**Exercise 1**

思路：动态构建树；Hill Climbing方法的本质是添加排序的深搜；Best-First Search方法的本质是添加了排序的广搜。

**public** **class** Treesearch

{

**public** **static** **void** main(String[] args)

{

Node T1 = **new** Node();

*initializecube*(T1);

Node T2 = **new** Node();

*initializecube*(T2);

Node T3 = **new** Node();

*initializecube*(T3);

Node T4 = **new** Node();

*initializecube*(T4);

System.***out***.println("Output:");

System.***out***.println("可达");

**long** starttime = System.*currentTimeMillis*();

*breadthfirst*(T1);

**long** endtime = System.*currentTimeMillis*();

**long** exacttime = endtime - starttime;

System.***out***.println("Breadth-First: " + exacttime + "ms");

starttime = System.*currentTimeMillis*();

*depthfirst*(T2 , 0);

endtime = System.*currentTimeMillis*();

exacttime = endtime - starttime;

System.***out***.println("Depth-First: " + exacttime + "ms");

starttime = System.*currentTimeMillis*();

*hillclimbing*(T3);

endtime = System.*currentTimeMillis*();

exacttime = endtime - starttime;

System.***out***.println("Hill Climbing: " + exacttime + "ms");

starttime = System.*currentTimeMillis*();

*bestfirstsearch*(T4);

endtime = System.*currentTimeMillis*();

exacttime = endtime - starttime;

System.***out***.println("Best-First Search: " + exacttime + "ms");

}

//先广搜索。

**public** **static** **boolean** breadthfirst(Node T)

{

Queue<Node> q = **new** LinkedList<>();

q.offer(T);

**while**(q.peek() != **null**)

{

Node t = q.poll();

t.visited = **true**;

*creatsons*(t);

**if**(t.cost == 0)

**return** **true**;

**for**(**int** i = 0 ; i < t.sons.size() ; i++)

{

**if**(t.sons.get(i).visited == **false**)

{

q.offer(t.sons.get(i));

}

}

}

**return** **false**;

}

//先深搜索。

**public** **static** **boolean** depthfirst(Node T , **int** counter)

{

counter++;

**if**(counter > 12)

**return** **false**;

**if**(T.cost == 0)

**return** **true**;

*creatsons*(T);

**for**(**int** i = 0 ; i < T.sons.size() ; i++)

{

**if**(*depthfirst*(T.sons.get(i),counter))

**return** **true**;

}

**return** **false**;

}

**public** **static** **boolean** hillclimbing(Node T)

{

**if**(T.cost == 0)

**return** **true**;

*creatsons*(T);

//添加排序，使每一次都优先遍历cost最小的节点

**for**(**int** i = 0 ; i < T.sons.size() ; i++)

{

**for**(**int** j = i ; j < T.sons.size() ; j++)

{

**if**(T.sons.get(i).cost > T.sons.get(j).cost)

{

Node temp = T.sons.get(i);

T.sons.set(i, T.sons.get(j));

T.sons.set(j, temp);

}

}

}

**for**(**int** i = 0 ; i < T.sons.size() ; i++)

{

**if**(*hillclimbing*(T.sons.get(i)))

**return** **true**;

}

**return** **false**;

}

**public** **static** **boolean** bestfirstsearch(Node T)

{

myQueue<Node> q = **new** myQueue<>();

q.offer(T);

**while**(!q.isempty())

{

Node t = q.poll();

t.visited = **true**;

*creatsons*(t);

**if**(t.cost == 0)

**return** **true**;

**for**(**int** i = 0 ; i < t.sons.size() ; i++)

{

**if**(t.sons.get(i).visited == **false**)

{

q.offer(t.sons.get(i));

}

}

//对队列进行排序，以保证每次取出的是cost最小的。

*sort*(q);

}

**return** **false**;

}

//动态生成树的方法。

**public** **static** **void** creatsons(Node t)

{

**int** x = 0, y = 0;

**for**(**int** i = 1 ; i <= 3 ; i++ )

{

**for**(**int** j = 1 ; j <= 3 ; j++)

{

**if**(t.magiccube[i][j] == -1)

{

x = i;

y = j;

}

}

}

**if**(x == 2 && y == 2)

{

Node t1 = **new** Node();

Node t2 = **new** Node();

Node t3 = **new** Node();

Node t4 = **new** Node();

*move*(t,t1,x,y,"up");

*move*(t,t2,x,y,"down");

*move*(t,t3,x,y,"right");

*move*(t,t4,x,y,"left");

**if**(*islegal*(t, t1))

{

t1.father = t;

t1.cost = *calcostfun*(t1.magiccube);

t.sons.add(t1);

}

**if**(*islegal*(t, t2))

{

t2.father = t;

t2.cost = *calcostfun*(t2.magiccube);

t.sons.add(t2);

}

**if**(*islegal*(t, t3))

{

t3.father = t;

t3.cost = *calcostfun*(t3.magiccube);

t.sons.add(t3);

}

**if**(*islegal*(t, t4))

{

t4.father = t;

t4.cost = *calcostfun*(t4.magiccube);

t.sons.add(t4);

}

}

**else** **if**(x == 1 && y == 1)

{

Node t2 = **new** Node();

Node t3 = **new** Node();

*move*(t,t2,x,y,"down");

*move*(t,t3,x,y,"right");

**if**(*islegal*(t, t2))

{

t2.father = t;

t2.cost = *calcostfun*(t2.magiccube);

t.sons.add(t2);

}

**if**(*islegal*(t, t3))

{

t3.father = t;

t3.cost = *calcostfun*(t3.magiccube);

t.sons.add(t3);

}

}

**else** **if**(x == 1 && y == 2)

{

Node t2 = **new** Node();

Node t3 = **new** Node();

Node t4 = **new** Node();

*move*(t,t2,x,y,"down");

*move*(t,t3,x,y,"right");

*move*(t,t4,x,y,"left");

**if**(*islegal*(t, t2))

{

t2.father = t;

t2.cost = *calcostfun*(t2.magiccube);

t.sons.add(t2);

}

**if**(*islegal*(t, t3))

{

t3.father = t;

t3.cost = *calcostfun*(t3.magiccube);

t.sons.add(t3);

}

**if**(*islegal*(t, t4))

{

t4.father = t;

t4.cost = *calcostfun*(t4.magiccube);

t.sons.add(t4);

}

}

**else** **if**(x == 1 && y == 3)

{

Node t2 = **new** Node();

Node t4 = **new** Node();

*move*(t,t2,x,y,"down");

*move*(t,t4,x,y,"left");

**if**(*islegal*(t, t2))

{

t2.father = t;

t2.cost = *calcostfun*(t2.magiccube);

t.sons.add(t2);

}

**if**(*islegal*(t, t4))

{

t4.father = t;

t4.cost = *calcostfun*(t4.magiccube);

t.sons.add(t4);

}

}

**else** **if**(x == 2 && y == 1)

{

Node t1 = **new** Node();

Node t2 = **new** Node();

Node t3 = **new** Node();

*move*(t,t1,x,y,"up");

*move*(t,t2,x,y,"down");

*move*(t,t3,x,y,"right");

**if**(*islegal*(t, t1))

{

t1.father = t;

t1.cost = *calcostfun*(t1.magiccube);

t.sons.add(t1);

}

**if**(*islegal*(t, t2))

{

t2.father = t;

t2.cost = *calcostfun*(t2.magiccube);

t.sons.add(t2);

}

**if**(*islegal*(t, t3))

{

t3.father = t;

t3.cost = *calcostfun*(t3.magiccube);

t.sons.add(t3);

}

}

**else** **if**(x == 2 && y == 3)

{

Node t1 = **new** Node();

Node t2 = **new** Node();

Node t4 = **new** Node();

*move*(t,t1,x,y,"up");

*move*(t,t2,x,y,"down");

*move*(t,t4,x,y,"left");

**if**(*islegal*(t, t1))

{

t1.father = t;

t1.cost = *calcostfun*(t1.magiccube);

t.sons.add(t1);

}

**if**(*islegal*(t, t2))

{

t2.father = t;

t2.cost = *calcostfun*(t2.magiccube);

t.sons.add(t2);

}

**if**(*islegal*(t, t4))

{

t4.father = t;

t4.cost = *calcostfun*(t4.magiccube);

t.sons.add(t4);

}

}

**else** **if**(x == 3 && y == 1)

{

Node t1 = **new** Node();

Node t3 = **new** Node();

*move*(t,t1,x,y,"up");

*move*(t,t3,x,y,"right");

**if**(*islegal*(t, t1))

{

t1.father = t;

t1.cost = *calcostfun*(t1.magiccube);

t.sons.add(t1);

}

**if**(*islegal*(t, t3))

{

t3.father = t;

t3.cost = *calcostfun*(t3.magiccube);

t.sons.add(t3);

}

}

**else** **if**(x == 3 && y == 2)

{

Node t1 = **new** Node();

Node t3 = **new** Node();

Node t4 = **new** Node();

*move*(t,t1,x,y,"up");

*move*(t,t3,x,y,"right");

*move*(t,t4,x,y,"left");

**if**(*islegal*(t, t1))

{

t1.father = t;

t1.cost = *calcostfun*(t1.magiccube);

t.sons.add(t1);

}

**if**(*islegal*(t, t3))

{

t3.father = t;

t3.cost = *calcostfun*(t3.magiccube);

t.sons.add(t3);

}

**if**(*islegal*(t, t4))

{

t4.father = t;

t4.cost = *calcostfun*(t4.magiccube);

t.sons.add(t4);

}

}

**else** **if**(x == 3 && y == 3)

{

Node t1 = **new** Node();

Node t4 = **new** Node();

*move*(t,t1,x,y,"up");

*move*(t,t4,x,y,"left");

**if**(*islegal*(t, t1))

{

t1.father = t;

t1.cost = *calcostfun*(t1.magiccube);

t.sons.add(t1);

}

**if**(*islegal*(t, t4))

{

t4.father = t;

t4.cost = *calcostfun*(t4.magiccube);

t.sons.add(t4);

}

}

}

//初始化根节点。

**public** **static** **void** initializecube(Node t)

{

t.magiccube[1][1] = 2;

t.magiccube[1][2] = 3;

t.magiccube[1][3] = -1;

t.magiccube[2][1] = 1;

t.magiccube[2][2] = 8;

t.magiccube[2][3] = 5;

t.magiccube[3][1] = 7;

t.magiccube[3][2] = 4;

t.magiccube[3][3] = 6;

t.cost = *calcostfun*(t.magiccube);

}

//移动魔方块的方法。

**public** **static** **void** move(Node t1 , Node t2 , **int** x , **int** y , String choice )

{

**for**(**int** i = 1 ; i <= 3 ; i++ )

{

**for**(**int** j = 1 ; j <= 3 ; j++)

{

t2.magiccube[i][j] = t1.magiccube[i][j];

}

}

**if**(choice.equals("up"))

{

**int** temp = t2.magiccube[x][y];

t2.magiccube[x][y] = t2.magiccube[x-1][y];

t2.magiccube[x-1][y] = temp;

}

**if**(choice.equals("down"))

{

**int** temp = t2.magiccube[x][y];

t2.magiccube[x][y] = t2.magiccube[x+1][y];

t2.magiccube[x+1][y] = temp;

}

**if**(choice.equals("right"))

{

**int** temp = t2.magiccube[x][y];

t2.magiccube[x][y] = t2.magiccube[x][y+1];

t2.magiccube[x][y+1] = temp;

}

**if**(choice.equals("left"))

{

**int** temp = t2.magiccube[x][y];

t2.magiccube[x][y] = t2.magiccube[x][y-1];

t2.magiccube[x][y-1] = temp;

}

}

//判断路径中是否已经经过t1时的状态（动态生成树的时候要用到）

**public** **static** **boolean** islegal(Node t , Node t1)

{

**int** flag1 = 0 , flag2 = 0;

Node visitor = t;

**while**(visitor.father != **null**)

{

visitor = visitor.father;

**for**(**int** i = 1 ; i <= 3 ; i++)

{

**for**(**int** j = 1 ; j <= 3 ; j++)

{

**if**(visitor.magiccube[i][j] != t1.magiccube[i][j])

flag1 = 1;

}

}

**if**(flag1 == 0)

flag2 = 1;

flag1 = 0;

}

**if**(flag2 == 0)

**return** **true**;

**else**

**return** **false**;

}

//计算代价函数（在错误位置上的数字个数）

**public** **static** **int** calcostfun(**int**[][] magiccube)

{

**int** counter = 0;

**if**(magiccube[1][1] != 1)

counter++;

**if**(magiccube[1][2] != 2)

counter++;

**if**(magiccube[1][3] != 3)

counter++;

**if**(magiccube[2][1] != 8)

counter++;

**if**(magiccube[2][3] != 4)

counter++;

**if**(magiccube[3][1] != 7)

counter++;

**if**(magiccube[3][2] != 6)

counter++;

**if**(magiccube[3][3] != 5)

counter++;

**return** counter;

}

//对队列进行排序（为实现最佳优先方法）

**public** **static** **void** sort(myQueue<Node> q)

{

**int** i , j;

**for**(i = 0 ; i < q.queue.size() ; i++)

{

**for**(j = i ; j < q.queue.size() ; j++)

{

**if**(q.queue.get(i).cost > q.queue.get(j).cost)

{

Node temp = q.queue.get(i);

q.queue.set(i, q.queue.get(j));

q.queue.set(j, temp);

}

}

}

}

}

**class** myQueue<E>

{

**public** List<E> queue = **new** ArrayList<>();

**public** **boolean** isempty()

{

**if**(queue.size() == 0)

**return** **true**;

**else**

**return** **false**;

}

**public** **void** offer(E obj)

{

queue.add(obj);

}

**public** E poll()

{

E obj = queue.get(0);

queue.remove(obj);

**return** obj;

}

}

**class** Node

{

**boolean** visited = **false**;

**int** cost;

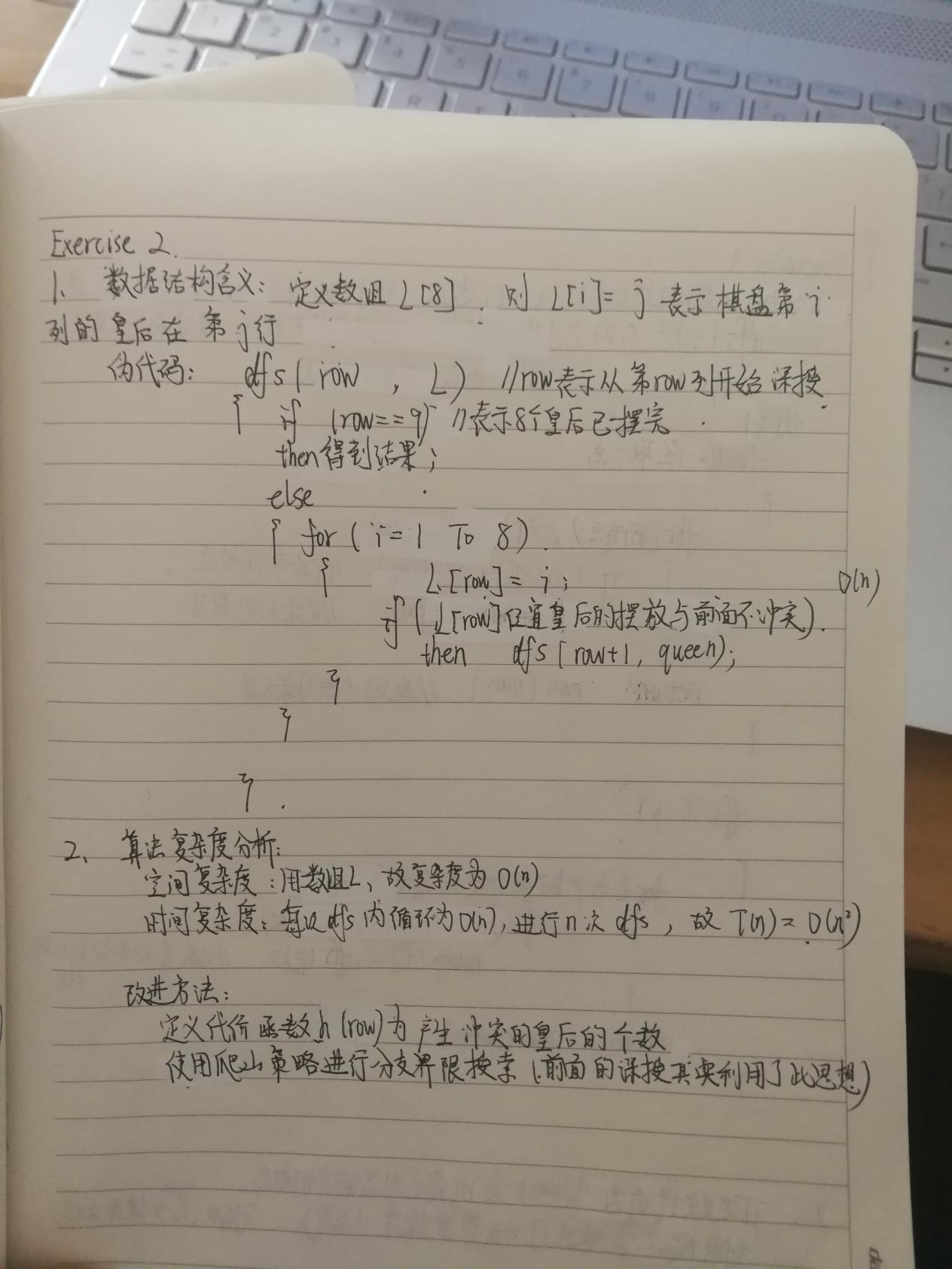
**int**[][] magiccube = **new** **int**[4][4];

Node father = **null**;

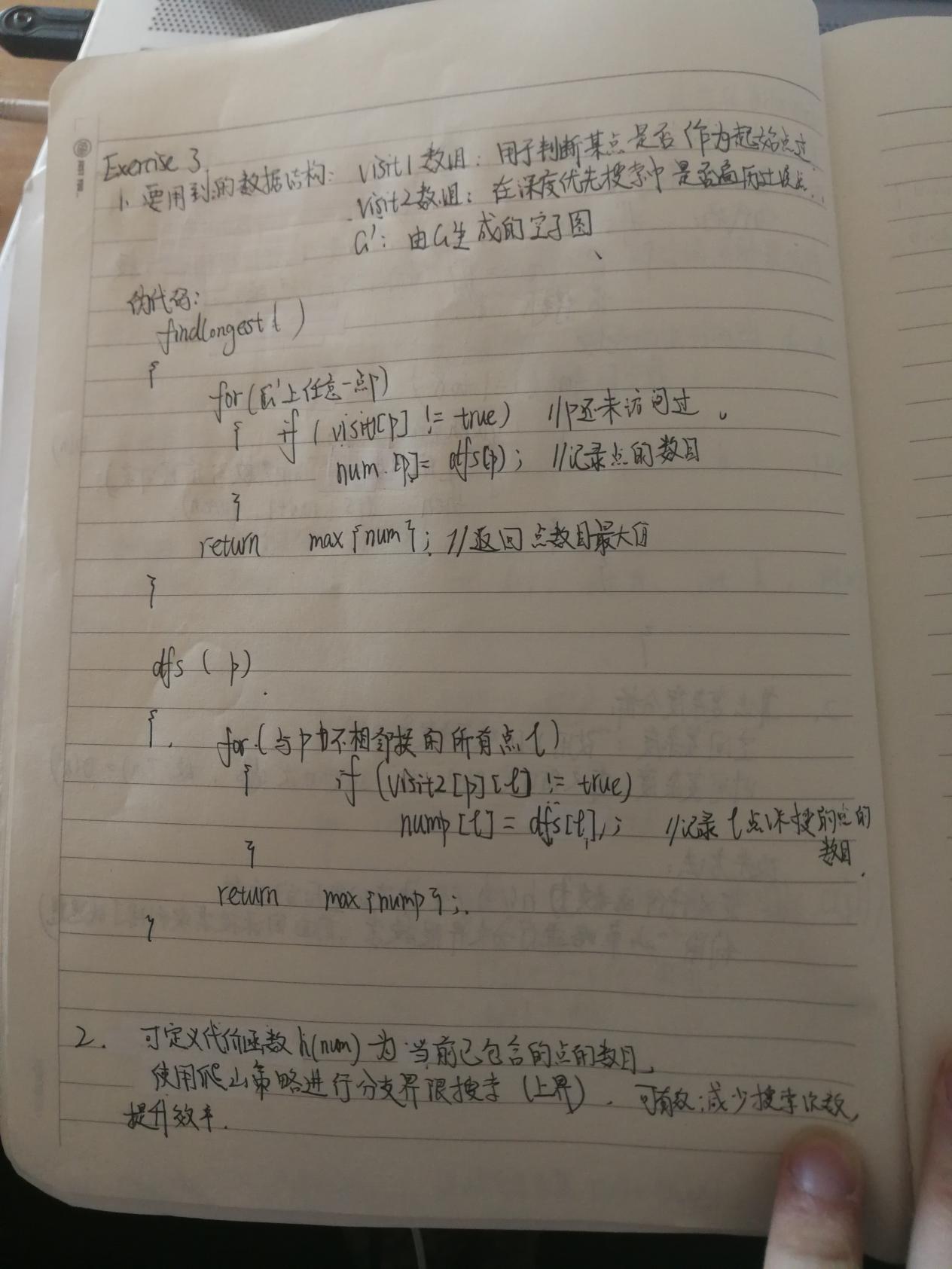
List<Node> sons = **new** ArrayList<>();

}

**Exercise 2**

****

**Exercise 3**



**Exercise 4**

