ESO207A: Data Structures and Algorithms
Assignment 1

Sarthak Rout - 190772

17 September 2020

---

1 Task 1: Pseudo Code

Algorithm 1 O($nlog(n)$) Algorithm for Counting Inversions

```
sentinel ← ∞
dac_count ← 0 { Global Counter }
function MERGE(ar, x, y)
  size1 ← (x+y)/2 − x + 1 , size2 ← y − (x+y)/2 − 1 + 1
  v1 ← array containing elements of ar indexed from x to (x+y)/2
  v2 ← array containing elements of ar indexed from (x+y)/2 + 1 to y
  v1[last] ← sentinel , v2[last] ← sentinel
  iterator1 ← 0 , iterator2 ← 0, ar_index ← 0
while  iterator1 < size1 or iterator2 < size2 do
  if v1[iterator1] > v2[iterator2] then
      ar[x + ar_index] ← v2[iterator2]
      iterator2 ← iterator2 + 1
      ar_index ← ar_index + 1
      dac_count ← dac_count + by size1 - iterator1
      continue
  else
      ar[x + ar_index] ← v1[iterator1]
      iterator1 ← iterator1 + 1
      ar_index ← ar_index + 1
      continue
  end if
end while
return
end function
function MERGE_SORT(ar, a, b)
if  a ≥ b then
  return
end if
  MERGE_SORT(ar, a, (a+b)/2)
  MERGE_SORT(ar, (a+b)/2 + 1, b)
  MERGE(ar, a, b)
return
end function
function COUNTINVERSIONS(ar, n)
  MERGE_SORT(ar, 0, n − 1)
return  DAC_COUNT
end function
COUNTINVERSIONS(ar, n)
return  =0
```

2 Task 2: Loop Invariant and Correctness of the Algorithm

Most of the algorithm is in Merge function. So, it's loop invariant are discussed first.

1. **$0 \leq$ iterator1 $\leq$ size1 , $0 \leq$ iterator2 $\leq$ size2 and $0 \leq$ ar_index $\leq$ size1 + size2**
   Initially, all of these variables are zero and v1[size1] and v2[size2] $= \infty$. Before that, at any iteration step : both iterator1 $\leq$ size1 and iterator2 $\leq$ size2 . If they reach the value ∞ , then they don't get pushed into the original array because they larger than any value and we are choosing the smaller value.
   If both of them reach the value of size1 and size2, such that v[iterator1] $= \infty$ and v[iterator2] $= \infty$ the loop is terminated. Due to the first two inequalities, as ar_index is incremented whenever iterator1 or iterator2 is incremented, so it follows.
   So, this loop invariant is satisfied.

2. **ar[x...x + ar_index - 1] is a permutation of v1[0...iterator1 -1] $\cup$ v2[0...iterator2-1]** Initially, this is trivially true as there are no elements. At step k: Assume the statement that ar[x...x + ar_index - 1] is a permutation is true. Either anyone element from v1 or v2 is pushed and the corresponding iterator incremented along with ar_index.
   If v1[iterator1] is smaller , iterator1 is incremented and ar_index is incremented or vice-versa. In both of the cases, this property doesn't break.

3. **ar[x...x + ar_index - 1] is sorted**
   Initially, there is no elements in the range ar[x...x + ar_index - 1]. At any step : assume ar[x...x + ar_index - 2] is sorted. Also, v1 and v2 are sorted as they are returned from *Merge_Sort* function, which returns sorted arrays as per above assumption after termination.
   So, by above invariant (2) , v1[iterator1 - 1] and v1[iterator2 -1] are both bigger than all others elements that are present in ar[x...x + ar_index - 2], so, next element that will be pushed will definitely maintain the non-decreasing order.
   So, we can conclude that this loop invariant always holds true.

4. In *Merge_Sort* call, **Count of number of *inter-inversions* (described next) remains constant**. In other words, for two disjoint subarrays of ar , ar[a...b] and ar[c...d], count of number of inversions such that both elements are not of same subarray is constant even if both these arrays are permutated simultaneously.
   Consider i and j, two indices belonging to both of these subarrays such that a $\leq$ i $\leq$ b, and c $\leq$ j $\leq$ d and ar[i] $>$ ar[j] and hence an inversion. After permutation, let i $\rightarrow$ i' and j $\rightarrow$ j'. Even now, i' $\leq$ b $<$ c $\leq$ j' $\Rightarrow$ i' $<$ j' and ar[i'] $>$ ar[j'] as before.
   So, it is still an inversion after permutation and this invariant is satisfied.

5. Before and after *Merge* call, **The sum of number of inversions of the array at any time and *dac_count* is same as number of inversions present in the initial array**.
   Initially, it is true as *dac_count* is 0. At any step: when v1[iterator1] $>$ v2[iterator2], v2[iterator2] is pushed to the array. But, along with iterator1 , all the elements after it in v1 are also greater than v1[iterator1].
   We have reduced the number of inversions in the array by $(size1 - 1) - (iterator1 - 1) = size1 - iterator1$. $size1 - 1$ as iterator is zero-based index (obviously, we are not including the sentinel value) and $iterator - 1$ as, we are including this index too. So, we must increase value of *dac_count* by this much as in the step in the algorithm. At the end, the array is sorted, so, number of inversions now is zero and *dac_count* stores the total number of inversions. So, this loop invariant is correct and algorithm calculates the number of inversions correctly.