COMPSCI 308: Design and Analysis of Algorithms Homework 2

Luyao Wang

April 1, 2024

1. Divide and Conquer

(a) Programming

1. Iterative Fibonacci number computation

```
def fibonacci_iterative(n: int) -> int:
    if n < 0:
        raise Exception("n should be larger than or equal to 0")
    elif n == 0:
        return 0
    elif n == 1:
        return 1

    fib_minus_2 = 0
    fib_minus_1 = 1
    fib = 0

    for i in range(3, n + 1):
        fib = fib_minus_2 + fib_minus_1
        fib_minus_2 = fib_minus_1
        fib_minus_1 = fib

    return fib
```

The time complexity is $O(n)$ because we use one for loop with i from 3 to n, inside the for loop, each execution is constant time.

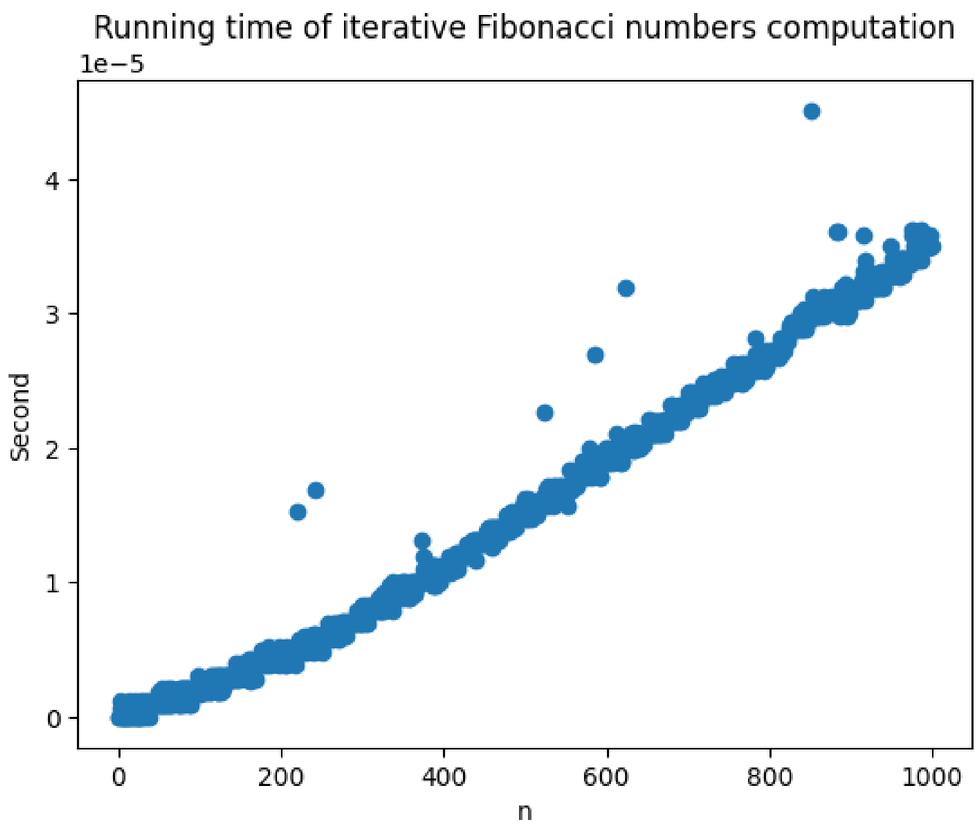


Figure 1: Running time of iterative Fibonacci numbers computation

2. Recursive Fibonacci number computation

```
def fibonacci_recursive(n: int) -> int:
    if n < 0:
        raise Exception("n should be larger than or equal to 0")
    elif n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

The time of computing Fibonacci recursively is $T(n) = T(n-1) + T(n-2) + O(1)$.

Fibonacci can be mathematically represented as a linear recursive function $F(n) = F(n-1) + F(n-2)$.

The characteristic equation for this function will be $x^2 = x + 1$. Solving this by quadratic formula we can get the roots as $x = \frac{1+\sqrt{5}}{2}$ and $x = \frac{1-\sqrt{5}}{2}$.

For the Fibonacci function $F(n) = F(n-1) + F(n-2)$ the solution will be:

$$F(n) = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$$

$T(n)$ and $F(n)$ are asymptotically the same as both functions are representing the same thing.

$$T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

$$T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

From the experiment results in the table, we can see that the ratio between two consecutive ns is close to $\frac{1+\sqrt{5}}{2}$, or the golden ratio.

Table 1: Running time for different n

n	**seconds**
35	1.12
36	1.82
37	2.95
38	4.78
39	7.76
40	12.56

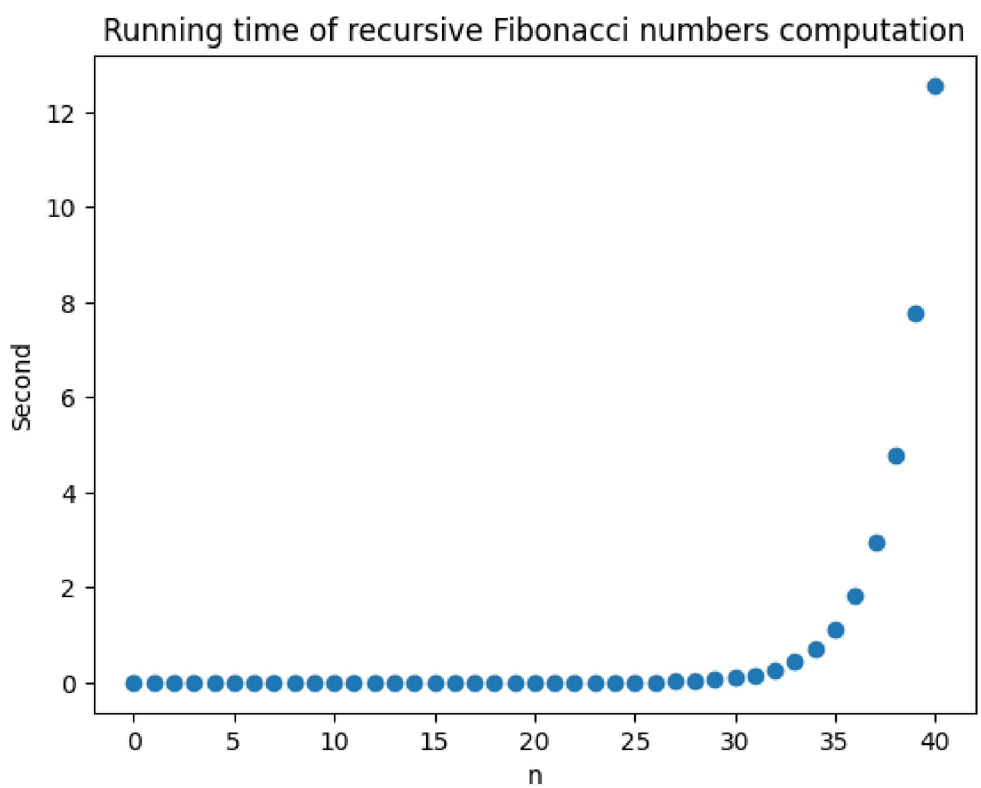
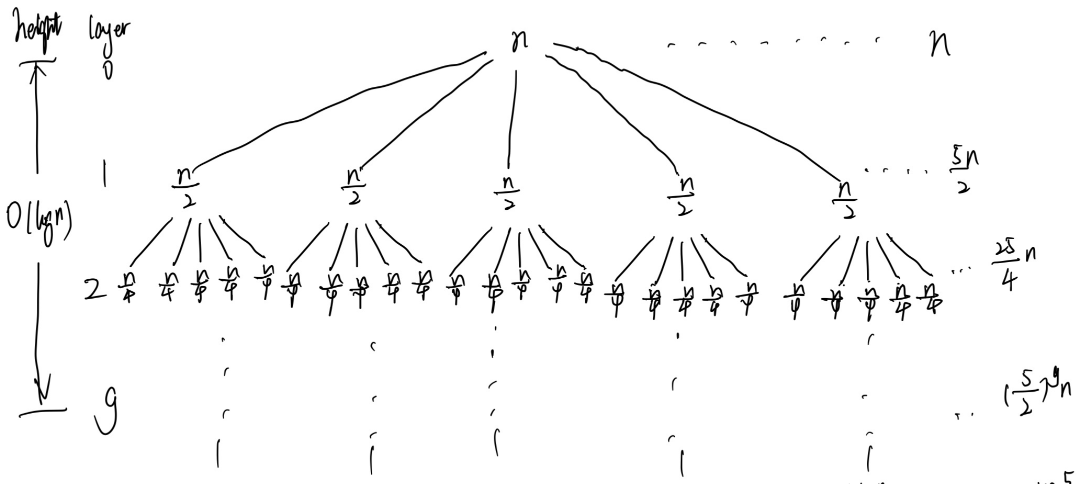


Figure 2: Running time of recursive Fibonacci numbers computation

(b) Recursion Tree

Recurrence equation representing the algorithm:

$$T(n) = 5T(\frac{n}{2}) + n$$



$r = \frac{5}{2} > 1$, $T(n) = (1 + r + r^2 + r^3 + \cdots + r^{\log_2 n}) n = \frac{r^{1+\log_2 n} - 1}{r-1} n = \frac{5}{3} n^{\log_2 5} - \frac{2}{3} n$

$= O(n^{\log_2 5})$

(c) Asymptotic Analysis

1. First algorithm

$$T(n) = 4T(n-1) + 1$$

Solve by subtituiton:

$$\begin{aligned}
T(n) &= 4T(n-1) + 1 \\
&= 4(4T(n-2) + 1) + 1 \\
&= 4 \cdot 4T(n-2) + 1 + 4 \\
&= 4 \cdot 4(4T(n-3) + 1) + 1 + 4 \\
&= 4 \cdot 4 \cdot 4T(n-3) + 1 + 4 + 16 \\
&= \cdots \\
&= 4^i T(n-i) + \sum_{t=0}^{i-1} 4^t \\
&= 4^i T(n-i) + \frac{4^i - 1}{3}
\end{aligned}$$

Base case $i = n - 1$, $T(1) = 1$.

Therefore,

$$\begin{aligned} T(n) &= 4^i T(n-i) + \frac{4^i - 1}{3} \\ &= 4^{n-1} + \frac{4^{n-1} - 1}{3} \\ &= \frac{4^n - 1}{3} \\ &= \Theta(n^4) \end{aligned}$$

2. Second algorithm

$$T(n) = 3T(\frac{n}{3}) + n^5$$

By Master Theorem, $a = 3$, $b = 3$, $n^{\log_b a} = n$ is polynomially smaller than $n5$.

Therefore, case 3 of Master Theorem fits, $T(n) = \Theta(n^5)$

3. Conclusion

Considering the asymptotic behaviors of the two algorithms, the first is preferable.

2. Dynamic Programming

(a) Pseudocode

Algorithm 1 Dynamic Programming Algorithm for Maximum Success Score

```
function MaxSuccessScore(i, j, memo)
    if memo[i][j] is defined then
        return memo[i][j]
    end if
    maxScore ← 0
    for k = i to j do
        score ← cheatIndex[k]
        if k > i then
            score× = cheatIndex[k − 1]
        end if
        if k < j then
            score× = cheatIndex[k + 1]
        end if
        if k > i then
            score+ = MaxSuccessScore(i, k − 1, memo)
        end if
        if k < j then
            score+ = MaxSuccessScore(k + 1, j, memo)
        end if
        maxScore ← max(maxScore, score)
    end for
    memo[i][j] ← maxScore
    return maxScore
end function
function CalculateMaxSuccessScore(cheatIndex)
    n ← length(cheatIndex)
    memo ← new DiagonalMatrix(n, n)
    return MaxSuccessScore(1, n, memo)
end function
```

(b) Asymptotic Analysis

In this algorithm, the number of subproblems is determined by the range of indices i and j. Since each subproblem corresponds to a specific range, there are a total of $O(n^2)$ possible subproblems, where n is the length of the `cheatIndex` array.

For each subproblem, the algorithm performs a loop from i to j, resulting in a linear time complexity of $O(n)$ for each subproblem. Within this loop, the algorithm performs constant-time operations such as multiplication, addition, and memoization.

More formally,

$$
\begin{aligned}
T(n) &= \sum_{k=1}^{n}(n+1-k)k \\
&= \sum_{k=1}^{n}(n+1)k - k^2 \\
&= n\frac{n+1+n^2+n}{2} - \frac{n(n+1)(2n+1)}{6} \\
&= \frac{n^3+3n^2+2n}{6}
\end{aligned}
$$

Therefore, the overall time complexity of the algorithm is $O(n^3)$.