



Ceng 111 – Fall 2020

Week 9

Recursion

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.



Today

■ Recursion




Administrative Notes

- Live sessions schedule change
 - Tue 13:40 Session (a.k.a. common session)
 - Wed 10:40 Session
 - ~~Wed 15:40: Section 2~~
 - ~~Thu 15:40: Section 1~~
- Social session
- The labs
- THE1 makeup: 13 December 17:00.
- Midterm: 12 December 13:00
- Office Hour: Friday 16:00



DEALING WITH BULKY PROBLEMS



What happens when the problem gets bigger?

- What do we mean by “bigger”?
- “bigger” means more in size.
- For example, it is very easy to compute the letter grade of a student given his midterm, homework and final grades.
 - How about computing the letter grades of a few hundred students?
- We call these “bigger” cases, “bulky problems”.

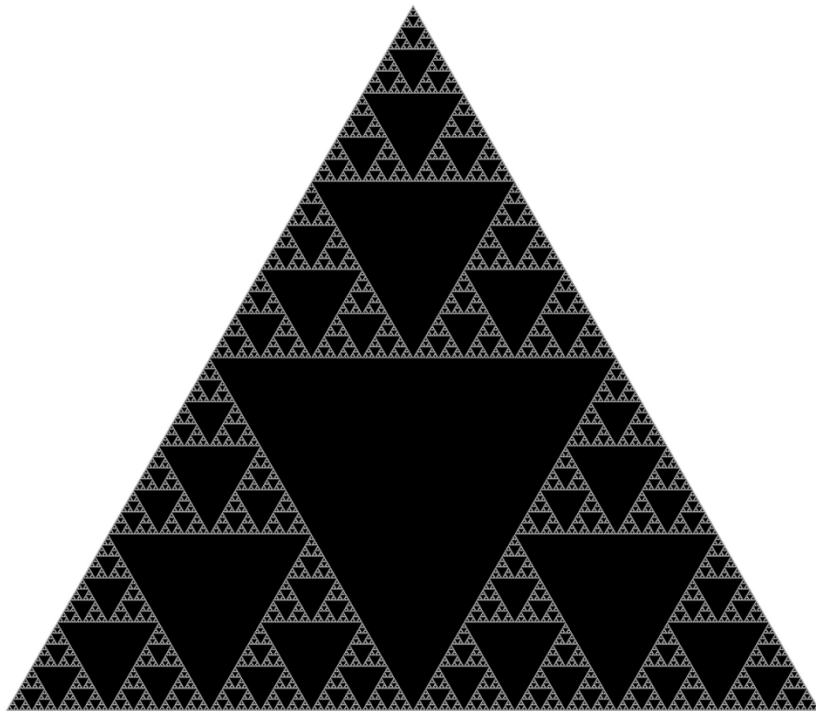


What do we do for “bulky” problems?

1. Recursion
 - A powerful tool of the functional paradigm
2. Iteration
 - A powerful tool of the imperative paradigm

Recursion: An action wizard

■ What is recursion?





Recursion: An example

■ Definition of factorial:

Except for zero, the factorial of a natural number is defined to be the number obtained by multiplication of all the natural numbers lesser or equal to it. The factorial of zero is defined, and is set to be 1.

- More formally:

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

$$0! = 1$$

Recursion: an example (cont'd)

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

- A careful look at the formal definition:

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



$$N! = (N-1)! \times N \quad N \in \mathbf{N}, \ N > 0$$

$$0! = 1$$

**Factorial uses
its own definition!**



Recurrence & Recursion

$$N! = (N - 1)! \times N \quad N \in \mathbf{N}, \quad N > 0$$

$$0! = 1$$

- This is called **recurrence relation/rule**.
- Algorithms which make use of recurrence relations are called **recursive**.

Recursion: an example (cont'd)

- Let us look at the pseudo-code:

$$N! = (N - 1)! \times N \quad N \in \mathbf{N}, \quad N > 0$$

$$0! = 1$$



```
define factorial(n)  
  if n  $\stackrel{?}{=} 0$  then  
    return 1  
  else  
    return n  $\times$  factorial(n - 1)
```





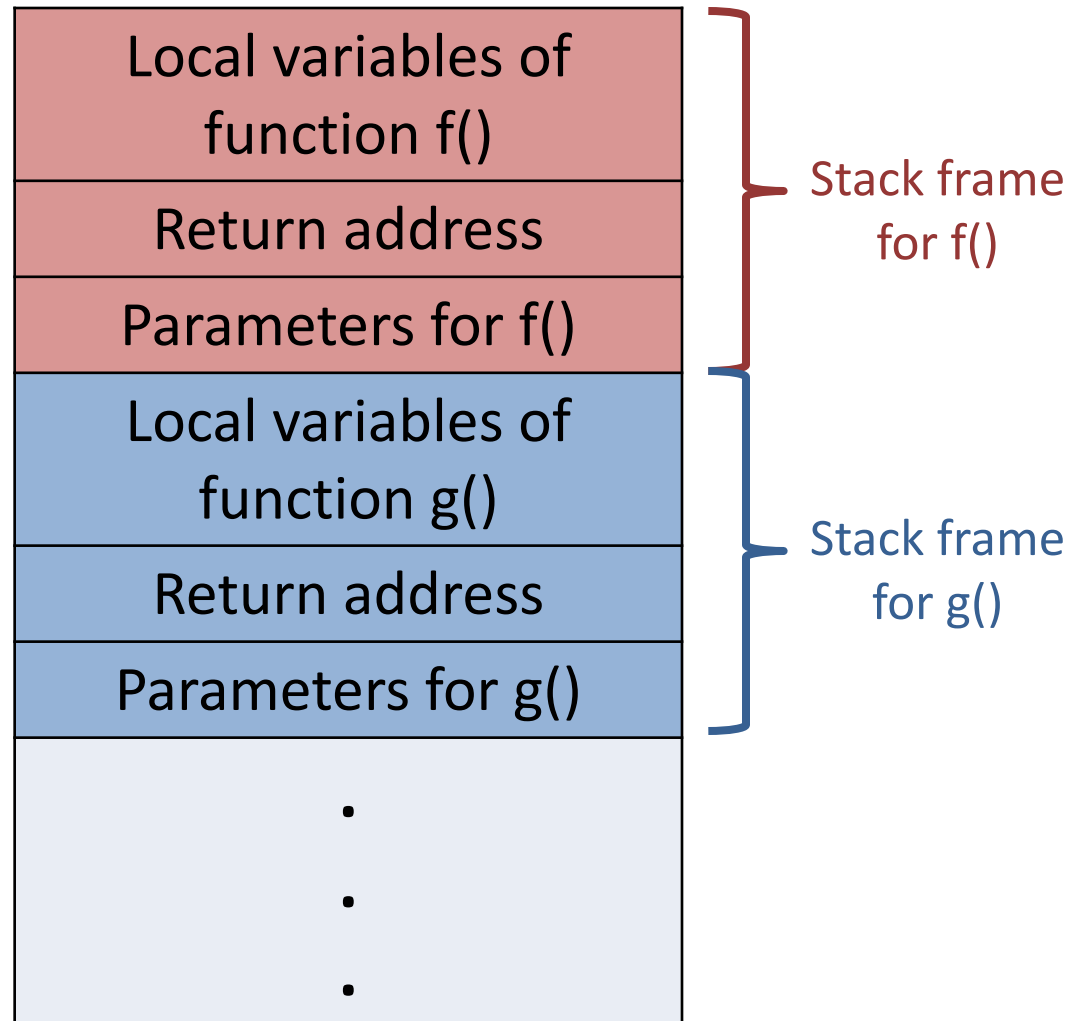
Recursion: another example

- Recursive definition of even or odd (a mutually recursive example):
 - 0 is even.
 - 1 is odd.
 - A number is even if one less the number is odd.
 - A number is odd if one less the number is even.



What happens when we call a function?

```
1 def f(a):  
2     b = 10  
3     return b+a  
4  
5 def g(c):  
6     d = 3  
7     return c + d + f(c)
```



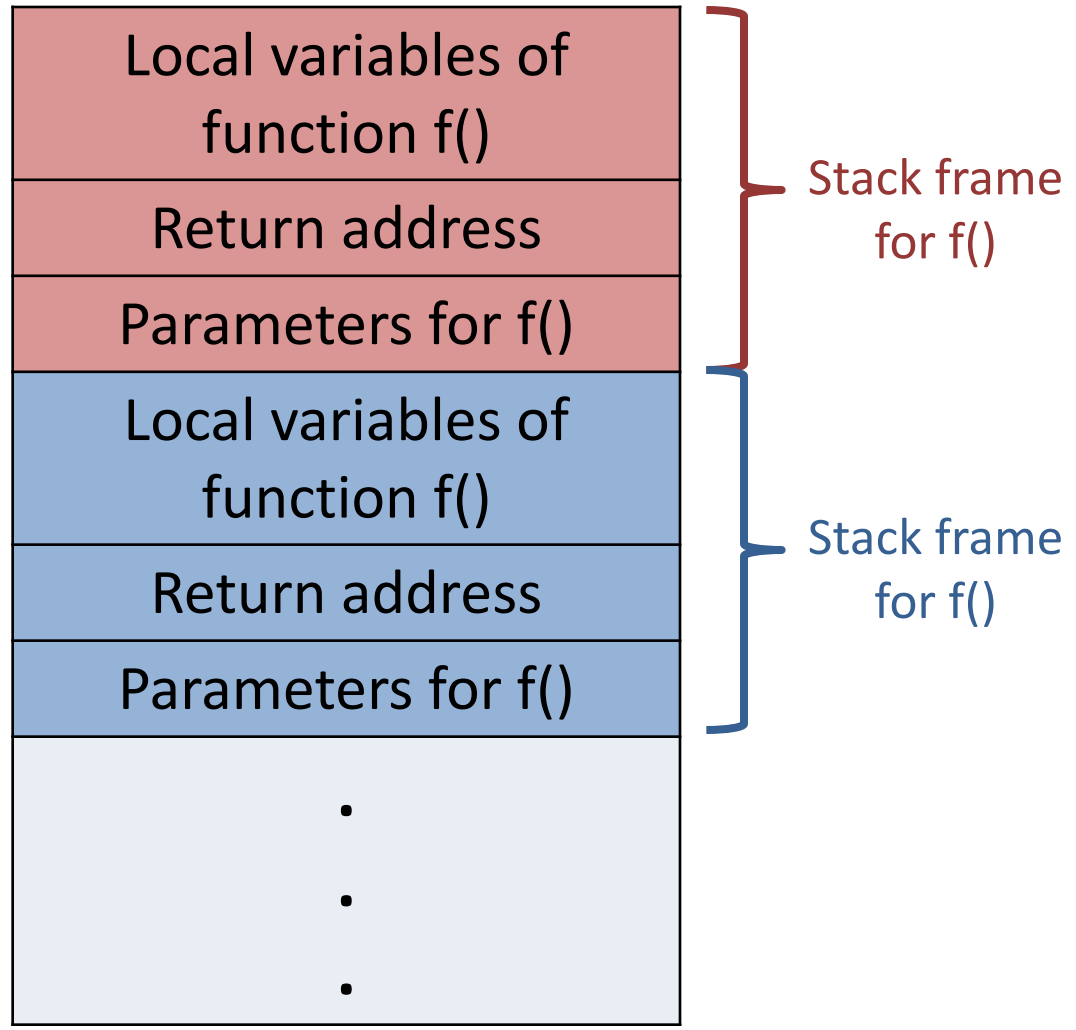


What happens with recursive functions?

```
1 def f(a):  
2     if a==0:  
3         return a  
4     return a+f(a-1)
```



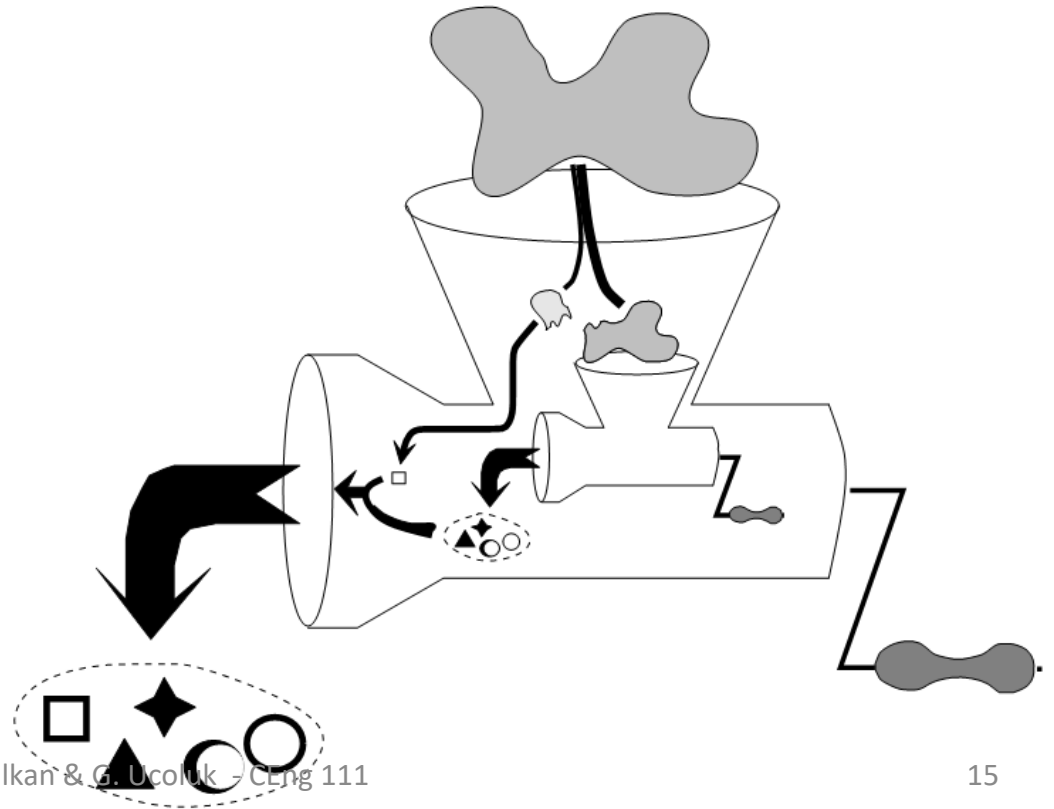
**In other words,
each call to f() is treated
like a different function
call.**





For what can we use recursion?

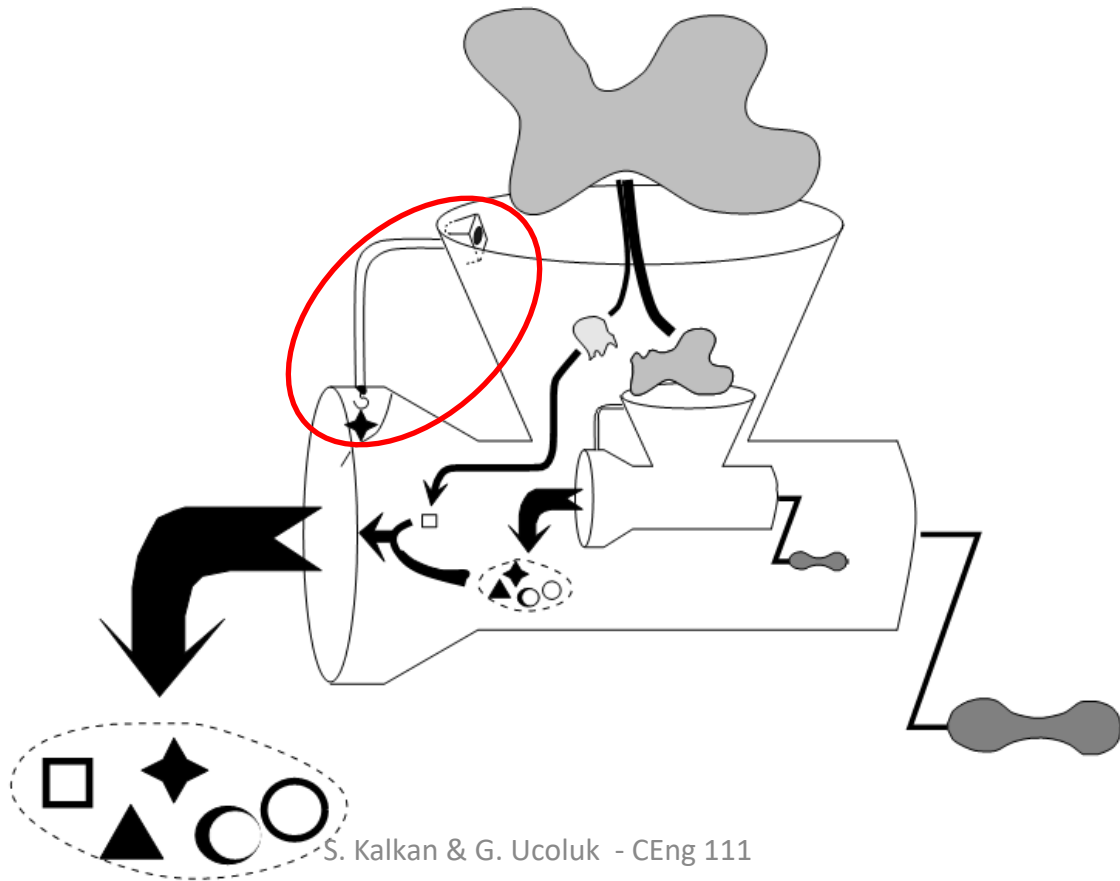
- Not all problems are **suited** to be solved recursively.
- The required properties for the problem:
 1. Scalability
 2. Downsize-ability
 3. Constructability



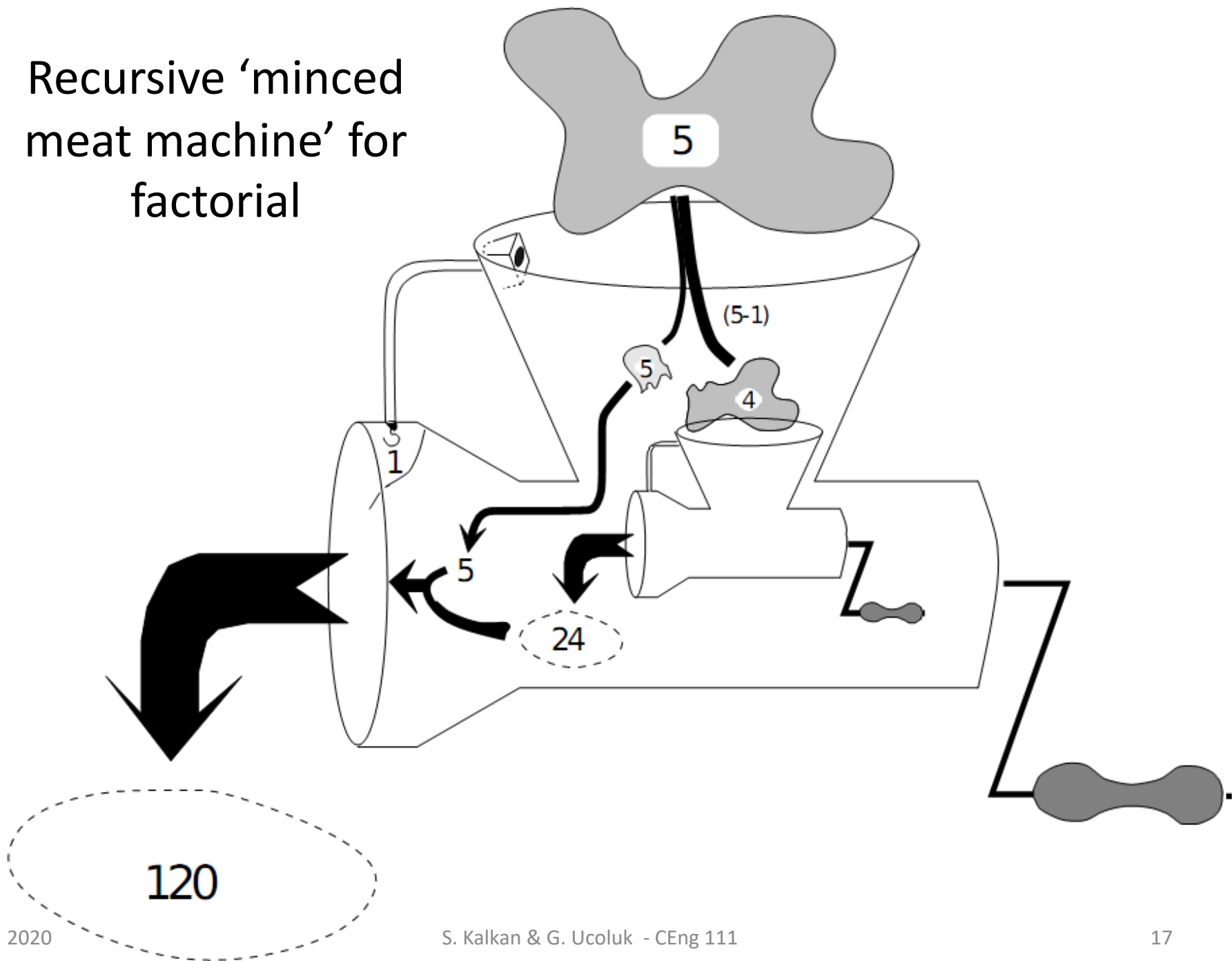


The recursive 'minced meat machine'

- We need a sensor to detect 'no meat'



Recursive 'minced meat machine' for factorial





Now let us lay down some rules while/for using recursion:

RULE I

Looking at the problem's data content decide on a suitable data representation. This data representation shall allow easy shrinking and expansion with the change in problem's data.



Now let us lay down some rules while/for using recursion:

RULE II

Start with the terminating condition. Here you inspect the input data to the function to be minimal (as small as it is allowed). If it is the case, perform what the function is expected to do for this situation. [Depending on what the function is devised for, either you return an appropriate value or you take an appropriate action.]



Now let us lay down some rules while/for using recursion:

RULE III

Now it is turn to handle the non-minimal condition of the input data. You have to imagine that you partition this input data. The key idea of this partitioning is that *at least* one of the pieces shall be of the type of the partitioned data. **You have the right (without any cause) to assume that a further call of the same function with the piece(s) that remained to be of the same type will return/do the correct result/action.** It might be possible to do the partitioning in different ways. Bearing in mind the alternatives apply the next rule.



Now let us lay down some rules while/for using recursion:

RULE IV

Having in hand one hand the correct values/actions returned/-done for the pieces that conserved the input type and in the other hand the pieces themselves which have not conserved the type (if any), you will seek to find the answer to the following question:

With those in my hand how can I construct the desired value/action for the original input data?

If you cannot figure out a solution consider the other alternatives that surfaced with the application of RULE III (if any).




Applications of the rules

Application of Rule I

First question: What data-type shall we use?

The subject to the factorial operation is a natural number and the result is again a *natural number*. So the suitable data-type for the representation of the input data is the integer type. But, due to the nature of the problem we will restrict all our integers to be natural numbers only.



Second question: Is this data-type partitionable in the terms explained above?

Yes, indeed. There exist operations like subtraction and division defined over the *integers* which produce *smaller* pieces (in this case all of integer types as well).



Applications of the rules

Application of Rule II

Having decided about the data-type (integer in this case), it is time to set the terminating condition. We know that the smallest integer the factorial function would allow as argument is 0. And the value to be returned for this input value is 1. So the first line of

```
define factorial(n)  
  if  $n \stackrel{?}{=} 0$  then  
    return 1  
  else  
    ...
```

Applications of the rules

Application of Rule III

The question is: How can we partition the argument (in this case n), so that at least one of the pieces remains to be an integer?

1. Partitioning where the parts are of the same type (for a number n):
 - a. $n = n_1 + n_2$ such that $n_1 = n_2$ or $n_1 = 1 + n_2$
 - b. $n = n_1 + n_2 + \dots + n_k$
2. Partitioning where **one part is small** and the other part is big:
 - a. $n_1 = n - 1, \quad n_2 = 1$
 - b. $n_1 = n - 2, \quad n_2 = 2$



Applications of the rules

- Application of the rule IV (i.e., constructing the result from the results of the partitions).
 - Consider using equal partitioning for factorial (i.e., $n = n_1 + n_2$ such that $n_1 = n_2$ or $n_1 = 1 + n_2$)

Given $n, n_1, n_2, n_1!, n_2!$ can we compute $n!$ in an easy manner?

No! So => Partitioning is important.

Partition in a way that you can construct later.



Applications of the rules

- Consider the partitioning $n_1 = n - 1$, $n_2 = 1$
- Then, we have the following for Rule IV (for construction):

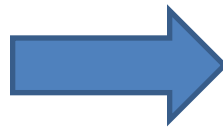
- n

- 1

- $n - 1$

- $1!$

- $(n - 1)!$



$$n! = n \times (n - 1)!$$



Coming back to the factorial example

```
define factorial(n)  
  if  $n \stackrel{?}{=} 0$  then  
    return 1  
  else  
    return  $n \times \textit{factorial}(n - 1)$ 
```



Another example for recursion

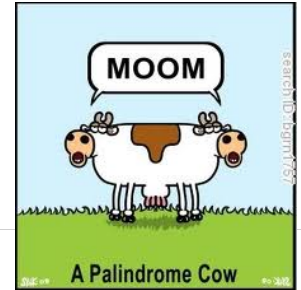
- Reversing a list / string
 - $[a, b, c] \rightarrow [c, b, a]$
- Divide into head ('a') and tail ([b, c]):
 - $\text{head}([a, b, c]) \rightarrow a$
 - $\text{tail}([a, b, c]) \rightarrow [b, c]$
- Now, **$\text{reverse}([a, b, c]) = \text{reverse}([b, c]) + [a]$** .

```
define reverse(S)  
  if  $S \stackrel{?}{=} \emptyset$  then  
    return  $\emptyset$   
  else  
    return  $\text{reverse}(\text{tail}(S)) \oplus [\text{head}(S)]$ 
```

Another example: Palindromes

■ Remember palindromes?

- kek, mom, elele, ...



```
1 def is_palindrome(String):
2     if len(String) < 2:
3         return True
4     return String[0] == String[-1] and is_palindrome(String[1:-1])
5
6 >>> is_palindrome("elma")
7 False
8 >>> is_palindrome("elle")
9 True
10 >>> is_palindrome("elele")
11 True
12 >>> is_palindrome("elale")
13 True
14 >>> is_palindrome("elalem")
15 False
```



Searching for a number in a list

■ Binary search

1	3	4	6	7	8	10	13	14
---	---	---	---	---	---	----	----	----

The list of
items (L)

<div><div>2</div><div>If $x < 4$, it has to be on the left</div></div> <div><div>2</div><div>If $x > 4$, it has to be on the right</div></div>				
-48	-13	4	25	109

Query (x)

1 Compare with the
middle item: if they
are equal, we found x

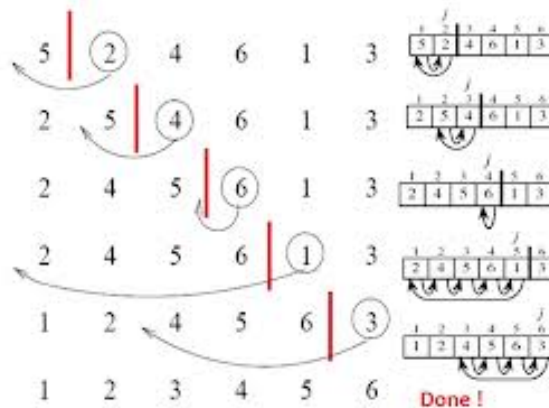


Binary search

```
1 def bin_search(x, L):  
2     if len(L) == 0: return False  
3     Mid_index = len(L) // 2  
4     if x == L[Mid_index]: return True  
5     if x < L[Mid_index]: return bin_search(x, L[0:Mid_index])  
6     if x > L[Mid_index]: return bin_search(x, L[Mid_index+1:])
```



Another example: insertion sort



```
1 def insertOne(element, aList):
2     if len(aList) == 0:
3         return [element]
4     elif element < aList[0]:
5         return [element] + aList
6     else:
7         return aList[0:1] + insertOne(element, aList[1:])
8
9 def insertion_sort(aList):
10     if len(aList) <= 1:
11         return aList
12     else:
13         return insertOne(aList[0], insertion_sort(aList[1:]))
```



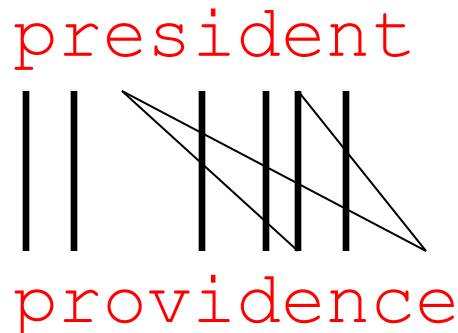

Another example: Longest Common Sequence

For instance,

Sequence 1: president

Sequence 2: providence

Its LCS is priden.



Another example: Longest Common Sequence

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

```
1 def larger(a, b):
2     return a if a > b else b
3
4 def lcs(s1, s2):
5     if len(s1) == 0 or len(s2) == 0:
6         return 0
7     elif s1[-1] == s2[-1]:
8         return 1+lcs(s1[:-1], s2[:-1])
9     else:
10        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

$$fib_{1,2} = 1$$

$$fib_n = fib_{n-1} + fib_{n-2} \quad \exists \quad n > 2$$

```
define fibonacci(n)
```

```
  if n < 3 then
```

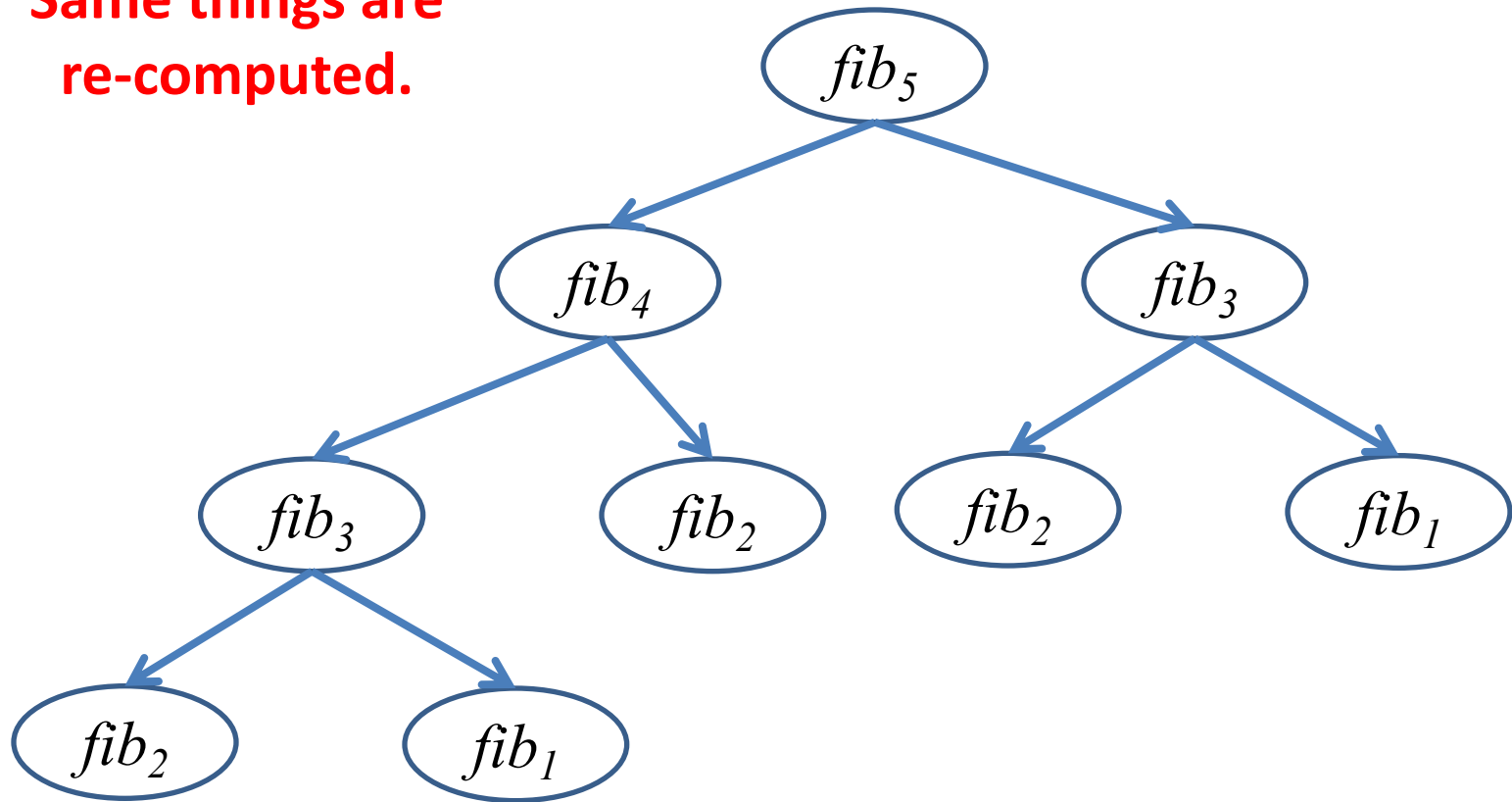
```
    return 1
```

```
  else
```

```
    return fibonacci(n - 1) + fibonacci(n - 2)
```

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:

```
1 def fib(n):
2     results = [-1]*(n+1)
3     results[0] = 0
4     results[1] = 1
5     return recursive_fib(results, n)
6
7 def recursive_fib(results, n):
8     if results[n] < 0:
9         results[n] = recursive_fib(results, n-1)+recursive_fib(results,n-2)
10    else:
11        print "using previous result"
12    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

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

```
1 def fib(n):  
2     if n == 0:  
3         return n  
4     else:  
5         return recursive_fib(n, 0, 0, 1)  
6  
7 def recursive_fib(n, i, f0, f1):  
8     if n == i:  
9         return f1  
10    else:  
11        return recursive_fib(n, i+1, f1, f0+f1)
```