重 庆 大 学

学 生 实 验 报 告

实验课程名称 算法设计与分析

开课实验室 DS1501

学 院 大数据与软件学院 年级 2023 专业班 软工 01

学 生 姓 名 学 号

开 课 时 间 2024 至 2025 学年第 2 学期

总 成 绩	
教师签名	付春雷

大数据与软件学院制

《算法设计与分析》实验报告

开课实验室：DS1501 **2025 年 6 月 2 日**

学院	大数据与软件学院	年级、专业、班	23 级软件工程 01	姓名		成绩	
课程名称	算法设计与分析	实验项目名称	贪心法实验和动态规划法实验	指导教师	付春雷		
教师评语							

教师签名：

年 月 日

一、实验目的

- 掌握贪心法的一般求解过程，掌握动态规划法求解的基本步骤。
- 熟练掌握“一个序列中出现次数最多元素问题的求解算法”的实现，熟练掌握“0/1 背包问题的动态规划算法”的实现。
- 主要任务：实现教材配套实验指导书“第 2 章 2.7.1 小节 求解一个序列中出现次数最多的元素问题”和“第 8 章 求解 0/1 背包问题的动态规划算法”。

二、使用仪器、材料

PC 微机；
Windows 操作系统，VSCODE+MINGW64 编译环境（不限）；

三、实验步骤

1. 求解一个序列中出现次数最多的元素问题

使用贪心法求解，嵌套两层循环遍历次数

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    int n;
    cin >> n;
    vector<int> as(n),count_as(n);

    for (int i = 0; i < n; i++)
    {
        cin >> as[i];
    }
    int count_max = 0,value_min=0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (as[j] == as[i]) {
                count_as[i]++;

            }
```

```
            }
            if (count_as[i] >= count_max) {
                count_max = count_as[i];
                value_min = as[i];
            }
        }

        for (int i = 0; i < n; i++) {
            if (count_max == count_as[i]&&as[i]<value_min)
            {
                value_min = as[i];
            }
        }

        cout << value_min << endl;
        return 0;
    }
```

2. 有 n 个重量分别为 w1、w2、…、wn,的物品(物品编号为 1~n),它们的价值分别为 v1、v2、v3…、vn..,给定一个容量为 W 的背包。设计从这些物品中选取一部分物品放人该背包的方案,每个物品要么选中要么不选中,要求选中的物品不仅能够放到背包中,而且具有最大的价值。那么现在，用 c++实现该 0/1 背包问题，我们给出有 1，2，3，4 个物品，重量分别为 5，3，2，1，价值分别为 4，4，3，1，当 W=6 时求出所有解和最佳解，考虑使用动态规划法求解最优

在遍历的同时使用回溯法求解了所有符合条件的解

```
#include <iostream>
#include <vector>
using namespace std;

// 动态规划法求最佳解
void dp_knapsack(vector<int>& weights, vector<int>& values, int W) {
    int n = weights.size();
    // 创建 DP 表
    vector<int> dp(W + 1, 0);

    // 填充 DP 表
    for (int i = 0; i < n; i++) {
        for (int w = W; w >= weights[i]; w--) {
            if (dp[w] < dp[w - weights[i]] + values[i]) {
                dp[w] = dp[w - weights[i]] + values[i];
            }
        }
    }

    // 输出最佳解的价值
    cout << "最佳解价值: " << dp[W] << endl;
}

// 回溯法全局变量
vector<vector<int>> all_solutions;
vector<int> best_items;
vector<int> current;
int best_val;
vector<int> weights_ref;
vector<int> values_ref;
int W_ref;

// 递归回溯函数
```

```
    void dfs(int i, int w, int v) {
        int n = weights_ref.size();
        if (i == n) {
            if (w <= W_ref) {
                all_solutions.push_back(current); //  记录所有解
                if (v > best_val) {
                    best_val = v;            //  更新最佳解
                    best_items = current;
                }
            }
            return;
        }

        //  选择当前物品
        if (w + weights_ref[i] <= W_ref) {
            current.push_back(i + 1);
            dfs(i + 1, w + weights_ref[i], v + values_ref[i]);
            current.pop_back();
        }

        //  不选择当前物品
        dfs(i + 1, w, v);
    }

    //  回溯法求所有解和最佳解
    void backtrack_knapsack(vector<int>& weights, vector<int>& values, int W) {
        //  初始化全局变量
        weights_ref = weights;
        values_ref = values;
        W_ref = W;
        all_solutions.clear();
        best_items.clear();
        current.clear();
        best_val = 0;

        //  开始回溯
        dfs(0, 0, 0);

        //  输出所有解
        cout << "\n 所有解：" << endl;
        for (auto& sol : all_solutions) {
            int w = 0, v = 0;
            cout << "物品: ";
            for (int item : sol) {
                cout << item << " ";
                w += weights[item - 1];
                v += values[item - 1];
            }
            cout << "=>  重量:" << w << "  价值:" << v << endl;
        }

        //  输出最佳解
        cout << "\n 最佳解：";
        for (int item : best_items) cout << item << " ";
        cout << "=>  总重量:";
        int total_w = 0;
        for (int item : best_items) total_w += weights[item - 1];
        cout << total_w << "  总价值:" << best_val << endl;
```

```
}

int main() {
    vector<int> weights = { 5, 3, 2, 1 }; //  物品重量
    vector<int> values = { 4, 4, 3, 1 };    //  物品价值
    int W = 6; //  背包容量

    //  使用动态规划求最佳解
    cout << "===  动态规划法  ===" << endl;
    dp_knapsack(weights, values, W);

    //  使用回溯法求所有解和最佳解
    cout << "\n===  回溯法  ===" << endl;
    backtrack_knapsack(weights, values, W);

    return 0;
}
```

四、实验过程原始记录(数据、图表、计算等)

0/1 背包：

```
=== 动态规划法 ===
最佳解价值: 8

=== 回溯法 ===

所有解:
物品: 1 4 => 重量:6 价值:5
物品: 1 => 重量:5 价值:4
物品: 2 3 4 => 重量:6 价值:8
物品: 2 3 => 重量:5 价值:7
物品: 2 4 => 重量:4 价值:5
物品: 2 => 重量:3 价值:4
物品: 3 4 => 重量:3 价值:4
物品: 3 => 重量:2 价值:3
物品: 4 => 重量:1 价值:1
物品: => 重量:0 价值:0

最佳解: 2 3 4 => 总重量:6 总价值:8
```

求解最大值

```
10
1 1 25 104 1 5 5 12 8 7
1

D:\代码文件\c++\exam1\x64\Debug\ex
```

五、实验结果及分析

结果都已对应显示在原始数据记录中，结果都与预期的分析符合。