DSA555 Lab 2 Analysis

Factorial Function (Recursive version):

Step 0 - Code Snippet:

```
//accepts an integer, recursively calculates the factorial of the integer and returns the result
unsigned int factorial (unsigned int n){
    int result = 1; // 1 operation (factorial result)

    if (n > 1) // 1 operation (checks value of 'n' to begin recursive calculation)
    {
        // recursively multiplies 'n' by returned result from the function call of 1 less than 'n'
        // and stores the updated value in 'result'
        result = n * factorial(n - 1); // 3 operations (= , * , -)
    }

    return result; // 1 operation
}
```

Step 1 - Variable and Function Declarations:

- Let **n** represent the value to find its factorial
- Let **T(n)** represent number of operations needed to find the factorial of **n** using the recursive factorial function

Step 2 – Count number of operations:

- `int result = 1;` → 1
- `n > 1` → 1
- `result = n * factorial(n - 1);` → 3+T(n-1)
    - 3 → =, *, -
    - T(n-1) → factorial(n-1)
- `return result;` → 1

Step 3 – Write expression for T(n):

For n > 1:

$$T(n) = 1 + 1 + 3 + T(n-1) + 1$$

$$= 6 + T(n-1)$$

For n <= 1:

$T(n) = 1 + 1 + 1 = 3$

Step 4 – Simplify:

$T(n) = 6 + T(n-1)$ → 6+6+…+6+3

$= 6(n-1) + 3$

$= 6n - 6 + 3$

$= 6n - 3$

Step 5 – State final result:

Therefore, T(n) **is** O(n)

- Constants 6 and -3 are dropped (insignificant as **n** grows larger)

Power Function (Recursive version):

Step 0 – Code Snippet:

```
//accepts a double and an integer
/*recursively calculates and returns the base raised to an exponent and decreases the exponent
until it reaches 0*/
double power (double base, unsigned int n)
{
    if (n == 0) //1 (==)
    {
        return 1; // 1 (return)
    }
    else
    {
        //returns value from recursive calculation of base multiplied by value returned from power
        //function of 1 less than the current exponent
        return base * power(base, n - 1); // 3+T(logn-1) (return, *, -)
    }
}
```

Step 1 – Variable and Function Declarations:

- Let **n** represent the exponent in which the base will multiply itself **n** times
- Let **T(n)** represent the number of operations needed to recursively calculate the **n**th exponent of the base

Step 2 – Count number of operations:

Operations done once:

- `n == 0` → 1 operation
- `return 1;` → 1 operation

Operations done recursively:

- `return base * power(base, n - 1)` → 3 operations
  - `return` → 1
  - `base * power(base, n - 1)` → 1
  - `n - 1` → 1

Step 3 – Write expression for T(n):

Logarithms review:

$\log_{10} n$ → mathematical version of logarithms with base of 10

$\log_2 n$ → computer science version of logarithms with base of 2 (binary)

$b^n = a \rightarrow \log_b a = n$ → $\log_b a$ is the exponent the base (b) needs to be raised to get n

Examples:

$2^{n-1} = a$

$2^0 = 1$

$2^1 \rightarrow 2^{1-1} \rightarrow 2^0 * 2 = 2$

$2^2 \rightarrow 2^{2-1} \rightarrow 2^{1-1} * 2 = 4$

$2^3 \rightarrow 2^{3-1} \rightarrow 2^{2-1} * 2^{1-1} * 2 = 8$

$2^4 \rightarrow 2^{4-1} \rightarrow 2^{3-1} * 2^{2-1} * 2^{1-1} * 2 = 16$

$(\log_2 1 - 1) * 2 = 2$

$(\log_2 2 - 1) * 2 = 4$

$$T(n) = 1 + 1 + 3 + T(\log_2 n - 1)$$

Step 4 – Simplify:

$$T(n) = 1 + 1 + 3 + T(\log_2 n - 1)$$

For **n > 0:**

$= 5 + T(\log_2 n - 1)$

$= 5(\log_2 n - 1)$

$= 5 \log_2 n - 5$

For **n <= 0:**

$T(n) = 1 + 1$

$= 2$

Step 5 – State final result:

Therefore, $T(n)$ **is** $O(\log_2 n)$

- $5 \log_2 n - 5$
  - $5 \log_2 n$ is dominating term
  - Constants 5 and 5 are dropped

Fibonacci Function Reflection (Recursive):

Note: No analysis performed on this function

1) Did you find it easier to write the recursive fibonacci() function or the iterative version?
   The Recursive version.

2)
   a. What do you think the run time of your recursive fibonacci function is (stated with big-O notation)?
      O(2^n)

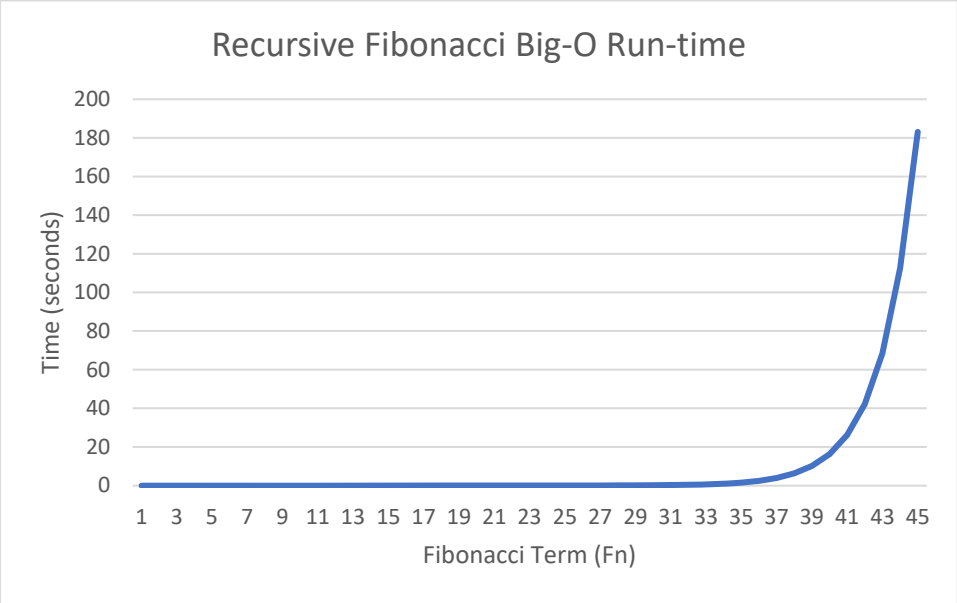
   b. Explain why you think this.
      Every time a new term in the sequence must be calculated, the function must recursively call itself twice to recalculate all previous terms. reach the previous two terms with respect to the new term and add them together. The recalculation of previous terms through recursive calls exponentially builds up the running time until the calculation for the new term is reached.

c. Modify the file lab2timing.cpp to get a timing of running the fibonacci function. Record the time needed to run fib for n ranging from 21 to 45 inclusive (values smaller than 21 are pretty small but feel free to run and include them if you want).

d. Take the data you generated and create a line graph using a spreadsheet program such as google spreadsheet or excel.

Line graph organized as follows:

i. x axis represents n
ii. y axis represents time
iii. provide clear labeling and titles
iv. place the image of the graph into your wiki page as part of your answer.



e. Given the timing, and your original guess, does the run time fit your original hypothesis?
Yes