P16230 [蓝桥杯 2026 省 A] 综合应变指标
题解

1 题意概述

给定长度为 N 的整数序列 A，选取三个分割点 i, j, k（满足 $1 \le i < j < k < N$）将序列划分为四个非空连续子段，最大化四段元素和的绝对值之和：

$$|A_1 + \cdots + A_i| + |A_{i+1} + \cdots + A_j| + |A_{j+1} + \cdots + A_k| + |A_{k+1} + \cdots + A_N|$$

2 第一层思考：前缀和改写目标函数

定义前缀和 $P_m = \sum_{t=1}^{m} A_t$（$P_0 = 0$）。四段的元素和分别为 P_i、$P_j - P_i$、$P_k - P_j$、$P_N - P_k$。目标函数改写为：

$$f(i, j, k) = |P_i| + |P_j - P_i| + |P_k - P_j| + |P_N - P_k|$$

这一步把” 子段求和” 全部消解成了前缀和之差的绝对值，后续只需操作 P 数组。

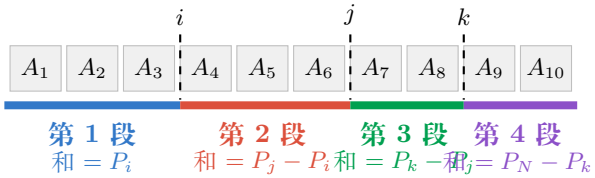

Figure 1: 将序列 A 在分割点 i, j, k 处划分为四段。每段的元素和均可用前缀和 P 简洁表示。

直接枚举三个分割点的时间复杂度为 $O(N^3)$，对 $N = 10^5$ 不可接受。能否把三重循环逐层拆解，每层都只做线性扫描？

3 第二层思考：$O(N^2)$ 的朴素代码

问题本身就是分段的，自然地逐段转移。定义四层 DP：`dp1[i]` 表示只切出第一段、该段以位置 i 结尾时的最大贡献；`dp2[i]` 表示前两段的最大贡献，第二段以 i 结尾；`dp3`、`dp4` 同理。答案为 `dp4[N]`。

第一层直接算：

```
dp1[i] = abs(preA[i]);
```

从 `dp1` 到 `dp2`、从 `dp2` 到 `dp3`、从 `dp3` 到 `dp4`，转移结构完全一样，写成一个通用函数：

```
void excute_dp(const vector<ll> &dp1, vector<ll> &dp2) {
  for (int i = 1; i <= N; ++i) {
    for (int j = 1; j <= i; ++j) {                    // ← 内层循环
      dp2[i] = max(dp2[i], dp1[j] + abs(preA[i] - preA[j]));
    }
  }
}
```

调用三次 `excute_dp`，每次 $O(N^2)$。瓶颈就在内层 `for(j)` 循环——能不能干掉它？

4 第三层思考：干掉内层循环

盯着内层循环体看：

```
dp2[i] = max(dp2[i], dp1[j] + abs(preA[i] - preA[j]));
```

abs 只有两种展开。代入后，把只跟 j 有关的项括起来：

```
dp1[j] + preA[i] - preA[j]  =  (dp1[j] - preA[j]) + preA[i]
//                               ^^^^^^^^^^^^^^^^^^    ^^^^^^^^
//                               只跟 j 有关            只跟 i 有关

dp1[j] + preA[j] - preA[i]  =  (dp1[j] + preA[j]) - preA[i]
//                               ^^^^^^^^^^^^^^^^^^    ^^^^^^^^
//                               只跟 j 有关            只跟 i 有关
```

所以内层循环等价于：

```
dp2[i] = max(
  max_over_j(dp1[j] - preA[j]) + preA[i],
  max_over_j(dp1[j] + preA[j]) - preA[i]
);
```

max_over_j(dp1[j] - preA[j]) 和 max_over_j(dp1[j] + preA[j]) 不依赖 i！而且随着 i 递增，j 的候选范围只增不减（$j \le i$）。用两个变量滚动维护前缀最大值就行了，内层循环彻底消失：

```
void excute_dp(const vector<ll> &from, vector<ll> &to) {
  ll Mplus = -INF, Mminus = -INF;
  for (int i = 1; i <= N; ++i) {
    Mplus  = max(Mplus,  from[i] + preA[i]);   // 维护 max(dp1[j]+preA[j])
    Mminus = max(Mminus, from[i] - preA[i]);   // 维护 max(dp1[j]-preA[j])
    to[i]  = max(Mminus + preA[i], Mplus - preA[i]);
  }
}
```

双重循环 → 单层循环，$O(N^2) \to O(N)$。调用三次，总计 $O(N)$。

4.1 图像理解

上面的优化可以用图 2 直观看懂。

对每个候选 j，dp1[j] + abs(preA[i] - preA[j]) 是关于 preA[i] 的一条 V 形曲线——谷底在 preA[j]，谷底高 dp1[j]，两臂斜率 ±1。内层循环就是对所有 V 形取最大值（上包络）。

关键：所有 V 形的斜率全一样（±1），不同的只是截距。同斜率的平行线取最大值，只有截距最大的一条存活。所以上包络只由两条直线决定——截距就是代码里的 Mminus 和 Mplus。

i 每递增一步，候选集多出一条新 V 形。新 V 形的截距和当前 Mminus、Mplus 取一次 max 就完成更新——这就是循环体里那两行 max 在干的事。

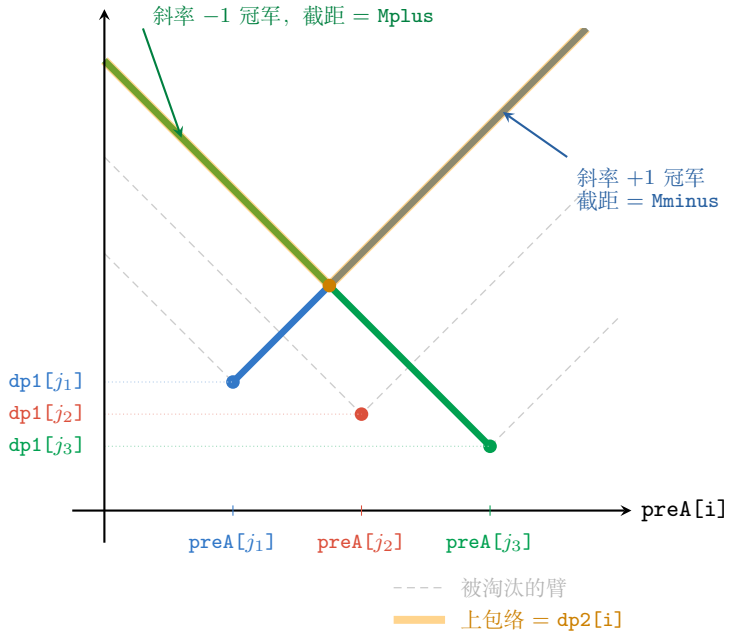

Figure 2: 每个候选 j 产生一条 V 形（谷底高 dp1[j]，位于 preA[j]）。所有 V 形斜率一样（±1），上包络只由截距最大的两条臂决定——截距就是代码里的 Mminus 和 Mplus。其余臂（灰色虚线）全被淘汰。

5 代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

static constexpr ll INF = (1ll << 61) - 1;

struct Solve {
  ll N;
  vector<ll> A, preA;
  vector<ll> dp1, dp2, dp3, dp4;

  // 核心：优化后的转移，O(N)
  void excute_dp(const vector<ll> &from, vector<ll> &to) {
    ll Mplus = -INF, Mminus = -INF;
    for (int i = 1; i <= N; ++i) {
      Mplus  = max(Mplus,  from[i] + preA[i]);
      Mminus = max(Mminus, from[i] - preA[i]);
      to[i]  = max(Mminus + preA[i], Mplus - preA[i]);
    }
  }

  Solve() {
    cin >> N;
    A.resize(N + 2);
    for (int i = 1; i <= N; ++i)
      cin >> A[i];
    preA.resize(N + 2);
    partial_sum(A.begin(), A.end(), preA.begin());

    dp1.resize(N + 2, -INF);
    for (int i = 1; i <= N; ++i)
      dp1[i] = abs(preA[i]);

    dp2.resize(N + 2, -INF);
```

```
    dp3.resize(N + 2, -INF);
    dp4.resize(N + 2, -INF);
    excute_dp(dp1, dp2);
    excute_dp(dp2, dp3);
    excute_dp(dp3, dp4);
    cout << dp4[N] << "\n";
  }
};

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  Solve solve;
  return 0;
}
```

6 复杂度分析

- **时间复杂度**：$O(N)$。`excute_dp` 单次 $O(N)$，调用三次，总计 $O(N)$。
- **空间复杂度**：$O(N)$。四个 DP 数组加前缀和数组。