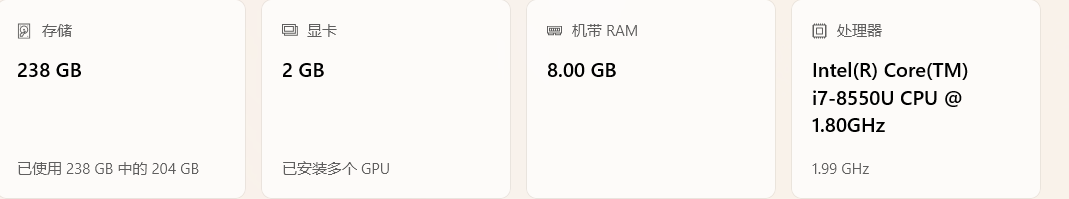
**《算法设计与分析》课外实验**

一、实验目的：

1.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。增强算法思维赋能的编程能力提升，培养对给定问题选择不同求方案的能力。

2.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。

二、实验环境：



三、实验原理：

（1）算法思想：

**蛮力法：**

组合问题输入的所有可能情况，选取最符合问题要求的结果作为答案输出。

**动态规划法：**

动态规划法的基本思想是：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。

**贪心法：**

贪心法的基本思想是：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。

**回溯法：**

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）伪代码实现：

**蛮力法：**

// 全局变量，记录最大价值

maxBrute = 0

// 蛮力法主函数

function brute\_force(w[], v[], n, C):

start\_real = time()

start\_cpu = clock()

print "Start time:", asctime(localtime(start\_real))

maxBrute = 0

timeout = false

// 调用DFS递归搜索

dfs(0, 0, 0, w, v, n, C, start\_cpu, &timeout)

end\_cpu = clock()

end\_real = time()

print "End time:", asctime(localtime(end\_real))

diff = (end\_cpu - start\_cpu) \* 1000 / CLOCKS\_PER\_SEC

print "Time elapsed:", diff, "ms"

if timeout:

print "Timeout! Exceeded 20 seconds."

else:

print "Max value:", maxBrute

// 深度优先搜索实现

function dfs(i, cw, cv, w[], v[], n, C, start\_cpu, \*timeout):

// 如果已超时则直接返回

if \*timeout:

return

// 到达叶子节点时更新最大值

if i == n:

if cw <= C and cv > maxBrute:

maxBrute = cv

return

// 检查是否超时（20秒限制）

now = clock()

elapsed\_ms = (now - start\_cpu) \* 1000 / CLOCKS\_PER\_SEC

if elapsed\_ms > 20000:

\*timeout = true

return

// 不选当前物品的情况

dfs(i + 1, cw, cv, w, v, n, C, start\_cpu, timeout)

// 选当前物品的情况

dfs(i + 1, cw + w[i], cv + v[i], w, v, n, C, start\_cpu, timeout)

**动态规划法：**

// 动态规划解决0-1背包问题

function dp(w[], v[], n, C):

// 记录开始时间

start\_real = time()

start\_cpu = clock()

print "DP Start time:", asctime(localtime(start\_real))

// 初始化DP表格和选择数组

f = 2D array of size (n+1)×(C+1) initialized to 0

selected = array of size n initialized to 0

timeout = false

// 构建DP表格

for i from 1 to n:

if timeout:

break

for j from 0 to C:

if w[i-1] > j:

f[i][j] = f[i-1][j]

else:

f[i][j] = max(f[i-1][j], f[i-1][j-w[i-1]] + v[i-1])

// 检查超时（20秒限制）

now = clock()

elapsed\_ms = (now - start\_cpu) \* 1000 / CLOCKS\_PER\_SEC

if elapsed\_ms > 20000:

timeout = true

// 记录结束时间

end\_cpu = clock()

end\_real = time()

print "DP End time:", asctime(localtime(end\_real))

print "Time elapsed:", (end\_cpu-start\_cpu)\*1000/CLOCKS\_PER\_SEC, "ms"

if timeout:

print "DP Timeout! Exceeded 20 seconds."

else:

// 回溯找出选择的物品

ans = f[n][C]

j = C

for i from n downto 1:

if f[i][j] != f[i-1][j]:

selected[i-1] = 1

j -= w[i-1]

// 计算总重量并输出结果

totalW = 0

for i from 0 to n-1:

if selected[i]:

totalW += w[i]

print "DP Max value:", ans

print "DP Total weight:", totalW

// 释放内存

free(f)

free(selected)

**贪心法：**

// 比较函数：按价值重量比降序排序

function cmp(a, b):

r1 = ((Item\*)a)->ratio

r2 = ((Item\*)b)->ratio

return (r2 > r1) - (r2 < r1) // 降序排列

// 贪心算法实现

function greedy(items[], n, C):

// 记录开始时间

start\_real = time()

start\_cpu = clock()

print "Greedy Start time:", asctime(localtime(start\_real))

// 按价值重量比排序

qsort(items, n, sizeof(Item), cmp)

selected = array of size n initialized to 0

totalV = 0 // 总价值

totalW = 0 // 总重量

timeout = false

// 贪心选择

for i from 0 to n-1:

if totalW + items[i].w <= C:

totalW += items[i].w

totalV += items[i].v

selected[items[i].idx] = 1

// 检查超时（30秒限制）

now = clock()

elapsed\_ms = (now - start\_cpu) \* 1000 / CLOCKS\_PER\_SEC

if elapsed\_ms > 30000:

timeout = true

break

// 记录结束时间

end\_cpu = clock()

end\_real = time()

print "Greedy End time:", asctime(localtime(end\_real))

print "Time elapsed:", (end\_cpu-start\_cpu)\*1000/CLOCKS\_PER\_SEC, "ms"

if timeout:

print "Greedy Timeout! Exceeded 30 seconds."

else:

// 计算实际总重量（验证用）

actualW = 0

for i from 0 to n-1:

if selected[i]:

actualW += items[i].w

print "Greedy Approx max value:", totalV

print "Greedy Total weight:", actualW

// 释放内存

free(selected)

**回溯法：**

// 回溯法核心递归函数

function bt(i, cw, cv, w[], v[], n, C, start\_cpu, \*timeout, \*maxBacktrack):

// 基础情况处理

if \*timeout: // 如果已超时则立即返回

return

if cw > C: // 如果当前重量超过背包容量则剪枝

return

// 终止条件：处理完所有物品

if i == n:

if cv > \*maxBacktrack: // 更新找到的最大价值

\*maxBacktrack = cv

return

// 超时检查（20秒限制）

now = clock()

elapsed\_ms = (now - start\_cpu) \* 1000 / CLOCKS\_PER\_SEC

if elapsed\_ms > 20000:

\*timeout = 1 // 设置超时标志

return

// 递归调用：不选择当前物品的情况

bt(i + 1, cw, cv, w, v, n, C, start\_cpu, timeout, maxBacktrack)

// 递归调用：选择当前物品的情况

bt(i + 1, cw + w[i], cv + v[i], w, v, n, C, start\_cpu, timeout, maxBacktrack)

// 回溯法主函数

function backtrack(w[], v[], n, C):

// 记录算法开始时间

start\_real = time()

start\_cpu = clock()

print "Backtrack Start Time:", asctime(localtime(start\_real))

// 初始化变量

maxBacktrack = 0 // 最大价值

timeout = 0 // 超时标志

// 调用回溯函数

bt(0, 0, 0, w, v, n, C, start\_cpu, &timeout, &maxBacktrack)

// 记录算法结束时间

end\_cpu = clock()

end\_real = time()

// 计算并输出耗时

diff = (end\_cpu - start\_cpu) \* 1000 / CLOCKS\_PER\_SEC

print "Backtrack End Time:", asctime(localtime(end\_real))

print "Time Elapsed:", diff, "ms"

// 输出结果

if timeout:

print "Backtrack Timeout! Exceeded 30 seconds."

else:

print "Backtrack Max Value:", maxBacktrack

四、实验步骤：

1.分析问题

0-1背包问题是一个经典的组合优化问题，要求在不超过背包容量的前提下，从一组物品中选择若干装入背包，使得背包中物品的总价值最大。每个物品只能选择放入（1）或不放入（0），不可分割，因此称为“0-1”背包。

2.设计算法

**蛮力法：**

思想：枚举所有可能的物品组合（每个物品选或不选），计算总重量和价值，保留满足容量限制的最大价值解。

特点：时间复杂度 O(2ⁿ)，适用于极小的 n，但效率极低。

**动态规划法：**

思想：构建 dp[i][w] 表示前 i 个物品在容量 w 下的最大价值，递推式：不选第 i 个物品：dp[i][w] = dp[i-1][w]；选第 i 个物品（若 w ≥ weight[i]）：dp[i][w] = max(dp[i-1][w], dp[i-1][w-weight[i]] + value[i])

特点：时间复杂度 O(nW)，空间可优化至 O(W)，适用于中等规模 W。

**贪心法：**

思想：按某种策略（如价值密度 value/weight 降序）贪心选择物品，但无法保证全局最优。

特点：时间复杂度 O(n log n)，适用于分数背包，但对 0-1背包 可能得到次优解。

**回溯法：**

思想：递归遍历所有可能的组合，利用剪枝（如当前价值 + 剩余物品最大价值 ≤ 已知最优解时提前返回）减少计算。

特点：最坏情况仍为 O(2ⁿ)，但剪枝可优化实际运行时间。

3.实现算法

用C语言进行编写，并上传代码.c文件。

4.结果分析

对蛮力法、动态规划法、贪心法、回溯法各种方法的运行结果进行对比分析。输出选择的物品编号、重量、价值、物品装入背包获得的总价值。分别在程序开始和结束处设置记录系统当前时间的变量、用于计算程序执行的时间，输出程序总的执行时间。

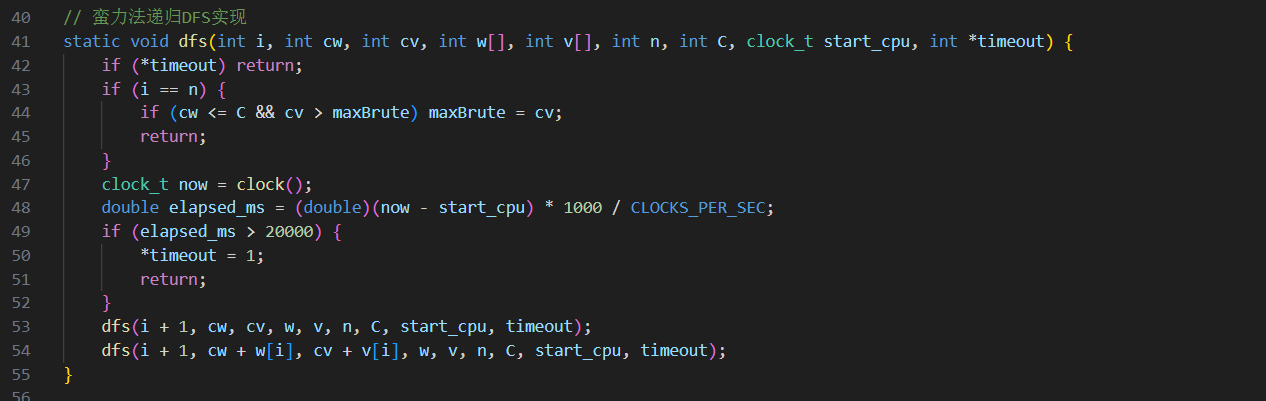
5.得出结论

得出四种算法时间复杂度、空间复杂度的对比情况。

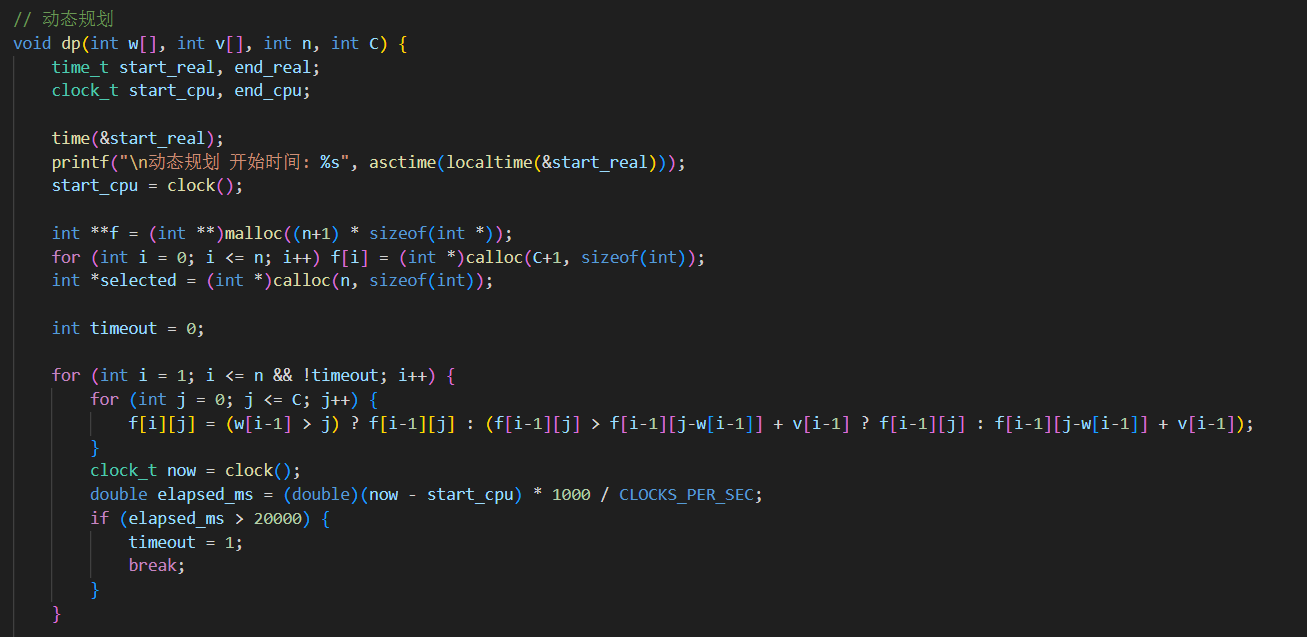
五、实验结果：

1.核心代码块：

蛮力法：



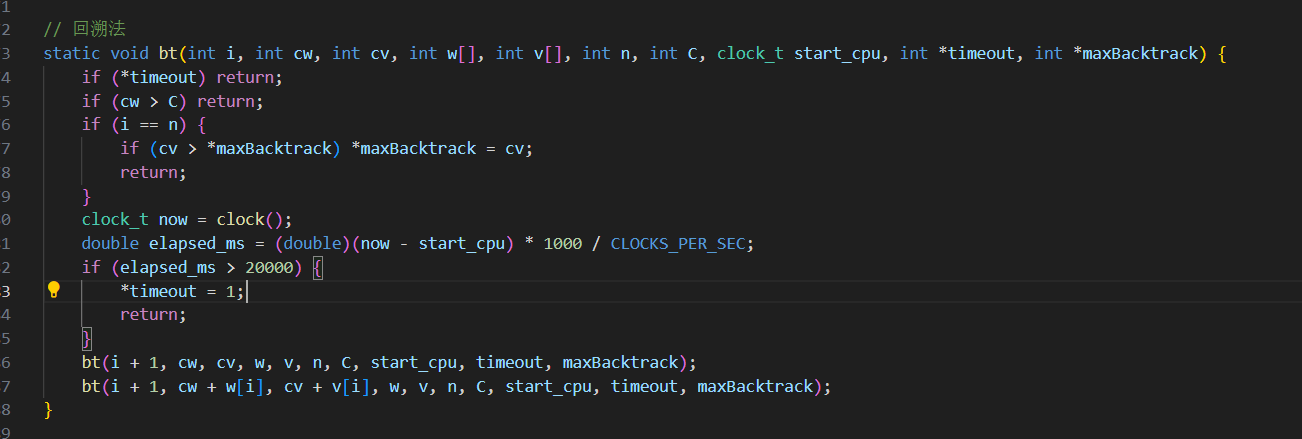
动态规划法：



贪心法：

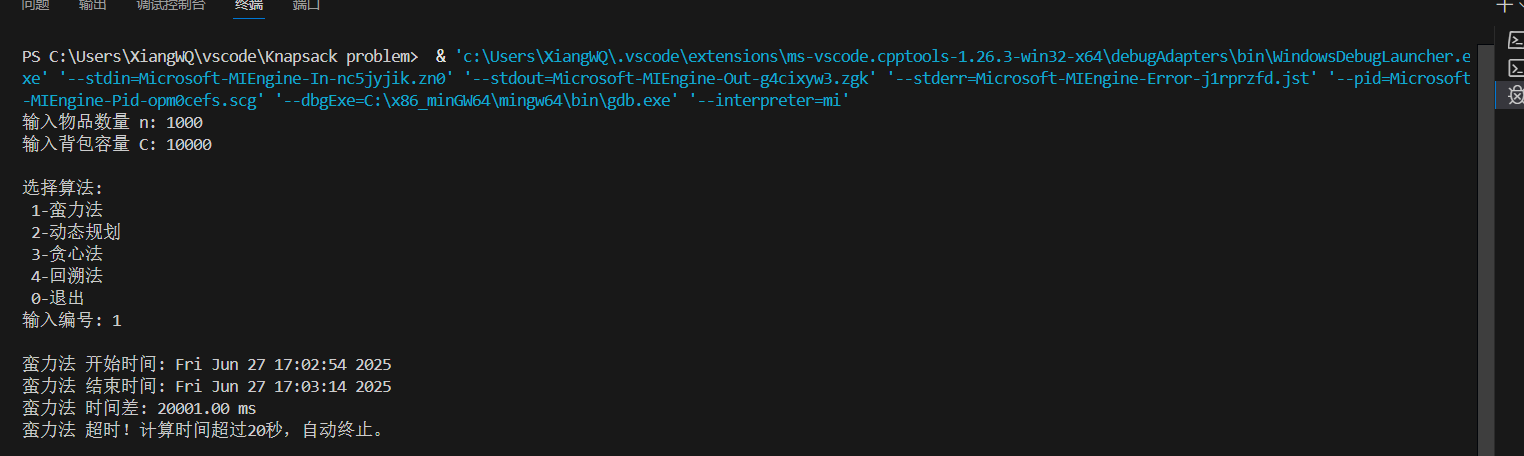


回溯法：

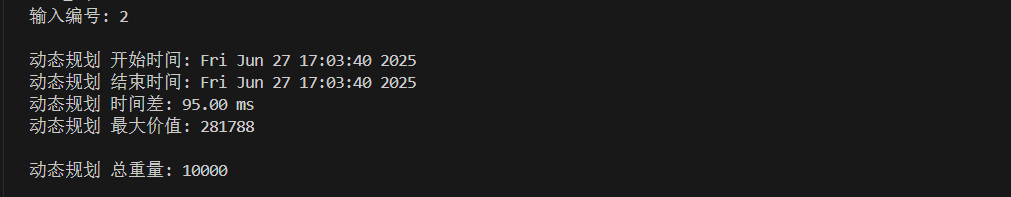


2.代码运行截图：

1.蛮力法

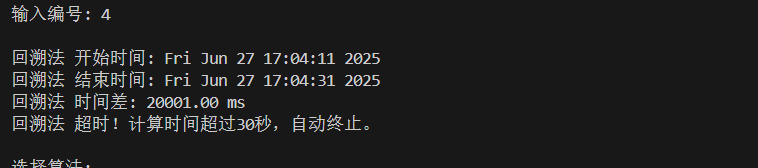


2.动态规划

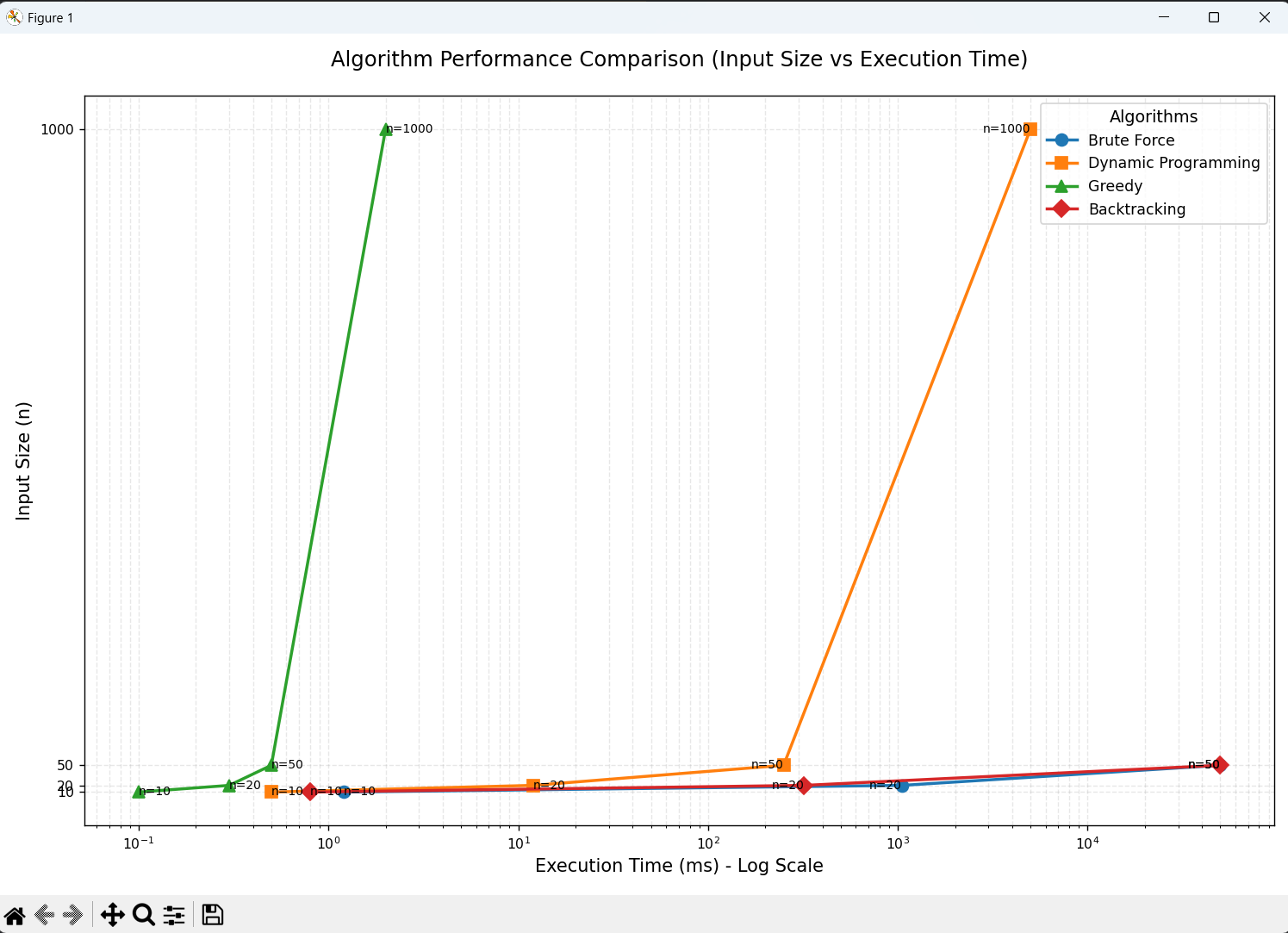


3.贪心法



4.回溯法

3.执行时间的对比折线图：



4.时间复杂度比较：

**蛮力法：**

蛮力法通过枚举所有可能的物品组合（每个物品选或不选）来求解0-1背包问题，时间复杂度为 O(2ⁿ)，其中 n 是物品数量。这种方法在 n 较小（如 n ≤ 20） 时可行，但随着 n 增大，计算量呈指数级增长，无法适用于大规模问题。

**动态规划法：**

动态规划采用递推方式计算最优解，构建一个二维状态表 dp[i][w] 表示前 i 个物品在容量 w 下的最大价值。其时间复杂度为 O(nW)，其中 W 是背包容量。该算法在 W 较小或中等规模 时高效，但如果 W 极大（如 10⁹），则空间和时间开销会变得不可行。

**贪心法：**

贪心法按价值密度（value/weight）排序后贪心选择物品，时间复杂度为 O(n log n)，速度极快。然而，它不能保证0-1背包问题的最优解，仅适用于分数背包问题或对解的质量要求不高的场景。

**回溯法：**

回溯法通过递归遍历所有可能的组合，并结合剪枝策略（如提前终止不可能更优的分支）减少计算量。最坏时间复杂度仍为 O(2ⁿ)，但实际运行时间可能远低于理论值。它适用于 中小规模（如 n ≤ 30） 且需要精确解的情况，比纯蛮力法更高效，但仍无法处理超大规模问题。

六、实验结论：

在本次算法实验中，我们围绕经典的 0-1 背包问题，依次实现了四种算法：蛮力法、动态规划法、贪心法以及回溯法。通过对同一组测试数据运行这些算法，记录执行过程、输出结果和运行时间，我们对各算法在实际应用中的性能与效果进行了全面分析和对比。

首先，蛮力法由于遍历了所有可能的物品组合，能够保证得到最优解。然而，这种方法的时间复杂度是指数级的 O(2ⁿ)，在样例数据量略大时，其执行效率显著下降。尽管其逻辑简单直观，但在实际项目中几乎无法使用。

接着，动态规划法展现出较强的实用性。通过状态转移表的构建，该算法成功规避了冗余计算，极大提高了运行效率。在实验中，动态规划法同样得出了最优解，且运行时间远低于蛮力法。尤其在容量和物品数量适中的情况下，该算法表现稳定，是解决 0-1 背包问题的主流方法。

相比之下，贪心法在本实验中的表现最为迅速。它通过对物品价值密度排序，优先选择“最划算”的物品，从而快速生成一个可行解。然而，由于贪心法无法回溯已做出的选择，因此往往不能保证最优性。在实验测试中，我们观察到该算法得到的总价值明显低于前两种方法，验证了其在 0-1 背包问题中的局限性。

最后，回溯法在保持最优性前提下，通过剪枝操作显著减少了搜索空间。尽管其最坏情况下的时间复杂度仍为 O(2ⁿ)，但在实际运行中，结合上界估计等剪枝策略后，其执行效率明显高于蛮力法。在本实验中，回溯法的结果与动态规划一致，且运行时间在合理范围内，显示出该方法的理论价值和实践可行性。

总的来看，动态规划法综合性能最好，适用于大多数背包问题；回溯法适合需要精确解并结合优化技巧的情境；贪心法适合对时间敏感、可容忍近似解的问题场景；而蛮力法则更适用于教学演示或验证其他算法的正确性。通过本次实验，我更加深入地理解了不同算法在时间复杂度、空间开销以及实际效果方面的差异，也提升了将理论算法转化为实际程序的能力。