from Recursion to Iteration

1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays
2. converting recursion into iteration
   - an iterative version with a stack of parameters
3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

MCS 275 Lecture 18
Programming Tools and File Management
Jan Verschelde, 20 February 2017

from Recursion to Iteration

1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays
2. converting recursion into iteration
   - an iterative version with a stack of parameters
3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

Quicksort Revisited

using arrays

Recall the idea of Quicksort:

1. choose x and partition list in two:
   left list: $\leq x$ and right list: $\geq x$
2. sort the lists left and right

Our first implementation of Lecture 16
is *recursively functional*.

$\rightarrow$ Python's builtin lists handle all data

`pro`: convenient for programming
`con`: multiple copies of same data

Goals: 1. use arrays for data efficiency,
2. turn recursion into iteration.

arrays of random integers

```
from array import array as Array

def main():
    """
    Generates a random array of integers
    and applies quicksort.
    """
    low = int(input('Give lower bound : '))
    upp = int(input('Give upper bound : '))
    nbr = int(input('How many numbers ? '))
    ans = input('Extra output ? (y/n) ')
    from random import randint
    nums = [randint(low, upp) for _ in range(nbr)]
    data = Array('i', nums)
    print('A =', data)
    recursive_quicksort(data, 0, nbr, ans == 'y')
    print('A =', data)
```

from Recursion to Iteration

1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays

2. converting recursion into iteration
   - an iterative version with a stack of parameters

3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

partitioning arrays

Take x in the middle of the array.
Apply scan and swap, for $i < j$:
if A[i] $> x$ and A[j] $< x$: A[i], A[j] = A[j], A[i]

For example: `A = [31 93 49 37 56 95 74 59]`
At the middle: $x = 56$ (== `A[4]`)

Start with $i = 0$ and $j = 8$.
Increase i while A[i] $< x$, end at $i = 1$.
Decrease j while A[j] $> x$, end at $j = 4$.
Swap A[1] and A[4], and continue scanning A.

```
A = [31 93 49 37 56 95 74 59]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
```

from Recursion to Iteration

1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays

2. converting recursion into iteration
   - an iterative version with a stack of parameters

3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

the function `partition()`

The specification and documentation:

```
def partition(arr, first, last):
    """
    Partitions arr[first:last] using as pivot
    the middle item x.  On return is (i, j, x):
    i > j, all items in arr[i:last] are >= x,
    all items in arr[first:j+1] are <= x.
    """
```

the body of `partition()`

```
def partition(arr, first, last):

    pivot = arr[(first+last)//2]
    i = first
    j = last-1
    while i <= j:
        while arr[i] < pivot:
            i = i+1
        while arr[j] > pivot:
            j = j-1
        if i < j:
            (arr[i], arr[j]) = (arr[j], arr[i])
        if i <= j:
            (i, j) = (i+1, j-1)
    return (i, j, pivot)
```

checking postconditions for correctness

Important to verify the correctness:

```
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
```

```
def check_partition(arr, first, last, i, j, pivot):
    """
    Prints the result of the partition
    for a visible check on the postconditions.
    """
    print('i = %d, j = %d, x = %d' % (i, j, pivot))
    print('arr[%d:%d] =' % (first, j+1), \
        arr[first:j+1], '<=', pivot)
    print('arr[%d:%d] =' % (i, last), \
        arr[i:last], '>=', pivot)
```

a recursive quicksort

```
def recursive_quicksort(data, first, last, verbose=True):
    """
    Sorts the array data in increasing order.
    If verbose, then extra output is written.
    """
    (i, j, pivot) = partition(data, first, last)
    if verbose:
        check_partition(data, first, last, i, j, pivot)
    if j > first:
        recursive_quicksort(data, first, j+1, verbose)
    if i < last-1:
        recursive_quicksort(data, i, last, verbose)
```

Important: ***first*** sort `data[first:j+1]`.

from Recursion to Iteration

1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays

2. converting recursion into iteration
   - an iterative version with a stack of parameters

3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

Converting Recursion into Iteration

a stack for the parameters of the calls

Recursion is executed via a stack.

For quicksort, we store first and last index
of the array to sort.

With every call we push `(first, last)` on the stack.

As long as the stack of indices is not empty:

1. pop the indices `(first, last)` from the stack
2. we partition the array `A[first:last]`
3. push `(i, last)` and then `(first, j+1)`

running the iterative code

```
A = [31 93 49 37 56 95 74 59]
S = [(0, 8)]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
S = [(0, 4), (4, 8)]
i = 3, j = 1, x = 49
A[0:2] = [31 37] <= 49
A[3:4] = [56] >= 49
S = [(0, 2), (4, 8)]
i = 2, j = 0, x = 37
A[0:1] = [31] <= 37
A[2:2] = [] >= 37
S = [(4, 8)]
...
```

an iterative quicksort

```
def iterative_quicksort(nbrs, verbose=True):
    """
    The iterative version of quicksort
    uses a stack of indices in nbrs.
    """
    stk = []
    stk.insert(0, (0, len(nbrs)))
    while stk != []:
        if verbose:
            print('S =', stk)
        (first, last) = stk.pop(0)
        (i, j, pivot) = partition(nbrs, first, last)
        if verbose:
            check_partition(nbrs, first, last, i, j, pivot)
        if i < last-1:
            stk.insert(0, (i, last))
        if j > first:
            stk.insert(0, (first, j+1))
```

from Recursion to Iteration

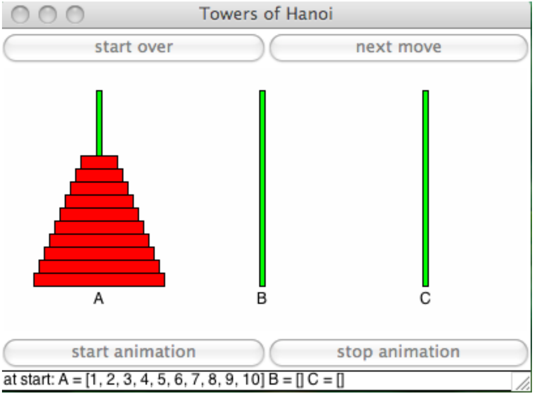
1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays

2. converting recursion into iteration
   - an iterative version with a stack of parameters

3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

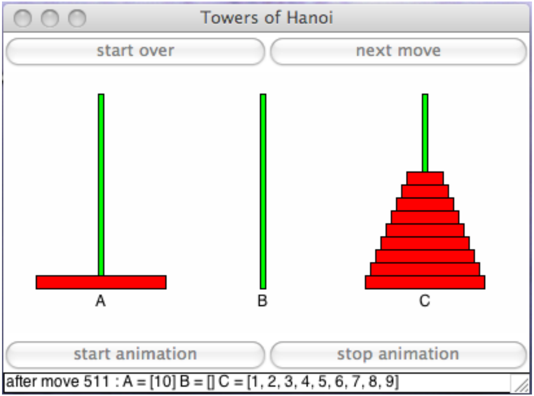
Towers of Hanoi

moving a pile of disks



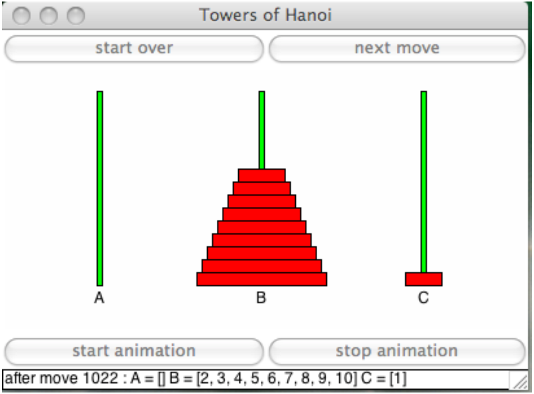
Towers of Hanoi

the middle stage



Towers of Hanoi

the last move



from Recursion to Iteration

1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays

2. converting recursion into iteration
   - an iterative version with a stack of parameters

3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

rules of the game

Move a pile of disks from peg A to B:

1. no larger disk on top of smaller disk,
2. use peg C as intermediary location.

A GUI is just an interface to a program...

1. keep solution in a separate script,
2. GUI is primarily concerned with display.

We need a `"get_next_move"` function.

inverting control in a loop

Consider the following pseudo code:

```
def f(n):
    if n == 0:        # base case
       write result
    else:
       recursive call(s)
```

The recursive `f` controls the calls to `write`.

```
while True:
    result = get_next();
    if no result: break
    write result
```

The pace of writing controls the computation.

using stacks

The `get_next()` function

1. stores all current values of variables and parameters before returning to the caller,
2. when called again, restores all values.

For the towers of Hanoi we will

1. first convert to an iterative solution,
2. then adjust to an inverted function.

Our data structures:

```
A = ('A',range(1,n+1))
B = ('B',[])
C = ('C',[])
```

a recursive solution, the base case

```
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.
    """
    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
```

a recursive solution, the general case

```
def hanoi(nbr, apl, bpl, cpl, k, move):

    if nbr == 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
```

from Recursion to Iteration

1. Quicksort Revisited
   - using arrays
   - partitioning arrays via scan and swap
   - recursive quicksort on arrays
2. converting recursion into iteration
   - an iterative version with a stack of parameters
3. Inverting Control in a Loop
   - a GUI for the towers of Hanoi
   - an interface to a recursive function
   - inverting an iterative solution

an iterative solution for the towers of Hanoi

A stack of arguments of function calls:

```
stk = [(n, 'A', 'B', 'C', k)] # symbols on the stack
while len(stk) > 0:
    top = stk.pop(0)
```

The recursive code:

```
if n == 1:
    move disk from A to B
else:
    move n-1 disks from A to C, B is auxiliary
    move n-th disk from A to B
    move n-1 disks from C to B, A is auxiliary
```

Not only arguments of function calls go on the stack!

moves on the stack

Observe that `B[1].insert(0,A[1].pop(0))` is performed in both the base case and the general case.

In all cases, we move a disk from A to B,
but only in the base case can we execute directly,
in the general case we must store the move.

A move is stores as a string on the stack:

```
top = stk.pop(0)
if isinstance(top, str):
    eval(top)
```

The move is stores as `B[1].insert(0,A[1].pop(0))` ready for execution, triggered by `eval`.

iterative_hanoi, part I

```
def iterative_hanoi(nbr, A, B, C, k):
    """
    The iterative version uses a stack of function calls.
    On the stack are symbols for the piles,
    not the actual piles!
    """
    stk = [(nbr, 'A', 'B', 'C', k)]
    cnt = 0
    while len(stk) > 0:
        top = stk.pop(0)
        if isinstance(top, str):
            eval(top)
            if top[0] != 'w':
                cnt = cnt + 1  # a move, not a write
        else:
```

iterative_hanoi, part II

```
else:
    (nbr, sa, sb, sc, k) = top
    move = sb + '[1].insert(0,' + sa + '[1].pop(0))'
    if nbr == 1:
        # move disk from A to B
        eval(move)
        cnt = cnt + 1
        write(k, cnt, nbr, A, B, C)
    else: # observe that we swap the order of moves!
        # move nbr-1 disks from C to B, A is auxiliary
        stk.insert(0, (nbr-1, sc, sb, sa, k+1))
        # move nbr-th disk from A to B
        stk.insert(0, ("write(%d,cnt,%d,A,B,C)" % (k, nbr)))
        stk.insert(0, move)
        # move nbr-1 disks from A to C, B is auxiliary
        stk.insert(0, (nbr-1, sa, sc, sb, k+1))
```

inverting hanoi

```
def get_next_move(stk, A, B, C):
    """
    Computes the next move, changes the stack stk,
    and returns the next move to the calling routine.
    """

def inverted_hanoi(nbr, apl, bpl, cpl, k):
    """
    This inverted version of the towers of Hanoi gives
    the control to the writing of the piles.
    """
    stk = [(nbr, 'A', 'B', 'C', k)]
    cnt = 0
    while True:
        move = get_next_move(stk, apl, bpl, cpl)
        if move == '':
            break
        cnt = cnt + 1
        pre = 'after move %d :' % cnt
        write_piles(pre, apl, bpl, cpl)
```

body of get_next_move

```
while len(stk) > 0:
    top = stk.pop(0)
    if isinstance(top, str):
        eval(top)
        return top
    else:
        (nbr, sap, sbp, scp, k) = top
        move = sbp + '[1].insert(0,' + sap + '[1].pop(0))'
        if nbr == 1:
            eval(move) # move disk from A to B
            return move
        else: # observe that we swap the order of moves!
            # move nbr-1 disks from C to B, A is auxiliary
            stk.insert(0, (nbr-1, scp, sbp, sap, k+1))
            # move nbr-th disk from A to B
            stk.insert(0, move)
            # move nbr-1 disks from A to C, B is auxiliary
            stk.insert(0, (nbr-1, sap, scp, sbp, k+1))
    return ''
```

Exercises

1. Give Python code to enumerate all permutations of an array *without* making a copy of the array.
2. Two natural numbers m and n are input to the Ackermann function A. For $m = 0$: $A(0, n) = n + 1$, for $m > 0$: $A(m, 0) = A(m - 1, 1)$, and for $m > 0$, $n > 0$: $A(m, n) = A(m - 1, A(m, n - 1))$.
   1. Give a recursive Python function for A.
   2. Turn the recursive function into an iterative one.
3. Write an iterative version of the GUI to draw Hilbert's space filling curves.