COMP SCI 577 Homework 01

Divide and Conquer

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

20 September 2022

3 [Graded]

Algorithm

Explanation

With an input of array of integers Arr, we want to calculate the inversion number of the sorted tree Inv_M with the swapping property, the sorted array $Sorted_M$ without the swapping property, and the tree itself $Tree_M$. We first break Arr which is the original tree, or array to sort from the middle into two identical pieces Arr_L and Arr_R. Then we recursively call the function itself on the separated pieces to get the Inv, $Sorted$, and $Tree$ values for both sides (L and R). After that, we first call `Merge` to merge $Sorted_L$ and $Sorted_R$ to form $Sorted_M$, as well as counting the cross inversion number $CrossInv_{L<R}$ if there is no swap between the left and right subtrees. Then we call `Merge` to merge it in a reverse direction to see if the cross inversion number decreases for $CrossInv_{R<L}$ if the two sides are swapped; the $Sorted$ returned is not used and ignored. If $CrossInv_{R<L}$ is smaller, then we should swap the two subtrees so that $Tree_M$ is formed by the original right part at left, and the original left part at right; otherwise, $Tree_M$ should have the original left part at left, and the original right part at right, as unmodified. Finally, we return $Inv_M = Inv_L + Inv_R + \min(CrossInv_{L<R}, CrossInv_{R<L})$, $Sorted_M$, and $Tree_M$.

The subroutine `Merge`$(Sorted_L, Sorted_R)$ is the combination of the functions `Count-Cross` and `Merge` introduced in the lecture on September 13 [1] without modification compared to the function used in counting inversions with two sorted sub-arrays.

Pseudo Code

Function `MergeSort`(Arr)**:**
  $Mid \leftarrow \texttt{Size}(Arr)/2$;
  $Arr_L \leftarrow Arr[0 : Mid)$;
  $Arr_R \leftarrow Arr[Mid : \texttt{Size}(Arr))$;
  $Inv_L, Sorted_L, Tree_L \leftarrow \texttt{MergeSort}(Arr_L)$;
  $Inv_R, Sorted_R, Tree_R \leftarrow \texttt{MergeSort}(Arr_R)$;
  $CrossInv_{L<R}, Sorted_M \leftarrow \texttt{Merge}(Sorted_L, Sorted_R)$;
  $CrossInv_{R<L}$, ~~$None$~~ $\leftarrow \texttt{Merge}(Sorted_R, Sorted_L)$;
  if $CrossInv_{L<R} < CrossInv_{R<L}$ **then**
    $Tree_M \leftarrow [Arr_L \dots Arr_R]$;
  else
    $Tree_M \leftarrow [Arr_R \dots Arr_L]$;
  $Inv_M \leftarrow Inv_L + Inv_R + \min(CrossInv_{L<R}, CrossInv_{R<L})$;
  return $Inv_M, Sorted_M, Tree_M$;

Function `Merge` $(Sorted_L, Sorted_R)$**:**
  $CrossInv, Index_L, Index_R \leftarrow 0$;
  $Sorted_M \leftarrow []$;
  while $Index_L < \texttt{Size}(Sorted_L)$ **and** $Index_R < \texttt{Size}(Sorted_R)$ **do**
    if $Sorted_L[Index_L] > Sorted_R[Index_R]$ **then**
      $CrossInv \leftarrow CrossInv + (\texttt{Size}(Sorted_L) - Index_L)$;
      $Sorted_M \leftarrow [Sorted_M \dots Sorted_R[Index_R]]$;
      $Index_R \leftarrow Index_R + 1$;
    else
      $Sorted_M \leftarrow [Sorted_M \dots Sorted_L[Index_L]]$;
      $Index_L \leftarrow Index_L + 1$;
  while $Index_L < \texttt{Size}(Sorted_L)$ **do**
    $Sorted_M \leftarrow [Sorted_M \dots Sorted_L[Index_L]]$;
    $Index_L \leftarrow Index_L + 1$;
  while $Index_R < \texttt{Size}(Sorted_R)$ **do**
    $Sorted_M \leftarrow [Sorted_M \dots Sorted_R[Index_R]]$;
    $Index_R \leftarrow Index_R + 1$;
  return $CrossInv, Sorted_M$;

Code (Python)

```python
from typing import List, Tuple

def merge(left_sorted: List[int], right_sorted: List[int]):
  left_ptr: int = 0
  right_ptr: int = 0
  cross_inv: int = 0
  sorted: List[int] = list()
  while left_ptr < len(left_sorted) and right_ptr < len(
     right_sorted):
    if left_sorted[left_ptr] > right_sorted[right_ptr]:
      cross_inv += len(left_sorted) - left_ptr
      sorted.append(right_sorted[right_ptr])
      right_ptr += 1
    else:
      sorted.append(left_sorted[left_ptr])
      left_ptr += 1
  while left_ptr < len(left_sorted):
    sorted.append(left_sorted[left_ptr])
    left_ptr += 1
  while right_ptr < len(right_sorted):
    sorted.append(right_sorted[right_ptr])
    right_ptr += 1
  return cross_inv, sorted

def mergesort(to_sort: List[int]) -> Tuple[int, List[int], List[
   int]]:
  n: int = len(to_sort)
  if n <= 1:
    return 0, to_sort, to_sort
  mid: int = int(n / 2)
  left_arr: List[int] = to_sort[0: mid]
  right_arr: List[int] = to_sort[mid: n]
  left_inv, left_sorted, left_tree = mergesort(left_arr)
  right_inv, right_sorted, right_tree = mergesort(right_arr)
```



```python
  cross_inv_lr, sorted = merge(left_sorted, right_sorted)
  cross_inv_rl, _ = merge(right_sorted, left_sorted)
  if cross_inv_lr < cross_inv_rl:
    tree = left_tree + right_tree
  else:
    tree = right_tree + left_tree
  inv = left_inv + right_inv + min(cross_inv_lr, cross_inv_rl)
  return inv, sorted, tree

if __name__ == "__main__":
  A: List[int] = list(int(x) for x in input().split(" "))
  inv, sorted, tree = mergesort(A)
  print("inv:", inv)
  print("sorted:", sorted)
  print("tree:", tree)
```

Test Cases

Input: $[4,2,1,3]$
Output: $Inv \leftarrow 1$; $Sorted \leftarrow [1,2,3,4]$; $Tree \leftarrow [1,3,2,4]$

Input: $[1,4,2,8,5,7,3,6]$
Output: $Inv \leftarrow 7$; $Sorted \leftarrow [1,2,3,4,5,6,7,8]$; $Tree \leftarrow [1,4,2,8,3,6,5,7]$

Correctness

Induction

Claim: The algorithm is correct for any array with a size of 2^n with $n \in \mathbb{N}$.
Base Case: $n = 0$. As an array with a size of 1 is already sorted without any other integer to compare, the inversion number Inv_M is 0, and the sorted array and the tree array $Sorted_M$ and $Tree_M$ are kept unmodified. Thus, the algorithm is correct for any array with a size of 1.
Inductive Step: Assume that the algorithm is correct for $n = k-1$. For $n = k$, what we can swap is only the whole left part and the whole right part which are both with a size of 2^{k-1}. To argue the correctness of the merged tree $Tree_M$ and the inversion number Inv_M, we should

arrange $Tree_L$ and $Tree_R$ in a way so that the merged $[Tree_L \dots Tree_R]$ or $[Tree_R \dots Tree_L]$ has the minimal inversion number Inv_M which is counted by $Inv_L + Inv_R + \min(CrossInv_{L<R}, CrossInv_{R<L})$. The correctness of $\texttt{Merge}(Sorted_L, Sorted_R)$ yielding $CrossInv_M$ and $Sorted_M$ has been proved in the lecture [1]. This counting technique could still apply, as the sorted properties of $Sorted_L$ and $Sorted_R$ are kept, as well as that a parent node could only switch its two subtrees, instead of modifying the nodes in the subtrees. Then we only need to compare the inversion numbers in two situations: the original order of $Tree_L$ and $Tree_R$, and the reversed order of $Tree_L$ and $Tree_R$, which should only rely on $CrossInv_{L<R}$ and $CrossInv_{R<L}$ yielded in these two situations (as Inv_L and Inv_R are immutable). As we want the minimal inversion number, we should only swap the two subtrees if $CrossInv_{R<L}$ is smaller than $CrossInv_{L<R}$. Thus, the algorithm is correct for any array with a size of 2^k.

Termination

The size of 2^n of the array is halved in each recursion, namely that we deduct the power by 1 every time, for all $n > 0$. Then the size of the array should always converge to $2^0 = 1$, and the base case will be reached. Thus, the algorithm terminates after $\log 2^n = n$ recursions.

Complexity

For the subroutine $\texttt{Merge}(Sorted_L, Sorted_R)$, its complexity introduced in the lecture [1] is $O(n)$ with $n = \texttt{Size}(L) + \texttt{Size}(R)$, which is also stated in Theorem 2 and Theorem 5 of [2].

For the main function $\texttt{MergeSort}(Arr)$, according to Master Theorem, the recurrence is of the form $T(n) = 2T(\frac{n}{2}) + f(n)$ with $f(n) = O(n)$, as there is there are two recursive calls for Arr_L and Arr_R respectively, as well as one $O(n)$ call to $\texttt{Merge}$. Then we can see that the function's recurrence is the same as the original Merge Sort introduced in the lecture [1] and Theorem 3 of [2], whose run time complexity is $O(n \log n)$.

References

[1] Dieter van Melkebeek (2022) *COMP SCI 577 Lecture Note*, 13 September 2022, University of Wisconsin-Madison. https://canvas.wisc.edu/courses/308877/files/27719231.

[2] Deeparnab Chakrabarty (2020) *CS31 (Algorithms), Spring 2020 : Lecture 3*, Dartmouth College. https://www.cs.dartmouth.edu/~deepc/Courses/S20/lecs/lec3.pdf.