UIT2201 Programming and Data Structures

Introduction to Recursion

Chandrabose Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

May 18, 2022

Introduction

- We saw that divide and conquer strategy naturally results in a recursive implementation
- Recursive algorithms can be easily analyzed using recurrence equations
- Properties of recursive structures can be proved using structural induction
- In this lecture, we will discuss about recursive algorithms and their implementations

- Consider a classic problem to compute the factorial of a number n

Iteration Vs Recursion

- Consider a classic problem to compute the factorial of a number n
- $n! = 1 \times 2 \times \cdots \times (n-1) \times n$

Iteration Vs Recursion

- Consider a classic problem to compute the factorial of a number n
- $n! = 1 \times 2 \times \cdots \times (n-1) \times n$

```
def fact(n):
    ans = 1
    for i in range(2, n+1):
        ans *= i
    return ans
```

Iteration Vs Recursion

- Consider a classic problem to compute the factorial of a number n
- $n! = 1 \times 2 \times \cdots \times (n-1) \times n$

```python
def fact(n):
    ans = 1
    for i in range(2, n+1):
        ans *= i
    return ans
```

- How much of work is done here?

Iteration Vs Recursion

- Consider a classic problem to compute the factorial of a number n
- $n! = 1 \times 2 \times \cdots \times (n-1) \times n$

```
def fact(n):
    ans = 1
    for i in range(2, n+1):
        ans *= i
    return ans
```

- How much of work is done here?
- $c(n-1) + d$

Iteration Vs Recursion

- Divide and conquer strategy leads to a simple recursive solution

Iteration Vs Recursion

- Divide and conquer strategy leads to a simple recursive solution
- $n! = n \times (n-1)!$

Iteration Vs Recursion

- Divide and conquer strategy leads to a simple recursive solution
- $n! = n \times (n-1)!$

```
def fact_rec(n):
    if (n < 2):
        return 1
    else:
        return n * fact_rec(n-1)
```

Iteration Vs Recursion

- Divide and conquer strategy leads to a simple recursive solution
- $n! = n \times (n-1)!$

```
def fact_rec(n):
    if (n < 2):
        return 1
    else:
        return n * fact_rec(n-1)
```

- There is a lot of clarity in the code, and it closely resembles the mathematical definition of factorial function

Iteration Vs Recursion

- Divide and conquer strategy leads to a simple recursive solution
- $n! = n \times (n-1)!$

```
def fact_rec(n):
    if (n < 2):
        return 1
    else:
        return n * fact_rec(n-1)
```

- There is a lot of clarity in the code, and it closely resembles the mathematical definition of factorial function
- How much work is done here?

- Divide and conquer strategy leads to a simple recursive solution
- $n! = n \times (n-1)!$

```
def fact_rec(n):
    if (n < 2):
        return 1
    else:
        return n * fact_rec(n-1)
```

- There is a lot of clarity in the code, and it closely resembles the mathematical definition of factorial function
- How much work is done here?

$$T(1) = d$$

$$T(n) = T(n-1) + c$$

What happens in recursion?

fact_rec(4)

What happens in recursion?

fact_rec(4)

$$= \quad 4 * \text{fact\_rec}(3)$$

What happens in recursion?

fact_rec(4)

$$
\begin{aligned}
&= 4 * \text{fact\_rec}(3) \\
&= 4 * (\ 3 * \text{fact\_rec}(2)\ )
\end{aligned}
$$

What happens in recursion?

fact_rec(4)

$$
\begin{aligned}
&= 4 * \text{fact\_rec}(3)\\
&= 4 * (\ 3 * \text{fact\_rec}(2)\ )\\
&= 4 * (\ 3 * (\ 2 * \text{fact\_rec}(1)\ )
\end{aligned}
$$

What happens in recursion?

fact_rec(4)

$$
\begin{aligned}
&= 4 * \text{fact\_rec}(3) \\
&= 4 * (\ 3 * \text{fact\_rec}(2)\ ) \\
&= 4 * (\ 3 * (\ 2 * \text{fact\_rec}(1)\ )
\end{aligned}
$$

[base case reached; recursive calls start returning]

What happens in recursion?

fact_rec(4)

$$
\begin{aligned}
&= 4 * \text{fact\_rec}(3) \\
&= 4 * (\ 3 * \text{fact\_rec}(2)\ ) \\
&= 4 * (\ 3 * (\ 2 * \text{fact\_rec}(1)\ )\ ) \\
&\quad \text{[base case reached; recursive calls start returning]} \\
&= 4 * (\ 3 * (\ 2 * 1\ )\ )
\end{aligned}
$$

What happens in recursion?

fact_rec(4)

$$
\begin{aligned}
&= 4 * \text{fact\_rec}(3) \\
&= 4 * (\ 3 * \text{fact\_rec}(2)\ ) \\
&= 4 * (\ 3 * (\ 2 * \text{fact\_rec}(1)\ )\ ) \\
&\ \text{[base case reached; recursive calls start returning]} \\
&= 4 * (\ 3 * (\ 2 * 1\ )\ ) \\
&= 4 * (\ 3 * 2\ )
\end{aligned}
$$

What happens in recursion?

fact_rec(4)

$$
\begin{aligned}
&= 4 * \text{fact\_rec}(3) \\
&= 4 * (\ 3 * \text{fact\_rec}(2)\ ) \\
&= 4 * (\ 3 * (\ 2 * \text{fact\_rec}(1)\ )\ ) \\
&\ \text{[base case reached; recursive calls start returning]} \\
&= 4 * (\ 3 * (\ 2 * 1\ )\ ) \\
&= 4 * (\ 3 * 2\ ) \\
&= 4 * 6
\end{aligned}
$$

What happens in recursion?

fact_rec(4)

= 4 * fact_rec(3)
= 4 * (3 * fact_rec(2))
= 4 * (3 * (2 * fact_rec(1))
[base case reached; recursive calls start returning]
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24

What happens in recursion?

fact_rec(4)

$$
\begin{aligned}
&= 4 * \text{fact\_rec}(3) \\
&= 4 * (\ 3 * \text{fact\_rec}(2)\ ) \\
&= 4 * (\ 3 * (\ 2 * \text{fact\_rec}(1)\ )\ ) \\
&\ \text{[base case reached; recursive calls start returning]} \\
&= 4 * (\ 3 * (\ 2 * 1\ )\ ) \\
&= 4 * (\ 3 * 2\ ) \\
&= 4 * 6 \\
&= 24
\end{aligned}
$$

- Function calls require context information to be stored in a stack
- Context gets popped out when the call returns
- Depth of recursion is limited by the size of the system stack!

Tail Recursion

- Tail recursion is a special case of recursion, where a recursive call appears as the last individual statement and not in an expression

Tail Recursion

- Tail recursion is a special case of recursion, where a recursive call appears as the last individual statement and not in an expression
- A good compiler may optimize a tail recursive function by not creating a new stack context!

Tail Recursion

- Tail recursion is a special case of recursion, where a recursive call appears as the last individual statement and not in an expression
- A good compiler may optimize a tail recursive function by not creating a new stack context!
- Tail recursion can be achieved by using "accumulator" pattern!

Tail Recursion

- Tail recursion is a special case of recursion, where a recursive call appears as the last individual statement and not in an expression
- A good compiler may optimize a tail recursive function by not creating a new stack context!
- Tail recursion can be achieved by using "accumulator" pattern!

```
def fact_tail(n):
    def fact_tail_rec(n, acc):
        if (n < 2):
            return acc
        else:
            return fact_tail_rec(n-1, n*acc)

    return fact_tail_rec(n, 1)
```

What happens in tail recursion?

fact_tail(4)

What happens in tail recursion?

fact_tail(4)

$$= \quad \text{fact\_tail\_rec}(4,\ 1)$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1)\\
&= \text{fact\_tail\_rec}(3,\ 4*1)
\end{aligned}
$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1)\\
&= \text{fact\_tail\_rec}(3,\ 4*1)\\
&= \text{fact\_tail\_rec}(2,\ 3 * 4)
\end{aligned}
$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1)\\
&= \text{fact\_tail\_rec}(3,\ 4*1)\\
&= \text{fact\_tail\_rec}(2,\ 3 * 4)\\
&= \text{fact\_tail\_rec}(1,\ 2 * 12)
\end{aligned}
$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1) \\
&= \text{fact\_tail\_rec}(3,\ 4*1) \\
&= \text{fact\_tail\_rec}(2,\ 3 * 4) \\
&= \text{fact\_tail\_rec}(1,\ 2 * 12)
\end{aligned}
$$

[base case reached; recursive calls start returning]

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1)\\
&= \text{fact\_tail\_rec}(3,\ 4{*}1)\\
&= \text{fact\_tail\_rec}(2,\ 3 * 4)\\
&= \text{fact\_tail\_rec}(1,\ 2 * 12)\\
&\ \text{[base case reached; recursive calls start returning]}\\
&= 24
\end{aligned}
$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1) \\
&= \text{fact\_tail\_rec}(3,\ 4*1) \\
&= \text{fact\_tail\_rec}(2,\ 3 * 4) \\
&= \text{fact\_tail\_rec}(1,\ 2 * 12) \\
&\quad \text{[base case reached; recursive calls start returning]} \\
&= 24 \\
&= 24
\end{aligned}
$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1) \\
&= \text{fact\_tail\_rec}(3,\ 4{*}1) \\
&= \text{fact\_tail\_rec}(2,\ 3 * 4) \\
&= \text{fact\_tail\_rec}(1,\ 2 * 12) \\
&\ \text{[base case reached; recursive calls start returning]} \\
&= 24 \\
&= 24 \\
&= 24
\end{aligned}
$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1)\\
&= \text{fact\_tail\_rec}(3,\ 4{*}1)\\
&= \text{fact\_tail\_rec}(2,\ 3 * 4)\\
&= \text{fact\_tail\_rec}(1,\ 2 * 12)\\
&\ \text{[base case reached; recursive calls start returning]}\\
&= 24\\
&= 24\\
&= 24\\
&= 24
\end{aligned}
$$

What happens in tail recursion?

fact_tail(4)

$$
\begin{aligned}
&= \text{fact\_tail\_rec}(4,\ 1) \\
&= \text{fact\_tail\_rec}(3,\ 4*1) \\
&= \text{fact\_tail\_rec}(2,\ 3 * 4) \\
&= \text{fact\_tail\_rec}(1,\ 2 * 12) \\
&\quad \text{[base case reached; recursive calls start returning]} \\
&= 24 \\
&= 24 \\
&= 24 \\
&= 24
\end{aligned}
$$

- Since recursive call is the last independent statement, no more work is needed when the call returns
- So, a compiler may be able to optimize this and execute the function in the same stack frame (like an iteration)
- However, a compiler may not perform this optimization (which is the case with most of the non-functional languages) and depth of recursion may be limited

Another Example: Fibonacci Series

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

Another Example: Fibonacci Series

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

```
def fib(n):
    current = 1; old = 0
    if (n == 0):
        return old
    elif (n == 1):
        return current
    else:
        for i in range(2, n+1):
            current, old = current+old, current
        return current
```

Recursive version

```
def fib_rec(n):
    if (n == 0):
        return 0
    elif (n < 3):
        return 1
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

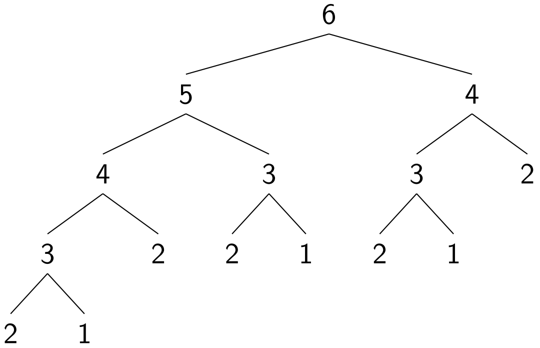
Recursive version

```
def fib_rec(n):
    if (n == 0):
        return 0
    elif (n < 3):
        return 1
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

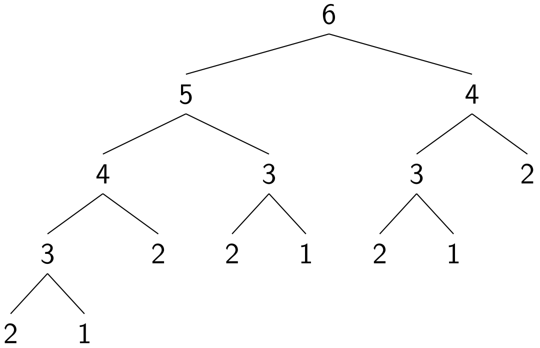
$$T(1) = T(2) = d$$

$$T(n) = T(n-1) + T(n-2) + c$$

Is recursion computationally expensive?

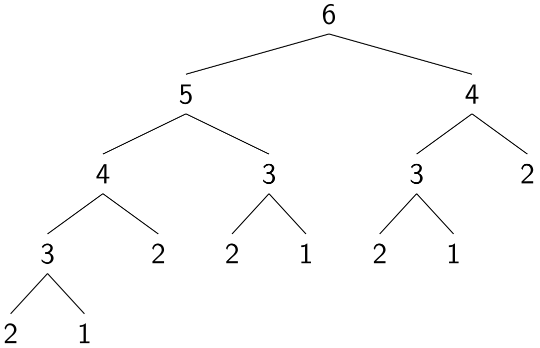


Is recursion computationally expensive?



Is it possible to rewrite this function with tail recursion?

Is recursion computationally expensive?



Is it possible to rewrite this function with tail recursion?

Hint: Use two accumulators!

Tail recursive version

```
def fib_tail(n):
    def fib_tail_rec(n, acc1, acc2):
        if (n < 3):
            return acc1
        else:
            return fib_tail_rec(n-1, acc1+acc2, acc1)

    if (n == 0):
        return 0
    elif (n < 3):
        return 1
    else:
        return fib_tail_rec(n, 1, 1)
```

Tail recursive version

```
def fib_tail(n):
    def fib_tail_rec(n, acc1, acc2):
        if (n < 3):
            return acc1
        else:
            return fib_tail_rec(n-1, acc1+acc2, acc1)

    if (n == 0):
        return 0
    elif (n < 3):
        return 1
    else:
        return fib_tail_rec(n, 1, 1)
```

$$T(1) = T(2) = d$$

$$T(n) = T(n-1) + c$$

What happens in tail recursion?

fib_tail(6)

What happens in tail recursion?

fib_tail(6)

$$= \quad \text{fib\_tail\_rec}(6, 1, 1)$$

What happens in tail recursion?

fib_tail(6)

$$
\begin{aligned}
&= \text{fib\_tail\_rec}(6, 1, 1) \\
&= \text{fib\_tail\_rec}(5, 2, 1)
\end{aligned}
$$

What happens in tail recursion?

fib_tail(6)

$$\begin{aligned} &= \text{fib\_tail\_rec}(6, 1, 1) \\ &= \text{fib\_tail\_rec}(5, 2, 1) \\ &= \text{fib\_tail\_rec}(4, 3, 2) \end{aligned}$$

What happens in tail recursion?

fib_tail(6)

$$
\begin{aligned}
&= \text{fib\_tail\_rec}(6, 1, 1)\\
&= \text{fib\_tail\_rec}(5, 2, 1)\\
&= \text{fib\_tail\_rec}(4, 3, 2)\\
&= \text{fib\_tail\_rec}(3, 5, 3)
\end{aligned}
$$

What happens in tail recursion?

fib_tail(6)

$$
\begin{aligned}
&= \text{fib\_tail\_rec}(6, 1, 1)\\
&= \text{fib\_tail\_rec}(5, 2, 1)\\
&= \text{fib\_tail\_rec}(4, 3, 2)\\
&= \text{fib\_tail\_rec}(3, 5, 3)\\
&= \text{fib\_tail\_rec}(2, 8, 5)
\end{aligned}
$$

What happens in tail recursion?

fib_tail(6)

$$
\begin{aligned}
&= \text{fib\_tail\_rec}(6, 1, 1)\\
&= \text{fib\_tail\_rec}(5, 2, 1)\\
&= \text{fib\_tail\_rec}(4, 3, 2)\\
&= \text{fib\_tail\_rec}(3, 5, 3)\\
&= \text{fib\_tail\_rec}(2, 8, 5)\\
&\quad \text{[base case reached; recursive calls start returning]}
\end{aligned}
$$

What happens in tail recursion?

fib_tail(6)

$$
\begin{aligned}
&= \text{fib\_tail\_rec}(6, 1, 1)\\
&= \text{fib\_tail\_rec}(5, 2, 1)\\
&= \text{fib\_tail\_rec}(4, 3, 2)\\
&= \text{fib\_tail\_rec}(3, 5, 3)\\
&= \text{fib\_tail\_rec}(2, 8, 5)\\
&\ \text{[base case reached; recursive calls start returning]}\\
&= 8
\end{aligned}
$$

What happens in tail recursion?

fib_tail(6)

$$
\begin{aligned}
&= \text{fib\_tail\_rec}(6, 1, 1) \\
&= \text{fib\_tail\_rec}(5, 2, 1) \\
&= \text{fib\_tail\_rec}(4, 3, 2) \\
&= \text{fib\_tail\_rec}(3, 5, 3) \\
&= \text{fib\_tail\_rec}(2, 8, 5) \\
&\quad \text{[base case reached; recursive calls start returning]} \\
&= 8
\end{aligned}
$$

- Note that tail recursion has completely changed the complexity class!

What happens in tail recursion?

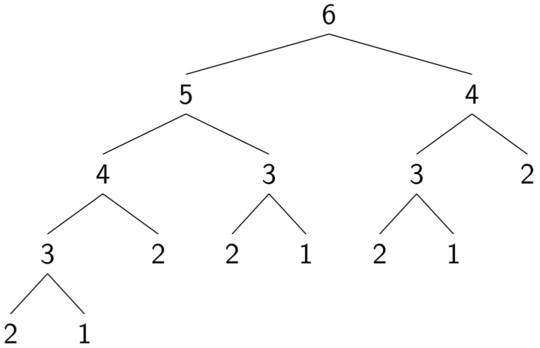
fib_tail(6)

$$
\begin{aligned}
&= \text{fib\_tail\_rec}(6, 1, 1)\\
&= \text{fib\_tail\_rec}(5, 2, 1)\\
&= \text{fib\_tail\_rec}(4, 3, 2)\\
&= \text{fib\_tail\_rec}(3, 5, 3)\\
&= \text{fib\_tail\_rec}(2, 8, 5)\\
& \text{[base case reached; recursive calls start returning]}\\
&= 8
\end{aligned}
$$

- Note that tail recursion has completely changed the complexity class!
- So, even if a compiler does not perform any stack optimization, tail recursion still helps!

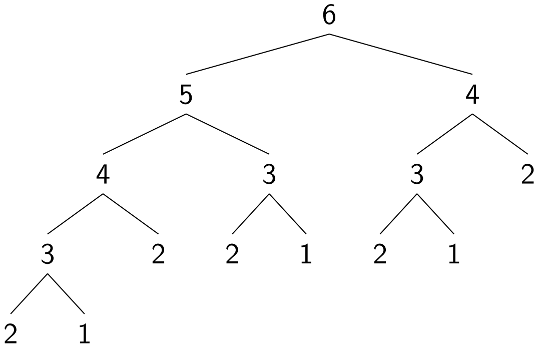
Dynamic Programming

- Let us again look at what happens with standard recursion



Dynamic Programming

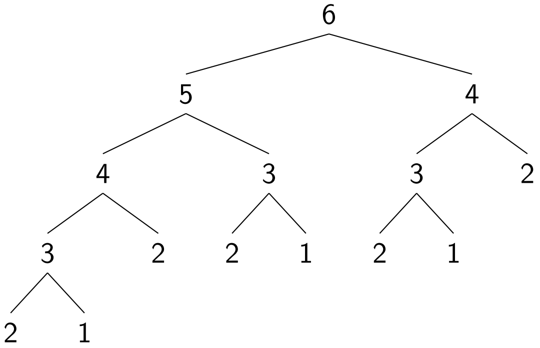
- Let us again look at what happens with standard recursion



There is a lot of repetitive work!

Dynamic Programming

- Let us again look at what happens with standard recursion



There is a lot of repetitive work!

Is it possible to store intermediate results to avoid repetitive computation?

Memoization

```python
known = {0:0, 1:1, 2:1}

def fib_dict(n):
    if n in known:
        return known[n]
    res = fib_dict(n-1) + fib_dict(n-2)
    known[n] = res
    return res
```

Memoization

```
known = {0:0, 1:1, 2:1}

def fib_dict(n):
    if n in known:
        return known[n]
    res = fib_dict(n-1) + fib_dict(n-2)
    known[n] = res
    return res
```

Point to note: Time Vs Space Trade-off

Memoization

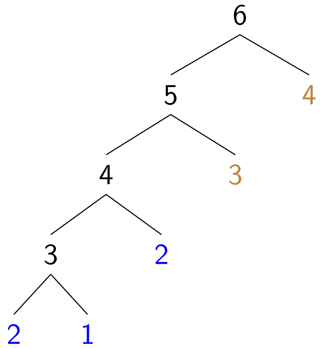
```
known = {0:0, 1:1, 2:1}

def fib_dict(n):
    if n in known:
        return known[n]
    res = fib_dict(n-1) + fib_dict(n-2)
    known[n] = res
    return res
```

Point to note: Time Vs Space Trade-off
Another point to note: How much time is required to check for an entry in the look-up table?

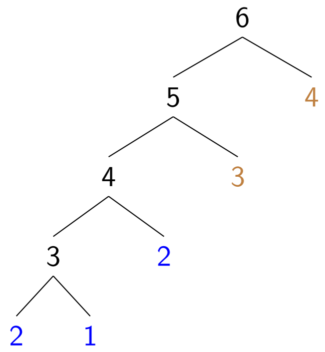
Dynamic Programming

- What happens with dynamic programming?



Dynamic Programming

- What happens with dynamic programming?



- Exponential time execution may be turned in to linear time!
- But space requirement grows with n (in this case it is linear growth)
- When we use dictionary like structure, it may be possible to dynamically manage which results need to be persisted with

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

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

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

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

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

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

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}$$

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

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

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

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

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\vdots$$

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

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

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\vdots$$

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

Formulate as matrix multiplication problem

$$\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = M \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

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

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}$$

$$\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\vdots$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Fibonacci through matrix multiplication

```
import numpy as np

def fib_mat(n):
    if (n == 0):
        return 0
    elif (n < 3):
        return 1
    else:
        return ((np.matrix('1 1; 1 0', dtype='object')
                    ** (n-1)).item(0)
            )
```

Fibonacci through matrix multiplication

```
import numpy as np

def fib_mat(n):
    if (n == 0):
        return 0
    elif (n < 3):
        return 1
    else:
        return((np.matrix('1␣1;␣1␣0', dtype='object')
                        ** (n-1)).item(0)
            )
```

What is the complexity of computing $M^{(n-1)}$?

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done?

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done? $O(n)$

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done? $O(n)$
- Is it possible to do better? Think of alternate decomposition.

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done? $O(n)$
- Is it possible to do better? Think of alternate decomposition.
- Consider an easier case:

$$M^{16} = (M^8)^2 = ((M^4)^2)^2 = (((M^2)^2)^2)^2$$

- Just 4 multiplications instead of 16!

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done? $O(n)$
- Is it possible to do better? Think of alternate decomposition.
- Consider an easier case:

$$M^{16} = (M^8)^2 = ((M^4)^2)^2 = (((M^2)^2)^2)^2$$

- Just 4 multiplications instead of 16!
- What can be done if n is not a power of 2?

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done? $O(n)$
- Is it possible to do better? Think of alternate decomposition.
- Consider an easier case:

$$M^{16} = (M^8)^2 = ((M^4)^2)^2 = (((M^2)^2)^2)^2$$

- Just 4 multiplications instead of 16!
- What can be done if n is not a power of 2?
- Decompose in terms of powers of 2!

$$M^{25} = M^{16}.M^8.M$$

$$M^{15} = M^8.M^4.M^2.M$$

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done? $O(n)$
- Is it possible to do better? Think of alternate decomposition.
- Consider an easier case:

$$M^{16} = (M^8)^2 = ((M^4)^2)^2 = (((M^2)^2)^2)^2$$

- Just 4 multiplications instead of 16!
- What can be done if n is not a power of 2?
- Decompose in terms of powers of 2!

$$M^{25} = M^{16}.M^8.M$$

$$M^{15} = M^8.M^4.M^2.M$$

- Can we generalize this and find out how to decompose M^n?

How do we effectively compute exponentiation?

- How do we effectively compute M^n ?
- Simple divide and conquer: $M^n = M \times M^{n-1}$
- How many multiplications are done? $O(n)$
- Is it possible to do better? Think of alternate decomposition.
- Consider an easier case:

$$M^{16} = (M^8)^2 = ((M^4)^2)^2 = (((M^2)^2)^2)^2$$

- Just 4 multiplications instead of 16!
- What can be done if n is not a power of 2?
- Decompose in terms of powers of 2!

$$M^{25} = M^{16}.M^8.M$$

$$M^{15} = M^8.M^4.M^2.M$$

- Can we generalize this and find out how to decompose M^n?
- Look at the binary bit pattern of n!!!

Right to left evaluation

1	1	0	0	1	
M^{16}	M^8	M^4	M^2	M	term
$*$	$*$			$*$	result

Right to left evaluation

1	1	0	0	1	
M^{16}	M^8	M^4	M^2	M	term
*	*			*	result

- Initialize term to M
- Initialize result to M if $b_0 = 1$, otherwise initialize to 1
- Iterate over the bits b_1 to b_k:
  - term = term * term
  - If $b_i = 1$, then result = result * term

Exercise

- It is also possible to do left to right evaluation of the bit pattern of n.
- Read about Horner's rule for evaluating a polynomial
- Read about how to use that idea for left to right evaluation of bit pattern to compute M^n
- Implement (in Python, without using Numpy or any such packages) both the left-to-right and right-to-left evaluations for computing M^n
- Re-implement the computation of n^{th} Fibonacci number using your exponentiation functions