**《算法设计与分析》课外实践作业**

1. **实验设置**
2. 实验目的

通过编程实现经典算法并通过程序执行开销对算法性能进行分析，并对解决同一问题的不同算法进行性能的对比分析，深入理解时间复杂度渐进性态和和增长率的概念，增强本专业学生算法思维赋能的编程能力提升，培养学生对给定问题选择不通求解方案的能力。

使用C语言编程实现0-1背包问题的不同求解算法，并测试不同输入规模下程序的执行时间和占用空间，深入理解蛮力法、动态规划法、贪心法和回溯法的基本思想，并与理论分析的结论进行对比，强化对不同算法思想及复杂度的理解、以及各类经典算法设计与分析的技巧，为复杂工程问题的求解奠定基础。

1. 实验环境

·硬件：

CPU：13th Gen Intel(R) Core(TM) i9-13980HX

主存：16GB

硬盘：NVMe HFS001TEJ9X101N

·软件：

编译器：VSCode + GCC8.1.0

操作系统：Windows 11

1. **实验原理**
2. 蛮力法

·算法思想：

枚举所有可能的物品组合，计算每个组合的总重量和总价值，在满足重量约束的条件下找出最大价值解。适用于小规模问题（n ≤ 20）。

·伪码：

Brute(n, C, items):

   max\_value = 0

   best\_selection = 0

   total\_combinations = 2^n

   for i = 0 to total\_combinations-1:

        current\_weight = 0

        current\_value = 0

        for j = 0 to n-1:

            if i的第j位为1:

                current\_weight += items[j].weight

                current\_value += items[j].value

        if current\_weight ≤ C and current\_value > max\_value:

            max\_value = current\_value

            best\_selection = i

   记录best\_selection对应的物品选择方案

   return max\_value

·设计步骤：

1. 输入物品数n和背包容量C
2. 若n > 20，跳过（避免指数爆炸）
3. 生成0到2ⁿ-1的所有二进制组合
4. 对每个组合：

·计算总重量和总价值

·若满足重量约束且价值更大，更新最优解

1. 输出最大价值和选中物品

时间复杂度：O(nlogn)

1. 动态规划法

·算法思想：

使用dp[j]表示容量为j时的最大价值。逆序更新一维DP数组，避免重复选择。空间优化后复杂度为O(n×C)。

·伪码：

DP(n, C, items):

   dp[0..C] = 0

   for i = 0 to n-1:

        for j = C down to items[i].weight:

            dp[j] = max(dp[j], dp[j - items[i].weight] + items[i].value)

   j = C

   for i = n-1 down to 0:

        if dp[j] == dp[j - items[i].weight] + items[i].value:

            选中items[i]

            j -= items[i].weight

   return dp[C]

·设计步骤：

1. 检查内存需求（避免超过2GB）
2. 初始化一维DP数组（大小为C+1）
3. 双层循环：

·外层遍历物品

·内层逆序更新DP数组

1. 反向回溯确定选中物品
2. 输出最大价值和方案

时间复杂度：O(n\*C) (C为背包容量)

1. 贪心法

·算法思想：

按单位价值（价值/重量）降序排序，优先选择高单位价值物品。非精确解，但时间复杂度仅O(n log n)。

·伪码：

Greedy(n, C, items):

   按value/weight降序排序items

   current\_weight = 0

   total\_value = 0

   for i = 0 to n-1:

        if current\_weight + items[i].weight ≤ C:

            选中items[i]

            current\_weight += items[i].weight

            total\_value += items[i].value

   return total\_value

·设计步骤：

1. 计算物品单位价值
2. 按单位价值降序排序
3. 顺序扫描物品：能放入则选择并更新重量和价值
4. 输出总价值和选中物品

时间复杂度：O(n2)

1. 回溯法

·算法思想：

使用优先队列（最大堆）存储活节点。节点上界=当前价值+剩余容量贪心价值。优先扩展上界大的节点，通过剪枝加速搜索。

·伪码：

BranchAndBound(n, C, items):

   按单位价值降序排序

   pq = 优先队列(按bound降序)

   添加根节点(level=-1, value=0, weight=0)

   max\_value = 0

   while pq非空:

        u = pq.pop()

        if u.bound ≤ max\_value:

            continue  // 剪枝

        扩展u的子节点v（选/不选下个物品）：

           if 选择后重量≤C:

               更新max\_value和最优解

           if v.bound > max\_value:

               pq.push(v)

   return max\_value

·设计步骤：

1. 按单位价值降序排序
2. 初始化优先队列（最大堆）
3. 循环处理队列：

·弹出上界最大的节点

·剪枝：上界≤当前最优则跳过

·扩展子节点（选/不选）

·计算新节点上界（剩余容量贪心值）

·满足约束且上界够大则入队

1. 输出最优解和选中物品

时间复杂度：O(2n)

1. **实验数据**

表 1容量为1000的0-1背包物品统计信息（节选）

|  |  |  |
| --- | --- | --- |
| **物品编号** | **物品重量** | **物品价值** |
| 1 | 9 | 388.88 |
| 2 | 74 | 226.13 |
| 3 | 31 | 297.81 |
| 4 | 73 | 118.55 |
| 5 | 28 | 119.44 |
| 6 | 4 | 269.29 |
| 7 | 17 | 233.6 |
| 8 | 48 | 203.33 |
| 9 | 84 | 318.09 |
| 10 | 20 | 157.08 |
| …… | …… | …… |
| 1000 | 46 | 258.64 |

1. **实验结果**

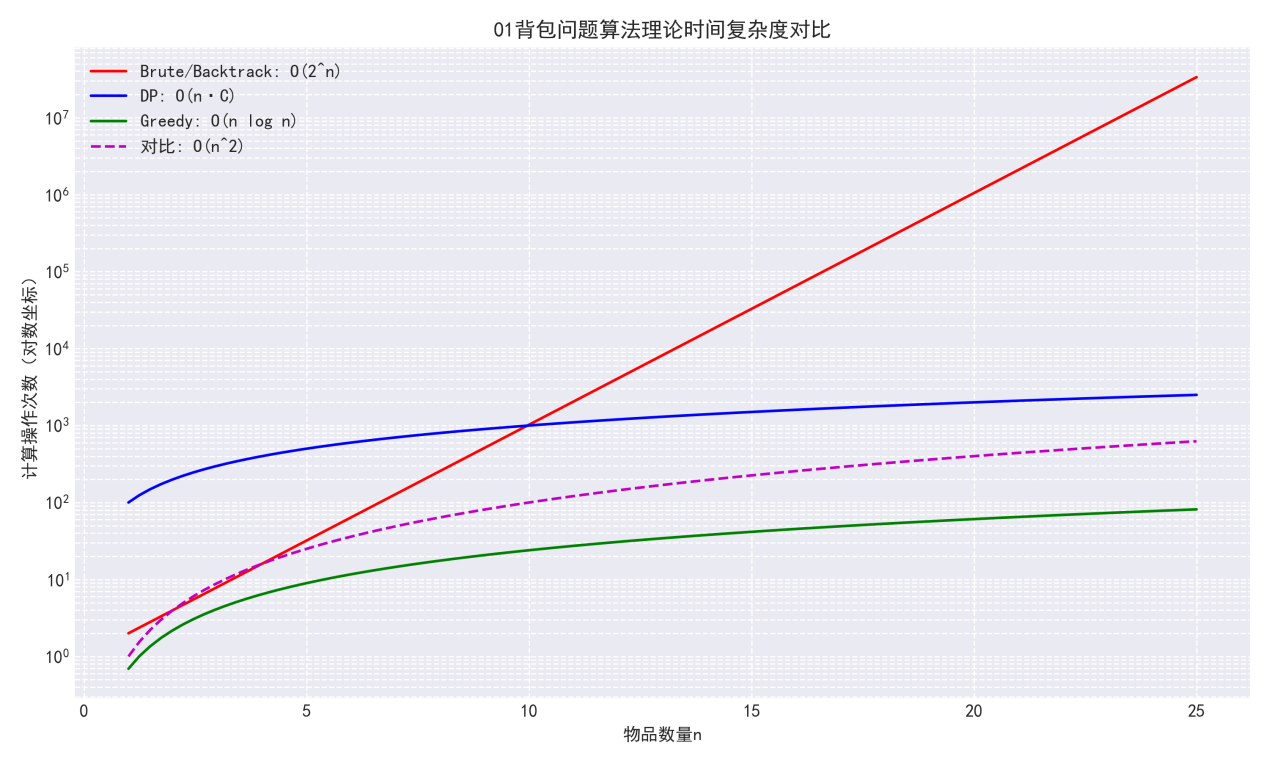


图 10-1背包问题四种算法理论时间复杂度对比

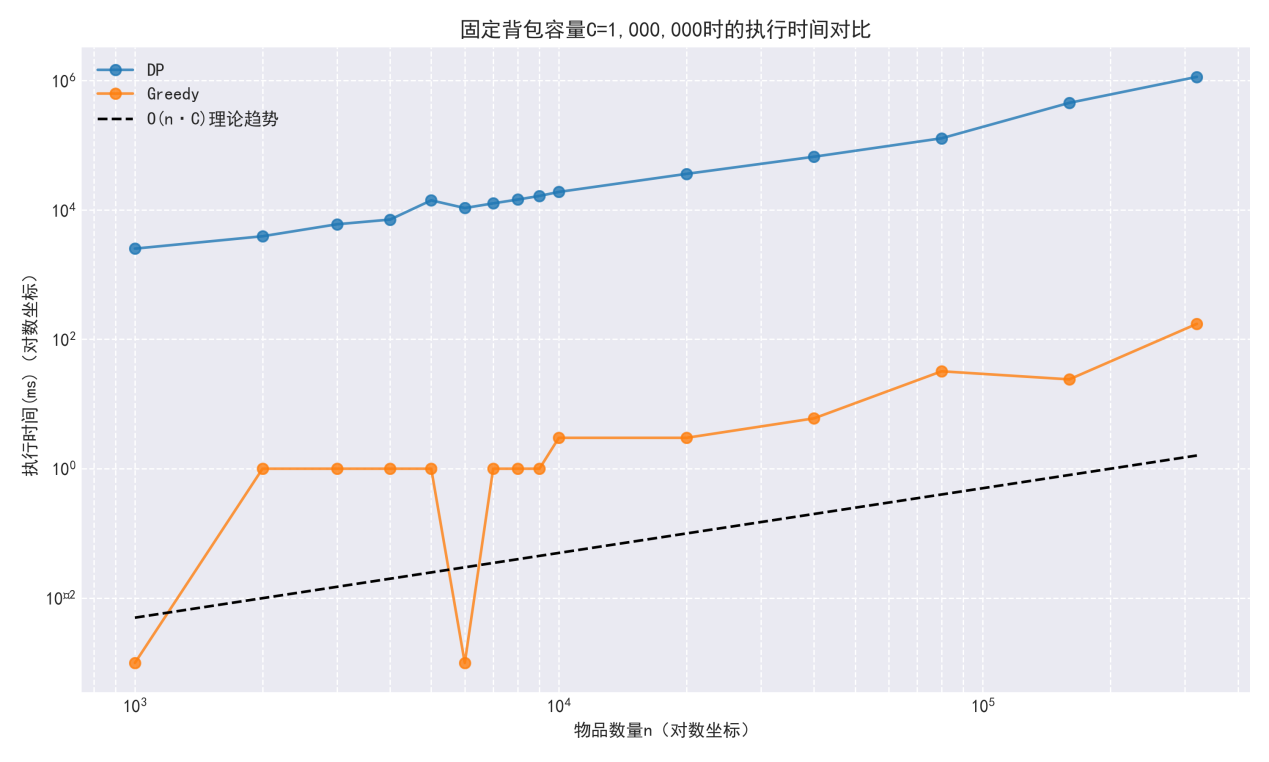


图 2背包容量C=1000000时执行时间对比

根据上述两个图像可得

1. 贪心法和动态规划法基本符合理论时间复杂度曲线
2. 贪心法的一些离群点来自于随机数据中的偶然特殊值

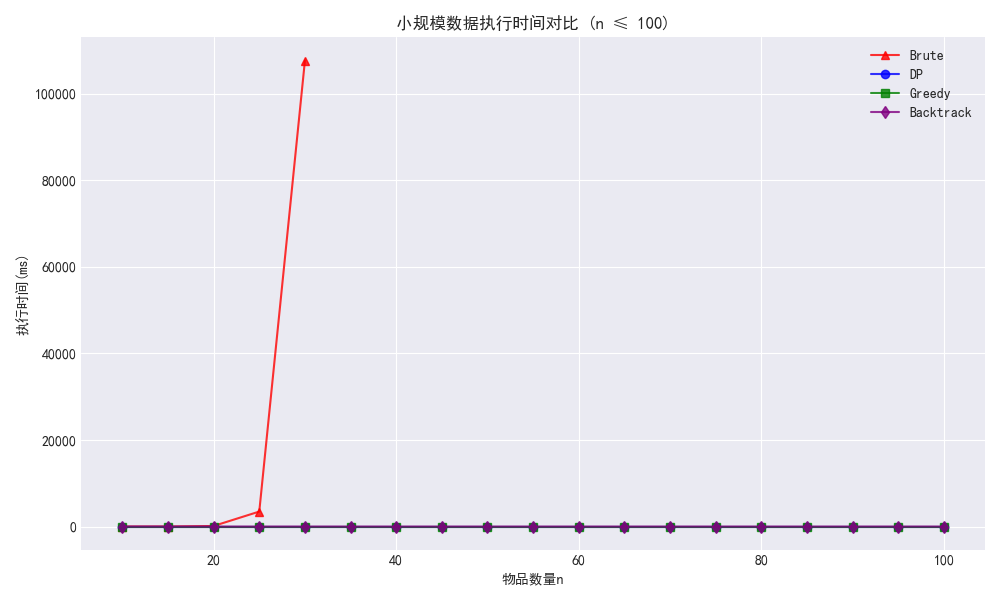


图 3小规模数据执行时间对比

根据上述两个图像可得

1. 在小规模数据量时，数据规模对四种算法速率影响不大
2. 蛮力法在25-30规模时就发生了跃变，符合指数级时间复杂度

·算法性能分析与结论

贪心法：

优点：执行时间最短，适用于大规模问题。

缺点：无法保证最优解。

适用场景：对执行时间敏感且对解的精确度要求不高的场景。

动态规划法：

优点：可以求得最优解，适用于中等规模问题。

缺点：时间和空间复杂度高，背包容量较大时性能会大大下降。

适用场景：背包容量较小且物品数量适中的场景。

回溯法：

优点：可以求得最优解，可求出多个最优解。

缺点：时间复杂度仍较高，仅适用于中等规模问题。

适用场景：多个最优解的场景。

蛮力法：

优点：实现简单，能求得最优解。

缺点：时间复杂度极高，仅适用于极小规模问题。

适用场景：物品数量极少的场景。

**附录：**

1. **核心代码片段**

***// 蛮力法***

double brute(int n, int C, Item \*items, int \*chosen, int \*cnt) {

    if (n > 30) {

        printf("蛮力法跳过\n", n);

        return -1;

    }

    double max\_v = 0.0;

    int final\_selection = 0;

    long long total\_combinations = 1LL << n;

    for (long long i = 0; i < total\_combinations; i++) {

        int current\_weight = 0;

        double current\_value = 0.0;

        for (int j = 0; j < n; j++) {

            if (i & (1LL << j)) {

                current\_weight += items[j].weight;

                current\_value += items[j].value;

            }

        }

*// 更新最优解*

        if (current\_weight <= C && current\_value > max\_v) {

            max\_v = current\_value;

            final\_selection = i;

        }

    }

*// 记录选中的物品*

    \*cnt = 0;

    memset(chosen, 0, n \* sizeof(int));

    for (int j = 0; j < n; j++) {

        if (final\_selection & (1LL << j)) {

            chosen[j] = 1;

            (\*cnt)++;

        }

    }

    return max\_v;

}

***// 动态规划法***

double dynamicProgramming(int n, int C, Item \*items, int \*chosen, int \*cnt) {

*// 检查内存需求 (C+1个long long元素)*

    size\_t mem = (C + 1) \* sizeof(long long);

    if (mem > 2000000000) { *// 2GB内存限制*

        printf("动规内存分配失败！\n");

        return -1;

    }

*// 转long long减少计算时间*

    long long \*dp = calloc(C + 1, sizeof(long long));

    if (!dp) {

        printf("动规内存分配失败！\n");

        return -1;

    }

*// 动态规划按行填表*

    for (int i = 0; i < n; i++) {

        int w = items[i].weight;

        long long v = (long long)(items[i].value \* 100 + 0.5);

        for (int j = C; j >= w; j--) {

            if (dp[j - w] + v > dp[j]) {

                dp[j] = dp[j - w] + v;

            }

        }

    }

*// 记录选中的物品*

    \*cnt = 0;

    memset(chosen, 0, n \* sizeof(int));

    for (int j = C; j >= 0; j--) {

        if (dp[j] > 0) {

            int i = 0;

            while (i < n && items[i].weight > j) i++;

            if (i < n) {

                chosen[i] = 1;

                (\*cnt)++;

            }

        }

    }

*// 计算最大价值*

    double max\_v = dp[C] / 100.0;

    free(dp);

    \*cnt = 0; *// 动态规划不记录具体物品选择*

    return max\_v;

}

***// 贪心法***

double greedy(int n, int C, Item \*items, int \*chosen, int \*cnt) {

*// 按单位价值排序*

    Item \*sorts = malloc(n \* sizeof(Item));

    memcpy(sorts, items, n \* sizeof(Item));

    qsort(sorts, n, sizeof(Item), compare);

    double total\_v = 0.0;

    int curr\_w = 0;

    \*cnt = 0;

    memset(chosen, 0, n \* sizeof(int));

*// 遍历物品，贪心选择*

    for (int i = 0; i < n; i++) {

        if (curr\_w + sorts[i].weight <= C) {

            curr\_w += sorts[i].weight;

            total\_v += sorts[i].value;

            chosen[sorts[i].id - 1] = 1;

            (\*cnt)++;

        }

    }

    free(sorts);

    return total\_v;

}

***// 回溯法***

double backtrack(int n, int C, Item \*items, int \*chosen, int \*cnt) {

    if (n > 100) {

        printf("回溯法跳过");

        return -1;

    }

*// 按单位价值降序排序*

    Item \*sorts = malloc(n \* sizeof(Item));

    memcpy(sorts, items, n \* sizeof(Item));

    qsort(sorts, n, sizeof(Item), compare);

*// 初始化优先队列*

    PriorityQueue \*pq = create\_queue(1000000);

    Node tmp\_n, curr\_n;

    double max\_v = 0.0;

    int \*best\_chosen = calloc(n, sizeof(int));

    int \*curr\_chosen = calloc(n, sizeof(int));

*// 初始化根节点*

    curr\_n.level = -1;

    curr\_n.weight = 0;

    curr\_n.value = 0;

    curr\_n.bound = bound(curr\_n, n, C, sorts);

    push(pq, curr\_n);

*// 回溯搜索*

    while (pq->size > 0) {

        tmp\_n = pop(pq);

        if (tmp\_n.bound > max\_v) {

*// 选择下一个物品*

            curr\_n.level = tmp\_n.level + 1;

            if (curr\_n.level >= n) continue;

*// 标记选择*

            curr\_n.weight = tmp\_n.weight + sorts[curr\_n.level].weight;

            curr\_n.value = tmp\_n.value + sorts[curr\_n.level].value;

            curr\_chosen[curr\_n.level] = 1;

            if (curr\_n.weight <= C && curr\_n.value > max\_v) {

                max\_v = curr\_n.value;

                memcpy(best\_chosen, curr\_chosen, n \* sizeof(int));

            }

            curr\_n.bound = bound(curr\_n, n, C, sorts);

            if (curr\_n.bound > max\_v && curr\_n.level < n - 1) {

                push(pq, curr\_n);

            }

*// 不选下一个物品*

*// 标记不选择*

            curr\_n.weight = tmp\_n.weight;

            curr\_n.value = tmp\_n.value;

            curr\_chosen[curr\_n.level] = 0;

            curr\_n.bound = bound(curr\_n, n, C, sorts);

            if (curr\_n.bound > max\_v && curr\_n.level < n - 1) {

                push(pq, curr\_n);

            }

        }

    }

*// 记录选中的物品*

    \*cnt = 0;

    memset(chosen, 0, n \* sizeof(int));

    for (int i = 0; i < n; i++) {

        if (best\_chosen[i]) {

            chosen[sorts[i].id - 1] = 1;

            (\*cnt)++;

        }

    }

    free(curr\_chosen);

    free(sorts);

    free\_queue(pq);

    free(best\_chosen);

    return max\_v;

}

1. **Github项目地址**

https://github.com/ClumsySun/Algorithm-Design-and-Analysis-Practical-Assignment.git