Krithika Kannan

CS 2336.501

HW 9

11/8/2020

Question 3

22.5:

- In section 18.3 we computed the Fibonacci Numbers recursively. The time complexity of finding the Fibonacci number recursively at a specific index is O(2^n). This is not an efficient algorithm to find the Fibonacci number at a given index. This section of Chapter 22 describes how to find an efficient way to compute the Fibonacci Number. A new Fibonacci number is obtained by adding the preceding two numbers in the sequence. If we store the two preceding numbers as “f0 and f1” (as shown in the code on page 872), then “f2” can be calculated by adding “f0 and f1.” This results in the new algorithm being non-recursive with the time complexity of O(n). The non-recursive method is more efficient than the recursive method for finding the Fibonacci number. Now we can answer the questions in the end of section 22.5: “What is dynamic programming? Give an example of dynamic programming. Why is the recursive Fibonacci algorithm inefficient, but the non-recursive Fibonacci algorithm efficient?” So, dynamic programming solves subproblems to get an overall solution. The non-recursive Fibonacci problem that we looked at in this section would be an example of dynamic programming. The Fibonacci algorithm in inefficient because of repeated computations of subproblems, which leads to the time complexity of O(2^n). However, the non-recursive Fibonacci algorithm is efficient because we solve each subproblem only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.

22.7:

- This section goes through 4 different ways to find prime numbers and assesses their efficiency. The first method is by brute-force, which is checks whether 2, 3, 4, 5, . . . , or n – 1 divides n. If not, n is prime. The second method check divisors up to sqrt(n). For the second method, we can prove that if n is not a prime, n must have a factor that is greater than1 and less than or equal to sqrt(n). This reduces the time complexity compared to the first method. The 3rd method is to check prime divisors up to sqrt(n). The method is not efficient if you have to compute Math.sqrt(number) for every iteration. So, we can compute it in the beginning of the method for code reuse. To determine whether i is prime, the algorithm checks whether 2, 3, 4, 5, . . . , and sqrt(i) are divisible by i. This can be further improved by the Sieve of Eratosthenes. For each prime number k, the algorithm sets primes[k \* i] to false (line 19). This is performed n / k − k + 1 times in the for loop. The Sieve of Eratosthenes algorithm is good for a small n such that the array primes can fit in the memory.

Complexities of the 4 methods:

