# 2022秋-算法设计与分析

# 动态规划算法分析实验报告

|  |  |
| --- | --- |
| 姓名： |  |
| 学号： |  |

|  |
| --- |
| 2022年11月12日 |

目录

[2022秋-算法设计与分析 1](#_Toc119229733)

[动态规划算法分析实验报告 1](#_Toc119229734)

[1. 实验要求 2](#_Toc119229735)

[2. 实验报告 2](#_Toc119229736)

[2.1 基于两种递归公式的动态规划算法实现 2](#_Toc119229737)

[2.2 算法打桩以记录关键操作次数 4](#_Toc119229738)

[2.3 统计两种算法平均运行时间和关键操作次数 5](#_Toc119229739)

[2.4 改变物品种类规模并进行统计分析 6](#_Toc119229740)

[2.4 使用一维数组解决背包问题 8](#_Toc119229741)

[3. 实验总结 9](#_Toc119229742)

# 1. 实验要求

对动态规划算法进行分析，理解算法的工作流程。具体要求如下：

1. 针对整数背包问题，实现基于两种递推公式的动态规划算法；

2. 在算法中打桩，记录关键操作次数（例如entry查表次数等）；

3. 以物品种类的大小n为输入规模，固定n，随机产生大量测试样本，统计两种算法的平均运行时间和关键操作次数，并进行记录；

4. 改变物品种类规模，对不同规模问题各算法的结果对比分析，通过统计画图，与理论值进行对照分析，并撰写实验报告。

5. 考虑使用一维数组的方式解决整数背包问题，并记录其平均运行时间和关键操作次数，与上述两种算法进行对比。

# 2. 实验报告

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。问题的名称来源于如何选择最合适的物品放置于给定背包中。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。

## 2.1 基于两种递归公式的动态规划算法实现

背包问题的子问题由参数k, y界定。其中，k表示考虑对物品1, 2, …, k的选择，y表示背包总重量不超过y，表示转入前k种物品，总重不超过y的最大价值。基于子问题的划分，可以得出两种递归公式：

第一种递推公式：

第二种递推公式：

下面分别实现基于两种递推公式的背包问题动态规划算法。首先创建头文件KnapSack.h，声明两个函数KnapSack1()和KnapSack2()，用于实现两种解法；然后分别编写KnapSack1.cpp和KnapSack2.cpp，分别实现KnapSack1()和KnapSack2()函数。

在KnapSack1()中，根据递推公式，使用三重循环来计算出每个F[i][j]的值，代码实现如下图所示：

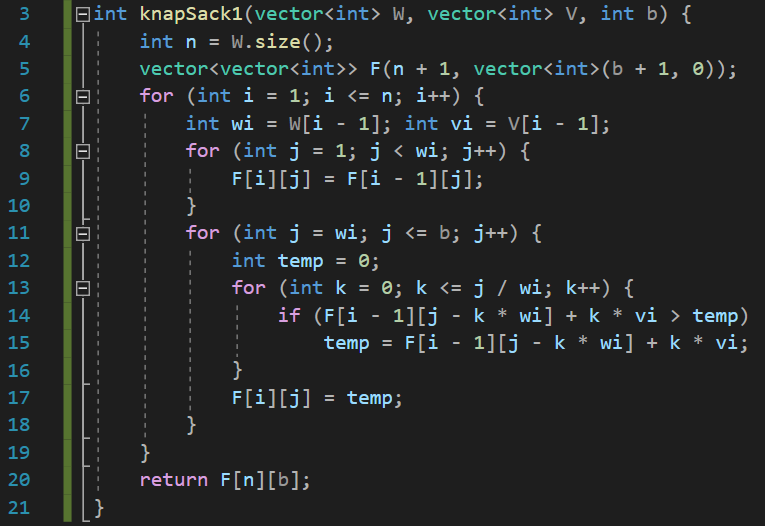


图1：knapSack1()函数实现

在KnapSack2()中，根据递推公式，使用二重循环来计算出每个F[i][j]的值，前二重循环与KnapSack1()基本一致，代码实现如下图所示：

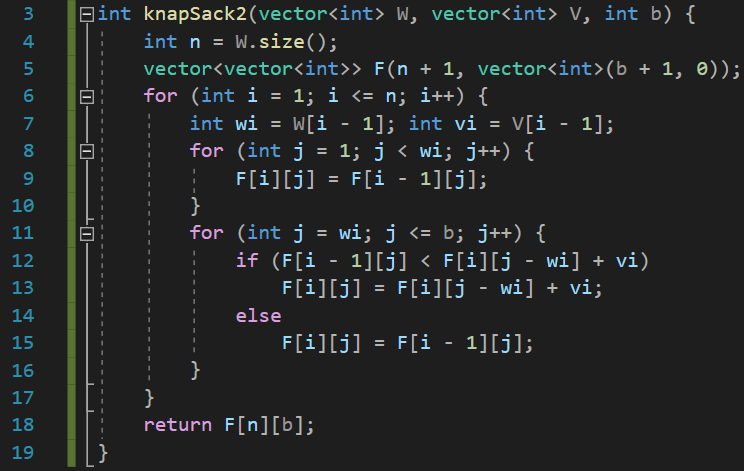


图2：knapSack2()函数实现

创建main.cpp，编写main()函数，定义W为{2, 3, 4, 7}，V为{1, 3, 5, 9}，调用knapSack1()和knapSack2()，结果如下图所示：

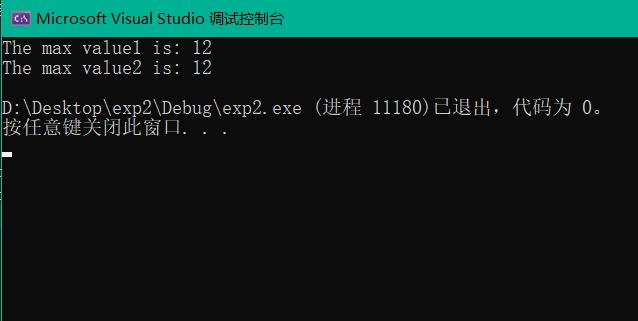


图3：运行结果

## 2.2 算法打桩以记录关键操作次数

定义“关键操作次数”为比较次数和访存次数。由于两种递推公式的实现中，写入内存的次数均为n\*b次，因此，由写入内存而引起的访存次数可以不纳入计算。

以knapSack1()函数为例进行打桩，定义整型变量compr和visit分别表示比较次数和访存次数，在进入每个for循环和整个for循环结束后，比较次数均增加1，在每次访问内存后，访存次数均增加1，代码实现如下图所示：

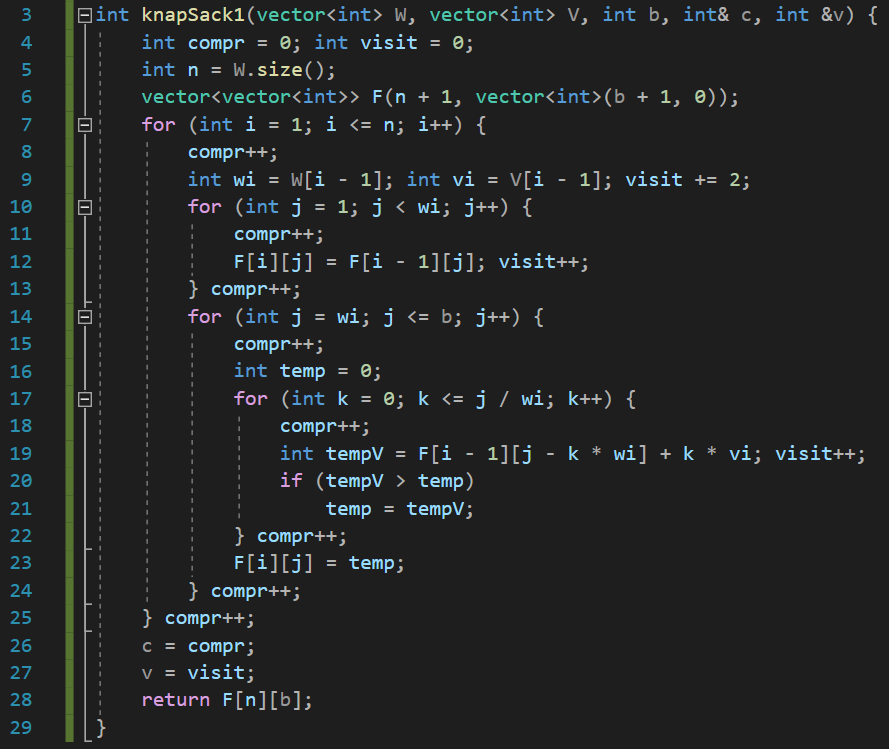


图4：knapSack1()函数打桩

类似地，对knapSack2()函数打桩，代码实现如下图所示：

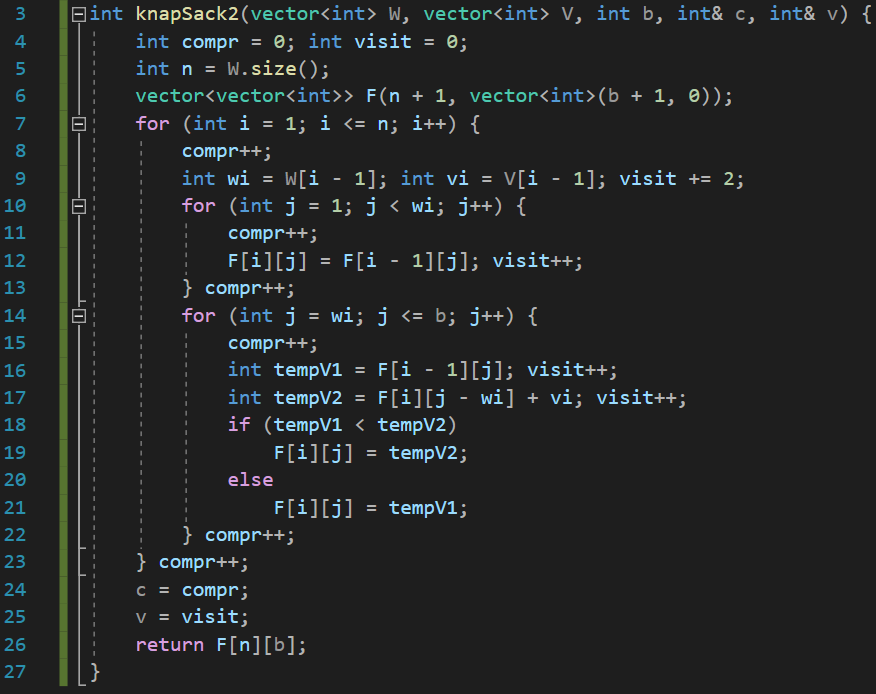


图5：knapSack2()函数打桩

修改main()函数使其输出比较次数和访存次数，运行程序，结果如下图所示：



图6：打桩后运行结果

## 2.3 统计两种算法平均运行时间和关键操作次数

修改main()函数，固定物品种类的大小n，随机产生各物品的重量和价格，使用clock()函数计算出运行时间，使用变量c和v记录关键操作次数，最后求出平均运行时间和关键操作次数。修改后的main()函数如下图所示：

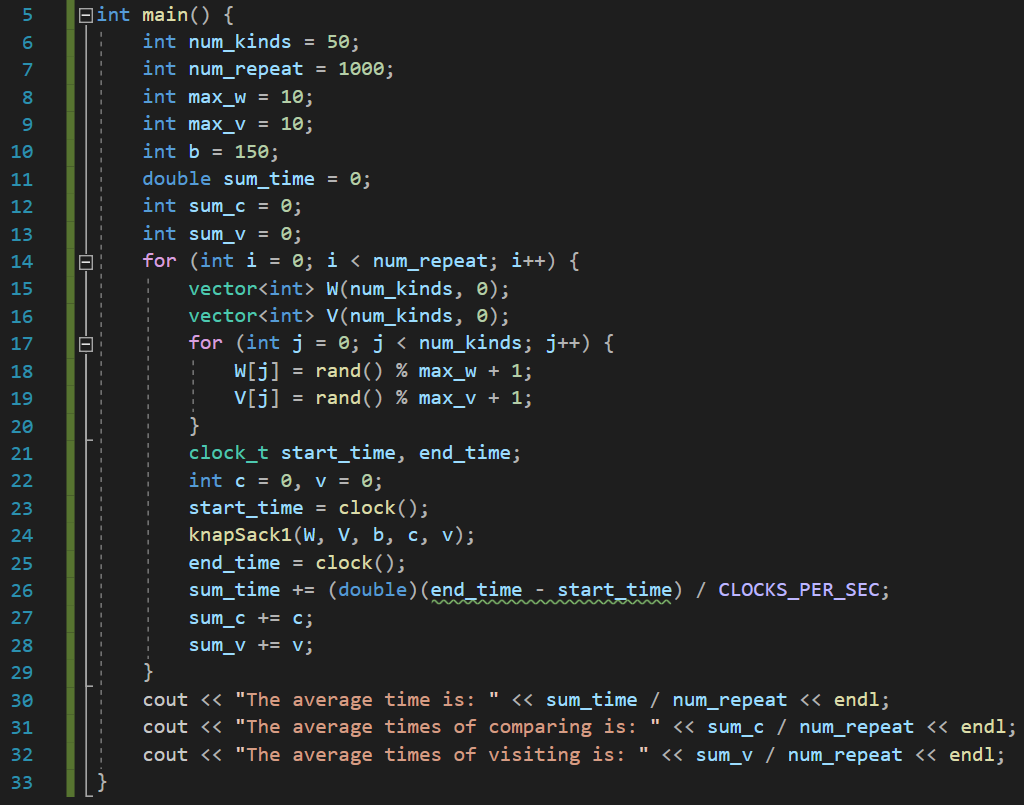


图7：修改后的main()函数

设置物品种类数量为50，循环次数为1000，物品最大重量为10，物品最大价值为10，背包最大承重为150，运算结果如下表所示：

表1：基于两种递推公式的算法运行表现统计

|  |  |  |  |
| --- | --- | --- | --- |
| 算法种类 | 平均运行时间 | 平均比较次数 | 平均访存次数 |
| 算法1 | 0.005917 | 185730 | 171127 |
| 算法2 | 0.000829 | 7651 | 14875 |

显然，算法2的运行效率远远高于算法1，因为算法1相比于算法2多了一层循环，由该循环造成比较次数和访存次数远远大于算法2，而由于较多的比较和访存，导致算法1的运行时间也较长。

## 2.4 改变物品种类规模并进行统计分析

其他参数保持不变，设置物品种类规模分别为50, 100, 150, 200, 250, 300，分别运行程序并统计出两种算法的平均运行时间，平均比较次数，平均访存次数。

不同物品种类规模下的平均运行时间统计结果如下图表所示：

表2：不同物品种类规模下的平均运行时间

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 算法种类 | 50 | 100 | 150 | 200 | 250 | 300 |
| 算法1 | 0.005946 | 0.012822 | 0.019723 | 0.025723 | 0.031972 | 0.038218 |
| 算法2 | 0.000822 | 0.001776 | 0.002421 | 0.003128 | 0.003961 | 0.005351 |

图8：不同物品种类规模下的平均运行时间

不同物品种类规模下的平均比较次数统计结果如下图表所示：

表3：不同物品种类规模下的平均比较次数

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 算法种类 | 50 | 100 | 150 | 200 | 250 | 300 |
| 算法1 | 185730 | 371003 | 555668 | 741937 | 926712 | 1111418 |
| 算法2 | 7651 | 15301 | 22951 | 30601 | 38251 | 45901 |

图9：不同物品种类规模下的平均比较次数

不同物品种类规模下的平均访存次数统计结果如下图表所示：

表4：不同物品种类规模下的平均运行时间

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 算法种类 | 50 | 100 | 150 | 200 | 250 | 300 |
| 算法1 | 171127 | 341902 | 511869 | 683533 | 853714 | 1023817 |
| 算法2 | 14875 | 29749 | 44624 | 59501 | 74373 | 89249 |

图10：不同物品种类规模下的平均访存次数

显然，两种算法的分析中，物品种类规模与平均运行时间、平均比较次数和平均访存次数均呈线性关系，这与理论分析高度一致。

## 2.4 使用一维数组解决背包问题

使用一维数组解决背包问题其实是对knapSack2()的改进，knapSack2()中进行比较的是F[i – 1][j]和F[i][j- wi] + vi，可以将其改变为F[j]和F[j – wi] + vi的比较，代码实现如下图所示：

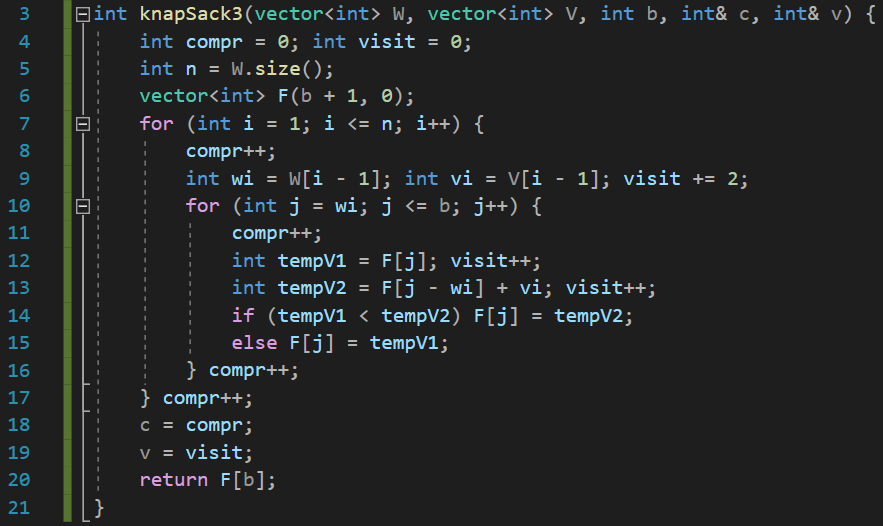


图11：knapSack3()代码实现

设置物品种类规模为300，其他参数保持不变，分别运行三种算法，统计出平均运行时间、平均比较次数和平均访存次数如下图所示：

表5：三种算法的运行表现统计

|  |  |  |  |
| --- | --- | --- | --- |
| 算法种类 | 平均运行时间 | 平均比较次数 | 平均访存次数 |
| 算法1 | 0.037527 | 1112400 | 1024799 |
| 算法2 | 0.005108 | 45901 | 89249 |
| 算法3 | 0.002085 | 44250 | 87899 |

显然，算法1的平均运行时间、平均比较次数和平均访存次数都远大于算法2和算法3；算法2和算法3的平均比较次数和平均访存次数基本一致，因为它们的算法逻辑都是一样的，需要进行比较和访存的位置数量也基本一样，但是算法3的运行时间更短，运行效率更高，我认为原因有2点：第1，算法3相比于算法2少了对j在[1, wi)之间的赋值过程，由于算法3仅有一维数组，相当于直接进行过了赋值；第2，算法3的一维数组比算法2的二维数组访存效率更高，在访存次数差不多的情况下，时间更短。

# 3. 实验总结

1. 通过本次实验，我亲自实现了完全背包问题的三种算法，对自己的编码能力和分析能力得到极大提升；

2. 通过本次实验，我亲自观察到了不同输入规模对三种算法的影响，第一次将对背包问题的复杂度分析应用于实践，当看到理论和实践高度一致时，内心非常激动；

3. 本次实验实现了完全背包问题，课外仍需学习一下01背包、多重背包等问题。