Cheet Sheet 笔试

Cheet Sheet 笔试

1.总

1.常见算法的时间复杂度：

2.数据结构：

·逻辑结构：

·物理（存储）结构：数据在计算机内存中的存储方式

2.线性结构、哈希、栈和队列

1.处理散列表的冲突：

2.栈

中序转后序：调度场算法

合法出栈序列：

3.树

1.二叉树

1.二叉树的遍历：

2.Huffman树：

3.堆：

4.字符串的KMP算法：

4.图：

1.表示方法：

2.遍历方法，dfs、bfs：

最大权值连通块：

有向图判环：

3.图的算法：

1.单源最短路径-Dijkstra：表记录最短距离，每次选择最近未处理节点进行更新

2.每对顶点之间最短路-Floyd算法：

3.最小生成树-Prim算法：贪心，对于每个节点在现有的树中找最短的路径

4.最小生成树-Kruskal算法：贪心，把边排序，逐一处理，不可成环

5.拓扑排序-Kahn算法：维护一个入度为零的表

5.排序

1.冒泡排序：

2.选择排序：

3.插入排序：

4.希尔排序：递减增量排序

5.归并排序：

6.快速排序：

1.总

1.常见算法的时间复杂度：

二分查找、BST查找：logn

算法	时间	空间	描述
顺序表			
插入	n		移动元素，平均n/2次
删除	n		移动平均(n-1)/2次

算法	时间	空间	描述
按值查找	n		
链表			
头尾插创建	n		
按值/序查找	n		
插入、删除	1		
二叉树			
创建二叉树	n	n	类似先序遍历
遍历	n	n	递归遍历node
BST插入删除	logn		
图			
邻接矩阵		V2	
邻接表		V+E(2E)	
邻接矩阵BFS	V2		
邻接表BFS	V+E		
邻接矩阵DFS	V2		
邻接表BFS	V+E		
prim	V2，改进可达ElogV		稠密图
kru	ElogE		边稀疏，顶点多
dijk	V2		
floyd	V3		
拓扑排序	V+E		
平均查找长度ASL			
顺序查找	$\sum_{i=1}^{n} P_i(n-i+1)=\frac{n+1}{2}$		不成功n+1，平均为二者平均
有序顺序查找	$ASL_{false}=n/2+n/(n+1)$		
二分查找	logn		

排序

表 8.1　各种排序算法的性质

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
2 路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

2.数据结构：

·逻辑结构：

线性表：数组、链表

非线性表：树、堆、图、哈希表

·物理（存储）结构：数据在计算机内存中的存储方式

连续：数组适合访问，不适合增删

分散：链表适合增删，不适合访问

2.线性结构、哈希、栈和队列

1.处理散列表的冲突：

开放地址法：迭代哈希函数；链地址法：同义词链表

2.栈

中序转后序：调度场算法

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
   - 如果是操作数（数字），则将其添加到输出栈。
   - 如果是左括号，则将其推入运算符栈。
   - 如果是运算符：
     - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
     - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
     - 将当前运算符推入运算符栈。
   - 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。

3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

合法出栈序列：

先判断是否在原栈里，如果在就出栈入栈直到该元素，如果不在就必须是新栈顶，否则不合理

3.树

1.二叉树

满二叉树：所有层的节点都被填满

完全二叉树：只有最底层的节点未被填满，且从左向右填充

完满二叉树：所有节点的度为0或2

1.二叉树的遍历：

前中后序，按层次遍历

前中序转后序：

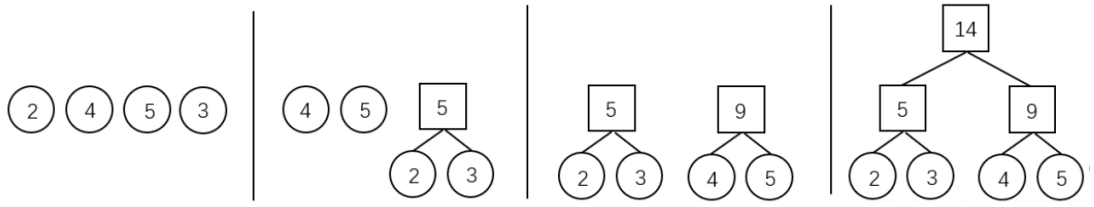
前序定根，中序分左右子树，递归

2.Huffman树：

Huffman树的带权路径长度：每个源码节点乘上到这个节点的路径长度求和

Huffman 算法用于构造一棵 Huffman 树, 算法步骤如下:

1. **初始化**: 由给定的 n 个权值构造 n 棵只有一个根节点的二叉树, 得到一个二叉树森林 F.
2. **选取与合并**: 从二叉树集合 F 中选取根节点权值**最小**的两棵二叉树分别作为左右子树构造一棵新的二叉树, 这棵新二叉树的根节点的权值为其左、右子树根结点的权值和.
3. **删除与加入**: 从 F 中删除作为左、右子树的两棵二叉树, 并将新建立的二叉树加入到 F 中.
4. 重复2, 3步, 当森林中只剩下一棵二叉树时, 这棵二叉树就是霍夫曼树.



3.堆：

二叉堆的插入、删除：

插入：插入一个元素到末位，从下往上递归判断是否交换
删除：去除顶端元素，将末位元素挪至顶端，从上向下递归判断是否交换

二叉堆的性质：画图判断，基本是二倍关系

4.字符串的KMP算法：

计算前缀表-根据前缀表移动两个指针进行匹配

next的作用：找到相等的前缀和后缀位置

4.图：

1.表示方法：

邻接表：外层表示节点，内层表示每个节点相连的节点-稀疏

邻接矩阵：二维矩阵，格中位置表示边的权重

2.遍历方法，dfs、bfs：

最大权值连通块：

```python
def max_weight(n, m, weights, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = [False] * n
    max_weight = 0

    def dfs(node):
        visited[node] = True
        total_weight = weights[node]
        for neighbor in graph[node]:
            if not visited[neighbor]:
                total_weight += dfs(neighbor)
        return total_weight

    for i in range(n):
        if not visited[i]:
            max_weight = max(max_weight, dfs(i))

    return max_weight

# 接收数据
n, m = map(int, input().split())
weights = list(map(int, input().split()))
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
print(max_weight(n, m, weights, edges))
```

有向图判环：

```python
def has_cycle(n, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)

    color = [0] * n

    def dfs(node):
        if color[node] == 1:
            return True
        if color[node] == 2:
            return False

        color[node] = 1
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True
        color[node] = 2
        return False

    for i in range(n):
        if dfs(i):
            return "Yes"
    return "No"

# 接收数据
n, m = map(int, input().split())
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
```

3.图的算法：

1.单源最短路径-Dijkstra：表记录最短距离，每次选择最近未处理节点进行更新

Dijkstra算法：Dijkstra算法用于解决单源最短路径问题，即从给定源节点到图中所有其他节点的最短路径。算法的基本思想是通过不断扩展离源节点最近的节点来逐步确定最短路径。具体步骤如下：

- 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为0，其他节点的距离为无穷大。
- 选择一个未访问的节点中距离最小的节点作为当前节点。
- 更新当前节点的邻居节点的距离，如果通过当前节点到达邻居节点的路径比已知最短路径更短，则更新最短路径。
- 标记当前节点为已访问。
- 重复上述步骤，直到所有节点都被访问或者所有节点的最短路径都被确定。

Dijkstra算法的时间复杂度为O(V^2)，其中V是图中的节点数。当使用优先队列（如最小堆）来选择距离最小的节点时，可以将时间复杂度优化到O((V+E)logV)，其中E是图中的边数。

2.每对顶点之间最短路-Floyd算法：

5.1.3 多源最短路径Floyd-Warshall算法

求解所有顶点之间的最短路径可以使用**Floyd-Warshall算法**，它是一种多源最短路径算法。Floyd-Warshall算法可以在有向图或无向图中找到任意两个顶点之间的最短路径。

算法的基本思想是通过一个二维数组来存储任意两个顶点之间的最短距离。初始时，这个数组包含图中各个顶点之间的直接边的权重，对于不直接相连的顶点，权重为无穷大。然后，通过迭代更新这个数组，逐步求得所有顶点之间的最短路径。

具体步骤如下：

1. 初始化一个二维数组`dist`，用于存储任意两个顶点之间的最短距离。初始时，`dist[i][j]`表示顶点i到顶点的直接边的权重，如果i和j不直接相连，则权重为无穷大。
2. 对于每个顶点k，在更新`dist`数组时，考虑顶点k作为中间节点的情况。遍历所有的顶点对(i, j)，如果通过顶点k可以使得从顶点i到顶点j的路径变短，则更新`dist[i][j]`为更小的值。

   `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`
3. 重复进行上述步骤，对于每个顶点作为中间节点，进行迭代更新`dist`数组。最终，`dist`数组中存储的就是所有顶点之间的最短路径。

Floyd-Warshall算法的时间复杂度为O(V^3)，其中V是图中的顶点数。它适用于解决稠密图（边数较多）的最短路径问题，并且可以处理负权边和负权回路。

3.最小生成树-Prim算法：贪心，对于每个节点在现有的树中找最短的路径

朴素 Prim 算法

$O(V^2)$, 稠密图一般选用此算法.

```python
def prim(graph: List[List[int]], n: int):
    # 稠密图用邻接矩阵graph, 其中存边权, 无边存无穷大
    curDist = [float('inf') for _ in range(n)] # 点到当前树的最小距离, 是边权
    inMST = [False for _ in range(n)] # 标记是否已加入到MST中
    totalWeight = 0

    for _ in range(n): # 每次加一个点一条边到树中(第一次只加点不加边)
        # 1. 通过枚举点找到连接树和树外一点的最短边
        minNode = None
        for node in range(n):
            if not inMST[node] and (minNode is None
                    \ or curDist[node] < curDist[minNode]):
                minNode = node

        # 2. 把最短边及其连接的树外点加入到MST中(第一次循环只加点不加边)
        if i != 0:  # 当然也可将起点的 curDist 初始化为 0, 则此处无需判断
            totalWeight += curDist[minNode]
            # 如果这条最短边为inf, 就代表该树外点与树中任一点都不连通, 即原图是不连通的
        inMST[minNode] = True

        # 3. 更新树外节点到树的最小距离
        for nb in graph[minNode]:
            curDist[nb] = min(curDist[nb], graph[minNode][nb])

    return totalWeight
```

4.最小生成树-Kruskal算法：贪心，把边排序，逐一处理，不可成环

图的连通性：

1. **连通分量（Connected Component）：** 在无向图中，一个连通分量是指图中的一个极大连通子图，其中的任意两个节点都可以通过图中的边互相到达。连通分量可以理解为无向图的极大连通子图。
2. **强连通分量（Strongly Connected Component）：** 在有向图中，一个强连通分量是指图中的一个极大子图，其中的任意两个节点都可以通过有向路径互相到达。强连通分量可以理解为有向图的极大连通子图。

Kruskal算法是一种用于解决最小生成树（Minimum Spanning Tree，简称MST）问题的贪心算法。给定一个连通的带权无向图，Kruskal算法可以找到一个包含所有顶点的最小生成树，即包含所有顶点且边权重之和最小的树。

以下是Kruskal算法的基本步骤：

1. 将图中的所有边按照权重从小到大进行排序。
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：
    - 选择排序后的边集中权重最小的边。
    - 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。
4. 返回最小生成树的边集作为结果。

Kruskal算法的核心思想是通过不断选择权重最小的边，并判断是否会形成环路来构建最小生成树。算法开始时，每个顶点都是一个独立的连通分量，随着边的不断加入，不同的连通分量逐渐合并为一个连通分量，直到最终形成最小生成树。

实现Kruskal算法时，一种常用的数据结构是并查集（Disjoint Set）。并查集可以高效地判断两个顶点是否在同一个连通分量中，并将不同的连通分量合并。

5.拓扑排序-Kahn算法：维护一个入度为零的表

拓扑排序（Topological Sorting）是对有向无环图（DAG）进行排序的一种算法。它将图中的顶点按照一种线性顺序进行排列，使得对于任意的有向边 (u, v)，顶点 u 在排序中出现在顶点 v 的前面。

拓扑排序可以用于解决一些依赖关系的问题，例如任务调度、编译顺序等。

下面是拓扑排序的一种常见算法，称为**Kahn算法**：

1. 计算每个顶点的入度（Indegree），即指向该顶点的边的数量。
2. 初始化一个空的结果列表 `result` 和一个队列 `queue`。
3. 将所有入度为 0 的顶点加入队列 `queue`。
4. 当队列 `queue` 不为空时，执行以下步骤：
    - 从队列中取出一个顶点 `u`。
    - 将 `u` 添加到 `result` 列表中。
    - 对于顶点 `u` 的每个邻接顶点 `v`，减少 `v` 的入度值。
    - 如果顶点 `v` 的入度变为 0，则将 `v` 加入队列 `queue`。
5. 如果 `result` 列表的长度等于图中顶点的数量，则拓扑排序成功，返回结果列表 `result`；否则，图中存在环，无法进行拓扑排序。

5.排序

- 不论列表是否有序，顺序搜索算法的时间复杂度都是 $O(n)$ 。
- 对于有序列表来说，二分搜索算法在最坏情况下的时间复杂度是 $O(\log n)$
- 基于散列表的搜索算法可以达到常数阶。
- 冒泡排序、选择排序和插入排序都是 $O(n^2)$ 算法。

1.冒泡排序：

逐个交换，每轮确定最后一个值O(n2)

2.选择排序：

每次遍历选择最大值放在最后，n-1轮O(n2)一般比冒泡快

3.插入排序：

维护有序子列表，逐个插入新元素O(n2)

4.希尔排序：递减增量排序

增量从n到1，最后插入排序O(n)~O(n2)之间

5.归并排序：

拆分成1或2个元素，排序，两个列表合并的时候通过比较左边第一个数字大小进行合并O(nlogn)

6.快速排序：

找到一个基准值，例如第一个，对其余列表移动左右指标，左指标找到第一个大于ptvot的，右指标找到第一个小于pivot的，两个互换位置，继续行进，直到只剩一个元素，即为pivot的位置，此时左右分别快排

快排不需要归并排序的额外空间，一般是O(nlogn)但是最坏情况O(n2)