Assignment 1

Ritwik Sahani - EE18BTECH11038

Github repository

https://github.com/ritvix23/C−DataStructures/tree/master/Assignment1

1 Problem

There are n unsorted arrays: A1 , A2 , . . . , An . Assume that n is odd. Each of A1 , A2 , . . . , An contains n distinct elements. There are no common elements between any two arrays. The worst-case time complexity of computing the median of the medians of A1 , A2 , . . . , An is -

1) $O(n)$
2) $O(nlogn)$
3) $O(n^2)$
4) $\Omega(n^2logn)$

2 Solution

2.1 Vagueness in question

The question does not specify any algorithm for which the worst case is to be analyzed. Therefore, let us take two different algorithms, first brute force approach and then a comparatively efficient approach, and analyze worst cases for each one of them.

2.2 Brute Force Approach

Brute force approach involves sorting the arrays. After sorting, the task of finding the median of medians becomes trivial. The exact steps are -

- Sort each array - $O(nlogn)$ for each array, $O(n^2logn)$ on the whole.
- Find the median of each sorted array - the element at index $(n-1)/2$ (0-indexed) - for this left multiply the matrix with the following $1 \times n$ matrix (that has one at $(n+1)/2$ and all other entries are zero) -
$$\begin{bmatrix} 0 & 0 & . & . & 1 & . & . & 0 & 0 \end{bmatrix}$$
- This gives another $1 \times n$ matrix that contains the medians of the individual rows.
- Sort this new matrix - $O(nlogn)$.
- Find the median of this new matrix by right mutiplying the matrix with the following $n \times 1$ matrix (that has one at $(n+1)/2$ and all other entries are zero) -
$$\begin{bmatrix} 0 \\ 0 \\ . \\ . \\ 1 \\ . \\ . \\ 0 \\ 0 \end{bmatrix}$$
- The resulting 1×1 matrix has just one entry which is the median of medians - $O(1)$.

Let us now analyze the time complexity of this approach -

$$T(n) = O(n^2logn + n + nlogn + 1) \implies O(n^2logn)$$

2.3 Efficient Approach

It should be intuitively clear that we are doing a lot of redundant work in sorting the arrays, while all we need is the middle element after sorting. Let us now look at an efficient approach to find this middle element without the need to sort the entire array. For a while, let us assume we have a routine - $qselect(k, array)$, that returns k^{th} element, after sorting, in $array$ in linear time ($O(n)$). Now, consider the following algorithm -

- Use this routine on each array to find $n/2$ index element - basically the median of each array - $O(n)$ for each array, $O(n^2)$ on the whole.
- Store these n medians in a separate array - $O(n)$
- Use the routine again on this new array to get median - $O(n)$.

Let us now analyze the time complexity for this algorithm -

$$T(n) = O(n^2 + n + n + 1) \implies O(n^2)$$

2.4 *Uncovering qselect(..,..)*

Let us see what we already know about this routine -

- Parameters - number k, array arr (possibly unsorted)
- Returns - k^{th} index element in arr after sorting.
- Should do the required job in linear time.

The starting point to design such an algorithm is the following idea -

Find which part of the array the required element might lie in, then truncate and recurse.

The exact algorithm consists of following steps -

- Choose a pivot element from the array (choose uniformly at random).
- Partition the array so that all the elements less than the pivot element would lie on the left of the pivot element and all the elements greater than the pivot would lie on its right side after the partitioning - this can be done in O(n) by comparing and swapping other elements with pivot element.
- After the prev step, the pivot would be exactly at the same index as it would if the array was sorted - call this index i_P.
- Now if $i_P = k \implies$ the element we are looking for is the pivot element, we are done, otherwise if $i_P > k \implies$ lies in the sub-array on the left to the pivot, in any other case it lies in the right sub-array.
- Recurse on the appropriate sub-array determined from the prev step.

Pseudo code for qselect is as follows -

Algorithm 1 QSELECT(A[0 .. n-1], k)

```
if n is 1 then
    return A[0]
else
    choose a pivot element A[p]
    i_P ← PARTITION(A[0..n − 1], p)
    if k < i_P then
        return qselect(A[0..i_p-1], k)
    else if k > r then
        return qselect(A[i_P+1..n-1], k - i_p)
    else
        return A[i_P]
    end if
end if
```

Algorithm 2 PARTITION(A[0 .. n-1], k)

```
swap A[p] ⟷ A[n]
l ← 0
for
    i ←0 to n-2 do
    if A[i] < A[n] then
        l ← l + 1
        swap A[l] ⟷ A[i]
    end if
end for
swap A[n] ⟷ A[l + 1]
return l + 1
```

Theoretical time complexity can be calculated from the recursive relation -

$T(N) = T(N/2) + O(N)$

$\implies T(N) = O(N)$

the next section verifies the theoretical results.

3 Verification

To verify the theoretical results, I measured the execution times of both the algorithms and plotted them with the theoretical bounds -

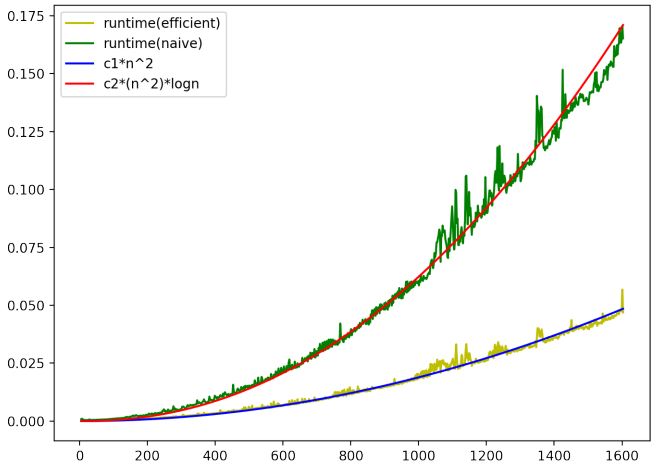

Fig. 1: Execution Times

We can clearly observe that the execution times align with the theoretical findings.

This plot can be generated through the following python script

https://github.com/ritvix23/C−DataStructures/blob/master/Assignment1/codes/timer.py