01背包问题的递归和递推方法

> 记忆化搜索和动态规划

问题描述

- 已知有第i个物品重量为w_i，价值为v_i，以及背包的总容量W
- 求只能放前i个物品的情况下，容量为j的背包所能达到的最大总价值

求解思路

方程描述

- 在动态规划问题中，很重要的一点是先将状态描述好：`state(weight , i)`表示背包剩余容量为为`weight`，只能放前`i`个物品的背包中的物品最大总价值
- 定义好新加入的物品

```
typedef struct Item
{
    int value;
    int weight;
};

Item item[N];
```

- 定义好状态转移方程

```
state(weight , i) = max{ state(weight , i - 1) , state(weight - item[i].weight , i - 1) + item[i].value }
```

`state(weight , i - 1)`表示对于`item[i]`选择不取

> 表示 $weight$ 重量的物品最优选取方案中不包括第 i 个物品

`state(eight - item[i].weight , i - 1) + item[i].value`表示对于`item[i]`选择取

> 表示 $weight$ 重量的最有选取方案包括第 i 个物品

中间状态填充

我们知道，物品的质量并非狭义连续的，换而言之，只考虑已有物品的情况下，总质量增长的最小单元并不是1

比如，六个物品的质量为$1,3,7,8,11$，这样，$weight$ 的取值只能是这些值的加和，假如我们需要求取 $weight = 13$ 的情况，就会发现事实上 13 并不是这些值的加和，而是处于两个加和之间的**中间状态**

很有可能，在某个状态转移方程中需要使用到这些质量不属于任何物品质量加和的“中间状态”

因此，在给出一个状态转移方程后，**需要对中间状态的情况进行填充**

我们可以整理出状态转移的核心代码：

递归版核心代码

```
    int state(int weight , int i)
    {
        if (!i)
        {
            return 0;
        }
        else if (weight <= item[i].weight)
        {
            return state(weight , i - 1);
        }
        else
        {
            return max( state(weight , i - 1) , state(weight - item[i].weight , i
- 1) + item[i].value )
        }
    }
```

递归版本优化

通过一个二维数组来记录搜索的结果，实现记忆化搜索，就能够实现重复部分不进行反复搜索，进而达到节约运算次数的目的

```
    int state[weight + 1][cnt + 1];  // 设置一个 weight * cnt 大小的数组
    memset(state , 0 , sizeof(state)); // 将初值全部设置为0
    int state(int weight , int i)
    {
        if (!i)
        {
            return 0;
        }
        else if (weight <= item[i].weight)
        {
            if(!state[weight][i])  // 如果值不为0，说明这个状态已经求解过，直接返回求
解得到的状态即可
            {
                state[weight][i] = state(weight , i - 1);
            }
            return state[weight][i];
        }
        else
        {
            if(!state[weight][i])
```

```
            {
                state[weight][i] =  max( state(weight , i - 1) , state(weight -
item[i].weight , i - 1) + item[i].value )
            }
            return state[weight][i];
        }
    }
```

递推版核心代码

```
    int state[weight + 1][cnt + 1];  // 设置一个 weight * cnt 大小的数组
    memset(state , 0 , sizeof(state)); // 将初值全部设置为0
    for (int i = 1; i <= n; ++i)
    {
        for (int j = Weight; l >= item[i].weight; --l)
        {
            state[j] = max(state[j], state[j - item[i].weight] + item[i].value);
        }
    }
```

问题理解&时间复杂度分析

- 由于对于一件物品只有取和不取两种情况，相当于计算机中的 0 和 1 （有或无），这类问题才被称为01背包问题
- 事实上，01背包问题其实就是一个**不断向前提问：到底包不包括第 i 个物品**的过程
- 01背包问题中动态规划和 dfs 的不同之处
    - 首先我们需要了解 dfs 方法如何解决这类问题：对每一个分支进行深度优先搜索（先不考虑剪枝的问题），一共需要$O(2^n)$的时间复杂度
    - 然后考虑01背包问题，事实上，01背包问题的最劣时间复杂度为$O(weight \times n)$
- 01背包问题的数学本质其实是**数学归纳法**的演变

程序代码清单：已经附在本文中