基 础 算 法

谢秋锋

长沙市长郡中学

2017 年 7 月 7日

递推

基本概念

递推算法是一种简单的算法，即通过已知条件，利用特定关系得出中间推论，直至得到结果的算法。
递推中的每个状态由先前已经得到的状态和特定的递推关系确定，逐个求出直到得到最终状态。
OI 中常见的题型为简单的动态规划问题，转移方程即为递推关系。
一般来说会配合其它知识一起出题，并使用一些优化方法来优化递推效率。

NOIP2016 组合数问题

Problem

组合数 $\binom{n}{m}$ 表示的是从 n 个物品中选出 m 个物品的方案数。举个例子，从 $(1,2,3)$ 三个物品中选择两个物品可以有 $(1,2),(1,3),(2,3)$ 这三种选择方法。
根据组合数的定义，我们可以给出计算组合数 $\binom{n}{m}$ 的一般公式：

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

其中 $n! = 1 \times 2 \times \cdots \times n$ 。
小葱想知道如果给定 n, m 和 k ，对于所有的 $0 \leq i \leq n, 0 \leq j \leq min(i, m)$ 有多少对 (i, j) 满足 $\binom{n}{m}$ 是 k 的倍数。

Data Range

每个测试点共有 t 组数据，对于这些数据 k 的值相同。
对于 100% 的数据，$n, m \leq 2000, k \leq 21, t \leq 10^4$ 。

NOIP2016 组合数问题

组合数有一个经典的递推公式：

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

很好理解，就是表示这 n 个物品中第一个选/不选的方案数之和。$\binom{n-1}{m}$ 为不选，$\binom{n-1}{m-1}$ 为选。

NOIP2016 组合数问题

也可以通过数学方法验证：

$$\begin{aligned}
\binom{n-1}{m}+\binom{n-1}{m-1} &= \frac{(n-1)!}{m!(n-m-1)!}+\frac{(n-1)!}{(m-1)!(n-m)!} \\
&= \frac{(n-1)!(n-m)}{m!(n-m)!}+\frac{(n-1)!m}{m!(n-m)!} \\
&= \frac{(n-1)!n}{m!(n-m)!} \\
&= \binom{n}{m}
\end{aligned}$$

NOIP2016 组合数问题

所以我们可以在 $O(nm)$ 的时间内求出所需的所有组合数。
但是组合数是阶乘级别的，存不下，如何快速判断这些组合数能否被 k 整除？

NOIP2016 组合数问题

所以我们可以在 $O(nm)$ 的时间内求出所需的所有组合数。

但是组合数是阶乘级别的，存不下，如何快速判断这些组合数能否被 k 整除？

只要递推时计算它们在模 k 意义下的值即可。那么结果为 0 就表示被 k 整除。

然后新建一个 $n \times m$ 的数组，若 $\binom{n}{m}$ 能被 k 整除则第 n 行第 m 列标记为 1，否则为 0，并计算它的二维前缀和，就可以 $O(1)$ 回答了。

NOIP2016 组合数问题

```
int main(){
    int T=gi(),k=gi();
    for(int i=0;i<=2000;i++){
        c[i][0]=1;
        for(int j=1;j<=i;j++)c[i][j]=(c[i-1][j-1]+c[i-1][j])%k;//求出模 k 意义下的组合数
    }
    for(int i=0;i<=2000;i++)
        for(int j=0;j<=i;j++)b[i][j]=bool(c[i][j])^1;//标记
    for(int i=0;i<=2000;i++)
        for(int j=1;j<=2000;j++)b[i][j]+=b[i][j-1];
    for(int i=1;i<=2000;i++)
        for(int j=0;j<=2000;j++)b[i][j]+=b[i-1][j];//处理前缀和
    while(T--){
        int x=gi(),y=gi();
        printf("%d\n",b[x][y]);
    }
    return 0;
}
```

BZOJ4300 绝世好题

Problem

给定一个长度为 n 的数列 a，求 a 的子序列 b 的最长长度 len，使得在 b 序列中 $b_i \& b_{i-1} \neq 0 (2 \leq i \leq len)$。其中 $\&$ 为按位与运算。

Data Range

$n \leq 100000, a_i \leq 2 \times 10^9$。

BZOJ4300 绝世好题

显然，我们需要递推确定以每个 a_i 结尾的满足条件的 b 的最长长度 l_i。

BZOJ4300 绝世好题

显然，我们需要递推确定以每个 a_i 结尾的满足条件的 b 的最长长度 l_i。
根据题意，对每个 b_i 存在限制的只有它的前一个数。
那么当当前 a_i 作为 b_{len} 时只要查询 l_j 最大的 a_j 满足将 a_j 作为 b_{len-1} 时符合条件。
此时 $l_i = len = l_j + 1$ 。

BZOJ4300 绝世好题

根据题意，满足条件的 b_{len-1} 需要满足在二进制中至少存在某一位使得 b_{len} 和 b_{len-1} 均为 1 。

那么对于每个二进制位分开考虑，记录之前每一位是 1 的 a_j 的最大 l_j ，然后每一次在当前数有 1 的位置查询并取 max 即可。

最终答案为所有位置的 l 值的最大值。

复杂度 $O(nlog(2 \times 10^9))$ 。

BZOJ4300 绝世好题

```
const int K=31;
int main(){
    int n=gi(),ans=0;
    for(int i=1;i<=n;i++){
        int x=gi(),f=1;//f 即为 l_i
        for(int j=0,p=1;j<K;j++,p<<=1)if(x&p)f=max(f,g[j]+1);//使用满足
        for(int j=0,p=1;j<K;j++,p<<=1)if(x&p)g[j]=max(g[j],f);//更新记录
        ans=max(ans,f);//更新最终答案
    }
    printf("%d",ans);
    return 0;
}
```

斐波那契数列

Problem

定义斐波那契数列 f ，$f_0 = 0, f_1 = 1, f_i = f_{i-1} + f_{i-2} (i \geq 2)$ ，**给定** n ，**求** $f_n \bmod (10^9 + 7)$ 。

Data Range

$n \leq 10^{18}$ 。

斐波那契数列

斐波那契数列的递推式已知，但在本题中 n 有 10^{18} ，$O(n)$ 求解显然会超时。

我们需要的只有 f_n 这一个数。如何加速递推过程？

斐波那契数列

斐波那契数列的递推式已知，但在本题中 n 有 10^{18} ，$O(n)$ 求解显然会超时。

我们需要的只有 f_n 这一个数。如何加速递推过程？

需要用到一个新的工具——矩阵快速幂。

矩阵快速幂

矩阵是一个含有 $n \times m$ 个元素的一个方阵，可以理解为一个二维数组。

矩阵乘法的规则：$(A \times B)_{i,k} = \sum_j A_{i,j} \times B_{j,k}$ 。

矩阵快速幂

矩阵是一个含有 $n \times m$ 个元素的一个方阵，可以理解为一个二维数组。
矩阵乘法的规则：$(A \times B)_{i,k} = \sum_j A_{i,j} \times B_{j,k}$ 。
对于一个递推式，我们可以设计一个含有当前状态的状态矩阵 S 和一个转移矩阵 T ，使得 $S \times T$ 可以得到下一状态的状态矩阵。
这样，由于矩阵乘法具有结合律，我们就可以通过快速幂计算 $S \times T^n$ 的矩阵来得到第 n 个状态。

矩阵快速幂

常见的矩阵设计方法为：状态矩阵为一个大小为 $1 \times n$ 的矩阵，转移矩阵大小为 $n \times n$ ，那么 $S_{1,i}$ 就表示当前状态记录的第 i 个值，$T_{i,j}$ 就是 $S_{1,i}$ 向 $S'_{1,j}$ 的转移系数，也就是说写成递推式后 $S'_{1,j}$ 中含有几倍 $S_{1,i}$ 。

矩阵快速幂

举例来说，可以令斐波那契数的状态矩阵 S 为

$$\begin{pmatrix} f_i & f_{i-1} \end{pmatrix}$$

该矩阵对于一个状态记录了两个值：当前斐波那契数和上一个斐波那契数。

矩阵快速幂

举例来说，可以令斐波那契数的状态矩阵 S 为

$$\begin{pmatrix} f_i & f_{i-1} \end{pmatrix}$$

该矩阵对于一个状态记录了两个值：当前斐波那契数和上一个斐波那契数。

那么转移矩阵 T 为

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

矩阵快速幂

可以得到

$$
\begin{aligned}
S \times T &= \begin{pmatrix} f_i & f_{i-1} \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\
&= \begin{pmatrix} f_i + f_{i-1} & f_i \end{pmatrix} \\
&= \begin{pmatrix} f_{i+1} & f_i \end{pmatrix} \\
&= S'
\end{aligned}
$$

矩阵快速幂

这样，我们求出 $S \times T^n$ 后就可以直接得到答案了。

在矩阵乘法时要注意取模。

斐波那契数列

```
const int MOD=1e9+7;
struct matrix{
    int a[2][2];
    matrix(){memset(a,0,sizeof(a));}
    int* operator [](int x){return a[x];}
    matrix operator *(matrix &b){
        matrix c;
        for(int i=0;i<2;i++)
            for(int j=0;j<2;j++)
                for(int k=0;k<2;k++)c[i][k]=(c[i][k]+1ll*a[i][j]*b[j][k])%MOD;
        return c;
    }
}S,T;
int main(){
    lol n=gl();
    S[0][1]=1;//开始时可认为 f_0=0,f_{-1}=1
    T[0][0]=T[0][1]=T[1][0]=1;
    while(n){if(n&1)S=S*T;T=T*T;n>>=1;}//快速幂
    printf("%d",S[0][0]);
    return 0;
}
```

等比数列求和

Problem

给定一个等比数列的第一项 a_1 和公比 q，求其前 n 项的和对 m 取模的结果。

Data Range

$n \leq 10^{18}, 1 \leq m \leq 10^9$ 。

保证 a_1, q 的绝对值不超过 m 。

等比数列求和

仿照斐波那契数列的求法，我们先写出等比数列的递推式。

令 s_i 为其前 i 项的和，有：

$a_i = q \times a_{i-1}, s_i = s_{i-1} + a_i$ 。

等比数列求和

这样，我们可以对于每一个状态记录当前和与等比数列的下一项。即

$$S = \begin{pmatrix} s_i & a_{i+1} \end{pmatrix}$$

那么容易得到

$$T = \begin{pmatrix} 1 & 0 \\ 1 & q \end{pmatrix}$$

等比数列求和

有

$$\begin{aligned} S \times T &= \begin{pmatrix} s_i & a_{i+1} \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 1 & q \end{pmatrix} \\ &= \begin{pmatrix} s_i + a_{i+1} & q \times a_{i+1} \end{pmatrix} \\ &= \begin{pmatrix} s_{i+1} & a_{i+2} \end{pmatrix} \\ &= S' \end{aligned}$$

即可。

等比数列求和

```
struct matrix{
    int a[2][2];
    matrix(){memset(a,0,sizeof(a));}
    int* operator [](int x){return a[x];}
    matrix operator *(matrix &b){
        matrix c;
        for(int i=0;i<2;i++)
            for(int j=0;j<2;j++)
                for(int k=0;k<2;k++)c[i][k]=(c[i][k]+1ll*a[i][j]*b[j][k])%m;
        return c;
    }
}S,T;
int main(){
    lol n=gl();m=gi();
    S[0][1]=(gi()+m)%m;//读入 a_1
    T[1][1]=(gi()+m)%m;//读入 q
    T[0][0]=T[1][0]=1;
    while(n){if(n&1)S=S*T;T=T*T;n>>=1;}//快速幂
    printf("%d",S[0][0]);
    return 0;
}
```

HNOI2011 数学作业

Problem

小 C 数学成绩优异，于是老师给小 C 留了一道非常难的数学作业题：给定正整数 N 和 M，要求计算 $Concatenate(1..N) \bmod M$ 的值，其中 $Concatenate(1..N)$ 是将所有正整数 $1, 2, ..., N$ 顺序连接起来得到的数。例如，$N = 13$，$Concatenate(1..N) = 1234567891011213$ 。
小 C 想了大半天终于意识到这是一道不可能手算出来的题目，于是他只好向你求助，希望你能编写一个程序帮他解决这个问题。

Data Range

$n \leq 10^{18}, 1 \leq M \leq 10^9$ 。

HNOI2011 数学作业

首先写出递推关系式：

令 s_i 为前 i 个数连接得到的数，$c(i)$ 表示 i 的位数，有

$$s_i = 10^{c(i)} \times s_{i-1} + i$$

HNOI2011 数学作业

$c(i)$ 随 i 变化，无法在转移矩阵 T 中表示。

如果 $c(i)$ 固定，就可以套用之前的方法进行优化了！

HNOI2011 数学作业

考虑位数随着 i 的变化最多只会变化 18 次，因此可以按位数分段进行矩阵快速幂。

枚举位数，那么有

$$S = \begin{pmatrix} s_i & i+1 & 1 \end{pmatrix}$$

$$T = \begin{pmatrix} 10^{c(i+1)} & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

这里在状态矩阵中加了一个永远为 1 的值，用于辅助 i 每次加一。

HNOI2011 数学作业

```
int main(){
    lol n=gl(),now=0;mod=gl();
    S[0][1]=S[0][2]=1;
    for(lol s=10;now<n;s*=10){
        T[1][0]=T[1][1]=T[2][1]=T[2][2]=1;
        T[0][1]=T[0][2]=T[1][2]=T[2][0]=0;
        T[0][0]=s%mod;
        lol nd=min(s-1,n)-now;
        while(nd){
            if(nd&1)S=S*T;
            T=T*T;
            nd>>=1;
        }
        now=min(s-1,n);
    }
    printf("%d",S[0][0]);
    return 0;
}
```

贪心

基本概念

贪心算法是指，在对问题求解时，总是做出在当前状态来说最优的决策，而不考虑全局最优。
这样做可以省去全局分析所需要的步骤，然而仅当决策不具有后效性时才能保证对全局来说该决策也最优的。
如果一个问题能够使用贪心算法得到最优解，那么也称该问题具有最优子结构。

石子合并问题

Problem

给定 n 堆石子，每堆石子有若干个石子。现要将这些石子合并为一堆，每次可以合并任意两堆，合并的代价为合并得到的新石子堆的石子个数。问如何合并使得代价最小。

石子合并问题

这个问题的本质即为 Huffman 编码问题。合并的方法即为 Huffman 树的构造方法。
可以证明每次贪心地选择两堆最小的石子合并是最优的。
用堆维护最小石子堆即可。也可以用两个有序数组将除排序外的复杂度优化到 $O(n)$ 。

NOIP2013 积木大赛

Problem

春春幼儿园举办了一年一度的“积木大赛”。今年比赛的内容是搭建一座宽度为 n 的大厦，大厦可以看成由 n 块宽度为 1 的积木组成，第 i 块积木的最终高度需要是 h_i 。

在搭建开始之前，没有任何积木（可以看成 n 块高度为 0 的积木）。接下来每次操作，小朋友们可以选择一段连续区间 $[L, R]$ ，然后将第 L 块到第 R 块之间（含第 L 块和第 R 块）所有积木的高度分别增加 1 。

小 M 是个聪明的小朋友，她很快想出了建造大厦的最佳策略，使得建造所需的操作次数最少。但她不是一个勤于动手的孩子，所以想请你帮忙实现这个策略，并求出最少的操作次数。

Data Range

$1 \leq n \leq 100000, 0 \leq h_i \leq 10000$ 。

NOIP2013 积木大赛

考虑最优的建造策略：
每次选择最左边的一个还未搭好的积木，将其高度加一，并贪心地让右边尽可能多的积木高度也加一。
这样做显然不会使结果变差。

NOIP2013 积木大赛

计算方案：
考虑每一个积木作为最左端高度加一的次数。
当该积木比左边一个矮时显然次数为 0 。在搭建左边一个时一定将这个也搭建完成了。
当该积木比左边一个高时，还需要的次数即为左边不能提供给它的高度，也即它比左边一个高出的高度。

NOIP2013 积木大赛

```
int main(){
    int n=getint(),ans=0;
    for(int i=1;i<=n;i++){
        a[i]=getint();
        if(a[i]>a[i-1])ans+=a[i]-a[i-1];//高度差
    }
    cout<<ans;
    return 0;
}
```

大炮

Problem

“农气大炮”是 Chanxer 毕生精力凝结而成的心血。
“当大炮建成时，普天之下，莫非农土!”，现在 Chanxer 还有最后一步，就是为大炮装载农气续航系统，换句话说，就是上电池。
Chanxer 现在有 n 条能量棒，第 i 条的长度为 2^{k_i}，且拥有一个不稳定值 W_i 。
为了避免因为单次高强度攻击而导致大炮瘫痪，Chanxer 觉得采用能源分离装置，他准备了许多能量槽，每个能量槽都是条型的，横截面积恰好容得下一根能量棒插入，而能量槽的深度也是 2^t 的形式。
现在，Chanxer 需要在能量棒中选择一些插入能量槽中，为了保证续航时间的最大化，Chanxer 要求每个能量槽都必须完全装满，也就是说，插入其中的能量棒的总长度要正好等于能量槽的深度，与此同时，Chanxer 希望被选择的能量棒的总不稳定值最小。

Data Range

$1 \leq n \leq 10000, 0 \leq k_i \leq 1000, 0 \leq W_i \leq 10000, 0 \leq t \leq 1000$ ，能量槽总数不超过 5000 。

大炮

首先分析一下能量棒：两个长度为 2^k 的能量棒可以当作一个长度为 2^{k+1} 的能量棒使用。

因为小的能量棒可以填大的能量槽而大的不能填小的，所以我们可以考虑贪心地从小的能量槽开始往大的填。

大炮

对于每一种长度的能量槽，贪心地用该长度能量棒中代价最小的那些填充。剩下的再按照代价从小到大的顺序两两拼成更大的能量棒（如果有奇数个则扔掉最大的）。
显然拼起来对之后的决策不会有影响。因为不可能有更大的能量槽中用了奇数根该长度的能量棒。
注意如果某种长度没有能量槽也要把该长度的能量棒拼起来。

大炮

```
int main(){
    int n=gi();
    for(int i=1;i<=n;i++)c[i].x=gi(),c[i].y=gi();//能量棒的大小、代价
    sort(c+1,c+n+1);//大小为第一关键字，代价为第二关键字排序能量棒
    int m=gi(),pc=1,pd=1,ans=0;
    for(int i=1;i<=m;i++)d[i].x=gi(),d[i].y=gi();//题目要求的读入格式：能量槽的深度、该深度能量槽个数
    sort(d+1,d+m+1);//按深度排序能量槽
    for(int h=0;h<=1000;h++){//枚举深度
        int nt=0;for(int pp=1;pp+1<=t;pp+=2)p[++nt]=q[pp]+q[pp+1];//将上一深度剩下的能量棒合并
        int sb=0;while(pc<=n&&c[pc].x==h)tmp[++sb]=c[pc++].y;//将当前大小的能量棒提出
        t=nt+sb;int c[2]={1,1};
        for(int i=1;i<=t;i++){
            if(c[0]>nt||(c[1]<=sb&&tmp[c[1]]<p[c[0]]))q[i]=tmp[c[1]++];
            else q[i]=p[c[0]++];
        }//归并式排序
        reverse(q+1,q+t+1);
        for(;pd<=m&&d[pd].x==h;pd++){
            if(t<d[pd].y){printf("-1");return 0;}
            while(d[pd].y--)ans+=q[t--];
        }//按能量棒代价从小往大填充能量槽
        reverse(q+1,q+t+1);
    }
    printf("%d",ans);
    return 0;
```

总结

贪心算法的使用需要对问题的细致分析，大胆猜想，小心求证。
一个贪心结论的成立往往可以大幅提高程序运行的效率。
很多题目的分析都需要在部分步骤上运用贪心的思想来推进思路。
错误的贪心结论有时也可以引导思路，或者起到骗分的作用。
因此解题时活用贪心思想十分重要。

分治

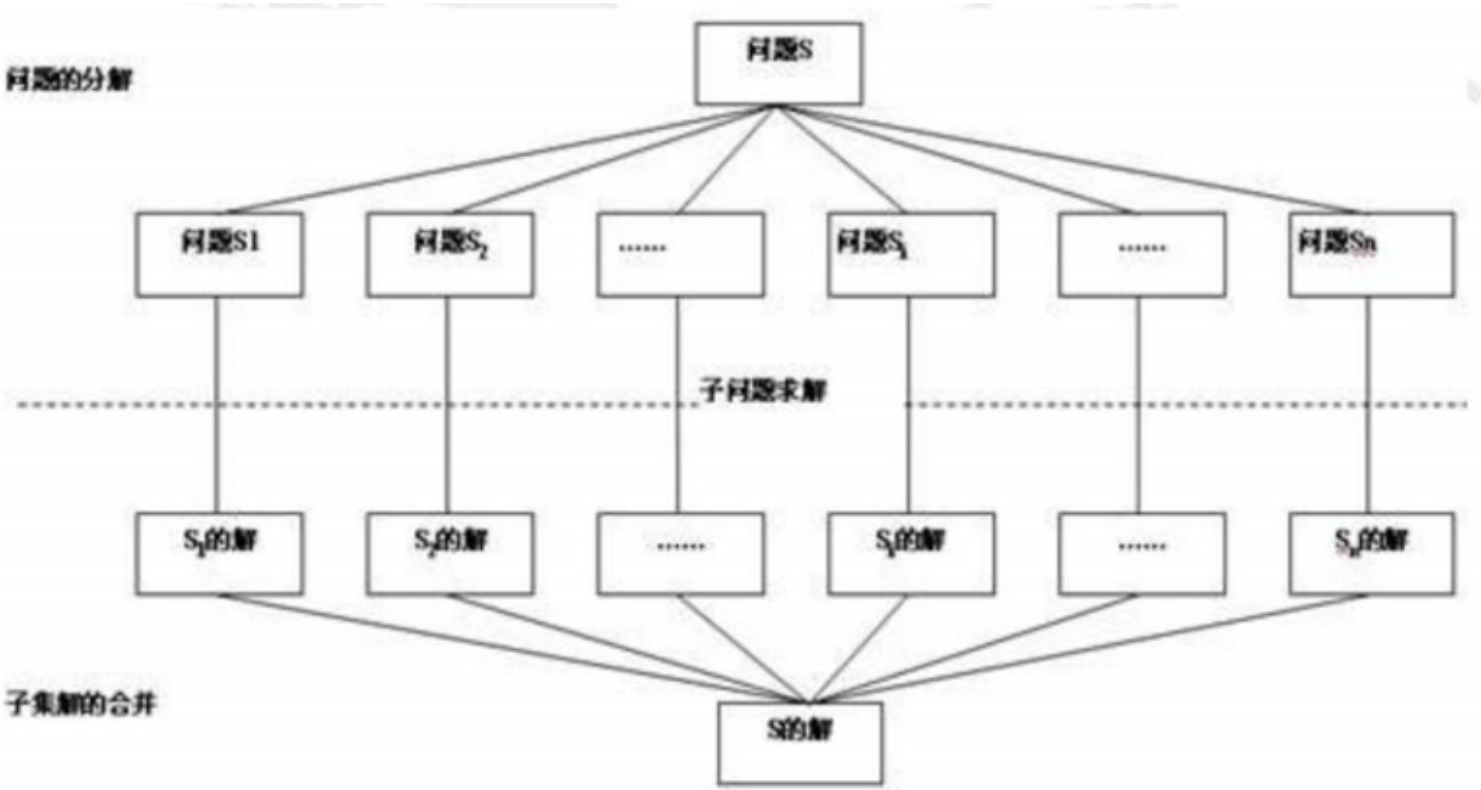
基本概念

分治法，又叫分治策略，顾名思义，分而治之。
它的基本思想：对于难以直接解决的规模较大的问题，把它分解成若干个能直接解决的相互独立的子问题，递归求出各子问题的解，再合并子问题的解，得到原问题的解。
通过减少问题的规模，逐步求解，能够明显降低解决问题的复杂度。

框架结构

```
int DAC(){
    if(问题不可分){
        直接求解;
        return ans;
    }else{
        分解该问题;//分解
        对分解出的每一部分递归求解;//解决
        由各个部分的解合并得出该问题的解;//合并
        return ans;
    }
}
```

过程图


问题的分解
问题S
问题S1
问题S2
.....
问题Si
.....
问题Sn
子问题求解
S1的解
S2的解
.....
Si的解
.....
Sn的解
子集解的合并
S的解

归并排序

归并排序是一种使用分治思想的高效排序方法。
归并排序的流程为：
将原序列平分成两半（分解）；
分别对分出的两个序列进行归并排序（解决）；
将两个有序序列合并为一个有序序列（合并）。

归并排序

合并有序序列的方法为：每次比较两个有序序列的最小值，将较小者加入新有序序列并从原序列删除。

归并排序

```
int a[MAXN],b[MAXN];//a 为需要排序的数组，b 为辅助数组
void merge(int l,int r){
    if(l==r)return;
    int m=(l+r)>>1;
    merge(l,m);
    merge(m+1,r);
    int i=l,j=m+1;
    for(int k=l;k<=r;k++)
        if(i<=m&&(j>r||a[i]<a[j]))b[k]=a[i++];
        else b[k]=a[j++];
    for(int i=l;i<=r;i++)a[i]=b[i];
}
```

平面最近点对

Problem

给定平面上的 n 个点，求欧几里德距离最近的两个点的距离。

Data Range

$n \leq 100000$ 。

平面最近点对

直接两两求距离是 $O(n^2)$ 的，不能解决。

考虑分治。

平面最近点对

将所有点按照横坐标排序，然后每次均匀分成两半。
那么最近点对要么在其中一半，要么每半各一个点组成点对。

平面最近点对

将所有点按照横坐标排序，然后每次均匀分成两半。
那么最近点对要么在其中一半，要么每半各一个点组成点对。
每次我们可以先递归分治求出两边的最近点对距离 ans 。
然后将距离分割线（就是之前把点分成两半的位置）小于 ans 的点提出（否则距离到对面的点一定大于 ans ）。
再将这些点左右分别按照纵坐标排序，枚举某一侧的点，和另一侧与它纵坐标差小于 ans 的点计算距离更新答案即可。

平面最近点对

由于保证了两侧点对距离不会小于 ans，所以与一个点纵坐标差小于 ans 的点实际上只会枚举不到 6 个（所枚举点的范围为另一侧一个高为 $2ans$ 宽为 ans 的长方形，其中最多放 6 个点使得两两距离不小于 ans 。而实际上因为要求差小于 ans 所以是小于 6 个）。

实现时可以用一个单调指针维护。

该算法的总时间复杂度为分治的复杂度 $O(nlogn)$ 。

```
struct point{
    double x,y;
    bool operator <(const point b) const{return x<b.x;}
}a[MAXN],b[MAXN];
#define sqr(x) ((x)*(x))
void work(int l,int r){
    if(l==r)return;
    int m=l+r>>1;double cen=a[m].x;
    work(l,m);work(m+1,r);//递归后两侧点分别有序
    int t=0,p=1;
    for(int i=m+1;i<=r;i++)if(a[i].x-cen<ans)b[++t]=a[i];//提出右侧横坐标距离小于 ans 的
    for(int i=l;i<=m;i++){
        if(cen-a[i].x>=ans)continue;
        while(p<=t&&a[i].y-b[p].y>=ans)p++;//p 为单调指针，指向右侧纵坐标最小的可行点
        for(int j=p;j<=t&&b[j].y-a[i].y<ans;j++)ans=min(ans,sqrt(sqr(a[i].x-b[j].x)+sqr(a[i].y-b[j].y)));
    }
    for(int i=l,p1=l,p2=m+1;i<=r;i++)
        if(p1<=m&&(p2>r||a[p1].y<a[p2].y))b[i]=a[p1++];
        else b[i]=a[p2++];
    for(int i=l;i<=r;i++)a[i]=b[i];//使用归并排序实现将点按照 y 坐标排序，保证复杂度
}
int main(){
    int n=gi();
    for(int i=1;i<=n;i++)scanf("%lf%lf",&a[i].x,&a[i].y);
    sort(a+1,a+n+1);ans=1e20;work(1,n);
    printf("%.6lf\n",ans);
    return 0;
}
```

二分答案

二分答案是很常用的一种分治方法，其通过每次判断某值是否满足某条件来将求极值问题转化为判定问题。

二分答案的思想是：开始时答案可能处于一个大区间 $[l, r]$ 内，那么每一次我们令 $m = \frac{l+r}{2}$ ，然后通过一定方法询问答案和 m 的大小关系。这样，可能的答案区间就缩小了一半。不断如此，只要 $log(r - l + 1)$ 次就可以确定答案。

因此二分答案也需要满足在答案两侧的区间内进行某种询问得到的结果应该是相反的。

框架结构

求合法最小值等

```
int Divide(int l,int r){
    while(l<r){
        int m=l+r>>1;
        if(check(m))r=m;
        else l=m+1;
    }
    return l;
}
```

求合法最大值等

```
int Divide(int l,int r){
    while(l<r){
        int m=l+r+1>>1;
        if(check(m))l=m;
        else r=m-1;
    }
    return l;
}
```

NOIP2001 一元三次方程求解

Problem

有形如：$ax^3+bx^2+cx+d=0$ 这样的一个一元三次方程。给出该方程中各项的系数（a,b,c,d 均为实数），并约定该方程存在三个不同实根（根的范围在 -100 至 100 之间），且根与根之差的绝对值 ≥ 1 。要求由小到大依次在同一行输出这三个实根（根与根之间留有空格），并精确到小数点后 2 位。

提示：记方程 $f(x)=0$ ，若存在 2 个数 x_1 和 x_2 ，且 $x_1<x_2$ ，$f(x_1)\times f(x_2)<0$ ，则在 (x_1,x_2) 之间一定有一个根。

NOIP2001 一元三次方程求解

由于精度要求不高，可以直接枚举答案，找到函数值最接近 0 的三个解即可。
但如果精度要求更高，例如精确到小数点后 8 位，直接枚举显然会超时。

NOIP2001 一元三次方程求解

由于精度要求不高，可以直接枚举答案，找到函数值最接近 0 的三个解即可。

但如果精度要求更高，例如精确到小数点后 8 位，直接枚举显然会超时。

但由于题目保证了根之差的绝对值大于等于 1，说明每两个相邻整数之间最多有一个根，可以利用这一点进行优化。

NOIP2001 一元三次方程求解

首先枚举相邻整数，可以使用题目给出的公式判断这两个整数间是否有解。如果有，问题变为寻找这两个整数之间的解。
此时使用二分法逐渐缩小根的范围，即可得到根在所需精度的数值。

NOIP2001 一元三次方程求解

假设开始时已知区间 $[l, r]$ 中有根，那么每次取区间中点 m 并询问 $[l, m]$ 和 $[m, r]$ 内哪个有根，即可将答案区间缩小一半。
当答案区间长度小于所需精度时即为答案。

NOIP2015 跳石头

Problem

一年一度的“跳石头”比赛又要开始了!
这项比赛将在一条笔直的河道中进行，河道中分布着一些巨大岩石。组委会已经选择好了两块岩石作为比赛起点和终点。在起点和终点之间，有 N 块岩石（不含起点和终点的岩石）。在比赛过程中，选手们将从起点出发，每一步跳向相邻的岩石，直至到达终点。
为了提高比赛难度，组委会计划移走一些岩石，使得选手们在比赛过程中的最短跳跃距离尽可能长。由于预算限制，组委会至多从起点和终点之间移走 M 块岩石（不能移走起点和终点的岩石）。

Data Range

令 L 为起点到终点的距离。
$0 \leq M \leq N \leq 50000, 1 \leq L \leq 10^9$ 。

NOIP2015 跳石头

很多选手看到这道题的第一思路是堆 + 贪心，复杂度 O(nlogn)，然而这样做并不对，可以构造出很多反例。
（启示我们贪心一定要尝试构造反例）

NOIP2015 跳石头

本题的正确做法是二分答案 + 贪心。
每次二分出一个答案 k，从第二块石头到最后一块石头依次判断，如果当前石头离上一块没有被去掉的石头距离小于 k，则说明当前石头需要被去掉。
如果去掉的石头数大于限定的 m 则说明答案小于 k，反之说明答案大于等于 k。

NOIP2015 跳石头

那么为什么这里的贪心又可以了呢？

NOIP2015 跳石头

那么为什么这里的贪心又可以了呢？
在堆 + 贪心的错误算法中，去掉某个石头会对之后的决策有影响，即有后效性。

NOIP2015 跳石头

那么为什么这里的贪心又可以了呢？
在堆 + 贪心的错误算法中，去掉某个石头会对之后的决策有影响，即有后效性。
但这里的贪心不存在这样的问题。因为本算法按顺序扫描，如果当前石头与前面一个石头的距离小于 k，那么一定要去掉其中一个；那么肯定要去掉当前石头而不会去掉前面的石头。因为后面的石头到当前石头的距离肯定小于到上一块石头的距离，去掉当前的一定不会更劣。

NOIP2015 跳石头

那么为什么这里的贪心又可以了呢？
在堆 + 贪心的错误算法中，去掉某个石头会对之后的决策有影响，即有后效性。
但这里的贪心不存在这样的问题。因为本算法按顺序扫描，如果当前石头与前面一个石头的距离小于 k，那么一定要去掉其中一个；那么肯定要去掉当前石头而不会去掉前面的石头。因为后面的石头到当前石头的距离肯定小于到上一块石头的距离，去掉当前的一定不会更劣。
总的时间复杂度 $O(nlogL)$ 。

NOIP2015 跳石头

```
int cac(int s){
    int ans=0,last=0;//last 表示上一个没有被去掉的石头
    for(int i=1;i<=n+1;i++){
        if(d[i]-d[last]<s)ans++;
        else last=i;
    }
    return ans;
}
int t(){
    int l=1,r=L;
    while(l<r){
        int mid=(l+r+1)>>1;//+1 防止死循环
        if(cac(mid)>m)r=mid-1;
        else l=mid;
    }
    return l;
}
int main(){
    L=getint(),n=getint(),m=getint();
    for(int i=1;i<=n;i++)d[i]=getint();
    d[n+1]=L;
    cout<<t();
    return 0;
}
```

总结

分治算法在原问题难以解决而分解成的小问题容易解决，且合并时可以利用解决小问题时得到的信息快速解决原问题时十分有效。
二分答案适用于很多求最值问题，将求最值转化为判定问题。