'],
                f[l][r - 1] + b[s[r] - 'a']});
        }
    }
printf("%d", f[1][m]);
```

- 时间复杂度：$O(N^2)$。

【例】添加号

有一个由数字 $1,2,\cdots,9$ 组成的数字串 (长度不超过 16)，问如何将 m 个加号“+”插入到这个数字串中，使所形成的算术表达式的值最小。

注意：加号不能加在数字串的最前面或最末尾，也不应有两个或两个以上的加号相邻，m 保证小于数字串的长度。

例如：数字串 79846，若需要加入两个加号，则最佳方案为 $79+8+46$，算术表达式的值 133。

【例】添加号

【输入格式】
第一行一个数字串，数字串由数字 $1, 2, \cdots, 9$ 组成且中间无空格。
第二行一个正整数 m，表示需要添加的加号数。
【输出格式】
一行一个整数，表示最小和。

【样例 1 输入】

```
79846
2
```

【样例 1 输出】

```
133
```

【样例 2 输入】

```
82363983742
3
```

【样例 2 输出】

```
2170
```

【例】添加号

- 在数字串中间添加一个加号可以将数字串分成两部分，然后分别在这两部分再添加号。
- 定义状态 $f(l,r,k)$ 表示数字串 $s[l:r]$ 中间添加 k 个加号后表达式的最小值。
- 考虑在 $s[l:r]$ 的位置 x 后添加一个加号使得数字串被分为 $s[l:x]$ 和 $s[x+1,r]$，那么还剩余 $k-1$ 个加号需要添加，枚举 $s[l:x]$ 中添加的加号数为 t，那么 $s[x+1,r]$ 中添加的加号数为 $k-1-t$，于是状态转移方程为：

$$f(l,r,k)=\min\{f(l,x,t)+f(x+1,r,k-1-t)\}$$

其中，$l\le x\le r-1, 0\le t\le x-l, 0\le k-1-t\le r-(x+1)$。

- 初始状态：$f(l,r,0)$ 为相应子串形成的数字，其余均为无穷大。
- 目标状态：$f(1,n,m)$。

【例】添加号

```
for(int i = 1; i <= n; ++i)   // 初始状态
{
  f[i][i][0] = s[i] - '0';
  for(int j = i + 1; j <= n; ++j) f[i][j][0] = f[i][j - 1][0] * 10 + s[j] - '0';
}
for(int len = 2; len <= n; ++len)                  // 阶段：区间长度
  for(int l = 1; l <= n - len + 1; ++l)            // 状态：区间左端点
  {
    int r = l + len - 1;                           // 状态：区间右端点
    for(int k = 1; k <= len - 1 && k <= m; ++k) // 状态：区间内加号个数
      for(int x = l; x < r; ++x)                   // 决策：枚举划分位置
        for(int t = 0; t <= x - l && t <= k - 1; ++t) // 决策：枚举左边区间加号数
          if((k - 1) - t <= r - (x + 1))           // 右区间加号个数应小于区间长度
            f[l][r][k] = min(f[l][r][k], f[l][x][t] + f[x + 1][r][k - 1 - t]);
  }
cout << f[1][n][m];
```

- **时间复杂度**：$O(N^3M^2)$。

【例】添加号

- 同时考虑两个区间以及区间内的加号个数使得枚举的决策过多，可以考虑第一个或最后一个加号的位置。
- 考虑最后一个加号的位置为 x，那么需要在 $s[l:x]$ 中添加 $k-1$ 个加号，这个加号后面的子串构成一个数字，状态转移方程为：

$$f(l,r,k)=\min\{f(l,x,k-1)+num(x+1,r)\}$$

其中，$l+k-1\le x\le r-1, num(x+1,r)=f(x+1,r,0)$ 为 $s[x+1:r]$ 构成的数字。

- 初始状态：$f(l,r,0)$ 为相应子串形成的数字，其余均为无穷大。
- 目标状态：$f(1,n,m)$。

【例】添加号

```
1  memset(f, 0x3f, sizeof(f));
2  for(int i = 1; i <= n; ++i)   // 初始状态
3  {
4    f[i][i][0] = s[i] - '0';
5    for(int j = i + 1; j <= n; ++j) f[i][j][0] = f[i][j - 1][0] * 10 + s[j] - '0';
6  }
7  for(int len = 2; len <= n; ++len)          // 阶段：区间长度
8    for(int l = 1; l <= n - len + 1; ++l)    // 状态：区间左端点
9    {
10     int r = l + len - 1;                   // 状态：区间右端点
11     for(int k = 1; k <= len - 1; ++k)      // 状态：区间内加号个数
12       for(int x = l + k - 1; x < r; ++x)   // 决策：枚举划分位置
13         f[l][r][k] = min(f[l][r][k], f[l][x][k - 1] + f[x + 1][r][0]);
14   }
15 cout << f[1][n][m];
```

- **时间复杂度**：$O(N^3M)$。

【例】添加号

- 通过观察可以发现

$$f(1,n,m)=\min\{f(1,x,m-1)+num(x+1,n)\}$$

$$f(1,x,m-1)=\min\{f(1,y,m-2)+num(y+1,x)\}$$

那么可以从 $f(1,1,k)\to f(1,2,k)\to\cdots\to f(1,n,k)$，那么状态的第一维可以省略，定义 $f(r,k)$ 表示区间 $s[1:r]$ 加入 k 个加号后表达式的最小值。

- 状态转移方程可以改为：

$$f(r,k)=f(x,k-1)+num(x+1,r)$$

其中，$1\le x<r, 0\le k-1\le x-1, 0\le k\le m, num(x+1,r)$ 为 $s[x+1:r]$ 生成的数字。

- 初始状态：$f(r,0)$ 为相应子串 $s[1:r]$ 形成的数字，其余均为无穷大。
- 目标状态：$f(n,m)$。

【例】添加号

例：在数字串 79846 插入 2 个加号。

num	1	2	3	4	5
1	7	79	798	7984	79846
2		9	98	984	9846
3			8	84	846
4				4	46
5					6

表：$num(i,j)$ 为 $s[i:j]$ 生成的数字

f	0	1	2
1	7		
2	79	16	
3	798	87	24
4	7984	163	91
5	79846	844	133

表：状态结果

$$f(5,2) = \min \left\{ \begin{array}{l} f(4,1) + num(5,5) = 163 + 6 = 169 \\ f(3,1) + num(4,5) = 87 + 46 = 133 \\ f(2,1) + num(3,5) = 16 + 846 = 862 \end{array} \right\} = 133$$

【例】添加号

```
memset(f, 0x3f, sizeof(f));
for(int i = 1; i <= n; ++i)  f[i][0] = num[1][i];  // 初始状态
for(int r = 1; r <= n; ++r)   // 阶段：左边区间 [1:r] 的大小
    for(int k = 1; k < r; ++k)  // 状态：插入加号数目不能多于数字个数
        for(int x = k; x < r; ++x) // 决策：最后一个加号插入位置
            f[r][k] = min(f[r][k], f[x][k - 1] + num[x + 1][r]);
cout << f[n][m];
```

- 时间复杂度：$\mathrm{O}(N^2M)$。

【例】多边形

多边形游戏是一款单人益智游戏。在游戏开始时，系统给定玩家一个有 n 个顶点的多边形，其包含 n 个顶点和 n 条边，每条边连接两个相邻的顶点。在每个顶点上写有一个整数，可正可负。在每条边上标有一个算符“+”(加号) 或“$*$”(乘号)。

第一步，玩家需要选择一条边，将它删除。接下来 $n-1$ 步，在每一步中，玩家选择一条边，把这条边以及连接的两个顶点用一个新的顶点代替，新顶点上的整数值等于删去的两个顶点上的数按照删去的边上标有的符号进行计算得到的结果。最终，游戏只仅剩一个顶点，顶点上的数值就是玩家的得分。

请你计算对于任意的有 n 个顶点的多边形，玩家最高能获得多少分，以及第一步有哪些策略可以使玩家获得最高分。保证 $1 \leq n \leq 50$ 且无论玩家如何操作，顶点上的数值均在 $[-32768, 32767]$ 之间。

【例】多边形

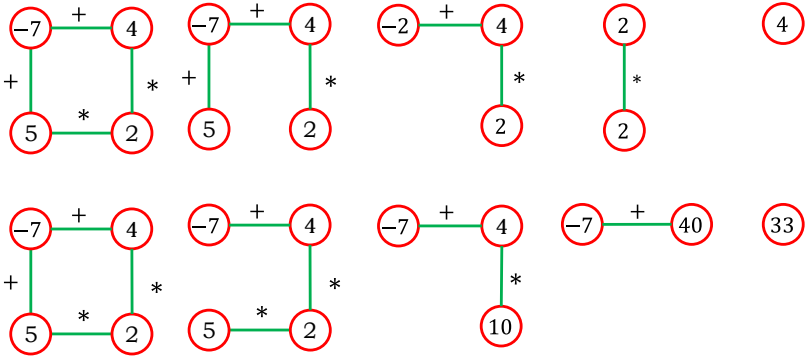


图: 两次游戏，玩家得分分别为 4 和 33

【例】多边形

【输入格式】
第一行一个整数 $n(n \leq 50)$，表示多边形的顶点数。
第二行用来描述多边形所有边上的符号以及所有顶点上的整数，从编号为 1 的边开始，按照边、点、边、… 的顺序描述。其中描述顶点即为输入顶点上的整数，描述边即为输入边上的符号。
【输出格式】
第一行一个整数，表示玩家能获得的最高分。
第二行列举出所有使得玩家能获得最高分时删除的第一条边，要求按照从小到大的顺序输出。

【样例输入】

```
4
+ -7 + 4 * 2 * 5
```

【样例输出】

```
33
1 2
```

【样例解释】
玩家能获得的最高分是 33，第一次删除第 1 或 2 条边，玩家均可获得最高分。

【例】多边形

- 首先需要枚举删除的边，那么多边形即断为一条链，为了处理方便，将断开后形成的链按照顺序整理为顶点数字在 $1 \sim n$，运算符在 $2 \sim n$ 的形式。
- 通过观察，任何一种玩法的最后一步操作的结果是基于符号两边的链区间的计算结果 (类似石子合并)。
- 定义 $f(l,r)$ 表示把第 l 到 r 个顶点合并成一个顶点后，顶点上的数值最大是多少。
- 此时，区间 $[l,r]$ 的值能否通过区间 $[l,k]$ 和区间 $[k+1,r]$ 合并求得呢?
  - 如果符号是“$+$”(加号)，那么显然是没有问题的。
  - 如果符号是“$*$”(乘号)，那么如果出现区间 $[l,k]$ 和区间 $[k+1,r]$ 的某一种方案 (并非最优) 的值为负数，那么二者的乘积结果更大，不满足最优子结构性质。
- 因此，最大值可能是由最小值合并得到的，于是定义 $d(l,r)$ 表示把第 l 到 r 个顶点合并成一个顶点后，顶点上的数值最小是多少。

【例】多边形

- 考虑状态转移方程：
  - 区间 $[l,r]$ 的最大值只可能是两个区间最大值相加、相乘，或两个最小值相乘，或一个最大值和最小值相乘。

$$f(l,r)=\max_{l\le k<r}\begin{cases}f(l,k)+f(k+1,r) & \text{op}\in\{+\}\\ \max\begin{cases}f(l,k)*f(k+1,r)\\ d(l,k)*d(k+1,r)\\ f(l,k)*d(k+1,r)\\ d(l,k)*f(k+1,r)\end{cases} & \text{op}\in\{*\}\end{cases}$$

  - 区间 $[l,r]$ 的最小值只可能是两个区间最小值相加、相乘，或两个最大值相乘，或一个最大值和最小值相乘。

$$d(l,r)=\min_{l\le k<r}\begin{cases}d(l,k)+d(k+1,r) & \text{op}\in\{+\}\\ \min\begin{cases}d(l,k)*d(k+1,r)\\ f(l,k)*f(k+1,r)\\ f(l,k)*d(k+1,r)\\ d(l,k)*f(k+1,r)\end{cases} & \text{op}\in\{*\}\end{cases}$$

【例】多边形

- 初始状态：$f(i,i)=d(i,i)=a_i$(注意此时的 a_i 不是原始输入的值，而是当前形成的链的第 i 个顶点的值)，其余的 $f(l,r)=-\infty, d(l,r)=\infty$。
- 和石子合并类似，以区间的长度作为阶段，枚举区间的中间点作为决策。

```
1 for(int i = 1; i <= n; ++i)   // 枚举删除的边
2 {
3     memset(f, 0xc0, sizeof(f));
4     memset(d, 0x3f, sizeof(d));
5     for(int j = i; j <= n; ++j) ta[j - i + 1] = a[j];       //整理链上的数字
6     for(int j = 1; j < i; ++j) ta[n - i + 1 + j] = a[j];
7     for(int j = i + 1; j <= n; ++j) tp[j - i + 1] = op[j]; //整理链上的运算符
8     for(int j = 1; j < i; ++j) tp[n - i + 1 + j] = op[j];
9     for(int j = 1; j <= n; ++j) f[j][j] = d[j][j] = ta[j];  // 初始状态
```

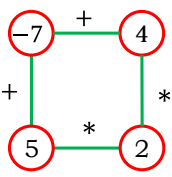
【例】多边形

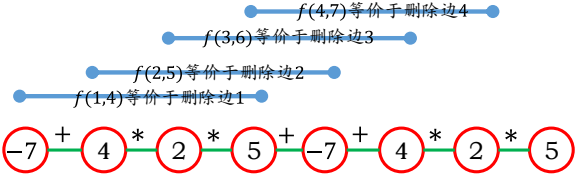
```
1     for(int len = 2; len <= n; ++len)   // 阶段
2         for(int l = 1; l <= n - len + 1; ++l) { // 状态
3             int r = l + len - 1;                // 状态
4             for(int k = l; k < r; ++k) {  // 决策
5                 if(tp[k + 1] == '+') {
6                     f[l][r] = max(f[l][r], f[l][k] + f[k + 1][r]);
7                     d[l][r] = min(d[l][r], d[l][k] + d[k + 1][r]);
8                 } else if(tp[k + 1] == '*') {
9                     f[l][r] = max(f[l][r], f[l][k] * f[k + 1][r]);
10                    f[l][r] = max(f[l][r], d[l][k] * d[k + 1][r]);
11                    f[l][r] = max(f[l][r], f[l][k] * d[k + 1][r]);
12                    f[l][r] = max(f[l][r], d[l][k] * f[k + 1][r]);
13                    d[l][r] = min(d[l][r], d[l][k] * d[k + 1][r]);
14                    d[l][r] = min(d[l][r], f[l][k] * f[k + 1][r]);
15                    d[l][r] = min(d[l][r], f[l][k] * d[k + 1][r]);
16                    d[l][r] = min(d[l][r], d[l][k] * f[k + 1][r]);
17                }
18            }
19        }
20    r[i] = f[1][n]; // 断开第 i 条边获得的最高分
21}
```

- 枚举删除边的时间复杂度为 $O(N)$，求解链的最高分的时间复杂度为 $O(N^3)$，故而整体时间复杂度为 $O(N^4)$。

【例】多边形

- 优化：任选一条边 (例如第 1 条)，然后把剩下的链复制一倍接在末尾[1]。





- 在这个长度为 $2n$ 的链上，所有长度为 n 的区间 $[i, i+n-1]$ 合并成一个顶点，等价于原问题删除一条边，将形成的链合并成一个顶点。
- 于是问题变为了一个长度为 $2n$ 的链的问题，而且只需要处理前 n 个阶段即可，答案显然为 $\max\limits_{1\le i\le n} f(i, i+n-1)$，时间复杂度为 $\mathrm{O}(N^3)$。

[1]破坏为链：任意选择一个位置断开，复制形成 2 倍长度的链。这种方法是解决 DP 中的环形结构的常用方法。

【例】多边形

```
1  memset(f, 0xc0, sizeof(f));
2  memset(d, 0x3f, sizeof(d));
3  for(int i = 1; i <= n; ++i)
4      f[i + n][i + n] = d[i + n][i + n] = d[i][i] = f[i][i] = a[i];
5  for(int len = 2; len <= n; ++len)   // 阶段
6      for(int l = 1; l <= 2 * n - len + 1; ++l)   // 状态
7      {
8          int r = l + len - 1;          // 状态
9          for(int k = l; k < r; ++k)  // 决策
10         {
11             // 状态转移不变
12         }
13     }
14 int ans = -1e9;
15 for(int i = 1; i <= n; ++i) ans = max(ans, f[i][i + n - 1]);
```

【例】金字塔

虽然探索金字塔是极其老套的剧情，但是有一队探险家还是到了某金字塔脚下。经过多年的研究，科学家对这座金字塔的内部结构已经有所了解。首先，金字塔由若干房间组成，房间之间连有通道。如果把房间看作节点，通道看作边的话，整个金字塔呈现一个有根树结构，节点的子树之间有序，金字塔有唯一的一个入口通向树根。并且，每个房间的墙壁都涂有若干种颜色的一种。

探险队员打算进一步了解金字塔的结构，为此，他们使用了一种特殊设计的机器人。这种机器人会从入口进入金字塔，之后对金字塔进行深度优先遍历。机器人每进入一个房间（无论是第一次进入还是返回），都会记录这个房间的颜色。最后，机器人会从入口退出金字塔。

显然，机器人会访问每个房间至少一次，并且穿越每条通道恰好两次（两个方向各一次），然后，机器人会得到一个颜色序列。但是，探险队员发现这个颜色序列并不能唯一确定金字塔的结构。现在他们想请你帮助他们计算，对于一个给定的颜色序列，有多少种可能的结构会得到这个序列。因为结果可能会非常大，你只需要输出答案对 10^9 取模之后的值。

【例】多边形

【输入格式】
一行一个字符串 S，长度不超过 300，表示机器人得到的颜色序列。
【输出格式】
一行一个整数，表示答案。

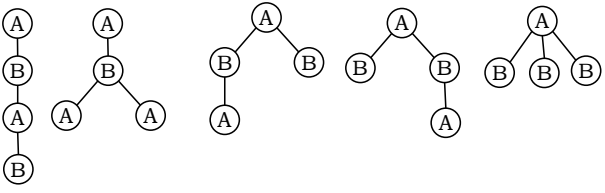
【样例输入】

```
ABABABA
```

【样例输出】

```
5
```

【样例解释】
注意：子树是有序的。

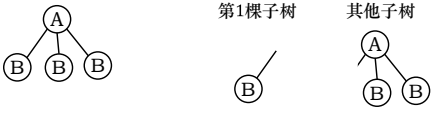


【例】金字塔

- 题目规定的访问顺序和 DFS 序有一定区别，但是和 DFS 序一样，任意一个子树也和区间等价。
- 定义 $f(l,r)$ 表示字符串 $s[l:r]$ 对应多少种金字塔结构。
- 首先，显然如果 $s[l:r]$ 表示一棵子树，那么 $s[l]$ 和 $s[r]$ 是必然是进入和离开树根时产生的。
- 那么对于它的每一颗子树，产生 $s[l+1:r-1]$ 中的某一段，同时相邻两个子树中间还有一个途径根结点产生的一个字符。
- 方案 1：枚举途径根结点产生的字符，这样时间复杂度较高。
- 方案 2(错误)：将 $s[l+1][r-1]$ 分成两个部分，每部分由若干个子树构成，但是这样可能会重复计数。例如，划分方案 A|BAB|A|B|A 和 A|B|A|BAB|A 最终产生的都是样例中的方案 5。

【例】金字塔

- 考虑枚举第一棵子树时由那一段构成的，枚举划分点 k，令字符串 $s[l+1:k-1]$ 构成第一棵子树，$s[k:r]$ 构成剩余部分的树。



- 如果 k 不相同，那么字符串 $s[l+1:k-1]$ 代表的子树必然不相同，不可能会重复。于是，有如下状态转移方程：
  - 如果 $s[l] \neq s[r]$，$f(l,r)=0$。
  - 如果 $s[l]=s[r]$，

$$f(l,r)=\sum f(l+1,k-1)\times f(k,r)$$

    ，其中 $i+2\leq k\leq r$ 且 $s[l]=s[k]$。
- 显然有 $f(i,i)=1$，其余均为 0。

【例】金字塔


```
1  for(int i = 1; i <= n; ++i) f[i][i] = 1;
2  for(int len = 2; len <= n; ++len)
3      for(int l = 1; l <= n - len + 1; ++l)
4      {
5          int r = l + len - 1;
6          if(s[l] != s[r]) continue;
7          for(int k = l + 2; k <= r; ++k)
8          {
9              if(s[k] != s[l]) continue;
10             f[l][r] += f[l + 1][k - 1] * f[k][r] % M;
11             f[l][r] %= M;
12         }
13     }
14 printf("%lld", f[1][n]);
```

- 时间复杂度：$\mathrm{O}(N^3)$。

小结

- 区间动态规划的典型特点就是区间合并。
- 状态通常是区间的范围，例如 $f(l,r)$ 表示合并第 l 堆到第 r 堆的最小体力消耗。
- 通常大区间的结果由小区间合并得到，例如 $f(l,r)$ 可以由 $f(l,k)$ 和 $f(k+1,r)$ 合并得出，因此区间动态规划的决策就是划分区间的方法。
- 区间动态规划的阶段通常是区间的长度。
- 区间动态规划的初始状态通常是长度为 1 的区间 (元区间)。
- 必要时，需要通过增加维度或增加状态来维护区间信息以满足最优子结构性质。

练习

- 石子合并【SYOI】(COGS 80)
- 最优矩阵连乘 (COGS 1559)
- 矩阵连乘 (COGS 1135)
- 最长括号匹配【SYOI】(COGS 4030)
- 最便宜的回文 (COGS 2424)
- 添加号 (COGS 1245)
- 添加号【NOI 1996】(COGS 124)
- 乘积最大【NOIP 2000PJ】【SYOI】(COGS 87)
- 多边形【IOI 1998】【SYOI】(COGS 1503)
- 能量项链【NOIP 2006】【SYOI】(COGS 116)
- 金字塔 (COGS 3162)

练习

- 加分二叉树【NOIP 2003】(COGS 106)
- 括号序列【CSP 2021】(COGS 3620)
- 折叠序列 (COGS 2996)
- 棋盘分割【NOI 1999】(COGS 100)
- 方块消除 (COGS 2034)
- 二叉查找树【NOI 2009】(COGS 408)