力扣——用力扣向城墙

【9.27】414

1.set

在 `set` 中，元素是自动按照升序排序的，因此：

插入元素：每次使用 `insert` 方法将元素添加到 `set` 时， `set` 会自动调整其内部结构，以保持所有元素的排序。这意味着最小的元素总是位于 `set` 的开始位置，即 `s.begin()` 所指向的地方。

2.set的erase（）库函数

1）**删除某个迭代器位置的元素**：

```
s.erase(s.begin());
```

2）**删除特定的值**：

```
s.erase(value);
```

3）**删除特定范围的元素**：

```
s.erase(start_iter, end_iter);
```

删除从 `start_iter` 到 `end_iter` 范围内的所有元素。

【9.28】11

1.双指针代表了什么？

双指针代表的是 **可以作为容器边界的所有位置的范围**。在一开始，双指针指向数组的左右边界，表示数组中所有的位置都可以作为容器的边界，因为我们还没有进行过任何尝试。在这之后，我们每次将对应的数字较小的那个指针 往 另一个指针 的方向移动一个位置，就表示我们认为 这个指针不可能再作为容器的边界了。

2.为什么对应的数字较小的那个指针不可能再作为容器的边界了？

在上面的分析部分，我们对这个问题有了一点初步的想法。这里我们定量地进行证明。

考虑第一步，假设当前左指针和右指针指向的数分别为 x 和 y，不失一般性，我们假设 x⩽y。同时，两个指针之间的距离为 t。那么，它们组成的容器的容量为：

min(x,y)⊠t=x⊠t

我们可以断定，如果我们保持左指针的位置不变，那么无论右指针在哪里，这个容器的容量都不会超过 x⊠t 了。注意这里右指针只能向左移动，因为 我们考虑的是第一步，也就是 指针还指向数组的左右

边界的时候。

> 我们任意向左移动右指针，指向的数为 y_1，两个指针之间的距离为 t_1，那么显然有 $t_1 < t$，并且 $\min(x, y_1) \leq \min(x, y)$：
>
> - 如果 $y_1 \leq y$，那么 $\min(x, y_1) \leq \min(x, y)$；
> - 如果 $y_1 > y$，那么 $\min(x, y_1) = x = \min(x, y)$。
>
> 因此有：
>
> $$\min(x, y_t) * t_1 < \min(x, y) * t$$

即无论我们怎么移动右指针，得到的容器的容量都小于移动前容器的容量。也就是说，这个左指针对应的数不会作为容器的边界了，那么我们就可以丢弃这个位置，将左指针向右移动一个位置，此时新的左指针于原先的右指针之间的左右位置，才可能会作为容器的边界。

这样以来，我们将问题的规模减小了 1，被我们丢弃的那个位置就相当于消失了。此时的左右指针，就指向了一个新的、规模减少了的问题的数组的左右边界，因此，我们可以继续像之前 考虑第一步 那样考虑这个问题：

求出当前双指针对应的容器的容量；

对应数字较小的那个指针以后不可能作为容器的边界了，将其丢弃，并移动对应的指针。

【9.29】13

1. 经典的表驱动

```
    int char2num(char c){
        static map<char, int> romanNum = {
            {'I', 1},
            {'V', 5},
            {'X', 10},
            {'L', 50},
            {'C', 100},
            {'D', 500},
            {'M', 1000}
        };
        return romanNum[c];
    }
```

【9.29】686

1. 判断是否是字符串的子串

在 C++ 中，可以使用 `std::string` 类提供的 `find()` 函数来判断一个字符串是否是另一个字符串的子串。`find()` 函数会在目标字符串中查找指定的子串，如果找到则返回子串在目标字符串中的起始位置，如果找不到则返回 `std::string::npos`。

方法一：最原始的，也是我理想中的思路

寻找多少个重复的a中，能找到b

```
// 计算a可能的重复次数
        int k = (b.size() + a.size() - 1) / a.size() + 1; // 向上取整
//可能重复1-k次
```

!!! Rabin-Karp 算法

!!! KMP算法

一句话概括：**快速从主串中找到模式串**。

KMP 为什么相比于朴素解法更快：

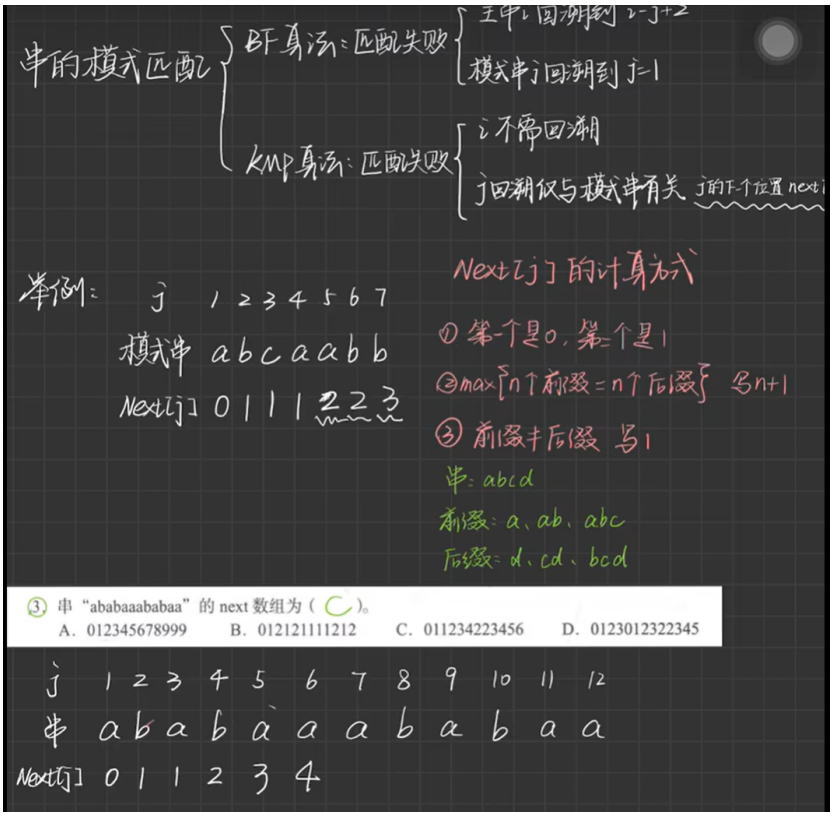
- 因为 KMP 利用已匹配部分中相同的「前缀」和「后缀」来加速下一次的匹配。
- 因为 KMP 的原串指针不会进行回溯（没有朴素匹配中回到下一个「发起点」的过程）。

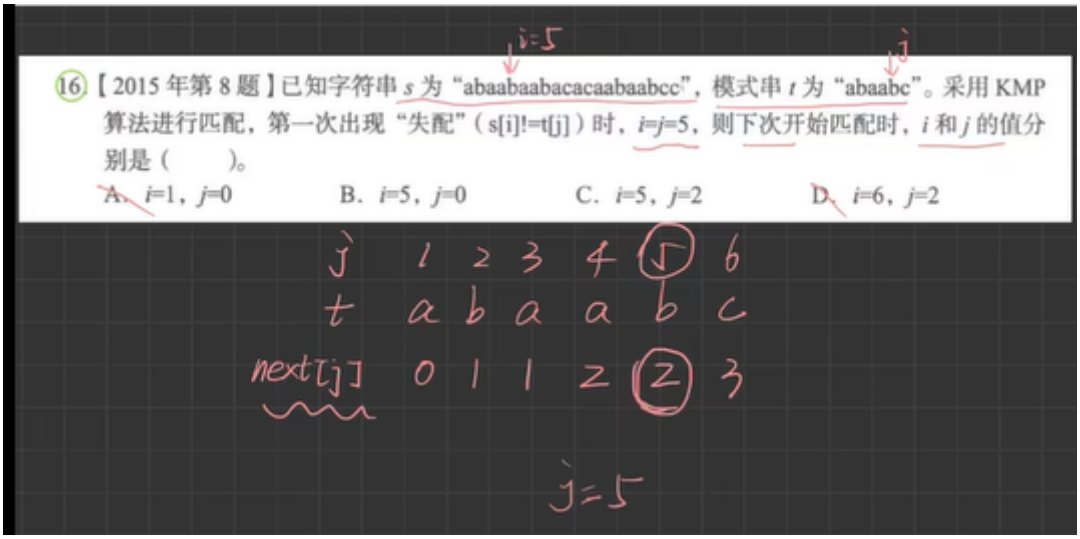
最长公共前后缀：ababa的最长公共前后缀是aba（不能等于子串本身）

比较指针停在第一个不匹配的字符上，在模式串比较指针左边，找最长公共前后缀；

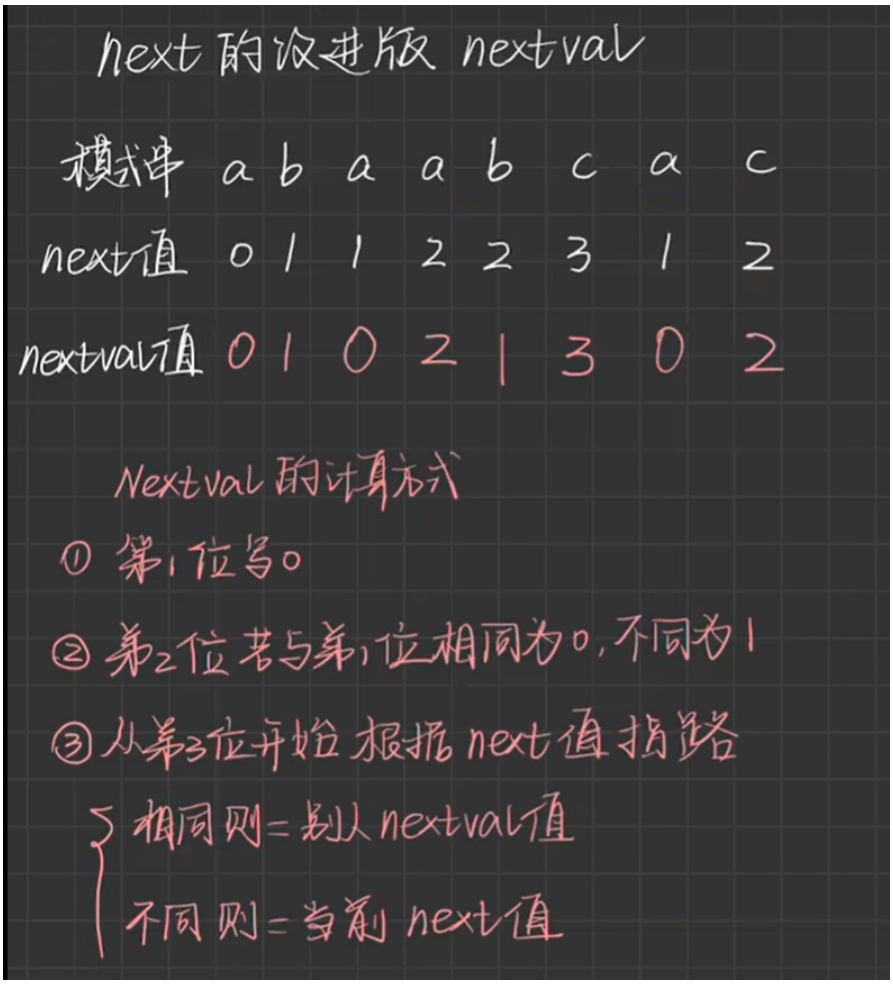
移动模式串，使公共前缀来到公共后缀原先的位置，从比较指针开始继续比较。

next数组由模式串决定


串的模式匹配
BF算法：匹配失败
主串i回溯到 i=j+2
模式串j回溯到 j=1
KMP算法：匹配失败
i不需回溯
j回溯仅与模式串有关 j的下一个位置 next[j]
举例：
j 1 2 3 4 5 6 7
模式串 a b c a a b b
Next[j] 0 1 1 1 2 2 3
Next[j]的计算方式
① 第一个是0，第二个是1
② max{n个前缀=n个后缀} 为n+1
③ 前缀≠后缀 为1
串：abcd
前缀：a、ab、abc
后缀：d、cd、bcd
③ 串“ababaaababaa”的next数组为（ C ）。
A. 012345678999 B. 012121111212 C. 011234223456 D. 0123012322345
j 1 2 3 4 5 6 7 8 9 10 11 12
串 a b a b a a a b a b a a
Next[j] 0 1 1 2 3 4


i=5
j
⑯【2015年第8题】已知字符串s为“abaabaabacacaabaabcc”，模式串t为“abaabc”。采用KMP算法进行匹配，第一次出现“失配”（s[i]!=t[j]）时，i=j=5，则下次开始匹配时，i和j的值分别是（ ）。
A. i=1，j=0 B. i=5，j=0 C. i=5，j=2 D. i=6，j=2
j 1 2 3 4 5 6
t a b a a b c
next[j] 0 1 1 2 2 3
j=5

衍生出nextval




```
1  void get_Next(string s, int next[])                //这个函数对字符串s进行预处理得到next数组
2  {
3          int j = 0;
4          next[0] = 0;          //初始化
5          for(int i = 1; i<s.size(); i++){          //i指针指向的是后缀末尾，j指针指向的是前缀末尾
6                  while(j>0&&s[i]!=s[j])    j = next[j-1];          //前后缀不相同，去找j前一位的最长相等前后缀
7                  if(s[i]==s[j])    j++;          //前后缀相同，j指针后移
8                  next[i] = j;          //更新next数组
9          }
10 }
11
12
13 int strSTR(string s, string t)          //这个函数是从s中找到t，如果存在返回t出现的位置，如果不存在返回-1
14 {
15         if(t.size()==0)          return 0;
16         get_Next(t, next);
17         int j = 0;
18         for(int i = 0; i < s.size(); i++){
19                 while(j>0&&s[i]!= t[j]){
```

```
                    j = next[j-1];
                }
                if(s[i]==t[j]) j++;
                if(j==t.size()) return i - t.size() + 1;
        }
        return -1;
}

```

```
void getNext(char* s,int len){
    next[0] = 0;
    int k = 0; //k = next[0]
    int i = 1;
    while(i < len){
        if(s[i] == s[k]){
            next[i++] = ++k; //next[j+1] = k+1;
        }else{
            if(k > 0)k = next[k-1]; //k = next[k-1]
            else{
                next[i++] = k; //next[j+1] = 0 回溯到头了，找不到相同前缀，则最大相同前后缀长度=0
            }
        }
    }
}

//返回模式串T中字串S第一次出现的位置下标，找不到则返回-1
int kmp(char *T, char* S){
    int len_T = strlen(T);
    int len_S = strlen(S);
    for(int i = 0,j = 0;i<len_T;i++){
        while(j > 0 && T[i] != S[j])j = next[j-1];
        if(T[i] == S[j])j++;
        if(j == len_S)return i-len_S+1;
    }
    return -1;
}

//返回模式串T中字串S出现的次数，找不到则返回0
int kmp(char *T, char* S){
    int sum = 0;
    int len_T = strlen(T);
    int len_S = strlen(S);
    for(int i = 0,j = 0;i<len_T;i++){
        while(j > 0 && T[i] != S[j])j = next[j-1];
```

```
        if(T[i] == S[j])j++;
        if(j == len_S){
            sum++;
            j = next[j-1];
        }
    }
    return sum;
}
```

结合题解理解KMP

我觉得求next的函数直接别下来，kmp匹配部分也直接背诵

```
class Solution {
public:
    int strStr(string &haystack, string &needle) {
        int n = haystack.size(), m = needle.size();
        if (m == 0) {
            return 0;
        }
        vector<int> pi(m); //存储每个位置之前的字符串的最长相等前后缀的长度

        //1.构造next数组pi
        for (int i = 1, j = 0; i < m; i++) {
            while (j > 0 && needle[i] != needle[j]) {
                j = pi[j - 1];
            }
            if (needle[i] == needle[j]) {
                j++;
            }
            pi[i] = j;
        }

        //KMP查找匹配位置
        for (int i = 0, j = 0; i - j < n; i++) { // b 开始匹配的位置是否超过第一个叠加的 a
            while (j > 0 && haystack[i % n] != needle[j]) { // haystack 是循环叠加的字符串，所以取 i % n
                j = pi[j - 1];
            }
            if (haystack[i % n] == needle[j]) {
                j++;
            }
            if (j == m) {
                return i - m + 1;
            }
```

```
        }
        return -1;
    }

    int repeatedStringMatch(string a, string b) {
        int an = a.size(), bn = b.size();
        int index = strStr(a, b);
        if (index == -1) {
            return -1;
        }
        if (an - index >= bn) {
            return 1;
        }
        return (bn + index - an - 1) / an + 2;
    }
};

```

【10.1】819

两个需要学习的数据结构：

`unordered_set<string>`：无序集合

用于存储不允许重复的元素，常用于快速查找、插入、删除。这里用来存储被禁止的单词。

- 方法：
- `emplace()` 插入一个元素到集合中，性能比 `insert()` 更高效。
- `count()` 检查集合中是否存在某个元素，返回 1（存在）或 0（不存在）。

`unordered_map<string, int>`：无序映射（哈希表）

用来存储键值对。在这段代码中，用于记录单词的出现频率。

- 方法：
- `[]` 根据键访问对应的值，**如果键不存在则插入一个默认值（还有buff！！！）**
- `insert()`

【10.1】8

1. 字符串取子串

str2 = str1.substr(beginIndex);

2. 比较字符串和字面量

字符串字面量（如 `"-"`）实际上是一个 `const char*` 类型（字符指针），而 `s[count]` 是一个 `char` 类型

```
if (s[count] == "-")  // 错误：s[count] 是 char，"-" 是 const char*
if (s[count] == '-')  // 正确：s[count] 是 char，'-' 也是 char
```

字符串（如 `std::string`）和字符串字面量（如 `"hello"`）的比较是可以直接进行的，因为 `std::string` 支持与 `const char*`（即字符串字面量）进行比较

```
if (s == "hello") //正确
```

3. 整数溢出的判断

如果整数数超过 32 位有符号整数范围 `[-2(31),  2(31 )- 1]`，需要截断这个整数，使其保持在这个范围内。具体来说，小于 `-2(31)` 的整数应该被舍入为 `-2(31)`，大于 `2(31 )- 1` 的整数应该被舍入为 `2(31 )- 1`。

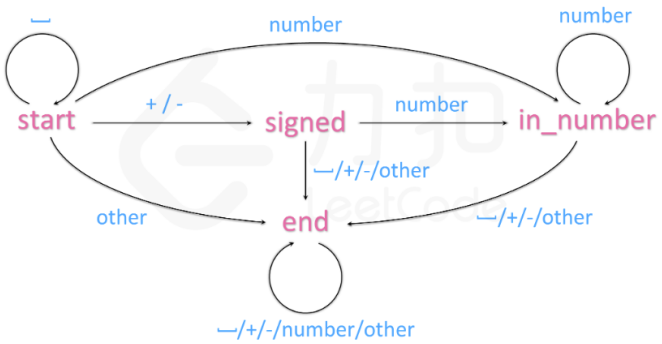
```
if (temp < INT_MIN) {
    temp = INT_MIN;
} else if (temp > INT_MAX) {
    temp = INT_MAX;
}
```

4. 转换字符串为数字 `stringstream`

使用 `std::stringstream` 来转换字符串为数字

```
#include <sstream> // 需要引入头文件
long long temp = 0;
std::stringstream ss(ansStr);
ss >> temp;
```

【!!!】巧妙的自动机！



我们也可以用下面的表格来表示这个自动机：

	' '	+/-	number	other
start	start	signed	in_number	end
signed	end	end	in_number	end
in_number	end	end	in_number	end
end	end	end	end	end

```
class Automaton {
    string state = "start";
    unordered_map<string, vector<string>> table = {
        {"start", {"start", "signed", "in_number", "end"}},
        {"signed", {"end", "end", "in_number", "end"}},
        {"in_number", {"end", "end", "in_number", "end"}},
        {"end", {"end", "end", "end", "end"}}
    };

    int get_col(char c) {
        if (isspace(c)) return 0;
        if (c == '+' or c == '-') return 1;
        if (isdigit(c)) return 2;
        return 3;
    }
public:
    int sign = 1;
    long long ans = 0;

    void get(char c) {
        state = table[state][get_col(c)];
        if (state == "in_number") {
            ans = ans * 10 + c - '0';
            ans = sign == 1 ? min(ans, (long long)INT_MAX) : min(ans, -(long
long)INT_MIN);
        }
        else if (state == "signed")
            sign = c == '+' ? 1 : -1;
```

```
    }
};

class Solution {
public:
    int myAtoi(string str) {
        Automaton automaton;
        for (char c : str)
            automaton.get(c);
        return automaton.sign * automaton.ans;
    }
};

作者：力扣官方题解
链接：https://leetcode.cn/problems/string-to-integer-atoi/solutions/183164/zi-fu-chuan-zhuan-huan-zheng-shu-atoi-by-leetcode-/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

【10.7】17

【!!!】回溯算法

在平板上

【10.7】1513

1.bug：在循环中，最后一组忘记处理

2.经验

先找最长的全1字符串，在根据长度计算子集

【10.8】21

1. `new` 分配动态内存给新的节点，而不是通过临时对象赋值

```
p->next = &ListNode(p2->val);
/*报错：
Line 20: Char 27: error: taking the address of a temporary object of type 'ListNode' [-Waddress-of-temporary]
20 |                 p->next = &ListNode(p1->val);
   |                           ^~~~~~~~~~~~~~~~~~
*/
```

原因：使用了 `&ListNode(p1->val)` 来试图创建一个新节点并将其地址赋值给 `p->next`。但是 `ListNode(p1->val)` 是一个临时对象，C++ 中的临时对象会在表达式结束后被销毁。因此，你试图获取这个临时对象的地址是无效的，这会导致运行时的错误。

改进：使用 `new` 运算符来分配动态内存给新的节点，而不是通过临时对象赋值。

```
1  p->next = new ListNode(p1->val);  // Dynamically allocate a new node
```

2. 链表合并节省时间

```
1 p->next = new ListNode(p1->val, p1->next);
2 p->next = p1;  //更快，更省空间
```

【10.8】206

1. 用递归做链表反转 !amazing

```
1  //递归：确认过眼神，是我写不出来的代码
2      ListNode* reverseList(ListNode* head) {
3          if(head == nullptr || head->next == nullptr){
4              return head;
5          }
6          ListNode *newHead = reverseList(head->next);
7          head->next->next = head;
8          head->next = nullptr;
9          return newHead;
10     }
```

假设链表为：

$$n_1 \to \ldots \to n_{k-1} \to n_k \to n_{k+1} \to \ldots \to n_m \to \varnothing$$

若从节点 n_{k+1} 到 n_m 已经被反转，而我们正处于 n_k。

$$n_1 \to \ldots \to n_{k-1} \to n_k \to n_{k+1} \leftarrow \ldots \leftarrow n_m$$

我们希望 n_{k+1} 的下一个节点指向 n_k。

所以，$n_k.next.next = n_k$。

因为代码中，是先递归再处理当前节点，所以途中，指针反转是从后向前进行的。

【10.8】237

1. 巧妙的思路

删除链表指定节点，但不知道头结点：

方法：和下一节点交换

```
1     void deleteNode(ListNode* node) {
2         while(node != NULL && node->next != NULL){  //这一行有两个条件，因为node = node->next后，node != NULL时，不能保证node->next != NULL
3             node->val = node->next->val;
4             node->next = node->next->next;
5             node = node->next;
6         }
7     }
```

2. nullptr和NULL的区别

【10.14】452

方法1

代码随想录

1. 思路

1.按左边界升序排序

2.对第i个气球，比较i的右边界和i+1的左边界

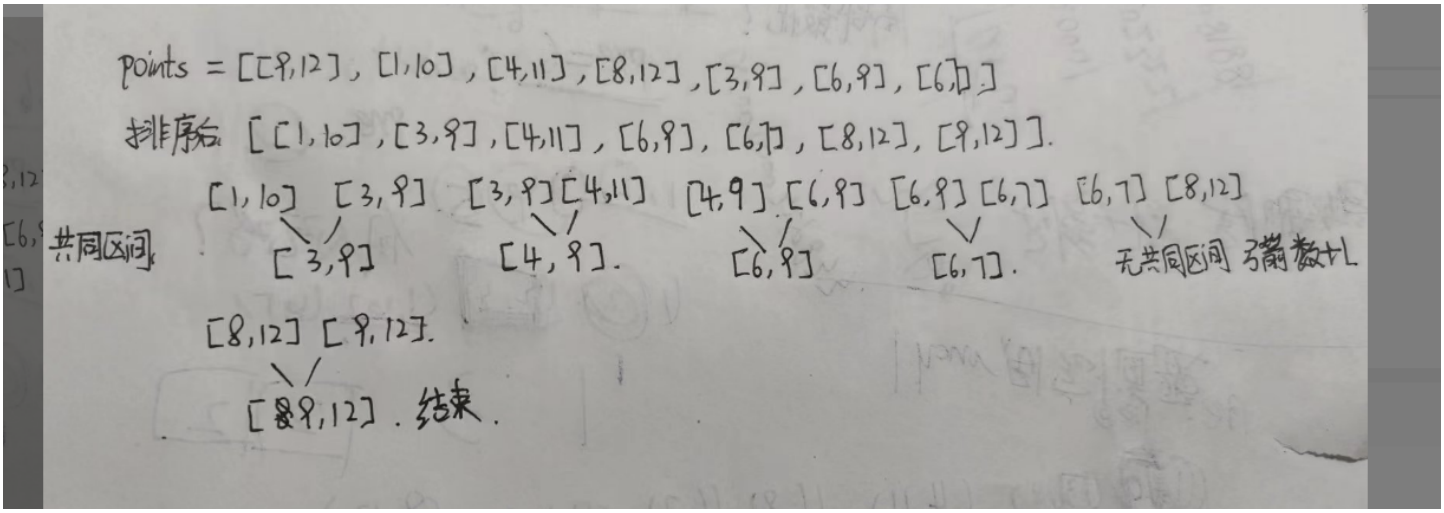
- 小，箭个数加一，i++
- 大，更新有边界为i和i+1中更小的右边界，考虑i+2的左边界

【和c++的第二次作业联系】

按照右边界排序更好、适应性更强？暂未探索

2. 循环很巧妙的从i=1开始

方法2



【10.18】27

【!!!双指针法】

双指针法（快慢指针法）在数组和链表的操作中是非常常见的，很多考察数组、链表、字符串等操作的面试题，都使用双指针法。

【10.18】{回溯-组合}39

回溯

【10.18】{回溯-组合}40

回溯plus

元素不能重复

1. 查找vector中是否已存在当前元素：

```
if(find(vec.begin(), vec.end(), target) != vec.end()){
    //已经存在，返回的是该元素的迭代器
}else{
    //没有找到，返回vec.end()
}
```

2. sort是一个就地排序算法，并不返回排序后的容器，返回void

```
1 if(find(t.begin(), t.end(), sort(path.begin(), path.end())) == t.end()) //不对
```

3. 思路（题解）

- 和39题的区别：

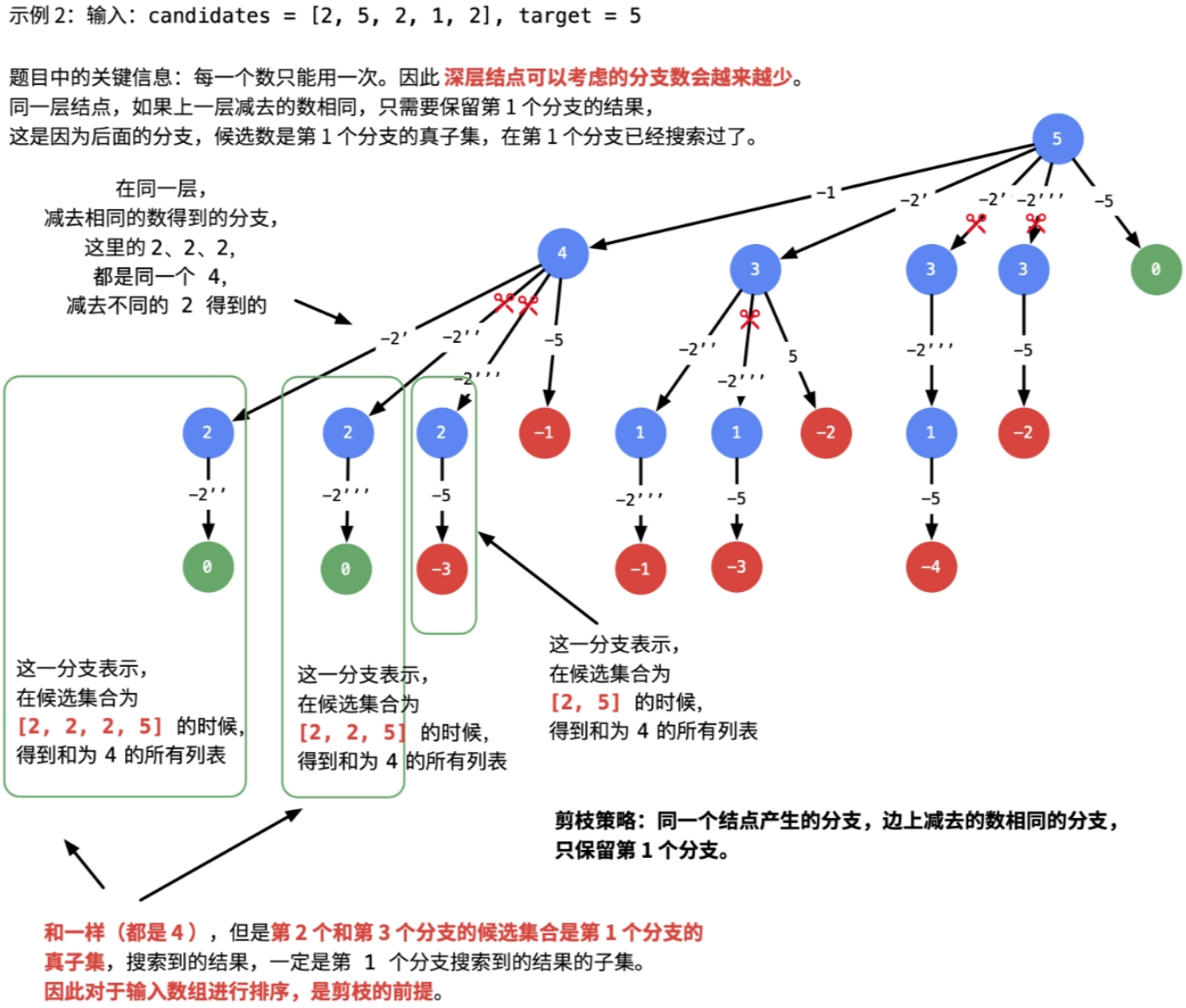
每个组合中每个下表的数字只能出现一次

- 如何去掉重复的集合（重点）

为了使得解集不包含重复的组合。有以下 2 种方案：

（1）使用 哈希表 天然的去重功能，但是编码相对复杂；

（2）这里我们使用和第 39 题和第 15 题（三数之和）类似的思路：不重复就需要按 顺序 搜索， 在搜索的过程中检测分支是否会出现重复结果 。注意：这里的顺序不仅仅指数组 candidates 有序，还指按照一定顺序搜索结果。



【10.20】{回溯-排列}46

回溯

【10.20】{回溯-排列}47

和40题一样画树状图，判断什么时候为了避免重复而剪枝

【10.21】{回溯-子集}78

对于每个元素，有“存在”和“不存在”两种情况

两种做法：

第一种，破坏了在回溯函数开头判断回收条件的做法

```
1 class Solution {
2 public:
3     vector<vector<int>> ans;
```

```
    vector<int> subset;
    vector<int> used;
    void bt(int idx, vector<int> &nums){
        ans.push_back(subset);
        for(int i = idx; i < nums.size(); i++){
            if((i > 0 && nums[i] == nums[i-1]) && used[i-1] == 0) continue;
            subset.push_back(nums[i]);
            used[i] = 1;
            bt(i+1, nums);
            subset.pop_back();
            used[i] = 0;
        }

    }
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        used.resize(nums.size(), 0);
        bt(0, nums);
        return ans;
    }
};
```

第二种，破坏了回溯函数中，单层中的for循环的结构

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> subset;
    vector<int> used;
    void bt(int idx, vector<int> &nums){
        if(idx == nums.size()){
            ans.push_back(subset);
            return;
        }
        bt(idx+1, nums);

        if(idx > 0 && nums[idx] == nums[idx-1] && used[idx-1] == 0) return;

        subset.push_back(nums[idx]);
        used[idx] = 1;
        bt(idx+1, nums);
        subset.pop_back();
        used[idx] = 0;


```

```
22
23      }
24      vector<vector<int>> subsetsWithDup(vector<int>& nums) {
25          sort(nums.begin(), nums.end());
26          used.resize(nums.size(), 0);
27          bt(0, nums);
28          return ans;
29      }
30  };
```

总结：不能重复问题

第一种，改造回溯函数的结构；

第二种，在单层for循环中加入条件判断，通常对总数据集排序，然后再for中判断前一个元素和后一个元素的关系，有时候要使用used数组，记录每个元素是否已被使用。

【10.22】{回溯-复杂}679

24点，

1. 我的思路（没过）

数字的排列本质是<4个数字的排列>，四个数字中可能有重复的，因此类比47题<不重复的全排列>

Expr = num op num　op有四种，每一次都是随机选择，单层循环有4种选择

expr有“加（）”和“不加（）”两种选择，类比子集问题中的“在自己中”和“不在子集中”

> 不好过的原因：运算符的优先级、数字的排列都很复杂

2. 题解

依次运算两个数-即将两个数合并成为一个数

一共有 4 个数和 3 个运算操作，因此可能性非常有限。一共有多少种可能性呢？

首先从 4 个数字中有序地选出 2 个数字，共有 $4 \times 3 = 12$ 种选法，并选择加、减、乘、除 4 种运算操作之一，用得到的结果取代选出的 2 个数字，剩下 3 个数字。

然后在剩下的 3 个数字中有序地选出 2 个数字，共有 $3 \times 2 = 6$ 种选法，并选择 4 种运算操作之一，用得到的结果取代选出的 2 个数字，剩下 2 个数字。

最后剩下 2 个数字，有 2 种不同的顺序，并选择 4 种运算操作之一。

因此，一共有 $12 \times 4 \times 6 \times 4 \times 2 \times 4 = 9216$ 种不同的可能性。

可以通过回溯的方法遍历所有不同的可能性。具体做法是，使用一个列表存储目前的全部数字，每次从列表中选出 2 个数字，再选择一种运算操作，用计算得到的结果取代选出的 2 个数字，这样列表中的数字就减少了 1 个。重复上述步骤，直到列表中只剩下 1 个数字，这个数字就是一种可能性的结果，如果结果等于 24，则说明可以通过运算得到 24。如果所有的可能性的结果都不等于 24，则说明无法通过运算得到 24。

实现时，有一些细节需要注意。

- 除法运算为实数除法，因此结果为浮点数，列表中存储的数字也都是浮点数。在判断结果是否等于 24 时应考虑精度误差，这道题中，误差小于 10^{-6} 可以认为是相等。
- 进行除法运算时，除数不能为 0，如果遇到除数为 0 的情况，则这种可能性可以直接排除。由于列表中存储的数字

【1102】{贪心}455

排序+双指针+贪心

【1102】{贪心}376 摆动序列

符合要求的最长子序列

踩过的坑

1. str取子串后，len改变了，要再次令len = str.size()。下次取子串可以重新赋给newStr，避免一些错误。
2. 在循环中，最后一组忘记处理