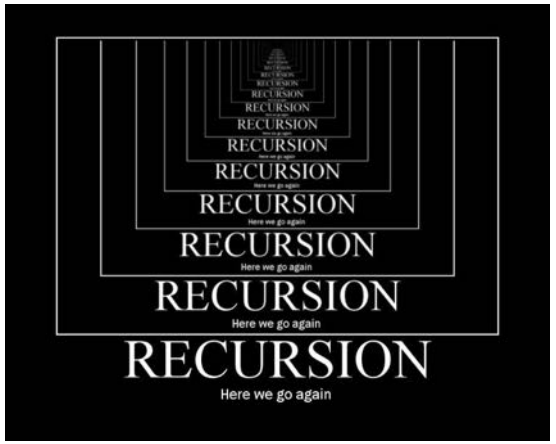
15-110 Principles of Computing – S19

Lecture 20:
Recursion, Divide-and-Conquer

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Recursion



"To understand recursion you must first understand recursion!" ☺

The adjective **recursive** originates from the Latin verb *recurrere*, which means to run back. A recursive definition or a recursive function it is "running back" in the sense of returning to itself (possibly with a simpler instance)

Informal definitions of **recursion:**

- A function that makes a call to itself
- More in general, a *thing* defined in terms of itself or of its type

A matryoshka doll is a wooden Russian doll that contains a smaller sized matryoshka doll

- ➢ This might lead to an **infinite regress!**
- ✓ *Properly defined* **mathematical recursion** solution is never infinite: the recursive calls to itself eventually reach an end, when some minimal input (base case) is reached

Infinite regression case

```
import time
def infinite_regress(n):
    print("At", n)
    time.sleep(0.01)  # makes the process go to "sleep" for the given amount of seconds
    infinite_regress(n-1)
```

```
infinite_regress(5)
# need to interrupt the kernel to stop the running process that would never end, in theory.
# In practice it will end when the memory stack for calling the function will get full.
# When this happens we get the following error:
# RecursionError: maximum recursion depth exceeded while calling a Python object
```

```
At 5
At 4
At 3
At 2
At 1
At 0
At -1
At -2
At -3
At -4
At -5
At -6
At -7
....
At -2955
At -2956
---------------------------------------------------------------------------
RecursionError
 Traceback (most recent call last) <ipython-input-19-6a84001301cd> in <module>()
----> 1 infinite_regress(5)
```

- ✓ To make a meaningful use of recursion, we need to define criteria (**base cases**) for letting the recursive process reach an end (and then possibly reuse the values generated during the recursion process)

A concrete example: computing the factorial

- Mathematical definition (e.g., $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$)

$0! = 1 \qquad 1! = 1$

$n! = n \times (n-1) \times (n-2) \times \cdots \times 1, \qquad \forall\, n \in \mathbb{N}, \qquad n > 1$

- We can rewrite it as:

$0! = 1 \qquad 1! = 1$

$n! = n \times (n-1)! \qquad \forall\, n \in \mathbb{N}, \qquad n > 1$

- In a more general form:

$f(n) \triangleq n! \qquad \forall\, n \geq 0$

$$\begin{cases} f(0) = 1 \qquad f(1) = 1 \\ f(n) = n \cdot f(n-1) \end{cases}$$

Recursive definition of the factorial function

A concrete example: computing the factorial

Recursive definition of the factorial function f:

$$\begin{cases} f(0) = 1 \quad f(1) = 1 & \leftarrow \textbf{Base case}\text{(s): it doesn't require recursion, where the recursive calls of } f \text{ stop} \\ f(n) = n \cdot f(n-1) & \leftarrow \textbf{Recursive case}\text{(s): the function calls itself, with a simpler instance / smaller value} \end{cases}$$

```
def factorial_recursive(n):
    if n == 1 or n == 0:
        return 1
    else:
        return n * factorial_recursive(n-1)
```

```
print(factorial_recursive(5))
```

```
120
```

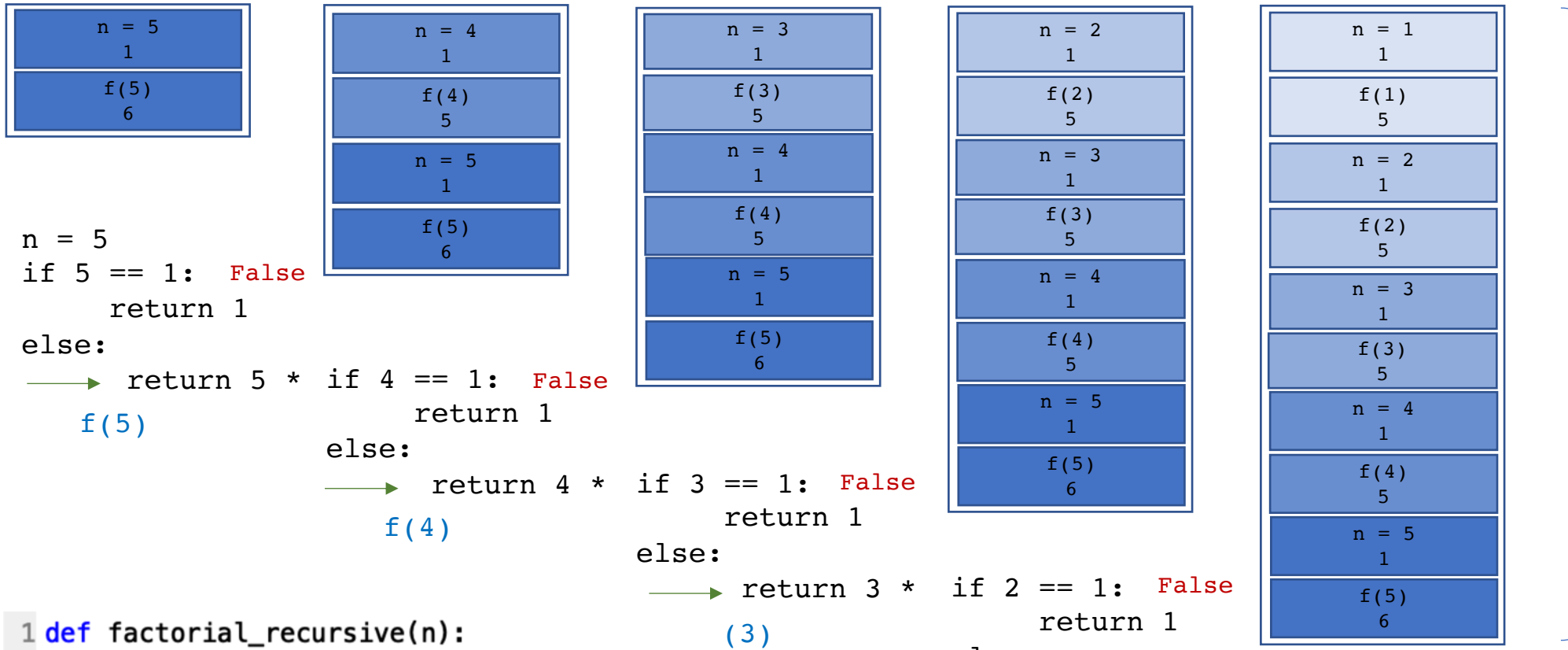
Let's look inside the recursive process …

```
1  def factorial_verbose(n):
2      print("factorial has been called with n =", n)
3      if n == 1 or n == 0:
4          return 1
5      else:
6          res = n * factorial(n-1)
7          print("intermediate result for {} * factorial({}): {}".format(n, n-1,res))
8          return res
9
10 print(factorial_verbose(5))
```

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for 2 * factorial(1): 2
intermediate result for 3 * factorial(2): 6
intermediate result for 4 * factorial(3): 24
intermediate result for 5 * factorial(4): 120
120
```

Let's look inside the recursive process …

- Let's consider the case $n = 5$ and let's expand the function calls:

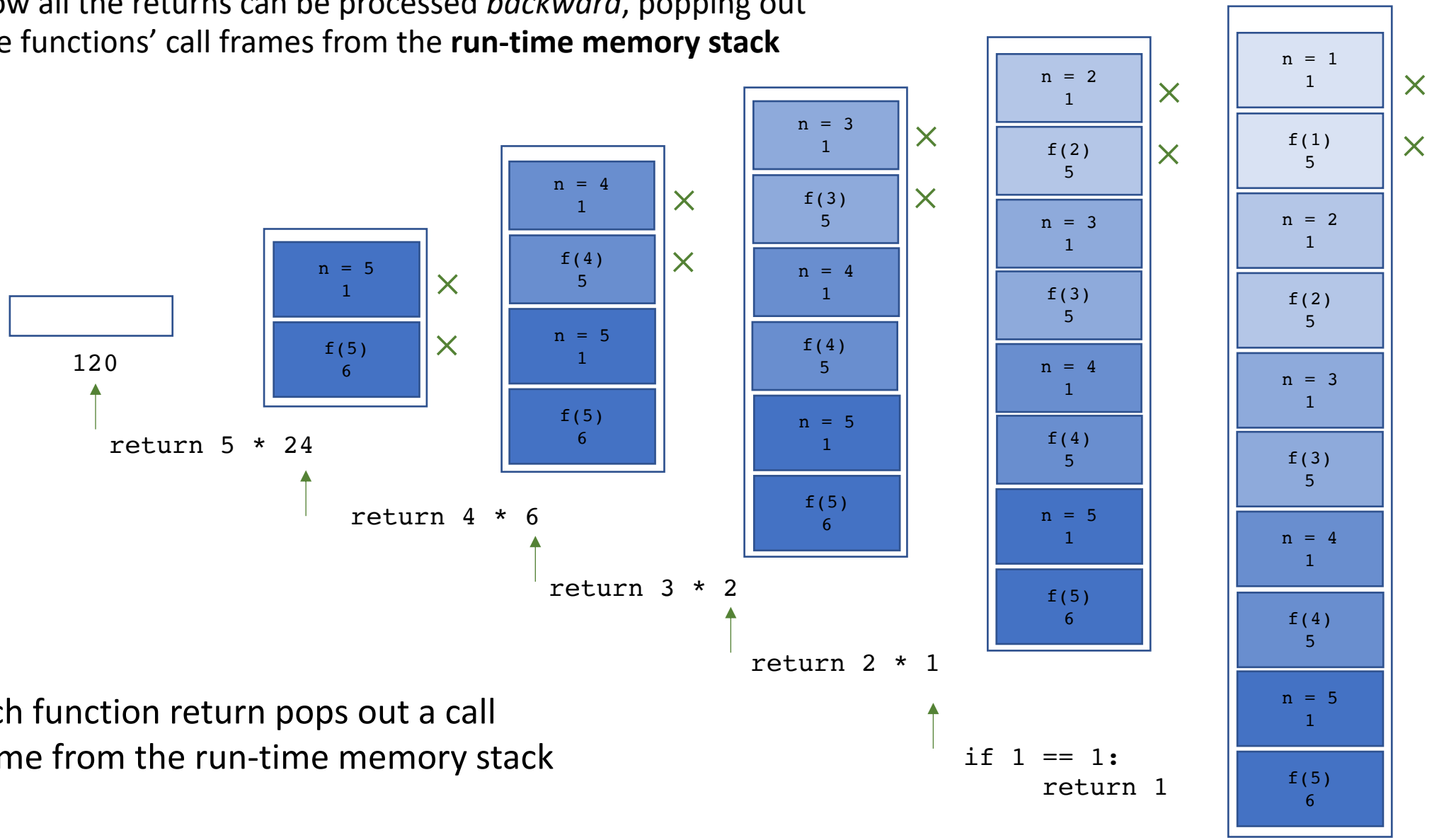



```
1 def factorial_recursive(n):
2     if n == 1:
3         return 1
4     else:
5         return n * factorial_recursive(n-1)
6 print(factorial_recursive(5))
```

Recursive calls end: the *base case*, that does not require recursion, is reached!

Let's look inside the recursive process …

- ✓ Now all the returns can be processed *backward*, popping out the functions' call frames from the **run-time memory stack**



Each function return pops out a call frame from the run-time memory stack

Explosion of the memory stack!

✓ `factorial(100)`

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

- Look at the output: The integer value of the factorial is crazily long!
- The number of digits in an integer number is only limited by the OS memory
- Numbers can grow a lot. The same isn't however in general true for floats

✓ `factorial(1000)`

❖ `factorial(3000)`

RecursionError

Traceback (most recent call last) <ipython-input-5-1f81c569e3b4> in <module>() ----> 1 factorial(3000)

```
import sys
sys.getrecursionlimit()
#sys.setrecursionlimit(4500)
```

```
3000
```

Iteration vs. Recursion for the factorial

- We can easily implement the factorial function using **iteration**

```
def factorial_it(n):
    v = 1
    for i in range(2, n+1):
        v *= i
    return v
```

- Is the iteration-based implementation faster than the recursion-based one?
- Let's check their running times for $n = 0, \cdots, 40$

Iteration seems to be a clear winner,
in this specific case ….

```
n=0, factorial recursion: 0.000002133, factorial iteration: 0.000002867, ratio: 0.744
n=1, factorial recursion: 0.000004467, factorial iteration: 0.000004467, ratio: 1.000
n=2, factorial recursion: 0.000004133, factorial iteration: 0.000004000, ratio: 1.033
n=3, factorial recursion: 0.000004067, factorial iteration: 0.000004133, ratio: 0.984
n=4, factorial recursion: 0.000003133, factorial iteration: 0.000002933, ratio: 1.068
n=5, factorial recursion: 0.000003400, factorial iteration: 0.000003000, ratio: 1.133
n=6, factorial recursion: 0.000003600, factorial iteration: 0.000003133, ratio: 1.149
n=7, factorial recursion: 0.000005733, factorial iteration: 0.000004467, ratio: 1.284
n=8, factorial recursion: 0.000004533, factorial iteration: 0.000003067, ratio: 1.478
n=9, factorial recursion: 0.000005067, factorial iteration: 0.000003400, ratio: 1.490
n=10, factorial recursion: 0.000005400, factorial iteration: 0.000003667, ratio: 1.473
n=11, factorial recursion: 0.000005733, factorial iteration: 0.000003467, ratio: 1.654
n=12, factorial recursion: 0.000008467, factorial iteration: 0.000003600, ratio: 2.352
n=13, factorial recursion: 0.000006333, factorial iteration: 0.000003867, ratio: 1.638
n=14, factorial recursion: 0.000007867, factorial iteration: 0.000007067, ratio: 1.113
n=15, factorial recursion: 0.000011333, factorial iteration: 0.000004200, ratio: 2.698
n=16, factorial recursion: 0.000007733, factorial iteration: 0.000004400, ratio: 1.758
n=17, factorial recursion: 0.000008200, factorial iteration: 0.000004467, ratio: 1.836
n=18, factorial recursion: 0.000008533, factorial iteration: 0.000004467, ratio: 1.910
n=19, factorial recursion: 0.000009000, factorial iteration: 0.000004600, ratio: 1.957
n=20, factorial recursion: 0.000009400, factorial iteration: 0.000004600, ratio: 2.043
n=21, factorial recursion: 0.000013067, factorial iteration: 0.000004800, ratio: 2.722
n=22, factorial recursion: 0.000010400, factorial iteration: 0.000005133, ratio: 2.026
n=23, factorial recursion: 0.000010600, factorial iteration: 0.000005133, ratio: 2.065
n=24, factorial recursion: 0.000011133, factorial iteration: 0.000005333, ratio: 2.088
n=25, factorial recursion: 0.000011400, factorial iteration: 0.000005400, ratio: 2.111
n=26, factorial recursion: 0.000012400, factorial iteration: 0.000005600, ratio: 2.214
n=27, factorial recursion: 0.000012600, factorial iteration: 0.000005600, ratio: 2.250
n=28, factorial recursion: 0.000013200, factorial iteration: 0.000006000, ratio: 2.200
n=29, factorial recursion: 0.000023067, factorial iteration: 0.000010667, ratio: 2.162
n=30, factorial recursion: 0.000013933, factorial iteration: 0.000006067, ratio: 2.297
n=31, factorial recursion: 0.000016200, factorial iteration: 0.000006867, ratio: 2.359
n=32, factorial recursion: 0.000014733, factorial iteration: 0.000007067, ratio: 2.085
n=33, factorial recursion: 0.000024533, factorial iteration: 0.000011600, ratio: 2.115
n=34, factorial recursion: 0.000026733, factorial iteration: 0.000008067, ratio: 3.314
n=35, factorial recursion: 0.000020200, factorial iteration: 0.000008333, ratio: 2.424
n=36, factorial recursion: 0.000022267, factorial iteration: 0.000012533, ratio: 1.777
n=37, factorial recursion: 0.000024333, factorial iteration: 0.000010267, ratio: 2.370
n=38, factorial recursion: 0.000018600, factorial iteration: 0.000008400, ratio: 2.214
n=39, factorial recursion: 0.000019200, factorial iteration: 0.000007333, ratio: 2.618
n=40, factorial recursion: 0.000018133, factorial iteration: 0.000007600, ratio: 2.386
```

Comparing running times: a helper function

- The previous results were produced with the custom helper function below, quite useful, give it a look!

```
import time
def compare_running_times(experiments, trials, algorithms, args=None):
    '''Takes as input two functions/algorithms and compares their running times
       over a given number of experiments/tests (+1). Each experiment consists of a number
       of trials, whose results are averaged. The results of the comparisons are printed out.
       Inputs: experiments (integer) specifies the number of tests, where the index of each
               experiment is used as input for the algorithms. trials (integer) specifies
               how many times each experiment is repeated, accounting for random variability.
               The running times observed in each trial are averaged to compute the average
               running time for each experiment. algorithms (list) is a list
               of two lists, where each list element contains in turn two elements, the
               first one is a string with the name of the algorithm, and the second is a
               variable holding the reference to a function (that implements the algorithm), e.g.,
               algorithms = [ ['label alg 1', function_alg_1],['label alg 2', function_alg_2] ].
               args (list) is a list of two lists with the additional arguments to pass to the
               two functions, if any. e.g. args = [x, y], or arg = [x, None] if the second algorithm
               doesn't take any additional parameter.
        Output: None is returned. However, for each trial, a summary of the observed running
                times for the two algorithms, as well as their ratio, is printed out.
    '''
    for n in range(experiments + 1):
        total = []
        msg = 'n={}, '.format(n)
        for a in (0,1):
            observed_times = []
            for r in range(trials):
                start = time.process_time()
                algorithm = algorithms[a][1]
                if args == None:
                    algorithm(n)
                else:
                    if args[a] != None:
                        algorithm(n, args[a])
                    else:
                        algorithm(n)
                end = time.process_time()
                observed_times.append(end - start)
            average_time = sum(observed_times) / trials
            total.append(average_time)
            msg += '{:s}: {:.9f}, '.format(algorithms[a][0], total[-1])
        msg += 'ratio: {:.3f}'.format(total[0] / total[1])
        print(msg)
```

```
n=0, factorial recursion: 0.000002133, factorial iteration: 0.000002867, ratio: 0.744
n=1, factorial recursion: 0.000004467, factorial iteration: 0.000004467, ratio: 1.000
n=2, factorial recursion: 0.000004133, factorial iteration: 0.000004000, ratio: 1.033
n=3, factorial recursion: 0.000004067, factorial iteration: 0.000004133, ratio: 0.984
n=4, factorial recursion: 0.000003133, factorial iteration: 0.000002933, ratio: 1.068
n=5, factorial recursion: 0.000003400, factorial iteration: 0.000003000, ratio: 1.133
n=6, factorial recursion: 0.000003600, factorial iteration: 0.000003133, ratio: 1.149
n=7, factorial recursion: 0.000005733, factorial iteration: 0.000004467, ratio: 1.284
n=8, factorial recursion: 0.000004533, factorial iteration: 0.000003067, ratio: 1.478
n=9, factorial recursion: 0.000005067, factorial iteration: 0.000003400, ratio: 1.490
n=10, factorial recursion: 0.000005400, factorial iteration: 0.000003667, ratio: 1.473
n=11, factorial recursion: 0.000005733, factorial iteration: 0.000003467, ratio: 1.654
n=12, factorial recursion: 0.000008467, factorial iteration: 0.000003600, ratio: 2.352
n=13, factorial recursion: 0.000006333, factorial iteration: 0.000003867, ratio: 1.638
n=14, factorial recursion: 0.000007867, factorial iteration: 0.000007067, ratio: 1.113
n=15, factorial recursion: 0.000011333, factorial iteration: 0.000004200, ratio: 2.698
n=16, factorial recursion: 0.000007733, factorial iteration: 0.000004400, ratio: 1.758
n=17, factorial recursion: 0.000008200, factorial iteration: 0.000004467, ratio: 1.836
n=18, factorial recursion: 0.000008533, factorial iteration: 0.000004467, ratio: 1.910
n=19, factorial recursion: 0.000009000, factorial iteration: 0.000004600, ratio: 1.957
n=20, factorial recursion: 0.000009400, factorial iteration: 0.000004600, ratio: 2.043
n=21, factorial recursion: 0.000013067, factorial iteration: 0.000004800, ratio: 2.722
n=22, factorial recursion: 0.000010400, factorial iteration: 0.000005133, ratio: 2.026
n=23, factorial recursion: 0.000010600, factorial iteration: 0.000005133, ratio: 2.065
n=24, factorial recursion: 0.000011133, factorial iteration: 0.000005333, ratio: 2.088
n=25, factorial recursion: 0.000011400, factorial iteration: 0.000005400, ratio: 2.111
n=26, factorial recursion: 0.000012400, factorial iteration: 0.000005600, ratio: 2.214
n=27, factorial recursion: 0.000012600, factorial iteration: 0.000005600, ratio: 2.250
n=28, factorial recursion: 0.000013200, factorial iteration: 0.000006000, ratio: 2.200
n=29, factorial recursion: 0.000023067, factorial iteration: 0.000010667, ratio: 2.162
n=30, factorial recursion: 0.000013933, factorial iteration: 0.000006067, ratio: 2.297
n=31, factorial recursion: 0.000016200, factorial iteration: 0.000006867, ratio: 2.359
n=32, factorial recursion: 0.000014733, factorial iteration: 0.000007067, ratio: 2.085
n=33, factorial recursion: 0.000024533, factorial iteration: 0.000011600, ratio: 2.115
n=34, factorial recursion: 0.000026733, factorial iteration: 0.000008067, ratio: 3.314
n=35, factorial recursion: 0.000020200, factorial iteration: 0.000008333, ratio: 2.424
n=36, factorial recursion: 0.000022267, factorial iteration: 0.000012533, ratio: 1.777
n=37, factorial recursion: 0.000024333, factorial iteration: 0.000010267, ratio: 2.370
n=38, factorial recursion: 0.000018600, factorial iteration: 0.000008400, ratio: 2.214
n=39, factorial recursion: 0.000019200, factorial iteration: 0.000007333, ratio: 2.618
n=40, factorial recursion: 0.000018133, factorial iteration: 0.000007600, ratio: 2.386
```

General scheme for designing a recursive function / solution

- Problem: design a (recursive) function f that solves some assigned and well-defined task
  - The recursive function will be called by some external code with some **input arguments**, x, related to the dimension of the task instance

How do we design the recursive function $f(x)$?

- Identify a **recursive relation between the function and itself**, such that at each recursive call the problem dimension x is downsized in some way, and moves towards a base case
- Identify one or more **base cases** for $f(x)$. The base cases, *that depend on the input* x, do not require recursion to be solved, such that they cause the function to immediately return the value and interrupt the regress process

Why do we need recursion?

- Often, we can use iteration instead of recursion, but recursion, when well designed, can be:
  - More *elegant* and *compact*
  - More *efficient, computationally*
- Some problems are *naturally defined in recursive terms* (recurrence equations, iterated maps)
- Some problems might actually *need recursion!*

- However, it may be true that we need some "practice" to learn to think recursively …
- Trying to **enroll the recursive process in our mind is not the way to go**, it wouldn't work, it'd be just frustrating. We can do this easily for iteration, but the almost unbounded regress process happening in recursion would be confusing for our minds
- Recursion is tightly related to the principle of **mathematical induction**, that can help to think recursively …

Mathematical induction

✓ Mathematical induction is a mathematical **proof** technique: given a certain property $P(n)$, prove that it holds for all integer numbers, $n = 0, 1, 2, \cdots$

- Proving by induction requires to prove the *truth* of two cases:
  - **Base case**, that proves that the property is true for $n = 0$ (or $n = 1$), that is for the first integer
  - **Induction case**, that proves that, if the property holds for a generic n (*inductive assumption,* the validity for n becomes a *fact*), it also holds for the *next* natural number, $n + 1$
  - If we can prove that *both* the induction and the base case hold, then we can assert that the property holds for *any* integer
- Induction is a way to perform inference and is at the foundation of the correctness of all computer programs!

Mathematical induction

- E.g., let the property $P(n)$ we want to assert be the following: the sum of the first n integers is expressed as

$$0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \qquad \text{(Gauss formula)}$$

➢ Base case (taking 0, but we could also take 1), $P(0)$: $0 = \frac{0(0+1)}{2}$ True!

➢ Inductive case, where we **assume** that $P(n)$ is true, and we use it to prove that then also $P(n+1)$ is true:

Let's write first the left-hand term for the case $(n+1)$: $(0 + 1 + 2 + \cdots + n) + (n+1)$

This can be rewritten as follows using the assumed *fact* that the formula holds for n:

$$\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)\big((n+1)+1\big)}{2}$$

The last expression is precisely the formula for $n+1$ True!

✓ Since both the base and the inductive case hold, the property holds for all integers

From mathematical induction to recursive function definitions

➢ When programming recursively, think inductively!

✓ Define a recursive (inductive) case, that recursively relates the function f for a generic input x to the value of the function for a *downsized input* $\hat{x}$ (e.g., $\hat{x} = x - 1$)

- When you write the recursive relation, **assume / trust** that the value returned by $f(\hat{x})$ is correct (this is the inductive thinking!) That is, it does the job of correctly computing the function for the input $\hat{x}$.
- E.g., in the case of the factorial, we have assumed that $f(n-1)$ does return the factorial of $n-1$ and we have used it for defining a recursive relation that computes $f(n)$

✓ Define base (non-recursive) cases for the input x, that return the value of the function for the ***minimal admitted inputs*** x, such that when the input regresses to these values, the recursive calls are interrupted and the process stops returning the result by **combining the results from the individual recursive calls**

- E.g., in the case of the factorial, the minimal input is for $n = 1$, that we know how to compute without recursion, the final result is obtained by multiplying the results from each individual function call

Another example: exponentiation

- Problem: compute $f(x, n) = x^n$ recursively, where n is an integer
- Base case: $f(x, 0) = 1$
- Recursive case: $f(x, n) = x \cdot f(x, n - 1)$

```
def exponentiate(x, n):
    if n == 0:
        return 1
    else:
        return x * exponentiate(x, n - 1)
```

```
exponentiate(2, 7)
```

128

```
def exponentiate(x, n):
    if n == 0:
        return 1
    if x == 0:
       return 0:
    else:
        return x * exponentiate(x, n - 1)
```

Optimized version for the special case when $x = 0$

Another simple example: summation

- Problem: compute $f(n) = 0 + 1 + 2 + \cdots n$ recursively, where n is an integer
- Base case: $f(0) = 0$
- Recursive case: $f(n) = n + f(n-1)$

```
def summation(n):
    if n == 0:
        return 0
    else:
        return n + summation(n - 1)
```

```
summation(100)
```

```
5050
```

Fibonacci sequence: naturally *defined* in a recursive way

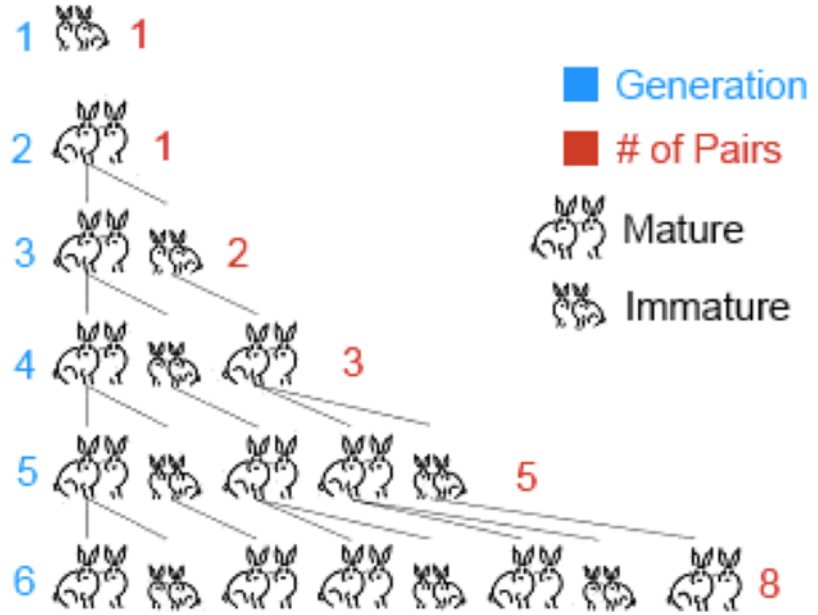
- Let's start with a pair of rabbits, one male, one female. Rabbits are able to mate at the age of one month so that at the end of its second month ($n = 2$) a female can produce another pair of rabbits. Suppose that our rabbits **never die** and that the female **always** produces one new pair (one male, one female) **every month** from the second month on.

➢ How many pairs f rabbits will there be after n months?

➢ Instance of a **discrete-time dynamical system**: the *state* of the system (the number of rabbit pairs) evolves over time under a growth law expressed by recurrence equations

Recurrence equations, two base cases:

$$F(0) = 1$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$



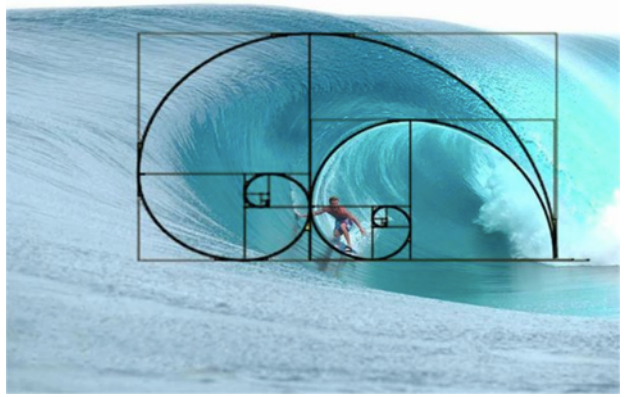
Fibonacci sequence: defined in a recursive way

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

```
for i in range(20):
    print(fibonacci(i))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

- Fibonacci numbers are “pervasive” in natural phenomena
- They capture growth modalities and ratios in nature
- http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html



Fibonacci sequence computed using iteration

- We can easily implement Fibonacci's recurrence equations using *iteration*

```
def fibonacci_iteration(n):
    past_state, new_state = 0, 1
    if n == 0:
        return 0
    for i in range(n-1):
        past_state, new_state = new_state, past_state + new_state
    return new_state
```

- In the code, notice the concurrent assignment of values to multiple variables, that are treated as a tuple. Remember, for tuples, `(a,b) = (2,3)` is equivalent to `a,b = 2,3`, round parentheses are not strictly necessary

Which version, the recursive or the iteration one, is faster?

Fibonacci: recursion vs. iteration

- Let's use our previous helper function to perform a comparison on running times

```
1 compare_running_times(35, 20, [ ['fibonacci recursion', fibonacci], ['fibonacci iteration', fibonacci_iteration]])
```

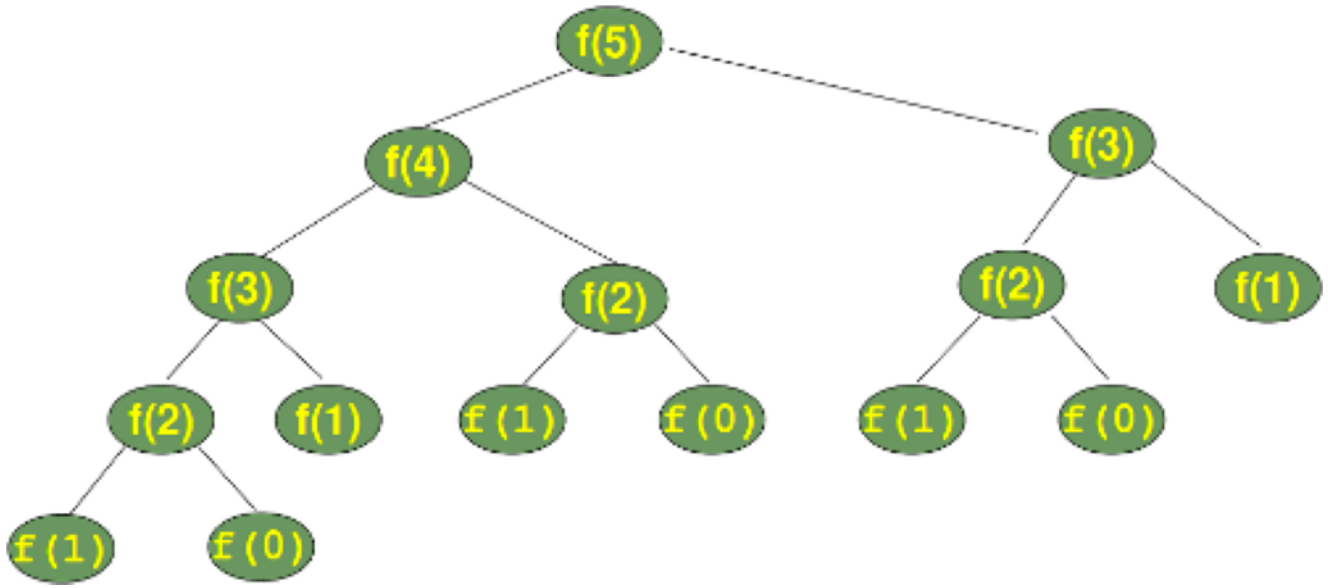
```
n=0, fibonacci recursion: 0.000002100, fibonacci iteration: 0.000002100, ratio: 1.000
n=1, fibonacci recursion: 0.000002000, fibonacci iteration: 0.000002800, ratio: 0.714
n=2, fibonacci recursion: 0.000002900, fibonacci iteration: 0.000003100, ratio: 0.935
n=3, fibonacci recursion: 0.000004500, fibonacci iteration: 0.000003800, ratio: 1.184
n=4, fibonacci recursion: 0.000004200, fibonacci iteration: 0.000003000, ratio: 1.400
n=5, fibonacci recursion: 0.000005900, fibonacci iteration: 0.000003100, ratio: 1.903
n=6, fibonacci recursion: 0.000015700, fibonacci iteration: 0.000005400, ratio: 2.907
n=7, fibonacci recursion: 0.000013200, fibonacci iteration: 0.000003200, ratio: 4.125
n=8, fibonacci recursion: 0.000019500, fibonacci iteration: 0.000003300, ratio: 5.909
n=9, fibonacci recursion: 0.000030200, fibonacci iteration: 0.000003300, ratio: 9.152
n=10, fibonacci recursion: 0.000047900, fibonacci iteration: 0.000003500, ratio: 13.686
n=11, fibonacci recursion: 0.000133300, fibonacci iteration: 0.000003800, ratio: 35.079
n=12, fibonacci recursion: 0.000170000, fibonacci iteration: 0.000004000, ratio: 42.500
n=13, fibonacci recursion: 0.000203700, fibonacci iteration: 0.000003900, ratio: 52.231
n=14, fibonacci recursion: 0.000374800, fibonacci iteration: 0.000004200, ratio: 89.238
n=15, fibonacci recursion: 0.000739300, fibonacci iteration: 0.000004400, ratio: 168.023
n=16, fibonacci recursion: 0.000883400, fibonacci iteration: 0.000004700, ratio: 187.957
n=17, fibonacci recursion: 0.001492600, fibonacci iteration: 0.000004500, ratio: 331.689
n=18, fibonacci recursion: 0.003357500, fibonacci iteration: 0.000006300, ratio: 532.937
n=19, fibonacci recursion: 0.004149500, fibonacci iteration: 0.000004800, ratio: 864.479
n=20, fibonacci recursion: 0.005755300, fibonacci iteration: 0.000004600, ratio: 1251.152
n=21, fibonacci recursion: 0.010408700, fibonacci iteration: 0.000004900, ratio: 2124.224
n=22, fibonacci recursion: 0.015016000, fibonacci iteration: 0.000005000, ratio: 3003.200
n=23, fibonacci recursion: 0.024282400, fibonacci iteration: 0.000005100, ratio: 4761.255
n=24, fibonacci recursion: 0.042411100, fibonacci iteration: 0.000007500, ratio: 5654.814
n=25, fibonacci recursion: 0.067926200, fibonacci iteration: 0.000005400, ratio: 12578.925
n=26, fibonacci recursion: 0.102938000, fibonacci iteration: 0.000005600, ratio: 18381.785
n=27, fibonacci recursion: 0.163930500, fibonacci iteration: 0.000005600, ratio: 29273.303
n=28, fibonacci recursion: 0.267603100, fibonacci iteration: 0.000005600, ratio: 47786.267
n=29, fibonacci recursion: 0.455059300, fibonacci iteration: 0.000005600, ratio: 81260.586
n=30, fibonacci recursion: 0.708885700, fibonacci iteration: 0.000005900, ratio: 120150.115
n=31, fibonacci recursion: 1.143951900, fibonacci iteration: 0.000005700, ratio: 200693.325
n=32, fibonacci recursion: 1.863286800, fibonacci iteration: 0.000006000, ratio: 310547.789
n=33, fibonacci recursion: 3.010595400, fibonacci iteration: 0.000006100, ratio: 493540.180
n=34, fibonacci recursion: 4.952022600, fibonacci iteration: 0.000006100, ratio: 811806.975
n=35, fibonacci recursion: 8.006804200, fibonacci iteration: 0.000006800, ratio: 1177471.217
```

The recursive implementation looks horribly slower than the iteration-based one!

Fibonacci: recursion vs. iteration

- What's going on?
- In the recursive implementation of the Fibonacci sequence, each recursive call does actually performs TWO recursive calls, one on $F(n-1)$ and one on $F(n-2)$
- The computation for each one of these calls, as well as for successive calls, does overlap quite a lot

The tree shows what happens for $n = 5$, where `f(i)` stands for calling the `fibonacci(i)` function



- We can notice that the subtree f(2) appears 3 times and the subtree for the calculation of f(3) two times, meaning the same computation is done repeatedly and redundantly
- If we would show the tree for f(6), f(4) would be called two times, f(3) three times and so on.
- The plain way recursion has been implemented doesn't remember previously calculated values!
- Can we overcome this issue?

Fibonacci: recursion with memoization

- We can implement a "memory cache" for our recursive calls, such that the same values are not computed over and over, but rather when a computation is performed, it is stored and reused from the cached memory if required later on
- The idea of optimizing the running performance of an algorithm by caching intermediate results and reuse them when / if the same inputs are presented, is called **memoization**

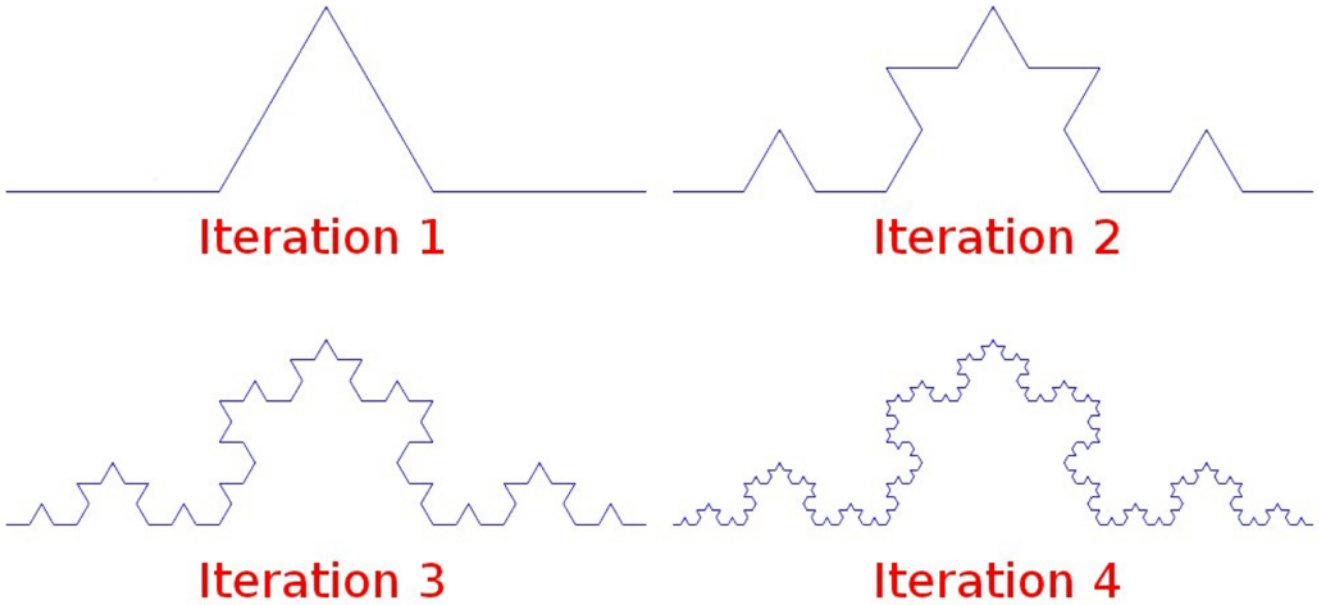
✓ Use a *dictionary* global variable to store in memory previously computed values of $f(n)$, and pass it as function argument

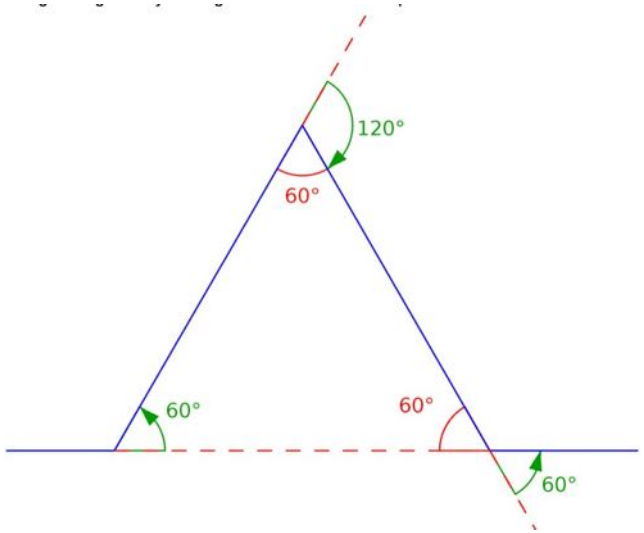
```
computed = {0:0, 1:1}
def fibonacci_memoization(n, computed):
    if n not in computed:
        computed[n] = fibonacci_memoization(n-1, computed) +
                      fibonacci_memoization(n-2, computed)
    return computed[n]
```

```
computed = {0:0, 1:1}
def fibonacci_memoization_g(n):
    global computed
    if n not in computed:
        computed[n] = fibonacci_memorization_g(n-1) +
                      fibonacci_memorization_g(n-2)
    return computed[n]
```

In this version, the global variable `computed` is not passed as an argument, but it is explicitly referred to as a global variable

Another example of recursive objects: fractals, Koch curve





https://en.wikipedia.org/wiki/Koch_snowflake

Improved exponentiation

- Problem: compute $f(x,n) = x^n$ recursively, where n is an integer

  - ❖ In the previous solution, the computation went through n levels of recursions
  - ❖ n might be large, it might take time and memory, we might need to adapt OS parameters …
  - ➢ Can we design a better algorithm?
  - ✓ Base case: $f(x,0) = 1$

- Let's notice the following properties:
  - If n is even: $x^n = \left(x^{n/2}\right)^2$ → Only $n/2$ recursion levels
  - If n is odd: $x^n = x \cdot \left(x^{(n-1)/2}\right)^2$ → Only $(n-1)/2$ recursion levels

- Recursive case: $f(x,n) = \begin{cases} y^2,\ y = f(x, {}^{n}/_{2}), & \text{if } n \text{ is even} \\ x \cdot y^2,\ y = f\left(x, {}^{(n-1)}/_{2}\right), & \text{if } n \text{ is odd} \end{cases}$

Another example: improved exponentiation

- ✓ Base case: $f(x, 0) = 1$
- ✓ Recursive case: $f(x, n) = \begin{cases} y^2, \ y = f(x, {}^{n}/_{2}), & \text{if } n \text{ is even} \\ x \cdot y^2, \ y = f\left(x, {}^{(n-1)}/_{2}\right), & \text{if } n \text{ is odd} \end{cases}$

```
def exponent_divide (x, n):
    global count  #count num of recursions
    count += 1
    # Base Cases
    if (n == 0):
        return 1
    if (x == 0):
        return 0

    # If n is even
    if (n % 2 == 0):
        y = exponent_divide(x, n/2)
        return y * y

    # If n is odd
    else:
        y = exponent_divide(x, (n-1)/2)
        return  x * (y * y)
```

```
import math
n = 1000
x = 5
count = 0
print(exponent_divide (x,n), count/2, math.log(n))
```

Number of recursions goes as $\log(n)$

- ✓ This is much faster than the original implementation!
- ✓ **Moral:** to get good computational performance, either use memoization whenever possible, or identify a recursive relation that implies a relatively low number of recursive calls

A problem that needs recursion

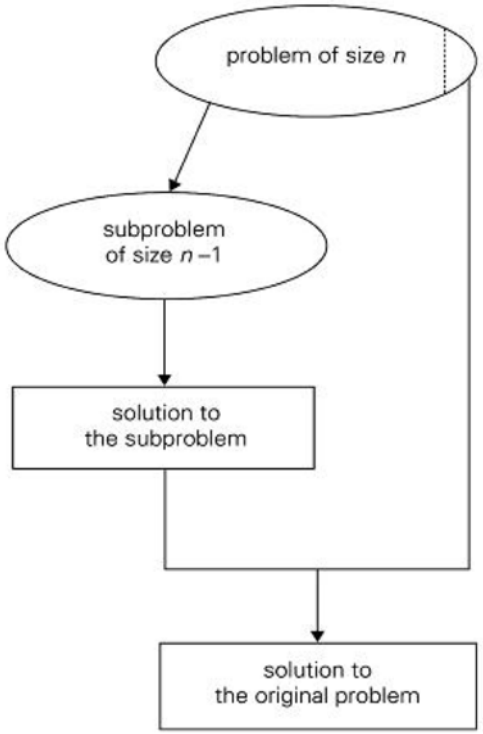
- Operating on a list of lists of lists … and we don't know the depth of each sublist …

Sum of [1,2, [1,2,3], [3,4, [1,2,[3,2]]]] ?

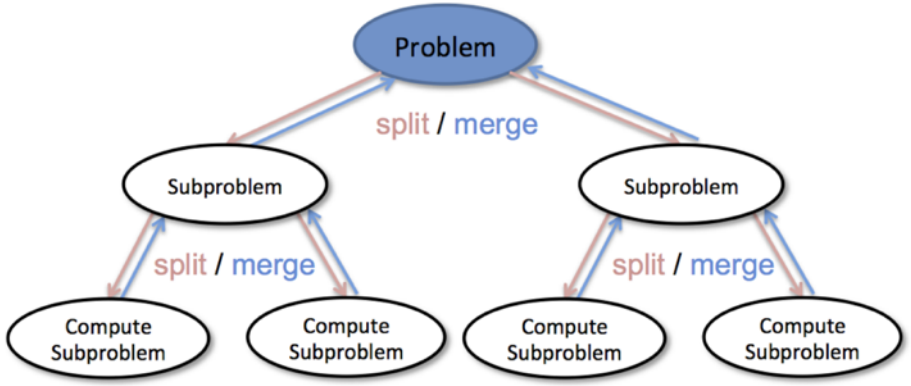
```
def nested_sum(l):
    total = 0
    for element in l:
        if (type(element) != type([])):
            total += element
        else:
            total += nested_sum(element)
    return total
print( "Sum is:", nested_sum([ 1,2, [1,2,3], [3,4, [1,2,[3,2]]] ]))
```

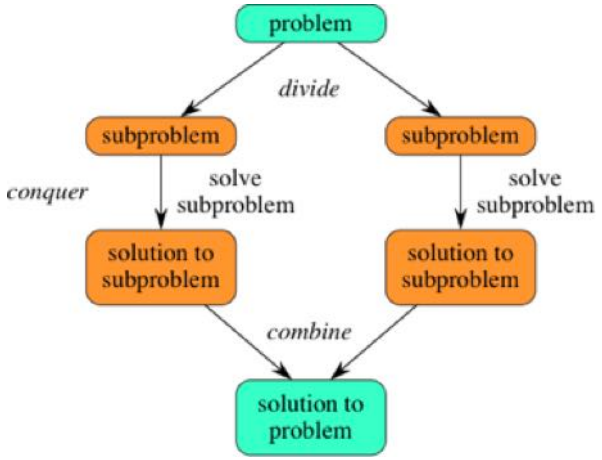
Decrease-and-conquer, Divide-and-conquer

- What have been doing so far with the recursive solution models?
- Take a problem of size n, recursively solved it for smaller and smaller sizes, until a minimal input size has been reached, and then combine all results to produce the desired result from the original input size n
- This a general approach to problem-solving called **decrease-and-conquer**



➢ **Divide-and-conquer**: A more general version of this way of proceeding consists in recursively dividing the original problem in easier sub-problems, until a minimal size of reached, or the solution to a sub-problem is easily found, and then combine the results from the individual sub-problems to get the solution to the (harder) original problem





Divide and conquer: reduce a complex problem to a set of easier ones

(**Plutarch**: *Parallel lives*, *Sertorius*, 120 A.D.)

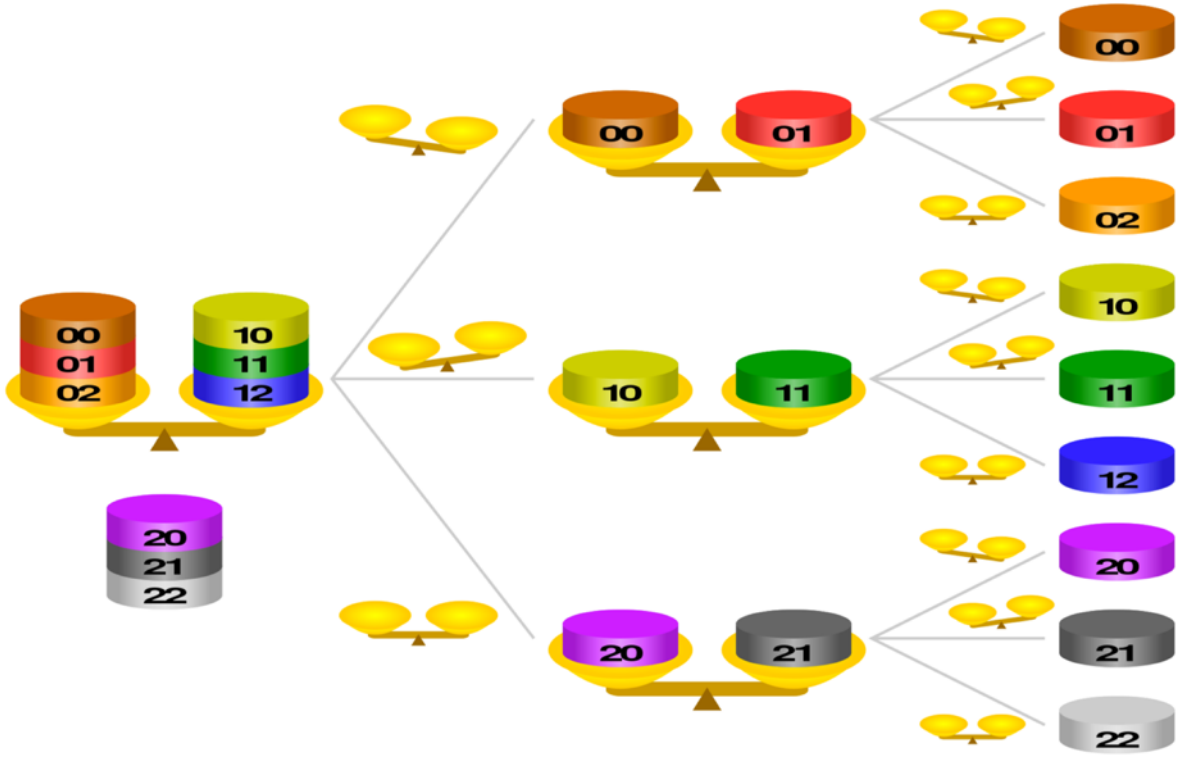
Sertorius called a general assembly and introduced before it two horses, one utterly weak and already quite old, the other large-sized and strong, with a tail that was astonishing for the thickness and beauty of its hair. By the side of the feeble horse stood a man who was tall and robust, and by the side of the powerful horse another man, small and of a contemptible appearance.

At a signal given them, the strong man seized the tail of his horse with both hands and tried to pull it towards him with all his might, as though he would tear it off; but the weak man began to pluck out the hairs in the tail of the strong horse one by one. The strong man gave himself no end of trouble to no purpose, made the spectators laugh a good deal, and then gave up his attempt; but the weak man, in a trice and with no trouble, stripped his horse's tail of its hair.

Then Sertorius rose up and said: "Ye see, men of my allies, that perseverance is more efficacious than violence, and that many things which cannot be mastered when they stand together yield when one masters them little by little. For irresistible is the force of continuity, by virtue of which advancing, time subdues and captures every power; and time is a kindly ally for all who act as diligent attendants upon opportunity, but a most bitter enemy for all who urge matters on unseasonably." By contriving from time to time such exhortations for the Barbarians, Sertorius taught them to watch for their opportunities.

Divide and conquer: how to look for the fake coin?

You have 9 coins among which one is fake and it is lighter than the others. You have also a balance scale and you can compare the weight of any two sets of coins. How can you find the fake coin (with 3 measurements only)?



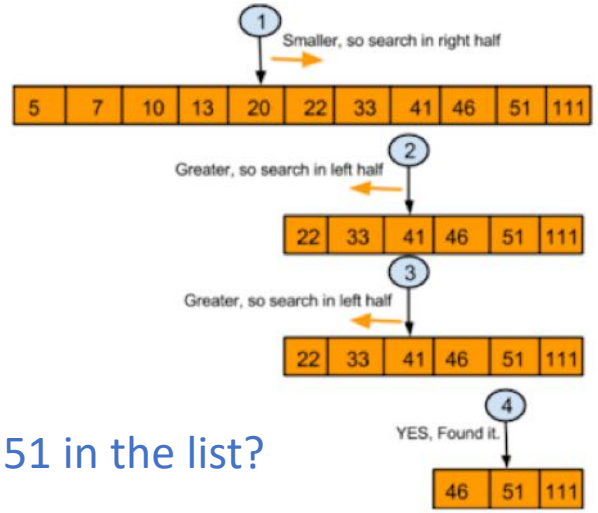
- Why is this an example of divide-and-conquer?
- Why would you use recursion here?

Divide-and-conquer: binary search

- Given a sorted list of values, find whether a given target value exists or not in the list

➢ A **binary search** divides the set of values into halves, and continues to narrow down the field of search until the value is found (or it returns not found). This is the way to proceed to solve the previous problem of the fake coin!

- `binary_search()` takes a list of values and the variables `start`, `end`, and a `key` value as arguments. The function searches for the key in the index range `[start: end – 1]`
- The base case consists of testing whether `start` is less than `end`. If not, -1 is returned, meaning `key` is not in `values`
- `mid`, the midpoint where to divide, is calculated as the floor of the average of sub-problem's `start` and `end`.
- Recursive case: if the element at index `mid` is less than `key`, binary search is called with `start=mid + 1` and if it is more than `key`, it is called with `end=mid`. Otherwise, `mid` is returned as the index of the found element.



Is 51 in the list?

```
def binary_search(values, start_pos, end_pos, key):
    '''Search key in list values from index (start_pos) to index (end_pos - 1).
        Returns the index of the key in the list, or -1 if not found.'''
    if not start_pos < end_pos:
        return -1
    mid = (start_pos + end_pos) // 2
    if values[mid] < key:
        return binary_search(values, mid + 1, end_pos, key)
    elif values[mid] > key:
        return binary_search(values, start_pos, mid, key)
    else:
        return mid
```