**广东海洋大学学生实验报告书（学生用表）**

**GDOU-B-11-112**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验名称 | 动态规划算法设计 | | | 课程名称 | | 算法设计与分析 | | | | 课程号 | | 32260035 |
| 学院(系) | 数学与计算机 | | 专业 | 物联网工程 | | | | 班级 | | 1224 | | |
| 学生姓名 | 黎川滔 | 学号 | 202211672411 | | 实验地点 | | 明德楼A1501 | | 实验日期 | | 2025.4.15 | |

# 实验目的

本次实验通过解决经典问题（如0-1背包、游艇租用、编辑距离等），掌握动态规划的子问题划分、状态转移方程设计和重叠子问题优化技巧。结合LeetCode题目巩固将动态规划应用于实际场景的能力。

# 实验环境

处理器：AMD Ryzen 5 6600H with Radeon Graphics。

LeetCode。

# 实验内容

## 3.1 实验任务1：0-1背包

n个商品组成集合，每个商品有两个属性vi和pi，分别表示体积和价值，背包容量为C。求满足背包容量为C的条件下，如何选择商品使价值最大。算法实例，背包限重C为10，vi（体积）与pi（价值）的值如下表所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| vi | 2 | 2 | 6 | 5 | 4 |
| pi | 6 | 3 | 5 | 4 | 6 |

尝试运用带备忘递归的方式实现问题求解：

（1）请写出动态数组的计算递推式；

P[i][c] = max(P[i-1][c], p[i-1] + P[i-1][c-v[i-1]]) if c >= v[i-1]

P[i][c] = P[i-1][c] if c < v[i-1]

P[i][c]表示前i个商品在容量为c时的最大价值，Rec[i][c]记录选择情况，1表示选，0表示不选。

（2）绘制表格P与Rec并根据算法填充数据；手绘亦可。在表格中标记出最优方案的选择商品过程，请用文字表述清决策过程；

表格P（最大价值表）：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i\c | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 2 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| 3 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 9 | 11 | 11 | 14 |
| 4 | 0 | 0 | 6 | 6 | 9 | 9 | 9 | 10 | 11 | 13 | 14 |
| 5 | 0 | 0 | 6 | 6 | 9 | 9 | 12 | 12 | 15 | 15 | 15 |

表格Rec（选择记录表）：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i\c | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

（3）根据伪代码写出完整程序，除了给定的测试集数据外，再自行构建一组数据进行测试。并展示测试数据与结果。

def knapsack\_memo(n, p, v, C):

memo = [[-1] \* (C + 1) for \_ in range(n + 1)]

rec = [[0] \* (C + 1) for \_ in range(n + 1)]

def dp(i, c):

if i == 0 or c == 0:

return 0

if memo[i][c] != -1:

return memo[i][c]

if v[i - 1] <= c:

include = p[i - 1] + dp(i - 1, c - v[i - 1])

exclude = dp(i - 1, c)

if include > exclude:

rec[i][c] = 1

memo[i][c] = include

else:

memo[i][c] = exclude

else:

memo[i][c] = dp(i - 1, c)

return memo[i][c]

max\_value = dp(n, C)

res = []

k = C

for i in range(n, 0, -1):

if rec[i][k] == 1:

res.append(i)

k -= v[i - 1]

res.reverse()

return max\_value, res

n = 5

v = [2, 2, 6, 5, 4]

p = [6, 3, 5, 4, 6]

C = 10

max\_val, items = knapsack\_memo(n, p, v, C)

print(f"测试1 - 最大价值: {max\_val}, 选择商品: {items}")

n2 = 4

v2 = [3, 4, 2, 5]

p2 = [4, 5, 3, 6]

C2 = 8

max\_val2, items2 = knapsack\_memo(n2, p2, v2, C2)

print(f"测试2 - 最大价值: {max\_val2}, 选择商品: {items2}")

文本

AI 生成的内容可能不正确。

（4）根据算法，完成时间复杂度分析。

子问题总数：O(n×C)，每个子问题求解时间：O(1)，总时间复杂度：O(n×C)。回溯求解方案需要O(n)时间整体时间复杂度为O(n×C)。

## 3.2 实验任务2：游艇租用问题

长江游艇俱乐部在长江上设置了n个游艇出租站1，2，…，n。游客可在这些游艇出租站租用游艇，并在下游的任何一个游艇出租站归还游艇。游艇出租站i到游艇出租站j之间的租金为r(i,j),1<=i<j<=n。试设计一个算法，计算出从游艇出租站1 到游艇出租站n所需的最少租金。

输入格式：

第1 行中有1 个正整数n（n<=200），表示有n个游艇出租站。接下来的第1到第n-1 行，第i行表示第i站到第i+1站,第i+2站, … , 第n站的租金。

输出格式:

输出从游艇出租站1 到游艇出租站n所需的最少租金。

输入样例:

3

5 15

7

输出样例:

12

1. 写出解题思路，总结出递推方程；

设dp[i]表示从第1站到第i站的最少租金。初始条件为dp[1] = 0（起点不需要租金），递推关系为：

dp[j] = min(dp[j], dp[i] + r(i,j)) for all i < j，其中r(i,j)是直接从i站到j站的租金。对于每个站点i（从1到n-1），遍历所有可能的j（从i+1到n），更新dp[j]的最小值。最终dp[n]即为从1站到n站的最少租金。

1. 分析该算法的时间复杂度；

双重循环遍历所有站点对(i,j)，其中i从1到n-1，j从i+1到n。因此时间复杂度为O(n²)。

（3）针对输入与输入要求，实现程序的代码，要求在报告中写明设计思路或添加注释说明；程序代码以文本的形式粘贴在报告中，以便于评阅人验证。

（4）自行构造不少于三组的测试集，并输出相应的结果。

def min\_rent(n, rents):

dp = [float('inf')] \* (n + 1)

dp[1] = 0 # 起点租金为0

for i in range(1, n):

for j in range(i + 1, n + 1):

if dp[i] + rents[i - 1][j - i - 1] < dp[j]:

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

return dp[n]

n = int(input())

rents = []

for i in range(n - 1):

rents.append(list(map(int, input().split())))

print(min\_rent(n, rents))

输入1：

3

5 15

7

输出1：

7

输入2：

4

2 6 9

3 8

5

输入2：

10

输入3：

5

1 3 5 7

2 4 6

8 3

5

输出3：

7

## 3.3 实验任务3：LeetCode72—编辑距离

class Solution:

def minDistance(self, word1: str, word2: str) -> int:

m, n = len(word1), len(word2)

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

# 初始化

for i in range(m + 1):

dp[i][0] = i

for j in range(n + 1):

dp[0][j] = j

# 动态规划填表

for i in range(1, m + 1):

for j in range(1, n + 1):

if word1[i-1] == word2[j-1]:

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

else:

dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1

return dp[m][n]

需要计算将 word1 转换成 word2 所需的最少操作次数。允许的操作包括：插入一个字符、删除一个字符、替换一个字符。设 dp[i][j] 表示将 word1 的前 i 个字符转换为 word2 的前 j 个字符所需的最少操作次数。如果 word1[i-1] == word2[j-1]，则 dp[i][j] = dp[i-1][j-1]；否则，dp[i][j] 可以来自以下三种操作的最小值：​​插入​​：dp[i][j-1] + 1、删除​​：dp[i-1][j] + 1、替换​​：dp[i-1][j-1] + 1。初始化的时间复杂度为O(m + n)，填表为O(m × n)，​​总时间复杂度为 O(m × n)。

## 3.4 实验任务4：LeetCode152—乘积最大子数组

class Solution:

def maxProduct(self, nums: List[int]) -> int:

def maxProduct(nums):

if not nums:

return 0

max\_prod = min\_prod = result = nums[0]

for num in nums[1:]:

if num > 0:

max\_prod, min\_prod = max(num, max\_prod \* num), min(num, min\_prod \* num)

elif num < 0:

max\_prod, min\_prod = max(num, min\_prod \* num), min(num, max\_prod \* num)

else:

max\_prod = min\_prod = 0

result = max(result, max\_prod)

return result

需要找到一个整数数组中乘积最大的非空连续子数组，并返回该乘积。子数组至少包含一个元素。max\_dp[i]：以 nums[i] 结尾的子数组的最大乘积。min\_dp[i]：以 nums[i] 结尾的子数组的最小乘积。时间复杂度​​：O(n)，其中 n 是数组的长度。对于每个 nums[i]：

如果 nums[i] 是正数：

max\_dp[i] = max(nums[i], max\_dp[i-1] \* nums[i])

min\_dp[i] = min(nums[i], min\_dp[i-1] \* nums[i])

如果 nums[i] 是负数：

max\_dp[i] = max(nums[i], min\_dp[i-1] \* nums[i])

min\_dp[i] = min(nums[i], max\_dp[i-1] \* nums[i])

如果 nums[i] 是零：

max\_dp[i] = 0

min\_dp[i] = 0

## 3.5 实验任务5：LeetCode64—最小路径和

from typing import List

class Solution:

def minPathSum(self, grid: List[List[int]]) -> int:

m, n = len(grid), len(grid[0])

dp = [[0] \* n for \_ in range(m)]

dp[0][0] = grid[0][0]

# 初始化第一行

for j in range(1, n):

dp[0][j] = dp[0][j-1] + grid[0][j]

# 初始化第一列

for i in range(1, m):

dp[i][0] = dp[i-1][0] + grid[i][0]

# 填充其他位置

for i in range(1, m):

for j in range(1, n):

dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]

return dp[m-1][n-1]

给定一个包含非负整数的 m x n 网格 grid，找出一条从左上角 (0, 0) 到右下角 (m-1, n-1) 的路径，使得路径上的数字总和最小。每次只能向下或向右移动一步。定义dp[i][j]：从 (0, 0) 到 (i, j) 的最小路径和。第一行​​：只能从左边过来，因此 dp[0][j] = dp[0][j-1] + grid[0][j]（j > 0）。第一列​​：只能从上边过来，因此 dp[i][0] = dp[i-1][0] + grid[i][0]（i > 0）。其他位置​​：可以从左边或上边过来，因此 dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]。边界条件：网格的行数 m 和列数 n 至少为 1。

## 4. 实验总结

在这次实验中，我通过资料的查阅成功实现了0-1背包问题的带备忘递归解法，并通过动态规划表（P和Rec）清晰展示了最优解的推导过程。在LeetCode题目的练习中，熟练应用动态规划解决字符串操作和路径优化问题，理解了状态转移方程的构建技巧。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 成绩 |  | 指导教师 |  | 日期 |  |