Algorithms and Logical Methods Assignment Report

Name: Weichen Wang

ID: 18232813

Question 1:

What is Merge Sort:

Merge sort(also commonly spelled mergesort) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output.

Pseudocode:

Algorithms merge(list[], temp[], leftPos, rightPos, rightEnd) is:

```
BEGIN
        leftEnd := rightPos - 1 // The left end position. Suppose the left and right columns are next
                                   to each other
        temPos := leftPos      // Cache the index of the first element of the left array
        numElement := rightEnd – leftPos + 1;

        WHILE leftPos <= leftEnd & rightPos <= rightEnd
                IF list[leftPos] > list[rightPos] THEN
                        temp[tempPos] := list[leftPos]
```

```
            leftPos := leftPos + 1
        ELSE
            Temp[tempPos] := list[rightPos]
            rightPos := rightPos + 1
        ENDIF
        tempPos := tempPos + 1
    ENDWHILE

    WHILE leftPos <= leftEnd
        temp[tempPos] := list[leftPos]
        tempPos := tempPos + 1
        leftPos := leftPos + 1
    ENDWHILE

    WHILE rightPos <= rightEnd
        temp[tempPos] := list[rightPos]
        tempPos := tempPos + 1
        rightPos := rightPos + 1
    ENDWHILE

    FOR I = 0 to numElments -1, rightEnd = rightEnd – 1
        List[rightEnd] := temp[rightEnd]
    ENDFOR
END
```

Algorithms sort(list[], temp[], leftPos, rightPos, rightEnd) is:

```
BEGIN
        IF left < right THEN
            center := (left + right)    // Find the middle point to divide the array into two halves
            sort(list, temp, left, center) // Call sort, recursively on the left array
            sort(list, temp, center + 1, right) // call sort, recursively on the right array
            merge(list, temp, center + 1, right) // merge the two halves sorted
        ENDIF
END
```

Algorithms mergeSort(list[]) is:

```
BEGIN
        Sort(list, temp, 0, list.lenght -1) // call sort, perform a merge sort on the data in list
END
```

Time Complexity

Time complexity can be expressed as following recurrence relation

$$T(n) = 2T(n/2) + \Theta(n)$$

Time complexity of Merge Sort is $\Theta(nLogn)$ in all 3 cases(worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer

Question 2

What is Quick Sort:

Quick Sort is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order

Pseudocode:

```
Algorithms partition(list[], start, end) is
BEGIN
    partition := list[end] // take the last element as the partition
    left := start             // define the index of left be start
    right := end - 1          // define the index of right be end – 1

    WHILE true
            WHILE left < end & list[left] <= partition
            left := left + 1
            ENDWHILE

            WHILE right >= start & list[right] >= partition
            right := right - 1
```

```
            ENDWHILE

            IF left < right THEN
                    temp := list[left]
                    list[left] := list[right]
                    list[right] := temp
            ELSE
                    temp := list[left]
                    list[left] := list[end]
                    list[end] := temp
                    break
            ENDIF

            IF left = end THEN
                    Break
            ENDIF
    return left // return the partition position
    ENDWHILE
END
```

Algorithms quickSort(list[], start, end) is:

```
BEGIN
        IF start >= end THEN // same point
```

```
        Return
    ELSE
        Middle := partition(list, start, end) // find partition point
        quickSort(list, start, middle - 1) // quickSort for part of left
        quickSort(list, middle + 1, end) // quickSort for part of right
    ENDIF
END
```

Space Complexity

The space used for quick sorting depends on the version used. With a fast-sorted version of the in-place split, only fixed extra space is used before any recursive calls. However, if it is necessary to generate O(log n) nested recursive calls, it needs to store a fixed amount of information on each of them. Since the best case requires up to O(log n) nested recursive calls, it requires O(log n) space. In the worst case, O(n) nested recursive calls are required, so O(n) space is required.

Question 3

What is Fibonacci sequence:

The series are 0, 1, 1, 2, 3, 5, 8..... Barring the first two terms in the sequence every other term is the sum of two previous terms. Take the following example 8 = 3 + 5 (This is the addition of 3 and 5).

Pseudocode:

Algorithms Fibonacci using recursion:

```
BEGIN
        READ position
        IF position = 0 OR position = 1 THEN
                fibanacci(position) = position
        ENDIF
      fibanacci(position) = fibanacci (position – 1) + fibanacci (position - 2)
END
```

Algorithms Fibonacci without using recursion:

```
BEGIN
        READ position
        IF position = 0 OR position = 1 THEN
                fibanacci(position) = position
        ENDIF
```

```
	READ x = 0, y = 1, sum = 0
	FOR n = 2 to position
		sum = x + y
		x = y
		y = sum
	ENDFOR
	return sum
END
```

For Algorithms Fibonacci using recursion:

1. Time Complexity Analysis (Big O notation)

We use the solution fibanacci(10) as an example to analyze the process of recursive solution. To obtain fibanacci(10), fibanacci(9) and fibanacci(8) are required. Similarly, to obtain fibanacci(9), we must first obtain fibanacci(8) and fibanacci(7)... We use a tree structure to represent this dependency.

```
                 f(10)
               /       \
          f(9)          f(8)
         /    \        /    \
     f(8)      f(7)  f(7)    f(6)
    /   \      /   \
f(7)  f(6)  f(6) f(5)
```

There are many nodes in this tree that will repeat, and the number of repeated nodes will increase sharply as n increases. This means that this amount of calculation will increase sharply as n increases. In fact, the time complexity calculated by the recursive method is incremented by the exponent of n, and the time complexity is approximately equal to O(2^n).

2. Space Complexity Analysis

The space occupied by each layer operation is 1, and it is recursively n-1 times. that is, it is approximately equal to n, the space complexity is O(n).

For Algorithms Fibonacci without using recursion:

1. Avoid the operation of repeated nodes, the computational time complexity of this idea is O(n).
2. Space Complexity O(1) is approximately equal to O(1).

Question 4

$$expBySquaring(x,n) = \begin{cases} expBySquaring(1/x, -n) & if\ \ n < 0 \\ 1 & if\ \ n = 0 \\ x & if\ \ n = 1 \\ expBySquaring(x * x, n/2) & if\ \ n\ is\ even \\ x * expBySquaring(x * x, (n-1)/2) & if\ \ n\ is\ odd \end{cases}$$

What is Exponentiation by squaring:

Exponentiation by squaring is an algorithm. It is used for quick working out large integer powers of a number. It is also known as the square-and-multiply algorithm or binary exponentiation. It uses the binary expansion of the exponent. It is of quite general use, for example in modular arithmetic.

Structure:

- Input x and n
- If n is Less than 0
  return expBySquaring(1/x, -n)
- Else if n equal to 0
  return 1
- Else if n equal to 1
  Re return turn x
- Else if n mod 2 = 0 (if n is even)
  return expBySquaring(x * x, n / 2)
- Else (if n is odd)

return x multiply by expBySquaring(x * x, (n-1) / 2)

- Print expBySquaring(x, n)

Pseudocode:

Algorithm expBySquaring(x , n):

```
BEGIN
        READ x, n
        IF n < 0 THEN
                expBySquaring(x, n) = expBySquaring(1/x, -n)
        ELSE IF n = 0 THEN
                expBySquaring(x, n) = 1
        ELSE IF n = 1 THEN
                expBySquaring(x, n) = x
        ELSE IF n % 2 = 0 THEN
                expBySquaring(x, n) = expBySquaring(x * x, n / 2)
        ELSE
                expBySquaring(x, n) = x * expBySquaring(x * x, (n-1) / 2)
        ENDIF
        WRITE expBySquaring(x, n)
END
```

References:

- https://en.wikipedia.org/wiki/Merge_sort
- https://simple.wikipedia.org/wiki/Exponentiation_by_squaring
- https://www.jianshu.com/p/5ed59728e1ed
- https://en.wikipedia.org/wiki/Quicksort
- https://www.geeksforgeeks.org/merge-sort/
- Lecture Notes – Enda Howley - Topic 5L big_o.pdf (Big O nation)