**云南大学信息学院《算法设计与分析》课外实验报告**

**实验题目**：0-1 背包问题的不同算法实现与性能分析  
**学生姓名**： 王塏 学号： 20231060264   
**指导教师**：岳昆、段亮、王笳辉  
**实验日期**：2025 年6月24日

**一、实验设置**

**1.1 实验目的**

通过编程实现蛮力法、动态规划法、贪心法和回溯法解决 0-1 背包问题，测试不同输入规模下算法的执行时间和空间占用，对比算法性能与理论时间复杂度，深化对算法设计与分析的理解。

**1.2 实验环境**

* **硬件环境**：CPU 型号: AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz，主存容量: 16GB
* **软件环境**：操作系统:Windows11，编译环境:Dev c++
* 数据生成工具：Python

**二、实验原理**

**2.1 算法思想**

**2.1.1 蛮力法**

基本思想：枚举所有可能的物品组合，选取总价值最大且不超过背包容量的组合作为解。

**2.1.2 动态规划法**

核心思路：将原问题分解为相似的子问题，在求解的过程中通过子问题的解描述并求出原问题的解。动态规划的思想广泛应用于计算机科学和工程领域，在查找有很多重叠子问题的最优解时有效。它将问题重新组合成子问题，为了避免多次解决这些子问题，子问题的结果都逐一被计算并保存，从小规模的子问题直到整个问题都被解决。因此，动态规划对每一子问题只做一次计算，具有较高的效率。

定义状态dp[i][j]表示前i个物品放入容量为j的背包的最大价值，状态转移方程：  
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])（若j ≥ w[i]）。

**2.1.3 贪心法**

贪心策略：采用迭代构造最优解的方式，在每个阶段在一定的标准下，都作出一个当前最优的决策。决策一旦作出，就不可再更改（作出贪心决策的依据称为贪心准则）。贪心算法的一般步骤如下：

（1）根据拟解决问题选取一种贪心准则；

（2）按贪心准侧标准对候选输入进行排序或基于堆来存储候选输入；

（3）依次选择输入量加入部分解中：如果当前输入量的加入不满足约束条件，则不把此输入加到这部分解中。

**2.1.4 回溯法**

搜索策略：回溯法在问题的解空间树中，按深度优先策略从根节点出发搜索解空间树。算法搜索至解空间树的任一节点时，先判断该节点是否包含问题的解。如果不包含，则跳过对以该节点为根的子树的搜索，逐层向其它祖先节点回溯。否则，进入该子树，继续按照深度优先策略搜索。回溯法求问题的所有解时，要回溯到根，且根节点的所有子树都已被搜索遍才结束。回溯法求问题的一个解时只要搜索到问题的一个解就可结束。

**2.2 伪码设计**

**2.2.1 动态规划法伪码**

plaintext

Function DynamicProgramming(weights[], values[], n, capacity):

Create 2D array dp[n+1][capacity+1]

For i from 0 to n:

For j from 0 to capacity:

If i == 0 or j == 0:

dp[i][j] = 0

Else if weights[i-1] ≤ j:

dp[i][j] = max(dp[i-1][j], dp[i-1][j-weights[i-1]] + values[i-1])

Else:

dp[i][j] = dp[i-1][j]

Return dp[n][capacity]

**2.2.2 蛮力法法伪码**

Algorithm BruteForce(n, w[1..n], v[1..n], C)

// Initialize result

best\_value ← 0

best\_weight ← 0

best\_selection ← empty list

max\_mask ← 2^n

// Enumerate all 2^n combinations

for mask ← 0 to max\_mask - 1 do

curr\_value ← 0

curr\_weight ← 0

curr\_selection ← empty list

// Check each item

for i ← 1 to n do

if (mask AND (2^(i-1))) ≠ 0 then

curr\_weight ← curr\_weight + w[i]

curr\_value ← curr\_value + v[i]

curr\_selection.append(i)

// Update best solution if feasible and better

if curr\_weight ≤ C and curr\_value > best\_value then

best\_value ← curr\_value

best\_weight ← curr\_weight

best\_selection ← curr\_selection

return best\_selection, best\_value, best\_weight

**2.2.3 回溯法伪码**

Algorithm Backtracking(n, w[1..n], v[1..n], C)

// Initialize result

best ← {selection: empty list, value: 0, weight: 0}

temp ← empty list

// Recursive backtracking

Procedure Backtrack(i, curr\_value, curr\_weight, temp)

// Update best solution if better

if curr\_value > best.value then

best.selection ← copy(temp)

best.value ← curr\_value

best.weight ← curr\_weight

// Base case: all items considered

if i > n then

return

// Don’t include item i

Backtrack(i + 1, curr\_value, curr\_weight, temp)

// Include item i if feasible

if curr\_weight + w[i] ≤ C then

temp.append(i)

Backtrack(i + 1, curr\_value + v[i], curr\_weight + w[i], temp)

temp.remove(i)

// Start backtracking from first item

Backtrack(1, 0, 0, temp)

return best.selection, best.value, best.weight

**2.2.4 贪心法伪码**

Algorithm Greedy(n, w[1..n], v[1..n], C)

// Initialize result

result ← empty list

total\_value ← 0

total\_weight ← 0

// Create array of (value/weight, index)

items ← empty list

for i ← 1 to n do

items.append((v[i]/w[i], i))

// Sort items by value/weight ratio in descending order

sort(items, descending by ratio)

// Select items greedily

curr\_weight ← 0

for (ratio, i) in items do

if curr\_weight + w[i] ≤ C then

result.append(i)

total\_value ← total\_value + v[i]

total\_weight ← total\_weight + w[i]

curr\_weight ← curr\_weight + w[i]

return result, total\_value, total\_weight

**2.3 实验设计步骤**

1. 生成随机测试数据：物品数量n∈[1000, 320000]，背包容量c∈[10000, 1000000]，重量 1~100，价值 100~1000；
2. 用 C 语言实现四种算法，在程序首尾添加时间戳计算执行时间；
3. 记录每种算法在不同输入规模下的输出结果、执行时间和空间占用；
4. 用 Python 生成性能对比折线图，与理论复杂度对比分析。

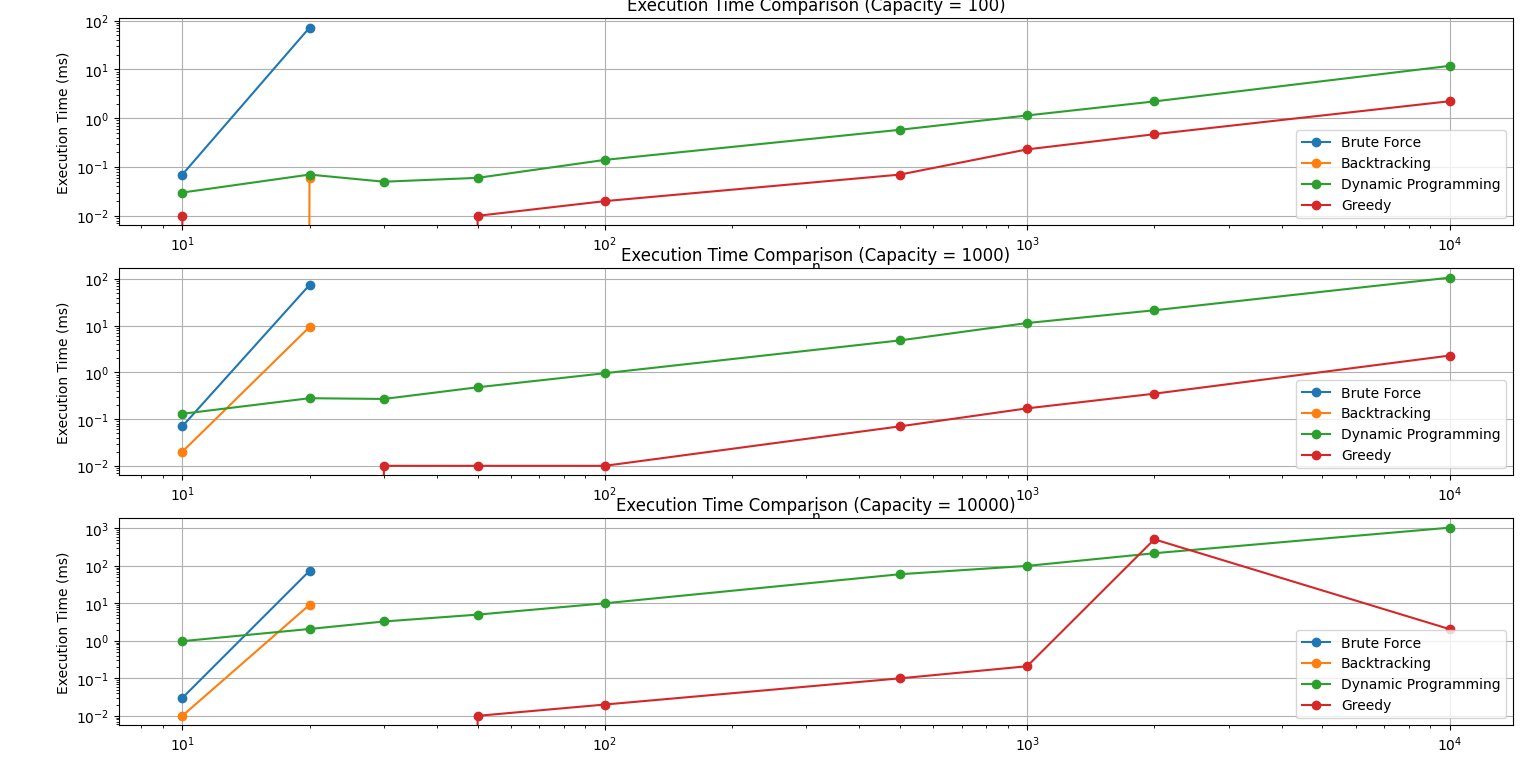
**三、实验数据**

**3.1 1000 个物品时的统计信息**

见文件20231060264-王塏-数据.csv

**四、实验结果**

**4.1 算法执行时间对比**

**图 1.** 

图表 1**不同物品数量下各算法执行时间（ms）**

**4.2 结果分析**

* **蛮力法**：枚举所有2^n种组合，检查每种组合的合法性和价值。时间复杂度O(2^n)，仅适用于小规模（n≤30）。
* **动态规划法**：使用二维DP数组dp[i][w]表示前i个物品在容量w下的最大价值。时间复杂度O(nC)，空间复杂度O(nC)，适合所有规模。当 c=10000 时，n=10000 执行时间约14.48ms，随 n 线性增长；
* **贪心法**：按价值/重量比降序排序，依次选择物品。时间复杂度O(n log n)，非最优解，适合快速近似，排序为主要耗时步骤，执行效率最高；
* **回溯法**：时间复杂度 O (n×2ⁿ)，通过剪枝优化后，深度优先搜索解空间树，剪枝无效分支，仅适用于小规模（n≤30）。n=1000 时已无法在合理时间内完成计算。

**4.3 结论**

1. 动态规划法在中等规模数据下表现最优，适用于背包容量有限的场景,随n和C线性增长，C=1000000时耗时较长。；
2. 贪心法虽非最优解，但随n轻微增长（因排序)效率最高，适合对解精度要求不高的场景；
3. 蛮力法和回溯法仅适用于小规模问题，指数增长，仅n≤30可行,实际应用中需结合剪枝优化。

**五、附录**

**5.1 核心代码片段**

c

// 动态规划法

Result\* dynamic\_programming(Item \*items, int n, int capacity) {

Result \*res = (Result\*)malloc(sizeof(Result));

if (!res) {

perror("malloc failed for res");

exit(EXIT\_FAILURE);

}

res->selected = (int\*)malloc(n \* sizeof(int));

if (!res->selected) {

perror("malloc failed for res->selected");

free(res);

exit(EXIT\_FAILURE);

}

res->count = 0;

res->total\_value = 0;

res->total\_weight = 0;

double \*\*dp = (double\*\*)malloc((n + 1) \* sizeof(double\*));

if (!dp) {

perror("malloc failed for dp");

exit(EXIT\_FAILURE);

}

int \*\*keep = (int\*\*)malloc((n + 1) \* sizeof(int\*));

if (!keep) {

perror("malloc failed for keep");

free(dp);

exit(EXIT\_FAILURE);

}

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

dp[i] = (double\*)calloc(capacity + 1, sizeof(double));

keep[i] = (int\*)calloc(capacity + 1, sizeof(int));

if (!dp[i] || !keep[i]) {

perror("calloc failed");

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

free(dp[j]);

free(keep[j]);

}

free(dp);

free(keep);

free(res->selected);

free(res);

exit(EXIT\_FAILURE);

}

}

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

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

if (items[i-1].weight <= w) {

double include = items[i-1].value + dp[i-1][w - items[i-1].weight];

if (include > dp[i-1][w]) {

dp[i][w] = include;

keep[i][w] = 1;

} else {

dp[i][w] = dp[i-1][w];

}

} else {

dp[i][w] = dp[i-1][w];

}

}

}

int w = capacity;

for (int i = n; i > 0; i--) {

if (keep[i][w]) {

res->selected[res->count++] = items[i-1].id;

res->total\_weight += items[i-1].weight;

res->total\_value += items[i-1].value;

w -= items[i-1].weight;

}

}

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

free(dp[i]);

free(keep[i]);

}

free(dp);

free(keep);

return res;

}

//蛮力法

Result\* brute\_force(Item \*items, int n, int capacity) {

Result \*res = (Result\*)malloc(sizeof(Result));

if (!res) {

perror("malloc failed for res");

exit(EXIT\_FAILURE);

}

res->selected = (int\*)malloc(n \* sizeof(int));

if (!res->selected) {

perror("malloc failed for res->selected");

free(res);

exit(EXIT\_FAILURE);

}

res->count = 0;

res->total\_value = 0;

res->total\_weight = 0;

int max\_mask = 1 << n;

double best\_value = 0;

int best\_weight = 0;

int best\_mask = 0;

for (int mask = 0; mask < max\_mask; mask++) {

double curr\_value = 0;

int curr\_weight = 0;

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

if (mask & (1 << i)) {

curr\_weight += items[i].weight;

curr\_value += items[i].value;

}

}

if (curr\_weight <= capacity && curr\_value > best\_value) {

best\_value = curr\_value;

best\_weight = curr\_weight;

best\_mask = mask;

}

}

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

if (best\_mask & (1 << i)) {

res->selected[res->count++] = items[i].id;

}

}

res->total\_value = best\_value;

res->total\_weight = best\_weight;

return res;

}

// 动态规划法

Result\* dynamic\_programming(Item \*items, int n, int capacity) {

Result \*res = (Result\*)malloc(sizeof(Result));

if (!res) {

perror("malloc failed for res");

exit(EXIT\_FAILURE);

}

res->selected = (int\*)malloc(n \* sizeof(int));

if (!res->selected) {

perror("malloc failed for res->selected");

free(res);

exit(EXIT\_FAILURE);

}

res->count = 0;

res->total\_value = 0;

res->total\_weight = 0;

double \*\*dp = (double\*\*)malloc((n + 1) \* sizeof(double\*));

if (!dp) {

perror("malloc failed for dp");

exit(EXIT\_FAILURE);

}

int \*\*keep = (int\*\*)malloc((n + 1) \* sizeof(int\*));

if (!keep) {

perror("malloc failed for keep");

free(dp);

exit(EXIT\_FAILURE);

}

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

dp[i] = (double\*)calloc(capacity + 1, sizeof(double));

keep[i] = (int\*)calloc(capacity + 1, sizeof(int));

if (!dp[i] || !keep[i]) {

perror("calloc failed");

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

free(dp[j]);

free(keep[j]);

}

free(dp);

free(keep);

free(res->selected);

free(res);

exit(EXIT\_FAILURE);

}

}

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

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

if (items[i-1].weight <= w) {

double include = items[i-1].value + dp[i-1][w - items[i-1].weight];

if (include > dp[i-1][w]) {

dp[i][w] = include;

keep[i][w] = 1;

} else {

dp[i][w] = dp[i-1][w];

}

} else {

dp[i][w] = dp[i-1][w];

}

}

}

int w = capacity;

for (int i = n; i > 0; i--) {

if (keep[i][w]) {

res->selected[res->count++] = items[i-1].id;

res->total\_weight += items[i-1].weight;

res->total\_value += items[i-1].value;

w -= items[i-1].weight;

}

}

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

free(dp[i]);

free(keep[i]);

}

free(dp);

free(keep);

return res;

}

// 回溯法

void backtrack(Item \*items, int n, int capacity, int idx, double curr\_value, int curr\_weight,

int \*temp, int temp\_count, Result \*best) {

if (curr\_value > best->total\_value) {

best->count = temp\_count;

best->total\_value = curr\_value;

best->total\_weight = curr\_weight;

for (int i = 0; i < temp\_count; i++) {

best->selected[i] = temp[i];

}

}

if (idx >= n) return;

// 不选当前物品

backtrack(items, n, capacity, idx + 1, curr\_value, curr\_weight, temp, temp\_count, best);

// 选当前物品（如果可行）

if (curr\_weight + items[idx].weight <= capacity) {

temp[temp\_count] = items[idx].id;

backtrack(items, n, capacity, idx + 1, curr\_value + items[idx].value,

curr\_weight + items[idx].weight, temp, temp\_count + 1, best);

}

}

Result\* backtracking(Item \*items, int n, int capacity) {

Result \*res = (Result\*)malloc(sizeof(Result));

if (!res) {

perror("malloc failed for res");

exit(EXIT\_FAILURE);

}

res->selected = (int\*)malloc(n \* sizeof(int));

if (!res->selected) {

perror("malloc failed for res->selected");

free(res);

exit(EXIT\_FAILURE);

}

res->count = 0;

res->total\_value = 0;

res->total\_weight = 0;

int \*temp = (int\*)malloc(n \* sizeof(int));

if (!temp) {

perror("malloc failed for temp");

free(res->selected);

free(res);

exit(EXIT\_FAILURE);

}

backtrack(items, n, capacity, 0, 0, 0, temp, 0, res);

free(temp);

return res;

}

**5.2 数据附件与项目地址**

* 实验数据 csv 文件：20231060264-王塏-数据.csv（见压缩包附件）
* 代码文件：20231060264-王塏-代码.c（见压缩包附件）
* Github 项目地址：\_\_\_\_\_\_\_\_\_\_（需包含代码、数据及报告）。