**200和整数对之间的情缘**:给你一个有N个整型数字的序列A，整数对(i,j)满足Ai-Aj是200的整数倍

int n,arr[int(2e5)],temp,ans = 0;

int main()

{

    cin >> n;

    for (int i = 0; i < n; n--){

        cin >> temp;

        arr[temp%200]++; //余数相同的数加在一起

    }

    for(int i = 0; i < 200; i++)

        ans += arr[i] \* (arr[i] - 1); // C(num,2)组合数

    cout<< ans / 2;

**模拟计算器**:给出n个数，和n - 1个运算符（只含有加减乘号，不含除号，按顺序填入n个数之间），要求输出该式的答案。

int n,num[100];

char flag[100];

int main(){

    cin>>n;

    for(int i = 0; i < n; i++) cin>>num[i];

    for(int i = 1; i < n; i++) cin>>flag[i];

    long long n1 = num[0], n2 = num[1];

    char op = flag[1];

    for(int i = 2; i < n; i++){

        if(flag[i]=='\*'){

            n2\*=num[i];

        }else{

            n1 = (op == '-') ? n1-n2 : (op == '+') ? n1+n2 : n1\*n2;

            op = flag[i];

            n2 = num[i];

        }

    }

    n1 = (op == '-') ? n1-n2 : (op == '+') ? n1+n2 : n1\*n2;

    cout<<n1;

**名望值排队**:n个人排队，每人有一名望值，进队规则如下：第一个人直接进队,新来的人发现队尾的人的名望值比自己大或者相等会选择离开,队伍最多5人，若队满且一人要进队时发现他的名望值比队尾的人大，他会把队首的人挤掉而继续排在队尾。

int n,arr[5],temp,head=0,tail=4,last=0,cur=0;

int main(){

    cin>>n;

    for(int i = 0; i < n; i++){

        cin>>temp;

        if(temp>last){

            last = temp;

            if(cur<5){

                arr[cur++] = i+1;

            }

            else{

                arr[head] = i+1;

                head = (head+1)%5;

                tail = (head+1)%5;

            }

        }

    }

    int len = (n>5)?5:n;

    for(int i = 0; i < len; i++)

        if(arr[(head+i)%5]!=0)

            cout<<arr[(head+i)%5]<<" ";

**台阶问题**:有n级台阶，每次可向上迈最多k级，问到达第级台阶有多少种不同方式。

int n,k,ans[int(1e6)];

int main(){

    ans[0] = 1;

    cin>>n>>k;

    for(int i = 1; i <= n; i++){

        ans[i] = 0;

        for(int j = 1; j<=k; j++){

            ans[i] = (ans[i] + ans[i-j]) % 100003;

        }

    }

    cout<<ans[n];

(分治）**汉诺塔**:现在给你一个n片圆盘的汉诺塔，并从小到大编号为1至。请你输出搬动n个圆盘最少次数的全过程。

//整体法分析，将最底下一个盘子与其上面的多个盘子分为两部分；不用考虑上部的移动，细化为小问题交给递归处理；

void move(int n,char from,char to){

    cout<<"Move disk "<<n<<" from "<<from<<" to "<<to<<endl;

}

// a是出发点，c是目的地

void Hanoi(int n, char a, char b, char c){

    if(n==1) move(n,a,c);

    else{

        Hanoi(n-1,a,c,b);   //上部先移动到辅助塔

        move(n,a,c);        //下部移动到目的地

        Hanoi(n-1,b,a,c);}} //上部移动到目的地

int main(){

    int n;

    cin>>n;

    Hanoi(n,'A','B','C');}

**逆序对**:对于一个序列 a ，如果有且ai > aj 且 i<j，则称 ai，aj 为一逆序对。求出给定序列中逆序对的数量（序列中可能存在重复数字）

const int maxn = 1e5 + 10;

int a[maxn], temp[maxn], n;

long long ans = 0;

void MergeSort(int l, int r){

    if (l >= r) return;

    int mid = (l + r) / 2; // 分为子问题

    MergeSort(l, mid);

    MergeSort(mid + 1, r);

    int tempstart = l, start1 = l, start2 = mid + 1, end1 = mid + 1, end2 = r;

    while (start1 < end1 && start2 <= end2){ // 分完后开始排序

        if (a[start1] <= a[start2]) temp[tempstart++] = a[start1++];

        else{

            temp[tempstart++] = a[start2++];

            ans += end1 - start1;  // 逆序对个数计算

        }

    }

    while (start1 < end1)

        temp[tempstart++] = a[start1++]; // 防止a[]中前面的数还没有移完

    while (start2 <= end2) temp[tempstart++] = a[start2++]; // 同上

    tempstart = l; start1 = l;

    while (tempstart <= r)

        a[start1++] = temp[tempstart++]; // 转移到a[]中

}

int main(){

    while(scanf("%d", &n) != EOF){

        for(int i = 1; i <= n; i ++)

            scanf("%d", &a[i]);

        MergeSort(1, n);

        printf("%d\n", ans);

    }

给你三个整数a,b,p，求（**a的b次方**余p）的值

long long a,b,p,t,ans;

int main(){

    cin>>t;

    while(t--){

        cin>>a>>b>>p;

        ans = 1;

        while(b){

            if(b&1)

                ans = (ans \* a) % p;

            a = (a \* a) % p;

            b/=2;

        }

        cout<<ans<<endl;

每组数据给出n m k表示有n个数，求**第k小**

long long arr[int(5e7)],n,m,k;

int quick\_k(int start,int end, int \_k){

    if(start == end) return arr[start];

    int base = arr[start]; // 以数组第0个元素为base

    int left = start, right = end;

    while(left < right){

        while(left < right && arr[right]>=base) right--;

        arr[left] = arr[right];

        while(left < right && arr[left]<=base) left++;

        arr[right] = arr[left];

    }

    arr[left] = base;

    if(left - start + 1 == \_k) return arr[left];

    else if(left - start >= \_k) return quick\_k(start, left-1, \_k);

    else return quick\_k(left+1, end, \_k-(left-start+1));

}

int main(){

    cin>>n>>m>>k;

    arr[0] = m;

    for(int i = 1; i < n; i ++) arr[i] = 1LL \* arr[i - 1] \* m % int(1e9 + 7);

    cout << quick\_k(0,n-1,k) << endl;

**循环赛日程表** (2的倍数)n个选手(编号1~n)进行循环赛：每个选手必须与其他n−1个选手各赛一次；每个选手一天只能赛一次；循环赛一共进行n−1天；设计一个n行n−1列的表，第i行第j列填入第i个选手第j天的对手；

int arr[32][32],n;

void solution(int n)

{

    if(n==1) return;

    n/=2;

    solution(n);    // 分割直到n=1时开始操作

    // 右上块产生

    for(int i = 1; i <= n; i++)

        for(int j = 1; j <= n; j++)

            arr[i][j+n] = arr[i][j]+n;

    // 右下块复制左上块

    for(int i = 1; i <= n; i++)

        for(int j = 1; j <= n; j++)

            arr[i+n][j+n] = arr[i][j];

    // 左下块复制右上块

    for(int i = 1; i <= n; i++)

        for(int j = 1; j <= n; j++)

            arr[i+n][j] = arr[i][j+n];

}

int main(){

    cin>>n;

    arr[1][1] = 1; // 设置初始块为第一位选手

    solution(n);

    for(int i = 1; i <= n; i++){

        for(int j = 2; j <= n; j++)

            cout<<arr[i][j]<<" ";

        cout<<endl;

**整数位乘**:以二进制形式给出两个数，求它们的乘积，也以二进制表示。

struct BigBinary{

    std::vector<int> x;     // 由低位到高位保存二进制位

    void check(){            // 检测格式

        x.push\_back(0);

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

            x[i+1] += x[i] >> 1;

            x[i] &= 1;

        }

        while(x[x.size()]>1){

            x.push\_back(x[x.size()-1] >> 1);

            x[x.size()-1] &= 1;

        }

        while(!x.empty() && x.back() == 0)

            x.pop\_back();

    }

    void Print(){

        check();

        for(int i = x.size()-1; i >= 0; i--) printf("%d", x[i]);

        if(x.empty()) printf("0");

        printf("\n");

    }

    BigBinary(){x.clear();}

    BigBinary(int start, int end, vector<int> \_x){

        x.clear();

        for(int i = start; i <= end; i ++) x.push\_back(\_x[i]);

    }

    BigBinary(char buf[]){

        x.clear();

        for(int i = strlen(buf) - 1; i >= 0; i --) x.push\_back(buf[i] == '1');

    }

};

BigBinary Add(const BigBinary &a, const BigBinary &b, int flag=1){

    BigBinary c;

    c.x = a.x;

    for(int i = 0; i < b.x.size(); i ++)

        c.x[i] += b.x[i] \* flag;

    return c;

}

BigBinary Mul(const BigBinary &a, const BigBinary &b){

    BigBinary c;

    c.x.resize(a.x.size() \* b.x.size() + 1);

    for(int i = 0; i < a.x.size(); i ++)

        for(int j = 0; j < b.x.size(); j ++)

            c.x[i + j] += a.x[i] \* b.x[j];

    return c;

}

void MulN2(BigBinary &a, int n\_2){

    int size = a.x.size();

    a.x.resize(size + n\_2);

    for(int j = size - 1; j >= 0; j--)

        a.x[a.x.size() - size + j] = a.x[j];

    for(int i = n\_2 - 1; i >= 0; i --)

        a.x[i] = 0;

}

BigBinary FasterMul(const BigBinary &a, const BigBinary &b){

    if(a.x.size() < 32) return Mul(a, b);

    int n\_2 = a.x.size() >> 1;

    BigBinary A(n\_2, a.x.size()-1, a.x);

    BigBinary B(0, n\_2-1, a.x);

    BigBinary C(n\_2, b.x.size()-1, b.x);

    BigBinary D(0, n\_2-1, b.x);

    BigBinary A\_C = FasterMul(A, C);

    BigBinary B\_D = FasterMul(B, D);

    // AD+BC = (A+B)\*(C+D)-AC-BD，该方式避免减法出现负数，恒有(A+B)\*(C+D) >= AC+BD

    BigBinary ADpBC = Add(Add(FasterMul(Add(A, B), Add(C, D)), A\_C , -1), B\_D, -1);

    MulN2(A\_C, n\_2 << 1);

    MulN2(ADpBC, n\_2);

    return Add(Add(A\_C, ADpBC), B\_D);

}

const int maxn = 1e5 + 10;

char buf[maxn];

int main(){

    while(scanf("%s", buf) != EOF)

    {

        BigBinary a(buf);

        scanf("%s", buf);

        BigBinary b(buf);

        // 提前设置size

        if(a.x.size() < b.x.size()) a.x.resize(b.x.size(), 0);

        else b.x.resize(a.x.size(), 0);

        FasterMul(a, b).Print();

    }

(动态规划）**最大子串和**:找一具有最大和的连续子数组，返回其最大和。

int n, a[int(1e5 + 10)], dp[int(1e5 + 10)],ans=-1e9;

int main(){

    cin>>n;

    for(int i = 1; i <= n; i ++){

        // 如果之前累加得到的是正数，则继续累加，累加得到负数，取消累加。

        cin>>a[i];

        dp[i] = max(dp[i-1]+a[i],a[i]);

        ans = max(dp[i],ans);

    }

    cout<<ans<<endl;

**方块堆塔**:有n种立方体，给出三条棱长，求最高叠多高(下方的底面长宽一定大于上方的).

struct{int x,y,z;} cubes[350];

int ans,n,arr[310][310],dp[310];

int dfs(int i){

    if(dp[i]!=-1) return dp[i];

    dp[i] = cubes[i].z;

    for(int j = 0; j <= 3\*n; j++)

        if(arr[i][j]) dp[i] = max(dp[i], dfs(j)+cubes[i].z);

    return dp[i];

}

int main(){

    while(cin>>n){

        ans = 0;

        for(int i = 0; i < 3\*n; i+=3){

            //每一种立方体的底面有三种情况

            cin>>cubes[i].x>>cubes[i].y>>cubes[i].z;//1

            cubes[i+1].x=cubes[i].y;// 2

            cubes[i+1].y=cubes[i].z;

            cubes[i+1].z=cubes[i].x;

            cubes[i+2].x=cubes[i].z;// 3

            cubes[i+2].y=cubes[i].x;

            cubes[i+2].z=cubes[i].y;

        }

        for(int i = 0; i < 3\*n; i++){

            dp[i] = -1; // 初始化dp数组

            for(int j = 0; j < 3\*n; j++)

                if(cubes[i].x>cubes[j].x&&cubes[i].y>cubes[j].y || cubes[i].x>cubes[j].y&&cubes[i].y>cubes[j].x )//两种情况，长宽可换位

                    arr[i][j] = 1;// 标记下标为j的立方体能在下标为i的立方体上

                else arr[i][j] = 0;

        }

        for(int i = 0; i < 3\*n; i++)

            ans=max(ans,dfs(i));

        cout<<ans<<endl;

**矩阵链相乘的乘法次**

int n,arr[310],dp[310][310];

int main(){

    while(cin>>n){

        for(int i = 0; i <= n; cin>>arr[i++]);

        for(int i = 0; i <= n; i++)

            for(int j = 0; j <= n; j++)

                dp[i][j]=0;

        for(int r = 2; r <= n; r++){

            int j = r;

            for(int i = 1; i <= n-(r-1); i++){// i为连乘起点位置，取值不能大于n-r+1

                j = i+(r-1);// j为连乘终点位置

                dp[i][j] = dp[i + 1][j] + arr[i - 1] \* arr[i] \* arr[j];//初始化值

                for(int k = i+1; k < j; k++){ //遍历由i到k的全部可能的划分点k，计算出最优的划分方案

                    int q = dp[i][k] + dp[k + 1][j] + arr[i - 1] \* arr[k] \* arr[j]; //计算划分的代价,意思是看在哪套括号最好

                    if (q < dp[i][j])

                        dp[i][j] = q; //最优的值保存在m[i][j]中

                }

            }

        }

        cout << dp[1][n] << endl;

**括号匹配II**

int dp[105][105];

string str;

bool judge(char a, char b){

    return ((a=='(' && b==')')||(a=='[' && b==']')||(a=='{' && b=='}'));

}

int main(){

    while(cin>>str){

        memset(dp,0,sizeof(dp));

        for(int i = 0; i < str.length(); i++) dp[i][i] = 1; // 自己对自己的符号缺失数为1

        for (int len = 1; len < str.length(); len++) // 从长度为2开始，记录每个长度为len的括号补充数

            for (int i = 0; i < str.length() - len; i++)

            {

                int j = i + len;

                dp[i][j] = 1e9; // 初始化一个极大值，方便后面缩小

                // 检查当前子字符串是否是一个有效的括号序列，如果是，我们更新最少插入次数为左右两边字符组成的子序列的最小插入次数。

                if (judge(str[i], str[j]))

                    dp[i][j] = min(dp[i][j], dp[i + 1][j - 1]);

                // 在当前子字符串计算各个分割点 k 处左右两边字符组成的子序列的最小插入次数之和，取最小值作为当前子字符串的最少插入次数。

                for (int k = i; k < j; k++)

                    dp[i][j] = min(dp[i][j],dp[i][k] + dp[k + 1][j]);

            }

        if (dp[0][str.length() - 1] != 0)

            cout << dp[0][str.length() - 1] << endl;

        else

            cout << "SZTU\_WOD YYDS!" << endl;

**物品无限的背包**问题(**01背包**则把dp[i][j-w[i]]+v[i]改为dp[i-1][j-w[i]]+v[i])

int n,b,w[1000],v[1000],dp[1001][10001];

int main(){

    cin>>n>>b;

    for(int i = 0; i <= n; i++) dp[i][0] = 0;

    for(int i = 1; i <= b; i++) dp[0][i] = 0;

    // i为要考虑拿的第i物品，j为背包容量

    for(int i = 1; i <= n; i++){

        cin>>w[i]>>v[i];

        for(int j = 1; j <= b; j++){

            dp[i][j] = (j>=w[i]) ? max(dp[i-1][j],dp[i][j-w[i]]+v[i]) : dp[i-1][j];

        }

    }

    cout<<dp[n][b]<<endl;

**最长上升子序列（小规模）**

int main(){

    int n,ans = 0,arr[1000],dp[1000];

    cin>>n>>arr[1];

    for(int i = 2; i <= n; i ++) // 生成序列

        arr[i] = 1LL \* (arr[i - 1] + 1) \* (arr[i - 1] + 1) % int(1e9 + 7);

    for(int k = 1; k <= n; k++){

        dp[k] = 1; // 初始化计算长度，自身至少长为 1

        for(int j = 1; j < k; j++)

            if(arr[j] < arr[k]) dp[k] = max(dp[k],dp[j] + 1);

        ans = max(dp[k],ans);

    }

    cout<<ans;

**最长上升子序列（大规模）**

int n,dp[int(1e6)],arr[int(1e6)];

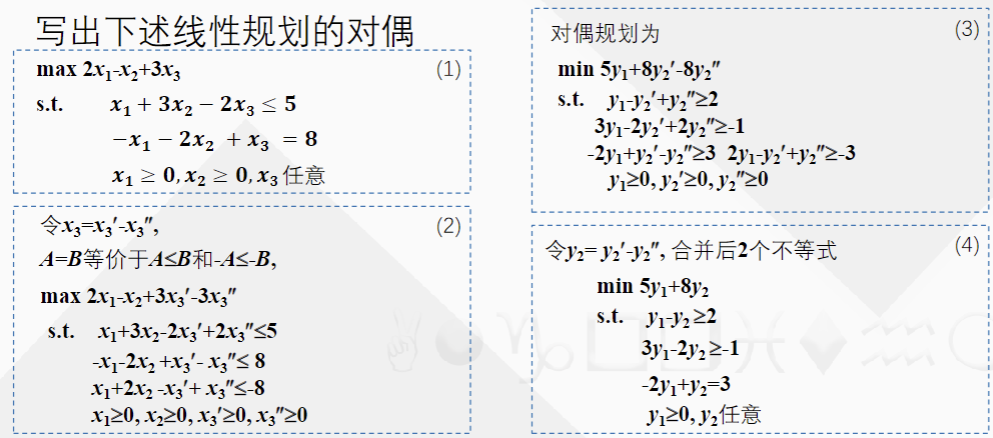
int main() {

    cin>>n>>arr[1];

    for(int i = 2; i <= n; i ++) // 生成序列

        arr[i] = 1LL \* (arr[i - 1] + 1) \* (arr[i - 1] + 1) % int(1e9 + 7);

    // dp的内容不能作为最终字串结果，但能得到最长字串长度。

    int len = 0,l,r,mid;

    for(int i=1;i<=n;i++){

        l=0; r=len;

        while(l<r){

            mid = (l+r+1)>>1;

            if(dp[mid]<arr[i]) l=mid;

            else r=mid-1;

        }

        len=max(len,l+1);

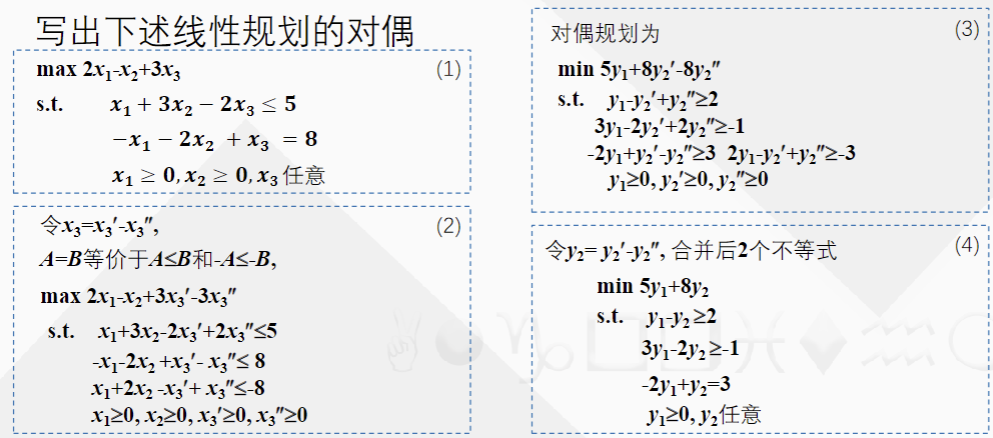
        dp[l+1]=arr[i];

    }

    cout<<len<<endl;

**最长公共子序列**

int dp[1001][1001];

string a,b;

int main(){

    for (int i = 0; i < 1001; i++)

        dp[0][i] = dp[i][0] = 0;

    while(cin>>a>>b){

        for(int i = 1; i <= a.length(); i++)

            for(int j = 1; j <= b.length(); j++){

                if(a[i-1]==b[j-1])

                    dp[i][j] = dp[i-1][j-1]+1;

                else

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

            }

        cout<<dp[a.length()][b.length()]<<endl;

(贪心）**活动选择**

struct Time{int start; int end;} arr[10000];

int n, cur\_end, ans;

bool comp(Time a, Time b){return a.end<b.end;}

int main(){

    cin>>n;

    for(int i = 0; i < n; i++)

        cin>>arr[i].start>>arr[i].end;

    sort(arr, arr+n, comp);

    ans = 1; cur\_end = arr[0].end;

    for(int i = 1; i < n; i++){

        if(cur\_end <= arr[i].start){

            cur\_end = arr[i].end;

            ans += 1;

        }

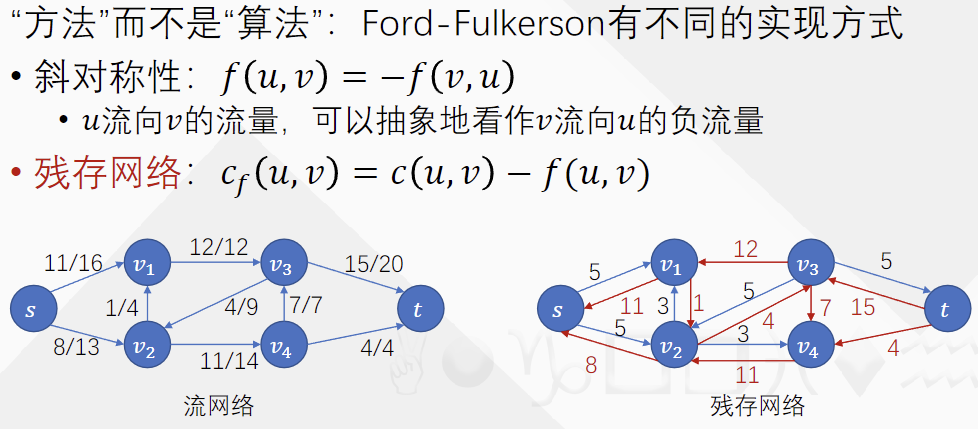
    }

    cout<<ans;

3

**最少拦截系统**

int t,arr[100],height,cnt,flag;

int main(){

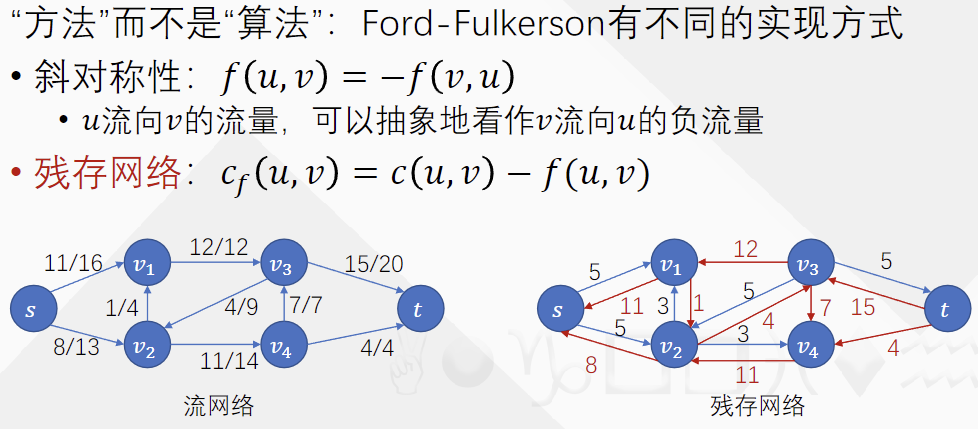
    while(cin>>t){

        cnt = 0;arr[0] = 0;

        while(t--){

            cin>>height;

            flag = 0;

            for(int i = 0; i < cnt; i++)

                if(height<arr[i]){

                    arr[i] = height;

                    flag = 1;

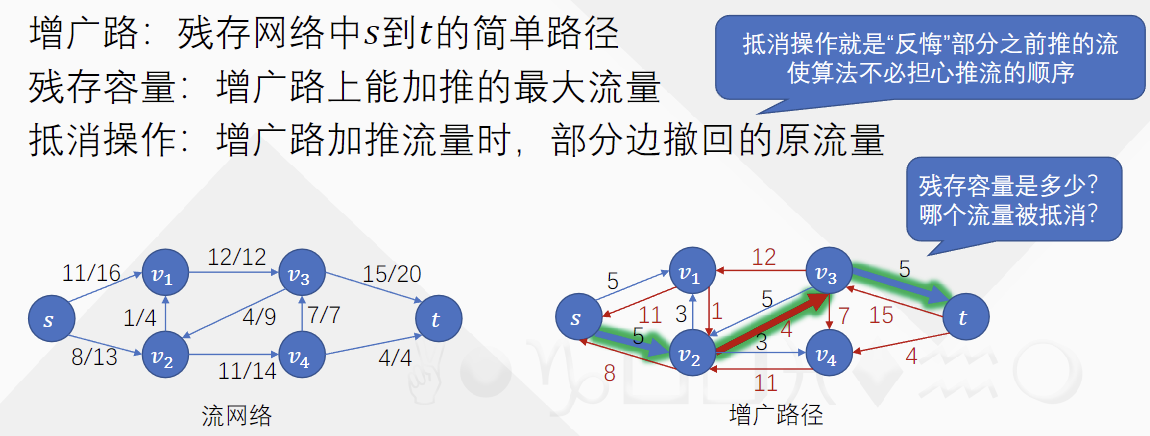
                    break;

                }

4

            if(!flag) arr[cnt++] = height;

        }

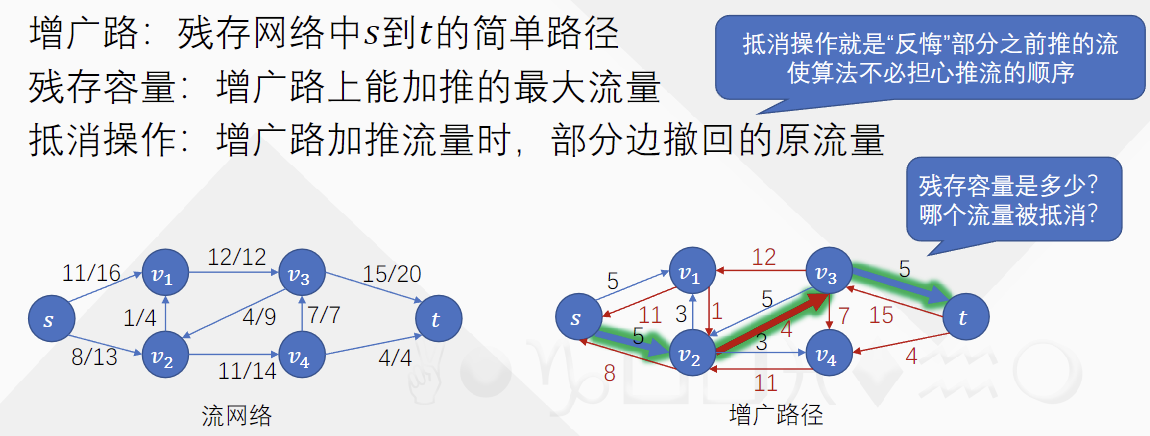
        cout<<cnt<<endl;

（回溯）

**枚举全排列**

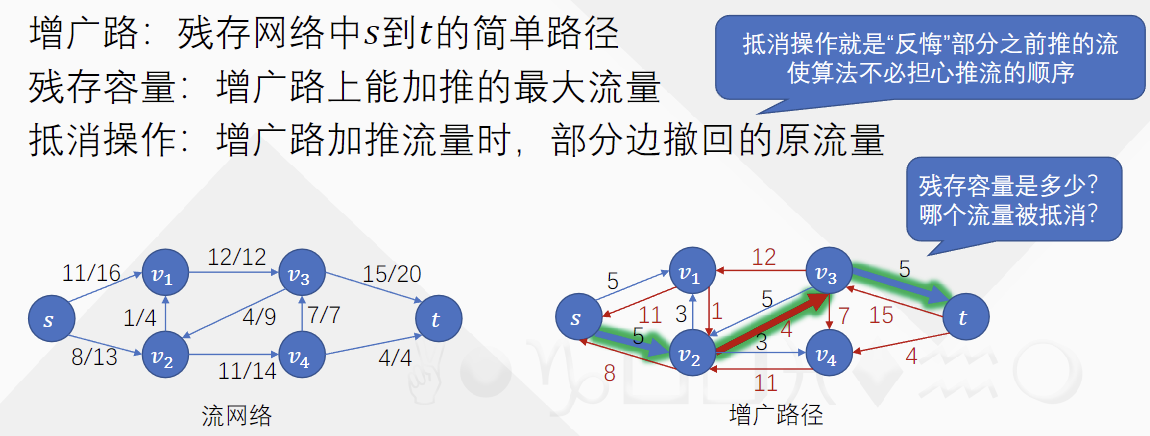
#include<bits/stdc++.h>

using namespace std;



int n,arr[8],ans[8],vis[8];

void dfs(int cur){

    if(cur==n){

        for(int i = 0; i < n; i++)

            cout<<ans[i]<<" ";

        cout<<endl;

        return;

    }

    for(int i = 0; i < n; i++){

        if(!vis[i]){

            vis[i] = 1;

            ans[cur] = arr[i];

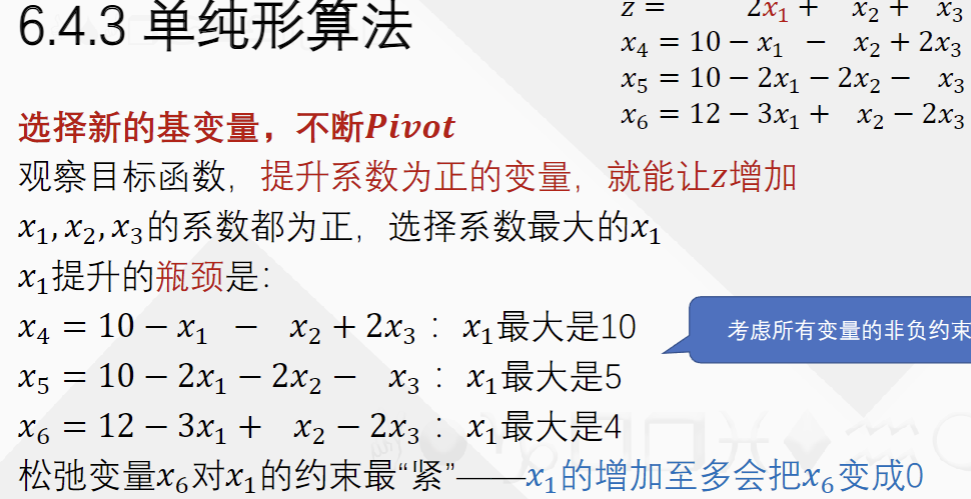
            dfs(cur+1);

            vis[i] = 0;

        }

    }

}

int main(){

    cin>>n;

    for(int i = 0; i < n; i++){

        cin>>arr[i];

        vis[i] = 0;

    }

    sort(arr,arr+n);

    dfs(0);

**生成可重排列**

int n,m,arr[8],ans[8],cnt[8];

void dfs(int cur){

    if(cur==n){

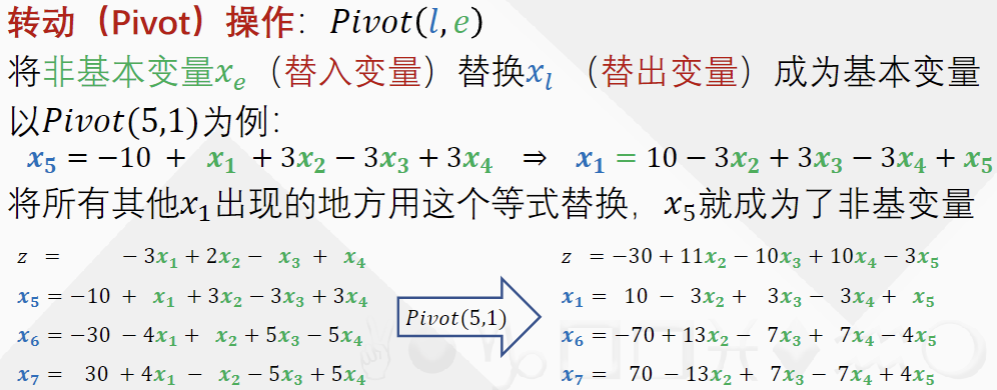
        for(int i = 0; i < n; i++)

            cout<<ans[i]<<" ";

        cout<<endl;

        return;

    }

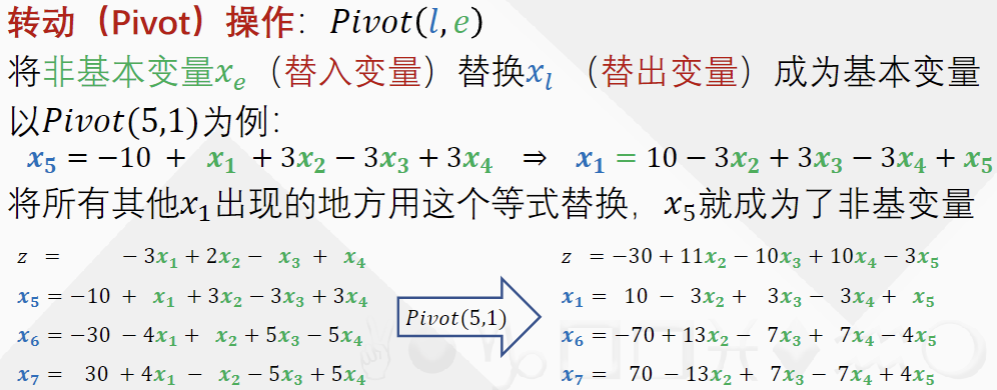
    for(int i = 0; i < m; i++){

        if(cnt[i]){

            cnt[i]--;

            ans[cur] = arr[i];

            dfs(cur+1);

            cnt[i]++;

        }

    }

}

int main(){

    cin>>n;

    m = 0;

    for(int i = 0; i < n; i++){

        cin>>arr[i];

        cnt[i] = 0;

    }

    sort(arr,arr+n);

    for(int i = 0; i < n; i++){

        if(m==0 || arr[m-1] != arr[i])

            arr[m++] = arr[i];

        cnt[m-1]++;

    }// 利用cnt计算重复元素个数，避免同一层相同元素重复调用;

    dfs(0);

**生成r子集**

int n,r,arr[10];

bool sel[10];

void dfs(int cur, int rcnt){

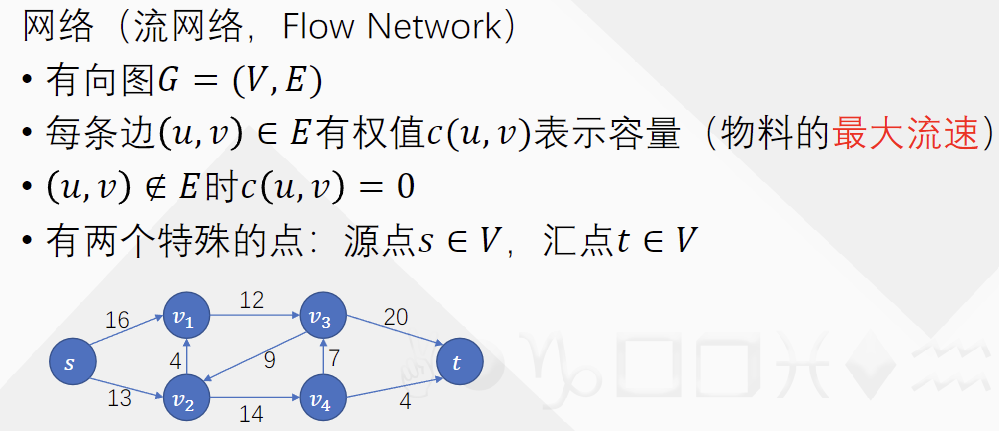
    if(cur==-1){

        if(rcnt != r) return;

1

        for(int i = 0; i < n; i++)

            if(sel[i]) cout<<arr[i]<<" ";

        cout<<endl;

        return;

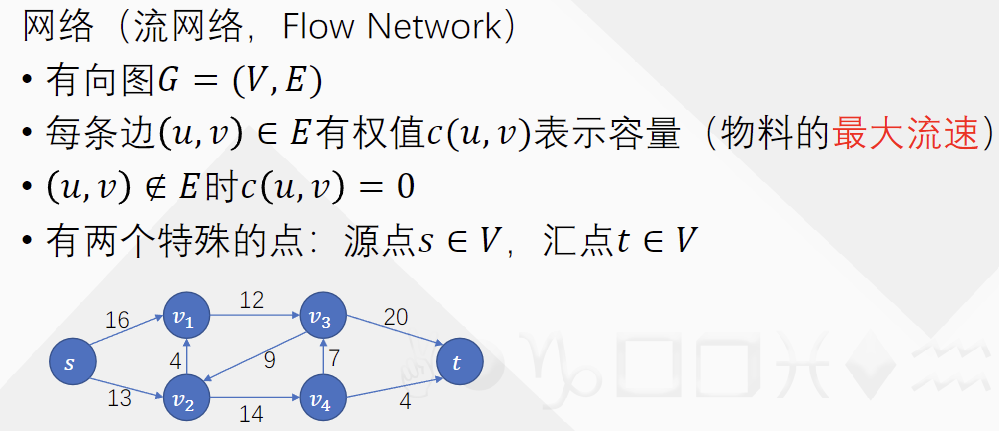
    }

    sel[cur] = false;

    dfs(cur-1,rcnt);

    sel[cur] = true;

    dfs(cur-1,rcnt+1);

}

int main(){

    while(cin>>n>>r){

        for(int i = 0; i < n; cin>>arr[i++]);

        dfs(n-1, 0);

**n皇后问题**

// 我的解法(模拟真棋盘，空间耗费多)

int n,cnt,qp[13][13];

bool check(int r, int c){

    for(int i = 0; i < r; i++)

        if( (qp[i][c]) || (c+(r-i)<n && qp[i][c+(r-i)]) || (c-(r-i)>=0 && qp[i][c-(r-i)]) ) return false;

    return true;

}

void dfs(int cur){ // 棋盘第cur行

    if(cur==n){ cnt++; return;}

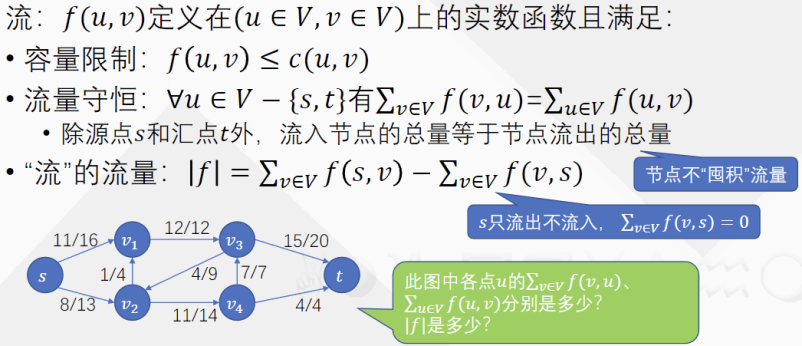
    for(int i = 0; i < n; i++)

        if(!qp[cur][i] && check(cur,i)){

            qp[cur][i] = 1;

2

            dfs(cur+1);

            qp[cur][i] = 0;

        }

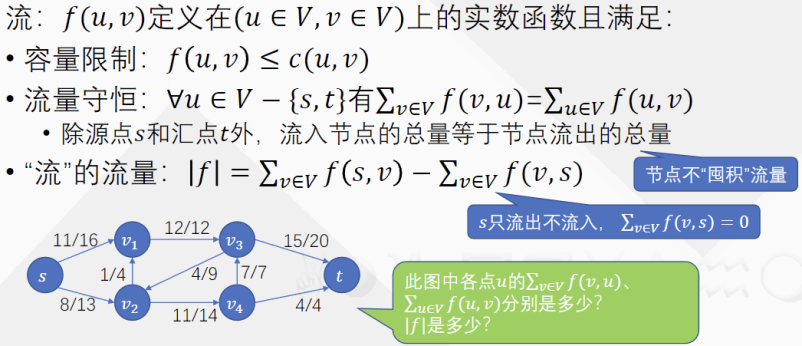
}

int main(){

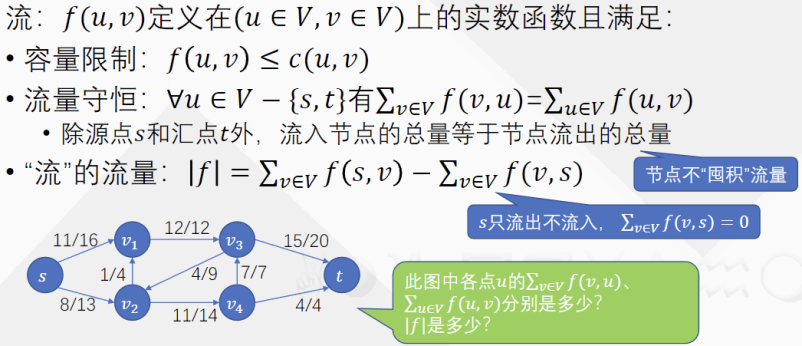
    while(cin>>n){

        cnt = 0;

        dfs(0);

        cout<<cnt<<endl;

// 答案解法(数值处理直线、对角线)

int n, count;

bool column[13], diag1[25], diag2[25];

void backtrack(int k) {

    if (k == n) {count++; return;}

    for (int i = 0; i < n; i++)

        if (!column[i] && !diag1[k + i] && !diag2[k - i + n - 1]) {

            column[i] = diag1[k + i] = diag2[k - i + n - 1] = true;

            backtrack(k + 1);

            column[i] = diag1[k + i] = diag2[k - i + n - 1] = false;

        }

}

int main() {

    while (cin >> n) {

        for (int i = 0; i < n; i++) column[i] = false;

        for (int i = 0; i < 2 \* n - 1; i++) diag1[i] = diag2[i] = false;

        count = 0;

        backtrack(0);

        cout << count << endl;

**着色问题**

int n,m,q,graph[11][11],color[11],r,c,cnt;

bool check(int cur, int col){

    for(int i = 0; i < n; i++)

        if(graph[cur][i] && color[i]==col) return false;

    return true;

}

void dfs(int cur){

    if(cur==n){cnt++;return;}

    for(int i = 1; i<=m; i++){

        if(check(cur,i)){

            color[cur] = i;

            dfs(cur+1);

            color[cur] = 0;

        }

    }

}

int main(){

    while(cin>>n>>m>>q){

        for(int i = 0; i < n; i++){

            color[i] = 0;

            for(int j = 0; j < n; j++)

                graph[i][j] = 0;

        }

        while(q--){

            cin>>r>>c;

            graph[r-1][c-1] = graph[c-1][r-1] = 1;

        }

        cnt = 0;

        dfs(0);

        cout<<cnt<<endl;

**最佳安排**

int n,arr[11][11],ans;

bool select[11];

void dfs(int cur, int sum){

    if(sum>=ans)return;

    if(cur == n){ans = min(sum, ans); return;}

    for(int i = 0; i < n; i++)

        if(!select[i]){

            select[i]=true;

            dfs(cur+1, sum + arr[cur][i]);

            select[i]=false;

        }

}

int main(){

    while(cin>>n){

        for(int i = 0; i < n; i++)

            for(int j = 0; j < n; j++)

                cin>>arr[i][j];

        ans = 1e9;

        dfs(0, 0);

        cout<<ans<<endl;

**合适的01**

string str;

int n,cnt;

void dfs(int cur, int sum){

    if(cur == n+1){

        if(sum>0) cnt++;

        return;

    }

    dfs(cur+1, sum); // 0 不用处理加减

    if(cur==0 || str[cur-1]=='+') dfs(cur+1,sum+1);

    else dfs(cur+1, sum-1);

}

int main(){

    while(cin>>str){

        cnt = 0;

        n = str.length();

        dfs(0,0);

        cout<<cnt<<endl;

**最大团**(分支界限)

完全图：如果无向图中的任何一对顶点之间都有一条边，这种无向图称为完全图。

完全子图(团)：给定无向图G=(V,E)。如果U⊆V，且对任意u，v⊆U 有(u，v) ⊆ E，则称U 是G 的完全子图。V是顶点集，E是边集。

最大团：G 的最大团是指G中所含顶点数最多的团。

空子图：在一个图的子图中，边集为空集的子图。简单来说，空子图包含一些顶点，但这些顶点之间没有任何边相连。

独立集：对于给定无向图G=(V,E)。如果顶点集合V\*⊆V，若V\*中任何两个顶点均不相邻，则称V\*为G的点独立集，或简称独立集。

最大独立集：G中所含顶点数最多的独立集。

int g[36][36],vis[36],n,m,ans,r,c;

bool check(int cur){

    for(int i = 0; i < n; i++)

        if(vis[i] && !g[cur][i]) return false;

    return true;

}

void dfs(int cur,int cnt){

    if(cur==n){ans = max(ans, cnt); return;}

    // 分支限界：剩下的点都算上也不如当前的界则回溯

    if(cnt + n - cur + 1 <= ans)return;

    if(check(cur)){

        vis[cur] = 1;

        dfs(cur+1,cnt+1); // 加 cur

    }

    vis[cur] = 0;

    dfs(cur+1,cnt); // 不加 cur

}

int main(){

    while(cin>>n>>m){

        memset(g, 0, sizeof(g));

        memset(vis, 0, sizeof(vis));

        for(int i = 0; i < m; i++){

            cin>>r>>c;

            g[r-1][c-1] = g[c-1][r-1] = 1;

        }

        ans = 0;

        dfs(0, 0);

        cout<<ans<<endl;

**旅行商**

int g[13][13],vis[13],n,ans,r,c,v;

int path[13]; //记录路径

int min\_dis[13]; //每个点出发的最小边长度

// 当前点 , 当前路径长度, 剩余最短路径长度

void dfs(int cur, int dis, int min\_dis\_sum)

{

    if (cur == n-1){ans = min(dis + g[path[cur]][0], ans); return;}

    if(dis + min\_dis\_sum >= ans) return; // 分支限界：已确认路长+剩下点最小边之和

    for(int i = 1; i < n; i ++) { //回路，1起点，2开始查找

        if(!vis[i]){

            vis[i] = true;

            path[cur+1] = i;

            dfs(cur + 1, dis + g[path[cur]][i], min\_dis\_sum - min\_dis[i]);

            vis[i] = false;

        }

    }

}

int main(){

    while(cin>>n){

        memset(g,0,sizeof(g));

        memset(vis,0,sizeof(vis));

        memset(min\_dis,1e9,sizeof(min\_dis));

        // fill(min\_dis, min\_dis+13, 1e9);

        int m = n \* (n - 1) / 2;

        for(int i = 0; i < m; i++){

            cin>>r>>c>>v;

            g[r-1][c-1] = g[c-1][r-1] = v;

            // 获取每个点最短路径长度

            min\_dis[r-1] = min(min\_dis[r-1],v);

            min\_dis[c-1] = min(min\_dis[c-1],v);

        }

        int min\_dis\_sum = 0; // 所有点最短路径长度之和

        for(int i = 0; i < n; i++) min\_dis\_sum += min\_dis[i];

        ans = 1e9;

        path[0] = 0; //回路，固定1为起点

        dfs(0, 0, min\_dis\_sum);

        cout << ans << endl;

**圆排列最小宽度**

int n,selected[12],arr[12];// 圆排列顺序

double ans,r[12],pos[12];// 圆的半径/圆心位置

double get\_pos(double R, int cur){ // 计算当前圆的圆心位置

    // 第一个圆，将圆心放置在半径处

    if (cur == 0) return R;

    // 遍历已排列的圆，找到与当前圆相切的最右位置

    // 要考虑当前圆比较大，碰到的不一定是左侧相邻的圆

    double d, cur\_pos = R;

    for (int i = 0; i < cur; i++){

        d = sqrt(pow(r[arr[i]] + R, 2) - pow(r[arr[i]] - R, 2));

        cur\_pos = max(cur\_pos, pos[i] + d);

    }

    return cur\_pos;

}

bool judge(int cur, double len) { // 分支界限

    double min\_r = 1e11;

    for (int i = 0; i < n; i++) // 找到未排列圆中半径最小的圆

        if (!selected[i]&&r[i]<min\_r) min\_r = r[i];

    // 计算未排列最小圆的圆心位置

    double min\_r\_pos = get\_pos(min\_r, cur);

    // 设未排列最小圆为所有未排列圆的半径,计算全部加入后的最右位置，若比当前最优解差，则剪枝

    double cur\_len = max(len, min\_r\_pos + min\_r \* ((n - cur) \* 2 - 1));

    return cur\_len < ans;

}

void dfs(int cur, double len)

{

    if (cur == n){ans = min(len,ans); return;}

    if(!judge(cur,len)) return;

    for (int i = 0; i < n; i++){

        if (!selected[i]){

            selected[i] = true;

            double cur\_pos = get\_pos(r[i], cur);

            // 更新排列的最右端位置

            // 考虑到最右端的圆可能会比较小，所以最右边的位置不一定是最右端的圆+其半径

            double real\_cur\_len = max(len, cur\_pos + r[i]);

            arr[cur] = i;

            pos[cur] = cur\_pos;

            dfs(cur + 1, real\_cur\_len);

            selected[i] = 0;

        }

    }

}

int main(){

    cin >> n;

    for (int i = 0; i < n; cin >> r[i++]);

    memset(selected,0,sizeof(selected));

    ans = 1e9;

    dfs(0, 0);

    cout << fixed << setprecision(6) << ans;

**线性规划例题**

const double eps = 1e-7;    // 定义一个很小的数，用于比较浮点数的精度

const double inf = 1e20;    // 定义一个很大的数，用于表示无穷大

// 标准型： max Σcx, s.t. ax<=b, x>=0

class Simplex{

public:

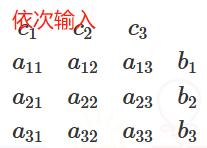
    vector<double> b, c;        // b: 约束条件右端项向量,c: 目标函数系数向量

    vector<vector<double>> a;   // a: 约束条件系数矩阵

    double z;                   // 目标函数值

    int m, n;                   // m: 约束条件个数，n: 自由变量个数

    Simplex(int \_m, int \_n);

    void Pivot(int e, int l);

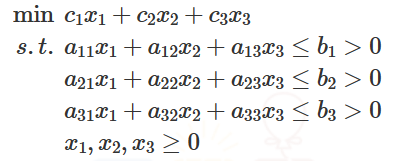
    double Solve();

};

// 构造函数，初始化Simplex对象

Simplex::Simplex(int \_m, int \_n) : m(\_m), n(\_n), z(0){

    a.resize(m + 10, vector<double>(n + 10, 0));

    b.resize(m + 10, 0);

    c.resize(n + 10, 0);

}

// 枢轴操作函数，实现高斯消元法

void Simplex::Pivot(int e, int l){

    b[l] /= a[l][e];

    for (int j = 0; j < n; j++)

        if (j != e) a[l][j] /= a[l][e];

    a[l][e] = 1 / a[l][e]; // 将主元逆转

    for (int i = 0; i < m; i++){

        if (i == l || (a[i][e] > -eps && a[i][e] < eps)) continue; // 排除当前行和近似为0的行

        b[i] -= a[i][e] \* b[l];

        for (int j = 0; j < n; j++)

            if (j != e) a[i][j] -= a[i][e] \* a[l][j];

        a[i][e] = -a[i][e] \* a[l][e];

    }

    z += c[e] \* b[l];                              // 更新目标函数值

    for (int j = 0; j < n; j++)

        if (j != e)                                // 排除主元列

            c[j] -= c[e] \* a[l][j];                // 更新目标函数系数

    c[e] \*= -a[l][e];                              // 更新主元系数

}

// 求解函数，使用单纯形法求解线性规划问题

double Simplex::Solve(){

    while (true){

        int e = -1, l = -1; // e: 入基变量索引，l: 出基变量索引

        double maxc = eps; // 初始化最大目标系数为很小的数

        // 找到最优解的入基变量,找目标函数里最大的正系数

        for (int j = 0; j < n; j++)

            if (c[j] > maxc){

                maxc = c[j];

                e = j; // 确认替入变量

            }

        if (e == -1) return z;// 如果没有入基变量，说明当前解是最优的

        // 有入基变量，则准备转动

        double minba = inf; // 初始化最小比值为无穷大

        // 找到最优解的出基变量,

        for (int i = 0; i < m; i++)

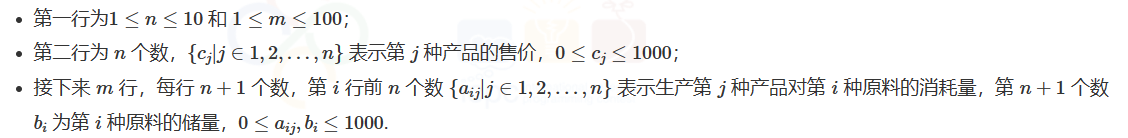
            if (a[i][e] > eps && minba > b[i] / a[i][e]){

                minba = b[i] / a[i][e]; // 找替入变量增加的瓶颈

                l = i; // 替出变量是卡住替入变量增长瓶颈那一行的松弛变量

            }

        if (l == -1) return inf;// 如果没有出基变量，说明无界解

        Pivot(e, l);

    }

}

int main(){

    Simplex spx(3, 3);

    cin >> spx.c[0] >> spx.c[1] >> spx.c[2];       // 输入目标函数系数

    for (int i = 0; i < 3; i++){

        // 转换为标准型：目标函数统一为max

        spx.c[i] \*= -1;

        // 输入约束条件

        for(int j = 0; j < 3; j++) cin >> spx.a[i][j];

        cin >> spx.b[i];

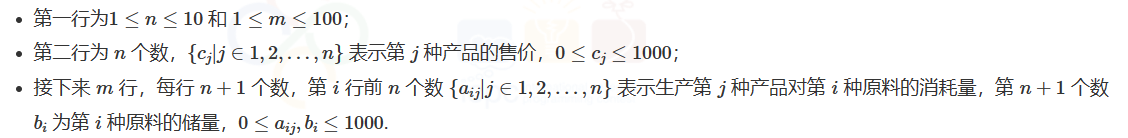
    }

    double res = spx.Solve();

    // 如果返回值接近无穷大，说明无解

    if (res >= inf - eps) cout<<"No solution"<<endl;

    else cout<<fixed<<setprecision(2)<<-res<<endl; // 输出最优解，并将其取反恢复原来的符号

}用m种原料生产种产品，每种n产品需要消耗各类原料，同时有个固定售价，求各类产品最优生产数量，使全部售卖得到的收入最高.

int main(){

    int m,n;

    cin>>n>>m;

    Simplex spx(m, n);

    for(int i = 0; i < n; i++){

        cin >> spx.c[i]; // 输入目标函数系数

        // spx.c[i] \*= -1;  // 转换为标准型：目标函数统一为max

    }

    for (int i = 0; i < m; i++){

        // 输入约束条件

        for(int j = 0; j < n; j++) cin >> spx.a[i][j];

        cin >> spx.b[i];

    }

    double res = spx.Solve();

    // // 如果返回值接近无穷大，说明无解

    cout<<fixed<<setprecision(6)<<res<<endl;

**防守战线**

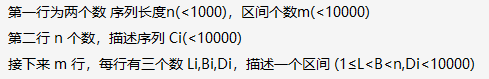
int main() {

    int n, m;

    cin>>n>>m;

    Simplex spx(n, m);

    vector<double> C(n);

    vector<int> L(m), R(m);

    vector<double> D(m);

    for(int i = 0; i < n; i ++)

        cin>>C[i];

    for(int i = 0; i < m; i ++)

        cin>>L[i]>>R[i]>>D[i];

    // 对偶转换

    for(int i = 0; i < n; i ++) {

        for(int j = 0; j < m; j ++)

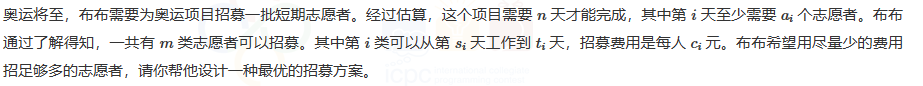
            spx.a[i][j] = i >= L[j] - 1 && i <= R[j] - 1;

        spx.b[i] = C[i];

    }

    for(int j = 0; j < m; j ++) spx.c[j] = D[j];

    cout<<(int)spx.Solve()<<endl;}

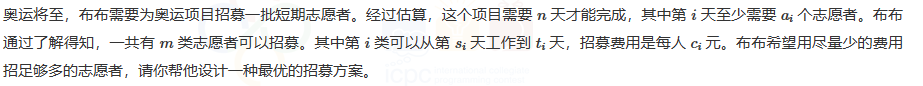
**志愿者招募**

int main(){

    int n, m;

    cin >> n >> m;

    Simplex spx(m, n);

    vector<int> S(m), T(m), C(m), A(n);

    for (int i = 0; i < n; i++)

        cin >> A[i];

    for (int i = 0; i < m; i++)

        cin >> S[i] >> T[i] >> C[i];

    // 对偶转换

    for (int i = 0; i < n; i++)

        spx.c[i] = A[i];

    for (int j = 0; j < m; j++){

        for (int i = 0; i < n; i++)

            spx.a[j][i] = (i + 1 >= S[j] && i + 1 <= T[j]);

        spx.b[j] = C[j];

    }

    cout << fixed << setprecision(0) << spx.Solve() << endl;

**最大流问题**

const int MAXN = 300; // 定义最大节点数为 300

int N,M,S,T; //点数量，边数量，源点，汇点

int C[MAXN][MAXN];  // 容量矩阵

int F[MAXN][MAXN];  // 流量矩阵，用于存储每条边当前的流量

int parent[MAXN];   // 父节点数组，用于存储增广路径中的父节点

bool bfs(){

    bool visited[MAXN] = {false};

    queue<int> queue;

    queue.push(S);// 源点访问

    visited[S] = true;

    while(!queue.empty()){

        int u = queue.front();

        queue.pop();

        for(int i = 0; i<N; i++)

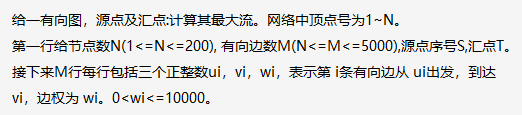
            if(!visited[i]&&C[u][i]>F[u][i]){

                queue.push(i);

                visited[i] = true;

                parent[i] = u;

                if(i==T) return true;

            }

    }

    return false;

}

int edmonds\_karp(){

    int max\_flow = 0;

    while(bfs()){

        int path\_flow = 1e9; // 初始化路径流为无限大

        for(int i = T; i != S; i = parent[i]){ // 找到最小的路径流

            int u = parent[i];

            path\_flow = min(path\_flow, C[u][i] - F[u][i]);

        }

        for(int i = T; i != S; i = parent[i]){

            int u = parent[i];

            F[u][i] += path\_flow;

            F[i][u] -= path\_flow;

        }

        max\_flow += path\_flow; // 增加路径流到最大流

    }

    return max\_flow;

}

int main(){

    // 初始化全局变量 C

    for (int i = 0; i < N; ++i)

        for (int j = 0; j < N; ++j) {

            C[i][j] = 0;

        }

    cin>>N>>M>>S>>T;

    --S;--T;

    int u,v,w; //边起点，边终点，边容量

    for(int i = 0; i < M; i++){

        cin>>u>>v>>w;

        --u;--v;

        C[u][v] += w;

    }

    cout<<edmonds\_karp()<<endl;

}

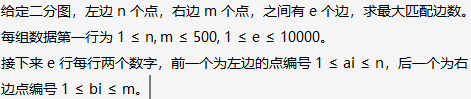
**基本的二分图**

int main() {

    int n, m, e; // 左右两侧顶点数和边数

    cin >> n >> m >> e;

    N = n + m + 2; // 总节点数，包括源点和汇点

    S = 0; // 源点

    T = N - 1; // 汇点

    // 初始化容量矩阵C

    memset(C, 0, sizeof(C));

    memset(F, 0, sizeof(F));

    // 读入边并构建图

    for (int i = 0; i < e; ++i) {

        int u, v;

        cin >> u >> v;

        // 左侧点u连接到右侧点v，u在源点后的编号是u，v在右侧的编号是v+n

        C[u][v + n] = 1; // 每条边的容量为1

    }

    for (int i = 1; i <= n; ++i) { // 源点连接到所有左侧点

        C[S][i] = 1; // 容量为1

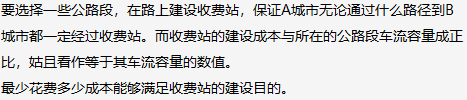
    }

    for (int i = 1; i <= m; ++i) { // 所有右侧点连接到汇点

        C[i + n][T] = 1; // 容量为1

    }

    cout << edmonds\_karp() << endl;

}

**收费站建设**

int main() {

    cin >> N >> M;

    S = 0; // 源点编号（A城市）

    T = 1; // 汇点编号（B城市）

    // 初始化容量矩阵C和流量矩阵F

    memset(C, 0, sizeof(C));

    memset(F, 0, sizeof(F));

    // 读入边并构建图

    for (int i = 0; i < M; ++i) {

        int u, v, c;

        cin >> u >> v >> c;

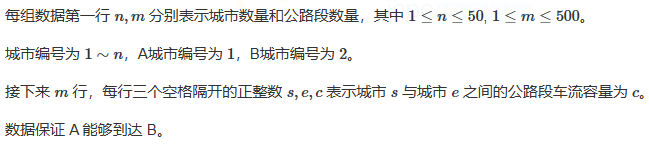
        --u; --v; // 将节点编号转换为从0开始

        C[u][v] += c; // 容量

        C[v][u] += c; // 因为无向图，反向边也要设置容量

    }

    cout << edmonds\_karp() << endl;

}

  
  
  
  
  
  
**较轻的小球**

int n, m, cur;

int solu(int l, int r){

    cin>>cur;

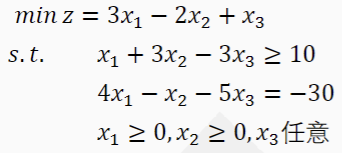
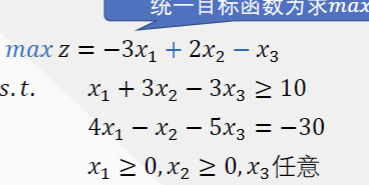
    int right = ((r-l+1)&1) ? r-1 : r;

    if(cur == 2) return r;

    else if(cur == 1){

        if(l == right-1) return right;

        return solu((l+right)/2+1,right);

    }

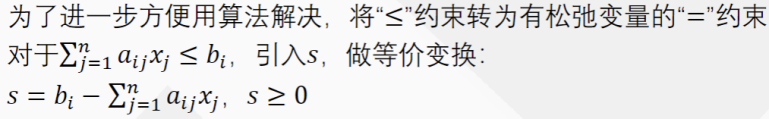
    else if(cur == 0){

        if(l == right-1) return l;

        return solu(l,(l+right)/2);

    }

}

int main(){

    while(cin>>n>>m)

        cout<<solu(1,n)<<endl;

}

**01饭卡**

int n,m,ans,min\_bal,arr[1001],dp[1001][1001];

int main(){

    while(cin>>n && n!=0){

        memset(dp,0,sizeof(dp));

        for(int i = 1; i <= n; cin>>arr[i++]); //考虑到与dp做兼容，从1开始.

        cin>>m;

        if(m<5){cout<<m<<endl; continue;}

        m -= 5;

        sort(arr+1,arr+n+1); // 对价格排序

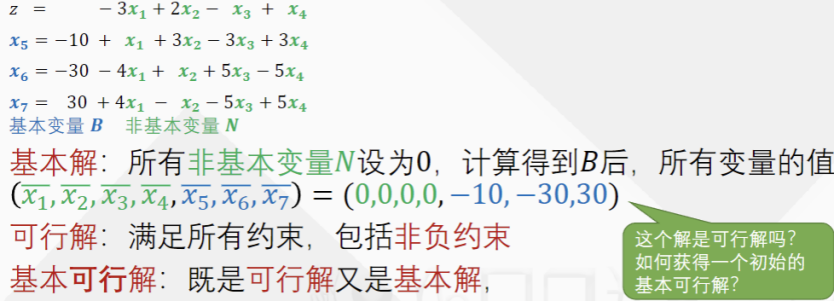
        int max\_price = arr[n]; // 得最大价

        for(int i = 1; i < n; i++) // 考虑第i件物品,到第n-1件，最后一件价格最大另外处理

            for(int j = 0; j <= m; j++) //余额

                if(j>=arr[i])

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

                else

                    dp[i][j] = dp[i-1][j];

        ans = m+5-dp[n-1][m]-max\_price;

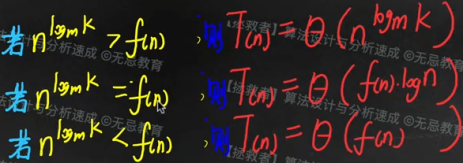
        cout << ans << endl;

**开机I**

int ans = 0, n, temp, arr[1010];

int main(){

    arr[0] = 0;

    cin>>n;

    for(int i = 1; i <= n; i++) cin>>arr[i];

    sort(arr,arr+n+1);

    for(int i = 1; i <= n; i++)

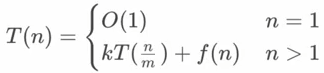
        ans += (temp += arr[i] + arr[i-1]);

    cout<<ans<<endl;

**合适的顺序**

int calculateWeightSum() { // 计算当前排列的权值和

    int sum = 0;

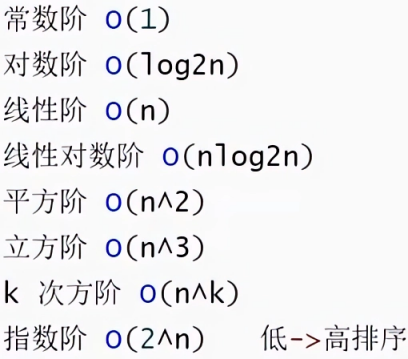
    for (int i = 1; i <= 8; ++i)

        sum += arr[i-1] \* arr[i] \* arr[i + 1];

    return sum;

}

int main() {

    while(cin >> arr[1]){ // 读入8个数

        for (int i = 2; i <= 8; ++i) cin >> arr[i];

        arr[0]=arr[9]=1;

        // 对所有排列进行计算

        int maxWeightSum = 0;

        sort(arr+1, arr+9);

        do {

            int currentWeightSum = calculateWeightSum();

            if (currentWeightSum > maxWeightSum)

                maxWeightSum = currentWeightSum;

        } while (next\_permutation(arr+1, arr+9));

        cout << maxWeightSum << endl;

**算法**是一系列解决问题的清晰指令准确而完整的描述。**特性**：可行性：由若干有限时间结束的基本操作组成；有穷性：执行有限次基本操作后终止；确定性：同样的输入输出相同的结果；有输入；有输出。》**最坏时间复杂度**：最坏情况下的运算执行次数；**平均时间复杂度**：根据输入的概率分布计算平均所需运算次数。》用定义域为自然数集N的函数来定义算法的渐进运行时间》**程序**：是算法用某种程序设计语言的具体实现。**程序与算法的区别**:程序可以不满足算法的第四点性质即有限性。》**分治**：递归地调用自身解决紧密相关的若干子问题，**步骤**：1，分解原问题为若干不相交的子问题；2，解决这些子问题，递归进行；3，合并这些子问题的解为原问题的解。**优化策略**：用一部分子问题的解表达另一部分子问题，减少计算子问题的个数》**贪心**：每次选择局部最优；证明贪心有反证法、归纳法、交换论证法。

**渐进上界符号O**：如果存在正的常数C和自然数 N0，使得当 N >= N0 时有 f(N)**<=**Cg(N)，则称函数 f(N) 当 N 充分大时上有界,且g(N)是它的一个上界，记为 T(N)= O(g(N))。

**渐进下界符号**Ω：如果存在正的常数C和自然数 N0，使得当 N >= N0 时有 f(N)**>=** Cg(N)，则称函数 f(N) 当 N 充分大时下有界且 g(N) 是它的一个下界，记为 T(N)= Ω(g(N))。

**符号θ：**若存在3个正常数C1，C2，N0,且N>=N0，有C1g(N)<=f(N)<= C2g(N),T(N)= θ(g(N))

**o记号：**f(n)=o(g(n))表示对任意正数c都存在常数n,使得对一切n≥n0有0≤f(n)< cg(n)

**ω记号：**f(n)=ω(g(n))表示对任意正数c都存在常数n0使得对一切n≥n0有0≤cg(n)<f(n)

**分治法的时间复杂性分析:**将规模为n的问题分成k个规模为n/m的子问题来求解。设分解阈值n0= 1，且用解最小子问题的算法求解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及将k个子问题的解合并为原问题的解需用 f(n)个单位时间。用 T(n)表示该分治法解规模为 n 的问题所需的计算时间。

