实验五

# 1．背包问题递归、备忘录、动态规划时间表

如上表所示，即为普通递归方法随着选取物品个数n的改变执行时间变化图。

如上表所示，即为备忘录方法随着选取物品个数n的改变执行时间变化图。

如上表所示，即为动态规划随着选取物品个数n的改变执行时间变化图。

# 2．产生差异的原因

从三张表可以看出，不管当选取的物品数量是多少时，得到最优解所需要的时间总是最久的，并且所花费的时间随着所选取的物品个数n呈指数增长。反而在备忘录和动态规划中，备忘录在选取数量较少的时候花费的时间比动态规划所花费的时间要多一点，但是随着n的增长，动态规划花费的时间却比备忘录花费的时间要多一些，但是两者在时间上的差异也并不是特别明显。产生这样差异的原因有几点：

1. 普通的递归方法递归次数多，且时间复杂度为，所以花费的时间会最多。
2. 备忘录也是通过递归的方式来查找最优解，但是与普通递归的区别又在于备忘录会记录之前已经得到的一个子问题的最优解，在第二次需要使用到子问题的最优解的时候可以直接使用。
3. 动态规划是自底向顶的来求出最终的最优解，通过先求出每个子问题的最优解，最终返回需要的最优解。
4. 其中备忘录也可以称为动态规划，这种动态规划的方式是自顶向底的方式，从表中可以看出他与动态规划的时间差异并不大，但是在n较小时花费的时间要多一些，在n较大的时候花费的时间却要少一些。这跟备忘录方法的原理有关，备忘录采用的是递归的方式来存储计算过子问题最优解，所以在n较小时，备忘录递归和返回值需要花费一部分时间，比起直接通过循环存储子问题最优解相对来说慢一点。但是当n增大时，通过循环获得的最终解需要把每个子问题的解都求出来，而使用备忘录的方式却不需要把每个子问题的解都求出来，只需要求出自己需要的就可以了，这时，递归和返回值的时间便低于循环花费的时间，可以忽略不记了。

# 3．代码

package lab;

/\*\*

\* @author 欧阳建荣

\* @create 2020-12-02 12:02

\*/

public class BackPackage {

public static void main(String[] args) {

runBack();

}

private static void runBack() {

int[] v = new int[]{8, 10, 6, 3, 7, 2, 12, 4, 7, 2, 6, 4, 5, 10, 11, 13, 5, 3, 8, 12, 15, 6, 2, 1, 5, 12, 11, 9, 10, 4};

int[] w = new int[]{4, 6, 2, 2, 5, 1, 5, 3, 10, 3, 6, 3, 4, 7, 8, 9, 3, 1, 6, 3, 9, 2, 3, 5, 7, 5, 9, 10, 2, 5};

int max = 30;

int capacity = 20;

for (int i = 5; i <= max; i += 5) {

System.out.println("n = " + i + ", capacity = 20:");

long startTime = System.nanoTime();

System.out.print("递归: " + val(i, capacity, w, v));

long endTime = System.nanoTime();

System.out.println(", time: " + (endTime - startTime));

startTime = System.nanoTime();

System.out.print("备忘录: " + memorandum(i, capacity, w, v));

endTime = System.nanoTime();

System.out.println(", time: " + (endTime - startTime));

startTime = System.nanoTime();

System.out.print("dp: " + dpTable(i, capacity, w, v));

endTime = System.nanoTime();

System.out.println(", time: " + (endTime - startTime));

}

}

/\*\*

\* 递归实现，返回第1至第i个物品可装入的最大值，当前背包容量c，w为重量，v为对应的价值

\*

\* @param i

\* @param c

\* @param w

\* @param v

\* @return

\*/

public static int val(int i, int c, int[] w, int[] v) {

if (i == 0) {

return 0;

} else if (c < w[i - 1]) {

// 如果当前背包容量小于物品的重量，则不放入，最大值同val(i-1,c,w,v)

return val(i - 1, c, w, v);

} else {

// 容量足够，选择放入和不放入

int a = val(i - 1, c - w[i - 1], w, v) + v[i - 1];

int b = val(i - 1, c, w, v);

return Math.max(a, b);

}

}

public static int memorandum(int m, int c, int[] w, int[] v) {

int[][] memo = new int[m + 1][c + 1];

// 初始化

for (int i = 0; i < c + 1; ++i) {

memo[0][i] = 0;

}

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

memo[i][0] = 0;

}

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

for (int j = 1; j < c + 1; ++j) {

memo[i][j] = -1;

}

}

return memo(m, c, w, v, memo);

}

private static int memo(int i, int c, int[] w, int[] v, int[][] memorandum) {

if (memorandum[i][c] != -1) {

return memorandum[i][c];

}

int r = 0;

if (c >= w[i - 1]) {

r = Math.max(memo(i - 1, c - w[i - 1], w, v, memorandum) + v[i - 1], memo(i - 1, c, w, v, memorandum));

} else {

r = memo(i - 1, c, w, v, memorandum);

}

return memorandum[i][c] = r;

}

public static int dpTable(int m, int c, int[] w, int[] v) {

int[][] dp = new int[m + 1][c + 1];

for (int i = 0; i < c + 1; ++i) {

dp[0][i] = 0;

}

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

dp[i][0] = 0;

}

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

for (int j = 1; j <= c; ++j) {

dp[i][j] = dp[i - 1][j];

if (w[i - 1] <= j) {

dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - w[i - 1]] + v[i - 1]);

}

}

}

return dp[m][c];

}

}

# 4．执行结果

