

1.0-1 背包

原文来源: <http://www.cnblogs.com/acm-bingzi/p/3149727.html>

2.1 动态规划算法

递推关系: $c[i][m] = \max\{c[i-1][m], c[i-1][m-w[i]]+p[i]\}$

从左往右, 从上往下, $c[i][j]$ 表示剩余 i 件物品、剩余 j 的容量所能达到的最大价值。剩余的物品只能是从小编号开始, 比如, 剩余两件物品, 则表示的是物品大小分别为 3、4 的两件剩余。

最大容量M	物品个数N													
10	3		C	0	1	2	3	4	5	6	7	8	9	10
物品大小w	物品价值p	编号	0	0	0	0	0	0	0	0	0	0	0	0
3	4	i= 1	1	0	0	0	4	4	4	4	4	4	4	4
4	5	i= 2	2	0	0	0	4	5	5	5	9	9	9	9
5	6	i= 3	3	0	0	0	4	5	6	6	9	10	11	11

```
#include<stdio.h>
#define N 10

/*
** 0-1 背包问题的动态规划算法 (Dynamic Programming Algorithm) 实现
*/
int c[N][N];    // c[i][j], 剩余 i 件物品、j 的容量条件下所能获得的最大价值
int w[N];       // 体积 (重量)
int p[N];       // 价值
int x[N];       // 第 i 件物品是否装入背包

/*
** n 件物品, 背包容量 m 的最大价值
*/
int knapsack( int n,int m )
{
    int i, j, k;
    /*
    ** 初始情况下假设所有的值都为 0
    ** 对于 n 件物品, 剩余物品件数的情况可能是[0,n];同理容量的取值范围是
    [0,m]
    */
    for( i = 0; i < n+1; i++ )
        for( j = 0; j < m+1; j++ ) {
            c[i][j] = 0;
```

```

    }

    /*
    ** 在 i 或 j 为 0 的情况下，剩余的最大价值为 0，故此处直接从下标为 1 的
    开始
    */
    for( i = 1; i < n+1; i++ ) {
        for( j = 1; j < m+1; j++ ) {
            if( w[i] <= j ) { // 如果当前物品的重量比剩余容量小，则可以选择放
            入或不放入
                /*
                ** 递推关系：c[i][m]=max{c[i-1][m], c[i-1][m-w[i]]+p[i]}
                ** 放入的情况下价值为：p[i]+c[i-1][j-w[i]]
                ** 不放入的情况下价值为：c[i-1][j]
                */
                if( p[i]+c[i-1][j-w[i]] > c[i-1][j] ) {
                    c[i][j]=p[i]+c[i-1][j-w[i]];
                } else {
                    c[i][j] = c[i-1][j];
                }
            } else { // 如果当前物品重量大于剩余容量，则不能放入
                c[i][j] = c[i-1][j];
            }
        }
    }

    return c[n][m];

}

/*
** 输出物品的放入情况
*/
void traceBack( int n, int m ) {

    int i;
    for ( i = n; i > 0; i-- ) {
        if ( c[i][m] == c[i-1][m] ) { // 说明第 i 件物品没有加入
            x[i] = 0;
        } else {
            x[i] = 1;
            m -= w[i];
        }
    }
}

```

```

        printf("n 个物品装入背包情况是: ");
        for ( i = 1; i < n+1; i++ ) {
            printf(" %d",x[i]);
        }
        puts("");
    }

int main() {

    int n, m;
    while( scanf("%d%d", &n, &m) != EOF ) {
        for( int i = 1; i <= n; i++ ) {
            scanf( "%d%d", &w[i], &p[i] );
        }
        printf( "背包能获得的最大价值 = %d\n", knapsack(n,m) );
        traceBack(n,m);
    }

    return 0;
}

```

2.2 回溯算法

深度优先遍历

假设给定图 G 的初态是所有顶点均未曾访问过。在 G 中任选一顶点 v 为初始出发点(源点)，则深度优先遍历可定义如下：首先访问出发点 v ，并将其标记为已访问过；然后依次从 v 出发搜索 v 的每个邻接点 w 。若 w 未曾访问过，则以 w 为新的出发点继续进行深度优先遍历，直至图中所有和源点 v 有路径相通的顶点(亦称为从源点可达的顶点)均已被访问为止。若此时图中仍有未访问的顶点，则另选一个尚未访问的顶点作为新的源点重复上述过程，直至图中所有顶点均已被访问为止。

剪枝函数

算法搜索至解空间树的任一结点时，总先判断该结点是否肯定不包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的系统搜索，逐层向其祖先结点回溯。

```
#include<stdio.h>
```

```

/*
** 用回溯算法（BackTracking Algorithm）实现 0-1 背包问题
*/

```

```

int c;        //背包能承受的最大重量
int n;        //背包能装的最多的物品数
int cw;       //背包现在的载重
int cp;       //背包现在的价值
int bestp;    //背包的最大价值
int w[50],p[50]; //物品的重量和价值
int x[50],bestx[50]; //最优状态下，如果第 i 个物品装入背包，那么 bestx[i]=1;否则为 0

```

```

int Bound(int i) //计算上界，限界函数
{
    int cleft=c-cw;//剩余容量
    int b=cp;
    //以物品单位重量价值递减序装入物品
    while(i<=n&&w[i]<=cleft)
    {
        cleft-=w[i];
        b+=p[i];
        i++;
    }
    if(i<=n) //装满背包
        b+=p[i]/w[i]*cleft;
    return b;
}

```

```

void backTrack(int i)
{
    if(i>n)
    {
        if(bestp<cp)
        {
            for(int j=1; j<=n; j++)
                bestx[j]=x[j];
            bestp=cp;
        }
        return;
    }
    if(cw+w[i]<=c) //搜索左子树
    {
        x[i]=1;
        cw+=w[i];
        cp+=p[i];
    }
}

```

```

        backTrack (i+1);
        cw-=w[i];
        cp-=p[i];
    }
    if(Bound(i+1)>bestp) //搜索右子树
    {
        x[i]=0;
        backTrack (i+1);
    }
}

int main()
{
    int i;
    while(scanf("%d%d",&n,&c)!=EOF)
    {
        for(i=1; i<=n; i++)
        {
            scanf("%d%d",&w[i],&p[i]);
        }
        cw=0;
        cp=0;
        bestp=0;
        backTrack(1);
        printf("背包能获得的最大价值 = %d\n",bestp);
        printf("n 个物品装入背包情况是: ");
        for(i=1;i<=n;i++)
            printf(" %d",bestx[i]);
        puts("");
    }
    return 0;
}

```

2. 矩阵连乘

问题描述：给定 n 个数字矩阵 A_1, A_2, \dots, A_n ，其中相邻的两个矩阵是可乘的，求矩阵连乘的加括号方法，使得所用的数值乘法运算次数最少。

2.3 动态规划算法

```
/*
** 矩阵连乘的动态规划算法
** p:保存 n 个矩阵的行数和列数,由于相邻两个矩阵可乘,故一共只需要 n+1
个元素
** n: 矩阵的个数
** m:m[i][j]表示从第 i 个矩阵到第 j 个的连乘积 A[i..j]所用的最少数乘次数
** s:s[i][j]表示从第 i 个矩阵到第 j 个应该从哪一个开始划分
*/
void matrixChain ( int *p, int n, int **m, int **s ) {

    int i, r, k;
    int temp;
    /*
    ** 同一个矩阵的连乘运算次数为 0
    */
    for ( i = 0; i < n; i++ ) {
        m[i][i] = 0;
    }

    for ( r = 1; r < n; r++ ) {    // r 是跨度
        for ( i = 0; i < n-r; i++ ) {
            j = i+r;
            // 将除第一个矩阵外的所有其它矩阵放在一个括号内所需要的
            步骤数

            m[i][j] = m[i+1][j] + p[i]*p[i+1]*p[j+1];
            s[i][j] = i;
            /*
            ** 从中间位置加括号所需要的步骤数
            ** 划分成[i,k]和[k+1,j]两个矩阵序列（其中 i<k<j）
            ** 当 j=i+1 时，不会执行，因为没有其它从中间划分的方法
            ** 当 j>i+1 时，比如 j=i+2，则除了将后面两个矩阵先运算外，
            也可以将前面两个矩阵先运算
            ** 其它情况类似
            */
        }
    }
}
```

}

$$A_0:30*35; A_1:35*15; A_2:15*5; A_3:5*10; A_4:10*20; A_5:20*25$$
$$A_0:30*35; A_1:35*15; A_2:15*5; A_3:5*10; A_4:10*20; A_5:20*25$$

s	0	1	2	3	4	5
0		0	0	2	2	2
1			1	2	2	2
2				2	2	2
3					3	4
4						4
5						

表认真分析代码，好好体会上面这句话的道理。

上诉算法只是明确给出了矩阵最优连乘次序所用的数乘次数 $m[0][n-1]$ ，并为明确给出最后连乘次序，即完全加括号方法。但是以 $s[i][j]$ 为元素的 2 维数组却给出了足够信息。事实上， $s[i][j]=k$ 说明，计算连乘积 $A[i..j]$ 的最佳方式应该在矩阵 A_k 和 A_{k+1} 之间断开，即最优加括号方式为 $(A[i..k])(A[k+1..j])$ 。

3. 动态规划算法基础

3.1 最优子结构

在找寻最优子结构时，可以遵循一种共同的模式：

1) 问题的一个解可以是做一个选择。例如，选择一个前一个装配线装配站；或者选择一个下标以在该位置分裂矩阵链。做这种选择会得到一个或多个有待解决的子问题。

2) 假设对一个给定的问题，已知的是一个可以导致最优解的选择。不必关心如何确定这个选择，尽管假定它是已知的。

3) 在已知这个选择后，要确定哪些子问题会随之发生，以及如何最好地描述所得到的子问题空间。

4) 利用一种“剪贴”(cut-and-paste)技术，来证明在问题的一个最优解中，使用的子问题的解本身也必须是最优的。通过假设每一个子问题的解都不是最优解，然后导出矛盾，即可做到这一点。

3.2 动态规划算法与贪心算法的区别

第 16 章中将研究“贪心算法”，它与动态规划有着很多相似之处。特别地，贪心算法适用的问题也具有最优子结构。贪心算法与动态规划有一个显著的区别，就是在贪心算法中，是以自顶向下的方式使用最优子结构的。贪心算法会先做选择，在当时看起来是最优的选择，然后再求解一个结果子问题，而不是先寻找子问题的最优解，然后再做选择。

LeetCode 解题

1. Binary Tree Level Order Traversal

Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).

For example:

Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
 /  \
15   7
```

return its level order traversal as:

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

最初的思路

使用一个队列（不是循环队列，所以浪费的空间可能有点大）来进行宽度优先搜索，每一层用一个 null 分隔符进行标示，即可完成对每层元素的记录。

结果运行提示超时。

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
public class Solution {
```

```
    public List<List<Integer>> levelOrder(TreeNode root) {
```

```
        List<List<Integer>> result = null;
```

```
        List<Integer> list = null;
```

```
        TreeNode temp = null;
```

```
        TQueue queue = null;
```

```
        result = new ArrayList<List<Integer>>();
```

```
        queue = new TQueue();
```

```
        queue.push(root);
```

```

        queue.push(null);
        while ( !queue.isEmpty() ) {
            list = new ArrayList<Integer>();
            temp = queue.pop();
            while (temp != null) {
                list.add(temp.val);
                if ( temp.left != null ) {
                    queue.push(temp.left);
                }
                if ( temp.right != null ) {
                    queue.push(temp.right);
                }
                temp = queue.pop();
            }
            queue.push(null);
            result.add(list);
        }

        return result;
    }
}

```

```

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

```

```

class TQueue {

    private TreeNode[] nodes = null;
    private int length = 0;
    private int front = 0;
    private int tail = 0;

    public TQueue () {
        nodes = new TreeNode[100];
        length = nodes.length;
    }

    public void push ( TreeNode node ) {
        if ( isFull() ) {

```

```

        TreeNode[] temp = new TreeNode[(int)(length*1.3)];
        for ( int i = 0; i < length; i++ ) {
            temp[i] = nodes[i];
        }
        temp[front++] = node;

        nodes = temp;
        length = (int)(length*1.3);
    } else {
        nodes[front++] = node;
    }
}

public TreeNode pop () {
    if ( isEmpty() ) {
        return null;
    } else {
        return nodes[tail++];
    }
}

public boolean isEmpty () {
    return front==tail;
}

public boolean isFull () {
    return front==length;
}
}

```

改进后的方法

直接通过递归函数实现。

```

import java.util.ArrayList;
import java.util.List;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Solution {

```

```

public List<List<Integer>> levelOrder(TreeNode root) {

    List<List<Integer>> result = null;
    result = new ArrayList<List<Integer>>();
    order( result, 1, root );
    return result;

}

private void order ( List<List<Integer>> result, int level,
TreeNode node ) {

    if ( node != null) {
        List<Integer> list = null;
        /*
         * 如果result元素个数小于level, 说明该层的元素一个都还没有
         加进去, 则需要新建一个List用来存放数据
         */
        if ( result.size() < level) {
            list = new ArrayList<Integer>();
            result.add(list);
        }
        list = result.get(level-1);
        list.add(node.val);
        if ( node.left != null ) {
            order(result, level+1, node.left);
        }
        if ( node.right != null ) {
            order(result, level+1, node.right);
        }
    }
}
}

```

2. Count and Say

问题分析

The count-and-say sequence is the sequence of integers beginning as follows:

1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11.

11 is read off as "two 1s" or 21.

21 is read off as "one 2", then "one 1" or 1211.

Given an integer n , generate the n^{th} sequence.

Note: The sequence of integers will be represented as a string.

题目理解：给定一个整数，如果相邻的几个数位数字是相同的，则输出重复出现的次数加上该数字本身；继续向后查找直到所有的数位处理完。

按照上述思路，则：

input	output
1	11
11	21
21	1211

但实际运行时提示出错，在输入为 1 的时候输出结果为 11，而实际的应该是 1。接着发现题目意思理解错了，如下图所示，后面的输出结果是依赖与前面的输出结果的。比如第一个输出为 1；第二个输出是对 1 进行上述运算，即 11；同理，第三个是对 11 进行运算，即 21；接着依次是 1211、111221、312211、13112221、1113213211……

381 views



I think that given a number x , then the code is to change it a way to read the digits of x : input: 1, output: 11 input: 2, output: 21 input: 11332, output: 212312. my code gets a wrong answer: input: 1, my output is 11, and the expected output is 1, why? I cannot understand



Hi lamster, the problem is not about 'reading off the digits of input x ', but rather the input x is an INDEX in a number sequence, where the x th number is the 'read-off' of the $(x-1)$ th number (e.g. if the 102th number is 8553223, then the 103th number would be 1825132213). And you are asked to return the number which x corresponds to (e.g. if $x = 103$, then you are expected to output 1835132213).

If your input is 11332, then you need to find the 11332th number in the following number sequence: 1, 11, 21, 1211, 111221 ... and it is probably an extremely big number.

源代码

```
public class Solution {
    public String countAndSay(int n) {

        if ( n == 1 ) {
            return "1";
        } else {

            char current;    // the current char
            int count;       // the count of the current char
            String result = new String("1"); // the result
            int length = result.length();    // the length of the
result string
            StringBuilder strBuilder = new StringBuilder(); // store
the current result

            int i = 0;
            while ( n > 1 ) {
                for ( i = 0; i < length; i++ ) {
                    current = result.charAt(i);
                    count = 1;
                    // while the next char is the same as the current
char

                    while ( (i+1 < length) &&
                        (result.charAt(i+1) == current) ) {
                        count++;
                        i++;
                    }
                    // add the char and its count to the current result
                    strBuilder.append(count).append(current);
                }
                // update the result and its length, and clear the
content of strBuilder for the next loop
                result = strBuilder.toString();
                length = result.length();
                strBuilder = new StringBuilder();
                n--;
            }

            return result;
        }
    }
}
```

3. ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P A H N
A P L S I I G
Y I R
```

And then read line by line: "PAHNAPLSIIGYIR"


Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string text, int nRows);
```

`convert("PAYPALISHIRING", 3)` should return `"PAHNAPLSIIGYIR"`.

↑ +3


↓ votes



Best answer

Check this:
<https://oj.leetcode.com/discuss/1212/expected-output-of-abcde-4>

```
A      G
|      / |
B    F  H
|    /  |
C  E   i
|  /
D
```



answered Oct 25 by [coolzai](#) (590 points)
selected Oct 26 by [wcyhbm520](#)

[ask related question](#) [comment](#)

单纯只举一个例子，根本很难把问题描述清楚，第二张图是 row 为 4 的情况。
源代码：

```
public class Solution {

    public String convert(String s, int nRows) {

        StringBuilder result = new StringBuilder();
        int length = s.length();

        if ( nRows == 1) {
            return s;
        } else {
            int index = 0;
            /*
             * the first line
             */
            index = 0;
            while ( index < length ) {
                result.append(s.charAt(index));
```

```

        index += 2*(nRows-1);
    }

    /*
     * others lines
     */
    int flag = 0;
    for ( int i = 1; i < nRows-1; i++ ) {
        index = i;
        flag = 0;
        while ( index < length ) {
            if ( flag == 0 ) {
                result.append(s.charAt(index));
                index += 2*(nRows-i-1);
                flag = 1;
            } else {
                result.append(s.charAt(index));
                index += 2*i;
                flag = 0;
            }
        }
    }

    /*
     * the last line
     */
    index = nRows-1;
    while ( index < length ) {
        result.append(s.charAt(index));
        index += 2*(nRows-1);
    }
}

return result.toString();

}

}

```


4. MinStack

```
/*
 * Design a stack that supports push, pop, top, and retrieving the
 minimum element in constant time.
 * push(x) -- Push element x onto stack.
 * pop() -- Removes the element on top of the stack.
 * top() -- Get the top element.
 * getMin() -- Retrieve the minimum element in the stack.
 */
class MinStack {

    private int[] elements = null;
    private int length;
    private int size;
    private int minIndex;

    public MinStack () {
        elements = new int[50];
        length = 50;
        size = 0;
        minIndex = 0;
    }

    public void push(int x) {
        if ( isFull() ) {
            int[] temp = new int[length*2];
            for ( int i = 0; i < length; i++ ) {
                temp[i] = elements[i];
            }
            elements = temp;
            length *= 2;
            if ( x < elements[minIndex] ) {
                minIndex = size;
            }
            elements[size++] = x;
        } else if ( isEmpty() ) {
            elements[size++] = x;
            minIndex = 0;
        } else {
            elements[size] = x;
            if ( x < elements[minIndex] ) {
                minIndex = size;
            }
        }
    }
}
```

```

        }
        size++;
    }
}

public void pop() {
    if ( !isEmpty() ) {
        if ( minIndex == size-1 ) {
            // compute minIndex again
            minIndex = 0;
            for ( int i = 1; i < size-1; i++ ) {
                if ( elements[i] < elements[minIndex] ) {
                    minIndex = i;
                }
            }
        }
        size--;
    }
}

public int top() {
    return elements[size-1];
}

public int getMin() {
    return elements[minIndex];
}

private boolean isFull() {
    return size==length;
}

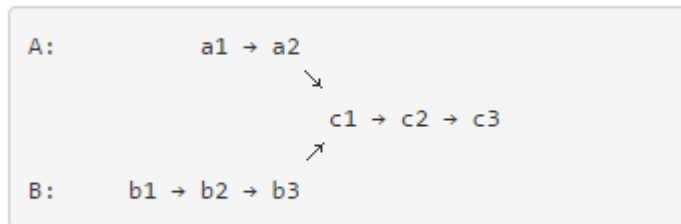
private boolean isEmpty () {
    return size==0;
}
}

```

5. Intersection of Two Linked Lists

问题描述

For example, the following two linked lists:



begin to intersect at node c1.

Notes:

- If the two linked lists have no intersection at all, return `null`.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in O(n) time and use only O(1) memory.

分析

时间复杂度

源代码

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}

public class Solution {
```

```

    public ListNode getIntersectionNode(ListNode headA, ListNode
headB) {

        /*
         * The time should be  $O(n)$ , thus we can not use two loops
to find the answer.
         * But if the two linked lists intersect, the length of the
intersected part must be the same.
        */
        ListNode result = null;
        ListNode tempA = headA;
        ListNode tempB = headB;
        int lengthA = 0;
        int lengthB = 0;

        // find the length of list A
        while ( tempA != null ) {
            tempA = tempA.next;
            lengthA++;
        }
        // find the length of list B
        while ( tempB != null ) {
            tempB = tempB.next;
            lengthB++;
        }
        // if anyone of the parameter is null, the result is null
        if ( lengthA == 0 ||
            lengthB == 0 ) {

        } else {
            int count = 0;
            tempA = headA;
            tempB = headB;
            if ( lengthA > lengthB ) {
                count = lengthA - lengthB;
                while ( count != 0 ) {
                    tempA = tempA.next;
                    count--;
                }
            } else {
                count = lengthB - lengthA;
                while ( count != 0 ) {
                    tempB = tempB.next;
                    count--;
                }
            }
        }
    }
}

```

```
        }  
    }  
    // find the intersection index  
    while ( tempA != tempB ) {  
        tempA = tempA.next;  
        tempB = tempB.next;  
    }  
    result = tempA;  
}  
  
return result;  
  
}
```

6. Reverse Integer

Reverse digits of an integer.

Example1: x = 123, return 321

Example2: x = -123, return -321

考虑到了末尾为 0 的情况，但是没有考虑到超过整型值范围的情况。

```
public class Solution {
    public int reverse(int x) {

        long result = 0L;
        int temp = x;
        if ( x < 0 ) {
            temp = -temp;
        }

        while ( temp != 0 ) {
            result = result*10 + temp%10;
            temp /= 10;
        }

        if ( x < 0 ) {
            result = -result;
        }

        if ( result > Integer.MAX_VALUE ||
            result < Integer.MIN_VALUE ) {
            return 0;
        } else {
            return (int)result;
        }
    }
}
```

7. N-Queens

The n -queens puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens attack each other.

Given an integer n , return all distinct solutions to the n -queens puzzle.

Each solution contains a distinct board configuration of the n -queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

For example,

There exist two distinct solutions to the 4-queens puzzle:

```
[
  [".Q..", // Solution 1
   "...Q",
   "Q...",
   "..Q."],

  [".Q..", // Solution 2
   "Q...",
   "...Q",
   ".Q.."]
]
```

回溯法解 N 皇后问题源代码

```
import java.util.ArrayList;
import java.util.List;

public class Solution {

    private List<String[]> list = null;
    private String[] aResult = null;
    private byte[][] queenPositions = null;
    private int queenNumbers = 0;

    public List<String[]> solveNQueens(int n) {

        list = new ArrayList<String[]>();
        queenPositions = new byte[n][n];
        for ( int i = 0; i < n; i++ ) {
            for ( int j = 0; j < n; j++ ) {
                queenPositions[i][j] = 0;
            }
        }
        queenNumbers = n;

        placeQueens(0, 0);

        return list;
    }
}
```

```

}

/**
 *
 * @param line 下一个要处理的行
 * @param flag 继续向前放置还是回溯：0表示继续放置，1表示回溯
 */
private void placeQueens ( int line, int flag ) {

    while ( line != -1 ) {
        if ( flag == 0 ) {
            if ( line == queenNumbers ) {
                /*
                 * 说明已经找到一种方法
                 */
                aResult = new String[queenNumbers];
                StringBuilder sBuilder = null;
                for ( int i = 0; i < queenNumbers; i++ ) {
                    sBuilder = new StringBuilder();
                    for ( int j = 0; j < queenNumbers; j++ ) {
                        if ( queenPositions[i][j] == 0 ) {
                            sBuilder.append('.');
                        } else {
                            sBuilder.append('Q');
                        }
                    }
                    aResult[i] = sBuilder.toString();
                }
                list.add( aResult );
                /*
                 * 回溯
                 */
                flag = 1;
                line--;
            } else {
                int i = 0;
                for ( i = 0; i < queenNumbers; i++ ) {
                    if ( canPlace( line, i ) ) {
                        queenPositions[line][i] = 1;
                        line++;
                        break;
                    }
                }
            }
        }
    }
}

```



```

        /*
        * 可能出现一行都无法放置的情况
        */
        if ( i == queenNumbers ) {
            flag = 1;
            line--;
        }
    }
} else {
    /*
    * 当第一层的回溯结束时，算法结束
    */
    if ( line == -1 ) {
        return;
    }
    int i = 0;
    for ( i = 0; i < queenNumbers; i++ ) {
        if ( queenPositions[line][i] == 1 ) {
            queenPositions[line][i] = 0;
            break;
        }
    }
    if( i != queenNumbers-1) {
        i += 1;
        while ( i < queenNumbers ) {
            if ( canPlace( line, i ) ) {
                queenPositions[line][i] = 1;
                line++;
                flag = 0;
                break;
            }
            i++;
        }
        if ( i == queenNumbers ) {
            line--;
        }
    } else {
        line--;
    }
}
}
}
}

```

```

/**
 * 判断是否能在第line(0<=line<=n-1)行, 第i(0<=i<=n-1)列的位置放
置
 * @param line
 * @param col
 * @return
 */
private boolean canPlace ( int line, int col ) {
    for ( int i = 0; i < line; i++ ) {
        if ( queenPositions[i][col] == 1 ||          // 垂线上是否有
皇后
            (col-line+i >= 0 && queenPositions[i][col-line+i] ==
1) || // 135度对角线上是否有皇后
            (col+line-i < queenNumbers &&
queenPositions[i][col+line-i] == 1) ) { // 45度对角线上是否有皇
后
            return false;
        }
    }
    return true;
}
}

```

8. Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only *once*.

For example,

Given `1->1->2`, return `1->2`.

Given `1->1->2->3->3`, return `1->2->3`.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode first = head;
        ListNode second = null;

        if ( first != null ) {
            second = first.next;
        }

        while ( second != null ) {
            if ( first.val == second.val ) {
                first.next = second.next;
                second = first.next;
            } else {
                first = second;
                second = first.next;
            }
        }

        return head;
    }
}
```

9. Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

[click to show spoilers.](#)

Some hints:

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

```
public class Solution {  
  
    public boolean isPalindrome(int x) {  
  
        if ( x < 0 ) {  
            return false;  
        }  
  
        String str = Integer.toString(x);  
        int length = str.length();  
        int count = length/2 - 1;  
        for ( int i = 0; i <= count; i++ ) {  
            if ( str.charAt(i) != str.charAt(length-1-i) ) {  
                return false;  
            }  
        }  
  
        return true;  
    }  
}
```

10. Best Time to Buy and Sell Stock

Say you have an array for which the i^{th} element is the price of a given stock on day i .

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

我的解题思路

想了好久才想出正确的结果，思路都在代码中。

```
import java.util.ArrayList;
import java.util.List;

public class Solution {

    public int maxProfit(int[] prices) {

        /*
         * 找出每一个上升段的最大最小值，分别记录在两个容器中，再依次遍历两个容器找出最大的差值
         */
        int max = 0, min = 0;
        int result = 0;
        if ( prices == null ) {
            return 0;
        }
        int length = prices.length;
        if ( length <= 1 ) {
            return 0;
        }

        List<Integer> minList = new ArrayList<Integer>();
        List<Integer> maxList = new ArrayList<Integer>();

        for ( int i = 0; i < length-1; i++ ) {
            if ( prices[i] < prices[i+1] ) {
                min = i;
                while ( i < length-1 && prices[i] <= prices[i+1] ) {
                    i++;
                }
                max = i;

                minList.add(min);
```

```

        maxList.add(max);
    /*
    * 第3次
    * 找出整个数组对应折线的每个上升段，找到一个后与前一个
    进行合并，合并时比较3种情况，选出最大的值
    * 运行时出错， 提升结果错误
    * 错误原因：局部最优不能导致整体最优，比如前两段分别是
    [10,20],[8,15],则合并的结果是[10,20],最大值是10，当与第3端[11,25]
    合并时，
    * 结果为[10,25],最大值为15，但实际上此时的最大值应该是
    [8,25]
    */
    /*
    if ( prices[max2]-prices[min2] >
prices[max2]-prices[min1] ) {
        temp1 = min2;
        temp2 = max2;
    } else {
        temp1 = min1;
        temp2 = max2;
    }
    if ( prices[max1]-prices[min1] >
prices[temp2]-prices[temp1] ) {
        temp1 = min1;
        temp2 = max1;
    }
    min1 = temp1;
    max1 = temp2;
    */
}

}

int count = minList.size();
int temp;
for ( int i = 0; i < count; i++ ) {
    for ( int j = i; j < count; j++ ) {
        temp =
prices[maxList.get(j)]-prices[minList.get(i)];
        if ( temp > result ) {
            result = temp;
        }
    }
}
}

```

```
return result;
```

```
/*
```

```
 * 第二次
```

```
*/
```

```
/*
```

```
int result = 0;
```

```
int max = 0, min = 0;
```

```
int length = prices.length;
```

```
int i, j;
```

```
for ( i = 0; i < length-1; i++ ) {
```

```
    max = i;
```

```
    min = i;
```

```
    for ( j = i+1; j < length; j++ ) {
```

```
        if ( prices[j] > prices[max] ) {
```

```
            max = j;
```

```
        }
```

```
    }
```

```
    for ( j = i+1; j < max; j++ ) {
```

```
        if ( prices[j] < prices[min] ) {
```

```
            min = j;
```

```
        }
```

```
    }
```

```
    if ( prices[max] - prices[min] > result ) {
```

```
        result = prices[max] - prices[min];
```

```
    }
```

```
    i = max;
```

```
}
```

```
return result;
```

```
*/
```

```
/*
```

```
 * 第一次：直接找出整个数组中最大和最小的元素，差值即为结果
```

```
 * 错误结果：输入[2,1]，输出1，实际应为0
```

```
*/
```

```
/*
```

```
int min = 0;
```

```
int max = 0;
```

```
int length = prices.length;
```

```
if ( length == 0 ) {
```

```

        return 0;
    }

    for ( int i = 1; i < length; i++ ) {
        if ( prices[i] < prices[min] ) {
            min = i;
        } else if ( prices[i] > prices[max] ) {
            max = i;
        }
    }

    if ( max > min ) {
        return prices[max]-prices[min];
    } else {

    }

    */
}
}
}

```

比较好的一种思路

```

public class Solution {
    public int maxProfit(int[] prices) {

        if(prices == null)return 0;
        if(prices.length < 2)return 0;

        int maxProfit = 0, min = prices[0], profitNow = 0;

        for(int i=1; i<prices.length; ++i){

            if(prices[i] > prices[i-1]){
                profitNow = prices[i] - min;
                maxProfit = profitNow > maxProfit ? profitNow : maxProfit;

            }else{

                min = prices[i] < min ? prices[i] : min;
            }

        }

        return maxProfit;

    }
}

```


11. Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

```
public class Solution {

    public String longestCommonPrefix(String[] strs) {

        StringBuilder result = new StringBuilder();

        char c;
        if ( strs == null || strs.length == 0 ) {

        } else {
            int i, j;
            for ( i = 0; i < strs[0].length(); i++ ) {
                c = strs[0].charAt(i);
                for ( j = 1; j < strs.length; j++ ) {
                    if ( i >= strs[j].length() || strs[j].charAt(i) !=
c ) {
                        /*
                         * 不能用break，因为一旦发现不匹配的，后面的都不
用比较了，这里必须跳出两次循环
                        */
                        //break;
                        return result.toString();
                    }
                }
                if ( j == strs.length ) {
                    result.append(c);
                }
            }
        }

        return result.toString();

    }

}
```

12. Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,

"A man, a plan, a canal: Panama" is a palindrome.

"race a car" is *not* a palindrome.

Note:

Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

```
public class Solution {
    public boolean isPalindrome(String s) {

        if ( s == null ) {
            return false;
        }
        if ( s.equals("") ) {
            return true;
        }

        int i = 0;
        int j = s.length()-1;
        char c1, c2;
        while ( i < j ) {
            c1 = s.charAt(i);
            while ( (i < j) &&
                !( ( c1 >= 48 && c1 <= 57 ) ||
                  ( c1 >= 65 && c1 <= 90 ) ||
                  ( c1 >= 97 && c1 <= 122 ) ) ) {
                i++;
                c1 = s.charAt(i);
            }
            if ( i == j ) {
                return true;
            }
            c2 = s.charAt(j);
            while ( (i < j) &&
                !( ( c2 >= 48 && c2 <= 57 ) ||
                  ( c2 >= 65 && c2 <= 90 ) ||
                  ( c2 >= 97 && c2 <= 122 ) ) ) {
                j--;
                c2 = s.charAt(j);
            }
            if ( c1 != c2 ) {
                return false;
            }
            i++;
            j--;
        }
        return true;
    }
}
```

```
        j--;
        c2 = s.charAt(j);
    }
    if ( i == j ) {
        return true;
    }
    //System.out.println(i + "," + j);
    if ( c1==c2 || c1-c2==32 || c2-c1==32 ) {
        i++;
        j--;
    } else {
        return false;
    }
}

return true;
}
}
```

13. Pascal's Triangle

Given *numRows*, generate the first *numRows* of Pascal's triangle.

For example, given *numRows* = 5,

Return

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

```
public class Solution {
    public List<List<Integer>> generate(int numRows) {

        List<List<Integer>> result = new
ArrayList<List<Integer>>();
        List<Integer> list = null;
        List<Integer> last = null;

        int i, j;
        for ( i = 0; i < numRows; i++ ) {
            list = new ArrayList<Integer>();
            list.add(1);
            if ( i > 0 ) {
                last = result.get(i-1);
                for ( j = 1; j < i; j++ ) {
                    list.add(last.get(j-1)+last.get(j));
                }
                list.add(1);
            }
            result.add(list);
        }

        return result;
    }
}
```

14. Pascal's Triangle 2

Given an index k , return the k^{th} row of the Pascal's triangle.

For example, given $k = 3$,

Return `[1,3,3,1]`.

Note:

Could you optimize your algorithm to use only $O(k)$ extra space?

```
public class Solution2 {
    public List<Integer> getRow(int rowIndex) {

        if ( rowIndex < 0 ) {
            return null;
        }

        List<Integer> result = new ArrayList<Integer>();
        int []elements = new int[rowIndex+1];

        if ( rowIndex == 0 ) {
            elements[0] = 1;
        } else if ( rowIndex == 1 ) {
            elements[0] = 1;
            elements[1] = 1;
        } else {
            elements[0] = 1;
            int i, j;
            for ( i = 1; i <= rowIndex; i++ ) {
                for ( j = i; j > 0; j-- ) {
                    elements[j] += elements[j-1];
                }
            }
        }

        for ( int i = 0; i <= rowIndex; i++ ) {
            result.add(elements[i]);
        }

        return result;
    }
}
```

15. Merge Sorted Array

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note:

You may assume that A has enough space (size that is greater or equal to $m + n$) to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

```
/*
 * 将B数组的内容合并到A数组中，假设A的容量足够大，A、B都已经排好序
 */
public class Solution {
    // 假设是按升序排列
    public void merge(int A[], int m, int B[], int n) {
        int i = 0, j = 0, k = 0;
        for ( i = m-1; i >= 0; i-- ) {
            A[i+n] = A[i];
        }

        i = n;
        j = 0;
        k = 0;

        while ( i <= m+n-1 &&
                j <= n-1 ) {
            if ( A[i] < B[j] ) {
                A[k++] = A[i++];
            } else {
                A[k++] = B[j++];
            }
        }
        /*
         * 继续放置剩余的元素
         */
        if ( i == m+n ) {
            while ( j <= n-1 ) {
                A[k++] = B[j++];
            }
        } else {
            while ( i <= m+n-1 ) {
                A[k++] = A[i++];
            }
        }
    }
}
```

16. Remove Duplicates from Sorted Array

Given a sorted array, remove the duplicates in place such that each element appear only *once* and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

For example,

Given input array A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

```
public class Solution {
    public int removeDuplicates(int[] A) {

        if ( A == null ||
            A.length == 0 ) {
            return 0;
        }

        int i = 0, j = 1;
        while ( j < A.length ) {
            if ( A[i] != A[j] ) {
                i++;
                A[i] = A[j];
                j++;
            } else {
                j++;
            }
        }

        return i+1;
    }
}
```

17. Add Binary

Given two binary strings, return their sum (also a binary string).

For example,

a = "11"

b = "1"

Return "100".

```
public class Solution {
    public String addBinary(String a, String b) {

        if ( a == null ) {
            return b;
        }
        if ( b == null ) {
            return a;
        }

        StringBuilder strBuilder = new StringBuilder();
        int alength = a.length();
        int blength = b.length();
        int jinwei = 0;
        int temp;
        if ( alength < blength ) {
            int count = blength - alength;
            for ( int i = alength - 1; i >= 0; i-- ) {
                temp = (a.charAt(i) - '0') + (b.charAt(i + count) - '0') +
jinwei;
                strBuilder.insert(0, (temp % 2));
                // strBuilder.insert(0, (temp % 2 + '0'));
                jinwei = temp / 2;
            }
            for ( int i = count - 1; i >= 0; i-- ) {
                temp = (b.charAt(i) - '0') + jinwei;
                strBuilder.insert(0, (temp % 2));
                jinwei = temp / 2;
            }
            // 最后的进位
            if ( jinwei == 1 ) {
                strBuilder.insert(0, '1');
            }
        }
    }
}
```



```

    }
} else {
    int count = alength-blength;
    for ( int i = blength-1; i >= 0; i-- ) {
        temp = (b.charAt(i)-'0') + (a.charAt(i+count)-'0') +
jinwei;
        strBuilder.insert(0, (temp%2));
        jinwei = temp / 2;
    }
    for ( int i = count-1; i >= 0; i-- ) {
        temp = (a.charAt(i)-'0') + jinwei;
        strBuilder.insert(0, (temp%2));
        jinwei = temp / 2;
    }
    // 最后的进位
    if ( jinwei == 1 ) {
        strBuilder.insert(0, '1');
    }
}

return strBuilder.toString();

}
}

```

18. Length of Last Word

Given a string *s* consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,

Given *s* = "Hello World",

return 5.

```
public class Solution {
    public int lengthOfLastWord(String s) {
        if ( s == null || s.equals("") ) {
            return 0;
        }

        int length = s.length();
        int result = 0;
        boolean begin = false;
        int i = 0;
        for ( i = length-1; i >= 0; i-- ) {
            if ( s.charAt(i) != ' ' ) {
                begin = true;
                break;
            }
        }

        if ( begin ) {
            while ( i >= 0 && s.charAt(i) != ' ' ) {
                result++;
                i--;
            }
        }

        return result;
    }
}
```

19. Add Two Number

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 → 4 → 3) + (5 → 6 → 4)

Output: 7 → 0 → 8

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}

public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        if ( l1 == null ) {
            return l2;
        } else if ( l2 == null ) {
            return l1;
        } else {
            ListNode temp1 = l1;
            ListNode temp2 = l2;
            ListNode node = null;
            ListNode result = null;
            ListNode current = null;
            int jinwei = 0;
            int sum = 0;

            // 头结点
            if ( temp1 != null && temp2 != null ) {
                sum = temp1.val + temp2.val;
                jinwei = sum/10;
                result = new ListNode(sum%10);
                current = result;
                temp1 = temp1.next;
                temp2 = temp2.next;
            }

            // 中间的元素
            while ( temp1 != null && temp2 != null ) {
                sum = temp1.val + temp2.val + jinwei;
                jinwei = sum/10;
```

```

        node = new ListNode(sum%10);
        current.next = node;
        current = node;
        temp1 = temp1.next;
        temp2 = temp2.next;
    }
    // 剩余没有加完的
    if ( temp1 == null ) {
        while ( temp2 != null ) {
            sum = temp2.val + jinwei;
            jinwei = sum/10;
            node = new ListNode(sum%10);
            current.next = node;
            current = node;
            temp2 = temp2.next;
        }
        if ( jinwei != 0 ) {
            node = new ListNode(jinwei);
            current.next = node;
        }
    } else {
        while ( temp1 != null ) {
            sum = temp1.val + jinwei;
            jinwei = sum/10;
            node = new ListNode(sum%10);
            current.next = node;
            current = node;
            temp1 = temp1.next;
        }
        if ( jinwei != 0 ) {
            node = new ListNode(jinwei);
            current.next = node;
        }
    }
    return result;
}

}
}


```

20. Implement strStr()

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Update (2014-11-02):

The signature of the function had been updated to return the *index* instead of the pointer. If you still see your function signature returns a `char *` or `String`, please click the reload button  to reset your code definition.

```
public class Solution {
    public int strStr(String haystack, String needle) {
        if ( haystack == null ||
            needle == null ) {
            return -1;
        }
        int length1 = haystack.length();
        int length2 = needle.length();

        int i, j;
        for ( i = 0; i <= length1-length2; i++ ) {
            for ( j = 0; j < length2; j++ ) {
                if ( haystack.charAt(i+j) != needle.charAt(j) ) {
                    break;
                }
            }
            if ( j == length2 ) {
                return i;
            }
        }
        return -1;
    }
}
```

21. Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,



The root-to-leaf path 1->2 represents the number 12.

The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

基本思路

使用一个辅助栈进行宽度优先搜索（BFS），搜索过程中将每个节点的值修改为其对应的数值大小，最后对所有的叶子节点值进行求和。

源代码如下：

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Solution {
    public int sumNumbers(TreeNode root) {

        if ( root == null ) {
            return 0;
        }

        MStack stack = new MStack();
        TreeNode temp = null;
        int result = 0;

        stack.push(root);
```

```

        while ( !stack.isEmpty() ) {
            temp = stack.pop();
            if ( temp.left == null &&
                temp.right == null ) {
                result += temp.val;
            } else {
                if ( temp.right != null ) {
                    temp.right.val += temp.val*10;
                    stack.push(temp.right);
                }
                if ( temp.left != null ) {
                    temp.left.val += temp.val*10;
                    stack.push(temp.left);
                }
            }
        }

        return result;
    }
}

class MStack {
    private TreeNode[] elements;
    private int size;
    private int length;

    public MStack () {
        length = 50;
        elements = new TreeNode[length];
        size = 0;
    }

    public TreeNode pop () {
        return elements[--size];
    }

    public void push ( TreeNode node ) {
        if ( !isFull() ) {
            elements[size++] = node;
        } else {
            TreeNode[] temp = new TreeNode[length*2];
            for ( int i = 0; i < length; i++ ) {
                temp[i] = elements[i];
            }

```

```

        temp[size++] = node;
        length *= 2;
        elements = temp;
    }
}

public boolean isEmpty () {
    return size==0;
}

private boolean isFull () {
    return size==length;
}
}

```

不修改节点值的思路

在上诉思路中，将节点入栈时修改了其 **val** 值，为了不对节点的值进行修改，可以建立一个新的对象，包含一个 **TreeNode** 对象和一个值，这样即可避免对原节点的值进行修改。

22. Remove Duplicates from Sorted List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.

For example,

Given `1->2->3->3->4->4->5`, return `1->2->5`.

Given `1->1->1->2->3`, return `2->3`.

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}

public class Solution {
    public ListNode deleteDuplicates(ListNode head) {

        ListNode equalBefore = head; // 相等元素前的元素
        ListNode equalBegin = null; // 第一个相等的元素
        ListNode equalEnd = null; // 最后一个相等的元素

        equalBegin = equalBefore;
        if ( equalBegin != null ) {
            equalEnd = equalBegin.next;
        }

        /*
         * 如果最前面几个元素相等的情况，则先删除
         */
        while ( equalEnd != null && equalBegin == head ) {
            if ( equalBegin.val == equalEnd.val ) {
                do {
                    equalEnd = equalEnd.next;
                } while ( equalEnd != null && equalBegin.val ==
equalEnd.val );
                head = equalEnd;
                equalBegin = head;
                if ( equalBegin != null ) {
                    equalEnd = equalBegin.next;
                }
            } else {
```

```

        equalBegin = equalEnd;
        equalEnd = equalBegin.next;
    }
}

/*
 *
 */
equalBefore = head; // 此时head已经变化，必须对equalBefore
重新赋值
while ( equalEnd != null ) {
    if ( equalBegin.val == equalEnd.val ) {
        do {
            equalEnd = equalEnd.next;
        } while ( equalEnd != null && equalBegin.val ==
equalEnd.val );
        equalBefore.next = equalEnd;
        equalBegin = equalEnd;
        if ( equalBegin != null ) {
            equalEnd = equalBegin.next;
        }
    } else {
        equalBefore = equalBegin;
        equalBegin = equalEnd;
        equalEnd = equalBegin.next;
    }
}

return head;
}
}

```

23. Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}

public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if ( l1 == null ) {
            return l2;
        } else if ( l2 == null ) {
            return l1;
        } else {
            ListNode current1 = l1;
            ListNode before1 = null;
            ListNode temp1 = null;
            ListNode current2 = l2;
            ListNode before2 = null;
            ListNode temp2 = null;
            if ( current1.val <= current2.val ) {
                before1 = current1;
                current1 = before1.next;
                while ( current1 != null && current2 != null ) {
                    if ( current1.val <= current2.val ) {
                        before1 = current1;
                        current1 = before1.next;
                    } else {
                        temp2 = current2.next;
                        current2.next = current1;
                        before1.next = current2;
                        current2 = temp2;
                        before1 = before1.next;
                    }
                }
            }
            if ( current1 == null ) {
                before1.next = current2;
            }
        }
    }
}
```

```

    }
    return l1;
} else {
    before2 = current2;
    current2 = before2.next;
    while ( current1 != null && current2 != null ) {
        if ( current2.val <= current1.val ) {
            before2 = current2;
            current2 = before2.next;
        } else {
            temp1 = current1.next;
            current1.next = current2;
            before2.next = current1;
            current1 = temp1;
            before2 = before2.next;
        }
    }
    if ( current2 == null ) {
        before2.next = current1;
    }
    return l2;
}
}
}
}
}

```

存在的问题

对输入的参数内容进行了修改，而且在程序外可见，这种禁止这种现象，最好重新建立一个链表表示合并后的结果。