诚信应考,考试作弊将带来严重后果!

考试中心填写:

___年___月___日
考 试 用

湖南大学课程考试试卷（闭卷）

课程名称：**算法分析与设计**；课程编码：CS06256 试卷编号：B；考试时间：120 分钟

题 号	一	二	三	四	五	六	总分
应得分							
实得分							
评卷人							

湖南大学课程考试试卷

专业班级：

学号：

姓名：

湖南大学教务处考试中心

装订线（题目不得超过此线）

一、判断题（3*8=24 分）

对于下面每一个问题，判定其真（T）或者假（F），并给出你的解释

1、 对于所有正的 f(n)、g(n)和 h(n)，如果 f(n) = O(g(n)) 而且 f(n) = Ω(h(n))，那么 g(n) + h(n) =Ω(f(n))

答案：对. f(n) = O(g(n)) ⇒ g(n) = Ω(f(n)).

2、 任何一个 n 个点的二叉搜索树高度都是 O(logn)。

答：错，最好状态下是 O(log n)，最坏情况完全可能是 O(n)

3、 无论输入数据是否是已经排好序的，计数排序的运行时间都一样。

答：对，任意排序输入数据下计数排序的运行时间都和最坏情况一样。

4、 数组[17, 12, 5, 10, 9, 4]是一个最大堆。

答：对

5、 一个无向图的不同深度优先搜索森林含有相同数量的树。

答：对。在无向图中，图的每个连通分量在深度优先搜索（DFS）中将是一棵单独的树。

6、 哪怕图上有权重和为负的环路，Dijkstra 算法也一定会终止。

答：对，有负权环路，Dijkstra 算法无法得正确解，但是还是会结束

7、 所有有向无环图都只有一个拓扑序。

答：错，某些优先级约束可能未指定，并且给定的有向无环图（DAG）可能存在多种排序方式。例如，一个图 G = (V,E) =({a, b, c}, {(a, b), (a, c)}）具有有效的拓扑排序 [a, b, c] 或 [a; c; b]。。

8、 如果一个有向图 G 是有环的但是去掉一条边就是没有环的，那么 G 上做深度优先搜索就只会有一条后向边。

答：对，树的有一个定义就是连通但无环，有环但是去掉一条边就没环，说明这个图也就比树多一条边

二、计算题（4*5=20 分）

求解下列递归式

1、$T(n) = 25 \times T(\frac{n}{5}) + n$

答案：根据主定理：$T(n)=O(n^2)$

2、$T(n) = T(\frac{26n}{27}) + 1$

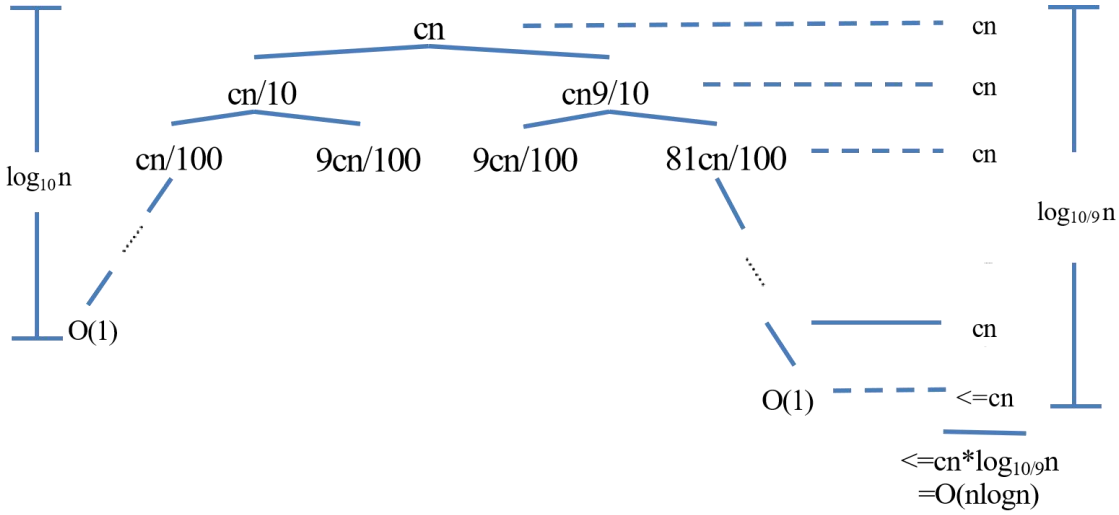
答案：根据主定理：$T(n)=O(\log n)$

3、$T(n) = 15 \times T(\frac{n}{16}) + n\log n$

答案：根据主定理：$T(n)=O(n \times \log n)$

4、$T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + n$

答案：根据下面递归树，$T(n)=O(n \times \log n)$



5、$T(n) = T(n-1) + n$

答案：根据递归公式，T(n)=T(n-1)+n=T(n-2)+(n-1)+n=T(n-3)+(n-2)+(n-1)+n = …… = 1+2+3+……+(n-1)+n=n(n+1)/2=O(n^2)

三、欧拉回路（16 分）

考强连通有向图 G=(V, E)中的一个欧拉回路是指一条遍历图 G 中每条边恰好一次的环路。不过，这条环路可以多次访问同一个点。

a) 证明：图 G 中有一条欧拉回路当且经当对于图中的每个点 v，有 v 的入度等于出度。

b) 给出一个复杂度为 O(E)的算法找出 G 中的一条欧拉回路。

答案：a:

欧拉回路是指在一个图中经过每条边恰好一次并且能够回到起点的闭合路径。要证明一个图 G 中存在欧拉回路的充分必要条件是每个顶点的入度等于出度，我们可以分两部分来证明：

必要性（如果图 G 中存在欧拉回路，则每个顶点的入度等于出度）：

假设图 G 中存在一条欧拉回路，那么这条回路会经过图中的每一条边一次，并且回到起点。由于每条边都被经过一次，每个顶点在欧拉回路中作为起点出发的次数（出度）必然等于作为终点到达的次数（入度）。这是因为每次一个顶点作为起点出发，在欧拉回路中必然会再次回到这个顶点，完成一次闭环。因此，每个顶点的入度必须等于出度。

充分性（如果图 G 中每个顶点的入度等于出度，则图 G 中存在欧拉回路）：

假设图 G 中每个顶点的入度都等于出度。这意味着每个顶点都有相同数量的边进入和离开，从而保证了可以从任何一个顶点出发，沿着边行进，最终回到出发点，同时确保每条边都被经过一次。具体构造欧拉回路的方法是从任意一个顶点出发，沿着任意一条边行进，每经过一条边后，选择下一个顶点继续行进，直到回到出发点。由于每个顶点的入度和出度相等，这样的行进过程不会在任何顶点处停止，最终能够形成一个闭合的回路，即欧拉回路。

综上所述，图 G 中存在欧拉回路的充分必要条件是每个顶点的入度等于出度。

b:

从任意一个起始点 v 开始遍历，直到再次到达点 v，即寻找一个环，这会保证一定可以到达点 v，因为遍历到任意一个点 u，由于其出度和入度相同，故 u 一定存在一条出边，所以一定可以到达 v。将此环定义为 C，如果环 C 中存在某个点 x，其有出边不在环中，则继续以此点 x 开始遍历寻找环 C’，将环 C、C’连接起来也是一个大环，如此往复，直到图 G 中所有的边均已经添加到环中。

数据结构如下：

(1) 使用循环链表 CList 存储当前已经发现的环；

(2) 使用一个链表 L 保存当前环中还有出边的点；

(3) 使用邻接表存储图 G

使用如下的步骤可以确保算法的复杂度为 O(E)：

(1) 将图 G 中所有点入 L，取 L 的第一个结点

(2) 直接取其邻接表的第一条边，如此循环往复直到再次到达点 v 构成环 C，此过程中将 L 中无出边的点删除。环 C 与环 CList 合并，只要将 CList 中的点 v 使用环 C 代替即可。

(3) 如果链表 L 为空表示欧拉回路过程结束，否则取 L 的第一个结点，继续步骤(2)

四、0-1 背包问题（15 分）

假设有 n 个物品，每个物品 i 的价值为 v_i，大小为 s_i。包的容量为 S，要求

从这 n 个物品中挑选若干物品装进包中，在所有装进包中物品的大小小于等于包容量 S 的前提下，包中物品总价值最大。

a) 定义子问题：从物品 0、1、……、i - 1、 i 中选取若干物品置于容量为 X 的包中，并获取了最佳收益。这个问题的解记录为 F(i, X)。证明下述递归公式：

$$F(i, X) = \begin{cases} F(i-1, X)，如果 s_i > X \\ \max\{F(i-1, X-s_i) + v_i, F(i-1, X)\}，如果 s_i \le X \end{cases}$$

b) 基于 a 问的性质，写出相应动态规划算法伪代码。

c) 根据 b 问的递归公式，利用动态规划算法求解 0-1 背包问题，我们会用到一个 O(n×S)的数组，这里记为 Knapsack，且 Knapsack(i, j)=F(i,X)。这里假设有 4 个物品，背包容量限制是 8。其中，物品 1 的 v_1=3、s_1=2；物品 2 的 v_2=4、s_2=3；物品 3 的 v_3=5、s_3=4；物品 4 的 v_4=6、s_4=5。画出算法终止时刻的数组 Knapsack，并给出相应的解；

答案：a：

这个公式的含义是，对于每个子问题，我们比较“不选择当前物品”和“选择当前物品”这两种策略下的最大价值，取两者之间的较大值作为当前子问题的解。

对于上述两种情况：

1. **不选择第 i 件物品：这种情况下，问题退化为只从前 i-1 件物品中选择，背包的容量仍然是 X，即 F(i, X) = F(i-1, X)。**
2. **选择第 i 件物品：如果选择了第 i 件物品，那么背包的剩余容量变为 X - s_i，此时能获得的最大价值是第 i 件物品的价值加上剩余容量下从前 i-1 件物品中选择的最大价值，即 F(i, X) = v_i+ F(i-1, X - s_i)，其中 vi 为第 i 件物品的价值。**

需要注意的是，这个递归公式还隐含了两个边界条件：

当 X < s_i 时，即背包容量小于第 i 件物品的大小时，不可能选择这件物品，因此 F(i, X)

= F(i-1, X)。

当 i = 0 时，即没有物品可选或背包容量为 0 时，F(i, X) = 0。

b：基于上述递归公式，我们可以编写 0-1 背包问题的动态规划算法伪代码。该算法使用二维数组 dp[i][j]来存储子问题的解，其中 dp[i][j]表示对于前 i 件物品，在容量为 j 的背包中能够获得的最大价值。

伪代码如下：

```
function knapsack(weights, values, n, capacity):
    dp = array[0..n][0..capacity] of 0
    for i from 1 to n:
        for j from 1 to capacity:
            if weights[i] > j:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = max(dp[i-1][j], values[i] + dp[i-1][j - weights[i]])
    return dp[n][capacity]
```

c：最终答案：

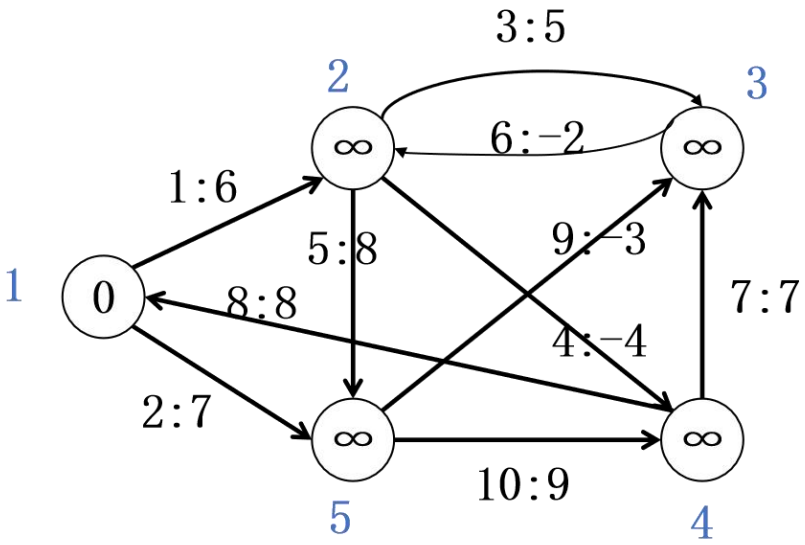
i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

也就是选择的物品是物品 4 和物品 2

五、Bellman-Ford 算法（15 分）

给定一个带权图 G=(V, E, W)和一个源点 s，计算 s 到其他 V 中所有点的距离。这里假设边上的权重可以是负数，因此我们就需要用 Bellman-Ford 算法。

a) 描述 Bellman-Ford 算法的思路和流程，写出伪代码；



b) 对于上图所示的图，以 v_1 为起点，给出 Bellman-Ford 算法按照每条边上编号进行第一轮松弛之后的结果（注意，每条边上的二元对 a:b，表示这是第 a 条边，权重为 b，比如 v_1 指向 v_2 的边上面有 1:6 表示第 1 条边，边权为 6）；

c) 对于上图所示的图，以 v_1 为起点，给出 Bellman-Ford 算法算出的最终起点到所有点距离。

答案：a:

Bellman-Ford 算法是一种用于计算单源最短路径问题的算法，它可以处理图中存在负权重边的情况。算法的核心思想是通过迭代地松弛图中的边来更新顶点的最短路径估计，直到所有边的最短路径都被计算出来。

算法流程如下：

1. 初始化：对于图 G(V, E)，其中 V 是顶点集合，E 是边集合，设置源顶点 s 的最短路径为 0，其余顶点的最短路径为无穷大。

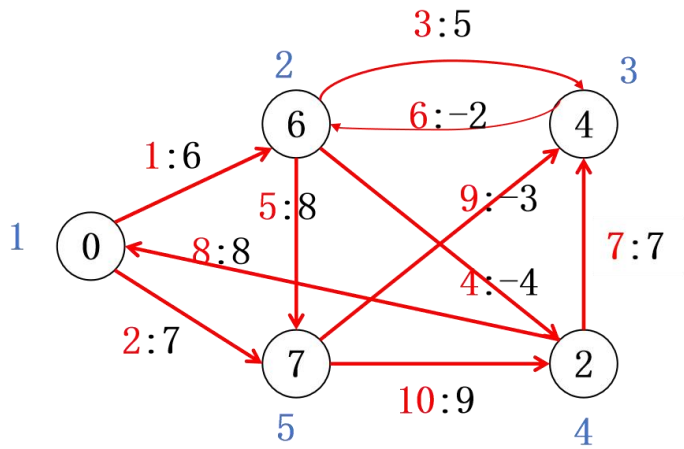
2. 松弛操作：对图中的所有边进行 V-1 次松弛操作。在每次松弛中，遍历所有的边(u, v) ∈ E，如果存在通过边(u, v)可以使从源点 s 到顶点 v 的最短路径变得更短，则更新顶点 v 的最短路径。

3. 检测负权重环：再次遍历所有的边(u, v) ∈ E，如果还能找到可以松弛的边，说明存在负权重环。

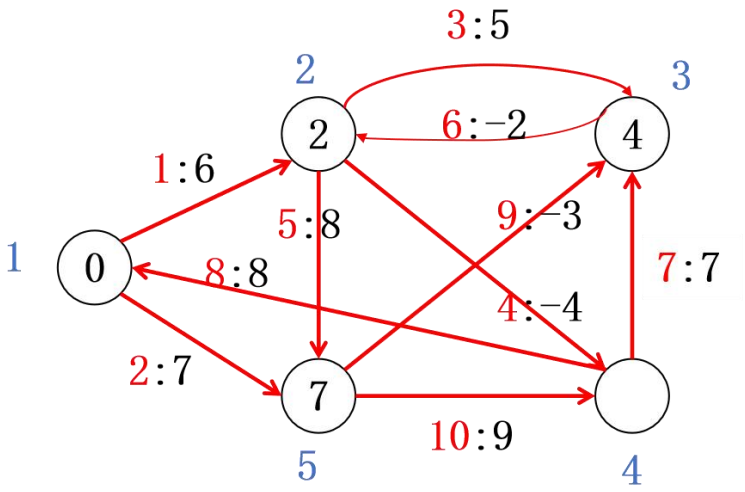
算法伪代码：

```
1  Bellman-Ford(G,w,s)
2    for v in V:
3      d[v]= ∞; p[v]=NULL;
4       d[s]=0;
5   for i from 1 to |V|-1 :
6       for each e (u, v) ∈E:
7           Relax(u,v,w)
8   for each e (u, v) ∈E:
9       if d[v] > d[u] + w(u, v)
10          return FALSE
11    return True
```

b：第一轮迭代结束之后，各个点上结果如下：



c:各个点上最终结果如下：



六、最小平均完成时间调度问题（10 分）

假定给定任务集合 S={a_1, a_2, …, a_n}，其中任务 a_i 在启动后需要 p_i 个时间单位完成。你有一台计算机来运行这些任务，每个时刻只能运行一个任务。令 c_i 表示任务 a_i 的**完成时间**，即任务 a_i 执行完的时间。你的目标是最小化平均完成时间，即最小化$(1/n)\sum_{i=1}^{n} c_i$。例如，假定有两个任务 a_1 和 a_2，p_2=3，p_2=5，如果 a_2 首先运行，然后运行 a_1，则 c_2=5，c_1=8，平均完成时间为(5+8)/2=6.5。如果 a_1 先于 a_2 执行，则 c_1 =3，c_2=8，平均完成时间为(3 + 8)/2 = 5.5。

a. 设计算法，求平均完成时间最小的调度方案。任务的执行都是非抢占的，即一旦 a_i 开始运行，它就持续运行 p_i 个时间单位。证明你的算法能最小化平均完成时间，并分析算法的运行时间。

b. 现在假定任务并不是在任意时刻都可以开始执行，每个任务都有一个**释放时间** r_i，在此时间之后才可以开始。此外假定任务执行是可以**抢占的**（preemption），这样任务可以被挂起，然后再重新开始。例如，一个任务 a_i 运行时间 p_i=6，释放时间 r_i=1，它可能在时刻 1 开始运行，在时刻 4 被抢占。然后在时刻 10 恢复运行，在时刻 11 再次被抢占，最后在时刻 13 恢复运行，在时刻 15 运行完毕。任务 a_i 共运行了 6 个时间单位，但运行时间被分割成三部分。

在此情况下，a_i 的完成时间为 15。设计算法，对此问题求解平均运行时间最小的调度方案。证明你的算法确实能最小化完成时间，分析算法的运行时间。

答案：a:

按照从最小到最大的处理时间对任务进行排序，并按该顺序运行它们。要知道这个贪心解是最优的，首先要观察这个问题表现出最优的子结构:如果我们在一个最优解中运行第一个任务，那么我们通过以最小化平均完成时间的方式运行剩下的任务来获得一个最优解。设o为最优解。设 a 为处理时间最小的任务，b 为在o中运行的第一个任务。设 G 为我们在o中运行 a 和 b 的顺序转换得到的解。这就通过 a 和 b 处理时间的差异，减少了在 a 和 b 之间 a 的完成时间和 G 中所有任务的完成时间。由于其他所有的补全时间保持不变，G 的平均补全时间小于或等于o的平均补全时间，证明贪心解是最优解。运行时间复杂度为O(nlgn)，因为我们必须首先对元素排序。

b:

在不失一般性的前提下，我们假定每个任务都是一个单位时间任务。应用与第(a)部分相同的策略，除非这次我们想要添加到计划旁边的任务还不允许运行，否则我们必须跳过它。由于可能有许多处理时间短但发布时间较晚的任务，所以运行时间复杂度变为o(n^2),因为我们可能要花费O(n)个时间来决定下一步添加哪个任务。