Algorithms and Data Structures

Spring 2019

Assignment 3

Date: February 28, 2019

Taiyr Begeyev

Problem 3.1 Fibonacci Numbers

(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

```
/*
 ----------           .--
\--     ---/ -----     |--|   ---.--.  -------
  |     |     \--  \    |   | <    |   | \-   -- \
  |     |      / -- \- |   |   \---   |   |   | \/
  |-----|     (----   / |--|   / ----|   |--|
                 \/           \/
/*
    (1) Naive Recursion
*/
unsigned long long fibonacciNaive(int n) {
    if(n < 2)
        return n;
    else
        return fibonacciNaive(n - 2) + fibonacciNaive(n - 1);
}

/*
    (2) Bottom Up Approach
*/
unsigned long long fibonacciBottomUp(int n) {
    unsigned long long arr[n + 1];
    if (n < 2)
        return n;
    else {
        arr[0] = 0;
        arr[1] = 1;

        for (int i = 2; i <= n; i++)
            arr[i] = arr[i - 2] + arr[i - 1];
    }
    return arr[n];
}

/*
    (3) Closed form method
*/
unsigned long long fibonacciClosedForm(int n) {
    unsigned long long result = (1 / sqrt(5)) * ceil(pow(((1 + sqrt(5)) / 2), n));
    return result;
}

unsigned long long power(int base, int exp) {
    if(exp == 0)
        return 1;
    else
        return base * power(base, exp - 1);
}

/*
    (4) Matrix Representation
*/
unsigned long long matrixToThePow(unsigned long long arr[2][2], int exp) {
    unsigned long long dummy[2][2];
```

```
    unsigned long long basic[2][2] = {{1, 1}, {1, 0}};

    if (exp == 0) {
        return arr[0][1];
    }
    else {
        for (int i = 0; i < 2; i++) {
            for(int j = 0; j < 2; j++) {
                dummy[i][j] = 0;
                for (int k = 0; k < 2; k++)
                    dummy[i][j] = dummy[i][j] + arr[i][k] * basic[k][j];
            }
        }
        return matrixToThePow(dummy, exp - 1);
    }
}

unsigned long long fibonacciMatrix(int n) {
    if(n == 0)
        return 0;
    unsigned long long arr[2][2] = {{1, 1}, {1, 0}};
    return matrixToThePow(arr, n - 1);
}
```

(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 (self-written 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 0, 1, 2, 3, 4, 5, 6, 8, 10, 13, 16, 20, 25, 32, 40, 50, 63, ...

```
#include "fibonacciNumbers.h"
#include <chrono>
#include <iostream>
#include <cmath>
#include <fstream>
using namespace std;
using namespace std::chrono;

int main() {
    ofstream naiveFile, bottomUpFile, closedFormFile, matrixFile;

    int choice;
    cout << "What function would you like to run" << endl;
    cout << "1. Naive Recursive" << endl;
    cout << "2. Bottom Up Approach" << endl;
    cout << "3. Closed Form" << endl;
    cout << "4. Matrix Represenation" << endl;
    cout << "Enter the corresponding number: ";
    cin >> choice;

    int n = 0;
    switch(choice) {
        case 1:
            naiveFile.open("fibonacciNaive.dat", ios::out | ios::trunc);
            /* (1) Naive method*/
            while(true) {
                // if it is more than 45, it takes bunch of damn time
                if (n > 100) {
```

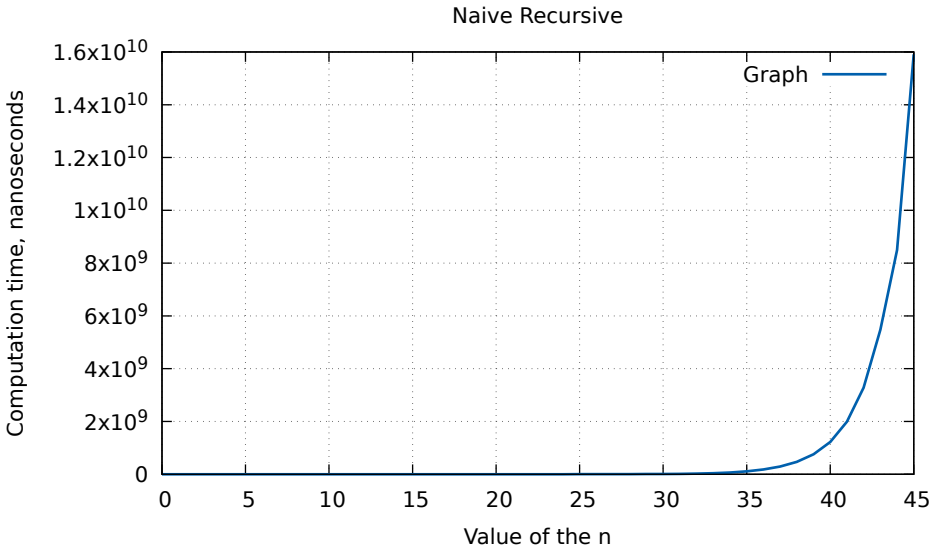
```
                naiveFile.close();
                break;
            }

            high_resolution_clock::time_point t1 = high_resolution_clock::now();
            fibonacciNaive(n);
            high_resolution_clock::time_point t2 = high_resolution_clock::now();

            double duration = duration_cast<nanoseconds>(t2 - t1).count();

            naiveFile << n << " " << duration << endl;
            n++;
        }

        break;
    case 2:
        bottomUpFile.open("fibonacciBottomUp.dat", ios::out | ios::trunc);
        /* (2) Bottom Up Approach */
        n = 0;
        while(true) {
            if (n > 100) {
                bottomUpFile.close();
                break;
            }

            high_resolution_clock::time_point t1 = high_resolution_clock::now();
            fibonacciBottomUp(n);
            high_resolution_clock::time_point t2 = high_resolution_clock::now();

            double duration = duration_cast<nanoseconds>((t2 - t1)).count();

            bottomUpFile << n << " " << duration << endl;
            n++;
        }
        break;
    case 3:
        closedFormFile.open("fibonacciClosedForm.dat", ios::out | ios::trunc);
        /* (3) Closed Form */
        while(true) {
            if (n > 100) {
                closedFormFile.close();
                break;
            }

            high_resolution_clock::time_point t1 = high_resolution_clock::now();
            fibonacciClosedForm(n);
            high_resolution_clock::time_point t2 = high_resolution_clock::now();

            double duration = duration_cast<nanoseconds>(t2 - t1).count();

            closedFormFile << n << " " << duration << endl;
            n++;
        }
    case 4:
        matrixFile.open("fibonacciMatrix.dat", ios::out | ios::trunc);
        /* (4) Matrix represenation */
            while(true) {
            if (n > 100) {
                matrixFile.close();
                break;
```

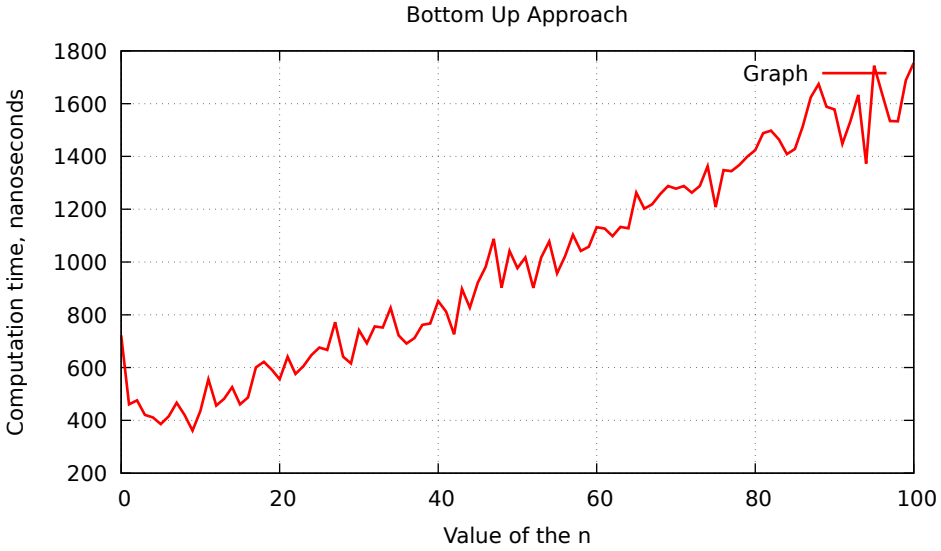
```
            }

            high_resolution_clock::time_point t1 = high_resolution_clock::now();
            fibonacciMatrix(n);
            high_resolution_clock::time_point t2 = high_resolution_clock::now();

            double duration = duration_cast<nanoseconds>(t2 - t1).count();

            matrixFile << n << " " << duration << endl;
            n++;
        }
    }
```

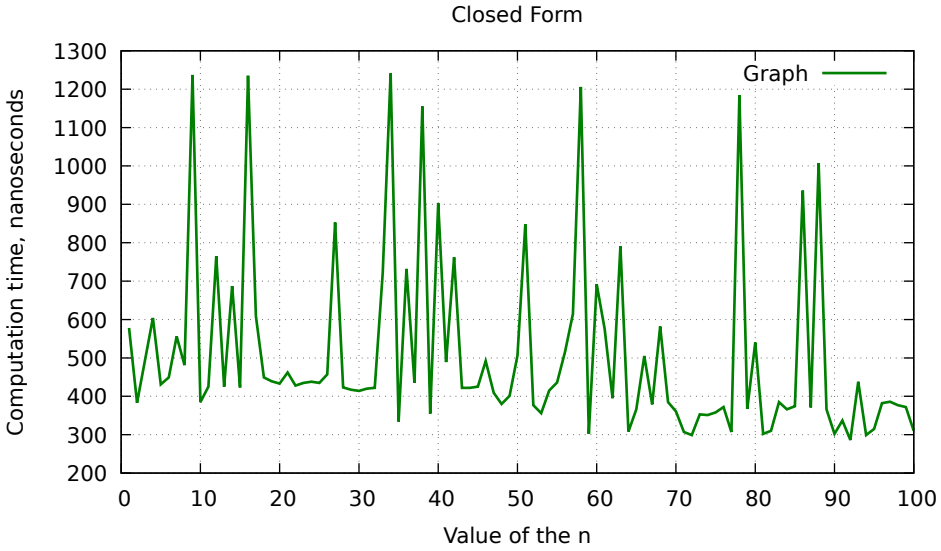
n	Naive Recursive	Bottom Up	Closed Form	Matrix Representation
0	737	722	38890	809
1	416	461	1102	602
2	355	476	827	796
3	341	421	40980	886
4	511	411	1132	995
5	727	386	661	1272
6	862	416	676	1765
8	1684	421	531	2278
10	3168	436	581	2617
13	10809	482	581	3148
16	41541	487	581	3408
20	340335	556	587	4808
25	1826180	676	587	6089
32	27526598	756	581	7398
40	1259024080	852	737	9863
50	———	977	602	12354
63	———	1133	576	16900
75	———	1208	521	20724
90	———	1578	511	25436
100	———	1754	491	27147

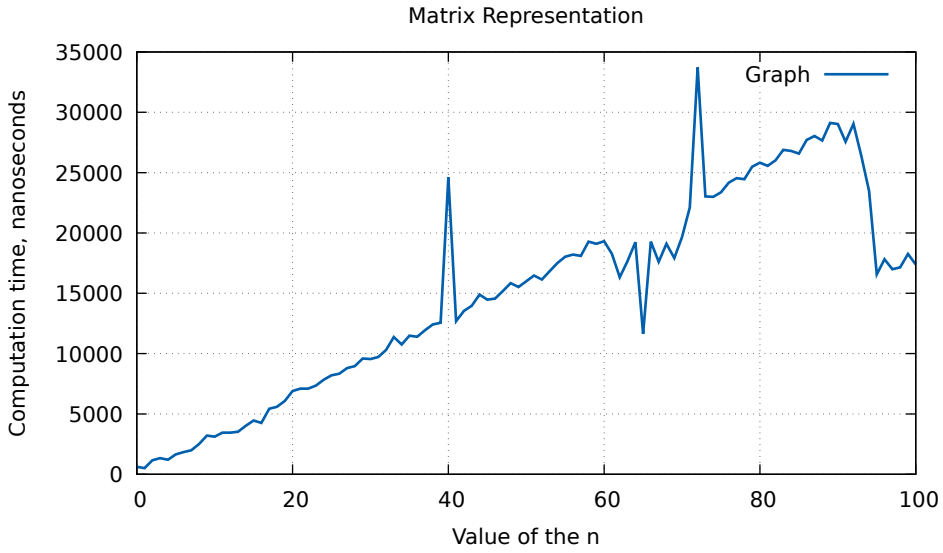
(c) For the same n, do all methods always return the same Fibonacci number? Explain your answer.
No! For all inputs such as $n >= 72$ Closed Formula approach differs from other methods and the larger n gets the bigger difference in results is.
Examples:
n = 72
Bottom Up: 498454011879264
Closed Formula: 498454011879265
Matrix: 498454011879264
n = 85
Bottom Up: 259695496911122585
Closed Formula: 259695496911123328
Matrix: 259695496911122585
Remark: we don't take Naive Recursive method into consideration. It takes huge amount of time to compute those
Since we use round the result every time we return it in Closed Formula Approach, after some time it starts to be imprecise.

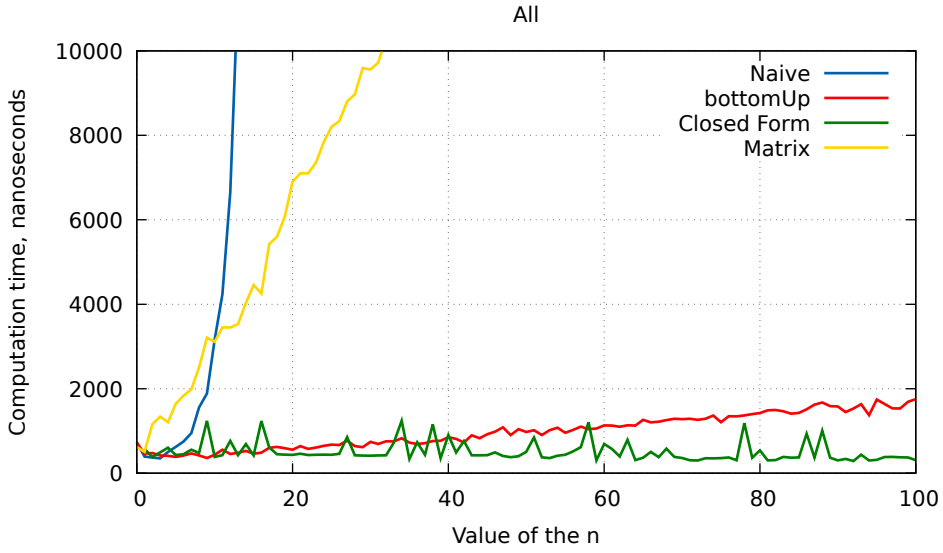
(d) Plot your results in a line plot, so that the four approaches can be easily compared. Briefly interpret your results After plotting all four approaches it is clear that Closed Form approach takes less time than other approaches, but the drawback is that it is not always precise. Bottom Up is universal approach for that matter.


Naive Recursive
Graph
Computation time, nanoseconds
1.6x10^10
1.4x10^10
1.2x10^10
1x10^10
8x10^9
6x10^9
4x10^9
2x10^9
0
0
5
10
15
20
25
30
35
40
45
Value of the n


Bottom Up Approach
Graph
Computation time, nanoseconds
1800
1600
1400
1200
1000
800
600
400
200
0
20
40
60
80
100
Value of the n


Closed Form
Graph
Computation time, nanoseconds
1300
1200
1100
1000
900
800
700
600
500
400
300
200
0
10
20
30
40
50
60
70
80
90
100
Value of the n





Problem 3.2 Divide and Conquer and Solving Recurrences

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, subtraction, 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

To demonstrate the basic idea of the brute force algorithm, let us start with a case of two-digit integers, for example, 23 and 14. These can be written as follows:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0$$
$$14 = 1 \cdot 10^1 + 4 \cdot 10^0$$

Now let's multiply them:

$$(2 \cdot 10^1 + 3 \cdot 10^0)(1 \cdot 10^1 + 4 \cdot 10^0) = (2 \cdot 1)10^2 + (2 \cdot 4 + 3 \cdot 1)10^1 + (3 \cdot 4)10^0 = 322$$

For any pair of two digit numbers $a = a_1 a_0$ and $b = b_1 b_0$ their product c can be computed by the formula

$$c = a \cdot b = c_2 10^2 + c_1 10^1 + c_0$$

where $c_2 = a_1 \cdot b_1$ is the product of their first digits, $c_0 = a_0 \cdot b_0$ is the product of their second digits, $c_1 = (a_1 + a_0)(b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of b's digit minus the sum of c_2 and c_0
Therefore, Time complexity = n^2 one digit multiplications

$$T(n) = O(n^2)$$

(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
Algorithm

```
func (x, y)
    t = max(amount of x digits, amount of y digits)
    if t = 1
        return x * y
    xLeft = left half of x
    xRight = right half of x
    yLeft = left half of y
    yRight = right half of y
    k1 = func(xLeft, yLeft)
    k2 = func(xRight, yRight)
    k3 = func(xRight + xLeft, yLeft + yRight)
    return k1 * 2^n + (k3 - k1 - k2) * 2^n(n/2) + k2
```

We use Gaussian approach above
Divide and Conquer Method
Example, A = 2135 and B = 4014

$$A = (21 \cdot 10^2 + 35)(40 \cdot 10^2 + 14) = 21 \cdot 40 \cdot 10^4 + (21 \cdot 14 + 35 \cdot 40)10^2 + 35 \cdot 14$$

In general, if $A = A_1A_2$ and $B = B_1B_2$, where A and B are n-digit, A_1, A_2, B_1, B_2 are n/2-digit numbers.

$$A \cdot B = A_1 \cdot B_1 \cdot 10^n + (A_1 \cdot B_2 + A_2 \cdot B_1)10^{n/2} + A_2 \cdot B_2$$

The recurrence is clearly, but we can make it better

$$T(n) = 4T(n/2)$$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) \cdot (B_1 + B_2) - A_1 \cdot B_1 + (A_1 \cdot B_2 + A_2 \cdot B_1) + A_2 \cdot B_2$$

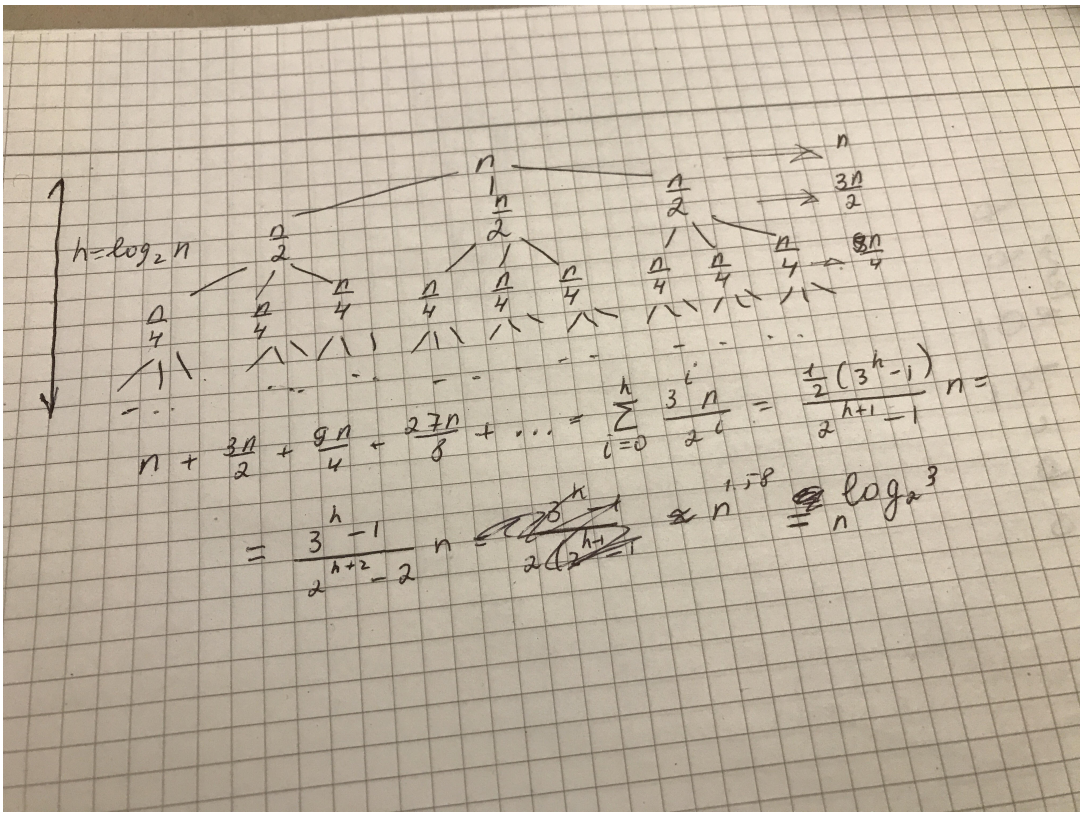
which requires only 3 multiplications

(c) Derive a recurrence for the time complexity of the Divide Conquer algorithm you developed for subpoint (b)
The recurrence is clearly

$$T(n) = 3T(n/2) + \Theta(n)$$

because the addition is linear.

(d) Solve the recurrence in subpoint (c) using the recursion tree method



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

$T(n) = O(n^{log_2 3})$

1. Extract $a, b, f(n)$

$$a = 3, b = 2, f(n) = n$$

2. Determine $n^{\log_b a}$

$$n^{\log_b a} = n^{log_2 3}$$

3. Compare $f(n)$ and $n^{\log_b a}$

Since $n^{log_2 3} < n$, then $n^{log_b a} < f(n)$

Remark: $log_2 3 \approx 1.58$

4. Determine the appropriate case and apply it

Thus case 3:

Since $\frac{2n}{3} \leq cn$ for $c > 0$, i.e $c = \frac{2}{3}$

$$f(n) = \Omega(n^{log_2 3 + \epsilon}) \implies$$

where $\epsilon \approx 0.58$

$$T(n) = \Theta(n^{log_2 3})$$