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];
            }
        }
```

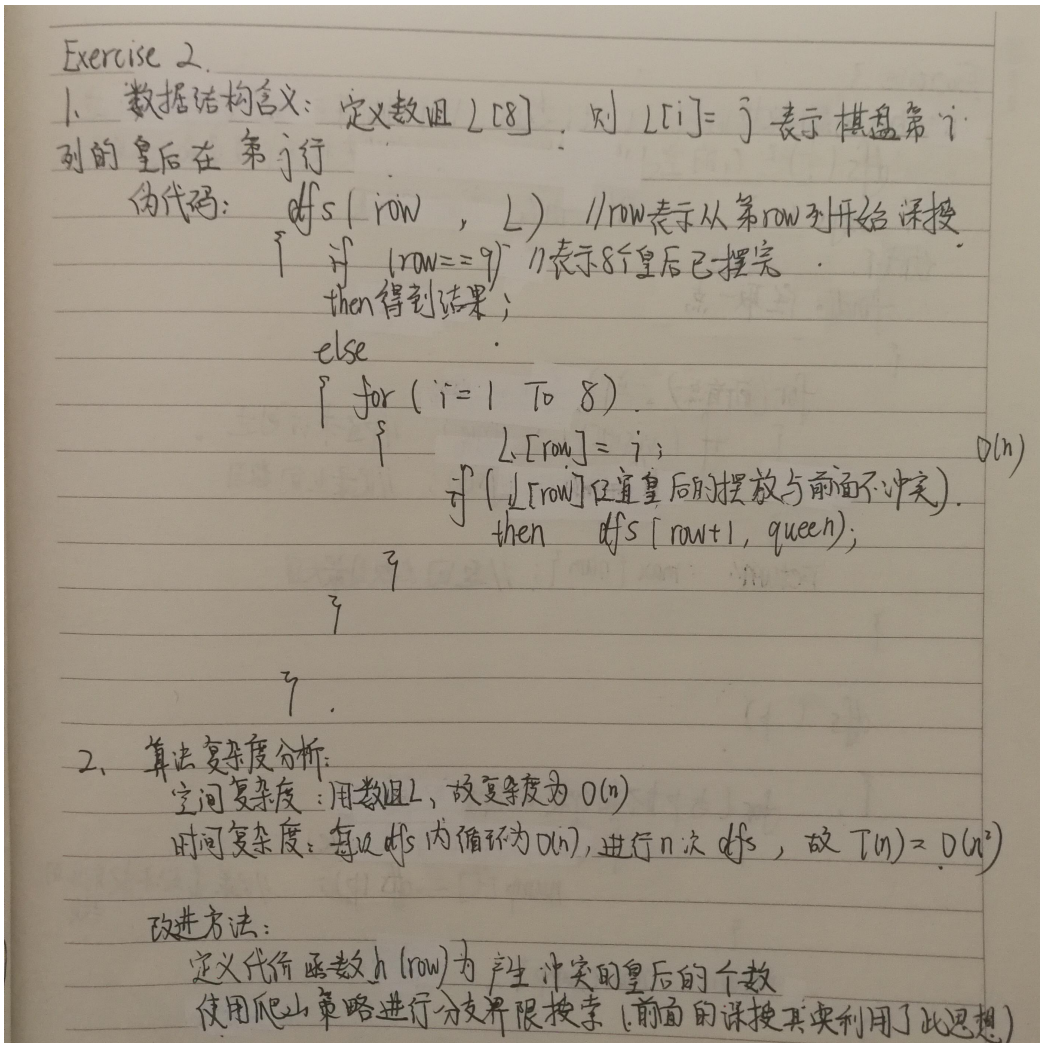
```
    if(choice.equals("up"))
    {
        int temp = t2.magiccube[x][y];
        t2.magiccube[x][y] = t2.magiccube[x-1][y];
        t2.magiccube[x-1][y] = temp;
    }
    if(choice.equals("down"))
    {
        int temp = t2.magiccube[x][y];
        t2.magiccube[x][y] = t2.magiccube[x+1][y];
        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);
```

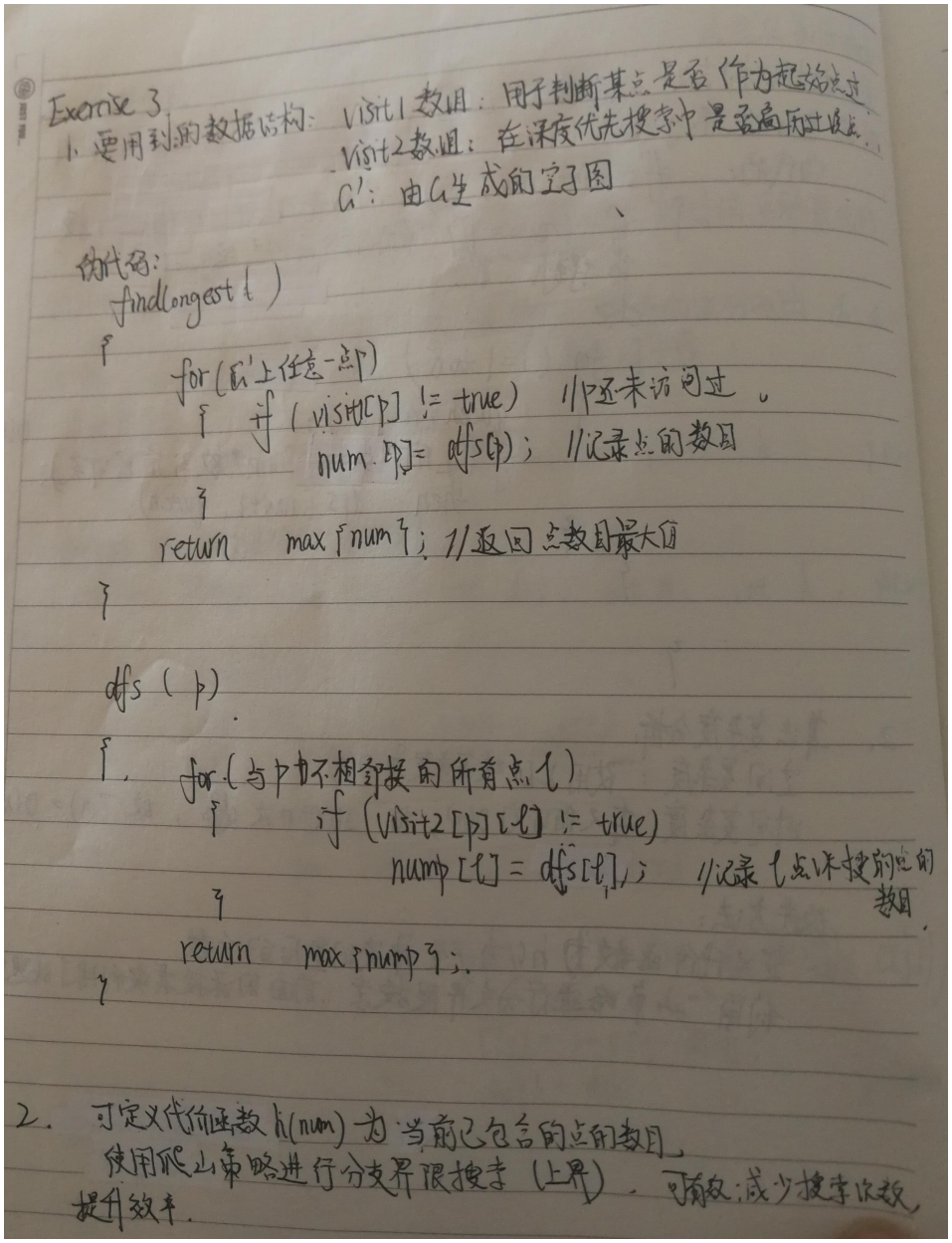
```
                    q.queue.set(i, q.queue.get(j));
                    q.queue.set(j, temp);
                }
            }
        }
    }

}



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 2.
1. 数据结构含义：定义数组 L[8]，则 L[i]=j 表示棋盘第 i 列的皇后在第 j 行
伪代码：
dfs(row, L) //row表示从第row列开始深搜
{ if (row==9) //表示8行皇后已摆完
then 得到结果；
else
{ for (i=1 To 8)
{ L[row]=i;
if (L[row]位置皇后的摆放与前面不冲突)
then dfs(row+1, queen);
}
}
}
O(n)
2. 算法复杂度分析：
空间复杂度：用数组L，故复杂度为 O(n)
时间复杂度：每次dfs内循环为O(n)，进行n次dfs，故 T(n)=O(n²)
改进方法：
定义代价函数 h(row)为产生冲突的皇后的个数
使用爬山策略进行分支界限搜索（前面的深搜其实利用了此思想）

Exercise 3



Exercise 3

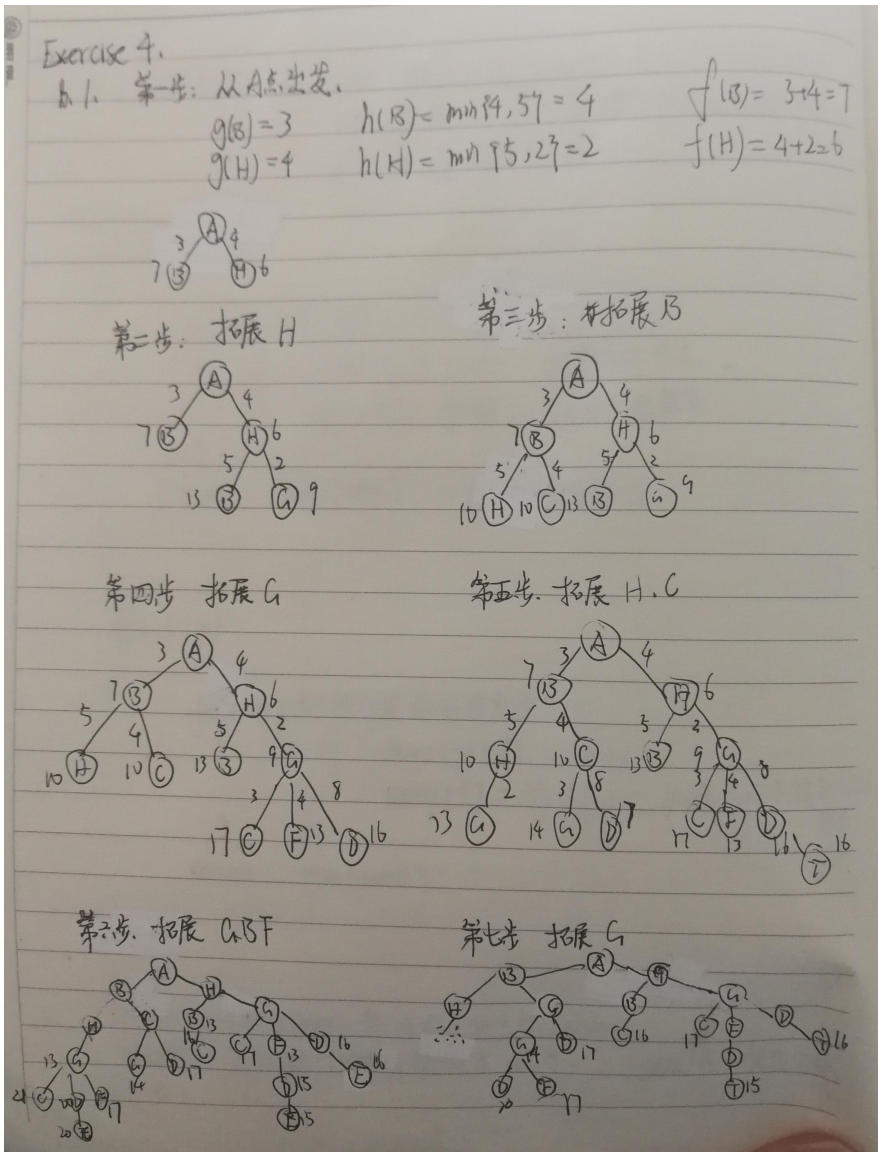
1. 使用到的数据结构：visit1 数组：用于判断某点是否作为起始点过。
visit2 数组：在深度优先搜索中是否遍历过该点。
G'：由G生成的子图

伪代码：

```
findlongest ( )
{
    for(G'上任意一点p)
    {  if ( visit[p] != true)   //p还未访问过。
           num.p= dfs(p);  //记录点的数目
    }
    return   max{num};  //返回点数目最大值
}

dfs ( p)
{.   for(与p相不相邻接的所有点t)
     {       if (visit2[p][t] != true)
                 nump[t] = dfs[t];  //记录t点以及搜索点的数目.
     }
     return   max{nump};.
}
```

2. 可定义代价函数 h(num) 为当前已包含的点的数目。
使用爬山策略进行分支界限搜索（上界）。可减少搜索次数，提升效率。

Exercise 4


Exercise 4.
h.1. 第一步：从A点出发。
g(B)=3 h(B)=min{4,5}=4 f(B)=3+4=7
g(H)=4 h(H)=min{5,2}=2 f(H)=4+2=6
第二步：拓展H
第三步：拓展B
第四步 拓展G
第五步. 拓展H.C
第六步. 拓展C.B.F
第七步 拓展G

第八步，扩展E

故最优解为 A-H-G-F-D-T

2. 用到的数据结构 heap：以f排序的小根堆，way：存储路径

```
findminway()
{  for(与A相邻的点p)
   {  g(p) = d(Ap);
      way(p).add(Ap);
      h(p) = min{d(p与非way上的点)};
      f(p) = g(p) + h(p);
      heap.add(p:(f(p).way(p)));//向堆中添加包含f(p)与way(p)的数据结构;
   }
   while(heap.top != E) //堆顶非终点E.
   {  k = heap.top;
      for(与k相邻的点l)
      {  g(l) = way(k).length + d(kl);
         way(l).add(way(k)+kl);
         h(l) = min{d(l与非way(l)上的点)};
         f(l) = g(l) + h(l);
         heap.add(l:(f(l), way(l)));
      }
   }
   return heap.top.way. //得到最短路径.
}
```