C++ 算法讲义：前缀和

第一部分：问题引

1. 场景模拟：高频区间查询

假设这样一个数据处理场景：班级里有 100,000 名学生站成一排，已知每个人的考试分数。

此时，班主任频繁提出查询需求："第 3 名到第 25 名学生的总分是多少？"

最直观的解法是编写一个 `for` 循环，将区间内的分数逐一累加。

但这仅仅是单次查询。如果系统在极短时间内（例如 1 秒内）发起了 10 万次不同区间的查询请求（如"第 10 到 80 名"、"第 2 到 99,999 名"），会发生什么？

2. 暴力解法的缺陷分析

如果每次询问都通过 `for` 循环遍历相加，单次查询的时间复杂度为 $O(N)$。如果有 M 次查询，总时间复杂度将高达 $O(N \cdot M)$。

在 N 和 M 均为 100,000 的量级下，程序需要执行约 100 亿次运算，这在任何算法竞赛都会导致超时（TLE）。

除时间复杂度高之外，暴力解法的核心缺陷在于**存在大量的重复计算。**

例如，程序刚刚计算完"第 2 名到第 6 名"的总和，紧接着又要计算"第 2 名到第 5 名"的总和。在暴力解法中，程序会无视之前的计算成果，将第 2 到第 5 名的数据重新遍历累加一遍。

思考：

是否存在一种算法结构，能够将计算过的中间结果缓存下来，消除重复遍历？无论查询的区间跨度有多大，都能在常数时间 $O(1)$ 内直接返回答案？

第二部分：类比思考

想象你正驾驶汽车行驶在一条笔直的高速公路上。这条公路被划分为了多个**路段。**

1. **真实路段（原数组 a）**：每一段路都有其独立的长度。例如第 1 段路长 10 公里，第 2 段路长 20 公里。
2. **累计里程碑（前缀和数组 S）**：在每一段路结束的终点处，都设立了一块里程碑，上面记录着"从公路起点行驶至此的总里程"。
    - 驶出第 1 段路时，里程碑标记为 10。
    - 驶出第 2 段路时，里程碑标记为 30（即 10 + 20）。

第 i 个里程碑的位置，对应第 i 段路的终点。

第三部分：推导公式

- 现在，我们需要计算**"第 3 段路 + 第 4 段路 + 第 5 段路"**的总长度（即求原数组区间 `[3, 5]` 的和）。

  依据直觉，许多初学者会尝试直接使用两端的里程碑相减：**用第 5 个碑的读数减去第 3 个碑的读数。**

  直觉陷阱剖析：

如果你以“第 3 个碑”作为起点进行计算，请注意你当前所处的位置：你在**第 3 段路的终点**！

如果你从这里开始计算距离，你所包含的第一段路将是**第 4 段路**。这样一来，第 3 段路的长度被完全遗漏了。

为了确保计算结果**完整包含**第 3 段路，我们的计算基准点必须设置在第 3 段路的**起点**。

而在累计里程碑系统中，第 3 段路的起点，在物理位置上等同于**第 2 段路的终点**，即**第 2 个里程碑**所在的位置。

因此，计算第 3 到第 5 段路总距离的正确等式为：

第 5 个里程碑的读数 − 第 2 个里程碑的读数。

抽象为通用算法公式：

若需求取原数组从第 L 个元素到第 R 个元素的总和，设前缀和数组为 S：

- 区间的终点标记为：$S[R]$。
- 区间的起点标记（必须是第 L 段开始前的位置）为：$S[L-1]$。
- 区间查询公式：

  $$Sum(L,R) = S[R] - S[L-1]$$

第四部分：前缀和数组 与 如何高效求取?

而我们刚才反复提到的那个用来记录里程碑的数组 S[]，其实就叫做：
—— 前缀和数组 (Prefix Sum Array)

简单的来说，前缀和数组就是一个用来存储累加和的辅助数组。

那么如何在程序中高效地生成这个 S 数组（即建立所有的里程碑）呢?

在构建第 i 个里程碑时，我们无需从第 1 段路重新累加。只需查阅紧邻的前一个里程碑的数值，再加上当前这一段路的长度即可。

公式：

当前前缀和 = 前一个前缀和 + 当前元素的值

$$S[i] = S[i-1] + a[i]$$

第五部分：C++ 代码

```
long long a[100005]; // 原数组
long long s[100005]; // 前缀和数组（里程碑）
// 第一步：处理前缀和数组

for (int i = 1; i <= n; i++) {
    cin >> a[i];
    // 递推：当前总和 = 前一个总和 + 当前数字
    s[i] = s[i-1] + a[i];
}


// 第二步：查询第 L 个元素到第 R 个元素的区间和
for (int i = 0; i < q; i++) {
```

```
    int L, R;
    cin >> L >> R;

    // 终点 - 起点的前一个
    cout << S[R] - S[L-1] << endl;
}
```

1. **数组下标从 1 开始！**

- 如果要求区间 `[1, 5]` 的和，套公式是 $S[5] - S[0]$。我们刚好就默认 $S[0] = 0$，非常完美。
- 如果你非要从下标 0 开始存数据（求 `[0, 4]`），公式就会变成 $S[4] - S[-1]$。数组下标变成负数，程序直接越界崩溃！**所以，写前缀和，数组从 1 开始！否则你就要特判下标为 -1 的情况了。**

2. **前缀和数组务必开 `long long`！**

- 原数组 a 的数字可能不大，但十万个数加在一起很有可能超过 `int` 的上限（约 21 亿）。为了不爆数据，前缀和数组请开 `long long`。