ASSIGNMENT 1 – COMP 252

Alexandre St-Aubin

January 16, 2024

Exercise 1 Consider a Fibonacci sequence starting with $x_0 = 0$, $x_1 = 1$. For positive integers k and n, we would like to compute $x_n \mod k$ and are using the ram model of computation in which standard arithmetic operations, including "mod" take constant time. Describe how you would proceed in two cases: (1) $k = 627$, (2) $k = n$. In both cases, give your complexity in $O(\cdot)$.

Solution:
Case (1): It is known that for any integer k, the sequence of *Fibonacci* numbers modulo k has a period. Let $\pi(k)$, the *Pisano Period,* denote the length of this period. A list of the *Fibonacci* sequence modulo 627 of length $\pi(627)$ can be computed beforehand.

Algorithm 1: Find the Fibonacci sequence number x_n modulo 627.

input : The Pisano Period *pisanoPeriod* for $k = 627$, an array $fibSeqMod$ of the first *pisanoPeriod* integers of the Fibonacci sequence modulo 627, and the index n of the Fibonacci sequence.

output: $x_n \mod 627$

```
1 x ← n mod pisanoPeriod
2 return fibSeqMod[x]
```

Note that in this first case, the Pisano period and in particular, the first $\pi(627)$ elements modulo 627 of the Fibonacci sequence are constant, no matter the input n. It follows that the time complexity is

$$O(1)$$

Case (2):

Algorithm 2: Fibonacci Modulo N

```
Function fib((n)):
F ← [[1,1],[1,0]];
if n == 0 then
    return 0;
power((F,n−1));
return F[0][0];
Function multiply((F,M)):
x ← F[0][0]·M[0][0]+F[0][1]·M[1][0];
y ← F[0][0]·M[0][1]+F[0][1]·M[1][1];
z ← F[1][0]·M[0][0]+F[1][1]·M[1][0];
w ← F[1][0]·M[0][1]+F[1][1]·M[1][1];
F[0][0] ← x;
F[0][1] ← y;
F[1][0] ← z;
F[1][1] ← w;
Function power((F,n)):
if n == 0 or n == 1 then
    return;
M ← [[1,1],[1,0]];
power((F,n//2));
multiply((F,F));
if n is odd then
    multiply((F,M));
```

Exercise 2 In the bit model of computation, give an efficient algorithm for determining whether a given integer n is a perfect square, and determine its worst-case complexity in big oh notation as a function of n.

Solution:

Algorithm 3: Check whether an integer n is a perfect square.

input : An integer n.
output: True or False.

```
1  left ← 1
2  right ← n
3  while left ≤ right do
4  |   mid ← left + ⌊(right − left)/2⌋
5  |   mid_squared = mid ∗ mid
6  |   if n = mid_squared then
7  |   |   return True
8  |   else
9  |   |   if n < mid_squared then
10 |   |   |   right ← mid − 1
11 |   |   else if n > mid_squared then
12 |   |   |   left ← mid + 1
13 return False
```

We now determine the above algorithm's worst case time complexity in the bit model, denoting it T_n. In the worst case, line **3** will run $\log_2 n$ times, let i be the iteration number. We safely estimate that in the worst case, at iteration i, we have,

$$|mid| = \lfloor \log_2 n \rfloor + 1$$

where $|\cdot|$ denotes the bit length of an integer. Now, we don't need to worry about the additions in each iteration, as they take linear time and are dwarfed by the multiplications/divisions, which take quadratic time. Hence, we conclude that the worst case time complexity in the bit model for this algorithm is

$$T_n = O(\underbrace{\log_2 n}_{\text{\# of iterations}} \times \underbrace{(\log_2 n)^2}_{\text{multiplication}}) = O((\log_2 n)^3)$$

Exercise 3 Assuming a RAM (uniform cost) model of computation, design a recursive divide-and-conquer style $O(n)$ worst-case time algorithm for the following problem. We are given an array $x[1],...,x[n]$ of (possibly negative) integers, and are asked to find two indices $i \leq j$ such that $x[i]+...+x[j]$ is maximal. Prove your claim.

Solution:

Algorithm 4: Maximum Subarray recursive, divide-and-conquer algorithm.

input : An array of integers arr.
output: Two integers, *first*, *last*, representing the first and last index of the maximal subarray.

```
1  prefix, suffix ← arr;
2  for i ← 1 to length(arr) − 1 do
3      prefix [i] += max(0, prefix[i − 1]);
4  for i ← length(arr) − 2 to −1 do
5      suffix [i] += max(0, suffix[i + 1]);
6  return maxSubArray (arr, 0, length(arr));
7  Function maxSubArray (array, left, right):
8      if left = right then
9          return array[left];
10     mid ← ⌊(left + right)/2⌋;
11     return max(maxSubArray(array, left, mid), maxSubArray(array, mid +
       _1, right), prefix[mid] + suffix[mid + 1]);
```