ESO207A: Data Structures and Algorithms
Assignment 1 Solution

Yatharth Goswami

September 20, 2020

1. Problem 1

The problem demanded us to write pseudo code for counting the number of inversions present in an array using Divide and Conquer.

```
Algorithm 1: Count the number of inversions in an Array $A$
  Input: Array $A$ containing n integers
  Output: Number of inversions present in $A$
Function CountInversions(Array A, Length n):
    sorted = Empty array of size n
    ans = MergeSort($A$, sorted, 0, $n-1$)
    return ans
Function MergeSort(Array A, Temp_Array sorted, start l, end r):
    count = 0
    if $r > l$ then
        $mid = \left\lfloor \frac{l+r}{2} \right\rfloor$
        $count = count + MergeSort$(A, sorted, l, mid)
        $count = count + MergeSort$(A, sorted, mid+1, r)
        $count = count + Merge$(A, sorted, l, mid, r)
    return count
Function Merge(Array A, Temp_Array sorted, start l, partition_at mid, end r):
    $count = 0$, $i = l$, $j = mid + 1$, $k = l$
    while $i \leq mid$ AND $j \leq r$ do
        if A[i] <= A[j] then
            $sorted[k] = A[i]$
            $k = k + 1$
            $i = i + 1$
        else
            $sorted[k] = A[j]$
            $k = k + 1$
            $j = j + 1$
            $count = count + mid + 1 - i$
    while $i \leq mid$ do
        $sorted[k] = A[i]$
        $k = k + 1$
        $i = i + 1$
    while $j \leq r$ do
        $sorted[k] = A[j]$
        $k = k + 1$
        $j = j + 1$
    for $i = l$ to $r$ do
        $A[i] = sorted[i]$
    return count
```

2. Problem 2

The problem demanded us to prove the correctness of the algorithm used above using "Loop Invariance Technique".

Proof. We will prove the correctness of the above algorithm by first proving the correctness of the **Merge** function and then the correctness of **MergeSort** function.

Part (a) : Proving Correctness of Merge function

We will prove this using the technique of loop invariants. The Merge function is taking in a part of an array A starting at position l, ending at position r and also takes in a temporary array of same size as that of original array A. The part of array taken as input is already partitioned at index mid into two sorted arrays $A[l, mid]$ and $A[mid+1, r]$, where

$$mid = \left\lfloor \frac{l+r}{2} \right\rfloor$$

$$A[l, mid] -> \text{Slice of } A \text{ starting at } l \text{ and ending at } mid$$

$$A[mid+1, r] -> \text{Slice of } A \text{ starting at } mid+1 \text{ and ending at } r$$

The function returns the number of inversions present in the range $[l, r]$ given that the subarrays $A[l, mid]$ and $A[mid+1, r]$ are sorted. The function also merges these two sorted arrays and sorts the subarray $A[l, r]$ as a whole.

For proving these two properties of the merge function we introduce the following loop invariants :

i Number of inversions in subarray $A[l, mid] = 0$

ii Number of inversions in subarray $A[mid+1, j] = 0$

iii Variable $count$ contains the number of inversions present in the subarray $A[l, j-1]$

iv $sorted[l, k-1]$ is a permutation of $A[l, i-1] \cup A[mid+1, j-1]$

v $sorted[l, k-1]$ contains the smallest $(k-l)$ elements of the array $A[l, mid+1]$ and $A[mid+1, r]$ combined

vi $l \leq i \leq (mid+1) \leq j \leq r+1$, $k = i + j - mid - 1$

vii $sorted[l, k-1]$ is sorted

The first three invariants are needed to prove that merge returns the number of inversions present in the array $A[l, r]$ and the last four are the same invariants as told in the class to prove that merge sorts the subarray $A[l, r]$ as a whole.

Now we will prove all of the invariants by proving the invariants at three different points in the program at line 14 (before the start of the loop), and then by assuming that if they hold at line 16, and $i \leq mid \text{ and } j \leq r$ they will also hold at line 24 when the first loop ends. While loops that start at lines 25 and 29 are essentially same as the first while loop at line 15, so we will only focus on proving the invariance for the first while loop but we will talk about other loops in termination part of the loop invariant proof.

Initialization: At the initialization step $i = l,\ k = l\ and\ j = mid + 1$ so invariants number 2,4,5, and 7 are trivially true and invariant 1 is true since the array $A[l, mid]$ is already sorted and therefore contains no inversions and hence invariant 3 is also true. And plugging in these initial values of i, j, k leads to invariant number 6.

Maintenance: Now, we will assume that the invariants hold for a particular iteration k and we will show that it holds for $k+1$ as well. So, we will show each of them one by one.

- i Since, we are not changing any part of original array A in the loop from 15 to 24 so this invariant will remain true, since the subarray $A[l, mid]$ remains sorted.
- ii Similar to the first invariant since we are not changing any part of original array A in the loop from 15 to 24 and therefore $A[mid+1, j]$ being a subarray of $A[mid+1, r]$ is also sorted and hence the number of inversions will remain zero.
- iii Now, for proving this invariant we will have two cases based on which of the **if-else** part of the loop gets executed.
  - a **Case (a):** *IF* part of the loop gets executed, then we can see that j doesn't get incremented inside the loop, hence *count* still holds the number of inversions in the array $A[l, j-1]$.
  - b **Case (b):** *ELSE* part of the loop gets executed, from the above two invariants we can say that any invariant that exists in the array $A[l, j-1]$ only exists between elements of $A[l, mid]$ and $A[mid+1, j-1]$ respectively and not inside any of them since they have zero inversions inside them as proved in above invariants. Now if $A[i] > A[j]$ then we can say that since $A[i, mid]$ is sorted so $A[p] \geq A[i] > A[j]$ where $p \in [i, mid]$ and hence each of them will add to the inversion count, so $count = count + mid + 1 - i$ and hence the count now contains the number of inversions in $A[l, j]$ and also since j gets incremented here, so the invariance holds for next loop as well.
- iv Now, if the invariant is true at the start and we are adding either $A[i]$ or $A[j]$ and the loop invariant will hold.
- v If the array already contains the $k - l$ smallest elements among $A[l, mid+1]$ and $A[mid+1, r]$ and since $A[i]$ and $A[j]$ are the smallest elements left in the arrays $A[i, mid]$ and $A[j, r]$ respectively and adding the smaller one among them implies that the array *sorted* now contains the smallest $k + 1 - l$ elements, which makes it true for next iteration as well.
- vi Since, either i or j increases by 1 and k also increases by 1. This condition holds invariably.
- vii By 5 we can say, that $sorted[l, k-1] \leq A[i], A[j]$, hence adding the smaller one among them will not disturb the sorted array and now $sorted[l, k]$ will become sorted which makes the invariant true for next iteration.

Termination : On termination of the loop we would have either $i > mid\ or\ j > r$.

- i **Case (a):** If $i > mid$ therefore $i = mid + 1$, by invariant number 6 we have $k = j$ therefore by 3 we have the number of inversions present in array $A[l, j-1]$. Now since by 5 it also happens that we have seen the $k - l$ smallest elements and the elements remaining now are all sorted since $j \geq mid + 1$, so by looking at these elements we will add no more inversions since these are bigger than all the seen elements and appear at the end of the array in sorted order. And now we can insert these remaining elements in our *sorted* array and the array will remain sorted afterwards as well.

So, after completing the while loop on line 29 we would have inserted an additional $r+1-j$ elements and adding $k-l$ to this will lead to inserting $r+1-l$ elements in total into *sorted* array (since k was equal to r before the start of the loop) and it will remain sorted as discussed above. Also, since we are adding no more inversions after that point we can similarly say that we have counted all of the inversions present in $A[l,r]$ before entering loop at line 29. Now, we are putting these sorted elements in original array $A[l,r]$ in loop in line 33.

ii **Case (b):** If $j > r$ therefore $j = r+1$, now for the *sorted* being the sorted version of array $A[l,r]$ after the loop 25 ends we will use just the similar argument as stated above. While for *count* containing the number of inversions in the array $A[l,r]$ we can say that since $j = r+1$ therefore it holds the number of inversions in $A[l,r]$ by invariant 3, which is the total array.

Hence proved, that merge sorts the original array and returns the count of inversions present in it and hence it's correctness.

Part (b): Proving Correctness of MergeSort function

To prove the correctness of the algorithm, we need to prove correctness of the MergeSort function as well, which can be easily proved using induction.
Recursive Property: Sorts the Elements in $A[l,r]$ and returns the number of inversions present in $A[l,r]$.
Base Case: $n = 1$ or A contains a single element (which is trivially sorted and contains zero inversions.)
Inductive Hypothesis: Assume that MergeSort correctly sorts and counts the number of inversions for $n = 1,2,3...,k$ elements.
Inductive Step: Merge sort correctly sorts and finds inversion count for $k+1$ elements.

Proof. We prove this by establishing an observation.
Observation 1: Counting the number of inversions in an array is same as dividing the array into two parts and counting number of inversions present inside these two arrays individually and adding inversions present between these two arrays.

Proof. We can easily prove this observation by noticing that any element of an array can have a smaller element on the right side of it which can be either on the right side of the partition or not. If it is on the right side of the partition it adds to the number of inversions between the two partitioned arrays, otherwise it adds to inversions present inside the two arrays individually. Also, notice that the second type of inversion *i.e.* between the two arrays can also be counted by sorting both the arrays and then counting for it, since every element which is in the second array will still remain to the right of the element of the first array. □

Now first recursive call in the MergeSort function will find the number of inversions in array of size $(k+1)/2 \leq k$ which therefore by induction hypothesis implies that the MergeSort correctly finds the number of inversions in this array and correctly sorts this as well.
Second recursive call also works on array of size $(k+1)/2 \leq k$ which implies that the MergeSort correctly finds the number of inversions in this array and correctly sorts this as well.
In the third step since we have proved the correctness of Merge function above so, we will get the number of inversions present between the two arrays.
Hence by **Observation 1** we proved that this function returns the number of inversions present in the array of size $k+1$ and sorts this array as well. □

Thereby, we proved the correctness of both the functions used in the program and hence the correctness of the program as whole. □

Part (c): Time Complexity Analysis

Now we will show that time complexity of the above algorithm is $O(nlogn)$.

Proof. The above program for just one element takes constant time. When we have $n > 1$ elements, we can break the running time ($T(n)$) as follows:

- **Time taken to divide into subproblems:** In this, we are just computing the middle of the subarray, which takes constant time. Thus $T_1(n) = \Theta(1)$.
- **Time taken to solve recursively the two subproblems:** We recursively solve two subproblems, each of size $n/2$ which contributes $2T(n/2)$ to the running time.
- **Time taken to combine solutions from the two subproblems:** To see that the **Merge** function runs in $\Theta(n)$ time, note that in each loop every operation takes constant amount of time and at the end of the while loops $i = mid+1$ and $j = r+1$ since the condition of loops demand this in the end. So, i started with l and ended at $mid+1$ and j started at $mid+1$ and ended at $r+1$ and in each iteration of all the loops only one of them got incremented so in all time taken by three while loops is $\Theta(mid+1-l+r+1-mid-1) = \Theta(r+1-l)$ which is same as $\Theta(n)$. Also the last *for* loop consists of operations happening in constant time and iterates over the size of array so it also takes $\Theta(n)$ time. So, in all the running time of this **Merge** function turns out to be $T_2(n) = \Theta(n)$

When we add all these individual times we get the following function

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

Now, this is just the same as time complexity of Merge Sort algorithm and can be solved either using the master theorem or tree recursion method.

Using Master Theorem we can compare that in this case, $b = 2$ and $a = 2$ and $f(n) = \Theta(n)$ and therefore it satisfies the second case of masters theorem since $f(n)$ is $\Theta(n^{log_a b})$. Therefore the time complexity would be $\Theta(n^{log_a b} logn)$ which is $\Theta(nlogn)$, which is same as $O(nlogn)$. $\square$