**深 圳 大 学 实 验 报 告**

**课程名称：­ 算法设计与分析**

**实验项目名称： 动态规划——鸡蛋掉落实验**

**学院： 计算机与软件学院**

**专业： 计算机科学与技术**

**指导教师： 杨烜**

**报告人： 郑雨婷 学号：2021150122 班级： 高性能**

**实验时间： 2023/4/20——2023/5/13**

**实验报告提交时间： 2023/5/13**

**教务部制**

|  |  |
| --- | --- |
| 实验目的与要求：  （1）掌握动态规划算法设计思想。  （2）掌握鸡蛋坠落问题的动态规划解法。 | |
| 方法、步骤：  阅读鸡蛋掉落的问题描述，分析列出解决问题的动态规划方程。根据动态规划方程，设计动态规划的不同实现方式。  对于蛮力法，改变f和e的值，对小数据模型利用蛮力法测试算法的正确性；对于其他非蛮力法，给出每一种算法的原理、伪代码、并通过对大量样本的测试结果，统计不同算法的时间效率和空间效率与输入规模的关系，并与理论分析的基本运算次数做比较，验证理论分析结论的正确性。 | |
| 实验过程及内容：  一、暴力法  1.1算法原理  动态规划是一种用以求解具有某种**以阶段划分、最优性质**的问题的算法。动态规划将问题划分为更小的子问题，通过子问题的最优解来重构原问题的最优解。在鸡蛋掉落问题中，步骤如下：  **①什么是子问题？**  当鸡蛋数小于e 或楼层数小于f时即为子问题。  **②什么操作能够暴露子问题？**  当扔一个鸡蛋后，有碎与不碎两种情况，分别对应了两种不同的子问题。  **③找到原问题与子问题之间关系、写出动态规划方程。**  定义符号：T[e][f]表示e个蛋，f层楼时的最少试验次数  第一个鸡蛋从第k层扔：若鸡蛋碎了，那么只需要在第1层到k-1层中寻找答案即可，还剩下e-1个鸡蛋，即T[e−1][k−1]；若鸡蛋没碎，那么需要在第k+1层到f层中寻找答案，此时还剩下e个鸡蛋，即T[e][f−k]。我们需要确保能够找到门槛层，因此需要考虑最坏情况，两者取大，再加上本次扔鸡蛋的一次试验，那么从k层扔所需要的最少试验次数为1+max(T[e−1][k−1],T[e][f−k])。    图1 暴力法原理示意图  遍历1到f层，取试验次数最少的那一层即可。因此：  状态转移方程：  边界条件： T[1][f]=f; T[e][0]=0;  1.2 伪代码实现  int Recursion(e,f){  if e==1： return f  if f ==0： return 0  int min=f;  for i=1 to f  int tmp\_max=1+ max(Recursion(e-1,i-1),Recursion(e,f-i));  if tmp\_max<min： min=tmp\_max;  return min;  }  1.3 算法实际执行时间  暴力法各数据规模下运行平均时间如下表1所示，单位毫秒：    表1 暴力法在不同规模下实测运行时间  1.4 算法效率分析  将表1数据绘制成曲线图2如下：    图2 暴力法在不同规模下实测运行时间  观察可以发现，暴力法的运行时间是呈指数增长的。用leetcode测试发现在结果正确但是运行超时，限制时间内能处理的最大规模为:e=3 f=25。  二、带备忘录的自顶向下  2.1算法原理  画出暴力法的递归树图3，观察这颗递归树，发现存在**大量重复计算**，所以这个算法效率很低。可以设置一个二维数组作为备忘录，算过的答案填入备忘录中，下次使用时直接查找，这样就会省去重复计算的时间，提升效率。    图3 暴力法递归树（部分）  2.2 伪代码实现  M[][]  int Recursion\_Memory(e,f){  if e==1： return f  if f ==0： return 0  if M[e][f]!=0 return M[e][f]  int min=f;  for i=1 to f  int tmp\_max=1+ max(Recursion\_Memory(e-1,i-1), Recursion\_Memory(e,f-i));  if tmp\_max<min： min=tmp\_max;  M[e][f]=min;  return min;  }  2.3 算法实际执行时间  带备忘录的自顶向下算法各数据规模下运行平均时间如下表2所示，单位毫秒：    表2带备忘录的自顶向下算法在不同规模下实测运行时间  表2数据绘制成曲图4如下：    图4 带备忘录的自顶向下算法在不同规模下实测运行时间  2.4 算法效率分析  动态规划算法的时间复杂度就是子问题个数×函数本身的复杂度。函数本身的复杂度就是忽略递归部分的复杂度，这里函数中有一个for循环，所以函数本身的复杂度是O(f)。子问题个数也就是不同状态组合的总数O(ef)。  所以算法的总时间复杂度是 O(),空间复杂度为O(ef)。我选取e=30,f=2000数据作为基准，通过之间的比例关系，计算得到各实验数据中算法执行时间的理论值。  理论曲线和实际曲线拟合程度较高，时间复杂度分析正确。用leetcode测试发现在结果正确但是运行超时，限制时间内能处理的最大规模为: : e=5 f=10000。    图5 带备忘的自顶向下理论实际对比  三、自底向上——打表  3.1算法原理  带备忘的自顶向下时在算T[e][f]时发现需要用到时才去计算，既然我们知道一定会用到，可以提前算好填入表中，那么就可以从T[1][0]开始，把e\*f规模的表保存好。这种做法称为自底向上。可以省去递归调用函数时消耗的时间。  3.2 伪代码实现  void table( i, j) {  int min = j;  for k = 1 to j:  int tmp\_max = max(T[i - 1][k - 1], T[i][j - k]) + 1;  if tmp\_max < min: min = tmp\_max;  T[i][j] = min;  }  void NO\_Recursion(int e, int f) {  for i = 0 to f:  T[1][i] = i  for i = 0 to e:  T[i][0] = 0;  for i = 2 to e:  for j=1 to f:  table(i, j);  cout << T[e][f] << endl;  }  3.3 算法实际执行时间  自底向上算法各数据规模下运行平均时间如下表3所示，单位毫秒：    表3 自底向上算法在不同规模下实测运行时间  表3数据绘制成曲线图6如下：    图6自底向上算法在不同规模下实测运行时间  3.4 算法效率分析  自底向上打表算法时间复杂度为 O(),空间复杂度为O(ef)，与带备忘的自顶向下相同，只是省去了递归调用的时间。我选取e=30,f=2000数据作为基准，通过之间的比例关系，计算得到各实验数据中算法执行时间的理论值。    图7 自底向上理论实际对比  理论曲线和实际曲线拟合程度较高，时间复杂度分析正确。用leetcode测试发现在结果正确但是运行超时，限制时间内能处理的最大规模为: : e=7 f=10000  四、带二分查找的动态规划  4.1算法原理  在确定第一次该从哪一层开始扔时，要遍历所有的f层，所有复杂度依旧是平方级别的。因此希望可以不用遍历所有的f层，我们观察到T[e][f]是一个关于f单调递增函的数，也就是说在鸡蛋数e固定的情况下，楼层数f越多，需要的试验次数一定不会变少。在状态转移方程中，当k递增时，T1=𝑇[𝑒−1][𝑘−1]单调递增,T2=𝑇[𝑒][𝑓−𝑘]单调递减。将T1和T2在同一坐标系中画出，要找出使得T1、T2中较大的值最小的k。若T[e][f]为连续的，k即为为T1和T2交点的横坐标，但楼层数f是离散的，我们可以找到距离交点最近的左右两个整数点，比较取小即可。    图8 二分查找示意图  4.2 伪代码实现  int MM[][];  int dp(e,f) {  //边界条件  low = 1, high = f;  while (low + 1 < high) {  mid = (low + high) / 2;  t1 = T(e - 1, mid - 1);  t2 = T(e, f - mid);  if t1 < t2 : low = mid;  else if :t1 > t2 : high = mid;  else : ow = high = mid;  }  MM[e][f]=1 + min(max(T(e-1,low-1), T(e, f-low)),max(T(e-1,high-1), T(e, f-high)));  return MM[e][f];  }  4.3 算法实际执行时间  带二分查找算法各数据规模下运行平均时间如下表4所示,单位微秒：    表4 带二分查找算法在不同规模下实测运行时间  表4数据绘制成曲线图9如下：    图9 带二分查找算法在不同规模下实测运行时间  4.4 算法效率分析  用二分查找得到最优的那个 k要 O(logf) 的时间，所以算法时间复杂度为，空间复杂度为O(ef)。我选取e=70,f=2000数据作为基准，通过之间的比例关系，计算得到各实验数据中算法执行时间的理论值。    图10 二分查找理论实际对比  理论曲线和实际曲线存在一定差距，但整体趋势正确。用leetcode测试，全部样例都通过。  五、逆向思维法  5.1算法原理  反过来想这个问题：如果我们可以做m次操作，而且有e个鸡蛋，那么我们能找到答案的最高的f是多少？定义符号：dp[e][m]表示e个蛋，m次尝试时能确定的最高楼层数。当dp[e][m]==f时，m则为需要实验的最少次数。  在第x层扔出鸡蛋，若碎了，能确定的楼层一定小于x；若没碎，能确定的楼层数大于等于x。因为dp[e][m]表示能确定的最高楼层数，所以是鸡蛋没碎的情况。那么能确定的楼层是x上方、x下方和x本身。即为dp[e][m-1]+dp[e-1][m-1]+1。  状态转移方程：dp[e][m] = dp[e][m - 1] + dp[e - 1][m - 1] + 1  边界条件：dp[1][m]=m;  dp[m][1]=1;    图11 逆向思维原理图  5.2 伪代码实现  int superEggDrop(E,f) {  int dp [E+ 1][F+ 1];  //边界条件  m= 0;  while (dp[E][m] < f) {  m++;  for e = 2 to E:  dp[e][m] = dp[e][m - 1] + dp[e - 1][m - 1] + 1;  }  return m;  }  5.3 算法实际执行时间  逆向思维算法各数据规模下运行平均时间如下表5所示，单位微妙：    表5 逆向思维算法在不同规模下实测运行时间  表4数据绘制成曲线图12如下：    图12 逆向思维算法在不同规模下实测运行时间  5.4 算法效率分析  逆向思维算法时间复杂度为，空间复杂度为O(ef)。经过实际运行，我选取e=10,f=2000数据作为基准，通过之间的比例关系，计算得到各实验数据中算法执行时间的理论值。    图13 逆向思维理论实际对比  理论曲线和实际曲线存在一定差距，但整体趋势正确。用leetcode测试，全部样例都通过。 |
| 数据处理分析：  1.将时间复杂度为O()的两种算法（自底向上和自顶向下）放在一起比较分析。e=30时两种算法的实际运行时间比较如下图：    图14 自底向上和自顶向下比较  根据图表可以看出，自顶向下明显快于自底向上，与之前分析的结果一致，自底向上省去了递归调用函数的时间。  2. 将两种微妙级的算法（带二分查找和逆向思维）放在一起比较分析。e=10时两种算法的实际运行时间比较如下表：    图15 二分查找和逆向思维比较  根据图表可以看出，逆向思维明显快于二分查找，与之前分析的时间复杂度结果一致，逆向思维时间复杂度为O(ef),二分查找时间复杂度为O(eflogf),所以逆向思维明显快于二分查找。 |

|  |
| --- |
| 实验结论：  在这次实验中，我通过对鸡蛋掉落问题的分析，得出解决问题的动态规划方程。第一种暴力法的时间效率很低，本质上是一种穷举法。在自顶向下和自底向上的对比中，明白就算时间复杂度理论上相同，实际运行中也会因为其他因素导致运行时间有较大差异。在第逆向思维算法中，用了不同的思路，得出了不同的动态规划方程，这说明动态规划只是一个思想，并不是解决问题的某个具体方法，同一个问题可以有不一样的动态规划解法。  总之，本次实验思想难度不大，重点在于熟悉理解动态规划的思想后进行应用。然后设计策略进行优化。 |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  年 月 日 |
| 备注： |

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后10日内。