第3章 穷举法

3.1 LintCode1068——寻找数组的中心索引★

问题描述：给定一个整数数组 nums，设计一个算法返回此数组的“中心索引”。中心索引定义为这样的元素，该元素左边的数字之和等于其右边的数字之和。如果不存在这样的中心索引，返回－1；如果有多个中心索引，返回最左侧的那一个。例如，nums＝{1，7，3，6，5，6}，答案是 3，因为 nums[3]元素的左边和右边的数字之和均为 11。要求设计如下成员函数：

```
int pivotIndex(vector < int > &nums) { }
```

解法 1：采用直接穷举法，对于每个元素 nums[i]，求左边元素的和 left 以及右边元素的和 right，若 left＝right，则返回 i。最后返回－1。对应的程序如下：

```
class Solution {
public:
  int pivotIndex(vector < int > &nums) {
      int n = nums.size();
      int left,right;
      for (int i = 0;i < n;i++) {
         left = 0;
         for(int j = 0;j < i;j++)
            left += nums[j];
         right = 0;
         for(int j = i + 1;j < n;j++)
            right += nums[j];
         if (right == left)
            return i;
      }
      return - 1;
  }
};
```

上述算法的时间复杂度为 $O(n^2)$，程序提交时超时(time limit exceeded)。

解法 2：在枚举 nums[i]时，左边元素的和 left 是依次求出的，例如求出 nums[$i-1$]的 left，则 nums[i]的 left 等于 left＋nums[$i-1$]。另外可以事先求出全部元素的和 sum，当 nums[i]对应的左边元素的和为 left 时，则右边元素的和等于 sum－left－nums[i]，这样算法的时间复杂度降为 $O(n)$。对应的程序如下：

```
#include < numeric >
class Solution {
public:
  int pivotIndex(vector < int > &nums) {
      int n = nums.size();
      int sum = accumulate(nums.begin(),nums.end(),0);
      int left = 0,right;
      for (int i = 0;i < n;i++) {
         if (i!= 0)
            left += nums[i - 1];
         right = sum - left - nums[i];
         if (right == left)
            return i;
```

```
        }
        return -1;
    }
};
```

上述程序提交后通过，执行用时为 81ms，内存消耗为 5.37MB。

3.2 LintCode1517——最大子数组★

问题描述：给定一个由 n 个整数构成的数组 A 和一个整数 $K(1\leqslant K\leqslant n\leqslant 100)$，设计一个算法从所有长度为 K 的 A 的连续子数组中返回最大的连续子数组，如果两个数组中第一个不相等的元素在 A 中的值大于 B 中的值，则定义子数组 A 大于子数组 B。例如，$A=\{1,2,4,3\}$，$B=\{1,2,3,5\}$，则 A 大于 B，因为 $A[2]>B[2]$。要求设计如下成员函数：

```
vector<int> largestSubarray(vector<int> &A, int K) {
```

解法 1：采用穷举法，首先取 A 的前 K 个元素存放在 ans 中作为答案，然后直接枚举每个子数组 $A[i..j]$，若其长度等于 K，则与 ans 比较大小将较大者存放在 ans 中，最后返回 ans 即可。对应的程序如下：

```
class Solution {
public:
  vector<int> largestSubarray(vector<int> &A, int K) {
    int n=A.size();
    if(n<K) return {};
    vector<int> ans;
    ans=vector<int>(A.begin(),A.begin()+K);
    for(int i=0;i<n;i++) {
       for(int j=i;j<n;j++) {
          if(j-i+1==K) {
             vector<int> cura=vector<int>(A.begin()+i,A.begin()+j+1);
             if(cura>ans) ans=cura;
          }
       }
    }
    return ans;
  }
};
```

上述程序提交后通过，执行用时为 41ms，内存消耗为 5.53MB。

解法 2：由于这里的比较对象是长度为 K 的子数组，而子数组的 K 个元素是相邻的，所以改为用 i 直接枚举子数组 $A[i..i+K-1]$即可，这样算法的时间复杂度降为 $O(n)$。对应的程序如下：

```
class Solution {
public:
  vector<int> largestSubarray(vector<int> &A, int K) {
    int n=A.size();
    if(n<K) return {};
    vector<int> ans;
    ans=vector<int>(A.begin(),A.begin()+K);
    for(int i=0;i<n-K+1;i++) {
       vector<int> cura=vector<int>(A.begin()+i,A.begin()+i+K);
       if(cura>ans) ans=cura;
```

```
        }
        return ans;
    }
};
```

上述程序提交后通过，执行用时为 41ms，内存消耗为 4.76MB。

3.3 LintCode1338——停车困境★

问题描述：停车位是一条很长的直线，每米都有一个停车位，停车场里停着 $n(1\leqslant n\leqslant 10^5)$ 辆小车，所有车位上面的车都是唯一的，用 cars 表示。现在要通过建造遮雨棚来遮雨、挡雨，要求至少有 $k(1\leqslant k\leqslant n)$ 辆小车的车顶被遮雨棚遮盖，设计一个算法求遮雨棚遮盖 k 辆小车的车顶需要的最小长度。例如，cars={2,10,8,17}，$k=3$，答案是 9，即建造长度为 9 的遮雨棚，遮盖从第 2 个到第 10 个的所有停车位。要求设计如下成员函数：

```
int ParkingDilemma(vector<int> &cars, int k) { }
```

解：如果直接采用穷举法会超时，改进方法是将 cars 递增排序，然后将每 k 个元素看成一个遮雨棚(恰好遮盖 k 辆小车)，求出其长度，通过比较求最小长度 ans。例如，cars={2,10,8,17}、$k=3$ 时求遮雨棚的最小长度如图 3.1 所示。对应的程序如下：

```
class Solution {
  const int INF = 0x3f3f3f3f;
public:
  int ParkingDilemma(vector<int> &cars, int k) {
      sort(cars.begin(),cars.end());
      int ans = INF;
      for (int i = 0;i < cars.size() - k + 1;i++) {
          ans = min(ans,cars[i + k - 1] - cars[i] + 1);
      }
      return ans;
  }
};
```

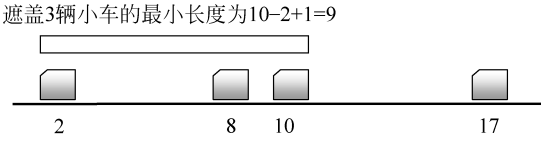

图 3.1 $k=3$ 时遮雨棚的最小长度

上述程序提交后通过，执行用时为 41ms，内存消耗为 5.44MB。

3.4 LintCode993——数组划分 I★

问题描述：给定一个有 $2n$ 个整数的数组 nums，设计一个算法把这些整数分成 n 组，例如(a_1,b_1)、(a_2,b_2)、……、(a_n,b_n)，并且使 i 从 1 到 n 的 $\min(a_i,b_i)$之和尽可能大。其中 n 是一个范围为[1,10 000]的正整数，数组中元素的范围为[−10 000,10 000]。例如，nums={1,4,3,2}，答案是 4，这里 $n=2$，数组中的元素分为数对(1,2)和(3,4)。要求设计如下成员函数：

```
int arrayPairSum(vector<int> &nums) { }
```

解：如果直接采用穷举法会超时，改进方法是将数组 nums 递增排序，取出所有下标为偶数的元素求和。例如，nums={1,4,3,2}，排序后 nums={1,2,3,4}，依次两两一组的结果满足题目要求，每组的最小值即该组的前一个元素，取出 1 和 3 并且累计，结果为 4。对应的程序如下：

```
class Solution {
public:
  int arrayPairSum(vector<int> &nums) {
     sort(nums.begin(),nums.end());
     int ans = 0;
     for (int i = 0;i < nums.size();i += 2)
        ans += nums[i];
     return ans;
  }
};
```

上述程序提交后通过，执行用时为 61ms，内存消耗为 5.48MB。

3.5 LintCode406——和大于 s 的最小子数组★★

问题描述：给定一个由 n 个正整数组成的数组和一个正整数 s，设计一个算法求满足其和大于或等于 s 的最小长度的子数组，如果无解则返回 -1。例如，nums={2,3,1,2,4,3}，$s=7$，答案是 2，因为子数组{4,3}是该条件下的最小长度子数组。要求设计如下成员函数：

```
int minimumSize(vector<int> &nums, int s) { }
```

解法 1：如果直接枚举 nums[i..j]，求出元素和 sum，当 sum$\geqslant s$ 时求最小长度 $j-i+1$，时间复杂度为 $O(n^3)$，一定会出现超时。改为用前缀和数组，设 psum[i]为 nums 中前 $i+1$ 个元素的和，在求出 psum 数组后，有 nums[$i+1$..j]的元素和等于 psum[j]$-$psum[i]，这样通过枚举 i 和 j 求出元素和大于或等于 s 的最小子数组长度 $j-i$。对应的程序如下：

```
class Solution {
public:
  int minimumSize(vector<int> &nums, int s) {
     int n = nums.size();
     if(n == 0) return -1;
     int psum[n];
     psum[0] = nums[0];
     for(int i = 1;i < n;i++)
        psum[i] = psum[i-1] + nums[i];
     int ans = n + 1;                          //存放答案,取最大值
     for(int i = -1;i < n-1;i++) {
        for(int j = i;j < n;j++) {
           int sum = psum[j] - psum[i];        //求 nums[i+1..j]的元素之和
           if(sum >= s) ans = min(ans,j-i);
        }
     }
     if(ans == n + 1)return -1;
     else return ans;
  }
};
```

上述算法的时间复杂度为$O(n^2)$，程序提交时仍然超时(time limit exceeded)。

解法2：用i、j两个指针表示一个窗口为nums[$i-1..j-1$](i、j均从0开始表示初始窗口为空)，先保持i不变通过后移j找到元素和大于或等于s的窗口nums[$i..j$]，若不存在这样的窗口则结束，找到这样的窗口后i后移缩小窗口，但始终保证窗口的元素和不小于s，在所有这样的窗口中求最小长度。例如，nums={2,3,1,2,4,3}，$s=7$，首先置ans=7，$i=0$，$j=0$，求解过程如下：

(1) j后移到等于4，找到sum为2+3+1+2=8，此时sum≥s，再后移i同时从sum中减去nums[i]直到sum<s，此时$i=1$(sum为8-2=6)，说明nums[$i-1..j-1$]即nums[0..3]是一个满足要求的窗口，长度为4，置ans=4。

(2) $i=1$，$j=4$，sum=6，后移到$j=5$，找到sum为6+4=10，此时sum≥s，再后移i同时从sum中减去nums[i]直到sum<s，此时$i=3$(sum为10-3-1=6)，说明nums[$i-1..j-1$]即nums[2..4]是一个满足要求的窗口，长度为3，置ans=3。

(3) $i=3$，$j=5$，sum=6，后移到$j=6$，找到sum为6+3=9，此时sum≥s，再后移i同时从sum中减去nums[i]直到sum<s，此时$i=5$(sum为9-2-4=3)，说明nums[$i-1..j-1$]即nums[4..5]是一个满足要求的窗口，长度为2，置ans=2。

最终结果是ans=2，返回2。对应的程序如下：

```
class Solution {
public:
  int minimumSize(vector<int> &nums, int s) {
     int n = nums.size();
     int ans = n + 1;                          //存放答案,取最大值
     int i = 0, j = 0;
     int sum = 0;
     while (j < n) {
        while (j < n && sum < s)               //扩大窗口
           sum += nums[j++];
        if (sum < s) break;
        while (i < j && sum >= s)              //缩小窗口
           sum -= nums[i++];
        ans = min(ans, j - i + 1);
     }
     if (ans == n + 1)return -1;
     else return ans;
  }
};
```

在上述算法中i、j指针均后移，时间复杂度为$O(n)$。上述程序提交后通过，执行用时为163ms，内存消耗为20.55MB。

3.6 LintCode1331——英语软件★

问题描述：小林是班级的英语课代表，他想开发一款软件处理班上同学的成绩(所有成绩在0～100的范围内)。小林的软件有一个神奇的功能，能够通过一个百分数来反映各同学的成绩在班上的位置，即“成绩超过班级多少百分比的同学”。设这个百分比为p，某同学考了s分，则可以通过p=(分数不超过s的人数-1)/班级总人数×100%计算出p。请帮助小林设计一下这个软件。其中score数组表示所有同学的成绩，score[i]表示第一个同学

的成绩，ask 数组中的 ask[i]表示询问 ask[i]个人的成绩百分比，每询问一次输出对应的成绩百分比，不需要输出百分号，答案向下取整（为避免精度丢失，优先计算乘法）。例如，score＝{100，98，87}，ask＝{1，2，3}，表示共有 3 个人，第一个人到第三个人的成绩分别是 100、98 和 97，要求求出第一个人到第三个人的成绩百分比，答案是{66，33，0}，即第一个人的成绩为 100，超过了 66%的学生。要求设计如下成员函数：

```
vector < int > englishSoftware(vector < int > &score, vector < int > &ask) { }
```

解：这里采用前缀和数组 psum，先遍历 score 求出成绩为 s 的人数 psum[s]，再通过遍历 psum 并计算 psum[i]＋＝psum[i－1]求出分数小于或等于 s 的人数 psum[s]。这样对于序号 ask[i]，对应的成绩百分比为 rnk＝(psum[score[ask[i]－1]]－1)×100/n，将其添加到 ans 中，最后返回 ans 即可。对应的程序如下：

```
class Solution {
public:
  vector < int > englishSoftware(vector < int > &score, vector < int > &ask) {
      int n = score.size();
      vector < int > psum(101,0);
      for(int s:score)
          psum[s]++;
      for(int i = 1;i < 101;i++)
          psum[i] += psum[i - 1];
      vector < int > ans;
      for(int i = 0;i < ask.size();i++) {
          int rnk = (psum[score[ask[i] - 1]] - 1) * 100/n;
          ans.push_back(rnk);
      }
      return ans;
  }
};
```

上述程序提交后通过，执行用时为 41ms，内存消耗为 2.29MB。

3.7 LintCode397——最长上升连续子序列★

问题描述：给定一个长度为 n 的整数数组 A，其中所有元素是唯一的，设计一个算法找出该数组中的最长上升连续子序列(LICS)，这里的最长上升连续子序列可以定义为从右到左或从左到右的序列。例如，A＝{5，4，2，1，3}，答案是 4，其最长上升连续子序列为{5，4，2，1}；A＝{5，1，2，3，4}，答案是 4，其最长上升连续子序列为{1，2，3，4}。要求设计如下成员函数：

```
int longestIncreasingContinuousSubsequence(vector < int > &A) { }
```

解：用 ans 存放答案(初始为 0)，用 incnt 和 decnt 分别表示递增和递减连续子序列的长度(初始均为 1)，用 i 从 1 开始遍历 A，若 A[i－1]＜A[i]，则执行 incnt++和 decnt＝1，否则执行 decnt++和 incnt＝1，然后在 ans、incnt 和 decnt 中取最大值存放到 ans 中。遍历完毕返回 ans 即可。对应的程序如下：

```
class Solution {
public:
  int longestIncreasingContinuousSubsequence(vector < int > &A) {
```

```
        int n = A.size();
        if (n <= 2) return n;
        int incnt = 1;
        int decnt = 1;
        int ans = 0;
        for (int i = 1;i < n;i++) {
            if (A[i - 1]< A[i]) {
                incnt++;
                decnt = 1;
            }
            else if (A[i - 1]> A[i]) {
                decnt++;
                incnt = 1;
            }
            ans = max(ans, decnt);
            ans = max(ans, incnt);
        }
        return ans;
    }
};
```

上述程序提交后通过,执行用时为 41ms,内存消耗为 4.3MB。

3.8 LeetCode1534——统计好三元组★

问题描述:给定一个含 $n(3\leqslant n\leqslant 100)$个整数的数组 arr,以及 a、b、c 三个整数,设计一个算法求其中好三元组的数量。如果三元组(arr[i],arr[j],arr[k])满足下列全部条件,则认为它是一个好三元组:

(1) $0\leqslant i<j<k<n$。

(2) $|arr[i]-arr[j]|\leqslant a$,$|arr[j]-arr[k]|\leqslant b$,$|arr[i]-arr[k]|\leqslant c$。

例如,arr={3,0,1,1,9,7},$a=7$,$b=2$,$c=3$,答案是 4,一共有 4 个好三元组,即(3,0,1)、(3,0,1)、(3,1,1)和(0,1,1)。要求设计如下成员函数:

```
int countGoodTriplets(vector < int > & arr, int a, int b, int c) { }
```

解:直接采用简单穷举法,枚举全部情况,累计满足题目要求的好三元组的数量 ans,最后返回 ans 即可。对应的程序如下:

```
class Solution {
public:
  int countGoodTriplets(vector < int > & arr, int a, int b, int c) {
      int n = arr.size();
      int ans = 0;
      for(int i = 0;i < n - 2;i++) {
          for(int j = i + 1;j < n - 1;j++) {
              for(int k = j + 1;k < n;k++) {
                  if(abs(arr[i] - arr[j])< = a && abs(arr[j] - arr[k])< = b && abs(arr[i] - arr[k])< = c)
                      ans++;
              }
          }
      }
      return ans;
  }
};
```

上述程序提交后通过，执行用时为 48ms，内存消耗为 8MB。

3.9 LeetCode204——计数质数★★

问题描述：给定一个整数 $n(0\leqslant n\leqslant 5\times 10^6)$，设计一个算法求所有小于非负整数 n 的质数的个数。例如，$n=10$，答案是 4，小于 10 的质数是 2、3、5、7。要求设计如下成员函数：

```
int countPrimes(int n) { }
```

解：如果直接采用穷举法判断 1～n 的每个整数是否为质数，并累计其中质数的个数，这样一定会超时。这里采用质数筛选法，对于质数 i，则 i 的 i 倍整数、$i+1$ 倍整数等一定不是质数，再累计其中质数的个数 ans，最后返回 ans 即可。对应的程序如下：

```
class Solution {
public:
  int countPrimes(int n) {
     bool prime[n + 1];                          //prime[i]表示整数 i 是否为质数
     memset(prime,true,sizeof(prime));           //预设均为质数
     for(int i = 2;i * i < n;i++) {
        if(prime[i]) {                           //i 是质数,置其倍数为非质数
           for(int j = i * i;j < n;j += i) prime[j] = false;
        }
     }
     int ans = 0;
     for(int i = 2;i < n;i++) {                  //累计质数的个数
        if(prime[i]) ans++;
     }
     return ans;
  }
};
```

上述程序提交后通过，执行用时为 148ms，内存消耗为 11.3MB。

3.10 LeetCode187——重复的 DNA 序列★★

问题描述：DNA 序列由一系列核苷酸组成，给定一个表示 DNA 序列的字符串 s，设计一个算法返回在 DNA 分子中出现不止一次的长度为 10 的所有序列（子字符串），可以按任意顺序返回。例如，输入 s ="AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"，答案是{"AAAAACCCCC","CCCCCAAAAA"}。要求设计如下成员函数：

```
vector < string > findRepeatedDnaSequences(string s) { }
```

解：采用简单穷举法，用 unordered_map < string，int >类型的哈希表 cntmap 实现子字符串的计数，用 ans 存放答案。i 从 0 开始遍历 s，提取当前长度为 10 的子串 ss，累计其个数，若个数为 2 将其添加到 ans 中。遍历完毕返回 ans 即可。对应的程序如下：

```
class Solution {
public:
  vector < string > findRepeatedDnaSequences(string s) {
     vector < string > ans;
```

```
        unordered_map<string,int> cntmap;
        int n=s.size();
        if(n<10) return ans;
        for(int i=0;i<n-9;i++) {
            string ss=s.substr(i,10);
            cntmap[ss]++;
            if(cntmap[ss]==2)
                ans.push_back(ss);
        }
        return ans;
    }
};
```

上述程序提交后通过,执行用时为 52ms,内存消耗为 22.9MB。

3.11 LeetCode2018——判断单词是否能放入填字游戏内★★

问题描述:给定一个 $m\times n$ 的矩阵 board,它代表一个填字游戏的当前状态。在填字游戏的格子中包含小写英文字母(已填入的单词)、表示空格的" "和表示障碍格子的"#"。如果满足以下条件,则可以水平(从左到右或者从右到左)或竖直(从上到下或者从下到上)填入一个单词:

(1) 该单词不占据任何'#'对应的格子。

(2) 每个字母对应的格子要么是" "(空格)要么与 board 中已有字母的匹配。

(3) 若单词是水平放置,则该单词左边和右边相邻的格子(可以是边缘)不能为" "或小写英文字母。

(4) 若单词是竖直放置,则该单词上边和下边相邻的格子(可以是边缘)不能为" "或小写英文字母。

给定一个字符串 word,设计一个算法当 word 可以被放入 board 中时返回 true,否则返回 false。例如,board={{"#"," ","#"},{" "," ","#"},{"#","c"," "}},word="abc",答案为 true,因为 word 可以放在中间一列。要求设计如下成员函数:

```
bool placeWordInCrossword(vector<vector<char>>& board, string word) { }
```

解:假设 word 的长度为 len,首先找到一个非"#"的位置(i,j),试探沿着 di 方向是否能够放置 word,可以放置的条件检测如下。

(1) 位置(i,j)的 di 反方向位置一定是"#",即检测首位置的 di 方向上的前一个位置是否满足条件(3)。

(2) 位置(i,j)的 di 方向的后面第 len−1 个位置是存在的,如果存在后面第 len 个位置,则该位置必须是"#",即检测尾位置的 di 方向上的后一个位置是否满足条件(3)。

(3) word 是否可以从(i,j)位置开始放置,直到放置完毕。

如果上述条件均满足,返回 true,否则试探位置(i,j)的其他方位,若位置(i,j)试探完毕,则找到其他非"#"的位置继续试探,全部试探完毕返回 false。对应的程序如下:

```
class Solution {
public:
```

```
bool placeWordInCrossword(vector<vector<char>>& board, string word) {
    int dx[] = {0,1,0,-1};                                  //水平方向的偏移量(顺时针方向)
    int dy[] = {1,0,-1,0};                                  //垂直方向的偏移量
    int len = word.size();
    int m = board.size(),n = board[0].size();
    for (int i = 0;i < m;i++) {
        for (int j = 0;j < n;j++) {
            if (board[i][j] == '#') continue;               //单词不能占据'#'
            for (int di = 0;di < 4;di++) {
                int xx = i + dx[(di + 2) % 4];
                int yy = j + dy[(di + 2) % 4];
                if (xx >= 0 && xx < m && yy >= 0 && yy < n) {
                    if (board[xx][yy]!= '#')                //首位置的条件(3)检测
                        continue;
                }
                xx = i + dx[di] * (len - 1);
                yy = j + dy[di] * (len - 1);
                if (xx < 0 || xx >= m || yy < 0 || yy >= n)  //超界时跳过
                    continue;
                xx += dx[di]; yy += dy[di];
                if (xx >= 0 && xx < m && yy >= 0 && yy < n) {
                    if (board[xx][yy]!= '#')                //尾位置的条件(3)检测
                        continue;
                }
                bool flag = true;
                for (int k = len - 1;k >= 0;k -- ) {        //检测中间是否匹配
                    xx -= dx[di];
                    yy -= dy[di];
                    if (board[xx][yy]!= ' ' && board[xx][yy]!= word[k]) {
                        flag = false;
                        break;
                    }
                }
                if (flag) return true;
            }
        }
    }
    return false;
  }
};
```

上述程序提交后通过，执行用时为 160ms，内存消耗为 58.3MB。

3.12 LeetCode2151——基于陈述统计最多好人数★★★

问题描述：游戏中存在两种角色，即好人（该角色只说真话）和坏人（该角色可能说真话，也可能说假话）。给定一个 $n \times n$（$2 \leqslant n \leqslant 15$）的二维整数数组 statements，表示 n 个玩家对彼此角色的陈述。具体来说，statements[i][j] 可以是下列值之一：0 表示 i 的陈述认为 j 是坏人，1 表示 i 的陈述认为 j 是好人，2 表示 i 没有对 j 作出陈述。玩家不会对自己进行陈述，即对所有 statements[i][i]=2。设计一个算法根据这 n 个玩家的陈述求可以认为是好人的最大数目。要求设计如下成员函数：

```
int maximumGood(vector<vector<int>>&statements) { }
```

解：在 n 个人中每个人都可能是好人或者坏人，这样有 2^n 种情况，每种情况用一个子集表示。例如 $n=3$ 时，3 个人的编号为 0～2，有 $2^3=8$ 种情况，每种情况用 3 个二进制位表示，0 表示坏人，1 表示好人，如二进制数 101 表示玩家 0 和 2 是好人，玩家 1 是坏人。这样恰好用十进制数 0～2^n-1 表示全部情况。

对于每种情况 s，用 sum 累计好人的人数(初始为 0)，遍历 s 的每一位，若某一位 i 是 1，说明玩家 i 是好人，所谓好人只说真话，也就是说若 statements[i][j]<2 成立(说明玩家 i 对玩家 j 有陈述)，而在情况 s 中玩家 j 的好人值为(s>>j)&1，若两者不相等，说明情况 s 是矛盾的，忽略该情况。如果情况 s 中每个好人和实际陈述相符，则累计其中好人的个数 sum，在所有这样的情况中通过比较求最大 sum。对应的程序如下：

```
class Solution {
public:
  int maximumGood(vector<vector<int>>&statements) {
      int n=statements.size();
      int ans=0;
      for (int s=1;s<1<<n;s++) {
          int sum=0;                                          //子集 s 中好人的个数
          for (int i=0;i<n;i++) {
              if ((s>>i)&1) {                                 //枚举 s 中的好人 i
                  for (int j=0;j<n;j++) {                     //枚举 i 的所有陈述
                      if (statements[i][j]<2 && statements[i][j]!=((s>>j)&1)) {
                          goto next;                          //该陈述与 s 中的情况矛盾则忽略 s
                      }
                  }
                  sum++;
              }
          }
          ans=max(ans,sum);
          next:;
      }
      return ans;
  }
};
```

上述程序提交后通过，执行用时为 96ms，内存消耗为 8.2MB。

3.13 POJ2000——金币

时间限制：1000ms，空间限制：30 000KB。

问题描述：国王向忠诚的骑士支付金币，第一天支付给骑士一枚金币，在接下来的两天(第二天和第三天)中每一天支付给骑士两枚金币，在接下来的 3 天(第四天、第五天和第六天)中每一天支付给骑士 3 枚金币，在接下来的 4 天(第七天、第八天、第九天和第十天)中每一天支付给骑士 4 枚金币。这种支付模式将无限期地继续下去，即在连续 n 天每天收到 n 个金币后，骑士将在接下来的 $n+1$ 天连续收到 $n+1$ 个金币，其中 n 为任意正整数。设计一个算法，计算在给定天数内(从第一天开始)国王支付给骑士的金币总数。

输入格式：输入至少包含一行，但不超过 21 行。输入的每一行(最后一行除外)都包含一个测试用例的数据，仅包含一个表示天数的整数(范围为 1～10 000)，输入的结束由包含

数字 0 的行表示。

输出格式：每个测试用例只有一行输出，这一行包含来自输入行的天数，后跟一个空格以及在给定天数内支付给骑士的金币总数，从第一天开始。

输入样例：

```
10
6
7
11
15
16
100
10000
1000
21
22
0
```

输出样例：

```
10 30
6 14
7 18
11 35
15 55
16 61
100 945
10000 942820
1000 29820
21 91
22 98
```

解：如果简单地枚举每一天的金币数可能超时，改进方法是将金币数相同的若干天看成一个段，先求出 n 天对应的前一个段的天数 pred，累计到 pred 为止的金币数 sum，第 n 天对应段的每天金币数为 coin，则答案等于 sum＋(n－pred) * coin。

例如，n＝16，如表 3.1 所示，对应的段号为 6，该段的每天金币数 coin＝6，前一个段号为 5，前面 5 个段的总金币数 sum 为 1×1＋2×2＋3×3＋4×4＋5×5＝55，前面 5 个段的总天数 pred＝15，答案为 sum＋(n－pred)×coin＝55＋1×6＝61。

表 3.1　n＝16 时的计算过程

段号	1	2		3			4				5					6
天	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
金币	1	2	2	3	3	3	4	4	4	4	5	5	5	5	5	6
合并	1×1	2×2		3×3			4×4				5×5					

对应的程序如下：

```
#include<iostream>
using namespace std;
int Coins(int n) {
    if (n==1) return 1;
    int coin=1;
    int sum=1;
    int d=1;
```

```
    int pred;
    while (d < n) {
        if (coin > 1)
            sum += coin * coin;
        coin++;
        pred = d;
        d += coin;
    }
    return sum + (n - pred) * coin;
}
int main() {
    int n;
    cin >> n;                                          //读入第一个测试用例
    while (n > 0) {
        cout << n << " " << Coins(n) << endl;
        cin >> n;                                      //读入下一个测试用例
    }
    return 0;
}
```

上述程序提交后通过，执行用时为 65ms，内存消耗为 176KB。

3.14 POJ1013——假币问题

时间限制：1000ms，空间限制：10 000KB。

问题描述：Sally 有 12 枚银币，其中有 11 枚真币和一枚假币，假币看起来和真币没有什么区别，只是重量不同。Sally 不知道假币比真币轻还是重，于是他向朋友借了一架天平。朋友希望 Sally 称 3 次就能找出假币并且确定假币是轻还是重。例如，如果 Sally 用天平称两枚银币，发现天平平衡，说明两枚都是真的，如果 Sally 用一枚真币与另一枚银币比较，发现它比真币轻或重，说明它是假币。经过精心安排每次的称量，Sally 保证能在称 3 次后确定假币。

输入格式：第一行包含整数 n，表示测试用例的数目。对于每个测试用例输入 3 行，每行表示一次称量的结果，Sally 事先将银币标号为 A～L，每次称量的结果用 3 个以空格隔开的字符串表示，即天平左边放置的银币、天平右边放置的银币和平衡状态，其中平衡状态用"up"、"down"和"even"表示，分别为左端重、右端重和平衡。天平左右的银币数总是相等的。

输出格式：输出哪一个标号的银币是假币，并说明它比真币轻还是重(is heavy 或者 is light)。

输入样例：

```
1
ABCD EFGH even                    //表示 ABCD 银币的重量等于 EFGH 银币的重量
ABCI EFJK up                      //表示 ABCI 银币的重量大于 EFJK 银币的重量
ABIJ EFGH even                    //表示 EFGH 银币的重量等于 EFGH 银币的重量
```

输出样例：

```
K is the counterfeit coin and it is light.
```

解：用数组 a 表示 12 个银币的重量，真币的重量为 0。设计 balanced()算法用于判断当 12 个银币的重量已知时是否满足 3 次称量的情况。对于每个银币 i，置 $a[i]=-1$(假设

银币 i 是较轻的假币),如果 balanced()算法返回 true 说明假设成立,否则置 $a[i]=1$(假设银币 i 是较重的假币),如果 balanced()算法返回 true 说明假设成立,否则说明银币 i 是真币,置 $a[i]=0$ 继续。最后根据成立的假设输出答案。对应的程序如下:

```
#include<iostream>
int a[12];
char left[3][7],right[3][7],state[3][7];
bool balanced(){                                    //判断当前的情况是否满足
  for(int i=0;i<3;i++){                             //枚举3次称量的情况
     int leftw=0,rightw=0;
     for(int j=0;left[i][j];j++)                    //求出左端银币的总重量
        leftw+=a[left[i][j]-'A'];
     for(int j=0;right[i][j];j++)                   //求出右端银币的总重量
        rightw+=a[right[i][j]-'A'];
     if(leftw>rightw && state[i][0]!='u')           //违反"up"的称量结果
        return false;
     if(leftw==rightw && state[i][0]!='e')          //违反"even"的称量结果
        return false;
     if(leftw<rightw && state[i][0]!='d')           //违反"down"的称量结果
        return false;
   }
   return true;
}
int main(){
   int n,no;
   scanf("%d",&n);
   while (n--){
     for (int i=0;i<3;i++)
        scanf("%s%s%s",left[i],right[i],state[i]);
     for(int i=0;i<12;i++)
        a[i]=0;
     for(no=0;no<12;no++){                          //枚举每个银币
        a[no]=1;                                    //假设银币i是较轻的假币
        if(balanced()) break;
        a[no]=-1;                                   //假设银币i是较重的假币
        if(balanced()) break;
        a[no]=0;
     }
     printf("%c is the counterfeit coin and it is %s.\n",no+'A',a[no]>0?"heavy":"light");
  }
  return 0;
}
```

上述程序提交后通过,执行用时为 0ms,内存消耗为 92KB。

3.15 POJ1256——字谜

时间限制:1000ms,空间限制:10 000KB。

问题描述:编写一个程序从给定的一组字母中生成所有可能的单词。例如,给定单词"abc",通过探索 3 个字母的所有不同组合,输出单词"abc"、"acb"、"bac"、"bca"、"cab"和"cba"。单词中的一些字母可能会出现不止一次,在这样的情况下不应多次生成相同的单词,并且应按字母升序输出单词。

输入格式:输入由几个单词组成。第一行包含一个数字,表示后面的单词数。以下每

一行包含一个单词。一个单词由从 A 到 Z 的大写或小写字母组成，大写和小写字母被认为是不同的，每个单词的长度小于 13。

输出格式：对于输入的每个单词，输出应该包含可以使用给定单词的字母生成的所有不同单词，从同一个输入生成的单词应该按字母升序输出。一个大写字母位于相应的小写字母之前。

输入样例：

```
3
aAb
abc
acba
```

输出样例：

```
Aab
Aba
aAb
abA
bAa
baA
abc
acb
bac
bca
cab
cba
aabc
aacb
abac
abca
acab
acba
baac
baca
bcaa
caab
caba
cbaa
```

解：题目就是要产生字符串 str 的全排列，需要注意以下两点。

(1) 排列要除重，不会输出重复的排列。

(2) 按字母升序输出，这里的字母升序不是指 ASCII 码顺序，而是 A＜a＜B＜b＜C＜c，以此类推。

为了简单，采用 STL 中的 next_permutation()函数枚举排列，该函数具有自动除重功能，为了满足(2)，需要制定这样的比较关系：转换为大写字母相同时按 ASCII 码比较，例如 A＜a、B＜b 等，转换为大写字母不相同时按转换的大写字母比较，例如 A＜B、A＜b、b＜C 等。对应的程序如下：

```
#include<iostream>
#include<string.h>
#include<algorithm>
#define N 14
using namespace std;
```

```
char str[N];
struct Cmp {
    bool operator()(char& a,char& b) {          //制定比较关系
      if(toupper(a) == toupper(b))
          return a < b;
      else
          return toupper(a)< toupper(b);
    }
};
int main() {
    int n,m;
    scanf("%d",&n);
    while(n--){
      scanf("%s",str);
      m = strlen(str);
      sort(str,str + m,Cmp());
      printf("%s\n",str);
      while(next_permutation(str,str + m,Cmp()))
          printf("%s\n",str);
    }
    return 0;
}
```

上述程序提交后通过，执行用时为 63ms，内存消耗为 92KB。

3.16 POJ3187——倒数和

时间限制：1000ms，空间限制：65 536KB。

问题描述：有这样一个游戏，以一定的顺序写从 1 到 n（$1 \leqslant n \leqslant 10$）的数字，然后将相邻的数字相加以产生一个数字少一个的新列表，重复这个过程，直到只剩下一个数字。例如，游戏的一个实例（当 $n=4$ 时）可能是这样的：

```
3   1   2   4
  4   3   6
    7   9
      16
```

现在给定 n 和最后一行的数字，请编写程序输出第一行的数列。

输入格式：输入一行包含两个以空格分隔的整数，即 n 和最后一行的数字。

输出格式：输出第一行的数列，如果有多个这样的数列，选择字典顺序最小的一个。

输入样例：

```
4 16
```

输出样例：

```
3 1 2 4
```

解：所求答案一定是 $1 \sim n$ 的某个排列。采用穷举法，用数组 a 枚举 $1 \sim n$ 的全部排列（初始 a 取值为 $1 \sim n$）。对于每个排列 a，以其作为第一行，用数组 b 保存其累计结果，最后一行 $b[n-1]$ 仅包含一个整数，若 $b[n-1][0]=\text{sum}$，则输出对应的 a。为了简单，采用 STL 中的 next_permutation()函数枚举排列，由于该函数产生的排列是递增的，所以满足要求的第一个 a 就是最小排列。对应的程序如下：

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
#define MAXN 11
int main(){
    int a[MAXN];
    int n,sum;
    scanf("%d %d",&n,&sum);
    for(int i=1;i<=n;i++)                    //产生初始排列 1～n
        a[i-1]=i;
    int b[MAXN][MAXN];
    memset(b,0,sizeof(b));
    do{
        for(int i=0;i<n;i++)                 //由 a 作为第一行
            b[0][i]=a[i];
        for(int i=1;i<n;i++){                //求出其他 n-1 行
            for(int j=0;j<n-i;j++){
                b[i][j]=b[i-1][j]+b[i-1][j+1];
            }
        }
        if(b[n-1][0]==sum) break;            //找到满足要求的 a
    }while(next_permutation(a,a+n));         //枚举下一个排列
    for(int i=0;i<n;i++){                    //输出 a
        if(i==0) printf("%d",a[i]);
        else printf(" %d",a[i]);
    }
    return 0;
}
```

上述程序提交后通过，执行用时为 63ms，内存消耗为 88KB。