Recursive Algorithms

1. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively
2. Accumulating Parameters
   - tracing recursive functions automatically
   - computing with accumulating parameters
3. Recursive Problem Solving
   - check if a word is a palindrome

MCS 275 Lecture 8
Programming Tools and File Management
Jan Verschelde, 27 January 2017

Recursive Algorithms

1. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively
2. Accumulating Parameters
   - tracing recursive functions automatically
   - computing with accumulating parameters
3. Recursive Problem Solving
   - check if a word is a palindrome

computing factorials recursively

rule based programming

Let n be a natural number.
By $n!$ we denote *the factorial* of n.

Its recursive definition is given by two rules:

1. for $n \leq 1$: $n! = 1$
2. if we know the value for $(n - 1)!$
   then $n! = n \times (n - 1)!$

Recursion is similar to mathematical proof by induction:

1. first we verify the trivial or base case,
2. assuming the statement holds for all values
   smaller than n – the induction hypothesis –
   we extend the proof to n.

the function factorial

```
def factorial(nbr):
    """
    Computes the factorial of nbr recursively.
    """
    if nbr <= 1:
        return 1
    else:
        return nbr*factorial(nbr-1)

def main():
    """
    Prompts the user for a number
    and returns the factorial of it.
    """
    nbr = int(input('give a number n : '))
    fac = factorial(nbr)
    print('n! = ', fac)
    print('len(n!) = ', len(str(fac)))
```

tracing recursive functions

Calling `factorial` for $n = 5$:

```
factorial(5) call #0: call for n-1 = 4
 factorial(4) call #1: call for n-1 = 3
  factorial(3) call #2: call for n-1 = 2
   factorial(2) call #3: call for n-1 = 1
    factorial(1) call #4: base case, return 1
   factorial(2) call #3: returning 2
  factorial(3) call #2: returning 6
 factorial(4) call #1: returning 24
factorial(5) call #0: returning 120
```

Computes in the returns:

return `1`, `1*2`, `1*2*3`, `1*2*3*4`, `1*2*3*4*5`

Recursive Algorithms

1. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively

2. Accumulating Parameters
   - tracing recursive functions automatically
   - computing with accumulating parameters

3. Recursive Problem Solving
   - check if a word is a palindrome

running the recursive factorial

```
$ python factorial.py
give a number : 79
n! =  894618213078297528685144171539831652
069808216779571907213868063227837990693501
8605333618108410101760000000000000000000
len(n!) =  117
```

Exploiting Python long integers:

```
$ python factorial.py
give a number : 1234
...
RuntimeError: maximum recursion depth exceeded
```

An exception handler will compute $n!$ iteratively.

stack of function calls

The execution of recursive functions
requires a stack of function calls.

For example, for `n = 5`, the stack grows like

```
    factorial(1) call #4: base case, return 1
   factorial(2) call #3: returning 2
  factorial(3) call #2: returning 6
 factorial(4) call #1: returning 24
factorial(5) call #0: returning 120
```

New function calls are pushed on the stack.
Upon return, a function call is popped off the stack.

computing factorials iteratively

```
def factexcept(nbr):
    """
    When the recursion depth is exceeded
    the factorial of nbr is computed iteratively.
    """
    if nbr <= 1:
        return 1
    else:
        try:
            return nbr*factexcept(nbr-1)
        except RuntimeError:
            print('run time error raised')
            fac = 1
            for i in range(2, nbr+1):
                fac = fac*i
            return fac
```

Recursive Algorithms

1. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively

2. Accumulating Parameters
   - tracing recursive functions automatically
   - computing with accumulating parameters

3. Recursive Problem Solving
   - check if a word is a palindrome

tracing recursive functions

Tracing the execution of a recursive function means: displaying for each function call:

1. the value for the input parameters
2. what is computed inside the function
3. the return value of the function

$\rightarrow$ we need the number of each function call

Use an *accumulating parameter* `k`:

```
def factotrace(nbr, k):
```

increment `k` with each recursive call:

```
    return factotrace(nbr-1, k+1)
```

running `factotrace.py`

Call `factotrace` for `nbr` = 5 and `k = 0`:

```
factotrace(5,0): call for nbr-1 = 4
 factotrace(4,1): call for nbr-1 = 3
  factotrace(3,2): call for nbr-1 = 2
   factotrace(2,3): call for nbr-1 = 1
    factotrace(1,4): base case, return 1
   factotrace(2,3): returning 2
  factotrace(3,2): returning 6
 factotrace(4,1): returning 24
factotrace(5,0): returning 120
```

At call `k`, we indent with `k` spaces.

the function `factotrace.py`

```
def factotrace(nbr, k):
    """
    Prints out trace information in call k
    the initial value for k should be zero.
    """
    prt = k*' '
    prt = prt + 'factotrace(%d,%d):' % (nbr, k)
    if nbr <= 1:
        print(prt + ' base case, return 1')
        return 1
    else:
        print(prt + ' call for nbr-1 = ' + str(nbr-1))
        fac = nbr*factotrace(nbr-1, k+1)
        print(prt + ' returning %d' % fac)
        return fac
```

Recursive Algorithms

1. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively
2. Accumulating Parameters
   - tracing recursive functions automatically
   - computing with accumulating parameters
3. Recursive Problem Solving
   - check if a word is a palindrome

factorial in accumulator

Like we add for the number of the function call,
we can multiply in the accumulator.

```
def factaccu(nbr, fac):
    """
    Accumulates the factorial of nbr in fac
    call factaccu initially with fac = 1.
    """
    if nbr <= 1:
        return fac
    else:
        return factaccu(nbr-1, fac*nbr)
```

tracing factorial computations

Call `factaccu` for `nbr = 5` and `fac = 1`:

```
factaccu(5,1) call #0: call for nbr-1 = 4
 factaccu(4,5) call #1: call for nbr-1 = 3
  factaccu(3,20) call #2: call for nbr-1 = 2
   factaccu(2,60) call #3: call for nbr-1 = 1
    factaccu(1,120) call #4: returning 120
   factaccu(2,60) call #3: returning 120
  factaccu(3,20) call #2: returning 120
 factaccu(4,5) call #1: returning 120
factaccu(5,1) call #0: returning 120
```

Computes `1*5`, `1*5*4`, `1*5*4*3`, `1*5*4*3*2`,
returns 120.

automatic tracing of `factaccu`

```
def factatrace(nbr, fac, k):
    """
    Accumulates the factorial of nbr in fac,
    k is used to trace the calls
    initialize fac to 1 and k to 0.
    """
    prt = k*' '
    prt = prt + 'factatrace(%d,%d,%d)' % (nbr, fac, k)
    if nbr <= 1:
        print(prt + ' returning ' + str(fac))
        return fac
    else:
        print(prt + ' call for nbr-1 = ' + str(nbr-1))
        result = factatrace(nbr-1, fac*nbr, k+1)
        print(prt + ' returning %d' % result)
        return result
```

the output of `factatrace`

Call `factatrace` for `nbr = 5`, `fac = 1`, `k = 0`:

```
factatrace(5,1,0) call for nbr-1 = 4
 factatrace(4,5,1) call for nbr-1 = 3
  factatrace(3,20,2) call for nbr-1 = 2
   factatrace(2,60,3) call for nbr-1 = 1
    factatrace(1,120,4) returning 120
   factatrace(2,60,3) returning 120
  factatrace(3,20,2) returning 120
 factatrace(4,5,1) returning 120
factatrace(5,1,0) returning 120
```

Recursive Algorithms

1. Recursive Functions
   - computing factorials recursively
   - computing factorials iteratively

2. Accumulating Parameters
   - tracing recursive functions automatically
   - computing with accumulating parameters

3. Recursive Problem Solving
   - check if a word is a palindrome

palindromes

If reading a word forwards and backwards is the same, then the word is a palindrome.

Examples: mom, dad, rotor.

```
>>> s = 'motor'
>>> L = [c for c in s]
>>> L
['m', 'o', 't', 'o', 'r']
>>> L.reverse()
>>> L
['r', 'o', 't', 'o', 'm']
>>> t = ''.join(L)
>>> t
'rotom'
>>> s == t
False
```

Palindromes

If reading a word forwards and backwards is the same, then the word is a palindrome.

Examples: mom, dad, rotor.

Problem: define the function

```
def is_palindrome(word):
    """
    Returns True if word is a palindrome,
    and returns False otherwise.
    """
```

Three base cases:

1. the word is empty or only one character long
2. first and last character are different
3. the word consists of two equal characters

the function is_palindrome

```
def is_palindrome(word):
    """
    Returns True if word is a palindrome,
    and returns False otherwise.
    """
    if len(word) <= 1:
        return True
    elif word[0] != word[len(word)-1]:
        return False
    elif len(word) == 2:
        return True
    else:
        short = word[1:len(word)-1]
        return is_palindrome(short)
```

the function `main()`

```
def main():
    """
    Prompts the user for a word and checks
    if it is a palindrome.
    """
    word = input('give a word : ')
    prt = 'the word \"' + word + '\" is '
    if not is_palindrome(word):
        prt = prt + 'not '
    prt = prt + 'a palindrome'
    print(prt)

if __name__ == "__main__":
    main()
```

running the script `palindromes.py`

Giving on input a string of characters:

```
$ python palindromes.py
give a word : palindromes
the word "palindromes" is not a palindrome
```

Because of the `input()` returns a string:

```
$ python palindromes.py
give a word : 1234321
the word "1234321" is a palindrome
```

The palindrome tester works just as well on numbers.

Exercises

1. The nth Fibonacci number F_n is defined for $n \geq 2$ as $F_n = F_{n-1} + F_{n-2}$ and $F_0 = 0$, $F_1 = 1$. Give a Python function `Fibonacci` to compute F_n.
2. Use an accumulating parameter `k` to `Fibonacci` to count the function calls. When tracing the execution, print with `k` spaces as indentations.
3. Write an equivalent C function to compute factorials recursively. Use a main interactive program to test it.
4. Extend `is_palindrome` with an extra accumulating parameter `k` to keep track of the function calls. Trace the execution with this extended function, using `k` spaces as indentations.
5. Write a recursive function to sum a list of numbers.

Summary and Assignments

Background material for this lecture:

- §5.5 in *Computer Science: an overview*,
- start of chapter 9 of *Python Programming*.