*三种背包问题以及其解法*

**Intro:**

背包问题(Knapsack problem)是一种组合优化的NP完全问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。相似问题经常出现在商业、组合数学，计算复杂性理论、密码学和应用数学等领域中。也可以将背包问题描述为决定性问题，即在总重量不超过V的前提下，总价值是否能达到X？

下文统一用V表示背包的总容量，N表示物品数量

如果我们使用深搜的办法，枚举每一样物品是否被装入背包的话，时间复杂度将会是O()，这显然是不可接受的。因此，接下来我们将介绍三种不同的背包问题的动态规划解决算法，他们的复杂度如下:

|  |  |  |
| --- | --- | --- |
| 背包类型 | 时间复杂度 | 空间复杂度 |
| 01背包 | O(NV) | O(NV)或O(V) |
| 完全背包 | O(NV) | O(V) |
| 多重背包 | O(NV)或O(VΣki) | O(NV)或O(VΣki) |

**Part I:***01背包*

01背包是**背包问题**中最简单的问题。可以基本阐述为：用一个容量有V的背包去装数量为N，且体积和其价格各不相同的物品，使装满背包后的总价值最大。01背包问题的约束条件是给定几种物品，每种物品有且只有一个，并且有权值和体积两个属性。

我们假设目前有空间为V的背包，一共有N个物品，它能装下的物品的最大总价值为x。对于这个问题，它的答案可以转化为以下两个问题的答案中的最大值：

1. *空间为V-(第N号物品的空间)的背包，共有(前)N-1个物品，它能装下的物品的最大总价值为x1*
2. *空间为V的背包，共有(前)N-1个物品，它能装下的物品的最大总价值为x2*

这两个问题分别对应着取或不取N号物品的时候的最优解，对二者取max就可以得到我们想要的答案。

编写程序时，每件物品的体积我们用一个数组w[n]来表示，每件物品的价值则用p[n]来与之一一对应。定义一个二维数组dp[n][V]，dp[i][j]表示前i个物品在j的空间下的能取得的最大价值。根据上文的方法，类似的，我们列出以下的状态转移方程：

dp[i][j]=max(dp[i-1][j], dp[i-1][j–w[i]]+p[i]);

我们在判断是否选择第i件物品时，比较 **不放入物品i时前i-1次选择的总价值** 和 **将第i件物品放入背包后的总价值** 的大小，取二者的最大值。

求解背包问题的具体过程



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 0 | 0 | 0 | 0 | 50 | 50 | 50 | 50 | 50 | 50 |
| 2 |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |

1号物品

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 0 | 0 | 0 | 0 | 50 | 50 | 50 | 50 | 50 | 50 |
| 2 | 0 | 0 | 20 | 20 | 50 | 50 | 50 | 70 | 70 | 70 |
| 3 |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |

2号物品

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 0 | 0 | 0 | 0 | 50 | 50 | 50 | 50 | 50 | 50 |
| 2 | 0 | 0 | 20 | 20 | 50 | 50 | 50 | 70 | 70 | 70 |
| 3 | 0 | 0 | 20 | 40 | 50 | 50 | 50 | 70 | 80 | 80 |
| 4 |  |  |  |  |  |  |  |  |  |  |

3号物品

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 0 | 0 | 0 | 0 | 50 | 50 | 50 | 50 | 50 | 50 |
| 2 | 0 | 0 | 20 | 20 | 50 | 50 | 50 | 70 | 70 | 70 |
| 3 | 0 | 0 | 20 | 40 | 50 | 50 | 50 | 70 | 80 | 80 |
| 4 | 0 | 0 | 20 | 40 | 50 | 50 | 50 | 70 | 80 | 90 |

4号物品，可以看到最终算出的答案是90，现在你非常开心

完整解决01背包问题的代码如下：

//num:物品数量 container:背包空间

void Resolve\_KnapSack()

{

    for (int i=1;i<=num;++i) //遍历物品，规划最优选择

    {

        for (int j=1;j<=container;j++)

        {

            if (j>=w[i])

            {

                dp[i][j]=max(dp[i-1][j], dp[i-1][j–w[i]]+p[i]);

            }

            else

            {

                dp[i][j]=dp[i-1][j];

            }

        }

    }

}

**Part II:***完全背包*

完全背包是在01背包基础上衍生出的问题，可以用同样思想的动态规划算法来解决。变形后的问题则为：有N种物品和一个容量为V的背包，每种物品都有无限件可以装入，我们用数组w[i]来储存物品的体积，用p[i]来储存物品的价值，编程计算将哪些物品装入背包可使这些物品的体积总和不超过背包容量，且价值总和最大。

由于每种物品都可以有无限件可拿，所以考虑计算当背包空间不断扩大时的最优方案。我们**枚举并比较每一个满足j>=w[i]的i号物品，在选择它时所得到的价值大小，进而计算出最优解**（j表示当前的背包内物品总体积）。此外，需要注意的是选择什么都不取也是一种方案，因此我们让初始的dp[j]=dp[j-1]。

列出状态转移方程：

dp[j]=max(dp[j],dp[j-w[i]]+p[i]);

*i的取值从1到N*

道理同上，dp为储存最优子问题的数组。

解决完全背包问题的具体代码如下：

//num:物品数量 container:背包空间

void Resolve\_KnapSack()

{

    for(int j=1;j<=container;++j)

{

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

        for(int i=1;i<=num;++i) //遍历物品，规划最优选择

        {

            if(j>=w[i])

            {

                dp[j]=max(dp[j],dp[j-w[i]]+p[i]);

            }

        }

    }

**Part III:***多重背包*

在原本的问题上，我们新增加以下条件：每一种物品，都有个可以取最多ki个。

我们可以直接把这k个相同的物品拆开，当成k个不同的物品做一遍01背包，这样子的时间复杂度是O()。当较小的时候，我们可以这么做，但是当它较大的时候，直接拆分就不那么好用了。

这个问题存在不同的解法，这里只介绍其中一种较为简单的做法。

我们可以将这k个相同物品通过二进制进行拆分，使得拆分以后仍然可以用这些物品来表示1到k个原物品（比如，13拆成1+2+4+6后就可以表示1到13中的任意数字）。类似的，我们将k拆分成1,2,...,,k-+1（x为满足方程的解中最大的整数解），就可以表示从1到k的任意一个数字。

|  |  |  |  |
| --- | --- | --- | --- |
| 物品名称 | 物品个数 | (单个)物品价值 | 占用空间 |
| A(原物品) | 11 | 3 | 4 |
| B(1个原物品) | 1 | 3 | 4 |
| C(2个原物品) | 1 | 6 | 8 |
| D(4个原物品) | 1 | 12 | 16 |
| E(4个原物品) | 1 | 12 | 16 |

在这张表格中可以看到，11个A物品被拆分成了4个物品(B、C、D、E)

因此，对与k个i类物品，我们将它拆成了logk个不同的物品，在此基础上做一次01背包就可以在O()的时间复杂度内解决该问题。实现代码如下：

//v[i]:i号物品的价值 w[i]:i号物品占用的空间 dn:拆分后物品个数 num:物品数量

int dn;

void divide(int k,int val,int weight)

{

    int x=1;

while(k-x>=0)

{

        p[++dn]=x\*val;

        w[dn]=x\*weight;

        k-=x;x=x\*2;

    }

}

int pre()

{

    int k,val,weight;

for(int i=1;i<=num;++i)

{

        scanf("%d%d%d",&k,&val,&weight);

        divide(k,val,weight);

    }

}

//接下来是普通的01背包

**Part IV:***结语*

首先要指出的是，背包的动态规划解法并不适用与所有情况。例如，当背包的空间大小W非常大而物品数量N很小的时候，动态规划算法的时间复杂度反而远高于深搜等暴力算法。

除了上述的三种常见的背包以外，还有其他许多不同的背包问题，感兴趣的同学可以进一步的深入学习，希望这篇科普短文能为你的编程学习提供一些灵感，谢谢~

本期周报相关代码已经上传github，欢迎前往查阅各期周报代码。

算法周报代码仓库：https://github.com/sysuacmm/AlgorithmnWeekly

*文案：杨智桓，游少濠*