## 算法复习

[算法复习](#算法复习)  
 [关于贪心算法的规则最优解问题](#关于贪心算法的规则最优解问题)  
 [关于贪心算法的正确性检验问题](#关于贪心算法的正确性检验问题)  
 [例题4.3](#例题43)  
 [例题4.4](#例题44)  
 [例题4.7](#例题47)  
 [例题4.8](#例题48)  
 [例题4.14](#例题414)  
 [例题4.19](#例题419)  
 [例题4.20](#例题420)

#### 关于贪心算法的规则最优解问题

贪心算法的每一步规则并不总是固定的，它取决于问题的具体情况。

1. 规则可能固定：

某些问题中，贪心策略的每一步规则都是固定的，例如：  
活动选择问题： 每次选择结束时间最早的活动，这个规则是固定的。  
最小生成树问题： 每次选择权值最小的边，这个规则也是固定的。

1. 规则可能不固定：

很多情况下，贪心策略的每一步规则需要根据当前状态进行调整，例如：  
背包问题： 贪心策略可能是每次选择单位价值最高的物品，但如果背包容量有限，可能需要调整策略，选择更小的物品来最大化总价值。  
最短路径问题： 贪心策略可能是每次选择距离当前节点最近的节点，但如果存在更远的节点，但最终能到达目标节点的路径更短，则需要调整策略。  
总结：

贪心算法的每一步规则并不总是固定的，它取决于问题的具体情况。  
对于一些问题，贪心策略的每一步规则是固定的，例如活动选择问题和最小生成树问题。  
对于其他问题，贪心策略的每一步规则可能需要根据当前状态进行调整，例如背包问题和最短路径问题。  
关键在于： 贪心算法的每一步规则都是为了在当前状态下获得最优解，但并不保证最终得到全局最优解。

#### 关于贪心算法的正确性检验问题

以活动选择问题为例：

问题描述： 给定一组活动，每个活动都有开始时间和结束时间，要求选择一个最大的活动集合，使得这些活动之间互不冲突。  
贪心规则： 每次选择结束时间最早的活动。  
反证法步骤：

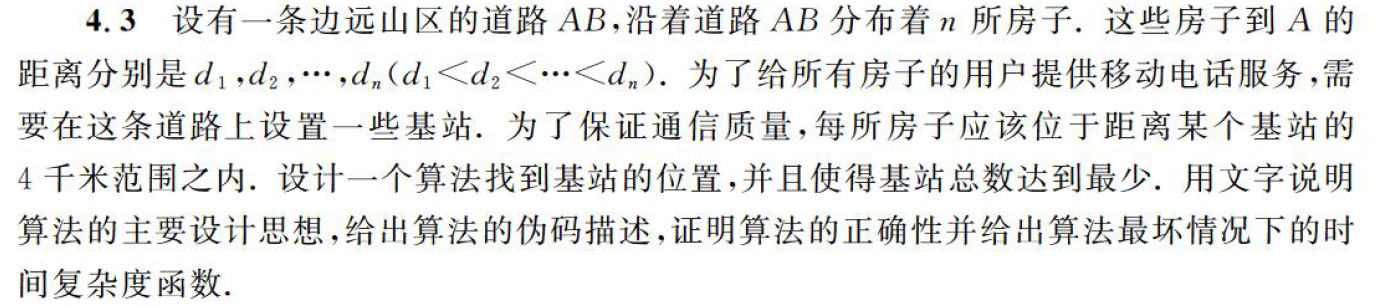
假设： 假设贪心算法得到的解不是全局最优解，即存在一个更大的活动集合，它包含了贪心算法没有选择的活动。  
分析： 设贪心算法选择的活动集合为 S，而更优的活动集合为 T，且 T 包含了 S 中没有的活动 a。  
矛盾： 由于 a 不在 S 中，说明 a 的结束时间比 S 中最后一个活动的结束时间晚。而贪心算法每次都选择结束时间最早的活动，所以 S 中最后一个活动的结束时间一定早于 a 的结束时间。  
结论： 由于 a 的结束时间比 S 中最后一个活动的结束时间晚，所以 a 的开始时间也一定比 S 中最后一个活动的结束时间晚。因此，a 与 S 中的活动不冲突，可以加入到 S 中，这与假设 T 包含了 S 中没有的活动 a 矛盾。  
因此，贪心算法得到的解是全局最优解。

总结：

通过反证法，我们证明了贪心算法得到的解是全局最优解。这个例子说明了，对于某些问题，贪心算法能够得到全局最优解，而反证法可以用来证明这一点。

需要注意的是： 并非所有问题都适用反证法来判断贪心解是否为最优解。对于一些问题，贪心算法可能无法得到全局最优解，而反证法也无法证明这一点。

#### 例题4.3



贪心算法  
设计思想：

贪心算法的思想是每次都选择最优的方案，即尽可能地将基站放置在离当前房子最远的位置，同时满足所有房子都在4公里范围内。具体来说，算法从第一个房子开始，将第一个基站放置在距离第一个房子4公里处。然后，继续往后遍历房子，如果当前房子距离上一个基站超过4公里，则在当前房子距离上一个基站4公里处放置一个新的基站。

def greedy\_algorithm(distances):

输入 distances ,一个列表，包含所有房子到A点的距离。

输出：stations ,一个列表，包含所有基站的位置。

stations = []
  
 last\_station = 0
  
 for i in range(len(distances)):
  
 if distances[i] - last\_station > 4:
  
 stations.append(distances[i] - 4)
  
 last\_station = distances[i] - 4
  
 return stations

正确性证明：

贪心算法的正确性可以通过反证法来证明。假设存在一种比贪心算法更优的方案，即使用更少的基站就能覆盖所有房子。那么，在该方案中，必定存在一个基站，它覆盖了两个或多个房子，并且这两个房子之间距离大于4公里。而贪心算法在遇到这种情况时，会选择在距离第一个房子4公里处放置一个新的基站，而不是将基站放置在两个房子之间。因此，贪心算法的方案不会比其他方案更差，即贪心算法是正确的。

时间复杂度：

贪心算法的时间复杂度为O(n)，其中n为房子的数量。因为算法需要遍历所有房子一次，并在遍历过程中进行常数时间的操作。

最坏情况：

贪心算法的最坏情况是所有房子都均匀分布在道路上，并且相邻两栋房子之间的距离都大于4公里。在这种情况下，贪心算法需要放置n个基站，才能覆盖所有房子。

动态规划算法  
设计思想：

动态规划算法的思想是将问题分解成子问题，并利用子问题的解来求解原问题。具体来说，算法定义一个二维数组dp，其中dp[i] [j]表示前i个房子使用j个基站时的最小覆盖范围。算法从第一个房子开始，依次计算每个房子的最小覆盖范围，并利用前一个房子的最小覆盖范围来更新当前房子的最小覆盖范围。

def dynamic\_programming(distances):

输入：distances , 一个列表，包含所有房子到A点的距离。

输出：stations ,一个列表，包含所有基站的位置。

n = len(distances)
  
 dp = [[float('inf')] \* (n + 1) for \_ in range(n + 1)]
  
 dp[0][0] = 0
  
 for i in range(1, n + 1):
  
 for j in range(1, i + 1):
  
 \# 使用j个基站覆盖前i个房子
  
 for k in range(j):
  
 \# 第j个基站覆盖的最后一个房子是第k个房子
  
 if distances[i - 1] - distances[k] <= 4:
  
 dp[i][j] = min(dp[i][j], dp[k][j - 1])
  
 \# 找到使用最少基站覆盖所有房子的方案
  
 min\_stations = n
  
 for j in range(n + 1):
  
 if dp[n][j] != float('inf'):
  
 min\_stations = j
  
 break
  
 \# 回溯找到基站位置
  
 stations = []
  
 i = n
  
 j = min\_stations
  
 while i > 0 and j > 0:
  
 for k in range(j):
  
 if dp[i][j] == dp[k][j - 1] and distances[i - 1] - distances[k] <= 4:
  
 stations.append(distances[k] + 4)
  
 i = k
  
 j -= 1
  
 break
  
 return stations

时间复杂度：动态规划算法的时间复杂度为O(n^3)，其中n为房子的数量。因为算法需要遍历所有房子，并对于每个房子，需要遍历所有可能的基站数量，以及所有可能的覆盖范围。

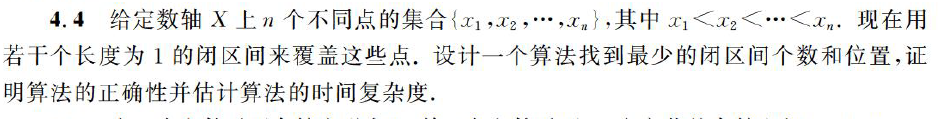
最坏情况：

动态规划算法的最坏情况与贪心算法相同，即所有房子都均匀分布在道路上，并且相邻两栋房子之间的距离都大于4公里。在这种情况下，动态规划算法需要放置n个基站，才能覆盖所有房子。

如果需要快速找到一个近似最优解，则可以使用贪心算法；如果需要找到最优解，则可以使用动态规划算法。

贪心算法是用问题的规则来计算最优解，动态规划算法是用子问题的最优来计算最优解

#### 例题4.4



题目：给定数轴X上n个不同点的集合{x1,x2,...,xn},其中x1<x2<...<xn.现在用 若干个长度为1的闭区间来覆盖这些点，设计一个算法找到最少的闭区间个数和位置,证明算法的正确性并估计算法的时间复杂度.

设计思想：贪心算法的思想是每次都选择最优的方案，即尽可能地将闭区间放置在离当前点集合中最远的位置，同时保证覆盖所有点。具体来说，算法从数轴上第一个点开始，选择一个长度为1的闭区间，尽可能地覆盖尽量多的点。然后，继续往后遍历点集合，如果当前点不在当前选定的最后一个闭区间内（即超出了当前最大的右端点），则选择一个新的区间覆盖当前点，并更新最大的右端点。重复这个过程直到所有点都被覆盖。

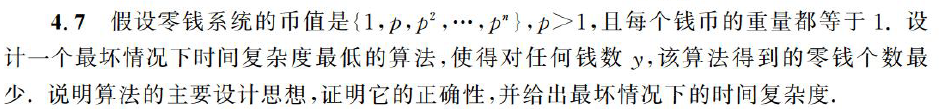
greedy\_min\_intervals(points):
  
 """
  
 贪心算法求解最少闭区间个数和位置。
  
  
 Args:
  
 points: 一个列表，包含所有数轴上的点位置。
  
  
 Returns:
  
 intervals: 一个列表，包含所有闭区间的右端点位置。
  
 """
  
 if not points:
  
 return []
  
  
 points.sort() # 排序所有点的位置
  
 intervals = [] # 存储闭区间的右端点位置
  
 max\_right = points[0] # 初始时最大的右端点为第一个点的位置
  
  
 for point in points:
  
 if point > max\_right: # 如果当前点超出了当前最大的右端点
  
 intervals.append(max\_right) # 添加一个新的闭区间
  
 max\_right = point # 更新最大的右端点
  
  
 intervals.append(max\_right) # 添加最后一个闭区间
  
  
 return intervals

正确性证明：贪心算法的正确性可以通过反证法来证明。假设存在一种比贪心算法更优的方案，即使用更少的闭区间就能覆盖所有点。那么，在该方案中，必定存在一个闭区间，它覆盖了两个或多个点，并且这两个点之间的距离大于1。而贪心算法在遇到这种情况时，会选择在当前能覆盖的最远的点的位置放置闭区间，而不是将闭区间放置在两个点之间。因此，贪心算法的方案不会比其他方案更差，即贪心算法是正确的。

时间复杂度：贪心算法的时间复杂度主要由排序操作决定，为 O(n log n)，其中 n 是点集合的大小。排序完成后，遍历点集合并选择闭区间的操作仅需要线性时间 O(n)。

最坏情况：贪心算法的最坏情况是所有点都均匀分布在数轴上，并且相邻两个点之间的距离都大于1。在这种情况下，贪心算法需要放置 n 个闭区间，才能覆盖所有点。

#### 例题4.7



题目：假设零钱系统的币值是{1,p,p2,...,pn},p>1,且每个钱币的重量都等于1.设 计一个最坏情况下时间复杂度最低的算法,使得对任何钱数y,该算法得到的零钱个数最 少，说明算法的主要设计思想,证明它的正确性,并给出最坏情况下的时间复杂度.

设计思想：本算法旨在使用贪心策略，确保对任意给定的钱数 ( y )，找零时所需的零钱个数最少。零钱系统的币值集合为 ({1,p,p2,...,pn})，其中 ( p > 1 )，每个币值的重量都为1。

算法步骤：

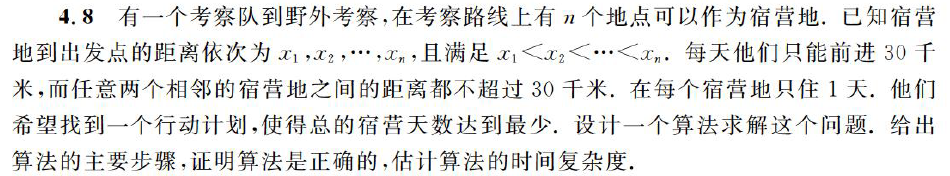
1. 初始化一个空列表，用来存储选定的零钱。
2. 从最大的币值 ( p^k ) 开始，如果 ( pi <= y )，则选择尽可能多的 ( p^i ^) 进入找零列表，直到 ( y ) 减为0。
3. 每次选择的币值数量取决于 ( )，即每次尽量用尽可能多的当前币值。

def min\_coins\_greedy(y, p):
  
 """
  
 使用贪心算法找零，使得零钱个数最少。
  
  
 Args:
  
 y: 需要找零的金额。
  
 p: 零钱系统中的最大币值。
  
  
 Returns:
  
 coins\_used: 一个列表，包含所有使用的零钱币值。
  
 """
  
 coins\_used = []
  
 while y > 0:
  
 max\_coin = 1
  
 while max\_coin <= y:
  
 max\_coin \*= p
  
 max\_coin //= p
  
 coins\_used.append(max\_coin)
  
 y -= max\_coin
  
 return coins\_used

正确性证明：贪心算法的正确性可以通过反证法来证明。假设存在一种更优的找零方案，比贪心算法得到的方案所需的零钱更少。然而，贪心算法每次都选择当前可用的最大币值进行找零，因此其方案不会比其他方案更差，而且在某些情况下能达到最优解。

时间复杂度：贪心算法的时间复杂度主要取决于币值 ( p ) 的数量以及钱数 ( y ) 的大小。在最坏情况下，当 ( y ) 为一个非常大的数时，贪心算法可能需要多次迭代来找零，但每次迭代的操作是常数时间的 ( O(1) )。因此，总的时间复杂度为 ( O(\log\_p y) )，其中 ( \log\_p y ) 表示以 ( p ) 为底的对数。

#### 例题4.8



设计思想：本问题可以使用贪心算法来解决。贪心算法的核心思想是每次选择当前能达到的最远的宿营地作为下一个宿营地，以尽可能减少总的宿营天数。

算法步骤：

1. 初始化一个变量 current\_position 表示当前考察队所在的位置，初始为出发点的位置。
2. 初始化一个变量 camp\_count 表示已经确定的宿营天数，初始为0。
3. 使用一个循环遍历每个宿营地的位置：
   * 对于当前位置 current\_position，找到能覆盖的最远的下一个宿营地。具体步骤是找到最远的满足 x\_i - \text{current\_position} \leq 30 的宿营地。
   * 更新 current\_position 为下一个宿营地的位置。
   * 增加 camp\_count 表示多了一天的宿营。
4. 当所有宿营地都遍历完毕后，camp\_count 即为所求的最少宿营天数。

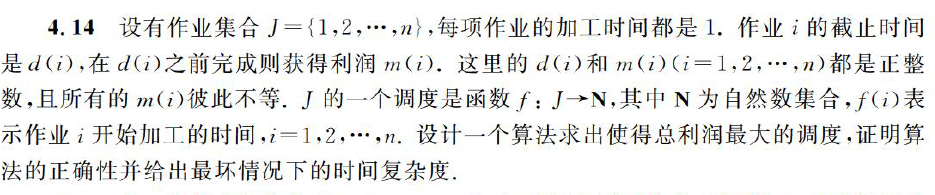
python复制代码def min\_camping\_days(distances):  
 """  
 使用贪心算法确定考察队的最少宿营天数。  
  
 Args:  
 distances: 宿营地到出发点的距离列表，已经排序。  
  
 Returns:  
 min\_days: 最少的宿营天数。  
 """  
 n = len(distances)  
 current\_position = 0 # 当前考察队所在位置（出发点）  
 camp\_count = 0 # 已确定的宿营天数  
  
 i = 0  
 while i < n:  
 # 找到能覆盖的最远的下一个宿营地  
 next\_camp = distances[i]  
 while i < n and distances[i] - current\_position <= 30:  
 next\_camp = distances[i]  
 i += 1  
   
 # 更新当前位置为下一个宿营地的位置  
 current\_position = next\_camp  
 # 增加宿营天数  
 camp\_count += 1  
   
 return camp\_count

正确性证明：贪心算法的正确性可以通过反证法来证明。假设存在一种更优的行动计划，能用更少的宿营天数完成整个考察路线。然而，贪心算法每次都选择当前能到达的最远的宿营地，因此其选择策略不会使总的宿营天数比其他策略更多。因此，贪心算法的方案是最优的。

时间复杂度：贪心算法的时间复杂度主要取决于遍历宿营地的次数，即 O(n)O(n)O(n)，其中 nnn 是宿营地的数量。在每次循环中，我们对当前位置进行一次遍历以找到下一个宿营地，而这些操作都是线性时间的。因此，总的时间复杂度为 O(n)O(n)O(n)。

最坏情况：贪心算法的最坏情况是每个宿营地之间的距离都是30公里，此时需要遍历所有的宿营地，即需要 nnn 天才能完成整个考察路线。

#### 例题4.14



设计思想：本问题可以通过贪心算法来解决。贪心算法的主要思想是优先选择能够获取最大利润的作业进行调度，以尽可能使总利润最大化。

算法步骤：

1. 首先，将所有作业按照截止时间 ( d(i) ) 的先后顺序进行排序，即从小到大排列。
2. 初始化一个数组 schedule，用于记录作业的调度时间。
3. 遍历排序后的作业集合：
   * 对于每个作业 ( i )，选择一个尽可能晚的时间点将其调度，但必须保证在其截止时间 ( d(i) ) 之前完成。
   * 如果选择的时间点已经被占用，尝试选择更早的时间点。
   * 将作业 ( i ) 的调度时间记录在 schedule 中。
4. 最终，计算所有调度作业的总利润。

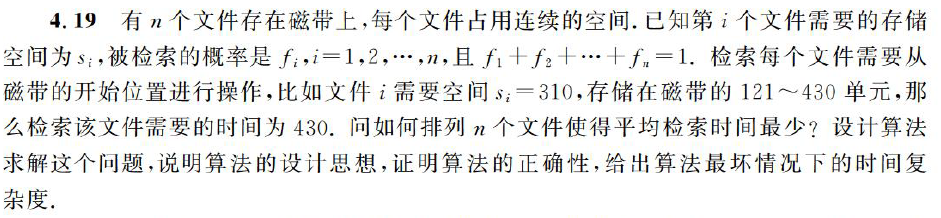
def schedule\_jobs(J, d, m):  
 """  
 使用贪心算法进行作业调度，使得总利润最大化。  
  
 Args:  
 J: 作业集合，表示为列表 [1, 2, ..., n]。  
 d: 每项作业的截止时间，表示为列表 [d(1), d(2), ..., d(n)]，已按截止时间升序排列。  
 m: 每项作业完成时的利润，表示为列表 [m(1), m(2), ..., m(n)]，已按作业索引对应。  
  
 Returns:  
 total\_profit: 最大化的总利润。  
 schedule: 作业的调度时间，表示为列表 [f(1), f(2), ..., f(n)]。  
 """  
 n = len(J)  
 schedule = [0] \* n # 初始化作业调度时间数组  
 total\_profit = 0  
   
 # 按照截止时间 d(i) 升序遍历作业  
 for i in range(n):  
 # 尝试将作业 J[i] 安排在尽可能晚的时间点  
 for j in range(min(d[i], n)-1, -1, -1):  
 if schedule[j] == 0: # 时间点 j 还未被占用  
 schedule[j] = J[i]  
 total\_profit += m[J[i] - 1] # J[i] 在列表中的索引需要减去1  
 break  
   
 return total\_profit, schedule

正确性证明：贪心算法的正确性可以通过反证法来证明。假设存在一种比贪心算法更优的调度方案，即使用更小的总利润完成所有作业。然而，贪心算法每次选择尽可能晚的时间点调度作业，以使得后续仍有更多机会选择其他作业，并且保证每个作业在其截止时间前完成。因此，贪心算法得到的方案不会比其他方案更差，即贪心算法是正确的。

时间复杂度：贪心算法的时间复杂度主要取决于排序作业的时间复杂度和遍历作业的时间复杂度。排序作业的时间复杂度为 ( O(n \log n) )，遍历作业的时间复杂度为 ( O(n^2) )。因此，总的时间复杂度为 ( O(n^2) )，其中 ( n ) 是作业的数量。

最坏情况：贪心算法的最坏情况是所有作业的截止时间 ( d(i) ) 都很大，且作业数量 ( n ) 较大。在这种情况下，算法需要对作业进行完全的排序，并且可能需要在每个时间点上进行搜索找到合适的调度时间。因此，时间复杂度为 ( O(n^2) )。

#### 例题4.19



设计思想：本问题可以通过贪心算法来解决。贪心算法的主要思想是优先选择检索概率较大的文件放置在磁带的开始位置，以减少平均检索时间。

算法步骤：

1. 将所有文件按照其检索概率 ( f\_i ) 从大到小排序。
2. 初始化一个变量 total\_time 记录总的平均检索时间。
3. 遍历排序后的文件集合：
   * 对于每个文件，计算其检索时间，并累加到 total\_time 中。
   * 每次累加时，文件在磁带的起始位置都从零开始计算。
4. 最终，计算平均检索时间 average\_time，即 total\_time / n。

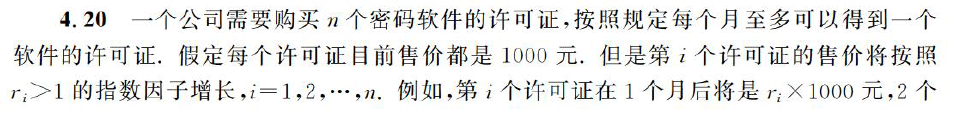
def minimize\_average\_seek\_time(sizes, probabilities):  
 """  
 使用贪心算法排列文件以最小化平均检索时间。  
  
 Args:  
 sizes: 文件占用的存储空间列表，按顺序排列。  
 probabilities: 文件被检索的概率列表，按顺序排列。  
  
 Returns:  
 average\_time: 最小化的平均检索时间。  
 arrangement: 文件的排列顺序列表。  
 """  
 n = len(sizes)  
 total\_time = 0  
 arrangement = []  
  
 # 创建文件列表并按照检索概率降序排序  
 files = [(sizes[i], probabilities[i]) for i in range(n)]  
 files.sort(key=lambda x: x[1], reverse=True)  
  
 # 计算平均检索时间  
 for i in range(n):  
 file\_size = files[i][0]  
 file\_prob = files[i][1]  
 seek\_time = file\_size \* (i + 1) # 文件在磁带的起始位置从0开始  
 total\_time += seek\_time  
 arrangement.append(files[i])  
  
 average\_time = total\_time / n  
 return average\_time, arrangement

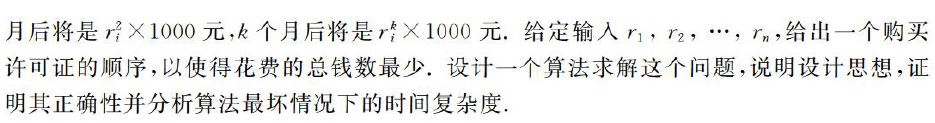
正确性证明：贪心算法的正确性可以通过反证法来证明。假设存在一种更优的排列方案，能够使平均检索时间更少。然而，贪心算法每次选择检索概率较大的文件放置在磁带的开始位置，以尽可能减少每个文件的检索时间。因此，贪心算法得到的方案不会比其他方案更差，即贪心算法是正确的。

时间复杂度：贪心算法的时间复杂度主要取决于对文件排序的时间复杂度，即 ( O(n \log n) )，其中 ( n ) 是文件的数量。在排序完成后，遍历文件列表并计算平均检索时间的过程是线性的，时间复杂度为 ( O(n) )。因此，总的时间复杂度为 ( O(n \log n) )。

最坏情况：贪心算法的最坏情况是在对文件进行排序时，需要 ( O(n \log n) ) 的时间复杂度。此外，遍历文件列表并计算平均检索时间也需要 ( O(n) ) 的时间复杂度。因此，算法的最坏情况下的总时间复杂度仍然是 ( O(n \log n) )。

#### 例题4.20





设计思想：本问题可以通过贪心算法来解决。贪心算法的主要思想是优先选择当前价格最低的许可证购买，以尽可能减少总花费。

算法步骤：

1. 将所有许可证按照其指数因子 ( r\_i ) 从小到大排序。
2. 初始化一个变量 total\_cost 记录总的购买花费。
3. 遍历排序后的许可证集合：
   * 对于每个许可证，计算当前购买时的花费，并累加到 total\_cost 中。
   * 每次累加时，将当前许可证的价格乘以相应的指数因子 ( r\_i^k )，其中 ( k ) 表示该许可证的购买顺序（从0开始）。
4. 最终，得到总的最小花费 total\_cost。

def minimize\_total\_cost(prices, factors):  
 """  
 使用贪心算法购买许可证使得总花费最少。  
  
 Args:  
 prices: 许可证的初始价格列表，按顺序排列。  
 factors: 许可证的指数因子列表，按顺序排列。  
  
 Returns:  
 total\_cost: 最少的总花费。  
 order: 购买许可证的顺序列表，表示许可证的索引按照购买顺序。  
 """  
 n = len(prices)  
 total\_cost = 0  
 order = []  
  
 # 创建许可证列表并按照指数因子升序排序  
 licenses = [(prices[i], factors[i]) for i in range(n)]  
 licenses.sort(key=lambda x: x[1]) # 按照指数因子从小到大排序  
  
 # 计算总花费  
 for k in range(n):  
 price = licenses[k][0] \* (licenses[k][1] \*\* k)  
 total\_cost += price  
 order.append(prices.index(licenses[k][0])) # 记录许可证的索引按照购买顺序  
  
 return total\_cost, order

正确性证明：贪心算法的正确性可以通过反证法来证明。假设存在一种更优的购买顺序，能够使总花费更少。然而，贪心算法每次选择当前价格最低的许可证购买，以尽可能减少当前的购买花费。因此，贪心算法得到的方案不会比其他方案更差，即贪心算法是正确的。

时间复杂度：贪心算法的时间复杂度主要取决于对许可证排序的时间复杂度，即 ( O(n \log n) )，其中 ( n ) 是许可证的数量。在排序完成后，遍历许可证列表并计算总花费的过程是线性的，时间复杂度为 ( O(n) )。因此，总的时间复杂度为 ( O(n \log n) )。

最坏情况：贪心算法的最坏情况是在对许可证进行排序时，需要 ( O(n \log n) ) 的时间复杂度。此外，遍历许可证列表并计算总花费也需要 ( O(n) ) 的时间复杂度。因此，算法的最坏情况下的总时间复杂度仍然是 ( O(n \log n) )。