Lecture 3

Yash Mehan

25 Aug, 2021

Multiple ways of solving problems

We'll pick a problem which we know how to solve, and approach the same problem in different ways.

Problem 1

Finding the nth fibonacci number. WAP which takes in N and outputs F_N.

$F_N = F_{N-1} + F_{N-2}$

Before we begin, 3 questions:

- for solving the problem, is the algorithm correct? Can you prove its correctness? We can show correctness by taking multiple test cases, but that is not mathematically rigorous.
- Will that take finite time? How much time?
- Can we do better, more efficient?

Algorithm #1 Recursion

```
int fib(int n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    return fib(n-1) + fib(n-2);
}
```

The correctness is much obvious, (if we dont have a memory crunch)

Total number of recursions would exponentially blow up.

Can we do better? Yes

Algorithm #2 DP

Recursion with memoisation

Sure do recursion, but do check in each recursive call is there a value already filled in the memoisation table. If yes, we are going to use that value which was calculated previously, why calculate again?

```
int mem[100];
memset(memset, -1, 100*sizeof(int));
int fib(int n)
{
    if(n==0) return 0;
    if(mem[n] != -1){return mem[n];}
    mem[n] = fib[n-1] + fib[n-2];
    return mem[n];
}
```

Addition of large integers: F_N is $0.694n$ bits long, and addition when n is large is $O(N)$

Overall runtime complexity is $O(N^2)$

But can we do better?

Algorithm #3

Earlier to calculate Nth number, we had to compute for all numbers from 1 to N. Can we skip those calculations?

Matrix multiplication

$$\begin{bmatrix} F_N & F_{N-1} \end{bmatrix} = \begin{bmatrix} F_0 & F_1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$$

How to do matrix exponentiation efficiently? Binary exponentiation and overload the multiplication operators. $O(M(n)logn)$ where $M(n)$ is the complexity for multiplication of matrices with large items.

Algorithm #4 Recurrence Relation

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

Since we are dealing with computers with finite memory, this has accuracy issues, irrational numbers, that too multiplied with themselves many times. in Algorithm 3 and 4 are essentially the same, diagonalise the matrix $\sqrt{5}$ will come up again. This brings us to our second problem.

An $O(nlog^2n)$ algorithm exists for computing F_N. Can we do better? No one knows.

Problem 2

How long does it take for multiplying two large numbers?

- n bit number

Repeated addition

adding a number itself repeatedly. $O(n2^n)$, where n is the number, because adding large numbers is $O(length)$ itself and doing that $\sim 2^n$ times.

School way

$O(n^2)$

```
   11010101
  *01111101
  ---------
  110101010
 000000000
110101010
and so on..
```

Divide and Conquer: Karatsuba Algorithm

Prologue: multiplying two complex numbers $(a + ib)(c + id)$. How many multiplication operations are needed to compute this? 4 multiplications, one addition and one subtraction. Since multiplications are costlier, we dont worry about addition and subtraction. Can we do in 3 multiplications?

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

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

How does divide and conquer help in multiplying large numbers, say, n bits each?

- divide the n bit integers, say, x and y into 2 integers, with equal $(\frac{n}{2})$ or nearly equal number of bits, say, a and b.
- So, $x = a \cdot 2^{\frac{n}{2}} + b$.
- Similarly $y = c \cdot 2^{\frac{n}{2}} + d$
- Do 3-multiplications-technique, just let $2^{\frac{n}{2}}$ as the i here.
- do the add-subtract and $\frac{n}{2}$-place shift to obtain the final answer.

Now the task is to multiply two $\frac{n}{2}$ bit numbers, this divide and conquer can be applied again, and again...

Suppose it took $T(n)$ time to multiply two n bit numbers, so here we have: $T(n) = 3T(\frac{n}{2}) +$ time for add/subtract/shifting

$\implies T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + T(1 + \frac{n}{2}) + \Theta(n)$

By using Master Theorem (next lec), we can find the time complexity of this multiplication algorithm as $O(n^{\log_2 3}) = O(n^{1.585})$

Can we do better? Yes, but later. (FFT)

Fast Fourier Transform

$O(n \cdot logn \cdot log(logn))$

Even faster:

$O(n \cdot logn \cdot 2^{O(log^* n)})$

log^* means how many logs to get output 1 or less.

Best known yet:

$O(nlogn)$

But what is the absolute best algorithm? No one knows whether we can improve the current best.