

1、三角形最小路径和 (动态规划)

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

相邻的结点 在这里指的是 下标与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。

链接：<https://leetcode-cn.com/problems/triangle>

错：

比较下一层，选取最小的，然后继续比较下一层。。。实际上，可能存在其他路径，某一层可能较大，但整体最小

```
#include "pch.h"
#include <iostream>
#include <vector>

using namespace std;

// 使用递归进行查找
class Solution {
public:

    void Moving(int index, int i, vector<vector<int>>& triangle, int& sum)
    {
        if (i >= triangle.size()) return;

        if (triangle[i][index] > triangle[i][index + 1])
        {
            sum = sum + triangle[i][index + 1];
            Moving(++index, ++i, triangle, sum);
        }
        else
        {
            sum = sum + triangle[i][index];
            Moving(index, ++i, triangle, sum);
        }
    }

    int minimumTotal(vector<vector<int>>& triangle)
    {
        int sum = 0;
        if(triangle.size()>1)
            Moving(0, 1, triangle,sum);
        return sum+triangle[0][0];
    }
};

int main()
{
    vector<vector<int>> triangle = {{ 2 }, {3, 4 }, {6, 5, 7 }, {4, 1, 8, 3 } };
    // vector<vector<int>> triangle = {{ -1 }, {2, 3 }, {6, 5, 71, -1, -3 }}; 测试错误，结果应为-1
    Solution s;
    cout<<s.minimumTotal(triangle);
}
```

利用动态规划解题：需要找到状态变量（Dp），写出状态方程。

```
class Solution {  
  
public:  
  
    int minimumTotal(vector<vector<int>>& triangle)  
  
    {  
  
        •     int n = triangle.size();  
  
        •     vector<vector<int>> result(n, vector<int>(n, 0)); // 二维数组用来存放某点到头的路径  
  
        •     for (int i = 0; i < triangle.size(); i++)  
  
        •     {  
  
            •         for (int j = 0; j < triangle[i].size(); j++)  
  
            •         {  
  
                •             if (i == 0) result[i][j] = triangle[i][j];  
  
                •             if ((j == 0) && (i)) result[i][j] = result[i-1][j] + triangle[i][j];  
  
                •             if (i == j && i) result[i][j] = result[i - 1][j-1] + triangle[i][j];  
  
                •             if (j < i && (j > 0)) result[i][j] = min(result[i - 1][j - 1], result[i - 1][j]) + triangle[i][j];  
  
            •         }  
  
            •     }  
  
        •     return *min_element(result[n-1].begin(), result[n-1].end());  
  
    }  
  
};
```

三步问题

三步问题。有个小孩正在上楼梯，楼梯有n阶台阶，小孩一次可以上1阶、2阶或3阶。实现一种方法，计算小孩有多少种上楼梯的方式。结果可能很大，你需要对结果模1000000007。

来源：力扣（LeetCode）

链接：<https://leetcode-cn.com/problems/three-steps-problem-lcci>

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

```
class Solution {  
  
public:  
  
    int waysToStep(int n)
```

```

    {
        vector<int> f(n);

        for (int i = 1; i <= n; i++)
        {
            switch(i)
            {
                case 1:f[0] = 1; break;
                case 2:f[1] = 2; break;
                case 3:f[2] = 4; break;
                default:
                    f[i-1] = ((f[i - 2] + f[i - 3]) % 1000000007+ f[i - 4])% 1000000007;
            }
        }
        return f[n-1];
    }
};

```

最小路径和

给定一个包含非负整数的 $m \times n$ 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

假设点 (x, y) 时的状态为 $c(x,y)$ ，则向前推可得状态方程： $f(x,y) = \min(f(x,y-1), f(x-1,y)) + c(x,y)$

但需考虑边界问题 ($x=0$ or $y=0$ 时，无需进行 \min 判断，直接相加即可。)

```

class Solution {

public:
    int minPathSum(vector<vector<int>>& grid)
    {
        int row = grid.size(); // 取行数
        vector<vector<int>> result(row, vector<int>(grid[0].size())); // 存取某点到右上角
        // 的路径和

        // 分别求每一点到右上角的最小路径和
    }
};

```

```

•     for (int i = 0; i < row; i++)
•     {
•         for (int j = 0; j < grid[0].size(); j++)
•         {
•             if ((i == 0) && (j == 0)) result[i][j] = grid[i][j];
•             else if (i == 0) result[i][j] = result[i][j - 1] + grid[i][j];
•             else if (j == 0) result[i][j] = result[i - 1][j] + grid[i][j];
•             else result[i][j] = min(result[i - 1][j], result[i][j - 1]) + grid[i][j];
•         }
•     }
•     return result[row - 1][grid[0].size() - 1]; // 注意返回的是什么，本题要求到右下角，故
      返回最后一个
}

};


```

乘积最大子数组

给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

$$\begin{array}{ll}
 c(x) > 0 & \begin{cases} 0 < f(x-1) < 1 \\ f(x-1) > 1 \end{cases} \\
 & \begin{array}{l} c(x) \\ c(x) \cdot \max(f(x-1)) \end{array} \Rightarrow \max \left[c(x), c(x) \cdot \max[f(x-1)] \right]
 \end{array}$$

$$\begin{array}{ll}
 c(x) < 0 & \begin{cases} -1 < f(x-1) < 0 \\ f(x-1) > -1 \end{cases} \\
 & \begin{array}{l} c(x) \\ c(x) \cdot \min(f(x-1)) \end{array} \Rightarrow \max \left[c(x), c(x) \cdot \min[f(x-1)] \right]
 \end{array}$$

```

class Solution {
public:
    int maxProduct(vector<int>& nums)
    {
        int n = nums.size();
        vector<int> result_max(n);
        vector<int> result_min(n);
    }
};


```

```

•     for (int i = 0; i < n; i++)
•     {
•         if (i == 0)
•         {
•             result_max[i] = nums[i];
•             result_min[i] = nums[i];
•         }
•         else if (nums[i] >= 0)
•         {
•             result_max[i] = max(nums[i], nums[i]*result_max[i-1]);
•             result_min[i] = min(nums[i], nums[i]*result_min[i - 1]);
•         }
•         else
•         {
•             result_max[i] = max(nums[i], nums[i] * result_min[i - 1]);
•             result_min[i] = min(nums[i], nums[i] * result_max[i - 1]);
•         }
•     }
•     return *max_element(result_max.begin(),result_max.end());
}

```

上述代码和绘图没错，但是忽略了一点，即提干中说明是“整数”，及不存在元素大于-1小于1的情形。那么代码可进一步优化。

```

class Solution {
public:
    int maxProduct(vector<int>& nums)
    {
        int n = nums.size();
        vector<int> result_max(n,nums[0]);
        vector<int> result_min(n,nums[0]);

```

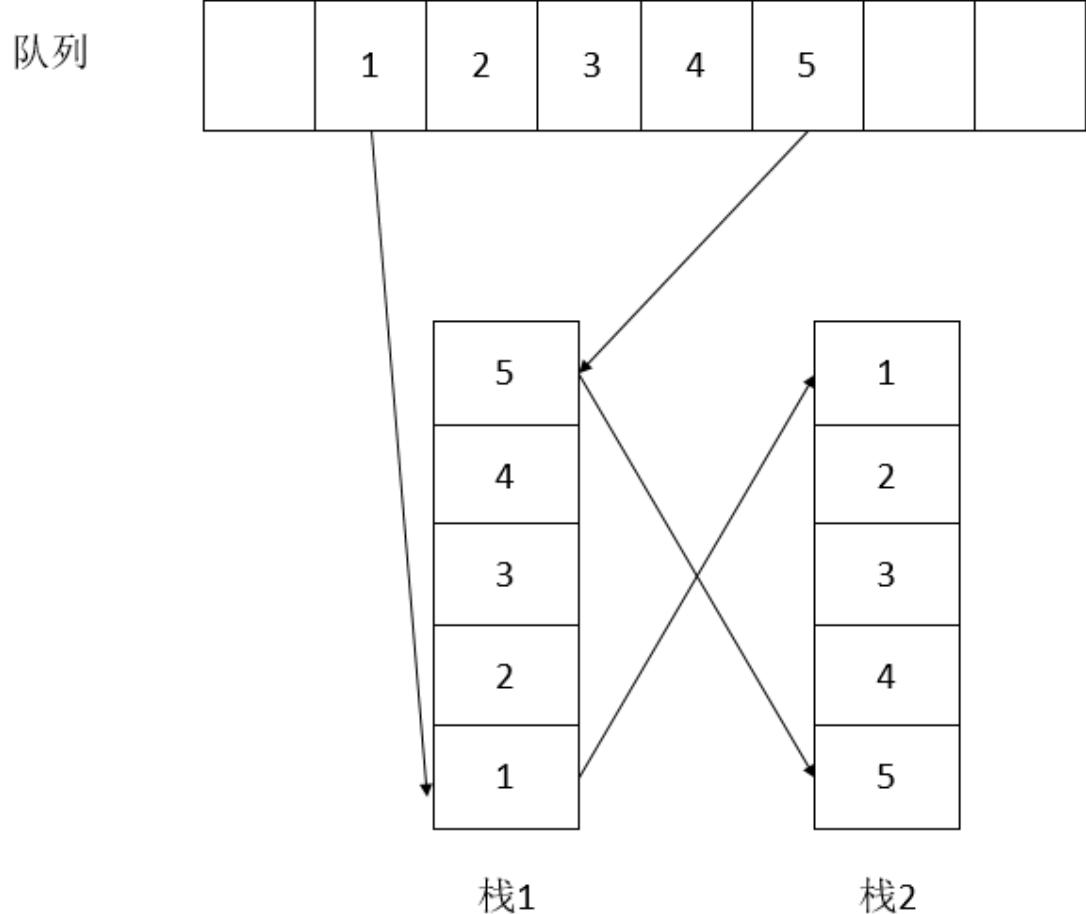
```
•
•     for (int i = 1; i < n; i++)
•     {
•         result_max[i] = max(nums[i] * result_min[i - 1], max(nums[i], nums[i] *
result_max[i - 1]));
•         result_min[i] = min(nums[i] * result_max[i - 1], min(nums[i], nums[i] *
result_min[i - 1]));
•
•     }
•     return *max_element(result_max.begin(), result_max.end());
}
};
```

2、栈（剑指Offer）

剑指 Offer 09. 用两个栈实现队列

题干：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)

分析：队列为先入先出型，栈为先入后出型。若使用两个栈，则某一元素先入后出（第一个栈），再转移到另一个栈。此时，栈顶元素即为第一个进第一个栈的元素。



代码：

```

class CQueue {

public:
    CQueue() {
        }

    void appendTail(int value) {
        •     s1.push(value);
        }

    int deleteHead() {
        •     int output;

        •     if (s2.empty()) // 此操作非常重要。假设插入插入删除 插入插入删除（删除时应该先将第一次
                      // 插入的删除完，再进行元素填充）
    }
}

```

```

    •     {

    •         while (!s1.empty())

    •         {

    •             s2.push(s1.top());

    •             s1.pop();

    •         }

    •     }

    •     if (s2.empty())

    •     {
    •         output = -1;

    •     }
    •     else

    •     {

    •         output = s2.top();

    •         s2.pop();

    •     }

    •     return output;

}

private:

stack<int> s1,s2;

};

```

注意：1、删除操作中，if(s2.empty())不可缺。。（只有当12345全部delete之后，才可往s2中加元素，不然，5可能最后才回pop）

2、while (!s1.empty()) ，重复循环直到将s1清空。

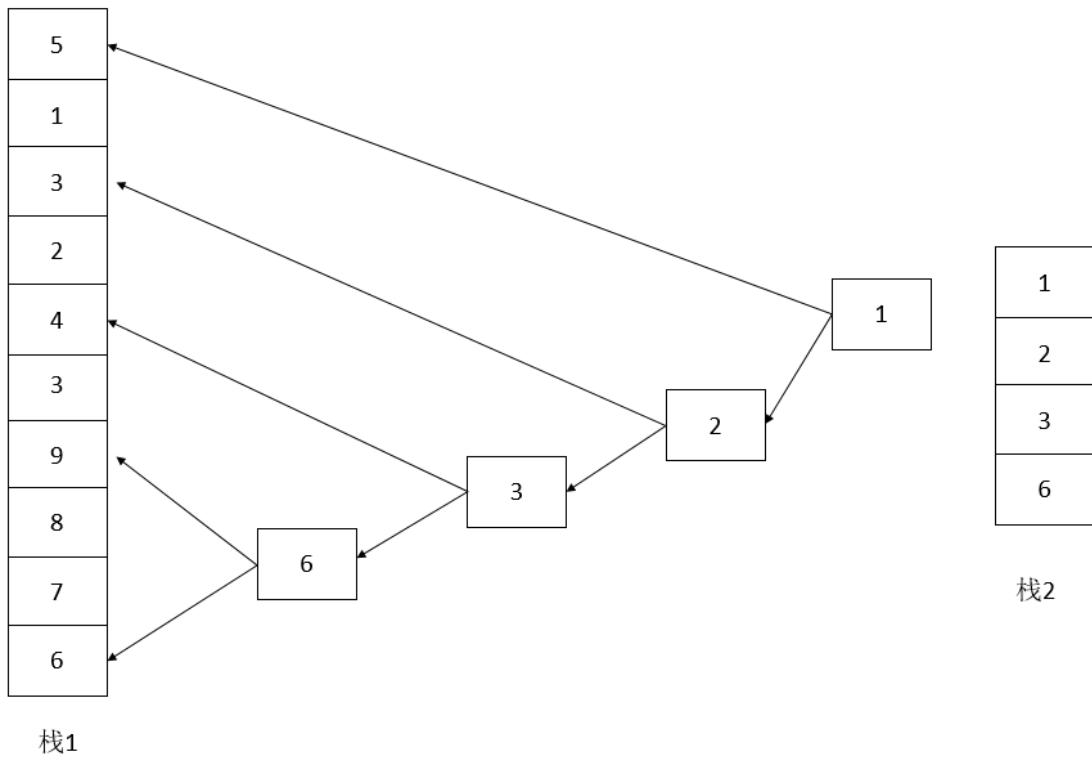
剑指 Offer 30. 包含min函数的栈

题干：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)。

分析：1、在定义的类型中创建一个属性，即min。push时，判断min与push元素的大小，并将小的放入min中。pop时，许判断min与pop元素是否一致，若一致，，则。。

2、上述分析在pop时无法实现的原因是因为，pop了当前最小元素后，无法更新新的最小元素。若能在一个地方，存储最小元素组，pop完最小，则出现次小元素，以此类推。

3、可采用另一个栈存最小元素。栈的优势在于先入后出，最小元素栈中，从栈顶到栈底从小到大排序，每一个元素都对应原栈的一段空间，且为该段空间的最小元素。



栈1

代码：

```

class MinStack {
public:
    /** initialize your data structure here. */
    MinStack() {

    }

    // push时需判断s2是否为空，若空则将元素push。若为空，则需比较top与x
    void push(int x) {
        s1.push(x);
        if (s2.empty()) s2.push(x);
        else if (s2.top() >= x) s2.push(x);    // =非常重要，因为pop时即使相等也得pop
    }

    void pop() {
        if (!s1.empty()) // pop时，一定要判断
        {
            if (s2.top() == s1.top()) s2.pop();
            s1.pop();
        }
        ;
    }

    int top() {
        if(!s1.empty())
            return s1.top();
        return -1;
    }

    int min() {
        if(!s2.empty())
            return s2.top();
        return -1;
    }
}

```

```
private:  
    stack<int> s1, s2;  
};
```

剑指 Offer 59 - II. 队列的最大值

题干：请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数max_value、push_back 和 pop_front 的均摊时间复杂度都是O(1)。

若队列为空，pop_front 和 max_value 需要返回 -1

分析：利用两个栈定义队列，再用一个栈存放最大值。 (x)

进一步分析：队列的特点是先进先出，若要判断其max或min，不能直接采用栈的方式。假设队列元素为 942353637。若采用栈的方式第一个元素9低于个进，同样也会第一个出去。由于后面数均小于9，辅助栈中也不存在其他元素，也就是说，9一出去，辅助栈为空，无法判断。。。。。那么需要利用一个辅助队列，辅助队列不仅要留最大数，还得留小于最大数的数。

按照下图进一步分析：

队列2中第一个进入的元素是9，若只保留最大数，则9出去后，辅助队列不再存在其他元素，无法求最大值，故需将7也放入队列2中，即步骤2-1。

之后再进8，由于8大于7，9出队列之后、8出队列之前，最大值均是8，7无作用，故可删除7，进8， (步骤2-2)

同样，9仅队列后，8也不发挥作用，故删除8 (步骤2-3)

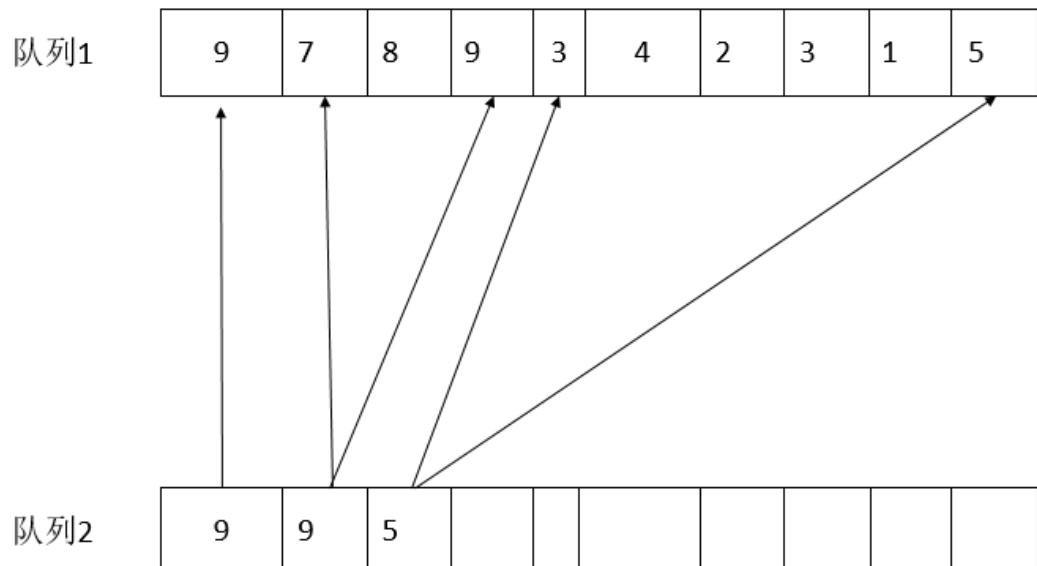
之后进入3，再进入4， $4 > 3$ ，故保留4即可。

以此类推

直到最后一个元素5进去，5大于3，故3删除，5大于4，4删除。辅助队列仅存995.

队列1	9	7	8	9	3	4	2	3	1	5
队列2-1	9	7								
队列2-2	9	8								
队列2-3	9	9								
队列2-4	9	9	4							
队列2-5	9	9	4	3						
队列2-6	9	9	4	3	5					
队列2-7	9	9	5							

总结起来，辅助队列中存在元素的作用为：辅助队列元素能掌控的范围为队列中其位置之前的元素。第2个9的范围为1-3.5的范围为4-9. (可将辅助队列看做是将军，即队列1中的小兵一定得被辅助队列的将军所掌管)



```
class MaxQueue {
public:
    MaxQueue() {
```

```
}
```

```
int max_value() {  
    if (q2.empty()) return -1;  
    return q2.front();  
}
```

```
void push_back(int value) {  
    // q1 push  
    q1.push_back(value);  
    // q2 push  
    while (!q2.empty() && q2.back() < value)  
    {  
        q2.pop_back();  
    }  
    q2.push_back(value);  
}
```

```
int pop_front() {  
    if (q1.empty()) return -1;  
    else {  
        if (q1.front() == q2.front()) q2.pop_front();  
        int popvalue = q1.front();  
        q1.pop_front();  
        return popvalue;  
    }  
}
```

```
private:
```

```
deque<int> q1, q2;
```

```

};

/*
 * Your MaxQueue object will be instantiated and called as such:
 *
 * MaxQueue* obj = new MaxQueue();
 *
 * int param_1 = obj->max_value();
 *
 * obj->push_back(value);
 *
 * int param_3 = obj->pop_front();
 */


```

3、堆（剑指Offer）

剑指 Offer 40. 最小的k个数

题干：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。

分析：1、最直观的方法就是用sort进行排序，然后取k个最小值。nlogn

2、堆排序法，放进去再拿出来。但本题并没有要求全部排序，而是k个排序，故可仅创建一个k长度大小的堆：若k+1元素小于堆顶（大顶堆），则弹出堆顶并push该元素，以此类推直到结束。。。最后再将k个堆移到数组中。

3、与2类似，没必要对所有元素进行快速排序，只需对k个元素排序即可。

```

class Solution {

public:

    vector<int> getLeastNumbers(vector<int>& arr, int k) {
        •     priority_queue<int> q;
        •     vector<int> a(k);
        •     if(k == 0) return a;    // 需注意

        •     // 构建大顶堆
        •     for (int i = 0; i < k; i++)
        •         q.push(arr[i]);
    }
}

```

```

•     // 循环比较.若元素小于堆顶元素, 则替换

•     for (int j = k; j < arr.size(); j++)

•     {

•         if (q.top() > arr[j])

•             {

•                 q.pop();

•                 q.push(arr[j]);

•             }

•     }

•     // 将大顶堆中的元素放入数组中, 并返回值

•     for (int m = 0; m < k; m++)

•     {

•         a[m] = q.top();

•         q.pop();

•     }

•     return a;

}

};


```

4、排序

快速排序

指针交换法：

首先选定基准元素Pivot，并且设置两个指针left和right，指向数列的最左和最右两个元素：



接下来是**第一次循环**，从right指针开始，把指针所指向的元素和基准元素做比较。如果**大于等于**pivot，则指针向**左**移动；如果**小于**pivot，则right指针停止移动，切换到left指针。

在当前数列中， $1 < 4$ ，所以right直接停止移动，换到left指针，进行下一步行动。

轮到left指针行动，把指针所指向的元素和基准元素做比较。如果**小于等于**pivot，则指针向**右**移动；如果**大于**pivot，则left指针停止移动。

由于left一开始指向的是基准元素，判断肯定相等，所以left右移一位。



由于 $7 > 4$, left指针在元素7的位置停下。这时候,我们让left和right指向的元素进行交换。



接下来，我们进入**第二次循环**，重新切换到right向左移动。right先移动到8， $8 > 2$ ，继续左移。由于 $2 < 8$ ，停止在2的位置。



切换到left 6>4 停止在6的位置



当left和right指针重合之时，我们让pivot元素和left与right重合点的元素进行交换。此时数列左边的元素都小于4，数列右边的元素都大于4。这一轮交换终告结束。



```
#include "pch.h"  
#include <iostream>  
  
using namespace std;
```

```

void swap1(int& a, int& b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
void QSort(int *arr, int len, int *left,int *right)
{
    int pivot = *left;
    int *phead = left; // 因为要迭代,
    int *ptail = right;
    while (left != right)
    {
        if (*right >= pivot) right--;
        else
        {
            if (*left <= pivot) left++;
            else swap1(*left, *right);
        }
    }
    swap1(*left, *phead);
    if (phead == ptail) return; // 迭代退出
    QSort(arr, len, phead,left);
    QSort(arr, len, right+1,ptail);
}

void QuickSort(int *arr,int len)
{
    if (len < 1) return;
    QSort(arr, len, &arr[0],&arr[len-1]);
}

void Display(int *arr,int len)
{
    for (int i = 0; i < len; i ++ )
        cout << arr[i] << " ";
    cout << endl;
}
int main()
{
    int arr[] = { 1,3,2,8,6,0,33,64,22 };
    QuickSort(arr, sizeof(arr) / sizeof(arr[0]));
    Display(arr, sizeof(arr) / sizeof(arr[0]));

}

```

剑指 Offer 45. 把数组排成最小的数

题干：输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

分析：假设数字为5，38.，组合后385才为最小数字，，故采用直接拼接的方法组合。。转化为字符串

```

class Solution {

public:

    string minNumber(vector<int>& nums) {

```

```

    •     vector<string> str;
    •
    •     string ans;
    •
    •     // 转化为字符串数组
    •
    •     for (int i = 0; i < nums.size(); i++)
    •
    •     {
    •         str.push_back(to_string(nums[i]));
    •
    •     }
    •
    •     // 排序比较，利用sort函数
    •
    •     sort(str.begin(), str.end(), [](string a, string b) {return a + b < b + a;
    • });
    •
    •
    •     // 连接
    •
    •     for (int i = 0; i < str.size(); i++)
    •
    •         ans += str[i];
    •
    •     return ans;
    •
    • }
    •
};

}

```

5、位运算

剑指 Offer 15. 二进制中1的个数

题干：请实现一个函数，输入一个整数，输出该数二进制表示中 1 的个数。例如，把 9 表示成二进制是 1001，有 2 位是 1。因此，如果输入 9，则该函数输出 2。

```

class Solution {

public:

    int hammingWeight(uint32_t n) {
        •     int res = 0;
        •
        •     while (n)
    
```

```

•     {
•         if(n & 1)
•             res++;
•         n >>= 1;
•     }
•     return res;
}

```

```

class Solution {
public:
    int hammingWeight(uint32_t n) {
        int res = 0;
        while (n)
        {
            res++;
            n &= n - 1;
        }
        return res;
    }
};

```

剑指 Offer 39. 数组中出现次数超过一半的数字

题干：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。

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。

分析：1、对数组直接进行排序，处在中间位置的即为数字。（ $n \log n$ ）

2、摩尔投票法：核心理念为“**正负抵消**”，即假设众数为1，其余数为0，总体相加一定大于0。那么在实际编程时，用一个votes来计数。可先假设第一个数字为众数，若第二个数字不是该数，则抵消；若是该数，则加1。以此类推直至循环遍历完，（时间复杂度N，空间1）

```

class Solution {

public:

    int majorityElement(vector<int>& nums) {
        •     sort(nums.begin(),nums.end());
        •     return nums[nums.size()/2];
    }
};

```

```

class Solution {

public:

```

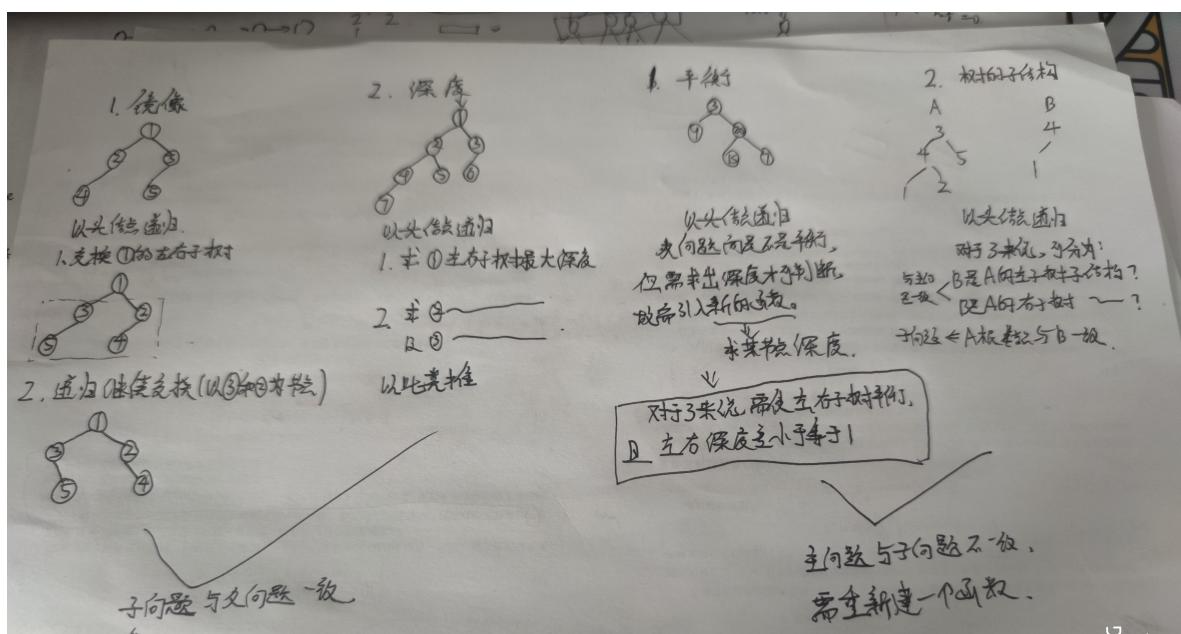
```

int majorityElement(vector<int>& nums) {
    •     int votes = 1;
    •     int x;
    •     x = nums[0];
    •     for (int i = 1; i < nums.size(); i++)
    •     {
    •         if (!votes) x = nums[i];
    •         if (x == nums[i]) votes++;
    •         else votes--;
    •     }
    •     return x;
}

```

5、树

1、对于普通树，可采用递归：按头节点进行递归（即参数为头节点）。对于27题：由于要求镜像，对于节点root来说，需要将root->left与root->right交换一下，并以此类推。对于55，要求求最大深度，对于root的最大深度为左子树与右子树的最大深度，故直接递归就可。



2、对于普通树，有时需要辅助函数。

3、普通树：递归+遍历。68题，需要采用后序遍历（先左、再右、后根），先寻找左右结点是否符合要求，再将结果返回根节点。

4、普通树，按照层来遍历。32题层序遍历，利用队列的先入先出特性；28题，判断是否对称，对称可通过一层一层比较，故需层递归。

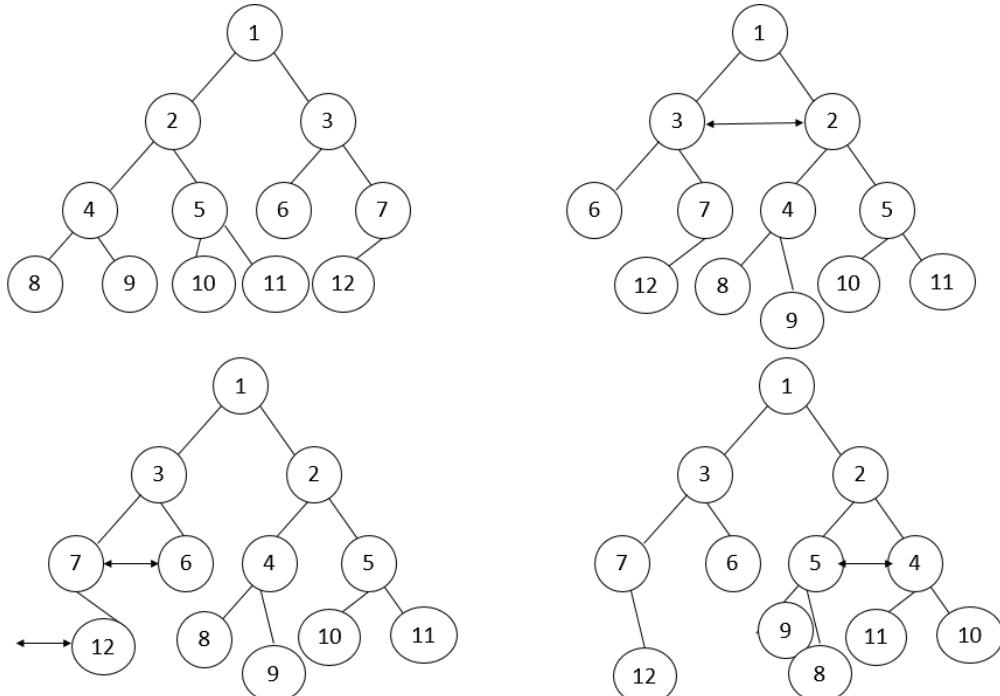
5、回溯：34题

6、二叉搜索树特点：中序遍历为从小到大。。。可见1008题。利用顺序构造单调栈进行计算。

剑指 Offer 27. 二叉树的镜像

题干：请完成一个函数，输入一个二叉树，该函数输出它的镜像。

分析：考虑到树是由结点构成，其优势在于便于断链和重构，所以要往链方面考虑。采用递归的方法，每次将左右节点交换，如下图所示。



```
// 链表的特性
class Solution {
public:

    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;

        TreeNode *tmp = root->left;
        root->left = root->right;
        root->right = tmp;
        mirrorTree(root->left);
        mirrorTree(root->right);
        return root;
    }
};

// 遍历+递归
class Solution {
public:

    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;
```

```

        TreeNode *tmp = root->left;
        root->left = mirrorTree(root->right);
        root->right = mirrorTree(tmp);
        return root;
    }
};

// 迭代
class Solution {
public:
    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;
        queue<TreeNode *> q;
        q.push(root);

        while(!q.empty())
        {
            TreeNode *cur = q.front();
            q.pop();

            if(cur->right) q.push(cur->right);
            if(cur->left) q.push(cur->left);

            TreeNode *tmp = cur->left;
            cur->left = cur->right;
            cur->right = tmp;
        }
        return root;
    }
};

```

剑指 Offer 55 - I. 二叉树的深度

题干：输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

分析：因为只给了根节点，只能通过遍历进行求深度

```

class Solution {

public:

    int maxDepth(TreeNode* root) {
        •     if(!root) return 0;

        •     return max(maxDepth(root->left),maxDepth(root->right))+1;
    }

};

// 双百

class Solution {
public:

    TreeNode* mirrorTree(TreeNode* root) {
        if(!root) return NULL;

        TreeNode *tmp = root->left;

```

```

        root->left = mirrorTree(root->right);
        root->right = mirrorTree(tmp);
        return root;
    }
}; // 后序遍历

// 迭代，利用层序遍历（可先看32）
class Solution {
public:
    int maxDepth(TreeNode* root) {
        queue<TreeNode*> q;
        if(root == NULL) return 0;

        q.push(root);
        int depth = 0;
        while(q.empty() == 0)
        {
            int len = q.size();
            for(int i=0;i<len;i++)
            {
                TreeNode *cur = q.front();
                q.pop();

                if(cur->left) q.push(cur->left);
                if(cur->right) q.push(cur->right);
            }
            depth++;
        }
        return depth;
    }
};

```

剑指 Offer 54. 二叉搜索树的第k大节点

题干：给定一棵二叉搜索树，请找出其中第k大的节点。

分析：二叉搜索树的特点为：若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；它的左、右子树也分别为二叉搜索树。

根据上述特点，可以按右、中、左的顺序遍历（反中序遍历）。

```

class Solution {

public:
    int kthLargest(TreeNode* root, int k) {
        •     if(!root) return 0;
        •
        •     kthLargest(root->right, k);
        •     if(result.size()<k)
        •         result.push(root->val);
        •     kthLargest(root->left, k);
    }
};

```

```

    }

    return result.top();

}

private:

    stack<int> result;

};

}

```

```

// 迭代，中序是左-根-右，递增。。那么找第k大，则右-根-左
class Solution {
public:
    int kthLargest(TreeNode* root, int k) {
        stack<TreeNode *> s;
        TreeNode *cur = root;
        int res;

        while(cur || !s.empty())
        {
            if(cur)
            {
                s.push(cur);
                cur = cur->right;
            }
            else
            {
                cur = s.top();
                s.pop();

                k--;
                res = cur->val;
                if(k == 0) return res;

                cur = cur->left;
            }
        }
        return res;
    }
};

```

剑指 Offer 32 - II. 从上到下打印二叉树 II

题干：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

分析：利用层序遍历，唯一麻烦的是需要分层打印。可多加一层循环

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 * };

```

```

*     TreeNode *right;
*     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        queue<TreeNode*> q;
        vector<vector<int>> res;
        if(root == NULL) return res;

        q.push(root);
        while(q.empty() == 0)
        {
            int len = q.size();
            vector<int> level;
            for(int i=0;i<len;i++)
            {
                TreeNode *cur = q.front();
                q.pop();

                if(cur->left) q.push(cur->left);
                if(cur->right) q.push(cur->right);
                level.push_back(cur->val);
            }
            res.push_back(level);
        }
        return res;
    }
};

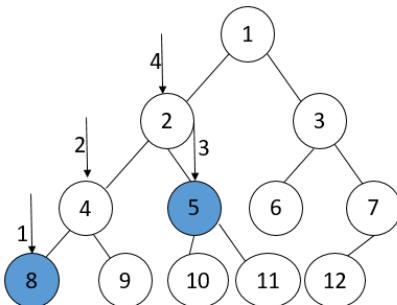
```

面试题68 - II. 二叉树的最近公共祖先

题干：给定一个二叉树，找到该树中两个指定节点的最近公共祖先。百度百科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）。”说明：

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

分析：公共祖先共有三种情况：1、p为两者的公共祖先（即q为p的子树）；2、q为二者公共祖先；3、pq不为祖先。若为1，则左或右return值为p，另一return值为空；2与1类似。若为3，则公共祖先处的两返回值分别为p、q。下图进一步分析

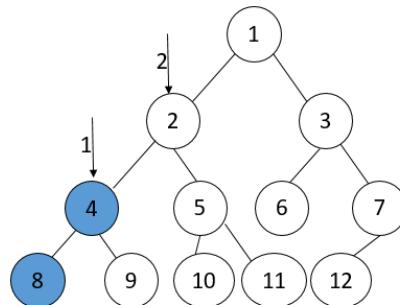


后序：

4->left == 8, 所以返回8。4->right != 5, return NULL
4接收到8和NULL之后，继续向上返回8
等等

2->right == 5, return 5。

2既有左返回值，也有右返回值，故return2
再往上，1->right = NULL（遍历后），故最终return2



后序：

2->left == 4, 所以返回4。2->right != 8, return NULL
2接收到4和NULL之后，继续向上返回4
1->right == NULL(遍历后)，故最终return 4；

注：图为方便解释，将值用了进去，实际编程时，返回的是指向该值的指针。

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        TreeNode *left, *right;
        // 1、既可表示空、也可表示root为NULL
        if (!root) return NULL;
        // 1、若root等于p或q
        if (root == p || root == q) return root;

        left = lowestCommonAncestor(root->left, p, q);
        right = lowestCommonAncestor(root->right, p, q);

        // 2、若左右均有
        if (left && right) return root;
        // 2、若左有或右有
        if (left && !right) return left;
        if (!left && right) return right;
        // 2、先假设左右均为NULL，则
        return NULL;
    }
};

// 迭代
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        while(root)
        {
            if(root->val >p->val && root->val > q->val)
                root = root->left;
            else if(root->val <p->val && root->val < q->val)
                root = root->right;
            else
                break;
        }
        return root;
    }
};
```

剑指 Offer 55 - II. 平衡二叉树

题干：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。

```
class Solution {

public:

    int GetDepth(TreeNode *root)

    {
        •     if(!root) return 0;

        •     return max(GetDepth(root->left),GetDepth(root->right))+1;
```

```

    }

    bool isBalanced(TreeNode* root) {
        if(!root) return 1; // 如果结点为空，则说明该节点的数为平衡树
        return abs(GetDepth(root->left)-GetDepth(root->right))<2 && isBalanced(root->left) && isBalanced(root->right);
    }
};


```

```

class Solution {

public:
    int GetDepth(TreeNode *root)
    {
        return !root? 0:max(GetDepth(root->left),GetDepth(root->right))+1;
    }

    bool isBalanced(TreeNode* root) {
        // 如果结点为空，则说明该节点的数为平衡树
        return !root? 1:abs(GetDepth(root->left)-GetDepth(root->right))<2 && isBalanced(root->left) && isBalanced(root->right);
    }
};

// 错
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        int min_level = -10000;
        queue<TreeNode *> q;
        if(root) q.push(root);
        TreeNode *cur = root;
        bool flag = 1; // 判断是否第一次使min_level为0
        while(!q.empty())
        {
            int len = q.size();
            min_level++;
            cout<<min_level;
            for(int i=0;i<len;i++)
            {
                cur = q.front();
                q.pop();
                if(flag && (!cur->left || !cur->right))
                {
                    min_level = 0;
                    flag = 0;
                }
            }
        }
    }
};


```

```

        if(cur->left) q.push(cur->left);
        if(cur->right) q.push(cur->right);
    }
}

return min_level > 1?0:1;
};

}

```

剑指 Offer 28. 对称的二叉树

题干：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。

分析：1、镜像对称的中序排列也是对称的。所以可先中序排列，再双指针比较。（错，中序对称的不一定为镜像树，例如：[1,2,2,2,null,2]）

代码如下：

```

class Solution {
public:
    void dfs(TreeNode *root)
    {
        if(!root) return ;

        dfs(root->left);
        nums.push_back(root->val);
        dfs(root->right);
    }

    bool isSymmetric(TreeNode* root) {
        dfs(root);

        int left = 0;
        int right = nums.size()-1;
        while(left<right)
        {
            if(nums[left] != nums[right]) return 0;
            left++;
            right--;
        }
        return 1;
    }

private:
    vector<int> nums;
};

// 层序递归

/**
 * 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) {
        queue<TreeNode *> q;
        if(!root) return true;

        q.push(root->left);
        q.push(root->right);

        while(!q.empty())
        {
            TreeNode * A = q.front();
            q.pop();
            TreeNode * B = q.front();
            q.pop();

            if(A==NULL && B == NULL) continue;
            if(A==NULL || B==NULL) return 0;

            if(A->val != B->val) return 0;

            q.push(A->left);
            q.push(B->right);
            q.push(A->right);
            q.push(B->left);
        }
        return 1;
    }
};

```

2、按层进行递归。

```

class Solution {

public:
    bool helper(TreeNode *Left, TreeNode *Right)

    {
        • // 按层迭代

        • // 迭代结束条件

        • if (!Left && !Right) return 1;

        • if ((!Left && Right) || (Left && !Right)) return 0;

        • if (Left->val != Right->val) return 0;

        • // 进入下一层判断
    }
};

```

```

    •     return helper(Left->left, Right->right) && helper(Left->right, Right->left);

}

bool isSymmetric(TreeNode* root) {
    •     if (!root) return 1;
    •     return helper(root->left, root->right);
}

};

```

剑指 Offer 07. 重建二叉树

题干：输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。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，给出

```

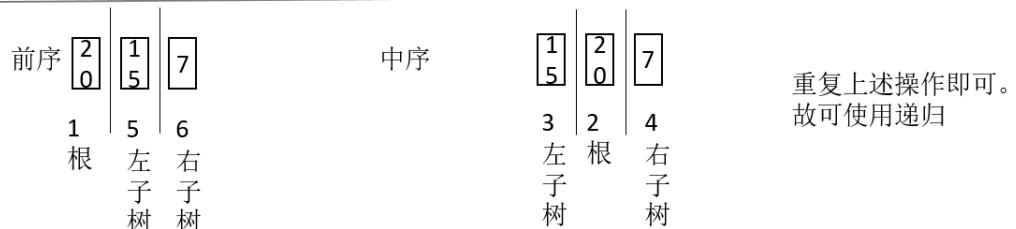
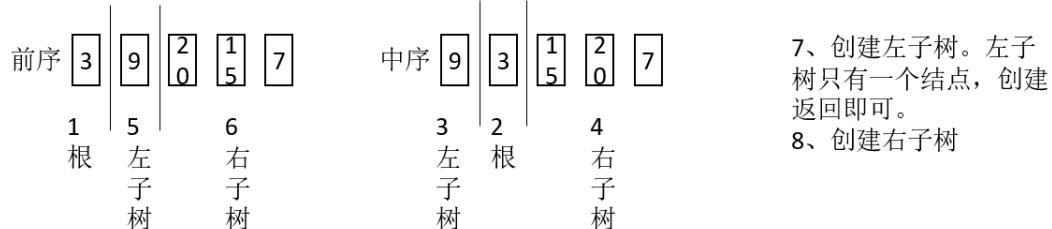
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]

```

分析：

前序遍历 [3,9,20,15,7] 中序遍历 [9,3,15,20,7]

1、根据前向遍历，知根节点为3，2、在中序遍历中遍历找到3的索引，
3、之前的均为左子树，4、后面的为右子树。5、6、计算中序左子树长度，便可将前序遍历也分开。



```

/**
 * 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 *helper(vector<int>& preorder, int pre_start, int pre_end, vector<int>&
inorder, int in_start, int in_end, unordered_map<int, int>&hm)
{
    // 递归终止条件
    if (pre_start > pre_end) return NULL;

    TreeNode * root = new TreeNode(preorder[pre_start]);

    int head_index = hm[preorder[pre_start]];
    int left_nodes = head_index - in_start;
    root->left = helper(preorder, pre_start+1, pre_start+left_nodes, inorder,
in_start, head_index-1, hm);
    root->right = helper(preorder, pre_start+left_nodes+1, pre_end, inorder,
head_index+1, in_end, hm);

    return root;
}
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {

    unordered_map<int, int> hm;
    // 建立哈希表，方便查询
    for (int i = 0; i < inorder.size(); i++)
    {
        hm[inorder[i]] = i;
    }
    return helper(preorder, 0, preorder.size()-1, inorder, 0,
inorder.size()-1, hm);
}

```

剑指 Offer 32 - I. 从上到下打印二叉树

题干：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

分析：层序排列，比32-II简单

```

class Solution {
public:
    vector<int> levelOrder(TreeNode* root) {
        queue<TreeNode *> q;
        q.push(root);
        vector<int> res;
        if(root == NULL) return res;

        while(!q.empty())
        {
            TreeNode *cur = q.front();
            q.pop();

            if(cur->left) q.push(cur->left);
            if(cur->right) q.push(cur->right);

            res.push_back(cur->val);
        }
        return res;
    }
};

```

剑指 Offer 32 - III. 从上到下打印二叉树 III

题干：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。

分析：在32-ii的基础上修改即可。。之前是利用了先进先出队列，本题可用双端队列，奇数偶数层分开，若从后面取，则push在前面，，从前面取，push在后面。

```
/*
 * 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<vector<int>> levelOrder(TreeNode* root) {
        • // 队列
        • deque<TreeNode *> d;
        • // 存放数据的容器
        • vector<vector<int> > result;
        • bool flag = true;
        • //
        • d.push_back(root);
        • if(!root) return result;

        • while (!d.empty())
        • {
        •     vector<int> level;
        •     int times = d.size();
        •     for (int i = 0; i < times; i++)
        •     {
        •         TreeNode * current;
```

```

•     if (flag)

•     {

•         current = d.front();

•         d.pop_front();

•         if (current->left) d.push_back(current->left);

•         if (current->right) d.push_back(current->right);

•     }

•     else

•     {

•         current = d.back();

•         d.pop_back();

•         if (current->right) d.push_front(current->right);

•         if (current->left) d.push_front(current->left);

•     }

•     level.push_back(current->val);

• }

• result.push_back(level);

• flag = !flag;

• }

• return result;

}

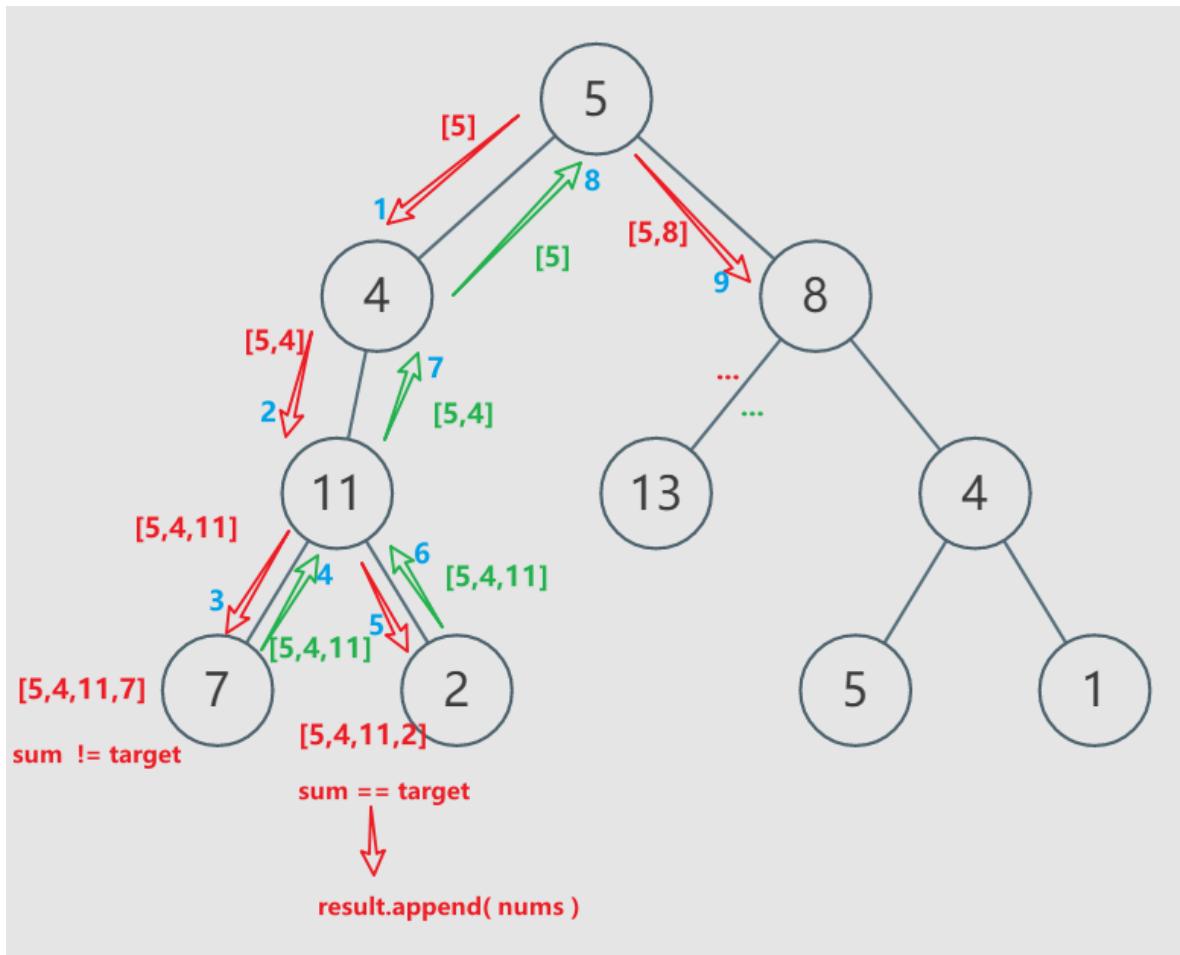
};


```

剑指 Offer 34. 二叉树中和为某一值的路径

题干：输入一棵二叉树和一个整数，打印出二叉树中节点值的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。

分析：看到题首先想到的是递归，即每次输入节点和值，看能不能走到叶节点。但本文需要返回走过的路径，即可以把题目看成：1、先找到所有从根节点到叶节点的路径 2、若该路径符合要求，则push一下。3、由于一般用递归。根-左-右，故为前序。



```

class Solution {
public:
    vector<vector<int>> result; // 存放符合条件的路径
    vector<int> path; // 存放每次走的路径

    void dfs(TreeNode* root, int sum)
    {
        // 递归出口
        if (!root) return;

        // 前序，与根有关的操作
        path.push_back(root->val);
        sum = sum - root->val;
        if (!sum && !root->left && !root->right) result.push_back(path);

        dfs(root->left, sum );
        dfs(root->right, sum );

        // 回溯，每返回上一层，需将path pop一次
        path.pop_back();
    }
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        dfs(root, sum);
        return result;
    }
};

```

剑指 Offer 26. 树的子结构

题干：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。

与剑指 Offer 55 - II. 平衡二叉树类似

若B是A的子结构，则有：1、A、B结点一致 2、B是A的左子树的子结构 3、B是A的右子树的子结构。

```
class Solution {  
  
public:  
  
    bool helper(TreeNode* A, TreeNode* B)  
  
    {  
  
        • if(B == NULL) return true;  
  
        • if(A == NULL) return false; // A无, B有, 则B一定不是A的子结构  
  
        • if(A->val != B->val) return false;  
  
        • return helper(A->left,B->left) && helper(A->right,B->right);  
    }  
  
    bool isSubStructure(TreeNode* A, TreeNode* B) {  
  
        • if(A == NULL || B == NULL) return false;  
  
        • return helper(A,B) || isSubStructure(A->left, B) || isSubStructure(A->right, B);  
    }  
};
```

6、数组

若是排序数组，则首先考虑二分法。普通数组可考虑哈希表。

剑指 Offer 04. 二维数组中的查找

题干：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

分析：针对下面对矩阵，可以想办法减小搜索量。。如果搜索20，则可先让20和对角线比较，大于17，小于30，则可将范围缩小在倒数第一行和倒数第一列。以此递归。

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]  
]
```

该法不易实现，较为麻烦。

从二维数组的右上角开始查找。如果当前元素等于目标值，则返回 true。如果当前元素大于目标值，则移到左边一列。如果当前元素小于目标值，则移到下边一行。

```
class Solution {  
public:  
    bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {  
        if(matrix.empty() || matrix[0].empty()) { // 注：1、判断二维容器时hi，应同时判断两个  
        }  
        return false;  
    }  
    int line = matrix[0].size();  
    int row = matrix.size();  
    int l_c = line-1;  
    int r_c = 0;  
    while (l_c >= 0 && r_c < row)  
    {  
        if (matrix[r_c][l_c] > target) l_c--;  
        else if (matrix[r_c][l_c] < target) r_c++;  
        else return 1;  
    }  
    return 0;  
};
```

剑指 Offer 53 - II. 0 ~ n-1中缺失的数字

题干：一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0 ~ n-1之内。在范围0 ~ n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

分析：首先想到的是快慢指针，一个指针在中间，一个指针在开头，然后移动判断。但二分法效率更高。

```
class Solution {  
public:  
    int missingNumber(vector<int>& nums) {  
        int start = 0;  
        int end = nums.size() - 1;  
        while (start <= end) // 注意：1、需要等于  
        {  
            int middle =(start+end) / 2;  
            if (nums[middle] != middle) end = middle - 1;  
            else start = middle + 1;  
        }  
        return start; //注意：2、由于索引的特殊情况，可等于这个  
    };
```

剑指 Offer 29. 顺时针打印矩阵

题干：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。

分析：设置上下左右四个边界，先将一圈存储进去，一圈存储完毕后，将边界缩小，重新顺时针循环。

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> result;
        if(matrix.empty() || matrix[0].empty()) return result;
        int count = 0;
        int left = 0;
        int top = 0;
        int bottom = matrix.size();
        int right = matrix[0].size();

        int sum = matrix[0].size() * matrix.size();
        while (count < sum)
        {
            for (int i = left; i < right && count < sum; i++)
            {
                result.push_back(matrix[top][i]);
                count++;
            }
            for (int j = top+1; j < bottom && count < sum; j++)
            {
                result.push_back(matrix[j][right-1]);
                count++;
            }
            for (int i = right - 2; i >= left && count < sum; i--)
            {
                result.push_back(matrix[bottom-1][i]);
                count++;
            }
            for (int j = bottom - 2; j > top && count < sum; j--)
            {
                result.push_back(matrix[j][left]);
                count++;
            }
            left++;
            top++;
            right--;
            bottom--;
        }
        return result;
    }
};
```

剑指 Offer 53 - I. 在排序数组中查找数字 I

题干：统计一个数字在排序数组中出现的次数。

分析：利用二分查找找到目标数字，然后向两边延伸。

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int start = 0;
        int end = nums.size()-1;
        int count = 0;
```

```

•     int change;
•     int middle;
•     bool flag = 0;
•     if(nums.empty()) return 0;
•     while(start <= end)
•     {
•         middle = (start+end)/2;
•         if(nums[middle] < target) start = middle +1;
•         else if(nums[middle] >target) end = middle-1;
•         else
•         {
•             flag = 1;
•             break;
•         }
•     }
•     if(flag)
•     {
•         change = middle;
•         while(change>=0 && change <nums.size() &&nums[change] == target)
•         {
•             change++;
•             count++;
•         }
•         change = middle-1;
•         while(change>=0 && change <nums.size() &&nums[change] == target)
•         {
•             change--;
•             count++;
•         }
•     }
// another way
if(!flag) return 0;
int left,right;

left=middle-1;
right = middle;
while(right<nums.size() && nums[right] == target) right++;
while(left >= 0 && nums[left] == target) left--;
return right - left - 1;
•     return count;

}
};

#

```

剑指 Offer 03. 数组中重复的数字

题干：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。

分析：利用哈希表，键值为数组元素，value为次数

```

class Solution {
public:
    int findRepeatNumber(vector<int>& nums) {
        unordered_map<int, int> hm;
        for (auto num : nums)
        {
            if (hm[num]++) return num;
        }
        return -1;
    }
};

```

7、哈希

剑指 Offer 50. 第一个只出现一次的字符

题干：在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

分析：1、对于无序字符串，要找到次数为1的字符，，题目与3差不多，均需采用哈希表，时间复杂度为O(N)，即一定需要全部遍历完所有字符，才能知道谁出现次数小于2。。

2、优化的关键问题在于遍历完一次后，如何快速找到第一个字符，，想到可以用变量，，变量过程中若次数 == 1，则变量赋值。。。这样的问题是，变量保存的是最后一个出现为一次的字符。（错，若是aabc，该思路不对）

3、有序哈希表： Map<Character, Boolean> dic = new LinkedHashMap<>();

```

class Solution {
public:
    // map 与 unorderedmap
    // map:
    /*优点:
     * 有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作
     * 红黑树，内部实现一个红黑树使得map的很多操作在lg n的时间复杂度下就可以实现，因此效率非常的高
     * 缺点： 空间占用率高，因为map内部实现了红黑树，虽然提高了运行效率，但是因为每一个节点都需要额外保存父节点、孩子节点和红 / 黑性质，使得每一个节点都占用大量的空间
     * 适用处：对于那些有顺序要求的问题，用map会更高效一些
    */
    // unordered_map:
    /*优点： 因为内部实现了哈希表，因此其查找速度非常的快
     * 缺点： 哈希表的建立比较耗费时间
     * 适用处：对于查找问题，unordered_map会更加高效一些，因此遇到查找问题，常会考虑一下用unordered_map
     * 总结：
     */
    char firstUniqChar(string s) {
        map<char, int> m;
        for (auto s_sub : s)
        {
            m[s_sub]++;
        }
        for (auto s_su : s)
        {
            if (m[s_su] == 1) return s_su;
        }
    }
};

```

```

    }
    return '';
}
};

```

剑指 Offer 48. 最长不含重复字符的子字符串

题干：请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。

分析：动态规划+hashmap。。。假设第*i*个字符最长子字符串为f(*i*)，若f(*i*)不在f(*i*-1)中，则f(*i*) = f(*i*-1)+1.else f(*i*)=1;分析不完整，，当为ABCAD时，按上述分析，最长为3（ABC），实际上位4（BCAD）。故不能直接判断f(*i*)在不在f(*i*-1)中

```

class Solution {
public:
    // 利用哈希表存储，作用为快速查找。
    // 利用动态规划求得第i个字符所在的最大字符串
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> hm;
        vector<int> result(s.size() + 1);
        result[0] = 0;
        if(s.size() == 0) return 0;
        for (int i = 1; i <= s.size(); i++)
        {
            if (hm.find(s[i - 1]) != hm.end()) // 找到相同元素
            {
                hm.clear();
                hm[s[i - 1]] = i;
                result[i] = 1;
            }
            else
            {
                hm[s[i - 1]] = i;
                result[i] = result[i - 1] + 1;
            }
        }
        return *max_element(result.begin(), result.end());
    }
};

```

进一步分析：考虑到上述问题，若有重复，则需找到重复值的索引*i*。比较当前最长长度len与索引差(j-i)，若j-i>len,说明当前长度中无重合数字，加1即可。。。若j-i<len，则需要将当前长度更改为j-i。

```

class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> hm;
        int res = 0; // 存储最大长度
        int cur = 0; // 存储当前最大长度
        for (int i = 0; i < s.size(); i++)
        {
            // 有重复
            if (hm.find(s[i]) != hm.end() && i - hm[s[i]] > cur ? cur++ : cur = i - hm[s[i]]);
            else cur++;
            hm[s[i]] = i;
            res = max(res, cur);
        }
    }
};

```

```
    }
    return res;
}
};

#
```

8、链表

剑指 Offer 22. 链表中倒数第k个节点

题干：输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。例如，一个链表有6个节点，从头节点开始，它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个节点是值为4的节点。

```
class Solution {
public:
    ListNode* getKthFromEnd(ListNode* head, int k) {
        if (!head) return NULL;
        if (!head->next && k > 1) return NULL;
        if (!head->next && k == 1) return head;
        ListNode *pslow = head;
        ListNode *pfast = head;
        for (int i = 1; i < k; i++)
        {
            if (!pfast) return NULL; // 防止k过大
            pfast = pfast->next;
        }
        while (pfast->next)
        {
            pslow = pslow->next;
            pfast = pfast->next;
        }
        return pslow;
    }
}
```

剑指 Offer 06. 从尾到头打印链表

题干：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）。

分析：考虑到要从后往前打印，且链表不能通过地址寻找，故利用栈的先入后出原则。

```
class Solution {
public:
    vector<int> reversePrint(ListNode* head) {
        stack<int> s;
        vector<int> result;
        if(!head) return result;
        // 入栈
        s.push(head->val);
        while (head->next)
        {
            head = head->next;
            s.push(head->val);
        }
        while (!s.empty())
        {
            result.push_back(s.top());
            s.pop();
        }
        return result;
    }
}
```

```

    }
    // 出栈放在数组。
    while (!s.empty())
    {
        result.push_back(s.top());
        s.pop();
    }
    return result;
}
};

```

剑指 Offer 24. 反转链表

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。

```

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

```

剑指 Offer 52. 两个链表的第一个公共节点

题干：输入两个链表，找出它们的第一个公共节点。

分析：1、利用哈希表，表的key为结点地址，value为结点值，当find到一样的值后，返回value

```

class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        if (!headA || !headB) return NULL;
        unordered_map<ListNode*, int> hm;
        while (headA)
        {
            hm[headA] = headA->val;
            headA = headA->next;
        }
        while (headB)
        {
            if (hm.find(headB) != hm.end()) return headB;
            else headB = headB->next; // HEADb无需存入
        }
        return NULL;
    }
};

```

2、走同样长的路，看能不能相遇。我们使用两个指针 node1 , node2 分别指向两个链表 headA , headB 的头结点，然后同时分别逐结点遍历，当 node1 到达链表 headA 的末尾时，重新定位到链表 headB 的头结点；当 node2 到达链表 headB 的末尾时，重新定位到链表 headA 的头结点。这样，当它们相遇时，所指向的结点就是第一个公共结点。

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode *nodeA = headA;
        ListNode *nodeB = headB;

        while (nodeA != nodeB)
        {
            nodeA = !nodeA ? headB : nodeA->next; // 注: 不能有! nodeA->next, , node为空时, 没有next
            nodeB = !nodeB ? headA : nodeB->next; // 退出循环的条件为, 两者都为NULL。或者交点。若用 // !nodeA->next 则假设无交点, 则永远不会相等。
        }
        return nodeA;
    }
};
```

剑指 Offer 18. 删除链表的节点

题干：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。

分析：

```
class Solution {
public:
    ListNode* deleteNode(ListNode* head, int val) {
        if (!head) return NULL; // 若空, 返回NULL
        ListNode *pfirst = new ListNode; // 创建伪头节点,
        pfirst->next = head;
        ListNode *pchange = head;
        ListNode *pfront = pfirst;
        while (pchange)
        {
            if (pchange->val != val) pchange = pchange->next;
            else
            {
                pfront->next = pchange->next;
                return pfirst->next;
            }
            pfront = pfront->next; // 防止删除head
        }
        return NULL;
    }
};
```

/*
1、头节点为空，直接返回null
2、头节点不为空，但删除的为第一个节点，则需返回第二个（加个判断，，或建立新头节点，直接进行3操作）
3、删除的不为第一个结点，则正常删除操作
*/

```

class Solution {
public:
    ListNode* deleteNode(ListNode* head, int val) {
        ListNode * pcur = head;
        if(!head) return NULL;
        if(head->val == val) return head->next;

        while(pcur->next)
        {
            if(pcur->next->val == val)
            {
                pcur->next = pcur->next->next;
                return head;
            }
            pcur = pcur->next;
        }
        return head;
    }
};

```

剑指 Offer 35. 复杂链表的复制

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。

```

/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};

*/
/*
1、在现有的链表上复制节点,
2、对random指针进行处理
3、进行分离
*/
class Solution {
public:
    Node* copyRandomList(Node* head) {
        Node *cur = head;
        if(!cur) return NULL;
        // 1、复制结点
        while(cur)
        {
            Node *newnode = new Node(cur->val);
            newnode->next = cur->next;
            cur->next = newnode;
            cur = newnode->next;
        }

        // 2、处理random指针

```

```

Node *orig = head;
Node *copy = head->next;
while(orig)
{
    copy = orig->next;
    if(orig->random)
        copy->random = orig->random->next;
    orig = copy->next;
}

// 3、断开
Node *copyhead = head->next;
Node *pcur = copyhead;
Node *pcur1 = head;
while(pcur->next)
{
    pcur1->next = pcur->next;
    pcur1 = pcur1->next;
    pcur->next = pcur1->next;
    pcur = pcur->next;
}
pcur1->next = NULL;
pcur->next = NULL;
return copyhead;
}
};

```

剑指 Offer 49. 丑数

题干：我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。

分

析

设已知长度为 n 的丑数序列 x_1, x_2, \dots, x_n ，求第 $n+1$ 个丑数 x_{n+1} 。根据递推性质，丑数 x_{n+1} 只可能是以下三种情况其中之一（索引 a, b, c 为未知数）：

$$x_{n+1} = \begin{cases} x_a \times 2 & , a \in [1, n] \\ x_b \times 3 & , b \in [1, n] \\ x_c \times 5 & , c \in [1, n] \end{cases}$$

a, b, c 分别为距离最近的满足要求的丑数，因此关键在于如何找到 x_a, x_b, x_c 。由于数组是递增的，所以可知，如果 $2=2*1$ ，下次用到 2 乘的数一定要是 2 乘 2，即 4 。3 乘 5 乘也类似，与他们相乘的丑数也是会递增的。因此，

```

class Solution {
public:
    int nthUglyNumber(int n) {
        vector<int> dp(n+1, 0);
        dp[0] = 1;
        int a=0;int b=0;int c = 0;
        for(int i =1;i< n;i++)
        {
            dp[i] = min(min(dp[a]*2,dp[b]*3),dp[c]*5);
            if(dp[i] == dp[a]*2) a++;
            if(dp[i] == dp[b]*3) b++;
            if(dp[i] == dp[c]*5) c++;
        }
    }
};

```

```

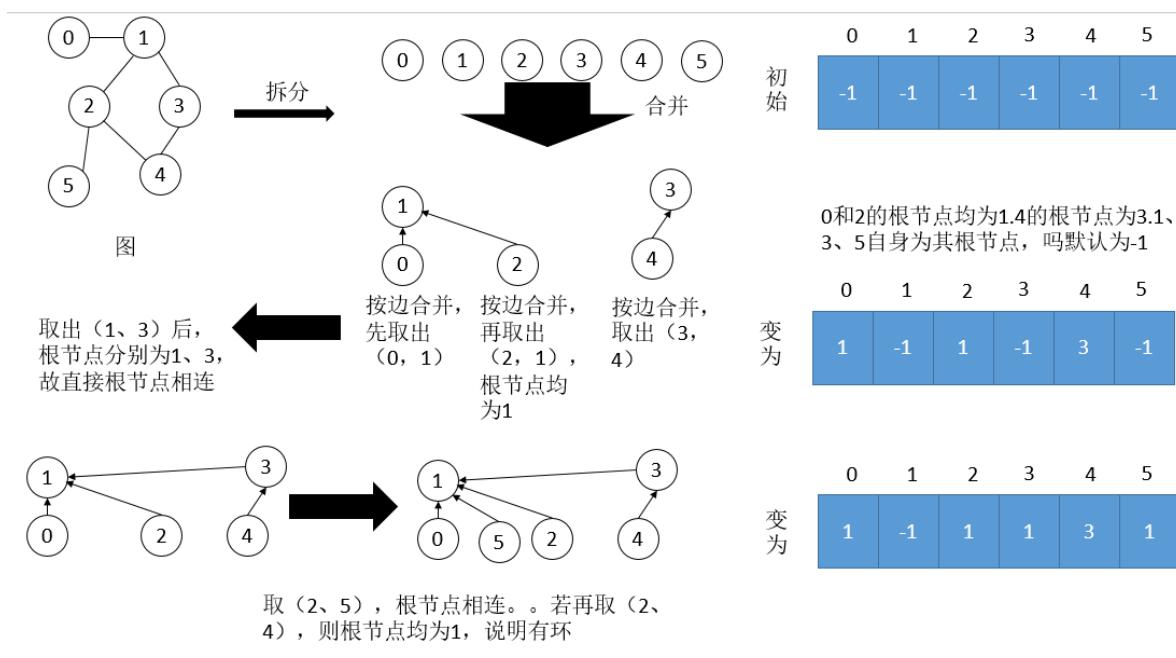
    }
    return dp[n-1];
}
};

```

并查集

并查集最大的作用就是检查图上有没有环。简单的说，并查集就是先将图拆分，再查找集合的交点。如下图所示：

按相连的边依次取出分开的结点，判断结点的根节点是否一致，若一致则说明有环；不一致则将根节点相连；按此步骤不断循环。<https://www.bilibili.com/video/BV13t411v7Fs?from=search&seid=314877532056002176>



未分组

剑指 Offer 16. 数值的整数次方

题干：实现函数double Power(double base, int exponent)，求base的exponent次方。不得使用库函数，同时不需要考虑大数问题。

分析：首先想到的是使用递归进行计算。但是，指数幂容易产生指数爆炸，需要一种方法避免指数爆炸。本题中采用的是快速幂算法。详细讲解可见，非常易懂https://blog.csdn.net/qq_19782019/article/details/85621386

可一步一步分析：产生指数爆炸的主要问题在于指数幂过大，因此需要想办法减小幂指数。以 2^{10} 为例， $2^{10} = (2^2)^5$ ，这样就将原来的10变为了5（可当作二分）；以此类推， $4^5 = 4 \times 4^4 = 4 \times 16^2 = 4 \times 32^1 = 4 \times 32$ 。上述操作可以分为：1、当指数幂为偶数时，底数平方，指数减半；2、指数幂为奇数时，分离出一个底数之后指数变为偶数，继续1的操作。。到最后可以总结出规律，即 2^{10} 实际上为指数为奇数的底数乘积（ 4×32 ；二者指数均为1）。那么代码可写为：

```
class Solution {

public:

    double myPow(double x, int n) {
        •     double result = 1; // 用来存放指数为奇数的底数乘积
        •     long b1 = n; // 主要是解决 n = -2147483648时, n = -n超出界限
        •     if(n < 0) // n<0时, 需修改一下才能适用于下面代码
        •     {
        •         x = 1/x;
        •         b1 = -b1;
        •     }
        •     while(b1 > 0)
        •     {
        •         if(b1 % 2) // 指数为奇数时, 分离出一个底数, 再将底数平方, 指数减半
        •         {
        •             result = result * x;
        •         }
        •         x = x*x;
        •         b1 = b1/2;
        •     }
        •     return result;
    }
};
```

可进一步修改：

- 1、奇偶数判断：原先用 $b1 \% 2$,现用 $b1 \& 1$ （结果为0，偶数；结果为1，奇数）
- 2、除2操作， $b1>>1$ ；左移一位相当于除以2.

```
class Solution {

public:

    double myPow(double x, int n) {
```

```

•     double result = 1; // 用来存放指数为奇数的底数乘积

•     long b1 = n; // 主要是解决 n = -2147483648时，n = -n超出界限

•     if(n < 0) // n<0时，需修改一下才能适用于下面代码

•     {

•         x = 1/x;

•         b1 = -b1;

•     }

•     while(b1 > 0)

•     {

•         if(b1 & 1) // 指数为奇数时，分离出一个底数，再将底数平方，指数减半

•         {

•             result = result * x;

•         }

•         x = x*x;

•         b1 >>= 1;

•     }

•     return result;

}

};


```

剑指 Offer 59 - I. 滑动窗口的最大值

题干：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。

分析：此题和59题 队列最大值类似，都需要通过队列先入先出的原则进行求解。每次需要在窗口中找到最大值，故可以考虑将最大值放入队列，但若窗口最大值出去，则无法判断剩下当中谁是次大值，需要重新计算最大值。。。因此参考59的解题方式，在窗口移动时，比较即将进入的值与已有值得大小，若队列中有值小于该值，则pop。。这一步可以使队列中处于不递增的状态，能保证队列开头为最大值。。

```

class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> result; // 存放窗口最大值
        deque<int> d; // 双端队列，便于队尾pop
        if(nums.empty()) return result; // 注：2、为判断空数组，出错
        // 构建第一个窗口
        d.push_back(nums[0]);
        for (int i = 1; i < k; i++)
        {

```

```

•     while (!d.empty() && nums[i] > d.back()) d.pop_back(); // 注: 3、pop前未判断
对列是否为空
•     d.push_back(nums[i]);
•   }
•   result.push_back(d.front()); // 第一个窗口的最大值
•   // 开始循环
•   for (int j = k; j < nums.size(); j++)
•   {
•     while (!d.empty() && nums[j] > d.back()) d.pop_back(); // 若新进元素大于之前队列
元素，则pop
•     if (!d.empty() && nums[j - k] == d.front()) d.pop_front(); // 若准备出去的元素为
队列最大元素，则pop
•     d.push_back(nums[j]); // 注: 1、忘记push，结果出错
•     result.push_back(d.front());
•   }
•   return result;
}
};


```

剑指 Offer 58 - I. 翻转单词顺序

题干：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串 "I am a student."，则输出 "student. a am I"。

分析：1、利用栈辅助逆序：出现空格判断第一个栈是否为空，若不为空，则将栈1送入栈2，并在最后入一空格。

```

class Solution {
public:
    void EraseSpace(string &s)
    {
        //ch可换成其他字符
        const char ch = ' ';
        s.erase(s.find_last_not_of(" ") + 1);
        s.erase(0, s.find_first_not_of(" "));
    }
    string reverseWords(string s) {
        string result;
        bool flag = 1;
        EraseSpace(s);
        // 若不为空格，则入栈1，遇到空格，判断栈1是否为空，不空则将栈1数据入栈2
        for (auto c : s)
        {
            if (c != ' ')
            {
                s1.push(c);
            }
            else
            {
                while (!s1.empty())
                {
                    s2.push(s1.top());
                    s1.pop();
                    if (s1.empty()) s2.push(' ');
                }
            }
        }
        while (!s1.empty())
        {
            s2.push(s1.top());
        }
    }
};


```

```

        s1.pop();
    }
    while (!s2.empty())
    {
        result.push_back(s2.top());
        s2.pop();
    }
    return result;
}
private:
stack<char> s1;
stack<char> s2;
};

```

2、双指针：从后往前，当遇到空格，则将两个指针之间的内容

```

class Solution {
public:
    void EraseSpace(string &s)
    {
        s.erase(s.find_last_not_of(" ") + 1); // 反向找到第一个不为空的，并删除
        s.erase(0, s.find_first_not_of(" ")); // 正向找到第一个不为空的并删除
    }
    string reverseWords(string s) {

        if (s.empty()) return "";
        // 1、删除头尾部空格
        EraseSpace(s);

        // 字符串区间
        int pfront = s.size() - 1;
        int pback = s.size() - 1;

        // 2、找单词
        string res;
        while(pfront >= 0)
        {
            if (s[pfront] != ' ') pfront--;
            else
            {
                res.append(s, pfront + 1, pback - pfront); // 将单词复制给结果字符串
                res += ' '; // 加空格
                while (s[pfront] == ' ') pfront--; // 类似于删除尾部空格的操作
                pback = pfront;
            }
        }
        // 3、由于最后一个不为空格，故最后一次的数据不会存储，需在下面代码存储
        res.append(s, pfront+1, pback - pfront);
        return res;
    }
};

```

剑指 Offer 25. 合并两个排序的链表

题干：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。

分析：两种方法进行合并

1、在原链表l1基础上进行插入操作。若l1先为空，则next指向剩余的l2即可，若l2先为空，则return (该方法比较繁琐)

2、新建链表，每次比较插入。

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

class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode *pfirst = new ListNode(0);
        ListNode *cur = pfirst;
        while (l1 && l2) {
            if (l1->val >= l2->val)
            {
                cur->next = l2;
                l2 = l2->next;
            }
            else
            {
                cur->next = l1;
                l1 = l1->next;
            }
            cur = cur->next;
        }
        if (l1 && !l2) cur->next = l1;
        if (l2 && !l1) cur->next = l2;
        return pfirst->next;
    }
};
```

二分法变式：

1、给定一升序数组，找到第一次出现的数字

```
class Solution {
public:
    vector<int> BinarySearch(vector<int> &nums, int target)
    {
        int c_left = 0;
        int c_right = nums.size() - 1;
        vector<int> res;

        // 1、找到重复数字第一次出现的下标(即重复数字的最左边)
        while (c_left < c_right) // 循环结束条件(若存在，则在相等时退出)
        {
            int middle = c_left + c_right >> 1; // 取中间
            // 大于等于target时，有可能middle处的值就为target，故right = middle，而不能-1
            // 故每次只修改左边界就行
            if (nums[middle] >= target) c_right = middle;
```

```

        else c_left = middle + 1;
    }
    if (nums[c_left] == target) res.push_back(c_left);

    // 注意截断的问题
    // 2、找到重复数字最后一次出现的下标(即重复数字的最右边)
    c_left = 0;
    c_right = nums.size() - 1;
    while (c_left < c_right) // 循环结束条件（若存在，则在相等时退出）
    {
        // 由于小于等于target处可能包含target，故此情况下left = middle，以防出去
        // 即固定左边界，使右边界趋近
        int middle = c_left + c_right + 1 >> 1; // 由于/2时会发生截断。（仅剩两个元素时，middle应处于右边，故需加1）
        // 故每次只修改右边界就行
        if (nums[middle] <= target) c_left = middle; // 修改处
        else c_right = middle - 1;
    }
    if (nums[c_left] == target) res.push_back(c_left);
    return res;
}

// 3、找到第一个刚好大于等于target的数（类似于1.找第一个重复数字）
int BinarySearch2(vector<int> &nums, int target)
{
    int c_left = 0;
    int c_right = nums.size() - 1;
    while (c_left < c_right)
    {
        int middle = c_left + c_right >> 1;
        if (nums[middle] >= target) c_right = middle;
        else c_left = middle + 1;
    }

    // 与1的不同点在此。。由于要求第一个大于target，故其前面一个元素需小于target（此时得考虑越界情况）
    if (nums[c_left] >= target && (!c_left || nums[c_left - 1] < target)) return
c_left;
    return -1;
}

};

/*给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。*/

你可以假设数组中无重复元素。*/
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int start = 0;
        int end = nums.size()-1;

        while(start < end)
        {
            int middle = start + (end - start)/2;
            if(nums[middle] >= target) end = middle;
            else start = middle + 1;
        }
        if(nums[start] >= target) return start;
        return start + 1;
    }
}

```

```
};
```

前缀和

给一数组A，

前缀和：新建一数组B，数组中每一项B[i]保存A中[0...i]的和；

```
// 题1：输入n个数的数列，所有相邻m数的和有n-m+1个，求最小值
// 即求相邻m个元素的最小和
// 分析1、暴力法，二次循环，外层为第几次，内层为求和。。。再比较大小
// 分析2、前缀和，依次求出前i个和，在循环时，相减即可求得某一区间的值
#include "pch.h"
#include <iostream>
#include <unordered_map>
#include <algorithm>
using namespace std;

int main()
{
    int n, m; // n为数组长度、m为求和区间的长度
    cin >> n; // 6
    cin >> m; // 3
    vector<int> nums(n, 0);
    vector<int> sum(n, 0);

    // 给数组赋值，求前缀和
    cin >> nums[0]; // 10 4 1 5 5 2
    sum[0] = nums[0];
    for (int i = 1; i < n; i++)
    {
        cin >> nums[i];
        sum[i] = sum[i-1]+nums[i];
    }

    int min_nums = INT_MAX;
    for (int i = 0; i < n - m; i++)
    {
        min_nums = min(min_nums, sum[i + m] - sum[i]);
    }
    cout << min_nums << endl; //10
    return 0;
}
```

1524. 和为奇数的子数组数目

题干：给你一个整数数组 arr。请你返回和为 奇数 的子数组数目。由于答案可能会很大，请你将结果对 $10^9 + 7$ 取余后返回。

输入：arr = [1,3,5]

输出：4

解释：所有的子数组为 [[1],[1,3],[1,3,5],[3],[3,5],[5]]。

所有子数组的和为 [1,4,9,3,8,5]。

奇数和包括 [1,9,3,5]，所以答案为 4。

分析：首先想到的是动态规划，即第i个数所包含的和为奇数的子数组数目。最重要的是找到状态方程：

一开始的想法是定义dp_odd[i]为前i个元素包含的奇数子数组数目。。。但实例中给定[1,5]不算子数组，故该思路错。

假设 $dp[i]$ 为以*i*结束的包含包含的和为奇数的子数组数目，那么怎么求这个数？若 $nums[i]$ 为奇数，根据偶数+奇数=奇数的规则，应找出 $dp[i-1]$ 所包含的和为偶数的子数组数目。。故，**定义变量时需进一步定义 $dp_odd[i]$ 与 $dp_even[i]$ ，前者为以*i*为区间末尾的包含的和为奇数的子数组数目，后者为以*i*为区间末尾的包含的和为偶数的子数组数目。**最终，从前到后遍历一次，将各奇数子数组相加。那么进一步，状态方程可得（ $nums[i]$ 为偶数时，原先 $nums[i-1]$ 为奇数的自动变为 $dp_odd[i]$ 。当 $dp_even[i]$ 除之前的偶数外，还有其本身，故加1）：

$nums[i] == odd$ 时， $dp_o[i] = dp_e[i - 1] + 1$ 、 $dp_e[i] = dp_o[i - 1]$ 。

$nums[i] == even$ 时， $dp_o[i] = dp_e[i - 1]$ 、 $dp_e[i] = dp_e[i - 1] + 1$

```

class Solution {
public:
    int numOfSubarrays(vector<int>& arr) {
        int len = arr.size();
        if (arr.empty()) return 0;
        // 定义以i为末尾的子区间
        vector<int> dp_odd(len, 0);
        vector<int> dp_even(len, 0);

        // 总数
        long long sum = 0;

        // 边界条件
        if (arr[0] % 2)
        {
            dp_odd[0] = 1;
            sum++;
        }
        else
        {
            dp_even[0] = 1;
        }
        for (int i = 1;i<len;i++)
        {
            if (arr[i] % 2)
            {
                dp_odd[i] = dp_even[i - 1] + 1; // 奇数的话，与之前的偶数和相加，即可得到奇数（个数再加其本身）
                dp_even[i] = dp_odd[i - 1];
                sum = (sum+dp_odd[i]) % 1000000007;
            }
            else
            {
                dp_odd[i] = dp_odd[i - 1]; // 偶数的话，与之前奇数相加
                dp_even[i] = dp_even[i - 1] + 1;
                sum = (sum+dp_odd[i]) % 1000000007;
            }
        }
        // 求总的奇数和
        // for (int i = 0; i < len; i++)
        // {
        //     sum = (sum+dp_odd[i]) % 1000000007;
        // }
        return sum;
    }
};

```

另，若考虑前缀和，可提高效率。若前*i*项的和为偶数，那么，偶数- (*i*-1) 项所包含的奇子数组，则为当前增加的奇子数组的个数。（动态规划考虑的是第*i*项包含的和为奇数的子数组，而前缀和则是考虑前*i*项和）

```
class Solution {
public:
    int numOfSubarrays(vector<int>& arr) {
        long long sum = 0;
        long long res = 0;
        int odd = 0;
        int even = 1; // 空数组和为0
        for (int i = 0; i < arr.size(); i++) {
            sum = (sum + arr[i]);
            res += sum % 2 ? even : odd; // 若和为奇数，则多增了(i-1)时偶数个子数组
            if (sum % 2) odd++;
            else even++;
        }
        return res % 1000000007;
    }
};
```

// 参考自zerotrac大佬

1523. 在区间范围内统计奇数数目

题干：给你两个非负整数 *low* 和 *high*。请你返回 *low* 和 *high* 之间（包括二者）奇数的数目。

分析：直接进行high-low计算奇数的时候，由于high、low的奇偶性问题，导致无法有效计算，不妨先计算出high到0的奇数个数，再计算出*low-1*到0处的奇数个数，然后相减。

```
class Solution {
public:
    int count(const int& num) // const &只是为了减少内存占用
    {
        if(num < 1) return 0;
        return (num+1)>>1;
    }
    int countOdds(int low, int high) {
        return count(high) - count(low-1);
    }
};
```

1535. 找出数组游戏的赢家

给你一个由 不同 整数组成的整数数组 *arr* 和一个整数 *k*。

每回合游戏都在数组的前两个元素（即 *arr[0]* 和 *arr[1]*）之间进行。比较 *arr[0]* 与 *arr[1]* 的大小，较大的整数将会取得这一回合的胜利并保留在位置 0，较小的整数移至数组的末尾。当一个整数赢得 *k* 个连续回合时，游戏结束，该整数就是比赛的赢家。

返回赢得比赛的整数。

题目数据 保证 游戏存在赢家

```
// 实际上不需要判断k次，若k > arr.size-1，则一定会对比循环一圈，那么最大值一定是最后的胜者
// 但如果k小于长度：1、在一圈以内，有值获胜=k，则该值胜。。但若一圈后还没找到胜者，那么最大值为胜者。
```

```

class Solution {
public:
    int getWinner(vector<int>& arr, int k) {

        int len = arr.size() - 1;
        if (len - 1 < k) return *max_element(arr.begin(), arr.end()); // 对比k-1次就可以找打最大值

        int win = 0;
        int win_index = 0;

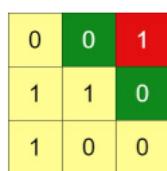
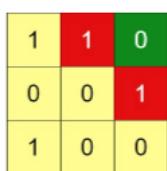
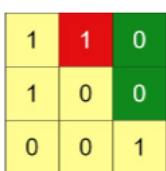
        for (int i = 1; i < len; i++)
        {
            if (arr[win_index] > arr[i])
            {
                win++;
            }
            else
            {
                win_index = i;
                win = 1;
            }
            if (win == k) return arr[win_index];
        }
        return *max_element(arr.begin(), arr.end());
    };
};

```

1536. 排布二进制网格的最少交换次数

给你一个 $n \times n$ 的二进制网格 grid，每一次操作中，你可以选择网格的 相邻两行 进行交换。一个符合要求的网格需要满足主对角线以上的格子全部都是 0。请你返回使网格满足要求的最少操作次数，如果无法使网格符合要求，请你返回 -1。主对角线指的是从 $(1, 1)$ 到 (n, n) 的这些格子。

示例 1：

	\rightarrow		\rightarrow		\rightarrow	
Swap row 1 with row 2		Swap row 2 with row 3		Swap row 1 with row 2		

输入：grid = [[0,0,1],[1,1,0],[1,0,0]]
输出：3

示例 2：



输入：grid = [[0,1,1,0],[0,1,1,0],[0,1,1,0],[0,1,1,0]]
输出：-1
解释：所有行都是一样的，交换相邻行无法使网格符合要求。

分析：1、计算每行后面连续0的个数；2、判断0的个数是否符合对角线要求、符合的话计算排序次数（依次从上往下排）

```
// 1、计算每行后面连续0的个数；2、判断0的个数是否符合对角线要求，符合的话计算排序次数（依次从上  
往下排）  
class Solution {  
public:  
    int minSwaps(vector<vector<int>>& grid) {  
        int row = grid.size();  
        int line = grid[0].size();  
        vector<int> zeros_num(row, 0);  
  
        int res = 0;  
        // 从后往前计算0的个数  
        for (int i = 0; i < row; i++)  
        {  
            int count = 0; // 0的个数  
            for (int j = line - 1; j >= 0; j--)  
            {  
                if (grid[i][j] == 0) count++;  
                else break;  
            }  
            zeros_num[i] = count;  
        }  
  
        // 从上到下判断  
        for (int i = 0; i < row; i++)  
        {  
            if (zeros_num[i] >= row - 1 - i) continue; // 若符合，则继续下一行  
  
            int j;  
            for (j = i+1; j < row; j++) // 寻找大值并交换  
            {  
                if (zeros_num[j] < row - 1 - i) continue;  
                while (j>i)  
                {  
                    swap(zeros_num[j], zeros_num[j - 1]);  
                    j--;  
                    res++;  
                }  
                break;  
            }  
            if (j == row) return -1;  
        }  
        return res;  
    }  
};
```

1537. 最大得分

你有两个有序且数组内元素互不相同的数组 `nums1` 和 `nums2`。

一条合法路径 定义如下：

选择数组 `nums1` 或者 `nums2` 开始遍历（从下标 0 处开始）。

从左到右遍历当前数组。

如果你遇到了 `nums1` 和 `nums2` 中都存在的值，那么你可以切换路径到另一个数组对应数字处继续遍历（但在合法路径中重复数字只会被统计一次）。

得分定义为合法路径中不同数字的和。

请你返回所有可能合法路径中的最大得分。

由于答案可能很大，请你将它对 $10^9 + 7$ 取余后返回。

示例 2：

输入：`nums1 = [1,3,5,7,9]`, `nums2 = [3,5,100]`

输出：109

解释：最大得分由路径 `[1,3,5,100]` 得到。

示例 3：

输入：`nums1 = [1,2,3,4,5]`, `nums2 = [6,7,8,9,10]`

输出：40

解释：`nums1` 和 `nums2` 之间无相同数字。

最大得分由路径 `[6,7,8,9,10]` 得到。

示例 4：

输入：`nums1 = [1,4,5,8,9,11,19]`, `nums2 = [2,3,4,11,12]`

输出：61

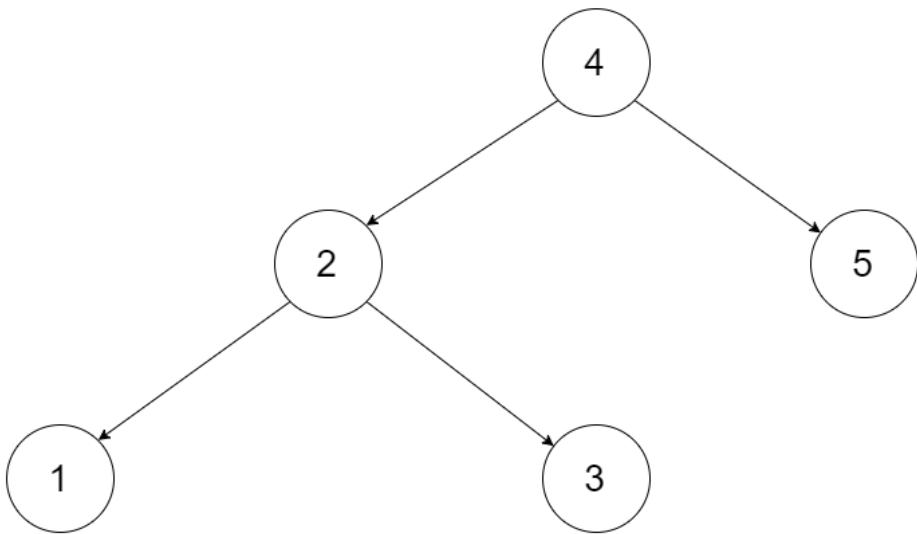
提示：

- $1 \leq \text{nums1.length} \leq 10^5$
- $1 \leq \text{nums2.length} \leq 10^5$
- $1 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^7$
- `nums1` 和 `nums2` 都是严格递增的数组。

0804

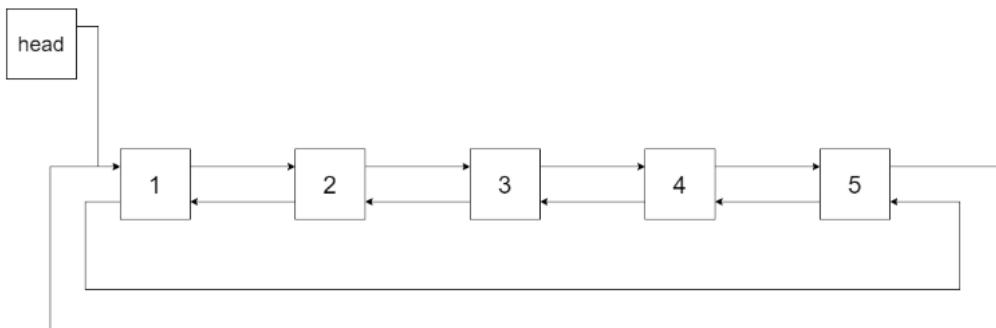
剑指 Offer 36. 二叉搜索树与双向链表

题干: 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。



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。



分析：一是二叉搜索树：一定会用到特征即左子树小于中间节点，右子树大于中间节点。。另，中序排序递增。题中要求转化为排序双向链表，则一定会采用中序遍历。

```

class Solution {
public:
    void helper(Node *root)
    {
        if (!root) return;
        helper(root->left);
        if (!head) head = root;
        else
        {
            root->left = pre;
            pre->right = root;
        }
        pre = root;
        helper(root->right);
    }
    Node* treeToDoublyList(Node* root) {
        if(!root) return NULL;
        helper(root);

        head->left = pre;
        pre->right = head;
        return head;
    }
private:
}

```

```
    Node *head;
    Node *pre, *cur;
};
```

剑指 Offer 63. 股票的最大利润

题干:假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少？

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
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。
```

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下，没有交易完成，所以最大利润为 0。
```

限制：

$0 \leq$ 数组长度 $\leq 10^5$

分析:只要找出第*i*个值之前的最小值，令f(i)-i即可

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(prices.empty()) return 0;

        int min_num = prices[0]; // 保留前i个元素的最小值
        int max_num = 0;          // 保留最大收益值

        for(int p : prices)
        {
            max_num = p - min_num > max_num ? p - min_num:max_num;
            min_num = p > min_num? min_num:p;
        }

        return max_num >= 0 ? max_num:0;
    }
};
```

剑指 Offer 31. 栈的压入、弹出序列

题干：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 {4,5,3,2,1} 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。

示例 1：

```
输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出：true
解释：我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

示例 2：

```
输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出：false
解释：1 不能在 2 之前弹出。
```

提示：

1. $0 \leq \text{pushed.length} == \text{popped.length} \leq 1000$
2. $0 \leq \text{pushed}[i], \text{popped}[i] < 1000$
3. pushed 是 popped 的排列。

分析：想通过找规律解决，但规律不明显，故需换种方法。

辅助栈：

```
class Solution {
public:
    bool validateStackSequences(vector<int>& pushed, vector<int>& popped)
    {
        stack<int> s;
        int len = pushed.size();
        int index = 0;

        for (int i = 0; i < len; i++)
        {
            s.push(pushed[i]);
            while (!s.empty() && s.top() == popped[index])
            {
                s.pop();
                index++;
            }
        }
        if(s.empty())
            return 1;
        return 0;
    }
};
```

0805

剑指 Offer 45. 把数组排成最小的数

输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

示例 1:

```
输入: [10,2]
输出: "102"
```

示例 2:

```
输入: [3,30,34,5,9]
输出: "3033459"
```

提示:

- $0 < \text{nums.length} \leq 100$

说明:

- 输出结果可能非常大，所以你需要返回一个字符串而不是整数
- 拼接起来的数字可能会有前导 0，最后结果不需要去掉前导 0

分析：1、首先考虑如何比较拼接数的大小，可按位比较，但若从字符串比较则更简单： $20+21 < 21+20$ ，则 2021 的组合最小。2、数组长度最大为100，故可用循环，那么可按此规律进行sort排序。

```
class Solution {
public:
    string minNumber(vector<int>& nums) {
        vector<string> str;
        string ans;
        // 转化为字符串数组
        for (int i = 0; i < nums.size(); i++)
        {
            str.push_back(to_string(nums[i]));
        }

        // 排序比较，利用sort函数
        sort(str.begin(), str.end(), [](string a, string b)-> bool {return a + b < b + a; });

        // 连接
        for (int i = 0; i < str.size(); i++)
            ans += str[i];
        return ans;
    }
};
```

剑指 Offer 14-I. 剪绳子

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：

```
输入: 2  
输出: 1  
解释: 2 = 1 + 1, 1 * 1 = 1
```

示例 2:

```
输入: 10  
输出: 36  
解释: 10 = 3 + 3 + 4, 3 * 3 * 4 = 36
```

提示：

- $2 \leq n \leq 58$

分析：

- 设将长度为 n 的绳子切为 a 段：

$$n = n_1 + n_2 + \dots + n_a$$

- 本题等价于求解：

$$\max(n_1 \times n_2 \times \dots \times n_a)$$

以下数学推导总体分为两步：① 当所有绳段长度相等时，乘积最大。② 最优的绳段长度为 3。

数学推导：

- 以下公式为“算术几何均值不等式”，等号当且仅当 $n_1 = n_2 = \dots = n_a$ 时成立。

$$\frac{n_1 + n_2 + \dots + n_a}{a} \geq \sqrt[a]{n_1 n_2 \dots n_a}$$

| 推论一：将绳子以相等的长度等分为多段，得到的乘积最大。

- 设将绳子按照 x 长度等分为 a 段，即 $n = ax$ ，则乘积为 x^a 。观察以下公式，由于 n 为常数，因此当 $x^{\frac{1}{a}}$ 取最大值时，乘积达到最大值。

$$x^a = x^{\frac{n}{a}} = (x^{\frac{1}{a}})^n$$

- 根据分析，可将问题转化为求 $y = x^{\frac{1}{a}}$ 的极大值，因此对 x 求导数。

$$\begin{aligned}\ln y &= \frac{1}{x} \ln x && \text{取对数} \\ \frac{1}{y} \dot{y} &= \frac{1}{x^2} - \frac{1}{x^2} \ln x && \text{对 } x \text{ 求导} \\ \dot{y} &= \frac{1 - \ln x}{x^2} && \text{整理得}\end{aligned}$$

- 令 $\dot{y} = 0$ ，则 $1 - \ln x = 0$ ，易得驻点为 $x_0 = e \approx 2.7$ ；根据以下公式，可知 x_0 为极大值点。

$$\dot{y} \begin{cases} > 0 & , x \in [-\infty, e) \\ < 0 & , x \in (e, \infty] \end{cases}$$

- 由于切分长度 x 必须为整数，最接近 e 的整数为 2 或 3。如下式所示，代入 $x = 2$ 和 $x = 3$ ，得出 $x = 3$ 时，乘积达到最大。

- 口算对比方法：给两数字同时取 6 次方，再对比。

$$\begin{aligned}[y(3)]^6 &= (3^{1/3})^6 = 9 \\ [y(2)]^6 &= (2^{1/2})^6 = 8\end{aligned}$$

推论二：尽可能将绳子以长度 3 等分为多段时，乘积最大。

切分规则：

1. 最优：3。把绳子尽可能切为多个长度为 3 的片段，留下的最后一段绳子的长度可能为 0, 1, 2 三种情况。
2. 次优：2。若最后一段绳子长度为 2；则保留，不再拆为 1 + 1。
3. 最差：1。若最后一段绳子长度为 1；则应把一份 3 + 1 替换为 2 + 2，因为 $2 \times 2 > 3 \times 1$ 。

```
class Solution {
public:
    int cuttingRope(int n) {
        if(n <= 3) return n-1;

        int cnt = n/3; // 判断几个3
        int res = n%3; // 余数

        if(res == 2) return pow(3,cnt)*2;
        if(res == 1) return pow(3,cnt-1)*4;

        return pow(3,cnt);
    }
};
```

剑指 Offer 38. 字符串的排列

输入一个字符串，打印出该字符串中字符的所有排列。你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例：

```
输入: s = "abc"
输出: ["abc", "acb", "bac", "bca", "cab", "cba"]
```

限制：

$1 \leq s$ 的长度 ≤ 8

剑指 Offer 46. 把数字翻译成字符串

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 “l” , …… , 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。

分析：分析实例1：1处有1中翻译；2处可以为12，也可以为1、2共两种；2处有1、2、2，12、2，1、22三种；5处有1、2、2、5，12、2、5，12、25，1、22、5，1、2、25五种。从中可找到状态方程，若

- 可被翻译的两位数区间：当 $x_{i-1} = 0$ 时，组成的两位数是无法被翻译的（例如 00, 01, 02, …），因此区间为 $[10, 25]$ 。

$$dp[i] = \begin{cases} dp[i-1] + dp[i-2] & , 10x_{i-1} + x_i \in [10, 25] \\ dp[i-1] & , 10x_{i-1} + x_i \in [0, 10) \cup (25, 99] \end{cases}$$

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"mzi"
```

```
class Solution {
public:
    int translateNum(int num) {
        string s = to_string(num);
        int len = s.size();
        vector<int> res(len, 1);

        res[0] = 1;

        for(int i=1;i<len;i++)
        {
            auto pre = s.substr(i - 1, 2);
            if(i == 1)
            {
                if(pre >= "10" && pre<="25") res[i] = 2;
                else res[i] = 1;
            }
            else{
                if(pre>="10" && pre<="25") res[i] = res[i-1]+res[i-2];
                else res[i] = res[i-1];
            }
        }
        return res[len-1];
    }

    // 
    class Solution {
public:
    int translateNum(int num) {
        string src = to_string(num);
        int p = 1, q = 0, r = 1;

        auto pre = src.substr(0, 2);
        if (pre <= "25" && pre >= "10") {
            r = 2;
        }
        for (int i = 2; i < src.size(); ++i) {

            pre = src.substr(i - 1, 2);
            q = r;
            if (pre <= "25" && pre >= "10") {
                r += p;
            }
        }
    }
};
```

```

        p = q;
    }
    return r;
}

```

0806

剑指 Offer 20. 表示数值的字符串

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"、"0123"都表示数值，但"12e"、"1a3.14"、"1.2.3"、"+-5"、"-1E-16"及"12e+5.4"都不是。

分析：有限状态自动机：考虑总共出现的几种状态，分别串在一起。（leetcode有测试案例报错）

```

class Solution {
public:
    bool isNumber(string s) {
        unordered_map<char, int> hm;
        vector<unordered_map<char, int>> fsm = { { { ' ', 0}, { 's', 1}, { 'n', 2},
{ '.', 3} }, // 0
                                                { { '.', 3}, { 'n', 2} },
// 1符号
                                                { { 'n', 2}, { '.', 3}, { ' ', 8}, { 'e', 7} },
// 2数字
                                                { { 'n', 4 } },
// 3小数点
                                                { { 'n', 4}, { 'e', 7}, { ' ', 8 } },
// 4小数点后数字
                                                { { 'n', 6 } },
// 5e后符号
                                                { { 'n', 6}, { ' ', 8 } },
// 6e后数
                                                { { 'n', 6}, { 's', 5 } },
// 7e
                                                { { ' ', 8 } }};

int len = s.length();
int p = 0; // p相当于上述vector的行
for (char c : s)
{
    if (c >= '0' && c <= '9') c = 'n';
    else if (c == 'e' || c == 'E') c = 'e';
    else if (c == '+' || c == '-') c = 's';

    if (fsm[p].find(c) == fsm[p].end()) return 0; // 找不到

    p = fsm[p][c];
    cout<<p<<endl;
}

```

```

    if (p == 8 || p == 6 || p == 4 || p == 2) // 注3.不算数字（代码中是这样）
        return 1;

    return 0;
};

}

```

剑指 Offer 33. 二叉搜索树的后序遍历序列 (非递归树遍历)

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。

分析：可用递归，但本题采用辅助栈：先参考1008题。

1008题是先序遍历：根-左-右，故小于栈顶元素的一定为栈顶元素的左子节点，而大于栈顶元素的不一定为栈顶元素的右子节点（因为遍历的时候先根再左后右，，栈顶元素可能为左节点，也可能为根节点，不确定）。同理，如果将本题的后序遍历逆序，即变为根-右-左，，则与1008类似，只不过，若大于栈顶元素，则为栈顶元素的右子节点；小于栈顶元素，不一定为栈顶元素的左子节点（先根再右最后左）。

```

/*
1、将根节点入栈
2 若元素大于之前pop的元素，则return false
3、判断栈顶元素与数组中的大小
    2.1、栈顶元素小于数组元素，直接入栈
    2.2、栈顶元素大于数组元素， pop直到小于，然后入栈

*/
class Solution {
public:
    bool verifyPostorder(vector<int>& postorder) {

        stack<int> s;
        int len = postorder.size();

        int pivot = INT_MAX;
        for (int i = len - 1; i >= 0; i--) {
            if (postorder[i] > pivot) return false;

            while (!s.empty() && s.top() > postorder[i])
            {
                pivot = s.top();
                s.pop();
            }
            s.push(postorder[i]);
        }
        return true;
    }
};

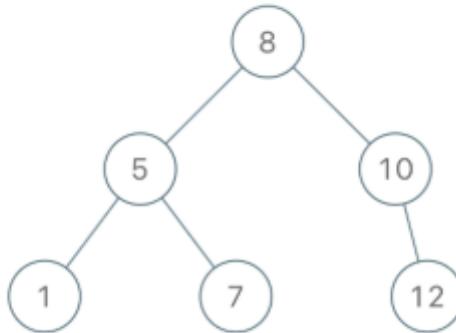
```

*1008. 先序遍历构造二叉树

返回与给定先序遍历 preorder 相匹配的二叉搜索树 (binary search tree) 的根结点。

示例：

输入： [8,5,1,7,10,12]
输出： [8,5,10,1,7,null,12]



提示：

1. $1 \leq \text{preorder.length} \leq 100$
2. 先序 preorder 中的值是不同的。

分析：

1、先序遍历：根-左-右。采用递归，每次构建根节点：第一个根节点为8，则8后为左右子树，又因为右子大于根节点，故10以后为右子树。。。对于左右子树，可按上述操作进一步构建节点。

```
class Solution {
public:
    TreeNode* helper(vector<int>& preorder, int start, int end){
        // 递归结束条件
        if(start > end) return NULL;

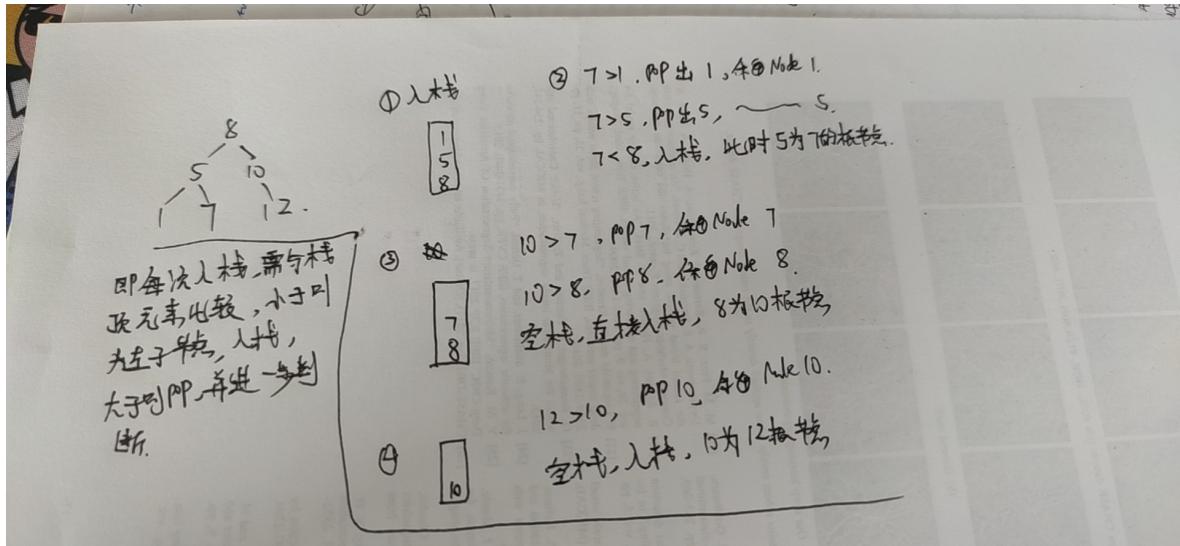
        // 递归
        TreeNode *root = new TreeNode(preorder[start]);
        int m = start+1;          // start为左子树根节点
        while(m <= end && preorder[start] > preorder[m]) m++;

        // 递归点
        root->left = helper(preorder,start+1,m-1);
        root->right = helper(preorder,m,end);

        // 返回值
        return root;
    }
    TreeNode* bstFromPreorder(vector<int>& preorder) {
        int len = preorder.size();

        return helper(preorder,0,len-1);
    }
};
```

2、若不用递归，可采用辅助栈。先序遍历的特点为根-左-右，且为二叉搜索树，故根>左、根<右。。那么对于先序来说，以实例[8,5,1,7,10,12]，5<8,1<5，所以这三个结点肯定是根结点的左节点。。如果将这3个数放入栈中，并每进一个数与栈顶对比一次，若小于栈顶元素，则入栈（说明为栈顶元素的左子结点）；若大于栈顶元素，则需pop（注，pop的时候需保留pop的指针，因为此时pop的可能是即将入栈的根节点）



```
/*
辅助栈：
1、建立根节点（先序的第一个元素），入栈
2、判断栈顶元素和先序元素的值大小：
    2.1、先序元素小于栈顶元素，说明为栈顶元素的左子节点，相连，并入栈
    2.2、先序元素大于栈顶元素，说明为右节点(但不确定是否为当前栈顶元素的右节点)
        2.2.1：找到该先序元素的根节点：即小于该元素，但其根节点的根节点大于先序元素
        2.2.2：循环pop，每次pop都保留pop出的值(可能为根节点)
        2.2.3：找到根节点，相连，并入栈。
*/
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& preorder) {
        stack<TreeNode*> s;
        int len = preorder.size();

        TreeNode *root = new TreeNode(preorder[0]); // 根节点
        s.push(root);

        for (int i = 1; i < len; i++) {
            TreeNode *pchange = new TreeNode(preorder[i]); // 构建当前节点
            // 2.1、先序元素小于栈顶元素，说明为栈顶元素的左子节点，相连，并入栈
            if (preorder[i] < s.top()->val) // 2.1 小于栈顶元素，直接入栈
            {
                s.top()->left = pchange; // 找到栈顶元素左子节点
                s.push(pchange);
            }
            // 2.2、先序元素大于栈顶元素，说明为右节点(但不确定是否为当前栈顶元素的右节点)
            else
            {
                TreeNode *tmp = s.top();
                s.pop();

                // 2.2.2：循环pop，每次pop都保留pop出的值(可能为根节点)
                while (!s.empty() && preorder[i] > s.top()->val)
                {
                    tmp->right = s.top();
                    s.pop();
                }
                tmp->right = pchange;
                s.push(tmp);
            }
        }
        return root;
    }
};
```

```

        tmp = s.top();
        s.pop();
    }
    //2.2.3: 找到根节点，相连，并入栈。
    tmp->right = pchange;
    s.push(pchange);
}
}
return root;
};


```

94. 二叉树的中序遍历

给定一个二叉树，返回它的中序遍历。

分析：递归的话比较容易，尽量使用非递归

```

/*
 * 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> inorderTraversal(TreeNode* root) {
        // 不用递归，用迭代进行中序遍历
        stack<TreeNode *> s;
        vector<int> res;
        TreeNode *current = root;
        while(current || !s.empty())
        {
            // 存在左节点，则先将其装入栈中，直到找到最底的左子节点，然后可指向右节点，再进行
            // 遍历
            if(current)
            {
                s.push(current);
                current = current->left;
            }
            else
            {
                current = s.top();
                s.pop();
                res.push_back(current->val);
                current = current->right;
            }
        }
        return res;
    };
}
```

```

// 递归
class Solution {
public:
    void helper(TreeNode *root)
    {
        if(!root) return;

        helper(root->left);
        res.push_back(root->val);
        helper(root->right);
    }
    vector<int> inorderTraversal(TreeNode* root) {
        helper(root);
        return res;
    }
private:
    vector<int> res;
};

```

144. 二叉树的前序遍历

给定一个二叉树，返回它的 前序遍历。

分析：递归

```

class Solution {
public:
    void helper(TreeNode *root)
    {
        if(!root) return;

        res.push_back(root->val);
        helper(root->left);
        helper(root->right);
    }
    vector<int> preorderTraversal(TreeNode* root) {
        helper(root);
        return res;
    }
private:
    vector<int> res;
};

```

```

/**
 * 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> preorderTraversal(TreeNode* root) {
        stack<TreeNode *> st;
        vector<int> res;

        TreeNode *current = root;

```

```

while(current || !st.empty())
{
    if(current)
    {
        res.push_back(current->val); // 注意：先序遍历应先将根节点放入
        st.push(current);
        current = current->left;
    }
    else
    {
        current = st.top();
        st.pop();
        current = current->right;
    }
}
return res;
};


```

145. 二叉树的后序遍历

给定一个二叉树，返回它的 后序遍历。

```

class Solution {
public:
    void helper(TreeNode *root)
    {
        if(!root) return;

        helper(root->left);
        helper(root->right);
        res.push_back(root->val);

    }
    vector<int> postorderTraversal(TreeNode* root) {
        helper(root);
        return res;
    }
private:
    vector<int> res;
};


```

```

class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        stack<TreeNode *> st;
        vector<int> res;

        TreeNode *current = root;
        if (root != NULL) st.push(root);
        while(!st.empty())
        {
            current = st.top();
            st.pop();
            if(current)
            {
                st.push(current);
                st.push(NULL);
            }
        }
    }
};


```

```

        if(current->right) st.push(current->right);
        if(current->left) st.push(current->left);
    }
    else
    {
        current = st.top();
        res.push_back(current->val);
        st.pop();
    }
}
return res;
}
};

```

递归法写先序后序中序

首先，先写出先序中序的简洁版，然后利用模板写出三种顺序。（后序用模板即可）。

对于先序遍历，顺序为根-左-右，那么dfs时利用栈来进行。。在存储时，最先用到的要放在栈的最上面：先放入根节点，然后弹出根节点（先序中根节点第一个输出），再将root->right和root->left存入栈（由于先左后右，故先把右节点放在栈底）。所以此时，root->left相当于了根节点，再重复上述步骤即可。

```

class Solution {
public:

    vector<int> preorderTraversal(TreeNode* root)
    {
        vector<int> res;
        stack<TreeNode *> s;

        // 由于前序遍历，需要将根节点第一个pop，所以可先让其进去
        if (root) s.push(root);
        TreeNode * cur = NULL;
        while (!s.empty())
        {
            cur = s.top();
            s.pop();

            res.push_back(cur->val);
            // 最先用到的要放到前面，最后用到的放后面
            if (cur->right) s.push(cur->right);
            if (cur->left) s.push(cur->left);
        }
        return res;
    }
};

```

前序遍历方便写的原因是，每次将根弹出并存入结果集就可，之后不会再出现该节点的相关操作。。而中序遍历，需要先保存最左节点，然后再处理根节点，所以需要先把根节点保存，在左节点输出完毕后，再处理根节点。所以需要移动指针来辅助完成操作。

```

class Solution {
public:

    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        stack<TreeNode *> s;
        TreeNode * cur = root;

```

```

// 差别在于，前序遍历第一步操作就是pop，所以可在循环前入栈
// 中序需要保存，并不需要pop，所以先不入栈
while (cur || !s.empty())
{
    // 不断把左子节点放入
    if (cur)
    {
        s.push(cur);
        cur = cur->left;
    }
    // 不存在左子节点，则说明该保存根节点，和处理右节点
    else
    {
        cur = s.top(); // pop根
        s.pop();
        res.push_back(cur->val); // 存根
        // 两种情况，若右为空，则下一步继续处理根节点（因为是从左下往上处理，无需再考
        虑左节点）
        // 若不为空，则将该右节点作为根，不断存左节点
        cur = cur->right;
    }
}
return res;
}

};


```

最后 <https://leetcode-cn.com/problems/binary-tree-inorder-traversal/solution/wan-quan-mo-fang-di-gui-bu-bian-yi-xing-miao-sha-q/>。上述链接中的模板可直接写出三种排列。对于后序，由于每次最先访问到根节点，但需要最后才pop，故不妨将根节点先入栈，且用NULL来标识这是根节点，当遇到NULL时，说明该pop根节点了。。

```

class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        stack<TreeNode *> st;
        vector<int> res;

        TreeNode *current = root;
        if (root != NULL) st.push(root);
        while(!st.empty())
        {
            current = st.top();
            st.pop();
            if(current)
            {
                st.push(current);
                st.push(NULL);
                if(current->right) st.push(current->right);
                if(current->left) st.push(current->left);
            }
            else
            {
                current = st.top();
                res.push_back(current->val);
                st.pop();
            }
        }
        return res;
    }
};


```

```
};
```

0807-BFS/DFS

目前看来，如果涉及到数，递归的话不需要队列/栈，，迭代则需队列/栈/而矩阵相关问题，不需要队列/栈。

*剑指 Offer 13. 机器人的运动范围

地上有一个m行n列的方格，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？

示例 1：

```
输入: m = 2, n = 3, k = 1  
输出: 3
```

示例 2：

```
输入: m = 3, n = 1, k = 0  
输出: 1
```

提示：

- $1 \leq n, m \leq 100$
- $0 \leq k \leq 20$

分析：利用到了广度优先搜索：<https://www.bilibili.com/video/BV1az411b7jn>

	1	2	3	4	5
1	start	1			
2	1	2			
3	2		6		
4	3	4	5	6	7
5	4		6		end

```

#include "pch.h"
#include <iostream>
#include <stack>
#include <vector>
#include <queue>
using namespace std;

// 迷宫问题，从1、1开始走，到8、8结束，问最少的路径。
/*
5
1 0 0 0 0
1 0 1 1 1
1 0 1 0 1
1 0 1 0 1
1 1 1 0 1
*/
class Point
{
public:
    int c_x;
    int c_y;
    int c_step;
};

int main()
{
    int n;
    cin >> n;
    int a[100][100] = {0}; // 存放迷宫数据的数组，默认为0，不能走(注：若不加0，则全部为随机，加0，也只是吧第一个赋值，其余默认为0)
    int b[100][100] = {0}; // 判断某点是否走过，某人为0，都可走
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            cin >> a[i][j];
        }
    }

    queue<Point> q; // 存放每点的信息
}

```

```

Point start = { 1,1,0 };
q.push(start);

// bfs
while (!q.empty())
{
    // 当前所处位置
    int x = q.front().c_x;
    int y = q.front().c_y;
    int step = q.front().c_step;

    Point tmp;

    if (x == n && y == n)
    {
        cout << "步数为" << step << endl;
        break;
    }
    // 上
    if (a[x][y - 1] && !b[x][y - 1]) // a代表能走, b判断是否走过
    {
        tmp.c_x = x;
        tmp.c_y = y - 1;
        tmp.c_step = step + 1;
        q.push(tmp);

        b[x][y - 1] = 1; // 代表已走过, 不能再走
    }
    // 下
    if (a[x][y + 1] && !b[x][y + 1]) // a代表能走, b判断是否走过
    {
        tmp.c_x = x;
        tmp.c_y = y + 1;
        tmp.c_step = step + 1;
        q.push(tmp);

        b[x][y + 1] = 1; // 代表已走过, 不能再走
    }
    // 左
    if (a[x-1][y] && !b[x - 1][y]) // 由于从11开始, 且00处a为0, 所以不用担心越界
    {
        tmp.c_x = x - 1;
        tmp.c_y = y;
        tmp.c_step = step + 1;
        q.push(tmp);

        b[x - 1][y] = 1; // 代表已走过, 不能再走
    }
    // 右
    if (a[x+1][y] && !b[x+1][y]) // a代表能走, b判断是否走过
    {
        tmp.c_x = x + 1;
        tmp.c_y = y;
        tmp.c_step = step + 1;
        q.push(tmp);

        b[x + 1][y] = 1; // 代表已走过, 不能再走
    }
    q.pop(); // 某点将路径走完之后, 即可弹出
}

```

```
}
```

257. 二叉树的所有路径

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:



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

分析：要求所有路径，故采用深度优先（dfs+栈）。那么最主要的问题在于如何处理路径，为了不搞混，可在push入栈时直接将路径也push进去。

```
// 存放在栈中的数据结构
class Node {
public:
    TreeNode * c_n;
    string c_path;
};

class Solution {
public:
    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> res; // 存放最后的结果
        if(!root) return res;

        stack<Node> s;
        Node pchange = {root,to_string(root->val)}; // stack存储的类型
        s.push(pchange); // 先把头节点放入

        while (!s.empty())
        {
            pchange = s.top();
            TreeNode *ptmp_TreeNode = pchange.c_n;
            string path = pchange.c_path;
            s.pop();

            if (!ptmp_TreeNode->left && !ptmp_TreeNode->right) res.push_back(path);
            if (ptmp_TreeNode->left) s.push({ ptmp_TreeNode->left , path + "->" +
to_string(ptmp_TreeNode->left->val) });
        }
    }
};
```

```

        if (ptmp_TreeNode->right) s.push({ ptmp_TreeNode->right , path + "->" +
to_string(ptmp_TreeNode->right->val) });

    }
    return res;
}
};

```

```

/*
先序遍历，使用递归时，套用先序遍历即可
*/
class Solution {
public:

void dfs(TreeNode* root,string path,vector<string> &res)
{
    if(!root) return;
    path += to_string(root->val);
    if(!root->left && !root->right)
    {
        res.push_back(path);
        return;
    }

    path += "->";
    dfs(root->left,path,res);
    dfs(root->right,path,res);

}
vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> res;
    string path = "";

    dfs(root,path,res);
    return res;
}c++
};


```

100. 相同的树

给定两个二叉树，编写一个函数来检验它们是否相同。如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1:

输入: 1 1
 / \ / \
 2 3 2 3

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

输出: true

示例 2:

输入: 1 1
 / \
 2 2

[1,2], [1,null,2]

输出: false

分析：1、递归，先判断根节点时的情况（a、p有q没有 q有p没有 都为false b、p、q都么有，则true，c、值不相同，false）

```
class Solution {  
public:  
    bool isSameTree(TreeNode* p, TreeNode* q) {  
        if(!p && !q) return 1;  
        if((!p && q) || (!q &&p)) return 0;  
  
        if(p->val != q->val) return 0;  
        return isSameTree(p->left,q->left) && isSameTree(p->right,q->right);  
    }  
};
```

2、广度优先遍历+队列

```
// 新建两个队列，分别存放两个数的广度优先搜索节点  
// 分别将头节点放入，然后开始判断  
class Solution {  
public:  
    bool isSameTree(TreeNode* p, TreeNode* q) {  
        queue<TreeNode *>p1, p2;  
        if (!p && !q) {  
            return true;  
        } else if (p == nullptr || q == nullptr) {  
            return false;  
        }  
  
        p1.push(p);  
        p2.push(q);  
  
        while (!p1.empty() && !p2.empty())  
        {
```

```

        TreeNode *tmp1 = p1.front();
        p1.pop();
        TreeNode *tmp2 = p2.front();
        p2.pop();

        // 根据根结点判断
        if (tmp1->val != tmp2->val) return 0; // 1、根节点
    自身是否相等
        if ((tmp1->left == NULL) ^ (tmp2->left == NULL)) return 0; // 2、左子树
    是否均存在或不存在
        if ((tmp1->right == NULL) ^ (tmp2->right == NULL)) return 0; // 3、右子树
    是否均存在或不存在

        // push
        if (tmp1->left) p1.push(tmp1->left);
        if (tmp1->right) p1.push(tmp1->right);
        if (tmp2->left) p2.push(tmp2->left);
        if (tmp2->right) p2.push(tmp2->right);
    }
    return 1;
}
};


```

剑指 Offer 12. 矩阵中的路径

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。

示例 1：

```
输入: board = [[ "A", "B", "C", "E"], [ "S", "F", "C", "S"], [ "A", "D", "E", "E"]], word =
"ABCCED"
输出: true
```

示例 2：

```
输入: board = [[ "a", "b"], [ "c", "d"]], word = "abcd"
输出: false
```

提示：

- $1 \leq \text{board.length} \leq 200$
- $1 \leq \text{board[i].length} \leq 200$

0810

696. 计数二进制子串

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都是组合在一起的。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。

示例 2：

输入： "10101"

输出： 4

解释：有4个子串：“10”，“01”，“10”，“01”，它们具有相同数量的连续1和0。

注意：

- $s.length$ 在1到50,000之间。
- s 只包含 “0” 或 “1” 字符。

分析：需要找规律，规律为：相邻01串的最小连续长度，即为这1相邻01串的字串个数。以001110进行分析，从0开始，2个0，3个1，则这两个的组合，一共有 $\min(2, 3)$ 字串，即2..。再继续，3个1，1个0，再取最小值，即为1个字串。所以结果为 $2+1=3$ 个。

```
class Solution {
public:
    int countBinarySubstrings(string s) {
        int len = s.length();
        if(len == 1) return 0;

        int slow = 0;      // 类似于指针，记录位置
        int fast = 1;
        int cnt_0 = 1;     // 第一种子串的长度

        // 找到与第一个字符不一样的字符
        while(fast<len && s[slow] == s[fast])
        {
            cnt_0++;
            fast++;
        }

        int cnt_1 = 0;
        int res = 0;
        while(fast<len)
        {
            slow = fast-1;
```

```

        while(fast<len && s[fast] != s[slow])
        {
            cnt_1++;
            fast++;
        }
        res += min(cnt_0,cnt_1);
        cnt_0 = cnt_1;
        cnt_1 = 0;
    }
    return res;
};

}

```

0811-动态规划

<https://www.bilibili.com/medialist/play/ml990309268/p1>

343. 整数拆分

给定一个正整数 n ，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。返回你可以获得的最大乘积。

示例 1:

```

输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1。

```

示例 2:

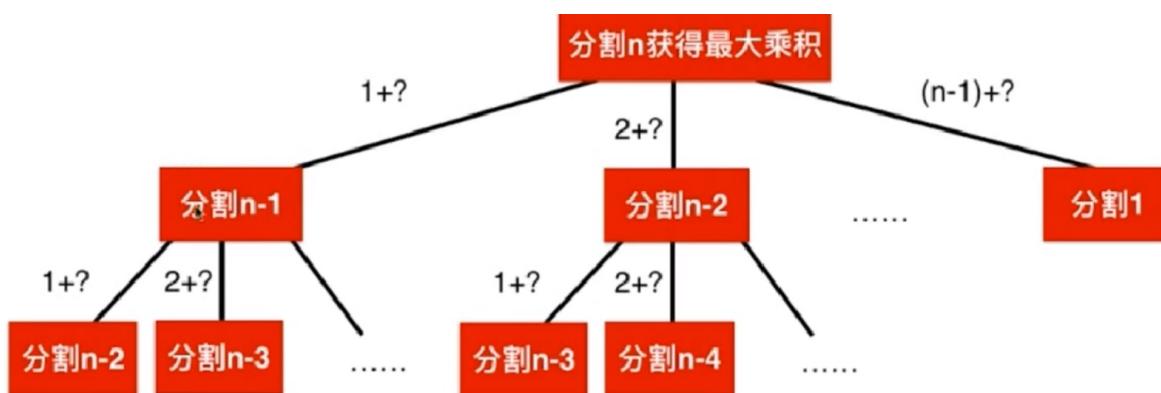
```

输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。

```

说明: 你可以假设 n 不小于 2 且不大于 58。

分析：如果暴力循环的话：至少需要分割 $\lfloor n/2 \rfloor$ 次，每次分割有 $n-i$ 个值，需要比较。



按上图其实可看出可以采用递归的方式去解决该问题。

```

// n = 27时，超出时间限制

class Solution {
public:

    int helper(int n)
    {
        if(n == 1) return 1;
        int res = 0;

        for(int i=1;i<n;i++) // 至少分割成两部分，故不能从0开始
        {
            // 注意：题中要求至少分割两次，故需从1开始到n-1结束。i
            // 但实际上第一次循环分割完毕后，之后的循环中，n可分可不分，故需比较i*(n-i)
            res = max(max(res,i * helper(n-i)),i*(n-i));
        }
        return res;
    }

    // 递归法解决
    // 第一次循环分割，得到n-1组结果。这n-1组结果可采用分割函数继续分割
    int integerBreak(int n) {
        return helper(n);
    }
};

```

可进一步优化，从图中可看到，有很多重复的数字可以循环使用，那么必节约时间。

```

class Solution {
public:

    int helper(int n,vector<int> &memo)
    {
        if(n == 1) return 1;
        int res = 0;

        // 1、
        if(memo[n] != 0) return memo[n]; // 0代表未被覆盖，若没有被覆盖，则执行下面的
        for循环
        for(int i=1;i<n;i++) // 至少分割成两部分，故不能从0开始
        {
            // 注意：题中要求至少分割两次，故需从1开始到n-1结束。i
            // 但实际上第一次循环分割完毕后，之后的循环中，n可分可不分，故需比较i*(n-i)
            res = max(max(res,i * helper(n-i,memo)),i*(n-i));
        }
        // 2、
        memo[n] = res; // return的值为当前n的最大值，故令memo记录此值
        return res;
    }

    // 递归法解决
    // 第一次循环分割，得到n-1组结果。这n-1组结果可采用分割函数继续分割
    int integerBreak(int n) {
        vector<int> memo(n+1,0); // 由于要记录1 - n的最大数，但数组是从0开始的，所以给定
        // 大小为n+1
        return helper(n,memo);
    }
};

```

再一步优化，动态规划

```

class Solution {
public:

    // 动态规划解决
    // 定义一个数组dp用来存储第i个数所能得到的最大值 ——一个循环
    // 依次从 j=1, -> j=i-j进行分割，并将求得的最大值放于dp中，外层循环
    int integerBreak(int n) {
        vector<int> dp(n+1, 0); // 由于要记录1 - n的最大数，但数组是从0开始的，所以给定大小为n+1

        for(int i=1;i<=n;i++) // 第几个数字
        {
            for(int j=1;j<i;j++) // 分割的数(当前数为i, 故最大可分割的数为1和i-1)
            {
                // dp[i-j]*j代表拆后两个数的乘积, j*(i-j)代表, 当前值不再拆的乘积
                dp[i] = max(max(dp[i-j]*j, dp[i]), j*(i-j));
            }
        }
        return dp[n];
    }

};

```

279. 完全平方数

给定正整数 n ，找到若干个完全平方数（比如 $1, 4, 9, 16, \dots$ ）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。

示例 1:

```

输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.

```

示例 2:

```

输入: n = 13
输出: 2
解释: 13 = 4 + 9.

```

```

class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);

        for (int i = 1; i <= n; i++)
        {
            int len = sqrt(i);
            if (float(sqrt(i)) - len == 0.0)
            {
                dp[i] = 1;
                continue;
            }
            for (int j = 1; j <= len; j++)
            {

```

```

        dp[i] = min(dp[i - pow(j, 2)] + dp[pow(j, 2)], dp[i]);
    }
}
return dp[n];
};

}

```

91. 解码方法

一条包含字母 A-Z 的消息通过以下方式进行了编码：

```

'A' -> 1
'B' -> 2
...
'Z' -> 26

```

给定一个只包含数字的非空字符串，请计算解码方法的总数。

示例 1：

```

输入: "12"
输出: 2
解释: 它可以解码为 "AB" (1 2) 或者 "L" (12)。

```

示例 2：

```

输入: "226"
输出: 3
解释: 它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6)。

```

分析：与剑指offer46类似，但输入的数字可能不在范围内。注意X0和0X两种特殊情况。X0是一种情况，不可当作两种。若首位数字为0，则直接返回0；若存在30，则直接输出0

```

class Solution {
public:
    int numDecodings(string s) {
        int len = s.length();
        int pre = 1;
        int cur = 1;

        if(s[0] == '0') return 0;
        for(int i=1;i<len;i++)
        {
            int tmp = cur;
            if(s[i] == '0')
            {
                if(s[i-1] == '1' || s[i-1] == '2') cur = pre;
                else return 0;
            }
            else
            {
                if(s.substr(i-1,2) >= "10" && s.substr(i-1,2)<= "26") cur = cur +
pre;
            }
        }
        return cur;
    }
};

```

```
        pre = tmp;
    }
}
return cur;

};

}
```

62. 不同路径

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径？



例如，上图是一个 7×3 的网格。有多少可能的路径？

示例 1:

输入: $m = 3, n = 2$

输出: 3

解释:

从左上角开始，总共有 3 条路径可以到达右下角。

1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右

示例 2:

输入: $m = 7, n = 3$

输出: 28

提示：

- $1 \leq m, n \leq 100$
- 题目数据保证答案小于等于 $2 * 10^9$

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> res(m, vector<int>(n, 0));
        for(int x = 0; x < m; x++)
        {
            for(int y = 0; y < n; y++)
            {
                if(x == 0 && y == 0) res[0][0] = 1;
                else if(x == 0 && y != 0) res[x][y] = res[x][y-1];
                else if(x != 0 && y == 0) res[x][y] = res[x-1][y];
                else res[x][y] = res[x-1][y]+res[x][y-1];
            }
        }
        return res[m-1][n-1];
    }
}
```

```
};
```

63. 不同路径 II

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？



网格中的障碍物和空位置分别用 1 和 0 来表示。

说明： m 和 n 的值均不超过 100。

示例 1：

输入：

```
[  
    [0,0,0],  
    [0,1,0],  
    [0,0,0]  
]
```

输出：2

解释：

3x3 网格的正中间有一个障碍物。

从左上角到右下角一共有 2 条不同的路径：

1. 向右 -> 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右 -> 向右

```
// 在上题的基础上，进一步节省空间  
class Solution {  
public:  
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {  
  
        int m = obstacleGrid.size();  
        int n = obstacleGrid[0].size();  
        vector<int> res(n, 0);  
  
        for(int x = 0; x < m; x++)  
        {  
            for(int y = 0; y < n; y++)  
            {
```

```

        if(!obstacleGrid[x][y])
    {
        if(x == 0 && y == 0) res[0] = 1;
        else if(x == 0 && y != 0) res[y] = res[y-1];
        else if(x != 0 && y == 0) res[y] = res[y];
        else res[y] += res[y-1];
    }
    else res[y] = 0;
}
return res[n-1];
};

}

```

198. 打家劫舍

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，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果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金额的非负整数数组，计算你 不触动警报装置的情况下，一夜之内能够偷窃到的最高金额

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 。

```

示例 2：

```

输入：[2,7,9,3,1]
输出：12
解释：偷窃 1 号房屋（金额 = 2），偷窃 3 号房屋（金额 = 9），接着偷窃 5 号房屋（金额 = 1）。
偷窃到的最高金额 = 2 + 9 + 1 = 12 。

```

提示：

- $0 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums[i]} \leq 400$

分析：

```

// 以2、7、9、3、1为例进行分析。。
// res存储的为，到第i个房间时，所能偷到的最大价值。。那么，有两种可能，1是res【i-1】，2是
// res[i-2]+第i个数字。
// 以3为例，要么偷得的为3+7（3+res[1]），要么为9+2(res[2])。
class Solution {
public:
    int rob(vector<int>& nums)
    {
        int len = nums.size();
        vector<int> res(len,0);

```

```

if(len == 0) return 0;

if(len >= 1) res[0] = nums[0];
if(len >= 2) res[1] = max(res[0],nums[1]);

for(int i=2;i<len;i++)
{
    res[i] = max(res[i-1],res[i-2]+nums[i]);
}

return *max_element(res.begin(),res.end());
}
};

```

```

// 进一步优化
class Solution {
public:
    int rob(vector<int>& nums)
    {
        int len = nums.size();
        if(len == 0) return 0;

        int pre = 0;
        int cur = nums[0];
        for(int i=1;i<len;i++)
        {
            int tmp = cur;
            cur = max(cur,pre+nums[i]);
            pre = tmp;
        }

        return cur;
    }
};

```

213. 打家劫舍 II

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。这个地方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金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

示例 1:

输入: [2,3,2]
输出: 3
解释: 你不能先偷窃 1 号房屋 (金额 = 2)，然后偷窃 3 号房屋 (金额 = 2)，因为他们是相邻的。

示例 2:

输入: [1,2,3,1]
输出: 4
解释: 你可以先偷窃 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。

分析：该题与198相似，只不过本题中是环状街道，即最后一个房子与第一个房子相连。以(1、2、3、1)为例，环状实际上为(1, 2, 3, 1, 1, 2, 3, 1)即多了一个条件，除相邻元素不能直接相连以外，第1个房子与第四个房子也不能同时被偷。那么可分类讨论，分别考虑有第4个房子和没第4个房子时的最大价值，然后再求二者最大值即为最后结果。

```
class Solution {
public:
    int helper(vector<int>& nums,int start,int end)
    {
        int cur = nums[start];
        int pre = 0;
        for(int i = start+1;i <= end;i++)
        {
            int tmp = cur;
            cur = max(cur,pre + nums[i]);
            pre = tmp;
        }
        return cur;
    }
    int rob(vector<int>& nums) {
        int len = nums.size();
        if(len == 0) return 0;
        if(len == 1) return nums[0];
        if(len == 2) return max(nums[0],nums[1]);

        return max(helper(nums,0,len-2),helper(nums,1,len-1));
    }
};
```

337. 打家劫舍 III (未作)

在上次打劫完一条街道之后和一圈房屋后，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，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

输入: [3,2,3,null,3,null,1]



输出: 7

解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

示例 2:

输入: [3,4,5,1,3,null,1]



输出: 9

解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.

0812-动态规划

背包总结

背包问题通常是多种物品有多个属性，且已知条件为某属性被受限，求另一属性的最大/最小/等于/存在不存在。以0-1背包为例解释：n个物品具有的属性为重量和价值，其中总重量C将重量的属性限制住，求最大价值，即求另一属性的特征。

针对背包问题：

1、先判断属于0-1背包还是完全背包。

2、看是求最大值/最小值/等值/是否存在/排列/组合（排列/组合问题通常出现在完全背包中）。

确定了背包类型及要求的问题后，即可下手做题。

1、0-1背包

0-1背包是指物品只能使用一次，故通常其状态方程为：

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-wi] + vi)$$

式中： $dp[i][j]$ ：将前*i*个物品放入容量为*j*的背包中可得的最大价值。 $dp[i-1][j]$ 是指不取第*i*个物品所的最大价值。 $dp[i-1][j-wi] + vi$ 则为取*i*个物品时所得的最大价值。

同样，若求最小值则为 $dp[i][j] = \min(dp[i-1][j], dp[i-1][j-wi] + vi)$ 。

若求等值个数则为

$$dp[i][j] = dp[i-1][j] + dp[i-1][j-wi])$$

式中： $dp[i][j]$ ：前*i*个元素可组成和为*j*的个数。 $dp[i-1][j]$ 是指不取第*i*个物时，前*i-1*个物品可组成和为*j*的个数。 $dp[i-1][j-wi] + vi$ 则为取*i*个物品时可组成和为*j*的个数。

或求是否存在题，状态转移为：

$$dp[i][j] = dp[i-1][j] || dp[i-1][j-wi])$$

式中： $dp[i][j]$ ：前*i*个元素是否可组成和为*j*的个数。 $dp[i-1][j]$ 是指不取第*i*个物时，前*i-1*个物品是否可组成和为*j*的个数。 $dp[i-1][j-wi] + vi$ 则为取*i*个物品时是否可组成和为*j*的个数。

在了解完0-1基本概念及相关问题后，可进一步编程实现。建议先看下文中[0-1背包问题](#)，详细介绍了背包问题的思路及优化方法。。。现以一道0-1背包题解释优化后的代码（套路，理解了即可解决许多问题）。

输入：第一行输入两个整数N和X（N代表物品个数、X代表背包容量）

第二行输入第1个物品的价格A1和所占容量B1

第N+1行代表第第1个物品的价格A2和所占容量B2

输出：在背包容量X内所的最大价格

分析：首先，该题符合0-1背包。其次要求计算最大价格。那么套用之前的状态转移方程可得代码：

```
#include "pch.h"
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main() {

    int N, X;
    cin >> N >> X;

    vector<int> A(N, 0);
    vector<int> B(N, 0);
    for (int i = 0; i < N; i++)
    {
        cin >> A[i] >> B[i];
    }

    // 1、优化后计算只需一维vector即可
    vector<int> dp(X + 1, 0);

    for (int i = 0; i < N; i++)
    {
```

```

// 2、根据状态方程知道，每次获得值时，其实只需要上一层的值及该层之前的值，故只要逆序
获得值，即不会覆盖上一层的值
for (int j = X; j >= B[i]; j--)
{
    dp[j] = max(dp[j], dp[j - B[i]]+A[i]); // 3、dp[j]为不取第i个值时，所得最
大价值。另一个为取该值时(j-A[i])容量所得最大价值
}
cout << dp[X];
}

```

另：leetcode416为判断是否存在问题，可自行思考，现附上代码：

```

class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int len = nums.size();

        // 求总数
        int sum = 0;
        for(int i=0;i<len;i++)
        {
            sum += nums[i];
        }
        // 若为奇数，直接返回0
        if(sum % 2) return 0;

        int target = sum >> 1;

        vector<bool> dp(target+1,0);
        dp[0] = true; // 初始值可先忽略，然后在分析的时候就可知应为多少

        for(int i=0;i<len;i++)
        {
            for(int j=target;j>=nums[i];j--) // 注意范围
            {
                dp[j] = dp[j] || dp[j-nums[i]];
            }
        }

        return dp[target];
    }
};

```

2、完全背包

完全背包与0-1的差别在于，完全背包中每个物品的取值无限制，故对于 $dp[i][j]$ ，当取*i*时，则 $dp[i][j]$ 为 $dp[i][j-wi]+vi$ （由于*i*有无限个，故取*i*时，则仍位于*i*处，而不是*i-1*处。这处即为二者的区别）。由于状态转移方程有些许差别，其在代码编写过程中，也不一样。。以leetcode322为例：

```

class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int len = coins.size();

        // vector<long> dp(amount+1,INT_MAX);
        vector<long> dp(amount+1,amount+1); // 1、若选用INT_MAX过于浪费空间（因为硬币不
可能为0，至少为1，故最多也不超过amount+1）

```

```

dp[0] = 0; // 2、初始值在分析问题时确定

for(int i=0;i<len;i++)
{
    for(int j = coins[i];j<=amount;j++) // 3、注：无限背包需要本层的之前元素，故直接从头覆盖即可（不懂得见状态方程）
    {
        dp[j] = min(dp[j],dp[j-coins[i]]+1); // 每次取i都得加一个硬币，这也是为啥dp[0] = 0
    }
}

return dp[amount] >= amount+1 ? -1:dp[amount];
}
};

```

另外，还需注意的是，完全背包在求得到目标的组合个数时，通常会有排序和不排序两种方法。具体编程实现也有区别，见leetcode518与377.

```

class Solution {
public:
    int change(int amount, vector<int>& coins) {

        int len = coins.size();
        vector<int> dp(+1,0);

        dp[0] = 1;
        for(int i=0;i<len;i++) // 与排列顺序无关，故按之前套路来即可
        {
            for(int j = coins[i];j<=amount;j++)
            {
                dp[j] = dp[j]+ dp[j-coins[i]];
            }
        }
        return dp[amount];
    }
};

```

```

class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        int len = nums.size();

        vector<long> dp(target+1,0);

        dp[0] = 1; // 1、状态转移方程求得是组合，与硬币数不一样，故初始化应为1
        for(int i=1;i<= target;i++) // 求排列的问题，则应该按此种模板进行
        {
            for(int j = 0;j<len;j++)
            {
                if(nums[j] <= i)
                    dp[i] = (dp[i] + dp[i-nums[j]])%INT_MAX; // dp[i-nums[j]] 意味着，目标值为i-nums[i]对应点组合数
            }
        }
        return dp[target];
    }
};

```

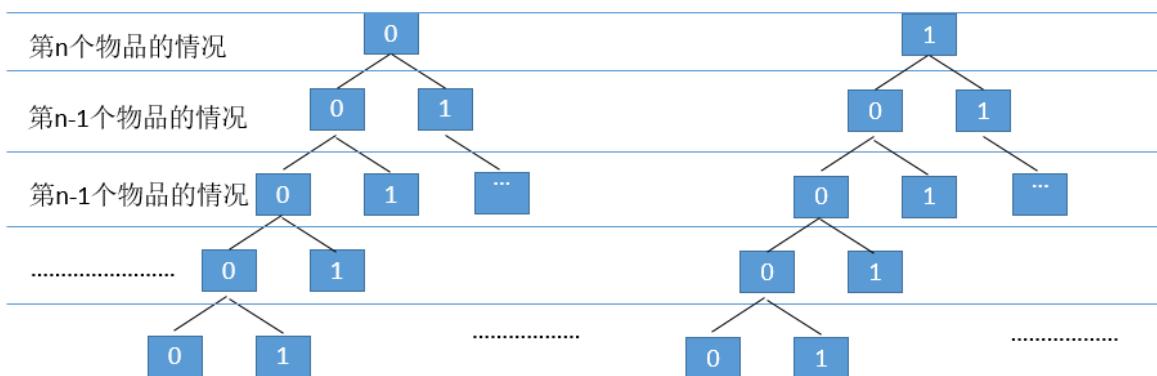
0-1背包问题

给定n个重量为 $w_1, w_2, w_3, \dots, w_n$ ，价值为 $v_1, v_2, v_3, \dots, v_n$ 的物品和容量为C的背包，求这个物品中一个最有价值的子集，使得在满足背包的容量的前提下，包内的总价值最大。

0-1背包问题指的是每个物品只能使用一次

分析：同样，若没有思路可以考虑暴力循环的方法，即先找到多少种情况（ 2^n ），再每一种情况是否符合要求并求价值，求取最大值， $O((2^n)^n)$ 。

若进一步考虑，可定义表达式： $F(n, c)$ ，意思为将n个物品装入容量为c的背包中，能取得的最大值。如下图所示，采用递归的方式可以避免暴力。对于第n个物品，可以有两种取法，取（1）或不取（0），当取的时候，则最大值为 $v(n) + F(n-1, c-w_n)$ ，即将n-1个物品装入 $c-w_n$ 的容量中，所能得到的最大价值；当不取的时候，则为 $F(n-1, c)$ ，因为没取n，故容量不变。那么，若采取递归计算的话，实际上每层求 $\max(v(n) + F(n-1, c-w_n), F(n-1, c))$ ，并返回给最上层即可。



```
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// 递归法解决0-1背包问题

class Solution
{
public:

    // index为第几个物品，c为当前容量,
    int getMaxValue( vector<int> &weight, vector<int> &value, int index, int c)
    {
        if (index < 0 || c <= 0) return 0;

        int res = 0;

        // 两种情况，取index和不取index，并求最大值
        if (weight[index] <= c) res = value[index] + getMaxValue(weight, value, index-1, c - weight[index]);
        res = max(res, getMaxValue(weight, value, --index, c));

        return res;
    }
};
```

```

int main()
{
    vector<int> weight = { 2,1,3,2 };
    vector<int> value = { 12,10,20,15 };
    Solution s;
    cout<<(s.getMaxValue(weight, value, weight.size() - 1, 5)); // 37
    return 0;
}

```

可采用记忆化搜索对优化

```

#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// 递归法解决0-1背包问题

class Solution
{
public:

    // index为第几个物品, c为当前容量,
    int getMaxValue( vector<int> &weight, vector<int> &value, int index, int c,
    vector<vector<int> > &memo)
    {
        if (index < 0 || c <= 0) return 0;

        // 1、
        if (memo[index][c]) return memo[index][c];
        int res = 0;

        // 两种情况, 取index和不取index, 并求最大值
        if (weight[index] <= c) res = value[index] + getMaxValue(weight, value, index-1, c - weight[index], memo);
        res = max(res, getMaxValue(weight, value, index-1, c, memo));

        // 2、
        memo[index][c] = res;
        return res;
    }
};

int main()
{
    vector<int> weight = { 2,1,3,2 };
    vector<int> value = { 12,10,20,15 };
    // 3、
    vector<vector<int> > memo(weight.size(), vector<int>(6, 0));
    Solution s;
    cout<<(s.getMaxValue(weight, value, weight.size() - 1, 5, memo));
    return 0;
}

```

进一步，由于该递归中存在最优子结构，故可从下到上采用动态规划的办法。可参考机器走路问题82。不过本题中二维条件不够明显，需进一步分析。首先，题中要求，n个物品最多可装入容量为c的背包的最大价值，写成表达式为F(n,c)，那么这是一个二维函数，将82中的竖坐标当作n，横坐标当作容量c，那么F(i, ci)代表前i个物品装入ci背包中能得到的最大价值。。接下来需要找状态转移方程，由于背包可装可不装，那就分情况讨论，若i装，则 $F(i, ci) = vi + f(i-1, ci-wi)$ ：即第i个物品的价值，加上，前i-1个物品装入 $ci-wi$ 背包中的最大价值。若i不装，则 $F(i, ci) = F(i-1, ci)$ ：即前i-1个物品装入 ci 背包中的最大价值。故状态转移方程为 $F(i, ci) = \max(vi + f(i-1, ci-wi), F(i-1, ci))$ 。

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// 动态规划解决0-1背包
int main()
{
    vector<int> weight = { 2,1,3,2 }; // 物品重量
    vector<int> value = { 12,10,20,15 }; // 物品价值

    // 当成二维图形，横坐标为容量，纵坐标为物品
    int c = 5;
    vector<vector<int>> dp(weight.size(), vector<int>(c+1, 0));

    // 第一列不用赋初值，因为第一列的容量都为0，故不可能放入物品
    // 先将第一行赋初值
    for (int i = 0; i <= 5; i++)
    {
        dp[0][i] = weight[0] <= i ? value[0] : 0;
    }
    for (int i = 1; i < weight.size(); i++)
    {
        for (int j = 1; j <= 5; j++)
        {
            if (weight[i] <= j) dp[i][j] = value[i] + dp[i - 1][j - weight[i]];
            dp[i][j] = max(dp[i-1][j], dp[i][j]);
        }
    }
    cout << dp[weight.size()-1][5];
    return 0;
}
```

动态规划的优化基本上均是尽可能降低空间复杂度。通常是二维变一维，一维变变量。

- 1、从上述循环中可以看到，每次循环只用到了两行数据，故可用求余进行处理。
- 2、再进一步分析，可从右向左进行刷新，这样只保存一行数据就可不断求得值

```
// 1、变为两行数组
// 动态规划解决0-1背包
int main()
{
    vector<int> weight = { 2,1,3,2 }; // 物品重量
    vector<int> value = { 12,10,20,15 }; // 物品价值

    // 当成二维图形，横坐标为容量，纵坐标为物品
    int c = 5;
    // 优化1、
```

```

vector<vector<int>> dp(2, vector<int>(c+1, 0));

// 第一列不用赋初值，因为第一列的容量都为0，故不可能放入物品
// 先将第一行赋初值
for (int i = 0; i <= 5; i++)
{
    dp[0][i] = weight[0] <= i ? value[0] : 0;
}
for (int i = 1; i < weight.size(); i++)
{
    for (int j = 1; j <= 5; j++)
    {
        // 优化2、
        if (weight[i] <= j) dp[i % 2][j] = value[i] + dp[(i - 1) % 2][j -
weight[i]];
        dp[i % 2][j] = max(dp[(i-1) % 2][j], dp[i % 2][j]);
    }
}
// 优化3、
cout << dp[(weight.size()-1) % 2][5];
return 0;
}

```

```

// 动态规划解决0-1背包
int main()
{
    vector<int> weight = { 2,1,3,2 };    // 物品重量
    vector<int> value = { 12,10,20,15 }; // 物品价值

    // 当成二维图形，横坐标为容量，纵坐标为物品
    int c = 5;

    // 优化1、
    vector<int> dp(c+1, 0);

    for (int i = 0; i <= 5; i++)
    {
        dp[i] = weight[0] <= i ? value[0] : 0;
    }

    for (int i = 1; i < weight.size(); i++)
    {
        // 逆序考虑，则不会覆盖数据
        for (int j = c; j >= 0; j--)
        {
            int tmp = dp[j];
            // 优化2、
            if (weight[i] <= j) dp[j] = value[i] + dp[j - weight[i]];
            dp[j] = max(dp[j], tmp);
        }
    }
    // 优化3、
    cout << dp[c];
    return 0;
}

```

分组背包问题——有依赖背包

王强今天很开心，公司发给N元的年终奖。王强决定把年终奖用于购物，他把想买的物品分为两类：主件与附件，附件是从属于某个主件的，下表就是一些主件与附件的例子：

主件	附件
电脑	打印机，扫描仪
书柜	图书
书桌	台灯，文具
工作椅	无

如果要买归类为附件的物品，必须先买该附件所属的主件。每个主件可以有0个、1个或2个附件。附件不再有从属于自己的附件。王强想买的东西很多，为了不超出预算，他把每件物品规定了一个重要度，分为5等：用整数1~5表示，第5等最重要。他还从因特网上查到了每件物品的价格（都是10元的整数倍）。他希望在不超过N元（可以等于N元）的前提下，使每件物品的价格与重要度的乘积的总和最大。

设第j件物品的价格为v[j]，重要度为w[j]，共选中了k件物品，编号依次为j1, j2, ……, jk，则所求的总和为：

$v[j_1]*w[j_1]+v[j_2]*w[j_2]+\dots+v[j_k]*w[j_k]$ 。（其中 * 为乘号）

请你帮助王强设计一个满足要求的购物单。log.csdn.net/qq_27139155

输入描述:

输入的第 1 行，为两个正整数，用一个空格隔开： $N \ m$
(其中 N (<32000) 表示总钱数， m (<60) 为希望购买物品的个数。)

从第 2 行到第 $m+1$ 行，第 j 行给出了编号为 $j-1$ 的物品的基本数据，每行有 3 个非负整数 $v \ p \ q$

(其中 v 表示该物品的价格 ($v < 10000$)， p 表示该物品的重要度 ($1 \sim 5$)， q 表示该物品是主件还是附件。如果 $q=0$ ，表示该物品为主件，如果 $q>0$ ，表示该物品为附件， q 是所属主件的编号)

输出描述:

输出文件只有一个正整数，为不超过总钱数的物品的价格与重要度乘积的总和的最大值 (<200000)。

示例1

输入

```
1000 5  
800 2 0  
400 5 1  
300 5 1  
400 3 0  
500 2 0
```

输出

```
2200
```

https://blog.csdn.net/qq_27139155

分析：1、物品分附件和主件，若买附件，则必须先买主件。（不能单独将物品作为纵坐标）

2、钱是10的整数倍，故横坐标只需要10的整数倍即可，无需从1开始

3、结果要求求得价值乘权重的最大值。

上述三条为本题与01背包不一样，但主要差别在于第一条。。由于必须买主件才能买附件，即不能只买附件不能买主件，那么假设有物品a、b、c，a为主件，b、c为附件，则每次只有(a) (a, b) (a, c) (a, b, c) 或 (a) 五种情况中选取一种，那么本题纵坐标不再为物品种类，而为物品组类。每次涉及到该组时，先计算F(i, c)下，该组的最大权重*价值。

0816

300. 最长上升子序列

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例：

输入： [10, 9, 2, 5, 3, 7, 101, 18]

输出： 4

解释： 最长的上升子序列是 [2, 3, 7, 101]，它的长度是 4。

说明：

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 $O(n^2)$ 。

进阶：你能将算法的时间复杂度降低到 $O(n \log n)$ 吗？

```
// 最长上升子序列
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int len = nums.size();
        vector<int> dp(len, 0);

        if(len == 0) return 0;
        for (int i = 0; i < len; i++) {
            int max_len = 0; // 记录前i-1中，比num[i]小的数的最长上升子序列
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) max_len = max(max_len, dp[j]);
            }
            dp[i] = max_len + 1;
        }
        return *max_element(dp.begin(), dp.end());
    }
};
```

376. 摆动序列

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）可能是正数或负数。少于两个元素的序列也是摆动序列。

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的最后一个差值为零。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

示例 1：

```
输入: [1,7,4,9,2,5]
输出: 6
解释: 整个序列均为摆动序列。
```

示例 2：

```
输入: [1,17,5,10,13,15,10,5,16,8]
输出: 7
解释: 这个序列包含几个长度为 7 摆动序列，其中一个可为
[1,17,10,13,10,16,8]。
```

分析：一看到求最短最长等问题就可以考虑动态规划。设 $dp[i]$ 为前*i*个元素中最长摆动序列的长度，对于第*i*个元素，有两种情况，一是 $num[i]-nums[i-1]$ 与 $num[i-1] - num[i-2]$ 异号，则 $dp[i] = dp[i-1]+1$;若同号，则 $dp[i]=dp[i-1]$

```
// 最长摆动子序列
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {

        int len = nums.size();
        vector<int> dp(len, 1);
        if (len < 2) return len; // 少于两个元素也是(注: 不包括2)

        if (nums[1]-nums[0] != 0) dp[1] = 2;

        for (int i = 2; i < len; i++) {
            dp[i] = dp[i - 1] + ((nums[i]-nums[i-1]) * (nums[i-1]-nums[i-2]) < 0);
        }
        return dp[len - 1];
    }
};

// 错误代码，上述代码只考虑了相邻元素，忽略了相邻元素相等的情形。[3,3,3,2,5] 就会出错
```

分析2：上述代码发生错误的原因是，可能出现相邻元素相等的情况，且相邻元素后若是上升即（1，1，1，2）则需要考虑相等元素之前下降的序列。。。针对此问题，可以定义两个变量 $dp_up[i]$ 与 $dp_down[i]$ 用来表示前*i*个元素中，以上升/下降为止的序列的最长长度

```
// 最长摆动子序列
class Solution {
```

```

public:
    int wiggleMaxLength(vector<int>& nums) {

        int len = nums.size();
        int dp_up = 1;
        int dp_down = 1;

        if (len < 2) return len; // 少于两个元素也是

        if (nums[0] != nums[1]) // 若相等，则最长为1
            nums[1] - nums[0] > 0 ? dp_up = 2 : dp_down = 2; // 判断第二个元素为上升还是下降

        for (int i = 2; i < len; i++) {
            // 若i为上升，则需找到之前的下降
            if (nums[i] - nums[i - 1] > 0) dp_up = dp_down + 1;
            else if (nums[i] - nums[i - 1] < 0) dp_down = dp_up + 1;
        }
        return max(dp_down, dp_up);
    }
};

```

注：本题与300的不同处在于：300最长序列值不一定会随长度越长而递增，所以最后需要遍历dp求最大值，而本题中dp要么增大要么不变，故只需要求最后一步的dp即可得到最大值。

*1143. 最长公共子序列

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长公共子序列的长度。

一个字符串的 子序列是指这样一个新的字符串：它是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）后组成的新字符串。

例如，“ace”是“abcde”的子序列，但“aec”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同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

示例 1:

```
输入: text1 = "abcde", text2 = "ace"
输出: 3
解释: 最长公共子序列是 "ace", 它的长度为 3。
```

示例 2:

```
输入: text1 = "abc", text2 = "abc"
输出: 3
解释: 最长公共子序列是 "abc", 它的长度为 3。
```

示例 3:

```
输入: text1 = "abc", text2 = "def"
输出: 0
解释: 两个字符串没有公共子序列, 返回 0。
```

提示:

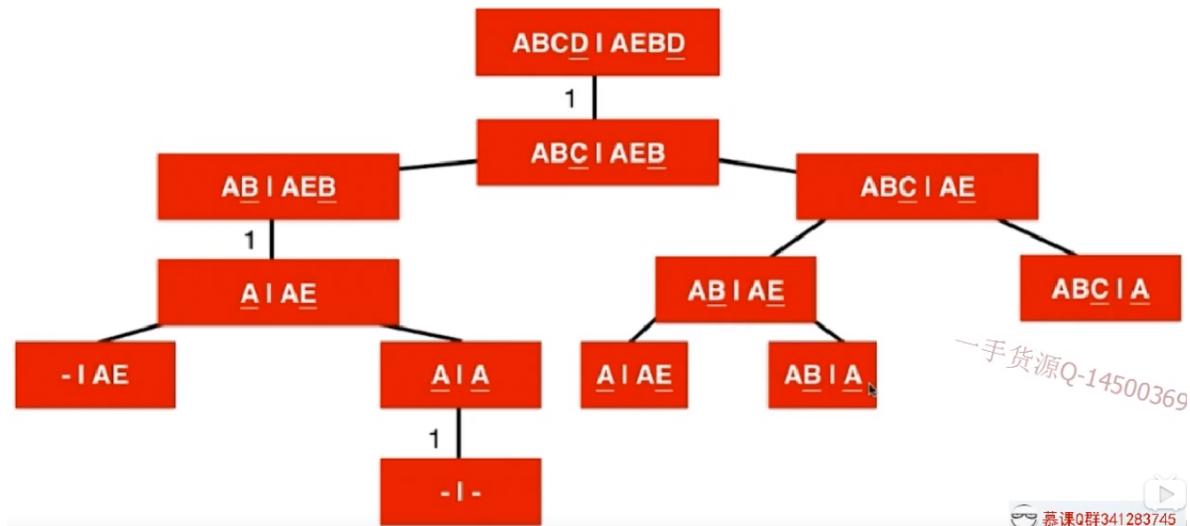
- $1 \leq \text{text1.length} \leq 1000$
- $1 \leq \text{text2.length} \leq 1000$
- 输入的字符串只含有小写英文字符。

分析：不知道该题怎么解的时候，尝试用暴力循环。以abcd|acde为例，暴循时，先令abcd的a固定，挨个与acde判断，直至结束或相等，，a=a故，len+1..。然后从b判断，b != c、b != d、b != e故b没有，再c=c，len+1；依次循环判断。。时间复杂度为O(n^2)。

```
// 实际上，暴力循环也不好写，因为题中子序列要求相对顺序不变，下列代码求得的长度则会改变相对顺序
// 错误案例: "oxcpqrsvwf"  "shmtulqrypy"
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        // 暴力循环
        int len1 = text1.length();
        int len2 = text2.length();

        int lcs = 0;
        for (int i = 0; i < len1; i++) {
            for (int j = 0; j < len2; j++) {
                if (text1[i] == text2[j])
                {
                    lcs++;
                    text2[j] = '/'; // 避免重复
                    break;
                }
            }
        }
        return lcs;
    }
};
```

考虑递归，如果text1[0] = text2[0]，则len = max(BCD,EBD的公共长度)。



```
// 超时

// 最长公共子序列
class Solution {
public:
    int helper(string text1, int start1, string text2, int start2) {
        // 递归出口
        if (start1 >= text1.length() || start2 >= text2.length()) return 0;

        if (text1[start1] == text2[start2]) return helper(text1, start1 + 1, text2,
start2 + 1); //若相同，则两个字符串指针均向后一位

        // 返回值
        return max(helper(text1, start1, text2, start2 + 1), helper(text1, start1 + 1,
text2, start2));
    }
    int longestCommonSubsequence(string text1, string text2) {
        // 递归
        return helper(text1, 0, text2, 0);
    }
};
```

从递归图中可以看到，存在重复计算的步骤，故可通过记忆化搜索/动态规划进行计算。通过递归图可以看出，与背包问题类似，故可假设纵坐标为text1的各字符，横坐标为text2的各字符

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {

        int row = text1.length();
        int line = text2.length();
        vector<vector<int>> dp(row+1, vector<int>(line+1, 0));
        for (int i = 1; i <= row; i++) {
            for (int j = 1; j <= line; j++) {
                if (text1[i-1] == text2[j-1]) dp[i][j] = dp[i - 1][j - 1] + 1;
                else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
```

```

        return dp[row][line];
    }
};

// 空间优化
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {

        int row = text1.length();
        int line = text2.length();
        vector<vector<int>> dp(2, vector<int>(line+1, 0));
        for (int i = 1; i <= row; i++) {
            for (int j = 1; j <= line; j++) {
                if (text1[i-1] == text2[j-1]) dp[i%2][j] = dp[(i - 1)%2][j - 1] + 1;
                else dp[i%2][j] = max(dp[(i - 1)%2][j], dp[i%2][j - 1]);
            }
        }
        return dp[row%2][line];
    }
};

```

0817-等值背包

416. 分割等和子集

给定一个只包含正整数的非空数组。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。

注意:

1. 每个数组中的元素不会超过 100
2. 数组的大小不会超过 200

示例 1:

输入: [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: [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.

分析：可将此题转化为背包问题。即设 $F(i, c)$ ： i 个物品是否可以装入容量为 c 的背包中。那么两种情况，一种装 i , $F(i-1, c-w_i)$, 一种不装 i , $F(i-1, c)$ 。

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int len = nums.size();

        // 求总数
        int sum = 0;
        for(int i=0;i<len;i++)
        {
            sum += nums[i];
        }
        if(sum % 2) return 0;

        int target = sum >> 1;

        vector<bool> dp(target+1,0);
        // 初始化第一行
        for(int i=1;i<=target;i++)
        {
            dp[i] = (nums[0] == i);
        }
        dp[0] = 1; // 加不加都可；以1 5 11 5 为例，若不加这个，则11处为0.但会在5, 5, 1处为1.不影响结果

        for(int i=1;i<len;i++)
        {
            for(int j=target;j>0;j--)
            {
                bool tmp = dp[j];
                if(nums[i] <= j) dp[j] = dp[j-nums[i]]; // 注意下标
                dp[j] = dp[j] || tmp;
            }
        }

        return dp[target];
    }

};

// 优化
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int len = nums.size();

        // 求总数
        int sum = 0;
        for(int i=0;i<len;i++)
        {
            sum += nums[i];
        }
        if(sum % 2) return 0;

        int target = sum >> 1;

        vector<bool> dp(target+1,0);
```

```

dp[0] = true;

for(int i=0;i<len;i++)
{
    for(int j=target;j>=nums[i];j--) // 注意范围
    {
        dp[j] = dp[j] || dp[j-nums[i]];
    }
}

return dp[target];
}
};

```

322. 零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

示例 1:

```

输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1

```

示例 2:

```

输入: coins = [2], amount = 3
输出: -1

```

说明:

你可以认为每种硬币的数量是无限的。

分析：该题为无限背包问题，对于这类问题，只需要在0-1的基础上改变一下即可。设 $dp[i][j]$ 为前*i*种硬币所能组成和为*j*的最小硬币个数，则有两种情况，取*i*，则 $dp[i][j] = dp[i-1][j-coins[i]] + 1$ ；不取*i*，则 $dp[i][j] = dp[i-1][j]$ ；再取二者最小值。

```

class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int len = coins.size();

        // vector<long> dp(amount+1, INT_MAX);
        vector<long> dp(amount+1, amount+1); // 若选用INT_MAX过于浪费空间（因为硬币不可能为0，至少为1，故最多也不超过amount+1）

        dp[0] = 0;

        for(int i=0;i<len;i++)
        {
            for(int j = coins[i];j<=amount;j++) // 注意此处范围
            {
                dp[j] = min(dp[j], dp[j-coins[i]]+1);
            }
        }
    }
};

```

```

    }
    return dp[amount] >= amount+1 ? -1:dp[amount];
}
};

```

518. 零钱兑换 II

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

```

输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1

```

示例 2:

```

输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。

```

示例 3:

```

输入: amount = 10, coins = [10]
输出: 1

```

注意:

你可以假设 :

- $0 \leq \text{amount}$ (总金额) ≤ 5000
- $1 \leq \text{coin}$ (硬币面额) ≤ 5000
- 硬币种类不超过 500 种
- 结果符合 32 位符号整数

分析：同样是完全背包，但题目要求不一样，故定义 $dp[i]$ s为前*i*种硬币可组成和为s，共有多少种方式。。同样，假设不取*i*，则 $dp[i]$ s = $dp[i-1]$ s，若取*i*则 $dp[i]$ s = $dp[i]$ s-coins[i]]。故总数为二者相加

```

// 本题为空间复杂度优化版的，可先看0-1背包的优化过程
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        int len = coins.size();

```

```

vector<int> dp(amount+1, 0); // 由于dp取值只与本轮之前的值与上一轮当前值有关，故
一维即可

dp[0] = 1;
for(int i=0;i<len;i++)
{
    for(int j = coins[i];j<=amount;j++)
    {
        dp[j] = dp[j]+ dp[j-coins[i]]; // 左式dp[j]为i-1的
    }
}
return dp[amount];
}

};


```

377. 组合总和 IV

给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。

示例：

```

nums = [1, 2, 3]
target = 4

```

所有可能的组合为：

```

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

```

请注意，顺序不同的序列被视作不同的组合。

因此输出为 7。

进阶：

如果给定的数组中含有负数会怎么样？

问题会产生什么变化？

我们需要在题目中添加什么限制来允许负数的出现？

分析：第一眼看上去，该题是一个完全背包的问题。。但通过实例可知，该题实际上求的是排列个数，而完全背包问题求得是组合个数(即与数字的顺序无关)。我们可定义dp[i]为目标值为i的数有多少种排列。。

```

class Solution {
public:
    int combinationSum4(vector<int>& nums, int target) {
        int len = nums.size();

        vector<long> dp(target+1, 0);

        dp[0] = 1;
        for(int i=1;i<= target;i++) // 求排列的问题，则应该按此模式
        {

```

```

        for(int j = 0;j<len;j++)
        {
            if(nums[j] <= i)
                dp[i] = (dp[i] + dp[i-nums[j]])%INT_MAX; // dp[i-nums[j]] 意味
            着，目标值为i-nums[i]对应点组合数
        }
    }
    return dp[target];
}
};

```

139. 单词拆分

给定一个**非空**字符串 s 和一个包含**非空**单词列表的字典 $wordDict$ ，判定 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。

```

示例 3：

```

输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false

```

分析1：双指针加哈希表：哈希表的目的是加快查询速率，双指针则是找单词，但该思路并不完善

```

class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_map<string,int> hm;
        int len_s = s.length();
        int len_w = wordDict.size();

        // 将字典中元素放入哈希表
        for(int i=0;i<len_w;i++)
        {
            hm[wordDict[i]] = i;
        }

        // 双指针查询
        int slow = 0;

```

```

int fast = 0;
while(fast < len_s)
{
    if(hm.find(s.substr(slow,fast-slow+1)) != hm.end() ) slow = fast+1;

    fast++;
}
if(slow == len_s) return 1;
return 0;
};

}

```

分析2：通常，字符串或者数组元素进行处理时，也可采用动态规划。。例如本题设dp[i]为前i个元素是否可拆成字典中相关子字母。。对于第i个，需要重头判断，若在dp[j] 为true，则需判断j-> 是否也为字典中的单词，若是，说明dp[i]为true，若循环完也不存在，则dp[i]为false。。。可先看300题的思路

```

class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_map<string,int> hm;
        int len_s = s.length();
        int len_w = wordDict.size();

        // 将字典中元素放入哈希表
        for(int i=0;i<len_w;i++)
        {
            hm[wordDict[i]] = i;
        }

        vector<int> dp(len_s + 1,0);
        dp[0] = true; // 例如，dog，若不设其为1的话，则会显示dog为0
        // 设置两个循环，第一个循环用来判断前i个字符中是否有单词
        // 第二哥循环则是进一步判断。。不懂得可先看300题
        for(int i=1;i <= len_s;i++)
        {
            for(int j=i-1;j>=0;j--)
            {
                if(dp[j] && hm.find(s.substr(j,i-j)) != hm.end())
                {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[len_s];
    };
};

```

494. 目标和

给定一个非负整数数组， a_1, a_2, \dots, a_n ，和一个目标数， S 。现在你有两个符号 $+$ 和 $-$ 。对于数组中的任意一个整数，你都可以从 $+$ 或 $-$ 中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例：

输入： nums: [1, 1, 1, 1, 1], S: 3

输出： 5

解释：

-1+1+1+1+1 = 3

+1-1+1+1+1 = 3

+1+1-1+1+1 = 3

+1+1+1-1+1 = 3

+1+1+1+1-1 = 3

一共有5种方法让最终目标和为3。

提示：

- 数组非空，且长度不会超过 20。
- 初始的数组的和不会超过 1000。
- 保证返回的最终结果能被 32 位整数存下。

0819-回溯法

<https://www.bilibili.com/video/BV167411V7xn?from=search&seid=8133277288273728377>

问题1、素数环：

输入正整数n，把整数1,2,3,4.....n组成一个环，使得相邻的两个整数之和均为素数。输出是，从整数1开始逆时针排列，n<=16；

样例输入：

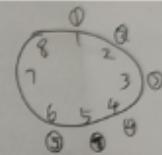
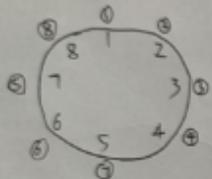
6

样例输出：

1 4 3 2 5 6

1 6 5 2 3 4

分析：该题若用枚举法，则需 $n!$ 次方，显然不适合。。该题求的是将n个数字放入n个位置，且要求相邻元素和为质数。可先对每个位置放置，然后判断是否符合质数的要求，，当将n个位置全放完后，若仍符合要求，则输出a(定义一个n个位置的数组)。直到最后完成，则回溯一个。。将最后两个



第一步：第1个位置为1，第2个位置为(4质).
一直到第5个位置，若放5($5+4=9$ 质)，
故不能放5，他不行，放7($7+4=11$)；然
后放第6个位置，(此时还未，5.6.8三1数)，
5.7.9.6.9，故放6($6+7=13$)。然后第7个位置，
放5；最后成8。
且 $8+1=9$ (合)，故不行，必需回溯。

第二步：回到7处，取出5， $5+1=6$ (合)不可，继续
回溯，到6处，取出6， $6+1=7$ (质)，~~故选6~~
~~故放5~~~~放5~~
此时放5 & 8($5+7=12, 8+7=15$ A) 24：继续回溯，
取出7.如此类推。

```
#include "pch.h"
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int cnt = 0; // 记录个数

// 输出n个位置的数字
void Display(int n, vector<int> &a)
{
    cnt++;
    for (int i = 1; i <= n; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
};

// 判断相邻元素是否为质数
bool IsZhishu(int m, int n)
{
    int sum = m + n;
    if (sum == 1) return 1;
    for (int i = 2; i <= sqrt(sum); i++)
    {
        if (sum % i == 0) return 0;
    }
    return 1;
}

// 深度优先遍历
void dfs(int location, int n, vector<int> &a, vector<int> &b)
{
    if (location == n + 1)
    {
        if (IsZhishu(a[n], a[1])) Display(n, a);
        return;
    }
    // 首先应该放第location个位置。。每个位置存放的数据都从1开始遍历，由于使用过的数字已标记，故不会重复
    for (int i = 1; i <= n; i++)
    {
        // a[i]==0说明没有被使用过...另，相邻元素为i与a[location-1]注意
        if (b[i] == 0 && IsZhishu(i, a[location - 1]))
        {
            b[i] = 1;
            a[location] = i;
            dfs(location + 1, n, a, b);
            b[i] = 0;
        }
    }
}
```

```

        b[i] = 1; // 表明该数被使用了
        a[location] = i; // 将第location的位置赋值
        dfs(location + 1, n, a, b);
        // 上述if else 只有将最后一个位置填完后，才会运行执行此步代码
        b[i] = 0; // 表明该数未被使用
    }
}
int main() {

    int n;
    cin >> n;

    vector<int> a(10 + 1, 0); // 用来存储数字
    vector<int> b(10 + 1, 0); // 用来判断该数字是否使用

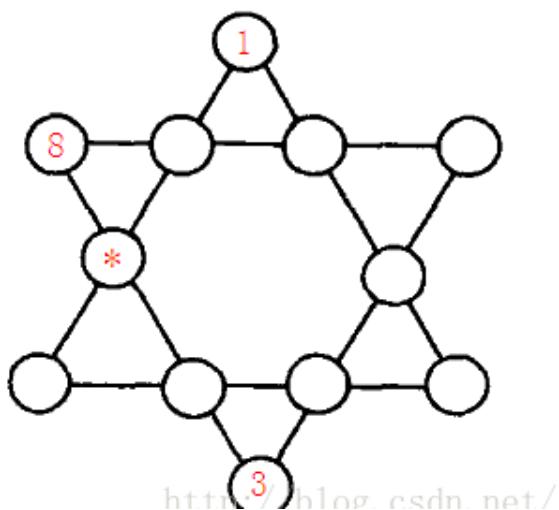
    dfs(1, n, a, b); // 放第一个位置
    cout << cnt; // 480
}

```

问题2、六角填数

如图【1.png】所示六角形中，填入1~12的数字。使得每条直线上的数字之和都相同。

图中，已经替你填好了3个数字，请你计算星号位置所代表的数字是多少？



<http://blog.csdn.net/>

```

#include "pch.h"
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void IsEqual(vector<int> &a)
{
    int i, b[6];

    b[0] = a[1] + a[3] + a[6] + a[8];
    b[1] = a[1] + a[4] + a[7] + a[11];
    b[2] = a[2] + a[3] + a[4] + a[5];
    b[3] = a[2] + a[6] + a[9] + a[12];
    b[4] = a[5] + a[7] + a[10] + a[12];
    b[5] = a[8] + a[9] + a[10] + a[11];

    for (i = 1; i < 6; i++)
        if (b[i] != b[i - 1])

```

```

        return;
    cout << a[6] << endl;
    return;
}

void dfs(int location, vector<int> &a, vector<int> &b, int n)
{
    // 若为1、2、12，说明已存在值，故直接跳到下一个节点
    if (location == 1 || location == 2 || location == 12)
    {
        dfs(location + 1, a, b, n);
    }

    // 若结束，则判断是否符合条件
    if (location == 13)
    {
        IsEqual(a);
        return;
    }
    // 若正常，则填值
    for (int i = 1; i <= 12; i++)
    {
        // 若该结点未被占用，则可填值
        if (b[i] == 0)
        {
            b[i] = 1;
            a[location] = i;

            dfs(location + 1, a, b, n);
            b[i] = 0;
        }
    }
}
}

int main()
{
    vector<int> a(13, 0); // 存节点的值
    vector<int> b(13, 0); // 用来判断某数是否被使用
    // 设定初始值
    a[1] = 1;
    a[2] = 8;
    a[12] = 3;
    b[1] = b[8] = b[3] = 1;

    int n = 12;
    dfs(1, a, b, n);
}

```

问题3、N皇后问题

N*N的棋盘中，不能存在同一行、同一列或同一对角线

```

#include "pch.h"
#include <iostream>
#include <algorithm>

```

```

#include <vector>
#include <unordered_map>
using namespace std;

int cnt = 0;

// 判断是否处在同一对角线
void Judge(vector<int> &a, int N)
{
    // 对角线即斜率为1/-1
    for (int i = 1; i < N; i++)
    {
        for (int j = i+1; j <= N; j++)
        {
            if (abs(a[i] - a[j]) == abs(i - j))
                return;
        }
    }
    cnt++;
}

void dfs(int location, vector<int> &a, vector<bool> &b, int N)
{
    if (location == N + 1)
    {
        Judge(a, N); // 判断是否在一条线上
        return;
    }

    // 放置第location的N皇后
    for (int i = 1; i <= N; i++)
    {
        if (b[i] == 0) // 保证不在同一行和同一列
        {
            b[i] = 1;
            a[location] = i;

            dfs(location + 1, a, b, N);
            b[i] = 0;
        }
    }
}

int main()
{
    int N;
    cin >> N; // 8

    vector<int> a(N + 1, 0); // 存放放置的位置（横坐标）
    vector<bool> b(N + 1, 0); // 存放某位置是否放置过

    dfs(1, a, b, N);
    cout << cnt; // 92
}

```

以上三个问题都是一种类型，，代码也有套路。。即将N个数填入N个位置，(数不可相同)。

0820-回溯

问题1、数字拆分：

任何一个大于1的自然数n总可以拆分为若干个小于n的自然数之和。

分析：1、完全背包

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    int N;
    cin >> N;

    vector<int> dp(N + 1, 0);
    dp[0] = 1;

    for (int i = 1; i < N; i++)
    {
        for (int j = i; j <= N; j++)
        {
            dp[j] = dp[j] + dp[j - i];
        }
    }
    cout << dp[N];
}
```

2、回溯

以7为例进行分析，7可分成7个1相加，即存在7个位置，每次往里面放数字。注：与上面不同的是，本题中数字可重复，即不需要再设定一个数组判定是否存在，，另，由于目标数字为7，，则需传递一个剩余数字的参数，当为0时，提前返回递归，无需进行后续操作。

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int cnt;

void display(vector<int>&a, int location)
{
    for (int i = 1; i <= location; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}

void dfs(int location, vector<int>&a, int N, int target)
{
```

```

if (target == 0)
{
    display(a, location - 1);
    cnt++;

    return;
}

if (target < 0)
    return;

for (int i = a[location-1]; i < N; i++) // 为避免重复，保证后面元素大于前面元素
{
    a[location] = i;
    dfs(location + 1, a, N, target - i);

    // 为了能体现出回溯,
    /*
    a[location] = i;
    target -= i;
    dfs(location + 1, a, N, target);
    target += i;
    */
}

int main()
{
    int N;
    cin >> N;

    vector<int> a(N + 1, 1); // 注: 1、某位置存放的数字（初始化为1是为了上面的dfs）

    dfs(1, a, N, N);
    cout << cnt;
}

```

进一步地，若要考虑排序，则可得

```

#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
// 动态规划
int main()
{
    int N;
    cin >> N; // 7

    vector<int> dp(N + 1, 0);

    dp[0] = 1;

    for (int i = 1; i <= N; i++)
    {
        for (int j = 0; j < N; j++) // 不能为N+0，故小于N

```

```

    {
        if(j <= i)
            dp[i] = dp[i] + dp[i - j];
    }
    cout << dp[N]; // 63
}

```

```

#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int cnt;

void display(vector<int>&a, int location)
{
    for (int i = 1; i <= location; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}

void dfs(int location, vector<int>&a, int N, int target)
{
    if (target == 0)
    {
        display(a, location - 1);
        cnt++;

        return;
    }

    if (target < 0)
        return;

    // 排序与不排序的差别仅在于，填坑用的值，此处从1开始，而排序的话应大于之前元素
    for (int i = 1; i < N; i++) // 为避免重复，保证后面元素大于前面元素
    {

        a[location] = i;
        dfs(location + 1, a, N, target - i);

    }
}

int main()
{
    int N;
    cin >> N;

    vector<int> a(N + 1, 1); // 注：1、某位置存放的数字（初始化为1是为了上面的dfs）

    dfs(1, a, N, N);
    cout << cnt;
}

```

问题2、排列组合

输出自然数1-n所有不重复的排列，即n的全排列，要求所产生的任一数字序列中不允许出现的重复的数字。

分析：可假设n个位置，然后使用回溯。注，要求不出现重复数字，故需使用一个数组来判断是否使用。每当location到了N+1的时候，输出即可。。此题更像是素数环之类的题（但更简单，因为没有限制条件）。

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int cnt = 0;

void display(vector<int> &a, int n)
{
    for (int i = 1; i <= n; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}

void dfs(int location, vector<int> &a, vector<int> &b, int n)
{
    if (location == n + 1)
    {
        display(a, n);
        cnt++;
        return;
    }

    for (int i = 1; i <= n; i++)
    {
        if (b[i] == 0)
        {
            b[i] = 1;
            a[location] = i;

            dfs(location + 1, a, b, n);
            b[i] = 0;
        }
    }
}

int main()
{
    int n;
    cin >> n;

    vector<int> a(n + 1, 0); // 存放每个位置的数字
    vector<int> b(n + 1, 0); // 判断某数字是否使用

    dfs(1, a, b, n);
    cout << cnt << endl;
}
```

问题3、排列组合进阶版

以5为例，分别求其中3个数的排列和组合。。

对于排列来说，只需修改一下location的大小即可

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int cnt = 0;

void display(vector<int> &a, int n)
{
    for (int i = 1; i <= n; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}

void dfs(int location, vector<int> &a, vector<int> &b, int n, int m)
{
    if (location == m + 1) // 取n的m个排列，只需放在m个位置即可，故m+1时，需输出
    {
        display(a, n);
        cnt++;
        return;
    }

    for (int i = 1; i <= n; i++)
    {
        if (b[i] == 0)
        {
            b[i] = 1;
            a[location] = i;

            dfs(location + 1, a, b, n, m);
            b[i] = 0;
        }
    }
}

int main()
{
    int n;
    cin >> n;

    int m;
    cin >> m;
    vector<int> a(n + 1, 0); // 存放每个位置的数字
    vector<int> b(n + 1, 0); // 判断某数字是否使用

    dfs(1, a, b, n, m);
    cout << cnt << endl;
}
```

对于组合，可参考问题1中的组合问题，即后面的元素需要小于前面的元素。

```
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>
```

```

using namespace std;

int cnt = 0;

void display(vector<int> &a, int n)
{
    for (int i = 1; i <= n; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}

void dfs(int location, vector<int> &a, vector<int> &b, int n, int m)
{
    if (location == m + 1) // 取n的m个元素，只需放在m个位置即可，故m+1时，需输出
    {
        display(a, n);
        cnt++;
        return;
    }

    for (int i = 1; i <= n; i++)
    {
        if (b[i] == 0 && i > a[location-1]) // 1、排列与组合的差别就在于此
        {
            b[i] = 1;
            a[location] = i;

            dfs(location + 1, a, b, n, m);
            b[i] = 0;
        }
    }
}
int main()
{
    int n;
    cin >> n;

    int m;
    cin >> m;
    vector<int> a(n + 1, 0); // 存放每个位置的数字
    vector<int> b(n + 1, 0); // 判断某数字是否使用

    dfs(1, a, b, n, m);
    cout << cnt << endl;
}

```

剑指 Offer 12. 矩阵中的路径

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。

示例 1：

```
输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"  
输出：true
```

示例 2：

```
输入：board = [["a","b"],["c","d"]], word = "abcd"  
输出：false
```

提示：

- $1 \leq \text{board.length} \leq 200$
- $1 \leq \text{board[i].length} \leq 200$

分析：题中是要求某一路径符合要求，由此可知可采用dfs+回溯求各路径，然后如果达到要求，直接return true即可。

但与之前回溯不一样的是，该题是一个二维图。

```
#include "pch.h"  
#include <iostream>  
#include <vector>  
#include <algorithm>  
  
using namespace std;  
/*  
 i、j:当前所在位置  
 row、line:总共的行列是数  
 used:判断是否被使用  
 c:字符处于word中的位置  
 */  
  
bool dfs(int i, int j, int row, int line, vector<vector<bool>> &used,  
 vector<vector<char>>& board, string word, int c)  
{  
     // 边界:注意，下面的边界是有先后顺序的。  
     if (i < 0 || i >= row || j < 0 || j >= line || used[i][j] == true)  
         return false;  
  
     // 不相等  
     if (board[i][j] != word[c])  
         return false;  
  
     if (c == word.length() - 1)  
         return true;  
  
     used[i][j] = true;
```

```

        bool flag = dfs(i + 1, j, row, line, used, board, word, c + 1) ||
        dfs(i, j + 1, row, line, used, board, word, c + 1) ||
        dfs(i - 1, j, row, line, used, board, word, c + 1) ||
        dfs(i, j - 1, row, line, used, board, word, c + 1);

    used[i][j] = false;

    return flag;
};

// 遍历起始点，然后从起始点开始前后左右的移动
class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        int row = board.size();
        int line = board[0].size();

        vector<vector<bool>> used(row, vector<bool>(line, false));

        bool flag = dfs(i, j, row, line, used, board, word, 0);
        if (flag) return true;
        return false;
    }
};

int main()
{
    vector<vector<char>> board = {{'A', 'B', 'C', 'E'}, {'S', 'F', 'C', 'S'}, {'A', 'D', 'E', 'E'}};
    string word = "ABCED";
    Solution s;
    cout << s.exist(board, word);
    return 0;
}

```

```

#include "pch.h"
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>
using namespace std;
class Solution {
public:

    bool dfs(vector<vector<char>>& board, int p, int q, string word, int index,
    vector<vector<bool>>& used)
    {
        if (index >= word.size())
            return 1;

        if (p >= board.size() || q >= board[0].size() || used[p][q] || board[p][q] != word[index])
            return 0;

        used[p][q] = 1;
        bool flag = dfs(board, p + 1, q, word, index + 1, used)
                    || dfs(board, p, q + 1, word, index + 1, used)
                    || dfs(board, p - 1, q, word, index + 1, used)
                    || dfs(board, p, q - 1, word, index + 1, used);

        return flag;
    }
};

```

```

        || dfs(board, p - 1, q, word, index + 1, used)
        || dfs(board, p, q - 1, word, index + 1, used);
    used[p][q] = 0;
    return flag;
}
bool exist(vector<vector<char>>& board, string word) {
    int col = board.size();
    int line = board[0].size();
    vector<vector<bool>> used(col, vector<bool>(line, 0));
    for (int i = 0; i < col; i++)
    {
        for (int j = 0; j < line; j++)
        {
            bool flag = dfs(board, i, j, word, 0, used);
            if (flag) return 1;
        }
    }
    return 0;
};

int main()
{
    return 0;
}

```

剑指 Offer 13. 机器人的运动范围

地上有一个 $m \times n$ 列的方格，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 $[3,5,3,7]$ ，因为 $3+5+3+7=18$ 。但它不能进入方格 $[3,5,3,8]$ ，因为 $3+5+3+8=19$ 。请问该机器人能够到达多少个格子？

示例 1：

```

输入：m = 2, n = 3, k = 1
输出：3

```

示例 2：

```

输入：m = 3, n = 1, k = 0
输出：1

```

提示：

- $1 \leq n, m \leq 100$
- $0 \leq k \leq 20$

分析：本题也是一个二维图，其实在12题的基础上，本题就不难解决。差别在于，每次换一个位置均需判断这个位置是否符合要求。且本题不需要回溯，因为要计算符合要求的位置

```

class Solution {
public:
    bool isFit(int i, int j, int k) // 若满足则返回1，不然返回0

```

```

{
    int sum = 0;
    for (; i; i = i / 10)
    {
        sum += i % 10;
    }

    for (; j; j = j / 10)
    {
        sum += j % 10;
    }

    if (sum <= k) return 1;
    return 0;
}

/*
    i、j:当前所在位置
    used:判断是否被使用
*/
int dfs(int i, int j,int m,int n,int k, vector<vector<int> >&used)
{
    if (i < 0 || j < 0 || i >= m || j >= n || used[i][j] || isFit(i, j, k) == 0)
return 0;

    used[i][j] = 1;
    cnt = dfs(i + 1, j, m, n, k, used)+ dfs(i, j+1, m, n, k, used)+1;
    return cnt;
};

public:
    int movingCount(int m, int n, int k) {

        vector<vector<int> > used(m, vector<int>(n, 0));
        return dfs(0, 0,m,n,k,used);
    }
private:
    int cnt = 0;
};

```

200. 岛屿数量

给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

```

#include "pch.h"
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm>
using namespace std;

class Solution {
public:

    bool dfs(vector<vector<char> > &grid, int p, int q)

```

```

    {
        if (p < 0 || q < 0 || p >= grid.size() || q >= grid[0].size() || grid[p][q] != '1')
            return 0;

        grid[p][q] = '0';
        dfs(grid, p + 1, q);
        dfs(grid, p - 1, q);
        dfs(grid, p, q + 1);
        dfs(grid, p, q - 1);
        return 1;
    }

    int numIslands(vector<vector<char> >& grid) {
        int res = 0;
        for (int i = 0; i < grid.size(); i++)
        {
            for (int j = 0; j < grid[0].size(); j++)
            {
                bool flag = dfs(grid, i, j);
                if (flag)
                    res++;
            }
        }
        return res;
    }
};

int main()
{
    vector <vector<char> >grid = {
        {'1', '1', '0', '0', '0'},
        {'1', '1', '0', '0', '0'},
        {'0', '0', '1', '0', '0'},
        {'0', '0', '0', '1', '1'}
    };
    Solution s;
    cout<<s.numIslands(grid);

    return 0;
}

```

0821-字符串加乘

43. 字符串相乘

给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

示例 1:

输入: num1 = "2", num2 = "3"
输出: "6"

示例 2:

输入: num1 = "123", num2 = "456"
输出: "56088"

说明 :

1. num1 和 num2 的长度小于110。
2. num1 和 num2 只包含数字 0-9。
3. num1 和 num2 均不以零开头，除非是数字 0 本身。
4. 不能使用任何标准库的大数类型（比如 BigInteger）或直接将输入转换为整数来处理。

剑指 Offer 43. 1 ~ n整数中1出现的次数

输入一个整数 n , 求1 ~ n这n个整数的十进制表示中1出现的次数。

例如 , 输入12 , 1 ~ 12这些整数中包含1 的数字有1、10、11和12 , 1一共出现了5次。

示例 1:

输入: n = 12
输出: 5

示例 2:

输入: n = 13
输出: 6

限制 :

- $1 \leq n < 2^{31}$

```
class Solution {
public:

    int getLength(int n)
    {
        int len = 0;
        for(;n;n = n/10)
        {
            len++;
        }
        return len;
    }

    int countDigitOne(int n) {
        // 找规律的题
        /*
         * 将数分成三部分：当前位、低位、高位
        */
    }
}
```

```

        另外还需要一个数，用来表示当前位所处的位置（个位的话为1，十位的话为10，百位则为
100）
        当当前位数为0时，则当前位总共出现1的次数为：高位*位数（20的话，个位出现1的次数为
11， $1.2=2*1$ ）
        当当前位数为1时，则当前位总共出现1的次数为：高位*位数+低位+1（112时，112、111、
110、19-10 故为13）
        当当前位数为2-9时，则当前位总共出现1的次数为：（高位+1）*位数
    */

    // 首先求出位数
    int len = getLength(n);
    int cur = n % 10;
    int low = 0;
    int high = n/10;
    int cnt = 0;
    for(int i=0;i<len;i++)
    {
        if(cur == 0) cnt += high * pow(10.0,i);
        else if(cur == 1) cnt += high * pow(10.0,i) + low + 1;
        else cnt += (high+1)*pow(10.0,i);

        low += cur*pow(10.0,i);
        cur = high % 10;
        high = high /10;
    }
    return cnt;
}
};

```

0822—前缀和

面试题 17.24. 最大子矩阵

给定一个正整数和负整数组成的 $N \times M$ 矩阵，编写代码找出元素总和最大的子矩阵。返回一个数组 $[r1, c1, r2, c2]$ ，其中 $r1, c1$ 分别代表子矩阵左上角的行号和列号， $r2, c2$ 分别代表右下角的行号和列号。若多个满足条件的子矩阵，返回任意一个均可。

示例：

输入： [[-1,0], [0,-1]]	输出： [0,1,0,1] 解释： 输入中粗黑的元素即为输出所表示的矩阵
---	---

说明：

- $1 \leq \text{matrix.length}, \text{matrix}[0].length \leq 200$

分析：若是一行数据，然后求最长连续字串的最大和，假设前*i*个元素的最大和为dp[i]，dp[i]的状态方程为，若dp[i-1]<0，则dp[i] = num[i]，日票dp[i-1]>0，则dp[i] = dp[i-1]+num[i]。。。那么本题其实可以将各连续行组合一下，然后再按照一维进行计算。

```
#include "pch.h"
#include <iostream>
#include <string>
#include <unordered_map>
#include<vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    vector<int> getMaxMatrix(vector<vector<int>>& matrix) {

        int col = matrix.size();
        int line = matrix[0].size();
        vector<vector<int>> sum_num(col, vector<int>(line, 0));
        // 计算每层的前缀和
        for (int j = 0; j < line; j++)
        {
            sum_num[j][0] = matrix[j][0];
        }
        for (int i = 0; i < col; i++)
        {
            for (int j = 1; j < line; j++)
            {
                sum_num[i][j] = sum_num[i][j - 1] + matrix[i][j];
            }
        }

        // 将二维转一维，再利用动规计算
        int left_x = 0;
        int left_y = 0;
        int right_x = 0;
        int right_y = 0;
        int max_sum = 0;
        for (int i = 0; i < col; i++) // 起始行
        {
            vector<int> sum(line, 0);
            for (int j = i; j < line; j++) // 终止行
            {
                // 动规
                int su = 0;
                for (int ii = 0; ii < line; ii++)
                {
                    sum[ii] += matrix[j][ii]; // 加上最新行

                    if (su > 0)
                    {
                        su += sum[ii];
                    }
                    else // 若之前为负，则可直接重新开始
                    {

                        su = sum[ii];
                        left_x = ii; // 确定了右下坐标
                        left_y = j;
                    }
                }
            }
        }
    }
}
```

```
        if (su > max_sum)
        {
            max_sum = su;
            right_x = ii; // 确定了右下坐标
            right_y = j;
        }
    }
};

int main()
{
}
```