week2_lec3

Main Ideas

- Nomenclature Of Problems
- Calculating the n^{th} Fibonacci Number F_n
- Algorithms For Multiplying Large Integers

Nomenclature of Problems

1. Tractable - Any problem that is solvable by a polynomial-time algorithm.

   Upper Bound : Polynomial in nature

2. Intractable - Any problem that cannot be solved by a polynomial-time algorithm.

   Lower Bound : Exponential in nature

Problems

Calculating the n^{th} Fibonacci Number F_n

1.Recursion Algorithm

Fibonacci number n is calculated using: $F_N = F_{N-1} + F_{N-2}$

Time Complexity : T(n) = T(n-1) + T(n-2) : Exponential

The problem: WAP to take N and give F_N

2.Memoization Algorithm

This significantly lowers the complexity of time and space.

Time Complexity : $O(n^2)$

```
int fib(int n) {
     int f[n + 2];
     int i;
     f[0] = 0;
     f[1] = 1;
```

```
    for (i = 2; i <= n; i++)
    {
        f[i] = f[i - 1] + f[i - 2];
    }
    return f[n];
}
```

3.Using Matrix Multiplication

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n * \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Time Complexity : $O(M(n)logn)$[M(n) is the time taken for n bit matrix multiplication]

4.Direct Formula

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

Time Complexity: $O(nlog^2n)$.

Problem in this method is that precision is very low.

Even if precise storage of 5 is achievable, it will still have to be raised to the nth power, which is worse than method 3.

Algorithms For Multiplying Large Integers

Traditional Method

Integers are expressed in binary form and multiplied directly with repeated addition.

Time Complexity:$O(n^2)$

Karatsuba Algorithm

The Karatsuba algorithm is a rapid multiplication method that uses the divide and conquer paradigm to multiply two n-digit integers.

Multiplying two complex no.s

$(a+ib)(c+id) = (ac-bd) + i(ad+bc)$

This operation naively takes four multiplications namely: ac, bd, ad, bc

Instead we can:

- Compute ac
- Compute bd
- Compute $(a+b)(c+d)$

and then obtain $(ad+bc) = (a+b)(c+d) - ac - bd$

So two multiply two n-bit integers x and y, partition each of them into parts that contain $n/2$ of the bits each i.e.

$x = 2^{n/2}a + b$

$y = 2^{n/2}c + d$

therefore

$x \cdot y = (2^{n/2}a + b) \cdot (2^{n/2}c + d)$

Which is similar to the above method of computing complex products but instead of i we have $2^{n/2}$

Time Complexity: $O(nlog_2^3) = O(n^{1.585})$

Algorithms with better complexities than *O*(*n*1.585)

1. Fast Fourier Transform - $O(n*logn*log(logn))$
2. Faster - $O(n*logn*2O(log*n))$
3. Fastest -$O(nlogn)$