Ceng 111 – Fall 2021
Week 11a

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.

What do we do for “bulky” problems?

Previously on CENG111!

1. Recursion
   - A powerful tool of the functional paradigm

2. Iteration
   - A powerful tool of the imperative paradigm

Previously on CENG111!

Recursion: an example (cont'd)

$$
\begin{aligned}
N! &= 1 \times 2 \times \cdots \times (N-1) \times N \qquad N \in \mathbf{N},\ \ N > 0 \\
0! &= 1
\end{aligned}
$$

- A careful look at the formal definition:

$$N! = \underbrace{1 \times 2 \times \cdots \times (N-1)}_{(N-1)!} \times N \qquad N \in \mathbf{N},\ \ N > 0$$

$$
\begin{aligned}
N! &= (N-1)! \times N \qquad N \in \mathbf{N},\ \ N > 0 \\
0! &= 1
\end{aligned}
$$

Factorial uses its own definition!

Previously on CENG111!

Recursion: an example (cont'd)

- Let us look at the pseudo-code:

$$
\begin{aligned}
N! &= (N-1)! \times N \qquad N \in \mathbf{N},\ \ N > 0 \\
0! &= 1
\end{aligned}
$$

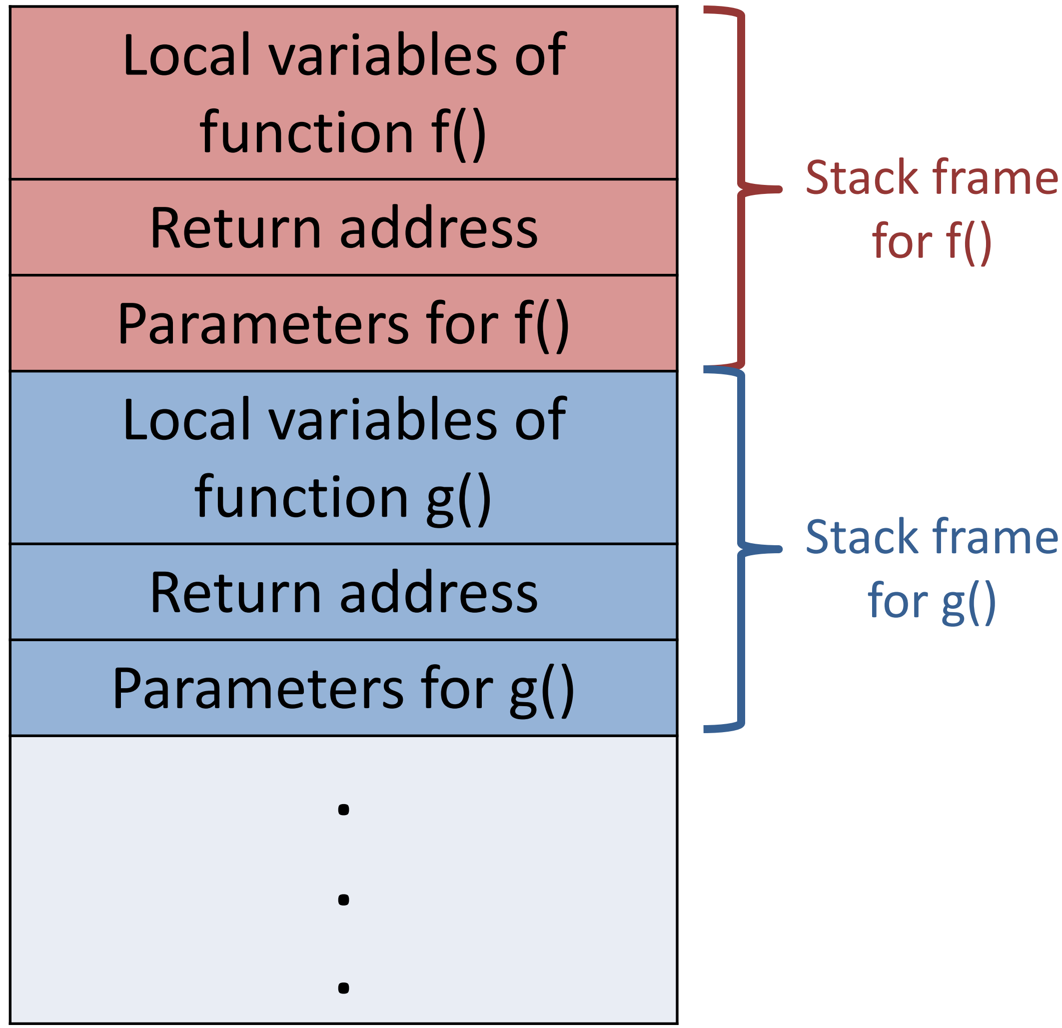
$$
\begin{aligned}
&\textbf{define } factorial(n) \\
&\quad \textbf{if } n \stackrel{?}{=} 0 \textbf{ then} \\
&\qquad \textbf{return} \quad 1 \\
&\quad \textbf{else} \\
&\qquad \textbf{return} \quad n \times factorial(n-1)
\end{aligned}
$$

What happens when we call a function?

Previously on CENG111!

```python
def f(a):
    b = 10
    return b+a

def g(c):
    d = 3
    return c + d + f(c)
```

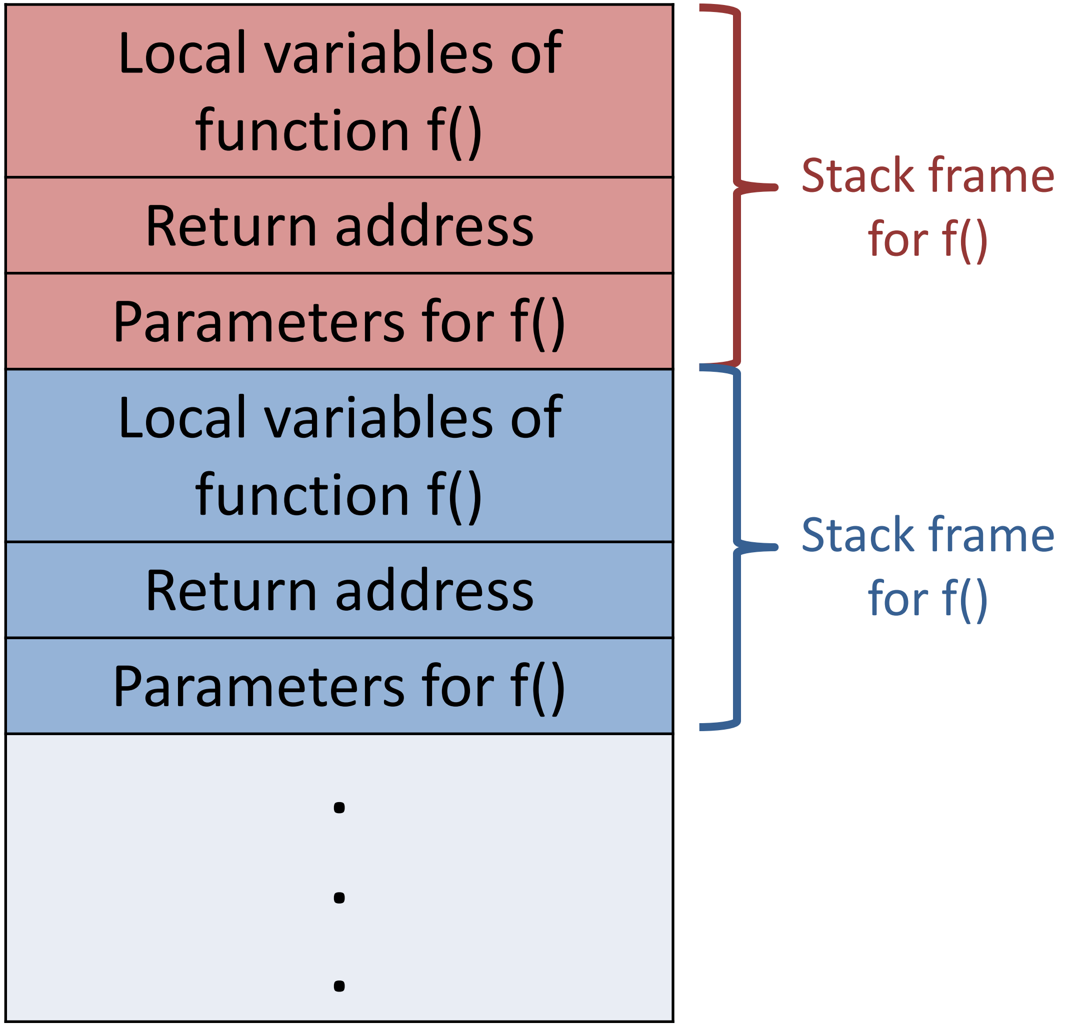


What happens with recursive functions?

Previously on CENG111!

```python
def f(a):
    if a==0:
        return a
    return a+f(a-1)
```


In other words, each call to f() is treated like a different function call.

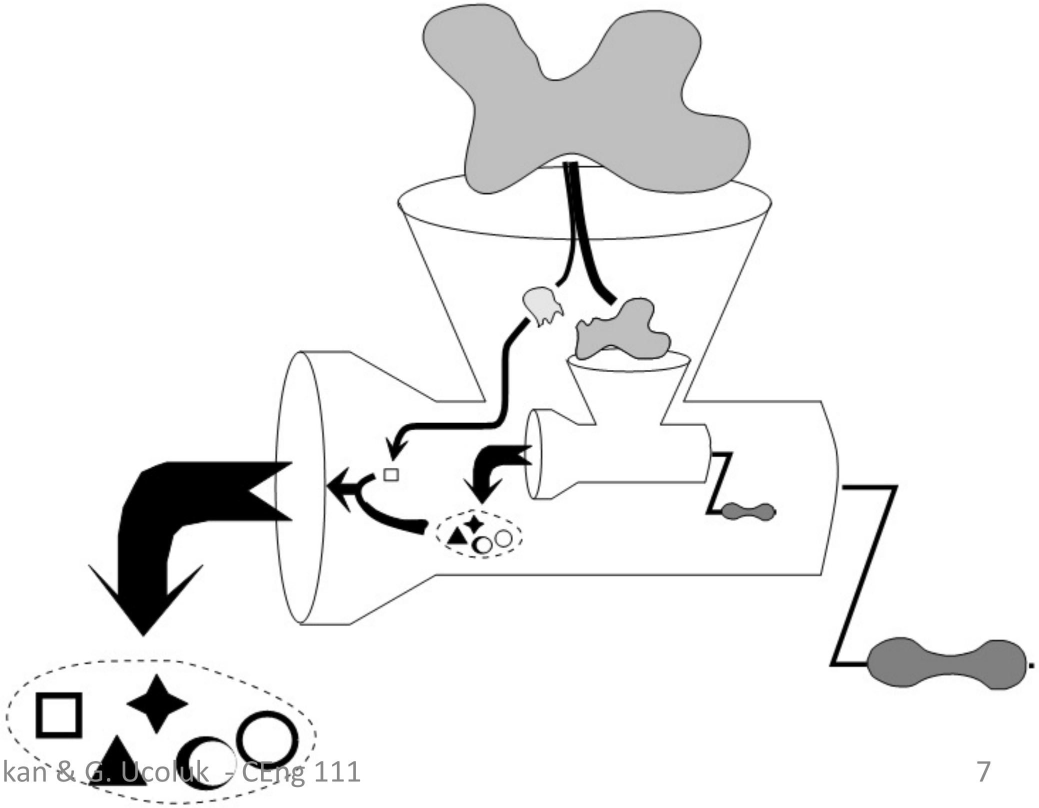


METU Computer Engineering

Previously on CENG111!

For what can we use recursion?

- Not all problems are **suited** to be solved recursively.
- The required properties for the problem:
  1. Scalability
  2. Downsize-ability
  3. Constructability



Today

- Recursion

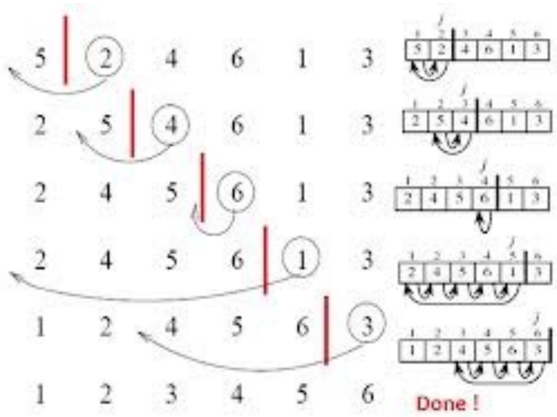
Administrative Notes

- Final:
  - 5 Feb December, Saturday, 13:30

Another example: insert an item into ordered list

```
def insert(x, L):
    if len(L) == 0:
        return [x]
    if x < head(L):
        return [x] + L
    else:
        return [head(L)] + insert(x, tail(L))
```

Another example: insertion sort




```python
def insert(x, L):
    if len(L) == 0:
        return [x]
    if x < head(L):
        return [x] + L
    else:
        return [head(L)] + insert(x, tail(L))


def insertion_sort(L):
    if len(L) <= 1:
        return L
    else:
        return insert(head(L), insertion_sort(tail(L)))
```

Another example: Longest Common Sequence

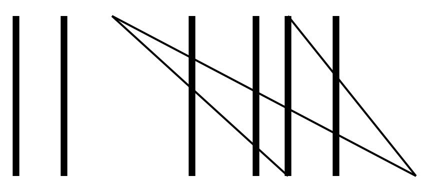
For instance,

Sequence 1: president

Sequence 2: providence

Its LCS is priden.

president



providence

Another example: Longest Common Sequence

$$lcs(i,j)=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ lcs(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i=b_j\ ,\\ \max(lcs(i,j-1),lcs(i-1,j)) & \text{if } i,j>0 \text{ and } a_i\neq b_j.\end{cases}$$

```python
def larger(a, b):
    return a if a > b else b

def lcs(s1, s2):
    if len(s1) == 0 or len(s2) == 0:
        return 0
    elif s1[-1] == s2[-1]:
        return 1+lcs(s1[:-1], s2[:-1])
    else:
        return larger(lcs(s1[:-1], s2), lcs(s1, s2[:-1]))
```

More examples for recursion

http://inventwithpython.com/blog/2011/08/11/recursion-explained-with-the-flood-fill-algorithm-and-zombies-and-cats/

When to avoid recursion!

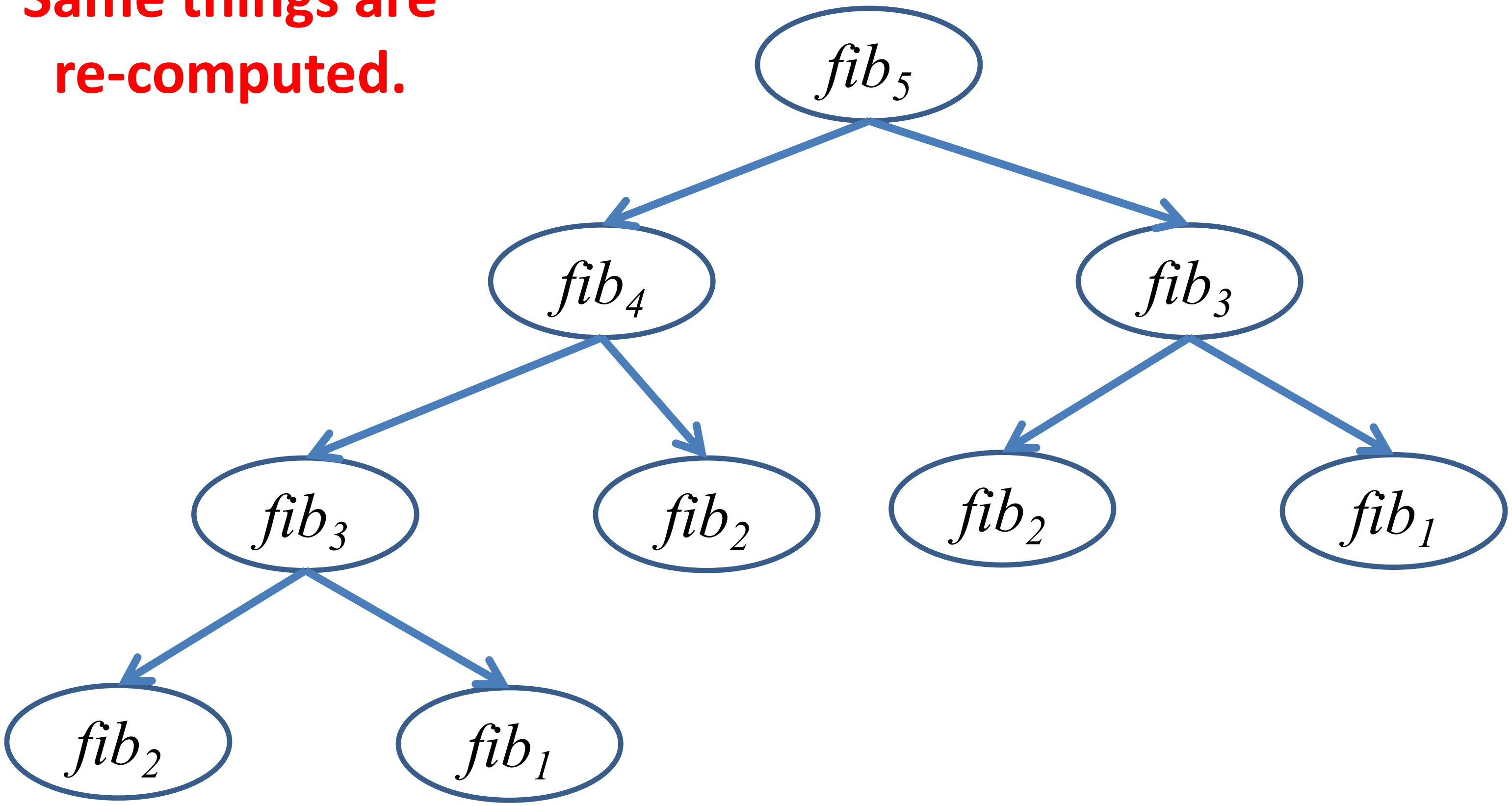
- Example: fibonacci numbers

$$\begin{aligned} fib_{1,2} &= 1 \\ fib_n &= fib_{n-1} + fib_{n-2} \quad \ni \quad n > 2 \end{aligned}$$

$$\begin{array}{l} \textbf{define } fibonacci(n) \\ \quad \textbf{if } n < 3 \textbf{ then} \\ \quad\quad \textbf{return } 1 \\ \quad \textbf{else} \\ \quad\quad \textbf{return } fibonacci(n-1) + fibonacci(n-2) \end{array}$$

So, what is the problem with the recursive definition?

Same things are re-computed.



Alternatives to the naïve version of recursive fibonacci - 1

- Store intermediate results:

```
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]
```

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

Alternatives to the naïve version of recursive fibonacci - 2

Computer Engineering

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

```
def fib(n):
    if n == 0 or n == 1: return n
    return fib_recursive(2, n, 1, 0)


def fib_recursive(i, n, fib_prev, fib_prev_prev):
    if i == n:
        return fib_prev + fib_prev_prev
    else:
        return fib_recursive(i+1, n, fib_prev+fib_prev_prev, fib_prev)
```

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.