CSIE5432 Mini Homework 2

Ching-Chung, Huang
r09921092

October 2020

1 Introduction

We are asked to count all reverse pairs in a given array which consists of ordinal objects that may or may not be numbers in general. Besides, objects may not be all distinct. In this paper, two methods are given. First, we propose a naïve method, which takes $\Theta(n^2)$ time for all inputs, where n is the length of the given array. Next, we develop a $O(n \log(n))$ method using the divide-and-conquer technique.

2 Naïve Method

2.1 Main Idea

Traverse the given array and find all smaller elements from its right. Count the number of all such elements and return the final value as we are done traversing the array. Two loops are needed. The first loop is responsible for traversing the array, while the second loop finds all the smaller elements on the right of a particular array element.

2.2 Pseudocode

Algorithm 1: BruteForce(A)

```
1:  n ← len(A)
2:  count ← 0
3:  for i = 1 to n − 1 do
4:      for j = i + 1 to n do
5:          if A[i] > A[j] then
6:              count++
7:          end if
8:      end for
9:  end for
10: return count
```

2.3 Analysis

The main cost of the BruteForce(A) algorithm is the nested loop from line 3 through 9. The **for** loop on line 3 operates $n-1$ times, and on the i-th looping, it activates another **for** loop, which takes $n-i$ steps. Thus, the total cost of the nested loop is $(n-1)+(n-2)+\cdots+1 = n(n-1)/2 \in \Theta(n^2)$. Notice that the number of looping is independent of the pattern of inputs. Therefore, the BruteForce(A) algorithm takes $\Theta(n^2)$ time for all kinds of inputs.

2.4 Implementation Using Python

```
1 def BruteForce(A):
2     n = len(A)
3     count = 0
4     for i in range(n-1):
5         for j in range(i + 1, n):
6             if (A[i] > A[j]):
7                 count += 1
8     return count
```

Listing 1: BruteForce(A)

3 Divide and Conquer

3.1 Main Idea

When sorting an array using the MergeSort algorithm, we spilt the array equally into two parts, say the left part and the right part. Next, sort each part recursively. Finally, merge the two sorted subarrays and derive the whole sorted array. Observe that if an element l in the left sorted subarray is larger than some element in the right sorted array, then a reverse pair (l, r) forms. From this, we see that while merging, r would be merged before l. Using this property, we can count the number of all reverse pair. This divide-and-conquer algorithm is a modified version of the MergeSort algorithm. Hence, it'd be no surprising that its time complexity is $O(n \log(n))$ in the worst case, where n is the length of the given array.

3.2 Pseudocode

To begin with, we declare a global variable r to count the number of reverse pairs. Since r is global, it will be updated by all the functions accessing to it. We merge sort the array using the MergeSort(X) algorithm. It is a recursive function that will call the Merge(A, B) function, which is an iterative subroutine for merging. The global variable r will be updated during the process of merging if necessary, either on the line 7 or 18 in the Merge(A, B) algorithm. As soon as the MergeSort(X) is complete, r is then the total number of reverse pair. Return this value and we are done counting. The declaration of r, the execution of MergeSort(X), and the return of r are all written in the main function Count(X).

Algorithm 2: Merge(A, B)

```
1:  Declare a global variable r
2:  C ← empty list
3:  i ← 0
4:  j ← 0
5:  while i ≠ len(A) and j ≠ len(B) do
6:     if A[i] ≤ B[j] then
7:        r ← r + len(C) − i
8:        C.append(A[i])
9:        i++
10:    else
11:       C.append(B[j])
12:       j++
13:    end if
14:    if j < len(B) then
15:       C ← C + B[j :]
16:    end if
17:    if i < len(A) then
18:       r ← r + (len(A) − i)(len(C) − i)
19:       C ← C + A[i :]
20:    end if
21:    return C
22: end while
```

Algorithm 3: MergeSort(X)

```
1: if len(X) = 1 then
2:    return X
3: else
4:    m ← ⌊len(X)/2⌋
5:    A ← MSort(X[: m])
6:    B ← MSort(X[m :])
7:    C ← Merge(A, B)
8:    return C
9: end if
```

Algorithm 4: Count(X)

```
1: Declare a global variable r
2: MergeSort(X)
3: return r
```

3.3 Analysis

First, let's analyze the time complexity of Merge(A, B). Suppose the total length of A and B is n. Then, thanks to line 9 and line 12, the **while** loop activates at most n times. Every operation in the **while** loop takes $O(1)$ time. Thus, Merge(A, B) takes $O(n)$ time. Next, we claim MergeSort(X) has time complexity $\Theta(n \log(n))$, where n is the length of the array X. The recursive steps are line 5 and 6, which takes $T(\frac{n}{2})$ since we are splitting X equally. In addition, on line 7 we see the Merge function, which takes $O(n)$ time to execute. Therefore, $T(n) = 2T(\frac{n}{2}) + O(n)$. From the master theorem (case 2), since $n^{\log_2(2)} = n = n \log^0(n)$, $T(n) = \Theta(n \log(n))$. Finally, the time complexity of Count(X) is of the same asymptotic order as MergeSort(X). Thus, we have shown Count(X) takes $O(nlog(n))$ time.

3.4 Implementation Using Python

```python
def Merge(A, B):
    global r
    C = list()
    i = 0
    j = 0
    while (i != len(A)  and j != len(B)):
        if A[i] <=  B[j]:
            r += (len(C) - i)
            C.append(A[i])
            i += 1

        else:
            C.append(B[j])
            j += 1
    if j < len(B):
        C += B[j:]
    if i < len(A):
        r += (len(A) - i) * (len(C) - i)
        C += A[i:]
    return C
```

Listing 2: Merge(A, B)

```python
def MergeSort(X):
    global r
    if len(X) == 1:
        return X
    else:
        m = len(X)//2
        A = MSort(X[:m])
        B = MSort(X[m:])
        C = Merge(A, B)
        return C
```

Listing 3: MergeSort(X)

```python
def Count(X):
  global r
  MergeSort(X)
  return r
```

Listing 4: Count(X)

Conclusion

The reverse pair problem can be solved in $O(n \log(n))$ time using the divide-and-conquer technique, which is quite promising. However, there might exist a more clever way to solve this problem with a lower time complexity.