Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - exponential complexity
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, 1 February 2017

Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - exponential complexity

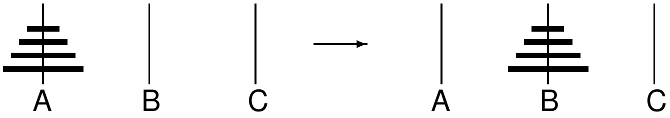
2. The Fibonacci Numbers
   - a simple recursion
   - an iterative algorithm

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.



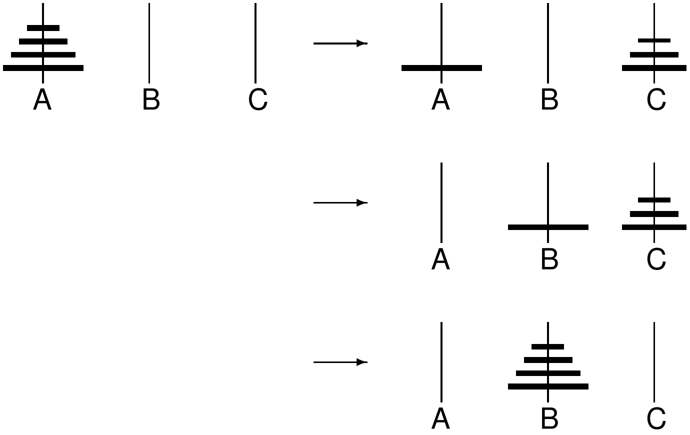
Task: move the disks from the first pile to the second, obeying the following rules:

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.

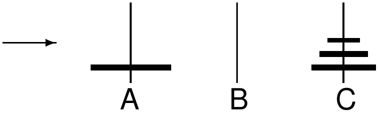
a recursive solution

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

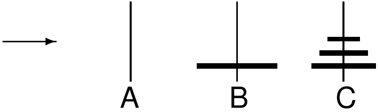


a recursive algorithm

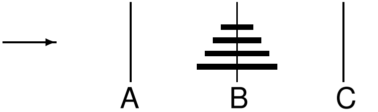
Base case: move one disk from A to B.
To move n disks from A to B:



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
   - exponential complexity

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 (*pop*), and add only at the top (*push*).

A pile of 4 disks of decreasing size:

```
>>> A = [x for x in 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]
```

a recursive Python function

```
def hanoi(nbr, apl, bpl, cpl):
    """
    Moves nbr disks from apl to bpl, cpl is auxiliary.
    """
    if nbr == 1:
        # move disk from apl to bpl
        bpl.insert(0, apl.pop(0))
    else:
        # move nbr-1 disks from apl to cpl, using bpl
        hanoi(nbr-1, apl, cpl, bpl)
        # move nbr-th disk from apl to bpl
        bpl.insert(0, apl.pop(0))
        # move nbr-1 disks from cpl to bpl, using apl
        hanoi(nbr-1, cpl, bpl, apl)
```

the main function

```
def main():
    """
    Prompts for the number of disks,
    initializes the piles, and
    then calls the hanoi function.
    """
    nbr = int(input('Give the number of disks :'))
    apl = [x for x in range(nbr)]
    bpl = []
    cpl = []
    print('A =', apl, 'B =', bpl, 'C =', cpl)
    hanoi(nbr, apl, bpl, cpl)
    print('A =', apl, 'B =', bpl, 'C =', cpl)
```

Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - exponential complexity

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

3. 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 1, n = 1 : A = [2, 3, 4] C = [1] B = []
  move 2, n = 2 : A = [3, 4] B = [2] C = [1]
   move 3, n = 1 : C = [] B = [1, 2] A = [3, 4]
 move 4, n = 3 : A = [4] C = [3] B = [1, 2]
   move 5, n = 1 : B = [2] A = [1, 4] C = [3]
  move 6, n = 2 : B = [] C = [2, 3] A = [1, 4]
   move 7, n = 1 : A = [4] C = [1, 2, 3] B = []
move 8, n = 4 : A = [] B = [4] C = [1, 2, 3]
   move 9, n = 1 : C = [2, 3] B = [1, 4] A = []
  move 10, n = 2 : C = [3] A = [2] B = [1, 4]
   move 11, n = 1 : B = [4] A = [1, 2] C = [3]
 move 12, n = 3 : C = [] B = [3, 4] A = [1, 2]
   move 13, n = 1 : A = [2] C = [1] B = [3, 4]
  move 14, n = 2 : A = [] B = [2, 3, 4] C = [1]
   move 15, n = 1 : C = [] B = [1, 2, 3, 4] A = []
```

piles are represented as tuples

The roles of the piles shift. Each pile is a 2-tuple:

1. the name of the pile comes first,
2. the second iterm is a list of numbers.

```
def write_piles(pre, apl, bpl, cpl):
    """
    Writes the contents of the piles apl, bpl,
    and cpl, after writing the string pre.
    Every pile is a 2-tuple: the name
    of the pile and a list of numbers.
    """
    sapl = '%s = %s' % apl
    sbpl = '%s = %s' % bpl
    scpl = '%s = %s' % cpl
    print(pre, sapl, sbpl, scpl)
```

writing with recursion depth

When we write the state of each pile, at each move,
we like to see the recursion depth.

The recursion depth equals the number of spaces.

```
def write(k, move, nbr, apl, bpl, cpl):
    """
    Writes the contents of piles apl, bpl, cpl,
    preceded by k spaces, writing the move
    and the number of disks nbr.
    """
    pre = k*' '
    pre += 'move %d, n = %d :' % (move, nbr)
    write_piles(pre, apl, bpl, cpl)
```

extra arguments

In addition to the number of disks and the tuple representations of the piles, we need:

1. a counter for the depth of the recursion, and
2. a counter for the number of moves.

```
def hanoi(nbr, apl, bpl, cpl, k, move):
    """
    Moves nbr disks from apl to bpl, cpl is auxiliary.
    The recursion depth is counted by k,
    move counts the number of moves.
    Writes the state of the piles after each move.
    Returns the number of moves.
    """
```

the new main function

```
def main():
    """
    Prompts user for the number of disks,
    initializes the stacks and calls hanoi.
    """
    nbr = int(input('Give number of disks : '))
    main4hanoi0(nbr)
    apl = ('A', list(range(1, nbr+1)))
    bpl = ('B', [])
    cpl = ('C', [])
    write_piles('at start :', apl, bpl, cpl)
    cnt = hanoi(nbr, apl, bpl, cpl, 0, 0)
    print('number of moves :', cnt)
    write_piles('  at end :', apl, bpl, cpl)
```

the extended function hanoi

```
def hanoi(nbr, apl, bpl, cpl, k, move):
    if nbr == 1:
        # move disk from A to B
        bpl[1].insert(0, apl[1].pop(0))
        write(k, move+1, nbr, apl, bpl, cpl)
        return move+1
    else:
        # move nbr-1 disks from A to C, B is auxiliary
        move = hanoi(nbr-1, apl, cpl, bpl, k+1, move)
        # move nbr-th disk from A to B
        bpl[1].insert(0, apl[1].pop(0))
        write(k, move+1, nbr, apl, bpl, cpl)
        # move nbr-1 disks from C to B, A is auxiliary
        move = hanoi(nbr-1, cpl, bpl, apl, k+1, move+1)
        return move
```

exponential complexity

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:

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

Solving the recurrence relation:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2^k T(n-k) + 2^{k-1} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - exponential complexity

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

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

```
def fibonacci(nbr):
    """
    Returns the nbr-th Fibonacci number.
    """
    if nbr == 0:
        return 0
    elif nbr == 1:
        return 1
    else:
        return fibonacci(nbr-1) + fibonacci(nbr-2)
```

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
```

tracing the execution

```
def fibotrace(nbr, k, cnt):
    """
    Returns (f, cnt) where f is the nbr-th Fibonacci
    number and cnt the number of function calls.
    Prints execution trace using the control parameter k.
    """
    outs = k*' ' + 'F(%d) = ' % nbr
    if nbr == 0:
        print(outs + '0')
        return (0, cnt)
    elif nbr == 1:
        print(outs + '1')
        return (1, cnt)
    else:
        print(outs + 'F(%d) + F(%d)' % (nbr-1, nbr-2))
        (fb1, cnt1) = fibotrace(nbr-1, k+1, cnt+1)
        (fb2, cnt2) = fibotrace(nbr-2, k+1, cnt+1)
        return (fb1+fb2, cnt1+cnt2)
```

Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - exponential complexity

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

In each step we add two numbers: the previous one
to the previous to the previous one.

```
def iterfib(nbr):
    """
    Iterative algorithm for nbr-th Fibonacci number.
    """
    if nbr == 0:
        return 0
    else:
        (first, second) = (0, 1)
        for _ in range(2, nbr+1):
            (first, second) = (second, first + second)
        return second
```

Observe how code gets shorter with tuple assignment.

Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - exponential complexity
2. The Fibonacci Numbers
   - a simple recursion
   - an iterative algorithm
3. Memoization
   - exponential complexity and cost
   - an efficient recursive Fibonacci

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*.

Recursion versus Iteration

1. The Towers of Hanoi
   - recursive problem solving
   - a recursive Python function
   - exponential complexity

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: `n in D`
- If `n in D`: `return D[n]`.
- Otherwise, store result `R`: `D[n] = R`.

storing data in function calls

To an argument of a function we assign a dictionary:

```
def storecalls(nbr, calls={}):
    """
    Stores the value of nbr in the dictionary calls.
    Each time we print the address of calls
    and all values stored in calls.
    The length of the dictionary records the number
    of times the function is called.
    """
    print('calls =', calls, 'with id =', id(calls))
    calls[len(calls)] = nbr
```

The `id()` function returns the identity of an object.
CPython returns the memory address of the object.

running `storecalls`

```
$ python storecalls.py
give a number (0 to exit) : 3
calls = {} with id = 4300915016
give a number (0 to exit) : 4
calls = {0: 3} with id = 4300915016
give a number (0 to exit) : 8
calls = {0: 3, 1: 4} with id = 4300915016
give a number (0 to exit) : 2
calls = {0: 3, 1: 4, 2: 8} with id = 4300915016
...
```

What just happened?

1. At the first call to the function, the arguments of the function are evaluated. With `calls = { }`, an empty dictionary is created and placed somewhere in memory.
2. At each following call, the *same* memory address is used.

dereferencing an address

Importing the function `storecalls` from the module `storecalls`:

```
>>> from storecalls import storecalls
>>> storecalls(3)
calls = {} with id = 4303043528
>>> storecalls(8)
calls = {0: 3} with id = 4303043528
>>> from ctypes import cast, py_object
>>> cast(4303043528, py_object).value
{0: 3, 1: 8}
```

In CPython, `id()` returns the address.
With `cast` we can dereference the address: given the address, we obtain the value of the object stored at that address.

Memoizing Fibonacci

```
def memfib(nbr, calls={}):
    """
    Returns the nbr-th Fibonacci number, using calls to
    memoize the values computed in previous calls.
    """
    if nbr in calls:      # first check the dictionary
        return calls[nbr]
    else:                 # compute the value recursively
        if nbr == 0:
            result = 0    # first base case
        elif nbr == 1:
            result = 1    # second base case
        else:
            result = memfib(nbr-1) + memfib(nbr-2)
        calls[nbr] = result # store in the dictionary
        return result
```

increasing the recursion limit

The recursion limit is the maximum depth of the Python interpreter stack. This limit prevents infinite recursion.

```
>>> from sys import getrecursionlimit
>>> getrecursionlimit()
1000
```

We can set the maximum depth of the Python interpreter stack:

```
>>> from sys import setrecursionlimit
>>> setrecursionlimit(2000)
>>> getrecursionlimit()
2000
```

We need this to compute large Fibonacci numbers.

the function `main()`

```
from sys import setrecursionlimit

def main():
    """
    Prompts the user for a number n,
    and computes the n-th Fibonacci number.
    """
    nbr = int(input('Give a number : '))
    setrecursionlimit(nbr+3)
    fib = memfib(nbr)
    print('The %d-th Fibonacci number is %d' % (nbr, fib))

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 `is_palindrome()` of Lecture 8.
5. Define a class `Tower` to represent a pile, with the operations to pop and push a disk. Use this class in a recursive hanoi function which prints all the moves.