Recursion versus Iteration

1. The Towers of Hanoi
   recursive problem solving
   a recursive Python function
   tracing: exponential time

2. The Fibonacci Numbers
   a simple recursion
   an iterative algorithm

3. Memoization
   exponential complexity and cost
   an efficient recursive Fibonacci

MCS 275 Lecture 10
Programming Tools and File Management
Jan Verschelde, 3 February 2010
Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - tracing: exponential time

2. The Fibonacci Numbers
   - a simple recursion
   - an iterative algorithm

3. Memoization
   - exponential complexity and cost
   - an efficient recursive Fibonacci
The Towers of Hanoi
an ancient mathematical puzzle

**Input:** disks on a pile, all of varying size, no larger disk sits above a smaller disk, and two other empty piles.

**Task:** move the disks from the first pile to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third pile as buffer.
The Towers of Hanoi
an ancient mathematical puzzle

**Input:** disks on a pile, all of varying size, no larger disk sits above a smaller disk, and two other empty piles.

**Task:** move the disks from the first pile to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third pile as buffer.
The Towers of Hanoi
an ancient mathematical puzzle

Input: disks on a pile, all of varying size, no larger disk sits above a smaller disk, and two other empty piles.

Task: move the disks from the first pile to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third pile as buffer.
The Towers of Hanoi
an ancient mathematical puzzle

**Input:** disks on a pile, all of varying size, no larger disk sits above a smaller disk, and two other empty piles.

**Task:** move the disks from the first pile to the second, obeying the following rules:
1. move one disk at a time,
2. never place a larger disk on a smaller one, you may use the third pile as buffer.
A recursive Solution

Assume we know how to move a stack with one disk less.

```
A  B  C
A  B  C
A  B  C
A  B  C
A  B  C
```
A recursive Solution

Assume we know how to move a stack with one disk less.
A recursive Solution

Assume we know how to move a stack with one disk less.

\[ \text{A} \quad \text{B} \quad \text{C} \quad \rightarrow \quad \text{A} \quad \text{B} \quad \text{C} \]

\[ \text{A} \quad \text{B} \quad \text{C} \quad \rightarrow \quad \text{A} \quad \text{B} \quad \text{C} \]

\[ \text{A} \quad \text{B} \quad \text{C} \quad \rightarrow \quad \text{A} \quad \text{B} \quad \text{C} \]

\[ \text{A} \quad \text{B} \quad \text{C} \]
A recursive Solution

Assume we know how to move a stack with one disk less.
A recursive Solution

Assume we know how to move a stack with one disk less.
A recursive Algorithm

Base case: move one disk from A to B.

To move $n$ disks from A to B:

1. Move $n-1$ disks from A to C using B as auxiliary pile.
2. Move $n$-th disk from A to B.
3. Move $n-1$ disks from C to B using A as auxiliary pile.
A recursive Algorithm

Base case: move one disk from A to B.
To move $n$ disks from A to B:

Move $n-1$ disks from A to C using B as auxiliary pile

Move $n$-th disk from A to B

Move $n-1$ disks from C to B using A as auxiliary pile
A recursive Algorithm

Base case: move one disk from A to B.
To move \( n \) disks from A to B:

- Move \( n - 1 \) disks from A to C using B as auxiliary pile.
- Move \( n \)-th disk from A to B.
- Move \( n - 1 \) disks from C to B using A as auxiliary pile.
A recursive Algorithm

Base case: move one disk from A to B.
To move $n$ disks from A to B:

- Move $n-1$ disks from A to C using B as auxiliary pile
- Move $n$-th disk from A to B
- Move $n-1$ disks from C to B using A as auxiliary pile
Recursion versus Iteration

1. The Towers of Hanoi
   recursive problem solving
   a recursive Python function
   tracing: exponential time

2. The Fibonacci Numbers
   a simple recursion
   an iterative algorithm

3. Memoization
   exponential complexity and cost
   an efficient recursive Fibonacci
Lists as Stacks

In a stack, we remove only the top element \((\text{pop})\), and add only at the top \((\text{push})\).

A pile of 4 disks of decreasing size:

```python
>>> A = range(1,5)
[1, 2, 3, 4]
```

To remove the top element:

```python
>>> A.pop(0)
1
[2, 3, 4]
```

To put an element on top:

```python
>>> A.insert(0,1)
[1, 2, 3, 4]
```
Lists as Stacks

In a stack, we remove only the top element (*pop*), and add only at the top (*push*).

A pile of 4 disks of decreasing size:

```python
>>> A = range(1, 5)
>>> A
[1, 2, 3, 4]
```

To remove the top element:

```python
>>> A.pop(0)
1
>>> A
[2, 3, 4]
```

To put an element on top:

```python
>>> A.insert(0, 1)
[1, 2, 3, 4]
```
Lists as Stacks

In a stack, we remove only the top element (*pop*), and add only at the top (*push*).

A pile of 4 disks of decreasing size:

```python
>>> A = range(1,5)
>>> A
[1, 2, 3, 4]
To remove the top element:

```python
>>> A.pop(0)
1
>>> A
[2, 3, 4]
```

To put an element on top:

```python
>>> A.insert(0,1)
[1, 2, 3, 4]
```
Lists as Stacks

In a stack, we remove only the top element (*pop*), and add only at the top (*push*).

A pile of 4 disks of decreasing size:

```python
>>> A = range(1,5)
>>> A
[1, 2, 3, 4]
```

To remove the top element:

```python
>>> A.pop(0)
1
>>> A
[2, 3, 4]
```

To put an element on top:

```python
>>> A.insert(0,1)
[1, 2, 3, 4]
```
A recursive Python Function

def Hanoi(n,A,B,C):
    """
    moves n disks from A to B, C is auxiliary
    returns the tuple (A,B,C)
    """

    if n == 1:
        # move disk from A to B
        B.insert(0,A.pop(0))
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A,C,B) = Hanoi(n-1,A,C,B)
        # move n-th disk from A to B
        B.insert(0,A.pop(0))
        # move n-1 disks from C to B, A is auxiliary
        (C,B,A) = Hanoi(n-1,C,B,A)
    return (A,B,C)
A recursive Python Function

def Hanoi(n, A, B, C):
    """
    moves n disks from A to B, C is auxiliary
    returns the tuple (A, B, C)
    """

    if n == 1:
        # move disk from A to B
        B.insert(0, A.pop(0))
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A, C, B) = Hanoi(n-1, A, C, B)
        # move n-th disk from A to B
        B.insert(0, A.pop(0))
        # move n-1 disks from C to B, A is auxiliary
        (C, B, A) = Hanoi(n-1, C, B, A)
    return (A, B, C)
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Memoization
exponential complexity and cost
an efficient recursive Fibonacci

A recursive Python Function

def Hanoi(n, A, B, C):
    """
    moves n disks from A to B, C is auxiliary
    returns the tuple (A, B, C)
    """
    if n == 1:
        # move disk from A to B
        B.insert(0, A.pop(0))
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A, C, B) = Hanoi(n-1, A, C, B)
        # move n-th disk from A to B
        B.insert(0, A.pop(0))
        # move n-1 disks from C to B, A is auxiliary
        (C, B, A) = Hanoi(n-1, C, B, A)
    return (A, B, C)
The Towers of Hanoi

recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers

a simple recursion
an iterative algorithm

Memoization

exponential complexity and cost
an efficient recursive Fibonacci

A recursive Python Function

```python
def Hanoi(n,A,B,C):
    """
    moves n disks from A to B, C is auxiliary
    returns the tuple (A,B,C)
    """
    if n == 1:
        # move disk from A to B
        B.insert(0,A.pop(0))
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A,C,B) = Hanoi(n-1,A,C,B)
        # move n-th disk from A to B
        B.insert(0,A.pop(0))
        # move n-1 disks from C to B, A is auxiliary
        (C,B,A) = Hanoi(n-1,C,B,A)
    return (A,B,C)
```
def Hanoi(n,A,B,C):
    """
    moves n disks from A to B, C is auxiliary
    returns the tuple (A,B,C)
    """
    if n == 1:
        # move disk from A to B
        B.insert(0,A.pop(0))
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A,C,B) = Hanoi(n-1,A,C,B)
        # move n-th disk from A to B
        B.insert(0,A.pop(0))
        # move n-1 disks from C to B, A is auxiliary
        (C,B,A) = Hanoi(n-1,C,B,A)
    return (A,B,C)
Recursion versus Iteration

1. The Towers of Hanoi
   recursive problem solving
   a recursive Python function
   tracing: exponential time

2. The Fibonacci Numbers
   a simple recursion
   an iterative algorithm

3. Memoization
   exponential complexity and cost
   an efficient recursive Fibonacci
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
moving the disk 1 from A to C
moving the disk 2 from A to B
moving the disk 1 from A to C
moving the disk 3 from A to B
moving the disk 1 from A to B
moving the disk 2 from B to C
moving the disk 1 from A to C
move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
moving the disk 4 from B to A
moving the disk 1 from C to A
moving the disk 2 from B to C
moving the disk 1 from A to B
move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
moving the disk 3 from C to A
moving the disk 1 from A to C
moving the disk 2 from B to C
move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
moving the disk 1 from C to A
$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
  move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
  move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
        move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
            move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
                move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
  move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
  move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
The Towers of Hanoi

recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers

a simple recursion
an iterative algorithm

Memoization

exponential complexity and cost
an efficient recursive Fibonacci

Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
  move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
  move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
Tracing the Execution

$ python hanoi.py
Give number of disks : 4

at start : A = [1, 2, 3, 4] B = [] C = []
moves 6, n = 2 : B = [] C = [2, 3] A = [1, 4]
moves 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
moves 9, n = 1 : C = [2, 3] B = [1, 4] A = []
moves 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
moves 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Tracing the Execution

```python
$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
  move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
  move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
```
Tracing the Execution

```
python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
  move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
  move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
```
$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
  move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
    move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
          move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
          move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
Tracing the Execution

```
$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
  move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
  move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
  move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
```
Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
$ python hanoi.py

Give number of disks : 4

at start : A = [1, 2, 3, 4] B = [] C = []


move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]


move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]


move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
Tracing the Execution

$ python hanoi.py
Give number of disks : 4
at start : A = [1, 2, 3, 4] B = [] C = []
  move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
    move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
      move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
The Towers of Hanoi recursive problem solving
a recursive Python function tracing: exponential time

The Fibonacci Numbers a simple recursion an iterative algorithm

Memoization exponential complexity and cost an efficient recursive Fibonacci

Extra Code for Tracing recursion level, count #moves

```python
def Hanoi(n,A,B,C,k,m):
    """
    moves n disks from A to B, C is auxiliary
    k is recursion level, m counts # moves
    writes status of piles after each move
    returns the tuple (A,B,C,m)
    """

in main():

    n = input('Give number of disks : ')
    A = ('A',range(1,n+1))
    B = ('B',[])
    C = ('C',[])
    (A,B,C,m) = Hanoi(n,A,B,C,0,0)

As the roles of the piles shift, we need to maintain their names when printing their contents.
Extra Code for Tracing

recursion level, count #moves

def Hanoi(n, A, B, C, k, m):
    """
    moves n disks from A to B, C is auxiliary
    k is recursion level, m counts # moves
    writes status of piles after each move
    returns the tuple (A, B, C, m)
    """

in main():
    n = input('Give number of disks : ')
    A = ('A', range(1, n+1))
    B = ('B', [])
    C = ('C', [])
    (A, B, C, m) = Hanoi(n, A, B, C, 0, 0)

As the roles of the piles shift, we need to maintain their names when printing their contents.
def Hanoi(n, A, B, C, k, m):
    
    if n == 1:
        # move disk from A to B
        m = m + 1
        B[1].insert(0, A[1].pop(0))
        write(k, m, n, A, B, C)
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A, C, B, m) = Hanoi(n-1, A, C, B, k+1, m)
        # move n-th disk from A to B
        m = m + 1
        B[1].insert(0, A[1].pop(0))
        write(k, m, n, A, B, C)
        # move n-1 disks from C to B, A is auxiliary
        (C, B, A, m) = Hanoi(n-1, C, B, A, k+1, m)
    
    return (A, B, C, m)
def Hanoi(n, A, B, C, k, m):
    """
    if n == 1:
        # move disk from A to B
        m = m + 1
        B[1].insert(0, A[1].pop(0))
        write(k, m, n, A, B, C)
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A, C, B, m) = Hanoi(n-1, A, C, B, k+1, m)
        # move n-th disk from A to B
        m = m + 1
        B[1].insert(0, A[1].pop(0))
        write(k, m, n, A, B, C)
        # move n-1 disks from C to B, A is auxiliary
        (C, B, A, m) = Hanoi(n-1, C, B, A, k+1, m)
    return (A, B, C, m)
Extended Function Hanoi

def Hanoi(n, A, B, C, k, m):
    "..."
    if n == 1:
        # move disk from A to B
        m = m + 1
        B[1].insert(0, A[1].pop(0))
        write(k, m, n, A, B, C)
    else:
        # move n-1 disks from A to C, B is auxiliary
        (A, C, B, m) = Hanoi(n-1, A, C, B, k+1, m)
        # move n-th disk from A to B
        m = m + 1
        B[1].insert(0, A[1].pop(0))
        write(k, m, n, A, B, C)
        # move n-1 disks from C to B, A is auxiliary
        (C, B, A, m) = Hanoi(n-1, C, B, A, k+1, m)
    return (A, B, C, m)
The Towers of Hanoi recursive problem solving
a recursive Python function tracing: exponential time

The Fibonacci Numbers a simple recursion an iterative algorithm

Memoization exponential complexity and cost an efficient recursive Fibonacci

Writing the States

Pile $A$ is a tuple $(A[0], A[1])$: $A[0]$ is name, $A[1]$ is list.

```python
def write_piles(s, A, B, C):
    "writes contents of piles, after s"
    sA = '%s = %s' % (A[0], A[1])
    sB = '%s = %s' % (B[0], B[1])
    sC = '%s = %s' % (C[0], C[1])
    print s, sA, sB, sC

def write(k, m, n, A, B, C):
    "writes contents of piles"
    s = k*''
    s = s + 'move %d, n = %d :' % (m, n)
    write_piles(s, A, B, C)
```

```ruby
def write_piles(s, A, B, C):
    "writes contents of piles, after s"
    sA = '%s = %s' % (A[0], A[1])
    sB = '%s = %s' % (B[0], B[1])
    sC = '%s = %s' % (C[0], C[1])
    print s, sA, sB, sC

def write(k, m, n, A, B, C):
    "writes contents of piles"
    s = k*''
    s = s + 'move %d, n = %d :' % (m, n)
    write_piles(s, A, B, C)
```
Writing the States

Pile A is a tuple (A[0], A[1]): A[0] is name, A[1] is list.

def write_piles(s, A, B, C):
    "writes contents of piles, after s"
    sA = '%s = %s' % (A[0], A[1])
    sB = '%s = %s' % (B[0], B[1])
    sC = '%s = %s' % (C[0], C[1])
    print s, sA, sB, sC

def write(k, m, n, A, B, C):
    "writes contents of piles"
    s = k*' '
    s = s + 'move %d, n = %d :' % (m, n)
    write_piles(s, A, B, C)
Writing the States

Pile A is a tuple \((A[0], A[1])\):
A[0] is name, A[1] is list.

```python
def write_piles(s, A, B, C):
    "writes contents of piles, after s"
    sA = '%s = %s' % (A[0], A[1])
    sB = '%s = %s' % (B[0], B[1])
    sC = '%s = %s' % (C[0], C[1])
    print s, sA, sB, sC

def write(k, m, n, A, B, C):
    "writes contents of piles"
    s = k*'
    s = s + 'move %d, n = %d :' % (m, n)
    write_piles(s, A, B, C)
```

Memoization

exponential complexity and cost
an efficient recursive Fibonacci
The Towers of Hanoi

recursive problem solving
a recursive Python function tracing: exponential time

The Fibonacci Numbers

a simple recursion an iterative algorithm

Memoization

exponential complexity and cost an efficient recursive Fibonacci

Exponential Execution Time

Observe: to move $n$ disks, we need

$n = 1 \rightarrow 1$ move \quad $n = 2 \rightarrow 3$ moves

$n = 3 \rightarrow 7$ moves \quad $n = 4 \rightarrow 15$ moves \ldots

Let $T(n)$ count number of moves for $n$ disks:

\[
T(1) = 1 \quad T(n) = 2T(n - 1) + 1.
\]

Solving the recurrence relation:

\[
\begin{align*}
T(n) & = 2T(n - 1) + 1 \\
& = 2(2T(n - 2) + 1) + 1 \\
& = 2^k T(n - k) + 2^{k-1} + \cdots + 2 + 1 \\
& = 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \\
& = 2^n - 1
\end{align*}
\]
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm
Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Exponential Execution Time

Observe: to move \( n \) disks, we need
\[ n = 1 \rightarrow 1 \text{ move} \quad n = 2 \rightarrow 3 \text{ moves} \]
\[ n = 3 \rightarrow 7 \text{ moves} \quad n = 4 \rightarrow 15 \text{ moves} \ldots \]

Let \( T(n) \) count number of moves for \( n \) disks:
\[
T(1) = 1 \quad T(n) = 2T(n - 1) + 1.
\]

Solving the recurrence relation:
\[
T(n) = 2T(n - 1) + 1 \\
= 2(2T(n - 2) + 1) + 1 \\
= 2^k T(n - k) + 2^{k-1} + \cdots + 2 + 1 \\
= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \\
= 2^n - 1
\]
Exponential Execution Time

Observe: to move $n$ disks, we need

$n = 1 \rightarrow 1$ move  
$n = 2 \rightarrow 3$ moves
$n = 3 \rightarrow 7$ moves  
$n = 4 \rightarrow 15$ moves . . .

Let $T(n)$ count number of moves for $n$ disks:

$$T(1) = 1 \quad T(n) = 2T(n-1) + 1.$$ 

Solving the recurrence relation:

$$T(n) = 2T(n-1) + 1$$
$$= 2(2T(n-2) + 1) + 1$$
$$= 2^k T(n-k) + 2^{k-1} + \cdots + 2 + 1$$
$$= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1$$
$$= 2^n - 1$$
The Towers of Hanoi

recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers

a simple recursion
an iterative algorithm

Memoization

exponential complexity and cost
an efficient recursive Fibonacci

Exponential Execution Time

Observe: to move \( n \) disks, we need
\[
\begin{align*}
n = 1 & \rightarrow 1 \text{ move} \\
n = 2 & \rightarrow 3 \text{ moves} \\
n = 3 & \rightarrow 7 \text{ moves} \\
n = 4 & \rightarrow 15 \text{ moves} \\
& \ldots
\end{align*}
\]

Let \( T(n) \) count number of moves for \( n \) disks:
\[
T(1) = 1 \quad T(n) = 2T(n-1) + 1.
\]

Solving the recurrence relation:
\[
\begin{align*}
T(n) &= 2T(n-1) + 1 \\
&= 2(2T(n-2) + 1) + 1 \\
&= 2^k T(n - k) + 2^{k-1} + \cdots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \\
&= 2^n - 1
\end{align*}
\]
Recursion versus Iteration

1. The Towers of Hanoi
   recursive problem solving
   a recursive Python function
   tracing: exponential time

2. The Fibonacci Numbers
   a simple recursion
   an iterative algorithm

3. Memoization
   exponential complexity and cost
   an efficient recursive Fibonacci
The Fibonacci Numbers

The $n$-th Fibonacci number $F_n$ is defined as

\[ F_0 = 0, \quad F_1 = 1, \quad n > 1 : \quad F_n = F_{n-1} + F_{n-2}. \]

def Fibonacci(n):
    
    """
    returns the $n$-th Fibonacci number
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
The Fibonacci Numbers

The $n$-th Fibonacci number $F_n$ is defined as

$$F_0 = 0, F_1 = 1, \quad n > 1 : F_n = F_{n-1} + F_{n-2}.$$ 

```python
def Fibonacci(n):
    """
    returns the n-th Fibonacci number
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
```
The Fibonacci Numbers

The $n$-th Fibonacci number $F_n$ is defined as

$$F_0 = 0, F_1 = 1, \quad n > 1 : F_n = F_{n-1} + F_{n-2}.$$ 

```python
def Fibonacci(n):
    """
    returns the n-th Fibonacci number
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
```
Computing $\text{Fibonacci}(5)$

```python
$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
   F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
         F(1) = 1
         F(0) = 0
         F(1) = 1
F(5) = 5
number of calls : 25
```
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm
Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Computing $\text{Fibonacci}(5)$

$\text{python fibonacci.py}$
Give n : 5
$F(5) = F(4) + F(3)$
$F(4) = F(3) + F(2)$

$F(3) = F(2) + F(1)$
$F(2) = F(1) + F(0)$
$F(1) = 1$
$F(0) = 0$
$F(1) = 1$
$F(2) = F(1) + F(0)$
$F(1) = 1$
$F(0) = 0$
$F(3) = F(2) + F(1)$
$F(2) = F(1) + F(0)$
$F(1) = 1$
$F(0) = 0$
$F(1) = 1$
$F(5) = 5$

number of calls : 25
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Computing Fibonacci(5)

$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
   F(4) = F(3) + F(2)
   F(3) = F(2) + F(1)
       F(2) = F(1) + F(0)
           F(1) = 1
           F(0) = 0
           F(1) = 1
       F(2) = F(1) + F(0)
           F(1) = 1
           F(0) = 0
           F(1) = 1
F(3) = F(2) + F(1)
   F(2) = F(1) + F(0)
   F(1) = 1
   F(0) = 0
   F(1) = 1
F(5) = 5
number of calls : 25
Computing $Fibonacci(5)$

$\texttt{python fibonacci.py}$

Give n : 5

$F(5) = F(4) + F(3)$

$F(4) = F(3) + F(2)$

$F(3) = F(2) + F(1)$

$F(2) = F(1) + F(0)$

$F(1) = 1$

$F(0) = 0$

$F(1) = 1$

$F(2) = F(1) + F(0)$

$F(1) = 1$

$F(0) = 0$

$F(1) = 1$

$F(3) = F(2) + F(1)$

$F(2) = F(1) + F(0)$

$F(1) = 1$

$F(0) = 0$

$F(1) = 1$

$F(5) = 5$

number of calls : 25
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm
Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Computing Fibonacci(5)

$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
            F(1) = 1
              F(0) = 0
   F(1) = 1
   F(0) = 0
F(2) = F(1) + F(0)
   F(1) = 1
   F(0) = 0
F(3) = F(2) + F(1)
   F(2) = F(1) + F(0)
      F(1) = 1
         F(0) = 0
   F(1) = 1
F(5) = 5
number of calls : 25
Computing $\text{Fibonacci}(5)$

$\text{\$ python fibonacci.py}$

Give n : 5

$F(5) = F(4) + F(3)$

$F(4) = F(3) + F(2)$

$F(3) = F(2) + F(1)$

$F(2) = F(1) + F(0)$

$F(1) = 1$

$F(0) = 0$

$F(1) = 1$

$F(2) = F(1) + F(0)$

$F(1) = 1$

$F(0) = 0$

$F(3) = F(2) + F(1)$

$F(2) = F(1) + F(0)$

$F(1) = 1$

$F(0) = 0$

$F(1) = 1$

$F(5) = 5$

number of calls : 25
Computing $Fibonacci(5)$

```
$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
  F(4) = F(3) + F(2)
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
      F(1) = 1
    F(2) = F(1) + F(0)
      F(1) = 1
      F(0) = 0
    F(1) = 1
  F(3) = F(2) + F(1)
    F(2) = F(1) + F(0)
      F(1) = 1
      F(0) = 0
    F(1) = 1
F(5) = 5
number of calls : 25
```
Computing $Fibonacci(5)$

```python
$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
         F(1) = 1
         F(0) = 0
   F(1) = 1
F(2) = F(1) + F(0)
   F(1) = 1
   F(0) = 0
F(3) = F(2) + F(1)
   F(2) = F(1) + F(0)
   F(1) = 1
   F(0) = 0
   F(1) = 1
F(5) = 5
number of calls : 25
```
$ \text{python fibonacci.py}$

Give n : 5

F(5) = F(4) + F(3)
    F(4) = F(3) + F(2)
        F(3) = F(2) + F(1)
            F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
        F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
    F(3) = F(2) + F(1)
        F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
    F(4) = F(3) + F(2)
        F(3) = F(2) + F(1)
            F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
        F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
    F(5) = F(4) + F(3)
        F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
                F(2) = F(1) + F(0)
                    F(1) = 1
                    F(0) = 0
                    F(1) = 1
            F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
        F(3) = F(2) + F(1)
            F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
    F(4) = F(3) + F(2)
        F(3) = F(2) + F(1)
            F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
        F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
    F(5) = 5

d number of calls : 25
Computing Fibonacci(5)

$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
         F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
          F(3) = F(2) + F(1)
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
           F(4) = F(3) + F(2)
              F(3) = F(2) + F(1)
                 F(2) = F(1) + F(0)
                    F(1) = 1
                    F(0) = 0
                    F(1) = 1
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
         F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
          F(3) = F(2) + F(1)
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
         F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
          F(3) = F(2) + F(1)
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
         F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
          F(3) = F(2) + F(1)
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
         F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
          F(3) = F(2) + F(1)
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
         F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
          F(3) = F(2) + F(1)
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
         F(4) = F(3) + F(2)
            F(3) = F(2) + F(1)
               F(2) = F(1) + F(0)
                  F(1) = 1
                  F(0) = 0
                  F(1) = 1
             F(2) = F(1) + F(0)
                F(1) = 1
                F(0) = 0
                F(1) = 1
         F(5) = 5
number of calls : 25
Computing $Fibonacci(5)$

```java
$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
  F(4) = F(3) + F(2)
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
        F(1) = 1
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
      F(1) = 1
    F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
        F(2) = F(1) + F(0)
          F(1) = 1
          F(0) = 0
          F(1) = 1
        F(2) = F(1) + F(0)
      F(3) = F(2) + F(1)
        F(2) = F(1) + F(0)
          F(1) = 1
          F(0) = 0
          F(1) = 1
    F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
        F(2) = F(1) + F(0)
          F(1) = 1
          F(0) = 0
          F(1) = 1
    F(5) = 5
number of calls : 25
```
Computing Fibonacci(5)

$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
         F(2) = F(1) + F(0)
            F(1) = 1
            F(0) = 0
            F(1) = 1
   F(2) = F(1) + F(0)
      F(1) = 1
      F(0) = 0
      F(1) = 1
F(3) = F(2) + F(1)
   F(2) = F(1) + F(0)
      F(1) = 1
      F(0) = 0
      F(1) = 1
F(5) = 5
number of calls : 25
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm
Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Computing Fibonacci(5)

$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
   F(4) = F(3) + F(2)
      F(3) = F(2) + F(1)
          F(2) = F(1) + F(0)
              F(1) = 1
              F(0) = 0
          F(1) = 1
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
F(3) = F(2) + F(1)
    F(2) = F(1) + F(0)
      F(1) = 1
      F(0) = 0
   F(1) = 1
F(0) = 0
F(1) = 1
F(5) = 5
number of calls : 25
$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
  F(4) = F(3) + F(2)
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
        F(1) = 1
    F(2) = F(1) + F(0)
      F(1) = 1
      F(0) = 0
    F(1) = 1
 F(3) = F(2) + F(1)
  F(2) = F(1) + F(0)
    F(1) = 1
    F(0) = 0
    F(1) = 1
 F(5) = 5
number of calls : 25
Computing \texttt{Fibonacci}(5)

\begin{verbatim}
$ python fibonacci.py
Give n : 5
F(5) = F(4) + F(3)
  F(4) = F(3) + F(2)
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
        F(1) = 1
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
        F(1) = 1
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
        F(1) = 1
  F(4) = F(3) + F(2)
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
        F(1) = 1
  F(4) = F(3) + F(2)
    F(3) = F(2) + F(1)
      F(2) = F(1) + F(0)
        F(1) = 1
        F(0) = 0
        F(1) = 1
F(5) = 5
number of calls : 25
\end{verbatim}
Computing \textit{Fibonacci(5)}

$\texttt{python fibonacci.py}$

Give n : 5

\[
F(5) = F(4) + F(3) \\
F(4) = F(3) + F(2) \\
F(3) = F(2) + F(1) \\
F(2) = F(1) + F(0) \\
F(1) = 1 \\
F(0) = 0 \\
\]

F(1) = 1

\[
F(2) = F(1) + F(0) \\
F(1) = 1 \\
F(0) = 0 \\
\]

F(3) = F(2) + F(1)

\[
F(2) = F(1) + F(0) \\
F(1) = 1 \\
F(0) = 0 \\
\]

F(5) = 5

number of calls : 25
Tracing the Execution

def Fibotrace(n, k, c):
    """
    returns (f,c) f is the n-th Fibonacci number
    and c counts the number of function calls
    prints execution trace using control parameter k
    """
    s = k*' '+ 'F(%d) = '% n
    if n == 0:
        print s + '0'
        return (0,c)
    elif n == 1:
        print s + '1'
        return (1,c)
    else:
        print s + 'F(%d) + F(%d)' % (n-1,n-2)
        (f1,c1) = Fibotrace(n-1, k+1, c+1)
        (f2,c2) = Fibotrace(n-2, k+1, c+1)
        return (f1+f2,c1+c2)
def Fibotrace(n,k,c):
    """
    returns (f,c) f is the n-th Fibonacci number and c counts the number of function calls
    prints execution trace using control parameter k
    """
    s = k*' ' + 'F(%d) = ' % n
    if n == 0:
        print s + '0'
        return (0,c)
    elif n == 1:
        print s + '1'
        return (1,c)
    else:
        print s + 'F(%d) + F(%d)' % (n-1,n-2)
        (f1,c1) = Fibotrace(n-1,k+1,c+1)
        (f2,c2) = Fibotrace(n-2,k+1,c+1)
        return (f1+f2,c1+c2)
Tracing the Execution

def Fibotrace(n, k, c):
    ""
    returns (f, c) f is the n-th Fibonacci number
    and c counts the number of function calls
    prints execution trace using control parameter k
    ""
    s = k*' ' + 'F(%d) = ' % n
    if n == 0:
        print s + '0'
        return (0, c)
    elif n == 1:
        print s + '1'
        return (1, c)
    else:
        print s + 'F(%d) + F(%d)' % (n-1, n-2)
        (f1, c1) = Fibotrace(n-1, k+1, c+1)
        (f2, c2) = Fibotrace(n-2, k+1, c+1)
        return (f1+f2, c1+c2)
def Fibotrace(n,k,c):
    """
    returns (f,c) f is the n-th Fibonacci number
    and c counts the number of function calls
    prints execution trace using control parameter k
    """
    s = k*' ' + 'F(%d) = ' % n
    if n == 0:
        print s + '0'
        return (0,c)
    elif n == 1:
        print s + '1'
        return (1,c)
    else:
        print s + 'F(%d) + F(%d)\n' % (n-1,n-2)
        (f1,c1) = Fibotrace(n-1,k+1,c+1)
        (f2,c2) = Fibotrace(n-2,k+1,c+1)
        return (f1+f2,c1+c2)
Recursion versus Iteration

1. The Towers of Hanoi
   recursive problem solving
   a recursive Python function
   tracing: exponential time

2. The Fibonacci Numbers
   a simple recursion
   an iterative algorithm

3. Memoization
   exponential complexity and cost
   an efficient recursive Fibonacci
Fibonacci with an iterative Algorithm

def F(n):
    """
    iterative way for n-th Fibonacci number
    """
    if n == 0:
        return 0
    else:
        a = 0
        b = 1
        for k in range(2, n+1):
            c = a + b
            a = b
            b = c
        return b
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Memoization
exponential complexity and cost
an efficient recursive Fibonacci

---

**Fibonacci with an iterative Algorithm**

def F(n):
    
    
    """
    iterative way for n-th Fibonacci number
    """

    if n == 0:
        return 0
    else:
        a = 0
        b = 1
        for k in range(2, n+1):
            c = a + b
            a = b
            b = c
        return b
Recursion versus Iteration

1. The Towers of Hanoi
   recursive problem solving
   a recursive Python function
   tracing: exponential time

2. The Fibonacci Numbers
   a simple recursion
   an iterative algorithm

3. Memoization
   exponential complexity and cost
   an efficient recursive Fibonacci
The towers of Hanoi problem is hard *no matter what* algorithm is used. Its *complexity* is exponential.

The first recursive computation of the Fibonacci numbers took long, its *cost* is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation.

Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of *Computer Science: an overview*. 
The towers of Hanoi problem is hard *no matter what* algorithm is used. Its *complexity* is exponential.

The first recursive computation of the Fibonacci numbers took long, its *cost* is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation.

Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of *Computer Science: an overview.*
Exponential Complexity and Cost

The towers of Hanoi problem is hard *no matter what* algorithm is used. Its *complexity* is exponential. The first recursive computation of the Fibonacci numbers took long, its *cost* is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation. Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of *Computer Science: an overview.*
Exponential Complexity and Cost

The towers of Hanoi problem is hard *no matter what* algorithm is used. Its *complexity* is exponential. The first recursive computation of the Fibonacci numbers took long, its *cost* is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation. Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of *Computer Science: an overview.*
The towers of Hanoi problem is hard *no matter what* algorithm is used. Its *complexity* is exponential.

The first recursive computation of the Fibonacci numbers took long, its *cost* is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation.

Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of *Computer Science: an overview.*
Exponential Complexity and Cost

The towers of Hanoi problem is hard *no matter what* algorithm is used. Its *complexity* is exponential. The first recursive computation of the Fibonacci numbers took long, its *cost* is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation.

Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of *Computer Science: an overview.*
Exponential Complexity and Cost

The towers of Hanoi problem is hard no matter what algorithm is used. Its complexity is exponential. The first recursive computation of the Fibonacci numbers took long, its cost is exponential. If the number of function calls exceeds the size of the results, we better use an iterative formulation. Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of Computer Science: an overview.
The towers of Hanoi problem is hard *no matter what* algorithm is used. Its *complexity* is exponential.

The first recursive computation of the Fibonacci numbers took long, its *cost* is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation.

Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of *Computer Science: an overview.*
Exponential Complexity and Cost

The towers of Hanoi problem is hard \textit{no matter what} algorithm is used. Its \textit{complexity} is exponential.

The first recursive computation of the Fibonacci numbers took long, its \textit{cost} is exponential.

If the number of function calls exceeds the size of the results, we better use an iterative formulation.

Using a stack to store the function calls, every recursive program can be transformed into an iterative one.

Background material for this lecture:

- pages 256-257 of \textit{Computer Science: an overview}.
Recursion versus Iteration

1. The Towers of Hanoi
   recursive problem solving
   a recursive Python function
   tracing: exponential time

2. The Fibonacci Numbers
   a simple recursion
   an iterative algorithm

3. Memoization
   exponential complexity and cost
   an efficient recursive Fibonacci
Memoization

Consider again the recursive computation of the Fibonacci numbers.

What if we store the results of previous function calls in a Python dictionary?

A dictionary is a set of key:value pairs.

1. key: type of the parameter,
2. value: type of the result.

Using the dictionary D:

- Every call with \( n \) starts with a lookup: \( D \cdot \text{has\_key}(n) \)
- If \( D \cdot \text{has\_key}(n) \): return \( D[n] \).
- Otherwise, store result \( R \): \( D[n] = R \).
Memoization

Consider again the recursive computation of the Fibonacci numbers.

What if we store the results of previous function calls in a Python dictionary?

A dictionary is a set of key:value pairs.

1. key: type of the parameter,
2. value: type of the result.

Using the dictionary D:

- Every call with n starts with a lookup: D.has_key(n)
- If D.has_key(n): return D[n].
- Otherwise, store result R: D[n] = R.
Memoization

Consider again the recursive computation of the Fibonacci numbers.

What if we store the results of previous function calls in a Python dictionary?

A dictionary is a set of key:value pairs.

1. key: type of the parameter,
2. value: type of the result.

Using the dictionary D:

- Every call with n starts with a lookup: D.has_key(n)
- If D.has_key(n): return D[n].
- Otherwise, store result R: D[n] = R.
The Towers of Hanoi
recursive problem solving
a recursive Python function
tracing: exponential time

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Memoization
exponential complexity and cost
an efficient recursive Fibonacci

Running `memofib.py`

$ python memofib.py
Give a number : 20
The 20-th Fibonacci number is 6765
The dictionary of function calls :
{0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8,
  7: 13, 8: 21, 9: 34, 10: 55, 11: 89, 12: 144,
  13: 233, 14: 377, 15: 610, 16: 987, 17: 1597,
  18: 2584, 19: 4181, 20: 6765}

To increase the maximum recursion depth:

```python
import sys
sys.setrecursionlimit(n+2)
```
Running `memofib.py`

```
$ python memofib.py
Give a number : 20
The 20-th Fibonacci number is 6765
The dictionary of function calls :
{0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, \n 7: 13, 8: 21, 9: 34, 10: 55, 11: 89, 12: 144, \n 13: 233, 14: 377, 15: 610, 16: 987, 17: 1597, \n 18: 2584, 19: 4181, 20: 6765}
```

To increase the maximum recursion depth:

```python
import sys
sys.setrecursionlimit(n+2)
```
class MemoizedFibonacci():
    ""
    The class exports a data attribute D that stores the results of all calls made to the functional attribute F.
    ""
    def __init__(self):
        ""
        Initializes the dictionary.
        ""
        self.D = {}

    def F(self,n):
        ""
        returns the n-th Fibonacci number
        ""
def F(self,n):
    """
    returns the n-th Fibonacci number
    """
    if self.D.has_key(n):
        return self.D[n]
    else:
        if n == 0:
            R = 0
        elif n == 1:
            R = 1
        else:
            R = self.F(n-1) + self.F(n-2)
        self.D[n] = R
    return R
from sys import setrecursionlimit

def main():
    ""
    Prompts the user for a number n, instantiates an object m of the class MemoizedFibonacci and applies the method F to the object m and shows m.D.
    ""

    m = MemoizedFibonacci()
    n = input('Give a number : ')
    setrecursionlimit(n+2)
    f = m.F(n)
    print 'The %d-th Fibonacci number is %d' % (n,f)
    print 'The dictionary of function calls :'
    print m.D

if __name__=='__main__': main()
Exercises

1. We define the Harmonic numbers $H_n$ as $H_1 = 1$ and $H_n = H_{n-1} + 1/n$. Write a recursive function for $H_n$.

2. Extend the recursive function for $H_n$ (see above) with a parameter to keep track of the number of function calls. Write an iterative function for $H_n$.

3. The number of derangements of $n$ elements is defined as $d_0 = 1$, $d_1 = 0$, and for $n > 1$: $d_n = (n - 1)(d_{n-1} + d_{n-2})$. Define a recursive Python function to compute $d_n$. Use memoization to make the function efficient.

4. Write an iterative version for the function
   
   ```python
   def is_palindrome():
       # function body
   ```
   of Lecture 8.

5. Describe the design for a GUI to show the moves to solve the problem of the towers of Hanoi.