Table of Contents

Hrbust ACM Book

Todo List

常见基础错误...ok

Author: 程星亮

- 语言基本用法
  - C++精度控制
  - 初始化
  - 。。。
- 。。。

基础

Author：李川皓

- 枚举
- 模拟
- 排序
- DFS
- BFS
- 二分

数学

Author：高放

- 欧几里得
- 扩展欧几里得 ...ok
- 中国剩余定理...ok
- 素数筛...ok
- 素数判定
- 欧拉函数计算

Author：王昊天

- 莫比乌斯函数计算
- 高斯消元
- 概率相关

图论...ok

Author：高云峰

- 强连通分量
- 双联通分量
- 拓扑排序
- 割点和桥
- 最短路 Dijkstra
- 最短路 SPFA
- 最短路 Floyed
- LCA
- 最小生成树 Prim
- 最小生成树 Kruskal
- 最大流 Dinic
- 最小割
- 费用流

Author：王昊天

- 差分约束系统
- 2-SAT
- 匈牙利算法
- KM算法

计算几何...ok

Author：林凡卿

- 叉积和点积...ok
- 多边形相关...ok

- 凸包...ok
- 扫描线...ok
- 半平面交...ok
- 。。。

数据结构...ok

Author：郭昊

- 并查集...ok
- 树状数组...ok
- 线段树...ok
- 二维线段树...ok
- 字典树...ok
- Splay...ok
- ST表...ok
- Link-Cut Tree...ok
- 树链剖分...ok

字符串...ok

Author：高放

- KMP...ok
- AC自动机...ok
- 后缀数组...ok
- 最长公共子串
- 最长回文子串...ok

动态规划...ok

Author：王昊天

- 基础动态规划...ok
- 树形DP...ok

- 状态压缩DP...ok
- 概率DP
- 动态规划优化...ok

博弈论

Author：李川皓

- Nim博弈
- SG函数
- 。。。。。

搜索...ok

Author：李宝佳

- A*...ok
- IDA*...ok
- 搜索优化...ok
- 记忆化搜索...ok

组合数学...ok

Author：高胜杰

- 容斥原理...ok
- 母函数...ok
- Polya原理...ok

语言...ok

Author：郭昊

- STL相关...ok

语言相关

---

author：郭昊

常见基础错误

手(shou)误(jian)

- 出错特征：程序执行流程出乎意料，结果不正确。
- 出错样例：

```
for (int i = 0; i < n; i++) {
  if (i = n) printf("%d\n", i);
  else printf("%d ", i);
}
```

- 治疗方法：剁手。多剁两次就记住了。

浮点数判等

- 出错特征：WA到死。
- 出错样例：

```
double a = 1/3*3;
double b = 1;
if (a == b) {
  printf("Yes");
}
```

- 治疗方法：

```
const double eps = 1e-5;
double a = 1/3*3;
double b = 1;
if (abs(a-b) < eps) {
  printf("Yes");
}
```

- 注意点：eps到底取多少？一般在1e-5到1e-8之间。有些题目卡eps。（就是莫名其妙的一个wa一个ac）

声明变量和使用变量太远.....

- 出错特征：Output Limit Error 或 WA 或 RE 或 TE 或 机器爆炸。
- 出错样例：

> 题目：计算a+b。
>
> 输入：t组数据，每组测试数据包含两个数a，b。
>
> 输出：对于没组数据，输出a+b的值。每两组输出之间换行隔开

```
#include <cstdio>
bool isFirst, t;
int a, b;
int main() {
  isFirst = true;
  scanf("%d", &t);
  while (t--) {
      scanf("%d%d", &a, &b);
      if (isFirst) {
          isFirst = false;
      } else {
          puts("");
      }
      printf("%d\n", a+b);
  }
  return 0;
}
```

```
数据：
3
1 2
2 3
3 4
```

- 治疗方法：先睡一觉。写出这种代码，你一定是太累了。

忘记初始化

- 出错特征：WA
- 出错样例：比如每次使用vis之前没有清false之类。
- 治疗方案：
  - 每个变量定义的同时就初始化。
  - 提交代码之前，检查所有定义的变量是否已经初始化。

数组开小了

- 出错特征：差别不大的会WA或TE。差别大的会RE。
- 出错样例：眼花手抖导致的数组少个0。，“树”类问题数组只开了n（应该要4n)
- 治疗方案：数组开的足够大。

Ctrl+C && Ctrl+V

- 出错说明：复制一段代码然后粘贴再修改的方式编程。常常出现没修改干净的问题。常出现在搜索题或输出图形的题。
- 出错特征：WA
- 出错样例：暂缺
- 治疗方法：
  - 不要复制代码。把能重用的地方封装成函数然后再用（往往比较费时间）
  - 采用复制代码方式。修改后然后检查**3**遍，当WA的时候，重点检查此处。优先重写此处。（即将复制的代码列为高危代码）
- PS：写工程的时候，不要复制代码……除非你的工程不需要维护（比如作业）

建议的代码书写方式

- 良好的代码风格。包括但不限于
  - 有意义的变量名（起名字真的是个技术，没那麼简单，就能找到，~~聊得来的伴~~）
  - 缩进
  - 大括号的位置（选择一个风格保持统一）
  - 有必要的空格使代码清晰（比如：`int a = (10*3/2 + 10)/3`）
  - C++式的变量声明方式（即等到要用的时候再声明，不要在函数开头声明一堆，然后再用）
- 防御性编程
  - 声明变量后立即初始化，不管是否必要。
  - 指针不用后立即清空，不管是否必要。

写在最后

“年代久远”的缘故，上面的错误已经找不到原来的代码了。 如果各位刷题时出现闹鬼代码，欢迎编辑在此，以供后人抓鬼。

基础知识

---

author：李川皓

枚举

简述

顾名思义，枚举便是依次列举出所有可能产生的结果，根据题中的条件对所得的结果进行逐一的判断，过滤掉那些不符合要求的，保留那些符合要求的，也可以称之为暴力算法。

应用场合

在竞赛中，并不是所有问题都可以使用枚举算法来解决（事实上，只有少数），只有当问题的所有解的个数不太多时，并在我们题目中可以接受的时间内得到问题的解，才可以使用枚举。

例题

例题1

题意：输入一个正整数n，按从小到大的顺序输出形如a/b=n的表达式，其中100<=a<=999，2<=n<=79，且b必须是素数。
思路：a和n的范围都很小，我们可以使用枚举，代码如下：

```
#include <stdio.h>
#include <cstring>
#include <algorithm>
using namespace std;

bool check(int x)
{
    bool flag = true;
    for (int i = 2; i*i <= x; ++i) {
        if (x%i == 0) {
            flag = false;
            break;
        }
    }
    return flag;
}

int main()
{
    int n;
    while (~scanf("%d", &n)) {
        for (int i = 100; i < 1000; ++i) {
            for (int j = 1; j < i; ++j) {
                if (check(j) && i%j == 0 && i/j == n) {
                    printf("%d/%d=%d\n", i, j, n);
                }
            }
        }
    }
    return 0;
}
```

例题2.hrbust 1565

题意：给出一个数字n(n<=1000000)请你计算出除了1和n之外n的因子数的个数。
要求：Time Limit: 1000 MS , Memory Limit: 10240 K
思路：首先我们通过分析n的范围和时限（1000ms）可以知道这道题可以使用枚举，我们可以通过枚举1到n中的每个整数，并判断该数是否是n的因子，使用一个count变量进行统计，时间复杂度是O(n)，代码如下：

```
int count=0;
for(int i=2;i<n;i++)
    if(n%i==0)
        count++;
printf("%d\n",count);
```

拓展：此题时限是1000ms，使用O(n)的算法可以过，如果把n的范围改成n<$${10^{12}}$$呢，O(n)的算法就会超时，那么应该怎么办？我们考虑这样一个规律：首先我们假设n是16，那么n的因子分别是1,2,4,8,16，我们可以得到这样一个规律：如果**a**是**n**的因子，那么**n/a**一定也是**n**的因子！所以我们可以将枚举的范围缩小到$$[1,\sqrt n]$$，这样我们可以把枚举的复杂度降低到O($$\sqrt n$$)。

模拟

简述

记得我刚开始参与ACM竞赛的时候，最喜欢做的就是模拟题，为什么？因为模拟题很少会用到算法，事实也是如此。模拟题考验的是我们的代码实现能力，简单的模拟题基本不用想，就是水题，但是比较难的模拟题，需要我们仔细思考，要寻找出一种能够在代码上相对来说比较好实现并且可以解决这道题的数据结构，这考验的是我们对数据结构的掌握和对题意向代码的转化。模拟题很耗时，比赛中一般会在2~3小时的时候开始做模拟题，耗时大概30~100分钟不等，只要静下心，考虑到题中的所有坑点，一般模拟题都可以AC。在这里给大家推荐几道我做过的不错的模拟题。

解题技巧

- 模拟的题型，基本难度不大，关键读懂题意。
- 赛场上不要着急于去快速的解决模拟题，因为这类题，一般做起来比较耗时。
- 想做好模拟题，需要有活跃的思维和对数据结构等知识的扎实的掌握，基础很重要！
- 有一些模拟题是可以通过刷题来锻炼出来解题能力的，不过太难的模拟题不推荐大家浪费太多时间在上面。
- 有一些模拟题，看似没有什么算法，很简单，但是会卡时间，需要大家想一下如何优化，所以大家不要盲目的去解决模拟题。

例题

例题1.poj 1068

题意：对于给出的原括号串，存在两种数字密码串：

1.p序列：当出现匹配括号对时，从该括号对的右括号开始往左数，直到最前面的左括号数，就是pi的值。

2.w序列：当出现匹配括号对时，包含在该括号对中的所有右括号数（包括该括号对）,就是wi的值。

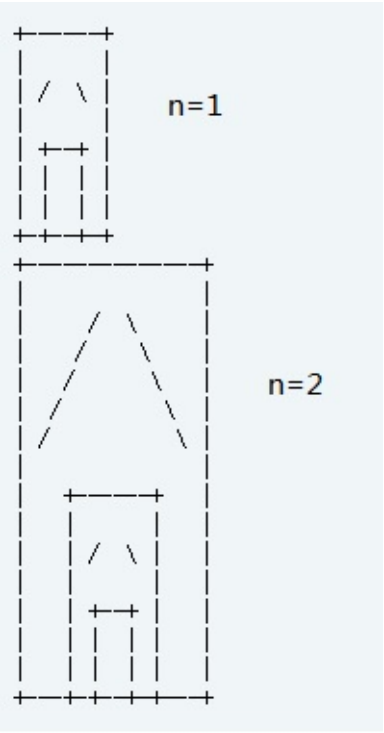
对给出的p数字串，求出对应的s串。串长限制均为20。
要求：Time Limit: 1000 MS , Memory Limit: 10000 K
思路：清楚了题意后，这道题并不是很难，直接模拟就行了，在小优的博客上学到了一个小技巧，和大家分享一下：在处理括号序列时可以把括号序列转化为01序列，左0右1，处理时比较方便
参考出处：http://user.qzone.qq.com/289065406/blog/1299127551

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

int main(void)
{
    int p[21]={0};
    int w[20];
    int str[40];
    int n;

    int cases;
    cin>>cases;
    while(cases--)
    {
        memset(str,0,sizeof(str));
        cin>>n;
        int i,j,k;
        for(i=1;i<=n;i++)
            cin>>p[i];
        for(j=0,i=1;i<=n;i++)
            for(k=0;;k++)
                if(k<p[i]-p[i-1])
                    j++;
                else if(k==p[i]-p[i-1])
                {
                    str[j++]=1;
                    break;
                }
```

```
        const int length=j;
        int count;
        for(i=0;i<length;i++)
            if(str[i]==1)
            {
                count=2;
                for(j=i-1;;j--)
                {
                    if(str[j]==0)
                    {
                        str[i]=str[j]='F';
                        break;
                    }
                    else
                        count++;
                }
                cout<<count/2<<' ';
            }
        cout<<endl;
    }
    return 0;
}
```

例题2.hrbust 2085

题意：给你一个数字 n 让你输出囧字的迭代



思路：这是一类经典的模拟题，迭代画图，我们考虑好这个囧字的迭代结构，给定一个n，从最外侧的囧字向内部构造即可，具体看代码。

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#define MAXN 1000
using namespace std;

void draw(const int n, char map[][MAXN], const int r,
const int c) //从最大的向里面构造
{
    int size = (1 << (n+2));
    map[r][c] = map[r+size-1][c+size-1] = map[r][c+size-
1] = map[r+size-1][c] = '+'; //构造四个角落
    for(int i=1; i<size-1; i++) //构造外边框
    {
        map[r][c+i] = map[r+size-1][c+i] = '-';
        map[r+i][c] = map[r+i][c+size-1] = '|';
    }
```

```
    if(n == 0) return ;
    for(int i=2; i<size/2-1; i++)   //构造内部
    {
        map[r+i][c+size/2-i] = '/';
        map[r+i][c+size/2+i-1] = '\\';
    }
    draw(n-1, map, r+size/2, c+size/4);  //迭代
}

int main()
{
    int cas, n;
    scanf("%d", &cas);
    while(cas--)
    {
        static char map[MAXN][MAXN];
        scanf("%d", &n);
        memset(map, ' ', sizeof(map));
        draw(n, map, 0, 0);
        int size = (1 << (n+2));
        for(int i=0; i<size; i++)
        {
            for(int j=0; j<size; j++) putchar(map[i][j]);
            putchar(10);
        }
    }
    return 0;
}
```

排序

简述

学习计算机的任何一门语言，都要掌握排序算法，参与竞赛更要掌握排序算法，将来找工作笔试面试的时候都会考到排序算法，所以说排序算法是十分重要的，我们要全面掌握并深入理解。

分类及比较

1.冒泡排序

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。

```
void bubble_sort(int arry[], int left, int right) {
    for(int i = left; i < right; i++) {
        for(int j = i; j < right; j++) {
            if(arry[j] > arry[j + 1]) {
                swap(arry[j], arry[j + 1]);
            }
        }
    }
}
```

时间复杂度：$$O(n^2)$$，空间复杂度：$$O(n)$$。

2.选择排序

首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

```
void swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}

void select_sort(int arry[], int left, int right) {
    for(int i = left; i < right; i++) {
        int Min = i;
        for(int j = i + 1; j <= right; j++) {
            if(arry[Min] > arry[j]) {
                Min = j;
            }
        }
        if(Min != i) {
            swap(arry[i], arry[Min]);
        }
    }
}
```

时间复杂度：$$O(n^2)$$，空间复杂度：$$O(n)$$。

3.直接插入排序

插入排序（Insertion Sort）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用inplace排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

```
void insert_sort(int arry[], int left, int right) {
    for(int i = left + 1; i <= right; i++) {
        int tmp = arry[i];
        int j = i - 1;
        while(j >= left && arry[j] > tmp) {
            arry[j + 1] = arry[j];
            j--;
        }
        arry[j + 1] = tmp;
    }
}
```

时间复杂度：$$O(n^2)$$，空间复杂度：$$O(n)$$。

4.快速排序

快速排序是各种笔试面试中常出的一类题型，其基本思想是：从序列中选取一个作为关键字，对序列排一次序，使得关键字左侧的数都比关键字小，右侧的都大于等于关键字（左右两侧的序列依然是无序的），然后 将左侧的序列按照同样的方法进行排序，将右侧序列也按照同样的方法排序，已达到整个序列有序。

```
void quick_sort(int arry[], int left, int right) {
    if(left >= right) return ;
    int i = left, j = right;
    int tmp = arry[left];
    do {
        while(arry[j] > tmp && i < j) {
            j--;
        }
        if(i < j) {
            arry[i] = arry[j];
            i++;
        }
        while(arry[i] < tmp && i < j) {
            i++;
        }
        if(i < j) {
            arry[j] = arry[i];
            j--;
        }
    } while(i != j);
    arry[i] = tmp;
    quick_sort(arry, left, i - 1);
    quick_sort(arry, i + 1, right);
}
```

时间复杂度：$$O(n\log n)$$；空间复杂度：$$O(n)$$。

5.堆排序

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

```
void Heap_just(int a[], int root, int heap_size) {
    if(root < heap_size) {
        int Min = root;
        int l_son = root << 1 | 1;
        int r_son = (root << 1) + 2;
        if(l_son < heap_size && a[Min] > a[l_son]) Min =
l_son;
        if(r_son < heap_size && a[Min] > a[r_son]) Min =
r_son;
        if(Min == root) return ;
        a[root] ^= a[Min];
        a[Min] ^= a[root];
        a[root] ^= a[Min];
        Heap_just(a, Min, heap_size);
    }
}

void build_heap(int a[], int n) {
    for(int i = n / 2; i >= 0; i--) {
        Heap_just(a, i, n);
    }
}

void Heap_sort(int a[], int n) {
    build_heap(a, n);
    for(int i = n - 1; i > 0; i--) {
        a[i] ^= a[0];
        a[0] ^= a[i];
        a[i] ^= a[0];
        Heap_just(a, 0, i);
    }
}
```

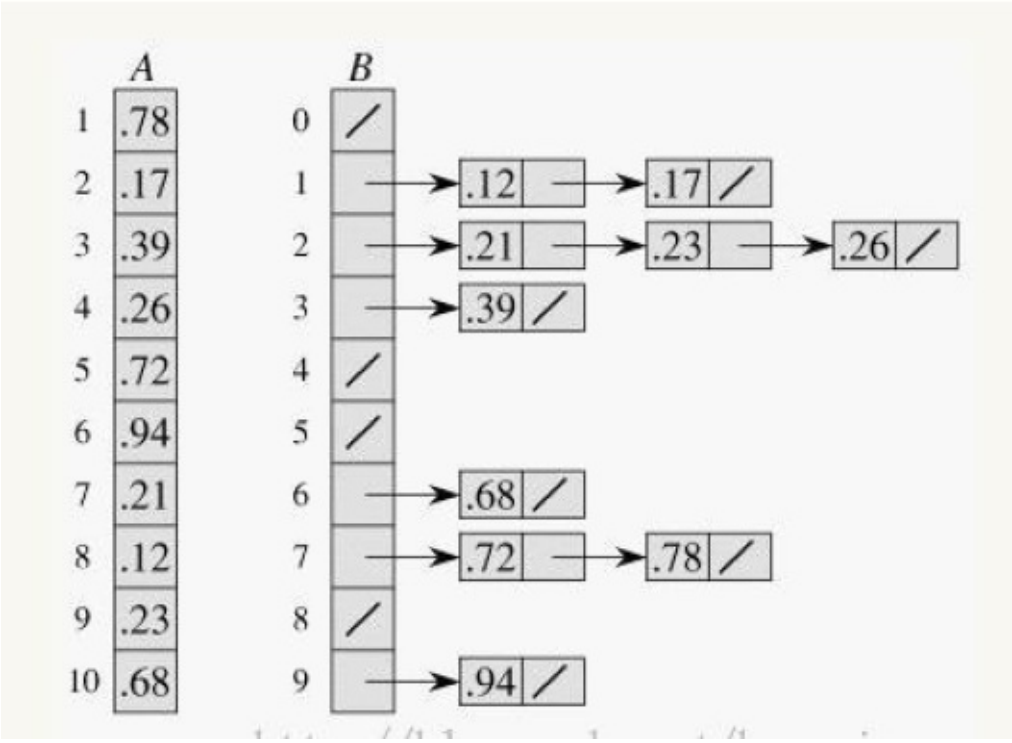
时间复杂度：$$O(n\log n)$$。

6.桶排序

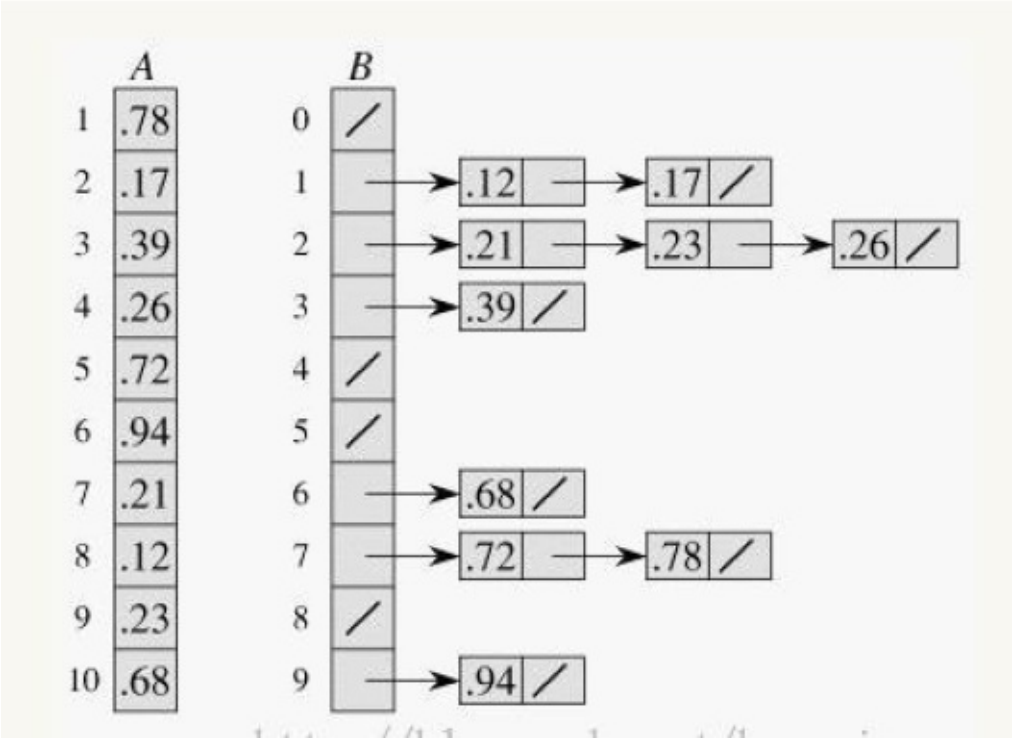
桶排序 (Bucket sort)或所谓的箱排序，是一个排序算法，工作的原理是将数组分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序）。 桶排序是稳定的，且在大多数情况下常见排序里最快的一种,比快排还要快，缺点是非常耗空间,基本上是最耗空间的一种排序算法，而且只能在某些情形下使用。

实现步骤：

1. 设置一个定量的数组当作空桶子。



2. 寻访串行，并且把项目一个一个放到对应的桶子去。
3. 对每个不是空的桶子进行排序。
4. 从不是空的桶子里把项目再放回原来的串行中。



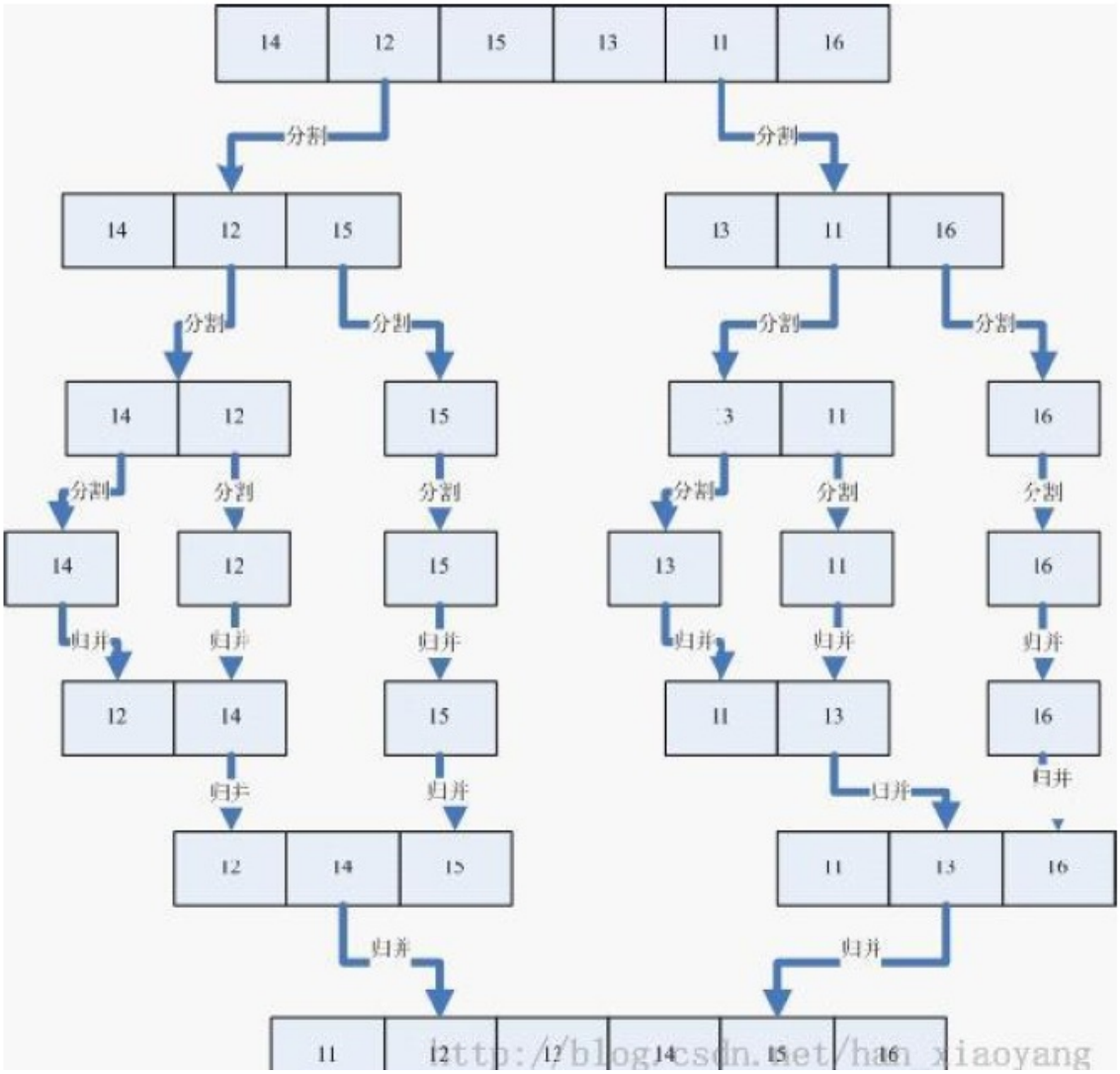
桶排序没有固定代码。
时间复杂度：$$O(n)$$。

7.归并排序

归并排序是一种稳定的算法，采用分治的思想，有序的子序列合并得到有序序列。

实现步骤：

1. 将序列分成长度为 n/2的两部分
2. 对于左右两部分采用分治的方法得到有序序列
3. 将左右两个有序序列合并得到整个有序序列


14
12
15
13
11
16
分割
归并
11
12
13
14
15
16

```
void arry_add(int arry[], int left, int mid, int right) {
    if(left >= right) return ;
    int i = left, j = mid + 1, k = 0;
    while(i <= mid && j <= right) {
        if(arry[i] <= arry[j]) {
            tmp[k++] = arry[i++];
        } else {
            tmp[k++] = arry[j++];
            cnt += (mid - i + 1);
        }
    }
    while(i <= mid) {
        tmp[k++] = arry[i++];
    }
    while(j <= right) {
        tmp[k++] = arry[j++];
    }
    for(i = 0; i < k; i++) {
        arry[i + left] = tmp[i];
    }
}
void merge_sort(int arry[], int left, int right) {
    if(left >= right) return ;
    int mid = (left + right) >> 1;
    merge_sort(arry, left, mid);
    merge_sort(arry, mid + 1, right);
    arry_add(arry, left, mid, right);
}
```

时间复杂度：$$O(n\log n)$$。

8.二分插入排序

二分（折半）插入（Binary insert sort)排序是一种在直接插入排序算法上进行小改动的排序算法。其与直接排序算法最大的区别在于查找插入位置时使用的是二分查找的方式，在速度上有一定提升。

```
void Binary_Insert_sort(int arry[], int first, int end) {
    for(int i = first + 1; i <= end; i++) {
        int low = first, high = i - 1;
        while(low <= high) {
            int mid = (low + high) >> 1;
            if(arry[mid] > arry[i]) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        int key = arry[i];
        for(int j = i; j > high + 1; j--) {
            arry[j] = arry[j - 1];
        }
        arry[high + 1] = key;
    }
}
```

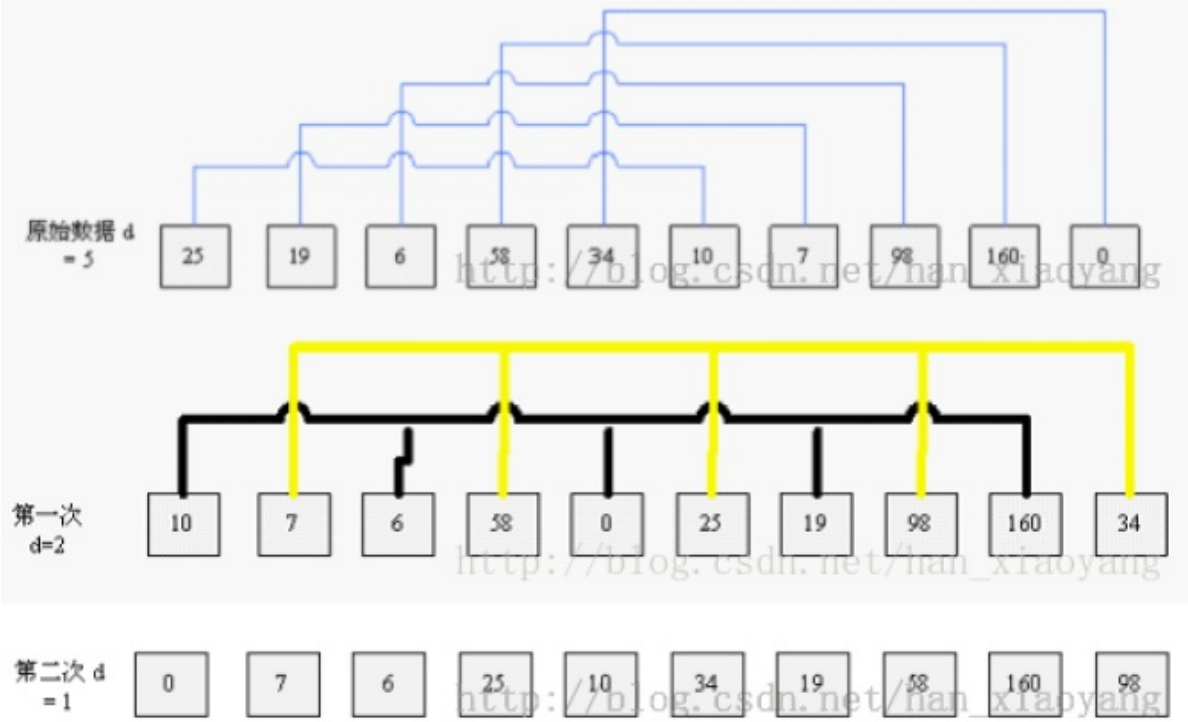
时间复杂度：$$O(n^2)$$，空间复杂度：$$O(n)$$。

9.希尔排序

希尔排序，也称递减增量排序算法，因DL · Shell于1959年提出而得名，是插入排序的一种高速而稳定的改进版本。

实现步骤：

1. 先取一个小于$$n$$的整数$$d1$$作为第一个增量，把文件的全部记录分成$$d1$$个组。
2. 所有距离为$$d1$$的倍数的记录放在同一个组中，在各组内进行直接插入排序。
3. 取第二个增量$$d2<d1$$重复上述的分组和排序。
4. 直至所取的增量$$dt=1$$，即所有记录放在同一组中进行直接插入排序为止。




```
void shell_sort(int arry[], int left, int right) {
    int n = right - left;
    int gep = 1;
    while(gep <= n) {
        gep = gep << 1 | 1;
    }
    while(gep >= 1) {
        for(int i = left + gep; i <= right; i++) {
            int temp = arry[i];
            int j = i - gep;
            while(j >= left && arry[j] > temp) {
                arry[j + gep] = arry[j];
                j -= gep;
            }
            arry[j + gep] = temp;
        }
        gep = (gep - 1) / 2;
    }
}
```

时间复杂度：小于$$O(n^2)$$。

10.鸡尾酒排序

鸡尾酒排序等于是冒泡排序的轻微变形。不同的地方在于从低到高然后从高到低，而冒泡排序则仅从低到高去比较序列里的每个元素。他可以得到比冒泡排序稍微好一点的效能，原因是冒泡排序只从一个方向进行比对(由低到高)，每次循环只移动一个项目。

```
void swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}

void cocktail_sort(int arry[], int left, int right) {
    int i = left, j = right;
    bool isSwap = false;
    do {
        for(int k = i; k < j; k++) {
            if(arry[k] > arry[k + 1]) {
                swap(arry[k], arry[k + 1]);
            }
        }
        isSwap = false;
        j--;
        for(int k = j; k > i; k--) {
            if(arry[k] < arry[k - 1]) {
                swap(arry[k], arry[k - 1]);
                isSwap = true;
            }
        }
        i++;
    } while(i <= j && isSwap);
}
```

时间复杂度：小于$$O(n^2)$$且大于$$O(n)$$。

11.计数排序

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

实现步骤：

1. 找出待排序的数组中最大和最小的元素。
2. 统计数组中每个值为i的元素出现的次数，存入数组C的第i项。
3. 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）。
4. 反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1 。

```
//只可以对非负数进行排序
int max(int a, int b) {
    return a > b ? a : b;
}

void count_sort(int arry[], int left, int right) {
    int Max = 0;
    int cnt = right - left + 1;
    for(int i = left; i <= right; i++) {
        Max = max(Max, arry[i]);
    }
    int *count = new int[Max + 1];
    int *tmp = new int[cnt + 1];
    for(int i = left; i <= right; i++) {
        ++count[arry[i]];
    }
    for(int i = 1; i <= Max; i++) {
        count[i] += count[i - 1];
    }
    for(int i = right; i >= left; i--) {
        tmp[count[arry[i]]--] = arry[i];
    }
    memcpy(arry + left, tmp + 1, (cnt)* sizeof(int));
    delete(count);
    delete(tmp);
}
```

时间复杂度：$$O(n+k)$$。

12.睡排序

```java
//java 代码
class SortThread extends Thread{
    int ms = 0;
    public SortThread(int ms){
        this.ms = ms;
    }
    public void run(){
        try {
            sleep(ms*10+10);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println(ms);
    }
}
```

竞赛中常用排序

1.库函数sort

- 头文件：`#include<algorithm>`
- 函数原型：`sort(begin,end)` 或 `sort(begin,end,cmp)`
- 实例

```cpp
#include<cstdio>
#include<iostream>
#include<algorithm>

using namespace std;

int a[5]={2,1,0,3,4};

int cmp1(int a,int b){
    return a>b;
```

```
}

int cmp2(int a,int b){
    return a<b;
}

void _print(){
    for(int i=0;i<5;i++)
        printf("%d ",a[i]);
    printf("\n");
}

int main(){
    sort(a,a+5);
    _print();
    //将输出0 1 2 3 4

    sort(a,a+5,cmp1);//使用cmp1函数中自定义的优先级进行排序
    _print();
    //将输出4 3 2 1 0

    sort(a,a+5,cmp2);//使用cmp2函数中自定义的优先级进行排序
    _print();
    //将输出0 1 2 3 4
}
```

2.结构体排序

这里我们也是用库函数sort，只不过需要该进一下自定义排序函数cmp。

```
#include<cstdio>
#include<iostream>
#include<algorithm>

using namespace std;
```

```
struct A{
    int a,b;
}aa[5];

void init(){
    aa[0].a=3;
    aa[0].b=2;
    aa[1].a=2;
    aa[1].b=4;
    aa[2].a=2;
    aa[2].b=5;
    aa[3].a=2;
    aa[3].b=3;
    aa[4].a=1;
    aa[4].b=9;
}

int cmp1(A aa,A bb){
    if(aa.a==bb.a)
        return aa.b>bb.b;
    return aa.a>bb.a;
}

int cmp2(A aa,A bb){
    if(aa.a==bb.a)
        return aa.b<bb.b;
    return aa.a>bb.a;
}

void _print(){
    for(int i=0;i<5;i++)
        printf("%d %d\n",aa[i].a,aa[i].b);
}

int main(){
    init();
```

```
    sort(aa,aa+5,cmp1);
    //使用cmp1函数中自定义的优先级进行排序,先按a降序排序，a值相等则按照b值降序排序
    _print();
    /*将输出
        3 2
        2 5
        2 4
        2 3
        1 9
    */

    sort(aa,aa+5,cmp2);
    //使用cmp2函数中自定义的优先级进行排序,先按a降序排序，a值相等则按照b值升序排序
    _print();
    /*将输出
        3 2
        2 3
        2 4
        2 5
        1 9
    */
}
```

广度优先搜索（BFS）

简述

广度优先搜索的英文简写是BFS(Breadth First Search)，属于图论中搜索算法中的一种，它所遵循的搜索策略是尽可能“广”地搜索图。下面我们通过和DFS同样的小例子来认识什么是BFS。

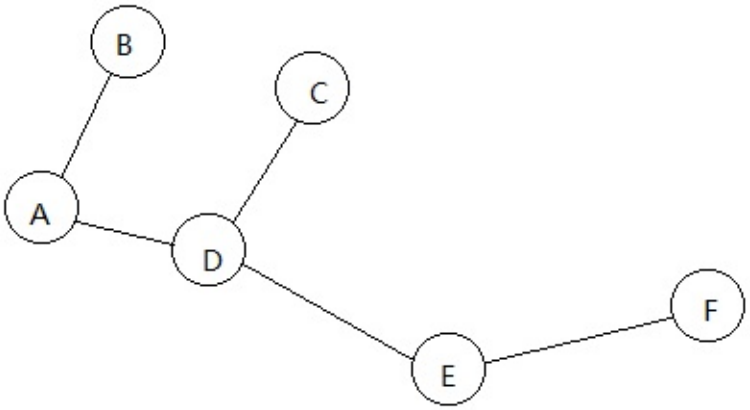
预备知识：队列

队列是一种特殊的线性表，是一种数据结构，它的特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。 它最主要的特点是先进先出。

对队列进行的最常见的操作分别是：1、插入，2、删除。对队列的操作就好像一群人在排队，排在离柜台最近的人称之为“队头”，离柜台最远的人称之为“队尾”。每新来一个人需要排队(假设我们都是有素质的人，不插队)，新来的人应该排在队尾，对吧？这就是插入操作。而柜台的办公人员会给队头的人先办公，然后依次进行，队头的人被服务完，他便会离开，这就是删除操作。

如何在代码中定义队列？

- queue< int> q; 定义一个整型队列
- q.push(a);　　将a元素插入到q队列的队尾
- q.pop();　　将q队列的队头从队列中删除
- q.front()　　q队列的队头元素
- q.back()　　q队列的队尾元素
- q.size()　　q队列的大小
- q.empty()　　返回值是1代表队列是空，否则队列不为空



我们观察如下这个无向图

我们从A点发起广度优先搜索，像DFS那样，我们设A点是1号点，B点是2号点，同理：C是3、D是4、E是5、F是6。我们用一个q数组来记录我们访问的顺序，q[i]=j代表我们访问的第i个节点是第j号节点。同时定义一个队列queue，初始时queue是空队列。由于我们从A点开始进行BFS，所以q[1]=1，这时我们把A节点加入队列，也就是将1插入队列，对吧？接下来，我们要把与A相连的点都走一遍，分别是B和D，所以q[2]=2,q[3]=4。此当访问到B点和D点时，我们要把2和4分别插入队列，此时，我们把A节点的相邻节点都走过了，就将A节点从队列中删除，此时的队列应该是{1,2,4}，删除A之后，队列变成了{2,4},这时，我们依次走2号节点(队头节点)的相邻节点，只有A节点，而我们又走过A，所以不把A加入到队列中，我们把B节点从队列中删除，此时队列中剩下一个4号节点，我们接下来就走D节点的相邻节点，走过的不插入队列，没走过的节点要插入队列，于是，我们把C节点插入了队列，队列变成了{4,3}。此时，D节点的相邻节点也都访问完毕，便把D几点从队列中删除，取出新的队头节点--C节点，访问C节点的相邻且未走过的节点，如此往复，直到最后，队列变为空，我们得到的节点访问顺序如下：

- q[1]=1;
- q[2]=2;
- q[3]=4;
- q[4]=3;
- q[5]=5;
- q[6]=6;

也就是说,我们通过从A点进行DFS，遍历整个图所经过节点的顺序是：A、B、D、C、E、F。这和我们用DFS访问时得到的顺序是一样的，但是意义不同，只是因为这个图太过于简单，造成了巧合。

模版

```
#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
    }
    return;
}

int main()
{
    ......
    return 0;
}
```

例题

例题1.hrbust 1012

题意：在一个数轴上，有一个农民位于$$n$$的位置处，有一头牛位于$$k$$的位置处，农民有三种走路方式：①若农民位于$$x$$，农民可以移动一步到$$x-1$$或$$x+1$$ ②若农民位于$$x$$，农民可以跳跃到$$2x$$处。问：农民需要最少多少步抓住那头牛？

要求：*Time Limit: 2000 MS , Memory Limit: 65536 K*

思路：基本上是一道*BFS*的入门题，我们从农民的起点开始广搜，通过农民的三种移动方式(*$$x-1$$*、*$$x+1$$*、*$$2x$$*)来向队列中插入节点，广搜到牛的位置即可，具体细节看代码吧。

```
#include <cstdio>
#include <algorithm>
#include <cmath>
#include <iostream>
#include <string.h>
using namespace std;
const int MAX=200001;
int queue[MAX],vis[MAX];

///visit函数记录是否被访问过，并且
///表示需要第几步走到这个位置
bool check(int x){
    ///检查该点是否合法，合法的条件是在数轴上，
    ///且这个点没有被访问过
    if(x>=0&&x<MAX&&vis[x]==-1) return 1;
    else return 0;
}
int BFS(int start,int end){
    memset(vis,-1,sizeof(vis));

    int front=0,back=0,now;///定义队首和队尾
    queue[back++]=start;///在队尾插入一个元素
    vis[start]=0;///该点被访问，并且是走到这一步需要0步

    while(front<=back){
        now=queue[front++];///每次取队首元素
        if(now==end) return vis[end];///如果队首元素是要找的
```

```
元素，返回结果
        if(check(now-1)){
            queue[back++]=now-1;///合法的点被插入到队尾
            vis[now-1]=vis[now]+1;///是由上一步走过来的，所以
vis要+1
        }
        if(check(now+1)){
            queue[back++]=now+1;
            vis[now+1]=vis[now]+1;
        }
        if(check(now*2)){
            queue[back++]=now*2;
            vis[now*2]=vis[now]+1;
        }
    }
}
int main()
{
    int start,end;
    while(scanf("%d%d",&start,&end)!=EOF)
    printf("%d\n",BFS(start,end));
}
```

例题2.hrbust 1143

题意：有一个泉眼，由于当地的地势不均匀，有高有低，这个泉眼不断的向外溶出水来，这意味着这里在不久的将来将会一个小湖。水往低处流，凡是比泉眼地势低或者等于的地方都会被水淹没，地势高的地方水不会越过。而且又因为泉水比较弱，当所有地势低的 地方被淹没后，水位将不会上涨，一直定在跟泉眼一样的水位上。 所有的地图都是一个矩形，并按照坐标系分成了一个个小方格，Leyni知道每个方格的具体高度。我们假定当水留到地图边界时，不会留出地图外，现在他想通过这些数据分析出，将来这里将会出现一个多大面积的湖

要求：Time Limit: 1000 MS , Memory Limit: 65536 K

思路：

- DFS那一节的例题，用BFS可以做么？

- 答案是肯定的，可以！
- 每次check找一下合法的，放进队列里面，总共有多少个合法的就是最终答案！

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <queue>
#define maxn 1000 + 10


using namespace std;

int final;
int vis[maxn][maxn];
int map[maxn][maxn];
int s[maxn][maxn];
int m,n;

int move[4][2] = {{0,1}, {0,-1}, {1,0}, {-1,0}};

struct point
{
    int x,y;
};

bool check(point a)
{
    if( a.x >= 1 && a.x <= n && a.y >= 1 && a.y <= m &&
vis[a.x][a.y] == 0 && map[a.x][a.y] <= final)
        return true;
    return false;
}


int bfs(int x,int y)
```

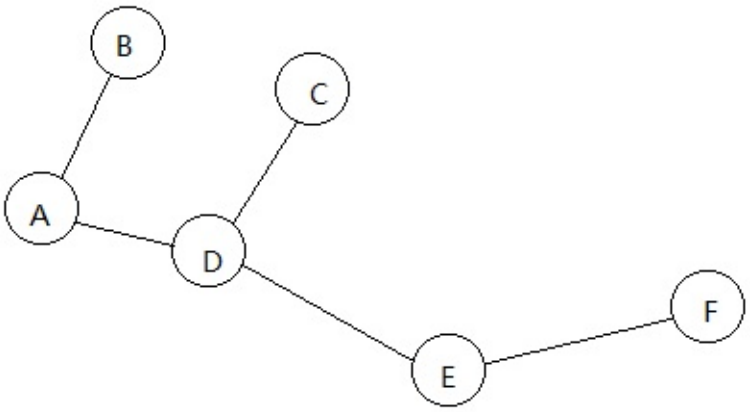
```
{
    queue<point> que;
    point now, temp;
    int count = 1;
    now.x = x;
    now.y = y;
    que.push(now);
    vis[x][y] = 1;
    s[x][y] = 1;

    while(!que.empty())
    {
        temp = que.front();
        que.pop();
      if(s[temp.x][temp.y] == 1)
       {

            for( int i = 0; i < 4; i++)
            {
                now.x = temp.x + move[i][0];
                now.y = temp.y + move[i][1];
                if(check(now))
                {
                      que.push(now);
                      vis[now.x][now.y] = 1;
                      count++;
                      s[now.x][now.y] = 1;
                }
           }
        }
    }
    return count;
}


int main()
```

```
{
    int  p1,p2;
    while(scanf("%d%d%d%d", &n, &m, &p1, &p2) != EOF)
    {
        memset(vis,0,sizeof(vis));
        memset(s,0,sizeof(s));

        for( int i = 1; i<= n; i++)
         for( int j = 1; j <= m ;j++)
             scanf("%d",&map[i][j]);
         final = map[p1][p2];
         printf("%d\n",bfs(p1,p2));

    }

}
```

深度优先搜索（DFS）

简述

深度优先搜索的英文简写是DFS(Depth First Search)，属于图论中搜索算法中的一种，它所遵循的搜索策略是尽可能“深”地搜索图。下面我们通过一个小例子来认识什么是DFS。



我们观察如下这个无向图

我们从A点发起深度优先搜索，我们设A点是1号点，B点是2号点，同理：C是3、D是4、E是5、F是6。我们用一个q数组来记录我们访问的顺序，q[i]=j代表我们访问的第i个节点是第j号节点。由于我们从A点开始进行DFS，所以q[1]=1，对吧？接下来，我们可以走A点相连的点，也就是B点或者D点(假设我们先走B点，这个顺序可以变),那么q[2]=2，走到B点之后发现和B点相连的点都走过了，这时，我们进行DFS最重要的一步：回溯，我们返回刚才走过的A点，发现A点相连的点还有D点没走，于是我们走到了D点，也就是q[3]=4，D点相连的没有走过的点有C和E，我们考虑先走C点，得到q[4]=3，这时发现C点相连的点我们也都走过了，回溯到上一节点：D点，发现D点周围还有没访问的节点，于是我们走到了E点，同时更新q数组，得到q[5]=5，然后继续走E周围没有走过的点：F点，得到q[6]=6，于是我们得到了入下的q数组：

- q[1]=1;
- q[2]=2;
- q[3]=4;
- q[4]=3;
- q[5]=5;
- q[6]=6;

也就是说,我们通过从A点进行DFS，遍历整个图所经过节点的顺序是：A、B、D、C、E、F。但是这并不是唯一的答案，从A点开始DFS还有可能是如下结果：

- ADCEFB
- ADEFCB
- ABDEFC

至于为什么，大家动脑想一下，很容易想到。

模版

```
#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(int x,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;
}
```

例题

例题1.hrbust 1143

题意：有一个泉眼，由于当地的地势不均匀，有高有低，这个泉眼不断的向外溶出水来，这意味着这里在不久的将来将会一个小湖。水往低处流，凡是比泉眼地势低或者等于的地方都会被水淹没，地势高的地方水不会越过。而且又因为泉水比较弱，当所有地势低的 地方被淹没后，水位将不会上涨，一直定在跟泉眼一样的水位上。 所有的地图都是一个矩形，并按照坐标系分成了一个个小方格，Leyni知道每个方格的具体高度。我们假定当水留到地图边界时，不会留出地图外，现在他想通过这些数据分析出，将来这里将会出现一个多大面积的湖
要求：Time Limit: 1000 MS , Memory Limit: 65536 K
思路：

- 可以枚举么？
- 枚举的话，首先该位置的高度必须低于泉眼，可是低于泉眼的地方一定可以被淹没么？不一定！
- 例如(1,5)那个位置处高度为1，但是“四面环山”，水进不来！所以还必须满足该点到达泉眼存在一条路径使得整条路径上的最高高度<=泉眼。
- 怎么做呢？
- DFS！

- 从起点开始DFS，找到所有DFS能够经过的点，个数便是答案！

```
#include <stdio.h>
#include <string.h>
bool used[1005][1005];
int a[1005][1005],n,m,p,q;
int ans;
int dx[4]={1,0,-1,0};
int dy[4]={0,1,0,-1};
void dfs(int x,int y){
   // printf("(%d,%d)\n",x,y);
    used[x][y]=1;
    ans++;
    for(int k=0;k<4;k++){
        int tx=x+dx[k],ty=y+dy[k];
        if (tx>0&&ty>0&&tx<=n&&ty<=m&&!used[tx]
[ty]&&a[tx][ty]<=a[p][q]){
            dfs(tx,ty);
        }
    }
}
int main(){
    while(scanf("%d%d%d%d",&n,&m,&p,&q)!=EOF){
        for(int i=1;i<=n;i++){
            for(int j=1;j<=m;j++){
                scanf("%d",&a[i][j]);
            }
        }
        ans=0;
        memset(used,0,sizeof(used));
        dfs(p,q);
        printf("%d\n",ans);
    }
}
```

二分

简述

在ACM竞赛中，或者说在程序员的码代码生涯中，二分作为一种基础的、能降低时间复杂度的算法，是我们必须要掌握的，也必须要熟练掌握，下面我们先通过下面这个小问题来了解什么是二分。

问题

我们定义所谓的 Special Number是指满足以下条件的数：

1. 该数里面每一个数字都不能重复，譬如123算是S.N.，而122就不算。
2. 该数不能有前导0，譬如01不算S.N，而1就是S.N。

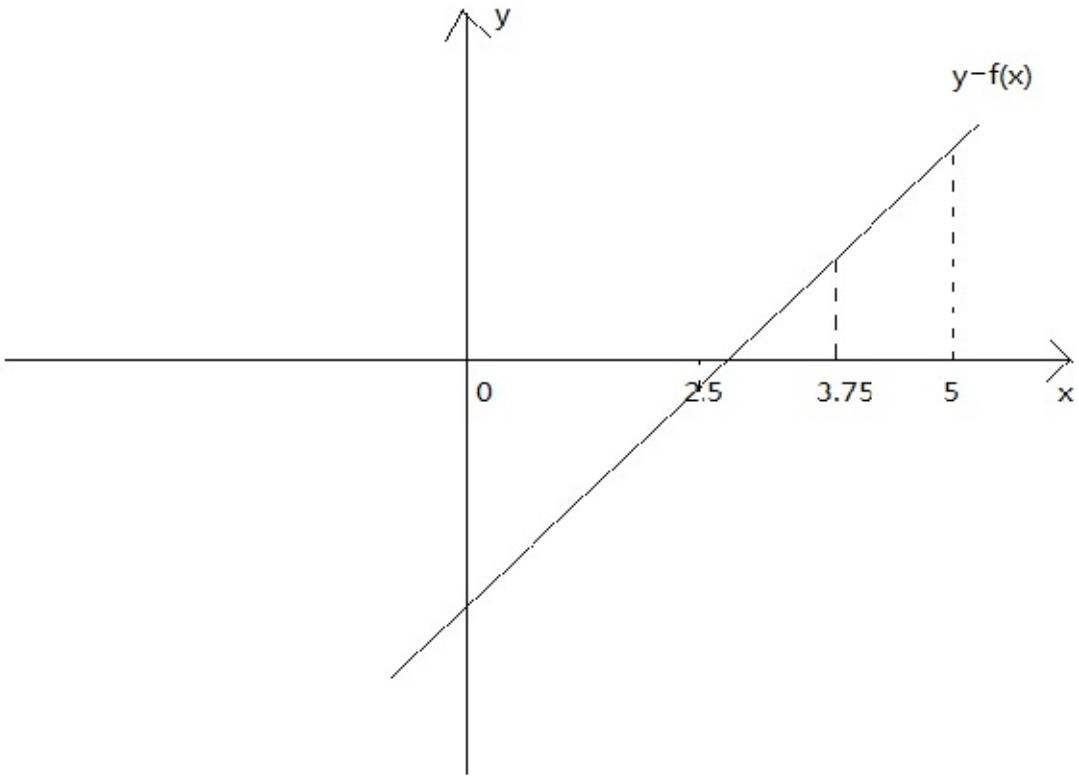
多组测试数据(大概20000组)，每组测试数据给出你一个n(n <= 10,000,000)，现在问你比n小的Special Number有多少个。

怎么做？

看了上一节书的同学，应该很容易想到使用枚举，我们可以枚举1~n-1(至于为什么不是从0开始，是因为.....请认眞看题~)，判断每一个数字是不是S.N。 但是考虑极端情况，在多组测试数据中n都是10,000,000。而给的时间只有1s，所以如果你用之前学过的枚举，毫无疑问就超时了...因为时间复杂度是O(n)。然而我们有一种新的方法来解决这道题，就是二分。

初涉二分

回顾高中学过的知识，如果求一个单调递增或者单调递减的函数与x轴的交点，即 $$f(x)=0$$ 时 $$x$$ 的值，$$x\in [0,5]$$



我们知道 $$f(0)<0$$,="" $$f(5)="">0$$，那么，$$f(x)$$ 与x轴的交点的横坐标一定落在区间 $$(0,5)$$ 内，这时我们取x为0和5的中值，即2.5，发现 $$f(2.5)<0$$，那么 $$f(x)$$="" 与x轴的交点的横坐标一定落在区间="" $$(2.5,5)$$="" 内，同理，我们取新的x为2.5和5的中值，发现="" $$f(3.75)="">0$$，那么 $$f(x)$$ 与x轴的交点的横坐标一定落在区间 $$(2.5,3.75)$$ 内.....如此往复，我们得到的答案就会越来越逼近交点。

这就是二分算法：通过不断检测左端点和右端点f(x)是否满足题意来减少未知数x的取值范围。这样最后肯定会不断逼近正确答案，直到最终左右端点非常靠近，端点值就是答案。

设计算x次能够得到答案，n是取值范围大小，即右端点-左端点。那么$$2^x$$=n, 即经过x次二分可以得到n。 所以二分的时间复杂度是log(n)，计算机里log(n)就表示log(n)，所以时间复杂度是O(log(n))。

回到我们刚才的问题，我们可以先预处理出来1~10,000,000里面的所有Special Number。然后二分这些S.N，直到找到第一个比n小的S.N，就可以知道答案了。代码如下：

```
#include <stdio.h>
#include <cstring>
#include <algorithm>
using namespace std;

int arr[1000000];

bool check(int x)
{
    bool vis[10], flag = true;
///flag作为哨兵变量出现，她用来监视这个数是不是有重复数字
    memset(vis, false, sizeof(vis)); ///memset是初始化函数，用于把数组初始化为某个值
    while (x) { ///这个while的作用是把给出的数分解出来，然后一个个判断是否出现过..例如：1234分解成1,2,3,4
        int tmp = x%10;
        x /= 10;
        if (vis[tmp]) {
            flag = false;   ///如果有重复的数字就把哨兵变量标记为false
            break;
        }
        vis[tmp] = true;
    }
    return flag;
}

int n;
bool ok(int x)
{
    return arr[x] < n;
}

int main()
{
```

```
    int tot = 0;
    for (int i = 1; i <= 10000000; ++i) { ///把范围内的符合答案的值都求出来
        if (check(i)) {
            arr[tot++] = i;
        }
    }
    //printf("_%d\n", tot);
    while (~scanf("%d", &n)) {
        int r = tot-1, l = 0, mid, res;///把左右边界初始化
        if (n <= 1) {
            puts("0");
            continue;
        }
        while (l <= r) {
            mid = (l+r)/2;
            if (arr[mid] < n) {
                l = mid+1;
                res = mid;
            } else {
                r = mid-1;
            }
        }
        printf("%d\n", res+1);///因为我们的左边是从0开始的，所以要+1
    }
    return 0;
}
```

模版

- 普通二分模版

```
int l,r,res;
//l, r初始化,问题答案的左边界和右边界的确定
while(l<=r){
  int mid=(l+r)/2;
  if (ok(mid)){
      res=mid;
      r= mid-1;
  }
  else
      l=mid+1;
}
```

但是除去常见的二分题目，还有一种需要大家注意的：浮点数型二分， 因为整数型二分当(l+r)/2的时候是向下取整，所以为了避免出现l=3，r=4，然后mid=(3+4)/2=3的情况(会造成无限循环)，所以整数型二分l=mid+1, 而浮点数型就不需要， 看一下模板：

- 浮点数二分模版

```
const double eps = 1e-7;
double l， r;
// l,r初始化，问题答案的左边界和右边界的确定
while (l + eps < r){
  double mid = (l + r) / 2.0;
  if (ok (mid)){
      l = mid;
      // r = mid;
  }
  else {
      r = mid;
      // l = mid;
  }
}
```

例题

例题1.hrbust 1530

题意：小M要举行Party，她有n个pie，已知每个pie的半径，问若要分给k个好朋友同等份的pie，每个好朋友最多可以拿多大的pie?每个好朋友得到的同等份的pie只能来自同一个pie。P.S..小M也想要分得一块同等份的pie..(答案精确到小数点后四位)
要求：Time Limit: 1000 MS , Memory Limit: 65535 K
思路：这道题的题意有点难理解，其实可以简化一下，就是有k+1个人，有 n个pie，每个pie的大小都是已知的，问每个人最多能分得多大的pie?例如pie大小为1,2,3。则如果每个人最多分得大小为2的pie，则最多分给两个人。

- 我们可以用二分，二分的是每个人可分得的pie的大小，这样最后就可以算出总共可以分给几个人，以此来逼近答案。
- 初始化的时候l=0，r=最大的pie的大小(π*$$r^2$$)，为了防止精度问题，我们最后才乘以π，表示最少每个人都没pie吃，最多每个人占据一个最大的pie。

代码：

```
#include <stdio.h>
#include <cstring>
#include <algorithm>
#include <cmath>
using namespace std;
#define eps 1e-8  ///定义精度值10^-8
#define pi acos(-1) ///定义pi值

int n, k;
int arr[10010];
int ok(double x)
{
    int ans = 0;
    for (int i = 0; i < n; ++i)
        ans += (int)arr[i]/x;       ///算出每一个pie按照x大小分可以分为几份
                                    ///然后累加起来，看一下所有
```

```
pie按照每份大小为x的时候总共可以得到几份
    return ans;
}

int main()
{
    int nCase;
    double l, r, mid;
    scanf("%d", &nCase);
    while (nCase--) {
        scanf("%d %d", &n, &k);
        l = r = 0;  ///左边界定义为0，表示最少每个人得到大小为0的pie
                    ///右边界定义为0，是为了到时候得出pie的最大值
        for (int i = 0; i < n; ++i) {
            scanf("%d", &arr[i]);
            arr[i] *= arr[i];  ///理论上我们应该算出每个pie的r*r*π
                               ///但是为了防止计算的时候出现精度误差，我们只计算r*r，最后才乘以π
            r = max(r, (double)arr[i]);   ///得到最多拿到的pie的大小，那就是最大的那个pie的大小
        }
      //  printf("%.4lf %.4lf\n", l, r);
        while (l+eps < r) {
            mid = (l+r)/2.0;
            if (ok(mid) >= k+1) l = mid;
            else r = mid;
        }
        printf("%.4lf\n", pi*l); ///用结果的r*r*π得到得到的pie的最终大小
    }
    return 0;
}
```

例题2.hdu 4282

题意：$$X^Z+Y^Z+XYZ=K,(X1,0<k<2^{31})$$。告诉我们K的大小，问符合这个等式的X,Y,Z的组合有多少种。
要求：Time Limit: 2000 MS , Memory Limit: 32768 K
错误思路：看到这道题，首先我们应该想到的是枚举，代码如下。但是观察K的范围和时限，枚举显然会超时。

```
int count=0;
for(long long x=0;x<=k;x++)
    for(long long y=x+1;y<=k;y++)
        for(long long z=2;z<=31;z++)

if(pow(x*1.0,z*1.0)+pow(y*1.0,z*1.0)+x*y*z==k)
                count++;
printf("%d\n",count);
```

这样做会超时，那我们应该怎么做？这一节我们讲的是二分，大家不妨向二分方向考虑一下。

正确思路：因为幂次的增长速度比较快，我们可以枚举X和Z,这样我们可以把三重循环降低到二重循环，然后二分搜索每一对(X,Z)是否存在对应的Y。对于A的B次方我们可以先打表储存，使用的时候直接调用，这样也可以降低一定的时间复杂度。
代码如下：

```
#include<cstdio>
#include<cstring>
using namespace std;
#define LL long long
LL mat[50001][32]={0};
bool ok(int x,int z,int cnt)
{
    int l=x+1,r=50000,mid;
    for(;l<=r;)
    {
        mid=(l+r)>>1;
```

```
        if(mat[mid][z]==0)
        {
            r=mid-1;
            continue;
        }
        if(mat[mid][z]+x*mid*z<cnt)
           l=mid+1;
        else if(mat[mid][z]+x*mid*z>cnt)
           r=mid-1;
        else
            return true;
    }
    return false;
}
int main()
{
    int k;
    for(int i=1;i<=50000;++i)
    {
        mat[i][1]=i;
        for(int j=2;j<=31;++j)
        {
            mat[i][j]=mat[i][j-1]*i;
            if(mat[i][j]>2147483648LL)
                break;
            //这个2147483648LL中的LL一定要加上

        }
    }
    while(scanf("%d",&k)!=EOF)
    {
        if(k==0)
            return 0;
        long long summ=0;
        int cnt;
        for(int x=1;x<=50000&&x<=k;++x)
```

```
            for(int z=2;z<=31;++z)
            {
                if(mat[x][z]==0)
                    break;
                cnt=k-mat[x][z];
                if(cnt-x*z<=0)
                    break;
                if(ok(x,z,cnt))
                    summ++;
            }
        printf("%I64d\n",summ);
        //这里的I64d和lld是一个意思
    }
    return 0;
}
```

动态规划

---

author：王昊天

动态规划（Dynamic Programming）基础

动态规划问题在比赛中的考点丰富，因为其代码难度一般不大，更多的是思考和发现模型的能力。所以有关于动态规划的部分将会区便于其他部分，会有更少的代码，而去展示更多的思考过程和理论知识。

动态规划和分治的区别

动态规划（Dynamic Programming）与分治方法相似，都是通过组合子问题的解来求解原问题。分治方法将问题划分为互不相交的子问题，递归的求解子问题，再讲他们组合起来，求出原问题的解。与之相反的，动态规划适用于子问题重叠的情况，即不同的子问题具有公共的子子问题。在这种情况下，分治算法会做许多不必要的工作，它会反复的求解这种公共子子问题。而动态规划算法中对每个子子问题都只求解一次，将其解保存在一个表格中，从而不用每次求解一个子子问题的时候都要重新计算，避免这种不必要的计算工作。（摘自《算法导论》第三版 第15章）

一般我们都用动态规划来求解最优化问题。他们的共同特征是，有很多个可行解，我们要在这些可行解中找到一个最大的（或者最小的）解。我们可能并不关心这个解是什么，有几个，只要这个解的评价函数对我们来说是我们想要的最大（或者最小）的可以了。

动态规划中的重要性质

1. 具有最优子结构
2. 状态无后效性

3. 有重叠子问题

最优子结构

最优子结构想表达的是，如果一个问题的最优解包含其子问题的最优解，那么我们就称这个问题有最优子结构性质。

也就是说，一个问题的最优解一定是由其各个子问题的最优解组合而成的。

关于是否满足最优子结构的判断

1. 证明问题最优解的第一个组成部分是一个选择。因为这样的选择会产生多个待解的资额问题。
2. 假定已经产生了一个最优的选择。
3. 判断根据这个最优的选择会产生哪些子问题。这些子问题应该如何表示。
4. 利用反证法。证明子问题如果不是最优解，那么可以将这个解从原问题中去掉，换上相对的最优解。从而产生了一个比假定更优的解，和假定是最优解矛盾。

重叠子问题

这个问题相对就容易理解很多。如果我们在递归求解的过程中，反复的求解了相同的子问题，那么就称这个问题有重叠子问题。

这也是动态规划在时间上相较于分治算法的关键。因为我们可以把这些重叠的子问题放在一个备忘录里记下来，避免重复不必要的计算。

状态的无后效性

状态的无后效性指的是，当一个状态被确定之后，一定不会被在其之后确定的状态所影响。也就是说，每个状态的确定都不会影响到之前已经确定的状态。 或者说，一个子问题的解是在子子问题中选择得到的，那么在计算出子问题的解之后，子问题的解不会影响到子子问题。

只有一个问题满足了这三个条件之后，我们才能考虑用动态规划算法去解决它。

设计动态规划算法的一般步骤

设计一个动态规划算法整体上分为两个步骤：

1. 设计状态表达式
2. 设计状态转移方程

状态表达式 就是对我们所说的问题的一般描述。状态表达式的选择又取决于我们如何给这个问题划分求解的阶段，也就是为这个问题的每一个阶段，设计一个独一无二的表达式来表示他们，并且这个状态表达式的选择应该是满足无后效性的。

状态转移方程 就是我们对一个问题如何由其子问题组成所做出的决策的一般描述。状态转移方程应该描述：

1. 一个状态的解可以在哪些状态中选择
2. 以何种条件在这些状态中选择出一个或多个，作为组成解状态的子状态
3. 被选择出的一个或多个状态以何种方式组合起来，形成了当前状态的解

当这两个问题被解决了之后，一个动态规划算法的框架已经基本成型了。剩下的工作就是：

1. 确定边界条件
2. 计算时间和空间复杂度
3. 判断是否满足题目要求，如果不满足应该如何优化，或者放弃这个思路

例子

说了这么多的理论知识，需要一个例题来看一下都是怎么被运用进去的。

> 有 `n` 个重量和价值分别问$$w_i$$和$$v_i$$的物品。从这些物品中选出总重量不超过 `W` 的物品，求所有挑选方案中价值总和的最大值。
> 数据范围：
>
> - $$ 1 \leq n \leq 100$$
> - $$ 1 \leq w_i, v_i \leq 100 $$
> - $$ 1 \leq W \leq 10000 $$

现在显然，我们的问题是，有 `n` 个物品，放进 `W` 的背包里。我们可以把这个问题记作：`(n, W)` 。这个也就是我们要最终求解问题。这个问题，可能的子问题有多少种呢？如果我们把最后一个物品的重量和价值分别记作 `lw` 和 `lv` 的话。如果我

们考虑选了这个物品进入最后的背包，那么子问题的范围应该是：`(0, n-lv)` 到 `(n-1, n-lv)` 。如果我们考虑不放入背包的话，`(0, n)` 到 `(n-1, n)` 。只不过这些问题在被推导出最终解的时候的方式是不一样的。

那么这些问题显然是满足我们上面说的三个条件的，有最优子结构、没有后效性和有重叠子问题。（这部分可以自己去考虑一下）。

我们可以根据上面的这个表示方法，设计出我们的状态表达式。我们用 `dp[i][j]` 去表示，当前考虑到第 `i` 个物品，放入一个大小是 `j` 的背包里所得到的最大的价值。这个状态的定义和我们上面对子问题的划分方式是一样的。

- 那么这个状态的的解可以在哪些解中选择呢？考虑我们的状态的表示，显然应该是 `dp[i-1][k]` 。如果选择要了第 `i` 的物品的话， `k` 的取值范围应该是 $$[0,j-w_i]$$；如果选择不要第 `i` 个物品的话， `k` 的取值范围应该是 $$[0,j]$$。
- 条件当然是在这些状态里面选择一个使最后价值最大的了。如果选择了第 `i` 个物品的话，结果应该是子问题的价值加上第 `i` 个物品的价值；不选的话，就应该是子问题的价值。
- 挑选出了一个之后，按照上面说的方法选择是否和第 `i` 个物品的价值求和。

所以我们就得到了我们的状态转移方程：

$$ dp[i][j] = max(max*{0 \leq k \leq j}{dp[i-1][k]}, max*{0 \leq k \leq j-v[i]}{dp[i-1][k]+w[i]}) $$

边界条件也很容易确定，当一件东西都没有的时候，也就是 `dp[0][0]` 的时候，价值显然是 `0` 。其他的时候都还没被计算到，可以赋值称无穷大，也可以判断一下是否被计算到。

时间复杂度方面，每个状态都被算了一次，每次计算枚举了 `j` 种状态，所以时间复杂度是$$O(n^2W)$$；空间复杂度是$$O(nW)$$。是满足题意的。

基础动态规划

基础动态规划模型：

1. 背包模型
2. 最长公共子序列
3. 最长上升子序列

背包模型

背包模型构成了动态规划中的一大类问题。比赛中的很多问题都是通过背包模型变形得到的。所以，背包模型在基础的 dp 里是非常重要的一部分。

01背包

01背包是所以背包问题的基础，所有和背包相关的问题都能找到01背包的影子。01背包问题可以被一般化为这样的一个模型。

题目模型

有 $$N$$ 件物品和一个容量为 $$V$$ 的背包。第 $$i$$ 个物品的体积和价值分别是 $$C_i$$ 和 $$W_i$$ 。求解将哪些物品放进包里可以使价值最大。

题目特点

这类题目的很明显的特点是，每个物品都只有一个，可以选择要或者不要。

状态表达式和状态转移方程

状态表达式：$$dp[i][j]$$ 去表示前 $$i$$ 个物品恰好放入一个容量为 $$j$$ 的背包里所得到的最大的价值。
状态转移方程：$$ dp[i][j] = max(dp[i-1][j], dp[i-1][j-C_i]+W_i) $$

这里很显然的是，`max` 表达式的前半部分表示的是不选择第 `i` 个物品的情况，而后半部分表示的是选择了第 `i` 个物品的情况。

实现

```
for (int i = 0; i <= V; i++) dp[0][i] = 0;
for (int i = 1; i <= n; i++) {
    for (int j = C[i]; j <= V; j++) {
        dp[i][j] = max(dp[i-1][j], dp[i-1][j-C[i]]+W[i]);
    }
}
```

复杂度分析

时间复杂度：$$ O(NV) $$
空间复杂度：$$ O(NV) $$

优化！

这里的时间复杂度已经没有办法再优化了，可是空间复杂度却还有优化的余地。考虑我们的第二层循环，这里的每一个 `dp[i][j]` 的取值是都只和第二维比 `j` 小的元素有关。那么是否可以通过改变枚举方向的方式，来把第一维用来保存当前位置的一列删掉呢。显然是可以的。考虑这样的代码：

```
for (int i = 0; i <= V; i++) dp[i] = 0;
for (int i = 1; i <= n; i++) {
    for (int j = V; j >= C[i]; j--) {
        dp[j] = max(dp[j], dp[j-C[i]]+W[i]);
    }
}
```

这里唯一的变化是改变了 `j` 的枚举方向。可以仔细思考一下。

完全背包

题目模型

有 $$N$$ 件物品和一个容量为 $$V$$ 的背包。第 $$i$$ 个物品的体积和价值分别是 $$C_i$$ 和 $$W_i$$ 。每个物品能无限制的使用任意个。求解将哪些物品放进包里可以使价值最大。

题目特点

大体和01背包类似，唯一不同的地方是每种物品有无限个。此时每种物品的策略已经不再是要或者不要了，而变成了要0个，要1个，要2个……

状态表达式和状态转移方程

状态表达式方面和01背包是一样的，依旧用$$dp[i][j]$$ 去表示前 $$i$$ 个物品恰好放入一个容量为 $$j$$ 的背包的最大价值。我们就可以很容易的按照原本的思路，写出状态转移方程：$$dp[i][j] = max_{0 \leq kC_i \leq V}{dp[i-1][V-kC_i]+KW_i}$$

复杂度分析

时间复杂度：$$ O(NV\sum\frac V{C_i}) $$
空间复杂度：$$ O(NV) $$

优化！转化成**01**背包！

01背包是所有背包的基础，也就是说所有的背包相关问题都能被转化成01背包的模型。即比如说这个问题，如果把每个物品不当成是无限个的，而是最多有$$\lfloor \frac V{C_i} \rfloor$$件的话，就可以把一种东西拆成$$\lfloor \frac V{C_i} \rfloor$$件东西。然后就可以被转化成了而基础的01背包问题。这是这种方法的时间复杂度和上面的方法是一样的，也是不能接受的。我们需要一个更加高效的拆分方法。

这里应该很容易就可以考虑到我们经常运用到的和二进制有关的拆分方法，也就是把第$$i$$种物品拆成费用为$$C_i2^k$$，价值为$$W_i2^k$$的若干件物品。这样每种物品不再被拆成了$$\lfloor \frac V{C_i} \rfloor$$个，而是$$\log_2\lfloor \frac V{C_i} \rfloor$$个。

再优化！

考虑下面这样的代码：

```
for (int i = 0; i <= V; i++) dp[i] = 0;
for (int i = 1; i <= n; i++) {
    for (int j = C[i]; j <= V; j++) {
        dp[j] = max(dp[j], dp[j-C[i]]+W[i]);
    }
}
```

可以看到，这段代码和01背包的空间优化版本是不是特别像？对的。首先要考虑在01背包中第二层循环是$$V \to C_i$$的原因，因为每件物品都只有一个，当前的选择不能被前面的选择所影响，如果是按照$$C_i \to V$$的顺序的话，当计算到一个值的时候，前面所用到的$$j-C[i]$$的这个值可能是已经选择了当前物品的情况，所以用来先计算体积大的部分（因为无后效性）。
而在这个问题中，显然是应该被前面计算的值所影响的，因为每件物品有无限个。这个应该很容易理解到。

多重背包

题目模型

有 $$N$$ 件物品和一个容量为 $$V$$ 的背包。第 $$i$$ 个物品的体积和价值分别是 $$C_i$$ 和 $$W_i$$ 。每个物品最多只有 $$M_i$$ 个物品可用。求解将哪些物品放进包里可以使价值最大。

题目特点

和完全背包很相似，唯一不同的地方就是每个物品的最大个数不再是 $$\frac V{C_i}$$，而变成了 $$ min(\frac V{C_i},M_i)$$。其他的部分都和完全背包是一样的。

关于背包的其他问题，可以参考背包九讲。

最长公共子序列（LCS）

题目模型

给定两个序列 $$X=\langle x_1, x_2, \ldots, x_m\rangle$$ 和 $$Y=\langle y_1, y_2, \ldots, y_n\rangle$$，求 $$X$$ 和 $$Y$$ 的最长公共子序列。

- 一个给定序列的子序列，就是将给定序列中零个或多个元素去掉之后得到的结果。
- 给定两个序列 $$X$$ 和 $$Y$$，如果 $$Z$$ 既是 $$X$$ 的子序列，又是 $$Y$$ 的子序列，我们称它是 $$X$$ 和 $$Y$$ 的公共子序列。

分析

首先是阶段的划分，我们把这两个序列的一个前缀，作为一个阶段。那么我们可以得到阶段的表示 $$C[i][j]$$ 。表示了，序列 $$X_i$$ 和 $$Y_j$$ 的LCS。（前缀的定义：给定一个序列 $$X=\langle x_1, x_2, \ldots, x_m\rangle$$，对应的 $$i=0,1,\ldots,m$$，定义 $$X$$ 的第 $$i$$ 前缀 $$X_i=\langle x_1, x_2, \ldots, x_i\rangle$$。）

按照这个阶段的划分，再来讨论这个问题是否存在最优子结构。这个显然是存在的。

> 定理**1**（LCS的最优子结构） 令 $$X=\langle x_1, x_2, \ldots, x_m\rangle$$ 和 $$Y=\langle y_1, y_2, \ldots, y_n\rangle$$ 为连个序列，$$Z=\langle z_1, z_2, \ldots, z_k\rangle$$ 为 $$X$$ 和 $$Y$$ 的任意 LCS。
>
> 1. 如果 $$x*m=y_n$$，则 *$$z_k=x_m=y_n$$* 且 *$$Z*{k-1}$$* 是 *$$X{m-1}$$* 和 *$$Y*{n-1}$$* 的一个LCS。
> 2. 如果 $$x*m \neq y_n $$，那么 *$$z_k \neq x_m $$* 意味着 *$$Z$$* 是 *$$X*{m-1}$$* 和 $$Y$$ 的一个LCS。
> 3. 如果 $$x*m \neq y_n $$，那么 *$$z_k \neq y_n $$* 意味着 *$$Z$$* 是 *$$X$$* 和 *$$Y*{n-1}$$* 的一个LCS。

满足了最优子结构之后，显然是满足无后效性的。最后我们要思考的问题，就是是否有重叠子问题。从上面的定理我们可以发现，在求 $$X=\langle x_1, x_2, \ldots, x_m\rangle$$ 和 $$Y=\langle y_1, y_2, \ldots, y_n\rangle$$ 的 LCS 的时候，我们需要求解一个或两个子问题。

1. 如果 $$x*m=y_n$$，我们就要求解 *$$X*{m-1}$$* 和 $$Y_{n-1}$$ 的一个LCS。然后把 $$x_m=y_n$$ 追加到这个序列的末尾。
2. 如果 $$x*m \neq y_n$$，我们就要求分别求解 *$$X*{m-1}$$* 和 $$Y$$ 的一个LCS ，和 $$X$$ 和 $$Y_{n-1}$$ 的一个LCS。并取较长的一个。

上面的全部分析，已经足以让我们得到状态表达式和状态转移方程了。这里就不说下去了。相信自己一定可以写出来。

最长上升子序列

最长上升子序列的问题，使用dp的解法并不是最优的。这里只是展示一个dp的思路。（作为一个dp的例题，希望可以帮助理解dp的基础内容。）

题目模型

给定一个序列 $$X=\langle x_1, x_2, \ldots, x_m \rangle$$，求 $$X$$ 的一个最长的子序列 $$Y=\langle y_1, y_2, \ldots, y_n \rangle$$，满足 $$y_1 < y_2 < \ldots < y_n$$。

分析

这个问题和上个问题有点不一样的地方是，我们对于阶段的划分不能简单的使用这个序列的前缀，因为只考虑一个前缀的最长上升子序列的话，是不能知道是否应该把当前要考虑的元素添加到一个前缀的最长上升子序列的末尾的。

所以，我们需要在状态的设计之中加入有关于一个最长上升子序列的最末尾元素的信息。（因为只要知道最末尾元素是什么，就可以判断当前元素是否可以被接在这个序列的后面了。）由此，状态的表达就变成了：$$dp[i]$$ 来表示以 $$x_i$$ 结尾的最长上升子序列的长度。

那么我们来考虑刚才说的状态转移， $$dp[i] = max_{1 \leq j < i, x_j < x_i} {dp[j]}+1$$。

树形DP

树形 DP 就是在树上做的 DP，大部分都是用子节点的信息推导出父节点的信息。在考虑问题可以被抽象出一个树状的模型的时候，可以尝试考虑一下树形DP。

树的表示

在树形 dp 中的树的形状各异，所以我习惯用 `vector` 来存储一个节点的所有子节点。用 `push_back` 方法去加入一个子节点，这将比自己实现的左子树右孩子的写法更安全可靠。（作为代价，可能会牺牲一点点的效率。）

例题（POJ 2342 Anniversary party）

很简单的一个树形 dp 的问题，可以很容易的考虑到每个人都有两种选择，也就是来或者不来。那么接下来的部分就容易很多了，我们可以很容易的得到状态表示，$$dp[i][0]$$ 表示第 $$i$$ 个人不来；$$dp[i][1]$$ 表示第 $$i$$ 个人来。

在状态的转移方面，可以得到，如果第 $$i$$ 个人不来的话，那么他的所有子节点就都有来或不来两种状态，也就是 $$dp[i][0] = max{\textbf{j is son of i}}{dp[j][0], dp[j][1]}$$；同时如果第 $$i$$ 个人来的话，那么他的所有子节点都只能不来，也就是是 $$dp[i][1] = max{\textbf{j is son of i}}{dp[j][0]}$$。

在得到了状态转移和表示之后，代码就很容易得到了。我们在写树形 dp 的时候，一般会写成类似于搜索的样子，只不过会在搜索中加入备忘录，这种写法也叫作记忆化搜索。

代码中还有关于把无根树转换成有根树的方法，任意选择一个节点作为根节点，从这个根节点开始 DFS，要注意因为在建边的时候都是双向边，所以在DFS的过程中，一定要注意判断当前节点是否是我们设置的父节点，防止循环的递归或者可能产生错误的答案。

```
#include <cstdio>
```

```
#include <cstring>
#include <algorithm>
#include <vector>
using namespace std;
const int maxn = 6010;

int dp[maxn][2];
vector<int> g[maxn];
int w[maxn];

int get_dp(int x, int s, int fa) {
    if (dp[x][s] != -1) {
        return dp[x][s];
    }
    dp[x][s] = 0;
    if (s) {
        dp[x][s] = w[x];
        for (size_t i = 0; i < g[x].size(); i++) {
            if (g[x][i] != fa) {
                dp[x][s] += get_dp(g[x][i], 0, x);
            }
        }
    }
    else {
        for (size_t i = 0; i < g[x].size(); i++) {
            if (g[x][i] != fa) {
                dp[x][s] += max(get_dp(g[x][i], 0, x),
get_dp(g[x][i], 1, x));
            }
        }
    }
    return dp[x][s];
}


int main() {
```

```
    int n;
    while (scanf("%d", &n) != EOF) {
        memset(dp, -1, sizeof dp);
        for (int i = 1; i <= n; i++) {
            g[i].clear();
        }

        for (int i = 1; i <= n; i++) {
            scanf("%d", &w[i]);
        }
        int u, v;
        while (scanf("%d%d", &u, &v), u||v) {
            g[u].push_back(v);
            g[v].push_back(u);
        }

        printf("%d\n", max(get_dp(1, 0, -1), get_dp(1, 1,
-1)));
    }
    return 0;
}
```

状态压缩DP

状态压缩DP，状压dp的难点在于状态的表示，状态的表示是否是满足无后效性的，是不是满足最优子结构的。是不是可以很容易的通过位运算的特性去用一个状态得到一个新状态。

准备工作

状压dp中少不了对状态的操作。再加上我们一般都会用二进制去表示状态，所以需要对 C++ 中的位运算部分有些了解。

操作符	意义	常用法
`<<`	左移	--
`>>`	右移	--
`&`	按位与	• 用于判断一位是不是1 • 用于清空一位的状态 • 求两个状态的交集
`|`	按位或	• 将一位置1 • 求两个状态的并集
`^`	按位异或	--
`~`	按位取反	--

- 判断第i为是否为1（这个位是从右数起，从0开始。） `if (x&(1<<i) {}` 或者 `if ((x>>i)&1) {}` 。前面两种写法都是可以的，这里要注意不要忘记了位运算两侧的括号。因为位运算的优先级很低。
- 设置第i位为1 `x |= 1<<i;`
- 设置第i位为0 `x &= ~(1<<i);`
- 切换第i位 `x ^= 1<<i;`

例题 CodeForces 580D. Kefa and Dishes

题目大意

有n种菜，选m种。每道菜有一个权值，有些两个菜按顺序挨在一起会有combo的权值加成。求最大权值。

分析

看到题目就会向二进制状态的方向考虑，理由是 `n` 和 `m` 的数据范围都在20左右，非常小。因为如果用二进制表示状态的话，最大也只能表示20位左右的状态，再多就基本一定会超时了。
在这道题目中，我们的状态表示：$$dp[st][i]$$，代表了当选用 `st` 这样的状态，并且最后一个选择的是第 `i` 个菜的时候的最大权值。保存最后的选择，是为了计算combo值，前面的状态就很容易理解了，用1来表示选择了，用0表示没有选择。状态转移方程上，写了这么多的dp题，应该很容易得到，这部分可以去看代码。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 20;
typedef long long LL;

int a[maxn];
int comb[maxn][maxn];
LL dp[(1<<18)+10][maxn];
LL ans = 0;
int n, m, k;

int Cnt(int st) {
    int res = 0;
    for (int i = 0; i < n; i++) {
        if (st&(1<<i)) {
            res++;
        }
    }
```

```
    return res;
}

int main() {
    memset(comb, 0, sizeof comb);
    scanf("%d%d%d", &n, &m, &k);
    for (int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    for (int i = 0; i < k; i++) {
        int x, y, c;
        scanf("%d%d%d", &x, &y, &c);
        x--;
        y--;
        comb[x][y] = c;
    }
    int end = (1<<n);
    memset(dp, 0, sizeof dp);
    for (int st = 0; st < end; st++) {
        for (int i = 0; i < n; i++) {
            if (st&(1<<i)) {
                bool has = false;
                for (int j = 0; j < n; j++) {
                    if (j != i && (st&(1<<j))) {
                        has = true;
                        dp[st][i] = max(dp[st][i],
dp[st^(1<<i)][j] + a[i] + comb[j][i]);
                    }
                }
                if (!has) {
                    dp[st][i] = a[i];
                }
            }
            if (Cnt(st) == m) {
                ans = max(ans, dp[st][i]);
            }
```

```
            }
        }

        cout << ans << endl;
        return 0;
}
```

动态规划的优化

前言

动态规划在解决问题的时候，利用的是保存中间状态来减少重复的计算次数的方法来减少时间消耗的。可是在很多问题中，动态规划的时间效率，或者空间复杂度都不能满足我们的要求（超出我们的限制），这时候就要考虑对算法进行时间或者空间上的优化。

时间复杂度优化

动态规划问题总的来说还是对枚举的优化，即保存中间状态来减少计算次数的方法。那么就可以大概的认为，一个动态规划问题的时间复杂度应该是：`时间复杂度 = 状态总数 * 状态转移总数 * 状态转移时间`。我们只要减少任意一个部分，都可以达到加速动态规划的方式。

改进状态表示

即通过改变状态的表示方式来减少状态总数的方法。这种方法通常是一开始想出来的状态表示方法并不够好，还有我们没有发现的关系或者条件。在这种时间，我们可以对状态的表示方法进行优化，用一种更优的表示方法。这样在转义数量和转移时间不变的情况下，时间复杂度就已经被减小了。

例题

题目大意

把 $$n$$ 个数字，分成 $$m$$ 组。每组的和不能大于 $$t$$。分组必须是有序的，并且保证所有的数字都是小于 $$t$$ 的。求最多能把多少数字划入分组里。

最初的分析

这里我们用 `w[i]` 来表示第 `i` 个数的值，用 `dp` 来表示我们的结果数组。那么我们可以显然的得到一个状态表示：$$dp[i][j][k]=x$$，来表示前 $$i$$ 个数字，分成 $$j$$ 组，还多了 $$k$$ 的时候，最多有多少个数字被划入了分组中。
可以显然的得到，这里整个问题的最优解应该是 $$dp[n][m][0]$$。通过状态的表示，我们可以得到状态转移方程：

$$ dp[i][j][k] \left{ \begin{array}{ll} max(dp[i-1][j][k-w[i]],dp[i-1][j][k]) & k\leq w[i],i \geq 1\ max(dp[i-1][j-1][t-w[i]],dp[i-1][j][k]) & k < w[i], i \geq 1\ 0 & 0\leq k < t \end{array} \right.

$$ 这样时间复杂度应该是 $$O(nmt)$$ 的。因为总状态数是 $$O(nmt)$$，每次状态转移的状态数为 $$O(1)$$，总时间为 $$O(nmt)$$。

改进

改进后的状态表示：$$dp[i][j]=(x,y)$$，表示在前 $$i$$ 中选 $$j$$ 个所需要的最少的分组为 $$x$$ 另外还需要 $$y$$。
通过状态的表示，我们可以得到状态转移方程：

$$ dp[i][j] = min(dp[i-1][j], g[i-1][j-1]+w[i]) $$ 其中，$$ (x, y) + w[i] = (x',y') $$计算方法为：

$$ \begin{array}{ll} x'=x,y'=y+w[i]&w[i]\leq t-y\ x'=x+1,y'=w[i]&w[i]>t-y \end{array}

$$ 时间复杂度从$$O(nmt)$$，降到了$$O(n^2)$$。这里空间复杂度也同时降低了。

决策单调性·四边形不等式

例题1

在一个操场上摆放着一排$$n(n\leq20)$$堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的2堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。求出将 $$n$$ 堆石子合并成一堆的最小得分和最大得分以及相应的合并方案。

最初的分析

这里最大得分和最小得分的思路是一样的。这里只取最小得分来说明问题。

令每堆石子的个数为：$$d[1\ldots n]$$。我们可以得到最基础的状态表示：$$m[i][j]$$ 来表示合并第 $$i$$ 个到第 $$j$$ 个石子堆所获得的最小得分。得到了状态转移方程和边界条件：

$$ \begin{array}{ll} m[i][j]=0 & i=j m[i][j]=min*{i<k\leq j}{m[i][k-1]+m[k][j]+\sum*{l=i}^{j}d[l]} \end{array}

$$ 并且把最优解保存在 $$s[i][j]$$ 中。（这里的 $$\sum_{l=i}^{j}d[l]$$ 部分可以用 $$O(n^2)$$ 的时间复杂度预处理，这样直接引用就好了。）
总的时间复杂度：$$O(n^3)$$

改进

- 定理**1** 当函数 $$w(i,j)$$ 满足 $$w(i,j)+w(i',j')\leq w(i',j)+w(i,j'),i\leq i'\leq j\leq j'$$ 时，则称函数 $$w$$ 满足四边形不等式。
- 定理**2** 当函数 $$w(i,j)$$ 满足 $$w(i',j)\leq w(i,j'),i\leq i'\leq j\leq j'$$ 时，则称函数 $$w$$ 关于区间包含关系单。

在这个问题中。另 $$w(i,j)=\sum_{l=i}^{j}d[l]$$，可以显然得到 $$w(i,j)$$ 是满足四边形不等式的，同时也满足区间包含关系单调。根据 $$m$$ 的递推式：

$$ \begin{equation} m[i][j]=min_{j<k\leq j}{m[i][k-1]+m[k][j]}+w(i,j) \end{equation}

$$ 对于满足四边形不等式的函数 $$w$$ 这里我们要证明，会使 $$m$$ 也满足四边形不等式。

这里当 $$i=i'$$ 或者 $$j=j'$$ 的时候，不等式显然成立。当 $$ij$$。只讨论前一种情况，因为后一种和前一种是镜像的。

$$ \begin{array}{rll} m[i][j]+m[j][j'] & = & w(i,j)+m[i][k-1]+m[k][j]+m[j][j']\ & \leq & w(i,j')+m[i][k-1]+m[k][j]+m[j][j']\ & \leq & w(i,j')+m[i][k-1]+m[k][j']\ & = & m[i][j'] \end{array}

$$ 当 $$iy$$。还是只讨论前一种，后一种是相似的。

$$ \begin{array}{rll} m[i][j]+m[i'][j'] & = & w(i,j)+m[i][z-1]+m[z][j]+w(i',j')+m[i'][y-1]+m[y][j]\ & \leq & w(i,j')+w(i',j)+m[i'][y-1]+m[i][z-1]+m[z][j]+m[y][j']\ & \leq & w(i,j')+w(i',j)+m[i'][y-1]+m[i][z-1]+m[y][j]+m[z][j']\ & = & m[i][j']+m[i'][j] \end{array}

$$ 综上，$$m[i][j]$$ 满足四边形不等式。其决策满足单调性，即：记 $$s[i][j]$$ 为 $$m[i][j]$$ 取最优值时的决策，有：

$$ \begin{equation} s[i][j]\leq s[i][j+1]\leq s[i+1][j+1], i \leq j \end{equation}

$$

这里要证明一下上面的那个式子成立。 令：$$m*{k}[i][j]=m[i][k-1]+m[k][j]+w(i,j)$$，要证明 *$$s[i][j]\leq s[i][j+1]$$*，也就是证明对于所有的 *$$i<k\leq k'\leq j$$* 且 *$$m*{k'}[i][j] \leq m_{k}[i][j]$$，有：

$$ m*{k'}[i][j+1]\leq m*{k}[i][j+1]

$$ 这里有一个更强的不等式，

$$ m*{k}[i][j]-m*{k'}[i][j]\leq m*{k}[i][j+1]-m*{k'}[i][j+1]$$ 即：$$ m*{k}[i][j]+m*{k'}[i][j+1]\leq m*{k}[i][j+1]+m*{k'}[i][j]$$ 展开得到：$$ m[k][j]+m[k'][j+1]\leq m[k'][j]+m[k][j+1]$$ 这恰好是 $$k\leq k'\leq j\leq j+1$$ 时的四边形不等式的形式。

所以，当$$w$$满足四边形不等式的时候，$$s[i][j]$$的取值满足单调性。

这样我们就可以改写我们的状态转移方程了，因为我们的决策的范围已经被缩小了。

$$ \begin{array}{ll} m[i][j]=0 & i=j\ m[i][j]=min_{s[i][j-1]\leq k\leq s[i+1][j]}{m[i][k-1]+m[k][j]}+w(i,j) \end{array}

$$ 时间复杂度：$$O(n^2)$$。

例题2 Post Office

这题简单的DP是可以通过的。不过如果把数据范围扩大了，就可以用上面说的四边形不等式优化去优化。略去证明，这道题直接给出伪代码：

```
1  for i ← 1 to n
2        do dp[1,i] ← w[1,i]
3           s[1,i] ← 0
4  for i ← 2 to n
5        do s[i,n+1] ← n
6           for j ← n downto 1
7                 do for k ← s[i−1,j] to s[i][j+1]
8                          do if dp[i,k]+dp[k+1][j]
9                                then dp[i,j] ← dp[i,k]+dp[k+1][j]
10                                    reg ← k
11                   dp[i,j] ← dp[i,j]+w[i,j]
```

决策单调性·斜率优化

在说斜率优化之前，要先说一下另一个优化方式。当我们发现一个dp的状态转移方程形如这样的形式的时候，$$ dp(i) = f(j) + g(i)$$也就是说，一个 $$i$$ 的结果可以化成一个于 $$i$$ 无关的函数的和另一个只与 $$i$$ 有关的形式的时候。我们可以对前面的和 $$i$$ 无关的部分放在一个优先队列里，这样可以把本来 $$O(n)$$ 的查找代价，变成 $$\log n$$ 的。

但是这终归是一个非常理想的方程啊，实际问题中这种良心满满的题目基本是没有的。所以一种把条件放的稍微弱一点的优化方式，就是斜率优化。也是通过结果和决策的关系，发现决策有事满足一些神奇的数量关系的。

例题 Print Article

状态表示：$$dp[i]$$来表示，从第一个到第 $$i$$ 个输出的最少花费。
很容易得到状态转移方程：

$$ \begin{equation} dp[i] = min_{0 \leq j < i}{dp[j]+M+sum[j+1 \ldots i]^2} \end{equation}

$$ 这个的时间复杂度可以很显然的计算出来，是 $$O(n^2)$$ 的。显然，我们是不能接受这种时间复杂度的，因为数据范围非常大。

改进

这里，我们先假设，上式中的 $$j$$ 取值为 $$k_1$$ 的时候，比取 $$k_2$$ 的时候更优。也就是：

$$ dp[k_1]+sum[k_1+1 \ldots i]^2 < dp[k_2] + sum[k_2+1 \dots i]^2$$ 其中，这里我们可以预处理得到一个 $$sum$$ 数组。那么：$$sum[k_1+1 \ldots i] = sum[i]-sum[k_1]$$，同理对于 $$k_2$$ 的部分。移项得到：

$$ \begin{equation} (dp[k_1]+sum[k_1]^2)-(dp[k_2]+sum[k_2]^2) < 2*sum[i]* (sum[k_1]-sum[k_2]) \end{equation}

```
$$
```

其中所有的项都是我们已经计算好得到了的。我们 $$f(i)=dp[i]+sum[i]^2$$，$$g(i)=sum[i]$$。上可以化成：

$$ \begin{equation} \frac{f(k_1)-f(k_2)}{g(k_1)-g(k_2)} < 2 \times sum[i] \end{equation}

$$ 如果以 $$f(i)$$ 为Y轴，以 $$g(i)$$ 为X轴，那么上面个式子就是两点之间的斜率。并把这个值表示为：$$\varphi(k_1,k_2)$$。

然后我们来说明他们之间的单调性关系。即，若有 $$\varphi(i,j)<\varphi(j,k)$$，那么 $$j$$ 这个点一定是不可能是最优解的。

证明如下：
当 $$\varphi(i,j)<2sum[x]$$，也就是说明 $$i$$="" 比="" $$j$$="" 更优，排除="" 点。="" 当="" $$\varphi(i,j)\geq="" 2sum[x]$$，说明="" 更优，但是此时="" $$\varphi(j,k)=""> \varphi(i,j) \geq 2sum[x]$$，也就是 $$k$$ 比 $$j$$ 更优，也就是 $$j$$ 是没用的。
也就是说，对于一个 $$k\varphi(j,k),k<j<i$$ ，也就是说整个图形呈现一个上凸的形状。如何维护图形的上凸可以用凸包的算法去解决。
唯一的就是在求解结果的时候，要从队列的头部一直找到第一个满足 $$\varphi(k_1,k_2)<2sum[i]$$ 的值。

伪代码：

```
1  Q ← Queue
2  for i ← 1 to n
3          do while Q.len > 1 and φ(Q.top(0), Q.top(1)) ≥ 2sum[i]
4                  do Q.pop()
5             dp[i] = dp[Q.top(0)] + M + (sum[i] − sum[Q.top(0)])^2
6             while Q.len > 1 and
7             φ(i, Q.bottom(0)) ≤ varphi(Q.bottom(1), Q.bottom(2))
8                  do Q.popback()
```

数据结构

---

author：郭昊

数据结构

并查集

并查集一般用于对动态连通性的判断，主要应用于判断两个元素是否在同一个集合，两个点是否连通，变量名等同性以及间接好友的判断。同时并查集经常作为其他模板的一部分实现某些功能。

并查集常用于的题型为判断某两个元素是否属于同一个集合，判断图是否连通或是否有环，或配合其他算法如最小生成树Kruskal，与DP共同使用等。

- 并查集模板

```
int fa[N];
void init(int n) {  // 不要忘记哦！！！
    for (int i = 0; i <= n; i++)
        fa[i] = i;
    }
}
void unin(int u, int v) {
    int fau = find(u);
    int fav = find(v);
    if (fau == fav) return;
    fa[fav] = fau;
}
int find(int u) {
    if (fa[u] != u) {
        fa[u] = find(fa[u]);
    }
    return fa[u];
}
```

- 更短一些的并查集模板

```
  int fa[N];

  void init(int n) {
      for (int i = 0; i <= n; fa[i] = i++);
  }

  void unin(int u, int v) {
      fa[find(v)] = find(u);
  }

  int find(int u) {
      return fa[u] == u ? fa[u] : fa[u] =
find(fa[u]);
  }
```

例题

- poj 1611 The Suspects

  有很多组学生，在同一个组的学生经常会接触，也会有新的同学的加入。但是SARS是很容易传染的，只要在改组有一位同学感染SARS，那么该组的所有同学都被认为得了SARS。现在的任务是计算出有多少位学生感染SARS了。假定编号为0的同学是得了SARS的。

  采用num[]存储该集合中元素个数，并在集合合并时更新num[]即可。然后找出0所在的集合的根节点x，因此，num[x]就是answer了。

代码如下：

```
#include <iostream>

using namespace std;

int n, m, k, t, f, p[30001], rank[30001], a, b;

int find(int x) {
    if (x == p[x]) return x;
```

```
    else return p[x] = find(p[x]);
}

void un(int x, int y) {
    a = find(x);
    b = find(y);
    if (a == b)
        return;
    if (rank[a] > rank[b])
        p[b] = a;
    else {
        p[a] = b;
        if (rank[a] == rank[b])
            rank[b]++;
    }
}

int main() {
    int i, sum;
    while (cin >> m >> n && (m || n)) {
        for (i = 0; i < m; i++) {
            p[i] = i;
            rank[i] = 0;
        }
        for (i = 0; i < n; i++) {
            cin >> k;
            if (k >= 1)
                cin >> f;
            for (int j = 1; j < k; j++) {
                cin >> t;
                un(f, t);
            }
        }
        sum = 1;
        for (i = 1; i < m; i++) {
            if (find(i) == find(0))
```

```
                sum++;
            }
            cout << sum << endl;
        }
        return 0;
}
```

- poj 1182 食物链(并查集经典题目)

    题目告诉有3种动物，互相吃与被吃，现在告诉你m句话，其中有真有假，叫你判断假的个数(如果前面没有与当前话冲突的，即认为其为真话)

    带权并查集和普通并查集最大的区别在于带权并查集合并的是可以推算关系的点的集合（可以通过集合中的一个已知值推算这个集合中其他元素的值）。而一般并查集合并的意图在于这个元素属于这个集合。带权并查集相当于把“属于”的定义拓展了一下，拓展为有关系的集合。

    本题用rank[x]记录x与x的最远的祖先的关系。 这里定义rank[x]=0表示x与x的祖先是同类。rank[x]==1表示x吃x的祖先。rank[x]==2表示x的祖先吃x；这样定义后就与题目中输入数据的D联系起来，(D-1)就可以表示x与y的关系。这样就可以用向量的形式去推关系的公式了。我们用f(x,father[x])表示rank[x]的值；

    代码如下:

```
#include<stdio.h>

int fa[50005]={0};
int rank[50005]={0};
int n;

void initial()
{
    for(int i=1;i<=n;i++)
    {
        fa[i]=i; rank[i]=0;
    }
}
```

```
int getfather(int x)
{
    if(x==fa[x])  return x;
    int oldfa = fa[x];
    fa[x]=getfather(fa[x]);
    rank[x]=(rank[x]+rank[oldfa])%3;  //用向量的形式很快就可以看出来
    return fa[x];
}

void unionset(int r,int x,int y)
{
    int fx,fy;
    fx=getfather(x); fy=getfather(y);
    if(fx==fy) return;
    fa[fx]=fy;
    rank[fx]=(rank[y]+r-rank[x]+3)%3;   // 这里同样可以用向量来推公式。另外需要注意的是，这里只更新了fx的rank值，而fx的儿子的rank值都没有更新会不会有问题。其实不碍事，由于我们每次输入一组数据我们都对x和y进行了getfather的操作（x>n || y>n ……)的除外。在执行getfather的操作时，在回溯的过程中就会把fx的儿子的rank值都更新了。
    return ;
}

int istrue(int d,int x,int y)
{
    int fx,fy,r;
    if(x>n || y>n || ((x==y)&&(d==2)) )
        return 0;
    fx=getfather(x); fy=getfather(y);
    if(fx!=fy)  return 1;
    else
    {
        if(rank[x]==((d-1)+rank[y])%3) return 1;  // 这个公式可以用向量来推：如果（ f(x,y) + f(y,father[y]））% 3 ==
```

```
f(x,father[x]) 则是正确的，否则是错的。这个形式可以用向量来表示，就是判断这个向量加法对不对   x--->y + y---> fx(fy) 是否等于 x--->fx(fy)
        else  return 0;
    }
}

int main()
{
    int k,i,x,y,d; int ans=0;
    scanf("%d%d",&n,&k);
    initial();
    for(i=1;i<=k;i++)
    {
        scanf("%d%d%d",&d,&x,&y);
        if( !istrue(d,x,y) )
            ans++;
        else
            unionset(d-1,x,y);
    }
    printf("%d\n",ans);
    return 0;
}
```

树状数组

树状数组(Binary Indexed Tree(BIT), Fenwick Tree)是一个查询和修改复杂度都为log(n)的数据结构。主要用于查询任意两位之间的所有元素之和，但是每次只能修改一个元素的值；经过简单修改可以在log(n)的复杂度下进行范围修改，但是这时只能查询其中一个元素的值。

树状数组十分容易实现，代码量小，时间复杂度低，并且经过数学处理后也可以实现成段更新。线段树也可以做到和树状数组一样的效果，但是代码要复杂得多。不过要注意，一般情况下树状数组能解决的问题线段树都能解决，反之有些线段树能解决的问题树状数组却不行。

代码如下:

```
int lowbit(int x)
{
    return x & (-x);
}
void modify(int x,int add)//一维
{
    while(x<=MAXN)
    {
        a[x]+=add;
        x+=lowbit(x);
    }
}
int get_sum(int x)
{
    int ret=0;
    while(x!=0)
    {
        ret+=a[x];
        x-=lowbit(x);
    }
    return ret;
```

```
}
void modify(int x,int y,int data)//二维
{
    for(int i=x;i<MAXN;i+=lowbit(i))
        for(int j=y;j<MAXN;j+=lowbit(j))
            a[i][j]+=data;
}
int get_sum(int x,int y)
{
    int res=0;
    for(int i=x;i>0;i-=lowbit(i))
        for(int j=y;j>0;j-=lowbit(j))
            res+=a[i][j];
    return res;
}
```

例题

- poj 2299 Ultra-QuickSort

  题意为给定一个序列，求该序列中的逆序数的对数。

  树状数组求解的思路：开一个能大小为这些数的最大值的树状数组，并全部置0。从头到尾读入这些数，每读入一个数就更新树状数组，查看它前面比它小的已出现过的有多少个数sum，然后用当前位置减去该sum，就可以得到当前数导致的逆序对数了。把所有的加起来就是总的逆序对数。

  题目中的数都是独一无二的，这些数最大值不超过999999999，但n最大只是500000。如果采用上面的思想，必然会导致空间的巨大浪费，而且由于内存的限制，我们也不可能开辟这么大的数组。因此可以采用离散化的方式，把原始的数映射为1-n一共n个数，这样就只需要500000个int类型的空间。

代码如下：

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

```
const int N = 500005;

struct Node
{
    int val;
    int pos;
};

Node node[N];
int c[N], reflect[N], n;

bool cmp(const Node& a, const Node& b)
{
    return a.val < b.val;
}

int lowbit(int x)
{
    return x & (-x);
}

void update(int x)
{
    while (x <= n)
    {
        c[x] += 1;
        x += lowbit(x);
    }
}

int getsum(int x)
{
    int sum = 0;
    while (x > 0)
    {
```

```
        sum += c[x];
        x -= lowbit(x);
    }
    return sum;
}

int main()
{
    while (scanf("%d", &n) != EOF && n)
    {
        for (int i = 1; i <= n; ++i)
        {
            scanf("%d", &node[i].val);
            node[i].pos = i;
        }
        sort(node + 1, node + n + 1, cmp);    //排序
        for (int i = 1; i <= n; ++i) reflect[node[i].pos]
= i;    //离散化
        for (int i = 1; i <= n; ++i) c[i] = 0;    //初始化
树状数组
        long long ans = 0;
        for (int i = 1; i <= n; ++i)
        {
            update(reflect[i]);
            ans += i - getsum(reflect[i]);
        }
        printf("%lld\n", ans);
    }
    return 0;
}
```

- poj 3468 A Simple Problem with Integers

    题意：给你一个数列，每次询问一个区间的和，或者每次将一个区间的所有元素都加上一个数.

    树状数组也可以实现区间更新，不过相比要比线段树区间更新难理解一

些，具体过程如下：

首先，看更新操作update(s,t,d)把区间A[s]...A[t]都增加d，我们引入一个数组delta[i]，表示A[i]...A[n]的共同增量，n是数组的大小。那么update操作可以转化为：令delta[s] = delta[s]+d，表示将A[s]...A[n]同时增加d，但这样A[t+1]...A[n]就多加了d，所以再令delta[t+1] = delta[t+1]-d，表示将A[t+1]...A[n]同时减d。

然后来看查询操作query(s, t)，求A[s]...A[t]的区间和，转化为求前缀和，设sum[i] = A[1]+...+A[i]，则 A[s]+...+A[t] = sum[t] - sum[s-1]， 那么前缀和sum[x]又如何求呢？它由两部分组成，一是数组的原始和，二是该区间内的累计增量和,把数组A的原始值保存在数组org中，并且delta[i]对sum[x]的贡献值为delta[i]*(x+1-i)，那么sum[x] = org[1]+...+org[x] + delta[1]*x + delta[2]*(x-1) + delta[3]*(x-2)+...+delta[x]*1 = org[1]+...+org[x] + segma(delta[i]*(x+1-i)) = segma(org[i]) + (x+1)*segma(delta[i])-segma(delta[i]*i)，1<=i<=x=segma(org[i]-delta[i]*i)+(x+1)*delta[i], i<=1<=x.

这里就可以转化为两个个数组,这其实就是三个数组org[i],delta[i]和delta[i]*i的前缀和，org[i]的前缀和保持不变，事先就可以求出来，delta[i]和delta[i]*i的前缀和是不断变化的，可以用两个树状数组来维护。

代码如下：

```
#include<cstdio>
#include<cstring>
using namespace std;

long long int n;
long long bit0[200000],bit1[200000];

long long sum(long long *b,int i)
{
    long long s=0;
    while(i>0)
    {
        s+=b[i];
        i-= i &(-i);
    }
    return s;
```

```
}

void add(long long *b,long long  int i,long long int v)
{
    while(i<=n)
    {
        b[i]+=v;
        i+=i&(-i);
    }
}


int main()
{
    long long int i,j,k,a,b,c,m;
    long long res;
    char ch[100];
    while(scanf("%lld%lld",&n,&m)!=EOF)
    {
        memset(bit0,0,sizeof(bit0));
        memset(bit1,0,sizeof(bit1));
        for(i=1;i<=n;i++)
        {
            scanf("%lld",&a);
            add(bit0,i,a);
        }
        for(k=0;k<m;k++)
        {
            scanf("%s",ch);
            if(ch[0]=='C')
            {
                scanf("%lld%lld%lld",&a,&b,&c);
                add(bit0,a,-c*(a-1));
                add(bit1,a,c);
                add(bit0,b+1,b*c);
                add(bit1,b+1,-c);
```

```
            }
            else
            {
                res=0;
                scanf("%lld%lld",&a,&b);
                res+=sum(bit0,b)+sum(bit1,b)*b;
                res-=sum(bit0,a-1)+sum(bit1,a-1)*(a-1);
                printf("%lld\n",res);
            }
        }
    }
    return 0;
}
```

线段树

线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。

对于线段树中的每一个非叶子节点[a,b]，它的左儿子表示的区间为[a,(a+b)/2]，右儿子表示的区间为[(a+b)/2+1,b]。因此线段树是平衡二叉树，最后的子节点数目为N，即整个线段区间的长度。

使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为O(logN）。而未优化的空间复杂度为2N，因此有时需要离散化让空间压缩。

线段树有很多模板，而且基本上每道题都是稍稍改动或者根本不需改动就可以直接使用线段树的模板，这里提供几个相对简洁的模板：

```
//单点替换、单点增减、区间求和、区间最值
#include <cstdio>
#include <algorithm>
using namespace std;

#define lson l , m , rt << 1
#define rson m + 1 , r , rt << 1 | 1
const int maxn = 222222;

int MAX[maxn<<2];
int MIN[maxn<<2];
int SUM[maxn<<2];
int max(int a,int b){if(a>b)return a;else return b;}
int min(int a,int b){if(a<b)return a;else return b;}

void PushUP(int rt)
{
  MAX[rt] = max(MAX[rt<<1] , MAX[rt<<1|1]);
  MIN[rt] = min(MIN[rt<<1] , MIN[rt<<1|1]);
  SUM[rt] = SUM[rt<<1] + SUM[rt<<1|1];
}

void build(int l,int r,int rt) {
```

```
  if (l == r)
    {
    scanf("%d",&MAX[rt]);
    MIN[rt] = MAX[rt];
    SUM[rt] = MAX[rt];
    //printf("mi = %d\n",MIN[rt]);
  //    printf("ma = %d\n",MAX[rt]);
    return ;
  }
  int m = (l + r) >> 1;
  build(lson);
  build(rson);
  PushUP(rt);
}

void update(int p,int tihuan,int l,int r,int rt)
{
  if (l == r) {
    MAX[rt] = tihuan;
    MIN[rt] = tihuan;
    SUM[rt] = tihuan;
    return ;
  }
  int m = (l + r) >> 1;
  if (p <= m) update(p , tihuan ,lson);
  else update(p , tihuan , rson);
  PushUP(rt);
}

void update1(int p,int add,int l,int r,int rt)
{
  if (l == r) {
    SUM[rt] = SUM[rt] + add;
    return ;
  }
  int m = (l + r) >> 1;
```

```
  if (p <= m) update1(p , add ,lson);
  else update1(p , add , rson);
  PushUP(rt);
}

int query(int L,int R,int l,int r,int rt)
{
  if (L <= l && r <= R)
  {
    return MAX[rt];
  }
  int m = (l + r) >> 1;
  int ret = -1;
  if (L <= m) ret = max(ret , query(L , R , lson));
  if (R > m)  ret =  max(ret , query(L , R , rson));
  return ret;
}

int query1(int L,int R,int l,int r,int rt)
{
  if (L <= l && r <= R)
  {
    return MIN[rt];
  }
  int m = (l + r) >> 1;
  int ret = 99999;
  if (L <= m) ret = min(ret , query1(L , R , lson));
  if (R > m)  ret =  min(ret , query1(L , R , rson));
  return ret;
}

int queryhe(int L,int R,int l,int r,int rt)
{
  if (L <= l && r <= R)
  {
    return SUM[rt];
```

```
    }
    int m = (l + r) >> 1;
    int ret = 0;
    if (L <= m) ret += queryhe(L , R , lson);
    if (R > m)  ret +=  queryhe(L , R , rson);
    return ret;
}

int main()
{
    int n , m;
    while (~scanf("%d%d",&n,&m))
    {
        build(1 , n , 1);
        while (m --) {
            char op[2];
            int a , b;
            scanf("%s%d%d",op,&a,&b);
            if (op[0] == 'Q') //区间求最大
            {
                /* for(int i = 1;i<=10;i++)
                 printf("%d ",MAX[i]);
                puts("");*/
                printf("%d\n",query(a , b , 1 , n , 1));
            }
            else if(op[0]=='U') //单点替换
                update(a , b , 1 , n , 1);
            else if(op[0]=='M')//区间求最小
            {
                /*for(int i = 1;i<=10;i++)
                    printf("%d ",MIN[i]);
                puts("");*/
                printf("%d\n",query1(a , b , 1 , n , 1));
            }
            else if(op[0]=='H')//区间求和
            {
```

```
        printf("%d\n",queryhe(a , b , 1 , n , 1));
      }
      else if(op[0]=='S')//单点增加
      {
        scanf("%d%d",&a,&b);
        update1(a , b , 1 , n , 1);
      }
       else if(op[0]=='E')//单点减少
      {
        scanf("%d%d",&a,&b);
        update1(a , -b , 1 , n , 1);
      }
    }
  }
  return 0;
}
```

```
//区间替换
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <algorithm>

#define max(a,b) (a>b)?a:b
#define min(a,b) (a>b)?b:a
#define lson l , m , rt << 1
#define rson m + 1 , r , rt << 1 | 1
#define LL long long
const int maxn = 100100;
using namespace std;

int lazy[maxn<<2];
int sum[maxn<<2];
```

```
void PushUp(int rt)//由左孩子、右孩子向上更新父节点
{
  sum[rt] = sum[rt<<1] + sum[rt<<1|1];
}

void PushDown(int rt,int m) //向下更新
{
  if (lazy[rt]) //懒惰标记
  {
    lazy[rt<<1] = lazy[rt<<1|1] = lazy[rt];
    sum[rt<<1] = (m - (m >> 1)) * lazy[rt];
    sum[rt<<1|1] = ((m >> 1)) * lazy[rt];
    lazy[rt] = 0;
  }
}

void build(int l,int r,int rt)//建树
{
  lazy[rt] = 0;

  if (l== r)
  {
    scanf("%d",&sum[rt]);
    return ;
  }
  int m = (l + r) >> 1;
  build(lson);
  build(rson);
  PushUp(rt);
}

void update(int L,int R,int c,int l,int r,int rt)//更新
{
  //if(L>l||R>r) return;
  if (L <= l && r <= R)
  {
```

```
    lazy[rt] = c;
    sum[rt] = c * (r - l + 1);
    //printf("%d %d %d %d %d\n", rt, sum[rt], c, l, r);
    return ;
  }
  PushDown(rt , r - l + 1);
  int m = (l + r) >> 1;
  if (L <= m) update(L , R , c , lson);
  if (R > m) update(L , R , c , rson);
  PushUp(rt);
}

LL query(int L,int R,int l,int r,int rt)
{
  if (L <= l && r <= R)
  {
    //printf("%d\n", sum[rt]);
    return sum[rt];
  }
  PushDown(rt , r - l + 1);
  int m = (l + r) >> 1;
  LL ret = 0;
  if (L <= m) ret += query(L , R , lson);
  if (m < R) ret += query(L , R , rson);
  return ret;
}

int main()
{
  int  n , m;
  char str[5];

  while(scanf("%d%d",&n,&m))
  {
    build(1 , n , 1);
    while (m--)
```

```
    {
      scanf("%s",str);
      int a , b , c;
      if(str[0]=='T')
      {
        scanf("%d%d%d",&a,&b,&c);
        update(a , b , c , 1 , n , 1);
      }
      else if(str[0]=='Q')
      {
        scanf("%d%d",&a,&b);
        cout<<query(a,b,1,n,1)<<endl;
      }
    }
  }

  return 0;
}
```

```
//区间增减
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <algorithm>

#define max(a,b) (a>b)?a:b
#define min(a,b) (a>b)?b:a
#define lson l , m , rt << 1
#define rson m + 1 , r , rt << 1 | 1
#define LL __int64
const int maxn = 100100;
using namespace std;

LL lazy[maxn<<2];
```

```
LL sum[maxn<<2];

void putup(int rt)
{
  sum[rt] = sum[rt<<1] + sum[rt<<1|1];
}

void putdown(int rt,int m)
{
  if (lazy[rt])
  {
    lazy[rt<<1] += lazy[rt];
    lazy[rt<<1|1] += lazy[rt];
    sum[rt<<1] += lazy[rt] * (m - (m >> 1));
    sum[rt<<1|1] += lazy[rt] * (m >> 1);
    lazy[rt] = 0;
  }
}

void build(int l,int r,int rt) {
  lazy[rt] = 0;
  if (l == r)
  {
    scanf("%I64d",&sum[rt]);
    return ;
  }
  int m = (l + r) >> 1;
  build(lson);
  build(rson);
  putup(rt);
}

void update(int L,int R,int c,int l,int r,int rt)
{
  if (L <= l && r <= R)
  {
```

```
    lazy[rt] += c;
    sum[rt] += (LL)c * (r - l + 1);
    return ;
  }
  putdown(rt , r - l + 1);
  int m = (l + r) >> 1;
  if (L <= m) update(L , R , c , lson);
  if (m < R) update(L , R , c , rson);
  putup(rt);
}

LL query(int L,int R,int l,int r,int rt)
{
  if (L <= l && r <= R)
  {
    return sum[rt];
  }
  putdown(rt , r - l + 1);
  int m = (l + r) >> 1;
  LL ret = 0;
  if (L <= m) ret += query(L , R , lson);
  if (m < R) ret += query(L , R , rson);
  return ret;
}

int main()
{
  int n , m;int a , b , c;
  char str[5];
  scanf("%d%d",&n,&m);
  build(1 , n , 1);
  while (m--)
  {
    scanf("%s",str);
    if (str[0] == 'Q')
    {
```

```
        scanf("%d%d",&a,&b);
        printf("%I64d\n",query(a , b , 1 , n , 1));
      }
      else if(str[0]=='C')
      {
        scanf("%d%d%d",&a,&b,&c);
        update(a , b , c , 1 , n , 1);
      }
    }
    return 0;
}
```

最简单的应用就是记录线段是否被覆盖，并随时查询当前被覆盖线段的总长度。那么此时可以在结点结构中加入一个变量int count；代表当前结点代表的子树中被覆盖的线段长度和。这样就要在插入（删除）当中维护这个count值，于是当前的覆盖总值就是根节点的count值了。

另外也可以将count换成bool cover；支持查找一个结点或线段是否被覆盖。

实际上，通过在结点上记录不同的数据，线段树还可以完成很多不同的任务。例如，如果每次插入操作是在一条线段上每个位置均加k，而查询操作是计算一条线段上的总和，那么在结点上需要记录的值为sum。

这里会遇到一个问题：为了使所有sum值都保持正确，每一次插入操作可能要更新O(N）个sum值，从而使时间复杂度退化为O(N）。

解决方案是Lazy思想：对整个结点进行的操作，先在结点上做标记，而并非真正执行，直到根据查询操作的需要分成两部分。

根据Lazy思想，我们可以在不代表原线段的结点上增加一个值toadd，即为对这个结点，留待以后执行的插入操作k值的总和。对整个结点插入时，只更新sum和toadd值而不向下进行，这样时间复杂度可证明为O(logN）。

对一个toadd值为0的结点整个进行查询时，直接返回存储在其中的sum值；而若对toadd不为0的一部分进行查询，则要更新其左右子结点的sum值，然后把toadd值传递下去，再对这个查询本身，左右子结点分别递归下去。时间复杂度也是O(nlogN）。

例题

- hdu 1754 I Hate It

  题意：给出一个学生成绩的序列，有两个操作：1.修改一个学生的成绩，2.查询学生A到学生B之间所有学生的最高分。 典型的单点更新求区间最值问题，直接用线段树模板即可。

  代码如下：

```
#include<stdio.h>
#include<string.h>
#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1
#define max 200010

int N,M;
int sum[max<<2];

int Max(int a,int b)
{
    return a>b?a:b;
}

void PushUp(int rt)
{
    sum[rt]=Max(sum[rt<<1],sum[rt<<1|1]);
}

void build(int l,int r,int rt)
{
    if(l==r)
    {
        scanf("%d",&sum[rt]);
        return ;
    }
    int m=(l+r)>>1;
    build(lson);
    build(rson);
```

```
    PushUp(rt);
}

int query(int L,int R,int l,int r,int rt)
{

    if(l>=L&&r<=R)
    {
        return sum[rt];
    }
    int maxn=0;
    int m=(l+r)>>1;
    if(L<=m)
       maxn=Max(maxn,query(L,R,lson));
    if(R>m)
       maxn=Max(maxn,query(L,R,rson));
    return maxn;
}

void update(int L,int num,int l,int r,int rt)
{
    if(r==l)
    {
      sum[rt]=num;
      return ;
    }
    int m=(l+r)>>1;
    if(L<=m)
    update(L,num,lson);
    else
    update(L,num,rson);
    PushUp(rt);
}

int main()
{
```

```
    char op[2];
    int A,B;
    while(scanf("%d%d",&N,&M)!=EOF)
    {
        memset(sum,0,sizeof(sum));
        build(1,N,1);
        for(int i=0;i<M;i++)
        {
            scanf("%s%d%d",op,&A,&B);
            if(op[0]=='Q')
            {
                printf("%d\n",query(A,B,1,N,1));
            }
            else
            {
                update(A,B,1,N,1);
            }
        }
    }
    return 0;
}
```

- poj 3468 A Simple Problem with Integers

题意参见树状数组例题部分。

需要用到线段树的，update:成段增减，query:区间求和

介绍Lazy思想：lazy－tag思想，记录每一个线段树节点的变化值，当这部分线段的一致性被破坏我们就将这个变化值传递给子区间，大大增加了线段树的效率。

在此通俗的解释Lazy意思，比如现在需要对[a,b]区间值进行加c操作，那么就从根节点[1,n]开始调用update函数进行操作，如果刚好执行到一个子节点，它的节点标记为rt，这时tree[rt].l == a && tree[rt].r == b 这时我们可以一步更新此时rt节点的sum[rt]的值，sum[rt] += c * (tree[rt].r - tree[rt].l + 1)，注意关键的时刻来了，如果此时按照常规的线段树的update操作，这时候还应该更新rt子节点的sum[]值，而Lazy思想恰恰是暂时不更新rt子节点的sum[]值，到此就return，直到下次需要用到rt子节点的值的时候才去更新，这样避免许多可能无用的操作，从而节省时间 。

代码如下：

```
#include <iostream>
#include <cstdio>
using namespace std;
const int N = 100005;
#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1

__int64 sum[N<<2],add[N<<2];
struct Node
{
    int l,r;
    int mid()
    {
        return (l+r)>>1;
    }
} tree[N<<2];

void PushUp(int rt)
{
    sum[rt] = sum[rt<<1] + sum[rt<<1|1];
}

void PushDown(int rt,int m)
{
    if(add[rt])
    {
        add[rt<<1] += add[rt];
        add[rt<<1|1] += add[rt];
        sum[rt<<1] += add[rt] * (m - (m>>1));
        sum[rt<<1|1] += add[rt] * (m>>1);
        add[rt] = 0;
    }
}
```

```
void build(int l,int r,int rt)
{
    tree[rt].l = l;
    tree[rt].r = r;
    add[rt] = 0;
    if(l == r)
    {
        scanf("%I64d",&sum[rt]);
        return ;
    }
    int m = tree[rt].mid();
    build(lson);
    build(rson);
    PushUp(rt);
}

void update(int c,int l,int r,int rt)
{
    if(tree[rt].l == l && r == tree[rt].r)
    {
        add[rt] += c;
        sum[rt] += (__int64)c * (r-l+1);
        return;
    }
    if(tree[rt].l == tree[rt].r) return;
    PushDown(rt,tree[rt].r - tree[rt].l + 1);
    int m = tree[rt].mid();
    if(r <= m) update(c,l,r,rt<<1);
    else if(l > m) update(c,l,r,rt<<1|1);
    else
    {
        update(c,l,m,rt<<1);
        update(c,m+1,r,rt<<1|1);
    }
    PushUp(rt);
}
```

```
__int64 query(int l,int r,int rt)
{
    if(l == tree[rt].l && r == tree[rt].r)
    {
        return sum[rt];
    }
    PushDown(rt,tree[rt].r - tree[rt].l + 1);
    int m = tree[rt].mid();
    __int64 res = 0;
    if(r <= m) res += query(l,r,rt<<1);
    else if(l > m) res += query(l,r,rt<<1|1);
    else
    {
       res += query(l,m,rt<<1);
       res += query(m+1,r,rt<<1|1);
    }
    return res;
}

int main()
{
    int n,m;
    while(~scanf("%d %d",&n,&m))
    {
        build(1,n,1);
        while(m--)
        {
            char ch[2];
            scanf("%s",ch);
            int a,b,c;
            if(ch[0] == 'Q')
            {
                scanf("%d %d", &a,&b);
                printf("%I64d\n",query(a,b,1));
            }
```

```
            else
            {
                scanf("%d %d %d",&a,&b,&c);
                update(c,a,b,1);
            }
        }
    }
    return 0;
}
```

- poj 1151 Atlantis

题意：求矩形的面积并

题解： 求矩形的并，由于矩形的位置可以多变，因此矩形的面积一下子不好求，这个时候，可以采用“分割”的思想，即把整块的矩形面积分割成几个小矩形的面积，然后求和就行了。

这里我们可以这样做，把每个矩形投影到y坐标轴上来，然后我们可以枚举矩形的 x 坐标，然后检测当前相邻x坐标上y方向的合法长度，两种相乘就是面积。然后关键就是如何用线段树来维护那个“合法长度”　　线段树的节点这样定义

```
struct node {
    int left,right,cov;
    double len;
}
```

cov 表示当前节点区间是否被覆盖，len 是当前区间的合法长度

然后我们通过“扫描线”的方法来进行扫描，枚举 x 的竖边，矩形的左边那条竖边就是入边，右边那条就是出边了。然后把所有这些竖边按照 x 坐标递增排序，每次进行插入操作，由于坐标不一定为整数，因此需要进行离散化处理。每次插入时如果当前区间被完全覆盖，那么就要对 cov 域进行更新。入边 +1 出边 -1，更新完毕后判断当前节点的 cov 域是否大于 0 ，如果大于 0，那么当前节点的 len 域就是节点所覆盖的区间。否则，如果是叶子节点，则 len=0。 如果内部节点，则 len=左右儿子的 len 之和。

代码如下:

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

#define L(x) ( x << 1 )
#define R(x) ( x << 1 | 1 )

double y[1000];

struct Line
{
    double x, y1, y2;
    int flag;
} line[300];

struct Node
{
    int l, r, cover;
    double lf, rf, len;
} node[1000];

bool cmp ( Line a, Line b )
{
    return a.x < b.x;
}

void length ( int u )
{
    if ( node[u].cover > 0 )
    {
        node[u].len = node[u].rf - node[u].lf;
        return;
    }
    else if ( node[u].l + 1 == node[u].r )
        node[u].len = 0; /* 叶子节点，len 为 0 */
```

```
    else
        node[u].len = node[L(u)].len + node[R(u)].len;
}

void build ( int u, int l, int r )
{
    node[u].l = l; node[u].r = r;
    node[u].lf = y[l]; node[u].rf = y[r];
    node[u].len = node[u].cover = 0;
    if ( l + 1 == r ) return;
    int mid = ( l + r ) / 2;
    build ( L(u), l, mid );
    build ( R(u), mid, r );
}

void update ( int u, Line e )
{
    if ( e.y1 == node[u].lf && e.y2 == node[u].rf )
    {
        node[u].cover += e.flag;
        length ( u );
        return;
    }
    if ( e.y1 >= node[R(u)].lf )
        update ( R(u), e );
    else if ( e.y2 <= node[L(u)].rf )
        update ( L(u), e );
    else
    {
        Line temp = e;
        temp.y2 = node[L(u)].rf;
        update ( L(u), temp );
        temp = e;
        temp.y1 = node[R(u)].lf;
        update ( R(u), temp );
    }
```

```
    length ( u );
}

int main()
{
    //freopen("a.txt","r",stdin);
    int n, t, i, Case = 0;
    double  x1, y1, x2, y2, ans;
    while ( scanf("%d",&n) && n )
    {
        for ( i = t = 1; i <= n; i++, t++ )
        {
            scanf("%lf%lf%lf%lf",&x1, &y1, &x2, &y2 );
            line[t].x = x1;
            line[t].y1 = y1;
            line[t].y2 = y2;
            line[t].flag = 1;
            y[t] = y1;
            t++;
            line[t].x = x2;
            line[t].y1 = y1;
            line[t].y2 = y2;
            line[t].flag = -1;
            y[t] = y2;
        }

        sort ( line + 1, line + t, cmp );
        sort ( y + 1, y + t );
        build ( 1, 1, t-1 );
        update ( 1, line[1] );

        ans = 0;
        for ( i = 2; i < t; i++ )
        {
            ans += node[1].len * ( line[i].x - line[i-
1].x );
```

```
            update ( 1, line[i] );
        }
        printf ( "Test case #%d\n", ++Case );
        printf ( "Total explored area: %.2lf\n\n", ans );
    }
    return 0;
}
```

二维线段树

与一维线段树类似，把线段树的每一个区间端点想象为一棵新的线段树。我们可以用树套树的方式实现，即每个外层线段树的节点对应于一颗内层线段树。如果外层线段树根对应的区间是x方向的[1,n],那么内层线段树根节点对应的区间是y方向的[1,m],那么整个线段是可以存在一个n行m列的二维数组中。

也可以用一个外层线段树节点力存一颗内层线段树的方式来实现。

所有性质与线段树类似，插入，删除，查找等时间复杂度为O(logn * logm)

我们用一道例题来详细解释二维线段树的用法：

- poj 2155 Matrix

题意：每次操作可以是编辑某个矩形区域，这个区域的0改为1，1改为0，每次查询只查询某一个点的值是0还是1.

使用二维线段树，在修改的时候只需要先找到第一维的对应区间，在在这个区间的弟二维中查找对应区间，再做修改即可。而查找的时候，由于不同的第一维区间可能会有包含关系，所以需要对每个目标所在第一维区间查找第二维区间。

比如线段树的区间大小是３×３，那么在查找第一维区间是〔１，２〕，第二维区间是〔１，２〕时，就需要在线段树第一维的〔１，３〕和〔１，２〕两个区间对第二维进行查找，因为修改操作的时候可能修改了第一维的〔１，３〕区间，同时也修改了〔１，２〕区间，这样的话就不能仅仅只查找某一个第一维的区间。

至于本题的解法，我们可以在修改时标记某一个节点，那么这个节点以下的区间就都是要修改的，当我们在查找的时候，只需要统计查找到这个点时，一路上有多少个被修改的区间，是偶数说明呗修改回来了，是奇数那就是被修改了。

代码如下：

```
#include <stdio.h>
#include <string.h>
#define xlson kx<<1, xl, mid
#define xrson kx<<1|1, mid+1, xr
#define ylson ky<<1, yl, mid
#define yrson ky<<1|1, mid+1, yr
#define MAXN 1005
#define mem(a) memset(a, 0, sizeof(a))

bool tree[MAXN<<2][MAXN<<2];
int  X, N, T;
int num, X1, X2, Y1, Y2;
char ch;

void editY(int kx,int ky,int yl,int yr)
{
    if(Y1<=yl && yr<=Y2)
    {
        tree[kx][ky] = !tree[kx][ky];
        return ;
    }
    int mid = (yl+yr)>>1;
    if(Y1 <= mid) editY(kx,ylson);
    if(Y2 >  mid) editY(kx,yrson);
}

void editX(int kx,int xl,int xr)
{
    if(X1<=xl && xr<=X2)
    {
        editY(kx,1,1,N);
        return ;
    }
    int mid = (xl+xr)>>1;
    if(X1 <= mid) editX(xlson);
```

```
    if(X2 >  mid) editX(xrson);
}

void queryY(int kx,int ky,int yl,int yr)
{
    if(tree[kx][ky]) num ++;
    if(yl==yr) return ;
    int mid = (yl+yr)>>1;
    if(Y1 <= mid) queryY(kx,ylson);
    else queryY(kx,yrson);
}

void queryX(int kx,int xl,int xr)
{
    queryY(kx,1,1,N);
    if(xl==xr) return ;
    int mid = (xl+xr)>>1;
    if(X1 <= mid)queryX(xlson);
    else  queryX(xrson);
}


int main()
{
    while(~scanf("%d", &X))while(X--)
    {
        mem(tree);
        scanf("%d %d%*c", &N,&T);
        for(int i=0;i<T;i++)
        {
            scanf("%c %d %d%*c",&ch,&X1,&Y1);
            if(ch == 'C')
            {
                scanf("%d %d%*c", &X2, &Y2);
                editX(1,1,N);
            }
```

```
            else
            {
                num = 0;
                queryX(1,1,N);
                if(num & 1)printf("1\n");
                else printf("0\n");
            }
        }
        if(X) printf("\n");
    }
    return 0;
}
```

其实使用二维数组也可以解这道题，只是把线段树部分换为树状数组。代码更为短小简洁。

代码如下：

```
#include<cstdio>
#include<string>
#include<iostream>
#define N 1005

int c[N][N],n;

int bit(int n)
{
    return n&(-n);
}

int sum(int x,int y)
{
    int ans=0;
    for(int i=x;i>0;i-=bit(i))
        for(int j=y;j>0;j-=bit(j))
        {
```

```
            ans+=c[i][j];
        }
        return ans;
}

void up(int x,int y,int k)
{
    for(int i=x;i<=n;i+=bit(i))
        for(int j=y;j<=n;j+=bit(j))
        {
            c[i][j]+=k;
        }
}

int main()
{
    int cc,t,x1,x2,y1,y2,a,b;
    char ch;
    scanf("%d",&cc);
    for(int i=0;i<cc;i++)
    {
        memset(c,0,sizeof(c));
        scanf("%d%d",&n,&t);
        getchar();
        for(int j=0;j<t;j++)
        {
            scanf("%c",&ch);
            if(ch=='C')
            {
                scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
                getchar();
                up(x1,y1,1);
                up(x1,y2+1,1);
                up(x2+1,y1,1);
                up(x2+1,y2+1,1);
            }
```

```
            else
            {
                scanf("%d%d",&a,&b);
                getchar();
                printf("%d\n",sum(a,b)%2);
            }
        }
        printf("\n");
    }
    return 0;
}
```

字典树

字典树，又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来节约存储空间，最大限度地减少无谓的字符串比较，查询效率比哈希表高。

字典树与字典很相似,当你要查一个单词是不是在字典树中,首先看单词的第一个字母是不是在字典的第一层,如果不在,说明字典树里没有该单词,如果在就在该字母的孩子节点里找是不是有单词的第二个字母,没有说明没有该单词,有的话用同样的方法继续查找.字典树不仅可以用来储存字母,也可以储存数字等其它数据。

Trie的数据结构定义：

```
#define MAX 26
typedef struct Trie
{
    Trie *next[MAX];
    int v;    //根据需要变化
};

Trie *root;
```

next是表示每层有多少种类的数，如果只是小写字母，则26即可，若改为大小写字母，则是52，若再加上数字，则是62了，这里根据题意来确定。 v可以表示一个字典树到此有多少相同前缀的数目，这里根据需要应当学会自由变化。

Trie的查找（最主要的操作）：

(1) 每次从根结点开始一次搜索；

(2) 取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索；　　(3) 在相应的子树上，取得要查找关键词的第二个字母,并进一步选择对应的子树进行检索。　　(4) 迭代过程.....

(5) 在某个结点处，关键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找。

这里给出生成字典树和查找的模版：

生成字典树：

```
void createTrie(char *str)
{
    int len = strlen(str);
    Trie *p = root, *q;
    for(int i=0; i<len; ++i)
    {
        int id = str[i]-'0';
        if(p->next[id] == NULL)
        {
            q = (Trie *)malloc(sizeof(Trie));
            q->v = 1;    //初始v==1
            for(int j=0; j<MAX; ++j)
                q->next[j] = NULL;
            p->next[id] = q;
            p = p->next[id];
        }
        else
        {
            p->next[id]->v++;
            p = p->next[id];
        }
    }
    p->v = -1;   //若为结尾，则将v改成-1表示
}
```

查找:

```
int findTrie(char *str)
{
    int len = strlen(str);
    Trie *p = root;
    for(int i=0; i<len; ++i)
    {
        int id = str[i]-'0';
        p = p->next[id];
        if(p == NULL)   //若为空集，表示不存以此为前缀的串
            return 0;
        if(p->v == -1)   //字符集中已有串是此串的前缀
            return -1;
    }
    return -1;   //此串是字符集中某串的前缀
}
```

例题

- hdu 1251 统计难题

题意：在给出的字符串中找出由给出的字符串中出现过的两个串拼成的字符串。

字典树的模板题，先建字典数，然后再查询每个给定的单词。。

代码如下:

```
#include <iostream>
#include <string.h>
using namespace std;

const  int sonsum=26,base='a';
char s1[12],ss[12];

struct Trie
```

```
{
    int num;
    bool flag;
    struct Trie *son[sonsum];
    Trie()
    {
        num=1;flag=false;
        memset(son,NULL,sizeof(son));
    }
};

Trie *NewTrie()
{
    Trie *temp=new Trie;
    return temp;
}

void Inset(Trie *root,char *s)
{
    Trie *temp=root;
    while(*s)
    {
        if(temp->son[*s-base]==NULL)
        {
            temp->son[*s-base]=NewTrie();
        }
        else
            temp->son[*s-base]->num++;
        temp=temp->son[*s-base];
        s++;
    }
    temp->flag=true;
}

int search(Trie *root,char *s)
{
```

```
    Trie *temp=root;
    while(*s)
    {
        if(temp->son[*s-base]==NULL) return 0;
        temp=temp->son[*s-base];
        s++;
    }
    return temp->num;
}


int main()
{
    Trie *root=NewTrie();
    root->num=0;
    //while(cin.get(s1,12))
    while(gets(s1)&&strcmp(s1,"")!=0)
    {
        //if(strcmp(s1," ")==0)
        //break;
        Inset(root,s1);
    }
    while(cin>>ss)
    {
        int  ans=search(root,ss);
        cout<<ans<<endl;
    }


    return 0;
}
```

- poj 2001 Shortest Prefixes

题意：找出能唯一标示一个字符串的最短前缀，如果找不出，就输出该字符串。

用字典树即可

代码如下：

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<cstdlib>
using namespace std;

char list[1005][25];

struct node
{
    int count;
    node *childs[26];
    node()
    {
        count=0;
        int i;
        for(i=0;i<26;i++)
        childs[i]=NULL;
    }
};

node *root=new node;
node *current,*newnode;

void insert(char *str)
{
    int i,m;
    current=root;
    for(i=0;i<strlen(str);i++)
    {
```

```
            m=str[i]-'a';
            if(current->childs[m]!=NULL)
            {
                current=current->childs[m];
                ++(current->count);
            }
            else
            {
                newnode=new node;
                ++(newnode->count);
                current->childs[m]=newnode;
                current=newnode;
            }
        }
    }

    void search(char *str)
    {
        int i,m;
        char ans[25];
        current=root;
        for(i=0;i<strlen(str);i++)
        {
            m=str[i]-'a';
            current=current->childs[m];
            ans[i]=str[i];
            ans[i+1]='\0';
            if(current->count==1)    //可以唯一标示该字符串的前缀
            {
                printf("%s %s\n",str,ans);
                return;
            }
        }
        printf("%s %s\n",str,ans);  // 否则输出该字符串
    }
```

```
int main()
{
    int i,t=0;
    while(scanf("%s",list[t])!=EOF)
    {
        insert(list[t]);
        t++;
    }
    for(i=0;i<t;i++)
    search(list[i]);
    return 0;
}
```

- hdu 4825 Xor Sum

题意：给你一些数字，再询问Q个问题，每个问题给一个数字，使这个数字和之前给出的数字的异或和最大。

构造字典树，高位在前，低位在后，然后顺着字典树根向深处递归查询

代码如下：

```
#include<cstdio>
#include<cmath>
#include<cstring>
#include<queue>
#include<vector>
#include<map>
#include<set>
#include<string>
#include<iostream>
#include<functional>
#include<algorithm>

using namespace std;
typedef long long LL;
```

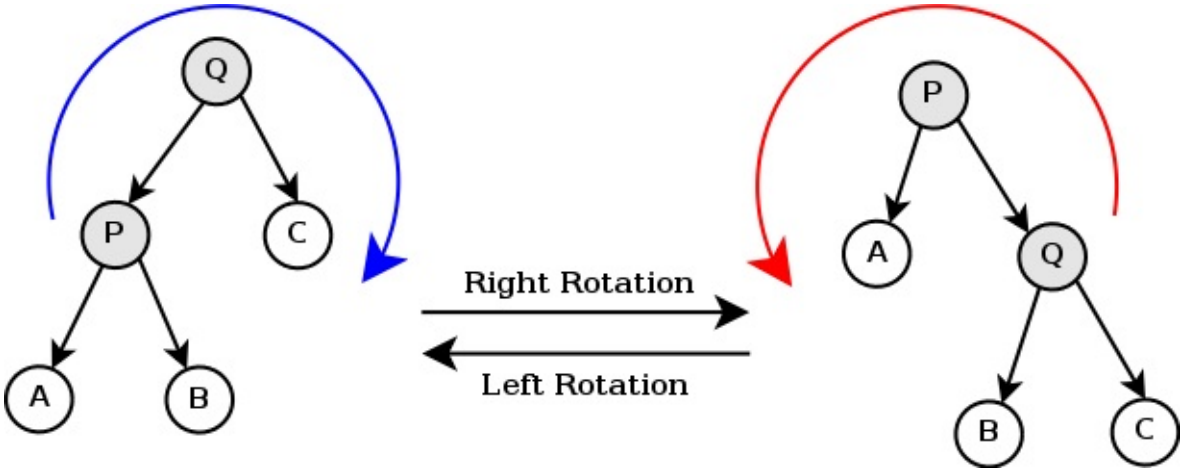
```
typedef pair<LL, int> PLI;

const int MX = 2e5 + 5;
const int INF = 0x3f3f3f3f;

struct Node {
    Node *Next[2];
    Node() {
        Next[0] = Next[1] = NULL;
    }
};

void trie_add(Node*root, int S) {
    Node *p = root;
    for(int i = 31; i >= 0; i--) {
        int id = ((S & (1 << i)) != 0);
        if(p->Next[id] == NULL) {
            p->Next[id] = new Node();
        }
        p = p->Next[id];
    }
}

int trie_query(Node*root, int S) {
    Node *p = root;
    int ans = 0;
    for(int i = 31; i >= 0; i--) {
        int id = ((S & (1 << i)) != 0);
        if(p->Next[id ^ 1] != NULL) {
            ans |= (id ^ 1) << i;
            p = p->Next[id ^ 1];
        } else {
            ans |= id << i;
            p = p->Next[id];
        }
    }
```

```
    return ans;
}

int main() {
    //freopen("input.txt", "r", stdin);
    int T, n, Q, t, ansk = 0;
    scanf("%d", &T);
    while(T--) {
        scanf("%d%d", &n, &Q);
        Node *root = new Node();

        for(int i = 1; i <= n; i++) {
            scanf("%d", &t);
            trie_add(root, t);
        }

        printf("Case #%d:\n", ++ansk);
        while(Q--) {
            scanf("%d", &t);
            printf("%d\n", trie_query(root, t));
        }
    }
    return 0;
}
```

Splay

Splay是一种二叉排序树，空间效率：O(n)，时间效率：O(logn)内完成插入、查找、删除操作，优点：每次查询会调整树的结构，使被查询频率高的条目更靠近树根。

Tree Rotation



树的旋转是splay的基础，对于二叉查找树来说，树的旋转不破坏查找树的结构。

Splaying

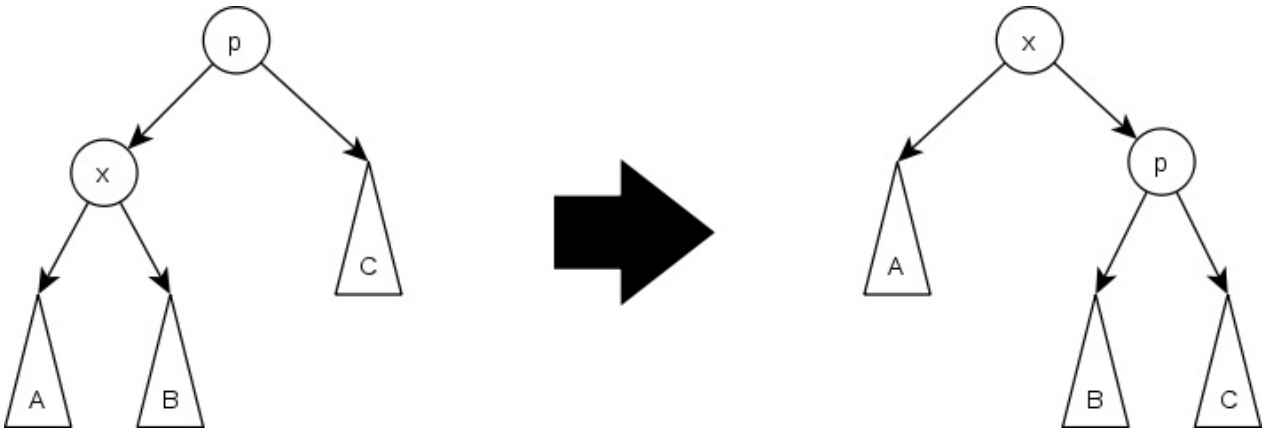
Splaying是Splay Tree中的基本操作，为了让被查询的条目更接近树根，Splay Tree使用了树的旋转操作，同时保证二叉排序树的性质不变。

Splaying的操作受以下三种因素影响：

- 节点x是父节点p的左孩子还是右孩子
- 节点p是不是根节点，如果不是
- 节点p是父节点g的左孩子还是右孩子

同时有三种基本操作：

Zig Step

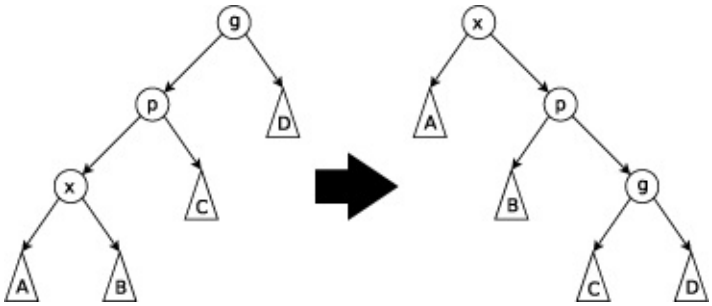


当p为根节点时，进行zip step操作。

当x是p的左孩子时，对x右旋；

当x是p的右孩子时，对x左旋。

Zig-Zig Step

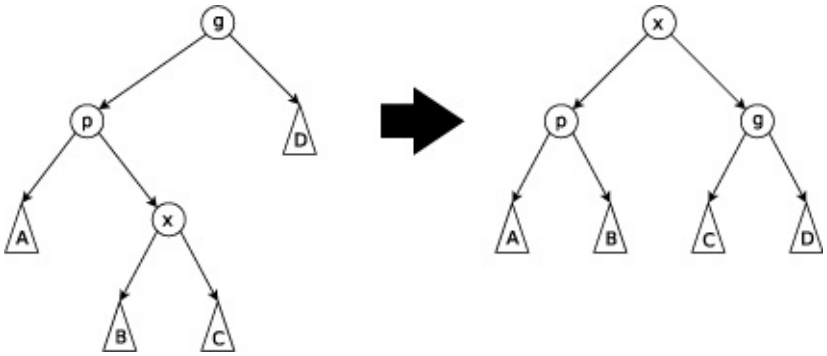


当p不是根节点，且x和p同为左孩子或右孩子时进行Zig-Zig操作。

当x和p同为左孩子时，依次将p和x右旋；

当x和p同为右孩子时，依次将p和x左旋。

Zig-Zag Step



当p不是根节点，且x和p不同为左孩子或右孩子时，进行Zig-Zag操作。

当p为左孩子，x为右孩子时，将x左旋后再右旋。

当p为右孩子，x为左孩子时，将x右旋后再左旋。

下面是splay的伪代码：

```
P(X) : 获得X的父节点，G(X) : 获得X的祖父节点（=P(P(X))）。
Function Buttom-up-splay:
    Do
        If X 是 P(X) 的左子结点 Then
            If G(X) 为空 Then
                X 绕 P（X）右旋
            Else If P（X）是G（X）的左子结点
                P(X) 绕G(X)右旋
                X 绕P(X)右旋
            Else
                X绕P(X)右旋
                X绕P(X)左旋 (P(X)和上面一句的不同，是原来的
G(X))
            Endif
        Else If X 是 P(X) 的右子结点 Then
            If G(X) 为空 Then
                X 绕 P（X）左旋
            Else If P（X）是G（X）的右子结点
                P(X) 绕G(X)左旋
                X 绕P(X)左旋
            Else
                X绕P(X)左旋
                X绕P(X)右旋 (P(X)和上面一句的不同，是原来的
G(X))
            Endif
        Endif
    While (P(X) != NULL)
EndFunction
```

仔细分析zig-zag，可以发现，其实zig-zag就是两次zig。因此上面的代码可以简化：

```
Function Buttom-up-splay:
    Do
        If X 是 P(X) 的左子结点 Then
            If P（X）是G（X）的左子结点
                P(X) 绕G(X)右旋
            Endif
```

```
            X 绕P(X)右旋
        Else If X 是 P(X) 的右子结点 Then
            If P（X）是G（X）的右子结点
                P(X) 绕G(X)左旋
            Endif
            X 绕P(X)左旋
        Endif
    While (P(X) != NULL)
EndFunction

Function Top-Down-Splay
 Do
      If X 小于 T Then
           If X 等于 T 的左子结点 Then
             右连接
           ElseIf X 小于 T 的左子结点 Then
             T的左子节点绕T右旋
             右连接
           Else X大于 T 的左子结点 Then
             右连接
             左连接
           EndIf
      ElseIf X大于 T Then
           IF X 等于 T 的右子结点 Then
             左连接
           ElseIf X 大于 T 的右子结点 Then
             T的右子节点绕T左旋
             左连接
           Else X小于 T 的右子结点‘ Then
             左连接
             右连接
           EndIf
      EndIf
 While  ！(找到 X或遇到空节点)
  组合左中右树
EndFunction
```

模板代码如下：

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

#define MAXN 100010

struct Node{
    int key, sz, cnt;
    Node *ch[2], *pnt;//左右儿子和父亲
    Node(){}

    Node(int x, int y, int z){
    key = x, sz = y, cnt = z;
    }

    void rs(){
    sz = ch[0]->sz + ch[1]->sz + cnt;
    }

}nil(0, 0, 0), *NIL = &nil;

struct Splay{//伸展树结构体类型
    Node *root;
    int ncnt;//计算key值不同的结点数，注意已经去重了
    Node nod[MAXN];

    void init(){// 首先要初始化
        root = NIL;
        ncnt = 0;
    }

    void rotate(Node *x, bool d){//旋转操作，d为true表示右旋
        Node *y = x->pnt;
        y->ch[!d] = x->ch[d];
```

```
        if (x->ch[d] != NIL)
            x->ch[d]->pnt = y;
        x->pnt = y->pnt;
        if (y->pnt != NIL){
            if (y == y->pnt->ch[d])
                y->pnt->ch[d] = x;
            else
                y->pnt->ch[!d] = x;
        }
        x->ch[d] = y;
        y->pnt = x;
        y->rs();
        x->rs();
    }

    void splay(Node *x, Node *target){//将x伸展到target的儿子位置处
        Node *y;
        while (x->pnt != target){
            y = x->pnt;
            if (x == y->ch[0]){
                if (y->pnt != target && y == y->pnt->ch[0])
                    rotate(y, true);
                    rotate(x, true);
            }
            else{
                if (y->pnt != target && y == y->pnt->ch[1])
                    rotate(y, false);
                    rotate(x, false);
            }
        }
        if (target == NIL)
            root = x;
    }
```

```
    /************************以上一般不用修改
**********************/

    void insert(int key){//插入一个值
        if (root == NIL){
            ncnt = 0;
            root = &nod[++ncnt];
            root->ch[0] = root->ch[1] = root->pnt = NIL;
            root->key = key;
            root->sz = root->cnt = 1;
            return;
        }
        Node *x = root, *y;
        while (1){
            x->sz++;
            if (key == x->key){
                x->cnt++;
                x->rs();
                y = x;
                break;
            }
            else if (key < x->key){
                    if (x->ch[0] != NIL)
                        x = x->ch[0];
                    else{
                        x->ch[0] = &nod[++ncnt];
                        y = x->ch[0];
                        y->key = key;
                        y->sz = y->cnt = 1;
                        y->ch[0] = y->ch[1] = NIL;
                        y->pnt = x;
                        break;
                    }
            }
            else{
```

```
                if (x->ch[1] != NIL)
                    x = x->ch[1];
                else{
                    x->ch[1] = &nod[++ncnt];
                    y = x->ch[1];
                    y->key = key;
                    y->sz = y->cnt = 1;
                    y->ch[0] = y->ch[1] = NIL;
                    y->pnt = x;
                    break;
                }
            }
        }
        splay(y, NIL);
    }

    Node* search(int key){//查找一个值，返回指针
        if (root == NIL)
            return NIL;
        Node *x = root, *y = NIL;
        while (1){
            if (key == x->key){
                y = x;
                break;
            }
            else if (key > x->key){
                if (x->ch[1] != NIL)
                x = x->ch[1];
                else
                    break;
            }
            else{
                if (x->ch[0] != NIL)
                    x = x->ch[0];
                else
                    break;
```

```
            }
        }
        splay(x, NIL);
        return y;
    }

    Node* searchmin(Node *x){//查找最小值，返回指针
        Node *y = x->pnt;
        while (x->ch[0] != NIL){//遍历到最左的儿子就是最小值
            x = x->ch[0];
        }
            splay(x, y);
            return x;
    }

    void del(int key){//删除一个值
        if (root == NIL)
            return;
        Node *x = search(key), *y;
        if (x == NIL)
            return;
        if (x->cnt > 1){
            x->cnt--;
            x->rs();
            return;
        }
        else if (x->ch[0] == NIL && x->ch[1] == NIL){
            init();
            return;
        }
        else if (x->ch[0] == NIL){
            root = x->ch[1];
            x->ch[1]->pnt = NIL;
            return;
        }
        else if (x->ch[1] == NIL){
```

```
            root = x->ch[0];
            x->ch[0]->pnt = NIL;
            return;
        }
        y = searchmin(x->ch[1]);
        y->pnt = NIL;
        y->ch[0] = x->ch[0];
        x->ch[0]->pnt = y;
        y->rs();
        root = y;
    }

    int rank(int key){//求结点高度
        Node *x = search(key);
        if (x == NIL)
            return 0;
        return x->ch[0]->sz + 1/* or x->cnt*/;
    }

    Node* findk(int kth){//查找第k小的值
        if (root == NIL || kth > root->sz)
            return NIL;
        Node *x = root;
        while (1){
            if (x->ch[0]->sz +1 <= kth && kth <= x-
>ch[0]->sz + x->cnt)
                break;
            else if (kth <= x->ch[0]->sz)
                x = x->ch[0];
            else{
                kth -= x->ch[0]->sz + x->cnt;
                x = x->ch[1];
            }
        }
        splay(x, NIL);
        return x;
```

```
    }
}sp;

int main(){
    sp.init();
    sp.insert(10);
    sp.insert(2);
    sp.insert(2);
    sp.insert(2);
    sp.insert(3);
    sp.insert(3);
    sp.insert(10);
    for (int i = 1; i <= 7; i++)
        cout << sp.findk(i)->key << endl;
    cout<<sp.searchmin(sp.root)->key<<endl;
    sp.del(2);
    sp.del(3);
    sp.del(1);
    return 0;
}
```

应用

Splay Tree可以方便的解决一些区间问题，根据不同形状二叉树先序遍历结果不变的特性，可以将区间按顺序建二叉查找树。

每次自下而上的一套splay都可以将x移动到根节点的位置，利用这个特性，可以方便的利用Lazy的思想进行区间操作。

对于每个节点记录size，代表子树中节点的数目，这样就可以很方便地查找区间中的第k小或第k大元素。

对于一段要处理的区间[x, y]，首先splay x-1到root，再splay y+1到root的右孩子，这时root的右孩子的左孩子对应子树就是整个区间。 这样，大部分区间问题都可以很方便的解决，操作同样也适用于一个或多个条目的添加或删除，和区间的移动。

- poj 2761 Feed the dogs

题意：求区间第k小数，可以用划分树来做，这里区间不会重叠，所以不可能有首首相同或尾尾相同的情况，读入所有区间，按照右端由小到大排序。然后通过维护splay进行第k小元素的查询操作。

代码如下：

```
#include<cstdio>
#include<cstring>
#include<iostream>
#include<algorithm>
using namespace std;
const int maxn=100100;
int n,m,sorted[maxn],tree[23][maxn],toleft[23][maxn];
void Build(int l,int r,int deep)
{
    if(l==r)
    return;
    int mid=(l+r)>>1;
    int same=mid-l+1;
    for(int i=l;i<=r;i++)
    if(tree[deep][i]<sorted[mid])
        same--;
    int ls=l,rs=mid+1;
    for(int i=l;i<=r;i++)
    {
    toleft[deep][i]=toleft[deep][i-1];
    if(tree[deep][i]<sorted[mid])
    {
        tree[deep+1][ls++]=tree[deep][i];
        toleft[deep][i]++;
    }
    else if(tree[deep][i]==sorted[mid])
    {
        if(same)
        {
```

```
            tree[deep+1][ls++]=tree[deep][i];
            toleft[deep][i]++;
            same--;
            }
            else
            tree[deep+1][rs++]=tree[deep][i];
        }
        else
            tree[deep+1][rs++]=tree[deep][i];
        }
        Build(l,mid,deep+1);
        Build(mid+1,r,deep+1);
    }
    int Query(int l,int r,int L,int R,int deep,int k)
    {
        if(l==r)
        return tree[deep][l];
        int mid=(L+R)>>1;
        int x=toleft[deep][l-1]-toleft[deep][L-1];
        int y=toleft[deep][r]-toleft[deep][L-1];
        int ry=r-L-y;
        int rx=l-L-x;
        int cnt=y-x;
        if(cnt>=k)
        return Query(L+x,L+y-1,L,mid,deep+1,k);
        else
        return Query(mid+rx+1,mid+ry+1,mid+1,R,deep+1,k-cnt);
    }
    int main()
    {
        while(scanf("%d%d",&n,&m)!=EOF)
        {
        for(int i=1;i<=n;i++)
        {
            scanf("%d",&sorted[i]);
            tree[0][i]=sorted[i];
```

```
    }
    sort(sorted+1,sorted+1+n);
    Build(1,n,0);
    while(m--)
    {
        int a,b,k;
        scanf("%d%d%d",&a,&b,&k);
        printf("%d\n",Query(a,b,1,n,0,k));
    }
    }
    return 0;
}
```

- bzoj 1588 营业额统计

题意：给出一个n个数的数列a,对于第i个元素ai定义fi=min(abs(ai-aj))，(1<=j<i)，其中f1=a1。输出sum(fi) (1<=i<=n)

splay模板题，代码如下：

```
#include<iostream>
#include<string>
#include<algorithm>
#include<cstdlib>
#include<cstdio>
#include<set>
#include<map>
#include<vector>
#include<cstring>
#include<stack>
#include<cmath>
#include<queue>
using namespace std;
#define CL(x,v); memset(x,v,sizeof(x));
#define INF 0x3f3f3f3f
#define LL long long
#define REP(i,r,n) for(int i=r;i<=n;i++)
```

```
#define RREP(i,n,r) for(int i=n;i>=r;i--)
const int MAXN=200010;
const int mod=1000000;

struct SplayTree {
    int sz[MAXN];
    int ch[MAXN][2];
    int pre[MAXN];
    int rt,top;
    inline void up(int x){
        sz[x]  = cnt[x]  + sz[ ch[x][0] ] + sz[ ch[x][1]
];
    }
    inline void Rotate(int x,int f){
        int y=pre[x];
        ch[y][!f] = ch[x][f];
        pre[ ch[x][f] ] = y;
        pre[x] = pre[y];
        if(pre[x]) ch[ pre[y] ][ ch[pre[y]][1] == y ] =x;
        ch[x][f] = y;
        pre[y] = x;
        up(y);
    }
    inline void Splay(int x,int goal){//将x旋转到goal的下面
        while(pre[x] != goal){
            if(pre[pre[x]] == goal) Rotate(x , ch[pre[x]]
[0] == x);
            else   {
                int y=pre[x],z=pre[y];
                int f = (ch[z][0]==y);
                if(ch[y][f] == x)
Rotate(x,!f),Rotate(x,f);
                else Rotate(y,f),Rotate(x,f);
            }
        }
        up(x);
```

```
        if(goal==0) rt=x;
    }
    inline void RTO(int k,int goal){//将第k位数旋转到goal的
下面
        int x=rt;
        while(sz[ ch[x][0] ] != k-1) {
            if(k < sz[ ch[x][0] ]+1) x=ch[x][0];
            else {
                k-=(sz[ ch[x][0] ]+1);
                x = ch[x][1];
            }
        }
        Splay(x,goal);
    }
    inline void vist(int x){
        if(x){
            printf("结点%2d : 左儿子  %2d   右儿子  %2d
%2d sz=%d\n",x,ch[x][0],ch[x][1],val[x],sz[x]);
            vist(ch[x][0]);
            vist(ch[x][1]);
        }
    }
    inline void Newnode(int &x,int c){
        x=++top;
        ch[x][0] = ch[x][1] = pre[x] = 0;
        sz[x]=1; cnt[x]=1;
        val[x] = c;
    }
    inline void init(){
        ch[0][0]=ch[0][1]=pre[0]=sz[0]=0;
        rt=top=0; cnt[0]=0;
        Newnode(rt,-INF);
        Newnode(ch[rt][1],INF);
        pre[top]=rt;
        sz[rt]=2;
    }
```

```
inline void Insert(int &x,int key,int f){
    if(!x) {
        Newnode(x,key);
        pre[x]=f;
        Splay(x,0);//效率的保证
        return ;
    }
    if(key==val[x]){
        cnt[x]++;
        sz[x]++;
        Splay(x,0);//不加会超时，囧啊
        return ;
    }else if(key<val[x]) {
        Insert(ch[x][0],key,x);
    } else {
        Insert(ch[x][1],key,x);
    }
    up(x);
}
void findpre(int x,int key,int &ans){
    if(!x)  return ;
    if(val[x] <= key){
        ans=val[x];
        findpre(ch[x][1],key,ans);
    } else
        findpre(ch[x][0],key,ans);
}
void findsucc(int x,int key,int &ans){
    if(!x) return ;
    if(val[x]>=key) {
        ans=val[x];
        findsucc(ch[x][0],key,ans);
    } else
        findsucc(ch[x][1],key,ans);
}
int cnt[MAXN];
```

```
    int val[MAXN];

}spt;
int main()
{
    int n;
    scanf("%d",&n);
    spt.init();
    int ans=0;
    int a;
    scanf("%d",&a);
    spt.Insert(spt.rt,a,0);
    ans=a;
    n--;
    while(n--)
    {
        a=0;//不知道为什么这里要赋值为0，不赋值就wa!!
        scanf("%d",&a);
        int x,y;
        spt.findpre(spt.rt,a,x);
        spt.findsucc(spt.rt,a,y);
        if(abs(a-x)<=abs(a-y))
        {
            ans+=abs(a-x);
        }
        else
        {
            ans+=abs(a-y);
        }
        spt.Insert(spt.rt,a,0);
    }
    printf("%d\n",ans);
    return 0;
}
```

ST表

ST算法（Sparse Table）：

它是一种动态规划的方法。以最小值为例。a为所寻找的数组，用一个二维数组 f(i,j)记录区间[i, i+2^j-1]区间中的最小值。其中f[i,0] = a[i];所以，对于任意 的一组(i,j)，f(i,j) = min{f(i,j-1),f(i+2^(j-1),j-1)}来使用动态规划计算出来。

这个算法的高明之处不是在于这个动态规划的建立，而是它的查询：它的查询效 率是O(1)！如果不细想的话，怎么弄也是不会想到有O（1）的算法的。假设我们要求 区间[m,n]中a的最小值，找到一个数k使得2^k<n-m+1，即k=[ln(b-a+1)/ln(2)] 这样， 可以把这个区间分成两个部分：[m,m+2^k-1]和[n-2^k+1,n]！我们发现，这两个区间 是已经初始化好的！前面的区间是f(m,k)，后面的区间是f(n-2^k+1,k)！这样，只要 看这两个区间的最小值，就可以知道整个区间的最小值！

小结：

稀疏表(SparseTable)算法是O(nlogn)-O(1)的，对于查询很多大的情况下比较好。

ST算法预处理：用dp[i,j]表示从i开始的，长度为2^j 的区间的RMQ，则有递推式

dp[i,j]=min{dp[i,j-1],dp[i+2j-1,j-1]}

即用两个相邻的长度为2j-1的块，更新长度为2j的块。因此，预处理时间复杂度为O(nlogn)。

这个算法记录了所有长度形如2k的所有询问的结果。

从这里可以看出，稀疏表算法的空间复杂度为 O(nlogn)。

模板如下：

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

int a[50001];
int f[50001][16];
int n;
```

```
void rmq_init() //建立： dp(i,j) = min{dp(i,j-1),dp(i+2^(j-1),j-1)   O(nlogn)
{
   for(int i=1; i<=n; i++)
      f[i][0] = a[i];
   int k=floor(log((double)n)/log(2.0)); //C/C++取整函数 ceil()大,floor()小
   for(int j=1; j<=k; j++)
      for(int i=n; i>=1; i--)
      {
         if(i+(1<<(j-1))<=n) //f(i,j) = min{f(i,j-1),f(i+2^(j-1),j-1)
            f[i][j]=min(f[i][j-1],f[i+(1<<(j-1))][j-1]);
      }
}

int rmq(int i,int j)  //查询：返回区间[i,j]的最小值      O(1)
{
    int k = floor(log((double)(j-i+1))/log(2.0));
    return min(f[i][k],f[j-(1<<k)+1][k]);
}

int main()
{
   scanf("%d",&n);
   for(int i=1; i<=n; i++)
      scanf("%d",&a[i]);
    rmq_init();
    printf("%d\n", rmq(2,5));
}
```

- poj 3264 Balanced Lineup

  题意：求区间的最大值和最小值。

  这道题有很多种方法（比如线段树），用ST表代码简洁，详细代码如下：

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<cmath>
#define N 100005
using namespace std;
int n,q,a[N];
int mx[N][18],mn[N][18];
void Rmq_Init(){
    int m=floor(log((double)n)/log(2.0));
    for(int i=1;i<=n;i++) mx[i][0]=mn[i][0]=a[i];
    for(int i=1;i<=m;i++)
        for(int j=n;j;j--){
            mx[j][i]=mx[j][i-1];
            mn[j][i]=mn[j][i-1];
            if(j+(1<<(i-1))<=n){
                mx[j][i]=max(mx[j][i],mx[j+(1<<(i-1))][i-
1]);
                mn[j][i]=min(mn[j][i],mn[j+(1<<(i-1))][i-
1]);
            }
        }
}
int Rmq_Query(int l,int r){
    int m=floor(log((double)(r-l+1))/log(2.0));
    int Max=max(mx[l][m],mx[r-(1<<m)+1][m]);
    int Min=min(mn[l][m],mn[r-(1<<m)+1][m]);
    return Max-Min;
}
int main(){
    while(scanf("%d%d",&n,&q)!=EOF){
        for(int i=1;i<=n;i++) scanf("%d",&a[i]);
        Rmq_Init();
        while(q--){
```

```
            int l,r;
            scanf("%d%d",&l,&r);
            printf("%d\n",Rmq_Query(l,r));
        }
    }
    return 0;
}
```

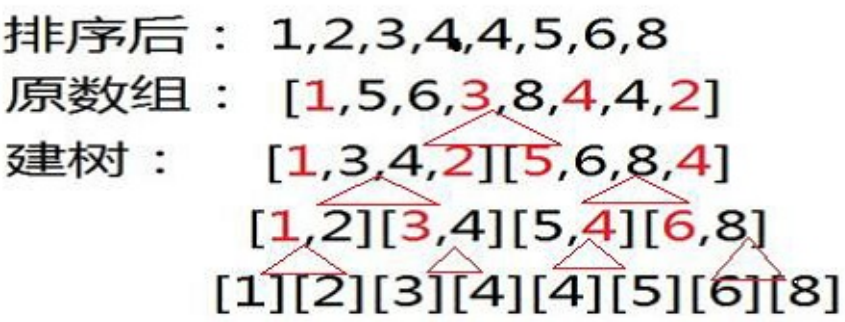
划分树

划分树是一种基于线段树的数据结构。主要用于快速求出(在log(n)的时间复杂度内）序列区间的第k大值。

查找整序列的第k大值往往采用。然而此方法会破坏原序列，并且需要O(n)的时间复杂度。抑或使用二叉平衡树进行维护，此方法每次查找时间复杂度仅为O(logn)。然而此方法丢失了原序列的顺序信息，无法查找出某区间内的第k大值。

划分树的基本思想就是对于某个区间，把它划分成两个子区间，左边区间的数小于右边区间的数。查找的时候通过记录进入左子树的数的个数，确定下一个查找区间，最后范围缩小到1，就找到了。

划分树定义为，它的每一个节点保存区间[lft,rht]所有元素，元素顺序与原数组（输入）相同，但是，两个子树的元素为该节点所有元素排序后(rht-lft+1)/2个进入左子树，其余的到右子树，同时维护一个num域，num[i]表示lft->i这个点有多少个进入了左子树。



如果由下而上看这个图，我们就会发现它和归并排序的（归并树）的过程很类似，或者说正好相反。归并树是由下而上的排序，而它确实是由上而下的排序，但这正是它可以用来解决第k大元素的理由所在。

划分树的存储结构（采用层次存储结构（由下而上，由左到右，每层两个孩子，见上图））

```
constint N=1e5+5;
int sorted[N];              //对原来集合中的元素排序后的值
struct node
{
    int valu[N];        //val记录第k层当前位置元素的值
    int num[N];         //num记录元素所在区间的当前位置之前进入左孩子的个数
    LL sum[N];          //sum记录比当前元素小的元素的和
}t[20];
```

划分树的建立

划分树的建立和普通的二叉树的建立过程差不多，仍然采取中序的过程（先根节点，然后左右孩子）。

树的建立相对比较简单，我们依据的是已经排好序的位置进行建树，所以先用快排将原集合还序。要维护每个节点的num域。

```
void build(int lft,int rht,int ind)
{
    if(lft==rht) return;
    int mid=MID(lft,rht);
    int same=mid-lft+1,ln=lft,rn=mid+1;
    for(int i=lft;i<=rht;i++)
        if(valu[ind][i]<order[mid]) same--;
    for(int i=lft;i<=rht;i++)
    {
        int flag=0;
        if((valu[ind][i]<order[mid])||valu[ind]
[i]==order[mid]&&same>0)
        {
            flag=1;
            valu[ind+1][ln++]=valu[ind][i];
            if(valu[ind][i]==order[mid]) same--;
            lsum[ind][i]=lsum[ind][i-1]+valu[ind][i];
        }
        else
        {
            lsum[ind][i]=lsum[ind][i-1];
            valu[ind+1][rn++]=valu[ind][i];
        }
        toLft[ind][i]=toLft[ind][i-1]+flag;
    }
    build(lft,mid,ind+1);
    build(mid+1,rht,ind+1);
}
```

划分树的查找

在区间[a,b]上查找第k大的元素，同时返回它的位置和区间小于[a,b]的所有数的和 。

1.如果t[p].num[b]-t[p].num[a-1]>=k，即，进入左孩子的个数已经超过k个，那么就往左孩子里面查找，同时更新[a,b]=>[lft+t[p].num[a-1]，lft+t[p].num[b]-1]

2.如果t[p].num[b]-t[p].num[a-1]<k，即，进入p的左孩子的个数小于k个，那么就要往右孩子查找第k-s(s表示进入左孩子的个数)个元素。同时更新sum域，因而这样求出的sum只是严格小于在[a,b]区间中第k大的数的和。

```
int query(int st,int ed,int k,int lft,int rht,int ind)
{
    if(lft==rht) return valu[ind][lft];
    /*
          lx表示从lft到st-1这段区间内有多少个数进入左子树
        ly表示从st到ed这段区间内有多少个数进入左子树
        rx表示从lft到st-1这段区间内有多少个数进入右子树
        ry表示从st到ed这段区间内有多少个数进入右子树
     */
    int mid=MID(lft,rht);
    int lx=toLft[ind][st-1]-toLft[ind][lft-1];
    int ly=toLft[ind][ed]-toLft[ind][st-1];
    int rx=st-1-lft+1-lx;
    int ry=ed-st+1-ly;
    if(ly>=k) return query(lft+lx,lft+lx+ly-
1,k,lft,mid,ind+1);
    else
    {
        isum+=lsum[ind][ed]-lsum[ind][st-1];
        st=mid+1+rx;
        ed=mid+1+rx+ry-1;
        return query(st,ed,k-ly,mid+1,rht,ind+1);
    }
}
```

- poj 2104 K-th Number

题意：有n个数字排成一列，有m个询问，格式为：left right k 即问在区间[left,right]第k大的数据为多少？

直接用划分树的模板即可。

代码如下:

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

#define MID(a,b) (a+((b-a)>>1))

typedef long long LL;
const int N=1e5+5;

struct P_Tree
{
    int n,order[N];
    int valu[20][N],num[20][N];
    LL sum[N],lsum[20][N],isum;

    void init(int len)
    {
        n=len;    sum[0]=0;
        for(int i=0;i<20;i++) valu[i][0]=0,num[i]
[0]=0,lsum[i][0]=0;
        for(int i=1;i<=n;i++)
        {
            scanf("%d",&order[i]);
            valu[0][i]=order[i];
            sum[i]=sum[i-1]+order[i];
        }
        sort(order+1,order+1+n);
        build(1,n,0);
    }

    void build(int lft,int rht,int ind)
    {
        if(lft==rht) return;
```

```
        int mid=MID(lft,rht);
        int same=mid-lft+1,ln=lft,rn=mid+1;
        for(int i=lft;i<=rht;i++)
            if(valu[ind][i]<order[mid]) same--;
        for(int i=lft;i<=rht;i++)
        {
            int flag=0;
            if((valu[ind][i]<order[mid])||(valu[ind]
[i]==order[mid]&&same))
            {
                flag=1;
                valu[ind+1][ln++]=valu[ind][i];
                lsum[ind][i]=lsum[ind][i-1]+valu[ind][i];
                if(valu[ind][i]==order[mid]) same--;
            }
            else
            {
                valu[ind+1][rn++]=valu[ind][i];
                lsum[ind][i]=lsum[ind][i-1];
            }
            num[ind][i]=num[ind][i-1]+flag;
        }
        build(lft,mid,ind+1);
        build(mid+1,rht,ind+1);
    }

    int query(int st,int ed,int k,int lft,int rht,int
ind)
    {
        if(lft==rht) return valu[ind][lft];

        int mid=MID(lft,rht);
        int lx=num[ind][st-1]-num[ind][lft-1];
        int ly=num[ind][ed]-num[ind][st-1];
        int rx=st-1-lft+1-lx;
        int ry=ed-st+1-ly;
```

```
        if(ly>=k) return query(lft+lx,lft+lx+ly-
1,k,lft,mid,ind+1);
        else
        {
            isum+=lsum[ind][ed]-lsum[ind][st-1];
            st=mid+1+rx;
            ed=mid+1+rx+ry-1;
            return query(st,ed,k-ly,mid+1,rht,ind+1);
        }
    }

}tree;

int main()
{
    int n,m;
    while(scanf("%d%d",&n,&m)!=EOF)
    {
        tree.init(n);
        for(int i=0;i<m;i++)
        {
            int a,b,c;
            scanf("%d%d%d",&a,&b,&c);
            int res=tree.query(a,b,c,1,n,0);
            printf("%d\n",res);
        }
    }
    return 0;
}
```

树链剖分

“在一棵树上进行路径的修改、求极值、求和”乍一看只要线段树就能轻松解决，实际上，仅凭线段树是不能搞定它的。我们需要用到一种貌似高级的复杂算法——树链剖分。

树链，就是树上的路径。剖分，就是把路径分类为重链和轻链。

记siz[v]表示以v为根的子树的节点数，dep[v]表示v的深度(根深度为1)，top[v]表示v所在的链的顶端节点，fa[v]表示v的父亲，son[v]表示与v在同一重链上的v的儿子节点（姑且称为重儿子），w[v]表示v与其父亲节点的连边（姑且称为v的父边）在线段树中的位置。只要把这些东西求出来，就能用logn的时间完成原问题中的操作。

重儿子：siz[u]为v的子节点中siz值最大的，那么u就是v的重儿子。

轻儿子：v的其它子节点。

重边：点v与其重儿子的连边。

轻边：点v与其轻儿子的连边。

重链：由重边连成的路径。

轻链：轻边。

剖分后的树有如下性质：

性质**1**：如果(v,u)为轻边，则siz[u] *2 < siz[v]*；

*性质*2*：从根到某一点的路径上轻链、重链的个数都不大于logn。

算法实现：

我们可以用两个dfs来求出fa、dep、siz、son、top、w。

dfs_1：把fa、dep、siz、son求出来，比较简单，略过。

dfs_2： 1.对于v，当son[v]存在（即v不是叶子节点）时，显然有top[son[v]] = top[v]。线段树中，v的重边应当在v的父边的后面，记w[son[v]] = totw+1，totw表示最后加入的一条边在线段树中的位置。此时，为了使一条重链各边在线段树中连续分布，应当进行dfs_2(son[v])；

2.对于v的各个轻儿子u，显然有top[u]=u，并且w[u]=totw+1，进行dfs_2过程。这就求出了top和w。将树中各边的权值在线段树中更新，建链和建线段树的过程就完成了。

修改操作：例如将u到v的路径上每条边的权值都加上某值x。

一般人需要先求LCA，然后慢慢修改u、v到公共祖先的边。

记f1 = top[u]，f2 = top[v]。

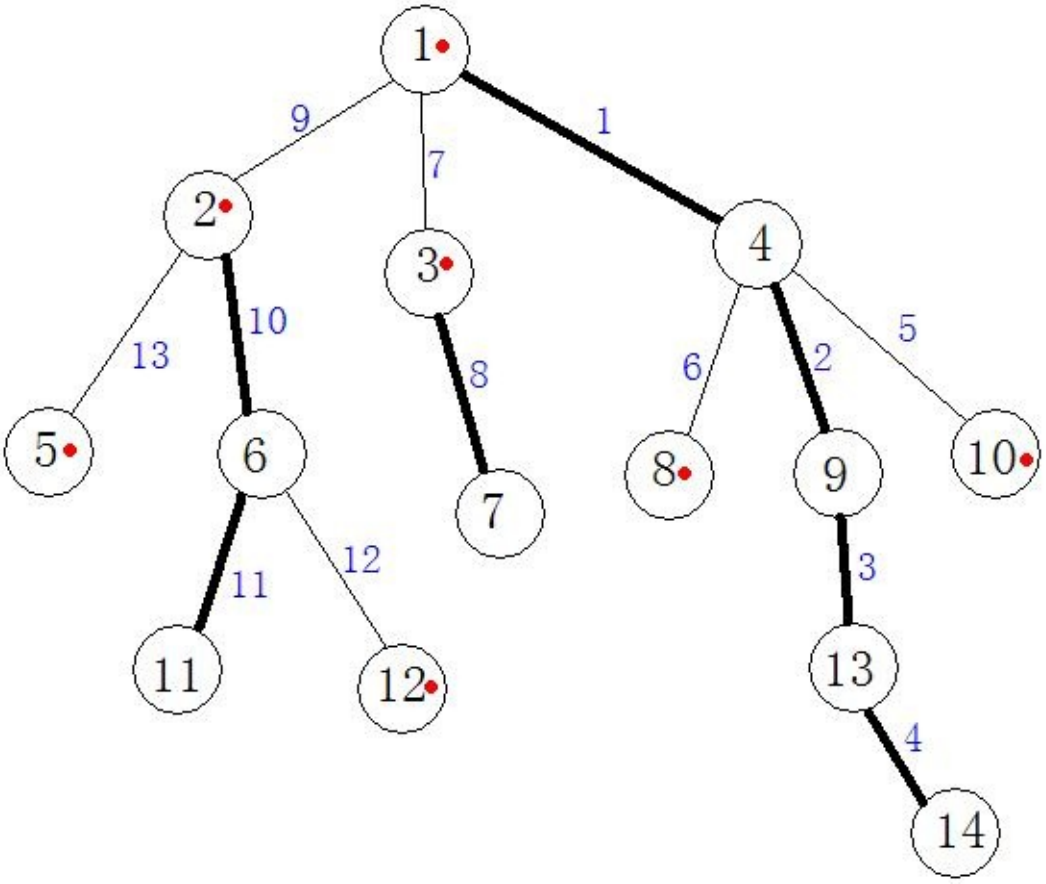
当f1 <> f2时：不妨设dep[f1] >= dep[f2]，那么就更新u到f1的父边的权值(logn)，并使u = fa[f1]。

当f1 = f2时：u与v在同一条重链上，若u与v不是同一点，就更新u到v路径上的边的权值(logn)，否则修改完成；

重复上述过程，直到修改完成。

求和、求极值操作：类似修改操作，但是不更新边权，而是对其求和、求极值。

如图所示，较粗的为重边，较细的为轻边。节点编号旁边有个红色点的表明该节点是其所在链的顶端节点。边旁的蓝色数字表示该边在线段树中的位置。图中1-4-9-13-14为一条重链。



当要修改11到10的路径时。

第一次迭代：u = 11，v = 10，f1 = 2，f2 = 10。此时dep[f1] < dep[f2]，因此修改线段树中的5号点，v = 4, f2 = 1；

第二次迭代：dep[f1] > dep[f2]，修改线段树中10--11号点。u = 2，f1 = 2；

第三次迭代：dep[f1] > dep[f2]，修改线段树中9号点。u = 1，f1 = 1；

第四次迭代：f1 = f2且u = v，修改结束。

- hdu 3966 Aragorn's Story

题意：给一棵树，并给定各个点权的值，然后有3种操作：
I C1 C2 K: 把C1与C2的路径上的所有点权值加上K
D C1 C2 K：把C1与C2的路径上的所有点权值减去K
Q C：查询节点编号为C的权值

分析：典型的树链剖分题目，先进行剖分，然后用线段树去维护即可

```
#include<cstdio>
#include<cstring>
#include<stack>
#include<algorithm>
using namespace std;

#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1

typedef int lld;

stack<int> ss;
const int maxn = 200010;
const int inf = ~0u>>2;
int M[maxn<<2];
int add[maxn<<2];

struct node{
    int s,t,w,next;
}edge[maxn*2];

int E,n;
int size[maxn] , fa[maxn] , heavy[maxn] , head[maxn] ,
vis[maxn];
int dep[maxn] , rev[maxn] , num[maxn] , cost[maxn]
,w[maxn];
int Seg_size;

inline void Max(int &x,int y){if(x<y) x=y;}
```

```
int find(int x){return x==fa[x]?x:fa[x]=find(fa[x]);}

void add_edge(int s,int t,int w)
{
    edge[E].w=w;
    edge[E].s=s;
    edge[E].t=t;
    edge[E].next=head[s];
    head[s]=E++;
}

void dfs(int u,int f)
{
    int mx=-1,e=-1;
    size[u]=1;
    for(int i=head[u];i!=-1;i=edge[i].next)
    {
        int v=edge[i].t;
        if(v==f) continue;
        edge[i].w=edge[i^1].w=w[v];
        dep[v]=dep[u]+1;
        rev[v]=i^1;
        dfs(v,u);
        size[u]+=size[v];
        if(size[v]>mx)
        {
            mx=size[v];
            e=i;
        }
    }
    heavy[u]=e;
    if(e!=-1)  fa[edge[e].t]=u;
}

inline void pushup(int rt){
```

```
    M[rt]=M[rt<<1]+M[rt<<1|1];
}

void pushdown(int rt,int m){
    if(add[rt]){
        add[rt<<1]+=add[rt];
        add[rt<<1|1]+=add[rt];
        M[rt<<1]+=add[rt]*(m-(m>>1));
        M[rt<<1|1]+=add[rt]*(m>>1);
        add[rt]=0;
    }
}

void build(int l,int r,int rt){
    M[rt]=add[rt]=0;
    if(l==r){
        return ;
    }
    int m=(l+r)>>1;
    build(lson);
    build(rson);
}

void update(int L,int R,int val,int l,int r,int rt){
    if(L<=l&&r<=R){
        M[rt]+=val;
        add[rt]+=val;
        return ;
    }
    pushdown(rt,r-l+1);
    int m=(l+r)>>1;
    if(L<=m) update(L,R,val,lson);
    if(R>m) update(L,R,val,rson);
    pushup(rt);
}
```

```
lld query(int L,int R,int l,int r,int rt){
    if(L<=l&&r<=R){
        return M[rt];
    }
    pushdown(rt,r-l+1);
    int m=(l+r)>>1;
    lld ret=0;
    if(L<=m) ret+=query(L,R,lson);
    if(R>m) ret+=query(L,R,rson);
    return ret;
}

void prepare()
{
    int i;
    build(1,n,1);
    memset(num,-1,sizeof(num));
    dep[0]=0;Seg_size=0;
    for(i=0;i<n;i++) fa[i]=i;
    dfs(0,0);
    for(i=0;i<n;i++)
    {
        if(heavy[i]==-1)
        {
            int pos=i;
            while(pos && edge[heavy[edge[rev[pos]].t]].t
== pos)
            {
                int t=rev[pos];

num[t]=num[t^1]=++Seg_size;//printf("pos=%d  val=%d
t=%d\n",Seg_size,edge[t].w,t);

update(Seg_size,Seg_size,edge[t].w,1,n,1);
                pos=edge[t].t;
            }
```

```
        }
    }
}

int lca(int u,int v)
{
    while(1)
    {
        int a=find(u),b=find(v);
        if(a==b) return dep[u] < dep[v] ? u : v;//a,b在同一条重链上
        else if(dep[a]>=dep[b]) u=edge[rev[a]].t;
        else v=edge[rev[b]].t;
    }
}

void CH(int u,int lca,int val)
{
    while(u!=lca)
    {
        int r=rev[u];//printf("r=%d\n",r);
        if(num[r]==-1) edge[r].w+=val,u=edge[r].t;
        else
        {
            int p=fa[u];
            if(dep[p] < dep[lca]) p=lca;
            int l=num[r];
            r=num[heavy[p]];
            update(l,r,val,1,n,1);
            u=p;
        }
    }
}

void change(int u,int v,int val)
{
```

```
    int p=lca(u,v);//    printf("p=%d\n",p);
    CH(u,p,val);
    CH(v,p,val);
    if(p){
        int r=rev[p];
       if(num[r]==-1) {
           edge[r^1].w+=val;//在此处发现了我代码的重大bug
           edge[r].w+=val;
       }
      else  update(num[r],num[r],val,1,n,1);
    }//根节点，特判
    else w[p]+=val;
}

lld solve(int u)
{
    if(!u) return w[u];//根节点，特判
    else
    {
        int r=rev[u];
        if(num[r]==-1) return edge[r].w;
        else return query(num[r],num[r],1,n,1);
    }
}

int main()
{
    int t,i,a,b,c,m,ca=1,p;
    while(scanf("%d%d%d",&n,&m,&p)!=EOF)
    {
        memset(head,-1,sizeof(head));
        E=0;
        for(i=0;i<n;i++) scanf("%d",&w[i]);
        for(i=0;i<m;i++)
        {
            scanf("%d%d",&a,&b);a--;b--;
```

```
            add_edge(a,b,0);
            add_edge(b,a,0);
        }
        prepare();
        char op[10];
        while(p--)
        {
            scanf("%s",&op);
            if(op[0]=='I')
            {
                scanf("%d%d%d",&a,&b,&c);a--;b--;
                change(a,b,c);
            }
            else if(op[0]=='D')
            {
                scanf("%d%d%d",&a,&b,&c);a--;b--;
                change(a,b,-c);
            }
            else
            {
                scanf("%d",&a);a--;
                printf("%d\n",solve(a));
            }
        }
    }
    return 0;
}
```

Link-Cut Tree

动态树(Link-Cut Tree)是一类要求维护森林的连通性的题的总称，这类问题要求维护某个点到根的某些数据，支持树的切分，合并，以及对子树的某些操作。其中解决这一问题的某些简化版（不包括对子树的操作）的基础数据结构就是LCT(link-cut tree)。

LCT的大体思想类似于树链剖分中的轻重链剖分,轻重链剖分是处理出重链来，

由于重链的定义和树链剖分是处理静态树所限，重链不会变化，变化的只是重链上的边或点的权值，由于这个性质，我们用线段树来维护树链剖分中的重链，但是LCT解决的是动态树问题（包含静态树），所以需要用更灵活的splay来维护这里的“重链”。

定义：

首先来定义一些量：

access(X)：表示访问X点（之后会有说明）。

Preferred child（偏爱子节点）：如果最后被访问的点在X的儿子P节点的子树中，那么称P为X的Preferred child，如果一个点被访问，他的Preferred child为null(即没有)。

Preferred edge（偏爱边）：每个点到自己的Preferred child的边被称为Preferred edge。

Preferred path（偏爱路径）：由Preferred edge组成的不可延伸的路径称为Preferred path。

这样我们可以发现一些比较显然的性质，每个点在且仅在一条Preferred path上，也就是所有的Preferred path包含了这棵树上的所有的点，这样一颗树就可以由一些Preferred path来表示（类似于轻重链剖分中的重链），我们用splay来维护每个条Preferred path，关键字为深度，也就是每棵splay中的点左子树的深度都比当前点小，右节点的深度都比当前节点的深度大。这样的每棵splay我们称为Auxiliary tree(辅助树)，每个Auxiliary tree的根节点保存这个Auxiliary tree与上一棵Auxiliary tree中的哪个点相连。这个点称作他的Path parent。

操作：

access(X)：首先由于preferred path的定义，如果一个点被访问，那么这个点到根节点的所有的边都会变成preferred edge，由于每个点只有一个preferred child，所以这个点到根节点路径上的所有的点都会和原来的preferred child断开，连接到这条新的preferred path上。假设访问X点，那么先将X点旋转到对应Auxiliary tree的根节点，然后因为被访问的点是没有preferred child的，所以将Auxiliary tree中根节点(X)与右子树的边断掉，左节点保留，将这个树的path parent旋转到对应Auxiliary tree的根节点，断掉右子树，连接这个点与X点，相当于合并两棵Auxiliary tree，不断地重复这一操作，直到当前X所在Auxiliary tree的path parent为null时停止，表示已经完成当前操作。

find root(x)：找到某一点所在树的根节点（维护森林时使用）。只需要access(X)，然后将X节点旋到对应Auxiliary tree的根节点，然后找到这个Auxiliary tree中最左面的点。

cut(x)：断掉X节点和其父节点相连的边。首先access(X)，然后将X旋转到对应Auxiliary tree的根节点，然后断掉Auxiliary tree中X和左节点相连的边。

link(join)(x,y)：连接点x到y点上。即让x称为y的子节点。因为x为y的子节点后，在原x的子树中，x点到根节点的所有的点的深度会被翻转过来，所以先access(x)，然后在对应的Auxiliary tree中将x旋转到根节点，，然后将左子树翻转(splay中的reverse操作)，然后access(y)，将y旋转到对应Auxiliary tree中的根节点，将x连到y就行了。

- hdu 2475 Box

  LCT的模板题，只需Link和Cut操作

```
#include<cstdio>
#include<cstring>
#include<iostream>
#define MAXN 50010
using namespace std;

int n;

struct LCT {
    int bef[MAXN], pre[MAXN], next[MAXN][2];

    void Init() {
        memset(pre, 0, sizeof(pre));
        memset(next, 0, sizeof(next));
    }

    inline void Rotate(int x, int kind) {
        int y, z;
        y = pre[x];
        z = pre[y];
        next[y][!kind] = next[x][kind];
        pre[next[x][kind]] = y;
        next[z][next[z][1] == y] = x;
        pre[x] = z;
        next[x][kind] = y;
        pre[y] = x;
    }
```

```
void Splay(int x) {
    int rt;
    for (rt = x; pre[rt]; rt = pre[rt])
        ;
    if (x != rt) {
        bef[x] = bef[rt];
        bef[rt] = 0;
        while (pre[x]) {
            if (next[pre[x]][0] == x)
                Rotate(x, 1);
            else
                Rotate(x, 0);
        }
    }
}

void Access(int x) {
    int father;
    for (father = 0; x; x = bef[x]) {
        Splay(x);
        pre[next[x][1]] = 0;
        bef[next[x][1]] = x;
        next[x][1] = father;
        pre[father] = x;
        bef[father] = 0;
        father = x;
    }
}

int Query(int x) {
    Access(x);
    Splay(x);
    while (next[x][0])
        x = next[x][0];
    return x;
```

```
    }

    void Cut(int x) {
        Access(x);
        Splay(x);
        bef[next[x][0]] = bef[x];
        bef[x] = 0;
        pre[next[x][0]] = 0;
        next[x][0] = 0;
    }

    void Join(int x, int y) {
        if (y == 0)
            Cut(x);
        else {
            int tmp;
            Access(y);
            Splay(y);
            for (tmp = x; pre[tmp]; tmp = pre[tmp])
                ;
            if (tmp != y) {
                Cut(x);
                bef[x] = y;
            }
        }
    }

} lct;
int INT() {
    char ch;
    int res;

    while (ch = getchar(), !isdigit(ch))
        ;

    for (res = ch - '0'; ch = getchar(), isdigit(ch);)
```

```
        res = res * 10 + ch - '0';

    return res;
}

char CHAR() {
    char ch, res;
    while (res = getchar(), !isalpha(res))
        ;
    while (ch = getchar(), isalpha(ch))
        ;
    return res;
}

int main() {
    bool flag = true;
    char ch;
    int i, x, y, q;
    while (~scanf("%d", &n)) {
        if (flag)
            flag = false;
        else
            putchar('\n');
        lct.Init();
        for (i = 1; i <= n; i++)
            lct.bef[i] = INT();
        q = INT();
        while (q--) {
            ch = CHAR();
            if (ch == 'Q') {
                x = INT();
                printf("%d\n", lct.Query(x));
            } else {
                x = INT(), y = INT();
                lct.Join(x, y);
            }
```

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

- poj 3237 Tree

题目大意：指定一颗树上有3个操作：询问操作,询问a点和b点之间的路径上最长的那条边的长度；取反操作，将a点和b点之间的路径权值都取相反数；变化操作，把某条边的权值变成指定的值。

```
#include<cstdio>
#include<cstring>
#include<queue>
#include<iostream>
#include<algorithm>
#define MAXN 100010
#define MAXM 200010
#define oo 0x7FFFFFFF
using namespace std;
bool vis[MAXN];
int first[MAXN], next[MAXM], v[MAXM], cost[MAXM], e;
struct LCT {
    bool neg[MAXN];
    int maxi[MAXN], mini[MAXN];
    int bef[MAXN], belong[MAXN];
    int next[MAXN][2], pre[MAXN], key[MAXN];
    void Init() {
        memset(next, 0, sizeof(next));
        memset(pre, 0, sizeof(pre));
        memset(neg, false, sizeof(neg));
        maxi[0] = -oo;
        mini[0] = oo;
    }
    inline void PushUp(int x) {
        maxi[x] = key[x];
```

```
        if (next[x][0])
            maxi[x] = max(maxi[x], maxi[next[x][0]]);
        if (next[x][1])
            maxi[x] = max(maxi[x], maxi[next[x][1]]);
        mini[x] = key[x];
        if (next[x][0])
            mini[x] = min(mini[x], mini[next[x][0]]);
        if (next[x][1])
            mini[x] = min(mini[x], mini[next[x][1]]);
    }
    inline void Not(int x) {
        if (x) {
            neg[x] ^= true;
            swap(maxi[x], mini[x]);
            key[x] = -key[x];
            maxi[x] = -maxi[x];
            mini[x] = -mini[x];
        }
    }
    inline void PushDown(int x) {
        if (neg[x]) {
            Not(next[x][0]);
            Not(next[x][1]);
            neg[x] = false;
        }
    }
    void Rotate(int x, int kind) {
        int y, z;
        y = pre[x];
        z = pre[y];
        PushDown(y);
        PushDown(x);
        next[y][!kind] = next[x][kind];
        pre[next[x][kind]] = y;
        next[z][next[z][1] == y] = x;
        pre[x] = z;
```

```
        next[x][kind] = y;
        pre[y] = x;
        PushUp(y);
    }
    void Splay(int x) {
        int rt;
        for (rt = x; pre[rt]; rt = pre[rt])
            ;
        if (rt != x) {
            bef[x] = bef[rt];
            bef[rt] = 0;
            PushDown(x);
            while (pre[x]) {
                if (next[pre[x]][0] == x)
                    Rotate(x, 1);
                else
                    Rotate(x, 0);
            }
            PushUp(x);
        }
    }
    void Access(int x) {
        int father;
        for (father = 0; x; x = bef[x]) {
            Splay(x);
            PushDown(x);
            pre[next[x][1]] = 0;
            bef[next[x][1]] = x;
            next[x][1] = father;
            pre[father] = x;
            bef[father] = 0;
            father = x;
            PushUp(x);
        }
    }
    void Change(int x, int y) {
```

```
        int t;
        t = belong[x - 1];
        Splay(t);
        key[t] = y;
    }
    void Negate(int x, int y) {
        Access(y);
        for (y = 0; x; x = bef[x]) {
            Splay(x);
            if (!bef[x]) {
                Not(y);
                Not(next[x][1]);
                return;
            }
            PushDown(x);
            pre[next[x][1]] = 0;
            bef[next[x][1]] = x;
            next[x][1] = y;
            pre[y] = x;
            bef[y] = 0;
            y = x;
            PushUp(x);
        }
    }
    int Query(int x, int y) {
        Access(y);
        for (y = 0; x; x = bef[x]) {
            Splay(x);
            if (!bef[x])
                return max(maxi[y], maxi[next[x][1]]);
            PushDown(x);
            pre[next[x][1]] = 0;
            bef[next[x][1]] = x;
            next[x][1] = y;
            pre[y] = x;
            bef[y] = 0;
```

```
            y = x;
            PushUp(x);
        }
        return 0;
    }
} lct;
int INT() {
    char ch;
    int res;
    bool neg;
    while (ch = getchar(), !isdigit(ch) && ch != '-')
        ;
    if (ch == '-') {
        res = 0;
        neg = true;
    } else {
        res = ch - '0';
        neg = false;
    }
    while (ch = getchar(), isdigit(ch))
        res = res * 10 + ch - '0';
    return neg ? -res : res;
}
char CHAR() {
    char ch, res;
    while (res = getchar(), !isalpha(res))
        ;
    while (ch = getchar(), isalpha(ch))
        ;
    return res;
}
inline void AddEdge(int x, int y, int val) {
    v[e] = y;
    cost[e] = val;
    next[e] = first[x];
    first[x] = e++;
```

```
}
void BFS(int x) {
    int i, y;
    queue<int> q;
    memset(vis, false, sizeof(vis));
    vis[x] = true;
    q.push(x);
    while (!q.empty()) {
        x = q.front();
        q.pop();
        for (i = first[x]; i != -1; i = next[i]) {
            y = v[i];
            if (!vis[y]) {
                lct.bef[y] = x;
                lct.key[y] = cost[i];
                lct.belong[i >> 1] = y;
                vis[y] = true;
                q.push(y);
            }
        }
    }
}
int main() {
    int c, i;
    char ch;
    int n, x, y, val;
    c = INT();
    while (c--) {
        n = INT();
        lct.Init();
        memset(first, -1, sizeof(first));
        for (e = 0, i = 1; i < n; i++) {
            x = INT(), y = INT(), val = INT();
            AddEdge(x, y, val);
            AddEdge(y, x, val);
        }
```

```
        BFS(1);
        while (ch = CHAR(), ch != 'D') {
            x = INT(), y = INT();
            if (ch == 'Q')
                printf("%d\n", lct.Query(x, y));
            else if (ch == 'C')
                lct.Change(x, y);
            else
                lct.Negate(x, y);
        }
    }
    return 0;
}
```

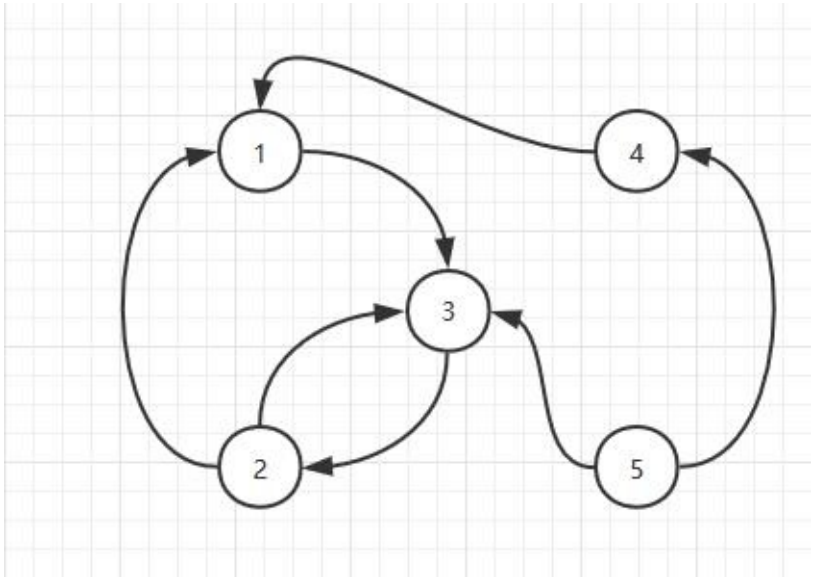
图论

---

author：高云峰

强连通分量

有向图强连通分量：在有向图中有一些点，这些点中如果能从A到B，那么一定从B能到达A，这些点组成的点数最多的子图是原图的一个强连通分量。 运用场合：有向图、有两两可达这种条件、往往通过把每个强连通分量缩点把原图化简成一棵树



如图：

- 1,2,3号点构成了一个强连通分量，它们可以互相到达
- 4号点自己构成了一个强连通分量，5号点自己构成了一个强连通分量
- 所以图中一共有3个强连通分量
- sccno[1] = 1 sccno[2] = 1 sccno[3] = 1 sccno[4] = 2 sccno[3] = 3
- sccno[i]代表第i个点所在的强连通分量的编号

模板代码：

```
void dfs(int u) { //从u点开始找一个强连通分量
    pre[u] = lowlink[u] = ++dfs_clock;
    S.push(u);
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        if (!pre[v]) {
            dfs(v);
            if (lowlink[u] > lowlink[v]) lowlink[u] =
lowlink[v];
        } else if (!sccno[v]) {
            if (lowlink[u] > pre[v]) lowlink[u] = pre[v];
        }
    }
    if (lowlink[u] == pre[u]) {
        scc_cnt++;
        for(;;) {
            int x = S.top(); S.pop();
            sccno[x] = scc_cnt;
            if (x == u) break;
        }
    }
}
void find_scc(int n) { //n是点的总数，点的范围1-n
    dfs_clock = scc_cnt = 0;
    MEM(sccno); MEM(pre); MEM(ans);
    for (int i = 1; i <= n; i++) {
        if (!pre[i]) dfs(i);
    }
}
```

POJ 3180 The Cow Prom

有N头奶牛，和M个绳索。每个绳索从一头奶牛处发出，射向另一头奶牛。现在奶牛们想要跳舞，如果沿着射向它的绳索往回找，找到另一头牛，并如此找下去，最后能找到自己发出的绳索，那么这头奶牛就跳舞成功了，并且在寻找过程中，串起

来的这些奶牛属于同一个集合。问最后这M个绳索连出了几个集合？
先明确什么是同一个集合，A->B->C->D->E->A这样A,B,C,D,E就是同一个集合，如果还有C->F->C，那么F也属于上面那个集合，所以这题就是求点数大于2的强连通分量的个数。
大白书的模板（Tarjan）中用了一个sccno[]数组，sccno[i]记录第i个点所属的强连通分量号然后我们只需要用一个cnt[]数组，记录每个块中点的个数，如果cnt[i] > 1，说明第i个块是符合条件的，答案数目加1，最后输出答案数。

```
#include <cstdio>
#include <cstring>
#include <stack>
#include <vector>
using namespace std;

#define MEM(a) memset(a, 0, sizeof(a))
const int maxv = 21000;
vector<int>G[maxv];
int pre[maxv], lowlink[maxv], sccno[maxv], ans[maxv],
dfs_clock, scc_cnt;
stack<int>S;

void dfs(int u) {
    pre[u] = lowlink[u] = ++dfs_clock;
    S.push(u);
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        if (!pre[v]) {
            dfs(v);
            if (lowlink[u] > lowlink[v]) lowlink[u] =
lowlink[v];
        } else if (!sccno[v]) {
            if (lowlink[u] > pre[v]) lowlink[u] = pre[v];
        }
    }
    if (lowlink[u] == pre[u]) {
        scc_cnt++;
```

```
        for(;;) {
            int x = S.top(); S.pop();
            sccno[x] = scc_cnt;
            if (x == u) break;
        }
    }
}
void find_scc(int n) {
    dfs_clock = scc_cnt = 0;
    MEM(sccno); MEM(pre); MEM(ans);
    for (int i = 1; i <= n; i++) {
        if (!pre[i]) dfs(i);
    }
}

int main() {
    int n, m, u, v;
    while (scanf("%d%d", &n, &m) != EOF) {
        for (int i = 1; i <= n; i++) G[i].clear();
        for (int i = 0; i < m; i++) {
            scanf("%d%d", &u, &v);
            G[u].push_back(v);
        }
        find_scc(n);
        for (int u = 1; u <= n; u++) {
            ans[sccno[u]]++;
        }
        int cnt = 0;
        for (int i = 1; i <= scc_cnt; i++)
            if (ans[i] > 1) cnt++;
        printf("%d\n", cnt);
    }
    return 0;
}
```

HDU 3816 The King's Problem

有一个n个点m条边的有向图，把这个图分成几个区域，使得每个区域中的任意两点u, v要么u能到v，要么v能到u，求最少要分成几个区域

缩点得到有向的树，即求这棵树的最少路径覆盖(点数 – 二分图的最大匹配)

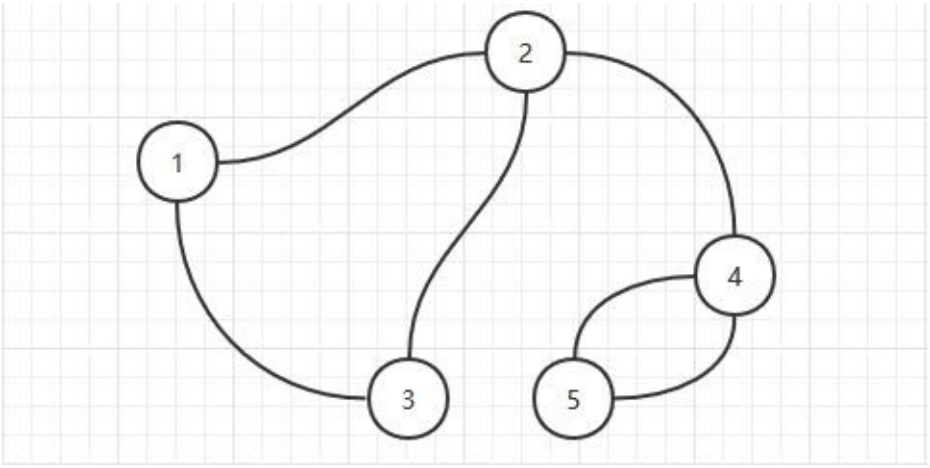
```
#include <cstdio>
#include <cstring>
#include <stack>
#include <vector>
using namespace std;

#define MEM(a) memset(a, 0, sizeof(a))
const int maxv = 15100;
vector<int>G[maxv], H[maxv];
int pre[maxv], lowlink[maxv], sccno[maxv], left[maxv],
dfs_clock, scc_cnt;
bool vis[maxv];
stack<int>S;

void dfs(int u) {
    pre[u] = lowlink[u] = ++dfs_clock;
    S.push(u);
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        if (!pre[v]) {
            dfs(v);
            if (lowlink[u] > lowlink[v]) lowlink[u] =
lowlink[v];
        } else if (!sccno[v]) {
            if (lowlink[u] > pre[v]) lowlink[u] = pre[v];
        }
    }
    if (lowlink[u] == pre[u]) {
        scc_cnt++;
        for(;;) {
```

```
            int x = S.top(); S.pop();
            sccno[x] = scc_cnt;
            if (x == u) break;
        }
    }
}

void find_scc(int n) {
    dfs_clock = scc_cnt = 0;
    MEM(sccno); MEM(pre);
    for (int i = 1; i <= n; i++) {
        if (!pre[i]) dfs(i);
    }
}

bool match(int u) {
    for (int i = 0; i <(int)H[u].size(); i++) {
        int v = H[u][i];
        if (!vis[v]) {
            vis[v] = 1;
            if (left[v] == -1 || match(left[v])) {
                left[v] = u;
                return true;
            }
        }
    }
    return false;
}

int main() {
    int n, m, u, v, t;
    scanf("%d", &t);
    while (t--) {
        scanf("%d%d", &n, &m);
        for (int i = 1; i <= n; i++) {
            G[i].clear();
```

```
            H[i].clear();
        }
        for (int i = 0; i < m; i++) {
            scanf("%d%d", &u, &v);
            G[u].push_back(v);
        }
        find_scc(n);
        for (int u = 1; u <= n; u++) {
            for (int i = 0; i < (int)G[u].size(); i++) {
                int v = G[u][i];
                if (sccno[u] != sccno[v])
                    H[sccno[u]].push_back(sccno[v]);
            }
        }
        int sum = 0;
        memset(left, -1, sizeof(left));
        for (int i = 1; i <= scc_cnt; i++) {
            MEM(vis);
            if (match(i)) sum++;
        }
        printf("%d\n", scc_cnt-sum);
    }
    return 0;
}
```

双连通分量

无向图中的双连通分量有两种：点双连通分量和边的连联通分量，如果去掉任意一个点之后，这个图还是连通的，就说这个图是点双连通的。如果去掉任意一条边之后这个图还是连通的，就说明这个图是边双连通的。



如图：

- 1，2，3构成了一个边双连通分量，切断1，2，3中的任意一条边它们还连着
- 4，5构成了一个边双连通分量，切断4，5中的任意一条边它们还连着
- 1，2，3构成了一个点双连通分量，去掉1，2，3中的任意一个点它们还连着

模板代码

```
void dfs(int u, int fa) {
    low[u] = pre[u] = ++dfs_clock;
    stakk[top++] = u;
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];

        if (!pre[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
        } else if (pre[v] < pre[u] && v != fa) {
            low[u] = min(low[u], pre[v]);
        }
    }
    if (pre[u] == low[u]) {
        while (top > 0 && stakk[top] != u) {
            low[stakk[--top]] = low[u];
        }
    }
}

void find_bcc(int n) {
    MEM(pre); MEM(low);MEM(deg);
    dfs_clock = top = 0;
    for (int i = 1; i <= n; i++)
        if (!pre[i]) dfs(i, -1);
}
```

POJ 3352 Road Construction

给出一个没有重边的无向图，求至少加入几条边使整个图成为一个边双连通分量

把图中所有的边双连通分量缩成一个点，原图就缩成了一棵树，要加的边数就是(所有度为1的点的个数 + 1)/2

```
#include <cstdio>
#include <cstring>
```

```cpp
#include <vector>
using namespace std;

#define MEM(a) memset(a, 0, sizeof(a))
#define pb push_back
const int maxv = 1010;

int pre[maxv], low[maxv], deg[maxv], stakk[maxv];
int dfs_clock, top;
vector<int> G[maxv];

void dfs(int u, int fa) {
    low[u] = pre[u] = ++dfs_clock;
    stakk[top++] = u;
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];

        if (!pre[v]) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
        } else if (pre[v] < pre[u] && v != fa) {
            low[u] = min(low[u], pre[v]);
        }
    }
    if (pre[u] == low[u]) {
        while (top > 0 && stakk[top] != u) {
            low[stakk[--top]] = low[u];
        }
    }
}

void find_bcc(int n) {
    MEM(pre); MEM(low);MEM(deg);
    dfs_clock = top = 0;
    for (int i = 1; i <= n; i++)
        if (!pre[i]) dfs(i, -1);
```

```
}
int main() {
    int n, m, u, v;
    while (scanf("%d%d", &n, &m) != EOF) {
        for (int i = 1; i <= n; i++) G[i].clear();

        for (int i = 0; i < m; i++) {
            scanf("%d%d", &u, &v);
            G[u].pb(v); G[v].pb(u);
        }
        find_bcc(n);
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < (int)G[i].size(); j++) {
                if (low[i] != low[G[i][j]]) {
                    deg[low[i]]++;
                    deg[low[G[i][j]]]++;
                }
            }
        }
        for (int i = 1; i <= n; i++)
            if (deg[i]/2 == 1) ans++;
        printf("%d\n", (ans+1)/2);
    }
    return 0;
}
```

LA 3523 Knights of the Round Table

亚瑟王要给一些骑士开会，但是这些骑士中有一些相互憎恨，所以他们不能在圆桌中相邻，为了投票不出现支持的人数和反对的人数相等的情况，每个圆桌中的骑士的个数必须为奇数个，问有多少个骑士一定不能参加任何一个会议。

因为可能要开好几桌会议，所以要求出图中所有的连通分量，又因为要求骑士的个数为奇数个，所以求每个联通分量中不在奇圈上的点的个数的和

```
#include <cstdio>
#include <stack>
#include <cstring>
#include <vector>
using namespace std;

#define MEM(a) memset(a, 0, sizeof(a))
const int maxv = 1100;
const int maxe = maxv*maxv;

vector<int>G[maxv], bcc[maxv];
int dfs_clock, bcc_cnt, pre[maxv], low[maxv],
bccno[maxv];
int color[maxv], odd[maxv], mapp[maxv][maxv];
struct Edge{
    int u, v;
    Edge(){}
    Edge(int a, int b) {
        u = a;
        v = b;
    }
};
stack<Edge> S;

bool bipartite (int u, int b) {
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        if (bccno[v] != b) continue;
        if (color[v] == color[u]) return false;
        if (!color[v]) {
            color[v] = 3 - color[u];
            if (!bipartite(v, b)) return false;
        }
    }
    return true;
```

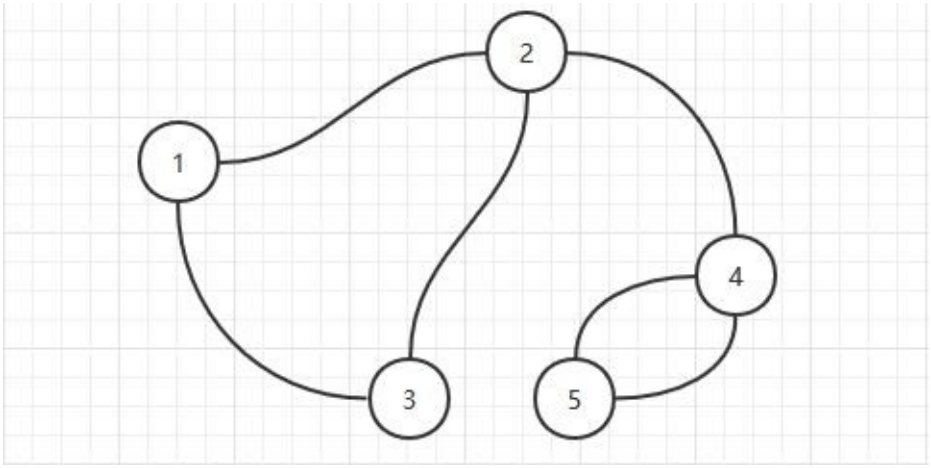
```
}

void tarjan(int u, int fa) {
    pre[u] = low[u] = ++dfs_clock;
    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        if (!pre[v]) {
            S.push(Edge(u, v));
            tarjan(v, u);
            low[u] = min(pre[v], low[u]);
            if (low[v] >= pre[u]) {
                bcc_cnt++;
                bcc[bcc_cnt].clear();
                for(;;) {
                    Edge x = S.top(); S.pop();
                    if (bccno[x.u] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(x.u);
                        bccno[x.u] = bcc_cnt;
                    }
                    if (bccno[x.v] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(x.v);
                        bccno[x.v] = bcc_cnt;
                    }
                    if (x.u == u && x.v == v) break;
                }
            }
        }
        else if (pre[v] < pre[u] && v != fa) {
            S.push(Edge(u, v));
            low[u] = min(low[u], pre[v]);
        }
    }
}

void find_bcc(int n) {
    dfs_clock = bcc_cnt = 0;
```

```
    MEM(pre); MEM(low);
    for (int i = 1; i <= n; i++)
        if (!pre[i]) tarjan(i, -1);
}
int main() {
    int n, m, u, v;
    while (scanf("%d%d", &n, &m) != EOF &&m && n) {
        MEM(mapp);
        for (int i = 1; i <= n; i++) G[i].clear();
        for (int i = 0; i < m; i++) {
            scanf("%d%d", &u, &v);
            mapp[u][v] = mapp[u][v] = 1;
        }
        for (int u = 1; u <= n; u++) {
            for (int v = u+1; v <= n; v++) {
                if (!mapp[u][v]) {
                    G[u].push_back(v);
                    G[v].push_back(u);
                }
            }
        }
        find_bcc(n);

        MEM(odd);
        for (int i = 1; i <= bcc_cnt; i++) { //对于图中的每
个点双连通分量
            MEM(color);
            for (int j = 0; j < (int)bcc[i].size(); j++)
bccno[bcc[i][j]] = i;
            //给双连通分量里的每个点标号
            int u = bcc[i][0]; // 找到代表的点
            color[u] = 1;
            if (!bipartite(u, i))   //判断如果这个双连通分量
不是二分图，那么这个连通分量里的点都合格
                for (int j = 0; j < (int)bcc[i].size();
j++) odd[bcc[i][j]] = 1;
```

```
        }
        int ans = n;
        for (int i = 1; i <= n; i++) if (odd[i]) ans--;
        printf("%d\n", ans);
    }
    return 0;
}
```

割点和桥

在一个无向图中，如果删去一个点，这个图的连通度增加了，删去的这个点就是原图的一个割点；如果删掉一条边，这个图的连通度增加了，删掉的这条边就是原图的一个桥。



如图：

- 2，4号点是割点，去掉之后整个图分成了两个
- 2—4之间的边是桥，去掉之后整个图分成了两个

Hdu 3394.Railway

在一个有n个点，m条边的无向图中，如果某条边不在任何一个回路中，则称这条边是无用的如果某条边被多个回路利用，则称这条边是冲突的，求这个图中的冲突的和无用的边的条数

很明显，图中无用的边就是桥，而冲突的边呢？在每个块中，如果边数大于点数，那么这个块中的每条边都是冲突边。

```
#include <cstdio>
#include <cstring>
#include <vector>
#include <stack>
using namespace std;

#define MEM(a) memset(a, 0, sizeof(a))
#define pb push_back
const int maxv = 10000;
```

```
const int maxe = 100000;

struct Edge {
    int u, v;
    Edge () {}
    Edge (int a, int b ) {
        u = a;
        v = b;
    }
};

int pre[maxv], low[maxv], iscut[maxv], bccno[maxv];
int dfs_clock, bcc_cnt, qiao, ans2;
vector<int> G[maxe], bcc[maxv];
stack<Edge> S;

void dfs(int u, int fa) {
    low[u] = pre[u] = ++dfs_clock;

    for (int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];

        if (!pre[v]) {
            S.push(Edge(u,v));
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= pre[u]) {
                if (low[v] > pre[u]) qiao++;
                bcc_cnt++; bcc[bcc_cnt].clear();
                for(;;) {
                    Edge x = S.top(); S.pop();
                    if (bccno[x.u] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(x.u);
                        bccno[x.u] = bcc_cnt;
                    }
                    if (bccno[x.v] != bcc_cnt) {
```

```
                        bcc[bcc_cnt].push_back(x.v);
                        bccno[x.v] = bcc_cnt;
                    }
                    if (x.u == u && x.v == v) break;
                }
                int cnt2 = 0;
                for (int j = 0; j <
(int)bcc[bcc_cnt].size(); j++) {
                    int k = bcc[bcc_cnt][j];
                    for (int h = 0; h < (int)G[k].size();
h++) {
                        int vv = G[k][h];
                        if (bccno[vv] == bcc_cnt) cnt2++;
                    }
                }
                if (cnt2/2 > (int)bcc[bcc_cnt].size())
ans2 += cnt2/2;
                cnt2 = 0;
            }
        } else if (pre[v] < pre[u] && v != fa) {
            S.push(Edge(u, v));
            low[u] = min(low[u], pre[v]);
        }
    }
}

void find_bcc(int n) {
    MEM(pre); MEM(iscut); MEM(bccno); MEM(low);
    dfs_clock = bcc_cnt = qiao = ans2 = 0;
    for (int i = 0; i < n; i++)
        if (!pre[i]) dfs(i, -1);
    for (int i = 0; i <= bcc_cnt; i++)
        bcc[i].clear();
}
int main() {
    //freopen("in.txt", "r", stdin);
```

```
    int n, m, u, v;
    while (scanf("%d%d", &n, &m) != EOF) {
        if (!n && !m) break;
        for (int i = 0; i < n; i++) G[i].clear();

        for (int i = 0; i < m; i++) {
            scanf("%d%d", &u, &v);
            G[u].pb(v); G[v].pb(u);
        }
        find_bcc(n);
        printf("%d %d\n", qiao, ans2);
    }
    return 0;
}
```

拓扑排序

给出一些大小关系，然后求出一个总的大小关系。比如：甲>丙， 丙>乙，可以的到一个大小关系：甲 > 丙 > 乙

模板代码：

```
void toporder(int n) {
    queue<int>q;
    for (int i = 0; i < n; i++) {
        if (!deg[i]) q.push(i);
        // deg[i] 表示第i个点的度数，这里先把度数为0的点加入队列
    }
    while (!q.empty()) {
        int u = q.front(); q.pop();
        // u是排到的点，这里根据情况写
        for (int i = 0; i < (int)G[u].size(); i++) {
            int v = G[u][i];
            if (!--deg[v]) {
                q.push(v);
            }
        }
    }
}
```

HRBUST 1631 技能修炼

输出拓扑排序的顺序

```
#include <cstdio>
#include <queue>
#include <cstring>
using namespace std;
#define MEM(a) memset(a,0,sizeof(a))
const int N = 510;
```

```
int digree[N];
int path[N];
int G[N][N];

int main() {
    int n, m;
    while (scanf("%d%d", &n, &m) != EOF) {
        int u, v;
        MEM(G),MEM(path),MEM(digree);
        for (int i = 1; i <= m; i++) {
            scanf("%d%d", &u, &v);
            G[u][v] = 1;
            digree[v]++;
        }
        priority_queue<int, vector<int>, greater<int> >q;
        for (int i = 1; i <= n; i++) {
            if (!digree[i]) q.push(i);
        }
        int l = 0;
        while (!q.empty()) {
            int cur = q.top();
            path[l++] = cur;
            q.pop();
            for (int i = 1; i <= n; i++) {
                if(G[cur][i]) {
                    digree[i]--;
                    if (!digree[i])
                        q.push(i);
                }
            }
        }
        for (int i = 0; i < l; i++) {
            printf("%d", path[i]);
            if (i != l-1) putchar(' ');
        }
        puts("");
```

```
    }
    return 0;
}
```

HDU 1811 Rank of Tetris

每组测试第一行包含两个整数N,M(0<=N<=10000,0<=M<=20000),分别表示要排名的人数以及得到的关系数。接下来有M行，分别表示这些关系，问能否确定唯一排序关系。

如果一次入队入度为零的点大于1则说明拓扑排序序列不唯一
如果排序的总个数小于给定的个数,则说明存在回路

```
#include <cstdio>
#include <cstring>
#include <vector>
#include <queue>
using namespace std;

const int MAXN = 10000 + 5;

int fa[MAXN];
vector<int>G[MAXN];
int deg[MAXN];
int num;
bool cert;

void init(int n) {
    num = n; cert = true;
    for (int i = 0; i <= n; i++) {
        fa[i] = i;
        G[i].clear();
        deg[i] = 0;
    }
}
```

```
int findfa(int u) {
    if (fa[u] != u) fa[u] = findfa(fa[u]);
    return fa[u];
}

void unin(int u, int v) {
    int fau = findfa(u);
    int fav = findfa(v);
    if (fau == fav) return;
    if (fau < fav) {
        fa[fav] = fau;
    } else {
        fa[fau] = fav;
    }
    num--;
}

void toporder(int n) {
    queue<int>q;
    for (int i = 0; i < n; i++) {
        if (!deg[i] && fa[i] == i) q.push(i);
    }
    while (!q.empty()) {
        if (q.size() > 1) cert = false;
        int u = q.front(); q.pop();
        num--;
        for (int i = 0; i < (int)G[u].size(); i++) {
            int v = G[u][i];
            if (!--deg[v]) {
                q.push(v);
            }
        }
    }
}
```

```
char r[MAXN][10];

int u[MAXN], v[MAXN];

int main() {
    //freopen("in.txt", "r", stdin);
    int n, m;
    while (scanf("%d%d", &n, &m) != EOF) {
        init(n);
        for (int i = 0; i < m; i++) {
            scanf("%d%s%d", &u[i], r[i], &v[i]);
            if (r[i][0] == '=') {
                unin(u[i], v[i]);
            }
        }
        for (int i = 0; i < m; i++) {
            int uu = fa[u[i]], vv = fa[v[i]];
            if (r[i][0] == '<') {
                G[vv].push_back(uu);
                deg[uu]++;
            } else if (r[i][0] == '>') {
                G[uu].push_back(vv);
                deg[vv]++;
            }
        }
        toporder(n);
        if (num > 0) puts("CONFLICT");
        else if (!cert) puts("UNCERTAIN");
        else puts("OK");
    }
    return 0;
}
```

最短路 Dijkstra

求一个图中两个点之间最短距离的最快的一种方法，要求路径的花费不能有负的。
时间复杂度：O(ElogE)

模板代码

```
struct Edge {
    int from, to, dist;
};
struct HeapNode {
    int d, u;
    bool operator  < (const HeapNode & rhs) const {
        return d > rhs.d;
    }
};

struct Dijkstra {
    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn];
    bool done[maxn];
    int d[maxn];
    int p[maxn];

    void init(int n) {
        this -> n = n;
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }
    void AddEdge(int from, int to, int dist) {
        edges.push_back((Edge){from, to, dist});
        m = edges.size();
        G[from].push_back(m-1);
    }
```

```
    void dijkstra(int s) {
        priority_queue<HeapNode> Q;
        for (int i = 0; i < n; i++) d[i] = INF;
        d[s] = 0;
        memset(done, 0, sizeof(done));
        Q.push((HeapNode){0, s});
        while (!Q.empty()) {
            HeapNode x = Q.top(); Q.pop();
            int u = x.u;
            if (done[u]) continue;
            done[u] = true;
            for (int i = 0; i < (int)G[u].size(); i++) {
                Edge &e = edges[G[u][i]];
                if (d[e.to] > d[u] + e.dist && d[u] +
e.dist <= c) {
                    d[e.to] = d[u] + e.dist;
                    if (pp[e.to]) d[e.to] = 0;
                    p[e.to] = G[u][i];
                    Q.push((HeapNode){d[e.to], e.to});
                }
            }
        }

    }
}
```

ZOJ 3794 Greedy Driver

在n个点m条边的有向图中，一个人有一个车，油箱容量为C。有p个城市可以用加油卡加任意多的油，有q个城市可以卖油，每单位的价格为q_i，在这个人有无数张加油卡的情况下，从1加满油跑到n，只卖一次油，最多可以卖出多少钱的油？

从1跑一次最短路，d[i]表示到i点最少消耗的油量，如果i点是加油站，更新d[i] = 0。left[i] = C – d[i]表示到达第i个点最多剩下多少单位的油，从n跑一次最短路，cost[i]代表从n到每个点的最小花费，剩的油的最大值就是max(p[i] * (left[i] – cost[i]))，i为每个可以卖油的城市。(松弛操作的时候注意油箱的容量限制)

```cpp
#include <cstdio>
#include <cstring>
#include <vector>
#include <queue>
using namespace std;

const int maxn = 1500;
const long long INF = 0x3f3f3f3f;
struct Edge {
    int from, to, dist;
};
struct HeapNode {
    int d, u;
    bool operator  < (const HeapNode & rhs) const {
        return d > rhs.d;
    }
};

int pp[maxn], c;
struct Dijkstra {
    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn];
    bool done[maxn];
    int d[maxn];
    int p[maxn];

    void init(int n) {
        this -> n = n;
        for (int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }
    void AddEdge(int from, int to, int dist) {
        edges.push_back((Edge){from, to, dist});
        m = edges.size();
```

```
        G[from].push_back(m-1);
    }
    void dijkstra(int s) {
        priority_queue<HeapNode> Q;
        for (int i = 0; i < n; i++) d[i] = INF;
        d[s] = 0;
        memset(done, 0, sizeof(done));
        Q.push((HeapNode){0, s});
        while (!Q.empty()) {
            HeapNode x = Q.top(); Q.pop();
            int u = x.u;
            if (done[u]) continue;
            done[u] = true;
            for (int i = 0; i < (int)G[u].size(); i++) {
                Edge &e = edges[G[u][i]];
                if (d[e.to] > d[u] + e.dist && d[u] +
e.dist <= c) {
                    d[e.to] = d[u] + e.dist;
                    if (pp[e.to]) d[e.to] = 0;
                    p[e.to] = G[u][i];
                    Q.push((HeapNode){d[e.to], e.to});
                }
            }
        }

    }
}G, H;

int q[maxn], left[maxn], city[maxn];

int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, x, y;
    while (scanf("%d%d%d", &n, &m, &c) != EOF) {
        H.init(n+1);
        G.init(n+1);
```

```
        for (int i = 0; i < m; i++) {
            int u, v, w;
            scanf("%d%d%d", &u, &v, &w);
            H.AddEdge(u, v, w);
            G.AddEdge(v, u, w);
        }
        memset(pp, 0, sizeof(pp));
        memset(q, 0, sizeof(q));
        scanf("%d", &x);
        for (int i = 0; i < x; i++) {
            scanf("%d", &y);
            pp[y] = 1;
        }
        scanf("%d", &x);
        for (int i = 0; i < x; i++) {
            scanf("%d%d", &city[i], &q[i]);

        }
        H.dijkstra(1);
        if (H.d[n] == INF) {
            puts("-1");
            continue;
        }
        G.dijkstra(n);
        for (int i = 0; i < x; i++) left[city[i]] = c -
H.d[city[i]];

        int ans = 0;
        for (int i = 0; i < x; i++)
            if(q[i] > 0 && left[city[i]] > 0 &&
left[city[i]] - G.d[city[i]] > 0)
                ans = max(ans, (left[city[i]] -
G.d[city[i]]) * q[i]);
        printf("%d\n", ans);
    }
    return 0;
```

```
}
```

最短路 SPFA

国人发明出来的一种计算单源最短路的方法，算是BF算法的一种简化版本，代码量较小，时间复杂度无法证明。因为代码量较小，速度又不错，经常与其他算法组合出现，可以用来判断负环。

模板代码：

```
struct edge{
    int from, to, cost;
    edge() {}
    edge(int _from, int _to, int _cost) {
        from = _from;
        to = _to;
        cost = _cost;
    }
};

vector<int>G[maxv];
vector<edge> edges;
int rank[maxv], dis[maxv];
bool inque[maxv];

void add(int u, int v, int w) {
    edges.push_back(edge(u, v, w));
    int m = edges.size();
    G[u].push_back(m-1);
}

int spfa(int s, int n) {
    for (int i = 0; i <= n; i++) {
        dis[i] = INF;
        rank[i] = 0;
        inque[i] = false;
    }
```

```
    dis[s] = 0;
    rank[s] = 1;
    inque[s] = true;
    queue<int>que;
    que.push(s);
    while (!que.empty()) {
        int u = que.front();
        inque[u] = false;
        que.pop();
        for (int i = 0; i < (int)G[u].size(); i++) {
            edge e = edges[G[u][i]];
            if (dis[e.to] > dis[u] + e.cost) {
                dis[e.to] = dis[u] + e.cost;
                if (!inque[e.to]) {
                    que.push(e.to);
                    inque[e.to] = true;
                    rank[e.to]++;
                    if (rank[e.to] >= n) return false;
                }
            }
        }
    }
    return true;
}
```

POJ 3259 Wormholes

有两种路，一种走完这条路需要的时间是正的，另一种需要的时间是负的，问有没有这样一条回路，走完整条回路后，需要的时间的和是负的(判负环)

判断每个点的入队次数，如果大于N（图中总的点数），就是有负环

```
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;
```

```
const int maxv = 1000;
const int INF = 1000000000;

struct edge{
    int from, to, cost;
    edge() {}
    edge(int _from, int _to, int _cost) {
        from = _from;
        to = _to;
        cost = _cost;
    }
};

vector<int>G[maxv];
vector<edge> edges;
int rank[maxv], dis[maxv];
bool inque[maxv];

void add(int u, int v, int w) {
    edges.push_back(edge(u, v, w));
    int m = edges.size();
    G[u].push_back(m-1);
}

int spfa(int s, int n) {
    for (int i = 0; i <= n; i++) {
        dis[i] = INF;
        rank[i] = 0;
        inque[i] = false;
    }
    dis[s] = 0;
    rank[s] = 1;
    inque[s] = true;
    queue<int>que;
    que.push(s);
```

```
    while (!que.empty()) {
        int u = que.front();
        inque[u] = false;
        que.pop();
        for (int i = 0; i < (int)G[u].size(); i++) {
            edge e = edges[G[u][i]];
            if (dis[e.to] > dis[u] + e.cost) {
                dis[e.to] = dis[u] + e.cost;
                if (!inque[e.to]) {
                    que.push(e.to);
                    inque[e.to] = true;
                    rank[e.to]++;
                    if (rank[e.to] >= n) return false;
                }
            }
        }
    }
    return true;
}

int main() {
    //freopen("in.txt", "r", stdin);
    int t, n, m, W, u, v, w;
    scanf("%d", &t);
    while (t--) {
        scanf("%d%d%d", &n, &m, &W);
        for (int i = 0; i <= n; i++) G[i].clear();
        edges.clear();
        for (int i = 0; i < m; i++) {
            scanf("%d%d%d", &u, &v, &w);
            add(u, v, w);
            add(v, u, w);
        }
        for (int i = 0; i < W; i++) {
            scanf("%d%d%d", &u, &v, &w);
            add(u, v, -w);
```

```
        }
        for (int i = 1; i <= n; i++) {
            add(n+1, i, 0);
        }
        if (spfa(n+1, n+1)) {
            puts("NO");
        } else {
            puts("YES");
        }
    }
    return 0;
}
```

HDU 4460 Friend Chains

求一个无向图中的最长的最短路
枚举以每个点为起点的最短路，维护最大值(就是个暴力)

```
#include <cstdio>
#include <cstring>
#include <string>
#include <map>
#include <queue>
#include <vector>
#include <algorithm>
using namespace std;

const int maxn = 1000 + 5;
const double INF = 1000000000;

struct edge{
    int from, to, cost;
    edge() {}
    edge(int _from, int _to, int _cost) {
        from = _from;
        to = _to;
```

```
        cost = _cost;
    }
};

vector<int>G[maxn];
vector<edge> edges;
bool inque[maxn];
int dis[maxn];

void add(int u, int v, int w) {
    edges.push_back(edge(u, v, w));
    int m = edges.size();
    G[u].push_back(m-1);
}

int spfa(int n, int s) {
    for (int i = 1; i <= n; i++) {
        dis[i] = INF;
    }
    dis[s] = 0;
    inque[s] = true;
    queue<int>que;
    que.push(s);
    while (!que.empty()) {
        int u = que.front();
        inque[u] = false;
        que.pop();
        for (int i = 0; i < (int)G[u].size(); i++) {
            edge e = edges[G[u][i]];
            if (dis[e.to] > dis[u] + e.cost) {
                dis[e.to] = dis[u] + e.cost;
                if (!inque[e.to]) {
                    que.push(e.to);
                    inque[e.to] = true;
                }
            }
```

```
        }
    }
    int ans = 0;
    for (int i = 1; i <= n; i++)
        ans = max(ans, dis[i]);
    return ans;
}

map<string, int> mp;

void init(int n) {
    for (int i = 0; i <= n; i++) G[i].clear();
    edges.clear();
    mp.clear();
}

int main() {
    //freopen("in.txt", "r", stdin);
    int n, m;
    while (scanf("%d", &n) != EOF && n) {
        init(n);
        int cnt = 1;
        char a[20], b[20];
        for (int i = 0; i < n; i++) {
            scanf("%s", a);
            if (mp[a] == 0) mp[a] = cnt++;
        }
        scanf("%d", &m);
        for (int i = 0; i < m; i++) {
            scanf("%s%s", a, b);
            int u = mp[a], v = mp[b];
            add(u, v, 1);
            add(v, u, 1);
        }
        int ans = 0;
        for (int i = 1; i <= n; i++) {
```

```
            ans = max(ans, spfa(n, i));
        }
        printf("%d\n", ans == INF ? -1 : ans);
    }
    return 0;
}
```

最短路 Floyed

求多源最短路的一种方法，经常做预处理用，能处理出来图中任意两个点之间的最短距离，时间复杂度O(n^3) 模板代码：

```
for (int k = 1; k <= n; k++) {
            for (int i = 1; i <= n; i++) {
                if (i == k) continue;
                for (int j = 1; j <= n; j++) {
                    if (i == j || k == j) continue;
                    dis[i][j] = min(dis[i][j], dis[i][k]
+ dis[k][j]);
                }
            }
        }
```

POJ 1125 Stockbroker Grapevine

```
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
using namespace std;

const int maxv = 500 + 5;

int dis[maxv][maxv];


int main() {
    int n, num, v, w;
//    freopen("in.txt", "r", stdin);
    while (scanf("%d", &n) != EOF && n) {
        memset(dis, 0x3f, sizeof(dis));
```

```
        for (int u = 1; u <= n; u++) {
            scanf("%d", &num);
            for (int j = 0; j < num; j++) {
                scanf("%d%d", &v, &w);
                dis[u][v] = min(dis[u][v], w);
            }
            dis[u][u] = 0;
        }

        for (int k = 1; k <= n; k++) {
            for (int i = 1; i <= n; i++) {
                if (i == k) continue;
                for (int j = 1; j <= n; j++) {
                    if (i == j || k == j) continue;
                    dis[i][j] = min(dis[i][j], dis[i][k]
+ dis[k][j]);
                }
            }
        }
        bool ok = true;
        int anstime = 0x3f, ansid = 0;
        for (int i = 1; i <= n; i++) {
            int minn = 0;
            for (int j = 1; j <= n; j++) {
                if (i == j) {
                    continue;
                }
                if (dis[i][j] == 0x3f) {
                    ok = false;
                    goto end;
                }
                minn = max(minn, dis[i][j]);
            }
            if (minn < anstime) {
                anstime = minn;
                ansid = i;
```

```
            }
        }
end:
        if (ok) {
            printf("%d %d\n", ansid, anstime);
        } else {
            puts("disjoint");
        }

    }
    return 0;
}
```

次短路与第K短路

次短路是除了最短路之外第二短的路，这条路的长度有可能和最短路一样长。
第K短路就是第K短的路，鉴于这两个算法都是特别模板的题，直接上例子

HRBUST 1050 Hot Pursuit II

求次短路：Dijkstra的dist数组和vis数组再加一维，松弛的时候讨论当前的路小于最短路，或者大于最短路但小于次短路这两种情况，就能维护一个次短路了

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
#include <algorithm>
using namespace std;

const int maxn = 1000 + 5;
const int INF = 0x3f3f3f3f;

struct Node {
    int v, c, flag;
    Node (int _v = 0, int _c = 0, int _flag = 0) : v(_v),
c(_c), flag(_flag) {}
    bool operator < (const Node &rhs) const {
        return c > rhs.c;
    }
};

struct Edge {
    int v, cost;
    Edge (int _v = 0, int _cost = 0) : v(_v), cost(_cost)
{}
};
```

```
vector<Edge>E[maxn];
bool vis[maxn][2];
int dist[maxn][2];

void Dijkstra(int n, int s) {
    memset(vis, false, sizeof(vis));
    for (int i = 1; i <= n; i++) {
        dist[i][0] = INF;
        dist[i][1] = INF;
    }
    priority_queue<Node>que;
    dist[s][0] = 0;
    que.push(Node(s, 0, 0));
    while (!que.empty()) {
        Node tep = que.top(); que.pop();
        int u = tep.v;
        int flag = tep.flag;
        if (vis[u][flag]) continue;
        vis[u][flag] = true;
        for (int i = 0; i < (int)E[u].size(); i++) {
            int v = E[u][i].v;
            int cost = E[u][i].cost;
            if (!vis[v][0] && dist[v][0] > dist[u][flag]
+ cost) {
                dist[v][1] = dist[v][0];
                dist[v][0] = dist[u][flag] + cost;
                que.push(Node(v, dist[v][0], 0));
                que.push(Node(v, dist[v][1], 1));
            } else if (!vis[v][1] && dist[v][1] > dist[u]
[flag] + cost) {
                dist[v][1] = dist[u][flag] + cost;
                que.push(Node(v, dist[v][1], 1));
            }
        }
    }
}
```

```
void addedge(int u, int v, int w) {
    E[u].push_back(Edge(v, w));
}

int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, v, w;
    while (scanf("%d", &n) != EOF) {
        for (int i = 0; i <= n; i++) E[i].clear();
        for (int u = 1; u <= n; u++) {
            scanf("%d", &m);
            for (int j = 0; j < m; j++) {
                scanf("%d%d", &v, &w);
                addedge(u, v, w);
            }
        }
        Dijkstra(n, 1);
        printf("%d\n", dist[n][1]);
    }
    return 0;
}
```

POJ 2449 Remmarguts' Date

求S点到T点的第K短路的长度

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
#include <algorithm>
using namespace std;

const int maxn = 1000 + 5;
const int INF = 0x3f3f3f3f;
```

```
int s, t, k;

bool vis[maxn];
int dist[maxn];

struct Node {
    int v, c;
    Node (int _v = 0, int _c = 0) : v(_v), c(_c) {}
    bool operator < (const Node &rhs) const {
        return c + dist[v] > rhs.c + dist[rhs.v];
    }
};

struct Edge {
    int v, cost;
    Edge (int _v = 0, int _cost = 0) : v(_v), cost(_cost)
{}
};

vector<Edge>E[maxn], revE[maxn];

void Dijkstra(int n, int s) {
    memset(vis, false, sizeof(vis));
    for (int i = 1; i <= n; i++) dist[i] = INF;
    priority_queue<Node>que;
    dist[s] = 0;
    que.push(Node(s, 0));
    while (!que.empty()) {
        Node tep = que.top(); que.pop();
        int u = tep.v;
        if (vis[u]) continue;
        vis[u] = true;
        for (int i = 0; i < (int)E[u].size(); i++) {
            int v = E[u][i].v;
            int cost = E[u][i].cost;
            if (!vis[v] && dist[v] > dist[u] + cost) {
```

```
                dist[v] = dist[u] + cost;
                que.push(Node(v, dist[v]));
            }
        }
    }
}

int astar(int s) {
    priority_queue<Node> que;
    que.push(Node(s, 0)); k--;
    while (!que.empty()) {
        Node pre = que.top(); que.pop();
        int u = pre.v;
        if (u == t) {
            if (k) k--;
            else return pre.c;
        }
        for (int i = 0; i < (int)revE[u].size(); i++) {
            int v = revE[u][i].v;
            int c = revE[u][i].cost;
            que.push(Node(v, pre.c + c));
        }
    }
    return -1;
}

void addedge(int u, int v, int w) {
    revE[u].push_back(Edge(v, w));
    E[v].push_back(Edge(u, w));
}

int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, u, v, w;
    while (scanf("%d%d", &n, &m) != EOF) {
        for (int i = 0; i <= n; i++) {
```

```
            E[i].clear();
            revE[i].clear();
        }
        for (int i = 0; i < m; i++) {
            scanf("%d%d%d", &u, &v, &w);
            addedge(u, v, w);
        }
        scanf("%d%d%d", &s, &t, &k);
        Dijkstra(n, t);
        if (dist[s] == INF) {
            puts("-1");
            continue;
        }
        if (s == t) k++;
        printf("%d\n", astar(s));
    }
    return 0;
}
```

最近公共祖先

最近公共祖先，树上两个点都能到达的，距离这两个点的距离和最近的一个点。也经常用LCA求树上两点间的距离。

在线的倍增法O(nlogn)：

POJ 1330 Nearest Common Ancestors

模板题， 求树上两个点的LCA

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <algorithm>
using namespace std;

const int maxn = 10000 + 5;
const int DEG = 20;

struct edge {
    int to, nxt;
} e[maxn << 1];

int head[maxn], tot;

void addedge(int u, int v) {
    e[tot].to = v;
    e[tot].nxt = head[u];
    head[u] = tot++;
}

void init() {
    tot = 0;
    memset(head, -1, sizeof(head));
}
```

```cpp
int fa[maxn][DEG];
int deg[maxn];

void bfs(int root) {
    queue<int> q;
    deg[root] = 0;
    fa[root][0] = root;
    q.push(root);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int i = 1; i < DEG; i++)
            fa[u][i] = fa[fa[u][i-1]][i-1];
        for (int i = head[u]; i != -1; i = e[i].nxt) {
            int v = e[i].to;
            if (v == fa[u][0]) continue;
            deg[v] = deg[u] + 1;
            fa[v][0] = u;
            q.push(v);
        }
    }
}

int lca(int u, int v) {
    if (deg[u] > deg[v]) swap(u, v);
    int hu = deg[u], hv = deg[v];
    int tu = u, tv = v;
    for (int det = hv - hu, i = 0; det; det >>= 1, i++)
        if (det & 1) tv = fa[tv][i];
    if (tu == tv) return tu;
    for (int i = DEG - 1; i >= 0; i--) {
        if (fa[tu][i] == fa[tv][i]) continue;
        tu = fa[tu][i];
        tv = fa[tv][i];
    }
    return fa[tu][0];
```

```
}

int in[maxn];

int main() {
    //freopen("in.txt", "r", stdin);
    int t, n, u, v;
    scanf("%d", &t);
    while (t--) {
        scanf("%d", &n);
        init();
        memset(in, 0, sizeof(in));
        for (int i = 1; i < n; i++) {
            scanf("%d%d", &u, &v);
            addedge(u, v);
            addedge(v, u);
            in[v]++;
        }
        for (int i = 1; i <= n; i++) {
            if (!in[i]) {
                bfs(i);
                break;
            }
        }
        scanf("%d%d", &u, &v);
        printf("%d\n", lca(u, v));
    }
    return 0;
}
```

CDOJ 92 Journey

给出一棵有边权的树，再加上一条边，之后Q组询问，每次求两点之间的最短距离缩短了多少

添加一条边CD之后，原A，B间的最短路的走法可能变为 A-C-D-B 或者 A-D-B-C，和原来的A-B减一下就行了

```cpp
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int MAXN = 100000 + 5;
const int DEP = 20;

struct edge {
    int to, nxt, cost;
} e[MAXN << 1];

int head[MAXN], tot;

void init() {
    tot = 0;
    memset(head, -1, sizeof(head));
}

void addedge(int u, int v, int w) {
    e[tot].to = v;
    e[tot].cost = w;
    e[tot].nxt = head[u];
    head[u] = tot++;
}

int fa[MAXN][DEP];
int dep[MAXN], dis[MAXN], n;

void dfs(int u, int pre, int d) {
    fa[u][0] = pre;
    dep[u] = d;
    for (int i = head[u]; i != -1; i = e[i].nxt) {
        int v = e[i].to;
        if (v == pre) continue;
```

```
            dis[v] = dis[u] + e[i].cost;
            dfs(v, u, d + 1);
        }
    }

    void initlca(int root) {
        dis[root] = 0;
        dfs(root, -1, 0);
        for (int i = 1; i < DEP; i++) {
            for (int u = 1; u <= n; u++) {
                if (fa[u][i-1] == -1) fa[u][i] = u;
                else fa[u][i] = fa[fa[u][i-1]][i-1];
            }
        }
    }

    int lca(int u, int v) {
        if (dep[u] > dep[v]) swap(u, v);
        int hu = dep[u], hv = dep[v];
        int tu = u, tv = v;
        for (int det = hv - hu, i = 0; det; det >>= 1, i++)
            if (det & 1) tv = fa[tv][i];
        if (tu == tv) return tu;
        for (int i = DEP - 1; i >= 0; i--) {
            if (fa[tu][i] == fa[tv][i]) continue;
            tu = fa[tu][i];
            tv = fa[tv][i];
        }
        return fa[tu][0];
    }

    int main() {
        //freopen("in.txt", "r", stdin);
        int t, u, v, w, q;
        scanf("%d", &t);
        for (int _ = 1; _ <= t; _++) {
```

```
        printf("Case #%d:\n", _);
        scanf("%d%d", &n, &q);
        init();
        for (int i = 1; i < n; i++) {
            scanf("%d%d%d", &u, &v, &w);
            addedge(u, v, w);
            addedge(v, u, w);
        }
        initlca(1);
        scanf("%d%d%d", &u, &v, &w);
        while (q--) {
            int a, b;
            scanf("%d%d", &a, &b);
            int L1 = dis[a] + dis[b] - 2 * dis[lca(a,
b)];

            int L2 = dis[a] + dis[u] - 2 * dis[lca(a, u)]
            + dis[b] + dis[v] - 2 * dis[lca(b, v)] + w;

            int L3 = dis[a] + dis[v] - 2 * dis[lca(a, v)]
            + dis[b] + dis[u] - 2 * dis[lca(b, u)] + w;
            printf("%d\n", L1 - min(L1, min(L2, L3)));
        }
    }
    return 0;
}
```

最小生成树 Kruskal

Kruskal适用于点比较多边比较少的情况(稀疏图)

HDU 1875 畅通工程再续

```
#include <cstdio>
#include <cmath>
#include <cstring>
#include <set>
#include <algorithm>
using namespace std;

const int MAXN = 1000 + 5;
const double eps = 0;

int fa[MAXN];

struct point {
    double x, y;
}p[MAXN];

double dis(int i, int j) {
    return sqrt( (p[i].x - p[j].x) * (p[i].x - p[j].x) +
(p[i].y - p[j].y) * (p[i].y - p[j].y));
}

struct edge {
    int u, v;
    double w;
    bool operator < (const edge &rhs) const {
        return w < rhs.w;
    }
} e[MAXN * MAXN];
```

```
int findfa(int u) {
    if (fa[u] != u) fa[u] = findfa(fa[u]);
    return fa[u];
}

int m;
double cost;

void unin(int u, int v, double w) {
    int fau = findfa(u);
    int fav = findfa(v);
    if (fau == fav) return;
    if (fau < fav) {
        fa[fav] = fau;
    } else {
        fa[fau] = fav;
    }
    cost += w;
    m--;
}

int main() {
    //freopen("in.txt", "r", stdin);
    int n, t;
    scanf("%d", &t);
    while (t--) {
        scanf("%d", &n);
        for (int i = 1; i <= n; i++) {
            fa[i] = i;
            scanf("%lf%lf", &p[i].x, &p[i].y);
        }

        int cnt = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j < i; j++) {
                e[cnt].u = i;
```

```
                e[cnt].v = j;
                e[cnt++].w = dis(i, j);
            }
        }

        sort(e, e + cnt);
        m = n - 1, cost = 0;
        for (int i = 0; i < cnt; i++) {
            if (m == 0) break;
            //if (e[i].w < 10 || e[i].w  > 1000)
continue;
            if (e[i].w - 10 < eps || e[i].w - 1000 > eps)
continue;
            unin(e[i].u, e[i].v, e[i].w);
        }

        if (!m) printf("%.1f\n", cost * 100.0);
        else puts("oh!");
    }
    return 0;
}
```

HDU 4081 Qin Shi Huang's National Road System

给出一个图，每个节点是一个城市，权值代表人口，城市间的距离里为欧基里德距离。现在求这个图的一个生成树，使A/B尽可能的大，其中A是生成树的某条边的两个节点上人口数的和，B是这个生成树上除了刚才选中的那条边之外的所有边的距离的和

要B尽可能的小，先求原图的一个最小生成树，要添加一条边并替换最小生成树上的某一条边，使它仍是一个树。枚举要添加的边，维护一个A/B的最大值，即可求出答案，如果这条边是原生成树上的边，设原来的最小生成树上的边权和为EW，则B = EW – e[i].w; 如果要添加的边不是原来最小生成树上的边，B = EW – maxd[e[i].u][e[i].v]，maxd[i][j]表示在生成树上从i点到j点路上最长边的值

```
#include <cstdio>
```

```cpp
#include <cstring>
#include <cmath>
#include <queue>
#include <algorithm>
using namespace std;

const int MAXN = 1000 + 50;
int tot, cnt, n;

int head[MAXN], fa[MAXN], vis[MAXN];
double maxd[MAXN][MAXN];
bool G[MAXN][MAXN];

struct Edge{
    int u, v;
    double w;
    bool operator < (const Edge &rhs) const {
        return w < rhs.w;
    }
}E[MAXN * MAXN];

struct edge{
    int to, nxt;
    double cost;
}e[MAXN << 2];

void init() {
    tot = 0, cnt = 0;
    memset(head, -1, sizeof(head));
    memset(maxd, 0, sizeof(maxd));
    memset(G, 0, sizeof(G));
}

void addedge(int u, int v, double w) {
    e[tot].to = v;
    e[tot].cost = w;
```

```
    e[tot].nxt = head[u];
    head[u] = tot++;
}

struct point {
    int x, y;
    double w;
}p[MAXN];

void bfs(int s) {
    queue<int>q;
    memset(vis, 0, sizeof(vis));
    vis[s] = 1;
    q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int i = head[u]; i != -1; i = e[i].nxt) {
            int v = e[i].to;
            if (vis[v]) continue;
            maxd[s][v] = maxd[v][s] = max(maxd[s][u],
e[i].cost);
            q.push(v);
            vis[v] = 1;
        }
    }
}

void dfs(int rt, int u, int fa, double md) {
    for (int i = head[u]; i != -1; i = e[i].nxt) {
        int v = e[i].to;
        if (v == fa) continue;
        maxd[rt][v] = maxd[v][rt] = max(md, e[i].cost);
        dfs(rt, v, u, maxd[rt][v]);
    }
}
```

```
int findfa(int u) {
    if (fa[u] != u) fa[u] = findfa(fa[u]);
    return fa[u];
}

double kru(int n) {
    int m = n - 1;
    double ew = 0;
    for (int i = 0; i <= n; i++) fa[i] = i;
    sort(E, E + cnt);
    for (int i = 0; i < cnt; i++) {
        int u = E[i].u, v = E[i].v;
        double  w = E[i].w;
        int fau = findfa(u);
        int fav = findfa(v);
        if (fau == fav) continue;
        if (fau > fav) swap(fau, fav);
        fa[fav] = fau;
        ew += E[i].w;
        addedge(u, v, w);
        addedge(v, u, w);
        G[u][v] = G[v][u] = 1;
        if(--m <= 0) break;
    }
    return ew;
}

double dis(int i, int j) {
    return sqrt((p[i].x-p[j].x)*(p[i].x-p[j].x) +
(p[i].y-p[j].y)*(p[i].y-p[j].y));
}

int main() {
    //freopen("in.txt", "r", stdin);
    int t;
    scanf("%d", &t);
```

```
    while (t--) {
        init();
        scanf("%d", &n);
        for (int i = 1; i <= n; i++)
            scanf("%d%d%lf", &p[i].x, &p[i].y, &p[i].w);

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i == j) continue;
                E[cnt].u = i;
                E[cnt].v = j;
                E[cnt++].w = dis(i, j);
            }
        }

        double ew = kru(n);
        //for (int i = 1; i <= n; i++) dfs(i, i, -1, 0);
        for (int i = 1; i <= n; i++) bfs(i);

        double ans = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j < i; j++) {
                ans = max(ans, (p[i].w + p[j].w) / (ew -
maxd[i][j]));
            }
        }
        printf("%.2f\n", ans);
    }
    return 0;
}
```

最小树形图

有向图的最小生成树，建好图直接套模板

POJ 3164 Command Network

```
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
using namespace std;

const int MAXN = 100 + 5;
const int MAXM = 10000 + 5;
const double INF = 1e9;

bool G[MAXN][MAXN];
int pre[MAXN], id[MAXN], vis[MAXN];
int cnt;
double in[MAXN];

struct point {
    double x, y;
    double dis(const point & p) const {
        return sqrt((x - p.x) * (x - p.x) + (y - p.y) *
(y - p.y));
    }
}p[MAXN];

struct edge{
    int u, v;
    double w;
}e[MAXM];

void addedge(int u, int v, double w) {
```

```
    e[cnt].u = u;
    e[cnt].v = v;
    e[cnt++].w = w;
}

double zl(int rt, int n, int m) {
    double ans = 0;
    int v;
    while (1) {
        for (int i = 0; i < n; i++) in[i] = INF;
        for (int i = 0; i < m; i++) {
            if (e[i].w < in[e[i].v]) {
                pre[e[i].v] = e[i].u;
                in[e[i].v] = e[i].w;
            }
        }
        for (int i = 0; i < n; i++) {
            if (i != rt && in[i] == INF) {
                return -1.0;
            }
        }
        int tn = 0;
        memset(id, -1, sizeof(id));
        memset(vis, -1, sizeof(vis));
        in[rt] = 0;
        for (int i = 0; i < n; i++) {
            ans += in[i];
            v = i;
            while (vis[v] == -1 && v != rt) {
                vis[v] = i;
                v = pre[v];
            }
            if (v != rt && vis[v] == i) {
                for (int u = pre[v]; u != v; u = pre[u])
{
                    id[u] = tn;
```

```
                }
                id[v] = tn++;
            }
        }
        if (tn == 0) break;
        for (int i = 0; i < n; i++) {
            if (id[i] == -1) id[i] = tn++;
        }
        for (int i = 0; i < m;) {
            v = e[i].v;
            e[i].u = id[e[i].u];
            e[i].v = id[e[i].v];
            if (e[i].u != e[i].v) {
                e[i++].w -= in[v];
            } else {
                swap(e[i], e[--m]);
            }
        }
        n = tn;
        rt = id[rt];
    }
    return ans;
}



int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, u, v;
    while (scanf("%d%d", &n, &m) != EOF) {
        for (int i = 0; i < n; i++) {
            scanf("%lf%lf", &p[i].x, &p[i].y);
        }
        memset(G, 0, sizeof(G));
        for (int i = 0; i < m; i++) {
            scanf("%d%d", &u, &v);
```

```
            if (u != v) G[u-1][v-1] = 1;
        }
        cnt = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (!G[i][j]) continue;
                addedge(i, j, p[i].dis(p[j]));
            }
        }
        double ans = zl(0, n, cnt);
        if (ans == -1.0) puts("poor snoopy");
        else printf("%.2f\n", ans);
    }
    return 0;
}
```

HDU 4009 Transfer water

在山上有N户人家，每家的坐标为(xi, yi, zi)。每户人家要吃水，要么自己打井，花费为A *zi*，要么从别人的家引水渠代价为*B* 两家的曼哈顿距离，如果这家的海拔比供水的低，还要另外再买一个价值为C的水泵。问每家都有水吃的最低花费是多少。

每家都有水的目标一定能达成，大不了每家都打口井...

把所有能连水渠的两家连起来，就得到了一个有向图，答案一定一个森林。建立一个虚节点，向每个点连一条有向边，权值为在第i家挖井的代价，这样就把答案从森林变成了一棵树，求全图的一个最小树形图就是最低的代价。

```
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
using namespace std;

const int MAXN = 1000 + 5;
const int MAXM = 1000000 + 5;
```

```
const int INF = 0x3f3f3f3f;

int pre[MAXN], id[MAXN], vis[MAXN];
int cnt;
int in[MAXN];

struct point {
    int x, y, z;
    int dis(const point & p) const {
        return (abs(x - p.x) + abs(y - p.y) + abs(z -
p.z));
    }
}p[MAXN];

struct edge{
    int u, v, w;
}e[MAXM];

void addedge(int u, int v, int w) {
    e[cnt].u = u;
    e[cnt].v = v;
    e[cnt++].w = w;
}

int zl(int rt, int n, int m) {
    int ans = 0;
    int v;
    while (1) {
        for (int i = 0; i < n; i++) in[i] = INF;
        for (int i = 0; i < m; i++) {
            if (e[i].u != e[i].v && e[i].w < in[e[i].v])
{
                pre[e[i].v] = e[i].u;
                in[e[i].v] = e[i].w;
            }
        }
```

```
        for(int i = 0;i < n;i++)
            if(i != rt && in[i] == INF)
                return -1;
        int tn = 0;
        memset(id, -1, sizeof(id));
        memset(vis, -1, sizeof(vis));
        in[rt] = 0;
        for (int i = 0; i < n; i++) {
            ans += in[i];
            v = i;
            while (vis[v] != i && id[v] == -1 && v != rt)
{
                vis[v] = i;
                v = pre[v];
            }
            if (v != rt && id[v] == -1) {
                for (int u = pre[v]; u != v; u = pre[u])
{
                    id[u] = tn;
                }
                id[v] = tn++;
            }
        }
        if (tn == 0) break;
        for (int i = 0; i < n; i++) {
            if (id[i] == -1) id[i] = tn++;
        }
        for (int i = 0; i < m;) {
            v = e[i].v;
            e[i].u = id[e[i].u];
            e[i].v = id[e[i].v];
            if (e[i].u != e[i].v) {
                e[i++].w -= in[v];
            } else {
               swap(e[i], e[--m]);
            }
```

```
        }
        n = tn;
        rt = id[rt];
    }
    return ans;
}

int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, v, a, b, c;
    while (scanf("%d%d%d%d", &n, &a, &b, &c) != EOF) {
        if (n == 0) break;
        for (int i = 1; i <= n; i++) {
            scanf("%d%d%d", &p[i].x, &p[i].y, &p[i].z);
        }

        cnt = 0;

        for (int u = 1; u <= n; u++) {
            scanf("%d", &m);
            for (int j = 0; j < m; j++) {
                scanf("%d", &v);
                if (v == u) continue;
                if (p[v].z <= p[u].z)
                    addedge(u, v, b * p[u].dis(p[v]));
                else
                    addedge(u, v, c + b *
p[u].dis(p[v]));
            }
        }
        for (int v = 1; v <= n; v++) {
            addedge(0, v, p[v].z * a);
        }
        int ans = zl(0, n+1, cnt);
        printf("%d\n", ans);
    }
```

```
    return 0;
}
```

一般图的最大匹配(带花树)

不只是二分图，一般图也能求最大匹配

ZOJ 3316 GAME

平面上有N个点，两个人轮流取平面上曼哈顿距离不大于L的点，谁没有点取谁就输，问后手能不能赢。

求图中每个连通分量的最大匹配，如果每个匹配都是完全匹配，后手一定赢，他每次都走先手的匹配点即可；如果某个匹配不是完全匹配，先手在这个连通分量上随便找一个点开始走，后手最后一定找不到可走的点。

```
#include <cstdio>
#include <cstring>

const int MAXN = 500;
int N, L;
bool Graph[MAXN][MAXN];
int Match[MAXN];
bool InQueue[MAXN],InPath[MAXN],InBlossom[MAXN];
int Head,Tail;
int Queue[MAXN];
int Start,Finish;
int NewBase;
int Father[MAXN],Base[MAXN];
int Count;
struct point {
    int x, y;
}p[MAXN];

inline int abs(int a) {
    return  a > 0 ? a : -a;
}

void CreateGraph() {
```

```
    memset(Graph,false,sizeof(Graph));
    for (int i = 1; i <= N; i++)
        scanf("%d%d", &p[i].x, &p[i].y);
    scanf("%d", &L);
    for (int i = 1; i <= N; i++)
        for (int j = i+1; j <= N; j++)
            if (abs(p[i].x-p[j].x)+abs(p[i].y-p[j].y) <=
L)
                Graph[i][j] = Graph[j][i] = true;
}

void Push(int u) {
    Queue[Tail] = u;
    Tail++;
    InQueue[u] = true;
}

int Pop() {
    int res = Queue[Head];
    Head++;
    return res;
}

int FindCommonAncestor(int u,int v) {
    memset(InPath,false,sizeof(InPath));
    while(true) {
        u = Base[u];
        InPath[u] = true;
        if(u == Start) break;
        u = Father[Match[u]];
    }

    while(true) {
        v = Base[v];
        if(InPath[v])break;
        v = Father[Match[v]];
```

```
    }
    return v;
}

void ResetTrace(int u) {
    int v;
    while(Base[u] != NewBase) {
        v = Match[u];
        InBlossom[Base[u]] = InBlossom[Base[v]] = true;
        u = Father[v];
        if(Base[u] != NewBase) Father[u] = v;
    }
}

void BloosomContract(int u,int v) {
    NewBase = FindCommonAncestor(u,v);
    memset(InBlossom,false,sizeof(InBlossom));
    ResetTrace(u);
    ResetTrace(v);
    if(Base[u] != NewBase) Father[u] = v;
    if(Base[v] != NewBase) Father[v] = u;
    for(int tu = 1; tu <= N; tu++)
        if(InBlossom[Base[tu]]) {
            Base[tu] = NewBase;
            if(!InQueue[tu]) Push(tu);
        }
}

void FindAugmentingPath() {
    memset(InQueue,false,sizeof(InQueue));
    memset(Father,0,sizeof(Father));
    for(int i = 1;i <= N;i++)
        Base[i] = i;
    Head = Tail = 1;
    Push(Start);
    Finish = 0;
```

```
    while(Head < Tail) {
        int u = Pop();
        for(int v = 1; v <= N; v++)
            if(Graph[u][v] && (Base[u] != Base[v]) &&
(Match[u] != v)) {
                if((v == Start) || ((Match[v] > 0) &&
Father[Match[v]] > 0))
                    BloosomContract(u,v);
                else if(Father[v] == 0) {
                    Father[v] = u;
                    if(Match[v] > 0)
                        Push(Match[v]);
                    else {
                        Finish = v;
                        return;
                    }
                }
            }
    }
}

void AugmentPath() {
    int u,v,w;
    u = Finish;
    while(u > 0) {
        v = Father[u];
        w = Match[v];
        Match[v] = u;
        Match[u] = v;
        u = w;
    }
}

void Edmonds() {
    memset(Match,0,sizeof(Match));
    for(int u = 1; u <= N; u++)
```

```
        if(Match[u] == 0) {
            Start = u;
            FindAugmentingPath();
            if(Finish > 0)AugmentPath();
        }
}

void PrintMatch() {
    Count = 0;
    for(int u = 1; u <= N; u++)
        if(Match[u] > 0)
            Count++;
    //    printf("%d\n",Count);
    //  for(int u = 1; u <= N; u++)
    ////    if(u < Match[u])
    //    printf("%d %d\n",u,Match[u]);
    if (Count == N) printf("YES\n");
    else printf("NO\n");
}

int main() {
    while (scanf("%d", &N) != EOF) {
        CreateGraph();
        Edmonds();
        PrintMatch();
    }
    return 0;
}
```

HDU 4687 Boke and Tsukkomi

给出N个点M条边的一个图，求哪几条边不在这个图的任意一个最大匹配中。

设没删任何一条边的时候的最大匹配数为K，枚举每一条边e[i]，假设这条边在最大匹配上所以删去和e[i].u，e[i].v相连的每一条边，若此时的匹配数为K-1，此边一定在某个最大匹配上并把它标记。枚举M次后没有被标记的边就是不在任何一个最大

匹配上的边。

注意可能所有边都在最大匹配上，这时候要输出一个空行。

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;

const int MAXN = 500 + 5;
int N, M;
bool G[MAXN][MAXN];
int Match[MAXN];
bool InQueue[MAXN],InPath[MAXN],InBlossom[MAXN];
int Head,Tail;
int Queue[MAXN];
int Start,Finish;
int NewBase;
int Father[MAXN],Base[MAXN];
int Count;

struct point {
    int x, y;
}p[MAXN];

struct edge {
    int u, v;
}e[MAXN * MAXN];

void CreateG() {
    memset(G,false,sizeof(G));
    int u, v;
    for (int i = 0; i < M; i++) {
        scanf("%d%d", &u, &v);
        G[u][v] = G[v][u] = true;
        e[i+1].u = u; e[i+1].v = v;
```

```
    }
}

void Push(int u) {
    Queue[Tail] = u;
    Tail++;
    InQueue[u] = true;
}

int Pop() {
    int res = Queue[Head];
    Head++;
    return res;
}

int FindCommonAncestor(int u,int v) {
    memset(InPath,false,sizeof(InPath));
    while(true) {
        u = Base[u];
        InPath[u] = true;
        if(u == Start) break;
        u = Father[Match[u]];
    }

    while(true) {
        v = Base[v];
        if(InPath[v])break;
        v = Father[Match[v]];
    }
    return v;
}

void ResetTrace(int u) {
    int v;
    while(Base[u] != NewBase) {
        v = Match[u];
```

```
        InBlossom[Base[u]] = InBlossom[Base[v]] = true;
        u = Father[v];
        if(Base[u] != NewBase) Father[u] = v;
    }
}

void BloosomContract(int u,int v) {
    NewBase = FindCommonAncestor(u,v);
    memset(InBlossom,false,sizeof(InBlossom));
    ResetTrace(u);
    ResetTrace(v);
    if(Base[u] != NewBase) Father[u] = v;
    if(Base[v] != NewBase) Father[v] = u;
    for(int tu = 1; tu <= N; tu++)
        if(InBlossom[Base[tu]]) {
            Base[tu] = NewBase;
            if(!InQueue[tu]) Push(tu);
        }
}

void FindAugmentingPath() {
    memset(InQueue,false,sizeof(InQueue));
    memset(Father,0,sizeof(Father));
    for(int i = 1;i <= N;i++)
        Base[i] = i;
    Head = Tail = 1;
    Push(Start);
    Finish = 0;
    while(Head < Tail) {
        int u = Pop();
        for(int v = 1; v <= N; v++)
            if(G[u][v] && (Base[u] != Base[v]) &&
(Match[u] != v)) {
                if((v == Start) || ((Match[v] > 0) &&
Father[Match[v]] > 0))
                    BloosomContract(u,v);
```

```
                else if(Father[v] == 0) {
                    Father[v] = u;
                    if(Match[v] > 0)
                        Push(Match[v]);
                    else {
                        Finish = v;
                        return;
                    }
                }
            }
        }
    }

    void AugmentPath() {
        int u,v,w;
        u = Finish;
        while(u > 0) {
            v = Father[u];
            w = Match[v];
            Match[v] = u;
            Match[u] = v;
            u = w;
        }
    }

    void Edmonds() {
        memset(Match,0,sizeof(Match));
        for(int u = 1; u <= N; u++)
            if(Match[u] == 0) {
                Start = u;
                FindAugmentingPath();
                if(Finish > 0)AugmentPath();
            }
    }

    int PrintMatch() {
```

```
    Count = 0;
    for(int u = 1; u <= N; u++)
        if(Match[u] > 0)
            Count++;
    return Count;
    //  printf("%d\n",Count);
    //  for(int u = 1; u <= N; u++)
    //      if(u < Match[u])
    //          printf("%d %d\n",u,Match[u]);
}

vector<int>ans;

int vis[MAXN];

void Solve() {
    memset(vis, 0, sizeof(vis));
    ans.clear();
    int maxmatch = PrintMatch() / 2;
    for (int i = 1; i <= M; i++) {
        memset(G, 0, sizeof(G));
        int u = e[i].u, v = e[i].v;
        for (int j = 1; j <= M; j++) if (i != j){
            if (e[j].u == u || e[j].v == u || e[j].u == v
||
                    e[j].v == v) continue;
            G[e[j].u][e[j].v] = G[e[j].v][e[j].u] = 1;
        }
        Edmonds();
        int k = PrintMatch() / 2;
        if (k == maxmatch-1)
            vis[i] = 1;
    }
    for (int i = 1; i <= M; i++) if (!vis[i])
ans.push_back(i);
    int sz = ans.size();
```

```
    printf("%d\n", sz);
    if (sz > 0) {
        printf("%d", ans[0]);
        for (int i = 1; i < sz; i++) {
            printf(" %d", ans[i]);
        }
    }
    puts("");
}

int main() {
    //freopen("in.txt", "r", stdin);
    while (scanf("%d%d", &N, &M) != EOF) {
        CreateG();
        Edmonds();
        Solve();
    }
    return 0;
}
```

最大流 Dinic

一个图，每条边都有一个容量限制，一般有一个起点，一个终点，求达到平衡状态时，最多有多少流量从起点流向终点。难点在于建图，建好图直接套模板

HDU 3549 Flow Problem

```
#include <cstdio>
#include <queue>
#include <cstring>
#include <vector>
using namespace std;
const int maxn = 100;
const int maxm = 30000 + 5;
const long long INF = 0x3f3f3f;

struct Edge {
    int from, to, cap, flow;
    Edge(){}
    Edge(int from, int to, int cap, int flow) {
        this->from = from;
        this->to = to;
        this->cap = cap;
        this->flow = flow;
    }
};

struct Dinic {
    int n, m, s, t;
    Edge edges[maxm];
    int head[maxn];
    int nxt[maxm];
    bool vis[maxn];
    int d[maxn];
    int cur[maxn];
```

```
    void init(int n) {
        this -> n = n;
        memset(head, -1, sizeof(head));
        m = 0;
    }

    void AddEdge(int u, int v, int c) {
        edges[m] = Edge(u, v, c, 0);
        nxt[m] = head[u];
        head[u] = m++;
        edges[m] = Edge(v, u, 0, 0);
        nxt[m] = head[v];
        head[v] = m++;
    }

    bool BFS() {
        memset(vis, 0, sizeof(vis));
        queue<int>Q;
        Q.push(s);
        d[s] = 0;
        vis[s] = 1;
        while (!Q.empty()) {
            int x = Q.front(); Q.pop();
            for (int i = head[x]; i != -1; i = nxt[i]) {
                Edge& e = edges[i];
                if (!vis[e.to] && e.cap > e.flow) {
                    vis[e.to] = 1;
                    d[e.to] = d[x] + 1;
                    Q.push(e.to);
                }
            }
        }
        return vis[t];
    }
```

```
    int DFS(int x, int a) {
        if (x == t || a == 0) return a;
        int flow = 0, f;
        for (int& i = cur[x]; i != -1; i = nxt[i]) {
            Edge& e = edges[i];
            if (d[x] + 1 == d[e.to] && (f = DFS(e.to,
min(a, e.cap-e.flow))) > 0) {
                e.flow += f;
                edges[i^1].flow -= f;
                flow += f;
                a -= f;
                if (a == 0) break;
            }
        }
        return flow;
    }

    int Maxflow(int s, int t) {
        this -> s = s; this -> t = t;
        int flow = 0;
        while (BFS()) {
            for (int i = 0; i < n; i++)
                cur[i] = head[i];
            flow += DFS(s, INF);
        }
        return flow;
    }
} H;

int main() {
    //freopen("in.txt", "r", stdin);
    int n, m, u, v, c, i, t;
    scanf("%d", &t);
    for (int ii = 1; ii <= t; ii++) {
        scanf("%d%d", &n, &m);
        H.init(n + 1);
```

```
        for(i=1; i<=m; ++i) {
            scanf("%d%d%d", &u, &v, &c);
            H.AddEdge(u, v, c);
        }
        printf("Case %d: %d\n", ii, H.Maxflow(1, n));
    }
    return 0;
}
```

HDU 5352 MZL's City

题意有点难：有N个点的一个无向图，最开始每个点都是不可用的，每两点之间都没有边，然后会有M个操作：

```
1 x ,    可以把和x直接相连或间接相连的点变成可用的，但最多修复K个
2 x, y   可以在x和y之间连一条无向边
3 p      后面有p对数，每对两个数x， y代表x，y之间的边被损毁了
```

对于每次1操作，输出要修复的点的数量，使总共修复的点的数量最多，在这基础上输出字典序最小的解

每次1操作的时候都会得到一个图，既然想求总共修复的点的数量最多，那么用贪心的思想，每次1操作的时候都尽量修复最多的点： 然后想了一组样例： 先进性操作2建边：1-3 1-4 1-5 1-6 2-5 2-6 然后1开始修复，限制为3，然后拆除 1-5 1-6，然后2开始修复，限制为3，求最多修复多少个城市

正解应该是1先修复1，3和4，然后2修复2，5和6，这样最多修复了1,2,3,4,5,6这6个城市，但是如果第一次先修复了1，5和6，然后题目把边1-5,1-6断掉了，这样第二次只能修复2号点了，最终修复了1,2,5,6这4个点，这样是不对的，所以难的地方在于题目会删边，如果不删边，在建完这个图的时候跑个最大流就行了，然后考虑字典序的关系让越靠后的询问先跑，这样就能保证字典序最小了。

然后删边的问题怎么解决呢？想想删边带来的影响：删边可能会使原本可达的点变的不可达了，减少了总方案数，所以，直接模拟这个删边过程，最终结果就是进行1操作时，（设此点的编号为s），每次我们都能达到s能到达的所有点组成的连通

分量，这个连通分量是在上次1操作之后进行添边减边得到的，那么题意就简化成了官方题解那样：左右两排点，每个左边的点可以匹配一些右边的点，最多匹配K次，求最大匹配次数

```
#include <cstdio>
#include <queue>
#include <cstring>
#include <vector>
using namespace std;

const int maxn = 1000;
const int maxm = 100000;
const long long INF = 0x3f3f3f;

struct Edge {
    int from, to, cap, flow;
    Edge(){}
    Edge(int from, int to, int cap, int flow) {
        this->from = from;
        this->to = to;
        this->cap = cap;
        this->flow = flow;
    }
};

struct Dinic {
    int n, m, s, t;
    Edge edges[maxm];
    int head[maxn];
    int nxt[maxm];
    bool vis[maxn];
    int d[maxn];
    int cur[maxn];

    void init(int n) {
        this -> n = n;
```

```
        memset(head, -1, sizeof(head));
        m = 0;
    }

    void AddEdge(int u, int v, int c) {
        edges[m] = Edge(u, v, c, 0);
        nxt[m] = head[u];
        head[u] = m++;
        edges[m] = Edge(v, u, 0, 0);
        nxt[m] = head[v];
        head[v] = m++;
    }

    bool BFS() {
        memset(vis, 0, sizeof(vis));
        queue<int>Q;
        Q.push(s);
        d[s] = 0;
        vis[s] = 1;
        while (!Q.empty()) {
            int x = Q.front(); Q.pop();
            for (int i = head[x]; i != -1; i = nxt[i]) {
                Edge& e = edges[i];
                if (!vis[e.to] && e.cap > e.flow) {
                    vis[e.to] = 1;
                    d[e.to] = d[x] + 1;
                    Q.push(e.to);
                }
            }
        }
        return vis[t];
    }

    int DFS(int x, int a) {
        if (x == t || a == 0) return a;
        int flow = 0, f;
```

```
        for (int& i = cur[x]; i != -1; i = nxt[i]) {
            Edge& e = edges[i];
            if (d[x] + 1 == d[e.to] && (f = DFS(e.to,
min(a, e.cap-e.flow))) > 0) {
                e.flow += f;
                edges[i^1].flow -= f;
                flow += f;
                a -= f;
                if (a == 0) break;
            }
        }
        return flow;
    }

    int Maxflow(int s, int t) {
        this -> s = s; this -> t = t;
        int flow = 0;
        while (BFS()) {
            for (int i = 0; i < n; i++)
            cur[i] = head[i];
            flow += DFS(s, INF);
        }
        return flow;
    }
} H;

int n, m, k, id;
int g[maxn][maxn], vis[maxn][maxn], ans[maxn];
vector<int>G[maxn];

void init() {
    memset(g, 0, sizeof(g));
    memset(vis, 0, sizeof(vis));
    id = 0;
}
```

```
void dfs(int id, int u) {
    vis[id][u] = 1;
    G[id].push_back(u);
    for (int i = 1; i <= n; i++) {
        if (vis[id][i] || !g[u][i]) continue;
        dfs(id, i);
    }
}

void solve() {
    int s = 0, t = n + id + 1;
    H.init(t + 1);
    for (int i = 1; i <= id; i++) {
        H.AddEdge(s, i, k);
    }
    for (int i = 1; i <= n; i++) {
        H.AddEdge(i + id, t, 1);
    }
    int sum = 0;
    for (int i = id; i >= 1; i--) {
        for (int j = 0; j < (int)G[i].size(); j++) {
            int v = G[i][j];
            H.AddEdge(i, v + id, 1);
        }
        sum += H.Maxflow(s, t);
    }

    for (int i = H.head[s]; i != -1; i = H.nxt[i]) {
        int w = H.edges[i].flow;
        int v = H.edges[i].to;
        ans[v] = w;
    }

    printf("%d\n", sum);
    for (int i = 1; i <= id; i++) {
        printf("%d%c", ans[i], i == id ? '\n' : ' ');
```

```
    }
    for (int i = 0; i <= id; i++) G[i].clear();
}

int main() {
    //freopen("in.txt", "r", stdin);
    int tcase;
    scanf("%d", &tcase);
    while (tcase--) {
        scanf("%d%d%d", &n, &m, &k);
        int op, x, y, p;
        init();
        for (int i = 0; i < m; i++) {
            scanf("%d", &op);
            if (op == 1) {
                scanf("%d", &x);
                id++;
                dfs(id, x);
            } else if (op == 2) {
                scanf("%d%d", &x, &y);
                g[x][y] = 1;
                g[y][x] = 1;
            } else if (op == 3) {
                scanf("%d", &p);
                while (p--) {
                    scanf("%d%d", &x, &y);
                    g[x][y] = 0;
                    g[y][x] = 0;
                }
            }
        }
        solve();
    }
    return 0;
}
```

最小割

一个图，每条边都有一个代价，现在需要破坏一些边使图中的某两个点不连通，求最小的代价是多少。
根据最大流-最小割定理，最小割可以通过最大流来求。
最小割的模板同最大流 如果只要求割成两部分，并不在意隔开了哪些点，这是全图最小割问题，可以供SW算法，模板如下:

HDU 3691 Nubulsa Expo

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
#define mem(a) memset(a, 0, sizeof(a))

const int maxv = 500;
const int inf = 0x3f3f3f3f;

int v[maxv], d[maxv];
int G[maxv][maxv];
bool vis[maxv];

int Stoer_Wanger(int n) {
    int res = inf;
    for (int i = 1; i <= n; i++) v[i] = i;

    while (n > 1) {
        int k = 1, pre = 1;
        mem(vis); mem(d);
        for (int i = 2; i <= n; i++) {
            k = -1;
            for (int j = 2; j <= n; j++) {
                if ( !vis[ v[j] ] ) {
                    d[ v[j] ] += G[ v[pre] ][ v[j] ];
```

```
                    if (k == -1 || d[ v[k] ] < d[ v[j] ]
) {
                        k = j;
                    }
                }
            }
            vis[ v[k] ] = true;
            if (i == n) {
                res = min(res, d[ v[k] ]);
                for (int j = 1; j <= n; j++) {
                    G[ v[pre] ][ v[j] ] += G[ v[j] ][
v[k] ];
                    G[ v[j] ][ v[pre] ] += G[ v[j] ][
v[k] ];
                }
                v[ k ] = v[ n-- ];
            }
            pre = k;
        }
    }
    return res;
}

int main() {
    int n, m, s, u, v, w;
    while (scanf("%d%d%d", &n, &m, &s) != EOF && s) {
        mem(G);
        for (int i = 0; i < m; i++) {
            scanf("%d%d%d", &u, &v, &w);
            G[u][v] += w;
            G[v][u] += w;
        }
        printf("%d\n", Stoer_Wanger(n));
    }
    return 0;
}
```

POJ 3469 Dual Core CPU

有一台电脑配了一个双核的CPU，现在内存中有N个模块，每个模块在每个CPU中运行的耗费已经被估算出来了，设为Ai 和Bi。同时，其中有M对模块需要共享资源，如果他们运行在同一个CPU中，共享数据的耗费可以忽略不计，否则，还需要额外的费用。必须很好的安排这N个模块，使得总耗费最小。

如果将两个CPU分别视为源点和汇点、模块视为顶点，则可以按照以下方式构图：对于第 i 个模块在每个CPU中的耗费Ai,Bi,从源点向顶点 i 连接一条容量为 w 的弧、从顶点 i 向汇点连接一条容量为Bi的弧；对于a 模块与 b 模块在不同的CPU中运行造成的额外耗费w，从顶点a 向顶点 b连接一条容量为 w 的弧。此时每个顶点（模块）都和源点及汇点（两个CPU）相连，即每个模块都可以在任意一个CPU中运行。

不难发现，对于图中的任意一个割，源点与汇点必不连通。因此每个顶点（模块）都不可能同时和源点及汇点（两个CPU）相连，即每个模块只能在同一个CPU中运行。此时耗费即为割的容量。显然，当割的容量最小时，总耗费最小。故题目转化为求最小割的容量

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int maxn = 100000;
const int maxm = 1000000;
const int inf = 1000000000;

class Dinic {
public :
    struct Node {
        int u, v, f, next;
    }e[maxm];

    int head[maxn], p, lev[maxn], cur[maxn];
    int que[maxm];
```

```
    void init() {
        p = 0;
        memset(head, -1, sizeof(head));
    }

    void add(int u, int v, int f) {
        e[p].u = u; e[p].v = v; e[p].f = f; e[p].next =
head[u]; head[u] = p++;
        e[p].u = v; e[p].v = u; e[p].f = 0; e[p].next =
head[v]; head[v] = p++;
    }

    bool bfs(int s, int t) {
        int u, v, qin = 0, qout = 0;
        memset(lev, 0, sizeof(lev)); lev[s] = 1;
que[qin++] = s;
        while(qout != qin) {
            u = que[qout++];
            for(int i = head[u]; i != -1; i = e[i].next)
{
                if(e[i].f > 0 && !lev[v = e[i].v]) {
                    lev[v] = lev[u] + 1, que[qin++] = v;
                    if(v == t) return 1;
                }
            }
        }
        return 0;
    }

    int dfs(int s, int t) {
        int i, f, qin, u, k;
        int flow = 0;
        while(bfs(s, t)) {
            memcpy(cur, head, sizeof(head));
            u = s, qin = 0;
```

```
            while(1) {
                if(u == t) {
                    for(k = 0, f = inf; k < qin; k++)
                        if(e[que[k]].f < f) f =
e[que[i=k]].f;
                    for(k = 0; k < qin; k++)
                        e[que[k]].f -= f, e[que[k]^1].f
+= f;
                    flow += f, u = e[que[qin = i]].u;
                }
                for(i = cur[u]; cur[u] != -1; i = cur[u]
= e[cur[u]].next)
                    if(e[i].f > 0 && lev[u] + 1 ==
lev[e[i].v]) break;
                if(cur[u] != -1)
                    que[qin++] = cur[u], u = e[cur[u]].v;
                else {
                    if(qin == 0) break;
                    lev[u] = -1, u = e[que[--qin]].u;
                }
            }
        }
        return flow;
    }
}H;

int main() {
    int n, m;
    while (scanf("%d%d", &n, &m) != EOF) {
        int s = 0, t = n+1;
        H.init();
        int cpu1, cpu2;
        for (int i = 1; i <= n; i++) {
            scanf("%d%d", &cpu1, &cpu2);
            H.add(s, i, cpu1);
            H.add(i, t, cpu2);
```

```
        }
        int u, v, w;
        for (int i = 0; i < m; i++) {
            scanf("%d%d%d", &u, &v, &w);
            H.add(u, v, w);
            H.add(v, u, w);
        }
        printf("%d\n", H.dfs(s, t));
    }
    return 0;
}
```

费用流

最小费用最大流：在流量最大的前提下，总费用最小。
难点在于建图。
模板如下：

HDU 3376 Matrix Again

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;

const int maxv = 1e+6;
const int maxe = 1e+7;
typedef int Type;
const Type INF = 0x3f3f3f3f;

struct Edge {
    int u, v;
    Type cap, flow, cost;
    Edge() {}
    Edge(int u, int v, Type cap, Type flow, Type cost) {
        this->u = u;
        this->v = v;
        this->cap = cap;
        this->flow = flow;
        this->cost = cost;
    }
};

struct MCFC {
    int n, m, s, t;
    Edge edges[maxe];
    int first[maxv];
```

```
    int next[maxe];
    int inq[maxv];
    Type d[maxv];
    int p[maxv];
    Type a[maxv];
    int Q[maxe];

    void init(int n) {
        this->n = n;
        memset(first, -1, sizeof(first));
        m = 0;
    }

    void AddEdge(int u, int v, Type cap, Type cost) {
        edges[m] = Edge(u, v, cap, 0, cost);
        next[m] = first[u];
        first[u] = m++;
        edges[m] = Edge(v, u, 0, 0, -cost);
        next[m] = first[v];
        first[v] = m++;
    }

    bool BellmanFord(int s, int t, Type &flow, Type
&cost) {

        for (int i = 0; i < n; i++) d[i] = INF;
        memset(inq, false, sizeof(inq));
        d[s] = 0; inq[s] = true; p[s] = 0; a[s] = INF;
        int front, rear;
        Q[rear = front = 0] = s;
        while (front <= rear) {
            int u = Q[front++];
            inq[u] = false;
            for (int i = first[u]; i != -1; i = next[i])
{
                Edge& e = edges[i];
```

```
                if (e.cap > e.flow && d[e.v] > d[u] +
e.cost) {
                    d[e.v] = d[u] + e.cost;
                    p[e.v] = i;
                    a[e.v] = min(a[u], e.cap - e.flow);
                    if (!inq[e.v]) {Q[++rear] = e.v;
inq[e.v] = true;}
                }
            }
        }
        if (d[t] == INF) return false;
        flow += a[t];
        cost += d[t] * a[t];
        int u = t;
        while (u != s) {
            edges[p[u]].flow += a[t];
            edges[p[u]^1].flow -= a[t];
            u = edges[p[u]].u;
        }
        return true;
    }

    Type Mincost(int s, int t) {
        Type flow = 0, cost = 0;
        while (BellmanFord(s, t, flow, cost));
        return cost;
    }
}H;

int mapp[610][610];

int main() {
    int n, u, v, w;
    freopen("in.txt", "r", stdin);
    while (scanf("%d", &n) != EOF) {
        for (int i = 0; i < n; i++)
```

```
            for (int j = 0; j < n; j++)
                scanf("%d", &mapp[i][j]);
        int s = 0, t = 2*n*n-1;
        H.init(t+1);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                H.AddEdge(i*n+j, i*n+j+n*n, 1, -mapp[i]
[j]);
                if (j != n-1)
                    H.AddEdge(i*n+j+n*n, i*n+j+1, 1, 0);
                if (i != n-1)
                    H.AddEdge(i*n+j+n*n, i*n+j+n, 1, 0);
            }
        }
        H.AddEdge(s, n*n, 1, 0);
        H.AddEdge(n*n-1, t, 1, 0);
        printf("%d\n", -H.Mincost(s, t));
    }
    return 0;
}
```

hdu 3718 Similarity

给出A，B两个分别由k不同字符组成的长度相同的字符串，现在把B中的每种字符变成A中含有的某种字符，(不能变成相同的A）,求最多能有多少个字符匹配。

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;

typedef int Type;
const int maxv = 1e+5;
const int maxe = 1e+7;
const int INF = 0x3f3f3f3f;
```

```
struct Edge {
    int u, v;
    Type cap, flow, cost;
    Edge() {}
    Edge(int u, int v, Type cap, Type flow, Type cost) {
        this->u = u;
        this->v = v;
        this->cap = cap;
        this->flow = flow;
        this->cost = cost;
    }
};

struct MCFC {
    int n, m, s, t;
    Edge edges[maxe];
    int first[maxv];
    int next[maxe];
    int inq[maxv];
    Type d[maxv];
    int p[maxv];
    Type a[maxv];
    int Q[maxe];

    void init(int n) {
        this->n = n;
        memset(first, -1, sizeof(first));
        m = 0;
    }

    void AddEdge(int u, int v, Type cap, Type cost) {
        edges[m] = Edge(u, v, cap, 0, cost);
        next[m] = first[u];
        first[u] = m++;
        edges[m] = Edge(v, u, 0, 0, -cost);
        next[m] = first[v];
```

```
        first[v] = m++;
    }

    bool BellmanFord(int s, int t, Type &flow, Type
&cost) {

        for (int i = 0; i < n; i++) d[i] = INF;
        memset(inq, false, sizeof(inq));
        d[s] = 0; inq[s] = true; p[s] = 0; a[s] = INF;
        int front, rear;
        Q[rear = front = 0] = s;
        while (front <= rear) {
            int u = Q[front++];
            inq[u] = false;
            for (int i = first[u]; i != -1; i = next[i])
{
                Edge& e = edges[i];
                if (e.cap > e.flow && d[e.v] > d[u] +
e.cost) {
                    d[e.v] = d[u] + e.cost;
                    p[e.v] = i;
                    a[e.v] = min(a[u], e.cap - e.flow);
                    if (!inq[e.v]) {Q[++rear] = e.v;
inq[e.v] = true;}
                }
            }
        }
        if (d[t] == INF) return false;
        flow += a[t];
        cost += d[t] * a[t];
        int u = t;
        while (u != s) {
            edges[p[u]].flow += a[t];
            edges[p[u]^1].flow -= a[t];
            u = edges[p[u]].u;
        }
```

```
        return true;
    }

    Type Mincost(int s, int t) {
        Type flow = 0, cost = 0;
        while (BellmanFord(s, t, flow, cost));
        return cost;
    }
}H;
char A[10000 + 10], B[5], C[10000 + 10], vis[2][52 + 10];
int mapp[26 + 10][26 + 10];

int main() {
    int tcase, n, m, k;
    //freopen("in.txt", "r", stdin);
    scanf("%d", &tcase);
    while (tcase--) {
        scanf("%d%d%d", &n, &k, &m);
        for (int i = 0; i < n; i++) {
            scanf("%s", B);
            A[i] = B[0];
        }
        while (m--) {
            int s = 52, t = s+1;
            H.init(t+1);
            memset(mapp, 0, sizeof(mapp));
            for (int i = 0; i < n; i++) {
                scanf("%s", B);
                mapp[A[i] - 'A'][B[0] - 'A']++;
                C[i] = B[0];
            }
            for (int i = 0; i < 26; i++) {
                for (int j = 0; j < 26; j++) {
                    if (mapp[i][j]) {
                        H.AddEdge(i, j+26, 1, -mapp[i]
[j]);
```

```
                    }
                }
            }
            memset(vis, 0, sizeof(vis));
            for (int i = 0; i < n; i++) {
                int a = A[i] - 'A', b = C[i]-'A'+26;
                if (!vis[0][a]) {
                    vis[0][a] = 1;
                    H.AddEdge(s, A[i] - 'A', 1, 0);
                }
                if (!vis[1][b]) {
                    vis[1][b] = 1;
                    H.AddEdge(C[i]-'A'+26, t, 1, 0);
                }
            }
            double len = n * 1.0;
            printf("%.4f\n", -H.Mincost(s, t)*1.0/len);
        }
    }
    return 0;
}
```

字符串

---

author：高放

后缀数组

后缀数组又被称为字符串处理神器；
http://blog.csdn.net/xymscau/article/details/8798046 这里讲的非常好
实现rank排名是用到了倍增法和一个比较神奇的计数排序，时间复杂度是$$O(n\log n)$$。

- `height[i]` 存放的是排名第 `i` 的后缀与排名第 `i-1` 的后缀的最长前缀
- `sa[i]` 存的是排名第i的后缀是第几位开头的
- `rk[i]` 存放第i个位置开头的后缀的字典序排名

做后缀数组的题基本上会套模板就行，但是要理解这几个数组的意义。

POJ 2774

题意：给出两串字符，要你找出在这两串字符中都出现过的最长子串。

思路：先用个分隔符将两个字符串连接起来，再用后缀数组求出 `height` 数组的值，找出 `sa[i]` 与 `sa[i-1]` 分别在分割的两个字符串时最大的height值。

```
#include<iostream>
#include<string.h>
#include<stdio.h>
using namespace std;

#define rep(i,n) for(int i = 0;i < n; i++)
using namespace std;
const int size  = 200005,INF = 1<<30;
int
rk[size],sa[size],height[size],w[size],wa[size],res[size]
;
void getSa (int len,int up) {
    int *k = rk,*id = height,*r = res, *cnt = wa;
    rep(i,up) cnt[i] = 0;
    rep(i,len) cnt[k[i] = w[i]]++;
```

```
    rep(i,up) cnt[i+1] += cnt[i];
    for(int i = len - 1; i >= 0; i--) {
        sa[--cnt[k[i]]] = i;
    }
    int d = 1,p = 0;
    while(p < len){
        for(int i = len - d; i < len; i++) id[p++] = i;
        rep(i,len)    if(sa[i] >= d) id[p++] = sa[i] - d;
        rep(i,len) r[i] = k[id[i]];
        rep(i,up) cnt[i] = 0;
        rep(i,len) cnt[r[i]]++;
        rep(i,up) cnt[i+1] += cnt[i];
        for(int i = len - 1; i >= 0; i--) {
            sa[--cnt[r[i]]] = id[i];
        }
        swap(k,r);
        p = 0;
        k[sa[0]] = p++;
        rep(i,len-1) {
            if(sa[i]+d < len && sa[i+1]+d <len &&r[sa[i]]
== r[sa[i+1]]&& r[sa[i]+d] == r[sa[i+1]+d])
                k[sa[i+1]] = p - 1;
            else k[sa[i+1]] = p++;
        }
        if(p >= len) return ;
        d *= 2,up = p, p = 0;
    }
}
void getHeight(int len) {
    rep(i,len) rk[sa[i]] = i;
    height[0] =  0;
    for(int i = 0,p = 0; i < len - 1; i++) {
        int j = sa[rk[i]-1];
        while(i+p < len&& j+p < len&& w[i+p] == w[j+p]) {
            p++;
        }
```

```
        height[rk[i]] = p;
        p = max(0,p - 1);
    }
}
int getSuffix(char s[]) {
    int len = strlen(s),up = 0;
    for(int i = 0; i < len; i++) {
        w[i] = s[i];
        up = max(up,w[i]);
    }
    w[len++] = 0;
    getSa(len,up+1);
    getHeight(len);
    return len;
}const int maxa = 100000*2+1;
char str[maxa];
int main(){
    while(scanf("%s", str)!=EOF){
        int l = strlen(str);
        str[l] = ' ';
        scanf("%s", str+l+1);
        getSuffix(str);
        int ans = 0;
        int L = strlen(str);
        for(int i = 1;i < L; i++){
            if((sa[i-1] < l && sa[i] > l) || (sa[i-1] > l
&& sa[i] < l)){
                ans = max(ans, height[i]);
            }
        }
        printf("%d\n", ans);
    }
}
/*
abcde
bcde
```

```
*/
```

POJ 1743

题意：给一串数字，求变化相同，且不重叠的最长字符串。

思路：变化相同就是将字符串 `s[i]` 变成 `s[i]-s[i-1]` 。

那么再求后缀数组的话 `height[i]` 代表的是两个长度是 `height[i]+1` 变化相等，而如果 `s[i]` 与 `s[j]` 间距是 `n` 的话那么他们在实际字符串中的间距也是 `n` ，所以如果两个地方的 `height` 最小值是n的话他们的间距应该是 `n+1` 才行。

二分答案的方法这里讲的很好http://blog.sina.com.cn/s/blog_6635898a0102e0me.html

```
#include<iostream>
#include<string.h>
#include<stdio.h>
using namespace std;

#define rep(i,n) for(int i = 0;i < n; i++)
using namespace std;
const int size  = 200005,INF = 1<<30;
int
rk[size],sa[size],height[size],w[size],wa[size],res[size]
;
void getSa (int len,int up) {
    int *k = rk,*id = height,*r = res, *cnt = wa;
    rep(i,up) cnt[i] = 0;
    rep(i,len) cnt[k[i] = w[i]]++;
    rep(i,up) cnt[i+1] += cnt[i];
    for(int i = len - 1; i >= 0; i--) {
        sa[--cnt[k[i]]] = i;
    }
    int d = 1,p = 0;
```

```
    while(p < len){
        for(int i = len - d; i < len; i++) id[p++] = i;
        rep(i,len)    if(sa[i] >= d) id[p++] = sa[i] - d;
        rep(i,len) r[i] = k[id[i]];
        rep(i,up) cnt[i] = 0;
        rep(i,len) cnt[r[i]]++;
        rep(i,up) cnt[i+1] += cnt[i];
        for(int i = len - 1; i >= 0; i--) {
            sa[--cnt[r[i]]] = id[i];
        }
        swap(k,r);
        p = 0;
        k[sa[0]] = p++;
        rep(i,len-1) {
            if(sa[i]+d < len && sa[i+1]+d <len &&r[sa[i]]
== r[sa[i+1]]&& r[sa[i]+d] == r[sa[i+1]+d])
                k[sa[i+1]] = p - 1;
            else k[sa[i+1]] = p++;
        }
        if(p >= len) return ;
        d *= 2,up = p, p = 0;
    }
}
void getHeight(int len) {
    rep(i,len) rk[sa[i]] = i;
    height[0] =  0;
    for(int i = 0,p = 0; i < len - 1; i++) {
        int j = sa[rk[i]-1];
        while(i+p < len&& j+p < len&& w[i+p] == w[j+p]) {
            p++;
        }
        height[rk[i]] = p;
        p = max(0,p - 1);
    }
}
int getSuffix(int s[], int n) {
```

```
    int len = n,up = 0;
    /*for(int i = 0;i  < len; i++){
        printf("%d ", s[i]);
    }puts("");*/
    for(int i = 0; i < len; i++) {
        w[i] = s[i];
        up = max(up,w[i]);
    }
    w[len++] = 0;
    getSa(len,up+1);
    getHeight(len);
    return len;
}const int maxa = 100000*2+1;
int str[maxa];
int a[maxa];
int judge(int ans, int n){
    int l = sa[0], r = sa[0];
    for(int i = 0;i <= n; i++){
        if(height[i] >= ans){
            l = min(l, sa[i]);
            r = max(r, sa[i]);
            if(r - l > ans)
                return 1;
        }
        else{
            l = r = sa[i];
        }
    }
    return 0;
}
int main(){
    int n;
    while(scanf("%d", &n)!=EOF){
        if(n == 0)return 0;
        for(int i = 0; i < n; i++){
            scanf("%d", &a[i]);
```

```
        }
        /*a[n] = a[n-1];
        n++;*/
        for(int i = 0; i < n-1; i++){
            str[i] = a[i+1] - a[i] + 100;
        }
        str[n-1] = 0;
        getSuffix(str, n-1);
        int l = 0, r = n-1;
        while(l < r){
            int mid = (l+r) / 2;
            if(judge(mid, n-1)) l = mid+1;
            else r = mid ;
        }
        //printf("%d\n" , l);
        if(l < 5){
            printf("0\n");
        }else{
            printf("%d\n", l);
        }
    }
}
/*
abcde
bcde
*/
```

KMP

KMP是一种在线性时间内能处理两个字符串的包含关系的算法，例如求一个字符串里有没有另一个字符串，一个字符串里有几个另一个字符串（可重叠和不可重叠两种）。

贴个讲kmp的链接 http://blog.csdn.net/yutianzuijin/article/details/11954939

模板1(可重叠时)

```
#include<stdio.h>
#include<string.h>
void makeNext(const char P[],int next[])
{
    int q,k;
    int m = strlen(P);
    next[0] = 0;
    for (q = 1,k = 0; q < m; ++q)
    {
        while(k > 0 && P[q] != P[k])
            k = next[k-1];
        if (P[q] == P[k])
        {
            k++;
        }
        next[q] = k;
    }
}

int kmp(const char T[],const char P[],int next[])
{
    int n,m;
    int i,q;
    n = strlen(T);
```

```
    m = strlen(P);
    makeNext(P,next);
    for (i = 0,q = 0; i < n; ++i)
    {
        while(q > 0 && P[q] != T[i])
            q = next[q-1];
        if (P[q] == T[i])
        {
            q++;
        }
        if (q == m)
        {
            printf("Pattern occurs with shift:%d\n",(i-
m+1));
        }
    }
}

int main()
{
    int i;
    int next[20]={0};
    char T[] = "ababxbababcadfdsss";
    char P[] = "abcdabd";
    printf("%s\n",T);
    printf("%s\n",P );
    // makeNext(P,next);
    kmp(T,P,next);
    for (i = 0; i < strlen(P); ++i)
    {
        printf("%d ",next[i]);
    }
    printf("\n");

    return 0;
}
```

模板2(不可重叠时)

```
#include<stdio.h>
#include<string.h>
void makeNext(const char P[],int next[])
{
    int q,k;
    int m = strlen(P);
    next[0] = 0;
    for (q = 1,k = 0; q < m; ++q)
    {
        while(k > 0 && P[q] != P[k])
            k = next[k-1];
        if (P[q] == P[k])
        {
            k++;
        }
        next[q] = k;
    }
}

int kmp(const char T[],const char P[],int next[])
{
    int n,m;
    int i,q;
    n = strlen(T);
    m = strlen(P);
    makeNext(P,next);
    for (i = 0,q = 0; i < n; ++i)
    {
        while(q > 0 && P[q] != T[i])
            q = next[q-1];
        if (P[q] == T[i])
        {
```

```
            q++;
        }
        if (q == m)
        {
            printf("Pattern occurs with shift:%d\n",(i-
m+1));
            q = 0;
        }
    }
}

int main()
{
    int i;
    int next[20]={0};
    char T[] = "abababababa";
    char P[] = "aba";
    printf("%s\n",T);
    printf("%s\n",P );
    // makeNext(P,next);
    kmp(T,P,next);
    for (i = 0; i < strlen(P); ++i)
    {
        printf("%d ",next[i]);
    }
    printf("\n");

    return 0;
}
```

例题

POJ 1961

题目大意，求这个字符串到 `i` 为止有多少个循环串；

```
int k = i-next[i];
if((i+1)%k == 0 && (i+1)!= k)
+1, (i+1)/k);
```

例如一个字符串的第99为指向第96位，也就是说后4-99位和前1-96位是匹配的，就是说94到96与97到99是是匹配的，而且91-94与94-96是匹配的。

一直可以推到最前面。

代码

```
#include <iostream>
#include <string.h>
#include <map>
#include <stdio.h>
using namespace std;
const int maxa =1000005;
int next[maxa];
int vis[maxa];
int n;
void init_kmp(char str[])
{
    memset(vis, 0, sizeof(vis));
    next[0]=-1;
    for(int i=1; str[i]!=0; i++)
    {
        int j= next[i-1];
        while(str[j+1]!=str[i]&&j>=0)
            j= next[j];
        if(str[j+1] == str[i])
            next[i] = j + 1;
        else
            next[i] = -1;
    }
}
```

```
int main()
{
    char str[maxa];
    int d =1;
    while(scanf("%d", &n), n)
    {
        scanf("%s", str);
        printf("Test case #%d\n", d++);
        init_kmp(str);
        for(int i = 0; i < n; i++)
        {
            int k = i-next[i];
            if((i+1)%k == 0 && (i+1)!= k)
                printf("%d %d\n", i+1, (i+1)/k);
        }
        printf("\n");
    }
}
```

AC自动机

算法简述

ac自动机是一类在字典树上进行kmp的算法。

学习ac自动机一般考点都在于对字典树上的点做处理，一般考的都是在自动机上的dp，而这类的dp中几乎必定有一维表示的是自动机上的点。

模板

```
#include<iostream>
#include<string.h>
#include<stdio.h>
#include<algorithm>
#include<queue>
using namespace std;
const int maxa = 500000;
const int cha = 26;
int n, m, k;
struct Tire{
    int next[maxa][cha], fail[maxa], end[maxa];
    int root, L;
    int newnode(){
        for(int i = 0; i < cha; i++){
            next[L][i] = -1;
        }
        end[L++] = 0;
        return L-1;
    }
    void init(){
        L = 0;
        root = newnode();
```

```
    }
    void insert(char buf[]){
        int len = strlen(buf);
        int now = root;
        for(int i = 0; i < len; i++){
            if(next[now][buf[i] - 'a'] == -1)
                next[now][buf[i]-'a'] = newnode();
            now = next[now][buf[i]-'a'];
            //printf("%d ", now);
        }//puts("");
        end[now] ++;
    }
    void build(){
        queue<int>Q;
        fail[root] = root;
        for(int i = 0; i < cha; i++){
            if(next[root][i] == -1)
                next[root][i] = root;
            else{
                fail[next[root][i]] = root;
                Q.push(next[root][i]);
            }
        }
        while(!Q.empty()){
            int now = Q.front();
            Q.pop();
           // end[now] |= end[fail[now]];
            for(int i = 0; i < cha; i++){
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i];
                else{
                    fail[next[now][i]] = next[fail[now]]
[i];
                    Q.push(next[now][i]);
                   // printf("**%d %d\n",next[now]
[i],next[fail[now]][i]);
```

```
                }
            }
        }
    }
    int solve(char *s){
        int ans = 0, k = 0;
        for(int i = 0; s[i]; i++){
            int t = s[i] - 'a';
            k = next[k][t];
            int j = k;
            while(j){
                ans += end[j];
                //if(end[j]) printf("%d ",j);
                end[j] = 0;
                j = fail[j];
            }//puts("");
        }
        return ans;
    }
};
char buf[1000005];
Tire ac;
int main(){
    int t, n;
    scanf("%d", &t);
    while(t--){
        scanf("%d", &n);
        ac.init();
        //memset(ac.end, 0, sizeof(ac.end));
        for(int i = 0; i < n; i++){
            scanf("%s", buf);
            ac.insert(buf);
        }
        ac.build();
        scanf("%s", buf);
        printf("%d\n", ac.solve(buf));
```

```
    }
}
/*
abcdefg
bcdefg
cdef
de
e
ssaabcdefg
*/
```

HDU 3247

题目

给出 $$n$$ 个资源，$$m$$ 个病毒，将资源串拼接成一个串，必须包含所有的资源串，可以重叠，但是不能包含病毒。

思路

用**ac**自动机预处理广搜出所有文本串到文本串的最短安全距离（即不通过病毒串）之后用状压dp求出答案。

难点

难点在于繁琐了一些，bfs加ac自动机加状压dp，不过知道每步都细心一点其实也没什么问题。
关键点，要对自动机了解的深入一些才能做出来，**ac**自动机往往需要对模板进行改动，只会套模板没什么用。

```
#include<iostream>
#include<string.h>
#include<stdio.h>
#include<algorithm>
#include<queue>
```

```
using namespace std;
const int maxa = 60000;
const int cha = 2;
int dp[10][1024];
int dis[10][maxa];
int point[10];
int n, m, k;
int leng[10];
struct Tire{
    int next[maxa][cha], fail[maxa], end[maxa];
    int root, L;
    int newnode(){
        for(int i = 0; i < cha; i++){
            next[L][i] = -1;
        }
        end[L++] = 0;
        return L-1;
    }
    void init(){
        L = 0;
        root = newnode();
    }
    int insert(char buf[], int ii){
        int len = strlen(buf);
        int now = root;
        for(int i = 0; i < len; i++){
            int x = buf[i] -'0';
            if(next[now][x] == -1)
                next[now][x] = newnode();
            now = next[now][x];
            //printf("%d ", now);
        }//puts("");
        end[now] = ii;
        return now;
    }
    void build(){//printf("build\n");
```

```
        queue<int>Q;
        fail[root] = root;
        for(int i = 0; i < cha; i++){
            if(next[root][i] == -1)
                next[root][i] = root;
            else{
                fail[next[root][i]] = root;
                Q.push(next[root][i]);
            }
        }
        while(!Q.empty()){
            int now = Q.front();
            Q.pop();
            end[now] |= end[fail[now]];
            for(int i = 0; i < cha; i++){
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i];
                else{
                    fail[next[now][i]] = next[fail[now]]
[i];
                    Q.push(next[now][i]);
                   // printf("**%d %d\n",next[now]
[i],next[fail[now]][i]);
                }
            }
        }
    }
    void bfs(int ii){//printf("bfs\n");
        queue<int> que;
        int vis[maxa];
        memset(vis, 0, sizeof(vis));
        que.push(point[ii]);
        vis[0] = 1;
        dis[ii][point[ii]] = 0;
        while(!que.empty()){
            int now = que.front(); que.pop();//printf("%d
```

```
%d\n", dis[ii][now], now);
            for(int i =0; i < 2; i++){
                int Next = next[now][i];
                if(vis[Next] == 0 && end[Next] != -1){
                    dis[ii][Next] = dis[ii][now] +1;
                    vis[Next] = 1;
                    que.push(Next);
                }
                /*Next = fail[Next];
                while(Next){
                    if(vis[Next] == 0 && end[Next] != -1)
{
                        dis[ii][Next] = dis[ii][now] +1;
                        vis[Next] = 1;
                        que.push(Next);
                    }
                    Next = fail[Next];
                }*/
            }
        }
    }
    int solve(){
        memset(dis, -1, sizeof(dis));
        for(int i = 0;i < n; i++){
            bfs(i);
        }//printf("solve%d %d\n", dis[0][point[1]],
dis[1][point[0]]);//printf("%d %d\n", point[0],
point[1]);
        for(int i =0 ;i  < n; i++){
            for(int k = 0; k < (1<<n); k++){
                dp[i][k] = 10000000;
            }
        }
        for(int i = 0; i < n; i++){
            dp[i][(1<<i)] = leng[i];
        }
```

```
        for(int i =1 ;i < n; i++){
            for(int k = 0; k < n; k++){
                for(int j = 0; j < (1<<n); j++){
                    if(dp[k][j] < 10000000){
                        for(int h = 0; h < n; h++){
                            if(!(j&(1<<h)) && dp[k]
[j]!=-1){
                                dp[h][j|(1<<h)] =
min(dp[h][j|(1<<h)], dp[k][j] + dis[k][point[h]]);
                            }
                        }
                    }
                }
            }
        }
            int ans = 10000000;
        for(int i = 0; i < n; i++){
            ans = min(ans, dp[i][(1<<n)-1]);
        }
        return ans;
    }
};
char buf[1000005];
Tire ac;
int main(){
    int m;
    while(scanf("%d%d", &n, &m), n+m){
        ac.init();
        for(int i =0 ;i  < n; i++){
            scanf("%s", buf);
            leng[i] = strlen(buf);
            point[i]=ac.insert(buf, 1+i);
        }
        for(int i =0; i < m; i++){
            scanf("%s", buf);
            ac.insert(buf, -1);
```

```
        }
        ac.build();
        printf("%d\n", ac.solve());
    }
}
/*
abcdefg
bcdefg
cdef
de
e
ssaabcdefg
*/
```

LightOJ 1427

题目大意

给一个长度小于 $$10^6$$ 的字符串以及小于 500 个长度小于 500 的字符串，问每个字符串在大字符串中出现的次数。

分析

基础ac自动机问题，对模板做一些小修改就可以了。需要注意的是字符串有可能重复。
这是比较裸的思路，他的时间主要取决于主串的长度。

代码

```
#include<iostream>
#include<string.h>
#include<stdio.h>
#include<algorithm>
#include<queue>
```

```
using namespace std;
const int maxa = 500000;
const int cha = 26;
int n, m, k;
int ans[505];
struct Tire{
    int next[maxa][cha], fail[maxa], end[maxa];
    int root, L;
    int newnode(){
        for(int i = 0; i < cha; i++){
            next[L][i] = -1;
        }
        end[L++] = 0;
        return L
    }
    void init(){
        L = 0;
        root = newnode();
    }
    void insert(char buf[], int ii){
        int len = strlen(buf);
        int now = root;
        for(int i = 0; i < len; i++){
            if(next[now][buf[i] - 'a'] == -1)
                next[now][buf[i]-'a'] = newnode();
            now = next[now][buf[i]-'a'];
            //printf("%d ", now);
        }//puts("");
        ans[ii] = now;
    }
    void build(){
        queue<int>Q;
        fail[root] = root;
        for(int i = 0; i < cha; i++){
            if(next[root][i] == -1)
                next[root][i] = root;
```

```
            else{
                fail[next[root][i]] = root;
                Q.push(next[root][i]);
            }
        }
        while(!Q.empty()){
            int now = Q.front();
            Q.pop();
           // end[now] |= end[fail[now]];
            for(int i = 0; i < cha; i++){
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i];
                else{
                    fail[next[now][i]] = next[fail[now]]
[i];
                    Q.push(next[now][i]);
                   // printf("**%d %d\n",next[now]
[i],next[fail[now]][i]);
                }
            }
        }
    }
    void solve(char *s){
        //memset(ans, 0, sizeof(ans));
        int k = 0;
        for(int i = 0; s[i]; i++){
            int t = s[i] - 'a';
            k = next[k][t];
            int j = k;
            while(j){
                end[j]++;
                j = fail[j];
            }//puts("");
        }
        return ;
    }
```

```
};
char buf[1000005], buf1[1000005];
Tire ac;
int main(){
    int t, n;
    scanf("%d", &t);
    for(int cas = 1; cas <= t; cas++){
        scanf("%d", &n);
        ac.init();
        //memset(ac.end, 0, sizeof(ac.end));
        scanf("%s", buf1);
        for(int i = 0; i < n; i++){
            scanf("%s", buf);
            ac.insert(buf, i+1);
        }
        ac.build();
        ac.solve(buf1);
        printf("Case %d:\n", cas);
        for(int i = 1; i <= n; i++){
            printf("%d\n", ac.end[ans[i]]);
        }
    }
}
/*
abcdefg
bcdefg
cdef
de
e
ssaabcdefg
*/
```

这是改进过的，时间取决于树的规模，只考虑solve的过场的话时间至少优化掉四倍，从最终跑的时间来看也是这样的。

思路是在build树的时候将树上的每个节点按bfs的顺序存至qq数组中，遍历到树上某点的时候仅将以当前点为结尾的字符串数量加一，然后将qq数组逆序边历，遍历到j点的时候end[fail[j]] += end[j];

```
#include<iostream>
#include<string.h>
#include<stdio.h>
#include<algorithm>
#include<queue>
using namespace std;
const int maxa = 500000;
const int cha = 26;
int n, m, k;
int ans[505];
struct Tire{
    int next[maxa][cha], fail[maxa], end[maxa];
    int qq[maxa];
    int pos[maxa], tt;
    int root, L;
    int newnode(){
        for(int i = 0; i < cha; i++){
            next[L][i] = -1;
        }
        end[L++] = 0;
        return L-1;
    }
    void init(){
        tt = 0;
        L = 0;
        root = newnode();
    }
    void insert(char buf[], int ii){
        int len = strlen(buf);
        int now = root;
        for(int i = 0; i < len; i++){
            if(next[now][buf[i] - 'a'] == -1)
```

```
                next[now][buf[i]-'a'] = newnode();
            now = next[now][buf[i]-'a'];
            //printf("%d ", now);
        }//puts("");
        pos[ii] = now;
    }
    void build(){
        queue<int>Q;
        fail[root] = root;
        for(int i = 0; i < cha; i++){
            if(next[root][i] == -1)
                next[root][i] = root;
            else{
                qq[tt++] = next[root][i];
                fail[next[root][i]] = root;
                Q.push(next[root][i]);
            }
        }
        while(!Q.empty()){
            int now = Q.front();
            Q.pop();
           // end[now] |= end[fail[now]];
            for(int i = 0; i < cha; i++){
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i];
                else{
                    qq[tt++] = next[now][i];
                    fail[next[now][i]] = next[fail[now]]
[i];
                    Q.push(next[now][i]);
                   // printf("**%d %d\n",next[now]
[i],next[fail[now]][i]);
                }
            }
        }
    }
```

```
    void solve(char *s, int n){
        memset(ans, 0, sizeof(ans));
        int k = 0;
        for(int i = 0; s[i]; i++){
            int t = s[i] - 'a';
            k = next[k][t];
            end[k] ++;
        }
        for(int i = tt-1; i >= 0; i--){
            int j = qq[i];
                end[fail[j]] += end[j];
        }
        /*printf("*");
        for(int i = 0; i < L; i++){
            printf("%d ", end[i]);
        }puts("");*/
        for(int i = 1; i <= n; i++){
            printf("%d\n", end[pos[i]]);
        }
        return ;
    }
};
char buf[1000005], buf1[1000005];
Tire ac;
int main(){
    int t, n;
    scanf("%d", &t);
    for(int cas = 1; cas <= t; cas++){
        scanf("%d", &n);
        ac.init();
        //memset(ac.end, 0, sizeof(ac.end));
        scanf("%s", buf1);
        for(int i = 0; i < n; i++){
            scanf("%s", buf);
            ac.insert(buf, i+1);
        }
```

```
        ac.build();
        printf("Case %d:\n", cas);
        ac.solve(buf1, n);
    }
}
/*
abcdefg
bcdefg
cdef
de
e
ssaabcdefg
*/
```

最长回文子串

有这样一种题，求一个字符串的最长回文子串的长度。

hdu 3608就是求最长回文子串的一道题

有这样几种解法

暴力法

枚举所有子串且每次判断该子串是否是回文子串，时间复杂度是 $$O(n^3)$$,理所当然的T掉。

```
#include<iostream>
#include<stdio.h>
#include<string.h>
using namespace std;
const int maxa = 110000;
char str[maxa];
int main(){
    while(scanf("%s", &str)!=EOF){
        int ans = 0;
        for(int i = 0; str[i]; i++){
            for(int k = i+1; str[k]; k++){
                int ok = 1;
                for(int j = i; j <= (i+k)/2; j++){
                    if(str[j] != str[k-(j-i)]){
                        ok = 0;
                        break;
                    }
                }
                if(ok){
                    ans = max(ans, k-i+1);
                }
            }
        }
        printf("%d\n", ans);
    }
}
```

枚举中心法

首先将字符串每个字符的两遍都加上特殊字符，变成新串，例如原串为 `abab` ，新串为 `#a#b#a#b#` ，然后以每个点为中心，求出以该点为中心的最长回文串的长度，枚举中心的时间复杂度是 $$O(n)$$，枚举最长长度的时间复杂度是 $$O(n)$$，总时间复杂度是 $$O(n^2)$$，依旧超时。

```
#include<iostream>
#include<string.h>
#include<stdio.h>
using namespace std;
const int maxa = 111111;
char a[maxa];
char str[maxa*2];
int main(){
    while(scanf("%s", &a)!=EOF){
        int ans = 0;
        for(int i = 0; ; i++){
            if(a[i]){
                str[i*2] = '#';
                str[i*2+1] = a[i];
            }else{
                str[i*2] = '#';
                str[i*2+1] = 0;
                break;
            }
        }
        printf("%s\n", str);
        int len = strlen(str);
        for(int i = 0; str[i]; i++){
            for(int j = 0; i-j >= 0 && i+j < len; j++){
                if(str[i-j] == str[i+j]){
                    ans = max(ans, j);
                }else break;
            }
        }
        printf("%d\n", ans);
    }
}
```

动态规划

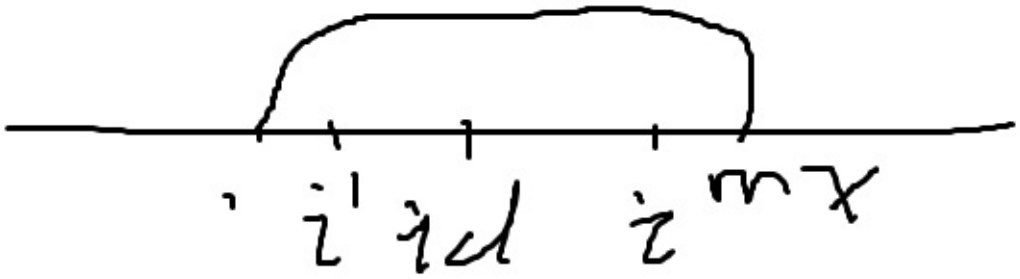
for循环枚举所有子串，当 $$str[i \ldots j-1]$$ 被枚举到的时候，$$str[i+1 \ldots j-1]$$ 也一定被枚举到，`dp[i][j]=1` 代表 $$str[i \ldots j]$$ 是回文子串，只要 `str[i] == str[j]` 并且，`dp[i+1][j-1]==1` ，那么 `dp[i][j]=1` ，时间复杂度为 $$O(n^2)$$，空间复杂度也是 $$O(n^2)$$.

```
#include<iostream>
#include<string.h>
#include<stdio.h>
using namespace std;
const int maxa = 1111;
char str[maxa];
int dp[maxa][maxa];
int main(){
    while(scanf("%s", &str)!=EOF){
        memset(dp, 0, sizeof(dp));
        int l = strlen(str);
        for(int i = 0; i < l; i++){
            dp[i][i] = 1;
        }
        int ans = 0;
        for(int j = 1; j < l; j++){

            for(int i = 0; str[i]; i++){
                if(i+j > l) break;
                int k = i+j;
                if(str[i] == str[k]){
                    if(i+1 > k-1 || dp[i+1][k-1]){
                        dp[i][k] = 1;
                        ans = max(ans, k-i+1);
                    }
                }
            }
        }
        printf("%d\n", ans);
    }
}
```

Manacher算法

将字符串中的每个字符的左右都加上一个串中不会出现的特殊字符，例如字符串为 `abcde` ，那么变换后的串为 `#a#b#c#d#e#` 。我们称变换后的串为 `str2` 。

另 `p[i]` 为 `str2` 中第 `i` 个字符为中心的最长回文串的边界到中心的距离， `i` 加上 `p[i]` ,就是以 `i` 为中心的最长回文子串的右边界，用两个辅助变量分别是 `id` ，和 `mx` 。 `mx` 代表之前所有回文串最右右边界， `id` 代表那最右右边界的点。



如上图所示 `i'` 代表 `i` 在 `id` 的对应左侧对应位置,如果以 `i` 为中心的回文串的边界不超过以 `id` 为中心的回文串的范围的话那么 `p[i] == p[i']` ，如果超过的话 `p[i] == 2*id-i` ，接下来在以为以为扩张 `p[i]` 。

由于每次扩张都会导致 `mx` 变大，而 `mx` 最大不会超过 `strlen(str2)` ，所以时间复杂度为 $$O(n)$$ 。

```
#include<iostream>
#include<stdio.h>
#include<string.h>
using namespace std;
const int maxa = 111111*2;
int p[maxa*2];
#define max(a,b) a>b?a:b;
#define min(a,b) a<b?a:b;
int rebuild(int n, char* a, char* str){
    for(int i = 0 ;i < n; i++){
        a[i*2+1] = '#';
        a[i*2+2] = str[i];
    }
```

```
    a[2*n+1] = '#';
    a[2*n+2] = 0;
    a[0] = '$';
    return 2*n+2;
}
int manachar(int n, char* a){
    int mx = 0, id = 0;
    int ans = 0;
   // printf("%s\n", a);
    for(int i = 0; i < n; i++){
        if(mx > p[i]){
            int i2 = 2*id - i;
            p[i] = min(mx - i, p[i2]);
        }else p[i] = 0;
        while(a[i-p[i]-1] == a[i+p[i]+1]){
            p[i]++;
        }
        ans = max(ans, p[i]);
        if(p[i] +i > mx){
            id = i;
            mx = p[i]+i;
        }
    }
    return ans;
}

char str[maxa*2];
char a[maxa*2];

int main(){
    while(scanf("%s", &str)!=EOF){
        int n = strlen(str);
        n = rebuild(n,a, str);
       // printf("fuck");
        printf("%d\n", manachar(n, a));
    }
```

```
}
```

后缀数组

将字符串倒置接在原来字符串后，两个字符串之间用特殊字符分割，例如原串为 `abc` ，新串即为 `abc#cba` 。

我们将新串的前半部分称为 `str1` ,后半部分称为 `str2` 。

此时分两种情况，一种是求最长奇回文串，一种是求最长偶回文串。

1. 求最长奇回文串。枚举以每个字符为中心时的奇回文长度，只要求出第 `i` 个字符在 `str2` 中的位置，在求出两点的 `rank` ，然后求出两点 `rank` 之间的最小 `height` ，长度乘2减1就是以此点为中心的最长奇回文的长度。
2. 求最长偶回文，求出某一节点在 `str2` 中对应的位置的下一个节点，同（1）步骤求出的值*2即为以此点为中心右侧的偶回文的长度。

时间复杂度是 $$O(n\log n)$$。

```
#include<iostream>
#include<string.h>
#include<stdio.h>
using namespace std;

#define rep(i,n) for(int i = 0;i < n; i++)
using namespace std;
const int size  = 222222,INF = 1<<30;
int
rk[size],sa[size],height[size],w[size],wa[size],res[size]
;
void getSa (int len,int up) {
    int *k = rk,*id = height,*r = res, *cnt = wa;
    rep(i,up) cnt[i] = 0;
    rep(i,len) cnt[k[i] = w[i]]++;
    rep(i,up) cnt[i+1] += cnt[i];
    for(int i = len - 1; i >= 0; i--) {
        sa[--cnt[k[i]]] = i;
```

```
    }
    int d = 1,p = 0;
    while(p < len){
        for(int i = len - d; i < len; i++) id[p++] = i;
        rep(i,len)    if(sa[i] >= d) id[p++] = sa[i] - d;
        rep(i,len) r[i] = k[id[i]];
        rep(i,up) cnt[i] = 0;
        rep(i,len) cnt[r[i]]++;
        rep(i,up) cnt[i+1] += cnt[i];
        for(int i = len - 1; i >= 0; i--) {
            sa[--cnt[r[i]]] = id[i];
        }
        swap(k,r);
        p = 0;
        k[sa[0]] = p++;
        rep(i,len-1) {
            if(sa[i]+d < len && sa[i+1]+d <len &&r[sa[i]]
== r[sa[i+1]]&& r[sa[i]+d] == r[sa[i+1]+d])
                k[sa[i+1]] = p - 1;
            else k[sa[i+1]] = p++;
        }
        if(p >= len) return ;
        d *= 2,up = p, p = 0;
    }
}
void getHeight(int len) {
    rep(i,len) rk[sa[i]] = i;
    height[0] =  0;
    for(int i = 0,p = 0; i < len - 1; i++) {
        int j = sa[rk[i]-1];
        while(i+p < len&& j+p < len&& w[i+p] == w[j+p]) {
            p++;
        }
        height[rk[i]] = p;
        p = max(0,p - 1);
    }
```

```
}
int getSuffix(char s[]) {
    int len = strlen(s),up = 0;
    for(int i = 0; i < len; i++) {
        w[i] = s[i];
        up = max(up,w[i]);
    }
    w[len++] = 0;
    getSa(len,up+1);
    getHeight(len);
    return len;
}const int maxa = 222222;
char str[maxa];
int rmp[maxa][32];
int log(int n){
    int cnt = 0;
    while(n){
        cnt ++;
        n /= 2;
    }
    return cnt - 1;
}
int RMQ(int n){
    for(int i = 0;i < n; i++){
        rmp[i][0] = height[i];
    }
    int l = log(n);
    for(int i = 1; i < l; i++){
        for(int j = 0; j+(1<<(i-1)) < n; j++){
            rmp[j][i] = min(rmp[j][i-1], rmp[j+(1<<(i-
1))][i-1]);
        }
    }
}
int r1r2(int a, int b){
    int j = log(b-a+1);
```

```
    return min(rmp[a][j], rmp[b-(1<<j)+1][j]);

}
int main(){
    while(scanf("%s", str)!=EOF){
        int l = strlen(str);
        str[l] = '#';
        for(int i = 0; i < l; i++){
            str[l+1+i] = str[l-1-i];
        }
        str[2*l+1] = 0;
        //printf("%s\n", str);
        getSuffix(str);
        int n = strlen(str);
        RMQ(n);
        int ans = 0;
        for(int i= 0; str[i] != '#'; i++){
            int i1 =  2*l-i;
            //printf("%d == i %d == i1\n", i, i1);
           // printf("%d %d\n", rk[i], rk[i1]);
            int a = rk[i], b = rk[i1];
           // printf("%d ", 2*r1r2(min(a, b) + 1, max(a,
b))-1);
            ans = max(ans, 2*r1r2(min(a, b) + 1, max(a,
b))-1);
            i1++;
            a = rk[i], b = rk[i1];
            ans = max(ans, 2*r1r2(min(a, b) + 1, max(a,
b)));
           // printf("%d\n", 2*r1r2(min(a, b) + 1, max(a,
b)));
        }
        printf("%d\n", ans);
    }
}
```

数论

---

author：高放

中国剩余定理

> 今有物不知其数,三三数之剩二,五五数之剩三,七七数之剩二,问物几何?——《孙子算经》

这是最早的关于中国剩余定理的研究。 中国剩余定理基本不会考到，但是学习证明的过程会对群的理解很有帮助。

模板

```
LL ex_crt(LL *m, LL *r, int n)
{
    LL M = m[1], R = r[1], x, y, d;
    for (int i = 2; i <= n; ++i)
    {
        ex_gcd(M, m[i], d, x, y);
        if ((r[i] - R) % d) return -1;
        x = (r[i] - R) / d * x % (m[i] / d);
        R += x * M;
        M = M / d * m[i];
        R %= M;
    }
    return R > 0 ? R : R + M;
}
```

例题

POJ 2891 就是一道标准的中国剩余定理题目，首先判断所有 `m[i]` 是否互质，随后套用模板即可。

```
#include<iostream>
#include<string>
#include<algorithm>
```

```
#include<cstdio>
#include<cstring>
#include<cmath>
#include<queue>
#include<map>
#include<stack>
#include<set>
#include<cstdlib>

using namespace std;

#define LL long long
const int inf = 0x3f3f3f3f;
const int maxn = 1e5 + 5;
int n;
void ex_gcd(LL a, LL b, LL &d, LL &x, LL &y)
{
    if (!b) {d = a, x = 1, y = 0;}
    else
    {
        ex_gcd(b, a % b, d, y, x);
        y -= x * (a / b);
    }
}
LL ex_crt(LL *m, LL *r, int n)
{
    LL M = m[1], R = r[1], x, y, d;
    for (int i = 2; i <= n; ++i)
    {
        ex_gcd(M, m[i], d, x, y);
        if ((r[i] - R) % d) return -1;
        x = (r[i] - R) / d * x % (m[i] / d);
        R += x * M;
        M = M / d * m[i];
        R %= M;
    }
```

```
    return R > 0 ? R : R + M;
}
int main()
{
    while (~scanf("%d",&n))
    {
        LL m[maxn], r[maxn];
        for (int i = 1; i <= n; ++i)
            scanf("%lld%lld", &m[i], & r[i]);
        printf("%lld\n",ex_crt(m,r,n));
    }
    return 0;
}
```

扩展欧几里得

扩展欧几里得的目的就是求乘法逆元，想要学扩展欧几里得首先要了解群论，了解群论，了解群论！ 重要的事情说三遍。

模板

```
long long ex_gcd(long long a, long long b, long long &x,
long long &y){
    if(b == 0){
        x = 1, y= 0;
        return a;
    }else{
        long long r = ex_gcd(b, a% b, y, x);
        y -= x*(a/b);
        return r;
    }
}
```

POJ 1061 青蛙的约会

这道题涉及的点也是对群论的考察，了解群论的特性的话这道题就变成一道水题了

```
#include<iostream>
#include<string.h>
#include<stdio.h>
#include<algorithm>

using namespace std;
long long ex_gcd(long long a, long long b, long long &x,
long long &y){
    if(b == 0){
```

```
        x = 1, y= 0;
        return a;
    }else{
        long long r = ex_gcd(b, a% b, y, x);
        y -= x*(a/b);
        return r;
    }
}
int main(){
    long long x, y, m, n, l;
    while(scanf("%lld%lld%lld%lld%lld", &x, &y, &m, &n,
&l)!=EOF){
        long long a = (m - n+l)%l;
        long long b = (y - x + l ) % l;
        long long c = __gcd(a, l);
        if(b % c != 0){
            printf("Impossible\n");
            continue;
        }
        a = a / c; b = b / c;
        l /= c;
        //printf("%lld %lld %lld\n", a, b, l);
        ex_gcd(a, l, x, y);
        x = (x+l) % l;
        x*= b;
        x %= l;
        printf("%lld\n", x);
    }
}
```

素数筛

模板

Primel1是常用的素数筛模板，Primer2能在线性时间筛选出素数。
n是素数的范围，p数组保存的是素数。

```
#include <stdio.h>
#include <string.h>
using namespace std;
const int N = 25600000;
bool a[N];
int p[N];
int n;

void Prime1() {
    memset(a, 0, n * sizeof a[0]);
    int num = 0, i, j;
    for(i = 2; i < n; ++i) if(!a[i]) {
        p[num++] = i;
        for(j = i+i; j < n; j +=i) {
            a[j] = 1;
        }
    }

}

void Prime2() {
    memset(a, 0, n*sizeof a[0]);
    int num = 0, i, j;
    for(i = 2; i < n; ++i) {
        if(!(a[i])) p[num++] = i;
        for(j = 0; (j<num && i*p[j]<n); ++j) {
            a[i*p[j]] = 1;
            if(!(i%p[j])) break;
        }
    }
}
int main(){
    n = 100;
    Prime2();
}
```

计算几何

计算几何在ACM/ICPC竞赛的题目中属于较容易的内容。但计算几何往往代码量多，有时需要按照多种情况进行讨论，还有考虑到复杂的浮点数精度问题，所以常常容易卡题。所以计算几何对平时模板的积累就非常重要。

---

author：林凡卿

浮点数相关的陷阱

误差修正

简述

因为被计算机表示浮点数的方式所限制，CPU在进行浮点数计算时会出现误差。如执行 `0.1 + 0.2 == 0.3` 结果往往为 `false` ，在四则运算中，加减法对精度的影响较小，而乘法对精度的影响更大，除法的对精度的影响最大。所以，在设计算法时，为了提高最终结果的精度，要尽量减少计算的数量，尤其是乘法和除法的数量。

浮点数与浮点数之间不能直接比较，要引入一个 `eps` 常量。 `eps` 是epsilon（$$\varepsilon$$）的简写，在数学中往往代表任意小的量。在对浮点数进行大小比较时，如果他们的差的绝对值小于这个量，那么我们就认为他们是相等的，从而避免了浮点数精度误差对浮点数比较的影响。eps在大部分题目时取 `1e-8` 就够了，但要根据题目实际的内容进行调整。

模板代码

```
// sgn返回x经过eps处理的符号，负数返回-1，正数返回1，x的绝对值如果足够小，就返回0。
const double eps = 1e-8;
int sgn(double x) { return x < -eps ? -1 : x > eps ? 1 : 0; }
```

整型比较	等价的浮点数比较
`a == b`	`sgn(a - b) == 0`
`a > b`	`sgn(a - b) > 0`
`a >= b`	`sgn(a - b) >= 0`
`a < b`	`sgn(a - b) < 0`
`a <= b`	`sgn(a - b) <= 0`
`a != b`	`sgn(a - b) != 0`

输入输出

用 `scanf` 输入浮点数时，`double` 的占位符是 `%lf`，但是浮点数 `double` 在 `printf` 系列函数中的标准占位符是 `%f` 而不是 `%lf`，使用时最好使用前者，因为虽然后者在大部分的计算机和编译器中能得到正确结果，但在有些情况下会出错（比如在POJ上）。

开方

当提供给C语言中的标准库函数 `double sqrt (double x)` 的 `x` 为负值时，`sqrt` 会返回 `nan`，输出时会显示成 `nan` 或 `-1.#IND00`（根据系统的不同）。在进行计算几何编程时，经常有对接近零的数进行开方的情况，如果输入的数是一个极小的负数，那么 `sqrt` 会返回 `nan` 这个错误的结果，导致输出错误。解决的方法就是将 `sqrt` 包装一下，在每次开方前进行判断。

示例代码

```
double mysqrt(double x) { return max(0.0, sqrt(x)); }
```

负零

大部分的标程的输出是不会输出负零的，如下面这段程序：

```c
int main() {
    printf("%.2f\n", -0.0000000001);
    return 0;
}
```

会输出 `-0.00` 。有时这样的结果是错误的，所以在没有Special Judge的题目要求四舍五入时，不要忘记对负零进行特殊判断。

但有的标程也不会进行这样的特殊判断，所以在WA时不要放弃摸索。

向量

简介

向量，又称矢量，是既有大小又有方向的量，向量的长度即向量的大小称为向量的模。在计算几何中，从$$A$$指向$$B$$的向量记作$$\vec{AB}$$。$$n$$维向量可以用$$n$$个实数来表示。向量的基本运算包括加减法、数乘、点积、叉积和混合积。使用向量这一个基本的数据结构，我们可以用向量表示点和更复杂的各种图形。

注意事项

我们一般用一个二维向量来表示点。注意，在有些计算几何相关的题目中，坐标是可以利用整形储存的。在做这样的题目时，坐标一定要用整形变量储存，否则精度上容易出错。具体的将点的坐标用整形变量储存可以需要使用一些技巧，比如计算中计算平方或将坐标扩大二倍等方式。

```
// Pt是Point的缩写
struct Pt {
    double x, y;
    Pt() { }
    Pt(double x, double y) : x(x), y(y) { }
};

double norm(Pt p) { return sqrt(p.x*p.x + p.y*p.y); }
double dist (Pt a, Pt b) { return (a-b).norm(); }
void print(Pt p) { printf("(%f, %f)", p.x, p.y); }
```

基本计算

加减法

$$\vec{a} \pm \vec{b} = (a_x \pm b_x, a_y \pm b_y)$$

向量的加减法遵从平行四边形法则和三角形法则。

示例代码

```
Pt operator - (Pt a, Pt b) { return Pt(a.x - b.x, a.y -
b.y); }
Pt operator + (Pt a, Pt b) { return Pt(a.x + b.x, a.y +
b.y); }
```

长度

向量$$\vec{a}=(a_x, a_y)$$的长度是$$\sqrt{a_x^2+a_y^2}$$。

示例代码

```
double len(Pt p) { return sqrt(sqr(p.x)+sqr(p.y)); }
```

数乘

$$a\vec{b} = (a b_x, a b_y)$$。

向量的数乘是一个向量和实数的运算。$$a$$如果是零，那么结果是一个零向量，如果$$a$$是一个负数，那么结果向量会改变方向。

示例代码

```
Pt operator * (double A, Pt p) { return Pt(p.x*A, p.y*A);
}
Pt operator * (Pt p, double A) { return Pt(p.x*A, p.y*A);
}
```

点积

又称内积。

$$\vec{a} \cdot \vec{b} = a_xb_x + a_yb_y = |\vec{a}||\vec{b}|\cos\theta$$，其中$$\theta$$是$$\vec{a}$$与$$\vec{b}$$的夹角。

应用

点积可以用来计算两向量的夹角。

$$ \cos\beta = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}

$$

示例代码

```
double dot(Pt a, Pt b) { return a.x * b.x + a.y * b.y; }
```

叉积

叉积又称外积。叉积运算得到的是一个向量，它的大小是$$\vec{a}$$和$$\vec{b}$$所构成的平行四边形的面积，方向与$$\vec{a}$$和$$\vec{b}$$所在平面垂直，$$\vec{a}$$、$$\vec{b}$$与$$\vec{a} \times \vec{b}$$成右手系。

设两向量$$\vec{a}=(a_x, a_y)$$与$$\vec{b}=(b_x, b_y)$$，它们在二维平面上的的叉积为：

$$\vec{a} \times \vec{b} = a_xb_y - a_yb_x$$

示例代码

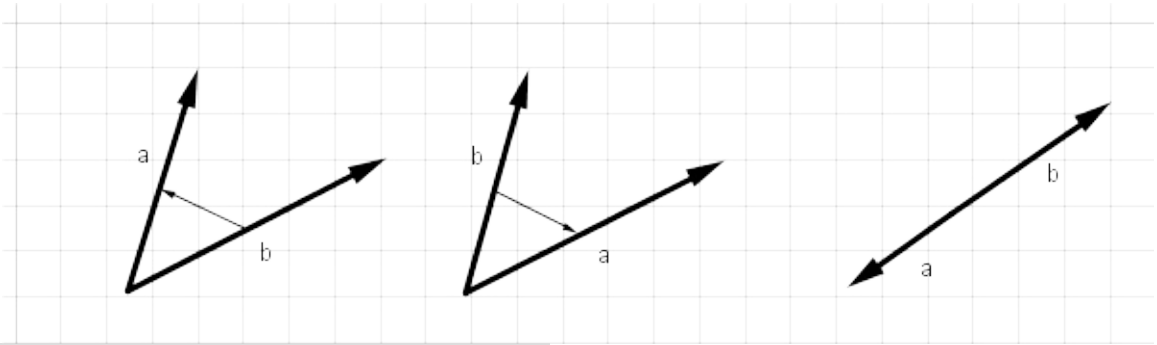
```
double det(Pt a, Pt b) { return a.x * b.y - a.y * b.x; }
```

性质与应用

叉积拥有两个重要的性质——面积与方向。

两向量叉积得到新向量的长度为这两个所构成的平行四边形的面积，利用这个性质我们可以求三角形的面积。

两向量叉积能反映出两向量方向的信息。如果$$\vec{a} \times \vec{b}$$的符号为正，那么$$\vec{b}$$在$$\vec{a}$$的逆时针方向；如果符号为负，那么$$\vec{b}$$在$$\vec{a}$$的顺时针方向；如果结果为零的话，那么$$\vec{a}$$与$$\vec{b}$$共线。



计算结果	**$$b$$**与**$$a$$**的方向		
$$	b \times a	> 0$$	$$a$$在$$b$$的逆时针方向
$$	b \times a	= 0$$	$$a$$与$$b$$共线
$$	b \times a	< 0$$	$$a$$在$$b$$的顺时针方向

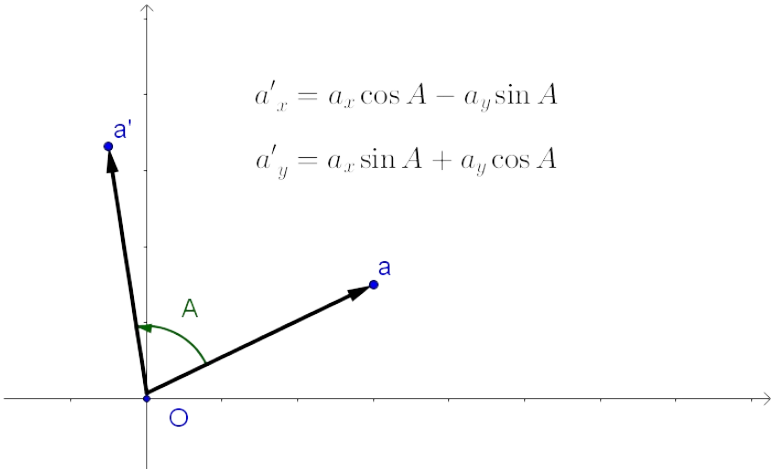
经典题目

- ZOJ 1010 Area

向量旋转

操作目的

将向量$$\vec{a}$$绕原点逆时针旋转$$A$$度。



模板代码

```
Pt rotate(Pt p, double a) {
    return Pt(p.x*cos(a) - p.y*sin(a), p.x*sin(a) +
p.y*cos(a));
}
```

线段

直线与线段的表示方法

我们可以用一条线段的两个端点来表示一条线段。直线的表示有两种方式，一种方式是使用二元一次方程$$y=kx+b$$来表示，另一种是用直线上任意一条长度不为零的线段来表示。由于使用方程表示接近垂直于某坐标轴的直线时容易产生精度误差，所以我们通常使用直线上的某条线段来表示直线。

```
struct Sg {
    Pt s, t;
    Sg() { }
    Sg(Pt s, Pt t) : s(s), t(t) { }
    Sg(double a, double b, double c, double d) : s(a, b),
t(c, d) { }
};
```

点在线段上的判断

判断点$$C$$在线段$$AB$$上的两条依据：

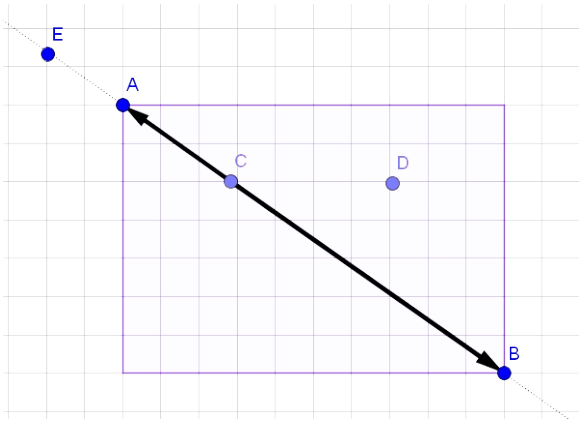
1. $$\vec{CA}\cdot\vec{CB} = 0$$。
2. $$C$$在以$$AB$$为对角顶点的矩形内。

示例代码

```
bool PtOnSegment(Pt s, Pt t, Pt a) {
    return !det(a-s, a-t) && min(s.x, t.x) <= a.x && a.x
<= max(s.x, t.x) &&
        min(s.y, t.y) <= a.y && a.y <= max(s.y, t.y);
}
```

另一种方法

判断点$$C$$在$$AB$$为对角线定点的矩形内较麻烦，可以直接判断$$\vec{CA}\cdot\vec{CB}$$的符号来判断$$C$$在直线$$AB$$上是否在$$AB$$之间。



示例代码

```
bool PtOnSegment(Pt p, Pt a, Pt b) {
    return !sgn(det(p-a, b-a)) && sgn(dot(p-a, p-b)) <=
0;
}
```

把上例代码中的 `<=` 改成 `<` 就能实现不含线段端点的点在线段上的判断。

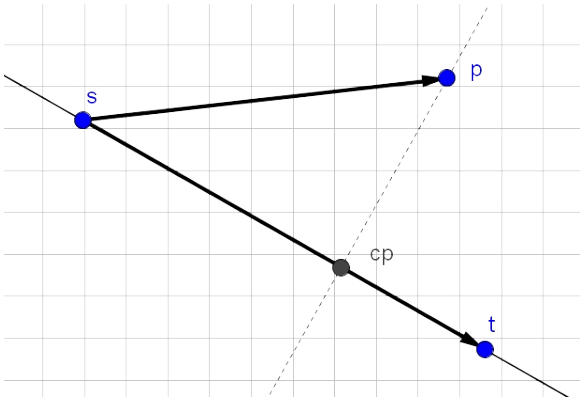
点在直线上的判断

点在直线上的判断很简单只要把点在线段上的判断的步骤2去掉即可。

示例代码

```
bool PtOnLine(Pt p, Pt s, Pt t) {
    return !sgn(det(p-a, b-a));
}
```

求点到直线的投影



示例代码

```
Pt PtLineProj(Pt s, Pt t, Pt p) {
    double r = dot(p-s, t-s) / (t - s).norm();
    return s + (t - s) * r;
}
```

判断直线关系

直线有相交和平行两种关系，靠叉乘能简单判断。

```
bool parallel(Pt a, Pt b, Pt s, Pt t) {
    return !sgn(det(a-b, s-t));
}
```

判断线段关系

线段有相交和不相交两种关系，通常按照以下步骤判断。

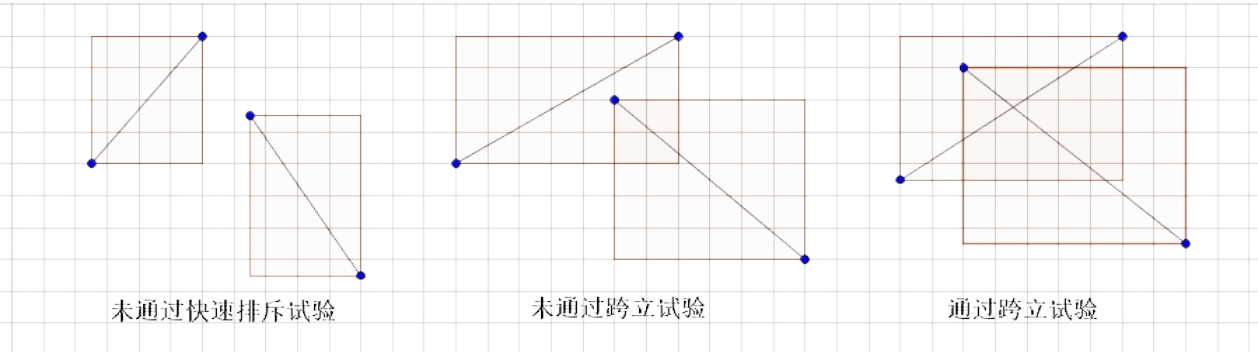
1. 快速排斥试验
2. 跨立试验

快速排斥试验

设以线段$$P_1P_2$$为对角线的矩形为$$R$$，设以线段$$Q_1Q_2$$为对角线的矩形为$$T$$，如果$$R$$和$$T$$不相交，显然两线段不会相交。

跨立试验

如果两线段相交，则两线段必然相互跨立对方。若$$P_1P_2$$跨立$$Q_1Q_2$$，则矢量$$\vec{Q_1P_1}$$和$$\vec{Q_1P_2}$$位于矢量$$\vec{Q_1Q_2}$$的两侧，即$$\vec{Q_1P_1} \times \vec{Q_1Q_2} \cdot \vec{Q_1Q_2} \times \vec{Q_1Q_2} < 0$$。上式可改写成$$\vec{Q_1P_1} \times \vec{Q_1Q_2} \cdot \vec{Q_1Q_2} \times \vec{Q_1P_2} > 0$$。当$$\vec{Q_1P_1} \times \vec{Q_1Q_2} = 0$$时，说明$$\vec{Q_1P_1}$$和$$\vec{Q_1Q_2}$$共线，但是因为已经通过快速排斥试验，所以$$P_1$$一定在线段$$Q_1Q_2$$上；同理，$$\vec{Q_1Q_2} \times \vec{Q_1P_2} = 0$$说明$$P_2$$一定在线段$$Q_1Q_2$$上。所以判断$$P_1P_2$$跨立$$Q_1Q_2$$的依据是：$$\vec{Q_1P_1} \times \vec{Q_1Q_2} \cdot \vec{Q_1Q_2} \times \vec{Q_1P_2} \geq 0$$。同理判断$$Q_1Q_2$$跨立$$P_1P_2$$的依据是：$$\vec{P_1Q_1} \times \vec{P_1P_2} \cdot \vec{P_1P_2} \times \vec{P_1Q_2} \geq 0$$。



经典题目

- ZOJ 1648 Circuit Board
- POJ 3304 Segments
- POJ 2653 Pick-up sticks

求点到线段的距离

求线段$$ab$$到点p最短距离的方法为：

根据点$$p$$到的投影点的位置进行判断的方法：

1. 判断线段$$pa$$和$$ab$$所成的夹角，如果是钝角，那么$$|pa|$$是点到线

段的最短距离。

2. 判断线段$$pb$$和$$ab$$所成的夹角，如果是钝角，那么$$|pb|$$是点到线段的最短距离。
3. 线段$$pa$$和线段$$pb$$与$$ab$$所成的夹角都不为钝角，那么点$$p$$到线段$$ab$$的距离是点$$p$$到直线$$ab$$的距离，这个距离可以用面积法直接算出来。

示例代码

```
double PtSegmentDist(Pt a, Pt b, Pt p) {
    if (sgn(dot(p-a, b-a)) <= 0) return (p-a).norm();
    if (sgn(dot(p-b, a-b)) <= 0) return (p-b).norm();
    return fabs(det(a-p, b-p)) / (a-b).norm();
}
```

经典题目

- URAL 1348 Goat in the Garden 2

三角形

三角形的面积

三角形的面积可以由叉积直接求出。

$$ S_{\triangle ABC} = |\frac{1}{2} \vec{AB} \times \vec{AC}|

$$

判断点在三角形内

判断点$$P$$在三角形 ABC 内部常用的又两种方法，面积法和叉积法。

面积法

$$ S{\triangle PAB} + S{\triangle PAC} + S{\triangle PBC} = S{\triangle ABC}

$$

叉积法

利用叉积的正负号判断，如图所示，$$AP$$在向量$$AC$$的顺时针方向，$$CP$$在向量$$BC$$的顺时针方向，$$BP$$在向量$$BC$$的顺时针方向，利用这一性质推广，那么可以利用叉积的正负号来判断一个点是否在一个凸多边形内部。

三角形的重心

三角形三条中线的交点叫做三角形重心。

性质

设三角形重心为$$O$$，$$BC$$边中点为$$D$$，则有$$AO = 2OD$$。

求重心的方法

三角形重心是三点坐标的平均值。

模板代码

```
Pt triangleMassCenter(Pt a, Pt b, Pt c) {
    return (a+b+c) / 3.0;
}
```

三角形的外心

三角形三边的垂直平分线的交点，称为三角形外心。

性质

外心到三顶点距离相等。过三角形各顶点的圆叫做三角形的外接圆，外接圆的圆心即三角形外心，这个三角形叫做这个圆的内接三角形。

```
Pt circumCenter(Pt a, Pt b, Pt c) {
    double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1*a1 +
b1*b1) / 2.0;
    double a2 = b.x - a.x, b2 = b.y - a.y, c2 = (a1*a1 +
b1*b1) / 2.0;
    double d = a1*b2 -a2*b1;
    return a + Pt(c1*b2 - c2*b1, a1*c2 - a2*c1) / d;
}
```

三角形的垂心

三角形三边上的三条高线交于一点，称为三角形垂心。

性质

锐角三角形的垂心在三角形内；直角三角形的垂心在直角的顶点；钝角三角形的垂心在三角形外。

求法

垂心可以根据外心、重心与垂心的关系（欧拉定理）得出。

模板代码

```
Pt Orthocenter(Pt a, Pt b, Pt c) {
    return triangleMassCenter(a, b, c) * 3.0 -
circumCenter(a, b, c) * 2.0;
}
```

三角形的内心

三角形内心为三角形三条内角平分线的交点。

性质

与三角形各边都相切的圆叫做三角形的内切圆，内切圆的圆心即是三角形内心，内心到三角形三边距离相等。这个三角形叫做圆的外切三角形。

多边形

简单多边形

简单多边形是边不相交的多边形，大部分我们在编程竞赛中的计算几何题目中的多边形都是简单多边形，所以在这个手册中所提到的多边形都是简单多边形。

判断点在多边形内

判断方法

判断点在多边形内：从该点做一条水平向右的射线，统计射线与多边形相交的情况，若相交次数为偶数，则说明该点在形外，否则在形内。为了便于交点在定点或射线与某些边重合时的判断，可以将每条边看成左开右闭的线段，即若交点为左端点就不计算。

示例代码

```
#define nxt(x) ((x+1)%n)

int PtInPolygon(Pt p, Polygon &a) {
    int num = 0, d1, d2, k, n = size(a);
    for (int i = 0; i < n; ++i) {
        if (PtOnSegment(p, a[i], a[nxt(i)])) {
            return 2;
        }
        k = sgn(det(a[nxt(i)]-a[i], p-a[i]));
        d1 = sgn(a[i].y - p.y);
        d2 = sgn(a[nxt(i)].y - p.y);
        if (k > 0 && d1 <= 0 && d2 > 0) num++;
        if (k < 0 && d2 <= 0 && d1 > 0) num--;
    }
    return num != 0;
}
```

经典题目

- POJ 2398 Toy Storage

多边形的面积

求法

多边形的面积可以靠三角剖分求得。对多边形的每一条边和原点$$O$$所组成的三角形通过叉积求有向面积并简单求和，就可以求得多边形的有向面积。而且通过求得的有向面积能判断出多边形中点的方向。如果输入多边形点的方向是按照逆时针给出的话，求得的面积就是正数，如果输入的多边形的点是按照顺时针给出的话，求得的面积就是负数。

示例代码

```
#define nxt(x) ((x+1)%n)

double polygon_area(const Polygon &p) {
    double ans = 0.0;
    int n = p.size();
    for (int i = 0; i < n; ++i)
        ans += det(p[i], p[nxt(i)]);
    return ans / 2.0;
}
```

多边形的重心

算法

将多边形分割为三角形的并，并对每个三角形求重心，然后以三角形的有向面积为权值将所有面积加权求和即可。

示例代码

```
#define nxt(x) ((x+1)%n)

Pt polygon_mass_center(const Polygon &p) {
    Pt ans = Pt(0, 0);
    double area = polygon_area(p);
    if (sgn(area) == 0) return ans;
    int n = p.size();
    for (int i = 0; i < n; ++i)
        ans = ans + (p[i]+p[nxt(i)]) * det(p[i],
p[nxt(i)]);
    return ans / area / 6.0;
}
```

多边形内的格点数

Pick公式

给定顶点坐标均是整点的简单多边形，有：

```
面积 = 内部格点数 + 边上格点数 / 2 - 1
```

边界的格点数

把每条边当作左开右闭的区间避免重复，一条左开右闭的线段$$AB$$上的格点数为$$gcd(B_x-A_x,B_y-A_y)$$。

```
int polygon_border_point_cnt(const Polygon &p) {
    int ans = 0;
    int n = p.size();
    for (int i = 0; i < n; ++i)
        ans += gcd(Abs(int(p[next(i)].x-p[i].x)),
Abs(int(p[next(i)].y-p[i].y)));
    return ans;
}

int polygon_inside_point_cnt(const Polygon &p) {
    return int(polygon_area(p)) + 1 -
polygon_border_point_cnt(p) / 2;
}
```

凸包

点的有序化

凸包算法多要先对点进行排序。点排序的主要方法有两种——极角排序和水平排序。

极角排序

极角排序一般选择一个点做极点，然后以这个点为中心建立极坐标，将输入的点按照极角从小到大排序，如果两个点的极角相同，那么将距离极点较远的点排在前面。

但在实践过程中，一般不进行真正的极角排序，而是通过进行叉积比较，并不真的计算出点的极角。这种做法可以避免使用对精度影响较大的三角函数运算，对精度影响较小。

水平排序

水平排序将所有点按照$$y$$坐标从小到大排列，$$y$$坐标相同的则按照$$x$$坐标从小到大排序。选取排序后最前面的$$A$$点和最后面的$$B$$点，将$$\vec{AB}$$右边的点按照次序取出，再将左侧的点按照次序逆序取出后连起来就是最终的结果。

凸包求法

Graham 扫描法

Graham算法是求解静态凸包较好的一种算法，时间复杂度为$$n\log n$$，尤其在求解大量点构成的凸包时，能消耗较少的时间。

算法简述

Graham算法首先要求将无序的点有序化（参见前面的点的有序化）。Graham算法会维持一个栈，栈中的点为输出凸包上的点。Graham算法首先选择一个肯定在凸包上的点入栈，并按照点的顺序将点依次入栈。如果栈中元素数大于1，那么每次入栈前，检查栈顶两个点和新入栈的点是否能保持凸性（设栈顶元素为$$P*0P_1$$，新入栈点为*$$Q$$*，那么它们要满足*$$\vec{P*{top-1}P*{top}} \times \vec{P*{top}Q} \geq 0$$*才能维持凸性）。如果不能满足，则不断出栈知道能满足凸性或元素数不大于1位置，并将新点入栈。重复上述过程，就能得到凸包。

Graham算法本身的时间复杂度为$$O(n)$$，所以算法时间复杂度取决与点的点的有序化的时间复杂度，通常为$$O(n\log n)$$，如果输入的点已经为有序化的点，那么无序排序，直接进行Graham算法，总时间复杂度为$$O(n)$$。

实现细节的注意事项

极角大小问题

实际实现Graham算法的极角排序并不是真正的按照极角大小排序，因为计算机在表示和计算浮点数时会有一定的误差。一般会利用叉积判断两个点的相对位置来实现极角排序的功能。假设以确定平面中最下最左的点（基准点）$$P$$，并已知平面上其它两个不同的点$$AB$$。若点$$A$$在向量$$PB$$的逆时针方向，那么我们认为$$A$$的极角大于$$B$$的极角，反之$$A$$的极角小于$$B$$的极角（具体实现应借助叉积）。

极角相同点的处理

在Graham算法中，经常会出现两个点极角相同的情况。对于具有相同极角的两个不同点$$AB$$，那么我们应该把$$AB$$两点的按照距离基准点距离的降序排列。而对于完全重合的两点，可以暂不做处理。

模板代码

```
typedef vector<Pt> Convex;

// 排序比较函数，水平序
bool comp_less(Pt a, Pt b) {
    return sgn(a.x-b.x) < 0 || (sgn(a.x-b.x) == 0 &&
sgn(a.y-b.y) < 0);
}

// 返回a中点计算出的凸包，结果存在res中
void convex_hull(Convex &res, vector<Pt> a) {
    res.resize(2 * a.size() + 5);
    sort(a.begin(), a.end(), comp_less);
    a.erase(unique(a.begin(), a.end()), a.end());
    int m = 0;
    for (int i = 0; i < int(a.size()); ++i) {
        while (m>1 && sgn(det(res[m-1] - res[m-2], a[i] -
res[m-2])) <= 0)
            --m;
        res[m++] = a[i];
    }
    int k = m;
    for (int i = int(a.size()) - 2; i >= 0; --i) {
        while (m>k && sgn(det(res[m-1] - res[m-2], a[i] -
res[m-2])) <= 0)
            --m;
        res[m++] = a[i];
    }
    res.resize(m);
    if (a.size() > 1) res.resize(m-1);
}
```

经典题目

- POJ 1113 Wall
- POJ 3348 Cows

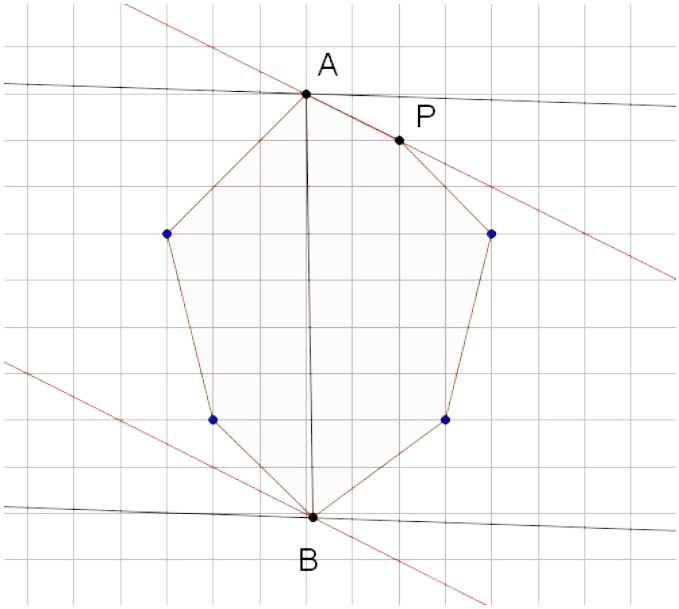
旋转卡壳

简介

旋转卡壳一般用于求凸包上最远的点的距离，这个距离也称为凸包的直径。旋转卡壳可以在$$O(n)$$的时间复杂度内找出最远的点对。

算法简述

如下图，假设$$A$$与$$B$$是凸包上最远的点对，那么过$$A$$和$$B$$分别做两条平行的直线并进行旋转，使过$$A$$的直线与$$A$$的相邻点$$P$$重合（图中红线）。那么点$$B$$一定凸包上距离直线$$AP$$最远的点，这样的话，$$\triangle_{ABP}$$一定以$$AP$$为底边以另一个凸包上的顶点形成的三角形中面积最大的。如果我们枚举凸包上所有的相邻点组成的边并找出距离这条边最远的点，最远点对一定在这个三角形中。因为在我们按照逆时针枚举所有边的同时，距离它们最远的点的变化方向也是逆时针的，所以整个算法的时间复杂度是$$O(n)$$的。



模板代码

```
// 计算凸包a的直径
double convex_diameter(const Convex &a, int &first, int
&second) {
    int n = a.size();
    double ans = 0.0;
    first = second = 0;
    if (n == 1) return ans;
    for (int i = 0, j = 1; i < n; ++i) {
        while (sgn(det(a[nxt(i)]-a[i], a[j]-a[i]) -
det(a[nxt(i)]-a[i], a[nxt(j)]-a[i])) < 0)
            j = nxt(j);
        double d = max((a[i]-a[j]).norm(), (a[nxt(i)]-
a[nxt(j)]).norm());
        if (d > ans) ans=d, first=i, second=j;
    }
    return ans;
}
```

经典题目

- POJ 2187 Beauty Contest
- POJ 2079 Triangle
- POJ 2007 Scrambled Polygon

半平面

简介

一条在平面上的直线能将平面分成两个部分，半平面就是其中的一半。

表示方法

- 半平面可以表示成$$Ax + By + C \geq 0$$这样二维坐标系下的方程形式。
- 半平面可以由一条有向线段的左边（或右边）来表示。

半平面的交

半平面的交集就是由许多半平面组成的交集区域，通常是一个凸多边形。

应用

半平面的交有很多应用，如平面上的线性规划、求多边形的核等。

半平面交求法

常见的半平面求交的方法有增量法、分治法和排序增量法等。

增量法

分治法

排序增量法

我们用一个向量$$\vec{P_1P_2}$$的左侧来描述一个半平面。首先将半平面按照极角排序，极角相同的则只保留最左侧的一个。然后用一个双端队列维护这些半平面：按照顺序插入，在插入半平面$$p_i$$之前判断双端队列尾部的两个半平面的交点是否在半平面$$p_i$$内，如果不是则删除最后一个半平面；判断双端队列尾

部的两个半平面交是否在半平面$$p_i$$内，如果不是则删除第一个半平面。插入完毕之后再处理一下双端队列两端多余的半平面，最后求出尾端和顶端的两个半平面的交点即可。

模板代码

```
// 计算半平面交
void halfplane_intersect(vector<HP> &v, Convex &output) {
    sort(v.begin(), v.end(), cmp_HP);
    deque<HP> q;
    deque<Pt> ans;
    q.push_back(v[0]);
    int n = v.size();
    for (int i = 1; i < n; ++i) {
        if (sgn(arg(v[i].t-v[i].s) - arg(v[i-1].t-v[i-
1].s)) == 0)
            continue;
        while (ans.size() > 0 && !satisfy(ans.back(),
v[i])) {
            ans.pop_back();
            q.pop_back();
        }
        while (ans.size() > 0 && !satisfy(ans.front(),
v[i])) {
            ans.pop_front();
            q.pop_front();
        }
        ans.push_back(crosspoint(q.back(), v[i]));
        q.push_back(v[i]);
    }
    while (ans.size() > 0 && !satisfy(ans.back(),
q.front())) {
        ans.pop_back();
        q.pop_back();
    }
```

```
    while (ans.size() > 0 && !satisfy(ans.front(),
q.back())) {
        ans.pop_front();
        q.pop_front();
    }
    ans.push_back(crosspoint(q.back(), q.front()));
    output = vector<Pt>(ans.begin(), ans.end());
}
```

经典题目

- POJ 2540 Hotter Colder

凸多边形交

算法简述

凸多边形的交可以直接通过使用半平面交的算法求得，将所有的凸多边形拆成多个半平面，并求这些半平面的交，就能求得凸多边形的交。

模板代码

```
// 凸多边形交
void convex_intersection(const Convex &v1, const Convex
&v2, Convex &out) {
    vector<HP> h;
    for (int i = 0, n = v1.size(); i < n; ++i)
        h.push_back(HP(v1[i], v1[nxt(i)]));
    for (int i = 0, n = v2.size(); i < n; ++i)
        h.push_back(HP(v2[i], v2[nxt(i)]));
    halfplane_intersect(h, out);
}
```

多边形的核

平面简单多边形的核是该多边形内部的一个点集，该点集中任意一点与多边形边界上一点的连线都处于这个多边形内部。

算法简述

多边形的核可以直接通过求多边形的边所在的直线表示的半平面的交求得。

经典题目

- POJ 1279 Art Gallery
- POJ 2451 Uyuws Concert

圆

圆与线求交

算法思路

将线段$$AB$$写成参数方程$$P=A+t(B-A)$$，带入圆的方程，得到一个一元二次方程。解出$$t$$就可以求得线段所在直线与圆的交点。如果$$0 \leq t \leq 1$$则说明点在线段上。

模板代码

```
void circle_cross_line(Pt a, Pt b, Pt o, double r, Pt
ret[], int &num) {
    double ox = o.x, oy = o.y, ax = a.x, ay = a.y, bx =
b.x, by = b.y;
    double dx = bx-ax, dy = by-ay;
    double A = dx*dx + dy*dy;
    double B = 2*dx*(ax-ox) + 2*dy*(ay-oy);
    double C = sqr(ax-ox) + sqr(ay-oy) - sqr(r);
    double delta = B*B - 4*A*C;
    num = 0;
    if (sgn(delta) >= 0) {
        double t1 = (-B - Sqrt(delta)) / (2*A);
        double t2 = (-B + Sqrt(delta)) / (2*A);
        if (sgn(t1-1) <= 0 && sgn(t1) >= 0)
            ret[num++] = Pt(ax + t1*dx, ay + t1*dy);
        if (sgn(t2-1) <= 0 && sgn(t2) >= 0)
            ret[num++] = Pt(ax + t2*dx, ay + t2*dy);
    }
}
```

经典题目

- POJ 1263 Reflections

圆与圆求交

```
// 计算圆a和圆b的交点，注意要先判断两圆相交
void circle_circle_cross(Pt ap, double ar, Pt bp, double
br, Pt p[]) {
    double d = (ap - bp).norm();
    double cost = (ar*ar + d*d - br*br) / (2*ar*d);
    double sint = sqrt(1.0 - cost*cost);
    Pt v = (bp - ap) / (bp - ap).norm() * ar;
    p[0] = ap + rotate(v, cost, -sint);
    p[1] = ap + rotate(v, cost, sint);
}
```

经典题目

- POJ 2546 Circular Area

圆与多边形交

算法思路

按照圆心为中心，将多边形三角剖分，并计算出每个三角形与圆交的面积后求和。求三角形与圆交面积的方法要按照情况讨论。

1. $$AB$$都在圆内，计算三角形$$OAB$$的面积。
2. $$A$$在圆内$$B$$不在圆内，这时计算一个三角形的面积和一个扇形的面积。
3. $$B$$在圆内$$A$$不在圆内，这时计算一个三角形的面积和一个扇形的面积。
4. $$AB$$都不在圆内，如果$$AB$$与圆无交点，则计算一个扇形的面积，否则

计算两个扇形和一个三角形的面积。

模板代码

```
// 圆与直线相交
void circle_cross_line(Pt a, Pt b, Pt o, double r, Pt
ret[], int &num) {
    double ox = o.x, oy = o.y, ax = a.x, ay = a.y, bx =
b.x, by = b.y;
    double dx = bx-ax, dy = by-ay;
    double A = dx*dx + dy*dy;
    double B = 2*dx*(ax-ox) + 2*dy*(ay-oy);
    double C = sqr(ax-ox) + sqr(ay-oy) - sqr(r);
    double delta = B*B - 4*A*C;
    num = 0;
    if (sgn(delta) >= 0) {
        double t1 = (-B - Sqrt(delta)) / (2*A);
        double t2 = (-B + Sqrt(delta)) / (2*A);
        if (sgn(t1-1) <= 0 && sgn(t1) >= 0)
            ret[num++] = Pt(ax + t1*dx, ay + t1*dy);
        if (sgn(t2-1) <= 0 && sgn(t2) >= 0)
            ret[num++] = Pt(ax + t2*dx, ay + t2*dy);
    }
}

// 计算扇形面积
double sector_area(Pt a, Pt b, double r) {
    double theta = atan2(a.y, a.x) - atan2(b.y, b.x);
    while (theta <= 0) theta += 2*PI;
    while (theta > 2*PI) theta -= 2*PI;
    theta = min(theta, 2*PI - theta);
    return r*r*theta / 2;
}

double area(Pt res[], int n, double r) {
```

```
    double ans = 0.0;
    for (int i = 0; i < n; ++i) {
        Pt a = res[i], b = res[next(i)];
        Pt p[2];
        int num = 0;
        int ina = sgn(a.norm() - r) < 0;
        int inb = sgn(b.norm() - r) < 0;
        int s = sgn(det(a, b));

        double part = 0.0;

        if (ina) {
            if (inb) {
                part = Abs(det(a, b)) / 2.0;
            }else{
                circle_cross_line(a, b, Pt(0, 0), r, p,
num);
                part = sector_area(b, p[0], r) +
fabs(det(a, p[0])) / 2.0;
            }
        }else{
            if (inb) {
                circle_cross_line(a, b, Pt(0, 0), r, p,
num);
                part = sector_area(p[0], a, r) +
fabs(det(p[0], b)) / 2.0;
            }else{
                circle_cross_line(a, b, Pt(0, 0), r, p,
num);
                if (num == 2) {
                    part = sector_area(a, p[0], r) +
sector_area(p[1], b, r)
                        + fabs(det(p[0], p[1])) / 2.0;
                }else{
                    part = sector_area(a, b, r);
                }
```

```
            }
        }
        ans += s * part;
    }
    return ans;
}

const int MaxN = 55;
Pt res[MaxN];
double r;
int n;

int main() {
    while (cin >> r >> n) {
        for (int i = 0; i < n; ++i)
            cin >> res[i].x >> res[i].y;
        double ans = Abs(area(res, n, r));
        cout << setiosflags(ios::fixed) <<
setprecision(2) << ans << endl;
    }

    return 0;
}
```

三维计算几何

三维凸包

算法简述

三维凸包一个较为容易实现的算法是增量法。先将所有的点打乱顺序，然后选择四个不共面的点组成一个四面体，如果找不到说明凸包不存在。然后遍历剩余的点，不断更新凸包。对遍历到的点做如下处理。

1. 如果点在凸包内，则不更新。
2. 如果点在凸包外，那么找到所有原凸包上所有分隔了对于这个点可见面和不可见面的边，以这样的边的两个点和新的点创建新的面加入凸包中。

模板代码

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <vector>
#include <iomanip>
#include <iostream>
#include <cmath>
using namespace std;

/* Macros */
/*********************************************************
**********************/
#define nxt(i) ((i+1)%n)
#define nxt2(i, x) ((i+1)%((x).size()))
#define prv(i) ((i+(x).size()-1)%n)
#define prv2(i, x) ((i+(x).size()-1)%((x).size()))
#define sz(x) (int((x).size()))
```

```
#define setpre(x) do{cout<<setprecision(x)
<<setiosflags(ios::fixed);}while(0)

/* Real number tools */
/*********************************************************
**********************/
const double PI = acos(-1.0);
const double eps = 1e-8;
double mysqrt(double x) {
    return x <= 0.0 ? 0.0 : sqrt(x);
}
double sq(double x) {
    return x*x;
}
int sgn(double x) {
    return x < -eps ? -1 : x > eps ? 1 : 0;
}

/* 3d Point */
/*********************************************************
**********************/
struct Pt3 {
    double x, y, z;
    Pt3() { }
    Pt3(double x, double y, double z) : x(x), y(y), z(z)
{ }
};
typedef const Pt3 cPt3;
typedef cPt3 & cPt3r;

Pt3 operator + (cPt3r a, cPt3r b) { return Pt3(a.x+b.x,
a.y+b.y, a.z+b.z); }
Pt3 operator - (cPt3r a, cPt3r b) { return Pt3(a.x-b.x,
a.y-b.y, a.z-b.z); }
Pt3 operator * (cPt3r a, double A) { return Pt3(a.x*A,
a.y*A, a.z*A); }
```

```
Pt3 operator * (double A, cPt3r a) { return Pt3(a.x*A,
a.y*A, a.z*A); }
Pt3 operator / (cPt3r a, double A) { return Pt3(a.x/A,
a.y/A, a.z/A); }
bool operator == (cPt3r a, cPt3r b) {
    return !sgn(a.x-b.x) && !sgn(a.y-b.y) && !sgn(a.z-
b.z);
}
istream& operator >> (istream& sm, Pt3 &pt) {
    sm >> pt.x >> pt.y >> pt.z; return sm;
}
ostream & operator << (ostream& sm, cPt3r pt) {
    sm << "(" << pt.x << ", " << pt.y << ", " << pt.z <<
")"; return sm;
}
double len(cPt3r p) { return mysqrt(sq(p.x) + sq(p.y) +
sq(p.z)); }
double dist(cPt3r a, cPt3r b) { return len(a-b); }
Pt3 unit(cPt3r p) { return p / len(p); }
Pt3 det(cPt3r a, cPt3r b) {
    return Pt3(a.y*b.z-a.z*b.y, a.z*b.x-a.x*b.z, a.x*b.y-
a.y*b.x);
}
double dot(cPt3r a, cPt3r b) {
    return a.x*b.x + a.y*b.y + a.z*b.z;
}
double mix(cPt3r a, cPt3r b, cPt3r c) {
    return dot(a, det(b, c));
}
/* 3d Line & Segment */
/*************************************************
**********************/
struct Ln3 {
    Pt3 a, b;
    Ln3() { }
    Ln3(cPt3r a, cPt3r b) : a(a), b(b) { }
```

```
};
typedef const Ln3 cLn3;
typedef cLn3 & cLn3r;

bool ptonln(cPt3r a, cPt3r b, cPt3r c) {
    return sgn(len(det(a-b, b-c))) <= 0;
}

/* 3d Plane */
/*********************************************************
**********************/
struct Pl {
    Pt3 a, b, c;
    Pl() { }
    Pl(cPt3r a, cPt3r b, cPt3r c) : a(a), b(b), c(c) { }
};
typedef const Pl cPl;
typedef cPl & cPlr;

Pt3 nvec(cPlr pl) {
    return det(pl.a-pl.b, pl.b-pl.c);
}

/* Solution  */
/*********************************************************
**********************/
bool cmp(cPt3r a, cPt3r b) {
    if (sgn(a.x-b.x)) return sgn(a.x-b.x) < 0;
    if (sgn(a.y-b.y)) return sgn(a.y-b.y) < 0;
    if (sgn(a.z-b.z)) return sgn(a.z-b.z) < 0;
    return false;
}

struct Face {
    int a, b, c;
    Face() { }
```

```
    Face(int a, int b, int c) : a(a), b(b), c(c) { }
};

void convex3d(vector<Pt3> &p, vector<Pl> &out) {
    sort(p.begin(), p.end(), cmp);
    p.erase(unique(p.begin(), p.end()), p.end());
    random_shuffle(p.begin(), p.end());
    vector<Face> face;
    for (int i = 2; i < sz(p); ++i) {
        if (ptonln(p[0], p[1], p[i])) continue;
        swap(p[i], p[2]);
        for (int j = i + 1; j < sz(p); ++j)
            if (sgn(mix(p[1]-p[0], p[2]-p[1], p[j]-p[0]))
!= 0) {
                swap(p[j], p[3]);
                face.push_back(Face(0, 1, 2));
                face.push_back(Face(0, 2, 1));
                goto found;
            }
    }
found:
    vector<vector<int> > mark(sz(p), vector<int>(sz(p),
0));
    for (int v = 3; v < sz(p); ++v) {
        vector<Face> tmp;
        for (int i = 0; i < sz(face); ++i) {
            int a = face[i].a, b = face[i].b, c =
face[i].c;
            if (sgn(mix(p[a]-p[v], p[b]-p[v], p[c]-p[v]))
< 0) {
                mark[a][b] = mark[b][a] = v;
                mark[b][c] = mark[c][b] = v;
                mark[c][a] = mark[a][c] = v;
            }else{
                tmp.push_back(face[i]);
            }
```

```
        }
        face = tmp;
        for (int i = 0; i < sz(tmp); ++i) {
            int a = face[i].a, b = face[i].b, c =
face[i].c;
            if (mark[a][b] == v) face.push_back(Face(b,
a, v));
            if (mark[b][c] == v) face.push_back(Face(c,
b, v));
            if (mark[c][a] == v) face.push_back(Face(a,
c, v));
        }
    }
    out.clear();
    for (int i = 0; i < sz(face); ++i)
        out.push_back(Pl(p[face[i].a], p[face[i].b],
p[face[i].c]));
}

vector<Pt3> p;
vector<Pl> out;

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        Pt3 pt;
        cin >> pt;
        p.push_back(pt);
    }
    convex3d(p, out);
    double area = 0.0;
    for (int i = 0; i < sz(out); ++i)
        area += len(det(out[i].a-out[i].b, out[i].b-
out[i].c));
    setpre(3);
```

```
    cout << area / 2.0 << "\n";
    return 0;
}
```

经典题目

- POJ 3528 Ultimate Weapon

数学

比赛中的数学知识是渗透在方方面面的。

- 概率
- 高斯消元法

---

author：王昊天

概率

基础知识

样本空间、事件和概率

样本空间 `S` 是一个集合，它的元素被称为基本事件。样本空间的一个子集被称为事件，根据定义所有的基本事件都互斥。

概率公理

如果有一种事件到实数的映射 $$P{}$$，满足：

1. 对任何事件 $$A$$，$$P{A}\geq0$$；
2. $$P{S}=1$$；
3. 对两个互斥事件，$$P{A\cup B}=P{A}+P{B}$$，

则可称 $$P{A}$$ 为事件 $$A$$ 的概率。

随机变量

如果对样本空间 $$S$$ 中的任意事件 $$e$$，都有唯一的实数 $$X(e)$$ 与之对应，则称 $$X=X(e)$$ 为样本空间 $$S$$ 上的随机变量，其中离散型随机变量与连续型随机变量较常见。

离散型随机变量及其概率分布

取值范围为有限或无限可数个实数的随机变量称为离散型随机变量。设离散型随机变量 $$X$$ 取值 $$x_i$$ 时的概率为 $$p_k (k=1,2,\ldots)$$，则称 $$X$$ 的所有取值以及对应概率为 $$X$$ 的概率分布，记做 $$P{X=x_k} = p_k (k=1,2,\ldots)$$。

常见的离散型随机变量的概率分布有两点分布，二项分布，几何分布，超几何分布，泊松分布。

连续型随机变量及其概率分布

如果 $$X$$ 是在实数域或区间上取连续值的随机变量，设 $$X$$ 的概率分布函数为 $$F(x)=P{X\leq x}$$，若存在非负可积函数 $$f(x)$$，使对任意的 $$x$$，有 $$F(x)=\int^x_{-\infty}f(t)dt$$，则称 $$X$$ 为连续型随机变量，称 $$f(x)$$ 为 $$X$$ 的概率密度函数。要注意，概率密度不是概率。常见的连续型随机变量分布有均匀分布，正态分布，指数分布。

连续型随机向量及其概率分布

如果 $$X_1,X_2,\ldots,X_N$$ 都是连续型随机变量，则称 $$(X_1,X_2,\ldots,X_N)$$ 为 **$$N$$** 维随机向量，其概率分布函数为 $$F(x_1,x_2,…,x_N)=P{X_1\leq x_1,X_2\leq x_2,\ldots,X_N\leq x_N}$$。若存在非负可积函数 $$f(x_1,x_2,\ldots,x_N)$$ 使得 $$F(x_1, x_2, \ldots, x_N)=\int_D f(x_1, x_2, \ldots, x_N)dx_1dx_2\ldots dx_N$$，其中等式右端表示 $$N$$ 重积分，就称 $$f(x_1,x_2,…,x_N)$$是 $$N$$ 为随机向量 $$(X_1,X_2,\ldots,X_N)$$ 的联合概率密度函数。

如果 $$X_1, X_2, \ldots, X_N$$ 相互独立，并且分别有概率密度函数 $$f_1(x_1), f_2(x_2), \ldots, f_N(x_N)$$，那么 $$f(x_1, x_2, \ldots, x_N) = f_1(x_1)f_2(x_2)\ldots f_N(x_N)$$。

数学期望

离散型随机变量的数学期望

设离散型随机变量 $$X$$ 的分布律为 $$P{X=x*k} = p_k(k=1,2,*dots)$$，若 *$$\sum^{\infty}*{k=1}|x*kp_k|$$* 存在，则称 *$$\sum^\infty*{k=1}x_kp_k$$* 为 $$X$$ 的数学期望，简称期望，记为 $$E(X)$$。

连续型随机变量的数学期望

设连续型随机变量 $$X$$ 的概率密度函数为 $$f(x)$$，若广义积分 $$\int^{+\infty} *{-\infty}|xf(x)|dx$$* 收敛，则称 *$$\int^{+\infty}*{-\infty}xf(x)dx$$* 为连续型随机变量 $$X$$ 的数学期望，记为 $$E(X)$$。

例题1 LastMarble (TopCoder SRM 349 div one 1000)

题目描述

有 `red` 个红球，`blue` 个蓝球在一个袋子中。两个玩家轮流从袋子中取球，每个人每次可以取 1，2 或 3 个球，但在他把球拿出袋子之前，他并不知道所取球的颜色。每次球被取出袋子后，它们的颜色被公布给所有人。取走最后一个红球的人输。现在已知有人在游戏开始前取走了 `removed` 个球，并且谁也不知道球的颜色。在两个玩家都采取最优策略时，先手的胜率是多少？

数据范围：$$1\leq red,blue\leq 100,0\leq removed\leq red-1$$。

分析

当 $$removed=0$$ 的时候，这个问题是很普通的动态规划问题。我们只需设 $$F(r,b)$$ 代表现在剩 $$r$$ 个红球，$$b$$ 个蓝球，面对当前局面的玩家所能得到的最大胜率。那么：

$$ F(0, b)=1(b\geq 0)\ F(r, b)=\max*{1\leq m\leq \min(r+b, 3)}(\sum*{0\leq p\leq r, 0\leq q\leq b, p+q=m}\frac{C^pr*C_b^q}{C*{r+b}^m}(1-F(r-p,b-q)))

$$ 其中 $$\frac{C^p*rC_b^q}{C*{r+b}^m}$$ 是取到 $$p$$ 个红球，$$q$$ 个蓝球的概率。$$F(red,blue)$$ 就是我们要的答案。

对于 $$removed>0$$ 的情况，我们显然可以知道：

定理 在 $$red$$ 个红球，$$blue$$ 个蓝球中先取 $$a$$ 个球，再取 $$b$$ 个球，剩余不同颜色的球数的概率分布与先取 $$b$$ 个球，再取 $$a$$ 个球所对应的剩余不同颜色的球数的概率分布是相同的。

上面的定理告诉了我们，取球的顺序和最终的结果没有关系。

我们设 $$F(r,b)$$ 表示当前有 $$r$$ 个红球，$$b$$ 个蓝球的，被事先取走了 $$removed$$ 个球（不知道它们的颜色），面对这个局面的玩家所能得到的最大胜率。当玩家取走 $$m$$ 个球时，根据上面的定理，新的局面与玩家先取走 $$m$$ 个球，再让 $$removed$$ 个球被取走所得到的局面完全一样！但是有一种边界情况要考虑:由于不能保证 $$removed\leq r$$，可能在一些情况下，取走 $$m$$ 个球后，玩家已经输了，而不能进行下面的游戏。要解决这种特殊情况，只需对 $$F$$ 的定义和动态规划方程略加修改：让 $$F(r,b)$$ 表示当前有 $$r$$ 个红球，$$b$$ 个蓝球，被取走了 $$removed$$ 个球但仍然至少还有 1 个红球的情况下，当前玩家的最大胜率。

我们用 $$\text{Pro}(r,b,k)$$ 表示有 $$r$$ 个红球，$$b$$ 个蓝球，取走 $$k$$ 个球而红球数仍大于 0 的概率，那么 $$\text{Pro}(r,b,k)=\sum*{a<r,0\leq k-a\leq b}\frac{C_r^aC_b^{k-a}}{C*{r+b}^k}$$。我们可以用递推式求解：

$$ \text{Pro}(0,b,0)=0;\ \text{Pro}(r,b,0)=1;(r>0)\ \text{Pro}(r,b,k)=\frac{r}{r+b}\text{Pro}(r-1,b,k-1)+\frac{b}{r+b}\text{Pro}(r,b-1,k-1)

$$ 再考虑最初的dp式，因为 $$r$$ 个红球，$$b$$ 个蓝球取走 $$removed$$ 个球仍有至少 $$1$$ 个红球的情况，包含了有 $$(r-p)$$ 个红球，$$(b-q)$$ 个蓝球，取走 $$removed$$ 个球后仍有至少 1 个红球的情况，因此在前者满足的基础上后者满足的概率是 $$\frac{\text{Pro}(r-p,b-q,removed)}{\text{Pro}(r,b,removed)}$$。由此我们得出最终的方程：

$$ \begin{array}{ll} F(r,b)=0 & (r+b=removed)\ F(r,b)=\max*{1\leq m\leq min(r+b,3)}(\sum*{p+q=m}\frac{\text{Pro}(r-p,b-q,removed)}{\text{Pro}(r,b,removed)}\frac{C^p*rC_b^q}{C*{r+b}^m}(1-F(r-p,b-q))) & (r+b>removed) \end{array}

$$ 最后的答案是$$F(red, blue)$$。

例题2 Randomness (UVA 11429)

题目描述

有一个随机数生成器能随机返回 $$1$$ 到 $$R$$ 的正整数，现在有 $$N$$ 个事件，要求第 $$i$$ 个事件的 发生概率是 $$\frac{a_i}{b_i}$$，用该随机数生成器设计一种事件触发装置，使随机数生成器的期望使用次数 $$\text{Exp}$$ 尽量少,求出 $$\text{Exp}$$（精确到 $$10^{-7}$$）。

数据范围：$$2\leq R\leq 1000,1\leq N\leq 1000, 1\leq a*i\leq b_i \leq 1000(1\leq i\leq N)$$*，且 *$$\sum^N*{i=1}\frac{a_i}{b_i}=1$$，所有的数都是正整数。

分析

为了观察题目规律，我们先看一个例子：
当 $$R=100,N=3,\frac{a*1}{b_1}=\frac{a_2}{b_2}=\frac{a_3}{b_3}=\frac{1}{3}$$ 时，最优策略可以是：当生成的数 *$$x\leq 99$$* 时执行事件 *$$x \text{mod} 3$$*，否则再使用一次随机数生成器，事件的选择与上一次一样。这样每个事件发生的概率是：*$$\sum*{i=0}^{\infty}\frac{33}{100^i}=\frac{\frac{33}{100}-0}{1-\frac{1}{100}}=\frac{1}{3}$$；$$\text{exp}=\sum_{i=0}^{\infty}\frac{1}{100^i}=\frac{1-0}{1-\frac{1}{100}}=\frac{100}{99}。$$

我们看到第一次使用生成器(定义为第一层生成器)时，有一些直接映射到事件，其余的每一个返回值分别对应一次新的生成器的使用，定义为第二层生成器。归纳地，定义第 $$i+1(2\leq i)$$ 层生成器为由第 $$i$$ 层生成器返回值引起的新的生成器的使用。
很明显，对事件 $$i$$ 而言，假设第 $$k$$ 层生成器中有 $$D(i,k)$$ 个返回值对应着事件 $$i$$ 的发生，一定有$$ \frac{a*i}{b_i}=\sum^{\infty}*{k=1}\frac{D(i,k)}{R^k}$$

这里 $$D(i,k)<R(k=1,2,\ldots)$$，因为如果某个 $$D(i,k)\geq R$$，我们把 $$D(i,k)$$ 减去 $$R$$，$$D(i,k-1)$$ 加 1，则等式仍成立，而 Exp 减小了。同时可以看出，这就是 $$\frac{a_i}{b_i}$$ 的 $$\frac{1}{R}$$ 进制表示，所以 $$D(i,k)$$ 的取值是唯一的。

为了计算 Exp，我们定义 $$H(i+1)$$ 为第 $$i$$ 层生成器产生的导致新的生成器使用的返回值个数，定义 $$H(1)=1$$，那么 $$\frac{H(i)}{R^i}$$ 就是第 $$i$$ 层生成器对 Exp 的贡献，我们有$$ \text{Exp}=\sum^{\infty}_{i=1}\frac{H(i)}{R^i}$$

至于求 $$H(i)$$,因为第 $$i$$ 层生成器中所有导致新生成器使用的返回值的个数，就是第 $$i$$ 层生成器被使用的个数，减去其中所有导致事件的返回值个数，也就是$$ H(i+1)=H(i)R-\sum_{k=1}^{N}D(i,k)$$

我们不可能求出所有的 $$H(i)$$，也没有必要，因为对所有的 $$m\geq 2$$ ，我们有$$ \sum^{\infty}*{k=m}\frac{H(k)}{R^k}<\sum^{\infty}*{k=m}\frac{NR}{R^k}=\frac{N}{R^{m-2}(R-1)}\leq\frac{N}{R^{m-2}}$$

这个式子告诉我们：只要 **$$m$$** 足够大，我们可以让 **Exp** 的误差小到任意程度，而且这个误差减小的速度是很快的。根据题中要求的精度，只要算到第 50 层就足够了。

高斯消元法

数学上，高斯消元法，是线性代数中的一个算法，可用来为线性方程组求解，求出矩阵的秩，以及求出可逆方阵的逆矩阵。当用于一个矩阵时，高斯消元法会产生出一个“行梯阵式”。

高斯消元法的原理是把方程组的系数矩阵通过初等变换成上三角矩阵的形式。

模板见小红书。

组合数学

---

author：高胜杰

容斥原理

算法简述

在集合S中至少具有$$P_1$$,$$P_2$$,...$$P_m$$中的一个元素的个数是：

$$\mid S_1\cup S_2 \cup S_3 \dotsb \cup S_n \mid = \sum \mid S_i \mid - \sum \mid S_i \cap S_j \mid + \dotsb + (-1)^{m+1} \mid S_1 \cap S_2 \dotsb \cap S_m \mid$$

主要运用场合及思路：

简单的讲：容斥原理的最重要的应用就是去重。如果完成一件事情有n类方式$$A_1$$,$$A_2$$...$$A_m$$,每一类进行方式$$A_i$$有$$M_i$$中方法(1 <= i <=n),但是这些方法在合并时存在重叠现象，这时可以选择尝试容斥原理。在比赛中单独使用容斥原理的情况并不多见，常见的问题有错排问题等。

模版

可以用二进制的思想来枚举所有可能的情况，若某位上置1则表示要选取该元素，最后统计1的数目的奇偶性来判断是加上还是减去所求的值。可参考以下结构：

```
for(int i = 0;i < (1 << fn);i ++){
    int cnt = 0;
    int num = 1;
    for(int j = 0;j < fn;j ++){
        if((1 << j) & i){
            num = num * fac[j];
            cnt ++;
        }
    }
    LL d = R / num;
    if(cnt & 1) ans -= 1LL*num * d * (d+1)/2;
    else ans += 1LL*num * d * (d+1)/2;
}
```

例题

hdu1695 GCD

题意：求有多少对(x,y)(1<=x<=b,1<=y<=d)满足$$gcd(x,y)=k$$。

思路：在很多题目当中都可以找到此题的身影。注意到gcd(x,y)=k,说明x,y都能被k整除，那么$$gcd(x/k,y/k)=1$$,于是本题就可以转化为求在$$[1,b/k],[1,d/k]$$两个区间内寻找有多少对数互质。假设b<=d,我们可以在$$[1,d/k]$$中枚举数i，对于每一个i，我们只需找到在$$[1,min(i-1,b/k)]$$中与i互质的个数，最后依次相加就可得到结果。 当i<=b/k时可以用欧拉函数求与i互质的个数，当b/k < i <= d/k时，区间中与i互质的个数 = b/k -（区间中与i不互质的个数）。
区间中与i不互质的数则就是i中素因子的倍数，将它们相加则就是答案，但是由于会有重叠部分，比如6既是2的倍数又是3的倍数，此时就可以用容斥原理来求解。

参考代码:

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
```

```
#define N 100005
#define LL long long
LL elur[N];
int num[N];
int p[N][20];
void init(){
    elur[1] = 1;
    for(int i = 2;i < N;i ++){
        if(!elur[i]){
            for(int j = i;j < N;j += i){
                    if(!elur[j])
                        elur[j] = j;
                    elur[j] = elur[j] * (i-1) / i;
                    p[j][num[j]++] = i;
            }
        }
        elur[i] += elur[i-1];
    }
}
int get(int b,int h){
    int ans = 0;
    for(int i = 1;i < (1 << num[b]);i ++){
        int cnt = 0,nu = 1;
        for(int j = 0;j < num[b];j ++){
            if((1 << j) & i){
                cnt ++;
                nu *= p[b][j];
            }
        }
        if(cnt & 1) ans += h/nu;
        else ans -= h/nu;
    }
    return ans;
}
int main(){
```

```
    int t,a,b,c,d,k;
    init();
    scanf("%d",&t);
    int ca = 1;
    while(t --){
        scanf("%d%d%d%d%d",&a,&b,&c,&d,&k);
        printf("Case %d: ",ca ++);
        if(k == 0){
            puts("0");
            continue;
        }
        if(b > d) swap(b,d);
        b /= k,d /= k;
        LL ans = elur[b];
        for(int i = b + 1;i <= d;i ++){
            ans += b - get(i,b);
        }
        printf("%I64d\n",ans);
    }
    return 0;
}
```

hdu4675 GCD of Sequence

题意：给定一个序列a:$$a_1$$,$$a_2$$,...$$a_n$$,(1<=$$a_i<=M)$$，将该序列修改K位后得到一个序列b:$$b_1$$,$$b_2$$...$$b_n$$,求分别有多少种方案使得b序列的最大公约数分别为[1,m];

思路：设$$f(d)$$表示公约数中含有d的方案数，$$g(n)$$表示最大公约数为n的方案数，则$$f(d) = \sum_{d \mid n} g(n)$$，$$f(d)$$的求法可以这么求：假设有a序列中有sum个数是d的倍数，则还有$$n-sum$$个数不是d倍数，若$$n-sum<=k$$则表示我们可以把$$n-sum$$个不是d的倍数的数改为d的倍数，有$$\lfloor M/d \rfloor ^{n-x}$$种方法，还剩下$$K-(n-sum)$$个数需要修改那些已经是d的倍数的数，所以需要再乘上

$$C*{sum}^{K-(n-sum)} \lfloor m/d-1 \rfloor ^ *{K-(n-sum)}$$，于是*$$f(d) = \lfloor M/d \rfloor ^{n-x} \times C*{sum}^{K-(n-sum)} \lfloor m/d-1 \rfloor ^ {K-(n-sum)}$$,若$$n-sum>k$$：则无法得到一个合法的序列。得到$$f(d)$$之后,由于$$f(d)$$包括

$$g(d),g(d \times 2),g(d \times 3)...g(d \times s) $$,我们只需减去$$g(d \times 2),g(d \times 3)...g(d \times s)$$就是最后的结果。

参考代码如下：

```
#include <iostream>
#include <string.h>
#include <algorithm>
#include <stdio.h>
using namespace std;
#define LL long long
const int N = 300050;
const int mod = 1000000007;
int num[N];
LL ans[N];
LL quick(LL a,LL b){
    LL ans = 1;
    while(b){
        if(b&1) ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}
LL fac[N],inv[N];
LL cal(LL n,LL m){
    return (fac[n] * inv[m] % mod ) * inv[n-m] % mod;
}
int main(){
    int n,m,k;
    fac[0] = 1;
    inv[0] = 1;
    for(int i = 1;i < N;i ++){
```

```
        fac[i] = fac[i-1] * i % mod;
        inv[i] = quick(fac[i],mod-2);
    }
    while(~scanf("%d%d%d",&n,&m,&k)){
        memset(num,0,sizeof(num));
        int nu;
        for(int i = 0;i < n;i ++){
            scanf("%d",&nu);
            num[nu] ++;
        }
        for(int i = m;i >= 1;i --){
            int sum = 0;
            for(int j = i;j <= m;j += i)
                sum += num[j];
            if(n-sum > k){
                ans[i] = 0;
                continue;
            }

            ans[i] = (quick(m/i,n-sum) * quick(m/i-1,k-
n+sum) % mod) * cal(sum,k-n+sum) % mod;

            for(int j = i * 2;j <= m;j += i)
                ans[i] = (ans[i] - ans[j] + mod) % mod;
        }
        printf("%I64d",ans[1]);
        for(int i = 2;i <= m;i ++)
            printf(" %I64d",ans[i]);
        puts("");
    }
    return 0;
}
```

hdu4407 Sum

题意：给定一个$$1,2,3...n$$的一个原始数列，有2种操作。操作1：1 x y p 求区间[x,y]内与p互质的数只和；操作2：2 x c 将第x个数改为c。一共操作m次(1<= m <= 1000)(1<=n<=400000)

思路：注意到m最大只有1000，这是此题的关键，于是对于每一次询问我们可以先求出在原始数列中与p互质的数之和（分解质因数，容斥，与hdu1695类似），最后若区间内数字有修改再处理一下就是答案了。

参考代码：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
#include <map>
using namespace std;

#define LL long long
const int N = 400005;
int pri[N],pn;
int fac[N],fn;
bool is[N];
void init(){
    memset(is,false,sizeof(is));
    pn = 0 ;
    for(int i = 2;i < N;i ++){
        if(!is[i]){
            pri[pn ++] = i;
            for(int j = i+i;j < N;j += i)
                is[j] = true;
        }
    }
}
void get(int n){
    fn = 0;
    for(int i = 0;pri[i] <= n / pri[i] && i < pn;i ++){
        if(n % pri[i] == 0){
```

```
            fac[fn ++] = pri[i];
            while(n % pri[i] == 0)
                n /= pri[i];
        }
    }
    if(n != 1) fac[fn ++] = n;
}
LL solve(int R){
    LL ans = 0;
    for(int i = 1;i < (1 << fn);i ++){
        int cnt = 0;
        int num = 1;
        for(int j = 0;j < fn;j ++){
            if((1 << j) & i){
                num = num * fac[j];
                cnt ++;
            }
        }
        LL d = R / num;
        if(cnt & 1) ans += 1LL*num * d * (d+1)/2;
        else ans -= 1LL*num * d * (d+1)/2;
    }
    return 1LL*R*(R+1)/2 - ans;
}
int gcd(int m,int n){
    return n == 0 ? m : gcd(n,m % n) ;
}
map<int,int>ref;
int main(){
    int t;
    init();
    scanf("%d",&t);
    while(t --){
        int n,m,p;
        scanf("%d%d",&n,&m);
        ref.clear();
```

```
        while(m --){
            int op;
            scanf("%d",&op);
            if(op == 1){
                int l,r;
                scanf("%d%d%d",&l,&r,&p);
                if(l > r) swap(l,r);
                get(p);
                LL ans = solve(r) - solve(l-1);
                map<int,int> :: iterator it =
ref.begin();
                while(it != ref.end()){
                    int pos = (*it).first;
                    if(pos <= r && pos >= l){
                        if(gcd(p,pos) == 1) ans -= pos;
                        if(gcd(p,(*it).second) == 1) ans
+= (*it).second;
                    }
                    it ++;
                }
                printf("%I64d\n",ans);
            }else {
                int x,c;
                scanf("%d%d",&x,&c);
                ref[x] = c;
            }
        }
    }
    return 0;
}
```

uvalive7040 color

题意：长度为N的序列，有M种颜色，用恰好K种颜色进行染色，且相邻元素颜色不同，求方案数。

思路：首先想到要使相邻元素互不相同则一共有$$ K(K-1)^{N-1}$$种方案，这是颜色数不多于K的方案数，于是很容易想到将K置换为K-1，则就是颜色不多于K-1的方案数，两者相减，得到最后的结果。不过这是错误的，原因是你不知道从K种颜色中选择了哪K-1种颜色，而且这样算有交集的情况出现。于是很自然想到用容斥原理解决:$$K(K-1)^{N-1}- \sum_2^{K-1} (-1)^{k-i} C_K^i i (i-1)^{N-1} $$最后还要再乘上$$C_M^K$$表示从M种颜色中选择K种颜色。

参考代码：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
#define LL long long
const LL mod = 1000000007;
#define N 1000000
LL C[N+10],inv[N+10];
void ini(){
    inv[1] = 1;
    for(int i = 2;i <= N;i ++){
        inv[i] = (mod - mod/i) * inv[mod % i] % mod;
    }
}
void getc(int n){
    C[0] = 1;
    for(int i = 1;i <= n;i ++){
        C[i] = (C[i-1]%mod * 1LL*(n-i+1)% mod *inv[i] %
mod) % mod;
        //printf("%d  %lld\n",i,C[i]);
    }
}
LL quick(LL n,int k){
    LL res = 1;
    while(k){
        if(k & 1) res = res * n % mod;
```

```
        n = n * n % mod;
        k >>= 1;
    }
    return res;
}
LL solve(int n,int m,int k){
    LL ans = 1LL * k * quick(1LL*(k-1),n-1) % mod;
    int t = -1;
    getc(k);
    for(int i = k - 1 ;i >= 2;i --){
        ans = ans + (C[i] *i % mod * quick(1LL*(i-1),n-1)
* t % mod + mod) % mod;
        ans %= mod;
        t = -t;
    }
    for(int i = 1;i <= k;i ++)
        C[i] = (C[i-1]%mod * 1LL*(m-i+1)% mod *inv[i] %
mod) % mod;
    return ans * C[k] % mod;
}
int main(){
    int t;
    int n,m,k;
    ini();
    scanf("%d",&t);
    int ca = 1;
    while(t --){
        scanf("%d%d%d",&n,&m,&k);
        printf("Case #%d: ",ca ++);
        printf("%lld\n",solve(n,m,k));
    }
    return 0;
}
```

hdu5072 Coprime

题意：有n个数，从中选择3个元素组成$$(a,b,c)$$的形式，求有多少对a，b，c满足三者之间两两互质或两两都不互质。

思路：在组合数学中有一个模型叫做同色三角形，对应此题可以这么求：此题的对立情况是在这个三元组中恰有两个元素两两互质或两两不互质（用$$f(n)$$表示），用总的情况减去$$f(n)$$就是答案。从n个数当中选择3个数共有$$C_n^3$$种情况， 假设对于每一个元素有$$k_i$$个元素与其互质，则有$$n-k_i-1$$个元素与其不互质，所以$$ f(n)=\sum k_i(n-k_i-1) /2$$。所以此题的答案就是：$$C_n^3 - \sum k_i(n-k_i-1)$$。 对于$$k_i$$与hdu1695类似这里不再叙述。

参考代码：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
#include <vector>
using namespace std;
#define N 100000
bool vis[N+10];
int pri[N+10];
int pnum;
int a[N+10];

void ini()
{
    pnum = 0;
    memset(vis,false,sizeof(vis));
    for(int i = 2; i <= N; i ++)
    {
        if(!vis[i])
        {
            pri[pnum++] = i;
            for(int j = i+i; j <= N; j += i)
                vis[j] = true;
        }
```

```
    }
}
int n;
int fac[50];
int num[N+10];
void div_fac(int tt,int &cnt)
{
    for(int j = 0; j < pnum && pri[j] * pri[j] <= tt; j
++)
    {
        if(tt % pri[j] == 0)
        {
            fac[cnt++] = pri[j];
            while(tt % pri[j] == 0)
            {
                tt /= pri[j];
            }
        }
    }
    if(tt != 1) fac[cnt++] = tt;
}

void solve()
{
    for(int i = 1; i <= n; i ++)
    {
        int cnt = 0;
        div_fac(a[i],cnt);
        for(int j = 1; j < (1<<cnt); j ++)
        {
            int temp = 1;
            for(int k = 0; k < cnt; k ++)
                if((1<<k) & j)
                    temp *= fac[k];
            num[temp] ++;
        }
```

```
    }
}
long long get()
{
    long long ans = 0;
    for(int i = 1; i <= n; i ++)
    {
        int cnt = 0;
        div_fac(a[i],cnt);
        long long res = 0;
        for(int j = 1; j < (1<<cnt); j ++)
        {
            int temp = 1;
            int Time = 0;
            for(int k  = 0; k < cnt; k ++)
            {
                if((1<<k) & j)
                {
                    temp *= fac[k];
                    Time ++;
                }
            }
            if(Time & 1) res += num[temp];
            else res -= num[temp];
        }
        if(res == 0) continue;
        ans += (res - 1)*(n - res);
    }
    return ans;
}
int main()
{
    int t;
    ini();
    //freopen("a.txt","r",stdin);
    scanf("%d",&t);
```

```
    while(t --)
    {
        scanf("%d",&n);
        for(int i = 1; i <= n; i ++)
        {
            scanf("%d",&a[i]);
        }
        memset(num,0,sizeof(num));
        solve();
        long long ans = 1LL*n*(n-1)*(n-2)/6;
        printf("%lld\n",ans - get()/2);
    }
    return 0;
}
```

母函数

算法简述

普通母函数：对于序列$$a_0,a_1,a_2,\dotsb,$$构造一函数$$G(x)=a_0+a_1x+a_2x^2+\dotsb$$称函数G(x)是序列$$a_0,a_1,a_2,\dotsb$$的母函数。
指数型母函数：对于序列$$a_0,a_1,a_2,\dotsb,$$函数$$G(x)=a_0+a_1x+{a_2\over2!}x^2+{a_3\over3!}x^3+\dotsb$$称为序列$$a_0,a_1,a_2,...$$的指数型母函数。这样对于一个多重集，其中$$a_1$$重复了$$n_1$$次，$$a_2$$重复了$$n_2$$次，$$a_k$$重复了$$n_k$$次，如果$$n=n_1+n_2+...+n_k$$从n个元素中取r个元素排列，不同的排列数所对应的指数型母函数为$$G(x)=(1+{x\over1!}+{x^2\over2!}+\dotsb+{x^{n_1}\over n_1!})(1+{x\over1!}+{x^2\over2!}+\dotsb+{x^{n_2}\over n_2!})\dotsb(1+{x\over1!}+{x^2\over2!}+\dotsb+{x^{n_k}\over n_k!})$$。

主要运用场合及思路

母函数可以帮助我们有效的优化算法，解决问题，往往需要我们有良好的分析问题的能力，能将问题转化为母函数的模型上，而且往往需要进行数学运算求解。

模板

普通母函数(hdu2082)：

```
#include <stdio.h>
#include <string.h>
int a[55],b[55]; //a数组保存最后的结果
int main(){
    int n;
    scanf("%d",&n);
    while(n --){
        memset(a,0,sizeof(a));
        memset(b,0,sizeof(b));
        a[0] = 1;
        int num;
        for(int i = 1;i <= 26;i ++){
            scanf("%d",&num);
            for(int j = 0;j <= 50;j ++) //j表示前面i个表达式累乘得到的表达式里第j个变量
                for(int k = 0;k <= num && k*i+j <= 50;k ++)//k表示的是此时计算的多项式的第k个指数
                    b[k*i+j] += a[j];
            for(int j = 0;j <= 50;j ++){
                a[j] = b[j];
                b[j] = 0;
            }
        }
        int ans = 0;
        for(int i = 1;i <= 50;i ++)
            ans += a[i];
        printf("%d\n",ans);
    }
    return 0;
}
```

指数型母函数(hdu1521)：

```
#include <stdio.h>
#include <algorithm>
#include <stdio.h>
using namespace std;

double fac[] =
{1,1,2,6,24,120,720,5040,40320,362880,3628800};
int main(){
    double num1[11],num2[11],a[11];
    int n,m;
    while(~scanf("%d%d",&n,&m)){
        for(int i = 0;i < n;i ++)
            scanf("%lf",&a[i]);
        for(int i = 0;i <= m;i ++)
            num2[i] = num1[i] = 0.0;
        for(int i = 0;i <= a[0];i ++)
            num1[i] = 1.0 / fac[i];
        for(int i = 1;i < n;i ++){
            for(int j = 0;j <= m;j ++)
                for(int k = 0;k <= a[i] && k + j <= m;k
++)
                    num2[j+k] += num1[j]/fac[k];
            for(int j = 0;j <= m;j ++){
                num1[j] = num2[j];
                num2[j] = 0;
            }
        }
        printf("%.0lf\n",num1[m] *1.0* fac[m]);
    }
    return 0;
}
```

例题

hdu5616

题意：有n个质量已知的砝码和一个天平，砝码可以放在天平的左端或者右端，求能否称出某个质量。

思路：此题有多种解法，可以构造母函数：$$G(x) = (1+x^{\pm a_1})(1+x^{\pm a_2})\dotsb(1+x^{\pm a_n})$$，注意到此时可以把砝码放在天平的左端或者右端，所以$$a_n$$的指数的系数可正可负，最后若求能否称出$$w_i$$，只需查看$$x^{w_i}$$的系数是否为0就可以了。

参考代码：

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;

int a[2010],b[2010];
int num[25];
int main(){
    int t;
    scanf("%d",&t);
    while(t --){
        int n,sum = 0;
        scanf("%d",&n);
        for(int i = 1;i <= n;i ++){
            scanf("%d",&num[i]);
            sum += num[i];
        }
        memset(a,0,sizeof(a));
        memset(b,0,sizeof(b));
        a[0] = 1;
        for(int i = 1;i <= n;i ++){
            for(int j = 0;j <= sum;j ++)
                for(int k = 0;k <= 1 && k*num[i]+j <=
sum;k ++){
                        b[k*num[i]+j] += a[j];
```

```
                        b[abs(k*num[i]-j)] += a[j];
                }
            for(int j = 0;j <= sum;j ++){
                a[j] = b[j];
                b[j] = 0;
            }
        }
        int m;
        scanf("%d",&m);
        while(m --){
            int w;
            scanf("%d",&w);
            if(w > sum) {
                printf("NO\n");
                continue;
            }
            if(a[w]) printf("YES\n");
            else printf("NO\n");
        }
    }
    return 0;
}
```

poj3734

题意：有一排砖，数量为n，有红蓝绿黄4种颜色，其中染成红和绿颜色的砖块的数量必须为偶数个，求可有多少种染色方案。

思路：根据题意构造一个指数型母函数：$$G(x) = (1+{x \over 1!}+{x^2 \over 2!}+\dotsb)^2(1+{x^2 \over 2!}+{x^4 \over 4!}+ \dotsb)^2$$，根据泰勒展开$$e^x = 1+{x \over 1!}+{x^2 \over 2!}+\dotsb,e^{-x} = 1-{x \over 1!}+{x^2 \over 2!}+\dotsb$$，所以$$G(x) = (e^x)^2({e^x+e^{-x} \over 2})^2={e^{4x}+2e^{2x}+1 \over 4}=\sum (4^{n-1}+2^{n-1}+1){x^n \over n!}$$，所以最终答案为$$4^{n-1}+2^{n-1}$$。

参考代码：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
#define mod 10007
int quick(int a,int b){
    int ans = 1;
    while(b){
        if(b&1) ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}
int main(){
    int t;
    scanf("%d",&t);
    while(t --){
        int n;
        scanf("%d",&n);
        printf("%d\n",(quick(4,n-1)+quick(2,n-1)) % mod);
    }
    return 0;
}
```

poj 1322 Chocolate

题意：一个口袋中装有巧克力，巧克力的颜色有c种。现从口袋中取出一个巧克力，若取出的巧克力与桌上已有巧克力颜色相同，则将两个巧克力都取走，否则将取出的巧克力放在桌上。设从口袋中取出每种颜色的巧克力的概率均等。求取出n个巧克力后桌面上剩余m个巧克力的概率。

思路：由于从口袋中取n个巧克力对应有$$c^n$$种情况，这种情况会考虑到不同颜色的巧克力之间的排列关系，所以适用于指数型母函数来解决。 桌上剩余的m个巧克力颜色一定互不相同，所以此题可以转化为有m种巧克力取出了奇数次，c-m中

巧克力取出了偶数次，与上一题类似，因此我们可以构造母函数：$$G(x) = ({e^x-e^{-x} \over 2})^m ({e^x+e^{-x} \over 2})^{c-m}$$。最终的情况数就是G(x)中$$x^n$$的系数$$g_n \times n! \times C^m_c$$，n!是指数型母函数所需要乘的，$$C^m_c$$表示从c种颜色中取m种颜色取了奇数次。因此最终的结果就是$${g_n \times n! \times C^m_c \over c^n}$$。

参考代码：

```
#include <stdio.h>

double po[111], ne[111], pp[111], nn[111];

double powmod(double x, int n) {
    double ret = 1;
    while(n) {
        if(n&1) ret *= x;
        x *= x;
        n /= 2;
    }
    return ret;
}

int c, n, m;
// 由于C(n, k)可能会很大，不能直接预处理出组合数
double cal(double ret, int n, int k) {
    if(n-k < k) k = n-k;
    for(int i = n;i > n-k; i--)
        ret *= i;
    for(int i = 1;i <= k; i++)
        ret /= i;
    return ret;
}

void solve() {
    int i, j;
    for(i = 0;i <= c; i++)  {
```

```
        po[i] = ne[i] = pp[i] = nn[i] = 0;
    }
    double chu = powmod(1.0/2, m);
    for(i = 0;i <= m; i++) {
        int now = i-m+i;
        int flag = 1;
        if((m-i)&1) flag = -1;
        if(now >= 0) po[now] += cal(chu*flag, m, i); //
保存e^(kx)的系数
        else    ne[-now] += cal(chu*flag, m, i);  // 保存
e^(-kx)的系数
    }
    chu = powmod(1.0/2, c-m);
    for(i = 0;i <= c-m; i++) {
        double cur = cal(chu, c-m, i);
        for(j = 0;j <= m; j++) {
            int now = j+i-(c-m)+i;
            if(now >= 0)    pp[now] += po[j]*cur;  // 直
接合并系数
            else    nn[-now] += po[j]*cur;
        }
        for(j = 0;j <= m; j++) {
            int now = -j + i-(c-m)+i;
            if(now >= 0)    pp[now] += ne[j]*cur;
            else    nn[-now] += ne[j]*cur;
        }
    }
    double ans = 0;
    for(i = 1;i <= c; i++) {
        ans += cal( pp[i]*powmod((double)i/c, n),  c, m);
    }
    for(i = 1;i <= c; i++) {
        if(n&1) nn[i] = -nn[i];
        ans += cal( nn[i]*powmod((double)i/c, n),  c, m);
    }
    printf("%.3lf\n", ans);
```

```
}

int main() {
    while(scanf("%d", &c) != -1 && c) {
        scanf("%d%d", &n, &m);
        if(m > n || m > c || (n-m)%2==1) {
            puts("0.000");  continue;
        }
        // 尤其要注意n等于0 && m等于0 要特判
        if(n == 0 && m == 0) {
            puts("1.000");  continue;
        }
        solve();
    }
    return 0;
}
```

polya定理

算法简述

Burnside定理：设G是$$N={1,2,\dotsb,n}$$上的置换群，G是N上可引出不同的等价类，其中不同的等价类的个数为$${1 \over |G|} \sum c_1(g)$$,其中，$$c_1(g)$$为置换g中不变元的个数，及g中1阶循环的个数。

polya定理：设$$G={a_1,a_2,\dotsb,a_g}$$是n个对象的置换群，用m种颜色给这n个对象着色，则不同的着色方案数为：$${1\over|G|}{m^{c(a_1)}+m^{c(a_2)}+\dotsb+m^{c(a_g)}}$$。其中$$c(a_i)$$为置换$$a_i$$的循环节数$$(i=1 \dotsb g)$$。

主要运用场合及思路

Burnside定理和Polya定理是组合数学中，用来计算全部互异的组合状态的个数的一个十分高效、简便的工具。一般在题目中出现经过旋转、翻转等方式重合的计数问题时往往会用到polya定理或Burnside定理，但由于题目中的取值范围很大，有时就会用到欧拉函数进行优化。
它们求解的一般步骤是：

1. 确定置换群。
2. polya：计算每个置换下循环节个数。Burnside：求解每个置换下本质不同的方案数即在该置换下保持不变的方案数。有时会用到组合数学或动态规划的方法进行计数。
3. 代入公式得到答案。

模版

1. poj2409(裸的polya)

```
#include<cstdio>
typedef long long LL;
LL Pow(LL a, LL b) {
 LL ans;
 for (ans = 1; b; b >>= 1) {
     if (b & 1)
         ans *= a;
     a *= a;
 }
 return ans;
}
int GCD(int x, int y) {
 return y ? GCD(y, x % y) : x;
}
int main() {
 int n, k, i;
 LL ans;
 while (scanf("%d%d", &k, &n), n || k) {
     if (n & 1)
         ans = Pow(k, n / 2 + 1) * n;
     else
         ans = Pow(k, n / 2 + 1) * (n / 2)
                 + Pow(k, n / 2) * (n / 2);
     for (i = 1; i <= n; i++)
         ans += Pow(k, GCD(n, i));
     ans = printf("%lld\n", ans / (2 * n));
 }
 return 0;
}
```

2. 用欧拉函数进行优化，可用dfs或循环的方式枚举因子，一般形式如下：

```
//循环枚举因子。
for(int i = 1;i*i <= n;i ++){
        if(i * i == n){
            ans = (ans + pow(n,i-1,p)*eular(i,p)%p)
% p;
        }
        else if(n % i == 0){
            ans += (pow(n,i-1,p)*eular(n/i,p)%p +
pow(n,n/i-1,p)*eular(i,p)%p) % p;
            ans %= p;
        }
    }
dfs枚举因子。cntf为素因子个数，num数组存放素因子
void dfs(int i, int now) {
 if(i == cntf) {
     ans = (ans + eular(now)*pow(n, n/now - 1))%MOD;
     if(now*now != n)    ans = (ans +
eular(n/now)*pow(n, now - 1))%MOD;
     return ;
 }
 int tmp = 1, j;
 for(j = 0; j <= num[i]; ++j) {
     if(LL(now*tmp)*LL(now*tmp) > n) return ;
     dfs(i + 1, now*tmp);
     tmp *= fac[i];
 }
}
```

例题

hdu5080 Colorful Toy

题意：平面上有n个点m条边，用c种颜色给这m条边染色，如果两个方案经过旋转后重合则视为一种方案，求有多少种染色方案。

思路：此题就是简单的polya定理的应用，关键在于枚举旋转角度后对应点的处理。

参考代码：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
#include <vector>
using namespace std;
#include <math.h>
#define LL long long

#define EXIT exit(0);
#define DEBUG puts("Here is a BUG");
#define CLEAR(name, init) memset(name, init,
sizeof(name))
const double eps = 1e-8;
const int MAXN = (int)1e9 + 5;
using namespace std;
const LL mod = 1LL*1000000007;
#define Vector Point

int dcmp(double x)
{
    return fabs(x) < eps ? 0 : (x < 0 ? -1 : 1);
}

struct Point
{
    double x, y;

    Point(const Point& rhs): x(rhs.x), y(rhs.y) { }
//拷贝构造函数
    Point(double x = 0.0, double y = 0.0): x(x), y(y) { }
//构造函数
```

```
    friend istream& operator >> (istream& in, Point& P)
    {
        return in >> P.x >> P.y;
    }
    friend ostream& operator << (ostream& out, const
Point& P)
    {
        return out << P.x << ' ' << P.y;
    }

    friend Vector operator + (const Vector& A, const
Vector& B)
    {
        return Vector(A.x+B.x, A.y+B.y);
    }
    friend Vector operator - (const Point& A, const
Point& B)
    {
        return Vector(A.x-B.x, A.y-B.y);
    }
    friend Vector operator * (const Vector& A, const
double& p)
    {
        return Vector(A.x*p, A.y*p);
    }
    friend Vector operator / (const Vector& A, const
double& p)
    {
        return Vector(A.x/p, A.y/p);
    }
    friend bool operator == (const Point& A, const Point&
B)
    {
        return dcmp(A.x-B.x) == 0 && dcmp(A.y-B.y) == 0;
    }
```

```
    friend bool operator < (const Point& A, const Point&
B)
    {
        return A.x < B.x || (A.x == B.x && A.y < B.y);
    }
    friend bool operator != (const Point &A,const Point
&B)
    {
        return dcmp(A.x-B.x) != 0 || dcmp(A.y-B.y) != 0;
    }
    void in(void)
    {
        scanf("%lf%lf", &x, &y);
    }
    void out(void)
    {
        printf("%lf %lf", x, y);
    }
}p[55],p2[55];


double Dot(const Vector& A, const Vector& B)
{
    return A.x*B.x + A.y*B.y;    //点积
}
double Length(const Vector& A)
{
    return sqrt(Dot(A, A)+eps);
}
double Angle(const Vector& A, const Vector& B)
{
    return acos(Dot(A, B)/Length(A)/Length(B));    //向量
夹角
}
double Cross(const Vector& A, const Vector& B)
{
```

```
    return A.x*B.y - A.y*B.x;    //叉积
}
//向量绕起点旋转
Vector Rotate(const Vector& A, const double& rad)
{
    return Vector(A.x*cos(rad)-A.y*sin(rad),
A.x*sin(rad)+A.y*cos(rad));
}

bool cmp(int A,int B)
{
    if(dcmp(Cross(p[A],p[B]) != 0))
    {
        return Cross(p[A],p[B]) > 0;
    }
    else return Length(p[A]) < Length(p[B]);
}

 int n,m,c;
 int Next[55];
 bool flag;

 vector <int> vec;
 int map[55][55];
bool check1(double ang,int dif){
    int sz = vec.size();
    for(int i = 0;i < sz;i ++){
        p2[Next[vec[i]]] = Rotate(p[vec[i]],ang);
        if(p2[Next[vec[i]]] != p[Next[vec[i]]]) return
false;
    }
    return true;
}
bool check2(int dif){
        for(int i = 0;i < n;i ++)
            for(int j = 0;j < n;j ++)
```

```
                if(map[i][j] != map[Next[i]][Next[j]])
return false;
    return true;
}
long long quick_mod(LL a,int k){
    LL ans = 1;
    while(k){
        if(k&1){
            ans *= a;
            ans %= mod;
        }
        a = a * a;
        a %= mod;
        k >>= 1;
    }
    return ans;
}
bool vis[100] ;
long long cal(int dif){
    memset(vis,false,sizeof(vis));
    int num = 0;
    for(int i = 0;i < n;i ++){
        if(!vis[i]){
            vis[i] = true;
            num ++;
            int j = Next[i];
            while(j != i){
                vis[j] = true;
                j = Next[j];
            }
        }
    }
    return quick_mod(1LL*c,num);
}
void gcd(LL a,LL b,LL& d,LL & x,LL & y){
    if(!b) { d = a ; x = 1;y = 0;}
```

```
    else {gcd(b,a%b,d,y,x); y-=x*(a/b);}
}
LL inv(LL a){
    LL d,x,y;
    gcd(a,mod,d,x,y);
    return d == 1 ? (x + mod) % mod : -1;
}
int main(){
    int t;
//    freopen("a.txt","r",stdin);
    scanf("%d",&t);
    while(t --){
        scanf("%d%d%d",&n,&m,&c);
        double sumx = 0,sumy = 0;
        vec.clear();
        memset(map,0,sizeof(map));
        for(int i = 0;i < n;i ++){
            p[i].in();
            sumx += p[i].x;
            sumy += p[i].y;
        }
        Point center(sumx/n,sumy/n);
        for(int i = 0;i < n;i ++){
            p[i] = p[i]-center;
        }
        flag = false;
        center.x = 0;
        center.y = 0;
        for(int j = 0;j < n;j ++){
            if(p[j] == center) Next[j] = j;
            else vec.push_back(j);
        }
        for(int i = 0;i < m;i ++){
            int u,v;
            scanf("%d%d",&u,&v);
            u--;v--;
```

```
            map[u][v]  = map[v][u] = 1;

        }
        int ti = 0;
        long long ans = 0;
        int sz = vec.size();
        sort(vec.begin(),vec.end(),cmp);

        for(int i = 0;i < sz;i ++){

                for(int j = 0;j < sz;j ++){
                    Next[vec[j]] = vec[(j+i)%sz];
                }

            Vector tmp(p[vec[0]]);
            Vector tmp2(p[Next[vec[0]]]);

            double ang = atan2(tmp2.y,tmp2.x) -
atan2(tmp.y,tmp.x);

            int dif  = i;//
            if(check1(ang,dif) && check2(dif)){
                ans += cal(dif);
                ans %= mod;
                ti ++;
            }
        }

        ans *= inv(ti);
        ans %= mod;
        printf("%lld\n",ans);
    }
    return 0;
}
```

bzoj1488

题意：求两两互不同构的含n个点的简单图有多少种。

思路：点重新标号的过程就是置换的过程，对于两点之间有无边相连可以想象成有每条边可以染无色和有色两种颜色，那么我们就需要求出每种置换对应边循环的个数。经过分析我们可以知道每一种共轭类对应的置换其边置换是一样的。然后我们考虑边置换的循环节个数，当一条边的两个端点在一个长度为$$L_i$$的循环中时会产生$$L_i/2$$个边循环节，当一条边的两个端点分别在两个长度为$$L_i,L_j$$的循环中时会产生$$gcd(L_i,L_j)$$个边循环。对于形如$$(1)^{c_1}(2)^{c_2}\dotsb(n)^{c_n} $$的共轭类，会有$${n! \over c_1!c_2!\dotsb c_n!1^{c_1}2^{c_2}\dotsb n^{c_n}}$$个元素，我们可dfs枚举每一种共轭类便可求解。

参考代码：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;

#define N 65
#define mod 997
int fac[N];
int n;
int quick(int n,int k){
    int ans = 1;
    while(k){
        if(k & 1) ans = ans * n % mod;
        n = n * n % mod;
        k >>= 1;
    }
    return ans;
}
int _inv(int n){
    return quick(n,mod-2);
```

```
}
void ini(){
    fac[0] = 1;
    for(int i = 1;i < N;i ++){
        fac[i] = fac[i-1] * i % mod;
    }
}
int num[N][2];
int cnt,ans;
int gcd(int n,int m){
    return m == 0 ? n : gcd(m,n % m);
}
void cal(){
    int x = 0;
    for(int i = 0;i < cnt;i ++){
        x += num[i][1]*(num[i][1]-1)/2 * num[i][0] +
num[i][0] / 2 * num[i][1];
        for(int j = i+1;j < cnt;j ++){
            x += num[i][1] * num[j][1] * gcd(num[i]
[0],num[j][0]);
        }
    }
    int res = quick(2,x);

    res = res * fac[n] % mod;
    for(int i = 0;i < cnt;i ++){
        res = res* _inv(fac[num[i][1]]) % mod *
_inv(quick(num[i][0],num[i][1])) % mod;
        res %= mod;
    }
    ans = (ans + res) % mod;
}
void dfs(int now,int left){
    if(left == 0){
        cal();
        return ;
```

```
    }

    if(now > left) return ;
    dfs(now + 1,left);

    for(int i = 1;i * now <= left;i ++){
        num[cnt][0] = now;
        num[cnt++][1] = i;
        dfs(now+1,left-i*now);
        cnt --;
    }
}

int main(){
    ini();
    while(~scanf("%d",&n)){
        cnt = 0,ans = 0;
        dfs(1,n);

        ans = ans * _inv(fac[n]) % mod;
        printf("%d\n",ans);
    }
    return 0;
}
```

poj2154 Color

题意：n种颜色对一串n个珠子的项链进行染色，求有多少种染色方案(若两条项链经过旋转后能完全重合，则视为同种方案)(n<=1000000000)。

思路：此题为经典的polya题目，将项链顺时针旋转i格后，其循环节个数为$$gcd(n,i)$$，所以此题的最终结果应为$${1 \over n} \sum n ^ {gcd(n,i)}$$。但是n的数目为1000000000，枚举i会超时。我们不妨换个角度想，$$gcd(n,i)$$的个数是很少的，所以我们可以枚举每个$$gcd(n,i)$$然后乘上相应的个数就是答案了。假

设$$gcd(n,i)=x_i,a_ix_i=n,b_ix_i=i则gcd(a_i,b_i)=1$$，所以$$gcd(n,i)$$的个数也就是所有不大于$$a_i$$的数中与其互质的数目，我们可以用欧拉函数或容斥原理求得。

参考代码：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;

#define N 36000
int pri[N];
bool is[N];
int pcnt;

void ini(){
    pcnt = 0;
    memset(is,false,sizeof(is));
    for(int i = 2;i < N;i ++){
        if(!is[i]){
            pri[pcnt ++] = i;
            for(int j = i+i;j < N;j += i)
                is[j] = true;
        }
    }
}
int eular(int n,int p){
    int ans = n;
    for(int i = 0;i < pcnt && pri[i] * pri[i] <= n;i ++){
        if(n % pri[i] == 0){
            ans = ans - ans / pri[i];
            while(n % pri[i] == 0)
                n /= pri[i];
        }
```

```
    }
    if(n > 1) ans = ans - ans/n;
    return ans % p;
}
int quick(int n,int k,int p){
    int ans = 1;
    n %= p;
    while(k){
        if(k&1) ans = ans * n % p;
        n = n * n % p;
        k >>= 1;
    }
    return ans;
}

int main(){
    ini();
    int t;
    scanf("%d",&t);
    while(t --){
        int n,p;
        scanf("%d%d",&n,&p);
        int ans = 0;
        for(int i = 1;i*i <= n;i ++){
            if(i * i == n){
                ans = (ans + quick(n,i-1,p)*eular(i,p)%p)
% p;
            }
            else if(n % i == 0){
                ans += (quick(n,i-1,p)*eular(n/i,p)%p +
quick(n,n/i-1,p)*eular(i,p)%p) % p;
                ans %= p;
            }
        }
        printf("%d\n",ans);
    }
```

```
    return 0;
}
```

POJ 2888 Magic Bracelet

题意：给一条n个珠子的项链用m种颜色进行染色，其中有k对颜色$$(c_i,c_j)$$不能出现在相邻的珠子上，如果两种方案可通过旋转使其一致则视为一种方案，求有多少种不同方案数。

思路：此题可用Burnside定理做，一共有n种置换，对于每种置换我们需要求出在该置换下保持不变的着色方案数。由于项链每旋转i个珠子，就会有$$n/gcd(n,i)$$个长度为$$L_i=gcd(n,i)$$的循环节，而且可以发现项链中任意相邻的长度为$$L_i$$的珠子处于不同的循环中，所以此时保持不变的着色方案数等价于长度为$$gcd(n,i)$$的项链满足合法条件的方案数。构造一个合法矩阵g，其中$$g(a_i,a_j)=1$$表示颜色$$a_i,a_j$$可以相邻，$$g(a_i,a_j)=0$$表示不可以，将其看作一个无向图，那么$$g^k$$表示一个点经过k条路之后到达某点的方法数，此时$$g(a_i,a_i)$$表示从$$a_i$$种颜色开始染色最后染完一条项链的方案数，将其累加就是最后的答案。

参考代码：

```
#include <iostream>
#include <cstring>
#include <cstdio>
using namespace std;
const int max_n=15;
const int MOD=9973;
int N;

struct Mat
{
    int mat[max_n][max_n];
};

Mat operator*(Mat a,Mat b)
{
    Mat c;
```

```
    memset(c.mat,0,sizeof(c.mat));
    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            for(int k=0; k<N; k++)
            {
                if(a.mat[i][k] && b.mat[k][j])
                {
                    c.mat[i][j]=(c.mat[i][j]+a.mat[i]
[k]*b.mat[k][j])%MOD;
                }
            }
        }
    }
    return c;
}

Mat operator^(Mat A,int x)
{
    Mat c;
    memset(c.mat,0,sizeof(c.mat));
    for(int i=0; i<N; i++)    c.mat[i][i]=1;
    for(; x; x>>=1)
    {
        if(x&1)
        {
            c=c*A;
        }
        A=A*A;
    }
    return c;
}

int mypow(int a,int b)
{
```

```
    int res=1;
    for(; b; b>>=1)
    {
        if(b&1)
        {
            res*=a;
            res%=MOD;
        }
        a*=a;
        a%=MOD;
    }
    return res;
}

int M,K;

int PHI(int n)
{
    int i,res=n;
    long long j;
    for(i=2,j=4LL; j<=(long long)n; i++,j+=i+i-1)
    {
        if(!(n%i))
        {
            res=res/i*(i-1);
            while(!(n%i))    n/=i;
        }
    }
    if(n>1)    res=res/n*(n-1);
    return res%MOD;
}

Mat A;
int solve(int p)
{
    int res=0;
```

```
    Mat C=A^p;
    for(int i=0; i<N; i++)
    {
        res+=C.mat[i][i];
        res%=MOD;
    }
    return res;
}
int main()
{
    int T;

    scanf("%d",&T);

    for(int ncas=1; ncas<=T; ncas++)
    {
        scanf("%d%d%d",&M,&N,&K);
        int u,v;
        for(int i=0; i<N; i++)
        {
            for(int j=0; j<N; j++)
            {
                A.mat[i][j]=1;
            }
        }
        for(int i=0; i<K; i++)
        {
            scanf("%d%d",&u,&v);
            A.mat[u-1][v-1]=A.mat[v-1][u-1]=0;
        }

        int ans=0;

        for(int p=1; p*p<=M; p++)
        {
            if(M%p==0)
```

```
            {
                if(p*p==M)
                {
                    ans = (ans+ PHI(p)*solve(p) )%MOD;
                }
                else
                {
                    ans = (ans+ PHI(p)*solve(M/p)+
PHI(M/p)*solve(p) )%MOD;
                }
            }
        }
        int inv=mypow((M%MOD),MOD-2);
        ans*=inv;
        ans%=MOD;
        printf("%d\n",ans);
    }
    return 0;
}
```

搜索

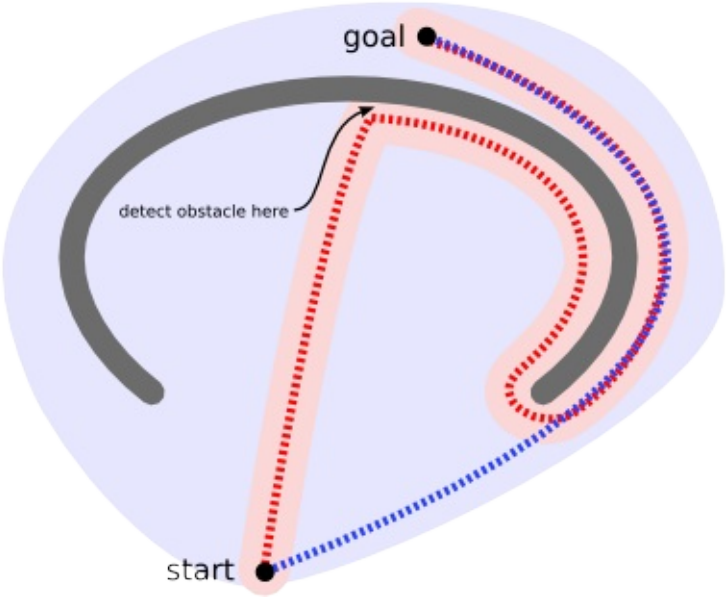
---

author：李宝佳

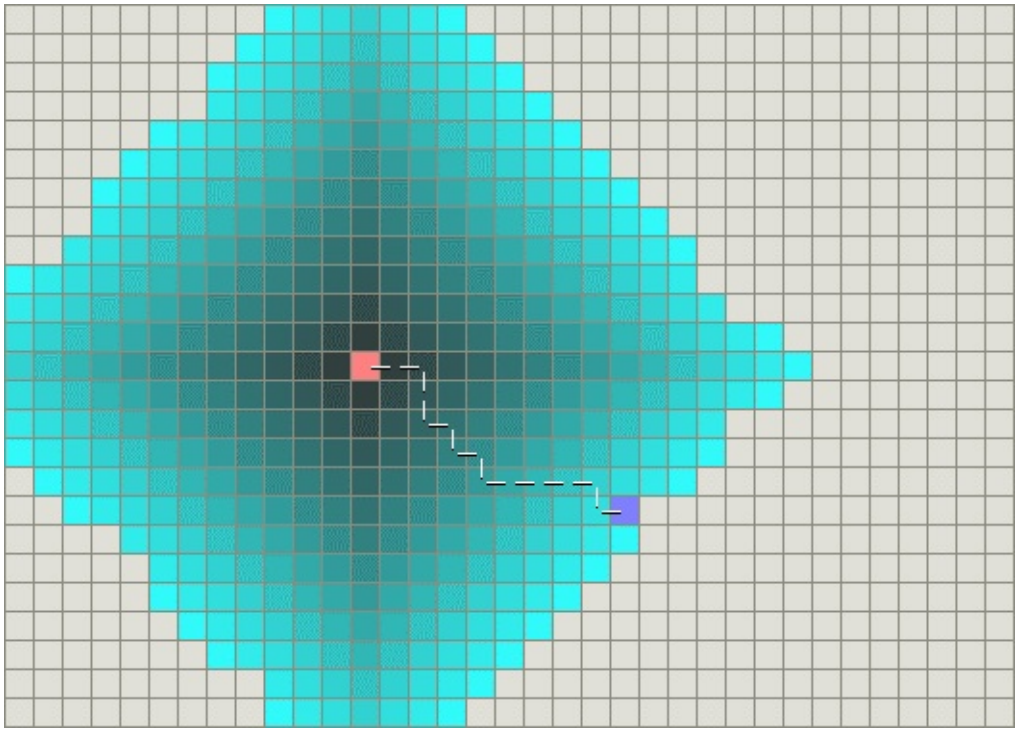
a*算法

算法简述

我们尝试解决的问题是：对象从起点出发，移动到终点。路径搜索的目标：找到一条路径，避免障碍物，敌人等，并且把代价（时间，距离，金钱等）最小化。
移动一个简单的对象看起来是容易的，但是路径搜索可能是比较复杂的。考虑下面情况：



初始节点 start ，目标节点 goal，灰色部分表示障碍物，第一种情况，我们只考虑怎样能使得我们目前所花费的是最少，搜索到的路径会如图中粉色部分所示。同时它所搜索的范围是淡粉色部分，另一种情况，我们考虑所要走的下一个节点，是下一步的所有可能中，距离goal的花费是最少的。我们所得到的路径时蓝色部分，搜索的范围是淡蓝色部分。
注意以上是两种搜索方法，两种算法所考虑的代价是不同的，这就直接影响了我们的搜索代价。并且从图中两种方法的搜索路径和扫描范围可以看出，第1种情况搜索代价少，但是所花费的最终结果并不是最优的。想法第2种情况最终花费虽然是最优的，但是搜索代价非常大。

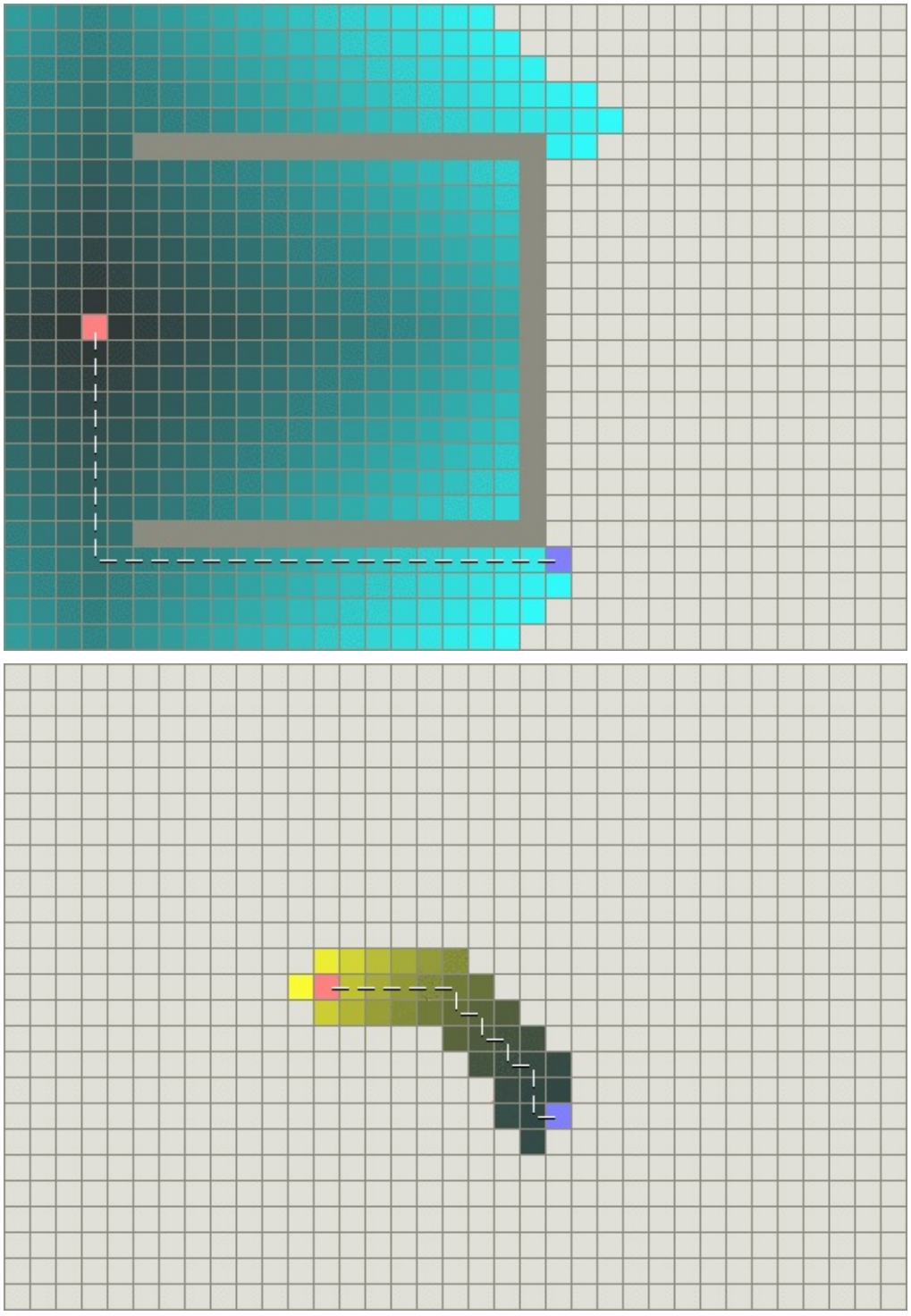


dijkstra’s algorithm and best-first-search

在介绍a*之前，我们先来看这两种算法。 dijkstra算法从物体所在的初始节点开始，在访问途中的节点时候，它迭代检查的是当前节点的临近节点，并且把和该节点最近的尚未访问的节点加入待访问队列，直到到达目标节点。dijkstra算法保证能在一般简单且边是非负的图中，找到一条从初始节点到目标节点的最短路径。下图中粉色代表初始节点，紫色代表目标节点，蓝色表示扫描过的区域。颜色越浅表示离初始节点越远。

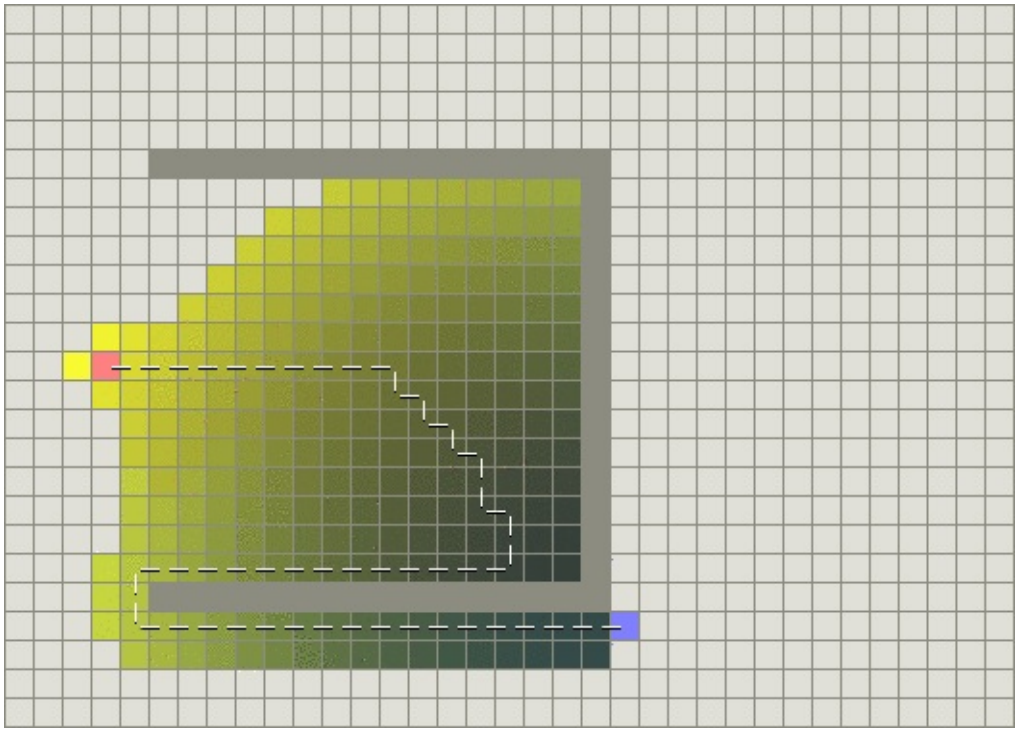
最佳优先搜算法与dijkstra算法的不同之处就在于其是采用启发式函数。它能够评估任意节点到目标节点的代价。与选择距离初始节点最近的算法不同。它选择离目标节点的估计代价最小的节点。最佳优算法不能保证一定会找到一条路径。但是相同情况下它会比dijkstra算法快很多。图中黄色渐变部分表示最佳优先搜索算法所扫描过的部分，其颜色越深表示离目的节点越近。

那么现在来让我们来看一个相对较复杂的图。下图较上图中多加了一个u型的障碍物。



此障碍物不可穿越。

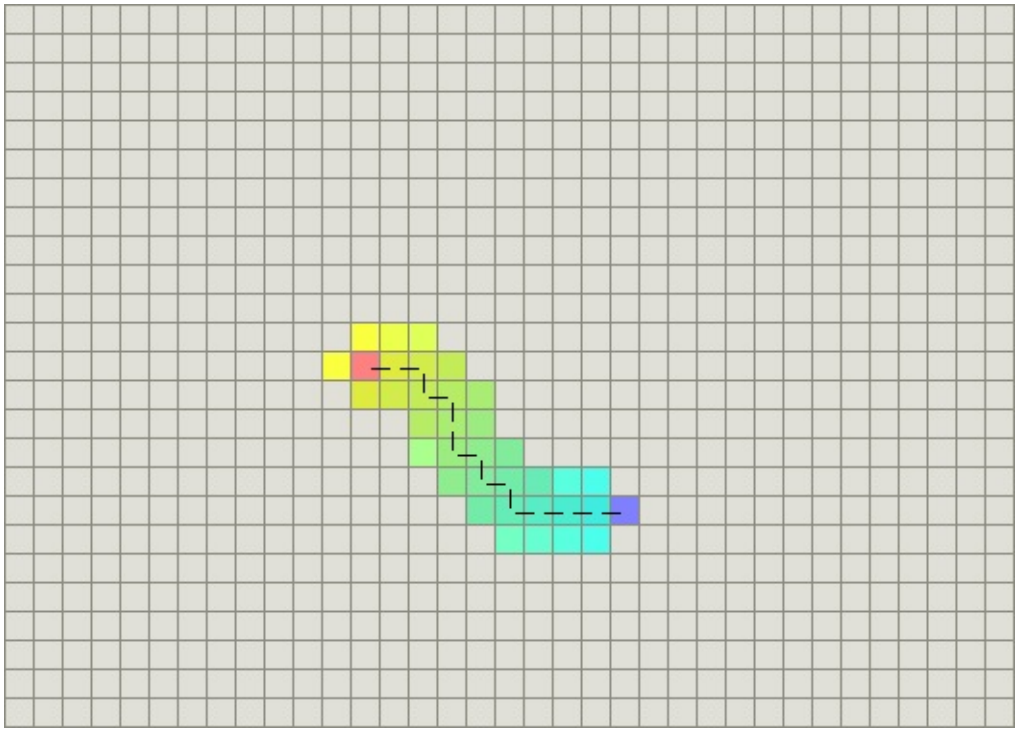
上图中虚线表示用dijstra算法所找的的路径，同样淡蓝色表示扫码过的路径。由此可看出dijkstra确实可以确保找的一条路径，但是扫描的代价很大。



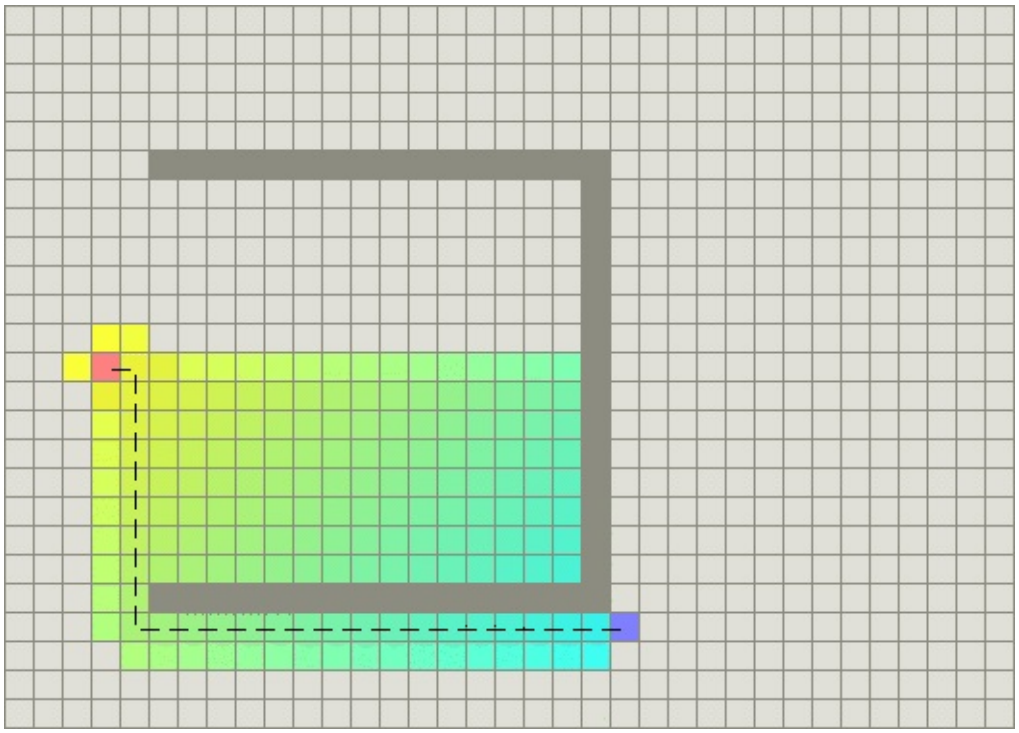
上图中采用最佳优先搜索算法，可以看出虽然最佳优先搜索算法虽在扫描小部分的情况下可以找的一条路径，但是并不是最优的。
问题就在于最佳优先搜索算法是基于贪心的。它是试图尽可能的向目标节点移动，尽管这条路并不是很正确。它仅仅考虑了到达目标节点的代价，而并未考虑到目前为止已花费的代价。
基于两者的缺点和优点，1968年发明了a*算法。其把启发式算法和常规的算法结合在了一起，但是与单纯的启发式算法不同。在村庄最短路径的情况下，a*保证一定能找到最短路径。

主要运用场合及思路

a*算法；a*（a-star)算法是一种静态路网中求解最短路径最有效的直接搜索方法。估价值与实际值越接近，估价函数取得就越好。

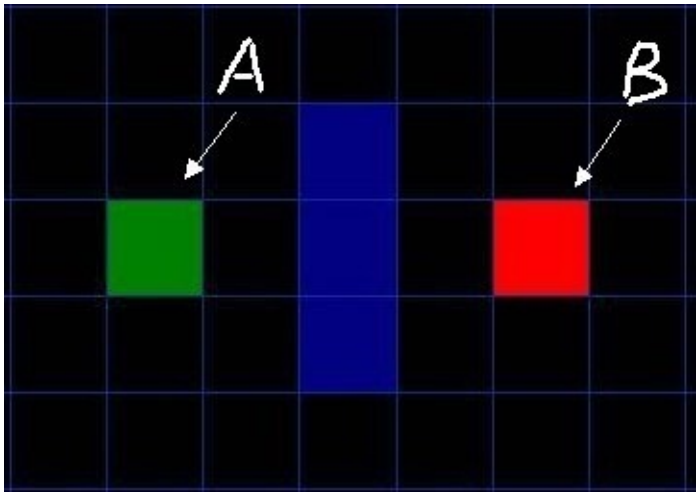


a*采用启发式函数，在简单情况下它和最佳优先算法一样快。



在有障碍物时，a*和dijkstra算法一样可以找的一条最优路径。

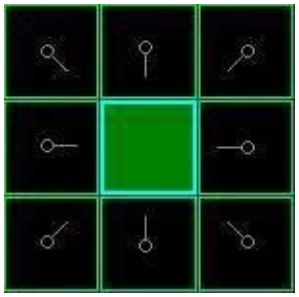
a*算法把dijkstra算法和最佳优先算法相结合，g(n)表示从初始节点到当前节点n的代价，h(n)表示从当前节点n到目标节点的启发式估计代价。a*权衡这两者，每次进行主循环时，它检测f(n)中最小的节点n,其中f(n) = g(n) + h(n)。



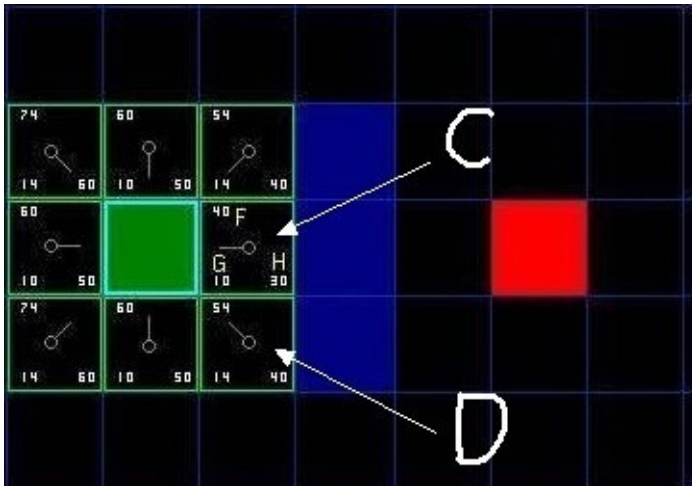
循环时，它检测f(n)中最小的节点n,其中f(n) ＝ g(n) + h(n)。
首先，如图所示简易地图, 其中绿色方块的是起点 (用 a 表示), 中间蓝色的是障碍物, 红色的方块 (用 b 表示) 是目的地.
假设我们当前求的是最短路，那g(n)就表示从起点到其的最短代价，而h(n)就表示从当前节点到目标节点的估计代价。

具体寻路步骤：

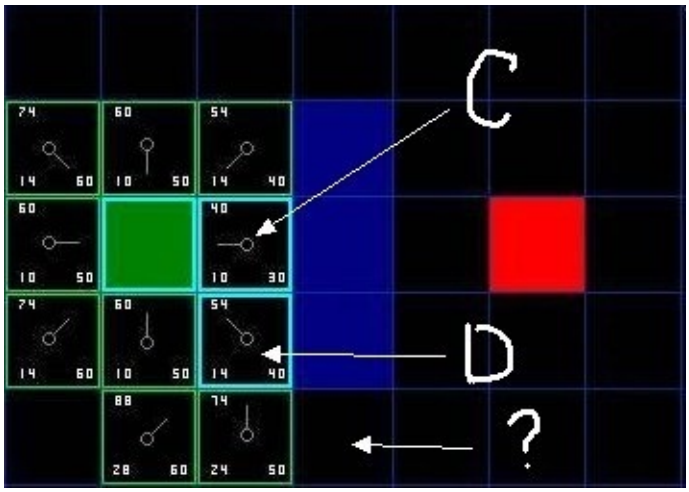
1. 从起点开始，把它作为一个待处理的节点，存入待处理队列。这个队列就是一个等待处理的队列。
2. 取起点a可以到达的所有未访问过的邻居节点，将他们放入队列，并设置他们的父亲节点为a，
3. 在待处理队列中删除起点a，节点并标记已访问过，不需要再次访问。



图中浅绿色描边的方块表示已经加入待处理队列，等待处理. 荧光标记的起点 a 表示已经放入 已访问队列, 它不需要再执处理。
从待处理队列中找出相对最优的节点, 那么什么是最优呢，最优世通过公式f(n) ＝ g(n) + h(n) 来计算的.（g(n)表示从初始节点到当前节点n的代价，h(n)表示从当前节点n到目标节点的启发式估计代价）。
我们假设横向移动一个格子的耗费为10, 为了便于计算, 沿斜方向移动一个格子耗费是14. 图中方块的左上角数字表示 f(n), 左下角表示 gg(n) 右下角表示 h(n) 看看是否跟你心里想的结果一样?



4. 从待处理队列中选择 f(n) 值最低的方格 ，本图当前步骤是c (绿色起始方块 a 右边的方块), 把其从待处理队列中删除，并放入到已访问队列中。
5. 检查其所有可到达的未访问过的队列中的节点，如果其不在待处理队列中，则把他们加入待处理队列，否则转到步骤6，计算这些节点的f(n), g(n), h(n)。并设置他们的父亲节点为c。

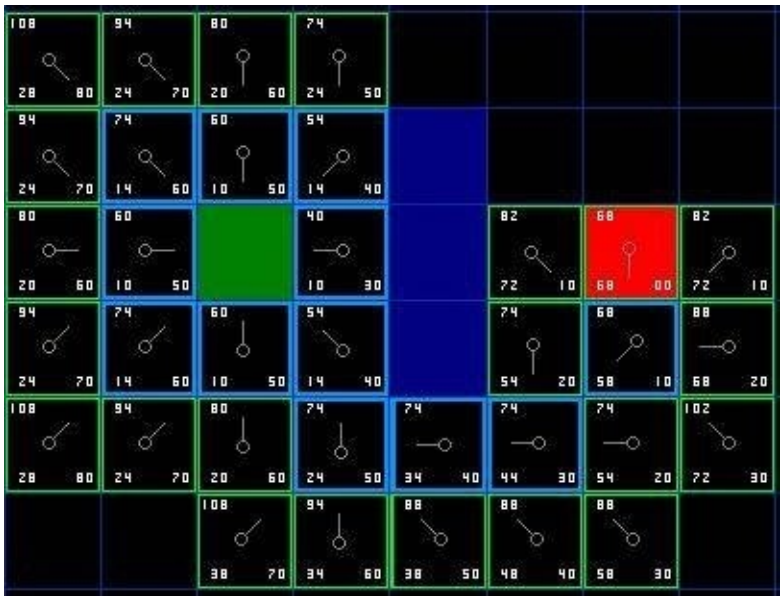


6. 对于已经存在待处理队列中的节点（如d），检查通过当前节点（即本例c）到达它的话，g(n)的值是否更优一些，如果是，那么就把其父亲节点标记成当前节点（本例中c）然后从新计算其 f(n), g(n)， 不过h(n) 不需要计算（其实对于本例求最短路中，每个及诶单的h(n)是不变的）。如果经过当前节点得到的代价反而不是较优的话，我们什么也不用做。

   如上图, 我们选中了 c 因为它的 f(n) 值最小, 我们把它从待处理对了中删除, 并把它加入 已访问队列. 它右边上下三个都是墙, 所以不考虑它们. 它左边是起始方块, 已访问过了, 也不考虑. 所以它周围的候选方块就只剩下 4 个. 让我们来看看 c 下面的那个格子, 它目前的 g 是14, 如果通过 c 到达它的话, g将会是 10 + 10, 这比 14 要大, 因此我们什么也不做.

   然后我们继续从待处理队列中找出 f 值最小的（即重复4，5，6步骤）, 但我们发现 待处理队列中，处于c 上面的和下面的同时为 54, 这时怎么办呢? 这时随

便取哪一个都行, 比如我们选择了 c 下面的那个方块 d.
d 右边以及右上方的都是墙, 所以不考虑, 但为什么右下角的没有被加进 待处理队列呢? 因为如果 c 下面的那块也不可以走, 想要到达 c 右下角的方块就需要从 "方块的角" 走了, 在程序中设置是否允许这样走. (图中的示例不允许这样走)
我们一直重复这些步骤知道我们访问到目标节点为止。



7. 起时我们得到路径的过程是一个回溯的过程，因为之前已经记录了各个节点的父亲节点。 伪代码：

```
do
{
 fleast=heap.pop();                //从queue取fcost最小的元素
 current=fleast;                        //取出的元素作为当前节点
 least.flag = 1 //1:visited 0:not on queue not visited
 for(int i=current.x-1;i<current.x+1;i++)      //考察当前节点的所有相邻节点
 for(int j=current.y-1;j<current.y+1;j++)
 {
  if(map[i][j]!=unwalkable||!onclose(node(i,j)))
//相邻节点可通并且没有考察过
  {
  if(!onopen(node(i,j)))
//相邻节点不在queue中
  {
  heap.add(node(i,j));
```

```
//加入queue
  node(i,j).parent=current;                                    //
设置相邻节点的父节点为当前节点
  node(i,j).calculate(f,g,h);
//重新计算fcost,gcost,hcost

  if(node(i,j)==dest)
//如果加入的节点是目标点则找到路径
  path=find;
  }
  else
  {
  temp.gcost=parent.gcost+addcost;          //相邻节点已经在
queue中
  node(i,j).hcost=10*(abs(i-parent.x)+abs(j-parent.y));
  node(i,j).fcost=node(i,j).gcost+node(i,j).hcost;
  if(tempgcost<node(i,j).gcost)
  node(i,j).parent=current;
  node(i,j).recaculate(f,g,h);
//重新计算fcost,gcost,hcost

 }
 }

}while(!hp.isempty())                                          //
如果堆空则没有找到路径
if(path==find)
outpath;
//输出路径
else
cout<<"path not find;!"<<endl;
```

模板代码

```
#define maxn  111111
#define inf 0x1f1f1f1f
int head[maxn], head1[maxn], ecnt1, ecnt2;
int dist[maxn];
bool vis[maxn];
int sta, end;
struct edge
{
    int v, t, next;
    edge(){}
    edge( int _v, int _t)
    {
        v = _v;
        t = _t;
    }
}e[maxn], e1[maxn];

void init()
{
    memset(head, -1, sizeof(head));
    memset(head1, -1, sizeof(head1));
    ecnt1 = ecnt2 = 0;
}
void addedge( int u, int v, int t)
{
    e[ecnt1] = edge(v,t);
    e[ecnt1].next = head[u];
    head[u] = ecnt1++;


    e1[ecnt2] = edge(u,t);
    e1[ecnt2].next = head1[v];
    head1[v] = ecnt2++;
}
```

```
struct point
{
    int v, g;
    point(){};
    point(int _v, int _g)
    {
        v = _v;
        g = _g;
    }
    bool operator <(const point &a) const
    {
        return dist[v] + g > dist[a.v] + a.g;
    }
};
 priority_queue<point> que;

int a_start( int k)
{
    while(!que.empty()) que.pop();
    point now, tem;
    now = point(sta, 0);
    que.push(now);


    while(!que.empty())
    {
        now = que.top();
        que.pop();
        if(now.v == end)
         {
            if(k > 1)
              k--;
            else
             return now.g;
         }
        for( int i = head[now.v]; i != -1; i = e[i].next)
```

```
            {
                que.push(point(e[i].v,e[i].t + now.g));
            }
        }
        return -1;
    }
```

POJ 2449

题目描述：给出一个图，一个起点个一个终点，求这两点间的第k短路。
思路及关键的：是可以走重复的路的，所以如果一张图中有一个环的话，无论求第几短路都是存在的。 对于a* ，估价函数 = 当前值+当前位置到终点的距离，即f（p）=g（p）+h（p），每次扩展估价函数值中最小的一个。对于k短路来说，g（p)为当前从s到p所走的长度，h（p）为从p到 t 的最短路的长度，则f（p）的意义就是从s按照当前路径走到 p 后要走到终点 t 一共至少要走多远。也就是说我们每次的扩展都是有方向的扩展，这样就可以提高求解速度和降低扩展的状态数目。为了加速计算，h（p）需要从a*搜索之前进行预处理，只要将原图的所有边反向，再从终点 t 做一次单源最短路径就可以得到每个点的h（p）了。

第一种方法是a*+dijkstra

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <queue>
using namespace std;
#define maxn  111111
#define inf 0x1f1f1f1f
int head[maxn], head1[maxn], ecnt1, ecnt2;
int dist[maxn];
bool vis[maxn];
int sta, end;
struct edge
{
```

```
    int v, t, next;
    edge(){}
    edge( int _v, int _t)
    {
        v = _v;
        t = _t;
    }
}e[maxn], e1[maxn];

void init()
{
    memset(head, -1, sizeof(head));
    memset(head1, -1, sizeof(head1));
    ecnt1 = ecnt2 = 0;
}
void addedge( int u, int v, int t)
{
    e[ecnt1] = edge(v,t);
    e[ecnt1].next = head[u];
    head[u] = ecnt1++;


    e1[ecnt2] = edge(u,t);
    e1[ecnt2].next = head1[v];
    head1[v] = ecnt2++;
}

struct point
{
    int v, g;
    point(){};
    point(int _v, int _g)
    {
        v = _v;
        g = _g;
    }
```

```
    bool operator <(const point &a) const
    {
        return dist[v] + g > dist[a.v] + a.g;
    }
};
 priority_queue<point> que;
void dij( int sour)
{


    memset(dist, inf, sizeof(dist));
    memset(vis, 0, sizeof(vis));
    dist[sour] = 0;
    while(!que.empty())  que.pop();
    point now ;
    now = point(sour,0);
    que.push(now);
    point tem;
    while(!que.empty())
    {
        now = que.top();
        que.pop();

        if(vis[now.v])
         continue;
        vis[now.v] = true;


        for( int i = head1[now.v]; i != -1; i =
e1[i].next)
        {
            int t = e1[i].v;
            int cost = e1[i].t;
            if(dist[t] > dist[now.v] + cost)
             {
                 dist[t] = dist[now.v] + cost;
```

```
                que.push(point(t,0));
            }
        }
    }
}


int a_start( int k)
{
    while(!que.empty()) que.pop();
    point now, tem;
    now = point(sta, 0);
    que.push(now);


    while(!que.empty())
    {
        now = que.top();
        que.pop();
        if(now.v == end)
         {
            if(k > 1)
              k--;
            else
             return now.g;
         }
        for( int i = head[now.v]; i != -1; i = e[i].next)
        {
            que.push(point(e[i].v,e[i].t + now.g));
        }
    }
    return -1;
}
```

```
int main()
{
    int n, m, k;
    while(scanf("%d %d",&n, &m) != eof)
    {
        init();
        int u, v, t;
        for( int i = 0; i < m; i++ )
        {
            scanf("%d %d %d",&u, &v, &t);
            addedge(u, v, t);
        }
        scanf("%d %d %d",&sta, &end, &k);
        dij(end);
        if(!dist[sta] == inf)
         {
             printf("-1\n");
             continue;
         }
        if(sta == end)
         k++;
        printf("%d\n",a_start(k));
    }
    return 0;
}
```

第二种采用a*＋spfa

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cstdio>
#include <queue>
#define maxn 1005
```

```
#define maxm 500005
#define inf 1000000000
using namespace std;
struct node
{
    int v, w, next;
}edge[maxm], revedge[maxm];
struct a
{
    int f, g, v;
    bool operator <(const a a)const {
        if(a.f == f) return a.g < g;
        return a.f < f;
    }
};
int e, vis[maxn], d[maxn], q[maxm * 5];
int head[maxn], revhead[maxn];
int n, m, s, t, k;
void init()
{
    e = 0;
    memset(head, -1, sizeof(head));
    memset(revhead, -1, sizeof(revhead));
}
void insert(int x, int y, int w)
{
    edge[e].v = y;
    edge[e].w = w;
    edge[e].next = head[x];
    head[x] = e;
    revedge[e].v = x;
    revedge[e].w = w;
    revedge[e].next =revhead[y];
    revhead[y] = e++;
}
void spfa(int src)
```

```
{
    for(int i = 1; i <= n; i++) d[i] = inf;
    memset(vis, 0, sizeof(vis));
    vis[src] = 0;
    int h = 0, t = 1;
    q[0] = src;
    d[src] = 0;
    while(h < t)
    {
        int u = q[h++];
        vis[u] = 0;
        for(int i = revhead[u] ; i != -1; i =
revedge[i].next)
        {
            int v = revedge[i].v;
            int w = revedge[i].w;
            if(d[v] > d[u] + w)
            {
                d[v] = d[u] + w;
                if(!vis[v])
                {
                    q[t++] = v;
                    vis[v] = 1;
                }
            }
        }
    }
}
int astar(int src, int des)
{
    int cnt = 0;
    priority_queue<a>q;
    if(src == des) k++;
    if(d[src] == inf) return -1;
    a t, tt;
    t.v = src, t.g = 0, t.f = t.g + d[src];
```

```
    q.push(t);
    while(!q.empty())
    {
        tt = q.top();
        q.pop();
        if(tt.v == des)
        {
            cnt++;
            if(cnt == k) return tt.g;
        }
        for(int i = head[tt.v]; i != -1; i =
edge[i].next)
        {
            t.v = edge[i].v;
            t.g = tt.g + edge[i].w;
            t.f = t.g + d[t.v];
            q.push(t);
        }
    }
    return -1;
}
int main()
{
    int x, y, w;
    while(scanf("%d%d", &n, &m) != eof)
    {
        init();
        for(int i = 1; i <= m; i++)
        {
            scanf("%d%d%d", &x, &y, &w);
            insert(x, y, w);
        }
        scanf("%d%d%d", &s, &t, &k);
        spfa(t);
        printf("%d\n", astar(s, t));
    }
```

```
    return 0;
}
```

ida*

算法简介

ida*算法是对迭代加深搜索的优化，是dfs与估计函数的完美结合。避免了广度优先搜索算法占用搜索空间太大的缺点，也减少了深度优先搜索算法的盲目性。它通过设置一个搜索过程中的最大深度，当此状态距离目标状态超过此深度时，便不去搜索。这样便进行了剪枝。ida*的初始深度是起始状态到目标状态的估计距离。随着搜索深入调整这个最大深度值，但这个并不像迭代加深搜索每次只能加一，其增量视具体情况而定。ida总的来说省去了a\的判重时间和空间以及存储当前状态的堆，因此空间效率很高。

主要运用场合和思路

ida*算法就是基于迭代加深的a*算法。其最典型的应用就是八数码问题和十五数码问题。由于改成了深度优先的方式，与a*比起来，ida*更实用，其不需要判重，也不需要排序，并且空间需求减少。 ida*算法着重的部分就是：首先将初始状态节点的h值设为maxh，然后进行深度优先搜索，在搜索的过程中忽略所有h大于maxh的节点；如果没有找到最终的解，则加大这个maxh，再重复上述搜索过程，直到找到一个解。在保证h的计算满足a*算法的要求下，这样找到的解一定是最优的。

POJ 2286

题意：一个矩型的棋盘，上面有1、2、3各8个，要求通过8种操作使得最后中间的八个格子数字相同。
解题思路：ida*搜索，首先，每个格子有三种状态，总共有3^24种状态，即使通过1、2、3都只有8个排除一些内存也是不够的，相反，此题给了15s的时限，便可以用时间效率不如bfs，但是空间上完爆bfs的ida*了。

1. 记录八种旋转的改变数组位置，然后在设定的depth范围内dfs。
2. 两个剪枝：a)当前操作是上一次操作的逆操作。b)当前状态最好情况也无法在depth之内完成任务，即使中间8个格子中最多的数字在最好情况下凑成目标态也超过了depth。

```
#include<cstdio>
#include<algorithm>
#include<cstring>
using namespace std;
int rot[8][7]=
{
    0,2,6,11,15,20,22,//A
    1,3,8,12,17,21,23,//B
    10,9,8,7,6,5,4,//C
    19,18,17,16,15,14,13,//D
    23,21,17,12,8,3,1,//E
    22,20,15,11,6,2,0,//F
    13,14,15,16,17,18,19,//G
    4,5,6,7,8,9,10//H
};
int res[]={5,4,7,6,1,0,3,2};
int depth;
bool check(char s[])
{
    char ch=s[6];
    for(int i=0;i<3;i++)
        if(ch!=s[i+6]||ch!=s[15+i])
            return false;
    return ch==s[11]&&ch==s[12];
}
void rotate(int k,char s[])
{
    char ch=s[rot[k][0]];
    for(int i=0;i<6;i++)
        s[rot[k][i]]=s[rot[k][i+1]];
    s[rot[k][6]]=ch;
}
bool IDAstar(int k,char st[],char op[],int la)
{
    int cnt[4];
```

```
    cnt[1]=cnt[2]=cnt[3]=0;
    for(int i=0;i<3;i++)
        cnt[st[i+6]-'0']++,cnt[st[15+i]-'0']++;
    cnt[st[11]-'0']++,cnt[st[12]-'0']++;
    cnt[0]=max(max(cnt[1],cnt[2]),cnt[3]);
    if(k+8-cnt[0]>=depth)
        return false;
    for(int i=0;i<8;i++)
    {
        if(la!=-1&&res[i]==la)
            continue;
        op[k]='A'+i;
        rotate(i,st);
        if(check(st))
        {
            op[k+1]='\0';
            return true;
        }
        else if(IDAstar(k+1,st,op,i))
            return true;
        rotate(res[i],st);
    }
    return false;
}
int main()
{
    char ch;
    while(scanf(" %c",&ch),ch!='0')
    {
        char st[25];
        st[0]=ch;
        for(int i=1;i<24;i++)
            scanf(" %c",&st[i]);
        depth=1;
        if(check(st))
        {
```

```
            printf("No moves needed\n%c\n",st[6]);
        }
        else
        {
            char op[200];
            op[0]='\0';
            while(!IDAstar(0,st,op,-1))
                depth++;
            printf("%s\n%c\n",op,st[6]);
        }
    }
    return 0;
}
```

POJ 1077

比较典型的八数码问题

```
#include<cstdio>
#include<cstring>
#include<cmath>
#include<iostream>
using namespace std;

void initial(int *maze,int &space)
{
    char grid;
    for(int i=0; i<9; i++)
    {
        cin>>grid;
        if(grid!='x')maze[i]=grid-'1'+1;
        else
        {
            space=i;
            maze[i]=9;
```

```
        }
    }
}

bool isResolve(int *maze,int space)
{
    int s=abs(double(2-space/3))+abs(double(2-space%3));
    for(int i=0; i<9; i++)
    {
        for(int j=0; j<i; j++)
        {
            if(maze[i]>maze[j])s++;
        }
    }
    if(s&1)return false;
    else return true;
}
int h(int *maze)
{
    int h=0;
    for(int i=0; i<9; i++)
    {
        if(maze[i]!=9) h+=abs(double((maze[i]-1)/3-
i/3))+abs(double( (maze[i]-1)%3-i%3));
    }
    return h;
}

int next[9][4]=
{
    {-1,3,1,-1},
    {0,4,2,-1},
    {1,5,-1,-1},
    {-1,6,4,0},
    {3,7,5,1},
    {4,8,-1,2},
```

```
    {-1,-1,7,3},
    {6,-1,8,4},
    {7,-1,-1,5}
};

bool isAns(int *maze)
{
    for(int i=0; i<8; i++)if(maze[i+1]-maze[i]!=1)return
false;
    return true;
}
int pathLimit;
int path[362890],pathLen;

bool IDAStar(int *maze,int len,int space)
{
    if(len==pathLimit)
    {
        if(isAns(maze))
        {
            pathLen=len;
            return true;
        }
        return false;
    }
    for(int i=0; i<4; i++)
    {
        if(next[space][i]!=-1)
        {
            if(len>0&&abs(double(i-path[len-
1]))==2)continue;//!!不考虑相反的方向
            swap(maze[space],maze[next[space][i]]);

            path[len]=i;

if(h(maze)+len<=pathLimit&&IDAStar(maze,len+1,next[space]
```

```
[i]))
                return true;
            swap(maze[space],maze[next[space][i]]);

        }
    }
    return false;
}
char dir[4]= {'l','d','r','u'};
void output()
{
    for(int i=0; i<pathLen; i++)
    {
        switch(path[i])
        {
        case 0:
            cout<<'l';
            break;
        case 1:
            cout<<'d';
            break;
        case 2:
            cout<<'r';
            break;
        case 3:
            cout<<'u';
        }
    }
}
int main()
{
    int maze[9],space;
    initial(maze,space);

    pathLimit=h(maze);
    if(isResolve(maze,space))
```

```
    {
        while(!IDAStar(maze,0,space))pathLimit++;
        output();
    }
    else cout<<"unsolvable";
    cout<<endl;
    return 0;
}
```

hdu1043

跟上一道题目很类似，也是八数码问题，此问题还有一种解法就是康拓展开

```
#include<stdio.h>
#include<string.h>
#include<iostream>
#include<queue>
#include<string>
using namespace std;
const int MAXN=1000000;//最多是9!/2
int fac[]={1,1,2,6,24,120,720,5040,40320,362880};//康拖展开判重
//         0!1!2!3! 4! 5!  6!  7!   8!    9!
bool vis[MAXN];//标记
string path[MAXN];//记录路径
int cantor(int s[])//康拖展开求该序列的hash值
{
    int sum=0;
    for(int i=0;i<9;i++)
    {
        int num=0;
        for(int j=i+1;j<9;j++)
          if(s[j]<s[i])num++;
        sum+=(num*fac[9-i-1]);
    }
    return sum+1;
}
```

```
struct Node
{
    int s[9];
    int loc;//“0”的位置
    int status;//康拖展开的hash值
    string path;//路径
};
int move[4][2]={{-1,0},{1,0},{0,-1},{0,1}};//u,d,l,r
char indexs[5]="durl";//和上面的要相反，因为是反向搜索
int aim=46234;//123456780对应的康拖展开的hash值
void bfs()
{
    memset(vis,false,sizeof(vis));
    Node cur,next;
    for(int i=0;i<8;i++)cur.s[i]=i+1;
    cur.s[8]=0;
    cur.loc=8;
    cur.status=aim;
    cur.path="";
    queue<Node>q;
    q.push(cur);
    path[aim]="";
    while(!q.empty())
    {
        cur=q.front();
        q.pop();
        int x=cur.loc/3;
        int y=cur.loc%3;
        for(int i=0;i<4;i++)
        {
            int tx=x+move[i][0];
            int ty=y+move[i][1];
            if(tx<0||tx>2||ty<0||ty>2)continue;
            next=cur;
            next.loc=tx*3+ty;
            next.s[cur.loc]=next.s[next.loc];
```

```
            next.s[next.loc]=0;
            next.status=cantor(next.s);
            if(!vis[next.status])
            {
                vis[next.status]=true;
                next.path=indexs[i]+next.path;
                q.push(next);
                path[next.status]=next.path;
            }
        }
    }


}
int main()
{
    char ch;
    Node cur;
    bfs();
    while(cin>>ch)
    {
        if(ch=='x') {cur.s[0]=0;cur.loc=0;}
        else cur.s[0]=ch-'0';
        for(int i=1;i<9;i++)
        {
            cin>>ch;
            if(ch=='x')
            {
                cur.s[i]=0;
                cur.loc=i;
            }
            else cur.s[i]=ch-'0';
        }
        cur.status=cantor(cur.s);
        if(vis[cur.status])
        {
```

```
            cout<<path[cur.status]<<endl;
        }
        else cout<<"unsolvable"<<endl;
    }
    return 0;
}
```

搜索优化

算法简介

搜索是一种基本的算法，在建立一个搜索算法时，首先要考虑的问题其一是建立一个合适的算法模型，其二是选择合适的数据结构。然而，搜索方法的时间复杂度大多是指数级别的，简单而不加优化的搜索，其时间效率往往低的不能忍受。本章节所讨论的内容是在建立了合适的算法模型后，对程序进行了基本的优化。一个基本的优化就是剪枝。
搜索的过程可以看做是一个从树根出发，遍历一棵倒置的树，而所谓剪枝就是通过某种判断，避免一些不要的过程。应用剪枝优化的核心问题是设计剪枝判断方法，即确定哪些枝条应当舍弃，哪些枝条应当保留。并且剪枝算法应当遵守正确性，准确性，高效性。
怎样剪枝其具体判断视情况而定。应用比较灵活。下面有几种比较典型的方法。

α-β剪枝（极大极小搜索原理，常用于博弈）

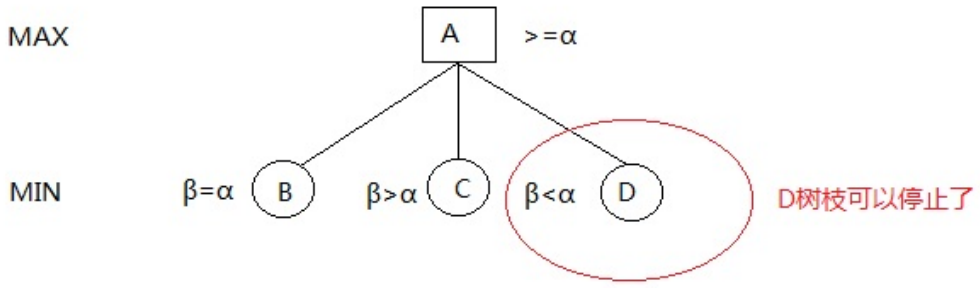
为了提高搜索的效率，通过对值的上下限进行评估，从而减少需要进行评估节点的范围。

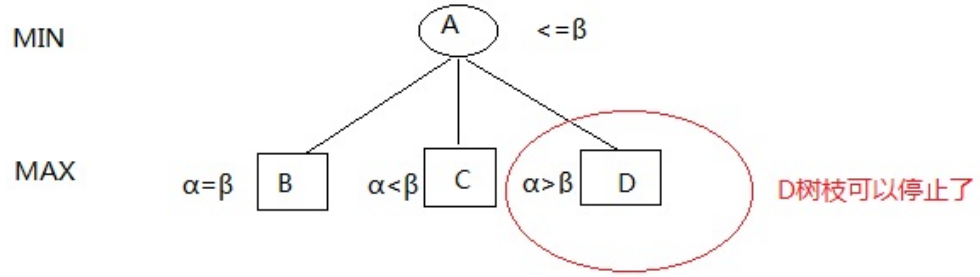
主要概念：

MAX节点的评估下限值α： 作为MAX节点，假定它的MIN节点有N个，那么当它的第一个MIN节点的评估值为α时，则对于其它节点，如果有高于α的节点，就取那最高的节点值作为MAX节点的值；否则，该点的评估值为α。
MIN节点的评估上限值β： 作为MIN节点，同样假定它的MAX节点有N个，那么当它的第一个MAX节点的评估值为β时，则对于其他节点，如果有低于β的节点，就取最低的节点值作为MIN节点的值；否则，该点的评估值为β 。

主要思想：

可以分为两个步骤，分别为α剪枝和β剪枝。
如图：





参考：

- 《对弈程序基本技术》专题 最小-最大搜索：http://www.xqbase.com/computer/search_minimax.htm
- 《对弈程序基本技术》专题 Alpha-Beta搜索：http://www.xqbase.com/computer/search_alphabeta.htm
- Wikipedia MinMax ：http://en.wikipedia.org/wiki/Minimax
- Wiki Alpha–beta pruning：http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
- Minmax Explained：http://ai-depot.com/articles/minimax-explained/1/
- 讲解极小极大 (Minimax Explained) [译]：http://www.starming.com/index.php?action=plugin&v=wave&tpl=union&ac=viewgrouppost&gid=34694&tid=15725
- 最小最大原理与搜索方法：http://blog.pfan.cn/rickone/16930.html

主要运用场合和思路

极大极小搜索策略一般都是使用在一些博弈类的游戏之中， 这样策略本质上使用的是深度搜索策略，所以一般可以使用递归的方法来实现。在搜索过程中，对本方有利的搜索点上应该取极大值，而对本方不利的搜索点上应该取极小值。 极小值和极大值都是相对而言的。 在搜索过程中需要合理的控制搜索深度，搜索的深度越深，效率越低。

POJ 1568

问题：给出一个4x4 tic-tac-toe 的棋局的局面，问先手 "x" 是不是能找在接下来的一步中找到一个必胜局面，如果有，输出第一个落子位置（按顺序）。

```
#include <iostream>
using namespace std;
#define inf 100000000
int state[5][5],chess,xi,xj;
char ch;
int minfind(int,int,int);
int maxfind(int,int,int);
bool over(int x,int y) {
     bool flag = false;
     int row[5],col[5];
     memset(row,0,sizeof(row));
     memset(col,0,sizeof(col));
     for (int i=0;i<4;i++)
         for (int j=0;j<4;j++) {
             if (state[i][j]=='x') {
                row[i]++;
                col[j]++;
             }
             if (state[i][j]=='o') {
                row[i]--;
                col[j]--;
             }
         }
     if (row[x]==-4 || row[x]==4 || col[y]==-4 ||
col[y]==4)
```

```
        flag = true;
    int tot1 = 0, tot2 = 0;
    for (int i=0;i<4;i++) {
        if (state[i][i]=='x') tot1++;
        if (state[i][3-i]=='x') tot2++;
        if (state[i][i]=='o') tot1--;
        if (state[i][3-i]=='o') tot2--;
    }
    if ((tot1==4 || tot1==-4) && x==y)        flag =
true;
    if ((tot2==4 || tot2==-4) && x==3-y)      flag =
true;
    return flag;
}
int maxfind(int x,int y,int mini) {
    int tmp, maxi = -inf;
    if (over(x,y)) return maxi;
    if (chess==16) return 0;
    for (int i=0;i<4;i++)
        for (int j=0;j<4;j++)
            if (state[i][j]=='.') {
                    state[i][j]='x';
                    chess++;
                    tmp = minfind(i,j,maxi);
                    chess--;
                    state[i][j]='.';
                    maxi = max(maxi, tmp);
                    if (maxi>=mini) return maxi;
            }
    return maxi;
}
int minfind(int x,int y,int maxi) {
    int tmp, mini = inf;
    if (over(x,y)) return mini;
    if (chess==16) return 0;
    for (int i=0;i<4;i++)
```

```
        for (int j=0;j<4;j++)
            if (state[i][j]=='.') {
                        state[i][j]='o';
                        chess++;
                        tmp = maxfind(i,j,mini);
                        chess--;
                        state[i][j]='.';
                        mini = min(mini, tmp);
                        if (mini<=maxi) return mini;
            }
    return mini;
}
bool tryit() {
     int tmp, maxi = -inf;
     for (int i=0;i<4;i++)
         for (int j=0;j<4;j++)
             if (state[i][j]=='.') {
                state[i][j] = 'x';
                chess++;
                tmp = minfind(i,j,maxi);
                chess--;
                state[i][j] = '.';
                if (tmp>=maxi) {
                        maxi = tmp;
                        xi = i; xj = j;
                }
                if (maxi==inf) return true;
              }
     return false;
}
int main() {
    while (scanf("%c",&ch)) {
        if (ch=='$') break;
        scanf("%c",&ch);

        chess = -4;
```

```
        for (int i=0;i<4;i++)
            for (int j=0;j<5;j++) {
                scanf("%c",&state[i][j]);
                chess += state[i][j]!='.';
            }
        if (chess<=4) {  //强力剪枝
                    printf("#####\n");
                    continue;
        }
        if (tryit()) printf("(%d,%d)\n",xi,xj);
        else printf("#####\n");
    }
    return 0;
}
```

位运算剪枝

HDU 2553 N皇后问题

题意：n*n棋盘要放n个皇后，要求意2个皇后不允许处在同一排，同一列，也不允许处在与棋盘边框成45角的斜线上。最多几种方案

剪枝：

1. 一个数和它的负数取与，得到的是最右边的1,负数在计算机里面的表示是原码取反加1.
2. 按照行来进行枚举，同时将两个对角线和列的状态压缩表示。详细解释的话，我们可以这样想，对于任何一行，都有n个位置，那么这n个位置相应的列是否有皇后.
3. 对于对角线，要单拿出来说一下。由于(i, j) (i+1, j-1)只能有一个值，而第i行的第一个位置，在主对角线的情况下对之后的行是没有影响的；同理，次对角线的情况，每一行的最后一个元素对于后面的行的是没有影响的，并且是第i行的第j个位置和i+1行的j+1的位置是对应。因此，在向下一层遍历的时候，要把主对角线向左进行移位，次对角线向右进行移位。这样，使得相应的位置可以对上。新增的置为0。

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

int n, ans;
int limit;

void dfs( int h, int r, int l ) {
    if ( h == limit ) {  //说明n列都站满了
        ans++;
        return;
    }
    int pos = limit & (~(h|r|l)), p; //pos的二进制位0的，并且这个limit不能省，limit保证了pos高于n的位为0
    while ( pos ) {
        p = pos & (-pos); //这个运算，即原码和补码取与的运算，可以得出最右边的1；
        pos -= p;
        dfs( h+p, (r+p)<<1, (l+p)>>1 );
    }
}
int main()
{
    while ( scanf("%d", &n) != EOF && n ) {
        ans = 0;
        limit = ( 1<<n ) - 1; //limit的二进制是n个1；
        //cout << limit << endl;
        dfs( 0, 0, 0 );
        printf("%d\n", ans);
    }
}
```

奇偶剪枝

首先举个例子，有如下4*4的迷宫，'.'为可走路段，'X'为障碍不可通过
S...
....
....
...D
从S到D的最短距离为两点横坐标差的绝对值+两点纵坐标差的绝对值 = abs(Sx - Dx) + abs(Sy - Dy) = 6，这个应该是显而易见的。
遇到有障碍的时候呢？ S.XX
X.XX
...X
...D
你会发现不管你怎么绕路，最后从S到达D的距离都是最短距离+一个偶数，这个是可以证明的
而我们知道：
奇数 + 偶数 = 奇数
偶数 + 偶数 = 偶数
因此不管有多少障碍，不管绕多少路，只要能到达目的地，走过的距离必然是跟最短距离的奇偶性是一致的。
所以如果我们知道从S到D的最短距离为奇数，那么当且仅当给定的步数T为奇数时，才有可能走到。如果给定的T的奇偶性与最短距离的奇偶性不一致，那么我们就可以直接判定这条路线永远不可达了。
这里还有个小技巧，我们可以使用按位与运算来简化奇偶性的判断。我们知道1的二进制是1，而奇数的二进制最后一位一定是1，而偶数的二进制最后一位一定是0。所以如果数字&1的结果为1，那么数字为奇数，反之为偶数。

ZOJ Problem Set - 2110 Tempter of the Bone

题意：有一只狗要吃骨头，结果进入了一个迷宫陷阱，迷宫里每走过一个地板费时一秒，该地板 就会在下一秒塌陷，所以你不能在该地板上逗留。迷宫里面有一个门，只能在特定的某一秒才能打开，让狗逃出去。现在题目告诉你迷宫的大小和门打开的时间，问你狗可不可以逃出去，可以就输出YES,否则NO。

思路：搜索+剪枝

1. 如果当前时间即步数(step) >= T 而且还没有找到D点，则剪掉。

2. 设当前位置(x, y)到D点(dx, dy)的最短距离为s，到达当前位置(x, y)已经花费时间(步数)step，那么，如果题目要求的时间T - step < s，则剪掉。
3. 对于当前位置(x, y)，如果，(T-step-s)是奇数，则剪掉(奇偶剪枝)。
4. 如果地图中，可走的点的数目(xnum) < 要求的时间T，则剪掉(路径剪枝)。

```
#include <iostream>
#include <cmath>
using namespace std;
int N, M, T, sx, sy, ex, ey;
char map[6][6];
const int dir[4][2] = { { 0, 1 }, { 1, 0 }, { 0, -1 }, {
-1, 0 } };
bool solved =  false, arrd[6][6];
int Distance ( int x, int y )
{
    return abs ( (double)x - ex ) + abs ( (double)y - ey
); // 当前点(x,y)到终点(ex,ey)的最短距离
}
void DFS ( int x, int y, int step )
{
    if ( solved ) return;
    if ( map[x][y] == 'D' && step == T ) {
        solved = true;
        return;
    }
    if ( step >= T ) return;                    // 当前时
间即步数(step) >= T 而且还没有找到D点
    int dis = T - step - Distance ( x, y );
    if ( dis < 0 || dis % 2 ) return;           // 剩余步
数小于最短距离或者满足奇偶剪枝条件
    for ( int i = 0; i < 4; i += 1 ) {
        int tx = x + dir[i][0];
        int ty = y + dir[i][1];
        int tstep = step + 1;
        if ( tx >= 0 && tx < N && ty >= 0 && ty < M &&
map[tx][ty] != 'X' && !arrd[tx][ty]) {
```

```
            arrd[tx][ty] = true;
            DFS ( tx, ty, tstep );
            arrd[tx][ty] = false;
        }
    }
}
int main ( int argc, char *argv[] )
{
    while ( cin >> N >> M >> T, N+M+T ) {
        solved = false;
        int xnum = 0;                           // 记
录'X'的数量
        for ( int i = 0; i < N; i += 1 ) {
            cin.get();
            for ( int j = 0; j < M; j += 1 ) {
                cin >> map[i][j];
                arrd[i][j] = false;
                if ( map[i][j] == 'S' ) {
                    sx = i;
                    sy = j;
                    arrd[i][j] = true;
                }
                else if ( map[i][j] == 'D' ) {
                    ex = i;
                    ey = j;
                }
                else if ( map[i][j] == 'X' ) {
                    xnum++;
                }
            }
        }
        if ( N * M - xnum > T ) { // 可通行的点必须大于要求的
步数，路径剪枝。
            DFS ( sx, sy, 0 );
        }
        if ( solved )
```

```
            cout << "YES" << endl;
        else
            cout << "NO" << endl;
    }
    return 0;
}
```

记忆化深度优先搜索

算法介绍

算法上依然是搜索的流程，但是搜索到的一些解用动态规划的那种思想和模式作一些保存。一般说来，动态规划总要遍历所有的状态，而搜索可以排除一些无效状态。更重要的是搜索还可以剪枝，可能剪去大量不必要的状态，因此在空间开销上往往比动态规划要低很多。
记忆化算法在求解的时候还是按着自顶向下的顺序，但是每求解一个状态，就将它的解保存下来，以后再次遇到这个状态的时候，就不必重新求解了。这种方法综合了搜索和动态规划两方面的优点，因而还是很有实用价值的。根据记忆化搜索的思想，它是解决重复计算，而不是重复生成，也就是说，这些搜索必须是在搜索扩展路径的过程中分步计算的题目，也就是“搜索答案与路径相关”的题目，而不能是搜索一个路径之后才能进行计算的题目，必须要分步计算，并且搜索过程中，一个搜索结果必须可以建立在同类型问题的结果上，也就是类似于动态规划解决的那种。也就是说，他的问题表达，不是单纯生成一个走步方案，而是生成一个走步方案的代价等，而且每走一步，在搜索树/图中生成一个新状态，都可以精确计算出到此为止的费用，也就是，可以分步计算，这样才可以套用已经得到的答案一个公式简单地说：记忆化搜索=搜索的形式+动态规划的思想。

主要运用场合及思路

记忆化深度优先搜索

程序框架

```
const
  nonsense=-1; {这个值可以随意定义，表示一个状态的最优值未知}
var
…… {全局变量定义}

function work(s:statetype):longint;
var
…… {局部变量定义}
begin
  if opt[s]<>nonsense then
  begin
work:=opt[s];
exit;
end; {如果状态s的最优值已经知道，直接返回这个值}
枚举所有可以推出状态S的状态S1
begin
  temp:=由S1推出的S的最优值 {有关work(s1)的函数} ;
  if (opt[s]=nonsense) or (temp优于opt[s])
    then opt[s]:=temp;
end;
work:=opt[s]; {用动态规划的思想推出状态s的最优值}
end;
```

记忆化深度优先搜索采用了一般的深度优先搜索的递归结构，但是在搜索到一个未知的状态时它总把它记录下来，为以后的搜索节省了大量的时间。可见，记忆化搜索的实质是动态规划，效率也和动态规划接近；形式是搜索，简单易行，也不需要进行什么拓扑排序了。

hdu1078

题意：题意：老鼠偷吃，有n*n的方阵，每个格子里面放着一定数目的粮食，老鼠每次只能水平或竖直最多走k步，每次必须走食物比当前多的格子，问最多吃多少食物。
解题思路：记忆化搜索。

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;

int n,k,dp[105][105],a[105][105];
int to[4][2] = {1,0,-1,0,0,1,0,-1};

int check(int x,int y)
{
    if(x<1 || y<1 || x>n || y>n)
        return 1;
    return 0;
}

int dfs(int x,int y)
{
    int i,j,l,ans = 0;
    if(!dp[x][y])
    {
        for(i = 1; i<=k; i++)
        {
            for(j = 0; j<4; j++)
            {
                int xx = x+to[j][0]*i;
                int yy = y+to[j][1]*i;
                if(check(xx,yy))
                    continue;
                if(a[xx][yy]>a[x][y])
                    ans = max(ans,dfs(xx,yy));
            }
        }
        dp[x][y] = ans+a[x][y];
    }
    return dp[x][y];
```

```
}

int main()
{
    int i,j;
    while(~scanf("%d%d",&n,&k),n>0&&k>0)
    {
        for(i = 1; i<=n; i++)
            for(j = 1; j<=n; j++)
                scanf("%d",&a[i][j]);
        memset(dp,0,sizeof(dp));
        printf("%d\n",dfs(1,1));
    }

    return 0;
}
```

UVa 10118 Free Candies（记忆化搜索经典）

题意：
有4堆糖果，每堆有n（最多40）个，有一个篮子，最多装5个糖果，我们每次只能从某一堆糖果里拿出一个糖果，如果篮子里有两个相同的糖果，那么就可以把这两个（一对）糖果放进自己的口袋里，问最多能拿走多少对糖果。糖果种类最多20种.

思路：

1. 这一题有点逆向思维的味道，dp[a, b, c, d] 表示从每堆中分别拿 a, b, c, d 个时，最多能拿多少个糖果;
2. 注意一点：当拿到 a, b, c, d 时，不能再拿了，此时结果肯定就会固定。利用这一点性质，采用记忆化搜索能有效的减少重复子结构的计算;
3. 题目是只有 0 0 0 0 这一个出发点的，根据这个出发点进行深搜，最终得出结果。
4. 本题可谓是深搜 + 记忆化搜索的经典，状态不是那么明显，子结构也不是那么好抽象，因为转移的末状态并不是固定的，是在不断的搜索中求出来的;

```
#include <iostream>
#include <algorithm>
```

```
#include <cstdlib>
#include <cstring>
#include <cstdio>
using namespace std;

const int MAXN = 41;
int pile[4][MAXN], dp[MAXN][MAXN][MAXN][MAXN];
int n, top[4];

int dfs(int count, bool hash[]) {
    if (dp[top[0]][top[1]][top[2]][top[3]] != -1)
        return dp[top[0]][top[1]][top[2]][top[3]];

    if (count == 5)
        return dp[top[0]][top[1]][top[2]][top[3]] = 0;

    int ans = 0;
    for (int i = 0; i < 4; i++) {
        if (top[i] == n) continue;
        int color = pile[i][top[i]];
        top[i] += 1;
        if (hash[color]) {
            hash[color] = false;
            ans = max(ans, dfs(count-1, hash) + 1);
            hash[color] = true;
        } else {
            hash[color] = true;
            ans = max(ans, dfs(count+1, hash));
            hash[color] = false;
        }
        top[i] -= 1;
    }
    return dp[top[0]][top[1]][top[2]][top[3]] = ans;
}

int main() {
```

```
    while (scanf("%d", &n) && n) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < 4; j++)
                scanf("%d", &pile[j][i]);
        bool hash[25];
        memset(dp, -1, sizeof(dp));
        memset(hash, false, sizeof(hash));
        top[0] = top[1] = top[2] = top[3] = 0;
        printf("%d\n", dfs(0, hash));
    }
    return 0;
}
```

记忆化宽度优先搜索

程序框架

边界条件进入队列
找到一个没有扩展过的且肯定是最优的状态，没有则退出
扩展该状态，并修改扩展到的最优值
返回到第二步

记忆化宽度优先搜索实际上是把一般的动态规划倒过来了。动态规划在计算每一个状态的最优值时，总是寻找可以推出该状态的已知状态，然后从中选择一个最优的决策。而记忆化宽度优先搜索是不断地从已知状态出发，看看能推出哪些其它的状态，并修改其它状态的最优值。

这个算法有两大优点：

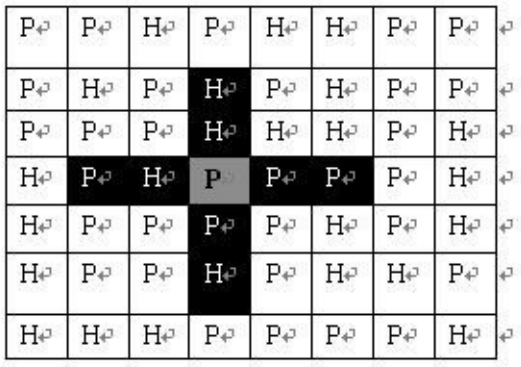
1. 与记忆化深度优先搜索相似，避免了对错综复杂的拓扑关系的分析
2. 化倒推为顺推，更加符合人类的思维过程，而且在顺推比倒推简单时，可以使问题简单化

例子

下面我们通过一个例子，具体问题具体分析

[问题描述]

司令部的将军们打算在N*M*的网格地图上部署他们的炮兵部队。一个*N*M的地图由N行M列组成，地图的每一格可能是山地（用“H” 表示），也可能是平原（用“P”表示），如下图。在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；一支炮兵部队在地图上的攻击范围如图中黑色区域所示：



如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。
现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。
数据限制：N<=100 M<=10

[算法分析]

这一题目若用图论的模型来做，就是一个最大独立集问题，而最大独立集问题确实一个NP问题。并且我们也做过这种题目的原型题目，大部分都必须通过搜索来解决。故我们需要考察这个问题的特殊性。
由数据限制可知 `mmax<<nmax` 。注意这是唯一一个与原型题目不同的地方。可是如何使用呢？我们先考虑一种极端情况，例如m=1的情况。这个时候，我们可以用如下的动态规划方程来解出结果：

$$c[0] = 0$$

$$c[i] = \begin{cases} \max\limits_{1 \le i \le n} \{c[i-3]+1, c[i-1]\} & \text{第 i行第 1个格子为平地} \\ c[i-1] & \text{第 i行第 1个格子为山地} \end{cases}$$

其中c[i]的含义是在前i行各自中可以放置的最多炮兵数，c[n]即为解答。

这种方法是否可以扩展到m列呢？
我们这时用c[i]表示m列时在前i行格子中可以放置的最多炮兵数。然后把每一行可能放置炮兵的方法求出来，然后就可以只考虑行与行之间放置炮兵的关系了。不过这样我们还是无法写出规划方程。观察这个时候情况与m=1式情况的不同：就是对于每以列我们多需要保存各自的c[i-3]和c[i-1]。有鉴于此，我们需要增设一个参数p来满足动态规划的需要。这里p应是一个m位的数组，并且每一位都有三种选择，第k列的三种选择分别代表该列取c[i-1]、c[i-2]、c[i-3]时所对应的最值。

这样，我们有：

$$c[0, p] = 0$$

$$c[i, p] = \max\{ \max_{j \in J} \{c[i-1, p \oplus j] + |j|\}, \max_{q \in p^-} \{c[i, q]\}\}$$

最后的答案就是 $\max_{j \in J} \{c[n, j]\}$ 。

我们虽然写出了状态转移方程式，但是如果用这样的方程式进行动态规划，实现起来是非常烦琐的，尤其涉及到了参数p的倒推，颇为复杂。于是我们又将目光投向了记忆化搜索——化倒推为顺推。把算法进行一些修改：c[0,p]为边界状态；以阵地的行作为阶段进行搜索，从c[I,p]推出所有c[I+1,p]，再从c[I+1,p]推出c[I+2,p]，……，直到c[n,p]。这样，使用记忆化搜索使得问题得到了圆满的解决！

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<cstring>
#include<algorithm>
using namespace std;

const int MAXN=105;

int N,M,sum,ans;
int now[MAXN],a[MAXN],b[MAXN],f[MAXN][MAXN][MAXN];
char s[MAXN],ch;

void Init()
{
     scanf("%d%d\n",&N,&M);
```

```
        for (int i=1;i<=N;i++)
        {
            for (int j=1;j<=M;j++)
            {
                scanf("%c",&ch);
                if (ch=='H') now[i]+=1<<(M-j);
            }
            scanf("%c",&ch);
        }

}

void Change(int x)
{
        memset(s,0,sizeof(s));
        int len=0;
        while (x>0)
        {
              if (x%2==1) s[len++]='1'; else s[len++]='0';
              x/=2;
        }
        while (len<=M) s[len++]='0';
        //strrev(s);
        for (int i=0;i<M/2;i++) swap(s[i],s[M-1-i]);
}

void Solve()
{
        int temp;
        bool flag;
        for (int i=0;i<=(1<<M)-1;i++)
        {
            temp=0;
            flag=1;
            Change(i);
            for (int j=0;j<M;j++)
```

```
            if (s[j]=='1')
            {
                        if (j+1<=M && s[j+1]=='1')
{flag=0;break;}
                        if (j+2<=M && s[j+2]=='1')
{flag=0;break;}
                        temp++;
            }
        if (flag)
        {
                a[++sum]=i;
                b[sum]=temp;
        }
    }

    for (int i=1;i<=sum;i++)
        f[1][i][1]=b[i];
    for (int i=2;i<=N;i++)
        for (int k1=1;k1<=sum;k1++)
            for (int k2=1;k2<=sum;k2++)
                if (((a[k1]&a[k2])==0) &&
((a[k1]&now[i])==0) && ((a[k2]&now[i-1])==0))
                    for (int k3=1;k3<=sum;k3++)
                        if (((a[k1]&a[k3])==0) &&
((a[k2]&a[k3])==0) && ((a[k3]&now[i-2])==0))
                            f[i][k1][k2]=max(f[i][k1]
[k2],f[i-1][k2][k3]+b[k1]);
    for (int i=1;i<=sum;i++)
        for (int j=1;j<=sum;j++)
            if (((a[i]&a[j])==0) && ((a[i]&now[N])==0)
&& ((a[j]&now[N-1])==0))
                ans=max(ans,f[N][i][j]);
    printf("%d\n",ans);
}

int main()
```

```
{
    Init();
    Solve();
    return 0;
}
```

小结一下：首先，在“m<=10”的启发下，找到了动态规划的算法，这是主要的一个步骤；其次，我们选择了记忆化搜索，使得算法的实现变得容易了许多，这也是不可或缺的一步。总而言之，记忆化宽度优先搜索将宽度优先搜索的结构和动态规划的思想有机地结合在一起，为我们的解题提供了一个新的途径。

list

STL中的list就是一双向链表，可高效地进行插入删除元素。

1.list构造函数

```
list<int> L0;        // 空链表

list<int> L1(9);    // 建一个含个默认值是的元素的链表

list<int> L2(5,1); // 建一个含个元素的链表

list<int> L3(L2);  // 建一个L2的copy链表

list<int> L4(L0.begin(), L0.end());//建一个含L0一个区域的元素
```

2.assign()分配值，有两个重载

```
L1.assign(4,3);                                    //
L1(3,3,3,3)

L1.assign(++list1.beging(), list2.end());   // L1(2,3)
```

3.operator= 赋值重载运算符

```
L1 = list1;   // L1(1,2,3)
```

4.front()返回第一个元素的引用

```
int nRet = list1.front()    // nRet = 1
```

5.back()返回最后一元素的引用

```
int nRet = list1.back()     // nRet = 3
```

6.begin()返回第一个元素的指针(iterator)

```
it = list1.begin();    // *it = 1
```

7.end()返回最后一个元素的下一位置的指针(list为空时end()=begin())

```
it = list1.end();
--it;                          // *it = 3
```

8.rbegin()返回链表最后一元素的后向指针(reverse_iterator or const)

```
list<int>::reverse_iterator it = list1.rbegin();  // *it
= 3
```

9.rend()返回链表第一元素的下一位置的后向指针

```
list<int>::reverse_iterator it = list1.rend(); // *(--
riter) = 1
```

10.push_back()增加一元素到链表尾

```
list1.push_back(4)       // list1(1,2,3,4)
```

11.push_front()增加一元素到链表头

```
list1.push_front(4)      // list1(4,1,2,3)
```

12.pop_back()删除链表尾的一个元素

```
list1.pop_back()          // list1(1,2)
```

13.pop_front()删除链表头的一元素

```
list1.pop_front()           // list1(2,3)
```

14 · clear()删除所有元素

```
list1.clear();   // list1空了,list1.size() =0
```

15.erase()删除一个元素或一个区域的元素(两个重载函数)

```
list1.erase(list1.begin());                 // list1(2,3)
list1.erase(++list1.begin(),list1.end()); // list1(1)
```

16.remove()删除链表中匹配值的元素(匹配元素全部删除)

```
//list对象L1(4,3,5,1,4)
L1.remove(4);                // L1(3,5,1);
```

17.remove_if()删除条件满足的元素(遍历一次链表)，参数为自定义的回调函数

```

// 小于2的值删除

bool myFun(const int& value) { return (value < 2); }

list1.remove_if(myFun);    // list1(3)
```

18.empty()判断是否链表为空

```
bool bRet = L1.empty(); //若L1为空，bRet = true，否则bRet = false。
```

19.max_size()返回链表最大可能长度

```
list&lt;int&gt;::size_type nMax = list1.max_size();//
nMax = 1073741823
```

20 · size()返回链表中元素个数

```
list&lt;int&gt;::size_type nRet = list1.size();      //
nRet = 3
```

21.resize()重新定义链表长度(两重载函数)

```
list1.resize(5)    // list1 (1,2,3,0,0)用默认值填补

list1.resize(5,4)    // list1 (1,2,3,4,4)用指定值填补
```

22.reverse()反转链表:

```
list1.reverse();     // list1(3,2,1)
```

23.sort()对链表排序，默认升序(可自定义回调函数)
list对象L1(4,3,5,1,4)

```
L1.sort();                    // L1(1,3,4,4,5)

L1.sort(greater<int>()); // L1(5,4,4,3,1)
```

24.merge()合并两个有序链表并使之有序

```
// 升序

list1.merge(list2);           // list1(1,2,3,4,5,6) list2现为空

// 降序

L1(3,2,1), L2(6,5,4)

L1.merge(L2, greater<int>()); // list1(6,5,4,3,2,1) list2现为空
```

25.splice()对两个链表进行结合(三个重载函数) 结合后第二个链表清空

```
list1.splice(++list1.begin(),list2);

// list1(1,4,5,6,2,3) list2为空

 list1.splice(++list1.begin(),list2,list2.begin());

// list1(1,4,2,3); list2(5,6)

list1.splice(++list1.begin(),list2,++list2.begin(),list2.end());

//list1(1,5,6,2,3); list2(4)
```

26.insert()在指定位置插入一个或多个元素(三个重载函数)

```
list1.insert(++list1.begin(),9);  // list1(1,9,2,3)

list1.insert(list1.begin(),2,9);  // list1(9,9,1,2,3);

list1.insert(list1.begin(),list2.begin(),--
list2.end());//list1(4,5,1,2,3);
```

27.swap()交换两个链表(两个重载)

```
list1.swap(list2);   // list1（4，5，6） list2（1，2，3）
```

28.unique()删除相邻重复元素

```
L1(1,1,4,3,5,1)

L1.unique();          // L1(1,4,3,5,1)
```

示例：

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

list<int> g_list1;
list<int> g_list2;

////////////////////////////////////////////////////////////
////////////////

// 初始化全局链表
void InitList()
{
    //  push_back()增加一元素到链表尾
```

```
    g_list1.push_back(1);
    g_list1.push_back(2);
    g_list1.push_back(3);

    // push_front()增加一元素到链表头
    g_list2.push_front(6);
    g_list2.push_front(5);
    g_list2.push_front(4);
}

// 输出一个链表
void ShowList(list<int>& listTemp)
{
    //  size()返回链表中元素个数
    cout << listTemp.size() << endl;

    for (list<int>::iterator it = listTemp.begin(); it !=
listTemp.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << endl;
}

////////////////////////////////////////////////////////////
////////////////

// 构造函数，空链表
void constructor_test0()
{
    list<int> listTemp;
    cout << listTemp.size() << endl;
}

// 构造函数，建一个含三个默认值是0的元素的链表
void constructor_test1()
```

```
{
    list<int> listTemp(3);
    ShowList(listTemp);
}

// 构造函数，建一个含五个元素的链表，值都是1
void constructor_test2()
{
    list<int> listTemp(5, 1);
    ShowList(listTemp);
}

// 构造函数，建一个g_list1的copy链表
void constructor_test3()
{
    list<int> listTemp(g_list1);
    ShowList(listTemp);
}

// 构造函数，listTemp含g_list1一个区域的元素[_First, _Last)
void constructor_test4()
{
    list<int> listTemp(g_list1.begin(), g_list1.end());
    ShowList(listTemp);
}

// assign()分配值，有两个重载
// template <class InputIterator>
// void assign ( InputIterator first, InputIterator last
);
// void assign ( size_type n, const T& u );
void assign_test()
{
    list<int> listTemp(5, 1);
    ShowList(listTemp);
```

```
    listTemp.assign(4, 3);
    ShowList(listTemp);

    listTemp.assign(++g_list1.begin(), g_list1.end());
    ShowList(listTemp);
}

// operator=
void operator_equality_test()
{
    g_list1 = g_list2;
    ShowList(g_list1);
    ShowList(g_list2);
}

// front()返回第一个元素的引用
void front_test7()
{
    cout << g_list1.front() << endl;
}

// back()返回最后一元素的引用
void back_test()
{
    cout << g_list1.back() << endl;
}

// begin()返回第一个元素的指针(iterator)
void begin_test()
{
    list<int>::iterator it1 = g_list1.begin();
    cout << *++it1 << endl;

    list<int>::const_iterator it2 = g_list1.begin();
    it2++;
    // (*it2)++;    //     *it2 为const 不用修改
```

```
    cout << *it2 << endl;

}

// end()返回 [最后一个元素的下一位置的指针] (list为空时end()=
begin())
void end_test()
{
    list<int>::iterator it  = g_list1.end();    // 注意
是：最后一个元素的下一位置的指针
    --it;
    cout << *it << endl;
}

//  rbegin()返回链表最后一元素的后向指针
void rbegin_test()
{
    list<int>::reverse_iterator it = g_list1.rbegin();
    for (; it != g_list1.rend(); ++it)
    {
        cout << *it << ' ';
    }
    cout << endl;
}

//  rend()返回链表第一元素的下一位置的后向指针
void rend_test()
{
    list<int>::reverse_iterator it = g_list1.rend();
    --it;
    cout << *it << endl;
}

// push_back()增加一元素到链表尾
void push_back_test()
{
```

```
    ShowList(g_list1);
    g_list1.push_back(4);
    ShowList(g_list1);
}

// push_front()增加一元素到链表头
void push_front_test()
{
    ShowList(g_list1);
    g_list1.push_front(4);
    ShowList(g_list1);
}

// pop_back()删除链表尾的一个元素
void pop_back_test()
{
    ShowList(g_list1);
    cout << endl;

    g_list1.pop_back();
    ShowList(g_list1);

}

// pop_front()删除链表头的一元素
void pop_front_test()
{
    ShowList(g_list1);
    cout << endl;

    g_list1.pop_front();
    ShowList(g_list1);
}

// clear()删除所有元素
void clear_test()
```

```
{
    ShowList(g_list1);
    g_list1.clear();
    ShowList(g_list1);
}

// erase()删除一个元素或一个区域的元素(两个重载函数)
void erase_test()
{
    ShowList(g_list1);
    g_list1.erase(g_list1.begin());
    ShowList(g_list1);

    cout << endl;

    ShowList(g_list2);
    g_list2.erase(++g_list2.begin(), g_list2.end());
    ShowList(g_list2);
}

// remove()删除链表中匹配值的元素(匹配元素全部删除)
void remove_test()
{
    ShowList(g_list1);
    g_list1.push_back(1);
    ShowList(g_list1);

    g_list1.remove(1);
    ShowList(g_list1);
}

bool myFun(const int& value) { return (value < 2); }
// remove_if()删除条件满足的元素(会遍历一次链表)
void remove_if_test()
{
    ShowList(g_list1);
```

```
    g_list1.remove_if(myFun);
    ShowList(g_list1);
}


// empty()判断是否链表为空
void empty_test()
{
    list<int> listTemp;
    if (listTemp.empty())
        cout << "listTemp为空" << endl;
    else
        cout << "listTemp不为空" << endl;
}


//  max_size()返回链表最大可能长度:1073741823
void max_size_test()
{
    list<int>::size_type nMax = g_list1.max_size();
    cout << nMax << endl;
}


// resize()重新定义链表长度(两重载函数):
void resize_test()
{
    ShowList(g_list1);
    g_list1.resize(9);          // 用默认值填补
    ShowList(g_list1);
    cout << endl;

    ShowList(g_list2);
    g_list2.resize(9, 51);      // 用指定值填补
    ShowList(g_list2);
}
```

```
// reverse()反转链表
void reverse_test()
{
    ShowList(g_list1);
    g_list1.reverse();
    ShowList(g_list1);
}


// sort()对链表排序，默认升序(两个重载函数)
void sort_test()
{
    list<int> listTemp;
    listTemp.push_back(9);
    listTemp.push_back(3);
    listTemp.push_back(5);
    listTemp.push_back(1);
    listTemp.push_back(4);
    listTemp.push_back(3);

    ShowList(listTemp);
    listTemp.sort();
    ShowList(listTemp);

    listTemp.sort(greater<int>());
    ShowList(listTemp);
}

// merge()合并两个升序序链表并使之成为另一个升序.
void merge_test1()
{
    list<int> listTemp2;
    listTemp2.push_back(3);
    listTemp2.push_back(4);
```

```
    list<int> listTemp3;
    listTemp3.push_back(9);
    listTemp3.push_back(10);

    ShowList(listTemp2);
    cout << endl;
    ShowList(listTemp3);
    cout << endl;

    listTemp2.merge(listTemp3);
    ShowList(listTemp2);
}


bool myCmp (int first, int second)
{ return ( int(first)>int(second) ); }

// merge()合并两个降序链表并使之成为另一个降序.
void merge_test2()
{
    list<int> listTemp2;
    listTemp2.push_back(4);
    listTemp2.push_back(3);

    list<int> listTemp3;
    listTemp3.push_back(10);
    listTemp3.push_back(9);

    ShowList(listTemp2);
    cout << endl;
    ShowList(listTemp3);
    cout << endl;

    // listTemp2.merge(listTemp3, greater<int>());    //
第二个参数可以是自己定义的函数如下
    listTemp2.merge(listTemp3, myCmp);
```

```
    ShowList(listTemp2);
}

// splice()对两个链表进行结合(三个重载函数),结合后第二个链表清空
// void splice ( iterator position, list<T,Allocator>& x
);
// void splice ( iterator position, list<T,Allocator>& x,
iterator i );
// void splice ( iterator position, list<T,Allocator>& x,
iterator first, iterator last );
void splice_test()
{
    list<int> listTemp1(g_list1);
    list<int> listTemp2(g_list2);

    ShowList(listTemp1);
    ShowList(listTemp2);
    cout << endl;

    //
    listTemp1.splice(++listTemp1.begin(), listTemp2);
    ShowList(listTemp1);
    ShowList(listTemp2);

    //
    listTemp1.assign(g_list1.begin(), g_list1.end());
    listTemp2.assign(g_list2.begin(), g_list2.end());
    listTemp1.splice(++listTemp1.begin(), listTemp2,
++listTemp2.begin());
    ShowList(listTemp1);
    ShowList(listTemp2);

    //
    listTemp1.assign(g_list1.begin(), g_list1.end());
    listTemp2.assign(g_list2.begin(), g_list2.end());
    listTemp1.splice(++listTemp1.begin(), listTemp2,
```

```
++listTemp2.begin(), listTemp2.end());
    ShowList(listTemp1);
    ShowList(listTemp2);

}

//  insert()在指定位置插入一个或多个元素(三个重载函数)
// iterator insert ( iterator position, const T& x );
// void insert ( iterator position, size_type n, const T&
x );
// template <class InputIterator>
// void insert ( iterator position, InputIterator first,
InputIterator last );
void insert_test()
{
    list<int> listTemp1(g_list1);
    ShowList(listTemp1);
    listTemp1.insert(listTemp1.begin(), 51);
    ShowList(listTemp1);
    cout << endl;

    list<int> listTemp2(g_list1);
    ShowList(listTemp2);
    listTemp2.insert(listTemp2.begin(), 9, 51);
    ShowList(listTemp2);
    cout << endl;

    list<int> listTemp3(g_list1);
    ShowList(listTemp3);
    listTemp3.insert(listTemp3.begin(), g_list2.begin(),
g_list2.end());
    ShowList(listTemp3);

}

// swap()交换两个链表(两个重载)
```

```
void swap_test()
{
    ShowList(g_list1);
    ShowList(g_list2);
    cout << endl;

    g_list1.swap(g_list2);
    ShowList(g_list1);
    ShowList(g_list2);
}

bool same_integral_part (double first, double second)
{ return ( int(first)==int(second) ); }

// unique()删除相邻重复元素
void unique_test()
{
    list<int> listTemp;
    listTemp.push_back(1);
    listTemp.push_back(1);
    listTemp.push_back(4);
    listTemp.push_back(3);
    listTemp.push_back(5);
    listTemp.push_back(1);
    list<int> listTemp2(listTemp);

    ShowList(listTemp);
    listTemp.unique();    // 不会删除不相邻的相同元素
    ShowList(listTemp);
    cout << endl;

    listTemp.sort();
    ShowList(listTemp);
    listTemp.unique();
    ShowList(listTemp);
    cout << endl;
```

```
    listTemp2.sort();
    ShowList(listTemp2);
    listTemp2.unique(same_integral_part);
    ShowList(listTemp2);

}
```

set

set关联式容器。set作为一个容器也是用来存储同一数据类型的数据类型，并且能从一个数据集合中取出数据，在set中每个元素的值都唯一，而且系统能根据元素的值自动进行排序。应该注意的是set中数元素的值不能直接被改变。C++ STL中标准关联容器set, multiset, map, multimap内部采用的就是一种非常高效的平衡检索二叉树：红黑树，也成为RB树(Red-Black Tree)。RB树的统计性能要好于一般平衡二叉树，所以被STL选择作为了关联容器的内部结构。

常用操作：

begin(),返回set容器的第一个元素

end(),返回set容器的最后一个元素

clear(),删除set容器中的所有的元素

empty(),判断set容器是否为空

max_size(),返回set容器可能包含的元素最大个数

size(),返回当前set容器中的元素个数

rbegin(),返回的值和end()相同

rend(),返回的值和rbegin()相同

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    set<int> s;
    s.insert(1);
    s.insert(2);
    s.insert(3);
    s.insert(1);
    cout<<"set 的 size 值为 ："<<s.size()<<endl;
    cout<<"set 的 maxsize的值为 ："<<s.max_size()<<endl;
    cout<<"set 中的第一个元素是 ："<<*s.begin()<<endl;
    cout<<"set 中的最后一个元素是:"<<*s.end()<<endl;
    s.clear();
    if(s.empty())
    {
        cout<<"set 为空 ！！！"<<endl;
    }
    cout<<"set 的 size 值为 ："<<s.size()<<endl;
    cout<<"set 的 maxsize的值为 ："<<s.max_size()<<endl;
    return 0;
}
```

count() 用来查找set中某个某个键值出现的次数。这个函数在set并不是很实用，因为一个键值在set只可能出现0或1次，这样就变成了判断某一键值是否在set出现过了。

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    set<int> s;
    s.insert(1);
    s.insert(2);
    s.insert(3);
    s.insert(1);
    cout<<"set 中 1 出现的次数是 :"<<s.count(1)<<endl;
    cout<<"set 中 4 出现的次数是 :"<<s.count(4)<<endl;
    return 0;
}
```

erase(iterator) ,删除定位器iterator指向的值

erase(first,second),删除定位器first和second之间的值

erase(key_value),删除键值key_value的值

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    set<int> s;
    set<int>::const_iterator iter;
    set<int>::iterator first;
    set<int>::iterator second;
    for(int i = 1 ; i <= 10 ; ++i)
    {
        s.insert(i);
    }
    //第一种删除
    s.erase(s.begin());
    //第二种删除
    first = s.begin();
    second = s.begin();
    second++;
    second++;
    s.erase(first,second);
    //第三种删除
    s.erase(8);
    cout<<"删除后 set 中元素是 :";
    for(iter = s.begin() ; iter != s.end() ; ++iter)
    {
        cout<<*iter<<" ";
    }
    cout<<endl;
    return 0;
}
```

find()，返回给定值值得定位器，如果没找到则返回end()。

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    int a[] = {1,2,3};
    set<int> s(a,a+3);
    set<int>::iterator iter;
    if((iter = s.find(2)) != s.end())
    {
        cout<<*iter<<endl;
    }
    return 0;
}
```

insert(key_value); 将key_value插入到set中 ，返回值是pair<set<int>::iterator,bool>，bool标志着插入是否成功，而iterator代表插入的位置，若key_value已经在set中，则iterator表示的key_value在set中的位置。

inset(first,second);将定位器first到second之间的元素插入到set中，返回值是void.

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    int a[] = {1,2,3};
    set<int> s;
    set<int>::iterator iter;
    s.insert(a,a+3);
    for(iter = s.begin() ; iter != s.end() ; ++iter)
    {
        cout<<*iter<<" ";
    }
    cout<<endl;
    pair<set<int>::iterator,bool> pr;
    pr = s.insert(5);
    if(pr.second)
    {
        cout<<*pr.first<<endl;
    }
    return 0;
}
```

lower_bound(key_value)，返回第一个大于等于key_value的定位器

upper_bound(key_value)，返回最后一个大于等于key_value的定位器

```cpp
#include <iostream>
#include <set>

using namespace std;

int main()
{
    set<int> s;
    s.insert(1);
    s.insert(3);
    s.insert(4);
    cout<<*s.lower_bound(2)<<endl;
    cout<<*s.lower_bound(3)<<endl;
    cout<<*s.upper_bound(3)<<endl;
    return 0;
}
```

map

Map是STL的一个关联容器，它提供一对一（其中第一个可以称为关键字，每个关键字只能在map中出现一次，第二个可能称为该关键字的值）的数据处理能力，由于这个特性，它完成有可能在我们处理一对一数据的时候，在编程上提供快速通道。这里说下map内部数据的组织，map内部自建一颗红黑树(一种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，所以在map内部所有的数据都是有序的.
我们通常用如下方法构造一个map： map<int, string> mapStudent;
在构造map容器后，我们就可以往里面插入数据了。这里讲三种插入数据的方法。
第一种：用insert函数插入pair数据
第二种：用insert函数插入value_type数据
第三种：用数组方式插入数据

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
    map<int,string> studentMessage;
    map<int,string>::iterator iter;
    studentMessage.insert(pair<int , string>
(54090101,"Mike"));
    studentMessage.insert(pair<int , string>
(54090102,"Sam"));
    studentMessage.insert(pair<int , string>
(54090103,"Jake"));
    //begin获取map中的第一个元素的迭代器,并且等于rend
    //end获取map中的最后一个元素下一位置的迭代器，并且等于rbegin
    cout<<"迭代器中的元素如下："<<endl;
    for(iter = studentMessage.begin() ; iter !=
studentMessage.end() ; ++iter)
    {
```

```
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    //看看max_size和size的值得意义
    cout<<"map 的 max_size 的值："
<<studentMessage.max_size()<<endl;
    cout<<"map 的 size 的值："<<studentMessage.size()
<<endl;
    //看看empty和clear的使用
    studentMessage.clear();
    if(studentMessage.empty())
    {
        cout<<"The map is Empty !!"<<endl;
    }
    else
    {
        cout<<"The map is not Empty !!"<<endl;
    }
    return 0;
}
```

map中用来查找的函数是find，但是能完成查找功能的函数却并不止这一个，比如count也是可以完成查找的，因为map中的键值是不允许重复的，所以一个键值只能出现一次，这说明count的返回值就只能是0或1了，那么显然这就能完成查找了，但是用count来完成查找并不是最优的选择，因为原来的本意是用count来完成计数的，而map中之所以有这个count函数，就是为了STL提供统一的接口，这样说来map中的upper_bound和lower_bound，equel_range等函数组合起来也是可以完成查找功能.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main()
{
    map<int,string> studentMessage;

studentMessage.insert(map<int,string>::value_type(54090101
,"Mike"));

studentMessage.insert(map<int,string>::value_type(54090102
,"Sam"));

studentMessage.insert(map<int,string>::value_type(54090103
,"Jake"));
    if(studentMessage.find(54090101) !=
studentMessage.end())
    {
        cout<<"find success !!"<<endl;
    }
    if(studentMessage.count(54090101))
    {
        cout<<"count success !!"<<endl;
    }
    return 0;
}
```

Java

java在比赛中最常用的就是大数了。
基本函数：
1.valueOf(parament); 将参数转换为制定的类型
比如 int a=3;
BigInteger b=BigInteger.valueOf(a);
则b=3;
String s=”12345”;
BigInteger c=BigInteger.valueOf(s);
则c=12345；

2.add(); 大整数相加

```
    BigInteger a=new BigInteger(“23”);
    BigInteger b=new BigInteger(“34”);
    a.add(b);
```

3.subtract(); 相减
4.multiply(); 相乘
5.divide(); 相除取整
6.remainder(); 取余
7.pow(); a.pow(b)=a^b
8.gcd(); 最大公约数
9.abs(); 绝对值
10.negate(); 取反数
11.mod(); a.mod(b)=a%b=a.remainder(b);
12.max(); min();
13.punlic int comareTo();
14.boolean equals(); 是否相等
15.BigInteger构造函数：
一般用到以下两种： BigInteger(String val);
将指定字符串转换为十进制表示形式；

BigInteger(String val,int radix); 将指定基数的 BigInteger 的字符串表示形式转换为 BigInteger

Ⅱ.基本常量：

```
A=BigInteger.ONE    1
B=BigInteger.TEN    10
C=BigInteger.ZERO   0
```

Ⅲ.基本操作

1. 读入： 用Scanner类定义对象进行控制台读入,Scanner类在java.util.*包中

```
Scanner cin=new Scanner(System.in);// 读入
while(cin.hasNext())   //等同于!=EOF
{
int n;
BigInteger m;
n=cin.nextInt(); //读入一个int;
m=cin.BigInteger();//读入一个BigInteger;
System.out.print(m.toString());
}
```

示例：

```
import java.util.Scanner;
import java.math.*;
import java.text.*;
public class Main
{
public static void main(String args[])
{
   Scanner cin = new Scanner ( System.in );
   BigInteger a,b;
   int c;
   char op;
   String s;
```

```
    while( cin.hasNext() )
    {
     a = cin.nextBigInteger();
     s = cin.next();
     op = s.charAt(0);
     if( op == '+')
     {
      b = cin.nextBigInteger();
      System.out.println(a.add(b));
     }
     else if( op == '-')
     {
      b = cin.nextBigInteger();
      System.out.println(a.subtract(b));
     }
     else if( op == '*')
     {
      b = cin.nextBigInteger();
      System.out.println(a.multiply(b));
     }
     else
     {
      BigDecimal a1,b1,eps;
      String s1,s2,temp;
      s1 = a.toString();
        a1 = new BigDecimal(s1);
      b = cin.nextBigInteger();
      s2 = b.toString();
      b1 = new BigDecimal(s2);
      c = cin.nextInt();
      eps = a1.divide(b1,c,4);
      //System.out.println(a + " " + b + " " + c);
      //System.out.println(a1.doubleValue() + " " +
b1.doubleValue() + " " + c);
      System.out.print( a.divide(b) + " " + a.mod(b) + "
```

```
");
     if( c != 0)
     {
      temp = "0.";
      for(int i = 0; i < c; i ++) temp += "0";
      DecimalFormat gd = new DecimalFormat(temp);
      System.out.println(gd.format(eps));
     }
     else System.out.println(eps);
    }
   }
   }
}
```

Python

个人觉得，python作为一门脚本语言拥有者和其他语言一样的能力，而且各大oj也相继支持python。虽然在正式比赛中不能使用提交，但是还是可以用python测试代码，生成测试数据等，python的简洁与优雅值得大家一学。 个人推荐两本书，《python 学习手册》 Mart Lutz著，比较全面的一本入门图书，但是还是适合有些基础来看。 《python 入门经典，以解决计算问题为导向的Python编程实践》适合零基础来学习，但是只是入门图书不够全面。

博弈论](bo_yi_lun.md)

---

author：李川皓

巴什博弈

简述

什么是巴什博弈：只有一堆n个物品，两个人轮流从这堆物品中取物， 规定每次至少取一个，最多取m个。最后取光者得胜。

分析

我们称先进行游戏的人为先手，另一个人为后手。

1、如果$$n=m+1$$，那么由于一次最多只能取$$m$$个，所以，无论先手拿走多少个，后手都能够一次拿走剩余的物品，后者取胜。
2、如果$$n=(m+1)*r*+*s*$$，(*r*为任意自然数，*$$s≤m$$),*先手要拿走*s*个物品，如果后手拿走*$$k(k≤m)$$*个，那么先手再拿走*$$m+1-k$$*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。我们得到如下结论:要保持给对手留下**$$(m+1)$$**的倍数，就能最后获胜。

必胜态必败态

只要$$n$$不能整除$$m+1$$,那么必然是先手取胜，否则后手取胜。

变形

如果我们规定最后取光者输，那么又会如何呢？

$$(n-1)$$%$$(m+1)==0$$则后手胜利 先手会重新决定策略，所以不是简单的相反的。
例如$$n=15，m=3$$
后手 先手 剩余
0 2 13
1 3 9
2 2 5

3 1 1
1 0 0
先手胜利 输的人最后必定只抓走一个，如果>1个，则必定会留一个给对手

例题

例题1.hdu 1846

题意：本游戏是一个二人游戏,有一堆石子一共有n个,两人轮流进行,每走一步可以取走1...m个石子,最先取光石子的一方为胜,如果游戏的双方使用的都是最优策略，请输出哪个人能赢。
要求：Time Limit: 1000 MS , Memory Limit: 32768 K
思路：最基础的巴什博奕模版题，不多说。

```
#include<cstdio>

int main()
{
    int t,n,m;
    scanf("%d",&t);
    while(t--)
    {
        scanf("%d%d",&n,&m);
        if(n%(m+1))
            printf("first\n");
        else
            printf("second\n");
    }

    return 0;
}
```

例题2.hdu 2147

题意：题目大意：就是有一个游戏，在一个$$n*m$$的矩阵中起始位置是$$（1，m）$$，走到终止位置$$（n,1）$$；游戏规则是只能向左，向下，左下方向走，先走到终点的为获胜者。

要求：*Time Limit: 1000 MS , Memory Limit: 1000 K*

*思路：我们可以寻找必败态和必胜态，我们假设必败态是P，必胜态是N，那么当$$n=8$$、$$m=9$$时，必胜态必败态矩阵如下：

NNNNNNNNN
PNPNPNPNP
NNNNNNNNN
PNPNPNPNP
NNNNNNNNN
PNPNPNPNP
NNNNNNNNN
PNPNPNPNP

多试几组样例，我们可以很容易的发现，当$$n$$和$$m$$都为奇数的时候，按照题意，应输出“What a pity！”，否则输出“Wonderful！”。

```
#include<cstdio>
#include<cstring>

int main()
{
    int n,m;
    while(~scanf("%d%d",&n,&m))
    {
        if(n==0&&m==0)
            return 0;
        if(n%2==1&&m%2==1)
            printf("What a pity!\n");
        else
            printf("Wonderful!\n");
    }
    return 0;
}
```

威佐夫博奕

简述

威佐夫博弈(Wythoff Game)：有两堆各若干个物品，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

分析

我们用$$（ak，bk）$$$$（ak ≤ bk ,k=0，1，2，...,n)$$表示两堆物品的数量并称其为局势，如果甲面对$$（0，0）$$，那么甲已经输了，这种局势我们称为奇异局势。前几个奇异局势是：$$（0，0）$$、$$（1，2）$$、$$（3，5）$$、$$（4，7）$$、$$（6，10）$$、$$（8，13）$$、$$（9，15）$$、$$（11，18）$$、$$（12，20）$$。 可以看出：$$a0=b0=0$$,$$ak$$是未在前面出现过的最小自然数,而 $$bk= ak + k$$。

必胜态必败态

满足 $$ak =k*（1+\sqrt5）/2，bk= ak + k$$，后手必胜，否则先手必胜。

例题

例题1.hdu 1527

题意：有两堆石子，数量任意，可以不同。游戏开始由两个人轮流取石子。游戏规定，每次有两种不同的取法，一是可以在任意的一堆中取走任意多的石子；二是可以在两堆中同时取走相同数量的石子。最后把石子全部取完者为胜者。现在给出初始的两堆石子的数目，如果轮到你先取，假设双方都采取最好的策略，问最后你是胜者还是败者。
要求：Time Limit: 1000 MS , Memory Limit: 32768 K
思路：基本是威佐夫博奕模版题。

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<cmath>

using namespace std;

int main()
{
    int a,b;
    while(scanf("%d%d",&a,&b)!=EOF)
    {
        int aa=max(a,b);
        int bb=min(a,b);
        int k=aa-bb;
        int temp=(k*(1+sqrt(5))/2);
        //printf("aa=%d   bb=%d    k=%d
temp=%d\n",aa,bb,k,temp);
        if(bb==temp)
            printf("0\n");
        else
            printf("1\n");
    }
    return 0;
}
```

Nim博弈

简述

通常的Nim游戏的定义是这样的：有若干堆石子，每堆石子的数量都是有限的，合法的移动是“选择一堆石子并拿走若干颗（不能不拿）”，如果轮到某个人时所有的石子堆都已经被拿空了，则判负（因为他此刻没有任何合法的移动）。

分析

这游戏看上去有点复杂，先从简单情况开始研究吧。如果轮到你的时候，只剩下一堆石子，那么此时的必胜策略肯定是把这堆石子全部拿完一颗也不给对手剩，然后对手就输了。如果剩下两堆不相等的石子，必胜策略是通过取多的一堆的石子将两堆石子变得相等，以后如果对手在某一堆里拿若干颗，你就可以在另一堆中拿同样多的颗数，直至胜利。如果你面对的是两堆相等的石子，那么此时你是没有任何必胜策略的，反而对手可以遵循上面的策略保证必胜。如果是三堆石子……好像已经很难分析了，看来我们必须要借助一些其它好用的（最好是程式化的）分析方法了，或者说，我们最好能够设计出一种在有必胜策略时就能找到必胜策略的算法。

定义$$P-position$$和$$N-position$$，其中$$P$$代表$$Previous$$，$$N$$代表$$Next$$。直观的说，上一次移动石子的人有必胜策略的局面是$$P-position$$，也就是“后手可保证必胜”或者“先手必败”，现在轮到移动石子的人有必胜策略的局面是$$N-position$$，也就是“先手可保证必胜”。更严谨的定义是：

- 无法进行任何移动的局面（也就是$$terminal position$$）是$$P-position$$；
- 可以移动到$$P-position$$的局面是$$N-position$$；
- 所有移动都导致$$N-position$$的局面是$$P-position$$。

按照这个定义，如果局面不可能重现，或者说$$positions$$的集合可以进行拓扑排序，那么每个$$position$$或者是$$P-position$$或者是$$N-position$$，而且可以通过定义计算出来。

以$$Nim$$游戏为例来进行一下计算。比如说我刚才说当只有两堆石子且两堆石子数量相等时后手有必胜策略，也就是这是一个$$P-position$$，下面我们依靠定义证明一下$$(3,3)$$是一个$$P-position$$。首先$$(3,3)$$的子局面（也就是通过合法移动可以导致的局面）有$$(0,3)(1,3)(2,3)$$（显然交换石子堆的位置不影响其性

质，所以把$$(x,y)$$和$$(y,x)$$看成同一种局面），只需要计算出这三种局面的性质就可以了。$$ (0,3)$$的子局面有$$(0,0)、(0,1)、(0,2)$$，其中$$(0,0)$$显然是$$P-position$$，所以$$(0,3)$$是$$N-position$$（只要找到一个是$$P-position$$的子局面就能说明是$$N-position$$）。$$(1,3)$$的后继中$$(1,1)$$是$$P-position$$（因为$$(1,1)$$的唯一子局面$$(0,1)$$是$$N-position$$），所以$$(1,3)$$也是$$N-position$$。同样可以证明$$(2,3)$$是$$N-position$$。所以$$(3,3)$$的所有子局面都是$$N-position$$，它就是$$P-position$$。通过一点简单的数学归纳，可以严格的证明“有两堆石子时的局面是$$P-position$$当且仅当这两堆石子的数目相等”。

根据上面这个过程，可以得到一个递归的算法——对于当前的局面，递归计算它的所有子局面的性质，如果存在某个子局面是$$P-position$$，那么向这个子局面的移动就是必胜策略。当然，可能你已经敏锐地看出有大量的重叠子问题，所以可以用$$DP$$或者记忆化搜索的方法以提高效率。但问题是，利用这个算法，对于某个$$Nim$$游戏的局面$$(a1,a2,...,an)$$来说，要想判断它的性质以及找出必胜策略，需要计算$$O(a1a2...*an)$$个局面的性质，不管怎样记忆化都无法降低这个时间复杂度。所以我们需要更高效的判断$$Nim$$游戏的局面的性质的方法。

必胜态必败态

对于一个$$Nim$$游戏的局面$$(a1,a2,...,an)$$，它是$$P-position$$当且仅当$$a1$$^$$a2$$^...^$$an=0$$，其中^表示异或(xor)运算。

例题

例题1.hdu 1849

题意：游戏的规则是这样的：

- 棋盘包含1*n个方格，方格从左到右分别编号为0，1，2，...，n-1；
- m个棋子放在棋盘的方格上，方格可以为空，也可以放多于一个的棋子；
- 双方轮流走棋；
- 每一步可以选择任意一个棋子向左移动到任意的位置（可以多个棋子位于同一个方格），当然，任何棋子不能超出棋盘边界；
- 如果所有的棋子都位于最左边（即编号为0的位置），则游戏结束，并且规定最后走棋的一方为胜者。

给出初始棋子状态，输出先手赢，或者后手赢。

要求：Time Limit: 1000 MS , Memory Limit: 32768 K
思路：

```
#include<cstdio>
#include<cstring>

int num[1500];
int sg[1500];

int main()
{
    int n;
    while(~scanf("%d",&n))
    {
        if(n==0)
            return 0;
        int sum=0;
        memset(num,0,sizeof(num));
        for(int i=1;i<=n;i++)
        {
            scanf("%d",&num[i]);
            sum^=num[i];
        }
        if(sum)
            printf("Rabbit Win!\n");
        else
            printf("Grass Win!\n");
    }
    return 0;
}
```

扩充

相信大家觉得一道Nim博弈的例题根本不够做，不用担心，都在下一节（SG函数）呢，大家继续往下看吧~

SG函数

简述

想要学好博弈论，不只是要了解巴什博奕、威佐夫博奕等等典型的博弈问题，这些都只是基础，而SG函数，是我们想要走向博弈大师必须掌握的知识，接下来的分析，可能枯燥无味，但是眞正看懂之后，会受益匪浅。

分析

现在我们来研究一个看上去似乎更为一般的游戏：给定一个有向无环图和一个起始顶点上的一枚棋子，两名选手交替的将这枚棋子沿有向边进行移动，无法移动者判负。事实上，这个游戏可以认为是所有Impartial Combinatorial Games（公平组合游戏，以下简称ICG）的抽象模型。也就是说，任何一个ICG都可以通过把每个局面看成一个顶点，对每个局面和它的子局面连一条有向边来抽象成这个“有向图游戏”。下 面我们就在有向无环图的顶点上定义Sprague-Garundy函数。

首先定义mex(minimal excludant)运算，这是施加于一个集合的运算，表示最小的不属于这个集合的非负整数。例如mex{0,1,2,4}=3、 mex{2,3,5}=0、mex{}=0。

对于一个给定的有向无环图，定义关于图的每个顶点的Sprague-Garundy函数g如下：g(x)=mex{ g(y) | y是x的后继 }。

来看一下SG函数的性质。首先，所有的terminal position所对应的顶点，也就是没有出边的顶点，其SG值为0，因为它的后继集合是空集。然后对于一个g(x)=0的顶点x，它的所有后继y都满足 g(y)!=0。对于一个g(x)!=0的顶点，必定存在一个后继y满足g(y)=0。

以上这三句话表明，顶点x所代表的postion是P-position当且仅当g(x)=0（跟P-positioin/N-position的定义的 那三句话是完全对应的）。我们通过计算有向无环图的每个顶点的SG值，就可以对每种局面找到必胜策略了。但SG函数的用途远没有这样简单。如果将有向图游 戏变复杂一点，比如说，有向图上并不是只有一枚棋子，而是有n枚棋子，每次可以任选一颗进行移动，这时，怎样找到必胜策略呢？

让我们再来考虑一下顶点的SG值的意义。当g(x)=k时，表明对于任意一个0<=i<k，都存在x的一个后继y满足g(y)=i。也就是 说，当某枚棋子的SG值是k时，我们可以把它变成0、变成1、……、变成k-1，但绝对不能保持k不变。不知道你能不能根据这个联想到Nim游戏，Nim 游戏的规则就是：每次选择一堆数量为k的石子，可以把它变成0、变成1、……、变成k-1，但绝对不能保持k不变。这表明，如果将n枚棋子所在的顶点的 SG值看作n堆相应数量的石子，那么这个Nim游戏的每个必胜策略都对应于原来这n枚棋子的必胜策略！

对于n个棋子，设它们对应的顶点的SG值分别为(a1,a2,...,an)，再设局面(a1,a2,...,an)时的Nim游戏的一种必胜策略是把 ai变成k，那么原游戏的一种必胜策略就是把第i枚棋子移动到一个SG值为k的顶点。这听上去有点过于神奇——怎么绕了一圈又回到Nim游戏上了。

其实我们还是只要证明这种多棋子的有向图游戏的局面是P-position当且仅当所有棋子所在的位置的SG函数的异或为0。这个证明与上节的 Bouton's Theorem几乎是完全相同的，只需要适当的改几个名词就行了。

刚才，我为了使问题看上去更容易一些，认为n枚棋子是在一个有向图上移动。但如果不是在一个有向图上，而是每个棋子在一个有向图上，每次可以任选一个棋子（也就是任选一个有向图）进行移动，这样也不会给结论带来任何变化。

所以我们可以定义有向图游戏的和(Sum of Graph Games)：设G1、G2、……、Gn是n个有向图游戏，定义游戏G是G1、G2、……、Gn的和(Sum)，游戏G的移动规则是：任选一个子游戏Gi 并移动上面的棋子。Sprague-Grundy Theorem就是：g(G)=g(G1)^g(G2)^...^g(Gn)。也就是说，游戏的和的SG函数值是它的所有子游戏的SG函数值的异或。

再考虑在本文一开头的一句话：任何一个ICG都可以抽象成一个有向图游戏。所以“SG函数”和“游戏的和”的概念就不是局限于有向图游戏。我们给每个 ICG的每个position定义SG值，也可以定义n个ICG的和。所以说当我们面对由n个游戏组合成的一个游戏时，只需对于每个游戏找出求它的每个局 面的SG值的方法，就可以把这些SG值全部看成Nim的石子堆，然后依照找Nim的必胜策略的方法来找这个游戏的必胜策略了！

我们再看Nim游戏：有n堆石子，每次可以从第1堆石子里取1颗、2颗或3颗，可以从第2堆石子里取奇数颗，可以从第3堆及以后石子里取任意颗……我们可 以把它看作3个子游戏，第1个子游戏只有一堆石子，每次可以取1、2、3颗，很容易看出x颗石子的局面的SG值是x%4。第2个子游戏也是只有一堆石子， 每次可以取奇数颗，经过简单的画图可以知道这个游戏有x颗石子时的SG值是x%2。第3个游戏有n-2堆

石子，就是一个Nim游戏。对于原游戏的每个局 面，把三个子游戏的SG值异或一下就得到了整个游戏的SG值，然后就可以根据这个SG值判断是否有必胜策略以及做出决策了。其实看作3个子游戏还是保守了 些，干脆看作n个子游戏，其中第1、2个子游戏如上所述，第3个及以后的子游戏都是“1堆石子，每次取几颗都可以”，称为“任取石子游戏”，这个超简单的 游戏有x颗石子的SG值显然就是x。其实，n堆石子的Nim游戏本身不就是n个“任取石子游戏”的和吗？

所以，对于我们来说，SG函数与“游戏的和”的概念不是让我们去组合、制造稀奇古怪的游戏，而是把遇到的看上去有些复杂的游戏试图分成若干个子游戏，对于 每个比原游戏简化很多的子游戏找出它的SG函数，然后全部异或起来就得到了原游戏的SG函数，就可以解决原游戏了。

模版

- 计算从1-n范围内的SG值

```
/*
Array(存储可以走的步数，Array[0]表示可以有多少种走法)
Array[]需要从小到大排序
1.可选步数为1-m的连续整数，直接取模即可，SG(x)=x%(m+1);
2.可选步数为任意步，SG(x) = x;
3.可选步数为一系列不连续的数，用GetSG(计算)
*/
int SG[MAX],hash[MAX];
void init(int Array[],int n)
{
    int i,j;
    memset(SG,0,sizeof(SG));
    for(i=0; i<=n; i++)
    {
        memset(hash,0,sizeof(hash));
        for(j=1; j<=Array[0]; j++)
        {
            if(i<Array[j])
                break;
            hash[SG[i-Array[j]]]=1;
        }
        for(j=0; j<=n; j++)
        {
            if(hash[j]==0)
            {
                SG[i]=j;
                break;
            }
        }
    }
}
```

有些时候预处理出来所有的SG值会超时，而且有些SG值是用不到的，这时候我们会用到下面的模版，获得单独的一个SG值。

- 获得一个单独的SG值

```
int s[101],sg[10001],k; //k为可走步数，s数组存储可走步数（0~k-1）
int getsg(int m)
{
    int hash[101]={0};
    int i;
    for(i=0; i<k; i++)
    {
        if(m-s[i]<0)
            break;
        if(sg[m-s[i]]==-1)
            sg[m-s[i]]=getsg(m-s[i]);
        hash[sg[m-s[i]]]=1;
    }
    for(i=0;; i++)
        if(hash[i]==0)
            return i;
}
```

例题

例题1.hdu 1848

题意：

- 这是一个二人游戏;
- 一共有3堆石子，数量分别是m, n, p个；
- 两人轮流走;
- 每走一步可以选择任意一堆石子，然后取走f个；
- f只能是菲波那契数列中的元素（即每次只能取1，2，3，5，8...等数量）；
- 最先取光所有石子的人为胜者；
- 假设双方都使用最优策略，请判断先手的人会赢还是后手的人会赢。

要求：Time Limit: 1000 MS , Memory Limit: 32768 K
思路：乍一瞅，像不像Nim博弈？不过单纯的Nim博弈无法解决它，我们要用到这一节学到的SG函数，套用上面的模版，相信大家能看明白代码~

```
#include <iostream>
using namespace std;
#define MAX 1005
#include<cstdio>
#include<cstring>

int SG[MAX], hash[MAX];
void init(int Array[], int n)
{
    int i, j;
    memset(SG, 0, sizeof(SG));
    for(i = 0; i <= n; i++)
    {
        memset(hash, 0, sizeof(hash));
        for(j = 1; j <= Array[0]; j++)
        {
            if(i < Array[j])
                break;
            hash[SG[i - Array[j]]] = 1;
        }
        for(j = 0; j <= n; j++)
        {
            if(hash[j] == 0)
            {
                SG[i] = j;
                break;
            }
        }
    }
}

int main()
```

```
{
    int m,n,p;
    int fibo[30];
    fibo[0]=19;
    fibo[1]=1;
    fibo[2]=1;
    for(int i=3;i<=19;i++)
        fibo[i]=fibo[i-1]+fibo[i-2];
    init(fibo,1000);
    //puts("lalal");
    while(~scanf("%d%d%d",&m,&n,&p))
    {
        if(m==0&&n==0&&p==0)
            return 0;
        if((SG[m]^SG[n]^SG[p])==0)
            printf("Nacci\n");
        else
            printf("Fibo\n");
    }
    return 0;
}
```

例题2.hdu 1846

题意：本游戏是一个二人游戏,有一堆石子一共有n个,两人轮流进行,每走一步可以取走1...m个石子,最先取光石子的一方为胜,如果游戏的双方使用的都是最优策略，请输出哪个人能赢。

要求：Time Limit: 1000 MS , Memory Limit: 32768 K

思路：大家看见这道题的题面一定觉得特别熟悉，没错，这道题就是巴什博奕中的例题1，在这里我们用SG函数的方法来解决这道题。在第一个模版中的注释中的第一条中我们可以看到，可选步数为$$1$$~$$m$$的连续整数，直接取模即可，$$SG(x)=x$$%$$(m+1)$$，所以这道巴什博奕的题我们能够直接得到SG值，从而解决它！

```
#include<cstdio>
#include<cstring>

int main()
{
    int t;
    scanf("%d",&t);
    while(t--)
    {
        int n,m;
        scanf("%d%d",&n,&m);
        int SG[1500];
        memset(SG,0,sizeof(SG));
        for(int i=1;i<=n;i++)
            SG[i]=i%(m+1);
        if(SG[n])
            printf("first\n");
        else
            printf("second\n");
    }
    return 0;
}
```

例题3.hdu 1536

题意：

- 输入K表示一个集合的大小，之后输入集合表示对于这对石子只能去这个集合中的元素的个数
- 输入一个m表示接下来对于这个集合要进行m次询问
- 接下来m行 每行输入一个n，表示有n个堆，每堆有ni个石子，问这一行所表示的状态是赢还是输 如果赢输出W否则L

要求：Time Limit: 1000 MS , Memory Limit: 32768 K
思路：如果做明白了前面两个例题，大家很容易就能想到，无非是多调用几次SG函数的模版啊，年少无知的我当时也是这么想的，但是。。。超时了~所以我们要

动用第二个模版啦，采用类似于“记忆化搜索”的方式，用到哪个SG值，现求就好了，求过的就存好，下一次用到直接调用，这样能省去不少时间。
超时代码：

```
#include<cstdio>
#include<cstring>

int SG[10005],hash[10005];
void init(int Array[],int n)
{
    int i,j;
    memset(SG,0,sizeof(SG));
    for(i=0; i<=n; i++)
    {
        memset(hash,0,sizeof(hash));
        for(j=1; j<=Array[0]; j++)
        {
            if(i<Array[j])
                break;
            hash[SG[i-Array[j]]]=1;
        }
        for(j=0; j<=n; j++)
        {
            if(hash[j]==0)
            {
                SG[i]=j;
                break;
            }
        }
    }
}

int main()
{
    int snum;
    while(scanf("%d",&snum))
```

```
    {
        if(snum==0)
            return 0;
        int s[10005];
        memset(s,0,sizeof(s));
        s[0]=snum;
        for(int i=1; i<=snum; i++)
            scanf("%d",&s[i]);
        //init(s,10005);
        int n;
        scanf("%d",&n);
        memset(SG,0,sizeof(SG));
        for(int i=1; i<=n; i++)
        {
            int nn;
            scanf("%d",&nn);
            int sum=0;
            while(nn--)
            {
                int nnn;
                scanf("%d",&nnn);
                if(SG[nnn]==0)
                    init(s,nnn);
                sum^=SG[nnn];
            }
            if(sum)
                printf("W");
            else
                printf("L");
        }
        printf("\n");
    }
    return 0;
}
```

正确代码：

```
#include<cstdio>
#include<cstring>
#include<algorithm>

using namespace std;
int sg[10005],hash[10005];
int s[100005];
int k;
int getsg(int m)
{
    int hash[101]= {0};
    int i;
    for(i=0; i<k; i++)
    {
        if(m-s[i]<0)
            break;
        if(sg[m-s[i]]==-1)
            sg[m-s[i]]=getsg(m-s[i]);
        hash[sg[m-s[i]]]=1;
    }
    for(i=0;; i++)
        if(hash[i]==0)
            return i;
}

int main()
{
    int snum;
    while(scanf("%d",&snum))
    {
        k=snum;
        if(snum==0)
            return 0;
        memset(s,0,sizeof(s));
```

```
        for(int i=0; i<snum; i++)
            scanf("%d",&s[i]);
        sort(s,s+k);
        //init(s,10005);
        int n;
        scanf("%d",&n);
        memset(sg,-1,sizeof(sg));
        for(int i=1; i<=n; i++)
        {
            int nn;
            scanf("%d",&nn);
            int sum=0;
            while(nn--)
            {
                int nnn;
                scanf("%d",&nnn);
                if(sg[nnn]==-1)
                    sg[nnn]=getsg(nnn);
                sum^=sg[nnn];
            }
            if(sum)
                printf("W");
            else
                printf("L");
        }
        printf("\n");
    }
    return 0;
}
```