东南大学
网络空间安全学院

算法作业 #1

学生姓名: 李盛; 学号: *229221*

Course: 算法设计与分析 – Professor: 陶军
Due date: *3* 月 *27* 日, *2022* 年

第一题

1. 设 X[0:n-1] 和 Y[0:n-1] 为两个数组, 每个数组中含有 n 个已排好序的数。试设计一个 $O(logn)$ 时间的分治算法, 找出 X 和 Y 的 $2n$ 个数的中位数, 并证明算法的时间复杂性为 $O(logn)$.

Solution.

(a) Similar to binary search, this method works by first calculating medians of the two sorted arrays and then comparing them.

- Calculate the medians $X[n/2]$ and $Y[n/2]$ of the input arrays X[ ] and Y[ ] respectively.
- If $X[n/2] == Y[n/2]$, return $X[n/2]$.
- If $X[n/2] < Y[n/2]$, then median is present in the set of right half of X[ ]and left half of Y[ ].
- If $X[n/2] > Y[n/2]$, then median is present in the set of left half of X[ ]and right half of Y[ ].
- Repeat the above process unti we reach the base case $n == 1$ or $n == 2$.

Without loss of generality, pseudocode is shown as Algorithm 1, assuming n is a power of 2.

Algorithm 1 Median(X,Y,n)

if $n == 1$ **then**
  return $\frac{X[1]+Y[1]}{2}$
end if
if $n == 2$ **then**
  return $\frac{max(X[1],Y[1])+min(X[2]+Y[2])}{2}$
end if
if $X[n/2] == Y[n/2]$ **then**
  return $X[n/2]$
else if $X[n/2] < Y[n/2]$ **then**
  return $Median(X[n/2+1 \ldots n], Y[1 \ldots n/2], n/2)$
else
  return $Median(X[1 \ldots n/2], n/2], Y[n/2+1 \ldots n], n/2)$
end if

(*b*) Time complexity analysis
The running time of comparing the medians of two sorted arrays is $O(1)$. The overall running time is therefore $T(n) = T(n/2) + O(1)$. According to Master Method, we have $a = 1$, $b = 2$ and $k = 0$, $T(n) = O(n^{log_b a} log^{k+1} n) = O(logn)$.

(*c*) Implementation code with Python.

```python
class Solution(object):
    def Median_arrays(self, X, Y, n):
        if n == 1:
            return (X[0]+Y[0])/2
        elif n == 2:
            return (max(X[0],Y[0])+min(X[1],Y[1]))/2
        else:
            m1 = self.median(X,n)
            m2 = self.median(Y,n)
            if m1 == m2:
                return m1

            elif m1 > m2:
                if n % 2 == 0:
                    return self.Median_arrays(X[: n//2],
                                              Y[n//2: ], n//2)
                else:
                    return self.Median_arrays(X[: n//2+1],
                                              Y[n//2: ], n//2+1)
            else:
                if n % 2 == 0:
                    return self.Median_arrays(X[n//2: ],
                                              Y[: n//2], n//2)
                else:
                    return self.Median_arrays(X[n//2: ],
                                              Y[: n//2+1], n//2+1)

    def median(self, arr, n):
        if n % 2 == 0:
            return (arr[n//2] + arr[n//2-1])/2
        else:
            return arr[n//2]
```

第二题

有一实数序列 $a_1, a_2, \ldots, a_N$, 若 $i < j$ 且 $a_i > a_j$, 则 (a_i, a_j) 构成了一个逆序对, 请使用分治方法求整个序列中逆序对个数，并分析算法的时间复杂性。例如: 序列 $(4, 3, 2)$ 逆序对有 $(4, 3), (4, 2), (3, 2)$ 共 3 个

Solution.

(*a*) Problem 2 could be solved in a similar way with merge sort.

- Divide: separate list into two halves A and B.
- Conquer: recursively count inversions in each list.
- Combine: count inversions (a, b) with a ∈ A and b ∈ B.
- Return sum of three counts.

Count-Inversions pseudocode is shown as Algorithm 2.

Algorithm 2 Count-Inversions(A, l, r)

if $l < r$ **then**
  $mid = \left\lfloor \frac{L+r}{2} \right\rfloor$
  $left = Count - Inversions(A, l, mid)$
  $Right = Count - Inversions(A, mid, r)$
  $inv = Merge - Sort - Count(A, l, mid, r) + left + right$
end if
return inv

(*b*) The combine step, i.e. count inversions (a, b) with a ∈ A and b ∈ B, assuming A and B are sorted could be summarized as follows.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i \leq b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append the left elements to the sorted list.

Merge-Sort-Count pseudocode is shown as Algorithm 3.

Algorithm 3 Merge-Sort-Count(A, l, mid, r)

```
inv = 0
n_1 = mid − l + 1
n_2 = r − mid
Let L[1...n_1] and R[1...n_2] be new arrays
for i = 1...n_1 do
    L[i] = A[l + i − 1]
end for
for j = 1...n_2 do
    R[j] = A[mid + j]
end for
i, j, k = 1, 1, l
while i ≠ n_1 + 1 and j ≠ n_2 + 1 do
    if L[i] ≤ R[j] then
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
        inv = inv + n_1 − i + 1        ▷ R[j] is inverted with elements in L[i...n_1]
    end if
    k = k + 1
end while
if i == n_1 + 1 then
    for m = j...n_2 do
        A[k] = R[m]
        k = k + 1
    end for
end if
if j == n_2 + 1 then
    for m = i...n_1 do
        A[k] = L[m]
        k = k + 1
    end for
end if
return inv
```

(*c*) Time complexity analysis
The running time 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 Sort is $\Theta(nlogn)$, so is this algorithm.

(*d*) Implementation code with Python.

```python
class Solution(object):
    def Count_Inversions(self, A, l, r):
        if l < r:
            mid = (l + r)//2
            left = self.Count_Inversions(A, l, mid)
```

```
            right = self.Count_Inversions(A, mid+1, r)
            inv = self.Merge_Sort_Count(A, l, mid, r)
                                           + left +right
            return inv
        return 0

    def Merge_Sort_Count(self, A, l, mid, r):
        inv = 0
        L = A[l:mid+1]
        R = A[mid+1:r+1]
        i,j,k = 0,0,l
        while i != mid-l+1 and j != r-mid:
            if L[i] <= R[j]:
                A[k] = L[i]
                i += 1
            else:
                A[k] = R[j]
                j += 1
                inv += mid-l-i+1
            k += 1
        if i == mid - l +1:
            for m in range(j, r-mid):
                A[k] = R[m]
                k += 1
        if j == r-mid:
            for m in range(i, mid-l+1):
                A[k] = L[m]
                k += 1
        return inv
```