PS9-231830106

Problem 1

(a):

证明：
设这条边为e_{max}。假设存在这样一个最小生成树T，包含了e_{max}，我们可以构造一个生成树T'使得T'不包含e_{max}且权重和小于T。

1. 删除e_{max}：由于e_{max}在T_{max}中，因此删除e_{max}后一定会将图分为两个连通分量A和B。
2. 由于e_{max}在一个环里，因此一定存在这个环里的另一条边不在T_{max}中（否则T_{max}中存在环）且这条边可以连接A和B，设这条边为e_0.
3. 添加e_0：则$T_{max} - e_{max} + e_0$也为G的一个生成树且权重和小于T_{max}
   与假设矛盾，因此最小生成树中不可能包含e_{max}，证毕。

(b):

不同意。
反例：
V={a,b,c,d} E={(a,b)=1,(a,c)=2, (a,d)=3}
A={(a,b)} T={(a,b),(a,c),(c,d)} S={a,b}
显然(a,d)是对于A安全的边，但是(a,d)并非S和V-S的轻边。

(c):

不同意
反例：
V={a,b,c,d} E={(a,b)=1,(a,c)=2, (a,d)=1,(b,c)=3}
E1={a,d},E2={b,c}
Bacon教授算法得出的最小生成树为{(a,d),(a,b),(b,c)}，而实际最小生成树为{(a,d),(a,c),(a,b)}。

Problem 2

(a):

错误
(1,2)(3,4)(1,6)

(b):

错误
(1,6)(2,3)(4,5)

(c):

正确
只需把活动策略完全逆序，会发现这个方法和原版的方法一样，同样正确。

(d):

错误
(1,4)(3,5)(4,8)

(e):

错误
(1,3)(2,4)(2,4)(3,5)(4,6)(5,7)(6,8)(6,8)(7,9)

(f):

错误
(1,4)(3,5)(4,8)

(g):

错误
(1,3)(2,4)(2,4)(3,5)(4,6)(5,7)(6,8)(6,8)(7,9)

(h):

正确：
证明：

1. 证明去除包含关系一定不影响：
   假设存在一个最优的活动安排，又有一个活动A，存在一个活动B，B比A开始晚结束早，则显然将A换为B能安排的活动数量只增不减，因此删除A不会影响最大活动数量。
2. 证明去除包含关系后选择结束时间最晚正确：
   选择结束时间最晚等价于选择开始时间最早，不妨考虑在去除包含关系后选择开始时间最早的活动。只需证明这一次选择的开始时间最早的活动A即为结束时间最早的活动。假设存在一个活动B结束时间比A结束更早，又因为A是开始最早的活动，因此B开始比A晚结束又比A早所以B一定被A包含，但是我们已经去除，矛盾，说明不存在这样一个B结束时间比A早，A就是结束时间最早的活动，因此和原本的贪心相同。
   证毕

Problem 3

1. 思路：利用Prim算法的变种生成一个最大生成树T，则所有不在T中的边显然包含在一个环里，则这些没有被选中的边构成了最小反馈边集F。
2. 证明：已知G=F+(G-F)，若删去F后G中没有环，则说明G-F是一颗生成树，若F的权重和要求最小，而G的权重和不变，则要求G-F的权重和最大，则G-F是一颗最大生成树。而在这最大生成树建立后，假设有某个边p=(u,v)不在T中，显然p的两端u,v已经连通，则加上p后G中一定存在一个环。说明了p一定在F中，证毕。
3. 伪代码：

```
def minimum_feedback_edge_set(G):
    # 1. 计算最大生成树 (可以使用 Kruskal 或 Prim)
    MST = maximum_spanning_tree(G)

    # 2. 初始化反馈边集
    feedback_edges = []

    # 3. 遍历图中的所有边
    for edge in G.edges:
        if edge not in MST.edges:
            feedback_edges.append(edge)

    # 4. 选择反馈边集中的最小权重边
    return feedback_edges
```

Problem 4

(a):

正确：
每次删除一个边(u,v)时，(u,v)删除不会影响G的连通型，因此(u,v)一定在一个环C中，且由于逆序删除，(u,v)一定是C中权重最大的边，因此由problem1的a问可以知道(u,v)一定不在最小生成树T中。又在这种逆序的删除操作后，G中没有环，且最小生成树T中的边没有被删除，则显然剩下的T就是最小生成树

(b):

错误
对于一个环，这种算法不会返回唯一的答案，若环各边权重不一样很显然算法返回值不唯一，与最小生成树定义矛盾，算法错误。

(c):

正确
这个算法保证了返回的T中没有环，且每个环只删一条边，说明T一定是一颗生成树，又删除

的边由于替换保证了删除最大边，因此达成了和a一样的效果，所以和a的正确性一样。

Problem 5

1. 思路：当重量升序与价值降序的排序相同时，性价比最高的物品同时就是最轻的物品，此时问题具有最优子结构性质，则显然一直选择重量最轻的就行
2. 伪代码

```
def greedy_knapsack(items, capacity):
    # items 是一个包含 (weight, value) 元组的列表
    # 按物品的重量升序排序
    items.sort(key=lambda x: x[0])

    total_value = 0
    remaining_capacity = capacity

    for weight, value in items:
        if weight <= remaining_capacity:
            total_value += value
            remaining_capacity -= weight

    return total_value
```

Problem 6

1. 思路：最少需要的颜色数量就是区间最大的重叠数量，因此可以先将左端点和右端点统一认定为事件（开始和结束），再对事件起始时间的升序排序，排序后遍历并维护一个最大重叠数，在一次遍历后即可求出最少需要的颜色数。
2. 伪代码

```
def min_colors(L, R):
    # 将左端点和右端点合并为事件，并按端点排序
    events = []
    for i in range(len(L)):
        a, b = L[i], R[i]
        events.append((a, 'start'))  # 左端点是区间开始
        events.append((b, 'end'))    # 右端点是区间结束

    # 按照端点位置排序，若位置相同，'start' 排在 'end' 前面
    events.sort(key=lambda x: (x[0], x[1] == 'end'))

    # 遍历事件，维护活跃区间数
```

```
    active_intervals = 0
    max_active = 0

    for event in events:
        if event[1] == 'start':
            active_intervals += 1  # 新的区间开始，活跃区间数加1
            max_active = max(max_active, active_intervals)  # 更新最大活跃区间数
        else:
            active_intervals -= 1  # 区间结束，活跃区间数减1

    return max_active  # 最少颜色数即为最大活跃区间数
```