# 算法设计与分析实验报告

# 实验二 0-1背包

|  |  |
| --- | --- |
| **院系：** | **信息工程学院** |
| **班级：** | **15级网络工程2班** |
| **学号：** | **2015551621** |
| **姓名：** | **王 康** |
| **任课教师：** | **王 婷** |
| **成绩：** |  |

**湘 潭 大 学**

**2018年6月实验二 0-1背包**

1. **实验内容**

分别编程实现动态规划算法和贪心法求 0-1 背包问题的最优解，分析比较两种算法的时间复杂度并验证分析结果。

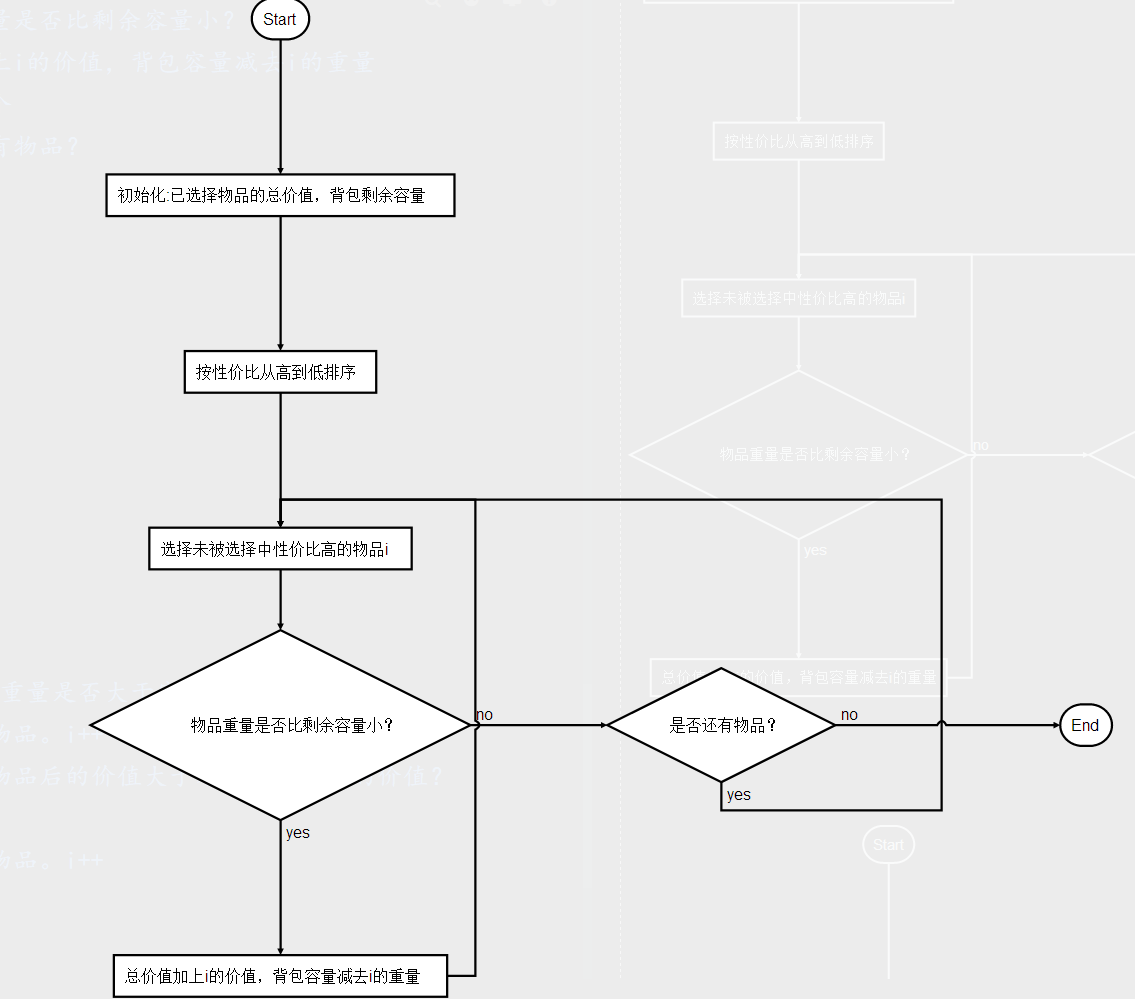
**二．实验目的**

1、掌握动态规划算法和贪心法解决问题的一般步骤，学会使用动态规划和贪心法解决实际问题；

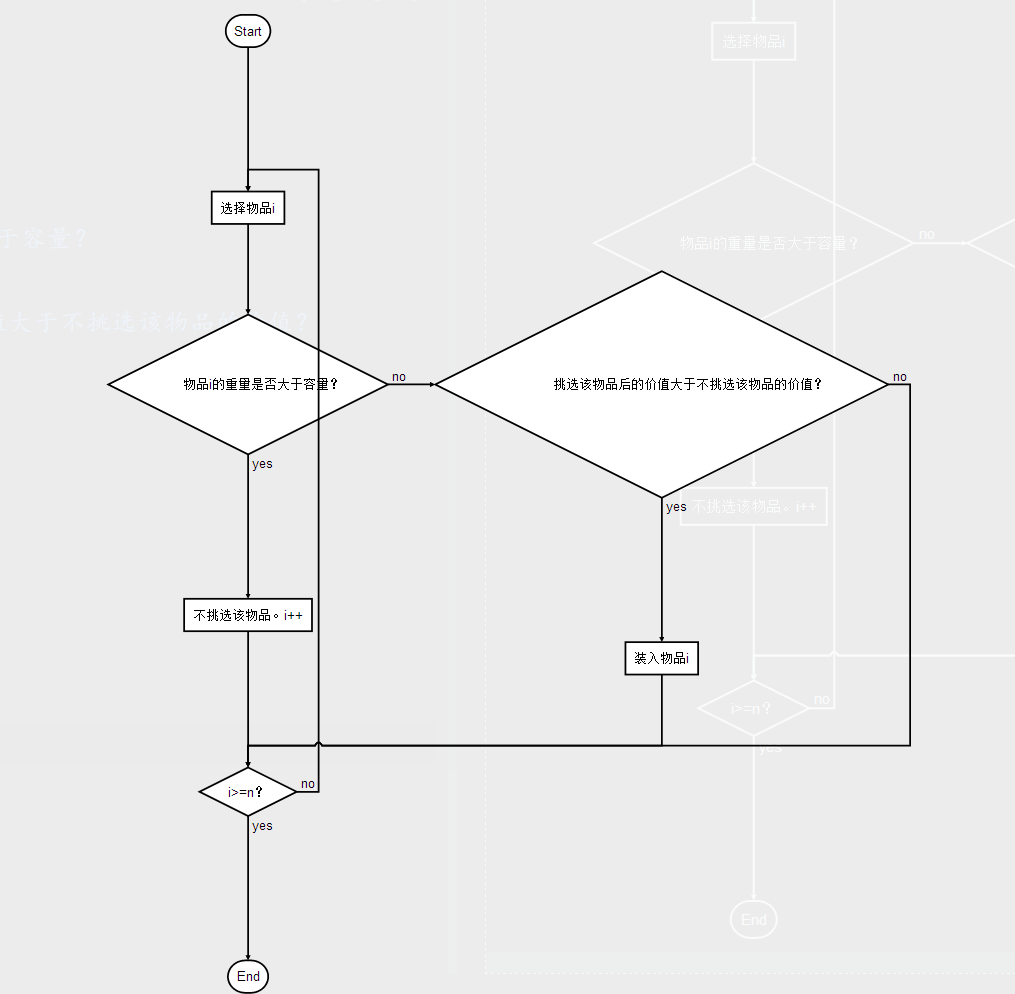
2、掌理解动态规划算法和贪心法的异同及各自的适用范围。

**三. 算法描述**

1、贪心法解决0-1背包算法流程图



1. 动态规划解决0-1背包算法流程图



**四. 算法实现**

1. **数据结构及函数说明**

Ⅰ、数据结构

1. 利用Goods结构体保存背包信息

struct Goods{

int w; //物品重量

int v; //物品价值

double quality; //物品价值性价比

int No; //物品编号

int selected = 0; //是否被选择

};

1. 用二维数组Table[n][n]维护选择的信息

Ⅱ、函数说明

**A、比较函数的回调函数cmp1()、cmp2()**

bool cmp1(Goods a, Goods b){

return a.quality > b.quality;

}

bool cmp2(Goods a, Goods b){

return a.No < b.No;

}

说明：作为sort函数的回调函数，对结构体中的元素排序。

1. **贪心算法Gready\_01()**

/\*\*

\* @author Swking

\* @method Gready\_01

\* @parame

\* Goods\* goods 物品

\* int n 物品总和

\* int capacity 背包容量

\* int& execNum 执行次数

\* @return valueSum 价值总和

\*/

int Gready\_01(Goods\* goods, int n, int capacity, int& exceNum);

说明：对物品利用贪心算法求解。

1. **贪心算法Gready\_01()**

/\*\*

\* @author Swking

\* @method Dynamic\_01

\* @parame

\* Goods\* goods 物品

\* int n 物品总和

\* int capacity 背包容量

\* int\*\* Table 解空间表

\* int& execNum 执行次数

\* @return valueSum 价值总和

\*/

int Dynamic\_01(Goods\* goods, int n, int capacity, int\*\* Table, int& execNum);

说明：对物品利动态规划算法求解。

1. **源程序代码**

#include<bits/stdc++.h>

#include<iostream>

#include<algorithm>

#include<cstdio>

#include<fstream>

#include<ctime>

using namespace std;

struct Goods{

int w; //物品重量

int v; //物品价值

double quality; //物品价值性价比

int No; //物品编号

int selected = 0; //是否被选择

};

bool cmp1(Goods a, Goods b){

return a.quality > b.quality;

}

bool cmp2(Goods a, Goods b){

return a.No < b.No;

}

/\*\*

\* @author Swking

\* @method Gready\_01

\* @parame

\* Goods\* goods 物品

\* int n 物品总和

\* int capacity 背包容量

\* int& execNum 执行次数

\* @return valueSum 价值总和

\*/

int Gready\_01(Goods\* goods, int n, int capacity, int& exceNum){

int valueSum = 0;

for(int i=0; i<n; i++){

exceNum++;

if(goods[i].w <= capacity){ //物品小于背包剩余容量，则被选择

goods[i].selected = 1;

capacity -= goods[i].w;

valueSum += goods[i].v;

}

}

return valueSum;

}

/\*\*

\* @author Swking

\* @method Dynamic\_01

\* @parame

\* Goods\* goods 物品

\* int n 物品总和

\* int capacity 背包容量

\* int\*\* Table 解空间表

\* int& execNum 执行次数

\* @return valueSum 价值总和

\*/

int Dynamic\_01(Goods\* goods, int n, int capacity, int\*\* Table, int& execNum){

//Table中的第一个量为物品编号，第二个量为总容量

int valueSum;

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

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

/\*

\* 1、j<goods[i-1].w时，放不下物品i，直接不选择，即Table[i][j] = Table[i-1][j]

\* 2、否则选择对与当前状态最优的结果，即选择 装入背包和不装入背包中价值的较大者，即

\* Table[i][j] = max(Table[i-1][j], Table[i-1][j-goods[i-1].w]+goods[i-1].v);

\*

\*/

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

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

}else{

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

}

execNum++;

}

}

/\* 通过Table表回溯寻找路径

\* 1、Table[i][j] = Table[i-1][j]时，说明第i个物品没有被选择，则回溯到Table[i-1][j]

\* 2、Table[i][j] = Table[i-1][j-goods[i-1].w]+goods[i-1].v,则第i个物品被选择，并回溯到装i物品之前Table[i-1][j-goods[i-1].w]

\*/

int i=n;

int j=capacity;

while(i>0&&j>0){

if(Table[i][j] == Table[i-1][j]){

i--;

}else{

j -= goods[i-1].w;

goods[i-1].selected = 1;

i--;

}

//execNum++;

}

valueSum = Table[n][capacity]; //capacity处的值则为背包总价值

return valueSum;

}

int main()

{

fstream fin,fout;

/\*

\* 模拟数据

\* 第一行为背包容量c

\* 第二行为物品个数n

\* 第三行为物品重量w

\* 第四行为物品价值v

\*/

fout.open("in.txt",ios::out);

cout << "producting the file..." << endl;

//第一组数据，贪心能得到最优解

fout << 5 << endl;

fout << 3 << endl;

fout << 1 << ' ' << 2 << ' ' << 3 << endl;

fout << 6 << ' ' << 10 << ' ' << 12 << endl << endl;

//第二组数据，贪心得不到最优解

fout << 15 << endl;

fout << 5 << endl;

fout << 5 << ' ' << 4 << ' ' << 7 << ' ' << 2 << ' ' << 6 << endl;

fout << 12 << ' ' << 3 << ' ' << 10 << ' ' << 3 << ' ' << 6 << endl << endl;

srand(time(NULL));

int exampleSum = 20 + rand()%30;

while(exampleSum--){

int capacity = 100 + rand()%100;

fout << capacity << endl;

int elementNum = 10 + rand()%30;

fout << elementNum << endl;

for(int i=0; i<elementNum; i++){

fout << 1 + rand()%99 << ' ';

}

fout << endl;

for(int i=0; i<elementNum; i++){

fout << 1 + rand()%99 << ' ';

}

fout << endl << endl;

}

fout.close();

/\*

\* 读取文件数据

\*/

cout << "reading the file..." << endl;

fin.open("in.txt",ios::in);

fout.open("out.csv",ios::out);

cout << "Computing..." << endl;

while(!fin.eof()){

int capacity, n;

n = 0;

fin >> capacity >> n;

if(n!=0){

Goods\* goods = new Goods[n+1];

for(int i=0; i<n; i++){

fin >> goods[i].w; //读物品重量w

goods[i].No = i+1; //从1开始编号

}

for(int i=0; i<n; i++){

fin >> goods[i].v; //读物品价值v

goods[i].quality = goods[i].v / goods[i].w; //计算物品性价比

}

/\*

\* 按性价比排序

\*/

sort(goods,goods+n,cmp1);

int valueSum;

/\*

\* 贪心

\*/

int execNum = 0;

valueSum = Gready\_01(goods, n, capacity, execNum);

/\*

\* 将计算结果写入文件

\*/

sort(goods,goods+n,cmp2);

fout << "Gready:" << ','<< "ValueSum:" << ',' << valueSum << ',' << "exceNum:" << ',' << execNum << endl;

fout << ',' << "No." << ',';

for(int i=0; i<n; i++){

fout << goods[i].No << ',';

}

fout << endl << ',' << "Selected" << ',';

for(int i=0; i<n; i++){

fout << goods[i].selected << ',';

}

fout << endl;

/\*

\* 动规

\*/

//清除贪心的结果

for(int i=0; i<n; i++){

goods[i].selected = 0;

}

//创建表格，储存抉择路径

int\*\* Table = new int\* [n+1];

for(int i=0; i<=n; i++){

Table[i] = new int[capacity+1];

}

//初始化第一列为0

for(int i=0; i<=n; i++){

Table[i][0] = 0;

}

//初始化第一行为0

for(int i=0; i<=capacity; i++){

Table[0][i] = 0;

}

execNum = 0;

valueSum = Dynamic\_01(goods, n, capacity, Table, execNum);

fout << "Dynamic:" << ',' << "ValueSum:" << ',' << valueSum << ',' << "exceNum:" << ',' << execNum << endl;

fout << ',' << "No." << ',';

for(int i=0; i<n; i++){

fout << goods[i].No << ',';

}

fout << endl << ',' << "Selected" << ',';

for(int i=0; i<n; i++){

fout << goods[i].selected << ',';

}

//打印Table

fout << endl << "Table" << ',';

for(int i=0; i<=capacity; i++){

fout << i << ',';

}

fout << endl;

for(int i=0; i<=n; i++){

fout << i << ',';

for(int j=0; j<=capacity; j++){

fout << Table[i][j] << ',';

}

fout << endl;

}

fout << endl;

//释放空间

for(int i=0; i<n; i++){

delete [] Table[i];

}

delete[] Table;

}

}

fout.close();

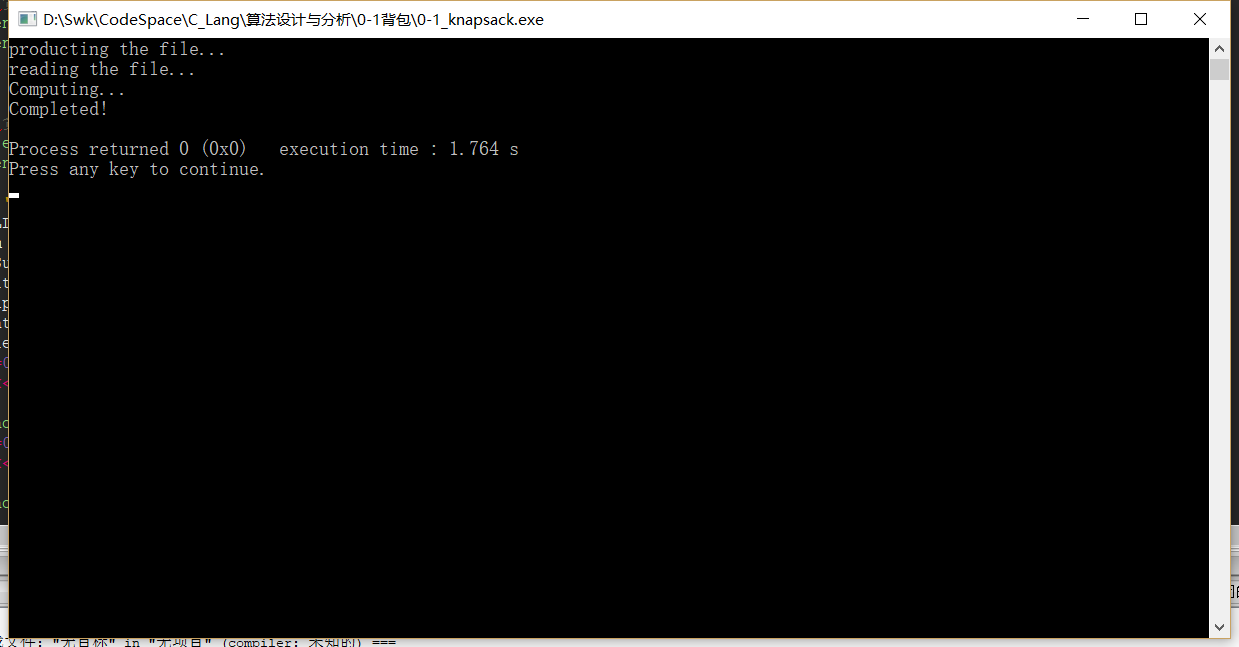
fin.close();

cout << "Completed!" << endl;

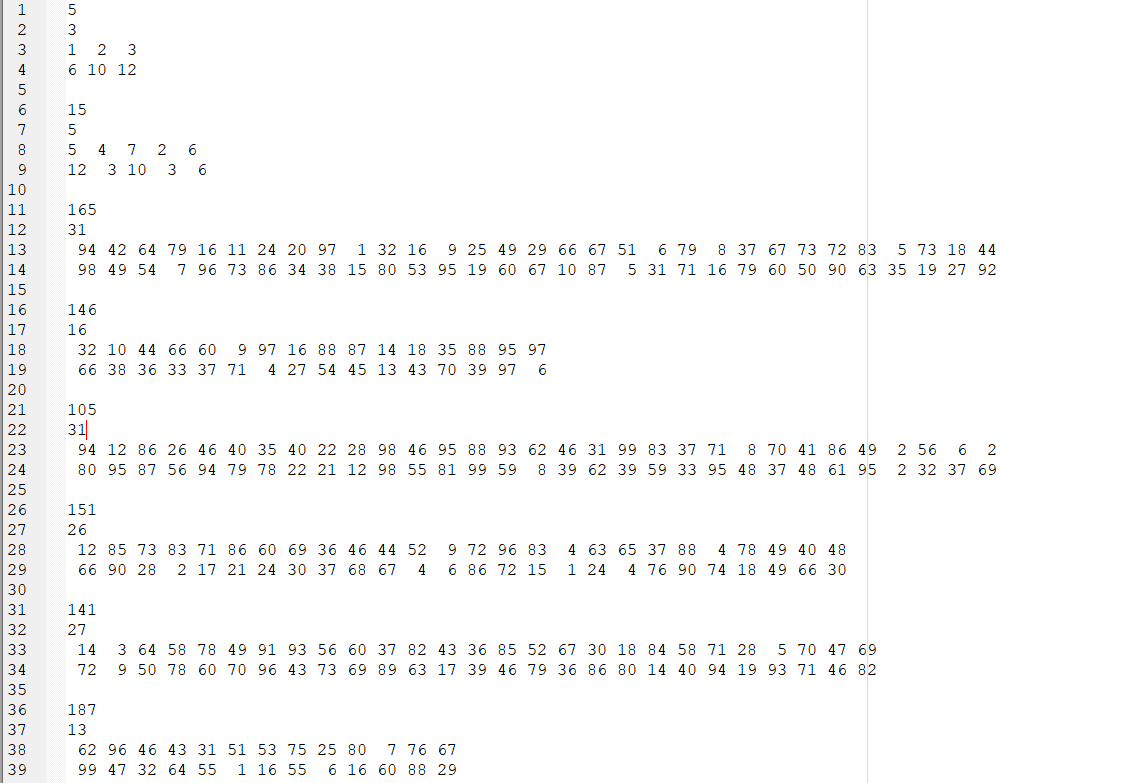
return 0;

}

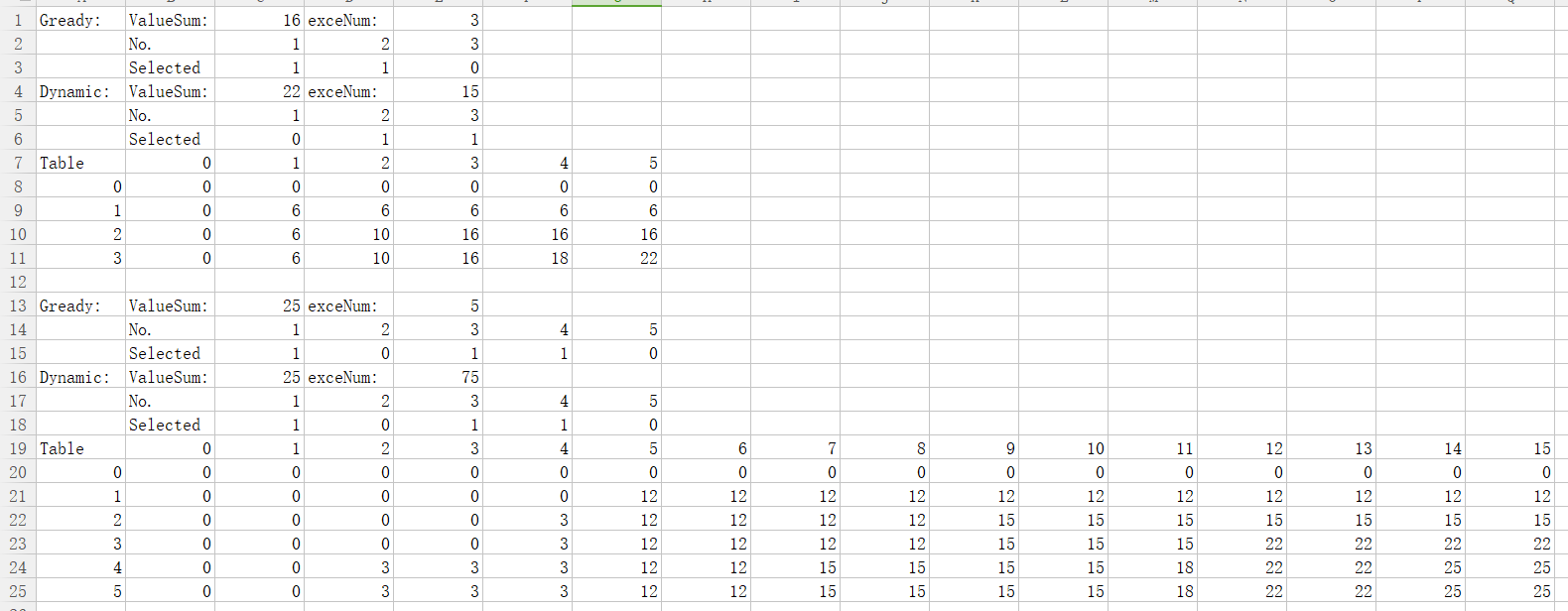
1. **程序运行结果**
2. 控制台输出信息



1. 样本文件in.txt，前面两组数据为手动模拟写入，后面的数据全为随机生成。每个样例共4行，第一行为背包容量，第二行为物品个数，第三行为物品的重量，第四行是物品对应重量的价值。



1. 运行程序将结果输出到out.csv文件中，如图：Gready处为用贪心法求0-1背包得到的结果，ValueSum为价值总和，exceNum为程序执行次数，No.为物品的编号，Selected为物品的选择状态。同样Dynamic处为动态规划解决0-1背包求得的结果数据，其中Table表为记录表。



1. **实验结果分析**
2. 贪心法求解0-1背包
   1. 贪心法每次对物品做出判断时，必定会给出物品的状态（选择或者不选），解决问题需要对n个物品做出判断，所以时间复杂度为：

T(n) = O(n)

* 1. 由第一个和第二个数据例子可以看出，贪心法有时并不能得到最优解，贪心法的核心思想是每次都选择当前的最优解，并不考虑全局，每次挑选了性价比最大的物品，可能使得最后背包空间剩余，使得没有被利用，从而不是最优解。

1. 动态规划求解0-1背包
   1. 从实验结果可以看出，动态规划求解0-1背包其实就是填Table表的过程，填完Table表后，求解过程也就结束了，则时间复杂度为：

T(n) = O(nc) {n|n为物品个数；c|c为背包容量}

* 1. 动态规划是将背包问题划分成一个一个的子结构，只需要在子结构中找到的是最优解，那么子结构合并起来也必定是最优解。所以用动态规划求得的解必定是最优解。

1. 贪心法与动态规划求解0-1背包的比较
   1. 贪心法求0-1背包问题虽然时间复杂度只是线性时间，但是他有时求得的解不是最优解。
   2. 动态规划时间复杂度比贪心法要高，但是每次求得的解是最优解，主要是要维护Table表。
2. **结论**
3. 贪心法的思想是每次都选择当前的最优解，并不考虑全局，每次挑选了性价比最大的物品，可能使得最后背包空间剩余。
4. 贪心法能得到0-1背包问题的一个解，但是该解可能不是最优解。
5. 贪心法思想简单，时间复杂度底O(n)，只需要对n件物品做出判断即可。
6. 动态规划的思想是通过状态转移方程来构建Table表。
7. 动态规划通过Table表回溯求解过程，可以找到一组最优值
8. 动态规划能等到一组最优解，但耗时较多，当背包容量c较大时，效率不是很高。