空间计算：
总内存 = 单个元素占用内存 × 数组总元素总数

int 类型：每个元素占用 4字节（Byte）
long long 类型：每个元素占用 8字节（Byte）
内存单位换算：
- 1KB = 1024 Byte
- 1MB = 1024KB = 1024×1024 ≈ 100万 Byte（精确值1,048,576 Byte）
- 题目中常说的“256MB”“512MB”即指总内存上限。

实例 1：定义 int a[1000000]（100 万个元素）

总字节数：1,000,000 × 4 = 4,000,000 Byte

换算为 MB：4,000,000 ÷ 1024 ÷ 1024 ≈ 3.81 MB

实例2：定义 long long c[1000][1000]（1000行1000列）

总元素数：1000 × 1000 = 1,000,000

总字节数：1,000,000 × 8 = 8,000,000 Byte

换算为MB：8,000,000 ÷ 1024 ÷ 1024 ≈ 7.63 MB

数组定义：long long arr[300000][300000]

单个元素内存：long long 类型占 8 Byte

总元素数：行数 × 列数 = 300,000 × 300,000 = 90,000,000,000

总字节数 = 总元素数 × 单个元素字节数

= 90,000,000,000 × 8= 720,000,000,000 Byte

1MB = 1024 × 1024 = 1,048,576 Byte

总内存（MB）= 720,000,000,000 ÷ 1,048,576 ≈ 686,645.5 MB

1GB = 1024 MB

总内存（GB）= 686,645.5 ÷ 1024 ≈ 670.55 GB

# 区间dp

TEACHING COURSEWARE POWPOINT

授课时间：2025.08.15

# 目录

01

# 复习

TEACHING
COURSEWARE

TEACH

# 概念



动态规划的核心思想
动态规划是一种解决复杂问题的方法，它将问题分解为相对简单的子问题，通过解决子问题并保存子问题的解（避免重复计算），最终高效地解决原问题。其核心在于：

重叠子问题： 原问题可以分解成若干子问题，这些子问题在求解过程中会被重复计算多次。

最优子结构： 原问题的最优解包含其子问题的最优解。也就是说，我们可以通过组合子问题的最优解来构造原问题的最优解。

状态转移方程：描述如何从子问题的解推导出当前问题的解（是前两个性质的具体体现）。

# 解题关键



定义状态：
这是最重要也最困难的一步。
状态就是描述子问题的一组变量。你需要找到能唯一确定一个子问题并刻画其“进展程度”的变量。
通常用一个数组 dp[i] 或 dp[i][j] 来表示状态，其中 i，j 等是状态变量（也叫状态参数）。
问需要存储什么信息才能在后续步骤中利用已经解决的更小规模的问题
例子：

斐波那契数：dp[i] 表示第 i 个斐波那契数。

爬楼梯：dp[i] 表示爬到第 i 阶楼梯的方法数。

02

# 区间dp

TEACHING
COURSEWARE

TEACH

# 区间dp

区间的表示与前缀和：

区间 [l, r] 的 “总和 / 总质量” 是高频需求

区间的枚举顺序：

线性 DP 通常按 “i 从 1 到 n” 枚举（前 i 个元素），但区间 DP 要 “先算小区间，再算大区间”—— 因为大区间的解依赖小区间的解。

举例：要算 [1,4]（长度 4），必须先算完所有长度 1（[1,1]、[2,2]...）、长度 2（[1,2]、[2,3]...）、长度 3（[1,3]、[2,4]）的区间。

03

# 例题

TEACHING
COURSEWARE

TEACH

# 例题



## 题目描述

复制 Markdown 展开 进入 IDE 模式

设有 $N(N \leq 300)$ 堆石子排成一排，其编号为 $1, 2, 3, \cdots, N$。每堆石子有一定的质量 $m_i$ $(m_i \leq 1000)$。现在要将这 $N$ 堆石子合并成为一堆。每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻。合并时由于选择的顺序不同，合并的总代价也不相同。试找出一种合理的方法，使总的代价最小，并输出最小代价。

## 输入格式

第一行，一个整数 $N$。

第二行，$N$ 个整数 $m_i$。

## 输出格式

输出文件仅一个整数，也就是最小代价。

## 输入输出样例

| 输入 #1 | 复制 | 输出 #1 | 复制 |
| --- | --- | --- | --- |
| 4<br>2 5 3 1 | | 22 | |

# 例题



状态定义：dp[l][r] 表示 —— 将 区间 [l, r] 内的所有石子合并成一堆，所需要的 最小总代价。

比如 dp[1][4] 就是合并第 1 到第 4 堆石子的最小代价。

前缀和数组：sum[r] - sum[l-1] 表示 区间 [l, r] 内所有石子的总质量（记为 total[l][r]）。

这是因为每次合并两堆石子时，“合并代价” 就是这两堆的质量和 —— 而无论怎么合并 [l, r]，最终一定会有一步是 “合并 [l, k] 的总堆” 和 “[k+1, r] 的总堆”，这一步的代价就是 total[l][r]（两堆总质量之和）。

# 例题

定义 DP 状态：设dp[l][r]表示合并第l堆到第r堆石子的最小代价（l和r均从 1 开始，对应石子堆的编号）。

预处理前缀和：用sum[r]表示前r堆石子的总质量，通过sum[r] - sum[l-1]可快速计算出第l堆到第r堆的总质量（合并两堆石子的代价等于两堆总质量，需重复使用）。

初始化最小区间：当区间长度为 1（即l == r）时，只有一堆石子，无需合并，代价为 0，故dp[l][l] = 0。

枚举区间长度与起点：按区间长度从小到大枚举（从 2 到 N），确保计算大区间时，所有小区间的解已得出。对每个长度len，枚举所有可能的起点l，并计算终点r = l + len - 1。

枚举分割点计算代价：将区间[l, r]拆分为[l, k]和[k+1, r]（k从l到r-1），合并[l, r]的代价等于合并[l, k]的代价 + 合并[k+1, r]的代价 + 两区间总质量（即sum[r] - sum[l-1]）。取所有分割点的最小代价作为dp[l][r]的值。

# 例题



## 代码解释

输入与前缀和：先读取石子堆数n和每堆质量，再通过前缀和数组sum快速计算任意区间的总质量，避免重复累加。

DP 表初始化：dp是二维数组，初始时长度为 1 的区间代价为 0，其他区间先设为极大值。

区间遍历：按长度从 2 开始遍历（长度 1 无需处理），对每个起点l计算终点r，再遍历所有分割点k，计算合并代价并更新最小值。

输出结果：dp[1][n]存储了合并所有石子堆的最小代价，直接输出即可。

# 例题



```cpp
#include<bits/stdc++.h>
using namespace std;
// 静态数组大小：题目中N≤ 300，直接定义足够大的数组
int m[305];      // 存储每堆石子质量（1-based）
int sum[305];    // 前缀和数组（sum[r] = 前r堆总质量）
int dp[305][305];  // DP表：dp[l][r]为合并l到r堆的最小代价
int main() {
    int n;
    cin >> n;
    // 1. 读取石子质量并计算前缀和
    sum[0] = 0;
    for (int i = 1; i <= n; ++i) {
        cin >> m[i];
        sum[i] = sum[i - 1] + m[i];
    }
```

# 例题



```
    // 2. 初始化DP表：长度为1的区间代价为0
    // 3. 枚举区间长度（从2到n，长度1无需处理）
    for (int len = 2; len <= n; ++len) {
        // 枚举起点l，确保终点r = l + len - 1 ≤ n
        for (int l = 1; l + len - 1 <= n; ++l) {
            int r = l + len - 1;
            dp[l][r] = INT_MAX;  // 初始设为极大值，方便后续取最小值
            // 枚举分割点k，拆分为[l,k]和[k+1,r]
            for (int k = l; k < r; ++k) {
                // 计算当前分割点的代价：子区间代价 + 合并当前两堆的代价（区间总质量）
                int cost = dp[l][k] + dp[k + 1][r] + (sum[r] - sum[l - 1]);
                if (cost < dp[l][r]) {
                    dp[l][r] = cost;  // 更新最小代价
                }
            }
        }
    }
    // 输出合并所有石子（1到n堆）的最小代价
    cout << dp[1][n] << endl;
    return 0;
}
```

04

# 例题进阶

TEACHING
COURSEWARE

TEACH

# 例题进阶



## 题目描述

复制 Markdown　展开　进入 IDE 模式

在一个圆形操场的四周摆放 $N$ 堆石子，现要将石子有次序地合并成一堆，规定每次只能选相邻的 2 堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。

试设计出一个算法,计算出将 $N$ 堆石子合并成 1 堆的最小得分和最大得分。

## 输入格式

数据的第 1 行是正整数 $N$，表示有 $N$ 堆石子。

第 2 行有 $N$ 个整数，第 $i$ 个整数 $a_i$ 表示第 $i$ 堆石子的个数。

## 输出格式

输出共 2 行，第 1 行为最小得分，第 2 行为最大得分。

## 输入输出样例

| 输入 #1 | 复制 | 输出 #1 | 复制 |
| --- | --- | --- | --- |
| 4<br>4 5 9 4 | | 43<br>54 | |

## 说明/提示

$1 \leq N \leq 100$，$0 \leq a_i \leq 20$。

# 例题进阶

这道题的唯一核心差异是 “环形结构”，而解决方案（环形转线性：将数组展开为 2N 长度）非常直观：

原理好理解：圆形石子堆没有固定起点，把数组 “复制一份接在后面”，就能用线性区间覆盖所有可能的 “断环方式”（比如 N=4 的 [4,5,9,4] 展开为 [4,5,9,4,4,5,9,4]，任意长度为 4 的子区间都对应一种断环结果）；

# 例题进阶



```cpp
#include<bits/stdc++.h>
using namespace std;
int a[305];      // 展开后的石子质量数组（1-based）
int sum[305];    // 前缀和数组
int min_dp[305][305];  // 最小得分DP表
int max_dp[305][305];  // 最大得分DP表
int main() {
    int n;
    cin >> n;
    // 1. 读取石子质量并展开为2N长度（环形转线性）
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
        a[i + n] = a[i];  // 后N个元素复制前N个，实现环形展开
    }
    // 2. 计算前缀和（用于快速获取区间总质量）
    sum[0] = 0;
    for (int i = 1; i <= 2 * n; ++i) {
        sum[i] = sum[i - 1] + a[i];
    }
```

# 例题进阶



```
// 3. 初始化DP表：长度为1的区间无需合并，得分0
for (int i = 1; i <= 2 * n; ++i) {
    min_dp[i][i] = 0;
    max_dp[i][i] = 0;
}
// 4. 区间DP：按长度从小到大枚举（从2到n，因最终要合并n堆）
for (int len = 2; len <= n; ++len) {  // 注意：len最大为n（合并n堆）
    // 枚举起点l，确保终点r = l + len -1 ≤ 2n
    for (int l = 1; l + len - 1 <= 2 * n; ++l) {
        int r = l + len - 1;
        // 初始化最小得分（设极大值）、最大得分（设极小值）
        min_dp[l][r] = INT_MAX;
        max_dp[l][r] = 0;
        // 枚举分割点k，拆分为[l,k]和[k+1,r]
        for (int k = l; k < r; ++k) {
            // 区间[l,r]的总质量（合并时的基础得分）
            int total = sum[r] - sum[l - 1];
            // 更新最小得分：子区间最小得分之和 + 总质量
            min_dp[l][r] = min(min_dp[l][r], min_dp[l][k] + min_dp[k + 1][r] + total);
            // 更新最大得分：子区间最大得分之和 + 总质量
            max_dp[l][r] = max(max_dp[l][r], max_dp[l][k] + max_dp[k + 1][r] + total);
        }
    }
}
```

# 例题进阶



```
    // 5. 找所有可能的n堆区间（断环点不同），取最值
    int ans1 = INT_MAX;
    int ans2 = 0;
    for (int l = 1; l <= n; ++l) {  // 起点l从1到n，对应不同断环方式
        int r = l + n - 1;
        ans1 = min(ans1, min_dp[l][r]);
        ans2 = max(ans2, max_dp[l][r]);
    }
    cout << ans1 << endl;
    cout << ans2 << endl;

    return 0;
}
```