DSA Written-exam Cheatsheet

1.排序算法

1.Compare the time complexity of different sorting algorithms

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$nlog^2n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heapsort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection	
Merge sort	$nlogn$	$nlogn$	$nlogn$	n	Yes	Merging	Highly parallelizable (up to *O*(log *n*) using the Three Hungarian's Algorithm)
Timsort	n	$nlogn$	$nlogn$	n	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Quicksort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning	Quicksort is usually done in-place with *O*(log *n*) stack space.
Shellsort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Insertion sort	n	n^2	n^2	1	Yes	Insertion	*O*(n + d), in the worst case over sequences that have *d* inversions.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.

注：

a.稳定的排序：归并排序、插入排序、冒泡排序

b.不利情况：冒泡排序（完全逆序，$O(n^2)$）；选择排序（完全有序，$O(n^2)$)；插入排序（完全逆序，$O(n^2)$）；快速排序（完全有序，$O(n^2)$）

2.具体实现

```
1.冒泡排序
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
```

```
2.选择排序
def selection_sort(arr):
    for p in range(len(arr)-1, 0, -1):
        position = 0
        for location in range(1, p+1):
            if arr[location] > arr[position]:
                position = location
        if p != position:
            arr[p], arr[position] = arr[position], arr[p]
3.快速排序（分治）
def quick_sort(arr, left, right):
    if left < right:
        position = partition(arr, left, right)
        quick_sort(arr, left, position-1)
        quick_sort(arr, position+1, right)
def partition(arr, left, right):
    i = left
    j = right-1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
4.归并排序
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]
        merge_sort(left)
        merge_sort(right)
        i, j, k = 0, 0, 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
```

```
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
        while j > len(right):
            arr[k] = right[j]
            j += 1
            k += 1
5.插入排序
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
6.希尔排序
def shell_sort(arr, n):
    gap = n // 2
    while gap > 0:
        j = gap
        while j < n:
            i = j - gap
            while i >= 0:
                if arr[i+gap] > arr[i]:
                    break
                else:
                    arr[i+gap], arr[i] = arr[i], arr[i+gap]
                i -= gap
            j += 1
        gap //= 2
7.堆排序
def heapify(arr, n, i):
    largest = i
    l = 2*i + 1
    r = 2*i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
def heapsort(arr):
    n = len(arr)
```

```
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

2.concepts

逻辑结构：集合、线性、树、图

存储结构：

1.顺序结构：结点在内存中连续存放，所有结点占据一片连续的内存空间。如list。

2.链接结构：结点在内存中可不连续存放，每个结点中存有指针指向其前驱结点和/或后继结点。如链表，树。

3.索引结构：将结点的关键字信息拿出来单独存储，并且为每个关键字x配一个指针指向关键字为x的结点，这样便于按照关键字查找到相应的结点。

4.散列结构：设置散列函数，散列函数以结点的关键字为参数，算出一个结点的存储位置。

注：数据的逻辑结构和存储结构无关（一种逻辑结构的数据，可以用不同的存储结构来存储；树结构、图结构可以用链接结构存储，也可以用顺序结构存储；线性结构可以用顺序结构存储，也可以用链接结构存储。）

3.线性表

·线性表中的元素属于相同的数据类型，即每个元素所占的空间必须相同。

1.顺序表

元素在内存中连续存放，随机访问。

元素个数小于容量时，append操作复杂度O(1)；元素个数等于容量时，append导致重新分配空间，且要拷贝原有元素到新空间，复杂度O(n)。（重新分配空间时，新容量为旧容量的k倍(k>1且固定)，可确保append操作的平均复杂度是O(1)。Python的list取k=1.2左右）

2.链表

访问第i个元素，复杂度为O(n)；已经找到插入或删除位置的情况下，插入和删除元素的复杂度O(1),且不需要复制或移动结点。

形式：单链表、循环单链表、双向链表、循环双向链表

4.栈

```
#中序转后序
def midToSuffix(s):
    s = s.split()
    stack,result = [],[]
```

```
    priority = {"/": 1, "*": 1, "+": 2, "-": 2}
    for x in s:
        if x == "(":
            stack.append(x)
        elif x == ")":
            while stack[-1] != "(":
                result.append(stack.pop())
            stack.pop()
        elif x in "/*+-":
            while len(stack) >= 1 and \
                  stack[-1] != '(' and priority[stack[-1]] <= priority[x]:
                result.append(stack.pop())
            stack.append(x)
        else:
            result.append(x)
    while stack != []:
        result.append(stack.pop())
    return " ".join(result)
print(midToSuffix(input()))

#计算中序表达式
def countMid(s):
    s = s.split()
    stkNum, stkOp = [], []
    priority = {"/": 1, "*": 1, "+": 2, "-": 2}
    for x in s:
        if x == "(":
            stkOp.append(x)
        elif x == ")":
            while stkOp[-1] != "(":
                op = stkOp.pop()
                a, b = stkNum.pop(), stkNum.pop()
                result = eval(str(b) + op + str(a))
                stkNum.append(result)
            stkOp.pop()
        elif x in "/*+-":
            while len(stkOp) >= 1 and stkOp[-1] != '(' and priority[stkOp[-1]]    <=
priority[x]:
                op = stkOp.pop()
                a, b = stkNum.pop(), stkNum.pop()
                result = eval(str(b) + op + str(a))
                stkNum.append(result)
            stkOp.append(x)
        else:  # 如果是数字，直接入栈
            stkNum.append(float(x))
    # 清空运算符栈中的剩余运算符
```

```python
    while len(stkOp) > 0:
        op = stkOp.pop()
        a, b = stkNum.pop(), stkNum.pop()
        result = eval(str(b) + op + str(a))
        stkNum.append(result)
    return stkNum[-1]

#合法出栈序列
def is_valid_pop_sequence(origin,pop_sequence):
    if len(pop_sequence) != len(origin):
        return False
    stack = []
    bank = list(origin)
    for i in pop_sequence:
        while (not stack or stack[-1] != i) and bank:
            stack.append(bank.pop(0))
        if not stack or stack[-1] != i:
            return False
        stack.pop()
    return True
origin = input().strip()
while True:
    try:
        s = input().strip()
        if is_valid_pop_sequence(origin, s):
            print('YES')
        else:
            print('NO')
    except EOFError:
        break
```

5.队列

·判满：不维护size, 浪费queue中一个单元的存储空间，(tail + 1) % capacity == head 即为满。如果不浪费，就无法区分head == tail是队列空导致，还是队列满导致。

6.二叉树

树的二叉树表示法（儿子-兄弟表示法)：树的前序遍历序列，和其儿子兄弟树的前序遍历序列一致；树的后序遍历序列，和其儿子兄弟树的中序遍历序列一致。

森林：不相交的树的集合，就是森林；森林有序，有第1棵树、第2棵树、第3棵树之分；森林可以表示为树的列表，也可以表示为一棵二叉树

一个二叉树是二叉搜索树，当且仅当其中序遍历序列是递增序列。

二叉排序树删除结点：

法1： 找到X的中序遍历后继结点，即X右子树中最小的结点Y，用Y的key和value覆盖X中的key和value，然后递归删除Y。（如何找Y: 进入X的右子结点，然后不停往左子结点走，直到没有左子结点为止。）

法2：找到X的中序遍历前驱结点，即X左子树中最大的结点Y，用Y的key和value覆盖X中的key和value，然后递归删除Y。（如何找Y: 进入X的左子结点，然后不停往右子结点走，直到没有右子结点为止。）

```
#AVL
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1

class AVL:
    def __init__(self):
        self.root = None
def insert(self, v):
    if self.root is None:
        self.root = Node(v)
    else:
        self.root = self._insert(v, self.root)

def _insert(self, v, node):
    if node is None:
        return Node(v)
    elif v < node.val:
        node.left = self._insert(v, node.left)
    else:
        node.right = self._insert(v, node.right)
    node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))
    balance = self._get_balance(node)
    if balance > 1:
        if v < node.left.val:
            return self.rotate_right(node)
        else:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)
    elif balance < -1:
        if v > node.right.val:
            return self.rotate_left(node)
        else:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)
    return node


def _get_height(self, node):
```

```python
    def _get_height(self, node):
        if node is None:
            return 0
        return node.height

    def _get_balance(self, node):
        if node is None:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def rotate_left(self, node):
        nd = node.right
        tmp = nd.left
        nd.left = node
        node.right = tmp
        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))
        nd.height = 1 + max(self._get_height(nd.left), self._get_height(nd.right))
        return nd

    def rotate_right(self, node):
        nd = node.left
        tmp = nd.right
        nd.right = node
        node.left = tmp
        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))
        nd.height = 1 + max(self._get_height(nd.left), self._get_height(nd.right))
        return nd

    def _prefix(self, node):
        if node is None:
            return []
        return [node.val]+self._prefix(node.left)+self._prefix(node.right)

    def prefix(self):
        return ' '.join(map(str, self._prefix(self.root)))
```

7.堆

添加、删除元素O(logn)；无序列表原地建堆O(n)

```python
#二叉堆
class BinHeap:
    def __init__(self):
        self.list = [0]
        self.size = 0
```

```
    def up(self, i):
        while i // 2 > 0:
            if self.list[i] < self.list[i // 2]:
                tmp = self.list[i // 2]
                self.list[i // 2] = self.list[i]
                self.list[i] = tmp
            i //= 2

    def heappush(self, k):
        self.list.append(k)
        self.size += 1
        self.up(self.size)

    def min(self, i):
        if i*2+1 > self.size:
            return i*2
        else:
            if self.list[i*2] < self.list[i*2+1]:
                return i*2
            else:
                return i*2+1

    def down(self, j):
        while (j*2) <= self.size:
            t = self.min(j)
            if self.list[j] > self.list[t]:
                tmp = self.list[j]
                self.list[j] = self.list[t]
                self.list[t] = tmp
            j = t

    def heappop(self):
        ans = self.list[1]
        self.list[1] = self.list[self.size]
        self.size -= 1
        self.list.pop()
        self.down(1)
        return ans
```

8.图

·不加堆优化的Prim 算法适用于密集图，加堆优化的适用于稀疏图。

·一个图的两棵最小生成树，边的权值序列排序后结果相同。

·拓扑排序：用队列存放入度变为0的点。每个顶点出入队列一次，每个顶点连的边都要看一次，复杂度O(E+V)

·关键路径：AOE网络上权值之和最大的路径(不唯一）。决定了活动完成的最短时间

递推求earliestTime[i]

1.初始条件：对每个入度为0的顶点k(事件k)， earlistTime[k] = 0.

2.拓扑排序

3.按拓扑序列的顺序递推每个事件的最早开始时间：对拓扑序列中的顶点 i,若边<i, j>存在且权值为Wij，则:earliestTime[j] = max(earliestTime[j], earliestTime[i] + Wij)

递推求latestTime[i]

1.求出全部活动都完成的最早时刻 T

2.初始条件：对每个出度为0的顶点k(事件k)， latestTime [k] = T

3.拓扑排序

4.按拓扑序列的逆序递推每个事件的最晚开始时间：对拓扑逆序列中的顶点 j,若边<i , j>存在且权值为Wij， 则: latestTime[i] = min(latestTime[i], latestTime[j] - Wij)

```python
#最短路Floyd算法
def floyd(G):  #G是邻接矩阵，顶点编号从0开始算,无边则边权值为INF
    n = len(G)
    INF = 10 ** 9
    prev = [[None for i in range(n)] for j in range(n)]
    dist = [[INF for i in range(n)] for j in range(n)]

    # 初始化邻接矩阵和前驱数组
    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            else:
                if G[i][j] != INF:  # 如果顶点i到顶点j有边
                    dist[i][j] = G[i][j]
                    prev[i][j] = i  # 记录j的前驱为i

    # Floyd-Warshall 算法核心部分
    for k in range(n):  # 中间节点
        for i in range(n):  # 起始节点
            for j in range(n):  # 终止节点
```

```
            for j in range(n):  # 终止节点
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    prev[i][j] = prev[k][j]  # 更新j的前驱为k

    return dist, prev
#dist[i][j]就是i到j的最短路 prev[i][j]是i到j的最短路上j的前驱 prev[i][prev[i][j]]是j的前驱的前驱
```

9.散列冲突

处理散列表冲突的常见方法包括以下几种：

1. 链地址法（Chaining）：使用链表来处理冲突。每个散列桶（哈希桶）中存储一个链表，具有相同散列值的元素会链接在同一个链表上。
2. 开放地址法（Open Addressing）：
    - 线性探测（Linear Probing）：如果发生冲突，就线性地探测下一个可用的槽位，直到找到一个空槽位或者达到散列表的末尾。
    - 二次探测（Quadratic Probing）：如果发生冲突，就使用二次探测来查找下一个可用的槽位，避免线性探测中的聚集效应。
    - 双重散列（Double Hashing）：如果发生冲突，就使用第二个散列函数来计算下一个槽位的位置，直到找到一个空槽位或者达到散列表的末尾。
3. 再散列（Rehashing）：当散列表的**装载因子（load factor）**超过一定阈值时，进行扩容操作，重新调整散列函数和散列桶的数量，以减少冲突的概率。
4. 建立公共溢出区（Public Overflow Area）：将冲突的元素存储在一个公共的溢出区域，而不是在散列桶中。在进行查找时，需要遍历溢出区域。

这些方法各有优缺点，适用于不同的应用场景。选择合适的处理冲突方法取决于数据集的特点、散列表的大小以及性能需求。