**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 | 分治法设计 | | | 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 | | 2025.3.25 | |

# 实验目的

本次实验旨在掌握分治法的基本思想及其在算法设计中的应用。通过具体问题（如假币查找、中位数求解、芯片测试等）实践分治法的实现，理解其递归模式和效率优势。与此同时，比较不同分治策略（如二分法与三分法）的异同，分析其时间复杂度及适用场景。

# 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：求解查找假币总问题

编写一个实验程序查找假币问题。有n(n > 3)个硬币，其中有一个假币，且假币较轻，采用天平称重方式找到这枚假币，并给出操作步骤。假设真币重量为个人学号最后两位模10加2，如202211672411号同学，学后最后两位为11，模10加2结果为3；假币的重量比真币的重量少2；采用二分法求解；写出算法的递归式，分析其时间复杂度。

def find\_counterfeit(coins, left, right, student\_number):

"""

二分法递归查找假币

:param coins: 硬币列表

:param left: 当前查找范围的左边界

:param right: 当前查找范围的右边界

:param student\_number: 真币的重量

:return: 假币的索引

"""

# 如果只有一个硬币，直接返回假币

if left == right:

return left

# 计算中间位置

mid = left + (right - left) // 2

# 根据硬币数量奇偶性调整分组

if (right - left + 1) % 2 == 0:

# 偶数个硬币，平分两组

group1\_left, group1\_right = left, mid

group2\_left, group2\_right = mid + 1, right

else:

# 奇数个硬币，中间那个单独处理

group1\_left, group1\_right = left, mid - 1

group2\_left, group2\_right = mid + 1, right

# 称重比较两组

group1\_weight = sum(coins[group1\_left:group1\_right + 1])

group2\_weight = sum(coins[group2\_left:group2\_right + 1])

if group1\_weight < group2\_weight:

# 假币在第一组

return find\_counterfeit(coins, group1\_left, group1\_right, student\_number)

elif group1\_weight > group2\_weight:

# 假币在第二组

return find\_counterfeit(coins, group2\_left, group2\_right, student\_number)

else:

# 两组重量相等，假币就是中间那个

return mid

def main():

import random

# 根据学号计算真币重量

student\_number = (11 % 10) + 2

# 假币质量为真币质量减1

quality\_counterfeit = student\_number - 1 # 结果为2

# 硬币数量

n = 10

# 初始化硬币数组全为真币

coins = [student\_number] \* n

# 随机选择一个硬币假币

counterfeit\_index = random.randint(0, n - 1)

coins[counterfeit\_index] = quality\_counterfeit

print(f"硬币列表: {coins}")

print(f"真币重量: {student\_number}, 假币重量: {quality\_counterfeit}")

# 查找假币

found\_index = find\_counterfeit(coins, 0, n - 1, student\_number)

print(f"找到的假币索引: {found\_index}, 重量: {coins[found\_index]}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

# 3.2 实验任务2：求解查找假币总问题

在实验任务1的基础上，把二分法修改成三分法。

def find\_counterfeit\_three(coins, left, right, student\_number):

"""

使用三分法递归查找假币

:param coins: 硬币列表

:param left: 当前查找范围的左边界

:param right: 当前查找范围的右边界

:param student\_number: 真币的重量

:return: 假币的索引

"""

if left == right:

return left

# 将硬币分成三组

import math

total = right - left + 1

group\_size = math.ceil(total / 3) # 向上取整

# 确定三个分组的边界

group1\_end = left + group\_size - 1

group2\_end = group1\_end + group\_size

# 称重比较前两组

group1\_weight = sum(coins[left:group1\_end + 1])

group2\_weight = sum(coins[group1\_end + 1:group2\_end + 1])

if group1\_weight < group2\_weight:

# 假币在第一组

return find\_counterfeit\_three(coins, left, group1\_end, student\_number)

elif group1\_weight > group2\_weight:

# 假币在第二组

return find\_counterfeit\_three(coins, group1\_end + 1, group2\_end, student\_number)

else:

# 假币在第三组

return find\_counterfeit\_three(coins, group2\_end + 1, right, student\_number)

def main():

import random

# 根据学号计算真币重量

student\_number = (11 % 10) + 2

# 假币质量为真币质量减1

quality\_counterfeit = student\_number - 1 # 结果为2

# 硬币数量

n = 1000

# 初始化硬币数组全为真币

coins = [student\_number] \* n

# 随机选择一个硬币假币

counterfeit\_index = random.randint(0, n - 1)

coins[counterfeit\_index] = quality\_counterfeit

print(f"硬币列表: {coins}")

print(f"真币重量: {student\_number}, 假币重量: {quality\_counterfeit}")

# 查找假币

found\_index = find\_counterfeit\_three(coins, 0, n - 1, student\_number)

print(f"找到的假币索引: {found\_index}, 重量: {coins[found\_index]}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

这两个实验任务分别采用二分法和三分法进行假硬币的查找，接下来将分析两种做法的异同。

分组的边界是最需要注意的地方，在二分法中：

# 计算中间位置

mid = left + (right - left) // 2

上面代码中关于mid的计算，采用的写法是左边界加上当前范围的半长度。假设硬币数量为10，那么left为0，right为9，因此(right - left) // 2的结果是4。这样的写法会将[0, 9]的10个硬币划分为[0, 4]的左区间和[5, 9]的右区间，可见硬币总数为偶数的情况下，左右区间的硬币数量一致。但如果硬币数量为11，mid的计算结果会是5，那么左区间为[0, 5]，而右区间为[6, 10]，可见右区间硬币数量比左区间多了1个。因此，二分法需要对硬币数量的奇偶分类讨论：

# 根据硬币数量奇偶性调整分组

if (right - left + 1) % 2 == 0:

# 偶数个硬币，平分两组

group1\_left, group1\_right = left, mid

group2\_left, group2\_right = mid + 1, right

else:

# 奇数个硬币，中间那个单独处理

group1\_left, group1\_right = left, mid - 1

group2\_left, group2\_right = mid + 1, right

当硬币总数为偶数时，平分两组；而当硬币总数为奇数时，中间的mid单独处理。处理过程比较直观，如果group1的硬币总质量小于group2，那么假硬币一定在group1；反之亦然。但如果group1和group2的硬币总质量相等，那么假硬币就是mid。

对于三分法而言，它的比较方式也是两组比较，如果group1和group2的硬币总质量相等，那么假硬币在group3中。首先，三分法分组数量的计算方式为：

total = right - left + 1

group\_size = math.ceil(total / 3) # 向上取整

total是硬币总数，之所以没有把n传入函数，这是因为每次递归时分组的硬币数量都不一样，因此需要重新计算。group\_size是分组数量，通过除以3后向上取整。假设硬币数量为10个，那么每组的大小为4。对于三分法的分组数量划分方式，让前两组相等，第三组尽可能相等：

# 确定三个分组的边界

group1\_end = left + group\_size - 1

group2\_end = group1\_end + group\_size

以硬币总数为10为例，group1和group2各4 个，group3有2个。后面的思路和二分法一样。二分法的时间复杂度是，三分法的时间复杂度是。

# 3.3 实验任务3：中位数问题

设X[ 0 : n - 1]和Y[ 0 : n – 1 ]为两个数组，每个数组中含有n个已排好序的数（排序方法自定）。找出X和Y的2n个数的中位数。数据采用随机数生成，种子利用个人学号与学号除以2分别进行填充两个数组。利用分治策略试设计一个时间的算法求出这2n个数的中位数。写出其递归模式，并分析时间复杂度；提供程序源代码。

import random

def find\_median\_sorted\_arrays(X, Y, n):

if n == 1:

return (X[0] + Y[0]) / 2 # 偶数长度时取平均

mid = n // 2

if X[mid] <= Y[mid]:

return find\_median\_sorted\_arrays(X[mid:], Y[:mid + 1], n - mid)

else:

return find\_median\_sorted\_arrays(X[:mid + 1], Y[mid:], n - mid)

def main():

student\_id = 11

random.seed(student\_id)

n = 10 # 数组长度

X = sorted([random.randint(1, 100) for \_ in range(n)])

Y = sorted([random.randint(1, 100) for \_ in range(n)])

print("X:", X)

print("Y:", Y)

# 计算中位数

median = find\_median\_sorted\_arrays(X, Y, n)

print("Median:", median)

if \_\_name\_\_ == "\_\_main\_\_":

main()

在分析上述代码之前，需要理解合并后具有2n个元素的有序数组的性质：恰好有n-1个数比它小，且恰好有n-1个数比它大，中位数是第n小的数（或n和n+1的平均数）。

假设数组X = [1, 3, 5, 7, 9]和数组Y = [2, 4, 6, 8, 10]，数组长度n=5。在算法中，首先计算mid = 2，此时有X[mid] = 5，Y[mid] = 6，满足X[mid] <= Y[mid]的条件，那么分治的划分为：X的右半部分[5, 7, 9]和Y的左半部分加上中间元素[2, 4, 6]。这样划分的依据是：X[mid] <= Y[mid]说明X的左半部分[1, 3]和Y的右半部分[8, 10]不可能包含中位数，中位数只可能在X的右半部分或Y的左半部分。

# 3.4 实验任务4：芯片测试问题

有n片芯片，已知其中好芯片比坏芯片至少多1片。现在需要通过测试从中找出 1 片好芯片。

测试的方法是：将2片芯片放到测试台上，2片芯片互相测试并报告测试结果：“好”或者“坏”。假定好芯片的报告是正确的，坏芯片的报告是不可靠的（可能是对的，也可能是错的）。试设计一个算法，使用最少的测试次数来找出1片好芯片。

对形成的完整程序要求如下：

（1）每个芯片，记录其好或者坏的标记，以及对应的序号；

（2）构造芯片测试集，要求好的芯片至少比坏的芯片多1；生成数据集并乱序。

（3）测试数据对中，如果A芯片为好芯片，则返回B芯片的真实情况；如果A芯片为坏芯片，则随机返回好或坏的结果。

建议采用智能体进行问题的求解。

回答以下几个问题：

（1）测试过程中有没有找不到好芯片的情况；如果有，是哪里出了问题；

（2）你在编写程序时遇到了哪些情况，分别是如何解决的。

（3）你感觉这个问题哪些是较难理解的项。

（4）第1片坏芯片，为什么最多测n-2次。

（5）适当总结你解决该问题的过程与感悟，如提示词的提供，反馈过程中的沟通与交流等。

import random

from typing import List, Tuple

class Chip:

def \_\_init\_\_(self, id: int, is\_good: bool):

self.id = id

self.is\_good = is\_good

def test\_chips(a: Chip, b: Chip) -> bool:

"""模拟芯片测试：A 测试 B 的结果"""

if a.is\_good:

return b.is\_good # 好芯片的报告是真实的

else:

return random.choice([True, False]) # 坏芯片随机返回

def find\_one\_good\_chip(chips: List[Chip]) -> Chip:

"""找出 1 片好芯片"""

candidates = chips.copy()

while len(candidates) > 1:

next\_round = []

for i in range(0, len(candidates) - 1, 2):

a, b = candidates[i], candidates[i + 1]

# A 测试 B 和 B 测试 A

a\_says = test\_chips(a, b)

b\_says = test\_chips(b, a)

if a\_says and b\_says: # 互相报告好，保留一片

next\_round.append(a)

# 否则丢弃这两片

if len(candidates) % 2 == 1: # 奇数个，保留最后一片

next\_round.append(candidates[-1])

candidates = next\_round

return candidates[0] # 最后剩下的必是好芯片

def main():

random.seed(202211672411) # 固定随机种子

n = 10 # 芯片总数

good\_num = (n // 2) + 1 # 好芯片比坏芯片至少多 1

bad\_num = n - good\_num

# 生成芯片（前 good\_num 是好芯片）

chips = [Chip(i, True) for i in range(good\_num)] + \

[Chip(i + good\_num, False) for i in range(bad\_num)]

random.shuffle(chips) # 打乱顺序

print("初始芯片状态：")

for chip in chips:

print(f"芯片 {chip.id}: {'好' if chip.is\_good else '坏'}")

# 找出 1 片好芯片

good\_chip = find\_one\_good\_chip(chips)

print(f"\n找到的好芯片是 {good\_chip.id}（验证：{'正确' if good\_chip.is\_good else '错误'}）")

if \_\_name\_\_ == "\_\_main\_\_":

main()

# 3.5 实验任务5：LeetCode

215. 数组中的第K个最大元素

调库写法：

class Solution:

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

return sorted(nums)[-k]

最小堆写法：

class Solution:

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

# 查小用最大堆；查大用最小堆

# if k >= len(nums) or k < 0:

# return nums[0]

heap = [] # 初始化最小堆

for num in nums:

if len(heap) < k: # 如果最小堆内的元素还不到k个，那就直接添加

heapq.heappush(heap, num)

else:

if num > heap[0]: # 维护最小堆，堆顶最大

heapq.heappop(heap) # 踢出

heapq.heappush(heap, num) # 入堆，会自动调整大小

return heap[0]

912. 排序数组

实践一下基数排序，之前应用于学号的升序（字符串），但对于数值型数组排序就需要进一步处理了。

class Solution:

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

if not nums:

return nums

# 分离正数和负数

positives = [num for num in nums if num >= 0]

negatives = [-num for num in nums if num < 0] # 取绝对值

# 对正数进行基数排序

positives\_sorted = self.radix\_sort(positives)

# 对负数进行基数排序（取绝对值后排序，再反转并加负号）

negatives\_sorted = [-num for num in self.radix\_sort(negatives)[::-1]]

return negatives\_sorted + positives\_sorted

def radix\_sort(self, nums):

if not nums:

return nums

max\_num = max(nums)

max\_digits = len(str(max\_num))

for i in range(max\_digits):

buckets = [[] for \_ in range(10)]

exp = 10 \*\* i # 计算当前位数（个位、十位、百位...）

for num in nums:

digit = (num // exp) % 10 # 提取当前位的数字

buckets[digit].append(num)

nums = [num for bucket in buckets for num in bucket]

return nums

# 实验总结

我在这次实验的收获非常大，尤其是二分法和三分法的应用，这让我理解了分治的思想，并且能拓展到归并排序等算法中。同时，我按照老师的要求，去力扣平台刷题巩固，这对我的分析能力的提升有显著的促进。

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