Project 2

Ayush Singh (G29283616) | ayushsingh@gwu.edu

# Problem Statement

We are given an array A [1..n] of sorted integers that has been circularly shifted some positions to the right. For example, [5, 15, 27, 29, 35, 42] is a sorted array that is the original array, while [27, 29, 35, 42, 5, 15] has been shifted 6 positions. We can obviously find the largest element in A in O (n) time. Describe an O (log n) algorithm.

# Implementation Characteristics

Although, we all know that we need can search for the highest element in a Linear Time O(n), but our constraint is here to reduce to O (log n) time. We can most certainly us the Binary Search Algorithm which is already know to give O (log n) time, but as the array is not sorted, we need to modify the algorithm just a wee bit. If we notice the sorted array which has been rotated, we will observe that when we’ll divide it in 2 different halves, left and right, one of the halves will still sorted. This interesting analogy helps us to apply the binary search technique. All that is left for us is to find that which of the halves is actually sorted. If the left half is sorted, then find the element in that half, else, find the element in the other half. And Vice versa for the right half. No matter, in which half the highest element is present, we search recursively in the half where element might be present. The main analogy of this problem is that even though, the array is not sorted after the rotation, we can still use the binary search algorithm, because one of the half arrays will be sorted. This allows us to find a key element in O (log n) time. If an element is bigger than the element before it and after it, then we have found the highest element.

# Experimental Analysis

## Program Listing

int b\_search (int arr[],int low,int high)

{ int mid=(high/2);

if(high==1)

return 1;

if(arr[low]>arr[mid])

{return b\_search(arr,low,mid);}

if(arr[low]<arr[mid])

{

if(arr[mid]<arr[mid+1])

return mid + b\_search(arr,mid+1,high);

else

return mid;

}

}

## Data Normalization Notes

To normalize the theoretical values obtained and to compare the actual experimental results, we will have to multiply the theoretical values with 19,724.31 as the values are in nanoseconds.

## Output Numerical Data

|  |  |  |  |
| --- | --- | --- | --- |
| N | Experimental (ns) | Theoretical (log(n)) (ns) | Adjusted Theoretical Value (\*19,724.31) |
| 1000 | 218500 | 9.96 | 196454.1353 |
| 4000 | 245500 | 11.96 | 235902.7569 |
| 16,000 | 286200 | 13.96 | 275351.3784 |
| 64,000 | 309600 | 15.96 | 314800 |
| 256,000 | 346200 | 17.96 | 354248.6216 |
| 1,024,000 | 387000 | 19.96 | 393697.2431 |
| 4,096,000 | 410600 | 21.96 | 433145.8647 |
| **Average** | **314800** | **15.96** |  |
| **Ratio** |  | **19724.31078** |  |

## Graph

## Graph Observations

The graph above shows that both the lines coincide very well but there are some value of n where the lines do not coincide. This shows that even though, the theoretical values seem very fitting, the experimental values still have some sort of deviation. But, the overall hue of both the graph lines indicate that the time complexity calculated must be similar to that which was calculated.

# Conclusions

The analysis of the algorithm as O (log(n)) seems to be supporting the graph. May be a better analysis of the scatterplots can be revealed if we change programming environments, but this graph seems to give the user a good idea that the complexity, nevertheless, is O (log(n)).