Ceng 111 – Fall 2015
Week 11b

Iteration

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.
Alternatives to the naïve version of recursive fibonacci

Store intermediate results:

```python
def fib(n):
    results = [-1]*(n+1)
    results[0] = 0
    results[1] = 1
    return recursive_fib(results, n)

def recursive_fib(results, n):
    if results[n] < 0:
        results[n] = recursive_fib(results, n-1) + recursive_fib(results, n-2)
    else:
        print "using previous result"
    return results[n]
```

```bash
>>> fib(6)
using previous result
using previous result
using previous result
using previous result
using previous result
using previous result
```

2015 S. Kalkan & G. Ucoluk & A. Cosar - CEng 111
Alternatives to the naïve version of recursive fibonacci - 2

- Go bottom to top:
  - Accumulate values on the way

```python
def fib(n):
    if n == 0:
        return n
    else:
        return recursive_fib(n, 0, 0, 1)

def recursive_fib(n, i, f0, f1):
    if n == i:
        return f1
    else:
        return recursive_fib(n, i+1, f1, f0+f1)
```
Other times to avoid recursion

- When you have a limit on the memory
- When you have a limit on time
- When “divide & conquer” is not trivial/straightforward.
Consider these two implementations:

• The second implementation uses “tail recursion”.

• Tail recursion → the result of the called function is not used by the calling function.

```python
def fact1(n):
    if n == 0:
        return 1
    else:
        return n * fact1(n-1)

def fact2(n):
    fact_helper(n, 1)

def fact_helper(n, r):
    if n == 0:
        return r
    else:
        return fact_helper(n-1, n*r)
```
Tail recursion &
iteration

Then, we can implement the tail-recursion version like on the right.

```python
def fact2(n):
    fact_helper(n, 1)

def fact_helper(n, r):
    if n == 0:
        return r
    else:
        return fact_helper(n-1, n*r)

r = n * r
n = n - 1

while n != 0
```
More properly,

\[
r = 1 \\
\text{while } n \neq 0 \\
r = n \times r \\
n = n - 1
\]

```python
def fact2(n):
    fact_helper(n, 1)

def fact_helper(n, r):
    if n == 0:
        return r
    else:
        return fact_helper(n-1, n*r)
```
Iteration in Python

- **while statement**

```python
1 while <condition>:
2     <statements>
```

- **Example:**

```python
1 L = [2, 4, -10, "c"]
2 i = 0
3 while i < len(L):
4     print L[i], "@
5     i += 1
```
Iteration in Python

- for statement:

```python
1 for <var> in <list>:
2     <statements>
```

- Example:

```python
1 for x in [2, 4, -10, "c"]:
2     print x, "@"
```

```plaintext
2 @
4 @
-10 @
c @
```
Examples for Iteration

Searching an item in a list

```python
def is_member(Item, List):
    for x in List:
        if Item == x:
            return True
    return False
```

VS.

```python
def is_member(Item, List):
    length = len(List)
    i = 0
    while i < length:
        if Item == List[i]:
            return i
    return -1
```
Today

■ Finish up Iteration

■ Note:
  - THE2
  - Midterm solution session
    - today at 17:45 in BMB-1.
Nested Loops in Python

- You can put one loop within another one
  - No limit on nesting level

```python
for i in range(1, 10):
    print i, ":",
    for j in range(1, i):
        print j, "-",
    print ":"
```

```
1:
2: 1 -
3: 1 - 2 -
4: 1 - 2 - 3 -
5: 1 - 2 - 3 - 4 -
6: 1 - 2 - 3 - 4 - 5 -
7: 1 - 2 - 3 - 4 - 5 - 6 -
8: 1 - 2 - 3 - 4 - 5 - 6 - 7 -
9: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 -
```
Break statements

while <cond-1>:
    <statements-before-break>
    if <cond-2>:
        break
    <statements-after-break>
    <statements-after-while>

for <var> in <list>:
    <statements-before-break>
    if <cond>:
        break
    <statements-after-break>
    <statements-after-for>
```
1 x = 4
2 List = [1, 4, -2, 3, 8]
3 for m in List:
   4     print m
   5     if m == x:
   6         print "I have found a match"
   7         break
```
Continue statements

```python
while <cond-1>:
    <statements-before-continue>
    if <cond-2>:
        continue
    <statements-after-continue>
    <statements-after-while>
```

```python
for <var> in <list>:
    <statements-before-continue>
    if <cond>:
        continue
    <statements-after-continue>
    <statements-after-for>
```

- `<var>` will point to the next item in the list.
Loops with “else:” parts

- The “else:” part is executed when the loop exits.
- If you use a “break” statement, the “else” part is not executed.

```
1 while <cond>:
2     <statements>
3 else:
4     <else-statement>
```
Types of Iterations

Pre-testing
Types of Iterations
Types of Iterations

Postmod-posttesting
Examples for Iteration

What does the following do?

```python
def f(List):
    length = len(List)
    changed = 1
    while changed:
        changed = 0
        i = 0
        while i < length-1:
            if List[i] > List[i+1]:
                (List[i], List[i+1]) = (List[i+1], List[i])
                changed = 1
            i += 1
```
Counting sort

\[
\text{for } i \leftarrow 1 \text{ to } k \\
\quad \text{do } C[i] \leftarrow 0 \\
\text{for } j \leftarrow 1 \text{ to } n \\
\quad \text{do } C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright \quad C[i] = |\{\text{key} = i\}| \\
\text{for } i \leftarrow 2 \text{ to } k \\
\quad \text{do } C[i] \leftarrow C[i] + C[i-1] \quad \triangleright \quad C[i] = |\{\text{key} \leq i\}| \\
\text{for } j \leftarrow n \text{ downto } 1 \\
\quad \text{do } B[C[A[j]]] \leftarrow A[j] \\
\quad C[A[j]] \leftarrow C[A[j]] - 1
\]
def csort(A):
    # Assume that the numbers are in the range 1,...,k
    k = max(A)
    C = [0] * k

    # Count the numbers in A
    for x in A:
        C[x-1] += 1

    # Accumulate the counts in C
    i = 1
    while i < k:
        C[i] += C[i-1] i += 1

    # Place the numbers into correct locations
    B = [0] * len(A)
    for x in A:
        B[C[x-1]-1] = x C[x-1] -= 1

    return B