Spring 2019 CS372 Assignment 3 solutions

1) Given:

```
Parallel-Product(A[1..n])              cost    times (if n>1)

if n = 1 then return                   c1      1
for i := 1 to n/2 do                   c2      n/2 + 1
     A[i] = A[i]*A[i+n/2]              c3      n/2
Parallel-Product(A[1..n/2])            T(n/2)  1
```

Let T(n) denote the running time of `Parallel-Product` on the input of size n.
Then, a recurrence relation for T(n) is T(n) = T(n/2) + O(n).

2.1) *Answer:*
5, 9.
Explanation: everything to the left of x must be < x, and everything to the right of x must be > x.

2.2) *Answer:* When all the elements in A are the same, notice that the comparison in line 4 of PARTITION is always satisfied and i therefore is incremented in each iteration. Since initially i = p - 1 and i + 1 is returned the returned value is r.

2.3) *Answer:* If the elements in A are the same, then by question 1.2 the returned element from each call to PARTITION(A, p, r) is r thus yielding the worst-case partitioning. The recurrence for the worst-case partitioning is T(n) = T(n-1) + Θ(n). The total running time is $O(n^2)$.

2.4) *Answer:* $T(n) = T(\frac{2n}{9}) + T\left(\frac{7n}{9}\right) + cn$

3) *Answer:*

The following is a general outline of the algorithm followed by pseudo-code.

- We sort A (this takes O(n log n) time)
- We sort B (this takes O(n log n) time)
- We iterate through A and B with two pointers, always advancing the one pointing to the smaller value. When they point to equal values, then print the common value and advance both pointers. It takes O(n) time.

The running time of this algorithm is O(n log n + n log n + n) = O(n log n).

Algorithm CommonElements(A, B, n)
Input: array A, array B, size n of A and B

```
MergeSort(A)
MergeSort(B)
a_index = 0
b_index = 0
while (a_index<n and b_index<n) do
        if (A[a_index] = B[b_index]) then
                print A[a_index]
                a_index = a_index+1
                b_index = b_index +1
        else if (A[a_index] < B[b_index]) then
                        a_index = a_index +1
             else
                        b_index = b_index +1
```

4) *Answer:*

First examine the middle element A[n/2]. If A[n/2] = n/2 , we are done. If A[n/2] > n/2 , then every subsequent element will also be bigger than its index since the array values grow at least as fast as the indices. Similarly, if A[n/2] < n/2, then every previous element in the array will be less than its index by the same reasoning. So after the comparison, we only need to examine half of the array. We can recurse on the appropriate half of the array. If we continue this division until we get down to a single element and this element does not have the desired property, then such an element does not exist in A.
The algorithm is presented below. Initial call is Equal_Index(A,1,n).

```
Equal_Index (A, p, r) {
        if (p>r) {
                return 0;
        } else {
                midpoint = (p+r)/2;
                if (A[midpoint] == midpoint){
                        return 1;
                } else if (A[midpoint] > midpoint) {
                        Equal_Index(A, p, midpoint-1);
                }else{ // A[midpoint] < midpoint
                        Equal_Index(A, midpoint+1, r);
                }
        }
}
```

We do a constant amount of work with each function call. So our recurrence relation is
T(n) = T(n/2)+O(1). By the Master Theorem T(n) = O(log n).

5) *Answer:*

a) Let T(i) be the time to merge arrays 1 to i. This consists of the time taken to merge arrays 1 to i-1 and the time taken to merge the resulting array of size (i-1)n with the ith array. Thus, there exists some constant, c, such that $T(i) \leq T(i-1) + cni$. Hence,

$T(k) \leq T(1) + cn \sum_{i=2}^{k} i = O(nk^2)$.

b) Divide the array into two sets, each of k/2 arrays. Recursively merge the arrays within the two sets and finally merge the resulting two sorted arrays into the output array. The base case of the recursion is k = 1, when no merging needs to take place. The running time is given by $T(k) = 2\mathrm{T}(k/2) + O(nk)$. Thus, by the Master Theorem, $T(k) = O(nklog(k))$.

6) *Answer:*

We keep M + 1 counters, one for each of the possible values of the array elements. We can use these counters to compute the number of elements of each value by a single O(n)-time pass through the array. Then, we can obtain a sorted version of x by filling a new array with the prescribed numbers of elements of each value, looping through the values in ascending order.

The Ω (n log n) bound does not apply in this case, as this algorithm is not comparison-based.