查找算法

在LeetCode刷题或者面试过程中发现，查找问题一直是不可避免的。对任何数据结构的遍历过程无非就是查找过程。

我们需要针对某些数据结构的特点如何正确地、高效地进行查找，而查找的过程最需要注意边界控制。

下面以二分查找为例。

1. 二分查找★★☆

目的：在一个含有N个元素的有序数组中有效地的定位目标值。

思想：假设在有序数组arr中查找元素k，返回k所在的下标（索引值）。设`arr[low,high]`是当前的查找区间，确定该区间的中间位置$mid = \lfloor (low + high)//2 \rfloor$，然后将待查的k值与arr[mid]比较:

- 若`k==arr[mid]`，说明找到k，则查找成功并且终止。
- 若`k<arr[mid]`，根据数组有序的前提，目标值k在左边的区域中，索引的范围改为[low, mid-1]
- 若`k>arr[mid]`，目标值在右边的区域中，查找索引范围改为[mid+1, high]。

时间复杂度：$O(log_2 n)$

代码实现

```
# -*- coding: utf-8 -*-
# @Time      : 2020-04-11 23:28
# @Author    : yuzhou_1su
# @ContactMe : https://blog.csdn.net/yuzhou_1shu
# @File      : Binary_Search.py
# @Software  : PyCharm


def binary_search1(arr, item):
    """
    二分查找的非递归实现1

```

```
    :param arr: 有序数组
    :param item: 待查元素
    :return: 找到待查元素的所有；如果找不到，则返回None
    """
    low = 0
    high = len(arr) - 1  # 注意此处，high索引能取到

    while low <= high:  # 条件是low<=high，区间中没有元素时结束
        mid = (low + high) // 2
        curr_item = arr[mid]

        if curr_item == item:
            return mid
        elif item < curr_item:
            high = mid - 1  # high = mid - 1
        else:
            low = mid + 1
    return None


def binary_search2(arr, item):
    """
    左边界为n的二分查找

    :param arr: 给定一个有序数组
    :param item: 待查找的元素
    :return: 找到待查元素的所有；如果找不到，则返回None
    """

    low = 0
    high = len(arr)  # 此处 high的索引不能取到
    while low < high:  # 条件是low<high,区间中有一个元素时也结束
        mid = (low + high) // 2
        if arr[mid] == item:
            return mid
        elif item < arr[mid]:
            high = mid  # high = mid
        else:
            low = mid + 1

    return None


def binary_search3_by_recursion(arr, item, low, high):
    """
    二分查找的递归实现
```

```
    :param arr: 给定一个有序数组
    :param item: 待查找的元素
    :param low: 左边界
    :param high: 右边界
    :return: 找到待查元素的所有；如果找不到，则返回None
    """

    # 递归终止条件
    if low > high:
        return None
    mid = low + (high - low) // 2
    if arr[mid] == item:
        return mid
    elif arr[mid] > item:
        return binary_search3_by_recursion(arr, item, low, mid-1)
    else:
        return binary_search3_by_recursion(arr, item, mid+1, high)
```

> 二分查找边界问题探讨：*二分查找有几种写法?*

2. 顺序查找★☆☆

如果数组无序的话，只能通过循环遍历进行查找。

时间复杂度：$O(n)$

```
# -*- coding: utf-8 -*-
# @Time      : 2020-09-09 18:06
# @Author    : yuzhou_1su
# @ContactMe : https://blog.csdn.net/yuzhou_1shu
# @File      : linear_search.py
# @Software  : PyCharm


def linear_search(sequence, target):
    """线性查找
    :param sequence: 待查找序列，可以无序
    :param target: 待查元素
    :return: 找到待查元素的所有；如果找不到，则返回None
    """

    for i, v in enumerate(sequence):
        if v == target:
```

```
            return i
    return None

```

3. 索引查找

增加一个索引表，索引表的每一项称为索引项，索引项的一般形式：`(Key, Value)`。

索引查找的过程是：先在索引表中快速查找（索引表中可以按关键字有序排序，例如采用二分查找），找到关键字，然后通过对应的地址找到主数据表中的元素。

分块查找是一种典型的索引查找，其性能介于顺序查找和二分查找之间。

4. 高频面试题：

4.1 Top K问题？

4.2 10w条数据如何找出前一百条频繁数据？

排序算法

一般排序算法最常考的：快速排序和归并排序。这两个算法体现了分治算法的核心观点，而且还有很多出题的可能。

> *更多细节请参考刘宇波老师的：不能白板编程红黑树就是基础差？别扯了。*

1. 常见的排序算法

排序算法很多，除了能写出常见排序算法的代码，还需要了解各种排序的时空复杂度、稳定性、使用场景、区别等。

1.1 选择排序

1.1.1 思想

对于给定的一组序列，第一轮比较选择最小（或最大）的值，然后将该值与索引第一个进行交换；接着对不包括第一个确定的值进行第二次比较，选择第二个记录与索引第二个位置进行交换，重复到只剩最后一个记录位置。

> *案例：幼儿园排队，老师先让站成一队，带第一个小朋友依此跟其他小朋友逐个比较，选出个子最矮的，然后依此进行*

1.1.2 实现

```
def selection_sort(gList):
    """选择排序
    :param gList: 给定的一组序列
    :return: 返回排好序的序列
    """
    length = len(gList)
    for i in range(length - 1):
        flag = i
        for j in range(i+1, length):
            if gList[flag] > gList[j]:
                flag = j
        # 如果最小值的索引与最小值相对应，则无需再次交换
        if flag != i:
            gList[flag], gList[i] = gList[i], gList[flag]

    return gList

```

1.1.3 选择排序分析

- 时间复杂度：最好、最坏、平均的时间复杂度都为$O(n^2)$
- 空间复杂度: $O(1)$
- 稳定性：不稳定

1.2 冒泡排序

1.2.1 思想

对于给定的一组序列含n个元素，从第一个开始对相邻的两个记录进行比较，当前面的记录大于后面的记录，交换其位置，进行一轮比较和换位之后，最大记录在第n个位置；然后对前（n-1）个记录进行第二轮比较；重复该过程直到进行比较的记录只剩下一个时为止。

> *案例：冒泡，像气泡一样往上升*

1.2.2 实现

```python
def bubble_sort(gList):
    """冒泡排序"""
    length = len(gList)
    for i in range(length):
        for j in range(i+1, length):
            if gList[i] > gList[j]:
                gList[i], gList[j] = gList[j], gList[i]
    return gList
```

1.2.3 冒泡排序分析

- 时间复杂度：
    - 最好时间复杂度：$O(n)$
    - 最坏时间复杂度：$O(n^2)$
    - 平均时间复杂度：$O(n^2)$
- 空间复杂度：$O(1)$
- 稳定性：稳定的排序

1.3 插入排序

1.3.1 思想

对于给定的一组记录，初始时假设第一个记录自成一个有序序列，其余的记录为无序序列；接着从第二个记录开始，按照记录的大小依次将当前处理的记录插入到其之前的有序序列中，直至最后一个记录插入到有序序列中为止。

> *案例：抓扑克牌*

1.3.2 实现

```python
def insertion_sort(gList):
    """插入排序"""
    length = len(gList)
    for i in range(1, length):
        temp = gList[i]  # 当前的待插入的值
        j = i - 1  # 前一个值
        while j >= 0:
            if gList[j] > temp:
                gList[j+1] = gList[j]  # 插入的动作
                gList[j] = temp  # 插入完毕
            j -= 1
    return gList
```

1.3.3 插入排序分析

- 时间复杂度
    - 最好时间复杂度：$O(n)$
    - 最坏时间复杂度: $O(n^2)$
    - 平均时间复杂度: $O(n^2)$
- 空间复杂度: $O(1)$
- 稳定性：稳定的排序

1.4 归并排序 ☆☆★

1.4.1 思想

利用递归与分治技术将数据序列划分成为越来越小的半子表，再对半子表排序，最后再用递归步骤将排好序的半子表合并成为越来越大的有序序列。其中“归”代表的是递归的意思，即递归地将数组折半地分离为单个数组。

给定一组序列含n个元素，首先将每两个相邻的长度为1的子序列进行归并，得到n/2（向上取整）个长度为2或1的有序子序列，再将其两两归并，反复执行此过程，直到得到一个有序序列为止。

1.4.2 实现

```python
def merge_sort(gList: list) -> list:
    """归并排序
    :param gList: 给定序列
    :return: 升序排列后的集合
    """

    def merge(left: list, right: list) -> list:
        """merge left and right
        :param left: left list
        :param right: right list
        :return: merge reslut
        """
        i, j = 0, 0
        result = []
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result += left[i:]
        result += right[j:]
```

```
        return result

    if len(gList) <= 1:
        return gList
    num = len(gList) // 2
    left = merge_sort(gList[:num])
    right = merge_sort(gList[num:])
    return merge(left, right)


if __name__ == '__main__':
    gList = [3, 5, 2, 4, 1]
    print("----排序前:", gList)
    print("----归并排序后: ", merge_sort(gList))
```

1.4.3 归并排序分析

- 时间复杂度: 最好、最坏和平均情况$O(nlogn)$
- 空间复杂度: $O(n)$
- 稳定性:稳定

> *题目：100个有序数列如何合成一个大数组?*

1.5 快速排序☆★★

1.5.1 思想

高效的排序算法，它采用“分而治之”的思想，把大的拆分为小的，小的再拆分为更小的。其原理是：对于一组给定的记录，通过一趟排序后，将原序列分为两部分，其中前部分的所有记录均比后部分的所有记录小，然后再依次对前后两部分的记录进行快速排序，递归该过程，直到序列中的所有记录均有序为止。

1.5.2 实现

```
# -*- coding: utf-8 -*-

def quick_sort(gList, left=0, right=None) -> list:
    """快速排序
    :param gList: 给定一组序列
    :param left:
    :param right:
    :return: 升序排序后的序列
    """
```

```
    if right is None:
        right = len(gList)-1

    if left > right:
        return gList

    key = gList[left]
    low = left
    high = right

    while left < right:
        while left < right and gList[right] >= key:
            right -= 1
        gList[left] = gList[right]

        while left < right and gList[left] <= key:
            left += 1
        gList[right] = gList[left]
    gList[right] = key
    quick_sort(gList, low, left-1)
    quick_sort(gList, left+1, high)
    return gList


if __name__ == '__main__':
    gList = [3, 5, 2, 4, 1, 6, 7]
    print("----排序前:", gList)
    print("----快速排序后: ", quick_sort(gList))

```

1.5.3 快速排序分析

- 时间复杂度:
    - 最坏时间复杂度: $O(n^2)$
    - 最好时间复杂度: $O(nlogn)$
    - 平均时间复杂度: $O(nlogn)$
- 空间复杂度: $O(logn)$
- 稳定性: 不稳定

> *扩展: 随机快排*

1.6 希尔排序

1.6.1 思想

希尔排序也称为“缩小增量排序”。它的基本原理是：首先将待排序的元素分成多个子序列，使得每个子序列的元素个数相对较少，对各个子序列分别进行直接插入排序，待整个待排序序列“基本有序后”，再对所有元素进行一次直接插入排序。

1.6.2 实现

```python
# -*- coding: utf-8 -*-
def shell_sort(gList) -> list:
    """希尔排序"""
    length = len(gList)
    step = 2
    group = length // step
    while group > 0:
        for startPos in range(group):
            gap_insertion_sort(gList, startPos, group)
        group = group // 2
    return gList


def gap_insertion_sort(gList, start, gap):
    for i in range(start+gap, len(gList), gap):
        curr_value = gList[i]
        pos = i

        while pos >= gap and gList[pos-gap] > curr_value:
            gList[pos] = gList[pos-gap]
            pos = pos - gap
        gList[pos] = curr_value


if __name__ == '__main__':
    gList = [5, 4, 2, 1, 7, 3, 6]
    print("-----yuzhoulsu-----", gList)
    print("-----希尔排序后:", shell_sort(gList))
```

1.6.3 希尔排序分析

- 时间复杂度：
    - 最好时间复杂度：$O(n)$
    - 最坏时间复杂度：$O(n^s)(1<s<2)$
    - 平均时间复杂度：$O(nlogn)$
- 空间复杂度:$O(1)$

- 稳定性: 不稳定

1.7 堆排序

> *堆是一种特殊的树形数据结构，其每个结点都有一个值，通常提到的堆都是指一棵完全二叉树，根结点的值小于（或大于）两个子结点的值，同时根结点的两个子树也分别是一个堆。*

1.7.1 算法思想:

对于给定的序列，初始把这些记录看成一刻顺序存储的二叉树，然后将其调整为一个大顶堆，然后将堆的最后一个元素与堆顶元素进行交换后，堆的最后一个元素即为最大记录；接着将前(n-1)个元素重新调整为一个大顶堆，在将堆顶元素与当前堆的最后一个元素进行交换后得到次大的记录，重复该过程直到调整的堆中只剩一个元素为止，该记录即为最小记录，此时可得到一个有序序列。

过程：1. 构建堆；2. 交换堆顶元素与最后一个元素的位置

1.7.2 实现

```
def heapify(unsorted, index, heap_size):
    largest = index
    left_index = 2 * index + 1
    right_index = 2 * index + 2
    if left_index < heap_size and unsorted[left_index] > unsorted[largest]:
        largest = left_index

    if right_index < heap_size and unsorted[right_index] > unsorted[largest]:
        largest = right_index

    if largest != index:
        unsorted[largest], unsorted[index] = unsorted[index], unsorted[largest]
        heapify(unsorted, largest, heap_size)


def heap_sort(unsorted):
    """堆排序"""
    length = len(unsorted)
    for i in range(length // 2 - 1, -1, -1):
        heapify(unsorted, i, length)
    for i in range(length - 1, 0, -1):
        unsorted[0], unsorted[i] = unsorted[i], unsorted[0]
        heapify(unsorted, 0, i)
```

```
    return unsorted


if __name__ == '__main__':
    gList = [5, 4, 2, 1, 7, 3, 6]
    print("-----yuzhou1su-----", gList)
    print("-----堆排序后:", heap_sort(gList))

```

1.7.3 堆排序分析

时间复杂度：主要耗费在创建堆和反复调整堆上，最坏情况下，时间复杂度也为$O(nlogn)$

稳定性：不稳定

1.8 计数排序

> *推荐文章: https://www.cnblogs.com/xiaochuan94/p/11198610.html*

1.8.1 算法思想

对于某种整数K，计数排序假定每个元素都是1到K范围内的整数。 计数排序的基本思想是为每个输入元素x确定小于x的元素数量， 此信息可用于直接将其放置在正确的位置。例如，如果10个元素小于x，则x属于输出中的位置11。

1.8.2 实现

```
# -*- coding: utf-8 -*-
# @Time      : 2020-09-10 14:31
# @Author    : yuzhou_1su
# @ContactMe : https://blog.csdn.net/yuzhou_1shu
# @File      : counting_sort.py
# @Software  : PyCharm


def counting_sort(unsorted):
    """计数排序
    :param unsorted: 给定一组序列
    :return: 升序序列
    """
    if unsorted is []:
        return []
    # 根据给定序列求信息
    coll_len = len(unsorted)

```

```
    coll_max = max(unsorted)
    coll_min = min(unsorted)

    # 创建计数数组
    counting_arr_length = coll_max + 1 - coll_min
    counting_arr = [0] * counting_arr_length

    # 计数操作
    for number in unsorted:
        counting_arr[number - coll_min] += 1

    # 将每个位置与它的前一个相加。counting_arr[i]统计出多少个
    # element <= i的元素
    for i in range(1, counting_arr_length):
        counting_arr[i] = counting_arr[i] + counting_arr[i - 1]

    # 创建保存升序结果的数组
    ordered = [0] * coll_len
    for i in reversed(range(0, coll_len)):
        ordered[counting_arr[unsorted[i] - coll_min] - 1] = unsorted[i]
        counting_arr[unsorted[i] - coll_min] -= 1

    return ordered


if __name__ == '__main__':
    gList = [5, 4, 2, 1, 3, 6]
    print("-----yuzhou1su: ", gList)
    print("-----计数排序后:", counting_sort(gList))
```

1.8.3 计数排序分析

时间复杂度: $O(n) \quad if\ K = O(n)$

空间复杂度: $O(n) \quad if\ K = O(n)$

> *Ps: 如果K特别大，时间复杂度会很高；如果面试官让你设计数据规模小的线性排序算法，可能就是考察计数排序*

1.9 桶排序

1.9.1 算法思想

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。为了使桶排序更加高效，我们需要做到这两点：

1. 在额外空间充足的情况下，尽量增大桶的数量
2. 使用的映射函数能够将输入的 N 个数据均匀的分配到 K 个桶中

> *菜鸟教程：桶排序*

1.9.2 实现

```python
# -*- coding: utf-8 -*-
# @Time      : 2020-09-10 15:30
# @Author    : yuzhou_1su
# @ContactMe : www.yuzhou_1su@163.com
# @File      : bucket_sort.py
# @Software  : PyCharm
import math


def insertion_sort(collection):
    for i in range(1, len(collection)):
        temp = collection[i]
        index = i
        while index > 0 and temp < collection[index - 1]:
            collection[index] = collection[index-1]
            index -= 1
        collection[index] = temp


def bucket_sort(collection):
    code = hashing(collection)
    buckets = [list() for _ in range(code[1])]
    for i in collection:
        x = rehashing(i, code)
        buck = buckets[x]
        buck.append(i)

    for bucket in buckets:
        insertion_sort(bucket)

    ndx = 0
    for buc in range(len(buckets)):
        for val in buckets[buc]:
```

```
            collection[ndx] = val
            ndx += 1
    return collection


def hashing(collection):
    m = collection[0]
    for i in range(1, len(collection)):
        if m < collection[i]:
            m = collection[i]
    result = [m, int(math.sqrt(len(collection)))]
    return result


def rehashing(i, code):
    return int(i / code[0] * (code[1] - 1))


if __name__ == '__main__':
    gList = [5, 4, 2, 1, 3, 6]
    print("-----yuzhou1su: ", gList)
    print("-----桶排序后:", bucket_sort(gList))
```

1.9.3 桶排序分析

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

1.10 基数排序

1.10.1 算法思想

与计数排序/桶排序类似，基数排序跟输入元素相关。比如：根据基数d对给定序列进行排序，这意味着所有的数字都是d位数。过程：

1. 取每个元素的最低有效位
2. 根据该数字对元素列表进行排序，但保持相同数字的元素顺序
3. 用更高有效位重复排序，直到最高位

1.10.2 实现

```python
def radix_sort(unsorted):
    radix = 10
    max_len = False
    tmp, placement = -1, 1
    while not max_len:
        max_len = True
        buckets = [list() for _ in range(radix)]
        for i in unsorted:
            tmp = int(i / placement)
            buckets[tmp % radix].append(i)
            if max_len and tmp > 0:
                max_len = False
        a = 0
        for b in range(radix):
            buck = buckets[b]
            for i in buck:
                unsorted[a] = i
                a += 1
        # move to next digit
        placement *= radix
    return unsorted


if __name__ == '__main__':
    gList = [5, 4, 2, 1, 3, 6]
    print("-----yuzhou1su: ", gList)
    print("-----基数排序后:", radix_sort(gList))

```

1.10.3 基数排序分析

基数排序适用于位数小的数字序列。

- 时间复杂度: $O(nlog(r)\ m)$，其中r为所采取的基数，而m为堆数
- 稳定性：稳定

1.11 其他排序

- 拓扑排序：在一个有向图中，对所有的节点进行排序，要求没有一个节点指向它前面的节点。
- 外部排序：大文件的排序，即待排序的记录存储在外存储器上，待排序的文件无法一次装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到排序整个文件的目的。

- 位图排序：当待排序数据规模较大，而堆内存大小又没有限制时，位图排序则最高效。
- **Tim-sort**：Python的list标准排序算法，由Tim Peters设计。本质上是一种自下而上的归并排序，利用一些数据的初始运行，之后进行额外的插入排序。Tim-sort也成为Java7中数组排序的默认算法。

2. 各种排序算法比较?

序号	排序方法	平均时间复杂度	最好	最坏	辅助存储	是否稳定	备注
1	选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	稳定性取决于实现，n小性能好
2	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	n小性能好
3	插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	大部分已有序较好
4	归并排序	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	稳定	n大性能好
5	快速排序	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$	不稳定	n大性能好
6	希尔排序	$O(nlogn)$	$O(n)$	$O(n^s)$	$O(1)$	不稳定	s是所选分组
7	堆排序	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	不稳定	n大性能好
8	计数排序	$O(n)$			$O(n)$	稳定	整数较小情况下，性能较好
9	桶排序	$O(n)$			$O(n)$	稳定	n小性能好
10	基数排序	$O(nlog(r)m)$			$O(nm)$	稳定	r为所采取的基数，而m为堆数

根据上图总结：

- 不稳定算法有：选择、快速、希尔、堆

  > *记忆口诀：快选七(希)堆不稳定*

- 时间复杂度$O(n^2)$：选择、冒泡、插入
- 时间复杂度$O(nlogn)$：快速、归并、堆、希尔
- 时间复杂度$O(n)$：计数、桶
- 空间复杂度$O(1)$：选择、插入、冒泡、希尔、堆
- 空间复杂度$O(n)$：归并、计数、桶
- 空间复杂度$O(logn)$：快速排序

3.总结

一定要根据数据的规模、规律来给出合适的算法，不能觉得快速排序名字就以为是快速的，切记不能什么排序问题都回答快排。

1. 虽然插入排序和冒泡排序平均速度较慢，但当初始序列整体或局部有序时，这两者效率较高
2. 排序数据较小，且不要求稳定的情况下，选择排序效率较高；要求稳定，选择冒泡排序。
3. 堆排序在更大的序列上往往优于快速排序和归并排序。
4. 针对小数据追求线性时间复杂度，考虑计数排序和桶排序
5. 除了上面几种常见的排序算法，还有众多其他排序算法，每种排序算法都有其最佳适用场合。具体情况具体分析。

最后，感谢大家阅读。我是`yuzhou_1su`，一个头发比想法多的研究僧。

如果觉得文章还不错，请一定帮忙点个赞。谢谢🤝

> *参考资料：*
>
>
> - *《数据结构与算法：python语言实现》克尔.·T·古德里奇 / 罗伯托·塔玛西亚 / 迈克尔·H·戈德瓦瑟等著*
> - *《Python程序员面试算法宝典》张波 楚秦等编著*
> - *TheAlgorithms/Python/sorts*
>