Amer Din

Nov 8, 2020

Analyzing Section 22.5 & 22.7

**22.5 Finding Fibonacci Numbers Using Dynamic Programming:**

Based on the initial algorithm provided, we can deduce that the 3 cases provide 3 different trajectories for the Fibonacci number. As we see in the first base case, if the argument is 0, it ends the program; In the second base case, we can see the number exiting when the argument is equal to 1; And finally, when the reduction and recursive call. In the case, we can see for each call, the complexity edges nearer to the exponential time due to the recursive nature of the call. In other words, to reach Fib(4), we must find Fib(3) and so on and so forth. This results in the algorithm taking O(2^n) as the recursive call. As each index increases linearly, the result of the function increases exponentially, causing a bigger complexity in the end.

**22.7 Efficient Algorithms for Finding Prime Numbers:**

Regarding this algorithm, we can see that initially a prime number can be formed when the number’s divisor is 1 or itself. The algorithm therefore starts by checking each number in linear time O(n) to see if it is divisible by n or n/2. Consequently, we can then examine if a number is not prime if the number, n has a factor that is greater than 1 and less than or equal to sqrt(n); The time complexity is thus o(sqrt(n)). The enhancement of this algorithm can come when we use the Math.sqrt() function only for perfect squares rather than for every number, thus saving time.