|  |
| --- |
| **Bryce1010-ACM模板** |
|  |
|  |
| **杨贤** |
|  |

目录

[1 STL标准模板库 1](#_Toc499159382)

[1.1 STL pair 1](#_Toc499159383)

[1.2 STL Set 2](#_Toc499159384)

[1.3 STL Vector 3](#_Toc499159385)

[1.4 STL String 5](#_Toc499159386)

[1.5 STL stack 6](#_Toc499159387)

[1.6 STL Queue 7](#_Toc499159388)

[1.7 STL Map 8](#_Toc499159389)

[1.8 lower\_bound和Upper\_bound 11](#_Toc499159390)

[2. 复杂的数论 12](#_Toc499159391)

[2.1 大数阶乘分割法 12](#_Toc499159392)

[2.2 大数阶乘长度Stirling公式 13](#_Toc499159393)

[2.3 欧拉函数 13](#_Toc499159394)

[2.4 快速幂 15](#_Toc499159395)

[2.5 GCD与LCM与ExtGcd 16](#_Toc499159396)

[2.6 素数测试（判断素数） 17](#_Toc499159397)

[2.7 求逆元 29](#_Toc499159398)

[.2.8 原根 32](#_Toc499159399)

[2.9 莫比乌斯函数求法 35](#_Toc499159400)

[2.10 ACM博弈 38](#_Toc499159401)

[2.11数论相关公式 43](#_Toc499159402)

[3. 有趣的字符串 44](#_Toc499159403)

[3.1 判断回文串的方法 44](#_Toc499159404)

[3.2 KMP匹配 46](#_Toc499159405)

[3.3 扩展KMP 50](#_Toc499159406)

[3.4 strstr函数 52](#_Toc499159407)

[4. 图论 52](#_Toc499159408)

[4.1 最小生成树的解法 52](#_Toc499159409)

[4.2 二维曼哈顿最小生成树 56](#_Toc499159410)

[5. 数据结构 62](#_Toc499159411)

[5.1 树状数组 区间求和 62](#_Toc499159412)

[5.2 线段树 64](#_Toc499159413)

[6. 搜索 71](#_Toc499159414)

[6.1 DFS 71](#_Toc499159415)

[6.2 BFS 72](#_Toc499159416)

[7. 动态规划 74](#_Toc499159417)

[7.1 最长公共子序列lcs 74](#_Toc499159418)

[7.2 最长上升子序列 75](#_Toc499159419)

[8．精选技巧 76](#_Toc499159420)

[8.1 矩阵快速幂 76](#_Toc499159421)

[8.2 矩阵运算 78](#_Toc499159422)

[8.4 数据类型的取值范围 81](#_Toc499159423)

[8.5 基本数学公式 82](#_Toc499159424)

## 1 STL标准模板库

### 1.1 STL pair

STL的<utility>头文件中描述了一个看上去非常简单的模板类pair，用来表示一个二元组或元素对，并提供了按照字典序对元素对进行大小比较的比较运算符模板函数。

例如，想要定义一个对象表示一个平面坐标点，则可以：

|  |
| --- |
| pair<double, double> p1;  cin >> p1.first >> p1.second; |

pair模板类需要两个参数：首元素的数据类型和尾元素的数据类型。pair模板类对象有两个成员：first和second，分别表示首元素和尾元素。

在<utility>中已经定义了pair上的六个比较运算符：<、>、<=、>=、==、!=，其规则是先比较first，first相等时再比较second，这符合大多数应用的逻辑。   
当然，也可以通过重载这几个运算符来重新指定自己的比较逻辑。

除了直接定义一个pair对象外，如果需要即时生成一个pair对象，也可以调用在<utility>中定义的一个模板函数：make\_pair。make\_pair需要两个参数，   
分别为元素对的首元素和尾元素。

|  |
| --- |
| Pair类型概述  pair是一种模板类型，其中包含两个数据值，两个数据的类型可以不同，基本的定义如下：    pair<int, string> a;  表示a中有两个类型，第一个元素是int型的，第二个元素是string类型的，如果创建pair的时候没有对其进行初始化，则调用默认构造函数对其初始化。    pair<string, string> a("James", "Joy");  也可以像上面一样在定义的时候直接对其初始化。    由于pair类型的使用比较繁琐，因为如果要定义多个形同的pair类型的时候，可以时候typedef简化声明：  typedef pair<string, string> author;  author pro("May", "Lily");  author joye("James", "Joyce"); |

|  |
| --- |
| Pair对象的操作    对于pair类，由于它只有两个元素，分别名为first和second，因此直接使用普通的点操作符即可访问其成员  pair<string, string> a("Lily", "Poly");  string name;  name = pair.second;  生成新的pair对象  可以使用make\_pair对已存在的两个数据构造一个新的pair类型：  int a = 8;  string m = "James";  pair<int, string> newone;  newone = make\_pair(a, m); |

### 1.2 STL Set

set模版类的定义在头文件<set>中。

定义set对象的示例代码如下：

|  |
| --- |
| set<int> s;  set<double> ss;  set的基本操作：  s.begin() // 返回指向第一个元素的迭代器  s.clear() // 清除所有元素  s.count() // 返回某个值元素的个数  s.empty() // 如果集合为空，返回true(真）  s.end() // 返回指向最后一个元素之后的迭代器，不是最后一个元素  s.equal\_range() // 返回集合中与给定值相等的上下限的两个迭代器  s.erase() // 删除集合中的元素  s.find() // 返回一个指向被查找到元素的迭代器  s.get\_allocator() // 返回集合的分配器  s.insert() // 在集合中插入元素  s.lower\_bound() // 返回指向大于（或等于）某值的第一个元素的迭代器  s.key\_comp() // 返回一个用于元素间值比较的函数  s.max\_size() // 返回集合能容纳的元素的最大限值  s.rbegin() // 返回指向集合中最后一个元素的反向迭代器  s.rend() // 返回指向集合中第一个元素的反向迭代器  s.size() // 集合中元素的数目  s.swap() // 交换两个集合变量  s.upper\_bound() // 返回大于某个值元素的迭代器  s.value\_comp() // 返回一个用于比较元素间的值的函数 |

**multiset**

在<set>头文件中，还定义了另一个非常实用的模版类multiset（多重集合）。多重集合与集合的区别在于集合中不能存在相同元素，而多重集合中可以存在。

定义multiset对象的示例代码如下：

|  |
| --- |
| multiset<int> s;  multiset<double> ss; |

multiset和set的基本操作相似，需要注意的是，集合的count()能返回0（无）或者1（有），而多重集合是有多少个返回多少个。

### 1.3 STL Vector

在STL的<vector>头文件中定义了vector（向量容器模版类），vector容器以连续数组的方式存储元素序列，可以将vector看作是以顺序结构实现的线性表。当我们在程序中需要使用动态数组时，vector将会是理想的选择，vector可以在使用过程中动态地增长存储空间。   
vector模版类需要两个模版参数，第一个参数是存储元素的数据类型，第二个参数是存储分配器的类型，其中第二个参数是可选的，如果不给出第二个参数，将使用默认的分配器。

下面给出几个常用的定义vector向量对象的方法示例：

|  |
| --- |
| vector<int> s;  // 定义一个空的vector对象，存储的是int类型的元素  vector<int> s(n);  // 定义一个含有n个int元素的vector对象  vector<int> s(first, last);  // 定义一个vector对象，并从由迭代器first和last定义的序列[first, last)中复制初值  vector的基本操作：  s[i] // 直接以下标方式访问容器中的元素  s.front() // 返回首元素  s.back() // 返回尾元素  s.push\_back(x) // 向表尾插入元素x  s.size() // 返回表长  s.empty() // 表为空时，返回真，否则返回假  s.pop\_back() // 删除表尾元素  s.begin() // 返回指向首元素的随机存取迭代器  s.end() // 返回指向尾元素的下一个位置的随机存取迭代器  s.insert(it, val) // 向迭代器it指向的元素前插入新元素val  s.insert(it, n, val)// 向迭代器it指向的元素前插入n个新元素val  s.insert(it, first, last)  // 将由迭代器first和last所指定的序列[first, last)插入到迭代器it指向的元素前面  s.erase(it) // 删除由迭代器it所指向的元素  s.erase(first, last)// 删除由迭代器first和last所指定的序列[first, last)  s.reserve(n) // 预分配缓冲空间，使存储空间至少可容纳n个元素  s.resize(n) // 改变序列长度，超出的元素将会全部被删除，如果序列需要扩展（原空间小于n），元素默认值将填满扩展出的空间  s.resize(n, val) // 改变序列长度，超出的元素将会全部被删除，如果序列需要扩展（原空间小于n），val将填满扩展出的空间  s.clear() // 删除容器中的所有元素  s.swap(v) // 将s与另一个vector对象进行交换  s.assign(first, last)  // 将序列替换成由迭代器first和last所指定的序列[first, last)，[first, last)不能是原序列中的一部分  // 要注意的是，resize操作和clear操作都是对表的有效元素进行的操作，但并不一定会改变缓冲空间的大小  // 另外，vector还有其他的一些操作，如反转、取反等，不再一一列举  // vector上还定义了序列之间的比较操作运算符（>、<、>=、<=、==、!=），可以按照字典序比较两个序列。 |

### 1.4 STL String

|  |
| --- |
| string对象的定义和初始化以及读写  string s1;      默认构造函数，s1为空串  string s2(s1);   将s2初始化为s1的一个副本  string s3("valuee");   将s3初始化一个字符串面值副本  string s4(n,'c');   将s4初始化为字符'c'的n个副本  cin>>s5;  读取有效字符到遇到空格  getline(cin,s6);  读取字符到遇到换行，空格可读入，知道‘\n’结束  getline(cin,s7,'a'); 一个直到‘a’结束，其中任何字符包括'\n'都能够读入 |

|  |
| --- |
| string对象中字符的处理（头文件cctype）     isalnum(c)  如果c是字母或数字，返回true      isalpha(c)  如果c是字母，返回true      iscntrl(c)  c是控制符，返回true      isdigit(c)  如果c是数组，返回true      isgraph(c)  如果c不是空格，则可打印，，则为true      islower(c)  如果c是小写字母，则为true      isupper(c)  如果c是大写字符，则为true      isprint(c)  如果c是可打印的字符，则为true      ispunct(c)  如果c是标点符号，则为true      isspace(c) 如果c是空白字符，则为true      isxdigit(c) 如果c是十六进制数，则为true      tolower(c) 如果c是大写字符，则返回其小写字母，否则直接返回c      toupper(c)  跟tolower相反 |

|  |
| --- |
| string对象中一些函数   /\*-------------------------插入函数----------------------------------包括迭代器操作和下标操作，下标操作更灵活\*/  s.insert( it , p );  把字符串p插入到it的位置  s.insert(p,n,t)；   迭代器p元素之前插入n个t的副本  s.insert(p,b,e);      迭代器p元素之前插入迭代器b到e之间的所有元素  s.insert(p,s2,poe2,len); 在下标p之前插入s2下标从poe2开始长度为len的元素  s.insert(pos,cp,len);  下标pos之前插入cp数组的前len个元素。  /\*-----------------------替换函数-------------------------------\*/  s.assign(b,e);  用迭代器b到e范围内的元素替换s  s.assign(n,t)；  用n个t的副本替换s  a.assign(s1,pos2,len);从s1的下标pos2开始连续替换len个。  s.replace ( 3 , 3 , " good " ) ;   从第三个起连续三个替换为good  s.substr(i,j)   截取s串中从i到j的子串  //string::npos  判断字符串是否结束  /\*-----------------------删除函数-----------------------------\*/  s.erase( 3 )||s.erase ( 0 , 4 ) ;  删除第四个元素或第一到第五个元素  /\*----------------------其他函数-----------------------------\*/  s.find ( " cat " ) ;  超找第一个出现的字符串”cat“，返回其下标值，查不到返回4294967295，也可查找字符；  s.append(args); 将args接到s的后面  s.compare ( " good " ) ;  s与”good“比较相等返回0，比"good"大返回1，小则返回-1；  reverse ( s.begin(), s.end () );  反向排序函数，即字符串反转函数 |

### 1.5 STL stack

tack模版类的定义在<stack>头文件中。   
stack模版类需要两个模版参数，一个是元素类型，另一个是容器类型，但是只有元素类型是必要的，在不指定容器类型时，默认容器的类型为deque。

定义stack对象的示例代码如下：

|  |
| --- |
| stack<int> s;  stack<string> ss;  s.push(x); // 入栈  s.pop(); // 出栈  s.top(); // 访问栈顶  s.empty(); // 当栈空时，返回true  s.size(); // 访问栈中元素个数 |

### 1.6 STL Queue

queue模版类的定义在<queue>头文件中。   
queue与stack相似，queue模版类也需要两个模版参数，一个元素类型，一个容器类型，元素类型时必须的，容器类型时可选的，默认为deque类型。

定义queue对象的示例代码必须如下：

|  |
| --- |
| queue<int> q;  queue<double> qq;  q.push(x); // 入队列  q.pop(); // 出队列  q.front(); // 访问队首元素  q.back(); // 访问队尾元素  q.empty(); // 判断队列是否为空  q.size(); // 访问队列中的元素个数 |

**priority\_queue**

在<queue>头文件中，还定义了另一个非常有用的模版类priority\_queue（优先队列）。优先队列与队列的差别在于优先队列不是按照入队的顺序出队，而是按照队列中元素的优先权出队列（默认为大者优先，也可以通过指定算子来指定自己的优先顺序）。

priority\_queue模版类有三个模版参数，第一个是元素类型，第二个是容器类型，第三个是比较算子。其中后两者都可以忽略，默认容器为vector，默认算子为less，即小的往前排，大的往后排（出队列时列尾元素先出队）。   
定义priority\_queue对象的代码示例：

定义priority\_queue对象的代码示例：

|  |
| --- |
| priority\_queue<int> q;  priority\_queue<pair<int, int> > qq; // 注意在两个尖括号之间一定要留空格，防止误判  priority\_queue<int, vector<int>, greater<int> > qqq;// 定义小的先出队列  q.empty() // 如果队列为空，则返回true，否则返回false  q.size() // 返回队列中元素的个数  q.pop() // 删除队首元素，但不返回其值  q.top() // 返回具有最高优先级的元素值，但不删除该元素  q.push(item) // 在基于优先级的适当位置插入新元素 |

初学者在使用priority\_queue时，最困难的可能就是如何定义比较算子了。如果是基本数据类型，或已定义了比较运算符的类，可以直接使用STL的less算子和greater算子——默认为使用less算子。如果要定义自己的比较算子，方法有多种，这里介绍其中一种：重载比较运算符。优先队列试图这两个元素x和y代入比较运算符（对于less算子，调用x < y，对于greater算子，调用x > y），若结果为真，则x排在y前面，y将先出队列，反之，则y排在x前面，x将先出队列。

### 1.7 STL Map

在STL的头文件中<map>中定义了模版类map和multimap，用有序二叉树表存储类型为pair<const Key, T>的元素对序列。序列中的元素以const Key部分作为标识，map中所有元素的Key值必须是唯一的，multimap则允许有重复的Key值。

可以将map看作是由Key标识元素的元素集合，这类容器也被称为“关联容器”，可以通过一个Key值来快速决定一个元素，因此非常适合于需要按照Key值查找元素的容器。   
map模版类需要四个模版参数，第一个是键值类型，第二个是元素类型，第三个是比较算子，第四个是分配器类型。其中键值类型和元素类型是必要的。

定义map对象的代码示例：

|  |
| --- |
| map<string, int> m;  /\* 向map中插入元素 \*/  m[key] = value; // [key]操作是map很有特色的操作,如果在map中存在键值为key的元素对, 则返回该元素对的值域部分,否则将会创建一个键值为key的元素对,值域为默认值。所以可以用该操作向map中插入元素对或修改已经存在的元素对的值域部分。  m.insert(make\_pair(key, value)); // 也可以直接调用insert方法插入元素对,insert操作会返回一个pair,当map中没有与key相匹配的键值时,其first是指向插入元素对的迭代器,其second为true;若map中已经存在与key相等的键值时,其first是指向该元素对的迭代器,second为false。  /\* 查找元素 \*/  int i = m[key]; // 要注意的是,当与该键值相匹配的元素对不存在时,会创建键值为key（当另一个元素是整形时，m[key]=0）的元素对。  map<string, int>::iterator it = m.find(key); // 如果map中存在与key相匹配的键值时,find操作将返回指向该元素对的迭代器,否则,返回的迭代器等于map的end()(参见vector中提到的begin()和end()操作)。  /\* 删除元素 \*/  m.erase(key); // 删除与指定key键值相匹配的元素对,并返回被删除的元素的个数。  m.erase(it); // 删除由迭代器it所指定的元素对,并返回指向下一个元素对的迭代器。  /\* 其他操作 \*/  m.size(); // 返回元素个数  m.empty(); // 判断是否为空  m.clear(); // 清空所有元素 |

|  |
| --- |
| typedef map<int, string, less<int> > M\_TYPE  typedef M\_TYPE::iterator M\_IT  typedef M\_TYPR::const\_iterator M\_CIT  int main()  {  M\_TYPR myTestMap;  myTestMap[3] = "No.3";  myTestMap[5] = "No.5";  myTestMap[1] = "No.1";  myTestMap[2] = "No.2";  myTestMap[4] = "No.4";  M\_IT itStop = myTestMap.find(2);  cout << "myTestMap[2] = " << itStop->second << endl;  itStop->second = "No.2 After modification";  cout << "myTestMap[2] = " << itStop->second << endl;  cout << "Map contents:" << endl;  for (M\_CIT it = myTestMap.begin(); it != myTestMap.end(); it++)  {  cout << it->second << endl;  }  return 0;  }  int main()  {  map<string, int> m;  m["one"] = 1;  m["two"] = 2;  // 几种不同的 insert 调用方法  m.insert(make\_pair("three", 3));  m.insert(map<string, int>::value\_type("four", 4));  m.insert(pair<string, int>("five", 5));  string key;  while (cin >> key)  {  map<string, int>::iterator it = m.find(key);  if (it == m.end())  {  cout << "No such key!" << endl;  }  else  {  cout << key << " is " << it->second << endl;  cout << "Erased " << m.erase(key) << endl;  }  }  return 0;  } |

### 1.8 lower\_bound和Upper\_bound

STL中的每个算法都非常精妙，接下来的几天我想集中学习一下STL中的算法。

　　ForwardIter lower\_bound(ForwardIter first, ForwardIter last,const \_Tp& val)算法返回一个非递减序列[first, last)中的第一个大于等于值val的位置。

     ForwardIter upper\_bound(ForwardIter first, ForwardIter last, const \_Tp& val)算法返回一个非递减序列[first, last)中第一个大于val的位置。

|  |
| --- |
| #include <iostream>  #include <algorithm>//必须包含的头文件  using namespace std;  int main(){  int point[10] = {1,3,7,7,9};  int tmp = upper\_bound(point, point + 5, 7) - point;//按从小到大，7最多能插入数组point的哪个位置  printf("%d\n",tmp);  tmp = lower\_bound(point, point + 5, 7) - point;////按从小到大，7最少能插入数组point的哪个位置  printf("%d\n",tmp);  return 0;  } |

## 2. 复杂的数论

### 2.1 大数阶乘分割法

大数处理的经典思路，将大数划分成N个数组，每个数组的值存的数的长度统一为m，进位为c 最后再进行格式化输出，巧妙的避开了爆数据范围的问题。

|  |
| --- |
| #include <stdio.h>  #define \_MAX 100000000  int main()  {  int n, i, j, m;  long long a[10000], c;  scanf("%d",&n);  m = 0;  a[0] = 1;  for(i = 1; i <= n; i++)  {  c = 0;  for(j = 0; j <= m; j++)  {  a[j] = a[j] \* i + c;  c = a[j] / \_MAX;  a[j] %= \_MAX;  }  if(c > 0)  {  m++;  a[m] = c;  }  }  printf("%lld", a[m]);  for(i = m - 1; i >= 0; i--)  printf("%0.8lld", a[i]);  printf("\n");  return 0;  } |

### 2.2 大数阶乘长度Stirling公式

斯特林公式可以用来估算某数的大小，结合lg可以估算某数的位数，或者可以估算某数的阶乘是另一个数的倍数。

|  |
| --- |
| Stirling公式：sqrt(2\*pi\*n)\*pow(n/e,n)  Stirling公式求阶乘位数：log10（sqrt（2\*M\_PI\*n）)+n\*log10(n/M\_E)  注意强制类型转换成double型  #include <iostream>  #include<math.h>  using namespace std;  typedef long long ll;  /\*斯特林公式  sqrt(2\*pi\*n)\*pow(n/e,n)  \*/  int main()  {  ll n;  cin>>n;  //对斯特林公式开log求阶乘位数  ll l=log10((long double)(sqrt(2\*M\_PI\*n)))+n\*log10((long double)n/M\_E)+1;  cout<<l<<endl;  return 0;  } |

### 2.3 欧拉函数

（1）直接欧拉 返回N的欧拉函数值

|  |
| --- |
| #include <iostream>  using namespace std;  int euler(int n)  {  int res=n,a=n;  for(int i=2;i\*i<=a;i++)  {  if(a%i==0)  {  res=res/i\*(i-1);//先进行除法是为了防止溢出  while(a%i==0)a/=i;  }  }  if(a>1)res=res/a\*(a-1);  return res;  }  int main()  {  int n;  cin>>n;  cout<<euler(n)<<endl;  return 0;  } |

2、筛法求欧拉函数，返回1-N之间所有的欧拉函数值，保存到euler[i]中

|  |
| --- |
| void Init(){  euler[1]=1;  for(int i=2;i<Max;i++)  euler[i]=i;  for(int i=2;i<Max;i++)  if(euler[i]==i)  for(int j=i;j<Max;j+=i)  euler[j]=euler[j]/i\*(i-1);//先进行除法是为了防止中间数据的溢出  } |

### 2.4 快速幂

|  |
| --- |
| #include <iostream>  using namespace std;  typedef long long ll;  ll mod\_pow(ll x,ll n,ll mod)  {  ll res=1;  while(n>0)  {  //如果二进制最低位为1，则乘上x^(2^i)  if(n&1)  {  res=res\*x%mod;  }  x=x\*x%mod;  n>>=1;  }  return res;  }  int main()  {  ll a,b,c;  scanf("%lld%lld%lld",&a,&b,&c);  printf("%lld\n",mod\_pow(a,b,c));  return 0;  } |

### 2.5 GCD与LCM与ExtGcd

（1）GCD和LCM

|  |
| --- |
| #include<iostream>  using namespace std;  typedef long long ll;  ll gcd(ll m,ll n)  {  return (m==0)?n:gcd(n%m,m);  }  ll lcm(ll a,ll b)  {  return (a/gcd(a,b)\*b);  }  int main()  {  int a,b;  cin>>a>>b;  cout<<lcm(a,b)<<endl;  return 0;  } |

（2）ExtGcd扩展欧几里得

|  |
| --- |
| /\*  \* 求x，y使得gcd(a, b) = a \* x + b \* y;  \*/  int extgcd(int a, int b, int &x, int &y)  {  if (b == 0)  {  x = 1;  y = 0;  return a;  }  int d = extgcd(b, a % b, x, y);  int t = x;  x = y;  y = t - a / b \* y;  return d;  } |

### 2.6 素数测试（判断素数）

（1）素数测试

|  |
| --- |
| #include<cstdlib>  #include<cstdio>  int modularExponent(int a, int b, int n) {  int ret = 1;  for (; b; b >>= 1, a = (int) ((long long) a \* a % n)) {  if (b & 1) {  ret = (int) ((long long) ret \* a % n);  }  }  return ret;  }  bool millerRabin(int n,int a) {  if (n == 1 || (n != 2 && !(n % 2)) || (n != 3 && !(n % 3)) || (n != 5 && !(n % 5)) || (n != 7 && !(n % 7))) {  return false;  }  int r = 0, s = n - 1, j;  if(!(n%a)) return false;  while(!(s&1)){ s >>= 1; r++; }  long long k = modularExponent(a, s, n);  if(k == 1) return true;  for(j = 0; j < r; j++, k = k \* k % n)  if(k == n - 1) return true;  return false;  }  bool miller\_Rabin(int n)//  {  int a[]={2,3,5,7},i;//能通过测试的最小素数为 3215031751(此数超int)  for(i=0;i<4;i++){  if(!millerRabin(n,a[i]))return false;  }  return true;  }  int main()  {  int n,x;  scanf("%d",&n);  while(n--){  scanf("%d",&x);  printf("%s\n",miller\_Rabin(x)?"Yes":"No");  }  return 0;  } |

（2）大神代码

|  |
| --- |
| #include<cstdlib>  #include<cstdio>  int modularExponent(int a, int b, int n) {  int ret = 1;  for (; b; b >>= 1, a = (int) ((long long) a \* a % n)) {  if (b & 1) {  ret = (int) ((long long) ret \* a % n);  }  }  return ret;  }  bool millerRabin(int n,int a) {  if (n == 1 || (n != 2 && !(n % 2)) || (n != 3 && !(n % 3)) || (n != 5 && !(n % 5)) || (n != 7 && !(n % 7))) {  return false;  }  int r = 0, s = n - 1, j;  if(!(n%a)) return false;  while(!(s&1)){ s >>= 1; r++; }  long long k = modularExponent(a, s, n);  if(k == 1) return true;  for(j = 0; j < r; j++, k = k \* k % n)  if(k == n - 1) return true;  return false;  }  bool miller\_Rabin(int n)//  {  int a[]={2,3,5,7},i;//能通过测试的最小素数为 3215031751(此数超int)  for(i=0;i<4;i++){  if(!millerRabin(n,a[i]))return false;  }  return true;  }  int main()  {  int n,x;  scanf("%d",&n);  while(n--){  scanf("%d",&x);  printf("%s\n",miller\_Rabin(x)?"Yes":"No");  }  return 0;  } |

3、查找小于等于MAXN的素数（生成连续素数表）

|  |
| --- |
| /\*  \* 素数筛选，查找出小于等于MAXN的素数  \* prime[0]存素数的个数  \*/  const int MAXN = 100000;  int prime[MAXN + 1];  void getPrime()  {  memset(prime, 0, sizeof(prime));  for (int i = 2; i <= MAXN; i++)  {  if (!prime[i])  {  prime[++prime[0]] = i;  }  for (int j = 1; j <= prime[0] && prime[j] <= MAXN / i; j++)  {  prime[prime[j] \* i] = 1;  if (i % prime[j] == 0)  {  break;  }  }  }  } |

（4）大数素数测试

|  |
| --- |
| #define MAXL 4  #define M10 1000000000  #define Z10 9  const int zero[MAXL - 1] = {0};  struct bnum  {  int data[MAXL]; // 断成每截9个长度  // 读取字符串并转存  void read()  {  memset(data, 0, sizeof(data));  char buf[32];  scanf("%s", buf);  int len = (int)strlen(buf);  int i = 0, k;  while (len >= Z10)  {  for (k = len - Z10; k < len; ++k)  {  data[i] = data[i] \* 10 + buf[k] - '0';  }  ++i;  len -= Z10;  }  if (len > 0)  {  for (k = 0; k < len; ++k)  {  data[i] = data[i] \* 10 + buf[k] - '0';  }  }  }  bool operator == (const bnum &x)  {  return memcmp(data, x.data, sizeof(data)) == 0;  }  bnum & operator = (const int x)  {  memset(data, 0, sizeof(data));  data[0] = x;  return \*this;  }  bnum operator + (const bnum &x)  {  int i, carry = 0;  bnum ans;  for (i = 0; i < MAXL; ++i)  {  ans.data[i] = data[i] + x.data[i] + carry;  carry = ans.data[i] / M10;  ans.data[i] %= M10;  }  return ans;  }  bnum operator - (const bnum &x)  {  int i, carry = 0;  bnum ans;  for (i = 0; i < MAXL; ++i)  {  ans.data[i] = data[i] - x.data[i] - carry;  if (ans.data[i] < 0)  {  ans.data[i] += M10;  carry = 1;  }  else  {  carry = 0;  }  }  return ans;  }  // assume \*this < x \* 2  bnum operator % (const bnum &x)  {  int i;  for (i = MAXL - 1; i >= 0; --i)  {  if (data[i] < x.data[i])  {  return \*this;  }  else if (data[i] > x.data[i])  {  break;  }  }  return ((\*this) - x);  }  bnum & div2()  {  int i, carry = 0, tmp;  for (i = MAXL - 1; i >= 0; --i)  {  tmp = data[i] & 1;  data[i] = (data[i] + carry) >> 1;  carry = tmp \* M10;  }  return \*this;  }  bool is\_odd()  {  return (data[0] & 1) == 1;  }  bool is\_zero()  {  for (int i = 0; i < MAXL; ++i)  {  if (data[i])  {  return false;  }  }  return true;  }  };  void mulmod(bnum &a0, bnum &b0, bnum &p, bnum &ans)  {  bnum tmp = a0, b = b0;  ans = 0;  while (!b.is\_zero())  {  if (b.is\_odd())  {  ans = (ans + tmp) % p;  }  tmp = (tmp + tmp) % p;  b.div2();  }  }  void powmod(bnum &a0, bnum &b0, bnum &p, bnum &ans)  {  bnum tmp = a0, b = b0;  ans = 1;  while (!b.is\_zero())  {  if (b.is\_odd())  {  mulmod(ans, tmp, p, ans);  }  mulmod(tmp, tmp, p, tmp);  b.div2();  }  }  bool MillerRabinTest(bnum &p, int iter)  {  int i, small = 0, j, d = 0;  for (i = 1; i < MAXL; ++i)  {  if (p.data[i])  {  break;  }  }  if (i == MAXL)  {  // small integer test  if (p.data[0] < 2)  {  return false;  }  if (p.data[0] == 2)  {  return true;  }  small = 1;  }  if (!p.is\_odd())  {  return false; // even number  }  bnum a, s, m, one, pd1;  one = 1;  s = pd1 = p - one;  while (!s.is\_odd())  {  s.div2();  ++d;  }  for (i = 0; i < iter; ++i)  {  a = rand();  if (small)  {  a.data[0] = a.data[0] % (p.data[0] - 1) + 1;  }  else  {  a.data[1] = a.data[0] / M10;  a.data[0] %= M10;  }  if (a == one)  {  continue;  }  powmod(a, s, p, m);  for (j = 0; j < d && !(m == one) && !(m == pd1); ++j)  {  mulmod(m, m, p, m);  }  if (!(m == pd1) && j > 0)  {  return false;  }  }  return true;  }  int main()  {  bnum x;  x.read();  puts(MillerRabinTest(x, 5) ? "Yes" : "No");  return 0;  } |

（5）判断小于MAXN得数是不是素数

|  |
| --- |
| /\*  \* 素数筛选，判断小于MAXN的数是不是素数  \* notprime是一张表，false表示是素数，true表示不是  \*/  const int MAXN = 1000010;  bool notprime[MAXN];  void init()  {  memset(notprime, false, sizeof(notprime));  notprime[0] = notprime[1] = true;  for (int i = 2; i < MAXN; i++)  {  if (!notprime[i])  {  if (i > MAXN / i) // 阻止后边i \* i溢出（或者i,j用long long)  {  continue;  }  // 直接从i \* i开始就可以，小于i倍的已经筛选过了  for (int j = i \* i; j < MAXN; j += i)  {  notprime[j] = true;  }  }  }  } |

### 2.7 求逆元

（1）扩展欧几里得法

|  |
| --- |
| /\*  扩展欧几里得法（求ax+by=gcd）  返回d=gcd(a,b);和对应等式ax+by=d中的x、y  \*/  typedef long long ll;  ll extendGcd(ll a,ll b,ll &x,ll &y)  {  if(a==0&&b==0)  {  return -1;  }  if(b==0)  {  x=1;  y=0;  return a;  }  ll d=extendGcd(b,a%b,y,x);  y-=a/b\*x;  return d;  }  //求逆元ax=1(mod n)  ll modReverse(ll a,ll n)  {  ll x,y;  ll d=extendGcd(a,n,x,y);  if(d==1)  return (x%n+n)%n;  else  return -1;//表示无逆元  } |

（2）简洁写法

|  |
| --- |
| /\*  求逆元的简洁写法  只能求a<m的情况，且a与m互质  求ax=1（mod m）的x的值，即逆元（0<a<m）  \*/  typedef long long ll;  ll inv(ll a,ll m)  {  if(a==1)  return 1;  return inv(m%a,m)\*(m-m/a)%m;  } |

(3) 欧拉函数求逆元（费马小定理）

|  |
| --- |
| /\*  欧拉函数法  mod为素数，而且a和m互质  \*/  //快速幂取模  ll powM(ll a,ll b,ll m)  {  ll tmp=1;  if(b==0)return 1;  if(b==1)return a%m;  tmp=powM(a,a>>1,m);  tmp=tmp\*tmp%m;    if(b&1)  {  tmp=tmp\*a%m;  }  return tmp;  }  ll inv(ll a,ll m)  {  return powM(a,m-2,m);  } |

（4）求阶乘逆元

|  |
| --- |
| typedef long long ll;  const ll MOD = 1e9 + 7; // 必须为质数才管用  const ll MAXN = 1e5 + 3;  ll fac[MAXN]; // 阶乘  ll inv[MAXN]; // 阶乘的逆元  ll QPow(ll x, ll n)  {  ll ret = 1;  ll tmp = x % MOD;  while (n)  {  if (n & 1)  {  ret = (ret \* tmp) % MOD;  }  tmp = tmp \* tmp % MOD;  n >>= 1;  }  return ret;  }  void init()  {  fac[0] = 1;  for (int i = 1; i < MAXN; i++)  {  fac[i] = fac[i - 1] \* i % MOD;  }  inv[MAXN - 1] = QPow(fac[MAXN - 1], MOD - 2);  for (int i = MAXN - 2; i >= 0; i--)  {  inv[i] = inv[i + 1] \* (i + 1) % MOD;  }  } |

### .2.8 原根

1.原根定义：设m>1,gcd(a,m)=1,使得这里写图片描述成立的最小的r，称为a对模m的阶。

2.定理：如果模m有原根，那么他一共有这里写图片描述个原根。这里的函数表示[1,m)中与m互质的个数

3.定理：如果p为素数，那么素数p一定存在原根，并且模p的原根的个数为这里写图片描述个。

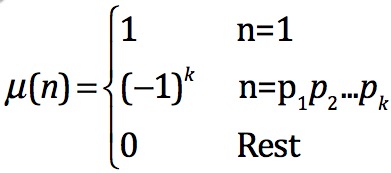
4.定理：假设m是正整数，a是整数，如果a模m的阶等于这里写图片描述，则称a为模m的一个原根。

5.模m有原根的充要条件：m=2,4,P^a,2\*P^a…….   
求模素数P的原根的方法：对P-1素因子分解，即P-1=(P1^a1)(P2^a2)…..(Pk^ak)。，若恒有这里写图片描述成立，那么g就是P的原根(对于合数而言,只需要把p-1换成这里写图片描述即可)

|  |
| --- |
| #include <stdio.h>  #include <math.h>  #include <string.h>  #include <stdlib.h>  #include <iostream>  #include <sstream>  #include <algorithm>  #include <set>  #include <queue>  #include <stack>  #include <map>  #include <bitset>  #pragma comment(linker, "/STACK:102400000,102400000")  using namespace std;  typedef long long LL;  const int inf=0x3f3f3f3f;  const double pi= acos(-1.0);  const double esp=1e-7;  const int Maxn=1e6+10;  int prime[Maxn];//存储素数  int sprime[Maxn];//存储P-1的素因子  bitset<Maxn>pri;//结果只有0和1，判断是否为素数  int k;//记录Maxn以内的素数个数  int cnt;//记录素因子的个数  void is\_prime()  {  pri.set();//将所有的二进制数都标为1  for(int i=2; i<Maxn; i++) {  if(pri[i]) {  prime[k++]=i;  for(int j=i+i; j<Maxn; j+=i)  pri[j]=0;  }  }  }  void Divide(int n)//将n分解为素因子  {  cnt=0;  int t=(int)sqrt(1.0\*n);  for(int i=0; prime[i]<=t; i++) {  if(n%prime[i]==0) {  sprime[cnt++]=prime[i];  while(n%prime[i]==0)//因为有可能有多个peime[i]  n/=prime[i];  }  }  if(n>1)  sprime[cnt++]=n;//可能只有自己一个素因子  }  LL modexp(LL a,LL b,int mod)//快速幂取余  {  LL res=1;  while(b>0) {  a=a%mod;  if(b&1)  res=res\*a%mod;  b=b>>1;  a=a\*a%mod;  }  return res;  }  int main()  {  int p;  is\_prime();  while(~scanf("%d",&p)) {  Divide(p-1);  for(int g=2; g<p; g++) {  int flag=1;  for(int i=0; i<cnt; i++) {  int t=(p-1)/sprime[i];  if(modexp(g,t,p)==1) {  flag=0;  break;  }  }  if(flag) {  int root=g;  printf("%d\n",root);  break;//去掉break的话是求所有的原根，加上break是求最小的原根、  }  }  }  return 0;  } |

### 2.9 莫比乌斯函数求法

http://img.blog.csdn.net/20160707164935705



http://img.blog.csdn.net/20160707165105677

（1）单独求解

|  |
| --- |
| #include <iostream>  using namespace std;  typedef long long ll;  //计算a是否可以mod b  int MOD(int a,int b)  {  return a-a/b\*b;  }  //计算莫比乌斯函数  //如果一个数包含平方因子，那么miu(n)=0  //如果哟个数不包含平方因子，且有k个不同的质因子，那么miu(n)=(-1)^k  int miu(int n)  {  int cnt,k=0;  for(int i=2;i\*i<n;i++)  {  if(MOD(n,i))  {  continue;  }  cnt=0;  k++;  while(MOD(n,i)==0)  {  n/=i;  cnt++;  }  if(cnt>=2)  {  return 0;  }  }  if(n!=1)  {  k++;  }  return MOD(k,2)?-1:1;  }  int main()  {  //cout << "Hello world!" << endl;  ll n;  cin>>n;  cout<<miu(n)<<endl;  return 0;  } |

（2）线性筛法求解】

|  |
| --- |
| /\*  \* 莫比乌斯反演公式  \* ￼￼￼￼￼￼￼线性筛法求解积性函数（莫比乌斯函数）  \*/  const int MAXN = 1000000;  bool check[MAXN + 10];  int prime[MAXN + 10];  int mu[MAXN + 10];  void Moblus()  {  memset(check, false, sizeof(check));  mu[1] = 1;  int tot = 0;  for (int i = 2; i <= MAXN; i++)  {  if (!check[i])  {  prime[tot++] = i;  mu[i] = -1;  }  for (int j = 0; j < tot; j++)  {  if (i \* prime[j] > MAXN)  {  break;  }  check[i \* prime[j]] = true;  if (i % prime[j] == 0)  {  mu[i \* prime[j]] = 0;  break;  }  else  {  mu[i \* prime[j]] = -mu[i];  }  }  }  } |

### 2.10 ACM博弈

（1）Bash博弈

1、问题模型：只有一堆n个物品，两人轮流从这堆物品中取物，最多取m个，最后取光者胜。

2、解决思路：当n=m+1时，由于一次最多取m个，无论先取者拿走多少个，后取者都能一次拿走剩余的物品，后者取胜，所以当一方面对n%(m+1)==0的时候，其面临的是必败局势。所以当n==(n+1)\*r+s（r为任意自然数，s<=m）时，如果先取者要拿走s个物品，后取者拿走x（x<=m）个物品，那么先取者再拿走m+1-k个，结果剩下（m+1）（r-1）个，以后保持这样的取法，那么先取者肯定获胜。总之，要保留给对手留下（m+1）的倍数，就能最后获胜。

3、变形：条件不变，改为最后取光的人输。

解决方法：当(n-1)%(m+1)==0时，后手胜利。

|  |
| --- |
| #define \_MAX 10000  int a[\_MAX];  int b[\_MAX];  int bash(int N, int K)  {  if (N % (K + 1) == 0)  {  return 2;  }  return 1;  }  int main()  {  int T;  scanf("%d", &T);  for (int i = 0; i < T; i++)  {  scanf("%d%d", a + i, b + i);  }  for (int i = 0; i < T; i++)  {  if (bash(a[i], b[i]) == 1)  {  printf("A\n");  }  else  {  printf("B\n");  }  }  return 0;  } |

（2）威佐夫博弈

1、问题模型：有两堆各若干个物品，两个人轮流从某一堆或同时从两堆取出同样多的物品，规定每次取出一个，多者不限，最后取光者得胜。

2、解决思路：A:设（ai,bi）(ai<=bi,i=0,1,2,3.....,n)表示两堆物品的数量并称其为局势，如果甲面对（0,0）,那么甲输了，这种局势我们称为其一局势。

奇异局势的前几项是：(0,0)、(1,2)、（3,5）、（4,7）、（6，10）、（8,13）、（9,15）、（11,18）、（12,20）

任给一个局势(a,b),如下公式判断它是不是奇异局势：

ak =[k（1+√5）/2]，bk= ak + k  （k=0，1，2，…,n 方括号表示取整函数）。（证明见百度百科）也就是黄金分割点。

3、满足上述公式的局势性质：

（1）任何自然数都包含在一个且仅有一个奇异局势中。

（2）任何操作都可以把奇异局势变为非奇异局势。

若只改变奇异局势（ak,bk）的某一个分量，那么另一个分量不可能在其他奇异局势中，所以必然是非奇异局势。

（3）采用适当的方法，可以将非奇异局势变为奇异局势。

  假设面对的局势是（a,b），若 b = a，则同时从两堆中取走 a 个物体，就变为了奇异局势（0，0）；如果a = ak ，b > bk，那么，取走b  – bk个物体，即变      为奇异局势；如果 a = ak ，  b < bk ,则同时从两堆中拿走 ak – ab – ak个物体,变为奇异局势（ ab – ak , ab – ak+ b – ak）；如果a > ak ，            b= ak + k,则从第一堆中拿走多余的数量a – ak 即可；如果a < ak ，b= ak + k,分两种情况，第一种，a=aj （j < k）,从第二堆里面拿走 b – bj 即可； 第      二种，a=bj （j < k）,从第二堆里面拿走 b – aj 即可。

4、结论

两个人如果都采用正确操作，那么面对非奇异局势，先拿者必胜；反之，后拿者必胜。

|  |
| --- |
| int main()  {  int t, a, b, m, k;  scanf("%d", &t);  while (t--)  {  scanf("%d%d", &a, &b);  if (a > b)  {  a ^= b;  b ^= a;  a ^= b;  }  m = b - a;  k = (int)(m \* (1 + sqrt(5)) / 2.0);  //m = ? \* a  //k = m / ?  //?:黄金分割数  //如果a == k，则为后手赢，否则先手赢（奇异局）  printf("%s\n", a == k ? "B" : "A");  }  return 0;  } |

[威佐夫博弈 V2](http://www.51nod.com/onlineJudge/questionCode.html#!problemId=1185)（大数）

|  |
| --- |
| #include <iostream>  #include <cstdio>  #include <cstring>  #include <cstdlib>  #include <cmath>  using namespace std;  typedef long long LL;  LL tmp[3] = {618033988,749894848,204586834};  LL MOD = 1000000000;  int main()  {  int T;  LL m, n;  cin>>T;  while(T--)  {  cin>>m>>n;  if(m < n)  swap(n, m);  LL cha = m - n;  LL ta = cha/MOD, tb = cha%MOD;  LL tp = tb\*tmp[2];  tp = ta\*tmp[2] + tb\*tmp[1] + tp/MOD;  tp = ta\*tmp[1] + tb\*tmp[0] + tp/MOD;  tp = cha + ta\*tmp[0] + tp/MOD;  if(tp == n)  puts("B");  else  puts("A");  }  return 0;  } |

（3）Nim博弈

1、问题模型：有三堆各若干个物品，两个人轮流从某一堆取人一多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

2、解决思路：

用（a,b,c）表示某种局势，显然(0,0,0)是第一种局势，无论谁面对奇异局势，都必然失败。

第二种是（0，n,n），只要与对手拿走一样多的物品，最后都将导致(0,0,0)。

搞定这个问题必须把必败态炸出：（a,b,c）是必败态等价于a^b^c==0（^表示异或运算）

3、推广：

如果我们面对的是一个非奇异局势(a,b,c),那么如何变成奇异局势呢？

假设a<b<c,我们只要将c变成a^b即可，因为a^b^(a^b)=(a^a)^(b^b)=0^0=0。要将c变成a^b，只要c-(a^b)。

|  |
| --- |
| int main(int argc, const char \* argv[])  {  int N, stone, tag = 0;  scanf("%d", &N);  while (N--)  {  scanf("%d", &stone);  tag ^= stone;  }  //tag为0则为后手赢，否则为先手赢  printf("%c\n", tag == 0 ? 'B' : 'A');  return 0;  } |

### 2.11数论相关公式

|  |
| --- |
| 欧拉定理  对于互质的整数a和n，有a^φ(n) ≡ 1(mod n)  费马定理  a是不能被质数p整除的正整数，有a^(p-1) ≡ 1(mod p)  Polya定理  设G是p个对象的一个置换群，用k种颜色去染这p个对象，若一种染色方案在群G的作用下变为一种方案，则这两个方案当作是同一种方案，这样的不同染色方案数为：  L = 1 / |G| x ∑(k^C(f)), f ∈ G  C(f)为循环节，|G|表示群的置换方法数。  对于n个位置的手镯，有n种旋转置换和n种翻转置换。  对于旋转置换：  C(f[i]) = gcd(n, i)，i表示旋转i颗宝石以后，i = 0时，gcd(n, 0) = n；  对于翻转置换：  如果n为偶数：则有n / 2个置换C(f) = n / 2，有n / 2个置换C(f) = n / 2 + 1；如果n为奇数：则有n个置换C(f) = n / 2 + 1。  欧拉函数 φ(n)  φ(n)积性函数，对于一个质数p和正整数k，有  φ(p^k) = p^k - p^(k-1) = (p - 1)p^(k - 1) = p^k(1 - 1 / p)  公式  当n > 1时，1 … n中与n互质的整数和为nφ(n) / 2。  2n 位数  len=(int)(n∗long10(2))+1,(2n−1同样适用)  默慈金数 HVD 问题  m[i]={i,m[i−1]∗(2∗i+1)+m[i−2]∗(3∗i−3)i+2,if i∈{1,2}if i>2 |

## 3. 有趣的字符串

### 3.1 判断回文串的方法

（1）中心扩展法

|  |
| --- |
| string findLongestPalindrome(string &s)  {  const int length=s.size();  if(length == 1)return s;  if(length == 0)return NULL;  int maxlength=0;  int start;  for(int i=0;i<length;i++)//长度为奇数  {  int j=i-1,k=i+1;  while(j>=0&&k<length&&s.at(j)==s.at(k))  {  if(k-j+1>maxlength)  {  maxlength=k-j+1;  start=j;  }  j--;  k++;  }  }  for(int i=0;i<length;i++)//长度为偶数  {  int j=i,k=i+1;  while(j>=0&&k<length&&s.at(j)==s.at(k))  {  if(k-j+1>maxlength)  {  maxlength=k-j+1;  start=j;  }  j--;  k++;  }  }  if(maxlength>0)  return s.substr(start,maxlength);  return NULL;  } |

（2）Manacher算法

|  |
| --- |
| #define min(x, y) ((x)<(y)?(x):(y))  #define max(x, y) ((x)<(y)?(y):(x))  string findLongestPalindrome3(string s)  {  int length=s.size();  for(int i=0,k=1;i<length-1;i++)//给字符串添加 # {  s.insert(k,"#");  k=k+2;  }  length=length\*2-1;//添加#后字符串长度  int \*rad=new int[length]();  rad[0]=0;  for(int i=1,j=1,k;i<length;i=i+k)  {  while(i-j>=0&&i+j<length&&s.at(i-j)==s.at(i+j))  j++;  rad[i]=j-1;  for(k=1;k<=rad[i]&&rad[i-k]!=rad[i]-k;k++)//镜像,遇到rad[i-k]=rad[i]-k停止，这时不用从j=1开始比较  rad[i+k]=min(rad[i-k],rad[i]-k);  j=max(j-k,0);//更新j  }  int max=0;  int center;  for(int i=0;i<length;i++)  {  if(rad[i]>max)  {  max=rad[i];  center=i;  }  }  return s.substr(center-max,2\*max+1);  } |

### 3.2 KMP匹配

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  void get\_next(char \*t,int \*next);  int index\_kmp(char \*s,char \*t,int index);  int main(void)  {  char s[]="hello world!";  char t[]="world";  /\*  int next[strlen(t)];  get\_next(t,next);  int i;  for(i=0;i<strlen(t);i++)  {  printf("%d,",next[i]);  }  \*/  int m=index\_kmp(s,t,0);  printf("index:%d\n",m);  exit(0);  }  /\*  在字符串s中，从下标index开始查找是否含有字符串t.如果有，返回t在s中的开始位置；如果没有，返回-1。  (使用KMP算法实现)  注：字符数组s和t中，不再保存字符串长度。  \*/  int index\_kmp(char \*s,char \*t,int index)  {  int next[strlen(t)];  get\_next(t,next);  int i=index,j=0;  while(s[i]!='\0' && t[j]!='\0')  {  if(s[i]==t[j])  {  i++;  j++;  continue;  }  else  {  j=next[j]; //从模式匹配数组中，获取要回溯到的结点  }  if(0==j) //单独处理第一个字符  {  if(s[i]==t[j])  {  i++;  j++;  }  else  {  i++;  }  }  }  if(t[j]=='\0') //表示字符串t中，所有字符已匹配完毕  {  return i-strlen(t); //因为i以匹配至s中t字符串的结尾。因为要返回的是s中t的开始下标，故i-strlen(t).  }  else  {  return -1;  }  }  /\*  KMP算法之next数组代码  next数组定义：当模式匹配串T失配的时候，next数组对应的元素知道应该用T串的哪个元素进行下一轮的匹配。  \*/  void get\_next(char \*t,int \*next)  {  int i=0; //Prefix 前缀  int j=1; //Postfix 后缀  next[0]=0; //自定义的，0和1都从0开始匹配  next[1]=0;  while(t[j]!='\0')  {  if(t[i]==t[j]) //若前后字符匹配，则向前推进  {  i++;  j++;  next[j]=i;  continue;  }  else  {  i=next[i]; //前后字符不匹配，则回溯。注意，此时是i和j不匹配，因此，根据next数组定义，要回溯到next[i]的值。  }  if(0==i) //当回溯到首字符时，单独进行处理  {  if(t[i]==t[j])  {  next[++j]=++i;  }  else  next[++j]=i;  }  }  } |

### 3.3 扩展KMP

扩展KMP的应用：

给出模板串S和串T，长度分别为Slen和Tlen，要求在线性时间内，对于每个S[i]（0<=i<Slen)，求出S[i..Slen-1]与T的

最长公共前缀长度，记为extend[i]（或者说，extend[i]为满足S[i..i+z-1]==T[0..z-1]的最大的z值）。

扩展KMP可以用来解决很多字符串问题，如求一个字符串的最长回文子串和最长重复子串。

|  |
| --- |
| /\*  \* 扩展KMP  \* next[i]:x[i...m-1]的最长公共前缀  \* extend[i]:y[i...n-1]与x[0...m-1]的最长公共前缀  \*/  void preEKMP(char x[], int m, int next[])  {  next[0] = m;  int j = 0;  while (j + 1 < m && x[j] == x[j + 1])  {  j++;  }  next[1] = j;  int k = 1;  for (int i = 2; i < m; i++)  {  int p = next[k] + k - 1;  int L = next[i - k];  if (i + L < p + 1)  {  next[i] = L;  }  else  {  j = std::max(0, p - i + 1);  while (i + j < m && x[i + j] == x[j])  {  j++;  }  next[i] = j;  k = i;  }  }  return ;  }  void EKMP(char x[], int m, char y[], int n, int next[], int extend[])  {  preEKMP(x, m, next);  int j = 0;  while (j < n && j < m && x[j] == y[j])  {  j++;  }  extend[0] = j;  int k = 0;  for (int i = 1; i < n; i++)  {  int p = extend[k] + k - 1;  int L = next[i - k];  if (i + L < p + 1)  {  extend[i] = L;  }  else  {  j = std::max(0, p - i + 1);  while (i + j < n && j < m && y[i + j] == x[j])  {  j++;  }  extend[i] = j;  k = i;  }  }  return ;  } |

### 3.4 strstr函数

原型：char \*strstr(const char \*str1,const char \*str2);

#include<string.h>

找出str2字符串在str1字符串中第一次出现的位置（不包括str2的串结束符）。返回该位置的指针，如果找不到，返回空指针。

|  |
| --- |
| int main()  {  char \*s="Golden"  char \*l="go";  char \*p=strstr(s,l);  } |

## 4. 图论

### 4.1 最小生成树的解法

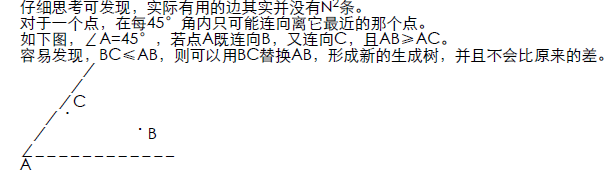
（1）Prim算法

|  |
| --- |
| /\*  下标为1-n  返回最小生成树的权值，返回-1说明无连通  \*/  #define inf 0x3f3f3f3f  int G[1001][1001];  int vis[1001],lowc[1001];  void init(int x,int y,int v)  {  G[x][y]=v;  G[y][x]=v;  return;  }  int prim(int G[][1001],int n){  int i,j,p,minc,res=0;  memset(vis,0,sizeof(vis));//全部初值为0表示没有访问过；  vis[1]=1;  for(i=2;i<=n;i++)  lowc[i]=G[1][i];  for(i=2;i<=n;i++){  minc=inf;  p=-1;  for(j=1;j<=n;j++){  if(vis[j]==0&&lowc[j]<minc)  {minc=lowc[j];p=j;}  }  if(inf==minc) return -1;//原图不连通  res+=minc;  vis[p]=1;  for(j=1;j<=n;j++){//更新lowc[]  if(vis[j]==0&&lowc[j]>G[p][j])  lowc[j]=G[p][j];  }  }  return res;  } |

（2）Kruskal算法

|  |
| --- |
| /\*  \* Kruskal算法求MST  \* 对边操作，并排序  \* 切记：初始化赋值问题（tol）  \*/  const int MAXN = 110; // 最大点数  const int MAXM = 10000; // 最大边数  int F[MAXN]; // 并查集使用  struct Edge  {  int u; // 起点  int v; // 终点  int w; // 权值  } edge[MAXM]; // 存储边的信息  int tol; // 边数，加边前赋值为0  void addEdge(int u, int v, int w)  {  edge[tol].u = u;  edge[tol].v = v;  edge[tol++].w = w;  return ;  }  bool cmp(Edge a, Edge b)  {  // 排序函数，将边按照权值从小到大排序  return a.w < b.w;  }  int find(int x)  {  if (F[x] == x)  {  return x;  }  else  {  return F[x] = find(F[x]);  }  }  int Kruskal(int n) // 传入点数，返回最小生成树的权值，如果不连通则返回-1  {  for (int i = 0; i <= n; i++)  {  F[i] = i;  }  sort(edge, edge + tol, cmp);  int cnt = 0; // 计算加入的边数  int ans = 0;  for (int i = 0; i < tol; i++)  {  int u = edge[i].u;  int v = edge[i].v;  int w = edge[i].w;  int tOne = find(u);  int tTwo = find(v);  if (tOne != tTwo)  {  ans += w;  F[tOne] = tTwo;  cnt++;  }  if (cnt == n - 1)  {  break;  }  }  if (cnt < n - 1)  {  return -1; // 不连通  }  else  {  return ans;  }  } |

### 4.2 二维曼哈顿最小生成树



所以我们只要求一个点在其45°角的区域内离他最近的点就行了，而这可以用线段树或树状数组解决

我们以y轴正半轴往右偏45°角的区域为例：

点j在点i的这个区域要满足的条件是：

yj-xj>yi-xi

且xj>xi

那么我们将点以x为第一关键字，y为第二关键字，排序后倒序插入线段树

线段树的线段这一维是离散后的y-x，值是y+x

我们要求的是大于yi-xi的最小的y+x，而xj>xi这个条件已经由插入顺序满足了

这样我们成功的解决了这个区域的点

而其他区域的点我们可以通过坐标变换转移到这个区域

由于对称性，我们注意到其实只要求x轴或y轴正半轴所在的四个区域就行了

那么这个问题就这样解决了

|  |
| --- |
| #include <map>  #include <set>  #include <cmath>  #include <ctime>  #include <stack>  #include <queue>  #include <cstdio>  #include <memory>  #include <cctype>  #include <bitset>  #include <string>  #include <vector>  #include <climits>  #include <cstring>  #include <iostream>  #include <iomanip>  #include <algorithm>  #include <functional>  //#define FIN freopen("input.txt","r",stdin);  //#define FOUT freopen("output.txt","w+",stdout);  using namespace std;  typedef long long ll;  const int INF = 0x3f3f3f3f;  const int mod = 1e9 + 7;  const double eps=1e-8;  const double Pi=acos(-1.0);  const int N=50010;  struct point  {  int x,y,id;  bool operator<(const point p)const  {  return x!=p.x?x<p.x:y<p.y;  }  } p[N];  struct BIT  {  int min\_val,pos;  void init()  {  min\_val=INF;  pos=-1;  }  } bit[N];  int par[N];//并查集中父亲  int hight[N];//并查集树的高度  struct edge  {  int u,v,cost;  };  edge G[N<<2];//边集（边数）  int V,E;//顶点数和边数  int get\_Manhadm\_dis(point a,point b)  {  return abs(a.x-b.x)+abs(a.y-b.y);  }  void addedge(int u,int v,int w)  {  G[E].u=u;  G[E].v=v;  G[E++].cost=w;  }  int lowbit(int x)  {  return x&(-x);  }  void update(int x,int val,int pos)  {  for(int i=x; i>=1; i-=lowbit(i))  if(val<bit[i].min\_val)  {  bit[i].min\_val=val;  bit[i].pos=pos;  }  }  int ask(int x,int m)  {  int min\_val=INF;  int pos=-1;  for(int i=x; i<=m; i+=lowbit(i))  if(bit[i].min\_val<min\_val)  {  min\_val=bit[i].min\_val;  pos=bit[i].pos;  }  return pos;  }  void make\_edge()  {  int a[N],b[N];  for(int dir=0; dir<4; dir++)  {  if(dir==1||dir==3)  for(int i=0; i<V; i++)  swap(p[i].x,p[i].y);  else if(dir==2)  for(int i=0; i<V; i++)  p[i].x=-p[i].x;  sort(p,p+V);  for(int i=0; i<V; i++)  a[i]=b[i]=p[i].y-p[i].x;  sort(b,b+V);  int m=unique(b,b+V)-b;  for(int i=1; i<=m; i++)  bit[i].init();  for(int i=V-1;i>=0; i--)  {  int pos=lower\_bound(b,b+m,a[i])-b+1;  int ans=ask(pos,m);  if(ans!=-1)  addedge(p[i].id,p[ans].id,get\_Manhadm\_dis(p[i],p[ans]));  update(pos,p[i].x+p[i].y,i);  }  }  }  //并查集初始化  void Init\_union\_find(int n)  {  for(int i=0; i<n; i++)  {  par[i]=i;  hight[i]=0;  }  }  //查询树的根  int find(int x)  {  if(par[x]==x)  return x;  else  return par[x]=find(par[x]);  }  //合并x和y所属的集合  void unite(int x,int y)  {  x=find(x);  y=find(y);  if(x==y)  return ;  if(hight[x]<hight[y])  par[x]=y;  else  {  par[y]=x;  if(hight[x]==hight[y])  hight[x]++;  }  }  //判断x和y是否属于同一个集合  bool same(int x,int y)  {  return find(x)==find(y);  }  bool cmp(const edge& a,const edge& b)  {  return a.cost<b.cost;  }  int kruskal()  {  sort(G,G+E,cmp);//按照edge.cost的顺序从小到大排列  Init\_union\_find(V);//并查集初始化  int ans=0;  for(int i=0; i<E; i++)  {  edge e=G[i];  if(!same(e.u,e.v))  {  unite(e.u,e.v);  ans+=e.cost;  }  }  return ans;  }  int main()  {  scanf("%d",&V);  for(int i=0; i<V; i++)  {  scanf("%d %d",&p[i].x,&p[i].y);  p[i].id=i+1;  }  E=0;  make\_edge();  printf("%d\n",kruskal());  } |

## 5. 数据结构

### 5.1 树状数组 区间求和

|  |
| --- |
| #include<iostream>  #include<stdio.h>  #include<algorithm>  #include<string.h>  typedef long long ll;  using namespace std;  const int maxn=50010;  ll N,army[maxn];  ll lowbit(ll k)  {  return k&(-k);  }  void modify(ll x,ll add)//更新操作  {  while(x<=N)  {  army[x]+=add;  x+=lowbit(x);  }  }  ll get\_sum(ll x)//区间求和  {  ll ret=0;  while(x>0)  {  ret+=army[x];  x-=lowbit(x);  }  return ret;  }  int main()  {  scanf("%lld",&N);  ll d;  for(int i=1;i<=N;i++)  {  scanf("%lld",&d);  modify(i,d);  }  long long q;  scanf("%lld",&q);  ll s,l;  while(q--)  {  scanf("%lld%lld",&s,&l);  printf("%lld\n",get\_sum(l+s-1)-get\_sum(s-1));  //printf("%d\n",);  }  return 0;  } |

### 5.2 线段树

（1）区间查询，无单点更新

|  |
| --- |
| #include <iostream>  #include<stdio.h>  #include<string.h>  #include<algorithm>  using namespace std;  const int maxn=200010;  int sum[maxn<<2];//数组开四倍  int h,w,n;  //p表示下标  void pushUp(int p)  {  //p表示下标，赋值为两个儿子中的最大值  sum[p]=max(sum[p<<1],sum[p<<1|1]);  }  //建树  void build(int l,int r,int p)  {  if(l==r)//区间长度为0时结束递归  {  sum[p]=w;  return ;  }  //否则的话，分别递归左儿子和右儿子  int mid=(r+l)>>1;  build(l,mid,p<<1);  build(mid+1,r,p<<1|1);  pushUp(p);  }  //区间查询  //线段树的最高顶点是表示所有行里面最大的宽度  int query(int l,int r,int p,int num)  {  if(l==r)//找到了一个完全重合的区间  {  sum[p]-=num;//进行对应的查询操作  return l;  }  int ans;  int mid=(r+l)>>1;//下面要注意，先保存ans，然后更新值后再返回  if(sum[p<<1]>=num)  {  ans=query(l,mid,p<<1,num);  }  else  {  ans=query(mid+1,r,p<<1|1,num);  }  pushUp(p);  return ans;  }  int main()  {  while(scanf("%d%d%d",&h,&w,&n)!=EOF)  {    memset(sum,0,sizeof(sum));  if(h>n)h=n;  int temp;  build(1,h,1);//建树区间为[1,h]，下标开始序号为1  for(int i=0;i<n;i++)  {  scanf("%d",&temp);  if(sum[1]<temp)  cout<<"-1"<<endl;  else  cout<<query(1,h,1,temp)<<endl;  }  }  return 0;  } |

（2）单点更新+区间求和

|  |
| --- |
| #include<stdio.h>  #include<string.h>  #include<algorithm>  using namespace std;  #define MAXN 50010  #define lson l,mid,p<<1  #define rson mid+1,r,p<<1|1  #define mem(a) memset(a,0,sizeof(a[0]))  int sum[MAXN<<2];  void pushUp(int p)  {  sum[p]=sum[p<<1]+sum[p<<1|1];  }  void build(int l,int r,int p)  {  if(l==r){  scanf("%d",&sum[p]);  return;  }  int mid=(l+r)>>1;  build(lson);  build(rson);  pushUp(p);  }  int query(int l,int r,int p,int i,int j)  {  if(l>=i&&r<=j){  return sum[p];  }  int ans=0;  int mid=(l+r)>>1;  if(mid<j) ans+=query(rson,i,j);  if(mid>=i) ans+=query(lson,i,j);  return ans;  }  void update(int l,int r,int p,int i,int j)  {  if(l==r){  sum[p]+=j;  return ;  }  int mid=(l+r)>>1;  if(mid>=i)  update(lson,i,j);  else  update(rson,i,j);  pushUp(p);  }  int main()  {  int n,T;  int cases=1;  char str[10];  //freopen("in.txt","r",stdin);  scanf("%d",&T);  while(T--)  {  mem(sum);  scanf("%d",&n);  printf("Case %d:\n",cases++);  int i,j;  build(1,n,1);  while(scanf("%s",str)&&strcmp(str,"End"))  {  scanf("%d%d",&i,&j);  if(strcmp(str,"Query")==0)  printf("%d\n",query(1,n,1,i,j));  else if(strcmp(str,"Add")==0)  update(1,n,1,i,j);  else if(strcmp(str,"Sub")==0)  update(1,n,1,i,-j);  }  }  return 0;  } |

（3）区间查询最值

|  |
| --- |
| #include<stdio.h>  #include<string.h>  #include<algorithm>  using namespace std;  #define MAXN 200010  #define lson l,mid,p<<1  #define rson mid+1,r,p<<1|1  int sum[MAXN<<2];  void pushUp(int p)  {  sum[p]=max(sum[p<<1],sum[p<<1|1]);  }  void build(int l,int r,int p)  {  if(l==r){  scanf("%d",&sum[p]);  return ;  }  int mid=(l+r)>>1;  build(lson);  build(rson);  pushUp(p);  }  int query(int l,int r,int p,int a,int b)  {  if(l>=a&&b>=r)  return sum[p];  int ans=0;  int mid=(l+r)>>1;  if(mid>=a)  ans=max(ans,query(lson,a,b));  if(mid<b)  ans=max(ans,query(rson,a,b));  return ans;  }  void update(int l,int r,int p,int a,int b)  {  if(l==r){  sum[p]=b;  return ;  }  int mid=(l+r)>>1;  if(a<=mid) update(lson,a,b);  else update(rson,a,b);  pushUp(p);  }  int main()  {  int n,m;  char ch[5];  //freopen("in.txt","r",stdin);  while(~scanf("%d%d",&n,&m))  {  int a,b;  build(1,n,1);  while(m--)  {  scanf("%s%d%d",ch,&a,&b);  if(ch[0]=='Q')  printf("%d\n",query(1,n,1,a,b));  else  update(1,n,1,a,b);  }  }  return 0;  } |

## 6. 搜索

### 6.1 DFS

|  |
| --- |
| #include<cstdio>  #include<cstring>  #include<cstdlib>  using namespace std;  const int maxn=100;  bool vst[maxn][maxn]; //访问标记  int map[maxn][maxn]; //坐标范围  int dir[4][2]={0,1,0,-1,1,0,-1,0}; //方向向量，(x,y)周围的四个方向  bool CheckEdge(intx,int y) //边界条件和约束条件的判断  {  if(!vst[x][y]&&...) //满足条件  return 1;  else // 与约束条件冲突  return 0;  }  void dfs(int x,int y)  {  vst[x][y]=1; //标记该节点被访问过  if(map[x][y]==G) //出现目标态G  {  ...... //做相应处理  return;  }  for(int i=0;i<4;i++)  {  if(CheckEdge(x+dir[i][0],y+dir[i][1])) //按照规则生成下一个节点  dfs(x+dir[i][0],y+dir[i][1]);  }  return;//没有下层搜索节点，回溯  }  int main()  {  ......  return 0;  } |

### 6.2 BFS

|  |
| --- |
| #include<cstdio>  #include<cstring>  #include<queue>  #include<algorithm>  using namespace std;  const int maxn=100;  bool vst[maxn][maxn]; //访问标记  int dir[4][2]={0,1,0,-1,1,0,-1,0}; //方向向量  struct State //BFS 队列中的状态数据结构  {  int x,y; //坐标位置  int Step\_Counter; //搜索步数统计器  };  State a[maxn];  bool CheckState(State s) //约束条件检验  {  if(!vst[s.x][s.y]&&...) //满足条件  return 1;  else //约束条件冲突  return 0;  }  void bfs(State st)  {  queue<State>q; //BFS队列  State now,next; //定义2个状态，当前和下一个  st.Step\_Counter=0; //计数器清零  q.push(st); //入队  vst[st.x][st.y]=1; //访问标记  while(!q.empty())  {  now=q.front(); //取队首元素进行扩展  if(now==G) //出现目标态，此时为Step\_Counter的最小值，可以退出即可  {  ...... //做相关处理  return;  }  for(int i=0;i<4;i++)  {  next.x=now.x+dir[i][0]; //按照规则生成下一个状态  next.y=now.y+dir[i][1];  next.Step\_Counter=now.Step\_Counter+1; //计数器加1  if(CheckState(next)) //如果状态满足约束条件则入队  {  q.push(next);  vst[next.x][next.y]=1; //访问标记  }  }  q.pop(); //队首元素出队  } |

## 7. 动态规划

### 7.1 最长公共子序列lcs

分析:动态规划

dp[i][j] 表示字符串A以第i个位置 ,字符串B以第j个位置的最长公共子序列的长度

dp[i][j] = dp[i - 1][j - 1] + 1 if a[i] == a[j]

else dp[i][j] == max(dp[i - 1][j] , dp[i][j - 1]);

最大长度就是 dp[n][m] ,n 为A的长度 ,m为B的长度

还原字符串 ,只需要回到 dp[i][j] 刚开始发生改变的地方即可

|  |
| --- |
| #include<stdio.h>  #include<string.h>  #include<algorithm>  using namespace std;  const int maxn = 1E3 + 10;  char a[maxn],b[maxn],ans[maxn];  int dp[maxn][maxn];  int main(){  scanf("%s%s",a + 1,b + 1);  int n = strlen(a+1),m = strlen(b+1);  memset(dp,0,sizeof(dp));  for(int i = 1 ; i <= n ; ++i){  for(int j = 1 ; j <= m ; ++j){  if(a[i] == b[j]){  dp[i][j] = dp[i-1][j-1] + 1;  }else dp[i][j] = max(dp[i][j-1],dp[i-1][j]);  }  }  int cur = 0;  for(int i = n,j = m;dp[i][j];--i,--j){//返回到第一次更新值的地方  while(dp[i][j] == dp[i - 1][j]) --i;  while(dp[i][j] == dp[i][j - 1]) --j;  ans[cur++] = a[i];  }  reverse(ans,ans+cur);  ans[cur] = '\0';  printf("%s\n",ans);  return 0;  } |

### 7.2 最长上升子序列

Nlog（N）复杂度解法

|  |
| --- |
| #include <iostream>  #include<cstdio>  #include<string.h>  using namespace std;  #define Maxn 50010    typedef long long ll;  ll arr[Maxn],ans[Maxn],len;        int main()  {  ll p,i,j,k;  //scanf("%d",&T);  //while(T--)  //{  scanf("%lld",&p);  for(i=1;i<=p;i++)  {  scanf("%lld",&arr[i]);    }  ans[1]=arr[1];  len=1;  for(i=2;i<=p;i++)  {  if(arr[i]>ans[len])  ans[++len]=arr[i];  else{  ll pos =lower\_bound(ans+1,ans+len,arr[i])-ans;  ans[pos]=arr[i];  }    }  printf("%lld\n",len);  // }  return 0;  } |

## 8．精选技巧

### 8.1 矩阵快速幂

|  |
| --- |
| #include<cstdio>  #include<iostream>  #include<vector>  using namespace std;  typedef long long ll;  typedef vector<long long>vec;  typedef vector<vec>mat;  const ll N=1000000009;  mat mul(mat a,mat b) //矩阵乘法  {  mat c(a.size(),vec(b[0].size()));  for(ll i=0;i<a.size();i++)  {  for(ll k=0;k<b.size();k++)  {  for(ll j=0;j<b[0].size();j++)  {  c[i][j]=(c[i][j]+a[i][k]\*b[k][j])%N;  }  }  }  return c;  }  mat solve\_pow(mat a,ll n) //快速幂  {  mat b(a.size(),vec(a.size()));  for(ll i=0;i<a.size();i++)  {  b[i][i]=1;  }  while(n>0)  {  if(n&1)  b=mul(b,a);  a=mul(a,a);  n>>=1;  }  return b;  }  ll n;  void solve()  {  mat a(2,vec(2));  while(~scanf("%lld",&n)&&n!=-1)  {  a[0][0]=1,a[0][1]=1;  a[1][0]=1,a[1][1]=0;  a=solve\_pow(a,n);  printf("%lld\n",a[1][0]);  }  }  int main()  {  solve();  return 0;  } |

### 8.2 矩阵运算

|  |
| --- |
| #include <iostream>  #include <stdio.h>  #include <string.h>  #include <algorithm>  #include <math.h>  #include <queue>  #include <stack>  #include <vector>  using namespace std;  #define INF 0x3f3f3f  #define pi acos(-1.0)  #define MAX 1000010  #define N 105  struct Mat  {  int p[N][N];  };  int n;  Mat mul(Mat a,Mat b)  {  Mat c;  memset(c.p,0,sizeof(c.p));  for(int i = 0; i < n; i++)  for(int j = 0; j < n; j++)  for(int k = 0; k < n; k++)  c.p[i][j] += a.p[i][k]\*b.p[k][j];  return c;  }  int main()  {  scanf("%d",&n);  Mat A,B;  for(int i = 0; i < n; i++)  for(int j = 0; j < n; j++)  scanf("%d",&A.p[i][j]);  for(int i = 0; i < n; i++)  for(int j = 0; j < n; j++)  scanf("%d",&B.p[i][j]);  Mat C = mul(A,B);  for(int i = 0; i < n; i++)  for(int j = 0; j < n; j++)  printf("%d%c",C.p[i][j],j==n-1?'\n':' ');  } |

8.3 输入外挂

（1）整数

|  |
| --- |
| inline bool scan\_d(int &num)  {  char in;bool IsN=false;  in=getchar();  if(in==EOF) return false;  while(in!='-'&&(in<'0'||in>'9')) in=getchar();  if(in=='-'){ IsN=true;num=0;}  else num=in-'0';  while(in=getchar(),in>='0'&&in<='9'){  num\*=10,num+=in-'0';  }  if(IsN) num=-num;  return true;  } |

（2）浮点数

|  |
| --- |
| inline bool scan\_lf(double &num)  {  char in;double Dec=0.1;  bool IsN=false,IsD=false;  in=getchar();  if(in==EOF) return false;  while(in!='-'&&in!='.'&&(in<'0'||in>'9'))  in=getchar();  if(in=='-'){IsN=true;num=0;}  else if(in=='.'){IsD=true;num=0;}  else num=in-'0';  if(!IsD){  while(in=getchar(),in>='0'&&in<='9'){  num\*=10;num+=in-'0';}  }  if(in!='.'){  if(IsN) num=-num;  return true;  }else{  while(in=getchar(),in>='0'&&in<='9'){  num+=Dec\*(in-'0');Dec\*=0.1;  }  }  if(IsN) num=-num;  return true;  } |

### 8.4 数据类型的取值范围

| **数据类型** | **取值范围** |
| --- | --- |
| char | -128 ~ 127 (1 Byte，大约3位) |
| short | -32768 ~ 32767 (2 Bytes，大约五位) |
| unsigned short | 0 ~ 65536 (2 Bytes，大约五位) |
| int | -2147483648 ~ 2147483647 (4 Bytes，大约十位) |
| unsigned int | 0 ~ 4294967295 (4 Bytes，大约十位) |
| long | == int |
| long long | -9223372036854775808 ~ 9223372036854775807 (8 Bytes，大约十九位) |
| unsigned long long | 0 ~ 18446744073709551615（大约二十位） |
| \_\_int64 | == long long |
| unsigned \_\_int64 | == unsigned long long |
| double | 1.7 \* 10^308 (8 Bytes) |

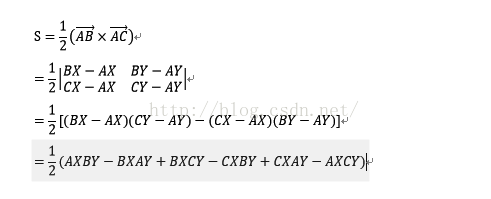
### 8.5 基本数学公式

1.   海伦公式求面积

http://img.blog.csdn.net/20160713164451694?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

**公式描述**：公式中a，b，c分别为三角形三边长，p为半周长，S为三角形的面积。

2.   矢量向量求面积



3.   点到直线的距离公式

方法一：距离公式直接求

http://img.blog.csdn.net/20160713164514773?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

**公式描述：**公式中的直线方程为Ax+By+C=0，点P的坐标为(x0,y0)。但是直线方程不是够直接。**推荐使用方法二。**

方法二：先用海伦公式求面积然后求三角形高

4.   点到线段的距离公式[或：点到线段最近的点]

有以下四种情况：

* 点在线段上，距离为0；
* 线段是一个点，用两点公式求；
* 三点构成直角三角形或者钝角三角形，那么直角或者钝角即为点到线段最近的点；
* 三点构成锐角三角形，那么距离为三角形的高，点到线段最近的点。

|  |
| --- |
| #include <cmath>  #include <queue>  #include <vector>  #include <cstdio>  #include <string>  #include <cstring>  #include <iomanip>  #include <iostream>  #include <algorithm>  using namespace std;  //#pragma comment(linker, "/STACK:1024000000,1024000000")  #define FIN freopen("input.txt","r",stdin)  #define FOUT freopen("output.txt","w",stdout)  #define fst first  #define snd second  typedef \_\_int64 LL;  //typedef long long LL;  typedef unsigned int uint;  typedef pair<int, int> PII;  const int INF = 0x3f3f3f3f;  const double eps = 1e-6;  const int MAXN = 3 + 5;  const int MAXM = 600 + 5;  // 判断浮点数与0的大小关系  int sgn(double x) {  if (fabs(x) < eps) return 0;  if (x < 0) return -1;  else return 1;  }  struct Point {  double x, y;  Point() {}  Point(double \_x, double \_y) : x(\_x), y(\_y) {}  } pc, p[MAXN];  typedef Point Vect;  /\*\*  \* 两点距离公式  \*  \*/  double getDist(const Point &p1, const Point &p2) {  int tx = p1.x - p2.x;  int ty = p1.y - p2.y;  return sqrt(tx \* tx + ty \* ty);  }  /\*\*  \* 由两点求向量  \*  \*/  Vect getVect(const Point& p1, const Point &p2) {  return Vect(p2.x - p1.x, p2.y - p1.y);  }  /\*\*  \* 求矢量叉积  \* @param v1 [description]  \* @param v2 [description]  \* @return [description]  \*/  double xmult(const Vect& v1, const Vect& v2) {  return v1.x \* v2.y - v2.x \* v1.y;  }  /\*\*  \* 矢量叉积求面积  \*  \*/  double getArea1(const Point &p0, const Point &p1, const Point &p2) {  Vect v1 = getVect(p0, p1);  Vect v2 = getVect(p0, p2);  return 0.5 \* getVectProduct(v1, v2);  }  /\*\*  \* 海伦公式求面积  \*  \*/  double getArea2(const Point &p0, const Point &p1, const Point &p2) {  double p0p1 = getDist(p0, p1);  double p0p2 = getDist(p0, p2);  double p1p2 = getDist(p1, p2);  double x = (p0p1 + p0p2 + p1p2) / 2.0;  return sqrt(x \* (x - p0p1) \* (x - p0p2) \* (x - p1p2));  }  /\*\*  \* 利用海伦公式或者叉积公式求点到直线的距离  \* @param p0 [点]  \* @param p1 [直线上的点1]  \* @param p2 [直线上的点2]  \* @return [点到直线的距离]  \*/  double point2line(const Point &p0, const Point &p1, const Point &p2) {  double area = getArea1(p0, p1, p2);  // double area = getArea2(p0, p1, p2);  double p1p2 = getDist(p1, p2);  return 2 \* area / p1p2;  }  /\*\*  \* 获取点到线段的最小距离  \* @param p0 [点]  \* @param p1 [线段端点1]  \* @param p2 [线段端点2]  \* @return [点到线段的距离]  \*/  double point2lineSeg\_Near(const Point &p0, const Point &p1, const Point &p2) {  double p0p1 = getDist(p0, p1);  double p0p2 = getDist(p0, p2);  double p1p2 = getDist(p1, p2);  // 点在线段上  if (sgn(p0p1 + p0p2 - p1p2) == 0) return 0;  // 线段两个端点p1，p2重合  if (sgn(p1p2) == 0) return p0p1;  // ∠p0p1p2 为直角或者钝角  if (p0p2 \* p0p2 >= p0p1 \* p0p1 + p1p2 \* p1p2) return p0p1;  // ∠p0p2p1 为直角或者钝角  if (p0p1 \* p0p1 >= p0p2 \* p0p2 + p1p2 \* p1p2) return p0p2;  // ∠p0p1p2 和 ∠p0p2p1 都是锐角,等价于求点到直线的距离  return point2line(p0, p1, p2);  }  /\*\*  \* 求点到线段的最长距离  \* @param p0 [点]  \* @param p1 [线段端点1]  \* @param p2 [线段端点2]  \* @return [最长距离]  \*/  double point2lineSeg\_Far(const Point &p0, const Point &p1, const Point &p2) {  double p0p1 = getDist(p0, p1);  double p0p2 = getDist(p0, p2);  return max(p0p1, p0p2);  }  int T;  double R;  int main() {  #ifndef ONLINE\_JUDGE  FIN;  #endif // ONLINE\_JUDGE  scanf("%d", &T);  while (T --) {  scanf("%lf %lf %lf", &pc.x, &pc.y, &R);  for (int i = 0; i < 3; i ++) {  scanf("%lf %lf", &p[i].x, &p[i].y);  }  double mi[3], ma[3];  mi[0] = point2lineSeg\_Near(pc, p[0], p[1]);  mi[1] = point2lineSeg\_Near(pc, p[0], p[2]);  mi[2] = point2lineSeg\_Near(pc, p[1], p[2]);  ma[0] = point2lineSeg\_Far(pc, p[0], p[1]);  ma[1] = point2lineSeg\_Far(pc, p[0], p[2]);  ma[2] = point2lineSeg\_Far(pc, p[1], p[2]);  bool suc = false;  for (int i = 0; i < 3; i ++) {  if (sgn(mi[i] - R) <= 0 && sgn(ma[i] - R) >= 0) {  suc = true;  break;  }  }  puts(suc ? "Yes" : "No");  }  return 0;  } |