WOLS1E

September 24, 2018

1 Time Complexity of Fibonacci Algorithms

1.1 *Cost and Time Complexity of Recursive Algorithm*:

```
                                             Cost  Time
def rFib(num):
    sum = 0                                  C1      1
    if num < 2:                              C2      1
        return num
    else:
        sum = rFib(num - 2) + rFib(num - 1)  C3      T(n) = T(n - 1) + T(n - 2) + C
    return sum
```

1.1.1 *Equation 1:*

Substitution Method:

```
    T(0) = 1
    T(1) = 1
Assumption:
    T(n - 1) = T(n - 2)

Therefore:
    Upper Bound:
        T(n) = 2T(n-1) + C1
             = 4T(n-2) + C2
             = 8T(n-3) + C3
             = 16T(n-4) + C4
             = 2^k(n-k) + C5

    In terms of T(0):
        n - k = 0 -> n = k
        T(n) = 2^n * T(0) + C
```

$\longrightarrow$ **T(n) = O(2^n)**

1.1.2 *Cost and Time Complexity of Iterative Algorithm*:

```
                                Cost  Time
def fib(num):
    f = [0, 1]                   C1     1
    for i in range(1, num):      C2     n+1
        sum = f[i - 1] + f[i]    C3     n
        f.append(sum)            C4     n
    return f[num]
```

1.1.3 *Equation 2:*

```
T(n) = C1 + C2(n+1) + C3(n) + C4(n)
```

$\longrightarrow$ **T(n) = O(n)**

2 *Graphs of Recursive and Iterative functions:*

```
In [13]: fibNumber1 = []
         executionTime1 = []
         for i in range(30):
             fibNumber1.append(i)
             executionTime1.append(timeit.timeit(
                 stmt=f"rFib({i})", setup=rFibS, number=10))
```

Figure 1

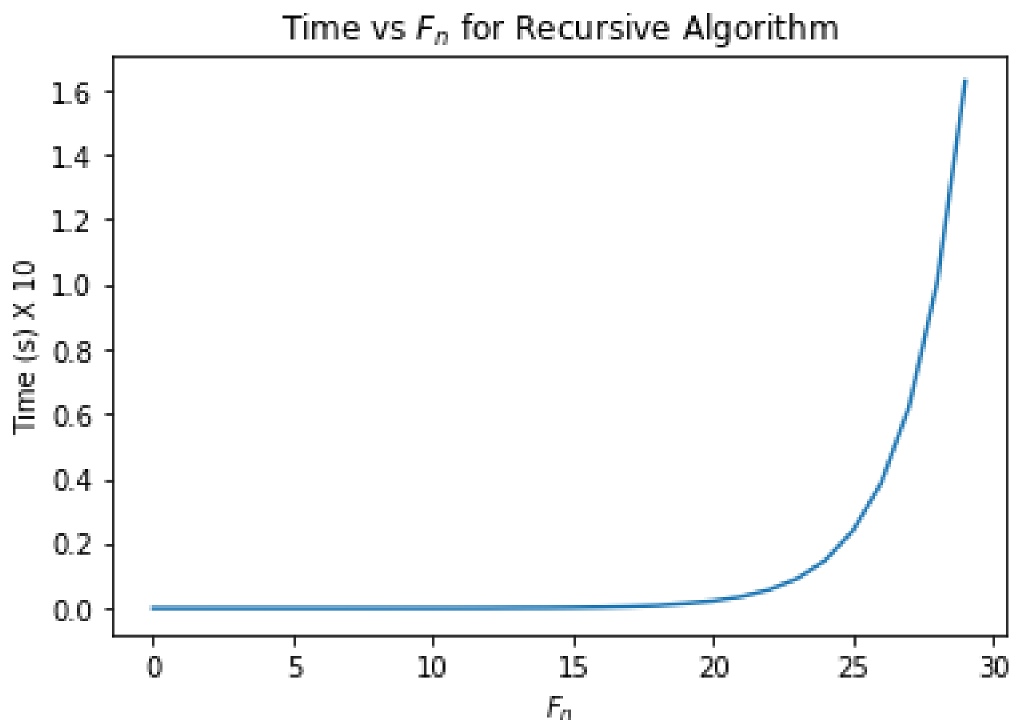

figure 1: Growth of recursive algorithm = $O(2^n)$.

```
In [14]: fibNumber2 = []
         executionTime2 = []
         for i in range(30):
             fibNumber2.append(i)
             executionTime2.append(timeit.timeit(
                 stmt=f"fib({i})", setup=fibS, number=10))
```

Figure 2

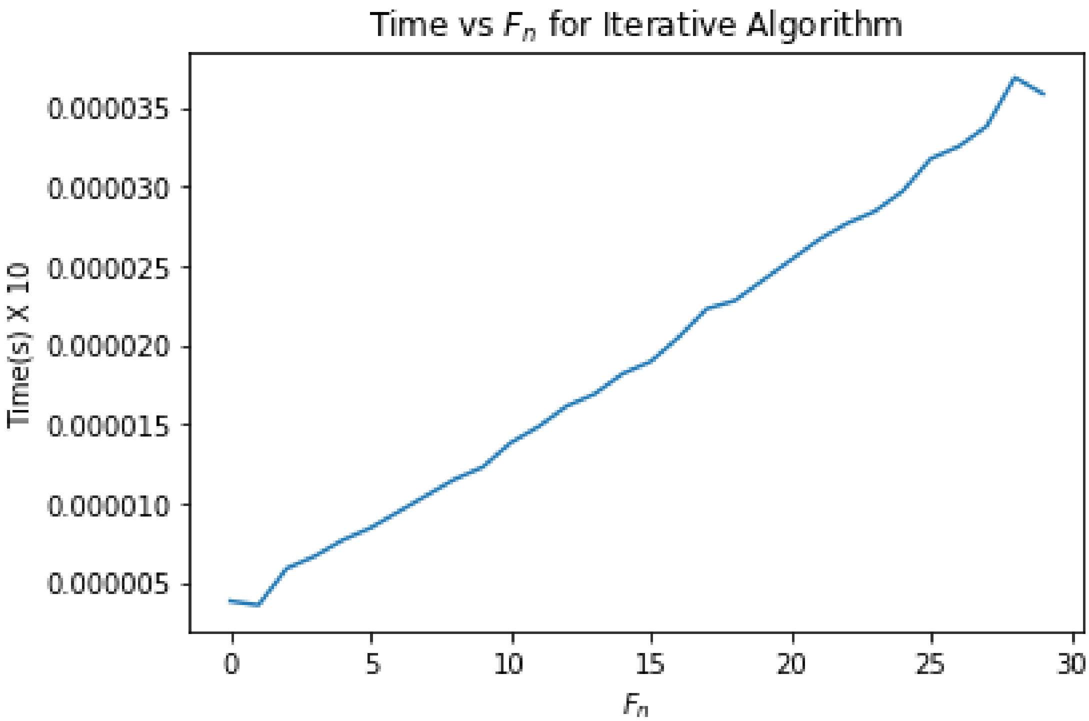

figure 2: Growth of iterative algorithm = O(n).

```python
In [15]: from matplotlib import pyplot as plt
         import timeit

         plt.plot(fibNumber1, executionTime1, label="Recursive")
         plt.plot(fibNumber2, executionTime2, label="Iterative")
```

Figure 3

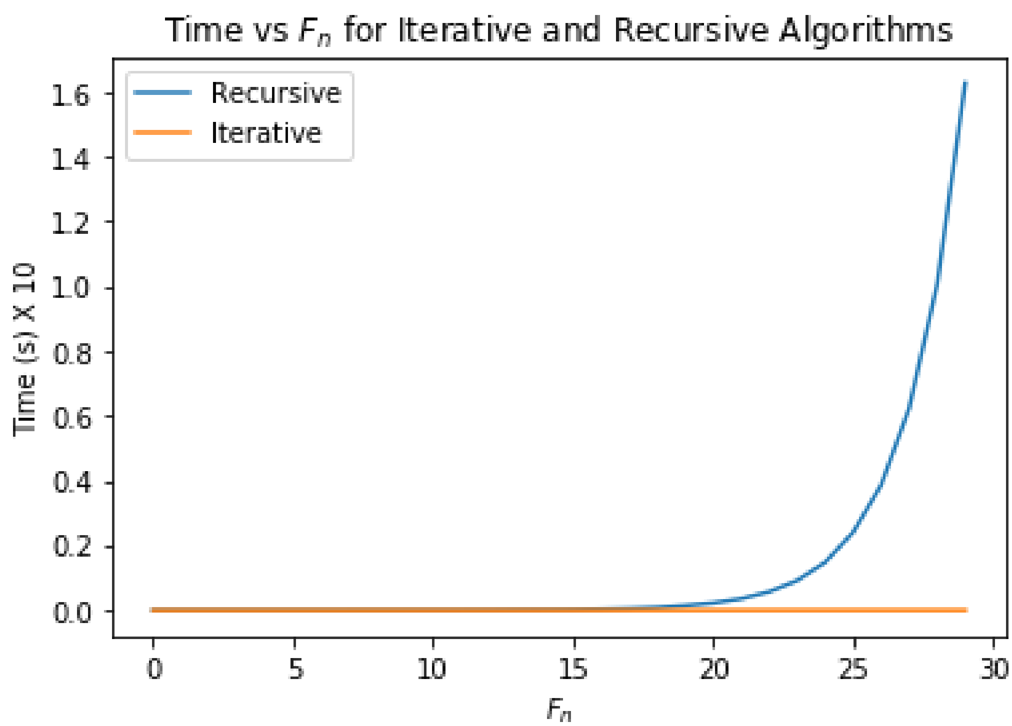

figure 3: Growth of recursive and interative algorithms.

2.0.1 *Percent Difference Between Algorithms:*

```
In [16]: def compare(num):
             iterative = timeit.timeit(stmt=f"fib({num})", setup=fibS, number=10)

             recursive = timeit.timeit(stmt=f"rFib({num})", setup=rFibS, number=10)

             percentDiff = (abs(recursive-iterative)/((recursive+iterative)/2))*100

             if iterative <= recursive:
                 return percentDiff
             if recursive < iterative:
                 return -percentDiff
```

```
In [17]: fibNumbers = []
         percentDiffs = []
         for i in range(30):
             fibNumbers.append(i)
             percentDiffs.append(compare(i))
```

Figure 4

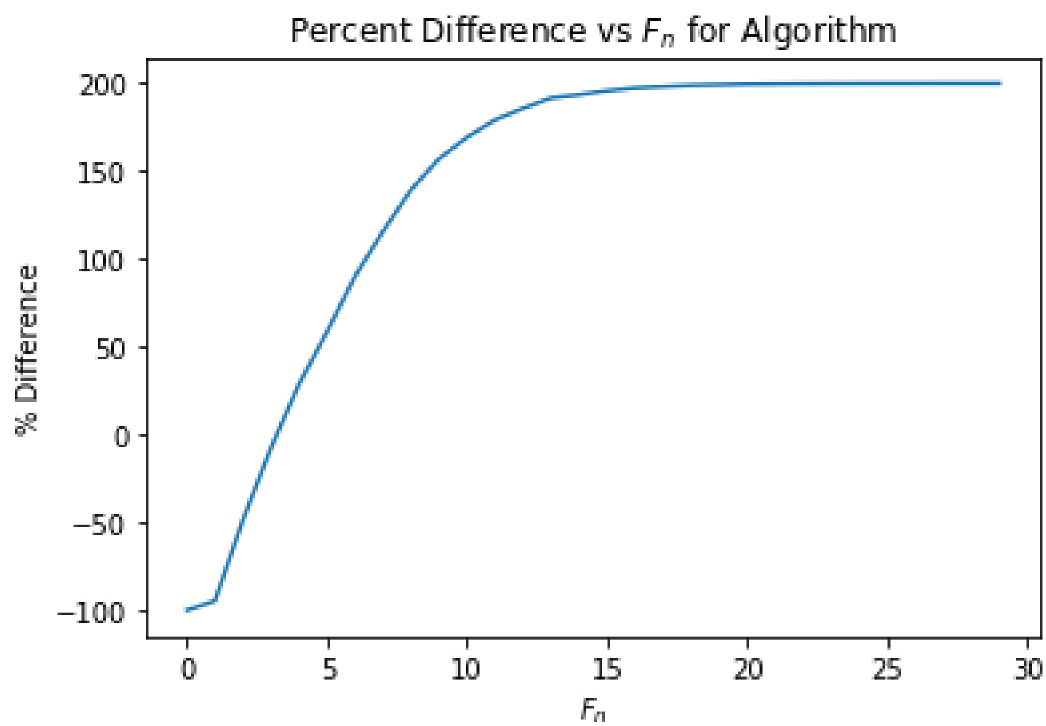

figure 4: Percent difference between recursive and iterative algorithms.

2.0.2 *Discussion:*

Utilizing the substitution method we were able to arrive at an upper bound of O(2^n) for the recursive algorithm in equation 1. With a known initial condition for T(0) and by making the assumption that T(n-2) $\approx$ T(n-1) we were able to use mathematical induction to arrive at a conclusion. The time complexity for the iterative algorithm is a more straightforward deduction and is shown to be O(n) in equation 2. These equations are corroborated by the run-time analysis demonstrated in figures 1-3. Figure 1 demonstrates the exponential nature of the recursive algorithm as F_n gets larger. Figure 2 shows a linear time complexity for the iterative algorithm, and figure 3 plots both run-time analyses on the same axes giving a visual representation of how their growth differs. Figure 4 demonstrates a comparison between the iterative and recursive functions by plotting their percent difference ($\frac{\Delta V}{\frac{\sum V}{2}}$X100) as F_n increases. We can see for small F_n the two algorithms are equal or recusrive may be more efficient. However, as F_n increases we can see that the run-time of the recursive algorithm dominates and the percent difference calulation becomes $\frac{1}{1/2}$X100.