入门模拟

模拟题是一类“题目怎么说，你就怎么做”的题目，这类题目不涉及算法，完全只是根据题目的描述来进行代码的编写，考察的是代码能力。

一、乙级题目：

1、程序运行时间：

不妨简单假设常数 CLK_TCK 为 100。现给定被测函数前后两次获得的时钟打点数，请你给出被测函数运行的时间。

（1）输入格式：输入在一行中顺序给出 2 个整数 C1 和 C2。注意两次获得的时钟打点数肯定不相同，即 C1 < C2，并且取值在 [0,107]。

（2）输出格式：在一行中输出被测函数运行的时间。运行时间必须按照 `hh:mm:ss`（即2位的`时:分:秒`）格式输出；不足 1 秒的时间四舍五入到秒。

```
#define CLK_TCK 100

int main(){
    int time, h, m,s;
    double c1, c2, seconds;
    scanf("%lf%lf", &c1, &c2);
    seconds = round( (c2 - c1)/CLK_TCK );
    time = (int) seconds;
    h = time/3600;                          // 计算经过了多少个小时
    m = (time - 3600*h)/60;
    s = time - 3600*h - 60*m;
    printf("%02d:%02d:%02d\n", h, m, s);
    return 0;
}
```

2、数字分类：

（1）给定一系列正整数，请按要求对数字进行分类，并输出以下 5 个数字：

- *A*1 = 能被 5 整除的数字中所有偶数的和；
- *A*2 = 将被 5 除后余 1 的数字按给出顺序进行交错求和，即计算 $n1-n2+n3-n4\cdots$；
- *A*3 = 被 5 除后余 2 的数字的个数；
- *A*4 = 被 5 除后余 3 的数字的平均数，精确到小数点后 1 位；
- *A*5 = 被 5 除后余 4 的数字中最大数字。

（2）输入格式：每个输入包含 1 个测试用例。每个测试用例先给出一个不超过 1000 的正整数 *N*，随后给出 *N* 个不超过 1000 的待分类的正整数。数字间以空格分隔。

（3）输出格式：对给定的 *N* 个正整数，按题目要求计算 *A*1~*A*5 并在一行中顺序输出。数字间以空格分隔，但行末不得有多余空格。若分类之后某一类不存在数字，则在相应位置输出 `N`。

```
#include <iostream>
using namespace std;
int main(){
    int n;
    int a[5] = {0}, count[5] = {0};                 //count用于记录该类数字的数量
    scanf("%d", &n);
    int array[n];
    for(int i=0;i<n;i++){
        scanf("%d", &array[i]);
        if(array[i] % 5 == 0){                      //被5整除的偶数之和
            if(array[i] % 2 ==0){
                a[0] += array[i];
                count[0]++;
            }
        }
        else if(array[i] % 5 == 1){                 //除5余1的数的交错求和
            if(count[1] % 2 == 0) a[1] += array[i];
            else a[1] -= array[i];
            count[1]++;
        }
        else if(array[i] % 5 == 2){                 //除5余2的数字个数
            count[2]++;
        }
        else if(array[i] % 5 == 3){                 //除5余3的数字的平均数
            a[3] += array[i];
            count[3]++;
        }
        else if(array[i] % 5 == 4){                 //除5余4的数字中的最大值
            if(a[4] < array[i]){
                a[4] = array[i];
                count[4]++;
            }
        }
    }

    if(count[0] == 0) printf("N ");
    else printf("%d ",a[0]);
    if(count[1] == 0) printf("N ");
    else printf("%d ",a[1]);
    if(count[2] == 0) printf("N ");
    else printf("%d ",count[2]);
    if(count[3] == 0) printf("N ");
    else printf("%.1f ", (double)a[3]/count[3]);
    if(count[4] == 0) printf("N");
    else printf("%d",a[4]);
    return 0;
}
```

3、锤子、剪刀、布：

（1）输入格式：输入第 1 行给出正整数 *N*（≤105），即双方交锋的次数。随后 *N* 行，每行给出一次交锋的信息，即甲、乙双方同时给出的的手势。`C` 代表“锤子”、`J` 代表“剪刀”、`B` 代表“布”，第 1 个字母代表甲方，第 2 个代表乙方，中间有 1 个空格。

（2）输出格式：输出第 1、2 行分别给出甲、乙的胜、平、负次数，数字间以 1 个空格分隔。第 3 行给出两个字母，分别代表甲、乙获胜次数最多的手势，中间有 1 个空格。如果解不唯一，则输出按字母序最小的解。

```
#include <iostream>
using namespace std;
int main(){
    int n, jid=0, yid=0;
    int jia_sheng=0, jia_ping=0, jia_fu=0;        // 记录甲胜、平、负的次数
    int yi_sheng=0, yi_ping=0, yi_fu=0;           // 记录乙胜、平、负的次数
    int jia[3]={0}, yi[3]={0};                    // 按照BCJ的顺序记录甲乙所
用手势的次数
    char fiture[2], konge;

    scanf("%d", &n);
    for(int i=0;i<n;i++){
        getchar();
        scanf("%c %c", &fiture[0], &fiture[1]);
        // printf("%c %c\n", fiture[0], fiture[1]);
        // 甲赢
        if((fiture[0] == 'C' && fiture[1] == 'J') || (fiture[0] == 'J' &&
fiture[1] == 'B') || (fiture[0] == 'B' && fiture[1] == 'C')){
            jia_sheng++;yi_fu++;
            if(fiture[0] == 'B') jia[0]++;
            else if(fiture[0] == 'C') jia[1]++;
            else if(fiture[0] == 'J') jia[2]++;
        }
        // 乙赢
        else if((fiture[0] == 'J' && fiture[1] == 'C') || (fiture[0] == 'B'
&& fiture[1] == 'J') || (fiture[0] == 'C' && fiture[1] == 'B')){
            jia_fu++;yi_sheng++;
            if(fiture[1] == 'B') yi[0]++;
            else if(fiture[1] == 'C') yi[1]++;
            else if(fiture[1] == 'J') yi[2]++;
        }
        // 平局
        else{
            jia_ping++;yi_ping++;
        }
    }
    printf("%d %d %d\n", jia_sheng, jia_ping, jia_fu);
    printf("%d %d %d\n", yi_sheng, yi_ping, yi_fu);

    for(int i=1;i<3;i++){
        if(jia[jid] < jia[i]) jid = i;
```

```
        if(yi[yid] < yi[i]) yid = i;
    }
    if(jid == 0) printf("B ");
    else if(jid == 1) printf("C ");
    else if(jid == 2) printf("J ");

    if(yid == 0) printf("B");
    else if(yid == 1) printf("C");
    else if(yid == 2) printf("J");

    return 0;
}
```

4、一元多项式求导：

设计函数求一元多项式的导数。（注：*x****n*（*n*为整数）的一阶导数为*n*x*n*−1。）

（1）输入格式：以**指数递降**方式输入多项式非零项系数和指数（绝对值均为不超过 1000 的整数）。数字间以空格分隔。

（2）输出格式：以与输入相同的格式输出导数多项式非零项的系数和指数。数字间以空格分隔，但结尾不能有多余空格。

注意“零多项式”的指数和系数都是 0，但是表示为 `0 0`。

```
#include <iostream>
using namespace std;
int main(){
    int a[1010] = {0};
    int xishu, zhishu, count=0;     //count记录不为0导数项的个数
    while(scanf("%d %d", &xishu, &zhishu) != EOF){
        a[zhishu] = xishu;
    }
    a[0] = 0;                       //零次项求导后为0
    for(int i=1;i<=1000;i++){
        a[i-1] = a[i] * i;          //求导公式
        a[i] = 0;                   //此句不可省略
        if(a[i-1] != 0) count++;
    }
    if(count == 0) printf("0 0"); //特判
    else{
        for(int i=1000;i>=0;i--){
            if(a[i] != 0){
                printf("%d %d", a[i], i);
                count--;
                if(count != 0) printf(" ");
            }
        }
    }
    return 0;
}
```

二、甲级题目：

1、A+B Format：

Calculate *a*+*b* and output the sum in standard **format** -- that is, the **digits** must be separated into groups of three by **commas** (unless there are less than four digits).

Input Specification:

Each input file contains one test case. Each case contains a pair of **integers** *a* and *b* where −10^6≤*a*,*b*≤10^6. The numbers are separated by a **space**.

Output Specification:

For each test case, you should output the sum of *a* and *b* in one line. The sum must be written in the standard format.

Sample Input:

```
-1000000 9
```

Sample Output:

```
-999,991
```

```
#include <iostream>
using namespace std;
int main() {
    int a, b;
    cin >> a >> b;
    string s = to_string(a + b);
    int len = s.length();
    if(s[0] == '-'){
        cout<< '-';
        s.erase(0, 1);    //从位置 0 开始，删除 1 个字符
    }
    int count = 0;        //从低位到高位记录当前位置
    for(int i=s.length()-1;i>=0;i--){       //此处为什么s.length()不能替换成
len
        count++;
        if(count % 3 ==0 && i>0){           //i>0,即位置 0 之前无需添加逗号
            s.insert(i, ",");
        }
    }
    cout<<s<<endl;
    return 0;
}
```

2、A+B for Polynomials

This time, you are supposed to find *A*+*B* where *A* and *B* are two **polynomials**.

Input Specification:

Each input file contains one test case. Each case occupies 2 lines, and each line contains the information of a polynomial: $K\ N_1\ a_{N_1}\ N_2\ a_{N_2}\ \dots\ N_K\ a_{N_K}$

where *K* is the number of **nonzero** terms in the polynomial, N_i and a_{N_i} (*i*=1,2,⋯,*K*) are the **exponents** and **coefficients**, **respectively**.

It is given that $1 \le K \le 10$, $0 \le N_K < \cdots < N_2 < N_1 \le 1000$.

Output Specification:

For each test case you should output the sum of *A* and *B* in one line, with the same format as the input. Notice that there must be NO extra space at the end of each line. **Please be accurate to 1 decimal place.**

Sample Input:

```
2 1 2.4 0 3.2
2 2 1.5 1 0.5
```

Sample Output:

```
3 2 1.5 1 2.9 0 3.2
```

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int main(){
    int k1, k2, zhishu, count=0;
    double a[1111] = {0};
    double xishu;

    scanf("%d", &k1);
    for(int i=0;i<k1;i++){
        scanf("%d %lf", &zhishu, &xishu);
        a[zhishu] = xishu;                  //a[n]表示n次项的系数
    }

    scanf("%d", &k2);
    for(int i=0;i<k2;i++){
        scanf("%d %lf", &zhishu, &xishu);
        a[zhishu] += xishu;                 //对系数直接相加
    }
    for(int i=0;i<1111;i++){
        //记录非零项个数，双精度浮点数判零方法（大于1e-15，即大于0）
        if(fabs(a[i]) > 1e-15) count++;
```

```
    }
    printf("%d", count);
    for(int i=1110;i>=0;i--){
        if(a[i] != 0) printf(" %d %.1f", i, a[i]);
    }
    return 0;
}
```

3、$A \cdot B$ Polynomials

This time, you are supposed to find ***A×B*** where *A* and *B* are two polynomials.

Input Specification:

Each input file contains one test case. Each case occupies 2 lines, and each line contains the information of a polynomial: $K\ N_1\ a_{N_1}\ N_2\ a_{N_2}\ \dots\ N_K\ a_{N_K}$, where *K* is the number of nonzero terms in the polynomial, N_i and a_{N_i} $(i = 1, 2, \cdots, K)$ are the **exponents** and **coefficients**, respectively.

It is given that $1 \le K \le 10$, $0 \le N_K < \cdots < N_2 < N_1 \le 1000$.

Output Specification:

For each test case you should output the product of *A* and *B* in one line, with the same format as the input. Notice that there must be **NO** extra space at the end of each line. Please be accurate up to 1 decimal place.

Sample Input:

```
2 1 2.4 0 3.2
2 2 1.5 1 0.5
```

Sample Output:

```
3 3 3.6 2 6.0 1 1.6
```

```
#include <iostream>
using namespace std;

int main(){
    int k, k1, k2, zhishu, count=0;
    double xishu;
    double a1[1010]={0}, a2[1010]={0}, multiply[2020]={0};

    scanf("%d", &k1);
    for(int i=0;i<k1;i++){
        scanf("%d%lf", &zhishu, &xishu);
        a1[zhishu] = xishu;
    }
    scanf("%d", &k2);
    for(int i=0;i<k2;i++){
```

```
        scanf("%d%lf", &zhishu, &xishu);
        a2[zhishu] = xishu;
    }

    for(int i=1009;i>=0;i--){
        for(int j=1009;j>=0;j--){
            multiply[i+j] = a1[i] * a2[j] + multiply[i+j];
        }
    }
    for(int i=2019;i>=0;i--){
        if(multiply[i] != 0) count++;
    }
    printf("%d", count);
    for(int i=2019;i>=0;i--){
        if(multiply[i] != 0) printf(" %d %.1f", i, multiply[i]);
    }
    return 0;
}
```

4、Shuffling Machine

Shuffling is a **procedure** used to **randomize** a **deck** of playing cards.

Because standard shuffling techniques **are seen as weak**, and in order to avoid "**inside jobs**" where employees collaborate with gamblers by performing **inadequate** shuffles, many **casinos** employ **automatic shuffling machines**. Your task is to **simulate** a shuffling machine.

The machine shuffles a deck of 54 cards according to a given random order and repeats for a given number of times. It is assumed that the initial status of a card deck is in the following order:

```
S1, S2, ..., S13,       // 黑桃
H1, H2, ..., H13,       // 红桃
C1, C2, ..., C13,       // 梅花
D1, D2, ..., D13,       // 方块
J1, J2                  // 大小王
```

where "S" stands for "Spade", "H" for "Heart", "C" for "Club", "D" for "Diamond", and "J" for "Joker".

A given order is a **permutation** of **distinct integers** in [1, 54].

If the number at the $i-th$ position is j, it means to **move the card from position *i* to position *j***. For example, suppose we only have 5 cards: S3, H5, C1, D13 and J2. Given a shuffling order {4, 2, 5, 3, 1}, the result will be: J2, H5, D13, S3, C1. If we are to repeat the shuffling again, the result will be: C1, H5, S3, J2, D13.

Input Specification:

Each input file contains one test case. For each case, the first line contains a positive integer *K* (≤20) which is **the number of repeat times**.

Then the next line contains **the given order**. All the numbers in a line are separated by a space.

Output Specification:

For each test case, print the shuffling results in one line. All the cards are separated by a space, and there must be no extra space at the end of the line.

Sample Input:

```
2
36 52 37 38 3 39 40 53 54 41 11 12 13 42 43 44 2 4 23 24 25 26 27 6 7 8 48
49 50 51 9 10 14 15 16 5 17 18 19 1 20 21 22 28 29 30 31 32 33 34 35 45 46
47
```

Sample Output:

```
S7 C11 C10 C12 S1 H7 H8 H9 D8 D9 S11 S12 S13 D10 D11 D12 S3 S4 S6 S10 H1 H2
C13 D2 D3 D4 H6 H3 D13 J1 J2 C1 C2 C3 C4 D1 S5 H5 H11 H12 C6 C7 C8 C9 S2 S8
S9 H10 D5 D6 D7 H4 H13 C5
```

```
int main(){
    int repeat_times, time;
    int order[54] = {0};
    char *cards[54] = { "S1", "S2", "S3", "S4", "S5", "S6", "S7", "S8",
"S9", "S10", "S11", "S12", "S13",
                        "H1", "H2", "H3", "H4", "H5", "H6", "H7", "H8",
"H9", "H10", "H11", "H12", "H13",
                        "C1", "C2", "C3", "C4", "C5", "C6", "C7", "C8",
"C9", "C10", "C11", "C12", "C13",
                        "D1", "D2", "D3", "D4", "D5", "D6", "D7", "D8",
"D9", "D10", "D11", "D12", "D13",
                        "J1", "J2" };

    char *shuffling_order[54];
    scanf("%d", &repeat_times);
    for(int i=0;i<54;i++){
        scanf("%d", &order[i]);
    }
    for(int i=0;i<repeat_times;i++){
        if(i%2 ==0){
            for(int j=0;j<54;j++){
                shuffling_order[order[j]-1] = cards[j];       // 第1、3、
5...次
            }
        }
```

```
        else{
            for(int j=0;j<54;j++){
                cards[order[j]-1] = shuffling_order[j];       // 第2、4、
6...次
            }
        }
    }
    if(repeat_times%2 ==0){     // 共进行偶数次，最后结果存储于cards
        for(int i=0;i<54;i++){
            if(i != 53) cout<<cards[i]<<' ';
            else cout<<cards[i]<<endl;
        }
    }
    else{                       // 共进行奇数次，最后结果存储于
shuffling_order
        for(int i=0;i<54;i++){
           if(i != 53) cout<<shuffling_order[i]<<' ';
            else cout<<shuffling_order[i]<<endl;
        }
    }
    return 0;
}
```

5、Shortest Distance

The task is really simple： given *N* **exits** on a highway which forms a simple cycle, you are supposed to tell the shortest distance between any pair of exits.

Input Specification:

Each input file contains one test case.

For each case, **the first line** contains an integer *N* (in [3,10^5]), followed by *N* integer distances $D_1 D_2 \cdots D_N$, where D_i is the distance between the *i*-th and the (*i*+1)-st exits, and D_N is between the *N*-th and the 1st exits. All the numbers in a line are separated by a space.

The second line gives a positive integer *M* (≤10^4), with *M* lines follow, each contains a pair of exit numbers, provided that the exits are numbered from 1 to *N*. It is **guaranteed** that the total round trip distance is **no more than** 10^7.

Output Specification:

For each test case, print your results in *M* lines, each contains the shortest distance between the corresponding given pair of exits.

Sample Input:

```
5 1 2 4 14 9
3
1 3
2 5
4 1
```

Sample Output:

```
3
10
7
```

一种暴力解决办法：（可能导致运行超时，超过200ms）

```
#include <iostream>
using namespace std;

void MinDist(int inexit, int outexit, int distance[],int len){
    //此处不能用 int len = sizeof(distance) / sizeof(distance[0]); 计算数组长度
    //此时distance是一个指向distance数组首地址的指针，sizeof(distance)将返回指针的大小，而不是数组的大小

    int ij=0, ji=0;
    for(int k=inexit;k<outexit;k++){
        ij += distance[k];    //计算从入口i到出口j的距离
    }
    int k = outexit;
    while(k != inexit){
        ji += distance[k];    //计算从出口j到入口i的距离
        k = (k+1)%len;
    }
    printf("%d\n", (ij < ji ? ij : ji));
}

 int main(){
     int n,m;
     scanf("%d", &n);
     int distance[n];    //distance[i],表示从出口i,到出口i+1的距离
     for(int i=0;i<n;i++){
         scanf("%d", &distance[i]);
     }
     int len = sizeof(distance) / sizeof(distance[0]);    //计算总共有多少个同一出入口

     scanf("%d", &m);
     int pairs[m][2];    //要求的出口对距离
     for(int i=0;i<m;i++){
         scanf("%d%d", &pairs[i][0], &pairs[i][1]);
         pairs[i][0]--;  //与distance中的下标对齐
```

```
            pairs[i][1]--;
        }

        for(int i=0;i<m;i++){
            int inexit = pairs[i][0], outexit = pairs[i][1];
            if(inexit < outexit) MinDist(inexit, outexit, distance, n);
            else if(inexit > outexit) MinDist(outexit, inexit, distance, n);
            else printf("0\n");     //同一出入口，距离为0;
        }
    }
```

对上述代码进行优化：

- 使用 `sum` 记录总距离；
- 使用 `dist[]` 数组记录从0号结点到 i 号结点的距离；
- 比较 `temp = dist[left-1] - dist[right-1]` 与 `sum - temp` 的大小（环状结构）；

```
// -*- coding: utf-8 -*-
// @Author      : quanchenliu
// @Time        : 2024/4/9
// @Function    : 求最短距离
#include <iostream>
#include <algorithm>
using namespace std;

int main(){
    int n, m, inexit, outexit, sum=0;
    scanf("%d", &n);
    int a[n+1]={0}, dist[n+1]={0};    //distance[i],表示从出口i,到出口i+1的距离
    for(int i=1;i<=n;i++){
        scanf("%d", &a[i]);
        sum += a[i];        //累计总距离
        dist[i] = sum;      //dist[i]记录从0 号结点到 (i+1)%n 号结点的距离
    }

    scanf("%d", &m);
    for(int i=0;i<m;i++){
        scanf("%d%d", &inexit, &outexit);
        if(inexit > outexit) swap(inexit, outexit);
        int temp = dist[outexit-1] - dist[inexit-1];    //与distance中的下标对齐
        printf("%d\n", min(temp, sum-temp));
    }
    return 0;
}
```

6、A+B and C (64bit)

Given three integers *A*, *B* and *C* in $(-2^{63}, 2^{63})$, you are supposed to tell whether *A*+*B*>*C*.

Input Specification:

The first line of the input gives the positive number of test cases, *T* (≤10).

Then *T* test cases follow, each consists of a single line containing three integers *A*, *B* and *C*, separated by single spaces.

Output Specification:

For each test case, output in one line `Case #X: true` if *A*+*B*>*C*, or `Case #X: false` otherwise, where *X* is the case number (starting from 1).

Each line should ends with `'\n'`.

Sample Input:

```
3
1 2 3
2 3 4
9223372036854775807 -9223372036854775808 0
```

Sample Output:

```
Case #1: false
Case #2: true
Case #3: false
```

- 显然 long long 类型的数据 a、b 的表示范围为-2^{63}~$2^{63}-1$，故而可以使用 long long 类型来表示 a、b；
- 但是，a+b的结果会发生溢出：
  - 正溢出：a>0、b>0、a+b<0；
  - 负溢出：a<0、b<0、a+b>0；
- **为什么不能使用 double 类型的数据来进行处理呢？**很可能会损失精度。

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
    int n,i=1;
    long long a,b,c;
    scanf("%d", &n);
    while(scanf("%lld%lld%lld", &a, &b, &c) != EOF){
        long long  sum = a+b;
        bool flag;
        if(a>0 && b>0 && sum<0) flag = true;           //正溢出
        else if(a<0 && b<0 && sum>0) flag = false;     //负溢出
```

```
        else{
            if(sum > c) flag = true;
            else flag = false;
        }
        if(flag) printf("Case #%d: true\n", i);
        else printf("Case #%d: false\n", i);
        i++;
    }
    return 0;
}
```