CS 253
Section 02
Fall 2019
Christopher Schmitt Assignment 1

Problem 1

Table 1: Toy Data (0-9)

	Best Case		Average Case		Worst Case	
Algorithm	Exchanges	Comparisons	Exchanges	Comparisons	Exchanges	Comparisons
Bubble Sort	0	9	20	43	45	45
Selection Sort	9	45	9	45	9	45
Insertion Sort	0	10	20	30	45	55

Table 2: Test Data (0-1999)

	Best Case		Average Case		Worst Case	
Algorithm	Exchanges	Comparisons	Exchanges	Comparisons	Exchanges	Comparisons
Bubble Sort	0	1999	1004333	1997649	1999000	1999000
Selection Sort	1999	1999000	1999	1999000	1999	1999000
Insertion Sort	0	2000	1004333	1006333	1999000	2001000

These results can be explained by analyzing the complexity of the pseudocode of each of these three algorithms. Both bubble sort and insertion sort display very similar performance characteristics. An analysis of the pseudocode would indicate that both of these two algorithms have a computational complexity of $\mathcal{O}(n^2)$. The empirical data support this, showing that the number of comparisons increases with n^2 where n is the size of the input. It should be noted that both bubble sort and insertion sort are sensitive to their inputs (also refected in the best/worst case data points).

Selection sort, however, demonstrates some unique properties not found in either insertion sort or bubble sort. The data clearly show that, unlike bubble and insertion sort, selection sort is not sensitive to the input. Because selection sort is not sensitive to its input, it has no “worst” case or “best” case in the traditional sense. The data show that selection sort always takes the same number of operations to complete for any given input size (that is to say that there is a unique number of operations performed for every unique value of n). Like the other two, however, selection sort is also $\mathcal{O}(n^2)$ With the number of operations growing proportionally to the square of the input.

Problem 2

Binary Search

$$\begin{tikzcd}
 & & & \{0,1,2,\mathbf{3},4,5,6\} \arrow[dll, dash] \arrow[drr, dash] & & & \\
 & \{0,\mathbf{1},2\} \arrow[dl, dash] \arrow[dr, dash] & & & & \{4,\mathbf{5},6\} \arrow[dl, dash] \arrow[dr, dash] & \\
\{\mathbf{0}\} & & \{\mathbf{2}\} & & \{\mathbf{4}\} & & \{\mathbf{6}\}
\end{tikzcd}$$

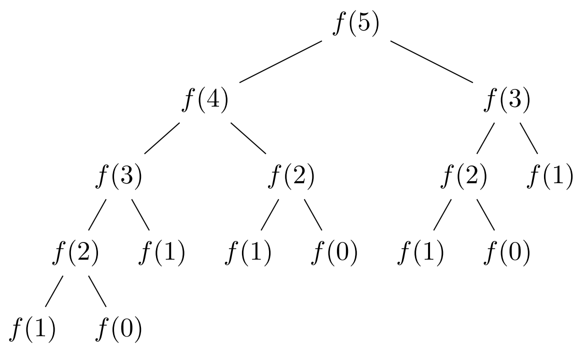
Recursive binary search is a searching algorithm that operates on sorted lists. Binary search starts by comparing the target value with value at the center of the list. If the values do not match, half the list is eliminated (this is possible because the list ist ordered, so the target value will not be in one of the halves) and the process is repeated on the remaining sublist. Analysis of the pseudocode indicates that this algorithm runs in $\log n$ operations (the size of the input is chopped in half at each call). The tree of recursive calls supports this conclusion, as the depth of the tree is equal to the $log2$ of the input size and a maximum of one path is followed. This means the computational complexity of binary search is $\mathcal{O}(\log n)$.

Factorial

$$\begin{tikzcd}
\text{factorial}(3) = 3 \times f(n-1) \arrow[d, dash] \\
\text{factorial}(2) = 2 \times f(n-1) \arrow[d, dash] \\
\text{factorial}(1) = 1 \times f(n-1) \arrow[d, dash] \\
\text{factorial}(0) = 1
\end{tikzcd}$$

The recursive factorial algorithm uses the recursive definition of factorial where $f(n) = n \times f(n-1)$ and $f(0) = 1$. This function has a complexity of$\mathcal{O}(n)$, which can be seen in the tree of recursive calls, where the depth of the tree grows linearly with the size of the input.

Fibonacci



The recursive Fibonacci algorithm uses the recursive definition of Fibonacci where $f(n) = f(n-1)+f(n-2)$ and $f(1) = 1$ and $f(0) = 0$ The big-o complexity of the recursive Fibonacci algorithm can be determined by solving the recurrence relation. The characteristic equation for the algorithm will be $n^2 - n - 1 = 0$ which has roots at $\frac{1-\sqrt{5}}{2}$ and $\frac{1+\sqrt{5}}{2}$ These roots can now be substituted into $f(n) = (a_1)^n + (a_2)^n$ leaving $f(n) = (\frac{1-\sqrt{5}}{2})^n + (\frac{1+\sqrt{5}}{2})^n$. Dropping the lower term leaves a Computational complexity of $\mathcal{O}((\frac{1-\sqrt{5}}{2})^n)$, which is supported bu the tree of recursive calls.

Implementations

```java
/**
* Binary search determines weather a given element exists whithin
* a sorted list or not.
*/
public static <T extends Comparable<T>> Boolean binarySearch(T object, List<T> list, Integer start, Integer end) {
  int middle = (start + end) / 2;

  if (end < start) return false;
  if (object.compareTo(list.get(middle)) < 0) return binarySearch(object, list, start, middle - 1);
  if (object.compareTo(list.get(middle)) > 0) return binarySearch(object, list, middle + 1, end);
  if (object.compareTo(list.get(middle)) == 0) return true;

  return false;
}
```

```java
/**
 * Compute the factorial of n by using the recursive definition.
*/
public static Integer factorial(Integer n) {
  return n == 0 ? 1 : n * factorial(n - 1);
}
```

```java
/**
 * Compute the nth fibonacci number by using the recusive defintion.
 */
public static Integer fibonacci(Integer n) {
  return n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2);
}
```