Leetcode python

堆

347. 前 K 个高频元素

堆排序处理海量数据的topK，分位数 非常合适，因为它不用将所有的元素都进行排序，只需要比较和根节点的大小关系就可以了，同时也不需要一次性将所有的数据都加载到内存。

因此有必要不引入库，自己用python实现研究一下

原则：最大堆求前n小，最小堆求前n大。

- 前k小：构建一个k个数的最大堆，当读取的数小于根节点时，替换根节点，重新塑造最大堆
- 前k大：构建一个k个数的最小堆，当读取的数大于根节点时，替换根节点，重新塑造最小堆

总体思路

- 建立字典遍历一次统计出现频率
- 取前k个数，构造**规模为k的最小堆** minheap
- 遍历规模k之外的数据，大于堆顶则入堆，维护规模为k的最小堆 minheap
- 如需按频率输出，对规模为k的堆进行排序

```
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        def heapify(arr, n, i):
            smallest = i  # 构造根节点与左右子节点
            l = 2 * i + 1
            r = 2 * i + 2
            if l < n and arr[l][1] < arr[i][1]:  # 如果左子节点在范围内且小于父节点
                smallest = l
            if r < n and arr[r][1] < arr[smallest][1]:
                smallest = r
            if smallest != i:  # 递归基:如果没有交换，退出递归
                arr[i], arr[smallest] = arr[smallest], arr[i]
                heapify(arr, n, smallest)  # 确保交换后，小于其左右子节点

        # 哈希字典统计出现频率
        map_dict = {}
        for item in nums:
            if item not in map_dict.keys():
                map_dict[item] = 1
            else:
                map_dict[item] += 1

        map_arr = list(map_dict.items())
        lenth = len(map_dict.keys())
        # 构造规模为k的minheap
        if k <= lenth:
            k_minheap = map_arr[:k]
            # 从后往前维护堆，避免局部符合而影响递归跳转，例:2,1,3,4,5,0
            for i in range(k // 2 - 1, -1, -1):
                heapify(k_minheap, k, i)
            # 对于k:，大于堆顶则入堆，维护规模为k的minheap
            for i in range(k, lenth): # 堆建好了，没有乱序，从前往后即可
                if map_arr[i][1] > k_minheap[0][1]:
                    k_minheap[0] = map_arr[i] # 入堆顶
                    heapify(k_minheap, k, 0)  # 维护 minheap
        # 如需按顺序输出，对规模为k的堆进行排序
        # 从尾部起，依次与顶点交换再构造minheap，最小值被置于尾部
        for i in range(k - 1, 0, -1):
            k_minheap[i], k_minheap[0] = k_minheap[0], k_minheap[i]
            k -= 1 # 交换后，维护的堆规模-1
            heapify(k_minheap, k, 0)
        return [item[0] for item in k_minheap]
```

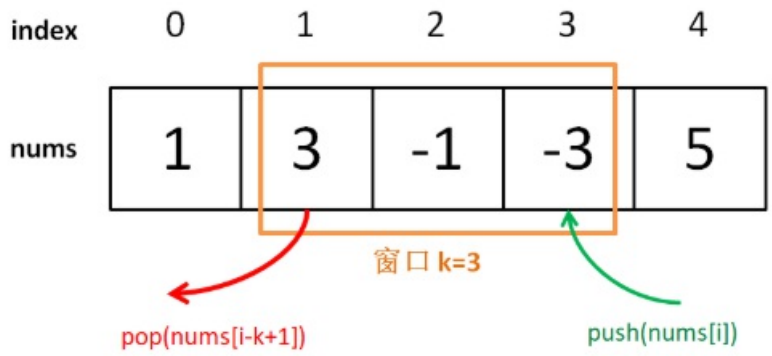
队列

双向队列

239. 滑动窗口最大值

> TODO: 用动态规划再做一次

双向队列




```
from collections import deque

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        # 建立双向队列,储存索引便于滑窗判断
        window = deque(); res = []
        for i in range(len(nums)):
            # 构建单调队列的push操作，注意用nums恢复索引
            while window and nums[window[-1]] <= nums[i]:
                window.pop()
            window.append(i)
            # 从k-1开始res append()
            if i >= k - 1:
                res.append(nums[window[0]])
            # 如果单调队列最大值落于滑窗之外，popleft()
            if window[0] == i-k+1:
                window.popleft()
        return res
```

动态规划

55. 跳跃游戏

动态规划，贪心

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        left_good = len(nums) - 1 # index
        for i in range(left_good, -1, -1):
            if nums[i] + i >= left_good:
                left_good = i
        return left_good == 0
```

参看 官方题解 四种方案思路很清楚

70. 爬楼梯

动态规划, 它的最优解可以从其子问题的最优解来有效地构建。

第 `i` 阶可以由以下两种方法得到：

- 在第 `(i-1)` 阶后向上爬 1 阶。
- 在第 `(i-2)` 阶后向上爬 2 阶。

所以到达第 `i` 阶的方法总数就是到第 `i-1` 阶和第 `i-2` 阶的方法数之和。

令 `dp[i]` 表示能到达第 `i` 阶的方法总数，则有（同斐波那契数）：
`dp[i] = dp[i-1] + dp[i-2]`

```
class Solution:
    def climbStairs(self, n: int) -> int:
        f0 = 1
        f1 = 2
        if n == 1: return f0
        if n == 2: return f1
        for i in range(n-2):
            f2 = f0 + f1
            f0 = f1
```

```
            f1 = f2
        return f2
```

杂

58. 最后一个单词的长度

```
class Solution:
    def lengthOfLastWord(self, s: str) -> int:
        l = 0
        flag = 0
        for i in s[::-1]:
            if not i.isspace():
                l += 1
                flag = 1
            if i.isspace() and flag: break
        return l
```

67. 二进制求和

```
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        grap = abs(len(a) - len(b))
        if len(a) > len(b):
            b = '0' * grap + b
        else: a = '0' * grap + a;
        s = ''
        add = 0
        for i in range(-1, -len(a)-1, -1):
            res = int(a[i]) + int(b[i]) + add
            add = 0
            if res > 1:
                res = res % 2
                add = 1
            s += str(res)
        if add == 1: s += str(1)
        return s[::-1]
```

66. 加一

```
class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:
        if digits[-1] != 9: digits[-1] += 1
        else:
            flag = 0; add = 0
            for i in range(len(digits)-1, -1, -1):
                if digits[i] == 9:
                    digits[i] = 0
                else: flag = 1; add = i; break
            if flag: digits[add] += 1
            else: digits.insert(0, 1)
        return digits
```