Part II 一些数学题

一、完数VS盈数

> 一个数如果恰好等于它的各因子(该数本身除外)之和，如：6=3+2+1。则称其为“完数”；若因子之和大于该数，则称其为“盈数”。 求出2到60之间所有“完数”和“盈数”。

输入描述:

> 题目没有任何输入。

输出描述:

> 输出2到60之间所有“完数”和“盈数”，并以如下形式输出： E: e1 e2 e3 ……(ei为完数) G: g1 g2 g3 ……(gi为盈数) 其中两个数之间要有空格，行尾不加空格。

完整代码实现：

```
#include <stdio.h>
#define NUM 60
int judge(int num) {
    int i, sum = 0;
    for(i = 1; i < num; i++) {
        if(num % i == 0) {
            sum += i;
        }
    }
    if(sum == num) {
        return 1;           // 相等为完数
    } else if(sum > num) {
        return 0;           // 大于为盈数
    } else {
        return -1;          // 其他情况
    }
}

int main() {
    int e[NUM], g[NUM];
    int ecount = 0, gcount = 0;
    int i;

    // 判断2 - 60
    for(i = 2; i <= 60; i++) {
        if(judge(i) == 1) {
            e[ecount++] = i;
        } else if(judge(i) == 0) {
```

```
            g[gcount++] = i;
        }
    }

    //输出
    printf("E:");
    for(i = 0; i < ecount; i++) {
        printf(" %d", e[i]);
    }
    printf("\n");

    printf("G:");
    for(i = 0; i < gcount; i++) {
        printf(" %d", g[i]);
    }
    printf("\n");
    return 0;
}
```

二、N的阶乘

> 输入一个正整数N，输出N的阶乘。

输入描述:

> 正整数$N(0 \leq N \leq 1000)$

输出描述:

> 输入可能包括多组数据，对于每一组输入数据，输出N的阶乘

示例1 输入

> 4 5 15

输出

> 24 120 1307674368000

```
#include<stdio.h>
#define MaxSize 500
#define MOD 1000000          //表示data数组中每个元素存储6位
typedef struct {
    int data[MaxSize];
    int len;
} BigInt;
BigInt multiply(BigInt x, int y) {
    int i, g = 0, z;                // g表示进位，z暂存中间结果
    for(i = 0; i < x.len; i++) {
```

```
        z = x.data[i] * y + g;
        x.data[i] = z % MOD;          // 得到向高位进位后的结果
        g = z / MOD;                  // 得到进位
    }
    while(g) {
        x.data[i++] = g % MOD;        // 向最高位进位
        g /= MOD;
    }
    x.len = i;
    return x;
}
int main() {
    int i, n;
    BigInt r;
    while(scanf("%d", &n) != EOF) {
        r.data[0] = 1;
        r.len = 1;
        for(i = 2; i <= n; i++) {
            r = multiply(r, i);
        }
        //输出最高位
        printf("%d", r.data[r.len - 1]);
        //从后往前输出，每个元素输出六位，类似于计算机中的小端字节序存储数据
        for(i = r.len - 2; i >= 0; i--) {
            printf("%06d", r.data[i]);
        }
        printf("\n");
    }
    return 0;
}
```

三、矩阵幂

给定一个n*n的矩阵，求该矩阵的k次幂，即P^k。 **输入描述:**

> 第一行：两个整数n（$2 \leq n \leq 10$）、k（$1 \leq k \leq 5$），两个数字之间用一个空格隔开，含义如上所示。 接下来有n行，每行n个正整数，其中，第i行第j个整数表示矩阵中第i行第j列的矩阵元素P_{ij}且（$0 \leq P_{ij} \leq 10$）。另外，数据保证最后结果不会超过10^8。

输出描述:

> 对于每组测试数据，输出其结果。格式为： n行n列个整数，每行数之间用空格隔开，注意，每行最后一个数后面不应该有多余的空格。

示例1 输入

```
2 2 9 8 9 3
```

输出

```
153 96 108 81
```

完整代码实现：

```
#include <stdio.h>
#define N 11

 //求result*a，结果存入result
void matrixPow(int result[N][N], int a[N][N], int n) {
    int i, j, k;
    int c[N][N];        //用来暂时保存结果
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            c[i][j] = 0;
        }
    }
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            for(k = 0; k < n; k++) {
                c[i][j] += result[i][k] * a[k][j];
            }
        }
    }
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            result[i][j] = c[i][j];
        }
    }
}

int main() {
    int i, j;
    int n, k;
    int a[N][N], result[N][N];
    while(scanf("%d %d", &n ,&k) != EOF) {
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                scanf("%d", &a[i][j]);
                result[i][j] = a[i][j];
            }
        }
        for(i = 1; i < k; i++) {
            matrixPow(result, a, n);
        }
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                printf("%d", result[i][j]);
                if(j == n - 1) {
                    printf("\n");
                } else {
                    printf(" ");
```

```
                }
            }
        }
    }
    return 0;
}
```

时间复杂度为$O(k*n^3)$ 利用快速幂的思想，稍加优化：

```
#include <stdio.h>
#define N 11

 //求result*a, 结果存入result
void matrixPow(int result[N][N], int a[N][N], int n) {
    int i, j, k;
    int c[N][N];        //用来暂时保存结果
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            c[i][j] = 0;
        }
    }
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            for(k = 0; k < n; k++) {
                c[i][j] += result[i][k] * a[k][j];
            }
        }
    }
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            result[i][j] = c[i][j];
        }
    }
}

int main() {
    int i, j;
    int n, k;
    int a[N][N], result[N][N];
    while(scanf("%d %d", &n ,&k) != EOF) {
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                scanf("%d", &a[i][j]);
                if (i == j) {
                    result[i][j] = 1;       // 初始化为单位矩阵
                } else {
                    result[i][j] = 0;
                }
            }
        }
        while (k) {
            if (k & 1) {
```

```
                matrixPow(result, a, n);
            }
            matrixPow(a, a, n);
            k >>= 1;
        }
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                printf("%d", result[i][j]);
                if(j == n - 1) {
                    printf("\n");
                } else {
                    printf(" ");
                }
            }
        }
    }
    return 0;
}
```

时间复杂度为$O(log_2 k * n^3)$

小结：模拟题考点主要在思路，代码能力以及细节应对能力，其实一些与数学相关的题目，在机试题目中很多也是对模拟思路的考查，在平时的练习中，要针对性的练习一些比较复杂的模拟题，关于数学部分，主要是与质数相关的考查以及上述题目中的一些知识点。

Part III C++标准模板库

一、vector的常用用法

vector也被称为“变长数组”，也即“长度根据需要而自动改变的数组”。在考试题中，有时会碰到只用普通数组会超内存的情况，这种情况使用vector会让问题的解决便捷许多。另外，vector还可以用来以邻接表的方式储存图，这对无法使用邻接矩阵的题目(结点数太多)、又害怕使用指针实现邻接表的读者是非常友好的，写法也非常简洁。

如果要使用vector，则需要添加vector头文件，即 `#include <vector>` 。除此之外，还需要 在头文件下面加上一句 “using namespace std;", 这样就可以在代码中使用vector了。下 面介绍一些vector的常用用法。

1.vector的定义

单独定义一个vector：

```
vector <typename> name;
```

上面这个定义其实相当于是一维数组name[SIZE]，只不过其长度可以根据需要进行变化，比较节省空间，说通俗了就是“变长数组”。 和一维数组一样，这里的typename可以是任何基本类型，例如int、double、 char、结构 体等，也可以是STL标准容器，例如vector、set、 queue 等。**需要注意的是，如果typename也是一个STL容器，定义的时候要记得在>>符号之间加上空格，因为一些使用C++ 11之前标准的编译器会把它视为移位操作，导致编译错误。** 下面是一些简单的例子:

```
vector<int> name ;
vector<double> name ;
vector<char> name;
vector<node> name; / /node是结构体的类型
```

如果typename是vector,就是下面这样定义:

```
vector<vector<int> > name; //>>之间要加空格
```

可以很容易联想到二维数组的定义，即其中一维是一个数组的数组。那么vector数组也是一样，即Arrayname[]中的每一个元素都是一个vector。初学者可以**把vector数组当作两个 维都可变长的二维数组理解。** 然后来看定义vector数组的方法:

```
vector<typename> Arrayname [arraySize];
```

例如

```
vector<int> vi [100] ;
```

这样 `Arrayname[0] ~ Arrayname[arraySize - 1]` 中每一个都是一个 vector 容器。 与vector<vector > name不同的是，这种写法的一维长度已经固定为arraySize，另一 维才是“变长”的(注意体会这两种写法的区别)。

2. vector容器内元素的访问

vector一般有两种访问方式：通过下标访问或通过迭代器访问。下面分别讨论这两种访 问方式。 **(1)通过下标访问** 和访问普通的数组是一样，对一个定义为vector<typename> vi的vector容器来说，直接 访问vi[index]即可(如vi[0]、vi[1])。当然，这里下标是从0到vi.size() - 1，访问这个范围外的元素可能会运行出错。

(2)通过迭代器访问 迭代器(iterator)可以理解为一种类似指针的东西，其定义是:

```
vector<typename>::iterator it;
```

这样it就是一个vector<typename>::iterator型的变量，其中typename就是定义vector 时填写的类型。下面是typename为int和double的举例:

```
vector<int>::iterator it;
vector<double>::iterator it;
```

这样就得到了迭代器it，并且可以通过*it来访问vector里的元素。 例如，有这样定义的一个vector容器:

```
vector<int> vi;
for(int i = 1;i <= 5; i++) {      //循环完毕后 vi中元素为1 2 3 4 5
    vi.push_back(i); //push_back(i) 在vi的末尾添加元素i，即依次添加1 2 3 4 5
}
```

需要注意的是， vi[i]和*(vi.begin() + i)是等价的。 begin()函数的作用为取vi的首元素地址，那么这里还要提到end()函数。和begin()不同的是， **end()并不是取vi的尾元素地址，而是取尾元素地址的下一个地址。** end()作为迭代器末尾标志，不储存任何元素。美国人思维比较习惯左闭右开，在这里begin()和end()也是如此。

3.vector常用函数实例解析 (1).push_back() 顾名思义，push_back(x)就是在vector后面添加一个元素x，时间复杂度为0(1)。

(2).pop_back() 有添加就会有删除，pop_back()用以删除vector的尾元素，时间复杂度为0(1)。

(3).size() size()用来获得vector中元素的个数，时间复杂度为0(1)。size()返回的是unsigned 类型，不过一般来说用%d不会出很大问题，这一点对所有STL容器都是一样的。

(4).clear() clear()用来清空vector中的所有元素，时间复杂度为O(N),其中N为vector中元素的个 数。

(5).insert() insert(it, x)用来向vector的任意迭代器it处插入一个元素x，时间复杂度O(N)。

(6).erase() erase()有两种用法:删除单个元素、删除一个区间内的所有元素。时间复杂度均为O(N)。 1.删除单个元素。 erase(it)即删除迭代器为it处的元素。

2.删除一个区间内的所有元素。 erase(first, last)即删除[first, last)内的所有元素。 如果要删除这个vector内的所有元素，正确的写法应该是

```
vi.erase(vi.begin(), vi.end())
```

当然，更方便的清空vector的方法是使用vi.clear()。

一个完整的例子：

```
#include <vector>
#include <iostream>

using namespace std;

int main() {
    vector<int> vi;
    for(int i = 0;i < 10;i++) {
        vi.push_back(i);
    }

    // 访问方法
    cout << "直接利用数组: ";
    for(int i = 0;i < 10;i++) {
        cout<< vi[i] << " ";
    }
    cout << endl;

    cout << "利用迭代器: " ;
    vector<int>::iterator it;
    for(it = vi.begin();it != vi.end();it++) {
        cout<< *it << " ";
    }
    cout << endl;

    for(int i = 0;i < 5;i++) {
        vi.pop_back();
    }

    for(int i = 0;i < vi.size();i++) {
```

```
        cout << vi[i] << " ";
    }
    cout << endl;

    vi.insert(vi.begin() + 2, -1);              //将-1插入vi[2]的位置
    for(int i = 0;i < vi.size();i++) {
        cout << vi[i] << " ";
    }
    cout << endl;

    vi.erase(vi.begin() + 2);                   //删除-1
    for(int i = 0;i < vi.size();i++) {
        cout << vi[i] << " ";
    }
    cout << endl;

    vi.erase(vi.begin(), vi.begin() + 2);                   //删除0, 1
    for(int i = 0;i < vi.size();i++) {
        cout << vi[i] << " ";
    }
    cout << endl;

    vi.clear();
    cout << vi.size() << endl;

    return 0;
}
```

输出结果：

```
直接利用数组: 0 1 2 3 4 5 6 7 8 9
利用迭代器: 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4
0 1 -1 2 3 4
0 1 2 3 4
2 3 4
0
```

4.vector的常见用途 (1)储存数据 ①vector本身可以作为数组使用，而且一些元素个数不确定的场合可以很好地节省空 间。 ②有些场合需要根据一些条件把部分数据输出在同一行, 数据中间用空格隔开。由于输 出数据的个数是不确定的，为了更方便地处理最后一个满足条件的数据后面不输出额外的空格，可以先用vector记录所有需要输出的数据，然后一次性输出。 (2)用邻接表存储图 使用vector实现邻接表可以避免使用指针。

二、set的常用用法

set翻译为集合，是**一个内部自动有序且不含重复元素的容器**。在机试中，有可能出现需 要去掉重复元素的情况，而且有可能因这些元素比较大或者类型不是int型而不能直接开散列表，在这种情况下就可以用set来保留元素本身而不考虑它的个数。当然，上面说的情况也可以通过再开一个数组进行下标和元素的对应来解决，但是set提供了更为直观的接口，并且加入set之后可以实现自动排序，因此熟练使用set可以在做某些题时减少思维量。

如果要使用set,需要添加set头文件，即#include <set>. 除此之外，还需要在头文件下 面加上一句:“using namespace std;",这样就可以在代码中使用set了。

1.set的定义 单独定义一个set:

```
set<typename> name;
```

其定义的写法其实和vector基本是一样的，或者说其实大部分STL都是这样定义的。这 里的typename依然可以是任何基本类型，例如int、 double、char、结构体等，或者是STL标 准容器，例如vector、set、queue 等。 和前面vector中提到的一样，如果**typename是-一个STL容器，那么定义时要记得在>> 符号之间加上空格，** 因为一些使用C++ 11之前标准的编译器会把它视为移位操作，导致编译错误。下面是一些简单的例子:

```
set<int> name;
set<double> name ;
set<char> name ;
set<node> name;//node 是结构体的类型
```

set数组的定义和vector相同:

```
set<typename> Arrayname[arraySize];
```

例如

```
set<int> a[100];
```

这样Arrayname[0] ~ Arrayname[arraySize - 1]中的每一个都是一个set容器。

2.set容器内元素的访问 set只能通过迭代器(iterator)访问:

```
set<typename>::iterator it;
```

typename就是定义set时填写的类型，下面是typename为int和char的举例:

```
set<int>::iterator it;
set<char>::iterator it;
```

这样就得到了迭代器it,并且可以通过*it来访问set里的元素。 由于**除开vector和string之外的STL容器都不支持*(it + i)的访问方式**，因此只能按如 下方式枚举:

```
// 注意，不支持it < st.end()的写法
for (set<int>::iterator it = st.begin(); it != st.end(); it++) {
    ...
}
```

注意： **set内的元素自动递增排序，并且自动去除了重复元素。 3.set常用函数实例 (1).insert()** insert(x)可将x插入set容器中，并自动递增排序和去重，时间复杂度O(logN)，其中N 为set内的元素个数。

(2).find() find(value)返回set中对应值为value的迭代器，时间复杂度为O(logN)，N为set内的元 素个数。

(3).erase() erase()有两种用法：删除单个元素、删除一个区间内的所有元素。 ①删除单个元素。 删除单个元素有两种方法:

- st.erase(it)，it为所需要删除元素的迭代器。时间复杂度为0(1)。可以结合find()函数 来使用。

- st.erase(value)，value为所需要删除元素的值。时间复杂度为O(logN)， N为set内的元素个数。

②删除一个区间内的所有元素。 st.erase(first, last)可以删除一个区间内的所有元素，其中first为所需要删除区间的起始迭代器，而last则为所需要删除区间的末尾迭代器的下一个地址，也即为删除[first, last)。时间复杂度为O(last - first)。

(4).size() size()用来获得set内元素的个数，时间复杂度为0(1)。

(5).clear() clear()用来清空set中的所有元素，复杂度为O(N)，其中N为set内元素的个数。

一个完整的例子：

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

int main() {
    set<int> st;
    st.insert(100);
    st.insert(200);
    st.insert(100);
    st.insert(300);

    for (set<int>::iterator it = st.begin(); it != st.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;

    set<int>::iterator it = st.find(200);
    cout << *it << endl;

    st.erase(st.find(100));
    for (set<int>::iterator it = st.begin(); it != st.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;

    st.erase(200);
    for (set<int>::iterator it = st.begin(); it != st.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;

    st.insert(20);
    st.insert(10);
    st.insert(30);
    st.insert(40);
    it = st.find(30);
    st.erase(it, st.end());

    for (set<int>::iterator it = st.begin(); it != st.end(); it++) {
        cout << *it << " ";
    }
```

```
    cout << endl;

    st.clear();
    cout << st.size() << endl;

    return 0;
}
```

输出结果：

```
100 200 300
200
200 300
300
10 20
0
```

3.set 的常见用途 set最主要的作用是自动去重并按升序排序，因此碰到需要去重但是却不方便直接开数组 的情况，可以尝试用set解决。 延伸: set 中元素是唯一的，如果需要处理不唯一的情况，则需要使用multiset。另外，C++ 11标准中还增加了unordered set，以散列代替set内部的红黑树(Red Black Tree, 一种 自平衡二叉查找树)实现，使其可以用来处理只去重但不排序的需求，速度比set要快得多， 有兴趣的读者可以自行了解，此处不多作说明。

三、string的常用用法

在C语言中，一般使用字符数组char str[]来存放字符串，但是使用字符数组有时会显得操作麻烦，而且容易因经验不足而产生一些错误。为了使编程者可以更方便地对字符串进行操作，C++在STL中加入了string 类型，对字符串常用的需求功能进行了封装，使得操作起来更方便，且不易出错。 如果要使用string，需要添加string头文件，即#include <string> (注意string.h和string 是不一样的头文件)。除此之外，还需要在头文件下面加上一句：“using namespace std;"，这样就可以在代码中使用string了，下面来看string的一些常用用法。

1.string 的定义 定义string的方式跟基本数据类型相同，只需要在string后跟上变量名即可：

```
string str;
```

如果要初始化，可以直接给string类型的变量进行赋值：

```
string str = "abcd";
```

2.string 中内容的访问 (1)通过下标访问 一般来说，可以直接像字符数组那样去访问string；而**如果要读入和输出整个字符串，则只能用cin和cout。**

(2)通过迭代器访问 一般仅通过(1)即可满足访问的要求，但是有些函数比如insert()与erase()则要求以迭代器为参数，因此还是需要学习一下string迭代器的用法。 由于string不像其他STL容器那样需要参数，因此可以直接如下定义：

```
string::iterator it;
```

这样就得到了迭代器it，并且可以通过*it来访问string里的每一位。

string和vector一样，支持直接对迭代器进行加减某个数字，如str.begin()+3的写法是可行的。

3.string常用函数 事实上，string的函数有很多，但是有些函数并不常用，因此下面就几个常用的函数举例。 **(1) operator+=(string重载了+=运算符)** 这是string的加法，可以将两个string直接拼接起来。

(2) compare operator 两个string类型可以直接使用一、!=、 <、<=、>、>=比较大小，比较规则是字典序。

(3) length()/size() length()返回string的长度，即存放的字符数，时间复杂度为O(1)。size()和length()基本相同。

(4) insert() string的insert()函数有很多种写法，这里给出几个常用的写法，时间复杂度为O(N)。 ①insert(pos, string)，在pos号位置插入字符串string。 ②insert(it, it2, it3)，it 为原字符串的欲插入位置，it2和it3为待插字符串的首尾迭代器，用来表示串[it2, it3)将被插在it的位置上。

(5) erase() erase()有两种用法：删除单个元素、删除一个区间内的所有元素。时间复杂度均为O(N)。 ①删除单个元素。 str.erase(it)用于删除单个元素，it为需要删除的元素的迭代器。 ②删除一个区间内的所有元素。 删除一个区间内的所有元素有两种方法：

- str.erase(first, last)，其中first为需要删除的区间的起始迭代器，而last则为需要删除的区间的末尾迭代器的下一个地址，也即为删除[first, last)。
- str.erase(pos, length)，其中pos为需要开始删除的起始位置，length为删除的字符个数。

(6) clear() clear()用以清空string中的数据，时间复杂度一般为0(1)。

(7) substr() substr(pos, len)返回从pos号位开始、长度为len的子串，时间复杂度为O(len)。

(8) find() str.find(str2)，当str2是str的子串时，返回其在str中第一次出现的位置；如果str2不是 str的子串，那么返回string::npos。

- 注意：string::npos是一个常数，其本身的值为-1，但由于是unsigned int类型，因此实际上也可以认为是unsigned int类型的最大值。string::npos用以作为find函数失配时的返回值。（经验证，可能跟操作系统的位数有关，但这个影响不大。）

str.find(str2, pos)，从str的pos号位开始匹配str2，返回值与上相同。 时间复杂度为O(nm)，其中n和m分别为str和str2的长度。

(9).replace() str.replace(pos, len, str2)把str从pos号位开始、长度为len的子串替换为str2。 str.replace(it1, it2, str2)把str的迭代器[it1, it2)范围的子串替换为str2。 时间复杂度为O(str.length())

一个完整的例子：

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

int main () {
    string str1 = "Hello";
    string str2 = "World";
    string str3;

    str3 = str1;
    cout << "str3 : " << str3 << endl;

    str3 = str1 + str2;
```

```
    cout << "str1 + str2 : " << str3 << endl;

    cout << "str3.length() :  " << str3.length() << endl;
    cout << "str3.size() :  " << str3.size() << endl;

    str3.insert(5, " ");
    cout << "str3 : " << str3 << endl;

    str3.insert(str3.begin() + 6, str1.begin(), str1.end());
    cout << "str3 : " << str3 << endl;

    str2.erase(str2.begin() + 3);
    cout << "str2 : " << str2 << endl;

    str2.erase(str2.begin() + 1, str2.end());
    cout << "str2 : " << str2 << endl;

    str3.erase(6, 5);
    cout << "str3 : " << str3 << endl;


    cout << str3.substr(0, 5) << endl;
    cout << str3.find("World") << endl;
    cout << str3.find("Word") << endl;
    cout << UINT_MAX << endl;

    cout << "str3 : " << str3 << endl;
    cout << str3.replace(0, 3, "Word") << endl;

    str3.clear();
    cout << "str3.length() :  " << str3.length() << endl;
    return 0;
}
```

输出结果：

```
str3 : Hello
str1 + str2 : HelloWorld
str3.length() :  10
str3.size() :  10
str3 : Hello World
str3 : Hello HelloWorld
str2 : Word
str2 : W
str3 : Hello World
Hello
6
18446744073709551615
4294967295
str3 : Hello World
Wordlo World
str3.length() :  0
```

四、map的常用用法

map为映射，也是常用的STL容器。众所周知，在定义数组时(如int array[100])， 其实是定义了一个从int型到int型的映射，比如array[0] = 25、array[4] = 36就分别是将0映射到25、将4映射到36。一个double 型数组则是将int 型映射到double型，例如db[0]= 3.14，db[1]=0.01。但是，无论是什么类型，它总是将int型映射到其他类型。这似乎表现出一个弊端：当需要以其他类型作为关键字来做映射时，会显得不太方便。例如有一本字典，上面提供了很多的字符串和对应的页码，如果要用数组来表示“字符串-->页码”这样的对应关系，就会感觉不太好操作。这时，就可以用到map，因为**map可以将任何基本类型(包括STL容器)映射到任何基本类型(包括STL容器)，也就可以建立string型到int型的映射。**

还可以来看一个情况：这次需要判断给定的一些数字在某个文件中是否出现过。按照正 常的思路，可以开一个bool型hashTable[max_size]，通过判断hashTable[x]为true还是false来确定x是否在文件中出现。但是这会碰到一个问题，如果这些数字很大(例如有几千位)， 那么这个数组就会开不了。而这时map就可以派上用场，因为可以把这些数字当成一些字符 串，然后建立string至int的映射(或是直接建立int至int的映射)。

如果要使用map，需要添加map头文件，即#include <map>。 除此之外，还需要在头文 件下面加上一句:“using namespace std;"，这样就可以在代码中使用map了。下面来看map 的一些常用用法。

1.map的定义 单独定义一个map：

```
map<typename1, typename2> mp;
```

map和其他STL容器在定义上有点不一样，因为map需要确定映射前类型(键key)和映射后类型(值value)，所以需要在<>内填写两个类型，其中第一个是键的类型，第二个是值的类型。如果是int型映射到int型，就相当于是普通的int型数组。

而如果是字符串到整型的映射，必须使用string而不能用char数组：

```
map<string, int> mp;
```

这是因为char数组作为数组，是不能被作为键值的。如果想用字符串做映射，必须用string. 前面也说到， map的键和值也可以是STL容器，例如可以将一个set容器映射到一个字符串：

```
map<set<int>, string> mp;
```

2.map容器内元素的访问 map一般有两种访问方式:通过下标访问或通过迭代器访问。下面分别讨论这两种访问 方式。 **(1)通过下标访问** 和访问普通的数组是一样的，例如对一个定义为map<char, int> mp的map来说，就可以 直接使用mp['c']的方式来访问它对应的整数。于是，当建立映射时，就可以直接使用mp['c']=20这样和普通数组-样的方式。**但是要注意的是，map中的键是唯一的。**

(2)通过迭代器访问 map迭代器的定义和其他STL容器迭代器定义的方式相同：

```
map<typename1, typename2>::iterator it;
```

typename1和typename2就是定义map时填写的类型，这样就得到了迭代器it。 map迭代器的使用方式和其他STL容器的迭代器不同，因为map的每一对映射都有两个typename，这决定了必须能通过一个it来同时访问键和值。事实上， **map可以使用it -> first来访问键，使用it -> second来访问值。**

map会以键从小到大的顺序自动排序，即按a<m<r的顺序排列这三对映射。这是由于map内部是使用红黑树实现的(set也是)，在建立映射的过程中会自动实现从小到大的排序功能。

3.map常用函数实例解析 (1) find() find(key)返回键为key的映射的迭代器，时间复杂度为O(logN)，N为map中映射的个数。

(2) erase() erase()有两种用法：删除单个元素、删除一个区间内的所有元素。①删除单个元素。 删除单个元素有两种方法：

- mp.erase(it)，it为需要删除的元素的迭代器。时间复杂度为0(1)。
- mp.erase(key)，key为欲删除的映射的键。时间复杂度为O(logN)，N为map内元素的个数。

②删除一个区间内的所有元素。 mp.erase(first, last)，其中first为需要删除的区间的起始迭代器，而last则为需要删除的区间的末尾迭代器的下一个地址，也即为删除左闭右开的区间[first, last)。时间复杂度为O(last - first)。

(3) size() size()用来获得map中映射的对数，时间复杂度为0(1)。

(4) clear() clear()用来清空map中的所有元素，复杂度为0(N)，其中N为map中元素的个数。

一个完整的例子：

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

int main() {
    map<char, int> mp;
    mp['a'] = 1;
    mp['b'] = 2;
    mp['c'] = 3;

    map<char, int>::iterator it = mp.find('b');
    cout << it -> first << " " << it -> second << endl << endl;

    mp.erase(it);
    for (map<char, int>::iterator it = mp.begin(); it != mp.end(); it++) {
        cout << it -> first << " " << it -> second << endl;
    }
    cout << endl;

    mp.erase('c');
    for (map<char, int>::iterator it = mp.begin(); it != mp.end(); it++) {
        cout << it -> first << " " << it -> second << endl;
    }
    cout << endl;

    mp['d'] = 4;
    mp['e'] = 5;
    for (map<char, int>::iterator it = mp.begin(); it != mp.end(); it++) {
        cout << it -> first << " " << it -> second << endl;
    }
    cout << endl;

    it = mp.find('d');
    mp.erase(it, mp.end());
```

```
    for (map<char, int>::iterator it = mp.begin(); it != mp.end(); it++) {
        cout << it -> first << " " << it -> second << endl;
    }
    cout << endl;

    cout << mp.size() << endl;

    mp.clear();
    cout << mp.size() << endl;
    return 0;
}
```

输出结果：

```
b 2

a 1
c 3

a 1

a 1
d 4
e 5

a 1

1
0
```

4.map的常见用途 ①需要建立字符(或字符串)与整数之间映射的题目，使用map可以减少代码量。 ②判断大整数或者其他类型数据是否存在的题目，可以把map当bool数组用。 ③字符串和字符串的映射也有可能会遇到。 延伸: map的键和值是唯一的，而如果一个键需要对应多个值，就只能用multimap。另外，C++ 11标准中还增加了unordered_map，以散列代替map内部的红黑树实现，使其可以用来处理只映射而不按key排序的需求，速度比map要快得多，有兴趣的读者可以自行了解，此处不多作说明。

五、queue的常用用法

queue为队列，在STL中主要则是实现了一个先进先出的容器。概念与数据结构中的队列一致。 **1.queue的定义** 要使用queue，应先添加头文件#include <queue>，并在头文件下面加上“using namespace std;"，然后就可以使用了。其定义的写法和其他STL容器相同，typename可以是任意基本数据类型或容器：

```
queue< typename > name;
```

2.queue容器内元素的访问 由于队列(queue)本身就是一种先进先出的限制性数据结构，因此在STL中**只能通过 front()来访问队首元素，或是通过back()来访问队尾元素。**

3.queue常用函数实例解析 (1) push() push(x)将x进行入队，时间复杂度为0(1)。

(2) front()、back() front()和back()可以分别获得队首元素和队尾元素，时间复杂度为0(1)。

(3) pop() pop()令队首元素出队，时间复杂度为0(1)。

(4) empty() empty()检测queue是否为空，返回true则空，返回false则非空。时间复杂度为0(1)。

(5) size() size()返回queue内元素的个数，时间复杂度为0(1)。

一个完整的例子：

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

int main() {
    queue<int> q;
    for (int i = 1; i <= 5; i++) {
        q.push(i);
    }
    cout << q.front() << " " << q.back() << endl;

    for (int i = 1; i <= 3; i++) {
        q.pop();
    }
    cout << q.front() << " " << q.back() << endl;

    if (q.empty()) {
        cout << "Empty" << endl;
    } else {
        cout << "Not Empty" << endl;
    }

    cout << q.size() << endl;
    return 0;
}
```

输出结果：

```
1 5
4 5
Not Empty
2
```

4.queue的常见用途 当需要实现广度优先搜索时，可以不用自己手动实现一个队列，而是用queue作为代替，以提高程序的准确性。 另外有一点注意的是，使用front()和pop()函数前，必须用empty()判断队列是否为空，否则可能因为队空而出现错误。 延伸: STL的容器中还有两种容器跟队列有关，分别是双端队列(deque)和优先队列 (priority_queue)，前者是首尾皆可插入和删除的队列，后者是使用堆实现的默认将当前队列最大元素置于队首的容器。

六、priority_queue的常用用法

priority_queue又称为优先队列，其底层是用堆来进行实现的。在优先队列中，队首元素 一定是当前队列中优先级最高的那一个。例如在队列有如下元素，且定义好了优先级：

```
桃子(优先级3)
梨子(优先级4)
苹果(优先级1)
```

那么出队的顺序为梨子(4)→桃子(3)→苹果(1)。 当然，可以在任何时候往优先队列里面加入(push)元素，而优先队列底层的数据结构堆 (heap)会随时调整结构，使得每次的队首元素都是优先级最大的。

关于这里的优先级则是规定出来的。例如上面的例子中，也可以规定数字越小的优先级 越大。

1.priority_queue的定义 要使用优先队列，应先添加头文件#include <queue>，并在头文件下面加上“using namespace std;"，然后就可以使用了。其定义的写法和其他STL容器相同，typename 可以是任意基本数据类型或容器：

```
priority_queue< typename > name;
```

2.priority_queue容器内元素的访问 和队列不一样的是，优先队列没有front()函数与back()函数，而**只能通过top()函数来访 问队首元素(也可以称为堆顶元素)**，也就是优先级最高的元素。

3.priority_queue常用函数 (1) push() push(x)将令x入队，时间复杂度为O(logN)，其中N为当前优先队列中的元素个数。

(2) top() top()可以获得队首元素(即堆顶元素)，时间复杂度为0(1)。

(3) pop() pop()令队首元素(即堆顶元素)出队，时间复杂度为O(logN)，其中N为当前优先队列中的元素个数。

(4) empty() empty()检测优先队列是否为空，返回true则空，返回false则非空。时间复杂度为0(1)。

(5) size() size()返回优先队列内元素的个数，时间复杂度为0(1)。

由于函数用法与队列类似，因此这里不累述了，主要讲解一下**priority_queue内元素优先级的设置。**

4.priority_queue内元素优先级的设置 如何定义优先队列内元素的优先级是运用好优先队列的关键，下面分别介绍基本数据类 型(例如int、double、 char) 与结构体类型的优先级设置方法。 **(1)基本数据类型的优先级设置** 此处指的基本数据类型就是int型、double 型、char型等可以直接使用的数据类型，优先队列对它们的优先级设置一般是数字大的优先级越高，因此队首元素就是优先队列内元素最 大的那个(如果char型，则是字典序最大的)。对基本数据类型来说，下面两种优先队列的 定义是等价的(以int型为例，注意最后两个>之间有一个空格)：

```
priority_queue<int> q;
priority_queue<int, vector<int>, less<int> > q;
```

可以发现，第二种定义方式的尖括号内多出了两个参数: 一个是vector<int>，另一个是 less<int>。其中vector<int>(也就是第二个参数)填写的是来承载底层数据结构堆(heap)的容器，如果第一个参数是double型或char型，则此处只需要填写vector<double>或vector<char>；而第三个参数less<int>则是对第一个参数的比较类， **less<int>表示数字大的优先级越大，而greater<int>表示数字小的优先级越大。** 因此，如果想让优先队列总是把最小的元素放在队首，只需进行如下定义:

```
priority_queue<int, vector<int>, greater<int> >q;
```

5.priority_queue的常见用途 priority_queue可以解决一些贪心问题 (例如9.8节)，也可以对Dijkstra算法进行优化(因 为优先队列的本质是堆)。 有一点需要注意， **使用top()函数前，必须用empty()判断优先队列是否为空，否则可能 因为队空而出现错误。**

七、stack的常用用法

stack为栈，是STL中实现的一个后进先出的容器，与数据结构中的栈概念一致。 **1.stack的定义** 要使用stack,应先添加头文件#include <stack>，并在头文件下面加上“using namespace std;"，然后就可以使用了。 其定义的写法和其他STL容器相同，typename 可以任意基本数据类型或容器：

```
stack< typename > name;
```

2.stack容器内元素的访问 由于栈(stack)本身就是一种后进先出的数据结构，在STL的stack中**只能通过top()来 访问栈顶元素。**

3.stack常用函数 (1) push() push(x)将x入栈，时间复杂度为O(1)。 **(2) top()** top()获得栈顶元素，时间复杂度为0(1)。 **(3) pop()** pop()用以弹出栈顶元素，时间复杂度为O(1)。 **(4) empty()** empty()可以检测stack内是否为空，返回true为空，返回false为非空，时间复杂度为0(1)。 **(5) size()** size()返回stack内元素的个数，时间复杂度为0(1)。

4.stack的常见用途 stack用来模拟实现一些递归,防止程序对栈内存的限制而导致程序运行出错。一般来说， 程序的栈内存空间很小，对有些题目来说，如果用普通的函数来进行递归，一旦递归层数过 深(不同机器不同，约几千至几万层)，则会导致程序运行崩溃。如果用栈来模拟递归算法的 实现，则可以避免这一方面的问题 (不过这种应用出现较少)。

八、algorithm 头文件下的常用函数

使用algorithm头文件，需要在头文件下加一行 “using namespace std;"才能正常使用。

1.max()、min()和 abs() max(x, y)和min(x, y)分别返回x和y中的最大值和最小值，且参数必须是两个(可以是 浮点数)。如果想要返回三个数x、y、z的最大值，可以使用max(x, max(y, z))的写法。 abs(x)返回x的绝对值。注意: x必须是整数，浮点型的绝对值请用math头文件下的fabs。

2.swap() swap(x, y)用来交换x和y的值。

3.reverse() reverse(it, it2)可以将数组指针在[it, it2)之间的元素或容器的迭代器在[it, it2)范围内的元素进行反转。

4.next_permutation() next_permutation()给出一个序列在全排列中的下一个序列。 示例如下：

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

int main() {
    int a[10] = {1, 2, 3};
    do {
        cout << a[0] << a[1] << a[2] << endl;
    } while (next_permutation(a, a + 3));
    return 0;
}
```

输出结果：

```
123
132
213
231
312
321
```

在上述代码中，使用循环是因为next_permutation在已经到达全排列的最后一个时会返回 false，这样会方便退出循环。而使用do...while语句而不使用while语句是因为序列123本 身也需要输出，如果使用while会直接跳到下一个序列再输出，这样结果会少一个123。

5.sort() 顾名思义，sort就是用来排序的函数，它根据具体情形使用不同的排序方法，效率较高。 一般来说，不推荐使用C语言中的qsort函数，原因是qsort用起来比较烦琐，涉及很多指针 的操作。而且sort在实现中避免了经典快速排序中可能出现的会导致实际复杂度退化到$0(n^2)$的极端情况。 **(1).如何使用sort排序** sort函数的使用必须加上头文件“include <algorithm>”和“using namespace std;"，其使 用的方式如下：

```
sort (首元素地址(必填),尾元素地址的下一个地址(必填),比较函数(非必填));
```

可以看到，sort的参数有三个，其中前两个是必填的，而比较函数则可以根据需要填写，如果不写比较函数，则默认对前面给出的区间进行递增排序。

(2).如何实现比较函数cmp 下面介绍对基本数据类型、结构体类型、STL容器进行自定义规则排序时cmp的写法。 示例如下：

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
bool cmp(int a, int b) {
    return a > b;
}

int main() {
    vector<int> vi;
    vi.push_back(3);
    vi.push_back(1);
    vi.push_back(2);
    sort(vi.begin(), vi.end(), cmp);
    for (int i = 0; i < 3; i++) {
        cout << vi[i] << " ";
    }
    cout << endl;
    return 0;
}
```

输出结果：

```
3 2 1
```

6.lower_bound()和upper_bound() **lower_bound()和upper_bound()需要用在一个有序数组或容器中。** lower_bound(first, last, val)用来寻找在数组或容器的[first,last)范围内**第一个值大于等于val的元素的位置**，如果是数组，则返回该位置的指针；如果是容器，则返回该位置的迭代器。 upper_bound(first, last, val)用来寻找在数组或容器的[first, last)范围内**第一个值大于val的元素的位置**，如果是数组，则返回该位置的指针；如果是容器，则返回该位置的迭代器。

显然，如果数组或容器中没有需要寻找的元素，则lower_bound()和upper_bound()均返回 可以插入该元素的位置的指针或迭代器(即假设存在该元素时，该元素应当在的位置)。 lower_bound()和upper_bound()的复杂度均为O(log(last - first))。

示例如下：

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


int main() {
    int a[10] = {1, 2, 2, 3, 3, 3, 5, 5, 5, 5};

    //寻找-1
    int *lowerPos = lower_bound(a, a + 10, -1);
    int *upperPos = upper_bound(a, a + 10, -1);
    cout << lowerPos - a << " " << upperPos - a << endl;

    //寻找1
    lowerPos = lower_bound(a, a + 10, 1);
    upperPos = upper_bound(a, a + 10, 1);
    cout << lowerPos - a << " " << upperPos - a << endl;

    //寻找3
    lowerPos = lower_bound(a, a + 10, 3);
    upperPos = upper_bound(a, a + 10, 3);
    cout << lowerPos - a << " " << upperPos - a << endl;

    //寻找4
    lowerPos = lower_bound(a, a + 10, 4);
    upperPos = upper_bound(a, a + 10, 4);
    cout << lowerPos - a << " " << upperPos - a << endl;

    //寻找5
    lowerPos = lower_bound(a, a + 10, 5);
    upperPos = upper_bound(a, a + 10, 5);
    cout << lowerPos - a << " " << upperPos - a << endl;

    //寻找6
    lowerPos = lower_bound(a, a + 10, 6);
    upperPos = upper_bound(a, a + 10, 6);
    cout << lowerPos - a << " " << upperPos - a << endl;
    return 0;
}
```

输出结果：

```
0 0
0 1
3 6
6 6
6 10
10 10
```