

Introduction to Algorithms

Irvin Avalos

1 Introduction to Divide and Conquer Algorithms

We begin by examining the implementation of obtaining the n^{th} Fibonacci number, where the

```
1 function Fibonacci(n):
2   | if n = 0 then
3   |   | return 0;
4   | if n = 1 then
5   |   | return 1;
6   | return Fibonacci(n - 1) + Fibonacci(n - 2);
```

number of compute steps is given by

$$T(n) = \begin{cases} C_1 & n = 0 \\ C_2 & n = 1 \\ C_3 + T(n - 1) + T(n - 2) & \text{else} \end{cases} \quad (1)$$

Here $T(n)$ is bounded between $2^{n/2}$ and 2^n leading to a time complexity of $O(2^n)$. This is obviously very bad and so one of the main things we must always ask ourselves when designing an algorithm is: *Can we do better?* In this case we can do better by moving away from a recursive approach to an iterative one. This is because, through each recursion we sometimes recompute values that can be found elsewhere in the recursive tree, e.g., $f_{n-4} = f_{n-3} + f_{n-2}$ where $f_{n-3} = f_{n-2} + f_{n-1}$ and thus end up computing f_{n-2} twice. This should make light of why, for higher values of n , the number of computer steps needed to compute f_n is 2^n . The reason why the iterative method

```
1 function FibonacciIterative(n):
2   | if n = 0 then
3   |   | return 0;
4   | // initialize an array f of size n and
5   | // populate index 0 and 1 with  $f_0, f_1$  respectively
6   | f[0..n];
7   | f[0] = 0, f[1] = 1;
8   | // populate the rest of f with  $i^{\text{th}}$  Fibonacci no.
9   | for i ← 2 to n do
10  |   | f[i] = f[i - 1] + f[i - 2];
11  | return f[n];
```

reduces the number of operations is due to,

- Array initialization of f takes *constant work*
- Setting each f_i is a constant operation C

In total, this approach takes nC computer steps leading to a time complexity of $O(n)$, much better than $O(2^n)$.

1.1 Efficiency of an Algorithm

From the previous example, it should become clear that when designing an algorithm it is important to analyze its running time for very large inputs. Under this perspective, we do not care about constants and amongst multiple terms, e.g., n, n^2, n^3 we choose the one that *grows* the fastest.

Definition (Big O Notation). Let $f(n)$ and $g(n)$ be functions from $\mathbb{Z}^+ \rightarrow R$ where R denotes the number of computer steps. Define $f = O(g)$ if and only if $f \leq g$ and if there exists a constant c such that $\forall n \in \mathbb{Z}^+, f(n) \leq c g(n)$.

1.2 Integer Addition and Multiplication

An operation that we use a good amount of the time is that of adding and multiplying two or more different quantities together. In the case of addition, the algorithm that we are taught runs in $O(n)$ time since it is just an element-wise operation, e.g., adding two arbitrary numbers a and b can be written as

$$\begin{aligned} a + b &= (a_0 + a_1 \times 10 + a_2 \times 100 + \dots) + (b_0 + b_1 \times 10 + b_2 \times 100 + \dots) \\ &= (a_0 + b_0) + 10 (a_1 + b_1) + 100 (a_2 + b_2) + \dots \end{aligned}$$

However, the typical algorithm that we are taught for multiplication proves to be very inefficient. For example, suppose that we multiply two arbitrary numbers c and d ,

$$\begin{aligned} c \times d &= c \times (d_0 + d_1 \times 10 + d_2 \times 100 + \dots) \\ &= (c \times d_0) + (c \times d_1 \times 10) + (c \times d_2 \times 100) + \dots \end{aligned}$$

where c can be further decomposed into $c = c_0 + c_1 \times 10 + c_2 \times 100 + \dots$, and thus yielding a run time of $O(n^2)$ since each base 10 value of c is multiplied to each base 10 of d .

1.2.1 Multiplication using Divide and Conquer Approach

At this point, you should be questioning yourself as to whether or not we can improve the runtime of the typical multiplication algorithm (*Recall: Can we do better?*). Unsurprisingly we can do better by breaking the problem into *smaller* sub-problems, solving them, and combining the results at the end. Now suppose that we are multiplying two numbers x and y each consisting of n digits. Before continuing, we will make one important assumption on n in that $n = 2^k$ for $k \in \mathbb{Z}^+$ (i.e., n is an integer multiple of 2). Thus,

$$x = x_L^{n/2} + x_R \quad \text{and} \quad y = y_L^{n/2} + y_R$$

and their multiplication becomes

$$\begin{aligned} x \times y &= (x_L^{n/2} + x_R) \times (y_L^{n/2} + y_R) \\ &= x_L y_L 2^n + (x_L y_R + x_R y_L)^{n/2} + x_R y_R \end{aligned}$$

Putting this all together we can devise the following algorithm, where the $T(n) :=$ number of steps taken to multiply two n -digit numbers is given by

$$T(n) = 4T(n/2) + O(n)$$

and the total work is then,

$$1 \times O(n) + 4 \times O(n/2) + 4^2 \times O(n/2^2) + \dots + 4^{\log_2 n} \times O\left(\frac{n}{2^{\log_2 n}}\right).$$

One way to think about this is to envision the recursive tree and notice that for the k^{th} level we create a total of 4^k sub-problems. Through each recursion we also split the two input integers into two halves (left and right) and so we have at most $\log_2 n$ levels in the tree. Now, make note of the following fact,

Input: x, y are n -bit positive integers
Output: $x \times y$

```

1 function Mult( $x, y$ ):
2   if  $n = 1$  then
3     return  $x \times y$ ;
4    $x_L, x_R = \text{left } \lceil n/2 \rceil, \text{right } \lfloor n/2 \rfloor$  bits of  $x$ ;
5    $y_L, y_R = \text{left } \lceil n/2 \rceil, \text{right } \lfloor n/2 \rfloor$  bits of  $y$ ;
6    $p_1, p_2, p_3, p_4 = \text{Mult}(x_L, y_L), \text{Mult}(x_L, y_R), \text{Mult}(x_R, y_L), \text{Mult}(x_R, y_R)$ ;
7   return  $p_1 2^n + (p_2 + p_3) 2^{n/2} + p_4$ ;
```

- Given a geometric progression: $1, \alpha, \alpha^2, \dots, \alpha^k$ with $\alpha > 1$, the runtime is equivalent to the last term, i.e., $O(\alpha^k)$

Given this we can now say that the time complexity of the total work is

$$O\left(4^{\log_2 n} \times \frac{n}{2^{\log_2 n}}\right) = O\left(\left(2^{\log_2 n}\right)^2 \times \frac{n}{n}\right) = O\left(n^2\right).$$

However, we said we could improve it by subdividing the problem into smaller ones but we have not improved its time complexity at all. Well notice that

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

where we have already compute the term $-(x_L y_L + x_R y_R)$ which means that one multiplication rather than two suffices such that, $T(n) = 3T(n/2) + O(n)$ and so

$$\begin{aligned} T(n) &= O(n) + 3O(n/2) + 3^2O(n/2^2) + \dots + 3^{\log_2 n}O\left(\frac{n}{2^{\log_2 n}}\right) \\ &= O\left(3^{\log_2 n} \times \frac{n}{2^{\log_2 n}}\right) \\ &= O\left(n^{\log_2 3}\right) \\ &\approx O\left(n^{1.59}\right) \end{aligned}$$

While it may not look like we have made a major improvement one can easily see that for *large* sized integers $n^{1.59}$ grows much slower than n^2 , and thus we have achieved a *faster* runtime.