Python: Recursive Functions

Recursive Functions

Recall factorial function:

```
def factorial(n):
   i=0
   fact=1
   while (i<n):
      i=i+1
      fact=fact*i
   return fact
```

Iterative Algorithm

Loop construct (while) can capture computation in a set of **state variables** that update on each iteration through loop

Recursive Functions

Alternatively:

Consider

$5! = 5 \times 4 \times 3 \times 2 \times 1$

can be re-written as $5! = 5 \times 4!$

In general $n! = n \times (n-1)!$

factorial(n) = n * factorial(n-1)

Recursive Functions

Alternatively:

Consider

5! = 5x4x3x2x1

can be re-written as 5!=5x4!

Recursive Algorithm

function calling itself

In general n! = nx(n-1)!

factorial(n) = n * factorial(n-1)

Recursive Functions

Recursion involves two steps.

Step 1. Keep on reducing the size of the problem till we reach a point where the solution is known.

Factorial(5) = 5 x Factorial(4)
= 5 x 4 x Factorial(3)
= 5 x 4 x 3 x Factorial (2)
= 5 x 4 x 3 x 2 x Factorial(1)
= 5 x 4 x 3 x 2 x 1 x Factorial(0)

It is known that Factorial(0) is 1

Step 2. Now work backwards by calling the functions again but this time with known values

Recursive Functions

Now we can solve Factorial(1) as

Factorial(1) = 1 x Factorial(0) = 1

Known
Base case

Similarly,
Factorial(2) = 2 x Factorial(1) = 2
Factorial(3) = 3 x Factorial(2) = 6
Factorial(4) = 4 x Factorial(3) = 24

Finally,
Factorial(5) = 5 x Factorial(4) = 120

Recursive Functions

```
def factorial( n):
    if (n == 0):
        return 1
    else:
        return n * factorial(n – 1)
```

Base case

Recursive step

Recursive Functions

- No computation in first phase, only function calls
- Deferred/Postponed computation
  - after function calls terminate, computation starts
- Sequence of calls have to be remembered

```
fact (4)
4 * fact (3)
4 * (3 * fact (2))
4 * (3 * (2 * fact (1)))
4 * (3 * (2 * (1 * fact (0))))
4 * (3 * (2 * (1 * 1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Execution trace for n = 4

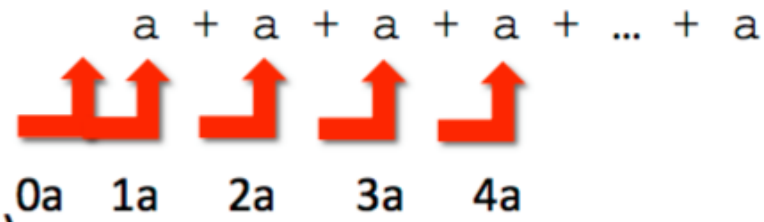
Courtesy Prof PR Panda CSE Department IIT Dehi

Another Example (Iterative)

- "multiply a * b" is equivalent to "add a to itself b times"
- capture **state** by
  - an **iteration** number (i) starts at b

    i ← i-1 and stop when 0
  - a current **value of computation** (result)

    result ← result + a



Iterative

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

iteration

current value of computation, a running sum

current value of iteration variable

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Another Example (Recursive)

- **recursive step**
  - think how to reduce problem to a **simpler/ smaller version** of same problem
- **base case**
  - keep reducing problem until reach a simple case that can be **solved directly**
  - when b = 1, a*b = a

$$a*b = \underbrace{a + a + a + a + \ldots + a}_{\text{b times}}$$

$$= a + \underbrace{a + a + a + a + \ldots + a}_{\text{b-1 times}}$$

$$= a + \boxed{a * (b-1)}$$

recursive reduction

Recursive

```python
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

base case

recursive step

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/

Recursive Functions

- Size of the problem reduces at each step
- The nature of the problem remains the same
- There must be at least one terminating condition

- Simpler, more intuitive
  - For inductively defined computation, recursive algorithm may be natural
    - close to mathematical specification
- Easy from programing point of view
- May not efficient computation point of view

GCD Algorithm

$$\gcd(a, b) = \begin{cases} b & \text{if } a \bmod b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

```
def gcd(a,b):
 r=a%b
 while (r !=0 ):
   a=b
   b=r
   r=a%b
 return b
```

Iterative Algorithm

```
def gcd(a,b):
   if (a%b == 0):
     return b
   else:
     return gcd(b,a%b)
```

Recursive Algorithm

GCD Algorithm

$$\text{gcd}(a, b) = \begin{cases} b & \text{if } a \bmod b = 0 \\ \text{gcd}(b, a \bmod b) & \text{otherwise} \end{cases}$$

```
def gcd(a,b):
    if (a%b == 0):
      return b
    else:
      return gcd(b,a%b)
```

```
gcd (6, 10)
  gcd (10, 6)
    gcd (6, 4)
      gcd (4, 2)
      2
    2
  2
2
```

Fibonacci Numbers

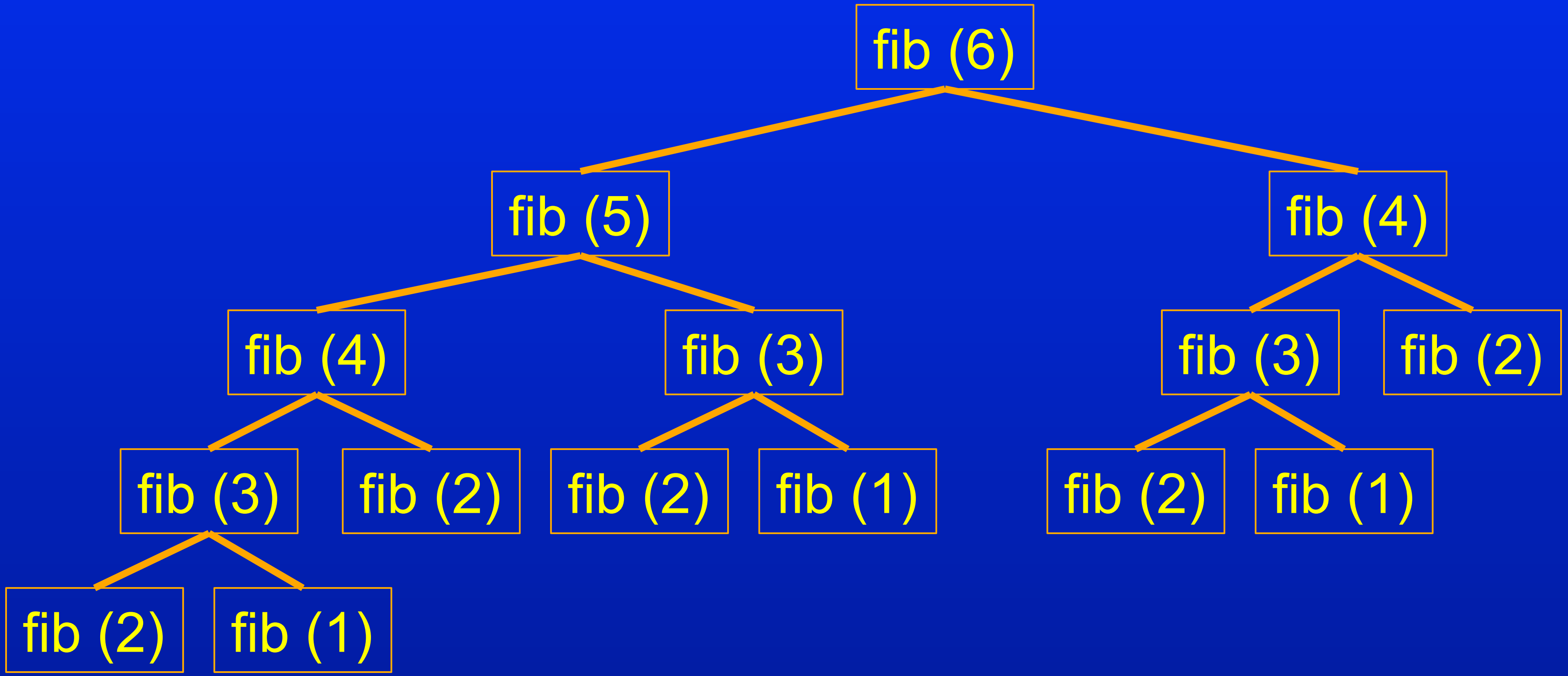
$$
\text{fib}(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \text{fib}(n-1) + \text{fib}(n-2) & n > 2 \end{cases}
$$

Recursive Algorithm

```
def fib(n):
  if (n==1):
    return 0
  if (n==2):
    return 1
  else:
    return fib(n-1)+fib(n-2)
```

Courtesy Prof PR Panda CSE Department IIT Dehi

Fibonacci Numbers



Courtesy Prof PR Panda CSE Department IIT Dehi

Power Function

- Write a function power (x,n) to compute the n^{th} power of x

$$\text{power}(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x * \text{power}(x,n-1) & \text{otherwise} \end{cases}$$

```
def power(x,n):
  if (n==0):
    return 1
  else:
     return x*power(x,n-1)
```

Power Function

- Efficient power function

- Fast Power
  - fpower(x,n) = 1 for n = 0
  - fpower(x,n) = x * (fpower(x, n/2))^2 if n is odd
  - fpower(x,n) = (fpower(x, n/2))^2 if n is even

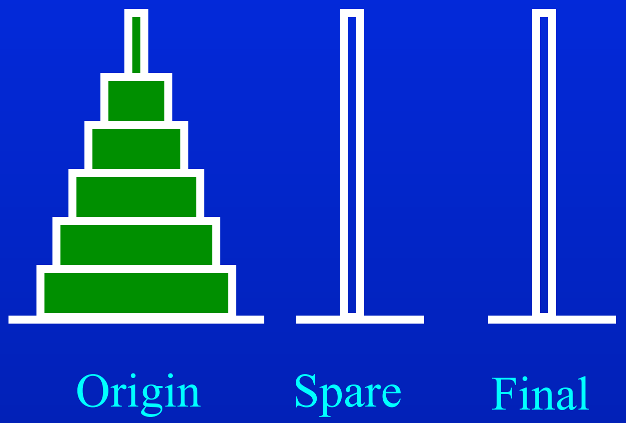
Power Function

- Efficient power function

```
def fpower(x, n):
 if (n==0):
    return 1
 else:
  y = fpower(x,int(n/2))
  if (n%2 == 0):
   return y*y
  else:
   return x*y*y
```

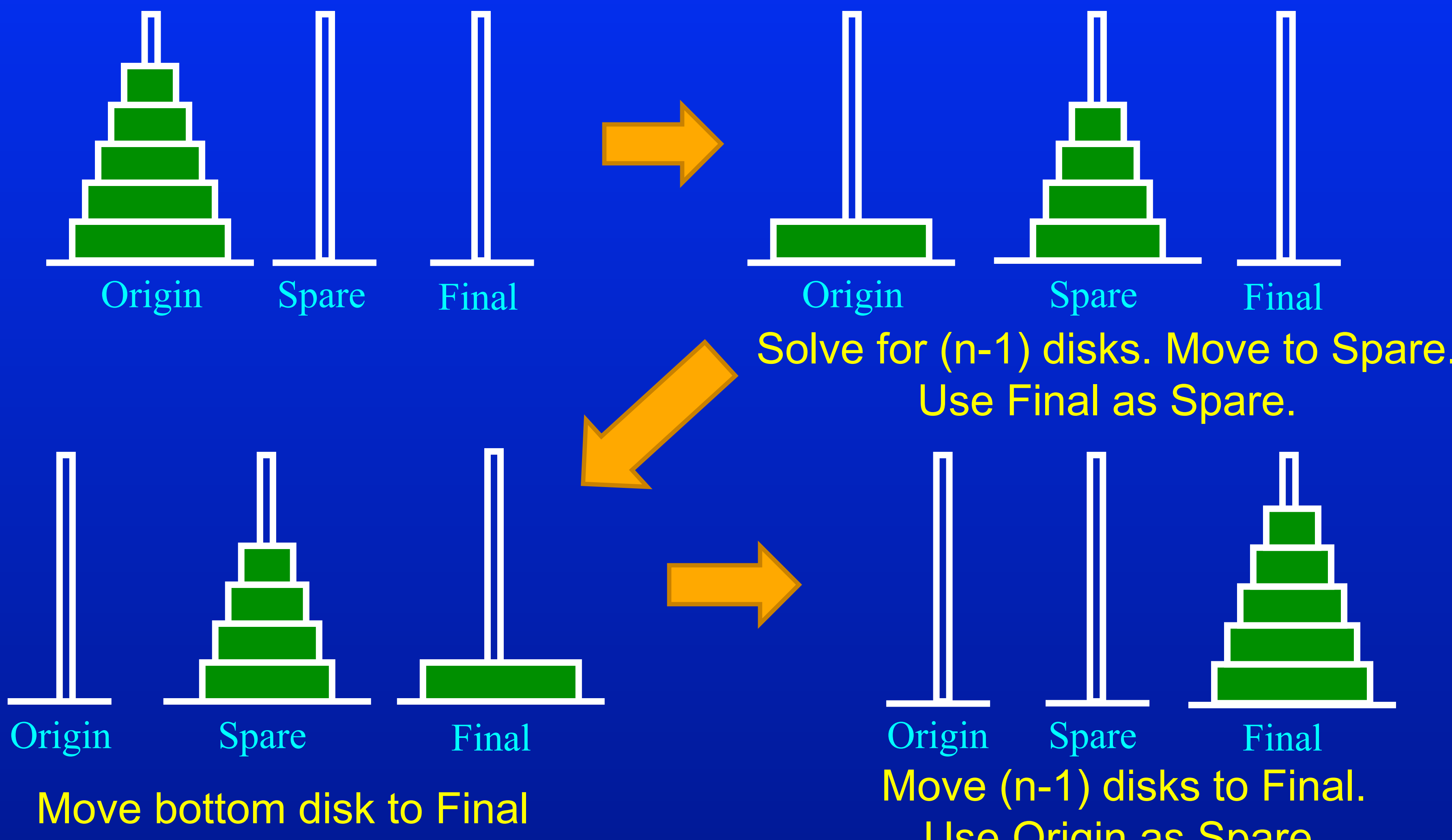
Towers of Hanoi Problem

- 64 gold discs with different diameters
- Three poles: (Origin, Spare, Final)
- Transfer all discs to final pole from origin
  - one at a time
  - spare can be used to temporarily store discs
  - no disk should be placed on a smaller disk
- Intial Arrangement:
  - all on origin pole, largest at bottom, next above it, etc.



Courtesy Prof PR Panda CSE Department IIT Dehi

3-Step Strategy



Courtesy Prof PR Panda CSE Department IIT Dehi

Recursive Solution

- Use algorithm for (n-1) disks to solve n-disk problem
- Use algorithm for (n-2) disks to solve (n-1) disk problem
- Use algorithm for (n-3) disks to solve (n-2) disk problem
- ...
- Finally, solve 1-disk problem:
  – Just move the disk!

Recursive Solution

```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))


def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

Source:https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/