Solutions to the Midterm Exam

Solution 1:

(a) There are n calls to the output statement for each recursive call, except when $n = 1$ and then there are 0 calls. There are three recursive to subarrays of length $n/3$, and so this leads to the following recurrence, where n is a power of 3.

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 3T(n/3) + n \qquad \text{for } n > 1. \end{aligned}$$

(b) We solve this by expansion (substitution).

$$\begin{aligned} T(n) &= 3T(n/3) + n \\ &= 3(3T(n/9) + n/3) + n \;=\; 9T(n/9) + 2n \\ &= 9(3T(n/27) + n/9) + 2n \;=\; 27T(n/27) + 3n \\ &= \ldots \\ &= 3^k T(n/3^k) + kn. \end{aligned}$$

Since we know $T(1) = 0$, we let $k = \log_3 n$. This gives us

$$\begin{aligned} T(n) &= 3^{\log_3 n} T(1) + n(\log_3 n) \\ &= n \log_3 n. \end{aligned}$$

This is $\Theta(n \log n)$. We could also get an asymptotic bound by applying the Master Theorem. We have $a = b = 3$ and $k = 1$. Since $a = b^k$ it follows that the running time is $\Theta(n \log n)$.

Solution 2: This can be done in $O(V + E)$ time by DFS. Start a DFS of G at vertex s. We detect and delete each back edge that is incident to s.

```
DFSvisit(Graph G, Vertex u) {
  color[u] = gray
  for each (v in Adj(u)) {
    if (color[v] == white) {        // (u,v) is a tree edge
      pred[v] = u
      DFSvisit(v)
    }
    else if (v != pred[u]) {        // (u,v) is a back edge
      if (v == s)
        delete (u,v) (and of course (v,u)) from G
    }
  }
}
```

To establish correctness we claim that (1) the resulting graph G' is connected and (2) s is not in any simple cycle. To see the first claim, observe that because we have only deleted back edges, the graph is still connected through its tree edges. To prove the second claim, observe that in any DFS tree, if there is no back edge to the root, then the root cannot be in any simple cycle. The reason is that any path that leaves the root descends into one of its subtrees along some edge e. Because there are no cross edges in the DFS of a undirected tree, the only way to return to the root is along edge e, but then such a cycle would not be simple. Since each back edge to s generates a simple cycle, every edge we delete lies on a simple cycle through s.

Solution 3:

(a) We will prove this by contradiction. Suppose that $e = (u, v)$ is the maximum weight edge on some simple cycle. We claim that e cannot be in the MST of G. Suppose to the contrary that this edge is in the MST, denoted T. Remove this edge from the MST. This splits the MST into two subtrees T_u and T_v, where u is in T_u and v is in T_v. If we follow the path from u to v on the cycle, there is at least on edge (x, y) on this cycle such that $x \in T_v$ and $y \in T_u$. By removing (u, v) and adding edge (x, y) we form a new spanning tree T' whose weight is

$$w(T') = w(T) - w(u, v) + w(x, y).$$

Since (u, v) is the maximum weight edge on the cycle and since edge weights are distinct, it follows that $w(T') < w(T)$, but this contradicts the hypothesis that T is the MST.

(b) If the weight of an edge $e = (u, v)$ that is not in the MST decreases, then the MST may change. In particular, its addition creates a simple cycle, and the highest weight edge on this cycle must be deleted. The MST can be updated in $O(V + E)$ time. We first compute the path from u to v in the original MST. One way to do this is to run a DFS starting at u, but ignoring all but the marked edges of G. When v is discovered (it must be, because the spanning tree is connected) follow the predecessor links to reconstruct the path. Then determine the maximum weight edge e' on this path. If this weight is greater than $w(e)$, then replace e' with e (by marking e and unmarking e'). Otherwise, leave the MST unchanged.

To see why this is correct, observe that the addition of e creates a simple cycle. From part (a) we know that the highest weight edge on this cycle must be deleted. Once this heavy edge is removed, we are left with a spanning tree. To show that this is the new MST, it suffices to argue that none of the non-MST graph edges could enter the new MST. The reason is that each non-MST edge e'' was not in the MST because it was the heaviest edge on some simple cycle. By decreasing the weight of e, each such edge is still the heaviest weight on some cycle.

(c) If the weight of a nontree edge e increases, then the MST does not change. The reason is that e was not in the MST becasue it was the maximum weight edge on some simple cycle. By increasing its weight, it is still the maximum weight edge on this cycle. Since the weights of the other edges have not changed, the MST is unchanged.

Solution 4:

(a) In the worst case (of an inversely sorted array) every pair of elements is inverted. Thus the number of inversions is equal to the number of combinations of 2 elements from an n element set, that is $n(n - 1)/2$.

(b) This is done by a modification of the Merge procedure used in MergeSort. It merges two arrays A and B and returns the result in an array C. As we merge the two lists, whenever we discover that element $B[j]$ lies between $A[i-1]$ and $A[i]$ we know that it is inverted with respect to the $m-i+1$ elements of $A[i..m]$. We add this value to a counter. We return the value of the counter and the sorted array C.

```
InvertMerge(array A, array B, int m) {
  array C[1..2m]              // holds final sorted list
  i = j = k = 1
  invCount = 0                // inversion counter
  while (i < m && j < m) {
    if (A[i] <= B[j]) C[k++] = A[i++]
    else {
      iCount += m-i+1          // update counter
      C[k++] = B[j++]
    }
  }
  while (i < m) C[k++] = A[i++] // copy remainder of list
  while (j < m) C[k++] = B[j++]
  return (C, invCount)
}
```

The procedure runs in $O(m)$ time since each iteration of each loop either copies an element of A or an element of B, and there are at most $2m$ total elements.

(c) The algorithm uses divide-and-conquer. It partitions the array A into two subarrays, it invokes itself recursively on each subarray to count the number of inversions and sort them. Then it then invokes `InverMerge()` to the two subarrays, to merge them and count the number of inversions between them.

```
Inversion(array A, int n) {
  if (n == 1) return 0            // single element, no inversions
  m = n/2
  invLeft  = Inversion(A[1..m])   // inversions on left side
  invRight = Inversion(A[m+1..n]) // inversions on right side
  (A, invBetween) = InvertMerge(A[1..m], A[m+1..n], m)
  return invLeft + invRight + invBetween
}
```

The running time is the same as MergeSort, which is $O(n \log n)$.