Jack Kapino

CSC 253 Data Structures and Files

Homework 1

Problem 1:

Insertion Sort- The Insertion Sort Algorithm has two nested loops, which means that as the number of elements (n) in the array grows it will take approximately n * n longer to perform the sorting. The big-O notation for this algorithm in the majority of cases is O(n^2).

Best Case: O(N) Where the elements in the list/array are almost sorted. The inner while loop will never go through all the elements in the array because condition(array[i-1] > array[i]) will never be met. The best-case efficiency for insertion sort is O(N). Using protoType1.txt which consists of 10 elements already in sorted sequence (1-10). Insertion sort preforms 9 comparisons one for each element in the list. It also has 0 swaps since the list is ordered. Applying the same insertion sort algorithm to protoType4.txt which consists of 2000 integers (1-2000). 1999 comparisons are preformed and 0 swaps.

Average Case: O(N^2) The average case was determined by using ProtoType3.txt a text file of 10 integer in the range of (1-10) in random sequence. Insertion sort algorithm preforms 29 swaps and 35 comparisons. Using data from protoType6.txt which contains 2000 integers (1-2000) in random sequence. 1,025,715 swaps are preformed along with 1,027,710 comparisons.

Worst Case: O(N^2): Worst case would be where elements are sorted in reverse order so that and every element requires a swap since every element meets the condition (array[i-1] > array[i]). Using data from Prototype2.txt text file with 10 integers sequenced (10-1). Worst case scenario 45 comparisons and 45 swaps are made. Using protoType5.txt containing 2,000 integers sequenced (2000 to 1). The insertion sort algorithm makes 1,999,000 swaps and 1,999,00 comparisons.

Selection Sort- Selection sort Algorithm traverses through the array searching for the minimum value and placing it at the beginning of the array. The time efficiency for this algorithm is 0(N^2) it is directly proportional to the square of the size of the input data due to the nested for loop.

Best Case: O(N^2) The best scenario is when all the elements are already or almost sorted. Since this algorithm has no easy exit scenario all the elements must be checked regardless of if the list is already in order. (n^2 - n)/2 comparisons are always made to the data set regardless of the order. Using protoType1.txt which consists of 10 integers in ascending order, 45 comparisons are made and 0 swaps.

Worst Case: O(N^2) The worst case time efficiency for selection sort comes when the list is in a random order such that every element must be checked in order to find the minimum value of the list. Using ProtoType3.txt 10 integers in random order, the selection sort algorithm will preform 9 swaps and 45 comparisons. The random order of the list requires the most amount of swaps versus ascending (1-10) and descending (10-1) ordered lists.

Average Case: O(N^2) The average case time efficiency for selection sort is still O(N^2) since (n^2 - n)/2 comparisons are always made.

Bubble Sort:- Bubble sort algorithm passes through an array/list multiple times each pass compares adjacent elements in the list, and exchanging elements if they r in the list. Each pass bubbles the largest element to the end of the list. Bubble sort has a time complexity of O(N^2) and is only useful for small data sets.

Best Case O(N): The best case time efficiency occurs when the list Is already sorted and no swaps will occur. In order to achieve linear time complexity O(N) with bubble sort some sort of flag would be used to indicate that the list is ordered. If after 1 iteration through the entire list no swaps occurred then a break from the loop could be placed there to get linear time complexity Using ProtoType1.txt 10 ascending (1-10) integers 0 swaps are made and 45 comparisons are made.

Average Case 0(N^2): The average case for this algorithm is determined using sets of random integers from protoType3.txt 10 Integers in random order. That data set resulted in 29 swaps and 45 comparisons. The second set of data protoType6.txt contains 2000 integers that sorted randomly. With 2000 random elements there is 1,025,715 swaps and 1,999,000 comparisons made.

Worst Case O(N^2): The worst case time complexity for bubble sort algorithm comes when the list is in descending order and every element requires a swap.

Time Complexity:

Algorithm	Best Case:	Worst Case:	Average Case:
Insertion Sort	O(N)	O(N^2)	O(N^2)
Selection Sort	O(N^2)	O(N^2)	O(N^2)
Bubble Sort	O(N)	O(N^2)	O(N^2)

protoType1.txt=(1-10), protoType2.txt =(10-1), protoType3.txt =(random order 10 elements)

protoType4.txt = (1-2,000), protoType5.txt =(2,000-1), protoType6.txt = (random order 2k elements)

Algorithm Compares & Swaps:

Algorithm	Prototype /data used	Number of Swaps	Number of Compares
Insertion Sort	ProtoType1.txt	0	9
Insertion Sort	ProtoType2.txt	45	45
Insertion Sort	ProtoType3.txt	29	45
Insertion Sort	ProtoType4.txt	0	1999
Insertion Sort	ProtoType5.txt	1,999,000	1,999,000
Insertion Sort	ProtoType6.txt	1,025,715	1,027,710
Selection Sort	ProtoType1.txt	0	45
Selection Sort	ProtoType2.txt	5	45
Selection Sort	ProtoType3.txt	9	45
Selection Sort	ProtoType4.txt	0	1,999,000
Selection Sort	ProtoType5.txt	1000	1,999,000
Selection Sort	ProtoType6.txt	1989	1,999,000
Bubble Sort	ProtoType1.txt	0	45
Bubble Sort	ProtoType2.txt	45	45
Bubble Sort	ProtoType3.txt	29	45
Bubble Sort	ProtoType4.txt	0	1,999,000
Bubble Sort	ProtoType5.txt	1,999,000	1,999,000
Bubble Sort	ProtoType6.txt	1,025,715	1,999,000

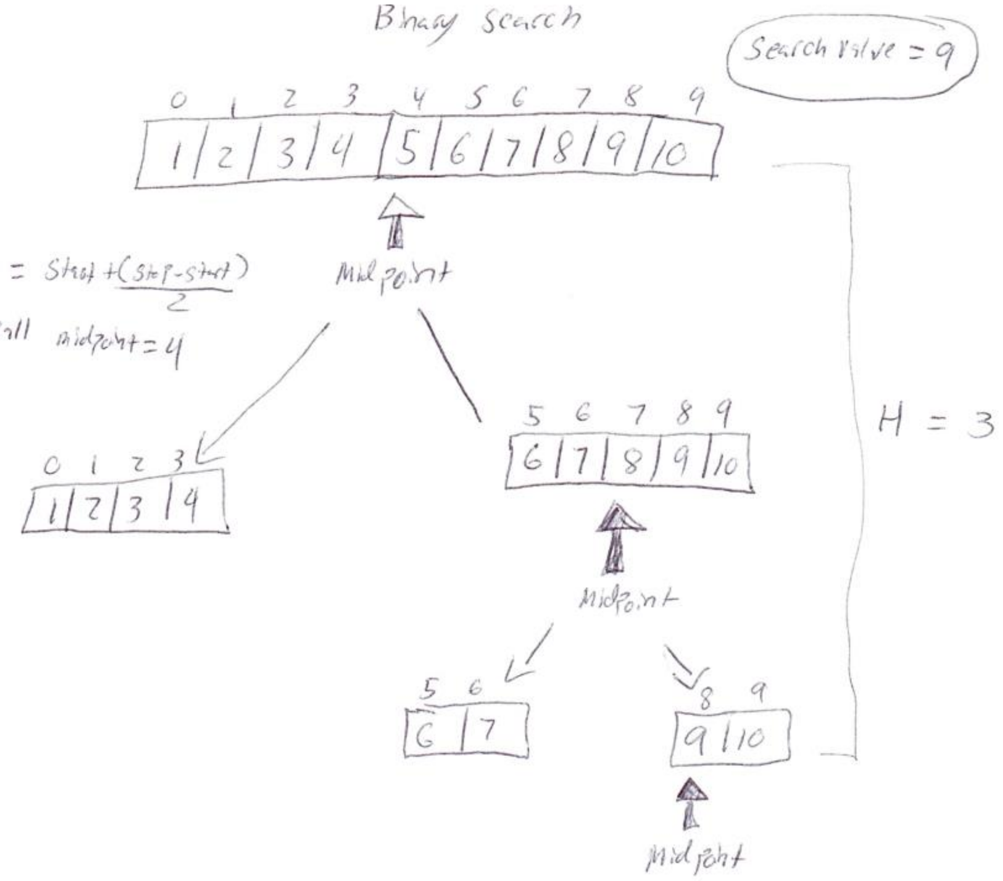
Problem 2:

Binary Search – The binary search algorithm creates a midpoint and compares that midpoint with a search value specified by the parameters. After each comparison the search range is divided into half of current search range. The Time complexity of this algorithm is **O(log N),** the amount of divides need for any size of data can be thought of as(log(Base 2) N = K) . Where N is the number of elements in the list, and K will be equal to the total number of times a user will need divide the list until the last element is reached and the search value is found or not found.

Factorial -The recursive factorial function simply works by accepting an integer parameter n. If n is equal to 1 a base case scenario comes into play causing recursion to end. The recursive calls multiplying the current number by the factorial of the natural number one less than it creating a stack of calls. Big O notation for factorial algorithm **is O(N)** the function calls execute every time the value of n is decremented, meaning the function is called recursively n times.

Fibonacci- The n-th Fibonacci number algorithm time complexity starts with 0(1) constant time for the base case of the algorithm where n=1. The next part of algorithm the answer/recursive call f(n-1) +f(n-2)+f(n-3)…. is exponential time **O(2^N).** Where the time taken to calculate fib(n) is equal to the sum of time taken to calculate fib(n-1) and fib(n-2).

Binary Search



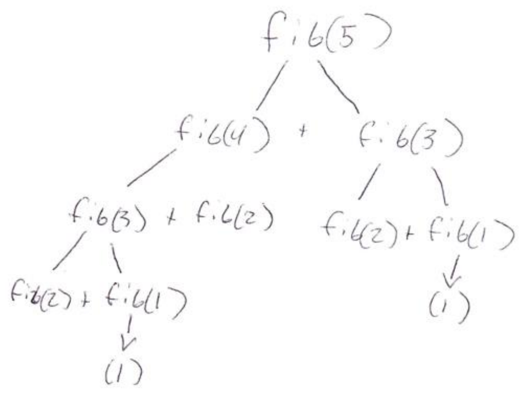
$\text{midpoint} = \text{start} + \frac{(\text{stop} - \text{start})}{2}$

1st call midpoint = 4

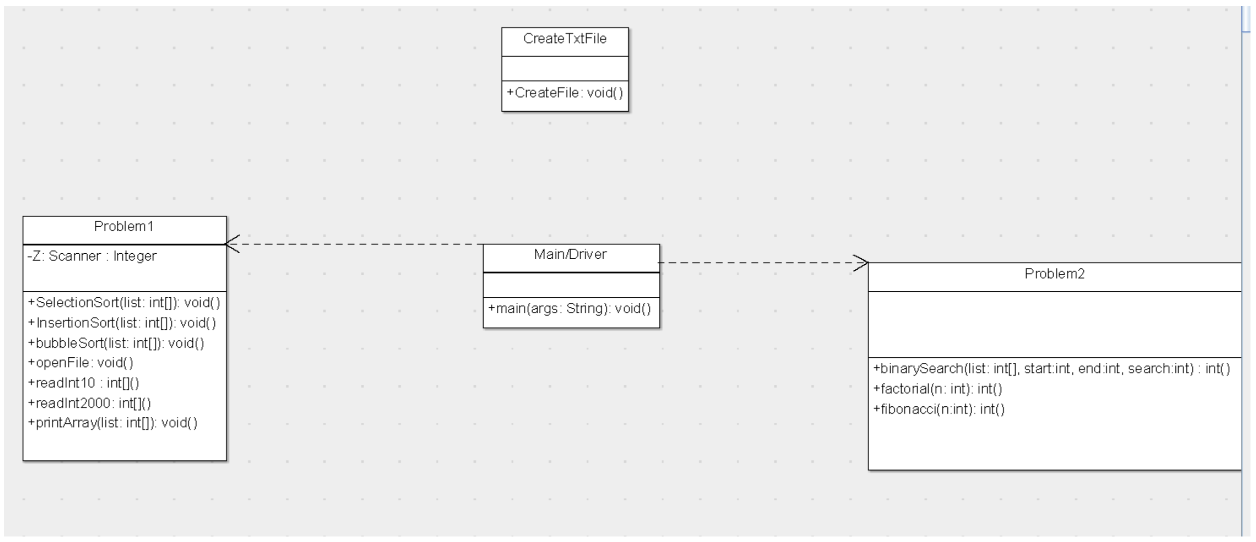
• Height of Tree = # of Divides on the Data Set

Total value = 3,628,800

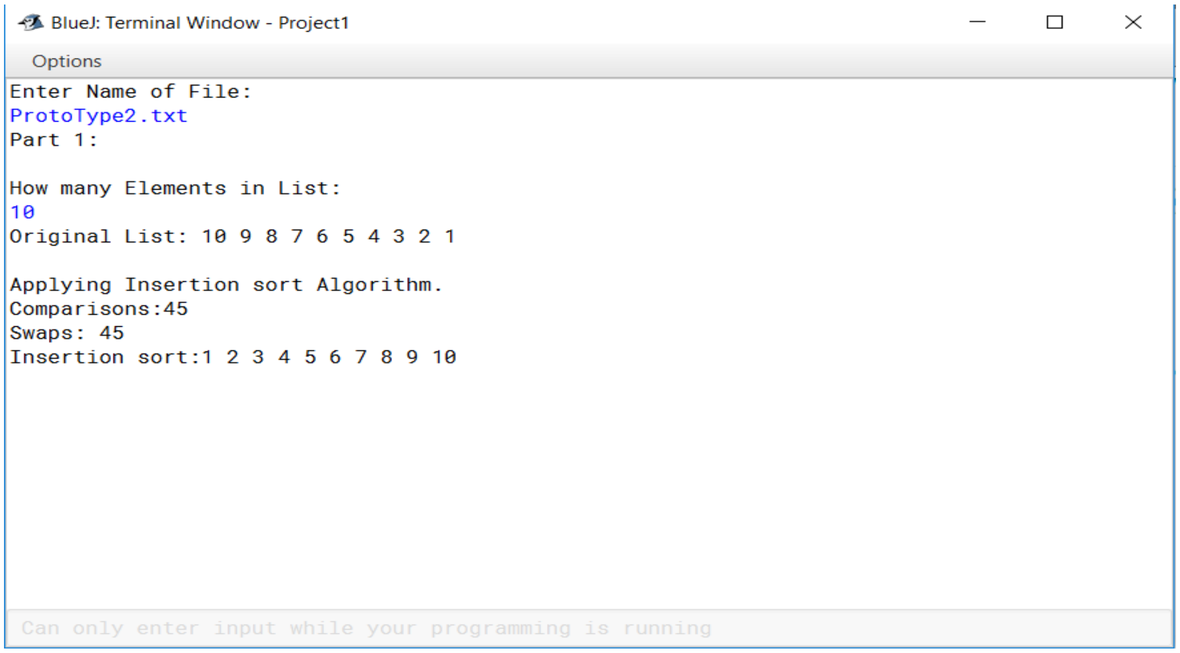
$$\begin{tikzcd}
 & \text{factorial}(10) \arrow[d] \\
\boxed{3,628,800} & 10 \times \text{factorial}(9) \arrow[d] \\
\boxed{362880} & 9 \times \text{factorial}(8) \arrow[d] \arrow[ul, bend left] \\
\boxed{40320} & 8 \times \text{factorial}(7) \arrow[d] \arrow[ul, bend left] \\
\boxed{5040} & 7 \times \text{factorial}(6) \arrow[d] \arrow[ul, bend left] \\
\boxed{720} & 6 \times \text{factorial}(5) \arrow[d] \arrow[ul, bend left] \\
\boxed{120} & 5 \times \text{factorial}(4) \arrow[d] \arrow[ul, bend left] \\
\boxed{24} & 4 \times \text{factorial } 3 \arrow[d] \arrow[ul, bend left] \\
\boxed{6} & 3 \times \text{factorial}(2) \arrow[d] \arrow[ul, bend left] \\
\boxed{2} & 2 \times \text{factorial}(1) \arrow[d] \arrow[ul, bend left] \\
 & \boxed{\text{Base case}} = 1 \arrow[ul, bend left]
\end{tikzcd}$$


fib(5)
fib(4) + fib(3)
fib(3) + fib(2)
fib(2) + fib(1)
fib(2) + fib(1)
(1)
(1)

UML Diagram


CreateTxtFile
+CreateFile: void()
Problem1
-Z: Scanner : Integer
+SelectionSort(list: int[]): void()
+InsertionSort(list: int[]): void()
+bubbleSort(list: int[]): void()
+openFile: void()
+readInt10 : int[]()
+readInt2000: int[]()
+printArray(list: int[]): void()
Main/Driver
+main(args: String): void()
Problem2
+binarySearch(list: int[], start:int, end:int, search:int) : int()
+factorial(n: int): int()
+fibonacci(n:int): int()

Program 1 output:

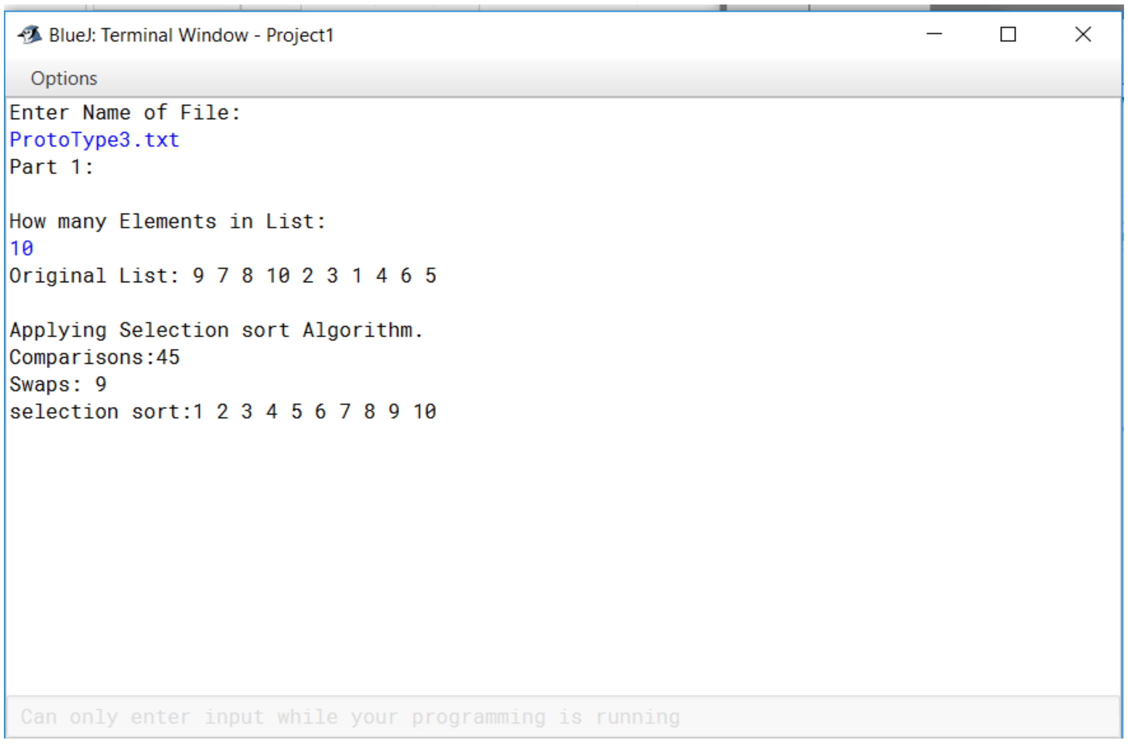

BlueJ: Terminal Window - Project1
Options

```
Enter Name of File:
ProtoType2.txt
Part 1:

How many Elements in List:
10
Original List: 10 9 8 7 6 5 4 3 2 1

Applying Insertion sort Algorithm.
Comparisons:45
Swaps: 45
Insertion sort:1 2 3 4 5 6 7 8 9 10
```

Can only enter input while your programming is running

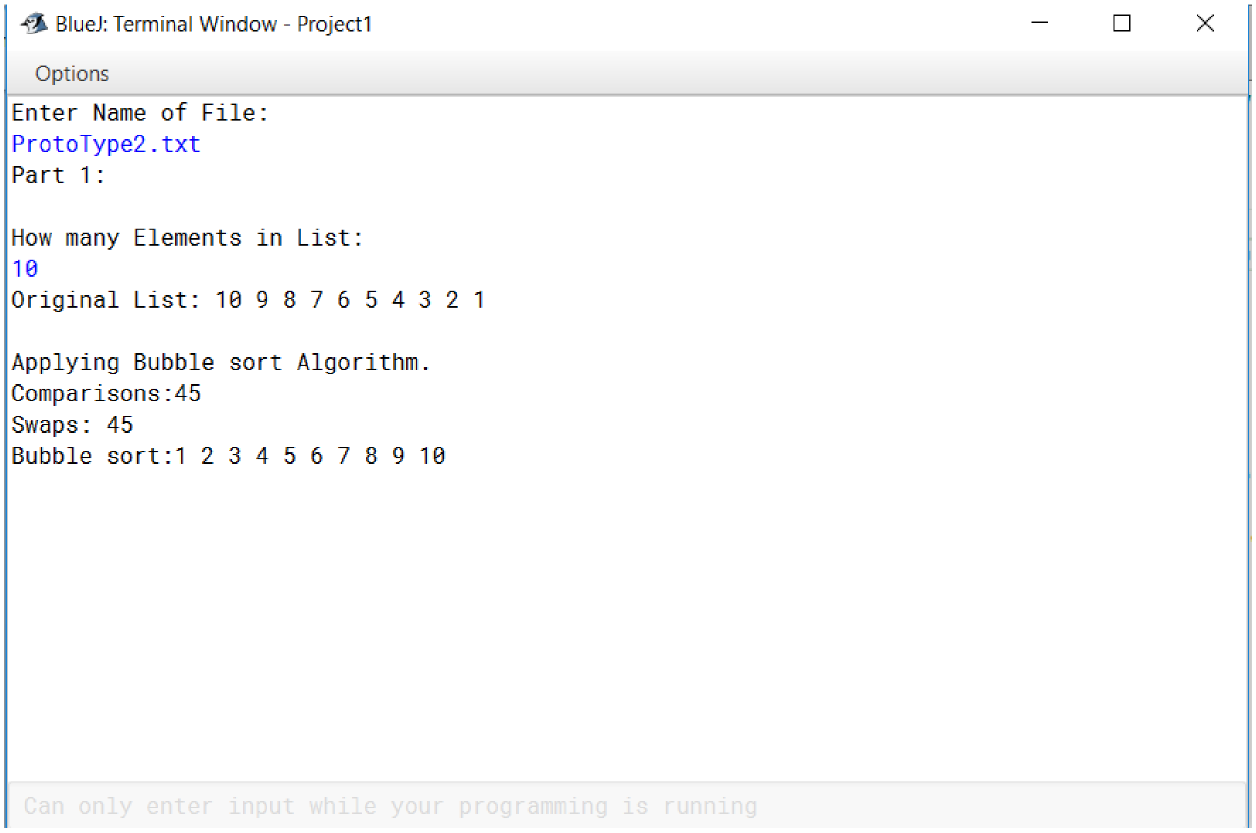

BlueJ: Terminal Window - Project1
Options

```
Enter Name of File:
ProtoType3.txt
Part 1:

How many Elements in List:
10
Original List: 9 7 8 10 2 3 1 4 6 5

Applying Selection sort Algorithm.
Comparisons:45
Swaps: 9
selection sort:1 2 3 4 5 6 7 8 9 10
```

Can only enter input while your programming is running


BlueJ: Terminal Window - Project1
Options

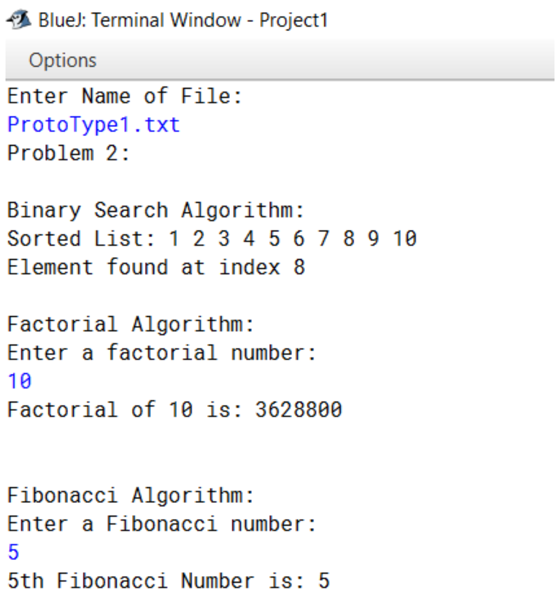
```
Enter Name of File:
ProtoType2.txt
Part 1:

How many Elements in List:
10
Original List: 10 9 8 7 6 5 4 3 2 1

Applying Bubble sort Algorithm.
Comparisons:45
Swaps: 45
Bubble sort:1 2 3 4 5 6 7 8 9 10
```

Can only enter input while your programming is running

Problem 2 program Output:



```
Enter Name of File:
ProtoType1.txt
Problem 2:

Binary Search Algorithm:
Sorted List: 1 2 3 4 5 6 7 8 9 10
Element found at index 8

Factorial Algorithm:
Enter a factorial number:
10
Factorial of 10 is: 3628800


Fibonacci Algorithm:
Enter a Fibonacci number:
5
5th Fibonacci Number is: 5
```