Day 6

《算法竞赛入门经典（第二版）》第七章

暴力求解法

暴力求解法（暴力法）也称为穷举法、蛮力法，它要求调设计者找出所有可能的方法，然后选择其中的一种方法，若该方法不可行则试探下一种可能的方法。

暴力法也是一种直接解决问题的方法，常常直接基于问题的描述和所涉及的概念定义。

暴力法不是一个最好的算法，但当我们想不出更好的办法时，它也是一种有效的解决问题的方法。

暴力法的优点是逻辑清晰，编写程序简洁。在程序设计竞赛时，时间紧张，相对于高效的、巧妙的算法，暴力法编写的程序简单，能更快地解决问题。同时蛮力法也是很多算法的基础，可以在蛮力法的基础上加以优化，得到更高效的算法。

而且，某些情况下，算法规模不大，使用优化的算法没有必要，而且某些优化算法本身较复杂，在规模不在时可能因为复杂的算法浪费时间，反而不如简单的暴力搜索。

使用暴力法常用如下几种情况：

（1）搜索所有的解空间；

（2）搜索所有的路径；

（3）直接计算；

（4）模拟和仿真。

枚举、递归

枚举是基于逐个尝试答案的一种问题求解策略。

火柴棒等式

给你n（$n \le 24$）根火柴棍，你可以拼出多少个形如“$A+B=C$”的等式？等式中的A、B、C是用火柴棍拼出的整数（若该数非零，则最高位不能是0），数字的形状和电子表上的一样。

注意：

1．加号与等号各自需要两根火柴棍。

2．如果$A \neq B$，则$A+B=C$与$B+A=C$视为不同的等式（A，B，$C \ge 0$）。

3．n根火柴棍必须全部用上。

直接枚举A和B（事实证明只到3数）。为了加快速度，事先把222以内各个数所用的火柴数位算出来。

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

int matches[223], n;
void getmatches();
// 把火柴棍事先算好
int count=0;
```

```
int main() {
    getmatches();
    cin>>n;
    for (int i=0;i<=111;i++)
        for (int j=0;j<=111;j++) {
            int k=i+j;
            if(matches[i]+matches[j]+
                        matches[k]+4==n)
                count++;
        }
    cout<<count;
    return 0;
}
```

```
void getmatches() {
    matches[0]=6;
    matches[1]=2;
    matches[2]=5;
    ……
// 事先编一个程序来产生这一部分代码。
    matches[222]=15;
}
```

梵塔问题

已知有三根针分别用1、2、3表示。在一号针中从小到大放*n*个盘子，现要求把所有的盘子从1针全部移到3针。移动规则是：使用2针作为过渡针，每次只移动一块盘子，且每根针上不能出现大盘压小盘，找出移动次数最小的方案。

这是一个经典的递归问题。

递归的思路：如果想把n个盘子放到3针上，就应该先把n-1个盘子放到2针上。然后把1针最底部的盘子移动到3针，再把2针上的n-1个盘子移动到3针上。

```cpp
void move(int n, int a, int b, int c){
// a是盘子来源，b是暂存区、c是目标
    if (n==1)
        cout<<a<<"->"<<c<<endl;
        // 只有一根针时直接移动
    else{
        move(n-1,a,c,b);
        // 先把n-1个盘子移动到#2上
        cout<<a<<"->"<<c<<endl;
        move(n-1,b,a,c);
        // 把n-1个盘子移动到#3上
    }
}
```

选择客栈

丽江河边有n家（$2 \le n \le 200,000$）很有特色的客栈，客栈按照其位置顺序从1到n编号。每家客栈都按照某一种色调进行装饰（总共k种，用整数$0 \sim k-1$表示，且$0 < k \le 50$），且每家客栈都设有一家咖啡店，每家咖啡店均有各自的最低消费。

两位游客一起去丽江旅游，他们喜欢相同的色调，又想尝试两个不同的客栈，因此决定分别住在色调相同的两家客栈中。晚上，他们打算选择一家咖啡店喝咖啡，要求咖啡店位于两人住的两家客栈之间（包括他们住的客栈），且咖啡店的最低消费不超过p（$0 \le p$，最低消费≤ 100）。

他们想知道总共有多少种选择住宿的方案，保证晚上可以找到一家最低消费不超过p元的咖啡店小聚。

对于30%的数据，有$n \leq 100$;

对于50%的数据，有$n \leq 1,000$;

对于100%的数据，有 $2 \leq n \leq 200,000, 0 < k \leq 50, 0 \leq p \leq 100, 0 \leq$最低消费$\leq 100$。

首先考虑30%（$n \leq 100$）的数据。很明显，本题需要用枚举法解决。

枚举法的思路很明显：枚举区间[i，j]（$i \neq j$，且i、j色调相同），再判断区间内是否有最低消费不超过p的咖啡店。时间复杂度为O(n^3)。

现在开始优化算法。可以看到，判断区间内咖啡店的时间为O(n)，有减少的余地。令c[i]表示[1，i]之间最低消费不超过p的咖啡店个数(事实上不是咖啡店个数。如果i、j并列出现“√×”的情况，那么c[j]－c[i]也应该大于0)，那么可以看出，如果c[j]－c[i]＞0，说明i、j之间有符合要求的咖啡店。这样判断区间的时间就降到了O(1)。

有这一步做基础，为了使思路清晰，就可以把不同颜色的客栈分开了。

在处理j时，我们要统计1～j－1与j的解的个数。处理j＋1时又要处理1～j－1，造成了重复计算。

设sum[j]为[1，j]到[i，j]中解的个数。如果j＋1无法消费，那么由于j的存在，有sum[j+1]＝sum[j]；如果j＋1可以消费，那么j＋1可以和1到j中任何一个客栈组合，那么sum[j+1]＝j。

最后将每种颜色的所有sum相加，就可以在O(n)的时间内得出答案了。

```cpp
#include <iostream>
#include <cstring>
using namespace std;
int n,k,p;
int c[51][200001], top[51];
long long total=0;

int main(){
    memset(top,-1,sizeof(top));
    memset(c,0,sizeof(c));
    ios::sync_with_stdio(false);
    cin>>n>>k>>p;
    int cnt=0;
    bool P=false;
    for (int i=0; i<n; i++) {
        int x,y;
        cin>>x>>y;
        int t=++top[x];
        if (y<=p || P)
            cnt++;
        c[x][t]=cnt;
        P = y<=p;
    }
    for (int color=0; color<k; color++) {
        int prev=c[color][0];
        long long sum=0;

        for (int i=0; i<=top[color]; i++) {
            if (c[color][i]-prev>0) sum=i;
            total+=sum;
            prev=c[color][i];
        }
    }
    cout<<total<<endl;
    return 0;
}
```

回溯

回溯法是一种系统的搜索问题的解的方法。它的基本思想是：从一条路前行，能进则进，不能进则退回来，换一条路再试。回溯法是一种通用的解题方法，是一个既带有系统性又带有跳跃性的搜索算法；

它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。**（系统性）**

算法搜索至解空间树的任一结点时，判断该结点为根的子树是否包含问题的解，如果肯定不包含，则跳过以该结点为根的子树的搜索，逐层向其祖先结点回溯。否则，进入该子树，继续深度优先的策略进行搜索。**（跳跃性）**

这种以深度优先的方式系统地搜索问题的解得算法称为回溯法，它适用于解一些组合数较大的问题。

问题的解空间

问题的解向量：回溯法希望一个问题的解能够表示成一个n元式$(x_1,x_2,\ldots,x_n)$ 的形式。

显约束：对分量x_i的取值限定。

隐约束：为满足问题的解而对不同分量之间施加的约束。

解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组， 构成了该实例的一个解空间。

注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。

应用回溯法的时候，首先明确定义问题的解空间。解空间至少应该包含问题的一个解。确定了解空间后，回溯法从开始结点出发，以深度优先的方法搜索整个解空间。

对于回溯法一般可以采用递归方式来实现。

基本思想

搜索从开始结点（根结点）出发，以深度优先搜索整个解空间。

这个开始结点成为活结点，同时也成为当前的扩展结点。在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为新的活结点，并成为当前扩展结点。

如果在当前的扩展结点处不能再向纵深方向扩展，则当前扩展结点就成为死结点。

此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点；直到找到一个解或全部解。

基本步骤

①针对所给问题，定义问题的解空间；

②确定易于搜索的解空间结构；

③以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

常用剪枝函数

①用约束函数在扩展结点处剪去不满足约束的子树；

②用限界函数剪去得不到最优解的子树。

二类常见的解空间树

①子集树：当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树称为子集树。子集树通常有2^n个叶子结点，其总结点个数为2^{n+1} -1，遍历子集树时间为$\Omega(2^n)$ 。如0-1 背包问题，叶结点数为2^n，总结点数2^{n+1}；

②排列树：当所给问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。排列树通常有n!个叶子结点，因此，遍历排列树需要$\Omega(n!)$的计算时间。如TSP问题，叶结点数为n!，遍历时间为$\Omega(n!)$。

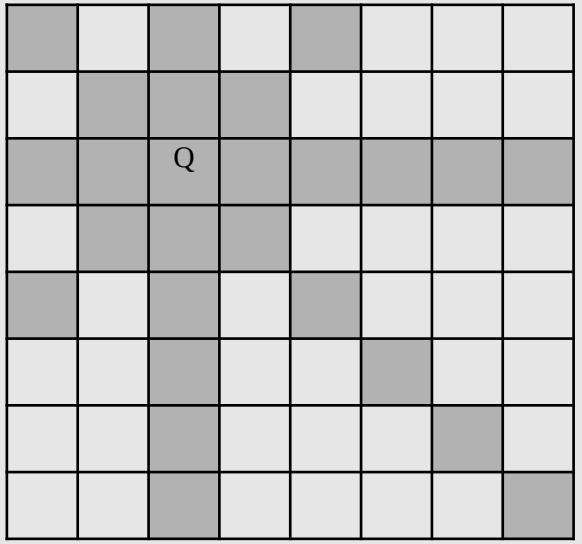
用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为h(n)，则回溯法所需的计算空间通常为O(h(n))。而显式地存储整个解空间则需要$O(2^{h(n)})$或O(h(n)!)内存空间。

算法框架

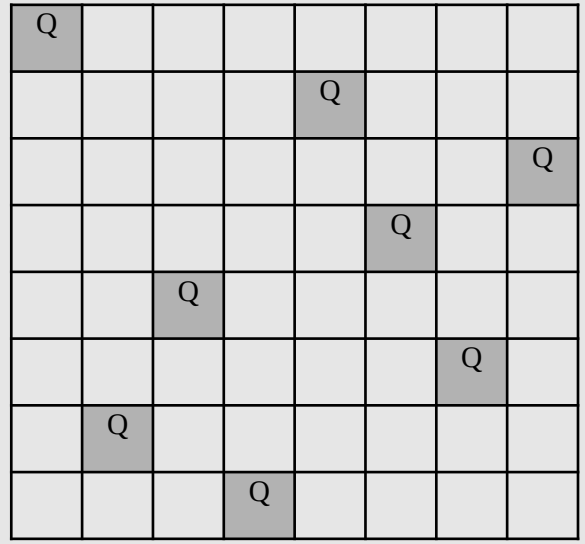
回溯法对解空间作深度优先搜索，因此在一般情况下可用递归函数来实现回溯法

八皇后问题

在棋盘上放置8个皇后，使得它们互不攻击，此时每个皇后的攻击范围为同行同列和对角线，要求找出所有解，如图所示。



(a)皇后的攻击范围



(b)一个可行解

思路一：把问题转化为“从64个格子中选一个子集”，使得“子集中恰好有8个格子，且任意两个选出的格子都不在同一行、同一列或同一个对角线上”。这是子集枚举问题，不是一个好的模型。

思路二：把问题转化为“从64个格子中选8个格子”，这是组合生成问题。比思路一好，但是仍然不是很好。

思路三：由分析可得，恰好每行每列各放置一个皇后。如果用C[x]表示第x行皇后的列编号，则问题变成了全排列生成问题。

在编写递归枚举程序之前，需要深入分析问题，对模型精雕细琢。一般还应对解答树的结点数有一个粗略的估计，作为评价模型的重要依据。

四皇后问题的完整解答树只有18个结点，比4!=24小。原因是有些结点无法继续扩展。

在各种情况下，递归函数将不再递归调用本身，而是返回上一层调用，称这种现象为**回溯（backtracking）**。

```cpp
#include <cstdio>
int C[50], tot = 0, n, nc = 0;
void search(int cur) {
    int i, j;
    nc++;       //nc状态空间结点数
    if(cur == n)   tot++;  //递归边界。只要走到了这里，所有皇后必然不冲突
    else for(i = 0; i < n; i++) {
        int ok = 1;
        C[cur] = i;                 //尝试把cur行的皇后放在第i列
        for(j = 0; j < cur; j++)  //检查是否和前面的皇后冲突
            if(C[cur] == C[j] || cur-C[cur] == j-C[j] || cur+C[cur] == j+C[j]) {
                ok = 0;
                break;
            }
        if(ok)   search(cur+1);    //如果合法，则继续递归
    }
}
```

```cpp
int main() {
    scanf("%d",&n);
    search(0);
    printf("%d\n", tot);
    printf("%d\n", nc);
    return 0;
}
```

既然是逐行放置的，则皇后肯定不会横向攻击，因此只需检查是否纵向和斜向攻击即可。条件cur-C[cur] == j-C[j] || cur+C[cur] == j+C[j]用来判断皇后(cur,C[cur])和(j,C[j])是否在同一条对角线上。其原理可以用图说明。

0	1	2	3	4	5	6	7
-1	0	1	2	3	4	5	6
-2	-1	0	1	2	3	4	5
-3	-2	-1	0	1	2	3	4
-4	-3	-2	-1	0	1	2	3
-5	-4	-3	-2	-1	0	1	2
-6	-5	-4	-3	-2	-1	0	1
-7	-6	-5	-4	-3	-2	-1	0

(a)格子(x,y)的y-x值标识了主对角线

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

(b)格子(x,y)的x+y值标识了副对角线

上面的程序还可改进：利用二维数组vist[2][]直接判断当前尝试的皇后所在的列和两个对角线是否已有其他皇后。注意到主对角线标识y-x可能为负，存取时要加上n。完整的程序如下

```cpp
#include <cstdio>
int C[50], vis[3][50], tot = 0, n, nc = 0;

void search(int cur) {
    int i, j;
    nc++;
    if(cur == n)   tot++;
    else for(i = 0; i < n; i++) {
        if(!vis[0][i] && !vis[1][cur+i] && !vis[2][cur-i+n]) {
            //利用二维数组直接判断
            C[cur] = i;  //如果不打印解，整个C数组都可以省略
            vis[0][i] = vis[1][cur+i] = vis[2][cur-i+n] = 1;  //修改全局变量
            search(cur+1);
            vis[0][i] = vis[1][cur+i] = vis[2][cur-i+n] = 0; //切记一定要改回来
        }
    }
}

int main() {
    scanf("%d",&n);
    memset(vis, 0, sizeof(vis));
    search(0);
    printf("%d\n", tot);
    printf("%d\n", nc);
    return 0;
}
```

上面的程序关键的地方是vis数组的使用。vis数组表示已经放置的皇后占据了哪些列、主对角线和副对角线。将来放置的皇后不应该修改这些值。一般地，如果要回溯法中修改了辅助的全局变量，则一定要及进把它们恢复原状（除非故意保留修改）。另外，千万不要忘记在调试之前把vis数组清空。

如果在回溯法中使用了辅助的全局变量，则一定要及时把它们恢复原状。例如，若函数有多个出口，则需在每个出口处恢复被修改的值。

素数环

输入正整数n，把整数1,2,3,…,n组成一个环，使得相邻两个整数之和均为素数。输出时从整数1开始逆时针排列。同一个环应恰好输出一次。n≤16。

样例输入:

6

样例输出:

1 4 3 2 5 6

1 6 5 2 3 4

素数环问题的程序实际上主要由求素数和整数1,2,3,…,n的排列构成。

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;
int is_prime(int x) {
//判数一个整数x是否是一个素数
    for(int i = 2; i*i <= x; i++)
        if(x % i == 0)
            return 0;
    return 1;
}
```

```cpp
int main() {
    int n, A[50], isp[50];
    scanf("%d", &n);
    for(int i = 2; i <= n*2; i++)
        //生成素数表，加快后继判断
        isp[i] = is_prime(i);
    for(int i = 0; i < n; i++)    A[i] = i+1;
        //第一个排列
    do {
        int ok = 1;
        for(int i = 0; i < n; i++) if(!isp[A[i]+A[(i+1)%n]]) {
        //判断合法性
            ok = 0;
            break;
        }
        if(ok) {
            for(int i = 0; i < n; i++) printf("%d ", A[i]);
                //输出序列
            printf("\n");
        }
    }while(next_permutation(A+1, A+n));
    return 0;
}
```

运行后，发现当n=12时就已经很慢了，而当n=16根本出不来。
下面采取回溯法来解决此问题

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

int is_prime(int x) { //判数一个整数x是否是一个素数
  for(int i = 2; i*i <= x; i++)
    if(x % i == 0)  return 0;
  return 1;
}

int n, A[50], isp[50], vis[50];

  void dfs(int cur) {
    if(cur == n && isp[A[0]+A[n-1]]){//递归边界，别忘测试第一个数和最后一个数
      for(int i = 0; i < n; i++) printf("%d ", A[i]);
      printf("\n");
    }
    else for(int i = 2; i <= n; i++)  //尝试放置每个数i
      if(!vis[i] && isp[i+A[cur-1]]) {
  //如果i没有用过，并且与前一个数之和为素数
        A[cur] = i;
        vis[i] = 1;              //设置标志
        dfs(cur+1);
        vis[i] = 0;              //清除标志
      }
  }

int main() {
    scanf("%d", &n);
    for(int i = 2; i <= n*2; i++)
        isp[i] = is_prime(i);
    memset(vis, 0, sizeof(vis));
    A[0] = 1;
    dfs(1);
    return 0;
}
```

路径寻找问题（隐式图搜索）

对很多问题可以用图的遍历算法解决，但这些问题的图却不是事先给定、从程序读入的，而是由程序动态生成的。

回溯法是按照深度优先顺序遍历的，它的优点是空间很节省：只有递归栈中结点需要保存。换句话说，回溯法的空间开销和访问的最深结点的深度成正比。

树不仅可以深度优先遍历，还可以宽度优先遍历。好处是找到的第一个解一定是离根最近的解，但空间开销却大增加（队列中结点很多）。

八数码问题

编号为1～8的8个正方形滑块摆成3行3列（有一个格式留空），如图所示。每次可以把与空格相邻的滑块（有公共边才算相邻）移动到空格中，而它原来的位置就成为了新的空格。给定初始局面和目标局面（用0表示空格），你任务是计算出最少的移动步数。如果无法到达目标局面，则输出-1。

2	6	4
1	3	7
	5	8

8	1	5
7	3	6
4		2

样例输入：

```
2 6 4 1 3 7 0 5 8
8 1 5 7 3 6 4 0 2
```

样例输出：

```
31
```

把八数码问题归结为图上的最短路问题，其中每个状态就是9个格式中的滑块编号（从上到下、从左到右地把它们放在一个包含9个元素的数组中）。

```cpp
typedef int State[9];           //定义“状态”类型
const int MAXSTATE = 1000000;
State st[MAXSTATE], goal;       //状态数组。所有状态都保存在这里
int dist[MAXSTATE];             //距离数组
//如果需要打印方案，可以在这里加一个“父亲编号”数组int fa[MAXSTATE]
const int dx[] = {-1, 1, 0, 0};
const int dy[] = {0, 0, -1, 1};


//BFS，返回目标状态在st数组下标
int bfs() {
  init_lookup_table();          //初始化查找表
  int front = 1, rear = 2;      //不使用下标0,因为0被看作“不存在”
  while(front < rear) {
    State& s = st[front];       //用“引用”简化代码
    if(memcmp(goal, s, sizeof(s)) == 0) //找到目标状态，成功返回
        return front;
    int z;
    for(z = 0; z < 9; z++)   if(!s[z]) break;  //找“0”的位置
    int x = z/3, y = z%3;                      //获取行列位置（0～2）
```

注意，此处用到了cstring库中的memcmp和memcpy完成整块内存的比较和复制，比用循环比较和循环赋值要快。

```
for(int d = 0; d < 4; d++) {
        int newx = x + dx[d];
        int newy = y + dy[d];
        int newz = newx * 3 + newy;
        if(newx >= 0 && newx < 3 && newy >= 0 && newy < 3) { //如果移动合法
          State & t = st[rear];
          memcpy(&t, &s, sizeof(s));  //扩展新结点
          t[newz] = s[z];
          t[z] = s[newz];
          dist[rear] = dist[front] + 1;  //更新新结点的距离值
          if(try_to_insert(rear))   rear++;//如果成功插入查找表，修改队尾指针
        }
    }
    front++;                                  //扩展完毕后再修改队首指针
  }
  return 0;                                   //失败
}
```

注意，此处用到了cstring库中的memcmp和memcpy完成整块内存的比较和复制，比用循环比较和循环赋值要快。

```
int main() {
  for(int i = 0; i < 9; i++)     //起始状态
    scanf("%d", &st[1][i]);
  for(int i = 0; i < 9; i++)     //目标状态
    scanf("%d", &goal[i]);
  int ans = bfs();               //返回目标状态的下标
  if(ans > 0)   printf("%d\n", dist[ans]);
  else printf("-1\n");
  return 0;
}
```

注意，在调用bfs函数之前设置好st[1]和goal。对于上面的程序还有init_lookup_table()和try_to_insert(rear)没有实现，这两个函数的功能是初始化查找表和插入元素。在查找表中，避免将同一个结点访问多次，所以要判重复，如果不判重复，时间和空间将产生极大的浪费。

结点查找表的实现：方法一

把排列“变成”整数，然后只开一个一维数组。即设计一套排列的编码（encoding）和解码（decoding）函数，把0～8的全排列和0～362879的整数一一对应起来。

```
int vis[36288], fact[9];
void init_lookup_table() {
  fact[0] = 1;
  for(int i = 1; i < 9; i++) fact[i] = fact[i-1] * i;
}

int try_to_insert(int s) {
  int code = 0;                    //把st[s]映射到整数code
  for(int i = 0; i < 9; i++) {
    int cnt = 0;
    for(int j = i+1; j < 9; j++)
if(st[s][j] < st[s][i])   cnt++;
    code += fact[8-i] * cnt;
  }
  if(vis[code]) return 0;
  return vis[code] = 1;
}
```

上面尽管原理巧妙，时间效率也非常高，但编码解码法的适用范围不大：如果隐式图的总结数非常大，编码也将会很大，数组还是存不下。

结点查找表的实现：方法二

使用哈希（hash）技术。只需要设计一个所谓的哈希数h(x)，然后将任意结点x映射到某个给定范围[0,M-1]的整数即可，其中M是程序员根据可用内存大小自选的。在理想情况下，只需开一个大小为M的数组就能完成判重，但此时往往会有不同结点的哈希值相同，因此需要把哈希值相同的状态组织成链表。

```
const int MAXHASHSIZE = 1000003;
int head[MAXHASHSIZE], next[MAXSTATE];
void init_lookup_table() { memset(head, 0, sizeof(head)); }
int hash(State& s) {
  int v = 0;
  for(int i = 0; i < 9; i++) //随便算。例如，把9个数字组合成9位数
 v = v * 10 + s[i];
  return v % MAXHASHSIZE;
//确保hash函数值不超过hash表的大小的非负整数
}
```

```
int try_to_insert(int s) {
  int h = hash(st[s]);
  int u = head[h];    //从表头开始查找链表
  while(u) {
    if(memcmp(st[u], st[s], sizeof(st[s])) == 0)
//找到了，插入失败
return 0;
    u = next[u];      //顺着链表继续找
  }
  next[s] = head[h];  //插入到链表中
  head[h] = s;
  return 1;
}
```

迭代加深搜索

例题：埃及分数

在古埃及，人们使用单位分数的和(形如$1/a$的, a是自然数)表示一切有理数。
如：$2/3 = 1/2 + 1/6$,但不允许$2/3 = 1/3 + 1/3$,因为加数中有相同的。
对于一个分数a/b,表示方法有很多种，但是哪种最好呢?
首先，加数少的比加数多的好，其次，加数个数相同的，最小的分数越大越好。
如:

$$19/45 = 1/3 + 1/12 + 1/180$$
$$19/45 = 1/3 + 1/15 + 1/45$$
$$19/45 = 1/3 + 1/18 + 1/30,$$
$$19/45 = 1/4 + 1/6 + 1/180$$
$$19/45 = 1/5 + 1/6 + 1/18.$$

最好的是最后一种，因为$1/18$比$1/180, 1/45, 1/30, 1/180$都大。
给出$a, b(0 < a < b < 1000)$,编程计算最好的表达方式。

Input
一行包含$a, b\ (0 < a < b < 1000)$, 。
Output
每组测试数据若干个数，自小到大排列，依次是单位分数的分母。
Sample Input
19 45
Sample Output
5 6 18

```cpp
// 埃及分数问题
// Rujia Liu

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

int a, b, maxd;//深度上限

typedef long long LL;

LL gcd(LL a, LL b) {
  return b == 0 ? a : gcd(b, a%b);
}

// 返回满足1/c <= a/b的最小c
inline int get_first(LL a, LL b) {
  return b/a+1;
}
```

```cpp
const int maxn = 100 + 5;

LL v[maxn], ans[maxn];

// 如果当前解v比目前最优解ans更优，更新ans
bool better(int d) {
  for(int i = d; i >= 0; i--) if(v[i] != ans[i]) {
    return ans[i] == -1 || v[i] < ans[i];
  }
  return false;
}
```

```
int main() {
  int kase = 0;
  while(cin >> a >> b) {
    int ok = 0;
    for(maxd = 1; maxd <= 100; maxd++) {
      memset(ans, -1, sizeof(ans));
      if(dfs(0, get_first(a, b), a, b)) { ok = 1; break; }
    }
    cout << "Case " << ++kase << ": ";
    if(ok) {
      cout << a << "/" << b << "=";
      for(int i = 0; i < maxd; i++) cout << "1/" << ans[i] << "+";
      cout << "1/" << ans[maxd] << "\n";
    } else cout << "No solution.\n";
  }
  return 0;
}
```

```
// 当前深度为d，分母不能小于from，分数之和恰好为aa/bb
bool dfs(int d, int from, LL aa, LL bb) {
  if(d == maxd) {
    if(bb % aa) return false; // aa/bb必须是埃及分数
    v[d] = bb/aa;
    if(better(d)) memcpy(ans, v, sizeof(LL) * (d+1));
    return true;
  }
  bool ok = false;
  from = max(from, get_first(aa, bb)); // 枚举的起点
  for(int i = from; ; i++) {
    // 剪枝：如果剩下的maxd+1-d个分数全部都是1/i，加起来仍然不超过aa/bb，则无解
    if(bb * (maxd+1-d) <= i * aa) break;
    v[d] = i;
    // 计算aa/bb - 1/i，设结果为a2/b2
    LL b2 = bb*i;
    LL a2 = aa*i - bb;
    LL g = gcd(a2, b2); // 以便约分
    if(dfs(d+1, i+1, a2/g, b2/g)) ok = true;
  }
  return ok;
}
```

如果决定用DFS做一道题，那么：

- 如果能套用模板或经典的DFS模型，就直接套用。
- 如果不能套用，就先写爆搜，并命名为#0版本。
- 对于每一个版本，调试正确之后，再进行剪枝，并且每次只加一个剪枝，调试正确后再继续。
- 每次加剪枝都应该开一个新的版本，不能直接修改原版本。
  如果时间来不及了，就按照从强到弱或从好写到难写的顺序来加剪枝。
- 在动态查错之前应该进行一次静态查错。
- 调试时，如果出现RE（运行时错误）或死循环，就在出错的状态处停止，并检查其所有的祖先状态（显然这需要记下每次的决策），看看这些决策是否合法。
- 如果程序正常结束，但结果错误，应该从以下三个角度入手：
  1. 明明有解，输出无解：可以把正解的各个决策一步一步代入，看看程序里面是在哪一步出了问题（明明合法的决策它没有作出），从而方便检查；
  2. 输出不合法的解：可以把形成这个不合法解的决策一步一步代入，看看是哪一步出了问题；
  3. 输出合法但非最优的解：必然是某些最优性剪枝错误或者最优性判断出了问题，可以直接到这些地方去检查，实在不行也可以把最优解代入检查。
- 有时也可以用分段检查的方法，即把代码中的各个过程或者片段截取下来，将几组小的输入代入，看看执行之后，所有在该片段里改变了值的全局变量是否正确，进而发现这里的错误。
- 千万不要一下子输出全部的状态！这样不仅不容易查出结果，还把容易把自己绕晕。
  事实上，只有在状态总数不太多，且相对关系明了、顺序整齐的时候（比如一般的DP等），可以一下子输出全部状态来检查，否则就不能这么做。

对于**BFS**，其实它比**DFS**容易调试，因为所有的状态都储存在队列里。因此，可以在状态结点中记下每个点的前趋，然后，哪里出了问题，就直接把它的前趋状态全部输出。

另外，**BFS**一般没有“剪枝”这一说（除了判重），且决策过程一般是比较整齐的，因此更容易调试。

	DFS	BFS
优势	1．比较适合回溯类搜索 2．参数传递、状态修改和恢复都比较方便 3．自顶向下地处理问题 4．记忆化搜索容易实现 5．能很快到达解答树的底端	1．解决**“最少步数”、“深度最小”**问题 2．问题的解靠近解答树的根结点 3．启发式搜索在BFS中更容易实现 4．能立刻停止搜索
缺点	1．使用递归算法容易导致栈溢出 2．有时不容易输出方案 3．不易立即结束搜索	1．空间一般比DFS大 2．状态重复的排除有时耗时多

迭代加深搜索

广度优先搜索可以用迭代加深搜索代替。迭代加深搜索实质是限定下界的深度优先搜索，即首先允许深度优先搜索搜索k层搜索树，若没有发现可行解，再将k+1代入后再进行一次以上步骤，直到搜索到可行解。这个“模仿广度优先搜索”搜索法比起广搜是牺牲了时间，但节约了空间。

在需要做BFS，但没有足够的空间，时间却很充裕的情况下，可以考虑迭代加深搜索。

迭代加深搜索的时间不会超过等效BFS占用时间的两倍，当数据规模增大时，二者的差距还会逐渐减小，而迭代加深搜索的空间占用和DFS一样小。

搜索的优化

没有经过优化的搜索叫“爆搜”或“裸搜”。爆搜能承受的规模不大，原因有二：一是搜索时产生了无用的结点，二是重复计算。

优化时从以下几方面入手：确定更合适的搜索顺序、剪枝（可行性剪枝、最优化剪枝）、降低考察和扩展结点的代价，必要时可采用高级的搜索方法，或使用状态压缩。

此外，写搜索程序时应该先从朴素算法入手，逐步优化，以降低编程复杂度、减少不必要的失误。