20240514 数算B-12班-笔试（模考）

Updated 1548 GMT+8 May 24, 2024

2024 spring, Complied by Hongfei Yan

说明：一、二、三，我有找到2022年答案。

四、五，是我做的答案，大家看对否。

一．选择题（30 分，每小题 2 分）

1. 双向链表中的每个结点有两个引用域，prev 和 next，分别引用当前结点的前驱与后继，设 p 引用链表中的一个结点，q 引用一待插入结点，现要求在 p 前插入 q，则正确的插入操作为（ D ）。

   A：p.prev=q; q.next=p; p.prev.next=q; q.prev=p.prev; B：q.prev=p.prev; p.prev.next=q; q.next=p; p.prev=q.next; C：q.next=p; p.next=q; p.prev.next=q; q.next=p; **D：p.prev.next=q; q.next=p; q.prev=p.prev; p.prev=q.**

   #假设链表是 $A<->B<->C$，要在 B 前插入 Q，那么会得到 $A<->Q<->B<->C$的链表。这是在结点 B 前插入 Q 结点的步骤：

   1. `p.prev.next = q;` 这一步是把 p 的前一个结点的 next 指针指向 q。在例子中，是把 A 的 next 指针指向 Q。
   2. `q.next = p;` 这一步是把 q 的 next 指针指向 p。在例子中，是把 Q 的 next 指针指向 B。
   3. `q.prev = p.prev;` 这一步是把 q 的 prev 指针指向 p 的前一个结点。在例子中，是把 Q 的 prev 指针指向 A。
   4. `p.prev = q;` 这一步是把 p 的 prev 指针指向 q。在例子中，是把 B 的 prev 指针指向 Q。

2. 给定一个 N 个相异元素构成的有序数列，设计一个递归算法实现数列的二分查找，考察递归过程中栈的使用情况，请问这样一个递归调用栈的最小容量应为（ D ）。 A：N B：N/2 C：$\lceil \log_{2}(N) \rceil$ **D：$\lceil \log_{2}(N+1) \rceil$**

   #当N=1时，函数调用用到1次栈。

   #当N=2时，需要两个帧：二分查找，只能往一侧走，所以是函数调用+递归1次，等于2。

   #当N=3是，因为是二分查找，中间节点已经比完不用考虑，情况同N=2。

   #当N=4时，第一次递归二分查找，剩下 1 2 4；按照最坏情况考虑，1 2 还需要2次。结果是3。

3. 数据结构有三个基本要素:逻辑结构、存储结构以及基于结构定义的行为(运算)。下列概念中(B)属于存储结构。 A:线性表 **B:链表** C:字符串 D:二叉树

   #在这些选项中，有些描述的是数据的逻辑结构，而有些是存储结构。逻辑结构指的是数据对象中数据元素之间的相互关系，而存储结构是指数据结构在计算机中的表示（也就是内存中的存储形式）。

   A: **线性表** - 这是一种逻辑结构，它描述元素按线性顺序排列的规则。 B: 链表 - 这是一种存储结构，它是线性表的链式存储方式，通过节点的相互链接来实现。

正确答案是 B: 链表，因为它指的是数据的物理存储方式，即内存中的链式存储结构。

4. 为了实现一个循环队列（或称环形队列），采用数组 Q[0..m-1]作为存储结构,其中变量 rear 表示这个循环队列中队尾元素的实际位置，添加结点时按 rear=(rear+1) % m 进行指针移动，变量length 表示当前队列中的元素个数，请问这个循环队列的队列首位元素的实际位置是（ B ）。 A：rear-length **B：(1+rear+m-length) % m** C：(rear-length+m) % m D：m-length

   #length = rear - head + 1，再对环形队列的特点做调整，得到B。

5. 给定一个二叉树，若前序遍历序列与中序遍历序列相同，则二叉树是（ D ）。 A：根结点无左子树的二叉树 B：根结点无右子树的二叉树 C：只有根结点的二叉树或非叶子结点只有左子树的二叉树 **D：只有根结点的二叉树或非叶子结点只有右子树的二叉树**

   #因为在前序遍历中，根节点总是首先访问的，而在中序遍历中，根节点必然在中间。

6. 用 Huffman 算法构造一个最优二叉编码树，待编码的字符权值分别为{3，4，5，6，8，9，11，12}，请问该最优二叉编码树的带权外部路径长度为（ B ）。（补充说明：树的带权外部路径长度定义为树中所有叶子结点的带权路径长度之和；其中，结点的带权路径长度定义为该结点到树根之间的路径长度与该结点权值的乘积）（ B ） A：58 **B：169** C：72 D：18

   #为了构造哈夫曼树，我们遵循一个重复的选择过程，每次选择两个最小的权值创建一个新的节点，直到只剩下一个节点为止。我们可以按照以下步骤操作：

   1. 将给定的权值排序：{3, 4, 5, 6, 8, 9, 11, 12}。
   2. 选择两个最小的权值：3 和 4，将它们组合成一个新的权值为 7 的节点。

      现在权值变为：{5, 6, 7, 8, 9, 11, 12}。
   3. 再次选择两个最小的权值：5 和 6，将它们组合成一个新的权值为 11 的节点。

      现在权值变为：{7, 8, 9, 11, 11, 12}。
   4. 选择两个最小的权值：7 和 8，将它们组合成一个新的权值为 15 的节点。

      现在权值变为：{9, 11, 11, 12, 15}。
   5. 选择两个最小的权值：9 和 11，将它们合并成一个新的权值为 20 的节点。

      现在权值变为：{11, 12, 15, 20}。
   6. 选择两个最小的权值：11 和 12，合并成一个新的权值为 23 的节点。

      现在权值变为：{15, 20, 23}。
   7. 选择两个最小的权值：15 和 20，合并成一个新的权值为 35 的节点。

      现在权值变为：{23, 35}。
   8. 最后，合并这两个节点得到根节点，权值为 23 + 35 = 58。

   现在我们可以计算哈夫曼树的带权外部路径长度（WPL）。

```
          (58)
        /      \
      (23)       (35)
      /  \       /   \
    (11)(12)  (20)    (15)
             / \       / \
           (9)(11)   (7)(8)
              / \   / \
             (5)(6)(3) (4)
```

现在让我们计算每个叶子节点的带权路径长度：

- 权值 3 的节点路径长度为 4，WPL部分为 3 * 4 = 12。
- 权值 4 的节点路径长度为 4，WPL部分为 4 * 4 = 16。
- 权值 5 的节点路径长度为 4，WPL部分为 5 * 4 = 20。
- 权值 6 的节点路径长度为 4，WPL部分为 6 * 4 = 24。
- 权值 9 的节点路径长度为 3，WPL部分为 9 * 3 = 27。
- 权值 8 的节点路径长度为 3，WPL部分为 8 * 3 = 24。
- 权值 11 的节点路径长度为 2，WPL部分为 11 * 2 = 22。
- 权值 12 的节点路径长度为 2，WPL部分为 12 * 2 = 24。

将所有部分的 WPL 相加，我们得到整棵哈夫曼树的 WPL：

WPL = 12 + 16 + 20 + 24 + 27 + 24 + 22 + 24 = 169

7. 假设需要对存储开销 1GB (GigaBytes) 的数据进行排序，但主存储器（RAM）当前可用的存储空间只有 100MB (MegaBytes)。针对这种情况，（ B ）排序算法是最适合的。 A：堆排序 **B：归并排序** C：快速排序 D：插入排序

   #对于这种情况，最适合的排序算法是归并排序（B）。

   归并排序是一种外部排序算法，它的主要思想是将数据分成较小的块，然后逐步合并这些块以获得有序的结果。由于主存储器的可用空间有限，归并排序非常适合这种情况，因为它可以在有限的主存中进行部分排序，并将排序好的部分写入外部存储（磁盘）中。然后再将不同部分进行合并，直到得到完全排序的结果。

   堆排序（A）通常需要对整个数据集进行排序，因此不适合主存储器有限的情况。

   快速排序（C）通常是一种原地排序算法，它需要频繁地交换数据，这可能导致频繁的磁盘访问，不适合主存储器有限的情况。

   插入排序（D）的时间复杂度较低，但它需要频繁地移动数据，这可能导致频繁的磁盘访问，也不适合主存储器有限的情况。

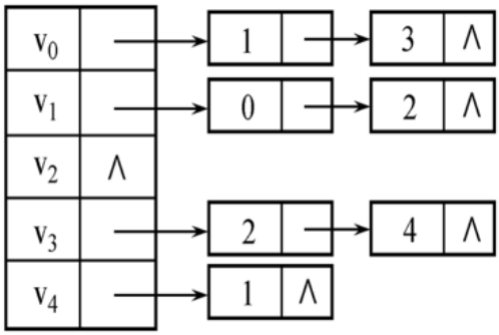
8. 已知一个无向图 G 含有 18 条边，其中度数为 4 的顶点个数为 3，度数为 3 的顶点个数为 4，其他顶点的度数均小于 3，请问图 G 所含的顶点个数至少是（ C ）。 A: 10 B: 11 **C: 13** D: 15

   #一条边贡献两个度数

9. 给定一个无向图 G，从顶点 V0 出发进行无向图 G 的深度优先遍历，访问的边集合为：{(V0,V1),(V0,V4),(V1,V2), (V1,V3), (V4,V5), (V5,V6)}，则下面哪条边（C）不能出现在 G 中？ A: (V0, V2) B: (V4, V6) **C: (V4, V3)** D: (V0, V6)

   #在生成树中，跨两颗子树的边会与DFS过程产生矛盾。

10. 已知一个有向图 G 的邻接入边表（或称逆邻接表）如下图所示，从顶点 v0 出发对该图 G 进行深度 优先周游，得到的深度优先周游结点序列为（A）。 A： V0V1V4V3V2 B：V0V1V2V3V4 C：V0V1V3V2V4. D：V0V2V1V3V4



11. 若按照排序的稳定性和不稳定性对排序算法进行分类，则（D）是不稳定排序。 A：冒泡排序 B：归并排序 C：直接插入排序 **D：希尔排序**

    #根据排序算法的稳定性，如果需要选择一个不稳定排序算法，选项D：希尔排序是正确的选项。

    稳定排序算法是指，当有两个相等的元素A和B，且在排序前A出现在B的前面，在排序后A仍然会出现在B的前面。而不稳定排序算法则无法保证这种相对顺序。

    冒泡排序（A）和直接插入排序（C）都属于稳定排序算法，它们在比较和交换元素时会考虑相等元素的顺序关系。

    归并排序（B）是一种稳定排序算法，它通过分治的思想将待排序的序列划分为较小的子序列，然后逐步合并这些子序列并保持相对顺序。

    希尔排序（D）是一种不稳定排序算法，它使用间隔序列来对数据进行分组，然后对每个分组进行插入排序。在插入排序的过程中，相等元素的顺序可能会发生变化。

12. 以下（C）分组中的两个排序算法的最坏情况下时间复杂度的大 O 表示相同。 A：快速排序和堆排序 B：归并排序和插入排序 C：**快速排序和选择排序** D：堆排序和冒泡排序

    #选项C：快速排序和选择排序中的两个排序算法的最坏情况下时间复杂度的大 O 表示相同。

    快速排序和选择排序都属于不同的排序算法，但它们的最坏情况下的时间复杂度都是O(n^2)。

    快速排序的最坏情况下时间复杂度发生在每次选择的基准元素都划分出了一个很小的子序列，使得递归的深度达到了n，导致时间复杂度为O(n^2)。

    选择排序的最坏情况下时间复杂度发生在每次选择最小（或最大）元素时，需要遍历未排序部分的所有元素，导致时间复杂度为O(n^2)。

13. 设结点 X 和 Y 是二叉树中任意的两个结点，在该二叉树的先根周游遍历序列中 X 出现在 Y 之前，而在其后根周游序列中 X 出现在 Y 之后，则 X 和 Y 的关系是（C）。 A：X 是 Y 的左兄弟 B：X 是 Y 的右兄弟 **C：X 是 Y 的祖先** D：X 是 Y 的后裔

#如果X和Y分属不交的子树，那么他们在先根和后根周游中的前后关系是一致的。出现反转的话，他们只能是祖先/后裔关系。

14. 考虑一个森林 F，其中每个结点的子结点个数均不超过 2。如果森林 F 中叶子结点的总个数为 L，度数为 2 结点（子结点个数为 2）的总个数为 N，那么当前森林 F 中树的个数为（ C ）。 A：L-N-1 B：无法确定 **C：L-N** D：N-L

    #一棵树中，叶节点比二度节点数量多一。

15. 回溯法是一类广泛使用的算法，以下叙述中不正确的是（ C ）。 A：回溯法可以系统地搜索一个问题的所有解或者任意解 B：回溯法是一种既具备系统性又具备跳跃性的搜索算法 **C：回溯算法需要借助队列数据结构来保存从根结点到当前扩展结点的路径** D：回溯算法在生成解空间的任一结点时，先判断当前结点是否可能包含问题的有效解，如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向祖先结点回溯

二．判断

（10 分，每小题 1 分；对填写“Y”, 错填写“N”）

1. （ Y ）对任意一个连通的、无环的无向图，从图中移除任何一条边得到的图均不连通。
2. （ Y ）给定一棵二叉树，前序周游序列和中序周游序列分别是 HGEDBFCA 和 EGBDHFAC 时，其后序周游序列必是 EBDGACFH。
3. （ N ）假设一棵二叉搜索树的结点数值在 1 到 1000 之间，现在查找数值为 363 的结点。以下三个序列皆有可能是查过的序列：A). 2, 252, 401, 398, 330, 344, 397, 363; B). 925, 202, 911, 240, 912, 245,363; C). 935, 278, 347, 621, 299, 392, 358, 363。

    #911子树左子节点240的右子节点912，不能比911大

4. （ N ）构建一个含 N 个结点的（二叉）最小值堆，建堆的时间复杂度大 O 表示为 $O(Nlog_2N)$。
5. （ Y ）队列是动态集合，其定义的出队列操作所移除的元素总是在集合中存在时间最长的元素。
6. （ N ）任一有向图的拓扑序列既可以通过深度优先搜索求解，也可以通过宽度优先搜索求解。

    #必须是DAG

7. （ Y ）对任一连通无向图 G，其中 E 是唯一权值最小的边，那么 E 必然属于任何一个最小生成树。
8. （ N ）对一个包含负权值边的图，迪杰斯特拉(Dijkstra)算法能够给出最短路径问题的正确答案。
9. （ N ）分治算法通常将原问题分解为几个规模较小但类似于原问题的子问题，并要求算法实现写成某种递归形式，递归地求解这些子问题，然后再合并这些子问题的解来建立原问题的解。

    #递归的求解子问题，最后子问题足够小可以直接求解，形式不一定是递归。

    #分治和动态规划都是将问题分解为子问题，然后合并子问题的解得到原问题的解。但是不同的是，分治法分解出的子问题是不重叠的，因此分治法解决的问题不拥有重叠子问题，而动态规划解决的问题拥有重叠子问题。

10. （ Y ）考察某个具体问题是否适合应用动态规划算法，必须判定它是否具有最优子结构性质。

#如果一个问题的最优解可以由其子问题的最优解有效地构造出来，那么称这个问题拥有**最优子结构(Optimal Substructure)**。最优子结构保证了动态规划中原问题的最优解可以由子问题的最优解推导而来。因此，一个问题必须拥有最优子结构，才能使用动态规划去解决。例如数塔问题中，每一个位置的 dp 值都可以由它的两个子问题推导得到。 至此，重叠子问题和最优子结构的内容已介绍完毕。需要指出，**一个问题必须拥有重叠子问题和最优子结构，才能使用动态规划去解决。**

三．填空（20 分，每题 2 分）

1. 目标串长是 n，模式串长是 m，朴素模式匹配算法思想为：从目标串第一个字符开始，依次与模式串字符匹配；若匹配失败，则尝试匹配的目标串起始字符位置往后移一位，重新开始依次和模式串字符匹配；……. ；直到匹配成功或遍历完整个目标串为止。则该算法中字符的最多比较次数是 _ _ _ _ （使用大 O 表示法）。O(mn)

2. 在一棵含有 n 个结点的树中，只有度（树节点的度指子节点数量）为 k 的分支结点和度为 0 的终端（叶子）结点，则该树中含有的终端（叶子）结点的数目为： _ _ _ _ 。n - (n-1)/k

   #设叶节点x个，分支节点n-x个，树的性质有节点数等于所有节点度数和+1，(n-x)*k + 1 = n, x = n - (n-1)/k

3. 对一组记录进行非递减排序，其关键码为[46，70，56，38，40，80]，则利用快速排序的方法，以第一个记录为基准得到的第一次划分结果为 _ _ _ _ 。

   #[38, 40, 46, 56, 70,80]。基本知识点，由于算法有多种实现，只需要46把数列分开了就行

4. 对长度为 3 的顺序表进行查找，若查找第一个元素的概率为 1/2，查找第二个元素的概率为 1/4，查找第三个元素的概率为 1/8，则执行任意查找需要比较元素的平均个数为 _ _ _ _ 。

   #$1*(1/2) + 2*(1/4) + 3*(1/8) + 3*(1/8) = 1.75$, 还有1/8的失败查询概率。

5. 设有一组记录的关键字为{19，14，23，1，68，20，84，27，55，11，10，79}，用链地址法（拉链法）构造散列表，散列函数为 H(key)=key MOD 13, 散列地址为 1 的链中有 _ _ _ _ 个记录。4

6. 删除长度为 n 的顺序表的第 i 个数据元素需要移动表中的 _ _ _ _ 个数据元素，（1<=i<=n）。n - i

7. 已知以数组表示的小根堆为[8，15，10，21，34，16，12]，删除关键字 8 之后需要重新建堆，在此过程中，关键字的比较次数是 _ _ _ _ _ 。3

8. 在广度优先遍历、拓扑排序、求最短路径三种算法中，可以判断出一个有向图是否有环（回路）的是 _ _ _ _ 。拓扑排序

   #求最短路径和广度优先遍历中，发现重复访问的节点时，并不知道该节点是否当前节点的祖先拓扑排序有一种实现是用入度来判断，输出入度为0的节点。

9. 有 n（n>=2）个顶点的有向强连通图最少有 _ _ _ _ _ 条边。n

10. 若栈 S1 中保存整数，栈 S2 中保存运算符，函数 F() 依次执行下述各步操作： ①从 S1 中依次弹出两个操作数 a 和 b (先弹出 a，再弹出 b)； ②从 S2 中弹出一个运算符 op; ③执行相应的运算 b op a; ④将运算结果压入 S1 中。 假定 S1 中的操作数依次是 5，8，3，2（2 在栈顶），S2 中的运算符依次是*，-，//（//在栈顶）。调用三次 F() 后，S1 栈顶保存的值是 _ _ _ _ 。35

四．简答（3题，共20分）

1. （7 分）试用 Dijkstra 算法求出下图中顶点 1 到其余各顶点的最短路径，写出算法执行过程中各步的状态，填入下表。

```
graph LR
    1((1)) --> |30|2((2)); 1--> |10|5((5)); 1 --> |60|4((4))
    2((2)) --> |20|3((3))
    3((3)) --> |15|4((4))
    5((5)) --> |15|2; 5 --> |7|6((6))
    6((6)) --> |3|2; 6 --> |16|3; 6 --> |8|7
    7((7)) --> |6|3; 7 --> |3|4; 7 --> |10|8((8))
    8 --> |6|4
```

顶点1到其他顶点的最短路径长度

所选顶点	U(已确定最短路径的顶点集合)	T(未确定最短路径的顶点集合)	2	3	4	5	6	7	8
初态	{1}	{2, 3, 4, 5, 6, 7, 8}	30	∞	60	10	∞	∞	∞
1	{1, 5}	{2, 3, 4, 6, 7, 8}	25	∞	60	10	17	∞	∞
5	{1, 5, 6}	{2, 3, 4, 7, 8}	20	33	60	10	17	25	∞
6	{1, 5, 6, 2}	{3, 4, 7, 8}	20	33	60	10	17	25	∞
2	{1, 5, 6, 2, 7}	{3, 4, 8}	20	31	28	10	17	25	35
第5步	{1, 5, 6, 7, 2, 4}	{3, 8}	20	31	28	10	17	25	35
4	{1, 5, 6, 7, 2, 4, 3}	{8}	20	31	28	10	17	25	35
3	{1, 5, 6, 7, 2, 4, 3, 8}	{}	20	31	28	10	17	25	35

2. （6 分）给定一组记录的关键码集合为{18，73，5，10，68，99，27}，回答下列 3 个问题： a) 画出按照记录顺序构建形成的二叉排序（搜索）树（2 分）； b) 画出删除关键码为 73 后的二叉排序树（2 分）。 c) 画出令原关键码集合（未删除 73 前）查询效率最高的最优二叉排序树（仅需考虑关键码查询成功时的效率，且集合内每个关键码被查询概率相等）（2 分）。

#二叉排序树是一种特殊的二叉树，它具有以下性质：

1. 对于树中的每个节点，其左子树中的所有节点的关键码值小于该节点的关键码值。
2. 对于树中的每个节点，其右子树中的所有节点的关键码值大于该节点的关键码值。
3. 左右子树也是二叉排序树。

在构建二叉排序树时，我们按照以下原理：

- 从根节点开始，依次遍历关键码集合中的每个关键码。
- 对于每个关键码，如果树为空，则将关键码作为树的根节点。如果树不为空，则按照二叉排序树的性质，找到适当的位置插入关键码，以保持二叉排序树的性质不变。

具体来说，在插入关键码时，我们从根节点开始，逐级比较关键码值，并根据比较结果决定向左子树或右子树移动，直到找到合适的插入位置。如果待插入的关键码已经存在于树中，则不执行任何操作（可以视为树中已经存在该关键码）。

这样，通过按照关键码集合中记录的顺序依次插入节点，就可以构建出一个二叉排序树。

让我们逐个回答这些问题：

a) 二叉排序树构建过程如下：

```
        18
       /  \
      5    73
       \   / \
       10 68 99
          /
         27
```

b) 删除关键码为 73 后的二叉排序树：

```
      18                                    18
     /  \                                 /    \
    5    68          或者                5      99
     \   / \                              \     /
     10 27 99                            10  68
                                                /
                                               27
```

c) 查询效率最高的最优二叉排序树：

由于关键码集合内每个关键码被查询概率相等，最优二叉排序树应该是平衡树，即左右子树高度差不超过1。因此，我们可以构建一个高度平衡的二叉排序树。

```
        27
       /  \
      10   73
     /  \  / \
    5   18 68 99
```

这是一棵高度平衡的二叉排序树，查询效率最高，因为平衡树的查找时间复杂度是 O(log n)。

3. （7 分）奇偶交换排序如下所述：对于原始记录序列 {a1, a2, a3, ..., an}，第一趟对所有奇数 i，将ai 和 ai+1 进行比较，若 ai > ai+1，则将二者交换；第二趟对所有偶数 i；第三趟对所有奇数 i;第四趟对所有偶数 i，...，依次类推直到整个记录序列有序为止。代码如下：

```
def ExSort(a, n):  # a[1..n]为待排序记录，n为记录数目

    change1 = change2 = True  # 标志变量，bool型
    if n <= 0:
        return "Error"
    while (change1 or change2):

        change1 = False  # 奇数，
        for i in range(1, n, 2):
            if a[i] > a[i+1]:
               a[i], a[i+1] = a[i+1], a[i]
               change1 = True

        if not change1 and not change2:
            break

        change2 = False  # 偶数
        for i in range(2, n, 2):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
                change2 = True
```

a) 请写出序列 {18，73，5，10，68，99，27，10} 在前 4 趟排序中每趟排序后的结果。（2 分） b) 奇偶交换排序是否是稳定的排序？（1 分） c) 在序列为初始状态为“正序”和“逆序”两种情况下，试给出序列长度为 n 的情况下，排序过程所需进行的关键码比较次数和记录的交换次数？（4 分）

#c)

- 正序情况下： 关键码比较次数：n//2 + (n-1)//2，分奇偶情况化简下来就是 n-1。 即每两趟需要 n-1 次比较。 记录交换次数：0。由于序列已经有序，不需要进行交换，
- 逆序情况下： 逆序尝试下来应该是在第n趟排序后实现正序，然后再经过两趟确认。每两趟要n-1次比较，总计需要 n/2 个两趟；再两趟确认n-1次。所以总的比较次数是前两个数相乘，加上第三个数，即 n(n-1)/2 + (n-1)。

  关键码比较次数：n(n-1)/2 + (n-1)

  记录交换次数：n(n-1)/2，因为需要俩俩比较确定位置。

a) 序列 {18, 73, 5, 10, 68, 99, 27, 10} 在前4趟排序中每趟排序后的结果如下：

```
def ExSort(a, n):  # a[1..n]为待排序记录，n为记录数目
    change1 = change2 = True  # 标志变量，bool型
    if n <= 0:
        return "Error"
    cnt = 0
    while (change1 or change2):
        change1 = False  # 奇数，
        for i in range(1, n, 2):
            if a[i] > a[i+1]:
               a[i], a[i+1] = a[i+1], a[i]
```

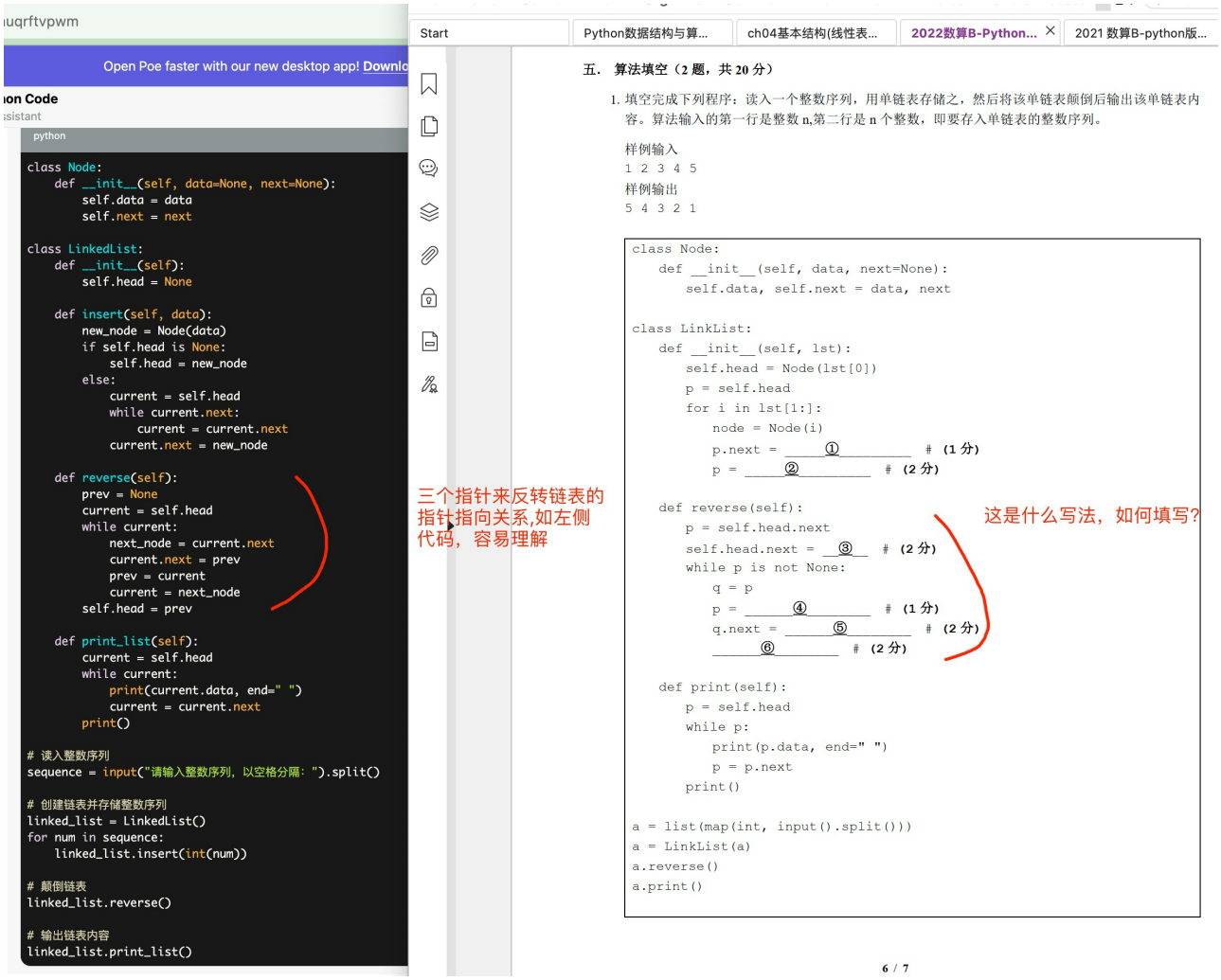
```
                change1 = True

        cnt += 1; print(f"pass {cnt}: {a[1:]}")
        if not change1 and not change2:
            break

        change2 = False  # 偶数
        for i in range(2, n, 2):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
                change2 = True

        cnt += 1; print(f"pass {cnt}: {a[1:]}")
        if cnt == 4:
            break

# 题面是奇数第一趟，偶数是第二趟，这也没有都都比较，才一半，怎么算一趟？题面有问题吧
a = [0] + [18, 73, 5, 10, 68, 99, 27, 10]
ExSort(a, len(a)-1)
"""
pass 1: [18, 73, 5, 10, 68, 99, 10, 27]
pass 2: [18, 5, 73, 10, 68, 10, 99, 27]
pass 3: [5, 18, 10, 73, 10, 68, 27, 99]
pass 4: [5, 10, 18, 10, 73, 27, 68, 99]
"""
```

b) 奇偶交换排序是稳定的排序。稳定排序是指如果两个元素相等，在排序后它们的相对顺序仍然保持不变。奇偶交换排序在交换过程中只涉及相邻的两个元素，因此相等元素之间的相对顺序不会改变。

五．算法填空（2题，共20分）

1. 填空完成下列程序：读入一个整数序列，用单链表存储之，然后将该单链表颠倒后输出该单链表内容。算法输入的一行是 n 个整数，即要存入单链表的整数序列。

   样例输入 1 2 3 4 5 样例输出 5 4 3 2 1




```python
class Node:
    def __init__(self, data, next = None):
        self.data, self.next = data, next

class LinkedList:
    def __init__(self, lst):
        self.head = Node(lst[0])
        p = self.head
        for i in lst[1:]:
            p.next = Node(i)     # 等号右侧填空（1分）
            p = p.next                  # 等号右侧填空（2分）

    def reverse(self): # 把head当pre用，天才 said by 胡睿诚
        p = self.head.next
        self.head.next = None    # 等号右侧填空（2分）
        while p is not None:
            q = p
            p = p.next                  # 等号右侧填空（1分）
            q.next = self.head  # 等号右侧填空（2分）
            self.head = q           # 留空行，此行代码需要填写（2分）

    def reverse_3p(self): # 常规思维：三个指针来反转链表的指针指向关系
        prev = None
        current = self.head
```

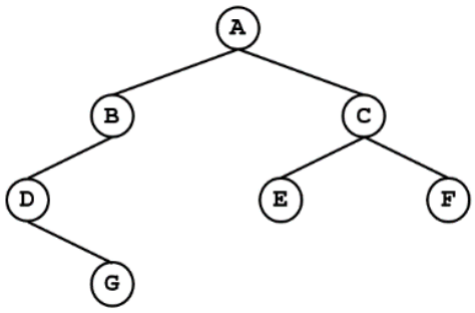
```
            while current:
                next_node = current.next
                current.next = prev
                prev = current
                current = next_node
            self.head = prev

        def print_list(self):
            p = self.head
            while p:
                print(p.data, end=" ")
                p = p.next
            print()

    #a = list(map(int, input().split()))
    a = [1, 2, 3, 4, 5]
    b = a.copy()
    a = LinkedList(a)
    b = LinkedList(b)
    a.reverse()
    b.reverse_3p()
    a.print_list()
    b.print_list()
    """
    5 4 3 2 1
    5 4 3 2 1
    """

    """
    5 4 3 2 1
    """
```

`reverse_3p`这段代码是用来反转一个链表的指针指向关系的。下面是对代码的逐行解释：

1. `prev = None`: 初始化一个变量`prev`，用于保存当前节点的前一个节点，初始值为`None`。
2. `current = self.head`: 初始化一个变量`current`，指向链表的头节点。
3. `while current:`: 进入一个循环，只要`current`不为空（即还有节点未遍历）就继续执行。
4. `next_node = current.next`: 将`current`节点的下一个节点保存到`next_node`中，以备后续使用。
5. `current.next = prev`: 将`current`节点的指针指向前一个节点`prev`，实现了反转指针的方向。
6. `prev = current`: 将`prev`更新为当前节点`current`，为下一轮循环做准备。
7. `current = next_node`: 将`current`更新为下一个节点`next_node`，继续遍历链表。
8. `self.head = prev`: 最后将链表的头节点更新为反转后的链表的头节点`prev`，完成整个链表的反转。

这段代码使用了三个指针：`prev`、`current`、`next_node`，通过不断更新它们的指向关系，实现了链表的反转。

2. 填空完成下列程序：输入一棵二叉树的扩充二叉树的先根周游（前序遍历）序列，构建该二叉树，并输出它的中根周游（中序遍历）序列。这里定义一棵扩充二叉树是指将原二叉树中的所有空引用增加一个表示为@的虚拟叶结点。譬如下图所示的一棵二叉树， 输入样例： ABD@G@@@CE@@F@@ 输出样例： DGBAECF




```
s = input()
ptr = 0

class BinaryTree:
    def __init__(self, data, left=None, right=None):
        self.data, self.left, self.right = data, left, right

    def addLeft(self, tree):
        self.left = tree

    def addRight(self, tree):
        self.right = tree

    def inorderTraversal(self):
        if self.left:
            self.left.inorderTraversal()    # (1分)
        print(self.data, end="")
        if self.right:
            self.right.inorderTraversal()   # (1分)

def buildTree():
    global ptr
    if s[ptr] == "@":
        ptr += 1
        return None             # (2分)
    tree = BinaryTree(s[ptr])   # (1分)
    ptr += 1
    tree.addLeft(buildTree())   # (2分)
    tree.addRight(buildTree())  # (2分)

    return tree

tree = buildTree()  # (1分)
tree.inorderTraversal()

"""
sample input:
```

```
ABD@G@@@CE@@F@@

sample output:
DGBAECF
"""
```

笔试中，对于程序阅读理解，要求还是挺高的。因为AC的代码通常有多种写法，如果考出来写的不规范代码，就有点难受。例如：上面程序，递归程序带着全局变量，难受。

较好的写法是：

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def buildTree(preorder):
    if not preorder:
        return None

    data = preorder.pop(0)
    if data == "@":
        return None

    node = TreeNode(data)
    node.left = buildTree(preorder)
    node.right = buildTree(preorder)

    return node

def inorderTraversal(node):
    if node is None:
        return []

    result = []
    result.extend(inorderTraversal(node.left))
    result.append(node.data)
    result.extend(inorderTraversal(node.right))

    return result

preorder = input()
tree = buildTree(list(preorder))

inorder = inorderTraversal(tree)
print(''.join(inorder))

"""
sample input:
ABD@G@@@CE@@F@@
```

```
sample output:
DGBAECF
"""
```