复习 2 综合复习

题量: 19 满分: 510.0

作答时间:2023-12-29 12:45 至 01-07 12:00

一. 单选题（共 2 题）

1. （单选题） 【算法基础】冒泡排序算法的时间复杂度为（ A ）。

O（n^2）

B

O（n）

C

O（nlogn）

D

O（logn）

2. （单选题） 【算法基础】选择排序算法的时间复杂度为（ ）。

A

O（n^2）

B

O（n）

C

O（nlogn）

D

O（logn）

二. 填空题（共 3 题）

3. （填空题） 【算法基础】 nˆ2+10n-1 和 14+5/5+1/nˆ2 的渐进复杂度分别是 O( )和 O( )。（a 的 n 次方请表示为 a^n 的形式）

第 1 空

n^2

第 2 空

n^2

4. （填空题） 【算法基础】j 结合 NP 完全理论，分析旅行售货员问题属于哪一类问题？

第 1 空

5. （填空题） 【算法基础】讨论 O(1)和 O(2)的区别

第 1 空

三. 资料题（共 14 题）

6. （资料题）

【算法基础】如下 2022 美团笔试题，给出程序代码。
小团从某海鲜市场买到了一块 3 核 CPU 来应对他需要进行的高性能计算任务.小团-共有 n 个计算任务,编号分别为 1 到 n.编号为的任务需要运行 a 秒.为了避免进程切换带来的开销,小团只能同时运行三个任务.也就是说他需要将这 n 个任务分成三组，并分别分配到 cpu 的三个核心上现在他想知道完成所有任务至少需要多少秒。
输入描述:
第一行有一个正整数 n(1<=n<=100),代表计算任务的数量
第二行有 n 个大小不超过 1000 的整数,空格隔开,分别代表编号为 1 到 n 的计算任务所需的运行时长
输入的数据保证答案不超过 10000

输出描述:
输出一个整数,代表完成所有任务所需要的时间

第 1 空

7. （资料题）

【分治法】针对最大子段和问题：
（1）请利用分治法设计一个求解算法，针对输出一个序列，求出其最大子段和；给出算法设计方案和核心代码。
（2）利用 master 主定理分析该算法的算法复杂度；
（3）利用蛮力法设计最大子段和的求解思路，给出算法设计方案和核心代码、算法复杂度。

答案：

(1)

设计思路：

将字段一分为二，

递归计算左侧字段的最大和，

递归计算右侧字段的最大和
以及计算中间向左侧和右侧的最大和结果
比较上述三段的最大和，其中最大的为整个字段最终的最大和。

（2）
代码：

```
Int sum =0;
Int MaxSum(int a[],left,right){
    If(left == right) a[left] >0?a[left]:0；     // 递归终止条件，当只剩下一个元素的时候，如果这个元素大于 0，就是最大值；
    else{
    Int med = (left + right)/2;
    Int leftMaxsum = MaxSum(a,left,med);
    Int rightMaxsum = MaxSum(a,med+1,right);

    // 算中间部分的和=从中间向左的最大和（lefts)+从中间向右的最大和（rights)

    Int s1=0,lefts = 0;

    for(int i =med;med>=left;i--){
    Lefts+=a[i];
    If(lefts>s1) s1=lefts;
    }

    Int s2=0,rights=0;
    for(int i =med+1;med<right;i++){
    rights+=a[i];
    If(lrights>s1) s2=rights;
    }
    Int medSum = s1+s2;
    If(medSum <leftMaxsum ) sum = leftMaxsum ;
    If(medSum <rightMaxsum ) sum = rightMaxsum;
    Else{
    Sum = medSum;
    }
    }
}
```

时间复杂度：

时间复杂度递归方程

$$T(n)=\begin{cases} O(1), & n=1 \\ 2T\left(\frac{n}{2}\right)+O(n), & n>1 \end{cases}$$

主定理解方程得 T(n)=O(nlogn)

(3) 蛮力法：

代码：

穷举法就是将n中所有的字段组合找出来，比较字段的和的大小，选出最大的来。

```
int MaxSubsequenceSum(const int array[], int n, int &maxi, int &maxj){
    int tempSum, maxSum = INT_MIN; // 使用INT_MIN初始化maxSum
    for (int i = 0; i < n; i++) {   // 子序列的各个起始位置
        for (int j = i; j < n; j++) {   // 子序列终止的各个位置
            tempSum = 0;
            for (int k = i; k <= j; k++) {   // 求各个子序列的和
                tempSum += array[k];
            }
            if (tempSum > maxSum) {  // 使用 > 来确定是否更新maxSum
                maxSum = tempSum;    // 更新最大的子序列的和
                maxi = i;
                maxj = j;
            }
        }
    }
    return maxSum;
}
```

蛮力法（Brute Force）

- **时间复杂度**：O(n^3)
- **空间复杂度**：O(1)
- **描述**：通过三层嵌套循环，检查所有可能的子数组和，并找出最大值。
- **优点**：实现简单。
- **缺点**：效率极低，对于大数组不实用。

8. （资料题）

【分治法】设 a[0:n-1]是已排好序的数组。请设计一个类二分搜索算法，使得当搜索元素 x 不在数组中时，返回小于 x 的最大元素位置 i 和大于 x 的最小元素位置 j。当搜索元素在数组中时，i 和 j 相同，均为 x 在数组中的位置。可选择算法流程图、伪代码或描述的形式进行算法说明。

第 1 空

为了设计这样一个改进的二分搜索算法，我们需要修改传统的二分搜索以找到小于x的最大元素位置i和大于x的最小元素位置j。这个算法的关键在于，当x不在数组中时，`left`和`right`指针最终会停留在x应该插入的位置的两边。

以下是算法的步骤：

1. **初始化**：设置左指针`left = 0`和右指针`right = n - 1`。
2. **二分搜索**：
   - 当`left <= right`：
     - 计算中间位置`mid = left + (right - left) // 2`。
     - 如果`a[mid] == x`，找到元素x，返回`(mid, mid)`。
     - 如果`a[mid] < x`，则更新`left = mid + 1`。
     - 如果`a[mid] > x`，则更新`right = mid - 1`。
3. **处理未找到的情况**：
   - 如果x没有在数组中找到，此时`left`是第一个大于x的元素位置，而`right`是最后一个小于x的元素位置。
   - 特殊情况处理：
     - 如果`left`超出数组右边界（`left >= n`），则设置`j = -1`（没有大于x的元素）。
     - 如果`right`小于0，则设置`i = -1`（没有小于x的元素）。
4. **返回结果**：返回`(i, j)`，其中`i = right`，`j = left`。

下面是该算法的伪代码表示：

```python
def binary_search_recursive(a, left, right, x):
    if left > right:
        # 处理边界情况
        return right, left

    mid = left + (right - left) // 2
    if a[mid] == x:
        return mid, mid
    elif a[mid] < x:
        return binary_search_recursive(a, mid + 1, right, x)
    else:
        return binary_search_recursive(a, left, mid - 1, x)
```

只写上面思路就可以了

9. （资料题）

【动态规划】对于序列 X={C，D，B，A}、Y={A，B，C，B，A，D，B}，请分析两组序列是否具有公共子序列？并根据动态规划的思想，填写问题求解的二维最优子结构数组。（50分）

第 1 空

10. （资料题）

【动态规划】问题描述：桌上有 n 张牌，编号为 1 到 n，每张牌上有一个数字，第 i 张牌的数字为，现在小方和小明两个人玩游戏，轮流抽牌，每人一次只能抽一张牌，小明先抽。每次抽牌只能抽取最上面的牌或者最下面的牌，他们两个都是随机抽取，小明每次抽取上面的牌的概率为 p，抽取下面的牌的概率为 1-p。小方每次抽取上面的牌的概率为 q，抽取下面的概率为 1-q。最后他们的得分为所有牌上的数字之和，求小明的分数期望。

输入描述：第一行为三个数

，第二行为 n 个数，为每张牌上的分数。请利用回溯法进行分析分析与评价，求解小明的分数期望。

第 1 空

11. （资料题） **【贪心算法】设某通信电文有 a,b,c,d,e 五个字符组成，他们在电文中出现的次数分别是 14，7，10，3，21。给出他们的 huffuman 编码，画出对应的 huffuman 树，并解释其所用的算法类型。**

第 1 空

要创建 Huffman 编码并画出对应的 Huffman 树，我们首先需要了解 Huffman 编码的基本原理。Huffman 编码是一种用于无损数据压缩的算法。它通过为使用频率较高的字符分配更短的编码，而为使用频率较低的字符分配更长的编码，从而减少整体的平均编码长度。

字符及其频率：

- a: 14
- b: 7
- c: 10
- d: 3
- e: 21

以下是构建 Huffman 树并生成 Huffman 编码的步骤：

1. **创建叶子节点**: 对于每个字符和其频率，创建一个叶子节点。
2. **构建树**: 选择两个频率最低的节点作为子节点，创建一个新的内部节点，其频率是这两个节点的频率之和。重复这个步骤，直到只剩下一个节点。这个节点是树的根节点。
3. **分配编码**: 从根节点开始，向左走分配编码"0"，向右走分配编码"1"。每个叶子节点的路径决定了其字符的 Huffman 编码。

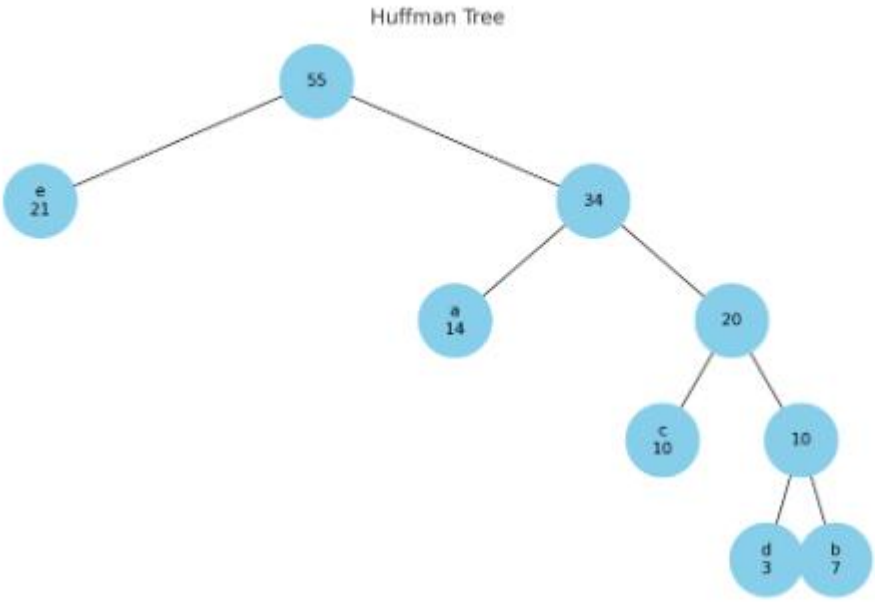
让我们按照这些步骤来生成 Huffman 编码和树。

根据 Huffman 算法，我们得到了以下编码：

- e: 0
- a: 10
- c: 110
- d: 1110
- b: 1111

接下来，我将画出对应的 Huffman 树。在这棵树中，每个叶子节点代表一个字符，而内部节点表示这些字符的组合。树的每个左分支代表"0"，每个右分支代表"1"。这样，从根节点到每个叶子节点的路径就形成了该叶子节点字符的 Huffman 编码。 (>_<)

Huffman Tree



这是构建的 Huffman 树。在这棵树中：

- 每个叶子节点代表一个字符和它在文本中的频率。
- 内部节点表示这些字符的组合的频率。
- 从根节点到每个叶子节点的路径形成了该叶子节点字符的 Huffman 编码。

在这个例子中，字符 'e' 有最高的频率（21），所以它得到了最短的编码 '0'。相反，字符 'd' 和 'b' 有最低的频率，所以它们得到了最长的编码 '1110' 和 '1111'。这种编码方法有效地减少了整体编码的平均长度，从而实现了数据的压缩。 (>-)

Huffman 编码是一种贪心算法的应用。贪心算法是一种在每个步骤中都选择当前看起来最优的选择的算法，即它做出局部最优解的选择，希望这样的选择能导致全局最优解。在 Huffman 编码中，这种方法体现在构建编码树的过程中：

1. **频率最低的节点合并**：在每一步中，选择两个频率最低的节点合并。这是一个贪心的选择，因为它试图通过合并最不常用的字符来最小化编码的总长度。
2. **递归构建树**：这个过程递归地继续，每次都选择当前最低频率的两个节点（无论它们是原始的叶子节点还是已经合并过的内部节点），直到所有节点都合并成一个树。
3. **编码长度与频率成反比**：在最终的 Huffman 树中，最常用的字符（最高频率）具有最短的编码路径，而最不常用的字符（最低频率）具有最长的编码路径。这是贪心策略的结果，因为它在每一步都试图为当前最常见的字符集合分配尽可能短的新编码。

总之，Huffman 编码算法使用贪心策略来逐步构建一个最优的二进制编码树，其中每个字符的编码长度与其在输入数据中的频率成反比。这种方法确保了总体编码长度最小化，从而实现有效的数据压缩。

12. （资料题） **【贪心算法】思考 Dijkstra 算法求解单元最短路径的主要思路？试分析该问题为何具有最优子结构性质？**

Dijkstra算法是一种解决单源最短路径问题的经典算法。它用于在加权图中找到从一个特定的起始顶点到图中所有其他顶点的最短路径。这种算法是典型的贪心算法的应用。

Dijkstra算法的主要思路

1. **初始化**：将所有顶点的最短路径估计值设为无穷大，除了源点，其路径值设为0。
2. **选择最近的顶点**：在未处理的顶点中选择一个距离最小的顶点。初始时，这将是源点。
3. **更新邻居的距离**：对于选定顶点的所有未处理的邻居，检查是否可以通过选定的顶点改善到邻居的最短路径。如果是，则更新邻居的最短路径值。
4. **标记为已处理**：完成对选定顶点的处理，将其标记为已处理，以便不再重新考虑它。
5. **重复**：重复上述过程，直到所有顶点都被处理过。

为何具有最优子结构性质

一个问题具有最优子结构性质，意味着问题的最优解包含其子问题的最优解。Dijkstra算法之所以有效，是因为单源最短路径问题具有最优子结构。这可以从以下几点来理解：

1. **局部最优选择**：在每个步骤中，通过选择当前距离最短的顶点，算法保证了这一选择在后续步骤中不会导致更高的总成本。换句话说，从源点到该顶点的最短路径在后续的任何路径选择中不会改变。
2. **独立的子问题**：选择一个顶点的最短路径时，不受其他顶点的选择影响。每个顶点的最短路径计算可以看作是独立的子问题，其解决方案不会影响到解决其他顶点的最短路径问题。
3. **无后效性**：一旦顶点的最短路径被确定，就不会因为后续的选择而改变。这意味着每一步的选择只依赖于当前的状态，而不依赖于解决问题的路径或未来的决策。

因此，通过局部最优选择（即选择当前看起来最近的顶点），Dijkstra算法能够构建全局最优解（即从源点到所有其他顶点的最短路径）。这种最优子结构是贪心算法成功应用于单源最短路径问题的关键。

第 1 空

13. （资料题） **【贪心算法】设 C={0，1，…，n-1}是 n 个字符的集合。证明关于 C 的任何最优前缀码可以表示为长度为 2n-1+nlogn 位的编码序列。**

要证明关于字符集 $C = \{0, 1, \ldots, n-1\}$ 的任何最优前缀码可以表示为长度为 $2n - 1 + n \log n$ 位的编码序列，我们首先需要理解最优前缀码的性质和表示方法。

一个最优前缀码是一种特殊类型的二进制编码，其中没有任何编码是另一个编码的前缀。这样的编码可以通过一棵二叉树来表示，其中每个叶子节点代表集合 C 中的一个字符。在一棵有 n 个叶子节点的二叉树中，总共会有 $n-1$ 个内部节点（因为在构建二叉树时，每次合并两个节点会减少一个节点，直到剩下一个根节点）。

我们可以将这棵树的结构编码为一个二进制序列。对于每个节点，我们可以用 '0' 表示内部节点，用 '1' 表示叶子节点。遍历树（例如，通过前序遍历），我们可以生成一个长度为 $2n-1$ 的序列，因为每个叶子节点和内部节点都会被访问一次。

接下来，对于每个叶子节点，我们需要表示它所代表的字符。因为有 n 个字符，我们需要 $\lceil \log n \rceil$ 位来唯一地表示每个字符（这里的对数是以2为底的）。所以，对于 n 个叶子节点，我们需要 $n \cdot \lceil \log n \rceil$ 位来表示所有字符。

因此，整个编码序列的长度是树结构的长度加上字符表示的长度，即：
$2n - 1 + n \cdot \lceil \log n \rceil$

由于 $\lceil \log n \rceil$ 可以近似为 $\log n$（尤其是对于较大的 n），我们可以说编码序列的长度约为 $2n - 1 + n \log n$ 位。这就完成了证明。

在表达式 $2n - 1 + n \log_2(n)$ 中，$2n - 1$ 部分表示的是用于描述哈夫曼树（或任何最优前缀码树）结构的位数。

1. **哈夫曼树结构**：
   - 在哈夫曼树中，每个字符都表示为一个叶子节点。如果有 n 个字符，那么就有 n 个叶子节点。
   - 对于任何二叉树，如果有 n 个叶子节点，那么就有 $n-1$ 个内部节点（这是二叉树的一个性质）。内部节点是除叶子节点外的所有节点。
2. **表示树结构**：
   - 我们可以用一个简单的二进制方案来表示树的结构：'0' 表示内部节点，'1' 表示叶子节点。
   - 在进行树的前序遍历时（先访问节点，然后是左子树，最后是右子树），每个节点都被记录下来。因此，对于 $n-1$ 个内部节点和 n 个叶子节点，总共需要 $(n-1) + n = 2n - 1$ 位来表示整个树的结构。
3. **总结**：
   - $2n - 1$ 位用于描述树的结构，即哪些是内部节点，哪些是叶子节点。
   - $n \log_2(n)$ 位用于在树的叶子节点中标识实际的字符。

将这两部分结合起来，我们就得到了整个最优前缀编码树的完整二进制表示：它既包括了树的结构信息（$2n - 1$ 位），也包括了用于标识每个字符的信息（$n \log_2(n)$ 位）。这种编码方式保证了整个编码树可以被有效地重建，从而允许正确的编码和解码过程。

有点难啊

14. （资料题） **【回溯法】设某一机器由 n 个部件组成，每种部件都可以从 m 个不同的供应商处购得。设 Wij 是从供应商 j 处购得的部件 i 的重量，Cij 是相应的价格。试设计一个算法，给出总价格不超过 C 的最小重量机器设计。**

第 1 空

15. （资料题） **【回溯法】设有 n 个立方体，每个立方体的每面都用红、黄、蓝、绿等 n 种颜色之一染色，要把这 n 个立方体叠成一个方型柱体，使得柱体的每个侧面均有 n 个不同的颜色。试设计一个回溯算法，计算出 n 个立方体的一种满足要求的叠置方案。**

第 1 空

16. （资料题） **【分支限界】以 0/1 背包为例，对比和比较回溯法和分支限界法求解问题的异同？分析节点扩展过程**

第 1 空

17. （资料题） **【分支限界】试设计 0/1 背包问题队列式分支限界法，并使算法在运行结束时输出最优解和最优值；同时分析算法复杂度。**

第 1 空

18. （资料题）

【综合】迷宫问题

迷宫实验是取自心理学的一个古典实验。在该实验中，把一个老鼠从一个无顶大盒子的门放入，在盒中设置了许多墙，对行进方向进行多处阻挡。盒子仅有一个出口，在出口处放置一块奶酪，吸引老鼠在迷宫中寻找道路以达到出口。对同一只老鼠重复进行上述实验，一直到老鼠从入口到出口，而不走错一步。老鼠经多次试验终于得到它学习走迷宫的路线。迷宫由 m 行 n 列的二维数组设置，0 表示无障碍，1 表示有障碍。设入口为（1，1），出口为（m，n），每次只能从一个无障碍单元移到周围四个方向上任一无障碍单元。编程实现对任意设定的迷宫，求出一条从入口到出口的道路。

第 1 空

19. （资料题） **【综合】针对 0/1 背包问题，分别给出蛮力法、动态规划算法、回溯法、分支限界法进行问题求解时的核心代码。**

针对 0/1 背包问题，我可以提供四种方法的核心代码：蛮力法、动态规划算法、回溯法和分支限界法。这些方法在求解同一问题时有着不同的效率和复杂性。

0/1 背包问题简介

0/1 背包问题是一种经典的组合优化问题。给定一组物品，每个物品有一个重量和一个价值，确定应该将哪些物品放入背包中，以使得背包中的总价值最大，同时不超过背包的容量限制。

1. 蛮力法

蛮力法遍历所有可能的物品组合，选出满足重量限制且价值最大的组合。

pythonCopy code

```
def brute_force_knapsack(values, weights, W):
    n = len(values)
    max_value = 0

    for i in range(2**n):
        combo = [int(j) for j in format(i, f'0{n}b')]
        weight = sum(combo[j] * weights[j] for j in range(n))
        value = sum(combo[j] * values[j] for j in range(n))

        if weight <= W and value > max_value:
            max_value = value

    return max_value
```

2. 动态规划算法

动态规划算法通过构建一个表格来逐步求解子问题，最终得到整个问题的解。

pythonCopy code

```
def dynamic_programming_knapsack(values, weights, W):
    n = len(values)
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, W + 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][W]
```

3. 回溯法

回溯法通过逐个考虑物品是否放入背包，回溯搜索所有可能的解。

pythonCopy code

```
def backtrack_knapsack(values, weights, W):
    n = len(values)
    max_value = [0]

    def backtrack(i, current_weight, current_value):
        if i == n:
            if current_value > max_value[0]:
                max_value[0] = current_value
            return
```

```
        # 不放当前物品
        backtrack(i + 1, current_weight, current_value)

        # 放当前物品（如果不超重）
        if current_weight + weights[i] <= W:
            backtrack(i + 1, current_weight + weights[i], current_value + values[i])

    backtrack(0, 0, 0)
    return max_value[0]
```

4. 分支限界法

分支限界法类似于回溯法，但使用贪心策略来剪枝。

pythonCopy code

```
# 这是一个简化的分支限界法示例 def branch_and_bound_knapsack(values, weights, W):
    # 需要实现节点类、优先队列等结构，以及相应的贪心策略和剪枝逻辑。
    # 由于代码较长，这里不展开具体实现。
    pass
```

每种方法都有其适用场景和限制。蛮力法适用于小规模问题，动态规划适用于中等规模问题，回溯和分支限界法在某些情况下可以处理较大规模的问题，但也可能受限于时间和空间复杂度。