Algorithms Analysis and Design

Week 2 - Diary

Ayan Agrawal (2020101034)

Lecture 3 : Fibonacci

1. Introduction

- For sometimes, we will be solving Tractable problems now and will return to intractable/ hard problems later on.
- Let's discuss "What is Tractable algorithm?" first of all.
    - Any algorithm that takes exponential amount of time or more is called as Intractable algorithm.

      $$T(n) \geq O(p^n), \text{ where } p \in \mathbb{R} \text{ and } p > 1$$

      Also, only 2 types of algorithms are possible,i.e., tractable or intractable. Therefore, any algorithm which isn't intractable is tractable.

      For any algorithm, *its correctness, performance and optimality* is the thing that matters to decide whether it is a desirable solution or not.

Two problems were discussed and solved with various algorithms. One of them was calculating n^{th} Fibonacci Number and Multiplying of Large Integers.

2. Problem 1 - n^{th} Fibonacci number

- We will gradually find better and more optimal solutions for computing n^{th} Fibonacci number.

$$F_n = F_{n-1} + F_{n-2} \text{ where } n > 1,$$
$$F_1 = 1 \text{ and } F_0 = 0$$

1. Algorithm 1 : Naive Solution

```
function fibonacci(n):
    if n = 0: return 0
    else if n = 1 : return 1
    else (return fibonacci(n-1) + fibonacci(n-2))
```

This algorithm is the direct implementation of the definition of the Fibonacci numbers. Here,

$$T(n) = T(n-1) + T(n-2) \text{ where } n > 1,$$
$$T(1) = 2 \text{ and } T(0) = 1$$

We will calculate golden ratio ϕ , which is given by

$$\phi = \frac{1+\sqrt{5}}{2} = \lim_{n \to \infty} \frac{F_{n+1}}{F_n} \approx 1.618 \approx 2^{0.694}$$

By this, we can observe that n^{th} Fibonacci number has approx. $0.694 * n$ bits, and

$$F_n \approx 2^{0.694 * n}$$

Therefore,

$$T(n) \geq F_n$$

Which means ***Naive algorithm is intractable.***

2. Algorithm 2 : Iteration

```
function fibon(n):
    if n <=1: return n
    p = 0, q = 1
    for i in 2...n:
        c = p + q
        a = b
        b = c
    return c
```

- This algorithm is actually of complexity $O(n^2)$ but seems as $O(n)$. This happenned because F_n is $\approx 0.694 * n$ bits long, which adds Addition operation time $O(n)$.

3. Algorithm 3 : Matrix Multiplication

We can write the Fibonacci sequence as

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

$$\implies \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

- Here, we will use binary exponentiation, $O(\log N)$ matrix multiplication would be enough, making the complexity $O(P(n) \log N)$, where $P(n)$ is the time complexity of multiplying 2 n- bit integers since each matrix multiplication involves multiplying 4 integers of at most $O(n)$ bits.
- This algorithm narrows down to $O(n^{1.585} \log n)$ which is better than 2nd algorithm's $O(n)$. $P(n)$ part is $O(n^{1.585})$ using the Karatsuba's algorithm for speeding up multiplication of large integers.

4. Algorithm 4 : Binet's Formula (a.k.a Recurrence formula)

We know,

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

$$F_n = \left[\frac{(\frac{1+\sqrt{5}}{2})^n + (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}\right]$$, $\forall n \geq 0$ where $[x]$ denotes G.I.F.

This algorithm has similar complexity as 3rd algorithm, i.e., $O(P(n) \log n)$.

3. Problem 2 - Karatsuba Algorithm for Multiplication of Large integers

- It is a ***Divide and Conquer*** algorithm which multiplies 2 n -bit integers in less than $O(n^2)$.
- We know that multiplying 2 n-bit integers cost us $O(n^2)$ operations of single digit multiplication, addition and shift operations.
- The idea behind this algorithm is that we split up 2 numbers to be multiplied in two-halves which are high-order bits and low-order bits.

Consider 2 integers say `p` and `q` ,

$$p = (p_1 p_0) = 2^{n/2} p_1 + p_0, \text{ and}$$
$$q = (q_1 q_0) = 2^{n/2} q_1 + q_0$$

$$\text{Here } p_1 \text{ and } q_1 \text{ are leftmost n/2 bits of p and q respectively, } p_0 \text{ and } q_0 \text{ are rightmost n/2 bits of p and q respectively.}$$

Now, multiplication of both will be,

$$\implies pq = (2^{n/2} p_1 + p_0)(2^{n/2} q_1 + q_0)$$
$$\implies pq = 2^n \underbrace{p_1 q_1}_{1} + 2^{n/2} \underbrace{((p_0 + p_1)(q_0 + q_1)}_{2} - p_1 q_1 - \underbrace{p_0 q_0}_{3}) + p_0 q_0$$

$$\text{We multiplied 3 } n/2\text{-digits integers , and performed addition, subtraction and bit-shifting of } n/2\text{-digit integers.}$$

Now,

$$T(n) = \underbrace{T(n/2)}_{p_1 q_1} + \underbrace{T(n/2)}_{p_0 q_0} + \underbrace{T(1 + n/2)}_{(p_0 + p_1)(q_0 + q_1)} + \underbrace{\theta(n)}_{\text{add, sub and bit-shift}}$$

$$\implies T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

Again, the question is *"Can we do better?"* . Yes, we can do better. Algorithm based on FFT (Fast fourier theorem) is of complexity $O(n \log{(n)} (\log{(\log{(n)})}))$, these are faster than Karatsuba algorithm. Though, the fastest known algorithm for multiplying large integers runs in $O(nlogn)$.