ASSIGNMENT 1 – COMP 252

Alexandre St-Aubin and Jonathan Campana

January 18, 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, since $k = 627$ is constant.

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 $x \leftarrow n \mod pisanoPeriod$
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

$$O(1),$$

since the only thing our algorithm needs to compute is a modulo, and access an index of a finite array (repeats with n growing rather than increasing), both of which can be done in constant time in the RAM model.

Case (2):
We implement the algorithm to compute the n^{th} Fibonacci number using a matrix decomposition of the form

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n,$$

as seen in class. It should be noted that simply multiplying the above matrix $n-1$ times to get the n^{th} element of the Fibonacci sequence yields a time complexity $O(n)$. Thus, we apply a neat little trick, namely, recursive matrix multiplication to bring down this bound to $O(\log n)$. The algorithm is the following:

Algorithm 2: Calculate n^{th} Fibonacci Number modulo n

input : n: an integer
output: Fibonacci number at position n

```
1  Function Fibonacci(n):
2      if n ≤ 1 then
3          return n ;
4      A ← [1 1; 1 0];
5      B ← FastExp(A, n − 1);
6      return B[0][1] mod n;
7  Function FastExp(M, n):
8      if n = 1 then
9          return M;
10     r ← n mod 2 ;
11     m ← ⌊n/2⌋ ;
12     return (FastExp(M, m))^2 ∗ M^r
```

To obtain the n^{th} fibonacci number modulo n, FIBONACCI(n) is called first, which will call FAST-EXP(M,n) to compute the matrix multiplication. The former function clearly operates in constant time in the RAM model. As for the latter, it will have at most $\lfloor \log_2 n \rfloor$ recursive calls, hence at most the same amount of squarings and multiplications. Since these computations take constant time in the RAM model, we conclude that the algorithm runs in $O(\log n)$ time complexity.

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, and we safely estimate that in the worst case, at each of those iterations 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

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

Exercise 3 Maximal Subarray problem.

Solution:

In the following algorithm, PREFIXTUPLEARRAY[i] is a tuple, with the first element being the maximal sum of a subarray **ending** at i, and the second being the index at which this subarray begins. On the other hand, SUFFIXTUPLEARRAY will contain the maximal sum of a subarray **starting** at i as its first element, and the index at which it ends as its second element. The complexity for the computation of prefix is always $O(n)$, as we must loop through each element of the array, similarly for suffix.

The function MAXSUBARRAY is a divide-and-conquer algorithm that divides the array into smaller arrays, until only its length is of length 1. Then, it will compare each array that was divided in 2, and return a tuple representing the maximal sum within those two arrays at the first index, and another tuple with the beginning and ending indices of the subarray with maximal sum at the second index. There are three possibilities for the maximal subarray:

(i) The subarray is entirely contained in the **left** part of the parent array that was divided in 2;

(ii) The subarray is entirely contained in the **right** part of the parent array that was divided in 2;

(iii) The subarray is **split** between the two arrays on the left and the right. This is where the PREFIXTUPLEARRAY and SUFFIXTUPLEARRAY will be used.

The recurrence of the function described above is given by

$$T_n = 2T_{n/2} + O(1).$$

We shall use the Master Theorem to compute its complexity. We notice that

$$n^{\log_b(a)}/f(n) = n^{\log_2(2)}/1 = n,$$

and if we let $\varepsilon = 1/2$, then for any $n > 1$, we have $n > n^{\varepsilon} = \sqrt{n}$. Thus the first case of the theorem applies, and we conclude that

$$T_n = O(n^{\log_b(a)}) = O(n).$$

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

input : An array of integers arr.
output: A tuple representing the first and last index of the maximal subarray.

```
1  prefixTupleArray, suffixTupleArray ← arr;
   // prefixTupleArray and suffixTupleArray are arrays of tuples, prefixTupleArray
      [i][0] contains arr [i]
2  for i ← 1 to length(arr) − 1 do
3      prefixTupleArray [i][0] += max(0, prefixTupleArray[i − 1]);
4      if prefixTupleArray[i − 1] ≥ 0 then
5          prefixTupleArray [i][1]← prefixTupleArray[i − 1][1];
6      else
7          prefixTupleArray [i][1] ← i;
8  for i ← length(arr) − 2 to −1 do
9      suffixTupleArray [i][0] += max(0, suffixTupleArray[i + 1]);
10     if suffixTupleArray[i + 1] ≥ 0 then
11         suffixTupleArray [i][1] ← suffixTupleArray[i + 1][1];
12     else
13         suffixTupleArray [i][1] ← i;
   // prefixTupleArray [i][1] will hold the beginning index of the max subarray
      ending at i, and suffixTupleArray [i][1], starting at i.
14 Function maxSubArray (array, left, right):
15     if left = right then
16         return (array[left], (left, left)); // return a tuple
17     mid ← ⌊(left + right)/2⌋;
18     l ← maxSubArray(array, left, mid);
19     r ← maxSubArray(array, mid + 1, right);
20     a ← (prefixTupleArray[mid][0] + suffixTupleArray[mid +
         1][0], (prefixTupleArray[mid][1], suffixTupleArray[mid + 1][1]));
21     maxTuple ← l;
22     if maxTuple[0] < r[0] then
23         maxTuple ← r;
24     if maxTuple[0] < a[0] then
25         maxTuple ← a;
26     return maxTuple;
27 return maxSubArray (arr, 0, length(arr))[1];
```