

常用STL

2020年3月1日 14:12

C++ STL从广义来讲包括了三类：算法，容器和迭代器。

算法包括排序，复制等常用算法，以及不同容器特定的算法。容器就是数据的存放形式，包括序列式容器和关联式容器，序列式容器就是list，vector等，关联式容器就是set，map等。

迭代器就是在不暴露容器内部结构的情况下对容器的遍历

标准库中的容器主要分为三类：顺序容器、关联容器、容器适配器。

顺序容器包括五种类型：

array数组：固定大小数组，支持快速随机访问，但不能插入或删除元素；

vector动态数组：支持快速随机访问，尾位插入和删除的速度很快；

deque双向队列：支持快速随机访问，首尾位置插入和删除的速度很快；（可以看作是vector的增强版，与vector相比，可以快速地在首位插入和删除元素）

list双向链表：只支持双向顺序访问，任何位置插入和删除的速度都很快；

forward_list单向链表：只支持单向顺序访问，任何位置插入和删除的速度都很快

关联容器包含两种类型：

map容器：map关联数组：用于保存关键字-值对；

multimap：关键字可重复出现的map；

unordered_map：用哈希函数组织的map；

unordered_multimap：关键词可重复出现的unordered_map；

set容器：set：只保存关键字；

multiset：关键字可重复出现的set；

unordered_set：用哈希函数组织的set；

unordered_multiset：关键词可重复出现的unordered_set；

容器适配器包含三种类型：stack栈、queue队列、priority_queue优先队列。

map的下标运算符[]的作用是：将key作为下标去执行查找，并返回相应的值；如果不存在这个key，就将一个具有该key和value的某人值插入这个map

erase()函数，只能删除内容，不能改变容量

erase成员函数，它删除了itVect迭代器指向的元素，并且返回要被删除的itVect之后的迭代器，

clear()函数，只能清空内容，不能改变容量大小；如果要想在删除内容的同时释放内存，那么你可以选择deque容器。

unordered_map，根据C++标准：任何对STL集合的读取访问都是线程安全的。写入操作不是线程安全的。如果你混合使用读写操作，它也不是线程安全的

vector底层是一个动态数组

包含三个迭代器，start和finish之间是已经被使用的空间范围，end_of_storage是整块连续空间包括备用空间的尾部

当空间不够装下数据（vec.push_back(val)）时，会自动申请另一片1.5/2倍更大的空间，将原来的数据拷贝到新的内存空间，接着释放原来的那片空间【vector内存增长机制】

vec.clear() 其存储空间不释放，仅仅是清空了里面的数据

reserve是改变这个capacity大小到n 只能扩容 总体内存空间

resize()可以size的大小到n，也有改变默认值的功能 如果空间不足也可以扩容，主要是调整可用空间vector.size

size与capacity的区别

size表示当前vector中有多少个元素 (finish – start) , 而capacity函数则表示它已经分配的内存中可以容纳多少元素 (end_of_storage – start)

vector的底层实现要求连续的对象排列，引用并非对象，没有实际地址，因此vector的元素类型不能是引用

vector迭代器失效的情况

当插入一个元素到vector中，由于引起了内存重新分配，所以指向原内存的迭代器全部失效。

当删除容器中一个元素后,该迭代器所指向的元素已经被删除，那么也造成迭代器失效。 erase方法会返回下一个有效的迭代器，所以当我们删除某个元素时，需要it=vec.erase(it)

1) 顺序容器(比如vector、 deque) erase迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list除外)，所以不能使用erase(it++)的方式，但是erase的返回值是下一个有效迭代器； It = c.erase(it);

2) 关联容器(比如map、 set、 multimap、 multiset等) erase迭代器只是被删除元素的迭代器失效，但是返回值是void，所以要采用erase(it++)的方式删除迭代器； c.erase(it++)

插入操作

对于vector和string，如果容器内存被重新分配，iterators,pointers,references失效；如果没有重新分配，那么插入点之前的iterator有效，插入点之后的iterator仍然失效

对于list和forward_list，所有的iterator,pointer和refercnce有效。

删除操作

对于vector和string，删除点之前的iterators,pointers,references有效；

对于list和forward_list，所有的iterator,pointer和refercnce有效

对于关联容器map来说，如果某一个元素已经被删除，那么对应迭代器就失效了

vector如何释放空间

如果需要空间动态缩小，可以考虑使用deque。如果vector，可以用swap()来帮助你释放内存

```
vector().swap(Vec); //清空Vec的内存；
```

vector扩容为什么要以1.5倍或者2倍扩容？根据查阅的资料显示，考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以2倍的方式扩容，或者以1.5倍的方式扩容。以2倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间

vector vec(10,100); 创建10个元素,每个元素值为100

vec.resize(r,vector(c,0)); 二维数组初始化

reverse(vec.begin(),vec.end()) 将元素翻转

sort(vec.begin(),vec.end()); 排序，默认升序排列

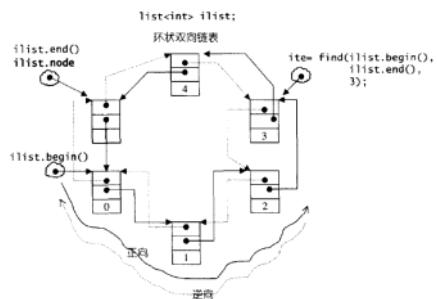
vec.push_back(val); 尾部插入数字

vec.size(); 向量大小

find(vec.begin(),vec.end(),1); 查找元素

iterator = vec.erase(iterator) 删除元素

list的底层是一个双向链表，以结点为单位存放数据，结点的地址在内存中不一定连续，每次插入或删除一个元素，就配置或释放一个元素空间。list不支持随机存取，适合需要大量的插入和删除，而不关心随即存取的应用场景



`list.push_back(elem)` 在尾部加入一个数据

`list.pop_back()` 删除尾部数据

`list.push_front(elem)` 在头部插入一个数据

`list.pop_front()` 删除头部数据

`list.size()` 返回容器中实际数据的个数

`list.sort()` 排序，默认由小到大

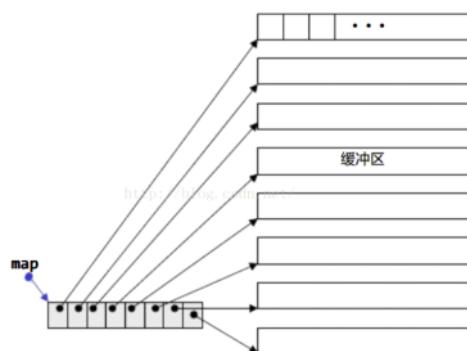
`list.remove()` 移除元素

`list.back()` 取尾部数据

`list.erase(iterator)` 删除一个元素，参数是迭代器，返回的是删除迭代器的下一个位置

`deque`是一个双向开口的连续线性空间（双端队列），在头尾两端进行元素的插入跟删除操

作都有理想的时间复杂度



`vector`可以随机存储元素（即可以通过公式直接计算出元素地址，而不需要挨个查找），但在非尾部插入删除数据时，效率很低，适合随机访问频繁的场景

选择使用`vector`而非`deque`，因为`deque`的迭代器比`vector`迭代器复杂很多

`list`不支持随机存储，适用于频繁插入和删除的场景，比如读少写多

`deque.push_back(elem)` 在尾部加入一个数据。

`deque.pop_back()` 删除尾部数据。

`deque.push_front(elem)` 在头部插入一个数据。

`deque.pop_front()` 删除头部数据。

`deque.size()` 返回容器中实际数据的个数。

`deque.at(idx)` 传回索引idx所指的数据，如果idx越界，抛出`out_of_range`。

对于STL里的`map`容器，`count`方法与`find`方法，都可以用来判断一个key是否出现，

`mp.count(key) > 0`统计的是key出现的次数，因此只能为0/1，而`mp.find(key) != mp.end()`

则表示key存在

常用操作：

`map`是顺序容器支持下标访问，`set`是关联容器

it `map.begin()` 返回指向容器起始位置的迭代器（iterator）

it `map.end()` 返回指向容器末尾位置的迭代器

`bool map.empty()` 若容器为空，则返回true，否则false

it `map.find(k)` 寻找键值为k的元素，并用返回其地址

`int map.size()` 返回map中已存在元素的数量

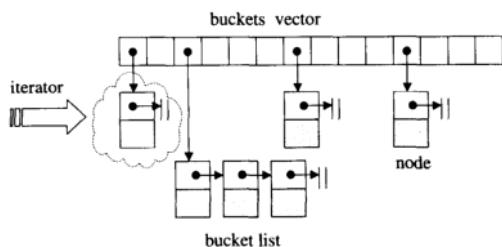
```

map.insert({int,string}) 插入元素
for (itor = map.begin(); itor != map.end() { if (itor->second == "target")
map.erase(itor++); // erase之后，令当前迭代器指向其后继。
else
++itor;
}

```

unordered_map 底层使用 hashtable+bucket 的实现原理，每个特定的 key 会通过特定的哈希运算映射到一个 hashtable 特定的位置，处理 hash 冲突的方法就是在相同 hash 值的元素位置下面挂 bucket (桶)，当数据量在 8 以内使用链表来实现桶，当数据量大于 8 则自动转换为红黑树结构 也就是有序 map 的实现结构

hashtable：可以看作是一个数组 或者 vector 之类的连续内存存储结构（可以通过下标来快速定位时间复杂度为 O(1)）。



hashtable 中的 **bucket** 维护的是自己定义的，由 **hashtable_node** 数据结构组成的链表，
bucket 聚合体本身使用 **vector** 进行存储

unordered_map 底层设计使用的是 hashtable， hashtable 槽数是根据需要分配的，但是一般都是 2 的 n 次方大小 (unordered_map 底层实现既是如此)。这种设计在计算桶号的时候有一个优势就是可以使用按位与 (&) 来加快计算

例如 $9 \% 8$ 这里 $8 = 2^3$ 的 3 次方，所以相当与把 9 的二进制 (1001) 右移 3 位，移掉的是 001，那么余数 (移掉的就是多余的数) 就是 1

也可以按照 2 的幂次对桶的容量进行扩容 源码里使用的是 allocator 中的 allocate 申请空间

遍历 oldHashMap 里面每一个不空的桶，把里面的 list 结点 splice 摘下来，放入新的哈希表 mHashMap

```

*****
*****  

#include<bits/stdc++.h>
using namespace std;

const int hashsize = 12;//哈希容量
template<typename T, typename V>
struct Hashnode{ //定义结点结构体
    T _key,
    V _value;
};

template<typename T, typename V> //使用拉链法实现哈希表类
class HashTable{
public:
    HashTable(): vec(hashsize){} //类中的vector需要构造函数指定大小链数
    ~HashTable();
    bool insert_data(const K &key);
    int hash(const T &key);
    bool hash_find(const T &key);

private:
    vector<list<HashNode<T, V>>> vec; //哈希链存放在一个vector内
}

template<typename K, typename V> //哈希函数 除留取余

```

```

int HashTable<K, V>:: hash(const T& key){
    return key%13;
}

template<typename K, typename V>
bool HashTable<K,V>:: hash_find(const T&key){
    int index = hash(key); //计算哈希值
    for(auto it = vec[index].begin(); it!= vec[index].end(); it++){
        if(it ->_key == key){ //如果找到一个已有元素
            cout<<it ->_value<<endl;
            return true;
        }
    }
    return false;
}

template<typename K, typename V>
bool HashTable<T, V>::insert_data(const T&key, const V &value){
    HashNode<T,V> node;
    node._key = key;
    node._value = value;
    for(int i = 0; i <hash.size(); i++){
        if(i == hash(key)){ //将节点加入链表
            vec[i].push_back(node);
            return true;
        }
    }
}

int main(int argc, char *argv[]){
    HashTable<int,int> ht; //测试对象
}

```

find函数用法

查找从指定位置开始的第一次出现的目标字符串：

```

#include<iostream>
#include<csdio>
using namespace std;

int main(){
    string s1 = "abcdef";
    string s2 = "de";
    int ans = s1.find(s2, 2); //从S1的第二个字符开始查找子串S2
    cout<<ans<<endl;
}

```

虽然可以使用 cin 和 >> 运算符来输入字符串，但它可能会导致一些需要注意的问题。

当 cin 读取数据时，它会传递并忽略任何前导白色空格字符（空格、制表符或换行符）。一旦它接触到第一个非空格字符即开始阅读，当它读取到下一个空白字符时，它将停止读取。以下面的语句为例：

```
1 | cin >> name;
```

可以输入 "Mark" 或 "Twain"，但不能输入 "Mark Twain"，因为 cin 不能输入包含嵌入空格的字符串。下面程序演示了这个问题：

```
1 | // This program illustrates a problem that can occur if
2 | // cin is used to read character data into a string object.
3 | #include <iostream>
4 | #include <string> // Header file needed to use string objects
5 | using namespace std;
6 | int main()
7 | {
8 |     string name;
9 |     string city;
10 |    cout << "Please enter your name: ";
11 |    cin >> name;
12 |    cout << "Enter the city you live in: ";
13 |    cin >> city;
14 |    cout << "Hello, " << name << endl;
15 |    cout << "You live in " << city << endl;
16 |
17 | }
```

程序输出结果：

```
Please enter your name : John Doe
```

```
Enter the city you live in: Hello, John
```

```
You live in Doe
```

请注意，在这个示例中，用户根本没有机会输入 city 城市名。因为在第一个输入语句中，当 cin 读取到 John 和 Doe 之间的空格时，它就会停止阅读，只存储 John 作为 name 的值。在第二个输入语句中，cin 使用键盘缓冲区中找到的剩余字符，并存储 Doe 作为 city 的值。

为了解决这个问题，可以使用一个叫做 getline 的 C++ 函数。此函数可读取整行，包括前导和嵌入的空格，并将其

为了解决这个问题，可以使用一个叫做 getline 的 C++ 函数。此函数可读取整行，包括前导和嵌入的空格，并将其存储在字符串对象中。

getline 函数如下所示：

```
1 | getline(cin, inputLine);
```

其中 cin 是正在读取的输入流，而 inputLine 是接收输入字符串的 string 变量的名称。下面的程序演示了 getline 函数的应用：

```
1 | // This program illustrates using the getline function
2 | //to read character data into a string object.
3 | #include <iostream>
4 | #include <string> // Header file needed to use string objects
5 | using namespace std;
6 | int main()
7 | {
8 |     string name;
9 |     string city;
10 |    cout << "Please enter your name: ";
11 |    getline(cin, name);
12 |    cout << "Enter the city you live in: ";
13 |    getline(cin, city);
14 |    cout << "Hello, " << name << endl;
15 |    cout << "You live in " << city << endl;
16 |
17 | }
```

程序输出结果：

```
Please enter your name: John Doe
```

```
Enter the city you live in: Chicago
```

```
Hello, John Doe
```

```
You live in Chicago
```

4.2 stringstream用于空格分割的字符串的切分

`stringstream` 主要是用在將一個字串分割，可以先用 `clear()` 以及 `str()` 將指定字串設定成一开始的內容，再用 `>>` 把個別的資料輸出，例如：

```
string s;
stringstream ss;
int a, b, c;
getline(cin, s);
ss.clear();
ss.str(s);
ss >> a >> b >> c;
```

常用STL

2020年3月9日 15:34

```
int temp = 2, sum = 0;
while(n--);
{
    sum +=temp;
    temp +=3;
}
cout << sum << endl;
```

while循环加 ; ? ? ?

if()内用赋值符号写 ? ? ? ?

error: expression is not assignable

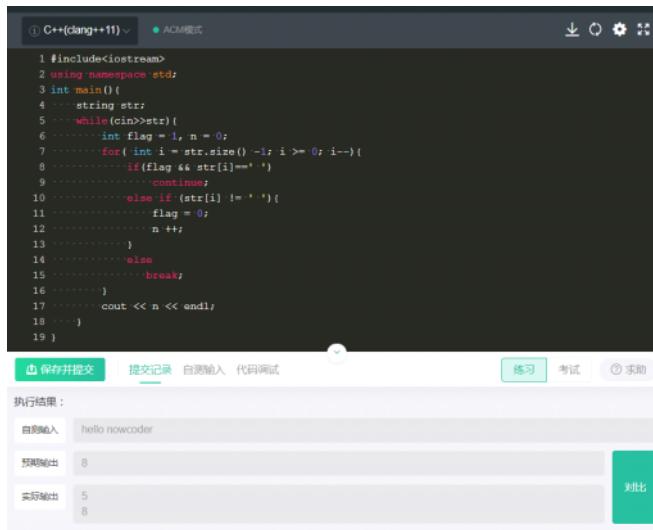
while(cin >> s) 连续读取两行内容作为一行

while(getline(cin,s))分两行读取 空格也会读入

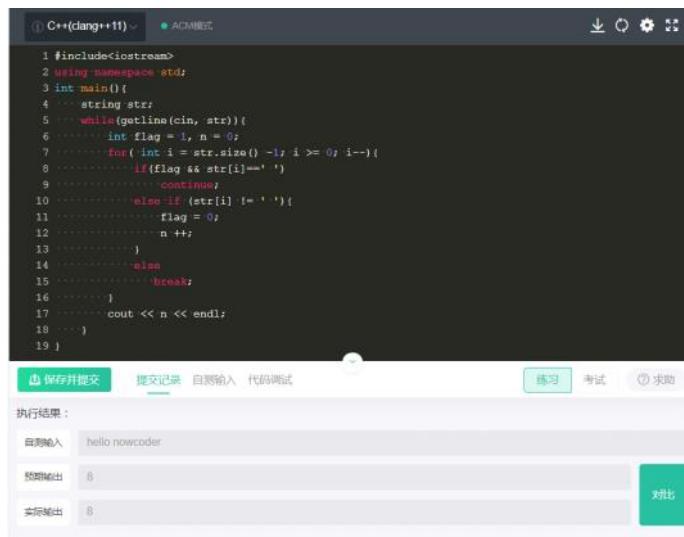
在while输入前面定义的变量使用//不使用cin读入的变量

写错变量名 ? ? ?

语句结尾不加 ;



```
#include<iostream>
using namespace std;
int main(){
    string str;
    while(cin>>str){
        int flag = 1, n = 0;
        for( int i = str.size() -1; i >= 0; i--){
            if(flag && str[i]== ' '){
                continue;
            } else if (str[i] != ' '){
                flag = 0;
                n++;
            }
        }
        cout << n << endl;
    }
}
```



```
#include<iostream>
using namespace std;
int main(){
    string str;
    while(getline(cin, str)){
        int flag = 1, n = 0;
        for( int i = str.size() -1; i >= 0; i--){
            if(flag && str[i]== ' '){
                continue;
            } else if (str[i] != ' '){
                flag = 0;
                n++;
            }
        }
        cout << n << endl;
    }
}
```

map中find函数和count函数

原创 wangxin0205 2020-10-04 23:23:11 161 收藏 2

分类专栏： C++学习 文章标签： c++ 数据结构

版权

find()

在map中查找关键字 (key) 为 k 的元素，返回指向它的迭代器。若k不存在，返回 map::end.

返回值是一个迭代器，成功返回迭代器指向要查找的元素，失败返回的迭代器指向end

count()

统计map中关键字 (key) 为 k 的元素的个数，对于 map，返回值不是 1 (存在)，就是 0 (不存在)

返回值是一个整数，1 表示有这个元素，0 表示没有这个元素。只会返回这两个数中的 1 个。可以用于判断某值是否存在。

while(getline(cin,line))语句

注意这里默认回车符停止读入,按Ctrl+Z或键入EOF回车即可退出循环。

在这个语句中，首先getline从标准输入设备上读入字符，然后返回给输入流cin，注意了，是cin，所以while判断语句的真实判断对象是cin，也就是判断当前是否存在有效的输入流。在这种情况下，我想只要你的电脑不中毒不发神经你的输入流怎么会没有效？所以这种情况下不管你如何输入都跳不出循环，因为你的输入流有效，跳不出循环。

然而有些同学误以为while判断语句的判断对象是line（也就是line是否为空），然后想通过直接回车（即输入一个空的line）跳出循环，却发现怎么也跳不出循环。这是因为你的回车只会终止getline()函数的读入操作。getline()函数终止后又进行while () 判断（即判断输入流是否有效，你的输入流当然有效，满足条件），所以又运行getline()函数。

所以，以下的写法根本不可能让你推出while () 循环的：

```
while(getline(cin,line))
```

```
cout<<line<<endl;
```

C++ 中substr函数有三种用法，如下所示：

```
假设string s("student12");
string x=s.substr()          //默认时的长度为从开始位置到尾
string y=s.substr(5)          //获得字符串s中 从第5位开始到尾的字符串
string z=s.substr(5,3);       //获得字符串s中 从第5位开始的长度为3的字符串
```

stringstream的常见用法

以stringstream数据流保存数据，然后用getline()间隔读取string类数据，达到分割string字符串的目的，并利用str.data()或者str.c_str()以及atoi()转化为int。

```
1 #include <sstream>
2 #include <string>
3 #include <iostream>
4 #include <cstdio>
5 #include <vector>
6 using namespace std;
7 int main()
8 {
9     vector<int> res;
10    string str= "1.2.3.4.5";
11    stringstream ss(str);
12    string sub_str;
13    while(getline(ss,str,'.'))//以.为间隔分割内容
14    {
15        const char *p=str.data();//string类型为const char *,所以转化类型必须匹配
16        //const char *p=str.c_str();
17        res.push_back(atoi(p));
18    }
19    for(auto x:res){
20        cout<<x<<endl;
21    }
22    return 0;
23 }
```

1.find

说明：查找字符串str1首字符在另一个字符串str中出现的位置，但是str1必须为其子字符串

```
1 str1.find(str2);           // 从串str1中查找时str2，返回str2中首个字符在str1中的地址
2 str1.find(str2,5);         // 从str1的第5个字符开始查找str2
3 str1.find("usage");       // 如果usage在str1中查找到，返回u在str1中的位置
4 str1.find("o");           // 查找字符o并返回地址
5 str1.find("of big",2,2);   // 从str1中的第二个字符开始查找of big的前两个字符
```

2.find_first_of

用法：str.find_first_of(str1 , pos)

说明：从pos位置开始查找str1，从前往后，只要查到str1中的任何一个字符有则返回其在str中的索引值

3.find_last_of

用法：str.find_last_of(str1,pos)

说明：从pos位置开始查找，从后往前，查到str1中的任何一个字符则返回其在str中的索引值

例题：leetcode 345题：反转字符串中的元音字母

输入：hello
返回：holle

```
1 class solution {
2 public:
3     string reverseVowels(string s) {
4         int left=0,right=s.size()-1;
5         while(left<right)
6         {
7             left=s.find_first_of("aeiouAEIOU",left);
8             right=s.find_last_of("aeiouAEIOU",right);
9             if(left<right)
10            {
11                swap(s[left++],s[right--]);
12            }
13        }
14        return s;
15    }
16};
```

```
46 ..... //int n, invalid = 0;
47 ..... int n = 0; int invalid = 0;
48 ..... for(int i = 1; i < s1.size(); i++){
49 .....     if(s1[i]>='0' && s1[i] <='9')
50 .....         n = n*10 + (s1[i] - '0');
```

这个n是随机初始的，并不是与invalid都设为0；必须分开独立进行初始化处理！！！

```
... vector<int> res(n, 0);
... for(int i = 0; i <n; ++i)
...     cin >> res[i];
```

这里用到了下标访问，如果不进行初始化就会出现段错误，一般段错误都是因为出现下标非法访问或者循环无限

```
vector<vector<int> > maze;
vector<pair<int,int> > path_temp;
vector<pair<int,int> > path_best;
void MazeTrack(int i, int j){
    maze[i][j] = 1;
    path_temp.push_back(make_pair(i,j));

    maze = vector<vector<int> >(N, vector<int>(M, 0));
    path_temp.clear();
    path_best.clear();
    for (auto &i : maze) ... //因为这是一个全局变量
    ... //for(auto &j : i) ... //不加这个引用就错了
    for (auto &j : i)
        cin >> j;
    ...
```

这里由于maze是一个全局变量，使用引用改变全局变量的内容，void函数也用到了maze

c++的几个内置函数

islower(char c) 是否为小写字母
isupper(char c) 是否为大写字母
isdigit(char c) 是否为数字
isalpha(char c) 是否为字母
isalnum(char c) 是否为字母或者数字
toupper(char c) 字母小转大
tolower(char c) 字母大转小

```
1 | #include <iostream>
2 | #include <string>
3 |
4 | using namespace std;
```

基础知识

2020年2月1日 9:06

题目描述

计算字符串最后一个单词的长度，单词以空格隔开。

输入描述:

输入一行，代表要计算的字符串，非空，长度小于5000。

输出描述:

输出一个整数，表示输入字符串最后一个单词的长度。

示例1

输入
hello nowcoder

flag 作为开始查找到单词的标识
再遇到空格时可以借助标识与空格判断 找到了一个单词

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 int main(){
4     string s;
5     while(getline(cin,s)){
6         int flag = 1; int n = 0;
7         for(int i = s.size()-1; i >= 0; i--){
8             if(flag && s[i] == ' ')
9                 continue;
10            else if(s[i] != ' '){
11                flag = 0;
12                n++;
13            }
14            else if(flag == 0 && s[i] == ' ')
15                break;
16        }
17        cout << n << endl;
18    }
19 }
20 }
```

从string后面开始向前遍历

添加一个标置位 表示单词开始

可以用s.size() 加s[i]表示元素

题目描述

写出一个程序，接受一个由字母、数字和空格组成的字符串，和一个字母，然后输出输入字符串中该字母的出现次数。不区分大小写。

输入描述:

第一行输入一个由字母和数字以及空格组成的字符串，第二行输入一个字母。

输出描述:

输出输入字符串中含有该字符的个数。

示例1

输入
ABCabc
A

如果要将字符转成大写字符使用 toupper(char c)

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 int main(){
4     string s;
5     char c;
6     int n = 0;
7     getline(cin,s);
8     cin >> c;
9     for(auto i = s.begin(); i < s.end(); i++){
10        if(toupper(*i) == toupper(c))
11            n++;
12    }
13    cout << n << endl;
14 }
15 return 0;
16 }
```

这里题目明确说明了只有一行输入 就不能用while(getline(cin,s))
for循环string可以用迭代器遍历 auto i = s.begin(), i < s.end() , i++
加 *i作为元素访问

题目描述

明明想在学校中请一些同学一起做一项问卷调查，为了实验的客观性，他先用计算机生成了N个1到1000之间的随机整数(N<1000)，对于其中重复的数字，只保留一个，把其余相同的数去掉，不同的数对应着不同的学生的学号。然后再把这些数从小到大排序，按照排好的顺序去找同学做调查。请你协助明明完成“去重”与“排序”的工作(同一个测试用例里可能会有多组数据(用于不同的调查)，希望大家能正确处理)。

注：测试用例保证输入参数的正确性，答题者无需验证。测试用例不止一组。
当没有新的输入时，说明输入结束。

输入描述：

注意：输入可能有多组数据(用于不同的调查)。每组数据都包括多行，第一行先输入随机整数的个数N，接下来的N行再输入相应个数的整数。具体格式请看下面的“示例”。

输出描述：

返回多行，处理后的结果。

往set中插入一个元素 用st.insert(x)

如果要遍历这个set 用迭代器遍历 for(auto it = st.begin(), it != st.end(), it++)

*it就表示set中的每一个元素 利用set可以自动去重、字典序排序

st.end()表示最后一个xx空符

```
1 //机试排序可以用vector::sort函数
2 //利用set容器可以自动去重·排序
3 #include <iostream>
4 #include <set>
5 using namespace std;
6 int main(){
7     int N, n;
8     set<int> ss;
9     while(cin >> N){
10         ss.clear();
11         while(N--){
12             cin >> n;
13             ss.insert(n);
14         }
15         for(auto it = ss.begin(); it != ss.end(); it++)
16             cout << *it << endl;
17     }
18     return 0;
19 }
```

对于这种先输入一个N 再输入N行 的输入类型 可以先用while(cin>>N){}里面再接一个while(N--)(cin>>n)

关联窗口的操作ss.clear(); ss.insert(n) 如果是定义的 set<int> ss; insert操作会自动按顺序进行插入且不会插入重复元素

关联容器的迭代器不支持<运算符，只可以用it != ss.end()作为迭代终止表示

题目描述

·连续输入字符串，请按长度为8拆分每个字符串后输出到新的字符串数组；
·长度不是8整数倍的字符串在后面补数字0，空字符串不处理。

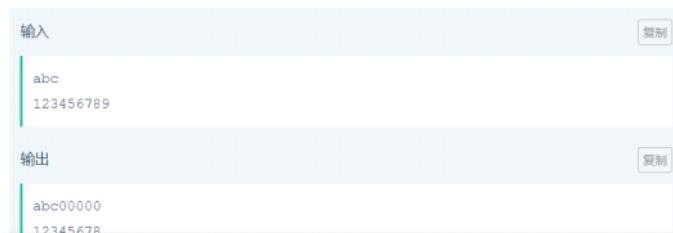
输入描述：

连续输入字符串(输入多次，每个字符串长度小于100)

输出描述：

输出到长度为8的新字符串数组

示例1



s=s.substr(0, 8) 返回从0下标开始的8个字符构成的子串

s=s.substr(8) 只有一个参数 返回从下标8开始的子串

s= s.append(n, '0'); 返回一个串 在str的尾部append n个 '0' 字符

```

1 #include<bits/stdc++.h>
2 using namespace std;
3 int main(){
4     string s;
5     while(getline(cin, s)){
6         while(s.size() > 8){
7             cout << s.substr(0, 8) << endl;
8             s = s.substr(8);
9         }
10        s = s.append(8-s.size(), '0');
11        cout << s << endl;
12    }
13    return 0;
14}
15 }
```

while与if的区别 while可以进行多次判断到条件结束，

而if只会判断一次，剩下的就不管它了

string类型的substr函数的使用：

s = s.substr(初始位置 可以为0, 从初始出发n位) 返回一个子串

s.substr(n) 从初始位置n出发一直到字符串结束 返回一个子串

s.append(多少个, '字符') 字符串末尾添加n个字符

题目描述

写出一个程序，接受一个十六进制的数，输出该数值的十进制表示。

输入描述:

输入一个十六进制的数值字符串。注意：一个用例会同时有多组输入数据，请参考帖子
<https://www.nowcoder.com/discuss/276>处理多组输入的问题。

输出描述:

输出该数值的十进制字符串。不同组的测试用例用\n隔开。

```

1 #include<bits/stdc++.h>
2 using namespace std;
3 int main(){
4     int n;
5     while(cin >> hex >> n){
6         cout << n << endl;
7     }
8     return 0;
9 }
```

cin>>hex>>n 接收一个16进制的输入，用n保存再将n输出到cout

cin >> hex >> n; 直接cout << n << endl 默认以十进制的形式输出

题目描述

功能:输入一个正整数，按照从小到大的顺序输出它的所有质数因子(重复的也要列举)(如180的质因子为2 2 3 3 5)
 最后一个数后面也要有空格

输入描述:

输入一个long型整数

输出描述:

按照从小到大的顺序输出它的所有质数的因子，以空格隔开。最后一个数后面也要有空格。

示例1



for 除数a 最多从2到 sqrt(x) , 找到x的平方根

```

while(x % a == 0){
    cout << a << ' ';
    x /= a;
}
```

执行完**for** , 如果x > 1

```
    cout << x << endl;
```

```

1 #include<iostream>
2 #include<cmath>
3 using namespace std;
4 int main(){
5     long input;
6     while(cin >> input){
7         for(int a = 2; a <= sqrt(input); a++){
8             while(input % a == 0){
9                 cout << a << " ";
10                input = input / a;
11            }
12        }
13        if(input > 1)
14            cout << input << " ";
15    }
16    return 0;
17 }

```

首先要定义一个除数 用来作取余，找质因子 因此除数从2开始到 n平方根结束
 每次取余 再除再取余 这里要用while循环 一直到 多个重复质因子全部除完
 请注意结尾的那个数如果大于1就是最后一个没有取余的质因子
 注意以空格隔开 往cout << a << ' ' 一个空格，并不结束当前输出 endl是结束输出与换行

题目描述

数据表记录包含索引和数值（Int范围的正整数），请对索引相同的记录进行合并，即将相同索引的数值进行求和运算，输出按照key值升序进行输出。

输入描述:

先输入键值对的个数
 然后输入成对的Index和value值，以空格隔开

输出描述:

输出合并后的键值对（多行）

示例1

输入	输出
<pre> 4 0 1 0 2 1 2 3 4 </pre>	<pre> 0 3 1 2 </pre>

mp[i] 如果mp[i]没有 则创建一个i元素 值为默认值 int为0

if(mp.count(key) != 0) 输出只有0 or 1

if(mp.find(key) != mp.end()) auto it 如果不是end()迭代器，则说明找到了一个key

map的元素是pair类型 遍历map

for (auto x : map) x中一个

```

1 #include<bits/stdc++.h>
2 using namespace std;
3 int main(){
4     int n;
5     while(cin >> n){
6         map<int,int> mp;
7         while(n--){
8             int templ, temp2;
9             cin >> templ >> temp2;
10            //if(mp.find(templ) != mp.end())
11            if(mp.count(templ) != 0)
12                mp[templ] += temp2;
13            else
14                mp[templ] = temp2;
15        }
16        for (auto x : mp)
17            cout << x.first << " " << x.second << endl;
18    }
19    return 0;
20 }

```

定义一个map : map<int,int> mp

mp[key] += temp; mp[key2] = temp; 这是mp元素操作

mp.count(key) != 0; 统计map中的key关键字是否有

mp.find(key) != mp.end() 查找mp中的key关键字是否有

mp元素是pair类型，可以用x.first, x.second访问key 与 value

for(auto it = m.begin(); it != m.end(); it++) //请注意关联容器不支持< , vector容器可以用it<mp.end()
 cout << it->first << ' ' << it->second << endl;

for(auto it = memo.begin(); it != memo.end(); it++){

```
cout << (*it).first << ' ' << it->second << endl;
}
```

输入一个int型整数，按照从右向左的阅读顺序，返回一个不含重复数字的新的整数。
保证输入的整数最后一位不是0。

输入描述:

输入一个int型整数

输出描述:

按照从右向左的阅读顺序，返回一个不含重复数字的新的整数

示例1



```
输入  
9876673  
输出  
37689
```

```
#include<iostream>
#include<vector>
using namespace std;
int main(){
    int m;
    int n;
    vector<int> hs(10, 0);
    while(cin>> m) {
        while(m>0) {
            n = m%10;
            m = m/10;
            if(hs[n] == 0) {
                hs[n] += 1;
                cout<< n;
            }
        }
        return 0;
    }
}
```

用取余再除10的方法依次取最右边的位数 用一个vector保存 0-10已经出现的数字 初始为0 若有加1

```
int main() {
    int n;
    while(cin >> n) {
        string str = to_string(n);
        reverse(str.begin(), str.end());
        unordered_set<char> st;
        string res;
        for(auto x : str){
            if(st.count(x) == 0) {
                st.insert(x);
                res += x;
            }
        }
        cout << stoi(res) << endl;
    }
}
```

先将输入整数采用 to_string(nums)转为 string类型

再使用 string的 reverse(str.begin(), str.end()) 翻转string

注意使用一个哈希表 unordered_set<char> set; 保存 str中已经出现的字符 ,若没有则使用insert(x)

插入字符，并将该字符拼接到空字符串res res+= x;

最终得到的是一个字符串 res 输出要求是一个整数 使用 stoi(res) 得到一个整数

```
...//string s = reverse(str.begin(), str.end());
...reverse(str.begin(), str.end());
```

string 的reverse函数是直接对原string进行操作 不要写出这种返回代码 返回值不是string！！直接在原string 操作就行了

题目描述

编写一个函数，计算字符串中含有的不同字符的个数。字符在ASCII码范围内(0~127)，换行表示结束符，不算在字符串里。不在范围内的不作统计。多个相同的字符只计算一次。
例如，对于字符串abaca而言，有a、b、c三种不同的字符，因此输出3。

输入描述:

输入一行没有空格的字符串。

输出描述:

输出范围在(0~127)字符的个数。

```
int main() {
    string str;
    while(cin >> str) {
        set<char> st;
        for(auto x : str) {
            if(x >= 0 && x <= 127)
                st.insert(x);
        }
        cout << st.size() << endl;
}
```

使用st.size()可以返回set里面元素的个数，st.insert(x)对于已有的元素insert操作不会执行，set保证每个key的count都最多为1

题目描述

将一个英文语句以单词为单位逆序排放。例如“I am a boy”，逆序排放后为“boy a am I”
所有单词之间用一个空格隔开，语句中除了英文字母外，不再包含其他字符

输入描述:

输入一个英文语句，每个单词用空格隔开。保证输入只包含空格和字母。

输出描述:

得到逆序的句子

示例1

输入	<input type="text" value="I am a boy"/>	复制
输出	<input type="text" value="boy a am I"/>	复制

sort函数可以对vector<string>内元素按字典序进行排序

```
int main() {
    string str;
    vector<string> vec;
    while(cin >> str) {
        vec.push_back(str);
    }
    for(int i = vec.size() - 1; i >= 0; i--) {
        cout << vec[i] << " ";
    }
    cout << vec[0] << endl;
    return 0;
}
```

利用while(cin >> str) cin遇到空格结束读取，每次while循环读取一个string/单词
用一个vector存取 定义vector<string> vec; 使用vec.push_back(str)将每次读取的单词存入vec
从vec的尾部开始遍历输出单词 注意每输出一个vec[i] 需要接一个<<''>>；结尾的vec[0]单独输出再接一个<<endl>>；

题目描述

给定n个字符串，请对n个字符串按照字典序排列。

输入描述:

输入第一行为一个正整数n(1≤n≤1000)，下面n行为n个字符串(字符串长度≤100)，字符串中只含有大小写字母。
注意用来累加的变量res一定要记得初始为0
要不然这个累加变量的值会随机初始很大

输出描述:

数据输出n行，输出结果为按照字典序排列的字符串

```
int main() {
    int n; string str;
    while(cin >> n) {
        ...while(n > 0) {
            ...res += (n%2);
            ...n /= 2;
        }
        ...cout << res;
    }
}
```

```

int main() {
    int n; string str;
    while(cin >> n) {
        vector<string> vec;
        while(n--) {
            cin >> str;
            vec.push_back(str);
        }
        sort(vec.begin(), vec.end());
        for(auto x : vec)
            cout << x << endl;
    }
    return 0;
}

```

这里用到了vector类型的sort函数，将输入的string存放到一个vector内，vector的元素是string类型，再利用vector容器的sort函数 sort(vec.begin(), vec.end())对vector容器内的元素进行排序

题目描述

输入一个int型的正整数，计算出该int型数据在内存中存储时1的个数。

输入描述:

输入一个整数（int类型）

输出描述:

这个数转换成2进制后，输出1的个数

```

int main() {
    int n; int res = 0;
    while(cin >> n) {
        while(n != 0) {
            //while(n > 0) {
            if(n & 1 == 1)
                res += 1;
            n >>= 1;
        }
        cout << res << endl;
    }
}

```

这里请注意是 $n >>= 1$ 不是 $n >> 1$!!! 即 $n = n >> 1$; 位运算更高级

这个从ss流中读取串t，由于下面用到了下标操作
需要判断t是否为空，否则会出现访问out of range错误
加一个if(t.empty()) continue; 这组；不要，继续读取下一个
以；分隔的串t

题目描述

开发一个坐标计算工具，A表示向左移动，D表示向右移动，W表示向上移动，S表示向下移动。从(0,0)点开始移动，从输入字符串里面读取一些坐标，并将最终输入结果输出到输出文件里面。

输入：

合法坐标为A(或者D或者W或者S) + 数字 (两位以内)

坐标之间以;分隔。

非法坐标点需要进行丢弃。如AA10; A1A; \$%\$; YAD; 等。

下面是一个简单的例子 如：

A10;S20;W10;D30;X;A1A;B10A11;;A10;

处理过程：

起点(0,0)

+ A10 = (-10,0)

+ S20 = (-10,-20)

+ W10 = (-10,-10)

+ D30 = (20,-10)

+ x = 无效

+ A1A = 无效

+ B10A11 = 无效

+ 一个空不影响

+ A10 = (10,-10)

结果(10, -10)

```

1
int main() {
    string s, t;
    while(getline(cin, s)) {
        stringstream ss(s);
        pair<int,int> p(0,0);
        while(getline(ss,t,';')) {
            if(t.empty())
                continue;

```

如果用while(getline(cin,s,';')) 会出现每次读取一行，每行只有一个；坐标

要先将这一整行读取进来，当作一个输入，再将这个string转换为stringstream，利用；将其分隔，

每次处理一个string

stringstream ss(s) 定义一个stringstream类型 ss 用s进行初始化

pair<int,int> p(0,0) 定义一个pair类型 p 表示坐标 左右上下 数值

```

    string str = t.substr(1);
    if(regex_match(str, regex("[0-9]*"))){
        switch(t[0]){
            case 'A': p.first -= stoi(str); break;
            case 'D': p.first += stoi(str); break;
            case 'W': p.second += stoi(str); break;
            case 'S': p.second -= stoi(str); break;
            default: break;
        }
    }
}
cout <<p.first << ',' <<p.second<<endl;

```

每一次得到t，将str定位到t.substr(1)，如A10使用if (regex_match(str, regex(" [0-9]*")))表示匹配成功
再根据t[0]字母方向对pair元素值进行修改

这种方法主要利用了stringstream 对整个输入中的每个 A10;单元进行分隔处理 逐一处理 再累加

第二种方法 利用string 的auto found = find_first_of(')'

```

int main(){
    string str;
    while(getline(cin,str)){
        pair<int,int> point(0,0);
        auto found = str.find_first_of(';');
        int start = 0;
        while(found != string::npos){
            string s1 = str.substr(start,found - start);
            start = found + 1;
            found = str.find_first_of(';',found + 1);
            if(s1.size()>1 && s1.size()<= 3){
                char c = s1[0];
                int n = 0; int invalid = 0;
                for(int i = 1; i<s1.size(); i++){
                    //if(s1[i]>=0 && s1[i]<=9)
                    if(s1[i]>='0' && s1[i]<='9')
                        n = n*10 + (s1[i] - '0');
                    else{
                        invalid = 1;
                        break;
                    }
                }
            }
        }
    }
}

```

注意s1[i] >= '0' 与 s1[i] >= 0 是两个不同的概念，看要s[i]的类型采用哪一个，如果这里比较运算符内 不加'0' 就错了 n*10+(s[i]- '0') 会将string的char类型转为 int型

invalid = 1 与 Invalid == 1 更是两个不同的概念！！！

find_first_of(';')返回找到的position 若未找到 返回一个 string::npos while(f != string::npos) 一直遍历查找
found=find_first_of(';', found+1) 从 found+1 位置处开始查找 未找到符合的就break，进行下一次查找
注意stoi/atoi函数的使用！！！ 但是你要判断每个字符的有效性就必须遍历每个字符了

for循环必须是知道遍历的终点，且取决于遍历的步长 while循环的条件直接跨度可以更大

```

    if(invalid == 0)
        switch(c){
            case 'A': point.first -= n; break;
            case 'D': point.first += n; break;
            case 'W': point.second += n; break;
            case 'S': point.second -= n; break;
        }
    }
    cout <<point.first<<',' <<point.second<<endl;
}
return 0;

```

注意switch 的语法 switch (n) {case 'A' :; break;

} 请注意字符 A表示 要加 ''；还有就是每一个case注意添加break语句！！！

题目描述

密码要求：

1. 长度超过8位
2. 包括大小写字母、数字、其它符号，以上四种至少三种
3. 不能有相同长度大于2的子串重复

输入描述：

一组或多组长度超过2的字符串。每组占一行

输出描述：

如果符合要求输出：OK，否则输出NG

```
bool kind_of(string str){  
    int flag[4] = {0};  
    for(auto i : str){  
        if(i >= 'A' && i <= 'Z')  
            flag[0] = 1;  
        else if(i >= 'a' && i <= 'z')  
            flag[1] = 1;  
        else if(i >= '0' && i <= '9')  
            flag[2] = 1;  
        else  
            flag[3] = 1;  
    }  
    int res = 0;  
    res = flag[0] + flag[1] + flag[2] + flag[3];  
    if(res >= 3)  
        return true;  
    else  
        return false;  
}
```

定义第一个bool判断函数

```
bool is_three(string str){  
    string extr;  
    for(int i = 0; i < str.size() - 3; i++){  
        extr = str.substr(i, 3);  
        if(str.find(extr, i + 3) == -1)  
            continue;  
        else  
            return false;  
    }  
    return true;  
}
```

```
int main(){  
    string s;  
    while(getline(cin, s)){  
        if(s.size() > 8 && kind_of(s) && is_three(s))  
            cout << "OK" << endl;  
        else  
            cout << "NG" << endl;  
    }  
    return 0;  
}
```

定义第二个bool判断函数 有无长度大于2的重复子串

如果str.find(extr, i + 3) == -1，说明当前位置的子串没有重复，continue；循环继续，不能就此return true！当然若找到了一个 != -1 的子串 可以直接return false

只有执行完整个循环才可以确定整个串中没有长度大于3的重复子串 这时才可以 return true

cout输出注意使用字符串的形式进行输出

题目描述

密码是我们生活中非常重要的东东，我们的那么一点不能说的秘密就全靠它了。哇哈哈。接下来渊子要在密码之上再加一套密码，虽然简单但也安全。

假设渊子原来一个BBS上的密码为zvbo9441987，为了方便记忆，他通过一种算法把这个密码变换成为YUANzhi1987，这个密码是他的名字和出生年份，怎么忘都忘不了，而且可以明目张胆地放在显眼的地方而不被别人知道真正的密码。

他是这么变换的，大家都知道手机上的字母：1--1，abc--2，def--3，ghi--4，jkl--5，mno--6，pqrs--7，tuv--8 wxyz--9，0--0，就这么简单，渊子把密码中出现的小写字母都变成对应的数字，数字和其他的符号都不做变换。

声明：密码中没有空格，而密码中出现的大写字母则变成小写之后往后移一位，如：X，先变成小写，再往后移一位，不就是y了嘛，简单吧。记住，z往后移是a哦。

```
int main(){
```

声明：密码中没有空格，而密码中出现的大写字母则变成小写之后往后移一位，如：X，先变成小写，再往后移一位，不就是y了嘛，简单吧。记住，z往后移是a哦。

```
int main(){
    string str;
    int id[26] = {2,2,2,3,3,3,4,4,4,5,5,5,6,6,6,7,7,7,7,8,8,8,9,9,9,9};
    while(cin >> str){
        for(auto i : str){
            if(i >= 'A' && i < 'Z'){
                //cout<<(char)(i-'A'+i+'a'-1);
                //cout<<char(tolower(i)+1);
                cout<<static_cast<char>(tolower(i) + 1);
            } else if(i == 'z'){
                cout << 'a';
            } else if(i >= 'a' && i <='z'){
                cout << id[i - 'a'];
            } else{
                cout << i;
            }
        }
        cout << endl;
    }
}
```

```
int main(){
    string str;
    while(cin >> str){
        map<char,int> mp;
        for(auto i : str){
            mp[i]++;
        }
        int min = 20;
        for(auto x : mp){
            if(x.second <= min)
                min = x.second;
        }
        for(auto i : str)
            if(mp[i] > min)
                cout << i;
        cout << endl;
    }
}
```

注意输出是字符，需要进行类型转换 上面的三种类型转换方法都可以使用

题目描述

实现删除字符串中出现次数最少的字符，若多个字符出现次数一样，则都删除。输出删除这些单词后的字符串，字符串中其它字符保持原来的顺序。

注意每个输入文件有多组输入，即多个字符串用回车隔开

输入描述:

字符串只包含小写英文字母，不考虑非法输入，输入的字符串长度小于等于20个字节。

输出描述:

删除字符串中出现次数最少的字符后的字符串。

示例1

```
int main(){
    int i, m, min;
    string str;
    while(cin >> str){
        int a[26] = {0};
        for(int i = 0; i < str.size(); i++)
            a[str[i] - 'a']++;
        min = a[str[0] - 'a'];
        for(int i = 0; i < str.size(); i++)
            if(a[str[i] - 'a'] <= min)
                min = a[str[i] - 'a'];
        for(int i = 0; i < str.size(); i++)
            if(a[str[i] - 'a'] > min)
                cout << str[i];
        cout << endl;
    }
    return 0;
}
```

用一个a[26]存放str中每个元素对应的出现次数，注意这个a[26]在每次while循环时都需要重新初始化为0，再进行下面的下标计数，作为统计不同str输入的字符次数

用一个map也可以作为统计实现

题目描述

编写一个程序，将输入字符串中的字符按如下规则排序。

规则 1：英文字母从 A 到 Z 排列，不区分大小写。

如，输入：Type 输出：epTy

规则 2：同一个英文字母的大小写同时存在时，按照输入顺序排列。

如，输入：BabA 输出：aABb

规则 3：非英文字母的其它字符保持原来的位置。

如，输入：By?e 输出：Be?y

注意有多组测试数据，即输入有多行，每一行单独处理（换行符隔开的表示不同行）

vector sort函数会默认对pair的first升序
如果first相等对second升序

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    string str;
    while(getline(cin,str)){
        vector<pair<char,int>> res;
        for(int i = 0; i < str.size(); i++){
            if((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A' && str[i] <= 'Z')){
                pair<char,int> tmp;
                tmp.first = tolower(str[i]);
                tmp.second = i;
                res.push_back(tmp);
            }
        }
        sort(res.begin(),res.end());
        int j = 0;
        for(int i = 0; i < str.size(); i++){
            if((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A' && str[i] <= 'Z')){
                cout << str[res[j].second];
                j++;
            }
        }
        cout << endl;
    }
}
```

```
#include<bits/stdc++.h>
using namespace std;
bool isLetter(char c) {
    if((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
        return true;
    else
        return false;
}
```

先写一个bool判断函数 判断是否为英文字符

```
void bubble_sort(string & str) {
    for(int k = str.size() - 1; k >= 0; k--) {
        int i = 0, j = 0;
        while(j <= k) {
            if(isLetter(str[j])) {
                if(tolower(str[i]) > tolower(str[j]))
                    swap(str[i], str[j]);
                i = j;
            }
            j++;
        }
    }
}
```

对英文字母进行冒泡排序，注意这里统一转成了小写字符进行排序，

冒泡排序 首先定义一个尾索引，再定义两个首索引 依次两两元素间进行比较 遇到其他字符不比较 直接j++

到下一个字符再与i进行比较 注意每一次比较完都会更新i = j， 再将j++，以实现每次都是相邻的两个元素进行比较

```
int main() {
    string str;
    while(getline(cin, str)) {
        bubble_sort(str);
        cout << str << endl;
    }
    return 0;
}
```

这个函数的参数是引用 string &str，main函数直接传入参数，该函数就可以更改传入的str，再直接到cout

查表法

```
if(isalpha(str[i]) || isdigit(str[i]))
    res += helper2[helper1.find(str[i])]
```

题目描述

1、对输入的字符串进行加解密，并输出。

2、加密方法：

当内容是英文字母时则用该英文字母的后一个字母替换，同时字母变换大小写，如字母a时

则替换为B；字母Z时则替换为a；

当内容是数字时则把该数字加1，如0替换1，1替换2，9替换0；

其他字符不做变化。

3、解密方法为加密的逆过程。

如果a[i] 是一个字符 那么 a[i] += 33 a[i] -= 31；都表示一个字符

如果是 a[i] - 'a' 字符相减 那得到的是一个整数 可以进行static_cast<char>(int) 转成字符static_cast(tolower(i) + 1) 转成字符

输入描述:

输入说明

输入一串要加密的密码

输入一串加密后的密码

输出描述:

输出说明

输出加密后的字符

输出解密后的字符

```
const string helper1 ="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ201234567890";
const string helper2 ="BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890";
void Encrypt(string str){
    string res;
    for(int i = 0; i < str.size(); i++){
        if(isalpha(str[i]) || isdigit(str[i]))
            res += helper2[helper1.find(str[i])];
        else
            res += str[i];
    }
    cout << res << endl;
}

void UnEncrypt(string str){
    string res;
    for (int i = 0; i < str.size(); i++){
        if(isalpha(str[i]) || isdigit(str[i]))
            res += helper1[helper2.find(str[i])];
        else
            res += str[i];
    }
    while(cin >> wp >> p){
```

这种加密题，一种方法是根据加密规则写出对应编码表，下标——对应，再使用find函数返回的下标

这种加密题，一种方法是根据加密规则写出对应编码表，下标一一对应，再使用find函数返回的下标作为对应编码数组的下标，将对应的字符累加到定义的空串res 得到最终的字符串；方法二：

```
string Encrypt(string &a){  
    for(int i = 0; i < a.size(); i++){  
        if(a[i] >= 'A' && a[i] <= 'Z'){  
            if(a[i] == 'Z')  
                a[i] = 'a';  
            else a[i] += 33;  
        }  
        else if(a[i] >= 'a' && a[i] <= 'z'){  
            if(a[i] == 'z')  
                a[i] = 'A';  
            else a[i] -= 31;  
        }  
        else if(a[i] >= '0' && a[i] <= '9'){  
            if(a[i] == '9')  
                a[i] = '0';  
            else a[i] += 1;  
        }  
    }  
    return a;  
}
```

对输入string 进行遍历，对于每一个a[i]进行对应的替换，注意a[i] 是一个字符 a[i]+=33 , B a[i]-=31; 返回的a[i]还是一个字符!!!

输入

abcdefg
BCDEFGH

输出

BCDEFGH
abcdefg

下标为奇数和下标为偶数的字符 分别从小到大排序

对于这种每组输入有2条/2行数据 同时又有多组输入

```
int main(){  
    string wp, p;  
    while(cin>>wp>>p){  
        Encrypt(wp);  
        UnEncrypt(p);  
        cout << wp << endl;  
        cout << p << endl;  
    }  
}
```

可以用这种连续输入的方法while(cin >>wp >>p)

字符 - 字符 如 str[i] - 'a' 同类型的减法 相当于ascii码相减 得到的结果是一个整数
字符 - 数字 如 str[i] - 31 会将 前面字符的ascii码操作， 得到的结果仍然是一个字符

位倒序：

```
int WeiDaoxu(int num) //位倒序操作  
{  
    int j=8,i=0;  
    int tmp[4] = { 0 };  
    for(i=0;i<4;i++)  
    {  
        tmp[i] = num%2;  
        num /= 2;  
    }  
    num = 0;  
    for(i = 0; i < 4; i++)  
    {  
        num += j * tmp[i];  
        j /= 2;  
    }  
    return num;
```

按照指定规则对输入的字符串进行处理。

详细描述：

将输入的两个字符串合并。

对合并后的字符串进行排序，要求为：下标为奇数的字符和下标为偶数的字符分别从小到大排序。这里的下标意思是字符在字符串中的位置。

对排序后的字符串进行操作，如果字符为'0'——'9'或者'A'——'F'或者'a'——'f'，则对他们所代表的16进制的数进行BT倒序的操作，并转换为相应的大写字母。如字符为'4'，为0100b，则翻转后为0010b，也就是2。转换后的字符为'2'；如字符为'7'，为0111b，则翻转后为1110b，也就是e。转换后的字符为大写'E'。

举例：输入str1为“dec”，str2为“fab”，合并为“decfab”，分别对“dca”和“efb”进行排序，排序后为“abcd”，转换后为“5D37BF”

注意本题含有多组样例输入

输入描述：

本题含有多组样例输入。每组样例输入两个字符串，用空格隔开。

输出描述：

输出转化后的结果。每组样例输出一行。

示例1

输入 复制
dec fab

```
int main(){
    string str1, str2;
    while(cin >> str1 >> str2) {
        string s, s1, s2;
        s = str1 + str2;
        int len = s.size();
        for(int i = 0; i < s.size(); ++i) {
            if(i % 2 == 0)
                s1 += s[i];
            else
                s2 += s[i];
        }
        sort(s1.begin(), s1.end());
        sort(s2.begin(), s2.end());
        s.clear();
        for(int i = 0, j = 0, k = 0; i < len; ++i) {
            if(i % 2 == 0)
                s += s1[j++];
            else
                s += s2[k++];
        }
        for(int i = 0; i < s.size(); ++i) {
            int n = helper1.find(s[i]);
            if(n != -1)
                s[i] = helper2[n];
        }
        cout << s << endl;
    }
}
```

```
    s.clear();
    for(int i = 0, j = 0, k = 0; i < len; ++i) {
        if(i % 2 == 0)
            s += s1[j++];
        else
            s += s2[k++];
```

这个地方不能用*i < s.size()*，因为前面已经用了*s.clear()*；这种方法称为查表法

题目描述

对字符串中的所有单词进行倒排。

说明：

- 1、构成单词的字符只有26个大写或小写英文字母；
- 2、非构成单词的字符均视为单词间隔符；
- 3、要求倒排后的单词间隔符以一个空格表示；如果原字符串中相邻单词间有多个间隔符时，倒排转换后也只允许出现一个间隔符；
- 4、每个单词最长20个字母；

输入描述:

输入一行以空格来分隔的句子

输出描述:

输出句子的逆序

示例1

The screenshot shows a simple text editor interface. On the left, there is an 'input' field containing the sentence "I am a student". On the right, there is an 'output' field containing the reversed sentence "student a am I". Each field has a 'copy' button next to it.

```
int main() {
    string str;
    while(getline(cin,str)){
        //vector<vector<int>> vec;
        vector<string> res;
        string temp;
        for(auto c : str){
            if((c>= 'a' && c<= 'z') || (c>= 'A' && c<= 'Z')){
                temp += c;
            }else if(temp.size()>0){ //空串push到vector会造成输出空格错误
                res.push_back(temp);
                temp.clear();
            }
        }
        if(temp.size() > 0) //不能将空串push到vector内·会造成空格·格式错误
            res.push_back(temp);

        for(int i = res.size()-1; i>=0; i--){
            cout << res[i] << ' ';
        }
        cout << res[0] << endl;
    }
    return 0;
}
```

题目描述

Catcher是MCA国的情报员，他工作时发现敌国会用一些对称的密码进行通信，比如像这些ABBA , ABA , A , 123321，但是他们有时会在开始或结束时加入一些无关的字符以防止别国破解。比如进行下列变化 ABBA->12ABBA,ABA->ABAKK,123321->51233214 。因为截获的串太长了，而且存在多种可能的情况（abaaab可看作是aba,或baaab的加密形式），Cather的工作量实在是太大了，他只能向电脑高手求助，你能帮Cather找出最长的有效密码串吗？

本题含有多组样例输入。

The screenshot shows a C++ code editor with the following code:

```
int main(){
    while(cin>> str){
        int maxn = 0, flag = 0;
        for(int i = 0; i<str.size(); i++){
            for(int j = str.size()-1; j>i;j--){
                flag = 0;
                int l, r;
                for(l = i, r = j; l <=(i+j)/2; l++,r--){
                    if(str[l]!=str[r]){
                        cout << 'a' << endl;
                        flag = 1;
                        break;
                    }
                }
                if(flag == 0){
                    maxn = max(maxn, (j-i+1));
                    break;
                }
            }
            cout << maxn << endl;
        }
        return 0;
}
```

$i \leq (i+j)/2$ 如果改成 $i < (i+j)/2$ 会出现问题 如 $i \leq 1$ 与 $i < 1$ 会忽略一组比较
这个表达式会先计算 $(i+j)/2$ 再与左边混合 如 $i \leq (0+3)/2 \dots i \leq 1$ 与 $i < 1$

题目描述

原理：ip地址的每段可以看成是一个0-255的整数，把每段拆分成一个二进制形式组合起来，然后把这个二进制数转变成一个长整数。

举例：一个ip地址为10.0.3.193

每段数字 相对应的二进制数

10 00001010

0 00000000

3 00000011

193 11000001

组合起来即为：00001010 00000000 00000011 11000001,转换为10进制数就是：
167773121，即该IP地址转换后的数字就是它了。

本题含有多组输入用例，每组用例需要你将一个ip地址转换为整数、将一个整数转换为ip地址。

```
int main(){
    long long n, a1, a2, a3, a4;
    char ch;
    while(cin >>a1>>ch>>a2>>ch>>a3>>ch>>a4){
        cin >> n;
        long long res = 0;
        res += (a1<<24) +(a2<<16) +(a3 << 8) +a4;
        a1 = n >>24;
        a2 = (n>>16) &255;
        a3 = (n >>8) &255;
        a4 = n &255;
        cout << res << endl <<a1 <<','<<a2<<','<<a3<<','<<a4<<endl;
    }
    return 0;
}
```

题目描述

有一种技巧可以对数据进行加密，它使用一个单词作为它的密匙。下面是它的工作原理：首先，选择一个单词作为密匙，如TRAILBLAZERS。如果单词中包含有重复的字母，只保留第1个，其余几个丢弃。现在，修改过的那个单词属于字母表的下面，如下所示：

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TRAILB ZES C D F G H J K M N O P Q U V W X Y

上面其他字母表中剩余的字母填充完整。在对信息进行加密时，信息中的每个字母被固定于顶上那行，并用下面那行的对应字母——取代原文的字母(字母字符的大小写状态应该保留)。因此，使用这个密匙，Attack AT DAWN(黎明时攻击)就会被加密为Tpplad TP ITVH。

请实现下述接口，通过指定的密匙和明文得到密文。

本题有多组输入数据。

输入描述:

先输入key和要加密的字符串

输出描述:

返回加密后的字符串

示例1

输入
nihao
ni

```
int main(){
    string key, str;
    while(cin >>key>>str){
        int flag[26] = {0};
        string table;
        for(auto c : key){
            c = toupper(c);
            if(flag[c - 'A'] == 0){
                table.push_back(c);
                flag[c - 'A'] = 1;
            }
        }
        for(int i = 'A'; i <= 'Z'; i++){
            if(table.find(i) == string::npos)
                table.push_back(i);
        }
        //for(auto c : str){ //若要修改str必须使用下标或者这种引用的方式:如果不用auto引用
            for(int i = 0; i < str.size(); i++){ //不会更改str对象要使用for(auto &c : str)
                if(isupper(str[i]))
                    str[i] = toupper(table[str[i]- 'A']);
                else if(islower(str[i]))
                    str[i] = tolower(table[str[i]- 'a']);
            }
            cout << str << endl;
        }
    }
}
```

题目描述

有一只兔子，从出生后第3个月起每个月都生一只兔子，小兔子长到第三个月后每个月又生一只兔子，假如兔子都不死，问每个月的兔子总数为多少？

本题有多组数据。

输入描述:

输入int型表示month

输出描述:

输出兔子总数int型

```
int main() {
    int n;
    while(cin>>n) {
        int shu1 = 1;
        int shu2 = 0;
        int shu3 = 0;
        while(--n){ // & &
            shu3 += shu2;
            shu2 = shu1;
            shu1 = shu3;
        }
        cout<<shu1 + shu2 + shu3<<endl;
    }
}
```

这里必须是 --n，因为n=1 已经有值了 如 n=3 只会进行2次繁衍迭代，循环两次

n=3 --n= 2; & ; --n=1 ;& --n =0 退出循环

n-- :n = 3 & n-- =2; & n-- =1; & n--=0 退出循环

题目描述

输入一行字符，分别统计出包含英文字母、空格、数字和其它字符的个数。

本题包含多组输入。

输入描述:

输入一行字符串，可以有空格

输出描述:

统计其中英文字符，空格字符，数字字符，其他字符的个数

```
int main() {
    string str;
    while(getline(cin, str)){
        int a = 0, b = 0, c = 0, d = 0;
        int len = str.size();
        for(int i = 0; i < len; i++){
            if(isalpha(str[i]))
                a++;
            else if(isspace(str[i])) // (str[i] == ' ')
                b++;
            else if(isdigit(str[i]))
                c++;
            else if(ispunct(str[i])) // else
                d++;
        }
        cout<<a<<endl<<b<<endl<<c<<endl<<d<<endl;
    }
}
```

输入

```
1qazxsw23 edcvfr45tgbn hy67uj m,ki89o1.\\";/p0-=\\]{
```

输出

```
26  
3  
10  
12
```

```

int main(){
    string str;
    //int a=0, b=0, c=0, d=0;
    while(getline(cin, str)){
        int a=0, b=0, c=0, d=0;
        for(auto i : str){
            if(isalpha(i))
                a++;
            else if(isdigit(i))
                b++;
            else if(isspace(i))
                c++;
            else if(ispunct(i))
                d++;
        }
        //cout<<a<<endl<<c<<endl<<b<<endl<<d<<endl;
        cout<<a<<endl<<c<<endl<<b<<endl<<d<<endl;
    }
}

```

这几个统计变量 a b c d 如果定义在while(getline(cin,str))外面 那么只会初始化一次
由于下面的是累加操作 对于每次输入str都应该先将 这四个计数变量初始为0 再进行累加
注意输出的形式是每一个进行一个换行 因此应该在每个输出接一个endl

题目描述

定义一个二维数组N*M (其中 $2 \leq N \leq 10, 2 \leq M \leq 10$) , 如 5×5 数组下所示 :

```

int maze[5][5] = {
0, 1, 0, 0, 0,
0, 1, 0, 1, 0,
0, 0, 0, 0, 0,
0, 1, 1, 1, 0,
0, 0, 0, 1, 0,
};

```

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。入口点为 $[0,0]$ 既第一空格是可以走的路。

```

int N, M;
vector<vector<int>> maze; //全局变量声明
vector<pair<int,int>> path_temp;
vector<pair<int,int>> path_best;
void MazeTrack(int i, int j){
    maze[i][j] = 1;
    path_temp.push_back(make_pair(i,j));
    if(i == N-1 && j == M-1)
        if(path_best.empty() || path_temp.size() < path_best.size())
            path_best = path_temp;
    if(i - 1 >= 0 && maze[i-1][j]==0)
        MazeTrack(i-1, j);
    if(i + 1 <= N-1 && maze[i+1][j]==0)
        MazeTrack(i+1, j);
    if(j - 1 >= 0 && maze[i][j-1]==0)
        MazeTrack(i, j-1);
    if(j + 1 <= M-1 && maze[i][j+1]==0)
        MazeTrack(i, j+1);
    maze[i][j] = 0;
    path_temp.pop_back();
}

```

```

int main(){
    while(cin >> N >> M){
        maze = vector<vector<int>>(N, vector<int>(M, 0));
        path_temp.clear();
        path_best.clear();
        //for(auto &i : maze) //如果要加引用，就都加引用
        //for(auto &j : i) //不加这个引用就错了
        //cin >> j;
        for(int i = 0; i < N; i++)
            for(int j = 0; j < M; j++)
                cin >> maze[i][j];
        MazeTrack(0, 0);
        for(auto i : path_best)
            cout << '(' << i.first << ', ' << i.second << ')' << endl;
    }
}

```

以递归回溯的方法找最短路径

注意二维数组初始输入赋值的方法 (都用引用//或者两层for循环)

输入描述:

整数N，后续N个名字

输出描述:

每个名称可能的最大漂亮程度

示例1

输入

```
2
zhangsan
lisi
```

输出

```
192
161
```

```
for(i=0; i<st.length(); i++){
    if(st[i]>='a' && st[i]<='z')
        a[st[i]-'a']++;
    else
        a[st[i]-'A']++;
}
sort(a,a+26); //sort 可以用于数组指针
for(i=25; i>=0; --i)
    res += a[i]*k--;
cout<<res<<endl;
```

```
int main(){
    int num= 0;
    string str;
    while(cin >> num){ //对于一个数正面接多种组数据，一种方法是用while(test--), 用while循环进行输入
        for(int i = 0; i < num; ++i){ //还有一种方法是用一个for循环，用for循环进行输入
            cin >> str;
            cout<< you_name(str)<< endl;
        }
    }
    return 0;
}
```

```
int main(){
    while(cin>>n){
        while(n--){
            cin >> str;
            int cnt[26]={0};
            int total = 0; int k = 26;
            vector<int> vec;
            for(auto c : str){
                ++cnt[toupper(c) - 'A'];
            }
            for(auto c : cnt){
                if(c != 0) vec.push_back(c);
            }
            sort(vec.begin(), vec.end());
            reverse(vec.begin(), vec.end());
            //for(int i = vec.size() - 1; i>=0; i--){
            //    total += vec[i]*(k--);
            //    for(auto i : vec){
            //        total += i*(k--);
            //    }
            cout << total << endl;
        }
    }
}
```

请注意如果不使用新式：for循环，用size() //就必须使用下标操作

```
string 类型的find函数："
string s1 = "abcdef";
string s2 = "de";
int ans = s1.find(s2); //在S1中查找子串S2
```

说明：如果查找成功则输出查找到的第一个位置，否则返回-1；

```
string s1 = "abcdef";
string s2 = "de";
int ans = s1.find(s2, 2); //从S1的第二个字符开始查找子串S2
```

```
string s1 = "adedef";
string s2 = "dek";
int ans = s1.find_first_of(s2); //在S1中查找子串S2
```

查找子串中的某个字符最先出现的位置。find_first_of()不是全匹配，而find()是全匹配

3.find_last_of()

这个函数与find_first_of()功能差不多，只不过find_first_of()是从字符串的前面往后面搜索，而find_last_of()是从字符串的后面往前面搜索。

Vector中的find函数：

find函数主要实现的是在容器内查找指定的元素，并且这个元素必须是基本数据类型的。

查找成功返回一个指向指定元素的迭代器，查找失败返回end迭代器。

find()函数的定义：

```
template <class InputIterator, class T>InputIterator find( InputIterator first, InputIterator last, const T& value) { while (first != last && *first != value) ++first; return first; }
int nums[] = { 3, 1, 4, 1, 5, 9 };
int num_to_find = 5;
int start = 0;
int end = 5;
int* result = find( nums + start, nums + end, num_to_find );
```

map的find和count函数

使用count，返回的是被查找元素的个数。如果有，返回1；否则，返回0。注意，map中不存在相同元素，所以返回值只能是1或0。

使用find，返回的是被查找元素的位置，没有则返回map.end()。

```
map<string,int> test;
test.insert(make_pair("test1",1));
test.insert(make_pair("test2",2));
```

如果map中存在所要查找的主键，就返回—

```
int count = test.count("test2");
cout<<"the value of count:"<<count<<endl;
map<string,int>::iterator it = test.find("test2");
```

如果map中没有索要查找的主键，就返回map.end();

Vector

1 基本操作

(1)头文件#include<vector>.

(2)创建vector对象，vector<int> vec;

(3)尾部插入数字：vec.push_back(a);

(4)使用下标访问元素，cout<<vec[0]<<endl;记往下标是从0开始的。

(5)使用迭代器访问元素.

语法：

```
vector向量, insert函数原型
1 插入位置, 插入值
2 iterator insert(iterator __position, const value_type& __x);
3 插入位置, 插入数量, 插入值
4 void insert(iterator __position, size_type __n, const value_type& __x);
5 插入位置, 迭代器开始位, 迭代器结束位
6 template<typename _InputIterator>
7 void insert(iterator __position, _InputIterator __first, _InputIterator __last)
```

```
vector向量, erase函数原型
1 iterator erase(iterator __position);
2 iterator erase(iterator __first, iterator __last);
```

```

21. vector::insert()

iterator insert ( iterator position, const T& x );
void insert ( iterator position, size_type n, const T& x );
template <class InputIterator>
void insert ( iterator position, InputIterator first, InputIterator last );

插入新的元素，

第一个函数，在迭代器指定的位置前插入值为x的元素
第二个函数，在迭代器指定的位置前插入n个值为x的元素
第三个函数，在迭代器指定的位置前插入另外一个容器的一段序列迭代器first到last
若插入新的元素后总得元素个数大于capacity，则重新分配空间

```

3 算法

(1) 使用reverse将元素翻转：需要头文件#include<algorithm>

reverse(vec.begin(),vec.end());将元素翻转(在vector中，如果一个函数中需要两个迭代器，一般后一个都不包含。)

(2) 使用sort排序：需要头文件#include<algorithm>，

sort(vec.begin(),vec.end());(默认是按升序排列,即从小到大)。

可以通过重写排序比较函数按照降序比较，如下：

定义排序比较函数：

```

bool Comp(const int &a,const int &b)
{
    return a>b;
}

```

调用时:sort(vec.begin(),vec.end(),Comp)，这样就降序排序。

语法：

```

vector向量, insert函数原型
1 1.插入位置, 插入值
2 iterator insert(iterator __position, const value_type& __x);
3 2.插入位置, 插入数量, 插入值
4 void insert(iterator __position, size_type __n, const value_type& __x);
5 3.插入位置, 迭代器开始位, 迭代器结束位
6 template<typename _InputIterator>
7 void insert(iterator __position, _InputIterator __first, _InputIterator __last)

```

题目描述

输入一个单向链表和一个节点的值，从单向链表中删除等于该值的节点，删除后如果链表中无节点则返回空指针。

链表的值不能重复。

构造过程，例如

```

1 <- 2
3 <- 2
5 <- 1
4 <- 5
7 <- 2

```

最后的链表的顺序为 2 7 3 1 5 4

删除结点 2

则结果为 7 3 1 5 4

链表长度不大于1000，每个节点的值不大于10000。

本题含有多组样例。

输入描述:

```

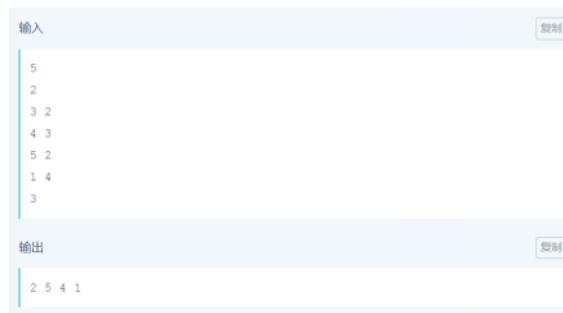
1 输入链表结点个数
2 输入头结点的值
3 按照格式插入各个结点
4 输入要删除的结点的值

```

输出描述:

输出删除结点后的序列，每个数后都要加空格

示例1



输入
5
2
3 2
4 3
5 2
1 4
3
输出
2 5 4 1

```
int main(){
    while(cin >> n >> headV) {
        int back, front;
        vector<int> res;
        res.push_back(headV);
        while(--n) {
            cin >> back >> front;
            auto it = find(res.begin(), res.end(), front);
            if(it != res.end())
                res.insert(it+1, back); //在it +1 前面插入元素 ··· 1 ·2 ·3 ··· v.begin() 在begin()前面 插入元素
            else
                res.push_back(back);
        }
        int m;
        cin >> m;
        auto it2 = find(res.begin(), res.end(), m);
        res.erase(it2);
        for(auto x : res)
            cout << x << ' ';
        cout << endl;
    }
}
```

vector中需要删除一个元素：

auto it = find(vec.begin(), vec.end(), m);
vec.erase(it); vec中找到的就是迭代器，string中找到的为pos map中找到的也是迭代器 只是string为常数pos

这里用while(--n)，当n=5 下面实际只有4个结点输入 用--n 循环只执行4次

如果用while(n--) n=5 循环执行5 次 而下面没有5个结点输入 就会出现循环无法退出 发生段错误

题目描述

输入一个单向链表，输出该链表中倒数第k个结点，链表的倒数第1个结点为链表的尾指针。

链表结点定义如下：

```
struct ListNode
{
    int    m_nKey;
    ListNode* m_pNext;
};
```

正常返回倒数第k个结点指针，异常返回空指针

本题有多组样例输入。

输入描述:

输入说明
1 输入链表结点个数
2 输入链表的值
3 输入k的值

输出描述:

输出一个整数

示例1

The screenshot shows a development environment with three panes:

- 输入 (Input):** Contains the following text:

```
8
1 2 3 4 5 6 7 8
4
```
- 输出 (Output):** Contains the number 5.
- 代码 (Code):** Displays a C++ program to calculate the Levenshtein distance between two strings. The code defines a linked list node structure and a main function that reads input values n, m, and k, creates a linked list, and then iterates through the list to print each node's data.

```
#include<bits/stdc++.h>
using namespace std;
struct ListNode{
    int data;
    ListNode* next;
};

int main(){
    int n, m, k;
    while(cin >> n){
        ListNode *p, *head;
        head = new ListNode;
        head->next = NULL;
        for(int i = 0; i < n; i++){
            cin >> m;
            p = new ListNode;
            p->data = m;
            p->next = head->next;
            head->next = p;
        }
        p = head;
        cin >> k;
        while(k--){
            p = p->next;
        }
        cout << p->data << endl;
    }
    return 0;
}
```

题目描述

Levenshtein 距离，又称编辑距离，指的是两个字符串之间，由一个转换成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。编辑距离的算法是首先由俄国科学家Levenshtein提出的，故又叫Levenshtein Distance。

Ex :

字符串A:abcdefg

字符串B: abcdef

通过增加或是删掉字符“g”的方式达到目的。这两种方案都需要一次操作。把这个操作所需要的次数定义为两个字符串的距离。

要求：

给定任意两个字符串，写出一个算法计算它们的编辑距离。

本题含有多组输入数据。

输入描述:

每组用例一共2行，为输入的两个字符串

```

int Distance(string a, string b){
    int n = a.size(), m = b.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
    for(int i = 1; i <= m; ++i)
        dp[0][i] = i;
    for(int i = 1; i <= n; ++i)
        dp[i][0] = i;
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= m; ++j) {
            int one = dp[i-1][j] + 1;
            int two = dp[i][j-1] + 1;
            int three = dp[i-1][j-1];
            if(a[i-1] != b[j-1])
                three += 1;
            dp[i][j] = min(min(one, two), three);
        }
    }
    return dp[n][m];
}
int main(){
    string a, b;
    while(cin >>a >>b){
        cout << Distance(a,b) << endl;
    }
    return 0;
}

```

```

int main(){
    string a, b;
    while(cin >>a >>b){
        cout << Distance(a,b) << endl;
    }
    return 0;
}

```

题目描述

1
 1 1 1
 1 2 3 2 1
 1 3 6 7 6 3 1
 1 4 10 16 19 16 10 4 1

以上三角形的数阵，第一行只有一个数1，以下每行的每个数，是恰好是它上面的数，左上角数到右上角的数，3个数之和（如果不存在某个数，认为该数就是0）。
 求第n行第一个偶数出现的位置。如果没有偶数，则输出-1。例如输入3，则输出2，输入4则输出3。

输入n($n \leq 1000000000$)

本题有多组输入数据，输入到文件末尾，请使用while(cin>>)等方式读入

输入描述:

输入一个int整数

输出描述:

输出返回的int值

示例1

输入	复制
4 2	复制
输出	复制
3 -1	复制

```
int main() {
    while(cin >> n) {
        if(n <= 2)
            cout << -1 << endl;
        else if(n % 2 == 1)
            cout << 2 << endl;
        else if(n % 4 == 2)
            cout << 4 << endl;
        else if(n % 4 == 0)
            cout << 3 << endl;
    }
    return 0;
}
```

while(cin >> n) 由于有换行符分隔，cin每次循环读取一个字符到n 找规律问题

题目描述

输出7有关数字的个数，包括7的倍数，还有包含7的数字（如17，27，37...70，71，72，73...）的个数（一组测试用例里可能有多组数据，请注意处理）

输入描述:

一个正整数N。（N不大于30000）

输出描述:

不大于N的与7有关的数字个数，例如输入20，与7有关的数字包括7, 14, 17.

```
int main() {
    long int n;
    //int num = 0;
    while(cin >> n) {
        int num = 0;
        string str;
        for(int i = 7; i <= n; i++) {
            str = to_string(i);
            //cout << str << endl;
            if((str.find('7') != string::npos) || (i % 7 == 0))
                num++;
        }
        cout << num << endl;
    }
    return 0;
}
```

使用到了 to_string(n)函数 整数转string类型

注意这个 num变量的初始位置，由于每次输入实例都会对Num进行累加，而每次输入实例统计完，num应该重置为0，因此num应该在while(cin >> n)循环体内部进行初始化工作，确保每次输入循环都会重新创建Num并初始为0供计数使用

题目描述

完全数（Perfect number），又称完美数或完备数，是一些特殊的自然数。它所有的真因子（即除了自身以外的约数）的和（即因子函数），恰好等于它本身。例如：28，它有约数1、2、4、7、14、28，除去它本身28外，其余5个数相加， $1+2+4+7+14=28$ 。
输入n，请输出n以内(含n)完全数的个数。计算范围， $0 < n \leq 500000$
本题输入含有多组样例。

输入描述:

输入一个数字n

输出描述:

输出不超过n的完全数的个数

示例1

输入

```
1000
7
100
```

输出

```
3
1
2
```

```
bool IsFull(int n) {
    int sum = 0;
    if(n <= 3) {
        return false;
    } else {
        for(int i = 1; i <= n; i++) {
            if(n % i == 0)
                sum += i;
        }
    }
    if(sum == n)
        return true;
    return false;
}

int main() {
    int n;
    while(cin >> n) {
        int total = 0;
        for(int i = 1; i <= n; i++) {
            if(IsFull(i))
                total++;
        }
        cout << total << endl;
    }
    return 0;
}
```

注意这个total由于每次输入循环都要重置这个total为0，因此total 应该在while(cin>>n)
循环体内部进行初始化处理，每次循环都会创建这个变量并初始为0 作为计数单元

题目描述

输入n个整数，输出其中最小的k个。
本题有多组输入样例，请使用循环读入，比如while(cin>>)等方式处理

输入描述:

第一行输入两个整数n和k
第二行输入一个整数数组

输出描述:

输出一个从小到大排序的整数数组

示例1

输入

```
5 2
1 3 5 7 2
```

输出

```
1 2
```

```

#include<bits/stdc++.h>
using namespace std;
int n, m, k;
int main() {
    while(cin >> n >> k) {
        vector<int> vec;
        while(n--) {
            cin >> m;
            vec.push_back(m);
        }
        sort(vec.begin(), vec.end());
        //cin >> k;
        for(int i = 0; i < k - 1; i++) {
            cout << vec[i] << ' ';
        }
        cout << vec[k-1] << endl;
    }
}

```

题目描述

题目描述

把m个同样的苹果放在n个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？（用K表示）5，1，1和1，5，1是同一种分法。

数据范围：0≤m≤10，1≤n≤10。

本题含有多组样例输入。

输入描述：

输入两个int整数

输出描述：

输出结果，int型

示例1

输入	复制
7 3	
输出	复制
8	

```

int put_apples(int m, int n) {
    if (m < 0)
        return 0;
    if (m == 1 || n == 1)
        return 1;
    return put_apples(m, n-1) + put_apples(m-n, n);
}
int main() {
    int m, n;
    while(cin >> m >> n) {
        cout << put_apples(m, n) << endl;
    }
}

```

有空盘子 那就是m个苹果放到n-1个盘子中

没有空盘子 那就是m-n个苹果放到n个盘子中 递归的思想

边界条件是M==1 || N==1 返回一种放法 m<0 返回0种放法 分解累加

输入一个正整数，计算它在二进制下的1的个数。

注意多组输入输出！！！！！！

输入描述：

输入一个整数

输出描述：

计算整数二进制中1的个数

```
int main() {
    int n;
    while(cin >> n) {
        int sum = 0;
        while(n > 0) {
            if(n&1 == 1)
                sum++;
            n >>= 1;
        }
        cout << sum << endl;
    }
}
```

题目描述

查找两个字符串a,b中的最长公共子串。若多个，输出在较短串中最先出现的那个。
注：子串的定义：将一个字符串删去前缀和后缀（也可以不删）形成的字符串。请和“子序列”的概念分开！

本题含有多组输入数据！

输入描述：

输入两个字符串

输出描述：

返回重复出现的字符

示例1

输入	abcdefgijklmnop abcsafjklmnopqrstuvwxyz	复制
输出	jklmноп	复制

```
int main(){
    string a, b ;
    while(cin >> a >> b){
        if(a.size() > b.size())
            swap(a,b);

        string temp, str_m;
        for(int i = 0; i < a.size(); i++){
            for(int j = i; j < a.size(); j++){
                temp = a.substr(i, j-i+1);
                if(b.find(temp) == string::npos)
                    break;
                else if(temp.size() > str_m.size())
                    str_m = temp;
            }
        }
        cout << str_m << endl;
}
```

注意 swap(a,b) 可以交换两个string 保证 a.size要小于b 在b中find a的子串
temp = a.substr(i,j-i+1) 注意长度是j-i+1位 i=0开始， 循环从j=i开始 依次取子串
在b中使用find(temp)查找这个子串 没有找到直接break， i右移一位 找到了
比较当前子串与存储最大长度的str_m.size()，如果当前 temp更大，则更新str_m，
最终输出的str_m就是最大子串的长度

stringstream 是将字符串变成字符串迭代器一样，将字符串流在依次拿出，它不会将空格作为流。这样就实现了字符串的空格切割

```

1 #include<bits/stdc++.h>
2 #include<iostream> //头文件
3 using namespace std;
4 int main(){
5     string str="nice to meet you";
6     stringstream stream(str);
7     string s;
8     while(stream>>s){
9         cout<<s<<endl;
10    }
11 }
12 }
```

输出：

```

1 nice
2 to
3 meet
4 you
```

```

12 int main()
13 {
14     stringstream mmap;
15     string s, wocao;
16     getline(cin, s);
17     mmap<<s;
18     while(mmap>>wocao)
19         cout<<wocao<<endl;
20     return 0;
21 }
```

输入：asd asd asd

输出：

```

asd
asd
asd
```

题目描述

有6条配置命令，它们执行的结果分别是：

命 令	执 行
reset	reset what
reset board	board fault
board add	where to add
board delete	no board at all
reboot backplane	impossible
backplane abort	install first
<i>he he</i>	unknown command

注意：**he he不是命令。**

为了简化输入，方便用户，以“最短唯一匹配原则”匹配：
1、若只输入一字符串，则只匹配一个关键字的命令行。例如输入：r，根据该规则，匹配命令reset，执行结果为：reset what；输入：res，根据该规则，匹配命令reset，执行结果为：reset what；

- 2、若只输入一字符串，但本条命令有两个关键字，则匹配失败。例如输入：reb，可以找到命令reboot backplane，但是该命令有两个关键词，所有匹配失败，执行结果为：unknown command
- 3、若输入两字符串，则先匹配第一关键字，如果有匹配但不唯一，继续匹配第二关键字，如果仍不唯一，匹配失败。例如输入：rb，找到匹配命令reset board 和 reboot backplane，执行结果为：unknown command。
- 4、若输入两字符串，则先匹配第一关键字，如果有匹配但不唯一，继续匹配第二关键字，如果唯一，匹配成功。例如输入：ba，无法确定是命令board add还是backplane abort，匹配失败。
- 5、若输入两字符串，第一关键字匹配成功，则匹配第二关键字，若无匹配，失败。例如输入：ba a，确定是命令board add，匹配成功。
- 6、若匹配失败，打印“unknown command”

输入描述:

多行字符串，每行字符串一条命令

输出描述:

执行结果，每条命令输出一行

示例1

The screenshot shows a terminal window with two tabs: '输入' (Input) and '输出' (Output). In the '输入' tab, several commands are listed:
reset
reset board
board add
board delete
reboot backplane
backplane abort

In the '输出' tab, the command 'reset what' is shown.

stringstream 可以完成字符串按空格划分

```
string str;
string a[6]={"reset","reset board","board add","board delete","reboot ba
string b[7]={"reset what","board fault","where to add","no board at all"
while(getline(cin,str)){
    string str1, str2;
    stringstream ss(str);
    ss >> str1 >> str2;
    string templ, temp2;
    int sum = 0, c = 0;
    if(str2.empty()){
        if(a[0].find(str1) == 0)
            cout << b[0] << endl;
        else
            cout << b[6] << endl;
    }
    else{
        for(int i = 1; i < 6; i++){
            stringstream ss2(a[i]);
            ss2>>templ>>temp2;
            if(templ.find(str1) == 0 && temp2.find(str2) == 0){
                sum++;
                c = i;
            }
        }
        if(sum == 1){
            cout << b[c] << endl;
        }
        else
            cout << b[6] << endl;
    }
}
```

利用stringstream 从ss>>str1>>str2 每次取出一个字符串/单词，利用find函数进行查找

将一行 // 以空格分隔的一行 或者数组元素 stringstream ss (X) 的形式初始化读入到 stringstream

再用ss>>str1从ss中取的时候 每调用一次>> 只会取出一个字符串/以空格为间隔取到str1/ temp1

题目描述

查找和排序

题目：输入任意（用户名，成绩）序列，可以获得成绩从高到低或从低到高的排列。相同成绩都按先录入排列在前的规则处理。

例如：

```
jack 70
peter 96
Tom 70
smith 67
从高到低 成绩
peter 96
jack 70
Tom 70
smith 67
从低到高
smith 67
jack 70
Tom 70
peter 96
```

注：0代表从高到低，1代表从低到高

本题含有多少组输入数据！

输入描述:

输入多行，先输入要排序的人的个数，然后分别输入他们的名字和成绩，以一个空格隔开

输出描述:

按照指定方式输出名字和成绩，名字和成绩之间以一个空格隔开

示例1

输入

```
3
0
fang 90
yang 50
ning 70
```

输出

```
fang 90
ning 70
```

```
#include<bits/stdc++.h>
using namespace std;
bool cmp0(const pair<string,int> &a, const pair<string,int> &b) {
    return a.second > b.second;
}
bool cmp1(const pair<string,int> &a, const pair<string,int> &b) {
    return a.second < b.second;
}
int main(){
    int n, m;
    while(cin >> n) {
        cin >> m;
        vector<pair<string,int>> res(n);
        for(int i = 0; i < n; i++) { //在需要输出时需要下标操作 不能使用while(n--)
            cin >> res[i].first >> res[i].second;
        }
        if(m == 0)
            stable_sort(res.begin(), res.end(), cmp0);
        else if(m == 1)
            stable_sort(res.begin(), res.end(), cmp1);
        for(auto x : res)
            cout << x.first << ' ' << x.second << endl;
    }
    return 0;
}
```

学会定义 `pair<string, int>` 放到vector里面排序

```
bool cmp0(const pair<string,int>& a, const pair<string,int>& b){
    return a.second > b.second;
}
vector<pair<string,int>> res(n);
for(int i = 0; i < n; i++){
    cin >> res[i].first >> res[i].second;
}
```

题目描述

公元前五世纪，我国古代数学家张丘建在《算经》一书中提出了“百鸡问题”：鸡翁一值钱五，鸡母一值钱三，鸡雏三值钱一。百钱买百鸡，问鸡翁、鸡母、鸡雏各几何？

详细描述：

接口说明

原型：

```
int GetResult(vector &list)
```

输入参数：

无

输出参数（指针指向的内存区域保证有效）：

list 鸡翁、鸡母、鸡雏组合的列表

返回值：

-1 失败

0 成功

```
int main() {
    int n;
    int m;
    while(cin >> n) {
        for(int x = 0; x <= 20; x++) {
            double y = (200 - 14 * x) / 8.0;
            double z = 100 - x - y;
            if(y == int(y) && y >= 0 && z >= 0)
                if(y == int(y) && y >= 0 && z >= 0)
                    cout << x << ' ' << y << ' ' << z << endl;
        }
    }
    return 0;
}
```

题目描述

根据输入的日期，计算是这年的第几天。。
测试用例有多组，注意循环输入

输入描述:

输入多行，每行空格分割，分别是年，月，日

输出描述:

成功：返回outDay输出计算后的第几天；

失败：返回-1

```
#include<bits/stdc++.h>
using namespace std;
const int days[] = {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304};
int main(){
    int y, m, d;
    while(cin >> y >> m >> d) {
        int ans = days[m-1] + d;
        if(((y%4 == 0 && y%100 != 0) || (y%400 == 0)) && m > 2)
            ans += 1;
        cout << ans << endl;
    }
    return 0;
}
```

解析规则：

- 参数分隔符为空格
- 对于“包含起来的参数”，如果中间有空格，不能解析为多个参数。比如在命令行输入 xcopy /s "C:\program files" "d:\t" 时，参数仍然是4个，第3个参数应该是字符串C:\program files，而不是C:\program，注意输出参数时，需要将“去掉，引号不存在嵌套情况。
- 参数不定长
- 输入由用例保证，不会出现不符合要求的输入

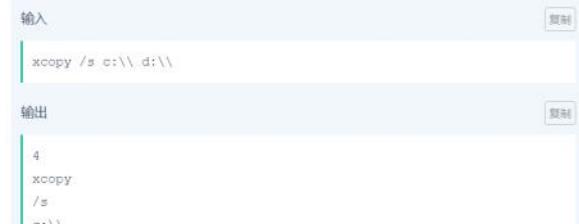
输入描述:

输入一行字符串，可以有空格

输出描述:

输出参数个数，分解后的参数，每个参数都独占一行

示例1



输入
xcopy /s c:\ d:\

输出
4
xcopy
/s
c:\
d:\

```
int main(){
    string str;
    while(getline(cin,str)){
        bool flag = false;
        string res ="";
        vector<string> vec;
        for(int i = 0; i < str.size(); i++){
            if(flag){
                if(str[i] != '\\')
                    res += str[i];
                else
                    flag = false;
            }
            else{
                if(str[i] == '\\')
                    flag = true;
                else if(str[i] == ' ')
                    vec.push_back(res);
                res = "";
            }
            else
                res += str[i];
        }
        vec.push_back(res);
        cout << vec.size() << endl;
        for(auto x : vec)
            cout << x << endl;
    }
}
```

题目描述

给定两个只包含小写字母的字符串，计算两个字符串的最大公共子串的长度。
注：子串的定义指一个字符串删除掉其部分前缀和后缀（也可以删）后形成的字符串。

输入描述:

输入两个只包含小写字母的字符串

输出描述:

输出一个整数，代表最大公共子串的长度

示例1

```
输入  
asdfas  
werasdfawer  
  
输出  
6
```

```
int main(){  
    string str1, str2;  
    while(cin >> str1 >> str2){  
        vector<vector<int>> dp(str1.size() + 1, vector<int>(str2.size() + 1, 0));  
        int max = 0;  
        for(int i = 1; i <= str1.size(); ++i){  
            for(int j = 1; j <= str2.size(); ++j){  
                if(str1[i-1] == str2[j-1])  
                    dp[i][j] = dp[i-1][j-1] + 1;  
                if(dp[i][j] > max)  
                    max = dp[i][j];  
            }  
        }  
        cout << max << endl;  
    }  
    return 0;
```

用动态规划的方法求最大公共子串 $dp[i][j]$ 表示 $a[0...i]b[0...j]$ 的最大公共子串长度

题目描述

给定一个正整数N代表火车数量， $0 < N < 10$ ，接下来输入火车入站的序列，一共N辆火车，每辆火车以数字1-9编号，火车站只有一个方向进出，同时停靠在火车站的列车中，只有后进站的出站了，先进站的才能出站。

要求输出所有火车出站的方案，以字典序排序输出。

输入描述:

有多组测试用例，每一组第一行输入一个正整数N（0

输出描述:

输出以字典序从小到大排序的火车出站序列号，每个编号以空格隔开，每个输出序列换行，具体见sample。

```
bool IsOutNum(vector<int> &push, vector<int> &pop, int len){  
    if(push.empty() || pop.empty() || len < 0)  
        return false;  
    stack<int> Stack;  
    int j = 0;  
    for(int i = 0; i < len; i++){  
        Stack.push(push[i]);  
        while(!Stack.empty() && Stack.top() == pop[j]){  
            Stack.pop();  
            ++j;  
        }  
    }  
    return Stack.empty();  
}
```

先进栈序列入栈，while栈不为空且栈顶指针==出栈序列首元素 stack.pop() ;++j

```

int main() {
    int N;
    while(cin >> N) {
        vector<int> pushNum(N, 0);
        vector<int> popNum(N, 0);
        for(int i = 0; i < N; i++) {
            cin >> pushNum[i];
            popNum[i] = pushNum[i];
        }
        sort(popNum.begin(), popNum.end());
        do{
            if(IsOutNum(pushNum, popNum, N)) {
                for(int i = 0; i < N-1; i++)
                    cout << popNum[i] << ' ';
                cout << popNum[N-1] << endl;
            }
        }while(next_permutation(popNum.begin(), popNum.end()));
    }
    return 0;
}

```

输入描述:

输入两个字符串。第一个为短字符串，第二个为长字符串。两个字符串均由小写字母组成。

输出描述:

如果短字符串的所有字符均在长字符串中出现过，则输出true。否则输出false。

```

string str1, str2;
int main(){
    while(cin>>str1>>str2){
        bool flag = true;
        for(auto c : str1){
            if(str2.find(c) == -1)
                flag = false;
        }
        if(flag)
            cout << "true" << endl;
        else
            cout << "false" << endl;
    }
}

```

s.find找到返回 0 没有返回-1

题目描述

找出给定字符串中大写字母(即'A'-'Z')的个数。

输入描述:

本题含有多组样例输入

对于每组样例，输入一行，代表待统计的字符串

输出描述:

对于每组样例，输出一个整数，代表字符串中大写字母的个数

```

3 string str;
4 int main(){
5     while(cin>>str){
6         //while(getline(cin,str)){
7         int cnt = 0;
8         for(auto c : str){
9             if(c>='A' && c<='Z')
10                 cnt++;
11         }
12         cout << cnt << endl;
13     }
14     return 0;
15 }

```

由 保存并提交 提交记录 自测输入 代码调试
10/11 组用例通过 运行时间 6ms

示例输入	A B cccD
预期输出	3
实际输出	1 1 1

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 string str;
4 int main() {
5     //while(cin>>str){
6     while(getline(cin, str)){
7         int cnt = 0;
8         for(auto c : str){
9             if(c>='A' && c<='Z')
10                 cnt++;
11         }
12         cout << cnt << endl;
13     }
14     return 0;
15 }
```

题目描述

给定一个仅包含小写字母的字符串，求它的最长回文子串的长度。

所谓回文串，指左右对称的字符串。

所谓子串，指一个字符串删掉其部分前缀和后缀（也可以不删）的字符串
(注意：记得加上while处理多个测试用例)

输入描述:

输入一个仅包含小写字母的字符串

输出描述:

返回最长回文子串的长度

```
int main(){
    string s;
    while(cin>>s) {
        int n = s.size(), Max = 0;
        string t = s;
        reverse(t.begin(), t.end());
        vector<vector<int>> dp(n+1, vector<int>(n+1, 0));
        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= n; j++) {
                if(s[i-1] == t[j-1])
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                Max = max(Max, dp[i][j]);
            }
        }
        cout << Max << endl;
    }
    return 0;
}
```

题目描述

求一个byte数字对应的二进制数字中1的最大连续数，例如3的二进制为00000011，最大连续2个1

本题含有多组样例输入。

输入描述:

输入一个byte数字

输出描述:

输出转成二进制之后连续1的个数

```
int main(){
    int n;
    while(cin>>n) {
        int k = 0;
        for(k = 0; n!=0; k++)
            n = n&(n>>1);
        cout << k << endl;
    }
}
```

题目描述

现在IPV4下用一个32位无符号整数来表示，一般用点分方式来显示，点将IP地址分成4个部分，每个部分为8位，表示成一个无符号整数（因此不需要用正号出现），如10.137.17.1，是我们非常熟悉的IP地址，一个IP地址串中没有空格出现（因为要表示成一个32数字）。
现在需要你用程序来判断IP是否合法。

注意本题有多组样例输入。

输入描述:

输入一个ip地址，保证是xx.xx.xx.xx的形式（xx为整数）

输出描述:

返回判断的结果YES or NO

```
int main(){
    int ip[4] = {0};
    char ch[3];
    while(cin>>ip[0]>>ch[0]>>ip[1] >> ch[1]>>ip[2]>>ch[2]>>ip[3]){
        if(ch[0] == '.' && ch[1] == '.' && ch[2] == '.'){
            if((ip[0] >= 0 && ip[0] <= 255) && (ip[1] >= 0 && ip[1] <= 255) && (ip[2]
                cout<<"YES"<<endl;
            else
                cout<<"NO"<<endl;
        }
        else
            cout<<"NO"<<endl;
    }
    return 0;
```

题目描述

功能:等差数列 2 , 5 , 8 , 11 , 14。 . . 。

输入:正整数N >0

输出:求等差数列前N项和

本题为多组输入，请使用while(cin>>)等形式读取数据

输入描述:

输入一个正整数。

输出描述:

输出一个相加后的整数。

```
int main(){
    int N;
    while(cin >> N){
        int temp = 2;
        int sum = 0;
        for(int i = 1; i <= N; i++){
            sum += temp;
            temp += 3;
        }
        cout << sum << endl;
    }
}
```

题目描述

输入整型数组和排序标识，对其元素按照升序或降序进行排序（一组测试用例可能会有多组数据）
本题有多组输入，请使用while(cin>>)处理

输入描述:

第一行输入数组元素个数

第二行输入待排序的数组，每个数用空格隔开

第三行输入一个整数0或1。0代表升序排序，1代表降序排序

```

bool isbig(const int a, const int b){
    return a > b;
}
int main() {
    int n;
    while(cin >> n) {
        vector<int> v;
        int m;
        for(int i = 0; i < n; i++) {
            cin >> m;
            v.push_back(m);
        }
        int k;
        cin >> k;
        if(k == 0)
            sort(v.begin(), v.end());
        else if(k == 1)
            sort(v.begin(), v.end(), isbig);
        for(int i = 0; i < v.size(); i++)
            cout << v[i] << ' ';
        cout << endl;
    }
}

```

输入描述:

一个只包含小写英文字母和数字的字符串。

输出描述:

一个字符串，为不同字母出现次数的降序表示。若出现次数相同，则按ASCII码的升序输出。

示例1

输入	<pre>aaddccdc 1b1bbbbbbbbb</pre>	复制
输出	<pre>cda b1</pre>	复制

```

bool comp1(const pair<char,int> &a, const pair<char,int> &b) {
    return a.second > b.second;
}

int main() {
    string str;
    while(cin >> str) {
        map<char,int> mp;
        for(auto c : str) {
            mp[c]++;
        }
        vector<pair<char,int>> vec(mp.begin(), mp.end());
        stable_sort(vec.begin(), vec.end(), comp1);
        for(auto x : vec)
            cout << x.first;
        cout << endl;
    }
    return 0;
}

```

可以使用stable_sort(vec.begin(), vec.end(), cmp)保证比较项相等时的稳定性

快排不是稳定的 如果次数相等 需要定义按第一项升序 < 不相等就是按第二项升序 >

```

bool comp1(const pair<char,int> &a, const pair<char,int> &b) {
    if (a.second == b.second)
        return a.first < b.first;
    return a.second > b.second;
}

```

题目描述

将一个字符串str的内容颠倒过来，并输出。str的长度不超过100个字符。

输入描述:

输入一个字符串，可以有空格

输出描述:

输出逆序的字符串

```
int main() {
    string str;
    while(getline(cin,str)) {
        reverse(str.begin(),str.end());
        cout << str << endl;
    }
}
```

题目描述

正整数A和正整数B 的最小公倍数是指能被A和B整除的最小的正整数值，设计一个算法，求输入A和B的最小公倍数。

输入描述:

输入两个正整数a和b。

输出描述:

输出a和b的最小公倍数。

```
int main() {
    int m, n;
    while(cin >>m>>n) {
        int res = 0;
        for(int i=1; i++; i++) {
            if(i%m == 0 && i%n == 0) {
                res = i;
                break;
            }
        }
        cout << res;
    }
}
```

缓存区留着\n 要用cin.ignore()

getline()之前存在cin时的使用

原创 2205 2016-02-26 17:21:33 2310 收藏 1
分类专栏: C++

版权

cin.getline()用来读取一行数据，但是当cin.getline()前面进行了cin输入的话，cin.getline()会把进行cin输入时行末丢弃的换行符读入，从而造成cin.getline()第一次获得的数据为一空行，如下所示：

```
int main()
{
    int rep;
    cin >> rep;
    for (int i=0; i<rep; i++)
    {
        cin.getline(str,25);
    }
}
此时cin.getline()所读入的第一行是空行，并且占据一次读入次数，造成只能再输入rep-1次数据。
解决办法
    cin.ignore();
```

我们知道，数组的第一个元素下标是 0。数组的范围也就是从 array[0] 到 array[size - 1]。但实际上C/C++支持正负下标。负下标必须在数组边界内；否则结果不可预知。以下代码显示了正数组和负数组下标：

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int intArray[1024];
6     for (int i = 0, j = 0; i < 1024; i++)
7     {
8         intArray[i] = j++;
9     }
10
11    cout << intArray[512] << endl; // 512
12
13    int *midArray = &intArray[512]; // 指向了数组中间的数据
14
15    cout << midArray[-256] << endl; // 256
16
17    cout << intArray[-256] << endl; // 得到不可预知的结果
...
```

今天偶然碰到C/C++的数组下标可以为负值，感到十分奇怪。平时写代码，下标都是从零开始，从来没考虑到这个问题。写了一下测试代码，居然通过了。但是值却是乱的。但是如果你写下标的值在数组的范围内，是可以输出一个正确的值的。

```
1 int a[5] = {1,2,3,4,5};  
2 int *ptr1 = (int*)(&a+1);  
3  
4 printf("%x,%x,",ptr1[-1]);
```

这个结果就是为5；

后来上网查了一下，也思考了一下，觉得其实挺正常的。数组的下标其实就是数组的头指针在移动，而在C/C++中，编译器是没有对数组进行越界检查的。例如上面的，你输出a[6]也是没错的，只是值是乱的而已。

这个特性其实在开发中几乎用不到，原因就是C/C++的编译器懒。在Java或者OC中，都是报错的。

算法知识竞赛题解

校招时部分企业笔试将禁止编程题跳出页面，为提前适应，练习时请使用在线自测，而非本地IDE。

题目描述

Catcher是MCA国的情报员，他工作时发现敌国会用一些对称的密码进行通信，比如像这些ABA , ABA , A , 123321，但是他们有时会在开始或结束时加入一些无关的字符以防止别国破解。比如进行下列变化 ABBA->12ABBA,ABA->ABAKK,123321->51233214 。因为截获的串太长了，而且存在多种可能的情况（abaaab可看作是aba,或baab的加密形式），Cather的工作量实在是太大了，他只能向电脑高手求助，你能帮Catcher找出最长的有效密码串吗？

本题含有多组样例输入。

输入描述:

输入一个字符串

输出描述:

返回有效密码串的最大长度

示例1

```
1 #include<bits/stdc++.h>  
2 using namespace std;  
3 string s;  
4 int main(){  
5     while(getline(cin,s)){  
6         vector<vector<int>> dp(s.size(), vector<int>(s.size(), 0));  
7         int maxlen = 0;  
8         int start = 0;  
9         for(int i = s.size() - 1; i >= 0; i--) { //从右往左  
10             for(int j = i; j < s.size(); j++) { //也是从j开始往右  
11                 if(s[i] == s[j]) {  
12                     if(j - i <= 1) { //情况一 和 情况二  
13                         dp[i][j] = true;  
14                     } else if (dp[i + 1][j - 1]) { //情况三  
15                         dp[i][j] = true;  
16                     }  
17                 }  
18                 if(dp[i][j] && j - i + 1 > maxlen){  
19                     maxlen = j - i + 1;  
20                     start = i;  
21                 }  
22             }  
23         }  
24         cout << maxlen << endl;  
25     }  
26 }
```

```
1 class Solution {  
2 public:  
3     int getLongestPalindrome(string A, int n) {  
4         // write code here  
5         int maxn = 0; int flag = 0;  
6         for(int i = 0; i <n; i++){  
7             for(int j = n; j >i; j--){  
8                 flag=0;  
9                 int l, r;  
10                for(l = i, r=j; l <=(i+j)/2; l++, r--){  
11                    if(A[l] != A[r]){  
12                        flag = 1;  
13                        break;  
14                    }  
15                }  
16                if(flag == 0){  
17                    maxn = max(maxn, (j-i+1));  
18                    break;  
19                }  
20            }  
21        }  
22        return maxn;  
23    }
```

迷宫问题

```

#include<bits/stdc++.h>
using namespace std;
vector<vector<int>> maze;
vector<pair<int,int>> tmp;
vector<pair<int,int>> best;
int N, M;
void MazeTrack(int i, int j){
    ... maze[i][j] = 1;
    ... tmp.push_back(make_pair(i,j));
    ... if(i == N-1 && j == M-1)
    ..... if(best.empty() || tmp.size() < best.size())
    ..... best = tmp;
    ... if(i - 1 >= 0 && maze[i - 1][j]==0)
    ..... MazeTrack(i-1, j);
    ... if(i + 1 <= N-1 && maze[i + 1][j]==0)
    ..... MazeTrack(i + 1,j);
    ... if(j - 1 >= 0 && maze[i][j - 1]==0)
    ..... MazeTrack(i, j - 1);
    ... if(j + 1 <= M-1 && maze[i][j + 1]==0)
    ..... MazeTrack(i, j + 1);
    ... maze[i][j] = 0;
    ... tmp.pop_back();
}

```

```

... while(cin >> N >>M) {
...     tmp.clear();
...     best .clear();
...     // vector<vector<int>> maze(N, vector<int>(M, 0)); //变量不允许声明两次·这是声明并定义！！
...     maze = vector<vector<int>>(N, vector<int>(M, 0)); //将这样一个temp矩阵赋值给maze
...     for(int i = 0; i <N; i++) { ... //用一个两层for就可以了·不需要再加while (N--)
...         for(int j = 0; j <M; j++)
...             cin >> maze[i][j];
...
...     }
...     MazeTrack(0,0);
...     for(auto x : best)
...         cout << ' (' << x.first << ',' << x.second << ') ' << endl;
}

```

本题是顺序建立链表，是学习链表的基础。链表的建立无非就是利用好头尾结点来进行新数据的储存，并且建立一个P结点作为游动指针来进行游动的储存

```

#include<bits/stdc++.h>
using namespace std;
typedef struct node
{
    int data;
    struct node*next;
}tree[1100];
int main()
{
    struct node*head,*tail,*p;
    head=new tree;
    head->next=NULL;
    tail=head;
    int n;
    scanf("%d",&n);
    for(int i=0;i<n;i++)
    {
        p=new tree;
        scanf("%d",&p->data);
        p->next=NULL;
        tail->next=p;
        tail=p;
    }
    p=head->next;
}

```

```

while(p)
{
    if(p->next==NULL)
    {
        printf("%d\n",p->data);
    }
    else
    {
        printf(" %d ",p->data);
    }
    p=p->next;
}

```

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 32M，其他语言64M 热度指数：54945

本题知识点：字符串

算法知识视频讲解

校招时部分企业笔试将禁止编程题跳出页面，为提前适应，练习时请使用在线自测，而非本地IDE。

题目描述

在命令行输入如下命令：

`xcopy /s c:\ d\`

各个参数如下：

参数1：命令字`xcopy`

参数2：字符串`s`

参数3：字符串`c:\`

参数4：字符串`d:\`

请编写一个参数解析程序，实现将命令行各个参数解析出来。

解析规则：

1.参数分隔符为空格

2.对于“包含起来的参数”，如果中间有空格，不能解析为多个参数。比如在命令行输入`xcopy /s "C:\program files" "d:\"`时，参数仍然是4个，第3个参数应该是字符串`C:\program files`，而不是`C:\program`，注意输出参数时，需要将“去掉，引号不存在嵌套情况。

3.参数不定长

4.输入由用例保证，不会出现不符合要求的输入。

输入描述：

输入一行字符串，可以有空格

//繁衍过程 每一次迭代表示一次繁衍

`n=3 1-2-3` 两次繁衍经历 循环条件`while(--n)`

`n=3` 循环两次 先减一再进入循环 对于`while(--n)`

数组名可以直接当成`sort`函数的指针使用

进行排序

`sort`函数使用迭代器

`reverse`函数也使用迭代器

容器类型的函数使用

默认是从小到大进行排序

注意声明的全局变量`maze`与路径`vector`

在`main()`中定义`N*M`的`maze`；`void MazeTrack`函数

可以直接使用`maze`与`path_temp/best` `vector`

开始时让每走过的一格都为1，

并将此格存入`vector` 注意使用`make_pair`

函数，将`i, j`转为`pair`类型再`push_back`到

`vector path_temp`；

判断是否`i, j`到达格子右下角如果是则看`path_best`是否

`empty` || `path_temp.size() < path_best.size()`，

存储最佳路径`path_best=path_temp`；

接下来再对四个方向进行

判断，条件是边界加格子是否为0通路，如果是，

则可以走到这一格，依次递归到右下角格

这些if语句/递归语句结束 再将`maze[i][j]=0`

依次退出递归返回时将格子还原为0，同时将`temp`中的

格子/步数递归返回退出一格

---递归回溯方法

`insert`是在这个位置之前插入元素

`erase`是直接抹掉这个位置的元素

用`vector`的`find`函数与`insert`函数 `rease`函数

实现链表元素的插入与删除

注意链表的定义`struct ListNode{int data; ListNode* next;};`

链表指针的定义`ListNode *p, *head;`

new的使用`head = new ListNode;`

链表结点赋值的方法`p->data = m;`

倒序构造链表的方法 每输入一个`cin>>m;` `p = new ListNode;`

<上一步 | 下一步>

① C++(clang++11) ▾

ACM模式

```

4 int main() {
5     while(getline(cin,s)){
6         vector<string> vec;
7         auto it = s.find_first_not_of(' ');
8         string res;
9         for(int i =it; i<s.size(); i++){
10             if(s[i]!=' '){
11                 if(s[i] == '\"'){
12                     auto next = s.find('\"',i+1);
13                     res = s.substr(i+1, next-i-1);
14                     vec.push_back(res);
15                     res.clear();
16                     i = next+1;
17                 }
18                 else
19                     res += s[i];
20             }
21             else{
22                 if(!res.empty()){
23                     vec.push_back(res);
24                     res.clear();
25                 }
26             }
27         }
28         if(res.size() >0)
29             vec.push_back(res);
30         cout<<vec.size()<<endl;
31         for(int i =0; i<vec.size(); i++)

```

```
p->data = m;
p->next = head->next;
head->next = p;
```

注意头结点是没有值的， p = head;
while(k--) p = p->next; 如p = 4 , 执行完4次循环正好指向5那个结点
使用p->data 输出结果

你看输入的格式是第一先有两个
如果写成while(cin >>n) 这样就错了
那么 5 , 2 都会赋给 n , 每次循环赋值一次
如果写成while(cin >>n >>k) 这样才是正确的
那么5 , 2会分别赋给 n , k ; 连续读取的写法

再就是这种输出在一行用空格隔开的输出
最后一个元素往往要单独写再加一个endl;
to_string(n);

将输入的以空格分隔的一行
通过stringstream>> 每个单词存放到
不同的str1 str2 判断str2.empty() 为空返回1

将已有的a[i]读入stringstream ss
从ss中再取时 每次/每一个>> 符号调用只能
取到一个单词 两个>>两个单词分别取到tmp1 tmp2
注意这里if语句括号的使用
不加就会报错 下面的c要用这个
在哪个a[i]中匹配到/find到的str
string的find 匹配成功返回0 否则返回npos/-1

注意自定义的comp谓词传的参数是引用
参数类型是vector的元素类型
这里谓词的定义就是一个return语句
元素比较语句 也可以添加其他成分

先输入第一行的while(cin >>n){
里面再输入第二行 cin >>m
}
定义vector的元素为vector<pair<string,int>> res (n)
初始化为n个pair对
采用for循环赋值 cin >> res[i].first >> res[i].second
自动以空格分隔分别赋值到 first与second
采用的是stable_sort函数 进行排序
每一行格式化输出 注意中间的一个空格与每一行的endl操作符

if语句里面别写出赋值符号，这种错误很低级，
也很难以发现！！！

这里如果不use else if 这个 if会与下面的那个else结合 构成一个if-else语句
如果用了else if 下面的那个else就会否定上面所有的if 结果就不会出现 “

这类遍历到尽头的循环，它只有在最后位置有空格出现时会进行push_back,有时直到
循环结束都不会遇到空格，res中仍存有最末尾的一个str ，因此在for循环结束时应该
再加一处vec.push_back(res) 让最末尾的str也放到vec vec.size()得到容器内元素个数
再输出容器内元素内容

判断是否为出栈序列
定义一个Stack，将输入vector中的元素依次压入 Stack，每次压入判断当前栈是否为空，且栈顶元素是否等
于出栈序列首元素，如果是，则当前栈顶元素出栈，出栈序列索引加1，再进行判断，当前栈顶元素与出栈序
列首元素是否相等，相等则再出栈，每次将入栈序列压入栈时进行判断，序列压完，for循环执行完，里面的
元素应该也已经全部出栈，如果执行完for循环，Stack为空则这个出栈序列符合入栈序列的要求

cin > vec[i]可以直接对vec元素进行赋值
这里使用了sort()函数与next_permutation()对vector元素进行全排列
do{ is out(), cout << ...} while(next_permutation);

如果输入中有空格，用while(cin > str)就是这种效果，表示3个实例输入

这里应该用while(getline(cin, str)) 输入一整行表示1个实例输入

```
for (k=0; n!= 0; k++)  
    n=n&(n>>1)
```

得到连续出现的1的次数

这个k必须是在for循环外面进行定义 否则不可以使用k 不能将k写到cout

for循环体内为局部变量 要使用这个计数变量必须在for外面进行定义

vector可以直接用一对迭代器指定范围

对vector容器进行初始赋值

mp中的元素不会有重复 且关键字会按照字典序进行排序

使用的是stable_sort函数 次数一样的对应关键字不动它

```
vector<pair<char,int>> vec(mp.begin(),mp.end());
```

```
stable_sort(vec.begin(),vec.end());
```

刷题总结2

2020年6月4日 12:47

剑指 Offer 03. 数组中重复的数字

难度 简单 ⚡ 368 ☆ 🔍 🌐 📁

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0 ~ n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

示例 1：

输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3

```
class Solution {
public:
    int findRepeatNumber(vector<int>& nums) {
        unordered_map<int,int> mp;
        for(auto x : nums){
            ++mp[x];
            if(mp[x] > 1)
                return x;
        }
        return 0;
    }
}
```

剑指 Offer 04. 二维数组中的查找

难度 中等 ⚡ 285 ☆ 🔍 🌐 📁

在一个 $n * m$ 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个高效的函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

示例：

现有矩阵 matrix 如下：

```
[  
    [1, 4, 7, 11, 15],  
    [2, 5, 8, 12, 19],  
    [3, 6, 9, 16, 22],  
    [10, 13, 14, 17, 24],  
    [18, 21, 23, 26, 30]  
]
```

给定 target = 5，返回 true。

给定 target = 20，返回 false。

```
class Solution {
public:
    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {
        int i = matrix.size() - 1, j = 0;
        while(i >= 0 && j < matrix[0].size()){
            if(matrix[i][j] > target) --i;
            else if(matrix[i][j] < target) ++j;
            else return true;
        }
        return false;
    }
};
```

剑指 Offer 05. 替换空格

难度 简单 ⚡ 93 ☆ 🔍 🌐 📁

请实现一个函数，把字符串 s 中的每个空格替换成 "%20"。

示例 1：

输入：s = "We are happy."
输出："We%20are%20happy."

```

class Solution {
public:
    string replaceSpace(string s) {
        int count = 0, len = s.size();
        for(char c : s)
            //if(c == ' ')
            if(c == ' ')           //注意赋值是右边赋给左边 j=2是新串增加了两个字符 让i j同时都指向一个表示空格的格位
                count++;
        s.resize(len + 2 * count);
        for(int i = len - 1, j = s.size() - 1; i < j; i--, j--) {
            if(s[i] != ' ')
                s[j] = s[i];
            else {
                s[j] = '0';
                s[j-1] = '2';
                s[j-2] = '%';
                j -= 2;
            }
        }
        return s;
    }
};

```

剑指 Offer 06. 从尾到头打印链表

难度 简单 128 ★★★

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

输入：head = [1,3,2]
输出：[2,3,1]

```

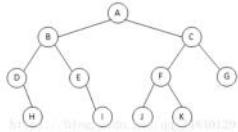
class Solution {
public:
    vector<int> reversePrint(ListNode* head) {
        vector<int> res;
        stack<int> s;

        ListNode *p = head;           //利用栈实现
        while(p){                     //创建一个指针指向传入链表的头结点
            s.push(p->val);         //while(p)
            p = p->next;             //stack<int> s;
        }                            //s.push(p->val);
        while(!s.empty()){           //p = p->next;
            res.push_back(s.top());   //while(!s.empty())
            s.pop();                  //res.push_back(s.top())
        }                            //s.pop();
        return res;
    }
};

```

(1) 先(根)序遍历 (根左右)
(2) 中(根)序遍历 (左根右)
(3) 后(根)序遍历 (左右根)

举个例子：



先(根)序遍历 (根左右) : A B D H E I C F J K G

中(根)序遍历 (左根右) : D H B E I A J F K C G

后(根)序遍历 (左右根) : H D I E B J K F G C A

以后(根)序遍历为例，每次都是先遍历左子树，然后再遍历右子树，最后再遍历根节点，以此类推，直至遍历完整棵树。

此外，还有一个命题：给定二叉树的任何一种遍历序列，都无法唯一确定相应的二叉树。但是如果知道了二叉树的中序遍历序列和左子树的另一种遍历序列，就可以唯一地确定二叉树。

对于任意一颗树而言，前序遍历的形式总是

[根节点, [左子树的前序遍历结果], [右子树的前序遍历结果]]

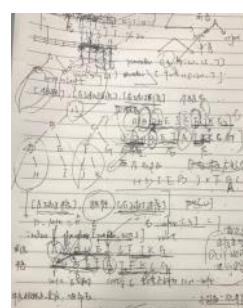
即根节点总是前序遍历中的第一个节点。而中序遍历的形式总是

[[左子树的中序遍历结果], 根节点, [右子树的中序遍历结果]]

只要我们在中序遍历中定位到根节点，那么我们就可以分别知道左子树和右子树中的节点数目。由于同一颗子树的前序遍历和中序遍历的长度显然是相同的，因此我们就可以对应到前序遍历的结果中，对上述形式中的所有**左右括号**进行定位。

这样以来，我们就知道了左子树的前序遍历和中序遍历结果，以及右子树的前序遍历和中序遍历结果，我们就可以递归地构造出左子树和右子树，再将这两颗子树接到根节点的左右位置。

细节

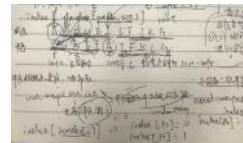


的所有**左右括号**进行定位。

这样以来，我们就知道了左子树的前序遍历和中序遍历结果，以及右子树的前序遍历和中序遍历结果，我们就可以递归地构造出左子树和右子树，再将这两颗子树接到根节点的左右位置。

细节

在中序遍历中对根节点进行定位时，一种简单的方法是直接扫描整个中序遍历的结果并找出根节点，但这样做的时间复杂度较高。我们可以考虑使用哈希表来帮助我们快速地定位根节点。对于哈希映射中的每个键值对，键表示一个元素（节点的值），值表示其在中序遍历中的出现位置。在构造二叉树的过程之前，我们可以对中序遍历的列表进行一遍扫描，就可以构造出这个哈希映射。在此后构造二叉树的过程中，我们就只需要 $O(1)$ 的时间对根节点进行定位了。



```
class Solution {
private:
    | unordered_map<int,int> index;
public:
    TreeNode* mybuildTree(const vector<int> &preorder, const vector<int> &inorder, int p_left, int p_right,
i_left, int i_right){
        if(p_left > p_right){
            return nullptr;
        }
        int p_root = p_left;
        int i_root = index[preorder[p_left]];
        TreeNode *root = new TreeNode(preorder[p_root]);
        int size_s = i_root - i_left;
        root -> left = mybuildTree(preorder, inorder, p_left+1, p_left + size_s, i_left, i_root -1);
        root -> right = mybuildTree(preorder,inorder,p_left+size_s+1,p_right,i_root+1,i_right);
        return root;
    }
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        int n = preorder.size();
        for(int i = 0; i < n; ++i){
            index[inorder[i]] = i;
        }
        return mybuildTree(preorder, inorder, 0, n-1, 0, n-1);
    }
};
```

剑指 Offer 09. 用两个栈实现队列

难度 简单 213 ☆ ⌂ ⌃ ⌁ ⌂

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead，分别完成在队列尾部插入整数和在队列头部删除整数的功能。（若队列中没有元素，deleteHead 操作返回 -1）

```
class CQueue {
| stack<int> stack1, stack2;
public:
    CQueue() {
        while(!stack1.empty())
            stack1.pop();
        while(!stack2.empty())
            stack2.pop();
    }

    void appendTail(int value) {
        stack1.push(value);
    }

    int deleteHead() {
        if(stack2.empty()){
            while(!stack1.empty()){
                stack2.push(stack1.top());
                stack1.pop();
            }
        }
        if(stack2.empty()) //如果队列中没有元素,stack2经push仍然为empty(),return -1
            return -1;
        else{
            int del = stack2.top();
            stack2.pop();
            return del;
        }
    }
};
```

剑指 Offer 10-I. 菲波那契数列

难度 简单 126 ☆ ⌂ ⌃ ⌁ ⌂

写一个函数，输入 n ，求斐波那契（Fibonacci）数列的第 n 项（即 $F(N)$ ）。

斐波那契数列的定义如下：

$$F(0) = 0, \quad F(1) = 1 \\ F(N) = F(N - 1) + F(N - 2), \text{ 其中 } N > 1.$$

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

答案需要取模 $1e9+7$ (1000000007)，如计算初始结果为：1000000008，请返回 1。

```

class Solution {
public:
    int fib(int n) {
        vector<int> dp;
        for(int i = 0; i <= n; i++){
            if(i == 0)
                dp.push_back(0);
            else if(i ==1)
                dp.push_back(1);
            else
                dp.push_back((dp[i-1] + dp[i-2])%1000000007);      //一维vector的动态规划方法
        }                                              //最终的dp[n]就是要求的结果
        return dp[n];
    }
}

```

剑指 Offer 12. 矩阵中的路径

难度 中等 290 ★ ⚡ ⚡ ⚡ ⚡

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再进入该格子。例如，在下面的 3×4 的矩阵中包含一条字符串“bfce”的路径（路径中的字母用加粗标出）。

```

[["a","b","c","e"],
["s","f","c","s"],
["a","d","e","e"]]

```

但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

```

public:
    bool exist(vector<vector<char>>& board, string word) {
        rows=board.size();
        cols=board[0].size();
        for(int i = 0; i < rows; i++){
            for(int j = 0 ; j < cols; j++){
                if(dfs(board,word,i,j,0)) return true;
            }
        }
        return false;
    }
private:
    int rows, cols;
    bool dfs(vector<vector<char>> &board, string word, int i, int j, int k){
        if(i >= rows || i < 0 || j >= cols || j < 0 || board[i][j] != word[k]) return false;
        // if(i >= rows || i < 0 || j >= cols || j < 0 || board[i][j] != word[k]) return false;
        if(k == word.size()-1) return true;
        board[i][j] = '\0';
        bool res=dfs(board,word, i +1, j , k+1) || dfs(board,word, i - 1, j , k+1) || dfs(board,word,i, j-1,k+1)||dfs(board,word,i,j+1,k+1);
        board[i][j] = word[k];
        return res;
    }
};

```

类中private中定义的数据public中的函数也可以使用，但是必须定义的是类中的全局变量，不能在函数中定义的
这个board必有传引用 &，否则会引起超时
dfs(board,word,i,j,0)调用时，避免实参到形参的copy //形参定义的是引用类型

剑指 Offer 13. 机器人的运动范围

难度 中等 260 ★ ⚡ ⚡ ⚡ ⚡

地上有一个 $m \times n$ 的方格，从坐标 $[0,0]$ 到坐标 $[m-1,n-1]$ 。一个机器人从坐标 $[0, 0]$ 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于 k 的格子。例如，当 k 为18时，机器人能够进入方格 [35, 37]，因为 $3+5+3+7=18$ 。但它不能进入方格 [35, 38]，因为 $3+5+3+8=19$ 。请问该机器人能够到达多少个格子？

```

class Solution {
public:
    int get(int x){
        int res = 0;
        while(x>0){
            res+= (x%10);
            x/=10;
        }
        return res;
    }
    int movingCount(int m, int n, int k) {
        vector<vector<int>> dp (m, vector<int>(n,0));      //初始的dp[0][0]=1 表示可达
        int sum = 0;
        dp[0][0] = 1;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if((get(i) + get(j))>k) continue;
                if(j - 1 >= 0) dp[i][j] = dp[i-1][j] | dp[i][j];
                if(j + 1 >= 0) dp[i][j] = dp[i][j+1] | dp[i][j];
                if(i - 1 >= 0) dp[i][j] = dp[i-1][j] | dp[i][j];
                if(i + 1 >= 0) dp[i][j] = dp[i+1][j] | dp[i][j];
                sum +=dp[i][j];
            }
        }
        return sum;
    }
}

```

剑指 Offer 14- I. 剪绳子

难度 中等 206 收起 回复 分享 举报

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（ m, n 都是整数， $n > 1$ 并且 $m > 1$ ），每段绳子的长度记为 $k[0], k[1] \dots k[m-1]$ 。请问 $k[0]*k[1]*\dots*k[m-1]$ 可能的最大乘积是多少？例如，当绳子的长度是 8 时，我们把它剪成长度分别为 2、3、3 的三段，此时得到的最大乘积是 18。

示例 1：

```
class Solution {
public:
    int cuttingRope(int n) {
        if(n<=3)
            return n-1;
        vector<int> res(n+1,0);
        res[1] = 1;
        res[2] = 2;
        res[3] = 3;
        int maxp = 0;
        int temp;
        for(int i =4; i <= n; i++)
            for(int j =0; j <= i/2; j++)
                temp = res[j]*res[i - j];
                maxp = max(maxp,temp);
                res[i] = maxp;
        }
        return res[n];
    }
};
```

动态规划的核心是要用一个容器将前面计算过的数据存起来，每次下面计算时都会用到前面的计算值 dp思想

剑指 Offer 16. 数值的整数次方

难度 中等 141 收起 回复 分享 举报

实现 $\text{pow}(x, n)$ ，即计算 x 的 n 次幂函数（即， x^n ）。不得使用库函数，同时不需要考虑大数问题。

```
class Solution {
public:
    double myPow(double x, int n) {
        if(x == 1 || n == 0) return 1;
        double ans = 1;
        long num = n;
        if(n < 0){
            x = 1/x;
            num = -num;
        }
        while(num >0){
            if(num &1 == 1){
                ans *= x;
            }
            x *=x;
            num >>= 1;
        }
        return ans;
    }
};
```

注意这里是 num>>= 1;
不是Num>>1; !!!写完整的赋值表达式

求问为什么要用 long num = n; 而不用int

△2 收起 回复 收起回复 △1 回复 △ 分享 △ 举报

Philos 🔍 15

因为如果 $n = -2^{31}$ ，那么后面 $num = -num = 2^{31}$ ，用 int 的话会溢出。

我们这里使用快速幂进行求解。我们看一下 n 的二进制形式一定是若干个 1 和 0 构成，比如 $9 = 1001$

$= 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$

$$x^9 = x^{2^0} * x^{2^1*0} * x^{2^2*0} * x^{2^3}$$

所以我们可以看出来，每次乘的值都是前一个值的 2 倍，当 n 对应位为 0 时跳过

负数幂和正数幂相同，因为除以一个数就相当于乘这个数的倒数。

剑指 Offer 18. 删除链表的节点

难度 简单 113 收起 回复 分享 举报

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。

返回删除后的链表的头节点。

注意：此题对比原题有改动

```

/* struct ListNode {
*     int val;
*     ListNode *next;
*     ListNode(int x) : val(x), next(NULL) {}
* };
*/
class Solution {
public:
    ListNode* deleteNode(ListNode* head, int val) {
        if(head == NULL) return NULL;
        if(head->val == val) return head->next;
        ListNode *p = head;
        while( p!=NULL && p->next->val != val)
            p = p->next;
        if(p->next != NULL)
            p->next = p->next->next;
        return head;
    }
};

```

剑指 Offer 20. 表示数值的字符串

难度 中等 通过率 177 / 209

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100"、"5e2"、"-123"、"3.1416"、"-1E-16"、"0123"都表示数值，但"12e"、"1a3.14"、"1.2.3"、"+-5"及"12e+5.4"都不是。

```

bool isNumber(string s) {
    int i = s.find_first_not_of(' ');
    if (i == string::npos) return false;
    int j = s.find_last_not_of(' ');
    s = s.substr(i, j - i + 1);
    if (s.empty()) return false;
    int e = s.find('e');
    int E = s.find('E');
    int pos = -1;

    if( e!= string :: npos)
        pos = e;
    else if (E != string :: npos)
        pos = E;
    if (pos == -1)
        return judgeP(s);
    else
        return judgeP(s.substr(0, pos)) && judgeS(s.substr(pos + 1));
}

int judgeP(string s) {
    int doc =0; int num = 0;
    int n = s.size();
    for (int i = 0; i < n; i++) {
        if (s[i] == '+' || s[i] == '-')
            if (i != 0) return false;
        else if (s[i] == '.')
            doc++;
            if(doc>1) return false;
        else if (s[i] >= '0' && s[i] <= '9')
            num++;
        else
            return false;
    }
    return num;
}

int judgeS(string s)
{
    int num = 0;
    for (int i = 0; i < s.size(); ++i) {
        if (s[i] == '+' || s[i] == '-')
            if (i != 0)
                return false;
        else if (s[i] >= '0' && s[i] <= '9')
            num++;
        else
            return false;
    }
    return num;
}

```

剑指 Offer 21. 调整数组顺序使奇数位于偶数前面

难度 简单 通过率 107 / 209

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

```

class Solution {
public:
    vector<int> exchange(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while(left < right){
            if((nums[left] & 1) != 0) {
                ++left;
                continue;
            }
            if( (nums[right] & 1) != 1) {
                --right;
                continue;
            }
            swap(nums[left],nums[right]);
        }
        return nums;
    }
}

```

这个花括号不加，有时写着写着就忘记了，
特别是下面如果有两条语句！！！

剑指 Offer 24. 反转链表

难度 简单 208 收藏 10A 举报

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例：

输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL

```

class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(!head) return NULL;
        ListNode *cur = head;
        ListNode *pre = NULL;
        ListNode *tmp = NULL;
        while(cur){
            tmp=cur->next;
            cur->next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }
}

```

剑指 Offer 25. 合并两个排序的链表

难度 简单 104 收藏 10A 举报

输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。

示例1：

输入 : 1->2->4, 1->3->4
输出 : 1->1->2->3->4->4

```

class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *head =new ListNode;
        ListNode *ret =head;
        while(l1 != NULL && l2 != NULL){
            if(l1->val < l2->val){
                head->next = l1;
                l1 = l1->next;
            }
            else{
                head->next = l2;
                l2 = l2->next;
            }
            head = head->next;
        }
        head->next = l1==NULL ? l2: l1;
        return ret->next; //这个ret保存了最开始的那个新建的head,
                           //从head->next 开始是l1/l2的结点值 因此返回的是ret->next;
    }
};

```

剑指 Offer 26. 树的子结构

难度 中等 245 收藏 分享

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

B是A的子结构，即 A中有出现和B相同的结构和节点值。

例如：

给定的树 A:

```
    3
   / \
  4   5
 / \
1   2
```

给定的树 B :

```
    4
   /
  1
```

返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。

```
class Solution {
public:
    bool isSubStructure(TreeNode* A, TreeNode* B) {
        bool ret = false;
        if(A == NULL || B == NULL)
            return false;
        if(A->val == B->val)
            ret = recur(A,B);
        if(!ret)
            ret = isSubStructure(A->left, B);
        if(!ret)
            ret = isSubStructure(A->right,B);
        return ret;
    }

    bool recur(TreeNode *A, TreeNode *B){
        if(B == NULL)
            return true;
        if(A == NULL)
            return false;
        if(A->val != B->val)
            return false;
        return recur(A->left,B->left) && recur(A->right, B->right);
    }
};
```

先判断A与B的值是否相等，如果相等，递归比较结构 找到了就返回

如果未找到，继续在A中找与B根结点相等的结点 如果相等，进入递归

剑指 Offer 27. 二叉树的镜像

难度 简单 123 收藏 分享

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

例如输入：

```
    4
   / \
  2   7
 / \   / \
1   3   6   9
```

镜像输出：

```
    4
   / \
  7   2
 / \   / \
9   6   3   1
```

```
* Definition for a binary tree node.
* struct TreeNode {
*     int val;
*     TreeNode *left;
*     TreeNode *right;
*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        if(!root)
            return NULL;
        swap(root->left,root->right);
        mirrorTree(root->left);
        mirrorTree(root->right);
        return root;
    }
};
```

剑指 Offer 28. 对称的二叉树

难度 简单 157 ★★★☆☆

请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```
1
 / \
2   2
 / \ / \
3 4 4 3
```

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的：

```
1
 / \
2   2
 \   \
3   3
```

示例 1：

```
* Definition for a binary tree node.
* struct TreeNode {
*     int val;
*     TreeNode *left;
*     TreeNode *right;
*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if(!root)
            return true;
        return judge(root->left, root->right);
    }
    bool judge(TreeNode *l, TreeNode *r){
        if(l && !r)
            return false;
        if(!l || !r)
            return true;
        if(l->val != r ->val)
            return false;
        return judge(l->left, r->right) && judge(l->right,r->left);
    }
}
```

剑指 Offer 29. 顺时针打印矩阵

难度 简单 220 ★★★☆☆

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。

示例 1：

```
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[1,2,3,6,9,8,7,4,5]
```

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        // if(matrix.size() == 0 || matrix[0].size() == 0) return {};
        if(matrix.size() == 0)
            return {};
        vector<int> res;
        int top = 0, bottom = matrix.size()-1, left = 0, right = matrix[0].size()-1;

        while(1){
            for(int i = left; i <= right; i++)
                res.push_back(matrix[top][i]);
            top++;
            if(top > bottom) break;
            for(int i = top; i <= bottom; i++)
                res.push_back(matrix[i][right]);
            right--;
            if(right < left) break;
            for(int i = right; i >= left; i--)
                res.push_back(matrix[bottom][i]);
            bottom--;
            if(bottom < top) break;
            for(int i = bottom; i > top; i--)
                res.push_back(matrix[i][left]);
            left++;
            if(left > right) break;
        }
        return res;
    }
}
```

需要判断matrix是否为空，否则若传入一个空的matrix，会出现引用绑定到空对象运行时错误，程序无法正常返回。注意函数的返回类型为vector<int>，判断语句不能return一个-1，或者只写一个return，可以return一个空的{}列表，表示返回一个空元素列表。这里必须要有鲁棒性判断！！！

模拟矩阵打印的过程，注意边界条件的判断！

剑指 Offer 31. 栈的压入、弹出序列

难度 中等 通过率 158 / 200

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如，序列 {1,2,3,4,5} 是某栈的压栈序列，序列 {4,3,5,1,2} 是该压栈序列对应的一个弹出序列，但 {4,3,5,1,2} 就不可能是该压栈序列的弹出序列。

```
class Solution {
public:
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
        stack<int> stack;
        int i = 0;
        for(auto x : pushed){
            stack.push(x);
            while(!stack.empty() && stack.top() == popped[i]){
                stack.pop();
                i++;
            }
        }
        return stack.empty();
    }
};
```

对于先序遍历，根左右，每一棵子树都会看成一棵完整的树按根左右的顺序遍历

二叉树的深度优先搜索DFS：

从根结点出发，沿左子树纵向遍历，直到找到叶子结点为止，再回溯到前一个结点进行右子树的遍历直到遍历完所有可达结点为止 根左右 --跟先序遍历 的过程是一样的

如何实现DFS算法：栈

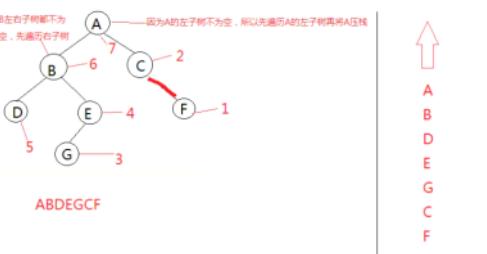
栈是一种先进后出的数据结构//后进先出

先将右子树压栈，再将左子树压栈，这样左子树就位于栈顶，可以保证先遍历左子树再遍历

右子树 出栈顺序就为遍历顺序

当我们在压栈时，必须确保该节点的左右子树都为空，如果不为空就需要先将它的右子树压

栈，再将左子树压栈。等左右子树都压栈之后，才将结点压栈。



二叉树的广度优先搜索BFS

从根结点出发，沿着树的宽度遍历树的结点，直到所有结点都遍历完为止

算法实现：队列 先进先出用来实现顺序遍历

根结点入队，再循环让左子树 右子树入队 出队即为遍历顺序

首先根节点先入队列

如果节点左右子节点，则将节点左右节点入队列



剑指 Offer 32 - I. 从上到下打印二叉树

难度 中等 通过率 76 / 200

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如：

给定二叉树: [3,9,20,null,null,15,7]，

```

* Definition for a binary tree node.
* struct TreeNode {
*     int val;
*     TreeNode *left;
*     TreeNode *right;
*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/
class Solution {
public:
    vector<int> levelorder(TreeNode* root) {
        vector<int> res;
        queue<TreeNode*> q;
        if (!root) return res;
        q.push(root);
        while(!q.empty()){
            TreeNode *cur = q.front(); //指针 = 指针 都指向同一块区域
            q.pop();
            res.push_back(cur->val);
            if(cur->left) q.push(cur->left);
            if(cur->right) q.push(cur->right);
        }
        return res;
    }
}

```

剑指 Offer 32 - II. 从上到下打印二叉树 II

难度 简单 95 收藏 分享 切换为英文 接收动态 反馈

从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。

例如：

给定二叉树: [3,9,20,null,null,15,7] .

```

3
/ \
9  20
/ \
15   7

```

返回其层次遍历结果：

```

[
[3],
[9,20],
[15,7]
]

```

```

class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        queue<TreeNode*> q;
        vector<vector<int>> vec;
        if(!root) return vec;
        q.push(root);
        q.push(root);
        while(!q.empty()){
            vector<int> res;
            int l = q.size();
            for(int i = 0; i < l; i++){
                cur = q.front();
                q.pop();
                res.push_back(cur->val);
                if(cur->left) q.push(cur->left);
                if(cur->right) q.push(cur->right);
            }
            vec.push_back(res);
        }
        return vec;
    }
};

```

队列的定义必须要指明队列元素的类型

判断根结点为空 return vec 要保证返回类型相同

这个vec相当于一个临时的计数保存容器，每次
循环一次都要清空，因此应该放到while循环体
内部进行定义，每次都定义一个新的vec

剑指 Offer 32 - III. 从上到下打印二叉树 III

难度 中等 88 收藏 分享 切换为英文 接收动态 反馈

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

例如：

给定二叉树: [3,9,20,null,null,15,7] .

```

3
/ \
9  20
/ \
15   7

```

返回其层次遍历结果：

```

[
[3],
[20,9],
[15,7]
]

```

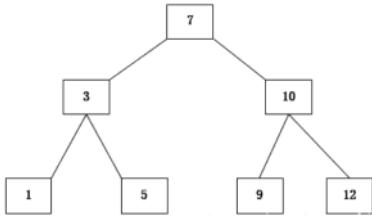
```

class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        queue <TreeNode*> q;
        vector<vector<int>> vec;
        if(!root) return vec;
        q.push(root);
        TreeNode *cur;
        while(!q.empty()){
            vector<int> res;
            int l = q.size();
            for(int i = 0; i < l; i++){
                cur = q.front();
                q.pop(); //元素出队
                res.push_back(cur->val); //存储遍历顺序
                if(cur->left) q.push(cur->left); //元素入队
                if(cur->right) q.push(cur->right); //元素入队
            }
            int k = vec.size();
            if((k & 1) == 1){ //奇数行来了
                reverse(res.begin(),res.end()); //vector的reverse函数
                vec.push_back(res);
            }
            else if((k & 1) == 0) //偶数行来了
                vec.push_back(res);
        }
        return vec;
    }
}

```

二叉搜索树/二叉排序树

根结点的值大于左子树结点，小于右子树右结点，对于每一个子树的结点都适用



剑指 Offer 33. 二叉搜索树的后序遍历序列

难度 中等 236 收藏 100 贡献 100

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 true，否则返回 false。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

```

5
/
2   6
/
1   3

```

示例 1：

输入: [1,6,3,2,5]
输出: false

示例 2：

输入: [1,3,2,6,5]
输出: true

```

class Solution {
public:
    bool verifyPostorder(vector<int>& postorder) {
        if (postorder.empty()) return true;
        return isBST(postorder, 0, postorder.size()-1);
    }
    bool isBST(vector<int>& post, int l, int r){
        if(l >= r)
            return true;
        int flag = post[r];
        int i = l; //下面的循环要用到这个i,从i+1开始起查询是否有小于flag的数,须在for外定义
        for(i = l; i < r; i++) //找第一个大于根结点的位置 该点到尾点之前为右子树
            if(post[i] > flag)
                break;
        for(int j = i + 1; j < r; j++)
            if(post[j] <= flag)
                return false;
        return isBST(post, l, i-1) && isBST(post,i,r-1);
    }
};

```

普通链表的复制过程

```

class Solution {
public:
    Node* copyRandomList(Node* head) {
        Node* cur = head;
        Node* dum = new Node(0), *pre = dum;
        while(cur != nullptr) {
            Node* node = new Node(cur->val); // 复制节点 cur
            pre->next = node; // 新链表的前驱节点 -> 当前节点
            cur = cur->next; // 遍历下一点
            pre = node; // 保存当前新节点
        }
        return dum->next;
    }
};

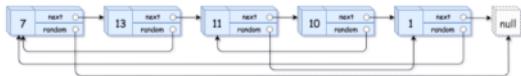
```

剑指 Offer 35. 复杂链表的复制

难度 中等 收藏 分享 为切换为英文 接收动态 反馈

请实现 copyRandomList 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 next 指针指向下一个节点，还有一个 random 指针指向链表中的任意节点或者 null。

示例 1：



```

class Solution {
public:
    Node* copyRandomList(Node* head) {
        if(head == nullptr)
            // return false;
            return nullptr;
        unordered_map<Node*, Node*> map;
        Node *cur = head;
        while(cur){ // 复制各节点，并建立“原节点 -> 新节点”的 Map 映射
            map[cur] = new Node(cur->val);
            cur = cur->next;
        }
        cur = head; // 构建新链表的 next 和 random 指向
        while(cur){
            map[cur]->next = map[cur->next];
            map[cur]->random = map[cur->random];
            cur = cur->next;
        }
        return map[head]; // 返回该头结点表示拷贝得到的链表 //就是一块内存区域
    }
};

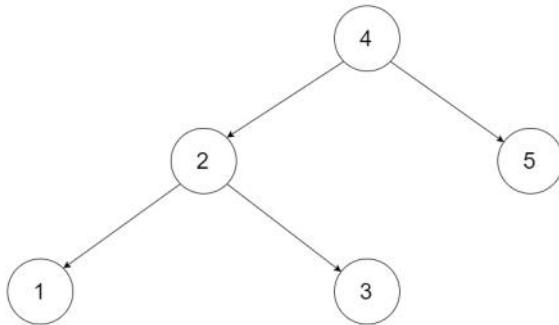
```

剑指 Offer 36. 二叉搜索树与双向链表

难度 中等 收藏 分享 为切换为英文 接收动态 反馈

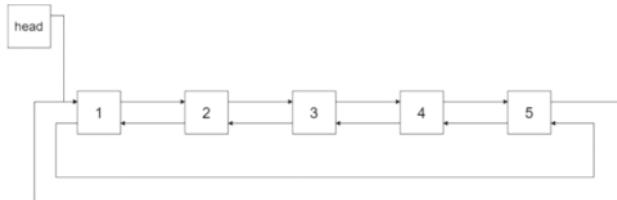
输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

为了让您更好地理解问题，以下面的二叉搜索树为例：



我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

下图展示了上面的二叉搜索树转化成的链表。“head”表示指向链表中有最小元素的节点。



特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

```

class Solution {
    //中序遍历二叉搜索树得到排序序列 再两个for循环构造双向链表,结点用left/right指针连接
    void inorder(vector<Node*> &order, Node *root){
        if(!root) return; //void函数的return语句必须是空语句
        inorder(order,root->left);
        order.push_back(root);
        inorder(order,root->right);
    }
public:
    Node* treeToDoublyList(Node* root) {
        if(!root) return nullptr;
        vector<Node*> res; //存放中序遍历的结果
        inorder(res,root);
        for(int i = 0; i < res.size()-1; i++)
            res[i] -> right = res[i+1];
        for(int j = res.size()-1; j > 0; j--)
            res[j] -> left = res[j-1];
        res[0] -> right = res[0];
        res[0] -> left = res[res.size()-1];
        Node* head = res[0];
        return head;
    }
}

```

快排原理

两个核心：1.哨兵划分 2.递归

哨兵划分：以数组某个元素一般取首元素为基准数 将所有小于基准数的元素移至其左边，将大于基准数的元素移至其右边 经过一轮哨兵划分 可以将数组排序问题拆分为两个较短数组的排序问题

递归：对左子数组和右子数组递归执行哨兵划分 直至子数组长度为1时终止递归

即可完成对整个数组的排序

快速排序和二分法的原理类似，都是以log时间复杂度缩小搜索区间

快排的平均时间复杂度： $O(N \log N)$

平均空间复杂度最好为 $O(\log N)$ 最差情况(输入数组完全倒序) 为 $O(N)$

输入整数数组 arr，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8 这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

输入：arr = [3,2,1], k = 2
输出：[1,2] 或者 [2,1]

示例 2：

输入：arr = [0,1,2,1], k = 1
输出：[0]

```

class Solution {
public:
    vector<int> getLeastNumbers(vector<int>& arr, int k) {
        quickSort(arr, 0, arr.size()-1);
        vector<int> res;
        res.assign(arr.begin(), arr.begin() + k);
        return res;
    }
    void quickSort(vector<int> &arr, int l, int r){
        if(l > r) return; //递归函数的边界条件,递归终止条件
        int i = l, j = r; //左右边界重合 子数组的长度为1
        while(i < j){
            while(i < j && arr[j] >= arr[l]) --j;
            while(i < j && arr[i] <= arr[l]) ++i;
            swap(arr[i],arr[j]);
        }
        swap(arr[l],arr[i]);
        quickSort(arr, l, i-1);
        quickSort(arr,i+1,r);
    }
};

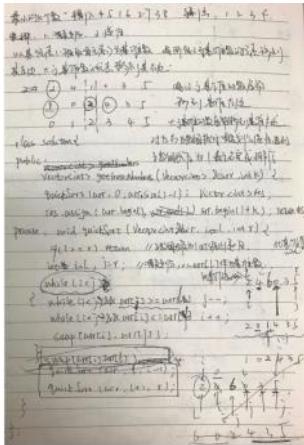
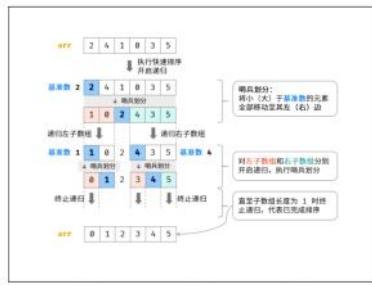
```

```

class Solution{
public:
    vector<int> getLeastNumbers(vector<int> &arr,int k){
        if(k >=arr.size())
            return arr;
        return quickSort(arr, k , 0, arr.size() -1);
    }
    vector<int> quickSort(vector<int> &arr, int k, int l, int r){
        // if(l >r) return[];
        int i = l, j = r;
        while(i < j){
            while(i < j && arr[j] >= arr[l]) j--;
            while(i < j && arr[i] <= arr[l]) i++;
            swap(arr[i],arr[j]);
        }
        swap(arr[i],arr[l]);
        if(i < k) return quickSort(arr, k, i+1, r); //递归右子数组
        if(i > k) return quickSort(arr, k, l, i-1); //递归左子数组 直到i=k
        vector<int> res; //如k,只需要找到k+1大的数,对右子数组排序就行了
        res.assign(arr.begin(), arr.begin() + k);
        return res;
    }
};

```

这个就是快速排序算法



归并排序/分治思想

分：不断将数组从中点位置划分/二分法，将整个数组的排序问题转化为子数组的

排序问题

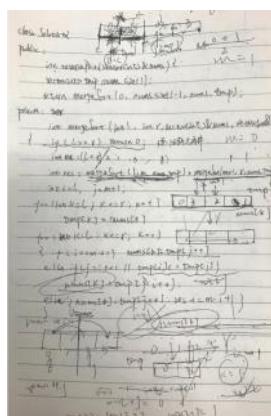
治：划分到子数组长度为1时，开始向上合并，不断将较短排序数组合并为较长排序数组
直至合并到原数组时完成排序

时间复杂度O(NlogN) 空间复杂度O(N) 辅助数组tmp占O(N)的额外空间

逆序对

合并阶段是合并两个排序数组的过程，当遇到左子数组当前元素>右子数组当前元素

即左子数组当前元素到末尾元素与右子数组当前元素构成若干逆序对



剑指 Offer 51. 数组中的逆序对

难度 困难 383 ★★★

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

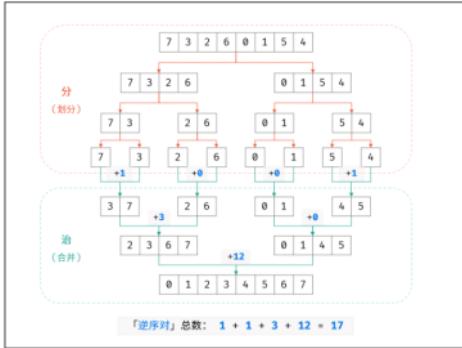
示例 1:

输入: [7,5,6,4]
输出: 5

```

class Solution {
public:
    int reversePairs(vector<int>& nums) {
        vector<int> tmp(nums.size());
        return mergeSort(0,nums.size()-1, nums, tmp);
    }
private:
    int mergeSort(int l, int r, vector<int>& nums, vector<int> &tmp){
        if(l>=r) return 0; //递归终止条件
        int m =(l+r)/2;
        int res = mergeSort(l, m, nums, tmp) + mergeSort(m+1, r, nums, tmp); //递归划分结束
        //合并阶段
        int i = l, j = m+1;
        for(int k = l; k <=r; ++k){
            tmp[k] = nums[k]; //nums为操作的排序数组
        }
        for(int k = l; k <=r; ++k){
            if(i==m+1) nums[k]=tmp[j++];
            else if(j == r+1 || tmp[i]<tmp[j]) nums[k] = tmp[i++]; //是将tmp/原数组赋值给新的nums数组
            else if(tmp[i] > tmp[j]) {nums[k] = tmp[j++], res += (m-i+1); }
        }
    }
    return res;
}

```



算法流程：

merge_sort() 归并排序与逆序对统计：

1. 终止条件：当 $l \geq r$ 时，代表子数组长度为 1，此时终止划分；
2. 递归划分：计算数组中点 m ，递归划分左子数组 $merge_sort(l, m)$ 和右子数组 $merge_sort(m + 1, r)$ ；
3. 合并与逆序对统计：
 1. 停存数组 $nums$ 闭区间 $[i, r]$ 内的元素至辅助数组 tmp ；
 2. 循环合并：设置双指针 i, j 分别指向左 / 右子数组的首元素；
 - 当 $i = m + 1$ 时：代表左子数组已合并完，因此添加右子数组当前元素 $tmp[j]$ ，并执行 $j = j + 1$ ；
 - 否则，当 $j = r + 1$ 时：代表右子数组已合并完，因此添加左子数组当前元素 $tmp[i]$ ，并执行 $i = i + 1$ ；
 - 否则，当 $tmp[i] \leq tmp[j]$ 时：添加左子数组当前元素 $tmp[i]$ ，并执行 $i = i + 1$ ；
 - 否则（即 $tmp[i] > tmp[j]$ ）时：添加右子数组当前元素 $tmp[j]$ ，并执行 $j = j + 1$ ；此时构成 $m - i + 1$ 个「逆序对」，统计添加至 res ；
4. 返回值：返回直至目前的逆序对总数 res ；

这就是归排算法

二分查找

如果目标值等于中间元素，则找到目标值

如果目标值较小，则继续在左侧搜索

如果目标值较大，则继续在右侧搜索

算法：

初始化指针 $left=0$, $right=n-1$

当 $left <= right$:

比较中间元素 $nums[pivot]$ 与目标值 $target$

如果 $target == nums[pivot]$ 返回 $pivot$

如果 $target < nums[pivot]$ 继续在左侧搜索 $right=pivot - 1$

如果 $target > nums[pivot]$ 继续在右侧搜索 $left=pivot + 1$

Python | Java | C++

```

class Solution {
public:
    int search(vector<int>& nums, int target) {
        int pivot, left = 0, right = nums.size() - 1;
        while (left <= right) {
            pivot = left + (right - left) / 2;
            if (nums[pivot] == target) return pivot;
            if (target < nums[pivot]) right = pivot - 1;
            else left = pivot + 1;
        }
        return -1;
    }
};

```

- 时间复杂度: $O(\log N)$
- 空间复杂度: $O(1)$

剑指 Offer 54. 二叉搜索树的第k大节点

难度 简单 142 收藏

给定一棵二叉搜索树，请找出其中第k大的节点。

```
class Solution {
public:
    vector<int> res; //BST的中序遍历结果就是结点值顺序排序结果
    void dfs(TreeNode* root){
        if(!root)
            return ;
        dfs(root -> left);
        res.push_back(root ->val);
        dfs(root ->right);
    }
    int kthLargest(TreeNode* root, int k) {
        dfs(root);
        reverse(res.begin(),res.end()); //第k大的结点
        return res[k-1];
    }
}
```

剑指 Offer 57 - II. 和为s的连续正数序列

难度 简单 246 收藏

输入一个正整数 target，输出所有和为 target 的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

```
class Solution {
public:
    vector<vector<int>> findContinuousSequence(int target) {
        vector<vector<int>> vec;
        vector<int> res;
        int sum = 0, limit = target / 2; //i < 9/2 i<4就错了 i<=4 4+5 这样才是对的
        //int sum = 0, limit = (target - 1) / 2; // (target - 1) / 2 等效于 target / 2 下取整
        for (int i = 1; i <= limit; ++i) {
            for (int j = i; j <= i; ++j) {
                sum += j;
                if (sum > target) {
                    sum = 0;
                    break;
                } else if (sum == target) {
                    res.clear();
                    for (int k = i; k <= j; ++k) {
                        res.emplace_back(k);
                    }
                    vec.emplace_back(res);
                    sum = 0;
                    break;
                }
            }
        }
        return vec;
    }
}
```

```
class Solution {
public:
    vector<vector<int>> findContinuousSequence(int target) {

```

```
        int i = 1, j = 2, s = 3;
        vector<vector<int>> res;
        while(i < j){
            if(s == target){
                vector<int> vec;
                for(int k = i; k <= j; k++)
                    vec.push_back(k);
                res.push_back(vec);
            }
            if(s >= target){
                s -= i;
                i++;
            }
            else{
                //s+= j;
                //j++;
                j++;
                s += j;
            }
        }
        return res;
    }
}
```

滑动窗口双指针法

```
/*
初始化： 左边界 i = 1 , 右边界 j = 2 , 元素和 s = 3 , 结果列表 res ;
循环： 当 i>j 时跳出；
当 s > target时： 向右移动左边界 i = i + 1 , 并更新元素和 s ;
当 s < target时： 向右移动右边界 j = j + 1 , 并更新元素和 s ;
当 s = target时： 记录连续整数序列，并向右移动左边界 i = i + 1 ;
*/
```

面试题59 - I. 滑动窗口的最大值

难度 困难 通过 237 收藏 复制

给定一个数组 `nums` 和滑动窗口的大小 `k`，请找出所有滑动窗口里的最大值。

示例:

输入: `nums = [1,3,-1,-3,5,3,6,7]`, 和 `k = 3`

输出: `[3,3,5,5,6,7]`

解释:

滑动窗口的位置

最大值

滑动窗口的位置	最大值
<code>[1 3 -1] -3 5 3 6 7</code>	<code>3</code>
<code>1 [3 -1 -3] 5 3 6 7</code>	<code>3</code>
<code>1 3 [-1 -3 5] 3 6 7</code>	<code>5</code>
<code>1 3 -1 [-3 5 3] 6 7</code>	<code>5</code>
<code>1 3 -1 -3 [5 3 6] 7</code>	<code>6</code>
<code>1 3 -1 -3 5 [3 6 7]</code>	<code>7</code>

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        if(nums.empty())
            return {};
        vector<int> res;
        int start = 0, end = k-1;
        while(end<nums.size()){
            // for; end < nums.size(); end++){ //遍历end下标 ,for与while可以等价
            int temp = INT_MIN;
            for(int i = start; i <= end; i++){
                temp = max(temp, nums[i]);
            }
            res.push_back(temp);
            start++; //end++;
            end++;
        }
        return res;
    }
};

class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> res;
        if(nums.empty())// 特殊情况的处理
            return {};
        deque<int> deQ;//维护一个双端队列
        for(int i = 0; i < k; i++)/* 滑动窗口大小小于k时,形成第一个滑窗
            while(!deQ.empty()&& deQ.back() < nums[i]) //保持队列首元素为最大值
                deQ.pop_back();
            deQ.push_back(nums[i]);
        */
        res.push_back(deQ.front()); //将第一个滑窗的最大值存入vec
        for(int i = k; i < nums.size() - 1; i++){
            if(deQ.front() == nums[i-k]) //如果队首元素超出了滑窗范围,出队列 不属于局部比较范围了
                deQ.pop_front();
            deQ.pop_front();
            while(!deQ.empty() && deQ.back() < nums[i]) //保持队首元素为当前区域最大值
                deQ.pop_back();
            deQ.push_back(nums[i]);
            res.push_back(deQ.front()); //将每次移动一步滑窗的最大值保存到时vec
        }
        return res;
    }
};
```

top

2020年11月5日 19:22

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：1118702

本题知识点：链表

■ 算法知识视频讲解

题目描述

输入一个链表，反转链表后，输出新链表的头头。

示例1

输入

(1,2,3)

返回值

(3,2,1)

说明：本题目包含复杂数据结构 ListNode，[点击查看相关信息](#)

~关联企业

```
1 /*
2 struct ListNode{
3     int val;
4     struct ListNode *next;
5     ListNode(int x) : val(x), next(NULL) {}
6 }
7 */
8
9 class Solution {
10 public:
11     ListNode* ReverseList(ListNode* pHead) {
12         ListNode *pre = NULL;
13         ListNode *cur = pHead;
14         ListNode *nex = NULL;
15         while(cur){
16             nex = cur->next;
17             cur->next = pre;
18             pre = cur;
19             cur = nex;
20         }
21     }
22     return pre;
23 }
24
25 }
```

//你都不会让指针指向头结点？？？

然后每次快指针走两步，慢指针走一步，所以进行一次操作以后，快慢指针之间的距离是+1。再操作一次快-2, -3... 直到操作到最后，快慢指针之间的距离为0，即两个指针相遇，所以，如果链表有环的话，快慢指针一定可以在n次操作后相遇

C++

```
1 class Solution {
2 public:
3     bool hasCycle(ListNode *head) {
4         ListNode *fast = head;
5         ListNode *slow = head;
6         while(slow != NULL && fast->next != NULL && fast->next->next != NULL) {
7             fast = fast->next->next;
8             slow = slow->next;
9             if(fast == slow)
10             {
11                 return true;
12             }
13         }
14         return false;
15     }
16 };
17 }
```

```
6 /**
7 * Definition for singly linked list.
8 */
9 class Solution {
10 public:
11     bool hasCycle(ListNode *head) {
12         ListNode *fast = head;
13         ListNode *slow = head;
14         while(slow != NULL && fast != NULL && fast->next != NULL && fast->next->next != NULL) {
15             fast = fast->next->next;
16             slow = slow->next;
17             if(fast == slow)
18                 return true;
19         }
20         return false;
21     }
22 }
```

题目描述

给定一个数组，找出其中最小的K个数。例如数组元素是4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4。如果K>数组的长度，那么返回一个空的数组

示例1

输入

//if语句加； 你真是个傻逼

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：821031

本题知识点：栈

■ 算法知识视频讲解

题目描述

用两个栈来实现一个队列，完成队列的Push和Pop操作。队列中的元素为int类型。

~关联企业

字节跳动

米哈游

深信服

vivo

~关联职位

研发

测试

前端

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：135888

本题知识点：数组 哈希

■ 算法知识视频讲解

题目描述

给出一个整数数组，请在数组中找出两个加起来等于目标值的数。
你给出的函数twoSum 需要返回这两个数字的下标 (index1, index2)，需要满足 index1 小于 index2。注意：下标是从1开始的
假设给出的数组中只存在唯一解
例如：
给出的数组为 [20, 70, 110, 150],目标值为90
输出 index1=1, index2=2

示例1

输入

```
2 public:
3     vector<int> GetLeastNumbers_Solution(vector<int> input,
4                                         //</int l=0,r = input.size()-1;
5                                         vector<int> res;
6     if(l> input.size())
7         return res;
8     quick_sort(input,0, input.size()-1);
9
10    res.assign(input.begin(),input.begin()+k);
11    return res;
12 }
```

① C++(clang++11) ▾ ● 核心代码模式

```
1 class Solution
2 {
3 public:
4     void push(int node) {
5         stack1.push(node);
6     }
7
8     int pop() {
9         if(stack2.empty()){
10             while(!stack1.empty()){
11                 stack2.push(stack1.top());
12                 stack1.pop();
13             }
14         }
15         if(stack2.empty()) return 0;
16         else{
17             int del = stack2.top(); //这两句是连在一起的，上面push完下面还要pop！
18             stack2.pop();
19             return del;
20         }
21     }
22 }
```

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：135888

本题知识点：数组 哈希

■ 算法知识视频讲解

题目描述

给出一个整数数组，请在数组中找出两个加起来等于目标值的数。
你给出的函数twoSum 需要返回这两个数字的下标 (index1, index2)，需要满足 index1 小于 index2。注意：下标是从1开始的
假设给出的数组中只存在唯一解
例如：
给出的数组为 [20, 70, 110, 150],目标值为90
输出 index1=1, index2=2

```
① C++(clang++11) ▾ ● 核心代码模式
1 class Solution {
2 public:
3     vector<int> twoSum(vector<int> &numbers, int target) {
4         // write code here
5         vector<int> res;
6         unordered_map<int,int> mp;
7         for(int i = 0; i < numbers.size(); i++)
8             mp[numbers[i]] = i;
9         for(int i = 0; i < numbers.size(); i++){
10             int sub = target-numbers[i];
11             if(mp.find(sub) != mp.end() && (mp[sub] != i)){
12                 res.push_back(i+1);
13                 res.push_back(mp[sub]+1);
14                 break;
15             }
16         }
17         return res;
18     }
19 }
20 }
```

题目描述

给定一个数组arr，返回arr的最长无的重复子串的长度(无重复指的是所有数字都不相同)。

示例1

输入
[2, 3, 4, 5]
返回值
4

示例2

输入
[2, 2, 3, 4, 3]
返回值
3

```
① C++(clang++11) ✓ 核心代码模式
1 #include<bits/stdc++.h>
2 using namespace std;
3 class Solution {
4 public:
5     /**
6      * @param arr int 数组
7      * @return int 长度
8      */
9
10    int maxLength(vector<int>& arr) {
11        // write code here
12        set<int> se;
13        int l = 0, r = 0;
14        int cnt = 1;
15        while(l < arr.size() && r < arr.size()){
16            if(se.find(arr[r]) == se.end()){
17                se.insert(arr[r++]);
18                cnt = max(cnt, r-l);
19            }
20            else{
21                se.erase(arr[l++]);
22            }
23        }
24        return cnt;
25    }
26 }
```

题目描述

给出一个仅包含字符'(',')','[和']'的字符串，判断给出的字符串是否是合法的括号序列。括号必须以正确的顺序关闭，"()"和"[]()"都是合法的括号序列，但"["和"]]"不合法。

示例1

输入
"[
返回值
false

示例2

输入
"[]"
返回值
true

```
3
4
5    * @param s string 字符串
6    * @return bool 为真型
7    */
8    bool isValid(string s) {
9        // write code here
10       stack<char> sk;
11       for(int i = 0; i < s.size(); i++){
12           if(sk.empty()){
13               sk.push(s[i]);
14               continue;
15           }
16           //if(s[i] == ')') {
17           //    if(sk.top() == '(') sk.pop();
18           if(s[i] == ')' && sk.top() == '(') sk.pop();
19           else if(s[i] == ']'){
20               if(sk.top() == '[') sk.pop();
21           }
22           else if(s[i] == ']') {
23               if((sk.top() == '[')) sk.pop();
24           }
25           else sk.push(s[i]);
26       }
27       return sk.empty();
28   }
29 }
```

409. 最长回文串

难度 简单 316 ★★★☆☆

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造出的最长的回文串。

在构造过程中，请注意区分大小写。比如 "aa" 不能当做一个回文字符串。

注意：

假设字符串的长度不会超过 1010。

示例 1：

输入：
"abcccccdd"
输出：
7

解释：
我们可以构造的最长的回文串是"dcacacd"，它的长度是 7。

在对char数组遍历的时候 只能出现一个个数为奇数的字符 所以我们直接记录有多少个字符出现次数为奇数就可以了啊 不用额外判断了

```
public int longestPalindrome(String s) {
    // 找出可以构成最长回文串的长度
    int[] arr = new int[128];
    for(char c : s.toCharArray()) {
        arr[c]++;
    }
    int count = 0;
    for (int i : arr) {
        count += (i % 2);
    }
    return count == 0 ? s.length() : (s.length() - count + 1);
}
```

时间限制 : C/C++ 1秒 , 其他语言2秒 空间限制 : C/C++ 64M , 其他语言128M

热度指数 : 1038497

本题知识点 : 数组

算法知识视频讲解

题目描述

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0，第1项是1）。

$n \leq 39$

示例1

输入

4

复制

返回值

3

复制

题目描述

请实现有重复数字的升序数组的二分查找。

给定一个元素有序的（升序）整型数组 nums 和一个目标值 target ，写一个函数搜索 nums 中的 target，如果目标值在返回下标，否则返回 -1。

示例1

输入

[1, 2, 4, 4, 5], 4

复制

返回值

2

复制

说明

从左到右，查找到第一个为4的，下标为2，返回2。

复制

示例2

牛客题库上做过的 | 题表中环的入口节点 | 相关的企业面试 | < 上一题 | 下一题 > | 我的提交

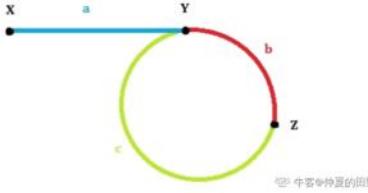
时间限制 : C/C++ 1秒 , 其他语言2秒 空间限制 : C/C++ 32M , 其他语言64M 热度指数 : 106100

本题知识点 : 链表 双指针

题解 | J

NC3：链表中环的入口节点

1. 第一步，找环中相汇点。分别用p1, p2指向链表头部。
2. p1每次走一步，p2每次走二步，直到p1==p2找到在环中的相汇点。



举报

- 那么我们可以知道fast指针走过 $a+b+c+b$
- slow指针走过 $a+b$
- 那么 $2(a+b) = a+b+c+b$
- 所以 $a = c$
- 那么此时让slow回到起点，fast依然停在Z，两个同时开始走，一次走一步
- 那么它们最终会相遇在Y点，正是环的起始点

时间限制 : C/C++ 1秒 , 其他语言2秒 空间限制 : C/C++ 256M , 其他语言512M 热度指数 : 16922

本题知识点 : 数组 动态规划

算法知识视频讲解

题目描述

给定一个 $n * m$ 的矩阵 a，从左上角开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，输出所有的路径中最小的路径和。

示例1

输入

[[1,3,5,9],[8,1,3,4],[5,0,6,1],[8,0,4,0]]

复制

返回值

12

① C++(clang++11) 核心代码模式

```
1 class Solution {
2 public:
3     int Fibonacci(int n) {
4         vector<int> dp;
5         for(int i = 0; i <= n; i++) {
6             if(i == 0)
7                 dp.push_back(0);
8             else if(i == 1)
9                 dp.push_back(1);
10            else
11                dp[i] = dp[i-1] + dp[i-2];
12            dp.push_back(dp[i-1] + dp[i-2]);
13        }
14        return dp[n];
15    }
16 }
```

① C++(clang++11) 核心代码模式

```
1 class Solution {
2 public:
3     /**
4      * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
5      */
6     // 如果目标值存在返回下标，否则返回 -1
7     #param nums: int整型vector<int>
8     #param target: int整型
9     #return: int整型
10    */
11    int search(vector<int>& nums, int target) {
12        // write code here
13        int mid, left = 0, right = nums.size() - 1;
14        while (left <= right) {
15            mid = left + (right - left) / 2;
16            if (nums[mid] == target) { // 有多个重复数它也是排序的找到第一个
17                while (mid > 0 && nums[mid] == target)
18                    mid--;
19            }
20            return mid + 1;
21        }
22        if (target < nums[mid]) right = mid - 1;
23        else left = mid + 1;
24    }
25    return -1;
26 }
```

① C/C++(clang++11) 核心代码模式

```
1 class Solution {
2 public:
3     /**
4      *
5      * @param matrix: int整型vector<vector<int>> the matrix
6      */
7     int minPathSum(vector<vector<int>> &matrix) {
8         // write code here
9         int n = matrix.size(), m = matrix[0].size();
10        for(int i = 1; i < n; i++)
11            matrix[i][0] += matrix[i-1][0];
12        for(int j = 1; j < m; j++)
13            matrix[0][j] += matrix[0][j-1];
14        for(int i = 1; i < n; i++)
15            for(int j = 1; j < m; j++)
16                matrix[i][j] = min(matrix[i-1][j], matrix[i][j-1]) + matrix[i][j];
17        return matrix[n-1][m-1];
18    }
19 }
20 }
```

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 32M, 其他语言64M 热度指数：101704

本题知识点： 数组 双指针

算法知识视频讲解

题目描述

给出两个有序的整数数组 A 和 B ，请将数组 B 合并到数组 A 中，变成一个有序的数组。

注意：

可以假设 A 数组有足够的空间存放 B 数组的元素。 A 和 B 中初始的元素数目分别为 m 和 n 。

▲ 关联企业

◀ 跟谁学 ▶ 字节跳动 ▶ 网易蓝火

▲ 关联职位

◀ 前端 ▶ 研发 ▶ 测试 ▶ 算法

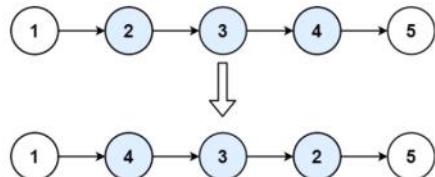
```
C++(dang++11) 核心代码模式
1 class Solution {
2 public:
3     void merge(int A[], int m, int B[], int n) {
4         int a = m-1;
5         int b = n-1;
6         for(int i = m+n-1; i >= 0; i--) { //需要填m+n次
7             if(b<0 || (a>=0 && A[a] >= B[b])) {
8                 A[i] = A[a]; //A的元素用完了就直接填B数组，A数组元素应大于B数组元素大，填A的元素
9                 a--;
10            } else{
11                A[i] = B[b];
12                b--;
13            }
14        }
15    }
16 }
17 }
```

<https://leetcode-cn.com/problems/reverse-linked-list-ii/solution/yi-ge-neng-yong-suo-you-lian-biao-t-vjx6/>

92. 反转链表 II
难度 中等 ⚡ 859 ☆ 困 高 亮

给你单链表的头指针 head 和两个整数 left 和 right，其中 left <= right。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表。

示例 1：



输入 : head = [1,2,3,4,5], left = 2, right = 4
输出 : [1,4,3,2,5]

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int l, int r) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;

        // 注意这里是 l 不是 1
        r -= l;
        // hh 就是 “哈哈”的意思 ...
        // 啊哈。hh 是 head 的意思，为了防止与 height 的简写 h 冲突
        ListNode* hh = dummyHead;
        while (l-- > 1)
            hh = hh->next;

        ListNode* prv = hh->next;
        ListNode* cur = prv->next;
        while (r-- > 0) {
            ListNode* nxt = cur->next;
            cur->next = prv;
            prv = cur;
            cur = nxt;
        }
        hh->next->next = cur;
        hh->next = prv;
        return dummyHead->next;
    }
}
```

题目描述 评论 (977) 题解 (1.6k) 提交记录 关闭 < 上一题解 5 / 1580 下一题解 >

Q 搜索题解 J 排序 + 写题解

不限 C++ Bash C C# Go Java JavaScript Kotlin PHP Python Python3 Ruby Rust Scala Swift TypeScript

线 排序 图 设计 递归 脑筋急转弯 Map 数组 哈希表 链表 双指针 字符串 字节跳动 推想 小白 哨兵 test 简单 指针 迭代 list 快慢指针 时间复杂度 递归算法 单调栈 模拟 cpp js

【动画模拟】很简单 一步一步搞懂如何用 迭代法逐步解决问

可乐可乐吗QAQ 2021-03-18

打卡,贴个c++简介版

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        auto dummyNode = new ListNode(-1),h = dummyNode;
        dummyNode->next = head;
        for(int i = 0; i < left - 1; i++,h = h->next);
        auto p = h->next;
        for(int i = 0; i < right - left; i++){
            auto q = p->next;
            p->next = q->next;
            q->next = h->next;
            h->next = q;
        }
        return dummyNode->next;
    }
};
```

25. K 个一组翻转链表
难度 困难 提交 1035 AC 8.4%

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

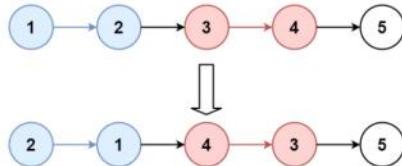
k 是一个正整数，它的值小于或等于链表的长度。

如果链表总长度不是 k 的整数倍，那么请将最后剩余的节点保持原顺序。

进阶：

- 你可以设计一个只使用常数额外空间的算法来解决此问题吗？
- 你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

示例 1：



题目... 随... < 上一题 25/2036 下一题 > 控制台 贡献

```
1 class Solution {
2     public:
3         ListNode* reverseKGroup(ListNode *head, int k) {
4             ListNode *dummy = new ListNode();
5             dummy->next = head;
6             dummy->next->head;
7             ListNode *pre = dummy, *cur = head;
8             //计算链表长度
9             int length = 0;
10            while(head){
11                head = head->next;
12                length++;
13            }
14            //对链表中每一组的分段
15            for(int i = 0; i < length / k; i++){
16                //循环将当前分段内的节点反转
17                for(int j = 0; j < k - 1; j++){
18                    ListNode *t = cur->next;
19                    cur->next = t->next;
20                    t->next = pre->next;
21                    pre->next = t;
22                }
23                //pre后移指向向后面一组待分段的第一个节点
24                pre = cur;
25                //cur只需后移一位（即为下一待分段的第一个节点）
26                cur = cur->next;
27            }
28        }
29    }
```

```
class Solution {
public:
    ListNode* dfs(ListNode* h, int k, int m, int b){
        if (b == m)    return h;//递归中止条件,m是可反转的次数,b是实际反转次数
        ListNode*ph = h,*pr = NULL,*t = NULL;//ph记录头指针,即首位
        int cnt = 0;
        /*反转链表基本操作*/
        while (cnt++ <= k){
            t = h->next;
            h->next = pr;
            pr = h;
            h = t;
        }
        ph->next = dfs(h,k,m,b+1); //pr是ph的前一个,即一组中的最后一个数,反转后就是第一个数
        return pr;
    }
    ListNode* reverseKGroup(ListNode* head, int k) {
        int sum = 0;
        ListNode *h = head;
        while (head){
            sum++;
            head = head->next;
        }
        return dfs(h,k,sum/k,0);
    }
};
```

时间限制 : C/C++ 1 秒 , 其他语言2秒 空间限制 : C/C++ 64M , 其他语言128M

热度指数 : 70437

本题知识点 : 链表

算法知识视频讲解

题目描述

将给出的链表中的节点每 k 个一组翻转，返回翻转后的链表

如果链表中的节点数不是 k 的倍数，将最后剩下的节点保持原样

你不能更改节点中的值，只能更改节点本身。

要求空间复杂度 $O(1)$

例如：

给定的链表是 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

对于 $k = 2$ ，你应该返回 $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 5$

对于 $k = 3$ ，你应该返回 $3 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 5$

示例1

输入

```
{1,2,3,4,5},2
```

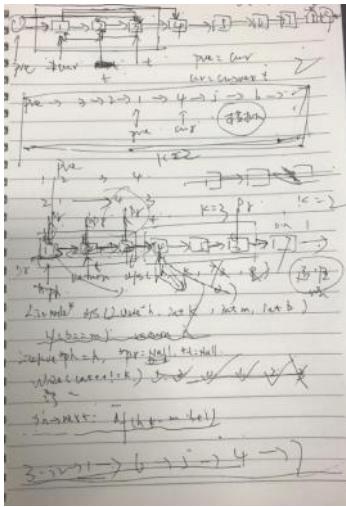
复制

返回值

复制

① C++(clang++11) 核心代码模式

```
13     ...@param:k:int整型.
14     ...@return:ListNode类
15     ...
16     ListNode* dfs(ListNode* h, int k, int m, int b) {
17         if(b==m) return h;
18         ListNode* ph = h, *pr=NULL, *t=NULL;
19         int cnt = 0;
20         int l = k;
21         /*反转链表基本操作*/
22         while (l--){ //k不可以直接作为while循环变量 注意这里
23             t = h->next;
24             h->next = pr;
25             pr = h;
26             h = t;
27         }
28         ph->next = dfs(h,k,m,b+1);
29         return pr;
30     }
31     ListNode* reverseKGroup(ListNode* head, int k) {
32         // write code here
33         int sum = 0;
34         ListNode *h = head;
35         while(head){
36             sum++;
37             head = head->next;
38         }
39         return dfs(h,k,sum/k,0);
```



时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：95860

本题知识点：

■ 算法知识视频讲解

题目描述

将两个有序的链表合并为一个新链表，要求新的链表是通过拼接两个链表的节点来生成的，且合并后新链表依然有序。

示例1

输入

[1], [2]

返回值

[1, 2]

示例2

输入

[1]

```
① C++(clang++11) ② 核心代码模式
1 class Solution {
2 public:
3     ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
4         if(l1 == NULL) return l2;
5         if(l2 == NULL) return l1;
6         ListNode *p1 = l1;
7         ListNode *p2 = l2;
8         ListNode *phead = new ListNode(0);
9         ListNode *pnode = phead;
10        while(p1 != NULL && p2 != NULL) {
11            if(p1->val <= p2->val) {
12                pnode->next = p1;
13                p1 = p1->next;
14            } else {
15                pnode->next = p2;
16                p2 = p2->next;
17            }
18            pnode = pnode->next;
19        }
20        if(p1 != NULL) pnode->next = p1;
21        if(p2 != NULL) pnode->next = p2;
22        return phead->next;
23    }
24 }
25 }
```

牛客网 - 上岸考过 - 求二叉树的层序遍历

左侧的企业面试

```
① C++(clang++11) ② 核心代码模式
1 /**
2  * reverse root TreeNode类
3  * @return int整型vector<vector<int>>
4  */
5
6 vector<vector<int>> levelOrder(TreeNode* root) {
7     // write code here
8     queue<TreeNode*> q;
9     vector<vector<int>> res;
10    if(!root) return res;
11    q.push(root);
12    TreeNode* cur;
13    while (!q.empty()) {
14        int i = q.size();
15        vector<int> vec;
16        for(int i = 0; i < i; i++) {
17            cur = q.front();
18            q.pop();
19            vec.push_back(cur->val);
20            if(cur->left) q.push(cur->left);
21            if(cur->right) q.push(cur->right);
22        }
23        res.push_back(vec);
24    }
25    return res;
26 }
```

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：57939

本题知识点：

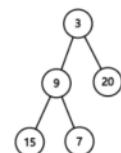
■ 算法知识视频讲解

题目描述

给定一个二叉树，返回该二叉树的之字形层序遍历。（第一层从左向右，下一层从右向左，一直这样交替）

例如：

给定的二叉树是[3,9,20,#,#,15,7]。



二叉树之字形层序遍历的结果是

[
[3]
[20, 9],
...]

```
① C++(clang++11) ② 核心代码模式
19 ..... if(!root) return vec;
20 ..... TreeNode* cur;
21 ..... //cur = root;
22 ..... queue<TreeNode*> q;
23 ..... q.push(root);
24 ..... while(!q.empty()){
25 .....     vector<int> res;
26 .....     int k = q.size();
27 .....     for(int i = 0; i < k; i++){//这个q是会变的，写成q.size()就错了！
28 .....         cur = q.front();
29 .....         q.pop(); //元素出队
30 .....         res.push_back(cur->val); //存储遍历顺序
31 .....         if(cur->left) q.push(cur->left); //元素入队
32 .....         if(cur->right) q.push(cur->right); //元素入队
33 .....     }
34 .....     .....//res.push_back(vec);
35 .....     int k = vec.size();
36 .....     if(k > 0){
37 .....         vec.push_back(res);
38 .....     }
39 .....     else_if((k > 0) == 1){
40 .....         reverse(res.begin(), res.end());
41 .....         vec.push_back(res);
42 .....     }
43 ..... }
```

类似的题目最好是一起刷，这样可以提高刷题效率：

1. leetcode 1 号算法题：两数之和
2. leetcode 167 号算法题：两数之和III - 输入有序数组
3. leetcode 170 号算法题：两数之和III - 数据结构设计
4. leetcode 653 号算法题：两数之和IV - 输入 BST
5. leetcode 15 号算法题：三数之和
6. leetcode 16 号算法题：四数之和

我想跟你说的是，你不会做算法题，并不是因为你不够聪明，而是因为不知道如何高效的刷题

如何高效刷题可以参考：[如何高效刷算法题](#)

如果你觉得刷算法题很痛苦、需要花费大量的时间的话，并不是因为你笨，原因就是你的【数据结构与算法】知识不成体系

如果你的【数据结构与算法】知识不成体系的话，那么：

1. 即使一道很简单的题目，你都要想很长的时间，还不一定会做
2. 然后，你就会看题解，觉得自己会了，但是过一段时间，你就又忘了，单纯的记住答案是不行的
3. 即使这道题目会了，那么题目稍微一变型，你又不会了

所以从现在开始，就每天坚持刷题，刷题的过程就是不断积累经验，不断进步的过程。

```
class Solution {  
public:  
    vector<vector<int>> threeSum(vector<int>& nums) {  
        int n = nums.size();  
        sort(nums.begin(), nums.end());  
        vector<vector<int>> ans;  
        // 枚举 a  
        for (int first = 0; first < n; ++first) {  
            // 需要和上一次枚举的数不相同  
            if (first > 0 && nums[first] == nums[first - 1]) {  
                continue;  
            }  
            // c 对应的指针初始指向数组的最右端  
            int third = n - 1;  
            int target = -nums[first];  
            // 枚举 b  
            for (int second = first + 1; second < n; ++second) {  
                // 需要和上一次枚举的数不相同  
                if (second > first + 1 && nums[second] == nums[second - 1]) {  
                    continue;  
                }  
                // 需要保证 b 的指针在 c 的指针的左侧  
                while (second < third && nums[second] + nums[third] > target) {  
                    --third;  
                }  
                // 如果指针重合，随着 b 后续的增加  
                // 就不会有满足 a+b+c=0 并且 b<c 的 c 了，可以退出循环  
                if (second == third) {  
                    break;  
                }  
                if (nums[second] + nums[third] == target) {  
                    ans.push_back({nums[first], nums[second], nums[third]});  
                }  
            }  
        }  
        return ans;  
    }  
}
```

时间限制 : C/C++ 1 秒, 其他语言 2 秒 空间限制 : C/C++ 64M, 其他语言 128M 热度指数 : 69332

本题知识点 : 数组 双指针

■ 算法知识视频讲解

题目描述

给出一个有 n 个元素的数组 S，在 S 中是否存在元素 a,b,c 满足 a+b+c=0？找出数组 S 中所有满足条件的三元组。

注意：

1. 三元组 (a, b, c) 中的元素必须按非降序排列。（即 a≤b≤c）
2. 解集中不能包含重复的三元组。

例如，给定的数组 S = [-10, 0, 10, 20, -10, -40]，解集为 (-10, 0, 10), (-10, -10, 20)

示例：

输入

{-2,0,1,1,2}

返回值

{(-2,0,2), (-2,1,1)}

时间限制 : C/C++ 3 秒, 其他语言 6 秒 空间限制 : C/C++ 64M, 其他语言 128M 热度指数 : 133782

本题知识点 : 排序 分治

■ 算法知识视频讲解

题目描述

有一个整数数组，请你根据快速排序的思路，找出数组中第 K 大的数。

给定一个整数数组 a，同时指定它的大小 n 和要找的 K (K 在 1 到 n 之间)，请返回第 K 大的数，保证答案存在。

示例 1

输入

[1, 3, 5, 2, 2], 5, 3

返回值

2

```
1 class Solution {  
2 public:  
3     ... vector<vector<int>> threeSum(vector<int>& num) {  
4         ... int n = num.size();  
5         ... vector<vector<int>> res;  
6         ... sort(num.begin(),num.end());  
7         ... for(int first = 0; first < n; first++){  
8             ... if(first>0 && num[first] == num[first-1])  
9                 ... continue;  
10            ... int third = n-1; int target = -num[first];  
11            ... for(int second = first+1; second < n; second++){  
12                ... if(second>first+1 && num[second] == num[second-1])  
13                    ... continue;  
14                ... while(second<third && num[second]+num[third] >target)  
15                    ... third--;  
16                ... if(second == third)  
17                    ... break;  
18                ... if(num[second] + num[third] == target)  
19                    ... res.push_back({num[first],num[second],num[third]});  
20            ... }  
21        ... }  
22        ... return res;  
23    ... }  
24 ... }  
25 ... }
```

```
1 class Solution {  
2 public:  
3     ... int findKth(vector<int> a, int n, int K) {  
4         ... // write code here  
5         ... quick_sort(a,0,n-1);  
6         ... return a[n-K];  
7         ... } ... // ||| 这里如果不引用不会改变实参的值 - 操作符拷贝到的副本  
8         ... void quick_sort(vector<int> &arr, int l, int r){  
9             ... if(l >= r) return; //递归函数的边界条件，递归终止条件  
10            ... int i = l, j = r; //左右边界重合 子数组的长度为1  
11            ... while(i < j){  
12                ... while( i < j && arr[i]>=arr[j]) --j;  
13                ... while( i < j && arr[i]<=arr[j]) ++i;  
14                ... swap(arr[i],arr[j]);  
15            ... }  
16            ... swap(arr[l],arr[i]);  
17            ... quick_sort(arr, l, i-1);  
18            ... quick_sort(arr,i+1,r);  
19        ... }  
20    ... }
```

牛客竞赛-上周考过 > 跳台阶

时间限制 : C/C++ 1秒, 其他语言2秒 空间限制 : C/C++ 64M, 其他语言128M 热度指数 : 797629
本题知识点 : (进阶)

算法知识视频讲解

题目描述

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

关联企业

字节跳动 小米 深信服

关联职位

前端 研发 算法 测试

```
C++(clang++11) 核心代码模式
1 class Solution {
2 public:
3     int jumpFloor(int number) {
4         vector<int> dp;
5         for(int i = 0; i <= number; i++){
6             if(i == 0)
7                 dp.push_back(0);
8             else if(i == 1)
9                 dp.push_back(1);
10            else if(i == 2)
11                dp.push_back(2);
12            else
13                dp.push_back((dp[i-1] + dp[i-2]));
14            //下标越界不是number !
15            //dp.push_back((dp[number-1] + dp[number-2]));
16        }
17        return dp[number];
18    }
19 }
```

牛客竞赛-上周考过 > 子数组的最大累加和问题

时间限制 : C/C++ 2秒, 其他语言4秒 空间限制 : C/C++ 256M, 其他语言512M 热度指数 : 39605
本题知识点 : 分治 动态规划

算法知识视频讲解

题目描述

给定一个数组arr，返回子数组的最大累加和。
例如，arr = [1, -2, 3, 5, -2, 6, -1]，所有子数组中，[3, 5, -2, 6]可以累加出最大的和12，所以返回12。
题目保证没有全为负数的数据
要求
时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$

示例1

输入

```
[1, -2, 3, 5, -2, 6, -1]
```

返回值

```
12
```

LRU

题目描述

设计LRU缓存结构，该结构在构造时确定大小，假设大小为K，并有如下两个功能

- set(key, value)：将记录(key, value)插入该结构
- get(key)：返回key对应的value值

要求

- set/get方法的时间复杂度为 $O(1)$
- 某个key的set或get操作一旦发生，认为这个key的记录成了最常用的记录。
- 当缓存的大小超过K时，移除最不经常使用的记录，即set或get最近远的。

若opt=1，接下来两个整数x, y，表示set(x, y)
若opt=2，接下来一个整数x，表示get(x)，若x未出现过或已被移除，则返回-1
对于每个操作，输出一个答案

示例1

输入

```
[[1,1,1], [1,2,2], [1,3,2], [2,1], [1,4,4], [2,2]] , 3
```

返回值

```
[1,-1]
```

```
C++(clang++11) 核心代码模式
1 class Solution {
2 public:
3     int maxsumofSubarray(vector<int>& arr) {
4         // write code here
5         int cnt = 0;
6         int maxn = 0;
7         for(int i = 0; i < arr.size(); i++) {
8             //cnt+=arr[i];
9             cnt += arr[i];
10            if(cnt < 0)
11                cnt = 0;
12            else if(cnt >= maxn)
13                maxn = cnt;
14        }
15        return maxn;
16    }
17 }
```

LRU

题目描述

vector<int> LRU(vector<vector<int> & operators, int k) {
 // write code here
 vector<int> ret;
 for(auto o : operators){
 if(o[0]==1)
 set(o[1],o[2],k);
 else if(o[0] == 2)
 get(o[1],ret);
 }
 return ret;
}
private:
unordered_map<int,int> map;
list<int> keys;
void set(int key, int value, int k){
 if(keys.size()== k){
 int del = keys.back();
 keys.pop_back();
 map.erase(del);
 }
 map[key] = value;
 keys.push_front(key);
}

```
void get(int key, vector<int>& ret){  
    auto found = map.find(key);  
    if(found == map.end())  
        ret.push_back(-1);  
    else{  
        ret.push_back(found->second);  
        keys.remove(found->first);  
        keys.push_front(found->first);  
    }  
}  
unordered_map<int,int> map; list<int> keys; keys.size() == k int del = keys.back() keys.pop_back() map.erase(key) keys.push_front(key).  
keys.remove(key) /found->first, keys.push_front(key) = /found->first
```

倒数第k个结点，先让fast先走k步 如果此时 fast 为 null 说明删除的是head结点 return head->next

再执行while(fast->next) slow 与 fast同时前进

循环结束 slow的下一个节点就是倒数第k个节点 slow->next = slow->next->next;

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：77477

本题知识点：

栈 栈操作

算法知识提纲讲解

题目描述

给定一个链表，删除链表的倒数第 n 个节点并返回链表的头指针。
例如，
给出的链表为 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$, $n = 2$ 。
删除了链表的倒数第 n 个节点之后，链表变为 $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ 。

备注：

题目保证 n 一定有效的。

请给出求出时间复杂度为 $O(n)$ 的算法。

示例1

输入

{1,2},2

返回值

{1}

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 256M，其他语言512M 热度指数：40009

本题知识点：

链表 链表操作

题目描述

以字符串的形式读入两个数字，编写一个函数计算它们的和，以字符串形式返回。
(字符串长度不大于100000，保证字符串仅由0~9这10个字符组成)

示例1

输入

"1", "99"

返回值

"100"

说明

1+99=100

```
C++(clang++11) 本地代码模式
6
7
8 class Solution {
9 public:
10    /*
11     * @param head: ListNode类
12     * @param n: int整型
13     * @return: ListNode类
14     */
15    ListNode* removeNthFromEnd(ListNode* head, int n) {
16        // write code here
17        // ListNode *fast;
18        // ListNode *slow;
19        ListNode *fast = head;
20        ListNode *slow = head;
21        for (int i = 0; i < n; i++) {
22            fast = fast->next; // 如果fast==null, 慢的一头就指向head
23        }
24        if (!fast) {
25            return head->next;
26        }
27        while (fast->next) { // 这里慢的slow不要删倒数n个节点, 而是倒数n+1个节点,
28            fast = fast->next; // fast不走到null, fast->next走到null
29            slow = slow->next;
30        }
31        slow->next = slow->next->next;
32        return head;
33    }
34}
```

```
C++(clang++11) 本地代码模式
11 string addStrings(string s, string t) {
12     string ans;
13     int count = 0;
14     int l = s.size() - 1, j = t.size() - 1;
15     while (l >= 0 || j >= 0 || count > 0) {
16         int a, b;
17         if (l >= 0)
18             a = s[l] - '0';
19         else
20             a = 0;
21         if (j >= 0)
22             b = t[j] - '0';
23         else
24             b = 0;
25         int tmp = a + b + count;
26         if (tmp >= 10) { // 这里是进位
27             count = 1;
28             tmp -= 10;
29         } else
30             count = 0; // 这里必须加else, count==0 延误进位
31         ans.insert(ans.begin(), tmp + '0');
32         l--;
33         j--;
34     }
35     return ans;
36 }
```

else count 0;

ans.insert(ans.begin(), tmp - '0');

i--; j--;

题目描述

给定两个字符串 str1 和 str2，输出两个字符串的最长公共子串。
题目保证 str1 和 str2 的最长公共子串存在且唯一。

示例1

输入

"1AB2345CD", "12345EF"

返回值

"2345"

备注：

$1 \leq |str_1|, |str_2| \leq 5\,000$

```
5 // @param str1 string字符串的 str1
6 // @param str2 string字符串的 str2
7 // @return string字符串
8 */
9 string LCS(string str1, string str2) {
10    // write code here
11    int n = str1.size(), m = str2.size();
12    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
13    int maxm = 0;
14    int index = 0;
15    for (int i = 1; i <= n; i++) {
16        for (int j = 1; j <= m; j++) {
17            if (str1[i-1] == str2[j-1])
18                dp[i][j] = dp[i-1][j-1] + 1;
19            if (dp[i][j] > maxm)
20                maxm = dp[i][j];
21            index = i;
22        }
23    }
24    string res = str1.substr(index-maxm, maxm);
25    return res;
26}
```

时间限制：C/C++ 3秒，其他语言6秒 空间限制：C/C++ 256M，其他语言512M 热度指数：17870

本题知识点：

动态规划

算法知识提纲讲解

题目描述

给定两个字符串 str1 和 str2，输出这两个字符串的最长公共子序列。如过最长公共子序列为空，则输出-1。

示例1

输入

"1A2C3D4B5E", "B1D23CA45B6A"

返回值

"123456"

说明

"123456" 和 "12346" 都是最长公共子序列，任你输出一个。

备注：

```
C++(clang++11) 本地代码模式
9 string LCS(string s1, string s2) {
10    // write code here
11    int n = s1.size(), m = s2.size();
12    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
13    int maxm = 0;
14    int index = 0;
15    for (int i = 1; i <= n; i++) {
16        for (int j = 1; j <= m; j++) {
17            if (s1[i-1] == s2[j-1])
18                dp[i][j] = dp[i-1][j-1] + 1;
19            else
20                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
21        }
22    }
23    string res = "";
24    for (int i = s1.size(); j = s2.size(); i >= 1 && dp[i][j] >= 1) {
25        if (s1[i-1] == s2[j-1])
26            res += s1[i-1];
27        i--;
28        j--;
29    }
30    while (if (dp[i-1][j] >= dp[i][j-1]) i--) {
31        res += j;
32    }
33    reverse(res.begin(), res.end());
34    if (res.empty())
35        return "-1";
36    else
37        return res;
38}
```

题目描述
输入两个字符串a,b中的最长公共子串。若有很多，输出在较短的串中最先出现的那个。
注：子串的定义：将一个字符串删去前缀和后缀（也可以不删）形成的字符串。请和“序列”的概念分开！

本题有多组输入数据！

输入描述:

输入两个字符串

输出描述:

返回最长出现的字符串

示例1

输入:
abcdefgijklmnp
abcasfjklmnpqrstuvw

输出:
jklmnp

```
2 using namespace std;
3 string a, b;
4 int main()
5 {
6     cin >> a >> b;
7     if (b.size() < a.size())
8     {
9         swap(a, b); // 将较短的那个串放在前面，a中短的位置，b为长子串
10    }
11    int n = a.size(), m = b.size(); // 注意n(i)表示的是子串为a[i-1]~b[i-1]
12    int maxn = INT_MIN; // 表示从 i = maxn 开始的最长子串
13    for (int i = 0; i < n; i++)
14    {
15        for (int j = 0; j < m; j++)
16        {
17            if (a[i] == b[j])
18            {
19                dp[i][j] = dp[i-1][j-1] + 1;
20            }
21            else // 表示s[i-1]和t[j-1]不相等
22            {
23                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
24            }
25        }
26    }
27    cout << a.substr(start, maxn);
28 }
```

时间限制 : C/C++ 3秒, 其他语言6秒 空间限制 : C/C++ 256M, 其他语言512M 热度指数 : 17870
本题知识点 : 动态规划

题解

【C++】《算法导论》中的动态规划

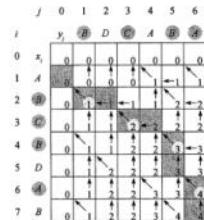


图15-8 图中给出了LCS-LENGTH对 $X=(A, B, C, B, D, A, B)$ 和 $Y=(B, D, C, A, B, A)$ 计算出的表 c 和表 b 。第1行和第1列的方格包含了 $c(i, j)$ 的值和 $b(i, j)$ 记录的箭头。表项 $c[7, 6]$ 表示右下角中的4即为 X 和 Y 的一个LCS(即 C, B, A)的长度。对所有 $i, j > 0$, 表项 $c[i, j]$ 仅依赖于是否 $x_i = y_j$, 以及 $c[i-1, j]$, $c(i, j-1)$ 和 $c[i-1, j-1]$ 的值。这些值都会在 $c(i, j)$ 之前计算出来。为了构造 LCS 中的元素, 从右下角开始沿着 $c(i, j)$ 的箭头前进即可。如图中阴影方格序列。阴影序列中每个“*”对应的表项(高亮显示)表示 $x_i = y_j$ 是 LCS 的一个元素

④ 牛客网牛客网

```
① C++(clang++11) ② 核心代码模式
9     string LCS(string s1, string s2) {
10    // write code here
11    int n = s1.size(), m = s2.size();
12    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
13    int maxn = 0;
14    // int index = 0;
15    for(int i = 1; i <= n; i++){
16        for(int j = 1; j <= m; j++){
17            if(s1[i-1] == s2[j-1]){
18                dp[i][j] = dp[i-1][j-1]+1;
19            }
20            else
21                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
22        }
23    }
24    string res="";
25    for(int i = s1.size(), j = s2.size(); dp[i][j]>=1;){
26        if(s1[i-1]==s2[j-1]){
27            res += s1[i-1];
28            i--;
29            j--;
30        }
31        else if(dp[i-1][j]>=dp[i][j-1]) i--;
32        else j--;
33    }
34    reverse(res.begin(),res.end());
35    if(res.empty()) return "-1";
36    else return res;
```

时间限制 : C/C++ 1秒, 其他语言2秒 空间限制 : C/C++ 64M, 其他语言128M 热度指数 : 29965
本题知识点 : 整数

牛客网牛客网

题目描述

给出的32位整数x翻转。
例如x=123, 返回321
例如x=-123, 返回-321

你注意到翻转后的数可能溢出吗? 因为给出的是32位整数, 则其数值范围为 $[-2^{31}, 2^{31}-1]$ 。翻转可能会导致溢出, 如果

反转后的结果会超出原数, 0。

示例1

输入:
-123

输出:
-321

```
① C++(clang++11) ② 核心代码模式
1 class Solution {
2 public:
3     ...
4     ...
5     * @param x int 整型
6     * @return int 整型
7     */
8     int reverse(int x) {
9     //long long res; // (你不能初始化这么长啊!!!!) 脑机??
10    long long res = 0; // 8个字节累加计数变量的限制
11    while(x != 0){ // 注意这里用long long类型那个会溢出
12        res = res*10 + x%10; // 避免溢出
13        if(res >= INT_MAX || res <= INT_MIN)
14            return 0;
15    }
16    return res;
17 }
18 }
```

只能从尾部添加：

反转完从原串前面开始依次比较 串尾相同字符个数的串是否相等 如果相等返回剩余的串 就是需要添加的那个串

问题分解

- 找到最长的回文子串
- 剩余部分就是需要添加的子串

使用Naive查找，寻找最大公共串

这里用到了：翻转子串==原子串 =>回文子串

从原串的开头开始找，比较是否与翻转串的末尾相同

【这里用到了本题的特征：已有的回文子串肯定出现在末尾，不会出现在中间】

```
1 string addPalindrome(string A, int n) {
2     string s = A;
3     reverse(s.begin(),s.end()); // 取得翻转串
4     for(int i=0;i<n;i++) // Naive查找
5         if(A.substr(i,n-i)==s.substr(0,n-i))//求最长公共子串
6             return s.substr(n-i,i); //返回公共串后面剩余字符串
7     return string("");
8 }
```

若可以在任意位置 处添加：

反转原串，再直接求最大公共序列就可以了

原串长度 - 公共子序列长度

那么我们只要把给定的字符串顺序倒转与原串求最长公共子序列，再用字符串总长度减去最长公共子序列的长度就是相差的字符个数，也就是答案

len - 子序列长度

比如mem和mcm的LCS是mem或mcm

可以添加一个e凑成 memem

也可以添加一个c凑成 mcecm

abcpa ab d c d b a

在任意位置处添加和删除 其实是一样的 删除元素 得到最大回文串 都是len - 子序列长度

m e c m
m c m e

abcpa
adcba

• 算法

- 1. 动态规划： $dp[i][j]$ 表示 word1 的前 i 个字符编辑成 word2 的前 j 个字符需要的最小操作数
- 2. 初始状态： $dp[0][0] = 0$, 次删除； $dp[0][i] = i$, 次插入
- 3. 过渡公式：
 - 当字符等于字符时： $dp[i][j] = dp[i-1][j-1]$, 不需要额外操作
 - 当字符不等于字符时： $dp[i][j] = \min(insert, delete, replace)$
 - int insert = $dp[j-1] + 1$; i 个编辑成 $i-1$ 个字符，再插入一个
 - int delete = $dp[i-1][j] + 1$; $i-1$ 个编辑成 i 个字母，再删除一个
 - int replace = $dp[i-1][j-1] + 1$; $i-1$ 个编辑成 $i-1$ 个字母，再将替换换成

题目描述

给定两个字符串 str1 和 str2，再给定三个整数 ic, dc 和 rc，分别代表插入、删除和替换一个字符的代价。请输出将 str1 编辑成 str2 的最小代价。

示例1

输入
abc,*adc*,5,3,2

返回值
2

示例2

输入
abc,*adc*,5,3,100

返回值
0

牛客题库-上脚看过 > 回答结果 -

时间限制 : C/C++ 1 秒, 其他语言 2 秒 空间限制 : C/C++ 256M, 其他语言 512M 流度函数 : 27050

本题知识点 : dh ts 序列

■ 算法知识进阶讲解

题目描述

给一个01矩阵，1代表是陆地，0代表海洋，如果两个1相邻，那么这两个1属于同一个岛。我们只考虑上下左右为相邻。岛屿 相邻的陆地可以组成一个岛屿（相邻上下左右）判断岛屿的个数。

示例1

输入
1111,1,0,0,0,0,1,0,1,1,1,0,0,0,1,1,1,1,0,0,0,0,0,1,0,0,1,1,1,1

返回值
3

备注:

01矩阵范围<=200*200

```
① C++(clang++11) • 本地代码编译
8 // 动态规划求解字符串编辑距离
9 // 基于动态规划的replace cost
10 // 从前往后遍历
11 // 
12 int minEditCost(string str1, string str2, int ic, int dc, int rc) {
13     if (str1.size() < str2.size()) {
14         int m = str1.size();
15         int n = str2.size();
16         vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
17         for (int i = 0; i < m; i++) {
18             for (int j = 0; j < n; j++) {
19                 if (str1[i] == str2[j]) {
20                     dp[i][j] = dp[i-1][j-1];
21                 } else {
22                     for (int k = 0; k < n; k++) {
23                         if (str1[i] == str2[k]) {
24                             dp[i][j] = dp[i-1][k] + ic;
25                         } else {
26                             int del = dp[i-1][j] + dc;
27                             int ins = dp[i][j-1] + ic;
28                             int repl = dp[i-1][j-1] + rc;
29                             dp[i][j] = min(min(ins, del), repl);
30                         }
31                     }
32                 }
33             }
34         }
35     }
36     return dp[m][n];
```

时间限制 : C/C++ 1 秒, 其他语言 2 秒 空间限制 : C/C++ 64M, 其他语言 128M 流度函数 : 467329

本题知识点 : 矩阵

■ 算法知识进阶讲解

```
① C++(clang++11) • 本地代码编译
4 // 从前往后遍历
5 // 使用 grid 存储字符 vector<vector<char>>
6 // 使用 sum 来统计
7 // 
8 int solve(vector<vector<char>> &grid) {
9     int res = 0;
10    int n = grid.size(), m = grid[0].size();
11    for (int i = 0; i < n; i++) {
12        for (int j = i; j < n; j++) {
13            if (grid[i][j] == '1') {
14                res++;
15                dfs(grid, i, j);
16            }
17        }
18    }
19    return res;
20 }
21 void dfs(vector<vector<char>> &grid, int x, int y) {
22     grid[x][y] = 0;
23     int a1 = -1, a2 = 1, b1 = 0, b2 = 1;
24     for (int i = 1; i < 4; i++) {
25         int a = a1 + i * a2, b = b1 + i * b2;
26         int n = x + a, int m = y + b; // 这里改用 a, b, n, m 会下标越界引起报错
27         if (n >= 0 && n < grid.size() && m >= 0 && m < grid[0].size()
28             && grid[n][m] == '1') {
29             dfs(grid, n, m);
30         }
31     }
32 }
```

题目限制

输入两个链表，找出它们的第一个公共节点。（注意因为传入数据是链表，所以错误测试数据的提示是用其他方式显示的，保证传入的数据是正确的）

说明：本题目包含复杂数据结构 ListNode，[点击查看相关信息](#)

~关联企业

字节跳动 商汤科技 新华智云 昆仑万维

~关联职位

测试 研发 算法 前端

```
① C++(clang++11) • 本地代码编译
1 / 
2 struct ListNode {
3     ~ListNode();
4     struct ListNode *next;
5     ListNode(int x) {
6         val = x;
7         next = NULL;
8     }
9 };
10 class Solution {
11 public:
12     ListNode* findFirstCommonNode(ListNode* pHead1, ListNode* pHead2) {
13         ListNode *p1 = pHead1;
14         ListNode *p2 = pHead2;
15         if (!pHead1 || !pHead2) return nullptr;
16         while (p1 != p2) {
17             if (p1 == NULL)
18                 p1 = p1->next;
19             else
20                 p1 = p1->next;
21             if (p2 == NULL)
22                 p2 = p2->next;
23             else
24                 p2 = p2->next;
25         }
26         return p1; // 我走过的路 你走过的路 原来就相遇了
27     }
28 }
```

题解 4 | 开源博客 | 离线

```

1 // map解法:
2 // 利用map的唯一键性质, 先把链表1放入map中
3 // 再依次把链表2放进去, 返回第一个值为2的键即可
4
5 class Solution {
6 public:
7     ListNode* FindFirstCommonNode( ListNode* pHead1, ListNode* pHead2 ) {
8         unordered_map<ListNode*, int> mp;
9         //将pHead1的所有结点放入map中
10        while(pHead1){
11            mp[pHead1]++;
12            pHead1 = pHead1->next;
13        }
14        while(pHead2){
15            if(mp[pHead2] == 2)
16                return pHead2; //如果存在, 则返回
17            pHead2 = pHead2->next;
18        }
19        return NULL;
20    }
21 }

```

声明函数指针是必须指定函数的 **返回值类型与参数类型** 表明这个指针, 只能指向这种形式的函数

使用函数指针 传递实参的时候是直接传递函数指针 (地址) 或函数名 (地址), 不用传递函数内的参数

在class类中的sort函数的参数cmp, 定义时要声明为static, 那么背后的原因是什么呢?

这是跟sort函数的定义有关系的, 下面是sort函数的源码 [官方文档](#)

```

1 template <class RandomAccessIterator, class Compare>
2 void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);

```

可以知道其实我们写参数cmp时, 是把函数名作为实参传递给了sort函数, 而sort函数内部是用一个函数指针去调用这个cmp函数的 (建议先看下 [一文搞懂什么是函数指针](#)), 我们知道class普通类成员函数cmp需要通过 **对象名.cmp()** 来调用, 而sort()函数早就定义好了, 那个时候哪知道你定义的是什么对象, 所以内部是直接 **cmp()** 的, 那你不加static时, 去让sort()直接用 **cmp()** 当然会报错

static静态成员函数 **不用加对象名, 就能直接访问函数** (这也是静态成员函数的一大优点) 所以加了static就不会报错

例题 沉迷单车的追风少年: 只是在class里面才需要这么写, 因为普通类成员函数, 都不能以函数指针的方式作为其他函数的入口参数, 因为普通成员函数在编译阶段, 会自动添加了入口参数, 这样这个函数指针的模板其实就改变了。 5月前 回复

题目描述

给出一组区间, 请合并所有重叠的区间。
请保证合并后的区间按区间起点升序排列。

示例1

输入

```
[[10,30), [20,60), [80,100), [150,180]]
```

返回值

```
[[10,60), [80,100), [150,180]]
```

关联企业

字节跳动

```

6 // 注意intervals::iterator, 和int(i)
7 // 注意Interval(a, int b): start(a), end(b){}
8 // ...
9 // ...
10 class Solution {
11 public:
12     static bool cmp(const Interval &a, const Interval &b){
13         return a.start < b.start;
14     }
15     vector<Interval> merge(vector<Interval> &intervals) {
16         sort(intervals.begin(), intervals.end(), cmp);
17         vector<Interval> res; // 如果是vector<vector<int>>默认排序行为
18         int i = 0, n = intervals.size(); // sort会忽略第一个元素, 相等再用第二个
19         int l, r; // 注意一个static
20         while(i < n){ // 要加一个static
21             l = intervals[i].start;
22             r = intervals[i].end;
23             while(i < n-1 && r >= intervals[i+1].start){
24                 i++;
25                 r = max(intervals[i].end, r);
26             }
27             res.push_back({l, r});
28             i++;
29         }
30         return res;
31     }

```

while(i<n) 会在循环体内++i 可以改成for(int i =0; i<n; i++) 在for内写了i++;
这种while内再套一个while 或者 for内再套一个while 并且共用一个递增变量i 的写法值得学习

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 64M, 其他语言128M 难度指数: 40246
本题知识点: 字符串

算法知识进阶讲解

题目描述

实现函数 atoi 。函数的功能为将字符串转化为整数
提示: 仔细思考所有可能的输入情况。这个问题没有给出输入的限制, 你需要自己考虑所有可能的情况。

示例1

输入

```
"123"
```

返回值

```
123
```

```

1 class Solution {
2 public:
3     //注意: std::string的编写: 主要考虑空字符, 正负号, 逗号, 非法字符等让等条件
4     int atoi(string str) {
5         string s(str);
6         int len = s.length(), flag = 1;
7         long res = 0;
8         int index = s.find_first_not_of(' ');
9         if(s[index] == '+') { s[index] = '-' }
10        if(s[index] == '-') flag = -1;
11        if (s[index+1] == '-') flag = -1;
12        else flag = 1;
13        //index++;
14        //index++;
15        for(;index<len;index++){
16            if(s[index]>='0' && s[index] <= '9'){
17                res = res *10 + (s[index] - '0');
18            }
19            if(res*flag >= INT_MAX) return INT_MAX;
20            if(res*flag <= INT_MIN) return INT_MIN;
21        }
22        else break;
23    }
24    if()
25    return res * flag;
26 }

```

时间限制 : C/C++ 1秒, 其他语言2秒 空间限制 : C/C++ 64M, 其他语言128M 热度指数 : 19091
 本题知识点 : 位运算 数组 数学 二分
■ 算法知识视频讲解

题目描述

从0,1,2,...,n这n+1个数中选择n个数, 找出这n个数中缺失的那个数, 要求O(n)尽可能小。

示例1

输入

```
[0,1,2,3,4,5,7]
```

返回值

```
6
```

示例2

输入

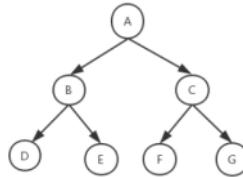
```
1
```

输出

```
int solve(vector<int>& a) {
    // write code here
    map<int,int> mp;
    for(auto i : a)
        mp[i]++;
    for(int i = 0; i <= a.size(); i++)
        if(mp[i] == 0)
            return i;
    }
}

//输入是有序的, 二分查找-但是不适合缺失元素出现在末尾处-可以加一个判断处理
int solve(vector<int>& a) {
    int l = 0, r = a.size() - 1;
    while(l < r) {
        int mid = (l+r)/2;
        if(a[mid] == mid) l = mid + 1;
        else if (a[mid] > mid) r = mid;
        ...
    }
    if(a[l] == l) return l+1;
    else return l;
}
```

高度为h, 由 2^h-1 个节点构成的二叉树称为满二叉树。



度 : 指的是一个节点拥有子节点的个数。如二叉树的节点的最大度数为2。

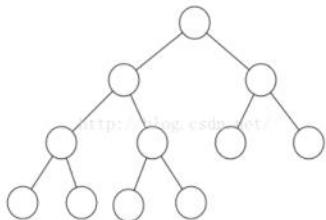
深度 : 数的层数, 根节点为第一层, 依次类推。

叶子节点 : 度为0的节点, 即没有子节点的节点。

树 : 树中的每一个节点, 可以有n(后续节点)个子节点, 但是每个节点只有一个前驱节点。

二叉树 : 除了叶子节点外, 每个节点只有两个分支, 左子树和右子树, 每个节点的最大度数为2。

满二叉树 : 除了叶结点外每一个结点都有左右子叶且叶结点都处在最底层的二叉树,



完全二叉树 : 只有最下面的两层结点度小于2, 并且最下面一层的结点都集中在该层最左边的若干位置的二叉树。
 也就是说, 在满二叉树的基础上, 我在最底层从右往左删去若干节点, 得到的都是完全二叉树。

所以说, 满二叉树一定是完全二叉树, 但是完全二叉树不一定是满二叉树

平衡二叉树 : 树的左右子树的高度差不超过1的数, 空树也是平衡二叉树的一种。

平衡二叉树, 又称AVL树。它或者是一棵空树, 或者是具有下列性质的二叉树 : 它的左子树和右子树都是平衡二叉树, 且左子树和右子树的高度之差的绝对值不超过1。

常用算法有 : [红黑树](#)、[AVL树](#)、[Treap等](#)。

平衡二叉树的调整方法

平衡二叉树是在构造二叉排序树的过程中, 每当插入一个新结点时, 首先检查是否因插入新结点而破坏了二叉排序树的平衡性, 若是, 则找出其中的最小不平衡子树, 在保持二叉排序树特性的前提下, 调整最小不平衡子树中各结点之间的链接关系, 进行相应的旋转, 使之成为新的平衡子树。

哈夫曼树 : 带权路径长度达到最小的二叉树, 也叫做最优二叉树。

不关心树的结构, 只要求带权值的路径达到最小值, 哈夫曼树可能是完全二叉树也可能是满二叉树。

时间限制 : C/C++ 5秒, 其他语言10秒 空间限制 : C/C++ 256M, 其他语言512M 热度指数 : 18832
 本题知识点 : 树 ds
■ 算法知识视频讲解

题目描述

给定一棵二叉树, 已经其中没有重复值的节点, 请判断该二叉树是否为搜索二叉树和完全二叉树。

示例1

输入

```
(2,1,3)
```

返回值

```
{true,true}
```

备注:

$n \leq 500000$

说明 : 本题自包含复杂数据结构TreeNode, [点击查看相关信息](#)

C++(clang++11) ✓ ● 本地代码模式

```
vector<bool> ans;
ans[0] = dfs(root);
ans[1] = bfs(root);
return ans;
}

bool dfs(TreeNode *root) {
    if(!root) return true;
    if(root->left && root->left->val > root->val) return false;
    if(root->right && root->right->val < root->val) return false;
    return dfs(root->left) && dfs(root->right);
}

bool bfs(TreeNode *root) {
    queue<TreeNode*> q;
    q.push(root);
    while(!q.empty()) {
        int nnn = q.size();
        for(int i = 0; i < q.size(); i++) {
            TreeNode *tmp = q.front();
            q.pop();
            if(tmp->left && tmp->right) {
                if(tmp->left->val > tmp->right->val) return false;
            }
            if(tmp->left) q.push(tmp->left);
            if(tmp->right) q.push(tmp->right);
        }
    }
    return true;
}
```

■ 算法知识视频讲解**题目描述**

假设你有一个数组，其中第 i 个元素是股票在第 i 天的价格。

你有一次买入和卖出的机会。（只有买入了股票以后才能卖出）。请你设计一个算法来计算可以获得的最大收益。

示例1

输入

[1, 4, 2]

复制

返回值

3

复制

题目描述

编写一个函数来验证输入的字符串是否有效的 IPv4 或 IPv6 地址。

IPv4 地址由十进制数和点表示，每个地址包含4个十进制数，范围为 0–255，用“.”分隔。比如，172.16.254.1；同时，IPv4 地址的数不会以 0 开头。比如，地址 172.16.254.01 是不合法的。

IPv6 地址由16进制的数字表示，每组表示 16 比特。这些组数字通过“:”分隔。比如，2001:db8:85a3:0:0:0:0:7334 是一个有效的地址。

2001:db8:85a3:0:0:0:0:7334 是一个有效的 IPv6 地址。但是，2001:db8:85a3:0:0:0:0:7334 是一个无效的 IPv6 地址，忽略 0 开头，忽略大小写。

然而，我们不能因为某个值为 0，而使用一个空的冒号，以至于出现 (:) 的情况。比如，

2001:db8:85a3::82E:0370:7334 是无效的 IPv6 地址。

同时，在 IPv6 地址中，多余的 0 也是不被允许的。比如，2001:db8:85a3:0:0:0:0:7334 是无效的。

说明：你可以认为给定的字符串里面没有空格或者其他特殊字符。

示例1

输入

"172.16.254.1"

复制

返回值

"IPv4"

复制

题目描述

求给定二叉树的最大深度。

最大深度是指树的根结点到最远叶子节点的最长路径上结点的数量。

示例1

输入

(1, 2)

复制

返回值

2

复制

递归的方法

利用队列进行层序遍历

■ 算法知识视频讲解**题目描述**

求给定二叉树的最大深度。

最大深度是指树的根结点到最远叶子节点的最长路径上结点的数量。

示例1

输入

(1, 2)

复制

返回值

2

复制

示例2

```
1 class Solution {
2 public:
3     ...
4     int maxProfit(vector<int>& prices) {
5         ...
6         // write code here
7         int minn = INT_MAX;
8         int ans = INT_MIN;
9         for(int i = 0; i<prices.size(); i++) {
10            minn = min(minn, prices[i]);
11            ans = max(ans,prices[i]-minn);
12        }
13        return ans;
14    }
15 }
```

```
1 /**
2  * @param IP string 表示一个IP地址
3  * @return string
4  */
5 string solve(string IP) {
6     // write code here
7     int len = IP.size();
8     string ans;
9     if (len >= 7 && len <= 15) {
10         ans = IPVALID(IP);
11     } else {
12         ans = IPV6VALID(IP);
13     }
14     return ans;
15 }
16 string IP4VALID(string IP) {
17     int start = 0;
18     for (int i = 0; i < IP.size(); i++) {
19         while (IP[i] != ':' && i < IP.size())
20             i++;
21         if (i - start > 4 || IP[i + 1] == ':')
22             return "Neither";
23         for (int j = start; j < i; j++)
24             if ((IP[j] >= '0' && IP[j] <= '9') || (IP[j] >= 'A' && IP[j] <
25             ...
26             );
27         else
28             return "Neither";
29         start = i + 1;
30     }
31     return "IPv4";
32 }
```

```
5     struct TreeNode* right;
6     ...
7     ...
8 }
9 class Solution {
10 public:
11     ...
12     ...
13     * @param root TreeNode*
14     * @return int
15     ...
16     int maxDepth(TreeNodes* root) {
17         ...
18         if (root == NULL) return 0;
19         return max(maxDepth(root->left),maxDepth(root->right)) +1;
20     }
21 }
```

给定一个二叉树，编写一个函数来获取这个树的最大宽度。树的宽度是所有层中的最大宽度。这个二叉树与满二叉树（full binary tree）结构相同，但一些节点为空。

每一层的宽度被定义为两个节点（该层最左和最右的非空节点，两端指向的 null 节点由计入长度）之间的长度。

示例 1：



输出：4

解释：最大值出现在树的第 3 层，宽度为 4 (5,3,null,9)。

示例 2：

输入：

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M 热度指数：32680

本题知识点： 数组 二分

算法知识视频讲解

题目描述

给出一个转动过的有序数组，你事先不知道该数组转了多少

(例如 0 1 2 4 5 6 7 可能变为 4 5 6 7 0 1 2)

在数组中搜索给出的目标值，如果能在数组中找到，返回它的索引，否则返回 -1。

假设数组中不存在重复项。

示例 1：

输入：

[1,0]

返回值：

-1

示例 2：

```

12 public:
13     int widthOfBinaryTree(TreeNode* root) {
14         if (root == nullptr) return 0;
15         // 保存最大的宽度
16         int res = 0; // 从树的广度优先遍历
17         queue q;
18         root->left = 1; // 对于根节点的编号
19         q.push(root);
20         while (!q.empty()) { // 基于自定义队头和尾获得当前层的宽度
21             res = max(res, q.back()-val - q.front()-val + 1);
22             int offset = q.front()-val; // 遍历宽小的差值
23             int n = q.size(); // 遍历宽大的差值
24             for (int i = 0; i < n; ++i) {
25                 TreeNode* curr = q.front();
26                 q.pop(); // 增小数
27                 curr->val += offset;
28                 if (curr->left) {
29                     curr->left = curr->val;
30                     q.push(curr->left);
31                 }
32                 if (curr->right) {
33                     curr->right = curr->val + 1;
34                     q.push(curr->right);
35                 }
36             }
37         }
38     }
39     return res;
40 }
  
```

① C++(clang++11) ● 标C代码模式

```

3 // Problem A: int search(vector<int> &A, int target) {
4 // write code here
5 int search(int& A, int n, int target) {
6     // write code here
7     int left = 0, right = n-1; int mid;
8     while(left <= right) {
9         mid = left + (right-left)/2;
10        if(A[mid] == target)
11            return mid;
12        if(A[mid] > A[left]){// 左侧有序(含A[left])
13            if(target <= A[left] && target <= A[mid])
14                right = mid-1;
15            else
16                left = mid+1;
17        }
18        else{// 右侧有序(含A[mid])
19            if(target > A[mid] && target <= A[right])
20                left = mid+1;
21            else
22                right = mid-1;
23        }
24    }
25    return -1;
26 }
27
28
29
30
31
  
```

牛客网-上周通过，在二叉树中找到两个节点的最近公共祖先...

上一题 | 下一题 | 我的提交

算法知识视频讲解

题目描述

给定一棵二叉树以及这棵树上的两个节点 o1 和 o2，请找到 o1 和 o2 的最近公共祖先。

示例 1：

输入：

[3,5,1,6,2,0,8,#,#,7,4],5,1

返回值：

3

说明：本题目包含复杂数据结构TreeNode，[点击查看相关信息](#)

关联企业

字节跳动 七牛云 京东数科 腾讯

① C++(clang++11) ● 标C代码模式

```

3 // Problem A: int lowestCommonAncestor(TreeNode* root, int o1, int o2) {
4 // write code here
5 int lowestCommonAncestor(TreeNode* root, int o1, int o2) {
6     // write code here
7     if(root == nullptr) return -1;
8     if(root->val == o1 || root->val == o2) return root->val;
9     int left = lowestCommonAncestor(root->left, o1, o2);
10    int right = lowestCommonAncestor(root->right, o1, o2);
11    if(left == -1 && right == -1) return root->val;
12    if(left == -1 && right != -1) return right;
13    if(left != -1 && right == -1) return left;
14    else return -1;
15 }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
  
```

从图中，大家可以看到，我们是如何回溯遍历整颗二叉树，将结果返回给头结点的！

整体代码如下：

```

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (root == q || root == p || root == NULL) return root;
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        if (left != NULL && right != NULL) return root;

        if (left == NULL && right != NULL) return right;
        else if (left != NULL && right == NULL) return left;
        else { // (left == NULL && right == NULL)
            return NULL;
        }
    }
};
  
```

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 64M, 其他语言128M 热度指数: 436979
本题知识点: 排序 双端队列

算法知识进阶理解

题目描述

给定一个数组和滑动窗口的大小，找出所有滑动窗口内数值的最大值。例如，如果输入数组[2,3,4,2,6,2,5,1]及滑动窗口的大小3，那么一共有8个滑动窗口，他们的最大值分别为4,4,6,6,5；针对数组[2,3,4,2,6,2,5,1]的滑动窗口有以下8个：[2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1]。窗口大于数组长度的时候，返回空。

示例1

输入
[2,3,4,2,6,2,5,1], 3

返回值
[4,4,6,6,5]

```
C++(clang++11) - 本地代码模式
1 class Solution {
2 public:
3     vector<int> maxInWindows(const vector<int>& num, unsigned int size) {
4         vector<int> res;
5         int n = num.size();
6         int max = INT_MIN;
7         if(size == 0 || num.empty() || n < size) return res;
8         for(int i = 0; i < n - size + 1; i++) {
9             for(int j = i; j < i + size; j++) {
10                 max = max(num[j], max);
11             }
12         }
13         res.push_back(max);
14         max = INT_MIN;
15     }
16     return res;
17 }
18 }
```

牛客网刷题上刷过 > 滑动窗口的最大值

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 64M, 其他语言128M 热度指数: 436979
本题知识点: 排序 双端队列

算法知识进阶理解

题目描述

给定一个数组和滑动窗口的大小，找出所有滑动窗口内数值的最大值。例如，如果输入数组[2,3,4,2,6,2,5,1]及滑动窗口的大小3，那么一共有8个滑动窗口，他们的最大值分别为4,4,6,6,5；针对数组[2,3,4,2,6,2,5,1]的滑动窗口有以下8个：[2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1]。窗口大于数组长度的时候，返回空。

示例1

输入
[2,3,4,2,6,2,5,1], 3

返回值
[4,4,6,6,5]

关联企业

```
C++(clang++11) - 本地代码模式
1 class Solution {
2 public:
3     vector<int> maxInWindows(const vector<int>& num, unsigned int size) {
4         vector<int> ret;
5         if (num.size() == 0 || size < 1 || num.size() < size) return ret;
6         int n = num.size();
7         deque<int> dq;
8         dq.push_back(0);
9         for (int i = 0; i < n - size + 1; i++) {
10             while (!dq.empty() && num[dq.back()] < num[i]) {
11                 dq.pop_back();
12             }
13             dq.push_back(i);
14             // 判断是否形成了窗口
15             if (dq.front() + size == i) {
16                 dq.pop_front();
17             }
18             // 判断是否形成了窗口
19             if (i >= size - 1) {
20                 ret.push_back(num[dq.front()]);
21             }
22         }
23         return ret;
24     }
25 }
26 }
```

牛客网刷题上刷过 > 滑动窗口的最大值

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 256M, 其他语言512M
热度指数: 6578
本题知识点: 排序 双端队列

算法知识进阶理解

题目描述

给定一个数组和滑动窗口的大小，找出所有滑动窗口内数值的最大值。例如，如果输入数组[2,3,4,2,6,2,5,1]及滑动窗口的大小3，那么一共有8个滑动窗口，他们的最大值分别为4,4,6,6,5；针对数组[2,3,4,2,6,2,5,1]的滑动窗口有以下8个：[2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1], [2,3,4,2,6,2,5,1]。窗口大于数组长度的时候，返回空。

示例1

输入
[2,3,4,2,6,2,5,1], 3

返回值
[4,4,6,6,5]

```
C++(clang++11) - 本地代码模式
1 class Solution {
2 public:
3     vector<int> maxInWindows(const vector<int>& num, unsigned int size) {
4         vector<int> res;
5         int n = num.size();
6         if(size == 0 || size > n || n < 0) return res;
7         deque<int> dq;
8         for(int i = 0; i < n; i++){
9             while(!dq.empty() && num[dq.back()] < num[i])
10                 dq.pop_back();
11             dq.push_back(i); // 将存在的元素i加入下标
12             if(dq.front() + size == i)
13                 dq.pop_front();
14             if(i >= size-1)
15                 res.push_back(num[dq.front()]);
16         }
17         return res;
18     }
19 }
```

牛客网刷题上刷过 > 输出二叉树的右视图

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 256M, 其他语言512M
热度指数: 6578
本题知识点: 树

算法知识进阶理解

题目描述

请根据二叉树的前序遍历，中序遍历恢复二叉树，并打印出二叉树的右视图

示例1

输入
[1,2,4,5,3],[4,2,5,1,3]

返回值
[1,3,5]

```
C++(clang++11) - 本地代码模式
1 class Solution {
2 public:
3     vector<int> rightSideView(vector<vector<int>> pre, vector<int>& inorder, int iStart, int iEnd) {
4         if(iStart > iEnd) return NULL;
5         TreeNode *root = new TreeNode(pre[0]);
6         int k = 0;
7         for(int i = 0; i < inorder.size(); i++) {
8             if(pre[0] == inorder[i]) {
9                 k++;
10            root->left = create(pre, start+1,inorder,iStart,k-1);
11            root->right = create(pre,start + k-iStart + 1, inorder, k+1, iEnd);
12        }
13    }
14    return root;
15 }
16 }
```

```
vector<int> solve(vector<int>& xianxu, vector<int>& zhongxu) {
    // write code here
    int n = xianxu.size();
    TreeNode *root = create(xianxu, 0, zhongxu, 0, n-1);
    vector<int> res;
    if(!root) return res;
    queue<TreeNode*> que;
    que.push(root);
    while(!que.empty()){
        int n = que.size();
        for(int i = 0; i < n; i++){
            TreeNode *node = que.front();
            que.pop();
            if(i == n-1){ // if语句加个 ; 你真是个XX
                res.push_back(node->val);
            }
            if( node->left)
                que.push(node->left);
            if( node->right)
                que.push(node->right);
        }
    }
    return res;
}
```

首先，假设有两个单词 "abcd" 和 "abcde"，那么在一本涵盖了所有字母排列的字典中，

这两个单词之间的单词也一定有前缀 "abcd"（可以看做26进制的数）

于是，我们就只需要比较最小字典序的字符串和最大字典序的字符串

那么 sort 一遍数组，然后取 strs[0] 与 strs[n-1]，比较求解即可

牛客竞赛-上周考过 > 最长公共前缀 > 相似的企业真题 < 上一题 | 下一题 > 我的提交

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M
难度指数：2329
本题知识点：字符串

算法知识视频讲解

题目描述

编写一个函数来查找字符串数组中的最长公共前缀。

示例1

输入

```
{ "abce", "abc", "abca", "abc", "abcc" }
```

返回值

```
"abc"
```

```
C++(clang++11) 核心代码模式
```

```
1 // 先对字符串排序，然后先比较第一个和最后一个的首字符，这两个字符必定是差值最大的两个
2 // 因为排序是先从第一个开始排的。如果这两个不等，跳出循环，否则，说明所有字符串的首
3 // 字符相等，那么接着判断第二个。
4 class Solution {
5 public:
6     string longestCommonPrefix(vector<string> &strs) {
7         if (!strs.size()) return "";
8         sort(strs.begin(), strs.end());
9         int i = 0, sz = strs.size(), l = min(strs[0].size(), strs[sz - 1].size());
10        for (i = 0; i < l;)
11            if (strs[0][i] == strs[sz - 1][i])
12                return strs[0].substr(0, i);
13            cout << strs[0] << endl;
14            cout << strs[sz - 1];
15        return strs[0] / substr(0, i);
16    }
17}
```

牛客竞赛-上周考过 > 排序 > 相似的企业真题 < 上一题 | 下一题 > 我的提交

题目描述

给定一个数组，请编写一个函数，返回该数组排序后的形式。

示例1

输入

```
[5,2,3,1,4]
```

返回值

```
[1,2,3,4,5]
```

示例2

输入

```
[5,1,3,2,5]
```

返回值

```
[1,2,3,5,6]
```

```
C++(clang++11) 核心代码模式
```

```
1 class Solution {
2 public:
3     ...
4     // 代码中的注释、方法名、参数名已经做过，请勿修改，直接通过方法执行的结果即可
5     // 请将正确的回答
6     // 注意：arr 是类型 vector<int> 的指针
7     // 注意：arr 是类型 vector<int> 的指针
8     void quick_sort(vector<int> &arr, int left, int right){
9         if (left > right)
10             return;
11         int pos = arr[left];
12         int l = left, r = right;
13         while (l < r)
14             while (l < r) arr[l] >= arr[right] -> l--;
15             while (l < r) arr[r] <= arr[left] -> r++;
16         swap(arr[l], arr[r]);
17         quick_sort(arr, left, l-1);
18         quick_sort(arr, l+1, right);
19     }
20     vector<int> MySort(vector<int> &arr) {
21         // 写出你的代码
22         quick_sort(arr, 0, arr.size() - 1);
23         return arr;
24     }
25 }
```

牛客竞赛-上周考过 > 建二叉树 > 相似的企业真题 < 上一题 | 下一题 > 我的提交

时间限制：C/C++ 1秒，其他语言2秒 空间限制：C/C++ 64M，其他语言128M
难度指数：118535
本题知识点：树 dia 数组

算法知识视频讲解

题目描述

输入某二叉树的前序遍历和中序遍历的结果，请输出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列[1,2,4,7,3,5,6,8]和中序遍历序列[4,7,2,1,5,3,6,8]，则重建二叉树并返回。

示例1

输入

```
[1,2,3,4,5,6,7], [3,2,4,1,6,5,7]
```

返回值

```
[1,2,5,3,4,6,7]
```

说明：本题目包含复杂数据结构TreeNode，[点此查看相关信息](#)

```
C++(clang++11) 核心代码模式
```

```
1 struct TreeNode {
2     int val;
3     *TreeNode* left;
4     *TreeNode* right;
5     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
6     ~TreeNode();
7 };
8
9
10 class Solution {
11 public:
12     *TreeNode* build_tree(vector<int> &pre, int pstart, vector<int> &inord, int istart, int iend)
13     {
14         if (istart > iend)
15             return NULL;
16         int k = 0;
17         for (int i = 0; i < inord.size(); i++)
18             if (inord[i] == pre[pstart])
19                 k = i;
20         *TreeNode* root = new TreeNode(pre[pstart]);
21         root->left = build_tree(pre, pstart + 1, inord, istart, k - 1);
22         root->right = build_tree(pre, pstart + k - istart + 1, inord, k + 1, iend);
23         return root;
24     }
25
26     *TreeNode* reConstructBinarytree(vector<int> &pre, vector<int> &in)
27     {
28         *TreeNode* root = build_tree(pre, 0, in, 0, in.size() - 1);
29         return root;
30     }
31 }
```

题目描述

分别按照二叉树先序、中序和后序打印所有的节点。

示例1

输入

```
(1,2,3)
```

返回值

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

备注：

$n \leq 10^6$

```
C++(clang++11) 核心代码模式
```

```
25
26     private:
27     void preorder(TreeNode *root, vector<int> &ret){
28         if (root){
29             ret.push_back(root->val);
30             preorder(root->left, ret);
31             preorder(root->right, ret);
32         }
33     }
34     void inorder(TreeNode *root, vector<int> &ret){
35         if (root){
36             inorder(root->left, ret);
37             ret.push_back(root->val);
38             inorder(root->right, ret);
39         }
40     }
41     void lastorder(TreeNode *root, vector<int> &ret){
42         if (root){
43             lastorder(root->left, ret);
44             lastorder(root->right, ret);
45             ret.push_back(root->val);
46         }
47 }
```

```

/*
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
class Solution {
public:
    ...
    vector<vector<int>> threeOrders(TreeNode* root) {
        // Write code here
        vector<int> pre, in, last;
        preorder(root, pre);
        inorder(root, in);
        lastorder(root, last);
        vector<vector<int>> res{pre, in, last};
        return res;
    }
};

```

题目描述

对于一个给定的字符串，我们需要在线性(也就是O(n))的时间里对它做一些变换。首先这个字符串中包含着一些空格，就像“Hello Word”一样，然后我们想要把字符串中的字符都由大写替换成小写，同时反转每个字符串的大小写。比如“Hello Word”变形后就变成了“wORLD”。

输入描述：

给定一个字符串s以及它的长度n(1≤n≤500)

返回描述：

请返回变形后的字符串，她归宿给定的字符串均由大小写字母和空格构成。

示例1

输入

```
"This is a sample",16
```

返回值

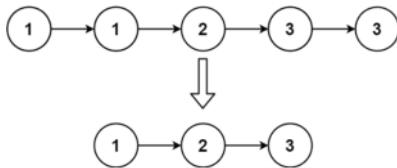
```
*SAMPLE A IS THIS*
```

```

1 class Solution {
2 public:
3     char transform (char ch){
4         if(ch>'z' || ch <='a')
5             return ch - 'a' + 'A';
6         else
7             return ch - 'A' + 'a';
8         //字符转换成大写，并反转大小写。
9         //从后往前，加进结果，说明从一个空格的后面到下一个空格之间是一个单词
10    string transform(string s, int n) {
11        int p = n-1;
12        string res;
13        for(int i = n-1; i >= 0; i--) {
14            if(s[i] == ' '){
15                for(int j = i+1; j <= p; j++)
16                    res += transform(s[j]);
17                res += ' ';
18                //从i+1 = 1到i
19                //i = n
20                p = i-1; //p--(错了！)
21            }
22        }
23        for(int j = 0; j <= p; j++)
24            res += transform(s[j]);
25        return res;
26    }
27}

```

-----保留出现一次



```

class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) {
            return head;
        }

        ListNode* cur = head;
        while (cur->next) {
            if (cur->val == cur->next->val) {
                cur->next = cur->next->next;
            }
            else {
                cur = cur->next;
            }
        }

        return head;
    }
};

```

题目描述

给出一个升序排序的链表，删除链表中的所有重复出现的元素，只保留链表中只出现一次的元素。

例如：

给出的链表为1 → 2 → 3 → 3 → 4 → 4 → 5, 返回1 → 2 → 5.

给出的链表为1 → 1 → 1 → 2 → 3, 返回2 → 3.

示例1

输入

```
(1,2,2)
```

返回值

```
(1)
```

```

1 class Solution {
2 public:
3     ListNode *deleteDuplicates(ListNode *head) {
4         if(head) return NULL;
5         ListNode *pHead = new ListNode();
6         pHead->next = head;
7         ListNode *cur = pHead;
8         while(cur->next && cur->next->next) {
9             ListNode *p = cur->next;
10            ListNode *pNext = cur->next->next;
11            if(p->val != pNext->val)
12                cur = p;
13            else{ //如果p->next->next不为空，且p->val为最末一个，仍与后面相同
14                while(pNext && pNext->val == p->val)
15                    pNext = pNext->next; //若要输出链表，将curr->next指向p
16                cur->next = pNext; //将链表的结点 所有置想法都删掉
17            }
18        }
19        return pHead->next;
20    }
21};
22

```

力扣 学习 题库 讨论 竞赛 求职 商店

题目描述 评论 (272) 解题 (554) 提交记录

剑指 Offer 30. 包含min函数的栈

难度：简单 通过率：123 / 123

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数在该栈中，调用 min、push 及 pop 的时间复杂度都是 O(1)。

示例：

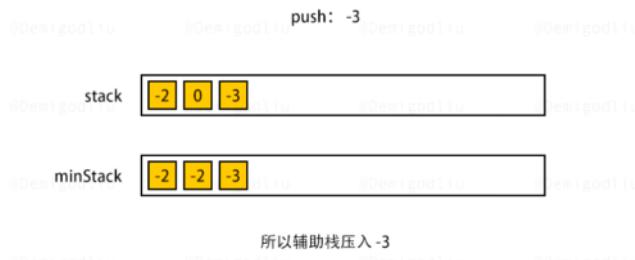
```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.min(); --> 返回 -3.
minStack.pop();
minStack.top(); --> 返回 0.
minStack.min(); --> 返回 -2.
```

提示：

- 各函数的调用总次数不超过 20000 次

```
1 class MinStack {
2     stack<int> min_stack;
3     stack<int> x_stack;
4 public:
5     /* initialize your data structure here. */
6     MinStack() {
7         min_stack.push(INT_MAX);
8     }
9
10    void push(int x) {
11        x_stack.push(x);
12        min_stack.push(std::min(min_stack.top(), x));
13    }
14
15    void pop() {
16        x_stack.pop();
17        min_stack.pop();
18    }
19
20    int top() {
21        return x_stack.top();
22    }
23
24    int min() {
25        return min_stack.top();
26    }
27 }
28
29 /*
```

图解演示



所以辅助栈压入 -3

stack

-2 0 -3

minStack

-2 2 -3

所以辅助栈压入 -3

只用记住一点，在辅助栈中，存放着每一位主栈元素对应的小值。

牛客网-上局考过 > 设计getMin功能的栈

时间限制 : C/C++ 2秒 , 其他语言4秒 空间限制 : C/C++ 256M , 其他语言512M 热度指数 : 11425

本题知识点 : 栈

算法知识视频讲解

题目描述

实现一个特殊功能的栈，在实现栈的基本功能的基础上，再实现返回栈中最小元素的操作。

示例1

输入

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

返回值

```
[1,2]
```

备注:

有三种操作种类，op1表示push，op2表示pop，op3表示getMin。你需要返回和op3出现次数一样多的数据，表示每次getMin的答案。

```
1 C++(clang++11) * 换行代码模式
2
3     stack<int> x_stack, min_stack;
4     //min_stack.push(INT_MAX); 要放到函数体里 在外面这样直接push不允许 虽然是成员变量也不是成员
5     vector<int> getMinStack(vector<vector<int>> > op) {
6         // write code here
7         vector<int> res;
8         min_stack.push(INT_MAX);
9         for (int i = 0; i < op.size(); i++) {
10             if (op[i][0] == 1) //如果vec第5个元素为1 代表push操作
11                 Push(op[i][1]); //将vec第1个元素 进行push
12             else if (op[i][0] == 2) //vec第5个元素为2 代表要执行pop操作
13                 Pop();
14             else //vec第5个元素为3 代表要执行getMin操作
15                 res.push_back(getMin());
16         }
17         return res;
18     }
19
20     void Push(int x) {
21         x_stack.push(x);
22         min_stack.push(min(min_stack.top(),x));
23     }
24
25     void Pop() {
26         x_stack.pop();
27         min_stack.pop();
28     }
29
30     int getMin(){
31         return min_stack.top();
32     }
33 }
```

力扣 学习 题库 讨论 竞赛 求职 商店

题目描述 评论 (488) 解题 (609) 提交记录

剑指 Offer 62. 圆圈中最后剩下的数字

难度：简单 通过率：348 / 348

0,1,...n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字（删除后从下一个数字开始计数）。求出这个圆圈里剩下的最后一个数字。

例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

示例 1：

```
输入: n = 5, m = 3
输出: 3
```

```
1 class Solution {
2 public:
3     int lastRemaining(int n, int m) {
4         int result = 0;
5         for (int i = 2; i <= n; ++i) {
6             result = (result + m) % i;
7         }
8         return result;
9     }
10 };
11 /*
12 最终剩下一个人时的安全位置肯定为0，反推安全位置在人数为n时的编号
13 人数为1: 0
14 人数为2: (0+m) % 2
15 人数为3: ((0+m) % 2 + m) % 3
16 人数为4: (((0+m) % 2 + m) % 3 + m) % 4
17 */
```

题目描述

编号为 1 到 n 的 n 个人围成一圈，从编号为 1 的人开始报数，报到 m 的人离开。下一个人继续从 1 开始报数。 $n - 1$ 轮结束以后，只剩下一个，问最后留下的这个人编号是多少？

示例1

输入
5, 2
返回值
3

```
① C++(clang++11) ✓ 核心代码模式
1 class Solution {
2 public:
3     /**
4      * @param n: int整型
5      * @param m: int整型
6      * @return: int整型
7      */
8     int lastRemaining(int n, int m) {
9         int result = 0;
10        for (int i = 2; i <= n; ++i) {
11            result = (result + m) % i;
12        }
13        return result+1;
14    }
15}
```

```
ListNode* createList(int n) {
    ListNode *head = new ListNode(1);
    ListNode *p = head;
    for(int i = 2; i <= n; i++) {
        ListNode *node = new ListNode(i);
        p->next = node;
        p = node;
    }
    p->next = head;
    return head;
}
int lastRemaining(int n, int m) {
    ListNode *head = createList(n);
    ListNode *p = head, *pre = NULL;
    while(p->next != p) {
        for(int i = 1; i < m; i++) {
            pre = p;
            p = p->next;
        }
        // 剪枝
        pre->next = p->next;
        delete p;
        p = pre->next;
    }
    return p->val;
}

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

题目描述 评论 (1.3k) 题解 (1.5k) 提交记录

234. 回文链表

难度 简单 940 收藏 分享

请判断一个链表是否为回文链表。

示例 1：

输入: 1->2
输出: false

示例 2：

输入: 1->2->2->1
输出: true

用栈实现

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        stack<int> s;
        ListNode *p = head;
        while(p){
            s.push(p->val);
            p = p->next;
        }
        p = head;
        while(p){
            if(p->val != s.top()){
                return 0;
            }
            s.pop();
            p = p->next;
        }
        return 1;
    }
};
```

快慢指针实现

```
① C++ 智能模式
9     * };
10    */
11    class Solution {
12 public:
13     bool isPalindrome(ListNode* head) {
14         vector<int> res;
15         while(head != nullptr){
16             res.emplace_back(head->val);
17             head = head ->next;
18         }
19         for(int i = 0, j = res.size()-1; i < j; i++,j--)
20             if(res[i] != res[j])
21                 return false;
22         return true;
23     }
24 }
```

```

class Solution {
public:
    bool isPalindrome(ListNode* head) {
        ListNode* q = head; //快指针
        ListNode* cur = head, * pre = NULL; //反转链表的模板
        while(q && q->next) {
            q = q->next->next; //2倍慢指针的速度
            ListNode* temp = cur->next; //反转链表的模板
            cur->next = pre;
            pre = cur;
            cur = temp;
        }
        if(q) cur = cur->next; //奇数链表处理
        while(pre) { //开始对比
            if(pre->val != cur->val) return false;
            pre = pre->next;
            cur = cur->next;
        }
        return true;
    }
};

```

题目描述 | 评论 (363) | 题解 (601) | 提交记录

剑指 Offer 55 - II. 平衡二叉树

难度 简单 | 138 | 通过 | 贡献 | 回复

输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。

示例 1：

给定二叉树 [3,9,20,null,null,15,7]

```

3
 / \
9  20
 / \
15  7

```

返回 true。

示例 2：

给定二叉树 [1,2,2,3,3,null,null,4,4]

```

1
 / \

```

剑指 Offer 33. 二叉搜索树的后序遍历序列

难度 中等 | 243 | 通过 | 贡献 | 回复

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 true，否则返回 false。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

```

5
 / \
2  6
 / \
1  3

```

示例 1：

输入：[1,6,3,2,5]
输出：false

```

bool isBST(vector<int>& post, int l ,int r){
    //第一步 递归终止条件
    if(l>r) return true;
    int flag = post[r];
    int k;
    for(int i = l; i < r; i++) //这里是以形参l为左界向右扫描 不是o ! !
        if(post[i]>flag){ //因为最终要递归 vector的数组下标 依赖于l r 边界
            k = i;
            break; //这个break很关键，因为你要找第一个大于根结点的元素，右边的为右子树部分
        }
    for(int j = k+1; j<r;j++)
        if(post[j] <=flag) return false;
    return isBST(post,l,k-1) && isBST(post,k,r-1);
}

```

解题思路：

- 后序遍历定义： [左子树 | 右子树 | 根节点] ，即遍历顺序为“左、右、根”。
- 二叉搜索树定义：左子树中所有节点的值 < 根节点的值；右子树中所有节点的值 > 根节点的值；其左、右子树也分别为二叉搜索树。

```

3 * struct TreeNode {
4 *     int val;
5 *     TreeNode *left;
6 *     TreeNode *right;
7 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
8 * };
9 */
10 class Solution {
11 public:
12     bool isBalanced(TreeNode* root) {
13         if(!root) return true;
14         if(dfs(root) == -1)
15             return false;
16         else
17             return true;
18     }
19     int dfs(TreeNode *root){
20         if(root != nullptr){
21             int left = dfs(root ->left);
22             if(left == -1) return -1;
23
24             int right = dfs(root ->right);
25             if(right == -1) return -1;
26             return abs(left-right)>1 ? -1 : max(left,right)+1;
27         }
28         else
29             return 0;
30     }
31 };

```

```

1 class Solution {
2 public:
3     bool verifyPostorder(vector<int>& postorder) {
4         if (postorder.empty()) return true;
5         return isBST(postorder, 0, postorder.size()-1);
6     }
7     bool isBST(vector<int>& post, int l ,int r){
8         if(l >= r)
9             return true; //穿过叶子结点返回true
10        int flag = post[r];//flag存放当前树根节点的值
11        int i = l; //下面的循环要用到这个i,从i+1开始起查询是否有小于flag的数,须在for外定义
12        for(i = l; i < r; i++) //找第一个大于根结点的位置 该点到尾点之前为右子树
13            if(post[i] > flag)
14                break;
15            for(int j = i +1; j < r; j++)
16                if(post[j] <= flag)
17                    return false;
18            return isBST(post, l ,i-1) && isBST(post,i,r-1); //递归判断左子树及右子树
19        }
20    };

```

题目描述 评论(1.2k) 题解(1.5k) 提交记录

智能模式

```

1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11};
12
13 class Solution {
14 public:
15     vector<int> vec; //这个全局变量很关键 要将root中的val放入这个全局vector
16     void traverse(TreeNode* root){
17         if(!root) return;
18         traverse(root->left);
19         vec.push_back(root->val);
20         traverse(root->right);
21     }
22     bool isValidBST(TreeNode* root) {
23         if(!root) return true;
24         traverse(root); //在下面的函数中可以操作上面函数存放的vector进行比较判断
25         for(int i = 1; i < vec.size(); i++) //逐位比较 得到比较结果
26             if(vec[i] <= vec[i-1]) //二叉搜索树的中序遍历 严格递增！！！
27                 return false;
28     }
29 }
```

示例 1：
输入：
`2
/\br/>1 3`
输出：true

示例 2：
输入：
`5
/\br/>1 4
/\br/>3 6`
输出：false
解释：输入为：[5,1,4,null,null,3,6]。
根节点的值为 5，但是其右子节点值为 4。

二叉查找树，也称二叉搜索树，或二叉排序树。其定义也比较简单，要么是一颗空树，要么就是具有如下性质的二叉树：

若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

任意节点的左、右子树也分别为二叉查找树；

没有键值相等的节点



由于普通的二叉查找树会容易失去“平衡”

平衡二叉搜索树，又被称为AVL树，且具有以下性质：

它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树



二叉查找树还有一个性质，即对二叉查找树进行中序遍历，即可得到有序的数列 递增序列

牛客竞赛-上周考过 > 找到字符串中最长无重复字符串

时间限制：C/C++ 2秒，其他语言4秒 空间限制：C/C++ 256M，其他语言512M 热度指数：83443 本题知识点：哈希 双指针 字符串

算法知识视频讲解

题目描述

给定一个数组arr，返回arr的最长无重复字符串的长度(无重复指的是一组所有数字都不相同)。

示例1

输入
`[2,3,4,5]`

返回值
`4`

```

1 // 注意 arr[i] 是 o ? ? ? ? 不是 - ! ! ! 它是一个随机数！！！
2 int maxLength(vector<int>& arr) {
3     // write code here
4     set<int> se;
5     int l = 0, r = 0; // 这样是正确的！！！！！！
6     int maxn = 0;
7
8     while(r < arr.size()){
9         if(se.count(arr[r]) == 0){
10             se.insert(arr[r++]);
11             maxn = max(maxn, r-l);
12         }
13         else{
14             se.erase(arr[l++]);
15         }
16     }
17     return maxn;
18 }
```

牛客竞赛-上周考过 > 括号序列

题目描述

给出一个仅包含字符'.'','(',')'和'{' 的字符串，判断给出的字符串是否合法的括号序列。括号必须以正确的顺序关闭，'()' 和 '()' 都是合法的括号序列，但 '()' 和 '()' 不合法。

示例1

输入
`"["`

返回值
`false`

```

1 // 注意 sk.top() == ')' -> sk.pop();
2 // 用栈实现括号匹配，空栈不可以调用 sk.top() 会报越界错误。
3 bool isValid(string s) {
4     stack<char> sk;
5     for(int i = 0; i < s.size(); i++){
6         if(sk.empty()){
7             sk.push(s[i]);
8         }
9         else if(s[i] == ')'){
10             if(sk.top() == '(') sk.pop();
11             else if(s[i] == '}'){
12                 if(sk.top() == '{') sk.pop();
13                 else if(s[i] == ']'){
14                     if(sk.top() == '[') sk.pop();
15                     else if(sk.top() == ')') sk.pop();
16                 }
17             }
18             else if(s[i] == ')') sk.pop();
19             else if(s[i] == '}'){
20                 if(sk.top() == '{') sk.pop();
21                 else if(sk.top() == ']') sk.pop();
22             }
23             else if(s[i] == ']'){
24                 if(sk.top() == '[') sk.pop();
25             }
26             else sk.push(s[i]); // 如何可以是一个 s[i] == ')' 也可以是 s[i] == '}' -> sk.top() != '('
27         }
28     }
29     return sk.empty();
30 }
```

牛客题库-上周考过 > 费波那契数列

相似的企业面试

<上一题 | 下一题>

我的提交 登录 / 注册

时间限制: C/C++ 1s, 其他语言 2s 空间限制: C/C++ 64M, 其他语言 128M 热度指数: 1046905 本题知识点: 数组

算法知识视频讲解

题目描述

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0，第1项是1）。

n ≤ 39

示例1

输入

```
4
```

返回值

```
3
```

```
1 class Solution {
2 public:
3     int Fibonacci(int n) {
4         vector<int> res;
5         for(int i = 0; i <= n; i++){
6             if(i == 0)
7                 res.push_back(0);
8             else if(i == 1)
9                 res.push_back(1); // 不加这个else,下面的else就成立了,因为条件i==0也成立,出现未预期的结果
10            else
11                res.push_back(res[i-1] + res[i-2]);
12        }
13        return res[n];
14    }
15 }
```

牛客题库-上周考过 > 二分查找 II

相似的企业面试

<上一题 | 下一题>

我的提交

时间限制: C/C++ 1s, 其他语言 2s 空间限制: C/C++ 64M, 其他语言 128M 热度指数: 1046905 本题知识点: 二分查找

题目描述

请实现一个搜索数字的升序数组的二分查找。
给定一个元素有序的（升序）整型数组 nums 和一个目标值 target , 写一个函数搜索 nums 中的 target 。如果目标值存在返回下标, 否则返回 -1。

示例1

输入

```
[1,2,3,4,5], 4
```

返回值

```
2
```

说明

从左到右, 查找到第一个为4的, 下标为2, 返回2。

```
1 #include <iostream>
2
3 class Solution {
4 public:
5     int search(vector<int>& nums, int target) {
6         int i = 0, j = nums.size() - 1, mid = 0;
7         while(i <= j){ // 分情况, 该处必须要是<=, 假如是<, 最左边的那个元素也可以为target
8             mid = i + (j - i) / 2;
9             if(nums[mid] == target){
10                 while(mid < 0 && nums[mid] == nums[mid - 1])
11                     mid = mid - 1; // mid == 0 时 mid == mid - 1, 则mid--, return那个mid
12                 return mid; // mid == -1 左边已经没有元素 直接return这个mid
13             }
14             else if(nums[mid] > target)
15                 j = mid - 1;
16             else
17                 i = mid + 1;
18         }
19         return -1;
20     }
21 }
```

示例2

亲密字符串

hareyukai 发布于 2019-11-13 1.2k 字符串 C++

1. 统计字符串A，B中字符不匹配的下标。
2. 不匹配的下标个数不等于0或2时，不能由A得到B。
3. 不匹配的下标个数等于0时，A与B中字符完全相同，还需要A中有重复字符。
4. 不匹配的下标个数等于2时，判断交换两对字符后是否匹配。

859. 亲密字符串

难度: 简单 142 支持 反馈 报错

给定两个由小写字母构成的字符串 A 和 B , 只要我们可以通过交换 A 中的两个字母得到与 B 相等的结果，就返回 true ; 否则返回 false 。

交换字母的定义是取两个下标 i 和 j (下标从 0 开始) , 只要 $i \neq j$ 就交换 $A[i]$ 和 $A[j]$ 处的字符。例如，在 “abcd” 中交换下标 0 和下标 2 的元素可以生成 “cbad” 。

示例 1：

```
输入： A = "ab", B = "ba"
输出： true
解释： 你可以交换 A[0] = 'a' 和 A[1] = 'b' 生成 "ba"，此时
A 和 B 相等。
```

```
1 class Solution {
2 public:
3     bool buddyStrings(const string& a, const string& b) {
4         if (a.size() != b.size()) return false;
5         vector<int> index_of_mismatch;
6         for (int i = 0; i < a.size(); i++)
7             if (a[i] != b[i]){
8                 index_of_mismatch.push_back(i);
9             }
10        if (2 < index_of_mismatch.size() || index_of_mismatch.size() == 1 ) return false;
11
12        if (index_of_mismatch.size() == 0) {
13            return set<char>(a.begin(), a.end()).size() < a.size();
14        } else if (index_of_mismatch.size() == 2) {
15            return a[index_of_mismatch[0]] == b[index_of_mismatch[1]] &&
16                  a[index_of_mismatch[1]] == b[index_of_mismatch[0]];
17        }
18        return false;
19    }
20 }
```

1668. 最大重复子字符串

难度: 简单 10 支持 反馈 报错

给你一个字符串 sequence , 如果字符串 word 连续重复 k 次形成的字符串是 sequence 的一个子字符串, 那么单词 word 的 重复值为 k 。单词 word 的 最大重复值 是单词 word 在 sequence 中最大的重复值。如果 word 不是 sequence 的子串, 那么重复值 k 为 0 。

给你一个字符串 sequence 和 word , 请你返回 最大重复值 k 。

示例 1：

```
输入：sequence = "ababc", word = "ab"
输出：2
解释："abab" 是 "ababc" 的子字符串。
```

```
1 class Solution {
2 public:
3     int maxRepeating(string sequence, string word) {
4         int cnt = 0;
5         if(sequence.find(word) == string::npos)
6             return 0;
7         else{
8             string res ="";
9             while(res.size() <= sequence.size()){
10                 res += word;
11                 if(sequence.find(res) != string::npos)
12                     cnt++;
13                 else
14                     break;
15             }
16         }
17         return cnt;
18    }
19 }
```

题目描述 评论 (5.0k) 题解 (5.7k) 提交记录

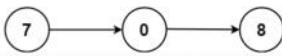
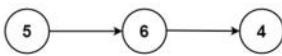
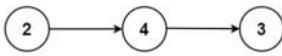
2. 两数相加

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：



输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[7,0,8]

题目描述 评论 (2.7k) 题解 (4.0k) 提交记录

9. 回文数

进阶 简单 1470 ☆ ★ ★

给你一个整数 x ，如果 x 是一个回文整数，返回 `true`；否则，返回 `false`。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。例如，`121` 是回文，而 `123` 不是。

示例 1：

输入： $x = 121$
输出：`true`

题目描述 评论 (5.0k) 题解 (5.7k) 提交记录

C++ * 简洁模式

```
14 ListNode * next;
15 ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
16 }
17 class Solution {
18 public:
19     ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
20         ListNode *head = nullptr, *tail = nullptr;
21         int cnt = 0;
22         while(l1 || l2){
23             int a, b;
24             if(l1){
25                 a = l1->val;
26                 l1 = l1->next;
27             }
28             else a=0;
29             if(l2){
30                 b = l2->val;
31                 l2 = l2->next;
32             }
33             else b=0;
34
35             //int sum = (a + b + cnt) % 10;
36             int sum = (a + b + cnt);
37             cnt = sum /10 ;
38             if(!head){
39                 head = tail = new ListNode(sum%10);
40             }
41             else{
42                 tail->next = new ListNode(sum%10);
43                 tail = tail->next;
44             }
45         }
46         if(cnt >0)
47             tail->next = new ListNode(cnt);
48
49     }
50 };
```

题目描述 评论 (2.7k) 题解 (4.0k) 提交记录

C++ * 简洁模式

```
1 class Solution {
2 public:
3     bool isPalindrome(int x) {
4         string str = to_string(x);
5         int i = 0, j = str.size()-1;
6         while(i<j){
7             if(str[i] != str[j])
8                 return false;
9             i++;
10            j--;
11        }
12        return true;
13    };
14 };
```

conclusion

冒泡排序

```
void bubble_sort(vector<int>& nums){  
    for(int i = 0; i < n; i++){  
        for(int j = 1; j < n - i; j++){  
            if(nums[j-1] > nums[j])  
                swap(nums[j-1], nums[j]);  
        }  
    }  
}
```

选择排序

```
vector<int> selectSort(vector<int>& nums) {  
    int n = nums.size();  
    for (int i = 0; i < n; ++i) {  
        int idx = i;  
        for (int j = i; j < n; ++j) {  
            if (nums[j] < nums[idx]) {  
                idx = j;  
            }  
        }  
        swap(nums[i], nums[idx]);  
    }  
    return nums;  
}
```

插入排序

```
vector<int> insertSort(vector<int>& nums) {  
    int n = nums.size();  
    for (int i = 0; i < n; ++i) {  
        for (int j = i; j > 0 && nums[j] < nums[j-1]; --j) {  
            swap(nums[j], nums[j-1]);  
        }  
    }  
    return nums;  
}
```

快速排序

```
void qSort(vector<int>& nums, int l, int r) {  
    if (l >= r) return;  
    int m = l;  
    for (int i = l; i < r; ++i) {  
        if (nums[i] < nums[r]) {  
            swap(nums[m++], nums[i]);  
        }  
    }  
    swap(nums[m], nums[r]);  
    qSort(nums, l, m-1);  
    qSort(nums, m+1, r);  
}
```

```
vector<int> quickSort(vector<int>& nums) {  
    int n = nums.size();  
    qSort(nums, 0, n-1);  
    return nums;  
}
```

归并排序

```
vector<int> mSort(vector<int>& nums, int l, int r) {  
    if (l >= r) return {nums[l]};  
    int m = l+(r-l)/2;  
    vector<int> lnums = mSort(nums, l, m);  
    vector<int> rnums = mSort(nums, m+1, r);  
    vector<int> res;  
    int i = 0, j = 0;  
    while (i <= m-l && j <= r-m-1) {  
        if (lnums[i] < rnums[j]) {  
            res.push_back(lnums[i++]);  
        } else {  
            res.push_back(rnums[j++]);  
        }  
    }  
    while (i <= m-l) {  
        res.push_back(lnums[i++]);  
    }  
    while (j <= r-m-1) {  
        res.push_back(rnums[j++]);  
    }  
}
```

```

return res;
}

希尔排序
vector<int> shellSort(vector<int>& nums) {
    int n = nums.size();
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; ++i) {
            for (int j = i; j-gap >= 0 && nums[j-gap] > nums[j]; j -= gap) {
                swap(nums[j-gap], nums[j]);
            }
        }
    }
    return nums;
}

```

计数排序

```

vector<int> countSort(vector<int>& nums) {
    int n = nums.size();
    if (!n) return {};
    int minv = *min_element(nums.begin(), nums.end());
    int maxv = *max_element(nums.begin(), nums.end());
    int m = maxv-minv+1;
    vector<int> count(m, 0);
    for (int i = 0; i < n; ++i) {
        count[nums[i]-minv]++;
    }
    vector<int> res;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < count[i]; ++j) {
            res.push_back(i+minv);
        }
    }
    return res;
}

```

桶排序

```

vector<int> bucketSort(vector<int>& nums) {
    int n = nums.size();
    int maxv = *max_element(nums.begin(), nums.end());
    int minv = *min_element(nums.begin(), nums.end());
    int bs = 1000;
    int m = (maxv-minv)/bs+1;
    vector<vector<int>> bucket(m);
    for (int i = 0; i < n; ++i) {
        bucket[(nums[i]-minv)/bs].push_back(nums[i]);
    }
    int idx = 0;
    for (int i = 0; i < m; ++i) {
        int sz = bucket[i].size();
        bucket[i] = quickSort(bucket[i]);
        for (int j = 0; j < sz; ++j) {
            nums[idx++] = bucket[i][j];
        }
    }
    return nums;
}

```

The screenshot shows an online judge interface with the following details:

- 题目描述**: 给定一个字符串，请设计一个高效算法，计算其中最长回文串的长度。
- 输入**: abc1234321ab, 12
- 返回值**: ?
- 代码** (C++):

```

1 class Solution {
2 public:
3     int getLongestPalindrome(string A, int n) {
4         // write code here
5         int maxn = 0;
6         int start = 0;
7         vector<vector<int>> dp(n, vector<int>(n, 0));
8         for(int i = A.size()-1; i >= 0; i--) {
9             for(int j = i; j < A.size(); j++) {
10                 if(A[i] == A[j]) {
11                     if(j-i <= 1)
12                         dp[i][j] = true;
13                     else
14                         dp[i][j] = dp[i+1][j-1];
15                 }
16                 if(dp[i][j] && (j-i+1 > maxn)) {
17                     maxn = j-i+1;
18                     start = i;
19                 }
20             }
21         }
22     }
23     return maxn;
24 }

```

53 开一个 $n \times n$ 的dp数组 记录 i从0-n-1 j从0-n-1 dp[i][j]表示i到j的回文串长度
 54 最大回文串 游标i 从最尾端开始往前扫 j从i开始往后扫
 55 如果发现 s[i] == s[j] 如果 $i-j < 1$ (j=i也为true) 则 dp[i][j] = true. 如果 $j-i > 1$ 则 dp[i][j] = dp[i+1][j-1] 看它是否为true
 56 第二个for循环内 游标每向前移动一次 如果 dp[i][j] 为true 且 $i+1 > maxn$ 更新maxn长度.

牛客网-上周考过 > 数组中出现次数超过一半的数字 > 相似的企业真题 < 上一题 | 下一题 > 我的提交 登录 / 注册

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 64M, 其他语言128M 热度指数: 682379
 本题知识点: 哈希 数组

算法知识视频讲解

题目描述
 数组中有一个数字出现的次数超过数组长度的一半, 请找出这个数字. 例如输入一个长度为9的数组 [1,2,3,2,2,5,4,2]. 由于数字2在数组中出现了5次, 超过数组长度的一半, 因此输出2. 如果不存在则输出0.

示例1

输入:
 [1,2,3,2,2,2,3,4,2]

返回值:
 2

执行结果:

```
4 可以先对这个 数组排序 一个数出现 次数超过了数组长度的一半 那排序完中间的那个元素也许是众数
5 设为cont 再遍历数组 如果元素==cont 则 sum++ 看sum是否大于 size() / 2
6
7 还有就是可以用map 将 nums[i]存起来 key是nums[i] value 是 出现次数直接++
8 由于只出现了一个 边加边判断 mp【nums[i】】的 value 如果大于 size() / 2 返回这个 nums[i];
```

牛客网-上周考过 > 锁表中环的入口节点 > 相似的企业真题 < 上一题 | 下一题 > 我的提交 登录 / 注册

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 32M, 其他语言64M 热度指数: 111249
 本题知识点: 链表 双指针

算法知识视频讲解

题目描述
 对于一个给定的链表, 返回环的入口节点, 如果没有环, 返回null.
 提示:
 你能给出不利用额外空间的解法么?

说明: 本题目包含复杂数据结构ListNode, 点此查看相关信息

关联企业
 科大讯飞 美团 字节跳动

关联职位
 研发 测试 算法

C++(clang++11) 核心代码模式

```
1 /**
2 * Definition for singly-linked-list.
3 * struct ListNode {
4 *     int val;
5 *     ListNode *next;
6 * };
7 */
8
9 class Solution {
10 public:
11     ListNode *detectCycle(ListNode *head) {
12         if(head == nullptr)
13             return nullptr;
14         ListNode *fast = head, *slow = head;
15         //while(fast && fast->next && fast->next->next) //你这个一开始fast就等于slow //都只有相遇了, 才将slow 移动到head 再一步一步走
16         while(fast && fast->next){ //while根本不执行
17             fast = fast->next->next;
18             slow = slow->next;
19             if(fast == slow){ //你这里不用() f, s走一步, 就会陷入下面的while循环
20                 slow = head; //只有相遇了, 才将slow 移动到head 再一步一步走
21                 while(slow != fast){ //你都不会写while循环
22                     slow = slow->next;
23                     fast = fast->next;
24                 }
25             }
26         }
27         return fast;
28     }
29 }
```

先定义 fast slow指针 , fast每次走2步, slow每次走1步 `while(fast && fast->next)` 循环, 如果 fast或者 fast->next为null 说明没有环
 fast slow相遇 再将 slow移动到 head , fast从相遇点开始, 两个指针每都走1步, fast slow再次相遇的那个位置 结点指针就是环的入口处 环的入口节点

牛客网-上周考过 > 矩阵的小秘密 > 相似的企业真题 < 上一题 | 下一题 > 我的提交 登录 / 注册

时间限制: C/C++ 1秒, 其他语言2秒
 空间限制: C/C++ 200M, 其他语言512M 热度指数: 18083
 本题知识点: 数组 动态规划

算法知识视频讲解

题目描述
 给出一个 $n \times n$ 的矩阵 A , 从左上角开始顺次只往右或者向下走, 最后到达右下角的位置, 路径上所有的数字累加起来就是路径和, 通过所有的路径中最小的路径和。

示例1

输入:
 [[1,3,2,0],[1,1,2,4],[1,0,0,1],[0,0,0,0]]

返回值:
 12

备注:
 $1 < n, m < 9000$

C++(clang++11) 核心代码模式

```
3 class Solution {
4 public:
5     ...
6     int minPathSum(vector<vector<int>> &matrix) {
7         // 先将 code 复制
8         int n = matrix.size();
9         int m = matrix[0].size();
10        vector<vector<int>> dp(m, vector<int>(n, 0));
11        dp[0][0] = matrix[0][0];
12        for(int i = 0; i < n; ++i) {
13            for(int j = 0; j < m; ++j) {
14                if(i == 0 && j == 0) {
15                    dp[0][0] = matrix[0][0] + mp(1-1)(0);
16                    mp(1)(0) = matrix[0][0] + mp(1-1)(0); // 1-1指向matrix[0][0]的值
17                } else if(j == m - 1) {
18                    mp(0)(j) = matrix[0][j] + mp(1-1)(j);
19                } else if(i == n - 1) {
20                    mp(i)(0) = matrix[i][0] + mp(1-1)(0);
21                } else if(i < n - 1 && j < m - 1) {
22                    mp(i)(j) = min(mp(i-1)(j), mp(i)(j-1)) + matrix[i][j];
23                }
24            }
25        }
26    }
}
```

```

.../
int minPathSum(vector<vector<int>>& matrix) {
    // write code here
    int n = matrix.size(); //这是直接在原始数组上操作赋值的方法
    int m = matrix[0].size();

    for(int i = 1; i < n; i++) {
        matrix[i][0] += matrix[i-1][0];
        for(int j = 1; j < m; j++) {
            matrix[0][j] += matrix[0][j-1];
            for(int i = 1; i < n; i++) {
                for(int j = 1; j < m; j++) {
                    matrix[i][j] = min(matrix[i-1][j], matrix[i][j-1]) + matrix[i][j];
                }
            }
        }
    }
    return matrix[n-1][m-1];
}

```

先计算矩阵的行列，首先需要遍历矩阵 第0行与第0列 算出 第0行与第0列的累加结果， $m[i][0] += m[i-1][0]$
 再从第1行与第1列开始 遍历矩阵每一个元素，当前的累加值 为 $\min(m[i-1][j], m[i][j-1]) + m[i][j]$ ；
 从左边过来的值与从右边过来的值 累加到当前元素 即为当前元素的累加和
 一直遍历到矩阵结束 返回最后一个元素的累加和 为最小路径

题目描述

判断给定的链表中是否有环。如果有环则返回true，否则返回false。
 你能给出空间复杂度 $O(1)$ 的解法么？

说明：本题目包含复杂数据结构 ListNode，[点击查看相关信息](#)

~ 关联企业

小米 腾讯 哈啰出行 美团 TP-LINK

~ 共享职位

```

7 // ...
8 */
9 class Solution {
10 public:
11     bool hasCycle(ListNode *head) {
12         ListNode *fast = head; //这个 fast 就是未定义的指针，用它就会引发未知越界错误
13         ListNode *slow = head;
14         if(!head) return false;
15         while(fast && fast->next) {
16             fast = fast->next->next;
17             slow = slow->next;
18             if(fast == slow)
19                 return true;
20         }
21     return false;
22 }

```

两个变量一直定义的时候，初始化一定要记得是单独初始化，否则就会是一个变量没有得到初始化，出现越界 或者它就是一个随机值 用它就错了

牛客题霸-上周考过 > 用两个栈实现队列 ▾ 相似的企业真题 < 上一题 | 下一题 >

时间限制：C/C++ 1秒，其他语言2秒
 空间限制：C/C++ 64M，其他语言128M 热度指数：827411
 本题知识点：栈

算法知识视频讲解

题目描述

用两个栈来实现一个队列，完成队列的Push和Pop操作。队列中的元素为int类型。

~ 关联企业

字节跳动 米哈游 深信服 vivo

~ 关联职位

研发 测试 前端

```

1 class Solution
2 {
3     stack<int> sk1;
4     stack<int> sk2;
5 public:
6     void push(int node) {
7         sk1.push(node);
8     }
9     int pop() {
10     if(sk2.empty()) {
11         while(!sk1.empty()) {
12             sk2.push(sk1.top());
13             sk1.pop();
14         }
15     }
16     if(sk2.empty()) return 0;
17     else {
18         int del = sk2.top();
19         sk2.pop();
20     }
21 }
22 }

```

入队是sk1直接push
 如果sk2为空 将sk1中所有元素push到sk2
 如果 sk2为空 返回0
 else sk2有元素 删除sk2栈顶元素

牛客题霸-上周考过 > 合并两个有序的数组 □ 相似的企业真题 < 上一题 | 下一题 > 我的提交

时间限制 : C/C++ 1秒 , 其他语言2秒
空间限制 : C/C++ 32M , 其他语言64M 热度指数 : 107543
本题知识点 : 数组 双指针

算法知识视频讲解

题目描述

给出两个有序的整数数组 A 和 B , 请将数组 B 合并到数组 A 中, 变成一个有序的数组
注意:
可以假设 A 数组有足够的空间存放 B 数组的元素 , A 和 B 中初始的元素数目分别为 m 和 n

关联企业

跟谁学 字节跳动 网易雷火

关联职位

```
① C++(clang++11) □ 核心代码模式
1 class Solution {
2 public:
3     void merge(int A[], int m, int B[], int n) {
4         int a = m-1;
5         int b = n-1;
6         for(int i = m+n-1; i>=0; i--) { //需要填m+n次
7             if(b<0 || (a>0 && A[a] >= B[b])) {
8                 A[i] = A[a]; //B数组的元素用完了就按序填A数组,A数组元素没用完且A元素大 填A的元素
9                 a--;
10            } else{
11                A[i] = B[b];
12                b--;
13            }
14        }
15    }
16 }
17 总共需要填m+n次
18 如果B数组中的元素用完了 或者A中还有元素且A中的元素大 填A中的元素 更新下标:if(b>=0 && A[a] > B[b]) {
19 否则 填B中元素 更新下标
20 }
```

时间限制 : C/C++ 1秒 , 其他语言2秒
空间限制 : C/C++ 256M , 其他语言512M 热度指数 : 49418
本题知识点 : 链表

算法知识视频讲解

题目描述

输入一个链表，输出该链表中倒数第k个结点。
如果该链表长度小于k，请返回空。

示例1

输入	<input type="text" value="1,2,3,4,5,1"/>	复制
返回值	<input type="text" value="5"/>	复制

```
① C++(clang++11) □ 核心代码模式
13 .....
14 ....* @param pHead·ListNode类·
15 ....* @param k·int类型·
16 ....* @return·ListNode类·
17 ..../
18 ListNode* FindKthToTail(ListNode* pHead, int k) {
19 ....// write code here
20 ....
21 .... ListNode *fast = pHead;
22 .... ListNode *slow = pHead;
23 .... while(k--){
24 ....     if(fast)
25 ....         fast = fast->next;
26 ....     else
27 ....         return nullptr;
28 .... }
29 .... while(fast){
30 ....     fast = fast->next;
31 ....     slow = slow->next;
32 .... }
33 .... return slow;
34 ....
35 };
36 注意长度小于k 在while(k--) 判断 if (fast) 如果fast有值 则长度>=k·
37 快指针先走k步,再fast slow一起走 fast到达末端 == null 此时的slow就是倒数第k个结点
```

题目描述 **评论 (459)** **点赞 (1.3k)** **提交记录**

剑指 Offer 24. 反转链表
难度 **简单** 822 ⚡ ⚡ ⚡ ⚡ ⚡

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例:
输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL

限制:
0 <= 节点个数 <= 5000

注意 : 本题与主站 206 题相同 : <https://leetcode-cn.com/problems/reverse-linked-list/>

```
① C++ □ 智能模式
1 /**
2  * Definition for singly-linked list.
3  * struct ListNode {
4  *     int val;
5  *     ListNode *next;
6  *     ListNode(int x) : val(x), next(NULL) {}
7  * };
8 */
9 class Solution {
10 public:
11     ListNode* reverseList(ListNode* head) {
12         if(!head) return NULL;
13         ListNode *cur = head;
14         ListNode *pre = NULL;
15         ListNode *tmp = NULL;
16         while(cur){
17             tmp=cur->next;
18             cur->next = pre;
19             pre = cur;
20             cur = tmp;
21         }
22         return pre;
23     }
24 };
25 反转链表采用三指针法 pre tmp = NULL , cur=head
```

92. 反转链表 II
难度 **中等** 859 ⚡ ⚡ ⚡ ⚡ ⚡

给你单链表的头指针 head 和两个整数 left 和 right , 其中 left < right 。请你反转从位置 left 到位置 right 的链表节点，返回反转后的链表。

示例 1:

输入 : head = [1,2,3,4,5], left = 2, right = 4
输出 : [1,4,3,2,5]

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int l, int r) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;

        // 注意这里是 l 不是 1
        r -= 1;
        // hh 就是 "哈哈" 的意思 ...
        // 别忘了 hh 是 head 的意思,为了防止与 height 的简写 h 冲突
        ListNode* hh = dummyHead;
        while (l-- > 1)
            hh = hh->next;

        ListNode* prv = hh->next;
        ListNode* cur = prv->next;
        while (r-- > 0) {
            ListNode* nxt = cur->next;
            cur->next = prv;
            prv = cur;
            cur = nxt;
        }
        hh->next->next = cur;
        hh->next = prv;
        return dummyHead->next;
    }
}
```

当l=2, r=4 , 三个指针 hh , prv , cur 分别指向前面的1 , l=2 , 与 l=3 ;

内部while 翻转r-1 次，注意后续将hh->next ->next 与 hh->next 的指向调整到对应的位
置处

25. K 个一组翻转链表

难度 困难 收藏 1035 分享

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

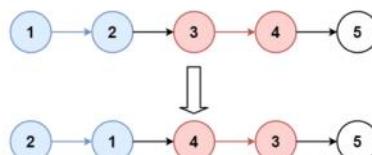
k 是一个正整数，它的值大小或者等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

进阶：

- 你可以设计一个只使用常数额外空间的算法来解决此问题吗？
- 你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

示例 1：



```
class Solution {
public:
    ListNode* dfs(ListNode* h, int k, int m, int b) {
        if (b == m) return h;//递归中止条件,m是可反转的次数,b是实际反转次数
        ListNode*ph = h,*pr = NULL,*t = NULL;//ph记录头指针,即首位
        int cnt = 0;
        /*反转链表基本操作*/
        while (cnt++ != k){
            t = h->next;
            h->next = pr;
            pr = h;
            h = t;
        }
        ph->next = dfs(h,k,m,b+1);//pr是ph的前一个,即一组中的最后一个数,反转后就是第一个数
        return pr;
    }
    ListNode* reverseKGroup(ListNode* head, int k) {
        int sum = 0;
        ListNode *h = head;
        while (head){
            sum++;
            head = head->next;
        }
        return dfs(h,k,sum/k,0);
    }
};
```

k个一组翻转链表 用递归的方法 注意递归的终止条件

递归内容 ph->next = dfs(h,k,m,b+1) return pr , 返回的是最后一个结点 ，传入的h是

下一个段的h

递归函数中while (cnt++ != k) 局部反转链表内容

牛客竞赛-上周考过 > 合并有序链表

相似的企业真题 <上一题 | 下一题>

时间限制 : C/C++ 1秒，其他语言2秒 空间限制 : C/C++ 64M，其他语言128M 热度指数 : 99493

本题知识点： 链表

■ 算法知识视频讲解

题目描述

将两个有序的链表合并为一个新链表，要求新的链表是通过拼接两个链表的节点来生成的，且合并后新链表依然有序。

示例1

输入

(1), (2)

返回值

(1, 2)

示例2

① C++(clang++11) ● 换行代码模式

```
1 class Solution {
2 public:
3     ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
4         if (l1 == NULL) return l2;
5         if (l2 == NULL) return l1;
6         ListNode *p1 = l1;
7         ListNode *p2 = l2;
8         ListNode *pHead = new ListNode();
9         ListNode *pNode = pHead;
10        while (p1 != NULL && p2 != NULL) {
11            if (p1->val <= p2->val) {
12                pNode->next = p1;
13                p1 = p1->next;
14            } else {
15                pNode->next = p2;
16                p2 = p2->next;
17            }
18            pNode = pNode->next;
19        }
20        if (p1 != NULL) pNode->next = p1;
21        if (p2 != NULL) pNode->next = p2;
22        return pHead->next;
23    }
24}
25};
```

首先定义一个pHead结点作为虚拟头结点，再定义一个游标 *pNode , 用来滑动指向下一个结点，同时在指向完，这个游标pNode自己也要进行更新，定义p1, p2分别指向 l1, l2 比较p1->val 与 p2 ->val 如果小于 则游标的下一个结点 指向p1 更新p1指针
如果大于 游标的下一个结点指向 p2 再更新p2指针 指向的结点 p2 = p2 ->next;
用while(p1 && p2) 循环退出，如果 p1 为空，则pNode游标指向p2结点即可
特殊情况 l1 为空 合并结果 直接一开始时返回 p2 即可

牛客竞赛-上周考过 > 求二叉树的层序遍历

相似的企业真题 <上一题 | 下一题>

时间限制 : C/C++ 1秒，其他语言2秒 空间限制 : C/C++ 64M，其他语言128M 热度指数 : 73612

本题知识点： 树 bfs

■ 算法知识视频讲解

题目描述

给定一个二叉树，返回该二叉树层序遍历的结果。（从左到右，一层一层地遍历）
例如：
给定的二叉树是[3,9,20,#,#,15,7],



该二叉树层序遍历的结果是
[
[3],
[9,20],
[15,7]
]

示例1

输入

① C++(clang++11) ● 换行代码模式

```
15 // vector<vector<int>> levelOrder(TreeNode* root) {
16 //     // write code here
17 //     if (!root) return {};
18 //     queue<TreeNode*> q;
19 //     q.push(root);
20 //     vector<vector<int>> res;
21 //     while (!q.empty()) {
22 //         int l = q.size();
23 //         vector<int> tmp;
24 //         for (int i = 0; i < l; i++) {
25 //             TreeNode* p = q.front();
26 //             q.pop();
27 //             tmp.push_back(p->val);
28 //             if (p->left) q.push(p->left);
29 //             if (p->right) q.push(p->right);
30 //         }
31 //         res.push_back(tmp);
32 //     }
33 //     return res;
34 // }
35 // }
36 // }
37
38 模拟点先入队
39 用queue实现队列，while循环的条件是队列不为空，通过队列的size来控制每一层打印的元素
40 用for循环打印，每遇到一个元素，在for内判断是否有子树，如果有将其push进队列
41 用for循环里面会有元素入队，这个队列的size是在for内会变的，因此for循环的条件不应该写q.size()
42 必须在for开始前用一个int 变量来保存q.size() 作为for循环打印条件！！！
```

时间限制 : C/C++ 1秒 , 其他语言2秒 空间限制 : C/C++ 64M , 其他语言128M 热度指数 : 60992

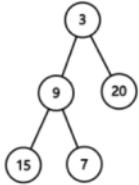
本题知识点 : 栈 树 bfs

算法知识视频讲解**题目描述**

给定一个二叉树，返回该二叉树的之字形层序遍历，(第一层从左向右，下一层从右向左，一直这样交替)

例如：

给定的二叉树是{3,9,20,#,#,15,7}，



该二叉树之字形层序遍历的结果是

```
[
[3],
[20,9],
[15,7]
]
```

```
① C++(clang++11) ✓ 核心代码模式
15     ...
16     > vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
17     >>> // write code here
18     >     vector<vector<int>> res;
19     >     queue<TreeNode*> q;
20     >     if(!root)
21     >         return res;
22     >     q.push(root);
23     >     while(!q.empty()){
24     >         int l = q.size();
25     >         vector<int> tmp;
26     >         for(int i = 0; i < l; i++){
27     >             TreeNode *p = q.front();
28     >             q.pop();
29     >             tmp.push_back(p->val);
30     >             if(p->left) q.push(p->left);
31     >             if(p->right) q.push(p->right);
32     >         }
33     >         int k = res.size();
34     >         if(k % 2 == 0)
35     >             res.push_back(tmp);
36     >         else{
37     >             reverse(tmp.begin(),tmp.end());
38     >             res.push_back(tmp);
39     >         }
40     >     }
41     >     return res;
42 }
```

队列的元素类型是 `TreeNode*`
在`q.push(root)`前一定要判断 `if(!root)`。

否则如果`root`为空 `push`操作就会引发段错误z型遍历就是对`vec`中的元素数量 `size`加了一个奇偶判断，如果为偶数则正常直接`push_back`如果为奇数则先对`tmp`中的元素作`reverse`再`push`到 `res`

时间限制 : C/C++ 1秒 , 其他语言2秒 空间限制 : C/C++ 64M , 其他语言128M 热度指数 : 72634

本题知识点 : 数组 双指针

算法知识视频讲解**题目描述**给出一个有n个元素的数组S，S中是否有元素a,b,c满足 $a+b+c=0$? 找出数组S中所有满足条件的三元组。

注意：

1. 三元组 (a, b, c) 中的元素必须按非降序排列。 (即 $a \leq b \leq c$)
2. 解集中不能包含重复的三元组。

例如，给定的数组 `s = [-10 0 10 20 -10 -40]`,解集为 $(-10, 0, 10) (-10, -10, 20)$ **示例1**

输入	<input type="text" value="[-2,0,1,1,2]"/>	<input type="button" value="复制"/>
返回值	<input type="text" value="[[[-2,0,2],[-2,1,1]]]"/>	<input type="button" value="复制"/>

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        sort(nums.begin(), nums.end());
        vector<vector<int>> ans;
        // 枚举 a
        for (int first = 0; first < n; ++first) {
            // 需要和上一次枚举的数不相同
            if (first > 0 && nums[first] == nums[first - 1]) {
                continue;
            }
            // c 对应的指针初始指向数组的最右端
            int third = n - 1;
            int target = -nums[first];
            // 枚举 b
            for (int second = first + 1; second < n; ++second) {
                // 需要和上一次枚举的数不相同
                if (second > first + 1 && nums[second] == nums[second - 1]) {
                    continue;
                }
                // 需要保证 b 的指针在 c 的指针的左侧
                while (second < third && nums[second] + nums[third] > target) {
                    --third;
                }
                // 如果指针重合，随着 b 后续的增加
                // 就不会有满足 a+b+c=0 并且 b < c 的 c 了，可以退出循环
                if (second == third) {
                    break;
                }
                if (nums[second] + nums[third] == target) {
                    ans.push_back({nums[first], nums[second], nums[third]});
                }
            }
        }
        return ans;
    }
}
```

三数之和，对排序完的数组采用三指针枚举法，(`first,second,third`) 每一次循环 代表每一位都需要考虑和上一次枚举的数不相同

牛客题霸-上周考过 > 寻找第K大 ▾

时间限制 : C/C++ 3秒, 其他语言6秒 空间限制 : C/C++ 64M, 其他语言128M 热度指数 : 139476

本题知识点 : 排 分治

算法知识视频讲解

题目描述

有一个整数数组, 请你根据快速排序的思路, 找出数组中第K大的数。
给定一个整数数组a, 同时给定它的大小n和要找的K(K在1到n之间), 请返回第K大的数, 保证答案存在。

示例1

输入

```
[1, 3, 5, 2, 2], 5, 3
```

返回值

```
2
```

① C++(clang++11) 核心代码模式

```
1 class Solution {
2 public:
3     int findKth(vector<int> a, int n, int K) {
4         // write code here
5         quick_sort(a, 0, n-1);
6         return a[n-K];
7     }
8     void quick_sort(vector<int> arr, int l, int r){
9         if(l >= r) return; // 递归函数的边界条件, 递归停止条件
10        int i = l, j = r; // 左右边界重合, 数组的长度为1
11        while(i < j){
12            while(i < j && arr[i] >= arr[l]) --j;
13            while(i < j && arr[i] <= arr[l]) ++i;
14            swap(arr[i],arr[j]);
15        }
16        swap(arr[l],arr[i]);
17        quick_sort(arr, l, i-1);
18        quick_sort(arr,i+1,r);
19    }
20    // 快排注意 递归底点条件: if(l >= r) return;
21    // 第k大的数, 从右边数起: n - K
22}
```

牛客题霸-上周考过 > 设计LRU缓存结构 ▾

时间限制 : C/C++ 2秒, 其他语言4秒 空间限制 : C/C++ 256M, 其他语言512M 热度指数 : 76163

本题知识点 : 模拟

算法知识视频讲解

题目描述

设计LRU缓存结构, 该结构在构造时确定大小, 假设大小为K, 并有如下两个功能

- set(key, value) : 将记录(key, value)插入该结构
- get(key) : 返回key对应的value值

示例1

输入

```
[[1,1,1],[1,2,2],[1,3,2],[2,1],[1,4,4],[2,2]], 3
```

返回值

```
[1,-1]
```

说明

第一次操作后: 最常使用的记录为 ("1", 1)
第二次操作后: 最常使用的记录为 ("2", 2), ("1", 1) 变为最不常用的
第三次操作后: 最常使用的记录为 ("3", 2), ("1", 1) 还是最不常用的
第四次操作后: 最常使用的记录为 ("1", 1), ("2", 2) 变为最不常用的
第五次操作后: 大小超过了3, 所以移除此时最不常用的记录 ("2", 2), 加入记录 ("4", 4), 并且为最常使用的记录, 然后 ("3", 2) 变为最不常用的记录

① C++(clang++11) 核心代码模式

```
1 class Solution {
2 public:
3     /* */
4     /* lru-design
5      * @param operators: int整型vector<vector<int>> the ops
6      * @param k: int整型 the k
7      * @return: int整型vector<int>
8      */
9     vector<int> LRU(vector<vector<int>> &operators, int k) {
10        // write code here
11        vector<int> ret;
12        for(auto x : operators){
13            if(x[0] == 1)
14                set(x[1],x[2],k);
15            else if(x[0] == 2)
16                get(x[1],ret); // 传一个vector ret给 get
17        }
18        return ret;
19    }
20
21 private:
22     unordered_map<int,int> map;
23     list<int> keys;
24     void set(int key, int value, int k){
25         if(keys.size() == k){
26             int del = keys.back();
27             keys.pop_back();
28             map.erase(del);
29         }
30         map[key] = value;
31         keys.push_front(key);
32     }
33     void get(int key, vector<int>& ret){
34         auto found = map.find(key);
35         if(found == map.end())
36             ret.push_back(-1);
37         else{
38             ret.push_back(found->second);
39             keys.remove(found->first);
40             keys.push_front(found->first);
41         }
42     }
43 }
```

定义一个list<int>
set函数中用一个map来保存 key-value对; 如果 keys.size() == k
int del = keys.back(), keys.remove(del), 也可以用keys.pop_back();
map.erase(del); map中删除这个元素

get函数中如果找到了 key, 将mp[key]压入ret;
同时将keys中的key删除, 再将key插入到list头部 用keys.push_front(key);

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 64M, 其他语言128M 热度指数: 80298

本题知识点: 链表 双指针

■ 算法知识视频讲解

题目描述

给定一个链表，删除链表的倒数第 n 个节点并返回链表的头指针。

例如，

给出的链表为: 1 → 2 → 3 → 4 → 5, $n = 2$.

删除了链表的倒数第 n 个节点之后,链表变为1 → 2 → 3 → 5.

备注:

题保证 n 一定是有请给出清晰的复杂度为 $O(n)$ 的算法。

示例1

输入
(1, 2), 2

复制

返回值
(2)

复制

① C++(clang++11) ▾ 核心代码模式

```

11
12     * @param head: ListNode类
13     * @param n: int整型
14     * @return: ListNode类
15     */
16    ListNode* removeNthFromEnd(ListNode* head, int n) {
17        // write code here
18        // ListNode *fast;
19        // ListNode *slow;
20        ListNode *fast = head;
21        ListNode *slow = head;
22        for(int i = 0; i < n; i++) {
23            fast = fast->next; //如果fast==null, 删除的为head结点
24        }
25        if(!fast)
26            return head->next;
27        while(fast->next){ //这里最终slow不是倒数第n个节点, 倒数第n+1个节点,
28            fast = fast->next; //fast不走到null, fast->next 走到null
29            slow = slow->next;
30        }
31        slow->next = slow->next->next;
32        return head;
33    }
34    快慢指针,先找到链表中的倒数第k个结点
35    注意如果fast==null,删除的为头结点
36    第二个while需要定位到 slow前面目标结点的前面一个,以便进行删除
37    条件为while(fast->next),删除就是 slow->next = slow->next->next

```

时间限制: C/C++ 1秒, 其他语言2秒 空间限制: C/C++ 256M, 其他语言512M 热度指数: 42401

本题知识点: 字符串 模拟

■ 算法知识视频讲解

题目描述

以字符串的形式读入两个数字,编写一个函数计算它们的和,以字符串形式返回。

(字符串长度不大于100000,保证字符串仅由'0'~'9'这10种字符组成)

示例1

输入
"1", "99"

复制

返回值
"100"

复制

说明
1+99=100

① C++(clang++11) ▾ 核心代码模式

```

12     // Write code here
13     string ans;
14     int count = 0;
15     int i = s.size() - 1, j = t.size() - 1;
16     while(i >= 0 || j >= 0 || count > 0) {
17         int a, b;
18         if(i >= 0)
19             a = s[i] - '0';
20         else
21             a = 0;
22         if(j >= 0)
23             b = t[j] - '0'; // 这里是第二个字符串s的字符
24         else
25             b = 0;
26         int tmp = a + b + count;
27         if(tmp >= 10) // 这里是-->就要进位
28             count = 1;
29         else
30             count = 0; // 这里必须要加else, count==0, 重复进位
31         ans.insert(ans.begin(), tmp + '0');
32         i--;
33         j--;
34     }
35     return ans;
36 }
37 }
38 };
39 模拟竖式加法

```

问题描述 :

给定一个整数数组和一个整数 k , 你需要找到该数组中和为 k 的连续的子数组的个数。

```

1 #include<iostream>
2 #include<vector>
3 using namespace std;
4
5 class Solution      //子数组个数问题
6 {
7 public:
8     int subarraySum(vector<int>& nums, int k)
9     {
10         int res = 0, n = nums.size();
11         for (int i = 0; i < n; ++i)
12         {
13             int sum = nums[i];
14             if (sum == k) ++res;
15             for (int j = i + 1; j < n; ++j)
16             {
17                 sum += nums[j];
18                 if (sum == k) ++res;
19             }
20         }
21         return res;
22     }
23 };

```

这种解法比较巧妙。用map建立hash表, 记录和为sum出现的次数。设 $b = \text{sum}[i] - k$, 那么 b 如果不是连续子数组的和则出现次数为0, 否则为实际出现次数, 然后累加即可。上代码

```

1 class Solution {
2 public:
3     int subarraySum(vector<int>& nums, int k) {
4         int n=nums.size();
5         int sum=0,res=0;
6         unordered_map<int,int> m;
7         m[0]=1;
8         for(int i=0;i<n;i++){
9             sum+=nums[i];
10            res+=m[sum-k];
11            m[sum]++;
12        }
13        return res;
14    }
15 };

```

```

/* 
遍历字符串 s, 若当前字符不是 ' ) ', 则直接进栈; 若当前字符是 ' ) ', 说明遇到了一对括号
则定义一个队列存储当前栈中 pop 出的字符(反转), 直到遇到 ' ( ' 为止, 之后再将反转后的字符串加入
栈中
*/

```

```
int main() {
    string str; //输入字符串
    cin >> str;
    string stk; //栈
    queue<char> que; //队列
    for (auto x : str) {
        if (x == ')') { //遇到了一对括号
            char temp;
            do { //反转这对括号中的字符 借助栈来实现反转
                temp = stk.back();
                stk.pop_back();
                if (temp != '(')
                    que.push(temp);
            } while (temp != '(');

            //将反转后的字符再次加入栈中
            while (!que.empty()) {
                temp = que.front();
                que.pop();
                stk.push_back(temp);
            }
        } else {
            stk.push_back(x);
        }
    }
    cout << stk << endl; //输出反转后的字符串
    return 0;
}
```

思维

2020年11月10日 18:09

二、数据结构的基本操作

对于任何数据结构，其基本操作无非遍历+访问。再具体一点就是：增删查改。

数据结构种类很多，但它们存在的目的都是在不同的应用场景，尽可能高效地增删查改。话说这不是数据结构的使命么？

如何遍历+访问？我们仍然从最高层来看，各种数据结构的遍历+访问无非两种形式：线性的和非线性的。

线性就是for/while 迭代为代表，非线性就是递归为代表。再具体一步，无非以下几种框架：

数组遍历框架，典型的线性迭代结构：

```
void traverse(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        // 迭代访问 arr[i]
    }
}
```

链表遍历框架，兼具迭代和递归结构：

```
/* 基本的单链表节点 */
class ListNode {
    int val;
    ListNode next;
}

void traverse(ListNode head) {
    for (ListNode p = head; p != null; p = p.next) {
        // 迭代访问 p.val
    }
}

void traverse(ListNode head) {
    // 递归访问 head.val
    traverse(head.next)
}
```

二叉树遍历框架，典型的非线性递归遍历结构：

```
/* 基本的二叉树节点 */
class TreeNode {
    int val;
    TreeNode left, right;
}

void traverse(TreeNode root) {
    traverse(root.left);
    traverse(root.right);
}
```

你看二叉树的递归遍历方式和链表的递归遍历方式，相似不？再看看二叉树结构和单链表结构，相似不？如果再多几条叉，N叉树你会不会遍历？

二叉树框架可以扩展为N叉树的遍历框架：

```
/* 基本的 N 叉树节点 */
class TreeNode {
    int val;
    TreeNode[] children;
}

void traverse(TreeNode root) {
    for (TreeNode child : root.children)
        traverse(child);
}
```

N叉树的遍历又可以扩展为图的遍历，因为图就是好几N叉棵树的结合体。你没说这是可能出现环的？这个很好办，用个布尔数组 visited 做标记就行了，这里就不写代码了。

所谓框架，就是套路。不管增删查改，这些代码都是永远无法脱离的结构，你可以把这个结构作为大纲，根据具体问题在框架上添加代码就行了

链表、树、图都是用指针表示的数据结构

1. 贪心算法

贪心算法是采用**贪心的策略**，保证每次操作都是**局部最优的**，从而得到**最终的结果是全局最优的**

如选取的贪心策略为每个人最多能吃自己最多数量的苹果

注意：

对数组或字符串排序是常见的操作，方便接下来的大小比较

若是对连续空间变量的操作，其实不需要明确区分数组和字符串，

它们本质上都是在连续空间上的有序变量集合，一个字符串“abc”也可以看作是一个数组['a','b','c']当然这是指的数组就是vector 字符串string

题目描述 评论 (692) ▲ 跟贴 (941) ④ 提交记录

⑤ 模板

455. 分发饼干

难度 简单 318 收藏 144

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 i ，都有一个胃口值 $g[i]$ ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 $s[j]$ ，如果 $s[j] \geq g[i]$ ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足最多数量的孩子，并输出这个最大数值。

示例 1：

```
输入: g = [1,2,3], s = [1,1]
输出: 1
```

贪心策略是：让饥饿度小的孩子吃尺寸小的饼干，以让饥饿度大的孩子有饼干吃

先对两个数组进行排序，再依次比较大小，计算有多少个对孩子满足条件。

```
1 class Solution {
2 public:
3     int findContentChildren(vector<int>& g, vector<int>& s) {
4         sort(g.begin(), g.end());
5         sort(s.begin(), s.end());
6         int i = 0, j = 0;
7         //int child;
8         while(i < g.size() && j < s.size()){
9             if(g[i] <= s[j]) i++;
10            j++;
11        }
12        return i;
13    }
14};
```

LC-分发糖果

135. 分发糖果

难度 **困难** 537 收藏 49 通过 34 提交 100
老师想给孩子们分发糖果，有 N 个孩子站成了一条直线。老师会根据每个孩子的表现，给他们糖果。为了避免他们争分。
你需要按照以下要求，帮助老师给这些孩子分发糖果：
 * 每个孩子至少分配到一个糖果。
 * 分得糖果的孩子必须比他两侧的孩子获得更多的糖果。
 那么这样下来，老师至少需要准备多少颗糖果呢？

示例 1：

输入：[1, 0, 2]
输出：3
解释：你可以分发给这三个孩子分发 2、1、2 粒糖果。

```
1 class Solution {
2 public:
3     int candy(vector<int>& ratings) {
4         int n = ratings.size();
5         vector<int> num(n, 1);
6         for(int i = 1; i < n; i++){
7             if(ratings[i] > ratings[i-1])
8                 num[i] = num[i-1] + 1; //num[i] = num[i-1]+1 表示
9         }
10        // for(int i = 1; i < n; i++){
11        //     for(int j = i-1; j > 0; j--){
12        //         if(ratings[i-1] > ratings[j]){
13        //             num[i-1] = max(num[i-1], num[j]+1);
14        //         }
15        //     }
16    }
17 }
```

测试用例 代码执行结果 测试题

先将所有孩子的糖果 num 初始化为 1，再进行两遍遍历，第一遍从左到右，每一次找到右边比左边大的，将 $num[i]$ 更新为前一个值加 1，第二遍从右到左，这里必须是从右边开始，每一次找到比右边元素大的数，若找到，再看这个元素是不是只比右边大 1 or 与右边相等， $num[i-1] = \max(num[i-1], num[i]+1)$ ；若本身大于不上 1 了，就还要更新 $num[i-1]$

这里的贪心策略就是从左到右遍历先比较右边的右规则，再从右到左遍历比较左边的左规则，注意第第二次比较不是直接+1了，而是要判断是否需要更新

Lambda表达式

Lambda 表达式是 C++11 的新特性，它允许程序员在函数内部创建一个匿名函数，C++11 中才支持 lambda 表达式，使用 lambda，可以直接代替回调函数。不需要再去写一个函数，直接在调用的地方写代码
C++ 中的 lambda 表达式可以看作是一个匿名函数，也可以当作一个内联函数来使用。此外，lambda 表达式是一种可调用对象。C++ 中可调用对象还有函数、函数指针和重载了“()”运算符的类等

括号三巨头（中括号、小括号、大括号）就是一个 lambda 表达式

[capture_list] (parameter_list) mutable exception -> return_type {function_body}

435. 无重叠区间
难度 **中等** 403 收藏 1% 通过 34 提交 100
给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。
注意：
 1. 可以认为区间的终点总是大于它的起点。
 2. 区间 [1,2] 和 [2,3] 的边界相互接触，但没有相互重叠。

示例 1：

输入：[[1,2], [2,3], [3,4], [1,3]]
输出：1

解释：移除 [1,3] 后，剩下的区间没有重叠。

```
1 class Solution {
2 public:
3     int eraseOverlapIntervals(vector<vector<int>>& intervals) {
4         if(intervals.empty()) return 0;
5         sort(intervals.begin(), intervals.end(), [] (vector<int> &a, vector<int> &b){return a[1] < b[1];});
6         int n = intervals.size();
7         int r = intervals[0][1]; int cnt = 0;
8         for(int i = 1; i < n; i++){
9             if(intervals[i][0] < r)
10                 cnt++;
11             else
12                 r = intervals[i][1];
13         }
14         return cnt;
15     }
16 }
17 }
```

测试用例 代码执行结果 测试题

这种无重叠区间，消除区间变为无重叠区间必须按照区间尾端点递增排序！！！

贪心策略为优先保留区间结尾小且不相交的区间，先将区间按尾结点从小到大排序，遍历所有区间，首先定义第一个区间的 r ，将每次遍历到的区间 i 的 start 与前面区间的 r 比较，如果小于 r ，则属于重叠区间，计数加 1，如果大于，属于独立区间，将 r 更新为当前区间的 end 元素
intervals[i][1]。这一篇循环结束， $i++$ ，循环继续，inrevals[i][0] 将继续与 r 进行比较，如果小于，计数 +1，如果大于，更新 r 到现在的 interval[i][1]，for 循环体末尾 $i++$ ，循环继续，直到 $i=n-1$ ；最终一个区间，看它的 start 是否小于前面的 r ，再到 $i=n$ ，循环结束，返回这个累加变量 cnt 次数

605. 种花问题
难度 **简单** 343 收藏 1% 通过 0 提交 100
假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花不能种植在相邻的地块上，它们会争夺养分，两者都会死去。
给你一个整数数组 flowerbed 表示花坛，由若干 0 和 1 组成，其中 0 表示没种植物，1 表示种植了花。还有一个数 n ，能否在不打破种植规则的情况下种入 n 朵花？能则返回 true，否则返回 false。

示例 1：

输入：flowerbed = [1,0,0,0,1], n = 1
输出：true

```
1 class Solution {
2 public:
3     bool canPlaceFlowers(vector<int>& flowerbed, int n) {
4         flowerbed.insert(flowerbed.begin(), 0);
5         flowerbed.insert(flowerbed.end(), 0);
6         int cnt = 0;
7         for(int i = 1; i < flowerbed.size()-1; i++){
8             if(flowerbed[i-1]==0 && flowerbed[i] == 0 && flowerbed[i+1] == 0){
9                 cnt++;
10                flowerbed[i] = 1;
11            }
12        }
13        if(cnt >= n) // return cnt >= n;
14        return true;
15        else
16        return false;
17    }
18 }
```

测试用例 代码执行结果 测试题

为避免首位和尾位的特殊情况，可以在首位和尾位都加一个 0，这样就和 000 相同了，只需要找到有多少个 0 的左边和右边都为 0，即满足条件，累加变量加 1，记得要将当前的 flowerbed[i] 的值设为 1，表示这里已经种了朵花了，对应该标记为 1，防止下一次循环 000 序列又找到一个 flowerbed[i]，最终 cnt 如果大于 n ，表示可以加入这么多插入这么多的花朵，每找到一个 000，就将 flowerbed[i] 表示为 1，表示可以插入一朵花朵，一直到遍历完整个 vector

452. 用少数量的箭引爆气球
难度 **中等** 391 收藏 1% 通过 0 提交 100
二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以坐标并不重要。因此只要知道开始和结束的位置就很够了。开始坐标总小于结束坐标。一支弓箭可以射向 x 轴上的不同点完全直地射出，在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 $x_{start} \times x_{end}$ ，且满足 $x_{start} \leq x \leq x_{end}$ ，则该气球会被引爆，可以射出的与之前的数量没有限制，与前一箭被射出之后，可以无限地射出，我们想找到使得所有气球全部被引爆，所需的弓箭的最少数量。
给你一个数组 points，其中 $points[i] = [x_{start}, x_{end}]$ ，返回引爆所有气球所必须射出的最小弓箭数。

示例 1：

输入：points = [[10,16],[2,8],[1,6],[7,12]]
输出：2
解释：对于该样例， $x = 6$ 可以引爆 [2,8], [1,6] 两个气球，以及 $x = 11$ 射爆另外两个气球

```
1 class Solution {
2 public:
3     int findMinArrowShots(vector<vector<int>>& points) {
4         sort(points.begin(), points.end(), [] (vector<int> &a, vector<int> &b){return a[1] < b[1];});
5         int r = points[0][1];
6         int n = points.size();
7         int cnt = n;
8         for(int i = 1; i < n; i++){
9             if(points[i][0] <= r){
10                 cnt--;
11             }
12             else
13                 r = points[i][1];
14         }
15         return cnt;
16     }
17 }
```

测试用例 代码执行结果 测试题

[0, 6] [2, 8] [2, 9] [9, 10] [7, 12]

可以认为，后面这个区间尾部靠得越近，区间重合度越高，这就是使用尾部排序的理由！！！当找到了当前区间头部大于前面区间的尾部，则说明一个区间划分完毕了（合并完了），由于这里是气球，边界相等时仍然划在一个区间内，如果不重叠区间则不能取等号

贪心就是我最多可以射 size 支箭，但是每合并一个区间可以少射一支箭

题目描述 | 评论 (463) | 跟贴 (793) | 提交记录 | C++ | 题解模式

763. 划分字母区间

难度 中等 | 487 | 收藏 | 分享 | 举报

字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

示例：

```
输入: S = "ababcbacadefegdehijhklij"
输出: [9,7,8]
解释:
划分结果为 "ababcbaca", "defegde", "hijklij"。
每个字母最多出现在一个片段中。
像 "ababcbacadefegde", "hijklij" 的划分是错误的，因为划分的片段数较少。
```

核心思想就是先要找出每个元素最后出现的下标/位置处，再从左到右遍历字符串，如果后面没有与当前元素相同元素（通过比较max下标确定是否更新下标），那么当前这一段就是一个区间，到 $end == i$ ，表示划分得到一个区间，一直到递增到字符串结尾处， $i < size()$, i最多可以是 $size() - 1$ ，此时再+1就会不满点条件退出循环体

406. 根据身高重建队列

难度 中等 | 855 | 收藏 | 分享 | 举报

假设有 n 个人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个人 $people[i] = [h_i, k_i]$ 表示第 i 个人的身高 h_i ，需要正好有 k_i 个身高大于或等于 h_i 的人。

请你重新构造并返回队列，数组 people 所表示的队列，返回的队列应该格式化为数组 queue，其中 $queue[i] = [h_j, k_j]$ 是队列中第 j 个人的属性。 $(queue[0] = \text{队列中队首的人})$ 。

示例 1：

```
输入: people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
输出: [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
解释:
身高为 0 的人身高为 s，没有身高更高或者相同的人排在他前面。
身高为 1 的人身高为 t，没有身高更高或者相同的人排在他前面。
身高为 2 的人身高为 r，有 2 个身高更高或者相同的人排在他前面，即排名为 0 和 1 的人。
身高为 3 的人身高为 l，有 1 个身高更高或者相同的人排在他前面，即排名为 1 的人。
```

身高从大到小排（身高相同k小的站前面）

{7 0} {7 1} {6 1} {5 0} {5 2} {4 4}

←

{5 2} 前面一定都比{5 2}高、那么{5 2}可以放心插入下标为2的位置。这样就确定了{5 2}前面一定有两个比它高的元素

按照身高从高到低排序完，再按k升序，前面有几个比自己高，进行插入，因为是从左到右按从高到低的人进行插入，比如，插入(6,1)，前面的人肯定都比6高，插入(7,1)前面的人肯定都7高，在1处插入是没问题的，插入(5,2)，由于前面插入的元素都比5高，在2处插入5表示前面有2人比它高，肯定是正确的，且这个小的数的插入不会影响到前面已经插入的大数的逻辑

局部最优：近每个人的逻辑+来插入

全局最优：由于是按从大到小的顺序插入的，后面的插入操作不影响前面整体的逻辑，也就是全局最优

如(6,1)，在1处，插入完成时前面只会有一个比6高，后面再怎么插入比6小的人都不会影响到前面只有一个人比6高这个逻辑，所以前提是一定要按从大到小的排序，再按元素的k值/位置 插入到vector/list que，得到最终的逻辑队列

665. 非递减数列

难度 中等 | 568 | 收藏 | 分享 | 举报

给你一个长度为 n 的整数数组，请你判断在最多改变 1 个元素的情况下，该数组能否变成一个非递减数列。

我们是这样定义一个非递减数列的：对于数组中任意的 $1 \leq i \leq n-2$ ，总满足 $nums[i] \leq nums[i + 1]$ 。

示例 1：

```
输入: nums = [4,2,3]
输出: true
解释: 你可以通过把第一个4变成1来使得它成为一个非递减数列。
```

思想：

以要采取贪心的策略，在遍历时，每次需要看连续的三个元素，需要尽量不放大 $nums[i+1]$ ，这样会让后续非递减更困难；如果缩小 $nums[i]$ ，但不破坏前面的子序列的非递减性；

遍历数组，如果遇到递减：还能修改：

修改方案1：将 $nums[i]$ 缩小至 $nums[i+1]$ ；

修改方案2：将 $nums[i+1]$ 放大至 $nums[i]$ ；不能修改了：直接返回false

若一次也不需要修改或只修改了一次，即只执行了一次中间的if语句，说明只修改了一次，可以返回true，注意循环是从1开始遍历，需要处理 $num[0]$ 与 $num[1]$ 的关系，并且影响到flag初始值，因为只可以一次修改

第3章 双指针

双指针主要用来进行遍历，协同完成任务，如果双指针指向同一数组，遍历方向相同且不会相交，则也称为滑动窗口，经常用于区间搜索，有时候也有两个指针的遍历方向相反，也可以用来搜索。

第一种：首尾指针

array is sorted

题目描述 | 评论 (33) | 跟贴 (1.3k) | 提交记录 | C++ | 模拟模式

167. 两数之和 II - 输入有序数组

难度: 简单 | 通过: 505 | 提交: 140 | 显示: 0

给定一个已按升序排序的整数数组 `numbers`，请你从数组中找出两个数满足相加之和等于目标数 `target`。

函数应该以长度为 2 的整数数组的形式返回这两个数的下标值。`numbers` 的下标从 1 开始计数，所以答案数组值应当满足 $1 \leq \text{answer}[0] < \text{answer}[1] \leq \text{numbers.length}$ 。

你可以假设每个输入只对应唯一的答案，而且你不能重复使用相同的元素。

```
1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& numbers, int target) {
4         int left = 0, right = numbers.size() - 1;
5         while(left < right){
6             if(numbers[left] + numbers[right] == target)
7                 break;
8             else if (numbers[left] + numbers[right] > target)
9                 right--;
10            else
11                left++;
12        }
13        return vector<int>{left+1, right+1};
14    }
15 }
```

示例 1：

输入: `numbers = [2,7,11,15]`, `target = 9`
输出: `[1,2]`
解释: 2 与 7 之和等于目标数 9，因此 `index1 = 1`, `index2 = 2`。

`++a` 和 `a++` 都是将 a 加 1，但是 `++a` 的返回值是 a，`a++` 的返回值是 `a+1`，如果只是需要递增 a 不需要返回值，则使用 `++a`，运行速度会快一些。

题目描述 | 评论 (1.9k) | 跟贴 (2.5k) | 提交记录 | C++ | 模拟模式

88. 合并两个有序数组

难度: 简单 | 通过: 930 | 提交: 10 | 显示: 0

给你两个有序整数数组 `nums1` 和 `nums2`，请你将 `nums2` 合并到 `nums1` 中，使 `nums1` 成为一个有序数组。

初始化 `nums1` 和 `nums2` 的元素数量分别为 `n` 和 `m`。你可以假设 `nums1` 的空间大小等于 `n + m`。这样它就有足够的空间保存来自 `nums2` 的元素。

示例 1：

```
1 class Solution {
2 public:
3     void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
4         int i = m-1, j = 0, pos = m + n - 1;
5         while(pos >= 0){
6             if(i < 0 && j < n && nums1[i] >= nums2[j]){
7                 nums1[pos] = nums2[j];
8                 pos--;
9                 j++;
10            } else
11                nums1[pos--] = nums1[i--];
12        }
13    }
14 }
```

第二种 快慢指针

142. 环形链表 II

难度: 中等 | 通过: 173 | 提交: 4 | 显示: 0

给定一个链表，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

为了表示链表中的环，我们使用整数 `pos` 来表示链表尾部接到链表中的位置（索引从 0 开始）。如果 `pos` 是 -1，则在该链表中没有环。注意，`pos` 仅仅对于实现该题目，并不会作为参数传递到函数中。

说明：不允许修改链表的节点。

进阶：

- 你能否使用 O(1) 空间解决此题？

示例 1：

```
1 class Solution {
2 public:
3     ListNode *detectCycle(ListNode *head) {
4         ListNode *fast = head, *slow = head;
5         // 如果 head 为 null，直接返回 null。
6         while(fast && fast->next){
7             fast = fast->next->next;
8             slow = slow->next;
9             // 如果它们相遇了，那么就存在环。
10            if(fast == slow) {
11                // 将它们都向后走一步，快一步慢一步地走，直到 fast = slow 时，它们相遇了。
12                fast = head;
13                while(fast != slow) {
14                    fast = fast->next;
15                    slow = slow->next;
16                }
17                return fast;
18            }
19        }
20        return NULL;
21    }
22 }
```

输入: `head = [3,2,0,-4]`, `pos = 1`
输出: `[3,2,0]`
解释: 在链表中有一个环，其尾部连接到第二个节点。

环的入口点：第二次相遇时的结点为入口结点

第三种 滑动窗口

题目描述 | 评论 (63) | 跟贴 (91) | 提交记录 | C++ | 模拟模式

76. 最小覆盖子串

难度: 困难 | 通过: 1125 | 提交: 4 | 显示: 0

给你一个字符串 `s`，返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `" "`。

注意：字符串中可能存在相同的字母。

示例 1：

```
1 class Solution {
2 public:
3     string minWindow(string s, string t) {
4         unordered_map<char, int> map;
5         for(char c : t)
6             map[c]++;
7         int left = 0, right = 0, need = t.size(), start = 0;
8         for(int right = 0; right < s.size(); right++){
9             if(map[s[right]] > 0)
10                 need--;
11             map[s[right]]--;
12             if(need == 0) { // 表示 t 中所有字符都出现的次数 - map[right] 可以保证 t 中有的字符不会一开始就减为 0。
13                 int min_len = right - left + 1; int min_left = INT_MAX; int start = 0;
14                 start = left;
15                 min_left = right - left + 1;
16             }
17             if(++map[s[left]] == 0) // 表示 s 中字母各减一个，还需要减 1，直到需要数 == t.size()。
18                 need++;
19             left++;
20         }
21     }
22 }
```

输入: `s = "ADOBECODEBANC"`, `t = "ABC"`
输出: `"BANC"`

示例 2：

输入: `s = "a"`, `t = "a"`
输出: `"a"`

提示：

- $1 \leq s.length, t.length \leq 10^5$
- s 和 t 由英文字母组成

力扣 学习 题库 讨论 完成 求职 商店

题目描述 | 评论 (211) | 跟贴 (321) | 提交记录 | C++ | 模拟模式

633. 平方数之和

难度: 中等 | 通过: 172 | 提交: 4 | 显示: 0

给定一个非负整数 `c`，你要判断是否存在两个整数 `a` 和 `b`，使得 $a^2 + b^2 = c$ 。

示例 1：

```
1 class Solution {
2 public:
3     bool judgeSquareSum(int c) {
4         long left = 0, right = sqrt(c);
5         while(left <= right){
6             long long sum = left*left + right*right;
7             if(sum == c)
8                 return true;
9             else if (sum > c)
10                right--;
11            else
12                left++;
13        }
14    }
15 }
```

long long本质上还是整型，只不过是一种超长的整型。
int型: 32位整型，取值范围为 $-2^{31} \sim (2^{31}-1)$ 。

long : 在32位系统是32位整型, 取值范围为-2^31 ~ (2^31 - 1); 在64位系统是64位整型, 取值范围为-2^63 ~ (2^63 - 1)
long long : 是64位的整型, 取值范围为-2^63 ~ (2^63 - 1)

c/c++中int , long , long long 的取值范围 :

```
unsigned int 0~4294967295  
int -2147483648~2147483647  
unsigned long 0~4294967095  
long -2147483548~2147483547  
long long的最大值 : 9223372036854775807  
long long的最小值 : -9223372036854775808  
unsigned long long的最大值 : 18446744073709551615 /205
```

类型	占用存储空间	数值范围
byte	1字节	-2^7 ~ 2^7-1 (-128~127)
short	2字节	-2^15 ~ 2^15-1 (-32768~32767)
int	4字节	-2^31 ~ 2^31-1 (-2147483648~2147483647) f9210
long	8字节	-2^63 ~ 2^63-1

680. 验证回文字符串 II
难度 普通 通过数 143 ★ ○ % ○ □
给定一个非空字符串 s，最多删除一个字符。判断是否能成为回文字符串。
示例 1:
输入: "aba"
输出: True
示例 2:
输入: "abca"
输出: True
解释: 你可以删除c字符。

说是说删除，其实不一定真的要erase那个元素，可以采用游标加1或者减1的方式巧妙的删除

524. 通过删除字母匹配到字典里最长单词
难度 中等 通过数 142 ★ ○ % ○ □
给定一个字符串和一个字符串字典，找到字典里面最长的字符串，该字符串可以通过删除给定字符串的某些字符来得到。如果答案不止一个，返回长度最长且字典序最小的字符串。如果答案不存在，则返回空字符串。
示例 1:
输入:
s = "abpcplea", d = ["ale","apple","monkey","plea"]
输出:
"apple"
示例 2:
输入:
s = "abpcplea", d = ["a","b","c"]
输出:
"a"
说明:
所有输入的字符串只包含小写字母。

```
1 class Solution {  
2 public:  
3     bool check_sub(string src, string dst){  
4         //for(int i=0, j=0;  
5         int i = 0, j = 0;  
6         for(;i<src.size() && j < dst.size(); i++){  
7             if(src[i] == dst[j])  
8                 j++;  
9             else  
10                j++;  
11         }  
12         return j == dst.size();  
13     }  
14     string findLongestWord(string s, vector<string>& dictionary) {  
15         /*  
16         bool cmp(string &a, string &b){ //函数内部定义函数 ??????????????????  
17             if(a.size() <= b.size()) //只能调用，不可以再定义函数  
18                 return a.size() < b.size();  
19             return a.size() > b.size();  
20         */  
21         auto cmp = [] (string &a, string &b){ //函数内部可以定义lambda表达式，并将它赋值给一个auto cmp，只是  
22             if(a.size() == b.size())  
23                 return a < b; //注意这条语句还是一个赋值语句，需要加；  
24             return a.size() > b.size();  
25         };  
26         sort(dictionary.begin(), dictionary.end(), cmp);  
27         for(int i = 0; i < dictionary.size(); i++){  
28             if(check_sub(s,dictionary[i]))  
29                 return dictionary[i];  
30         }  
31         return "";  
32     }  
33 }
```

先将字典中的字符串排序，优先级是长度，长度相等按字典序排，直接a**<**b；
再定义一个判断字符串的函数，怎么判断？双指针check_sub，如果s[i]==s[j]：则++；每一次循环以++遍历源串，直到循环结束，
看j的长度是不是递增到了i的size，这其中的++，就相当于删除操作！！！
对排序完的字典取出第一个符合条件的字符串

第四章 二分查找 前提是单调函数/区间

二分法，折半查找，每次查找将区间分成两部分，并取一部分继续查找长为O(n)的数组时间复杂度为O(logn)

如一个长度为5的数组，如果遍历最多要找5次/5次循环，如果二分，2次就可以找到

如果二分到不能再分了，区间找不到满足条件的解

区间写法：左闭右开 / 左闭右闭(可处理边界)

二分查找是每次移动半个区间

69.x 的平方根
难度 普通 通过数 664 ★ ○ % ○ □
实现 int sqrt(int x) 函数。
计算其返回 x 的平方根，其中 x 是非负整数。
由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。
示例 1:
输入: 4
输出: 2
示例 2:
输入: 8
输出: 2
说明: 8 的平方根是 2.82842...，
由于返回类型是整数，小数部分将被舍去。

```
1 class Solution {  
2 public:  
3     int mySqrt(int x) {  
4         if(x == 0) return 0;  
5         int left = 1, right = x/2;从 i 开始  
6         while(left <= right){  
7             int mid = left + (right - left)/2;  
8             if(x==mid*mid)  
9                 return mid;  
10            else if(x>mid*mid)  
11                left = mid + 1;  
12            else  
13                right = mid - 1;  
14        }  
15        //自己加一个校验性一撇以确定边界条件  
16        return right; //这里一定不能return right - 1 !!! 因为left<right，还会执行，如果 x/mid < mid; right = mid - 1  
17    }  
18 }
```

while(left <= right) 与 while(left < right) 写法的区别
这一部分也来自评论区网友的提问，我的理解如下：
首先说它们最主要的区别：
• while(left <= right) 在退出循环时有 left = right + 1，即 right 在左，left 在右；
• while(left < right) 在退出循环时有 left == right 成立。
我的经验是 left <= right 用在简单的二分题目中，如果题目要求找的数的性质很简单，可以用这种写法，在细节里体现到了跳出。
在一些复杂的题目中，例如第一题一维界的二分 (就像我做这个题目)，用 while(left < right) 其实更简单的，把要找的数移到最后，在退出循环之后，将觉得最需要的界限退出循环之后有 left == right 成立，这种思考问题的方式不容易出错。
while(left < right) 可以避免在判断：初界的时候就很麻烦地出现死循环的嫌疑，特别是很烦理解分支的取舍决策中问题的疑惑。不过通过练习和理解，把这个关过了，相信解决一些程度较大的部分查找问题也就顺利而容易了。建议大家尝试使用 while(left < right) 的方式去解决一些较困难的问题。

题目描述 [] [] [] []

34. 在排序数组中查找元素的第一个和最后一个位置
难度: 中等
贡献者: 魏

给定一个按升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。
如果数组中不存在目标值 target，返回 [-1, -1]。
进阶：
• 你可以设计并实现时间复杂度为 O(log n) 的算法解决此问题吗？

示例 1：
输入：nums = [5,7,7,8,8,10], target = 8
输出：[3,4]

示例 2：
输入：nums = [5,7,7,8,8,10], target = 6
输出：[-1,-1]

```

1 class Solution {
2 public:
3     int binarySearch(vector<int>& nums, int target, bool lower) {
4         int left = 0, right = (int)nums.size() - 1, ans = (int)nums.size();
5         while(left <= right) {
6             int mid = (left + right) / 2;
7             if (lower || (nums[mid] == target)) {
8                 right = mid - 1;
9             } else {
10                 left = mid + 1;
11             }
12         }
13         return ans;
14     }
15
16     vector<int> searchRange(vector<int>& nums, int target) {
17         int leftIdx = binarySearch(nums, target, true);
18         int rightIdx = binarySearch(nums, target, false) - 1;
19         if (leftIdx <= rightIdx && nums[leftIdx] == target) {
20             return vector<int>{leftIdx, rightIdx};
21         }
22         return vector<int>{-1, -1};
23     }
24 }
```

直观的思路肯定是从前往后遍历一遍，用两个变量记录第一次和最后一次遇见target的下标，但这个方法的时间复杂度为O(n)，没有利用到数组升序排列的条件
单独对nums[mid]==target的情况进行判断
如果要找第一个等于target的元素，那么就需要在nums[mid]==target的时候看一下前面的那个元素是否也等于target
如果要找的是最后一个等于target的元素，那么就需要在nums[mid]==target的时候看后一个元素是否等于target

```

class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int f=searchFirst(nums,target);
        int l=searchLast(nums,target);
        return {f,l};
    }
    int searchFirst(vector<int>& nums,int target){
        int left = 0;
        int right = nums.size()-1;
        while(left<right){
            int mid = (right - left)/2 + left;
            if(nums[mid]<target){
                left = mid + 1;
            }else if(nums[mid]==target){
                right = mid - 1;
            }else{
                // 如果当前元素已经是数组的第一个元素了，那么无需再向左看了，直接返回
                // 如果不是第一个元素，则需要看看前面是否还有元素满足条件
                if(mid == 0 || nums[mid-1]==target) return mid;
                else right = mid - 1;
            }
        }
        return -1;
    }
    int searchLast(vector<int>& nums,int target){
        int left = 0;
        int right = nums.size()-1;
        while(left<right){
            int mid = (right - left)/2 + left;
            if(nums[mid]<target){
                left = mid + 1;
            }else if(nums[mid]==target){
                right = mid - 1;
            }else{
                // 如果当前元素已经是数组的最后一个元素了，那么无需再向右看了，直接返回
                // 如果不是最后一个元素，则需要看看后面是否还有元素满足条件
                if(mid == nums.size()-1 || nums[mid+1]==target) return mid;
                else left = mid + 1;
            }
        }
        return -1;
    }
}
```

若arr只有一个元素，返回为[0,0]，此外这两个函数是单独按顺序运行的，
如果是空数组left=0, right=-1，空数组应该返回[-1,-1]，所以要有left<=right条件 再放入vector，否则要放入[-1,-1]
且数组中的确要有这个数因此要有num[left]==target；
第一个（一直查找）大于等于这个数的下标==上界，第一个大于（一直查找）这个数的下标-1 == 这个数的出现下界

81. 搜索旋转排序数组 II
难度: 中等
贡献者: 魏

已知存在一个按非降序排列的整数数组 nums，数组中的值不必互不相同。
在传递给你的未知数组 nums 中是否存在一个元素 target，使得 nums[i] < i < nums.length - 1 且满足
nums[i-1] >= nums[i] >= nums[i+1]，即 nums[i-1] >= target >= nums[i+1]。（下标从 0 开始计数，例如，[0,1,2,4,4,4,5,6,6,7] 在下标 3 处是加转后可成为 [4,5,6,6,7,0,1,2,4,4]。）
给你 旋转后的数组 nums 和一个目标值 target，编写一个函数来判断
数组中的目标值是否存在于数组中，如果 nums 中存在目标值
target，则返回 true，否则返回 false。

示例 1：
输入：nums = [2,5,6,0,0,1,2], target = 0
输出：true

示例 2：
输入：nums = [2,5,6,0,0,1,2], target = 3
输出：false

```

1 class Solution {
2 public:
3     bool search(vector<int>& nums, int target) {
4         if (nums.size() == 0) return false;
5         int start = 0, end = nums.size() - 1;
6         while(start <= end) {
7             int mid = start + (end - start) / 2;
8             if (nums[mid] == target) // 这个数字必须在前面，因为下面有 continue语句
9                 return true;
10            if (nums[mid] >= nums[start]) // 如果中间那一部分有序，只能<start，越过一位再判断剩余元素
11                start = mid + 1;
12            else {
13                continue;
14            }
15            if (nums[mid] < nums[start]) // 右区间有序
16                if (target >= nums[start] && target <= nums[mid]) // 如果元素在右区间那就在这个区间中寻找
17                    start = mid + 1;
18                else
19                    end = mid - 1;
20            else
21                if (target >= nums[start] && target <= nums[mid]) // 同样先看target元素是否在这个区间内，依据这个条件更新得范围
22                    end = mid - 1;
23                else
24                    start = mid + 1;
25        }
26        return false;
27    }
}
```

第一遍二分查找找到转折点，转折点将数组分为前后两个递增序列。

然后判断target值在哪个区间，如果两个区间都在，选择前半段区间（优先用前面部分）

在选定区间中用二分查找第一个不大于target的位置，找到后判断是否等于target，不是则返回-1

LC-旋转数组

题目描述 评论 (131) 答案 (167) 提交记录

面试题 10.03. 搜索旋转数组

难度 中等
贡献 49 收藏 13 热度 0

搜索旋转数组。给定一个排序后的数组，包含n个整数，但这个数组已被旋转过很多次了，次数不定。请编写代码找出数组中的某个元素。假设数组元素原先先是升序排列的。若有多个相同元素，返回索引值最小的一个。

示例1：

```
输入：arr = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10], target = 5
输出：0 (元素5在该数组中的索引)
```

示例2：

```
输入：arr = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10], target = 11
输出：-1 (没有找到)
```

提示：

- arr 长度范围在[1, 1000000]之间

通过次数 9,204 提交次数 23,011

请问您在精英招聘中遇到此题？

杜鹃 桦柏 实习 来面试

因为数组默认升序排列，所以无论数组怎么旋转，只需要找到头尾衔接点分别对左右进行二分查找即可
左右两边的数组一定是升序排列的，优先对左边进行二分查找，这个题是要返回第一个出现的索引

题目描述 评论 (1,4k) 答案 (1,8k) 提交记录

33. 搜索旋转排序数组

难度 中等
贡献 1338 收藏 13 热度 0

整数数组 nums 接受升序排列，数组中的值互不相同。

在迭代构造之前，nums 在预先未知的情况下 k ($0 \leq k < \text{nums.length}$) 上进行了旋转，使数组变为 $[\text{nums}[k], \text{nums}[k+1], \dots, \text{nums}[n-1], \text{nums}[0], \text{nums}[1], \dots, \text{nums}[k-1]]$ (下标从 0 开始计数)。例如， $[0,1,2,4,5,6,7]$ 在 T3 次翻转后可能变为 $[4,5,6,7,0,1,2]$ 。
给你旋转后的数组 nums 和一个整数 target，如果 nums 中存在这个目标值 target，则返回它的下标，否则返回 -1。

示例 1：

```
输入：nums = [4,5,6,7,0,1,2], target = 0
输出：4
```

示例 2：

```
输入：nums = [4,5,6,7,0,1,2], target = 3
输出：-1
```

这种不包含重复数字的旋转数组就可以直接按左右部分进行搜索就行了，只是注意搜索左边部分 if($\text{nums}[\text{mid}] \geq \text{nums}[\text{left}]$) 表示左边部分是有序的，这里是用的 \geq 表示左边区间

题目描述 评论 (524) 答案 (754) 提交记录

154. 寻找旋转排序数组中的最小值

难度 困难
贡献 353 收藏 13 热度 0

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次 旋转 后，得到插入数组。例如，原数组 $[\text{nums}[0], \text{nums}[1], \text{nums}[2], \dots, \text{nums}[n-1]]$ 在变化后可能得到：

- 若旋转 4 次，则可以得到 $[\text{nums}[4], \text{nums}[5], \text{nums}[6], \text{nums}[7], \text{nums}[0], \text{nums}[1], \text{nums}[2], \text{nums}[3]]$
- 若旋转 7 次，则可以得到 $[\text{nums}[7], \text{nums}[8], \text{nums}[9], \text{nums}[0], \text{nums}[1], \text{nums}[2], \text{nums}[3], \text{nums}[4], \text{nums}[5], \text{nums}[6]]$

注意，数组 $[\text{nums}[0], \text{nums}[1], \text{nums}[2], \dots, \text{nums}[n-1]]$ 旋转一次的结果为数组 $[\text{nums}[n-1], \text{nums}[0], \text{nums}[1], \text{nums}[2], \dots, \text{nums}[n-2]]$ 。
给你一个可能存在 重复 元素值的数组 nums，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 最小元素 。

示例 1：

```
输入：nums = [1,3,5]
输出：1
```

3412 模拟

首先，创建两个指针 leftleft, rightright 分别指向 numbersnumbers 首尾数字，然后计算出两指针之间的中间索引值 middlemiddle，然后我们会遇到以下三种情况：

middle > right : 代表最小值一定在 middle 右侧，所以 left 移到 middle + 1 的位置。左侧有序且递增

middle < right : 代表最小值一定在 middle 左侧或者就是 middle，所以 right 移到 middle 的位置。右侧有序且递增序列

middle 既不大于 left 指针的值，也不小于 right 指针的值，代表着 middle 可以等于 left 指针的值，或者 right 指针的值，我们这时候只能让 right 指针递减，来一个一个找最小值了

第三种情况就是以 mid 为中心无法判断哪一部分有序 $[1, 0, 1, 1, 1]$; $[1, 1, 1, 0, 1]$ 类似于前面与 start 比较同理，这也是有重复元素的处理方法，直接增减指针跳过这段比较范围 while 中是 \leq 一般在 while 循环体内会有 return 值，如果 while 中是 $<$ ，一般在 while 退出再会有 return 语句 (while 退出表示到了 left == right)

题目描述 评论 (255) 答案 (277) 提交记录

540. 有序数组中的单一元素

难度 中等
贡献 230 收藏 13 热度 0

给定一个只包含整数的有序数组，每个元素都会出现两次，唯有一个数只会出现一次，找出这个数。

示例 1：

```
输入：[1,1,2,3,3,4,4,8,8]
输出：2
```

示例 2：

```
输入：[3,3,7,7,10,11,11]
输出：10
```

注意：您的方案应该在 $O(\log n)$ 时间复杂度和 $O(1)$ 空间复杂度中运行。

第五章 排序算法

堆排序

堆 heap 是一个完全二叉树的数组，树上的第一个结点对应数组的一个元素，除了最底层，该树是完全填满的，而且是从左到右填充。
最大堆的每一个节点的值不大于父结点，根结点为最大值

堆的实现：由于是一棵完全二叉树，可以用数组来实现，数组下标对应堆中结点编号
堆建：将一个乱序的数组变成堆结构的数组，时间复杂度为 $O(n)$
push：将一个数值放进已经是堆结构的数组，时间复杂度为 $O(\log n)$

`pop` : 从最大堆中取出最大值或最小堆中取出最小值，并将剩余的数组保持堆结构 $O(\log n)$

建堆：堆是完全二叉树的数组，
堆中每个结点与其左右子结点都有编号对应关系
`left = parent * 2 + 1; right = parent * 2 + 2`

接下来对这棵树进行调整，从最后一个非叶子结点开始到根结点，每个结点都做下沉操作
下沉：从当前父结点的左右孩子结点中选出较大的那个，如果较大的值小于父结点，则跳出，
否则将它的值赋给父结点，
接着将父结点索引定位到上一步中选出的那个较大结点的位置，
不断下沉，直到当前结点不再有左结点，（也就没有右结点，即为叶子结点）

对于取topk个数 -堆排序解决topk问题

对于本题只涉及到建堆和push两个操作
首先对arr数组的前k个数建堆
再从k开始对剩下的数组进行遍历，每个元素都和堆顶元素比较
如果当前元素比堆顶元素大则不处理（这里是最大堆）；如果当前元素小则将堆顶元素
转换为当前元素，并调整堆结构
求最小的k个元素可以用大顶堆，求最大的k个数可以用小顶堆求解
这个就是搭配`<vector<int>`手写堆结构的实现过程

使用priority_queue (大根堆)

优先队列的定义：

它允许用户为队列中元素设置优先级，放置元素的时候不是直接放到队尾，而是放置到比它优先级低的元素前面，标准库默认使用`<`操作符来确定优先级关系。

注意：这里的小于号`<`规定了优先级，表示优先队列后面的元素都要小于优先队列前面的元素，因为优先队列队首的元素优先级最高，优先队列队尾元素的优先级最低，所以小于号`<`就规定了优先队列后面的元素都要小于优先队列前面的元素（尾部优先级小于头部优先级），也就是形成一个大根堆，降序排序，每次权值最大的会被弹出来

小根堆实现：

使用函数对象`greater<int>`来生成小根堆

注意：这里的大于号`>`规定了优先级，表示优先队列后面的元素都要大于优先队列前面的元素，因为优先队列队首的元素优先级最高，优先队列队尾元素的优先级最低，所以大于号`>`就规定了优先队列后面的元素都要大于优先队列前面的元素（尾部优先级小于头部优先级），也就是形成一个小根堆，升序排序，每次权值最小的会被弹出来。

`priority_queue<int, vector<int>, greater<int> > test;`

优先队列的用法：

使用`priority_queue`需要引用头文件`#include <queue>`
支持的顺序容器：`vector`, `queue`, 默认是`vector`。
`priority_queue`类，能按照有序的方式在底层数据结构中执行插入、删除操作。
priority queue的特有操作：
`q.pop()`: 删除优先队列`priority_queue`的最高优先级元素 (通过调用底层容器的`pop_back()`实现)
`q.push(item)`: 在`priority_queue`优先级顺序合适的位置添加一个值为`item`的元素 (通过调用底层容器的`push_back()`操作实现)
`q.emplace(args)`: 在`priority_queue`优先级顺序合适的位置添加一个由`args`构造的元素 (通过调用底层容器的`emplace_back()`操作实现)
`q.top()`: 返回`priority_queue`的首元素的引用 (通过调用底层容器的`front()`操作实现)
`q.empty()`: 判断`q`是否为空，空返回`true`，否则返回`false` (通过调用底层容器的`empty()`操作实现)
`q.size()`: 返回`q`中的元素个数 (通过调用底层容器的`size()`操作实现)
`swap(q,p)`: 交换两个优先队列`priority_queue` p, q的内容，p和q的底层容器类型也必须相同 (通过调用底层容器的`swap()`操作实现)
`q`

```
class Solution {
public:
    vector<int> getLeastNumbers(vector<int>& arr, int k) {
        vector<int> res;
        priority_queue<int> heap; //大根堆
        for ( auto x : arr){
            heap.push(x);
            if (heap.size() > k) heap.pop();
        }
        while (heap.size()) res.push_back(heap.top()), heap.pop();
        return res;
    }
};
```

复杂度分析

时间复杂度： $O(n\log k)$ ，其中 n 是数组 arr 的长度。由于大根堆实时维护前 k 小值，所以插入删除都是 $O(\log k)$ 的时间复杂度，
每个元素都需要进行一次入堆操作，所以一共需要 $O(n\log k)$ 的时间复杂度。

空间复杂度： $O(k)$ ，因为大根堆里最多 k 个数

最小的k个数，大根堆

方法一：堆

比较直观的想法是使用堆数据结构来辅助得到最小的 k 个数。堆的性质是每次可以找出最大或最小的元素。
我们可以使用一个大小为 k 的最大堆（大顶堆），将数组中的元素依次入堆，当堆的大小超过 k 时，便将堆出的元素从堆顶弹出。我们以数组 $[5, 4, 1, 3, 6, 2, 9]$ ， $k = 3$ 为例演示元素入堆的过程，如下面图所示：



这样，由于每次从堆顶弹出的数都是堆中最大的，最小的 k 个元素一定会留在堆里。这样，把数组中的元素全部入堆之后，堆中剩下的 k 个元素就是最大的 k 个数了。

两种方法的优劣性比较

在面试中，另一个常常问的问题就是这两种方法有何优劣。看起来分治法的快速选择算法的时间、空间复杂度都优于使用堆的方法，但是要注意到快速选择算法的几点局限性：

第一，算法需要修改原数组，如果原数组不能修改的话，还需要拷贝一份数组，空间复杂度就上去了。

第二，算法需要保存所有的数据。如果把数据看成输入流的话，使用堆的方法是来一个处理一个，不需要保存数据，只需要保存 k 元素的最大堆。而快速选择的方法需要先保存下来所有的数据，再运行算法。当数据量非常大的时候，甚至内存都放不下的时候，就麻烦了。所以当数据量大的时候还是用基于堆的方法比较好。

快速选择一般用于求解 k-th Element 问题，可以在 $O(n)$ 时间复杂度， $O(1)$ 空间复杂度完成求解工作。

快速选择的实现和快速排序相似，不过只需要找到第 k 大的枢（pivot）即可，不需要对其左右再进行排序。
与快速排序一样，快速选择一般需要先打乱数组，否则最坏情况下时间复杂度为 O(n²)

快排的时间复杂度：n log(n)

当数组是有序时：每一次划分都相当于白划分了，需要递归排序n次子数组，排一次需要划分n次，因此为O(n²)

```
1 class Solution {
2 public:
3     int findTheLargest(vector<int>& nums, int k) {
4         int target = nums.size()-k;
5         int l = 0, r = nums.size()-k;
6         while(l < r) {
7             int pivot = quick_select(nums,l,r);
8             if(pivot == target)
9                 return nums[l];
10            if(pivot < target)
11                l = pivot + 1;
12            if(pivot > target)
13                r = pivot - 1;
14        }
15        return nums[l];
16    }
17    int quick_select(vector<int> &nums, int l, int r) {
18        int i = l, j = r;
19        while(i < j) {
20            while(i < j && nums[j] >= nums[i]) j--;
21            while(i < j && nums[i] <= nums[j]) i++;
22            swap(nums[i],nums[j]);
23        }
24        swap(nums[l],nums[i]);
25        return i;
26    }
27 }
28 }
```

/* 这种是只取前k个元素，不要求前面中间元素有序，而本题是要求第k个最大的元素，显然若只取前面k个元素

那第n-k位并不一定是第k大的那个元素

```
void quick_sort(vector<int> &nums, int k, int l, int r){
    while(l >= r)
        return ;
    int i = l, j = r;
    while(i < j) {
        while(i < j && nums[j] >= nums[i]) j--;
        while(i < j && nums[i] <= nums[j]) i++;
        swap(nums[i],nums[j]);
    }
    swap(nums[l],nums[i]);
    if(i > k) quick_sort(nums, k, l, i-1);
    if(i < k) quick_sort(nums, k, i+1, r);
}
```

定义：priority_queue<Type, Container, Functional>

Type 就是数据类型，Container 就是容器类型（Container必须是用数组实现的容器，比如vector, deque等等，但不能用 list。STL里面默认用的是vector），Functional 就是比较的方式。

当需要用自定义的数据类型时才需要传入这三个参数，使用基本数据类型时，只需要传入数据类型，默认是大顶堆。

一般：

```
1 //升序队列，小顶堆
2 priority_queue<int,vector<int>,greater<int> > q;
3 //降序队列，大顶堆
4 priority_queue<int,vector<int>,less<int> > q;
5
6 //greater和less是std实现的两个仿函数（就是使一个类的使用看上去像一个函数，其实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类）
```

首先要包含头文件#include<queue>，他和queue不同的就在于我们可以自定义其中数据的优先级，让优先级高的排在队列前面优先出队。

优先队列具有队列的所有特性，包括队列的基本操作，只是在这基础上添加了一个排序，它本质上是一个堆实现的。

和队列基本操作相同：

- top 访问队头元素
- empty 队列是否为空
- size 返回队列内元素个数
- push 插入元素到队尾（并排序）
- emplace 原地构造一个元素并插入队列
- pop 弹出队头元素
- swap 交换内容

用pair做优先队列元素的例子：

规则：pair的比较，先比较第一个元素，第一个相等比较第二个。

```
1 class Solution {
2 public:
3     vector<int> topKFrequent(vector<int>& nums, int k) {
4         map<int, int> map;
5         for (int num : nums)
6             map[num]++;
7         priority_queue<pair<int, int>, vector<pair<int, int> , less<> > priorityQueue;
8         for (auto i=map.begin(); i != map.end(); i++) {
9             priorityQueue.push(make_pair(i->second, i->first));
10        }
11        vector<int> ans;
12        for (int i(0); i < k; i++) {
13            auto item = priorityQueue.top();
14            priorityQueue.pop();
15            ans.push_back(item.second);
16        }
17        return ans;
18    }
19 }
```

最大堆排

首先考虑使用堆排序

- 思路：首先在map中统计每一个数字出现的频次，然后把这些数字->频次的pair放到priority_queue中，然后取出前k个就好了

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        map<int, int> map;
        for (int num: nums)
            map[num]++;
        priority_queue<pair<int, int>, vector<pair<int, int>>, less<> priorityQueue;
        for (auto i=map.begin(); i!=map.end(); i++) {
            priorityQueue.push(make_pair(i->second, i->first));
        }
        vector<int> ans;
        for (int i=0; i<k; i++) {
            auto item = priorityQueue.top();
            priorityQueue.pop();
            ans.push_back(item.second);
        }
        return ans;
    }
};
```

时间复杂度：O(nlogn)，可以忽略统计需要的O(n)

空间复杂度：O(n)

54 3 1 大根堆，每次都弹出最大的那个元素，最终剩下最小的那个k个元素

1345 小根堆，每次都弹出最小的那个元素，最终剩下最大的那k个元素

最大的k个数 最小堆

优先队列的性质就是自动排序

方法一：最小堆排 O(Nlogk)

- 1.优先队列 priority_queue
- 2.最小堆（升序队列）：priority_queue <int,vector<int>,greater<int> > q //本方法用最小堆
- 3.最大堆（降序队列）：priority_queue <int,vector<int>,less<int> > q
- 4.优先队列和普通队列的区别：在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除，队头用 q.top() 而不是 q.front()
- 5.本题采用最小堆，若当前出现的元素次数大于优先队列中某个元素的次数，则替换

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> map;
        for(int i : nums) map[i]++; //遍历
        priority_queue< pair<int,int>, vector< pair<int,int> >, greater< pair<int,int> > q; //最小堆
        for(auto it : map)
            if(q.size() == k) { //队列满了
                if(it.second > q.top().first) { //堆排
                    q.pop();
                    q.push(make_pair(it.second, it.first));
                }
            }
            else q.push(make_pair(it.second, it.first));
        vector<int> res;
        while(q.size()) { //将优先队列中k个高频元素存入vector
            res.push_back(q.top().second);
            q.pop();
        }
        return vector<int>(res.rbegin(), res.rend());
    }
}
```

sort类似于快速排序，但是比快速排序更优化，所以时间复杂度应该也为O(Nlog(N))，堆排序，时间复杂度应该也为O(Nlog(N))

堆排序时间复杂度是nlogn没错，但是这道题只需要找前k个最大值，所以只需构建最多k个元素的堆，

时间复杂度为nlogk，优于nlogn。这题明确说了时间复杂度要优与nlogn，所以还是需要用堆做。

随后我们考虑桶排序

- 思路：为每一个数字设置一个桶，桶中存放着字典出现的次数，然后对桶进行排序，实际上，上面的算法就是使用堆排序的桶排序法进行排序，缺点是空间复杂度对指针进行排序，具体而言，因为初始化需要开辟k个桶，然后遍历归并，把每个归并放到桶桶中。最后，我们从桶桶由大到小取出k放入ans并返回。

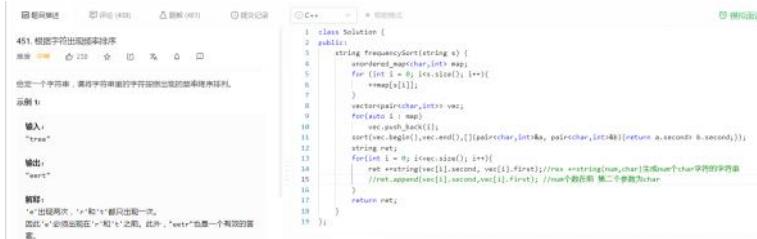
```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> counts;
        int maxCount = 0;
        for(int num: nums)
            maxCount = max(maxCount, ++counts[num]);
        vector<vector<int>> buckets(maxCount+1);
        for(int num: nums)
            buckets[counts[num]].push_back(num);
        vector<int> ans;
        for(int i=maxCount; i>=0; i--) {
            for(int num: buckets[i])
                if(ans.size() < k) ans.push_back(num);
        }
        return ans;
    }
};
```

时间复杂度：O(n)，桶排序的时间复杂度

空间复杂度：O(a + b)，a为不同数字的个数，b为最大频次

桶排序其实就只是计数排序的一个扩展而已，按照要求(某个量)初始n+1个桶 0-10 就初始11个桶 (max-min)+1

如果是区间跨度，则有 (max-min) / span + 1 个桶，因为要保证区间是左开右闭的区间



使用桶排序方法解题

• 思路：首先我们统计出每个字母的个数保存在map中，同时得出最大的次数。随后，我们构造最大的次数+1个桶，然后在根据每个数字出现的频次将其放到对应的桶中。最后，对这个频次桶序列，从后到前遍历，将频次个字母放到ans中。

```
class Solution {
public:
    string frequencySort(string s) {
        unordered_map<char, int> map;
        int maxCount = 0;
        for (char c : s) {
            maxCount = max(maxCount, ++map[c]);
        }
        vector<vector<int>> buckets(maxCount + 1);
        for (auto i = map.begin(); i != map.end(); i++) {
            buckets[i->second].push_back(i->first);
        }
        string ans;
        //有maxCount个桶次桶
        for (int i = maxCount; i >= 1; i--) {
            //从桶次大的元素进行遍历，按次之，这个桶里面的元素依次一样
            for (int j = 0; j < buckets[i].size(); j++) {
                //把这个字母放到ans中i次，也就是频次次
                for (int z = 0; z < i; z++)
                    ans.push_back(buckets[i][j]);
            }
        }
        return ans;
    }
}
```

思路二：优先队列

代码

```
class Solution {
public:
    string frequencySort(string s) {
        unordered_map<char, int> map;
        for (const auto &c : s) {
            ++map[c];
        }
        priority_queue<pair<int, char>, vector<pair<int, char>> pq;
        for (const auto &m : map) {
            pq.push({m.second, m.first});
        }
        string ret;
        while (!pq.empty()) {
            auto t = pq.top();
            pq.pop();
            ret.append(t.first, t.second);
        }
        return ret;
    }
}
```

因为优先队列默认是大根堆，且默认容器类型是vector，且优先队列对于pair类型的默认比较方式先比较第一个元素，因此在构造pair时可以先构造mp.second作为pair的第一个元素，mp.first作为pair的第二个元素，再将pair push到队列

STL priority_queue 自定义排序实现方法

注意，C++ 中的 struct 和 class 非常类似，前者也可以包含成员变量和成员函数，因此上面程序中，函数对象类 cmp 也可以用 struct 关键字创建：

```
01. struct cmp
02. {
03.     //重载 () 运算符
04.     bool operator()(T a, T b)
05.     {
06.         return a > b;
07.     }
08. };
```

由此可见，在 cmp 结构体中重载的 () 运算符中自定义排序规则，并将其实例化后作为 priority_queue 模板的第 3 个参数传入，即可实现为 priority_queue 容器类自定义比较函数。

要想知道此理解这种方式的实现原理，首先要搞清楚 std::less<T> 和 std::greater<T> 各自的底层实现。实际上，<function> 头文件中 std::less<T> 和 std::greater<T>，各自底层都采用的是函数指针实现的。比如，std::less<T> 的底层实现代码为：

```
01. template <typename T>
02. struct less {
03.     //重载 < 运算符
04.     bool operator()(const T &lhs, const T &rhs) const {
05.         return _lhs < _rhs;
06.     }
07. };
```

std::greater<T> 的底层实现代码为：

```
01. template <typename T>
02. struct greater {
03.     //重载 > 运算符
04.     bool operator()(const T &lhs, const T &rhs) const {
05.         return _lhs > _rhs;
06.     }
07. };
```

由此可见，std::less<T> 和 std::greater<T> 底层实现的主要不同在于，前者使用 < 号实现从大到小排序，后者使用 > 号实现从小到大排序。

```
struct node
{
    int x, y;
    node(int x, int y) : x(x), y(y){}
};

struct cmp
{
    bool operator()(node a, node b)
    {
        if(a.x == b.x) return a.y > b.y;
        else return a.x > b.x;
    }
};
```

优先队列的这个类型，其实有三个参数：priority_queue<class Type, class Container, class Compare>，即类型，容器和比较器，后两个参数可以省略，这样默认的容器就是vector，比较方法是less，也就是默认大根堆，可以自定义比较方法，但此时若有比较方法参数，则容器参数不可省略。

priority_queue<> 的可支持的容器必须是用数组实现的容器，如vector，deque，但不能是list（推荐vector）

比较方法可以写结构体重载()运算符，也可以用less，greater这些语言实现了的，建议手写重载结构体

如果不想写比较结构体的话，就将后面的两个参数省略，直接重载类型的<>运算符

```
class Solution {
public:
    class cmp {
    public:
        bool operator()(pair<char, int> &m, pair<char, int> &n) {
            return m.second < n.second;
        }
    };
    priority_queue<pair<char, int>, vector<pair<char, int>>, cmp> pque; //最大堆
    for (const auto &elem : counts) {
        pque.emplace(elem.first, elem.second);
    }
}
```

LC-颜色分类

题目描述 评论 (1.1k) 提交 (1.4k) 提交记录 C++ * 题解模式

75. 颜色分类

难度 中等 866 收藏 分享

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

示例 1：

```
输入：nums = [2,0,2,1,1,0]
输出：[0,0,1,1,2,2]
```

```
1 class Solution {
2 public:
3     void sortColors(vector<int>& nums) {
4         int ptr = 0;
5         for(int i = 0; i < nums.size(); i++){
6             if(nums[i] == 0){
7                 swap(nums[i],nums[ptr]);
8                 ptr++;
9             }
10        }
11        for(int i= ptr; i < nums.size(); i++){
12            if(nums[i] == 1){
13                swap(nums[i],nums[ptr]);
14                ptr++;
15            }
16        }
17    }
}
```

我们可以考虑对数组进行两次遍历。在第一次遍历中，我们将数组中所有的0交换到数组的头部。

在第二次遍历中，我们将数组中所有的1交换到头部的0之后。此时，所有的2都出现在数组的尾部，这样我们就完成了排序。

我们使用一个指针ptr表示「头部」的范围，ptr中存储了一个整数，表示数组nums从位置0到位置ptr-1都属于「头部」。

在第二次遍历中，我们从「头部」开始，从左向右遍历整个数组，如果找到了1，那么就需要将1与「头部」位置的元素进行交换，并将「头部」向后扩充一个位置。

遍历结束，所有的2都被交换到0之后，此时2只出现在「头部」之外的位置，因此排序完成。

使用双指针一次遍历

如果遍历到1直接交换并递增p1指针，如果遍历到0，先交换元素，再判断当p0 < p1，已经将一些1连续地放在头部，此时再进行swap num[i],num[p0]一定会把一个1交换出去，因此如果p0 < p1，执行完需要再将nums[i]与nums[p1]再交换，保证交换出去的那个1交换到时num[p1]。每执行一次swap num[p0],num[i]，两个指针都需要加1。

75. 颜色分类

难度 中等 866 收藏 分享

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

示例 1：

```
输入：nums = [2,0,2,1,1,0]
输出：[0,0,1,1,2,2]
```

```
1 class Solution {
2 public:
3     void sortColors(vector<int>& nums) { // 202110
4         int p0 = 0, p1 = 0;
5         for(int i = 0; i < nums.size(); i++){
6             if(nums[i] == 1){
7                 swap(nums[i],nums[p1]);
8                 p1++;
9             }
10            else if(nums[i] == 0){
11                swap(nums[i],nums[p0]);
12                if(p0 < p1)
13                    swap(nums[i],nums[p1]);
14                p0++;
15                p1++;
16            }
17        }
}
```

第六章 搜索 树和图结构中常见的两种方法

深度优先搜索 (DFS)

搜索到一个新的结点时，立即对该结点进行遍历，遍历需要用先入后出的栈来实现，也可以通过与栈等价的递归实现。

对于树结构而言，由于总是对新结点调用遍历，就像是向着深的方向前进。



考虑一棵树，如果遍历顺序是从左子结点到右子结点，按照优先向深的方向前进策略，如果采用递归实现

遍历过程：1 (起始) ->2 (向前推进) ->4 (无子结点，返回到2) ->2 (返回到1) ->3 (推进) ->1 (无子结点，返回到1)

对已经搜索过的结点进行标记可以防止遍历时重复搜索某个结点，这种方法称为状态记录

一般深度搜索类型的题可以分为主函数和辅函数，主函数用来遍历所有搜索位置，判断是否可以开始搜索，如果可以则进行辅函数搜索

辅函数实现深度搜索的递归调用，也可以用栈实现（栈与递归调用原理相同）

辅函数中注意递归边界条件的判定，边界判定一般两种写法，

一是判断求越界时进行下一步搜索，即判断放在调用递归前，如果不满足条件，return初值，一个return语句

第二种是先进行下一步搜索，待下一步搜索开始时再判断越界，即判断放在辅助函数第一行，如果不满足条件，单独return一个值，两个return语句

LC-岛屿面积

695. 岛屿的最大面积

难度 中等 482 收藏 分享

给定一个包含了一些 0 和 1 的非空二维数组 grid。

一个岛屿是由一些相邻的 1（代表土地）构成的组合，这里的「相邻」要求两个 1 必须在水平或者垂直方向上相邻。你可以假设 grid 的四个边都被 0（代表水）包围着。

找到给定的二维数组中最大的岛屿面积。（如果没有岛屿，则返回面积为 0。）

示例 1：

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,1,1,0,1,0,0,0,0,0,0,0,0],
 [0,1,0,0,1,1,0,0,1,0,1,0,0],
 [0,1,0,0,1,1,0,0,1,1,1,0,0],
 [0,0,0,0,1,1,0,0,1,1,1,0,0],
 [0,0,0,0,0,0,0,0,0,1,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,0,0,0,0,0,0,1,1,0,0,0,0]]
```

第二种递归写法：反正都要将当前遍历到的每一个元素的值都变为0，再去看能遍历多少次

```
int dfs(vector<vector<int>> &grid, int r, int c){
    if(r < 0 || r >= grid.size() || c < 0 || c >= grid[0].size() || grid[r][c] == 0)
        return 0;
    grid[r][c] = 0;
    return 1 + dfs(grid,r-1,c) +dfs(grid,r+1,c) +dfs(grid,r,c-1) +dfs(grid,r,c+1);
}
```

LC-省份数量

题目描述

547. 省份数量

难度 中等

544 支 544 贡 0 贡

有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直接相连，且城市 b 与城市 c 直接相连，那么城市 a 与城市 c 间接相连。

省份是一组直接或间接相连的城市，组内不含其他没有相连的城市。

给你一个 $n \times n$ 的矩阵 isConnected ，其中 $\text{isConnected}[i][j] = 1$ 表示城市 i 与城市 j 之间直接相连，而 $\text{isConnected}[i][j] = 0$ 表示二者不直接相连。

返回矩阵中 省份 的数量。

示例 1：

```

1 class Solution {
2 public:
3     int findCircleNum(vector<vector<int>>& isConnected) {
4         vector<int> visited(isConnected.size()); // 初始化一个访问矩阵都为0，表示都没有访问到
5         int res = 0;
6         for(int i = 0; i < isConnected.size(); i++){
7             if(visited[i] == 0){
8                 res++;
9                 dfs(isConnected, visited, i); //从这个城市出发，找到所有相邻的城市
10            }
11        }
12        return res;
13    }
14    void dfs(vector<vector<int>> &isConnected, vector<int> &visited, int i){
15        for(int j = 0; j < isConnected.size(); j++){
16            if(isConnected[i][j] == 1 & visited[j] == 0){ //相连且未访问
17                visited[j] = 1;
18                dfs(isConnected, visited, j);
19            }
20        }
21    }
22}

```

输入：`isConnected = [[1,1,0],[1,1,0],[0,0,1]]`
输出：`2`

这是一道标准的使用深度优先解决的问题，首先定义一个访问数组，判断当前的城市是否已经访问过，然后每个未访问的城市逐个遍历，找到相连的城市就标记为访问过，并对该城市进行深度优先，知道所有的相连的城市都找到，求无向图中连通分量的问题

这个题和岛屿问题不同，岛屿问题是上下左右四个方向连通的问题，而这个问题引入了连接等价关系，如 $1-2, 2-3$ ，则 $1-3$ 也连通，即这是无向图的连接问题，这个矩阵就是图的邻接矩阵

如：
 $\begin{matrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{matrix}$

从第一个元素`visit[0]`出发，`res++`；执行`dfs`从`isConnected[0][0]`出发，可以将`visited[0] - visited[3]`全部变为 1，表示都与`[0][0]`这个城市连通，那么主函数中的`dfs`只会执行1次，`res=1`

如果还有不连通的，则会有`visited[i]==0`，此时`res`可以再进行 +1，那继续从这个城市开始再执行`dfs`，看它与哪些城市连通，主函数从0开始，表示这个城市还未访问，`res+1`，表示这个城市将

连通，继续再看到一个`vis==0`，`res+1`，表示又来了一个还未连通的城市，每看到一个`vis[0]`，`res+1`，表示这个城市是还未连通的，如果都连通完了，`vis`都变为1，就还会`vis[0]`了，这个`res`就是连通分量结果

LC-太平洋大西洋水流问题

417. 太平洋大西洋水流问题

难度 中等

232 支 232 贡 0 贡

给定一个 $n \times n$ 的非负整数矩阵来表示一片大陆上每个单元格的高度。“太平洋”位于大陆的左边界和上边界，“大西洋”位于大陆的右边界和下边界。

规定水流只能按照上、下、左、右四个方向流动，且只能从高到低或者在同一高度上流动。

请找出那些水流既可以流动到“太平洋”，又能流动到“大西洋”的陆地单元的坐标。

提示：

- 1. 输出坐标的顺序不重要。
- 2. m 和 n 都小于 150。

限制：

给定下面的 5×5 矩阵：

太平洋 ~ ~ ~ ~ ~
~ 1 2 2 3 (5) *
~ 3 2 3 (4) *
~ 2 4 (5) 3 1 *
~ (6) (7) 1 4 5 *
~ (5) 1 1 2 4 *
* * * * * 大西洋

返回：

`[[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]]` (上图中带括号的单元)。

```

void dfs(vector<vector<int>>& matrix, vector<vector<bool>>& visited, int pre, int i, int j) {
    if (i < 0 || i >= m || j < 0 || j >= n || visited[i][j] || matrix[i][j] < pre) {
        return;
    }

    visited[i][j] = true;
    for (int k = 0; k < 4; k++) {
        int newx = i + dis[k][0];
        int newy = j + dis[k][1];
        dfs(matrix, visited, matrix[i][j], newx, newy);
    }
}

```

辅助函数另一种递归写法

```

/*
vector<int> direction{-1, 0, 1, 0, -1};
void dfs(vector<vector<int>>& matrix, vector<vector<bool>>& canReach, int r, int c) {
    if (canReach[r][c]) return;
    canReach[r][c] = true;
    int x, y;
    for (int i(0); i < 4; i++) {
        x = r + direction[i], y = c + direction[i + 1];
        if (x >= 0 && x < matrix.size() && y >= 0 && y < matrix[0].size()
            && matrix[r][c] <= matrix[x][y]) {
            dfs(matrix, canReach, x, y);
        }
    }
}

```

多源点DFS逆向思考，从大西洋和太平洋分别往里面流动，都能流到的即为合法结果点（从其对应的两个矩形的边开始遍历，遍历完成只需遍历一下记录矩阵）

虽然题目要求的是满足向下流能到达两个大洋的位置，如果我们对所有的位置进行搜索，那么在不剪枝的情况下复杂度会很高。

因此我们可以反过来想，从两个大洋开始向上流，这样我们只需要对矩形四条边进行搜索。搜索完成后，只需遍历一遍矩阵，满足条件的位置即为两个大洋向上流都能到达的位置处

回溯法

回溯法常用于需要记录结点状态的深度优先搜索，发现目前的结点（及其子结点）不是

需求目标时，回退到原来的结点继续搜索，并将在目前结点修改的状态还原。

这样可以始终只对图的总状态修改，而非每次遍历都新建一个图存储状态

写法与普通深度搜索相同，只是多了回溯的步骤

修改当前结点状态->递归子结点 -->回改当前结点状态

1.按引用传状态 2.所有的状态修改在递归完成时回改

回溯修改一般有同种情况，一种是修改最后一位输出，比如排列组合；一种是修改访问标记，如矩阵里搜索字符串

经典回溯问题

LC-全排列

46. 全排列

难度 中等 1316 收藏 0 书签

给定一个没有重复数字的序列，返回其所有可能的全排列。

示例：

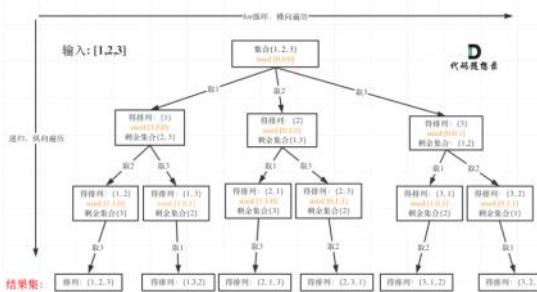
输入：[1,2,3]

输出：

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

```
1 class Solution {  
2 public:  
3     vector<vector<int>> res;  
4     vector<int> path;  
5     vector<vector<int>> permute(vector<int>& nums) {  
6         vector<bool> used(nums.size(), false);  
7         backtracking(nums, used);  
8         return res;  
9     }  
10    void backtracking(vector<int> &nums, vector<bool>& used){  
11        if(path.size() == nums.size()) {  
12            res.push_back(path);  
13            return; // 递归停止条件，path数组满了，表示得到了一个排列  
14        }  
15        for(int i = 0; i < nums.size(); i++) { // 以123为例  
16            if(used[i] == true) continue; // 如果这个数已经使用过  
17            used[i] = true; // 标记这个数已经使用过  
18            path.push_back(nums[i]);  
19            backtracking(nums, used); // 调用backtracking深度搜索，再找没有用过的数push_back到path，直到path满了存入res，return。  
20            path.pop_back(); // 将path末尾元素弹出，并将该元素标记为false 未使用  
21            used[i] = false;  
22        }  
}
```

我以[1,2,3]为例，抽象成树形结构如下：



题目描述 评估 (669) 凸显解 (906) 摆列记录 C++ 模拟面试

77. 组合 难度 中等 571 收藏 0 书签

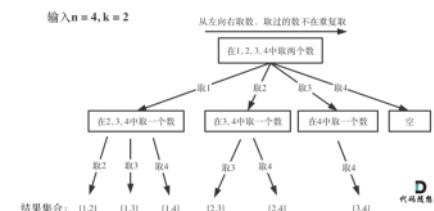
给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。

示例：

输入：n = 4, k = 2
输出：

```
[2,4],  
[3,4],  
[2,3],  
[1,2],  
[1,3],  
[1,4]
```

```
1 class Solution {  
2 public:  
3     vector<vector<int>> res; // 定义全局变量，存放最终结果，如果不用全局变量，则要体现在递归函数的参数中，影响可读性  
4     vector<int> path; // 全局变量，保存路径  
5     vector<vector<int>> combine(int n, int k) {  
6         backtracking(n, k, 1);  
7         return res;  
8     }  
9     void backtracking(int n, int k, int start){  
10        if(path.size() == k){ // 递归函数终止条件，当路径size == k，表示一条满足条件的路径，存入res。  
11            res.push_back(path);  
12            return;  
13        }  
14        // 每次从集合中选取元素，可选择的范围随着选择的进行而收缩，调整可选择的范围，就是设置start  
15        for(int i = start; i < n; i++){  
16            path.push_back(i); // 用for循环将当前值存入路径，并且用来进行遍历 将值存入path  
17            backtracking(n, k, i+1); // 每一次递归后 从i+1开始 取这些元素 进行进行深度搜索  
18            path.pop_back(); // backtracking 退出return，将取出path末尾的一个元素，以便前面元素添加到path --回溯，重置状态 还原  
19        }  
20    }  
21};
```



可以看出这个棵树，一开始集合是 1, 2, 3, 4，从左向右取数，取过的数，不在重复取。
第一次取1，集合变2, 3, 4，因为k为2，我们只需要再取一个数就可以了，分别取2, 3, 4，得到集合 [1,2][1,3][1,4]，以此类推。

组合问题是回溯法解决的经典问题，我们开始的时候给大家列举一个很形象的例子，就是n为100，k为50的话，直接想法就需要50层for循环。从而引出了回溯法就是解决这种层层for循环嵌套的问题。

然后进一步把回溯法的搜索过程抽象为树形结构，可以直观的看出搜索的过程。

接着用回溯法三部曲，逐步分析了函数参数、终止条件和单层搜索的过程

题目描述 评估 (917) 凸显解 (134) 摆列记录 C++ 模拟面试

79. 单词搜索 难度 中等 883 收藏 0 书签

给定一个 $n \times n$ 二维字符网格 board 和一个字符串单词 word，如果 word 存在于网格中，返回 true；否则，返回 false。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是指这些单元格相邻并共用相同的字母。同一个单元格内的字母不允许被重复使用。

示例 1：

A	B	C	E
S	F	C	S
A	D	E	E

输入：board = [[“A”, “B”, “C”, “E”], [“S”, “F”, “C”, “S”], [“A”, “D”, “E”, “E”]]
word = “ABCECEDDS”
输出：true

```
1 class Solution {  
2 public:  
3     int m, n;  
4     vector<vector<char>> direct = {{1, 0, 1, -1},  
5     bool exist(vector<vector<char>& board, string word) {  
6         m = board.size(); n = board[0].size();  
7         for(int i = 0; i < m; i++){  
8             for(int j = 0; j < n; j++){  
9                 if(dfs(board, word, i, j, 0))  
10                    return true; // 注意数组的起始位置，遍历整个网格，从网格中的每一个点开始搜索  
11            }  
12        }  
13        return false;  
14    }  
15    bool dfs(vector<vector<char>& board, string& word, int i, int j, int index){  
16        if(i < 0 || i > m - 1 || j < 0 || j > n - 1 || board[i][j] != word[index])  
17            return false;  
18        if(index == word.size() - 1)  
19            return true;  
20        if(board[i][j] == '\0')  
21            bool flag = dfs(board, word, i - 1, j, index + 1) || dfs(board, word, i + 1, j, index + 1)  
22            || dfs(board, word, i, j - 1, index + 1) || dfs(board, word, i, j + 1, index + 1);  
23            board[i][j] = word[index];  
24            return flag;  
25        }  
26    }  
27};  
28 }
```

两种递归方向的表示方法，一种是直接控制i与j的坐标，还有一种是定义一个全局方向，由这个全局方向修改i与j的坐标

```

bool dfs(vector<vector<char>> &board, string& word, int i, int j, int index){
    if(i<0 || i >m || j<0 || j>n || board[i][j] != word[index])
        return false;
    if(index == word.size()-1)
        return true;
    board[i][j] = '\0'; //当前board[i][j]已用，从这个点出发再进行dfs就不能用该点了；防止下一次又到该点将index+1
    // bool flag = dfs(board, word, i-1, j, index+1) || dfs(board, word, i+1,j,index+1)
    //           || dfs(board,word, i,j+1, index+1) ||dfs(board,word,i,j-1, index+1);
    bool flag = 0;
    for(int k = 0; k< 4; k++){ //定义一个全局方向vector<int> direc = {-1, 0, 1, 0, -1};
        int dx = direc[k], dy = j + direc[k+1];
        if(flag = dfs(board,word,dx,dy,index+1));
            if(flag) break;
    }
    board[i][j] = word[index];
    return flag;
}

```

如搜索AB：这个Index是从0开始的，单词下标也是从0开始，index+1，下标就是1，再执行dfs(...index+1),index=2,就会返回true,还原当前 board[i][j] = word[index] word[1]

这种是直接在原矩阵进行修改的方法，将访问过的元素直接赋值为'\0'（这种更优）

下面是单独开一个visited矩阵,将访问过的元素记录对应为true/1

```

class Solution {
public:
    int m, n;
    vector<vector<char>> board; //board:问题大小的visited矩阵作为记录
    vector<vector<bool>> visited; //board:问题大小的visited矩阵作为记录
    bool dfs(vector<vector<char>> &board, string word, int i, int j, int index){
        if(i<0 || i >m || j<0 || j>n || visited[i][j] || board[i][j] != word[index])
            return false;
        if(index == word.size()-1)
            return true;
        if(dfs(board, word,i, j , index+1));
            return true;
        visited[i][j] = true;
        board[i][j] = '\0';
        bool flag = dfs(board,word,i-1,j,index+1) || dfs(board,word,i+1,j,index+1)
                  || dfs(board,word,i,j+1, index+1) ||dfs(board,word,i,j-1, index+1);
        visited[i][j] = false;
        board[i][j] = word[index];
        return flag;
    }
}

```

深度搜索：栈的原理实现/递归实现 广度搜索：队列实现

广度优先搜索（宽度优先搜索 breadth-first search BFS）

一层一层遍历的，需要用先入先出的队列实现 而不是像DFS那样基于栈的原理实现

由于是按层次/宽度遍历，通常也可以用来处理最短路径问题



考虑一棵树，从1号结点开始，如果遍历顺序是从左子结点到右子结点，则按宽度方向前进

队列顶端元素变化过程为1->2->3->4

题目描述
评论 (113)
编辑 (180)
提交记录

C++
* 隐私模式

934. 最短的桥

难度 中等 151 收藏

在给定的二维二进制数组 A 中，存在两座岛屿。（岛屿是由四面相连的 1 形成的一个最大岛。）

现在，我们可以将 0 变为 1，以便两座岛屿连接起来，变成一座岛。

返回必须翻转的 0 的最小数目。（可以保证答案至少是 1。）

示例 1：

输入：A = [[0,1],[1,0]]
输出：1

示例 2：

输入：A = [[0,1,0],[0,0,0],[0,0,1]]
输出：2

示例 3：

输入：A = [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]
输出：1

提示：

- $2 \leq A.length == A[0].length \leq 100$
- $A[i][j] == 0$ 或 $A[i][j] == 1$

通过次数 15,820 提交次数 33,462

```

class Solution {
public:
    vector<int> direction{-1, 0, 1, 0, -1};

    int shortestBridge(vector<vector<int>> &A) {
        int m = A.size(), n = A[0].size();
        queue<pair<int, int>> points;
        //使用dfs寻找第一个岛屿，并把1变成2
        bool flipped = false;
        for (int i(0); i < m; ++i) {
            if (!flipped) break; //两次break表示退出整个for循环
            for (int j(0); j < n; ++j) {
                if (A[i][j] == 1) {
                    dfs(points, A, m, n, i, j); //执行第一次dfs，找到了第一个连通岛屿
                    flipped = true; //将变量flip=true，结合上一层的break，退出整个for循环
                    break;
                }
            }
        }
    }

    //bfs寻找第二个岛屿，把过程中的0变成2
    int x, y;
    int level = 0; //此时points中存放的是为 0 的元素 的坐标
    while (!points.empty()) {
        int n_points = points.size();
        ++level; // points是表示为 0 的点。从0 ->1最少需要翻转1次。
        while (n_points--) { //对一层0元素遍历看是否能找得到1的点，如果找到了，直接return level
            auto f = points.front(); //遍历是四个方向遍历着能否再找到第二个岛/值为1，从队首第一个0元素坐标开始遍历
            points.pop();
            for (int k(0); k < 4; ++k) {
                x = f.first + direction[k], y = f.second + direction[k + 1];
                if (x >= 0 && y >= 0 && x < m && y < n) {
                    if (A[x][y] == 2) continue; //如果值2，表示已经遍历过了，继续朝其他方向前进
                    if (A[x][y] == 1) return level; //如果找到1，则返回当前 level 层数
                    points.push({x, y}); //将遇到的0元素继续加入到points，作为下一层遍历的元素使用
                    A[x][y] = 2; //将再遇到的0元素继续变为2
                }
            }
        }
        //一次遍历结束了，还是没有找到一个1，则level++，需要翻转的次数加1
    }
    return 0;
}

void dfs(queue<pair<int, int>>& points, vector<vector<int>> &grid, int m, int n,
        int i, int j) {
    //dfs到达边界或者遇到为2的点/已经访问过了，则退出这个前进方向
    if (i < 0 || i < m || i == n || j == 0 || j == n || grid[i][j] == 2) return;
    //如果遇到为0的点，则也要退出这个前进方向，同时保存这个为0的点到队列，作为队列遍历的首层元素
    if (grid[i][j] == 0) {
        points.push({i, j});
        return;
    }
    //否则表示遇到的点都为1，将其变为2，从这个点出发四个方向继续进行dfs遍历
    //最终会将第一岛的所有1都变成了2，所有dfs函数执行完成退出
    grid[i][j] = 2; //将所有1都变为2
    dfs(points, grid, m, n, i + 1, j);
    dfs(points, grid, m, n, i - 1, j);
    dfs(points, grid, m, n, i, j + 1);
    dfs(points, grid, m, n, i, j - 1);
}
```

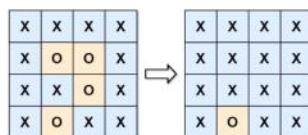
分区 follow谷歌老哥 的第 121 页

题目描述 | 提交记录 | 模拟面试

```
130. 被围困的区域
难度 中等 | 通过 400 | 提交 900 | 模拟面试
```

给定一个 $n \times n$ 的矩阵 board，由若干字符 'X' 和 'O'，找到所有被 'X' 围困的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例 1：



输入：board = [[“X”, “X”, “X”, “X”], [“X”, “O”, “O”, “X”], [“X”, “X”, “O”, “X”], [“X”, “O”, “X”, “X”]]
输出：[[“X”, “X”, “X”, “X”], [“X”, “X”, “X”, “X”], [“X”, “X”, “X”, “X”], [“X”, “O”, “X”, “X”]]
解释：被包围的区域不会存在边界上，换句话说，任何边界上的 ‘O’ 都不会被转化为 ‘X’。任何不在边界上，或不与边界上的 ‘O’ 相连的 ‘O’ 最终都会被填充为 ‘X’。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

LC-二叉树的所有路径

257. 二叉树的所有路径
 难度 简单 | 通过 406 | 提交 104 | 模拟面试

给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明：叶子节点是指没有子节点的节点。

示例：

输入：
1
/\
2 3
\
5
输出：[“1->2->5”, “1->3”]
解释：所有根节点到叶子节点的路径为：1->2->5, 1->3

通过次数: 110,721 | 提交次数: 164,907

询问您在高亮招聘中遇到此题？

杜振 | 校招 | 实习 | 未遇到

例子：

```
class Solution {
private:
    void traversal(TreeNode* cur, string path, vector<string>& result) {
        if (cur == NULL) {
            path += to_string(cur->val); // 中
            if (cur->left == NULL && cur->right == NULL) {
                result.push_back(path);
                return;
            }
            if (cur->left) traversal(cur->left, path + "->", result); // 左
            if (cur->right) traversal(cur->right, path + "->", result); // 右
        }
    }
};

vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> result;
    string path;
    if (root == NULL) return result;
    traversal(root, path, result);
    return result;
}
```

如上代码精简了不少，也隐藏了不少东西

注意在函数定义的时候void traversal(TreeNode* cur, string path, vector<string>& result), 定义的是string path, 每次都是复制赋值, 不用使用引用, 否则就无法做到回溯的效果。

那么在如上代码中,貌似没有看到回溯的逻辑, 其实不然, 回溯就隐藏在traversal(cur->left, path + "->", result);中的path + "->". 每次函数调用完, path依然是没有加上"->"的, 这就是回溯。

如果这里还不理解的话, 可以看这篇二叉树: 以为使用了递归, 其实还隐藏着回溯, 我在这篇中详细的解释了递归中如何隐藏着回溯。

综上以上, 第二种递归的代码虽然精简但把很多重要的点隐藏在了代码细节里, 第一种递归写法虽然代码多一些, 但是把每一个逻辑处理都完整的展现了出来了。

传引用的代码

在主函数 定义了一个路径变量path , 传的引用, 这个path只有一个, 作为形参传入修改会改变整个path的内容,

因此需要回溯操作, 每执行一次traversal , path中会多一个val, 需要将这个val弹出 ,

这样traversal执行结束返回, 就会执行下面的这条pop_back()语句, 弹出前面(上一层函数) push_back到path的那个元素

```
void traversal(TreeNode* cur, vector<int>& path, vector<string>& result) {
    path.push_back(cur->val);
    // 这才到了叶子节点
    if (cur->left == NULL && cur->right == NULL) {
        string sPath;
        for (int i = 0; i < path.size() - 1; i++) {
            sPath += to_string(path[i]);
            sPath += "->";
        }
        sPath += to_string(path[path.size() - 1]);
        result.push_back(sPath);
        return;
    }
    if (cur->left) {
        traversal(cur->left, path, result);
        path.pop_back(); // 回溯
    }
    if (cur->right) {
        traversal(cur->right, path, result);
        path.pop_back(); // 回溯
    }
}

public:
vector<string> binaryTreePaths(TreeNode* root) {
    vector<int> result;
    vector<string> path;
    if (root == NULL) return result;
    traversal(root, path, result);
    return result;
}
```

LC-组合问题

题目描述

评论 (33)

题解 (1.1K)

提交记录

47. 全排列 II

难度 中等

△ 684 ★ 10% □ 0% ▢

给定一个包含重复数字的序列 `nums`，按任意顺序 返回所有不重复的全排列。

示例 1：

输入: `nums = [1,1,2]`
输出:
[[1,1,2],
[1,2,1],
[2,1,1]]

示例 2：

输入: `nums = [1,2,3]`
输出: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

提示：

- $1 \leq n \leq 8$
- $-10 \leq \text{nums}[i] \leq 10$

通过次数 16,800 提交次数 256,172

题目描述

评论 (1.1K)

题解 (1.5K)

提交记录

48. 组合Ⅰ和Ⅱ

难度 简单

△ 1322 ★ 10% □ 6% ▢

给定一个无重复元素的数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中所有可以组成数字和为 `target` 的组合。

`candidates` 中的数字可以无限重复选取。

说明：

- 所有数字（包括目标数）都是正整数。
- 结果不能包含重复的组合。

示例 1：

输入: `candidates = [2,3,6,7], target = 7,`
所求解集为:
[
[7],
[2,2,3]
]

示例 2：

输入: `candidates = [2,3,5], target = 8,`
所求解集为:
[
[2,2,2,2],
[2,3,3]
]

题目描述

评论 (74)

题解 (1.0K)

提交记录

49. 组合Ⅰ和Ⅱ

难度 中等

△ 565 ★ 10% □ 5% ▢

给定一个数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中所有可以组成数字和为 `target` 的组合。

`candidates` 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 结果不能包含重复的组合。

示例 1：

输入: `candidates = [10,1,2,7,6,1,5], target = 8,`
所求解集为:
[
[1, 7],
[1, 2, 5],
[2, 6],
[1, 1, 6]
]

示例 2：

输入: `candidates = [2,5,1,2], target = 5,`
所求解集为:
[
]

C++

```
1 class Solution {
2 private:
3     vector<vector<int>> result;
4     vector<int> path;
5     void backtracking(vector<int>& candidates, int target, int sum, int startIndex, vector<bool>& used) {
6         if (sum > target) return; // 剪枝条件
7         if (sum == target) {
8             result.push_back(path);
9         }
10        for (int i = startIndex; i < candidates.size() && sum + candidates[i] <= target; i++) {
11            if (sum + candidates[i] > target) continue; // 剪枝条件
12            if (used[i]) continue; // 剪枝条件
13            sum += candidates[i];
14            path.push_back(candidates[i]);
15            backtracking(candidates, target, sum, i+1); // 不同i+1, 因为此时可以重复
16            sum -= candidates[i]; // 回溯
17            path.pop_back();
18        }
19    }
20 public:
21     vector<vector<int>> combinationSum1(vector<int>& candidates, int target) {
22         sort(candidates.begin(), candidates.end()); // 排序
23         backtracking(candidates, target, 0, 0);
24         return result;
25     }
26 }
```

题目描述

评论 (74)

题解 (1.0K)

提交记录

49. 组合Ⅰ和Ⅱ

难度 中等

△ 565 ★ 10% □ 5% ▢

给定一个数组 `candidates` 和一个目标数 `target`，找出 `candidates` 中所有可以组成数字和为 `target` 的组合。

`candidates` 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 结果不能包含重复的组合。

示例 1：

输入: `candidates = [10,1,2,7,6,1,5], target = 8,`
所求解集为:
[
[1, 7],
[1, 2, 5],
[2, 6],
[1, 1, 6]
]

示例 2：

输入: `candidates = [2,5,1,2], target = 5,`
所求解集为:
[
]

C++

```
1 class Solution {
2 private:
3     vector<vector<int>> result;
4     vector<int> path;
5     void backtracking(vector<int>& candidates, int target, int sum, int startIndex, vector<bool>& used) {
6         if (sum > target) return; // 剪枝条件
7         if (sum == target) {
8             result.push_back(path);
9         }
10        for (int i = startIndex; i < candidates.size() && sum + candidates[i] <= target; i++) {
11            if (sum + candidates[i] > target) continue; // 剪枝条件
12            if (used[i] == true) continue; // 剪枝条件
13            sum += candidates[i];
14            path.push_back(candidates[i]);
15            used[i] = true;
16            backtracking(candidates, target, sum, i+1, used); // 和yy.组合Ⅰ和Ⅱ的区别1. 这里是i+1, 每个数字在每个组合中只能使用一次
17            sum -= candidates[i];
18            path.pop_back();
19            used[i] = false;
20        }
21    }
22 public:
23     vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
24         vector<bool> used(candidates.size(), false); // used标记元素使用, 由于每个元素只能使用一次
25         path.clear();
26         result.clear();
27         sort(candidates.begin(), candidates.end()); // 最先对candidates排序, 让相同的元素放在一起
28         backtracking(candidates, target, 0, 0, used); // 从startIndex=0, sum=0, 开始遍历整个candidate vector
29         return result;
30     }
31 }
```

第七章 动态规划

本质：保存子问题的解，避免重复计算

关键：找到状态转移方程，这样就可以计算和存储子问题的解以求解最终的问题

同时也对动态规划进行空间压缩，以节省空间消耗

有些情况下，动态规划可以看成是带有状态记录的优先搜索，动态规划是自下而上的，先解决

子问题再解决上层问题，带有状态记录的优先搜索是自上而下的，从上层问题搜索到子问题，

若重复搜索到同一个子问题则进行状态记录，防止重复计算，即带状态记录的优先搜索

基本动态规划 一维

题目描述

评论 (2.2K)

题解 (0.7K)

提交记录

70. 爬楼梯

难度 简单

△ 1628 ★ 10% □ 0% ▢

假设你正在爬楼梯。需要 n 阶你才能到达楼梯。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼梯顶呢？

注意：给定 n 是一个正整数。

示例 1：

输入: 2
输出: 2
解释: 有两条方法可以爬到楼梯。
1. 1 阶 + 1 阶
2. 2 阶

定义一个数组 dp ， $dp[i]$ 表示走到第 i 阶的方法数。

每次可以走一步或者两步，所以第 i 阶可以从第 $i-1$ 或 $i-2$ 阶到达。

换句话说，走到第 i 阶的方法数即为走到第 $i-1$ 阶的方法数加上走到第 $i-2$ 阶的方法数。这样我们就得到了状态转移方程 $dp[i] = dp[i-1] + dp[i-2]$ 。

C++

```
1 class Solution {
2 public:
3     int climbStairs(int n) {
4         vector<int> dp;
5         //for(int i = 1; i < n; i++) //你真是个懒癌，这是一个开的数组，下标从1开始???? dp[0]没有得到初始化，那就是没有定义的！！！
6         for(int i = 1; i <= n; i++){
7             if(i==0)
8                 dp.push_back(0);
9             else if(i==1)
10                 dp.push_back(1);
11             else if(i==2) //如果这两个else不加，那这个if就会与下面else组成一个语句，前面两个if可以执行，但是下面那个else必执行
12                 dp.push_back(2);
13             else //只有两个else if 下面那个else会与上面所有组成一个语句，只有当i>2 else语句执行，这样就不会出现越界
14                 dp.push_back(dp[i-1] + dp[i-2]);
15         }
16         return dp[n];
17     }
18 }
```

题目描述

评论 (1.3k)

题解 (2.0k)

提交记录

C++

* 简洁模式

```

1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         int n = nums.size();
5         if(n==0) return 0;
6         if(n==1) return nums[0];
7         vector<int> dp(n,0);
8         dp[0] = nums[0];
9         dp[1] = max(nums[0],nums[1]);
10        for(int i = 2; i < n; i++) {
11            dp[i] = max(dp[i-2]+nums[i], dp[i-1]);
12        }
13    }
14 }
```

示例 1：

输入：[1,2,3,1]

输出：4

解释：偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。

偷窃到的最高金额 = 1 + 3 = 4。

测试用例 代码执行结果 阅试器

定义一个数组 dp , $dp[i]$ 表示抢劫到第 i 个房子时，可以抢劫的最大数量（用开的这个数组记录这些计算过程）考虑 $dp[i]$ ，此时可以抢劫的最大数量有两种，一种是我们选择不抢劫这个房子，此时累计的金额即为 $dp[i-1]$ ；另一种是我们选择抢劫这个房子，那么此前累计的最大金额只能是 $dp[i-2]$ ，因为我们不能够抢劫第 $i-1$ 个房子，否则会触发警报机关。因此本题的状态转移方程为 $dp[i] = \max(dp[i-1], nums[i-1] + dp[i-2])$

LC-等差数列

题目描述

评论 (0.0k)

题解 (31)

提交记录

C++

* 简洁模式

413. 等差数列划分

难度 中等

234

☆

□

△

▲

●

□

如果一个数列至少有三个元素，并且任意两个相邻元素之差相同，则该数列为等差数列。

例如，以下数列是等差数列：

1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9

以下数列不是等差数列。

1, 1, 2, 5, 7

1 //多加一个整数元素差数列的一个问题： 常常多了一个数，实际上一个数可以与前n-1个数形成不同等差数列 $dp[i]=dp[i-1]+1$ ，多一个整数元素的 $dp[i]$ 就有 $dp[i-1]$ 不同于不同的等差数列个数

示例 1：

1	3	1
1	5	1
4	2	1

输入：`grid = [[1,3,1],[1,5,1],[4,2,1]]`输出：`7`解释：因为路径 $1 \rightarrow 3 \rightarrow 1 \rightarrow 5 \rightarrow 1$ 的总和最小。

LC-01矩阵

题目描述

评论 (297)

题解 (489)

提交记录

C++

* 简洁模式

542. 01矩阵

难度 中等

414

☆

□

△

▲

●

□

给定一个由 0 和 1 组成的矩阵，找出每个元素到最近的 0 的距离。

说明：两个相邻元素间的距离为 1。

示例 1：

输入：
[[0,0,0],
 [0,1,0],
 [0,0,0]]输出：
[[0,0,0],
 [0,1,0],
 [0,0,0]]

示例 2：

输入：
[[0,0,0],
 [0,1,0],
 [1,1,1]]输出：
[[0,0,0],
 [0,1,0],
 [1,2,1]]

```

1 class Solution {/ / 542.01矩阵
2 public:
3     // 定义  $dp[i][j]$  表示该位置距离 0 最短的距离，由于动态规划保存之前状态的特性，我们更新状态的时候，要么从左上到右下，要么从右下到左上
4     // 从左上->右下遍历一次，更新左上的最近距离，再从右下->左上遍历一次，更新右下的最近距离
5     // 其实很容易想到  $dp[i][j]$  要么等于 0，要么等于  $\min(dp[i-1][j], dp[i][i-1], dp[i][j+1], dp[i+1][j+1]) + 1$ 
6     // 这个问题要解决的就是，当我们更新状态的时候，要么从左上到右下，要么从右下到左上，无论哪种更新方式都只能依赖两个前状态（比如从左上到右下时， $dp[i][j]$  只能依赖  $dp[i-1][j]$  和  $dp[i][j-1]$ ）。这里做两遍  $dp$  状态的更新来解决上述问题
7     vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
8         int m = mat.size(), n = mat[0].size();
9         vector<vector<int>> dp(m, vector<int>(n, INT_MAX-1)); // 开一个很大的矩阵，存储每个位置到 0 的最近距离
10        // 初始化每次向右或者向下一步，即从左上方开始遍历，当元素距离 0 的最近距离为 左边元素 or 上方元素距离 0 的距离取 min + 1
11        for(int i = 0; i < m; i++) { // 同时要与自己本身比较，因此分别将  $dp[i][j]$  本身 与 左边元素/上方元素的距离 + 1 取 min
12            for(int j = 0; j < n; j++) {
13                if(i == 0 || j == 0)  $dp[i][j] = 0$ ; // 第一次遍历， $dp[i][j]$  可以表示本身（包括了来自于左上方的距离），及它下方元素 or 右边元素距离 0 的距离 + 1
14                else {
15                    if(j > 0)  $dp[i][j] = \min(dp[i][j], dp[i][j-1] + 1)$ ;
16                    if(i > 0)  $dp[i][j] = \min(dp[i][j], dp[i-1][j] + 1)$ ;
17                }
18            }
19        }
20        // 下一次遍历， $dp[i][j]$  可以表示本身（包括了来自于左上方的距离），及它下方元素 or 右边元素距离 0 的距离 + 1
21        for(int i = m-1; i >= 0; i--) {
22            for(int j = n-1; j >= 0; j--) {
23                if(i < m-1)  $dp[i][j] = \min(dp[i][j], dp[i+1][j] + 1)$ ;
24                if(j < n-1)  $dp[i][j] = \min(dp[i][j], dp[i][j+1] + 1)$ ;
25            }
26        }
27    }
28 }
```

题目描述 评论 (563) 提交记录 C++ 模拟面试

221. 最大正方形
难度 中等 756 收藏 761 热门 0 举报
在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

示例 1：

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

```

1 class Solution {
2 public: //用 dp(i,j) 表示以 (i,j) 为右下角，且只包含 1 的正方形的边长最大值
3     int maximalSquare(vector<vector<char>>& matrix) {
4         int n = matrix.size(), m = matrix[0].size();
5         if(m==0 || n==0) return 0;
6         vector<vector<int>> dp(n, vector<int>(m, 0));
7         int maxside = 0;
8         for(int i = 0; i < n; i++){
9             for(int j = 0; j < m; j++){
10                 if(matrix[i][j]== '1'){
11                     if(i==0 || j==0)
12                         dp[i][j] = 1;
13                     else
14                         dp[i][j] = min(min(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1])+1;
15                 }
16             maxside = max(maxside, dp[i][j]);
17         }
18     }
19     return maxside * maxside;
20 }

```

那么如何计算 dp 中的每个元素值呢？对于每个位置 (i, j) ，检查在矩阵中该位置的值：

- 如果该位置的值是 0，则 $dp(i, j) = 0$ ，因为当前位置不可能在由 1 组成的正方形中；
- 如果该位置的值是 1，则 $dp(i, j)$ 的值由其上方、左方和左上方的三个相邻位置的 dp 值决定。具体而言，当前位置的元素值等于三个相邻位置的元素中的最小值加 1，状态转移方程如下：

$$dp(i, j) = \min(dp(i-1, j), dp(i-1, j-1), dp(i, j-1)) + 1$$

此外，还需要考虑边界条件。如果 i 和 j 中至少有一个为 0，则以位置 (i, j) 为右下角的最大正方形的边长只能是 1，因此 $dp(i, j)=1$ 。

下图也给出了计算 dp 值的过程。

原始矩阵

0	1	2	3	4
0	0	1	1	1
1	1	1	1	0
2	0	1	1	1
3	0	1	1	1
4	0	0	1	1

dp

0	1	2	3	4
0	0	1	1	0
1	1	1	2	2
2	0	1	2	3
3	0	1	2	3
4	0	0	1	2

表示：
dp[2][3] 表示 $dp(2, 3) = \min(dp(1, 3), dp(1, 2), dp(2, 2)) + 1 = 3$
dp[3][4] 表示 $dp(3, 4) = \min(dp(2, 4), dp(2, 3), dp(3, 3)) + 1 = 2$
dp[4][2] 表示 $dp(4, 2) = \min(dp(3, 2), dp(3, 1), dp(4, 1)) + 1 = 1$

题目描述 评论 (577) 提交记录 C++ 模拟面试

279. 完全平方数
难度 中等 853 收藏 769 举报
给定正整数 n ，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n 。你需要让组成和的完全平方数的个数最少。
给你一个整数 n ，返回和为 n 的完全平方数的 最少数量。
完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等一个整数自身的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1：

```

输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4

```

$dp[i]$ ：和为 i 的完全平方数的最少数量为 $dp[i]$ 。
 $dp[i]$ 可以由 $dp[i-j*j]$ 推出， $dp[i-j*j] + 1$ 便可以凑成 $dp[i]$ 即拆成一个数 == $dp[i - \text{一个数平方}] + 1$ 。
 $dp[10] = dp[10 - 3*3] + 1 = dp[1] + 1$

从递归公式 $dp[i] = \min(dp[i-j*j] + 1, dp[i])$ 中可以看出每次 $dp[i]$ 都要选最小的，所以非 0 下标的 $dp[i]$ 一定要初始化为最大值 INT_MAX，这样 $dp[i]$ 在递推的时候才不会被初始值覆盖。

位置 i 只依赖 $i-k^2$ 的位置，如 $i-1, i-4, i-9$ 等等，才能满足完全平方分割的条件。因此 $dp[i]$ 可以取的最小值即为 $1 + \min(dp[i-1], dp[i-4], dp[i-9], \dots)$

题目描述 评论 (116) 提交记录 C++ 模拟面试

91. 解码方法
难度 中等 834 收藏 130 举报
一条包含字母 A-Z 的消息通过以下映射进行了编码：

```

'A' -> 1
'B' -> 2
...
'Z' -> 26

```

解码 已编码的消息，所有数字必须基于上述映射的方法，反向映射时字母（可能有多条方法）。例如，“1110” 可以映射为：

- “AAAF”，将消息分成组为 (1 1 10 6)
- “KJF”，将消息分成组为 (11 18 6)

注意，消息不能分成组为 (1 11 06)，因为 “06” 不能映射为 “F”，这是由于 “c” 和 “g” 在映射中并不等价。

给你一个只含数字的非空字符串 s，请计算并返回 解码 方法的总数。

题目数据保证字符串 s 至多是一个 32 位的整数。

示例 1：

```

输入: s = "11"
输出: 2
解释: 可以解码为 "AB" (1 1) 或者 "L" (12)。

```

对于给定的字符串 s ，设它的长度为 n ，其中的字符从左到右依次为 $s[1], s[2], \dots, s[n]$ 。我们可以使用动态规划的方法计算出字符串 s 的解码方法数。
 $dp[i]$ 表示计算到 $s[1 \dots i]$ 字符的解码个数，开 size+1 的 dp ， $dp[i]$ 表示计算到 $s[1 \dots i-1]$ 的解码个数，因为 $dp[0]$ 表示 $s[0]$ 空串，认为空串的解码方法为 1 种，即 $dp[0]=1$ 。
 $\text{vector<int>} dp(n+1, 0) \rightarrow dp[i]$ 表示计算到 $s[0 \dots i]$ ， $dp[1]$ 表示计算到 $s[1]$ 。

• 第一种情况是我们使用了一个字符，即 $s[i]$ 进行解码，那么只要 $s[i] \neq 0$ ，它就可以被解码成 A ~ I 中的某个字母。由于剩余的前 $i-1$ 个字符的解码方法数为 f_{i-1} ，因此我们可以写出状态转移方程：

$$f_i = f_{i-1}, \text{ 其中 } s[i] \neq 0$$

• 第二种情况是我们使用了两个字符，即 $s[i-1]$ 和 $s[i]$ 进行编码，与第一种情况类似， $s[i-1]$ 不能等于 0，并且 $s[i-1]$ 和 $s[i]$ 组成的整数必须小于等于 26，这样它们就可以被解码成 J ~ Z 中的某个字母。

由于剩余的前 $i-2$ 个字符的解码方法数为 f_{i-2} ，因此我们可以写出状态转移方程：

$$f_i = f_{i-2}, \text{ 其中 } s[i-1] \neq 0 \text{ 并且 } 10 \cdot s[i-1] + s[i] \leq 26$$

需要注意的是，只有当 $i > 1$ 时才能进行转移，否则 $s[i-1]$ 不存在。

将上面的两种状态转移方程在对应的条件满足时进行累加，即可得到 f_i 的值。在动态规划完成后，最终的答案即为 f_n 。

LC-单词拆分 leetcode

难度 中等
点赞 959 反 166 为 0 回

描述 一个字符串 s 和一个包含若干单词的列表 wordDict，判断 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

示例 2：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
```

注意你可以重复使用字典中的单词。

3. dp数组如何初始化

从递归公式中可以看出，dp[i] 的状态依靠 dp[i] 是否为 true，那么 dp[0] 就是递归的根基，dp[0]一定要为 true，否则递归下去后面都是 false。

那么 dp[0] 有没有意义呢？

dp[0] 表示如果字符串为空的话，说明出现在字典里。

但题目中说了“给定一个非空字符串 s”所以测试数据中不会出现为 0 的情况，那么 dp[0] 初始为 true 完全就是为了推导公式。

下标非 0 的 dp[i] 初始化为 false，只要没有被覆盖说明都是不可拆分为一个或多个在字典中出现的单词。

以输入: s = "leetcode", wordDict = ["leet", "code"] 为例，dp 状态图：

输入 "leetcode"									
["leet", "code"]									
下标:	0	1	2	3	4	5	6	7	8
dp[i]:	1	0	0	0	1	0	0	0	1

D
公众号：代码随想录

dp[s.size()] 就是最终结果。
+8

300. 最长递增子序列

难度 中等
点赞 1502 反 166 为 0 回

给你一个整数数组 nums，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

示例 1：

```
输入: nums = [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长递增子序列是 [2,3,7,101]，因此长度为 4。
```

示例 2：

```
输入: nums = [0,1,0,3,2]
输出: 4
```

dp[i] 的定义：dp[i] 表示 i 之前（包括 i）的最长上升子序列

状态转移方程：如果当前位置大于前面的 dp[j] 位置的最长上升子序列等于 0-i-1 的最长上升子序列长度 + 1

if(nums[i] > nums[j]) dp[i] = max(dp[i], dp[j]+1); 这种写法在循环里表示 dp[i] 取 dp[j]+1 的最大值

dp 的初始化，初始为 1 即可，因为最长上升序列的最小长度为 1

确定遍历顺序：外层从 0 开始到结尾，内层从 0 到 i-1 (< i)

举例：dp 数组的推导

输入：[0,1,0,3,2]

下标 i: 0 1 2 3 4

i=1	1	2	1	1	1
i=2	1	2	1	1	1
i=3	1	2	1	3	1
i=4	1	2	1	3	3

D

1143. 最长公共子序列

难度 中等 537 收藏 例程 讨论

给定两个字符串 $text_1$ 和 $text_2$ ，返回这两个字符串的最长公共子序列的长度。如果不存在公共子序列，返回 0。

一个字符串的子序列 是指由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）而组成的新字符串。

- 例如，“ace”是“abcde”的子序列，但“ace”不是“abcde”的子序列。

两个字符串的公共子序列 是这两个字符串所共有的子序列。

```

1 class Solution {
2 public:
3     int longestCommonSubsequence(string text1, string text2) {
4         int m = text1.size(), n = text2.size();
5         vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
6         for(int i = 1; i <= m; i++) {
7             for(int j = 1; j <= n; j++) {
8                 if(text1[i-1] == text2[j-1])
9                     dp[i][j] = dp[i-1][j-1]+1;
10                else
11                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
12            }
13        }
14        return dp[m][n];
15    }
16 }
```

示例 1：

输入: $text_1 = "abcd", text_2 = "acba"$
输出: 3
解释: 最长公共子序列是 "ace"，它的长度为 3。

最长公共子序列问题是典型的二维动态规划问题。

假设字符串 $text_1$ 和 $text_2$ 的长度分别为 m 和 n ，创建 $m+1$ 行 $n+1$ 列的二维数组 dp ，其中 $dp[i][j]$ 表示 $text_1[0:i]$ 和 $text_2[0:j]$ 的最长公共子序列的长度。

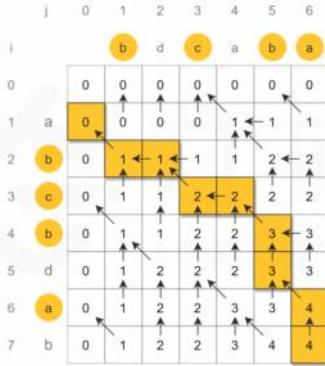
上述表示中， $text_1[0:i]$ 表示 $text_1$ 的长度为 i 的前缀， $text_2[0:j]$ 表示 $text_2$ 的长度为 j 的前缀。

考虑动态规划的边界情况：

- 当 $i = 0$ 时， $text_1[0:i]$ 为空，空字符串和任何字符串的最长公共子序列的长度都是 0，因此对任意 $0 \leq j \leq n$ ，有 $dp[0][j] = 0$ ；
- 当 $j = 0$ 时， $text_2[0:j]$ 为空，同理可得，对任意 $0 \leq i \leq m$ ，有 $dp[i][0] = 0$ 。

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & text_1[i-1] == text_2[j-1] \\ \max(dp[i-1][j], dp[i][j-1]), & text_1[i-1] \neq text_2[j-1] \end{cases}$$

最终计算得到 $dp[m][n]$ 即为 $text_1$ 和 $text_2$ 的最长公共子序列的长度。



在矩阵中只有相邻的两行会有用，之前的数据其实可以丢弃掉了，所以空间复杂的可以减少到 $O(\min(m, n))$ ，但是代码没有看懂

```

string res = "";
for(int i = s1.size(); i >= 0; ) {
    if(s1[i-1] == s2[j-1]) {
        res += s1[i-1];
        i--;
        j--;
    } else if(dp[i-1][j] >= dp[i][j-1]) i--;
    else j--;
}
reverse(res.begin(), res.end());
if(res.empty())
    cout << "-1";
else return res;

```

背包问题

背包问题是一种组合优化的 NP 完全问题：有 N 个物品和容量为 W 的背包，每个物品都有自己的体积 w 和价值 v ，求拿哪些物品可以使得背包装下物品的总价值最大。如果限定每种物品只能选择 0 个或 1 个，则问题称为 0-1 背包问题；如果不限定每种物品的数量，则问题称为无界背包问题或完全背包问题。

我们可以用动态规划来解决背包问题。以 0-1 背包问题为例，我们可以定义一个二维数组 dp 存储最大价值，

$dp[i][j]$ 表示前 i 件物品体积不超过 j 的情况下能达到的最大价值

在我们遍历到第 i 件物品时，在当前背包总容量为 j 的情况下，如果我们不将物品 i 放入背包，那么 $dp[i][j] = dp[i-1][j]$ ，即前 i 个物品的最大价值等于只取前 $i-1$ 个物品时的最大价值；如果我们将物品 i 放入背包，假设第 i 件物品体积为 w ，价值为 v ，那么我们得到 $dp[i][j] = dp[i-1][j-w] + v$ 。我们只需在遍历过程中对这两种情况取最大值即可，总时间复杂度和空间复杂度都为 $O(NW)$

```

int knapsack(vector<vector<int>> weights, vector<vector<int>> values, int N, int W) {
    vector<vector<int>> dp(N+1, vector<int>(W+1, 0));
    for (int i = 1; i <= N; ++i) {
        int w = weights[i-1], v = values[i-1];
        for (int j = 1; j <= W; ++j) {
            if (j >= w) {
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w] + v);
            } else {
                dp[i][j] = dp[i-1][j];
            }
        }
    }
    return dp[N][W];
}

```

$dp[i][j]$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
$i = 0$					
$i = 1$					
$i = 2$					
$i = 3$					
$i = 4$					

图 7.2: 0-1 背包问题 - 状态转移矩阵样例

在完全背包问题中，一个物品可以拿多次。假设我们遍历到物品 $i = 2$ ，且其体积为 $w = 2$ ，价值为 $v = 3$ ；对于背包容量 $j = 5$ ，最多只能装下 2 个该物品。

那么我们的状态转移方程就变成了 $dp[2][5] = \max(dp[1][5], dp[1][3] + 3, dp[1][1] + 6)$ 。

如果采用这种方法，假设背包容量无穷大而物体的体积无穷小，我们这里的比较次数也会趋于无穷大，怎么解决这个问题呢？

我们发现在 $dp[2][3]$ 的时候我们其实已经考虑了 $dp[1][3]$ 和 $dp[2][1]$ 的情况，而在时 $dp[2][1]$ 也已经考虑了 $dp[1][1]$ 的情况。因此，

对于拿多个物品的情况，我们只需考虑 $dp[2][3]$ 即可。

即 $dp[2][5] = \max(dp[1][5], dp[2][3] + 3)$ 。

这样就得到了完全背包问题的状态转移方程： $dp[i][j] = \max(dp[i-1][j], dp[i][j-w] + v)$ ，其与 0-1 背包问题的差别仅仅是把状态转移方程中的第二个 $i-1$ 变成了 i 。

```

int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; ++i) {
        int v = weights[i - 1], w = values[i - 1];
        for (int j = 1; j <= W; ++j) {
            if (j >= v) {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - w] + v);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[N][W];
}

```

数组子集分割问题/n数之和问题

LC-分割等和子集

```

416. 分割等和子集
难度: 中等
分类: 动态规划
状态: 待解决(通过) 通过率: 100% ①提交记录

说明: 只包含正整数的非空数组 nums，请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
示例 1:
输入: nums = [1,5,11,5]
输出: true
解释: 数组可以分割成 [1, 5, 5] 和 [11] 。
示例 2:
输入: nums = [1,2,3,5]
输出: false
解释: 数组不能分割成两个等和的子集。
提示:
1 <= nums.length <= 200
-10^5 <= nums[i] <= 100

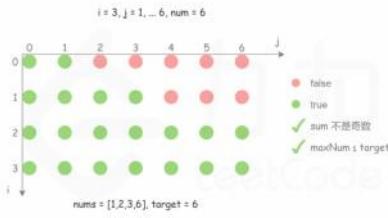
```

15 115 有4个物品 重量分别 为这么多 作为N+1行 列是重量
背包容量为0 ...target=11 作为target+1列 行是容量
dp[n][target]取到第n件物品 能否将容量为target的背包装满 即正好装到容量为target 看那个值的状态是否为true 可达

状态转移方程如下：

$$dp[i][j] = \begin{cases} dp[i-1][j] & dp[i-1][j - num[i]] \\ dp[i-1][j], & j < num[i] \end{cases}, \quad j \geq num[i]$$

最终得到 $dp[n-1][target]$ 即为答案。



同样的，我们也可以对本题进行空间压缩。注意对数字的遍历需要逆向（未看）

```

474. 一和零
难度: 中等
分类: 动态规划
状态: 待解决(通过) 通过率: 100% ①提交记录

说明: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
请你找出返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。
如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。
示例 1:
输入: strs = ["00", "0001", "110001", "1", "0"], m = 5,
n = 3
输出: 4
解释: 最多有 5 个 0 和 3 个 1 的最大子集是
{"00", "0001", "1", "0"}，因此答案是 4 。
其他满足要求的较小的子集包括 {"0001", "1"} 和
{"1", "0", "1", "0"}，{"110001"} 不满足要求，因为它含 4 个 1 ，
大于 n 的值 3 。
示例 2:
输入: strs = ["10", "0", "1"], m = 1, n = 1
输出: 2
解释: 最大的子集是 {"0", "1"}，所以答案是 2 。

```

二维写法

```

class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0)); // 默认初始化
        for (string str : strs) { // 遍历物品
            int oneNum = 0, zeroNum = 0;
            for (char c : str) {
                if (c == '0') zeroNum++;
                else oneNum++;
            }
            for (int i = m; i >= zeroNum; i--) { // 遍历背包容量且从后向前遍历！
                for (int j = n; j >= oneNum; j--) {
                    dp[i][j] = max(dp[i][j], dp[i - zeroNum][j - oneNum] + 1);
                }
            }
        }
        return dp[m][n];
    }
};

```

压缩方法

```

// 在动态规划中，如果第k个状态只与第k-1个状态有关，而不与其他的例如第i-k(0 < k < i)个状态有关，那么意味着此时
// 在空间上有优化的空间，我们可以采用滚动数组或者从后往前的方式直接来代替开辟更高维度的数组。
// 滚动数组可以理解，但另一种方式是从前往后的式填表，这是为什么呢？
// 我们可以举个例子，假设一个状态方程为 dp[i-1][j-1] = dp[i-1][j-1] + 1;
// 如果采用从后往前填表，那么我们的dp[i-1][j-1]应该是上一轮计算的结果，因为这一轮我们还没有更新过这个值
// 但如果采用从前往后填表，那么我们的dp[i-1][j-1]应该是这一轮计算的结果，因为这一轮我们已经更新过这个值
// 但是我们这二维dp数组是最初的三维dp数组的一个优化，因此，在状态迁移时，我们需要的是上一轮计算的dp[i-1][j-1]
// 这就是为什么我们要从前往后填表了，主要是保留上一轮计算的结果不被覆盖。

```

01背包 先从3维退化为2维dp

```
int findMaxSum(vector<string>& strs, int m, int n)
{
    int num = strs.size();
    int w0, w1;
    vector<vector<vector<int>> dp(num+1, vector<vector<int>(m+1, vector<int>(n+1, 0)));
    for(int k=1;k<=num;k++)
    {
        w0 = 0; w1 = 0;
        for(char &c:strs[k-1])
        {
            if(c=='0') w0 += 1;
            else w1 += 1;
        }
        for(int i=0;i<=m;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(i >= w0 && j >= w1)
                    dp[k][i][j] = max(dp[k-1][i][j], dp[k-1][i-w0][j-w1]+1);
                else
                    dp[k][i][j] = dp[k-1][i][j];
            }
        }
    }
    return dp[num][m][n];
}
```

二维Dp

```
vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
for(int i = 1; i <= num; i++)
{
    w0 = 0; w1 = 0;
    // 计算第i个字符串的两个重量
    for(char &c:strs[i-1])
        if(c=='0') w0 += 1; else w1 += 1;

    // 01背包，逆向状态更新dp，设每个物品的值为1
    for(int j = 0; j <= m; j++)
        for(int k = 0; k >= w1; k--)
            dp[j][k] = max(dp[j][k], 1+dp[j-w0][k-w1]);
}
return dp[m][n];
```

如果正向迭代的话会导致之前的状态被当前的状态覆盖，而我们需要的又恰好是之前的状态，所以在01背包问题中，要压缩状态必须是逆向的，而完全背包内部就不需要逆向。建议你可以手画一个最简单01背包的状态转移矩阵，然后再画一个二维的状态转移矩阵，你就会懂了，我之前压缩状态也卡了挺久，光看代码很难理解，手画状态转移矩阵就懂了。来自菜鸟的小建议。

分治法

巧解数学问题

[题目描述](#) [评论 \(676\)](#) [题解 \(632\)](#) [提交记录](#)

204.计数质数

难度 简单 674 收藏 7% 贡献

统计所有小于非负整数 n 的质数的数量。

示例 1：

输入： $n = 10$
输出：4
解释：小于 10 的质数一共有 4 个，它们是 2, 3, 5, 7。

示例 2：

输入： $n = 0$
输出：0

```
C++ * 草稿模式
1 class Solution {
2 public:
3     int isPurnum(int x){
4         for(int i=2; i*i <= x; i++){
            //在x的平方根内的范围 看有没有x的因素 如 9 就可以在[0,1]内看有没有3的因素
            if(x % i == 0)
                return 0;
        }
        return 1;
    }
    int countPrimes(int n) {
        // int res;// res是一个累加变量 你又不初以为0 你真是个 xx ? ?
        int res = 0;
        for(int i = 2; i < n; i++){
            res += isPurnum(i);
        }
        return res;
    }
};
```

埃拉托斯特尼筛法 (Sieve of Eratosthenes，简称埃氏筛法) 是非常常用的，判断一个整数是否是质数的方法。

其原理也十分易懂：从 1 到 n 遍历，假设当前遍历到 m ，则把所有小于 n 、且是 m 的倍数的整数标为和数；遍历完成后，没有被标为和数的数字即为质数

[题目描述](#) [评论 \(676\)](#) [题解 \(632\)](#) [提交记录](#)

204.计数质数

难度 简单 674 收藏 7% 贡献

统计所有小于非负整数 n 的质数的数量。

示例 1：

输入： $n = 10$
输出：4
解释：小于 10 的质数一共有 4 个，它们是 2, 3, 5, 7。

示例 2：

输入： $n = 0$
输出：0

```
C++ * 草稿模式
1 class Solution {
2 public:
3     int countPrimes(int n){
4         if(n < 2) return 0;
5         vector<bool> pre(n, true);
6         int count = 0; //1与n不是质数
7         for(int i = 2; i < n; ++i){
8             if(pre[i]){
9                 count += 1;
10                for(int j = 2*i; j < n; j+= i){
11                    if(pre[j]){
12                        pre[j] = false;
13                    }
14                }
15            }
16        }
17        return count;
18    }
};
```

[题目描述](#) [评论 \(239\)](#) [题解 \(254\)](#) [提交记录](#)

504.七进制数

难度 简单 83 收藏 7% 贡献

给定一个整数，将其转化为7进制，并以字符串形式输出。

示例 1：

输入：100
输出：“202”

示例 2：

输入：-7
输出：“-10”

注意：输入范围是 $[-1e7, 1e7]$ 。

```
C++ * 草稿模式
1 class Solution {
2 public:
3     string convertToBase7(int num) {
4         //if(num == 0) return "0"; //char类型还是 string类型！！！要用“表示string类型 ‘ ’ 表示char类型
5         if(num == 0) return "0";
6         bool is_nag = num<0;
7         if(is_nag) num = -num;
8         string res;
9         while(num){
10             int b = num % 7;
11             // res += to_string(b); res = res + to_string(b) 这样错了 12 :b: 0 0 1 1 ---> 1100
12             res = to_string(b) + res;
13             num /= 7;
14         }
15         return is_nag ? "-"+res : res;
16     }
};
```

题目描述 评估(400) 提交记录

172. 阶乘后的零

难度 简单 打赏 456 收藏 467

给定一个整数 n , 返回 $n!$ 结果尾数中零的数量。

示例 1:

输入: 3
输出: 0
解释: $3! = 6$, 尾数中没有零。

示例 2:

输入: 5
输出: 1
解释: $5! = 120$, 尾数中有 1 个零。

说明: 你算法的时间复杂度应为 $O(\log n)$ 。

```
C++ * 高能模式
1 class Solution {
2 public:
3     //首先末尾有多少个 0 ,只需要给当前数字以一个 10 就可以加一个 0
4     //当我们发现结果会有一个 0 ,原因就是 2 和 5 相乘构成一个 10 ,而对于 10 的话 ,其实也只有 2 * 5 可以构成 ,所以我们只需要找有多少对 2/5
5     int trailingZeroes(int n) {
6         int count = 0; //n= 25 1*5 ... *2*5 ... * 3*5...4*5 ...5*5 出现6个5 25/5=5 5/5=1 6个
7         while(n){ //n=50 .....5*5 6*5 7*5 8*5 9*5 2*5*5 12*5 50/5=10 10/5=2 12个
8             count += n/5; //规律就是除到0为止 累加个数 就是5因子的个数 !!!
9             n /= 5; //n不为0 还要继续相除 继续进行累加
10        }
11    return count;
12 }
13 } //125 /5=25 25/5=5 5/5=1 规律就是除到0为止 ,累加的count为因子5的个数 !
14 }//每隔 25 个数字 ,出现的是两个 5 ,所以除了每隔 5 个数算作一个 5 ,每隔 25 个数 ,还需要多算一个 5
15 //同理我们还会发现每隔 5 * 5 * 5 = 125 个数字 ,会出现 3 个 5 每隔125个数又要再多出现一个5 多出5+1=6个5 125/5 =25+6 5 符合前面规律
```

题目描述 评估(415) 提交记录

168. Excel表列名称

难度 简单 打赏 339 收藏 463

给定一个正整数，返回它在 Excel 表中相对应的列名称。

例如，

1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
...

示例 1:

输入: 1
输出: "A"

示例 2:

输入: 28
输出: "AB"

需要注意1 - A而不是0 - A
所以每次转化前需要 : n -= 1

```
C++ * 高能模式
1 class Solution {
2 public:
3     string convertToTitle(int columnNumber) {
4         string res = "";
5         while(columnNumber){
6             columnNumber -= 1;
7             int a = columnNumber%26;
8             res.push_back('A' + a);
9             columnNumber /= 26;
10        }
11        reverse(res.begin(),res.end());
12        return res;
13    }
14 }
15 //1-26 A-Z 1 ---- n-1 = 0 ; 1表示 A, A+0 1--- A;
16 //27-52 AA-AZ
17 //53-78 BA-BZ 53:n-1, 52%26 = 0 A; 52/26 = 2 n-1, B; 53 :BA 从1开始表示 A 而不是0表示A
```

```
class Solution {
public:
    string convertToTitle(int n) {
        stack<char> S;
        while(!S.empty())
        {
            S.pop();
        }
        string s = "";
        while(n)
        {
            n -= 1;
            int a = n % 26;
            S.push('A' + a);
            n = n / 26;
        }
        while(!S.empty())
        {
            s += S.top();
            S.pop();
        }
        return s;
    }
};
```

不采用string的reverse函数，采用栈先进后出的顺序，先将元素压入栈，再将栈顶元素依次添加到字符串res
如栈中元素为A， B -----> 字符串元素为BA

位运算

位运算利用二进制位运算的特性进行一些算法的优化和巧妙计算

\wedge 按位异或

$\&$ 按位与

$|$ 按位或

\sim 按位取反

$<<$ 左移位

$>>$ 右移位

461. 汉明距离

难度 简单 打赏 400 收藏 467

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。
给出两个整数 x 和 y ,计算它们之间的汉明距离。

注意：
 $0 \leq x, y < 2^{31}$

示例:

输入: $x = 1$, $y = 4$

输出: 2

解释:
1 (0 0 0 1)
4 (0 1 0 0)
↑ ↑

上面的箭头指出了对应二进制位不同的位置。

```
1 class Solution {
2 public:
3     int hammingDistance(int x, int y) {
4         int diff = x ^ y;//对两个数进行按位异或操作 ,得到异或结果 也是一个二进制数
5         int res = 0;
6         while(diff){
7             res += diff & 1; //对异或结果统计 1的个数
8             diff >>=1;
9         }
10        return res;
11    }
12};
```

题目描述 评论 (1.1k) 题解 (1.7k) 提交记录

136. 只出现一次的数字

难度 简单 1839 收藏 分享

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

示例 1：

输入： [2,2,1]
输出： 1

示例 2：

输入： [4,1,2,1,2]
输出： 4

题目描述 评论 (365) 题解 (351) 提交记录

342. 4 的幂

难度 简单 181 收藏 分享

给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 true；否则，返回 false。

整数 n 是 4 的幂次方需满足：存在整数 x 使得 $n = 4^x$

示例 1：

输入：n = 16
输出：true

题目描述 评论 (132) 题解 (112) 提交记录

318. 最大单词长度乘积

难度 中等 164 收藏 分享

给定一个字符串数组 words，找到 length(words[i]) * length(words[j]) 的最大值，并且这两个单词不会有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。

示例 1：

输入：["abccw", "baz", "fao", "bar", "xtfn", "abcdef"]
输出：16
解释：这两个单词为 "abccw", "xtfn"。

示例 2：

输入：["a", "ab", "abc", "d", "cd", "abcd"]
输出：4
解释：这两个单词为 "ab", "cd"。

示例 3：

输入：["a", "aa", "aaa", "aaaa"]
输出：0
解释：不存在这样的两个单词。

数据结构

c++ STL

1.Sequence containers : 维持顺序的容器

vector 动态数组 也可以当作stack来使用 O(1)随机读取

list 双向链表 可以当作stack queue 使用 不支持随机读取

deque 双端队列 O(1)随机读取，O(1) 首尾增删

array 固定大小的数组 不常用

forward_list 单向链表 不常用

2.Container Adaptors 基于其它容器实现的数据结构

stack 后入先出LIFO结构 基于deque实现 stack常用于深度搜索 字符串匹配 单调栈问题

queue 先入先出FIFO结构 基于deque实现 deque常用于广度搜索

priority_queue 最大值先出结构 基于vector实现堆结构 它可以在O(nlogn)的时间排序数组

O(logn)的时间插入删除任意值 O(1)的时间获得最大值 priority_queue常用于维护数据结构并快速获取最大/最小值

3.Associative Containers : 实现了排好序的数据结构

set 有序集合 元素不可重复 底层实现红黑树，即一种特殊的二叉查找树BST，它可以在O(nlogn)的时间排序数组，插入删除元素，查找元素，获取最大/最小值

set与priority_queue都可以用于维护数据结构并快速获取最大/最小值

priority_queue默认不支持删除任意值，而set获取最大/最小值的时间复杂度略高

multiset 支持重复元素的set

map 有序映射或有序表，在set的基础上加上映射关系，可以对每一个元素key存一个value

multimap 支持重复元素的map

4.Unordered Associative Containers : 对于每个Associative Containers 实现了哈希版本

unordered_set 哈希集合 可以在O(1)的时间快速插入 删除 查找元素 常用于快速查询一个元素是否在这个容器内

unordered_multiset 支持重复元素的unordered_set

unordered_map 哈希映射或哈希表 在unordered_set基础上加上映射关系，可以对每一个元素key存一个value值

如果key的范围已知且比较小，也可以用vector代替unordered_map 用位置表示key，每个位置的值表示value

unordered_multimap 支持重复元素的unordered_map

题目描述 评论 (709) 题解 (886) 提交记录

448. 找到所有数组中消失的数字

难度 简单 720 收藏 分享

给定一个范围 $1 \leq a[i] \leq n$ ($n = \text{数组大小}$) 的整型数组，数组中的元素一些出现了两次，另一些只出现一次。

找到所有在 $[1, n]$ 范围内没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为 $O(n)$ 的情况下完成这个任务吗？你可以将多余的数组存储在额外的空间内。

示例：

输入：
[4,3,2,7,8,2,3,1]

输出：
[5,6]

48. 旋转图像

描述 中等 通过 868 提交记录

给定一个 $n \times n$ 的二维矩阵 $matrix$ 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 **原地** 翻转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来翻转图像。

示例 1：

1	2	3
4	5	6
7	8	9

7	4	1
8	5	2
9	6	3

```

1 class Solution {
2 public:
3     //先逆置矩阵 再按行翻转每一行
4     void rotate(vector<vector<int>>& matrix) {
5         for(int i = 0; i < matrix.size(); ++i){
6             for(int j = 0; j < i; ++j){} //这里下标是到j<i 如 (1,0)与(0,1)swap, (2,0)与(0,2),(2,1)与(1,2)swap
7                 swap(matrix[i][j],matrix[j][i]); //矩阵转置的代码
8         }
9     }
10    for(int i = 0; i < matrix.size(); ++i)
11        reverse(matrix[i].begin(), matrix[i].end()); //如果是下标可以直接修改到原matrix
12    //for(auto &t : matrix) //如果是用auto的for循环在这里必须加&才可以修改到matrix 修改原数组要用引用
13    // reverse(t.begin(), t.end());
14 }

```

输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[[7,4,1],[8,5,2],[9,6,3]]

49. 在一个高宽的图中来搜索 $m \times n$ 矩阵 $matrix$ 中的一个目标值 $target$ 。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例 1：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```

1 class Solution {
2 public:
3     bool searchMatrix(vector<vector<int>>& matrix, int target) {
4         int m = matrix.size();
5         if(m == 0) return false;
6         int n = matrix[0].size();
7         int j = n - 1, i = 0; //定义遍历变量的初值 从左上角开始查找, 如果当前m[i][j]大于target左移一位 小于target则下移一位
8         while(i < m && j >= 0){ //如果遍历退出还未找到, 则说明找不到target
9             if(matrix[i][j] == target) //如果找到了, 返回true
10                 return true;
11             else if(matrix[i][j] > target)
12                 --j;
13             else if(matrix[i][j] < target)
14                 ++i;
15         }
16         return false; //循环退出了, 没有找到
17     }
18 }

```

769. 最多能完排列序的块

难度 中等 通过 130 提交记录

数组 arr 是 $[0, 1, \dots, arr.length - 1]$ 的一种排列。我们将这个数组分割成“几个块”，并使这些块分别进行排序，之后再连接起来，使得连接的结果和升序序列后的原因数组相同。

我们最多能将数组分成多少块？

示例 1：

输入：arr = [4,3,2,1,0]
输出：1
解释：
将数组分成2块或者更多块，都无法得到所需的结果。
例如，分成 [4, 3], [2, 1, 0] 的结果是 [3, 4, 0, 1, 2]，这不是有序的数组。

示例 2：

输入：arr = [1,0,2,3,4]
输出：4
解释：
我们可以把它分成两块，例如 [1, 0], [2, 3, 4]，
然而，分成 [1, 0], [2], [3], [4] 可以得到最多的块数。

232. 使用队列队列

难度 中等 通过 401 提交记录

请你仅使用两个线性队列实现先入先出队列。队列应当支持一般队列的所有操作（`push`, `pop`, `peek`, `empty`）：

实现 MyQueue 类：

- `void push(int x)` 将元素 x 插入队列的末尾。
- `int pop()` 从队列的开头移除并返回元素。
- `int peek()` 返回队列开头的元素。
- `boolean empty()` 如果队列为空，返回 `true`；否则，返回 `false`

说明：

- 你只能使用标准的队列操作——也就是只有 `push` to top, `pop` from top, `size`, 和 `empty` 操作是合法的。
- 你所使用的语言不允许使用队列。你可以使用 `list` 或者 `deque` (双端队列) 来模拟一个栈，只要是标准的操作都可以。

进阶：

- 你能否将每个操作均摊时间复杂度为 $O(1)$ 的队列？换句话说，执行 n 个操作的总时间复杂度为 $O(n)$ 。但是其中一个操作可能花费较长时间。

```

int peek() {
    if(stk2.empty()){
        while(!stk1.empty()){
            int tmp = stk1.top();
            stk1.pop();// 
            stk2.push(tmp);
        }
    }
    if(stk2.empty())
        return 0;
    return stk2.top();
}

/** Returns whether the queue is empty. */
bool empty() {
    if(stk1.empty() && stk2.empty())
        return true;
    return false;
    //return (stk1.empty() && stk2.empty());//简单写法
}

```

题目描述 | 评论 (921) | 题解 (1.3k) | 提交记录

155. 最小栈

难度 简单 | 打赏 893 | 收藏 | 反馈 | 举报

设计一个支持 `push` , `pop` , `top` 操作，并能在常数时间内检索到最小元素的栈。

- `push(x)` —— 将元素 `x` 推入栈中。
- `pop()` —— 跳出栈顶的元素。
- `top()` —— 获得栈顶元素。
- `getMin()` —— 检索栈中的最小元素。

示例:

输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[[],[-2],[0],[-3],[],[],[],[]]]

输出：
[null,null,null,null,-3,null,0,-3]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); --> 返回 -3.

模拟面试 | i | P | o | x

C++ * 普通模式

```
1 class MinStack {
2 public: //从外部建立一个新的栈，栈顶表示原栈里所有值的最小值。每当在原栈里插入一个数字
3     //若该数字小于等于新栈栈顶，则表示这个数字在原栈里是最大值，我们将同时插入新栈内
4     /* initialize your data structure here. */
5     stack<int> s, min_stk;
6     MinStack() {
7     }
8 }
9 void push(int val) {
10     s.push(val);
11     if(min_stk.empty() || min_stk.top() >= val)//如果min栈顶元素>=要加入的这个元素，则将val加入min栈
12     | min_stk.push(val); //向原栈push元素的同时，min栈始终将保持栈顶元素是最小的元素
13 }
14
15 void pop() {
16     if(!min_stk.empty() && min_stk.top() == s.top())
17     | min_stk.pop();
18     s.pop();
19 }
20
21 int top() {
22     return s.top();
23 }
24
25 int getMin() {
26     return min_stk.top();
27 }
28 };
29 //一个写起来更简单但是时间复杂度略高的方法是，我们每次插入原栈时，都向新栈插入一次
30 //原栈里所有值的最小值（新栈栈顶和待插入值中较小的一个）；每次从原栈里取出数字时，同样
31 //取出新栈的栈顶。这样可以避免判断，但是每次都要插入和取出。
```

LC-每日温度

题目描述 | 评论 (790) | 题解 (1.2k) | 提交记录

739. 每日温度

难度 中等 | 打赏 745 | 收藏 | 反馈 | 举报

请根据每日气温列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

例如，给定一个列表 `temperatures = [73, 74, 75, 71, 69, 72, 76, 73]`，你的输出应该是 `[1, 1, 4, 2, 1, 1, 0, 0]`。

提示：气温列表长度的范围是 `[1, 30000]`。每个气温的值均为华氏度，都是在 `[30, 100]` 范围内的整数。

通过次数 160,100 | 提交次数 239,363

请问您在精英招聘中遇到此题？

模拟面试 | i | P | o | x

C++ * 普通模式

```
1 class Solution {
2 public:
3     vector<int> dailyTemperatures(vector<int>& T) {
4         vector<int> ans(T.size(),0); //单向递减的栈，遍历到的元素如果小于栈顶元素则入栈，栈内元素自底向上是递减的
5         stack<int> simp;//维护一个栈，存放日期，从左到右遍历温度数组，如果当前温度小于栈顶温度，入栈当前日期，如果大于栈顶，计算栈顶日期
6         for(int i = 0; i < T.size(); ++i){ //对应的值 ans[i] = i - indice
7             while(i == simp.empty()){
8                 int indice = simp.top();
9                 if(T[i] <= t[indice])
10                     break;
11                 simp.pop();
12                 ans[indice] = i - indice;
13             }
14             simp.push(i); //如果到循环末尾时，还有t[i] < t[indice]，还有更小的元素，则这些indice对应的ans[indice]为0
15         }
16         return ans; //经过更改的ans作为输出结果
17     }
18 }
```

LC-合并k个升序链表

题目描述 | 评论 (13k) | 题解 (1.6k) | 提交记录

23. 合并K个升序链表

难度 困难 | 打赏 1292 | 收藏 | 反馈 | 举报

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

输入: lists = [[1,4,5],[1,3,4],[2,6]]

输出: [1,1,2,3,4,4,5,6]

解释：链表数组如下：

```
[1->4->5,
1->3->4,
2->6]
```

将它们合并到一个有序链表中得到。

1->1->2->3->4->5->6

示例 2：

输入: lists = []

输出: []

模拟面试

C++ * 普通模式

```
7     * ListNode* : val(x), next(nullptr) {}
8     * ListNode(x, Listnode* next) : val(x), next(next) {}
9     */
10
11 class Solution {
12 public:
13     struct cmp{
14         bool operator()(ListNode* l1, ListNode* &l2){
15             return l1->val > l2->val; //默认认为越大的根堆， 逆序变成小根堆
16         }
17     }; //结构体定义先加： 优先队列的比较定义是一个结构体函数
18     ListNode* mergeKLists(vector<ListNode*>& lists) {
19         if(lists.empty()) return nullptr;
20         priority_queue<ListNode*>, vector<ListNode*>, cmp> heap;
21         for(ListNode* node : lists){ //将每个链表的头放入堆中
22             if(node)
23                 heap.push(node); //push 操作要判断非空 否则容易报运行时错误
24         }
25         ListNode* dummy = new ListNode(0); //虚拟结点用于构造链表
26         ListNode* cur = dummy;
27         while(!heap.empty()){
28             ListNode* node = heap.top(); //每次取出小根堆的堆顶元素
29             cur->next = node;
30             cur = cur->next;
31             heap.pop();
32             if(node->next) //该链表的下一个非空节点，放入堆中
33                 heap.push(node->next);
34         }
35         return dummy->next; //返回构造的链表结果
36     }
37 }
```

`ListNode* dummy= new ListNode(0);
ListNode* cur = dummy;`

`while(1){`

```
    bool flag = true;
    for(auto &i : a){
        if(i == nullptr) continue;
        cur->next = i;
        cur = cur->next;
        i = i->next;
        flag = false;
    }
    if(flag) break;
}
```

}

直接暴力的方法：

//1、使用数组排序，将所有节点放到一个数组中，整体排序，再重新串连，时间复杂度O(NlogN)，空间复杂度O(N)

`Listnode* mergeKLists(vector<Listnode*>& lists) {`

```
if (lists.empty())
{
    return nullptr;
}

vector<Listnode*> arr;
for (int i = 0; i < lists.size(); ++i)
{
    Listnode* head = lists.at(i);
    while (head)
    {
        arr.push_back(head);
        head = head->next;
    }
}
```

取得链表的第*i*个节点，用的是`list.at()`，不是用下标访问！！！

力扣 学习 题库 讨论 竞赛 求职 商店

题目描述 评论 (719) 点解 (1.1k) 提交记录

239.滑动窗口最大值

难度 困难 976 ★ □ ★ □

给你一个整数数组 $nums$ ，有一个大小为 k 的滑动窗口从数组的最左侧移动到右侧的最右侧。你只能看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例 1：

```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置      最大值
-----  -----
[1 3 -1] -3 5 3 6 7   3
1 [3 -1 -3] 5 3 6 7   3
1 3 [-1 -3 5] 3 6 7   5
1 3 -1 [-3 5 3] 6 7   5
1 3 -1 -3 [5 3 6] 7   6
1 3 -1 -3 5 [3 6 7]   7
```

题目描述 评论 (8.2k) 点解 (12.3k) 提交记录

1.两数之和

难度 简单 11024 ★ □ ★ □

给定一个整数数组 $nums$ 和一个整数目标值 $target$ ，请你在该数组中找出和为目标值的那 $两个$ 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

示例 1：

```
输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9，返回 [0, 1]。
```

题目描述 评论 (218) 点解 (2.4k) 提交记录

128.最长连续序列

难度 困难 759 ★ □ ★ □

给定一个未排序的整数数组 $nums$ ，找出数字连续的最长序列(不要求序列元素在原数组中连续)的长度。

讲解：你可以设计并实现时间复杂度为 $O(n)$ 的解决方案吗？

示例 1：

```
输入：nums = [100,4,200,1,3,2]
输出：4
解释：最长连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

示例 2：

```
输入：nums = [0,3,7,2,5,8,4,6,0,1]
输出：9
```

题目描述 评论 (218) 点解 (2.4k) 提交记录

149.直线上最多的点

难度 简单 2128 ★ □ ★ □

给定一个二维平面，平面上有 n 个点，求最多有多少点在同一条直线上上。

示例 1：

```
输入：[[1,1],[2,2],[3,3]]
输出：3
解释：
|
| o
| o
+---->
0 1 2 3 4
```

示例 2：

```
输入：[[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
输出：4
解释：
+
| o
| o
+---->
0 1 2 3 4
```

题目描述 评论 (662) 点解 (704) 提交记录

303.区域和检索-数积不可变

难度 简单 336 ★ □ ★ □

给定一个整数数组 $nums$ ，求出数组从索引 i 到 j ($i \leq j$) 范围内元素的总和，包含 i 、 j 两点。

实现 `NumArray` 类：

- `NumArray(int[] nums)` 使用数组 $nums$ 初始化对象。
- `int sumRange(int i, int j)` 返回数组 $nums$ 从索引 i 到 j ($i \leq j$) 范围内元素的总和，包含 i 、 j 两点(也就是 $sum(nums[i], nums[i + 1], \dots, nums[j])$)。

示例：

```
[NumArray, "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
输出：
[null, 1, -1, -3]
```

304. 二维区域和检索 - 矩阵不可变

难度 中等 266 收藏 为本章

给定一个二维矩阵，计算其矩形范围内元素的总和。该子矩阵的左上角为 $(row1, col1)$ ，右下角为 $(row2, col2)$ 。

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

上面矩阵左上角 $(row1, col1) = (2, 0)$ ，右下角 $(row2, col2) = (4, 3)$ ，该子矩阵内元素的总和为 11。

示例：

```
给定 matrix = [
[3, 0, 1, 4, 2],
[5, 6, 3, 2, 1],
[1, 2, 0, 1, 5],
[4, 1, 0, 1, 7],
[1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
```

题目描述 评论(411) 跟贴(660) 提交记录

560. 和为K的子数组

难度 中等 891 收藏 为本章

给定一个整数数组和一个整数 k ，你需要找到该数组中和为 k 的连续的子数组的个数。

示例 1：

输入: nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

说明：

1. 数组的长度为 $[1, 20,000]$ 。
2. 数组中元素的范围是 $[-1000, 1000]$ ，且整数 k 的范围是 $[-1e7, 1e7]$ 。

前缀和+哈希表

题目描述 评论(411) 跟贴(660) 提交记录

560. 和为K的子数组

难度 中等 891 收藏 为本章

给定一个整数数组和一个整数 k ，你需要找到该数组中和为 k 的连续的子数组的个数。

示例 1：

输入: nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

说明：

1. 数组的长度为 $[1, 20,000]$ 。
2. 数组中元素的范围是 $[-1000, 1000]$ ，且整数 k 的范围是 $[-1e7, 1e7]$ 。

通过次数 107,670 提交次数 241,957

请问您在哪些招聘中遇到此题？

杜招 校招 实习 未遇到

2 3 2 3 -5 6 ; k=6 prei-prej=6 11-5=6
prei-k = prej 5

第十二章 字符串

题目描述 评论(904) 跟贴(1,3k) 提交记录

242. 有效的字母异位词

难度 简单 379 收藏 为本章

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1：

输入: s = "anagram", t = "nagaram"
输出: true

示例 2：

输入: s = "rat", t = "car"
输出: false

题目描述 评论(408) 跟贴(493) 提交记录

```
1 class NumMatrix {
2 public:
3     //构造一行计算一维前缀和，初始化时对矩阵的每一行计算前缀和，检索时对二维区域中的每一行计算子数组和
4     NumMatrix(vector<vector<int>> sum);
5     int m = matrix[0].size();
6     int n = matrix[0].size();
7     //sum.resize(m,n); 一维数组这样初始化的？？？一定要写出先赋值再 resize(m, vector<int>(n));
8     sum.resize(m,n,vector<int>(n));
9     for(int i = 0; i < m; i++){
10         for(int j = 0; j < n; j++){
11             sum[i][j+1] = sum[i][j] + matrix[i][j];
12         }
13     }
14 }
15
16 int sumRegion(int row1, int col1, int row2, int col2) {
17     int res = 0;
18     for(int i = row1; i <= row2; i++){
19         res += (sum[i][col2 + 1] - sum[i][col1]);
20     }
21     return res;
22 }
23 }
24 */
25 //Your NumMatrix object will be instantiated and called as such:
26 //NumMatrix* obj = new NumMatrix(matrix);
27 // * int param_1 = obj->sumRegion(row1,col1,row2,col2);
28 // * int param_1 =
```

题目描述 评论(411) 跟贴(660) 提交记录

560. 和为K的子数组

难度 中等 891 收藏 为本章

给定一个整数数组和一个整数 k ，你需要找到该数组中和为 k 的连续的子数组的个数。

示例 1：

输入:nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

说明：

1. 数组的长度为 $[1, 20,000]$ 。
2. 数组中元素的范围是 $[-1000, 1000]$ ，且整数 k 的范围是 $[-1e7, 1e7]$ 。

题目描述 评论(411) 跟贴(660) 提交记录

560. 和为K的子数组

难度 中等 891 收藏 为本章

给定一个整数数组和一个整数 k ，你需要找到该数组中和为 k 的连续的子数组的个数。

示例 1：

输入: nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

说明：

1. 数组的长度为 $[1, 20,000]$ 。
2. 数组中元素的范围是 $[-1000, 1000]$ ，且整数 k 的范围是 $[-1e7, 1e7]$ 。

通过次数 107,670 提交次数 241,957

请问您在哪些招聘中遇到此题？

杜招 校招 实习 未遇到

2 3 2 3 -5 6 ; k=6 prei-prej=6 11-5=6
prei-k = prej 5

题目描述 评论(408) 跟贴(493) 提交记录

```
1 class Solution {
2 public:
3     int subarraySum(vector<int>& nums, int k) {
4         // 记录以为开头,以为结尾的和有多少个
5         unordered_map<int,int> hash;
6         int ans = 0, sum = 0;
7         hash[0] = 1; // 1 1 1 1 1 1 ; k=3; hash[3-3] = hash[0] = 1
8         for(int i = 0; i < nums.size(); ++start) {
9             sum += nums[i];//要放在前面,不然k>0的时候会错 累加当前 前缀和
10            if(hash[sum-k])ans += hash[sum-k]; //如果hash[sum - k]有, 则注意这里是 ans += hash[sum - k] 再添加这么多个符合条件的数组
11            hash[sum]++;
12        }
13        return ans;
14    }
15 };
16 */
17 //前缀和 : nums 的第 0 项到 当前项 的和,
18 //用数组 prefixSum 表示, prefixSum[x]: 第 0 项到 第 x 项 的和。
19 //prefixSum[x] = nums[0] + nums[1] +...+nums[x]
20 //prefixSum[x]=nums[0]+nums[1]+...+nums[x]
21 //sum[j,i] = k <=> prefix[i] - prefix[j] == k <=> prefix[i] - k = prefix[j]
22 //sum中保存着前缀和, 如果满足条件的前缀和,则对ans更新 增加hash[sum - k]个 前面 有hash[sum] ++
23 //如h[0] = 1 -1 1 3 -3 3 , 如h[0] = 2 , k=3, 到h[0] = 2, if(hash[3 - 3]) 再加2个 ---->3; 1 1 1 -1 -1 -1
24 //到 h[0] = 3; if(hash[3 - 3]) 再加3个 ;---->0+3本身, 3, -3; 1 1 1 -1 -1 -3 ; 所以再加3个
25 }
```

题目描述 评论(904) 跟贴(1,3k) 提交记录

242. 有效的字母异位词

难度 简单 379 收藏 为本章

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1：

输入: s = "anagram", t = "nagaram"
输出: true

示例 2：

输入: s = "rat", t = "car"
输出: false

205. 同构字符串

难度: 简单 | 151 | 收藏 | 分享 | 提交记录 | 楼和面

给定两个字符串 s 和 t，判断它们是否是同构的。

如果 s 中的字符可以按某种映射关系替换成 t，那么这两个字符串就是同构的。

每个出现的字符都应映射到同一个字符上。相同的字符只能映射到同一个字符上。字符可以映射到自己本身。

示例 1：

```
输入：s = "egg", t = "add"
输出：true
```

示例 2：

```
输入：s = "foo", t = "bar"
输出：false
```

示例 3：

```
输入：s = "paper", t = "title"
输出：true
```

bool isIsomorphic(string s, string t) {
 //不需要遍历字符串，同时判断每个元素第一次出现的位置，如果相同继续循环，如果不相同return false
 for(int i = 0; i < s.size(); i++){
 if(s.find(s[i]) != t.find(t[i])) return false;
 }
 //foo bar bbbbaaba
 //aaabbbaa
 return true;
}

LC-计数二进制子串

题目描述 | 评论 (431) | 题解 (434) | 提交记录 | 楼和面

696. 计数二进制子串

难度: 简单 | 357 | 收藏 | 分享 | 提交记录 | 楼和面

给定一个字符串 s，计算具有相同数量 0 和 1 的非空（连续）子字符串的数量，并且这些子字符串中的所有 0 和所有 1 都是连接的。

重复出现的子串要计算它们出现的次数。

示例 1：

```
输入："00110011"
输出：6
解释：有6个子串具有相同数量的连续1和0："0011"、"01"、"1100"、"10"、"0011" 和 "01"。
请注意，一些重复出现的子串要计算它们出现的次数。
另外，"00110011"不是有效的子串，因为所有的0 (和1) 没有组合在一起。
```

这个实现的时间复杂度和空间复杂度都是O(n)

对于某一个位置 i，其实我们只关心 i-1 位置的 counts 值是多少，所以可以用一个 last 变量来维护当前位置的前一个位置，这样可以省去一个 counts 数组的空间

时间复杂度 O(n) 空间复杂度 O(1)

LC-基本计算器

2 - (1-3)

1 op = -1 1 -1
op = 1 1

224. 基本计算器

难度: 困难 | 633 | 收藏 | 分享 | 为切换为英文 | 推荐动态 | 反馈

给你一个字符串表达式 s，请你实现一个基本计算器来计算并返回它的值。

示例 1：

```
输入：s = "1 + 1"
输出：2
```

示例 2：

```
输入：s = " 2-1 + 2 "
输出：3
```

示例 3：

```
输入：s = "(1+(4*5+2)-3)+(6+8)"
输出：23
```

1 class Solution {
2 public:
3 int calculate(string s) {
4 stack<char> stk; //为了修正负号，用栈存储负号
5 int num = 0, ans = 0, op = 1;
6 stk.push(op);
7 for(char c : s){
8 if(c == '+') continue;
9 else if(isdigit(c)){
10 num = num * 10 + (c - '0');
11 }
12 else {
13 ans += num * op;
14 num = 0;
15 if(c == '-') op = -1;
16 else if(c == '+') op = 1;
17 else if(c == '(') stk.push(op); //将括号的符号放入栈顶
18 else if(c == ')') stk.pop(); //括号计算完毕，退出括号的这个符号，表示计算完一个括号表达式段
19 }
20 }
21 return ans + num * op; //最后一个元素并没有遇到符号，要加上它与前面符号的计算结果
22 }
23 }

227. 基本计算器 II

难度: 中等 | 392 | 收藏 | 分享 | 提交记录 | 楼和面

给你一个字符串表达式 s，请你实现一个基本计算器来计算并返回它的值。

整数除法仅保留整数部分。

示例 1：

```
输入：s = "3+2*2"
输出：7
```

示例 2：

```
输入：s = " 3/2 "
输出：1
```

示例 3：

```
输入：s = " 3+5 / 2 "
输出：5
```

提示：

```
1 class Solution {
2 public:
3     //遍历字符串 s，并用变量preSign 记录每个数字之前的运算符！！！第一个数字，其之前的运算符视为加号
4     //或者遍历到字符串末尾，认为是遇到了数字末尾，处理完该数字，更新preSign为当前运算符
5     int calculate(string s) {
6         vector<int> stk; //用vector表示stk也可以
7         int num = 0, n = s.length();
8         char preSign = '+';
9
10        for(int i = 0; i < n; i++){
11            if(isdigit(s[i])){
12                num = num * 10 + (s[i] - '0'); //以字符串为单位遍历，计算字符串代表的数字大小
13                if((isDigit(s[i]) && s[i] != '-' || i == n-1) || i == n-1){ //如果字符串末尾也要处理前面计算得到的num
14                    switch(preSign){ //根据最后一个保存的运算符，处理计算得到的数字
15                        case '+': stk.push_back(num); break;
16                        case '-': stk.push_back(-num); break;
17                        case '*': stk.back() *= num; //栈顶元素与当前num相乘，同时更新栈顶元素 2*2+6 遇到+，num计算完毕，处理前面的2 与栈顶元素2*2相乘
18                        case '/': stk.back() /= num; break;
19                    }
20                }
21                num = 0; //重置数位量为0
22            }
23        }
24        int res = accumulate(stk.begin(), stk.end(), 0); //线内元素累加 注意accumulate的第一个参数是迭代器，第三个参数是0，起始值/补偿值
25        return res;
26    }
27 }
```

第十三章 链表

单链表是由节点加指针构成的数据结构，每个结点存有一个值和一个指向下一结点的指针。
 因此很多链表问题可以用递归处理，不同于数组，链表并不能直接获取任意结点的值，必须通过指针先找到该结点再才能获取到值，同理，在未遍历到链表结尾时也无法知道链表的长度，除非依赖其他数据结构储存长度。

```

链表表示方法：
struct ListNode{
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

```

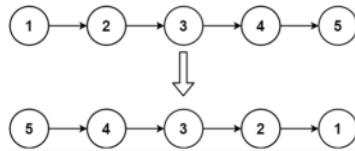
由于在进行链表操作时，尤其在删除结点时，经常会因为对当前结点进行操作导致指针出现问题
两个技巧：一是尽量处理当前结点的下一个结点而非当前结点本身，二是建立一个虚拟头结点dummy
使其指向当前链表的头结点，这样即使原链表的所有结点都删除，也有一个dummy在，返回dummy->next即可
一般写算法题不需要删除内存，删除一个结点直接进行指针操作无需回收内存，而实际做软件工程时对于无用内存
应该显式回收或使用智能指针

206. 反转链表

难度 简单 | 1730 | 通过 94 | 提交 144

给你单链表的头节点 head，请你反转链表，并返回反转后的链表。

示例 1：



输入 : head = [1,2,3,4,5]

输出 : [5,4,3,2,1]

递归的写法为：

```

ListNode* reverseList(ListNode* head, ListNode* prev=nullptr) {
    if (!head) {
        return prev;
    }
    ListNode* next = head->next;
    head->next = prev;
    return reverseList(next, head);
}

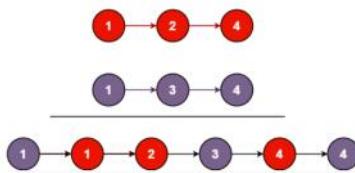
```

21. 合并两个有序链表

难度 简单 | 1703 | 通过 104 | 提交 276 | 调试记录

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1：



输入 : l1 = [1,2,4], l2 = [1,3,4]
输出 : [1,1,2,3,4,4]

示例 2：

输入 : l1 = [], l2 = []
输出 : []

递归写法：

```

ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    if (!l1) {
        return l2;
    }
    if (!l2) {
        return l1;
    }
    if (l1->val > l2->val) {
        l2->next = mergeTwoLists(l1, l2->next);
        return l2;
    }
    l1->next = mergeTwoLists(l1->next, l2);
    return l1;
}

```

链表输入

```

struct ListNode {
    int val;
    ListNode* next;
    ListNode() : val(0), next(nullptr) {};
    ListNode(int x) : val(x), next(nullptr) {};
};

ListNode* create(int n)
{
    auto dummy = new ListNode(0);
    auto pre = dummy;
    for (int i = 0; i < n; i++) {
        auto p = new ListNode(0);
        cin >> p->val;
        pre->next = p;
        pre = pre->next;
    }
    return dummy->next;
}

```

```

1 /**
2  * Definition for singly-linked list.
3  * struct ListNode {
4  *     int val;
5  *     ListNode *next;
6  *     ListNode() : val(0), next(nullptr) {}
7  *     ListNode(int x) : val(x), next(nullptr) {}
8  *     ListNode(int x, ListNode *next) : val(x), next(next) {}
9  * };
10 */
11 class Solution {
12 public:
13     ListNode* reverseList(ListNode* head) {
14         ListNode *pre = nullptr, *cur = head;
15         while(cur){
16             ListNode *tmp = cur->next;
17             cur->next = pre;
18             pre = cur;
19             cur = tmp;
20         }
21         return pre;
22     }
23 };

```

```

1 /**
2  * Definition for singly-linked list.
3  * struct ListNode {
4  *     int val;
5  *     ListNode *next;
6  *     ListNode() : val(0), next(nullptr) {}
7  *     ListNode(int x) : val(x), next(nullptr) {}
8  *     ListNode(int x, ListNode *next) : val(x), next(next) {}
9  * };
10 */
11 class Solution {
12 public:
13     ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
14         ListNode *dummy = new ListNode(0), *node = dummy;
15         while(l1 && l2){
16             if(l1->val <= l2->val){
17                 node->next = l1;
18                 l1 = l1->next;
19             }
20             else {
21                 node->next = l2;
22                 l2 = l2->next;
23             }
24             node = node->next;
25         }
26         node->next = l1? l1 : l2;
27         return dummy->next;
28     }
29 };
30 };

```

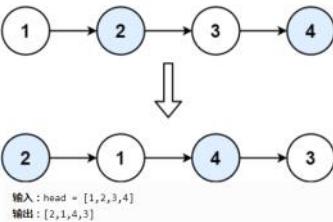
24. 两两交换链表中的节点

难度 中等 | 909 收藏 | 分享 | 举报

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是简单的改变节点内部的值，而是需要实际的进行节点交换。

示例 1：



- 时间复杂度：O(n)，其中 n 是链表的节点数量。需要对每个节点进行更新指针的操作。
- 空间复杂度：O(1)

递归写法

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        ListNode* newHead = head->next;
        head->next = swapPairs(newHead->next);
        newHead->next = head;
        return newHead;
    }
};
```

时间复杂度：O(n)，其中 n 是链表的节点数量。需要对每个节点进行更新指针的操作

空间复杂度：O(n)，其中 n 是链表的节点数量。空间复杂度主要取决于递归调用的栈空间

LC-奇偶链表

题目描述 提交 (660) 显示代码 (999) 提交记录

328. 奇偶链表

难度 中等 | 418 收藏 | 分享 | 举报

给定一个单链表，把所有的奇数节点和偶数节点分别拼接在一起。请注意，该题的奇数节点和偶数节点指的都是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(n)，nodes 为节点总数。

示例 1：

输入：1->2->3->4->5->NULL
输出：1->3->2->4->5->NULL

示例 2：

输入：2->1->3->5->6->4->7->NULL
输出：2->3->1->5->4->6->7->NULL

说明：

- 必须保证奇数节点和偶数节点的相对顺序。
- 第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

第十四章 树

单链表的升级版，二叉树，每个结点最多有两个子结点，leetcode树的表示默认：

```
struct TreeNode{
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
```

与链表的差别就是多了一个子结点的指针

树的递归：可以写那种 one-line code，将 if-else 判断语句压缩成问号冒号的形式但是新手建议使用 if-else 语句，通常树的递归写法与深度优先搜索的递归写法相同

递归三步：

1. 确定递归函数的参数和返回值：

参数：传入树的根结点

返回：这棵树的深度，返回值为 int 类型

int getDepth(TreeNode* node)

2. 确定递归终止条件：

如果为空，返回 0

if(node == null) return 0;

3. 确定单层递归的逻辑：

先求它左子树的深度，再求右子树的深度，取两者最大值
再加1（表示以当前结点为根结点的树的深度）

104. 二叉树的最大深度

难度 简单 | 875 收藏 | 分享 | 举报

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明：叶子节点是指没有子节点的节点。

示例：

给定二叉树 [3,9,20,null,null,15,7]，

```
3
 / \
9  20
 / \
15  7
```

返回它的最大深度 3。

```
1 /**
2  * Definition for singly-linked list.
3  */
4 struct ListNode {
5     int val;
6     ListNode *next;
7     ListNode() : val(0), next(nullptr) {}
8     ListNode(int x) : val(x), next(nullptr) {}
9     ListNode(int x, ListNode *next) : val(x), next(next) {}
10 }
11
12 class Solution {
13 public:
14     ListNode* oddEvenList(ListNode* head) {
15         ListNode *evenhead = head->next; //设置奇数头结点在头部位置
16         ListNode *odd = head; //偶数
17         ListNode *even = evenhead; //偶数
18         while (even && even->next){ //注意条件是even && even->next 同时考虑了奇数与偶数的边界
19             odd->next = even->next; //先将odd指向even (这里第一个head指的是odd头结点)
20             odd = odd->next; //更新奇数odd的结点
21             even->next = odd->next; //再将偶数的结点指向奇数的even结点
22             even = even->next; //更新偶数的结点
23         } //循环余数，但是阅读的话结点不容易，一直在理解
24         odd->next = evenhead; //将往循环移动的奇数结点的指向下一节点指向evenhead单元素结点处 将奇数与偶数分别串联起来，再进行拼接 挪地操作进行排序
25         return head; //返回头结点的链表 返回头结点的链表 表示整个链表转换过来
26     }
27 };
```

题目描述 提交 | i | P | C | x | o

```
1 /**
2  * Definition for singly-linked list.
3  */
4 struct ListNode {
5     int val;
6     ListNode *next;
7     ListNode() : val(0), next(nullptr) {}
8     ListNode(int x) : val(x), next(nullptr) {}
9     ListNode(int x, ListNode *next) : val(x), next(next) {}
10 }
11
12 class Solution {
13 public:
14     ListNode* oddEvenList(ListNode* head) {
15         if(head == nullptr) return head; //设置奇数头结点在头部位置
16         ListNode *evenhead = head->next; //odd      even
17         ListNode *odd = head; //偶数
18         ListNode *even = evenhead; //偶数
19         while (even && even->next){ //注意条件是even && even->next 同时考虑了奇数与偶数的边界
20             odd->next = even->next; //先将odd指向even (这里第一个head指的是odd头结点)
21             odd = odd->next; //更新奇数odd的结点
22             even->next = odd->next; //再将偶数的结点指向奇数的even结点
23             even = even->next; //更新偶数的结点
24         } //循环余数，但是阅读的话结点不容易，一直在理解
25         odd->next = evenhead; //将往循环移动的奇数结点的指向下一节点指向evenhead单元素结点处 将奇数与偶数分别串联起来，再进行拼接 挪地操作进行排序
26         return head; //返回头结点的链表 返回头结点的链表 表示整个链表转换过来
27 };
```

题目描述 提交 | i | P | C | x | o

```
1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 }
12
13 class Solution {
14 public:
15     int maxDepth(TreeNode* root) {
16         if(root == nullptr) return 0;
17         int left = maxDepth(root->left);
18         int right = maxDepth(root->right);
19         return max(left, right) + 1;
20     }
21 };
```

迭代法：使用层序遍历 二叉树的层数就是深度

一层一层遍历二叉树，使用队列保存每一层加入的结点，同时弹出当前层结点，队列操作的次数/弹出当前层结点与压入下一层结点 就是对应遍历的层数

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == NULL) return 0;
        int depth = 0;
        queue<TreeNode*> que;
        que.push(root);
        while(!que.empty()) {
            int size = que.size();
            depth++; // 遍历深度
            for (int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                que.pop();
                if (node->left) que.push(node->left);
                if (node->right) que.push(node->right);
            }
        }
        return depth;
    }
};
```

110. 平衡二叉树

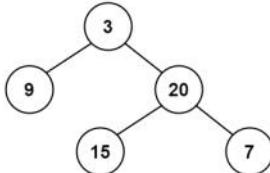
难度 简单 □ 682 ⚪ 10 ⚫ 0 ⚪ 0

给定一个二叉树，判断它是否是高度平衡的二叉树。

平衡中，一棵高度平衡二叉树定义为：

一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

示例 1：



输入：root = [3,9,20,null,null,15,7]

输出：true

```
1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  * };
8  * TreeNode() : val(0), left(nullptr), right(nullptr) {}
9  * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10 * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 */
12 class Solution {
13 public:
14     int helper(TreeNode *root){
15         if(root == nullptr)
16             return 0;
17         int left = helper(root->left);
18         int right = helper(root->right); //在处理子树时发现其已经不平衡了，则可以返回一个-1。递归函数结束返回时如果当前结点接收到返回值为-1 上面所有parent节点可以避免多余的判断 直接也返回-1一直到root
19         if(left == -1 || right == -1 || abs(left - right)>1) //如果这个abs(left-right)计算开销很大，就会节省很大的计算时间
20             return -1;
21         return 1 + max(left, right);
22     }
23
24     bool isBalanced(TreeNode* root) {
25         return helper(root) != -1;
26     }
27 };
```

复杂度分析

- 时间复杂度： $O(N)$ ，其中 N 为二叉树的节点数，即遍历一棵二叉树的时间复杂度，每个结点只被访问一次。
- 空间复杂度： $O(Height)$ ，其中 $Height$ 为二叉树的高度。由于递归函数在递归过程中需要为每一层递归函数分配栈空间，所以这里需要额外的空间且该空间取决于递归的深度，而递归的深度虽然为二叉树的高度，并且每次递归调用的函数量又只用了常数个变量，所以所需空间复杂度为 $O(Height)$ 。

101. 对称二叉树

难度 简单 □ 1374 ⚪ 10 ⚫ 0 ⚪ 0

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 `[1,2,2,3,4,3]` 是对称的。

```
1
/ \
2   2
/ \ / \
3  4 5  3
```

但是下面这个 `[1,2,2,1,3,3,3]` 则不是镜像对称的：

```
1
/ \
2   2
\   \
3   3
```

但是下面这个 `[1,2,2,1,3,3,3]` 则不是镜像对称的：

637. 二叉树的层平均值

难度 简单 □ 260 ⚪ 10 ⚫ 0 ⚪ 0

给定一个非空二叉树，返回一个由每层节点平均值组成的数组。

示例 1：

```
输入：
3
/
9 20
/ \
15 7
输出：[3, 14.5, 11]
解释：
第 0 层的平均值是 3， 第1层是 14.5， 第2层是 11。因此返回 [3, 14.5, 11]。
```

提示：

• 节点值的范围在32位有符号整数范围内。

通过次数 62,317 | 提交次数 90,533

询问你在哪段代码中遇到此题？

举报 | 报错 | 实习 | 未遇到

© 力扣 (LeetCode)版权所有

使用广度优先搜索进行层序遍历，使用一个队列实现，在开始遍历一层的结点时，当前队列中的结点个数/size 就是当前层的结点个数，只要每次控制遍历size个结点，就能保证每次遍历的都是当前层的结点-层序遍历

前中后序遍历 根：访问根结点；左 右：递归遍历左右子树
这是三种利用深度优先遍历二叉树的方式；考虑一棵二叉树

```
1
/ \
2   3
/ \ \
4   5   6
```

先序遍历先遍历根结点，再遍历左结点，再遍历右结点 根左右
void preorder(TreeNode* root){

```

visit(root);
preorder(root->left);
preorder(root->right);
}
1 - 2 - 4 - 5 - 3 - 6

中序遍历先遍历左结点，再遍历右结点 左根右
void inorder(TreeNode* root){
    inorder(root->left);
    visit(root);
    inorder(root->right);
}
4 - 2 - 5 - 1 - 3 - 6

后序遍历先遍历左结点，再遍历右结点，再遍历根结点 左右根
void postorder(TreeNode* root){
    postorder(root->left);
    postorder(root->right);
    visit(root);
}

```

4 - 5 - 2 - 6 - 3 - 1

题目描述 | 评论 (57) | 提交 (1.1k) | 提交记录

105.从前序与中序遍历序列构造二叉树

难度 中等 | 1028 收藏 分享 切换为英文 接收动态 反馈

根据一棵树的前序遍历与中序遍历构造二叉树。

注意：你可以假设该树中没有重复的元素。

例如，给出

前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

```

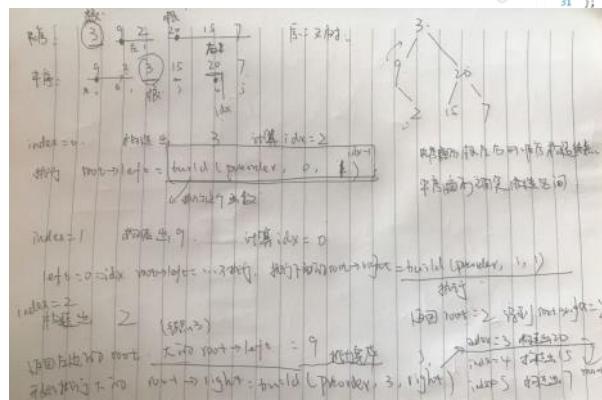
3
 / \
9  20
 /  \
15   7

```

通过次数: 191,431 | 提交次数: 274,710

询问您在题类招物中遇到此题? 社区 校招 实习 未遇到

© 力扣 (LeetCode) 版权所有



不使用递归，实现二叉树的前序遍历。

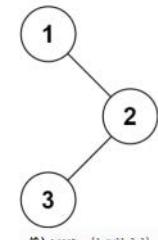
题目描述 | 评论 (780) | 提交 (1.4k) | 提交记录

144.二叉树的前序遍历

难度 简单 | 565 收藏 分享

给你二叉树的根节点 root ，请你按逆时针顺序 返回前序遍历。

示例 1：



输入：root = [1,null,2,3]
输出：[1,2,3]

二叉查找树 Binary Search Tree

对于每个根结点其左子结点的值小于它，右子结点的值大于它

对于一个二叉查找树可以在O(nlogn)的时间内查找一个值是否在

从根结点开始若当前值大于查找值向左下走，若当前值小于查找值，向右下走

因为二叉查找树是有序的，对二叉查找树中序遍历的结果即为排序好的数组

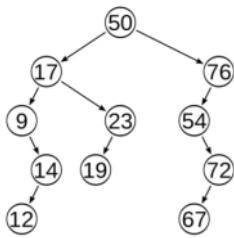


图 14.1: 二叉查找树样例

题目描述 | 评论 (0) | 提交 (446) | 检查记录

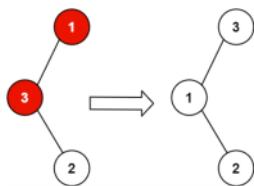
99. 修复二叉搜索树

难度: 中等 | 通过率: 40% | 提交数: 446 | 检查数: 0

给你二叉搜索树的根节点 root ，该树中的两个子节点被错误交换。请在不改变树的结构的情况下，恢复其正确性。

进阶：使用 O(n) 空间复杂度的算法很容易实现。你能想出一个只使用常数空间的解决方案吗？

示例 1：

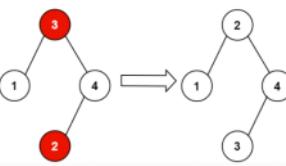


输入 : root = [1,3,null,null,2]

输出 : [1,2,null,null,3]

解释 : 3 不应该是 1 左孩子，因为 3 > 1。交换 1 和 3 使二叉搜索树有效。

示例 2：



输入 : root = [3,1,4,null,null,2]

输出 : [2,1,4,null,null,3]

解释 : 2 不应在 3 的右子树中，因为 2 < 3。交换 2 和 3 使二叉搜索树有效。

LC-修剪二叉树

题目描述 | 评论 (260) | 提交 (236) | 检查记录

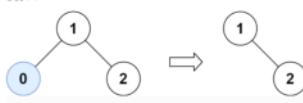
669. 修剪二叉搜索树

难度: 中等 | 通过率: 30% | 提交数: 849 | 检查数: 0

给你二叉搜索树的根节点 root ，同时给定最小边界 low 和最大边界 high 。通过修剪二叉搜索树，使得所有节点的值在 [low, high] 中。修剪树不应该改变保留树中的元数据相对位置（即，如果没被移除，原有的父子代关系都应该保留）。可以证明，在唯一的答案。

所以结果应该是经过修剪后的二叉搜索树的新的形状。注意，结果树可能与给定给你的边界发生改变。

示例 1：



输入 : root = [1,0,2], low = 1, high = 2

输出 : [1,null,2]

题目描述 | 评论 (1,0) | 提交 (1,5) | 检查记录

226. 转二叉树

难度: 简单 | 通过率: 849 | 提交数: 1,144 | 检查数: 0

翻转二叉树。

示例：

输入：

```

    4
   / \
  2   7
 / \ / \
1  3 6  9
  
```

输出：

```

    4
   / \
  7   2
 / \ / \
9  6 3  1
  
```

用栈进行迭代实现 注意是先压右子结点入栈 保证先遍历到左子结点

C++代码迭代法 (前序遍历)

```

class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL) return root;
        stack<TreeNode*> st;
        st.push(root);
        while(!st.empty()) {
            TreeNode* node = st.top(); // 左
            st.pop();
            swap(node->left, node->right);
            if(node->right) st.push(node->right); // 右
            if(node->left) st.push(node->left); // 左
        }
        return root;
    }
};
  
```

```

 1 /**
 2  * Definition for a binary tree node.
 3  * struct TreeNode {
 4  *     int val;
 5  *     TreeNode *left;
 6  *     TreeNode *right;
 7  * };
 8  * TreeNode(): val(0), left(nullptr), right(nullptr) {}
 9  * TreeNode(int x): val(x), left(nullptr), right(nullptr) {}
10  * TreeNode(int x, TreeNode *left, TreeNode *right): val(x), left(left), right(right) {}
11 */
12 class Solution {
13 public:
14     TreeNode* trimBST(TreeNode* root, int low, int high) {
15         if(root == nullptr)
16             return root; //root为空，直接返回
17         else if(root->val < low) //root->val < low: 说明目前root太小，root不在范围内，则需要去找root右边子树
18             return trimBST(root->right, low, high);
19         else if(root->val > high) //root->val > high: 说明现在root太大，root不在范围内，则需要去找root左边子树
20             return trimBST(root->left, low, high);
21         root->left = trimBST(root->left, low, high); //在范围内，那么继续构建它的左右子树即可（这里没有先后顺序的差异）
22         root->right = trimBST(root->right, low, high);
23         return root;
24     }
25 };
  
```

题目描述 | 评论 (1,0) | 提交 (1,5) | 检查记录

669. 修剪二叉搜索树

难度: 中等 | 通过率: 30% | 提交数: 849 | 检查数: 0

给你二叉搜索树的根节点 root ，同时给定最小边界 low 和最大边界 high 。通过修剪二叉搜索树，使得所有节点的值在 [low, high] 中。修剪树不应该改变保留树中的元数据相对位置（即，如果没被移除，原有的父子代关系都应该保留）。可以证明，在唯一的答案。

所以结果应该是经过修剪后的二叉搜索树的新的形状。注意，结果树可能与给定给你的边界发生改变。

示例 1：



输入 : root = [1,0,2], low = 1, high = 2

输出 : [1,null,2]

题目描述 | 评论 (1,0) | 提交 (1,5) | 检查记录

226. 转二叉树

难度: 简单 | 通过率: 849 | 提交数: 1,144 | 检查数: 0

翻转二叉树。

示例：

输入：

```

    4
   / \
  2   7
 / \ / \
1  3 6  9
  
```

```

    4
   / \
  7   2
 / \ / \
9  6 3  1
  
```

用栈进行迭代实现 注意是先压右子结点入栈 保证先遍历到左子结点

C++代码迭代法 (前序遍历)

```

 1 /**
 2  * Definition for a binary tree node.
 3  * struct TreeNode {
 4  *     int val;
 5  *     TreeNode *left;
 6  *     TreeNode *right;
 7  * };
 8  * TreeNode(): val(0), left(nullptr), right(nullptr) {}
 9  * TreeNode(int x): val(x), left(nullptr), right(nullptr) {}
10  * TreeNode(int x, TreeNode *left, TreeNode *right): val(x), left(left), right(right) {}
11 */
12 class Solution {
13 public:
14     TreeNode* invertTree(TreeNode* root) {
15         if (root == nullptr)
16             return root;
17         stack<TreeNode*> st;
18         st.push(root);
19         while(!st.empty()) {
20             TreeNode* node = st.top(); // 左
21             st.pop();
22             swap(node->left, node->right);
23             if(node->right) st.push(node->right); // 右
24             if(node->left) st.push(node->left); // 左
25         }
26         return root;
27     }
28 };
  
```

题目描述 提交 (525) □ 题解 (61) ○ 模拟面试 C++ * 题解提交

```
617. 合并二叉树
难度: 简单 | 688 | 10 | 为 | 反馈 | 举报

给出两个二叉树，想要将你将它们中的一个覆盖到另一个上。两个二叉树的一些节点需要重叠。
你需要将两个合成为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则为 NULL 的节点将直接作为两个二叉树的节点。
示例 1:
输入:
Tree 1           Tree 2
      1             2
     / \           / \
    3   2         1   3
   / \           \   \
  5   4         4   7
输出:
合并后的树:
      3
     / \
    4   5
   / \   \
  5   4   7

```

输出:

```
Tree 1           Tree 2
      1             2
     / \           / \
    3   2         1   3
   / \           \   \
  5   4         4   7
输出:
合并后的树:
      3
     / \
    4   5
   / \   \
  5   4   7

```

题目描述 提交 (518) □ 题解 (599) ○ 模拟面试 C++ * 题解提交

```
572. 另一个树的子树
难度: 简单 | 481 | 10 | 为 | 反馈 | 举报

给定两个非空二叉树  $s$  和  $t$ ，检验  $s$  中是否包含和  $t$  具有相同的结构和节点值的子树。 $s$  中一个子树包括  $s$  的一个节点和这个节点的所有子孙。 $s$  也可以看做自身的一棵树。
示例 1:
给定树:
3
/ \
4   5
/ \
1   2
给定的树 t:
4
/ \
1   2
返回 true，因为 t 与 s 的一个子树有相同的结构和节点值。
示例 2:
给定的树 s:

```

```
1 /**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 * };
 */
class Solution {
public:
    bool isSubtree(TreeNode* s, TreeNode* t) {
        if(s == nullptr || t == nullptr)
            return false;
        if(dfs(s,t))
            return true;
        return isSubtree(s->left,t) || isSubtree(s->right,t);
    }
    bool dfs(TreeNode* s, TreeNode* t){
        if(s == nullptr && t == nullptr)
            return true;
        if(s == nullptr || t == nullptr)
            return false;
        if(s->val != t->val)
            return false;
        return dfs(s->left,t->left) && dfs(s->right,t->right);
    }
};

```

题目描述 提交 (702) □ 题解 (602) ○ 模拟面试 C++ * 题解提交

```
404. 左叶子之和
难度: 简单 | 312 | 10 | 为 | 反馈 | 举报

计算给定二叉树的所有左叶子之和。
示例:
3
/ \
9  20
/ \
15  7
在这个二叉树中，有两个左叶子，分别是 9 和 15，所以返回 24
通过次数 80,105 | 提交次数 140,597
询问: 在哪部分代码中遇到此题?
杜恒 | 极客 | 实习 | 半遭到
© LeetCode 版权所有
```

```
1 /**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 * };
 */
class Solution {
public:
    int sum = 0;
    int sumOfLeftLeaves(TreeNode* root) {
        if(root == nullptr)
            return 0;
        dfs(root); //将root传入dfs函数对整棵树进行遍历
        return sum;
    }
    void dfs(TreeNode* root){
        if(root == nullptr)
            return;
        if(root->left != nullptr && root->left->left == nullptr && root->left->right == nullptr)
            sum += root->left->val;
        dfs(root->left); //对整棵树进行遍历，当我们遍历到它的左节点，执行该函数如果它的左子节点是一个叶子节点，那么就将它的左子节点的值累加计入答案
        dfs(root->right);
    }
};

```

题目描述 提交 (702) □ 题解 (602) ○ 模拟面试 C++ * 题解提交

```
513. 找树左下角的值
难度: 中等 | 108 | 10 | 为 | 反馈 | 举报

给定一个二叉树，在树的最后一行找到最左边的值。
示例 1:
输入:
2
/ \
1   3
输出:
1
示例 2:
输入:
1
/ \
2   3
/ \
4   5
/ \
6   7
输出:
6
输出:
6

```

```
1 /**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 * };
 */
class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        queue q;
        q.push(root);
        int res = 0;
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                TreeNode* node = q.front();
                q.pop();
                if(i == 0)
                    res = node->val;
                if(node->left) q.push(node->left);
                if(node->right) q.push(node->right);
            }
        }
        return res;
    }
};

```

538. 二叉搜索树转换为累加树

难度: 中等 | 523 | 收藏 | 反馈 | 举报

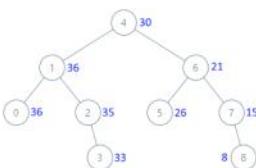
给出二叉搜索树的根节点，该树的节点值各不相同。请你将其转换为累加树（Greater Sum Tree），使每个节点 node 的新值等于原树中大于等于 node.val 的值之和。

提醒一下，二叉搜索树满足以下的规则：

- 节点的左子树仅包含值小于 节点键的节点。
- 节点的右子树仅包含值 大于 节点键的节点。
- 左右树的深度一致 - 二叉搜索树。

注意：本题和 108: <https://leetcode-cn.com/problems/search-tree-greater-sum-tree/> 相同

示例 1：



235. 二叉搜索树的最近公共祖先

难度: 困难 | 582 | 收藏 | 反馈 | 举报

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

而南开大学中最近公共祖先的定义为：“对于有根树 T 的两个结点 p, q, 最近公共祖先表示为一个结点 x，满足 x 是 p, q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

假设所给的二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]，p = 2, q = 8。

示例 1:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6

解释: 节点 2 和节点 8 的最近公共祖先是 6。

530. 二叉搜索树的最小绝对差

难度: 简单 | 251 | 收藏 | 反馈 | 举报

给你一棵所有节点为非负值的二叉搜索树，请你计算树中任意两节点的差的绝对值的最小值。

示例：

输入:

```
1
 \
 3
 /
 2
```

输出:

1

解释：
最小绝对差为 1，其中 2 和 1 的绝对值为 1 (或者 2 和 3)。

```
1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 }
12
13 class Solution {
14 public: //二叉搜索树的中序遍历是一个单调递增的有序序列。如果我们反向地中序遍历二叉搜索树，即可得到一个单调递减的有序序列。
15 //顺序要求我们遍历每一个节点的遍历次序为根的节点加上所有大于它的节点值之和
16 //只要遍历反向中序遍历二叉搜索树，记录过程中的节点值之和，即不断更新当前遍历的节点的节点值，即可得到题目要求的最小值
17
18    int sum = 0;
19    TreeNode* convertBST(TreeNode* root) {
20        if(root != nullptr){
21            convertBST(root->right);
22            sum += root->val;
23            root->val = sum;
24            convertBST(root->left); //可以不接收这个返回值，但是函数一定要有返回值，因为最终要返回整个树的根节点 root
25        }
26        return root; //两个大的convertBST 执行完，返回大的根节点 root 。函数中的root->val 表示直接对原始树往回遍历完的节点
27    }
}
```

```
1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9 }
10
11 //从根节点开始遍历
12 //如果当前节点的值大于 pp 和 qq 的值，说明 pp 和 qq 应该在当前节点的左子树，因此将当前节点移到它的左子节点
13 //如果当前节点的值小于 pp 和 qq 的值，说明 pp 和 qq 应该在当前节点的右子树，因此将当前节点移到它的右子节点
14 //如果当前节点的值不满足上述两条要求，那么说明当前节点就是「公共点」。此时，p 和 q 要么在当前节点的不同的子树中，要么其中一个就是当前节点。
15 class Solution {
16 public:
17     TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
18         TreeNode* ancestor = root;
19         while(true){
20             if(p->val < ancestor->val && q->val < ancestor->val)
21                 ancestor = ancestor->left;
22             else if(p->val > ancestor->val && q->val > ancestor->val)
23                 ancestor = ancestor->right;
24             else
25                 break;
26         }
27         return ancestor;
28     }
29 };
```

```
1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 }
12
13 class Solution {
14 public: //对于中序数组两元素绝对值最小一定是相邻的两个元素之间 先中序遍历BST
15     vector<int> res;
16     void dfs(TreeNode* root){
17         if(!root){
18             dfs(root->left);
19             res.push_back(root->val);
20             dfs(root->right);
21         }
22     }
23     int getMinimumDifference(TreeNode* root) {
24         dfs(root); //这一步不能少，要res都是空栈，必须先遍历，得到中序数组序列，第一步就是树的遍历
25         int ans = INT_MAX;
26         for(int i = 1; i < res.size(); i++){
27             int temp = res[i] - res[i-1];
28             ans = min(ans,temp);
29         }
30         return ans;
31     }
}
```

第十五章 图

图数据结构是树的升级版，通常分为有向图和无向图，有循环或无循环，所有结点相连或不相连

树就是一个相连的无向无环图，另一种很常见的是有向无环图

图通常有两种表示方法，假设图中一共有n个结点，m条边，第一种方法是邻接矩阵，建立一个n*n的矩阵，如果第i个结点连向第j个结点，则C[i][j] = 1，否则为0。如果是无向的，则矩阵是一个对称矩阵G[i][j] = G[j][i]。第二种表示方式是邻接链表，建立一个大小为n的数组，每个位置储存一个数组或链表，表示第i个结点连向其他结点，邻接矩阵空间开销更大，但是邻接链表不支持快速查找

i与j是否相连，此外可以用一个m*2的矩阵储存所有的边

拓扑排序 (topological sort) 是一种常见的，对有向无环图排序的算法。

给定有向无环图中的N个节点，我们把它们排序成一个线性序列；若原图中节点i指向节点j，则排序结果中i一定在j之前。拓扑排序的结果不是唯一的，只要满足以上条件即可

第十六章 更复杂的数据结构

指针实现的三种数据结构，c++自带的STL，以及这些数据结构进行嵌套联动

并查集 (union-find) 可以动态的连通两个点，可以快速判断两个点是否连通这一部分有专门练习题

其他复合数据结构 如LRU数据结构 采用list<pair<int,int>>链表储存信息的key与value 采用map进行快速搜索与辅助记录 其他如minstack数据结构 及实现待续...

框架

2021年2月6日 10:49

```
struct TreeNode{
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x):val(x), left(nullptr), right(nullptr) {}
}
```

数据结构最根本的就是数组 和 链表 其它的都是衍生结构！

数组连续存储，链表不连续存储，消耗空间更大

数据结构的基本操作

无非就是 遍历+访问

无非就是两种方式 线性和非线性

89

线性的就是以for/while 迭代 非线性的就是以递归为代表

数组遍历框架---迭代

```
void traverse(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        //这里是迭代访问arr[i] ...
    }
}
```

链表遍历框架 --- 迭代和递归结构

单链表结点

```
void traverse(ListNode* head){
    for (ListNode* p = head; p!=NULL; p = p ->next){
        //迭代访问 p->val...
    }
}
```

```
void traverse(ListNode* head){
    //这里是递归访问 head->val;
    traverse(head ->val);
}
```

二叉树遍历框架---典型非线性递归

```
void traverse (TreeNode* root) {
    traverse(root ->left);
    traverse(root ->right);
}
```

N叉树的遍历框架

```
class TreeNode{
    int val;
    TreeNode*[] children;
}
void traverse (TreeNode* root) {
    for (TreeNode* child : children)
        traverse(child);
}
```

数据结构的增删改查，都是遍历+访问，都是这些框架基础

二叉树的遍历框架：

```
void traverse (TreeNode* root){
    //前序遍历(前序访问)
    traverse(root ->left);
    //中序遍历...
    traverse(root ->right);
    //后序遍历...
}
```

所有的递归问题几乎都是这种树的遍历问题

动态规划套路框架

自顶向下的递归函数---分解子问题

自顶向下递归分解，从上到下的填充一个备忘录

体现在递归函数的参数

自底向上的迭代函数---也是分解子问题

自底向上的推导最终的结果，从下到上填充一个dp table

体现在dp表格的索引

dp 函数体现在函数参数，而 dp 数组体现在数组索引：

重叠子问题、最优子结构、状态转移方程（代码） 就是动态规划三要素

思考状态转移方程：

明确 base case(最基本的情形) 目标金额 amount 为 0 时算法返回 0，因为不需要任何硬币就已经凑出目标金额 // base case dp[0] = 0;

明确「状态」(原问题和子问题中会变的量) 硬币数量无限，硬币的面额也是题目给定的，只有目标金额会不断地向 base case 靠近，所以唯一的「状态」就是目标金额 amount

明确「选择」 目标金额为什么变化呢，因为你在选择硬币，你每选择一枚硬币，就相当于减少了目标金额 硬币的面值，就是你的「选择」

定义 dp 数组/函数的含义 题目要求我们计算凑出目标金额所需的最少硬币数量

dp 函数dp(n) 的定义：输入一个目标金额 n，返回凑出目标金额 n 的最少硬币数量 subproblem = dp(n - coin) res = min(res, 1 + subproblem)

dp 数组的定义：当目标金额为 i 时，至少需要 dp[i] 枚硬币凑出 dp[i] = min(dp[i], 1 + dp[i - coin]); 外层 for 循环在遍历所有状态内层 for 循环在求所有选择 面值类型

计算机所有的问题都是先思考如何穷举，再追求聪明的穷举 备忘录，dp table 就是空间换时间的思想 降低时间复杂度 return dp[amount]

amount 0 1 2 3 4 5 ... 9 10 11

index 0 1 2 3 4 5 ... 9 10 11

dp	0	1	1	2	2	1	...	3	2	3

1 + min(dp[4], dp[3], dp[0]) 1 + min(dp[10], dp[9], dp[6])

动态规划的穷举并不容易 有时需要一些递归函数

一般都是通过状态转移方程穷举所有解，再求最值

1.明确状态 dp函数的参数 dp数组的索引会变 --就是状态

2.明确选择 通过循环，递归的计算每一个选择

3.明确dp函数/数组的定义 通过选择明确定义 写出正确的状态转移方程

4.明确base case 最简单的情形

核心：找到子问题，找到状态转移方程

1.暴力递归 ---> 2.优化到：带备忘录的递归（消除重复状态冗余计算）---> 3.改写/优化到使用dp table迭代计算

dp数组就充当了递归函数的作用

动态规划解法代码框架

```
# 初始化 base case
dp[0][0][...]=base
# 进行状态转移
for 状态1 in 状态1的所有取值:
    for 状态2 in 状态2的所有取值:
        for ...
            dp[状态1][状态2][...] = 求最值(选择1, 选择2...)

class Solution {
    经典动态规划问题：最小编辑距离
public:
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size();
        vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
        for (int i = 1; i <= m; i++) {
            dp[i][0] = i; // 初始化 base case
        }
        for (int j = 1; j <= n; j++) {
            dp[0][j] = j;
        }
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (word1[i-1] == word2[j-1]) {
                    dp[i][j] = dp[i-1][j-1];
                } else {
                    dp[i][j] = min(dp[i-1][j], min(dp[i][j-1], dp[i-1][j-1])) + 1;
                }
            }
        }
        return dp[m][n];
    }
}
```

数组索引从0开始，索引范围为0-n-1，n个数

如果申请20 如果需要算f(20)，那就要返回memo[20]，需要申请21大小的空间

最优子结构 如全校的最高分可以由每个班的最高分相加 全校的最高分可以由每个班的最高分推导得到

但是 全校的最高分不能通过每个班的最高分差得到 全校的最高分=最高分-最低分 也许在不同班级

其实就分差这个问题来说它不符合最优子结构 即没办法通过子问题推导出规模更大的原问题

动态规划首先要知道 解决如何穷举的问题 状态转移方程本身就是一个暴力解 再加备忘录 用dp table 优化解

回溯算法解题套路框架

回溯算法即常说的DFS算法 本质上就是一种暴力穷举法

解决一个回溯问题，实际就是一个决策树的遍历过程

1.路径：已经作出的选择

2.选择列表：当前可以作的选择

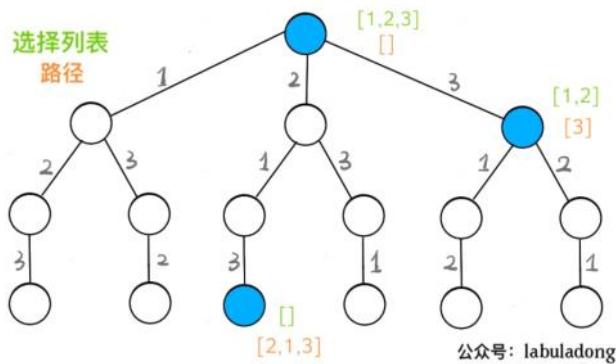
3.结束条件：到达决策树底层，无法再作选择的条件

框架：

```
def backtrack(路径, 选择列表):
    if 满足结束条件(触发结束条件):
        result.add(路径)
        return
    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

for循环里面的递归，在递归调用之前做选择，在递归调用之后撤销选择

如果明白了这几个名词，可以把「路径」和「选择」列表作为决策树上每个节点的属性，比如下图列出了几个节点的属性：



「结束条件」就是遍历到树的底层，在这里就是选择列表为空

定义的 backtrack 函数其实就像一个指针，

递归的过程相当于指针在这棵树上移动访问，同时要正确维护每个节点的属性

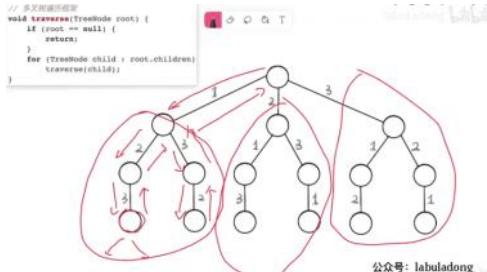
每当走到树的底层，其「路径」就是一个全排列

框架

```
// 回溯算法框架
List<Value> result;
void backtrack(路径, 选择列表) {
    if (满足结束条件) {
        result.add(路径);
        return;
    }
    for (选择 : 选择列表) {
        做选择;
        backtrack(路径, 选择列表);
        撤销选择;
    }
}

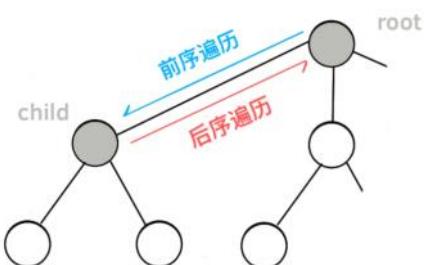
// 多叉树遍历框架
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    for (TreeNode child : root.children)
        traverse(child);
}

1 void traverse(TreeNode root) {
2     for (TreeNode child : root.children)
3         // 前序遍历需要的操作
4         traverse(child);
5         // 后序遍历需要的操作
6 }
```



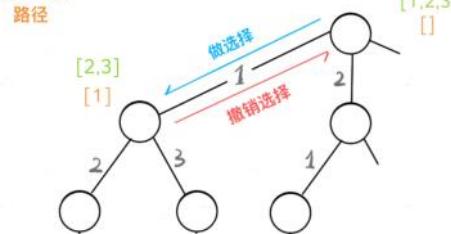
看成指针移动访问每个结点，进入/离开每个结点，可以理解

两个很有用的时间点



前序遍历的代码在进入某一个节点之前的那个时间点执行，
后序遍历代码在离开某个节点之后的那个时间点执行

选择列表



核心框架

```
for 选择 in 选择列表:
    # 做选择
    将该选择从选择列表移除
    路径.add(选择)
    backtrack(路径, 选择列表)
    # 撤销选择
    路径.remove(选择)
    将该选择再加入选择列表
```

我们只要在递归之前做出选择，在递归之后撤销刚才的选择，就能正确得到每个节点的选择列表和路径

动态规划的三个需要明确的点就是「状态」「选择」和「base case」，

对应着走过的「路径」，当前的「选择列表」和「结束条件」

某种程度上说，动态规划的暴力求解阶段就是回溯算法。只是有的问题具有重叠子问题性质，可以用 dp table 或者备忘录优化，将递归树大幅剪枝，这就变成了动态规划。而如果没有重叠子问题，也就是回溯算法问题了，复杂度非常高是不可避免的。

回溯算法就是一个暴力穷举的过程，遍历多叉树的过程，也就是DFS算法

LC-N皇后问题

51. N皇后

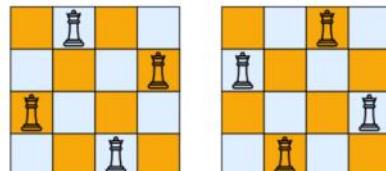
难度 困难 通过 894 提交 200

n皇后问题 研究的是如何将 n 个皇后放在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n ，返回所有不同的 n 皇后问题的解决方案。

每一种解法包含一个不同的 n 皇后问题的棋子放置方案，该方案中 ‘Q’ 和 ‘.’ 分别代表了皇后和空位。

示例 1：



输入：n = 4
输出：[["Q",".",".","Q"],[".","Q",".","Q"],["Q",".",".","."],
[".","Q",".","Q"]]
解释：如上图所示，4 皇后问题存在两个不同的解法。

示例 2：

输入：n = 1
输出：[[“.”]]

```
1 class Solution {
2     public:
3         vector<vector<string>> res;
4         vector<vector<string>> solveNQueens(int n) {
5             vector<string> board(n, string(n, '.'));
6             backtrack(board, 0);
7             return res;
8         }
9         void backtrack(vector<string> &board, int row){
10             if(row == board.size()){
11                 res.push_back(board);
12                 return;
13             }
14             int n = board[row].size();
15             for(int col = 0; col < n; col++){
16                 if(!isValid(board, row, col)){
17                     continue;
18                 //如果[row][col] = 'Q';
19                 board[row][col] = 'Q';
20                 backtrack(board, row + 1);
21                 board[row][col] = '.';
22             }
23         }
24         bool isValid(vector<string> &board, int row, int col){ //检查左上方，上方，右上方是否有Q
25             int n = board.size(); //检查列
26             for(int i = 0; i < n; i++){
27                 if(board[i][col] == 'Q')
28                     return false;
29             } //检查右上方
30             for(int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++){
31                 if(board[i][j] == 'Q')
32                     return false;
33             } //检查左上方
34             for(int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--){
35                 if(board[i][j] == 'Q')
36                     return false;
37             }
38             return true;
39         }
40     }
```

BFS算法框架

BFS的核心思想就是将问题抽象成图，从一个结点开始向四周扩散，也就是齐头并进，

一般用队列数据结构，每次将一个结点周围所有结点加入队列

BFS找到的路径一定是最短的，而DFS要搜索完整个空间，代价就是BFS空间复杂度大，DFS递归好写一些

BFS问题的本质就是在一幅“图”中找起点到终点的最近距离，BFS算法题其实都是在做这件事情

框架

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while(q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 到终点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj())
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
        }
        /* 到终点：更新步数在这里 */
        step++;
    }
}
```

队列 q，BFS 的核心数据结构；

cur.adj() 泛指 cur 相邻的节点，

比如说二维数组中，cur 上下左右四面的位置就是相邻节点；

visited 的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要 visited

二叉树的最大深度

起点就是 root 根节点，终点就是最靠近根节点的那个「叶子节点」，叶子节点就是两个子节点都是 null 的节点

```
while (!q.isEmpty()) {
    int sz = q.size();
    /* 将当前队列中的所有节点向四周扩散 */
    for (int i = 0; i < sz; i++) {
        TreeNode cur = q.poll();
        /* 判断是否到达终点 */
        if (cur.left == null && cur.right == null)
            return depth;
        /* 将 cur 的相邻节点加入队列 */
        if (cur.left != null)
            q.offer(cur.left);
        if (cur.right != null)
            q.offer(cur.right);
    }
    /* 这里增加步数 */
    depth++;
}
```

LC-打开转盘锁

题目描述 评论 (209) 跟贴 (345) 提交记录

752. 打开转盘锁 难度 中等 261

你有一个带四个圆形拨轮的转盘锁。每个拨轮都有10个数字：'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'。每个拨轮可以自由旋转；例如把 '9' 变为 '0'，'0' 变为 '9'。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 "0000"，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将无法被永久锁定，无法再被旋转。

字符串 target 代表可以解锁的数字，你需要给出最小的旋转次数，如果无法转动不能解锁，返回 -1。

示例 1：

```
输入: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出: 6
解释:
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，因为当转动到 "0102" 时这个锁就会被锁定。
```

示例 2：

```
输入: deadends = ["8888"], target = "0009"
输出: 1
解释:
把最后一位反向旋转一次即可 "0000" -> "0009"。
```

示例 3：

```
输入: deadends = ["8887","8889","8878","8898","8788","7888","9888"], target = "8888"
输出: -1
解释:
无法旋转到目标数字且不被锁定。
```

1 class Solution {
2 public:
3 string plus1(string s, int j){
4 if(s[j] == '9')
5 s[j] = '0';
6 else
7 s[j] = s[j] + 1; //s[j] = s[j] + '1' 错了，数字+数字 = ? ? ? 数字+ 数 =数字
8 return s;
9 }
10 string minus1(string s, int j){
11 if(s[j] == '0')
12 s[j] = '9';
13 else
14 s[j] = s[j] - 1;
15 return s;
16 }
17 int openLock(vector<string>& deadends, string target) {
18 string s = "0000";
19 unordered_set<string> visited;
20 for(auto &str : deadends) //将deadend数组元素写到visited内，表示无法继续旋转的串
21 visited.insert(str);
22 if(visited.count(s)) //初始特殊情况
23 return -1;
24 queue<string> q;
25 q.push(s); //初始压入队列
26 int step = 0; //每次首元素表示一个串，每扭一次，得到一次扭得结果，如果没有target，再以这些为基础，再扭一次，并将新结果重新入队
27 while(!q.empty()){
28 auto sz = q.size();
29 for(int i = 0; i < sz; i++){
30 string tmp = q.front();
31 q.pop();
32 if(tmp == target)
33 return step;
34 for(int j = 0; j < 4; j++){
35 s = plus1(tmp, j); //开始处理，比较队列内扭得的元素 是不是target
36 if(!visited.count(s)){
37 visited.insert(s);
38 q.push(s);
39 }
40 s = minus1(tmp, j);
41 if(!visited.count(s)){
42 visited.insert(s);
43 q.push(s);
44 }
45 }
46 }
47 step++;
48 }
49 //return step;
50 return -1; // 没有扭到结果
51 }
}

二分查找框架

寻找一个数、寻找左侧边界、寻找右侧边界

```
1 int binarySearch(int[] nums, int target) {
2     int left = 0, right = ...
3
4     while(...){
5         int mid = left + (right - left) / 2;
6         if (nums[mid] == target) {
7             ...
8         } else if (nums[mid] < target) {
9             left = ...
10        } else if (nums[mid] > target) {
11            right = ...
12        }
13    }
14    return ...
15 }
```

分析二分查找的一个技巧是：不要出现 else，而是把所有情况用 else if 写清楚，这样可以清楚地展现所有细节

寻找一个数（基本的二分搜索）

```

int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1; // 注意

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1; // 注意
        else if (nums[mid] > target)
            right = mid - 1; // 注意
    }
    return -1;
}

```

那么当我们发现索引 mid 不是要找的 target 时，下一步应该去搜索哪里呢？

当然是去搜索 [left, mid-1] 或者 [mid+1, right] 对不对？因为 mid 已经搜索过，应该从搜索区间中去除。

while 循环的条件中是 <= 因为初始化 right 的赋值是 nums.length - 1，即最后一个元素的索引

如果搜索到了区间[0,1]内 mid = 0; 如果 nums[mid] < target 即 nums[0] < target, left = mid + 1 = 1; 那么 target 就是 nums[1] 即相等条件成立 或者 left = mid + 1 = 2 找不到退出
如果 nums[mid] > target, 即 nums[0] > target，则找不到，退出 right = mid - 1 = 0 - 1 = -1 退出

有序数组 nums = [1,2,2,2,3] 如果想得到 target==2 的左侧边界索引 1，可以找到一个 target，然后向左或向右线性搜索，但是难以保证二分查找对数级的复杂度
也可以直接按照这种**搜索左侧边界**的框架改写：

```

1 int left_bound(int[] nums, int target) {
2     int left = 0, right = nums.length - 1;
3     // 搜索区间为 [left, right]
4     while (left <= right) {
5         int mid = left + (right - left) / 2;
6         if (nums[mid] < target) {
7             // 搜索区间变为 [mid+1, right]
8             left = mid + 1;
9         } else if (nums[mid] > target) {
10            // 搜索区间变为 [left, mid-1]
11            right = mid - 1;
12        } else if (nums[mid] == target) {
13            // 收缩右侧边界
14            right = mid - 1;
15        }
16    }
17    // 检查出界情况
18    if (left >= nums.length || nums[left] != target)
19        return -1;
20    return left;
21 }

```

区间【0, 1】 target==2 target ==-1 对应循环结束if 检查语句 这个 num[left] != target 就表明不是由于 == 导致的 right-1，如果相等，表示由于 == 导致的 right-1 越界，
left > right，原来的那个未越界的 left 索引就是对应相等的 target 索引 【0, 1】 left:0 right:1 mid:0 target:0 num[mid] == num[left] == target return left

当 target 比 nums 中所有元素都大时，会让 left = mid + 1 使得索引越界

搜索右侧边界：

```

int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 这里改成收缩右侧边界即可
            left = mid + 1;
        }
    }
    // 这里改为检查 right 越界的情况，见下图
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}

```

滑动窗口算法框架

```

void slideWindow(string s, string t){
    unordered_map<char, int> need, window;
    for(char c : t) need[c]++;
    int left = 0, right = 0;
    int valid = 0;
    while(right < s.size()){
        char c = s[right];
        // 右移窗口
        right++;
        // 对窗口内数据进行一系列更新
        while(wind needs shrink){ // 内层while循环 判断左侧窗口需要收缩
            char d = s[left]; // d是移出窗口的字符
            // 左移窗口
            left++;
            // 对窗口内数据进行一系列更新
        }
    }
}

```

LC-最小覆盖子串

某个字符在 window 的数量满足了 need 的需要，就要更新 valid，表示
有一个字符已经满足了需要，valid == need.size() 说明 t 中所有字符已经

覆盖，得到了一个解，开始收缩窗口优化这个解以寻找最小串即最优解

移动left收缩窗口时，窗口内的解都是可行解，边移动边更新最小串

```
76. 最小覆盖子串
难度 困难 通过率 1100 / 收藏 分享 切换为英文 接收动态 反馈
给你一个字符串 s，一个字符串 t。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在
在 t 中所有字符的子串，则返回空字符串 ""。
注意：如果 s 中存在这样的子串，我们保证它是唯一的答案。
```

示例 1：

```
输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
```

示例 2：

```
输入：s = "a", t = "a"
输出："a"
```

提示：

- 1 <= s.length, t.length <= 10⁵
- s 和 t 由英文字母组成

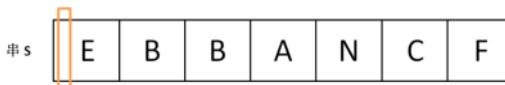
进阶：你能否设计一个在 O(n) 时间内解决此问题的算法吗？

通过次数 142,399 提交次数 342,780

```
2 class Solution {
3 public:
4     string minWindow(string s, string t) {
5         unordered_map<char, int> need, window;
6         for(char c : t) need[c]++;
7         int left = 0, right = 0, len = INT_MAX;
8         int valid = 0, int start = 0;
9         while(right < s.size()){
10             char c = s[right];
11             if(need.count(c)){
12                 window[c]++;
13                 if(window[c] == need[c])
14                     valid++;
15             }
16             right++; // 移动一次就要判断是否形成了覆盖串
17             while(valid == need.size()){
18                 if(right - left < len){
19                     len = right - left;
20                     start = left;
21                 }
22                 char d = s[left];
23                 if(need.count(d)){
24                     // window[d]--;
25                     if(window[d] == need[d]) // 必需要先判断！是否与当前窗内字符对应数量相等 如果相等先令valid-1 下次while循环不再执行
26                         valid--;
27                     window[d]--;
28                     // 将窗内字符 数量不相等减 valid不变 , 相等valid已经减1 窗内字符再减也无所谓
29                 }
30                 left++;
31             }
32         }
33         return len == INT_MAX ? ""
34             : s.substr(start, len);
35     };
36 }
```

初始状态：

窗口区间：[0, 0]



增加 right，直到窗口 [left, right] 包含了 t 中所有字符：

窗口区间：[0, 6)



这里的need与window c++哈希表其实就是python中的字典，相当于一个计数器的作用，保存对应字符的计数

window = {A: 1, B: 2, C: 1}

need = {A: 1, B: 1, C: 1} 这种数据结构就相当于python的字典，C++中的字典实现就是unordered_map与map

```
567. 字符串的排列
难度 中等 通过率 356 / 收藏 分享 反馈
给定两个字符串 s1 和 s2，写一个函数来判断 s2 是否包含 s1 的排列。
```

换句话说，第一个字符串的排列之一是第二个字符串的子串。

示例 1：

```
输入: s1 = "ab" s2 = "eidbaooo"
输出: True
解释: s2 包含 s1 的排列之一 ("ab")。
```

示例 2：

```
输入: s1 = "ab" s2 = "eidboaoo"
输出: False
```

提示：

- 输入的字符串只包含小写字母
- 两个字符串的长度都在 [1, 10,000] 之间

通过次数 87,248 提交次数 206,140

请问您在哪些招聘中遇到此题？

```
3 // 判断 s 中是否存在 t 的排列
4 bool checkInclusion(string t, string s) {
5     unordered_map<char, int> need, window;
6     for (char c : t) need[c]++;
7
8     int left = 0, right = 0;
9     int valid = 0;
10    while (right < s.size()) {
11        char c = s[right];
12        right++;
13        // 进行窗口内数据的一系列更新
14        if (need.count(c)) {
15            window[c]++;
16            if (window[c] == need[c])
17                valid++;
18        }
19        // 判断左窗口是否要收缩
20        while (right - left >= t.size()) { // 收缩时机是长度达到了排列长度/串t的长度
21            // if (right - left >= t.size()) (if也可以，因为排列即s中的包含t中所有字符的子串不能包含其他字符，每次到达排序长度就要判断滑动窗口，而不需要用while一直判断
22            // 在这里判断是否找到了合法的子串
23            if (valid == need.size()) // 如果这个长度内window字典里的字符与need字典相同，则这正好就是一个排列/子串
24                return true;
25            char d = s[left];
26            left++;
27            // 进行窗口内数据的一系列更新
28            if (window.count(d)) {
29                if (window[d] == need[d]) // 如果当前字符数量正好等于need中字符数量，则吃完这个字符，那window中的字符就不等于need中字符了，valid减1，等待下一个字符出现
30                    valid--;
31                window[d]--;
32            }
33        }
34    }
35    // 未找到符合条件的子串
36    return false;
37 }
```

对string 一维数组

正确做法：

```
1 #include<iostream>
2 #include<algorithm>
3 using namespace std;
4 int main()
5 {
6     string s="Qasdaaddj";
7     sort(s.begin(),s.end());
8     cout<<s<<endl;
9     return 0;
10 }
11 
```

string[N]二维数组：

```
1 #include<iostream>
2 #include<algorithm>
3 using namespace std;
4 int main()
5 {
6     string s[10]={"dad","a","adhad"};
7     sort(s,s+3);
8     for(int i=0;i<3;i++)
9         cout<<s[i]<<endl;
10    return 0;
11 } 
```

题目描述 评估 (438) 提交记录 C++ 模拟面试

438. 找到字符串中所有字母异位词
难度 中等 537 收藏 分享

给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。
字符串只包含小写字母，并且字符串 s 和 p 的长度都不超过 1000。
说明：
• 字母异位词指字母相同，但排列不同的字符串。
• 不考虑答案输出的顺序。

示例 1：

输入： s: "cbaebabacd" p: "abc"
输出： [0, 6]
解释：
起始索引等于 0 的子串是 "cba"，它是 "abc" 的字母异位词。
起始索引等于 6 的子串是 "bac"，它是 "abc" 的字母异位词。

示例 2：

```
1 class Solution {
2 public:
3     vector<int> findAnagrams(string s, string p) {
4         vector<int> res;
5         unordered_map<char, int> need, window;
6         int left = 0, right = 0;
7         int valid = 0;
8         for(char c : p) need[c]++;
9         while(right < s.size()){
10             char c = s[right];
11             right++;
12             if(need.count(c)){
13                 window[c]++;
14                 if(window[c] == need[c])
15                     valid++;
16             }
17             while(right - left >= p.size()){ //if也可以
18                 if(valid == need.size())
19                     res.push_back(left); //跟寻找字符串的排列一样，只是找到一个合法异位词（排列）将起始索引加入 res 即可
20                 char d = s[left];
21                 left++;
22                 if(need.count(d)){
23                     if(window[d] == need[d])
24                         valid--;
25                     window[d]--;
26                 }
27             }
28         }
29     }
30 }
```

LC-无重复的最长子串

3. 无重复字符的最长子串
难度 中等 5555 收藏 分享

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1：

输入： s = "abcabcbb"
输出： 3
解释： 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2：

输入： s = "bbbbbb"
输出： 1
解释： 因为无重复字符的最长子串是 "b"，所以其长度为 1。

```
1 class Solution {
2 public:
3     int lengthOfLongestSubstring(string s) {
4         unordered_map<char,int> window;
5         int left = 0, right = 0;
6         int res = 0;
7         while(right < s.size()){
8             char c = s[right];
9             right++;
10            window[c]++;
11            while(window[c] > 1){
12                char d = s[left];
13                window[d]--;
14                left++;
15            }
16            res = max(res,right - left + 1); //不含重复字符的最长子串长度 每次滑动窗口都要计算一个长度更新
17        }
18    }
19 }
```

窗口收缩的 while 条件是重复元素 当 window[c] 值大于 1 时，说明窗口中存在重复字符，不符合条件，就该移动 left 缩小窗口
注意更新结果 res

-----next_permutation(begin(), end())函数

对于next_permutation函数，其函数原型为：

```
#include <algorithm>
bool next_permutation(iterator start,iterator end)
```

当当前序列不存在下一个排列时，函数返回false，否则返回true

```
sort
do{
    xx
}while(next_permutation(vec.begin(), vec.end()));
```

```

1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4 int main()
5 {
6     int num[3]={1,2,3};
7     do
8     {
9         cout<<num[0]<<" "<<num[1]<<" "<<num[2]<<endl;
10    }while(next_permutation(num,num+3));
11    return 0;
12 }

```

输出结果为：

```

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

```

当我们把while(next_permutation(num,num+3))中的3改为2时，输出就变为了：

```

1 2 3
2 1 3
Process returned 0 (0x0) execution time: 0.0198 s
Press any key to continue.

```

由此可以看出，next_permutation(num,num+n)函数是对数组num中的前n个元素进行全排列，同时并改变num数组的值。

另外，需要强调的是，next_permutation（）在使用前需要对欲排列数组按升序排序，否则只能找出该序列之后的全排列数。比如，如果数组num初始化为2,3,1，那么输出就变为了：

```

2 3 1
3 1 2
3 2 1

```

LC-股票问题

状态机也就是一个文学词汇，实际上就是dp table 状态转移方程

穷举框架：

```

1 for 状态1 in 状态1的所有取值：
2     for 状态2 in 状态2的所有取值：
3         for ...
4             dp[状态1][状态2][...] = 择优(选择1, 选择2...)

```

前面是利用递归进行穷举，一步一步推进，这里是利用状态进行穷举

穷举的目的是根据对应的选择更新状态

三个状态参数：天数，交易次数，当前持有状态 1.表示持有 0表示未持有

```

1 dp[i][k][0 or 1]
2 0 <= i <= n-1, 1 <= k <= K
3 n 为天数，大 K 为最多交易数
4 此问题共 n × K × 2 种状态，全部穷举就能搞定。
5
6 for 0 <= i < n:
7     for 1 <= k <= K:
8         for s in {0, 1}:
9             dp[i][k][s] = max(buy, sell, rest)

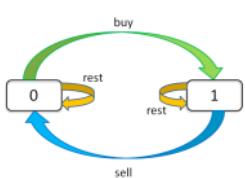
```

dp[3][2][1]：今天是第三天，手上持有股票，最多还可以进行2次交易

dp[2][3][0]：今天是第二天，手上没有股票，最多还可以进行3次交易

想求的答案是dp[n-1][k][0]从第0天开始的，即最后一天，最多进行K次交易，手上没有股票，已经卖出的利润

只要「持有状态」，可以画个状态转换图。



```

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
            max( 选择 rest ,          选择 sell   )

```

解释：今天我没有持有股票，有两种可能：

要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；

要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。

```

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
            max( 选择 rest ,          选择 buy   )

```

解释：今天我持有股票，有两种可能：

要么我昨天就持有股票，然后今天选择 rest，所以我今天还持有股票；

要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。

如果 buy，就要从利润中减去 prices[i]，如果 sell，就要给利润增加 prices[i]。今天的最大利润就是这两种选择中较大的那个。

而且注意 k 的限制，我们在选择 buy 的时候，把 k 减小了 1，当然也可以在 sell 的时候减 1，一样的

```

1 base case :
2 dp[-1][k][0] = dp[i][0][0] = 0
3 dp[-1][k][1] = dp[i][0][1] = -infinity
4
5 状态转移方程：
6 dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
7 dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

```

第一题， $k=1$

直接套状态转移方程，根据 base case，可以做一些化简：

```

1 dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
2 dp[i][1][1] = max(dp[i-1][1][1], dp[i-1][0][0] - prices[i])
3           = max(dp[i-1][1][1], -prices[i])
4 解释：k = 0 的 base case，所以 dp[i-1][0][0] = 0。
5
6 现在发现 k 都是 1，不会改变，即 k 对状态转移已经没有影响了。
7 可以进行进一步化简去掉所有 k：
8 dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
9 dp[i][1] = max(dp[i-1][1], -prices[i])

```

```

for (int i = 0; i < n; i++) {
    if (i - 1 == -1) {
        dp[i][0] = 0;
        // 解释：
        // dp[i][0]
        // = max(dp[-1][0], dp[-1][1] + prices[i])
        // = max(0, -infinity + prices[i]) = 0
        dp[i][1] = -prices[i];
        // 解释：
        // dp[i][1]
        // = max(dp[-1][1], dp[-1][0] - prices[i])
        // = max(-infinity, 0 - prices[i])
        // = -prices[i]
        continue;
    }
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];

```

但是这样处理 base case 很麻烦，而且注意一下状态转移方程，新状态只和相邻的一个状态有关，其实不用整个 dp 数组，只需要一个变量储存相邻的那个状态就足够了，这样可以把空间复杂度降到 O(1)：

```

121.买卖股票的最佳时机
难度 简单 1652 收藏 384
给定一个数组 prices，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。
你只能选择某一天买入这只股票，并选择在未来的某一个不同的日子卖出该股票。设计一个算法来计算你所能获取的最大利润。
返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0。

```

```

1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices) {
4         int n = prices.size();
5         int dp_i_0 = 0, dp_i_1 = INT_MIN; //本来未持有 本来就持有
6         for(int i = 0; i < n; i++){
7             dp_i_0 = max(dp_i_0, dp_i_1 + prices[i]); //从n=0这天开始，0表示未持有 的利润
8             dp_i_1 = max(dp_i_0, -prices[i]); //当天持有 的利润 前一天就持有rest，前一天未持有 //k-1，如果前一天未持有，说明它是今天买的
9             // 保存相邻的前一天状态
10        }
11     return dp_i_0; //7 1 5 3 6 4 0 5 6
12 } //最大 6 dp_0当天未持有 dp_0当天持有

```

示例 1：

输入：[7,1,5,3,6,4]
输出：
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5。
注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格；
同时，你不能在买入前卖出股票。

$k = +\infty$

如果 k 为正无穷，那么就可以认为 k 和 $k-1$ 是一样的。可以这样改写框架：

```

1 dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
2 dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
3           = max(dp[i-1][k][1], dp[i-1][k][0] - prices[i])
4
5 我们发现数组中的 k 已经不会改变了，也就是说不需要记录 k 这个状态：
6 dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
7 dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])

```

左边

当天未持有股票已经获得的利润 dp_i_0

当天持有股票已经获得的利润 dp_i_1

右边：前一天/本来 未持有 与 前一天/本来就持有

122. 买卖股票的最佳时机 II

难度 简单 | 1233 | 收藏 | 分享

给定一个数组 prices ，其中 $\text{prices}[i]$ 是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

```

1 class Solution {
2 public:
3     int maxProfit(vector<int>& prices) {
4         int n = prices.size();
5         int dp_i_0 = 0, dp_i_1 = INT_MIN; // 本来未持有 本来就持有
6         for(int i = 0; i < n; i++){
7             int tmp = dp_i_0;
8             dp_i_0 = max(dp_i_0, dp_i_1 + prices[i]); // 从n=0这天开始，0表示未持有 的利润
9             dp_i_1 = max(dp_i_1, tmp - prices[i]); // 当天持有 的利润 前一天就持有rest，前一天未持有
10        } // 加上之前未持有时已经有利润 // 前一天未持有已经获得的利润
11    }
12 }
13 
```

 $k = +\infty$ with cooldown

每次 sell 之后要等一天才能继续交易。只要把这个特点融入上一题的状态转移方程即可：

```

1 dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
2 dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
3 解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1。

```

309. 最佳买卖股票时机冷冻期

难度 中等 | 784 | 收藏 | 分享

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票（即冷冻期为 1 天）。

示例：

输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]

 $k = +\infty$ with fee

每次交易要支付手续费，只要把手续费从利润中减去即可。改写方程：

```

1 dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
2 dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)
3 解释：相当于买入股票的价格升高了。
4 在第一个式子里减也是一样的，相当于卖出股票的价格减小了。

```

714. 买卖股票的最佳时机扣手续费

难度 中等 | 490 | 收藏 | 分享

给定一个整数数组 prices ，其中第 i 个元素代表了第 i 天的股票价格；非负整数 fee 代表了交易股票的手续费。

你可以无数次地完成交易，但是每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易只需要为支付一次手续费。

 $k = 2$

$k = 2$ 和前面题目的情况稍微不同，因为上面的情况都和 k 的关系不太大，这道题由于没有消掉 k 的影响，所以必须要对 k 进行穷举：

```

1 int max_k = 2;
2 int[][] dp = new int[n][max_k + 1][2];
3 for (int i = 0; i < n; i++) {
4     for (int k = max_k; k >= 1; k--) {
5         if (i - 1 == -1) { /*处理 base case */}
6         dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
7         dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
8     }
9 }
10 // 穷举了  $n \times \max_k \times 2$  个状态，正确。
11 return dp[n - 1][max_k][0];

```

这里 k 取值范围比较小，所以可以不用 for 循环，直接把 $k = 1$ 和 2 的情况全部列举出来也可以

```

1 dp[i][2][0] = max(dp[i-1][2][0], dp[i-1][2][1] + prices[i])
2 dp[i][2][1] = max(dp[i-1][2][1], dp[i-1][1][0] - prices[i])
3 dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
4 dp[i][1][1] = max(dp[i-1][1][1], -prices[i])
5
6 int maxProfit_k_2(int[] prices) {
7     int dp_i10 = 0, dp_i11 = Integer.MIN_VALUE;
8     int dp_i20 = 0, dp_i21 = Integer.MIN_VALUE;
9     for (int price : prices) {
10         dp_i20 = Math.max(dp_i20, dp_i21 + price);
11         dp_i21 = Math.max(dp_i21, dp_i10 - price);
12         dp_i10 = Math.max(dp_i10, dp_i11 + price);
13         dp_i11 = Math.max(dp_i11, -price);
14     }
15     return dp_i20;
16 }

```

```

123. 买卖股票的最佳时机 III
难度 困难 781 收藏 46 举报
这是一个数组，它的第 1 个元素是一支给定的股票在第 1 天的价格。
设计一个算法来计算你所能获得的最大利润。你最多可以完成 两次 交易。
注意：你不能同时参与多笔交易（你必须在再次买入前出售掉之前的股票）。
示例 1：
输入：prices = [3,3,5,0,0,3,1,4]
输出：6
解释：在第 4 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出，这笔交易所获得利润 = 3-0 = 3。
随后，在第 7 天 (股票价格 = 1) 的时候买入，在第 8 天 (股票价格 = 4) 的时候卖出，这笔交易所获得利润 = 4-1 = 3

```

有状态转移方程和含义明确的变量名指导，相信你很容易看懂。其实我们可以故弄玄虚，把上述四个变量换成 a, b, c, d。这样当别人看到你的代码时就会大惊失色，对你肃然起敬.....

$k = \text{any integer}$

一次交易由买入和卖出构成，至少需要两天。所以说有效的限制 k 应该不超过 $n/2$ 如果超过，就没有约束作用了，相当于 $k = +\infty$ 。这种情况是之前解决过的问题

```

1 int maxProfit_k_any(int max_k, int[] prices) {
2     int n = prices.length;
3     if (max_k > n / 2)
4         return maxProfit_k_inf(prices);
5
6     int[][][] dp = new int[n][max_k + 1][2];
7     for (int i = 0; i < n; i++)
8         for (int k = max_k; k >= 1; k--) {
9             if (i - 1 == -1) { /* 处理 base case */
10                 dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
11                 dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
12             }
13         }
14     return dp[n - 1][max_k][0];
15 }

```

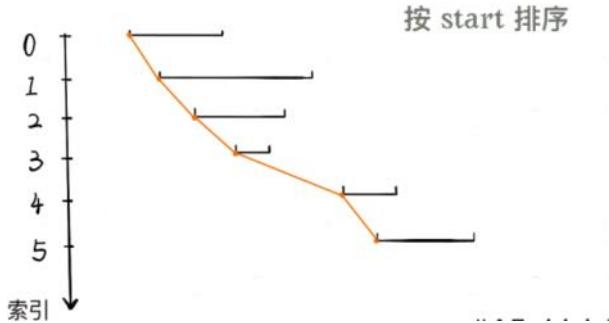
重点在于 dp 表示买入卖出股票得到的利润，状态转移方程为前一天得到的利润状态转移方程得到当天的利润

LC-区间问题

也就是线段问题，合并所有线段、找出线段交集

排序+画图

一般按照区间起点排序，若起点相同，则按照终点排序

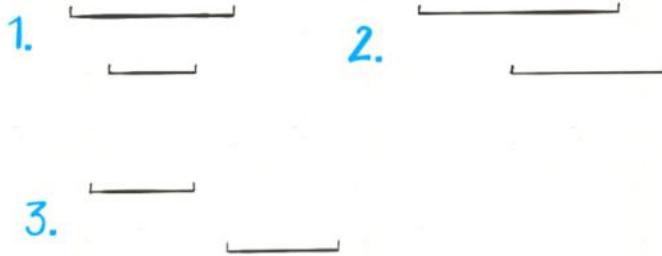


排序，两个相邻区间有三种相对位置，

对于情况一，找到了覆盖区间。

对于情况二，两个区间可以合并，成一个大区间。

对于情况三，两个区间完全不相交。



对于第三种情况需要更新 $left$ 和 $right$ 边界，因为是排序好了的区间数组，

接下来的区间起点都会从这个新的 $left$ 开始一些区间

1288. 删除被覆盖区间
难度 中等 ⚡ 45 ☆ 🔍

给你一个区间列表，请你删除列表中被其他区间所覆盖的区间。
只有当 $c \leq a$ 且 $b \leq d$ 时，我们认为区间 $[a, b]$ 被区间 $[c, d]$ 覆盖。
在完成所有删除操作后，请你返回列表中剩余区间的数目。

示例：

输入：intervals = [[1,4],[3,6],[2,8]]
输出：2
解释：区间 [3,6] 被区间 [2,8] 覆盖，所以它被删除了。

提示：
 • $1 \leq \text{intervals.length} \leq 1000$
 • $0 \leq \text{intervals}[i][0] < \text{intervals}[i][1] \leq 10^5$
 • 对于所有的 $i \neq j$: $\text{intervals}[i] \neq \text{intervals}[j]$

56. 合并区间
难度 中等 ⚡ 956 ☆ 🔍

以数组 intervals 表示若干个区间的集合，其中单个区间为 $\text{intervals}[i] = [\text{start}_i, \text{end}_i]$ ，请你合并所有重叠的区间，并返回一个不重叠的区间列表。该数组可能会包含输入中的所有区间。

示例 1：

输入：intervals = [[1,3],[2,6],[0,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6]。

示例 2：

输入：intervals = [[1,4],[4,5]]
输出：[[1,5]]
解释：区间 [1,4] 和 [4,5] 可被视为重叠区间。

966. 区间列表的交集
难度 中等 ⚡ 151 ☆ 🔍

给定两个由非空区间组成的列表，firstList 和 secondList，其中 firstList[i] = [start_i, end_i] 而 secondList[j] = [start_j, end_j]。每个区间都是成对不重叠的，并且已经排序。

求两个区间列表的交集。

形式上，如果 $[e_i, s_i]$ (其中 $e < s$) 是第一个列表 firstList 中的一个区间，而 $[e_j, s_j]$ (其中 $e < s$) 是第二个列表 secondList 中的一个区间，则 $e_i \leq e_j$ 且 $s_i \geq s_j$ 。两个区间的交集是一段实数，要么为空集，要么为区间。例如， $[1, 4]$ 和 $[3, 6]$ 的交集为 $[3, 4]$ 。

示例 1：

A:

1	1	1	2	2	3	3
---	---	---	---	---	---	---

B:

1	1	1	2	2	3	3
---	---	---	---	---	---	---

ans:

1	1	1	2	2	3	3
---	---	---	---	---	---	---

B:

0	4	8	12	16	20	24
---	---	---	----	----	----	----

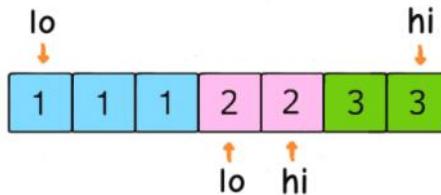
输入：firstList = [[0,1],[1,3],[2,6],[0,10],[15,18]], secondList = [[1,5],[4,5],[1,10],[2,12],[15,25]]
输出：[[1,5],[4,5],[1,10],[2,12],[15,25]]

nsum问题
两数之和
没有重复元素 排序+双指针

```
vector<int> twoSum( vector<int>& nums, int target){  
    sort(nums.begin(), nums.end());  
    int lo = 0, hi = nums.size() - 1;  
    while(lo < hi){  
        int sum = nums[lo] + nums[hi];  
        if(sum < target) lo++;  
        else if(sum > target) hi--;  
        else if(sum == target) return {lo, hi};  
    }  
    return {};
```

比如输入 $\text{nums} = [1,3,5,6]$, $\text{target} = 9$ ，那么算法返回两个元素 $[3,6]$ 。

如果 nums 中有多对元素之和等于 target ，需要返回所有和为 target 的元素对且不能重复
基本思路还是 排序 + 双指针



```

1 while (lo < hi) {
2     int sum = nums[lo] + nums[hi];
3     // 记录索引 lo 和 hi 最初对应的值
4     int left = nums[lo], right = nums[hi];
5     if (sum < target) lo++;
6     else if (sum > target) hi--;
7     else {
8         res.push_back({left, right});
9         // 跳过所有重复的元素
10        while (lo < hi && nums[lo] == left) lo++;
11        while (lo < hi && nums[hi] == right) hi--;
12    }
13 }

```

这样就可以保证一个答案只被添加一次，重复的结果都会被跳过，可以得到正确的答案。不过，受这个思路的启发，

其实前两个 if 分支也是可以做一点效率优化，**跳过相同的元素**

```

vector<vector<int>> twoSumTarget(vector<int>& nums, int target) {
    // nums 数组必须有序
    sort(nums.begin(), nums.end());
    int lo = 0, hi = nums.size() - 1;
    vector<vector<int>> res;
    while (lo < hi) {
        int sum = nums[lo] + nums[hi];
        int left = nums[lo], right = nums[hi];
        if (sum < target) {
            while (lo < hi && nums[lo] == left) lo++;
        } else if (sum > target) {
            while (lo < hi && nums[hi] == right) hi--;
        } else {
            res.push_back({left, right});
            while (lo < hi && nums[lo] == left) lo++;
            while (lo < hi && nums[hi] == right) hi--;
        }
    }
    return res;
}

```

双指针操作的部分虽然有那么多 while 循环，但是时间复杂度还是 O(N)，

而排序的时间复杂度是 O(NlogN)，所以这个函数的时间复杂度是 O(NlogN)

三数之和，穷举第一个数，且保证穷举的第一个数不同，剩下的target-nums[i]

就是两数之和 关键点在于，不能让第一个数重复，至于剩余的两个数复用的 **twoSum** 函数会保证它们不重复

所以代码中必须用一个 while 循环来保证 **3Sum** 中第一个元素不重复

LC-三数之和

15. 三数之和
难度 中等 ⚡ 3392 ⚡ ⚡ ⚡ ⚡ ⚡

给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a, b, c，使得 a + b + c = 0？请找出所有和为 0 且不重复的三元组。

注意：答案集中不可以包含重复的三元组。

示例 1：

输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]

示例 2：

输入：nums = []
输出：[]

示例 3：

输入：nums = [0]
输出：[]

提示：

- $0 \leq \text{nums.length} \leq 3000$
- $-10^5 \leq \text{nums}[i] \leq 10^5$

通过次数 526,906 | 提交次数 1,632,990

```

1 class Solution {
2 public:
3     vector<vector<int>> twoSum(vector<int>& nums, int start, int target){
4         /// 左侧改为从 start 开始，其他不变
5         int lo = start, hi = nums.size() - 1; //末尾元素的索引是 size - 1 !!!
6         vector<vector<int>> res;
7         while(lo < hi){
8             int sum = nums[lo] + nums[hi];
9             int left = nums[lo], right = nums[hi];
10            if(sum < target) lo++;
11            else if(sum > target) hi--;
12            else {
13                res.push_back({left, right});
14                while(lo < hi && nums[lo] == left) lo++;
15                while(lo < hi && nums[hi] == right) hi--;
16            }
17        }
18        return res;
19    }
20    vector<vector<int>> threeSum(vector<int>& nums) {
21        int target = 0;
22        sort(nums.begin(), nums.end());
23        vector<vector<int>> res;
24        //穷举第一个数
25        for(int i = 0; i < nums.size(); i++){
26            vector<vector<int>> ans = twoSum(nums, i+1, target-nums[i]);
27            //如果ans有结果，即有满足条件的二元组，则再加nums[i]就是三元组
28            for(vector<int> &tuple : ans){
29                tuple.push_back(nums[i]);
30                res.push_back(tuple);
31            }
32        }
33        //用一个while循环跳过第一个数字重复的情况，否则会出现重复结果 这里相当于i+1再+1 跳过下一个数
34        while(i < nums.size()-1 && nums[i] == nums[i+1]) i++; //执行完一次for i也要无条件加1
35    }
36 }

```

4 Sum 问题

穷举第一个数字，然后调用 **3Sum** 函数计算剩下三个数，最后组合出和为 **target** 的四元组

18. 四数之和
难度 中等 ⚡ 665 ⚡ ⚡ ⚡ ⚡ ⚡

给你一个包含 n 个整数的数组 nums，和一个目标值 target，判断 nums 中是否有四个元素 a, b, c, d，使得 $a + b + c + d = \text{target}$ 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：答案集中不可以包含重复的四元组。

示例 1：

输入：nums = [1,0,-1,0,-2,2], target = 0
输出：[[-2,-1,1,2],[-1,0,0,2],[-1,0,0,1]]

示例 2：

输入：nums = [], target = 0
输出：[]

提示：

- $0 \leq \text{nums.length} \leq 3000$
- $-10^5 \leq \text{nums}[i] \leq 10^5$
- $-10^9 \leq \text{target} \leq 10^9$

```

1 class Solution {
2 public:
3     vector<vector<int>> twoSum(vector<int>& nums, int start, int target){
4         /// 从 start 开始，其他的不变
5         int lo = start, hi = nums.size() - 1; //末尾元素的索引是 size - 1 !!!
6         vector<vector<int>> res;
7         while(lo < hi){
8             int sum = nums[lo] + nums[hi];
9             int left = nums[lo], right = nums[hi];
10            if(sum < target) lo++;
11            else if(sum > target) hi--;
12            else {
13                res.push_back({left, right});
14                while(lo < hi && nums[lo] == left) lo++;
15                while(lo < hi && nums[hi] == right) hi--;
16            }
17        }
18        return res;
19    }
20    vector<vector<int>> threeSum(vector<int>& nums, int start, int target) {
21        // * 从 nums[start] 开始，计算剩下的数组 nums 中所有和为 target 的三元组 */
22        vector<vector<int>> res; // 1 从 start 开始的带，其他的都不变
23        for(int i = start; i < nums.size(); i++){
24            vector<vector<int>> ans = twoSum(nums, i+1, target-nums[i]);
25            for(vector<int> &tuple : ans){
26                tuple.push_back(nums[i]);
27                res.push_back(tuple);
28            }
29        }
30        //用一个while循环跳过第一个数字重复的情况，否则会出现重复结果 这里相当于i+1再+1 跳过下一个数
31        while(i < nums.size()-1 && nums[i] == nums[i+1]) i++; //执行完一次for i也要无条件加1
32    }
33    vector<vector<int>> fourSum(vector<int>& nums, int target) {
34        int start = 0;
35        vector<vector<int>> res;
36        for(int i = start; i < nums.size(); i++){
37            vector<vector<int>> ans = threeSum(nums, i+1, target-nums[i]);
38            for(vector<int> &tuple : ans){
39                tuple.push_back(nums[i]);
40                res.push_back(tuple);
41            }
42        }
43        return res;
44    }

```

```

输入: nums = [], target = 0
输出: []

提示：
* 0 <= nums.length <= 200
* -109 <= nums[i] <= 109
* -109 <= target <= 109

通过次数 185,059 提交次数 457,751

询问您在精英指南中遇到此题？

社区 标签 实习 未通过
力扣 (LeetCode) 版面所有

```

nSum 函数

```

/* 注意：调用这个函数之前一定要先给 nums 排序 */
vector<vector<int>> nsumTarget(
    vector<int>& nums, int n, int start, int target) {
    int sz = nums.size();
    vector<vector<int>> res;
    // 至少是 2sum，且数组大小不能小于 n
    if (n < 2 || sz < n) return res;
    // 2sum 是 base case
    if (n == 2) {
        // 双指针那一套操作
        int lo = start, hi = sz - 1;
        while (lo < hi) {
            int sum = nums[lo] + nums[hi];
            int left = nums[lo], right = nums[hi];
            if (sum < target) {
                while (lo < hi && nums[lo] == left) lo++;
            } else if (sum > target) {
                while (lo < hi && nums[hi] == right) hi--;
            } else {
                res.push_back({left, right});
                while (lo < hi && nums[lo] == left) lo++;
                while (lo < hi && nums[hi] == right) hi--;
            }
        }
    } else {
        // n > 2 时，递归计算 (n-1)sum 的结果
        for (int i = start; i < sz; i++) {
            vector<vector<int>> sub;
            sub = nsumTarget(nums, n - 1, i + 1, target - nums[i]);
            for (vector<int>& arr : sub) {
                // (n-1)sum 加上 nums[i] 就是 nSum
                arr.push_back(nums[i]);
                res.push_back(arr);
            }
        }
    }
    return res;
}

```

比如说现在我们写 LeetCode 上的 **4Sum** 问题：

```

vector<vector<int>> fourSum(vector<int>& nums, int target) {
    sort(nums.begin(), nums.end());
    // n > 4，从 nums[0] 开始计算和为 target 的四元组
    return nsumTarget(nums, 4, 0, target);
}

```

再比如 LeetCode 的 **3Sum** 问题，找 **target == 0** 的三元组：

```

vector<vector<int>> threeSum(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    // n > 3，从 nums[0] 开始计算和为 0 的三元组
    return nsumTarget(nums, 3, 0, 0);
}

```

实际上就是把之前的题目解法合并起来了，**n == 2** 时是 **twoSum** 的双指针解法

, n > 2 时就是穷举第一个数字，然后递归调用计算 **(n-1)Sum**，组装答案

需要注意的是，调用这个 **nSum** 函数之前一定要先给 **nums** 数组排序，因为 **nSum** 是一个递归函数，如果在 **nSum** 函数里调用排序函数，那么每次递归都会进行没有必要的排序，效率会非常低

二叉树

遍历框架：

```

void traverse(TreeNode* root){
    //前序遍历
    traverse(root-> left);
    //中序遍历
    traverse(root -> right);
    //后序遍历
}

```

二叉树是最练习递归代码基本能力的题

如快速排序就是个二叉树的前序遍历，归并排序就是个二叉树的后序遍历

因为算法框架和代码逻辑几乎相同

快速排序逻辑：

若要对 **nums[lo..hi]** 排序，先找一个分界点 **P**，通过交换元素使得

nums[lo..p-1] 都小于 **nums[p]** **nums[p+1...hi]** 都大于 **nums[p]**

再递归的到 **nums[lo..p-1]** 与 **nums[p+1...hi]** 中寻找新的分界点

最终整个数组就排序了

```

void sort(int[] nums, int lo, int hi){
    //前序遍历位置
    //通过交换元素构建分界点p
    int p = partition(nums, lo, hi);
    sort(nums, lo, p-1);
    sort(nums, p+1, hi);
}

```

这种先构造分界点，再接着到左右子数组构造分界点，其实就相当于一个二叉树的前序遍历

归并排序逻辑：

若要对 $\text{nums}[\text{lo...hi}]$ 排序，先对 $\text{nums}[\text{lo...mid}]$ 排序，再对 $\text{nums}[\text{mid+1...hi}]$ 排序
最后再将这两个有序的子数组合并，整个数组就排序好了

```
void sort(int[] nums, int lo, int hi){  
    int mid = (lo + hi)/2;  
    sort(nums,lo,mid);  
    sort(nums, mid+1, hi);  
    //后序遍历位置  
    //合并两个排序好的子数组  
    merge(nums, lo, mid, hi);
```

这种先对左右子数组排序再合并，其实就相当于二叉树的后序遍历框架

分治算法也是如此，只要涉及递归都可以抽象成二叉树的问题

写递归算法的秘密

写递归算法的关键就是要 明确函数的定义是 什么

再相信这个定义，利用这个定义推导出最终结果 绝对还要跳出递归的细节

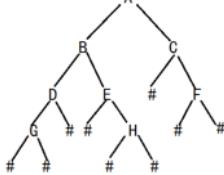
计算二叉树的结点个数

定义：count (root) 返回以root为根的树结点个数

```
int count ( TreeNode* root{  
    if(root == null) return 0;  
    return 1 + count(root->left) + count(root->right);  
}
```

count 函数就是用来算结点个数 利用count函数遍历所有结点每一个结点不为空，则返回值加1

生成二叉树



下面的代码实现中没有使用二叉树，而是使用数组来存储二叉树的结点，从而使得操作更方便。
生成二叉树的代码如下：

```
1 //二叉树结点，存储节点信息，需要的二叉树类，以及空指针  
2 #include <iostream>  
3 #include <vector>  
4  
5 //结点类  
6 class TreeNode {  
7 public:  
8     char data;  
9     TreeNode *left;  
10    TreeNode *right;  
11};  
12  
13 void generateTree(TreeNode* &root, std::vector<char> &inorder, std::vector<char> &preorder) {  
14     if(inorder.size() == 0) {  
15         return;  
16     }  
17     root = new TreeNode(inorder[0]);  
18     int index = -1;  
19     for(int i = 0; i < inorder.size(); i++) {  
20         if(inorder[i] == preorder[0]) {  
21             index = i;  
22         }  
23     }  
24     if(index != -1) {  
25         root->left = generateTree(root->left, inorder.begin() + 1, preorder.begin() + 1);  
26         root->right = generateTree(root->right, inorder.begin() + index + 1, preorder.begin() + index + 1);  
27     }  
28 }
```

写树相关的算法，先要明确当前root结点该做什么，再根据函数定义

递归调用子结点，递归调用会让孩子结点做同样的事情

226. 翻转二叉树

难度：简单 | 收藏 875 | 分享 | 难易度：简单

编辑一稿二叉树。

示例：

输入：

```
4  
 / \  
 2   7  
 / \ / \  
 1 3 6 9
```

输出：

```
4  
 / \  
 7   2  
 / \ / \  
 9 6 3 1
```

```
1 /**  
2  * Definition for a binary tree node.  
3  */  
4 struct TreeNode {  
5     int val;  
6     TreeNode *left;  
7     TreeNode *right;  
8 };  
9  
10 TreeNode(): val(0), left(nullptr), right(nullptr) {}  
11  
12 TreeNode(int x): val(x), left(nullptr), right(nullptr) {}  
13  
14 TreeNode(int x, TreeNode *left, TreeNode *right): val(x), left(left), right(right) {}  
15  
16  
17 class Solution {  
18 public: //只需要把二叉树上的每一个节点的左右子节点进行交换，最后的结果就是完全翻转之后的二叉树。  
19     TreeNode* invertTree(TreeNode* root) {  
20         if(root == nullptr) // // base case  
21             return root;  
22         // **** 前序遍历位置 ****// root 节点需要交换它的左右子节点  
23         swap(root->left, root->right); //交换左右子结点  
24         //让左右子节点继续翻转它们的子节点  
25         invertTree(root->left); //先序 遍历左子树  
26         invertTree(root->right); //遍历右子树  
27         return root;  
28     }  
29 }
```

如果将交换左右子结点的代码放到后序遍历的位置也是可以的，但是放到中序遍历的位置就不行

116. 填充每个节点的下一个右侧节点指针

难度 中等 □ 469 ☆ □ ★ ★ ★

给定一个完美二叉树，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。

初始状态下，所有 next 指针都被设置为 NULL。

进阶：

- 你只能使用常量级额外空间。
- 使用递归解题也符合要求，本题中递归程序占用的栈空间不计做额外的空间复杂度。

只依赖一个节点的话，肯定是没办法连接「跨父节点」的两个相邻节点的。

那么，我们的做法就是增加函数参数，一个节点做不到，我们就给他安排两个节点，「将每一层二叉树节点连接起来」可以细化成「将每两个相邻节点都连接起来」。

这样，connectTwo 函数不断递归，可以遍历到整棵二叉树所有相邻结点，将所有相邻节点都连接起来，也就避免了我们之前出现的问题，这道题就解决了。

connectTwo 函数的作用就是将 node1 结点的指针指向 node2，这个函数将不断的递归执行，直到遇到一个叶子结点将 return，终止递归，不再继续执行该函数。

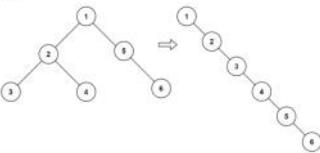
114. 二叉树展开为链表

难度 中等 □ 616 ☆ □ ★ ★ ★

给定二叉树的根结点 root，请将其展开为一个单链表：

- 展开前的单链表应同样使用 TreeNode，其中 right 子指针指向链表中下一个结点，而左子节点则为 null。
- 展开后的单链表应该与二叉树 为同构，能够访问。

示例 1：



输入：root = [1,2,5,3,4,null,6]

输出：[1,null,2,null,3,null,4,null,5,null,6]

示例 2：

输入：root = []

输出：[]

这就是利用遍历框架，遍历就是结点访问，利用结点访问的顺序可以在访问结点的同时对结点进行操作，也就是在遍历框架加入结点处理部分的代码。

根据遍历访问的顺序处理结点内容

递归算法的关键要明确函数的定义，相信这个定义，而不要跳进递归细节

写二叉树的算法题，都是基于递归框架的，我们先要搞清楚 root 节点它自己要做什么，根据题目要求选择使用前序，中序，后续的递归框架。

经典动态规划：子集背包问题-分割等和子集

背包问题：载重为W的背包和N个物品，每个物品有重量和价值两个属性w[i],val[i]

用这个背包装能装下的最大价值

对于分割等和子集：给一个载重为sum/2的背包和N个物品，每个物品重量为nums[i]

是否有一种装法 可以将背包装满

第一步：明确状态和选择

状态就是背包的容量和可以选择的物品

选择就是装入该物品和不装入该物品

第二步：明确dp数组的定义

dp[i][j] = x 表示对于前i个物品，当前背包容量为j，x为true表示可以装满

x为false表示不能恰好将背包装满

如dp[4][9] = true 表示对于容量为9的背包，若只用前4个物品，则可以有一种方法恰好装满

我们需要的答案就是dp[N][sum/2]

base case就是dp[...][0] = true, dp[0][...]=false

背包没有空间相当于装满了，没有物品可选必然不能装满背包

第三步：根据选择，进行状态转移

如果不将nums[i]算入子集，也就是不把这个第i个物品装入背包，那能否装满

取决于上一个状态，继承之前的结果

如果将nums[i]算入子集，也就是将这个第i个物品装入背包，那能否装满取决于

状态dp[i-1][j-nums[i]]

LC-分割等和子集

```
3 class Node {
4 public:
5     int val;
6     Node* left;
7     Node* right;
8     Node* next;
9
10    Node() : val(0), left(NULL), right(NULL), next(NULL) {}
11
12    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
13
14    Node(int _val, Node* _left, Node* _right, Node* _next)
15    : val(_val), left(_left), right(_right), next(_next) {}
16 }
17 */
18 class Solution {
19 public:
20     Node* connect(Node* root) {
21         if(root == NULL) return NULL;
22         connectTwo(root->left, root->right);
23         return root;
24     }
25     void connectTwo(Node* node1, Node* node2) {
26         if(node1 == NULL || node2 == NULL) return;
27         node1->next = node2;
28         connectTwo(node1->left, node1->right);
29         connectTwo(node2->left, node2->right);
30         connectTwo(node1->right, node2->left);
31     }
32 }
```

```
1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode *next;
9     TreeNode() : val(0), left(nullptr), right(nullptr) {}
10    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
11    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
12 }
13
14 class Solution {
15 public:
16     void flatten(TreeNode* root) { // 1 2 3 4 5
17         if (root == nullptr) return; // 1 2 3 4 5
18         flatten(root->left); // 2 3 4 5
19         if (root->left != nullptr) { // 3 4 5
20             auto pre = root->left; // 左子树的最后一个结点，即右子树的最左结点
21             while (pre->right != nullptr) pre = pre->right; // 找到左子树的最右结点
22             pre->right = root->right; // 左子树的最右结点接到右子树的最左
23             root->right = root->left; // 左子树的最右结点接到右子树的最左
24             root->left = nullptr; // 左子树的最右结点接到右子树的最左
25         }
26         // 2 3 4 5
27         // 3 4 5
28         // 4 5
29         // 5
30     }
31 }
```

这是利用遍历框架，遍历就是结点访问，利用结点访问的顺序可以在访问结点的同时对结点进行操作，也就是在遍历框架加入结点处理部分的代码。

根据遍历访问的顺序处理结点内容

递归算法的关键要明确函数的定义，相信这个定义，而不要跳进递归细节

写二叉树的算法题，都是基于递归框架的，我们先要搞清楚 root 节点它自己要做什么，根据题目要求选择使用前序，中序，后续的递归框架。

经典动态规划：子集背包问题-分割等和子集

背包问题：载重为W的背包和N个物品，每个物品有重量和价值两个属性w[i],val[i]

用这个背包装能装下的最大价值

对于分割等和子集：给一个载重为sum/2的背包和N个物品，每个物品重量为nums[i]

是否有一种装法 可以将背包装满

第一步：明确状态和选择

状态就是背包的容量和可以选择的物品

选择就是装入该物品和不装入该物品

第二步：明确dp数组的定义

dp[i][j] = x 表示对于前i个物品，当前背包容量为j，x为true表示可以装满

x为false表示不能恰好将背包装满

如dp[4][9] = true 表示对于容量为9的背包，若只用前4个物品，则可以有一种方法恰好装满

我们需要的答案就是dp[N][sum/2]

base case就是dp[...][0] = true, dp[0][...]=false

背包没有空间相当于装满了，没有物品可选必然不能装满背包

第三步：根据选择，进行状态转移

如果不将nums[i]算入子集，也就是不把这个第i个物品装入背包，那能否装满

取决于上一个状态，继承之前的结果

如果将nums[i]算入子集，也就是将这个第i个物品装入背包，那能否装满取决于

状态dp[i-1][j-nums[i]]

```

1 bool canPartition(vector<int>& nums) {
2     int sum = 0;
3     for (int num : nums) sum += num;
4     // 和为奇数时，不可能划分成两个和相等的集合
5     if (sum % 2 != 0) return false;
6     int n = nums.size();
7     sum = sum / 2;
8     vector<vector<bool>>
9         dp(n + 1, vector<bool>(sum + 1, false));
10    // base case
11    for (int i = 0; i <= n; i++)
12        dp[i][0] = true;
13
14    for (int i = 1; i <= n; i++) {
15        for (int j = 1; j <= sum; j++) {
16            if (j - nums[i - 1] < 0) {
17                // 背包容量不足，不能装入第 i 个物品
18                dp[i][j] = dp[i - 1][j];
19            } else {
20                // 装入或不装入背包
21                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
22            }
23        }
24    }
25    return dp[n][sum];
26 }

```

由于 i 是从 1 开始的，而数组索引是从 0 开始的，所以第 i 个物品的重量应该是 nums[i-1]

这里开的 dp 长度为 n+1，索引为 0 1 ...n， 第二维长度 sum+1 索引为 0 1 ..sum

开长度为 n 的 dp 也可以，注意 dp 的下标对应 nums 下标的索引意义为物品个数和对应的重量，dp 与 nums 要对应

```

416. 分割等和子集
通过 中等 | 提交 | 我的 | 举报
给你一个 只包含正整数 的非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
示例 1：
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。
示例 2：
输入：nums = [1,2,3,5]
输出：false
解释：数组不能分割成两个元素和相等的子集。
提示：
1 <= nums.length <= 200
1 <= nums[i] <= 100
通过次数 131,210 提交次数 262,674

```

```

1 class Solution {
2 public:
3     bool canPartition(vector<int>& nums) {
4         int n = nums.size();
5         if(n<2) return false;//不能划分
6         int sum = accumulate(nums.begin(),nums.end(),0);
7         // 和为奇数时，不可能划分成两个和相等的集合
8         if (sum % 2 != 0) return false;
9         int target = sum/2;
10        int maxNum = *max_element(nums.begin(), nums.end());//C++中*max_element(v.begin,v.end)找最大元素
11        //*min_element(v.begin,v.end)找最小元素
12        if (maxNum > target) { //如果最大元素大于target，那么除了maxNum以外的其他元素和将小于target，不能将num分为2组和相等的子集
13            return false;
14        }
15        if(sum & 1) return false;//sum为奇数，也不能划分
16
17        vector<vector<int>> dp(n, vector<int>(target + 1, 0));//累加的从0开始最多累加到target，累加和为0也要考虑，因为累加和为0就是不取元素
18        for(int i = 0; i <n; i++) { //这也是一个剪枝法 dp[i][0]为true
19            dp[i][0] = true; //下标到0内可以累加到0的方法，取0个元素也可以，因此无论为多少，dp[i][0] = 0
20            dp[0][nums[0]] = true; //下标为到范围内，累加为nums[0] 的取法一个为true，就是取第一个元素
21            for(int i = 1; i < n; i++) {
22                for(int j = 1; j <= target; j++) {
23                    if(j > nums[i]) { // 装入或不装入背包
24                        dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i]]; //当前这个元素装或不装都可以
25                    } else { //背包容量不足，不能装入第 i 个物品
26                        dp[i][j] = dp[i-1][j]; //由于j<nums[i] 要保证0...i下标累加和为j，当前元素不可以取，只能看到i-1 下标内累加和是否可以
27                    }
28                }
29            }
30        }
31    }

```

状态压缩（没有看懂）

```

1 bool canPartition(vector<int>& nums) {
2     int sum = 0, n = nums.size();
3     for (int num : nums) sum += num;
4     if (sum % 2 != 0) return false;
5     sum = sum / 2;
6     vector<bool> dp(sum + 1, false);
7     // base case
8     dp[0] = true;
9
10    for (int i = 0; i < n; i++) {
11        for (int j = sum; j >= 0; j--) {
12            if (j - nums[i] >= 0)
13                dp[j] = dp[j] || dp[j - nums[i]];
14        }
15    }
16    return dp[sum];

```

注意到 dp[i][j] 都是通过上一行 dp[i-1][..] 转移过来的，之前的数据都不会再使用了

解法思路完全相同，只在一行 dp 数组上操作 i 每进行一轮迭代，dp[i] 其实就相当于 dp[i-1][j]，所以只需要一维数组就够了用

唯一需要注意的是 j 应该从后往前反向遍历，因为每个物品（或者说数字）只能用一次，以免之前的结果影响其他的结果

滚动数组以及为什么要从右往左遍历

```

/*dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i]]*/
dp[n] (n <= j)其实是dp[i - 1][m]，dp[m] (m > j)其实是dp[i][m]
dp[j] = dp[j] || dp[j - nums[i]]; 这里的左边的dp[j]是我们需要更新的，当前层第i层的状态，右边的dp[j]是第i-1层的状态
按照思路其实可以修改成dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i]];
因为我们dp[i][j]只与上一层以及左上方相关，如果我们从左到右更新，那么dp[k] (k < j)其实是dp[i][k]，不符合要求
如果我们从右往左更新，那么左边的状态没有更新过，dp[k] (k < j)是dp[i - 1][k]符合要求
就像一个轮胎，它滚过去的部分就是新的状态。在它之前没有滚过去的部分就是旧状态

```

压缩到一维时，要采用逆序。dp[j] = dp[j] || dp[j - nums[i]] 可以理解为 dp[j] (新) = dp[j] (旧) || dp[j - nums[i]] (旧)，如果采用正序的话 dp[j - nums[i]] 会被之前的操作更新为新值。因为在一维情况下，是根据 dp[j] || dp[j - nums[i]] 来推 dp[j] 的值，如不逆序，就无法保证在外循环 i 值保持不变 j 值递增的情况下，dp[j - num[i]] 的值不会被当前所放入的 nums[i] 所修改，当 j 值未到达临界条件前，会一直被 nums[i] 影响，即重复的放入了多次 nums[i]，为了避免前面对后面产生影响，故用逆序。

举个例子，数组为 [2,2,3,5]，要找和为 6 的组合，i = 0 时，dp[2] 为真，当 i 自增到 1，j = 4 时，nums[i] = 2, dp[4] = dp[4] || dp[4 - 2] 为 true，当 i 不变，j = 6 时，dp[6] = dp[6] || dp[6 - 2]，而 dp[4] 为 true，所以 dp[6] = true，显然是错误的。故必须得纠正正在正序情况下，i 值不变时多次放入 nums[i] 的情况

一维数组的时候后面会用到前面的结果 所以要从大到小，二维的时候全部再上面

*<https://www.bilibili.com/video/BV1kp4y1e794?from=search&seid=722142665339590717>

9.71 01背包

```

for(i=1;i<=n;i++){
    for(j=1;j<=m;j++){
        if(f[i][j] == 0)
            f[i][j] = f[i-1][j];
        else
            f[i][j] = max(f[i-1][j], f[i-1][j-w[i]]+c[i]);
    }
}
printf("%d", f[n][m]);

```

i	0	1	2	3	4	5	6
0	0	0	0	0	0	0	w, c
1	0	0	5	5	5	3, 5	
2	0	0	3	5	5	8	2, 3
3	0	0	3	5	6	8	4, 6

用一维数组f[j]只记录一行数据。
让值顺序循环，顺序更新f[j]值会怎样？

```

for(i=1;i<=n;i++){
    for(j=1;j<=m;j++){
        if(f[i][j] == 0)
            f[j] = f[i];
        else
            f[j] = max(f[j], f[j-w[i]]+c[i]);
    }
}

```

因为j是顺序循环，f[j-w[i]]会先于f[j]更新，也就是说，用新值f[j-w[i]]去更新f[j]，所以出错。

9.71 01背包

```

for(i=1;i<=n;i++){
    for(j=m;j>=1;j--){
        if(f[j] > 0)
            f[i][j] = f[i-1][j];
        else
            f[i][j] = max(f[i-1][j], f[i-1][j-w[i]]+c[i]);
    }
}
printf("%d", f[n][m]);

```

i	0	1	2	3	4	5	6
0	0	0	0	0	0	0	w, c
1	0	0	5	5	5	3, 5	
2	0	0	3	5	5	8	2, 3
3	0	0	3	5	6	8	4, 6

用一维数组f[j]只记录一行数据。
让值逆序循环，逆序更新f[j]值。

```

for(i=1;i<=n;i++){
    for(j=m;j>=1;j--){
        if(f[j] > 0)
            f[i][j] = f[i-1][j];
        else
            f[i][j] = max(f[j], f[j-w[i]]+c[i]);
    }
}

```

因为j是逆序循环，f[j]会先于f[j-w[i]]更新，也就是说，用旧值f[j-w[i]]去更新f[j]，相当于用上一行的f[j-w[i]]去更新f[j]，所以正确。

第一章、手把手刷数据结构

1.递归反转整个链表

```

1 ListNode reverse(ListNode head) {
2     if (head.next == null) return head;
3     ListNode last = reverse(head.next);
4     head.next = head;
5     head.next = null;
6     return last;
7 }

```

明确递归函数每一个结点干的事情，即递归函数的定义

reverse函数做的事情/定义是就将一个结点head的下一点指向head，本身head结点再指向null

并返回反转完的结点，这里要用一个变量接收递归函数的返回值

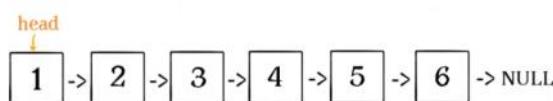
这里是后序遍历，下探完成，回溯时函数将访问每一个结点，最上面那层函数将返回尾结点head

用一个last接住它，last将随着每一层reverse的执行返回一起返回到最底层/第一层的那个reverse函数

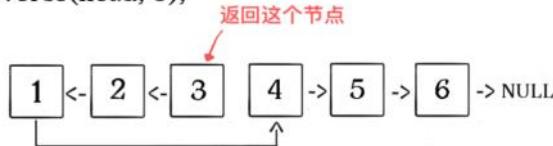
即尾结点通过last依次向前返回/return一直到最开始的那一层函数 它也会进行return last

反转链表前 N 个节点

比如说对于下图链表，执行 reverseN(head, 3)：



reverse(head, 3);

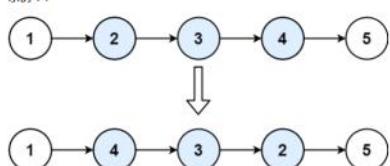


92. 反转链表 II

难度 中等 通过 915 提交

给你单链表的头指针 head 和两个整数 left 和 right，其中 left <= right。请你反转从位置 left 到位置 right 的链表节点，返回反转后的链表。

示例 1：



输入：head = [1,2,3,4,5], left = 2, right = 4
输出：[1,4,3,2,5]

示例 2：

递归的思想相对迭代思想，稍微有点难以理解，处理的技巧是：不要跳进递归，而是利用明确的定义来实现算法逻辑

但是递归函数最终都是有一个下探和回溯的过程，最终的访问都是通过递归函数的返回来访问每一个结点并对结点进行操作处理

```

6 *   ListNode() : val(0), next(nullptr) {}
7 *   ListNode(int x) : val(x), next(nullptr) {}
8 *   ListNode(int x, ListNode *next) : val(x), next(next) {}
9 *   ~ListNode();
10 */
11 class Solution {
12 public:
13     //反转前n个结点 //反转以 head 为起点的 n 个节点，返回新的头结点
14     ListNode* successor = nullptr;
15     ListNode* reverseN(ListNode* head, int n){
16         if(n == 1){
17             successor = head->next; //记录第n+1个结点 记录后继节点
18             return head;
19         } //以 head->next 为起点，需要反转 n - 1 个节点
20         ListNode* last = reverseN(head->next, n-1);
21         head->next->next = head;
22         head->next = successor;
23         return last;
24     }
25     ListNode* reverseBetween(ListNode* head, int m, int n) {
26         if(m == 1) //相当于反转前 n 个元素
27             return reverseN(head, n); //调用到反转的起点触发 base case
28         head->next = reverseBetween(head->next, m-1, n-1);
29         return head;
30     }
31 };

```

完全不跳入递归感觉难以想清楚，可以简单举例如用3个结点想一下大致的递归过程，确定遍历框架的顺序
处理的技巧：在开始想的时候不进入递归，先想清楚递归函数的定义，尽量不跳进递归，通过明确的定义实现

考虑效率的话还是使用迭代算法 递归操作链表并不高效
和迭代解法相比，虽然时间复杂度都是 $O(N)$ ，但是迭代解法的空间复杂度是 $O(1)$ ，而递归解法需要堆栈，空间复杂度是 $O(N)$

优化空间复杂度

先通过「双指针技巧」中的快慢指针来找到链表的中点

如果fast指针没有指向null，说明链表长度为奇数，slow还要再前进一步

从slow开始反转后面的链表，现在就可以开始比较回文串了

reverse函数很容易实现

算法总体的时间复杂度 $O(N)$ ，空间复杂度 $O(1)$

寻找回文串是从中间向两端扩展，判断回文串是从两端向中间收缩。

对于单链表无法直接倒序遍历，以造一条新的反转链表or新链表

可以利用链表的后序遍历，也可以用栈结构倒序处理单链表

到回文链表的判断问题，由于回文的特殊性，可以不完全反转链表，而是仅仅反转部分链表，将空间复杂度降到 $O(1)$

二叉树

细化题目要求，搞清楚根节点应该做什么，剩下的交给前序/中序/后序遍历框架

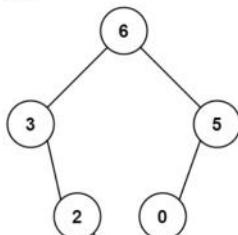
最大二叉树

654. 最大二叉树
难度 中等 294 收藏 分享 为切换为英文 纠错 反馈
给定一个不含重复元素的整数数组 nums ，一个以此数组直接递归构建的最大二叉树，定义如下：

- 二叉树的根是数组 nums 中的最大元素。
- 左子树是通过数组中 最大值左边部分 递归构造出的最大二叉树。
- 右子树是通过数组中 最大值右边部分 递归构造出的最大二叉树。

返回由给定数组 nums 构建的最大二叉树。

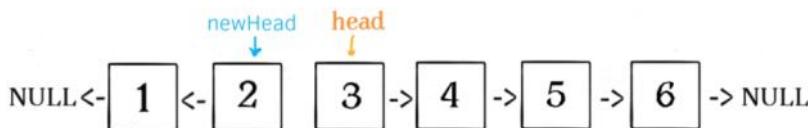
示例 1：



```
1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 };
12
13 class Solution {
14 public:
15     TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
16         return construct(nums, 0, nums.size()-1);
17     }
18     TreeNode* construct(vector<int> nums, int lo, int hi){
19         if(lo > hi)
20             return nullptr;
21         int maxVal = INT_MIN; int index = 0;
22         for(int i = lo; i <= hi; i++){
23             if(nums[i] > maxVal){
24                 maxVal = nums[i];
25                 index = i;
26             }
27         } //前序遍历从上往下找到最大值，并构造这个结点，剩下的左右子树在左数组 和 右数组中再继续构造
28         TreeNode *root = new TreeNode(maxVal);
29         root ->left = construct(nums,lo, index-1); //构造过程主要由这个Index来控制数组索引
30         root ->right = construct(nums, index+1, hi);
31         return root;
32     }
33 }
```

k个一组反转链表

要实现一个 reverse 函数反转一个区间之内的元素。在此之前我们再简化一下，给定链表头结点，如何反转整个链表



reverseKGroup(head, 2)

```
//反转以a为头结点的链表
ListNode* reverser(ListNode* a){
    ListNode* pre, *cur, *nxt;
    pre = null; cur = a; nxt = a;
    while(cur != null){
        nxt = cur ->next;
        cur ->next = pre;
        pre = cur;
        cur = nxt;
    }
    return pre
}
```

}

反转以a为头结点的链表，其实就是反转a到NULL之间的结点，如果是反转a到b之间的结点
只需要更改函数签名，并将null改成b [a,b) 左闭右开区间

```
/** 反转区间 [a, b) 的元素，注意是左闭右开 */
ListNode* reverse(ListNode* a, ListNode* b){ // 
    ListNode* pre, cur, nxt;
    pre = null; cur = a; nxt = a;
    while(cur != b){
        nxt = cur->next;
        cur->next = pre;
        pre = cur;
        cur = nxt;
    }
    return pre;
}
```

25. K 个一组翻转链表
 难度 中等 □ 1120 ☆ 收藏 □ 分享 ☰ 切换为英文 ☱ 接收动态 ☱ 反馈

给你一个链表，在 k 个节点一组进行翻转，请你返回翻转后的链表。

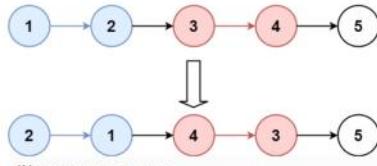
k 是一个正整数，它的值小于或者等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

进阶：

- 你可以设计一个只使用常数额外空间的算法来解决此问题吗？
- 你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

示例 1：



输入 : head = [1,2,3,4,5], k = 2

```
1 class Solution {
2     public:
3     //先反转以head开始的k个元素，将反转返回的头结点作为新的头结点返回，在刚开始将结点a的下一节点即为下一个结点的next指向
4     //先反转前 k 个元素，进归反转剩余部分到newhead结点并连接起来，从左到右 先遍历返回最初的那个newhead结点
5
6     ListNode* reverse(ListNode* a, ListNode* b){ //
7         ListNode* pre, *cur, *nxt;
8         pre = null; cur = a; nxt = a;
9         while(cur != b){ // 反转区间 [a, b) 的元素，注意是在右闭区间
10            nxt = cur->next;
11            cur->next = pre;
12            pre = cur;
13            cur = nxt;
14        }
15        return pre; // // 反转链表的头结点
16    }
17
18    ListNode* reverseGroup(ListNode* head, int k){ //迭代的方法
19        if(head == nullptr) return head; //遍历到了最末端结点
20        //遍历 [a, b) 包含 k 个结点将元素 1 2 3 4 5 ; k ==2 简述到 以保证左闭右开区间
21        ListNode *a,*b;
22        a = b = head;
23        for(int i = 0; i < k; i++){ // 不是 k 个，不需要反转
24            if(i < k-1) { //从 a 到 b - 1 执行反转
25                b = b->next;
26            } // 反转的 k 个元素
27            a = reverse(a,b); //先倒置从左到右的逆序每一个结点并执行反转
28            a->next = reverseGroup(b, k); // 1 2 3 4 5 ; k=2; 第一次b在3处，第二次b在5处
29        }
30        return head;
31    }
32    /* 递归的方法
33 }
```

然后到子树的中序数组中找到这个root的索引，左边部分就是左子树的长度，右边就是右子树的长度

再根据这个长度确定子树前序数组的区间，以得到下一次递归左右子树的前序数组

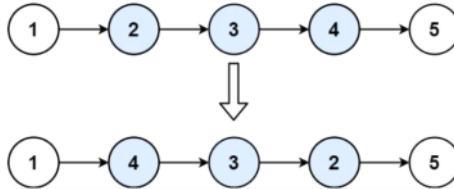
整体代码是按照前序遍历构造每一个结点

92. 反转链表 II

难度 中等 □ 1005 ☆ 收藏 □ 分享 ☰ 切换为英文 ☱ 接收动态 ☱ 反馈

给你单链表的头指针 $head$ 和两个整数 $left$ 和 $right$ ，其中 $left \leq right$ 。请你反转从位置 $left$ 到位置 $right$ 的链表节点，返回 反转后的链表。

示例 1：



输入 : head = [1,2,3,4,5], left = 2, right = 4

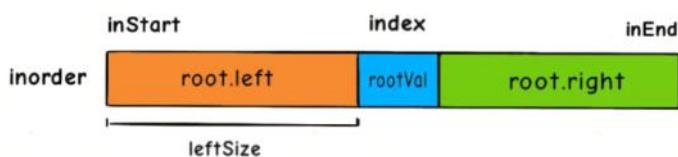
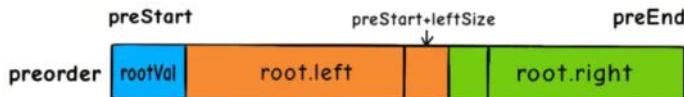
输出 : [1,4,3,2,5]

示例 2：

输入 : head = [5], left = 1, right = 1

输出 : [5]

```
58     private:
59     void reverseLinkedList(ListNode *head) {
60         ListNode *pre = nullptr;
61         ListNode *cur = head;
62         while (cur != nullptr) {
63             ListNode *next = cur->next;
64             cur->next = pre;
65             pre = cur;
66             cur = next;
67         }
68     }
69     public:
70     ListNode *reverseBetween(ListNode *head, int left, int right) {
71         ListNode *dummyNode = new ListNode(-1);
72         dummyNode->next = head;
73         ListNode *pre = dummyNode;
74         for (int i = 0; i < left - 1; i++) {
75             pre = pre->next;
76         } // 第 2 步：从 pre 再走 right - left + 1 步，来到 right 节点
77         ListNode *rightNode = pre;
78         for (int i = 0; i < right - left + 1; i++) {
79             rightNode = rightNode->next;
80         }
81         ListNode *leftNode = pre->next;
82         ListNode *curr = rightNode->next;
83         pre->next = rightNode;
84         // pre->next = nullptr; // 注意：切断链接
85         rightNode->next = nullptr;
86         reverseLinkedList(leftNode);
87         leftNode->next = curr;
88         return dummyNode->next;
89     }
90 }
```



首先要遍历数组找到最大值Maxval，即为根节点，再对Maxval左边的数组和右边的数组递归调用，作为root的左右子树，递归构造左右子树

根据前序和中序遍历构造二叉树

每一个子树前序遍历数组的第一个元素就是子树的root

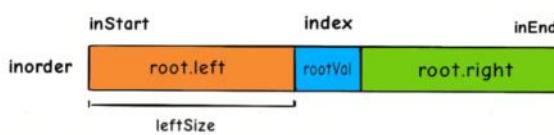
```
TreeNode build(int[] preorder, int preStart, int preEnd, 
              int[] inorder, int inStart, int inEnd) {
    if (prestart > preEnd) {
        return null;
    }

    // root 节点对应的值就是前序遍历数组的第一个元素
    int rootVal = preorder[preStart];
    // rootVal 在中序遍历数组中的索引
    int index = 0;
    for (int i = inStart; i < inEnd; i++) {
        if (inorder[i] == rootVal) {
            index = i;
            break;
        }
    }

    int leftSize = index - inStart;

    // 先构造出当前根节点
    TreeNode root = new TreeNode(rootVal);
    // 递归构造左右子树
    root.left = build(preorder, preStart + 1, preStart + leftSize,
                      inorder, inStart, index - 1);

    root.right = build(preorder, preStart + leftSize + 1, preEnd,
                       inorder, index + 1, inEnd);
    return root;
}
```



106. 从中序与后序遍历序列构造二叉树
难度 中等
贡献 517 ★ 10 % 0 回
根据一模一样的中序遍历与后序遍历构造二叉树。
注意：
你可以假设树中没有重复的元素。
例如，给出

中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]

```
3 * struct TreeNode {
4 *     int val;
5 *     TreeNode *left;
6 *     TreeNode *right;
7 * };
8 * TreeNode() : val(0), left(nullptr), right(nullptr) {}
9 * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10 * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 */
12 class Solution {
13 public:
14     * TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
15         if (inorder.size() == 0 || postorder.size() == 0) {
16             return nullptr;
17         }
18         return construct(inorder, 0, inorder.size() - 1, postorder, 0, postorder.size() - 1);
19     }
20     * TreeNode* construct(vector<int>& inorder, int inStart, int inEnd, vector<int>& postorder, int poStart, int poEnd) {
21         if (inStart > inEnd) {
22             return nullptr;
23         }
24         int rootVal = postorder[poEnd];
25         int index = 0;
26         for (int i = inStart; i < inEnd; i++) {
27             if (inorder[i] == rootVal) {
28                 index = i;
29                 break;
30             }
31         }
32         int leftSize = index - inStart;
33         int rightSize = inEnd - index;
34         TreeNode* root = new TreeNode(rootVal);
35         root->left = construct(inorder, inStart, index - 1, postorder, poStart, poStart + leftSize - 1);
36         root->right = construct(inorder, index + 1, inEnd, postorder, poStart + leftSize, poEnd - 1);
37         return root;
38     }
39 }
```

```

106. 从中序与后序遍历序列构造二叉树
难度 中等 点 517 收藏 0 举报
根据一棵树的中序遍历与后序遍历构造二叉树。
注意：你可以假设树中没有重复的元素。
例如，给出
中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]

返回如下的二叉树：
    3
   / \
  9  20
   \   \
    15  7

通过次数 10,528 提交次数 155,364
请问您在做本题时遇到过困难？
杜振 杨振 实习 未通过
13
14     * struct TreeNode {
15     *     int val;
16     *     TreeNode *left;
17     *     TreeNode *right;
18     * };
19
20 class Solution {
21 public:
22     TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
23         return construct(inorder, 0, inorder.size() - 1, postorder, 0, postorder.size() - 1);
24     }
25     TreeNode* construct(vector<int>& inorder, int inStart, int inEnd, vector<int>& postorder, int poStart, int poEnd){
26         if(poEnd > inEnd)
27             return nullptr;
28         int rootVal = postorder[poEnd];
29         int index = 0;
30         for(int i = inStart; i < inEnd; i++){
31             if(inorder[i] == rootVal){
32                 index = i;
33                 break;
34             }
35         }
36         //从代码框架上来看，还是一个前序遍历的框架，即先找到根结点构造出来，再找到根结点的左右子树构造的构造每一个节点，前序遍历从上到下按
37         int leftSize = index - inStart; //根据左右的顺序构造每一个节点，直到所有根结点构造完，返回最初的root
38         TreeNode* root = new TreeNode(rootVal);
39         root->left = construct(inorder, inStart, index - 1, postorder, poStart, poStart + leftSize - 1);
40         root->right = construct(inorder, index + 1, inEnd, postorder, poStart + leftSize, poEnd - 1);
41         return root;
42     }
43

```

递归的性质，就是你找到一个根节点，并对应有遍历数组

接下来这个根节点的左右子树也是有对应的遍历数组，每个子树与每个子树遍历数组都满足这些遍历顺序，你首先对一个大的二叉树这样操作了，找到了这个根结点应该做的事情，其实就是要将它构造出来，那剩下的子树处理也是这样重复调用/亦称递归调用这个定义的函数/亦称定义的递归函数，这个递归函数调用的过程也就是相当于一个指针在二叉树上移动访问亦称遍历每一个结点，顺序就是前序、中序、后序遍历就这三种框架

做二叉树的问题，关键是把题目的要求细化，搞清楚根节点应该做什么，然后剩下的事情抛给前/中/后序的遍历框架就行了

计算节点个数：标准的后序遍历框架

```

int count(TreeNode *root){
    if(root == nullptr)
        return 0;
    int left = count(root->left);
    int right = count(root->right);
    int res = left + right + 1;
    return res;
}

```

重复子树，借助一个数据结构让每个节点将自己子树序列化后的结果存入
即使用一个hashset记录子树

```

652. 寻找重复的子树
难度 中等 点 276 收藏 0 举报
给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要
返回其中任意一棵的根结点即可。
两棵树重复是指它们具有相同的结构以及相同的结点值。
示例 1：

    1
   / \
  2  3
   / \
  4  4

下面是两个重复的子树：

    2
   /
  4
和

    4

```

```

8     * struct TreeNode {
9     *     int val();
10    *     TreeNode *left;
11    *     TreeNode *right;
12    * };
13
14 class Solution {
15 public:
16     //unordered_set<string> memo;
17     //unordered_map<string, int> memo;
18     vector<TreeNode*> res;
19     vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
20         traverse(root);
21         return res;
22     }
23     //后序遍历一般需要返回值，因为后序遍历需要回溯，很多要用到底层的节点，//开始从下往上遍历记录子树序列化结果
24     string traverse(TreeNode* root){
25         if(root == nullptr)
26             return "";
27         string left = traverse(root->left);
28         string right = traverse(root->right);
29         string subtree = left + "," + right + "," + to_string(root->val);
30         /*
31         if(memo.count(subtree))//这样如果subtree 有多个就会出现多次push同一个root到res，出现重复元素
32             res.push_back(root);
33         else
34             memo.insert(subtree);
35         */
36         if(memo[subtree] == 1)
37             res.push_back(root);
38         memo[subtree]++;
39         return subtree;
40     }
41

```

BST的特性：

1.对于BST的每一个节点node，左子树节点都比node小，右子树节点值都比node大

2.对于BST的每一个节点node，它的左侧子树和右侧子树都是BST

二叉搜索树本身并不复杂，但是它可以算得上是数据结构的半壁江山，直接基于BST

的数据结构有AVL树，红黑树等，拥有了自平衡性质，可以提供 $\log N$ 级别的增删查改效率

还有B+树，线段树等结构都是基于BST的思想设计的

从算法角度，BST有一个重要性质，BST的中序遍历结果是升序的！

```

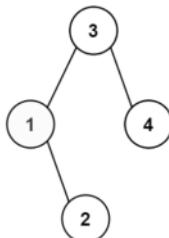
void traverse(TreeNode *root){
    if(root == nullptr)
        return;
    traverse(root->left);
    //中序遍历代码位置处
    print(root->val);
    traverse(root->right);
}

```

寻找第 K 小的元素

给定一个二叉搜索树的根节点 `root`，和一个整数 `k`，请你设计一个算法查找其中第 `k` 个最小元素（从 1 开始计数）。

示例 1：



输入：`root = [3,1,4,null,2]`, `k = 1`
输出：`1`

```

1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 }
12
13 class Solution {
14 public:
15     int res = 0;
16     int rank = 0;
17     int kthSmallest(TreeNode* root, int k) {
18         traverse(root, k);
19         return res;
20     }
21     //BST 的中序遍历其实就是升序排序的结果
22     void traverse(TreeNode* root, int k) {
23         if (root == nullptr)
24             return;
25         traverse(root->left, k);
26         rank++;
27         if (rank == k) {
28             res = root->val;
29             return;
30         }
31         traverse(root->right, k);
32     }
33 }
```

BST 的中序遍历代码可以升序打印节点的值：

```

void traverse(TreeNode root) {
    if (root == null) return;
    traverse(root.left);
    // 中序遍历代码的位置
    print(root.val);
    traverse(root.right);
}
  
```

那如果我想降序打印节点的值怎么办？很简单，只要把递归顺序改一下就行了：

```

void traverse(TreeNode root) {
    if (root == null) return;
    // 先递归遍历右子树
    traverse(root.right);
    // 中序遍历代码的位置
    print(root.val);
    // 后递归遍历左子树
    traverse(root.left);
}
  
```

这段代码可以从大到小降序打印 BST 节点的值，如果维护一个外部累加变量 `sum`，然后把 `sum` 赋值给 BST 中的每一个节点，不就将 BST 转化成累加树了

538. 把二叉搜索树转换为累加树
难度 中等 | 518 收藏 分享 切换为英文 接收动态 反馈

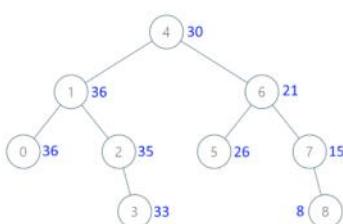
给出二叉搜索树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

提醒一下，二叉搜索树满足下列的条件：

- 节点的左子树仅包含值 小于 节点值的节点。
- 节点的右子树仅包含值 大于 节点值的节点。
- 左右子树也必须是二叉搜索树。

注意：本题和 103 题 <https://leetcode-cn.com/problems/maximum-binary-tree/> 相同

示例 1：



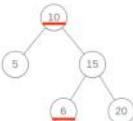
核心还是 BST 的中序遍历特性，只不过我们修改了递归顺序，降序遍历 BST 的元素值，从而契合题目累加树的要求

判断BST的合法性

```

bool isValidBST(TreeNode root) {
    if (root == null) return true;
    if (root.left != null && root.left.val < root.left.val)
        return false;
    if (root.right != null && root.right.val >= root.right.val)
        return false;
    return isValidBST(root.left)
        && isValidBST(root.right);
}
  
```

但是这个算法出错了错误，BST 的每个节点应该要小于右边子树的所有节点，下面这个二叉树竟然不是 BST，因为结合 10 的右子树中有一个节点 6，但是我们的算法会把它判定为合法 BST：



出现问题的原因在于，对于每一个节点 `root`，代码值检查了它的左右孩子节点是否符合左小右大的原则；但是根据 BST 的定义，`root` 的整个左子树都要小于 `root.val`，整个右子树都要大于 `root.val`。

问题是，对于某一个节点 `root`，他只能管得了自己左右子节点，怎么把 `root` 的约束传递给左右子树呢？

请看正确的代码：

```
boolean isValidBST(TreeNode root) {
    return isValidBST(root, null, null);
}

// 题目以 root 为根的子树必须满足 max.val > root.val > min.val */
boolean isValidBST(TreeNode root, TreeNode min, TreeNode max) {
    // base case
    if (root == null) return true;
    // 若 root.val 不符合 max 和 min 的限制，说明不是合法 BST
    if (min != null && root.val <= min.val) return false;
    if (max != null && root.val >= max.val) return false;
    // 确定左子树的最大值是 root.val，右子树的最小值是 root.val
    return isValidBST(root.left, min, root)
        && isValidBST(root.right, root, max);
}
```

我们通过使用辅助函数，增加函数参数列表，在参数中携带额外信息，将这种约束传递给子树的所有节点，这也是二叉树算法的一个小技巧吧。

```
10
5      19
12      21
11  16          11再小也不能比10大 16再大也不能比19大！！！
```

如果是在二叉树中寻找元素，可以这样写代码：

```
boolean isInBST(TreeNode root, int target) {
    if (root == null) return false;
    if (root.val == target) return true;
    // 当前节点没找到就递归地去左右子树寻找
    return isInBST(root.left, target)
        || isInBST(root.right, target);
}
```

这样写完全正确，但这段代码相当于穷举了所有节点，适用于所有普通二叉树。那么应该如何充分利用信息，把 BST 这个「左小右大」的特性用上？

很简单，其实不需要递归地搜索两边，类似二分查找思想，根据 `target` 和 `root.val` 的大小比较，就能排除一边。我们把上面的思路稍作改动：

```
boolean isInBST(TreeNode root, int target) {
    if (root == null) return false;
    if (root.val == target) return true;
    if (root.val < target)
        return isInBST(root.right, target);
    if (root.val > target)
        return isInBST(root.left, target);
    // root 该做的事做完了，顺带把框架也完成了，妙
}
```

于是，我们对原始框架进行改造，抽象出一套针对 BST 的遍历框架：

```
void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

这个代码框架其实和二叉树的遍历框架差不多，无非就是利用了 BST 左小右大的特性而已。

LC-二叉树插入元素

701. 二叉搜索树中的插入操作
难度：中等
贡献：216
收藏：10 分享：74
切换为英文
Q 换成动态
D 反馈
给常二叉搜索树（BST）的根节点和要插入树中的值，将值插入二叉搜索树。返回插入后二叉搜索树的根节点。输入数据保证，新值和原二叉搜索树中的任意节点值都不相同。
注意：可能存在多种有效的插入方式，只要能在插入后仍保持为二叉搜索树即可。你可以返回任意有效的结果。
示例 1：

输入：root = [4,2,7,1,3], val = 5
输出：[4,2,7,1,3,5]
解释：另一个满足题目要求可以通过的树是：
其实先判断右边也可以...

```
1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8 };
9 TreeNode::TreeNode() : val(0), left(nullptr), right(nullptr) {}
10 TreeNode::TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
11 TreeNode::TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
12
13 class Solution {
14 public:
15     TreeNode* insertIntoBST(TreeNode* root, int val) {
16         if (root == nullptr)
17             return new TreeNode(val);
18         if (root->val > val)
19             root->left = insertIntoBST(root->left, val); // 先左后右的原则
20         else
21             root->right = insertIntoBST(root->right, val);
22         return root;
23     }
24 }
```

BST 中插入一个数

对数据结构的操作无非遍历 + 访问，遍历就是「找」，访问就是「改」。具体到这个问题，插入一个数，就是先找到插入位置，然后进行插入操作。

上一个问题，我们总结了 BST 中的遍历框架，就是「找」的问题。直接套框架，加上「改」的操作即可。一旦涉及「改」，函数就要返回 `TreeNode` 类型，并且对

对数据结构的操作无非遍历 + 访问，遍历就是「找」，访问就是「改」。具体到这个问题，插入一个数，就是先找到插入位置，然后进行插入操作。

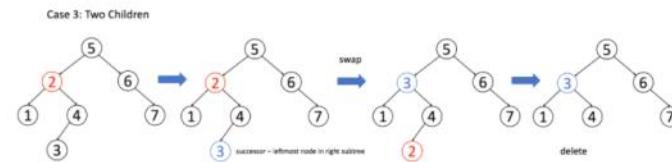
上一个问题，我们总结了 BST 中的遍历框架，就是「找」的问题。直接套框架，加上「改」的操作即可。一旦涉及「改」，函数就要返回 `TreeNode` 类型，并且对递归调用的返回值进行接收。

```
TreeNode insertIntoBST(TreeNode root, int val) {
    // 找到空位置插入新节点
    if (root == null) return new TreeNode(val);
    // if (root.val == val)
    //     BST 中一般不会插入已存在元素
    if (root.val < val)
        root.right = insertIntoBST(root.right, val);
    if (root.val > val)
        root.left = insertIntoBST(root.left, val);
    return root;
}
```

在 BST 中删除一个数

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root.val == key) {
        // 找到要删除的节点
    } else if (root.val > key) {
        // 去左子树找
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        // 去右子树找
        root.right = deleteNode(root.right, key);
    }
    return root;
}
```

情况 3：A 有两个子节点，麻烦了，为了不破坏 BST 的性质，A 必须找到左子树中最大的那个节点，或者右子树中最小的那个节点来接替自己。我们以第二种方式讲解。



```
if (root.left != null && root.right != null) {
    // 找到右子树的最小节点
    TreeNode minNode = getMin(root.right);
    // 把 root 改成 minNode
    root.val = minNode.val;
    // 然而去删除 minNode
    root.right = deleteNode(root.right, minNode.val);
}
```

三种情况分析完毕，填入框架，简化一下代码：

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root == null) return null;
    if (root.val == key) {
        // 这两个 if 把情况 1 和 2 都正确处理了
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;
        // 处理情况 3
        TreeNode minNode = getMin(root.right);
        root.val = minNode.val;
        root.right = deleteNode(root.right, minNode.val);
    } else if (root.val > key) {
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        root.right = deleteNode(root.right, key);
    }
    return root;
}

TreeNode getMin(TreeNode node) {
    // BST 最左边的结点是最近的
    while (node.left != null) node = node.left;
    return node;
}
```

因为我们一般不会通过 `root.val = minNode.val` 修改节点内部的值来交换节点，而是通过一系列略微复杂的链表操作交换 `root` 和 `minNode` 两个节点。

技巧：

- 1、如果当前节点会对下面的子节点有整体影响，可以通过辅助函数增长参数列表，**借助参数传递信息**。
- 2、在二叉树递归框架之上，扩展出一套 BST 代码框架：

```
void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

- 3、根据代码框架掌握了 BST 的增删查改操作

不同 BST 的个数：

前文动态规划相关的问题多次讲过消除重叠子问题的方法，无非就是加一个备忘录：

```
1 // 备忘录
2 int memo[10][10];
3
4 int numTrees(int n) {
5     // 备忘录的初始化为 0
6     memo = new int[n + 1][n + 1];
7     return count(1, n);
8 }
9
10 int count(int lo, int hi) {
11     if (lo > hi) return 1;
12     // 查备忘录
13     if (memo[lo][hi] != 0) {
14         return memo[lo][hi];
15     }
16
17     int res = 0;
18     for (int mid = lo; mid <= hi; mid++) {
19         int left = count(lo, mid - 1);
20         int right = count(mid + 1, hi);
21         res += left * right;
22     }
23     // 将结果存入备忘录
24     memo[lo][hi] = res;
25
26     return res;
27 }
```

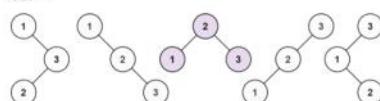
LC-不同的二叉搜索树个数

96. 不同的二叉搜索树

难度 中等
贡献 1190
收藏 150
点赞 0
分享 0

给你一个整数 n ，求恰好由 n 个节点组成且节点值从 1 到 n 互不相同的二叉搜索树有多少种？返回满足题意的二叉搜索树的种数。

示例 1：



输入： $n = 3$

输出：5

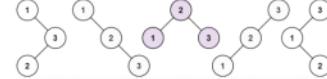
关键在于选取哪一个节点作为根节点 n 个节点组成的搜索二叉树，从 $1 \dots n$ 每一个节点都可以作为根节点使用

95. 不同的二叉搜索树 II

难度 中等
贡献 900
收藏 150
点赞 0
分享 0

给你一个整数 n ，请你生成并返回所有由 n 个节点组成且节点值从 1 到 n 互不相同的二叉搜索树。可以按 任意顺序 返回答案。

示例 1：



输入： $n = 3$

输出：[[1, null, 2, null, 3], [1, null, 3, 2], [2, 1, 3], [3, 1, null, 2], [3, 2, null, 1]]

示例 2：

输入： $n = 1$

输出：[[1]]

提示：

• n 的范围是 $1 \leq n \leq 19$

```
1 class Solution {
2 public:
3
4     //f(3) = f(0)\*f(2) + f(1)\*f(1) + f(2)\*f(0)
5     //f(4) = f(0)f(0)\*f(3)f(3) + f(1)f(1)\*f(2)f(2) + f(2)f(2)\*f(1)f(3) + f(3)f(3)\*f(0)f(0)
6     int numTrees(int n) {
7         vector<int> dp(n+1, 0);
8         dp[0] = 1;
9         for(int i = 1; i <= n; i++) { //从1...n的二叉搜索数数目
10             for(int j = 1; j <= i; j++) { //逐步选用1...n作为根节点
11                 dp[i] += dp[i-1]*dp[i-j]; //左侧i-1个数，右侧i-j个数 左侧树个数*右子树个数
12             }
13         }
14     } //考虑到dp[1]==1 这里将dp[0]初值设为1 可以顺利完成循环 计算过程中可以把这些存起来，方便随时使用
15 }
```

图的逻辑结构和具体实现

一幅图由节点和边构成

逻辑结构就是将图抽象成一个数据结构

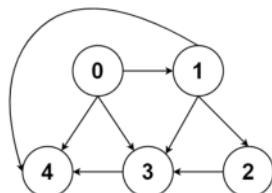
如图节点的实现：

```
class Vertex{
    int id;
    Vertex[] neighbors;
}
```

和多叉树节点相同

```
class TreeNode*{
    int val;
    TreeNode*[] children;
}
```

实际上很少用这个Vertex类实现图，常用邻接表和邻接矩阵来实现



用邻接表和邻接矩阵的存储方式如下：

邻接表		邻接矩阵				
		0	1	2	3	4
0	[4, 3, 1]	0				
1	[3, 2, 4]	1				
2	[3]	2				
3	[4]	3				
4	[]	4				

邻接表：将每个节点的邻居都存到一个列表，再将节点和这个列表关联起来，

这样就可以通过一个节点x找到它所有的相邻节点

邻接矩阵：一个二维布尔数组matrix，如果节点x和节点y是相连的，就将

matrix[x][y]设为true，如果想找节点x的邻居，扫描matrix[x][..]就行了

这两种存储图的方式各有优势

邻接表占用空间少，邻接矩阵里面空着很多位置，同时也需要更多的存储空间

邻接表无法快速判断两个节点是否相邻

如判断节点1和节点3，需要到邻接表里1对应的邻居列表里再查找3是否含有

但是邻接矩阵可以直接看matrix[1][3]效率高

如果是有向加权图的实现：

邻接表：不仅存储某个节点x的所有邻居节点，还存储x到每个邻居的权重

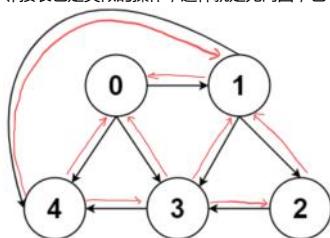
邻接矩阵：matrix[x][y]不再是一个布尔值，而是一个int值，0表示无连接，其他值表示权重

如果是无向图的实现：

所谓的无向图其实就相当于双向图

如果连接无向图中的节点x和y，将matrix[x][y]和matrix[y][x]都变成true就行了

邻接表也是类似的操作，这样就是无向图，也可以称为无向加权图



所有的数据结构都需要进行的操作：遍历 图的遍历

多叉树的遍历框架：

```
void traverse(TreeNoe* root){  
    if(root == null)  
        return;  
    for(TreeNode* child : root ->children)  
        traverse(child);  
}
```

图和多叉树的区别在于图也许包含环，从某个节点开始遍历，也许走一圈又回到了这个节点

如果图包含环，遍历框架需要一个visited数组

图遍历框架：

```
Graph graph;  
boolean[] visited;  
  
void traverse(Graph graph, int s){  
    if( visited[s] ) return;  
    //经过节点s  
    visited[s] = true;  
    for(TreeNode *neighbor : graph ->neighbors(s))  
        traverse(neighbor);  
    //离开节点s  
    visited[s] = false;  
}
```

这个 visited 数组的操作很像回溯算法的做选择和撤销选择，区别在于回溯算法

做选择和撤销选择在for循环里面，而这里对visited数组的操作在for循环外面

在for循环里面和for循环外面唯一的区别就是对根节点的处理

```
1 void traverse(TreeNode root) {  
2     if (root == null) return;  
3     System.out.println("enter: " + root.val);  
4     for (TreeNode child : root.children) {  
5         traverse(child);  
6     }  
7     System.out.println("leave: " + root.val);  
8 }  
9  
10 void traverse(TreeNode root) {  
11     if (root == null) return;  
12     for (TreeNode child : root.children) {  
13         System.out.println("enter: " + child.val);  
14         traverse(child);  
15         System.out.println("leave: " + child.val);  
16     }  
17 }
```

前者会正确打印所有节点的进入和离开信息，而后者唯独会少打印整棵树根节点的进入和离开信息

什么回溯算法框架会用后者？因为回溯算法关注的不是节点，而是树枝，不需要根节点信息

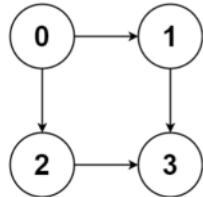
显然，对于这里「图」的遍历，我们应该把 visited 的操作放到 for 循环外面，否则会漏掉起始点的遍历

当有向图含有环的时候才需要 visited 数组辅助，如果不含环，连 visited 数组都省了，基本就是多叉树的遍历

797. 所有可能的路径
难度 中等 通过 126 收藏 分享 切换为英文 接收动态 反馈

给一个有 n 个结点的有向无环图，找到所有从 0 到 $n-1$ 的路径并输出（不要求按顺序）
二维数组中的第 i 个数组中的单元都表示有向图中 i 号结点所能到达的下一些结点（译者注：有向图是有方向的，即规定了 $a \rightarrow b$ 你就不能从 $b \rightarrow a$ ）空就是没有下一个结点了。

示例 1：

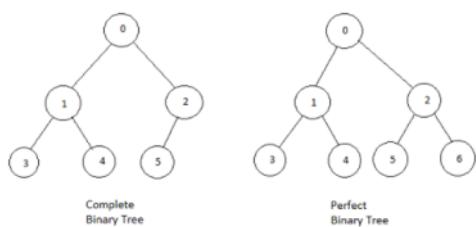


输入：graph = [[1,2],[3],[],[]]
输出：[[0,1,3],[0,2,3]]
解释：有两条路径 0 → 1 → 3 和 0 → 2 → 3

图的遍历其实和回溯框架差不多，只是它在for循环外面撤销操作和加入选择，保证图的每一个节点都遍历到，每次递归执行一次函数执行完成，需要将节点s移出路径，因为只有一个path，需要始终维护这个path路径的存储状态，图的遍历--回溯法--DFS搜索--递归搜索 几乎都是这样的题

```

1 class Solution {
2 public:
3     vector<vector<int>> res;
4     vector<int> path;
5     vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
6         traverse(graph, 0, path);
7         return res;
8     }
9
10 void traverse(vector<vector<int>> &graph, int s, vector<int> path){
11     //添加节点s到路径path
12     path.push_back(s);
13     int n = graph.size();
14     if(s == n-1){ //到达终点
15         res.push_back(path);
16         path.pop_back();
17         return;
18     }
19     //递归每一个与节点s相邻的节点
20     for(int v : graph[s]){
21         traverse(graph, v, path);
22     }
23     //节点s移出路径
24     path.pop_back();
25 }
  
```

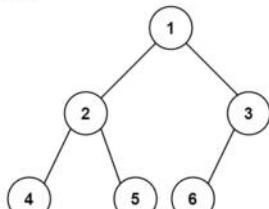


222. 完全二叉树的节点个数
难度 中等 通过 407 收藏 分享 反馈

给你一棵完全二叉树的根节点 $root$ ，求出该树的节点个数。

完全二叉树 的定义如下：在完全二叉树中，除了最底层节点可能不满格外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为 h 层，则该层包含 $1 - 2^h$ 个节点。

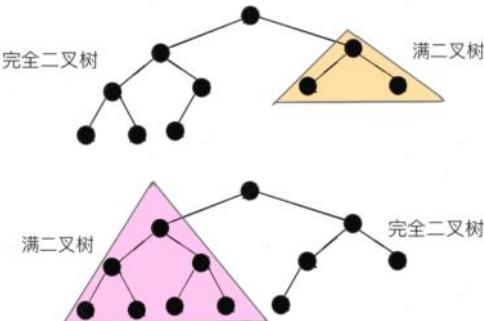
示例 1：



```

1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  * };
8  * TreeNode* : val(x), left(nullptr), right(nullptr) {}
9  * TreeNode* : val(x), left(nullptr), right(nullptr) {}
10 * TreeNode* : val(x), left(left), right(right) {}
11 */
12 class Solution {
13 public:
14     int countNodes(TreeNode* root) {
15         TreeNode *l = root, *r = root;
16         int hl = 0, hr = 0;
17         while(l != nullptr){
18             l = l->left;
19             hl++;
20         }
21         while(r != nullptr){
22             r = r->right;
23             hr++;
24         }
25         if(hl == hr)
26             return pow(2,hl)-1;
27         return 1 + countNodes(root->left) + countNodes(root->right);
28     }
29 }
  
```

这两个递归只有一个会真的递归下去，另一个一定会触发 $hl == hr$ 而立即返回，不会递归下去，一棵完全二叉树的两棵子树，至少有一棵是满二叉树：



算法的递归深度就是树的高度 $O(\log N)$ (平衡二叉树)，每次递归所花费的时间就是 while 循环，需要 $O(\log N)$ ，总体的时间复杂度是 $O(\log N * \log N)$

并查集

并查集是一种数据结构

并 代表合并

查 代表查找

集 代表这是一个以字典为基础的数据结构

基本功能是合并集合中的元素，并查找集合中的元素

并查集典型应用是解决有关连通分量的问题

并查集解决单个问题（添加 合并 查找）的时间复杂度都是 $O(1)$

并查集的实现：数据结构

它跟树相反，树这种数据结构每个节点会记录它的子节点，而并查集的每个节点会记录它的父节点

class UnionFind{

private: //每个节点记录它的父节点

```

    unordered_map<int,int> father
};

如果节点是相互连通的，从一个节点可以到达另一个节点，那么它们在同一棵树里，  

或者在同一个集合里，祖先是一样的

```

初始化

将一个新节点添加到并查集中，它的父节点应该为空

```

void add(int x){
    if(!father.count(x))
        father[x] = -1;
}

```

合并两个节点

如果发现两个节点是连通的，那么就要将它们合并，将谁当父节点一般没有区别

```

void merge(int x, int y){
    int root_x = find(x);
    int root_y = find(y);
    if(root_x != root_y)
        father[root_x] = root_y;
}

```

判断两个节点是否属于同一个连通分量，就需要判断它们的祖先是否相同

```

bool is_connected(int x, int y){
    return find(x) == find(y);
}

```

查找祖先的方法：如果节点的父节点不为空，那就不断迭代

```

int find(int x){
    int root = x;
    while(father[root] != -1)
        root = father[root];
    return root;
}

```

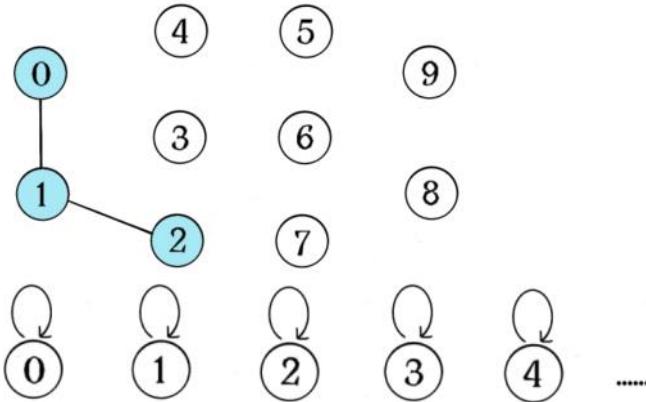
```

-----
class UF{
    void union(int p, int q); //将p和q连接
    boolean connected( int p, int q); //判断p q 是否连通
    int count(); //返回图中有多少个连通分量
}

```

union函数 和 connected函数

用森林表示连通性，设每一个树的节点都有一个指针指向其父结点
如果是根结点这个指针指向自己

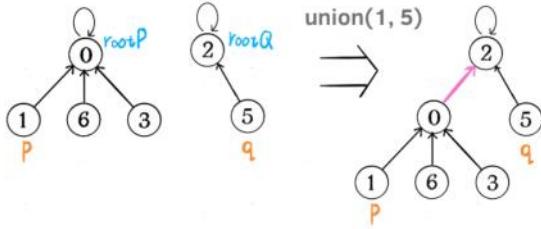


```

class UF{
private:
    int count;
    int parent[];
    UF(int n){
        this.count = n; //n为图的总节点数
        parent = new int[n];
        for(int i = 0; i < n; i++)
            parent[i] = i; //父节点指针初始指向自己
    }
    /* 其他函数 */
}

```

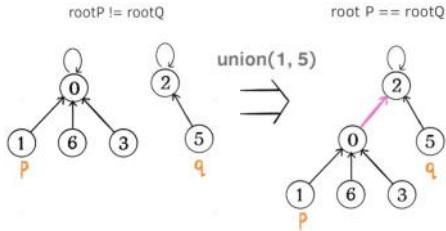
如果两个节点被连通，则让其中任意一个节点的根节点接到另一个节点的根节点



```
void union (int p, int q){ //将两棵树合并为一棵
    int rootP = find(p);
    int rootQ = find(q);
    if(rootP == rootQ)
        return;
    parent[rootP] = rootQ;
    count--;
}
```

```
int find(int x){ //返回某个节点x的根节点
    //根节点的parent[x] == x;
    while(parent[x] != x)
        x = parent[x];
    return x;
}
int count(){ //返回当前的连通分量个数
    return count;
}
```

这样如果节点p和节点q连通的话，它们一定拥有相同的根节点



```
boolean connected(int p, int q){ //判断是否连通
    int rootP = find(p);
    int rootQ = find(q);
    return rootP == rootQ;
}
```

以上就是Union-find 算法，通过数组来模拟出一个森林

主要API connected 和 union 中的复杂度都是由find函数造成的

所以它们的复杂度和find函数相同

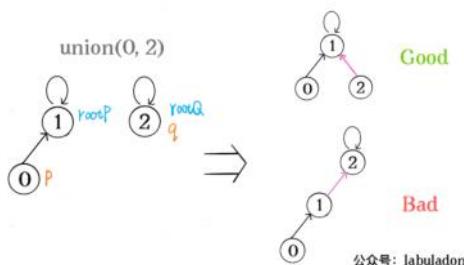
find的主要功能就是从某个节点向上遍历到树根，时间复杂度就是树的高度，
树的高度不一定是 $\log N$ ，只有平衡二叉树是 $\log N$ 高度，一般树如果是极度不平衡
就会退化为链表，高度变为N，那find的时间复杂度就是 $O(N)$
如果对于union和connected的调用非常频繁，而每次调用需要线性时间那就不行

关键在于如何想办法避免树的不平衡

union 过程：

```
void union( int p, int q){
    int rootP = find(p);
    int rootQ = find(q);
    if(rootP == rootQ)
        return;
    parent[rootP] = rootQ;
    //parent[rootQ] = rootP 也可以;
    count--;
}
```

我们一开始就简单的将p所在的树接到q所在的树下面，这样就也许会出现 头重脚轻的不平衡的状况如：



如果总是这样，那树也许生长得很不平衡，我们其实是希望，小一些的树接到大一些的树下面，这样就能避免头重脚轻，更平衡一些，方法是额外使用一个size数组，记录每棵树包含的节点个数，称为重量

```

class UF{
    int count;
    int[] parent;
    int[] size; // 新增一个数组记录树的“重量”
    UF(int n){
        this.count = n;
        parent = new int[n];
        size = new int[n];
        for(int i = 0; i < n; i++){
            parent[i] = i;
            size[i] = 1; // 最初每棵树只有一个节点 重量应该初始化 1
        }
    }
    /其他函数
}

```

比如size[3] = 5 表示以节点3为根的那棵树，总共有5个节点，这样可以修改一下union方法

```

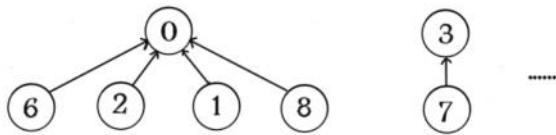
void union (int p , int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if(rootP == rootQ)
        return;
    if(size[rootP] > size[rootQ]{ // 小树接到大树下面，较平衡
        parent[rootQ] = rootP;
        size[rootP] += size[rootQ];
    }
    else{
        parent[rootP] = rootQ;
        size[rootQ] += size[rootP];
    }
    count--;
}

```

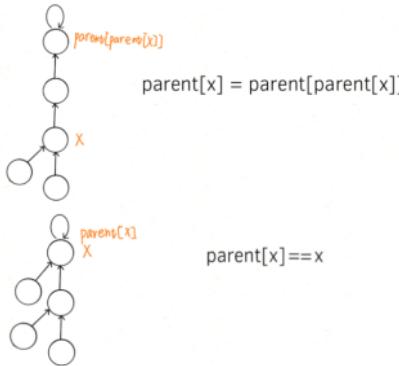
通过比较树的重量可以保证树的生长相对平衡，树的高度大致在 $\log N$ 这个数量级，极大的提高执行效率
同时也让find union connected 的时间复杂度都下降为 $O(\log N)$ 即使数量规模很大，时间也非常少

路径压缩

我们能不能压缩每棵树的高度，使树的高度始终保持为常数



这样find就可以以 $O(1)$ 的时间找到某一个节点的根节点，相应的connected union的复杂度都下降为 $O(1)$



只需要在find中加一行代码：

```

int find( int x){ 
    while(parent[x] != x){
        parent[x] = parent[parent[x]]; //进行路径压缩
        x = parent[x];
    }
    return x;
}

```

完整代码

```

class UF {
    // 连通分量个数
    private int count;
    // 存储一棵树
    private int[] parent;
    // 记录树的“重量”
    private int[] size;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }
}

```

```

public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootP == rootQ)
        return;

    // 小树接到大树下面，较平衡
    if (size[rootP] > size[rootQ]) {
        parent[rootQ] = rootP;
        size[rootP] += size[rootQ];
    } else {
        parent[rootP] = rootQ;
        size[rootQ] += size[rootP];
    }
    count--;
}

public boolean connected(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    return rootP == rootQ;
}

private int find(int x) {
    while (parent[x] != x) {
        // 进行路径压缩
        parent[x] = parent[parent[x]];
        x = parent[x];
    }
    return x;
}

public int count() {
    return count;
}
}

```

Union-Find 算法的复杂度可以这样分析：构造函数初始化数据结构需要 $O(N)$ 的时间和空间复杂度；
连通两个节点union、判断两个节点的连通性connected、计算连通分量count所需的时间复杂度均为 $O(1)$

算法的关键点有三个：

- 1.用parent数组记录每个节点的父节点，相当于指向父节点的指针，所以parent数组内实际存储着一个森林，若干棵多叉树
- 2.用size数组记录着每棵树的重量，目的是让union后的树依然拥有平衡性而不会退化为链表
- 3.infind函数中进行路径压缩，保证任意树的高度保持在一个常数，使用union connected API 时间复杂度为 $O(1)$

其实时间复杂度的话有了路径压缩就可以保证树的高度为常数，size数组不加也行，只是效率会更高一些

Union-Find算法的应用

```

990. 等式方程的可满足性
难度 中等 | 178 | 收藏 | 分享 | 切换为英文 | 捕获动态 | 反馈

给定一个由表示变量之间关系的字符串方程组组成的数组，每个字符串方程 equations[i] 的长度为 4，并采用两种不同的形式之一：“a==b” 或 “a!=b”。在这里，a 和 b 是小写字母（不一定不同），表示字母变量名。
只有当可以将参数分配给变量名，以便满足所有给定的方程时才返回 true，否则返回 false。
示例 1：

输入：["a==b","b!=a"]
输出：false
解释：如果同时指定 a = 1 且 b = 1，那么可以满足第一个方程，但无法满足第二个方程。应确保方法分配变量同时满足这两个方程。
示例 2：

输入：["b==a","a==b"]
输出：true
解释：我们可以指定 a = 1 且 b = 1 以满足这两个方程。
示例 3：

输入：["a==b","b==c","a==c"]
输出：true

```

```

1 class Solution {
2 public:
3     int parent[26];
4     bool equationsPossible(vector<string>& equations) {
5         for(int i = 0; i < 26; i++)
6             parent[i] = i; //根节点祖先指向自己
7         for(int i = 0; i < equations.size(); i++){
8             if(equations[i][1] == '='){
9                 int fa1 = find(equations[i][0] - 'a');
10                int fa2 = find(equations[i][3] - 'a');
11                if(fa1 != fa2)
12                    parent[fa1] = fa2; //和值连通分量 等于，合并集合
13            }
14        }
15        for(int i = 0; i < equations.size(); i++){
16            if(equations[i][1] == '!'){
17                int fa1 = find(equations[i][0] - 'a');
18                int fa2 = find(equations[i][3] - 'a');
19                if(fa1 == fa2)
20                    return false;
21            }
22        }
23        return true;
24    }
25    int find( int i){ //根据节点
26        if( i != parent[i]){
27            parent[i] = parent[parent[i]]; //顺便就路径压缩了
28            i = parent[i];
29        }
30        return i;
31    }
32 }
```

278. 第一个错误的版本

难度 简单 | 335 | 收藏 | 分享 | 反馈

你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以假设的版本之后的所有版本都是错的。

假设你有 n 个版本 [1, 2, ..., n]，你想找出导致之后所有版本出错的第一个错误的版本。

你可以通过调用 `bool isBadVersion(version)` 接口来判断版本号 `version` 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少调用 API 的次数。

示例：

给定 $n = 5$ ，并且 $version = 4$ 是第一个错误的版本。

调用 `isBadVersion(3) -> false`
 调用 `isBadVersion(5) -> true`
 调用 `isBadVersion(4) -> true`

所以，4 是第一个错误的版本。

```

1 // The API isBadVersion is defined for you.
2 // bool isBadVersion(int version);
3
4 class Solution {
5 public:
6     int firstBadVersion(int n) {
7         int left = 1, right = n;
8         while(left < right){
9             int mid = left + (right - left) / 2;
10            if(isBadVersion(mid))
11                right = mid;
12            else
13                left = mid + 1; // 此时有 left == right，区间端为一个点，即为答案
14        }
15        return left;
16    }
17 }
```

877. 石子游戏

难度 中等 ★ ★ ★ ★

亚历克斯和李用几堆石子玩游戏。偶数堆石子排成一行，每堆都有正整数颗石子 piles[i]。

游戏以谁手中的石子多来决定胜负。石子的总数是奇数，所以没有平局。

亚历克斯先手进行。亚历克斯先开始，每回合，他要从行的开始阶段选取石子堆的石头。这种情况下一直持续到没有更多的石子堆为止，此时手中石子最多的人就是获胜者。

假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 true，否则返回 false。

示例：

```
输入：[3,3,4,5]
输出：true
解释：
亚历克斯先开始，只能拿前 3 颗或后 5 颗石子。
假设他也拿了前 5 颗，这一行就变成了 [3,4,5]。
如果李拿走前 3 颗，那么剩下的是 [4,5]，亚历克斯拿走后 4 颗赢得 10 分。
如果李拿走后 4 颗，那么剩下的是 [3,4]，亚历克斯拿走后 4 颗赢得 9 分。
这表明，取第 5 颗石子对亚历克斯来说是一个胜利的举动，所以他返回 true。
```

```
/*
 * 从最左边的石子拿走 i 颗，就是每回合不操作石子，是最右边的石子 - 1 手直接被消灭。
 * 但是这个函数返回的是选择操作。
 * 如果李拿走前 i 颗，dp[i][j] = max(piles[i], dp[i+1][j]);
 * 但是这样要记录上一层操作的信息。
 *
 * 再试了一次，从最左边拿走 i 颗或后 i 颗。
 * Random root TreeNode;
 * @param n int
 * @return TreeNode
 */
public TreeNode cyclicShiftTree(TreeNode root, int n) {
    if(root==null) return null;
    LinkedList<TreeNode> pre = new LinkedList<>(); // // 1. 不存null值
    LinkedList<TreeNode> cur = new LinkedList<>(); // // 2. 但null值，空闲是pre的指向
    pre.addLast(root);
    while (!pre.isEmpty()){
        int size = pre.size();
        for (int i = 0; i < size; i++) {
            TreeNode temp = pre.pollFirst();
            pre.addLast(temp); // // 3. 但i为right
            cur.addLast(temp.right); // // 4. right指向left
            cur.addLast(temp.left);
        }
        int count = K*cur.size(); // // 5. K=30000, 但实际上不用操作太多，只用操作会快
        for (int i = 0; i < count; i++) { // // next层右移K位
            TreeNode temp = cur.pollFirst();
            cur.addLast(temp);
        }
        for (int i = 0; i < size; i++) { // // 6. pre的next的重新连起来，将next指向null并指向pre
            TreeNode temp = pre.pollLast();
            temp.left = cur.pollLast();
            temp.right = cur.pollLast();
            if(temp.left!=null) pre.addFirst(temp.left);
            if(temp.right!=null) pre.addFirst(temp.right);
        }
        cur.clear();
    }
    return root;
}
```

博主生窝@窝居

层序遍历构造二叉树：

```
TreeNode* build(vector<int> arr){
    queue<TreeNode*> q;
    if(arr.empty()) return nullptr;
    TreeNode* head = new TreeNode(arr[0]);
    q.push(head);
    TreeNode* node;
    int i = 1;
    while(!q.empty()){
        node = q.front();
        q.pop();
        if(i<arr.size()){
            node->left = new TreeNode(arr[i]);
            q.push(node->left);
            i++;
        }
        else
            node->left = nullptr;
        if(i<arr.size()){
            node->right= new TreeNode(arr[i]);
            q.push(node->right);
            i++;
        }
        else
            node->right = nullptr;
    }
    return head;
}
```

加油

2021年2月9日 8:18

```
-----  
55. 跳跃游戏  
难度 中等 通过 1311 提交 0%  
给定一个非负整数数组 nums，你最初位于数组的第一个下标。  
数组中的每个元素代表你在该位置可以跳跃的最大步数。  
判断你是否能够到达最后一个下标。  
  
示例 1：  
输入：nums = [2,3,1,1,4]  
输出：true  
解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。  
示例 2：  
输入：nums = [3,2,1,0,4]  
输出：false  
解释：无论怎样，总到达下标为 3 的位置。但该下标的最大跳跃长  
度是 0，所以永远不会到达最后一个下标。
```

动态规划+正推

用 i 表示位置，是否可达，初始的时候都是 0，只有 $dp[i] = 1$ ，因为起点一定是可达的，**这个初始条件在 dp 定义完一定要记得写，否则 dp 报错**

然后从位置 $i+1$ 开始遍历。对于位置 i ，如果发现 $dp[i] = 0$ ，那么此时从前面的位置无法到达它，所以直接返回 `false`

否则的话，它能到达的范围是 $i+1$ 到 $i+nums[i]$ ，所以把这部分的 dp 值都标记为 1

如果在从前往后遍历的过程中发现 当前位置 $i + nums[i] >= n - 1$ ，就说明当前位置直接就能跳到终点了，直接返回 `true`。

下标 0 1 2 3 4
vec[i] 2 3 1 1 4
dp[i] 1 1 1 i=2 i+nums[i] >= n-1 即剩余位置都可达

```
72. 爬楼梯  
难度 困难 通过 1763 提交 0%  
给定两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所需的最少操作数。  
你可以对一个单词进行如下三种操作：  
• 插入一个字符  
• 删除一个字符  
• 替换一个字符  
  
示例 1：  
输入：word1 = "horse"，word2 = "ros"  
输出：3  
解释：  
horse -> horse (将 'h' 替换为 'r')  
horse -> rose (删除 'r')  
rose -> ros (删除 'e')  
  
示例 2：  
输入：word1 = "intention"，word2 = "execution"  
输出：5
```

想象字符串从 a -----> abc

从 a -----> aba 那最后一个字符相等，则 $dp[i][j] = dp[i-1][j-1]$ 否则就可以参与替换 可以加 rep

$dp[i][j]$ 表示 $Str1..i$ -----> $Str2..j$ 需要的替换次数

那 $dp[0][0]$ 只能 inc j 个元素 $dp[0][0]$ 只能 del i 个元素 处理好初值的意义 循环再从 $i = 1$ $j = 1$ 开始 str[0] 对应 $dp[1]$ 第一个字符，str[4] 对应 $dp[5]$ 第 5 个字符

因此判断最后一个字符相等应该是

条件 $str1[i-1] == str2[j-1]$ $dp[i][j] = dp[i-1][j-1]$ $str1[0] == str2[0]$ 对应 $str1$ $str2$ 第一个字符相等 $dp[1][1] = dp[i-1][j-1]$

```
15. 不同子序列  
难度 困难 通过 574 提交 0%  
给定一个字符串 s 和一个字符串 t，计算在 s 的子序列中 t 出现的个数。  
字符串的一个子序列 指：通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，“ace” 是 “abcde”的一个子序列，而 “aec” 不是）  
题目数据保证答案符合 32 位带符号整数的范围。  
  
示例 1：  
输入：s = "rabbbit", t = "rabbit"  
输出：3  
解释：  
如上图所示，有 3 种方式从 s 中得到 "rabbit" 的方案。  
rabbbit  
rabbbit  
rabbbit  
  
示例 2：  
输入：s = "babgbag", t = "bag"  
输出：5  
解释：  
...bb bgg ----- bg  
12 345
```

到 2：取 b 345...中找 g ----- 到 4 取 g --- 返回

不取 g --- 到 5 中找 g

不取 b 345...中找 bg ----- 取 b --- 到 4 取 g --- 返回

不取 g --- 到 5 中取 g 与上面出现了重复计算 因为递归是两部分相加 前面那一次算出来了 右边这一加项就不要再算了---- 转成动态规划问题

确定 dp 数组（dp table）以及下标的含义： $dp[i][j]$ ：以 $i-1$ 为结尾的 s 的子序列中出现以 $j-1$ 为结尾的 t 的个数为 $dp[i][j]$

这一类问题，基本是要分析两种情况 $s[i-1]$ 与 $t[j-1]$ 相等 $s[i-1]$ 与 $t[j-1]$ 不相等

当 $s[i-1]$ 与 $t[j-1]$ 相等时， $dp[i][j]$ 可以由两部分组成。

一部分是用 $s[i-1]$ 来匹配，那么个数为 $dp[i-1][j-1]$ 一部分是不用 $s[i-1]$ 来匹配，个数为 $dp[i-1][j]$

从递推公式 $dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$ ；和 $dp[i][j] = dp[i-1][j]$ ；中可以看出 $dp[i][0]$ 和 $dp[0][j]$ 是一定要初始化的 根据 dp 的定义或者初始 j 值判断 不能凭感觉 $dp[i][0]$ 从 s 中找到空串的方法：一种 $dp[0][i]$ 从 0 个字符中找到长为 i 的 t : 0 种方法 注意这里 $dp[0][0]$ 是等于 1 的 0 个字符找空串有 1 种方法

确定遍历顺序：从递推公式 $dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$ ；和 $dp[i][j] = dp[i-1][j]$ ；中可以看出 $dp[i][j]$ 都是根据左上方和正上方推出来的

```

class Solution {
public:
    int maxDistinct(string s, string t) {
        int n = s.size(), m = t.size();
        vector<vector<long>> dp(m + 1, vector<long>(n + 1));
        for (int i = 0; i <= n; ++i) dp[0][i] = 1;
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (s[i - 1] == t[j - 1]) dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1];
                else dp[i][j] = dp[i - 1][j];
            }
        }
        return dp[m][n];
    }
};

```

这道题我还是会选择用记忆DFS方法.....

124. 二叉树中的最大路径和

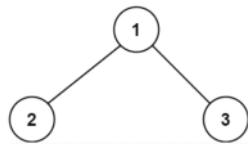
难度 困难 ⚡ 1171 ☆ 🔍 0 0

路径 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。一个节点在一条路径序列中至多出现一次。该路径 未必包含一个节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

给你一个二叉树的根节点 root，返回其 最大路径和。

示例 1：



输入：root = [1,2,3]

输出：6

解释：最优路径是 2 → 1 → 3，路径和为 2 + 1 + 3 = 6

-10

-5

-15

典型的后序遍历框架 用一个全局变量记录返回前的maxn

从节点 -5 和节点 -15 看起

-10 的 left 等于 -5 小于 0 应该不取 设为0

20的left = -15 小于 0 不取 7 取计算maxn 0 + 20 + 7 保存到全局区 计算完此时 20节点再返回 返回的应该是较大分支 20+7 作为-10节点的right

-10的left已经有了计算maxn = -10 + left + right 判断是否更新maxn -10节返回的也是-10这个节点+较大分支(max left, right) 作为上层节点的分支返回值参与上面的计算过程

根据后序遍历的特点以及考虑到有负数的情形 应该返回的是较大分支给上层计算而还是左边加右边，注意返回前参与计算maxn的left right要大于0才取它 计算结果返回给上一层函数使用

4. 寻找两个正序数组的中位数

难度 困难 ⚡ 4454 ☆ 收藏 🔍 分享 🌐 切换为英文 🌐 换收动态 🔍 反馈

给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2，请你找出并返回这两个正序数组的 中位数。

示例 1：

输入：nums1 = [1,3], nums2 = [2]

输出：2.00000

解释：合并数组 = [1,2,3]，中位数

示例 2：

输入：nums1 = [1,2], nums2 = [3,4]

输出：2.50000

解释：合并数组 = [1,2,3,4]，中位数 (2 + 3) / 2 = 2.5

```

1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11}
12
13 class Solution {
14 public:
15     int maxn = INT_MIN;
16     int maxPathSum(TreeNode* root) {
17         if(root == nullptr)
18             return maxn;
19         int dfa(TreeNode* root){
20             if(root == nullptr)
21                 return 0;
22             int left = max(0, dfa(root->left));
23             int right = max(0, dfa(root->right));
24             maxn = max(maxn, root->val + left + right);
25             return root->val + max(left , right);
26         };
27     };

```

-10

-5

-15

典型的后序遍历框架 用一个全局变量记录返回前的maxn

从节点 -5 和节点 -15 看起

-10 的 left 等于 -5 小于 0 应该不取 设为0

20的left = -15 小于 0 不取 7 取计算maxn 0 + 20 + 7 保存到全局区 计算完此时 20节点再返回 返回的应该是较大分支 20+7 作为-10节点的right

-10的left已经有了计算maxn = -10 + left + right 判断是否更新maxn -10节返回的也是-10这个节点+较大分支(max left, right) 作为上层节点的分支返回值参与上面的计算过程

根据后序遍历的特点以及考虑到有负数的情形 应该返回的是较大分支给上层计算而还是左边加右边，注意返回前参与计算maxn的left right要大于0才取它 计算结果返回给上一层函数使用

```

1 class Solution {
2 public:
3     double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
4         vector<int> nums;
5         int size1 = nums1.size(), size2 = nums2.size();
6         int n = size1 + size2;
7         nums.resize(n, 0);
8         int i1 = 0, i2 = 0, i = 0;
9         while(i1 < size1 || i2 < size2){ // 合并两个有序数组的过程
10             if(i2 >= size2 || (i1 < size1 && nums1[i1] < nums2[i2] ))
11                 nums[i++] = nums1[i1++];
12             else
13                 nums[i++] = nums2[i2++];
14
15         if(n % 2 == 0)
16             return ( (double)nums[n/2] + (double)nums[n/2 - 1] )/2;
17         else
18             return (double)nums[n / 2];
19     }
20 };
21 };

```

174. 地下城游戏

难度 困难 ⚡ 503 ☆ 🔍 0 0

一些恶魔抓住了公主（P），并将她关在了地下城的右下角。地下城是由 M × N 个房间组成的二维网格。我们英勇的骑士（K）最初被安置在左上角的房间里，他必须穿过地下城并到达公主所在的位置。

骑士的初始健康点数为一个整数。如果他的健康点数在某一时刻降至 0 或以下，他会立即死亡。

有些房间由恶魔守卫，因此骑士在进入这些房间时会失去健康点数（若房间里的值为负数，则表示骑士将损失健康点数）；其他房间要么是空的（房间里的值为 0），要么会增加骑士健康点数的魔法球（若房间里的值为正数，则表示骑士将增加健康点数）。

为了尽快到达公主，骑士决定每次只向右或向下移动一步。

编写一个函数来计算确保骑士能够拯救到公主所需的最低初始健康点数。

例如，考虑到如下布局的地下城，如果骑士遵循最佳路径 右 → 右 → 下 → 下，则骑士的初始健康点数至少为 7。

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

```

1 class Solution {
2 public:
3     int calculateMinimumHP(vector<vector<int>> &dungeon) {
4         int n = dungeon.size(), m = dungeon[0].size();
5         vector<vector<int>> dp(n + 1, vector<int>(m + 1, INT_MAX));
6         dp[n - 1][m - 1] = dp[n - 1][m] = 1;
7         for(int i = n - 1; i >= 0; i--){
8             for(int j = m - 1; j >= 0; j--){
9                 int minx = min(dp[i + 1][j], dp[i][j + 1]);
10                //dp[i][j] = min - dungeon[i][j]; dp[i][j]表示从 i, j 走到右下角需要的最小生命值 dungeon[i][j] 表示当前格子的损耗值
11                dp[i][j] = max (minx - dungeon[i][j], 1); //每个格子的需要的最小生命值为1
12            }
13        }
14        return dp[0][0];
15    }
16 };

```

从后往前遍历可以知道最终到达右下角需要的生命值，再通过每个格子的损耗可以倒推出前面每个格子需要的最小生命值，这样就不要考虑路径和的问题（如果从前往后遍历的话，相当于不知道最终的最小生命值，只知道最终到达的生命值大小应该大于1，需要记录到当前格子的路径和和到当前格子需要的最小生命值，因为不知道初始的生命值，只能用路径和和损耗来确定到当前格子需要的最小生命值）

注意初值 其他可以为INT_MAX 因为递推需要用到右下两个元素

dp[n-1][m-1]转移需要用到的dp[n-1][m]和 dp[n][m-1]均为无效值，给这两个值赋值为 1 计算出最终到达右下角需要的最小生命值 1 - dungeon[n-1][m-1] dp[0][0]

```

1 -3 3 *
0 -2 0 *
-3 -3 -3 *
* * * *

min(9,6)----4(第二行)
10----7 ----4
2 4-2取2 如果是5 , 4-5 = -1 也要取1 最小生命值为1

```

213. 打家劫舍 II

难度 中等 753 收藏 分享 模拟面试

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，这家所有的房屋都 围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果相邻两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放的金额的非负整数数组，计算你 在不触动警报装置的情况下，今晚能够偷窃的最大金额。

示例 1：

```

输入：nums = [2,3,2]
输出：3
解释：不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为它们是相邻的。

```

22 };

第一个和最后一个只能选一个偷，如果偷第一个计算下标0,size-2内的最大金额，如果偷最后一个，计算1, size-1内的最大金额，再取两种情况的最大值，注意处理第一个元素与第二个元素的初始值 也就是只有两个元素的情形

233. 数字 1 的个数

难度 困难 321 收藏 分享 切换为英文 接收动态 反馈

给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

示例 1：

```

输入：n = 13
输出：6

```

for i从1开始， $i <= n \quad i *= 10$;
首位出现的1个数为 $(n / (i * 10)) * i$
剩余的1个数为 $\max(\max(n \% (i * 10) - i + 1, 0), i)$

2000 每10个数 个位出现1个1
2001 每100个数 十位出现10个1
2002 每1000个数 百位出现100个1
...
2010
2011
...
2019
2020
2021

很简单，每 100 个数会出现 10 个十位数为 1 的数字，同样地，如果 n 的后面两位小于 10，则不用额外加次，如果后两位大于等于 10，则需要额外加次。
比如，n=2021 时，最后要加 10 次，n=2009 时，最后不要加 10 次，而 n=2015 时，最后要加 15-10+1=6 次，这一块，自己体会一下。
同样道理，可以推断出位数出现多少个 1，就很简单了，用公式将一表示为 (n 表示题目指定的参数，为统计哪位上的1)：
count = $(n / (1 * 10)^k) * k$ ， k 算的数量就要看 i 及其右边的位数，即 $n \% (i * 10)$ (记为 x)，是小于 i、大于等于 i，真大了多少：
• $x < i, k = 0$
• $i <= x < i + 1, k = x - i + 1$
• $x > i + 1, k = 1$
写成一行： $k = \min(\max(x - i + 1, 0), 1)$ ，请仔细体会。

完整公式为：count = $(n / (1 * 10)^k) * k + \min(\max(n \% (i * 10) - i + 1, 0), 1)$ 。

有了公式，我们很快就能计算出来 n = 2821 时，各位数一共会出现 $2 * 100 + \min(\max(21-100+1, 0), 100)=200$ 个1了，它们分别是 100,101,...,199,1100,1101,1109。

暴力枚举方法：遍历 1 ~ n 的所有数字，统计各数字里面 K 出现的个数.....

```

class Solution {
public:
    int countDigitOne(int n) {
        return CountProblem(n, 1);
    }

    int CountProblem(int n, int x) {
        int cnt = 0;
        for (int i = 1; i <= n; ++i) {
            int s = i;
            while (s > 0) {
                if (s % 10 == x) ++cnt;
                s /= 10;
            }
        }
        return cnt;
    }
};

```

```

1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         if(nums.size() == 0) return 0;
5         if(nums.size() == 1) return nums[0];
6         if(nums.size() == 2) return max(nums[0], nums[1]); //两个元素的情况单独处理返回 下面的就是size>3的情形
7         int res1 = robber(nums, 0, nums.size() - 2);
8         int res2 = robber(nums, 1, nums.size() - 1);
9         return max(res1, res2);
10    }
11    int robber(vector<int>& nums, int start, int end) {
12        // if (start == end)
13        // return nums[start]; //恰好有两个元素 上面执行函数 robber(nums, 0, 0) robber(nums, 1, 1); 下面开的是dp(2,0)
14        vector<int> dp(nums.size(), 0); //size为2 dp[2] 越界
15        dp[start] = nums[start];
16        dp[start+1] = max(nums[start], nums[start+1]); //size为3: robber(nums,0,1) robber(nums,1,2)都不回越界
17        for(int i = start+2; i <= end; ++i){
18            dp[i] = max(dp[i-2] + nums[i], dp[i-1]);
19        }
20        return dp[end];
21    }
22 };

```

```

1 class Solution {
2 public:
3     int countDigitOne(int n) {
4         double ans = 0;
5         for (int i = 1; i <= n; i *= 10) {
6             ans += (n / (i * 10)) * i * min(max(n % (i * 10) - i + 1, 0), i);
7         }
8         return ans;
9     }
10 };

```

312. 破气球

难度 困难 ✓ 777 收藏 分享 为切换为英文 抢先动态 反馈

有 n 个气球，编号为 0 到 $n - 1$ ，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。数据量 1 个气球，你可以获得 $\text{nums}[i - 1] * \text{nums}[i] * \text{nums}[i + 1]$ 枚硬币。这里的 $i - 1$ 和 $i + 1$ 代表和 i 相邻的两个气球的序号。如果 $i - 1$ 或 $i + 1$ 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

示例 1：

```
输入: nums = [3,1,5,8]
输出: 167
解释:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5 + 3*5*8 + 1*5*8 + 1*8*1 = 167
```

示例 2：

```
输入: nums = [1,5]
输出: 10
```

```
1 class Solution {
2 public:
3     vector<vector<int>> dp;
4     int solve(vector<int>& nums, int i, int j)
5     {
6         if(i > j)
7             return 0;
8         // 使用记忆化的值，避免重复计算
9         if(dp[i][j] > 0)
10            return dp[i][j];
11         for (int k = i; k <= j; k++)
12         {
13             int l = solve(nums, i, k - 1);
14             int r = solve(nums, k + 1, j);
15             int delta = nums[k] * nums[i - 1] * nums[j + 1];
16             dp[i][j] = max(dp[i][j], l + r + delta);
17         }
18         return dp[i][j];
19     }
20     int maxCoins(vector<int>& nums) {
21         int n = nums.size();
22         // 对nums数组进行操作
23         nums.insert(nums.begin(), 1);
24         nums.push_back(1);
25         // 初始化dp数组
26         dp = vector<vector<int>>(n + 2, vector<int>(n + 2, 0));
27         int ans = solve(nums, 1, n);
28         return ans;
29     }
}
```

对于区间 $[l, r]$ ，我们考虑最后一个被戳破的气球 k ，那么之前的步骤我们可以分为两步，也就是求 $[l, k-1]$ 和 $[k+1, r]$ 之间的最大分数

下标： 0 1 2 3 4 5

元素值 x 3 4 5 9 x

考虑区间 1 ~ 4 内 最后戳爆的气球是 k 从左边开始到右尾端结束

区间内最后戳爆的气球为 1 则 最后剩余 0 1 5 (2 0 5)

2 ~ 4 内 最后那个为 2 剩 1 2 5 ...

3 ~ 4 内 最后那个为 3 剩 2 3 5 ...

4 单个区间 最后那个为 4 剩 3 4 5

$dp[i][j]$: 从 i 到 j 的气球闭区间内能够获得的最大的分数

这就是依次考虑区间内最后戳爆的那个气球是区间内第几个，再根据左边的分数和右边的分数累加再加上当前的增量 往上返回

假设区间就是 2 3 4 只有这三个数

最后那个为 3 计算到最后选 $k=3$ 气球的增量分数 再加左边 气球的分数 如左边气球分数就是 2 再加右边气球的分数 就是 4 总的分数就是当前最后一个气球再加该区间之前/左右的分数

状态转移方程为：区间 dp 该区间内最后一个气球的分数 * 左右元素 那就是这个区间的邻近元素了 / 该区间内已经没有气球

$dp[i][j] = dp[i][k-1] + dp[k+1][j] + \text{nums}[i-1] * \text{nums}[k] * \text{nums}[j+1]$ ，其中 $i <= k <= j$

这个就是用相当于带记忆的 dp table + DFS 递归方法

可是我还是觉得回溯更好理解.....

```
ss Solution {
    lic:
    void solve(vector<int>& nums, int count, int& ans)
    {
        // 边界条件
        if(nums.size() == 0)
        {
            ans = max(ans, count);
            return;
        }

        for(int i = 0; i < nums.size(); i++)
        {
            int t = nums[i];
            // delta -> 增量
            int delta = nums[i] * (i <= 0 ? 1 : nums[i - 1]) * (i >= nums.size() - 1 ? 1 : nums[i + 1]);
            nums.erase(nums.begin() + i);
            solve(nums, count + delta, ans);
            nums.insert(nums.begin() + i, t);
        }
    }

    int maxCoins(vector<int>& nums) {
        int ans = 0;
        solve(nums, 0, ans);
        return ans;
    }
}
```

337. 打家劫舍 III

难度 中等 ✓ 546 收藏 分享 0 反馈

在一次打劫完一条街道之后和一盖房子后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子只有有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房子都排列类似于一棵二叉树”。如果两个直接相连的房子在同一晚上被打劫，房屋所有者会报警。

计算在不触动警报的情况下，小偷一晚能盗取的最大金额。

示例 1：

输入: [3,2,3,null,3,null,1]

```
      3
     / \
    2   3
     \   \
      1   3
```

输出: 7

解释: 小偷一晚最多能盗取的最高金额 = 3 + 3 + 1 = 7.

示例 2：

输入: [3,4,5,1,3,null,1]

```
2 * Definition for a binary tree node.
3 * struct TreeNode {
4     int val;
5     * TreeNode *left;
6     * TreeNode *right;
7     * TreeNodes *val(0), left(nullptr), right(nullptr) {}
8     * TreeNodes(TreeNode *val(x), TreeNodes *left(nullptr), TreeNodes *right(nullptr) {}
9     * TreeNodes(TreeNode *x, TreeNodes *left, TreeNodes *right) : val(x), left(left), right(right) {}
10    * };
11
12 class Solution {
13 public: // 定义一个pair 来描述当前节点 p->first 当前节点能获得的最大金额 p->right 当前节点不能获得的最大金额
14     pair<int, int> robTree(TreeNode* root) {
15         if(root == nullptr)
16             return {0, 0};
17         // 返回值 (dfs(root) ->first, dfs(root) ->second); pair<int, int> is not a pointer, you must to use -
18         return max(dfs(root).first, dfs(root).second);
19     }
20     typedef pair<int, int> pii;
21     pii dfs(TreeNode* root){
22         pii p = dfs(root);
23         // p.first<int, int> p;
24         if(root == nullptr)
25             return {0, 0}; // 根节点是dfs遍历 那么肯定要有递归终止的条件
26         pii l = dfs(root->left);
27         pii r = dfs(root->right);
28         int r0 = l.first + r.second + root->val; // r0 p->first 挑当前节点 能获得的最大利润
29         int r1 = max(l.first, l.second) + max(r.first, r.second);
30
31         return {r0, r1};
32     }
}
```

上面是通过返回值将当前节点的 偷与不偷 两种情况能获得的最大利润依次返回给上层

也可以定义两个全局map 记录当前的节点偷与不偷 两种情况能获得的最大利润

最终返回到最上层节点 所有节点的状态返回记录已经完成 此时记录就是的那个root节点 这个节点偷与不偷 能获得利润的最大值 取max

这种记录方法更好理解 ...

```

class Solution {
public:
    unordered_map<TreeNode*, int> f, g;

    void dfs(TreeNode* node) {
        if (!node) {
            return;
        }
        dfs(node->left);
        dfs(node->right);
        f[node] = node->val + g[node->left] + g[node->right];
        g[node] = max(f[node->left], g[node->left]) + max(f[node->right], g[node->right]);
    }

    int rob(TreeNode* root) {
        dfs(root);
        return max(f[root], g[root]);
    }
};

```

对二叉树做了一次后序遍历时间复杂度为O(N) 递归用到栈空间 空间复杂度为O(N) 哈希表的空间代价也是O(N) 所以空间复杂度也是O(N)
因为对于每个节点，我们只关心它的孩子节点们的f和g是多少，因此可以用一个pair表示某个节点的f和g值，在每次递归返回的时候，都把这个点对应的f和g返回给上一级调用，这样就节省了哈希表使用的空间，虽然空间复杂度不变但是性能得到了提高

354. 俄罗斯套娃信封问题
难度 困难 ⌂ 570 ☆ ⓘ ⓘ ⓘ

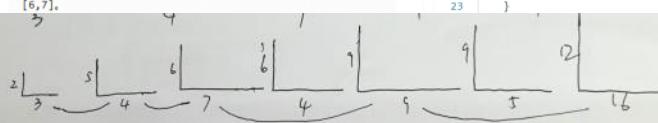
给你一个二维整数数组 envelopes，其中 envelopes[i] = [w_i, h_i]，表示第 i 个信封的宽度和高度。当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算 最多能有多少个 信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

注意：不允许旋转信封。

示例 1：

输入：envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出：3
解释：最多信封的个数为 3，组合为：[2,3] => [5,4] => [6,7]。



先对区间矩形长度排序，这样较小的就能装入较大的 长度相等的信封 按宽度降序排序 防止相同长度的信封重叠在一起
这样就是一个对宽度求最长上升序列问题

376. 摆动序列
难度 中等 ⌂ 486 ☆ ⓘ ⓘ ⓘ

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为 摆动序列。第一个差（如果存在的话）可能正数或负数。仅有一个元素或者两个不等元素的序列也视作摆动序列。

• 例如，[1, 7, 4, 9, 2, 5] 是一个 摆动序列，因为差值 (6, -3, 5, -7, 3) 是正负交替出现的。

• 相反，[1, 4, 7, 2, 5] 和 [1, 7, 4, 5, 5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它最后一个差值为零。

子序列 可以通过从原始序列中删除一些（也可以不删除）元素来获得，剩下的元素保持其原始顺序。

给你一个整数数组 nums，返回 nums 中作为 摆动序列 的 最长子序列 的长度。

虽然说是从局部最优推导全局最优，但是贪心算法其实就像一种常识性的理解和推导，证明只能用数学归纳法和反证法证明

390. 消除游戏
难度 中等 ⌂ 120 ⓘ ⓘ ⓘ

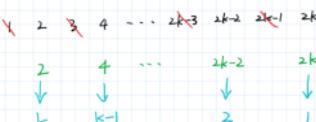
给出一个从 1 到 n 排成的魔法矩阵。
首先从左到右，从第一个数字开始，每隔一个数字进行删除，直到剩下的半数。
第二步，在剩下的数字中，从右到左，从倒数第一个数字开始，每隔一个数字进行删除，直到剩下下一个数字。
我们不断重复这两步，从左到右和从右到左交替进行，直到只剩下下一个数字。

返回长度为 n 的列表中，最后剩下的数字。

示例：

输入：
n = 9,
1 2 1 4 5 6 2 8 2
2 5 6 8
2 6
6

输出：



如果 n = 2k，那么如上图所示，第一轮消除了之后，剩下的数字就是绿色的偶数部分。

接着我们用 f(2k) 表示初始时 n = 2k 个数字最后剩下的编号，那么现在部分重新编号后最后剩下的数字还是 f(2k)，但是这次将 f(2k) 重新映射到绿色的偶数数字编号呢？

通过观察我们可以发现，绿色数字翻转 2，再长一点还能映射到它的编号，结果一定等于 k + 1，所以刚刚推倒了映射回去的公式：

$$f(2k) = 2(k+1 - f(k))$$

如果 $n = 2k$ ，那么从左往右遍历完成后，剩下的数字就是绿色的偶数部分。

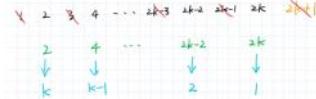
接着就要从右往左遍历地遍历了，把它们从右往左给绿色数字重新编号为 1 到 k ，问题就转化为为了 k 个数字的情况下，最后剩余的数字是几了。

假设我们用 $f(2k)$ 表示的是 $n = 2k$ 个数字最后剩下的编号，那么你必然分步到编号后最后剩下奇数的自然是 $f(k)$ ，但能怎么将 $f(k)$ 重新映射回绿色的偶数呢呢？

通过观察我们可以发现，绿色数字都是 2，再加上蓝色的剩余的编号，结果一定等于 $k + 1$ ，所以我们就能得到映射回去的公式：

$$f(2k) = 2(k + 1 - f(k))$$

比如你不知道 $f(k) = 2$ ，也就是绿色部分最后剩下的数字是 2，那么映射时绿色的偶数就是 $2k - 2$ ，这就是最初的映射了。



如果 $n = 2k + 1$ ，那么地上画所示，只需要在右面的偶数的 $2k + 1$ 就行了。

能看到一般的解法已经跳过了，所以绿色的映射规则和之前偶数的情况没有任何区别。所以之前的答案出来：

$$f(2k + 1) = 2(k + 1 - f(k))$$

找规律问题..... 约瑟夫环问题是 从开始位置 1 起每次删除第 m 个元素 递推公式为 i 从起始位置 2 遍历到第 n 个人 $(res + m) \% i$ 1 个人 res 刚刚开始是 0 位置 $m = 2$

$res = (res + 2) \% 2$ 若 2 个人 $res = (0 + 2) \% 2 = 0$ 继续 递推 $res = ((res + m) \% i + m) \% i$ 一直到 $i == n$ 2 个人为 0, 1 的位置编号 0 n 个人为 $0 \dots n - 1$ 其中的一个位置编号对应如果是位置编号从 1 开始 2 个人对应位置 0 0 1 2 3 4 5 6 7 ----> 1 2 3 4 5 6 7

....

689. 三个不重叠子数组的最大和

难度 **困难** ⚡ 131 收藏 分享 切换为英文 接收动态 反馈

给定数组 $nums$ 由整数组成，找到三个互不重叠的子数组的最大值。

每个子数组的长度为 k ，我们要使这 $3 \times k$ 项的和最大化。

返回每个区间起始索引的列表（索引从 0 开始）。如果有多个结果，返回字典序最小的一个。

示例：

输入: [1,2,1,2,6,7,5,1], 2

输出: [0, 3, 5]

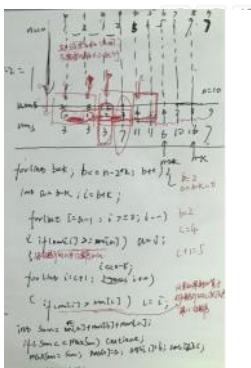
解释: 子数组 [1, 2], [2, 6], [7, 5] 对应的起始索引为 [0, 3, 5]。

我们也可以取 [2, 1]，但是结果 [1, 3, 5] 在字典序上更大。

注意:

- $nums.length$ 的范围在 $[1, 20000]$ 之间。
- $nums[i]$ 的范围在 $[1, 65535]$ 之间。
- k 的范围在 $[1, \lfloor \text{nums.length} / 3 \rfloor]$ 之间。

通过次数 3,046 提交次数 6,249



用暴力遍历区间的方法，先转成区间累加和，再从 $b=k$ 开始遍历查找累加和最大区间，注意左右开始的范围应该考虑区间的长度这个长度内的元素是在一起的，共同占据组成累加和区间

```
int priority(char bracket){  
    switch(bracket){  
        case '(':  
            return 1;  
        case '[':  
            return 2;  
        case ')':  
            return 3;  
        default:  
            return -1;  
    }  
}  
  
if(!stk.empty() && priority(stk.top())<priority(*ptr)){  
    cout<<"priority error!"<<endl;  
}
```

907. 子数组的最小值之和

难度 **中等** ⚡ 260 收藏 分享 切换为英文 接收动态 反馈

给定一个整数数组 arr ，找到 $\min(b)$ 的总和，其中 b 的范围为 arr 的每个（连续）子数组。

由于答案可能很大，因此返回答案模 $10^9 + 7$ 。

示例 1：

输入: arr = [3,1,2,4]
输出: 17
解释:

子数组为 [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4]，
[3,1,2,4]。

最小值为 3, 1, 2, 4, 1, 1, 2, 1, 1, 和为 17。

示例 2：

以 $nums[i]$ 作为最小值的子数组的数量 如 3 1 4 : 1 3 1 4 3 1 4 关键即为找到每个 $A[i]$ 对应的区间，满足 $A[i]$ 为该区间内的最小值

$(m+1) \times (n+1)$; m, n 为左右元素的个数 遍历每一个元素，找到以当前元素为最小值的区间有多少个 $_n$ $A[i]$ $_m$

公式 or 规律： $A[i]$ 左侧有 n 个连续的大于 $A[i]$ 的数，右侧有连续 m 个大于 $A[i]$ 的数，则 $A[i]$ 作为最小值的数组个数为 $(m+1) \times (n+1)$ 单调递增栈 尽量有多个连续大的数

```
1 class Solution {  
2 public:  
3     const int BASE = 1e9 + 7;  
4     int sumSubarrayMins(vector<int>& arr) {  
5         stack<int> stk; // 定义单调递增的栈 存放数组的下标  
6         arr.push_back(0); // 保证栈中所有元素都会被弹出计算  
7         int n = arr.size();  
8         long res = 0; // 2 3 1 3 4 2 0  
9         for(int i = 0; i < n; i++){  
10             while(!stk.empty() && arr[stk.top()] >= arr[i]){ // 来了一个小元素  
11                 int index = stk.top();  
12                 stk.pop();  
13                 int preindex = -1; if(!stk.empty()) preindex = stk.top();  
14                 int l_cnt = index - preindex - 1;  
15                 int r_cnt = i - index - 1; // 右边元素的数量  
16                 res += long(arr[index]) * (r_cnt + 1) * (l_cnt + 1) % BASE;  
17                 res = res % BASE;  
18             }  
19             stk.push(i);  
20         }  
21     }  
22 }  
23 }
```

```
1 class Solution {  
2 public:  
3     const int BASE = 1e9 + 7;  
4     int sumSubarrayMins(vector<int>& arr) {  
5         stack<int> stk; // 定义单调递增的栈 存放数组的下标  
6         arr.push_back(0); // 保证栈中所有元素都会被弹出计算  
7         int n = arr.size();  
8         long res = 0; // 2 3 1 3 4 2 0  
9         for(int i = 0; i < n; i++){  
10             while(!stk.empty() && arr[stk.top()] >= arr[i]){ // 来了一个小元素  
11                 int index = stk.top();  
12                 stk.pop();  
13                 int preindex = -1; if(!stk.empty()) preindex = stk.top();  
14                 int l_cnt = index - preindex - 1;  
15                 int r_cnt = i - index - 1; // 右边元素的数量  
16                 res += long(arr[index]) * (r_cnt + 1) * (l_cnt + 1) % BASE;  
17                 res = res % BASE;  
18             }  
19             stk.push(i);  
20         }  
21     }  
22 }  
23 }
```

如果出现了一个较小的数则触发计算子数组个数 1暂时不能入栈 直到计算完1是一个较大的数 否则直到为空再加入1

分别计算栈顶元素到前一个元素和到后一个元素之间有多少元素，这些元素肯定都比栈顶大 可以使用公式计算以A[i]为最小值的区间个数

考虑 2 3 1 3 4 2 0(补)

栈内元素 2 3 遇到小元素1 触发处理因为1不能入栈 处理栈中的元素维护单调递增

栈内元素 1 3 4 --- 3 4 出栈并处理 看它的前后有多少元素

栈内元素 1 2 在 5-2-1 有2个元素 右 6-5-1 有0个元素 2 * 3 * 1 = 342 : 2 * 42 = 342

特殊值考虑index = 0 执行pop(), preindex = -1;如果不为空 preindex 就是栈顶下一个下标 left = 0 - 1 + 1 = 0 个 右 i=2 2 - 0 - 1 = 1 个 23 : 2 * 1 * 2 = 2 * 23 两个

开始向arr push_back一个0 , 为了保证当栈内元素能全部弹出处理如 1 , 2 最终的top元素是肯定>= 0 , 所以都会弹出处理

每日温度那道题是维护单调递减的栈，这个求子数组最小值之和是维护单调递增的栈 找连续大于的个数此时左边就是大于这个数，右边也是大于这个数

1381. 设计一个支持批量操作的栈

难度 中等 51 收藏 分享 切换为英文 接收动态 反馈

请你设计一个支持下述操作的栈。

实现自定义类 CustomStack :

- CustomStack(int maxsize) : 用 maxsize 初始化对象，maxsize 是栈中最多能容纳的元素数量。栈在增长到 maxsize 之后就不支持 push 操作。
- void push(int x) : 如果栈未满长到 maxsize , 就将 x 添加到栈顶。
- int pop() : 弹出栈顶元素，并返回栈顶的值，或栈为空时返回 -1 。
- void inc(int k, int val) : 栈底的 k 个元素的值都增加 val 。如果栈中元素总数小于 k , 则栈中的所有元素都增加 val 。

示例：

输入：
["CustomStack","push","push","pop","push","push","increment","increment"]
[[3],[1],[2],[1],[2],[3],[4],[5,100],[2,100],[1],[1],[1]]
输出：
[null,null,null,2,null,null,null,10,202,201,-1]
解释：
CustomStack customStack = new CustomStack(3); // 栈是空的 []
customStack.push(1); // 栈变为 [1]
customStack.push(2); // 栈变为 [1, 2]
customStack.pop(); // 返回 2 -> 返回栈顶值 2。
栈变为 [1]

1031. 两个非重叠子数组的最大和

难度 中等 100 收藏 分享 切换为英文 接收动态 反馈

给出非负整数数组 A , 返回两个非重叠（连续）子数组中元素的最大和，子数组的长度分别为 L 和 M。（这里需要强调的是，长为 L 的子数组可以出现在长为 M 的子数组之前或之后。）

从形式上看，返回最大的 V , 而 V = (A[i] + A[i+1] + ... + A[i+L-1]) + (A[j] + A[j+1] + ... + A[j+M-1]) 并满足下列条件之一：

- 0 <= i < i + L - 1 < j < j + M - 1 < A.length 或
- 0 <= j < j + M - 1 < i < i + L - 1 < A.length 。

示例 1：

输入：A = [0,6,5,2,2,5,1,9,4], L = 1, M = 2
输出：20
解释：子数组的一种选择中，[9] 长度为 1，[6,5] 长度为 2。

还是一样的，之前那个题是限定了区间长度就是2这个题是限定区间长度有L M两种取值

从L+M开始 向后遍历 每次前面的区间有两种分法 L+M or M+L , 这样依次从前往后遍历所有L与M区间

sum 1 2 3 4 5 6 7 8 9 10 L = 1 M = 2

下标：0 1 2 3 4 5 6 7 8 9

元素：0 6 5 2 2 5 1 9 4 2

前缀：0 6 11 13 15 20

131. 分割回文串

难度 中等 807 收藏 分享 切换为英文 接收动态 反馈

给你一个字符串 s , 请将 s 分割成一些子串，使每个子串都是 回文串 。返回 s 所有可能的分割方案。

回文串 是正着读和反着读都一样的字符串。

示例 1：

输入：s = "aab"
输出：[["a", "a", "b"], ["aa", "b"]]
示例 2：

输入：s = "a"
输出：[["a"]]

提示：

- 1 <= s.length <= 16
- s 仅由小写英文字母组成

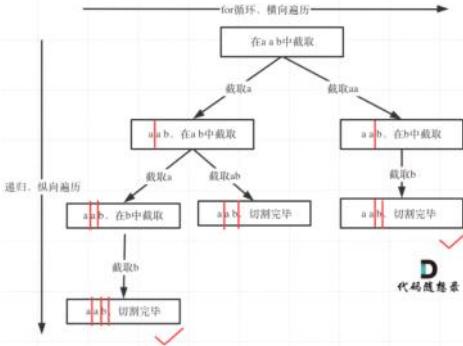
通过次数 125,687 | 提交次数 173,683

a a b
a ---> a ---> b

```
1 class CustomStack {
2 public:
3     vector<int> stk; //用vector实现栈是最容易的
4     int maxsize = 0; //定义全局变量maxsize
5     CustomStack(int maxsize) { //将构造函数传进来的参数 赋给 maxsize
6         maxsize = maxsize;
7     }
8
9     void push(int x) {
10        if(stk.size() == maxsize)
11            return;
12        else
13            stk.push_back(x);
14    }
15
16    int pop() {
17        if(stk.empty())
18            return -1;
19        else
20            int top = stk.back();
21            stk.pop_back();
22            return top;
23    }
24
25    void increment(int k, int val) {
26        int i = 0;
27        while(i < k && i < stk.size()){
28            stk[i] += val;
29            i++;
30        }
31    }
32}
```

```
1 class Solution {
2 public:
3     int maxSumTwoNoOverlap(vector<int>& nums, int L, int M) {
4         int n = nums.size();
5         vector<int> sum(n + 1, 0);
6         for(int i = 0; i < n; i++)
7             sum[i + 1] = sum[i] + nums[i]; //sum 1 = sum 0 + nums 0    sum 2 = sum 1 + nums 1
8
9         int lmax = 0, rmax = 0, res = 0;
10        for(int i = L + M; i < n; i++){
11            lmax = max(lmax, sum[i - M] - sum[i - L]); //L在M前 i是H的最后一个下标
12            res = max(res, lmax + sum[i] - sum[i - M]);
13            rmax = max(rmax, sum[i - L] - sum[i - M]); //M在L之前 i是L的最后一个下标
14            res = max(res, rmax + sum[i] - sum[i - L]);
15        }
16        return res;
17    }
18}
```

```
1 class Solution {
2 public:
3     vector<vector<string>> res;
4     vector<string> path;
5     void backtracking(const string& s, int startIndex){
6         if(startIndex >= s.size()){
7             res.push_back(path);
8             return;
9         }
10        for(int i = startIndex; i < s.size(); i++){
11            if(isPal(s, startIndex, i)){
12                string str = s.substr(startIndex, i - startIndex + 1);
13                path.push_back(str);
14            }
15            else continue;
16            backtracking(s, i + 1);
17            path.pop_back();
18        }
19    }
20    bool isPal(string s, int start, int end){
21        for(int i = start, j = end; i <= j; i++, j--)
22            if(s[i] != s[j])
23                return false;
24        return true;
25    }
26    vector<vector<string>> partition(string s) {
27        backtracking(s, 0);
28        return res;
29    }
30}
```



当切割线也就是 start 到达了字符串结尾，也就是这个 start 注意不是 i，是 start 到达了结尾，说明找到了一种是回文串的切割方式

93. 复原 IP 地址
难度 中等
贡献 653 收藏 分享 为切换为英文 接收动态 反馈

给定一个只包含数字的字符串，用以表示一个 IP 地址，返回所有可能从 s 获得的有效 IP 地址。你可以按任何顺序返回答案。

有效 IP 地址 正好由四个整数（每个整数位 0 到 255 之间组成，且不能含有前导 0），整数之间用“.”分隔。

例如：“0.0.1.201”和“192.168.1.1”是有效 IP 地址，但是“0.0.1.255.245”、“192.168.1.312”和“192.168.0.1”是无效 IP 地址。

示例 1：

输入：s = “2552551135”
输出：[“255.255.11.35”, “255.255.113.5”]

示例 2：

输入：s = “0000”
输出：[“0.0.0.0”]

```
1 class Solution {
2     public:
3         vector<string> res;
4         vector<string> restoreIpAddresses(string s) {
5             if(s.size() > 12)
6                 return res;
7             backtracking(s, 0, 0); //参数是startIndex 与 pointNum
8             return res;
9         }
10     void backtracking(string& s, int startIndex, int pointNum){
11         if(pointNum == 3){ //已经打了三个点 判断第四段是否满足条件 如果是将该打点字符串放入结果
12             if(isValid(s, startIndex, s.size() - 1)) //此时还可以判断越界
13                 res.push_back(s);
14             return;//3个点是递归终点条件
15         }
16         for(int i = startIndex; i < s.size(); i++){
17             if(isValid(s, startIndex, i)){
18                 s.insert(s.begin() + i + 1, '.');
19                 pointNum++;
20                 backtracking(s, i + 2, pointNum);
21                 pointNum--;
22                 s.erase(s.begin() + i + 1);
23             }
24             else
25                 break;
26         }
27     }
28     bool isValid(string s, int start, int end){
29         if(start > end) //只有两个点，再打一个点就会出现startIndex越界
30             return false;
31         if(s[start] == '0' && start != end) // 0开头的数字不合法
32             return false;
33         int num = 0;
34         for (int i = start; i <= end; i++) {
35             if (s[i] > '9' || s[i] < '0') // 遇到非数字字符不合法
36                 return false;
37             num = num * 10 + (s[i] - '0');
38             if (num > 255) { // 如大于255了不合法
39                 return false;
40             }
41         }
42         return true;
43     }
}
```

若不删除，求最大 连续子数组和的状态方程： $dp[i] = arr[i] + \max(dp[i-1], 0)$

66. 二叉树中所有距离为 K 的结点
难度 中等
贡献 412 收藏 分享 为切换为英文 接收动态 反馈

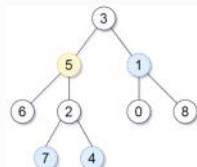
给定一个二叉树（具有根结点 root），一个目标结点 target，和一个整数值 k，
返回到目标结点 target 距离为 k 的所有结点的值的列表。答案可以以任何顺序返回。

示例 1：

输入：root = [5,3,1,6,2,0,8,null,null,7,4], target = 5, k = 2
输出：[7,4,1]
解释：

所求结点为与目标结点（值为 5）距离为 2 的结点。

值分别为 7、4、以及 1。

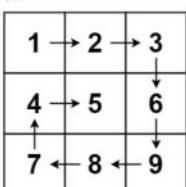


```
11 public:
12     unordered_map<TreeNode*, TreeNode*> parents;
13     vector<int> ans;
14     void findParents(TreeNode* node){
15         if(node->left != nullptr){
16             parents[node->left->val] = node;
17             findParents(node->left);
18         }
19         if(node->right != nullptr){
20             parents[node->right->val] = node;
21             findParents(node->right);
22         }
23     }
24     void findAns(TreeNode* node, TreeNode* from, int depth, int k){
25         if(node == nullptr)
26             return;
27         if(depth == k)
28             ans.push_back(node->val);
29         if(node->left != from)
30             findAns(node->left, node, depth+1, k); //由于每个结点值都是唯一的，哈希表的键可以用结点值代替
31         if(node->right != from)
32             findAns(node->right, node, depth+1, k);
33         if(parents[node->val] != from)
34             findAns(parents[node->val], node, depth+1, k);
35     }
36     //注意在深搜优先遍历时候要先访问 from，在浅搜时比照目标结点是否与来源结点相同
37     //如果相同相当于是上一个结点的父结点，进行剪枝
38     //如果不同相当于是上一个结点的子结点，继续深搜
39     vector<int> distance(TreeNode* root, TreeNode* target, int k) {
40         findParents(root); //从 root 出发 DFS，记录每个结点的父结点
41         // findAns(nullptr, 0, k);
42         findAns(target, nullptr, 0, k); //从 target 出发 DFS，寻找所有深度为 k 的结点
43         return ans;
44     }
45 }
```

54. 螺旋矩阵
难度 中等
贡献 853 收藏 分享 为切换为英文 接收动态 反馈

给你一个 m 行 n 列的矩阵 matrix，请按照螺旋顺序，返回矩阵中的所有元素。

示例 1：



输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[1,2,3,6,9,8,7,4,5]

```
1 class Solution {
2     public:
3         vector<int> spiralOrder(vector<vector<int>> matrix) {
4             vector<int> res;
5             if(matrix.size() == 0) return {};
6             int top = 0, bottom = matrix.size() - 1, left = 0, right = matrix[0].size() - 1;
7             while(true){
8                 for(int i = left; i <= right; i++)
9                     //res.push_back(bottom[i][top]);
10                res.push_back(matrix[top][i]);
11                top++; if(top > bottom) break;
12                for(int i = top; i <= bottom; i++)
13                    res.push_back(matrix[i][right]);
14                    right--; if(right < left) break;
15                for(int i = right; i >= left; i--)
16                    res.push_back(matrix[bottom][i]);
17                bottom--; if(bottom < top) break;
18                for(int i = bottom; i >= top; i--)
19                    res.push_back(matrix[i][left]);
20                    left++; if(left > right) break; //注意打完一圈也要进行判断或回味打印
21            }
22            return res;
23        }
24    };
25 }
```

57. 插入区间

难度 中等 | 468 收藏 分享 切换为英文 接收动态 反馈

给你一个无重叠的，按区间起始端点排序的区间列表。

在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。

示例 1：

```
输入：intervals = [[1,3],[6,9]], newInterval = [2,5]
输出：[[1,5],[6,9]]
```

示例 2：

```
输入：intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
输出：[[1,2],[3,8],[12,16]]
解释：这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。
```

示例 3：

```
输入：intervals = [], newInterval = [5,7]
输出：[[5,7]]
```

$(\text{rand2}() - 1) \times 2 + \text{rand2}() = ? :$

0	+	1	=	1
0	+	2	=	2
2	+	1	=	3
2	+	2	=	4

rand2 与 rand2 可以生成rand4

```
(\text{rand9}() - 1) \times 7 + \text{rand7}() = \text{result}
```

a	b
---	---

可以生成1-63内的随机数

这是通过rand9 与 rand7 生成 rand63

$(\text{RandX}() - 1) \times Y + \text{RandY}() = \text{RandXY}()$ 可以生成 1 到 XY 内的随机数

如何通过rand4 生成rand2 :

```
rand4() \% 2 + 1 = ?
1 \% 2 + 1 = 2
2 \% 2 + 1 = 1
3 \% 2 + 1 = 2
4 \% 2 + 1 = 1
```

事实上，只要rand_N()中N是2的倍数，就都可以用来实现rand2() 如果不是2的倍数，则产生的结果不是等概率的

现在是要实现rand10()，就需要先实现rand_N()，并且保证N大于10且是10的倍数

这样再通过rand_N()%10+1 就可以得到[1,10]范围的随机数了

$(\text{rand7}() - 1) \times 7 + \text{rand7}() ==> \text{rand49}()$

虽然实现的N不是10的倍数但是可以用到“拒绝采样”也就是说，如果某个采样结果不在要求的范围内，则丢弃它

470. 用 Rand7() 实现 Rand10()

难度 中等 | 220 收藏 分享 切换为英文 接收动态 反馈

已有方法 rand7() 可生成 1 到 7 范围内的均匀随机整数，试写一个方法 rand10() 生成 1 到 10 范围内的均匀随机整数。

不要使用系统的 Math.random() 方法。

示例 1：

```
输入：1
输出：[7]
rand4() \% 2 + 1 = rand2()
```

rand40() \% 10 + 1 = rand10()

$(\text{rand7}() - 1) \times 7 + \text{rand7}()$

其实用两个rand7 实现rand49 再拒绝采样就可以了，这里是更进一步优化了效率

```
int rand10() {
    while(true) {
        int num = (rand7() - 1)*7 + rand7(); // 概率生成(1,49)范围的随机数
        if(num <= 40)
            return num \% 10 + 1; // 拒绝采样，并返回[1,10]范围的随机数
        else
            continue;
    }
}
```

// 需要的是10的倍数，因此，不得不会丢弃[41, 49]这个数。优化的点就在于—我们能充分利用这些范围外的数，以减少丢弃的量，提高命中率从而提高随机数生成效率

109. 有序链表转换二叉搜索树

难度 中等 | 570 收藏 分享 切换为英文 接收动态 反馈

给定一个有序链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树指每一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

示例：

给定的有序链表： [-10, -3, 0, 5, 9]，

一个可能的答案是：[0, -3, 9, -10, null, 5]，它可以表示下面这个高度平衡二叉搜索树：

```
0
 / \
-3   9
 /   /
-10  5
```

通过次数 90,118 | 提交次数 118,056

请问您在哪儿招聘中遇到此题？ 杜绝 校招 实习 遭遇到

贡献者

举报

每次都将中间元素构造二叉树 形成一个高度平衡的BST满足要求

```
1 // The rand7() API is already defined for you.
2 // int rand7();
3 // @return a random integer in the range 1 to 7
4
5 class Solution {
6 public:
7     int rand10() {
8         while(true) {
9             int num = (rand7() - 1)*7 + rand7(); // 概率生成[1,49]范围的随机数
10            if(num <= 40)
11                return num \% 10 + 1; // 拒绝采样，并返回[1,10]范围的随机数
12        }
13    }
14 }
15 }
```

```
/*
10 */
11 /**
12 * Definition for a binary tree node.
13 * struct TreeNode {
14 *     int val;
15 *     TreeNode *left;
16 *     TreeNode *right;
17 * };
18 * TreeNode() : val(0), left(nullptr), right(nullptr) {}
19 * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
20 * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
21 */
22 class Solution {
23 public:
24     TreeNode* sortedListToBST(ListNode* head) {
25         vector<int> vec;
26         while(head != nullptr){
27             vec.push_back(head->val);
28             head = head->next;
29         }
30         return build(vec, 0, vec.size() - 1);
31     }
32     TreeNode* build(vector<int> vec, int lo, int hi){
33         if(lo > hi)
34             return nullptr;
35         int mid = lo + (hi - lo) / 2;
36         TreeNode* root = new TreeNode(vec[mid]);
37         root->left = build(vec, lo, mid - 1);
38         root->right = build(vec, mid + 1, hi);
39         return root;
40     }
41 }
```

```

TreeNode* sortedListToBST(ListNode* head) {
    if (head == nullptr) return nullptr; //边界条件的判断
    if (head->next == nullptr) return new TreeNode(head->val);
    //这里通过快慢指针 找到链表的中点结点 slow, pre就是中间结点 也就是slow的前一个结点
    ListNode* slow = head, *fast = head, *pre = nullptr;
    while (fast != nullptr && fast->next != nullptr) {
        pre = slow;
        slow = slow->next;
        fast = fast->next->next;
    }
    pre->next = nullptr; //链表断开为两部分，一部分是node的左子节点，一部分是node的右子节点

    TreeNode* node = new TreeNode(slow->val); //node就是当前slow节点      接着递归每次都找到中点作为根节点
    node->left = sortedListToBST(head); //从head节点到pre节点是node左子树的节点
    node->right = sortedListToBST(slow->next); //从slow.next到链表的末尾是node的右子树的结点
    return node;
}

```

面试题 08.11. 硬币
 难度 中等 218 收藏 分享 切换为英文 接收动态 反馈

硬币，恰好组成n的硬币，面值为25分、10分、5分和1分，编写代码计算n分有几种表示法。
 结果可能很大，你需要将结果模上1000000007。

示例1:

输入: n = 5
 输出: 2
 解释: 有两种方式可以凑成总金额:
 5=5
 5=1+1+1+1+1

示例2:

输入: n = 10
 输出: 4
 解释: 有四种方式可以凑成总金额:
 10=10
 10=5+5
 10=5+1+1+1+1+1
 10=1+1+1+1+1+1+1+1+1+1

2枚骰子 出现的点数为[2,12] 种数为 12 - 2 + 1 = 11 种 取值计算每种取值的个数 除以总的6*6 总个数

动态规划分析问题的状态时，不要分析整体，只分析最后一个阶段即可！
 因为动态规划问题都是划分为多个阶段的，各个阶段的状态表示都是一样，最终答案往往就是在最后一个阶段

对于这道题，最后一个阶段是什么呢？

通过题目我们知道一共投掷 nn 枚骰子，那最后一个阶段很显然就是：当投掷完 n 枚骰子后，各个点数出现的次数

这里的点数指的是前 n 枚骰子的点数和

首先用数组的第一维来表示阶段，也就是投掷完了几枚骰子，第二维来表示投掷完这些骰子后，可出现的点数，数组的值就表示，该阶段各个点数出现的次数。

所以状态表示就是这样的：dp[i][j]，表示投掷完 i 枚骰子后，点数 j 的出现次数

$$dp[n][j] = \sum_{i=1}^6 dp[n-1][j-i]$$

掷完第n枚骰子的点数和可以由当前掷的点数（当前第n枚可以掷点为1 2 3...6）与对应 第n-1枚骰子的点数和决定
 边界处理很简单，只要我们把可以直接知道的状态初始化就好了。

就是第一阶段的状态：投掷完 1 枚骰子后，它的可点数分别为 1, 2, 3, ..., 6 并且每个点数出现的次数都是 1

for (int i = 1; i <= 6; i++) { dp[1][i] = 1; }

剑指 Offer 60. n个骰子的点数
 难度 中等 280 收藏 分享 切换为英文 接收动态 反馈

把n个骰子扔在地上，所有骰子朝上的一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

你需要用一个浮点数数组返回答案，其中第i个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

示例 1:

输入: 1
 输出: [0.16667,0.16667,0.16667,0.16667,0.16667,0.16667]

示例 2:

输入: 2
 输出:
 [0.02778,0.05556,0.08333,0.11111,0.13889,0.16667,0.13889,0.11111,0.08333,0.0,

此外，因为每次计算只会用到 个骰子的方法数，所以第一个维度可以省去。但是注意计算的时候 就得逆序遍历了，这样才不会覆盖掉 个骰子的方案数，造成后面的计算错误
 每个阶段的状态都只和它前一阶段的状态有关，因此我们不需要用额外的一维来保存所有阶段。

用一维数组来保存一个阶段的状态，然后对下一个阶段可出现的点数 j 从大到小遍历，实现一个阶段到下一阶段的转换

```

class Solution {
public:
    vector<double> twoSum(int n) {
        int dp[70];
        memset(dp, 0, sizeof(dp));
        for (int i = 1; i <= 6; i++) {
            dp[i] = 1;
        }
        for (int i = 2; i <= n; i++) {
            for (int j = 6*i; j >= i; j--) {
                dp[j] = 0;
                for (int cur = 1; cur <= 6; cur++) {
                    if (j - cur < i - 1) {
                        break;
                    }
                    dp[j] += dp[j - cur];
                }
            }
            int all = pow(6, n);
            vector<double> ret;
            for (int i = n; i >= 6 * n; i++) {
                ret.push_back(dp[i] * 1.0 / all);
            }
            return ret;
        }
    }
}

```

如果 i=2 枚 j 最小取 2 cur = 1 j-cur = 2 -1 = 1 如果 j-cur < i-1 如 j-cur = 0 那break，没有可取的数

如果 i=3 j 最小取 3 j - cur < 2 j = 6 cur = 1...6 取 j-cur = 1 < i-1 < 2 要满足至少有3个骰子的要求

```

1 class Solution {
2 public:
3     int waysToChange(int n) {
4         vector<int> coins;
5         coins.push_back(1), coins.push_back(5), coins.push_back(10), coins.push_back(25);
6         vector<int> dp(1, 1);
7         dp[0] = 1; //1枚硬币的组合数当然为 1, 没有硬币也是一种情况 dp[0] 为组合数相同的硬币需实现
8         for(int coin : coins){
9             for(int i = coin; i <= n; i++){
10                 dp[i] = (dp[i] + dp[i - coin]) %1000000007;
11             }
12         }
13         return dp[n];
14     }
15     //多硬币求组合问题 必须先遍历硬币/物品再遍历 总数/数和容量
16     //如果先遍历容量会令会出错 6 = 1 * 5 & 5 = 1 * 1 重复计算的捷径 会更改前面计算的结果 !!!
17     //先遍历硬币再硬币
18     i = 1; dp[1] = dp[1] + dp[0] = 1 ... dp[2] = dp[2] + dp[1] = 1 dp[3] = 1 dp[10] = 1
19     i = 5; dp[5] = dp[5] + dp[0] = 2 ... dp[6] = dp[6] + dp[1] = 2+1=3 dp[7] = dp[7] + dp[2] = 3 dp[10] = 3
20     i = 10 dp[10] = dp[10] + dp[0] = 3+1
21     只要1-->可取1 5 10 ...可取 1 5 10 20 更新取得容量为n的方法数
22
23
24
25

```

```

1 class Solution {
2 public:
3     vector<double> diceProbability(int n) {
4         int dp[15][70];
5         memset(dp, 0, sizeof(dp));
6         for(int i = 1; i <= 6; i++)
7             dp[1][i] = 1;
8
9         for(int i = 2; i <= n; i++){
10             for(int j = i; j <= 6*i; j++){ //n枚骰子可掷出的总和
11                 for(int cur = 1; cur <= 6; cur++) //第i枚可掷6种点数
12                     if(j - cur < 0)
13                         break;
14                     dp[i][j] += dp[i-1][j - cur];
15             }
16         }
17         int all = pow(6, n);
18         vector<double> res;
19         for(int i = n; i >= 6*n; i++){
20             res.push_back(dp[n][i] * 1.0 / all);
21         }
22         return res;
23     }
24 }
25

```

剑指 Offer 46. 把数字翻译成字符串

难度 中等 272 收藏 分享 翻译为英文 接收动态 反馈

给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 "a"，1 翻译成 "b"，……，11 翻译成 "t"，……，25 翻译成 "z"。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。

示例 1：

输入：12258

输出：5

解释：12258有5种不同的翻译，分别是"bccfi", "bwfi", "bczi", "mcfi"和"nzi"

91. 解码方法

难度 中等 916 收藏 分享 翻译为英文 接收动态 反馈

一条包含字母 A-Z 的消息通过以下映射进行了编码：

'A' -> 1
'B' -> 2
...
'Z' -> 26

若 码 已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方法）。例如，“11306” 可以映射为：“F”，“EE”，“B”或“K”。

• “AAJF”，将消息分组为 (1 1 10 6)

• “KCF”，将消息分组为 (11 10 6)

注意：消息不能分组为 (1 11 06)，因为“06”不能映射为“F”，这是由于“6”和“06”在映射中不等价。

在映射中不等价的，在映射中不等价的，消息不能分组为 (1 11 06)，因为“06”不能映射为“F”，这是由于“6”和“06”在映射中不等价。

给出一个含数字的 菲空 字符串 s，将计算并返回 码 方法的 检数。

解码时间复杂度近似地是一个 $O(n^2)$ 的时间。

从左往右遍历整个字符串，判断当前字符是否可以单独编码，以及与前一个字符是否可以组成编码。如果可以，则到当前字符的编码种类数应该再加上与前一个字符结合编码一起总的种类数...

题目描述

一个四面体，顶点为 S, A, B, C。从 S 出发，每次任意选一条棱走到另一个顶点，可重复走遍所有顶点和棱。问走 k 次之后，回到 S 的方案数是多少？答案对 $10^9 + 7$ 取模。

题解

明显这是一道动态规划题目，我们令 $dp[i][0]$ 表示走了 i 次之后回到 S 的方案数，令 $dp[i][1]$ 表示走了 i 次之后在 A, B, C 的概率。注意到这里 A, B, C 是对称的，所以方案数应该完全相同，所以我们定义一个就行了。

那么 i 步回到 S 的方案数应该就是 i - 1 步在 A, B, C 的方案数之和：

$$dp[i][0] = dp[i-1][1] * 3$$

i 步在 A 的方案数就是 i - 1 步在 B, C 的方案数加上 i - 1 步在 S 的方案数：

$$dp[i][1] = dp[i-1][1] * 2 + dp[i-1][0]$$

当然空间还可以优化，因为只跟上一步有关，所以保存上一步两个状态值就行了。

```
int main() {
    int k;
    scanf("%d", &k);
    memset(dp, 0, sizeof dp);
    dp[0][0] = 1;
    for (int i = 1; i <= k; ++i) {
        dp[i][0] = (dp[i-1][1] * 3) % mod;
        dp[i][1] = (dp[i-1][1] * 2 + dp[i-1][0]) % mod;
    }
    printf("%lld\n", dp[k][0]);
    return 0;
}
```

按每一个元素/每一列（因为宽度为1）计算雨水

42.接雨水

难度 中等 2602 收藏 分享 翻译为英文 接收动态 反馈

给定 n 个非负整数表示每个高度为 i 的柱子的高，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：



输入：height = [0,1,0,2,1,0,1,3,1,1,2,1]

输出：4

解释：上面是由数组 [0,1,0,2,1,0,1,3,1,1,2,1] 表示的高宽图。在这种情况下，可以接 4 个单位的雨水（蓝色的分区示雨区）。

分析+1数组与开size数组的差异就在于有时如果要用到i-1操作 需先对首元素处理 再将从1开始遍历 如果开size+1的数组 先设定dp[0] 再从0开始遍历就行 dp[i+1] 从0开始遍历[0,n)

size+1也可以dp[i] 从1开始遍历 遍历区间：[1, n]。size一般只能从0开始遍历到[0,n) n-1

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

也可以用单调栈实现（维护单调递减栈，来了一个大于栈顶元素的弹出栈顶元素，计算雨水宽度与高度）



```

int trap(vector<int>& height) {
    int ans = 0, current = 0;
    stack<int> st;
    while (current < height.size()) {
        while (!st.empty() && height[current] > height[st.top()]) {
            int top = st.top();
            st.pop();
            if (!st.empty())
                break;
            int distance = current - st.top() - 1;
            int bounded_height = min(height[current], height[st.top()]) - height[top];
            ans += distance * bounded_height;
        }
        st.push(current++);
    }
    return ans;
}

```

求解超过一半的数字：

最简单用哈希表记录每个数字出现的次数，最后看哪个数字次数超过一半就行了

对数组从小到大进行排序，那么众数一定在 `nums[n/2]` 处

摩尔投票法

169. 多数元素

难度 阅读 1102 收藏 分享 切换为英文 接收动态 反馈

给定一个大小为 n 的数组，找出其中的多数元素。多数元素是指在数组中出现次数大于 $\lfloor n/2 \rfloor$ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1：

输入：[3,2,3]
输出：3

示例 2：

输入：[2,2,1,1,2,2]
输出：2

那如果题目不保证有这样的众数，那就需要对最后的那个 `cand` 判断

重新遍历一次数组 如果数组元素等于这个 `cand`，那么 `cnt` 数加一（遍历前将 `cnt=0`），如果这个 `cnt` 没有超过数组大小的 $1/2$ ，则没有这样的众数

229. 求众数 II

难度 中等 393 收藏 分享 切换为英文 接收动态 反馈

给定一个大小为 n 的整数数组，找出其中所有出现超过 $\lfloor n/3 \rfloor$ 次的元素。

进阶：尝试设计时间复杂度为 $O(n)$ 、空间复杂度为 $O(1)$ 的算法解决此问题。

示例 1：

输入：[3,2,3]
输出：[3]

示例 2：

输入：`nums = [1]`
输出：[1]

示例 3：

输入：[1,1,1,3,3,2,2,2]
输出：[1,2]

提示：

可以看成这样的情形：一个班里要选副班长，至多2位。每一个投一票，成为副班长得票必须超过总票数的三分之一。最多会产生两名

所有出现超过 $\lfloor n/3 \rfloor$ 次的元素 最多只有两个

moore投票法， $O(N)$ 时间， $O(1)$ 空间。本质上是利用两个变量 `cm`, `cn` 记录频率最高的两个元素 `m`, `n` 的频率，遇到 `m`, `n` 自增对应的频率，遇到非 `m`, 非 `n`，自减 `cm`, `cn`。最后再重置 `cm`, `cn` 为 0，再遍历一遍数组查看获取的最高频率的 `m`, `n` 的频率是否大于 $1/3$ 的总元素个数。因为也许最高频率的元素并不大于 $1/3$ 的总元素个数，需要对得到的那两个 `cand` 进行判断（比如 `[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9]`）

如何用最少的数字相加构成 $[1, n]$ 的所有数

1,2 -> [1,4]

1,2,4 -> [1,8]

1,2,4,8 -> [1,16]

1,2,4,8,16 -> [1,32]

.....

330. 按要求补齐数组

难度 困难 126 收藏 分享 切换为英文 接收动态 反馈

给定一个已排序的正整数数组 `nums`，和一个正整数 `n`，从 $[1, n]$ 区间内选取任意多个数字补充到 `nums` 中，使得 $[1, n]$ 区间内的任何数字都可以由 `nums` 中某几个数字的和来表示。请输出满足上述要求的最少需要补充的数字个数。

示例 1：

输入：`nums = [1,3]`, `n = 6`
输出：3
解释：

根据 `nums` 重写的组合有 $[1]$, $[3]$, $[1,3]$ ，可以得出 1 , 3 , 4 ,

现在如果我们把 2 添加到 `nums` 中，组合变为： $[1]$, $[2]$, $[3]$, $[1,3]$, $[2,3]$, $[1,2,3]$;

其和可以表示数字 1 , 2 , 3 , 4 , 5 , 6 ，能将覆盖 $[1, 6]$ 区间内的所有的数。

所以我们最少需要添加一个数字。

示例 2：

输入：`nums = [1,5,10]`, `n = 20`
输出：2
解释：

我们需要添加 $[2, 4]$ 。

1 5 10 $n = 20$

while(`add`<=n)

```
// 1 2 4 --> [1,7] add 1...2 4 5... 10      n=20      n=23
add = 1 add = 2; add = 2 小于5 add += add cnt ++
add = 4 小于5 add += add cnt ++
```

124 这三个数已经可以覆盖到 [1,8] add 大于 5 add + 5

add + 5 = 23 已经可以覆盖到 [1,23] 区间 当 add > n

退出while循环，表示已经可以全部覆盖到

495. 提莫攻击

难度 中等
161 收藏 分享 切换为英文 接收动态 反馈
在《英雄联盟》的世界中，有一个叫“提莫”的英雄，他的攻击可以让敌方英雄艾希（编者注：寒冰射手）进入中毒状态。现在，给出提莫对艾希的攻击时间序列和提莫攻击的中毒持续时间，你需要输出艾希的中毒状态总时长。
你可以认为提莫在给定的时间点进行攻击，并立即使艾希处于中毒状态。

示例 1：

输入：[1,4], 2

输出：4

原因：第 1 秒初，提莫开始对艾希进行攻击并使其立即中毒。中毒状态会维持 2 秒钟，直到第 2 秒末结束。

第 4 秒初，提莫再次攻击艾希，使得艾希获得另外 2 秒中毒时间。

所以最终输出 4 秒。

```
1 class Solution {
2 public:
3     int findPoisonedDuration(vector<int>& timeSeries, int duration) {
4         int sum = 0; int n = timeSeries.size();
5         if(timeSeries.size() == 0) return 0;
6         for(int i = 1; i < n; i++){
7             if(timeSeries[i] - timeSeries[i-1] > duration)
8                 sum += duration; // 两时间段大于duration, sum加duration * /
9             else // 两时间段小于duration, sum加两段间隔即可
10                sum += timeSeries[i] - timeSeries[i-1];
11         }
12         sum += duration; // 加上最后一项的攻击持续时间
13     }
14 }
```

556. 下一个更大元素 III

难度 中等
161 收藏 分享 切换为英文 接收动态 反馈

给你一个正整数 n ，请你找出符合条件的最小整数，其由重新排列 n 中存在的每位数字组成，并且其值大于 n ，如果不存在这样的正整数，则返回 -1。

注意：返回的整数应当是一个 32 位整数，如果存在满足题意的答案，但不是 32 位整数，同样返回 -1。

示例 1：

输入：n = 12

输出：21

示例 2：

输入：n = 21

输出：-1

```
1 class Solution {
2 public:
3     int nextGreaterElement(int n) {
4         string s = to_string(n);
5         int sz = s.size();
6         if(sz <= 1) return -1;
7         int i = sz - 2;
8         for(i = sz-2; i >= 0; i--){
9             if(s[i] < s[i+1])
10                 break;
11         }
12         if(i < 0) return -1; // 全递减序列 已经最大
13         int j = sz - i;
14         for(j = sz - 1; j > i;){
15             if(s[j] > s[i])
16                 break;
17         }
18         swap(s[i], s[j]);
19         reverse(s.begin() + i + 1, s.end()); // 此时i 右边的字符是降序的 剪掉这一部分 可以得到最小的序列
20         long res = stol(s); // 使用stol函数 将string 转成 long型
21         if(res > INT_MAX) // 判断结果 是否溢出 最大也为 INT_MAX
22             return -1;
23         return res;
24     }
25 }
```

如果 n 是保持递减的，比如 321，这种情况是无法找到的，那么就是 -1

921521

从右往左去找是否有更小的数字，找到比右边小的数字 x 就是需要和其他数字交换的

和谁交换呢？**依然是从右往左去找一个正好比 x 大的数字 y** ，那么交换它们

此时数字其实不是最小整数，因为后面是递减的，翻转后改为递增肯定是最小的，所以从 x 右边到结束来做一次翻转

861. 翻转矩阵后的得分

难度 中等
203 收藏 分享 切换为英文 接收动态 反馈

有一个二维矩阵 A 其中每个元素的值为 0 或 1。

移动是选择任一行或列，并转换该行或列中的每一个值：将所有 0 都改为 1，将所有 1 都改为 0。

在做出任意次数的移动后，将该矩阵的每一行都按照二进制数来解释，矩阵的得分就是这些数字的总和。

返回尽可能高的分数。

示例：

输入：[[0,0,1,1],[1,0,1,0],[1,1,0,0]]

输出：39

解释：

转换为 [[1,1,1,1],[1,0,0,1],[1,1,1,1]]

0b1111 + 0b1001 + 0b1111 = 15 + 9 + 15 = 39

也可以使用 $res = \text{pow}(2, m - 1 - j)$ 计算 2 的 n 次方函数

```
for (int i = 0; i < row; ++i) {
    for (int j = col - 1; j >= 0; --j) {
        res += A[i][j] * int(pow(2, col - 1 - j));
    }
}
return res;
```

```
1 class Solution {
2 public:
3     int matrixScore(vector<vector<int>>& grid) {
4         int n = grid.size(), m = grid[0].size();
5         for(int i = 0; i < n;){
6             if(grid[i][0] == 1) // 第一列必须全部为 1 否则要翻转 只有这样 每一行的二进制数才可以最大
7                 continue;
8             for(int j = 0; j < m; j++)
9                 grid[i][j] ^= 1;
10        }
11        int res = (1 << (m - 1)) * n; // 计算第一列的值 1 左移 m-1 位
12        for(int j = 1; j < m; j++){
13            int cnt = 0;
14            for(int i = 0; i < n; i++)
15                cnt += grid[i][j];
16            cnt = max(cnt, n - cnt);
17            res += (1 << (m - 1 - j)) * cnt;
18        }
19        return res;
20    }
21 }
```

926. 将字符串翻转到单调递增

难度 中等
102 收藏 分享 切换为英文 接收动态 反馈

如果一个由 ‘0’ 和 ‘1’ 组成的字符串，是以一些 ‘0’（可能没有 ‘0’）后面跟着一些 ‘1’（也可能没有 ‘1’）的形式组成的，那么该字符串是**单源递增**。

我们给出一个由字符 ‘0’ 和 ‘1’ 组成的字符串 s ，我们可以将任何 ‘0’ 翻转为 ‘1’ 或者将 ‘1’ 翻转为 ‘0’。

返回使 s 单调递增的最小翻转次数。

示例 1：

输入：“00110”

输出：1

解释：我们翻转最后一位得到 00111。

```
1 class Solution {
2 public:
3     int minFlipsMonoIncr(string s) {
4         int n = s.size();
5         int dp[n+1];
6         dp[0] = 0;
7         for (int i = n-1; i >= 0; i--) {
8             dp[i] = dp[i+1] + (s[i] == '1');
9         }
10        int res = dp[0];
11        for (int i = 0; i < n; i++) {
12            res = min(res, dp[0]-dp[i]+n-i-dp[i]);
13        }
14    }
15 }
```

如果我们用数组预处理出来位置 i 开始到最后 1 的数量，记为 $dp[i]$ (从 i 后面 1 的数量)

那么它后面 0 的数量就可以表示为 $n - i - dp[i]$ ，也就是后面的长度减去 1 的数量

而它前面 1 的数量可以表示为 $dp[0] - dp[i]$ ，也就是 1 的总数量减去 i 后面 1 的数量

那么总的修改次数就是 $n - i - dp[i] + dp[0] - dp[i]$ ，我们只需要遍历所有的 i ，找出最小值就行了

另外还需要比较一下 $dp[0]$ 的大小，也就是把所有的 1 都修改为 0

leetcode 1111

示例 1：

输入：seq = "((())*)"
输出：[0,1,1,1,1,0]

示例 2：

输入：seq = "((())*)"
输出：[0,0,0,1,1,0,1]
解释：示例的答案不唯一。
提示输出 A = "((())*", B = "(*))", max(depth(A), depth(B)) = 1，它们的深度是 1。
小。
像 [1,1,1,0,0,1,1,1]，也是正确结果，其中 A = "((())*)", B = "(*))"，
max(depth(A), depth(B)) = 1。

```
1 class Solution {
2 public:
3     vector<int> maxDepthAfterSplit(string seq) {
4         int n = seq.size();
5         vector<int> res;
6         int depth = 0;
7         for(char c : seq){
8             if(c == '('){// 遇到左括号，连续括号个数加 1。
9                 depth++;
10                res.push_back(depth % 2); // % 2 也可以写成 & 1
11            }
12            else{
13                res.push_back(depth % 2); // 遇到右括号，与当前栈顶括号分在一组，因此先取模，再 --
14                depth--;
15            }
16        }
17        return res;
18    }
19 }
20 }
```

提示：

* 1 < seq.size() <= 10000

题目要求我们把输入的整体有效字符串做一个重组，要求是只拆成两个部分 A 和 B，每个字符要么分到 A 要么分到 B，分到 A 标记为 0，分到 B 标记为 1。这个「嵌套深度」就是输入字符串，使用栈完成括号匹配，栈中最多连续出现的左括号 (的个数。

示例 2：

关键：输入需要根据使用「栈」完成括号匹配的过程，如果遍历到的左括号同时出现在栈里，这两个左括号一定不能分到同一组。

不能分在一组，因为它们会同时出现在线上。



注意：重组字符串的顺序保持了在输入字符串中的相对顺序。

说明：结果不唯一，但一定要保证连续的左括号 ((不被分在同一组，具体来说，就是完成「括号匹配」】

面试题 16.16. 部分排序

难度：中等 ⚡ 75 收藏 分享 翻译 切换为英文 接收动态 反馈

给定一个整数数组，编写一个函数，找出索引 n 和 m ，只要将索区间 $[m, n]$ 的元素排好序，整个数组就是有序的。注意： $n - m$ 尽量最小，也就是说，找出符合条件的最短序列。函数返回值为 $[m, n]$ ，若不存在这样的 m 和 n （例如整个数组是有序的），请返回 $[-1, -1]$ 。

示例：

输入：[1,2,4,7,10,11,7,12,6,7,16,18,19]
输出：[3,9]

提示：

* 0 <= len(array) <= 1000000

通过次数 13,996 提交次数 31,055

请问您在哪类招聘中遇到此题？ 杜绝 校招 实习 未遇到

只需要寻找最靠右的那个数（满足左边存在大于它的数），和最靠左的那个数（满足右边存在小于它的数），那么这两个数之间就是要排序的区间

846. 一手顺子

难度：中等 ⚡ 107 收藏 分享 翻译 切换为英文 接收动态 反馈

爱丽丝有一手 (hand) 由整数数组给定的牌。

现在她继续重新排列这些牌，使得每个组的大小都是 w ，且由 w 张连续的牌组成。

如果她可以完成分成组返回 `true`，否则返回 `false`。

注意：此题与 1296 重复；<https://leetcode-cn.com/problems/divide-array-in-sorts-of-k-consecutive-numbers/>

示例 1：

输入：hand = [1,2,3,6,2,3,4,7,8], w = 3
输出：true
解释：要带丝的手牌可以被重新排列为 [1,2,3], [2,3,4], [6,7,8]。

用 `map` 来保存每个数出现的次数。

从最小的数开始，以它作为顺子的开头，然后看顺子里的数是否在 `map` 里，在就次数减一，不在就直接返回 `false`。

接着重复上面步骤，最后直到 `map` 为空，最后返回 `true`

`map` 的特性就是你取它的第一个键值对，它的 `key` 就是最小的，这就很方便了

`count.begin() -> first;` 获取第一个元素

如果一个元素的值为0 调用`count.erase(i)`抹掉这个元素

1248. 统计「优美子数组」

难度：中等 ⚡ 179 收藏 分享 翻译 切换为英文 接收动态 反馈

给你一个整数数组 `nums` 和一个整数 `k`。

如果某个 连续 子数组中恰好有 `k` 个奇数数字，我们就认为这个子数组是「优美子数组」。

请返回这个数组中「优美子数组」的数目。

示例 1：

输入：nums = [1,1,2,1,1], k = 3
输出：2
解释：包含 3 个奇数的子数组是 [1,1,2,1] 和 [1,2,1,1]。

示例 2：

输入：nums = [2,4,6], k = 1
输出：0
解释：数列中不包含任何奇数，所以不存在优美子数组。

```
1 class Solution {
2 public:
3     vector<int> subSort(vector<int>& array) {
4         int n = array.size();
5         int maxn = INT_MIN, minn = INT_MAX;
6         int l = -1, r = -1;
7         for(int i = 0; i < n; i++){
8             if(array[i] < maxn){
9                 r = i;
10            }
11            else
12                maxn = array[i];
13        }
14        for(int i = n-1; i >= 0; i--){
15            if(array[i] > minn){
16                l = i;
17            }
18            else
19                minn = array[i];
20        }
21        return {l, r};
22    }
23 }
```

```
1 class Solution {
2 public:
3     bool isStraightHand(vector<int>& hand, int groupSize) {
4         map<int, int> count;
5         for(auto x : hand)
6             count[x]++;
7         while(count.size()){
8             int start = count.begin()->first; //取第一个元素为顺子开始的那个元素
9             for(int i = start; i < start + groupSize; i++){ // k = 3 0 1 2 ; 1 2 3 ;
10                if(count[start] == count.end()->second) //如果不能构成顺子 直接return false
11                    return false;
12                count[i]--; //使用次数减一
13                if(count[i])
14                    count.erase(i); //如果这个元素次数为0了 必须用count.erase(itr) 移除这个元素
15            }
16        }
17        return true;
18    }
19 }
```

```
1 class Solution {
2 public:
3     int numberOfSubarrays(vector<int>& nums, int k) {
4         vector<int> prefix(nums.size() + 1);
5         prefix[0] = 1;
6         int sum = 0, res = 0; //因为 sum == k 时 prefix[0] = 1 种方法
7         for(auto num : nums) //n个元素 n+1 sum = n ; prefix[n]+ res == prefix[0] = 1 = 0 - n 所以开的n+1大小的数组
8             sum += num;
9             prefix[sum]++; //当前元素不为0 也要加1 如果为0 相当于多了一种取法
10            if(sum == k)
11                res += prefix[sum - k];
12        }
13        return res;
14    }
15 }
```

计算前缀和数组 arr：遍历原数组，每遍历一个元素，计算当前的前缀和（即到当前元素为止，数组中有多少个奇数也就是多少个1）；

对上述前缀和数组，双重循环统计 $arr[j] - arr[i] == k$ 的个数，这样做是 $O(N^2)$

可以使用 Hash优化到 $O(N)$ ，键是「前缀和」，值是「前缀和的个数」

遍历原数组，每遍历到一个元素，计算当前的前缀和 sum，就在 res 中累加上前缀和为 sum - k 的个数

560. 和为K的子数组
难度 中等
1058 收藏 分享 切换为英文 接收动态 反馈

给定一个整数数组和一个整数 k ，你需要找到该数组中和为 k 的连续的子数组的个数。

示例 1：

输入: nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

说明：

1. 数组的长度为 $[1, 20,000]$ 。

2. 数组中元素的范围是 $[100, 1000]$ ，且整数 k 的范围是 $[-1e7, 1e7]$ 。

通过次数 108,580 | 提交次数 310,004

询问您在左侧问题中遇到此题？ 社区 | 检查 | 实习 | 未通过

贡献者

相关企业

992. K 个不同整数的子数组

难度 困难 310 收藏 分享 切换为英文 接收动态 反馈

给定一个正整数数组 A ，如果 A 的某个子数组中不同整数的个数恰好为 K ，则称 A 的这个子数组，不一定是连续的子数组为好子数组。

例如， $[1,2,3,1,2]$ 中有 3 个不同的整数：1，2，以及 3。

返回 A 中好子数组的数量。

示例 1：

输入: A = [1,2,1,2,3], K = 2
输出: 7
解释: 恰好由 2 个不同整数组成的子数组: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]。

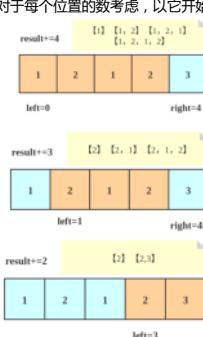
示例 2：

输入: A = [1,2,1,3,4], K = 3
输出: 3
解释: 恰好由 3 个不同整数组成的子数组: [1,2,1,3], [2,1,3], [1,3,4]。

当满足条件的子数组从 $[A, B, C]$ 增加到 $[A, B, C, D]$ 时，新子数组的长度为 4，同时增加的子数组为 $[D]$, $[C, D]$, $[B, C, D]$, $[A, B, C, D]$ 也为 4。

恰含有 k 个：最多含有 k 个 - 最多含有 $k-1$ 个

对于每个位置的数考虑，以它开始的【最多包含 k 个不同整数的子数组】，假设 left 指针指向当前位置，right 指向处为不满足点，则以 $A[left]$ 开始的【最多包含 k 个不同整数的子数组】有 $right - left - 1$ 个，



340. 全部包含 K 个不同字符的最长子串

难度 中等 101 收藏 分享 切换为英文 接收动态 反馈

给定一个字符串 s ，找出至多包含 k 个不同字符的最长子串 T 。

示例 1：

输入: s = "eceba", k = 2
输出: 3
解释: 则 T 为 "ece"，所以长度为 3。

```
/* 340. 至多包含k个不同字符的最长子串 */
int lengthOfLongestSubstringKDistinct(char * s, int k){
    int len = strlen(s);
    int cnt[128];
    int cntDiff = 0;
    int i = 0;
    int j = 0;
    int res = 0;
    memset(cnt, 0, sizeof(cnt));
    while (j < len) {
        cntDiff += (cnt[s[j++]]++ == 0); /* 统计不同字符串的个数 */
        while (cntDiff > k) {
            cntDiff -= (cnt[s[i++]]-- == 1); /* 窗口收缩 */
        }
        res = fmax(res, j - i); /* 求子串的最大长度 */
    }
    return res;
}
```

713. 乘积小于K的子数组

难度 中等 276 收藏 分享 切换为英文 接收动态 反馈

给定一个正整数数组 $nums$ 和整数 k 。

请找出该数组内乘积小于 k 的连续的子数组的个数。

示例 1：

输入: nums = [10,5,2,6], k = 100
输出: 8
解释: 8 个乘积小于 100 的子数组分别为: [10], [5], [2], [6], [10,5], [5,2], [2,6], [5,2,6]。
需要注意的是 [10,5,2] 并不是乘积小于 100 的子数组。

1 class Solution {

```
2 public:
3     int numSubarrayProductLessThanK(vector<int>& nums, int k) {
4         int left = 0, right = 0;
5         int mul = 1, res = 0;
6         int n = nums.size();
7         while(right < n){
8             int c = nums[right];
9             mul *= c;
10            right++;
11            while(mul >= k && left < right){ //这里不能忘记 left < right 这个条件
12                int d = nums[left];
13                mul /= d;
14                left++;
15            }
16            res += right - left; //如果此时left = 0 right = 3 (r指向了大于等于mul的那个元素) 则1与r之间就是 0 1 2
17            //3个元素 这三个元素乘积和小于k 那这三个元素可以组成的子数组就有三个
18        }
19        return res;
20    }
```

迭代器++和--很好理解。迭代器与整数相加减返回的还是一个迭代器，跟指针加上一个整数是类似的，指向的位置发生改变。下面的代码是合法的：

```
1 auto it = v.begin() + 2; // 迭代器it的位置为第三个元素
2 auto it1 = v.end() - 2; // 迭代器it1的位置为倒数第二个元素
cnt += isdigit(array[i][0] == 1) ? 1 : -1
```

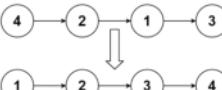
假设子数组 $[l, r]$ 是满足字母个数和数字个数相等的，那么这段子数组总和必然是 $0 \cdot dp[0] = -1$ 前缀和为0出现的下标在-1处 $1 - mp[0] = 0$ 1有两个元素 $1 - -1$ 距离为2 元素个数
如果我们用前缀和来进行优化的话， $\sum[r] - \sum[l-1] = 0$ ，也就是说 $\sum[r]$ 和 $\sum[l-1]$ 的值是相等的
用一个map记录前面前缀和的值和下标 如果出现相等则进行判断 当前的和前面相等的之前距离是否大于已有的 $r-l$ 这么长
如果更长 将 $l=mp[cnt]+1$ 作为左端点 $begin() + l$, $r=i+1$ 作为右端点的下一个end()迭代器 $begin() + r$

剑指 Offer II 077. 链表排序

难度 中等 | 38 | 收藏 | 分享 | 切换为英文 | 接收动态 | 反馈

给定链表的头结点 head，请将其按升序排列并返回排序后的链表。

示例 1：



```
5 *   ListNode* next;
6 *   ListNode() : val(0), next(nullptr) {}
7 *   ListNode(int x) : val(x), next(nullptr) {}
8 *   ListNode(int x, ListNode* next) : val(x), next(next) {}
9 * };
10 */
11
12 class Solution {
13 public:
14     ListNode* sortList(ListNode* head) {
15         if(head == nullptr || head->next == nullptr) return head;
16         ListNode* fast = head, *slow = head, *tail = head;
17         while(fast && fast->next){
18             tail = slow;
19             slow = slow->next;
20             fast = fast->next->next;
21         }
22         tail->next = nullptr; // 切分的链表尾节点应该指向nullptr
23         ListNode* l1 = sortList(head); // 第一步：递归将链表拆成两半
24         ListNode* l2 = sortList(slow);
25         return merge(l1, l2); // 合并排序 合并两个有序链表
26     }
}
```

```
ListNode* merge(ListNode* l1, ListNode* l2){
    ListNode *dummy = new ListNode();
    ListNode* p = dummy;
    while(l1 && l2){
        if(l1->val < l2->val){
            p->next = l1;
            l1 = l1->next;
        } else{
            p->next = l2;
            l2 = l2->next;
        }
        p = p->next;
    }
    if(l1 != nullptr) p->next = l1;
    if(l2 != nullptr) p->next = l2;
    return dummy->next;
}
```

61. 旋转链表

难度 中等 | 615 | 收藏 | 分享 | 切换为英文 | 接收动态 | 反馈

给你一个链表的头节点 head，旋转链表，将链表每个节点向右移动 k 个位置。

示例 1：

```
输入: head = [1,2,3,4,5], k = 2
输出: [4,5,1,2,3]
```

```
5 *   ListNode* next;
6 *   ListNode() : val(0), next(nullptr) {}
7 *   ListNode(int x) : val(x), next(nullptr) {}
8 *   ListNode(int x, ListNode* next) : val(x), next(next) {}
9 * };
10 */
11 //首先计算出链表的长度 n，并找到该链表的末尾节点，将其与头节点相连
12 //将到闭合为环的链表，找到新链表的最后一节节点 n-k%n n从1开始 将当前闭合为环的链表断开，即可得到我们所需要的结果
13 //当链表长度不大于1，或者k/n的商数时，新链表将与原链表相同，无需进行任何处理
14 class Solution {
15 public:
16     ListNode* rotateRight(ListNode* head, int k) {
17         if(k == 0 || head == nullptr || head->next == nullptr) return head;
18         int n = 1;
19         ListNode* cur = head;
20         while(cur->next){
21             cur = cur->next;
22             n++;
23         }
24         int add = n - k % n; //尾节点处
25         if(add == n) return head;
26         cur->next = head; //闭合链表
27         while(add--){ //在原链表内找到最后的一个节点
28             cur = cur->next;
29         }
30         ListNode* ret = cur->next; //带链表的新链表头
31         cur->next = nullptr; //原链表的新链表尾结点，它的next指向 nullptr 断开链表
32         return ret; //头节点
33 }
```

71. 简化路径

难度 中等 310 收藏 分享 切换为英文 接收动态 反馈

给你一个字符串 path，表示指向某一文件或目录的 Unix 风格 绝对路径（以 '/' 开头），请你将其转化为更简单的规范路径。

在 Unix 风格的文件系统中，一个点（'.'）表示当前目录本身；此外，两个点（'..'）表示目录切换到上级（指父目录）。两者都可以是复杂相对路径的组成部分。任意多个连续的斜杠（即，'///'）都被视为单个斜杠 '/'。对于此问题，任何其他格式的点（例如，'....'）均被视为文件/目录名称。

请注意，返回的 规范路径 必须遵循下述规则：

- 始终以斜杠 '/' 开头。
- 两个目录名之间必须只有一个斜杠 '/'。
- 最后一个目录名（如果存在）不能 以 '/' 结尾。
- 此外，路径仅包含从根目录到目标文件或目录的路径上的目录（即，不含 '.' 或 '..'）。

返回简化后得到的 规范路径。

```

1 class Solution {
2 public:
3     string simplifyPath(string path) {
4         stringstream ss(path);
5         string res, tmp;
6         vector<string> ans;
7         while(getline(ss, tmp, '/')){
8             if(tmp == "." || tmp == "") // /home/ 遇到第一个/ 读取到的是空字符串" " 没有读取数据 以/分隔读取
9                 continue;
10            else if(tmp == ".." && !ans.empty())
11                ans.pop_back();
12            else if(tmp != ".")
13                ans.push_back(tmp);
14        }
15        for(auto str: ans){
16            res += "/" + str;
17        }
18        if(res.empty())
19            return "/";
20        return res;
21    }
22 };//stringstream的用法，配合getline使用极佳！分隔符用"/"，然后根据获得的输入来对strs数组进行处理。这里的strs数组相当于一个栈，记录了有意义的目录，可以凭借它来生成一个标准的路径。

```

示例 1：

输入: path = "/home/"
输出: "/home"
解释: 注意，最后一个目录名后面没有斜杠。



525. 连续数组

难度 中等 453 收藏 分享 切换为英文 接收动态 反馈

给定一个二进制数组 nums，找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。

示例 1：

输入: nums = [0,1]
输出: 2
说明: [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。

示例 2：

输入: nums = [0,1,0]
输出: 2
说明: [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。

```

1 class Solution {
2 public:
3     int findMaxLength(vector<int>& nums) {
4         int maxn = 0;
5         unordered_map<int, int> mp;
6         int counter = 0; mp[counter] = -1;
7         int n = nums.size();
8         for (int i = 0; i < n; i++) {
9             int num = nums[i];
10            if (num == 1)
11                counter++;
12            else
13                counter--;
14            if (mp.count(counter)) {
15                int prev = mp[counter];
16                maxn = max(maxn, i - prev);
17            } else {
18                mp[counter] = i;
19            }
20        }
21        return maxn;
22    }

```



为什么要在哈希表中插入{0, -1}？

这是为了辅助讨论该连续数组的起始点在 index == 0 的位置的情况，如果最长连续数组在数组的最前方，考虑 10 or 01

由于以上碰1加一，碰0减一的操作，当0与1数量一致时（连续数组），其连续数组的和为零

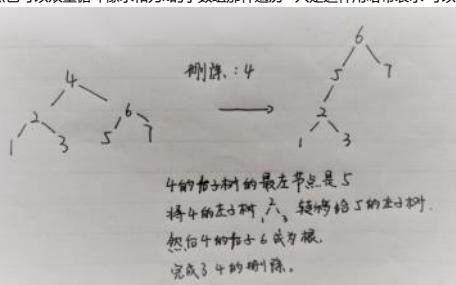
由于「0 和 1 的数量相同」等价于「1 的数量减去 0 的数量等于 0」，我们可以将数组中的 0 视作 -1，则原问题转换成「求最长的连续子数组，其元素和为 0」

维护一个变量 counter 存储 前缀和即可。具体做法是，遍历数组，当遇到元素 1 时将 counter 的值加 1

当遇到元素 0 时将 counter 的值减 1，遍历过程中使用哈希表存储每个前缀和第一次出现的下标

遍历结束时，即可得到 nums 中的有相同数量的 0 和 1 的最长子数组的长度

当然也可以双重循环像求和为 k 的子数组那样遍历，只是这样用哈希表求可以得到和为 0 的最大子数组长度。哈希表记录每个前缀和第一次出现的位置，当再出现时中间那一段和就为 0，那一段长度更新



450. 删除二叉搜索树中的节点

难度 中等 511 收藏 分享 切换为英文 接收动态 反馈

给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

1. 首先找到需要删除的节点；

2. 如果找到了，删除它。

说明：要求算法的平均复杂度为 O(h)，h 为树的高度。

示例：

root = [5,3,6,2,4,null,7]
key = 3

```

5
 \
 3
 / \
 6   2
    / \
   4   7

```

给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 [5,4,6,2,null,null,7]，如下图所示。

递归拼接

```

1 /**
2  * Definition for a binary tree node.
3  */
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11
12    class Solution {
13    public:
14        TreeNode* deleteNode(TreeNode* root, int key) {
15            if(root == nullptr) return nullptr;
16            if(key > root->val)
17                root->right = deleteNode(root->right, key); //如果目标节点大于当前节点值，则去右子树中删除
18            else if(key < root->val) //如果目标节点小于当前节点值，则去左子树中删除
19                root->left = deleteNode(root->left, key);
20            else //如果目标节点等于当前节点值，则分为三种情况
21                if(!root->left) return root->right; //如果左子树为空：则右子树接替其位置，删除了该节点
22                if(!root->right) return root->left; //如果右子树为空：其左子树接替其位置，删除了该节点
23                //否则：将其左子树接替到其右子树的最左节点的左子树上，右子树接替其位置，由上述删除了该节点
24                TreeNode* node = root->right;
25                while(node->left)
26                    node = node->left;
27                node->left = root->left;
28                root = root->right;
29            }
30        return root;
31    }

```

22. 括号生成

难度 中等 | 2020 | 收藏 | 分享 | 切换为英文 | 接收动态 | 反馈

数字 n 表示生成括号的对数, 请你设计一个函数, 用于能够生成所有可能的并且 **有效的** 括号组合。

有效括号组合需满足: 左括号必须以正确的顺序闭合。

示例 1:

输入: $n = 3$
输出: ["((()))","(()())","(())()","(()())","(())()"]

示例 2:

输入: $n = 1$
输出: ["()"]

721. 账户合并

难度 中等 | 298 | 收藏 | 分享 | 切换为英文 | 接收动态 | 反馈

给定一个列表 accounts, 每个元素 accounts[i] 是一个字符串列表, 其中第一个元素 accounts[i][0] 是 **名称(name)**, 其余元素是 emails 表示账户的邮箱地址。

现在, 我们会合并相同账户, 如果两个账户都有一个共同的邮箱地址, 则两个账户必定属于同一个人。请注意, 即使两个账户具有相同的名称, 它们也可能属于不同的人, 因为人们可能具有相同的名称。一个人最初可以拥有任意数量的账户, 但所有账户都具有相同的名称。

合并账户后, 按以下格式返回结果: 每个账户的第一个元素是名称, 其余元素是按 **ASCII 碎序排列** 的邮箱地址。账户本身可以 **任意顺序** 返回。

示例 1:

输入: accounts = [["John", "johnsmith@mail.com", "john00@mail.com"], ["John", "johnnybravo@mail.com"], ["John", "johnsmith@mail.com", "john_newyork@mail.com"], ["Mary", "mary@mail.com"], ["John", "john00@mail.com", "john_newyork@mail.com", "johnsmith@mail.com"], ["Doe", "johnnybravo@mail.com"], ["Mary", "mary@mail.com"]]
输出: [["John", "john00@mail.com", "john_newyork@mail.com", "johnsmith@mail.com"]]
解释:
第一个和第三个 John 是同一个人, 因为他们有共同的邮箱地址
"johnsmith@mail.com"。
第二个 John 和 Mary 是不同的人, 因为他们的邮箱地址没有被其他账户使用。
可以以任意顺序返回这些列表, 例如答案 [["Mary", "mary@mail.com"], ["John", "johnnybravo@mail.com"], ["John", "john00@mail.com", "john_newyork@mail.com", "johnsmith@mail.com"]]
也是正确的。

示例 2:

输入: accounts = [["Gabe", "Gabe@m.co", "Gabe3@m.co", "Gabe1@m.co"], ["Kevin", "Kevin3@m.co", "Kevin@m.co", "Kevin1@m.co"], ["Ethan", "Ethan@m.co", "Ethan0@m.co", "Ethan1@m.co"], ["Hanzo", "Hanzo3@m.co", "Hanzo1@m.co", "Hanzo0@m.co"], ["Fern", "Fern1@m.co", "Fern0@m.co", "Fern2@m.co"]]
输出: [[["Ethan", "Ethan0@m.co", "Ethan1@m.co", "Ethan@m.co"], ["Gabe", "Gabe1@m.co", "Gabe3@m.co"], ["Hanzo", "Hanzo1@m.co", "Hanzo3@m.co"]]]

一个包含 $1-n$ 的序列, 交换任意元素使得序列有序

如 3 5 4 2 1

用一个Map记录每个元素的下标, 再遍历每个元素

如果当前元素不是下标 i 位置, 将这个元素交换到它应该在的位置上面

限定了 $1-n$ 范围, 可以直接根据下标交换元素到对应位置; 如果没有限制元素范围, 那必须要这个map记录排序完, 元素应该处在的下标

再遍历每一个元素, 每次将当前这个元素交换到它应该在的下标处

0 1 2 3 4
3 5 4 2 1

4 5 3 2 1
2 5 3 4 1
5 2 3 4 1
1 2 3 4 5

这样每次交换至少让其中一个元素放到对应的正确位置上面

统计交换次数

如果是相邻的两个元素交换, 需要求逆序对

LC-二叉树和为某一值的路径

112. 路径总和

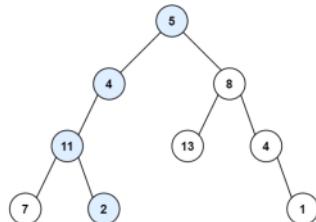
难度 中等 | 669 | 收藏 | 分享 | 切换为英文 | 接收动态 | 反馈

给定二叉树的根节点 root 和一个表示目标和的整数 targetSum , 判断该树中是否存在根节点到叶子节点的路径, 这条路径上所有节点值之和等于目标和 targetSum 。

叶子节点 是指没有子节点的节点。

示例 1:

```
1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     * TreeNode *left;
6  *     * TreeNode *right;
7  *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
8  * };
9 */
10 class Solution {
11 public:
12     bool hasPathSum(TreeNode* root, int targetSum) {
13         if (root == nullptr)//结束条件1: 找完一条路径但是没有找到符合条件的, 返回false
14             return false;
15         if (targetSum == root->val && root->left == nullptr && root->right == nullptr)//结束条件2: 找到一个根节点满足条件, 返回true
16             return true;
17         else
18         {
19             if (hasPathSum(root->left, targetSum - root->val))//向左搜索, 只要找到一个满足条件的根节点就能返回true
20                 return true;
21             if (hasPathSum(root->right, targetSum - root->val))//向右搜索, 只要找到一个满足条件的根节点就能返回true
22                 return true;
23         }
24     }
25 }
```



剑指 Offer 54. 二叉树中的第一棵树
 难度 中等 233 分享 为矩阵类的成员 可以修改为类的成员

输入一棵二叉树和一个整数，打印出二叉树中节点值之和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。

示例
 例如如下二叉树，以及目标和 target = 22。

```

      5
     / \
    4   8
   /   / \
  11  13  4
 / \   / \
 7  2  5  1
    
```

返回
`[{5,4,11,2}, {5,8,4,5}]`

LC-二叉树的所有路径

257. 二叉树的所有路径
 难度 简单 406 分享 为矩阵类的成员

给定一个二叉树，返回所有从根节点到叶节点的路径。

说明：叶子节点是深度没有子节点的节点。

示例：

输入：

```

 1
 / \
 2   3
 \   \
 5
    
```

输出：["1->2->5", "1->3"]

解题：所有根节点到叶节点的路径为：1->2->5, 1->3

通过次数 10,721 提交次数 164,907

询问您在做面试时遇到此题？

杜撰 权限：实习 | 题解

题目描述

评论 (333)

解答 (665)

提交记录

437. 路径总和 III

难度 中等 848 分享 为矩阵类的成员

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000, 1000000] 的整数。

示例：

root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

```

 10
 / \
 5   -3
 / \   \
 3   2  11
 / \   \
 -3  -2  1
    
```

返回 3，等于 8 的路径有：

1. 5 → 3
 2. 5 → 2 → 1
 3. -3 → 11

```

3 * struct TreeNode {
4 *     int val;
5 *     TreeNode *left;
6 *     TreeNode *right;
7 * };
8 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9 * };
10 class Solution {
11 public:
12     int sum = 0;
13     vector<vector<int>> res;
14     vector<vector<int>> pathSum(TreeNode* root, int target) {
15         vector<int> ans;
16         ans.push_back(0);
17         dfs(root, ans, sum, target);
18         return res;
19     }
20     vector<int> tmp;
21     void dfs(TreeNode* root, vector<int>&ans, int sum, int target){
22         if(root == nullptr)
23             return;
24         sum += root->val;
25         ans.push_back(root->val);
26         if(sum == target && root->left == nullptr && root->right == nullptr)
27             res.push_back(ans);
28         else{
29             if(root->left != nullptr)
30                 dfs(root->left, ans, sum, target);
31             if(root->right != nullptr)
32                 dfs(root->right, ans, sum, target);
33             sum -= root->val;
34             ans.pop_back();
35         }
36     }
37 //注意叶子节点，把原本的sum==target..修改为sum==target&&root->left==nullptr&&root->right==nullptr就可以了
38 //叶子节点这个很重要，不然会报错
39 }
    
```

```

1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  * };
8 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 * };
12 class Solution {
13     void traversal(TreeNode* root, string path, vector<string>& paths){//path是堆传递，paths是在主函数中的定义，是要返回的，传的是引用
14         if(root != nullptr){
15             path += to_string(root->val); //先将路径 先记录再加一个> 如果到了叶子结点直接将当前path存入vector<string> paths
16             if(path == target && root->left == nullptr && root->right == nullptr)
17                 paths.push_back(path);
18             traversal(root->left, path, paths); //遍历左子树
19             traversal(root->right, path, paths); //遍历右子树
20             path += ">"; //进入递归的 会记录栈，再进行剥削 遍历到叶子结点时就会存入path操作。整个操作完成后，所有路径都已经放入。
21         }
22     }
23 }
24 public:
25     vector<string> binaryTreePaths(TreeNode* root) {
26         vector<string> paths;
27         traversal(root, "", paths); //初始化path路径为空
28         return paths;
29     }
30 };
    
```

```

1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     TreeNode *left;
6  *     TreeNode *right;
7  * };
8 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
9 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
11 * };
12 class Solution { //双重递归的操作，这种题目需要从每个节点开始进行类似的计算，所以第一个递归用来遍历这些节点，第二个递归用来处理这些节点，进行深度优先搜索
13 public:
14     int count = 0;
15     int pathSum(TreeNode* root, int sum) { //首先先序遍历遍历每个节点，再以每个节点作为起始点遍历寻找满足条件的路径
16         if (!root) return 0;
17         dfs(root, sum); //每个节点都可以是开始点，搜索所有路径
18         pathSum(root->left, sum);
19         pathSum(root->right, sum);
20         return count; //返回总的路径
21     }
22     void dfs(TreeNode* root, int sum) {
23         if (!root) return;
24         if (sum - root->val == 0) count++; //到当前节点的一条路径满足条件 sum == root.val
25         dfs(root->left, sum - root->val); //递归左子树和右子树 直到搜索到一条路径
26         dfs(root->right, sum - root->val);
27     }
28 };
    
```

手写线程池

2021年2月26日 8:23

需要线程池的原因

我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一个问题：**如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间**

那就需要线程可以复用，就是执行完一个任务，并不被销毁，可以继续执行其他的任务

线程池的实现 主要分为 3 个部分，这三部分配合工作就可以得到一个完整的线程池：

1.任务队列，存储需要处理的任务，由工作的线程来处理这些任务

通过线程池提供的 API 函数，将一个待处理的任务添加到任务队列，或者从任务队列中删除已处理的任务会被从任务队列中删除。生产者线程，就是调用线程池函数往任务队列中添加任务的线程

2.工作线程，N个

线程池中维护了一定数量的工作线程，他们的作用是不停的读任务队列，从里边取出任务并处理，相当于是任务队列的消费者角色，如果任务队列为空，工作的线程将会被阻塞（使用条件变量 / 信号量阻塞），如果阻塞中有了新的任务，由生产者将阻塞解除，工作线程开始工作

3.管理者线程（不处理任务队列中的任务），1个

它的任务是周期性的对任务队列中的任务数量以及处于忙状态的工作线程个数进行检测，当任务过多的时候，可以适当的创建一些新的工作线程，当任务过少的时候，可以适当的销毁一些工作的线程

1.任务队列

1.1 类声明 TaskQueue.h

```
// 定义任务结构体 将这个Task修改为类模板 只要修改一个其他涉及的都要修改
using callback = void(*)(void*);
template<typename T>
struct Task
{
    Task()
    {
        function = nullptr;
        arg = nullptr;
    }
    Task(callback f, void* arg)
    {
        function = f;
        this->arg = (T*)arg;
    }
    callback function;
    void* arg;
    T* arg;
};

// 任务队列
template<typename T>
class TaskQueue
{
public:
    TaskQueue();
    ~TaskQueue();

    // 添加任务
    void addTask(Task<T> task); // Task 是一个模板类 ctrl+h 将所有Task替换为Task<T>； TaskQueue里面用到了模板类，那么TaskQueue也应该定义为模板类
    void addTask(callback func, void* arg);

    // 取出一个任务
    Task takeTask();

    // 获取当前队列中任务个数
    inline int taskNumber()
    {
        return m_queue.size();
    }

private:
    pthread_mutex_t m_mutex; // 互斥锁
    std::queue<Task> m_queue; // 任务队列
```

```

};

2.类定义 TaskQueue.cpp
-----
template<typename T> //""源文件的每个函数都需要加template<typename T>
TaskQueue<T>::TaskQueue() //TaskQueue是模板类 需要添加模板参数 模板函数
{
    pthread_mutex_init(&m_mutex, NULL);
}

TaskQueue::~TaskQueue()
{
    pthread_mutex_destroy(&m_mutex);
}

void TaskQueue::addTask(Task& task)
{
    pthread_mutex_lock(&m_mutex);
    m_queue.push(task);
    pthread_mutex_unlock(&m_mutex);
}

void TaskQueue::addTask(callback func, void* arg)
{
    pthread_mutex_lock(&m_mutex);
    Task task;
    task.function = func;
    task.arg = arg;
    m_queue.push(task);
    pthread_mutex_unlock(&m_mutex);
}

Task TaskQueue::takeTask()
{
    Task t;
    pthread_mutex_lock(&m_mutex);
    if (m_queue.size() > 0)
    {
        t = m_queue.front();
        m_queue.pop();
    }
    pthread_mutex_unlock(&m_mutex);
    return t;
}

```

2.线程池

2.1类声明 ThreadPool.h

```

Template <typename T>
class ThreadPool
{
public:
    ThreadPool(int min, int max);
    ~ThreadPool();

    // 添加任务
    void addTask(Task<T> task);
    // 获取忙线程的个数
    int getBusyNumber();
    // 获取活着的线程个数
    int getAliveNumber();

private:
    // 工作的线程的任务函数
    static void* worker(void* arg);
    // 管理者线程的任务函数
    static void* manager(void* arg);
    void threadExit();

private:
    pthread_mutex_t m_lock;
    pthread_cond_t m_notEmpty;
    pthread_t* m_threadIDs;
    pthread_t m_managerID;
    TaskQueue<T*>* m_taskQ; //这样修改当前的线程池类也是一个类模板
    int m_minNum;
    int m_maxNum;
    int m_busyNum;
    int m_aliveNum;
    int m_exitNum;
    bool m_shutdown = false;
};

```

2.2类定义 ThreadPool.cpp

```

Template <typename T>
ThreadPool<T>::ThreadPool(int minNum, int maxNum)
{
    // 实例化任务队列
    m_taskQ = new TaskQueue;
    do {
        // 初始化线程池

```

```

    m_minNum = minNum;
    m_maxNum = maxNum;
    m_busyNum = 0;
    m_aliveNum = minNum;

    // 根据线程的最大上限给线程数组分配内存
    m_threadIDs = new pthread_t[maxNum];
    if (m_threadIDs == nullptr)
    {
        cout << "malloc pthread_t[] 失败...." << endl;
        break;
    }
    // 初始化
    memset(m_threadIDs, 0, sizeof(pthread_t) * maxNum);
    // 初始化互斥锁,条件变量
    if (pthread_mutex_init(&m_lock, NULL) != 0 ||
        pthread_cond_init(&m_notEmpty, NULL) != 0)
    {
        cout << "init mutex or condition fail..." << endl;
        break;
    }

    //////////////////// 创建线程 ///////////////////
    // 根据最小线程个数, 创建线程
    for (int i = 0; i < minNum; ++i)
    {
        pthread_create(&m_threadIDs[i], NULL, worker, this);
        cout << "创建子线程, ID: " << to_string(m_threadIDs[i]) << endl;
    }
    // 创建管理者线程, 1个
    pthread_create(&m_managerID, NULL, manager, this);
    } while (0);
}

ThreadPool::~ThreadPool()
{
    m_shutdown = 1;
    // 销毁管理者线程
    pthread_join(m_managerID, NULL);
    // 唤醒所有消费者线程
    for (int i = 0; i < m_aliveNum; ++i)
    {
        pthread_cond_signal(&m_notEmpty);
    }

    if (m_taskQ) delete m_taskQ;
    if (m_threadIDs) delete[] m_threadIDs;
    pthread_mutex_destroy(&m_lock);
    pthread_cond_destroy(&m_notEmpty);
}

void ThreadPool::addTask(Task task)
{
    if (m_shutdown)
    {
        return;
    }
    // 添加任务, 不需要加锁, 任务队列中有锁
    m_taskQ->addTask(task);
    // 唤醒工作的线程
    pthread_cond_signal(&m_notEmpty);
}

int ThreadPool::getAliveNumber()
{
    int threadNum = 0;
    pthread_mutex_lock(&m_lock);
    threadNum = m_aliveNum;
    pthread_mutex_unlock(&m_lock);
    return threadNum;
}

int ThreadPool::getBusyNumber()
{
    int busyNum = 0;
    pthread_mutex_lock(&m_lock);
    busyNum = m_busyNum;
    pthread_mutex_unlock(&m_lock);
    return busyNum;
}

// 工作线程任务函数
void* ThreadPool::worker(void* arg)
{
    ThreadPool* pool = static_cast<ThreadPool*>(arg);
    // 一直不停的工作
    while (true)
    {
        // 访问任务队列(共享资源)加锁
        pthread_mutex_lock(&pool->m_lock);

```

```

// 判断任务队列是否为空, 如果为空工作线程阻塞
while (pool->m_taskQ->taskNumber() == 0 && !pool->m_shutdown)
{
    cout << "thread " << to_string(pthread_self()) << " waiting..." << endl;
    // 阻塞线程
    pthread_cond_wait(&pool->m_notEmpty, &pool->m_lock);

    // 解除阻塞之后, 判断是否要销毁线程
    if (pool->m_exitNum > 0)
    {
        pool->m_exitNum--;
        if (pool->m_aliveNum > pool->m_minNum)
        {
            pool->m_aliveNum--;
            pthread_mutex_unlock(&pool->m_lock);
            pool->threadExit();
        }
    }
}

// 判断线程池是否被关闭了
if (pool->m_shutdown)
{
    pthread_mutex_unlock(&pool->m_lock);
    pool->threadExit();
}

// 从任务队列中取出一个任务
Task task = pool->m_taskQ->takeTask();
// 工作的线程+1
pool->m_busyNum++;
// 线程池解锁
pthread_mutex_unlock(&pool->m_lock);
// 执行任务
cout << "thread " << to_string(pthread_self()) << " start working..." << endl;
task.function(task.arg);
delete task.arg;
task.arg = nullptr;

// 任务处理结束
cout << "thread " << to_string(pthread_self()) << " end working...";
pthread_mutex_lock(&pool->m_lock);
pool->m_busyNum--;
pthread_mutex_unlock(&pool->m_lock);
}

return nullptr;
}

// 管理者线程任务函数
void* ThreadPool::manager(void* arg)
{
    ThreadPool* pool = static_cast<ThreadPool*>(arg);
    // 如果线程池没有关闭, 就一直检测
    while (!pool->m_shutdown)
    {
        // 每隔5s检测一次
        sleep(5);
        // 取出线程池中的任务数和线程数量
        // 取出工作的线程池数量
        pthread_mutex_lock(&pool->m_lock);
        int queueSize = pool->m_taskQ->taskNumber();
        int liveNum = pool->m_aliveNum;
        int busyNum = pool->m_busyNum;
        pthread_mutex_unlock(&pool->m_lock);

        // 创建线程
        const int NUMBER = 2;
        // 当前任务个数>存活的线程数 && 存活的线程数<最大线程个数
        if (queueSize > liveNum && liveNum < pool->m_maxNum)
        {
            // 线程池加锁
            pthread_mutex_lock(&pool->m_lock);
            int num = 0;
            for (int i = 0; i < pool->m_maxNum && num < NUMBER
                && pool->m_aliveNum < pool->m_maxNum; ++i)
            {
                if (pool->m_threadIDs[i] == 0)
                {
                    pthread_create(&pool->m_threadIDs[i], NULL, worker, pool);
                    num++;
                    pool->m_aliveNum++;
                }
            }
            pthread_mutex_unlock(&pool->m_lock);
        }

        // 销毁多余的线程
        // 忙线程*2 < 存活的线程数目 && 存活的线程数 > 最小线程数量
        if (busyNum * 2 < liveNum && liveNum > pool->m_minNum)
        {

```

```

pthread_mutex_lock(&pool->m_lock);
pool->m_exitNum = NUMBER;
pthread_mutex_unlock(&pool->m_lock);
for (int i = 0; i < NUMBER; ++i)
{
    pthread_cond_signal(&pool->m_notEmpty);
}
}
return nullptr;
}

// 线程退出
template<typename T> // 模板函数 源文件中所有函数也要替换为模板函数
void ThreadPool<T>::threadExit()
{
    pthread_t tid = pthread_self();
    for (int i = 0; i < m_maxNum; ++i)
    {
        if (m_threadIDs[i] == tid)
        {
            cout << "threadExit() function: thread "
            << to_string(pthread_self()) << " exiting..." << endl;
            m_threadIDs[i] = 0;
            break;
        }
    }
    pthread_exit(NULL);
}

test.cpp
-----
#include "ThreadPool.h"
#include <sqifuninstd.h>
#include <stdio.h>
void taskFunc(void* arg) {
    int num = *(int *)arg;
    printf("pthread %ld is working, number= %d\n", pthread_self(), num);
    sleep(1);
}
int main() {
    ThreadPool pool(3, 10); // ThreadPool<int>
    for (int i = 0; i < 100; ++i) {
        int* num = new int(i + 100);
        pool.addTask(Task(taskFunc, num)); // Task<int>
    }
    sleep(20);
    return 0;
}

// 进一步还可以将线程池相关的类更改为模板类 使用Template<typename T>
// 使用模板类的不同：如果是声明和定义放在一个头文件则只需要包含一个头文件
// 如果是声明和定义分别放在头文件和源文件，则都需要包含这两个文件

```

```

thread 1403600147633776 end working...
thread 1403600147633776 start working...
thread 1403600147633776 is working, number = 195
thread 140360056727296 end working...
thread 140360056727296 start working...
thread 140360056727296 is working, number = 196
thread 140360065120000 end working...
thread 140360065120000 start working...
thread 140360065120000 is working, number = 197
thread 140360023156480 end working...
thread 140360023156480 start working...
thread 140360023156480 is working, number = 198
thread 140360031549184 end working...
thread 140360031549184 start working...
thread 140360031549184 is working, number = 199
thread 140360073512704 end working...
thread 140359933884160 end working...
thread 140359925491456 end working...
thread 140360048334592 end working...
thread 1403600147633776 end working...
thread 140360056727296 end working...
thread 140360023156480 end working...
thread 140360031549184 end working...
thread 140360065120000 end working...
thread 140360039941888 end working...*
threadExit() called, 140360073512704 exiting...
threadExit() called, 140359933884160 exiting...
threadExit() called, 140360048334592 exiting...
threadExit() called, 1403600147633776 exiting...
threadExit() called, 140360056727296 exiting...
threadExit() called, 140360023156480 exiting...
threadExit() called, 140360031549184 exiting...
threadExit() called, 140360065120000 exiting...
threadExit() called, 140360039941888 exiting...

```

100个任务一开始由3个线程执行

当工作的线程个数远远低于任务的个数，管理者线程会创建一些额外的线程帮助工作线程处理任务队列任务运行。