程设错题 6~10 20241015~20241020

1.经典问题：找质数

方法1：基本枚举法 O(n^3/2)

```
class Solution {
public:
    bool isprime(int n){
        if(n==2||n==3){
            return true;
        }
        for(int i=2;i*i<=n;i++){
            if(n%i==0){
                return false;
            }
        }
        return true;
    }
    int countPrimes(int n) {
        int sum=0;
        for(int i=2;i<n;i++){
            sum+=isprime(i);
        }
        return sum;
    }
};
```

方法2：埃氏筛

枚举没有考虑到数与数的关联性，因此难以再继续优化时间复杂度。接下来我们介绍一个常见的算法，该算法由希腊数学家厄拉多塞（Eratosthenes）提出，称为厄拉多塞筛法，简称埃氏筛。

我们考虑这样一个事实：**如果 x 是质数，那么大于 x 的 x 的倍数 2x,3x,… 一定不是质数**，因此我们可以从这里入手。

我们设 isPrime[i] 表示数 i 是不是质数，如果是质数则为 1，否则为 0。从小到大遍历每个数，如果这个数为质数，则将其所有的倍数都标记为合数（除了该质数本身），即 0，这样在运行结束的时候我们即能知道质数的个数。

这种方法的正确性是比较显然的：这种方法显然不会将质数标记成合数；另一方面，当从小到大遍历到数 x 时，倘若它是合数，则它一定是某个小于 x 的质数 y 的整数倍，故根据此方法的步骤，我们在遍历到 y 时，就一定会在此时将 x 标记为 isPrime[x]=0。因此，这种方法也不会将合数标记为质数。

当然这里还可以继续优化，对于一个质数 x，如果按上文说的我们从 2x 开始标记其实是冗余的，应该直接从 x⋅x 开始标记，因为 2x,3x,… 这些数一定在 x 之前就被其他数的倍数标记过了，例如 2 的所有倍数，3 的所有倍数等。

这种方法不用再判断单个数是否为质数，可以减少筛查从1~n的次数

埃氏筛的筛查方法：从一个已知的质数开始排除无限多比他大的合数

```
class Solution {
public:
```

```
    int countPrimes(int n) {
        vector<int> isPrime(n, 1);
        //初始化
        int ans = 0;
        for (int i = 2; i < n; ++i) {
            if (isPrime[i]) {
                //则该数为质数
                ans += 1;
                //接下来的操作：标志其倍数为合数
                //从n^2开始标记！(对于倍数kn,k<n,一定有数k为质数，在之前被标记过)
                if ((long long)i * i < n) {
                    for (int j = i * i; j < n; j += i) {
                        isPrime[j] = 0;
                    }
                }
            }
        }
        return ans;
    }
};
```

3.线性筛 O(n)

埃氏筛其实还是存在冗余的标记操作，比如对于 45 这个数，它会同时被 $3,5$ 两个数标记为合数，因此我们优化的目标是让每个合数只被标记一次，这样时间复杂度即能保证为 $O(n)$，这就是我们接下来要介绍的线性筛。

相较于埃氏筛，我们多维护一个 $primes$ 数组表示当前得到的质数集合。我们从小到大遍历，如果当前的数 x 是质数，就将其加入 $primes$ 数组。

另一点与埃氏筛不同的是，「标记过程」不再仅当 x 为质数时才进行，而是对每个整数 x 都进行。对于整数 x，我们不再标记其所有的倍数 $x\cdot x, x\cdot(x+1),\ldots$，而是只标记质数集合中的数与 x 相乘的数，即 $x\cdot primes_0, x\cdot primes_1,\ldots$，且在发现 $x \bmod primes_i = 0$ 的时候结束当前标记。

核心点在于：如果 x 可以被 $primes_i$ 整除，那么对于合数 $y = x\cdot primes_{i+1}$ 而言，它一定在后面遍历到 $\frac{x}{primes_i}\cdot primes_{i+1}$ 这个数的时候会被标记，其他同理，这保证了每个合数只会被其「最小的质因数」筛去，即每个合数被标记一次。

线性筛还有其他拓展用途，有能力的读者可以搜索关键字「积性函数」继续探究如何利用线性筛来求解积性函数相关的题目。

x=k*primes[i]，则对于数x*primes[i+1]=k*primes[i]*primes[i+1],可以在primes[i+1]遍历的时候取到

```
class Solution {
public:
    int countPrimes(int n) {
        vector<int> primes;
        vector<int> isPrime(n, 1);
        for (int i = 2; i < n; ++i) {
            if (isPrime[i]) {
                primes.push_back(i);
            }
            for (int j = 0; j < primes.size() && i * primes[j] < n; ++j) {
                isPrime[i * primes[j]] = 0;
                if (i % primes[j] == 0) {
                    break;
                }
            }
        }
        return primes.size();
```

```
    }
};
```

2.数组输出：平方矩阵问题

输入整数 N，输出一个 N 阶的回字形二维数组。

数组的最外层为 1，次外层为 2，以此类推。

输入格式

输入包含多行，每行包含一个整数 N。

当输入行为 $N=0$ 时，表示输入结束，且该行无需作任何处理。

输出格式

对于每个输入整数 N，输出一个满足要求的 N 阶二维数组。

每个数组占 N 行，每行包含 N 个用空格隔开的整数。

每个数组输出完毕后，输出一个空行。

思路：**最基本的数组输出：**两层for循环嵌套（通过外层循环每执行一次回车跳转到下一行进行输出）

解法一：曼哈顿距离+坐标法

```
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;
void printarray(int n);
int main(){
    int n;
    cin>>n;
    while(n!=0){
        printarray(n);
        cout<<endl;
        cin>>n;
    }
    return 0;
}
void printarray(int n){
    double maxi;
    int list[n][n];
    if(n%2==1){
        maxi=(n+1)/2;
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                list[i][j]=int(maxi-max(abs((n-1)/2-i),abs((n-1)/2-j)));
                //核心代码：对应位置上的数和其到中心点的曼哈顿距离有关！
                cout<<list[i][j]<<" ";
            }
            cout<<endl;
        }
    }else{
        maxi=n/2;
        for(int i=0;i<n;i++){
```

```
            for(int j=0;j<n;j++){
                list[i][j]=int(maxi+0.5-max(abs(double(n-1)/2-i),abs(double(n-
1)/2-j)));
                //偶数的情况不存在唯一的中心点，但中心点的作用只是通过曼哈顿距离的坐标表示来确
定对应位置上的数的值。故可以引入一个虚拟中心点（浮点数：(n-1)/2）
                cout<<list[i][j]<<" ";
            }
            cout<<endl;
        }
    }
}
```

优化解法：不再依赖中心点，直接根据坐标输出值

> ①通过观察回字形矩阵, 矩阵关于对角线是左上方和右下方对称的
> ②利用二维行列循环, **获取行列+1的最小值(即min(i + 1, j + 1))**, 可得如下图形(例如n == 4):
> 1 1 1 1
> 1 2 2 ~~2~~
> 1 2 ~~3 3~~
> 1 ~~2 3 4~~
> 可以看出未划横线部分（左上部分）满足题解,此时如果使图像沿着对角线翻转,再重合,即可求解答案
> ③翻转图像,采用min(n - i, n - j)即可, 得到图像如下(例如n == 4):
> ~~4 3 2~~ 1
> ~~3 3~~ 2 1
> ~~2~~ 2 2 1
> 1 1 1 1
> ④进行图像的重合, 对应位置取最小值即可求解min(Left上, right下)

```
#include <iostream>
#include <cmath>

using namespace std;

int main(){
 int n;
while (cin >> n, n){
 for (int i = 0; i < n; i ++){
     for(int j = 0; j < n; j ++){
         cout << min(min(i + 1, j + 1), min(n - i, n - j)) << " ";
         //通过一个min（）,输出对应的数字
     }
     cout << endl ;
 }
 cout << endl;
}

return 0;
```

方法二：**蛇形矩阵**求解

```
#include <iostream>
```

```
#include <cstring>
#include <cmath>

using namespace std;

const int N = 100 + 10;
int m[N][N];

int main(){
    int n;
    int dx[] = {-1, 0, 1, 0}, dy[] = {0, 1, 0, -1};
    //每个向量代表向不同方向移动

    while(cin >> n, n ){
        memset(m, 0, sizeof m);//初始化数组
        int d = 1, x = 0, y = 0;
        int cnt = 0;  // 表示改变方向次数
        int res = 1;  // 回形当前圈数
        for (int i = 0; i < n * n; i ++){
            //i是一个计数器，代表一共输出i个数
            int a = x + dx[d], b = y + dy[d];
            //(x,y) is the previous point,and (a,b)is the point that has moved
            //一个很关键的点：(a,b)相当于探路的坐标，并不会直接赋值给(x,y)。一旦if语句发现(a,b)异常(例如需要拐弯或进入小循环)，则会重新调整方向后再赋值给x,y
            m[x][y] = res;

            if (a < 0 || a >= n || b < 0 || b >= n || m[a][b]){
                //可能的需要改变方向的情况：走到底需要拐弯，走到头(m[a][b]已经被填充，输出一个非0值)
                d = (d + 1) % 4;
                //d的改变：1--2--3--4
                a = x + dx[d], b = y + dy[d];
                cnt ++;
                if (!(cnt % 4)) res ++;
                //当cnt==4时，代表转完了一圈，则进入小圈中
            }
            x = a, y = b;
        }

        for (int i = 0; i < n; i ++){
            for (int j = 0; j < n; j ++)
                cout << m[i][j] << ' ';
            cout << endl;
        }
        cout << endl;
    }

    return 0;
}
```

3.斐波那契数列

（1）基本的递归做法

```
#include <iostream>
using  namespace std;
int Fib(int x);
int main (){
    int n,m;
    cin>>n;
    for(int i=0;i<n;i++){
        cin>>m;
        cout<<Fib(m)<<endl;
    }
    return 0;
}
int Fib(int x){
    if(x==1||x==2){
        return 1;
    }else{
        return Fib(x-1)+Fib(x-2);
    }
}
```

（2）方法优化

```
#include <stdio.h>
using namespace std;
int t;
int main()
{
  cin>>t;
  while(t--)
  {
    int n;
    cin>>n;
    long long number=0,numberfront=1,numberfrofront;
    for(int i=0; i<=n; i++)
    {
      if (i==n){
          printf("Fib(%d) = %lld\n",n,number);
      }
      numberfrofront=number+numberfront;
      number=numberfront;
      numberfrofront=numberfront;
      //相当于做迭代，不断向前进（比递归的时间复杂度要低）
    }
  }
  return 0;
}
```

·此处也可以使用vector数组存储生成的每一项，可减少计算的复杂（不用每次函数调用的时候都要重新计算一遍）

4.ACwing 只出现一次的字符

给你一个只包含小写字母的字符串。

请你判断是否存在只在字符串中出现过一次的字符。

如果存在，则输出满足条件的字符中位置最靠前的那个。

如果没有，输出 `no`。

输入格式

共一行，包含一个由小写字母构成的字符串。

数据保证字符串的长度不超过 100000。

思路：构建一种映射的关系

直接使用两个数组构建映射的关系（前提是数组的长度已知）

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    string test;
    cin>>test;
    int list[1000]={0};
    for(auto i:test){
            list[i]++;
    }
    for(auto i:test){
        //注意这里遍历不是按list的顺序进行遍历，而是还是按照test的值(不同字符的ASCII码)来进行遍历，确保输出的始终是最大且最靠前的一项。
        if(list[i]==1){
            cout<<i;
            goto end;
        }
    }
    cout<<"no";
    end:
    return 0;
}
```

5.ACwing 字符串中最长的连续出现的字符

题目描述

求一个字符串中最长的连续出现的字符，输出该字符及其出现次数，字符串中无空白字符（空格、回车和tab），如果这样的字符不止一个，则输出第一个。

输入格式

第一行输入整数N，表示测试数据的组数。

每组数据占一行，包含一个不含空白字符的字符串，字符串长度不超过200。

输出格式

共一行，输出最长的连续出现的字符及其出现次数，中间用空格隔开。

解法①：最基本的思路

- 使用前后指针和flag判断连续字符
- 使用countlist数组记录每个字符连续出现的最大值
- 使用vector数组successful按顺序储存连续出现的字符，并且在其值更新时自动移动到序列尾端

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
void findthelongest(){
    int countlist[300]={0};
    vector <char> successful;
    int flag=0;
    int tempcount=0;
    string teststring;
    cin>>teststring;
    for(int i=0;i<teststring.length()-1;i++){
        if(teststring[i]==teststring[i+1]){
            flag=1;

 if(find(successful.begin(),successful.end(),teststring[i])==successful.end()){
                successful.push_back(teststring[i]);
            }


        }
        else{
            flag=0;
            if(tempcount>countlist[teststring[i]]){
                countlist[teststring[i]]=tempcount;

 successful.erase(remove(successful.begin(),successful.end(),teststring[i]));
                successful.push_back(teststring[i]);
            }
            tempcount=0;
        }
        if(flag==1){
            tempcount++;
            if(i==teststring.length()-2){
                if(tempcount>countlist[teststring[i]]){
                countlist[teststring[i]]=tempcount;

 successful.erase(remove(successful.begin(),successful.end(),teststring[i]));
                successful.push_back(teststring[i]);
                }
            }
        }
    }
    int max=0;
    char maxchar=teststring[0];
    if(!successful.empty()){
        for(auto tes:successful){
        if(countlist[tes]>max){
            maxchar=tes;
```

```
            max=countlist[tes];
        }
    }
    }
    cout<<maxchar<<" "<<max+1<<endl;
}
int main(){
    int N;cin>>N;
    for(int i=0;i<N;i++){
        findthelongest();
    }
    return 0;
}
```

解法②可移动的双指针

解法优化:

- 使用可移动的双指针ij，每次只有j向前不断移动直到遇到不同的元素，i最后追上j
- 适应覆盖的思想而不使用指针存储

```
#include <iostream>
using namespace std;
int main()
{
    int T;
    cin >> T;
    while(T --)
    {
        int maxn = -1;//maxn记录最大长度
        string str, maxs;//maxs记录最大长度时的字符
        cin >> str;
        for(int i = 0; i < str.size();)
        {
            int j = i;
            int cnt = 0;
            while(str[j] == str[i] && j < str.size())//当指针j没有越界且与指针i的内容相同时移动
                j ++, cnt ++;
            if(cnt > maxn)//更新最大值
                maxn = cnt, maxs = str[i];
            i = j ;//移动指针i
        }
        cout << maxs << " " << maxn << endl;
    }
}
```