

法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。



关注 小象学院

第五课 算法设计提高

林沐

内容概述

- 1.二分查找的基本算法，循环与递归实现
- 2.例1-区间查找(二分查找变型)
- 3.例2-岛屿数量(递归深度搜索的应用)
- 4.例3-求子集(回溯法与位运算法的应用)

算法设计提高

算法复杂度是指算法在编写成可执行程序后，运行时所需要**时间资源**和**内存资源**。

时间资源的开销由程序的**时间复杂度**描述，它一般定义为 $T(n)$ 是所求解问题**规模 n** 的函数。当问题的规模 n 趋向**无穷大**时，时间复杂度 $T(n)$ 的数量级称为**渐进时间复杂度**，记作 **$O(n)$** 。一般我们讨论的算法时间复杂度，就是 $O(n)$ 。例如， $O(n^2+n+5) = O(n^2)$

$O(n)$ 与问题规模相关的**循环**直接相关。我们利用交换排序算法来计算**数组的排序**，因为这个算法需要两层与 n 相关的循环实现，所以时间复杂度为 $O(n^2)$ ， n 为**数组的长度**(即问题的规模)。

如果算法的执行时间**不随着问题规模 n** 的增加而增长，即使算法中有上千条语句，其执行时间也不过是一个较大的常数。此类算法的时间复杂度是 $O(1)$ 。

内存资源的开销由程序的**空间复杂度**描述，程序在当今的个人电脑或服务器下运行，一般**不太需要**考虑内存开销(比起时间开销，内存开销基本不值一提)。

二分查找，问题引入

已知一个**排序数组**A，如 $A = [-1, 2, 5, 20, 90, 100, 207, 800]$ ，
另外一个**乱序数组**B，如 $B = [50, 90, 3, -1, 207, 80]$ ，
求B中的任意某个元素，是否在A中出现，**结果**存储在数组C中，**出现用1**代表，**未出现用0**代表，如， $C = [0, 1, 0, 1, 1, 0]$ 。

```
//返回结果数组元素个数    //排序数组        //排序数组长度
int search_array(int sort_array[], int s_n,
                //乱序数组        //乱序数组长度
                int random_array[], int r_n,
                //结果数组
                int result[]) {
```

思考，最暴力的方法是什么？**时间复杂度**是什么？有没有**更快**的方法？

二分查找算法

二分查找 又称 **折半查找**，首先，假设数组中元素是按**升序排列**，将数组**中间位置**的关键字与查找关键字比较：

- 1.如果两者**相等**，则**查找成功**；
- 2.否则利用**中间位置**将表分成**前、后**两个子数组：
 - 1)如果中间位置的关键字**大于**查找关键字，则进一步查找**前一子数组**
 - 2)否则进一步查找**后一子数组**

重复以上过程，**直到**找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

二分查找的**时间复杂度** $\log(n)$ ，**整体时间**复杂度 $n\log(n)$

例如，待搜索数字 $target == 2, 200$

数组 $A = [-1, 2, 5, 20, 90, 100, 207, 800]$

二分查找算法，举例1

待查找值 target = 2

下标: [0, 1, 2, 3, 4, 5, 6, 7]

nums: [-1, 2, 5, 20, 90, 100, 207, 800]

第1次搜索:

搜索区间:[0, 7] [-1, 2, 5, 20, 90, 100, 207, 800]

搜索target = 2 小于 nums[mid](mid==3) = 20:

第2次搜索:

搜索区间:[0, 2] [-1, 2, 5]

搜索target = 2 等于 nums[mid](mid==1) = 2:

故找到！

二分查找算法，举例2

待查找值 target = 200

下标: [0, 1, 2, 3, 4, 5, 6, 7]

nums: [-1, 2, 5, 20, 90, 100, 207, 800]

第1次搜索:

搜索区间[0, 7] [-1, 2, 5, 20, 90, 100, 207, 800]

搜索target = 200 大于 nums[mid](mid==3) = 20:

第2次搜索:

搜索区间: [4, 7] [90, 100, 207, 800]

搜索target = 200 大于 nums[mid](mid==5) = 100

第3次搜索:

搜索区间: [6, 7] [207, 800]

搜索target = 200 小于 nums[mid](mid==6) = 207

搜索区间: [6, 5]

该区间是一个非法区间，该表不存在，故target 不在数组中！

二分查找(循环), 课堂练习

//搜索到返回1

//否则返回 0

//排序数组

//排序数组长度

```
int binary_search(int sort_array[], int n, int target){
```

```
    int begin = 0;
```

```
    int end = n - 1;
```

//搜索目标

```
    while ( 1 ) {
```

```
        int mid = (begin + end) / 2;
```

```
        if (target == sort_array[mid]) {
```

```
            2
```

```
        }
```

```
        else if (target < sort_array[mid]) {
```

```
            3
```

```
        }
```

```
        else if (target > sort_array[mid]) {
```

```
            4
```

```
        }
```

```
    }
```

```
    5
```

```
}
```

3分钟填写代码,
有问题随时提出!

二分查找(循环), 实现

//搜索到返回1

//否则返回 0

//排序数组

//排序数组长度

```
int binary_search(int sort_array[], int n, int target){  
    int begin = 0;  
    int end = n - 1; //搜索目标  
    while (begin <= end) {  
        int mid = (begin + end) / 2;  
        if (target == sort_array[mid]){  
            return 1;  
        }  
        else if (target < sort_array[mid]){  
            end = mid - 1;  
        }  
        else if (target > sort_array[mid]){  
            begin = mid + 1;  
        }  
    }  
    return 0;  
}
```

例如, 待搜索数字target == 2, 200

数组 A = [-1, 2, 5, 20, 90, 100, 207, 800]

二分查找(递归), 课堂练习

//搜索到返回1

//否则返回 0

//待搜索的区间左端、右端

//搜索目标

```
int binary_search(int sort array[],  
                  int begin, int end, int target){
```

```
if ( 1 ) {  
    return 0;  
}
```

3分钟填写代码,
有问题随时提出!

```
int mid = 2
```

```
if (target == sort array[mid]){ //当找到时
```

```
3
```

```
}  
else if ( 4 ) { //??时候找左区间  
    return binary_search(sort array, begin, mid-1, target);  
}
```

```
else if ( 5 ) { //??时候找右区间  
    return binary_search(sort array, mid + 1, end, target);  
}
```

```
}
```

二分查找(递归), 实现

//搜索到返回1

//否则返回 0 //待搜索的区间左端、右端

//搜索目标
`int binary_search(int sort_array[],
 int begin, int end, int target){`

`if (begin > end) {
 return 0;`

`}`
`int mid = (begin + end) / 2;`

`if (target == sort_array[mid]){ //当找到时
 return 1;`

`}`
`else if (target < sort_array[mid]) { //??时候找左区间
 return binary_search(sort_array, begin, mid-1, target);`

`}`
`else if (target > sort_array[mid]) { //??时候找右区间
 return binary_search(sort_array, mid + 1, end, target);`

`}`

例如, 待搜索数字target == 2, 200

数组 A = [-1, 2, 5, 20, 90, 100, 207, 800]

二分查找:测试

```
int search_array(int sort_array[], int s_n,
                 int random_array[], int r_n,
                 int result[]){
    int cnt = 0;
    int i;
    for (i = 0; i < r_n; i++){
        result[cnt++] =
            binary_search(sort_array, s_n, random_array[i]);
    }
    return cnt;
}

int main(){
    int A[] = {-1, 2, 5, 20, 90, 100, 207, 800};
    int B[] = {50, 90, 3, -1, 207, 80};
    int C[10] = {0};
    int C_n = search_array(A, 8, B, 6, C);
    int i;
    for (i = 0; i < C_n; i++){
        printf("%d\n", C[i]);
    }
    return 0;
}
```



0
1
0
1
1
0

例1:区间查找

给定一个**排序数组nums**(nums中有**重复**元素)与**目标值target**，如果target在nums里**出现**，则返回target所在区间的**左右端点下标**，[左端点, 右端点]，如果target在nums里**未出现**，则返回[-1, -1]。

```
int* searchRange(int* nums, int numsSize, int target, int* returnSize) {
```

```
} 下标 = [0, 1, 2, 3, 4, 5, 6, 7]    target = 8, 返回: [3, 6]
```

```
nums = [5, 7, 7, 8, 8, 8, 8, 10]    target = 6, 返回: [-1, -1]
```

选自 **LeetCode 34. Search for a Range**

<https://leetcode.com/problems/search-for-a-range/description/>

难度:**Medium**

例1:思考

- 1.可否**直接**通过二分查找，很容易**同时求出**目标target所在区间的**左右端点**？
- 2.若**无法同时求出**区间左右端点，将对目标target的二分查找增加**怎样的限制条件**，就可**分别求出**目标target所在区间的**左端点与右端点**？

下标 = [0, 1, 2, 3, 4, 5, 6, 7]

target = 8, 返回: [3, 6]

nums = [5, 7, 7, 8, 8, 8, 8, 10]

target = 6, 返回: [-1, -1]

例1:算法思路(区间左端点)

查找区间**左端点**时，增加如下**限制条件**：

当 $\text{target} == \text{nums}[\text{mid}]$ 时，若此时 $\text{mid} == 0$ 或 $\text{nums}[\text{mid}-1] < \text{target}$ ，则说明 mid 即为区间左端点，返回；否则设置区间右端点为 $\text{mid}-1$ 。

target = 3

... **1** 3 3 3 3 5 ...



mid 区间左端点

[3 3 3 3 5 ...



mid = 0 区间左端点

... 1 3 **3** 3 3 5 ...



mid



[... 1 3 3]



区间右端点设置为mid - 1

例1:算法思路(区间右端点)

查找区间**右端点**时, 增加如下**限制条件**:

当 $\text{target} == \text{nums}[\text{mid}]$ 时, 若此时 $\text{mid} == \text{numsSize} - 1$ 或 $\text{nums}[\text{mid} + 1] > \text{target}$, 则说明 mid 即为区间右端点; 否则设置区间左端点为 $\text{mid} + 1$

target = 3

... 1 3 3 3 3 **5** ...



区间右端点 mid

... 1 3 3 3 3 **]**



mid = nums.size() - 1

区间右端点

... 1 3 3 3 3 5 ...



mid



[3 5 ...



区间左端点设置为mid + 1

例1:算法思路

查找区间的右端点:

target = 8;

位置: [0, 1, 2, 3, 4, 5, 6, 7]

元素: [5, 7, 7, 8, 8, 8, 8, 10]



区间右端点

第1次搜索:

[0, 7] [5, 7, 7, 8, 8, 8, 8, 10] mid = 3
nums[mid] == 8, nums[mid+1] == 8,

第2次搜索:

[4, 7] [8, 8, 8, 10] mid = 5
nums[mid] == 8, nums[mid+1] == 8,

第3次搜索:

[6, 7] [8, 10] mid = 6
nums[mid] = 8,
nums[mid+1] == 10 > target

故区间右端点为 6!

例1:课堂练习(区间左端点)

```
int left_bound(int nums[], int n, int target){
    int begin = 0;
    int end = n - 1;
    while (1){
        int mid = (begin + end) / 2;
        if (target == nums[mid]){
            if (2){
                return mid;
            }
        }
        else if (target < nums[mid]){
            4
        }
        else if (target > nums[mid]){
            5
        }
    }
    return -1;
}
```

3分钟填写代码，
有问题随时提出！

例1:实现(区间左端点)

```
int left_bound(int nums[], int n, int target){
    int begin = 0;
    int end = n - 1;
    while (begin <= end) {
        int mid = (begin + end) / 2;
        if (target == nums[mid]) {
            if (mid == 0 || nums[mid - 1] < target) {
                return mid;
            }
            end = mid - 1;
        }
        else if (target < nums[mid]) {
            end = mid - 1;
        }
        else if (target > nums[mid]) {
            begin = mid + 1;
        }
    }
    return -1;
}
```

target = 3

... 1 3 3 3 3 5 ...



mid 区间左端点

[3 3 3 3 5 ...



mid = 0 区间左端点

... 1 3 3 3 3 5 ...



mid



[... 1 3 3]



区间右端点设置为mid - 1

例1:课堂练习(区间右端点)

```
int right_bound(int nums[], int n, int target){
    int begin = 0;
    int end = n - 1;
    while(begin <= end){
        int mid = (begin + end) / 2;
        if (1){
            if (2){
                return mid;
            }
            3
        }
        else if (4){
            end = mid - 1;
        }
        else if (5){
            begin = mid + 1;
        }
    }
    return -1;
}
```

3分钟填写代码，
有问题随时提出！

例1:实现(区间右端点)

```
int right_bound(int nums[], int n, int target){
    int begin = 0;
    int end = n - 1;
    while(begin <= end){
        int mid = (begin + end) / 2;
        if (target == nums[mid]){
            if (mid == n - 1 || nums[mid + 1] > target){
                return mid;
            }
            begin = mid + 1;
        }
        else if (target < nums[mid]){
            end = mid - 1;
        }
        else if (target > nums[mid]){
            begin = mid + 1;
        }
    }
    return -1;
}
```

target = 3

... 1 3 3 3 3 5 ...



区间右端点 mid

... 1 3 3 3 3]



mid = nums.size() - 1

区间右端点

... 1 3 3 3 3 5 ...



mid



[3 5 ...



区间左端点设置为mid+1

例1:测试与leetcode提交结果

```
int* searchRange(int* nums, int numsSize, int target, int* returnSize) {
    int *result = (int *)malloc(sizeof(int) * 2);
    result[0] = left_bound(nums, numsSize, target);
    result[1] = right_bound(nums, numsSize, target);
    *returnSize = 2;
    return result;
}

int main() {
    int nums[] = {5, 7, 7, 8, 8, 8, 8, 10};
    int i;
    int *result;
    for (i = 0; i < 12; i++) {
        int result_size = 0;
        result = searchRange(nums, 8, i, &result_size);
        printf("%d : [%d, %d]\n", i, result[0], result[1]);
    }
    free(result);
    return 0;
}
```

```
0 : [-1, -1]
1 : [-1, -1]
2 : [-1, -1]
3 : [-1, -1]
4 : [-1, -1]
5 : [0, 0]
6 : [-1, -1]
7 : [1, 2]
8 : [3, 6]
9 : [-1, -1]
10 : [7, 7]
11 : [-1, -1]
```

Search for a Range

Submission Detail

87 / 87 test cases passed.

Status: Accepted

Runtime: 3 ms

Submitted: 0 minutes ago

例2:岛屿数量

用一个二维数组代表一张**地图**，这张地图由字符“0”与字符“1”组成，其中“0”字符代表**水域**，“1”字符代表**小岛土地**，小岛“1”被水“0”所**包围**，当小岛土地“1”在**水平和垂直**方向相连接时，认为是同一块土地。求这张地图中小岛的**数量**。

```
int numIslands(char **grid,  
               int gridSize, int gridColSize) {  
}
```

1个小岛:

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

3个小岛:

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

选自 **LeetCode 200. Number of Islands**

<https://leetcode.com/problems/number-of-islands/description/>

难度:**Medium**

例2:思考

```
int numIslands(char **grid,  
               int gridSize, int gridColSize) {  
}
```

1个小岛:

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

3个小岛:


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

思考:

- 1.统计岛屿的数量首先要能够探索**相连接**的小岛，如何对一个**完整**的小岛进行探索？
- 2.在探索过程中，使用什么样的数据结构对**已到达的位置**进行记录？
- 3.若已知一个**起始点**，如何设计**递归搜索**对某个位置进行遍历，将与该位置**相连**的位置都进行标记？
- 4.**整体的算法**是怎样的？

例2:分析，搜索独立小岛

给定该二维地图grid，与一个**二维标记数组mark**(初始化为0)，mark数组的每个位置都与grid**对应**，设计一个**搜索**算法，从该地图中的某个岛的**某个位置**出发，**探索**该岛的全部土地，将探索到的位置在mark数组中**标记**为1。

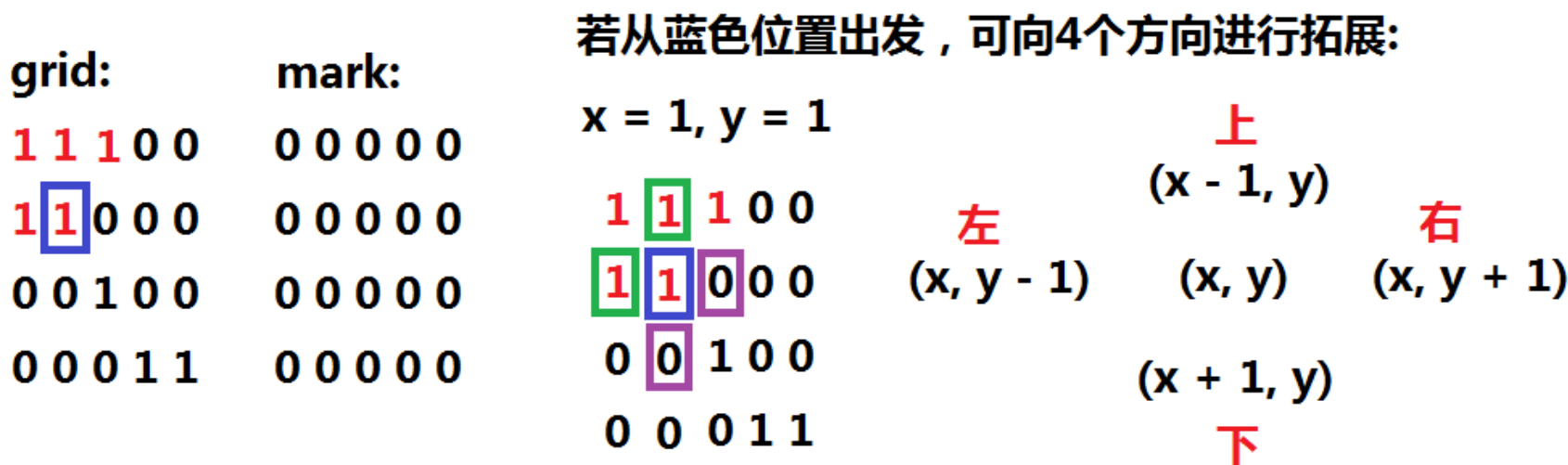
grid:	mark:		grid:	mark:
1 1 1 0 0	0 0 0 0 0		1 1 1 0 0	1 1 1 0 0
1 1 0 0 0	0 0 0 0 0		1 1 0 0 0	1 1 0 0 0
0 0 1 0 0	0 0 0 0 0		0 0 1 0 0	0 0 0 0 0
0 0 0 1 1	0 0 0 0 0		0 0 0 1 1	0 0 0 0 0

```
#define MAXN 500
void DFS(int mark[][MAXN], char **grid,
         int gridRowSize, int gridColSize,
         int x, int y) {

}
```

例2:算法设计

1. 标记当前搜索位置**已被搜索**(标记当前位置的mark数组为1)。
2. 按照方向数组的4个方向，**拓展**4个新位置newx、newy。
3. 若新位置**不在地图范围内**，则**忽略**。
4. 如果新位置**未曾到达**过(mark[newx][newy]为0)、且**是陆地**(grid[newx][newy]为“1”)，**继续递归搜索**该位置。



绿色位置代表可由蓝色位置拓展至，紫色代表无法进入(水域)。

例2:算法设计

若按照上、下、左、右的顺序深度搜索 从(1, 1)位置出发

图1: grid: mark:

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

图3: grid: mark:

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

图5: grid: mark:

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

图2: grid: mark:

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

图4: grid: mark:

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

图6: grid: mark:

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

回到了图3，继续递归

例2:课堂练习

```
#define MAXN 500
void DFS(int mark[][MAXN], char **grid,
         int gridRowSize, int gridColSize,
         int x, int y) {
    1
    static const int dx[] = {-1, 1, 0, 0};
    static const int dy[] = {0, 0, -1, 1};
    int i;
    for (i = 0; i < 4; i++) {
        int newx = 2
        int newy = 3
        if (newx < 0 || newx >= gridRowSize ||
            newy < 0 || newy >= gridColSize) {
            4
        }
        if (5) {
            DFS(mark, grid, gridRowSize, gridColSize, newx, newy);
        }
    }
}
```

3分钟填写代码，
有问题随时提出！

例2:实现

```
#define MAXN 500
void DFS(int mark[][MAXN], char **grid,
         int gridSize, int gridColSize,
         int x, int y) {
    mark[x][y] = 1;

    static const int dx[] = {-1, 1, 0, 0};
    static const int dy[] = {0, 0, -1, 1};
    int i;
    for (i = 0; i < 4; i++) {
        int newx = dx[i] + x;

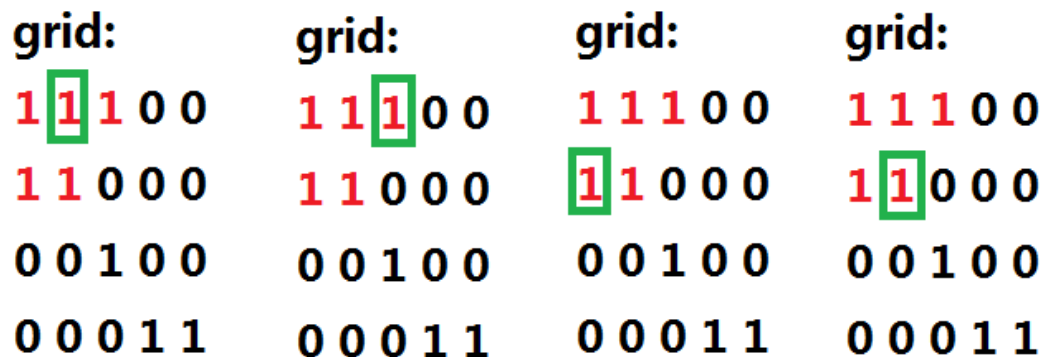
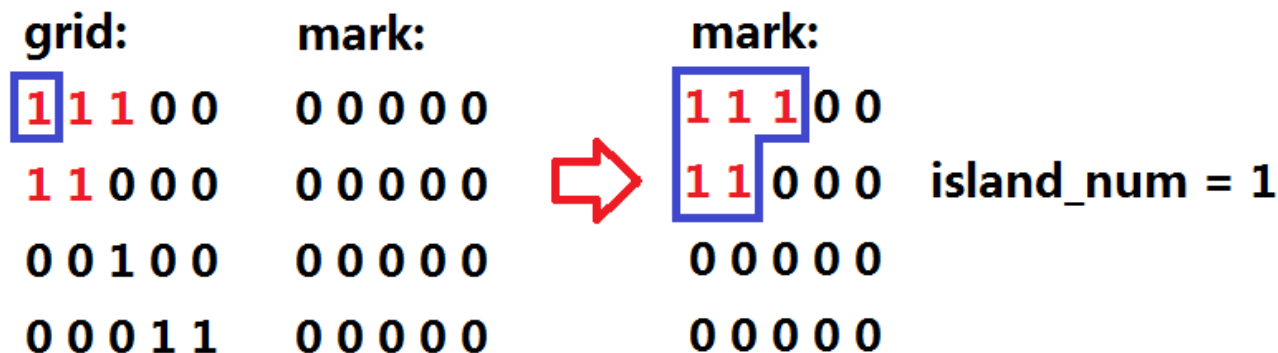
        int newy = dy[i] + y;

        if (newx < 0 || newx >= gridSize ||
            newy < 0 || newy >= gridColSize) {
            continue;
        }
        if (mark[newx][newy] == 0 && grid[newx][newy] == '1') {
            DFS(mark, grid, gridSize, gridColSize, newx, newy);
        }
    }
}
```

例2:算法设计

求地图中**岛屿的数量**:

- 1.设置岛屿数量**island_num** = 0;
- 2.设置mark数组, 并**初始化**。
- 3.遍历**地图grid**的上所有的点, 如果当前点是**陆地**, 且**未被访问**过, 调用**搜索**接口DFS(mark, grid, i, j) ; **完成**搜索后island_num++。



绿色的点都不会被搜索！

例2:算法设计

grid:

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

mark:

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



mark:

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

island_num = 2

grid:

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

mark:

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



mark:

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

island_num = 3

例2:整体代码实现

```
int numIslands(char **grid, int gridSize, int gridColSize) {  
    int island_num = 0;  
    int mark[MAXN][MAXN] = {0};  
    int i, j;  
    for (i = 0; i < gridSize; i++) {  
        for (j = 0; j < gridColSize; j++) {  
            if (mark[i][j] == 0 && grid[i][j] == '1') {  
                DFS(mark, grid, gridSize, gridColSize, i, j);  
                island_num++;  
            }  
        }  
    }  
    return island_num;  
}
```

//遍历grid的中的每个位置, 当该位置没有到达过且为陆地时, 进行搜索

例2:测试与leetcode提交结果

```
int main(){
    char **grid;
    char str[10][10] = {"11100", "11000", "00100", "00011"};
    int i, j;
    grid = (char **)malloc(sizeof(char *) * 10);
    for (i = 0; i < 4; i++){
        grid[i] = (char *)malloc(sizeof(char) * 10);
        for (j = 0; j < 5; j++){
            grid[i][j] = str[i][j];
        }
    }
    printf("%d\n", numIslands(grid, 4, 5));
    for (i = 0; i < 10; i++){
        free(grid[i]);
    }
    free(grid);
    return 0;
}
```

Number of Islands

Submission Details

47 / 47 test cases passed.

Status: **Accepted**

Runtime: 16 ms

Submitted: 0 minutes ago

3

请按任意键继续 . . .



小象学院
ChinaHadoop.cn

课间休息10分钟！

有问题提出！

例3: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]`

```
int** subsets(int* nums, int numsSize,  
              int** columnSizes, int *returnSize)  
{
```

选自 **LeetCode 78. Subsets**

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

难度:**Medium**

例3: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； n 个元素即有 **2^n 种**可能。

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3(方法1回溯法):预备知识(循环)

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

```
int main() {
    int nums[] = {1, 2, 3};
    int item[10] = {0};     //item , 生成各个子集的数组
    int itemSize = 0;

    int result[10][10] = {0};     //result存储最终结果
    int returnSize = 0;     //returnSize代表最终结果由多少个子集
    int columnSizes[10] = {0};     //columnSizes存储各个子集中元素个数

    int i, j;

    for (i = 0; i < 3; i++){
        item[itemSize] = nums[i];
        itemSize++;
        columnSizes[returnSize] = itemSize;
        for (j = 0; j < itemSize; j++){
            result[returnSize][j] = item[j];
        }
        returnSize++;
    }

    for (i = 0; i < returnSize; i++){
        for (j = 0; j < columnSizes[i]; j++){
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

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

请按任意键继续. . .

例3(方法1回溯法):预备知识(递归)

//nums[] = [1, 2, 3], 将子集[1],[1,2],[1,2,3]递归的加入result.

```
void generate(int i, int nums[], int numsSize,
              int item[], int itemSize,
              int result[][10], int columnSizes[], int *returnSize) {
    if (1) { //递归返回的条件
        return;
    }
    2 //为item添加一个元素
    itemSize++;
    columnSizes[*returnSize] = itemSize; //孩子集元素的个数
    int j;
    for (j = 0; j < itemSize; j++) { //生成result数组
        result[*returnSize][j] = 3
    }
    (*returnSize)++;
    generate(i + 1, nums, numsSize, item,
            itemSize, result, columnSizes, returnSize);
}
```

```
int main() {
    int nums[] = {1, 2, 3};
    int item[10] = {0};
    int itemSize = 0;
    int result[10][10] = {0};
    int returnSize = 0;
    int columnSizes[10] = {0};
    int i, j;

    generate(0, nums, 3, item, 0,
            result, columnSizes, &returnSize);

    for (i = 0; i < returnSize; i++) {
        for (j = 0; j < columnSizes[i]; j++) {
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

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

例3(方法1回溯法):预备知识(递归)

//nums[] = [1, 2, 3], 将子集[1],[1,2],[1,2,3]递归的加入result.

```
void generate(int i, int nums[], int numsSize,
             int item[], int itemSize,
             int result[][10], int columnSizes[], int *returnSize){
```

```
if ( i >= numsSize ) { //递归返回的条件
```

```
    return;
```

```
}
```

```
item[itemSize] = nums[i];
```

//为item添加一个元素

```
itemSize++;
```

```
columnSizes[*returnSize] = itemSize; //该子集元素的个数
```

```
int j;
```

```
for (j = 0; j < itemSize; j++){ //生成result数组
```

```
    result[*returnSize][j] = item[j];
```

```
}
```

```
(*returnSize)++;
```

```
generate(i + 1, nums, numsSize, item,
        itemSize, result, columnSizes, returnSize);
```

```
}
```

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:

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3(方法1回溯法):算法设计

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[] = []

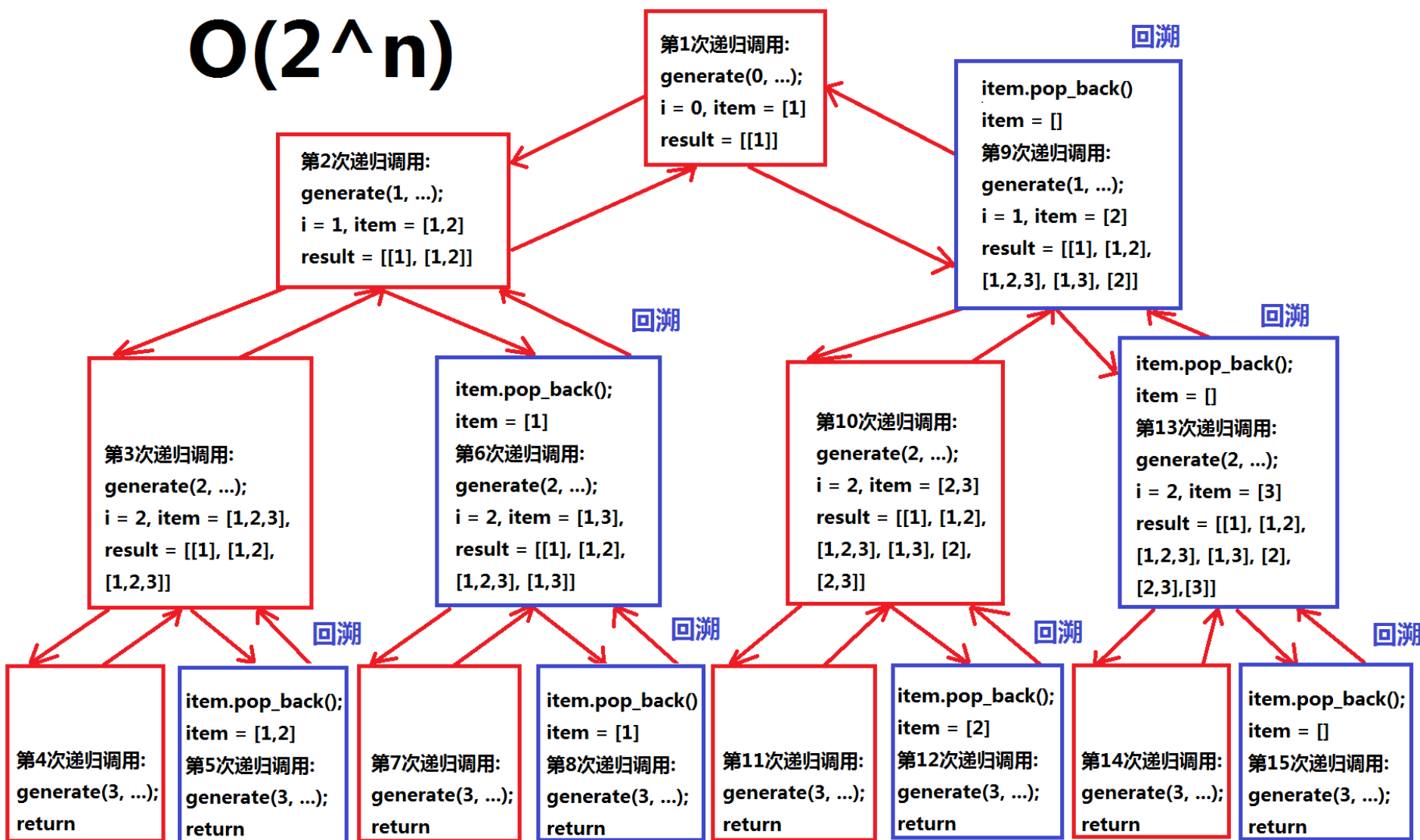
对于**元素1**，

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3(方法1回溯法):算法设计

$O(2^n)$



例3(方法1回溯法):实现

```
void generate(int i, int nums[], int numsSize,  
             int item[], int itemSize,  
             int **result, int columnSizes[], int *returnSize) {
```

```
    if (i >= numsSize) {  
        return;  
    }  
    item[itemSize] = nums[i];  
    itemSize++;  
    columnSizes[*returnSize] = itemSize;  
  
    int j;  
    for (j = 0; j < itemSize; j++) {  
        result[*returnSize][j] = item[j];  
    }  
    (*returnSize)++;  
    generate(i + 1, nums, numsSize, item,  
            itemSize, result, columnSizes, returnSize);
```

//完全一样的代码

```
    itemSize--;  
    generate(i + 1, nums, numsSize, item,  
            itemSize, result, columnSizes, returnSize);  
}
```

//增加了拿出的动作

第2次递归调用:
generate(1, ...);
i = 1, item = [1,2]
result = [[1], [1,2]]

第3次递归调用:
generate(2, ...);
i = 2, item = [1,2,3],
result = [[1], [1,2],
[1,2,3]]

item.pop_back();
item = [1]
第6次递归调用:
generate(2, ...);
i = 2, item = [1,3],
result = [[1], [1,2],
[1,2,3], [1,3]]

例3(方法1回溯法):整体代码

```
int** subsets(int* nums, int numsSize, int** columnSizes, int *returnSize)
{
    int item[20] = {0};
    int max_size = 1 << numsSize;
    int **result = (int **)malloc(sizeof(int *) * max_size);
    *columnSizes = (int *)malloc(sizeof(int *) * max_size);
    int i;
    for (i = 0; i < max_size; i++){
        (*columnSizes)[i] = 0;
        result[i] = (int *)malloc(sizeof(int) * 20);
    }
    *returnSize = 1;
    generate(0, nums, numsSize, item, 0, result, *columnSizes, returnSize);
    return result;
}
```

例3(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3(方法2位运算法):算法设计

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。整数的每个bit位对应了集合中的元素A,B,C; 当某bit位为1时代表集合存在这一位对应的元素, 为0时代表不存在。

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3(方法2位运算法):算法设计

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3(方法2位运算法):课堂练习

```
int** subsets(int* nums, int numsSize, int** columnSizes, int *returnSize){
    int max_size = 1 << numsSize; //最大的集合
    int **result = (int **)malloc(sizeof(int *) * max_size);
    *columnSizes = (int *)malloc(sizeof(int *) * max_size);
    int i, j;
    for (i = 0; i < max_size; i++){
        (*columnSizes)[i] = 0;
        result[i] = (int *)malloc(sizeof(int) * 20);
    }
    *returnSize = 0;
    for (i = 0; i < max_size; i++){ //遍历所有集合
        int item[20] = {0};
        int itemSize = 0;
        for (j = 0; j < numsSize; j++){
            if (1){
                item[itemSize++] = 2;
            }
        }
        (*columnSizes)[*returnSize] = 3;
        for (j = 0; j < itemSize; j++){
            result[*returnSize][j] = item[j];
        }
        (*returnSize)++;
    }
    return result;
}
```

3分钟时间填写代码，
有问题随时提出！

例3(方法2位运算法):实现

```
int** subsets(int* nums, int numsSize, int** columnSizes, int *returnSize){  
    int max_size = 1 << numsSize; //最大的集合  
    int **result = (int **)malloc(sizeof(int *) * max_size);  
    *columnSizes = (int *)malloc(sizeof(int *) * max_size);  
    int i, j;  
    for (i = 0; i < max_size; i++){  
        (*columnSizes)[i] = 0;  
        result[i] = (int *)malloc(sizeof(int) * 20);  
    }  
    *returnSize = 0;  
    for (i = 0; i < max_size; i++){ //遍历所有集合 //整数 i 代表从 0至 $2^n-1$  这  $2^n$ 个集合  
        int item[20] = {0}; // (1 < j)即为构造nums数组的第j个元素  
        int itemSize = 0; //若  $i \& (1 < j)$ 为真则 nums[j]放入item  
        for (j = 0; j < numsSize; j++){  
            if (i & (1 < j)) {  
                item[itemSize++] = nums[j];  
            }  
        }  
        (*columnSizes)[*returnSize] = itemSize;  
        for (j = 0; j < itemSize; j++){  
            result[*returnSize][j] = item[j];  
        }  
        (*returnSize)++;  
    }  
    return result;  
}
```

例3: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?

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

```
int main() {
    int nums[] = {1, 2, 3};
    int *columnSizes = 0;
    int returnSize = 0;
    int **result = subsets(nums, 3, &columnSizes, &returnSize);
    int i, j;
    for (i = 0; i < returnSize; i++) {
        if (columnSizes[i] == 0) {
            printf("[]");
        }
        for (j = 0; j < columnSizes[i]; j++) {
            printf("[%d]", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Subsets

Submission Details

10 / 10 test cases passed.

Status: Accepted

Runtime: 6 ms

Submitted: 0 minutes ago

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

结束

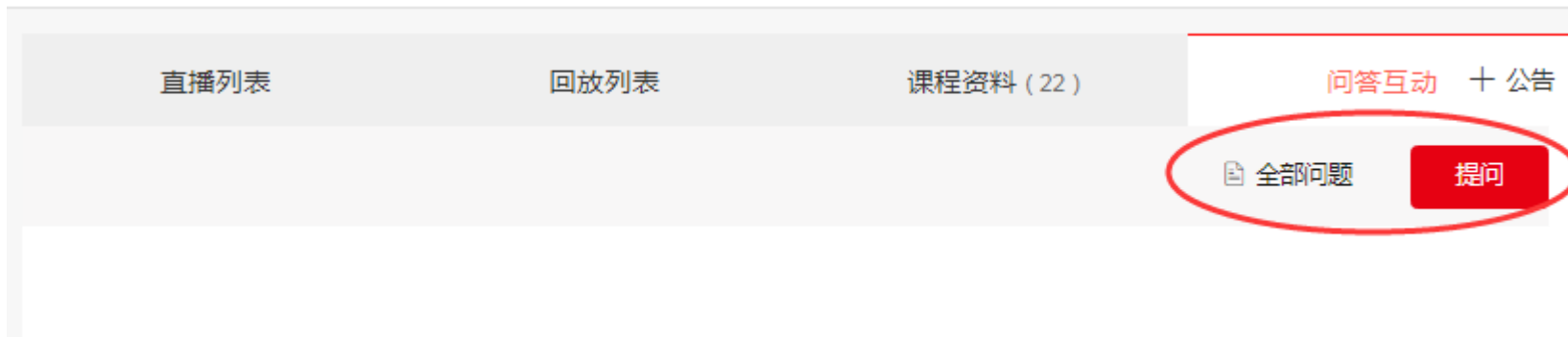
非常感谢大家！

林沐

问答互动

在所报课的课程页面，

- 1、点击“全部问题”显示本课程所有学员提问的问题。
- 2、点击“提问”即可向该课程的老师 and 助教提问问题。



联系我们

小象学院：互联网新技术在线教育领航者

— 微信公众号：**小象学院**

