THE UNIVERSITY OF WESTERN AUSTRALIA

Lecture 28
Recursion vs Iteration

□ □
迭代

Objectives

- To revise concept of recursion.
- To compare the performance of recursion and iteration.
- To understand when to use recursion and when to use iteration.

Revision: Recursive Definitions

- A description of something that refers to itself is called a *recursive* definition.
- All good recursive definitions have these two key characteristics:

  1. *There are one or more base cases for which no recursion is applied.*
  2. *All chains of recursion eventually end up at one of the base cases.*

  Put differently, each iteration must drive the computation toward a base case.

Recursion vs. Iteration

- There are similarities between iteration (looping) and recursion
- In fact, anything that can be done with a loop can be done with a simple recursive function!
  - *But some algorithms harder to set up with iteration*
- Some programming languages use recursion exclusively.
  - *Haskell, ML, Lisp (functional programming); Prolog (logic programming)*
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.

Recursion vs. Iteration

- In the factorial and binary search problems, the looping and recursive solutions use roughly the same algorithms, and their efficiency is nearly the same.
- Lets take another example: Fast Exponentiation

n.□[□]□□□， □□； □□

Example: Fast Exponentiation

- One way to compute a^n for an integer n is to multiply a by itself n times.
- This can be done with a simple accumulator loop:

```
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans *= a
    return ans
```

Example: Fast Exponentiation

- We can also solve this problem using recursion and divide & conquer approach.
- Using the laws of exponents, we know that $2^8 = 2^4 \times 2^4$. If we know 2^4, we can calculate 2^8 using one multiplication.
- What's 2^4? $2^4 = 2^2 \times 2^2$, and $2^2 = 2 \times 2$.
- $2 \times 2 = 4$, $2^2 \times 2^2 = 16$, $2^4 \times 2^4 = 256 = 2^8$
- We've calculated 2^8 using only three multiplications!

Example: Fast Exponentiation

- We can take advantage of the fact that $a^n = a^{n//2}(a^{n//2})$
- This algorithm only works when *n* is even. How can we extend it to work when *n* is odd?
- $2^9 = 2^4 \times 2^4 \times 2^1$

$$a^n = \begin{cases} a^{n//2}(a^{n//2}) & \text{if } n \text{ is even} \\ a^{n//2}(a^{n//2})(a) & \text{if } n \text{ is odd} \end{cases}$$

Example: Fast Exponentiation

- This method relies on integer division (if n is 9, then $n//2 = 4$).
- To express this algorithm recursively, we need a suitable base case.
- If we keep using smaller and smaller values for n, n will eventually be equal to 0 ($1//2 = 0$), and $a^0 = 1$ for any value except $a = 0$.

Example: Fast Exponentiation

```
# raises a to the int power n
def recPower(a, n):
  if n ==  0:
    return 1
  factor = recPower(a, n//2)
  if n%2 == 0:     # n is even
    return factor * factor
   # n is odd
  return factor * factor * a
```

- Here, a temporary variable called `factor` is introduced so that we don't need to calculate $a^{n//2}$ more than once, simply for efficiency.

Recursion vs. Iteration

- In the exponentiation problem:
  - *The iterative version takes linear time to complete*
  - *The recursive version executes in log time.*
  - *The difference between them is like the difference between a linear and binary search.*
- So… will recursive solutions always be as efficient or more efficient than their iterative counterpart?
- It depends

Recursion vs. Iteration

- The Fibonacci sequence is the sequence of numbers 1,1,2,3,5,8,…
  - *The sequence starts with two 1's*
  - *Successive numbers are calculated by finding the sum of the previous two numbers.*

Recursion vs. Iteration

```
def loopfib(n):
    # returns the nth Fibonacci number
    curr = 1
    prev = 1
    for i in range(n-2):
        curr, prev = curr+prev, curr
    return curr
```

- Note the use of simultaneous assignment to calculate the new values of `curr` and `prev`.
- The loop executes only *n-2* times since the first two values have already been provided as a starting point.

Recursion vs. Iteration

- The Fibonacci sequence also has a recursive definition:

$$fib(n) = \begin{cases} 1 & \text{if } n < 3 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

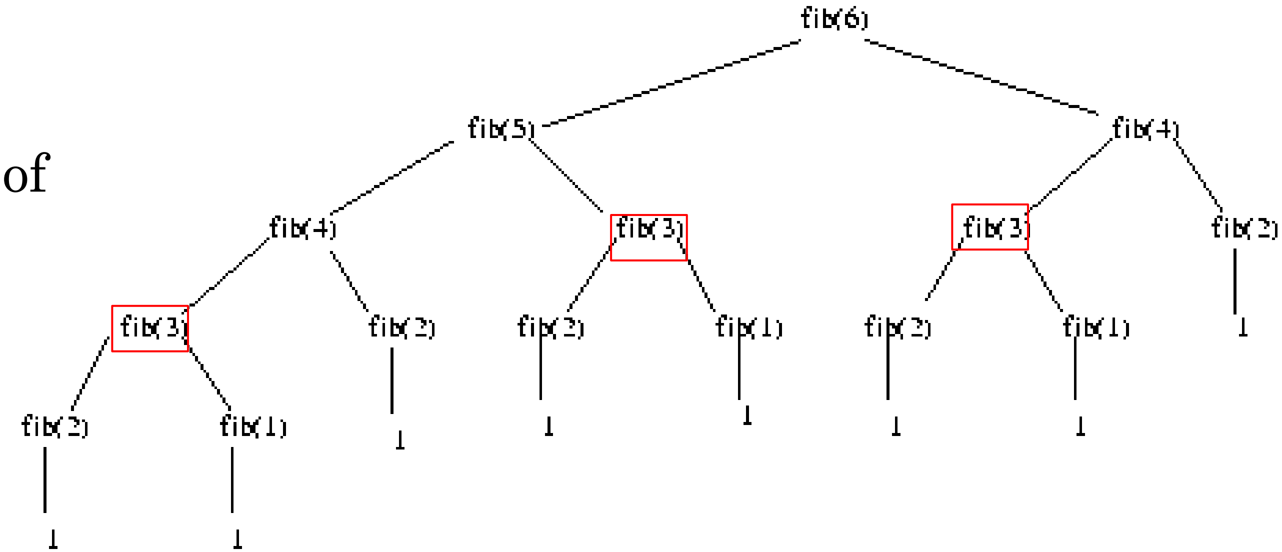
- This recursive definition can be directly turned into a recursive function!

```
def fib(n):
    if n < 3:
        return 1
    return fib(n-1)+fib(n-2)
```

Recursion vs. Iteration

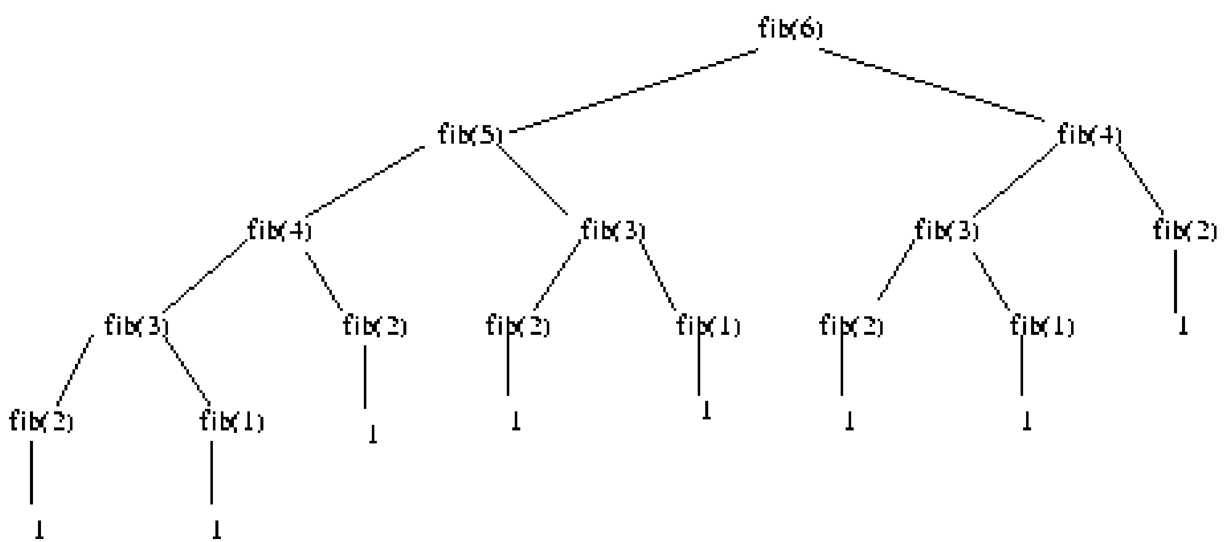
- This function obeys the rules that we've set out.
  - *The recursion is always based on smaller values.*
  - *There is a non-recursive base case.*
- So, this function will work great, won't it? – *Sort of…*
- The recursive solution is extremely inefficient, since it performs many duplicate calculations!

Recomputing of fib(3) shown



Recursion vs. Iteration

- To calculate fib(6), fib(4)is calculated twice, fib(3)is calculated three times, fib(2)is calculated four times… For large numbers, this adds up!



Recursion vs. Iteration

- Recursion is another tool in your problem-solving toolbox.
- Sometimes recursion provides a good solution because it is more elegant or efficient than a looping version.
- At other times, when both algorithms are quite similar, the edge goes to the looping solution on the basis of speed and (generally) simplicity of programming
- Avoid the recursive solution if it is terribly inefficient, unless you can't come up with an iterative solution (which sometimes happens!)

Summary

- We analyzed the recursion's performance and compared it to iterations (loops).
- We learned when to use recursion and when to use loops.