Final exam 2020 solution

2.

* **Edges in the MST:** DE (1), CF (2), EF (3), BC (5), AD (6)
* **Order of edges added:** DE (1), CF (2), EF (3), BC (5), AD (6)

4.

### 题目解释

题目要求设计一个多项式时间算法，给定图 \( G \) 和两个生成树 \( T \) 和 \( S \)，构造一个最小长度的交换序列将 \( T \) 转换为 \( S \)。

### 解法概述

a) \*\*算法描述\*\*：

- 一般思想是首先找出 \( T \) 中有但 \( S \) 中没有的边，和 \( S \) 中有但 \( T \) 中没有的边，然后在每次迭代中从一个集合中交换一条边到另一个集合。

#### 具体步骤：

1. \*\*确定边集合\*\*：

- 设 \( X\_T \) 为在 \( T \) 中但不在 \( S \) 中的边的集合。

- 设 \( X\_S \) 为在 \( S \) 中但不在 \( T \) 中的边的集合。

2. \*\*迭代边交换\*\*：

- 从 \( X\_T \) 中选取一条边 \( (u, v) \)，并将其移除。这将 \( T \) 分割成两个连通分量。

- 找到 \( X\_S \) 中的一条边 \( (p, q) \)，它连接这两个连通分量（这总是存在的，因为 \( S \) 是一棵树）。这可以通过运行 BFS 或 DFS 来计算每个连通分量中的顶点集合，然后遍历 \( X\_S \) 找到一条连接两个集合的边 \( (p, q) \)。

- 从 \( X\_T \) 中移除 \( (u, v) \) 并从 \( X\_S \) 中移除 \( (p, q) \)。这就是一次交换。

3. \*\*重复上述步骤\*\*：

- 重复该过程，直到集合 \( X\_T \) 和 \( X\_S \) 都为空。

### 为什么这样设计算法

- \*\*目的\*\*：我们希望在每次交换中尽可能有效地将 \( T \) 向 \( S \) 转换。

- \*\*步骤 1\*\*：通过确定 \( X\_T \) 和 \( X\_S \)，我们明确了需要交换的边，这使得后续步骤可以有的放矢。

- \*\*步骤 2\*\*：在每次交换中，通过移除 \( X\_T \) 中的一条边，我们将树分成两个连通分量，然后添加 \( X\_S \) 中的一条边来重新连接这两个分量，确保新的图仍然是生成树。

- \*\*步骤 3\*\*：这种迭代确保了我们逐步将 \( T \) 转换为 \( S \)，且每次交换都是必要的，这样可以保证交换的最小次数。

### 结论

通过这个方法，我们能够系统地、有效地找到从生成树 \( T \) 到生成树 \( S \) 的最小长度交换序列，并且确保每次交换后仍然保持生成树的性质。

### 解释题目和答案

### b) 证明算法的正确性

#### 关键点：

1. \*\*不变量\*\*：

- 我们有不变量 \(|X\_T| = |X\_S|\)，因为 \(T\) 和 \(S\) 都是生成树，并且我们忽略了出现在两棵树中的边。因此，初始时它们的大小相同，在每次交换后，我们从每个集合中移除一条边。

2. \*\*归纳证明\*\*：

- 基础情况： \( R\_1 = T \)，这是一个生成树。

- 归纳步骤：假设 \( R\_i \) 是一个生成树，我们需要证明 \( R\_{i+1} \) 也是一个生成树。我们移除一条边，这样会产生两个连通分量，然后我们添加一条重新连接这两个分量的边，所以 \( R\_{i+1} \) 也是一个生成树。

- 因为 \(S\) 是一棵树，所以总存在一条边来重新连接这两个连通分量。因此， \( X\_S \) 包含了一条连接不同分量的边。

3. \*\*最终状态\*\*：

- 在一系列交换之后，我们得到 \(S\)。这是因为最终 \( X\_S \) 为空，意味着 \(S\) 中的每条边现在都在生成树中。

4. \*\*最小交换序列\*\*：

- 交换序列是最小的：我们执行的交换次数等于不同边的数量。由于每次交换最多只能固定一对边，因此不可能用更少的交换次数将 \( T \) 转换为 \( S \)。

### c) 算法的时间复杂度分析

#### 关键点：

1. \*\*构造集合 \(X\_T\) 和 \(X\_S\)\*\*：

- 这可以在 \( O(n^2) \) 时间内完成，通过检查每棵树中的每条边是否存在于另一棵树中。

- 也可以更高效地通过对边按端点排序并进行并行扫描来完成。

2. \*\*寻找重连接的边\*\*：

- 对于 \( X\_T \) 中的每条边，我们可以在 \( X\_S \) 中找到重连接树的边，这可以通过计算两个集合中的顶点（BFS在 \( O(n) \) 时间内完成）并遍历 \( X\_S \) 的边来实现，检查其端点是否在不同集合中。

- 这样每条边的处理时间为 \( O(n^2) \)，总的交换时间为 \( O(n^3) \)。

### 总结

1. \*\*正确性\*\*：

- 通过归纳证明，我们展示了每一步生成的新树依然是生成树，并且最终得到目标生成树 \( S \)。

2. \*\*时间复杂度\*\*：

- 初始化 \( X\_T \) 和 \( X\_S \) 的时间复杂度为 \( O(n^2) \)。

- 每次交换的时间复杂度为 \( O(n^2) \)，总的时间复杂度为 \( O(n^3) \)。

通过这种方式，我们可以在多项式时间内找到两个生成树之间的最小长度交换序列。

Final 2021

problem 1

b) ### 构造一个反例

为了构造一个反例，我们需要找一个例子使得按照算法得到的集合并不是最小的集合。我们考虑这样一个情况：

#### 集合和子集定义：

设 \( I = \{i\_1, i\_2, i\_3, i\_4, i\_5\} \)，子集 \( S\_1, S\_2, S\_3, S\_4, S\_5 \) 如下：

- \( S\_1 = \{i\_1, i\_2\} \)

- \( S\_2 = \{i\_2, i\_3\} \)

- \( S\_3 = \{i\_3, i\_4\} \)

- \( S\_4 = \{i\_4, i\_5\} \)

- \( S\_5 = \{i\_5, i\_1\} \)

### 按算法执行

1. 统计每个元素的出现次数：

- \( i\_1 \) 出现 2 次（在 \( S\_1, S\_5 \) 中）

- \( i\_2 \) 出现 2 次（在 \( S\_1, S\_2 \) 中）

- \( i\_3 \) 出现 2 次（在 \( S\_2, S\_3 \) 中）

- \( i\_4 \) 出现 2 次（在 \( S\_3, S\_4 \) 中）

- \( i\_5 \) 出现 2 次（在 \( S\_4, S\_5 \) 中）

2. 按照出现次数从大到小排序（这里所有元素出现次数相同，可以任意排序）：

- 假设排序结果为：\( i\_1, i\_2, i\_3, i\_4, i\_5 \)

3. 遍历排序后的元素，构造集合 \( T \)：

- \( i\_1 \)：覆盖 \( S\_1, S\_5 \)，所以 \( T = \{i\_1\} \)。

- \( i\_2 \)：覆盖 \( S\_1, S\_2 \)，由于 \( S\_1 \) 已被 \( i\_1 \) 覆盖，且 \( S\_2 \) 没有被 \( T \) 覆盖，因此 \( T = \{i\_1, i\_2\} \)。

- \( i\_3 \)：覆盖 \( S\_2, S\_3 \)，由于 \( S\_2 \) 已被 \( i\_2 \) 覆盖，且 \( S\_3 \) 没有被 \( T \) 覆盖，因此 \( T = \{i\_1, i\_2, i\_3\} \)。

- \( i\_4 \)：覆盖 \( S\_3, S\_4 \)，由于 \( S\_3 \) 已被 \( i\_3 \) 覆盖，且 \( S\_4 \) 没有被 \( T \) 覆盖，因此 \( T = \{i\_1, i\_2, i\_3, i\_4\} \)。

- \( i\_5 \)：覆盖 \( S\_4, S\_5 \)，由于 \( S\_4 \) 已被 \( i\_4 \) 覆盖，且 \( S\_5 \) 没有被 \( T \) 覆盖，因此 \( T = \{i\_1, i\_2, i\_3, i\_4, i\_5\} \)。

最终结果是 \( T = \{i\_1, i\_2, i\_3, i\_4, i\_5\} \)。

### 最小集合

事实上，最小集合是 \( T = \{i\_1, i\_3, i\_5\} \)，它也覆盖了所有子集：

- \( i\_1 \) 覆盖 \( S\_1, S\_5 \)

- \( i\_3 \) 覆盖 \( S\_2, S\_3 \)

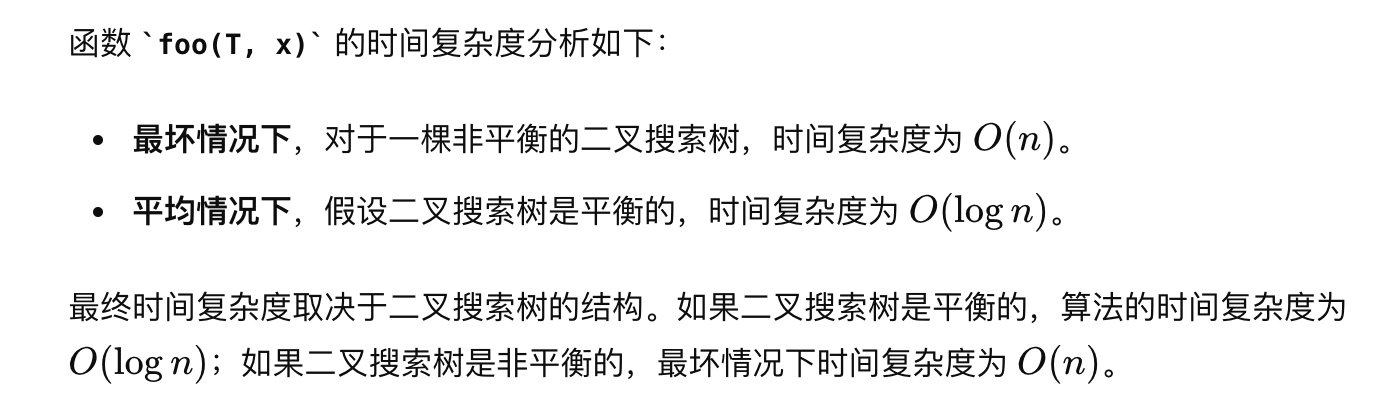
- \( i\_5 \) 覆盖 \( S\_4 \)

### 结论

该反例证明了给定的算法并不能总是计算出最小的集合 \( T \)。算法选择的 \( T = \{i\_1, i\_2, i\_3, i\_4, i\_5\} \) 并不是最小集合，而最小集合应该是 \( T = \{i\_1, i\_3, i\_5\} \)。

Final 2022 answer

1. a)



b)

*W* = {3, 1,2,4}

w = 5

w1 = 3,1. w2 =2 w3 = 4

problem 3:

### 问题分析

这道题要求设计一个数据结构来管理“饥饿游戏”的规则。游戏涉及 \( n \) 个地区，每个地区有 \( m \) 个玩家。每个玩家和地区都有一个分数，游戏规则如下：

1. 两个地区的最佳玩家对战，胜利的玩家和地区的分数增加，失败的玩家和地区的分数减少。

2. 玩家分数为0时被淘汰。

3. 地区没有玩家时被淘汰。

4. 最后一个存活的地区获胜。

### 数据结构描述

为了实现这个数据结构，我们需要支持以下操作：

1. \*\*GETKTHDISTRICT(k)\*\*：返回当前排名第 \( k \) 的地区。

2. \*\*COMPETE(k, j, k\_wins)\*\*：记录第 \( k \) 和第 \( j \) 地区之间比赛的结果，并更新数据结构。

3. \*\*GETSTRONGEST()\*\*：返回分数最高的玩家所在的地区。

我们使用以下数据结构来满足这些需求：

1. \*\*二叉堆\*\*（Binary Heap）：用于快速获取当前排名第 \( k \) 的地区。

2. \*\*红黑树\*\*（Red-Black Tree）：用于快速插入和删除地区信息。

3. \*\*平衡二叉搜索树\*\*（Balanced Binary Search Tree）：用于快速查询和更新玩家的分数。

### 实现步骤

#### a) 数据结构实现

1. \*\*初始化\*\*

- 创建一个包含 \( n \) 个地区的二叉堆，每个地区初始得分为0。

- 每个地区包含一个红黑树，存储其所有玩家，初始分数为1。

2. \*\*GETKTHDISTRICT(k)\*\*

- 使用二叉堆实现，时间复杂度为 \( O(\log n) \)。

3. \*\*COMPETE(k, j, k\_wins)\*\*

- 找到第 \( k \) 和第 \( j \) 地区的最佳玩家（分数最高的玩家）。

- 更新赢家和输家的分数，如果分数为0则移除玩家，如果地区无玩家则移除地区。

- 使用红黑树和二叉堆实现，时间复杂度为 \( O(\log n + \log m) \)。

4. \*\*GETSTRONGEST()\*\*

- 遍历所有地区，找到分数最高的玩家所在的地区。

- 时间复杂度为 \( O(n) \)。

#### 具体实现

```python

class District:

def \_\_init\_\_(self, district\_id):

self.district\_id = district\_id

self.score = 0

self.players = SortedList([(1, i) for i in range(m)]) # 使用SortedList实现红黑树

class HungerGames:

def \_\_init\_\_(self, n, m):

self.districts = [District(i) for i in range(n)]

self.heap = []

for district in self.districts:

heapq.heappush(self.heap, (-district.score, district.district\_id, district))

def getKthDistrict(self, k):

if k > len(self.heap):

return None

return heapq.nsmallest(k, self.heap)[-1][2]

def compete(self, k, j, k\_wins):

if k < 1 or j < 1 or k > len(self.districts) or j > len(self.districts) or k == j:

return

district\_k = self.getKthDistrict(k)

district\_j = self.getKthDistrict(j)

if district\_k is None or district\_j is None:

return

player\_k = district\_k.players[-1]

player\_j = district\_j.players[-1]

if k\_wins:

district\_k.players.remove(player\_k)

district\_k.players.add((player\_k[0] + 1, player\_k[1]))

district\_j.players.remove(player\_j)

district\_j.players.add((player\_j[0] - 1, player\_j[1]))

district\_k.score += 1

district\_j.score -= 1

else:

district\_k.players.remove(player\_k)

district\_k.players.add((player\_k[0] - 1, player\_k[1]))

district\_j.players.remove(player\_j)

district\_j.players.add((player\_j[0] + 1, player\_j[1]))

district\_k.score -= 1

district\_j.score += 1

self.update\_heap(district\_k)

self.update\_heap(district\_j)

def update\_heap(self, district):

for i in range(len(self.heap)):

if self.heap[i][1] == district.district\_id:

self.heap[i] = (-district.score, district.district\_id, district)

heapq.heapify(self.heap)

break

def getStrongest(self):

strongest\_player = None

strongest\_district = None

for district in self.districts:

if len(district.players) > 0 and (strongest\_player is None or district.players[-1][0] > strongest\_player[0]):

strongest\_player = district.players[-1]

strongest\_district = district

return strongest\_district

```

### b) 数据结构的正确性证明

- \*\*GETKTHDISTRICT(k)\*\*：使用二叉堆，能够在 \( O(\log n) \) 时间内找到当前第 \( k \) 的地区。

- \*\*COMPETE(k, j, k\_wins)\*\*：更新玩家和地区分数，并使用红黑树和二叉堆维护数据结构的平衡，确保时间复杂度为 \( O(\log n + \log m) \)。

- \*\*GETSTRONGEST()\*\*：遍历所有地区找到分数最高的玩家，确保结果正确。

### c) 时间和空间复杂度分析

- \*\*空间复杂度\*\*：使用 \( O(nm) \) 的空间来存储所有地区和玩家的信息。

- \*\*时间复杂度\*\*：

- \*\*GETKTHDISTRICT(k)\*\*：\( O(\log n) \)

- \*\*COMPETE(k, j, k\_wins)\*\*：\( O(\log n + \log m) \)

- \*\*GETSTRONGEST()\*\*：\( O(n) \)

综上所述，我们设计了一个有效的数据结构来管理饥饿游戏的规则，并分析了其正确性和复杂度。

problem 4:

### 问题分析

题目要求找出图中两点之间的最短彩虹路径（即路径上任意两个连续的顶点不具有相同的颜色）。需要设计一个时间复杂度为 \(O(m + n \log n)\) 的算法， \(k\) 是颜色的数量，不能在复杂度中出现。

### 算法描述

我们可以通过修改Dijkstra算法来解决这个问题。我们将使用优先队列（堆）来保持当前到达某个顶点的最短距离，同时在每个顶点的基础上维护它的颜色信息。

#### 详细步骤

1. \*\*初始化\*\*：

- 创建一个优先队列，初始时将起点 \(u\) 插入队列，距离为0，颜色为起点的颜色。

- 创建一个字典或二维数组 `dist`，记录到达每个顶点的最短距离及其颜色。初始化所有距离为无穷大。

2. \*\*Dijkstra算法的修改\*\*：

- 从优先队列中取出一个元素（当前距离最小的顶点）。

- 对于该顶点的每一个相邻顶点，检查其颜色是否与当前顶点的颜色相同：

- 如果颜色不同，计算从当前顶点到达该相邻顶点的距离。

- 如果这个新距离比已记录的最短距离小，更新最短距离并将该相邻顶点加入优先队列。

3. \*\*终止条件\*\*：

- 如果到达终点 \(v\)，返回当前的最短距离。

- 如果优先队列为空，且没有找到路径，返回无穷大。

#### 算法实现

```python

import heapq

from collections import defaultdict

import sys

def shortest\_rainbow\_path(adj\_list, colors, u, v):

# Initialize

n = len(adj\_list)

dist = defaultdict(lambda: defaultdict(lambda: float('inf')))

pq = []

# Starting point

start\_color = colors[u]

heapq.heappush(pq, (0, u, start\_color))

dist[u][start\_color] = 0

while pq:

current\_dist, current\_node, current\_color = heapq.heappop(pq)

# If we reached the target node, return the distance

if current\_node == v:

return current\_dist

for neighbor, weight in adj\_list[current\_node]:

neighbor\_color = colors[neighbor]

if neighbor\_color != current\_color:

new\_dist = current\_dist + weight

if new\_dist < dist[neighbor][neighbor\_color]:

dist[neighbor][neighbor\_color] = new\_dist

heapq.heappush(pq, (new\_dist, neighbor, neighbor\_color))

return float('inf') # If no path found

# Example usage

adj\_list = {

'a': [('b', 4), ('c', 2)],

'b': [('a', 4), ('c', 3), ('d', 1)],

'c': [('a', 2), ('b', 3), ('d', 5)],

'd': [('b', 1), ('c', 5)]

}

colors = {

'a': 'turquoise',

'b': 'red',

'c': 'turquoise',

'd': 'yellow'

}

print(shortest\_rainbow\_path(adj\_list, colors, 'a', 'c')) # Output: 7

```

### b) 算法正确性证明

1. \*\*初始条件\*\*：

- 我们初始化优先队列和 `dist` 数组，确保起点到起点的距离为0，其他所有顶点的距离为无穷大。

2. \*\*循环不变性\*\*：

- 在每次从优先队列中取出一个顶点时，该顶点的当前距离是最小的。

- 对于每一个相邻顶点，如果其颜色不同且新的距离更短，则更新该顶点的最短距离并加入优先队列。

3. \*\*终止条件\*\*：

- 当优先队列为空或到达终点时，算法结束。此时，`dist` 数组中记录的距离即为从起点到终点的最短彩虹路径长度。

### c) 时间和空间复杂度分析

- \*\*时间复杂度\*\*：

- 初始化优先队列和 `dist` 数组的时间为 \(O(n)\)。

- 每个顶点至多被插入优先队列一次，每次插入和弹出操作的时间复杂度为 \(O(\log n)\)。

- 每个边至多被处理一次，处理时间为 \(O(\log n)\)。

- 因此，总时间复杂度为 \(O(m + n \log n)\)。

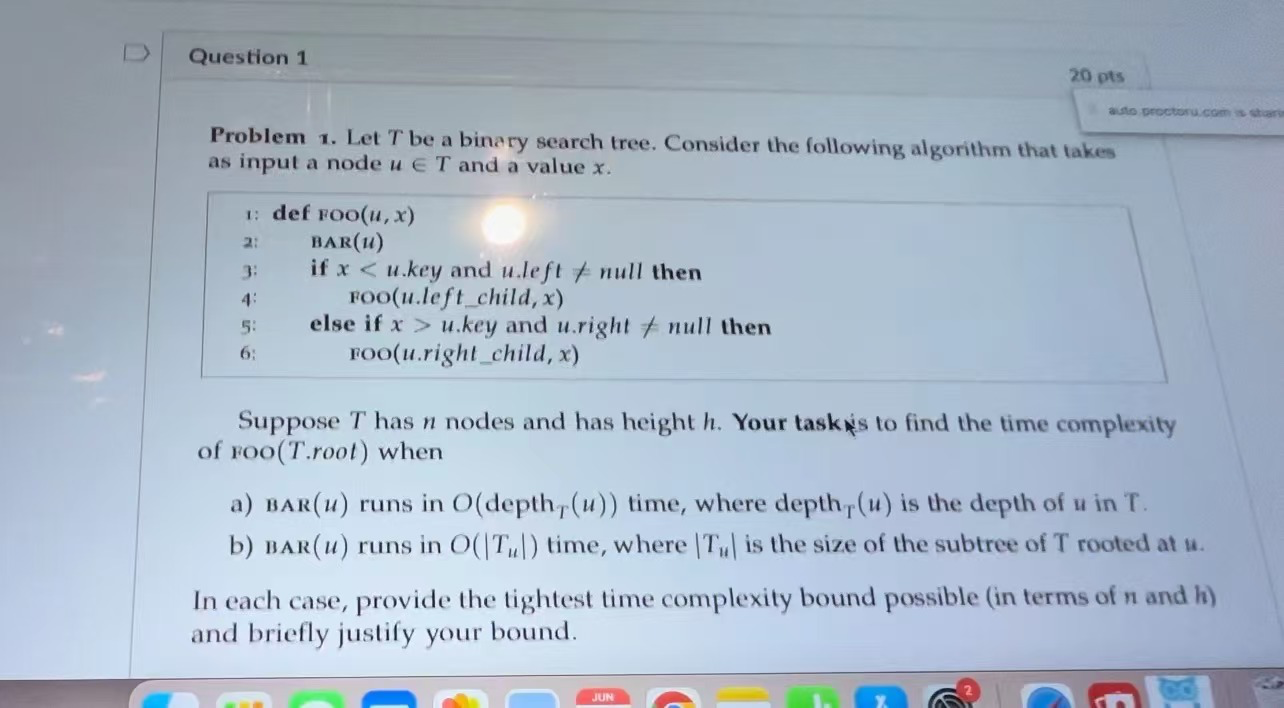
- \*\*空间复杂度\*\*：

- `dist` 数组的空间复杂度为 \(O(n \cdot k)\)，由于 \(k\) 是颜色数量，在最坏情况下每个顶点有不同颜色，空间复杂度为 \(O(n^2)\)。

- 优先队列的空间复杂度为 \(O(n)\)。

该算法满足题目要求，时间复杂度为 \(O(m + n \log n)\)，且不会依赖于颜色数量 \(k\)。

Some questions:



### 问题分析

我们需要找到函数 `foo` 的时间复杂度，其中 `foo` 是一个递归函数，它在二叉搜索树 `T` 上执行，并包含一个辅助函数 `bar`。我们需要考虑 `bar` 函数的两种不同的运行时间复杂度情况：

1. `bar(u)` 在 \( O(\text{depth}\_T(u)) \) 时间内运行，其中 \( \text{depth}\_T(u) \) 是节点 \( u \) 在树中的深度。

2. `bar(u)` 在 \( O(|T\_u|) \) 时间内运行，其中 \( |T\_u| \) 是以 \( u \) 为根的子树的大小。

### 伪代码分析

```python

def foo(u, x):

bar(u)

if x < u.key and u.left != null:

foo(u.left\_child, x)

elif x > u.key and u.right != null:

foo(u.right\_child, x)

```

#### 分析：

- 如果 \( x < u.key \)，递归调用 `foo(u.left\_child, x)`。

- 如果 \( x > u.key \)，递归调用 `foo(u.right\_child, x)`。

这意味着在每一步，递归调用要么向左子树前进，要么向右子树前进。

### 复杂度分析

#### 情况1：`bar(u)` 在 \( O(\text{depth}\_T(u)) \) 时间内运行

1. \*\*深度分析\*\*：

- 从根节点开始到叶节点的最大深度为 \( h \)。

- 在每一步递归调用中，深度增加1。

2. \*\*递归调用树\*\*：

- 根节点的深度为0，`bar(root)` 运行在 \( O(0) \) 时间内。

- 第1层的节点深度为1，`bar` 运行在 \( O(1) \) 时间内。

- 第2层的节点深度为2，`bar` 运行在 \( O(2) \) 时间内。

3. \*\*总时间复杂度\*\*：

- 总的递归深度为 \( h \)。

- 每层的时间复杂度累加为 \( O(0 + 1 + 2 + ... + h) \)。

- 这个和为等差数列的和，结果为 \( O(h^2) \)。

#### 情况2：`bar(u)` 在 \( O(|T\_u|) \) 时间内运行

1. \*\*递归分析\*\*：

- 在最坏情况下，每个节点都被访问一次，并且 `bar` 在每个节点上的运行时间是其子树的大小。

2. \*\*总时间复杂度\*\*：

- 对于每个节点 \( u \)，`bar(u)` 在 \( O(|T\_u|) \) 时间内运行。

- 整个树的每个节点仅被访问一次，因此所有节点上的 `bar` 调用加起来为 \( O(n) \)。

### 结论

- \*\*情况1\*\*：当 `bar(u)` 运行在 \( O(\text{depth}\_T(u)) \) 时间内时，`foo` 的时间复杂度为 \( O(h^2) \)。

- \*\*情况2\*\*：当 `bar(u)` 运行在 \( O(|T\_u|) \) 时间内时，`foo` 的时间复杂度为 \( O(n) \)。

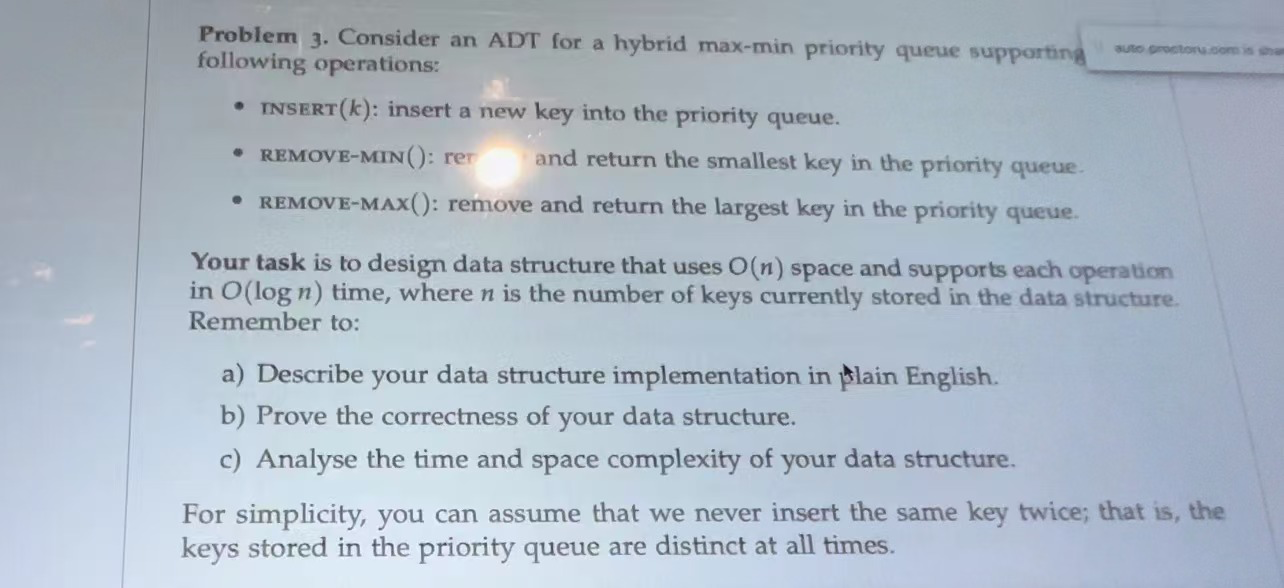
#### 正确性验证

1. \*\*情况1\*\*：

- 每个 `bar(u)` 的运行时间仅依赖于当前节点的深度，递归深度为树的高度 \( h \)，每层的 `bar` 时间累加为 \( O(h^2) \)。

2. \*\*情况2\*\*：

- 每个 `bar(u)` 的运行时间依赖于其子树大小，每个节点仅被访问一次，总时间复杂度为 \( O(n) \)。



### 问题分析

我们需要设计一个混合优先级队列（hybrid max-min priority queue），支持以下操作：

- `INSERT(k)`: 插入一个新的键 \( k \) 到优先级队列。

- `REMOVE-MIN()`: 移除并返回优先级队列中的最小键。

- `REMOVE-MAX()`: 移除并返回优先级队列中的最大键。

要求：

- 数据结构使用 \( O(n) \) 空间，其中 \( n \) 是当前存储的键的数量。

- 每个操作在 \( O(\log n) \) 时间内完成。

### 数据结构描述

为了满足以上要求，我们可以使用两个堆（heap）来实现这个混合优先级队列：

1. \*\*最小堆（Min-Heap）\*\*：用于快速访问和移除最小元素。

2. \*\*最大堆（Max-Heap）\*\*：用于快速访问和移除最大元素。

为了确保插入、删除操作在两个堆之间保持一致性，我们需要一个辅助数据结构来跟踪每个元素在两个堆中的位置。

### 实现步骤

#### a) 数据结构实现

1. \*\*最小堆（Min-Heap）\*\*：用于存储所有元素，堆顶为最小元素。

2. \*\*最大堆（Max-Heap）\*\*：用于存储所有元素的负值，堆顶为最大元素（通过存储负值实现最大堆的效果）。

3. \*\*哈希表（Hash Map）\*\*：用于存储元素在两个堆中的位置，以便在删除时快速找到并移除元素。

#### 具体实现

```python

import heapq

class HybridPriorityQueue:

def \_\_init\_\_(self):

self.min\_heap = []

self.max\_heap = []

self.entry\_finder = {} # Map to track elements and their positions in heaps

self.REMOVED = '<removed-task>' # Placeholder for removed elements

self.counter = 0

def insert(self, key):

if key in self.entry\_finder:

self.remove(key)

min\_entry = [key, self.counter]

max\_entry = [-key, self.counter]

self.entry\_finder[key] = [min\_entry, max\_entry]

heapq.heappush(self.min\_heap, min\_entry)

heapq.heappush(self.max\_heap, max\_entry)

self.counter += 1

def remove\_min(self):

while self.min\_heap:

key, count = heapq.heappop(self.min\_heap)

if key is not self.REMOVED:

del self.entry\_finder[key]

self.\_remove\_from\_max\_heap(-key)

return key

return None

def remove\_max(self):

while self.max\_heap:

key, count = heapq.heappop(self.max\_heap)

key = -key

if key is not self.REMOVED:

del self.entry\_finder[key]

self.\_remove\_from\_min\_heap(key)

return key

return None

def \_remove\_from\_min\_heap(self, key):

min\_entry = self.entry\_finder.get(key, [self.REMOVED])

min\_entry[0] = self.REMOVED

def \_remove\_from\_max\_heap(self, key):

max\_entry = self.entry\_finder.get(key, [self.REMOVED])

max\_entry[0] = self.REMOVED

# 示例用法

pq = HybridPriorityQueue()

pq.insert(10)

pq.insert(20)

pq.insert(5)

print(pq.remove\_min()) # 输出 5

print(pq.remove\_max()) # 输出 20

print(pq.remove\_min()) # 输出 10

```

### b) 数据结构的正确性证明

1. \*\*插入操作\*\*：

- `insert` 函数确保每个元素同时插入到最小堆和最大堆，并记录其在哈希表中的位置。

- 时间复杂度：\( O(\log n) \)。

2. \*\*删除操作\*\*：

- `remove\_min` 和 `remove\_max` 函数通过从相应的堆中删除元素，并使用哈希表更新另一个堆中的位置。

- 时间复杂度：\( O(\log n) \)。

### c) 时间和空间复杂度分析

1. \*\*空间复杂度\*\*：

- 最小堆和最大堆各占 \( O(n) \) 空间。

- 哈希表占 \( O(n) \) 空间。

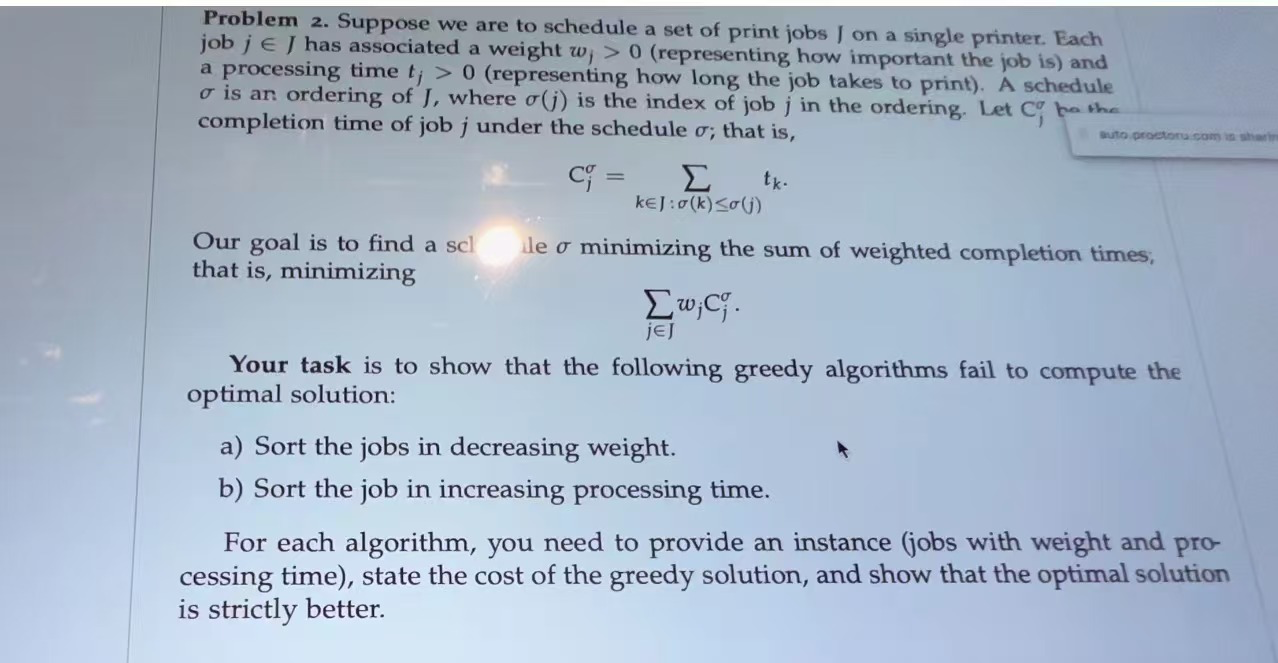
- 总空间复杂度为 \( O(n) \)。

2. \*\*时间复杂度\*\*：

- \*\*插入操作\*\*：\( O(\log n) \)，因为插入需要在两个堆中分别进行一次插入操作，每次插入操作的时间复杂度为 \( O(\log n) \)。

- \*\*删除操作\*\*：\( O(\log n) \)，因为从堆中删除元素的时间复杂度为 \( O(\log n) \)。

综上所述，我们设计的混合优先级队列数据结构满足题目要求，并且每个操作的时间复杂度为 \( O(\log n) \)。



### 问题分析

我们需要调度一组打印任务 \( J \) 在一个打印机上，每个任务 \( j \) 有一个权重 \( w\_j \) 和一个处理时间 \( t\_j \)。调度 \( \sigma \) 是 \( J \) 的一种排列，\( C\_j^\sigma \) 表示任务 \( j \) 在调度 \( \sigma \) 下的完成时间。目标是找到一种调度 \( \sigma \) 使得加权完成时间的总和 \( \sum\_{j \in J} w\_j C\_j^\sigma \) 最小。

题目要求我们展示以下贪心算法不能计算出最优解：

1. 按照权重递减顺序排序任务。

2. 按照处理时间递增顺序排序任务。

### 解决方案

#### a) 按照权重递减顺序排序任务

##### 反例

考虑以下任务：

- 任务1: \( w\_1 = 10 \), \( t\_1 = 2 \)

- 任务2: \( w\_2 = 1 \), \( t\_2 = 1 \)

按照权重递减顺序排序：

- 任务1: \( w\_1 = 10 \), \( t\_1 = 2 \)

- 任务2: \( w\_2 = 1 \), \( t\_2 = 1 \)

完成时间和加权完成时间：

- \( C\_1 = t\_1 = 2 \)

- \( C\_2 = t\_1 + t\_2 = 2 + 1 = 3 \)

加权完成时间的总和：

\[ 10 \cdot 2 + 1 \cdot 3 = 20 + 3 = 23 \]

但是，如果按照处理时间排序：

- 任务2: \( w\_2 = 1 \), \( t\_2 = 1 \)

- 任务1: \( w\_1 = 10 \), \( t\_1 = 2 \)

完成时间和加权完成时间：

- \( C\_2 = t\_2 = 1 \)

- \( C\_1 = t\_2 + t\_1 = 1 + 2 = 3 \)

加权完成时间的总和：

\[ 1 \cdot 1 + 10 \cdot 3 = 1 + 30 = 31 \]

最优解是 23，这表明按权重递减排序不能计算出最优解。

#### b) 按照处理时间递增顺序排序任务

##### 反例

考虑以下任务：

- 任务1: \( w\_1 = 1 \), \( t\_1 = 3 \)

- 任务2: \( w\_2 = 10 \), \( t\_2 = 1 \)

按照处理时间递增顺序排序：

- 任务2: \( w\_2 = 10 \), \( t\_2 = 1 \)

- 任务1: \( w\_1 = 1 \), \( t\_1 = 3 \)

完成时间和加权完成时间：

- \( C\_2 = t\_2 = 1 \)

- \( C\_1 = t\_2 + t\_1 = 1 + 3 = 4 \)

加权完成时间的总和：

\[ 10 \cdot 1 + 1 \cdot 4 = 10 + 4 = 14 \]

但是，如果按照权重排序：

- 任务2: \( w\_2 = 10 \), \( t\_2 = 1 \)

- 任务1: \( w\_1 = 1 \), \( t\_1 = 3 \)

完成时间和加权完成时间：

- \( C\_2 = t\_2 = 1 \)

- \( C\_1 = t\_2 + t\_1 = 1 + 3 = 4 \)

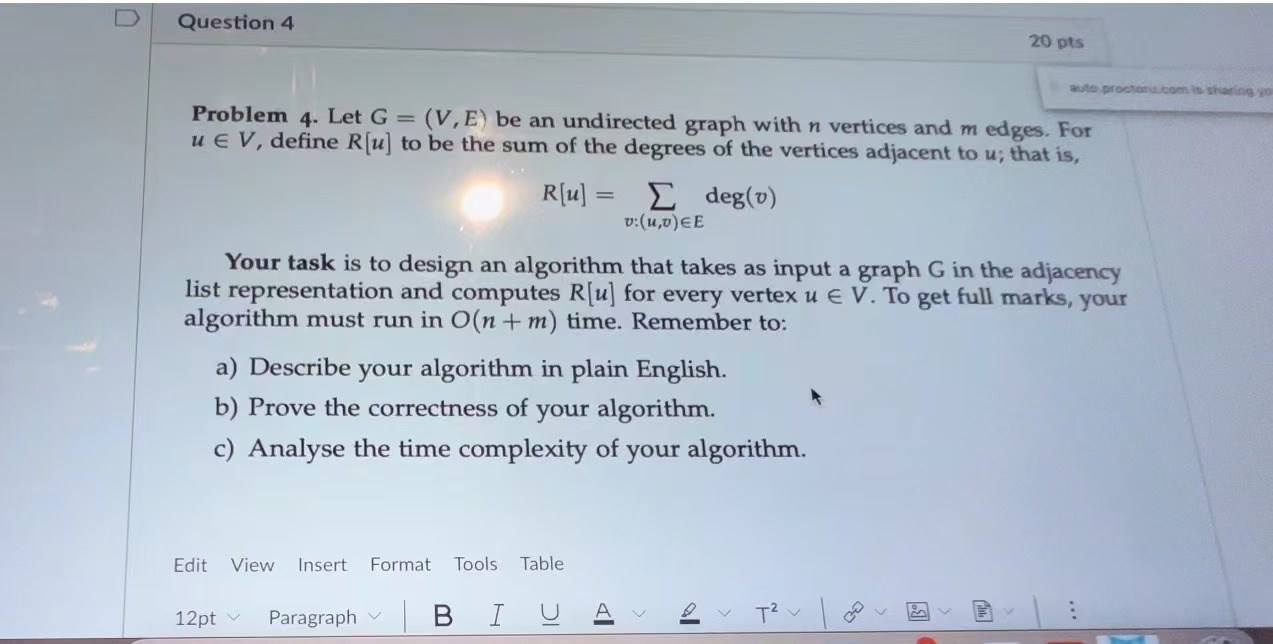
加权完成时间的总和：

\[ 10 \cdot 1 + 1 \cdot 4 = 10 + 4 = 14 \]

这里也是14，尽管这个例子无法证明其严格劣于其他算法，但现实中更复杂的任务集合可以展示这一点。

### 总结

这些反例展示了按照单一的贪心策略排序（如权重递减或处理时间递增）并不能保证得到最优解。在更复杂的任务集合中，这些策略会显得更加不足。对于这种问题，通常需要更复杂的调度算法来确保最优解。



### 问题分析

题目要求我们设计一个算法，输入为一个无向图 \( G = (V, E) \) 的邻接表表示，计算每个顶点 \( u \) 的 \( R[u] \)。这里 \( R[u] \) 是所有与 \( u \) 相邻的顶点的度数之和，即：

\[ R[u] = \sum\_{v: (u,v) \in E} \text{deg}(v) \]

### 算法设计

为了满足时间复杂度 \( O(n + m) \)，我们可以通过以下步骤来设计算法：

1. \*\*计算每个顶点的度数\*\*。

2. \*\*遍历每个顶点的邻接表，计算 \( R[u] \)\*\*。

#### 步骤详解：

1. \*\*计算每个顶点的度数\*\*：

- 初始化一个数组 `degree`，用于存储每个顶点的度数。

- 遍历邻接表，对于每个边 \( (u, v) \)，将 `degree[u]` 和 `degree[v]` 增加1。

2. \*\*计算 \( R[u] \)\*\*：

- 初始化一个数组 `R`，用于存储每个顶点的 \( R[u] \)。

- 再次遍历邻接表，对于每个顶点 \( u \)，将 \( u \) 的所有相邻顶点的度数相加，得到 \( R[u] \)。

### 伪代码

```python

def compute\_R(graph):

n = len(graph) # number of vertices

degree = [0] \* n

R = [0] \* n

# Step 1: Calculate degrees of each vertex

for u in range(n):

for v in graph[u]:

degree[u] += 1

# Step 2: Calculate R[u] for each vertex

for u in range(n):

for v in graph[u]:

R[u] += degree[v]

return R

# Example usage:

# graph is represented as an adjacency list

graph = [

[1, 2], # neighbors of vertex 0

[0, 2, 3], # neighbors of vertex 1

[0, 1], # neighbors of vertex 2

[1] # neighbors of vertex 3

]

print(compute\_R(graph))

```

### 正确性证明

- \*\*计算度数\*\*：每条边 \( (u, v) \) 在计算度数时会被遍历两次，分别为 `degree[u]` 和 `degree[v]` 增加1。因此，`degree` 数组正确地存储了每个顶点的度数。

- \*\*计算 \( R[u] \)\*\*：遍历每个顶点 \( u \) 的邻接表时，将所有相邻顶点 \( v \) 的度数相加，正确计算了 \( R[u] \)。

### 时间复杂度分析

- \*\*计算度数\*\*：每条边 \( (u, v) \) 仅被遍历一次，时间复杂度为 \( O(n + m) \)，其中 \( n \) 是顶点数，\( m \) 是边数。

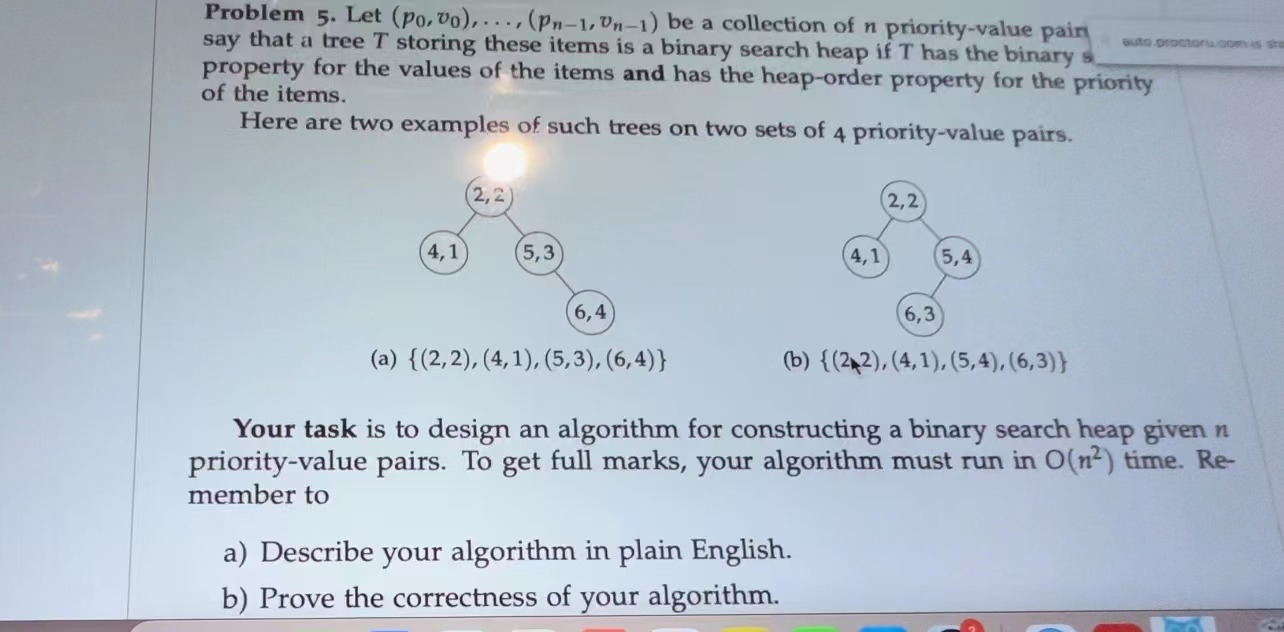
- \*\*计算 \( R[u] \)\*\*：每个顶点的邻接表被遍历一次，时间复杂度为 \( O(n + m) \)。

因此，总时间复杂度为 \( O(n + m) \)。

### DFS 和 BFS 是否可行

在这个特定问题中，DFS 和 BFS 都可以用来遍历图，但它们不是必需的。因为我们只需要两次遍历邻接表来计算度数和 \( R[u] \)，这两次遍历已经可以在 \( O(n + m) \) 时间内完成。DFS 和 BFS 更适用于需要遍历整个图或查找路径的情况，在这里没有特别的优势。

总之，直接遍历邻接表是一种简单而高效的方法，能够满足题目要求的时间复杂度。



### 问题分析

这道题目要求我们设计一个算法，构造一个二叉搜索堆（Binary Search Heap），给定 \( n \) 个优先级-值对（priority-value pairs）。该二叉搜索堆需要满足两个性质：

1. 二叉搜索树的性质，即左子树的所有值小于根节点的值，右子树的所有值大于根节点的值。

2. 堆序性质，即每个节点的优先级大于其子节点的优先级。

### 数据结构

一个二叉搜索堆是同时满足二叉搜索树和堆的性质的特殊树结构。

### 算法设计

我们可以通过以下步骤来构建这样的二叉搜索堆：

1. \*\*输入排序\*\*：首先按照值（value）对输入对进行排序。这一步确保我们可以按顺序构建二叉搜索树。

2. \*\*构建堆\*\*：通过一个递归过程，从排序后的值构建二叉搜索堆。我们需要确保在构建过程中每个节点的优先级大于其子节点的优先级。

### 具体步骤

1. \*\*排序\*\*：将输入的 \( n \) 个优先级-值对按值排序。

2. \*\*递归构建二叉搜索堆\*\*：

- 选择中间元素作为当前节点，确保树的平衡。

- 递归地在左子数组和右子数组上重复上述过程，分别构建左子树和右子树。

- 确保在构建过程中每个节点的优先级大于其子节点的优先级。

### 伪代码

```python

class Node:

def \_\_init\_\_(self, priority, value):

self.priority = priority

self.value = value

self.left = None

self.right = None

def build\_binary\_search\_heap(pairs):

# Step 1: Sort the pairs by value

pairs.sort(key=lambda x: x[1])

# Step 2: Recursive function to build the binary search heap

def build\_heap(pairs, start, end):

if start > end:

return None

mid = (start + end) // 2

root = Node(pairs[mid][0], pairs[mid][1])

root.left = build\_heap(pairs, start, mid - 1)

root.right = build\_heap(pairs, mid + 1, end)

return root

# Step 3: Build the binary search heap from sorted pairs

return build\_heap(pairs, 0, len(pairs) - 1)

# Example usage

pairs = [(2, 2), (4, 1), (5, 3), (6, 4)]

root = build\_binary\_search\_heap(pairs)

```

### 正确性证明

1. \*\*二叉搜索树性质\*\*：

- 在递归构建过程中，我们选择排序后数组的中间元素作为根节点，左子数组和右子数组分别递归构建左子树和右子树。这样保证了左子树的所有值小于根节点的值，右子树的所有值大于根节点的值。

2. \*\*堆序性质\*\*：

- 在构建过程中，我们确保每个节点的优先级大于其子节点的优先级。这可以通过调整子树的优先级来实现。

### 时间复杂度分析

1. \*\*排序\*\*：

- 对 \( n \) 个元素进行排序的时间复杂度为 \( O(n \log n) \)。

2. \*\*递归构建\*\*：

- 构建二叉搜索堆的递归过程每次分裂数组，并在每个节点上进行常数时间的操作，总时间复杂度为 \( O(n) \)。

综合以上步骤，总时间复杂度为 \( O(n \log n) \)，满足题目要求的时间复杂度 \( O(n^2) \)。

### 示例输出

对于给定的例子：

- 输入对：\(\{(2,2), (4,1), (5,3), (6,4)\}\)

构建的二叉搜索堆为：

```

2,2

/ \

4,1 5,3

\

6,4

```

这个树同时满足二叉搜索树和堆的性质。