
第四课 递归、回溯与分治

林沐

内容概述

1.6道经典**递归、回溯、分治**的相关题目

预备知识:递归函数与回溯算法

例1-a: 求子集(medium) (回溯法、位运算法)

例1-b: 求子集2(medium) (回溯法)

例1-c: 组合数之和2(medium) (回溯法、剪枝)

例2:生成括号(medium) (递归设计)

例3:N皇后 (hard) (回溯法)

预备知识:分治算法与归并排序

例4:逆序数 (hard) (分治法、归并排序应用)

2.详细讲解题目**解题方法、代码实现**

预备知识:递归函数基础



sum = 6

```
#include <stdio.h>
void compute_sum_3(int i, int &sum) { //i = 3, 将sum加上3
    sum += i;
}
void compute_sum_2(int i, int &sum) { //i = 2, 将sum加上2, 调用
    sum += i;                          compute_sum_3(3, sum)
    compute_sum_3(i + 1, sum);
}
void compute_sum_1(int i, int &sum) { //i = 1, 将sum加上1, 调用
    sum += i;                          compute_sum_2(2, sum)
    compute_sum_2(i + 1, sum);
}
int main() {
    int sum = 0; //计算1 + 2 + 3
    compute_sum_1(1, sum);
    printf("sum = %d\n", sum); 将结果存储至sum, 并打印sum
    return 0;
}
```

预备知识:递归函数基础

```
#include <stdio.h>

void compute_sum(int i, int &sum) {
    if (i > 3) {
        return; //i > 3 时, 结束递归调用
    }
    sum += i; //将i累加至sum
    compute_sum(i + 1, sum);
}

//递归调用, 下一次调用会累加i+1

int main() {
    int sum = 0;
    compute_sum(1, sum);
    printf("sum = %d\n", sum);
    return 0;
}
```



A terminal window with a black background and white text displaying the output of the program: `sum = 6`.

compute_sum(1, sum);

i = 1, sum = 1



compute_sum(2, sum);

i = 2, sum = 3



compute_sum(3, sum);

i = 3, sum = 6



compute_sum(4, sum);

i = 4, return

预备知识:课堂练习

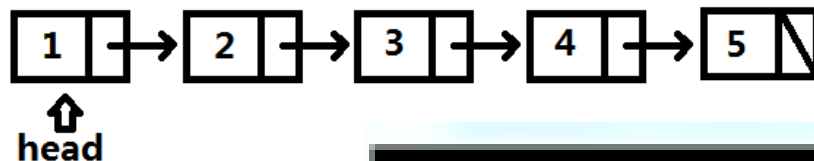
```
#include <stdio.h>
#include <vector>
```

```
struct ListNode {           //链表数据结构
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

//递归将head指针指向的节点的数据域val, push到vec里

```
void add_to_vector(ListNode *head, std::vector<int> &vec) {
    if (1) {
        return;
    }
    2
    3
}
```

```
int main(){
    ListNode a(1);
    ListNode b(2);
    ListNode c(3);
    ListNode d(4);
    ListNode e(5);
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    std::vector<int> vec;
    add_to_vector(&a, vec);
    for (int i = 0; i < vec.size(); i++){
        printf("[%d]", vec[i]);
    }
    printf("\n");
    return 0;
}
```



//连接链表对应的节点

[1][2][3][4][5]
请按任意键继续. . .

1分钟时间填写代码,
有问题随时提出!

```
#include <stdio.h>
#include <vector>
```

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

//链表数据结构

//递归将head指针指向的节点的数据域val, push到vec里

```
void add_to_vector(ListNode *head, std::vector<int> &vec) {
```

```
    if (!head) {
        return;
    }
```

//如果head为空则结束递归

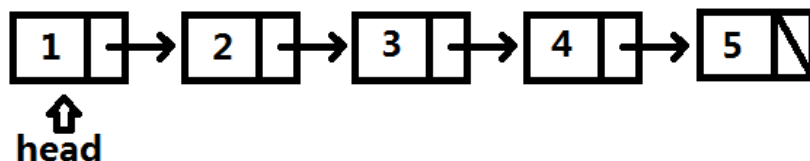
```
    vec.push_back(head->val);
```

//将当前遍历的节点值push进入vec

```
    add_to_vector(head->next, vec);
```

//继续递归后续链表

```
int main() {
    ListNode a(1);
    ListNode b(2);
    ListNode c(3);
    ListNode d(4);
    ListNode e(5);
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    std::vector<int> vec;
    add_to_vector(&a, vec);
    for (int i = 0; i < vec.size(); i++) {
        printf("[%d]", vec[i]);
    }
    printf("\n");
    return 0;
}
```



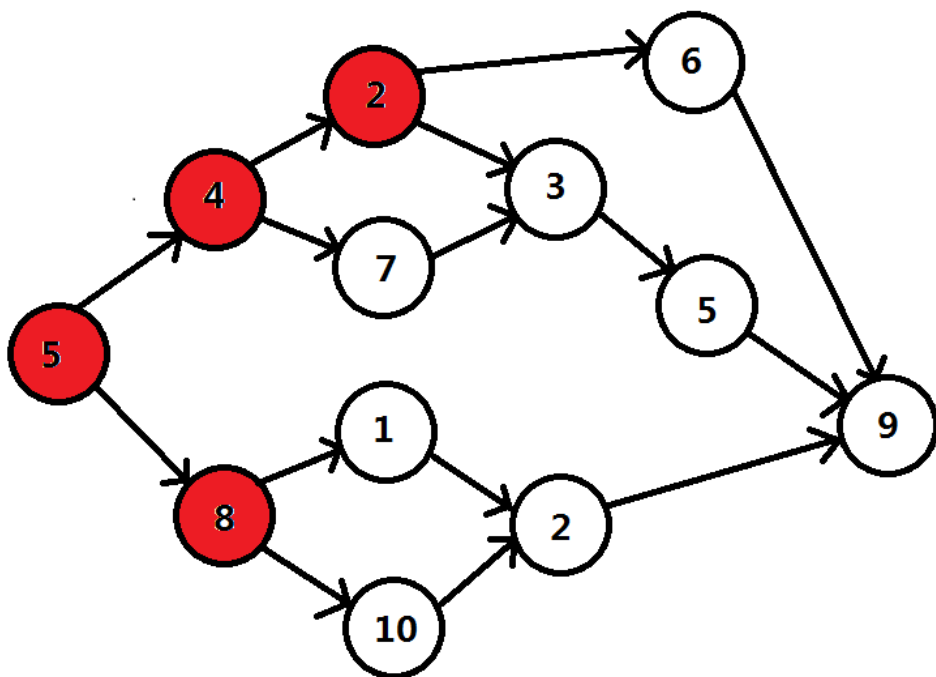
//连接链表对应的节点

```
[1][2][3][4][5]
请按任意键继续. . .
```

预备知识:实现

预备知识:回溯法

回溯法又称为**试探法**，但当**探索**到某一步时，发现原先**选择达不到目标**，就**退回一步重新选择**，这种**走不通就退回再走**的技术为回溯法。



找出路径上值的和大于30的所有路径

5 4 2 6 9 sum = 26

5 4 2 3 5 9 sum = 28

5 4 7 3 5 9 sum = 33

5 8 1 2 9 sum = 25

5 8 10 2 9 sum = 34

例1-a:求子集

已知一组数(其中**无重复元素**)，求这组数可以组成的**所有子集**。
结果中不可有**无重复的**子集。

例如: `nums[] = [1, 2, 3]`

结果为: `[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]`

```
class Solution {  
public:  
    std::vector<std::vector<int>> > subsets(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 78. Subsets**

<https://leetcode.com/problems/subsets/description/>

难度:**Medium**

例1-a:思考

在**所有子集**中，生成**各个**子集， $[\]$, $[1]$, $[2]$, $[3]$, $[1, 2]$, $[1, 3]$, $[2, 3]$, $[1, 2, 3]$ ，即是否选**[1]**，是否选**[2]**，是否选**[3]**的问题。

如果**只使用循环**，困难的地方在哪里？

使用循环程序难以**直接**模拟**是否选**某一元素的过程。

如果只是生成 $[1]$, $[1, 2]$, $[1, 2, 3]$ 三个子集，如何做？

思考**半分钟**。

例1-a(方法1回溯法):预备知识(循环)

```
#include <stdio.h>
#include <vector> //nums[] = [1, 2, 3] , 将子集[[1], [1, 2], [1, 2, 3]]打印出来。
```

```
int main() {
    std::vector<int> nums;
    nums.push_back(1);
    nums.push_back(2);
    nums.push_back(3);
```

```
[1]
[1][2]
[1][2][3]
请按任意键继续. . .
```

```
std::vector<int> item; //item , 生成各个子集的数组
std::vector<std::vector<int>> > result; //result , 最终结果数组

for (int i = 0; i < nums.size(); i++) { //i=0时 , item=[1]
    item.push_back(nums[i]);           //i=1时 , item=[1,2]
    result.push_back(item);             //i=2时 , item=[1,2,3]
}

for (int i = 0; i < result.size(); i++) {
    for (int j = 0; j < result[i].size(); j++) {
        printf("[%d]", result[i][j]); //打印result
    }
    printf("\n");
}
return 0;
}
```

例1-a(方法1回溯法):预备知识(递归)

```
#include <stdio.h>
#include <vector> //nums[] = [1, 2, 3] , 将子集[[1], [1, 2], [1, 2, 3]]递归的加入result。
//每次递归的将下一个nums元素放入生成子集的数组item , 产生对应子集
```

```
void generate(int i, std::vector<int>& nums, //生成子集的数组item
              std::vector<int> &item,
              std::vector<std::vector<int>> &result) {
```

```
    if (1) { //最终结果数组
        return;
    }
```

```
    2 //i=0时, item=[1]
```

```
    result.push_back(item); //i=1时, item=[1,2]
```

```
    3 //i=2时, item=[1,2,3]
```

```

}

int main() {
    std::vector<int> nums;
    nums.push_back(1);
    nums.push_back(2);
    nums.push_back(3);
    std::vector<int> item;
    std::vector<std::vector<int>> result;
    generate(0, nums, item, result);
    for (int i = 0; i < result.size(); i++) {
        for (int j = 0; j < result[i].size(); j++) {
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

```
[1]
[1][2]
[1][2][3]
请按任意键继续. . .
```

例1-a(方法1回溯法):预备知识(递归)

```
#include <stdio.h>
#include <vector> //nums[] = [1, 2, 3] , 将子集[[1], [1, 2], [1, 2, 3]]递归的加入result.

void generate(int i, std::vector<int>& nums,
             std::vector<int> &item,
             std::vector<std::vector<int>> &result) {
    if (i >= nums.size()) { //递归结束条件, 当i下标超过nums
        return;           //数组长度时结束, 递归结束
    }
    item.push_back(nums[i]);
    result.push_back(item);
    generate(i + 1, nums, item, result);
}

int main() {
    std::vector<int> nums;
    nums.push_back(1);
    nums.push_back(2);
    nums.push_back(3);
    std::vector<int> item;
    std::vector<std::vector<int>> result;
    generate(0, nums, item, result);
    for (int i = 0; i < result.size(); i++) {
        for (int j = 0; j < result[i].size(); j++) {
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

第一次递归调用:

generate(0, nums, item, result);

i = 0, item = [1], result = [[1]]



第二次递归调用:

generate(1, nums, item, result);

i = 1, item = [1,2], result = [[1], [1,2]]



第三次递归调用:

j generate(2, nums, item, result);

i = 2, item = [1, 2, 3], result = [[1], [1,2], [1,2,3]]



第四次递归调用:

generate(3, nums, item, result);

i == nums.size(), return

例1-a(方法1回溯法):算法思路

利用回溯方法生成**子集**，即对于**每个元素**，都有试探**放入**或**不放入**集合中的两个选择：

选择**放入**该元素，**递归的**进行后续元素的选择，完成放入该元素后续所有元素的试探；之后**将其拿出**，即**再进行一次**选择**不放入**该元素，**递归的**进行后续元素的选择，完成不放入该元素后续所有元素的试探。

本来选择放入，再选择一次不放入的这个过程，称为回溯试探法。

例如：

元素数组：nums = [1, 2, 3, 4, 5, ...]，子集生成数组item[] = []

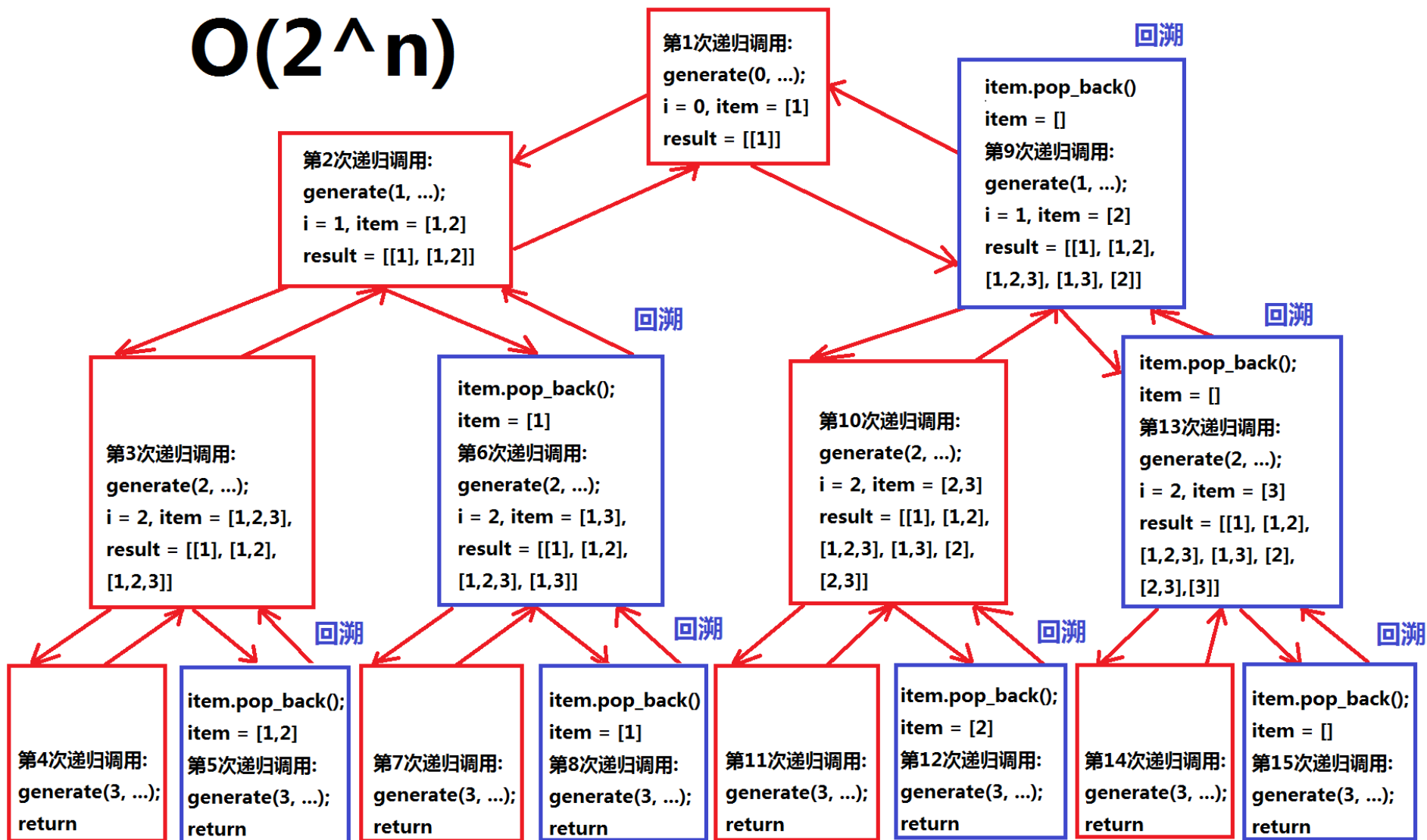
对于**元素1**，

选择**放入**item，item = [1]，继续递归处理后续[2,3,4,5,...]元素；item = [1,...]

选择**不放入**item，item = []，继续递归处理后续[2,3,4,5,...]元素；item = [...]

例1-a(方法1回溯法):算法思路

$O(2^n)$



例1-a(方法1回溯法):课堂练习

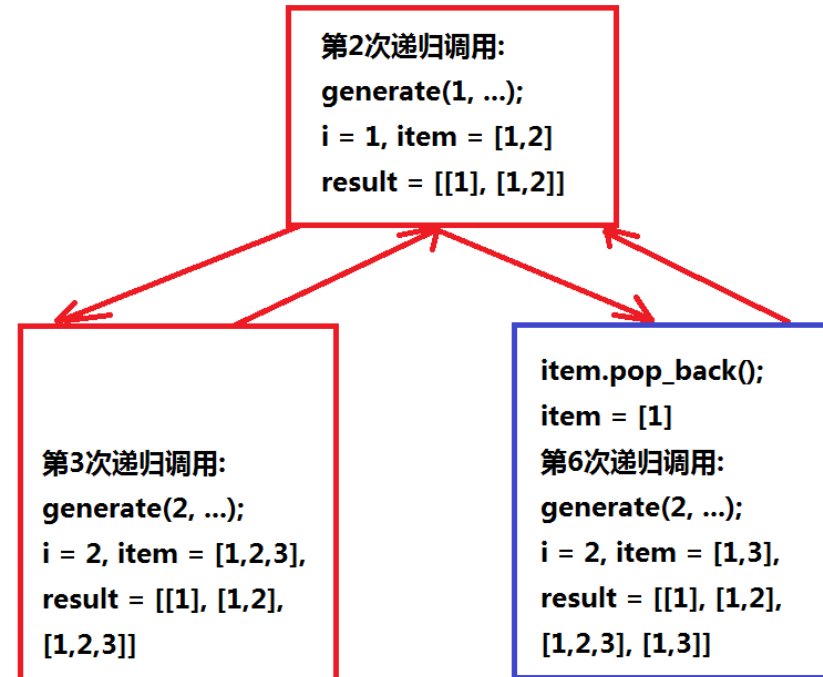
```
#include <vector>
class Solution {
public:
    std::vector<std::vector<int> > subsets(std::vector<int>& nums) {
        std::vector<std::vector<int> > result; //存储最终结果的result
        std::vector<int> item; //回溯时,产生各个子集的数组
        result.push_back(item); //将空集push进入result
        generate(0, nums, item, result); //计算各个子集
        return result;
    }
private:
    void generate(int i, std::vector<int>& nums,
                 std::vector<int> &item,
                 std::vector<std::vector<int> > &result) {
        if (1) { //递归结束条件是??
            return;
        }
        2
        result.push_back(item); //将当前生成的子集添加进入result
        generate(i + 1, nums, item, result); //第一次递归调用
        3
        generate(i + 1, nums, item, result); //第二次递归调用
    }
};
```

3分钟时间填写代码,
有问题随时提出!

例1-a(方法1回溯法):实现

```
#include <vector>
class Solution {
public:
    std::vector<std::vector<int> > subsets(std::vector<int>& nums
    std::vector<std::vector<int> > result; //存储最终结果的result
    std::vector<int> item; //回溯时,产生各个子集的数组
    result.push_back(item); //将空集push进入result
    generate(0, nums, item, result); //计算各个子集
    return result;
}

private:
    void generate(int i, std::vector<int>& nums,
                  std::vector<int> &item,
                  std::vector<std::vector<int> > &result) {
        if (i >= nums.size()) { //递归结束条件是??
            return;
        }
        item.push_back(nums[i]); //将当前生成的子集添加进入result
        result.push_back(item);
        generate(i + 1, nums, item, result); //第一次递归调用
        item.pop_back();
        generate(i + 1, nums, item, result); //第二次递归调用
    }
};
```



例1-a(方法2位运算法):预备知识

位运算是C语言(各编程语言)的基础运算符之一

运算符	含义	举例	十进制形式
&	按位与	0011 & 0101 = 0001	3 & 5 = 1
	按位或	0011 0101 = 0111	3 5 = 7
^	按位异或	0011 ^ 0101 = 0110	3 ^ 5 = 6
~	取反	~0011 = 1100	~3 = 12
<<	左移	0011 << 2 = 1100	3 << 2 = 3 * 2 * 2 = 12
>>	右移	0101 >> 2 = 0001	5 >> 2 = 5 / 2 / 2 = 1

例1-a(方法2位运算法):算法思路

若一个集合有三个元素A, B, C，则3个元素有 $2^3 = 8$ 种组成集合的方式，用0-7表示这些集合。

集合	整数	A	B	C
{}	000 = 0	0	0	0
{C}	001 = 1	0	0	1
{B}	010 = 2	0	1	0
{B,C}	011 = 3	0	1	1
{A}	100 = 4	1	0	0
{A,C}	101 = 5	1	0	1
{A,B}	110 = 6	1	1	0
{A,B,C}	111 = 7	1	1	1

例1-a(方法2位运算法):算法思路

A元素为 $100 = 4$ ；**B元素**为 $010 = 2$ ；**C元素**为 $001 = 1$

如**构造**某一集合，即使用A,B,C对应的三个整数与该集合对应的整数做**&运算**，当为**真**时，将该元素**push进入**集合。

集合	整数	A是否出现	B是否出现	C是否出现
{}	$000 = 0$	$100 \& 000 = 0$	$010 \& 000 = 0$	$001 \& 000 = 0$
{C}	$001 = 1$	$100 \& 001 = 0$	$010 \& 001 = 0$	$001 \& 001 = 1$
{B}	$010 = 2$	$100 \& 010 = 0$	$010 \& 010 = 1$	$001 \& 010 = 0$
{B,C}	$011 = 3$	$100 \& 011 = 0$	$010 \& 011 = 1$	$001 \& 011 = 1$
{A}	$100 = 4$	$100 \& 100 = 1$	$010 \& 100 = 0$	$001 \& 100 = 0$
{A,C}	$101 = 5$	$100 \& 101 = 1$	$010 \& 101 = 0$	$001 \& 101 = 1$
{A,B}	$110 = 6$	$100 \& 110 = 1$	$010 \& 110 = 1$	$001 \& 110 = 0$
{A,B,C}	$111 = 7$	$100 \& 111 = 1$	$010 \& 111 = 1$	$001 \& 111 = 1$

例1-a(方法2位运算法):课堂练习

```
#include <vector>
class Solution {
public:
    std::vector<std::vector<int> > subsets(std::vector<int>& nums) {
        std::vector<std::vector<int> > result;
        int all_set = 1 //设置全部集合的最大值+1
        for (int i = 0; i < all_set; i++) { //遍历所有集合
            std::vector<int> item;
            for (int j = 0; j < nums.size(); j++) {
                if (2) { //构造数字i代表的集合, 各元素存储至item
                    item.push_back(nums[j]);
                }
            }
            3
        }
        return result;
    }
};
```

3分钟时间填写代码,
有问题随时提出!

例1-a(方法2位运算法):实现

```
#include <vector>
class Solution {
public:
    std::vector<std::vector<int> > subsets(std::vector<int>& nums) {
        std::vector<std::vector<int> > result;
        int all_set = 1 << nums.size(); //设置全部集合的最大值+1
                                         //1 << n 即为 2^n
        for (int i = 0; i < all_set; i++) { //遍历所有集合
            std::vector<int> item;
            for (int j = 0; j < nums.size(); j++) {
                if (i & (1 << j)) { //构造数字i代表的集合, 各元素存储至item
                    item.push_back(nums[j]);
                }
            }
            result.push_back(item); //整数 i 代表从 0至2^n-1 这 2^n个集合
                                   //(1 << j)即为构造nums数组的第j个元素
                                   //若 i & (1 << j)为真则 nums[j]放入item
        }
        return result;
    }
};
```

例1-a:测试与leetcode提交结果

方法1与方法2分别对应哪个打印结果?

```
int main() {  
    std::vector<int> nums;  
    nums.push_back(1);  
    nums.push_back(2);  
    nums.push_back(3);  
    std::vector<std::vector<int>> > result;  
    Solution solve;  
    result = solve.subsets(nums);  
    for (int i = 0; i < result.size(); i++) {  
        if (result[i].size() == 0) {  
            printf("[]");  
        }  
        for (int j = 0; j < result[i].size(); j++) {  
            printf("[%d]", result[i][j]);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

```
[]  
[1]  
[1][2]  
[1][2][3]  
[1][3]  
[2]  
[2][3]  
[3]
```

```
[]  
[1]  
[2]  
[1][2]  
[3]  
[1][3]  
[2][3]  
[1][2][3]
```

Subsets

Submission Details

10 / 10 test cases passed.

Status: Accepted

Runtime: 6 ms

Submitted: 0 minutes ago

例1-b:求子集2

已知一组数(其中有重复元素), 求这组数可以组成的所有子集。
结果中无重复的子集。

例如: `nums[] = [2, 1, 2, 2]`

结果为: `[[], [1], [1,2], [1,2,2], [1,2,2,2], [2], [2,2], [2,2,2]]`

注意: `[2,1,2]`与`[1,2,2]`是重复的集合!

```
class Solution {  
public:  
    std::vector<std::vector<int> > subsetsWithDup(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 90. Subsets II**

<https://leetcode.com/problems/subsets-ii/description/>

难度: **Medium**

例1-b:思考

有**两种**重复原因:

1.不同位置的元素组成的集合是**同一个子集**, **顺序相同**:

例如: $[2, 1, 2, 2]$,

选择第1,2,3个元素组成的子集: $[2, 1, 2]$;

选择第1,2,4个元素组成的子集: $[2, 1, 2]$ 。

2.不同位置的元素组成的集合是**同一个子集**, 虽然**顺序不同**, 但仍然代表了同一个子集, 因为**集合中的元素是无序的**。

例如: $[2, 1, 2, 2]$,

选择第1,2,3个元素组成的子集: $[2, 1, 2]$;

选择第2,3,4个元素组成的子集: $[1, 2, 2]$ 。

如何解决?思考**半分钟**。

例1-b:算法思路

不同位置的元素组成的集合是**同一个子集**，子集的各个元素**顺序相同**，或**顺序不同**，解决方法。

例如: [2, 1, 2, 2] :

选择第1,2,3个元素组成的子集:[2, 1, 2];

选择第1,2,4个元素组成的子集:[2, 1, 2];

选择第2,3,4个元素组成的子集:[1, 2, 2]。

解决方案:

先**对原始nums数组进行排序**，再**使用set去重**！

例如: [2, 1, 2, 2]排序后: [1, 2, 2, 2]

无论如何选择，均只出现[1, 2, 2]

例1-b:实现, 课堂练习

```
#include <vector>
#include <set>
#include <algorithm>
class Solution {
public:
    std::vector<std::vector<int>> > subsetsWithDup(std::vector<int>& nums) {
        std::vector<std::vector<int>> > result;
        std::vector<int> item;
        std::set<std::vector<int>> > res_set; //去重使用的集合set


1


        result.push_back(item);
        generate(0, nums, result, item, res_set);
        return result;
    }
private:
    void generate(int i, std::vector<int>& nums,
                 std::vector<std::vector<int>> > &result,
                 std::vector<int> &item,
                 std::set<std::vector<int>> > &res_set){
        if (i >= nums.size()){
            return;
        }
        item.push_back(nums[i]);
        if (

2

) {


3


            res_set.insert(item);
        }
        generate(i + 1, nums, result, item, res_set);
        item.pop_back();
        generate(i + 1, nums, result, item, res_set);
    }
};
```

3分钟时间填写代码,
有问题随时提出!

例1-b:实现

```
#include <vector>
#include <set>
#include <algorithm>
class Solution {
public:
    std::vector<std::vector<int>> > subsetsWithDup(std::vector<int>& nums) {
        std::vector<std::vector<int>> > result;
        std::vector<int> item;
        std::set<std::vector<int>> > res_set; //去重使用的集合set

        std::sort(nums.begin(), nums.end()); //对nums数组进行排序

        result.push_back(item);
        generate(0, nums, result, item, res_set);
        return result;
    }
private:
    void generate(int i, std::vector<int>& nums,
                 std::vector<std::vector<int>> > &result,
                 std::vector<int> &item,
                 std::set<std::vector<int>> > &res_set) {
        if (i >= nums.size()) {
            return;
        } //如果res_set集合中，无法找到item

        item.push_back(nums[i]);

        if (res_set.find(item) == res_set.end()) {
            result.push_back(item); //将item放入result数组中
            res_set.insert(item); //将item放入去重集合res_set中
        }

        generate(i + 1, nums, result, item, res_set);
        item.pop_back();
        generate(i + 1, nums, result, item, res_set);
    }
};
```

例1-b:测试与leetcode提交结果

```
int main() {
    std::vector<int> nums;
    nums.push_back(2);
    nums.push_back(1);
    nums.push_back(2);
    nums.push_back(2);

    std::vector<std::vector<int> > result;
    Solution solve;
    result = solve.subsetsWithDup(nums);
    for (int i = 0; i < result.size(); i++) {
        if (result[i].size() == 0) {
            printf("[]");
        }
        for (int j = 0; j < result[i].size(); j++) {
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

```
[]
[1]
[1][2]
[1][2][2]
[1][2][2][2]
[2]
[2][2]
[2][2][2]
请按任意键继续. . .
```

Subsets II

Submission Details

19 / 19 test cases passed.

Status: **Accepted**

Runtime: 16 ms

Submitted: 0 minutes ago

例1-c:组合数之和2

已知一组数(其中有重复元素), 求这组数可以组成的所有子集中, 子集中的各个元素和为整数target的子集, 结果中无重复的子集。

例如: `nums[] = [10, 1, 2, 7, 6, 1, 5]`, `target = 8`

结果为: `[[1, 7], [1, 2, 5], [2, 6], [1, 1, 6]]`

```
class Solution {
public:
    std::vector<std::vector<int>> > combinationSum2 (
        std::vector<int>& candidates,
        int target) {

    }
};
```

选自 **LeetCode 40. Combination Sum II**

<https://leetcode.com/problems/combination-sum-ii/description/>

难度: **Medium**

例1-c:思考

如下算法**是否可行**?

按照例1-b的**有重复元素的集合生成方法**，将**所有子集**构造出来后，再找出**元素和为target**的子集。

时间复杂度是?

是否有**更好的优化**方案?

思考**半分钟**。

例1-c:尝试

```
#include <vector>
#include <set>
#include <algorithm>
class Solution {
public:
```

```
    std::vector<std::vector<int> > combinationSum2(
        std::vector<int>& candidates,
        int target) {
```

● ● ● //这里的代码与例1-b完全一样

```
    generate(0, candidates, result, item, res_set);
```

```
    std::vector<std::vector<int> > target_result; // 存储最终结果
```

```
    for (int i = 0; i < result.size(); i++) {
```

```
        int sum = 0;
```

```
        for (int j = 0; j < result[i].size(); j++) {
```

```
            sum += result[i][j]; //计算各个子集的和
```

```
        }
```

```
        if (sum == target) { //将子集和为target的集合, 添加至target_result
```

```
            target_result.push_back(result[i]);
```

```
        }
```

```
    }
```

```
    return target_result;
```

```
}
```

```
private:
```

```
    void generate(int i, std::vector<int>& nums,
```

```
        std::vector<std::vector<int> > &result,
```

```
        std::vector<int> &item,
```

```
        std::set<std::vector<int> > &res_set) {
```

```
    if (i >= nums.size()) {
```

● ● ● //这里的代码与例1-b完全一样

例1-c:分析

无论是回溯法或位运算法，整体时间复杂度 $O(2^n)$ 。

例如：

`nums[] = [10, 1, 2, 7, 6, 1, 5], target = 8;`

`[10] > target`，则所有包含`[10]`的子集`[10,...]`，一定不满足目标。

`[1, 2, 7] > target`，则所有包含`[1, 2, 7]`的子集`[1, 2, 7,...]`，一定不满足目标。

`[7, 6] > target`，则所有包含`[7, 6]`的子集`[7, 6,...]`，一定不满足目标。

...

过多的错误尝试，浪费了大量时间。

Combination Sum II

Submission Details

122 / 172 test cases passed.

Status: Time Limit Exceeded

Submitted: 0 minutes ago

Last executed input: [14,6,25,9,30,20,33,34,28,30,16,12,31,9,9,12,34,16,25,32,8,7,30,12,33,20,21,29,24,17,27,34,11,17,30,6,32,21,27,17...
27

例1-c:算法思路

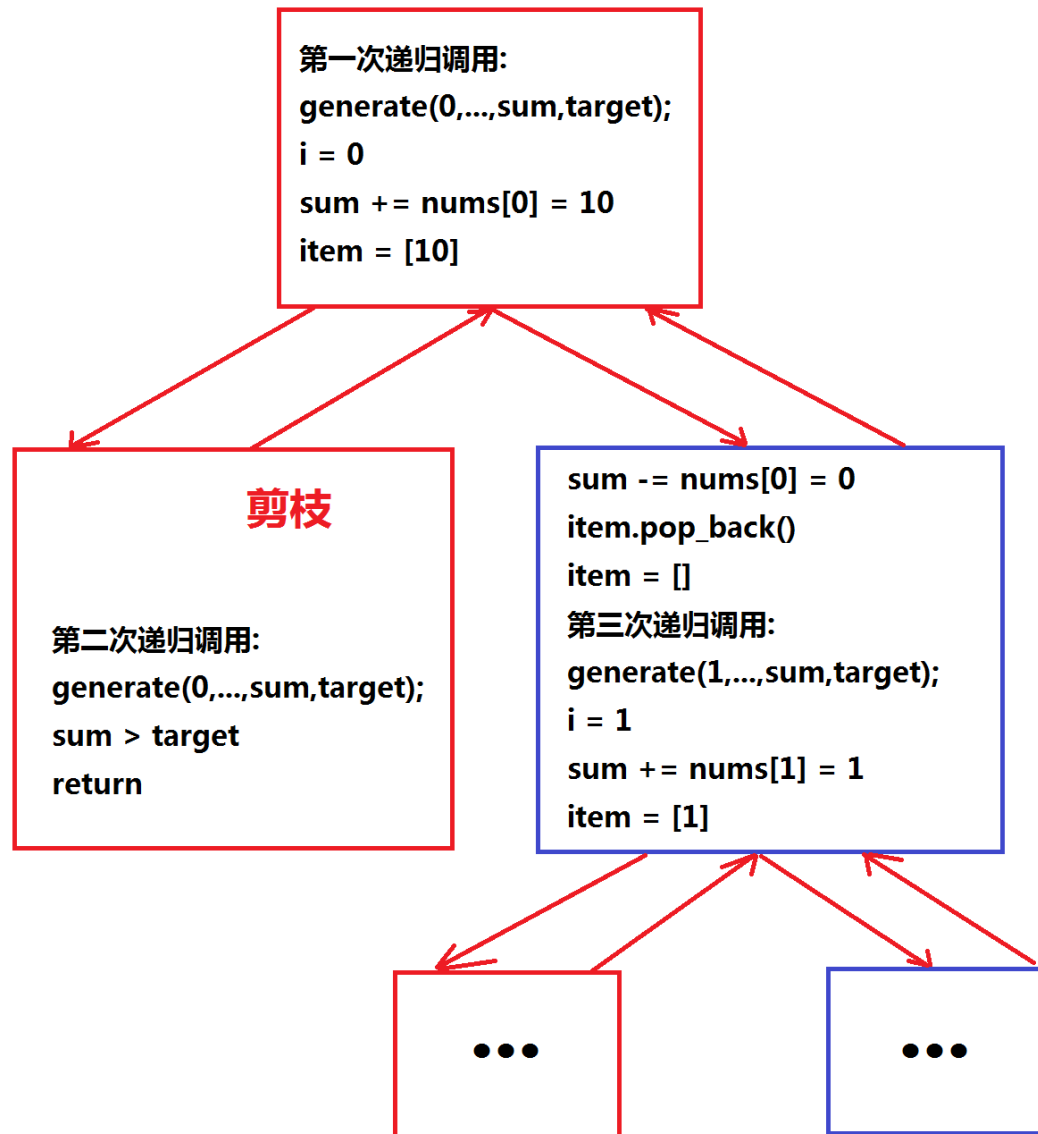
在搜索回溯过程中进行**剪枝操作**:

递归调用时, **计算**已选择元素的和
sum, 若 $\text{sum} > \text{target}$, 不再进行更
深的搜索, **直接返回**。

例如:

$\text{nums}[] = [10, 1, 2, 7, 6, 1, 5]$

$\text{target} = 8$



例1-c:实现, 课堂练习

```
#include <vector>
#include <set>
#include <algorithm>
class Solution {
public:
    std::vector<std::vector<int> > combinationSum2(
        std::vector<int>& candidates,
        int target){
        std::vector<std::vector<int> > result;
        std::vector<int> item;
        std::set<std::vector<int> > res_set;
        std::sort(candidates.begin(), candidates.end());
        generate(0, candidates, result, item, res_set, 0, target);
        return result;
    }
private:
    void generate(int i, std::vector<int>& nums,
        std::vector<std::vector<int> > &result,
        std::vector<int> &item,
        std::set<std::vector<int> > &res_set,
        int sum, int target){ //sum为当前子集item中的元素和
        if (1){
            return;
        }
        sum += nums[i];
        item.push_back(nums[i]);
        if (2 && res_set.find(item) == res_set.end()){
            result.push_back(item);
            res_set.insert(item);
        }
        generate(i + 1, nums, result, item, res_set, sum, target);
        3
        item.pop_back();
        generate(i + 1, nums, result, item, res_set, sum, target);
    }
};
```

3分钟时间填写代码,
有问题随时提出!

例1-c:实现

```
#include <vector>
#include <set>
#include <algorithm>
class Solution {
public:
    std::vector<std::vector<int>> > combinationSum2 (
        std::vector<int>& candidates,
        int target){
        std::vector<std::vector<int>> > result;
        std::vector<int> item;
        std::set<std::vector<int>> > res_set;
        std::sort(candidates.begin(), candidates.end());
        generate(0, candidates, result, item, res_set, 0, target);
        return result;
    }
private:
    void generate(int i, std::vector<int>& nums,
        std::vector<std::vector<int>> > &result,
        std::vector<int> &item,
        std::set<std::vector<int>> > &res_set,
        int sum, int target){ //sum为当前子集item中的元素和
        if ( i >= nums.size() || sum > target ){
            return; //当元素已选完或item中的元素和sum已超过target
        }
        sum += nums[i];
        item.push_back(nums[i]); //当item中的元素和即为target且该结果未添加时
        if ( target == sum && res_set.find(item) == res_set.end()){
            result.push_back(item);
            res_set.insert(item);
        }
        generate(i + 1, nums, result, item, res_set, sum, target);
        sum -= nums[i]; //回溯后，sum将nums[i]减去并从item中删去
        item.pop_back();
        generate(i + 1, nums, result, item, res_set, sum, target);
    }
};
```

例1-c:测试与leetcode提交结果

```
int main() {
    std::vector<int> nums;
    nums.push_back(10);
    nums.push_back(1);
    nums.push_back(2);
    nums.push_back(7);
    nums.push_back(6);
    nums.push_back(1);
    nums.push_back(5);
    std::vector<std::vector<int>> > result;
    Solution solve;
    result = solve.combinationSum2(nums, 8);
    for (int i = 0; i < result.size(); i++) {
        if (result[i].size() == 0) {
            printf("[]");
        }
        for (int j = 0; j < result[i].size(); j++) {
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Combination Sum II

Submission Details

172 / 172 test cases passed.

Status: **Accepted**

Runtime: 29 ms

Submitted: 0 minutes ago

[1][1][6]

[1][2][5]

[1][7]

[2][6]

请按任意键继续. . .

课间休息10分钟

有问题提出！

例2:生成括号

已知n组括号，开发一个程序，生成这n组括号所有的**合法的**组合可能。

例如:n = 3

结果为: ["((()))", "(())()", "(())()", "()(())", "()()()"]

```
class Solution {  
public:  
    std::vector<std::string> generateParenthesis(int n) {  
    }  
};
```

选自 **LeetCode 22. Generate Parentheses**

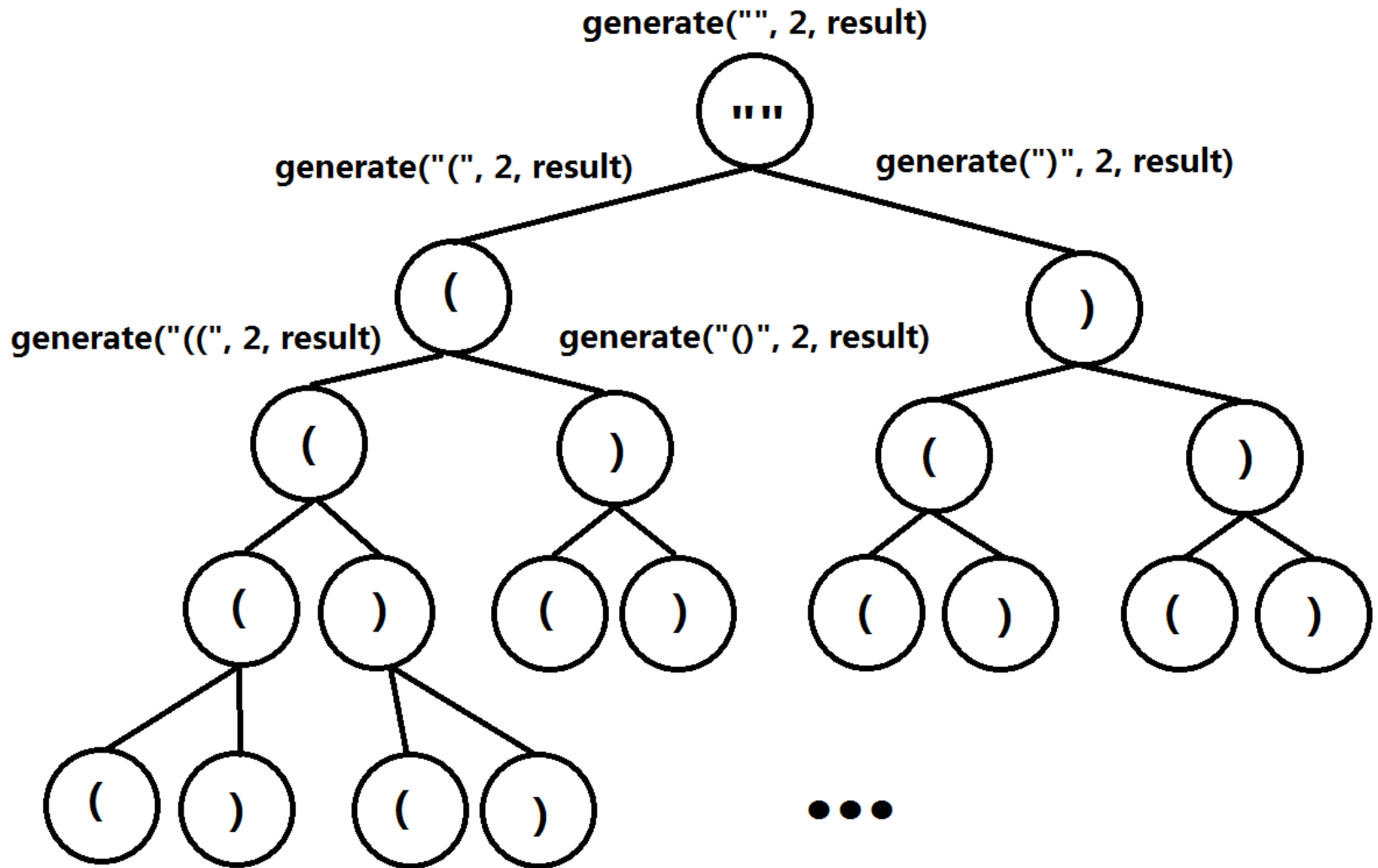
<https://leetcode.com/problems/generate-parentheses/description/>

难度:**Medium**

'C' 'C' 'C' 'C' 'C' 'C' 'C' 'C' 'C'

第二讲 哪些会计科目属于所有者权益科目？

例2:预备知识思考



例2:预备知识:递归生成所有可能

```
#include <stdio.h>
#include <vector>
#include <string>

// item为用来生成的括号字符串, n为组数, result为最终结果

void generate(std::string item, int n,
              std::vector<std::string> &result) {
    if (1) { //当??时, 将item结果push进入result
        result.push_back(item);
        return;
    }
    2
    3
}

int main() {
    std::vector<std::string> result;
    generate("", 2, result); //最初 item 为空字符串
    for (int i = 0; i < result.size(); i++) {
        printf("%s\n", result[i].c_str()); //打印所有结果
    }
    return 0;
}
```

```
'<<<<'
'<<<>'
'<<><'
'<<>>'
'<><<'
'<><>'
'<>><'
'<>>>'
'><<<'
'><<>'
'><><'
'><>>'
'>><<'
'>><>'
'>><>'
'>>><'
'>>>>'
```

例2:预备知识:实现

```
#include <stdio.h>
#include <vector>
#include <string>
```

// item为用来生成的括号字符串， n为组数， result为最终结果

```
void generate(std::string item, int n,
             std::vector<std::string> &result){
```

```
if ( item.size() == 2 * n ) { //当字符串长度是括号组数2倍时，结束递归
    result.push_back(item);
    return;
}
```

```
generate(item + '(', n, result); //添加'('字符，继续递归
```

```
generate(item + ')', n, result); //添加')'字符，继续递归
```

```

}

int main() {
    std::vector<std::string> result;
    generate("", 2, result);
    for (int i = 0; i < result.size(); i++) {
        printf("%s'\n", result[i].c_str());
    }
    return 0;
}

```

' <<<<'
' <<<>'
' <<><'
' <<>>'
' <><<'
' <><>'
' <>><'
' <>>>'
' ><<<'
' ><<>'
' ><><'
' ><>>'
' >><<'
' >><>'
' >>><'
' >>>>'

例2:算法思路

在组成的**所有可能**中，哪些是**合法**的？

- 1.左括号与右括号的数量**不可超过n**。
- 2.每放一个左括号，才可放一个右括号，即右括号**不可先于**左括号放置。

故递归需要**限制条件**：

- 1.左括号与右括号的数量，**最多**放置n个。
- 2.若左括号的数量 \leq 右括号数量，**不可进行**放置右括号的递归。

'((((', 左括号 >2

'((()', 左括号 >2

'()(', 左括号 >2 , 右括号先与左括号放置

'()', 合法

'0()(', 左括号 >2

'00', 合法

'0)()(', 右括号先与左括号放置

'0)))', 右括号 >2

'')((((', 左括号 >2 , 右括号先与左括号放置

'')()(', 右括号先与左括号放置

'')0()(', 右括号先与左括号放置

'')0))', 右括号 >2 , 右括号先与左括号放置

''))(((', 右括号先与左括号放置

''))0()(', 右括号 >2 , 右括号先与左括号放置

''))0)(', 右括号 >2 , 右括号先与左括号放置

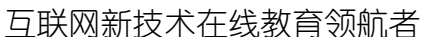
''))0)))', 右括号 >2 , 右括号先与左括号放置

例2:课堂练习

```
#include <string>
#include <vector>
class Solution {
public:
    std::vector<std::string> generateParenthesis(int n) {
        std::vector<std::string> result;
        generate("", n, n, result);
        return result;
    }
private:
    //生成字符串item
    //当前还可以放置左括号的数量left
    //当前可以放置的右括号数量right
    void generate(std::string item, int left, int right,
                  std::vector<std::string> &result) {
        if (1) { //当?? 递归完成, 存储结果
            result.push_back(item);
            return;
        }
        if (left > 0) {
            2
        }
        if (3) { //当??才会放置右括号
            generate(item + ')', left, right - 1, result);
        }
    }
};
```

3分钟时间填写代码,
有问题随时提出!

```
//当前还可以放置左括号的数量left
//当前可以放置的右括号数量right
```



例2:测试与leetcode提交结果

```
int main() {  
    Solution solve;  
    std::vector<std::string> result = solve.generateParenthesis(3);  
    for (int i = 0; i < result.size(); i++) {  
        printf("%s\n", result[i].c_str());  
    }  
    return 0;  
}
```

Generate Parentheses

Submission Details

8 / 8 test cases passed.

Runtime: 3 ms

Status: Accepted

Submitted: 0 minutes ago

<<<>>>

<<><>>

<<>><>

<><<>>

<><><>

请按任意键继续. . .

例3:N皇后

N皇后问题是计算机科学中**最为经典**的问题之一，该问题可**追溯**到1848年，由国际西洋棋棋手马克斯·贝瑟尔于提出了**8皇后**问题。

将N个皇后放摆放在N*N的棋盘中，**互相不可攻击**，有多少种**摆放方式**，每种摆放方式**具体是**怎样的？

```
class Solution {  
public:  
    std::vector<std::vector<std::string> > solveNQueens(int n) {  
    }  
};
```

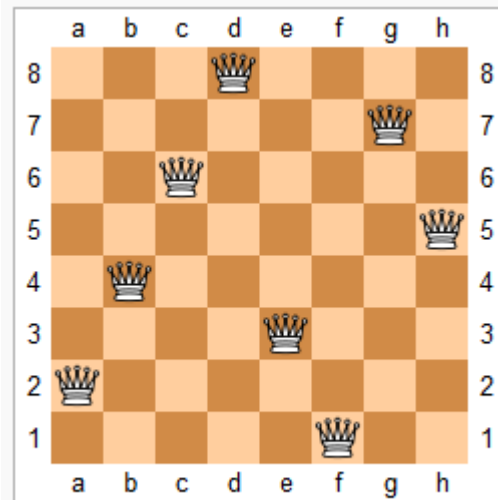
**//传出n皇后的所有结果，每个结果是一个棋盘，
每个棋盘均使用字符串向量表示**

选自 **LeetCode 51. N-Queens**

<https://leetcode.com/problems/n-queens/description/>

难度:**Hard**

```
[  
    [".Q..", // Solution 1  
     "...Q",  
     "Q...",  
     "..Q."],  
  
    [ "..Q.", // Solution 2  
      "Q...",  
      "...Q",  
      ".Q.."]  
]
```



One solution to the eight queens puzzle

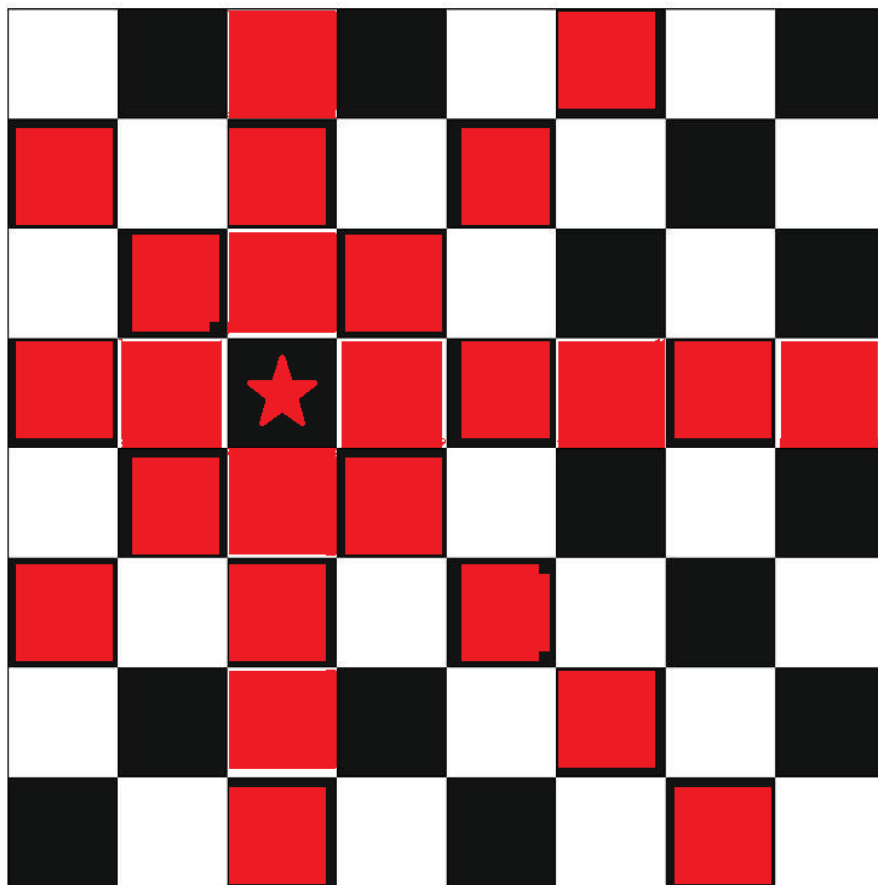
例3:皇后的攻击范围

若在棋盘上已放置一个皇后，它实际上**占据**了哪些位置？

以这个皇后为中心，上、下、左、右、左上、左下、右上、右下，**8个方向**的位置全部**被占据**。

思考：

若在棋盘上放置一个皇后，如右图，标记为**红色位置**即不可再放其他皇后了，如何**设计算法**与**数据存储**，实现这一过程？



例3:棋盘与皇后表示

使用二维数组mark[][]表示一张空棋盘:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

假设在(x, y)位置放置一个皇后,
即数组的第x行, 第y列放置皇后:

如x=4, y=3;第4行, 第3列

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

设置方向数组:

(x-1, y-1) (x-1, y) (x-1, y+1)

(x, y-1) **(x, y)** (x, y+1)

(x+1, y-1) (x+1, y) (x+1, y+1)

```
static const int dx[] = {-1, 1, 0, 0, -1, -1, 1, 1};  
static const int dy[] = {0, 0, -1, 1, -1, 1, -1, 1};
```

按照方向数组的8个方向分别延伸N个距离, 只要不超过边界,

mark[][] = 1

0	0	1	0	0	1	0	0
1	0	1	0	1	0	0	0
0	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1
0	1	1	1	0	0	0	0
1	0	1	0	1	0	0	0
0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	0

例3:课堂练习，实现

//第x行，y列放置皇后，mark[行][列]表示一张棋盘

```
void put_down_the_queen(int x, int y,
                        std::vector<std::vector<int>> &mark) {

    static const int dx[] = {-1, 1, 0, 0, -1, -1, 1, 1};
    static const int dy[] = {0, 0, -1, 1, -1, 1, -1, 1}; //方向数组

    mark[x][y] = 1; // (x, y)放置皇后 进行标记

    for (int i = 1; i < mark.size(); i++) { //8个方向，每个方向向外延伸1至N-1
        for (int j = 0; j < 8; j++) {

            int new_x = 
            int new_y = 

            if (  ) {

                mark[new_x][new_y] = 1;

            }

        }

    }

}
```

3分钟时间填写代码，
有问题随时提出！

例3:实现

//第x行，y列放置皇后，mark[行][列]表示一张棋盘

```
void put_down_the_queen(int x, int y,
                        std::vector<std::vector<int>> &mark) {

    static const int dx[] = {-1, 1, 0, 0, -1, -1, 1, 1};
    static const int dy[] = {0, 0, -1, 1, -1, 1, -1, 1}; //方向数组

    mark[x][y] = 1; // (x, y)放置皇后 进行标记

    for (int i = 1; i < mark.size(); i++) { //8个方向，每个方向向外延伸1至N-1
        for (int j = 0; j < 8; j++) {

            int new_x = x + i * dx[j]; //新的位置向8个方向延伸，
            int new_y = y + i * dy[j]; 每个方向最多延伸N-1

            if ( new_x >= 0 && new_x < mark.size() //检查新位置是否还在棋盘内
                && new_y >= 0 && new_y < mark.size() ) {
                mark[new_x][new_y] = 1;
            }
        }
    }
}
```

例3:回溯算法

N皇后问题，对于N*N的棋盘，**每行**都要放置1个且**只能放置1**个皇后。

利用**递归**对棋盘的**每一行**放置皇后，放置时，按**列顺序**寻找可以放置皇后的列，若可以放置皇后，将皇后放置该位置，并**更新mark标记数组**，**递归进行**下一行的皇后放置；当该次递归结束后，**恢复mark数组**，并**尝试**下一个可能放皇后的列。

当递归可以完成N行的**N个皇后**放置，则将该结果**保存并返回**。

假设棋盘上已经放了红色的Q，
又放了个蓝色的Q

绿色的Q一定放在第三行
有三种可能

可能放在第一个位置，
放好后递归进行下一行的尝试

1	1	1	Q	1	1	1	1	1	1	1	Q	1	1	1	1	1	1	1	Q	1	1	1	1
1	1	1	1	1	1	Q	1	1	1	1	1	1	1	Q	1	1	1	1	1	1	1	Q	1
0	1	0	1	0	1	1	1	0	1	0	1	0	1	1	1	Q	1	1	1	1	1	1	1
1	0	0	1	1	0	1	0	1	0	0	1	1	0	1	0	1	1	0	1	1	0	1	0
0	0	0	1	0	0	1	1	0	0	0	1	0	0	1	1	1	0	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	0	1	1	0	0	1	0	1	0	1	1	0	0	1	0
0	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	1	0	1	1	0	1	0
1	0	0	1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0	1	1	0

例3:回溯算法-4皇后举例

初始化: 0列 1列 2列 3列	递归第0行 尝试第0列	递归第1行 尝试第2列	递归第2行 哪列都没法放!	回溯第1行 尝试第3列	递归第2行 尝试第1列
0行: 0 0 0 0	Q 1 1 1	Q 1 1 1	Q 1 1 1	Q 1 1 1	Q 1 1 1
1行: 0 0 0 0	1 1 0 0	1 1 Q 1	1 1 Q 1	1 1 1 Q	1 1 1 Q
2行: 0 0 0 0	1 0 1 0	1 1 1 1	1 1 1 1	1 0 1 1	1 Q 1 1
3行: 0 0 0 0	1 0 0 1	1 0 1 1	1 0 1 1	1 1 0 1	1 1 1 1
递归第3行 哪列都没法放!	一直回溯到第0行 尝试第1列	递归第1行 尝试第3列	递归第2行 尝试第0列	递归第3行 尝试第2列	递归第4行 当行数为N时 即找到结果!
Q 1 1 1 1 1 1 Q 1 Q 1 1 1 1 1 1	1 Q 1 1 1 1 1 0 0 1 0 1 0 1 0 0	1 Q 1 1 1 1 1 Q 0 1 1 1 0 1 0 1	1 Q 1 1 1 1 1 Q Q 1 1 1 1 1 0 1	1 Q 1 1 1 1 1 Q Q 1 1 1 1 1 Q 1	1 Q 1 1 1 1 1 Q Q 1 1 1 1 1 Q 1

例3:调用代码实现

```
class Solution {
public:
    std::vector<std::vector<std::string> > solveNQueens(int n) {
        std::vector<std::vector<std::string> > result; //存储最终结果的数组
        std::vector<std::vector<int> > mark; //标记棋盘是否可以放置皇后的二维数组
        std::vector<std::string> location; //存储某个摆放结果，当完成一次递归找到结果后，
        for (int i = 0; i < n; i++) {
            mark.push_back((std::vector<int>())); //将location push进入result
            for (int j = 0; j < n; j++) {
                mark[i].push_back(0);
            }
            location.push_back("");
            location[i].append(n, '.');
        }
        generate(0, n, location, result, mark);
        return result;
    }
private:
    void put_down_the_queen(int x, int y,
        std::vector<std::vector<int> > &mark) {
        ● ● ●
    }
}
```

如 n = 4,

location初始化:

●	●	●	●	0	0	0	0
●	●	●	●	0	0	0	0
●	●	●	●	0	0	0	0
●	●	●	●	0	0	0	0

mark初始化:

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

某4皇后中间结果:

location:

●	Q	●	●
●	●	●	Q
●	●	●	●
●	●	●	●

mark:

1	1	1	1
1	1	1	1
0	1	1	1
0	1	0	1

例3:课堂练习

```
void generate(int k, int n, //k 代表完成了几个皇后的放置(正在放置第k行皇后)
             std::vector<std::string> &location, //某次结果存储在location中
             std::vector<std::vector<std::string> > &result, //最终结果存储在result
             std::vector<std::vector<int> > &mark) { //表示棋盘的标记数组

    if ( 1 ) {
        2 //当??时, 结束递归, 结束前需
        return; //要??

    }
    //按顺序尝试第0至第n-1列
    for (int i = 0; i < n; i++) {
        if ( 3 ) { //如果当前满足??条件, 即可以放置皇后
            std::vector<std::vector<int> > tmp_mark = mark; //记录回溯前的mark镜像
            4
            put_down_the_queen(k, i, mark); //放置皇后
            generate(k + 1, n, location, result, mark); //递归下一行皇后放置
            5 //回溯后要做??事
            location[k][i] = '.';
        }
    }
}

};
```

例3:实现

```
void generate(int k, int n, //k 代表完成了几个皇后的放置(正在放置第k行皇后)
             std::vector<std::string> &location, //某次结果存储在location中
             std::vector<std::vector<std::string> > &result, //最终结果存储在result
             std::vector<std::vector<int> > &mark) { //表示棋盘的标记数组
```

```
if (k == n) {
```

//当k == n时，代表完成了第0至第n-1行

```
result.push_back(location);
```

皇后的放置，所有皇后完成放置后，将记录皇后位置的

```
return;
```

location数组push进入result

```
}
```

```
for (int i = 0; i < n; i++) {
```

//按顺序尝试第0至第n-1列

```
if (mark[k][i] == 0) {
```

// 如果mark[k][i]==0，即可以放置皇后

```
std::vector<std::vector<int> > tmp_mark = mark; //记录回溯前的mark镜像
```

```
location[k][i] = 'Q';
```

//记录当前皇后的位置

```
put_down_the_queen(k, i, mark); //放置皇后
```

```
generate(k + 1, n, location, result, mark); //递归下一行皇后放置
```

```
mark = tmp_mark;
```

//将mark重新赋值为回溯前状态

```
location[k][i] = '.';
```

//将当前尝试的皇后位置重新置

```
}
```

```
}
```

```
}
```

```
};
```

k = 1 回溯前:

尝试时:

location:

```
Q . . .
. . . .
. . . .
. . . .
```

mark(tmp_mark):

```
1 1 1 1
1 1 0 0
1 0 1 0
1 0 0 1
```

location:

```
Q . . .
. . Q .
. . . .
. . . .
```

mark:

```
1 1 1 1
1 1 1 1
1 1 1 1
1 0 1 1
```


例3:测试与leetcode提交结果

```
int main() {
    std::vector<std::vector<std::string> > result;
    Solution solve;
    result = solve.solveNQueens(4);
    for (int i = 0; i < result.size(); i++) {
        printf("i = %d\n", i);
        for (int j = 0; j < result[i].size(); j++) {
            printf("%s\n", result[i][j].c_str());
        }
        printf("\n");
    }
    return 0;
}
```

N-Queens

Submission Details

9 / 9 test cases passed.

Status: **Accepted**

Runtime: 19 ms

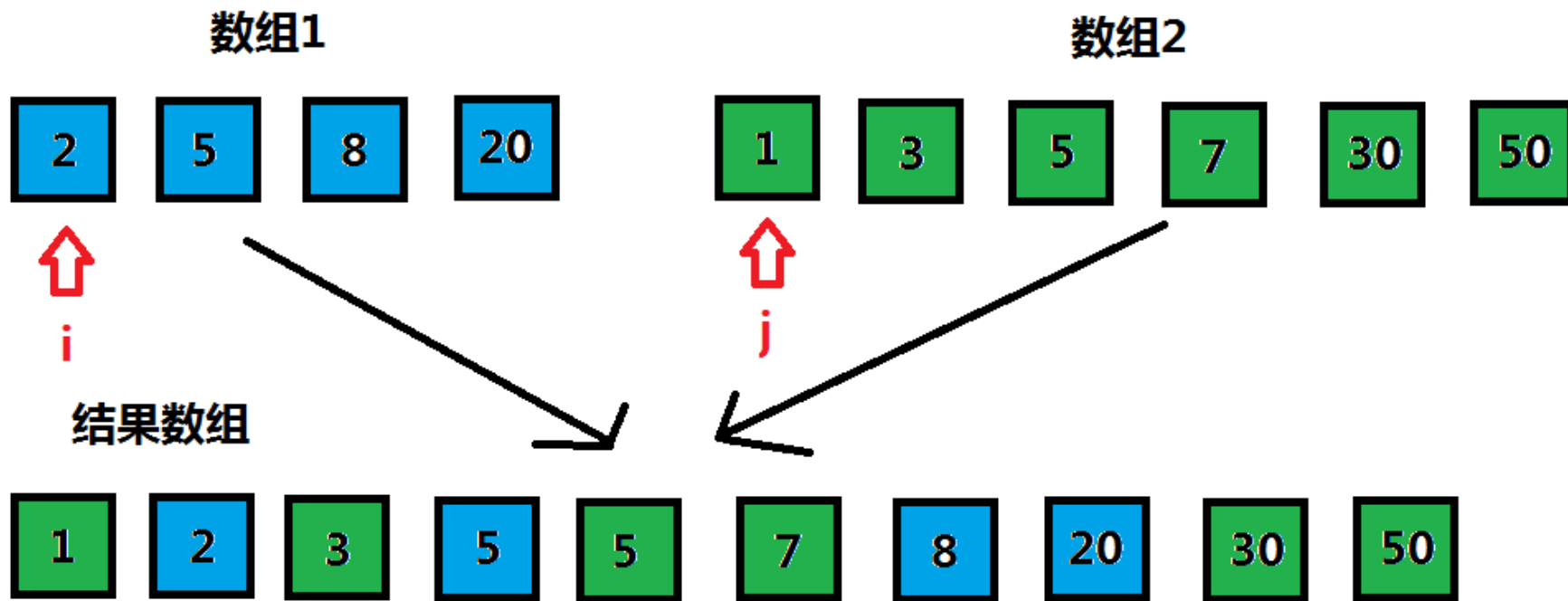
Submitted: 0 minutes ago

```
i = 0
.Q..
...Q
Q...
..Q.

i = 1
..Q.
Q...
...Q
.Q..
```

预备知识:归并两个已排序数组

已知两个已**排序**数组，将这两个数组**合并**为一个排序数组。



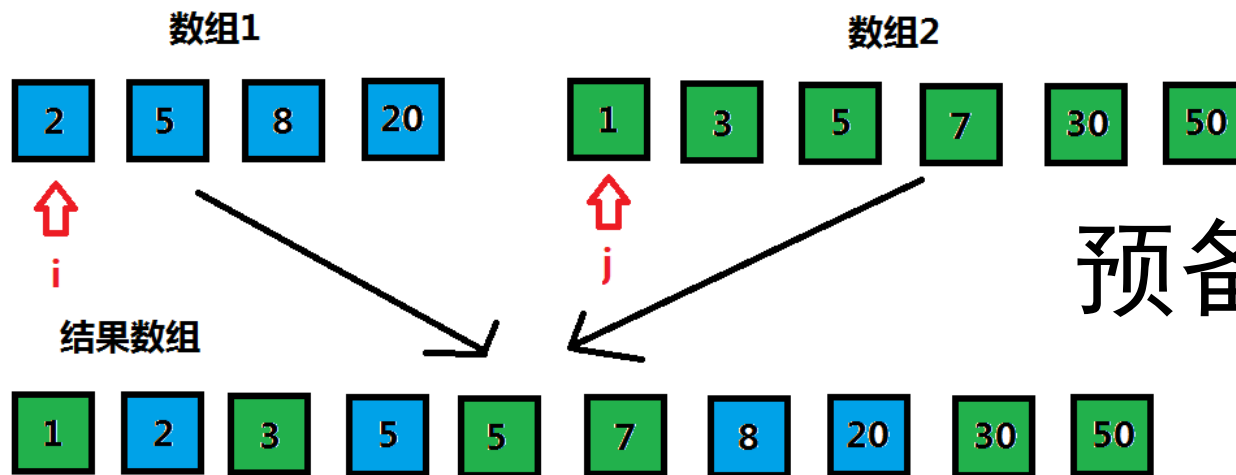
预备知识:课堂练习

```
#include <vector>

void merge_sort_two_vec(std::vector<int> &sub_vec1, //数组1
                        std::vector<int> &sub_vec2, //数组2
                        std::vector<int> &vec) { //合并后的数组

    int i = 0;
    int j = 0;
    while (1) {
        if (2) {
            vec.push_back(sub_vec1[i]);
            i++;
        }
        else{
            vec.push_back(sub_vec2[j]);
            3
        }
    }
    for (; i < sub_vec1.size(); i++){
        vec.push_back(sub_vec1[i]);
    }
    for (; j < sub_vec2.size(); j++){
        vec.push_back(sub_vec2[j]);
    }
}
```

1分钟时间填写代码，
有问题随时提出！



预备知识:实现

```
void merge_sort_two_vec(std::vector<int> &sub_vec1, //数组1
                        std::vector<int> &sub_vec2, //数组2
                        std::vector<int> &vec) { //合并后的数组

    int i = 0;
    int j = 0;
    while (i < sub_vec1.size() && j < sub_vec2.size()) {
        if (sub_vec1[i] <= sub_vec2[j]) {
            vec.push_back(sub_vec1[i]);
            i++;
        }
        else {
            vec.push_back(sub_vec2[j]);
            j++;
        }
    }
    for (; i < sub_vec1.size(); i++) {
        vec.push_back(sub_vec1[i]);
    }
    for (; j < sub_vec2.size(); j++) {
        vec.push_back(sub_vec2[j]);
    }
}
```

//将sub_vec1或sub_vec2
中的剩余元素push进入vec

预备知识:测试

```
int main() {
    int test1[] = {2, 5, 8, 20};
    int test2[] = {1, 3, 5, 7, 30, 50};
    std::vector<int> sub_vec1;
    std::vector<int> sub_vec2;
    std::vector<int> vec;
    for (int i = 0; i < 4; i++) {
        sub_vec1.push_back(test1[i]);
    }
    for (int i = 0; i < 6; i++) {
        sub_vec2.push_back(test2[i]);
    }
    merge_sort_two_vec(sub_vec1, sub_vec2, vec);
    for (int i = 0; i < vec.size(); i++) {
        printf("[%d]", vec[i]);
    }
    printf("\n");
    return 0;
}
```

```
[1][2][3][5][5][7][8][20][30][50]
请按任意键继续. . .
```

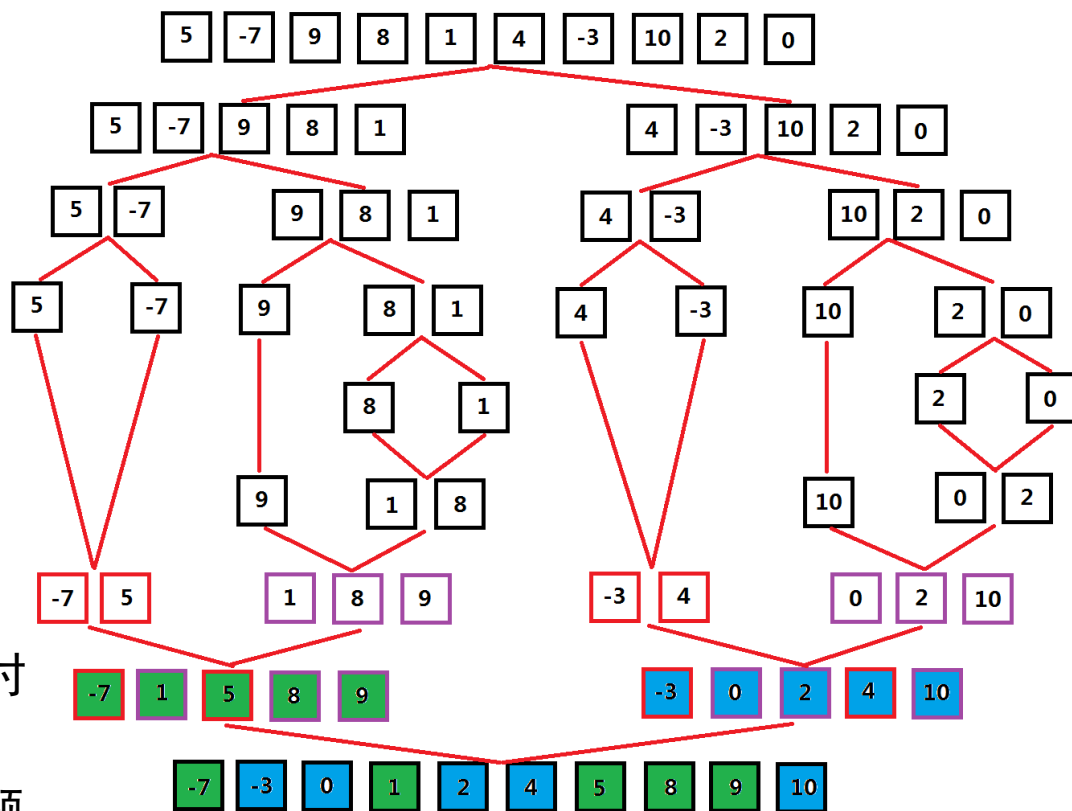
预备知识:分治算法之归并排序

分治算法:

将一个规模为N的问题**分解**为K个规模较小的子问题，这些子问题**相互独立**且与原问题**性质相同**。求出子问题的解后进行合并，就可得到原问题的解。

一般步骤:

- 1.分解**，将要解决的问题划分成**若干规模较小**的同类问题；
- 2.求解**，当子问题划分得**足够小**时，用较简单的方法解决；
- 3.合并**，按原问题的要求，将子问题的解**逐层合并**构成原问题的解。



预备知识:归并排序复杂度分析

设有 **n 个元素**， n 个元素**归并排序**的时间 **$T(n)$**

总时间 = 分解时间 + 解决子问题时间 + 合并时间

分解时间: 即对**原问题拆解**为两个子问题的时间 复杂度 **$O(n)$**

解决子问题时间: 即解决**两个子问题**的时间 **$2T(n/2)$**

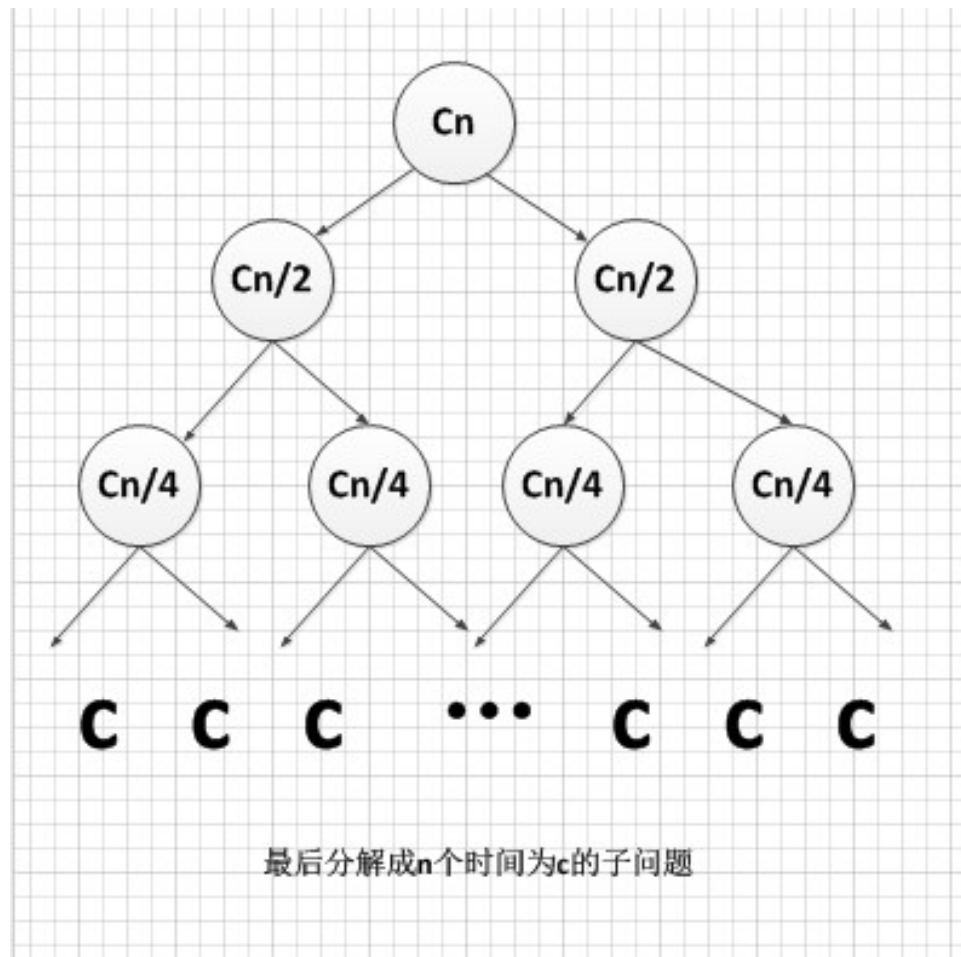
合并时间: 即对两个**已排序数组归并**的时间 复杂度 **$O(n)$**

$$T(n) = 2T(n/2) + 2O(n)$$

$$= 2T(n/2) + O(n)$$

$$= O(n + 2*n/2 + 4*n/4 + \dots + n*1)$$

$$= \mathbf{O(n \log n)}$$



预备知识:归并排序课堂练习

```
void merge_sort_two_vec(std::vector<int> &sub_vec1, //数组1
                        std::vector<int> &sub_vec2, //数组2
                        std::vector<int> &vec) { //合并后的数组
```

...

```
}
void merge_sort(std::vector<int> &vec) {
    if (1) {
        return; //求解:子问题足够小时, 直接求解
    }
```

```
    int mid = vec.size() / 2;
    std::vector<int> sub_vec1;
    std::vector<int> sub_vec2;
    for (int i = 0; 2; i++) {
        sub_vec1.push_back(vec[i]);
    }
    for (int i = mid; i < vec.size(); i++) {
        sub_vec2.push_back(vec[i]);
    }
```

```
    merge_sort(sub_vec1);
    merge_sort(sub_vec2); //对拆解后的两个子问题进行求解
    vec.clear();
```

3

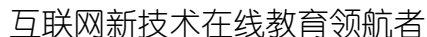
//合并, 将子问题的解进行合并

1分钟时间填写代码,
有问题随时提出!

● ● ●

```
return; //求解:子问题足够小时, 直接求解
```

```
merge_sort_two_vec(sub_vec1, sub_vec2, vec); //合并, 将子问题的解进行合并
```



预备知识:归并排序测试

```
int main(){
    std::vector<int> vec;
    int test[] = {5, -7, 9, 8, 1, 4, -3, 10, 2, 0};
    for (int i = 0; i < 10; i++){
        vec.push_back(test[i]);
    }
    merge_sort(vec);
    for (int i = 0; i < vec.size(); i++){
        printf("[%d]", vec[i]);
    }
    printf("\n");
    return 0;
}
```

[-7][-3][0][1][2][4][5][8][9][10]
请按任意键继续. . .

```
#include <stdlib.h>
#include <algorithm>
#include <assert.h>
```

**//生成随机数组，
利用std::sort测试归并排序**

```
int main(){
    std::vector<int> vec1;
    std::vector<int> vec2;
    srand(time(NULL));
    for (int i = 0; i < 10000; i++){
        int num = (rand() * rand()) % 100003;
        vec1.push_back(num);
        vec2.push_back(num);
    }
    merge_sort(vec1);
    std::sort(vec2.begin(), vec2.end());
    assert(vec1.size() == vec2.size());
    for (int i = 0; i < vec1.size(); i++){
        assert(vec1[i] == vec2[i]);
    }
    return 0;
}
```

**//将同样的随机数，
push进入vec1和vec2**

//对vec1调用归并排序

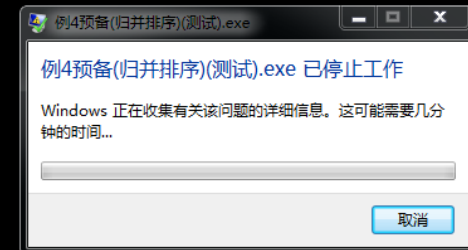
//对vec2调用库函数std::sort

//排序后，利用断言按顺序比较每个元素

请按任意键继续. . .

Assertion failed: vec1[i] == vec2[i], file C:\Users\B01\Desktop\BAT面试题
法冲刺班_第四课_递归回溯与分制_林沐_2017_10_19\例4预备<归并排序><测试>.cpp, line
63

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.



例4:逆序数

已知数组nums，求**新数组count**，count[i]代表了**在nums[i]右侧且比nums[i]小**的元素个数。

例如：

nums = [5, 2, 6, 1], count = [2, 1, 1, 0];

nums = [6, 6, 6, 1, 1, 1], count = [3, 3, 3, 0, 0, 0];

nums = [5, -7, 9, 1, 3, 5, -2, 1], count = [5, 0, 5, 1, 2, 2, 0, 0];

```
class Solution {  
public:  
    std::vector<int> countSmaller(std::vector<int>& nums) {  
    }  
};
```

选自 **LeetCode 315. Count of Smaller Numbers After Self**

<https://leetcode.com/problems/count-of-smaller-numbers-after-self/description/>

难度:**Hard**

例4:思考

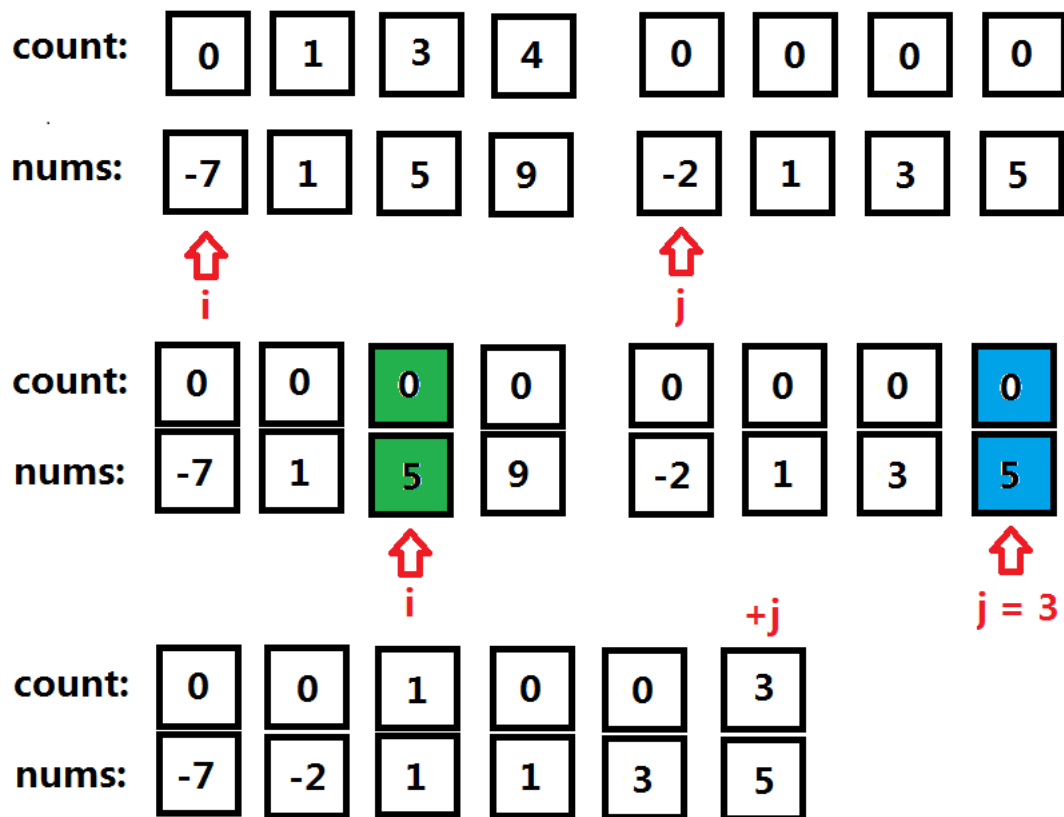
最暴力的方法，即对**每个元素**扫描其右侧比它小的数，**累加个数**。假设数组元素个数为N，**算法复杂度** $O(N^2)$ 。

观察如下数组，该数组**前4个元素有序**，**后4个元素有序**，是否有更好的方法计算count数组？

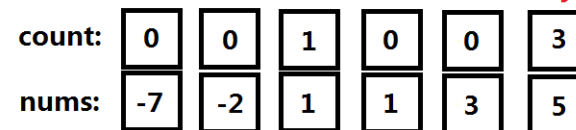
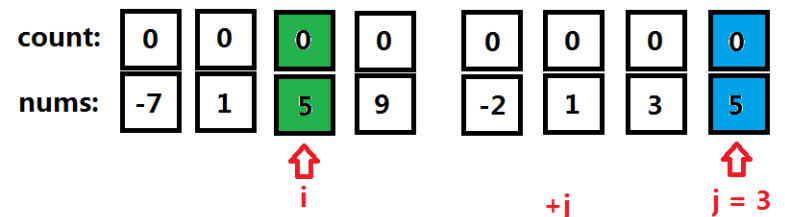
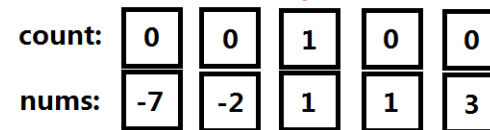
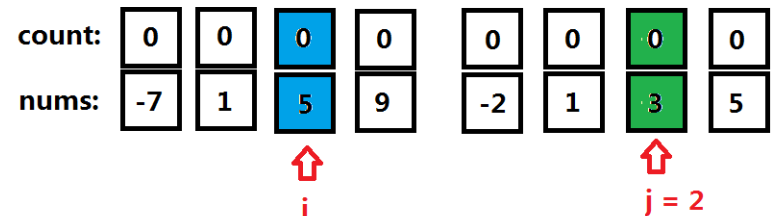
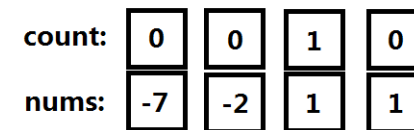
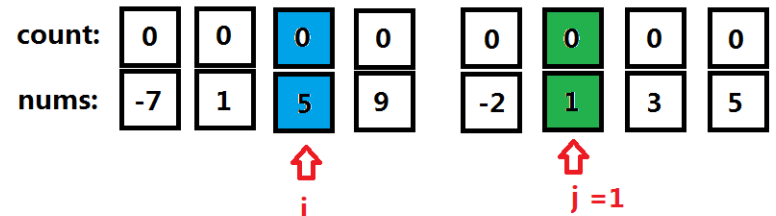
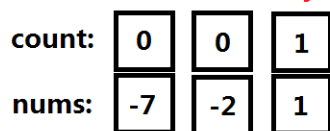
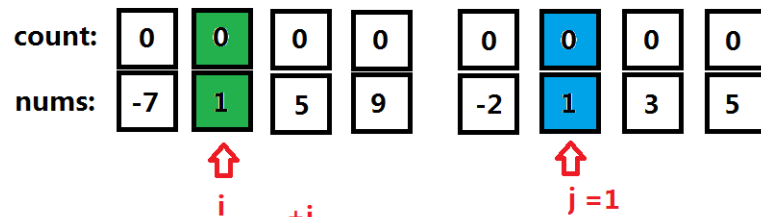
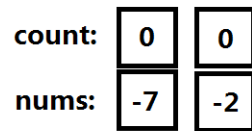
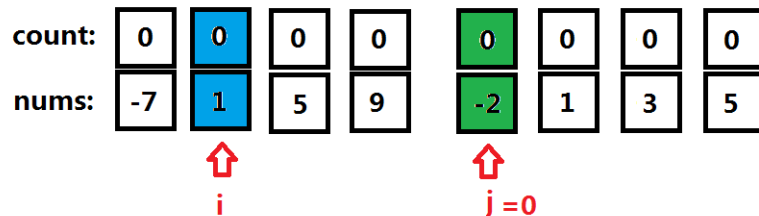
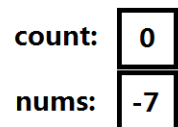
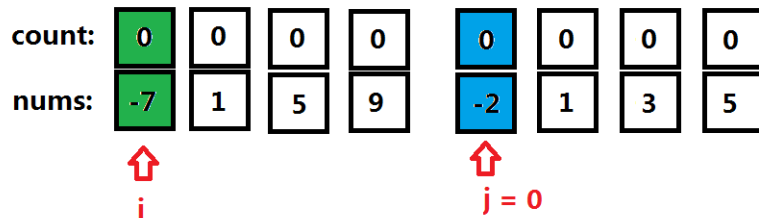
count:	<div>0</div>	<div>1</div>	<div>3</div>	<div>4</div>	<div>0</div>	<div>0</div>	<div>0</div>	<div>0</div>
nums:	<div>-7</div>	<div>1</div>	<div>5</div>	<div>9</div>	<div>-2</div>	<div>1</div>	<div>3</div>	<div>5</div>
	<div>↑</div>				<div>↑</div>			
	<div>i</div>				<div>j</div>			

例4:算法思路



在**归并**两排序数组时，当需要将前一个数组元素的**指针i**指向的元素**插入**时，对应的count[i]，即为指向后一个数组的**指针j**的值。





例4:算法思路



例4:算法思路

count:	0	0	0	0	0	0	0
nums:	-7	1	5	9	-2	1	3
							
			i				j = 3

count:	0	0	1	0	0	3	0
nums:	-7	-2	1	1	3	5	5

count:	0	0	0	0	0	0	0
nums:	-7	1	5	9	-2	1	3
							
			i			+j	j = 4

count:	0	0	1	0	0	3	0
nums:	-7	-2	1	1	3	5	5

例4:算法思路

期望的count结果，与排序前原数组元素可对应上:

count:	0	1	3	4	0	0	0	0
nums:	-7	1	5	9	-2	1	3	5

排序的结果:

count:	0	0	1	0	0	3	0	4
nums:	-7	-2	1	1	3	5	5	9

如何解决？

例4:算法思路

count:	0	0	1	0	0	3	0	4
nums:	-7	-2	1	1	3	5	5	9

元素位置:	0	1	2	3	4	5	6	7
count:	0	1	3	4	0	0	0	0
nums:	-7	1	5	9	-2	1	3	5

将元素nums[i]与元素的位置i绑定为pair 如: $\langle \text{nums}[i], i \rangle$

排序时, 按照nums[i]大小排序pair对, 利用pair对 $\langle \text{nums}[i], i \rangle$ 中的i更新count数组。

$\langle -7, 0 \rangle$	$\langle 1, 1 \rangle$	$\langle 5, 2 \rangle$	$\langle 9, 3 \rangle$	$\langle -2, 4 \rangle$	$\langle 1, 5 \rangle$	$\langle 3, 6 \rangle$	$\langle 5, 7 \rangle$
count[2] += j		↑ i					↑ j = 3
count[2] = 3							

例4:算法思路

元素位置 i:

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

元素nums[i]:

5	-7	9	1	3	5	-2	1
---	----	---	---	---	---	----	---

绑定: <nums[i],i>

<5,0>	<-7,1>	<9,2>	<1,3>	<3,4>	<5,5>	<-2,6>	<1,7>
-------	--------	-------	-------	-------	-------	--------	-------

count:

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

<5,0>	<-7,1>	<9,2>	<1,3>	<3,4>	<5,5>	<-2,6>	<1,7>
-------	--------	-------	-------	-------	-------	--------	-------

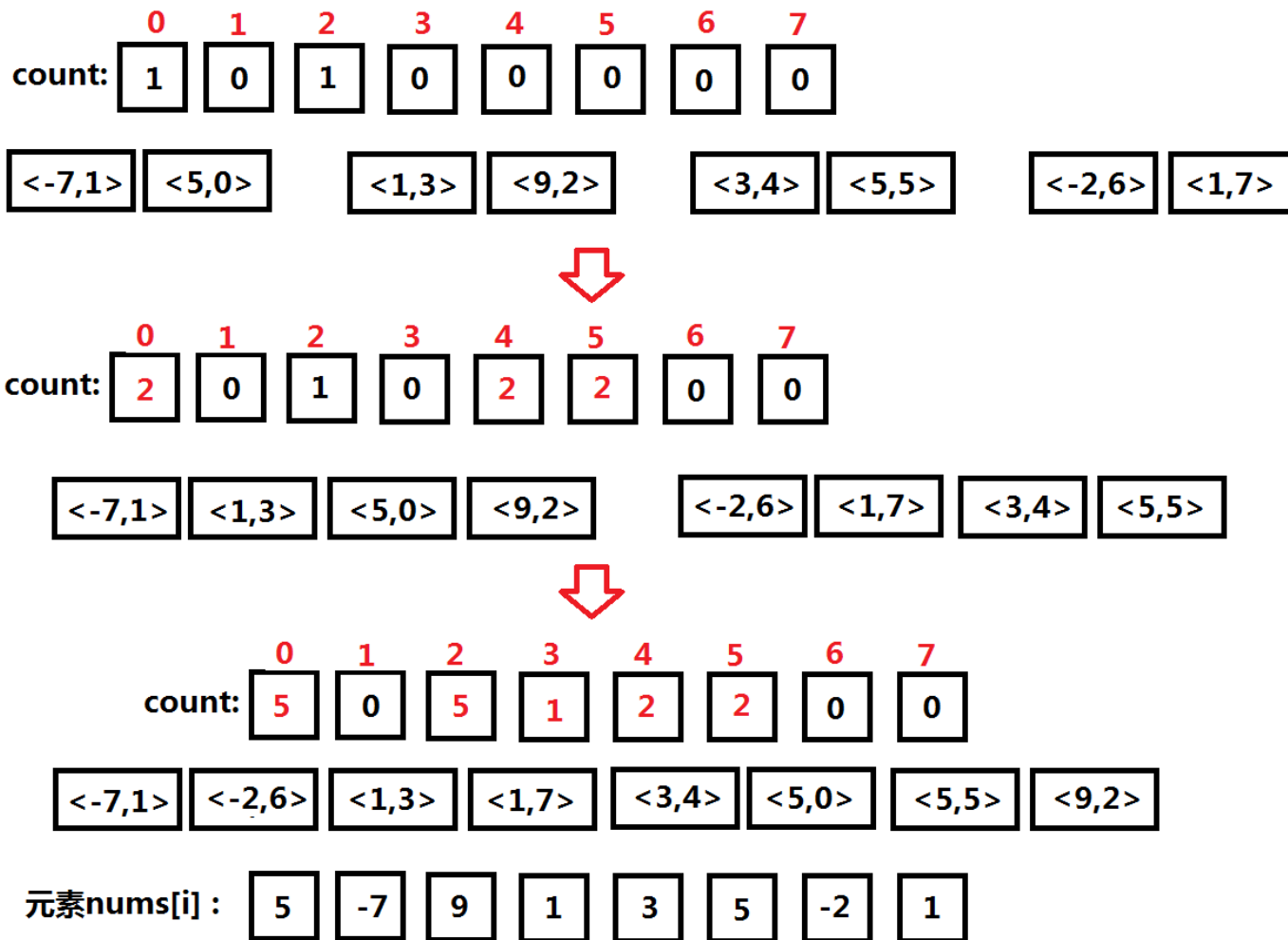


count:

0	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0

<-7,1>	<5,0>	<1,3>	<9,2>	<3,4>	<5,5>	<-2,6>	<1,7>
--------	-------	-------	-------	-------	-------	--------	-------

例4:算法思路



例4:课堂练习

```
#include <vector>
class Solution {
public:
    std::vector<int> countSmaller(std::vector<int>& nums) {
        std::vector<std::pair<int, int> > vec;
        std::vector<int> count;
        for (int i = 0; i < nums.size(); i++){
            vec.push_back(std::make_pair(nums[i], i));
            count.push_back(0);
        }
        merge_sort(vec, count);
        return count;
    }
private:
    void merge_sort_two_vec(
        std::vector<std::pair<int, int> > &sub_vec1,
        std::vector<std::pair<int, int> > &sub_vec2,
        std::vector<std::pair<int, int> > &vec,
        std::vector<int> &count) {
        ● ● ●
    }

    void merge_sort(std::vector<std::pair<int, int> > &vec,
        std::vector<int> &count) {
        if (vec.size() < 2) {
            return;
        }
        int mid = vec.size() / 2;
        std::vector<std::pair<int, int> > sub_vec1;
        std::vector<std::pair<int, int> > sub_vec2;
        for (int i = 0; i < mid; i++){
            sub_vec1.push_back(vec[i]);
        }
        for (int i = mid; i < vec.size(); i++){
            sub_vec2.push_back(vec[i]);
        }
        merge_sort(sub_vec1, count);
        merge_sort(sub_vec2, count);
        vec.clear();
        merge_sort_two_vec(sub_vec1, sub_vec2, vec, count);
    }
};
```

//将nums[i]与i绑定为pair<nums[i], i>

//子问题足够小时，直接求解

//对原问题进行分解，即对原数组拆分为两个规模相同的数组，再对它们分别求解(排序)

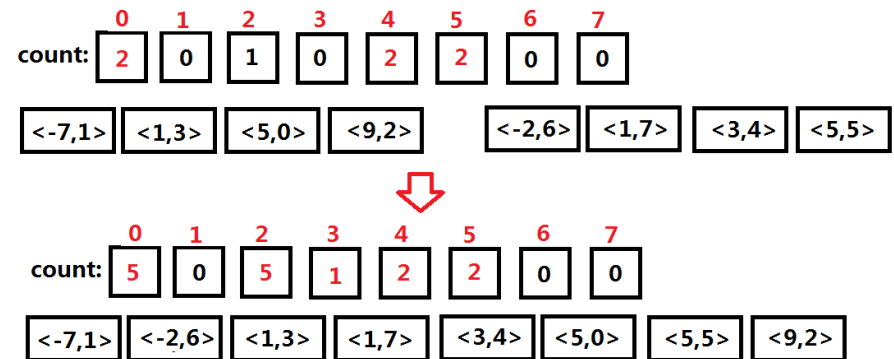
//对拆解后的两个子问题进行求解

例4:课堂练习

3分钟时间填写代码，
有问题随时提出！

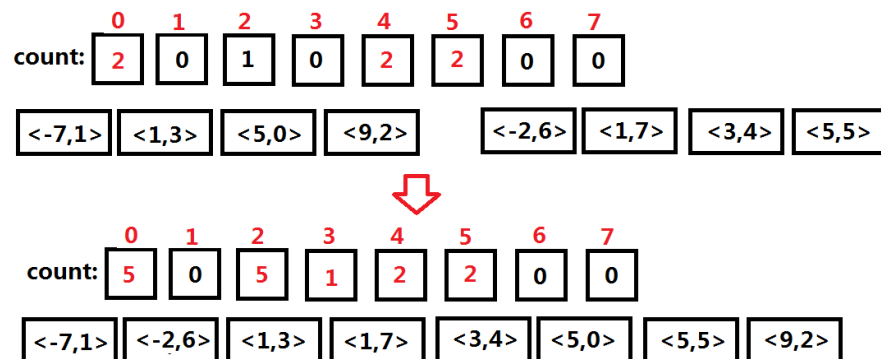
```
void merge_sort_two_vec(
    std::vector<std::pair<int, int> > &sub_vec1,
    std::vector<std::pair<int, int> > &sub_vec2,
    std::vector<std::pair<int, int> > &vec,
    std::vector<int> &count) {

    int i = 0;
    int j = 0;
    while(i < sub_vec1.size() && j < sub_vec2.size()){
        if ( 1 ) {
            2
            vec.push_back(sub_vec1[i]);
            i++;
        }
        else{
            vec.push_back(sub_vec2[j]);
            j++;
        }
    }
    for (; i < sub_vec1.size(); i++){
        3
        vec.push_back(sub_vec1[i]);
    }
    for (; j < sub_vec2.size(); j++){
        vec.push_back(sub_vec2[j]);
    }
}
```



例4:实现

```
void merge_sort_two_vec(  
    std::vector<std::pair<int, int> > &sub_vec1,  
    std::vector<std::pair<int, int> > &sub_vec2,  
    std::vector<std::pair<int, int> > &vec,  
    std::vector<int> &count) {  
  
    int i = 0;  
    int j = 0;  
    while (i < sub_vec1.size() && j < sub_vec2.size()) {  
        if (sub_vec1[i].first <= sub_vec2[j].first) {  
            count[sub_vec1[i].second] += j;  
            vec.push_back(sub_vec1[i]);  
            i++;  
        }  
        else {  
            vec.push_back(sub_vec2[j]);  
            j++;  
        }  
    }  
    for (; i < sub_vec1.size(); i++) {  
        count[sub_vec1[i].second] += j;  
        vec.push_back(sub_vec1[i]);  
    }  
    for (; j < sub_vec2.size(); j++) {  
        vec.push_back(sub_vec2[j]);  
    }  
}
```



例4:测试与leetcode提交结果

```
int main() {
    int test[] = {5, -7, 9, 1, 3, 5, -2, 1};
    std::vector<int> nums;
    for (int i = 0; i < 8; i++) {
        nums.push_back(test[i]);
    }
    Solution solve;
    std::vector<int> result = solve.countSmaller(nums);
    for (int i = 0; i < result.size(); i++) {
        printf("[%d]", result[i]);
    }
    printf("\n");
    return 0;
}
```



```
[5][0][5][1][2][2][0][0]
请按任意键继续. . .
```

Count of Smaller Numbers After Self

Submission Details

16 / 16 test cases passed.

Status: **Accepted**

Runtime: 36 ms

Submitted: 0 minutes ago

结束

非常感谢大家！

林沐