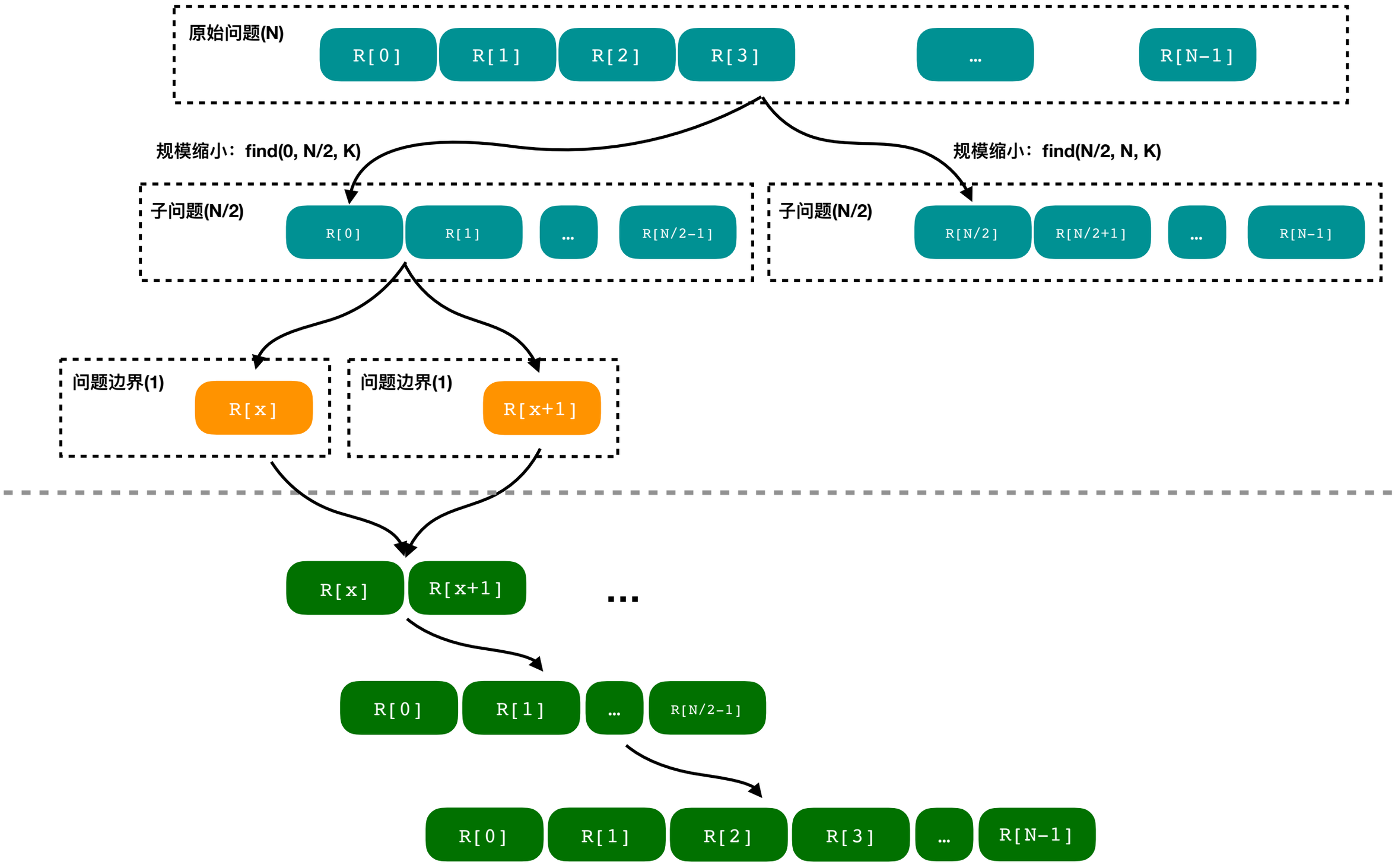
数据结构实验（9）

排序

目录

- 内部排序
  - 分治思想
    - 寻找 Pivot Point（枢轴、支点）
      - 归并排序：合并 Pivot Point 两侧的有序数组
        - `sorted( [0, N) ) = merge (sorted( [0, pivot) ), sorted( [pivot, N) ))`
      - 快速排序：按 Pivot Point 将数据分开，使得：
        - `max( [0, pivot) ) <= min( (pivot, N) )`

- 内部排序
  - 归并排序


原始问题(N)
R[0]
R[1]
R[2]
R[3]
...
R[N-1]
规模缩小：find(0, N/2, K)
规模缩小：find(N/2, N, K)
子问题(N/2)
R[0]
R[1]
...
R[N/2-1]
子问题(N/2)
R[N/2]
R[N/2+1]
...
R[N-1]
问题边界(1)
R[x]
问题边界(1)
R[x+1]
R[x]
R[x+1]
...
R[0]
R[1]
...
R[N/2-1]
R[0]
R[1]
R[2]
R[3]
...
R[N-1]

- 内部排序
  - 归并排序

```
template <typename T>
void mergesort(std::vector<T>& array, int start, int end) {
    if (end - start < 2) {
       return;
    }
    int mid = start + ((end - start) >> 1);

    mergesort(array, start, mid);
    mergesort(array, mid, end);

    merge(array, start, end, mid);
}
```

- 内部排序
  - 归并排序
    - 合并有序数组：
      - `sorted( [0, N) ) = merge (sorted( [0, pivot) ), sorted( [pivot, N) ))`
    - 逻辑：轮流从两个数组（队列）中，pop() 最小的元素，加入新队列中

- 内部排序
  - 归并排序

能否避免额外消耗O(N) 的空间复杂度?

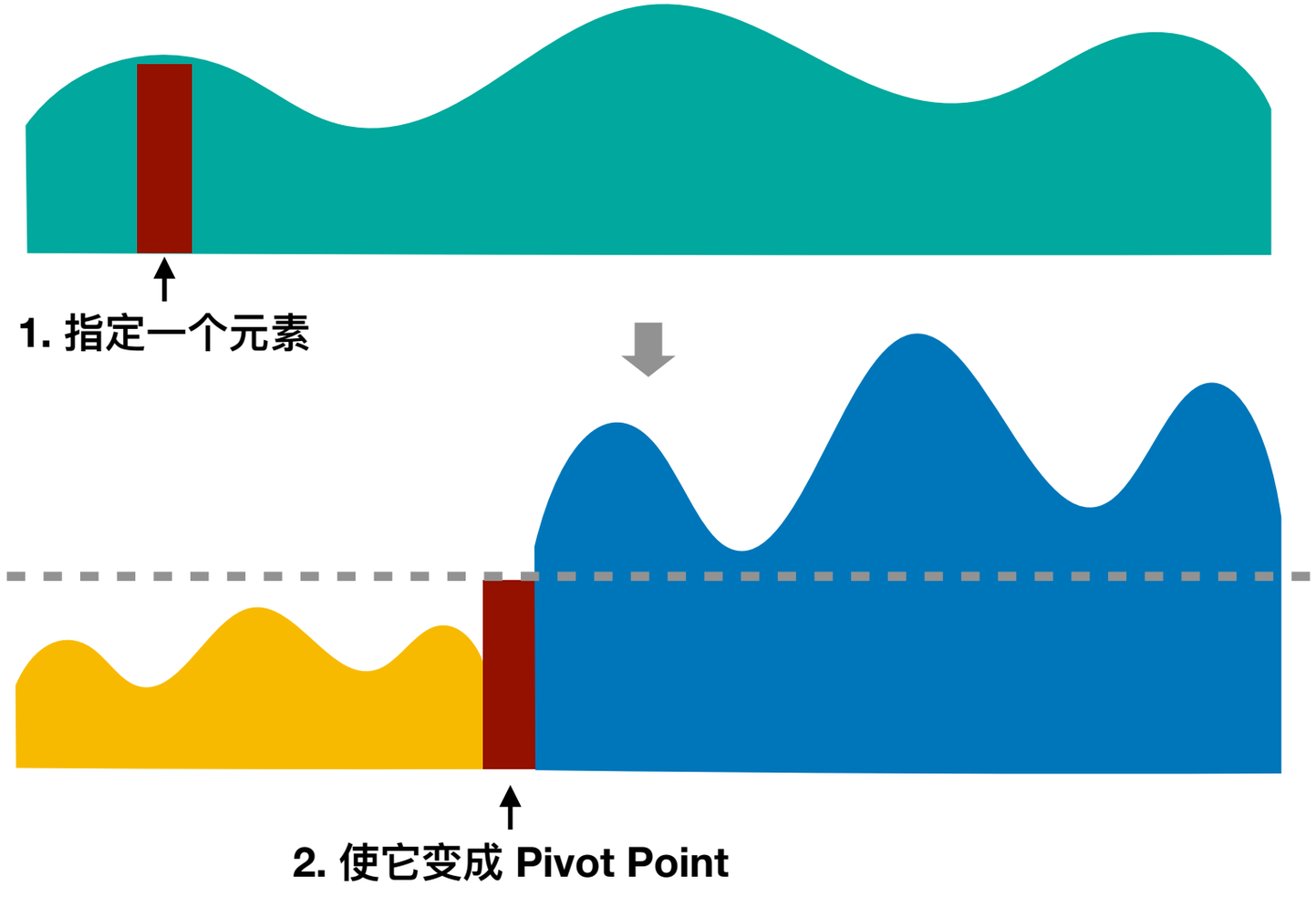
```
template <typename T>
void merge(std::vector<T>& array, int start, int end, int mid) {
    std::vector<T> array1(array.begin() + start, array.begin() + mid);
    std::vector<T> array2(array.begin() + mid, array.begin() + end);

    int i, j, k;
    for (i = 0, j = 0, k = start; i < array1.size() && j < array2.size();) {
        if (array1[i] < array2[j]) {
            array[k++] = array1[i++];
        } else {
            array[k++] = array2[j++];
        }
    }

    while (i < array1.size()) {
       array[k++] = array1[i++];
    }

   while (j < array2.size()) {
       array[k++] = array2[j++];
    }
}
```

- 内部排序
  - 归并排序

```
template <typename T>
void merge(std::vector<T>& array, int start, int end, int mid) {
    int start2 = mid;

    while (start < mid && start2 < end) {
        if (array[start] <= array[start2]) {
            start++;
        } else {
            int value = array[start2];
            int index = start2;

            while (index != start) {
                array[index] = array[index - 1];
                index--;
            }
            array[start] = value;

            start++;
            mid++;
            start2++;
        }
    }
}
```

回忆线性表的删除操作，O(N) 时间复杂度

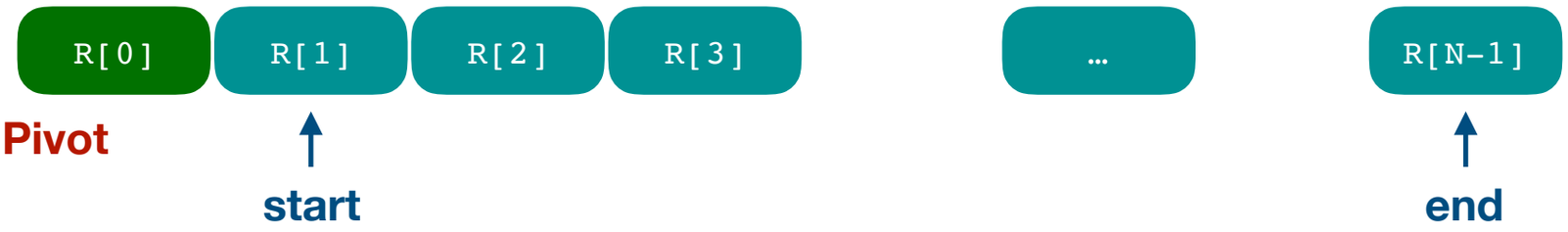
- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) < min( [pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 `Pivot Point`**



- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 Pivot Point**

```cpp
template<typename T>
void quicksort(const std::vector<T>& array, int start, int end) {
    // 分治算法的边界条件
    if (end - start < 2) {
        return;
    }

    // 寻找合适的元素，并使其变成 Pivot Point
    int pivot_point = partition(array, start, end);

    // 递归进行子序列的排序
    quicksort(array, start, pivot_point);
    quicksort(array, pivot_point + 1, end);
}
```

- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 `Pivot Point`**



- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 Pivot Point**

1. 选择 R[0] 作为 Pivot

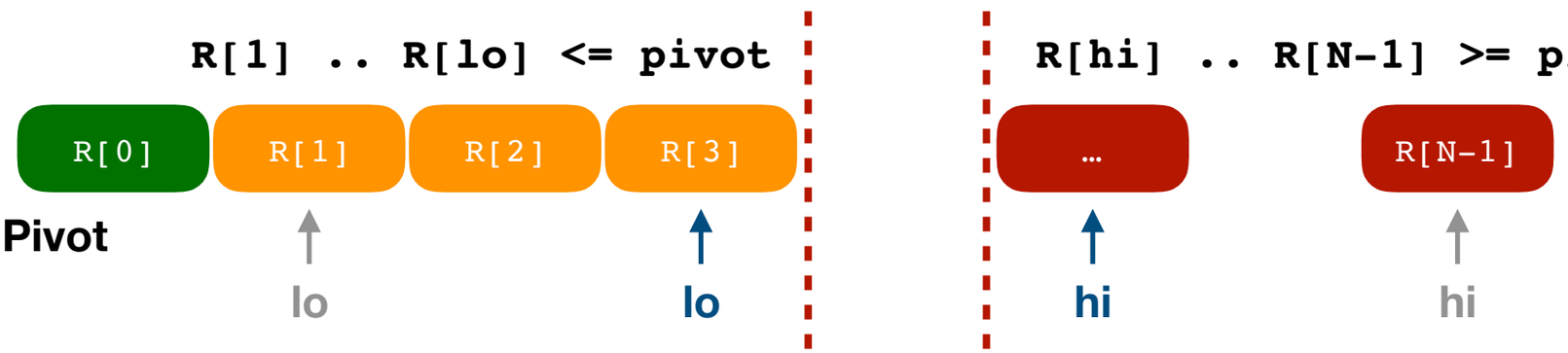
2. 向右移动 start，直到：

1) 找到第一个大于 Pivot 的 R[start]
OR
2) lo >= N

3. 向左移动 hi，直到：

1) 找到第一个小于 Pivot 的 R[hi]
OR
2) hi < 0

- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 Pivot Point**



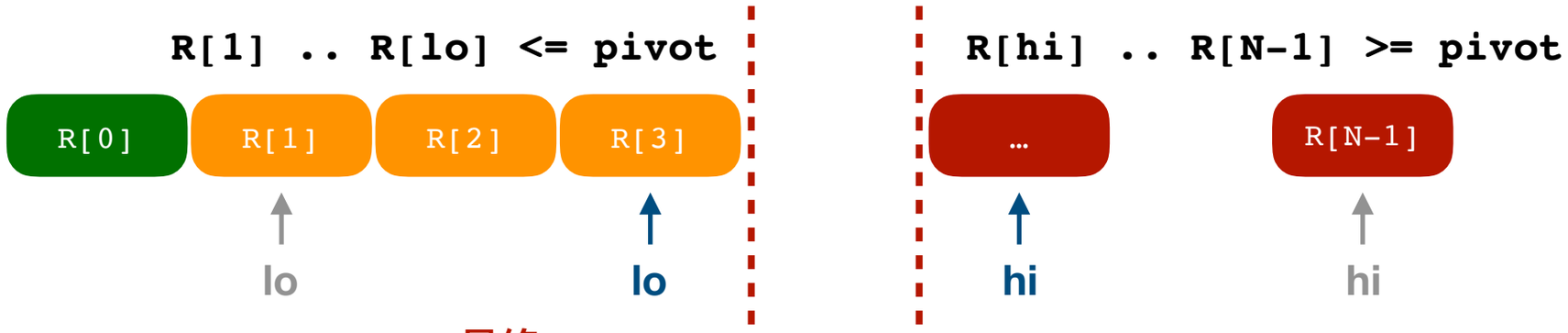
1. 选择 R[0] 作为 Pivot

2. 向右移动 start，直到：
   1) 找到第一个大于 Pivot 的 R[start]
      OR
   2) lo >= N

3. 向左移动 hi，直到：
   1) 找到第一个小于 Pivot 的 R[hi]
      OR
   2) hi < 0

4. 如果 lo < hi，那么交换 R[lo] 和 R[hi]

- 内部排序
  - 快速排序



5. 最终

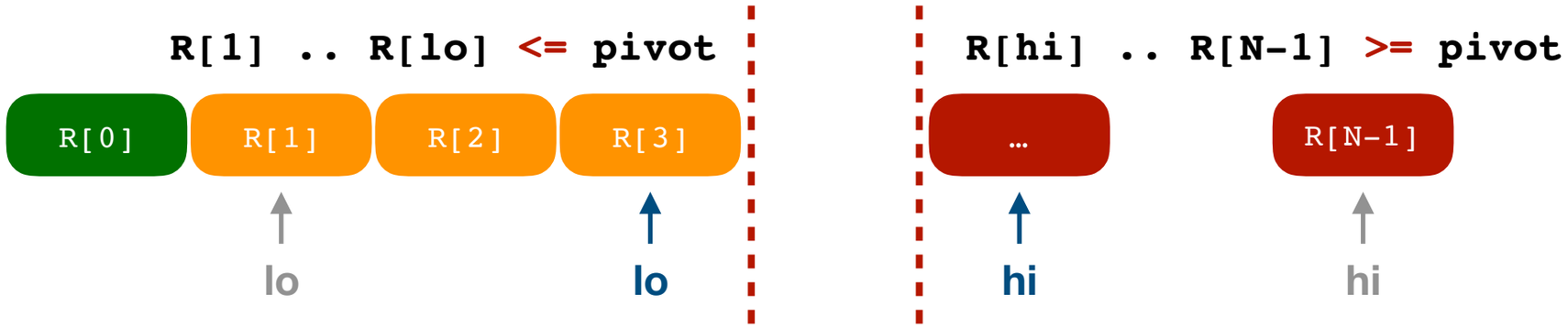
lo >= hi

R[1] .. R[lo-1] <= pivot 且 R[lo] > pivot 或 lo >= N

R[hi+1] .. R[N-1] >= pivot 且 R[hi] < pivot 或 hi < 0

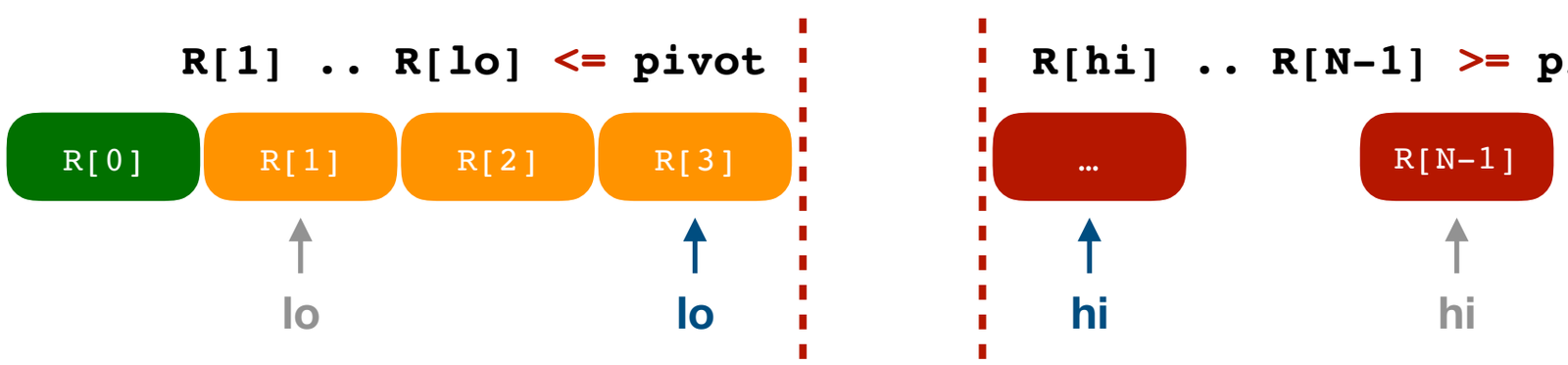
```cpp
template<typename T>
int partition(std::vector<T>& array, int start, int end) {
    int pivot_point = start;  // 1. R[0] 作为 Pivot

    int lo = start + 1;
    int hi = end - 1;

    while (true) {
        while (lo <= hi && array[lo] <= array[pivot_point])
            ++lo; // 2. 向右移动 lo，直到 R[lo] > Pivot
        while (hi >= lo && array[pivot_point] <= array[hi])
            --hi; // 3. 向左移动 hi，直到 R[hi] < Pivot

        if (lo >= hi) {
            break;
        }

        // 4. 若 lo < hi，则交换 R[lo] 和 R[hi]，同时向右移动 lo，向左移动 hi
        std::swap(array[lo++], array[hi--]);
    } // assert lo >= hi

    // 5. 交换 Pivot（R[0]）与 R[Pivot-Point]
    std::swap(array[pivot_point], array[hi]);
    return hi;
}
```

- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 `Pivot Point`**



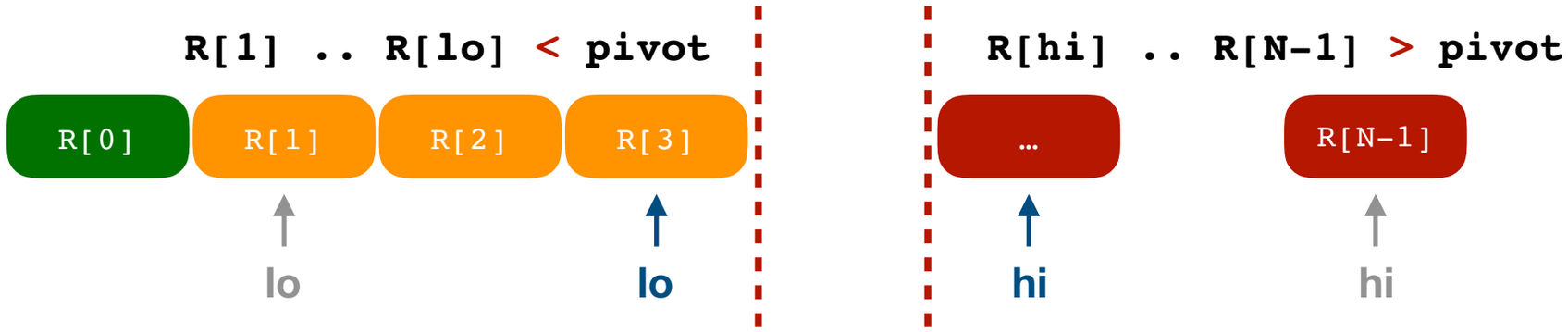
- 算法特点：单侧快速移动，减少交换
- 考虑一种极端情况：数组中大量（甚至全部）重复元素
  - `Pivot Point` 总是接近 `N-1`
  - `Partition` 后，递归子序列的长度接近 `N-1`
  - 分治思想下，二分递归退化为线性递归
  - 递归深度 `O(N)`，每次 `Partition` 的时间复杂度 `O(N)`，总 `O(N^2)`

- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 Pivot Point**



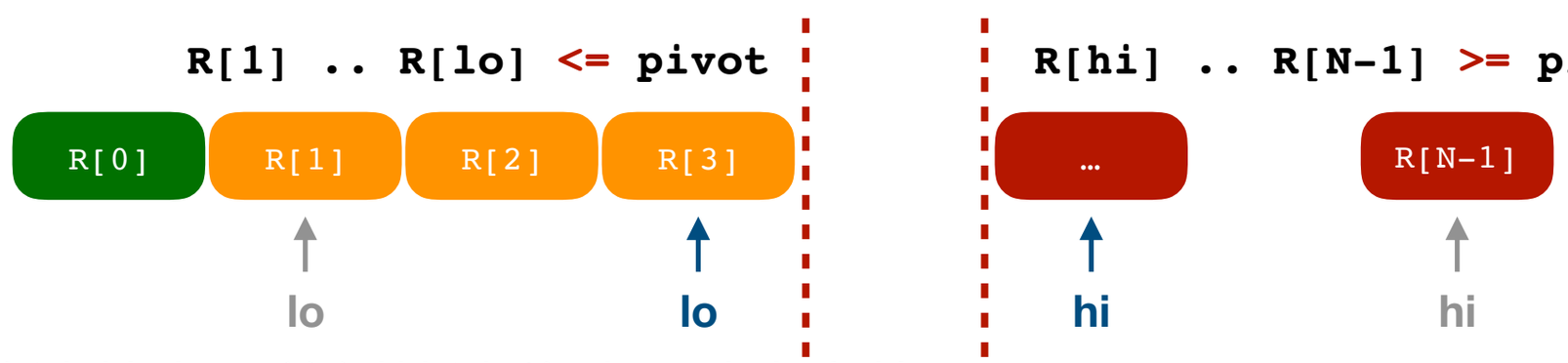
- 算法特点：单侧快速移动，减少交换
- 考虑一种极端情况：数组中大量（甚至全部）重复元素
  - 改进：两侧缓慢移动，增加交换

- 内部排序
  - 快速排序



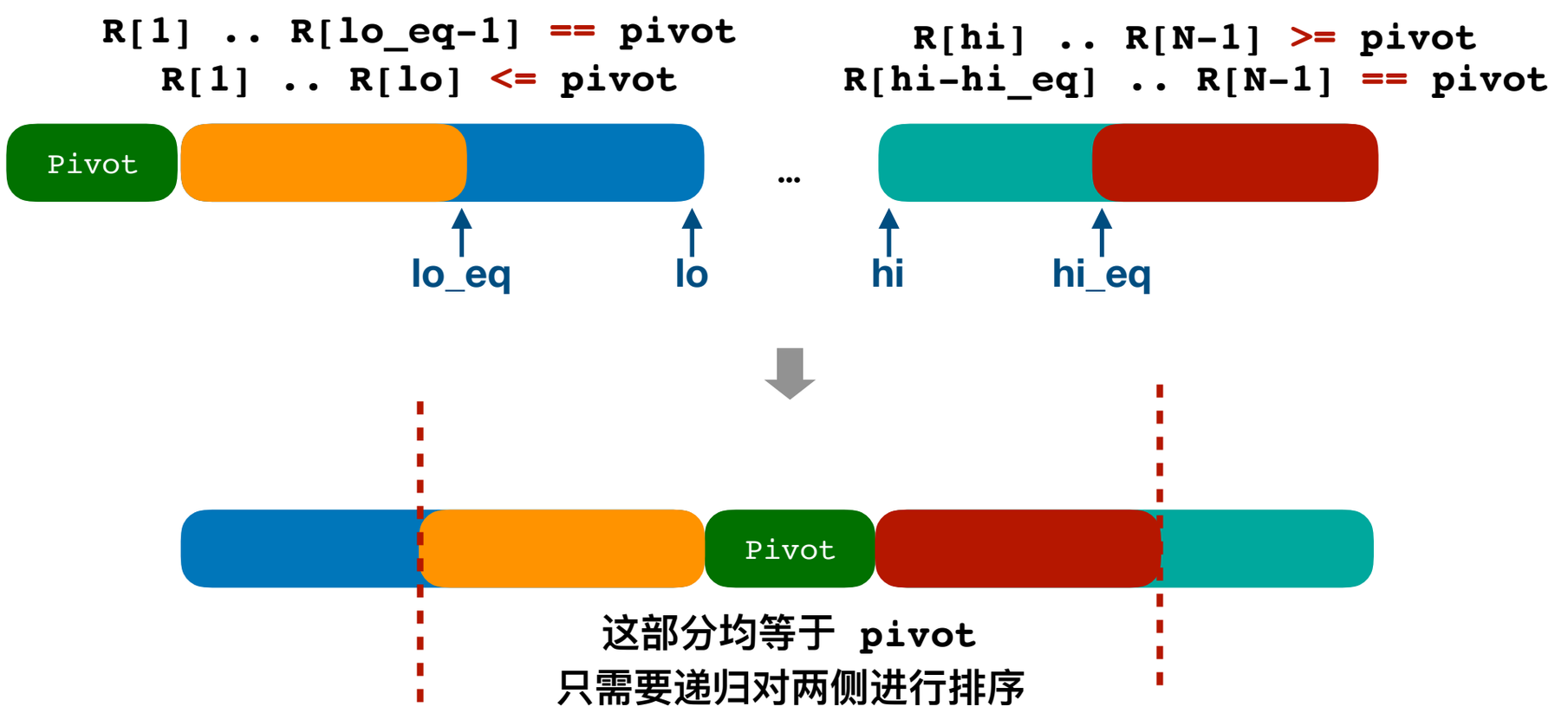

```
template<typename T>
int partition(std::vector<T>& array, int start, int end) {
    int pivot_point = start;  // 1. R[0] 作为 Pivot

    int lo = start + 1;
    int hi = end - 1;

    while (true) {
        while (lo <= hi && array[lo] < array[pivot_point])
            ++lo; // 2. 向右移动 lo，直到 R[lo] >= Pivot
        while (hi >= lo && array[pivot_point] < array[hi])
            --hi; // 3. 向左移动 hi，直到 R[hi] <= Pivot

        if (lo >= hi) {
            break;
        }

        // 4. 若 lo < hi，则交换 R[lo] 和 R[hi]，同时向右移动 lo，向左移动 hi
        std::swap(array[lo++], array[hi--]);
    } // assert lo >= hi

    // 5. 交换 Pivot（R[0]）与 R[Pivot-Point]
    std::swap(array[pivot_point], array[hi]);
    return hi;
}
```

- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 Pivot Point**



    - 算法特点：单侧快速移动，减少交换
    - 考虑一种极端情况：数组中大量（甚至全部）重复元素
      - 缺点：增加操作增加，一般情况下可能速度更慢
      - 改进：记录两侧与 Pivot 相等的元素，利用它们尽量保持 Pivot 在中间

- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 `Pivot Point`**
    - 算法特点：单侧快速移动，减少交换
    - 考虑一种极端情况：数组中大量（甚至全部）重复元素
      - 缺点：增加操作增加，一般情况下可能速度更慢
    - 改进：记录两侧与 `Pivot` 相等的元素，利用它们尽量保持 `Pivot` 在中间



- 内部排序
  - 快速排序

```
template<typename T>
std::pair<int, int> partition_opt(std::vector<T>& array, int start, int end) {
    int pivot_point = start;

    int lo = start + 1, lo_eq = start + 1;
    int hi = end - 1, hi_eq = end - 1;

    while (true) {
        while (lo <= hi && array[lo] <= array[pivot_point]) {
            if (array[lo] == array[pivot_point]) {
                std::swap(array[lo_eq], array[lo]);
                lo_eq++;
            }
            lo++;
        }
        while (hi >= lo && array[pivot_point] <= array[hi]) {
            if (array[pivot_point] == array[hi]) {
                std::swap(array[hi], array[hi_eq]);
                hi_eq--;
            }
            hi--;
        }

        if (lo >= hi) {
            break;
        }

        std::swap(array[lo++], array[hi--]);
    } // assert lo >= hi

    std::swap(array[pivot_point], array[hi]);

    int lo_eq_count = lo_eq - (start + 1);
    int hi_eq_count = end - 1 - hi_eq;

    std::swap_ranges(array.begin() + start + 1, array.begin() + start + 1 + lo_eq_count, array.begin() + hi - lo_eq_count);
    std::swap_ranges(array.begin() + hi + 1, array.begin() + hi + 1 + hi_eq_count, array.begin() + end - hi_eq_count);

    return std::make_pair(hi - lo_eq_count, hi + hi_eq_count);
}
```

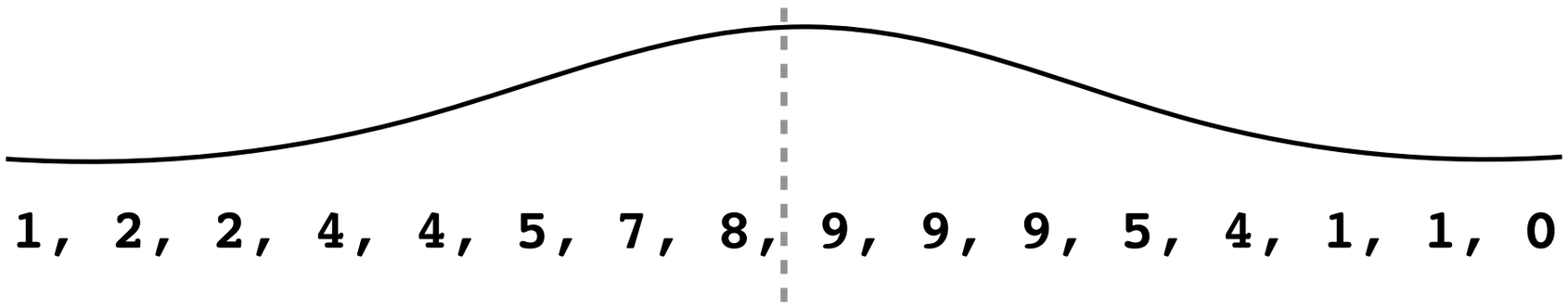
- 内部排序
  - 快速排序
    - 按 Pivot Point 将数据分开，使得：
      - `max( [0, pivot) ) <= min( (pivot, N) )`
    - 本质：**将数组每一个元素，逐个变成 Pivot Point**
    - 算法特点：单侧快速移动，减少交换
    - 考虑一种极端情况：数组中大量（甚至全部）重复元素
      - 改进：记录两侧与 Pivot 相等的元素，利用它们尽量保持 Pivot 在中间
  - 终极版本：学习 Redis 源代码，**pqsort.c**
    - 再改进：取 R[0]、R[N-1]、R[N/2] 三者的中位数作为 Pivot
    - 再再改进：当 N 小于一个常数时，使用冒泡排序、选择排序等简单排序算法
    - 再再再改进：尾递归 或者 无递归

- 内部排序
  - 其他快速排序的方法：
    - 优先队列: push() \ pop() \ top()
    - 二叉查找树: insert() \ delete() \ find_kth() \ find_greater() \ find_less()..

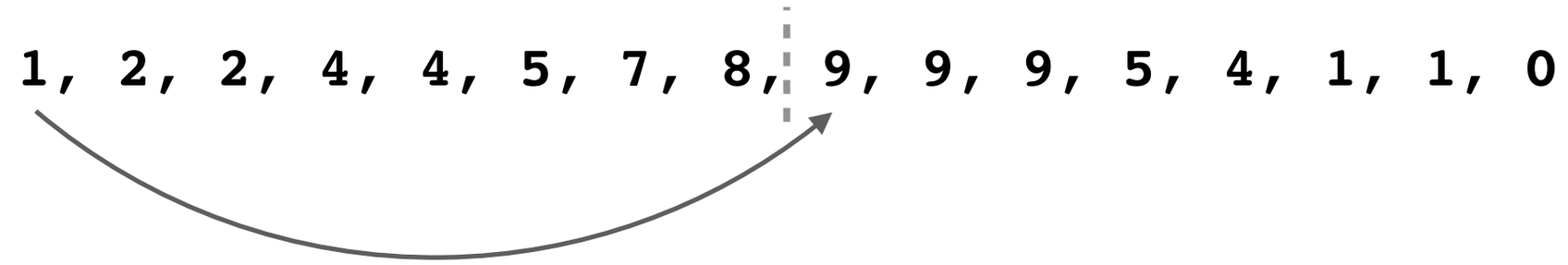
- 内部排序
  - 常见问题
    - 分治思想
      - LeetCode 33. <Search in Rotated Sorted Array>
      - LeetCode 240. <Search a 2D Matrix II>
      - LeetCode 327. <Count of Range Sum>
    - 查找
      - 中位数 / 众数
        - LeetCode 4. <Median of Two Sorted Arrays>
        - LeetCode 169. <Majority Element>
        - LeetCode 229. <Majority Element II>
      - 第 K 大的数
        - LeetCode 215. <Kth Largest Element in an Array>
        - LeetCode 703. <Kth Largest Element in a Stream>
  - 去重
    - LeetCode 56. <Merge Intervals>
  - 排序
    - LeetCode 136. Single Number（排序不是最优解）
    - LeetCode 164. <Maximum Gap>
    - LeetCode 870. <Advantage Shuffle>
    - LeetCode 1122. <Relative Sort Array>

目录

- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - 双调序列
      - 双调序列是一个先单调递增后单调递减（或者先单调递减后单调递增）的序列



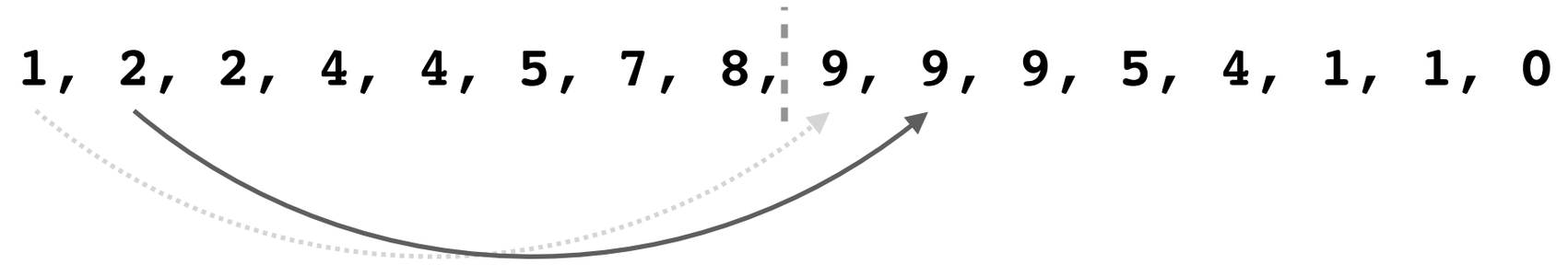
- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - Batcher定理
      - 将任意一个长为 2N 的双调序列 R 分为等长的两半 X 和 Y
      - 将 X 中的元素与 Y 中的元素按顺序一一比较，即 R[i] 与 R[i+N] 比较，将较大者放入 MAX 序列，较小者放入 MIN 序列：
        - MAX 和 MIN 序列仍然是双调序列
        - MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素



MIN 1

MAX 9

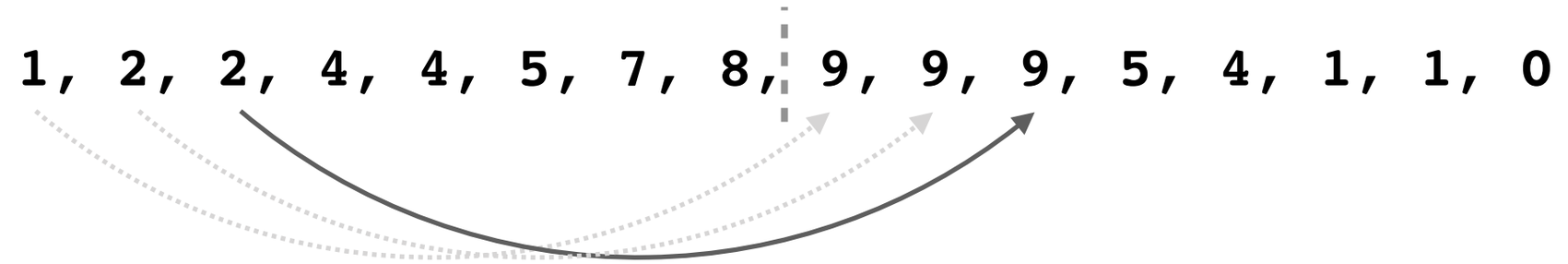
- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - Batcher定理
      - 将任意一个长为 2N 的双调序列 R 分为等长的两半 X 和 Y
      - 将 X 中的元素与 Y 中的元素按顺序一一比较，即 R[i] 与 R[i+N] 比较，将较大者放入 MAX 序列，较小者放入 MIN 序列：
        - MAX 和 MIN 序列仍然是双调序列
        - MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素



MIN 1, 2

MAX 9, 9

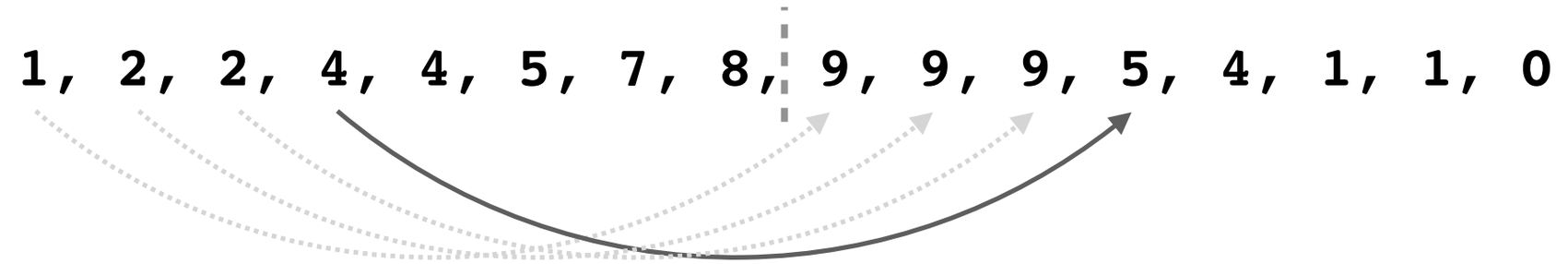
- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - Batcher定理
      - 将任意一个长为 2N 的双调序列 R 分为等长的两半 X 和 Y
      - 将 X 中的元素与 Y 中的元素按顺序一一比较，即 R[i] 与 R[i+N] 比较，将较大者放入 MAX 序列，较小者放入 MIN 序列：
        - MAX 和 MIN 序列仍然是双调序列
        - MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素



MIN 1, 2, 2

MAX 9, 9, 9

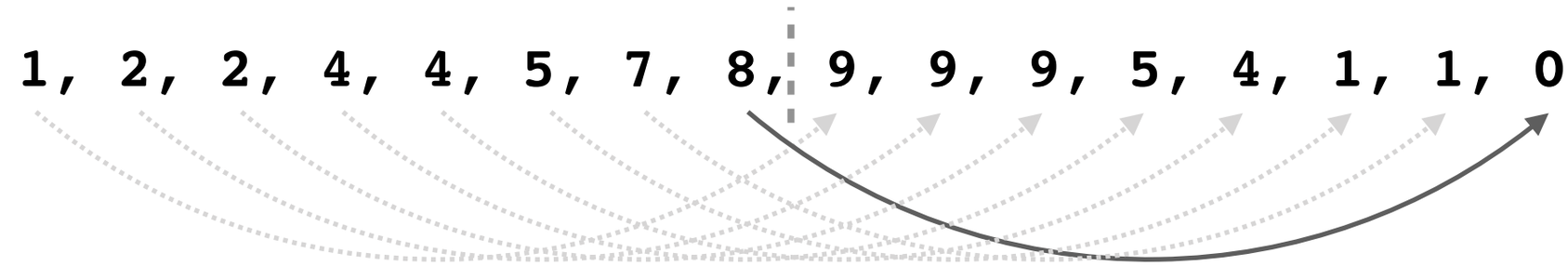
- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - Batcher定理
      - 将任意一个长为 2N 的双调序列 R 分为等长的两半 X 和 Y
      - 将 X 中的元素与 Y 中的元素按顺序一一比较，即 R[i] 与 R[i+N] 比较，将较大者放入 MAX 序列，较小者放入 MIN 序列：
        - MAX 和 MIN 序列仍然是双调序列
        - MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素



MIN 1, 2, 2, 4

MAX 9, 9, 9, 5

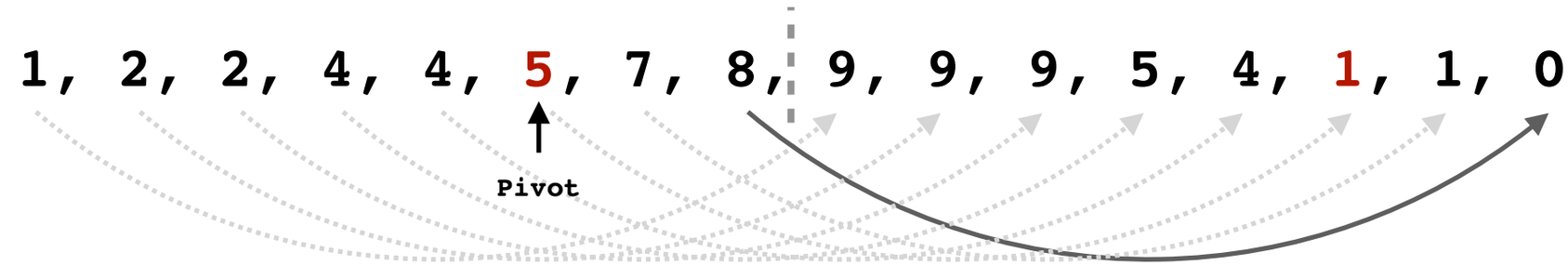
- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - Batcher定理
      - 将任意一个长为 2N 的双调序列 R 分为等长的两半 X 和 Y
      - 将 X 中的元素与 Y 中的元素按顺序一一比较，即 R[i] 与 R[i+N] 比较，将较大者放入 MAX 序列，较小者放入 MIN 序列：
        - MAX 和 MIN 序列仍然是双调序列
        - MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素



MIN 1, 2, 2, 4, 4, 1, 1, 0

MAX 9, 9, 9, 5, 4, 5, 7, 8

- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - Batcher定理
      - 将任意一个长为 2N 的双调序列 R 分为等长的两半 X 和 Y
      - 将 X 中的元素与 Y 中的元素按顺序一一比较，即 R[i] 与 R[i+N] 比较，将较大者放入 MAX 序列，较小者放入 MIN 序列：
        - MAX 和 MIN 序列仍然是双调序列
        - MAX 序列中的任意一个元素不小于 MIN 序列中的任意一个元素

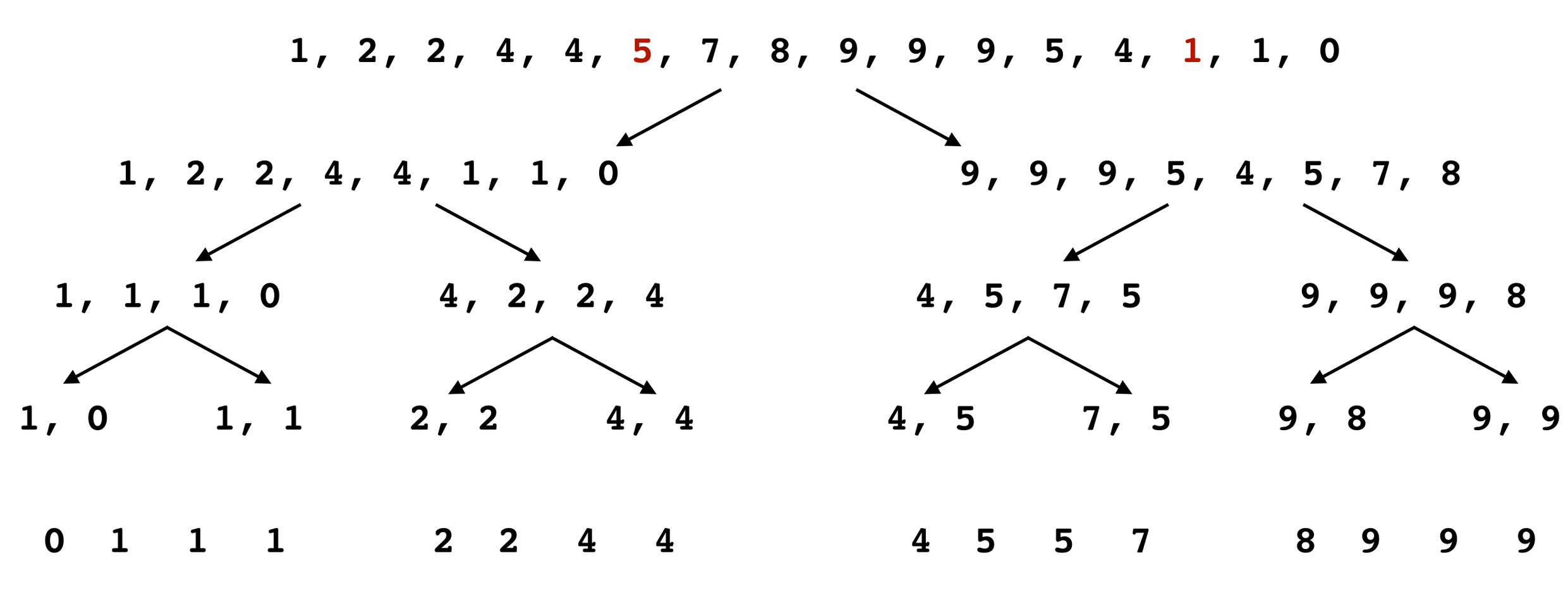


MIN 1, 2, 2, 4, 4, 1, 1, 0

MAX 9, 9, 9, 5, 4, 5, 7, 8

本质：找到了一个 Pivot Point，对先递增后递减的双调序列：
Pivot Point = min{ Pivot | R[Pivot] > R[N + Pivot]}
MIN = [[R[0] .. R[Pivot-1]]] + [R[N + Pivot] .. R[2N-1]]
MAX = [[R[N] .. R[N + Pivot-1]] + [R[Piovt] .. R[N]]
想想与 Quick Sort 的相似性?

- 并行化内部排序
  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - 双调排序
      - Batcher 定理，将长度为 N 的双调划分成两个长度为 N/2 的双调序列 MIN 和 MAX：
        - 将 MIN 排在“前面”，MAX 排在 “后面”
      - 对每个双调子序列递归划分，直到得到的子序列长度为1为止



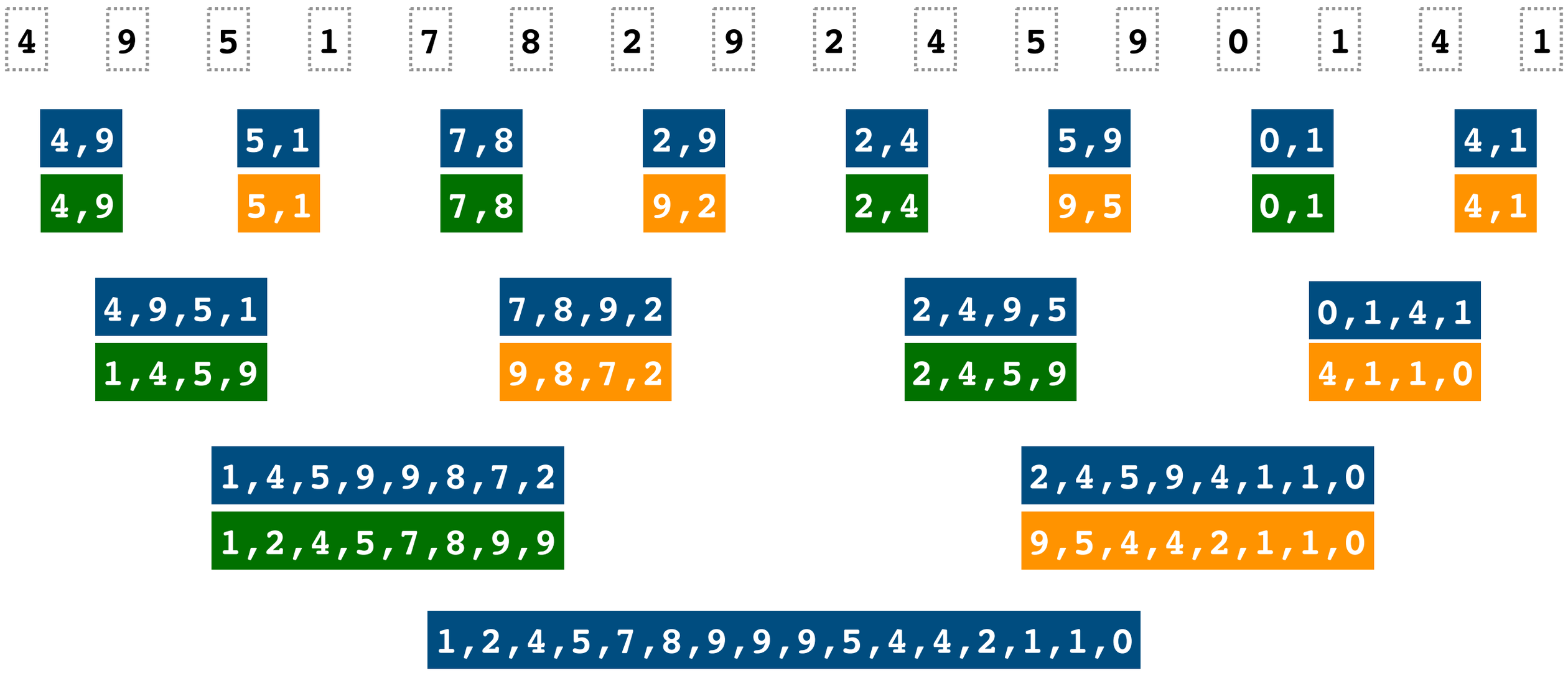
- 并行化内部排序

问题变成：如何将原始序列转换成双调序列

  - Data Independent Sort
    - 排序算法的执行过程与数据内容无关
  - 双调排序
    - 双调排序
      - 双调排序
        - Batcher 定理，将长度为 N 的双调划分成两个长度为 N/2 的双调序列
          MIN 和 MAX：
          - 将 MIN 排在“前面”，MAX 排在 “后面”
      - 对每个双调子序列递归划分，直到得到的子序列长度为1为止
- 优势：
  - 分治思想，按序比较
    - 宏观上，天然适合并行化
    - 微观上，通过 SSE、AVX 等 CPU SIMD 指令，或 GPU 的比较算子进行加速

- 并行化内部排序
  - 双调排序
    - 生成双调序列
      - 分治思想：
        - 双调排序的过程，是根据 Pivot 去 “分” 的过程
        - 生成双调序列，是先 “合”，再 “分”
- “合”：两个相邻单调子序列合并，生成一个双调子序列
- “分”：一个双调排序，将双调子序列变成单调子序列

- 并行化内部排序
  - 双调排序
    - 生成双调序列
      - “合”：两个相邻单调子序列合并，生成一个双调子序列
      - “分”：一个双调排序，将双调子序列变成（递增、递减）单调子序列



- 并行化内部排序
  - 双调排序
    - 最后一个问题：
      - 非2的幂次长度序列排序
        - 双调排序算法能对长度为2的幂的序列进行排序
        - 对于任意长度的序列，需要 Padding，让数组的大小填充到2的幂长度，进行排序