LeetCode 热题 HOT 100

1. 两数之和

思路

(暴力枚举) $O(n^2)$

两重循环枚举下标`i,j`，然后判断`nums[i]+nums[j]`是否等于`target`。

(哈希表) $O(n)$

使用C++中的哈希表`unordered_map<int, int> hash`

- 用哈希表存储前面遍历过的数，当枚举到当前数时，若哈希表中存在`target - nums[i]`的元素，则表示已经找到符合条件的两个数。
- 若不存在`target - nums[i]`的元素则枚举完当前数再把当前数放进哈希表中

时间复杂度：由于只扫描一遍，且哈希表的插入和查询操作的复杂度是$O(1)$，所以总时间复杂度是$O(n)$.

c++代码

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> hash;
        for(int i = 0; i < nums.size(); i++){
            if(hash.count(target - nums[i])){
                return {i, hash[target - nums[i]]};
            }
            hash[nums[i]] = i;
        }
        return {};
    }
};
```

2. 两数相加

思路 $O(n)$。

(模拟)

这是道模拟题，模拟我们小时候列竖式做加法的过程：

1. 从最低位至最高位，逐位相加，如果和大于等于`10`，则保留个位数字，同时向前一位进`1`。
2. 如果最高位有进位，则需在最前面补`1`。

具体实现

1. 同时从头开始枚举两个链表，将`l1`和`l2`指针指向的元素相加存到`t`中，再将`t % 10`的元素存到`dummy`链表中，再`t / 10`去掉存进去的元素，`l1`和`l2`同时往后移动一格。
2. 当遍历完所有元素时，如果`t != 0`，再把`t`存入到`dummy`链表中。

做有关链表的题目，有个常用技巧：添加一个虚拟头结点：`ListNode *head = new ListNode(-1);`，可以简化边界情况的判断。

时间复杂度：由于总共扫描一遍，所以时间复杂度是 $O(n)$。

c++代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(-1);
        ListNode* cur = dummy;
        int t = 0;
        while(l1 || l2){
            if(l1) t += l1->val, l1 = l1->next;
            if(l2) t += l2->val, l2 = l2->next;
            cur = cur->next = new ListNode(t % 10);
            t /= 10;
        }
        if(t) cur->next = new ListNode(t);
        return dummy->next;
    }
};
```

3. 无重复字符的最长子串

思路

(双指针扫描) $O(n)$

定义两个指针 $i, j(i <= j)$，表示当前扫描到的子串是 $[i, j]$ (闭区间)。扫描过程中维护一个哈希表 `unordered_map <chat,int>hash`，表示 $[i, j]$中每个字符出现的次数。

线性扫描时，每次循环的流程如下：

- 1.指针 `j` 向后移一位, 同时将哈希表中 `s[j]` 的计数加一，即 `hash[s[j]]++`；
- 2.假设 `j` 移动前的区间 `[i,j]` 中没有重复字符，则 `j` 移动后，只有 `s[j]` 可能出现 `2` 次。因此我们不断向后移动 `i`，直至区间 `[i,j]` 中 `s[j]` 的个数等于 `1` 为止；
- 3.当确保 `[i, j]` 中不存在重复元素时，更新 `res`；

时间复杂度分析：由于 `i`，`j` 均最多增加 `n` 次，且哈希表的插入和更新操作的复杂度都是 $O(1)$，因此，总时间复杂度 $O(n)$。

c++代码

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> hash;
        int res = 0;
        for(int j = 0, i = 0; j < s.size(); j++){
```

```
            hash[s[j]]++;
            while(hash[s[j]]> 1){
                hash[s[i]]--;
                i++;
            }
            res = max(res, j - i + 1);
        }
        return res;
    }
};
```

4. 寻找两个正序数组的中位数

思路

(递归) $O(log(n+m))$

找出两个正序数组的**中位数**等价于找出两个正序数组中的**第k小数**。如果两个数组的大小分别为`n`和`m`，那么第`k = (n + m)/2`小数就是我们要求的中位数。

如何寻找第k小的元素?

过程如下:

1、考虑一般情况，我们在`nums1`和`nums2`数组中各取前`k/2`个元素

我们默认`nums1`数组比`nums2`数组的有效长度小 。`nums1`数组的有效长度从`i`开始，`nums2`数组的有效长度从`j`开始，其中`[i,si - 1]`是`nums1`数组的前`k / 2`个元素，`[j, sj - 1]`是`nums2`数组的前`k / 2`个元素。

2、接下来我们去比较`nums1[si - 1]`和`nums2[sj - 1]`的大小。

- 如果`nums1[si - 1] > nums2[sj - 1]`，则说明`nums1`中取的元素过多，`nums2`中取的元素过少。因此`nums2`中的前`k/2`个元素一定都小于等于第`k`小数，即`nums2[j,sj-1]`中元素。我们可以舍去这部分元素，在剩下的区间内去找第`k - k / 2`小的元素，也就是说第`k`小一定在`[i,n]`与`[sj,m]`中。
- 如果`nums1[si - 1] <= nums2[sj - 1]`，同理可说明`nums2`中的前`k/2`个元素一定都小于等于第`k`小数，即`nums1[i,si-1]`中元素。我们可以舍去这部分元素，在剩下的区间内去找第`k - k / 2`小的元素，也就是说第`k`小一定在`[si,n]`与`[j,m]`中。

3、递归过程`2`，每次可将问题的规模减少一半，最后剩下的一个数就是我们要找的第`k`小数。

递归边界:

- 当`nums1`数组为空时，我们直接返回`nums2`数组的第`k`小数。
- 当`k == 1`时，且两个数组均不为空，我们返回两个数组首元素的最小值，即`min(nums1[i], nums2[j])`。

奇偶分析:

- 当两个数组元素个数的总和`total`为偶数时，找到第`total / 2`小`left`和第`total / 2 + 1`小`right`，结果是`(left + right / 2.0)`。
- 当`total`为奇数时，找到第`total / 2 + 1`小，即为结果。

时间复杂度分析: $k=(m+n)/2$，且每次递归 k 的规模都减少一半，因此时间复杂度是 $O(log(m+n))$.

c++代码

```cpp
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int tot = nums1.size() + nums2.size();
        if(tot % 2 == 0){
            int left = find(nums1, 0, nums2, 0, tot / 2);
            int right =find(nums1, 0, nums2, 0, tot / 2 + 1);
            return (left + right) / 2.0;
        }else{
            return find(nums1, 0, nums2, 0, tot / 2 + 1);
        }
    }

    int find(vector<int>& nums1,int i, vector<int>& nums2, int j, int k){
        if(nums1.size() - i > nums2.size() - j) return find(nums2, j, nums1, i, k);
        if(k == 1){
            //当第一个数组已经用完
            if(i == nums1.size()) return nums2[j];
            else return min(nums1[i], nums2[j]);
        }
        //当nums1数组为空时，我们直接返回nums2数组的第k小数。
        if (nums1.size() == i) return nums2[j + k - 1];
        int si = min((int)nums1.size(), i + k / 2), sj = j + k - k / 2;
        if(nums1[si - 1] > nums2[sj - 1]){
            return find(nums1, i, nums2, sj, k - (sj - j));
        }else{
            return find(nums1, si, nums2, j, k - (si - i));
        }
    }
};
```

10. 正则表达式匹配

思路

(动态规划) $O(nm)$

状态表示： `f[i][j]` 表示字符串 `s` 的前 `i` 个字符和字符串 `p` 的前 `j` 个字符能否匹配。

状态计算：

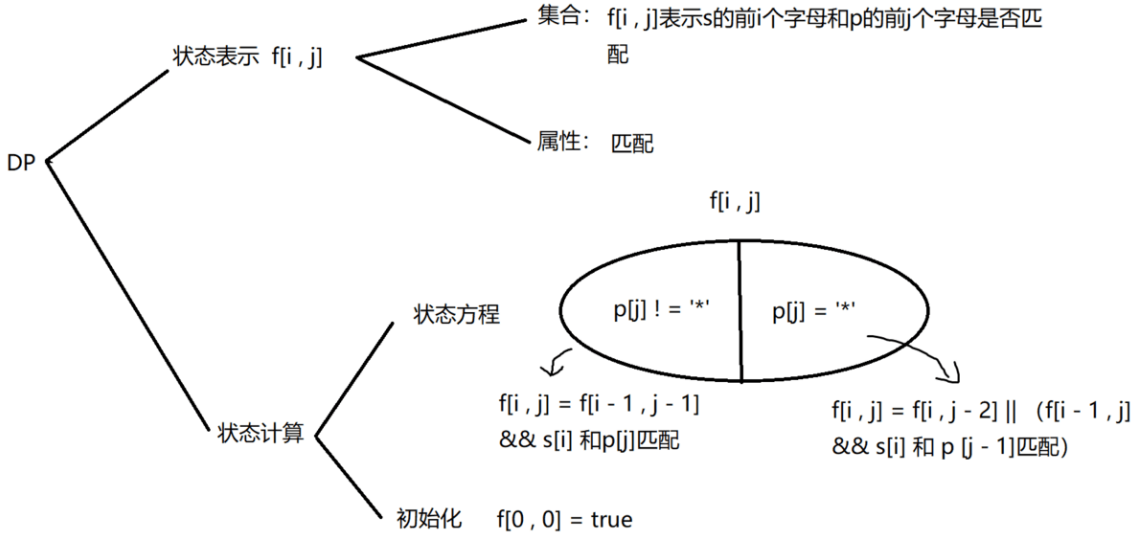
根据 `p[j]` 是什么来划分集合：

- 1、`p[j] != '*'`，即 `p[j]` 是字符，看 `p[j]` 和 `s[i]` 的关系。如果 `p[j] == s[i]`，则需判断 `s` 的前 `i - 1` 个字母 能否和 `p` 的前 `j -1` 个字母匹配，即 `f[i][j] == f[i - 1][j - 1]`，不匹配，无法转移。
- 2 `P[j]` 是匹配符:
  - 如果 `p[j] == '.'`，则 `p[j]` 和 `s[j]` 匹配，则需判断 `s` 的前 `i - 1` 个字母能否和 `p` 的前 `j -1` 个字母匹配，即 `f[i][j] == f[i - 1][j - 1]`。
  - `p[j] == '*'`，得看 `p[j - 1]` 和 `s[i]` 的关系。如果不匹配，即 `p[j - 1] != s[i]`，那么 `'*'` 匹配 `0` 个 `p[j - 1]`,则需判断 `s` 的前 `i` 个字母 能否和 `p` 的前 `j - 2` 个字母匹配，即 `f[i][j] == f[i][j - 2]`。如果匹配，即 `p[j - 1] == s[i] || p[j - 1] == '.'`，则需判断 `s` 的前 `i - 1` 个字母能否和 `p` 的前 `j` 个字母匹配，即 `f[i][j] == f[i - 1][j])`。

动态规划

从集合角度来考虑DP问题

DP问题：用某一个状态或某一个数来代表一类数，解决某一个集合的最大值，最小值，总个数



例如 s = "aab" p = "c*a*b"

（由于是s的前i个字母 和 p的前j个字母，所以两个字符窜前面都加上一个空串 “ ” ）

f[i , j]表示s的前i个字母和p的前j个字母是否匹配

1、p[j] != '*' , f[i , j] = f[i - 1 , j - 1] && s[i]和p[j]匹配

2、p[j] == '*'，需要枚举*表示多少个字母

f[i , j] = f[i , j - 2] || (f[i - 1 , j - 2] && s[i]和p[j - 1]匹配) || (f[i - 2 , j - 2] && s[i - 1 : i]和p[j - 1]匹配)...

由于 f[i - 1 , j] = f[i - 1 , j - 2] || (f[i - 2 , j - 2] && s[i - 1]和p[j - 1]匹配) || (f[i - 3 , j - 2] && s[i - 2 : i - 1]和p[j - 1]匹配)...

其中s[m:n]表示从s[m]到s[n]

观察可知 蓝色部分和红色部分只相差 s[i]和p[i - 1]匹配

则将蓝色部分分解出s[i]和p[i - 1]匹配 可得

f[i , j] = f[i , j - 2] || (f[i - 1 , j] && s[i]和p[j - 1]匹配)

总结:

```
f[i][j] == f[i - 1][j - 1], 前提条件为p[j] == s[i] || p[j] == '.'
f[i][j] == f[i][j - 2], 前提条件为p[j] == '*' &&  p[j - 1] != s[i]
f[i][j] == f[i - 1][j], 前提条件为p[j] == '*' && ( p[j - 1] == s[i] || p[j - 1] == '.')
```

c++代码

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int n = s.size(), m = p.size();
        s = ' ' +  s, p = ' ' + p;
        vector<vector<bool>> f(n + 1, vector<bool>(m + 1));
        f[0][0] = true;
        for(int i = 0; i <= n; i++)
            for(int j = 1; j <= m; j++){
                if(j + 1 <= m && p[j + 1] == '*') continue;
                if(i &&p[j] != '*'){
                    f[i][j] = f[i - 1][j - 1] && (s[i] == p[j] || p[j] == '.');
                }else if(p[j] == '*'){
                    f[i][j] = f[i][j - 2] || i && f[i - 1][j] && (s[i] == p[j - 1] || p[j - 1] == '.');
```

```
                }
            }
        return f[n][m];
    }
};
```

5. 最长回文子串

思路

(双指针) $O(n^2)$

- 1、枚举数组中的每个位置`i`，从当前位置开始向两边扩散
- 2、当回文子串的长度是奇数时，从`i - 1`,`i + 1`开始往两边扩散
- 3、当回文子串的长度是偶数时，从`i`，`i + 1`开始往两边扩散
- 4、找到以`i`为中心的最长回文子串的长度，若存在回文子串比以前的长，则更新答案。

图示:

时间复杂度分析：枚举数组中的每个位置`i`需要$O(n)$的时间复杂度，求回文子串需要$O(n)$的时间复杂度，因此总的时间复杂度为$O(n^2)$。

c++代码

```
class Solution {
public:
    string longestPalindrome(string s) {
        string res;
        for(int i = 0; i < s.size(); i++){
            int l = i, r = i + 1;  //回文串长度为偶数
            while(l >= 0 && r < s.size() && s[l] == s[r]) l--, r++;
            if(res.size() < r - l - 1) res = s.substr(l + 1, r - l - 1);
            l = i - 1, r = i + 1;  //回文串长度为奇数
            while(l >= 0 && r < s.size() && s[l] == s[r]) l--, r++;
            if(res.size() < r - l - 1) res = s.substr(l + 1, r - l - 1);
        }
        return res;
    }
};
```

11. 盛最多水的容器

思路

(双指针扫描) $O(n)$

过程如下：

1、定义两个指针`i`和`j`，分别表示容器的左右边界，初始化`i = 0`，`j = h.size() - 1`，容器大小为`min(i, j)*(j - i)`。

2、若`h[i] < h[j]`，则`i++`，否则`j--`，每次迭代更新最大值。

证明：

容器大小由短板决定, 移动长板的话, 水面高度不可能再上升, 而宽度变小了, 所以只有通过移动短板, 才有可能使水位上升。

时间复杂度分析： 两个指针总共扫描 n 次，因此总时间复杂度是 $O(n)$。

c++代码

```cpp
class Solution {
public:
    int maxArea(vector<int>& h) {
        int res = 0;
        for(int i = 0, j = h.size() - 1; i < j;){
            res = max(res, min(h[i], h[j])*(j - i));
            if(h[i] < h[j]) i++;
            else j--;
        }
        return res;
    }
};
```

15. 三数之和

思路

(排序 + 双指针) $O(n^2)$

- 1、将整个`nums`数组按从小到大排好序
- 2、枚举每个数，表示该数`nums[i]`已被确定，在排序后的情况下，通过双指针`l`，`r`分别从左边`l = i + 1`和右边`r = n - 1`往中间靠拢，找到`nums[i] + nums[l] + nums[r] == 0`的所有符合条件的搭配
- 3、在找符合条件搭配的过程中，假设`sum = nums[i] + nums[l] + nums[r]`
  若`sum > 0`，则`r`往左走，使`sum`变小
  若`sum < 0`，则`l`往右走，使`sum`变大
  若`sum == 0`，则表示找到了与`nums[i]`搭配的组合`nums[l]`和`nums[r]`，存到`ans`中
- 4、判重处理
  确定好`nums[i]`时，`l`需要从`i + 1`开始
  当`nums[i] == nums[i - 1]`，表示当前确定好的数与上一个一样，需要直接跳过
  当找符合条件搭配时，即`sum == 0`,需要对相同的`nums[l]`和`nums[r]`进行判重处理

时间复杂度分析： $O(n^2)$。

c++代码

```cpp
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());
        for(int i = 0; i < n; i++){
            if(i && nums[i] == nums[i - 1]) continue;  //跳过相同的i，保证每次都是新开始
            int l = i + 1, r = n - 1;
            while(l < r){
                int sum = nums[i] + nums[l] + nums[r];
                if(sum > 0) r--;
                else if(sum < 0) l++;
                else if(sum == 0){
                    res.push_back({nums[i], nums[l], nums[r]});
                    do l++; while(l < r && nums[l] == nums[l - 1]);  //跳过相同的l
```

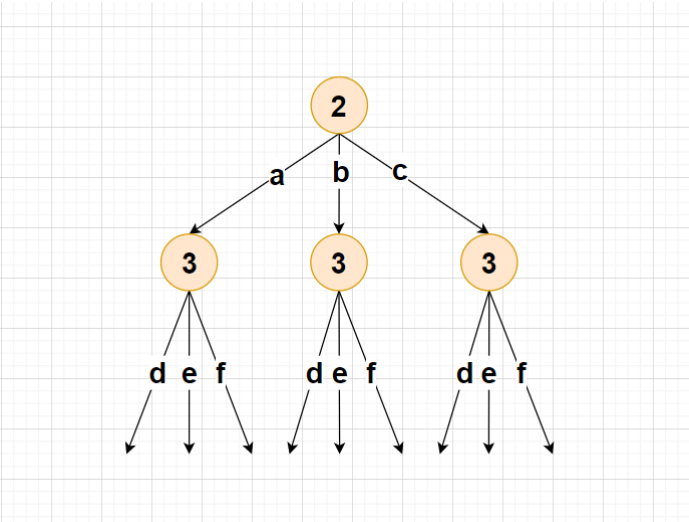
```
                    do r--; while(l < r && nums[r] == nums[r + 1]);  //跳过相同的r
                }
            }
        }
        return res;
    }
};
```

17. 电话号码的字母组合

思路

(回溯，哈希，组合排列) $O(4^n)$

对于字符串`23`来说，递归搜索树如下图所示：



递归函数设计：

```
void dfs(string& digits, int u, string path) {
```

`digits`字符串数组，`u`表示枚举到`digitis`的第`u`个位置，`path`用来记录路径。

解题过程如下：

1、将数字到字母的映射到哈希表中。

2、递归搜索每个数字对应位置可以填哪些字符，这里我们从哈希表中查找，并将其拼接到`path`后。

3、当`u == digits.size()`时，表示搜索完一条路径，将其加入答案数组中。

时间复杂度分析： 一个数字最多有`4`种情况，假设有`n`个数字，因此`4^n`种情况是一个上限，因此时间复杂度是$O(4^n)$。

c++代码

```
class Solution {
public:
    vector<string> res;
    string strs[10] = {
        "", "", "abc", "def",
        "ghi", "jkl", "mno",
        "pqrs", "tuv", "wxyz"
    };
```

```
    vector<string> letterCombinations(string digits) {
        if(digits.empty()) return res;
        dfs(digits, 0, "");
        return res;
    }

    void dfs(string digits, int u, string path){
        if(u == digits.size()){
            res.push_back(path);
            return ;
        }
        for(char c : strs[digits[u] - '0']){
            dfs(digits, u + 1, path + c);
        }
    }
};
```

19. 删除链表的倒数第 N 个结点

思路

(双指针) $O(n)$

具体过程如下:

1、创建虚拟头节点`dummy`，并让`dummy->next = head`。

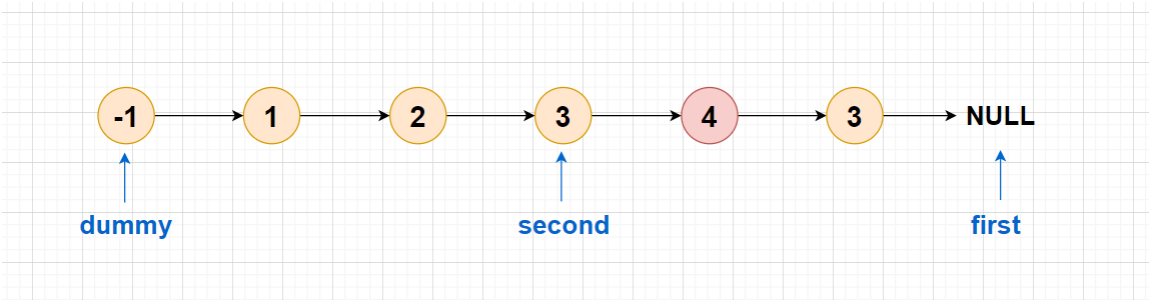
2、创建快指针`first`和慢指针`second`，并让其都指向`dummy`。

3、先让快指针`first`走`n + 1`步，而后`first`，`second`指针同时向后走，直到`first`指针指向空节点，此时`second`指向节点的下一个节点就是需要删除的节点，将其删除。

4、最后返回虚拟头节点的下一个节点。

解释:

始终保持两个指针之间间隔`n`个节点，在`first`到达终点时，`second`的下一个结点就是倒数第`n`个节点。



时间复杂度分析: 只遍历一次链表，因此时间复杂度为$O(n)$。

c++代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
```

```
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode(-1);
        dummy->next = head;
        ListNode* first = dummy;
        ListNode* second = dummy;
        for(int i = 0; i <= n; i++) first = first->next;
        while(first){
            first = first->next;
            second = second->next;
        }
        second->next = second->next->next;
        return dummy->next;
    }
};
```

20. 有效的括号

思路

(栈) $O(n)$

定义一个栈，从前往后枚举每个字符：

- 1、当遇到`'('`,`'{'`,`'['`左括号时，将元素压进栈中
- 2、当遇到`')'`,`']'`,`'}'`右括号时，
  - 如果栈不为空并且栈顶元素是对应的左括号，说明这是匹配的符号，将栈顶元素`pop`出即可。
  - 否则，表示不匹配，`return false`。
- 3、最后，若栈是空栈，表示所有字符都已经匹配好了，若不是空栈，表示还存在未能匹配好的子符

时间复杂度分析： 每个字符最多进栈出栈一次，因此时间复杂度为$O(n)$。

c++代码

```
class Solution {
public:
    bool isValid(string s) {
        stack<int> stk;
        for(int i = 0; i < s.size(); i++){
            if(s[i] == '(' || s[i] == '{' || s[i] == '[') stk.push(s[i]);
            else if(s[i] == ')'){
                if(!stk.empty() && stk.top() == '(') stk.pop();
                else return false;
            }
            else if(s[i] == '}'){
                if(!stk.empty() && stk.top() == '{') stk.pop();
                else return false;
            }
            else if(s[i] == ']'){
                if(!stk.empty() && stk.top() == '[') stk.pop();
                else return false;
```

```
            }
        }
        return stk.empty();
    }
};
```

21. 合并两个有序链表

思路

(线性合并) $O(n)$

解题过程如下：

1. 新建虚拟头节点 `dummy`，定义 `cur` 指针并使其指向 `dummy`。
2. 当 `l1` 或 `l2` 都不为为空时：
    - 若 `l1->val < l2->val`，则令 `cur` 的 `next` 指针指向 `l1` 且 `l1` 后移；
    - 若 `l1->val >=l2->val`，则令 `cur` 的 `next` 指针指向 `l2` 且 `l2` 后移；
    - `cur` 后移一步；
3. 将剩余的 `l1` 或 `l2` 接到 `cur` 指针后边。
4. 最后返回 `dummy->next`。

时间复杂度分析：

c++代码

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(-1);
        ListNode* cur = dummy;
        while(l1 && l2){
            if(l1->val < l2->val){
                cur->next = l1;
                l1 = l1->next;
            }else{
                cur->next = l2;
                l2 = l2->next;
            }
            cur = cur->next;
        }
        if(l1) cur->next = l1;
        if(l2) cur->next = l2;
        return dummy->next;
    }
};
```

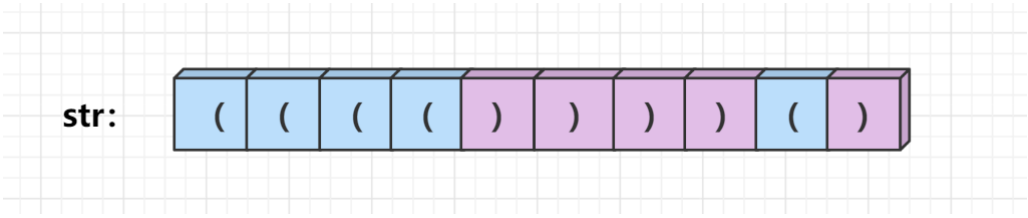
22. 括号生成

思路

(dfs) $O(C_{2n}^{n})$

首先我们需要知道一个结论，一个合法的括号序列需要满足两个条件：

- 1、左右括号数量相等
- 2、任意前缀中左括号数量 `>=` 右括号数量 （也就是说每一个右括号总能找到相匹配的左括号）

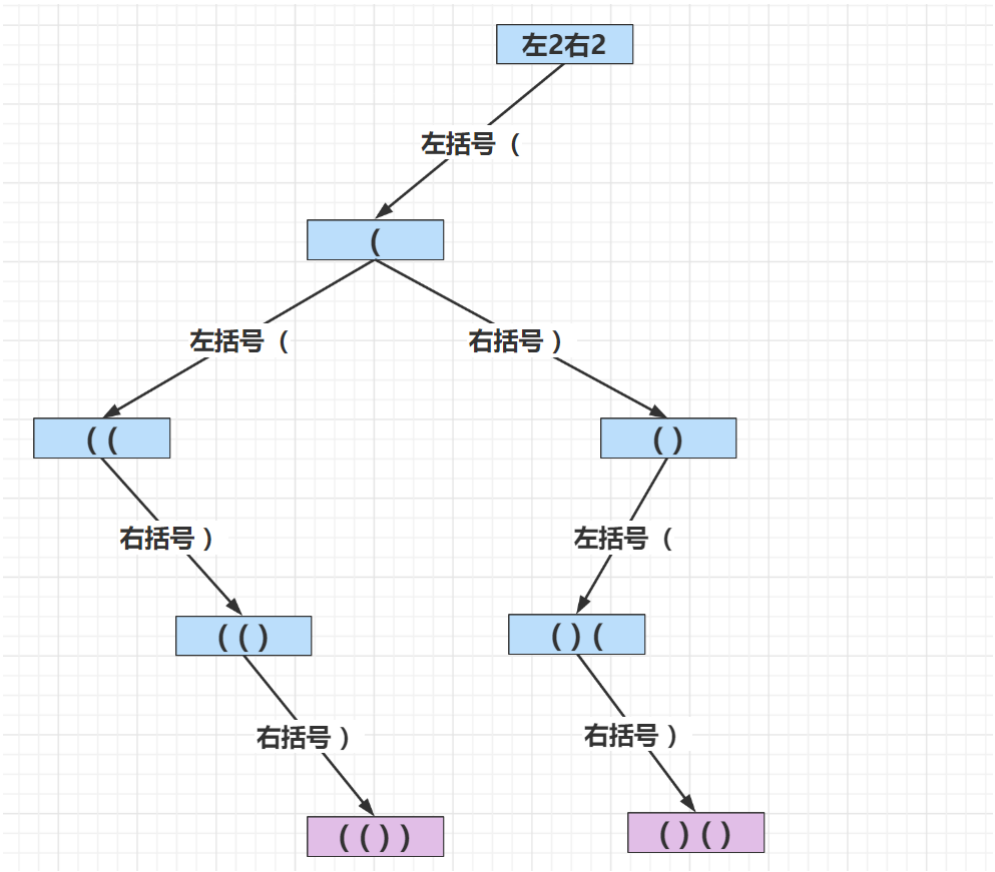


题目要求我们生成`n`对的合法括号序列组合，可以考虑使用深度优先搜索，将搜索顺序定义为枚举序列的每一位填什么，那么最终的答案一定是有`n`个左括号和`n`个右括号组成。

如何设计`dfs`搜索函数?

最关键的问题在于搜索序列的当前位时，是选择填写左括号，还是选择填写右括号？因为我们已经知道合法的括号序列任意前缀中左括号数量一定`>=`右括号数量，因此，如果左括号数量不大于`n`，我们可以放一个左括号，等待一个右括号来匹配。如果右括号数量小于左括号的数量，我们可以放一个右括号，来使一个右括号和一个左括号相匹配。

递归树如下：



递归函数设计

```
void dfs(int n ,int lc, int rc ,string str)
```

`n`是括号对数，`lc`是左括号数量，`rc`是右括号数量，`str`是当前维护的合法括号序列。

搜索过程如下：

- 1、初始时定义序列的左括号数量`lc`和右括号数量`rc`都为`0`。
- 2、如果`lc < n`，左括号的个数小于`n`，则在当前序列`str`后拼接左括号。
- 3、如果`rc < n && lc > rc`，右括号的个数小于左括号的个数，则在当前序列`str`后拼接右括号。
- 4、当`lc == n && rc == n`时，将当前合法序列`str`加入答案数组`res`中。

时间复杂度分析：经典的卡特兰数问题，因此时间复杂度为 $O(\frac{1}{n+1}C_{2n}^{n}) = O(C_{2n}^{n})$。

c++代码

```
class Solution {
public:
    vector<string> res;
    vector<string> generateParenthesis(int n) {
        dfs(n, 0, 0, "");
        return res;
    }

    void dfs(int n, int lc, int rc, string str){
        if(lc == n && rc == n){
            res.push_back(str);
            return ;
        }
        if(lc < n) dfs(n, lc + 1, rc, str + '(');
        if(rc < n && lc > rc) dfs(n, lc, rc + 1, str + ')');
    }
};
```

23. 合并K个升序链表

思路

(优先队列) $O(nlogk)$

我们可以通过双路归并合并两个有序链表，但是这题要求对多个链表进行并操作。 其实和双路归并思路类似，我们分别用指针指向该链表的头节点，每次找到这些指针中值最小的节点，然后依次连接起来，并不断向后移动指针。

如何找到一堆数中的最小值?

用小根堆维护指向`k`个链表当前元素最小的指针，因此这里我们需要用到优先队列，并且自定义排序规则，如下：

```
struct cmp{ //自定义排序规则
    bool operator() (ListNode* a, ListNode* b){
        return a->val > b->val;  // val值小的在队列前
    }
};
```

具体过程如下：

- 1、定义一个优先队列，并让`val`值小的元素排在队列前。
- 2、新建虚拟头节点`dummy`，定义`cur`指针并使其指向`dummy`。

- 3、首先将`k`个链表的头节点都加入优先队列中。
- 4、当队列不为空时：
  - 取出队头元素`t`（队头即为`k`个指针中元素值最小的指针）；
  - 令`cur`的`next`指针指向`t`，并让`cur`后移一位；
  - 如果`t`的`next`指针不为空，我们将`t->next`加入优先队列中；
- 5、最后返回`dummy->next`。

时间复杂度分析： $O(nlogk)$，n表示的是所有链表的总长度，k表示k个排序链表。

c++代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    struct cmp{ //自定义排序规则
        bool operator() (ListNode* a, ListNode* b){
            return a->val > b->val;  // val值小的在队列前
        }
    };
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if(lists.size() == 0) return nullptr;
        priority_queue<ListNode*, vector<ListNode*>, cmp> heap;
        auto dummy = new ListNode(-1), cur = dummy;
        for(ListNode* l : lists)  if(l) heap.push(l);
        while(heap.size()){
            ListNode* t = heap.top();  // k个指针中元素值最小的指针t取出来
            heap.pop();
            cur = cur->next = t;
            if(t->next)  heap.push(t->next);  //将t->next加入优先队列中
        }
        return dummy->next;
    }
};

```

31. 下一个排列

思路

(找规律) $O(n)$

对于数组排列问题，我们可以知道，如果一个数组是**升序数组**，那么它一定是最小的排列。如果是**降序数组**，那么它一定是最大的排列。

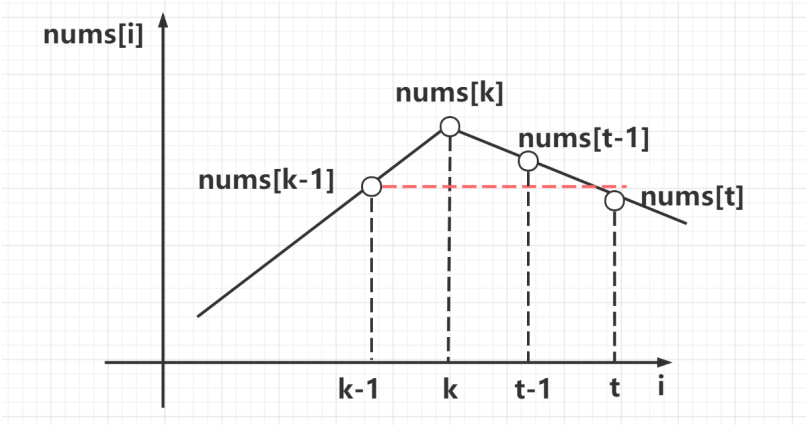
而找下一个排列就是从后往前寻找第一个出现降序的地方，把这个地方的数字与后边第一个比它大的的数字交换，再把该位置之后整理为升序。

换句话说，就是为了从后往前找，找到第一个“ 可以变大的数 “，而**从前往后的降序序列**已经最大了，因此第一个可以变大的数一定出现在**从前往后的升序序列**中，即**从后往前的第一个降序地方。**

具体过程如下：

- 1、从数组末尾往前找，找到 **第一个** 位置 `k`，使得 `nums[k-1] < nums[k]`，则从后往前看 `nums[k-1, nums[k]` 满足降序，`nums[k-1]` 就是**第一个可以变大的数。**
- 2、如果 `k <=0`，说明不存在这样的 `k` ，则数组是不递增的，直接将数组逆转即可。
- 3、如果存在这样的 `k`，则让 `t = k`，从前往后找到**第一个**位置 `t`，使得 `nums[t] <= nums[k-1]`，则 `nums[t-1]` 就是第一个大于 `nums[k-1]` 的数。
- 4、交换 `nums[k-1]` 与 `nums[t-1]`，然后将数组从 `k` 到末尾部分逆序。

图示过程



时间复杂度分析： 遍历整个数组需要的时间为$O(n)$

c++代码

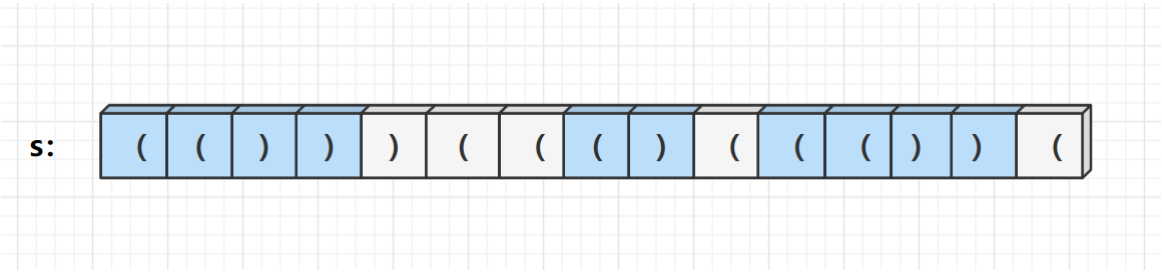
```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int k = nums.size() - 1;
        while(k > 0 && nums[k - 1] >= nums[k]) k--;
        if(k <= 0){  //从前往后降序
            reverse(nums.begin(), nums.end());
        }else{
            int t = k;
            while(t < nums.size() && nums[t] > nums[k - 1]) t++;
            swap(nums[t - 1], nums[k - 1]);
            reverse(nums.begin() + k, nums.end());
        }
    }
};
```

32. 最长有效括号

思路

(栈) $O(n)$

我们可以发现一个规律，每一段合法括号序列它在字符串 `s` 中出现的位置一定是连续且不相交的，如下图所示：



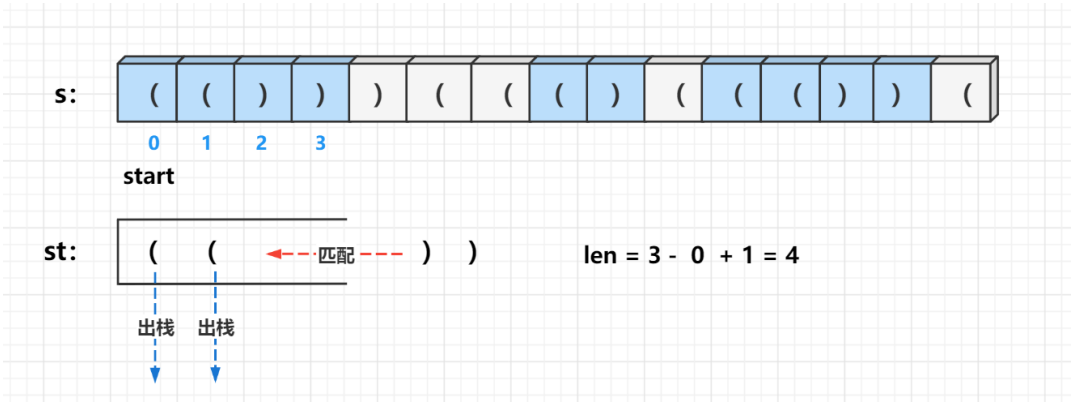
因此我们能想到的最直接的做法是找到每个可能的子串后判断它是否为合法括号序列，但这样的时间复杂度会达到 $O(n^3)$ 。

有没有一种更高效的做法?

我们知道栈在处理括号匹配有着天然的优势，于是考虑**用栈去判断序列的合法性。**遍历整个字符串`s`，把所有的合法括号序列按照右括号来分类，对于每一个右括号，都去求一下以这个右括号为右端点的最长的合法括号序列的左端点在什么位置。我们把每个右括号都枚举一遍之后，再取一个`max`，就是整个的最大长度。

具体过程如下:

- 1、用栈维护当前待匹配的左括号的位置，同时用`start`记录一个新的可能合法的子串的起始位置，初始设为`0`。
- 2、如果`s[i] =='('`，那么把`i`进栈。
- 3、如果`s[i] == ')'`，那么弹出栈顶元素 （代表栈顶的左括号匹配到了右括号），出栈后:
  - 如果栈为空，说明以当前右括号为右端点的合法括号序列的左端点为`start`，则更新答案`i - start + 1`。
  - 如果栈不为空，说明以当前右括号为右端点的合法括号序列的左端点为栈顶元素的下一个元素，则更新答案`i - (st.top() + 1) + 1`。



- 4、遇到右括号`)`且当前栈为空，则当前的`start`开始的子串不再可能为合法子串了，下一个合法子串的起始位置可能是`i + 1`，更新`start = i + 1`。
- 5、最后返回答案即可。

实现细节: 栈保存的是下标。

时间复杂度分析: 每个位置遍历一次，最多进栈一次，故时间复杂度为 $O(n)$。

c++代码

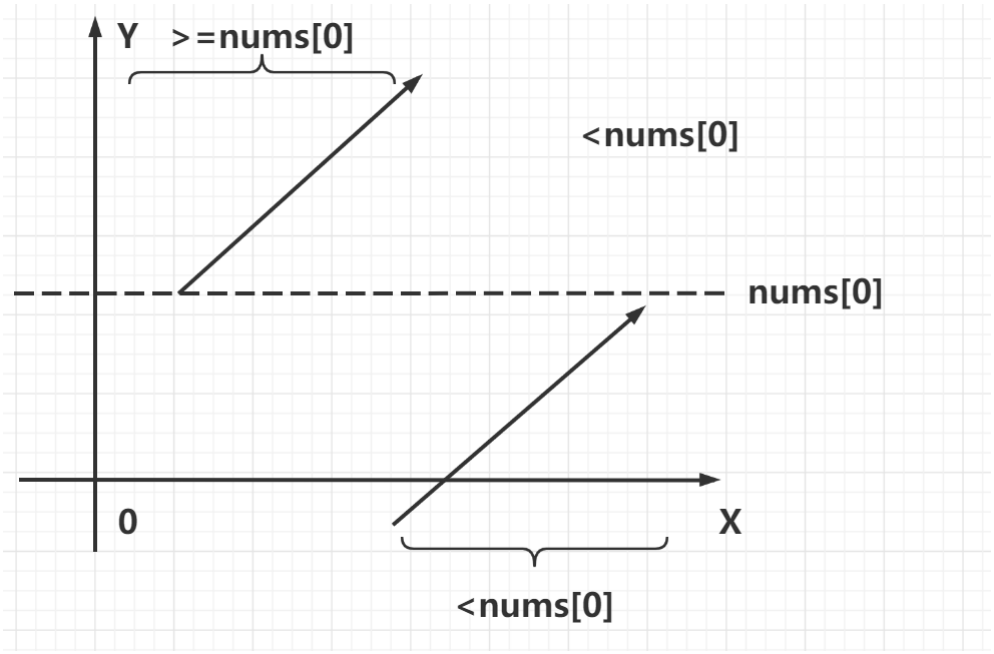
```
class Solution {
public:
    int longestValidParentheses(string s) {
        stack<int> stk;
        int res = 0, start = 0;
        for(int i = 0; i < s.size(); i++){
```

```
            if(s[i] == '(')  stk.push(i);
            else{
                if(!stk.empty()){
                    stk.pop();
                    if(stk.empty()) res = max(res, i - start + 1);
                    else res = max(res, i - stk.top());
                }else{
                    start = i + 1;
                }
            }
        }
        return res;
    }
};
```

33. 搜索旋转排序数组

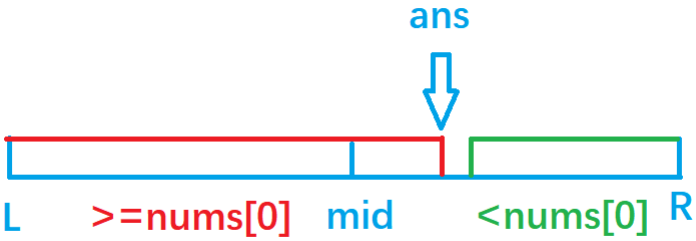
思路

(二分) $O(logn)$



1、先找到旋转点，在旋转点左边的点都大于等于`nums[0]`，右边的点都小于`nums[0]`，因此可以用二分找到该旋转点，即`>= nums[0]`的最右边界。

- 当`nums[mid] >= nums[0]`时，往右边区域找，`l = mid`。
- 当`nums[mid] < nums[0]`时，往左边区域找，`r = mid - 1`。



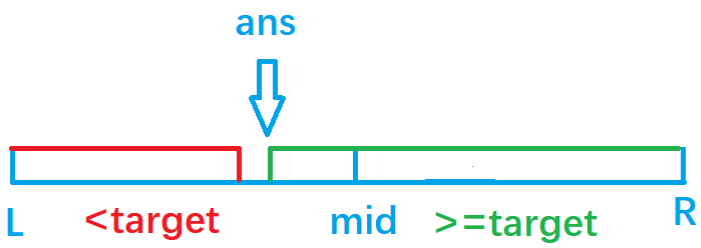
2、找到旋转点`l`后，可以知道`[0,l]`,`[l + 1, n - 1]`是两个有序数组，判断出`target`的值在哪个有序数组中，确定好二分的区间`[l,r]`。

- 当`target >= nums[0]`，说明`target`在`[0, l]`区间内，我们令`l = 0`，`r`保持不变。

- 否则，说明`target`在`[l + 1, n - 1]`区间内，我们令`l = r + 1`，`r = n - 1`。

3、在`[l,r]`区间中，由于该区域也具有单调性，通过二分找到该值的位置，即二分`>= target`的最左边界

- 当`nums[mid] >= target`时，往左边区域找，`r = mid`。
- 当`nums[mid] < target`时， 往右边区域找，`l = mid + 1`。



4、若最后找到的元素`nums[r] != target`，则表示不存在该数，返回`-1`，否则返回该数值。

时间复杂度分析： 二分的时间复杂度为 $O(logn)$

c++代码

```cpp
class Solution {
public:
    int search(vector<int>& nums, int target) {
        if(nums.empty()) return -1;
        int l = 0, r = nums.size() - 1;
        while(l < r){
            int mid = (l + r + 1) / 2;
            if(nums[mid] >= nums[0]) l = mid;
            else r = mid - 1;
        } // l == r
        if(target >= nums[0]) l = 0;
        else l = r + 1, r = nums.size() - 1;
        while(l < r){
            int mid = (l + r) / 2;
            if(nums[mid] >= target) r = mid;
            else l = mid + 1;
        }
        if(target == nums[r]) return r;
        return -1;
    }
};
```

34. 在排序数组中查找元素的第一个和最后一个位置

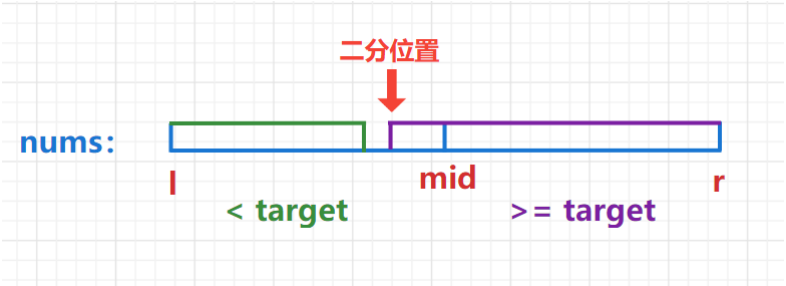
思路

(二分) $O(logn)$

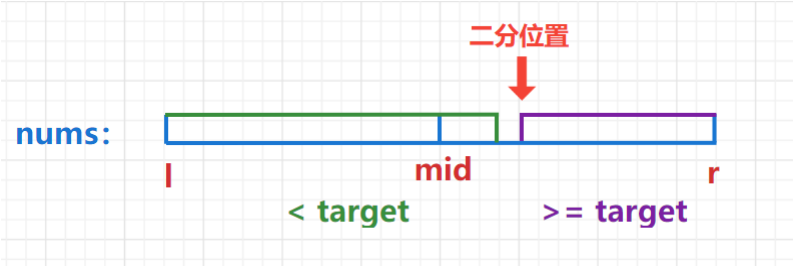
两次二分，第一次二分查找第一个`>=target`的位置，第二次二分查找最后一个`<=target`的位置。查找成功则返回两个位置下标，否则返回`[-1,-1]`。

第一次

- 1、二分的范围，`l = 0`， `r = nums.size() - 1`，我们去二分查找`>=target`的最左边界。
- 2、当`nums[mid] >= target`时，往左半区域找，`r = mid`。



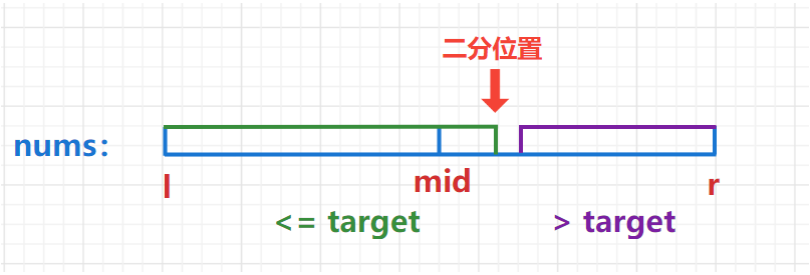
- 3、当`nums[mid] < target`时， 往右半区域找，`l = mid + 1`。



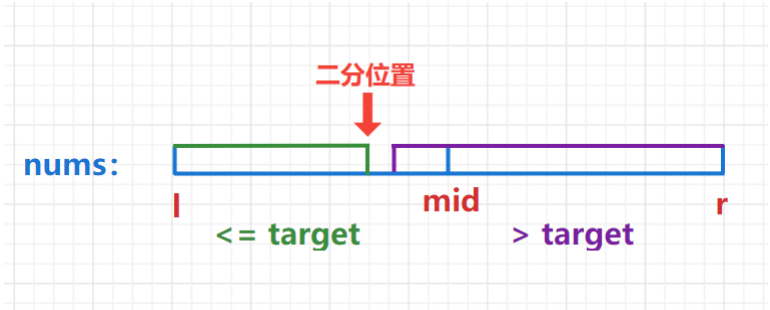
- 4、如果`nums[r] != target`，说明数组中不存在目标值`target`，返回`[-1, -1]`。否则我们就找到了第一个`>=target`的位置`L`。

第二次

- 1、二分的范围，`l = 0`， `r = nums.size() - 1`，我们去二分查找`<=target`的最右边界。
- 2、当`nums[mid] <= target`时，往右半区域找，`l = mid`。



- 3、当`nums[mid] > target`时， 往左半区域找，`r = mid - 1`。



- 4、找到了最后一个`<=target`的位置`R`，返回区间`[L,R]`即可。

时间复杂度分析： 两次二分查找的时间复杂度为 $O(logn)$。

c++代码

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        if(!nums.size()) return {-1, -1};
        int l = 0, r = nums.size() - 1;
        while(l < r){ //查找 >= target的最左边界
            int mid = (l + r) / 2;
            if(nums[mid] >= target) r = mid;
```

```
            else l = mid + 1;
        }
        if(nums[r] != target) return {-1, -1};
        int L = r;
        l = 0, r = nums.size() - 1;
        while(l < r){
            int mid = (l + r + 1) / 2;
            if(nums[mid] <= target) l = mid;
            else r = mid - 1;
        }
        return {L, r};
    }
};
```

39. 组合总和

思路

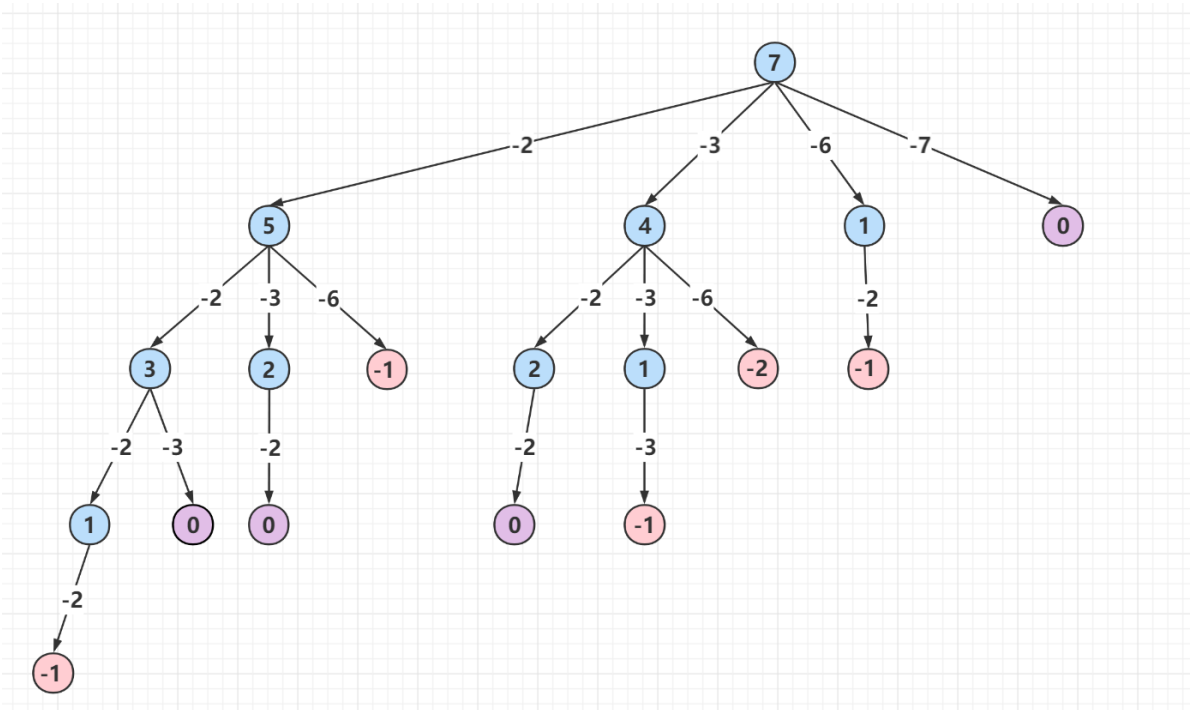
(dfs，递归)

递归枚举，枚举每个数字可以选多少次。

递归过程如下:

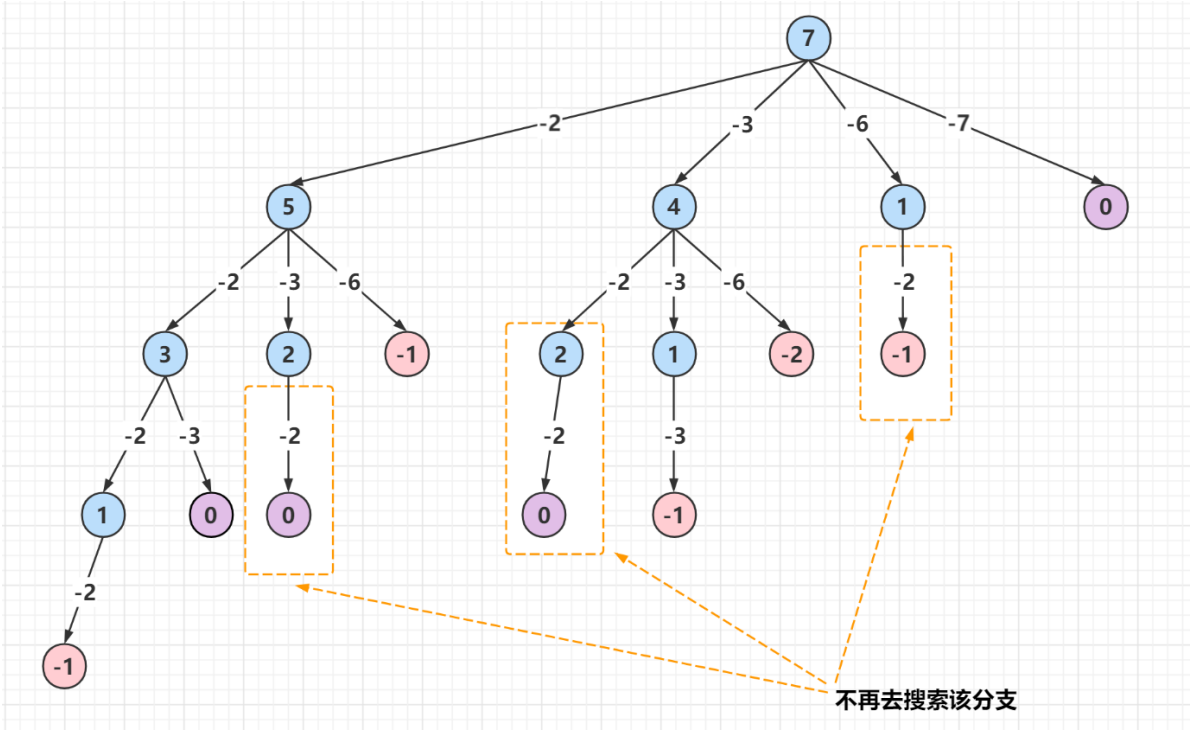
- 1、遍历数组中的每一个数字。
- 2、递归枚举每一个数字可以选多少次，递归过程中维护一个`target`变量。如果当前数字小于等于`target`，我们就将其加入我们的路径数组`path`中，相应的`target`减去当前数字的值。也就是说，每选一个分支，就减去所选分支的值。
- 3、当`target == 0`时，表示该选择方案是合法的，记录该方案，将其加入`res`数组中。

递归树如下，以`candidates = [2,3,6,7]`，`target = 7`为例。



最终答案为：`[[7],[2,2,3]]`，但是我们发现`[[2, 2, 3], [2, 3, 2], [3, 2, 2]`方案重复了。为了避免搜索过程中的重复方案，我们要去定义一个搜索起点，已经考虑过的数，以后的搜索中就不能出现，让我们的每次搜索都从当前起点往后搜索(包含当前起点)，直到搜索到数组末尾。这样我们人为规定了一个搜索顺序，就可以避免重复方案。

如下图所示，处于黄色虚线矩形内的分支都不再去搜索了，这样我们就完成了去重操作。



递归函数设计：

`void dfs(vector<int>&c,int u ,int target)`

变量`u`表示当前枚举的数字下标，`target`是递归过程中维护的目标数。

递归边界：

- 1、`if(target < 0)`，表示当前方案不合法，返回上一层。
- 2、`if(target == 0)`，方案合法，记录该方案。

时间复杂度分析： 无

c++代码

```cpp
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    vector<vector<int>> combinationSum(vector<int>& candidates, int target)
{
        dfs(candidates, 0, target);
        return res;
    }
    void dfs(vector<int>& c, int u, int target){
        if(target < 0) return;
        if(target == 0){
            res.push_back(path);
        }
        for(int i = u; i < c.size(); i++){
            if(c[i] <= target){
                path.push_back(c[i]);
                dfs(c, i, target - c[i]);
                path.pop_back();
            }
        }
    }
};
```

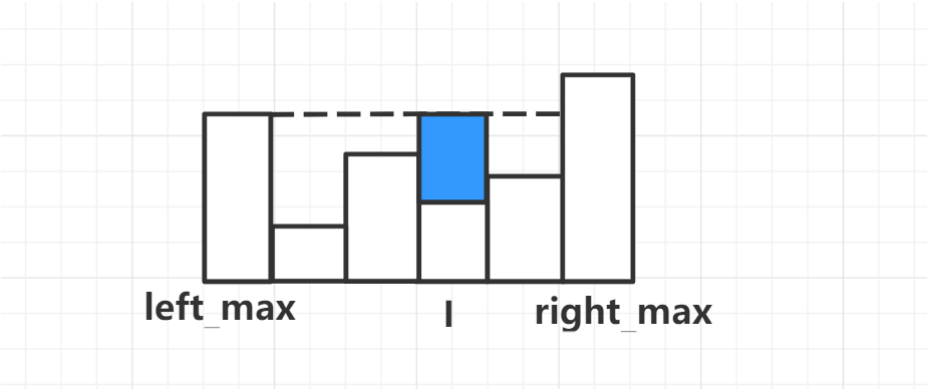
42. 接雨水

思路

(三次线性扫描) $O(n)$

1、观察整个图形，考虑对水的面积按 **列** 进行拆解、

2、注意到，每个矩形条上方所能接受的水的高度，是由它**左边最高的**矩形，和**右边最高的**矩形决定的。具体地，假设第`i`个矩形条的高度为`height[i]`，且矩形条**左边最高的** 矩形条的高度为`left_max[i]`，**右边最高的**矩形条高度为`right_max[i]`，则该矩形条上方能接受水的高度为`min(left_max[i], right_max[i]) - height[i]`。



3、需要分别从左向右扫描求`left_max`，从右向左求`right_max`，最后统计答案即可。

4、注意特判`n`为`0`。

时间复杂度分析： 三次线性扫描，故只需要 $O(n)$ 的时间。

空间复杂度分析： 需要额外 $O(n)$的空间记录每个位置左边最高的高度和右边最高的高度。

c++代码

```cpp
class Solution {
public:
    int trap(vector<int>& h) {
        int n = h.size();
        vector<int> left_max(n);  //每个柱子左边最大值
        vector<int> right_max(n); //每个柱子右边最大值
        left_max[0] = h[0];
        for(int i = 1; i < n; i++){
            left_max[i] = max(left_max[i - 1], h[i]);
        }
        right_max[n - 1] = h[n - 1];
        for(int i = n - 2; i >= 0; i--){
            right_max[i] = max(right_max[i + 1], h[i]);
        }
        int res = 0;
        for(int i = 0; i < n; i++){
            res += min(left_max[i], right_max[i]) - h[i];
        }
        return res;
    }
};
```

46. 全排列

思路

(dfs) $O(n \times n!)$

具体解题过程：

- 1、我们从前往后，一位一位枚举，每次选择一个没有被使用过的数。
- 2、选好之后，将该数的状态改成“已被使用”，同时将该数记录在相应位置上，然后递归下一层。
- 3、递归返回时，不要忘记将该数的状态改成“未被使用”，并将该数从相应位置上删除。

辅助数组：

```
vector<bool> st;          //标记数组
vector<int> path;         //记录路径
```

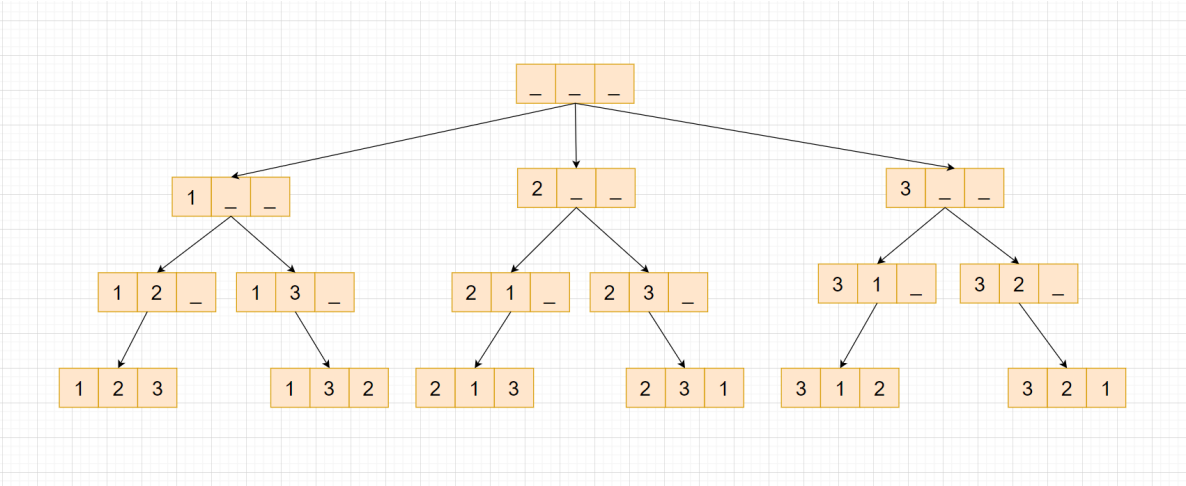
递归函数设计：

```
void dfs(vector<int>& nums, int u)
```

- `nums`是选择数组，`u`是当前正在搜索的答案数组下标位置。

递归搜索树

我们以`1`，`2`，`3`为例：



时间复杂度分析： $O(n \times n!)$，总共$n!$种情况，每种情况的长度为n。

c++代码

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> st;
    vector<int> path;
    vector<vector<int>> permute(vector<int>& nums) {
        st = vector<int>(nums.size() + 1,false);
        dfs(nums, 0);
        return res;
    }
    void dfs(vector<int>& nums, int u){
        if(u == nums.size()){
            res.push_back(path);
```

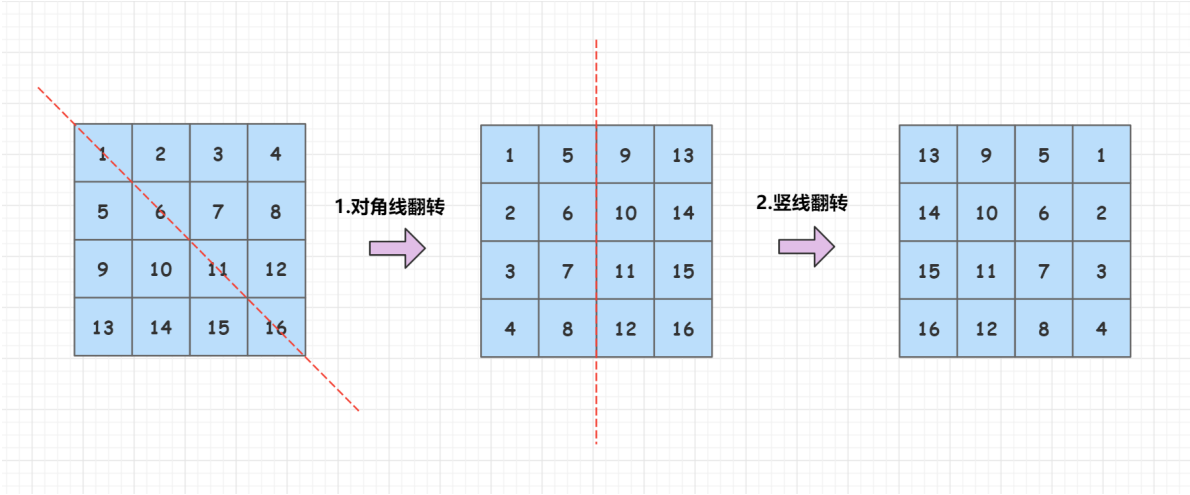
```
            return ;
        }
        for(int i = 0; i < nums.size(); i++){
            if(!st[i]){
                st[i] = true;
                path.push_back(nums[i]);
                dfs(nums, u + 1);
                st[i] = false;
                path.pop_back();
            }
        }
    }
};
```

48. 旋转图像

思路

(操作分解) $O(n^2)$

我们对观察样例，找规律发现：先以**左上-右下对角条线**为轴做翻转，再以**中心的竖线**为轴做翻转，就可以顺时针翻转90度。



因此可以得出一个结论，顺时针90度应该是左上/右下对角线翻转+左右翻转，或者右上/左下对角线翻转+上下翻转。

过程如下：

1. 先以左上-右下对角条线为轴做翻转；
2. 再以中心的竖线为轴做翻转；

时间复杂度分析：$O(n^2)$，额外空间：$O(1)$。

c++代码

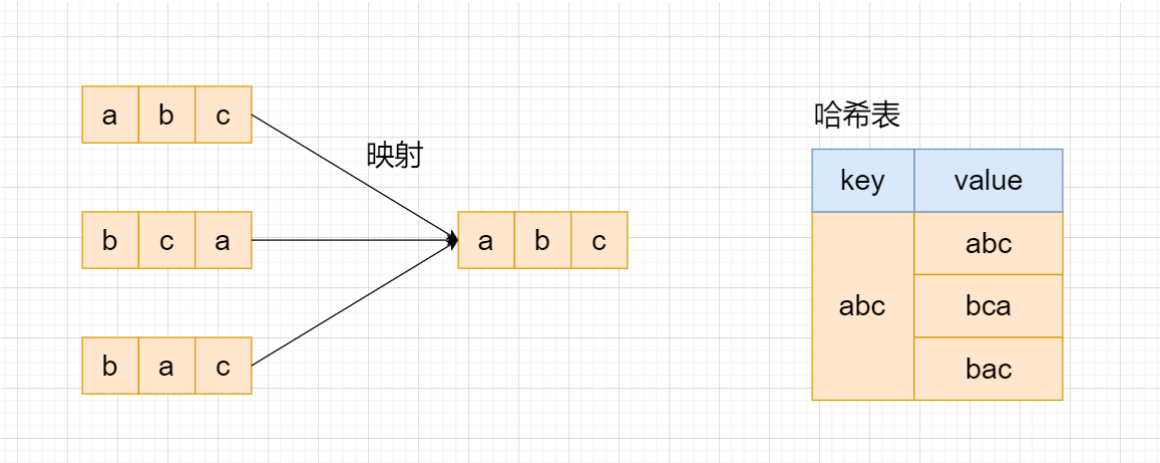
```cpp
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int n = matrix.size();
        for(int i = 0; i < n; i++)
            for(int j = 0; j < i; j++)
                swap(matrix[i][j], matrix[j][i]);
        for(int i = 0; i < n; i++)
            for(int j = 0, k = n - 1; j < k; j++, k--)
                swap(matrix[i][j], matrix[i][k]);
    }
};
```

49. 字母异位词分组

思路

(哈希 + 排序) $O(NLlogL)$

定义从 `string` 映射到 `vector<string>` 的哈希表：`unordered_map<string, vector<string>>`。然后将每个字符串的所有字符从小到大排序，将排好序的字符串作为 `key`，然后将原字符串插入 `key` 对应的 `vector<string>` 中。



具体过程如下：

- 1、定义一个 `string` 映射到 `vector<string>` 的哈希表。
- 2、遍历 `strs` 字符串数组，对于每个字符串 `str`：
  - 将 `str` 排序，作为哈希表的 `key` 值；
  - 将原 `str` 放入对应 `key` 值位置处；
- 3、最后遍历整个哈希表，将对应的 `vector<string>` 存入 `res` 中。

时间复杂度分析： 对于每个字符串，哈希表和 `vector` 的插入操作复杂度都是 $O(1)$，排序复杂度是 $O(LlogL)$。所以总时间复杂度是 $O(NLlogL)$。

c++代码

```cpp
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> hash;  //哈希表
        for(string str : strs){
            string nstr = str;
            sort(nstr.begin(), nstr.end());  //排序 将其作为key值
```

```
            hash[nstr].push_back(str);
        }
        vector<vector<string>> res;
        for(auto item : hash){
            res.push_back(item.second);
        }
        return res;
    }
};
```

53. 最大子数组和

思路

(动态规划) $O(n)$

状态表示： `f[i]`表示以`nums[i]`为结尾的最大连续子数组和。

状态计算：

如何确定`f[i]`的值？ 以`nums[i]`为结尾的连续子数组共分为两种情况：

- 只有`nums[i]`一个数，则`f[i] = nums[i]`；
- 以`nums[i]`为结尾的多个数，则`f[i] = f[i - 1] + nums[i]`。

两种情况取最大值，因此**状态转移方程为：** `f[i] = max(f[i - 1] + nums[i], nums[i])`。

初始化：

`f[0] = nums[0]`。

最后遍历每个位置的`f[i]`，然后其中的最大值即可。

时间复杂度分析： 只遍历一次数组，$O(n)$。

c++代码

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        vector<int> f(n + 1, 0);
        int res = nums[0];
        f[0] = nums[0];
        for(int i = 1; i < n; i++){
            f[i] = max(f[i - 1] + nums[i], nums[i]);
            res = max(res, f[i]);
        }
        return res;
    }
};
```

55. 跳跃游戏

思路

(贪心) $O(n)$

从前往后遍历`nums`数组，记录我们能跳到的最远位置`j`，如果存在我们不能跳到的下标`i`，返回`false`即可，否则返回`true`。

具体过程如下：

- 1、定义一个`j`变量用来记录我们可以跳到的最远位置，初始化`j = 0`。
- 2、遍历整个`nums[]`数组，`i`表示当前需要跳到的下标位置。
  - 若`j < i`，说明下标`i`不可达，则返回`false`；
  - 否则，说明`i`可达，则我们以`i`为起点更新可以跳到的最远位置`j`，即`j = max(j, i + nums[i])`；
- 3、如果可以遍历完整个数组，说明可以到达最后一个下标`i`，我们返回`true`。

时间复杂度分析： 只遍历一次数组，因此时间复杂度为$O(n)$。

c++代码

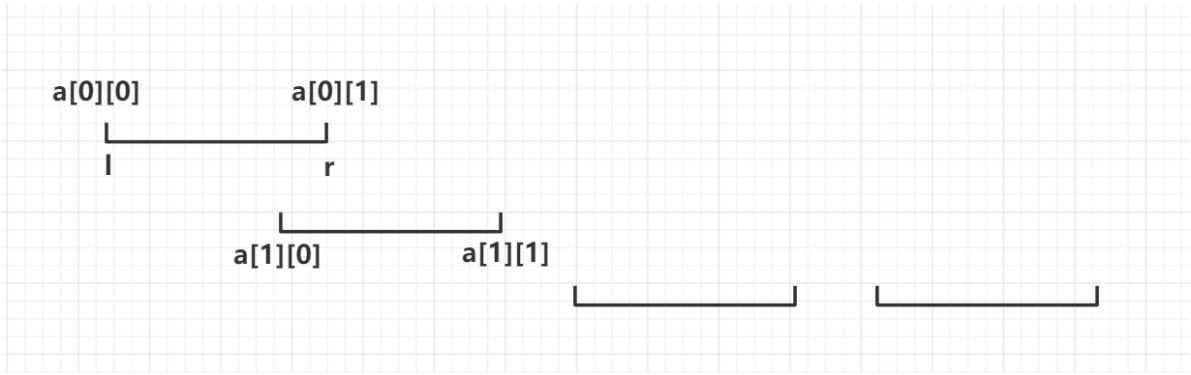
```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        for(int i = 0, j = 0; i < nums.size(); i++){
            if(j < i) return false;
            j = max(j, i + nums[i]);
        }
        return true;
    }
};
```

56. 合并区间

思路

(数组，排序) $O(nlogn)$

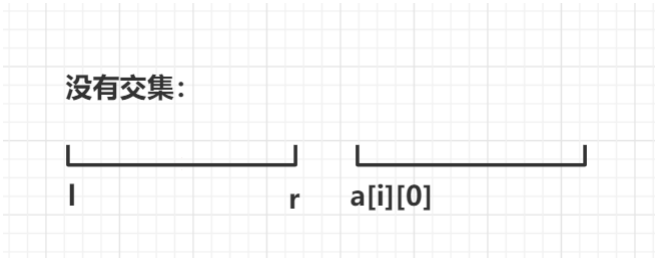
1、将所有的区间按照左端点从小到大排序



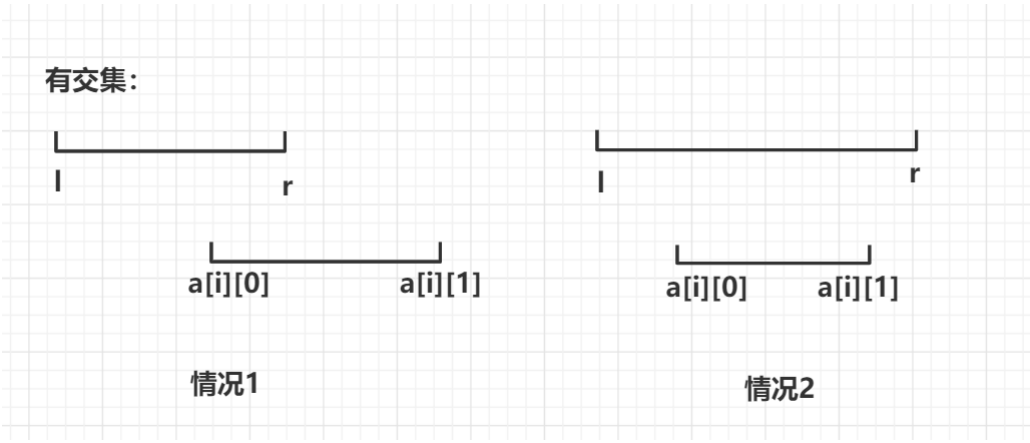
2、定义区间左端点`l = a[0][0]`，右端点`r = a[0][1]`（等价于两个左右指针），我们从前往后遍历每个区间：

- 如果当前区间和上一个区间没有交集，也就是说当前区间的左端点`>`上一个区间的右端点，即`a[i][0] > r`，说明上一个区间独立，我们将上一个区间的左右端点`[l,r]`加入答案数组中，并更新左端点`l`，右端点`r`为当前区间的左右端点，即`l = a[i][0], r = a[i][1]`。

  始终维持`l`和`r`为最新独立区间的左右端点。



- 如果当前区间和上一个区间有交集，即当前区间的左端点`<=`上一个区间的右端点，我们让左端点`l`保持不变，右端点`r`更新为`max(r,a[i][1])`，进行区间的合并 。



3、最后再将最后一个合并或者未合并的独立区间`[l,r]`加入答案数组中。

时间复杂度分析： 遍历区间数组的时间为$O(n)$，对区间数组进行排序的时间复杂度为$O(nlogn)$,因此总的时间复杂度为$O(nlogn)$

c++代码

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& a) {
        vector<vector<int>> res;
        sort(a.begin(), a.end());  //按照左端点排序
        int l = a[0][0], r = a[0][1];
        for(int i = 1; i < a.size(); i++){
            if(a[i][0] > r){
                res.push_back({l, r});
                l = a[i][0], r = a[i][1];
            }else{
                r = max(r, a[i][1]);
            }
        }
        res.push_back({l, r});
        return res;
    }
};
```