暨南大学本科上机课实验报告

学生姓名：肖健成　　学号：2021103268

学院：网络空间安全　　专业：网络空间安全

课程名称:《算法分析与设计》　　实验名称：　二分查找算法的改进

实验类型：验证-设计-综合　　实验时间：2023 年 11 月 3 日晚　　地点：N516

上机内容：2.3
（1）设计出二分查找算法的改进版；
（2）设计具有数据输入、处理和输出功能的二分查找算法的改进版，实现算法的编译和运行，记录实验过程并整理实验结果。

算法描述：

二分查找算法（Binary Search Algorithm）是一种在已排序数据集中查找目标元素的有效算法。它的核心思想是通过反复将数据集分成两半，并比较目标元素与中间元素的大小，从而确定目标元素在哪一半，然后继续在该半部分查找，直到找到目标元素或确定它不存在。

以下是二分查找算法的过程：

1.初始化两个指针 low 和 high，分别指向数据集的开头和结尾。
2.在每一轮迭代中，计算中间元素的索引 mid，通常通过 (low + high) // 2 计算。
3.比较中间元素与目标元素：
如果中间元素等于目标元素，搜索成功，返回中间元素的索引。
如果中间元素大于目标元素，说明目标元素位于左半部分，将 high 更新为 mid - 1，缩小搜索范围。
如果中间元素小于目标元素，说明目标元素位于右半部分，将 low 更新为 mid + 1，缩小搜索范围。
4.重复步骤 2 和 3，直到 low 大于 high，表示整个数据集已被搜索完毕，目标元素未找到，返回一个指定的未找到标记（通常为 -1）。

二分查找算法的优点在于它每一轮可以将搜索范围减半，因此其时间复杂度为 O(log n)，其中 n 是数据集的大小。这使得它在大型有序数据集中非常高效。然而，要求数据集必须是有序的，否则算法将无法正常工作。

下面是我使用的二分查找算法的三个改进点：

1. 使用位运算优化中点计算：

使用位运算来优化中点计算是一种常见的优化方法，特别是在涉及大量重复计算中点的算法中，如二分查找。中点计算通常涉及除法操作，而位运算则可以用来加速这个过程。在二分查找算法中，中点的计算通过 (low + high) // 2 来实现。这里我们将使

用位运算来代替除法操作。

中点的计算 (low + high) // 2 可以用右移操作 >> 来代替。右移操作将二进制数向右移动指定的位数，效果类似于除以 2 的 n 次方，其中 n 是右移的位数。

具体比较如下：

原始中点计算：mid = (low + high) // 2

位运算优化的中点计算：mid = low + ((high - low) >> 1)

这个优化的中点计算仍然保持了二分查找的核心思想，但通过避免除法操作来提高效率。这在某些情况下可能会显著提高算法的性能，尤其是在大规模数据集上。

2. 插值查找算法:

插值查找算法是一种改进的二分查找算法，其主要改进点在于中点的计算方式。插值查找算法使用了一种更智能的中点估算方法，通过根据目标值在数据范围内的相对位置来估计中点，从而更快地接近目标值。

插值查找算法的核心思想如下：

(1) 计算中点：不再使用固定的 (low + high) // 2 来计算中点，而是使用以下公式来估算中点：mid = low + (target - arr[low]) * (high - low) / (arr[high] - arr[low])

(2) 其中，target 是要查找的目标值，arr[low] 和 arr[high] 是范围内的最小和最大值。

(3) 比较中点与目标值：

如果中点处的元素等于目标值，查找成功，返回中点的索引。

如果中点处的元素小于目标值，说明目标值位于右半部分，将 low 更新为 mid + 1。

如果中点处的元素大于目标值，说明目标值位于左半部分，将 high 更新为 mid - 1。

重复步骤 1 和 2，直到 low 大于 high，表示整个数据集已被搜索完毕，目标元素未找到，返回一个指定的未找到标记（通常为 -1）。

插值查找算法的主要优点在于对有序数据集中等距离分布的数据效果良好。它倾向于更快地接近目标值，因此在某些情况下可能比二分查找更快。然而，对于不等距离分布的数据集，插值查找可能不如二分查找稳定。

3. 斐波那契查找算法是一种改进的二分查找算法，它使用黄金分割比例来确定划分点，以提高搜索效率。这个算法的核心思想是使用斐波那契数列来确定查找点，以使每一步都更接近目标元素，从而降低了查找时间。

以下是对斐波那契查找算法的分析：

1.斐波那契数列的生成：

首先，生成一个斐波那契数列，该数列中的每个元素都是前两个元素的和。通常，这个数列以 0 和 1 开始，然后依次生成下一个元素。

数列生成的终止条件是最大元素小于或等于要查找的数据集的长度。

2.计算斐波那契索引：

根据数据集的长度，找到斐波那契数列中不大于数据集长度的最大元素。这个元素的索引即为斐波那契索引。

通过斐波那契数列中的元素来确定查找点。

3.主要查找过程：

初始化 low 和 high 指针，分别指向数据集的开头和结尾。

计算查找点 mid，使用斐波那契数列中的元素，将 low 移动到 mid 处。

比较中点元素与目标元素：

如果中点元素等于目标元素，搜索成功，返回中点索引。

如果中点元素小于目标元素，将 low 更新为 mid + 1。

如果中点元素大于目标元素，将 high 更新为 mid - 1。

重复上述步骤，直到 low 大于 high，表示整个数据集已被搜索完毕，目标元素未找到，返回一个指定的未找到标记（通常为 -1）。

斐波那契查找算法的优点在于它不仅保留了二分查找的思想，还利用了斐波那契数列的黄金分割性质，可以更快地接近目标元素。这使得它在某些情况下比传统二分查找更高效。

下面第一个伪代码是使用位运算优化中点计算与插值查找算法的结合

```
function
binarySearchWithInterpolation(arr, target):
// 初始化搜索范围的下界和上界
low = 0
high = arr.length - 1

// 在搜索范围内继续查找
while low <= high:
    // 使用位运算计算中点位置
    mid = low + ((high - low) >> 1)

    // 使用插值查找算法估算中点位置
    // 这个估算会根据目标值在数组中的相对位置
    // 考虑已知的最小值和最大值来更精确地定位中点
    estimatedMid = low + ((high - low) * (target - arr[low])) / (arr[high] - arr[low])

    // 如果估算的中点位置超出搜索范围，将其调整为搜索范围的边界
    if estimatedMid < 0:
        mid = low
    elif estimatedMid > high:
        mid = high
    else:
        mid = estimatedMid

    // 如果中点元素与目标元素相等，返回中点的索引
    if arr[mid] == target:
        return mid

    // 如果中点元素小于目标元素，更新搜索范围的下界
    else if arr[mid] < target:
        low = mid + 1

    // 如果中点元素大于目标元素，更新搜索范围的上界
    else:
    high = mid - 1

// 如果未找到目标元素，返回 - 1
return -1
```

第二个伪代码是使用位运算优化中点计算与斐波那契查找算法的结合

```
function
binarySearchWithFibonacci(arr, target):
// 初始化斐波那契数列的初始元素
fibM_minus_2 = 0
fibM_minus_1 = 1
fibM = fibM_minus_1 + fibM_minus_2

// 计算斐波那契数列元素，直到
fibM
大于等于数组长度
while (fibM < arr.length):
    fibM_minus_2 = fibM_minus_1
    fibM_minus_1 = fibM
    fibM = fibM_minus_1 + fibM_minus_2

// 初始化搜索范围
low = 0
n = arr.length

while fibM > 1:
    // 计算当前分割点
    i = min(low + fibM_minus_2, n - 1)

    // 使用位运算计算中点
    mid = low + ((high - low) >> 1)

    // 如果中点元素等于目标元素，返回中点索引
    if arr[mid] == target:
        return mid
    // 如果中点元素小于目标元素，缩小搜索范围
    else if arr[mid] < target:
        low = mid + 1
        fibM = fibM_minus_1
        fibM_minus_1 = fibM_minus_2
        fibM_minus_2 = fibM - fibM_minus_1
    // 如果中点元素大于目标元素，缩小搜索范围
    else:
    high = mid - 1
    fibM = fibM_minus_2
    fibM_minus_1 = fibM_minus_1 - fibM_minus_2
    fibM_minus_2 = fibM - fibM_minus_1

// 检查剩下的元素
```

```
if (fibM_minus_1 == 1 and arr[low] == target):
      return low

// 如果未找到目标元素，返回 - 1
return -1
```

源程序：

使用位运算优化中点计算与插值查找算法的结合

```
def is_sorted(arr):
    for i in range(1, len(arr)):
        if arr[i - 1] > arr[i]:
            return False
    return True

def binary_search_with_interpolation(arr, target):
    # 在执行二分查找与插值查找之前，检查数组是否已排序
    if not is_sorted(arr):
        print("错误：数组不是升序的。")
        return -1

    # 初始化搜索范围的下界和上界
    low = 0
    high = len(arr) - 1

    # 在搜索范围内继续查找
    while low <= high:
        # 使用位运算计算中点位置
        mid = low + ((high - low) >> 1)

        # 使用插值查找算法估算中点位置
        estimated_mid = low + ((high - low) * (target - arr[low]))
// (arr[high] - arr[low])

        # 如果估算的中点位置超出搜索范围，将其调整为搜索范围的边界
        if estimated_mid < 0:
            mid = low
        elif estimated_mid > high:
            mid = high
        else:
            mid = estimated_mid

        # 如果中点元素与目标元素相等，返回中点的索引
        if arr[mid] == target:
```

```
            return mid

        # 如果中点元素小于目标元素，更新搜索范围的下界
        elif arr[mid] < target:
            low = mid + 1

        # 如果中点元素大于目标元素，更新搜索范围的上界
        else:
            high = mid - 1

    # 如果未找到目标元素，返回 -1
    return -1

# 获取用户输入的升序数组
input_str = input("输入升序数组，以逗号分隔：")
arr = [int(x) for x in input_str.split(",")]

# 获取用户输入的目标值
target = int(input("输入目标值："))

# 执行二分查找与插值查找
result = binary_search_with_interpolation(arr, target)
if result != -1:
    print(f"目标值 {target} 在数组中的索引是 {result}")
else:
    print(f"目标值 {target} 未在数组中找到")
```

使用位运算优化中点计算与斐波那契查找算法的结合

```
# 定义一个函数，用于检查数组是否以升序排列
def is_sorted(arr):
    for i in range(1, len(arr)):
        if arr[i - 1] > arr[i]:
            return False
    return True

# 定义一个函数，用于生成斐波那契数列，直到给定的数字 'n'
def generate_fibonacci(n):
    fibonacci = [0, 1]
    while fibonacci[-1] < n:
        next_value = fibonacci[-1] + fibonacci[-2]
        fibonacci.append(next_value)
    return fibonacci

# 定义一个函数，用于在排序数组 'arr' 中执行斐波那契搜索，寻找
'target' 值
```

```python
def fibonacci_search(arr, target):
    # 定义一个嵌套函数，用于找到大于或等于 'n' 的斐波那契数的索引
    def find_fibonacci_index(n):
        fibonacci = generate_fibonacci(n)
        for i in range(len(fibonacci)):
            if fibonacci[i] >= n:
                return i

    # 检查数组是否未排序，如果是，则打印错误消息
    if not is_sorted(arr):
        print("错误：数组不是升序的。")
        return -1

    # 初始化二分搜索的低位和高位指针
    low = 0
    high = len(arr) - 1
    n = len(arr)
    index = find_fibonacci_index(n)
    fibonacci = generate_fibonacci(n)

    # 执行斐波那契搜索
    while low <= high:
        mid = low + ((fibonacci[index - 1] - 1) >> 1)

        if mid < 0:
            mid = 0

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
            index -= 2
        else:
            high = mid - 1
            index -= 1

    # 如果未在数组中找到 'target'，则返回 -1
    return -1

# 获取用户输入的排序数组
input_str = input("输入升序数组，以逗号分隔：")
arr = [int(x) for x in input_str.split(",")]

# 获取用户输入的目标值
```

```
target = int(input("输入目标值："))

# 在执行斐波那契搜索之前，检查数组是否已排序
if is_sorted(arr):
    result = fibonacci_search(arr, target)
    if result != -1:
        print(f"目标值 {target} 在数组中的索引是 {result}")
    else:
        print(f"目标值 {target} 未在数组中找到")
else:
    print("错误：数组不是升序的。")
```

复杂度分析：
使用位运算优化中点计算与插值查找算法的结合：
显然，位运算优化重点计算不对复杂度有影响，仅仅只是对运算速度进行了优化，我们接下来不分析这部分。
以下是对插值查找的二分查找算法的复杂度分析：
1.最好情况时间复杂度：最好的情况是目标值在数组的中心，这时插值查找的二分查找表现得像二分查找。此时，时间复杂度为 O(log n)，其中 n 是数组的大小。
2.最坏情况时间复杂度：最坏的情况是插值查找的二分查找每次都在数组的一个极端位置，导致搜索范围不断缩小，但速度非常慢。在最坏情况下，时间复杂度可能接近 O(n)，但通常仍然为 O(log n)。
3.平均情况时间复杂度：在平均情况下，使用插值查找的二分查找通常具有比二分查找稍好的性能。但它的性能高度依赖于数据分布。如果数据分布均匀，插值查找通常能够更快地找到目标值。然而，如果数据不均匀分布，插值查找的二分查找的性能可能不如二分查找。
4.空间复杂度：因为我们使用的是循环而不是递归实现，插值查找的二分查找算法的空间复杂度通常为 O(1)，因为它只需要一些额外的变量来存储中间计算结果。
使用位运算优化中点计算与斐波那契查找算法的结合：
1.最好情况时间复杂度：最好的情况是目标值在数组的中间位置，此时斐波那契查找的二分查找算法的性能接近二分查找。因此，最好情况的时间复杂度为 O(log n)，其中 n 是数组的大小。
2.最坏情况时间复杂度：斐波那契查找的二分查找算法的最坏情况时间复杂度也为 O(log n)。这是因为斐波那契数列的性质确保了搜索范围按黄金分割比例逐渐缩小，不会出现线性搜索的情况。
3.平均情况时间复杂度：与最坏情况时间复杂度相同，斐波那契查找的二分查找的平均

情况时间复杂度为 O(log n)。这是由于查找范围以黄金分割比例缩小，而斐波那契数列的增长速度刚好满足这一要求。
4.空间复杂度：斐波那契查找的二分查找算法的空间复杂度通常为 O(1)，因为它只需要一些额外的变量来存储中间计算结果。

插值查找时间复杂度	插值查找空间复杂度	斐波那契查找时间复杂度	斐波那契查找空间复杂度
O(log n)	O(1)	O(log n)	O(1)

程序的运行结果及截图：
使用位运算优化中点计算与插值查找算法的结合：

```
输入升序数组，以逗号分隔：1,2,3,4,5
输入目标值：3
目标值 3 在数组中的索引是 2
```

```
输入升序数组，以逗号分隔：1,2,3,4,5
输入目标值：6
目标值 6 未在数组中找到
```

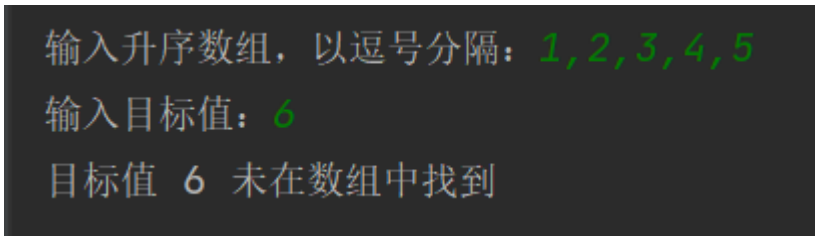
正确输入升序数组的结果

```
输入升序数组，以逗号分隔：1,3,2,5
输入目标值：3
错误：数组不是升序的。
```

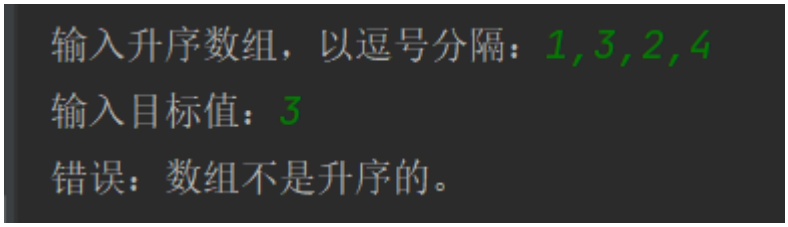
未输入升序数组的结果

使用位运算优化中点计算与斐波那契查找算法的结合：

```
输入升序数组，以逗号分隔：1,2,3,4,5
输入目标值：3
目标值 3 在数组中的索引是 2
```



正确输入升序数组的结果



未输入升序数组的结果

个人总结：

二分查找算法的核心思想是在有序数组中查找特定元素的位置，通过不断缩小搜索范围来提高搜索效率。以下是二分查找算法的核心步骤：

1.初始化：首先，确定要搜索的数组的左边界（low）和右边界（high）。通常，初始的左边界是数组的第一个元素的索引，初始的右边界是数组的最后一个元素的索引。

2.循环：在一个循环中，进行以下操作，直到找到目标元素或确定其不存在：

计算中点位置（mid）：通过取左边界（low）和右边界（high）的平均值来计算中点位置。这可以使用 (low + high) // 2 来实现。

比较中点元素：将目标元素与中点位置的元素进行比较。

如果目标元素等于中点元素，则找到了目标，返回中点的索引。

如果目标元素小于中点元素，说明目标元素位于左半部分，将右边界 (high) 更新为 mid - 1。

如果目标元素大于中点元素，说明目标元素位于右半部分，将左边界 (low) 更新为 mid + 1。

3.循环终止条件：循环在以下情况之一终止：

目标元素被找到，返回其索引。

左边界（low）大于右边界（high），表示目标元素不存在于数组中，返回 -1 或其他指定的值。

这个算法的核心思想是将搜索范围在每次循环迭代中减半，因此其时间复杂度通常为 O(log n)，其中 n 是数组的大小。这使得二分查找成为一种高效的搜索算法，尤其适用于有序数组或列表中查找目标元素的情况。

位运算优化中点计算：对运算速度进行了优化。

插值查找算法：

插值查找算法是一种改进的二分查找算法，其核心思想是根据目标元素在有序数组中的估计位置来提高搜索效率。与标准二分查找不同，插值查找会根据目标元素的估计位置来选择下一步搜索的范围。下面是插值查找算法的核心步骤：

1.初始化：确定要搜索的有序数组的左边界（low）和右边界（high），通常初始化为数组的第一个元素和最后一个元素的索引。

2.插值计算：计算目标元素在数组中的估计位置（mid）。插值查找使用以下公式来估计位置：mid = low + (target - arr[low]) * (high - low) / (arr[high] - arr[low])
3. 比较中点元素：将目标元素与估计位置（mid）的元素进行比较。
如果目标元素等于中点元素，则找到了目标，返回中点的索引。
如果目标元素小于中点元素，说明目标元素位于左半部分，将右边界（high）更新为 mid - 1。
如果目标元素大于中点元素，说明目标元素位于右半部分，将左边界（low）更新为 mid + 1。
4.循环终止条件：循环在以下情况之一终止：
目标元素被找到，返回其索引。
左边界（low）大于右边界（high），表示目标元素不存在于数组中，返回 -1 或其他指定的值。
插值查找算法的优势在于它根据目标元素的估计位置来快速缩小搜索范围。
斐波那契查找算法：

斐波那契查找算法是一种改进的二分查找算法，它使用斐波那契数列来确定搜索范围，以提高搜索效率。以下是斐波那契查找算法的核心步骤：
1.初始化：确定要搜索的有序数组的左边界（low）和右边界（high）。斐波那契查找算法需要使用斐波那契数列来确定搜索范围。初始化时，通过找到最小的斐波那契数（F[k]）满足 F[k] >= n，其中 n 是数组的大小，然后将右边界初始化为 F[k] - 1。
2.斐波那契数列计算：斐波那契查找会使用斐波那契数列中的前两个数（F[k-1]和 F[k-2]）来计算搜索位置。初始时，k 被设置为找到的斐波那契数 F[k] 的索引。
3.比较中点元素：在一个循环中，进行以下操作，直到找到目标元素或确定其不存在：
使用斐波那契数列计算中点位置（mid）：mid = low + F[k-1] - 1
比较中点元素和目标元素：
如果目标元素等于中点元素，则找到了目标，返回中点的索引。
如果目标元素小于中点元素，说明目标元素位于左半部分，将右边界（high）更新为 mid - 1，并将 k 递减 1。
如果目标元素大于中点元素，说明目标元素位于右半部分，将左边界（low）更新为 mid + 1，并将 k 递减 2。
4.循环终止条件：循环在以下情况之一终止：
目标元素被找到，返回其索引。
左边界（low）大于右边界（high），表示目标元素不存在于数组中，返回 -1 或其他指定的值。

斐波那契查找算法的优势在于它利用斐波那契数列的性质来动态确定搜索范围，可以更快地找到目标元素。

在这次的实验中，我加深了对二分查找算法的理解，加强了从理论到实践的能力。