DSA555 Lab 1 Analysis

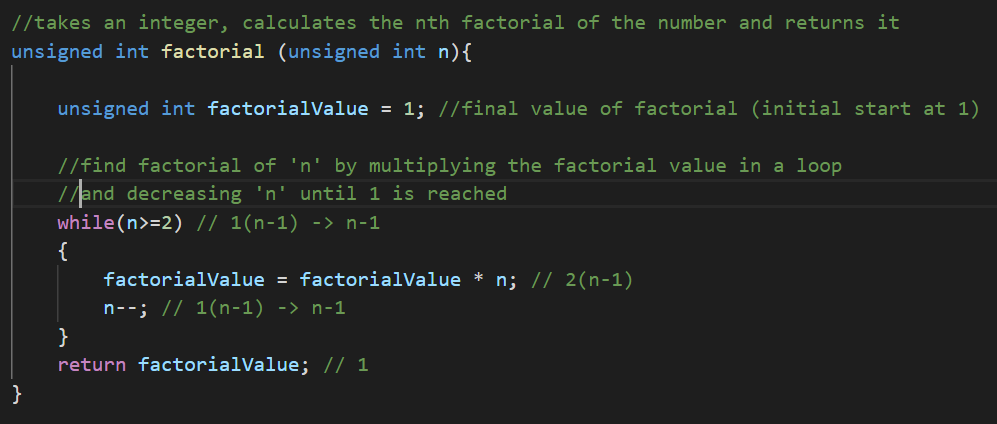
**Factorial Function:**

Factorial formula:

n! = n \* (n-1) \* (n-2) \* ... \* 3 \* 2 \* 1

Example: 4! = 4 \* 3 \* 2 \* 1 = 24

Step 0 - Code snippet:



Step 1 – Variable and Function Declarations:

* Let **n** represent the value of the factorial to find
* Let **T(n)** represent the number of operations performed to find the factorial of **n**

Step 2 – Count number of operations:

* Operations done once:
  + unsigned int factorialValue **= 1**; 🡪 1 operation
  + return factorialValue; 🡪 1 operation
* Operations done every loop iteration:
  + n**>=**2 🡪 1 operation
  + factorialValue = factorialValue \* n; 🡪 2 operations
    - factorialValue **=** factorialValue 🡪 1
    - factorialValue **\*** n 🡪 1
  + n**--**; 🡪 1 operation

Step 3 – Write expression for T(n)

* Number of operations done once:
  + 2
* Number of times loop runs:
* Number of operations done every loop:
  + 4
* Total number of operations:

Step 4 – Simplify:

First way:

Second way:

Step 5 – State final result:

Therefore, **is**

* 🡪 🡪 🡪
  + is dominating term
  + Constants (4 and -2) are dropped (they are insignificant as **n** becomes larger)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Power Function:**

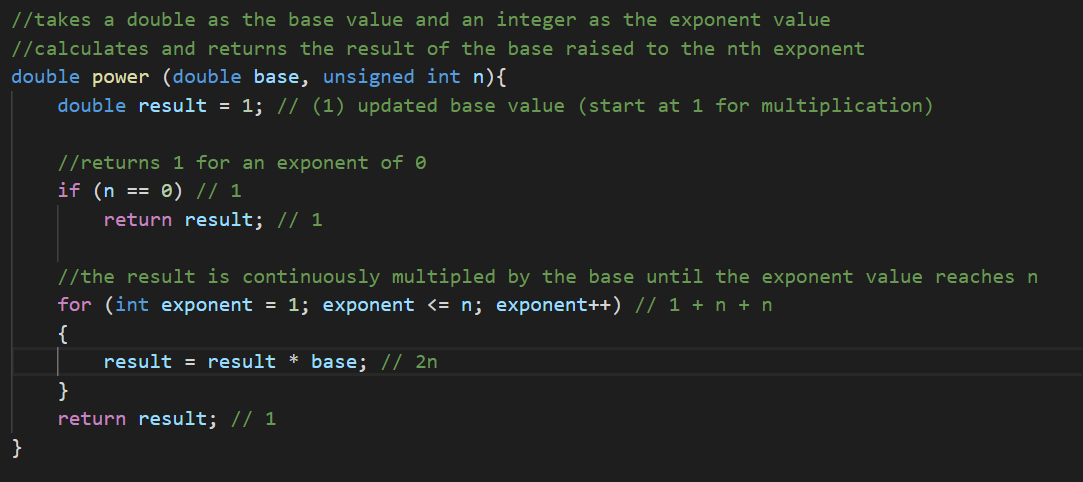
Formula:

🡪 b=base; n= number of times the base multiplies itself (exponent)

Example:

🡪 2\*2\*2\*2 = 16

Step 0 – Code snippet:



Step 1 – Variable and Function Declarations:

* Let **n** represent the exponent the base will multiply itself by
* Let **T(n)** represent the number of operations performed to get the result of the exponent

Step 2 – Count number of operations:

* Operations done once:
  + double result **=** 1;
  + n **==** 0
  + return result; (x2)
  + int exponent **=** 1;
* Operations done every loop run:
  + exponent **<=** n 🡪 1 operation
  + exponent**++** 🡪 1 operation
  + result = result \* base; 🡪 2 operations
    - result **=** result; 🡪 1 operation
    - result **\*** base; 🡪 1 operation

Step 3 – Write expression for T(n):

* + 4 🡪 operations done every loop
  + n 🡪 number of times loop runs
  + 5 🡪 operations done once (including return for exponent 0)

Step 4 – Simplify:

First way:

Second way:

Step 5 – state final result:

Therefore, **is**

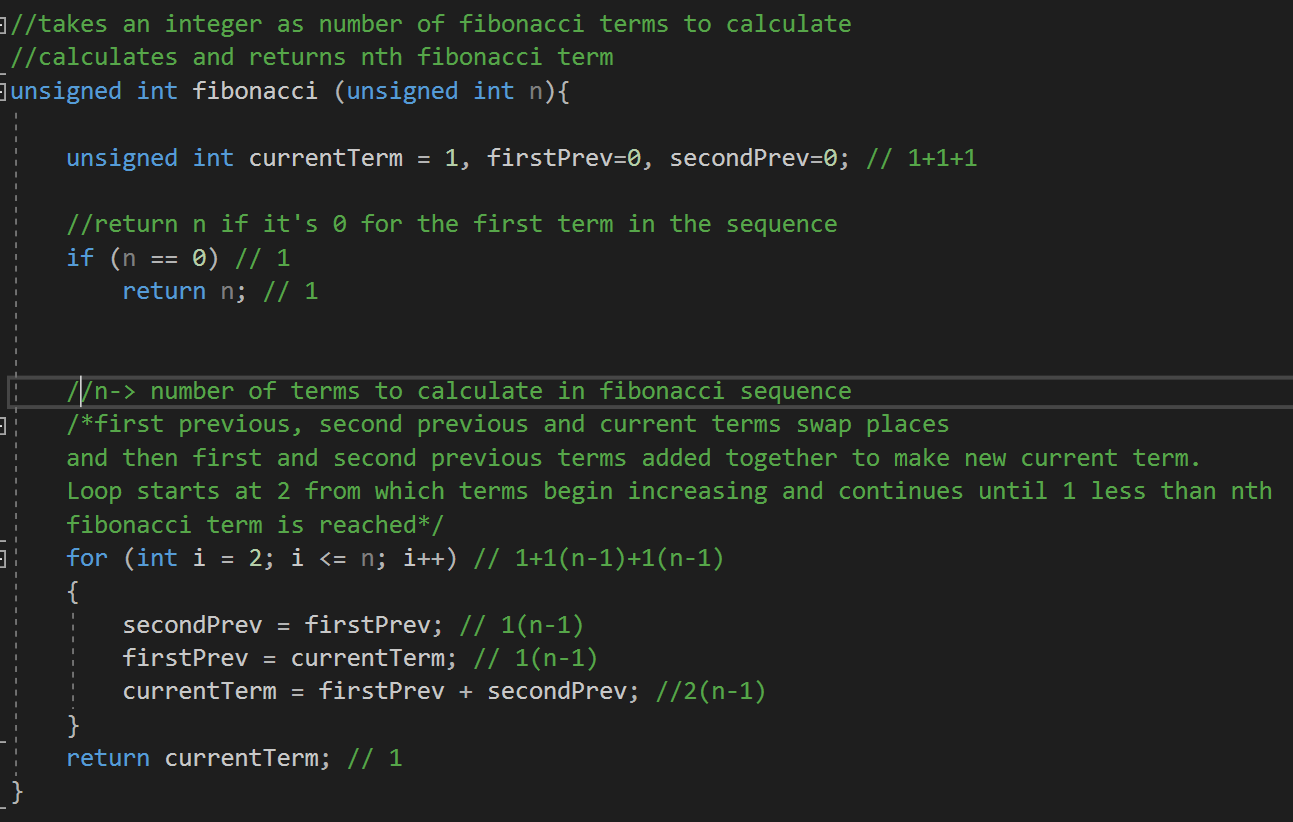
* 🡪 🡪 🡪
  + 4n is dominating term
  + Constants (4 and 5) are dropped

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Fibonacci Function**:

Formula:

Example:

Step 0 – Code Snippet:

Step 1 – Variable and Function Declarations:

* Let **n** represent the number of terms to calculate for the Fibonacci sequence
* Let **T(n)** represent the number of operations to perform to calculate up to **n**th term

Step 2 – Count number of operations:

* Operations performed once:
  + unsigned int currentTerm = 1
  + firstPrev=0
  + secondPrev=0
  + n == 0
  + return n
  + int i = 2
  + return currentTerm;
* Operations performed each loop:
  + i **<=** n 🡪 1
  + i++ 🡪 1
  + secondPrev = firstPrev; 🡪 1
  + firstPrev = currentTerm; 🡪 1
  + currentTerm = firstPrev + secondPrev; 🡪 2
    - currentTerm = firstPrev 🡪 1
    - firstPrev + secondPrev 🡪 1

Step 3 – Write expression for T(n)

* + 6 🡪 number of operations done in each loop
  + n-1 🡪 number of times loop runs
  + 7 🡪 number of operations done once

Step 4 – Simplify:

First way:

Second way:

Step 5 – state final result:

Therefore, **is**

* 🡪 🡪 🡪
* 6n is dominating term
* Constants (6 and 1) are dropped