Algorithms and Data Structures: Homework #5

Due on March 9, 2020 at 23:00 PM

Henri Sota

Problem 5.1

a) Implement all four methods to compute Fibonacci numbers that were discussed in the lecture: (1) naive recursive, (2) bottom up, (3) closed form, and (4) using the matrix representation.

Algorithm 1 Fibonacci Naive Recursive[n]

```
if $n < 2$ then
    return $n$
else
    return Fibonacci Naive Recursive($n - 1$) + Fibonacci Naive Recursive($n - 2$)
end if
```

Algorithm 2 Fibonacci Bottom Up[n]

```
Declare $L$: Array with $n$ elements
$F[1] = 0$
$F[2] = 1$
for $i = 3$ to $n$ do
    $F[i] = F[i-1] + F[i-2]$
end for
return $F[n]$
```

Algorithm 3 Power of a Number[n, p]

```
if $p == 0$ then
    return 1
else if $p == 1$ then
    return $n$
else
    if $p$ % $2 == 0$ then
        return Power of a Number($n$,$p/2$) * Power of a Number($n$,$p/2$)
    else
        return Power of a Number($n$,$(p-1)/2$) * Power of a Number($n$,$(p-1)/2$) * n
    end if
end if
```

Algorithm 4 Fibonacci Closed Form[n]

```
$\Phi = (1 + 5^{\frac{1}{2}}))/2$
return Power of a Number($\Phi$, n)
```

Algorithm 5 Matrix Multiply[m, n]

```
Declare $O$: 2D-Array with $2 \times 2$ elements
for $i = 1$ to 2 do
    for $j = 1$ to 2 do
        for $k = 1$ to 2 do
            $O[i][j] += M[i][k] * N[k][j]$
        end for
    end for
end for
return $O$
```

Algorithm 6 Power of a Matrix[m, p]

```
if $p == 0$ then
    return $[[1, 0], [0, 1]]$
else if $p == 1$ then
    return $m$
else
    if $p$ % $2 == 0$ then
        return Power of a Matrix($m$,$p/2$) * Power of a Matrix($m$,$p/2$)
    else
        return Power of a Matrix($m$,$(p-1)/2$) * Power of a Matrix($m$,$(p-1)/2$) * m
    end if
end if
```

Algorithm 7 Fibonacci Matrix[n]

```
Fibonacci = $[[1, 1], [1, 0]]$
result = Power of a Matrix(Fibonacci, n)
return result[1][2]
```

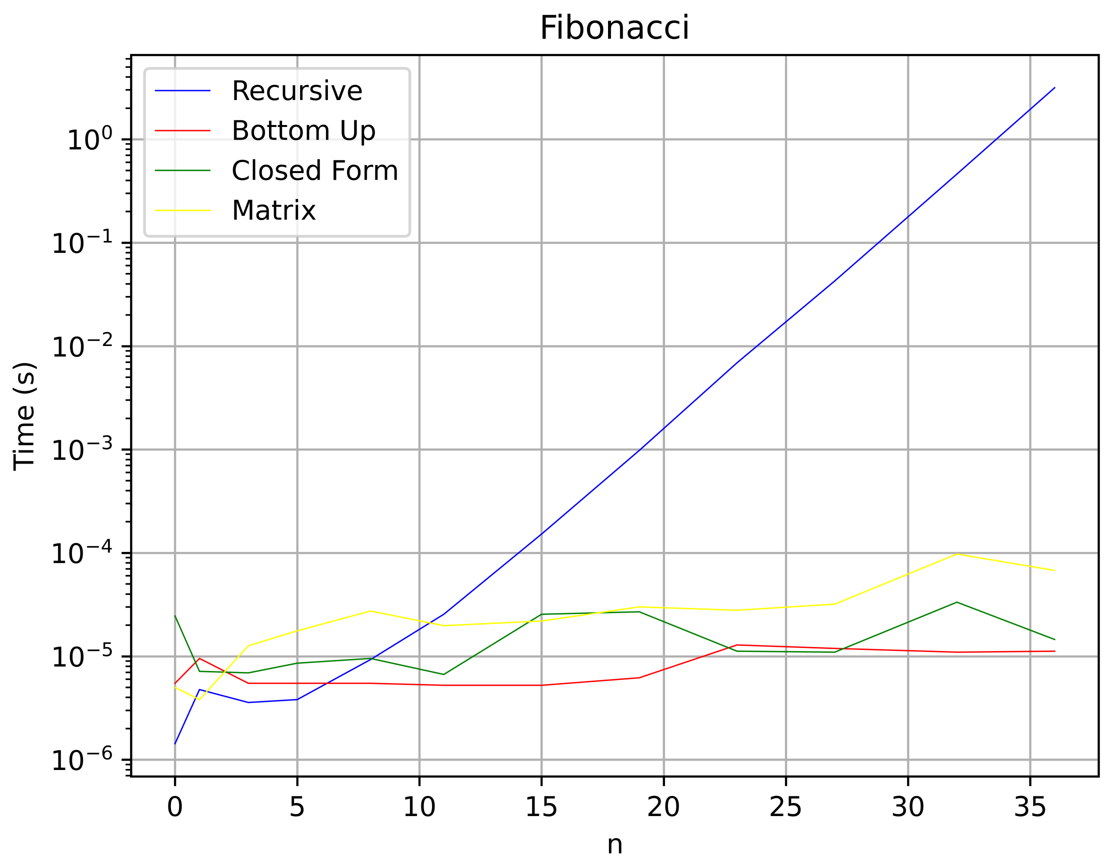
```python
def fibonacciNaiveRecursive(n):
    if (n < 2):
        return n
    else: return fibonacciNaiveRecursive(n - 1) + fibonacciNaiveRecursive(n - 2)

def fibonacciBottomUp(n):
    fibonacciSeq = [0, 1]
    for i in range(2, n+1):
        fibonacciSeq.append(fibonacciSeq[i-1] + fibonacciSeq[i-2])
    return fibonacciSeq[n]

def powerOfANumber(number, p):
    if (p ==  0):
        return 1
    elif (p == 1):
        return number
    else:
        if (p % 2 == 0):
            return powerOfANumber(number, p/2) * powerOfANumber(number, p/2)
        else:
            power = powerOfANumber(number, (p-1)/2)
            return power * power * number

def fibonacciClosedForm(n):
    powerPhi = powerOfANumber(phi, n)
    return round(powerPhi / math.sqrt(5.0))

def matrixMultiply(first, second):
    result = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            for k in range(2):
                result[i][j] += first[i][k] * second[k][j]
    return result

def powerOfAMatrix(matrix, p):
    if (p ==  0):
        return [[1, 0], [0, 1]]
    elif (p == 1):
        return matrix
    else:
        if (p % 2 == 0):
            return matrixMultiply(powerOfAMatrix(matrix, p/2),
                                  powerOfAMatrix(matrix, p/2))
        else:
            power = powerOfAMatrix(matrix, (p-1)/2)
            result = matrixMultiply(power, power)
            result = matrixMultiply(result, matrix)
            return result

def fibonacciMatrix(n):
    fibonacci = [[1, 1], [1, 0]]
    result = powerOfAMatrix(fibonacci, n)
    return result[0][1]
```

Listing 1: Implementation of 4 Fibonacci Sequence Methods in Python

b) Sample and measure the running times of all four methods for increasing n. For each method, stop the sampling when the running time exceeds some fixed amount of time (same for all methods). If needed, you may use classes or structs for large numbers (selfwritten or library components). Create a table with your results (max. 1 page).
Hint: The "gap" between samples should increase the larger n gets,
e.g. $n \in \{0, 1, 2, 3, 4, 5, 6, 8, 10, 13, 16, 20, 25, 32, 40, 50, 63, \cdots\}$.

n	Recursive	Bottom Up	Closed Form	Matrix
0.0	1.430511474609375e-06	5.4836273193359375e-06	2.4557113647460938e-05	5.0067901611328125e-06
1.0	4.76837158203125e-06	9.5367431640625e-06	7.152557373046875e-06	3.814697265625e-06
3.0	3.5762786865234375e-06	5.4836273193359375e-06	6.9141387939453125e-06	1.2636184692382812e-05
5.0	3.814697265625e-06	5.4836273193359375e-06	8.58306884765625e-06	1.7642974853515625e-05
8.0	9.298324584960938e-06	5.4836273193359375e-06	9.5367431640625e-06	2.7418136596679688e-05
11.0	2.5510787963867188e-05	5.245208740234375e-06	6.67572021484375e-06	1.9788742065429688e-05
15.0	0.00015211105346679688	5.245208740234375e-06	2.5510787963867188e-05	2.193450927734375e-05
19.0	0.0009799003601074219	6.198883056640625e-06	2.6941299438476562e-05	3.0040740966796875e-05
23.0	0.00689387321472168	1.2874603271484375e-05	1.1205673217773438e-05	2.7894973754882812e-05
27.0	0.04267311096191406	1.1920928955078125e-05	1.0967254638671875e-05	3.1948089599609375e-05
32.0	0.46109652519226074	1.0967254638671875e-05	3.337860107421875e-05	9.751319885253906e-05
36.0	3.1517608165740967	1.1205673217773438e-05	1.4543533325195312e-05	6.771087646484375e-05

Sampled data has been received until the time required for calculation of Fibonacci Recursive of the next value in the sample surpassed a set time limit. The time limit for the runtime of any of the implementations of the algorithms was set to 10 seconds. Implementation of the sampled algorithms is found in the file fibonacciSampled.py and the data produced by it is found inside the file dataSampled.txt.
There was no need for an implementation of a Big Number class or struct inside of Python as Python natively uses Big Number which can hold the values up to and around Fibonacci(1300).



c) For the same n, do all methods always return the same Fibonacci number? Explain your answer.
For small values of n, all 4 methods produce the same output. For bigger values of n, there is a slight difference between one method and the other 3 that grows bigger as n grows. The method that yields an incorrect result is the Closed Form approach. In practice, working with floating point numbers, in this case an irrational number such as Φ, will include a small margin of error as it is impossible to represent an infinite floating point number in a machine with reserved memory. By multiplying many times a floating point number, the margin of error grows and the rounding at the end will yield an incorrect answer. The other 3 methods do not introduce errors as they perform multiplication and addition only on integers, whole numbers, which can be correctly represented in machines as they are finite.

Problem 5.2

Consider the problem of multiplying two large integers a and b with n bits each (they are so large in terms of digits that you cannot store them in any basic data type like long long int or similar). You can assume that addition, substraction, and bit shifting can be done in linear time, i.e., in $\Theta(n)$.

a) Derive the asymptotic time complexity depending on the number of bits n for a brute-force implementation of the multiplication.
The brute force implementation of multiplying two large integers with n bits each is of time complexit $\Theta(n^2)$. The algorithm works by multiplying each bit of a with each bit of b. Each time there is a multiplication with a bit from a with each bit from b, the result is bit-shifted by the position of the bit in a. The result is calculated by summing up all the results from each bit multiplication.

$$\begin{array}{r} 1011 \\ \times\, 1110 \\ \hline 0000 \\ 1011 \\ 1011 \\ +\,1011 \\ \hline 10011010 \end{array} \tag{1}$$

Since each addition and bit-shift is considered to be done in linear time, the overall time complexity is $\Theta(n^2)$. Multiplying each bit from a with each bit from b is $T(n) = \Theta(n^2)$. Addition of n n-bit numbers takes also $\Theta(n^2)$ time.

$$\begin{aligned} T(n) &= T_{\text{multiplication}}(n) + T_{\text{addition}}(n) = \Theta(n^2) + \Theta(n^2) \\ &= 2\Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

b) Derive a Divide & Conquer algorithm for the given problem by splitting the problem into two subproblems. For simplicity you can assume n to be a power of 2.
We can rewrite a and b in base B as:

$$\begin{aligned} a &= a_l B^m + a_r \\ b &= b_l B^m + b_r \end{aligned}$$

where a_l and b_l represent the left half of the digits of a and b, a_r and b_r represent the right half of the digits of a and b and $m < n$.
Taking base 2, m to be $\frac{n}{2}$ and multiplying both numbers:

$$\begin{aligned} a \cdot b &= (a_l 2^{\frac{n}{2}} + a_r)(b_l 2^{\frac{n}{2}} + b_r) \\ &= a_l b_l * 2^n + 2^{\frac{n}{2}}(a_l b_r + a_r b_l) + a_r b_r \end{aligned}$$

Following this formula, the time complexity is still $\Theta(n^2)$ as there is the need to perform 4 multiplications between the halves of a and b, namely: $a_l b_l, a_l b_r, a_r b_l, a_r b_r$. The complexity is deduced from the recurrence formula:

$$T(n) = 4T(\frac{n}{2}) + \Theta(n)$$

In order to reduce the time complexity of this approach, the middle two products can be calculated by using addition and the other two products: $a_l b_l$, $a_r b_r$

$$\begin{aligned} a_l b_r + a_r b_l &= (a_l + a_r)(b_r + b_l) - a_l b_l - a_r b_r \\ &= a_l b_r + a_l b_l + a_r b_r + a_r b_l - a_l b_l - a_r b_r \\ &= a_l b_r + a_r b_l \end{aligned}$$

Substituting the equation above into the formula:

$$\begin{aligned} a \cdot b &= (a_l 2^{\frac{n}{2}} + a_r)(b_l 2^{\frac{n}{2}} + b_r) \\ &= a_l b_l * 2^n + 2^{\frac{n}{2}}((a_l + a_r)(b_r + b_l) - a_l b_l - a_r b_r) + a_r b_r \end{aligned}$$

As a result, there is one less multiplication needed for this approach. Pseudocode implementation:

Algorithm 8 Multiplication[a, b]

```
$n =$ bits in $a$ and $b$
**if** $n == 1$ **then**
    return $a * b$
**end if**
$a_l, b_l$ = left half bits of $a$, left half bits of $b$
$a_r, b_r$ = right half bits of $a$, right half bits of $b$
left = Multiplication($a_l, b_l$)
right = Multiplication($a_r, b_r$)
mixed = Multiplication($a_l + b_r, a_r + b_l$)
return left * $2^n$ + $2^{n/2}$(mixed−right−left) + right
```

c) Derive a recurrence for the time complexity of the Divide & Conquer algorithm you developed for subpoint (b).
Since there was one less call to multiply two halves of a and b, the recurrence equation becomes:

$$T(n) = 3T(\frac{n}{2}) + \Theta(n)$$

e) Validate the solution in subpoint (d) by using the master theorem to solve the recurrence again.

$$T(n) = 3T(\frac{n}{2}) + \Theta(n)$$

Master method can be used:
$a = 3$
$b = 2$

$$\begin{aligned} &n^{\log_2 3} = n^{1.58} \\ &f(n) = n \\ &f(n) = O(n^{\log_2 3 - \epsilon}) = O(n^{1.58-\epsilon}) \text{ where } \epsilon = \log_2 3 - 1 \approx 0.58 \end{aligned} \tag{2}$$

Case 1: $f(n)$ is polynomially smaller than $n^{\log_2 3}$
Result: $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$