PS10-231830106

Problem 1

1. 思路：将P和S都从大到小一一排序好后的数组建立一一对应即可，且需保留S和P中元素的原始索引，帮助函数建立一一对应关系
2. 伪代码

```
function AssignShoes(P, S):
    # step1: 将构建元素和索引配对的元组列表
    Indexed_P = [(i, P[i]) for i in range(n)]
    Indexed_S = [(i, S[i]) for i in range(n)]

    # step2: 对脚尺寸和鞋子尺寸排序
    Indexed_P_sorted = Sort(Indexed_P, 按元组的第二个元素排序)
Indexed_S_sorted = Sort(Indexed_S, 按元组的第二个元素排序)

    # step3: 初始化 π (排列数组)
    π = [0] * n

    # step4: 为每个人分配鞋子
    for i in range(n):
        index = Indexed_P[i][0]
        π[index] = Indexed_S[i][0]

    return π
```

3. 时间复杂度：
构建列表的时间复杂度为$O(n)$，排序的时间复杂度为$O(n \lg n)$，构建π数组的时间复杂度为$O(n)$，因此总时间复杂度为$O(n \lg n)$。
4. 正确性证明：
假设我们有两个排序后的数组：

- $P' = [P'[1], P'[2], \ldots, P'[n]]$（脚尺寸按从小到大排序）。
- $S' = [S'[1], S'[2], \ldots, S'[n]]$（鞋尺寸按从小到大排序）。

目标是通过匹配 $P'[i]$和 $S'[i]$，最小化以下公式：

$$\text{总差值} = \sum_{i=1}^{n} |P'[i] - S'[i]|$$

证明

1. 反设在最优匹配方案中存在一对反序对i、j，使得$P'[i]$和$S'[j]$被分配到不按顺序的鞋子尺寸，且 $i < j$且$P'[i] \leq P'[j]$，但鞋子的匹配是反向的，即 $S'[i] \geq S'[j]$。
2. 将 $P'[i]$和 $P'[j]$的鞋子匹配进行交换：

- 原差值为：

$$|P'[i] - S'[j]| + |P'[j] - S'[i]|$$

- 交换后差值为：

$$|P'[i] - S'[i]| + |P'[j] - S'[j]|$$

3. 由于$P'[i] \leq P'[j]$且$S'[i] \leq S'[j]$，根据三角不等式可以证明：

$$|P'[i] - S'[i]| + |P'[j] - S'[j]| \leq |P'[i] - S'[j]| + |P'[j] - S'[i]|$$

4. 因此，交换后差值会变小或相等，矛盾，原假设不成立。说明在最优的匹配中一定不存在这样一个反序对。因此最优匹配中有

$$\forall i, j \in \{1, 2, \ldots, n\}, \text{ 如果 } P[i] \leq P[j], \text{ 那么 } S[i] \leq S[j]$$

> 即证：最优配对π就是当P排序好时S也排序好

Problem 2

(a):

1. 思路：将tasks按所需时间的从小到大排序即可
2. 伪代码：

```
function scheduleTasks(S):
    Sorted_S = sort(S, key=S[i].pi)
    current_time = 0
    total_time = 0
    for each task ai in Sorted_S:
        ci = current_time + pi
        total_time += ci
        current_time = ci
    return total_time / n
```

3. 时间复杂度：
排序的时间复杂度为$O(n \lg n)$，计算总平均耗时的时间复杂度为$O(n)$，因此总时间复杂度为$O(n \lg n)$

(b):

1. 思路：因为每个程序有个释放时间，因此不能在任意时刻执行任意程序，所以需构建一个以剩余时间为key的优先队列，去搜索当前时刻下可以执行的程序中，剩余时间最短的那个。具体实现中，利用一个clock变量作为时间指示器，在每一个clock我们都检测有没有释放的task，并将task压入优先队列中；同时我们每次执行任务只执行一个clock，执行完后重新压入优先队列中。这样能够确保我们每次执行的task都是剩余时间最短的那个，保证了总平均时间最小。
2. 伪代码

```
function scheduleTasksWithPreemption(S):
    Sort tasks S by release time ri in increasing order
    current_time = 0
    total_time = 0
    priorityQueue = empty priority queue

    clock = 0
    n = length(S)
    completedTasks = 0
    remainingTime = [task.pi for each task in S]

    while completedTasks < n:
        while clock < n and S[clock].ri <= current_time:
            priorityQueue.push(S[clock], remainingTime[clock])
            clock += 1

        if not priorityQueue.isEmpty():
            task = priorityQueue.popMin()
            remainingTime[task.index] -= 1
            current_time += 1

            if remainingTime[task.index] == 0:
                completedTasks += 1
                total_time += current_time
            else:
                priorityQueue.push(task, remainingTime[task.index])
        else:
            if clock < n:
                current_time = S[clock].ri

    return total_time / n
```

3. 正确性证明：显然在没有释放时间的限制下，(a)中的答案，即优先执行剩余时间最短的为最优解，只要通过交换引理证明。本小问的证明类似，如果我们不按照选择当前剩余时间最短的任务去执行，则会存在这样一种情况，即当前执行的任务为a_i下一个执行的

任务为a_j,对应的$p_i \geq p_j$，则只考虑这两个任务时，先执行a_i后执行a_j的总平均用时为$t_1 = \frac{2p_i+p_j}{2}$，先执行a_j后执行a_i的总平均用时则为$t_2 = \frac{2p_j+p_i}{2}$，又因为$p_i \geq p_j$，因此有$t_1 \geq t_2$，可以发现先执行a_j后执行a_i为更优情况，即应该先执行剩余时间更短的task。在这种情况下，本算法的正确性显然可证。

Problem 3

(a):

初始状态:
s.dist = +inf s.parent = NULL
t.dist = +inf t.parent = NULL
x.dist = +inf x.parent = NULL
y.dist = +inf y.parent = NULL
z.dist = +inf z.parent = NULL
R = NULL

1. step 1
   s.dist = 0 s.parent = s
   t.dist = +inf t.parent = NULL
   x.dist = +inf x.parent = NULL
   y.dist = +inf y.parent = NULL
   z.dist = +inf z.parent = NULL
   R = {s}
2. step2
   s.dist = 0 s.parent = s
   t.dist = 4 t.parent = s
   x.dist = +inf x.parent = NULL
   y.dist = 5 y.parent = s
   z.dist = +inf z.parent = NULL
   R={s,t}
3. step3
   s.dist = 0 s.parent = s
   t.dist = 4 t.parent = s
   x.dist = 10 x.parent = t
   y.dist = 5 y.parent = s
   z.dist = +inf z.parent = NULL
   R={s,t,y}
4. step4
   s.dist = 0 s.parent = s
   t.dist = 4 t.parent = s
   x.dist = 9 x.parent = y
   y.dist = 5 y.parent = s

z.dist = 11 z.parent = y
R={s,t,y,x}

5. step5
s.dist = 0 s.parent = s
t.dist = 4 t.parent = s
x.dist = 9 x.parent = y
y.dist = 5 y.parent = s
z.dist = 11 z.parent = y
R={s,t,y,x,z}

(b):

(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)
初始
t.dist = +inf t.parent = NULL
s.dist = +inf s.parent = NULL
x.dist = +inf x.parent = NULL
y.dist = +inf y.parent = NULL
z.dist = +inf z.parent = NULL

1. step1
t.dist = 0 t.parent = t
s.dist = -2 s.parent = z
x.dist = 5 x.parent = t
y.dist = 5 y.parent = s
z.dist = -4 z.parent = t
2. step2
t.dist = 0 t.parent = t
s.dist = -2 s.parent = z
x.dist = 2 x.parent = y
y.dist = 5 y.parent = s
z.dist = -4 z.parent = t
3. step3
t.dist = 0 t.parent = t
s.dist = -2 s.parent = z
x.dist = 2 x.parent = y
y.dist = 5 y.parent = s
z.dist = -4 z.parent = t
没有变化，即为最终答案

(c):

拓扑排序结果rstxyz
初始

r.dist = 0 r.parent = NULL
s.dist = +inf s.parent = NULL
t.dist = +inf t.parent = NULL
x.dist = +inf x.parent = NULL
y.dist = +inf y.parent = NULL
z.dist = +inf z.parent = NULL

1. step1
   r.dist = 0 r.parent = NULL
   s.dist = 5 s.parent = r
   t.dist = 8 t.parent = r
   x.dist = +inf x.parent = NULL
   y.dist = +inf y.parent = NULL
   z.dist = +inf z.parent = NULL
2. step2
   r.dist = 0 r.parent = NULL
   s.dist = 5 s.parent = r
   t.dist = 7 t.parent = s
   x.dist = 11 x.parent = s
   y.dist = +inf y.parent = NULL
   z.dist = +inf z.parent = NULL
3. step3
   s.dist = 5 s.parent = r
   t.dist = 7 t.parent = s
   x.dist = 11 x.parent = s
   y.dist = 11 y.parent = t
   z.dist = 13 z.parent = t
4. step4
   s.dist = 5 s.parent = r
   t.dist = 7 t.parent = s
   x.dist = 11 x.parent = s
   y.dist = 10 y.parent = x
   z.dist = 12 z.parent = x
5. step5
   s.dist = 5 s.parent = r
   t.dist = 7 t.parent = s
   x.dist = 11 x.parent = s
   y.dist = 10 y.parent = x
   z.dist = 8 z.parent = y
   结束

Problem 4

(a):

1. 思路：从s点进行一次dfs且只搜索长度小于L的边，如果能够访问到t，则返回true，不能则返回false。
2. 伪代码：

```
def dfs(graph, u, t, L, visited):
    visited[u] = True
    if u == t:
        return True
    for v, weight in graph[u]:
        if not visited[v] and weight <= L:
            if dfs(graph, v, t, L, visited):
                return True
    return False

def isFeasibleRoute(graph, s, t, L):
    visited = {v: False for v in graph}
    return dfs(graph, s, t, L, visited)
```

3. 时间复杂度：dfs遍历的时间复杂度为$O(|V|+|E|)$，则显然总时间复杂度为$O(|V|+|E|)$。

(b):

1. 思路：使用和dijkstra算法相同的逻辑，不同在于x.dist的定义不再是x到s的最短距离，而是在s到x路径上最大的边的长度。将dist的更新逻辑由和改为取max即可。
2. 伪代码：

```
import heapq

def dijkstraMinTankCapacity(graph, s):
    dist = {v: float('inf') for v in graph}
    dist[s] = 0
    priority_queue = [(0, s)]

    while priority_queue:
        current_capacity, u = heapq.heappop(priority_queue)

        if current_capacity > dist[u]:
            continue
```

```
        for v, weight in graph[u]:
            # 更新dist[v]为当前路径上的最大边长度
            new_capacity = max(current_capacity, weight)
            if new_capacity < dist[v]:
                dist[v] = new_capacity
                heapq.heappush(priority_queue, (new_capacity, v))

    return dist
```

3. 时间复杂度：和dijkstra算法时间复杂度相同，因此总时间复杂度为$O((|V|+|E|)\log|V|)$。

Problem 5

1. 思路：由于所有边的权重都为负，因此所有环都是负环，因此可以通过Tarjan算法找到所有环，再从环上的点出发进行dfs加入负环集内，设置dist为-inf，最后通过dag-sssp算法对无环图进行操作即可。
2. 伪代码

```
Algorithm SPWNW(Graph G, Vertex s):

    dist(v) ← ∞ for all v ∈ V
    dist(s) ← 0
    inNegativeCycle ← False for all v ∈ V
    visited ← False for all v ∈ V
    onStack ← False for all v ∈ V
    stack ← empty stack
    negativeCycleSet ← empty set

Tarjan's Algorithm:
    SCCs ← TarjanSCCs(G)

    for each SCC ∈ SCCs:
        if SCC contains a cycle:
            if the sum of edge weights in the SCC < 0:
                Add all vertices in SCC to negativeCycleSet
                Perform DFS from all vertices in SCC:
                    Mark all reachable vertices as inNegativeCycle

    for each v ∈ negativeCycleSet:
        dist(v) ← −∞
```

```
    Construct reduced graph G' by removing all vertices in
negativeCycleSet
    Perform Topological Sort on G'
    for each vertex u in topological order:
        for each edge (u, v) ∈ G':
            if dist(u) ≠ ∞:
                dist(v) ← min(dist(v), dist(u) + weight(u, v))


    for each vertex v ∈ V:
        if inNegativeCycle[v]:
            print(f"dist({s}, {v}) = -∞")
        else if dist(v) == ∞:
            print(f"dist({s}, {v}) = ∞")
        else:
            print(f"dist({s}, {v}) = {dist(v)}")
```

3. 时间复杂度:
   Tarjan算法的时间复杂度$O(|V|+|E|)$，找到所有在负环上的点的BFS时间复杂度为$O(|V|+|E|)$，DAGSSSP算法的时间复杂度为$O(|V|+|E|)$，因此总时间复杂度为$O(|V|+|E|)$。

Problem 6

1. 思路：通过两次拓扑排序与遍历，第一次从源点正向拓扑遍历确定最早完成时间，第二次从汇点反向拓扑遍历确定最晚完成时间，如果一个点在关键路径上，则其最早完成时间和最晚完成时间一致。
2. 伪代码

```
def earliest_finish_time(graph, weights, source):
    topo_order = topological_sort(graph)
    earliest = {v: 0 for v in graph}

    for u in topo_order:
        for v in graph[u]:
            earliest[v] = max(earliest[v], earliest[u] + weights[v])

    return earliest

def latest_finish_time(graph, weights, sink, earliest):
    topo_order = reversed(topological_sort(graph))
```

```
    latest = {v: float('inf') for v in graph}
    latest[sink] = earliest[sink]

    for u in topo_order:
        for v in graph[u]:
            latest[u] = min(latest[u], latest[v] - weights[u])

    return latest

def critical_path(earliest, latest):
    critical_jobs = []
    for v in earliest:
        if earliest[v] == latest[v]:
            critical_jobs.append(v)

    return critical_jobs

def topological_sort(graph):
    visited = {v: False for v in graph}
    stack = []

    def dfs(u):
        visited[u] = True
        for v in graph[u]:
            if not visited[v]:
                dfs(v)
        stack.append(u)

    for v in graph:
        if not visited[v]:
            dfs(v)

    return stack[::-1]  # 返回逆序的拓扑排序结果
```

3. 时间复杂度：
   拓扑排序的时间复杂度为$O(|V|+|E|)$，一次遍历与更新的时间复杂度为$O(|V|+|E|)$，因此前两个任务的时间复杂度都为$O(|V|+|E|)$。最后一个任务只需要对每个点进行比较即可，时间复杂度为$O(|V|)$，因此总时间复杂度为$O(|V|+|E|)$。