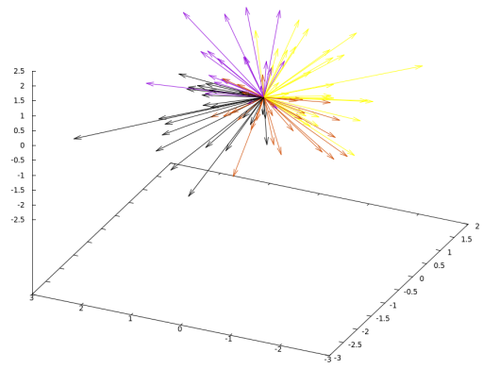
# 数据结构实验（1）

## 绪论与线性表

# 目录

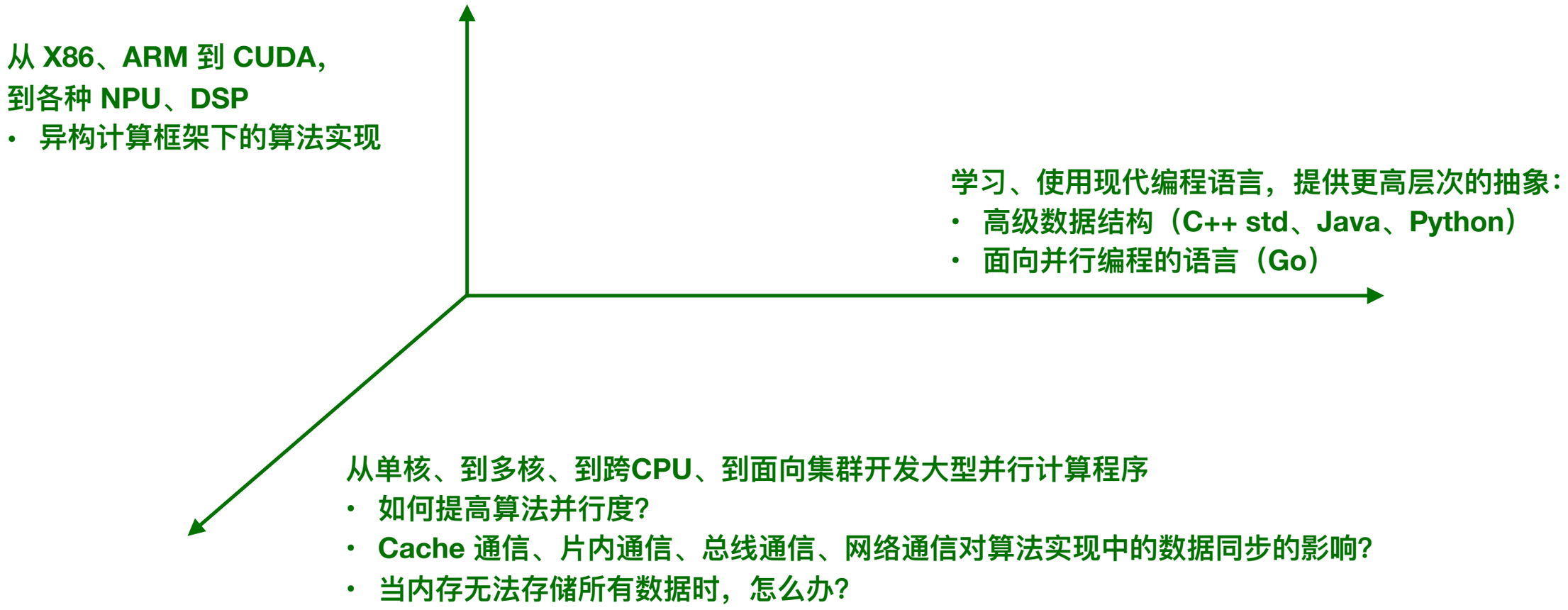
- 数据结构与算法是一门重要的课程
  - 学好这门课：
    - 数据结构与算法是整个计算机学科的奠基石


**现代文件系统在 磁盘=>SSD 的优化**



**K邻近问题与LSH**

- 数据结构与算法是一门重要的课程
  - 在一学期的课程里我们能学到什么：
    - 针对不同的问题规模，学会用不同的数据结构与算法，在确定的时间内解决问题



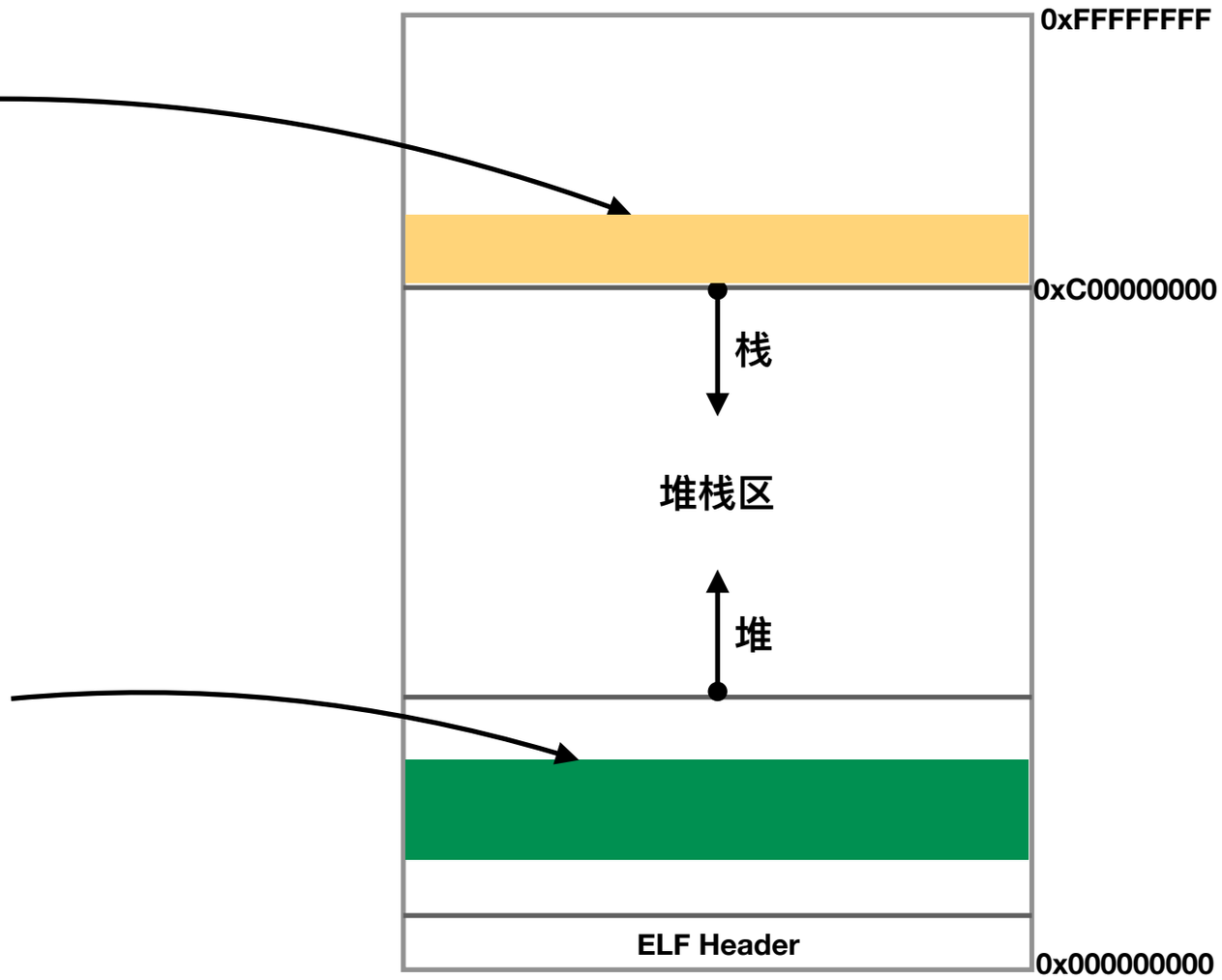
# 线性表

- 线性表的几种物理实现
- 两种经典计算机语言——递归思想、分治思想在线性表中的应用
- 线性表的典型应用：Bitmap

- 线性表的几种物理实现
  - C/C++ 数组
  - C++ std::vector
  - C++ std::list
  - C/C++ 自定义线性表
  - C/C++ 自定义链表
  - …

**什么时候该用哪个呢?**

- 线性表的几种物理实现
  - 按内存分配方式划分（1）—— 在哪个区域分配?
    - C/C++ 数组
    - C++ std::vector
    - C++ std::list
    - C/C++ 自定义线性表
    - C/C++ 自定义链表

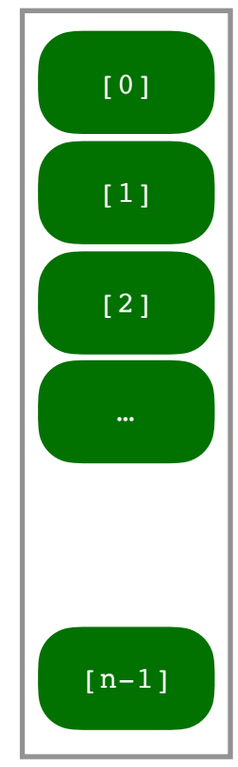


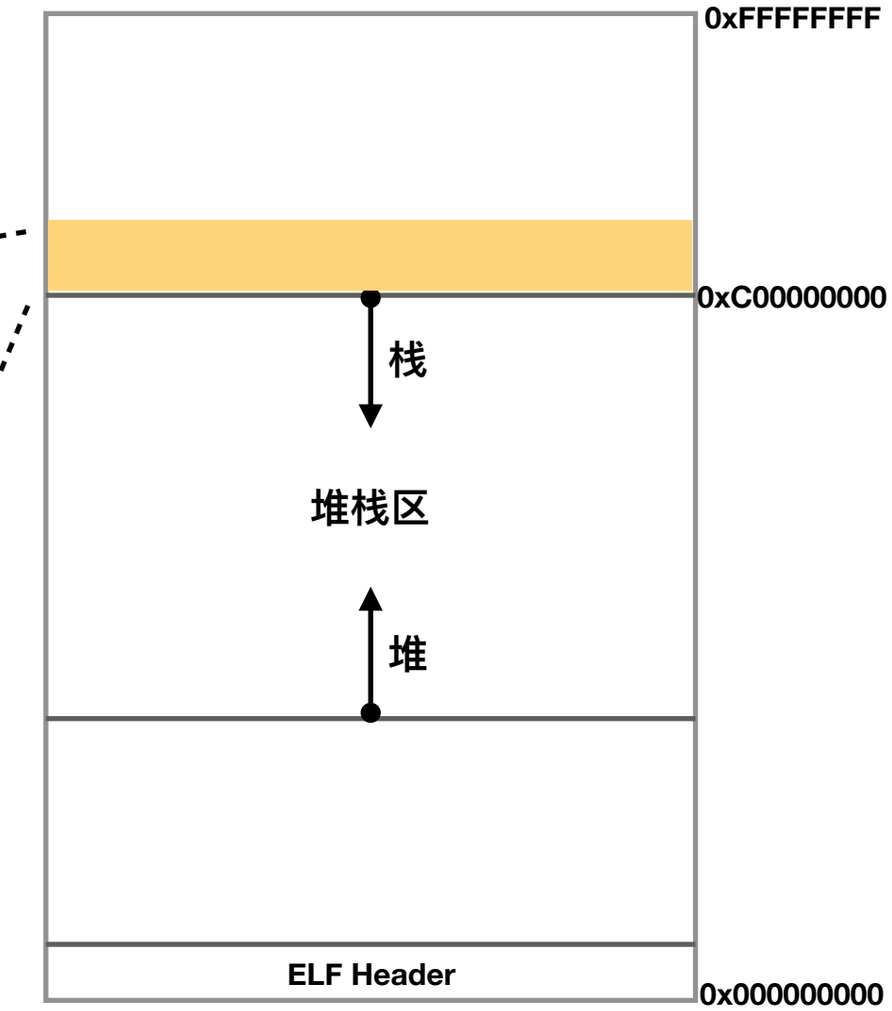
C/C++ 程序虚拟地址空间

问题：
C/C++ 程序执行时有哪些空间?
栈空间和堆空间的区别是什么?

- 线性表的几种物理实现
  - 按内存分配方式划分（2）——用什么方式管理内存?
    - C/C++ 数组

int A[n]




C/C++ 程序虚拟地址空间

- **数组大小固定**
- 元素通过 Index **快速**访问
- 存储在栈空间上连续地址，单个元素的长度为 K 字节，则第 i 个元素地址为 A + K*i

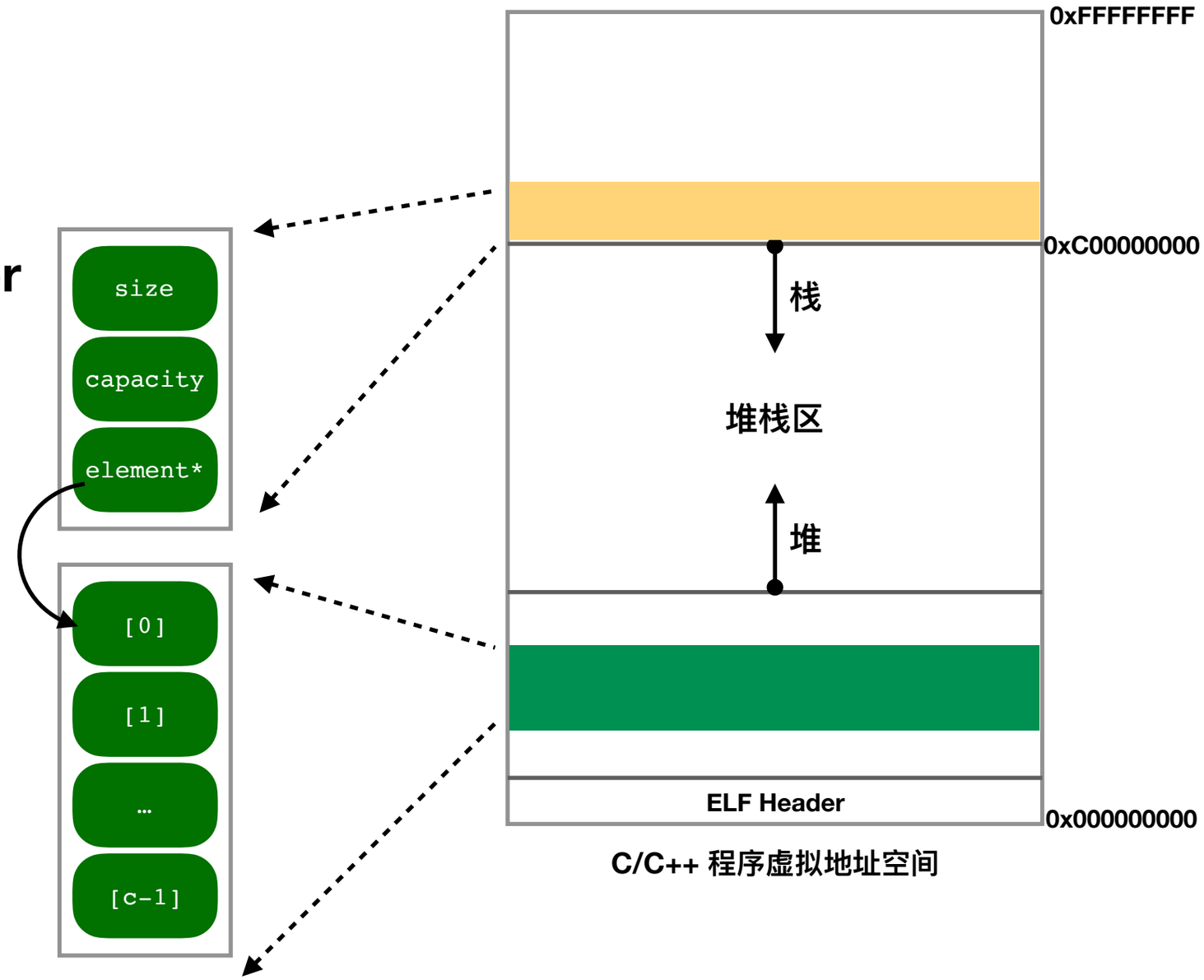
**问题：**

**struct XX a[n] 的地址空间?**

**栈空间通常有多大？分配效率？操作效率?**

**定义时还是使用时分配内存？定义时会初始化内存吗?**

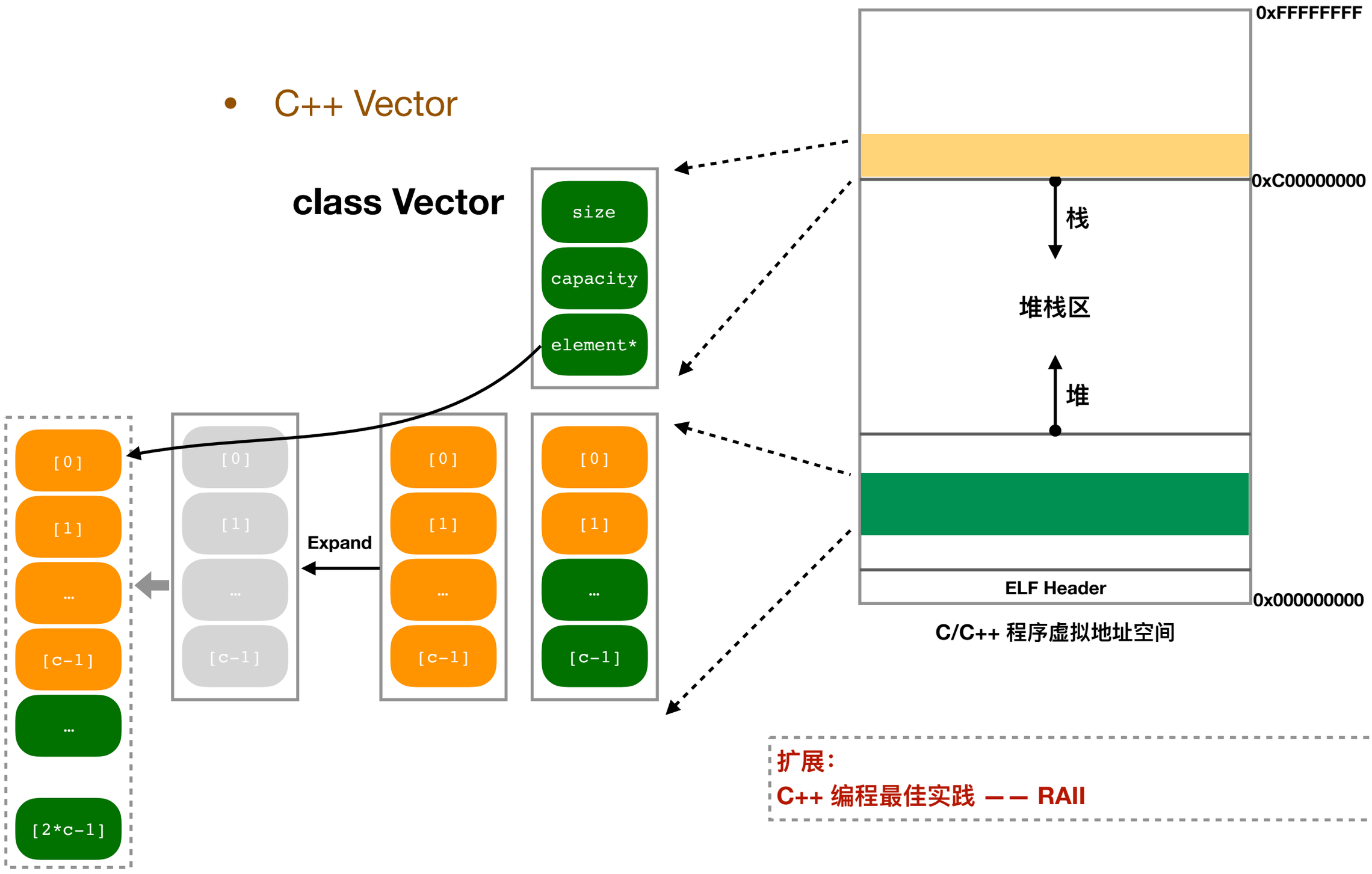
- 线性表的几种物理实现
  - 按内存分配方式划分（2）——用什么方式管理内存?
    - C++ Vector




- **Vector 大小可自增**
- 构造函数中申请堆空间，析构函数中释放空间
- 元素通过 Index **快速**访问
- 存储在堆空间上连续地址

**问题：**
**定义 std::vector<int> a, b; 那么 a=b 执行的是什么操作?**

- 线性表的几种物理实现
  - 按内存分配方式划分（2）——用什么方式管理内存?
    - C++ Vector




扩展:
C++ 编程最佳实践 —— RAII

- 穿插话题：C++ 编程最佳实践之 —— RAII

C/C++ 里，各种资源需要显示申请、释放，包括堆上的内存空间、数据库的连接、各种锁、文件句柄 等等。遗忘释放资源，或因为程序进入异常分支导致资源未释放，将造成各种严重的后果

```cpp
int main(int, char**) {

    // 普通申请堆上内存空间的方法
    int *a = new int[11];

    // Do something with a...

    // 易遗忘，或因 Bug 导致未执行释放
    delete[] a;

    return 0;
}
```

C/C++ 里，栈上构造的对象（离开作用域后）最终一定会被销毁，我们可以将资源封装，在构造函数中申请资源，在析构函数中释放资源。

```cpp
template<typename T>
struct Array {

private:
    T* elem;

public:

    Array(int n) {
        this->elem = new T[n];
    }
    ~Array() {
        delete[] this->elem;
    }

    T* get() {
        return this->elem;
    }

};

int main(int, char**) {
    // 通过 Array 对象，在堆空间创建线性表
    Array<int> arr(10);

    // Do somethign with arr.get() ...

    return 0;
}   // arr 会在离开作用域后自动调用析构函数，释放资源
```

- 线性表的几种物理实现
  - 按操作时间复杂度、空间复杂度划分

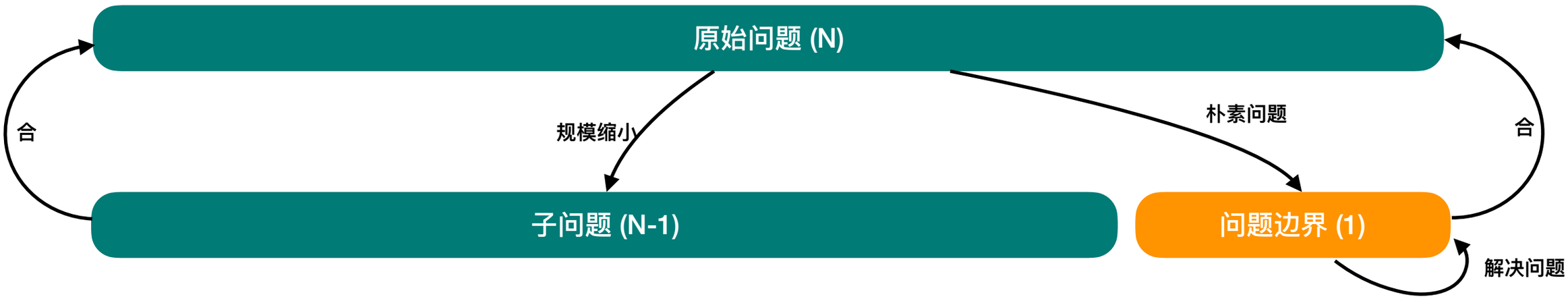
| | 插入 | 删除 | 随机访问 |
|---|---|---|---|
| 线性表 | | | |
| 链表 | | | |
| 队列 | | | |
| ... | | | |

  - 实际问题中，时间复杂度、空间复杂度 并不是绝对不变的

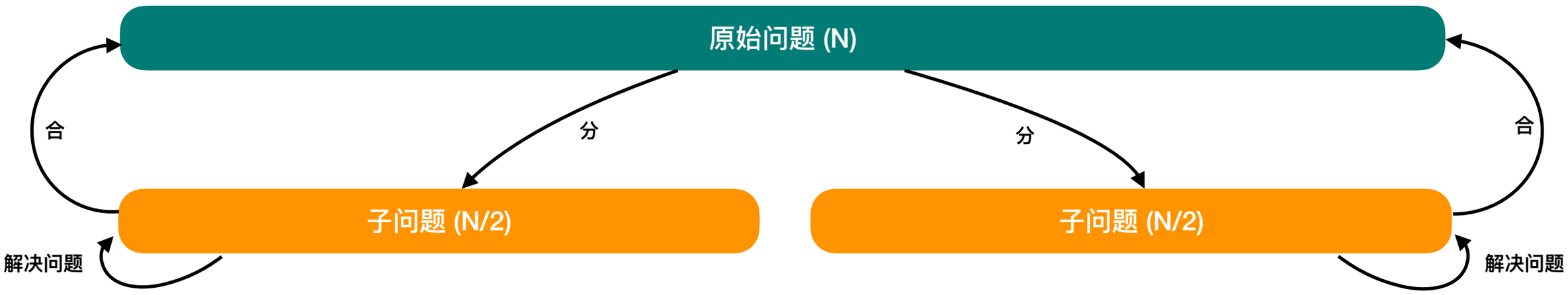
# 线性表

- 线性表的几种物理实现
- 两种经典计算机语言——递归思想、分治思想在线性表中的应用
- 线性表的典型应用：Bitmap

- 两种经典计算机语言
  - 为什么？—— 帮助记忆、理解线性表的常见操作，并学会解决新的问题
  - 递归思想

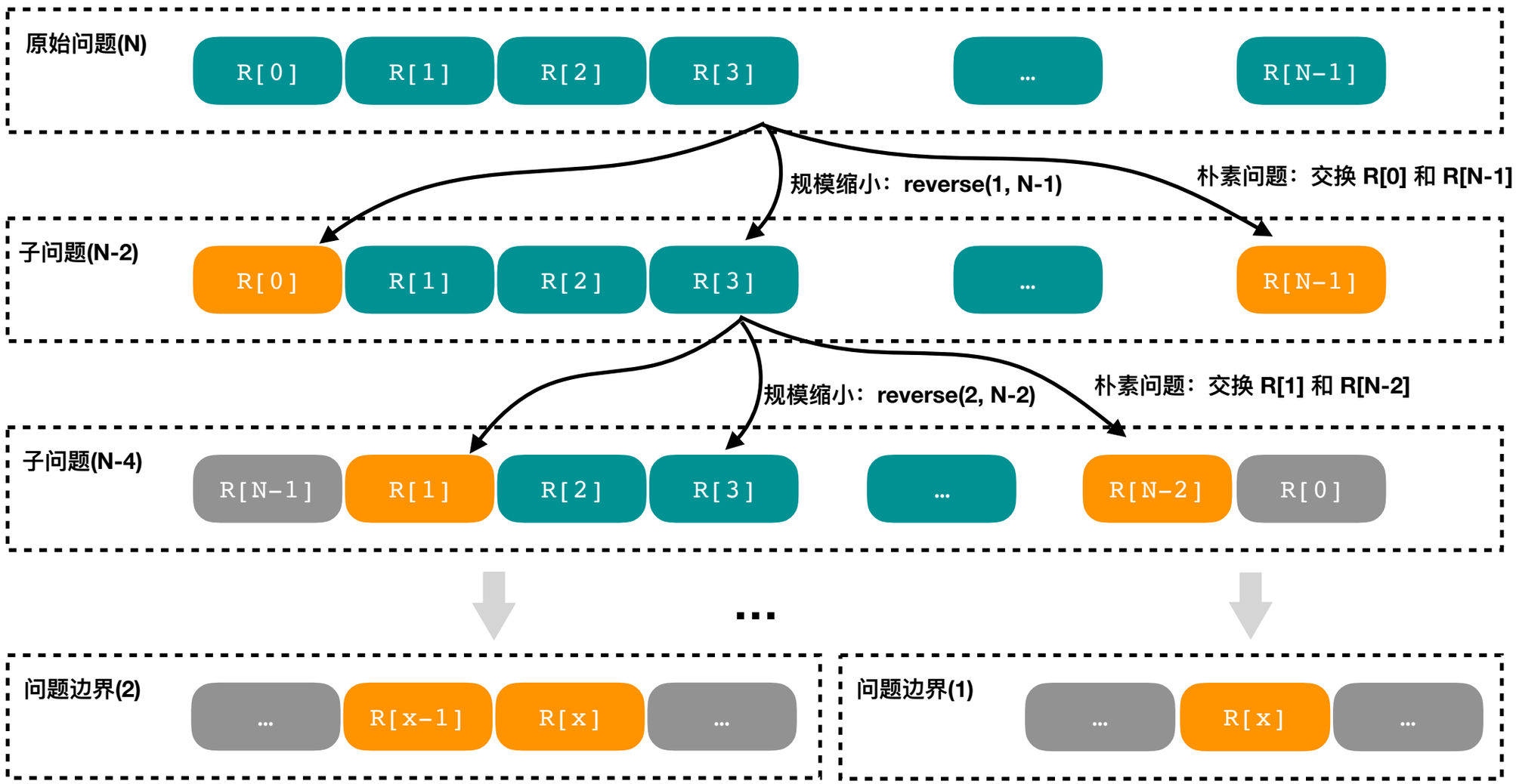



  - 分治思想




- 原始问题的规模缩小到一定的程度，就可以容易的解决；
- 原始问题可以分解为规模较小的相同问题（术语：具有最优子结构性质）；
- 可以通过合并子问题的解，得到原始问题的解。

- 两种经典计算机语言
  - 递归思想的应用（1）—— 翻转 vector<T>::reverse(0, N)

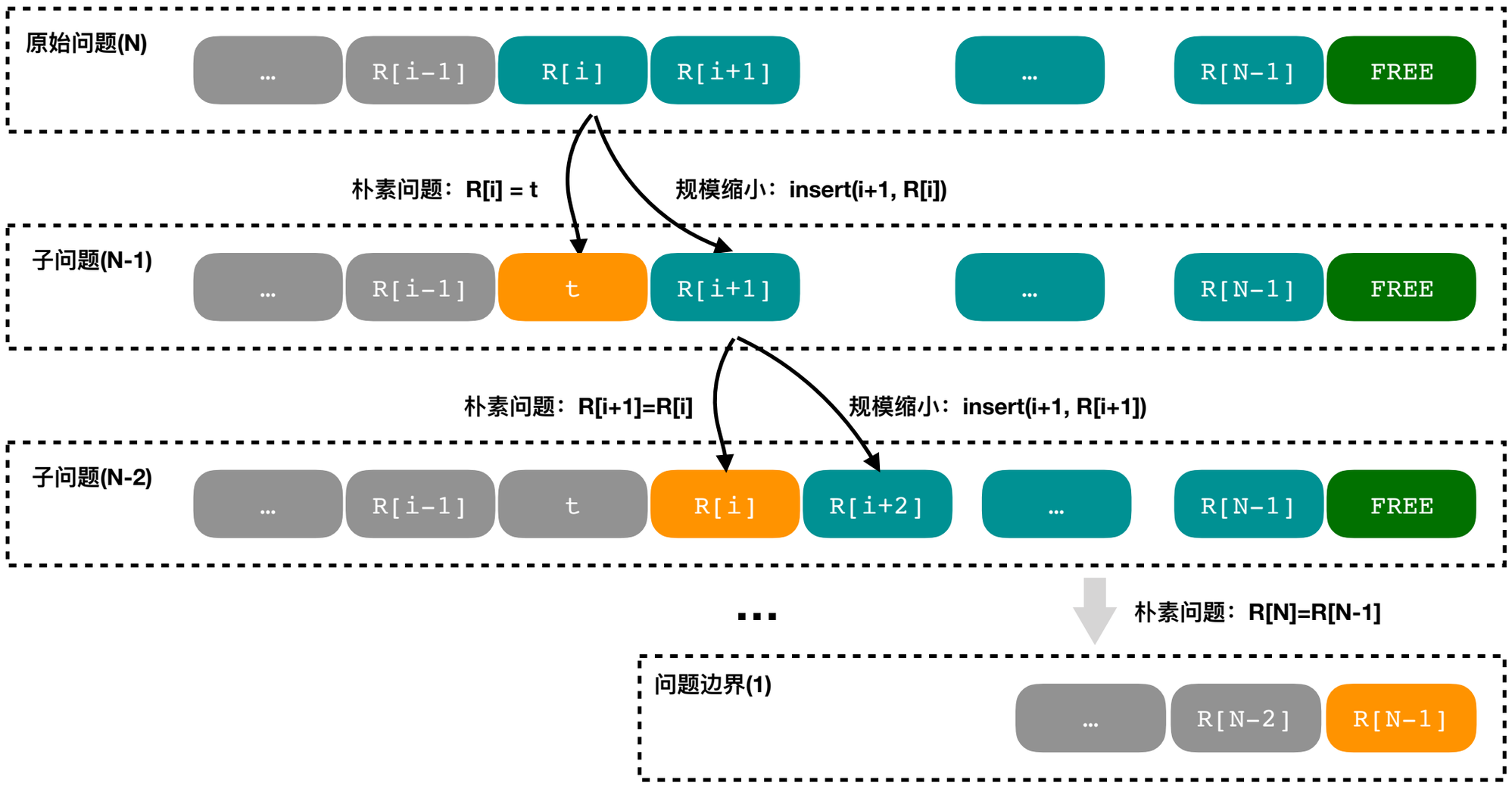


- 两种经典计算机语言
  - 递归思想的应用（1）—— 翻转 vector<T>::reverse(0, N)

```
// 线性表翻转算法
//
// @params r: 输入线性表
// @params start: 翻转的范围起点，闭区间
// @params end: 翻转的范围终点，开区间
//
template<typename T>
void reverse(T* r, int start, int end) {
    // 判断是否到达问题边界
    if (end - start <= 1) {
        // 到达问题边界，直接返回
    } else {
        // 朴素问题，交换线性表头尾元素
        T s = r[start];
        r[start] = r[end-1];
        r[end-1] = s;
        // 递归解决子问题(start+1, end-1)，翻转线性表中[start+1, end-1)的元素
        reverse(r, start + 1, end - 1);
    }
}
```

* 篇幅所限，课件中示例代码均只实现基本功能，不做下标越界、输入错误等合法性检查

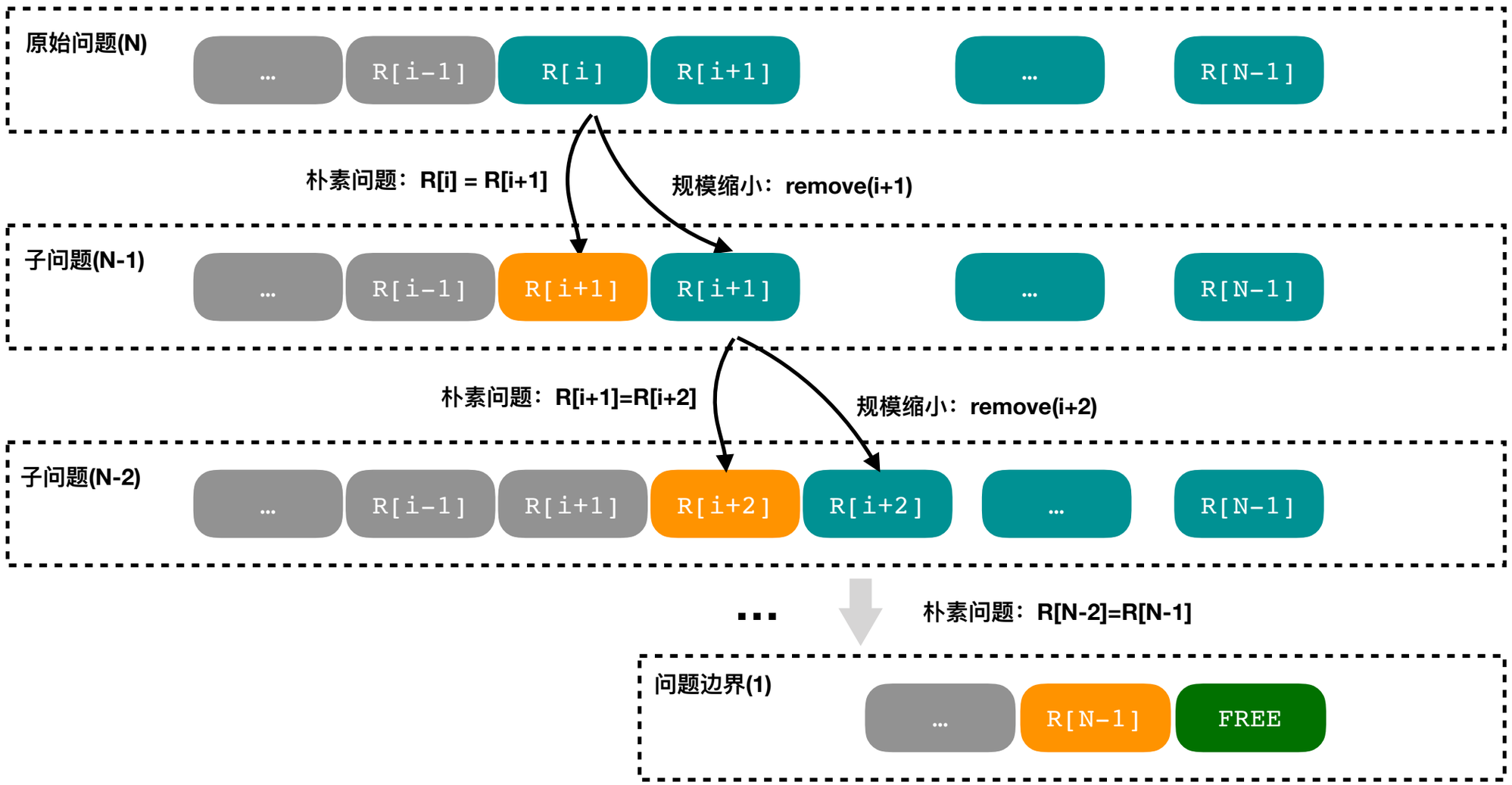
- 两种经典计算机语言
  - 递归思想的应用（2）—— 插入 vector<T>::insert(i, K)



- 两种经典计算机语言
  - 递归思想的应用（2）—— 插入 vector<T>::insert(i, K)

```
// 线性表插入算法
//
// @params r: 输入线性表
// @params size: 线性表的长度
// @params element: 待插入的元素
// @params pos: 元素插入的下标
//
template<typename T>
void insert(T* r, int size, const T& element, int pos) {
    // 判断是否到达问题边界
    if (pos == size) {
        // 到达问题边界，元素置0，并返回
    } else {
        // 递归解决子问题(r[pos], pos+1)，将当前 r[pos] 元素的值插入到下标 pos+1 处
        insert(r, size, r[pos], pos+1);
    }
    // 朴素问题，将元素 element 插入到下标 pos 处
    // 此处为何先解决子问题(r[pos], pos+1)，再解决朴素问题，将 element 赋值给 r[pos]?
    r[pos] = element;
}
```

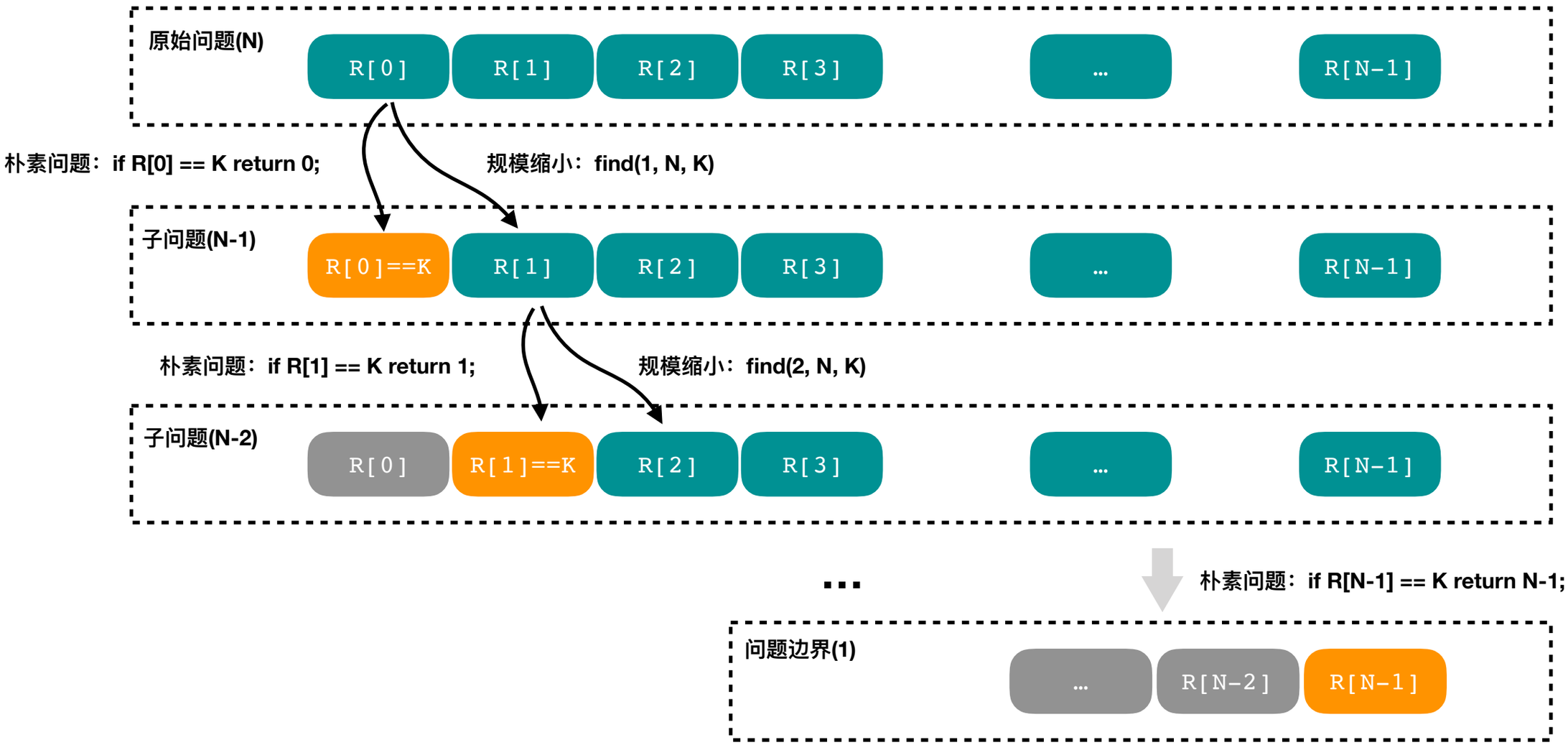
- 两种经典计算机语言
  - 递归思想的应用（3）—— 删除 vector<T>::remove(i)



- 两种经典计算机语言
  - 递归思想的应用（3）—— 删除 vector<T>::remove(i)

```
// 线性表删除算法
//
// @params r: 输入线性表
// @params size: 线性表的长度
// @params pos: 删除元素的下标
//
template<typename T>
void remove(T* r, int size, int pos) {
    // 判断是否到达问题边界
    if (pos == size-1) {
        // 到达问题边界，元素置0，并返回
        r[pos] = 0;
    } else {
        // 朴素问题，将下一个元素复制到当前下标
        r[pos] = r[pos+1];
        // 递归解决子问题(pos+1)，删除第 pos+1 位的元素
        remove(r, size, pos+1);
    }
}
```

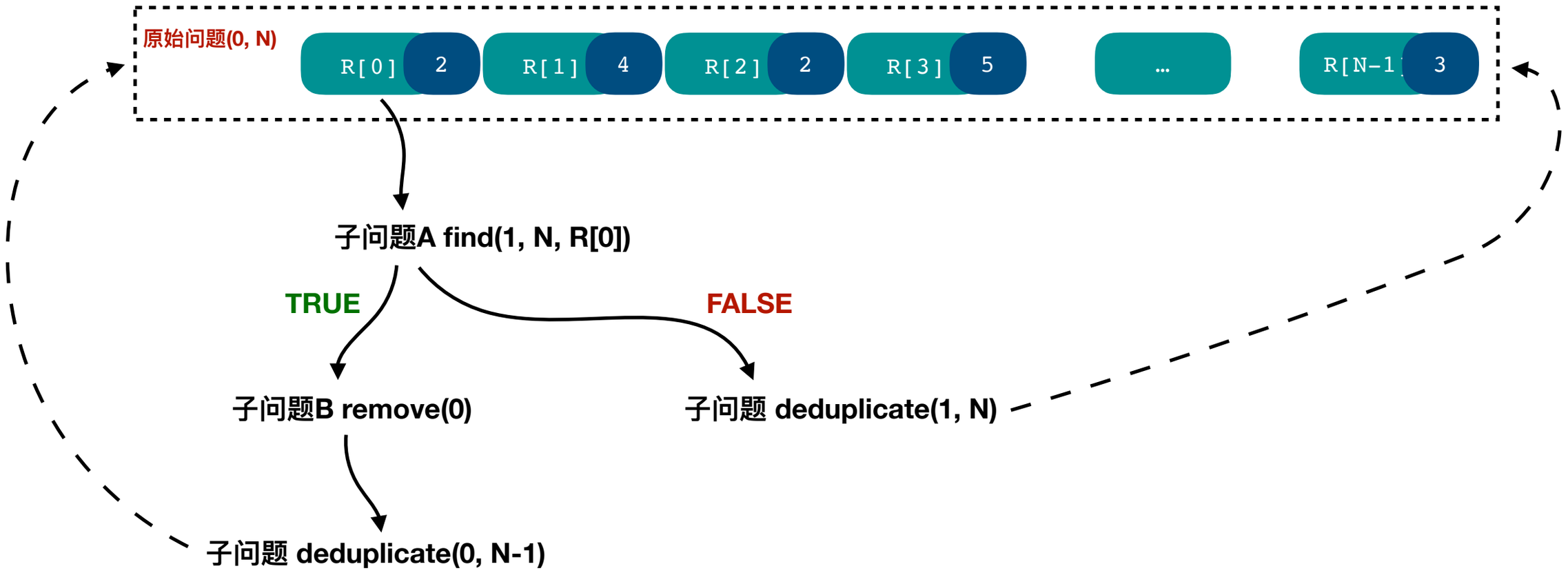
- 两种经典计算机语言
  - 递归思想的应用（4）—— 查找 vector<T>::find(0, N, K)



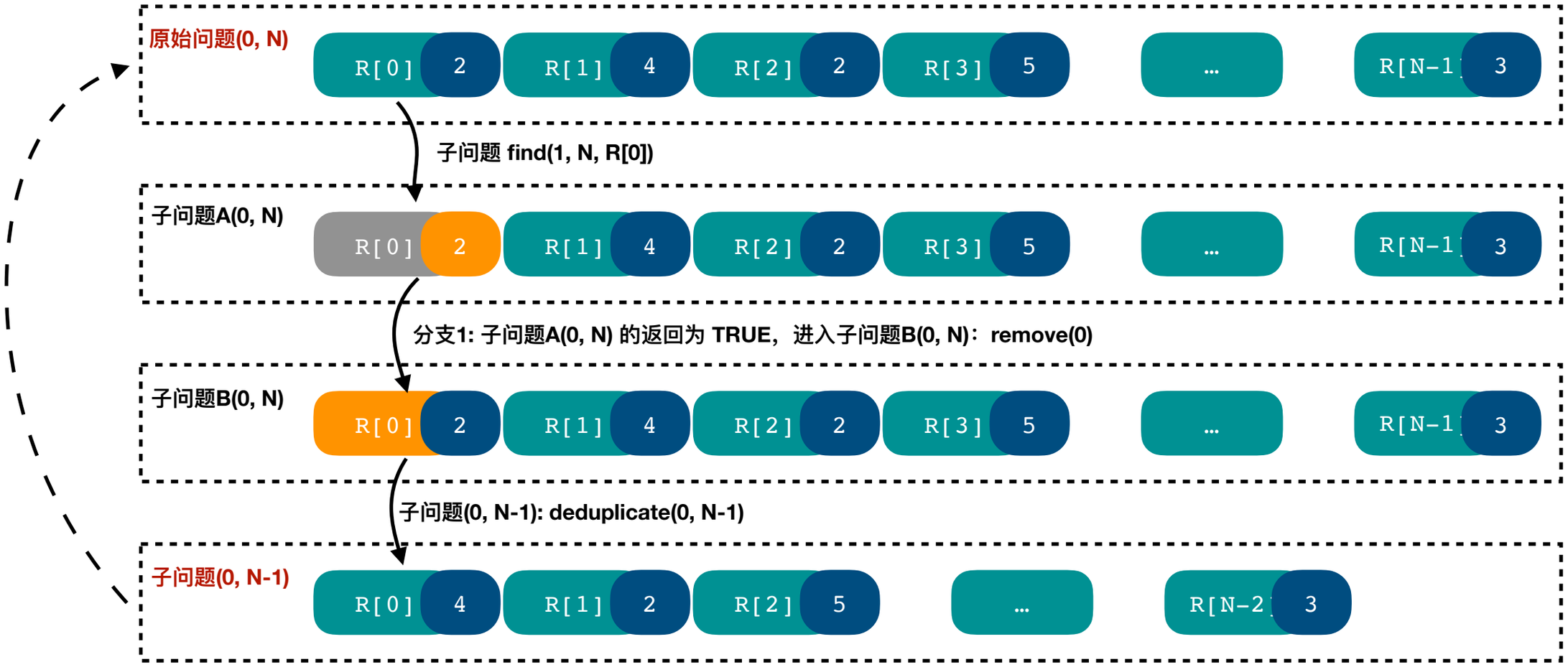
- 两种经典计算机语言
  - 递归思想的应用（4）—— 查找 vector<T>::find(0, N, K)

```
// 线性表查找算法
//
// @params r: 输入线性表
// @params size: 线性表的长度
// @params pos: 查找的元素
//
// @return 若元素存在，返回第一个等于该元素的下标；否则返回 -1
//
template<typename T>
int find(const T* r, int start, int size, const T& element) {
    // 判断是否到达问题边界
    if (start >= size) {
        // 到达问题边界，未找到该元素
        return -1;
    }
    // 判断是否找到该元素
    if (r[start] == element) {
        // 若找到该元素，则返回下标
        return start;
    } else {
        // 若未找到该元素，递归解决子问题(start+1, size)
        return find(r, start+1, size, element);
    }
}
```

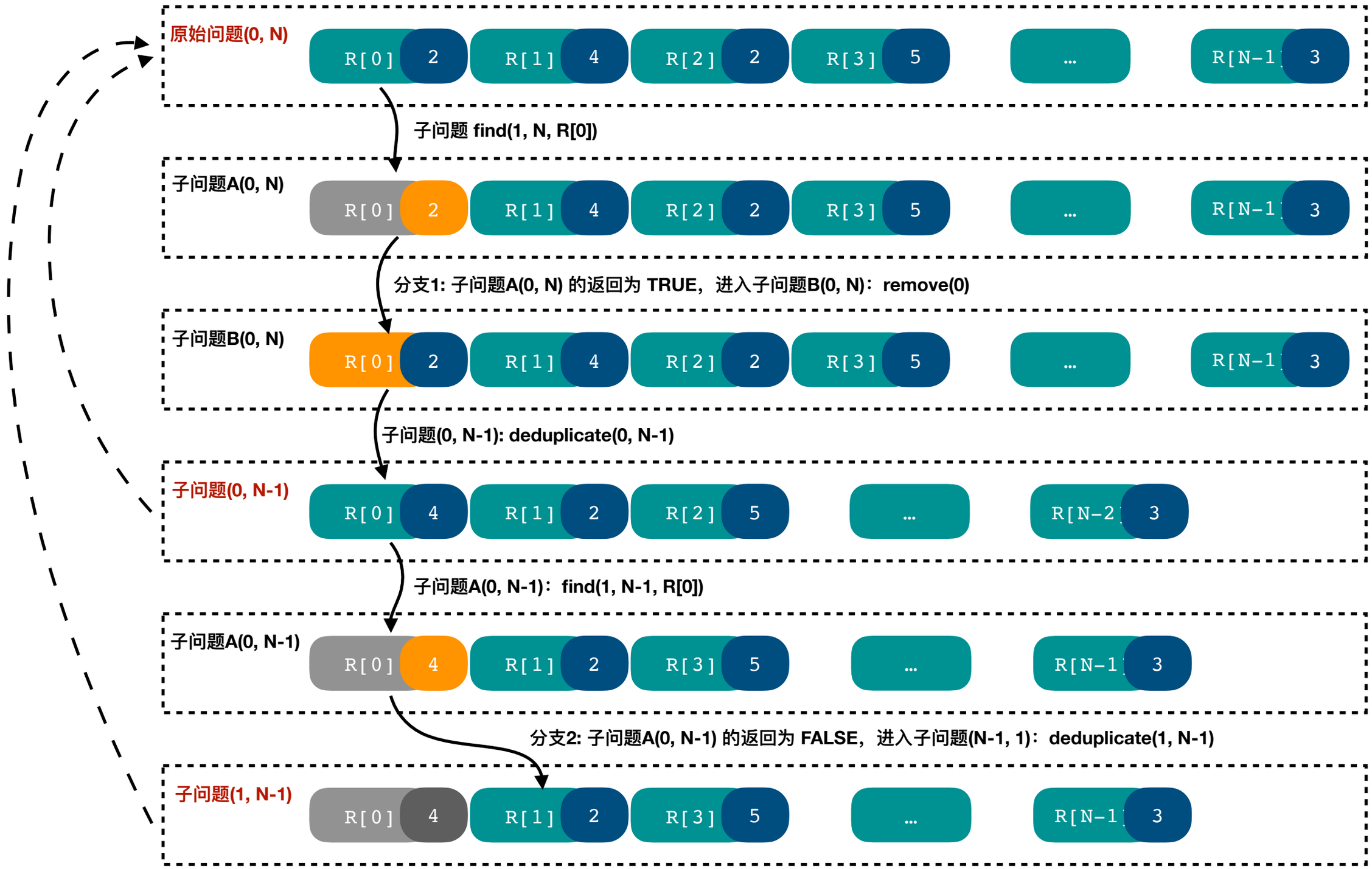
- 两种经典计算机语言
  - 递归思想的应用（5）—— 去重 vector<T>::deduplicate (0, N)



- 两种经典计算机语言
  - 递归思想的应用（5）—— 去重 vector<T>::deduplicate (0, N)



- 两种经典计算机语言
  - 递归思想的应用（5）—— 去重 vector<T>::deduplicate (0, N)



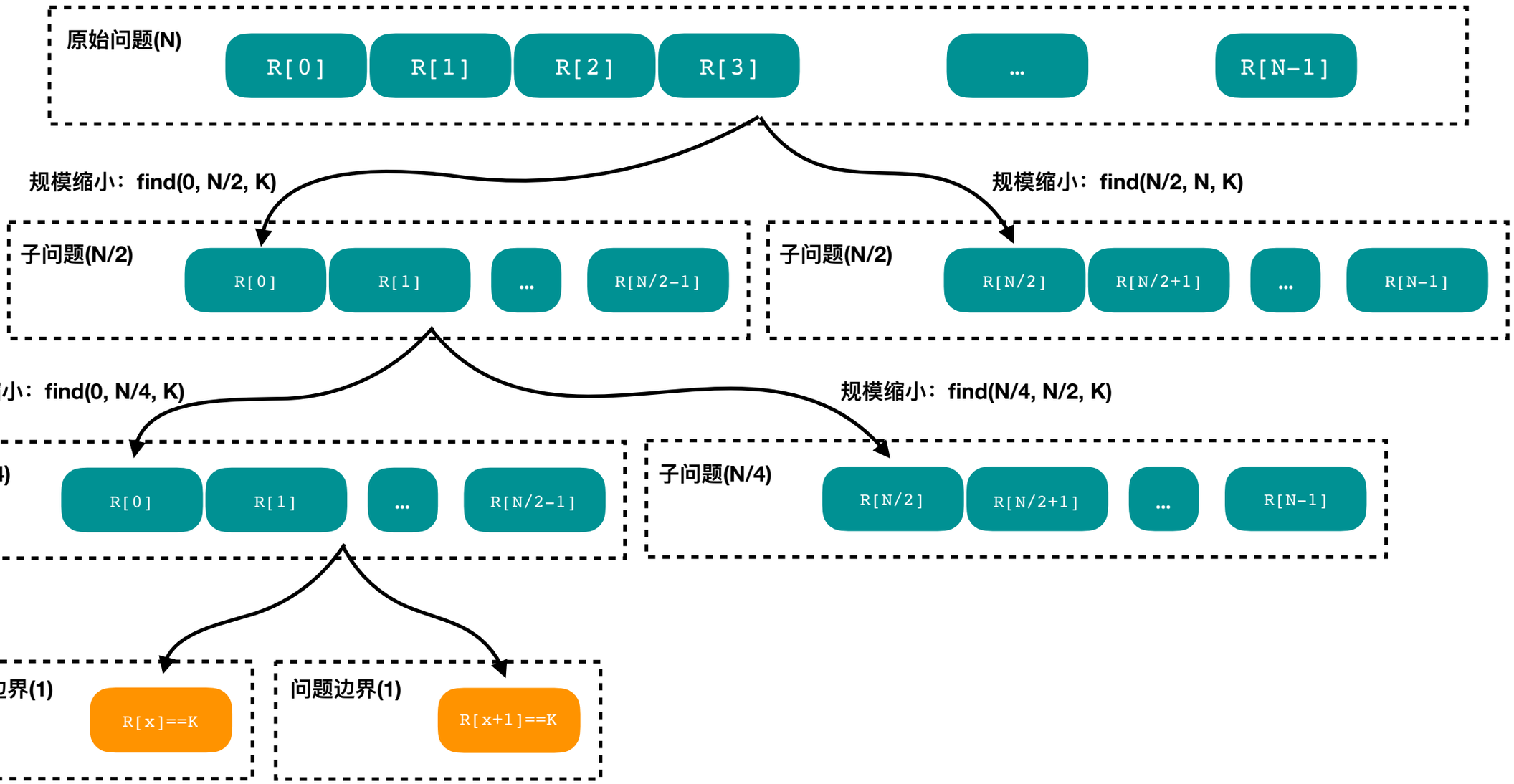
- 两种经典计算机语言
  - 递归思想的应用（5）—— 去重 vector<T>::deduplicate (0, N)

```
// 线性表去重算法
//
// @params r: 输入线性表
// @params start: 执行去重的下标起点
// @params end: 执行去重的下标终点
//
// @return 返回去重后的长度
//
template<typename T>
int deduplicate(T* r, int start, int end) {
    // 判断是否到达问题边界，若到达问题边界，去重完成，返回去重后的线性表长度
    if (start >= end) {
        return end;
    }
    // 子问题A(start+1, end)，判断线性表中是否存在与 r[start] 重复的元素
    if (find(r, start+1, end, r[start]) >= 0) {
        // 若存在与 r[start] 重复的元素，则求解子问题B，删除 r[start]
        remove(r, end, start);
        // 递归求解子问题 (start, end-1)，注意元素删除后线性表长度应减一
        return deduplicate(r, start, end - 1);
    } else {
        // 递归求解子问题 (start+1, end)
        return deduplicate(r, start + 1, end);
    }
}
```

- 两种经典计算机语言
  - 递归思想回顾：
    - **原始问题的规模缩小到一定的程度，就可以容易的解决；**
    - **原始问题可以分解为规模较小的相同问题（术语：具有最优子结构性质）；**
    - **可以通过合并子问题的解，得到原始问题的解。**
  - 扩展练习：—— 在一个整数序列中，求取其数值和最大的子序列。

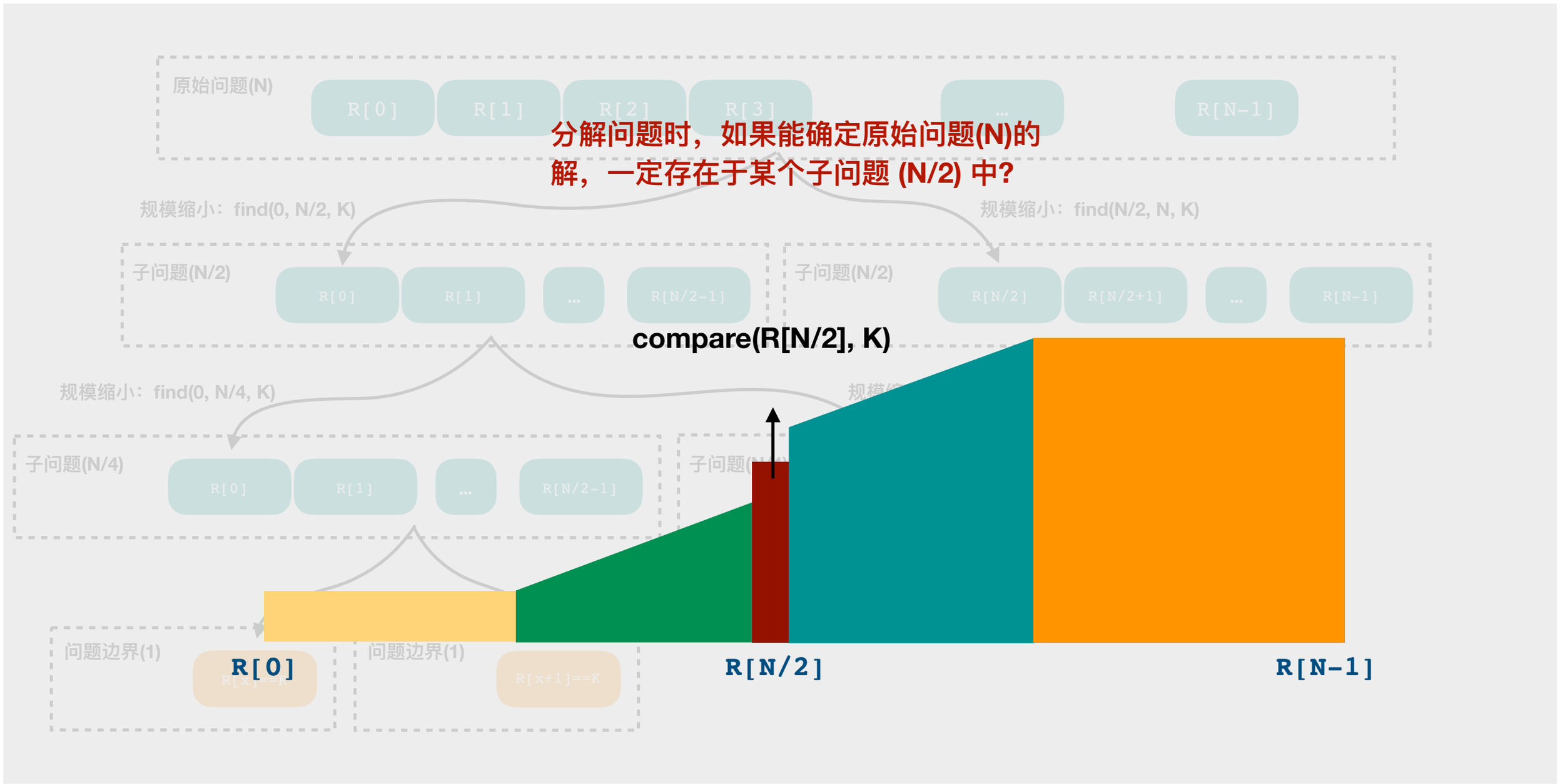
    **HDU OJ 1003. <Max Sum>**
    **http://acm.hdu.edu.cn/showproblem.php?pid=1003**

- 两种经典计算机语言
  - 分治思想的应用（1）—— 二分查找 vector<T>::binary_find (0, N, K)

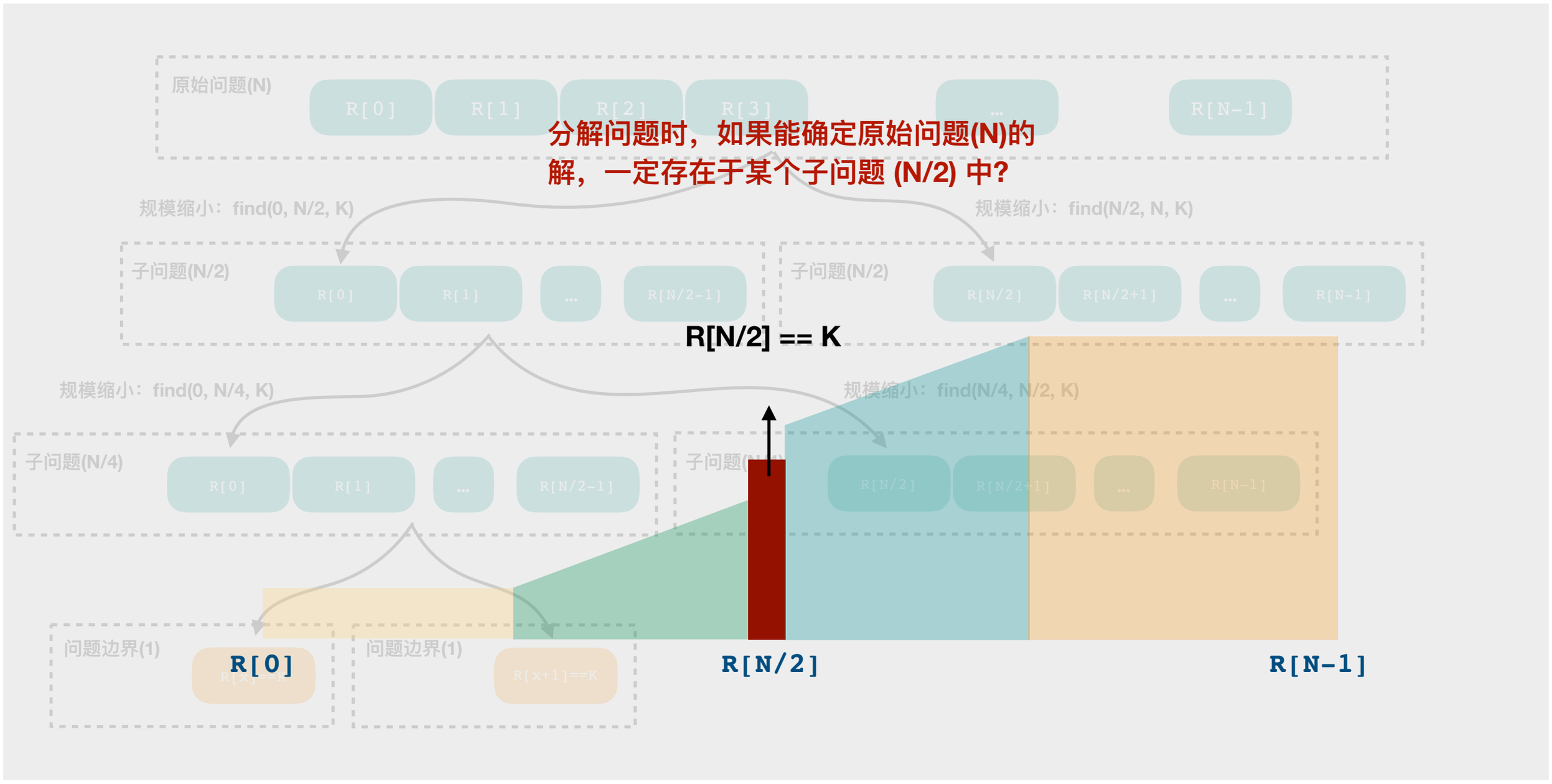



- 思考：
  - 分治思想与并行计算?

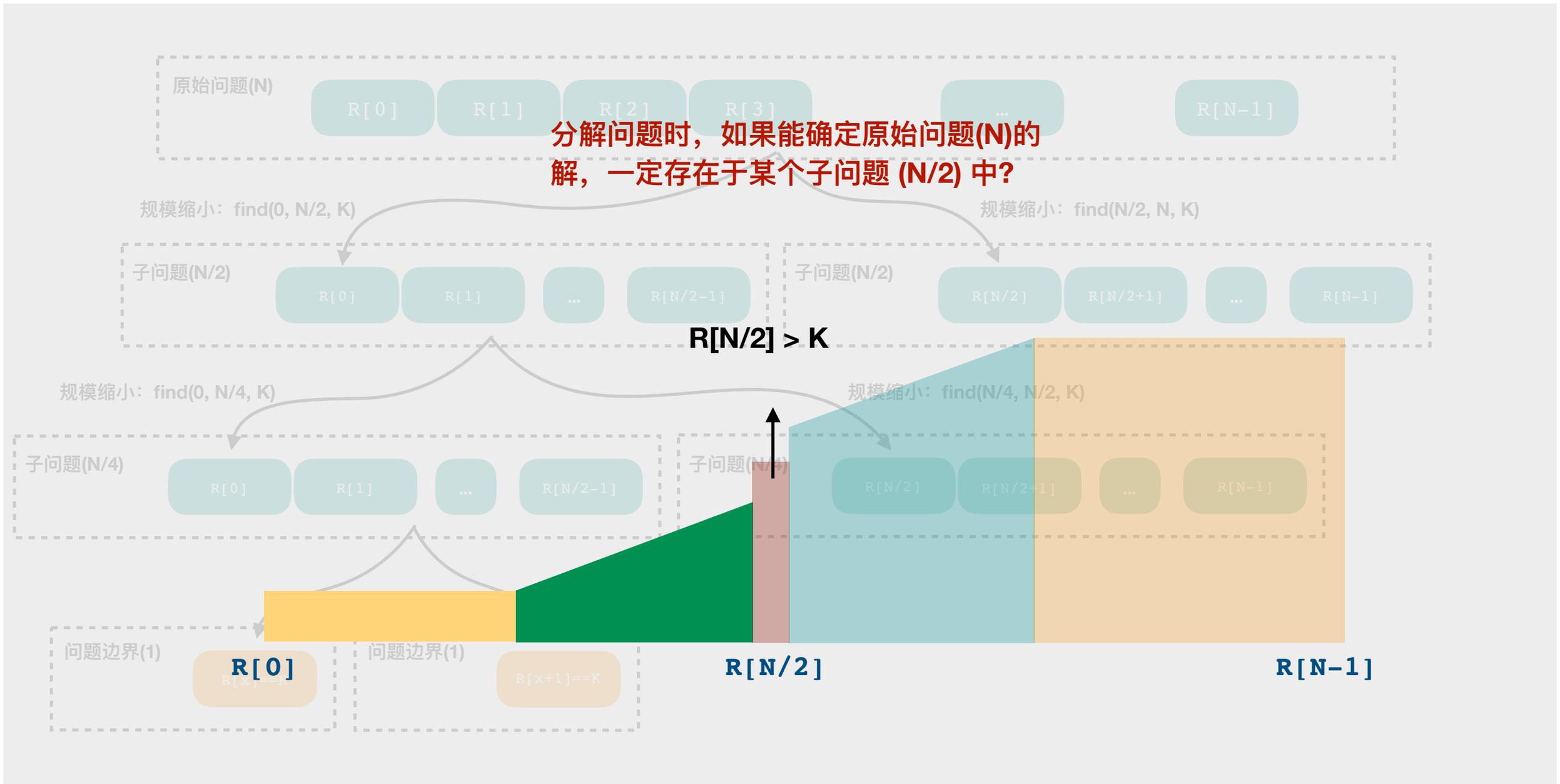
- 两种经典计算机语言
  - 分治思想的应用（2）—— 有序数组中的二分查找
    vector<T>::binary_search (0, N, K)



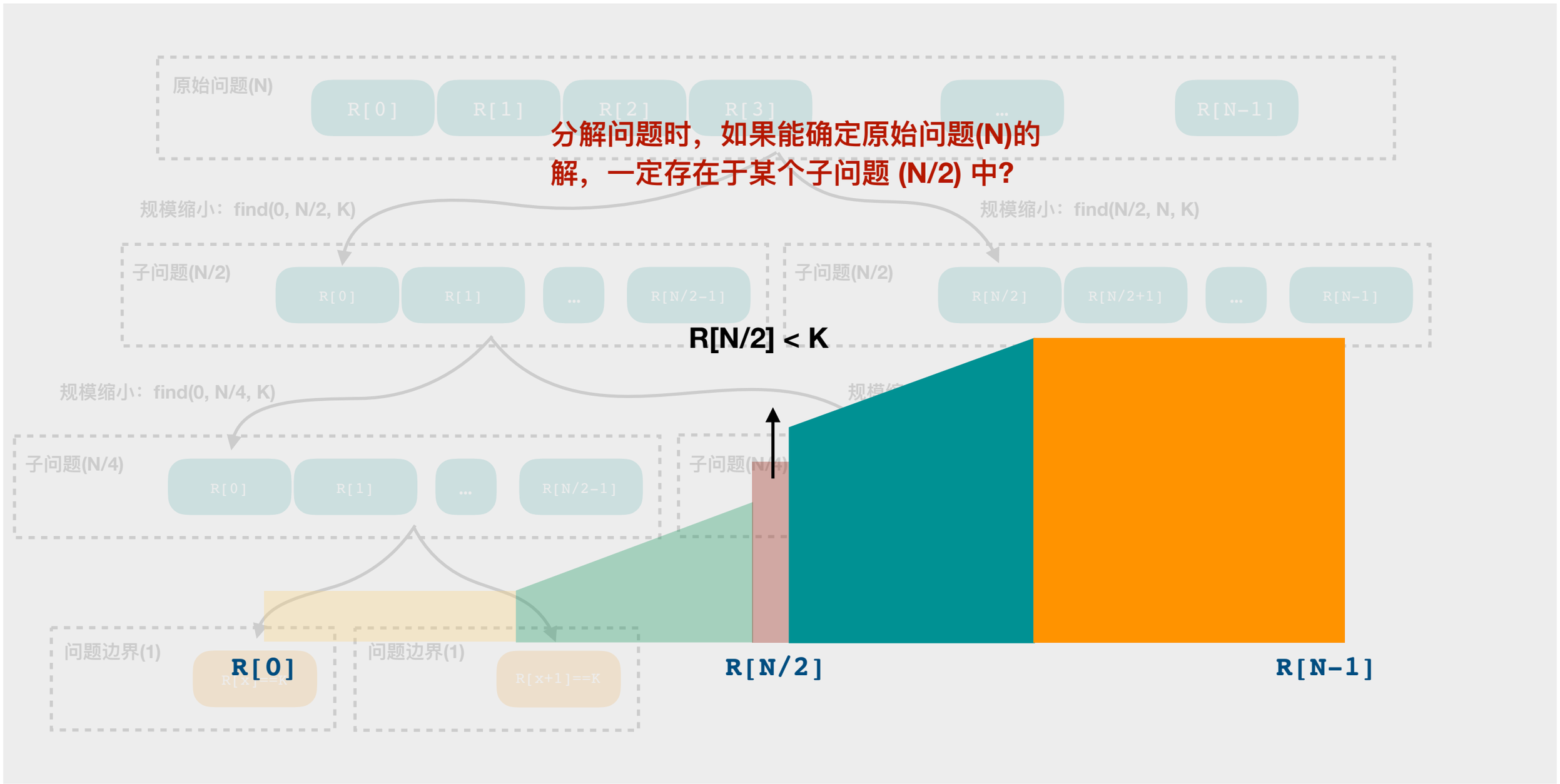
- 两种经典计算机语言
  - 分治思想的应用（2）—— 有序数组中的二分查找
    vector<T>::binary_search (0, N, K)



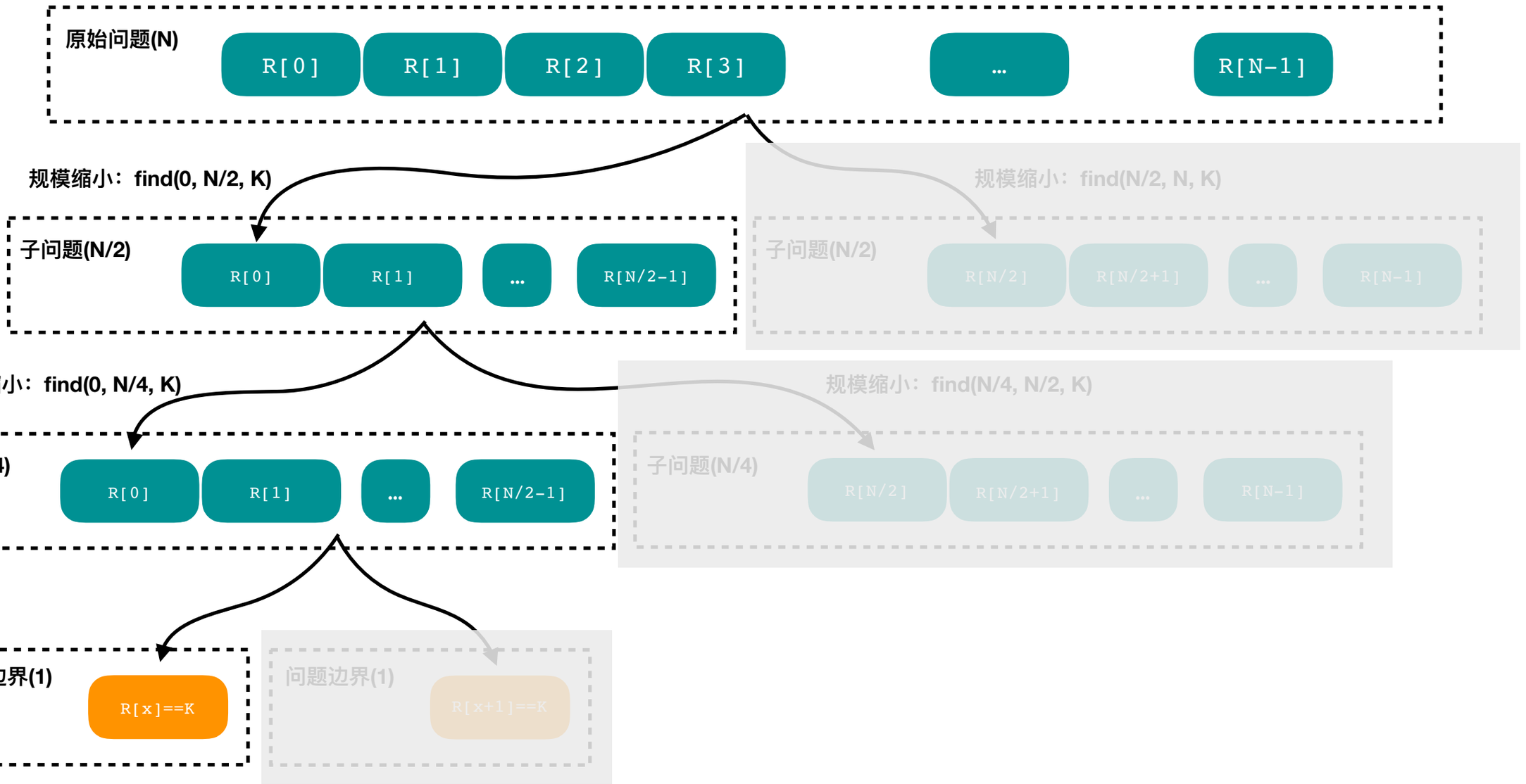
- 两种经典计算机语言
  - 分治思想的应用（2）—— 有序数组中的二分查找
    vector<T>::binary_search (0, N, K)



- 两种经典计算机语言
  - 分治思想的应用（2）—— 有序数组中的二分查找
    vector<T>::binary_search (0, N, K)



- 两种经典计算机语言
  - 分治思想的应用（2）—— 有序数组中的二分查找
    vector<T>::binary_search (0, N, K)



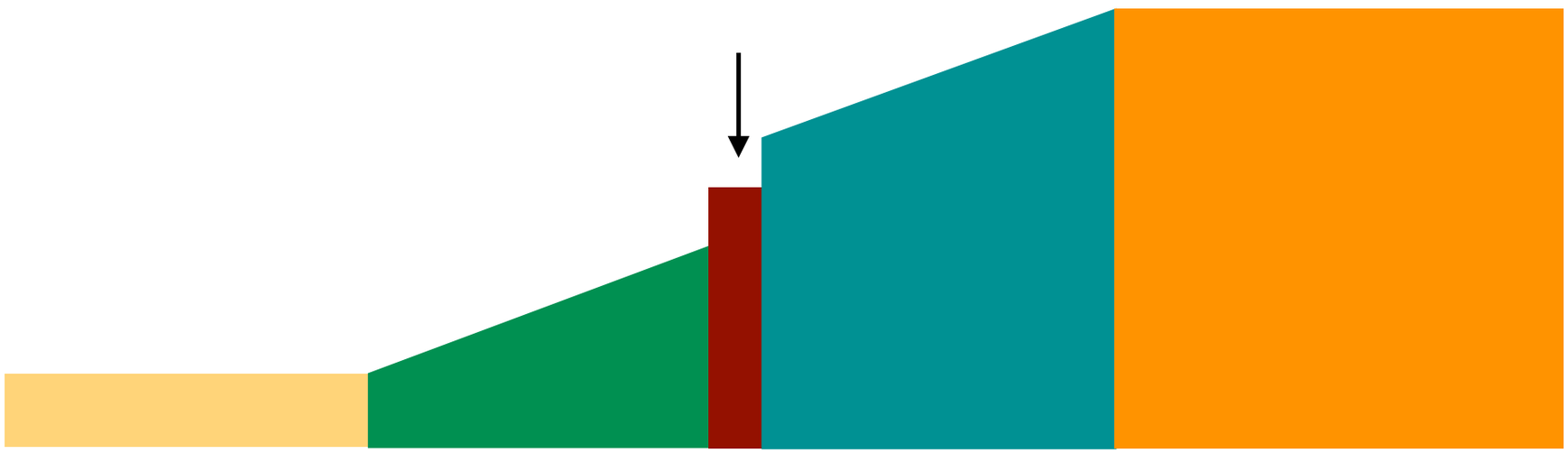
- 两种经典计算机语言
  - 分治思想的应用（2）—— 有序数组插入与删除
    vector<T>::sorted_insert (K)

```
// 以插入为例，先二分查找，再插入元素，会有什么问题?
sorted_insert(K) { insert(binary_search(K), K); }
```

**思考应用 (2) 两类情况：**

**1. 若数组中存在多个等于 K 的元素，我们返回的是第几个？（保持插入稳定性）**

**2. 反之，若数组中不存在等于 K 的元素，会如何执行？（保持有序）**



- 两种经典计算机语言
  - 分治思想的动机：
    - 不变的：
      - 原始问题的规模缩小到一定的程度，就可以容易的解决；
      - 原始问题可以分解为规模较小的相同问题（术语：具有最优子结构性质）；
      - 可以通过合并子问题的解，得到原始问题的解。
    - 变的：
      - 并行化是现代高性能编程的核心
      - 维护有序性、单调性，在分治中减少计算复杂性

- 扩展练习：—— 在一个整数序列中，求满足子序列和超过 K 的子序列的最短长度。

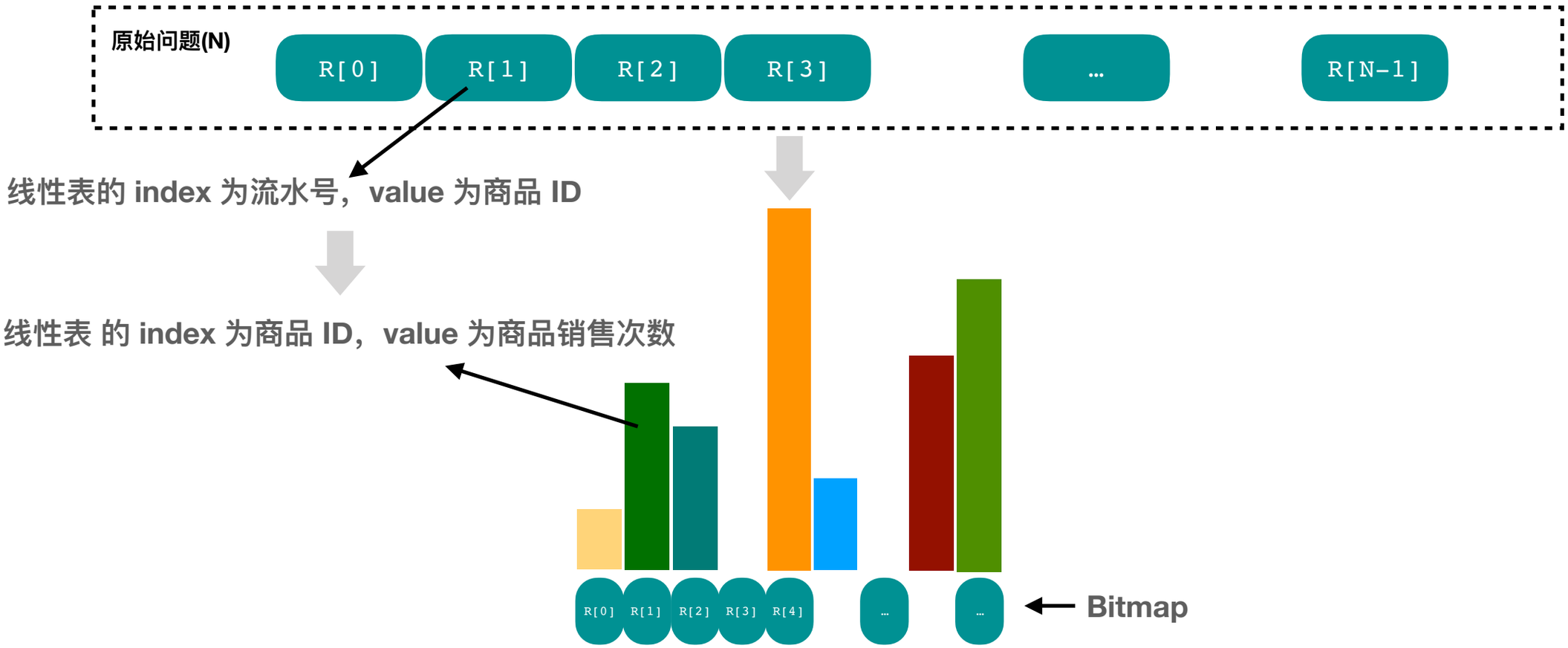
  **POJ 3061. <Subsequence>**
  **http://poj.org/problem?id=3061**

- 穿插话题2：函数式编程
  - 函数式编程是一种古老的编程方法，但很有趣，也对培养递归思维十分有帮助；
  - 学习方法：了解函数式编程的哲学，并在主流的编程语言中践行，**切勿走火入魔**
    - 尽量定义引用透明、无副作用的函数：函数运行不依赖外部变量，返回值只与输入参数有关的函数；
    - 用递归、分治思想设计算法，随时考虑**并行化**；
  - 推荐阅读：
    - 《使用递归的方式去思考》by IBM Developer

# 线性表

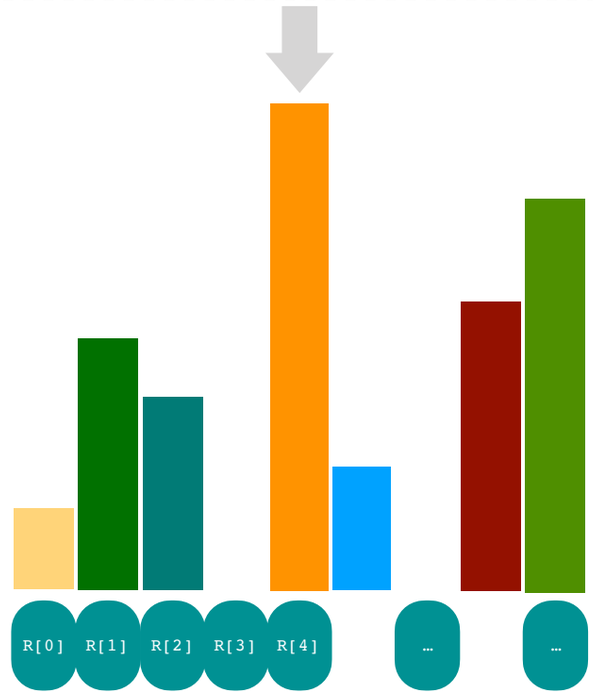
- 线性表的几种物理实现
- 两种经典计算机语言——递归思想、分治思想在线性表中的体现
- 线性表的典型应用：Bitmap

- 线性表的典型应用：Bitmap
  - 老问题——去重 vector<T>::deduplicate (0, N)
    - **递归思想：O(N^2)**
  - 新场景：
    - 如果 N 很大，但线性表中元素取值范围 [0, m)，且 m 不大
    - 例如：从一个超市的销售流水中，找出有销售记录的商品



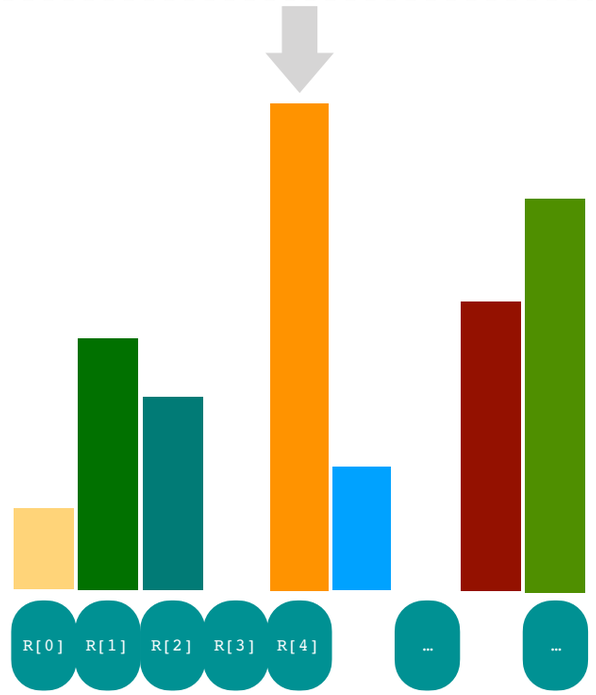
- 线性表的典型应用：Bitmap
  - 老问题——去重 vector<T>::deduplicate (0, N)
    - **递归思想：O(N^2)**
  - 新场景：
    - 如果 N 很大，但线性表中元素取值范围 [0, m)，且 m 不大
    - 例如：从一个超市的销售流水中，找出有销售记录的商品


```
// 使用 RAII 定义 Bitmap
Array<int> bitmap_(M);
int* bitmap = bitmap_.get();
// 初始化 Bitmap，所有元素置为 0
memset(bitmap, 0, sizeof(int) * M);
// 将销售流水记录中的商品信息保存在 Bitmap 中
for (int i = 0; i < N; ++i) {
    bitmap[r[i]] += 1;
}
// 遍历 Bitmap
for (int i = 0; i < M; ++i) {
    if (bitmap[i] > 0) {
        // 找到符合要求的商品 i
    }
}
```



- 线性表的典型应用：Bitmap、Hashmap
  - 老问题—— 去重 vector<T>::deduplicate (0, N)
    - **递归思想：O(N^2)**
  - 新场景：
    - 如果 N 很大，线性表中元素取值范围 [0, m)，且 m 也不小
    - 如果 N 很大，且线性表中元素是英文单词


- **问题来了：**
  - **如何将 [0, m) 的数字或 英文单词 映射到有限的 Bitmap 中?**
  - **如何处理映射后的冲突?**
  - **Bitmap 能像 Vector 那样动态扩容吗?**



- 线性表的典型应用：Bitmap、Hashmap
  - Bitmap 总结：
    - **需要寻找一种映射方法，将线性表中的元素映射到 Bitmap 中，转换为另一种集合表示**
  - 扩展阅读：
    - Hashmap：https://zh.wikipedia.org/zh-hans/哈希表
    - Bloomfilter：https://zh.wikipedia.org/zh-hans/布隆过滤器
  - 扩展练习：实现一个英文字典检索算法

    **POJ 2503. <Babelfish >**
    **http://poj.org/problem?id=2503**

# 扩展练习

**HDU OJ 1003. <Max Sum>**
**http://acm.hdu.edu.cn/showproblem.php?pid=1003**

**POJ 3061. <Subsequence>**
**http://poj.org/problem?id=3061**

**POJ 2503. <Babelfish >**
**http://poj.org/problem?id=2503**