检索算法实验程序

📋 目录

项目概述

实验目标

实现检索算法实验，包含7个功能模块，演示不同检索算法的效率差异。

核心功能

- **功能2**: 数组生成（随机/手动）
- **功能3**: 判断数组排序状态
- **功能4**: 顺序检索
- **功能5**: 多种方法检索（顺序、二分、三分）
- **功能6**: 查找峰值（二分、三分）
- **功能7**: 查找第k小元素（蛮力法、预排序）

代码结构

```
find.py
├── 导入模块 (6-9行)
├── 核心算法部分 (11-200行)
│   ├── sequentialSearch()        # 顺序查找
│   ├── binarySearch()            # 二分查找
│   ├── ternarySearch()           # 三分查找
│   ├── binarySearchPeak()        # 二分找峰值
│   └── ternarySearchPeak()       # 三分找峰值
├── 辅助算法部分 (202-330行)
│   ├── checkArrayOrder()         # 判断排序状态
│   ├── findKthSmallestBruteForce() # 蛮力法找第k小
│   ├── findKthSmallestPreSort()  # 预排序找第k小
│   └── quickSort()               # 快速排序
└── GUI部分 (333-784行)
    └── SearchGUI类
        ├── __init__()            # 界面初始化
        ├── generateRandomArray()  # 随机生成数组
```

```
├── inputArrayManually()    # 手动输入数组
├── checkOrder()            # 功能3
├── sequentialSearchFunc()  # 功能4
├── multipleSearch()        # 功能5
├── findPeak()              # 功能6
└── findKthSmallest()       # 功能7
```

核心算法详解

1. 顺序查找 (sequentialSearch)

设计思想:

- 最简单直观的查找方法
- 从第一个元素开始，逐个比较直到找到或遍历完
- 适用于任何数组（有序或无序）

代码实现:

```python
def sequentialSearch(arr, target):
    """
    顺序查找算法
    时间复杂度：O(n) - 最坏情况需要遍历所有元素
    空间复杂度：O(1) - 只使用常数额外空间
    返回：(位置索引, 比较次数)
    """
    comparisons = 0  # 统计比较次数
    for i in range(len(arr)):
        comparisons += 1  # 每次循环都是一次比较
        if arr[i] == target:
            return i, comparisons  # 找到，返回位置和比较次数
    return -1, comparisons  # 未找到，返回-1和比较次数
```

关键点:

- `comparisons` 计数器：每次循环都递增，精确统计比较次数
- 返回值：元组包含位置索引和比较次数，便于性能分析

---

2. 二分查找 (binarySearch)

设计思想:

- 利用数组有序的特性，每次排除一半的搜索空间
- 类似"猜数字"游戏：每次猜中间值，根据结果缩小范围
- 时间复杂度从O(n)降到O(log n)

代码实现:

```python
def binarySearch(arr, target):
    """
    二分查找算法（要求数组已排序）
    时间复杂度：O(log n) - 每次排除一半元素
    空间复杂度：O(1)
    返回：(位置索引, 比较次数, 最接近的元素信息)
```

```
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1
    closestIdx = -1      # 记录最接近元素的索引
    closestDiff = float('inf')  # 记录最小差值

    while left <= right:
        mid = (left + right) // 2  # 计算中间位置
        comparisons += 1

        if arr[mid] == target:
            return mid, comparisons, (arr[mid], mid)

        # 记录最接近的元素（用于未找到时返回）
        diff = abs(arr[mid] - target)
        if diff < closestDiff:
            closestDiff = diff
            closestIdx = mid

        comparisons += 1  # 这次比较用于决定搜索方向
        if arr[mid] < target:
            left = mid + 1   # 目标在右半部分
        else:
            right = mid - 1  # 目标在左半部分

    # 处理边界情况：如果closestIdx还是-1，使用边界值
    if closestIdx == -1:
        if left < len(arr):
            closestIdx = left
        elif right >= 0:
            closestIdx = right

    closestInfo = (arr[closestIdx], closestIdx) if closestIdx != -1 else None
    return -1, comparisons, closestInfo
```

关键点：

- **分治思想**：每次将问题规模减半
- **最接近元素**：未找到时返回最接近的元素，满足实验要求
- **边界处理**：确保在数组边界时也能正确返回

算法流程：

```
数组: [1, 3, 5, 7, 9, 11, 13]
查找: 7

第1次: mid=3, arr[3]=7 ✓ 找到!
比较次数: 1
```

---

3. 三分查找 (ternarySearch)

设计思想：

- 将区间分成三份，而不是两份
- 理论上比二分查找更快（$O(\log_3 n)$ vs $O(\log_2 n)$）

- 但实际应用中，由于每次需要比较两个点，常数因子更大

代码实现：

```
def ternarySearch(arr, target):
    """
    三分查找算法（要求数组已排序）
    时间复杂度：O(log₃ n) - 每次排除2/3的元素
    空间复杂度：O(1)
    返回：(位置索引, 比较次数, 最接近的元素信息)
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1
    closestIdx = -1
    closestDiff = float('inf')

    while left <= right:
        # 优化：区间太小时直接用顺序查找
        if right - left < 2:
            for i in range(left, right + 1):
                comparisons += 1
                diff = abs(arr[i] - target)
                if diff < closestDiff:
                    closestDiff = diff
                    closestIdx = i
                if arr[i] == target:
                    closestInfo = (arr[closestIdx], closestIdx) if closestIdx !=
-1 else None
                    return i, comparisons, closestInfo
            break

        # 将区间分成三份
        mid1 = left + (right - left) // 3   # 第一个三分点
        mid2 = right - (right - left) // 3  # 第二个三分点

        comparisons += 1
        if arr[mid1] == target:
            return mid1, comparisons, (arr[mid1], mid1)

        comparisons += 1
        if arr[mid2] == target:
            return mid2, comparisons, (arr[mid2], mid2)

        # 记录最接近的元素
        diff1 = abs(arr[mid1] - target)
        diff2 = abs(arr[mid2] - target)
        if diff1 < closestDiff:
            closestDiff = diff1
            closestIdx = mid1
        if diff2 < closestDiff:
            closestDiff = diff2
            closestIdx = mid2

        # 根据比较结果决定搜索区间
        comparisons += 1
        if target < arr[mid1]:
```

```
            right = mid1 - 1      # 目标在左1/3
        elif target > arr[mid2]:
            left = mid2 + 1       # 目标在右1/3
        else:
            left = mid1 + 1       # 目标在中间1/3
            right = mid2 - 1

    closestInfo = (arr[closestIdx], closestIdx) if closestIdx != -1 else None
    return -1, comparisons, closestInfo
```

关键点：

- **三区间划分**：`mid1` 和 `mid2` 将区间分成三份
- **优化处理**：区间太小时直接顺序查找，避免过度递归
- **三种情况**：目标可能在左1/3、中间1/3或右1/3

算法流程：

```
数组: [1, 3, 5, 7, 9, 11, 13]
查找: 7

第1次: mid1=2(arr[2]=5), mid2=4(arr[4]=9)
       5 < 7 < 9, 搜索中间区间 [6, 8]
第2次: 在中间区间找到 7
比较次数: 3
```

---

4. 二分查找峰值 (binarySearchPeak)

设计思想：

- 用于"先升后降"或"先降后升"的单峰数组
- 利用单调性：在峰值左侧单调递增，右侧单调递减
- 通过比较中间元素和相邻元素判断峰值位置

代码实现：

```
def binarySearchPeak(arr, findMax=True):
    """
    二分查找找峰值（最大值或最小值）
    用于先升后降或先降后升数组
    时间复杂度：O(log n)
    返回：(位置索引, 值, 比较次数)
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1

    while left < right:  # 注意：这里是 < 而不是 <=
        mid = (left + right) // 2
        comparisons += 1

        if findMax:
            # 找最大值：先升后降数组
            if arr[mid] < arr[mid + 1]:
                left = mid + 1   # 峰值在右侧（还在上升）
            else:
```

```
                right = mid      # 峰值在左侧（开始下降）
        else:
            # 找最小值：先降后升数组
            if arr[mid] > arr[mid + 1]:
                left = mid + 1   # 谷值在右侧（还在下降）
            else:
                right = mid      # 谷值在左侧（开始上升）

    return left, arr[left], comparisons
```

关键点：

- **单调性利用**：通过比较 `arr[mid]` 和 `arr[mid+1]` 判断趋势
- **边界条件**：使用 `left < right` 确保最终 `left == right` 时就是峰值
- **通用性**：`findMax` 参数支持找最大值或最小值

算法流程（找最大值）：

```
数组: [1, 3, 5, 7, 9, 6, 4]  (先升后降)
查找最大值

第1次: mid=3, arr[3]=7, arr[4]=9
       7 < 9，峰值在右侧，left=4
第2次: mid=5, arr[5]=6, arr[6]=4
       6 > 4，峰值在左侧，right=5
第3次: left=4, right=5, mid=4
       arr[4]=9, arr[5]=6
       9 > 6，right=4
       最终: left=right=4，最大值是9
```

---

5. 三分查找峰值 (ternarySearchPeak)

设计思想：

- 类似三分查找，将区间分成三份
- 比较两个三分点的值，判断峰值在哪一部分
- 理论上比二分查找更快

代码实现：

```
def ternarySearchPeak(arr, findMax=True):
    """
    三分查找找峰值（最大值或最小值）
    时间复杂度：O(log₃ n)
    返回：(位置索引, 值, 比较次数)
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1

    while right - left > 2:  # 区间大于2时继续
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        comparisons += 2  # 比较两次
        if findMax:
```

```
            if arr[mid1] < arr[mid2]:
                left = mid1 + 1   # 峰值在右侧2/3
            else:
                right = mid2 - 1  # 峰值在左侧2/3
        else:
            if arr[mid1] > arr[mid2]:
                left = mid1 + 1   # 谷值在右侧2/3
            else:
                right = mid2 - 1  # 谷值在左侧2/3

    # 剩余区间用顺序查找找最值
    if findMax:
        maxIdx = left
        for i in range(left + 1, right + 1):
            comparisons += 1
            if arr[i] > arr[maxIdx]:
                maxIdx = i
        return maxIdx, arr[maxIdx], comparisons
    else:
        minIdx = left
        for i in range(left + 1, right + 1):
            comparisons += 1
            if arr[i] < arr[minIdx]:
                minIdx = i
        return minIdx, arr[minIdx], comparisons
```

辅助算法详解

1. 判断数组排序状态 (checkArrayOrder)

设计思想:

- 分层次检查：先检查简单情况（完全升序/降序），再检查复杂情况（先升后降/先降后升）
- 使用Python的 `all()` 函数简化代码

代码实现:

```
def checkArrayOrder(arr):
    """
    判断数组的排序状态
    返回：0-未排序，1-升序，2-降序，3-先升后降，4-先降后升
    """
    if len(arr) <= 1:
        return 1  # 单个元素视为升序

    # 第一层：检查是否完全升序
    isAscending = all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))
    if isAscending:
        return 1

    # 第二层：检查是否完全降序
    isDescending = all(arr[i] >= arr[i + 1] for i in range(len(arr) - 1))
    if isDescending:
        return 2
```

```
    # 第三层：找峰值点（先升后降）
    peakIdx = -1
    for i in range(1, len(arr) - 1):
        if arr[i] > arr[i - 1] and arr[i] > arr[i + 1]:
            peakIdx = i
            break

    if peakIdx != -1:
        # 验证峰值前升序，峰值后降序
        beforeAsc = all(arr[i] <= arr[i + 1] for i in range(peakIdx))
        afterDesc = all(arr[i] >= arr[i + 1] for i in range(peakIdx, len(arr) -
1))
        if beforeAsc and afterDesc:
            return 3

    # 第四层：找谷值点（先降后升）
    valleyIdx = -1
    for i in range(1, len(arr) - 1):
        if arr[i] < arr[i - 1] and arr[i] < arr[i + 1]:
            valleyIdx = i
            break

    if valleyIdx != -1:
        # 验证谷值前降序，谷值后升序
        beforeDesc = all(arr[i] >= arr[i + 1] for i in range(valleyIdx))
        afterAsc = all(arr[i] <= arr[i + 1] for i in range(valleyIdx, len(arr) -
1))
        if beforeDesc and afterAsc:
            return 4

    return 0  # 未排序
```

设计亮点：

- **分层判断**：从简单到复杂，提高效率
- **验证机制**：找到峰值/谷值后，验证前后是否符合要求
- **边界处理**：单个元素视为升序

---

2. 快速排序 (quickSort)

设计思想：

- 分治算法：选择一个基准（pivot），将数组分成两部分
- 递归处理：对左右两部分递归排序
- 统计比较次数：每次比较都计数

代码实现：

```
def quickSort(arr):
    """
    快速排序算法
    时间复杂度：平均O(n log n)，最坏O(n²)
    空间复杂度：O(log n) - 递归栈
    返回：(排序后的数组, 比较次数)
    """
    if len(arr) <= 1:
```

```
        return arr, 0

    comparisons = 0
    arrCopy = arr.copy()  # 不修改原数组

    def partition(low, high):
        """
        分区函数：将数组分成两部分
        小于等于pivot的在左边，大于的在右边
        """
        nonlocal comparisons
        pivot = arrCopy[high]  # 选择最后一个元素作为基准
        i = low - 1  # 小于pivot的元素的索引

        for j in range(low, high):
            comparisons += 1
            if arrCopy[j] <= pivot:
                i += 1
                arrCopy[i], arrCopy[j] = arrCopy[j], arrCopy[i]

        # 将pivot放到正确位置
        arrCopy[i + 1], arrCopy[high] = arrCopy[high], arrCopy[i + 1]
        return i + 1  # 返回pivot的位置

    def quickSortRecursive(low, high):
        """递归排序函数"""
        nonlocal comparisons
        if low < high:
            pi = partition(low, high)  # 分区
            quickSortRecursive(low, pi - 1)  # 排序左半部分
            quickSortRecursive(pi + 1, high)  # 排序右半部分

    quickSortRecursive(0, len(arrCopy) - 1)
    return arrCopy, comparisons
```

关键点：

- **分区策略**：使用最后一个元素作为pivot（简单但可能不是最优）
- **原地排序**：在数组副本上操作，不修改原数组
- **比较统计**：每次 `arrCopy[j] <= pivot` 比较都计数

算法流程：

```
数组: [3, 1, 4, 1, 5, 9, 2, 6]
pivot = 6

分区过程:
[3, 1, 4, 1, 5, 2, 6, 9]
      ↑              ↑
   小于6          大于6

递归排序左右两部分...
```

---

3. 蛮力法找第k小 (findKthSmallestBruteForce)

设计思想：

- 选择排序思想：每次找最小元素，找k次
- 简单直观，但效率较低
- 时间复杂度O(n×k)，适合k较小的情况

代码实现：

```
def findKthSmallestBruteForce(arr, k):
    """
    蛮力法找第k个最小元素
    时间复杂度：O(n×k) - k次选择，每次O(n)
    空间复杂度：O(n) - 需要数组副本
    返回：(元素值, 位置索引, 比较次数)
    """
    if k < 1 or k > len(arr):
        return None, -1, 0

    comparisons = 0
    arrCopy = arr.copy()  # 不修改原数组

    # 执行k次选择排序
    for i in range(k):
        minIdx = i
        # 在剩余元素中找最小值
        for j in range(i + 1, len(arrCopy)):
            comparisons += 1
            if arrCopy[j] < arrCopy[minIdx]:
                minIdx = j
        # 将最小值交换到位置i
        arrCopy[i], arrCopy[minIdx] = arrCopy[minIdx], arrCopy[i]

    kthValue = arrCopy[k - 1]  # 第k个最小元素
    originalIdx = arr.index(kthValue)  # 在原数组中的位置
    return kthValue, originalIdx, comparisons
```

---

4. 预排序法找第k小 (findKthSmallestPreSort)

设计思想：

- 先排序，再取第k个元素
- 使用快速排序，时间复杂度O(n log n)
- 当k接近n时，比蛮力法更高效

代码实现：

```
def findKthSmallestPreSort(arr, k):
    """
    预排序法找第k个最小元素（使用快速排序）
    时间复杂度：O(n log n) - 排序时间
    空间复杂度：O(n)
    返回：(元素值, 位置索引, 比较次数)
    """
    if k < 1 or k > len(arr):
        return None, -1, 0

    # 使用快速排序
    sortedArr, comparisons = quickSort(arr)
```

```
    kthValue = sortedArr[k - 1]  # 排序后第k个元素
    originalIdx = arr.index(kthValue)  # 在原数组中的位置

    return kthValue, originalIdx, comparisons
```

两种方法对比：

- **蛮力法**：O(n×k)，k小时更快
- **预排序法**：O(n log n)，k大时更快

GUI设计思想

整体架构

设计原则：

1. **单一职责**：每个方法只负责一个功能
2. **用户友好**：清晰的界面布局，明确的提示信息
3. **数据保护**：算法操作数组副本，不修改原数组
4. **实时反馈**：输出框实时显示结果

界面布局设计

```
class SearchGUI:
    def __init__(self, root):
        # 1. 数组生成区域（顶部）
        #    - 输入框：数组长度n
        #    - 按钮：随机生成、手动输入、快速排序
        #    - 标签：显示当前数组

        # 2. 功能按钮区域（中部）
        #    - 6个功能按钮，分3行排列

        # 3. 输入框区域（中下部）
        #    - 查找元素输入框
        #    - k值输入框

        # 4. 输出文本框（下部）
        #    - 滚动文本框，显示所有结果

        # 5. 清空按钮（底部）
```

关键设计模式

1. 自动排序机制（功能5）

设计思想：智能判断数组状态，自动处理

```
def multipleSearch(self):
    # 检测数组状态
    originalOrderType = checkArrayOrder(self.currentArray)

    # 复制数组（保护原数组）
    searchArray = self.currentArray.copy()
```

```
    sortComparisons = 0
    isSorted = False

    # 智能处理：如果不是升序/降序，自动排序
    if originalOrderType not in [1, 2]:
        sortedArray, sortComparisons = quickSort(searchArray)
        searchArray = sortedArray
        isSorted = True
    elif originalOrderType == 2:
        # 降序数组反转成升序
        searchArray = searchArray[::-1]
```

优势：

- 用户无需手动排序
- 自动统计排序比较次数
- 原数组保持不变

2. 位置索引映射

问题：排序后查找的位置是排序数组中的位置，需要映射回原数组

解决方案：

```
# 情况1：数组被排序
if isSorted:
    if pos2 != -1:
        # 通过值找到在原数组中的位置
        originalPos2 = self.currentArray.index(searchArray[pos2])

# 情况2：降序数组被反转
elif originalOrderType == 2:
    if pos2 != -1:
        # 反转后的位置需要转换
        originalPos2 = len(searchArray) - 1 - pos2
```

3. 比较次数统计

设计：每个算法都精确统计比较次数

```
# 示例：二分查找
comparisons = 0
while left <= right:
    comparisons += 1  # 比较 arr[mid] == target
    if arr[mid] == target:
        return ...
    comparisons += 1  # 比较 arr[mid] < target
    if arr[mid] < target:
        ...
```

用途：

- 性能分析
- 算法效率对比
- 实验报告数据

完整代码

```
"""
检索算法实验程序
包含核心算法：顺序查找、二分查找、三分查找
"""

import random
import time
from tkinter import *
from tkinter import scrolledtext, messagebox

# ==================== 核心算法部分 ====================

def sequentialSearch(arr, target):
    """
    顺序查找算法
    时间复杂度：O(n)
    返回：(位置索引, 比较次数)
    """
    comparisons = 0
    for i in range(len(arr)):
        comparisons += 1
        if arr[i] == target:
            return i, comparisons
    return -1, comparisons


def binarySearch(arr, target):
    """
    二分查找算法（要求数组已排序）
    时间复杂度：O(log n)
    返回：(位置索引, 比较次数, 最接近的元素信息)
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1
    closestIdx = -1
    closestDiff = float('inf')

    while left <= right:
        mid = (left + right) // 2
        comparisons += 1

        if arr[mid] == target:
            return mid, comparisons, (arr[mid], mid)

        # 记录最接近的元素
        diff = abs(arr[mid] - target)
        if diff < closestDiff:
            closestDiff = diff
            closestIdx = mid

        comparisons += 1
        if arr[mid] < target:
```

```
            left = mid + 1
        else:
            right = mid - 1

    # 如果没找到，检查边界值
    if closestIdx == -1:
        if left < len(arr):
            closestIdx = left
        elif right >= 0:
            closestIdx = right

    closestInfo = (arr[closestIdx], closestIdx) if closestIdx != -1 else None
    return -1, comparisons, closestInfo


def ternarySearch(arr, target):
    """
    三分查找算法（要求数组已排序）
    时间复杂度：O(log3 n)
    返回：(位置索引, 比较次数, 最接近的元素信息)
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1
    closestIdx = -1
    closestDiff = float('inf')

    while left <= right:
        if right - left < 2:
            # 区间太小，用顺序查找
            for i in range(left, right + 1):
                comparisons += 1
                diff = abs(arr[i] - target)
                if diff < closestDiff:
                    closestDiff = diff
                    closestIdx = i
                if arr[i] == target:
                    closestInfo = (arr[closestIdx], closestIdx) if closestIdx !=
-1 else None
                    return i, comparisons, closestInfo
            break

        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        comparisons += 1
        if arr[mid1] == target:
            return mid1, comparisons, (arr[mid1], mid1)

        comparisons += 1
        if arr[mid2] == target:
            return mid2, comparisons, (arr[mid2], mid2)

        # 记录最接近的元素
        diff1 = abs(arr[mid1] - target)
        diff2 = abs(arr[mid2] - target)
        if diff1 < closestDiff:
            closestDiff = diff1
```

```
            closestIdx = mid1
        if diff2 < closestDiff:
            closestDiff = diff2
            closestIdx = mid2

        comparisons += 1
        if target < arr[mid1]:
            right = mid1 - 1
        elif target > arr[mid2]:
            left = mid2 + 1
        else:
            left = mid1 + 1
            right = mid2 - 1

    closestInfo = (arr[closestIdx], closestIdx) if closestIdx != -1 else None
    return -1, comparisons, closestInfo


def ternarySearchPeak(arr, findMax=True):
    """
    三分查找找峰值（最大值或最小值）
    用于先升后降或先降后升数组
    时间复杂度：O(log3 n)
    返回：(位置索引, 值, 比较次数)
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1

    while right - left > 2:
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        comparisons += 2
        if findMax:
            if arr[mid1] < arr[mid2]:
                left = mid1 + 1
            else:
                right = mid2 - 1
        else:
            if arr[mid1] > arr[mid2]:
                left = mid1 + 1
            else:
                right = mid2 - 1

    # 在剩余区间找最值
    if findMax:
        maxIdx = left
        for i in range(left + 1, right + 1):
            comparisons += 1
            if arr[i] > arr[maxIdx]:
                maxIdx = i
        return maxIdx, arr[maxIdx], comparisons
    else:
        minIdx = left
        for i in range(left + 1, right + 1):
            comparisons += 1
            if arr[i] < arr[minIdx]:
```

```
                minIdx = i
        return minIdx, arr[minIdx], comparisons


def binarySearchPeak(arr, findMax=True):
    """
    二分查找找峰值（最大值或最小值）
    用于先升后降或先降后升数组
    时间复杂度：O(log n)
    返回：(位置索引, 值, 比较次数)
    """
    comparisons = 0
    left = 0
    right = len(arr) - 1

    while left < right:
        mid = (left + right) // 2
        comparisons += 1

        if findMax:
            if arr[mid] < arr[mid + 1]:
                left = mid + 1
            else:
                right = mid
        else:
            if arr[mid] > arr[mid + 1]:
                left = mid + 1
            else:
                right = mid

    return left, arr[left], comparisons


# ==================== 辅助算法 ====================

def checkArrayOrder(arr):
    """
    判断数组的排序状态
    返回：0-未排序, 1-升序, 2-降序, 3-先升后降, 4-先降后升
    """
    if len(arr) <= 1:
        return 1  # 单个元素视为升序

    # 检查是否完全升序
    isAscending = all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))
    if isAscending:
        return 1

    # 检查是否完全降序
    isDescending = all(arr[i] >= arr[i + 1] for i in range(len(arr) - 1))
    if isDescending:
        return 2

    # 找峰值点
    peakIdx = -1
    for i in range(1, len(arr) - 1):
        if arr[i] > arr[i - 1] and arr[i] > arr[i + 1]:
            peakIdx = i
```

```
            break

    if peakIdx != -1:
        # 检查峰值前是否升序，峰值后是否降序
        beforeAsc = all(arr[i] <= arr[i + 1] for i in range(peakIdx))
        afterDesc = all(arr[i] >= arr[i + 1] for i in range(peakIdx, len(arr) -
1))
        if beforeAsc and afterDesc:
            return 3

    # 找谷值点
    valleyIdx = -1
    for i in range(1, len(arr) - 1):
        if arr[i] < arr[i - 1] and arr[i] < arr[i + 1]:
            valleyIdx = i
            break

    if valleyIdx != -1:
        # 检查谷值前是否降序，谷值后是否升序
        beforeDesc = all(arr[i] >= arr[i + 1] for i in range(valleyIdx))
        afterAsc = all(arr[i] <= arr[i + 1] for i in range(valleyIdx, len(arr) -
1))
        if beforeDesc and afterAsc:
            return 4

    return 0  # 未排序


def findKthSmallestBruteForce(arr, k):
    """
    蛮力法找第k个最小元素
    时间复杂度：O(n*k)
    返回：(元素值, 位置索引, 比较次数)
    """
    if k < 1 or k > len(arr):
        return None, -1, 0

    comparisons = 0
    arrCopy = arr.copy()

    for i in range(k):
        minIdx = i
        for j in range(i + 1, len(arrCopy)):
            comparisons += 1
            if arrCopy[j] < arrCopy[minIdx]:
                minIdx = j
        arrCopy[i], arrCopy[minIdx] = arrCopy[minIdx], arrCopy[i]

    kthValue = arrCopy[k - 1]
    # 在原数组中找到该值的位置
    originalIdx = arr.index(kthValue)
    return kthValue, originalIdx, comparisons


def findKthSmallestPreSort(arr, k):
    """
    预排序法找第k个最小元素（使用快速排序）
    时间复杂度：O(n log n)
```

```
    返回：(元素值, 位置索引, 比较次数)
    """
    if k < 1 or k > len(arr):
        return None, -1, 0

    # 使用快速排序
    sortedArr, comparisons = quickSort(arr)
    kthValue = sortedArr[k - 1]
    originalIdx = arr.index(kthValue)

    return kthValue, originalIdx, comparisons


def quickSort(arr):
    """
    快速排序算法
    时间复杂度：平均O(n log n)，最坏O(n^2)
    返回：(排序后的数组, 比较次数)
    """
    if len(arr) <= 1:
        return arr, 0

    comparisons = 0
    arrCopy = arr.copy()

    def partition(low, high):
        nonlocal comparisons
        pivot = arrCopy[high]
        i = low - 1

        for j in range(low, high):
            comparisons += 1
            if arrCopy[j] <= pivot:
                i += 1
                arrCopy[i], arrCopy[j] = arrCopy[j], arrCopy[i]

        arrCopy[i + 1], arrCopy[high] = arrCopy[high], arrCopy[i + 1]
        return i + 1

    def quickSortRecursive(low, high):
        nonlocal comparisons
        if low < high:
            pi = partition(low, high)
            quickSortRecursive(low, pi - 1)
            quickSortRecursive(pi + 1, high)

    quickSortRecursive(0, len(arrCopy) - 1)
    return arrCopy, comparisons


# ==================== GUI部分 ====================

class SearchGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("检索算法实验")
        self.root.geometry("900x750")
```

```
        self.currentArray = []

        # 数组生成区域
        arrayFrame = Frame(root)
        arrayFrame.pack(pady=10)

        Label(arrayFrame, text="数组长度n:", font=("Arial", 12)).pack(side=LEFT,
padx=5)
        self.nEntry = Entry(arrayFrame, width=15, font=("Arial", 12))
        self.nEntry.pack(side=LEFT, padx=5)

        Button(arrayFrame, text="随机生成数组", command=self.generateRandomArray,
               width=15, font=("Arial", 10)).pack(side=LEFT, padx=5)
        Button(arrayFrame, text="手动输入数组", command=self.inputArrayManually,
               width=15, font=("Arial", 10)).pack(side=LEFT, padx=5)
        Button(arrayFrame, text="快速排序", command=self.sortArray,
               width=15, font=("Arial", 10), bg="lightgreen").pack(side=LEFT,
padx=5)

        Label(arrayFrame, text="当前数组:", font=("Arial", 10)).pack(side=LEFT,
padx=5)
        self.arrayLabel = Label(arrayFrame, text="[]", font=("Arial", 10),
fg="blue")
        self.arrayLabel.pack(side=LEFT, padx=5)

        # 功能按钮区域
        buttonFrame1 = Frame(root)
        buttonFrame1.pack(pady=10)

        Button(buttonFrame1, text="功能3: 判断数组排序状态",
command=self.checkOrder,
               width=25, height=2, font=("Arial", 10)).pack(side=LEFT, padx=5)
        Button(buttonFrame1, text="功能4: 顺序检索",
command=self.sequentialSearchFunc,
               width=25, height=2, font=("Arial", 10)).pack(side=LEFT, padx=5)

        buttonFrame2 = Frame(root)
        buttonFrame2.pack(pady=10)

        Button(buttonFrame2, text="功能5: 多种方法检索",
command=self.multipleSearch,
               width=25, height=2, font=("Arial", 10)).pack(side=LEFT, padx=5)
        Button(buttonFrame2, text="功能6: 查找峰值", command=self.findPeak,
               width=25, height=2, font=("Arial", 10)).pack(side=LEFT, padx=5)

        buttonFrame3 = Frame(root)
        buttonFrame3.pack(pady=10)

        Button(buttonFrame3, text="功能7: 查找第k小元素",
command=self.findKthSmallest,
               width=25, height=2, font=("Arial", 10)).pack(side=LEFT, padx=5)

        # 输入框区域（用于功能4、5、7）
        inputFrame = Frame(root)
        inputFrame.pack(pady=10)

        Label(inputFrame, text="查找元素:", font=("Arial", 12)).pack(side=LEFT,
padx=5)
```

```
        self.targetEntry = Entry(inputFrame, width=15, font=("Arial", 12))
        self.targetEntry.pack(side=LEFT, padx=5)

        Label(inputFrame, text="k值:", font=("Arial", 12)).pack(side=LEFT,
padx=5)
        self.kEntry = Entry(inputFrame, width=15, font=("Arial", 12))
        self.kEntry.pack(side=LEFT, padx=5)

        # 输出文本框
        outputFrame = Frame(root)
        outputFrame.pack(pady=10, padx=10, fill=BOTH, expand=True)

        Label(outputFrame, text="输出结果:", font=("Arial", 12)).pack(anchor=W)
        self.outputText = scrolledtext.ScrolledText(outputFrame, width=90,
height=25,
                                                    font=("Consolas", 10))
        self.outputText.pack(fill=BOTH, expand=True)

        # 清空按钮
        Button(root, text="清空输出", command=self.clearOutput,
               width=15, font=("Arial", 10)).pack(pady=5)

    def clearOutput(self):
        """清空输出文本框"""
        self.outputText.delete(1.0, END)

    def appendOutput(self, text):
        """追加输出文本"""
        self.outputText.insert(END, text + "\n")
        self.outputText.see(END)
        self.root.update()

    def generateRandomArray(self):
        """随机生成数组"""
        try:
            n = int(self.nEntry.get())
            if n <= 0:
                messagebox.showerror("错误", "数组长度必须为正整数")
                return
            if n > 1000:
                messagebox.showwarning("警告", "数组长度过大，建议不超过1000")

            # 生成0到n*2范围内的n个不重复随机数
            maxVal = n * 2
            self.currentArray = random.sample(range(maxVal), n)

            self.arrayLabel.config(text=str(self.currentArray[:20]) + ("..." if
len(self.currentArray) > 20 else ""))
            self.appendOutput(f"\n{'='*60}")
            self.appendOutput(f"随机生成数组（长度={n}）")
            self.appendOutput(f"数组: {self.currentArray}")
            self.appendOutput("")
        except ValueError:
            messagebox.showerror("错误", "请输入有效的整数")
        except Exception as e:
            messagebox.showerror("错误", f"生成数组失败: {str(e)}")

    def inputArrayManually(self):
```

```
        """手动输入数组"""
        dialog = Toplevel(self.root)
        dialog.title("手动输入数组")
        dialog.geometry("500x200")

        Label(dialog, text="请输入数组元素（用逗号或空格分隔）:", font=("Arial",
12)).pack(pady=10)

        entry = Entry(dialog, width=60, font=("Arial", 12))
        entry.pack(pady=10)

        def confirm():
            try:
                text = entry.get().strip()
                # 支持逗号或空格分隔
                if ',' in text:
                    elements = [int(x.strip()) for x in text.split(',')]
                else:
                    elements = [int(x.strip()) for x in text.split()]

                # 检查是否有重复
                if len(elements) != len(set(elements)):
                    messagebox.showerror("错误", "数组元素必须互不相同")
                    return

                self.currentArray = elements
                self.nEntry.delete(0, END)
                self.nEntry.insert(0, str(len(elements)))
                self.arrayLabel.config(text=str(self.currentArray[:20]) + ("..."
if len(self.currentArray) > 20 else ""))
                self.appendOutput(f"\n{'='*60}")
                self.appendOutput(f"手动输入数组（长度={len(elements)}）")
                self.appendOutput(f"数组: {self.currentArray}")
                self.appendOutput("")
                dialog.destroy()
            except ValueError:
                messagebox.showerror("错误", "请输入有效的整数")
            except Exception as e:
                messagebox.showerror("错误", f"输入失败: {str(e)}")

        Button(dialog, text="确认", command=confirm, width=15, font=("Arial",
10)).pack(pady=10)

    def sortArray(self):
        """快速排序数组"""
        if not self.currentArray:
            messagebox.showwarning("警告", "请先生成或输入数组")
            return

        self.appendOutput(f"\n{'='*60}")
        self.appendOutput("快速排序")
        self.appendOutput(f"{'='*60}\n")

        self.appendOutput(f"排序前数组: {self.currentArray}")
        self.appendOutput(f"数组长度: {len(self.currentArray)}\n")

        startTime = time.time()
        sortedArray, comparisons = quickSort(self.currentArray)
```

```
        endTime = time.time()
        elapsedTime = (endTime - startTime) * 1000  # 转换为毫秒

        self.currentArray = sortedArray

        self.appendOutput(f"排序后数组: {sortedArray}")
        self.appendOutput(f"关键字比较次数: {comparisons}")
        self.appendOutput(f"执行时间: {elapsedTime:.6f} 毫秒")
        self.appendOutput("")

        # 更新显示
        self.arrayLabel.config(text=str(self.currentArray[:20]) + ("..." if
len(self.currentArray) > 20 else ""))

    def checkOrder(self):
        """功能3: 判断数组排序状态"""
        if not self.currentArray:
            messagebox.showwarning("警告", "请先生成或输入数组")
            return

        self.appendOutput(f"\n{'='*60}")
        self.appendOutput("功能3: 判断数组排序状态")
        self.appendOutput(f"{'='*60}\n")

        orderType = checkArrayOrder(self.currentArray)
        orderNames = {
            0: "未排序",
            1: "升序",
            2: "降序",
            3: "先升后降",
            4: "先降后升"
        }

        self.appendOutput(f"数组: {self.currentArray}")
        self.appendOutput(f"排序状态: {orderType} ({orderNames[orderType]})")
        self.appendOutput("")

    def sequentialSearchFunc(self):
        """功能4: 顺序检索"""
        if not self.currentArray:
            messagebox.showwarning("警告", "请先生成或输入数组")
            return

        try:
            target = int(self.targetEntry.get())
        except ValueError:
            messagebox.showerror("错误", "请输入有效的查找元素")
            return

        self.appendOutput(f"\n{'='*60}")
        self.appendOutput("功能4: 顺序检索算法")
        self.appendOutput(f"{'='*60}\n")

        self.appendOutput(f"数组: {self.currentArray}")
        self.appendOutput(f"查找元素: {target}\n")

        position, comparisons = sequentialSearch(self.currentArray, target)
```

```
        if position != -1:
            self.appendOutput(f"✓ 找到元素！")
            self.appendOutput(f"位置索引: {position}")
        else:
            self.appendOutput(f"✗ 未找到元素")
            self.appendOutput(f"位置索引: -1")

        self.appendOutput(f"关键字比较次数: {comparisons}")
        self.appendOutput("")

    def multipleSearch(self):
        """功能5: 多种方法检索（顺序、二分、三分）"""
        if not self.currentArray:
            messagebox.showwarning("警告", "请先生成或输入数组")
            return

        try:
            target = int(self.targetEntry.get())
        except ValueError:
            messagebox.showerror("错误", "请输入有效的查找元素")
            return

        originalOrderType = checkArrayOrder(self.currentArray)
        orderNames = ["未排序", "升序", "降序", "先升后降", "先降后升"]

        self.appendOutput(f"\n{'='*60}")
        self.appendOutput("功能5: 多种方法检索（顺序、二分、三分）")
        self.appendOutput(f"{'='*60}\n")

        self.appendOutput(f"原数组: {self.currentArray}")
        self.appendOutput(f"原数组状态: {orderNames[originalOrderType]}")
        self.appendOutput(f"查找元素: {target}\n")

        # 复制数组用于查找
        searchArray = self.currentArray.copy()
        sortComparisons = 0
        isSorted = False

        # 如果数组不是升序或降序，自动使用快排排序
        if originalOrderType not in [1, 2]:
            self.appendOutput(f"数组不是升序或降序，自动使用快速排序...")
            sortedArray, sortComparisons = quickSort(searchArray)
            searchArray = sortedArray
            isSorted = True
            self.appendOutput(f"排序完成，排序比较次数: {sortComparisons}")
            self.appendOutput(f"排序后数组: {searchArray}\n")
        elif originalOrderType == 2:
            # 如果是降序，反转数组用于查找
            searchArray = searchArray[::-1]
            self.appendOutput("注意: 数组为降序，已反转用于查找\n")

        # 顺序查找（在原数组上查找，不需要排序）
        pos1, comp1 = sequentialSearch(self.currentArray, target)

        self.appendOutput("【顺序查找】")
        if pos1 != -1:
            self.appendOutput(f"  找到! 位置索引: {pos1}")
        else:
```

```
            self.appendOutput(f"  未找到，位置索引: -1")
        self.appendOutput(f"  比较次数: {comp1}\n")

        # 二分查找（在排序后的数组上查找）
        pos2, comp2, closest2 = binarySearch(searchArray, target)
        # 映射位置索引到原数组
        if isSorted:
            # 排序后的位置，需要找到在原数组中的位置
            if pos2 != -1:
                originalPos2 = self.currentArray.index(searchArray[pos2])
            else:
                originalPos2 = -1
            if closest2:
                originalClosest2 = self.currentArray.index(closest2[0])
                closest2 = (closest2[0], originalClosest2)
        elif originalOrderType == 2:
            # 原数组是降序，反转后的位置需要转换
            if pos2 != -1:
                originalPos2 = len(searchArray) - 1 - pos2
            else:
                originalPos2 = -1
            if closest2:
                originalClosest2 = len(searchArray) - 1 - closest2[1]
                closest2 = (closest2[0], originalClosest2)
        else:  # 原数组是升序
            originalPos2 = pos2

        self.appendOutput("【二分查找】")
        if originalPos2 != -1:
            self.appendOutput(f"  找到！位置索引: {originalPos2} (原数组中的位置)")
        else:
            self.appendOutput(f"  未找到，位置索引: -1")
            if closest2:
                self.appendOutput(f"  最接近的元素: 值={closest2[0]}, 位置=
{closest2[1]} (原数组中的位置)")
        self.appendOutput(f"  查找比较次数: {comp2}")
        if sortComparisons > 0:
            self.appendOutput(f"  排序比较次数: {sortComparisons}")
            self.appendOutput(f"  总比较次数: {comp2 + sortComparisons}\n")
        else:
            self.appendOutput("")

        # 三分查找（在排序后的数组上查找）
        pos3, comp3, closest3 = ternarySearch(searchArray, target)
        # 映射位置索引到原数组
        if isSorted:
            # 排序后的位置，需要找到在原数组中的位置
            if pos3 != -1:
                originalPos3 = self.currentArray.index(searchArray[pos3])
            else:
                originalPos3 = -1
            if closest3:
                originalClosest3 = self.currentArray.index(closest3[0])
                closest3 = (closest3[0], originalClosest3)
        elif originalOrderType == 2:
            # 原数组是降序，反转后的位置需要转换
            if pos3 != -1:
                originalPos3 = len(searchArray) - 1 - pos3
```

```
            else:
                originalPos3 = -1
            if closest3:
                originalClosest3 = len(searchArray) - 1 - closest3[1]
                closest3 = (closest3[0], originalClosest3)
        else:  # 原数组是升序
            originalPos3 = pos3

        self.appendOutput("【三分查找】")
        if originalPos3 != -1:
            self.appendOutput(f"  找到! 位置索引: {originalPos3} (原数组中的位置)")
        else:
            self.appendOutput(f"  未找到，位置索引: -1")
            if closest3:
                self.appendOutput(f"  最接近的元素: 值={closest3[0]}, 位置=
{closest3[1]} (原数组中的位置)")
        self.appendOutput(f"  查找比较次数: {comp3}")
        if sortComparisons > 0:
            self.appendOutput(f"  排序比较次数: {sortComparisons}")
            self.appendOutput(f"  总比较次数: {comp3 + sortComparisons}\n")
        else:
            self.appendOutput("")

        self.appendOutput("")

    def findPeak(self):
        """功能6: 查找峰值（最大值或最小值）"""
        if not self.currentArray:
            messagebox.showwarning("警告", "请先生成或输入数组")
            return

        orderType = checkArrayOrder(self.currentArray)
        if orderType not in [3, 4]:
            messagebox.showwarning("警告", "此功能需要先升后降或先降后升数组，当前数组状
态为: " +
                                   ["未排序", "升序", "降序", "先升后降", "先降后升"]
[orderType])
            return

        self.appendOutput(f"\n{'='*60}")
        self.appendOutput("功能6: 查找峰值（二分和三分检索）")
        self.appendOutput(f"{'='*60}\n")

        self.appendOutput(f"数组: {self.currentArray}")
        self.appendOutput(f"数组状态: {'先升后降' if orderType == 3 else '先降后
升'}\n")

        findMax = (orderType == 3)  # 先升后降找最大值，先降后升找最小值

        # 二分查找峰值
        idx1, val1, comp1 = binarySearchPeak(self.currentArray, findMax)
        self.appendOutput("【二分检索】")
        self.appendOutput(f"  找到{'最大值' if findMax else '最小值'}: {val1}")
        self.appendOutput(f"  位置索引: {idx1}")
        self.appendOutput(f"  比较次数: {comp1}\n")

        # 三分查找峰值
        idx2, val2, comp2 = ternarySearchPeak(self.currentArray, findMax)
```

```
        self.appendOutput("【三分检索】")
        self.appendOutput(f"  找到{'最大值' if findMax else '最小值'}: {val2}")
        self.appendOutput(f"  位置索引: {idx2}")
        self.appendOutput(f"  比较次数: {comp2}\n")

        self.appendOutput("")

    def findKthSmallest(self):
        """功能7: 查找第k个最小元素"""
        if not self.currentArray:
            messagebox.showwarning("警告", "请先生成或输入数组")
            return

        try:
            k = int(self.kEntry.get())
            if k < 1 or k > len(self.currentArray):
                messagebox.showerror("错误", f"k值必须在1到
{len(self.currentArray)}之间")
                return
        except ValueError:
            messagebox.showerror("错误", "请输入有效的k值")
            return

        self.appendOutput(f"\n{'='*60}")
        self.appendOutput("功能7: 查找第k个最小元素（蛮力法和预排序）")
        self.appendOutput(f"{'='*60}\n")

        self.appendOutput(f"数组: {self.currentArray}")
        self.appendOutput(f"k = {k}\n")

        # 蛮力法
        val1, idx1, comp1 = findKthSmallestBruteForce(self.currentArray, k)
        self.appendOutput("【蛮力法】")
        self.appendOutput(f"  第{k}个最小元素: {val1}")
        self.appendOutput(f"  位置索引: {idx1}")
        self.appendOutput(f"  比较次数: {comp1}\n")

        # 预排序法（使用快速排序）
        val2, idx2, comp2 = findKthSmallestPreSort(self.currentArray, k)
        self.appendOutput("【预排序法（快速排序）】")
        self.appendOutput(f"  第{k}个最小元素: {val2}")
        self.appendOutput(f"  位置索引: {idx2}")
        self.appendOutput(f"  比较次数: {comp2}\n")

        if val1 == val2:
            self.appendOutput(f"✓ 两种方法结果一致: {val1}")
        else:
            self.appendOutput(f"⚠ 警告: 两种方法结果不一致! ")

        self.appendOutput("")


def main():
    root = Tk()
    app = SearchGUI(root)
    root.mainloop()
```

```
if __name__ == "__main__":
    main()
```