树状数组（Fenwick Tree），也称为二叉索引树（Binary Indexed Tree，BIT），是一种用于处理前缀和或区间查询问题的数据结构。它的主要作用如下：

1. **高效计算前缀和**：树状数组能够在 $O(\log n)$ 的时间复杂度内计算前缀和，即数组中从第一个元素到指定位置的所有元素的和。
2. **高效更新元素**：树状数组能够在 $O(\log n)$ 的时间复杂度内更新数组中的某个元素。更新后，树状数组会自动维护相应的前缀和。
3. **支持动态查询和修改**：与直接维护前缀和的简单数组不同，树状数组支持在不重新计算整个数组的情况下进行动态的查询和修改操作。

树状数组的应用场景

- **动态数组的区间和查询**：在动态更新数组元素的情况下，快速求解某一前缀的和，或是两个下标间元素的区间和。
- **求逆序对**：在计算一个数组中的逆序对数量时，树状数组可以用于高效计算逆序对的个数。
- **数列的动态排序**：在需要动态排序的场景下，可以使用树状数组来维护当前数列的顺序，快速找到插入位置等操作。

树状数组的基本操作

1. **单点更新**：更新数组中某个位置的值，并更新与之相关的前缀和。
2. **前缀和查询**：查询从第一个元素到某个位置的和。
3. **区间和查询**：通过两次前缀和查询得到区间和，即 $\text{sum}(l, r) = \text{sum}(r) - \text{sum}(l-1)$。

树状数组的设计使得它在很多需要频繁查询与更新的场景中，成为一种非常高效的工具。

树状数组的基本操作主要包括**单点更新**和**前缀和查询**。下面将用C++详细讲解这两种操作的实现。

1. 树状数组的基本结构

树状数组使用一个数组来存储信息，常用的表示方式是 `bit[]`，其中 `bit[i]` 表示在树状数组中与原数组 `arr[i]` 相关的某个区间的和。

树状数组的大小通常为 `n+1`，其中 `n` 是原数组的长度。索引 `0` 通常不使用，数组从索引 `1` 开始。

2. 单点更新操作

假设我们要将原数组的某个位置 `i` 的值增加 `delta`，树状数组的更新操作如下：

```
void update(int i, int delta, int n, vector<int>& bit) {
    while (i <= n) {
        bit[i] += delta;
        i += i & (-i);
    }
}
```

解释：

- `i` 是要更新的位置。
- `delta` 是要增加的值。
- `n` 是数组的大小。
- `bit[]` 是存储树状数组的数组。

更新过程：

- 每次更新时，`bit[i]` 的值增加 `delta`。
- 然后通过 `i += i & (-i)` 移动到树状数组的下一个位置进行更新。`i & (-i)` 计算出 `i` 的最低位 1 的值，对应于二进制中最低有效位。

例如，假设 `i = 5`，则 `i` 对应二进制为 `101`，`i & (-i)` 计算出的值为 `1`，表示更新与位置 `i` 相关的区间和。

3. 前缀和查询操作

假设我们要查询从数组起始位置到 `i` 的前缀和：

```
int query(int i, vector<int>& bit) {
    int sum = 0;
    while (i > 0) {
        sum += bit[i];
        i -= i & (-i);
    }
    return sum;
}
```

解释：

- `i` 是查询的终止位置。
- `bit[]` 是存储树状数组的数组。

查询过程：

- 从 `i` 开始，累加 `bit[i]` 的值到 `sum`。
- 然后通过 `i -= i & (-i)` 移动到前一个区间，继续累加。
- 直到 `i` 减少到 0 时，返回累加的 `sum` 值。

4. 区间和查询操作

区间 `[l, r]` 的和可以通过两次前缀和查询得到：

```
int rangeQuery(int l, int r, vector<int>& bit) {
    return query(r, bit) - query(l - 1, bit);
}
```

解释：

- `query(r)` 返回从起始位置到 `r` 的前缀和。
- `query(l - 1)` 返回从起始位置到 `l-1` 的前缀和。
- 两者相减即为 `[l, r]` 的区间和。

5. 完整示例

下面是一个完整的示例，演示如何使用树状数组进行更新和查询操作：

```cpp
#include <iostream>
#include <vector>

using namespace std;

// 更新操作：在位置 i 增加 delta
void update(int i, int delta, int n, vector<int>& bit) {
    while (i <= n) {
        bit[i] += delta;
        i += i & (-i);
    }
}

// 前缀和查询：查询前缀和 [1, i]
int query(int i, vector<int>& bit) {
    int sum = 0;
    while (i > 0) {
        sum += bit[i];
        i -= i & (-i);
    }
    return sum;
}

// 区间和查询：[l, r] 的区间和
int rangeQuery(int l, int r, vector<int>& bit) {
    return query(r, bit) - query(l - 1, bit);
}

int main() {
    int n = 10; // 数组大小
    vector<int> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // 原数组
    vector<int> bit(n + 1, 0); // 树状数组初始化为 0

    // 建立树状数组
    for (int i = 1; i <= n; ++i) {
        update(i, arr[i], n, bit);
    }

    // 查询 [1, 5] 的前缀和
    cout << "Sum of range [1, 5]: " << rangeQuery(1, 5, bit) << endl;

    // 更新位置 3，增加 6
    update(3, 6, n, bit);
```

```
    cout << "Sum of range [1, 5] after update: " << rangeQuery(1, 5, bit) << endl;

    return 0;
}
```

6. 运行结果

该程序首先建立树状数组，然后查询 `[1, 5]` 的区间和，输出 `15` （即 `1+2+3+4+5` ）。随后将位置 `3` 的值增加 `6` ，再次查询 `[1, 5]` 的区间和，输出 `21` （即 `1+2+9+4+5` ）。

通过上述示例，你可以看到树状数组在处理前缀和和区间和查询时的高效性。

查询 `i` 的最低有效位在树状数组中的作用是**确定哪些区间需要更新**或**哪些区间在查询时需要累加**。这背后涉及到树状数组的结构设计，具体来说，树状数组利用了区间的二进制表示来快速确定更新或查询的范围。

树状数组的结构和区间表示

树状数组的核心思想是使用一个一维数组来有效地管理前缀和。对于一个数组 `arr` ，树状数组通过一个额外的数组 `bit` 来存储某些区间的和，这样可以在 $O(\log n)$ 的时间内完成更新和查询。

树状数组如何划分区间

树状数组中的每个索引 `i` 代表了一个从 `i` 开始的区间，其长度是由 `i` 的最低有效位（LSB）决定的。例如：

- `i = 4` (二进制 `100`) 代表的区间为 `[4, 4]` ，即它只包含自己。
- `i = 6` (二进制 `110`) 代表的区间为 `[5, 6]` 。
- `i = 8` (二进制 `1000`) 代表的区间为 `[1, 8]` ，包含从 `1` 到 `8` 的所有元素。

这种划分是通过 `i` 的二进制表示及其最低有效位来实现的。例如：

- **最低有效位为 `1` 的数**： `i = 1, 2, 4, 8, ...` 代表的区间是从 `i` 开始到 `i` 结束的区间，这些数只有一个元素。
- **最低有效位为 `2` 的数**： `i = 3, 6, 12, ...` 代表的区间是从 `i - 1` 开始，到 `i` 结束的区间，这些数包含两个元素。

为什么要查询 `i` 的最低有效位

通过查询 `i` 的最低有效位，可以快速确定与 `i` 对应的区间以及哪些索引与其关联：

1. **更新操作**：

- 当你在位置 `i` 更新一个值时，需要更新所有包含该位置的区间的和。这些区间的索引就是根据 `i` 的最低有效位来计算的。
- 例如，当你更新 `i = 3` 时，你需要更新区间 `[3]`（由 `3` 本身表示）和区间 `[2, 3]`（由 `4` 表示），通过 `i += i & (-i)` 来跳到下一个区间并继续更新。

2. **查询操作**：
   - 当你查询前缀和时，需要累加所有包含在 `[1, i]` 中的区间的和。这些区间同样是通过 `i` 的最低有效位来确定的。
   - 例如，当你查询 `i = 6` 时，你需要累加 `[5, 6]`（由 `6` 表示）和 `[1, 4]`（由 `4` 表示）等区间的和。通过 `i -= i & (-i)`，你可以跳到前一个区间继续累加。

最低有效位的关键作用

通过 `i` 的最低有效位，树状数组能够实现以下两个关键功能：

1. **快速跳转到下一个相关区间**：在更新时，通过 `i += i & (-i)`，你可以快速找到下一个需要更新的区间索引，而不是逐一遍历所有索引。
2. **快速确定累加范围**：在查询时，通过 `i -= i & (-i)`，你可以快速跳转到下一个需要累加的区间索引，避免遍历所有可能的区间。

总结：

- 查询 `i` 的最低有效位帮助确定当前索引对应的区间大小和范围，使得树状数组能够高效地进行更新和查询操作。
- 这种设计利用了二进制的特性，确保每个操作的时间复杂度为 $O(\log n)$，大大提高了处理大规模数据时的效率。
  在使用树状数组查询区间 `[l, r]` 的和时，我们通常会通过以下方法来计算：

$$\text{Sum of range } [l, r] = \text{query}(r, \text{bit}) - \text{query}(l-1, \text{bit})$$

1. 树状数组中的前缀和查询

树状数组的查询函数 `query(i, bit)` 通常返回从数组的起始位置 `1` 到位置 `i` 的前缀和，即：

$$\text{query}(i, \text{bit}) = \text{Sum from 1 to } i$$

这意味着 `query(i, bit)` 给出的结果是从索引 `1` 到 `i` 的和。如果我们想计算任意区间 `[l, r]` 的和，我们需要减去 `[1, l-1]` 的和，以排除掉不在区间 `[l, r]` 中的部分。

2. 为什么要用 `l - 1`

计算区间 `[l, r]` 的和时，我们要得到的结果是：

$$\text{Sum from } l \text{ to } r = \text{Sum from 1 to } r - \text{Sum from 1 to } (l-1)$$

- **`query(r, bit)`** 返回 `[1, r]` 的和。
- **`query(l - 1, bit)`** 返回 `[1, l-1]` 的和。

通过减去 `[1, l-1]` 的和，我们就得到从 `l` 到 `r` 的和，因为 `[1, r]` 的和包含了 `[1, l-1]` 的部分，而我们只需要 `[l, r]` 。

3. 示例解释

假设我们有一个数组 `arr` ，树状数组 `bit` 已经建立好。我们想要计算区间 `[3, 6]` 的和：

```
int sum = query(6, bit) - query(2, bit);
```

- `query(6, bit)` 返回的是 `[1, 6]` 的和，假设它的值是 `21` 。
- `query(2, bit)` 返回的是 `[1, 2]` 的和，假设它的值是 `3` 。

那么 `21 - 3` 就是 `[3, 6]` 的和，即 `18` 。

总结

我们需要使用 `l - 1` 是因为 `query(r, bit)` 计算的是从 `1` 到 `r` 的前缀和，而 `query(l - 1, bit)` 则是计算从 `1` 到 `l-1` 的前缀和。通过计算差值，我们得到了 `[l, r]` 区间的和。

在树状数组中，区间修改（即对数组中一段连续的区间进行统一的增加或减少）通常是通过一些技巧来实现的。由于传统的树状数组主要支持单点更新和前缀和查询，要支持区间修改，我们可以采用一些特殊的设计思路。

1. 传统树状数组的局限性

通常，树状数组适合以下操作：

- **单点更新**：即在数组的某个位置 `i` 上加上或减去一个值 `delta` 。
- **前缀和查询**：查询从数组开始到某个位置 `i` 的所有元素的和。

然而，直接使用树状数组并不容易支持区间修改（例如同时对 `[l, r]` 区间内的所有元素增加 `delta` ）。

2. 双树状数组解决方案

为了实现区间修改，可以使用**两个树状数组**来分别维护差分数组。我们可以通过这两个树状数组来支持区间的加减操作。

思路

考虑一个数组 `arr` ，我们希望对其进行区间修改（比如对 `[l, r]` 区间内的所有元素增加一个值 `delta` ）。我们可以引入两个树状数组 `B1` 和 `B2` 来分别维护 `arr` 的差分信息：

- 树状数组 `B1` 用于处理与位置 `i` 成正比的影响。
- 树状数组 `B2` 用于处理常数影响。

如何更新和查询

1. **区间更新**：对 `[l, r]` 区间增加 `delta` ，分两步来做：
   - 更新 `B1` 数组： `B1[l] += delta` ， `B1[r+1] -= delta` 。
   - 更新 `B2` 数组： `B2[l] += delta * (l-1)` ， `B2[r+1] -= delta * r` 。

   具体实现：

   ```
   void range_update(int l, int r, int delta, vector<int>& B1, vector<int>& B2, int n)
       // 更新 B1
       update(l, delta, n, B1);
       update(r + 1, -delta, n, B1);
       // 更新 B2
       update(l, delta * (l - 1), n, B2);
       update(r + 1, -delta * r, n, B2);
   }
   ```

2. **前缀和查询**：查询前缀和 `sum(1...i)` 可以通过以下方式获得：
   - `sum(1...i) = query(i, B1) * i - query(i, B2)`

   具体实现：

   ```
   int prefix_sum(int i, vector<int>& B1, vector<int>& B2) {
       return query(i, B1) * i - query(i, B2);
   }
   ```

3. **区间查询**：查询区间 `[l, r]` 的和：
   - `sum(l...r) = prefix_sum(r) - prefix_sum(l-1)`

   具体实现：

```
int range_query(int l, int r, vector<int>& B1, vector<int>& B2) {
    return prefix_sum(r, B1, B2) - prefix_sum(l - 1, B1, B2);
}
```

代码示例

以下是完整的代码示例，包含区间修改和区间查询的实现：

```cpp
#include <iostream>
#include <vector>

using namespace std;

// 单点更新函数
void update(int i, int delta, int n, vector<int>& bit) {
    while (i <= n) {
        bit[i] += delta;
        i += i & (-i);
    }
}

// 单点查询函数
int query(int i, vector<int>& bit) {
    int sum = 0;
    while (i > 0) {
        sum += bit[i];
        i -= i & (-i);
    }
    return sum;
}

// 前缀和查询函数
int prefix_sum(int i, vector<int>& B1, vector<int>& B2) {
    return query(i, B1) * i - query(i, B2);
}

// 区间和查询函数
int range_query(int l, int r, vector<int>& B1, vector<int>& B2) {
    return prefix_sum(r, B1, B2) - prefix_sum(l - 1, B1, B2);
}

// 区间更新函数
void range_update(int l, int r, int delta, vector<int>& B1, vector<int>& B2, int n) {
    update(l, delta, n, B1);
    update(r + 1, -delta, n, B1);
    update(l, delta * (l - 1), n, B2);
    update(r + 1, -delta * r, n, B2);
}

int main() {
    int n = 8; // 数组长度
```

```
    vector<int> B1(n + 1, 0), B2(n + 1, 0); // 初始化两个树状数组

    // 对区间 [2, 5] 增加 3
    range_update(2, 5, 3, B1, B2, n);

    // 查询区间 [1, 5] 的和
    cout << "Sum of range [1, 5]: " << range_query(1, 5, B1, B2) << endl; // 输出应为 12

    return 0;
}
```

总结

通过使用两个树状数组，我们可以有效地实现区间更新（即对一个区间内的所有元素增加或减少某个值）和区间查询操作。这种方法的核心思想是利用差分数组的概念来维护区间的影响，从而在树状数组的基础上扩展了其功能。
在树状数组中使用“差分乘以位置”的技巧，主要是为了将区间更新转化为树状数组能够处理的前缀和查询。这个技巧通过两个树状数组的配合，确保在进行区间更新后，我们仍然可以高效地计算任意区间的和。

1. 问题背景

我们希望能够对数组 `arr` 的某个区间 `[l, r]` 内的所有元素同时加上一个值 `delta` ，并且在之后能够快速地查询任意区间的和。如果直接对每个元素单独加上 `delta` ，那么每次区间更新的复杂度将会是 $O(n)$，这对于大数组来说非常不高效。因此，我们需要一种巧妙的方式来保持操作的效率。

2. 差分数组与位置的关系

回顾差分数组的原理：对于一个数组 `arr` ，它的差分数组 `d` 是这样定义的：

$$d[i] = arr[i] - arr[i-1]$$

如果我们对区间 `[l, r]` 的所有元素加上一个值 `delta` ，差分数组会进行如下修改：

- `d[l] += delta` ：表示从位置 `l` 开始，所有元素增加 `delta` 。
- `d[r+1] -= delta` ：表示从位置 `r+1` 开始，恢复原来的数值（即对冲 `delta` 的影响）。

通过这个差分数组，我们可以将区间更新的影响从每个单独的元素转移到两个点上，从而使更新操作的复杂度降为 $O(\log n)$。

3. 为什么需要“差分乘以位置”

然而，仅仅使用一个差分数组并不足以处理查询区间和的问题。为了解决这个问题，我们使用两个树状数组：

- 一个用于存储基本的差分（ `B1` ）。
- 另一个用于存储差分乘以位置的结果（ `B2` ）。

这里的“差分乘以位置”其实是为了在查询时恢复出正确的区间和。具体来说，使用差分乘以位置的原因如下：

1. 维持累加效应的平衡

假设我们在区间 `[l, r]` 增加 `delta` ，我们希望每个位置 `i` （其中 `l ≤ i ≤ r` ）都增加 `delta` 。考虑查询前缀和时：

- 如果仅仅使用 `B1` 来存储差分数组，我们得到的是每个元素的差分累积量，这个累积量随着 `i` 增加。
- 然而，如果我们想要恢复出真正的区间和，需要考虑这些差分对每个元素的累积影响。这时候，需要在 `B2` 中存储差分乘以位置的结果，这样在计算前缀和时就能抵消不必要的多加部分。

2. 构建正确的前缀和公式

假设我们希望查询区间 `[1, i]` 的和：

- 直接使用 `B1` 的查询结果（即 `query(i, B1) * i` ）会累加过多的影响，因为 `query(i, B1)` 实际上只考虑到了基本差分，而没有考虑累积效应的差别。
- 通过减去 `query(i, B2)` ，我们就可以剔除多余的部分，得到正确的区间和。

因此，前缀和的计算公式为：

$$\text{prefix\_sum}(i) = i \cdot \text{query}(i, B1) - \text{query}(i, B2)$$

这个公式通过引入 `B2` 来校正查询结果，确保我们得到的前缀和是准确的。

4. 具体作用

- `B1` 负责记录直接的差分变化。
- `B2` 通过存储“差分乘以位置”的值，帮助校正前缀和中的多余累积，使得前缀和查询能够返回正确的值。

5. 总结

“差分乘以位置”的引入，是为了确保在树状数组处理区间更新时，我们能够正确计算任意前缀和或区间和。通过使用两个树状数组，一个记录基础差分，另一个记录差分乘以位置，我们可以在 $O(\log n)$ 时间内实现区间更新和区间查询，保持了树状数组的高效性。在树状数组中使用“差分乘以位置”的技巧，主要是为了将区间更新转化为树状数组能够处理的前缀和查询。这个技巧通过两个树状数组的配合，确保在进行区间更新后，我们仍然可以高效地计算任意区间的和。

1. 问题背景

我们希望能够对数组 `arr` 的某个区间 `[l, r]` 内的所有元素同时加上一个值 `delta` ，并且在之后能够快速地查询任意区间的和。如果直接对每个元素单独加上 `delta` ，那么每次区间更新的复杂度将会是 $O(n)$，这对于大数组来说非常不高效。因此，我们需要一种巧妙的方式来保持操作的效率。

2. 差分数组与位置的关系

回顾差分数组的原理：对于一个数组 `arr` ，它的差分数组 `d` 是这样定义的：

$$d[i] = arr[i] - arr[i-1]$$

如果我们对区间 `[l, r]` 的所有元素加上一个值 `delta` ，差分数组会进行如下修改：

- `d[l] += delta` ：表示从位置 `l` 开始，所有元素增加 `delta` 。
- `d[r+1] -= delta` ：表示从位置 `r+1` 开始，恢复原来的数值（即对冲 `delta` 的影响）。

通过这个差分数组，我们可以将区间更新的影响从每个单独的元素转移到两个点上，从而使更新操作的复杂度降为 $O(\log n)$。

3. 为什么需要“差分乘以位置”

然而，仅仅使用一个差分数组并不足以处理查询区间和的问题。为了解决这个问题，我们使用两个树状数组：

- 一个用于存储基本的差分（ `B1` ）。
- 另一个用于存储差分乘以位置的结果（ `B2` ）。

这里的“差分乘以位置”其实是为了在查询时恢复出正确的区间和。具体来说，使用差分乘以位置的原因如下：

1. 维持累加效应的平衡

假设我们在区间 `[l, r]` 增加 `delta` ，我们希望每个位置 `i` （其中 `l ≤ i ≤ r` ）都增加 `delta` 。考虑查询前缀和时：

- 如果仅仅使用 `B1` 来存储差分数组，我们得到的是每个元素的差分累积量，这个累积量随着 `i` 增加。
- 然而，如果我们想要恢复出真正的区间和，需要考虑这些差分对每个元素的累积影响。这时候，需要在 `B2` 中存储差分乘以位置的结果，这样在计算前缀和时就能抵消不必要的多加部分。

2. 构建正确的前缀和公式

假设我们希望查询区间 `[1, i]` 的和：

- 直接使用 `B1` 的查询结果（即 `query(i, B1) * i` ）会累加过多的影响，因为 `query(i, B1)` 实际上只考虑到了基本差分，而没有考虑累积效应的差别。
- 通过减去 `query(i, B2)` ，我们就可以剔除多余的部分，得到正确的区间和。

因此，前缀和的计算公式为：

$$\text{prefix\_sum}(i) = i \cdot \text{query}(i, B1) - \text{query}(i, B2)$$

这个公式通过引入 `B2` 来校正查询结果，确保我们得到的前缀和是准确的。

4. 具体作用

- `B1` 负责记录直接的差分变化。
- `B2` 通过存储“差分乘以位置”的值，帮助校正前缀和中的多余累积，使得前缀和查询能够返回正确的值。

5. 总结

“差分乘以位置”的引入，是为了确保在树状数组处理区间更新时，我们能够正确计算任意前缀和或区间和。通过使用两个树状数组，一个记录基础差分，另一个记录差分乘以位置，我们可以在 $O(\log n)$ 时间内实现区间更新和区间查询，保持了树状数组的高效性。