CS 570: Analysis of Algorithms – H1

Submitted by: Indronil Bhattacharjee

Problem 2-2

```
BUBBLESORT(A, n)
  for i=1 to n−1
    for j=n downto i+1
      if A[j] < A[j−1]
        exchange A[j] with A[j−1]
```

a) Let A′ denote the array A after BUBBLESORT(A, n) is executed. To prove that, A′[1] ≤ A′[2] ≤ … ≤ A′[n].

Besides this, else we need to prove that A′ contains the same elements as A, which is easily seen to be true because the only modification we make to A is swapping its elements, so the resulting array must contain a rearrangement of the elements in the original array.

b) Loop invariant: At the start of each iteration of the for loop of lines 2–4, $A[j] = \min\{A[k]: j \le k \le n\}$ and the subarray $A[j, \ldots, n]$ is a permutation of the values that were in $A[j, \ldots, n]$ at the time that the loop started.

Initialization: Initially, j=n, and the subarray $A[j, \ldots, n]$ consists of single element $A[n]$. The loop invariant trivially holds.

Maintenance: Considering an iteration for a given value of j. By the loop invariant, A[j] is the smallest value in A[j, …, n]. Lines 3–4 exchange $A[j]$ and $A[j\text{-}1]$ if $A[j]$ is less than $A[j\text{-}1]$, and so $A[j\text{-}1]$ will be the smallest value in $A[j\text{-}1, \ldots, n]$ afterward. Since the only change to the sub array $A[j\text{-}1, \ldots, n]$ is this possible exchange, and the subarray $A[j, \ldots, n]$ is a permutation of the values that were in $A[j, \ldots, n]$ at the time that the loop started, we see that $A[j\text{-}1, \ldots, n]$ is a permutation of the values that were in $A[j\text{-}1, \ldots, n]$ at the time that the loop started. Decrementing j for the next iteration maintains the invariant.

Termination: The loop terminates when j reaches i. By the statement of the loop invariant, $A[i] = \min\{A[k]: i \le k \le n\}$ and $A[i, \ldots, n]$ is a permutation of the values that were in $A[i, \ldots, n]$ at the time that the loop started.

d) The running time depends on the number of iterations of the for loop of lines 2–4. For a given value of i, this loop makes n-i iterations, and i takes on the values 1, 2, …, n-1. The total number of iterations is,

$$\sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

$$= \mathrm{n(n-1)} - \frac{\mathrm{n(n-1)}}{2}$$

$$= \frac{\mathrm{n(n-1)}}{2} = \frac{\mathrm{n}^2}{2} - \frac{\mathrm{n}}{2}$$

However, bubble sort also has best-case running time $\Theta(n^2)$ whereas insertion sort has best-case running time $\Theta(n)$ and worst-case running time $\Theta(n^2)$.

Problem 2-3

c) Initially, i = n. So, the upper bound of the summation is −1, so the sum evaluates to 0, which is the value of y. For preservation, suppose it is true for an i, then,

$$y = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1}\, x^k$$

$$= a_i + x \sum_{k=1}^{n-i} a_{k+i}\, x^{k-1}$$

$$= \sum_{k=0}^{n-i} a_{k+i}\, x^k$$

At termination, i = -1, so is summing up to n − 1, so executing the body of the loop a last time gets us the desired final result, $y = \sum_{k=0}^{n} a_k\, x^k$.

Problem 2-4

d) We'll call our algorithm Mod_Merge_Sort for Modified Merge Sort. In addition to sorting A, it will also keep track of the number of inversions. The algorithm works as follows. When we call Mod_Merge_Sort(A,p,q) it sorts A[p, ..., q] and returns the number of inversions amongst the elements of A[p..q], so left and right track the number of inversions of the form (i, j) where i and j are both in the same half of A. When Mod_Merge(A,p,q,r) is called, it returns the number of inversions of the form (i, j) where i is in the first half of the array and j is in the second half. Summing these up gives the total number of inversions in A. The runtime is the same as that of Merge-Sort because we only add an additional constant-time operation to some of the iterations of some of the loops. Since Merge is Θ(n logn), so is this algorithm.

Algorithm 1:

```
Mod_Merge_Sort(A, p, r)
      if p < r then
            q =  (p + r)/2
            left = Mod_Merge_Sort(A, p, q)
            right = Mod_Merge_Sort(A, q + 1, r)
            inv = Mod_Merge(A, p, q, r) + left + right
            return inv
      end if
      return 0
```

Algorithm 2:

```
Mod_Merge(A,p,q,r)
        n1 = q − p + 1
        n2 = r − q
        let L[1..n1] and R[1..n₂] be new arrays
        for i = 1 to n1
            L[i] = A[p + i − 1]
        for j = 1 to n2
            R[j] = A[q + j]
        i = 1
        j = 1
        for k = p to r
            if i > n1
                A[k] = R[j]
                j = j + 1
            else if j > n2
                A[k] = L[i]
                i = i + 1
            else if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
            else
                A[k] = R[j]
                j = j + 1
                inversions += n1 − i
        return inversions
```