CPSC 319 L01 W16 HW 1 Report

Part A

Complexity Analysis: Binary Search Tree

Worst Case: Key not found

The amount of elements to search is cut in half with each iteration or recursive call. The worst case occurs when the number of elements left to search is 1.

$$\frac{n}{2^m} = 1, \qquad where\ n = \#elements, m = \#iterations$$

Rearranging for m: $\quad m = \log_2 n$

∴ the algorithm's worst case runs in $O(\log n)$

Average Case: Assuming all the key is equally likely to be in any element (or not at all)

$level$	$iterations/recursions$
1	1
2	4
3	12
4	32

We can establish a relationship between the required number of iterations/recursions and the level of depth that the key is found in:

$$m = k(2^{k-1}), \qquad where\ k = level, m = \#iterations$$

Since there are essentially $\log n$ levels, the total amount of iterations is:

$$m = \sum_{k=1}^{\log n} k(2^{k-1})$$

Since Big-O notation is just an upper bound, we can find a close-enough summation that is always larger, and manipulate that into something that fits into Big-O notation:

$$m = \sum_{k=1}^{\log n} k(2^{k-1}) \leq \frac{\log n}{2} \sum_{k=1}^{\log n} k(2^k) = (n-1)\log n$$

Averaging the total iterations, we get the average complexity to be:

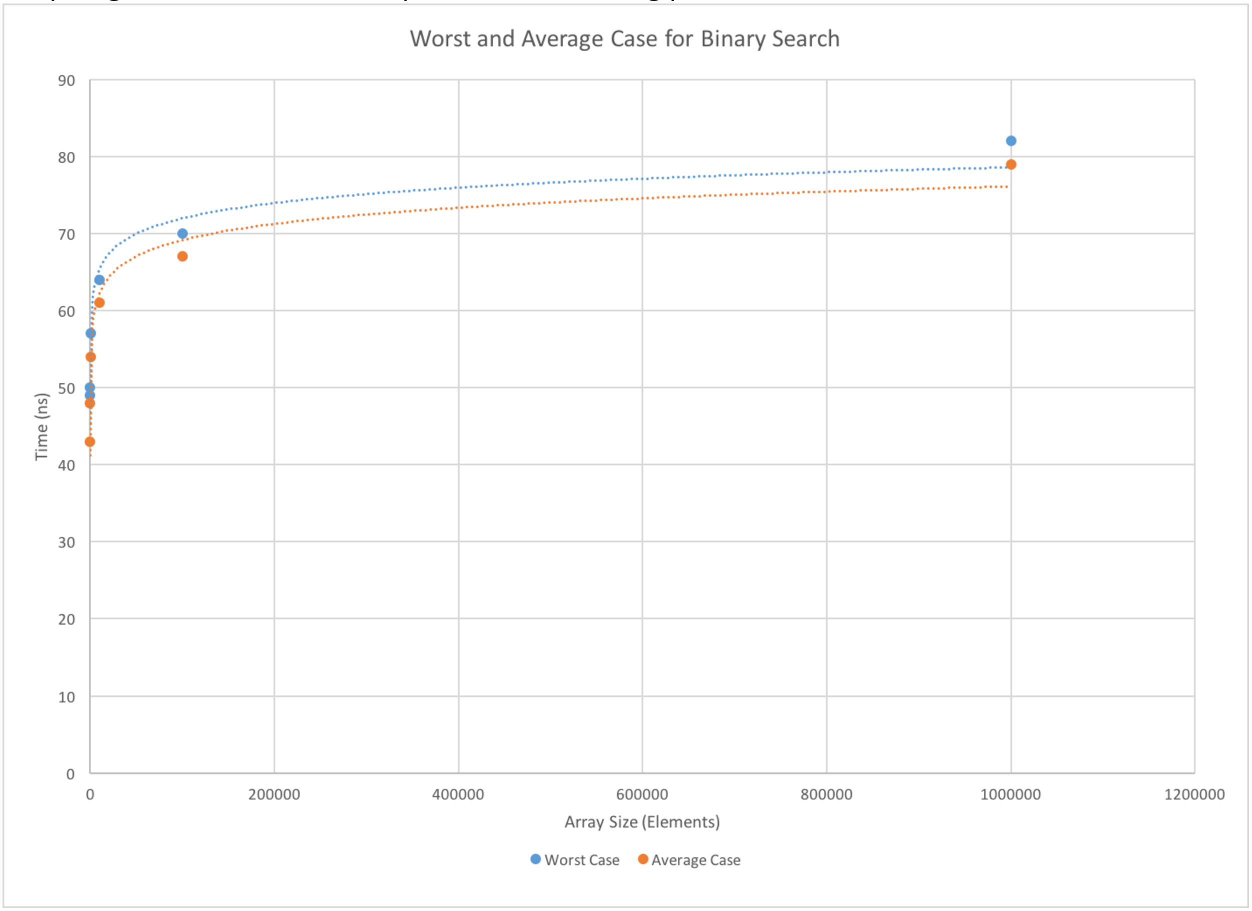
$$m \leq \frac{(n-1)\log n}{n} = \frac{1}{n}(n \log n - \log n)$$

∴ the algorithm's average case runs in $O(\log n)$

Using the BinarySearchBM program, which averages 500000 computation times for the algorithm on different sizes of arrays, we can see the experimental results:

```
Mitchell@ttys004 15:16 {0} [1]$ java BinarySearchBM
Binary Search Benchmark for array with length 10:
          Worse Case (Average over 500000 trials): 49ns
        Average Case (Average over 500000 trials): 43ns
Binary Search Benchmark for array with length 100:
          Worse Case (Average over 500000 trials): 50ns
        Average Case (Average over 500000 trials): 48ns
Binary Search Benchmark for array with length 1000:
          Worse Case (Average over 500000 trials): 57ns
        Average Case (Average over 500000 trials): 54ns
Binary Search Benchmark for array with length 10000:
          Worse Case (Average over 500000 trials): 64ns
        Average Case (Average over 500000 trials): 61ns
Binary Search Benchmark for array with length 100000:
          Worse Case (Average over 500000 trials): 70ns
        Average Case (Average over 500000 trials): 67ns
Binary Search Benchmark for array with length 1000000:
          Worse Case (Average over 500000 trials): 82ns
        Average Case (Average over 500000 trials): 79ns
Mitchell@ttys004 02:42 {0} [1]$
```

Graphing the results, we end up with the following plot:



As is clear from the logarithmic trend lines, both the worst and average case times are governed by the logarithm of the array size. This is consistent with the previous analysis. As the array size trends towards infinity, it appears that the average case is consistently 3ns faster than the worst case. Our analysis shows that that the average case is always less than $\frac{1}{n}(n\log n - \log n)$, which is also consistent with the plot.

Part B

An example implementation of Bubble-Down Bubble Sort is as follows:

```
for (int i = arr.length - 1; i > 0; i--) {
    for (int j = 0; j < i; j++) {
        if (arr[j] > arr[j + 1]) {
            int temp = arr[j + 1];
            arr[j + 1] = arr[j];
            arr[j] = temp;
        }
    }
}
```

Complexity Analysis: Bubble-Down Bubble Sort:

Since this implementation of Bubble-Sort does not stop comparing and iterating even if no swaps were made in the last iteration, the number of times it loops is always the same. The only portion of the algorithm that can contribute to making it faster is therefore the amount of swaps it does.

On a given iteration of the outer loop, the inner loop runs $n-i-1$ times. Summing this over the number of times the outer loop runs, we get the total number of iterations to be:

$$\sum_{i=0}^{n-2}(n-1-i) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Worst Case: Array is already sorted, but in the opposite direction

In this case, the algorithm swaps array elements every time it loops, so it swaps $\frac{1}{2}n^2 - \frac{1}{2}n$ times. ∴ this case has $O(n^2)$ comparisons and $O(n^2)$ swaps.

Best Case: Array is already sorted

In this case, the algorithm doesn't swap any array elements.
∴ this case has $O(n^2)$ comparisons and $O(1)$ swaps.

Average Case: Assuming all permutations of array state are equally likely:

The inner loop runs from 0 to $n-i-1$, so the total number of swaps in the inner loop is $\sum_{j=0}^{n-i-1} j$. Averaging these swaps together, we get

$$\frac{\sum_{j=0}^{n-i-1} j}{n-i} = \frac{n-i-1}{2}$$

The inner loop runs $n-2$ times, so summing every average swap gives us:

$$\sum_{i=0}^{n-2}\frac{n-i-1}{2} = \frac{n(n-1)}{4} = \frac{1}{4}n^2 - \frac{1}{4}n$$

∴ this case has $O(n^2)$ comparisons and $O(n^2)$ swaps.

How does this compare to Bubble-Up Bubble Sort?

Since the bubble-down implementation is not doing anything different than the bubble-up implementation (other than the direction of the swap, it makes sense that all cases share the same time complexities.

How can this algorithm be optimised for nearly-sorted arrays?

A large optimization would be to stop comparing elements and iterating the outer loop when the previous loop did not have to do any swaps. For example,

```
for (int i = arr.length - 1; i > 0; i--) {
    boolean prevSwap = false;
    for (int j = 0; j < i; j++) {
        if (arr[j] > arr[j + 1]) {
            prevSwap = true;
            int temp = arr[j + 1];
            arr[j + 1] = arr[j];
            arr[j] = temp;
        }
    }
    if (!prevSwap)
        break;
}
```