CP312: Algorithm Design & Analysis

Assignment #3

CP312, WLU, 2022

Instructor: Masoomeh Rudafshani

Due: Friday, February 11th, 2022

Declan Hollingworth
190765210
holl5210@mylaurier.ca

Nishant Tewari
190684430
tewa4430@mylaurier.ca

Q1. Is the following True or False? Justify your answer:
For every n > 1, there are n-element arrays that are sorted faster by insertion sort than by quicksort?

Given the way these two sorting algorithms work, the answer may very depend on the size of the array. Typically, insertion sort is faster than quicksort when the size of the array is small because quicksort uses recursion which is slower for very small arrays.

The time complexity of insertion sort is $O(n^2)$ and the time complexity of quick sort is $O(nlogn)$.

To gain a better understanding of how this two sorting algorithms run, we can calculate the runtime for sorting n sized arrays.

Size of array (n)	**Insertion Sort Time (seconds)**	**Quick Sort Time (seconds)**
50	0.0	0.0010490417480468755
100	0.0009996891021728516	0.0019960403442382812
500	0.025996685028076172	0.015000104904174805
1000	0.0809946060180664	0.07300043106079102

green indicates the faster runtime

Given that both these algorithms run incredibly fast, the runtime alone can't justify an answer. However, as the size of the array increased, the faster algorithm changed from insertion to quick sort around the n = 100 mark.

An example of when insertion sort would be faster would be with the array [1, 2, 4, 3]. The algorithm would only have to move 4 & 3, being much faster than the quick sort algorithm.

Based on the evidence provided above, we can conclude that insertion sort is faster for arrays with approximately 100 elements or less.

Q2. Design an algorithm to rearrange elements of a given array of n numbers so that all its negative elements precede all its positive elements. Analyze the time complexity of your algorithm and prove the correctness of your algorithm.
Hint: You can use a partitioning technique similar to that of quicksort.

```
def rearrangeArray(Arr, low, high):
        current = low - 1
        for i in range(low, high):
                if(Arr[i] < 0):
                current += 1
                Arr[current ], Arr[i] = Arr[i], Arr[current];
```

The algorithm uses a for loop to go through the elements of the array from low to high(referenced from one side of array to the other). The algorithm compasses the array value of where i is within the array, and it will check if the value is less than 0 and swap it with current value (Arr[curr]).

Since the algorithm will loop through the elements within the array n times(high), therefore the time complexity of this algorithm is Θ(n) and the space complexity is O(n).

<u>Proof of Correctness</u>

Loop Invariant:
The loop invariant for the pseudocode shown above is that the elements within the array are unsorted.

Initialization:
The elements within the array are unsorted before iterating through the for loop. Loop invariant is true because the array begins at the beginning of the array before it iterates which makes nothing sorted thus far.

Maintenance
Within the iteration of the for loop in the algorithm , if Arr[i] is less than 0, then we swap Arr[i] and Arr[current] and increment the value of current.

Termination

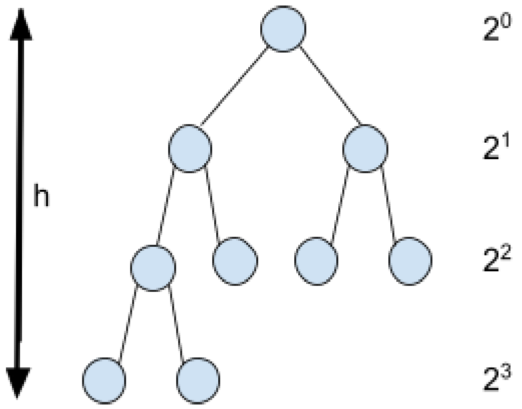
Q3. Prove that the problem of finding an element in an array having n element has a lower bound of Ω(log n). You can use a technique similar to what we have done in class for proving the lower bound for sorting algorithms.

When searching through an array of n elements, there is a variety of methods to complete the operation. The easiest option is a linearly traversing algorithm that walks through the array, which would be O(n). However, by using a recursive binary search tree method with a sorted array the runtime is much better. So when proving the lower bound of Ω(log n), we will be analyzing the binary search method.

Binary search code for reference (retrieved from Divide-and-Concquer.pdf, slide 4):

```
def binarySearch(A, low, high, key):
     if low > high:
          return false
     mid = (high + low) // 2
     if A[mid] = key:
          return true
     else if A[mid] > key:
          return binarySearch(a, low, mid)
     else (A[mid] < key):
          return binarySearch(a, mid + 1, high)
binarysearch(A, 0, len(A)-1, key)
```

As discussed in-class lectures, a decision-tree model can be used to analyze/prove the lower bound of an algorithm. With this method, we determine the time it takes to search the array, which is equal to the time it takes to travel from the root (initial position) to the lead (final position) this distance can be referred to as the height of the tree (h).



$$n \leq 2^{h+1} - 1$$
$$log(n) \leq (h + 1)log2 - log1$$
$$log(n) \leq (h + 1)$$
$$log(n) - 1 \leq h$$
$$h = \Omega(logn)$$

Given that the height of the tree spans $\Omega(logn)$ and the operation at each node is O(1), we can confirm the lower bound of a search algorithm is..
$logn * 1 = logn$

∴ the problem of finding an element in an array having n element has a lower bound of Ω(log n)

Q4. Suppose there is a road and the houses are scattered along the road.
The position of each house is specified by its distance from the western endpoint. We want to place cell-phone towers on the road so that all houses are covered by at least one tower. Design a greedy algorithm to find the minimum number of towers required. You need to write the pseudocode for the problem, the proof of optimality, and proof of correctness for the algorithm. Note: this is a one-dimensional problem. The houses and towers are all placed on one line.

```
def tiltedTowers (houses, radius):

#Checking if there are no houses, then return 0 towers
    if  len(houses) == 0:
        Return 0

#place the first tower the farthest away from the houses
    distance = houses[0] + radius
    towers +=1

    for i in range(len(houses)):
        if abs( i - distance) > radius
            tower +=1
            distance = i + radius
    Return towers
```

<u>Proof of Correctness:</u>

Loop Invariant:
For the iteration of the for loop in the algorithm, the elements within the houses array is a smaller distance from the farthest distance where the first tower is place away from the houses

Initialization:
Base case of the algorithm is shown by checking if houses are equal to 0 which would mean that there are zero towers needed. The first tower will be placed farthest away from the first house.

Maintenance:
Within the iteration of the for loop in the algorithm, if the current house has a distance greater than its radius from the next house in the list, then a tower must be placed to cover the distance between the house.

Termination:
The for loop will iterate through the length of the array houses where the condition will be in check to place a tower that covers the distance between the houses. The loop will terminate and return the # of towers needed to cover the houses with service.