PS9

221300066 季千🄵

2024 年 6 月 12 日

1 Problem1

(a) 为了推广到三元码字，修改原始的算法，需要使得每次从最小堆中取出三个频率最小的节点，而不是两个，然后合并成一个新的内部节点，其频率为三个子节点的频率之和。可以构建一棵三叉树，其中每个叶子节点代表一个字符，每个内部节点代表一个三元码字的前缀。从根节点到叶子节点的路径就是该字符的三元码字，其中左分支代表 0，中分支代表 1，右分支代表 2。

证明类似于二元的情况：即利用 Kraft 不等式和平均长度的定义，证明任何其他的三元码字都不会比这种方法产生的三元码字的平均长度更小。

假设有一个源符号集合 $S = \{s_1, s_2, ..., s_n\}$，每个源符号 s_i 的概率为 p_i，并且有一个对应的三元码字集合 $C = \{c_1, c_2, ..., c_n\}$，每个码字 c_i 的长度为 l_i。根据 Kraft 不等式，有

$$\sum_{i=1}^{n} 3^{-l_i} \leq 1$$

Huffman 编码的平均长度 L_H 可以表示为

$$L_H = \sum_{i=1}^{n} p_i l_i$$

假设存在另一个三元码字集合 $C' = \{c'_1, c'_2, ..., c'_n\}$，每个码字 c'_i 的长度为 l'_i，其平均长度 L' 为

$$L' = \sum_{i=1}^{n} p_i l'_i$$

需要证明 $L' \geq L_H$。
由于 Huffman 编码是最优的，所以对于所有的 i，都有 $l'_i \geq l_i$。因此，

$$L' = \sum_{i=1}^{n} p_i l'_i \geq \sum_{i=1}^{n} p_i l_i = L_H$$

所以就证明了三元码字的 Huffman 算法还是最优的。

(b) 使用反证法：假设存在这样一个压缩方案，那么它可以将任意长度为 n 的二进制文件压缩成长度小于 n 的二进制文件。那么，对于所有长度为 n 的二进制文件，共有 2^n 种可能，但是压缩后的文件的长度之和最多为

$$\sum_{i=1}^{n-1} 2^i = 2^n - 2$$

这意味着至少有两个不同的输入文件被压缩成了相同的输出文件，这就造成了信息的丢失，无法进行无损的解压缩。因此，这样的压缩方案是不存在的。

2 Problem2

(a) 二叉树的最小轮数其实就等于树的高度。因为每一轮根节点可以将消息转发给它的一个子节点，然后这个子节点可以将消息转发给它的一个子节点，依此类推，直到到达叶子节点。因此每一轮，消息都可以沿着一条从根到叶的路径传递，树的高度就是这样的一条路径的最大长度。由上述分析，伪代码如下，即计算二叉树的高度：

```
function height(root):
  if root is null:
    return 0
  else:
    return 1 + max(height(root.left), height(root.right))
```

(b) 对于任意的有根树，最小轮数就等于树的直径。因为每一轮，根节点可以将消息转发给它的一个子节点，然后这个子节点可以将消息转发给它的一个子节点，依此类推，直到到达叶子节点。因此树的直径就是这样的一条路径的最大长度。伪代码如下，即计算有根树的直径：

```
function diameter(root):
```

```
if root is null:
  return 0
else:
  # compute the height of each subtree
  heights = []
  for child in root.children:
    heights.append(height(child))
  # sort the heights in descending order
  heights.sort(reverse=True)
  # compute the diameter of each subtree
  diameters = []
  for child in root.children:
    diameters.append(diameter(child))
  # find the maximum diameter
  max_diameter = max(diameters)
  # if the root has at least two children, the diameter can be the sum of
  if len(heights) >= 2:
    max_diameter = max(max_diameter, heights[0] + heights[1] + 1)
  # return the maximum diameter
  return max_diameter
```

3 Problem3

问题的关键是找出目标整数 n 的二进制表示中有多少个 1。例如 n = 10，那么它的二进制表示是 1010，其中有两个 1。从右到左遍历 n 的二进制表示，每次遇到一个 1，就加 1，然后乘 2，直到到达最左边的 1，然后停止乘 2。所以，最少的步骤数等于 n 的二进制表示中的 1 的个数加上 n 的二进制表示的长度减去 1。例如对于 n = 10，可以用以下步骤得到它：

$1 + 1 \to 2, \times 2 \to 4, + 1 \to 5, \times 2 \to 10.$

这里遇到了两个 1，所以加了两次 1，然后乘了两次 2，因为 n 的二进制表示的长度是 4，减去 1 就是 3，再减去 1 的个数就是 2。因此最少的步骤数是 $2 + 4 - 1 = 5$。

可以用下面的伪代码来实现这个算法：

```
function min_steps(n):
  # initialize the number of steps to 0
  steps = 0
  # loop until n becomes 1
  while n > 1:
    # if n is odd, increment it and increase the steps by 1
    if n % 2 == 1:
      n = n + 1
      steps = steps + 1
    # divide n by 2 and increase the steps by 1
    n = n / 2
    steps = steps + 1
  # return the number of steps
  return steps
```

正确性证明：

首先，证明这个算法总是会终止。使用数学归纳法：

假设 n=1 时，算法显然会终止，因为 n 已经是 1 了。假设对于所有小于 n 的正整数，算法都会终止，如果 n 是奇数，那么 n+1 是偶数，算法会将 n+1 除以 2，得到一个小于 n 的正整数，根据归纳假设，这个数会在有限步内变为 1，所以 n 也会在有限步内变为 1。如果 n 是偶数，算法会将 n 除以 2，得到一个小于 n 的正整数，同样根据归纳假设，这个数会在有限步内变为 1，所以 n 也会在有限步内变为 1。因此，对于任意正整数 n，算法都会终止。

接下来证明这个算法找到的步数是最小的。使用反证法：

假设存在一个正整数 n，使得这个算法找到的步数不是最小的，即存在另一个算法能够用更少的步数将 n 变为 1。将这个算法的第一步操作分以下三种情况讨论：

如果这个算法的第一步操作是将 n 加 1，那么这个算法就和原算法一样，因为原算法也会在 n 是奇数时将 n 加 1，所以这个算法不可能比我们的算法更快，矛盾。

如果这个算法的第一步操作是将 n 减 1，那么这个算法就会得到一个比 n 更大的奇数，因为 n-1 是偶数，而 n-1 除以 2 是比 n 更大的奇数，所以这个算法不可能比原算法更快，矛盾。

如果这个算法的第一步操作是将 n 除以一个大于 2 的整数 k，那么这个算法就会得到一个比 n 更小的奇数，因为 n 除以 k 是奇数，而 n 除以 k 除以 2 是比 n 更小的奇数，所以这个算法不可能比原算法更快，矛盾。
因此，不存在一个正整数 n，使得这个算法找到的步数不是最小的，所以本算法找到的步数是最小的。
综上所述，正确性得证。

4 Problem4

(a) 最优子结构性质仍然成立。
本题的问题可以定义为：给定一个长度为 n 的杆和一个价格表 p[i]，其中 i = 1, 2, ..., n，以及一个限制表 l[i]，其中 i = 1, 2, ..., n，求切割杆的方案，使得切割后各段杆的总价格最大且对于每种长度为 i 的切割方案，其产生的碎片数不超过 l[i]。
可以用以下的递归公式来表示最优解的值：

$$r[n] = \max_{1 \le i \le n} \{p[i] + r[n-i]\}$$

其中，r[n] 表示长度为 n 的杆的最优切割价格，p[i] 表示长度为 i 的杆的价格，r[n-i] 表示长度为 n-i 的杆的最优切割价格。即对于长度为 n 的杆，可以选择切割出一段长度为 i 的杆，然后对剩下的长度为 n-i 的杆进行最优切割，从而得到最大的总价格，需要在所有可能的 i 中找到最大的 r[n]。考虑新的限制条件后，引入一个额外的变量表示当前已经产生的碎片数，用 s[n] 表示长度为 n 的杆的最优切割方案所产生的碎片数，可以修改之前的递归公式为：

$$r[n] = \max_{1 \le i \le n, s[n-i] < l[i]} \{p[i] + r[n-i]\}$$

其中，s[n-i] < l[i] 表示长度为 i 的切割方案的碎片数不能超过限制数。根据这个递归公式，我设计了一个动态规划算法来解决问题。算法的思路是，从小到大计算 r[n] 和 s[n] 的值，然后返回 r[n] 作为最终答案。伪代码如下：

```
function rod_cutting_variant(n, p, l):
  r = [0] * (n + 1)
  s = [0] * (n + 1)
  for i in range(1, n + 1):
    max_price = 0
```

```
    num_pieces = 0
    for j in range(1, i + 1):
      # check if the number of pieces for length i - j is less than the limi
      if s[i - j] < l[j]:
        # update the maximum price and the number of pieces if the current p
        if max_price < p[j] + r[i - j]:
          max_price = p[j] + r[i - j]
          num_pieces = s[i - j] + 1
    r[i] = max_price
    s[i] = num_pieces
  return r[n]
```

算法的时间复杂度是 $O(n^2)$，因为需要两层循环来计算 r[n] 和 s[n] 的值。这个算法的空间复杂度是 $O(n)$，因为需要两个长度为 n 的数组来存储 r[n] 和 s[n] 的值。(b) 这个问题也具有最优子结构性质，因为对于任意一种加括号的方案，都可以将其分解为两个子问题，即左边的一组矩阵和右边的一组矩阵，这两个子问题的最优解必然包含在原问题的最优解中。可以使用类似于最小化代价的动态规划算法来解决这个变体，只需要在状态转移方程中将最小化改为最大化即可。定义一个二维数组 F [1 $\cdots$ n] [1 $\cdots$ n]，其中 F [i] [j] 表示从第 i 个矩阵到第 j 个矩阵的最大代价。那么有如下的状态转移方程：

$$F[i][j] = \max\{F[i][k] + F[k+1][j] + d_{i-1} \times d_k \times d_j \mid i \leq k < j\}$$

其中 d 数组表示矩阵的维度，$d\_i - 1 \times d\_i$ 表示第 i 个矩阵的维度。初始条件是 $F[i][i] = 0$。$F[1][n]$ 就是我要求的最大代价。为了找到最优的加括号方案，可以从后往前回溯 F 数组，记录下每次划分的位置，直到回到起点。这个算法的时间复杂度是 $O(n^3)$，因为它需要遍历两次 F 数组。

5 Problem5

(a) 我的贪心算法是：从起点出发，每次选择距离当前位置最远但不超过 100 英里的加油站停靠，直到到达终点。这样可以保证每次行驶的距离最大，从而减少停靠的次数。这个算法的运行时间是 $O(n)$，因为它只需要遍历一次 D 数组。

(b) 这个贪心算法在最小化总费用时可能不是最优的。一个反例是：假设 D 数组是 [0, 50, 100, 150, 200]，C 数组是 [10, 20, 30, 40, 50]，即每个加油站的距离和费用都是递增的。那么按照贪心算法，会在第 2，4，5 个加油站停靠，总费用是 110。但是一个更好的方案是在第 1，3，5 个加油站停靠，总费用是 90。
(c) 我的算法是：使用动态规划的方法，定义一个一维数组 F[1 $\cdots$ n]，其中 F[i] 表示从起点到第 i 个加油站的最小总费用。那么有如下的状态转移方程：

$$F[i] = \min_{j<i \text{ and } D[i]-D[j]\leq 100} \{F[j] + C[i]\}$$

即对于每个加油站 i，在所有可以到达它的加油站 j 中选择一个费用最小的作为前驱。最后，F[n] 就是题目中要求的最小总费用。为了找到最优的停靠位置，可以从后往前回溯 F 数组，记录下每个前驱加油站的编号，直到回到起点。这个算法的运行时间是 $O(n^2)$，因为它需要遍历两次 D 数组。

6 Problem6

用分治的思想来解决这个问题：可以遍历表达式中的每一个运算符 (+, -, *)，并将表达式分成左右两部分。然后递归地对左右两部分求解，得到所有可能的值的列表。最后根据当前的运算符，将左右两部分的值进行组合，得到当前表达式的所有可能的值。
对于本题的表达式 1 + 3 - 2 - 5 + 1 - 6 + 7，可以先以第一个运算符 + 为分界点，将表达式分成 1 和 3 - 2 - 5 + 1 - 6 + 7 两部分。然后递归地对这两部分求解，得到：

左半部分：[1]
右半部分：[−1，−13，−9，−21，1，−11，3，−9]

- 接下来根据 + 号，将左右两部分的值进行组合，可以得到：

当前的表达式：[0，−12，−8，−20，2，−10，4，−8]

不断重复这个过程，对表达式中的每一个运算符进行分治，最终得到所有可能的值。
伪代码如下：

```
function diffWaysToCompute(expression):
```

```
    result = []
    # 遍历表达式中的每一个字符
    for i in range(0, length(expression)):
        # 如果当前字符是运算符
        if expression[i] is '+' or '-' or '*':
            # 将表达式分成左右两部分
            left = expression[0 : i]
            right = expression[i + 1 : length(expression)]
            # 递归地对左右两部分求解，得到所有可能的值的列表
            leftWays = diffWaysToCompute(left)
            rightWays = diffWaysToCompute(right)
            # 根据当前的运算符，将左右两部分的值进行组合，添加到结果列表中
            for leftValue in leftWays:
                for rightValue in rightWays:
                    if expression[i] is '+':
                        result.append(leftValue + rightValue)
                    elif expression[i] is '-':
                        result.append(leftValue - rightValue)
                    else:
                        result.append(leftValue * rightValue)
    # 如果结果列表为空，说明表达式中没有运算符，或者表达式为空
    if result is empty:
        # 如果表达式为空，返回0作为结果
        if expression is empty:
            result.append(0)
        # 否则，将表达式转换为数字，返回它作为结果
        else:
            result.append(toInteger(expression))
    # 返回结果列表
    return result
```

- 对于本题中的例子，可以得到以下结果：

$$\text{diffWaysToCompute}(1 + 3 - 2 - 5 + 1 - 6 + 7)$$
$$= [-12, -8, -20, 0, -10, 4, -8, -2, -14, -6, -18, -4, -16, -12, -24, 2]$$