Algorithms and Data Structures

Drishti Maharjan

March 15, 2019

Assignment 5

Problem 5.1

a) Code in lomoto.c

b) Code in hoare.c

c) Code in median.c

d)*note: Online compiler was used to generate time values to avoid biased values due to PC activities. Also, running loop 100,000 for a program at once caused issues for copying or reading into large files. So, the same program was run lesser number of times in one sitting and then repeated, making 100,000 subsequences in total for fixed length 1000. Firstly, median.c was run as it generated random numbers and saved it in a file. Then, in the other programs, those arrays were read and quicksort was applied for the sequences, each of length 1000. Due to file size limitations, I am not uploading all the files. Please also note that when the loops were divided into different sittings, sometimes the time computations were not as expected, so it was run multiple number of times for average values, and the final results are listed below(which definitely satisfies the expectations of the values). But, I have attached only the first 2000 sequences of randomly generated array of size 1000 (i.e. altogether, 20,000 values). Filename: array.txt*

After averaging the values, for an array size of 1000 for the same array in 3 methods per loop, the average time computations are listed below:

Time for lomoto = 0.000158

Time for hoare = 0.000133

Time for median = 0.000129

For random permutations, there might not be much difference on efficiency of 3 algorithms but, the efficiency can be significantly noted especially when array is sorted. We can see that, the Median method takes least amount of time, and Lomoto takes the max time. This is because of the underlying mechanism on how these algorithms traverse through the array, as explained below:

1 **Lomoto** : It takes the pivot as the highest or lowest element of the array(depends on how it is implemented), so it has to traverse the array for all elements to put pivot in right position. This method uses (n-1) comparisons to partition an array of length n. Lets say we have an array [1..n] and the index variable j traverses through the array and increments 'i' upon finding an element A[j] <= pivot x. So, there are (x-1) smaller elements than pivot x, means no of swaps = (x-1) for pivot x.

2 **Hoare** : The indices i and j traverse in opposite direction, or let's say towards each other until they cross paths, which will take place when they are at position x which is our pivot. Visualizing it, it would be something like dividing the array into 2 parts, one scanned by i and other by j. And, swap inside the loop swaps every pair of elements that are in the wrong partition. For sorted loops, Hoare does no swaps for wrong pairs as the elements are in the correct partition side, however for Lomuto, roughly n/2 swaps will take place. And, even for equal values, everytime a swap will occur for Lomuto. But, for Hoare, even though the pair is swapped, i and j will always meet in the middle, and efficient partitoning takes place as compared to Lomoto.

3 **Median of 3** : Even though, Hoare performed relatively better, it still performed bad for sorted cases (worst case). Median of 3 method, indtead reduces the likelihood of choosing a bad pivot. In my implementation, I have used median as the pivot value and then applied Hoare, to get the most optimal version. Median of 3 performs better for sorted arrays, as it doesn't do unnecessary swaps, and the element which is supposed to be placed at the middle(not just in term of index, but also in term of value) will be chosen as pivot and placed in the middle. It's more like choosing the true Median when we don't have a clue about how the input array is ordered. Hoare chooses middle element as pivot, irrespective of its real position, so if the pivot is the smallest/largest element, the algorithm is not efficient. This is improved in Median of 3, as smallest or largest value will be never taken as pivot. So, median of 3 performs better than the other two methods.

For smaller arrays and unfiormly distributed samples, the difference might not be visible, but for larger arrays with random sequences, having probability of worst cases, time complexities of the algorithms:
Median of three < Hoare < Lomoto

Reference: https://cs.stackexchange.com/questions/11458/quicksort-partitioning-hoare-vs-lomuto/103405

Problem 5.2

a) Code in 2pivot.c

b) **Worst case**

Possible worst case scenarios:
i) 2 pivots we choose are the highest and lowest number.
ii) Array is in ascending or descending order.

The time complexity of partitioning part is $\theta(n)$ as the partitioning loop checks for every element in the array or sub-array to finish partitioning. Then, for the number of recursion calls QuickSort as per the scenarios mentioned above:

i) When two pivots chosen are the highest and smallest value of the array, then the partion would be like
[pivot 1] [..>= pivot1 and <= pivot 2 ..] [pivot 2]
This means all the other elements other than the pivots will be placed in between the pivots. This is equivalent to having partition lengths of 0, (n-2), and 0 respectively.

ii) When the list is sorted in ascending or descending order, the pivots will be the 2 smallest or largest elements from the array. So, the partitions will be divided into length of 0,0, and (n-2) for ascending, and (n-2),0,0 for descending.

Thus, the function would be something like this:
$T(n) = T(0) + T(0) + T(n-2) + \theta(n)$
$T(n) = \theta(1) + \theta(1) + T(n-2) + \theta(n)$
If we visualize it in terms of recursion tree, we can ignore linear time and say that it would have two child nodes per root : T(n-2) and $\theta(n)$. We get height = (n-2). If we go all the way down to the leaf, we get
$\mathrm{T(n)} = \theta(n) + (n-2)\theta(n)$
$\mathrm{T(n)} = \theta(n) + \theta(n^2)$
$\mathrm{T(n)} = \theta(n^2)$
hence, time complexity for worst case would be $\theta(n^2)$

Best Case
The best case would be when all the 3 partitions will be divided into equal length(i.e. n/3).
The time complexity of partition would remain same: $\theta(n)$
The time complexity function would be:
$T(n) = T(n/3) + T(n/3) + T((n/3) - 2) + \theta(n)$
This is equivalent to $T(n) = 3T(n/3) + \theta(n)$
Applying Master Method on this function, we get $a = 3, b = 3, fn = \theta(n)$
Since, $\theta(n^{\log_3 3}) = fn = \theta(n)$
By case 2 of Master Method, time complexity = $\theta(nlogn)$
Therefore, the best case time complexity has been proved as $\theta(nlogn)$

c) Code in random.c

Problem 5.3

We know that, lg(xy) = lg(x) + lg(y)

Similarly, $lg(n!) = lg(1) + lg(2) + ..lg(n-1) + lg(n)$

Breaking into 2 parts: S = T + U
where, T = lg(1) + lg(2) + .. + lg(n/2)
$U = lg((n/2) + 1) + ...lg(n)$
T has 1st $n/2$ terms of S, and U has the rest.

Now, we will lower bound each of them.

Each term in T is at least log(1),
so, $T >= lg(1) + lg(1) + ... + lg(1)$
$T >= 0$

Looking at U:
Each term in U is at least lg(n/2), so,
$U >= lg(n/2) + lg(n/2) + .. + lg(n/2)$
$U >= n/2lg(n/2)$

Now, S = T + U so,
$S = T + U >= 0 + n/2lg(n/2)$
$Lowerbound = \Omega(n/2(lgn/2))$

For upper bound:
lg(n!) = lg(1) + lg(2) + .. + lg(n)
lg(n!) = lg(n) + lg(n) + ... + lg(n)
lg(n!) = nlg(n)
Upperbound =$O(nlgn)$

since $\theta = O \cap \Omega$
$lg(n!) = \theta(nlgn)$ proved