2)

a) True. For any edge pair e_1, e_2, if |e_1| > |e_2| and all e are positive, then |e_1|^2 > |e_2|^2.

b) True. Same reasoning.

4)

```
def is_subsequence(Array S', Array S):
    S'_len = len(S')
    S_len = len(S)
    if S'_len == 0:
        return True #empty sequence is subsequence of all sequences
    s'_index = 0
    s_index = 0
    while s'_index < S'_len and s_index < S_len:
        if S'[s'_index] == S[s_index]:
            s'_index++
            s_index++
        else:
            s_index++
    return s'_index >= S'_len
```

runtime = O(n) + O(m) + O(n) = O(2n + m) = O(n+m)

7)

```
def get_schedule(Jobs[] J, SuperComputerTime[] P, PCTime[] F, int n):
    len_P = len(P)
    sorted_P = sorted([(i, P[i]) for i in range(len(P))], key=lambda x: x[1]) #O(n) + O(nlogn) for the sort
    sorted_all = []
    index = 0
    temp = []
    # these two while statements both iterate through the same list, and so are O(n) combined
    while index < len_P:
        current_p = sorted_P[index][1]
        while index < len_P and current_p == sorted_P[index][1]:
            temp.append(sorted_P[index])
            index++
        temp = sorted(temp, key=lambda x: F[x[0]], reverse=True) #sorts by finishing time, longest first
        sorted_all += temp
        temp = []
        index++
```

runtime = O(n + n log n + n^2 log n) = O(n^2 log n) = O(n^2 log n)

9)

a) No. Let G = ((v_1, v_2, v_3, v_4, v_5), (v_a, v_b, a+b) for every a, b in |v| where a != b)

Then a minimum bottleneck tree is ((v1, v2, v3, v4, v5), ((v_1, v_5, 6), (v_1, v_4, 5), (v_4, v_2, 6), (v_2, v_3, 5)) is a minimum bottleneck tree, but not a minimum spanning tree.

b) Yes. If you have a minimum tree T, and a minimum tree T' with a lower bottleneck e', then T contains an edge e with a higher bottleneck. If you add e' to T and remove e, then the new tree T'' is a smaller tree than T, which is a contradiction.

14)

a)

```
def get_status_check_times(Times sensitive): #assumes times are sorted by start time
	first_start = sensitive[0].start
	first_end = sensitive[0].end
	current = 1
	len_s = len(sensitive)
	times = []
	while current < len_s:
		n = False
		current_item = sensitive[current]
		if first_end < current_item.start:
			current++
		if first_end >= current_item.start:
			times.append(current)
			first_start = sensitive[current+1].start
			first_end = sensitive[current+1].end
			current++
		if first_end > current_item.end:
			first_end = current_item.end
			current++
	return times
```

b) Yes. If k* is the largest set of processes that never overlap, then all other processes *must* overlap with those. Therefor, the invocations can be timed for the overlaps with no more than k* invocations.

16)

```
def check_account(Transactions x, Transactions suspect, Margins e): #assumes sorted
	count = 0
	for i in xrange(len(x)):
		start = suspect[count] - e[count]
		end = suspect[count] + e[count]
		if x[i].time > end:
			return False
		elif x[i].time < start:
			continue
```

```
        else:
            count++
    return True
```

23)

Yes. For any e in A, e is the edge of some minimum cost arborescence A'. If there was an edge e could be replaced with in A, it would also be replaced by it in A'. So A must be a minimum cost arborescence.

24)

Treenode looks like

```
class TreeNode():
    __init__(self):
        left_path = value
        right_path = value
        left = TreeNode #None if leaf
        right = TreeNode
        max_left_path = None
        max_right_path = None

def fix_tree(Tree t):

    def eval_tree(Tree h):
        if h.left ==None: #this is a leaf
            return 0

        h.max_left_path = left_path + eval_tree(h.left)
        h.max_right_path = right_path + eval_tree(h.right)

        return max(h.max_left_path, h.max_right_path)

    def fix_tree(Tree h):
        if h.max_left_path > h.max_right_path:
            h.right_path += h.max_left_path - h.max_right_path
        elif h.max_left_path < h.max_right_path
            h.left_path += h.max_right_path - h.max_left_path
        fix_tree(h.left)
        fix_tree(h.right)

    eval_tree(t)
    fix_tree(t)
```