作业回顾：非编程题

3.二进制称谓

1KB=2^{10}B≈10^{3}B，一篇500字纯文本作文

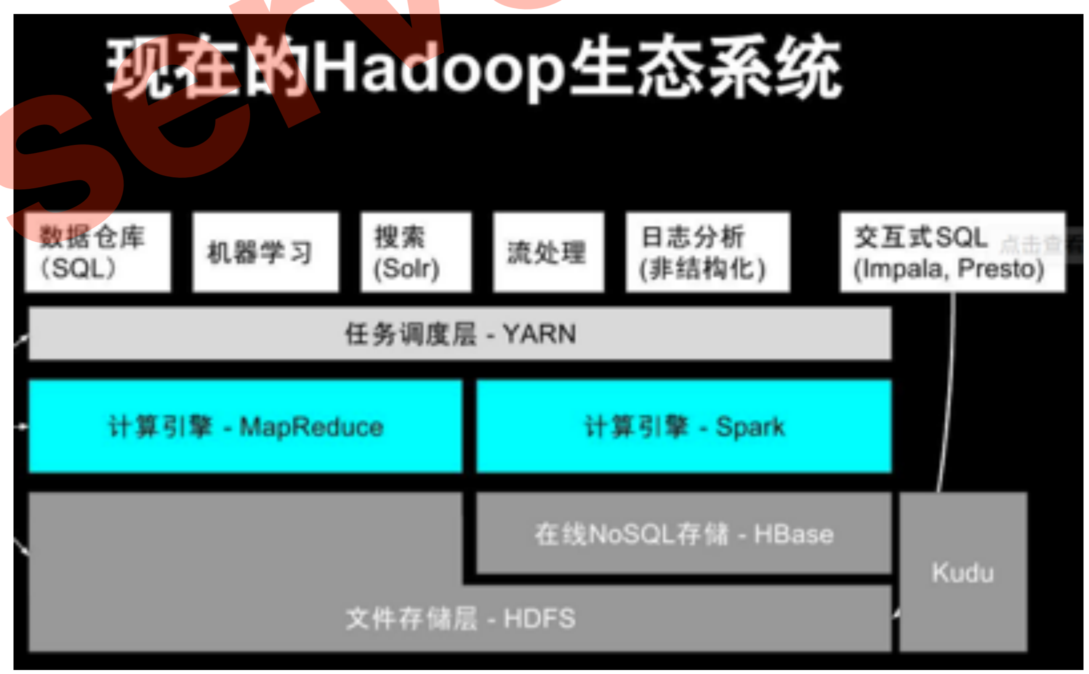
1MB=2^{20}B≈10^{6}B，一张高清照片

1GB=2^{30}B≈10^{9}B，一部高清电影

1TB=2^{40}B≈10^{12}B，一个移动硬盘的容量

1PB=2^{50}B≈10^{15}B，一个中型hadoop集群的总容量

1EB=2^{60}B≈10^{18}B，谷歌云存储总空间



作业回顾：编程题

1.heapsort.cpp

2.bankqueue.cpp

4.onefromthree.cpp

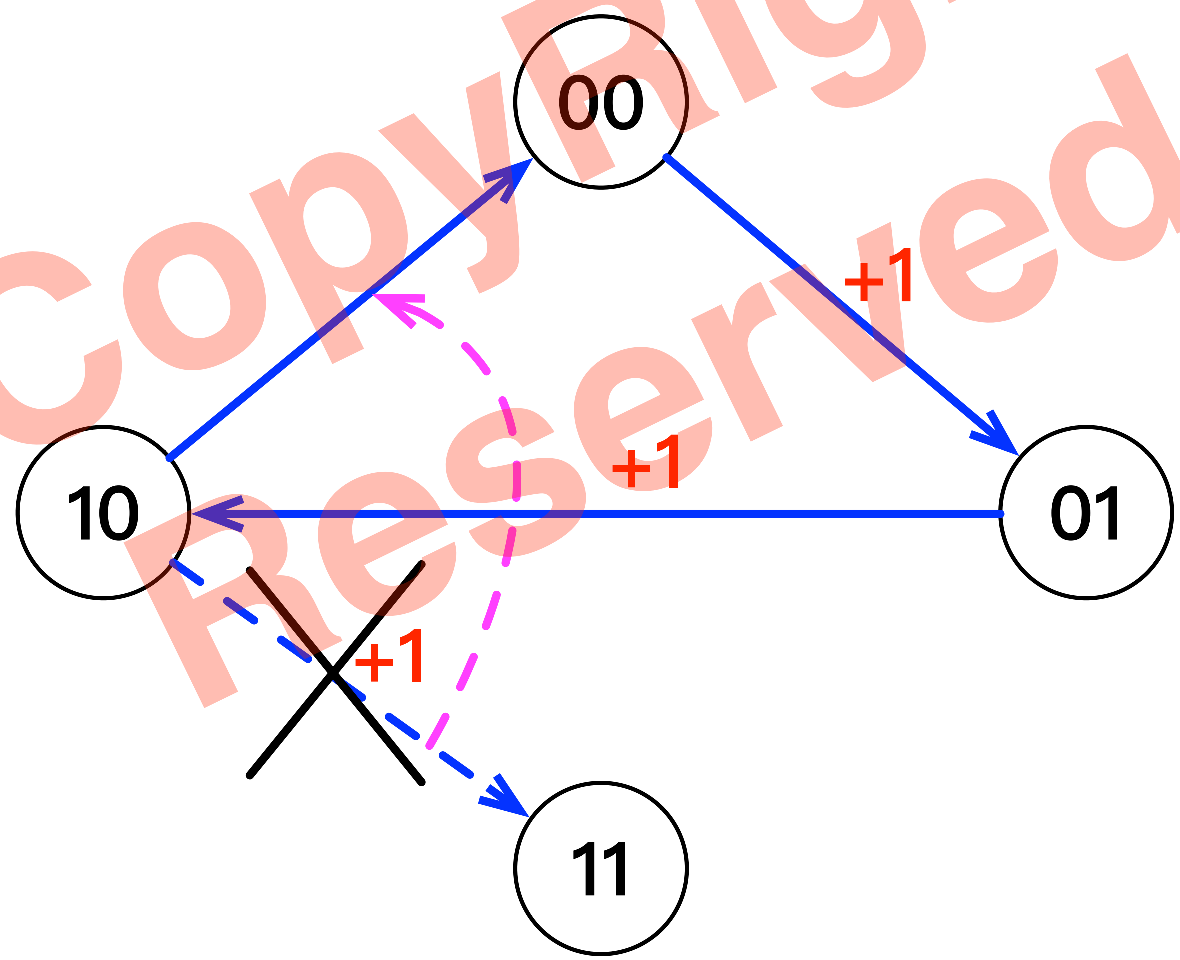
Welcome to Xcode

作业回顾：银行排队问题-思路

```
建立两个堆：
  1.以优先级为序的最大堆（等待者堆）
  2.以时间为顺序的最小堆（事件堆）
输入人的信息，为每个人创建一个到达事件，并进入事件堆
while (事件堆非空) {
  事件堆出堆一个事件event，及其对应的人person
  if (到达事件){
    if (无人在办理){
      person开始办理，输出person.id
      为person生成一个完成事件，时间为当前时间+person.t，进入事件堆
    } else {
      person进入等待者堆
    }
  } else if (完成事件) {
    if (等待者堆非空){
      等待者堆出堆一个人person2，person2开始办理，输出person2.id
      为person2生成一个完成事件，时间为当前时间+person2.t，进入事件堆
    } else{
      窗口空闲（无人办理）
    }
  }
}
```

作业回顾：重复数问题II-思路

→ 处理单独一个bit（也就是只有0和1）
→ 每3个1出现就归零
→ 因为要计数到3，所以需要2个二进制位：higherbits,lowerbits



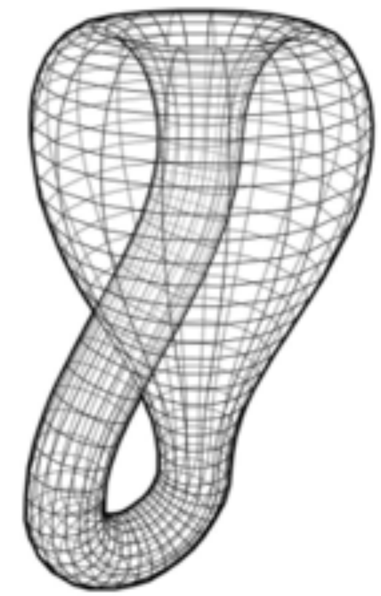
CS100 算法入门

回溯法

先讲一点哲学：状态空间

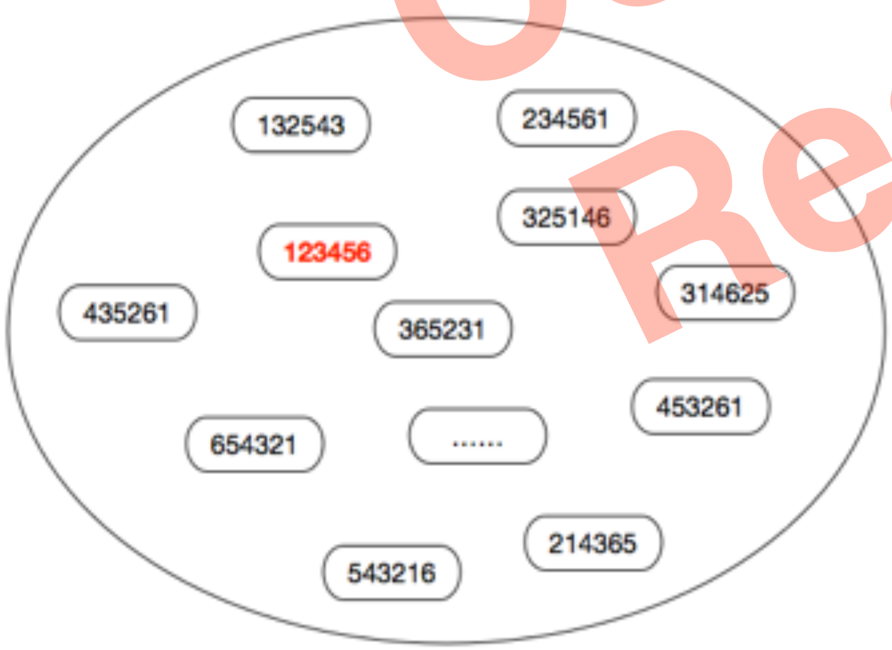
一类(抽象/具体的)对象的全集，称为一个状态空间

这里“空间”是一个广义的概念，不单指三维空间



很多算法问题建模的关键就是

从题目中抽象出合适的状态



一对一状态转移：递推算法

最大和子串问题

输入一个整数序列，输出其中总和最大的连续子序列之和

样例输入：

10

13 -21 11 -31 32 22 -12 33 23 -99

样例输出：

98

（注：对应的子序列为32 22 -12 33 23）

最大和子串问题：思路

输入数组a，数量n

状态选择(关键)：以f[i]表示**以a[i]结尾的子串中最大子串和**

f[0]=0（边界）

for (int i=1;i<=n;i++) {

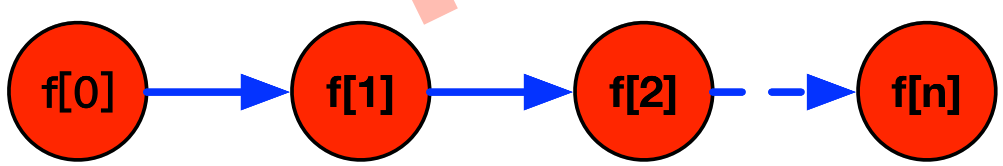
    f[i]=f[i-1]>0 ? f[i-1]+a[i] : a[i]（状态转移）

}

输出max{f[i]|i=0..n}

→ **So Easy，第一周就有人做出来了~~**

状态空间：链表(退化的树)



i	a[i]	f[i]
0		0
1	13	13
2	-21	-8
3	11	11
4	-31	-20
5	32	32
6	22	54
7	-12	42
8	33	75
9	23	98
10	-99	-1

一对多状态转移：回溯法

素数环问题

输入一个整数n，输出所有由1~n排成的环，满足任意两个相邻数之和都是素数

样例输入：

6

样例输出：

1 4 3 2 5 6

1 6 5 2 3 4

（注：输出都以1开头，其他循环的不用输出，如4 3 2 5 6 1）

顺便说一句：

把素数和加法整到一起一定不是容易搞定的事情

例如：哥德巴赫猜想

素数环问题：思路

输入n

状态选择：以环的前i个数为状态（很直观呗）

```
void solve(int a[],int i){
   if (i==n) {
      判断a是否符合素数环条件，符合就输出
   }
   for (a[i]=1;i<=n;a[i]++) {
       solve(a,i+1)
   }
}
```

调用solve(0)

→ **没什么办法，只能枚举1..n的全排列**
→ **但是n是输入的，不可能写n重循环出来**
→ **只能用递归来实现枚举**

→ **solve和search是竞赛中最出现的函数名称**
因为这用这两个函数名一般就是说：“我想不出什么精妙的好办法，只能使用暴力。。。”

素数环问题：辅助函数

```
11  #include "math.h"
12
13  using namespace std;
14
15  int isPrime(int n) {     // 判断质数
16      if (n==1 || n==2 || (n>2 && n%2==0)){
17          return 0;
18      }
19      for (int i=3;i<=sqrt(n);i+=2){
20          if (n%i==0){
21              return 0;
22          }
23      }
24      return 1;
25  }
```

→ 这里为什么行号断档了?

我为了突出算法的逻辑层次，颠倒了贴代码的顺序。你们在写代码的时候必须维持函数的有序调用，也就是写在前面的函数不能调用写在后面的

那两个函数要互相调用（又称为间接递归）怎么办？这时候可以使用函数签名（之前讲到过了）


```
51  void searchForRing(int n){
52      if (n%2==1) {    // 预判，也是一种优化
53          cout<<"奇数是不可能的"<<endl;
54          return;
55      }
56      int data[n+1];
57      for (int j=1;j<=n;j++) {     // 初始化排列
58          data[j]=j;
59      }
60      search(data,n,2);    // 为什么从2开始？因为循环对称性，第1位固定为1
61  }
```

素数环问题：解答

```
// 回溯法遍历
void search(int data[],int n,int i) {
    if (i==n){    // 递归边界，n个数都确定了位置
        if (isPrime(data[n]+data[1])) {
            // 最后判断首尾和是不是素数，是的话就找到一个解
            for (int j=1;j<=n;j++){
                cout<<data[j]<<" ";
            }
            cout<<endl;
        }
        return;
    }

    int tmp=data[i];
    for (int j=i;j<=n;j++){  // 枚举尚未确定的数字
        if (isPrime(data[j]+data[i-1])){ // 剪枝
            data[i]=data[j];    // 交换i,j位的数字
            data[j]=tmp;
            search(data,n,i+1);    // 递归搜索下一个数
            data[j]=data[i];    // 恢复第j位的数字
        }
    }
    data[i]=tmp;    // 恢复第i位的数字
}
```

→ 由于在递归边界要“回头”，所以称为“回溯法”（其实是所有递归算法的共同特征）

→ 回溯法与之前看到的递归的不同在于，每次调用中一般都会多次调用本函数自身

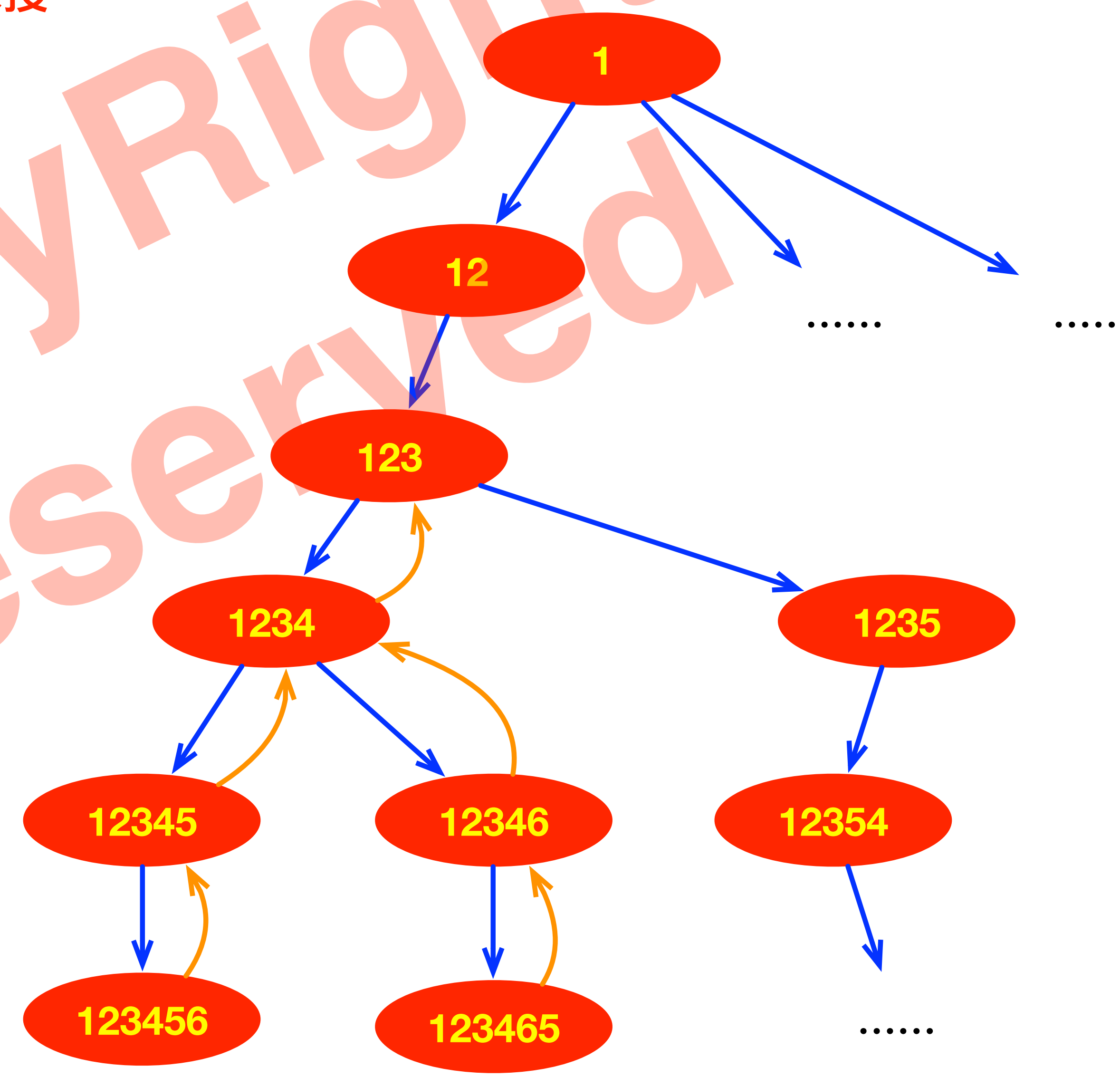
→ 所以总调用次数随递归深度增加往往是指数级的 $O(a^n)$。本质上还是属于暴力枚举

溯洄从之 道阻且长

素数环问题：状态空间

- 把回溯法每次调用画出来，就会形成一棵树，也就是这道题的状态空间
- 回溯法的顺序总是先往“深处”走，直到叶节点（无路可走）才回头
- 所以回溯法也称为深度优先搜索，简称深搜
- 深搜的过程就是遍历状态空间
- 但是没有正真建树，所以也称为隐式遍历

状态总数：6!=720



素数环问题：剪枝

```cpp
// 回溯法遍历
void search(int data[],int n,int i) {
    if (i==n){    // 递归边界，n个数都确定了位置
        if (isPrime(data[n]+data[1])) {
            for (int j=1;j<=n;j++){  // 最后判断首尾和是不是素数，是的话就找到一个解
                cout<<data[j]<<" ";
            }
            cout<<endl;
        }
        return;
    }

    int tmp=data[i];
    for (int j=i;j<=n;j++){  // 枚举尚未确定的数字
        if (isPrime(data[j]+data[i-1])){ // 剪枝
            data[i]=data[j];    // 交换i,j位的数字
            data[j]=tmp;
            search(data,n,i+1);    // 递归搜索下一个数
            data[j]=data[i];    // 恢复第j位的数字
        }
    }
    data[i]=tmp;    // 恢复第i位的数字
}
```

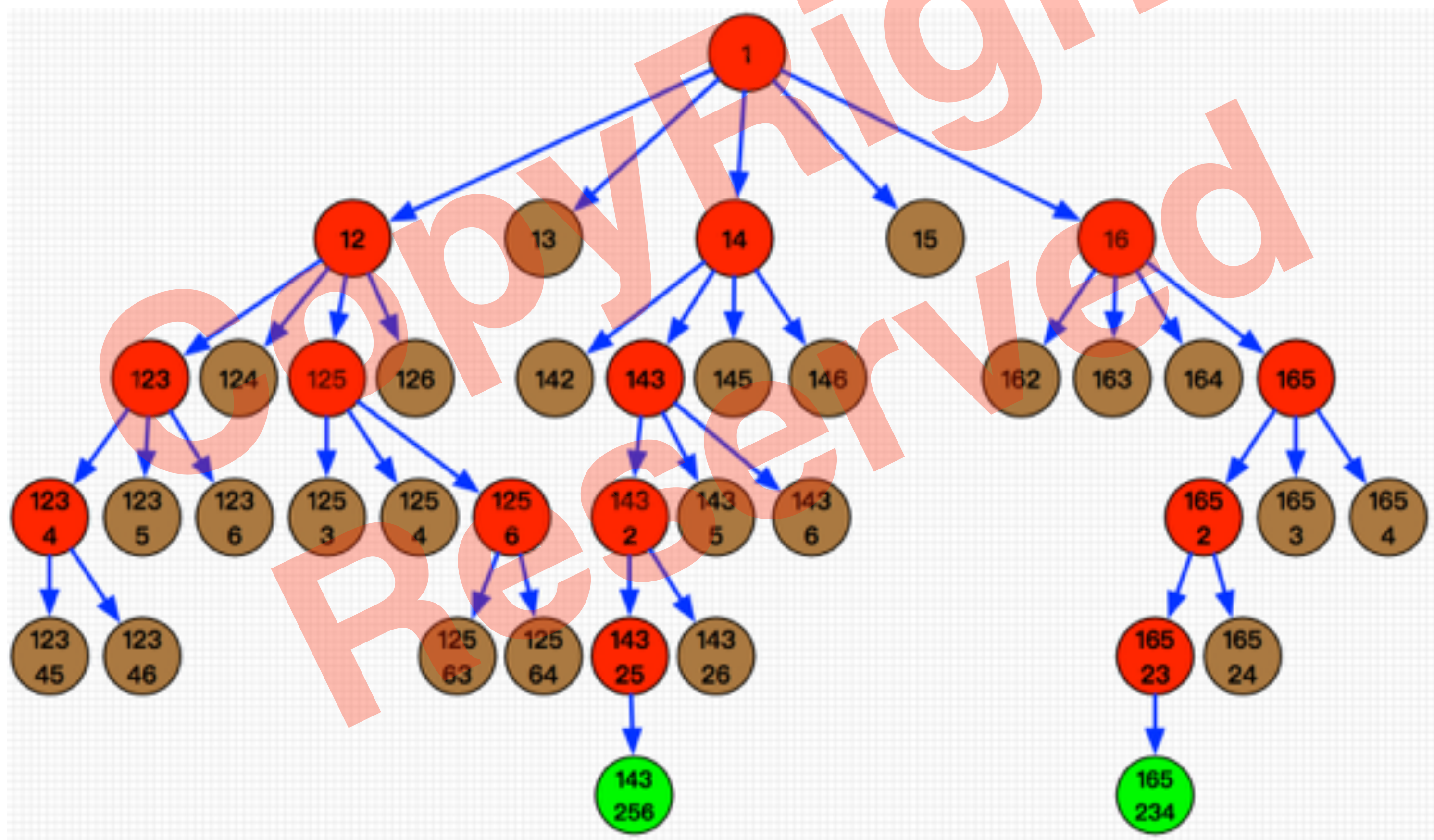
→ 没必要每次都到边界在回头

→ 中途发现有些和不是素数，就不需要再枚举下去了

→ 这种**提前回头**的优化方式称为**剪枝**

剪枝的效果

- → 回溯法试图遍历整个状态空间，时间效率是比较差的
- → 但由于剪枝的存在，实际速度可以大大提高
- → 因此**使用回溯法的关键就在于剪枝做的好不好**



剪枝效率：6!/42≈17倍

素数环问题：预计算优化

```
for (int j=i;j<=n;j++){  // 枚举尚未确定的数字
    if (isPrime(data[j]+data[i-1])){ // 剪枝
        data[i]=data[j];     // 交换i,j位的数字
        data[j]=tmp;
        search(data,n,i+1);    // 递归搜索下一个数
        data[j]=data[i];     // 恢复第j位的数字
    }
}
```

→ 每次调用solve都要用isPrime判断素数，其实判断的最大数值不会大于2n，会有很多重复判断

→ 所以你可以预先把1..2n每个数先判断好存起来，这种技术称为预计算，是一种常用技巧

代码不是很难，我不在这里贴了，留作作业

→ 这是一个开源项目Kylin（麒麟）
源码地址：https://github.com/KylinOLAP/Kylin/
第一个中国人主导研发的Apache基金会顶级开源项目
现已成立公司并融资

→ 这个项目号称可以在O(1)时间内解决大数据查询问题
其实就是做了预计算

状态转移类算法的一般步骤

1.选取合适的状态（关键）
2.确定状态转移的方式（关键，但一般不难）
3.写代码实现，递归or非递归

→ 你学的递推、分治法、回溯法大体上都属于状态转移类算法的类型。以后还会有更多
→ 怎么找到最合适的状态描述来解决问题，这是一大部分竞赛题的关键
→ 这一步其实就是算法的核心：建模

状态空间	线性结构	分支结构	网状结构
显式	数组 链表	树	图
隐式	递推	分治法 回溯法	动态规划

回溯法：剪枝技巧（选学）

矩形覆盖问题

平面上有n个点（n<=50），用k个矩形覆盖所有的点（k<=4），矩形之间不能重叠，并使矩形面积的总和最小。允许面积为0的退化矩形

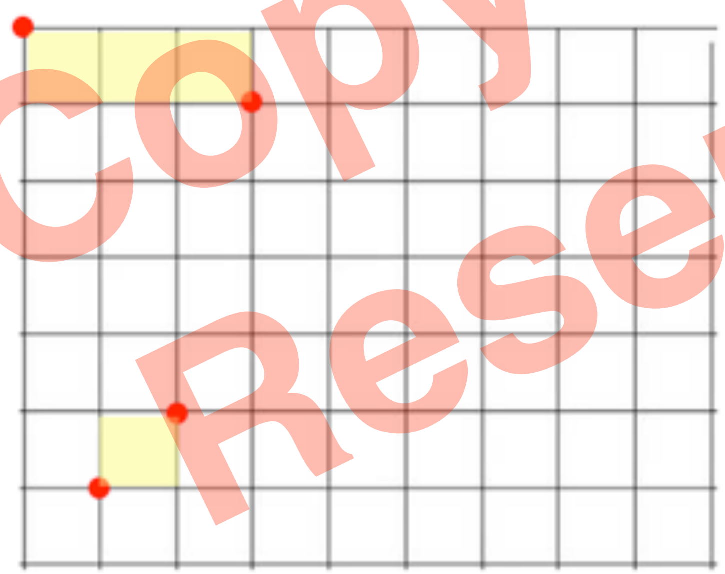
输入n,k和n个点的坐标，输出最小的面积

样例输入：

4 2
1 1
2 2
3 6
0 7

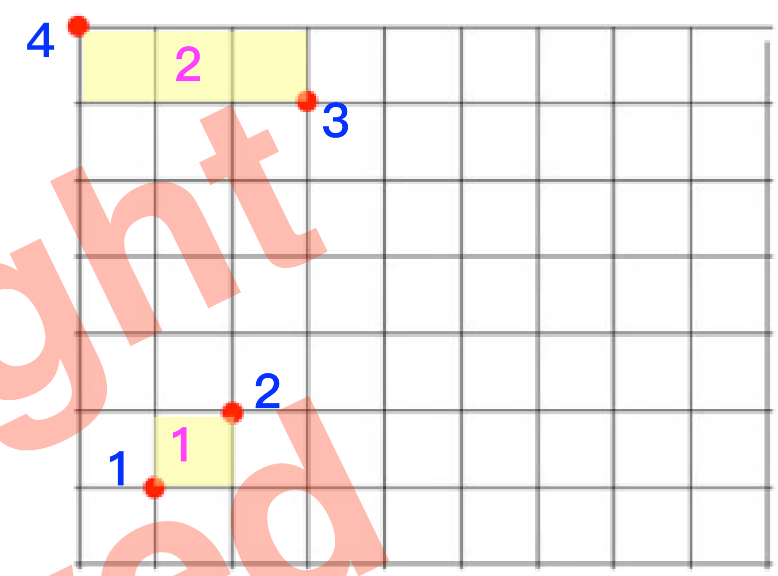
样例输出：

4



这道题目有点复杂，课上只讲思路
代码有兴趣的课后自己看

矩形覆盖问题：状态选择



→ 数据规模这么小，基本是暴力搜索无疑了

→ 怎么选择状态?

1. 以每个点属于哪个矩形为状态（$k^n \approx 10^{30}$）
   (1,1,2,2)
2. 以矩形的上下左右边经过哪个点为状态（$n^{4k} \approx 10^{27}$）
   ((2,1,1,2),(4,3,4,3))

→ **同一问题，为什么状态空间大小不同?**

因为同一种状态可能被重复描述

如以边为状态，以下其实是同一个状态

((1,2,3,4),(5,6,7,8))

((5,6,7,8),(1,2,3,4))

因为矩形之间没有顺序，而枚举过程有顺序

→ **这种由状态选择引起的冗余往往源于某种对称性，是剪枝可以考虑的方向**

矩形覆盖问题：数据结构

→ Point第3周就定义过了
→ Square是第3周的作业（多么完美的伏笔）

```
int n,k;
int result=INT_MAX; // 最优解
Point points[50];   // 输入数据

void squareCover(){
    scanf("%d%d", &n, &k);
    for (int i=0;i<n;i++){
        scanf("%d%d", &points[i].x, &points[i].y);
    }

    if (n<=k){   // 预判
        cout<<0<<endl;
        return;
    }

    solve(0, 0);

    cout<<result<<endl;
}
```

```
class Point{    // 点类型
public:
    int x,y;
    int covered=0;  // 是否已经被矩形覆盖
};

class Square{   // 矩形类型
public:
    Point left,right,top,bottom;    // 四条边上的点（可能相同）

    Square(Point l,Point r,Point t,Point b){
        left=l; right=r; top=t; bottom=b;
    }

    int area(){ // 返回面积
        return (top.y-bottom.y)*(right.x-left.x);
    }

    int isValid(){  // 判断是否构成合法的矩形
        return top.y>=left.y && top.y>=right.y && bottom.y<=left.y && bottom.y<=right.y
        && left.x<=top.x && left.x<=bottom.x && right.x>=top.x && right.x>=bottom.x;
    }

    int covers(Point pt){ // 判断矩形是否覆盖某个点
        return (top.y>=pt.y)&&(bottom.y<=pt.y)&&(left.x<=pt.x)&&(right.x>=pt.x);
    }
};
```

矩形覆盖问题：初步解答（回溯法）

```
/* @param i 当前试图放置第i+1个矩形
   @param areaNow 之前放置的矩形的总面积 */
void solve(int i,int areaNow){
    if (areaNow >= result){
        return; // 如果面积已经比最优解大，就不用算下去了
    }
    if (i==k){  // 递归边界，k个矩形已经枚举完毕
        for (int a=0;a<n;a++){  // 判断一下是不是所有点都覆盖到
            if (!points[a].covered){
                return;
            }
        }
        result=areaNow; // 更新最优解
        return;
    }
    // 枚举第i+1个矩形的四条边
    for (int a=0;a<n;a++){
        for (int b=0;b<n;b++){
            for (int c=0;c<n;c++){
                for (int d=0;d<n;d++){
                    Square sq=Square(points[a], points[b], points[c], points[d]);
                    if (sq.isValid()){  // 先判断合法
                        // 判断矩形不相交（当前矩形不会覆盖其他已经覆盖过的点）
                        int duplicate=0;
                        for (int x=0;x<n;x++){
                            if (sq.covers(points[x]) &&  points[x].covered){
                                duplicate=1;
                                break;
                            }
                        }

                        if (!duplicate){
                            // 放置当前矩形，更新被其覆盖的点（为什么不能再上一个循环里做？）
                            for (int x=0;x<n;x++){
                                if (sq.covers(points[x])){
                                    points[x].covered=1;
                                }
                            }
                            solve(i+1, areaNow+sq.area());  // 递归求解下一个

                            // 恢复被改动的全局状态，还原到递归调用之前的状态
                            for (int x=0;x<n;x++){
                                if (sq.covers(points[x])){
                                    points[x].covered=0;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

→ 其实已经（不经意间）进行了一些剪枝

→ 只是这些剪枝很容易想到

→ 下面再来看一些更高级的剪枝优化

矩形覆盖问题：局部贪心法

```
// 找到未被覆盖且y坐标最小的点
Point* findLowestPoint(){
    Point* pt=0;
    for (int a=0;a<n;a++){
        if (!points[a].covered && (pt==0 || points[a].
            pt=&points[a];
        }
    }
    return pt;
}
```

- 当前还没被覆盖的点里，找一个y坐标最小的点A
- 一定会有一个矩形的下边经过A
- OK，就选当前在枚举的矩形作为这个（下边经过A的）矩形
- 为什么可以“选”？因为矩形之间没有顺序，本质上就是利用对称性剪枝
- 于是只需要枚举另三条边：$n^{4k} \rightarrow n^{3k} \approx 10^{20}$
- 利用条件，有一部分选择是可以直接判断出来（而不是枚举），这种优化就叫局部贪心

矩形覆盖问题：边界裁剪

```
Square* findLastSquare(){
    Square* sq=0;
    for (int a=0;a<n;a++){
        if (!points[a].covered){
            if (sq==0){
                sq=new Square(points[a],points[a],points[a],points[a]);
            }else{
            if ( points[a].x<sq->left.x){
                sq->left=points[a];
            }
            if ( points[a].x>sq->right.x){
                sq->right=points[a];
            }
            if ( points[a].y<sq->bottom.y){
                sq->bottom=points[a];
            }
            if ( points[a].y>sq->top.y){
                sq->top=points[a];
            }}
        }
    }
    return sq;
}
```

→ **当只剩下最后一个矩形的时候，已经不用枚举了，可以直接解出来**
从没被盖住点里去最靠上下左右的四个点就行了（上面的函数）

→ **这样递归可以少一层，也就是递归边界变浅了一层：$n^{3k} \rightarrow n^{3(k-1)} \approx 10^{13}$**

→ **其实这个题可以裁剪不止一层，可以做到$n^{3(k-2)}$或者$n^{3(k-3)}$，甚至整个都可以不用回溯法**

→ **当然裁剪越深处理就越复杂，这里不细说了，你可以自己尝试一下**

矩形覆盖问题：优化解答

```
// 判断一个矩形是否覆盖到已被覆盖的点
int checkDuplicate(Square sq){
    for (int a=0;a<n;a++){
        if (sq.covers(points[a]) &&  points[a].covered){
            return 1;
        }
    }
    return 0;
}

// 标记/取消标记一个新矩形
void markSquare(Square sq,int mar
    for (int a=0;a<n;a++){
        if (sq.covers(points[a]))
            points[a].covered=mar
        }
    }
}
```

```
void solve(int i,int areaNow){
    if (areaNow >= result){
        return; // 如果面积已经比最优解大，就不用算下去了
    } else if (i==k){  // 递归边界，k个矩形已经枚举完毕
        // 因为上一层(i=k-1)一定会覆盖所有点，所以这里不用再判断了
        result=areaNow; // 更新最优解
        return;
    } else if (i==k-1) {     // 最后一个矩形（边界裁剪）
        Square* sq=findLastSquare();
        if (sq != 0 && !checkDuplicate(*sq)){  // 判断不与其他矩形重叠
            solve(i+1, areaNow+sq->area());  // 递归求解下一个（进i==k分支）
        }
    }

    // 找到最低的一个未覆盖的点，作为bottom（局部贪心）
    Point* bottom=findLowestPoint();
    if (bottom==0){
        return;
    }

    // 枚举其他三条边
    for (int a=0;a<n;a++){
        for (int b=0;b<n;b++){
            for (int c=0;c<n;c++){

                Square sq=Square(points[a], points[b], points[c], *bottom);
                if (sq.isValid() && !checkDuplicate(sq)){  // 判断合法，且不与其他矩形重叠
                    markSquare(sq, 1); // 放置当前矩形，更新被其覆盖的点
                    solve(i+1, areaNow+sq.area());  // 递归求解下一个
                    markSquare(sq, 0); // 恢复被改动的全局状态
                }
            }
        }
    }
}
```

边界裁剪

局部贪心

作业

1.宇宙有几维空间？爱因斯坦说4维。有人说有11维

去搜一下怎么画高维空间。截一张图作为作业

2.（选做）上次的小球下落问题（fallingballbetter.cpp）

是不是觉得模拟算法有点傻？是不是隐约感觉有规律？写一个更高效的算法，要求时间性能为O(D)（所以就要用递推了）

3.（选做）我经常先忽悠你用非最优算法，再布置作业让你去想更好的算法😏（比如最大和子串、小球下落）。素数环问题会不会也是我挖的坑？

请你去查一下什么是“**哈密尔顿回路问题**”（这个问题已证明没有有效算法）证明素数环其实就是一类哈密尔顿回路问题（所以这次我并没有给你埋坑）

4.实现带预计算优化的素数环问题（primeringbetter.cpp）

5.八皇后问题（eightqueen.cpp）

这是一个经典问题

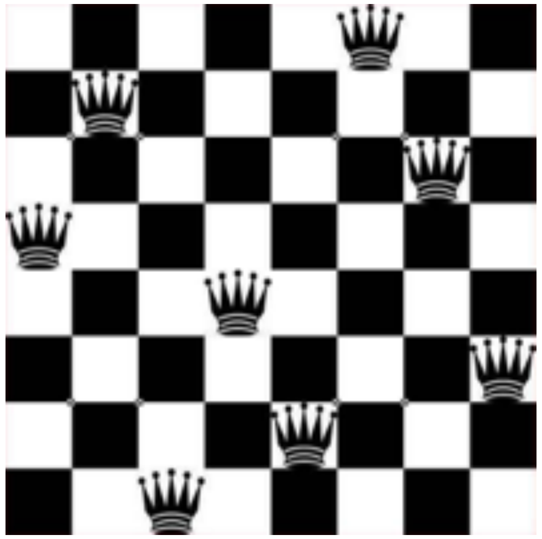
在国际象棋盘上放8个皇后，使其**互相不能攻击**

输出所有的解

样例输出：

6 2 7 1 4 8 5 3

……（还有很多行）



扩展阅读：博弈问题

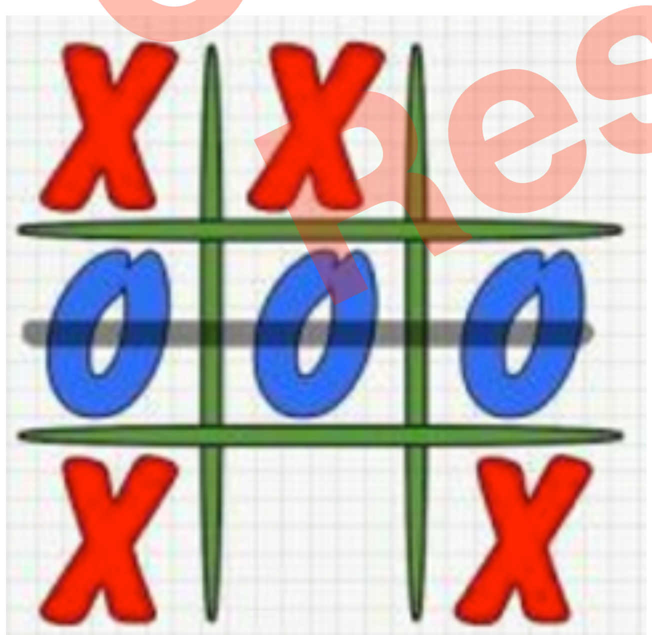
→ **人机对战的棋类游戏，本质上就是在一个状态空间中运动。棋局就是状态**

→ **有些游戏状态空间比较小（如井字棋），可以用回溯法枚举出每个状态，找到最优解**

→ **有些游戏状态数非常大（如围棋），不能完全枚举，这时就要在回溯法的基础上做一些估算和取舍，称为启发式搜索**

比如对棋局设计一个估值，局面越有利取值越高，**选取估值最高的一部分状态进行搜索**
这种思想就是局部贪心法剪枝的衍生，称为**启发式搜索**

→ **下围棋的AlphaGo使用的蒙特卡洛方法，本质上就是一种启发式搜索**



AlphaGo

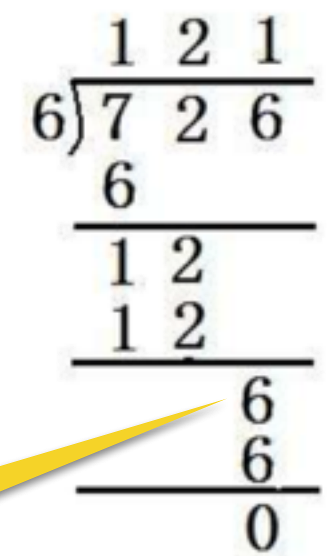
扩展阅读：人工智能的思考

→ **利用计算速度快来穷举最优解，算不算“智能”，这是个哲学问题**

→ **人与计算机的“思考”方式本质上是不同的**

人类擅长识图（形象思维）

电脑擅长识数（抽象思维）



人类不擅长

→ **用大量电子元件/虚拟单元模拟人脑（比如遗传算法），未必能获得智能**

因为智能可能是一种高层次的规律（层展）

电脑不擅长