CSCI 2300
Introduction to Algorithms: HW5

Sean Hyde
RIN: 662096071

Due: October 19th, 2024

1 2.16

1.1 Algorithm

Due to the unknown value of n, we must decide a starting value to check against the value of x. For this, we can use 2^1 as this is our base case since if n is below 2 the algorithm will not be viable. We can then check the value of x at position $a[2]$.

From there are two options, either, the value at $a[2] > x$ or the value at $a[2] < x$. Starting with the value of $a[2] > x$ we can determine that the value we are looking for must be in the bound of [0...2] as the position can't be greater than $a[2]$ since the array is sorted so the value we are looking for must be in a position lower than $a[2]$. From here we run binary search from the array created, $a[0...2]$, which will find the value x we are searching for by partitioning the array in 2 and comparing values and eliminating sides based on if the value at the middle of the partition is greater than x or less than x. If it is less than x it will eliminate the left/lower side, or if it is greater than x it will eliminate the right side and partition the left side and continue the operation until a value is found. However, if the value of $a[2] < x$ we can't derive any information from this as we don't know where to begin searching. We can then look for the position at $a[2^2]$/ $a[4]$ and check once more if the value at this position is greater to or less than the value. We will continue to iterate positions k times a^{1+k} until we find a position where the value at that index is greater than the value of x so we can conduct binary search which will find the value on the bounds of $a[0...2^1k]$ where the value at $a[2^k] > x$ so we can perform binary search.

1.2 Runtime

The runtime of this algorithm is $O(logn)$ as binary search's runtime is $O(logn)$ and the other operations in the algorithm are dominated by $O(logn)$.

2 2.17

The algorithm that can solve this problem can be stated as a modified version of binary search where we must compare the index of the position $a[i]$ with the value at $a[i]$.

2.1 Algorithm

We still begin by running binary search, however, if the index is bigger than the value, we must decrease the index we are searching for, and if the index is smaller than the value we are searching for we must increase the index as the array is sorted. For example, if the value of position $a[40] = 60$, we know that any other indices above 40 will have values larger than 60 meaning we can eliminate that half of the array. Similarly, we also know that the index that can potentially contain a value equal to its index must be below $a[40]$ since the array is sorted and the value at $a[40] = 60$, so any value that will potentially be equal to its index has to be below $a[40]$.

2.2 Runtime

The runtime of this algorithm is still going to be equal to Binary Search's typical runtime of $O(logn)$ as we are not changing the searching functionality and we are still eliminating half of the search space, so our runtime will remain equal to Binary Search's runtime of $O(logn)$.

3 2.19

3.1 (a)

The time complexity of the merging operation is $\frac{k(k+1)}{2}$ or $O(n^2)$.

3.2 (b)

To design an algorithm that is more efficient we can look to merge the array into subset arrays similar to how mergesort operates. We start by breaking the elements into singleton arrays. Then we begin with merging the 1st element with the 2nd element, the 3rd with the 4th, and so on until k. From here we can then continue merging in pairs until we result in a sorted array of size kn. This algorithm is more efficient as it's runtime is $O(kn)$

4 2.23

4.1 Algorithm

To solve this problem with the time constraint of $O(nlogn)$, we can divide the array at each recursive step into two halves until we reach our base case of if an array has one element that is the majority element. We will then find the majority candidate of the left half. If the majority element candidate from two arrays are the same we will return that as being a candidate for the entire array. If the candidates are different we count how many times it appears in the array. We then find the majority element of the right half following the same approach If either majority element candidate from the left or right half occurs more than $\frac{n}{2}$ times, that is the majority element.

4.2 Runtime

This algorithm runs in $O(nlogn)$ time as the iterative part of the algorithm where we must count for the occurrences of the majority element occur in $O(n)$ time and the dividing part of the algorithm occurs in $O(logn)$ time resulting in a final run time of $O(nlogn)$ time.