structure

位操作

与操作 &

i. 用以取出一个数的某些二进制位
ii. 取出一个数二进制中的最后一个 1：x&-x

或操作 |

用以将一个数的某些位设为 1

非操作 ~

用以间接构造一些数：~0u=4294967295=232-1

异或操作 ^

i. 不使用中间变量交换两个数：a=a^b;b=a^b;a=a^b;
ii. 将一个数的某些位取反

素数

费马小定理

如果 p 是素数，则 a^(p−1)≡1(mod p)对所有整数 a 都成立

GCD

GCD 结论

有俩个数 p,q,且 gcd(q,p)=1,则最大无法表示成 px+qy（x>=0,y>=0)的数是 pq-q-p.

区间内与 n 的 gcd 不小于 m 的个数

输入 m, n,求 1~n 之间中 gcd(x, n) >=m 的 x 个数。

找出 N 的所有大于等于 M 的因子(x1,x2,x3.....xi)，然后设 k=N/xi；
下面只需找出小于 k 且与 k 互质的数。
因为：设 y 与 k 互质且小于 k，那么 gcd（y*xi，k*xi）=xi；（xi 为 N 的因子，且 xi 大于等于 M）。

约瑟夫环的数学方法

问题描述：n 个人（编号 0~(n-1)），从 0 开始报数，报到(m-1)的退出，剩下的人继续从 0 开始报数。求胜利者的编号。

我们知道第一个人(编号一定是(m-1)%n) 出列之后，剩下的 n-1 个人组成了一个新的约瑟夫环（以编号为 k=m%n 的人开始）:
k k+1 k+2 ... n-2, n-1, 0, 1, 2, ... k-2
并且从 k 开始报 0。
现在我们把他们的编号做一下转换：
序列 1： 1, 2, 3, 4, …, n-2, n-1, n
序列 2： 1, 2, 3, 4, … k-1, k+1, …, n-2, n-1, n
序列 3：k+1, k+2, k+3, …, n-2, n-1, n, 1, 2, 3,…, k-2, k-1
序列 4：1, 2, 3, 4, …, 5, 6, 7, 8, …, n-2, n-1
变换后就完完全全成为了(n-1)个人报数的子问题，假如我们知道这个子问题的解：例如 x 是最终的胜利者，那么根据上面这个表把这个 x 变回去不刚好就是 n 个人情况的解吗？!! 变回去的公式很简单，相信大家都可以推出来(其实就是利用子问题的解等价转换人数等于 n 的解,因为 n 在转化成 n-1 时已经出队一个人了,剩下 n-1 的最后出队人仍然和 n 的解相同,只是需要映射将下标到人数为 n 的情况)：
∵ k=m%n;
∴ x' = x+k = x+ m%n ; 而 x+ m%n 可能大于 n
∴x'= (x+ m%n)%n = (x+m)%n
得到 x‘=(x+m)%n
如何知道(n-1)个人报数的问题的解？对，只要知道(n-2)个人的解就行了。(n-2)个人的解呢？当然是先求(n-3)的情况 ---- 这显然就是一个倒推问题！好了，思路出来了，下面写递推公式：
令 f 表示 i 个人玩游戏报 m 退出最后胜利者的编号，最后的结果自然是 f[n].
递推公式:
f[1]=0;
f[i]=(f[i-1]+m)%i; (i>1)
有了这个公式，我们要做的就是从 1-n 顺序算出 f 的数值，最后结果是 f[n]。因为实际生活中编号总是从 1 开始，我们输出 f[n]+1 由于是逐级递推，不需要保存每个 f，程序也是异常简单：

```
int getAns(int n,int m){
    int ans = 0;
    for (int i=2; i<=n; i++){
        ans=(ans+m)%i;
    }
    return ans+1;
}
```

pick 定理

如果顶点都为整数坐标点，面积=边点/2+内点-1

catalan 数

H(n+1) = 2(2n+1)/(n+2) * H(n)
H(n+1)=sum(H(i)*H(n-i)) (0<=i<=n)
H(n) = sum(C(n,i) * C(n,i)) /(n+1) (0<=i<=n)
H(n) ~= 4^n/(n^1.5 * sqrt(pi))

应用

1.n 个数的不同出栈序列
2.N 个+1 和 n 个-1 构成 2n 项 a1a2...an，其部分和满足 a1+a2+a3+...+an>=0，0<=k<=2n。满足这个序列的个数等于第 n 个 catakan 数。
3.括号匹配的合法个数
4.连乘的选择个数
5.n 个节点的二叉树的树的形态的个数。
6.n 个非叶子节点的满二叉树的形态数。
7.n*n 的矩阵中，从右下角到左上角的走法。
8.凸 n+2 边形进行三角分割数。
9.n 层的阶梯切割为 n 个矩阵的切割法数。
10.在一个 2*n 的格子中填入 1 到 2n 这些数字，使每个格子内的数值都比其右边和上边的所有数值都小的情况数。

随机函数

```
int rands() {
    static int x=1364684679;
    x+=(x<<2)+1;
    return x;
}
```

压位筛素数

```
typedef long long LL;
LL NN = 2147483647LL;
const int N= (2147483647>>3)+1;
const int M=14630853;
char is[N];
LL prm[M];
void setIs(int pos){
    is[pos>>3] &= ~(1<<(pos%8));
}

bool getIs(int pos){
    return is[pos>>3] & (1<<(pos%8));
}

int getprm(){
    int e = (int)(sqrt(0.0 + NN) + 1),k=0,i;
    memset(is, 0XFF, sizeof(is));
    prm[k++] = 2;
    setIs(0);
    setIs(1);
    for (i = 4; i < N; i += 2){
        setIs(i);
    }

    for(i=3;i<e;i+=2){
        if(getIs(i)){
            prm[k++]=i;
            for(int s=i+i,j=i*i;j<N;j+=s){
                setIs(j);
            }
        }
    }
    for (; i < N; i += 2)
        if (getIs(i))prm[k++] = i;
    return k;
}
```

矩阵

判断 A * B == C

随机算法

```
bool eq(int i,int j,int sz){
    int tmp=0,k;
    for(k=0;k<sz;k++){
        tmp += A[i][k]*B[k][j];
    }
    return tmp == C[i][j];
```

```
}

const int L = 10000;
bool randTest(int sz){
    int i,j,k;
    for(k=0;k<L;k++){
        i = rand()%sz;
        j = rand()%sz;
        if(!eq(i,j,sz))return false;
    }

    return true;
}
```

伪随机算法

A*B 的复杂度是 O(n^3)，A 是一维的话，复杂度就是 O(n^2)了。
所以 A*B == C 可以近似的转化为 1 * A * B == 1 * C
1 是一维的向量。

矩阵幂相加

给定矩阵 A，求 A + A^2 + A^3 + ... + A^k 的结果（两个矩阵相加就是对应位置分别相加）。输出的数据 mod m。k<=10^9。

这道题两次二分，相当经典。首先我们知道，A^i 可以二分求出。然后我们需要对整个题目的数据规模 k 进行二分。比如，当 k=6 时，有：

A + A^2 + A^3 + A^4 + A^5 + A^6 =(A + A^2 + A^3) + A^3*(A + A^2 + A^3)

应用这个式子后，规模 k 减小了一半。我们二分求出 A^3 后再递归地计算 A + A^2 + A^3，即可得到原问题的答案。

构造矩阵

求第第 n 个 Fibonacci 数 mod p

给定 n 和 p，求第 n 个 Fibonacci 数 mod p 的值，n 不超过 2^31

现在我们需要构造一个 2 x 2 的矩阵，使得它乘以(a,b)得到的结果是(b,a+b)。每多乘一次这个矩阵，这两个数就会多迭代一次。那么，我们把这个 2 x 2 的矩阵自乘 n 次，再乘以(0,1)就可以得到第 n 个 Fibonacci 数了。不用多想，这个 2 x 2 的矩阵很容易构造出来：

$$\begin{pmatrix}0 & 1\\ 1 & 1\end{pmatrix}\cdot\begin{pmatrix}a\\ b\end{pmatrix}=\begin{pmatrix}b\\ a+b\end{pmatrix}$$

求有向图中 A 走 k 步到 B 的方案数

给定一个有向图，问从 A 点恰好走 k 步（允许重复经过边）到达 B 点的方案数 mod p 的值

把给定的图转为邻接矩阵，即 A(i,j)=1 当且仅当存在一条边 i->j。令 C=A*A，那么 C(i,j)= Σ A(i,k)*A(k,j)，实际上就等于从点 i 到点 j 恰好经过 2 条边的路径数（枚举 k 为中转点）。类似地，C*A 的第 i 行第 j 列就表示从 i 到 j 经过 3 条边的路径数。同理，如果要求经过 k 步的路径数，我们只需要二分求出 A^k 即可。

用 1 x 2 的矩阵填充 M x N 的矩阵的方案数。

用 1 x 2 的多米诺骨牌填满 M x N 的矩形有多少种方案，M<=5，N<2^31，输出答案 mod p 的结果

堆

```
const int maxn = 10000;

struct Heap{
    int size;
    int array[maxn];
    void bulid();
    void insert(int val);
    int top();
    void pop();
    bool empty();
    void push_down(int pre);
    void push_up(int son);
    bool compare(int pre,int son);
};

void Heap::bulid(){
    size = 0;
}

void Heap::insert(int val){
    array[++size] = val;
    push_up(size);
}
```

```
int Heap::top(){
   return array[1];
}

void Heap::pop(){
   array[1] = array[size--];
   push_down(1);
}

bool Heap::empty(){
   return size == 0;
}
void Heap::push_down_loop(int pre){
      while(true){
          if((pre<<1|1)<= size && compare(pre<<1|1,pre)
&& compare(pre<<1|1,pre<<1)){
                  //判断右儿子是否是父亲
                  swap(array[pre],array[pre<<1|1]);
                  pre = pre<<1|1;
    }else if((pre<<1) <= size && compare(pre<<1,pre)){
                  //判断左儿子是否是父亲
                  swap(array[pre],array[pre<<1]);
                    pre = pre<<1;
            }else{
                  break;
            }
      }
}
void Heap::push_up(int son){
      if(son == 1)return ;
      int pre = son>>1;
      if(!compare(pre,son)){
            swap(array[pre],array[son]);
            push_up(pre);
      }
}
//堆的性质返回 true
//也就是儿子不大于父亲返回 true
bool Heap::compare(int pre,int son){
      return array[pre] >= array[son];//最大堆
     // return array[pre] <= array[son];//最小堆
}
```

树套树

线段树套平衡树

动态查询区间第 k 小，包括两个操作 Q x y k 和 C i j，查询区间 x y 的第 k 小和把第 i 个数安替换成 j。

```
const int N=50010;
const int INF = 0x3f3f3f3f;

int tree[N<<1];

struct treap {
      int key,wht,count,sz,ch[2];
} tp[N*20];

int nodecount;

int id(int l,int r) {
      return l+r | l!=r;
}

void init() {
      tp[0].sz=0;
      tp[0].wht=-INF;
      tp[0].key=-1;
      nodecount=0;
}

//update this tree's size
void update(int x) {

tp[x].sz=tp[tp[x].ch[0]].sz+tp[x].count+tp[tp[x].ch[1]].sz;
}

void rotate(int &x,int t) {
      int y=tp[x].ch[t];
      tp[x].ch[t]=tp[y].ch[!t];
      tp[y].ch[!t]=x;
      update(x);
      update(y);
      x=y;
}


void insert(int &x,int t) {
      if(!x) {
```

```
            x=++nodecount;
            tp[x].key=t;
            tp[x].wht=rand();
            tp[x].count=1;
            tp[x].ch[0]=tp[x].ch[1]=0;
        } else if(tp[x].key==t) {
            tp[x].count++;
        } else {
            int k=tp[x].key<t;
            insert(tp[x].ch[k],t);
            if(tp[x].wht<tp[tp[x].ch[k]].wht) {
                rotate(x,k);
            }
        }
        update(x);
}

void erase(int &x,int t) {
        if(tp[x].key==t) {
            if(tp[x].count==1) {
                if(! tp[x].ch[0] && ! tp[x].ch[1]) {
                    x=0;
                    return;
                }

rotate(x,tp[tp[x].ch[0]].wht<tp[tp[x].ch[1]].wht);
                erase(x,t);
            } else tp[x].count--;
        } else erase(tp[x].ch[tp[x].key<t],t);
        update(x);
}
```

```
int select(int x,int k) {
        if(! x) return 0;
        if(k<tp[x].key) return select(tp[x].ch[0],k);
        int q=0,p=tp[tp[x].ch[0]].sz+tp[x].count;
        if(k>tp[x].key) q=select(tp[x].ch[1],k);
        return p+q;
}

int a[N],n,m,ans;
void treeinsert(int l,int r,int i,int x) {
        insert(tree[id(l,r)],x);
        if(l==r) return;
        int m=(l+r)>>1;
        if(i<=m) treeinsert(l,m,i,x);
        if(i>m) treeinsert(m+1,r,i,x);
}

void del(int l,int r,int i,int x) {
        erase(tree[id(l,r)],x);
        if(l==r) return;
        int m=(l+r)>>1;
        if(i<=m) del(l,m,i,x);
        if(i>m) del(m+1,r,i,x);
}

void query(int l,int r,int L,int R,int x) {
        if(L<=l && R>=r) {
            ans+=select(tree[id(l,r)],x);
            return;
        }
        int m=(l+r)>>1;
        if(L<=m) query(l,m,L,R,x);
        if(R>m) query(m+1,r,L,R,x);
}

int main() {
        int tt;
        scanf("%d",&tt);
        while (tt--) {
            scanf("%d%d",&n,&m);
            init();
            memset(tree,0,sizeof(tree));
            for(int i=1; i<=n; i++) {
                scanf("%d",&a[i]);
                treeinsert(1,n,i,a[i]);
            }
            while (m--) {
                char s[5];
                int x,y,c;
                scanf("%s",s);
                if(s[0]=='C') {
                    scanf("%d %d",&x,&y);
                    del(1,n,x,a[x]);
                    a[x]=y;
                    treeinsert(1,n,x,a[x]);
                } else {
                    scanf("%d %d %d",&x,&y,&c);
                    int l=0,r=INF,mid;
                    while (l<r) {
                        ans=0;
                        mid=(l+r)>>1;
```

```
                query(1,n,x,y,mid);
                if(ans<c) l=mid+1;
                else r=mid;
            }
            printf("%d\n",l);
        }
    }
    return 0;
}
```

划分树

主要模拟归并排序。
求区间第 K 数时，判断这个的数进入左半部多少个，记为 p，如果大于 k，则第 k 数就在左半部，否则就在右半部，且为右半部的第 k-p 数。
这样递归下去就可以了。

input 输入的数据，下标从 1 开始，数据可以重复
sortedPos input 的第几个位置的值应该在当前位置
val input 的当前值应该在那个位置

```
int n;
int arr[N];//原数据,下标从 1 开始
int sortedPos[N];//排序后
int lfnum[20][N];//元素所在区间的当前位置进入左孩子的元素的个数
int val[20][N];//记录第 k 层当前位置的元素的值
bool cmp(const int &x,const int &y) {
    return arr[x]<arr[y];
}
void build(int l,int r,int d) {
    if(l==r) return;
    int mid=(l+r)>>1,p=0;
    for(int i=l; i<=r; i++) {
        if(val[d][i]<=mid) {
            val[d+1][l+p]=val[d][i];
            lfnum[d][i]=++p;
        } else {
            lfnum[d][i]=p;
            val[d+1][mid+i+1-l-p]=val[d][i];
        }
    }
    build(l,mid,d+1);
    build(mid+1,r,d+1);
}
//求区间[s,e]第 k 大的元素
int query(int s,int e,int k,int l=1,int r=n,int d=0) {
    if(l==r) return l;
    int mid=(l+r)>>1,ss,ee;
    ss=(s==l?0:lfnum[d][s-1]);
    ee=lfnum[d][e];
    if(ee-ss>=k) return query(l+ss,l+ee-1,k,l,mid,d+1);
    return
query(mid+1+(s-l-ss),mid+1+(e-l-ee),k-(ee-ss),mid+1,r,d+1);
}
int main() {
    int cas=0,m,l,r,k;
    while(scanf("%d%d",&n,&m)!=EOF) {
        for(int i=1; i<=n; i++){
            scanf("%d",arr+i),sortedPos[i]=i;
        }
        sort(sortedPos+1,sortedPos+n+1,cmp);
        for(int i=1; i<=n; i++){
            val[0][sortedPos[i]]=i;
        }
        build(1,n,0);
        while(m--) {
            scanf("%d%d%d",&l,&r,&k);
            printf("%d\n",arr[sortedPos[query(l,r,k)]]);
        }
    }
}
```

左偏树

左偏树(Leftist Tree)是一种可并堆的实现。

```
const int MAXN = 100010;

struct Node {
    int key, dist, lc, rc;
    void init(int _dist = 0, int _key = 0) {
        key = _key;
        dist = _dist;
        lc = rc = 0;
    }

} nodes[MAXN];
int memeryNode[MAXN];
int allocNode,allocMemery;
```

```
void initNode() {
	nodes[0].init(-1);//0 作为 NULL 节点
	allocNode = 1;
	allocMemery = 0;
}

int newNode(int x) {
	int tmp;
	if(allocMemery) {
		tmp = memeryNode[--allocMemery];
	} else {
		tmp = allocNode++;
	}
	nodes[tmp].init(0, x);
	return tmp;
}

void deleteNode(int A) {
	memeryNode[allocMemery++] = A;
}

int merge(int A, int B) {
	if (A != 0 && B != 0) {
		if (nodes[A].key < nodes[B].key) {
			swap(A, B);
		}
		nodes[A].rc = merge(nodes[A].rc, B);
		int &lc = nodes[A].lc;
		int &rc = nodes[A].rc;
		if (nodes[lc].dist < nodes[rc].dist) {
			swap(lc, rc);
		}
		nodes[A].dist = nodes[rc].dist + 1;
	} else {
		A = A == 0 ? B : A;
	}
	return A;
}


int pop(int A) {
	int t = merge(nodes[A].lc, nodes[A].rc);
	deleteNode(A);
	return t;
}

void insert(int v) {
	merge(0, newNode(v));
}
```

DLX

双向十字链表用 LRUD 来记录，LR 来记录左右方向的双向链表，UD 来记录上下方向的双向链表。

head 指向总的头指针，head 通过 LR 来贯穿的列指针头。
LRUD 的前 m 个一般作为其列指针头的地址。
rowHead[x]是指向其列指针头的地址。
colNum[x]录列链表中结点的总数。
selectRow[x]用来记录搜索结果。
col[x]代表 x 的列数
row[x]代表 x 的行数

```
//一般需要使用 A*或 IDA*优化。
//以 exact cover problem 的代码为例子
const int N = 1005;
const int M = 1005;
const int maxn = N*M;
int R[maxn], L[maxn], U[maxn], D[maxn];
int col[maxn], row[maxn];
int rowHead[M], selectRow[N],colNum[N];
int size,n,m;
bool flag;
//初始化
void init() {
	memset(rowHead,-1,sizeof(rowHead));
	for(int i=0; i<=m; i++) {
		colNum[i]=0;

		D[i]=U[i]=i;
		L[i+1]=i;
		R[i]=i+1;

		row[i] = 0;
		col[i] = i;
	}
	R[m]=0;
	size=m+1;
}
```

```
//插入一个点
void insert(int r,int c) {
	colNum[c]++;
	U[size]=U[c];
	D[size]=c;
	D[U[size]]=size;
	U[D[size]]=size;

	if(rowHead[r]==-1) {
		rowHead[r]=L[size]=R[size]=size;
	} else {
		L[size]=L[rowHead[r]];
		R[size]=rowHead[r];
		R[L[size]]=size;
		L[R[size]]=size;
	}
	row[size] = r;
	col[size] = c;
	size++;
}
//删除一列
void remove(int const& c) { //删除列
	int i,j;
	L[R[c]]=L[c];
	R[L[c]]=R[c];
	for(i=D[c]; i!=c; i=D[i]) {
		for(j=R[i]; j!=i; j=R[j]) {
			U[D[j]]=U[j],D[U[j]]=D[j];
			colNum[col[j]]--;
		}
	}
}
//恢复一列
void resume(int c) {
	int i,j;
	for(i=U[c]; i!=c; i=U[i]) {
		for(j=L[i]; j!=i; j=L[j]) {
			U[D[j]]=j;
			D[U[j]]=j;
			colNum[col[j]]++;
		}
	}
	L[R[c]]=c;
	R[L[c]]=c;
}
//搜索
void dfs(int k){
	int i,j,Min,c;
	if(R[0] == 0) {
		flag = true;//标记有解
		printf("%d",k);//输出有 k 行
		for(i=0; i<k; i++){
			printf(" %d",selectRow[i]);
		}
		printf("\n");
		return;
	}

	//select a col that has min 1
	for(Min=N,i=R[0]; i; i=R[i]){
		if(colNum[i]<Min){
			Min=colNum[i],c=i;
		}
	}
	remove(c);//删除该列

	//select a row in the delete col
	for(i=D[c]; i!=c; i=D[i]) {
		selectRow[k] = row[i];
		for(j=R[i]; j!=i; j=R[j]){
			remove(col[j]);
		}
		dfs(k+1);
		if(flag) return;//只要一组解
		for(j=L[i]; j!=i; j=L[j]){
			resume(col[j]);
		}
	}
	resume(c);

}
int main() {
	int i,j,num;
	while(scanf("%d%d",&n,&m)!=EOF) {
		init();
		for(i=1; i<=n; i++) {
			scanf("%d",&num);
			while(num--) {
				scanf("%d",&j);
				insert(i,j);//向第 i 行第 j 列插入 1
			}
		}
		flag = false;
```

```
            dfs(0);
            if(!flag){
                printf("NO\n");
            }
        }
        return 0;
}
```

AC 自动机

常用与解决多模式匹配问题。
例如：给出 n 个单词，再给出一段包含 m 个字符的文章，让你找出有多少个单词在文章里出现过。

一般由字典树和 KMP 结合而成。

传统字典树

```
const int kind = 26;
struct Tire {
    int fail;           //失败指针
    int next[kind];
    //Tire 每个节点的个子节点（最多个字母）
    int count;    //是否为该单词的最后一个节点
    void init() {               //构造函数初始化
        fail = 0;
        count = 0;
        memset(next,-1,sizeof(next));
    }
} tire[5000001];   //队列，方便用于 bfs 构造失败指针
int size = 0;
queue<int>que;
void init() {
    tire[0].init();
    size = 1;
    while(!que.empty())que.pop();
}

void insert(char *str) {
    int p = 0, i=0,index;
    while(str[i]) {
        index=str[i]-'a';
        if(tire[p].next[index] == -1) {
            tire[size].init();
            tire[p].next[index]= size++;
        }
        p = tire[p].next[index];
        i++;
    }
    tire[p].count++;          //在单词的最后一个节点 count+1，代表一个单词
}

void build_ac_automation() {
    int i,pre,pre_fail,child;

    tire[0].fail = -1;
    que.push(0);

    while(!que.empty()) {
        pre = que.front();
        que.pop();
        for(i=0; i<kind; i++) {
            if(tire[pre].next[i] != -1) {
                child = tire[pre].next[i] ;
                pre_fail = tire[pre].fail;
                while(pre_fail != -1) {
                    if(tire[pre_fail].next[i] != -1) {
                tire[ child ].fail = tire[pre_fail].next[i];
                        break;
                    }
                    pre_fail = tire[pre_fail].fail;
                }
                if(pre_fail == -1) {
                    tire[ child ].fail = 0;
                }
                que.push(child);
            }
        }
    }
}
int query(char* str) {
    int i=0,cnt=0,index,tmp;
    int p = 0;
    while(str[i]) {
        index=str[i]-'a';
while(tire[p].next[index] == -1 && p != 0) p=tire[p].fail;
        p = tire[p].next[index];
        tmp = p = (p==-1 ? 0:p);
        while(tmp != 0 && tire[tmp].count != -1) {
            cnt += tire[tmp].count;
            tire[tmp].count = -1;
            tmp = tire[tmp].fail;
```

```
            }
            i++;
        }
        return cnt;
}
int main() {
        char keyword[51];       //输入的单词
        char str[1000005];      //模式串
        int n;
        scanf("%d",&n);
        init();
        while(n--) {
                scanf("%s",keyword);
                insert(keyword);
        }
        build_ac_automation();
        scanf("%s",str);
        printf("%d\n",query(str));
        return 0;
}
```

数组式字典树

```
const int kind = 26,N=1000001;

int tire[N][kind],fail[N],word[N];
int size = 0;
queue<int>que;

void init() {
        memset(word,0,sizeof(word));
        memset(tire[0],0,sizeof(tire[0]));
        while(!que.empty())que.pop();
        fail[0] = 0;
        size=1;
}

void insert(char *str, int val) {
        int p = 0, i;
        for(; *str; str++) {
                i = *str - 'a';
                if(!tire[p][i]) {
                        memset(tire[size],0,sizeof(tire[size]));
                        word[size]=0;
                        tire[p][i]= size++;
                }
                p = tire[p][i];
        }
        word[p] += val;
}

void build_ac_automation() {
        int i,pre;

        for(i=0; i< kind; i++) {
                if(tire[0][i]) {
                        fail[tire[0][i]]=0;
                        que.push(tire[0][i]);
                }
        }
        while(!que.empty()) {
                pre = que.front();
                que.pop();
                for(i=0; i<kind; i++) {
                        if(tire[pre][i]) {
                                que.push(tire[pre][i]);
                                fail[tire[pre][i]] = tire[fail[pre]][i];
                        } else {
                                tire[pre][i] = tire[fail[pre]][i];
                        }
                }
        }
}
int query(char* str) {
        int cnt=0,i,tmp;
        int p = 0;
        for(; *str; str++) {
                i= *str-'a';
                while(tire[p][i] == 0 && p) p=fail[p];
                p = tire[p][i];
                tmp = p;
                while(tmp) {
                        cnt += word[tmp];
                        word[tmp] = 0;
                        tmp = fail[tmp];
                }
        }
        return cnt;
}
int main() {
        char keyword[51];       //输入的单词
        char str[1000005];      //模式串
        int n;
        scanf("%d",&n);
```

```
    init();
    while(n--) {
        scanf("%s",keyword);
        insert(keyword,1);
    }
    build_ac_automation();
    scanf("%s",str);
    printf("%d\n",query(str));
    return 0;
}
```

最少交换次数

只能交换相邻的数

给出一组数，通过不断交换两个相邻的数，可以使这组数按非递减顺序排列。问，最小的交换次数是多少？
思路：归并排序，其实答案就是逆序数的个数。

只能交换相邻的区间

使用 IDA*搜索。

离散化

```
/*
边界为 1
内部为 2
当边界无效时为 3

矩阵的周长就是偶数行奇数列的 1 的宽度 和 奇数列偶数行的高度
矩阵的面积就是里面 2 的面积计算方法:(r-l)*(d-u)
/*
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  1  1  1  0
 0  0  1  2  1  0
 0  0  1  1  1  0
 0  0  0  0  0  0
*/
struct B {
    double x1, y1, x2, y2;
    void init() {
        scanf("%lf%lf%lf%lf", &x1, &y1, &x2, &y2);
        if(x1 > x2) swap(x1, x2);
        if(y1 > y2) swap(y1, y2);
    };
};
int const inf = 0x3f3f3f3f, maxn =   20100;
int x1, y1, x2, y2, n, mx, my;
int m[maxn][maxn];
set<double> x, y;
set<double>::iterator si;
map<double, int> hx, hy;
map<int, double> hhx, hhy;
B b[maxn];

double getS(){
    double ans = 0;
    for(int i=3;i<mx;i+=2)
        for(int j=3;j<my;j+=2)
            if(m[i][j]==2)
                ans                                    +=
(hhx[i+1]-hhx[i-1])*(hhy[j+1]-hhy[j-1]);
    return ans;
}

double getL(){
    double   ans=0;
    for(int i=2;i<mx;i+=2)
        for(int j=3;j<my;j+=2)
            if(m[i][j]==1)
                ans += hhy[j+1]-hhy[j-1];

    for(int i=3;i<mx;i+=2)
        for(int j=2;j<my;j+=2)
            if(m[i][j]==1)
                ans += hhx[i+1]-hhx[i-1];
    return ans;
}

int main() {
    const bool debug = false;
    int i, j, k,cs=1;
    while(~scanf("%d", &n),n) {

        x.clear();y.clear();
        for(i = 0; i < n; i++) {
            b[i].init();
            x.insert(b[i].x1);x.insert(b[i].x2);
            y.insert(b[i].y1);y.insert(b[i].y2);
        }
        hx.clear();hy.clear();
```

```
        //把地图扩大二倍后，矩阵内部就可以被填充，矩阵边界就可以走了
        //对 x 离散化
    for(si=x.begin(),mx=2;si!=x.end();hx[*si]=mx,hhx[mx] = *si, si++,mx+=2) ;

        //对 y 离散化
        for(si=y.begin(),my=2;si!=y.end();hy[*si]=my,hhy[my] = *si, si++,my+=2);
        //初始化矩阵
        for(i = 0; i < mx; ++i){
            fill(m[i], m[i] + my, 0);
        }
        //填充矩阵，填充为 1
        for(i = 0; i < n; i++) {
            int xuper = hx[b[i].x2];
            int yuper = hy[b[i].y2];

            //填充上下边界
            for(j = hx[b[i].x1]; j <= xuper; j++){
    if(m[j][hy[b[i].y1]]==0)m[j][hy[b[i].y1]]=1;
    if(m[j][hy[b[i].y2]]==0)m[j][hy[b[i].y2]]=1;
            }
            //填充左右边界
            for(k = hy[b[i].y1]; k <= yuper; k++){
    if(m[hx[b[i].x1]][k]==0)m[hx[b[i].x1]][k]=1;
    if(m[hx[b[i].x2]][k]==0)m[hx[b[i].x2]][k]=1;
            }
            //填充矩阵内部
            for(j = hx[b[i].x1] + 1; j < xuper; j++) {
            for(k = hy[b[i].y1] + 1; k < yuper; k++) {
                    m[j][k]=2;
                }
            }
        }
        //此处已不属于周长，标记为 3
        for(i=1;i<mx-1;i++){
            for(j=1;j<my-1;j++){
                if(m[i][j] == 1 && m[i-1][j]&&m[i][j-1]&&m[i+1][j]&&m[i][j+1])
                    m[i][j]=3;
            }
        }
        double S=getS();
printf("Test case #%d\nTotal    area: %.2f\n\n",cs++,S);
        //int L = (int)getL();
        //printf("%d\n",L);
    }
    return 0;
}
```

DP

RMQ 问题

题目描述

RMQ 问题是求给定区间中的最值问题。
当然，最简单的算法是 O(n)的，但是对于查询次数很多（设置多大 100 万次），O(n)的算法效率不够。可以用线段树将算法优化到 O（logn)（在线段树中保存线段的最值）。

不过，Sparse_Table 算法才是最好的：它可以在 O(nlogn)的预处理以后实现 O(1)的查询效率。

思路分析：

下面把 Sparse Table 算法分成预处理和查询两部分来说明(以求最小值为例)。

预处理:

预处理使用 DP 的思想，f(i, j)表示[i, i+2^j - 1]区间中的最小值，我们可以开辟一个数组专门来保存 f(i, j)的值。
例如，f(0, 0)表示[0,0]之间的最小值,就是 num[0], f(0, 2)表示[0, 3]之间的最小值, f(2, 4)表示[2, 17]之间的最小值
注意，因为 f(i, j)可以由 f(i, j - 1)和 f(i+2^(j-1), j-1)导出,而递推的初值(所有的 f(i, 0) = i)都是已知的
所以我们可以采用自底向上的算法递推地给出所有符合条件的 f(i, j)的值。

查询:

假设要查询从 m 到 n 这一段的最小值，那么我们先求出一个最大的 k，使得 k 满足 2^k <= (n - m + 1).
于是我们就可以把[m, n]分成两个(部分重叠的)长度

为 2^k 的区间: [m, m+2^k-1], [n-2^k+1, n];
而我们之前已经求出了 f(m, k)为[m, m+2^k-1]的最小值, f(n-2^k+1, k)为[n-2^k+1, n]的最小值
我们只要返回其中更小的那个, 就是我们想要的答案, 这个算法的时间复杂度是 O(1)的.
例如, rmq(0, 11) = min(f(0, 3), f(4, 3))

样例代码

```
int st[20][N], ln[N], val[N];
void initrmq(int n) {
    int i, j, k, sk;
    ln[0] = ln[1] = 0;
    for (i = 0; i < n; i++) st[0][i] = val[i];
    for (i = 1, k = 2; k < n; i++, k <<= 1) {
        for (j = 0, sk = (k >> 1); j < n; ++j, ++sk) {
            st[i][j] = st[i-1][j];
            if (sk < n && st[i][j] > st[i-1][sk])
                st[i][j] = st[i-1][sk];
        }
        for (j=(k>>1)+1; j <= k; ++j) ln[j] = ln[k>>1] + 1;
    }
    for (j=(k>>1)+1; j <= k; ++j) ln[j] = ln[k>>1] + 1;
}
int query(int x, int y) { // min of { val[x] ... val[y] }
    int bl = ln[y - x + 1];
    return min(st[bl][x], st[bl][y-(1<<bl)+1]);
}
```

LCA 问题

对于该问题，最容易想到的算法是分别从节点 u 和 v 回溯到根节点，获取 u 和 v 到根节点的路径 P1，P2，其中 P1 和 P2 可以看成两条单链表，这就转换成常见的一道面试题:【判断两个单链表是否相交，如果相交，给出相交的第一个点。】。该算法总的复杂度是 O（n）（其中 n 是树节点个数）。
本节介绍了两种比较高效的算法解决这个问题，其中一个是在线算法（DFS+ST），另一个是离线算法（Tarjan 算法）。

在线算法 DFS+ST

(思想是：将树看成一个无向图，u 和 v 的公共祖先一定在 u 与 v 之间的最短路径上)：
（1）DFS：从树 T 的根开始，进行深度优先遍历（将树 T 看成一个无向图），并记录下每次到达的顶点。第一个的结点是 root(T)，每经过一条边都记录它的端点。由于每条边恰好经过 2 次，因此一共记录了 2n-1 个结点，用 E[1, ... , 2n-1]来表示。
（2）计算 R：用 R[i]表示 E 数组中第一个值为 i 的元素下标，即如果 R[u] < R[v]时，DFS 访问的顺序是 E[R[u], R[u]+1, …, R[v]]。虽然其中包含 u 的后代，但深度最小的还是 u 与 v 的公共祖先。
（3）RMQ：当 R[u] ≥ R[v]时，LCA[T, u, v] = RMQ(L, R[v], R[u])；否则 LCA[T, u, v] = RMQ(L, R[u], R[v])，计算 RMQ。
由于 RMQ 中使用的 ST 算法是在线算法，所以这个算法也是在线算法。

代码实现

```
const int N = 10001; // 1<<20;
int pnt[N], next[N], head[N]; // 邻接表
int e; // 边数
bool visited[N]; // 初始为 0，从根遍历
int id;
int dep[2*N+1], E[2*N+1], R[N]; // dep:dfs 遍历节点深度, E:dfs 序列, R:第一次被遍历的下标
void DFS(int u, int d);
int d[20], st[2*N+1][20];
int n;
void InitRMQ(const int &id) {
    int i, j;
    for( d[0]=1, i=1; i < 20; ++i ) d[i] = 2*d[i-1];
    for( i=0; i < id; ++i ) st[i][0] = i;
    int k = int( log(double(n))/log(2.0) ) + 1;
    for( j=1; j < k; ++j )
        for( i=0; i < id; ++i ) {
            if( i+d[j-1]-1 < id ) {
                st[i][j] = dep[ st[i][j-1] ] > dep[ st[i+d[j-1]][j-1] ] ? st[i+d[j-1]][j-1] : st[i][j-1];
            } else break; // st[i][j] = st[i][j-1];
        }
}
int Query(int x, int y) {
    int k; // x, y 均为下标:0...n-1
    k = int( log(double(y-x+1))/log(2.0) );
    return dep[ st[x][k] ] > dep[ st[y-d[k]+1][k] ] ? st[y-d[k]+1][k] : st[x][k];
}
```

```
void Answer(void) {
	int i, Q;
	scanf("%d", &Q);
	for( i=0; i < Q; ++i ) {
		int x, y;
		scanf("%d%d", &x, &y); //  查询 x,y 的 LCA
		x = R[x];
		y = R[y];
		if( x > y ) {
			int tmp = x;
			x = y;
			y = tmp;
		}
		printf("%d\n", E[ Query(x, y) ]);
	}
}
void DFS(int u, int d) {
	visited[u] = 1;
	R[u] = id;
	E[id] = u;
	dep[id++] = d;
	for( int i=head[u]; i != -1; i=next[i] )
		if( visited[ pnt[i] ] == 0 ) {
			DFS(pnt[i], d+1);
			E[id] = u;
			dep[id++] = d;
		}
}
```

离线算法（Tarjan 算法）

所谓离线算法，是指首先读入所有的询问（求一次 LCA 叫做一次询问），然后重新组织查询处理顺序以便得到更高效的处理方法。Tarjan 算法是一个常见的用于解决 LCA 问题的离线算法，它结合了深度优先遍历和并查集，整个算法为线性处理时间。

Tarjan 算法是基于并查集的，利用并查集优越的时空复杂度，可以实现 LCA 问题的 O(n+Q)算法，这里 Q 表示询问 的次数。

同上一个算法一样，Tarjan 算法也要用到深度优先搜索，算法大体流程如下：对于新搜索到的一个结点，首先创建由这个结点构成的集合，再对当前结点的每一个子树进行搜索，每搜索完一棵子树，则可确定子树内的 LCA 询问都已解决。其他的 LCA 询问的结果必然在这个子树之外，这时把子树所形成的集合与当前结点的集合合并，并将当前结点设为这个集合的祖先。之后继续搜索下一棵子树，直到当前结点的所有子树搜索完。这时把当前结点也设为已被检查过的，同时可以处理有关当前结点的 LCA 询问，如果有一个从当前结点到结点 v 的询问，且 v 已被检查过，则由于进行的是深度优先搜索，当前结点与 v 的最近公共祖先一定还没有被检查，而这个最近公共祖先的包涵 v 的子树一定已经搜索过了，那么这个最近公共祖先一定是 v 所在集合的祖先。

代码实现

```
int id[N];//初始化-1
int lcs[N][N],
int g[N][N];//邻接矩阵
int get(int i) {
	if (id[i] == i) return i;
	return id[i] = get(id[i]);
}
void unin(int i, int j) {
	id[get(i)] = get(j);
}
void dfs(int rt, int n) {
	int i;
	id[rt] = rt;
	for (i = 0; i < n; ++i) if (g[rt][i] && -1 == id[i]) {
			dfs(i, n);
			unin(i, rt);
		}
	for (i = 0; i < n; ++i) if (-1 != id[i])
			lcs[rt][i] = lcs[i][rt] = get(i);
}
```

矩阵相乘 DP

求矩阵状态

```
bool state[513];
int _map[513];
int _map_num;
int str[513][513];
int _val_map[513];

//第 bit 位置为 1 val,val 可以是 0 或 1，bit 是 1~8
void setBit(int& now,int bit,int val = 1) {
	bit--;
	if(val == 1) {
```

```
            now |= (1<<bit);
        } else {
            now &= ~(1<<bit);
        }
}
//得到第 bit 位的值
int getBit(int now,int bit) {
    bit--;
    return (now>>bit)&1;
}
//输出 c 的二进制
void outputState(int c) {
    printf(",\"");
    for(int i=8; i; i--) {
        printf("%d",getBit(c,i));
    }
    printf("\"\n");
}
//添加状态 c
int addState(int c) {
    if(state[c] == false) {
        state[c] = true;
        _map[_map_num] = c;
        _val_map[c] = _map_num;
        _map_num++;
    }
    return _val_map[c];
}
//判断 now 是否全 1
bool isPutAll(int now) {
    return now == 255;
}
//深搜得到状态
void dfs(int lev,int now,int next) {
    int nextState,i;

    if(isPutAll(now)) {
        nextState = addState(next);
        str[lev][nextState]++;
        return ;
    }

    //视情况修改
    if(getBit(now,8) == 0 && getBit(now,1) == 0) {
        setBit(now,8,1);
        setBit(now,1,1);
        dfs(lev,now,next);
        setBit(now,8,0);
        setBit(now,1,0);
    }

    for(i=8; i>0; i--) {
        if(getBit(now,i) == 0) {
            setBit(now,i,1);
            setBit(next,i,1);
            dfs(lev,now,next);
            setBit(now,i,0);
            setBit(next,i,0);
            break;
        }
    }
    for(i=8; i>1; i--) {
        if(getBit(now,i) == 0) {
            if(getBit(now,i-1) == 0) {
                setBit(now,i,1);
                setBit(now,i-1,1);
                dfs(lev,now,next);
                setBit(now,i,0);
                setBit(now,i-1,0);
            }
            break;
        }
    }
}
//生成状态
void bornState() {

    memset(state,false,sizeof(state));
    _map_num = 0;
    memset(str,0,sizeof(str));

    addState(0);

    for(int i=0; i<_map_num; i++) {
        dfs(i,_map[i],0);
    }
}
```

使用矩阵幂求答案

```
    bornState();
    sz = _map_num;
    Matrix matrix,ans;
```

```
int n,m;
matrix.init(str);//矩阵需要添加这个函数
while(scanf("%d%d",&n,&m),n) {
    MOD = m;
    ans = matrix.pow(n);
    printf("%d\n",ans.a[0][0]);
}
```

DFA+DP

区间个数询问

对于一个数字，首先把这个数字存在数组中

```
typedef  long long LL;
int str[100];
LL _sum[100][30][30];
int len;
int x,y;
LL dfs(int pos, int x_num, int y_num, bool yes) {
    if(pos < 0) {//判断是否结束
        return x_num == x && y_num == y;
    }
    if(x_num > x || y_num > y) {
        return 0;//判断是否已经不满足条件
    }
    //判断是否已经是 999 且已经计算过。
    if(yes && _sum[pos][x_num][y_num] != -1) {
        return _sum[pos][x_num][y_num];
    }
    //没计算过，开始计算
    LL ans = 0;
    int _max = yes ? 9 : str[pos];
    for(int i=0; i<=_max; i++) {
        ans += dfs(pos-1, x_num + (i == 4),
y_num + (i == 7), yes || i<str[pos]);
    }
    //保存计算过的
    if(yes) {
        _sum[pos][x_num][y_num] = ans;
    }

    return ans;
}
//查询，一般是用[0, val]
LL query(LL val) {
    if(val < 0) {
        return 0;
    }
    len = 0;
    if(val == 0) {
        str[len++] = val;
    } else {
        while(val) {
            str[len++] = val%10;
            val /= 10;
        }
    }
    return dfs(len-1,0,0,false);
}
```

区间内[left,right]第 k 个满足条件的

二分查找

```
LL query(LL left, LL right,  LL k) {

    LL ans_num =  query(left-1) + k;
    LL mid, mid_num;
    while(left < right) {
        mid = (left + right)/2;
        mid_num = query(mid);
        if(mid_num == ans_num){
            right = mid;
        }else if(mid_num > ans_num) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

DFA 查找

按位查找，从最高位开始确定应该是那个数字

调用前 ans = 0;

调用 dfs(len-1,0,0,false,query(left-1) + k);

```
void dfs(int pos, int x_num, int y_num, bool yes, LL k) {
    if(pos < 0)return ;
```

```
    int _max = yes ? 9 : str[pos];
    LL tmp;
    for(int i=0; i<=_max; i++) {
tmp = dfs(pos-1, x_num+(i == 4), y_num + (i == 7), yes || i<str[pos]);
        if(tmp>=k){
            ans = ans*10+i;
            dfs(pos-1,x_num + (i == 4), y_num + (i == 7), yes || i<str[pos],k);
            return;
        }else{
            k -= tmp;
        }
    }
}
```

搜索

回文串

Manacher 算法

主要用于求解最长回文子串。

这个算法有一个很巧妙的地方，它把奇数的回文串和偶数的回文串统一起来考虑了。
这个算法还有一个很好的地方就是充分利用了字符匹配的特殊性，避免了大量不必要的重复匹配。

```
const int MAX = 110003 << 2;
char oldstr[MAX];//原字符串
char str[MAX];
int p[MAX];//表示以 i 为中心的回文半径，
/*
p[i]-1 刚好是原字符串以第 i 个为中心的回文串长度。
*/

void Manacher(int n) {
    int mx=0;
    int id;//最长影响串的位置;
    p[0]=0;

    for(int i = 1; i < n; i++) {
        p[i]=1;//至少是 1
        if(mx>i) {
            p[i] = p[2 * id - i];
            if(mx - i < p[i]) p[i] = mx - i;
        }

        //向两端配匹
        while(str[i - p[i]] == str[i + p[i]]) p[i]++;
        if(i + p[i] > mx) {
            mx = i + p[i];
            id = i;
        }
    }
}

/*
预处理字符串
*/
int pre(char head='$', char middle='#', char end = '?') {
    int n=0;
    str[n++]=head;
    str[n++]=middle;
    for(int i = 0; oldstr[i]; i++) {
        str[n++] = oldstr[i];
        str[n++] = middle;

    }
    str[n]=end;
    return n;
}

int main() {
    int n;
    while(scanf("%s", oldstr) != EOF) {
        n = pre();
        Manacher(n);
        int ans=0;
        for(int i = 1; i < n; i++) {
            if(p[i] > ans) ans = p[i];

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

迭代加深搜索

对于一般的搜索，复杂度是 O(2^n)的复杂度。

对于不知道比较好的算法时，只有进行暴力搜索了。
但是 DFS 可能进去出不来，对于 BFS 又可能爆栈。
这是就要进行迭代搜索了。
每当加深一层深度时，次层的搜索的时间可以忽略不计了，因为相差一个数量级。

IDA*

这里以一个例子来讲解 IDA*.
问题：n 个数互不相同，可以对相邻的连续区间进行交换，最终使 n 个数达到升序。求最少交换次数。
这里假设是 1 到 n 的 n 个数。不是话可以进行映射。

```
int n,str[20], _maxDepth;
int hfunc() {//估计函数：还有多少个断点
    int depth = 0;
    for(int i=1; i<n; i++) {
        if(str[i] != str[i-1]+1) {
            depth++;
        }
    }
    return depth;
}
//交换区间
void move(int left, int mid, int right) {
    int i, j, tmp[20];;
    for(i=mid+1,j=0; i<=right; i++,j++) {
        tmp[j] = str[i];
    }
    for(i=left; i<=mid; i++,j++) {
        tmp[j] = str[i];
    }
    for(i=left,j=0; i<=right; i++,j++) {
        str[i] = tmp[j];
    }
}
//迭代加深搜索
//三个 while 是我加的优化，我们只能从断点开始划分交换区间，一个连续区间分开后显然答案不忧。
//一次交换最优情况下可以减少三个断点。
int dfs(int depth) {
    int left,mid,right,h;
    for(left=0; left<n-1; left++) {
while(left && str[left] == str[left-1]+1)left++;
        for(mid = left; mid<n-1; mid++) {
while(mid<n-1 && (str[mid]+1 == str[mid+1]))mid++;
            for(right = mid+1; right < n; right++) {
while(right<n-1&&(str[right]+1 == str[right+1]))right++;
                move(left, mid,right);
                if((h = hfunc()) == 0)return 1;
                if(depth*3 + h <= _maxDepth*3) {
                    if(dfs(depth+1)) {
                        return 1;
                    }
                }
                move(left,left + (right-mid-1),right);
            }

        }

    }
    return 0;
}

int main() {
    int cas,i;
    scanf("%d",&n);
    for(i=0; i<n; i++) {
        scanf("%d",&str[i]);
    }
    _maxDepth = (hfunc()+2)/3;
    if(_maxDepth) {
        while(dfs(1) == 0) {
            _maxDepth++;
        }
    }
    printf("%d\n",_maxDepth);
return 0;
}
```

STL

bitset

to_ulong Convert to unsigned long integer
to_string Convert to string
count Count bits set
size Return size
test Return bit value
any Test if any bit is set
none Test if no bit is set
set (int pos） Set bits
reset（int pos） Reset bits
flip Flip bits

图论

网络流

```
const int N = 100;
const int E = 1000;
#define typec int // type of cost
const typec inf = 0x3f3f3f3f; // max of cost
struct edge {
    int x, y, nxt;
    typec c;
} bf[E];
int ne, head[N], cur[N], ps[N], dep[N];
void init(){
    ne = 2;
    memset(head,0,sizeof(head));
}
void addedge(int x, int y, typec c) {
//无向图需调用两次 addedge
    // add an arc(x -> y, c); vertex: 0 ~ n-1;
    bf[ne].x = x;
    bf[ne].y = y;
    bf[ne].c = c;
    bf[ne].nxt = head[x];
    head[x] = ne++;
    bf[ne].x = y;
    bf[ne].y = x;
    bf[ne].c = 0;
    bf[ne].nxt = head[y];
    head[y] = ne++;
}
typec flow(int n, int s, int t) {
    typec tr, res = 0;
    int i, j, k, f, r, top;
    while (1) {
        memset(dep, -1, n * sizeof(int));
        for (f = dep[ps[0] = s] = 0, r = 1; f != r; )
            for (i = ps[f++], j = head[i]; j; j = bf[j].nxt)
{
                if (bf[j].c && -1 == dep[k = bf[j].y])
{
                    dep[k] = dep[i] + 1;
                    ps[r++] = k;
                    if (k == t) {
                        f = r;
                        break;
                    }
                }
            }
        if (-1 == dep[t]) break;
        memcpy(cur, head, n * sizeof(int));
        for (i = s, top = 0; ; ) {
            if (i == t) {
                for (k = 0, tr = inf; k < top; ++k)
                    if (bf[ps[k]].c < tr)
                        tr = bf[ps[f = k]].c;
                for (k = 0; k < top; ++k)
                    bf[ps[k]].c  -=  tr,  bf[ps[k]^1].c
+= tr;
                res += tr;
                i = bf[ps[top = f]].x;
            }
            for  (j=cur[i];  cur[i];  j  =  cur[i]  =
bf[cur[i]].nxt)
                if  (bf[j].c  &&  dep[i]+1  ==
dep[bf[j].y]) break;
            if (cur[i]) {
                ps[top++] = cur[i];
                i = bf[cur[i]].y;
            } else {
                if (0 == top) break;
                dep[i] = -1;
                i = bf[ps[--top]].x;
            }
        }
    }
    return res;
}
```

对于一个给定的度序列，看能不能形成一个简单无向图

Havel 算法的思想简单的说如下：

（1）对序列从大到小进行排序。

（2）设最大的度数为 t ，把最大的度数置 0，然后把最大度数后（不包括自己）的 t 个度数分别减 1（意思就是把度数最大的点与后几个点进行连接）

（3）如果序列中出现了负数，证明无法构成。如果序列全部变为 0，证明能构成，跳出循环。前两点不出现，就跳回第一步！

最大团

```
const int V = 1000;
int g[V][V], dp[V], stk[V][V], mx;
int dfs(int n, int ns, int dep) {
    if (0 == ns) {
        if (dep > mx) mx = dep;
        return 1;
    }
    int i, j, k, p, cnt;
    for (i = 0; i < ns; i++) {
        k = stk[dep][i];
        cnt = 0;
        if (dep + n - k <= mx) return 0;
        if (dep + dp[k] <= mx) return 0;
        for (j = i + 1; j < ns; j++) {
            p = stk[dep][j];
            if (g[k][p]) stk[dep + 1][cnt++] = p;
        }
        dfs(n, cnt, dep + 1);
    }
    return 1;
}
int clique(int n) {
    int i, j, ns;
    for (mx = 0, i = n - 1; i >= 0; i--) {
        for (ns = 0, j = i + 1; j < n; j++)
            if (g[i][j]) stk[1][ ns++ ] = j;
        dfs(n, ns, 1);
        dp[i] = mx;
    }
    return mx;
}
```

次小生成树

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
const int
MAXN=501,MAXM=MAXN*MAXN*4,INF=0x7FFFFFFF;
using namespace std;
struct edge {
    edge *next,*op;
    int t,c;
    bool mst;
}
ES[MAXM],*V[MAXN],*MST[MAXN],*CLS[MAXN];
int N,M,EC=-1,MinST,Ans,NMST;
int DM[MAXN],F[MAXN];
inline void addedge(edge **V,int a,int b,int c) {
    ES[++EC].next=V[a];
    V[a]=ES+EC;
    V[a]->t=b;
    V[a]->c=c;
    ES[++EC].next=V[b];
    V[b]=ES+EC;
    V[b]->t=a;
    V[b]->c=c;
    V[a]->op=V[b];
    V[b]->op=V[a];
    V[a]->mst=V[b]->mst=false;
}
void init() {
    int i,a,b,c;
    freopen("conf.in","r",stdin);
    freopen("conf.out","w",stdout);
    scanf("%d%d",&N,&M);
    for (i=1; i<=M; i++) {
        scanf("%d%d%d",&a,&b,&c);
        addedge(V,a,b,c);
    }
    Ans=INF;
}
void prim() {
    int i,j,Mini;
    for (i=1; i<=N; i++)
        DM[i]=INF;
    for (i=1; i;) {
        DM[i]=-INF;
        for (edge *e=V[i]; e; e=e->next) {
            if (e->c < DM[j=e->t]) {
                DM[j]=e->c;
                CLS[j]=e->op;
            }
        }
```

```
            Mini=INF;
            i=0;
            for (j=1; j<=N; j++)
                  if (DM[j]!=-INF && DM[j]<Mini) {
                        Mini=DM[j];
                        i=j;
                  }
      }
      for (i=2; i<=N; i++) {
            MinST+=CLS[i]->c;
            CLS[i]->mst=CLS[i]->op->mst=true;

addedge(MST,CLS[i]->t,CLS[i]->op->t,CLS[i]->c);
      }
}
void dfs(int i) {
      for (edge *e=MST[i]; e; e=e->next) {
            int j=e->t;
            if (F[j]==-INF) {
                  F[j]=F[i];
                  if (e->c > F[j])
                        F[j]=e->c;
                  dfs(j);
            }
      }
}
void smst() {
      int i,j;
      for (i=1; i<=N; i++) {
            for (j=1; j<=N; j++)
                  F[j]=-INF;
            F[i]++;
            dfs(i);
            for (edge *e=V[i]; e; e=e->next) {
                  if (!e->mst) {
                        NMST=MinST + e->c -F[e->t];
                        if (NMST < Ans)
                              Ans=NMST;
                  }
            }
      }
}
int main() {
      init();
      prim();
      smst();
      if (Ans==INF)
            Ans=-1;
      printf("Cost: %dnCost: %dn",MinST,Ans);
      return 0;
}
```

第 K 短路

使用 A*+djikstra 求第 k 短路。

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<queue>
#define MAXN 1001
using namespace std;
struct node {
      int p,g,h;
      bool operator < (node a) const {
            return a.g+a.h<g+h;
      }
};
struct node1 {
      int x,y,w,next;
} line[MAXN*100],line1[MAXN*100];



int n,m,i,link[MAXN],link1[MAXN],g[MAXN],s,e,k;
bool used[MAXN];
priority_queue<node> myqueue;
void djikstra() {
      int i,k,p;
      memset(used,0,sizeof(used));
      memset(g,0x7F,sizeof(g));
      g[e]=0;
      for (p=1; p<=n; p++) {
            k=0;
            for (i=1; i<=n; i++)
                  if (!used[i] && (!k || g[i]<g[k]))
                        k=i;
            used[k]=true;
            k=link1[k];
            while (k) {
                  if (g[line1[k].y]>g[line1[k].x]+line1[k].w)
                    g[line1[k].y]=g[line1[k].x]+line1[k].w;
                  k=line1[k].next;
            }
      }
```

```
        return ;
}
int Astar() {
        int t,times[MAXN];
        node h,temp;
        while (!myqueue.empty()) myqueue.pop();
        memset(times,0,sizeof(times));
        h.p=s;
        h.g=0;
        h.h=0;
        myqueue.push(h);
        while (!myqueue.empty()) {
                h=myqueue.top();
                myqueue.pop();
                times[h.p]++;
                if (times[h.p]==k && h.p==e) return h.h+h.g;
                if (times[h.p]>k) continue;
                t=link[h.p];
                while (t) {
                        temp.h=h.h+line[t].w;
                        temp.g=g[line[t].y];
                        temp.p=line[t].y;
                        myqueue.push(temp);
                        t=line[t].next;
                }
        }
        return -1;
}
int main() {
        scanf("%d%d",&n,&m);
        memset(link,0,sizeof(link));
        memset(link1,0,sizeof(link1));
        for (i=1; i<=m; i++) {

scanf("%d%d%d",&line[i].x,&line[i].y,&line[i].w);
                line[i].next=link[line[i].x];
                link[line[i].x]=i;
                line1[i].x=line[i].y;
                line1[i].y=line[i].x;
                line1[i].w=line[i].w;
                line1[i].next=link1[line1[i].x];
                link1[line1[i].x]=i;
        }
        scanf("%d%d%d",&s,&e,&k);
        if (s==e) k++;
        djikstra();
        printf("%d\n",Astar());
        return 0;
}
```

查分约束

如果一个系统由 n 个变量和 m 个不等式组成，形如 Xj-Xi<=b*k（i，j 属于[1，n]，k 属于[1，m] ）， 这样的系统称之为差分约束系统。差分约束系统通常用于求关于一组变量的不等式组。

求解差分约束系统可以转化为图论中单源最短路问题。

Xj-Xi<=k

==> d[v]-d[u]<=w[u,v]

==> 所以转化为图论求解

也就是 if(d[v]-d[u] >w[u,v])那么 d[v]-d[u]<=w[u,v] 。

路径距离初始化 dis[i]=INF

再增加一个源点 s，源点到所有定点的距离为 0（ 添加源点的实质意义为默认另外一系列的不等式组 Xi-Xo<=0）,再对源点利用 spfa 算法。

注意几个问题：

1、当 0 没有被利用的时候，0 作为超级源点。当 0 已经被利用了，将 n+1（未被利用）置为超级源点。

2、对于 Xj-Xi=k 可以转换为 Xj-Xi<=k Xj-Xi>=k 来处理。

3、若要判断图中是否出现负环，可以利用深度优先搜索。以前利用 spfa 是这样的（head->****->tail），当 head 和 tail 之间所有点都遍历完了才轮得上 tail 这个点，这样的话我们无法判断图中有没有负环，我们可以这样改变一样遍历顺序，head->tail->***->head。当深度优先搜索过程中下次再次遇见 head（也就是 head 节点依然在标记栈中）时，则可判断图中含有负环，否则没有。

4、当图连通时，只需要对源点 spfa 一次；当图不连通时，对每个定点 spfa 一次。

5、对于 Xj-Xi<k or Xj-Xi>k ，差分约束系统只针对>= or <=， 所以我们还要进行巧妙转换编程大于等于小于等于。

二分匹配

Hungary（匈牙利）算法

```
const int N=601;

//行为上部，列为下部，挑最少的点，使得所有的边和挑的点相连。最小顶点覆盖。
int n1,n2;
//n1,n2 为二分图的顶点集,其中 x∈n1,y∈n2
bool _map[N][N],vis[N];
int link[N];
//link 记录 n2 中的点 y 在 n1 中所匹配的 x 点的编号

int find(int x) {
    int i;
    for(i=0; i<n2; i++) {
        if(_map[x][i]&&!vis[i]) {
//x->i 有边,且节点 i 未被搜索
            vis[i] = true;//标记节点已被搜索
//如果 i 不属于前一个匹配 M 或被 i 匹配到的节点可以寻找到增广路
            if(link[i]==-1 || find(link[i])) {
                link[i]=x;//更新
                return true;//匹配成功
            }
        }
    }
    return false;
}
int mach() {
    int ans = 0;
    memset(link, -1, sizeof(link));
    for (int i = 0; i < n1; i++) {
        memset(vis,false,sizeof(vis));
        if (find(i))ans++;
//  如果从第 i 个点找到了增光轨，总数加一
    }
    return ans;
}
```

Hopcroft－Karp 算法

Hopcroft－Karp 算法是匈牙利算法的改进，时间复杂度 0(E*V^1/2)(怎么证的？)，算法实质其实还是匈牙利算法求增广路，改进的地方是在深度搜索增广路前，先通过广度搜索，查找多条可以增广的路线，从而不再让 dfs“一意孤行”。其中算法用到了分层标记防止多条增广路重叠，这里是精髓

```
typedef long long ll;
#define inf (int)1e10
#define maxn 50005
vector<int>vec[maxn];
int headU[maxn],headV[maxn];
int du[maxn],dv[maxn];
int uN,vN;
bool bfs() {
    queue<int>q;
    int dis=inf;
    memset(du,0,sizeof(du));
    memset(dv,0,sizeof(dv));
    for(int i=1; i<=uN; i++)
        if(headU[i]==-1)q.push(i);
    while(!q.empty()) {
        int u=q.front();
        q.pop();
        if(du[u]>dis)break;
        for(int i=0; i<vec[u].size(); i++)
            if(!dv[vec[u][i]]) {
                dv[vec[u][i]]=du[u]+1;
if(headV[vec[u][i]]==-1)dis=dv[vec[u][i]];
                else {
    du[headV[vec[u][i]]]=dv[vec[u][i]]+1;
                    q.push(headV[vec[u][i]]);
                }
            }
    }
    return dis!=inf;
}
bool dfs(int u) {
    for(int i=0; i<vec[u].size(); i++)
        if(dv[vec[u][i]]==du[u]+1) {
            dv[vec[u][i]]=0;
if(headV[vec[u][i]]==-1||dfs(headV[vec[u][i]])) {
                headU[u]=vec[u][i];
                headV[vec[u][i]]=u;
                return 1;
            }
        }
    return 0;
}
int Hopcroft() {
```

```
    memset(headU,-1,sizeof(headU));
    memset(headV,-1,sizeof(headV));
    int ans=0;
    while(bfs())
        for(int i=1; i<=uN; i++)
            if(headU[i]==-1&&dfs(i))ans++;
    return ans;
}
int main() {
    int u,v,w;
    while(~scanf("%d%d%d",&u,&v,&w)) {
        for(int i=0; i<maxn; i++)vec[i].clear();
        uN=u;
        int tu,tv;
        while(w--) {
            scanf("%d%d",&tu,&tv);
            vec[tu].push_back(tv);
        }
        printf("%d\n",Hopcroft());
    }
    return 0;
}
```

最大权匹配的 KM 算法

```
typedef long long ll;
#define inf (int)1e10
#define maxn 150
int edge[maxn][maxn];//邻接矩阵
int du[maxn],dv[maxn];//可行顶标
int head[maxn];//匹配节点的父节点
bool visu[maxn],visv[maxn];//判断是否在交错树上
int uN;//匹配点的个数
int slack[maxn];//松弛数组
bool dfs(int u) {
    visu[u]=true;
    for(int v=0; v<uN; v++)
        if(!visv[v]) {
            int t=du[u]+dv[v]-edge[u][v];
            if(t==0) {
                visv[v]=true;
                if(head[v]==-1||dfs(head[v])) {
                    head[v]=u;
                    return true;
                }
            } else slack[v]=min(slack[v],t);
        }
    return false;
}
int KM() {
    memset(head,-1,sizeof(head));
    memset(du,0,sizeof(du));
    memset(dv,0,sizeof(dv));
    for(int u=0; u<uN; u++)
        for(int v=0; v<uN; v++)
            du[u]=max(du[u],edge[u][v]);
    for(int u=0; u<uN; u++) {
        for(int i=0; i<uN; i++)slack[i]=inf;
        while(true) {
            memset(visu,0,sizeof(visu));
            memset(visv,0,sizeof(visv));
            if(dfs(u))break;
            int ex=inf;
            for(int v=0; v<uN; v++)if(!visv[v])
                    ex=min(ex,slack[v]);
            for(int i=0; i<uN; i++) {
                if(visu[i])du[i]-=ex;
                if(visv[i])dv[i]+=ex;
                else slack[i]-=ex;

            }
        }
    }
    int ans=0;
    for(int u=0; u<uN; u++)
        ans+=edge[head[u]][u];
    return ans;
}
int main() {
    //read;
    while(~scanf("%d",&uN)) {
        int sum=0;
        for(int i=0; i<uN; i++)
            for(int j=0; j<uN; j++) {
                scanf("%d",&edge[i][j]);
                sum+=edge[i][j];
            }
        printf("%d\n",sum-KM());
    }
    return 0;
}
```

强连通分支算法

有向图 G=(V, E)的一个强连通分支就是一个最大的顶点子集 C，对于 C 中每对顶点(s, t)，有 s 和 t 是强连通的，并且 t 和 s 也是强连通的，即顶点 s 和 t 是互达的。

Kosaraju 算法

Kosaraju 算法的解释和实现都比较简单，为了找到强连通分支，首先对图 G 运行 DFS，计算出各顶点完成搜索的时间 f；然后计算图的逆图 GT，对逆图也进行 DFS 搜索，但是这里搜索时顶点的访问次序不是按照顶点标号的大小，而是按照各顶点 f 值由大到小的顺序；逆图 DFS 所得到的森林即对应连通区域

矩阵储存

```
const int MAXV = 1024;
int g[MAXV][MAXV], dfn[MAXV], num[MAXV], n, scc, cnt;
void dfs(int k) {
    num[k] = 1;
    for(int i=1; i<=n; i++)
        if(g[k][i] && !num[i])
            dfs(i);
    dfn[++cnt] = k;//记录第 cnt 个出栈的顶点为 k
}
void ndfs(int k) {
    num[k] = scc;
    for(int i=1; i<=n; i++)
        if(g[i][k] && !num[i])
            ndfs(i);
}
void kosaraju() {
    int i, j;
    for(i=1; i<=n; i++)
        if(!num[i])
            dfs(i);
    memset(num, 0, sizeof num);
    for(i=n; i>=1; i--)
        if(!num[dfn[i]]) {
            scc++;
            ndfs(dfn[i]);
        }
    cout<<"Found: "<<scc<<endl;
}
```

邻接表储存

```
const int MAXN = 100;
vector<int> adj[ MAXN ] ; //正向邻接表
vector<int> radj[ MAXN ] ; //反向邻接表
vector<int> ord; //后序访问顺序
int vis[MAXN],cnt,n;

void dfs ( int v ) {
    vis[v]=1;
    for (int i = 0, u, _size = adj[v].size(); i< _size; i++ )
        if ( !vis[u=adj[v][i]] )
            dfs(u);
    ord.push_back(v);
}

void ndfs ( int v ) {
    vis[v]=cnt;
    for (int i = 0, u, _size = radj[v].size(); i < _size; i++ )
        if ( !vis[u=radj[v][i]] )
            ndfs(u);
}

void kosaraju () {
    int i;
    memset(vis,0,sizeof(vis));
    ord.clear();

    for ( i=0 ; i<n ; i++ )
        if ( !vis[i] )
            dfs(i);

    memset(vis,0,sizeof(vis));
    for ( cnt=0,i=ord.size()-1 ; i>=0 ; i-- )
        if ( !vis[ord[i]] ) {
            ndfs(ord[i]);
            cnt++;
        }
}
```

Tarjan 算法

```
#define   M 5010      //题目中可能的最大点数
int STACK[M],top=0;    //Tarjan 算法中的栈
bool InStack[M];       //检查是否在栈中
int DFN[M];            //深度优先搜索访问次序
int Low[M];            //能追溯到的最早的次序
```

```
int ComponentNumber=0;     //有向图强连通分量个数
int Index=0;                    //索引号
vector <int> Edge[M];              //邻接表表示
vector <int> Component[M];     //获得强连通分量结果
int InComponent[M]; //记录每个点在第几号强连通分量里
int ComponentDegree[M]; //记录每个强连通分量的度
void Tarjan(int i) {
    int j;
    DFN[i]=Low[i]=Index++;
    InStack[i]=true;
    STACK[++top]=i;
    for (int e=0; e<Edge[i].size(); e++) {
        j=Edge[i][e];
        if (DFN[j]==-1) {
            Tarjan(j);
            Low[i]=min(Low[i],Low[j]);
        } else if (InStack[j])
            Low[i]=min(Low[i],DFN[j]);
    }
    if (DFN[i]==Low[i]) {
        ComponentNumber++;
        do {
            j=STACK[top--];
            InStack[j]=false;
Component[ComponentNumber].push_back(j);
            InComponent[j]=ComponentNumber;
        } while (j!=i);
    }
}
```

```
void solve(int N) {
    memset(STACK,-1,sizeof(STACK));
    memset(InStack,0,sizeof(InStack));
    memset(DFN,-1,sizeof(DFN));
    memset(Low,-1,sizeof(Low));
    for(int i=0; i<N; i++)
        if(DFN[i]==-1)
            Tarjan(i);
}
```

Gabow 算法

类似于 tarjan 算法

几何

点

```
struct Point_2d {
    double x,y;
};
```

两点间的距离

二维

```
double distance_2d(Point_2d &a, Point_2d &b) {
    return(sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y)));
}
```

三维

```
double distance_3d(Point_3d &a, Point_3d&b) {
return(sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y)+(a.z-b.z)*(a.z-b.z)));
}
```

三角形面积

海伦公式

```
double triangleArea(Point_2d &a,Point_2d &b,Point_2d &c) {
    double aa,bb,cc,p,s;
    ab=distance_2d(a,b);
    bc=distance_2d(b,c);
    ca=distance_2d(c,a);
    p=(ab+bc+ca)/2;
    s=sqrt(p*(p-ab)*(p-bc)*(p-ca));
    return s;
}
```

圆的储存

```
struct C {
    P mid;
    double r;
    C(const P &_mid, const double &_r)
        :mid(_mid), r(_r) {}
    C() {}
    bool operator == (const C &a) const {
        return mid == a.mid && sgn(r - a.r) == 0;
    }
    bool in(const C &a) const {
```

```
        return sgn(r + dist(mid, a.mid) - a.r) <= 0;
    }
    const C &input() {
        mid.input();
        scanf("%lf", &r);
        return *this;
    }
    const C &output() const {
  printf("P: %.12lf %.12lf R: %.12lf\n", mid.x, mid.y, r);
    }
};
```

圆相交 返回两个交点

```
bool circles_intersection(const C &a, const C &b, P &c1,
P &c2) {
    double dd = dist(a.mid, b.mid);
    if (sgn(dd - (a.r + b.r)) >= 0) {
        return false;
    }
    double l = (dd + (SQR(a.r) - SQR(b.r)) / dd) / 2;
    double h = sqrt(SQR(a.r) - SQR(l));
    c1 = a.mid + (b.mid - a.mid).trunc(l) + (b.mid -
a.mid).turn_left().trunc(h);
    c2 = a.mid + (b.mid - a.mid).trunc(l) + (b.mid -
a.mid).turn_right().trunc(h);
    return true;
}
```