Recursion versus Iteration

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

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Exponential Complexity and Cost
recursion versus iteration
Recursion versus Iteration

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

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Exponential Complexity and Cost
recursion versus iteration
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.

A | B | C

→

A | B | C

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 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 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.

```
  A  B  C
```

2. Move $n$-th disk from A to B.

```
  A  B  C
```

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

```
  A  B  C
```
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:

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:

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
Recursion versus Iteration

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

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Exponential Complexity and Cost
recursion versus iteration
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 (\textit{pop}), and add only at the top (\textit{push}).

A pile of 4 disks of decreasing size:

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

To remove the top element:

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

To put an element on top:

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

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

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)
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 Towers of Hanoi

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Exponential Complexity and Cost
recursion versus iteration

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)
```
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)
Recursion versus Iteration

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

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Exponential Complexity and Cost
recursion versus iteration
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 = []
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 = []
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

Exponential Complexity and Cost recursion versus iteration

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

Exponential Complexity and Cost
recursion versus iteration

Tracing the Execution

$ python hanoi.py
Give number of disks: 4
at start : A = [1, 2, 3, 4] B = [] C = []
moving 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 = []
$ \text{python hanoi.py} \\
\text{Give number of disks : 4} \\
\text{at start : } A = [1, 2, 3, 4] \quad B = [] \quad C = [] \\
\text{move 1, n = 1 : } A = [2, 3, 4] \quad C = [1] \quad B = [] \\
\text{move 3, n = 1 : } C = [] \quad B = [1, 2] \quad A = [3, 4] \\
\text{move 6, n = 2 : } B = [] \quad C = [2, 3] \quad A = [1, 4] \\
\text{move 7, n = 1 : } A = [4] \quad C = [1, 2, 3] \quad B = [] \\
\text{move 8, n = 4 : } A = [] \quad B = [4] \quad C = [1, 2, 3] \\
\text{move 9, n = 1 : } C = [2, 3] \quad B = [1, 4] \quad A = [] \\
\text{move 12, n = 3 : } C = [] \quad B = [3, 4] \quad A = [1, 2] \\
\text{move 14, n = 2 : } A = [] \quad B = [2, 3, 4] \quad C = [1] \\
\text{move 15, n = 1 : } C = [] \quad B = [1, 2, 3, 4] \quad A = []$
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)
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)
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)
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)
```

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)
```

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

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Exponential Complexity and Cost
recursion versus iteration
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 ...

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 \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:

\[
\begin{align*}
T(n) & = 2T(n - 1) + 1 \\
& = 2(2T(n - 2) + 1) + 1 \\
& = 2^k T(n - k) + 2^{k-1} + \cdots + 2 + 1 \\
& = 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \\
& = 2^n - 1
\end{align*}
\]
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 . . .

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
\[ n = 1 \rightarrow 1 \text{ move} \quad n = 2 \rightarrow 3 \text{ moves} \]
\[ n = 3 \rightarrow 7 \text{ moves} \quad n = 4 \rightarrow 15 \text{ moves} \ldots \]

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

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

Solving the recurrence relation:

\[
T(n) &= 2T(n - 1) + 1 \\
&= 2(2T(n - 2) + 1) + 1 \\
&= 2^k T(n - k) + 2^{k-1} + \cdots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \\
&= 2^n - 1
\]
Recursion versus Iteration

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

The Fibonacci Numbers
- a simple recursion
- an iterative algorithm

Exponential Complexity and Cost
- recursion versus iteration
The Fibonacci Numbers

The \( n \)-th Fibonacci number \( F_n \) is defined as

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

def Fibonacci(n):
    """
    returns \( n \)-th \( n \)-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

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

```python
def Fibonacci(n):
    """
    returns \( n \)-th \( n \)-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

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

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

$\texttt{python fibonacci.py}$

\texttt{Give n : 5}

\texttt{F(5) = F(4) + F(3)}

\texttt{F(4) = F(3) + F(2)}

\texttt{F(3) = F(2) + F(1)}

\texttt{F(2) = F(1) + F(0)}

\texttt{F(1) = 1}

\texttt{F(0) = 0}

\texttt{F(1) = 1}

\texttt{F(2) = F(1) + F(0)}

\texttt{F(1) = 1}

\texttt{F(0) = 0}

\texttt{F(3) = F(2) + F(1)}

\texttt{F(2) = F(1) + F(0)}

\texttt{F(1) = 1}

\texttt{F(0) = 0}

\texttt{F(1) = 1}

\texttt{F(5) = 5}

\texttt{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 \texttt{Fibonacci(5)}

\$ python fibonacci.py
Give \texttt{n} : 5
\begin{align*}
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(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 \\
\text{number of calls : 25}
\end{align*}
Computing **Fibonacci(5)**

```bash
$ 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(3) = 5

number of calls : 25
```
Computing \textbf{Fibonacci}(5)

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

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

\$\text{python fibonacci.py}$

Give n : 5

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

number of calls : 25
Computing \textbf{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(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 \textbf{Fibonacci}(5)

$\text{python fibonacci.py}$

Give \textit{n} : 5

\begin{align*}
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 \\
\text{number of calls} : 25
\end{align*}
Computing $\textbf{Fibonacci}(5)$

```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(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(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
```
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)
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)
The Towers of Hanoi recursive problem solving

A recursive Python function

Tracing: exponential time

The Fibonacci Numbers a simple recursion

An iterative algorithm

Exponential Complexity and Cost

Recursion versus iteration

Tracing the Execution

```python
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)
Recursion versus Iteration

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

The Fibonacci Numbers
a simple recursion
an iterative algorithm

Exponential Complexity and Cost
recursion versus iteration
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
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

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

The Fibonacci Numbers
  a simple recursion
  an iterative algorithm

Exponential Complexity and Cost
  recursion versus iteration
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 longs, 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:

- Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

▶ Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

- Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

- Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

- Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

- Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

- Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

- Chapter 8 of *The Art & Craft of Computing*, in particular: read §8.3 for execution details.
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 longs, 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:

- Chapter 8 of The Art & Craft of Computing, in particular: read §8.3 for execution details.
Exercises

1. Write a Python function $F(n)$ which returns a list of the first $n$ Fibonacci numbers.

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

3. 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 $H_n$.

4. Write an iterative version for the function 
   `is_palindrome()` of Lecture 6.

5. Design a GUI to show the moves to solve the problem of the towers of Hanoi.