背包问题求解算法报告

姓名：孙智诚　张文韬
学号：2451503　2451500

2025 年 12 月 27 日

1　问题描述

给定 n 个物品，每个物品有体积 W_i $(i = 1, 2, \ldots, n)$，以及一个容量为 T 的背包。问题的目标是找出能够恰好装满背包的物品组合方案。

1.1　问题分析

这是一个经典的组合优化问题，具有以下特点：

- 每个物品只能被选择一次
- 选择的顺序不影响最终结果
- 只关心解的存在性，不要求最优解法

从算法设计角度分析，该问题可以采用多种求解策略：

回溯法　遍历 W 数组，当剩余容量无法容纳任何物品时及时回溯，直到寻找出能恰好装满 T 的数字组合。

可达集合法　构建可达集合 a，用以容纳 W 中任意元素之间通过加法运算能够达到的结果，最后将该集合与背包容量 T 比对，若 $T \in a$，则说明容量为 T 的背包可以被恰好装满。

本报告中，主要采用回溯法实现，因为该方法能够记录所有的解法，且时间、空间复杂度较法二均更低，适合展示算法思想和搜索过程，而法二则只做为该问题解决方法多样性的例证。

2 算法设计

2.1 算法思想

本算法采用回溯法求解背包问题的所有可行解。用显式栈 S 记录当前已选择的物品索引序列，用 $remain\_T$ 表示背包剩余容量。搜索过程等价于在一棵解空间树上进行深度优先遍历：每向下一层表示尝试将某个物品加入当前组合；当加入后仍满足 $remain\_T \geq 0$ 时继续扩展；若出现 $remain\_T < 0$（容量超出）或已无法继续选择物品，则当前分支不可行，需要回溯到上一层。

回溯时通过弹出栈顶元素撤销最近一次选择，并恢复相应的剩余容量，然后从被撤销物品的下一个编号继续尝试，从而避免重复枚举相同组合；当 $remain\_T = 0$ 时得到一组恰好装满背包的解并输出。最终当栈为空且不存在可继续扩展的分支时，说明所有可能组合均已遍历完成，算法结束。

2.2 数据结构

本算法使用的核心数据结构与状态变量如下：

- **物品体积表** W：用一维数组存储各物品体积。实现中采用 1 开始的下标（令 $W[0] = 0$ 作为占位），便于与题目编号对应。
- **显式栈** S：使用 Python 列表模拟栈结构，保存当前已选物品的索引序列。入栈操作对应“选择该物品并向下一层搜索”（`append`），出栈操作对应“撤销最近一次选择并回溯”（`pop`）。因此，S 即为一条从根到当前节点的搜索路径。
- **剩余容量** $remain\_T$：记录当前路径下背包尚未填满的容量。每次选择物品 i 时执行 $remain\_T \leftarrow remain\_T - W_i$；回溯撤销物品时执行 $remain\_T \leftarrow remain\_T + W_i$。
- **当前尝试指针** i：表示下一步准备尝试的物品编号。回溯时将 i 更新为“刚撤销物品的下一个编号”（$i \leftarrow top\_item + 1$），从而保证每个物品最多被选择一次，并避免重复枚举相同组合。
- **计数器** $solution\_count$：用于统计并编号输出的可行解数量。

通过以上结构，递归回溯过程被等价地实现为栈的压入或弹出与状态变量更新，无需函数递归调用栈。

2.3 算法伪代码

Algorithm 1 背包问题回溯求解

初始化：$S \leftarrow \emptyset$, $i \leftarrow 1$, $remain\_T \leftarrow T$
while true **do**
  while $i \leq n$ **and** $remain\_T - W[i] \geq 0$ **do**
    $S \leftarrow S \cup \{i\}$
    $remain\_T \leftarrow remain\_T - W[i]$
    if $remain\_T = 0$ **then**
      输出解 S
      回溯：$i \leftarrow S.pop() + 1$
      $remain\_T \leftarrow remain\_T + W[i-1]$
      continue
    end if
    $i \leftarrow i + 1$
  end while
  if $S = \emptyset$ **then**
    return
  end if
  回溯：$i \leftarrow S.pop() + 1$
  $remain\_T \leftarrow remain\_T + W[i-1]$
end while

3 程序实现

3.1 Python 实现

以下是使用 Python 实现的部分代码：

```python
def solve_package_all_solutions(W, n, T):
    S = []
    i = 1
    remain_T = T
    solution_count = 0

    while True:
        while i <= n and remain_T - W[i] >= 0:
            S.append(i)
            remain_T -= W[i]
```

```
            if remain_T == 0:
                solution_count += 1
                print(f"解 {solution_count}: 物品序号 {S}")
                top_item = S.pop()
                remain_T += W[top_item]
                i = top_item + 1
                continue
            i += 1

        if not S:
            return

        top_item = S.pop()
        remain_T += W[top_item]
        i = top_item + 1
```

Listing 1: 背包问题求解程序

4 实验结果

4.1 测试用例 1

输入：
物品体积：$W = [2, 3, 5, 7]$ 背包容量：$T = 10$

输出：
开始搜索所有可能的解...
解 1：物品序号 [1, 2, 3] | 体积 [2, 3, 5] | 总体积 =10
解 2：物品序号 [2, 4] | 体积 [3, 7] | 总体积 =10
搜索完成，共找到 2 个解。

4.2 测试用例 2

输入：
物品体积：$W = [2, 3, 5]$
背包容量：$T = 8$

输出：
开始搜索所有可能的解...

解 1：物品序号 [2, 3] | 体积 [3, 5] | 总体积 =8
搜索完成，共找到 1 个解。

4.3 测试用例 3（无解情况）

输入：
物品体积：$W = [3, 5, 7]$
背包容量：$T = 11$

输出：
开始搜索所有可能的解...
无解！无法恰好装满背包。

5 复杂度分析

5.1 时间复杂度

在最坏情况下，算法需要遍历所有可能的物品组合。设物品数量为 n，背包容量为 T，最小物品体积为 W_{min}，则搜索树的最大深度为 $O(T/W_{min})$ ，每个节点最多有 n 个分支，时间复杂度为 $O(n^{T/W_{min}})$，为指数级

5.2 空间复杂度

空间复杂度主要由显式栈决定：

$$S(n) = O(T/W_{min}) \tag{1}$$

可达集合法实现

除回溯法外，我们还给出了“可达集合法”的一种简单实现，用于判断背包容量 T 是否存在可行组合。

思想 提出一个可达集合 a，其中每个元素表示当前遍历过的元素通过加法能达到的所有值。初始时 $a = \{0\}$。当遍历到一个物品体积 i 时，令

$$d = \{x + i \mid x \in a\}$$

并将新得到的体积并入 a。若在更新过程中发现 $T \in a$，即可提前结束并判定背包恰好能被装满。

6 评价和改进

6.1 评价

本报告实现了基于回溯法的背包问题求解算法，能够找出所有满足条件的物品组合方案。算法思路还算清晰，实现简洁，适用于求解小规模的背包问题。

6.2 优化方向

尽管回溯法能够枚举所有可行解并直观体现搜索过程，但其缺点也较为明显：在最坏情况下需要遍历接近指数规模的解空间，随着物品数量增加运行时间会急剧上升；同时输出全部解的数量本身可能非常大，I/O 开销也会成为瓶颈。此外，若剪枝条件较弱或物品顺序不合适，会产生大量无效分支搜索与重复回溯。

针对上述问题，首先可以从**剪枝与预处理**入手提升搜索效率，例如先对物品体积进行排序，使得“不可能放下”的情况更早暴露；在搜索过程中，当 $remain\_T$ 已小于后续物品的最小体积时可直接回溯；或利用“后续物品体积总和不足以填满 $remain\_T$”等上界/下界条件，提前剪去必然无解的分支，从而显著减少无效探索。

可以引入记忆来避免重复搜索，回溯过程中常会多次遇到相同状态，例如到达同一物品编号 i 且剩余容量同为 $remain\_T$ 的情形；若该状态已被证实无法得到解，则再次遇到时可直接跳过，从而减少大量重复回溯。

附录

```python
def solve_package_all_solutions(W, n, T):

    S = []
    i = 1
    remain_T = T
    solution_count = 0

    print("开始搜索所有可能的解...")

    while True:
        while i <= n and remain_T - W[i] >= 0:
            S.append(i)
            remain_T -= W[i]
            if remain_T == 0:
                solution_count += 1
                print(f"解 {solution_count}: 物品序号 {S}", end="")
```

```
                volumes = [W[idx] for idx in S]
                print(f" | 体积 {volumes} | 总体积={sum(volumes)}")


                top_item = S.pop()
                remain_T += W[top_item]
                i = top_item + 1
                continue

            i += 1

        if not S:
            if solution_count == 0:
                print("无解！无法恰好装满背包。")
            else:
                print(f"搜索完成，共找到 {solution_count} 个解。")
            return


        top_item = S.pop()
        remain_T += W[top_item]
        i = top_item + 1




if __name__ == "__main__":

    print("=" * 70)
    print("测试用例1".center(70))
    print("=" * 70)
    W1 = [0, 2, 3, 5, 7]
    n1 = 4
    T1 = 10

    print(f"物品体积: {W1[1:n1+1]}")
    print(f"背包容量: {T1}\n")


    solve_package_all_solutions(W1, n1, T1)
```

```
57
58
59      print("\n" + "=" * 70)
60      print("测试用例2".center(70))
61      print("=" * 70)
62      W2 = [0, 2, 3, 5]
63      n2 = 3
64      T2 = 8
65
66      print(f"物品体积: {W2[1:n2+1]}")
67      print(f"背包容量: {T2}\n")
68
69      solve_package_all_solutions(W2, n2, T2)
70
71      #无解的情况
72      print("\n" + "=" * 70)
73      print("测试用例3 (无解)".center(70))
74      print("=" * 70)
75      W3 = [0, 3, 5, 7]
76      n3 = 3
77      T3 = 11
78
79      print(f"物品体积: {W3[1:n3+1]}")
80      print(f"背包容量: {T3}\n")
81
82      solve_package_all_solutions(W3, n3, T3)
```

Listing 2: 回溯法求解程序

```
1   def neng_zhuang_man_bu(W, T):
2   a = [0] #用来装"可以通过W内数字相加达到的数字"的数组，初始为0，这是不
    证自明的
3   for i in W:
4       if i > T:
5           continue
6       d = [x + i for x in a] #为a中每个元素都加上本次遍历到的数字
7       a = a + d #把新数组与旧数组合并
8       if T in a:
9           print(f"体积{T}刚好能被填满")
10          return True
```

```
11
12      print(f"体积{T}没法被{W}刚好填满")
13      return False
14
15
16  if __name__ == "__main__":
17      # 测试示例
18      W = [2, 3, 5]
19      T = 8
20      neng_zhuang_man_bu(W, T)
```

Listing 3: 可达集合法求解程序