Solution Report of Data Structure

Problem1 Maximum Subsequence Sum

问题来源 PTA2023春数据结构题目集

- 账号：幻光
- 网站：PTA
- 题号：01-2（2004年浙江大学计算机专业考研复试真题）

题目

题面

Given a sequence of K integers $N_1, N_2, \ldots, N_k$. A continuous subsequence is defined to be { Ni, Ni+1, ..., Nj } where 1≤i≤j≤K. The Maximum Subsequence is the continuous subsequence which has the largest sum of its elements. For example, given sequence { -2, 11, -4, 13, -5, -2 }, its maximum subsequence is { 11, -4, 13 } with the largest sum being 20.

Now you are supposed to find the largest sum, together with the first and the last numbers of the maximum subsequence.
Input Specification:

Each input file contains one test case. Each case occupies two lines. The first line contains a positive integer K (≤10000). The second line contains K numbers, separated by a space.
Output Specification:

For each test case, output in one line the largest sum, together with the first and the last numbers of the maximum subsequence. The numbers must be separated by one space, but there must be no extra space at the end of a line. In case that the maximum subsequence is not unique, output the one with the smallest indices i and j (as shown by the sample case). If all the K numbers are negative, then its maximum sum is defined to be 0, and you are supposed to output the first and the last numbers of the whole sequence.

Sample

- input:
  10
  10 1 2 3 4 -5 -23 3 7 -21
- output
  10 1 4

问题分析与算法思路

- 和求解最大子列问题类似，只是进行了扩充，在得到最大子列的同时还得把最大子列的前后索引保存下来
- 从左往右遍历，找到的是最大子列的右边索引，因而我们从右往左遍历，就能找到最大子列的左索引
- 考虑序列全都是负数的情况，即最后的和会等于0，因为没有任何值加进来。然后有负数和0的情况，也就是说得把0给辨别出来，因而设置一个更小的初始MaxSum=-1，-1是最小的负数，这样，只要检测到0，ThisSum就比-1大

实现代码

```
#include<iostream>
using namespace std;
#define maxsize 10001

int a[maxsize];
int dp[maxsize];
int start[maxsize];

int main(){
    int K,i,l,r,sum,b;
    cin>>K;
    for(i=0;i<K;i++){
        cin>>a[i];
    }
    dp[0] = a[0];
    sum = a[0];
    start[0] = 0;
    l = 0;
    r = 0;
    for(i=1;i<K;i++){
        if(dp[i-1]+a[i]>a[i]){
            dp[i] = dp[i-1]+a[i];
            start[i] = start[i-1];
            if(dp[i]>sum){
                sum = dp[i];
                l = start[i];
                r = i;
            }
        }else if(dp[i-1]+a[i]==a[i]){
            start[i] = start[i-1];
            dp[i] = a[i];
            if(dp[i]>sum){
                sum = dp[i];
                l = start[i];
                r = i;
            }
        }else{
            dp[i] = a[i];
            start[i] = i;
            if(dp[i]>sum){
                sum = dp[i];
                l = start[i];
                r = i;
            }
        }
    }
    if(sum >= 0){
        cout<<sum<<" "<<a[l]<<" "<<a[r];
    }else{
        cout<<0<<" "<<a[0]<<" "<<a[K-1];
    }

    return 0;
}
```

结果分析

由OJ机制知，正确

- 注意，对于 3 -2 4 而言，前面有一个3和-3，加起来为0，所以也放到了最大子列里面，同时前面的0也都放到了最大子列里面。因此需要将向左遍历里面判断改为大于等于

0	sample换1个数字。有正负，负数开头结尾，有并列最大和	答案正确	13	5 ms	476 KB
1	最大和序列中有负数	答案正确	2	5 ms	480 KB
2	并列和对应相同i但是不同j，即尾是0	答案正确	2	4 ms	328 KB
3	1个正数	答案正确	2	5 ms	460 KB
4	全是负数	答案正确	2	5 ms	384 KB
5	负数和0	答案正确	2	4 ms	476 KB
6	最大和前面有一段是0	答案正确	1	5 ms	324 KB
7	最大N	答案正确	1	27 ms	660 KB

Problem2 Pop Sequence

问题来源 PTA2023春数据结构题目集

- 账号：幻光
- 网站：PTA
- 题号：02-4（2013年PAT春季考试真题）

题目

题面

Given a stack which can keep M numbers at most. Push N numbers in the order of 1, 2, 3, ..., N and pop randomly. You are supposed to tell if a given sequence of numbers is a possible pop sequence of the stack. For example, if M is 5 and N is 7, we can obtain 1, 2, 3, 4, 5, 6, 7 from the stack, but not 3, 2, 1, 7, 5, 6, 4

Each input file contains one test case. For each case, the first line contains 3 numbers (all no more than 1000): M (the maximum capacity of the stack), N (the length of push sequence), and K (the number of pop sequences to be checked). Then K lines follow, each contains a pop sequence of N numbers. All the numbers in a line are separated by a space.

For each pop sequence, print in one line "YES" if it is indeed a possible pop sequence of the stack, or "NO" if not.

Sample

- input:
  - 5 7 5
    1 2 3 4 5 6 7
    3 2 1 7 5 6 4
    7 6 5 4 3 2 1
    5 6 4 3 7 2 1
    1 7 6 5 4 3 2
- output
  - YES
  - NO
  - NO
  - YES
  - NO

问题分析与算法思路

- 以输入数据的第二行为例，注意7是可能的，因为这个时候栈是空的，所以可以把4 5 6 7 依次push进去，并输出7。但是想输出5，必须先输出6，所以错误。
- 如果栈是空的，push进去一个当前应该push的数据进去
- 如果读入的数据值大于mystack.top()，说明还得继续push，push到等于current才能pop出current同值数据。
- 如果读入数据小于mystack.top()，那这种情况一定是错误的
- 如果当最后我们已经push了N个数，并且栈内也是空的，证明该顺序是可行的

代码实现

```cpp
#include<iostream>
#include<stack>
using namespace std;
int main(void){
    int K,N,M;
    cin>>M>>N>>K;
    stack<int> myStack;
    for(int i=0;i<K;i++){
        int current;
        int sum=0;
        int pushRe=1;
        int inputRe=0;

        for(int j =0;j<N;j++){
            cin>>current;
            inputRe++;
            if(myStack.empty()){
                myStack.push(pushRe);
                sum++;
                pushRe++;
                }
            if(current>myStack.top()){
                for(int k=pushRe;k<=current;k++){
                    myStack.push(k);
                    sum++;
                    pushRe++;
                }
            }
            if(sum>M){
                cout<<"NO"<<endl;
                break;
            }
            if(myStack.top()!=current){
                cout<<"NO"<<endl;
                break;
            }else{
                sum--;
                myStack.pop();
            }
            if(pushRe==N+1&&sum==0)
                cout<<"YES"<<endl;
        }
        while(inputRe<N)
        {
            cin>>current;
            inputRe++;
        }
        while(!myStack.empty())
            myStack.pop();
    }

    return 0;
}
```

结果分析

由OJ机制知，正确

- 注意，循环最后还得清空栈，否则会影响下一次循环

测试点	提示	结果	分数	耗时	内存
0	sample乱序，一般的Y&N	答案正确	15	4 ms	472 KB
1	达到最大size后又溢出	答案正确	3	4 ms	472 KB
2	M==N	答案正确	2	4 ms	324 KB
3	最大数	答案正确	2	4 ms	480 KB
4	最小数	答案正确	1	3 ms	464 KB
5	卡特殊错误算法（通过比较大小判断）	答案正确	2	3 ms	324 KB

Problem 3 Tree Traversals Again

问题来源 PTA2023春数据结构题目集

- 账号：幻光
- 网站：PTA
- 题号：03-3（2014年PAT秋季考试甲级真题）

题目

题面

An inorder binary tree traversal can be implemented in a non-recursive way with a stack. For example, suppose that when a 6-node binary tree (with the keys numbered from 1 to 6) is traversed, the stack operations are: push(1); push(2); push(3); pop(); pop(); push(4); pop(); pop(); push(5); push(6); pop(); pop(). Then a unique binary tree (shown in Figure 1) can be generated from this sequence of operations. Your task is to give the postorder traversal sequence of this tree.

Each input file contains one test case. For each case, the first line contains a positive integer N (≤30) which is the total number of nodes in a tree (and hence the nodes are numbered from 1 to N). Then 2N lines follow, each describes a stack operation in the format: "Push X" where X is the index of the node being pushed onto the stack; or "Pop" meaning to pop one node from the stack.

For each test case, print the postorder traversal sequence of the corresponding tree in one line. A solution is guaranteed to exist. All the numbers must be separated by exactly one space, and there must be no extra space at the end of the line.

Sample

```
input
  6
  Push 1
  Push 2
  Push 3
  Pop
  Pop
  Push 4
  Pop
  Pop
  Push 5
  Push 6
  Pop
  Pop

output
   3 4 2 6 5 1
```

问题分析与算法思路

- pop的顺序是中序遍历的顺序，而push的顺序是前序遍历的顺序，而通过前序和中序就可以确定树的结构
- 通过递归算法来循环找到目标值，每次输入的坐标中，前序列的开始与结尾间距与中序的是一样的
- 当开始坐标等于结尾坐标时，说明是个叶子，送进栈中。否则就判断，如果没有左子树，除去根以外剩下部分就是右子树。反之亦然。
- 如果左右子树都有，则分别递归。注意先递归右边的子树，先进行递归的会先进栈后出

代码实现

```
#include<iostream>
#include<stack>
#include<string>
using namespace std;
stack<int> PostBack;
stack<int> MidOpt;
int PreEdc[100];
int PreNum=0;
int MidEdc[100];
int MidNum=0;
int Num;
int Count;
void Read(){
    cin>>Num;
    string opt;
    int data;
    int TmpNum=2*Num;
    for(int i=0;i<TmpNum;i++){
        //因存在push和pop操作，所以总操作数目为2*Num
        cin>>opt;
        if(opt=="Push"){
            cin>>data;
            PreEdc[PreNum++]=data;
            MidOpt.push(data);
        }
        else
        {
            MidEdc[MidNum++]=MidOpt.top();
            MidOpt.pop();
        }

    }
}

int FindPoint(int start,int end,int A[],int a){
    for(int i=start;i<=end;i++){
        if(A[i]==a)
            return i;
    }
    return -1;
}

void FindLeafs(int StartPre,int EndPre,int StartMid,int EndMid){
    PostBack.push(PreEdc[StartPre]);
    if(StartPre==EndPre)//Means it is the leaf
        return;
    else{
        //找到根在中序遍历里的位置，目标元素即先序遍历的第一个元素
        int Root=FindPoint(StartMid,EndMid,MidEdc,PreEdc[StartPre]);
        if(Root==StartMid)//无左子树
            FindLeafs(StartPre+1,EndPre,Root+1,EndMid);
            //进入右子树，先序遍历的第一个元素为右子树的根，所以StartPre+1
            //中序遍历开始为根的后一个结点，因而为Root+1
        else if(Root==EndMid)//无右子树
            FindLeafs(StartPre+1,EndPre,StartMid,Root-1);
            //进入左子树，先序遍历的第一个元素为左子树的根，所以StartPre+1;
            //中序遍历结束为根的前一个结点，因而为Root-1
        else{//递归查找左右子树
        //先递归右侧，因为先递归先进栈后出栈
            FindLeafs(StartPre+1+Root-StartMid,EndPre,Root+1,EndMid);
            //进入右子树，先序遍历的第一个元素为右子树的根，所以StartPre+1+Root-StartMid
            FindLeafs(StartPre+1,StartPre+Root-StartMid,StartMid,Root-1);
            //进入左子树，先序遍历的第一个元素为左子树的根，所以StartPre+1，结尾为左子树的结尾，所以StartPre+Root-StartMid
        }
    }
}

int main(){
    Read();
    FindLeafs(0,Num-1,0,Num-1);
    cout<<PostBack.top();
    PostBack.pop();
    while(!PostBack.empty()){
        cout<<" "<<PostBack.top();
        PostBack.pop();
```

```
    }
    return 0;
}
```

结果分析

- 通过OJ机制知，正确
- 最后打印的时候，末尾一定不要有空格，否则程序也无法AC

测试点	提示	结果	分数	耗时	内存
0	sample 有单边有双边结点	答案正确	12	5 ms	496 KB
1	单边喇叭张开形	答案正确	4	5 ms	592 KB
2	交错	答案正确	4	5 ms	460 KB
3	N=1	答案正确	1	5 ms	328 KB
4	N=30，复杂组合	答案正确	4	5 ms	588 KB

Problem 4 Complete Binary Search Tree

问题来源 PTA2023春数据结构题目集

- 账号：幻光
- 网站：PTA
    - 题号：04-6（2013年PAT秋季考试甲级真题）

题目

Description

A Binary Search Tree (BST) is recursively defined as a binary tree which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than or equal to the node's key.
- Both the left and right subtrees must also be binary search trees.
  A Complete Binary Tree (CBT) is a tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.
  Now given a sequence of distinct non-negative integer keys, a unique BST can be constructed if it is required that the tree must also be a CBT. You are supposed to output the level order traversal sequence of this BST.
  For each test case, print in one line the level order traversal sequence of the corresponding complete binary search tree. All the numbers in a line must be separated by a space, and there must be no extra space at the end of the line.

Sample

```
Sample Input:
10
1 2 3 4 5 6 7 8 9 0

Sample Output:
6 3 8 1 5 7 9 0 2 4
```

问题分析与思路

- 通过前序遍历的顺序，可以确定根节点，然后通过中序遍历的顺序，可以确定左右子树的结构
- 先将数据进行排序，然后通过递归算法，将数据分为左右子树，然后递归，直到叶子节点
- 设置递归函数，输入的参数为左右子树的起始和结束位置，然后递归，直到叶子节点
- 递归的base case是，N=1，即只有一个元素，直接插入树中，N=0，即没有元素，直接返回，其余处理N，判断左子树和右子树的元素个数，然后递归，直到叶子节点，中间的数插入到树中

代码实现

```
#include<iostream>
#include<queue>
#include<string>
using namespace std;
int Data[1000];

int Outcome[1000];
int Count=0;

typedef int ElementType;
typedef struct Node *Position;
typedef Position Tree;
struct Node
{
    ElementType Data;
    Tree Left;
    Tree Right;
};

Tree Root=NULL;
queue<Tree> Q;

void ReadData(int N){
    for(int i=0;i<N;i++){
        cin>>Data[i];
    }
    int temp;
    for (int i = N-1;i > 0;i--) {
        for (int j = 0;j<i;j++) {
            if (Data[j] > Data[j+1]) {
                temp = Data[j+1];
                Data[j+1] = Data[j];
                Data[j] = temp;
            }
        }
    }
}

Tree Insert(Tree Root,ElementType X){
    if(Root==NULL){
        Root=new Node;
        Root->Data=X;
        Root->Left=Root->Right=NULL;
    }
    else if(X<Root->Data){
        Root->Left=Insert(Root->Left,X);
    }
    else{
        Root->Right=Insert(Root->Right,X);
    }
    return Root;
}

void PreHandleData(int N,int startL){
    if(N==0)
        return;
    else if(N==1){
        Root=Insert(Root,Data[startL]);
        Count++;
        return;
    }

    int NumR,NumL;
    int pow=1;
    int gen=1;
    int Num2;
    for(int i=0;i<20;i++){
        pow*=2;
        gen+=pow;
        if(gen>=N){
            Num2=i;
            gen-=pow;
            pow/=2;
            break;
        }
```

```
    }
    if(gen+pow>=N){
        NumL=(gen-1)/2+N-gen;
        NumR=N-NumL-1;//右边元素个数
    }else{
        NumL=(gen-1)/2+pow;//左边元素个数
        NumR=N-NumL-1;//右边元素个数
    }
    Root=Insert(Root,Data[startL+NumL]);
    PreHandleData(NumL,startL);
    PreHandleData(NumR,startL+NumL+1);
}

void LevelorderTraversal(Tree BST){
    Tree T;
    if(!BST)
        return;
    Count=0;
    Q.push(BST);
    while(!Q.empty()){
        T=Q.front();
        Q.pop();
        Outcome[Count++]=T->Data;
        if(T->Left)
            Q.push(T->Left);
        if(T->Right)
            Q.push(T->Right);
    }
}

void CompleteTree(int N){
    PreHandleData(N,0);
    LevelorderTraversal(Root);
    cout << Outcome[0];
    for (int i = 1;i<N;i++) {
        cout << " "<<Outcome[i];
    }
}

int main(){
    int N;
    cin>>N;
    ReadData(N);
    CompleteTree(N);
    return 0;
}
```

结果分析

- 通过OJ机制知，正确

测试点	提示	结果	分数	耗时	内存
0	sample 换数字， 但大小顺序不变	答案正确	18	4 ms	320 KB
1	完全平衡	答案正确	3	4 ms	488 KB
2	右边有余	答案正确	2	4 ms	484 KB
3	底层只多1个	答案正确	2	4 ms	324 KB
4	只有1个	答案正确	3	4 ms	324 KB
5	最大N随机	答案正确	2	5 ms	324 KB

- 可采用更简单的方式，不用建树。
    - 在每次递归的时候，在递归的程序里输入一个值代表位置，然后在调用的递归程序里把根按位置保存进去。
    - 比如计算的第一个根保存在下标为0的位置，然后在调用的递归程序中，输入分别为*20+1和*20+2，然后分别存入*21+1和*21+2和*22+1和*22+2 以此类推

Problem 5

问题来源 PTA2023春数据结构题目集

- 账号：幻光
- 网站：PTA
- 题号：04-5（2013年浙江大学计算机学院免试研究生上机考试真题）

题目

题面

An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.
Now given a sequence of insertions, you are supposed to tell the root of the resulting AVL tree.
For each test case, print the root of the resulting AVL tree in one line.

Sample

- input
  7
  88 70 61 96 120 90 65
- output
  88

问题分析与思路

- 当插入结点时，需要考虑树高的变化，因而需要考虑是否要左单旋，右单旋，左右双旋，右左双旋
- 在每个结点都更新树高，避免求树高的时候递归到叶结点

代码实现

```
#include<iostream>
using namespace std;
typedef int ElementType;
typedef struct AVLNode *Position;
typedef Position AVLTree;
#define Max(a,b) ((a)>(b)?(a):(b))
struct AVLNode
{
    ElementType Data;
    AVLTree Left;
    AVLTree Right;
    int Height;
};

int GetHeight(AVLTree Root){
    if(!Root)
        return 0;
    else
        return Root->Height;
}

int GetHeight2(AVLTree Root){
    if(!Root)
        return 0;
    if(Root->Left==NULL&&Root->Right==NULL)
        return 1;
    if(Root->Left&&!Root->Right)
        return 1+GetHeight(Root->Left);
    else if(Root->Right&&!Root->Left)
        return 1+GetHeight(Root->Right);
    else
        return 1+Max(GetHeight(Root->Left),GetHeight(Root->Right));
}

AVLTree SingleLeftRotation(AVLTree A){
    AVLTree B=A->Left;
    A->Left=B->Right;
    B->Right=A;
    A->Height=Max(GetHeight(A->Left),GetHeight(A->Right))+1;
    B->Height=Max(GetHeight(B->Left),A->Height)+1;
    return B;
}

AVLTree SingleRightRotation(AVLTree A){
```

```
    AVLTree B=A->Right;
    A->Right=B->Left;
    B->Left=A;
    A->Height=Max(GetHeight(A->Left),GetHeight(A->Right))+1;
    B->Height=Max(GetHeight(B->Right),A->Height)+1;
    return B;
}

AVLTree DoubleLeftRightRotation(AVLTree A){
    A->Left=SingleRightRotation(A->Left);
    return SingleLeftRotation(A);
}

AVLTree DoubleRightLeftRotation(AVLTree A){
    A->Right=SingleLeftRotation(A->Right);
    return SingleRightRotation(A);
}

AVLTree Insert(AVLTree Root,ElementType X){
    if(!Root){
        Root=(AVLTree)malloc(sizeof(struct AVLNode));
        Root->Data=X;
        Root->Height=0;
        Root->Left=Root->Right=NULL;
    }
    else if(X<Root->Data){
        Root->Left=Insert(Root->Left,X);
        if(GetHeight(Root->Left)-GetHeight(Root->Right)==2){
            if(X<Root->Left->Data)
                Root=SingleLeftRotation(Root);
            else
                Root=DoubleLeftRightRotation(Root);
        }
    }
    else if(X>Root->Data){
        Root->Right=Insert(Root->Right,X);
        if(GetHeight(Root->Right)-GetHeight(Root->Left)==2){
            if(X>Root->Right->Data)
                Root=SingleRightRotation(Root);
            else
                Root=DoubleRightLeftRotation(Root);
        }
    }
    Root->Height=Max(GetHeight(Root->Left),GetHeight(Root->Right))+1;
    return Root;
}

int main(){
    int N;
    cin>>N;
    AVLTree Root=NULL;
    for(int i=0;i<N;i++){
        int data;
        cin>>data;
        Root=Insert(Root,data);
    }
    cout<<Root->Data<<endl;
    return 0;
}
```

结果分析

- 通过OJ机制知，正确

测试点	提示	结果	分数	耗时	内存
0	fig 1 - LL	答案正确	4	6 ms	460 KB
1	fig 2 - RR	答案正确	4	9 ms	384 KB
2	fig 3 - RL	答案正确	4	9 ms	476 KB
3	fig 4 - LR	答案正确	4	7 ms	584 KB
4	深度LL旋转	答案正确	4	8 ms	540 KB
5	最大N，深度RL旋转	答案正确	4	6 ms	320 KB
6	最小N	答案正确	1	6 ms	456 KB

- 避免最终递归求树高，可减少时间开销

Problem 6 前n个数字二进制中1的个数

问题来源 剑指offerII 03

- 账号 zwr_magiclight
- 网站 https://leetcode-cn.com/problems/w3tCBm/
- 题号 剑指offerII 03

题目

题面

给定一个非负整数 n ，请计算 0 到 n 之间的每个数字的二进制表示中 1 的个数，并输出一个数组，其中$0 <= n <= 10^5$

示例

```
输入: n = 2
输出: [0,1,1]
解释:
0 --> 0
1 --> 1
2 --> 10
```

问题分析与算法思路

- 本题为计算二进制中1的个数，因而应当采用位运算的方法
- 最直观的为直接遍历，对于每个数字，计算其二进制中1的个数，时间复杂度为$O(nlogn)$
- 若要降低算法的时间和空间复杂度，可以采用动态规划的思路
    - 因对于正整数x，右移一位后，其二进制中1的个数为x/2的二进制中1的个数
    - 当x为奇数时，x的二进制中1的个数为x/2的二进制中1的个数+1
    - 当x为偶数时，x的二进制中1的个数为x/2的二进制中1的个数
    - 综合考虑以上两种情况，可得到状态转移方程：
    - $bits[x] = bits[x >> 1] + (x\&1)$
    - 该算法的时间复杂度为$O(n)$，空间复杂度为$O(1)$

代码实现

```java
//直接遍历（Brian Kernighans算法）
class Solution {
    public int[] countBits(int n) {
        int[] bits = new int[n + 1];//初始化数组
        for (int i = 0; i <= n; i++) {
            int count = 0;
            int num = i;
            while (num != 0) {
                if ((num & 1) == 1) {
                    count++;
                }
```

```
                num = num >> 1;
            }
            bits[i] = count;
        }
        return bits;
    }
}
```

```
//动态规划
class Solution {
    public int[] countBits(int n) {
        int[] bits = new int[n + 1];//初始化数组
        for (int i = 1; i <= n; i++) {
            bits[i] = bits[i >> 1] + (i & 1);
        }
        return bits;
    }
}
```

结果分析

- 由OJ结果知答案正确
- 基于考虑右移的基础上，也可以基于对最高位的考虑设计算法，具体可见Leetcode题解

Problem 7 单词长度最大乘积

问题来源 剑指offerII 03

- 账号 zwr_magiclight
- 网站 https://leetcode.cn/problems/aseY1I/
- 题号 剑指offerII 05

题目

题面

给定一个字符串数组 words，请计算当两个字符串 words[i] 和 words[j] 不包含相同字符时，它们长度的乘积的最大值。假设字符串中只包含英语的小写字母。如果没有不包含相同字符的一对字符串，返回 0。

示例

```
输入: words = ["abcw","baz","foo","bar","xtfn","abcdef"]
输出: 16
解释: 这两个单词为 "abcw", "xtfn"。
```

问题分析与算法思路

- 问题关键在于如何判断两个字符串是否包含相同字符，单纯暴力比较时间开销过大
- 采用Hash表的方法，将字符串中的字符映射到Hash表中，再比较两个字符串的Hash表，若有相同字符，则两个字符串不符合要求，否则符合要求
    - 因只有小写字母，故Hash表可用26位的二进制数表示，每一位代表一个字母，若该位为1，则该字符串中含有该字母
    - 该方法初始化Hash的时间开销为$O(nk)$, k代表平均每个字符串的长度；根据Hash判断有无相同字符，花费时间为$O(n^2)$
    - 总时间复杂度为$O(nk+n^2)$,空间复杂度为O(n)
- 采用位掩码的方式，将字符串中的字符映射到位掩码中，再比较两个字符串的位掩码，若有相同字符，则两个字符串不符合要求，否则符合要求。
    - 哈希表记录每个位掩码对应的最大单词长度，然后遍历哈希表中的每一对位掩码，如果这一对位掩码的按位与运算等于 0，则用这一对位掩码对应的长度乘积更新单词长度的最大乘积。
    - 由于每个单词的位掩码都不等于 0，任何一个不等于 0 的数和自身做按位与运算的结果一定不等于 0，因此当一对位掩码的按位与运算等于 0 时，这两个位掩码一定是不同的，对应的单词也一定是不同的
    - 该算法的时间复杂度为$O(nk+n^2)$,空间复杂度为O(n)

代码实现

```
//Hash表
class Solution {
    public int maxProduct(String[] words) {
        int n = words.length;
        int[] masks = new int[n];
        int[] lens = new int[n];
        int bitmask = 0;
        for (int i = 0; i < n; i++) {
            bitmask = 0;
            for (char ch : words[i].toCharArray()) {
                bitmask |= 1 << (ch - 'a');
```

```
            }
            masks[i] = bitmask;
            lens[i] = words[i].length();
        }
        int maxVal = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if ((masks[i] & masks[j]) == 0) {
                    maxVal = Math.max(maxVal, lens[i] * lens[j]);
                }
            }
        }
        return maxVal;
    }
}
```

```
//位掩码
class Solution {
    public int maxProduct(String[] words) {
        int n = words.length;
        int[] masks = new int[n];
        int[] lens = new int[n];
        int bitmask = 0;
        for (int i = 0; i < n; i++) {
            bitmask = 0;
            for (char ch : words[i].toCharArray()) {
                bitmask |= 1 << (ch - 'a');
            }
            masks[i] = bitmask;
            lens[i] = words[i].length();
        }
        int maxVal = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if ((masks[i] & masks[j]) == 0) {
                    maxVal = Math.max(maxVal, lens[i] * lens[j]);
                }
            }
        }
        return maxVal;
    }
}
```

结果分析

- 由OJ结果知答案正确
- 尽管两种算法时间复杂度和空间复杂度一致，但前面的解法在判断两个字符串是否包含相同的字符时，可能需要26次布尔运算，而新的解法只需要1次位运算，因此后面解法的时间效率更高。

Problem 8 和为k的子数组

问题来源 剑指offerII 010

- 账号 zwr_magiclight
- 网站 https://leetcode.cn/problems/QTMn0o/
- 题号 剑指offerII 010

题目

题面

给定一个整数数组和一个整数 k ，请找到该数组中和为 k 的连续子数组的个数。

示例

```
输入:nums = [1,1,1], k = 2
输出: 2
解释: 此题 [1,1] 与 [1,1] 为两种不同的情况
```

思路与算法

- 问题即以 i 结尾的和为 k 的连续子数组个数，我们需要统计符合条件的下标 j 的个数
- 采用枚举的思路，枚举 j 的下标，然后计算以 i 结尾的和为 k 的连续子数组个数
    - 为降低时间复杂度，计算[j,i]的和时，可在O(1)的时间复杂度内通过[j+1,i]的和求出
    - 算法的时间复杂度为$O(n^2)$，空间复杂度为O(1)
- 采用前缀和与哈希表的思路，令pre[i]作为[0,i]中所有数的和，则我们只要统计多少个前缀和为$pre[i]-k$的$pre[j]$即可
    - 建立哈希表mp,以和为key，以出现次数为value，从左往右边更新mp，则以i结尾的答案为mp[pre[i]-k]
    - 因pre[i]的计算仅与前一项有关，因而无需建立pre数组，直接用pre变量代替即可
    - 算法的时间复杂度为$O(n)$，空间复杂度为O(n)

代码实现

```
//枚举
class Solution {
    public int subarraySum(int[] nums, int k) {
        int n = nums.length;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int sum = 0;
            for (int j = i; j >= 0; j--) {
                sum += nums[j];
                if (sum == k) {
                    ans++;
                }
            }
        }
        return ans;
    }
}
```

```
//前缀和与哈希表
class Solution {
    public int subarraySum(int[] nums, int k) {
        int n = nums.length;
        int ans = 0;
        Map<Integer, Integer> mp = new HashMap<>();
        mp.put(0, 1);
        int pre = 0;
        for (int i = 0; i < n; i++) {
            pre += nums[i];
            if (mp.containsKey(pre - k)) {
                ans += mp.get(pre - k);
            }
            mp.put(pre, mp.getOrDefault(pre, 0) + 1);
        }
        return ans;
    }
}
```

结果分析

- 由OJ结果知答案正确
- 鉴于数组元素有正有负，不能保证在子数组中添加新的数字就能得到和更大的子数组。同样，也不能保证删除子数组最左边的数字就能得到和更小的子数组，因此双指针的思路不适用于本题。

Problem 9 二维子矩阵之和

问题来源 剑指offerII 013

- 账号 zwr_magiclight
- 网站 https://leetcode.cn/problems/O4NDxx/
- 题号 剑指offerII 013

题目

题面

给定一个二维矩阵 matrix，以下类型的多个请求：

- 计算其子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) 。

实现 NumMatrix 类：

- NumMatrix(int[][] matrix)给定整数矩阵 matrix 进行初始化
- int sumRegion(int row1, int col1, int row2, int col2)返回左上角 (row1, col1)、右下角 (row2, col2) 的子矩阵的元素总和。

示例

```
输入:
["NumMatrix","sumRegion","sumRegion","sumRegion"]
[[[[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]]],
 [2,1,4,3],
 [1,1,2,2],
 [1,2,2,4]]
输出:
[null, 8, 11, 12]
```

思路与算法

- 若直接暴力求和，则每次求和时间复杂度为$O(mn)$，空间复杂度为O(1)
- 采用一维前缀和的形式，每次求和需遍历m行，因而每次求和花费时间复杂度为$O(m)$，需维护一个前缀和数组，空间复杂度为O(mn)
- 为降低每次求和的时间复杂度，可采用二维前缀和的思路
    - 由矩阵性质分析，可得到矩阵中以(i,j)为右下角的子矩阵的和为
      $sumRegion(r1, col1, r2, col2) = sumRegion(r2, col2) - sumRegion(r1 - 1, col2) - sumRegion(r2, col1 - 1) + sumRegion(r1 - 1, col1 - 1)$
    - 因而初始化的时候，也可采用类似思路，初始化时，$sum[i][j] = sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1] + matrix[i][j]$

代码实现

```java
//一维前缀和
class NumMatrix {
    int[][] sum;
    public NumMatrix(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;
        sum = new int[m][n + 1];
        for (int i = 0; i < m; i++) {
            for (int j = 1; j <= n; j++) {
                sum[i][j] = sum[i][j - 1] + matrix[i][j - 1];
            }
        }
    }

    public int sumRegion(int row1, int col1, int row2, int col2) {
        int ans = 0;
        for (int i = row1; i <= row2; i++) {
            ans += sum[i][col2 + 1] - sum[i][col1];
        }
        return ans;
    }
}
```

```java
//二维前缀和
class NumMatrix {
    int[][] sum;
    public NumMatrix(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;
        sum = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                sum[i][j] = sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1] + matrix[i - 1][j - 1];
            }
        }
    }

    public int sumRegion(int row1, int col1, int row2, int col2) {
        return sum[row2 + 1][col2 + 1] - sum[row1][col2 + 1] - sum[row2 + 1][col1] + sum[row1][col1];
    }
}
```

结果分析

- 由OJ结果知答案正确
- 在设置矩阵时，可设置m+1行n+1列的矩阵，因而无需对边界进行特殊处理，代码更加简洁

Problem 10 含有重复元素集合的组合

问题来源 剑指offerII 082

- 账号 zwr_magiclight
- 网站 https://leetcode.cn/problems/4sjJUc/
- 题号 剑指offerII 082

题目

题面

给定一个可能有重复数字的整数数组 candidates 和一个目标数target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次，解集不能包含重复的组合。

示例

```
candidates = [10,1,2,7,6,1,5], target = 8
输出:
[
[1,1,6],
[1,2,5],
[1,7],
[2,6]
]
```

思路与算法

- 由于我们需要求出所有和为 target的组合，并且每个数只能使用一次，因此我们可以使用递归 + 回溯的方法来解决
- 由于数组中可能存在重复元素，因而在回溯的过程中，可能会出现重复的组合，因而需要对数组进行一定处理，如Hash映射等，以便于在回溯的过程中进行剪枝
    - 使用HashMap来统计candidates中各元素出现次数，将结果放进列表中
    - 递归时，当前第pos个元素，其值为fre[pos][0],次数为fre[pos][1]
    - 递归时，我们调用dfs（pos+1，res-i*fre[pos][0]）,res为当前剩余的值,i为选择次数
    - 空间复杂度为O(n),时间复杂度为$O(n * 2^n)$

代码实现

```java
class Solution {
    List<int[]> freq = new ArrayList<int[]>();
    List<List<Integer>> ans = new ArrayList<List<Integer>>();
    List<Integer> sequence = new ArrayList<Integer>();

    public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        Arrays.sort(candidates);
        for (int num : candidates) {
            int size = freq.size();
            if (freq.isEmpty() || num != freq.get(size - 1)[0]) {
                freq.add(new int[]{num, 1});
            } else {
                ++freq.get(size - 1)[1];
            }
        }
        dfs(0, target);
        return ans;
    }

    public void dfs(int pos, int rest) {
        if (rest == 0) {
            ans.add(new ArrayList<Integer>(sequence));
            return;
        }
        if (pos == freq.size() || rest < freq.get(pos)[0]) {
            return;
        }

        dfs(pos + 1, rest);

```

```
        int most = Math.min(rest / freq.get(pos)[0], freq.get(pos)[1]);
        for (int i = 1; i <= most; ++i) {
            sequence.add(freq.get(pos)[0]);
            dfs(pos + 1, rest - i * freq.get(pos)[0]);
        }
        for (int i = 1; i <= most; ++i) {
            sequence.remove(sequence.size() - 1);
        }
    }
}
```

结果分析

- 由OJ结果知答案正确
- 我们可将fre根据数排序，这样在递归时，可从小到大选择，这样可减少递归次数