P3 练习参考速查表（Cheat Sheet）

---

1. 排序算法

计数排序（Counting Sort）

```
def counting_sort(A):
    """
    计数排序: 对非负整数数组排序
    时间: O(n+u), 空间: O(n+u), 稳定
    """
    # 找最大键值: O(n)
    u = 1 + max([x.key for x in A])

    # 创建直接访问数组（链）: O(u)
    D = [[] for i in range(u)]

    # 插入到链: O(n)
    for x in A:
        D[x.key].append(x)

    # 按顺序读出: O(u)
    i = 0
    for chain in D:
        for x in chain:
            A[i] = x
            i += 1
```

基数排序（Radix Sort）

```
def radix_sort(A):
    """
    基数排序: 键范围u < n^c时, O(cn) = O(n)
    """
    n = len(A)
    u = 1 + max([x.key for x in A])

    # 计算需要的位数（base n）
    c = 1 + (u.bit_length() // n.bit_length())

    # 创建数字元组
    class Obj: pass
    D = [Obj() for a in A]

    for i in range(n):
        D[i].digits = []
        D[i].item = A[i]
        high = A[i].key
```

```
    for j in range(c):
        high, low = divmod(high, n)
        D[i].digits.append(low)

# 对每一位进行计数排序
for i in range(c):
    for j in range(n):
        D[j].key = D[j].digits[i]
    counting_sort(D)

# 输出结果
for i in range(n):
    A[i] = D[i].item
```

注：以上代码假设元素有.key 属性。对于简单整数数组，需要适当修改。

2. Python set（集合）

函数/操作	功能说明	时间复杂度
构建操作		
`s = set()`	创建空集合	$O(1)$
`s = {1, 2, 3}`	字面量初始化	$O(n)$
`s = set([1,2,3,2])`	从列表构建（自动去重）	$O(n)$
基本操作		
`s.add(x)`	添加元素 x	$O(1)$ 期望
`s.remove(x)`	删除元素 x（不存在抛异常）	$O(1)$ 期望
`s.discard(x)`	删除元素 x（不存在不报错）	$O(1)$ 期望
`x in s`	检查元素是否存在	$O(1)$ 期望
`len(s)`	返回集合大小	$O(1)$
集合运算		
`s1 & s2`	交集	$O(\min(\|s_1\|, \|s_2\|))$
`s1 | s2`	并集	$O(\|s_1\| + \|s_2\|)$
`s1 - s2`	差集	$O(\|s_1\|)$
`s1 ^ s2`	对称差	$O(\|s_1\| + \|s_2\|)$
其他操作		
`s.clear()`	清空集合	$O(1)$

集合运算复杂度解释

为什么交集是 $O(\min(|s_1|, |s_2|))$?

- 算法：遍历较小的集合，检查每个元素是否在另一个集合中
- 优化：Python 自动选择较小的集合遍历
- 时间：$\min(|s_1|, |s_2|)$ 次遍历 $\times$ $O(1)$ 检查 $= O(\min)$

为什么并集是 $O(|s_1| + |s_2|)$?

- 算法：必须访问两个集合的所有元素
- 步骤 1：复制 s_1 的所有元素
- 步骤 2：添加 s_2 中不在 s_1 的元素
- 时间：$O(|s_1|) + O(|s_2|) = O(|s_1| + |s_2|)$

为什么差集是 $O(|s_1|)$?

- 算法：遍历 s_1，检查每个元素是否在 s_2 中
- 只需遍历 s_1：$O(|s_1|)$ 次 $\times$ $O(1)$ 检查
- 不需要遍历 s_2（只用于 O(1) 查询）

3. Python dict（字典）

函数/操作	功能说明	时间复杂度
构建操作		
`d = {}`	创建空字典	$O(1)$
`d = {'a': 1, 'b': 2}`	字面量初始化	$O(n)$
`d = dict([('a',1),...])`	从列表构建	$O(n)$
基本操作		
`d[key] = value`	插入/更新键值对	$O(1)$ 期望
`value = d[key]`	访问值（不存在抛 KeyError）	$O(1)$ 期望
`value = d.get(key)`	访问值（不存在返回 None）	$O(1)$ 期望
`value = d.get(key, default)`	访问值（不存在返回默认值）	$O(1)$ 期望
`del d[key]`	删除键值对	$O(1)$ 期望
`value = d.pop(key)`	删除并返回值	$O(1)$ 期望
`key in d`	检查键是否存在	$O(1)$ 期望
`len(d)`	返回键值对数量	$O(1)$
`d.keys()`	返回所有键的视图	$O(1)$
`d.values()`	返回所有值的视图	$O(1)$
`d.items()`	返回 (key,value) 对的视图	$O(1)$
`d.setdefault(key, val)`	不存在则设置，返回值	$O(1)$ 期望
`d.update({...})`	批量更新	$O(m)$
`d.clear()`	清空字典	$O(1)$
遍历操作		
`for k in d`	遍历键	$O(n)$
`for v in d.values()`	遍历值	$O(n)$
`for k,v in d.items()`	遍历键值对	$O(n)$

4. Python heapq（堆）

函数	功能说明	复杂度
构建操作		
`heapq.heapify(list)`	将列表转换为堆（原地）	$O(n)$
基本操作		
`heapq.heappush(heap, item)`	插入元素到堆	$O(\log n)$
`heapq.heappop(heap)`	弹出并返回最小元素	$O(\log n)$
`heap[0]`	查看最小元素（不删除）	$O(1)$
`heapq.heapreplace(h, item)`	弹出最小 + 插入新元素	$O(\log n)$
`heapq.heappushpop(h, item)`	插入 + 弹出最小	$O(\log n)$
`heapq.nsmallest(k, iter)`	返回最小的 k 个元素	$O(n \log k)$
`heapq.nlargest(k, iter)`	返回最大的 k 个元素	$O(n \log k)$

nlargest/nsmallest 复杂度解释

为什么是 $O(n \log k)$ 而不是 $O(n \log n)$?

- 算法：维护大小为 k 的堆（不是所有元素建堆）
- 过程（以 nlargest 为例）：
  1. 前 k 个元素建最小堆：$O(k)$
  2. 遍历剩余 $n - k$ 个元素：
     - 如果元素 > 堆顶（第 k 大）
     - 弹出堆顶，插入新元素：$O(\log k)$
  3. 总共：$O(k) + (n - k) \cdot O(\log k) = O(n \log k)$
- 优势：当 $k \ll n$ 时，$O(n \log k) \ll O(n \log n)$
- 示例：$n = 1000000, k = 10$
  - 完全排序：$O(n \log n) \approx 20000000$
  - nlargest：$O(n \log k) \approx 3000000$（快 6 倍）

重要提示

- Python heapq 是最小堆，heap[0] 是最小元素
- 实现最大堆：将元素取负数
- 复杂对象：使用元组 `(-priority, data)`
- **heapify** 是 $O(n)$，不是 $O(n \log n)$!（Floyd 算法）

构建复杂度对比

关键：从列表构建堆，用 `heapify` 比逐个 `heappush` 快!

数据结构	构建方法	时间复杂度
set	`set([1,2,3])`	$O(n)$
dict	`dict([('a',1),...])`	$O(n)$
heap	`heapq.heapify(list)`	$O(n)$
heap	n 次 `heappush`	$O(n \log n)$

最大堆示例

```
# 方法1: 取负数
max_heap = []
heapq.heappush(max_heap, -5)        # 插入5
max_val = -heapq.heappop(max_heap)  # 弹出5

# 方法2: 元组
max_heap = []
heapq.heappush(max_heap, (-priority, item))
neg_pri, item = heapq.heappop(max_heap)
priority = -neg_pri
```

元组比较规则

Python 比较元组时逐元素比较:

```
(1, 'a') < (2, 'b')     # True  - 第一个元素决定
(1, 'b') < (1, 'a')     # False - 第一个相同，比第二个
(-5, 100) < (-3, 10)    # True  - -5 < -3
```

5. 常用模式

计数器模式

```
# 使用dict计数
count = {}
for item in items:
    count[item] = count.get(item, 0) + 1

# 或使用Counter
from collections import Counter
count = Counter(items)
```

去重模式

```
# 使用set去重
unique_items = list(set(items))

# 检查重复
seen = set()
for item in items:
    if item in seen:
        print("重复")
    seen.add(item)
```

Top-K 模式

```
# 方法1: 使用nlargest
import heapq
largest_k = heapq.nlargest(k, arr)

# 方法2: 维护k大小最小堆
heap = arr[:k]
heapq.heapify(heap)
for num in arr[k:]:
    if num > heap[0]:
        heapq.heapreplace(heap, num)
# heap中是最大的k个元素
```