Algorithms and Data Structure 3

Yiping Deng

February 26, 2018

Problem 1

a)

The implementation is included here.
The naive approach:

```
def fib(n):
    # We use the classical definition of Fibonacci number
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

The bottom-up approach:

```
def fib(n):
    """Return the nth Fibonacci number
    >>> fib(2)
    1
    >>> fib(3)
    2
    >>> fib(4)
    3
    >>> fib(5)
    5
    """
    if n == 0:
        return 0
    if n == 1:
        return 1
    n_1 = 0
    n_2 = 1
    for i in range(1, n):
        # execute n - 1 times
        n_2 = n_1 + n_2
        n_1 = n_2 - n_1
    return n_2
```

The formula approach:

```
import math

phi = (1.0 + math.sqrt(5.0)) / 2.0
one_minus_phi = 1.0 - phi
sqrt_5 = math.sqrt(5)
def fib(n):
    # use formula to calculate the number
```

```
    try:
        f_fib = (math.pow(phi, n) - math.pow(one_minus_phi, n)) / sqrt_5
    except OverflowError:
        f_fib = -1.0
    return int(round(f_fib))
```

The matrix approach:

```
class Matrix:
    def __init__(self, n11, n12, n21, n22):
        self.n11 = n11
        self.n12 = n12
        self.n21 = n21
        self.n22 = n22
    def mul(self, other):
        n11 = self.n11 * other.n11 + self.n12 * other.n21
        n12 = self.n11 * other.n12 + self.n12 * other.n22
        n21 = self.n21 * other.n11 + self.n22 * other.n21
        n22 = self.n21 * other.n12 + self.n22 * other.n22
        return Matrix(n11, n12, n21, n22)
    def p(self):
        print("{} {}\n{} {}".format(self.n11,
            self.n12, self.n21, self.n22))
        return

base = Matrix(1, 1, 1, 0)
def fib_matrix(n):
    if n == 1:
        return base
    m_half = fib_matrix(n // 2)
    # half value
    res = m_half.mul(m_half)
    if n % 2 == 1:
        # deal with reminder
        res = res.mul(base)
    return res

def fib(n):
    if n == 0:
        return 0
    return fib_matrix(n).n21
```

b)

The table is here

```
recursive:
n = 0, t = 572
n = 1, t = 620
n = 10, t = 45156
n = 20, t = 5037832
bottom up:
n = 0, t = 953
n = 1, t = 763
n = 10, t = 1812
n = 20, t = 2097
n = 50, t = 3862
n = 100, t = 8058
n = 200, t = 19836
n = 400, t = 46777
```

```
n = 800, t = 107050
n = 1600, t = 254011
n = 3200, t = 690364
formula:
n = 0, t = 4339
n = 1, t = 1049
n = 10, t = 1001
n = 20, t = 810
n = 50, t = 954
n = 100, t = 810
n = 200, t = 1001
n = 400, t = 858
n = 800, t = 954
n = 1600, t = 1812
n = 3200, t = 1525
n = 6400, t = 1812
n = 128000, t = 1812
m_times:
n = 0, t = 1240
n = 1, t = 3194
n = 10, t = 7963
n = 20, t = 9441
n = 50, t = 12397
n = 100, t = 15592
n = 200, t = 18024
n = 400, t = 20599
n = 800, t = 24032
n = 1600, t = 30183
n = 3200, t = 46777
n = 6400, t = 101852
```

and the code to generate the table

```python
from time import time
import matrix_fib
import formula_fib
import bottom_up_fib
import recursive_fib

def timer(f):
    # a functional wrapper
    # will return the result in microsec
    def wrapper(*arg, **kw):
        t1 = time()
        res = f(*arg, **kw)
        t2 = time()
        return int(round((t2 - t1) * 1000000000)), res
    return wrapper

def avgTimer(f):
    timer_f = timer(f)
    def wrapper(*arg, **kw):
        s = 0
        for _ in range(5):
            t, _ = timer_f(*arg, **kw)
            s += t
        return s // 5
    return wrapper
```

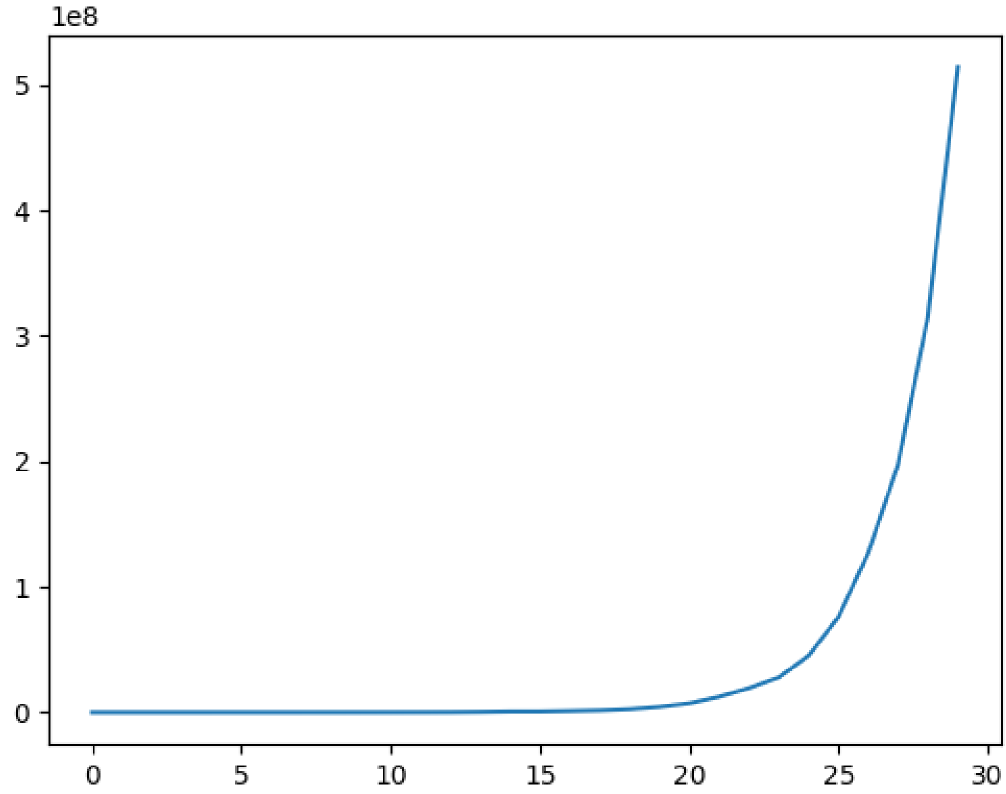
```python
def feeder(f):
    f_wrapped = avgTimer(f)
    t1 = time()
    n = 0
    res = []
    while (time() - t1) < 5.0:
        n += 1
        t = f_wrapped(n)
        res.append(t)
    return res

def calculate():
    """Quick check of correctness of 4 approaches
    >>> recursive_fib.fib(20) == bottom_up_fib.fib(20)
    True
    >>> formula_fib.fib(20) == recursive_fib.fib(20)
    True
    >>> matrix_fib.fib(20) == formula_fib.fib(20)
    True
    """
    recursive_times = feeder(recursive_fib.fib)
    bottom_up_times = feeder(bottom_up_fib.fib)
    formula_times = feeder(formula_fib.fib)
    matrix_times = feeder(matrix_fib.fib)
    return recursive_times, bottom_up_times, formula_times, matrix_times

sampleNum = {0, 1, 10, 20, 50, 100, 200, 400, 800, 1600, 3200, 6400, 128000}

def createTable(lst):
    count = 0
    for t in lst:
        if count in sampleNum:
            print("n = {}, t = {}".format(count, t))
        count += 1
if __name__ == "__main__":
    r_times, b_times, f_times, m_times = calculate()
    print("recursive:")
    createTable(r_times)
    print("bottom up:")
    createTable(b_times)
    print("formula:")
    createTable(f_times)
    print("m_times:")
    createTable(m_times)
```

c)

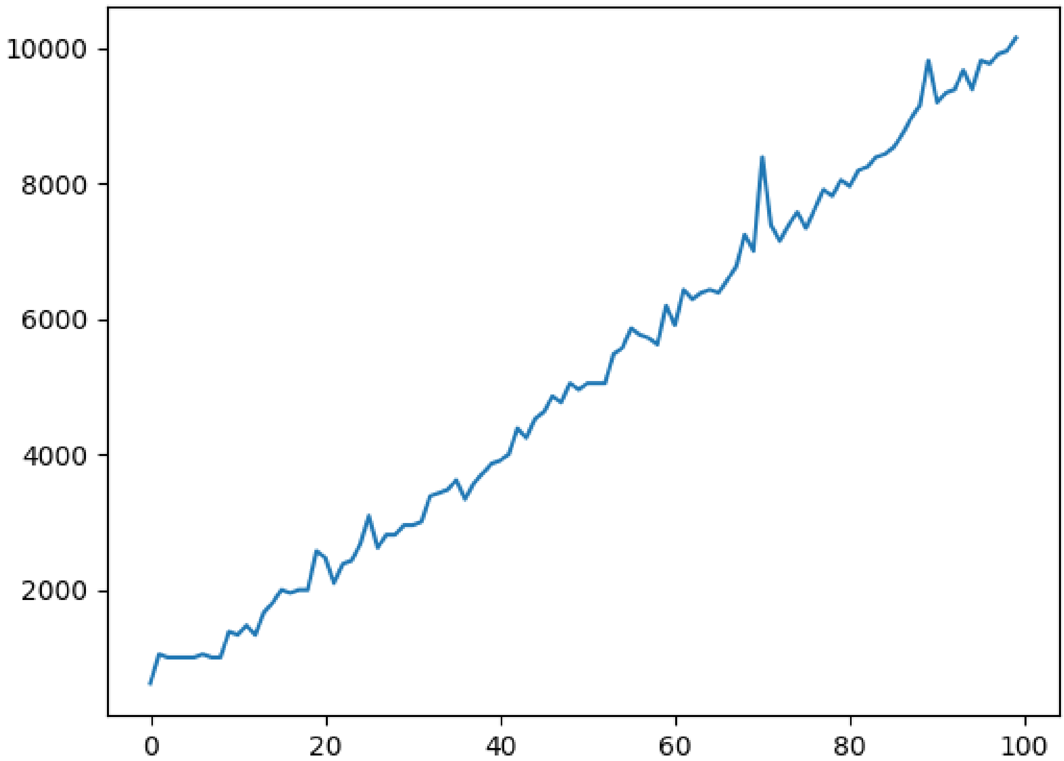
The recursive approach is applying the definition, so it is correct. The bottom up approaches maintains the loop invariant. Before and after the loop, the Fibonacci number n1 and n2 is maintained. They are adjacent Fibonacci number. The formula can be validated using induction. The last formula, similar to the previous one, is using the same induction technique. A quick check of the proof will be via a unit test. In the above code, you can see the unit test is running correctly. The unit test is in the calculate function. You can run it by doing **python -m doctest -v fib_timer.py**

d)

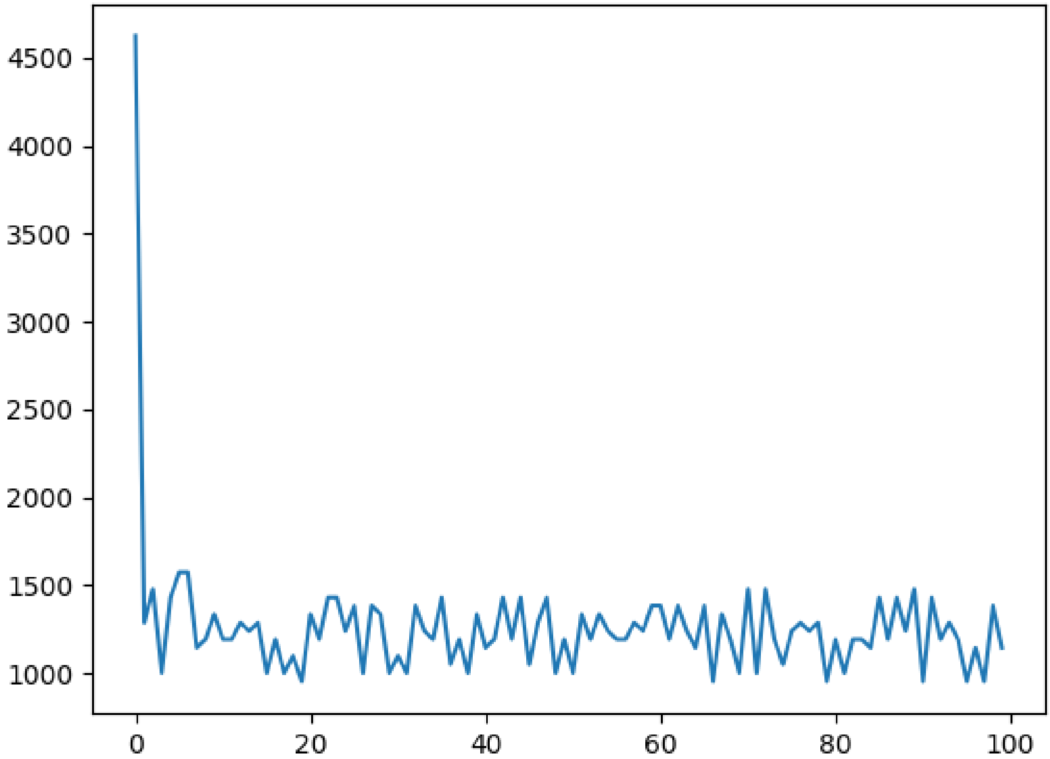
recursive approach



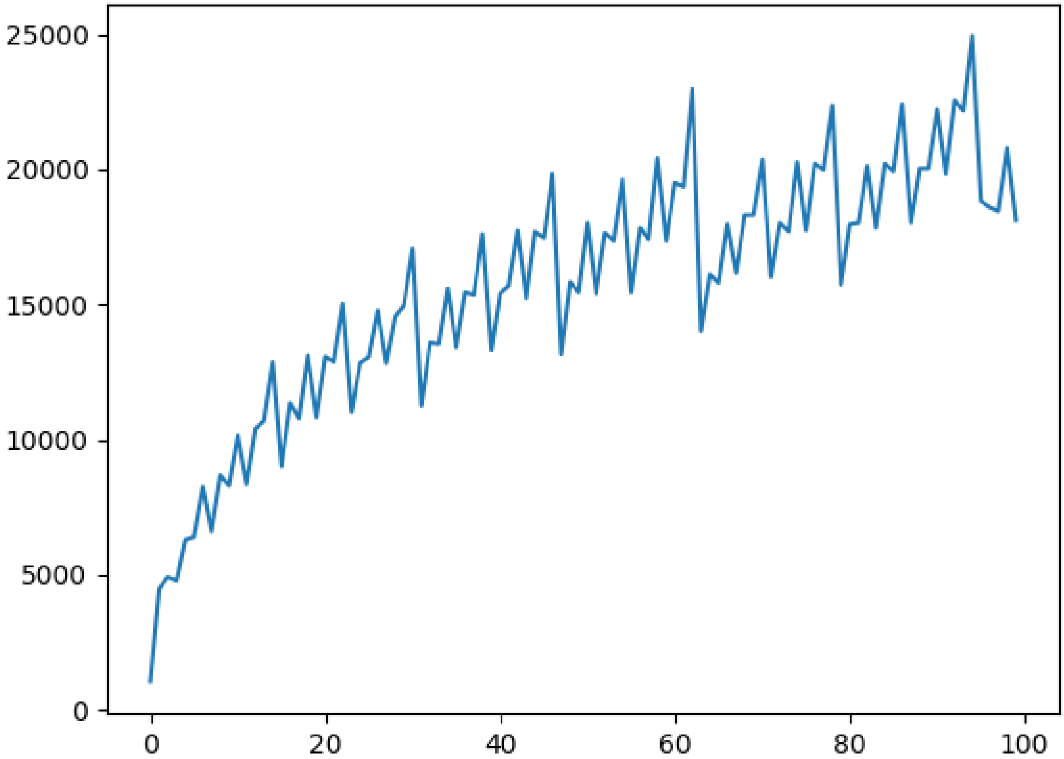
bottom up approach



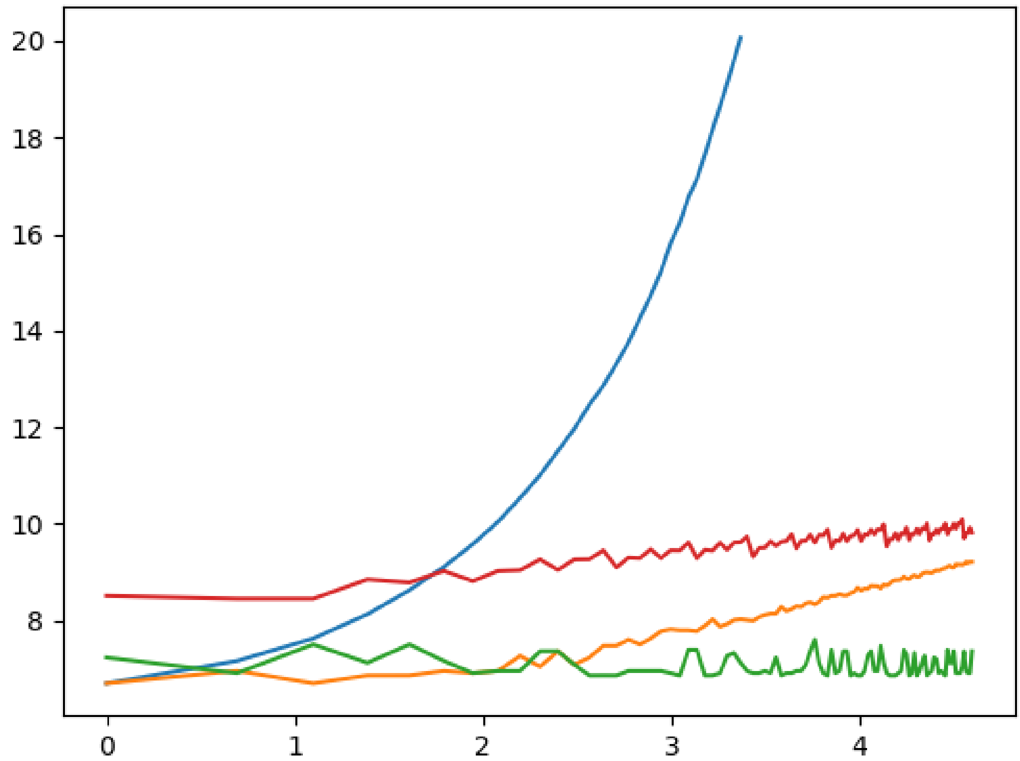
formula approach



matrix approach



the log-log plot



Problem 2

a)

The time complexity of addition of arbitrary length n array is $O(n)$, and also, the brute force approach will add for 2^n times. In total, it is $O(n * 2^n) = O(2^n)$ A smarter approach is using the algorithm below.

```
def mul(x, y):
    if x == 1:
        return y
    halved = mul(x // 2, y)

    res = halved << 1
    # bit shift 1 to double the value,
    # this operation is linear time in modern CPU

    if x % 2 == 1;
        res = res + y
        # this operation is in linear time
    return res
```

As you can see in this algorithm, use implement it by calculate the half value first, and sum it up. The total running time for this naive approach is $O(n^2)$

b)

Using Gaussian approach, the algorithm is below

```
def mul(x, y):
    n = max(num of digit of x, num of digit of y)
    if n == 1:
        return x * y
    x_l, x_r = left half of x, right half of x
    y_l, y_r = left half of y, right half of y
    p_1 = mul(x_l, y_l)
    p_2 = mul(x_r, y_r)
    p_3 = mul(x_l + x_r, y_l + y_r)
    return p_1 * 2^n + (p_3 - p_1 - p_2) * 2^(n / 2) + p_2
```

It uses the formula:

$$xy = (2^{n/2}x_l + x_r)(2^{n/2}y_l + y_r) = 2^n x_l y_l + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r$$
$$x_l y_r + x_r y_l = (x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r$$

c)

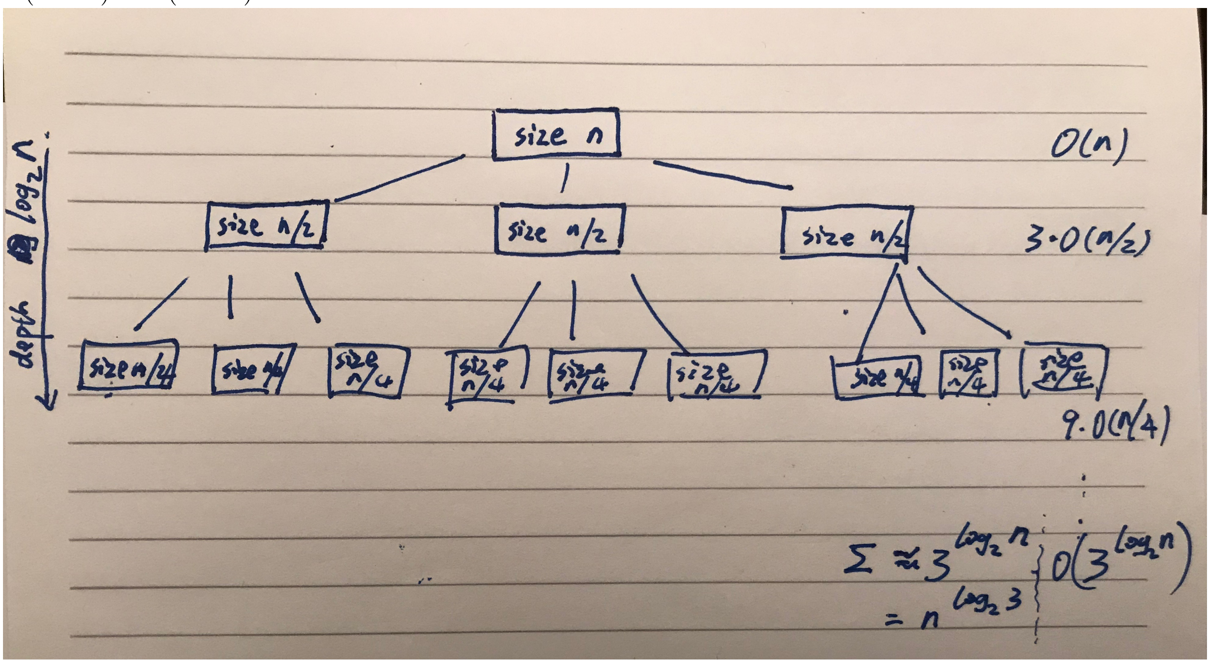
The recurrence is clearly

$$T(n) = 3T(n/2) + O(n)$$

because addition is linear, and the recursive call reduce the size by half.

d)

Using the tree method, the depth is $log_2 n$, and at the bottom level, it is $O(3^{log_2 n})$ operation. Such sum is a geometric series, bounded above by a the highest term by a constant factor. Thus, it is $O(3^{log_2 n}) = O(n^{log_2 3})$



e)

Using the master theorem, $a = 3$, $b = 2$, $d = 1$, thus $T(n) = O(n^{log_2 3})$