区间与环形 DP

按区间长度阶段枚举与“断环成链”$2n$ 窗口技巧

HGNU-ACM 实验室算法组

配套：区间斜线填表 + 环形 $2n$ 长链 + 《石子合并》《能量项链》《括号配对》全拆解





本章目录

区间 DP 核心：为什么按长度 len 枚举

环形 DP：断环成链与 $2n$ 窗口技巧

例题精讲（一）：石子合并与能量项链

承接第 1 章《线性 DP 与背包模型》的“按位置/容量递推”思想，本章把状态升级为二维连续区间 `dp[l][r]`，先学按长度 `len` 阶段填表，再学断环成链。

本章目录（续）

例题精讲（二）：括号配对与模型对比

解题五步法、常见错误与练习安排

第 3、4 节逐页精讲《石子合并》《能量项链》《括号配对》三道真题，第 5 节总结模型对比、高频错因与第 3 章预告。

先手算一个小例子：合并 3 堆石子 $[4, 5, 9]$

游戏规则

桌面上排成一排 3 堆石子，数量分别是 $4, 5, 9$。每次只能把相邻的两堆合并成一堆，花费的体力等于新堆的石子总数。怎样合并总花费最小？

方案	第 1 步合并	第 2 步合并	总花费
方案 A（先合前两堆）	$4+5=9$（花 9），剩 $[9, 9]$	$9+9=18$（花 18）	$9+18=$ **27**
方案 B（先合后两堆）	$5+9=14$（花 14），剩 $[4, 14]$	$4+14=18$（花 18）	$14+18=32$

- 无论怎么合，最后一步的代价都是全部总和 $4+5+9=18$；差别在于前面子区间的代价不同！

本页要点

倒着看合并过程：把整段 $[l, r]$ 合成一堆前，最后一步一定是合并相邻的左半段 $[l, k]$ 与右半段 $[k+1, r]$！

承接第 1 章：为什么线性 dp[i] 装不下这个问题？

- 在第 1 章《线性 DP》中，我们只用一维状态 dp[i] 表示“前 i 个元素的最优解”，因为每次决策只接在末尾。
- 但在“相邻合并”问题里，我们可以先合并中间的 $[2,3]$，再和左边的 1 或右边的 4 合并——子问题不再只是前缀 $[1,i]$，而是任意一段连续区间 $[l,r]$！
- 新状态定义：用二维数组 dp[l][r] 表示“把第 l 堆到第 r 堆这一整段连续区间合并成 1 堆的最小花费”。
- 新术语解释（无后效性）：只要知道 $[l,k]$ 合并的最小花费，不需要记住它内部先合哪两堆，因为最终重量永远是这段石子的总和！

本页要点

当决策可以在序列中间任意相邻处合并时，状态必须从前缀 dp[i] 升级为区间 dp[l][r]。

为什么需要区间 DP？暴力枚举合并顺序会 TLE

暴力想法与操作次数估算

如果有 n 堆石子，每次挑相邻两堆合并，第 1 步有 $n-1$ 种选法，第 2 步有 $n-2$ 种……粗略估算不同合并顺序的方案数高达 $(n-1)!$（去重后也有卡特兰数 $C_{n-1} \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$）。

规模 n	暴力搜索方案数（约）	能否在 1 秒内跑完？
$n = 5$	14 次	瞬间出解
$n = 20$	$> 1.7 \times 10^9$ 次	已经超时（**TLE**）
$n = 200$（题目上限）	$> 10^{116}$ 次	宇宙毁灭也跑不完！

- 暴力之所以慢，是因为同一段区间 $[l, r]$ 的最优合法被重复算了无数遍；把结果存进 `dp[l][r]` 就能避免重复计算。

本页要点

区间 DP 把指数级的 $(n-1)!$ 合并顺序压缩成 $O(n^2)$ 个区间状态，总时间降到 $O(n^3)$！

核心冲突：为什么不能像第 1 章那样按 l=1..n 顺序枚举？

假设我们照搬线性 DP 的习惯，外层写 `for (int l = 1; l <= n; ++l)`，内层写 `for (int r = l; r <= n; ++r)`，看看算 `dp[1][4]` 时会发生什么：

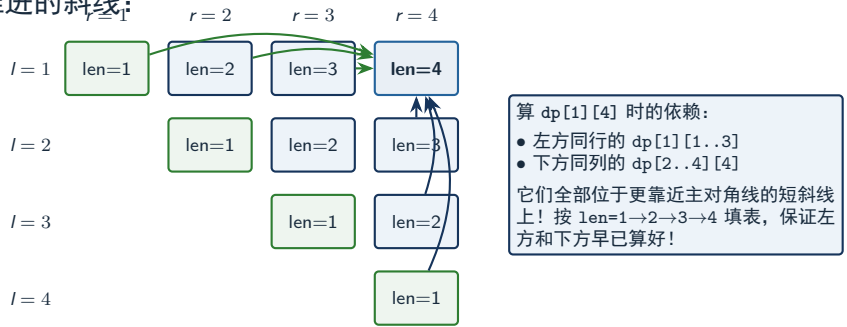
- 要算 `dp[1][4]`，需要尝试 3 个分界点 $k \in \{1, 2, 3\}$：
  1. $k = 1$：需要 `dp[1][1]` 和 `dp[2][4]`；
  2. $k = 2$：需要 `dp[1][2]` 和 `dp[3][4]`；
  3. $k = 3$：需要 `dp[1][3]` 和 `dp[4][4]`。
- 灾难来了！当外层刚走到 $l = 1$ 时，$l = 2, 3, 4$ 的循环都还没开始，`dp[2][4]`，`dp[3][4]`，`dp[4][4]` 全是初始垃圾值 0 或 ∞！
- 这叫“依赖的子问题还没算完”——大区间 $[l, r]$ 依赖的是比它更短的区间，而不是左端点更小的区间！

本页要点

区间 DP 的“阶段”不是左端点 l，而是区间长度 `len = r - l + 1`！必须先算完所有短区间，再算长区间！

TikZ 图解：dp[l][r] 二维表的“斜线填表”顺序

把 l 当行、r 当列（只填右上三角 $l \le r$），按长度 len = 1，2，3，4 就是一条条往右上角推进的斜线：



本页要点

斜线方向从主对角线（len=1）逐层推向右上角（len=n），算每个格子时它左边和下边都已就绪！

4×4 手算跟踪表：线性石子 $[4,5,9,4]$ 的斜线填表过程

前缀和 $s=[0,4,9,18,22]$（区间 $[l,r]$ 总和为 $s[r]-s[l-1]$），求最小合并花费：

阶段 len	区间 $[l,r]$	区间总和 cost	枚举分界点 k 计算	dp[l][r]
1	$[1,1],[2,2],[3,3],[4,4]$	——	只有 1 堆不用合并	均为 0
2	$[1,2]$ / $[2,3]$ / $[3,4]$	9 / 14 / 13	$0+0+\text{cost}$	9 / 14 / 13
3	$[1,3]$	18	$k=1:0+14+18=32$；$k=2:9+0+18=27$	**27**
3	$[2,4]$	18	$k=2:0+13+18=31$；$k=3:14+0+18=32$	**31**
4（终止）	$[1,4]$	22	$k=1:0+31+22=53$；$k=2:9+13+22=44$；$k=3:27+0+22=49$	**44**

- 终止步 $len=4$ 算出线性 $[1,4]$ 最小花费是 44（分界点 $k=2$，即先合 $[4,5]$ 和 $[9,4]$ 再合一起）。

本页要点

只有按 `len=1,2,3,4` 填表，算 $len=4$ 的 $[1,4]$ 时，所需的 $[1,3]=27$ 和 $[2,4]=31$ 才刚在 $len=3$ 那一轮算出来！

模板代码：线性区间 DP 标准三层循环

```
// 下标从 1 到 n，前缀和 s[i] = a[1] + ... + a[i]
vector<vector<long long>> dp(n + 1, vector<long long>(n + 1, 0));
for (int len = 2; len <= n; ++len) {            // 第一层：必须从短到长枚举长度！
    for (int l = 1; l + len - 1 <= n; ++l) {    // 第二层：枚举合法左端点 l
        int r = l + len - 1;                    // 由 l 和 len 直接算出右端点 r
        dp[l][r] = LLONG_MAX / 4;               // 求最小值必须设防溢出无穷大
        long long cost = s[r] - s[l - 1];       // O(1) 算出合并 [l, r] 的代价
        for (int k = l; k < r; ++k) {           // 第三层：枚举分割点 k in [l, r-1]
            dp[l][r] = min(dp[l][r], dp[l][k] + dp[k + 1][r] + cost);
        }
    }
}
cout << dp[1][n] << '\n';                       // 最终输出全区间 [1, n]
```

本页要点

口诀：外层 len、中层 l 算 r、内层 k 切分；求最小值必用 LLONG_MAX / 4！

为什么这样写？初值、前缀和与复杂度拆解

- 为什么 `len=1` 不用进循环？初始化二维数组全为 0，已经自动让所有 `dp[i][i] = 0`（单堆石子无需合并，花费为 0）。
- 为什么设 `LLONG_MAX / 4` 而不是 `LLONG_MAX`？状态转移里要把 `dp[l][k] + dp[k+1][r]` 相加！如果用 `LLONG_MAX`，两个极大值相加立刻溢出变成巨大的负数，被 `min` 当成“最优解”导致 **WA**！
- 为什么 `cost` 提到第三层循环外面？不管怎么切分 k，把 $[l,k]$ 和 $[k+1,r]$ 合并成 $[l,r]$ 的总重量永远是 $s[r]-s[l-1]$，提出来减少重复计算。
- 时间复杂度是多少？三层循环分别约 n 次，总次数 $\approx \frac{n^3}{6}$；当 $n=200$ 时仅约 1.3×10^6 次运算，不到 0.01 秒即可通过。

本页要点

把无关切分点 k 的 `cost` 提到内层循环外，且用 `LLONG_MAX / 4` 杜绝整数溢出！

为什么对？用循环不变量验证区间 DP 正确性

不变量阶段	在区间 DP 中成立的理由
初始化	在 `len = 2` 循环开始前，所有长度 1 的区间 `dp[i][i] = 0`，满足“单堆合并花费为 0”
保持	当计算长度为 `len` 的任意区间 $[l, r]$ 时，它的所有真子区间 $[l, k]$（长 $k - l + 1 <$ `len`）和 $[k+1, r]$（长 $r - k <$ `len`）均已在前面几轮算出最优解；穷举所有 $k \in [l, r-1]$ 取最小值，必得 $[l, r]$ 的全局最优解
终止	当 `len = n` 结束时，整个区间 $[1, n]$ 已被覆盖，`dp[1][n]` 即全部合法合并方案的最小值

- 这也反向印证了：只要保证“子问题长度严格小于当前长度”，不变量就牢不可破！

本页要点

长度递增保证了“短区间先于长区间算完”，这是区间 DP 正确性的唯一基石。

上机验证（1/2）：完整可运行短程序

把下面短程序保存运行（只需标准输入 n 和 n 个数），用于验证本节核心逻辑：

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; cin >> n;
    vector<long long> s(n + 1, 0);
    for (int i = 1, x; i <= n; ++i) { cin >> x; s[i] = s[i - 1] + x; }
    vector<vector<long long>> dp(n + 1, vector<long long>(n + 1, 0));
    for (int len = 2; len <= n; ++len)
        for (int l = 1; l + len - 1 <= n; ++l) {
            int r = l + len - 1; dp[l][r] = LLONG_MAX / 4;
            for (int k = l; k < r; ++k) dp[l][r] = min(dp[l][r], dp[l][k] + dp[k +
    1][r] + s[r] - s[l - 1]);
        }
    cout << dp[1][n] << '\n';
}
```

上机验证（2/2）：3 组数据实测与反事实追问

用上一页短程序测试以下 3 组线性石子数据：

测试类型	输入（n 与石子序列）	预期输出 dp[1][n]
普通 4 堆	$n=4$，序列 4 5 9 4	44（对应手算跟踪表结果）
最小边界 $n=1$	$n=1$，序列 10	0（len=2 循环直接跳过，输出 dp[1][1]=0）
全相等 4 堆	$n=4$，序列 3 3 3 3	24（两两先合 $(6+6)+(12)=24$）

反事实追问（面试/调试必考）

1. 如果把前两层循环改成 for (int l = 1; l <= n; ++l) for (int r = l + 1; r <= n; ++r)，输入 4 5 9 4 会输出什么？（算 dp[1][4] 时 dp[2][4]，dp[3][4] 还是 0，结果变成错误偏小的 31 导致 **WA**！）
2. 如果把 k < r 错写成 k <= r 会怎样？（会读取 dp[r+1][r]，当 $r=n$ 时直接数组越界 **RE**！）

对比：如果外层倒着枚举 l = n .. 1、内层 r = l+1 .. n 行不行？

- 有同学发现：如果我外层让 l 从 n 倒着枚举到 1，内层 r 从 $l+1$ 枚举到 n，算 dp[l][r] 时 dp[k+1][r]（$k+1>l$）不也已经算完了吗？
- 结论：在线性区间 DP 里，这种“左下往右上”倒序确实也能算对；但依然强烈建议统一写“外层 len”，理由有三：
  1. 阶段直观：len 直接对应“几堆合并”，思考问题和手算跟踪表时永远先想小块再想大块；
  2. 环形友好：断环成链到 $2n$ 后，按 len <= n 可以精准停在长度 n，不白算长度为 $n+1\ldots2n$ 的无用格子；
  3. 通用性强：几乎所有题解和竞赛模板都以 len 为第一维，统一写法不容易看错或写错！

本页要点

虽然线性表倒序 l 也能过，但只有“外层按 len 递增”能无缝衔接 $2n$ 断环成链且不浪费算力！

易错对照与适用条件

错误类型	错误写法	正确写法与后果
循环顺序颠倒	外层 `l`，内层 `r`	必须外层 `len`；否则读未计算右子区间（**WA**）
无穷大直接用极限值	`dp[l][r] = LLONG_MAX`	必须写 `LLONG_MAX / 4`；否则两项相加溢出成负数（**WA**）
切分点越界	`for (int k = l; k <= r; ++k)`	必须写 `k < r`；否则右侧为空区间或下标越界（**RE/WA**）

什么时候能用区间 DP？什么时候换相邻算法？

- 适用：操作限制在相邻元素或连续子段上合并/匹配，且总代价只与区间整体属性有关。
- 不适用：如果题目允许任意两堆（不相邻也能合，如“合并果子”），应换用小根堆贪心（$O(n \log n)$），不要用 $O(n^3)$ 区间 DP！

先看一个小例子：石子围成一圈会怎样？

还是刚刚的 4 堆石子 $[4,5,9,4]$，这次把它们摆在圆形操场上（第 4 堆和第 1 堆也相邻！）：

- 在线性 $[1,4]$ 上，第 1 堆的 4 和第 4 堆的 4 隔着 5 和 9，没法一开始就合并，最小花费是 **44**。
- 在圆环上，第 4 堆和第 1 堆紧挨着！先合这两堆 $4+4=8$，再与 5 合并得 $8+5=13$，最后与 9 合并得 $13+9=22$，总花费是 $8+13+22=$ **43**！
- $43<44$！这说明跨过首尾边界的合并方案可能是更优的，直接套线性区间 $[1,n]$ 会漏掉它！

本页要点

环形问题允许第 n 堆和第 1 堆直接合并，线性区间 `[1, n]` 根本装不下这种“跨越首尾”的区间！

为什么不能直接在转移时对下标取模 r % n？

很多初学者会想：“既然是个环，我直接让 dp[3][1] 表示从第 3 堆跨过 4 绕到第 1 堆不行吗？”——绝对不行，会立刻踩中三大灾难：

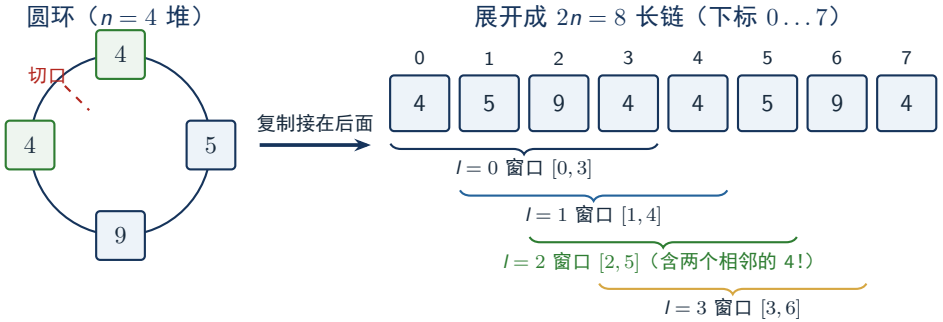
1. 长度算不对：当 $l = 3, r = 1$ 时，$r - l + 1 = -1$！你没法用 `l + len - 1` 直接算出带取模的右端点而不撞上合法线性区间。
2. 填表顺序彻底乱套：如果允许 $l > r$，斜线填表法（$l \le r$）失效；而且如果枚举 k 时不慎取模，可能把同一堆石子绕了一圈又合到自己身上！
3. 前缀和没法直接减：遇到 $l = 3, r = 1$ 时，不能简单写 `s[r] - s[l-1]`，到处要写 `if (l <= r) ... else ...` 分情况讨论，极易写错！

本页要点

千万别在二维 dp[l][r] 里对下标取模！取模会让区间左右端点大小颠倒、长度混乱、前缀和失效。

TikZ 图解：把圆环剪一刀，复制成 $2n$ 长链！

把长度为 n 的数组 $a_0, a_1, \ldots, a_{n-1}$ 复制一份接在后面，变成长度 $2n$ 的链：



本页要点

复制成 $2n$ 长链后，所有长为 n 的窗口 `[l, l+n-1]`（$l = 0 \ldots n-1$）正好对应圆环上的 n 种切法！

为什么这样就能不重不漏？“最后一条未合并的边”视角

- 关键观察：在圆环上进行 $n-1$ 次合并，一共只有 n 条相邻缝隙，最后一定有至少 1 条缝隙自始至终没有被跨过！
- 如果我们把这条“自始至终没被跨过的缝隙”用剪刀剪开，整个圆环的合并过程就等价于一条长度为 n 的直线段上的合并！
- 这 n 条可能的切口位置，恰好对应 $2n$ 长链中起点为 $l = 0, 1, \ldots, n-1$ 的 n 个长度为 n 的滑动窗口 $[l, l+n-1]$。
- 我们在 $2n$ 长链上跑一遍普通线性区间 DP（长度只需算到 `len <= n`），再把这 n 个窗口的结果取极值，就等于把所有环形合并方案都试了一遍！

本页要点

任何环形合并方案都至少留下一条没跨过的边，所以必属于某个长为 n 的线性窗口 `[l, l+n-1]`！

环形手算跟踪表：拆开 $[4,5,9,4]$ 的 4 个长度为 n 窗口

把 $[4,5,9,4]$ 复制成 8 格长链：$a=[4,5,9,4,4,5,9,4]$（下标 $0\ldots7$），列出所有长为 4 的窗口结果：

切口起点 l	窗口 $[l,l+3]$	对应序列	最小花费 `lo[l][l+3]`	最大花费 `hi[l][l+3]`
$l=0$	$[0,3]$	$[4,5,9,4]$	44	54
$l=1$	$[1,4]$	$[5,9,4,4]$	44	54
$l=2$	$[2,5]$	$[9,4,4,5]$	**43**（先合 $4+4=8$）	52
$l=3$	$[3,6]$	$[4,4,5,9]$	**43**（先合 $4+4=8$）	54
汇总（终止）	扫 $l=0\ldots3$	取全窗口 min / max	min = **43**	max = **54**

- 注意看 $l=0$ 窗口得出最小是 44，而 $l=2,3$ 窗口把两个 4 摆在了一起，顺利算出了全局最小 43！

本页要点

跑完 $2n$ 区间 DP 后，千万不能只输出 `dp[0][n-1]`，必须扫一遍 $l=0\ldots n-1$ 取全部窗口的最优值！

模板代码：断环成链 $2n$ 环形区间 DP

```
// a[0..n-1] 为原环，s[0..2n] 为复制后的前缀和
for (int i = 0; i < 2 * n; ++i) s[i + 1] = s[i] + a[i % n];
for (int len = 2; len <= n; ++len) {            // 长度仍只需枚举到 n!
    for (int l = 0; l + len <= 2 * n; ++l) {    // 左端点扫遍 2n 长链全部合法区间
        int r = l + len - 1;
        lo[l][r] = LLONG_MAX / 4;
        long long cost = s[r + 1] - s[l];
        for (int k = l; k < r; ++k)
            lo[l][r] = min(lo[l][r], lo[l][k] + lo[k + 1][r] + cost);
    }
}
long long ans = LLONG_MAX;
for (int l = 0; l < n; ++l) ans = min(ans, lo[l][l + n - 1]); // 扫所有 n 个窗口!
```

本页要点

注意两点：长度上限仍是 `len <= n`（不是 $2n$）；左端点条件是 `l + len <= 2 * n`!

为什么这样写？下标与数组大小细节

- 为什么 `len` 只枚举到 n 而不是 $2n$？我们只需要长度为 n 的窗口 `lo[l][l+n-1]`，没必要算长度为 $n+1 \dots 2n$ 的区间，还能省下一大半时间！
- 为什么左端点循环条件是 `l + len <= 2 * n`？因为 0-based 下标下，左端点 l、长 len 的右端点是 $r = l + len - 1$，要求 $r < 2n$ 即 $l + len \le 2n$。例如当 $l = n-1, len = n$ 时，$r = 2n-2 < 2n$ 完全合法。
- 为什么数组空间要开到 $2n$（而不是 n）？因为第 $n-1$ 个窗口 $[n-1, 2n-2]$ 的右端点高达 $2n-2$；若是下一节的《能量项链》用缝隙标记，甚至需要 $2n+1$ 个点！
- 用 `a[i % n]` 算前缀和的好处：不用写两遍拷贝循环，一行 `s[i+1] = s[i] + a[i%n]` 就把 $2n$ 长链前缀和建好。

本页要点

开数组永远按 $2n$（或 $2n+5$）留足余量，循环条件用 `l + len <= 2 * n`，保证不越界也不漏窗口。

正确性论证：切口等价不变量

不变量阶段	在环形 DP 中成立的理由
初始化	$2n$ 长链上任意单个节点 `lo[i][i] = 0`，对应 1 堆石子合并代价为 0
保持	当 `len` 从 2 增至 n 时，任何起点 l、长度 `len` 且不越过 $2n$ 的线性区间都已求得最优解
终止	环上任意合法合并树必有某条边未被跨过，从而对应某个起点 $l \in [0, n-1]$ 的区间 $[l, l+n-1]$；反之每个窗口也对应合法的环形合并树，故取所有 n 个窗口最小值必为全局最优

- 唯一例外边界：当 $n = 1$ 时，环上只有 1 堆石子，不需要合并，循环跳过，取 `lo[0][0] = 0` 依然正确！

本页要点

“环上所有合并树”与“n 个长 n 窗口的合并树并集”一一对应，取极值不重不漏！

线性区间 DP vs 环形区间 DP：一表看懂全部区别

对比维度	线性区间 DP	环形区间 DP（断环成链）
数组与前缀和长度	n（或 $n+1$）	$2n$（或 $2n+1$），用 `a[i % n]` 复制
外层长度 `len`	`len = 2 .. n`	依然是 `len = 2 .. n`（不用算到 $2n$!）
中层左端点 `l`	`l + len <= n`（0-based）	`l + len <= 2 * n`
最终答案取法	直接输出 `dp[0][n-1]`	扫描 $l = 0 \ldots n-1$，取 `dp[l][l+n-1]` 最值

本页要点

记住“三不变一变”：外层 `len<=n`、内层 `k`、转移方程全部不变；只把长度翻倍到 $2n$ 并扫全部 n 个窗口！

上机验证（1/2）：可运行的环形最小花费短程序

```cpp
#include <bits/stdc++.h>
using namespace std; int main() {
    int n; cin >> n; vector<long long> a(n), s(2 * n + 1, 0);
    for (int i = 0; i < n; ++i) cin >> a[i];
    for (int i = 0; i < 2 * n; ++i) s[i + 1] = s[i] + a[i % n];
    vector<vector<long long>> lo(2 * n, vector<long long>(2 * n, 0));
    for (int len = 2; len <= n; ++len)
        for (int l = 0; l + len <= 2 * n; ++l) {
            int r = l + len - 1; lo[l][r] = LLONG_MAX / 4;
            for (int k = l; k < r; ++k) lo[l][r] = min(lo[l][r], lo[l][k] + lo[k +
    1][r] + s[r + 1] - s[l]);
        }
    long long ans = LLONG_MAX;
    for (int l = 0; l < n; ++l) ans = min(ans, lo[l][l + n - 1]);
    cout << ans << '\n'; }
```

上机验证（2/2）：3 组数据实测与反事实追问

用上面短程序测试下面 3 组环形石子数据（输出最小总花费）：

测试类型	输入（n 与环形序列）	预期输出（最小花费）
题面样例	$n=4$，4 5 9 4	43（$l=2$ 窗口先合 $4+4$）
最小边界 $n=1$	$n=1$，7	0（不用合并，直接输出 `lo[0][0]=0`）
三堆不等环	$n=3$，1 2 3	9（先合 $1+2=3$，再合 $3+3=6$）

反事实追问

1. 如果复制了 $2n$ 数组，但循环里误写 `for (int l = 0; l + len <= n; ++l)` 会发生什么？（右端点永远跨不过 $n-1$，退化成线性 DP，样例输出 44 而不是 43，**WA**！）
2. 如果最后忘了扫 $l=0\ldots n-1$，只输出 `lo[0][n-1]` 会发生什么？（同样只看第 0 个切口，样例输出 44，**WA**！）

环形 DP 易错对照与调试指南

常见错误	错误表现	正确做法与原因
数组只开 n 没开 $2n$	访问 `lo[n][2n-2]` 越界崩溃	数组大小至少开到 $2n+5$（**RE**）
漏扫后半段窗口	只算到 `l + len <= n`	必须写 `l + len <= 2 * n`，才能覆盖跨首尾的区间（**WA**）
最后只输出 `lo[0][n-1]`	只能算到一种切口	必须对 $l=0\ldots n-1$ 取 min / max（**WA**）
`ans_lo` 初值太小	写 `ans_lo = 1e9`	当 $n=200$、权值较大时总和超 10^9，必须用 `LLONG_MAX`（**WA**）

本页要点

断环成链四步曲：开 $2n$、算 $2n$ 前缀和、循环到 `l+len <= 2*n`、扫 n 个窗口取极值！

例 1 | 石子合并（一本通 1569）[基础]：先读懂要求

大白话题意

在圆形操场上绕圈摆着 n 堆石子（$1 \le n \le 200$）。每次只能选相邻的两堆合并成新的一堆，本次得分为新堆的石子总数。求把全部石子合并成一堆的最小总得分和最大总得分。

项目	题目要求
输入	第 1 行整数 n；第 2 行 n 个整数，依次表示每堆石子的数量
处理	进行 $n-1$ 次相邻合并，分别求总得分的最小值与最大值
输出	共两行：第 1 行为最小总得分，第 2 行为最大总得分
原题样例	输入 `4 / 4 5 9 4` $\longrightarrow$ 输出 43（第一行）和 54（第二行）

本页要点

审题第一反应：看到“圆形操场”立刻想到断环成链 $2n$；看到“只能选相邻两堆”确定是区间 DP！

例 1 | 思路与建模：为什么不能像“合并果子”那样用小根堆？

错误想法：小根堆贪心（合并果子）

在 $[4,5,9,4]$ 中先挑最小的 4 和 4 合并（环上恰好相邻）；但下一轮剩下 $[8,5,9]$ 时，5 和 8、9 都相邻还行；若序列是 $[1,100,1,100]$，贪心会想合两个 1，但它们不相邻，规则禁止！

相邻限制

正确建模：环形区间 DP

1. 把数组复制成 $2n$ 长度，算前缀和 $s[0\ldots2n]$；
2. 设 `lo[l][r]` 为 $[l,r]$ 合并最小得分，`hi[l][r]` 为最大得分；
3. 枚举分界点 $k \in [l, r-1]$：

$$\texttt{lo[l][r]} = \min(\texttt{lo[l][k]} + \texttt{lo[k+1][r]}) + \text{cost}$$

$$\texttt{hi[l][r]} = \max(\texttt{hi[l][k]} + \texttt{hi[k+1][r]}) + \text{cost}$$

本页要点

“合并果子”允许任意两堆合并所以用小根堆；本题强制相邻合并，必须用区间 DP 枚举最后分界点 k！

例 1 | 状态转移拆解：区间总和为什么能用前缀和 $O(1)$ 算？

- 把 $[l,k]$ 和 $[k+1,r]$ 两堆合并时，无论前面经过了多少次合并，左堆此时的石子总重必然是 $\sum_{i=l}^{k} a_i$，右堆总重必然是 $\sum_{i=k+1}^{r} a_i$。
- 两堆拼成 $[l,r]$ 的本次得分 $=\sum_{i=l}^{r} a_i = s[r+1]-s[l]$（0-based 前缀和），它是一个只与 l,r 有关、与分割点 k 无关的常数！
- 因此转移方程可以放心写成：
  `lo[l][r] = min(lo[l][r], lo[l][k] + lo[k+1][r] + cost)`。
- 初值细节：单堆区间 $[i,i]$ 不需要合并，`lo[i][i] = hi[i][i] = 0`；算长度 `len >= 2` 时，`lo[l][r]` 先设为 `LLONG_MAX / 4`，`hi[l][r]` 设为 0。

本页要点

合并最后一步的得分永远是“整段石子的总重量”，用前缀和 $s[r+1]-s[l]$ 就能 $O(1)$ 算出来！

例 1 | 样例手推跟踪表：$[4,5,9,4]$ 如何得到 43 与 54

复制成 $2n=8$ 格：$[4,5,9,4,4,5,9,4]$。第 2 节已算出各窗口，下面跟踪关键区间 $[2,5]=[9,4,4,5]$ 的最优合并路径：

阶段 len	关键区间	cost	最佳分界点与计算过程	lo / hi
2	$[2,3]=[9,4]$, $[3,4]=[4,4]$, $[4,5]=[4,5]$	$13,8,9$	单堆 $0+0+\text{cost}$	$13,8,9$
3	$[2,4]=[9,4,4]$	17	最小：$k=2\Rightarrow 0+8+17=25$	lo=25
3	$[3,5]=[4,4,5]$	13	最小：$k=4\Rightarrow 8+0+13=21$	lo=21
4（最小窗口）	$[2,5]=[9,4,4,5]$	22	$k=2\Rightarrow 0+21+22=$ **43**	lo=43
4（最大窗口）	$[0,3]=[4,5,9,4]$	22	先合 $5+9=14$，再合 18，再合 22	hi=54

本页要点

四个长为 4 的窗口中，lo 在 $l=2,3$ 取到最小值 43；hi 在 $l=0,1,3$ 取到最大值 54！

例 1 | 核心步骤与相对模板的变化

按顺序计算	纸上得到	核对问题
读入并建 $2n$ 前缀和	$s[0 \dots 2n]$	`s[i+1] = s[i] + a[i%n]` 是否覆盖 $0 \dots 2n-1$?
按 `len = 2..n` 填表	更新所有 `lo[l][r]`, `hi[l][r]`	`lo[l][r]` 是否初值为 `LLONG_MAX/4`?
扫 n 个长度 n 窗口	`ans_lo=43`, `ans_hi=54`	是扫 $l = 0 \dots n-1$ 还是只取了 0?

相对基础环形模板改了哪里

1. 本题同时要求最小值和最大值，因此在同一套三层循环里顺便维护 `lo[l][r]`（取 min）和 `hi[l][r]`（取 max）；
2. 输出时分两行依次打印 `ans_lo` 和 `ans_hi`。

例 1 | 核心代码（1/2）：读入、$2n$ 前缀和与主循环

```
int n; if (!(cin >> n)) return 0;
vector<long long> a(n), s(2 * n + 1, 0);
for (auto &x : a) cin >> x;
for (int i = 0; i < 2 * n; ++i) s[i + 1] = s[i] + a[i % n];
vector<vector<long long>> lo(2 * n, vector<long long>(2 * n, 0));
vector<vector<long long>> hi(2 * n, vector<long long>(2 * n, 0));
for (int len = 2; len <= n; ++len) {        // 第一层必须枚举长度 len!
    for (int l = 0; l + len <= 2 * n; ++l) {
        int r = l + len - 1;
        lo[l][r] = LLONG_MAX / 4; hi[l][r] = 0;
        long long cost = s[r + 1] - s[l];
```

本页要点

使用 vector 动态开 $2n \times 2n$ 空间，既不会爆栈也能适配任意 $n \le 200$。

例 1 | 核心代码（2/2）：区间合并转移与环形汇总输出

```
        for (int k = l; k < r; ++k) {        // 枚举最后一次合并的分界线 k
            lo[l][r] = min(lo[l][r], lo[l][k] + lo[k + 1][r] + cost);
            hi[l][r] = max(hi[l][r], hi[l][k] + hi[k + 1][r] + cost);
        }
    }
}
long long ans_lo = LLONG_MAX, ans_hi = 0;
for (int l = 0; l < n; ++l) {                // 扫全部 n 个长度为 n 的窗口
    ans_lo = min(ans_lo, lo[l][l + n - 1]);
    ans_hi = max(ans_hi, hi[l][l + n - 1]);
}
cout << ans_lo << '\n' << ans_hi << '\n';
```

本页要点

完整可运行源码见 code/stone_merge.cpp；注意 ans_lo 和 ans_hi 分两行输出！

例 1 | 易错点与 3 组自测边界

- 常见错误 **1**：输出顺序写反（先输出最大再输出最小）——务必核对题面“第一行最小，第二行最大”（**WA**）。
- 常见错误 **2**：只开一维数组或忘了 `hi` 初始化为 0 ——若复用多组数据会污染（**WA**）。

自测类型	输入数据	预期输出
题面样例	`4 / 4 5 9 4`	`43 \n 54`
最小边界 $n=1$	`1 / 100`	`0 \n 0`（不用合并，0 次得分）
两堆边界 $n=2$	`2 / 3 7`	`10 \n 10`（只有一种合并方式）

例 2 | 能量项链（一本通 1570）[提高]：先读懂要求

大白话题意

n 颗能量珠围成一圈（$4 \le n \le 100$）。每颗珠子有“头标记”和“尾标记”，且第 i 颗珠子的尾标记等于第 $i+1$ 颗珠子的头标记（第 n 颗的尾等于第 1 颗的头）。若将标记为 (m, r) 和 (r, n) 的两颗相邻珠子聚合成 (m, n)，会释放 $m \times r \times n$ 的能量。求总能量的最大值。

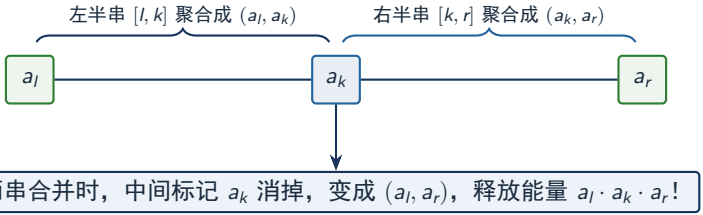
项目	题目要求
输入	第 1 行整数 n；第 2 行 n 个正整数，依次为第 $1 \ldots n$ 颗珠子的头标记
处理	找一种顺时针聚合顺序，使释放的总能量最大（答案不超过 2.1×10^9）
输出	仅一行一个整数，表示最大总能量
原题样例	输入 `4 / 2 3 5 10` $\longrightarrow$ 输出 `710`

本页要点

本题的难度在于：每颗珠子有两个标记（头和尾），怎么用“一维数组 + 区间下标”来表示它？

例 2 | 思路与建模：把珠子变成“两个缝隙标记之间的线段”

把珠子的头尾标记画成数轴上的节点 $a_0, a_1, \ldots, a_n$，每颗珠子就是相邻两个节点之间的一段：



- 关键建模：第 i 颗珠子（0-based）对应区间 $[i, i+1]$，长 `len = 1`！连续 *len* 颗珠子对应端点区间 $[l, l+\text{len}]$！
- 因此合并两段时共用中间端点 k：左段是 $[l,k]$，右段是 $[k,r]$（不是 $[k+1,r]$！），k 满足 $l<k<r$。

本页要点

《石子合并》是点区间 $[l,k]+[k+1,r]$；《能量项链》是缝隙区间 $[l,k]+[k,r]$，中间节点 k 被两段共用！

例 2 | 为什么不需要单独算“新珠子”的头尾标记数组？

- 很多同学疑惑：每次聚合两颗珠子后，头尾标记都会变，难道不要在 DP 状态里把“新珠子的头尾标记”存下来吗？
- 完全不用！把连续珠子 $[l, r]$ 聚合成只剩 1 颗珠子时，不管中间先合左边还是先合右边，消掉的都是内部标记 $a_{l+1} \dots a_{r-1}$，最左端的头标记永远是 a_l，最右端的尾标记永远是 a_r！
- 这正是它满足“无后效性”的原因：只要给定区间 $[l, r]$，聚合后的结果珠子必定是 (a_l, a_r)。
- 所以最终最后一步把 (a_l, a_k) 和 (a_k, a_r) 聚合时，释放的能量永远是 $a_l \cdot a_k \cdot a_r$！

本页要点

任意一段连续珠子 $[l, r]$ 最终一定聚合成 (a_l, a_r)，头尾标记被左右端点唯一锁死！

例 2 | 样例手推跟踪表：$[2,3,5,10]$ 如何聚出 710

把 $2,3,5,10$ 复制成 $2n+1=9$ 个标记点：$a=[2,3,5,10,2,3,5,10,2]$（下标 $0\ldots8$）。其中窗口 $[3,7]$ 对应的珠子序列恰好是 $(10,2),(2,3),(3,5),(5,10)$：

len	区间 $[l,r]$（珠数）	转移计算（加 $a_l\cdot a_k\cdot a_r$）	dp[l][r]
2	$[3,5]$（$10,2,3$）	$k=4:0+0+10\times2\times3=60$	60
2	$[4,6]$（$2,3,5$）	$k=5:0+0+2\times3\times5=30$	30
2	$[5,7]$（$3,5,10$）	$k=6:0+0+3\times5\times10=150$	150
3	$[3,6]$（$10,2,3,5$）	$k=4:0+30+10\times2\times5=130$；$k=5:60+0+10\times3\times5=210$	**210**
4（终止）	$[3,7]$（全部 4 颗）	$k=6:$ dp[3][6] $+0+10\times5\times10=210+500=$ **710**	**710**

本页要点

四个窗口 $[0,4]=250,\ [1,5]=300,\ [2,6]=460,\ [3,7]=710$，最大值恰好为原题样例的 710！

例 2 | 核心步骤与相对模板的变化

按顺序计算	纸上得到	核对问题
复制 $2n+1$ 个标记	$a[0 \dots 2n]$	是复制到 `i <= 2*n`（共 $2n+1$ 格）吗？
按 `len = 2..n` 算右端点	$r = l + \text{len}$	注意不用减 1，因为 $[l, r]$ 含 $r-l$ 颗珠子！
枚举中间标记 k 合并	$dp[l][r]$ 取 max	k 范围是 $l+1 \le k < r$，且右半段是 `dp[k][r]`！

相对基础环形模板改了哪里

1. 必须多复制第 $2n$ 号格：`a[2*n] = a[0]`，才能代表最后一颗珠子的尾标记；
2. 状态区间采用“标记端点法”：`r = l + len`，合并 $[l, k]$ 与 $[k, r]$，释放 $a[l] \cdot a[k] \cdot a[r]$，最后扫 $l = 0 \dots n-1$ 的 `dp[l][l+n]`。

例 2 | 核心代码（1/2）：复制 $2n+1$ 标记与数组初始化

```
int n; if (!(cin >> n)) return 0;
vector<long long> a(2 * n + 1);              // 必须开 2n+1，含最后一个尾标记 a[2n]！
for (int i = 0; i < n; ++i) cin >> a[i];
for (int i = n; i <= 2 * n; ++i) a[i] = a[i % n];
vector<vector<long long>> dp(2 * n + 1, vector<long long>(2 * n + 1, 0));
```

本页要点

因为 $1000^3 = 10^9$，多步累加最高接近 2.1×10^9（接近 int 上限），全用 long long 最稳妥！

例 2 | 核心代码（2/2）：缝隙区间转移与汇总输出

```
for (int len = 2; len <= n; ++len) {          // len 表示区间内包含的珠子颗数
    for (int l = 0; l + len <= 2 * n; ++l) {
        int r = l + len;                      // 端点跨度：共 len+1 个标记、len 颗珠子
        for (int k = l + 1; k < r; ++k)       // 中间标记 k 被左右两段共享
            dp[l][r] = max(dp[l][r], dp[l][k] + dp[k][r] + a[l] * a[k] * a[r]);
    }
}
long long ans = 0;
for (int l = 0; l < n; ++l) ans = max(ans, dp[l][l + n]);
cout << ans << '\n';
```

本页要点

完整可运行源码见 code/energy_necklace.cpp；注意最后取的是 dp[l][l + n] 而不是 l+n-1！

例 2 | 易错点与 3 组自测边界

- 常见错误 **1**：数组只开 $2n$ 却写 `i <= 2*n` 或访问 `a[2*n]` ——直接越界崩溃（**RE**）。
- 常见错误 **2**：习惯性写成 `dp[l][k] + dp[k+1][r]` ——少算了一颗珠子，逻辑彻底错位（**WA**）。

自测类型	输入数据	预期输出
题面样例	`4 / 2 3 5 10`	710
全相等 4 珠	`4 / 5 5 5 5`	375（每次合都释 125，共 3 次）
最小允许边界 $n = 4$	`4 / 1 1 1 1000`	2002000（保留 1000 当端点的策略最优）

例 3 | 括号配对（一本通 1572）[提高]：先读懂要求

大白话题意

给定一个由圆括号（ ）和方括号 [ ] 组成的字符串（长度 < 100）。规则允许在任意位置插入若干括号，使整个串成为合法配对串（即题目里的 GBE：空串合法；若 A 合法则 (A) 与 $[A]$ 合法；若 A, B 合法则 AB 合法）。求最少需要增加多少个字符。

项目	题目要求
输入	仅一行，为一个长度 < 100 的括号字符串
处理	在任意位置添加最少个数的字符，使整串变成合法括号串
输出	仅一个整数，表示增加的最少字符数
原题样例	输入 `[])` $\longrightarrow$ 输出 `1`（在最前面补一个 `(` 变成 `([])`）

本页要点

为什么不能只靠“左括号减右括号”计数？因为 `([)]` 左右数量相等却不合法，必须按区间匹配！

例 3 | 为什么不能直接用第 1 篇的“括号匹配栈”？

- 学过栈的同学第一反应往往是：从左到右扫一遍，能配对就弹栈，最后栈里剩几个字符就补几个？
- 为什么栈贪心是对的？咦，等等！让我们来测一组反例：（[)]！
  - 如果只按栈的顺序：遇到（压栈，遇到 [压栈，遇到) 时栈顶是 [匹配不上，栈会误以为) 必须单补，结果在（[)] 上给出要补 4 个或 2 个；而其实在（后面补] 再在后面补 [变成（[])[] 只需要补 2 个！
  - 更狠的例子：[](] 只需要在最右边补) 或内部补 1 个；但有些嵌套串里，是保留外层还是先消内层会影响全局最优！
- 栈只能“就近强行配对”，无法尝试“跳过当前字符让后面更优”的所有可能；只有区间 DP 能穷举全部合法结构！

本页要点

判断括号串是否合法用栈（$O(n)$）；要求“最少插入多少字符变合法”，必须用区间 DP（$O(n^3)$）！

例 3 | 思路与建模：非合并类区间 DP 的两种决策分支

设 `dp[l][r]` 表示把半开区间 $[l, r)$（即子串 $s[l \dots r-1]$）补全为合法串所需的最少字符数。考虑最左边的字符 $s[l]$：

方案 1：单独给 $s[l]$ 补一个搭档

无论 $s[l]$ 是左括号还是右括号，我都直接花 1 个字符给它配对好，剩下的问题变成 $[l+1, r)$：

```
dp[l][r] = 1 + dp[l+1][r]
```

两者取 min

方案 2：找后面的 $s[k]$ 跟 $s[l]$ 牵手

若存在 $k \in [l+1, r-1]$ 使得 $(s[l], s[k])$ 恰好是 `()` 或 `[]`，则让它俩包住中间，再拼上右侧剩余：

```
dp[l][r] = dp[l+1][k] + dp[k+1][r]
```

本页要点

把半开区间 $[l, r)$ 按“$s[l]$ 是自己单补，还是找某个 $s[k]$ 牵手”分类，正好覆盖所有合法结构！

例 3 | 为什么只枚举 $s[l]$ 的搭档就够了，不用再枚举其他断点？

- 很多同学会担心：题目说“若 A, B 合法则 AB 合法”，那我是不是还要像石子合并一样无条件枚举 k 做 `dp[l][k] + dp[k][r]` 拼起来？
- 其实已经包含了！看看最终合法串里最左边字符 $s[l]$（或它被补出的搭档）的命运，只有两种可能：
  1. 它自己和“新补出来的字符”配成一对（比如在 `]`）里，先补（ 配对，剩下 $[l+1, r)$；
  2. 它和原串中某个本来就存在的 $s[k]$ 配成一对，形成 $(A)B$ 或 $[A]B$，其中 A 对应 $[l+1, k)$，B 对应 $[k+1, r)$！
- 当 A 为空时 $[l+1, l+1)$ 自动为 0；当 B 为空时 $[r, r)$ 也自动为 0！半开区间 $[l, r)$ 让空串处理极其干净。

本页要点

用半开区间 `[l, r)` 时，所有 $l \geq r$ 的格子（空串）天生是 0，根本不用写一堆 `if` 特判越界！

例 3 | 样例手推跟踪表：输入 []）如何推导出 1

字符串 $s=$ "[])"，长度 $n=3$，按半开区间长度 len = 1，2，3 填表（未配对时先算方案 1）：

len	区间 $[l,r)$	子串	方案 1（单补）与方案 2（找 $s[k]$ 配对）	dp[l][r]
1	$[0,1)$，$[1,2)$，$[2,3)$	[，]，）	只有 1 个字符，只能单补：1+dp[1+1][r] = 1	均为 1
2	$[0,2)$	[]	方案 1 得 $1+1=2$；方案 2（$k=1$ 配对）：dp[1][1] + dp[2][2] = 0	**0**
2	$[1,3)$	]）	方案 1 得 $1+1=2$；] 和 ） 不能配对	2
3（终止）	$[0,3)$	[]）	方案 1：1 + dp[1][3] = 3；方案 2（$k=1$ 配对）：dp[1][1] + dp[2][3] = 0 + 1 = **1**	**1**

本页要点

在 $[0,3)$ 里，$k=1$ 让 [和] 牵手消掉（中间为空 0），只剩右侧 ）需要补 1 个字符！

例 3 | 核心步骤与相对模板的变化

按顺序计算	纸上得到	核对问题
整行读入并清理回车	字符串 s（长 n）	Windows 下 getline 是否去掉了末尾 '\r'？
从 len = 1..n 填表	dp[0..n][0..n]	初值是否全为 0（表示空串无需添加）？
单补与牵手取 min	dp[0][n] = 1	配对时是否严格检查了 () 和 []（左在前右在后）？

相对基础区间模板改了哪里

1. 不是环形，无需复制成 $2n$；
2. 使用半开区间 $[l, r)$，循环从 len = 1 开始，初值直接用方案 1 1 + dp[l+1][r] 兜底，无需设无穷大；
3. 方案 2 只在 $s[l]$ 是左括号且 $s[k]$ 是对应右括号时转移，拆分出 dp[l+1][k] + dp[k+1][r]。

例 3 | 核心代码（1/2）：安全读入与数组初始化

```
string s;
getline(cin, s);
if (!s.empty() && s.back() == '\r') s.pop_back(); // 兼容 Windows 换行符!
int n = (int)s.size();
vector<vector<int>> dp(n + 1, vector<int>(n + 1, 0));
for (int len = 1; len <= n; ++len) {          // 必须按长度从小到大枚举
    for (int l = 0; l + len <= n; ++l) {
        int r = l + len;
        dp[l][r] = 1 + dp[l + 1][r];          // 方案 1: 为 s[l] 单独补一个搭档
```

本页要点

为什么用 `getline` 而不用 `cin >> s`？如果测试数据是一个空行（空串），`getline` 能正确读出长 0 并输出 0！

例 3 | 核心代码（2/2）：括号牵手转移与输出

```
        for (int k = l + 1; k < r; ++k) {   // 方案 2: 找 s[k] 与 s[l] 配对
            if ((s[l] == '(' && s[k] == ')') ||
                (s[l] == '[' && s[k] == ']')) {
                dp[l][r] = min(dp[l][r],
                               dp[l + 1][k] + dp[k + 1][r]);
            }
        }
    }
}
cout << dp[0][n] << '\n';
```

本页要点

完整可运行源码见 code/bracket_pairing.cpp；注意)(不能配对，所以必须左括号在 l、右括号在 k！

例 3 | 易错点与 3 组自测边界

- 常见错误 **1**：允许 `)(` 也配对——`)(` 不是合法括号，至少要补成 `()()`（需加 2 个），必须判断 $s[l]$ 是左括号（**WA**）。
- 常见错误 **2**：忘了处理 `'\r'` ——在 Windows 本地测试文件末尾带 `\r` 会被算成多余字符（**WA**）。

自测类型	输入数据	预期输出
题面样例	`[])`	1
完美匹配	`([])()`	0（不用加任何字符）
全反序交叉	`)(`	2（不能牵手，各自补搭档变 `()()`）

横向对比：闭区间 $[l, r]$、缝隙 $[l, r]$ 与半开 $[l, r)$

做区间 DP 时，最容易混的就是右端点到底减不减 1、分割点是 $k+1$ 还是 k。一张表彻底分清：

代表例题	区间含义与右端点	切分后子区间	适用场景与特点
石子合并（1569）	闭区间 $[l, r]$，$r = l + \text{len} - 1$	$[l, k]$ 与 $[k+1, r]$	元素是独立石子堆，合并后不共享端点
能量项链（1570）	缝隙 $[l, r]$，$r = l + \text{len}$	$[l, k]$ 与 $[k, r]$	矩阵链乘/多边形三角剖分，共享中间顶点/标记 k
括号配对（1572）	半开 $[l, r)$，$r = l + \text{len}$	$[l+1, k)$ 与 $[k+1, r)$	字符串匹配，$l \geq r$ 天然为空串且值为 0

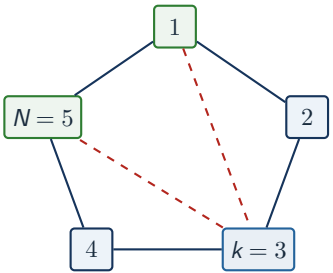
- 延伸说明：练习题单中的《凸多边形的划分》（1571）和《能量项链》属于同一类，按顶点缝隙 $[l, r]$ 划分，枚举中间顶点 k 构成三角形 (l, k, r) 即可。

本页要点

先想清楚“一个单元占几个数组格子”，就能一秒确定右端点要不要减 1、左右半段要不要共享 k！

延伸建模：凸多边形的划分（一本通 1571）与能量项链同构

把 N 个顶点的凸多边形顺时针标号为 $1 \dots N$，在多边形里连不相交的对角线切成 $N-2$ 个三角形：



观察底边 $(1,5)$ 所在的三角形：

它一定选了某个中间顶点 $k \in \{2,3,4\}$ 组成三角形 $(1,k,5)$！

选定 k 后，多边形被一刀劈成三块：

1. 左侧子多边形 `dp[1][k]`；
2. 右侧子多边形 `dp[k][5]`；
3. 中间三角形代价 $w_1 \cdot w_k \cdot w_5$！

注意：本题无需断环成链（固定底边 $(1,N)$ 即可），且点权 $\le 10^9$ 乘积达 10^{27}，需用 `__int128`！

本页要点

《凸多边形的划分》转移式 `dp[l][r] = min(dp[l][k] + dp[k][r] + w[l]*w[k]*w[r])` 与《能量项链》一模一样！

考场识别信号：看到哪些词就该往区间 DP 想？

- 信号 **1**（操作范围）：“每次只能合并/消除相邻两堆/两颗/两端字符”，或者“把序列/字符串拆成若干连续段”。
- 信号 **2**（数据范围）：$n \le 100 \sim 400$！看到 n 只有几百，几乎就是在暗示 $O(n^3)$ 的区间 DP（若 $n \le 2000$ 可能是 $O(n^2)$ 线性 DP，若 $n \le 10^5$ 多半是贪心/数据结构）。
- 信号 **3**（环形字眼）：“围成一圈”“圆形操场”“首尾相连的项链” $\Longrightarrow$ 立刻触发“复制成 $2n$ 长链”。
- 区间 **DP** 五步思考法：
  1. 判断线性还是环形（环形先复制 $2n$）；
  2. 选定区间含义（闭 $[l, r]$ / 缝隙 $[l, r]$ / 半开 $[l, r)$）；
  3. 写最后一步合并/配对的转移方程；
  4. 确定边界初值（$len = 1$ 为 0，求最小设 `LLONG_MAX/4`）；
  5. 按 `len` 从小到大填表并汇总答案。

本页要点

看到“相邻合并 + $n \le 400$”就想到区间 DP；再看到“圆形/项链”就加上“断环成链 $2n$”！

区间与环形 DP 高频错误对照与评测定位指南

题目 / 场景	常见错误写法	正确做法与原因
把“相邻合并”当堆贪心	套小根堆合并石子	相邻限制不允许隔山打牛挑最小两堆，必须用区间 DP（**WA**）
最外层先枚举左端点 l	外循环 `l`、内循环 `r`	会读取还没算完的右区间 $[k+1, r]$ 垃圾值！必须先枚举 `len`（**WA**）
无穷大溢出成负数	初值写 `LLONG_MAX` 或 `INT_MAX`	状态转移两项相加立刻溢出成负数，必须写 `LLONG_MAX / 4`（**WA**）
破环成链数组开小	复制只开 $2n$ 却读 `a[2n]`	项链标记需 $2n+1$ 格，且扫描全链时循环需到 `l+len <= 2*n`（**RE**）
环形只输出 `dp[0][n-1]`	忘了在 n 个起点里取最值	环可以从任意一条边切开，必须扫 `l = 0 .. n-1`（**WA**）

- 评测结果速查：**WA** 先查“是否按 `len` 递增”“无穷大除没除以 4”“有没有扫全 n 个起点”；**RE** 查“数组是否开到 $2n+5$”与“k 边界是否越界”；**TLE** 查是否把 $O(n^3)$ 写成了四层循环或深递归没记忆化。

本页要点

区间 DP 第一层循环永远是长度 `len`；环形题最后永远要对 n 个切口取最值！

评测结果（WA / RE / TLE / MLE）逐项排错流程

评测结果	区间 DP 最常见原因	5 秒自查动作
WA（答案错）	循环顺序写错 / 初值溢出 / 漏扫 n 个窗口	检查外层是不是 `len`；检查是不是 `LLONG_MAX/4`；检查是不是扫了 $l = 0 \dots n-1$
RE（运行崩）	环形数组没开够 $2n+5$ / k 越界	检查《能量项链》是否开了 $2n+1$；检查闭区间是否写了 `k < r`
TLE（超时）	没存前缀和每次暴力循环求和 / 环形算到 $len = 2n$	把区间和预处理成前缀和 $O(1)$；把外层上限改回 `len <= n`
MLE（超内存）	全局开了过大的 `long long dp[5005][5005]`	$n \le 400$ 时开 $2n+5 \le 810$ 仅占约 5 MB，切勿盲目开几千维

本页要点

遇到 **WA** 优先用 $n = 3$ 或 $n = 4$ 环形小样例打印每个窗口的 `dp[l][l+n-1]`，一眼就能看出哪一步算错！

提交前检查清单（六必查）

1. 三层循环的最外层是不是从短到长枚举长度 `len`，确保子区间绝对算完后再合并？
2. 求最小值初始化是不是用了安全的无穷大（如 `LLONG_MAX / 4`），并且长度 1 区间初值为 0？
3. 环形问题有没有把原数组复制接在末尾变成 $2n$（或 $2n+1$）长，并确保 `dp` 数组维度开够 $2n+5$？
4. 最后汇总环形答案时，是不是扫描了所有 `l = 0 .. n-1` 的长 n 窗口，而不是只输出了第 0 个区间？
5. 合并代价（如能量乘积之和、大石子堆总和）是否已用 `long long` 存储？
6. 当 $n = 1$（只有一堆石子）或空串时，循环是否安全跳过并输出 0？

最后一步

对 $n = 3$ 环形石子手推全部 3 种切口，与代码跑出结果一一对照后再交！

本章全部练习与完成标准

题目（一本通题号）	难度	核心思路提要
石子合并（1569）	基础	断环成链 $2n$ + 前缀和，按 `len=2..n` 同步维护最小 `lo` 与最大 `hi`
能量项链（1570）	提高	标记点复制 $2n+1$，缝隙区间 $[l,r]$ 共享中间点 k，转移加 $a[l]\cdot a[k]\cdot a[r]$
凸多边形的划分（1571）	提高	与能量项链同构的顶点区间 $[l,r]$ 划分三角形 (l,k,r)，注意乘积用 `__int128`
括号配对（1572）	提高	线性半开区间 $[l,r)$，左端单补 $1+$ `dp[l+1][r]` 与找搭档 $s[k]$ 取 min

完成标准

能一秒说出为什么不能从左到右枚举 l,r 而必须先枚举长度 `len`；能独立默写破环成链 $2n$ 模板；独立 AC 前两道环形经典题并掌握括号区间匹配。

课后安排与下一章预告

- 复习建议：合上课件再推一遍《能量项链》为什么用缝隙端点 `dp[l][k] + dp[k][r]` 而不写 `k+1`，以及为什么取模会毁掉填表顺序。
- 动手练习：把 `code/` 目录下的 `stone_merge.cpp`、`energy_necklace.cpp`、`bracket_pairing.cpp` 独立敲一遍并用自测边界数据验证。
- 下一章预告：第 3 章《树形与换根 DP》——不再在一维序列和圆环上合并区间，而是把“小块合成大块”的舞台搬到树上，先自底向上汇总子树，再通过二次扫描换根！

本章一句话

区间 DP 按长度从小到大斜线拼图，圆环题复制成双倍长链把全部切口一把收齐。

谢谢！