Algorithm Experiment
Methods of Fibonacci

李胜锐 2017012066

2020 年 2 月 27 日

Date Performed: 2020 年 2 月 27 日

1 Experiment Environment

Operating System: macOS 10.15.2
Processor: 2.6 GHz 6-Core Intel Core i7
Memory: 16GB
Language: python 3.7
IDE: PyCharm

2 Algorithm Analysis

2.1 Naive Recursive Algorithm

```
def naive_recursive(n):
    assert n >= 0 and isinstance(n, int)
    if n == 0:
        return 0
    if n == 1:
        return 1
    return naive_recursive(n - 1) + naive_recursive(n - 2)
```

This method makes use of the recursive equation: $F(n) = F(n-1) + F(n-2)$.
Until n is equal to 0 or 1: $F(0) = 0; F(1) = 1$
However, according to the theoretical analysis, its time complexity is:

$$T(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

$$T(n) = \Omega\left(\phi^n\right), \phi = (1+\sqrt{5})/2$$

Therefore, although the algorithm is easy to interprete, it cost to much time. In addition to this, it will also use a lot of space.

2.2 Bottom Up Algorithm

```
def bottom_up(n):
    arr = [0] * (n + 1)
    arr[1] = 1
    for i in range(2, n + 1):
```

```
        arr[i] = arr[i - 1] + arr[i - 2]
    return arr[n]
```

This method also makes use of the recursive equation: $F(n) = F(n-1) + F(n-2)$.
The initial conditions are: $F(0) = 0; F(1) = 1$
And it calculates each $F(n)$ one by one.
Its time complexity is:

$$T(n) = \Theta(n)$$

2.3 Recursive Squaring Algorithm

```
def matrix_power(n):
    assert n > 0 and isinstance(n, int)
    if n == 1:
        return Matrix(1, 1, 1, 0)
    if n % 2 == 0:
        return matrix_multiply(matrix_power(int(n / 2)),
        matrix_power(int(n / 2)))
    else:
        m1 = matrix_power(int(n / 2))
        m2 = matrix_multiply(m1, Matrix(1, 1, 1, 0))
        return matrix_multiply(m1, m2)

def recursive_squaring(n):  # 利用矩阵递归求解, 复杂度\Theta(lgn)
    result_matrix = matrix_power(n)
    return result_matrix.a_2
```

This method also makes use of the following theorem:

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

It is easy to prove with the following equation:

$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$$

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Thus, we can make use of the thought of **Divide and Conquer**.

Denote that $M^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$.

Obviously, $M^n = M^{n/2} \times M^{n/2}$ for even n.

And $M^n = M^{\lfloor n/2 \rfloor} \times M^{\lceil n/2 \rceil} = M^{\lfloor n/2 \rfloor} \times (M^{\lfloor n/2 \rfloor} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix})$ for odd n.

Thus, with the original matrix of power n, we are able to only do a few calulation to get the matrix of power approximately half of n. Therefore, its time complexity is:

$$T(n) = \Theta(\lg n)$$

3 Result Analysis

3.1 Comparision of 3 Methods When n is Small

表 1: Time Cost(second) of Three Methods When n is Small

n	Naive Recursive Algorithm	Bottom Up Algorithm	Recursive Squaring Algorithm
1	4.000000000004e-06	4.000000000004e-06	5.999999999950489e-06
2	4.000000000004e-06	4.000000000004e-06	1.2000000000012001e-05
3	2.0000000000575113e-06	4.000000000004e-06	9.99999999995449e-06
4	4.000000000004e-06	1.999999999946489e-06	1.6000000000016e-05
5	5.999999999950489e-06	4.000000000004e-06	1.0000000000065512e-05
6	1.0000000000065512e-05	1.999999999946489e-06	1.6000000000016e-05
7	1.4000000000069512e-05	4.000000000004e-06	1.2000000000012001e-05
8	2.399999999991298e-05	4.000000000004e-06	2.8000000000028002e-05
9	4.0000000000040004e-05	1.999999999946489e-06	1.799999999996249e-05
10	6.20000000000065e-05	4.000000000004e-06	2.199999999996649e-05
11	9.80000000000425e-05	4.000000000004e-06	1.6000000000016e-05
12	0.00015999999999993797	4.000000000004e-06	3.2000000000032e-05
13	0.00025600000000000034	4.000000000004e-06	2.199999999996649e-05
14	0.00041800000000002946	4.000000000004e-06	2.8000000000028002e-05

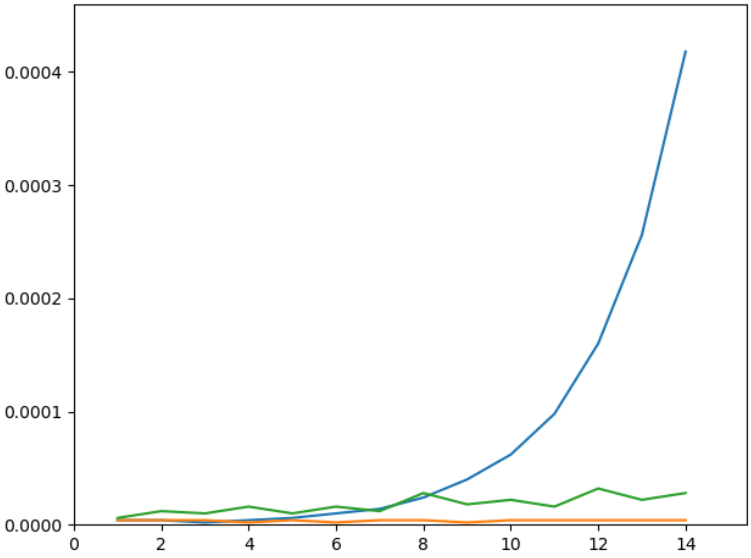

图 1: Time Cost(second) of 3 Methods When n is Small: Naive Recursive; Bottom Up; Recursive Squaring

1) We can judge from the result that Naive Recursive peforms **poorly** even when n is small. And its time cost has a expected exponential growth.

2) Bottom Up is very stable and cost the least time when n is small.

3) Recursive Squaring is not stable when n is small. But its time cost also hardly increases when n increases.

3.2 Time Cost(second) of the Latter two Methods When n is Large

表 2: Comparision of the Latter two Methods When n is Large

$log_{10}n$	Bottom Up Algorithm	Recursive Squaring Algorithm
1.0	9.999999999843467e-06	5.0000000000105516e-05
2.0	4.0000000000040004e-05	0.0002260000000000595
3.0	0.00040400000000007097	0.0006339999999998014
4.0	0.00994600000000001	0.006214000000000164
5.0	0.5764480000000001	0.052180000000000115
5.301029995663981	1.844554	0.10638600000000009
5.342422680822207	1.985662	0.09979799999999983
5.361727836017593	2.249656	0.1076700000000006
5.380211241711606	2.283073999999999	0.12300000000000111
5.3979400086720375	2.5866719999999983	0.14142800000000122
5.414973347970818	2.696092	0.10472000000000037
5.431363764158987	2.971315999999998	0.11842599999999948
5.447158031342219	3.2466979999999985	0.16259399999999857
5.4623979978989565	3.268186	0.12013999999999925
5.477121254719663	3.5991	0.13301600000000136

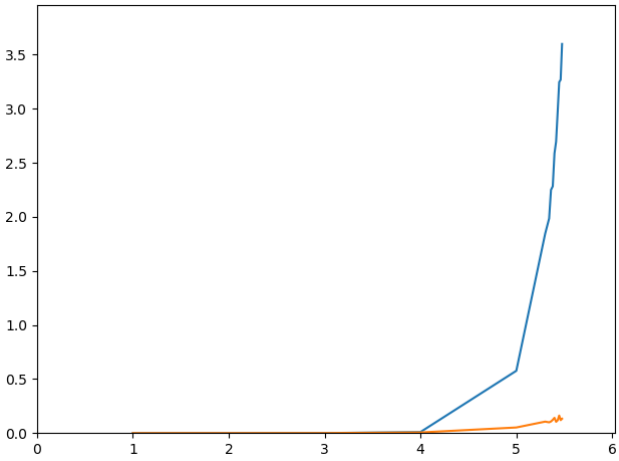

图 2: Time Cost(second) of the Latter two Methods When n is Large **(x-axis represents** $\log_{10} n$**)**: Bottom Up; Recursive Squaring

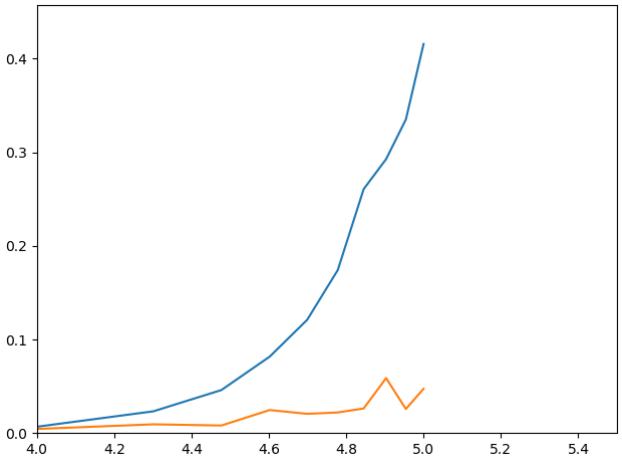

图 3: (Details) Time Cost(second) of the Latter two Methods When n is Large **(x-axis represents** $\log_{10} n$**)**: Bottom Up; Recursive Squaring

1) In the scale of $\lg n$, the time cost of Bottom Up significantly increases when $\log_{10} n > 4$. While the time cost of Recursive Squaring increases slowly.

2) Bottom Up is very stable and cost the least time when n is small.

3) Recursive Squaring is not stable even when $\log_{10} n > 4$.

3.3 Time Cost(second) of the Recursive Squaring Algorithm When n is Very Large

表 3: Time Cost(second) of the Recursive Squaring Algorithm

$log_{10}n$	$log_{10}n$	Recursive Squaring Algorithm
4.0	4.0	0.0042879999999999585
4.698970004336019	4.698970004336019	0.01596999999999993
5.0	5.0	0.03820599999999985
5.698970004336019	5.698970004336019	0.3261640000000001
6.0	6.0	1.006824
6.698970004336019	6.698970004336019	9.598276
7.0	7.0	28.861670000000004

1) It is out of our expectation that the time complexity of Recursive Squaring Algorithm is not strictly follow $log_{10}n$. It is probabily that when n is to large (greater than 1 billion), some basical calulation such as matrix multiplication will cost much more time.

4 Conclusions

1) The time complexities of the three algorithm are:

$$T_{NRA}(n) = \Omega\left(\phi^n\right), \phi = (1+\sqrt{5})/2$$

$$T_{BU}(n) = \Theta(n)$$

$$T_{RSA}(n) = \Theta(\lg n)$$

And the experiment result varified these theorems.

2) Naive Recursive Algorithm performed poorly even when n is very Small

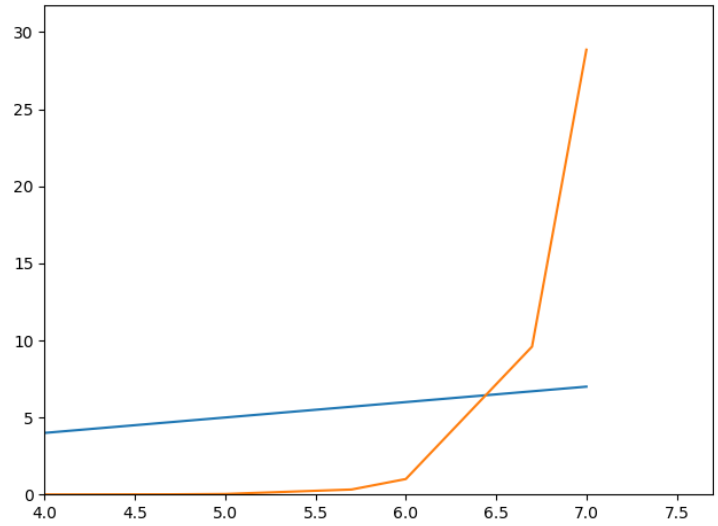

图 4: Time Cost(second) of the Recursive Squaring Algorithm: $log_{10}n$; Recursive Squaring

3) When $n < 10000$, it is better to use Bottom Up Algorithm. Its stablity is better than Recursive Squaring Algorithm and it is even faster.

4) When $n > 10000$, it is obviously that Recursive Squaring Algorithm outperforms all the other algorithms.

5) However, it is out of our expectation that the time complexity of Recursive Squaring Algorithm is not strictly follow $log_{10}n$. It is probabily that when n is to large (greater than 1 billion), some basical calulation such as matrix multiplication will cost much more time.