Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 1:
Study and Empirical Analysis of Algorithms for Determining Fibonacci N-th Term

Elaborated:
st. gr. FAF-211 Kalamaghin Arteom

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and analyze different algorithms for determining Fibonacci n-th term.

Tasks

1. Implement at least 3 algorithms for determining Fibonacci n-th term;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical Notes

An alternative to mathematical analysis of complexity is empirical analysis. This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established;
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm;
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties);
4. The algorithm is implemented in a programming language;
5. Generating multiple sets of input data;
6. Run the program for each input data set;
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction

The Fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers. For example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, … Mathematically we can describe this as: $x_n = x_{n-1} + x_{n-2}$.

Many sources claim this sequence was first discovered or "invented" by Leonardo Fibonacci. The Italian mathematician, who was born around A.D. 1170, was initially known as Leonardo of Pisa. In the 19th century, historians came up with the nickname Fibonacci (roughly meaning "son of the Bonacci clan") to distinguish the mathematician from another famous Leonardo of Pisa. There are others who say he did not. Keith Devlin, the author of Finding Fibonacci: The Quest to Rediscover the Forgotten Mathematical Genius Who Changed the World, says there are ancient Sanskrit texts that use the Hindu-Arabic numeral system - predating Leonardo of Pisa by centuries. But, in 1202 Leonardo of Pisa published a mathematical text, Liber Abaci. It was a “cookbook” written for tradespeople on how to do calculations. The text laid out the Hindu-Arabic arithmetic useful for tracking profits, losses, remaining loan balances, etc, introducing the Fibonacci sequence to the Western world.

Traditionally, the sequence was determined just by adding two predecessors to obtain a new number, however, with the evolution of computer science and algorithmics, several distinct methods for determination have been uncovered. The methods can be grouped in 4 categories, Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, and Benet Formula Methods. All those can be implemented naively or with a certain degree of optimization that boosts their performance during analysis.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations). Within this laboratory, we will be analyzing the 4 naïve algorithms empirically.

Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

Input Format

I’ve come up with two sets of input values of n to test these algorithms. The first one - {5, 7, 12, 17, 20, 22, 27, 30, 35, 37, 42, 45} is for the recursive algorithm; the numbers are relatively small due to the inefficiency of this method (calculating even the 50th number will take more than 100 seconds). Other algorithms will be tested on the second set - {10 000, 100 000, 200 000, 500 000, 1 000 000, 5 000 000, 10 000 000, 20 000 000, 50 000 000, 100 000 000, 200 000 000, 300 000 000}.

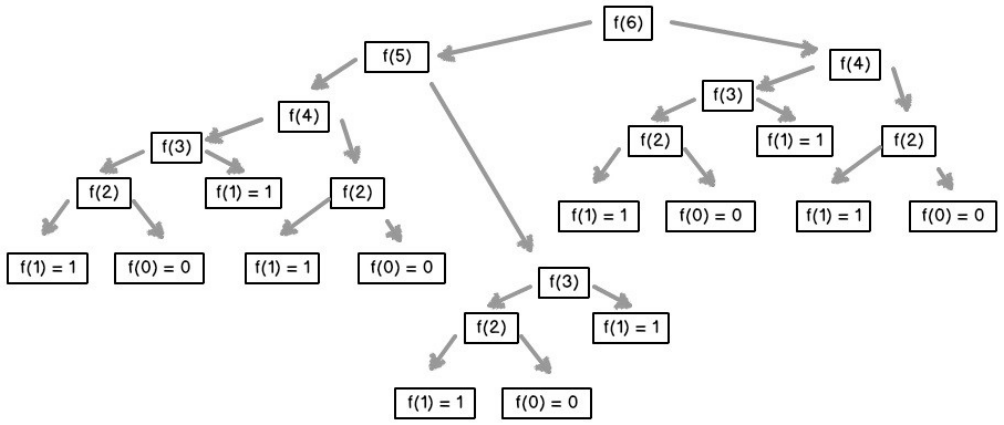
* The boundary value for the Fast Doubling algorithm is 500 000 (takes too much time to calc. further).

IMPLEMENTATIONS

All six algorithms will be implemented in their naive form in C++ and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used. The boundary value of n for the recursive one will be equal to 50, since it was experimentally calculated that the calculation of the 51st number by this method will take over 100 seconds. The runtime calculation will be tied to UNIX Epoch Time.

Recursive Method

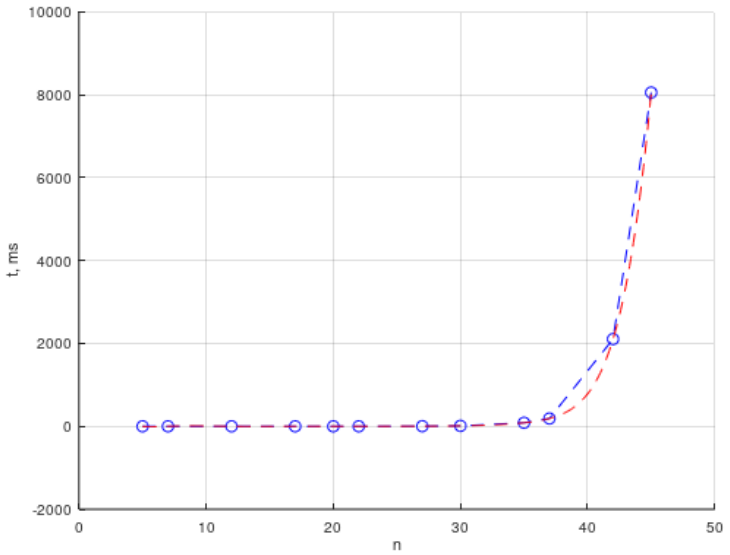
The recursive method, also considered the most inefficient method, follows a straightforward approach of computing the n-th term by computing its predecessors first, and then adding them. However, the method does it by calling upon itself a number of times and repeating the same operation, for the same term, at least twice, occupying additional memory and, in theory, doubling its execution time. **So the time complexity of this method is exponential.**



Implementation:

```
class RecursiveGenerator: public Generator {
    public:
        long long generate(int n) {
            if(n <= 1) {
                return n;
            }

            return generate(n-1) + generate(n-2);
        };
};
```

n	5	…	20	22	27	30	35	37	42	45
t, ms	0	…	0	0	3	11	86	189	2105	8054



Exponential growth is unambiguously expressed.

* In order to visualize smooth exponential growth of the values I used the Lagrange interpolation technique.

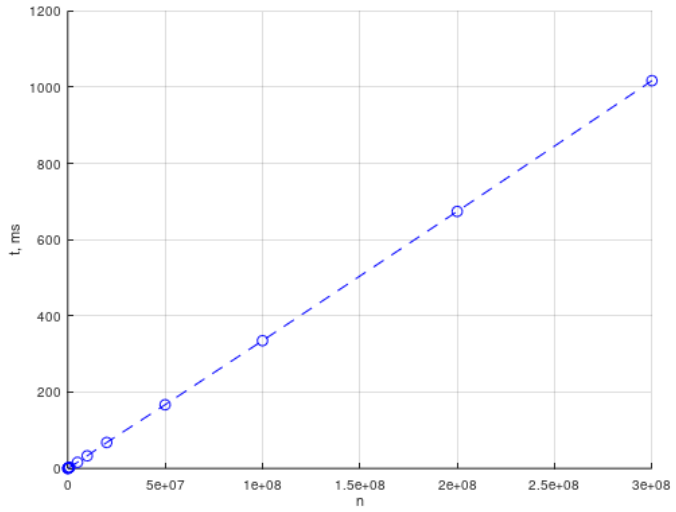
Dynamic Programming Method

In this method, we can avoid the repeated work done in the first method by storing the Fibonacci numbers calculated so far. So to start, we have to initialize two additional variables to store the two last calculated numbers. At initialization, we write in these variables the two first numbers in the Fibonacci series: 0 and 1. Then we perform a very straightforward computation and rewrite values in the following manner: <n1, n2> = <n2, n1 + n2>. **Time complexity for this method amounted to O(n).**

Implementation:

```
class DPGenerator: public Generator {
    public:
        long long generate(int n) {
            long long n1 = 1; long long n2 = 1;

            for(int i = 1; i < n-1; i++) {
                n2 = n1 + n2;
                n1 = n2 - n1;
            }

            return n2;
        };
};
```

n	1e4	…	1e6	5e6	1e7	2e7	5e7	1e8	2e8	3e8
t, ms	0	…	3	16	33	68	167	335	674	1017



O(n) is clearly visible.

‘Exercise 3’ Method

This method has a mathematical nature and was described in the annotation for this laboratory work, it exploits some features and dependencies between numbers in a sequence. T**ime complexity - O(n) : n / 2.**

* Above, I deliberately specify the constant, because its superiority over the previous and the next algorithms, which also run in linear time, becomes quite remarkable as the value of n increases.

Implementation:

```
class Example3Generator: public Generator {
    public:
        long long generate(int n) {
            long long n1 = 1; long long n2 = 0;
            long long k = 0; long long h = 1; long long t = 0;

            while(n > 0) {
                if(n % 2 == 1) {
                    t = n2 * h;
                    n2 = n1*h + n2*k + t;
                    n1 = n1*k + t;
                }

                t = h*h;
                h = 2*k*h + t;
                k = k*k + t;
                n = n / 2;
            };

            return n2;
        };
};
```

n	1e4	…	1e6	5e6	1e7	2e7	5e7	1e8	2e8	3e8
t, ms	0	…	0	0	0	0	0	0	0	0

Just… WOW!

Matrix Exponentiation Method

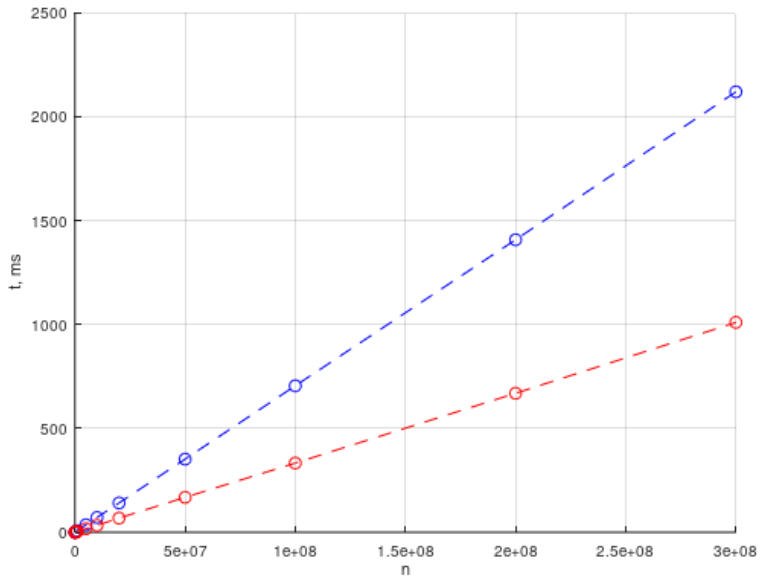
This is another O(n) that relies on the fact that if we n times multiply the matrix M = {{1, 1}, {1, 0}} to itself (in other words calculate power(M, n)), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix. The matrix representation gives the following closed expression for the Fibonacci numbers:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n+1} \end{bmatrix}$$

Implementation:

```
class MatrixExpGenerator: public Generator {
    public:
        long long generate(int n) {
            long long F[2][2] = {{1, 1}, {1, 0}};

            if (n == 0) {
                return 0;
            }

            pow(F, n);
            return F[0][0];
        };

        void pow(long long F[2][2], int n) {
            long long M[2][2] = {{1, 1}, {1, 0}};

            for(int i = 2; i < n; i++) {
                mult(F, M);
            }
        };

        void mult(long long m1[2][2], long long m2[2][2]) {
            long long m11 =  m1[0][0]*m2[0][0] + m1[0][1]*m2[1][0];
            long long m12 =  m1[0][0]*m2[0][1] + m1[0][1]*m2[1][1];
            long long m21 =  m1[1][0]*m2[0][0] + m1[1][1]*m2[1][0];
            long long m22 =  m1[1][0]*m2[0][1] + m1[1][1]*m2[1][1];

            m1[0][0] = m11;
            m1[0][1] = m12;
            m1[1][0] = m21;
            m1[1][1] = m22;
        };
};
```

n	1e4	…	1e6	5e6	1e7	2e7	5e7	1e8	2e8	3e8
t, ms	0	…	7	36	71	141	352	705	1408	2120



In this figure I draw graphs for both Matrix Exp. (blue) and DP (red) approaches. These are two algorithms with linear growth rate, but complex computations performed in the process of the matrix exponentiation makes it spend more time on calculation.

We can **speed up the exponentiation process** in the following manner, **greatly reducing the complexity to O(log(n))**:

```
// OPTIMIZATION
/*
if(n == 0 || n == 1) {
    return;
}

int M[2][2] = {{1, 1}, {1, 0}};

power(F, n/2);
multiply(F, F);

if (n % 2 != 0) {
    multiply(F, M);
}
*/
```

Fast Doubling Method

Formula for this algorithm is derived from the features of the matrix described above. Taking determinant on both sides, we get:

$$(-1)^n = F_{n+1}F_{n-1} - F_n^2$$

Moreover, since $A^nA^m = A^{n+m}$ for any square matrix A, the following identities can be derived (they are obtained from two different coefficients of the matrix product):

$$F_mF_n + F_{m-1}F_{n-1} = F_{m+n-1} \quad (1)$$

By putting n = n+1 in equation (1):

$$F_mF_{n+1} + F_{m-1}F_n = F_{m+n} \quad (2)$$

Putting m = n in equation (1):

$$F_{2n-1} = F_n^2 + F_{n-1}^2$$

Putting m = n in equation (2):

$$F_{2n} = (F_{n-1} + F_{n+1})F_n = (2F_{n-1} + F_n)F_n$$

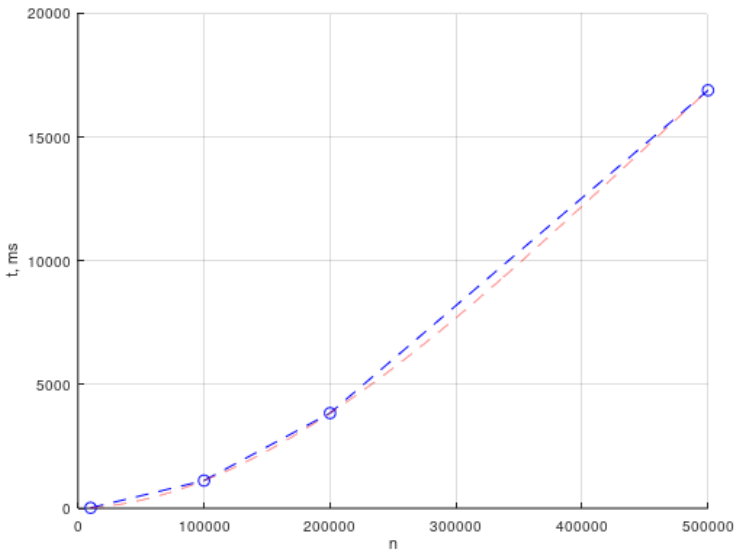
(by putting Fn+1 = Fn + Fn-1)

To get the formula to be proved, we simply need to do the following: if n is even, we can put k = n/2 / if n is odd, we can put k = (n+1)/2. **The presence of recursions makes this algorithm not as efficient as two previous ones, although its recursion tree is smaller than in the case of a simple recursive calculation.**

Implementation:

```
class FDGenerator: public Generator {
    public:
        long long generate(int n) {
            if (n == 0) {
                return 0;
            } else if(n == 1 || n == 2) {
                return 1;
            }

            long long k = (n & 1) ?
                (n+1)/2
                : n/2;

            long long ans = (n & 1) ?
                (generate(k)*generate(k) + generate(k-1)*generate(k-1))
                : (2*generate(k-1) + generate(k)) * generate(k);

            return ans;
        };
};
```

n	1e4	1e5	2e5	5e5	1e6	5e6	1e7	2e7	5e7	…
t, ms	18	1118	3844	16886	~	~	~	~	~	…



Kartik's K Sequence Method

1) **0**, 1, 1, **2**, 3, 5, **8**, 13, 21, **34**, 55, 89, **144**, …..

2) 0, **1**, 1, 2, **3**, 5, 8, **13**, 21, 34, **55**, 89, 144, …..

3) 0, 1, **1**, 2, 3, **5**, 8, 13, **21**, 34, 55, **89**, 144, …..

If you observed the bold numbers in the first line: …

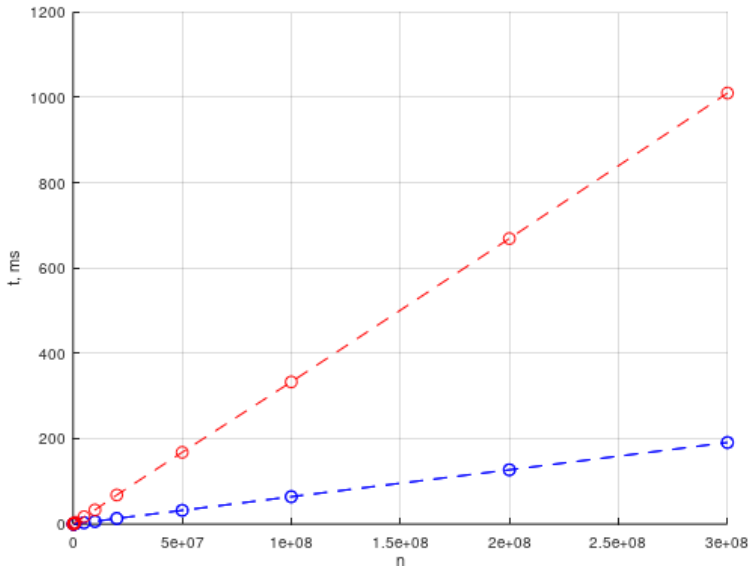
$$\mathbf{2} * 4 + \mathbf{0} = \mathbf{8} \ (7^{th}),\ \mathbf{8} * 4 + \mathbf{2} = \mathbf{34} \ (10^{th}),\ \mathbf{34} * 4 + \mathbf{8} = \mathbf{144} \ (13^{th})$$

$(N+1)^{th} * 4 + N^{th} = (N+2)^{th}$ which can be applied to all three parallel and shifting rules.
Thus, time complexity of this algorithm will be from O(log(n)) to O(n) : n / 3.

Implementation:

```
class KartiksGenerator: public Generator {
    public:
        long long generate(int n) {
            long long tmp = 0;
            long long n1, n2;

            if(n > 0) {
                if(n % 3 == 1) {
                    n1 = 1;
                    n2 = 3;
                } else if (n % 3 == 2) {
                    n1 = 1;
                    n2 = 5;
                } else if (n % 3 == 0) {
                    n1 = 2;
                    n2 = 8;
                }

                int m;
                if(n % 3 == 0) {
                    m = n/3 - 1;
                } else {
                    m = n/3;
                }

                for(int i = 0; i < m; i++) {
                    tmp = n2;
                    n2 = n2*4 + n1;
                    n1 = tmp;
                }

                return n1;
            } else {
                return -1;
            }
        };
};
```

n	1e4	…	1e6	5e6	1e7	2e7	5e7	1e8	2e8	3e8
t, ms	0	…	1	3	6	13	32	64	127	191



Here I also depicted two graphs: the performance of the DP approach (red) and the perf. of the K Sequence approach (blue), so the performance gain that comes from reducing the number of iterations by a factor of 3 in exchange for a slight complication of calculations becomes obvious.

CONCLUSION

Through Empirical Analysis, within this paper, six approaches have been tested in their efficiency at both their providing of accurate results (for relatively small values of n), as well as at the time complexity required for their execution, to delimit the scopes within which each could be used, as well as possible improvements that could be further done to make them more feasible.

Thus, the results for each algorithm will be summarized below in order of increasing performance.

The recursion-based algorithm (which is the easiest in implementation) showed the worst results (during testing, we didn't even reach the second set of test data, since already when working with the first set, the values of n in which did not even exceed 50 - it failed within the time limit set).

In the Fast Doubling implementation, by exploiting the mathematics behind the Fibonacci numbers and the Matrix exponentiation approach we are able to boost the performance that can be demonstrated by the recursion in terms of solving this problem, nevertheless the possibilities of this method are quite limited.

Matrix exponentiation approach is not that straightforward in terms of computations but with an optimisation of the exponentiation process can show a nice performance.

The method based on storing already calculated values in the manner of dynamic programming is also simple to implement, does not take up much additional space and does not involve complex calculations, but nevertheless requires a large number of iterations to obtain a result.

The latter approach - Kartik's K Sequence method is also based on memorizing already calculated values, but the exploitation of the described dependencies has significantly reduced the number of iterations required to get a result.

The third approach, in my opinion, is not as intuitive and simple as the others, but when tested, it shows fantastic results.

Thus, in the context of this laboratory work, I got acquainted both with domain-specific algorithms - algorithms for calculating Fibonacci numbers, and very useful optimization techniques that can be applied in performing routine tasks - matrix exponentiation. I also got acquainted with the UNIX Epoch Time feature, which can be very useful for empirically evaluating the efficiency of performing certain calculations.

* On the next page you can see the format in which I got the output to the console. All the source code you can find by accessing this link: *https://github.com/Starlight-Crusader/AA-Lab-I*.

```
RECURSIVE GENERATOR TESTS (cases0[i] - {5, 7, 12, 17, ...}):
n = 5 : 5, 0 ms
...
n = 22 : 17711, 1 ms
n = 27 : 196418, 4 ms
n = 30 : 832040, 8 ms
n = 35 : 9227465, 70 ms
n = 37 : 24157817, 184 ms
n = 42 : 267914296, 2042 ms
n = 45 : 1134903170, 8682 ms
+============================+
DP GENERATOR TESTS
(cases[i] - {10000, 100000, 200000, ...}):
n = 10000 : > 2^64, 0 ms
...
n = 500000 : > 2^64, 1 ms
n = 1000000 : > 2^64, 4 ms
n = 5000000 : > 2^64, 17 ms
n = 10000000 : > 2^64, 33 ms
n = 20000000 : > 2^64, 68 ms
n = 50000000 : > 2^64, 168 ms
n = 100000000 : > 2^64, 333 ms
n = 200000000 : > 2^64, 669 ms
n = 300000000 : > 2^64, 1010 ms
+============================+
E3 GENERATOR TESTS
(cases[i] - {10000, 100000, 200000, ...}):
...
n = 100000000 : > 2^64, 0 ms
n = 200000000 : > 2^64, 0 ms
n = 300000000 : > 2^64, 0 ms
+============================+
MATRIX EXP. GENERATOR TESTS
(cases[i] - {10000, 100000, 200000, ...}):
...
n = 200000 : > 2^64, 2 ms
n = 500000 : > 2^64, 3 ms
n = 1000000 : > 2^64, 7 ms
n = 5000000 : > 2^64, 36 ms
n = 10000000 : > 2^64, 71 ms
n = 20000000 : > 2^64, 141 ms
n = 50000000 : > 2^64, 352 ms
n = 100000000 : > 2^64, 705 ms
n = 200000000 : > 2^64, 1408 ms
n = 300000000 : > 2^64, 2120 ms
+============================+
KARTIK'S K SEQ. GENERATOR TESTS
(cases[i] - {10000, 100000, 200000, ...}):
...
n = 1000000 : > 2^64, 1 ms
n = 5000000 : > 2^64, 3 ms
n = 10000000 : > 2^64, 6 ms
n = 20000000 : > 2^64, 13 ms
n = 50000000 : > 2^64, 32 ms
n = 100000000 : > 2^64, 64 ms
n = 200000000 : > 2^64, 127 ms
n = 300000000 : > 2^64, 191 ms
+============================+
FAST DOUBLING GENERATOR TESTS
(cases[i] - {10000, 100000, 200000, ...}):
n = 10000 : > 2^64, 18 ms
n = 100000 : > 2^64, 1118 ms
n = 200000 : > 2^64, 3844 ms
n = 500000 : > 2^64, 16886 ms
n = 1000000 : ???, > 5000 ms
...
```