Cheat Sheet

python库

二分

```python
import bisect

a = [1, 2, 4, 4, 8]
print(bisect.bisect_left(a, 4))  # 输出: 2

print(bisect.bisect_right(a, 4))  # 输出: 4

print(bisect.bisect(a, 4))  # 输出: 4
```

优先队列

即该队列自动的保证顺序

```python
import heapq

data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapq.heapify(data)
print(data)  # 输出: [0, 1, 2, 3, 9, 5, 4, 6, 8, 7]

heapq.heappush(data, -5)
print(data)  # 输出: [-5, 0, 2, 3, 1, 5, 4, 6, 8, 7, 9]

print(heapq.heappop(data))  # 输出: -5
```

日期与时间

```python
import calendar, datetime

print(calendar.isleap(2020))  # 输出: True

print(datetime.datetime(2023, 10, 5).weekday())  # 输出: 3 (星期四)
```

数据结构

```python
import collections

# deque
dq = collections.deque([1, 2, 3])
dq.append(4)
print(dq)  # 输出: deque([1, 2, 3, 4])
dq.appendleft(0)
print(dq)  # 输出: deque([0, 1, 2, 3, 4])
dq.pop()
print(dq)  # 输出: deque([0, 1, 2, 3])
dq.popleft()
print(dq)  # 输出: deque([1, 2, 3])

dd = collections.defaultdict(int)
dd['a'] += 1
print(dd)  # 输出: defaultdict(<class 'int'>, {'a': 1})

od = collections.OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od)  # 输出: OrderedDict([('a', 1), ('b', 2), ('c', 3)])

Point = collections.namedtuple('Point', ['x', 'y'])
p = Point(11, 22)
print(p)  # 输出: Point(x=11, y=22)
print(p.x, p.y)  # 输出: 11 22
```

2. 使用f-string（Python 3.6+）：

```python
number = 3.14159
formatted_number = f"{number:.3f}"
print(formatted_number)  # 输出: 3.142
```

利用二分查找求最长下降子序列

```
import bisect

k = int(input())
a = list(map(int, input().split()))

# 反转数组来模拟下降子序列问题
sub = []

for i in range(k):
    # 由于我们在寻找递增子序列的 "位置"，所以我们在 sub 中寻找 "大于" 当前元素的位置
    pos = bisect.bisect_right(sub, a[i])  # 使用 bisect_right 来模拟下降序列的插入位置

    if pos < len(sub):
        sub[pos] = a[i]  # 更新该位置的值
    else:
        sub.append(a[i])  # 如果没有找到合适位置，说明我们可以增加一个新的元素
```

哈希表

求得一个数组最长的和为0的连续子序列的方法（实例为求最长平均值为a的字串）

```python
def longest_subarray_with_avg(arr, a):
    # 步骤 1: 将每个元素减去a
    diff = [x - a for x in arr]

    # 步骤 2: 使用哈希表存储前缀和
    prefix_sum = 0
    prefix_map = {0: -1}  # 前缀和为0时，起始索引为-1
    max_len = 0
    start_index = -1

    # 步骤 3: 遍历数组并计算前缀和
    for i in range(len(diff)):
        prefix_sum += diff[i]

        # 步骤 4: 如果前缀和已经出现过，计算子数组的长度
        if prefix_sum in prefix_map:
            length = i - prefix_map[prefix_sum]
            if length > max_len:
                max_len = length
                start_index = prefix_map[prefix_sum] + 1
        else:
            prefix_map[prefix_sum] = i

    # 返回最长子数组
    if max_len > 0:
        return arr[start_index:start_index + max_len]
    else:
        return []
```

算法

dfs模板

连通域染色

```python
directions = ((0, 1), (0, -1), (-1, 0), (1, 0),
              (1, -1), (1, 1), (-1, -1), (-1, 1))

def dfs(x, y):
    global n, m, color, board
    board[x][y] = color
    area = 1
    for d in directions:
        nx = x+d[0]
        ny = y+d[1]
        if any((nx < 0, ny < 0, nx >= n, ny >= m)):
            continue
        if board[nx][ny] == "W":
            area += dfs(nx, ny)
    return area
```

水淹七军

```python
def dfs_iterative(x, y, h, m, n, grid, water_height):
    stack = [(x, y)]
    water_height[x][y] = h

    while stack:
        cx, cy = stack.pop()

        for dx, dy in directions:
            nx = cx + dx
            ny = cy + dy
            if (0 <= nx < m and 0 <= ny < n and
                grid[nx][ny] < h and
                    water_height[nx][ny] < h):
                water_height[nx][ny] = h
                stack.append((nx, ny))
```

bfs模板

螃蟹采蘑菇

```python
def bfs(n, grid):
    start, end = find_start_end(n, grid)
    q = deque([start])
    visited = set()
    visited.add(tuple(start))
    flag = False
    while q:
        if flag:
            break
        current = q.popleft()
        if end in current:
            flag = True
            break
        for dx, dy in directions:
            nx1, ny1 = current[0][0]+dx, current[0][1]+dy
            nx2, ny2 = current[1][0]+dx, current[1][1]+dy
            if 0 <= nx1 < n and 0 <= ny1 < n and 0 <= nx2 < n and 0 <= ny2 < n:
                if grid[nx1][ny1] != 1 and grid[nx2][ny2] != 1:
                    next_state = [(nx1, ny1), (nx2, ny2)]
                    if tuple(next_state) not in visited:
                        visited.add(tuple(next_state))
                        q.append(next_state)
    return flag
```

筛法

```python
def euler_sieve(n):
    is_prime = [True] * (n + 1)
    primes = []
    for i in range(2, n + 1):
        if is_prime[i]:
            primes.append(i)
        for p in primes:
            if i * p > n:
                break
            is_prime[i * p] = False
            if i % p == 0:
                break
    return primes

n = 50
print(euler_sieve(n))  # 输出: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

dp

01背包

```python
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

weights = [1, 2, 3, 4]
values = [10, 20, 30, 40]
capacity = 5
print(knapsack(weights, values, capacity))  # 输出: 50
```

完全背包

```python
def knapsack_complete(weights, values, capacity):
    dp = [0] * (capacity + 1)
    for i in range(len(weights)):
        for w in range(weights[i], capacity + 1):
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    return dp[capacity]

weights = [1, 2, 3, 4]
values = [10, 20, 30, 40]
capacity = 5
print(knapsack_complete(weights, values, capacity))  # 输出: 50
```

必须装满的完全背包

```python
def knapsack_complete_fill(weights, values, capacity):
    dp = [-float('inf')] * (capacity + 1)
    dp[0] = 0
    for i in range(len(weights)):
        for w in range(weights[i], capacity + 1):
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    return dp[capacity] if dp[capacity] != -float('inf') else 0

weights = [1, 2, 3, 4]
values = [10, 20, 30, 40]
capacity = 5
print(knapsack_complete_fill(weights, values, capacity))  # 输出: 50
```

多重背包（二进制优化）

```python
def binary_optimized_multi_knapsack(weights, values, quantities, capacity):
    n = len(weights)
    items = []
    # 将每个物品拆分成若干子物品
    for i in range(n):
        w, v, q = weights[i], values[i], quantities[i]
        k = 1
        while k < q:
            items.append((k * w, k * v))
            q -= k
            k <<= 1
        if q > 0:
            items.append((q * w, q * v))
    # 动态规划求解01背包问题
    dp = [0] * (capacity + 1)
    for w, v in items:
        for j in range(capacity, w - 1, -1):
            dp[j] = max(dp[j], dp[j - w] + v)
    return dp[capacity]

weights = [1, 2, 3]
values = [6, 10, 12]
quantities = [10, 5, 3]
capacity = 15

print(binary_optimized_multi_knapsack(weights, values, quantities, capacity))  # 输出: 120
```

Dijkstra

```python
import heapq

def dijkstra(graph, start):
    n = len(graph)
    distances = {node: float('inf') for node in range(n)}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

graph = {
    0: [(1, 4), (2, 1)],
    1: [(3, 1)],
    2: [(1, 2), (3, 5)],
    3: []
}

start_node = 0
print(dijkstra(graph, start_node))  # 输出: {0: 0, 1: 3, 2: 1, 3: 4}
```

走山路（高度差为距离）

```python
from heapq import heappop, heappush


directions = ((1, 0), (-1, 0), (0, 1), (0, -1))
INF = float("inf")


def dijkstra(start_x, start_y, end_x, end_y):
    global m, n, terrian
    if terrian[start_x][start_y] == "#" or terrian[end_x][end_y] == "#":
        return "NO"
    pq = [(0, start_x, start_y)]
    distance = [[INF]*n for _ in range(m)]
    distance[start_x][start_y] = 0

    while pq:
        d, x, y = heappop(pq)
        if x == end_x and y == end_y:
            return d
        for dx, dy in directions:
            nx, ny = x+dx, y+dy
            if 0 <= nx < m and 0 <= ny < n and terrian[nx][ny] != "#":
                nd = d+abs(terrian[nx][ny]-terrian[x][y])
                if nd < distance[nx][ny]:
                    distance[nx][ny] = nd
                    heappush(pq, (nd, nx, ny))
    return "NO"
```

Kadane算法

最大子序列

```python
def max_subarray_sum(nums):
    max_sum = current_sum = nums[0]
    for num in nums[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum

nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(nums))  # 输出: 6
```

最大子矩阵

枚举 `l~r` 列的子式，按行求和后，求这些和的最大子序列即可。

最长上升子序列

dp方法

```python
def length_of_lis(nums):
    if not nums:
        return 0
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(length_of_lis(nums))  # 输出: 4
```

二分法

```
import bisect

def length_of_lis_binary(nums):
    if not nums:
        return 0
    tails = []
    for num in nums:
        pos = bisect.bisect_left(tails, num)
        if pos == len(tails):
            tails.append(num)
        else:
            tails[pos] = num
    return len(tails)

nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(length_of_lis_binary(nums))  # 输出: 4
```

其它

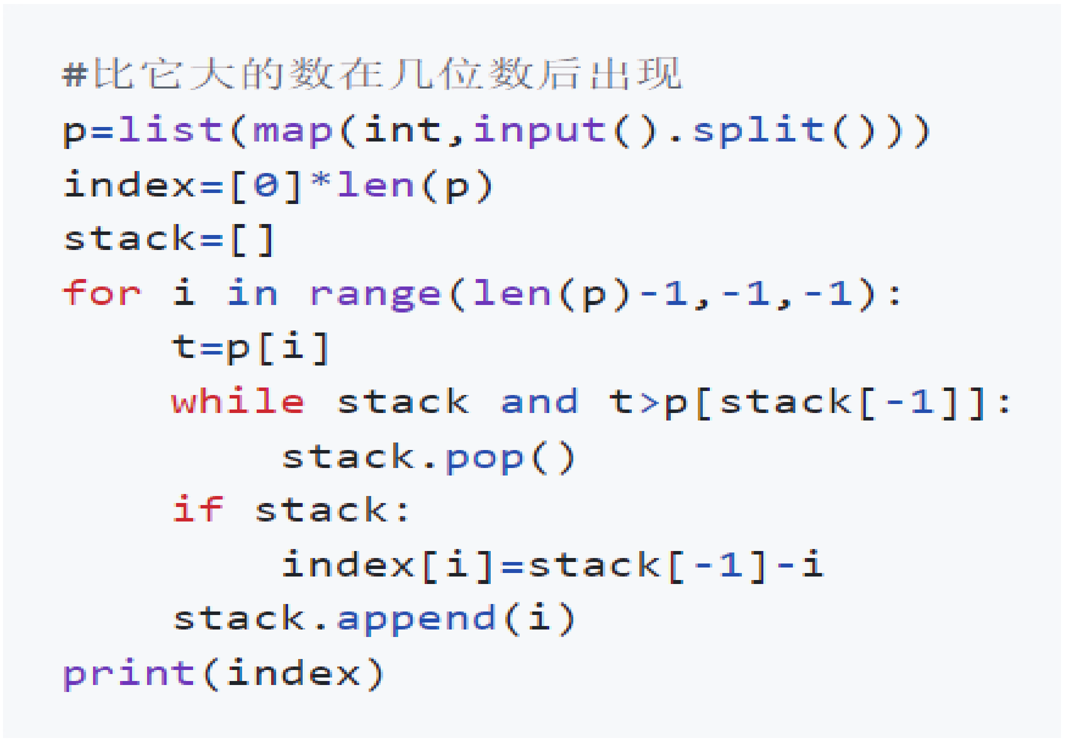
下一个全排列

```
def next_permutation(nums):
    i = len(nums) - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1
    if i >= 0:
        j = len(nums) - 1
        while nums[j] <= nums[i]:
            j -= 1
        nums[i], nums[j] = nums[j], nums[i]
    nums[i + 1:] = reversed(nums[i + 1:])
    return nums

nums = [1, 2, 3]
print(next_permutation(nums))  # 输出: [1, 3, 2]
```

Manacher算法

```python
def manacher(s):
    s = '#' + '#'.join(s) + '#'
    n = len(s)
    p = [0] * n
    c = r = 0
    for i in range(n):
        mirr = 2 * c - i
        if i < r:
            p[i] = min(r - i, p[mirr])
        while i + p[i] + 1 < n and i - p[i] - 1 >= 0 and s[i + p[i] + 1] == s[i - p[i] - 1]:
            p[i] += 1
        if i + p[i] > r:
            c, r = i, i + p[i]
    max_len, center_index = max((n, i) for i, n in enumerate(p))
    return s[center_index - max_len:center_index + max_len].replace('#', '')

s = "babad"
print(manacher(s))  # 输出: "bab" 或 "aba"
```

单调栈



```python
#比它大的数在几位数后出现
p=list(map(int,input().split()))
index=[0]*len(p)
stack=[]
for i in range(len(p)-1,-1,-1):
    t=p[i]
    while stack and t>p[stack[-1]]:
        stack.pop()
    if stack:
        index[i]=stack[-1]-i
    stack.append(i)
print(index)
```

递归加速

```
from functools import lru_cache
@lru_cache(maxsize=None)
def ...
```

前缀和

O（n2）降到O（n）

懒删除

需要删除一个元素时先对其打上标记，等到操作到该元素时再将其弹出去（由于一般list中删除的复杂度是线性的，操作时跳过打了标记的元素；但这种方法在heap中的运用更为典型）。打上标记后元素仍在序列中，但我们将其**视作已经不存在**进行操作；等到需要真正操作该元素（该元素已到堆顶）时，再做实质上的删除，这样避免了从heap中直接删除元素。

```
#每次要删除x时out[x]+=1
from heapq import heappop,heappush
while ls:
    x = heappop(ls)
    if not out[x]:
        new_min = x
        heappush(ls,x) #不需要弹出的，记得压回去
        break
    out[x]-=1
```

heapq小顶堆

`heappush(heap, item)`：将 `item` 的值加入 `heap` 中，保持堆的性质。

`heappop(heap)`：从 `heap` 中弹出并返回最小的元素，保持堆的性质。如果堆为空，则会引发 IndexError。

`heapify(data)`:将data变成小顶堆