CPSC 331 Assignment 1

Ali Akbari 30171539

February 2024

1.3.3 Iterative Time Test

For Questions 1.3.3 and 1.3.4 I decided to use nanoseconds as the functions were executing very quickly. I also standardized the algorithms to look for an element not in the array.

Time for an array of length 1: 179100 nanoseconds.

Time for an array of length 10: 183300 nanoseconds.

Time for an array of length 50: 188200 nanoseconds.

Time for an array of length 200: 193600 nanoseconds.

Time for an array of length 500: 201901 nanoseconds.

Time for an array of length 1000: 224500 nanoseconds.

The average run time for iterative binary search was: 195100.17 nanoseconds.

1.3.4 Recursive Time Test

Time for an array of length 1: 181900 nanoseconds.

Time for an array of length 10: 184500 nanoseconds.

Time for an array of length 50: 193800 nanoseconds.

Time for an array of length 200: 200900 nanoseconds.

Time for an array of length 500: 210500 nanoseconds.

Time for an array of length 1000: 228200 nanoseconds.

The average run time for recursive binary search was: 199966.67 nanoseconds.

1.3.5 Justification

Overall the times for recursive and iterative were very similar however, recursive was a bit slower. I believe this is the case because every time a function call is made in the recursive algorithm it must be allocated memory. For larger lengths of an array there must be more function calls and therefore more memory must be allocated. Then when the base case is reached all of the function calls must unwind and produce the final result. The more function calls there is the more time it will take to unwind and produce the final result. Compared to iterative where there are no function calls, therefore, it is clear why the recursive binary search is slower than the iterative version.

1.3.6 Correctness

To show the correctness of recursive binary search I am going to need to:

1. Show that the algorithm terminates.
2. Show that it outputs the correct value.

Let n be equal to $end - begin + 1$, ($n = end - begin + 1$). This then gives us the length of the array as n.

Base Case: when $n = 1$ or when the length of the array is 1 this means that $begin = end$, because ($1 = end - begin + 1$). Then in the algorithm $mid = (end + end)/2$, which would then be equal to $mid = (2 * end)/2$ and $mid = end$, this would also be the same for begin so $mid = begin$. This just shows that there is 1 element in the array as the beginning end and the middle are all the same. Then this must be the case where $x = a[mid]$, because we only have one index in the array we must be at the target. If the target is greater than the number at mid, $begin = mid + 1$ or in other words because begin and mid and end are all the same, $begin = begin + 1$. In the other case $end = end - 1$. Since they are all the same this would trigger the case where $begin > end$ and return -1.

Inductive Step:

Inductive Hypothesis: Assume that the algorithm Recursive Binary Search returns the correct index for all sizes of an sorted array for a length of n. Thus,

we need to show that the algorithm for Recursive Binary Search is true $n = k+1$ for all $n \geq 1$.

Inductive Claim:

Case 1: $x = arr[mid]$, for $k+1$ this means that if the target of the array is in the middle index it correctly returns the middle as the index of x. For $k+1$ this is also true because it returns the middle value if $x = arr[mid]$ for any length of an array.

Case 2: $x < arr[mid]$. This means that if the target is less than the middle element the target must be in the left half of the array. This is due to the fact that the array is already sorted. So the target must be found between $arr[begin]$ and $arr[mid-1]$. When an array of size $k+1$ is inputted the array is halved and the left side of the array is searched. Since $n = k+1$, $\frac{k+1}{2} = \frac{n}{2}$. By the IH all algorithms are true for a length of n therefore, the algorithm is true for a length of $\frac{n}{2}$. Each time this function is called it gets recursively smaller thus repeating until we find the element is not in the array or $x = arr[mid]$.

Case 3: $x > arr[mid]$. This means that if the target is greater than the middle element of the array the target must be in the right half of the array because the array is sorted. So the target must be found between $arr[mid+1]$ and $arr[end]$. Similar to case 2 when an array of size $k+1$ is inputted the array will be halved and the right half of the array will be looked through. This happens recursively and will keep giving smaller arrays until the target is either found and the index is returned or the target is not found and -1 is returned. Since $n = k+1$, $\frac{k+1}{2} = \frac{n}{2}$. By the IH all algorithms are true for a length of n therefore, the algorithm is true for a length of $\frac{n}{2}$.

Since the length of the array that is inputted is halved each time with each recursive call. The length of the array will eventually reach the base cases where the target is either in the array returning its index or it is not in the array returning -1. Thus, the recursive binary search algorithm is guaranteed to terminate.

2.3.1 Time Complexity Question 1

For each assignment statement $a = 0, b = 0, a = a + j$, and $b = b + k$ these are all big O(1). For each for loop they will all run n times because the loops are from 0 to $n-1$ so their time complexity is O(n). However, since two of them are nested for loops they will run $n * n$ times or n^2 times. Therefore their time complexity will be O(n^2). This time complexity dominates the O(n) of the other for loop so the whole time complexity is O(n^2). Since the best case will also always run n^2 times the big Ω is $\Omega(n^2)$. Because the time complexity for

big O and big Ω are the same the time complexity for big Θ will be big $\Theta(n^2)$. Therefore the time complexities for question 1 are big $\Theta(n^2)$, big $\Omega(n^2)$, big $O(n^2)$.

2.3.2 Time Complexity Question 2

For each assignment statement, $count = 0$, and $count = count + 1$ these have a run time of big O(1). The first for loop runs from $n/2$ to n so this loop has a run of time big $O(n)$ because the loop will run until what n is. For the second and third for loop, j and k double in each step of the loop. j and k start at 1 and double each time the loop runs until they reach n. Because you are doubling j and k you are going by powers of 2 for every step of j until you reach n. Therefore the number of steps will be equal to 2^j or 2^k. To find the number of iterations we can make this equal to n so $2^j = n$, solving this for j, yields $j = log(n)$ and this will be the same for the third for loop. Multiplying all of the time complexities brings $O(nlog^2n)$. In the best case scenario the first for loop will still run n times and the second and third for loops will still also be a run time of $log(n)$. This is because the loop is starting from 1 and incrementing to n by doubling. Therefore, the best case scenario is big $\Omega(nlog^2n)$. Since big O and big Ω are the same big Θ will be equal to big $\Theta(nlog^2n)$. Therefore, the time complexities for question 2 are big $O(nlog^2n)$, big $\Omega(nlog^2n)$, and big $\Theta(nlog^2n)$.

2.3.3 Time Complexity Question 3

The time complexity for this question will be exponential in regards to a. Every time a function call is made two additional calls with a smaller value of a is made (a-1) until it reaches the base cases. This results in a tree like structure where each node is a function call and there are twice as many child nodes as parent nodes so it grows exponentially. Therefore the time complexity in terms of big O is $O(2^a)$. In the best case scenario the algorithm will only run once. This is because of the if statement which will immediately return 1 if a value less than or equal to 0 is passed in as an argument. If a value less than or equal to 0 is passed in, it will only run once and will not run again. If a value bigger than 0 is passed in the function will run twice because of the recursive calls. Thus, making big Ω to be big $\Omega(1)$. Since the values of the big Ω and big O are different we are not able to give a value for big Θ. Therefore, the time complexities for question 3 are big $O(2^a)$, big $\Omega(1)$.

2.4.1 Space Complexity Question 1

The space complexity for this question is big O(1). This is because each statement is a assignment statement and no additional space is used. Since the worst

case scenario is constant time and we can not have run times better than constant time this means that big Ω is also big $\Omega(1)$. Therefore, because big O and big Ω are equal, big Θ will also be big $\Theta(1)$. The space complexities for question 1 are then big $\Theta(1)$, big $\Omega(1)$, and big $O(1)$.

2.4.2 Space Complexity Question 2

The space complexity for this question is big $O(n)$. This is because the array is initialized as being a size of $n + 1$. In the function there are two assignment statements and a for loop which only change the values of the numbers in the array not the size. Therefore, these would not contribute to the space complexity. The size of the array grows linearly with respect to n. The best case scenario would also require a space of n because the array is initialized as n+1 and is never changed throughout the function. Therefore, big Ω would be equal to big $\Omega(n)$. Since big Ω and big O are equal big Θ will also be equal to big $\Theta(n)$. The space complexities for question 2 are then big $\Theta(n)$, big $\Omega(n)$, and big $O(n)$.

2.4.3 Space Complexity Question 3

The space complexity for this question is big $O(m * n)$. This is because there is a 2d array initialized as a size of $(m + 1) * (n + 1)$. The size of this array grows in proportion to the size of the strings. Inside the double for loops we do not change the sizes of the arrays and therefore we do not change the space complexity as well. In the best case scenario the algorithm still requires space that is $m * n$. This is because the array is initialized as $(m + 1) * (n + 1)$. Big Ω would then be equal to big $\Omega(m * n)$. Since big Ω and big O are equal big Θ will also be equal to big $\Theta(m * n)$. The space complexities for question 3 are then big $\Theta(m * n)$, big $\Omega(m * n)$, and big $O(m * n)$.