Extended Introduction to Computer Science CS1001.py

Lecture 11:
Recursion and Recursive Functions (cont.)

Instructors: Daniel Deutch, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Yael Baran, Amir Gilad

School of Computer Science
Tel-Aviv University
Spring Semester 2016
http://tau-cs1001-py.wikidot.com

Lecture 10: Reminder

Recursion, and recursive functions

- Basic examples and definition of recursion
  - Fibonacci
  - factorial
- Binary search - revisited
- Sorting
  - QuickSort

Lecture 11 - Plan

Recursion, and recursive functions (cont.)

- Sorting (cont.)
  - MergeSort
- Towers of Hanoi
- Memoization

Another Manifestation of Recursion (with a twist)

(Cover of Ummagumma, a double album by Pink Floyd, released in 1969.
Taken from Wikipedia. Thanks to Yair Sela for the suggestion.)

Recursion (definition, reminder)

A function f(·), whose definition contains a call to f(·) itself, is called recursive.

A simple example is the factorial function, $n! = 1 \cdot 2 \cdot \ldots \cdot n$.
It can be coded in Python, using recursion, as follows:

```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

A second simple example are the Fibonacci numbers, defined by $F_1 = 1$, $F_2 = 1$, and for $n > 2$; $F_n = F_{n-2} + F_{n-1}$.

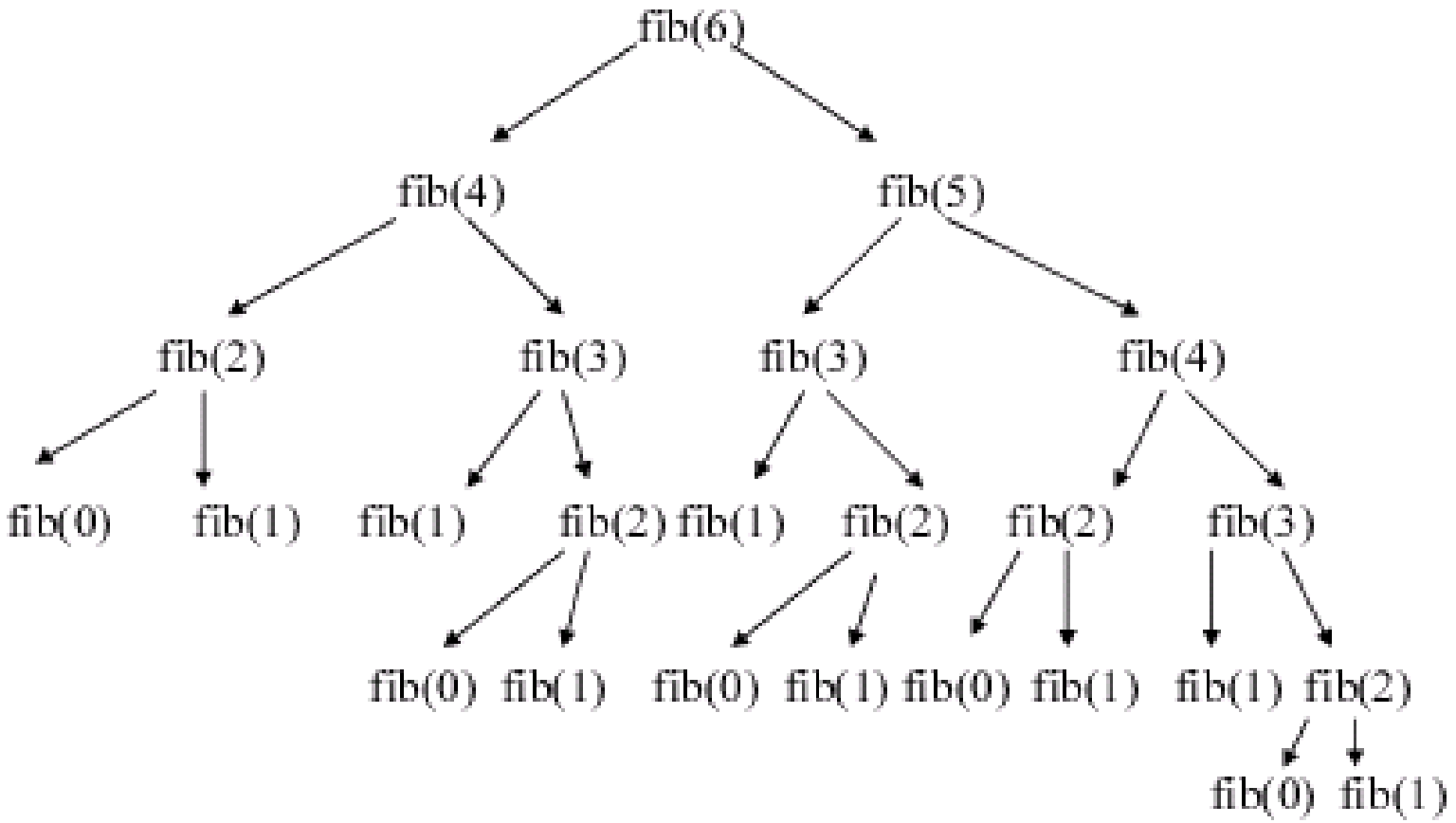
Recursion and Convergence (reminder)

Two “design principles" to the correct design of recursive functions:

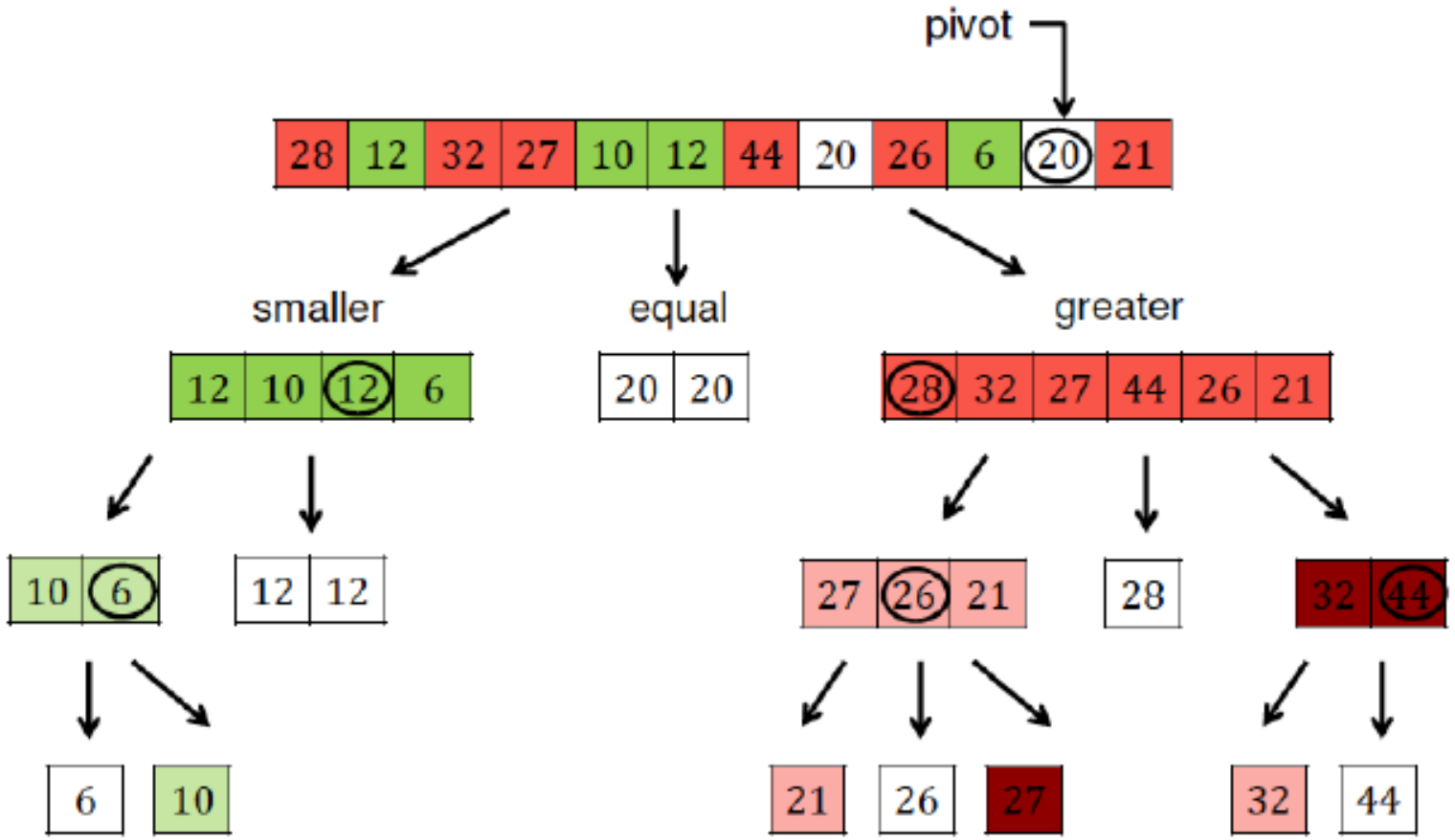
1. Have a base case (one or more), which is the halting condition (no deeper recursion). In the factorial example, the base case was the condition n==0. In the Fibonacci example, it was n<=1.
2. Make sure that all executions, or “runs", of the recursion will actually lead to one of these base cases.

Recursion Trees (reminder)

For example, here is the recursion tree for fibonacci, for n=6:



Quicksort: A Graphical Depiction (reminder)



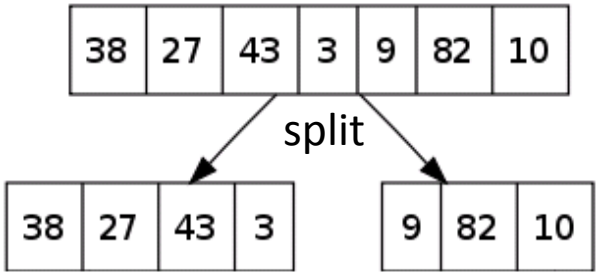
Quicksort: Python code (reminder)

```
import random     # a package for (pseudo) random generation
def quicksort(lst):
    if len(lst)<=1:      # empty lists or length one lists
        return lst
    else:
        pivot = random.choice(lst)
                # select a random element from the list
        smaller = [elem for elem in lst if elem < pivot]
        equal = [elem for elem in lst if elem == pivot]
        greater = [elem for elem in lst if elem > pivot]
                # ain't these selections neat?
        return quicksort(smaller) + equal + quicksort(greater)
                # two recursive calls
```

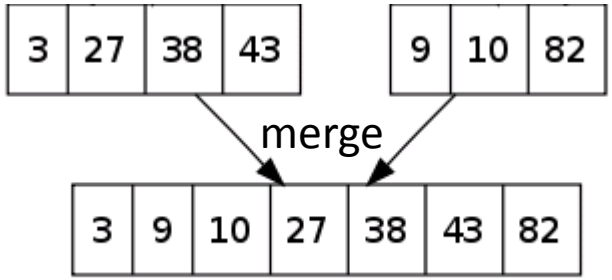
Merge Sort

Mergesort is a recursive, deterministic, sorting algorithm, i.e. it also follows a divide and conquer approach.

An input list (unsorted) is split to two -- elements with indices from 0 up to the middle, and those from the middle up to the end of the list.



If we could sort these 2 halves, we would be done by merging them.



Well, does anybody know a good sorting algorithm?

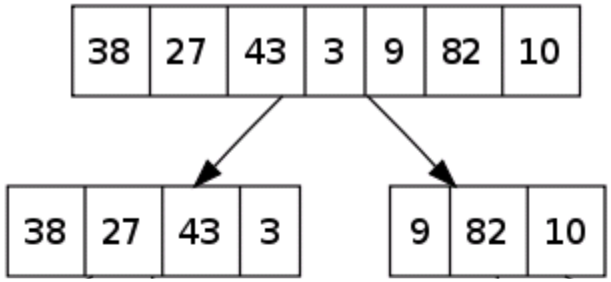
Merge Sort

An input list (unsorted) is split to two -- elements with indices from 0 up to the middle, and those from the middle up to the end of the list.

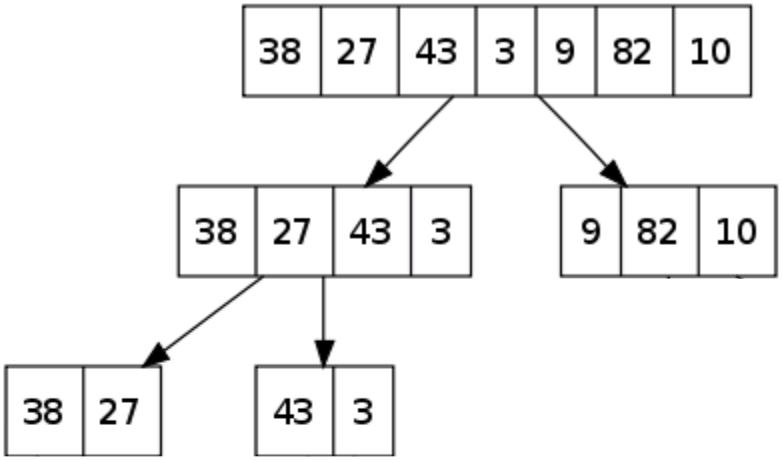
Each half is sorted recursively.

The two sorted halves are then merged to one, sorted list.

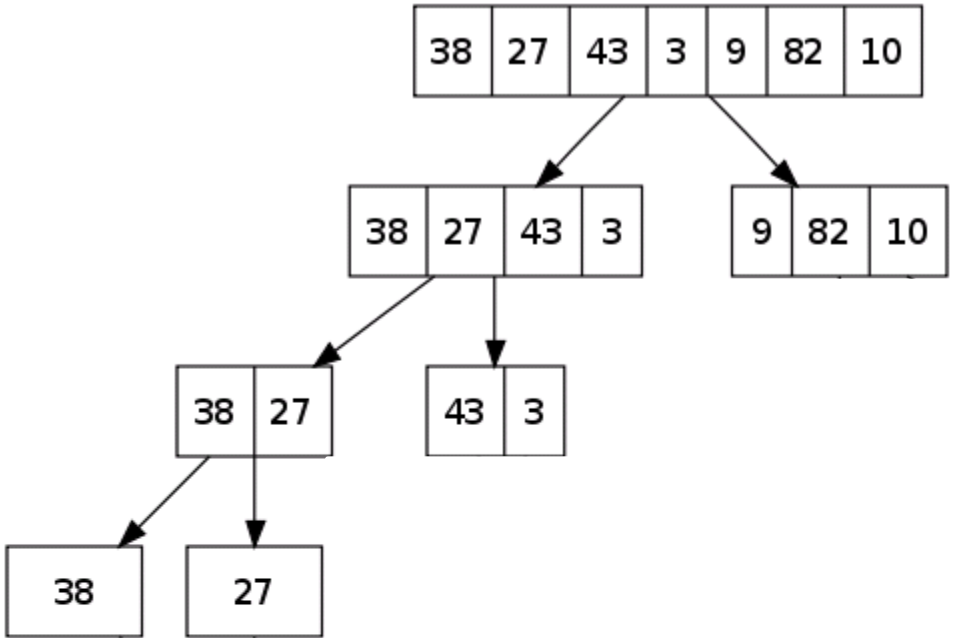
Merge Sort – example



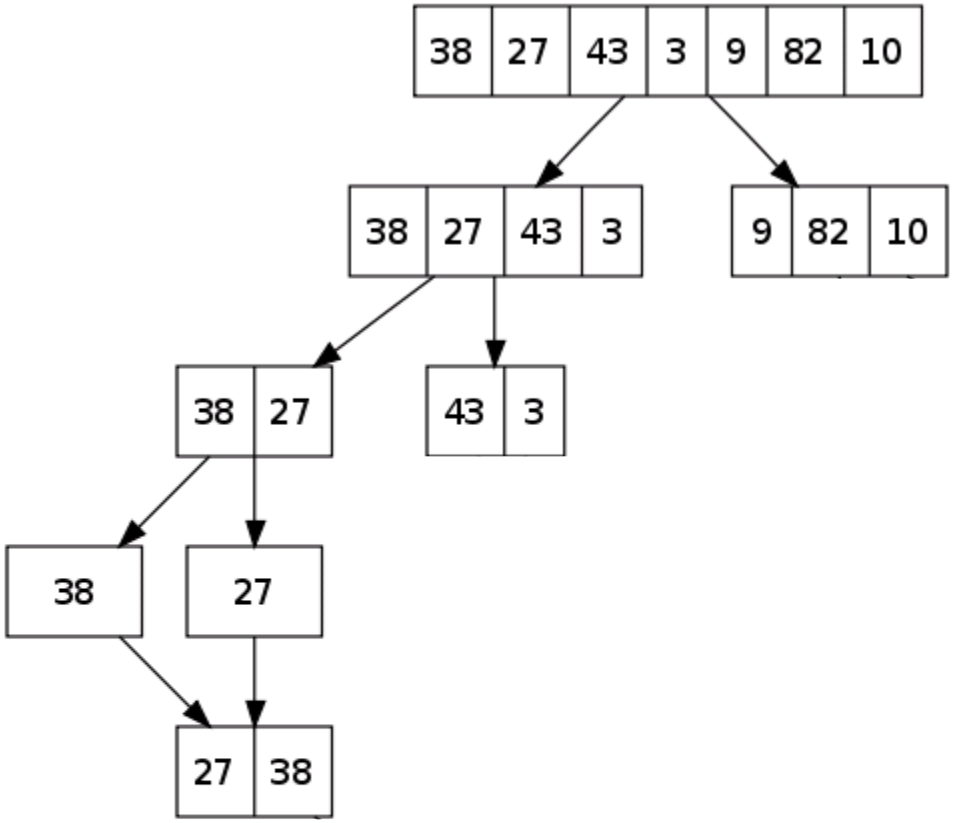
Merge Sort – example



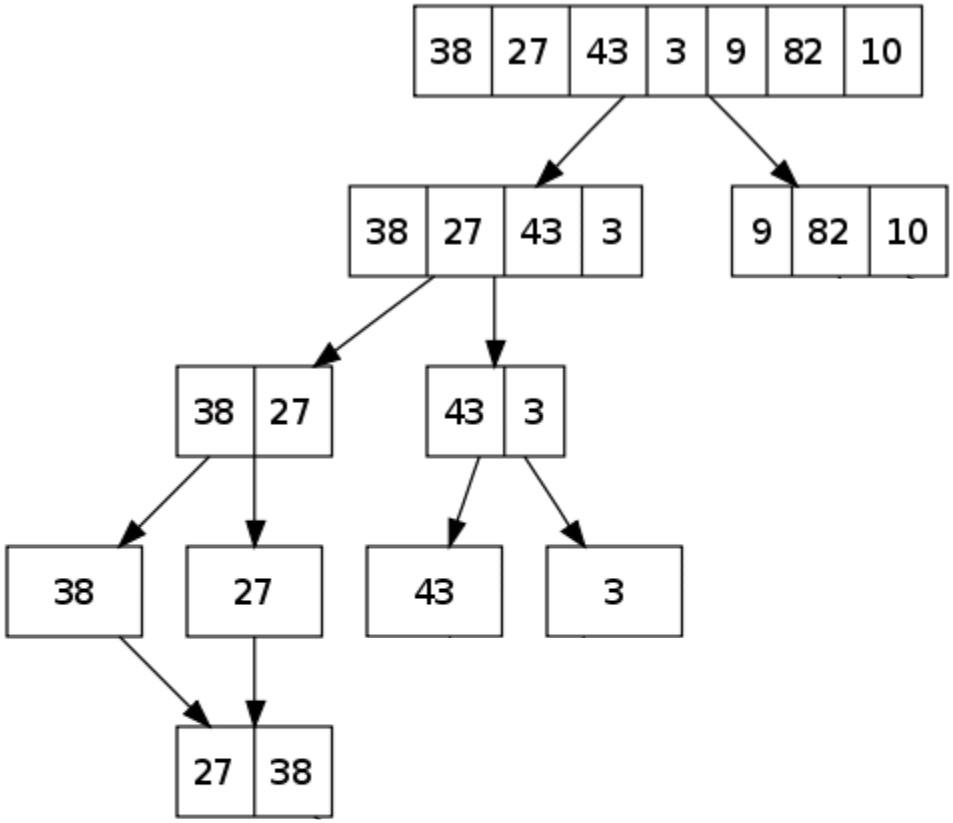
Merge Sort – example



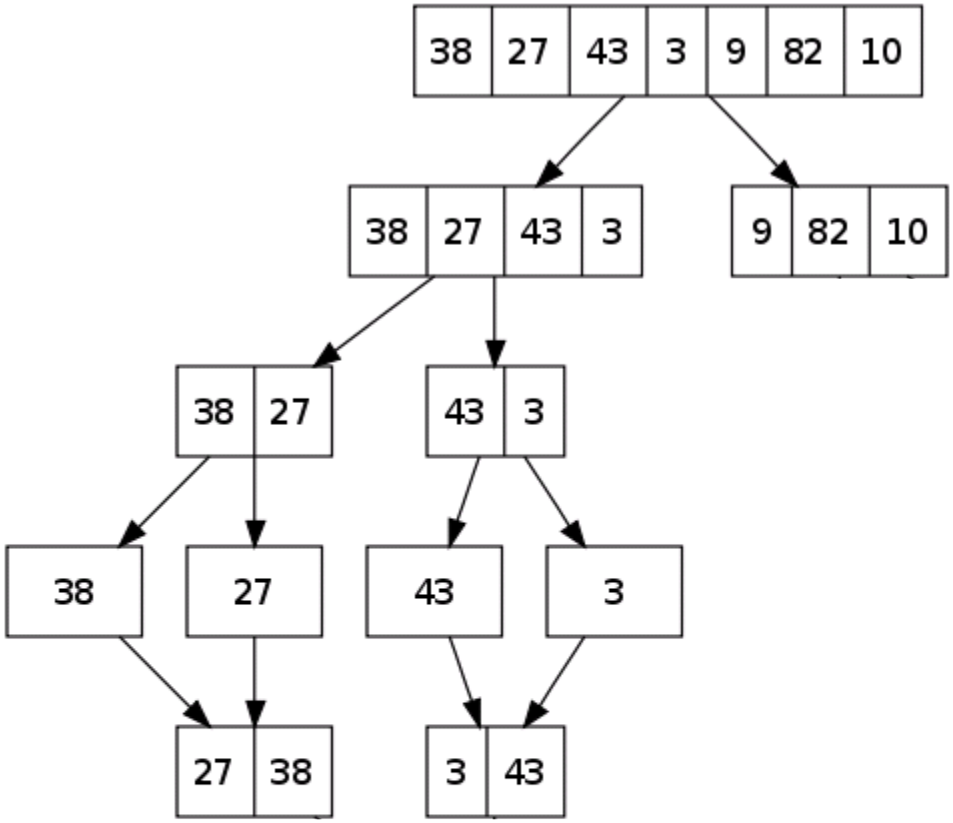
Merge Sort – example



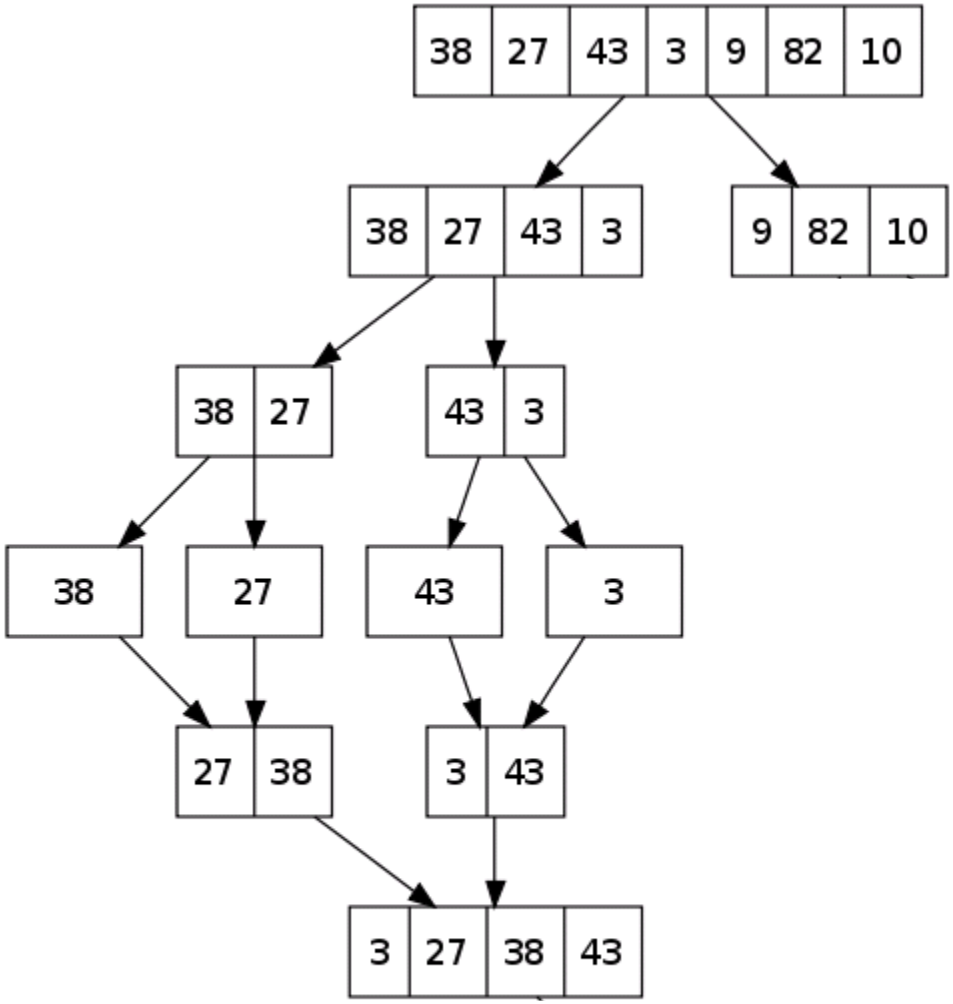
Merge Sort – example



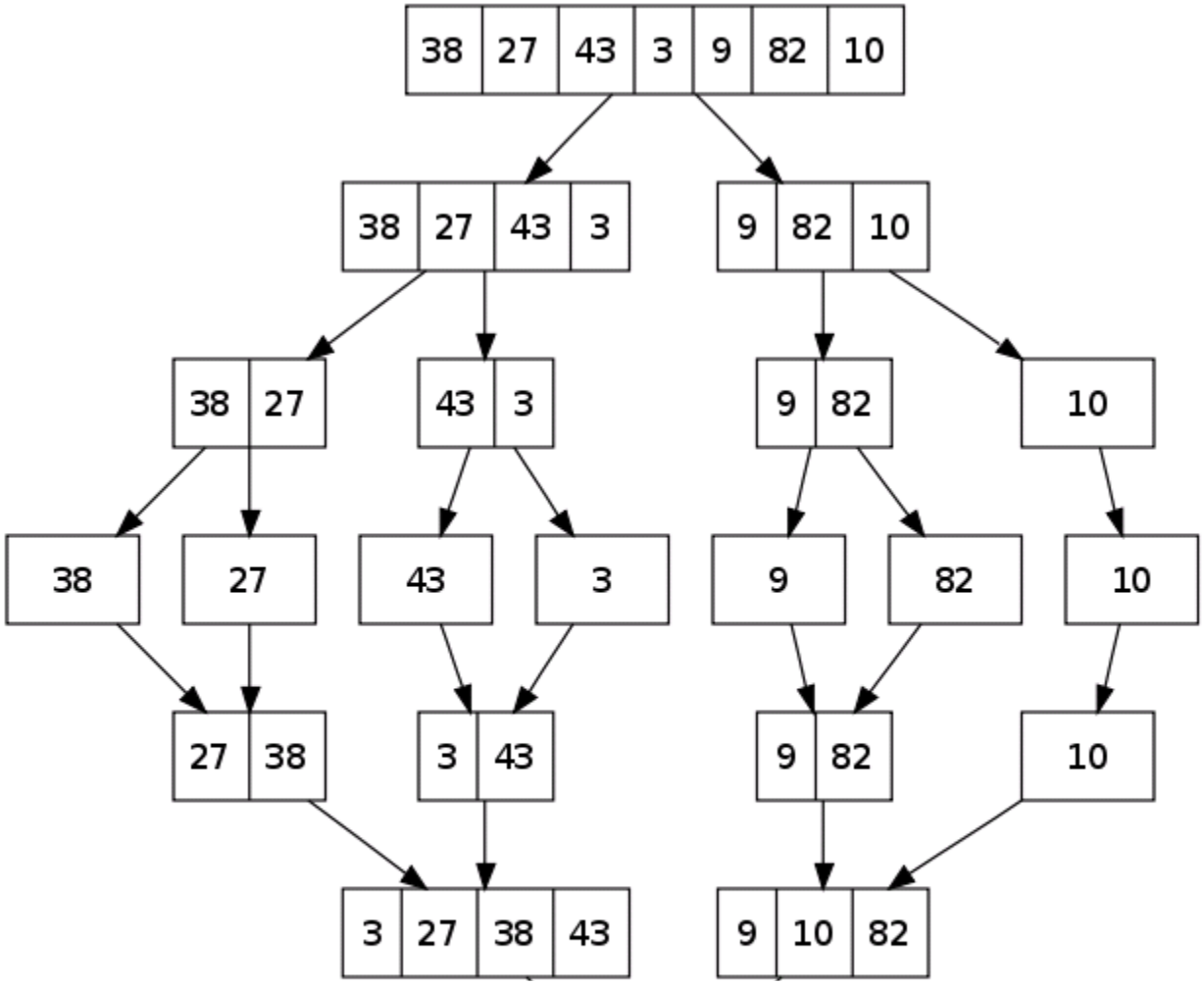
Merge Sort – example



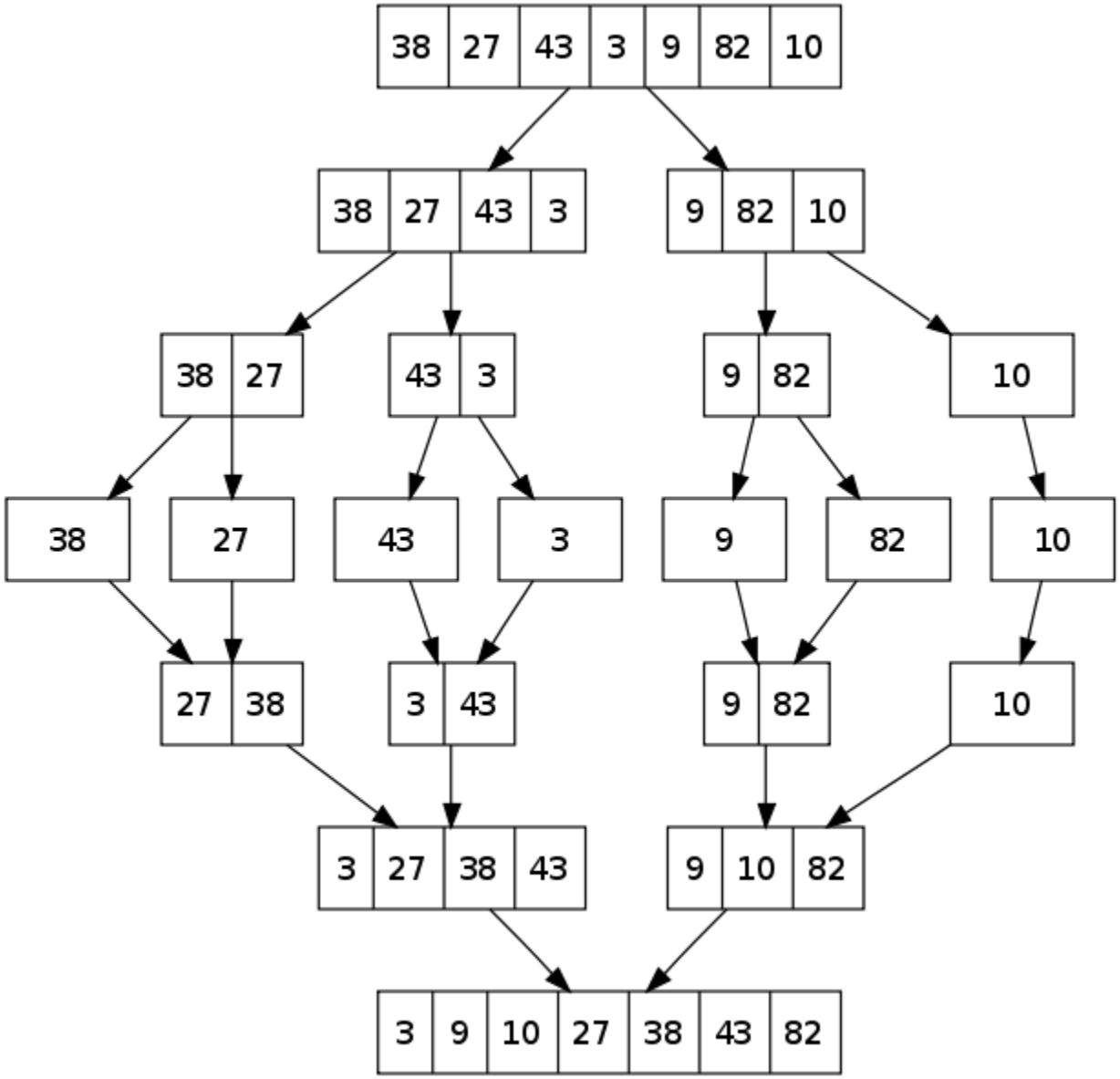
Merge Sort – example



Merge Sort – example



Merge Sort – example



Merge Sort, cont.

Suppose the input is the following list of length 13

[28, 12, 32, 27, 10, 12, 44, 20, 26, 6, 20, 21, 0].

We split the list in half, to

[28, 12, 32, 27, 10, 12] and [44, 20, 26, 6, 20, 21, 0].

And recursively sort the two smaller lists, resulting in

[10, 12, 12, 27, 28, 32] and [0, 6, 20, 20, 21, 26, 44].

We then merge the two lists, getting the final, sorted list

[0, 6, 10, 12, 12, 20, 20, 21, 26, 27, 28, 32, 44].

The key to the efficiency of merge sort is the fact that **as we saw**, merging two lists is done in time O(n+m), where n is the length of first list and m is the length of second list.

Merge (reminder)

```
def merge(A, B):
    ''' Merge list A of size n and list B of size m
        A and B must be sorted! '''
    n = len(A)
    m = len(B)
    C = [0 for i in range(n + m)]

    a=0; b=0; c=0
    while  a<n  and  b<m: #more element in both A and B
           if A[a] < B[b]:
               C[c] = A[a]
               a += 1
           else:
               C[c] = B[b]
               b += 1
           c += 1

    C[c:] = A[a:] + B[b:] #append remaining elements

    return C
```


Time complexity: O(n+m)

Merge Sort: Python Code

```
def mergesort (lst):
    """ recursive mergesort """
    n = len (lst)
    if n <= 1:
        _____________________________
    else:
        return merge(_____________________________ ,\
                     _____________________________ )
                # two recursive calls
```

Merge Sort: Python Code

```
def mergesort (lst):
    """ recursive mergesort """
    n = len (lst)
    if n <= 1:
      return lst
    else:
      return merge(mergesort(lst[0:n//2]) ,\
                   mergesort(lst[n//2:n]))
              # two recursive calls
```

Merge Sort: Complexity Analysis

Given a list with n elements, mergesort makes 2 recursive calls. One to a list with $\lfloor n/2 \rfloor$ elements, the other to a list with $\lceil n/2 \rceil$ elements.

The two returned lists are subsequently merged.

On board: Recursion tree and time analysis.

Question:

- Is there a difference between worst-case and best-case for mergesort?

Merge Sort: Complexity Analysis

The runtime of mergesort on lists with n elements satisfies the recurrence relation $T(n) = c \cdot n + 2 \cdot T(n/2)$, where c is a constant.

The solution to this relation is $T(n) = O(n \cdot \log n)$.

Recall that in the rec_slice_binary_search function, slicing resulted in O(n) overhead to the time complexity, which is disastrous for searching.

Here, however, we deal with sorting, and an O(n) overhead is asymptotically negligible.

A Three Way Race

Three sorting algorithms left Haifa at 8am, heading south. Which one will get to TAU first?
We will run them on random lists of lengths 200, 400, 800.

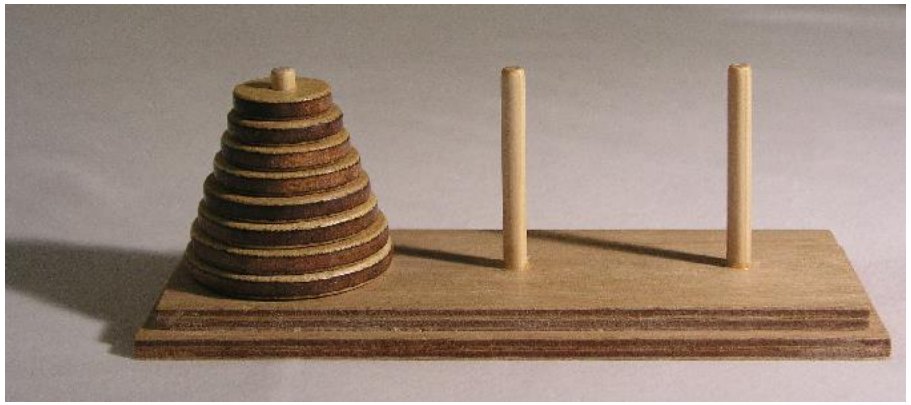
```
>>> from quicksort import *   #the file quicksort.py
>>> from mergesort import *   #the file mergesort.py

3 way race
quicksort
n= 200 0.17896895999999998
n= 400 0.38452376
n= 800 0.87327308
mergesort
n= 200 0.24297283999999997
n= 400 0.49345808000000013
n= 800 1.0856526
sorted # Python 's sort
n= 200 0.007834879999999877
n= 400 0.02002811999999965
n= 800 0.04940151999999998   # I think we have a winner!
```

The results, ahhhm, speak for themselves.

And Now For Something Completely Different: Towers of Hanoi

Towers of Hanoi is a well known mathematical puzzle, and no class on recursion, including this one (a recursive claim in itself :-), is complete without discussing it.



(figure from Wikipedia)

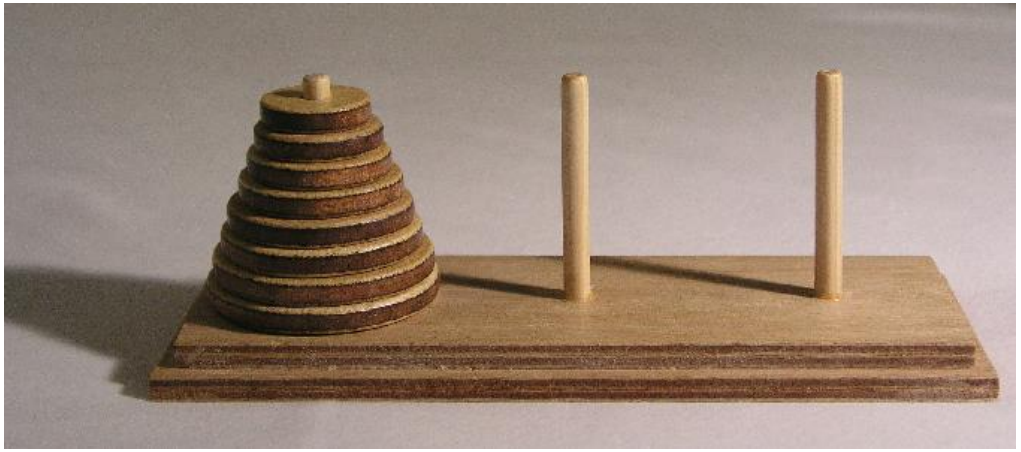
Towers of Hanoi - origin

The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle will be completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it.

(text from Wikipedia)

Towers of Hanoi

There are three rods, named A, B, C, and n disks of different sizes which can be placed onto any rod. The puzzle starts with all n disks in a stack in ascending order of size on one rod, say A, so that the smallest is at the top (see figure).

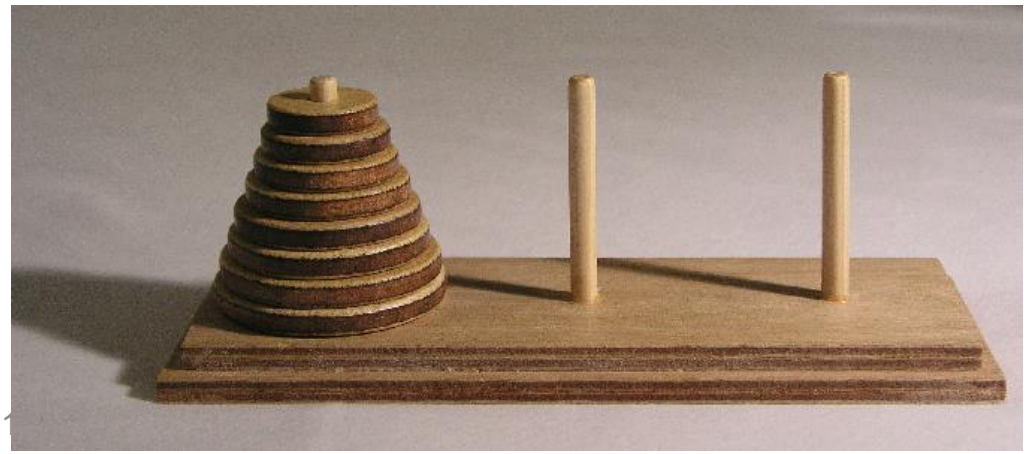


(figure from Wikipedia)

Towers of Hanoi: Rules of Game

The objective of the puzzle is to move the entire stack of all n disks to another rod, say C, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.



(figure and some text from Wikipedia)

Towers of Hanoi: recursive view

In order to think about a recursive solution, we should first have a recursive definition of a Hanoi tower:

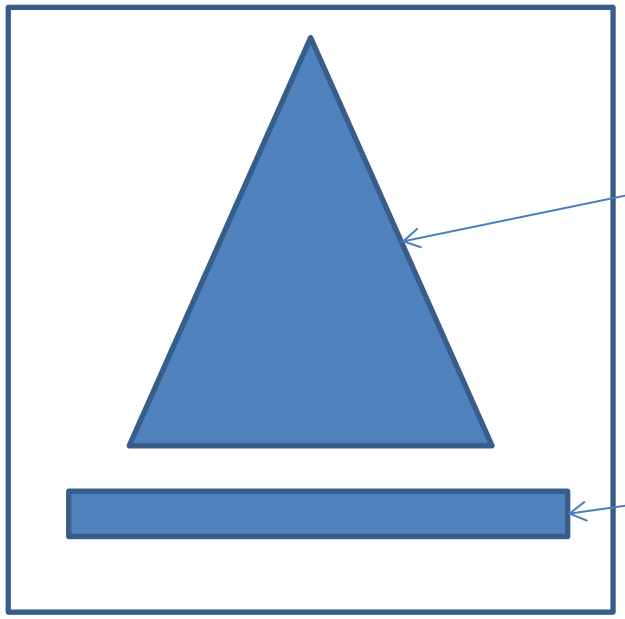
it is either empty,

This is the base case

or it is a tower on top of a larger disk (larger than all the disks in the tower on top).

Another possibility is to let the base case be a tower of one disk

Schematically:



A tower of n-1 disks

A larger disk

Towers of Hanoi: The algorithm

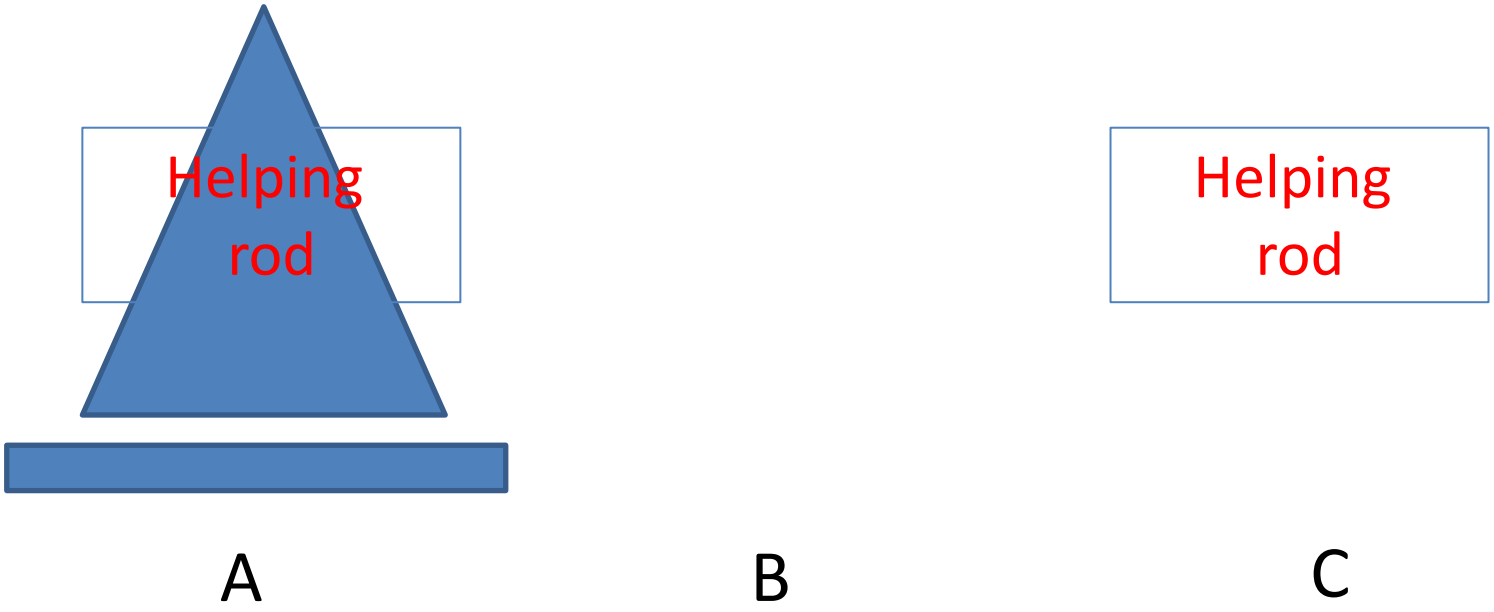
We can now describe a recursive algorithm to move a stack of n disks from rod A to rod C using rod B as a helping rod.

In the base case , when n=0, there is nothing to do.

The non- base case (n>0) will be shown in the next slide.

[If we chose n=1 as the base case, then the base case would be to move the single rod from A to C]

Towers of Hanoi: recursive algorithm

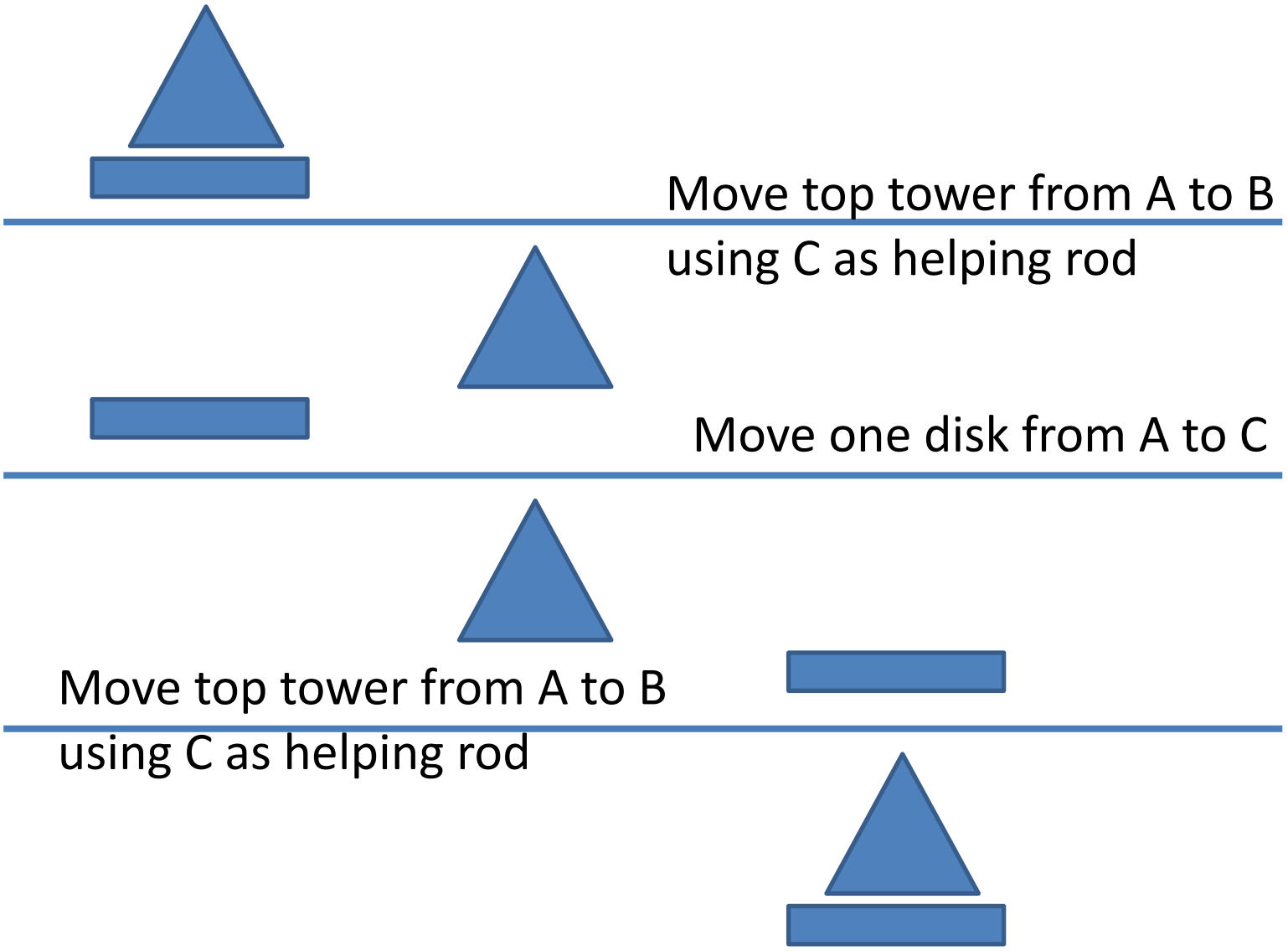


Move top tower from A to B using C as helping rod

Move one disk from A to C

Move top tower from B to C using A as helping rod

Towers of Hanoi: another picture



Towers of Hanoi: Recursive Solution

To move n disks from rod A to rod C, using B as a “helping rod":
If n = 0, there is nothing to do.
Otherwise (namely n > 0):

1) Move n - 1 disks from rod A to rod B, using C as a “helping rod".
2) Move the single disc n directly from rod A to rod C.
3) Move n - 1 disks from rod B to rod C, using A as a “helping rod".

Correctness: (no rules are violated)

- During the entire stage (1), disk n stays put on rod A. As it was the biggest of all n disks, no rule will be violated if some of the n - 1 disks are placed on top of it during the recursion in (1).
- In step (2), all n - 1 smaller disks are on rod B, so moving disc n directly from rod A to rod C is legal.
- The argument for step (3) is identical to the argument for step (1).

Towers of Hanoi: Number of Moves

Let us denote by H(n) the number of moves required to solve an n disc instance of the puzzle.

In the recursive solution outlined above, to solve an n discs instance we solve two instances of n - 1 discs, plus one actual move. This gives us the recursive relation

$$H(0) = 0$$

$$\text{For } n > 0,\ H(n) = 2 \cdot H(n - 1) + 1$$

whose solution is $H(n) = 2^n - 1$. (You should be able to verify the last equality, using induction.)

Time Complexity analysis

We claimed that the number of moves required to solve an instance with n disks is $H(n) = 2^n - 1$. Our program generates such a list of disk moves. It runs in $O(H(n))$ time = $O(2^n)$

The recursion depth here is “just" $O(n)$. But the size of the recursion tree is $O(2^n)$, which is exponential in n.

Optimality of Number of Moves

Hey, wait a minute. $H(n) = 2^n - 1$ is the number of moves in the solution presented above. Can't we find a more efficient solution?

This is very good thinking in general. But in this case, one can show (not even that hard) that $H(n) = 2^n - 1$ moves are required from any solution strategy. (Of course, more inefficient strategies do exist).

Towers of Hanoi: Python Code

We write a function of four arguments, HanoiTowers(start,via,target,n).
The first three arguments are the three rods, which have distinct names.
The last argument, n, is the number of discs.
The function returns an ordered list of discs moves.

```
def HanoiTowers (start ,via , target ,n):
    """ computes a list of discs steps to move a stack
     of n discs from rod " start " to rod " target " employing
     intermediate rod " via " """
     if n ==0:
          return [ ]
     else:
          return HanoiTowers (start , target ,via ,n -1) \
            + [str.format (" disk {} from {} to {}", n, start , target )] \
            + HanoiTowers (via ,start , target ,n -1)
```

Explaining the Mysterious str.format

str.format is a function that generates a string from “template" and from ordered arguments.

```
>>> str.format( "{} time {} equal {}", 5,6,30)
'5 times 6 equal 30'
>>> x=2; y=3; z=7
>>> str.format( "{} time {} equal {}", x,y,z)
'2 times 3 equal 7'        # garbage in, garbage out
>>> str.format("{} time {} equal {}","once upon a","all animals were","but some were more equal“ )
'once upon a time all animals were equal but some were more equal‘
```

There are more advanced forms of str.format, but we will probably not use them.

Towers of Hanoi: Running the Code

```
>>> Han = HanoiTowers ("A","B","C" ,3)
   # Han now is a list whose elements are the moves ( strings )
>>> for move in Han:
      print (move)
disk 1 from A to C
disk 2 from A to B
disk 1 from C to B
disk 3 from A to C
disk 1 from B to A
disk 2 from B to C
disk 1 from A to C
```

It is not a bad idea to verify that this does work ("trust, but check").
For small values of n, we could do this on board or on paper.

Towers of Hanoi: An Interesting Prefix Property (for reference only)

It is can be seen (not immediate, but not too hard) that to

move n + 2 disks from rod A to rod C, using B,

we first have to

move n disks from rod A to rod C, using B

(and then what?).

So the latter moves are a prefix of the former ones.

Memoization

Pitfalls of Using Recursion

Every modern programming language, including, of course, Python, supports recursion as one of the built-in control mechanism. However, recursion is not the only control mechanism in Python, and surely is not the one employed most often.

Furthermore, as we will now see, cases where "naive recursion" is highly convenient for writing code may lead to highly inefficient run times. For this reason, we will also introduce techniques to get rid of recursion. We note, however, that in some cases, eliminating recursion altogether requires very crude means.

Computing Fibonacci Numbers

We coded Fibonacci numbers, using recursion, as following:

```
def fibonacci(n):
    """ plain Fibonacci, using recursion """
    if n<2:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

But surely nothing could go wrong with such simple and elegant code... To investigate this, let us explore the running time:

Computing Fibonacci Numbers

But surely nothing could go wrong with such simple and elegant code... To investigate this, let us explore the running time:

```
>>> fibonacci(30)
1346269
>>> elapsed("fibonacci(30)")
0.31555
>>> elapsed("fibonacci(35)")
3.4169379999999996
>>> elapsed("fibonacci(40)")
38.288004
>>> elapsed("fibonacci(45)")
432.662887  # over 7 minutes !!
```

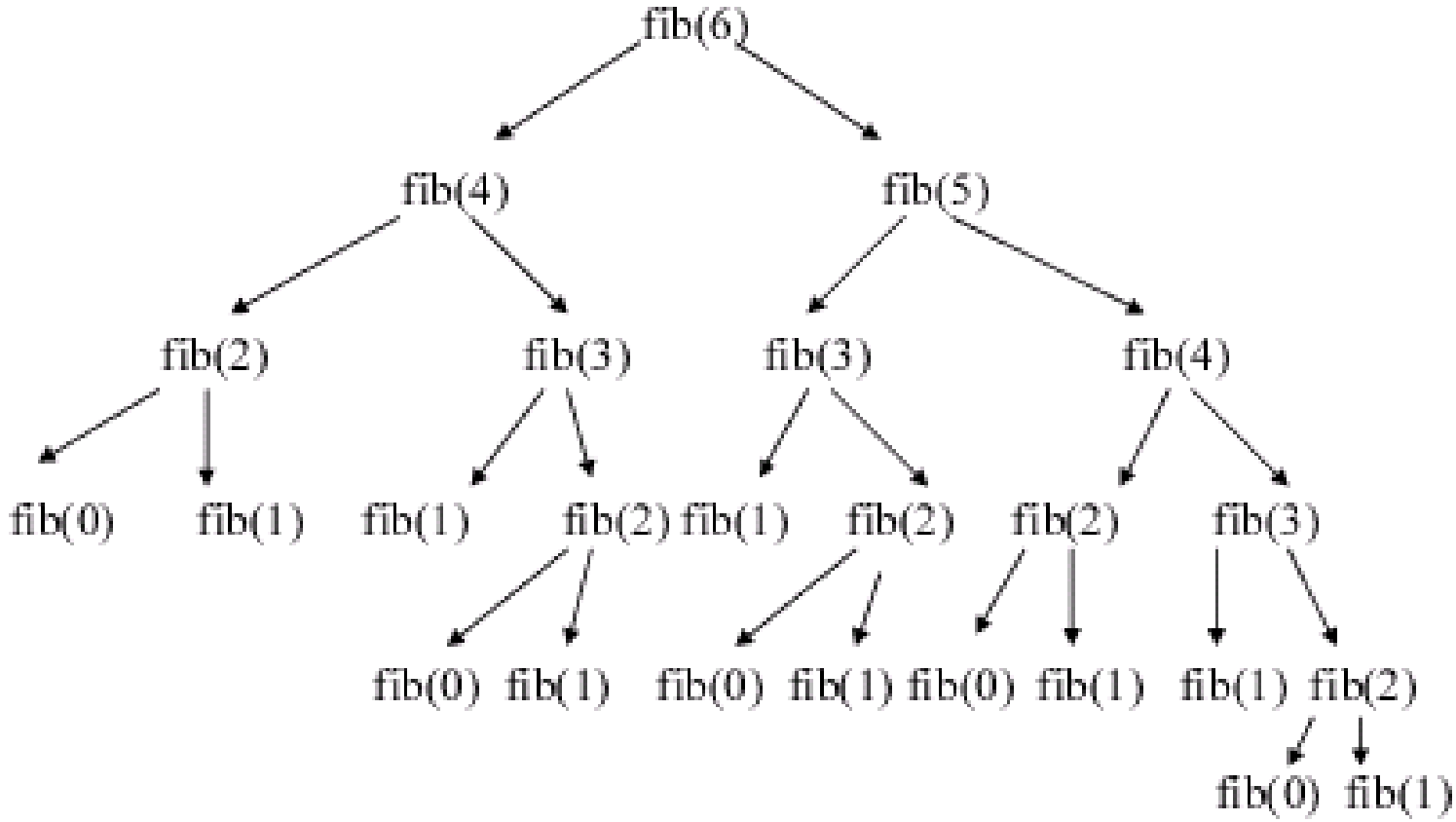
Inefficiency of Computing Fibonacci Numbers

What is causing this exponential growth in running time?

```
def fibonacci(n):
    """ plain Fibonacci, using recursion """
    if n<2:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Going over the computation mentally (or inserting print commands to track execution "physically"), we observe that `fibonacci(1)`, `fibonacci(2)`, `fibonacci(3)`, etc. are computed over and over. This is highly wasteful and causes a huge overhead.

Inefficiency of Computing Fibonacci Numbers



Count: Measuring the Inefficiency

We can easily modify the code, so it also counts the number of function invocations, using a global variable, count.

```
def count_fibonacci(n):
    """ recursive Fibonacci + counting no. of function invocations
    global count
    count+=1
    if n<2:
        return 1
    else:
        return count_fibonacci(n-1)+count_fibonacci(n-2)

>>> count=0
>>> count_fibonacci(30)
1346269
>>> count
2692537

>>> count=0
>>> count_fibonacci(40)
102334155
>>> count
204668309  # over 200 million invocations
```

Count vs. `fibonacci(n)`

```
>>> count=0
>>> count_fibonacci(20)
10946
>>> count
21891
>>> count=0
>>> count_fibonacci(30)
1346269
>>> count
2692537
>>> count=0
>>> count_fibonacci(40)
165580141
>>> count
331160281    # over 300 million invocations
```

Can you see some relation between the returned value and `count`?
Do you think this is a coincidence?

Try to prove, using induction: `count(n)` $= 2 \cdot F_n - 1$.

Intuition for Efficiently Computing Fibonacci Numbers

Instead of computing from scratch, we will introduce variables `fib[0]`, `fib[1]`, `fib[2]`,.... The value of each such variable will be computed just once. Rather than recomputing it, we will fetch the value from memory, when needed.

The technique of storing values instead of re-computing them has different names in different contexts: It is known as memorization, a term coined by Donald Michie in 1968. In programming languages like Lisp (of which Scheme is a variant), where recursion is used heavily, there are programs to do this optimization automatically, at run time. These are often termed memoization.

In other contexts, such as stringology, this technique (remember and reuse computed values, rather than re-computing them) is often used as part of dynamic programming.

Python's Dictionary (dict)

In the next version of the function, fibonacci2, we will exploit a highly useful data structure in Python, the dictionary (class dict). This is a mutable class, storing key:value pairs. The keys (but not the values) should be immutable.

```
>>> students={"Dzutv Rztvsud":48322167, "Wpbo Fgrv":26753752}
        # creating a dictionary
>>> "Dzutv Rztvsud" in students  # membership query
True
>>> "Al Capone" in students
False
>>> students["Dzutv Rztvsud"]  # retrieving value of existing key
48322167
>>> students["Al Capone"]=48322167  # inserting a new key + value
>>> students
{'Wpbo Fgrv': 26753752, 'Dzutv Rztvsud': 48322167,
              'Al Capone': 48322167}
>>> type(students)
<class 'dict'>
```

Python's dict does not support having different items with the same key (it keeps only the most recent item with a given key).

Fibonacci Numbers: Recursive Code with Memoization

We will use a dictionary – an indexed data structure that can grow dynamically. This dictionary, which we name `fib`, will contain the Fibonacci numbers already computed.
We initialize the dictionary with `fib_dict[0]=1` and `fib_dict[1]=1`.

```
# Initial fib dictionary with first two elements
fib_dict = {0:1,1:1}

def fibonacci2(n):
    """ recursive Fibonacci, employing
    memorization in a dictionary """
    #print(n)  # diagnostic printing

    if n not in fib_dict:
        res = fibonacci2(n-1) + fibonacci2(n-2)
        fib_dict[n] = res
    return fib_dict[n]
```

(Instead of the global variable fib_dict, we could and should use the envelope function mechanism. This time we were lazy...)

Recursive Fibonacci Code with Memoization: Execution

`fibonacci2` is recursive, with exactly the same control flow of `fibonacci`, only it stores intermediate values that were already computed. This small change implies a huge performance difference:

```
>>> fib_dict;fibonacci2(5);fib_dict # 3 for the price of 1
{0: 1, 1: 1}
8
{0: 1, 1: 1, 2: 2, 3: 3, 4: 5, 5: 8}  # in case you were wondering
>>> elapsed("fibonacci(35)")
3.46876281551331 5
>>> elapsed("fibonacci2(35)")
8.053997947143898e-05
>>> elapsed("fibonacci2(35)")
5.1364782827079125e-05  # fib_dict already has all the values,
                        #  so no computation needed
>>> elapsed("fibonacci(40)")
38.77776019960766
>>>fib_dict = {0:1, 1:1}; elapsed("fibonacci2(40)")
7.396528727099394e-05
>>> elapsed("fibonacci(45)")
451.18211404296727    # 7.5 minutes !!
>>>fib_dict = {0:1, 1:1}; elapsed("fibonacci2(45)")
6.287049416187074e-05
```

Pushing Performance to the Limit

```
>>> elapsed("fibonacci2(500)")
0.0004189785626902008
>>> elapsed("fibonacci2(500)")
3.811481833038144e-05

#How come the second time is about 10 times faster??

>>> elapsed("fibonacci2(1000)")
0.0003692017477066045
>>> elapsed("fibonacci2(1200)")
0.0001581480521757328

#1200 takes less time than 1000??
```

Well, sometimes global variables have positive side effects...

Pushing Performance to the Limit (2)

```
>>> elapsed("fibonacci2(500)")
0.00044685357908099484
>>> fib_dict={0:1,1:1}     #start from scratch
>>> elapsed("fibonacci2(500)")
0.00036834843087873 99
>>> fib_dict={0:1,1:1}    #start from scratch
>>> elapsed("fibonacci2(500)")
0.00049862146666615751
>>> fib_dict={0:1,1:1}    #start from scratch
>>> elapsed("fibonacci2(1000)")

Traceback (most recent call last):
    # removed most of the error message
    fib_dict[n] = fibonacci2(n-1)+fibonacci2(n-2)
RuntimeError:  maximum recursion depth exceeded
```

What the $\#*\&$ is going on?
fibonacci2(1000) worked perfectly well in the last slide, with no initialization of the dictionary between calls!

Python Recursion Depth

While recursion provides a powerful and very convenient means to designing and writing code, this convenience is not for free. Each time we call a function, Python (and every other programming language) adds another "frame" to the current "environment". This entails allocation of memory for local variables, function parameters, etc.

Nested recursive calls, like the one we have in `fibonacci2`, build a deeper and deeper stack of such frames.

Most programming languages' implementations limit this recursion depth. Specifically, Python has a nominal default limit of 1,000 on recursion depth. However, the user (you, that is), can modify the limit (within reason, of course).

Pushing Performance to the Limit (3)

Suppose we changed the order of calls inside fibonacci2.
First we call n-2, then n-1.

```
def fibonacci2_reverse(n):
    """ recursive Fibonacci, employing
    memorization in a dictionary """
    #print(n)  # diagnostic printing

    if n not in fib_dict:
        res = fibonacci2_reverse(n-2) + fibonacci2_reverse(n-1)
        #changed the order
        fib_dict[n] = res
    return fib_dict[n]
```

What is the recursion depth of fibonacci2 now?

Pushing Performance to the Limit (3)

```
>>> elapsed("fibonacci2(1000)")
0.0007446611521540704
>>> fib_dict={0:1,1:1}
>>> elapsed("fibonacci2(1500)")
0.0010333666790582896
>>> fib_dict={0:1,1:1}
>>> elapsed("fibonacci2(1900)")
0.0014571807037775386
>>> fib_dict={0:1,1:1}
>>> elapsed("fibonacci2(2000)")

Traceback (most recent call last):
    # removed most of the error message
    fib_dict[n] = fibonacci2(n-2)+fibonacci2(n-1)
RuntimeError:  maximum recursion depth exceeded
```

So as you have probably understood, Python evaluates expressions from left to right (except for when otherwise dictated by precedence of operators).

See `https://docs.python.org/3.3/reference/expressions.html#evaluation-order` .

Changing Python Recursion Depth

You can import the Python `sys` library, find out what the limit is, and also change it.

```
>>> import sys
>>> sys.getrecursionlimit()     # find recursion depth limit
1000
>>> sys.setrecursionlimit(20000)     # change limit to 20,000
>>> fibonacci2(3000)
66439046036696007228021784786602838424416351245278325940557976554262121 4
16121925739644981098299982039113222680280946513244634933199440943492601 90
45342723749188530316994678473551320635101099619382973181622585687336939 78
43735278975554894868417261317338143401291756224504216051010258971732359 90
66277020375643878651753054710112374884914025268612010403264702514559895 667
59021350105669097831249594364698255583142897013542271517846028657107806 2467
51070565698228205428466603218138388962758197532813714918090044122191248 5637
51216948117287242136678145773266185214783576618590189673133548401784031 9755
99690565107917098591441733043648980 01     # hurray
```

fibonacci2 - The Better Style (Envelope Functions)

Global variables may be risky and are better avoided.
We could write fibonacci2 in another way, as an envelope function (you have already seen this with the recursive binary search).

```
def fibonacci2_no_global(n):
    """ Envelope function for Fibonacci,
        employing memoization in a dictionary """

    fib_dict={0:1, 1:1} # LOCAL initial dictionary

    def fib2(n, fib_dict):
        if n not in fib_dict:
            res = fib2(n-2, fib_dict) + fib2(n-1, fib_dict)
            fib_dict[n] = res
        return fib_dict[n]

    return fib2(n, fib_dict)
```

Note: unlike in the binary search envelope function example, here we decided to put the function fib2 inside fibonacci2_better. fib2 is a local name, that is not recognized outside.

Fibonacci Numbers: Iterative (Non Recursive) Solution

We saw that memoization improved the performance of computing Fibonacci numbers dramatically (the function `fibonacci2`).

We now show that to compute Fibonacci numbers, the recursion can be eliminated altogether:

This time, we will maintain a list data structure, denoted `fibb`. Its elements will be `fibb[0]`,`fibb[1]`,`fibb[2]`,...,`fibb[n]` ($n+1$ elements altogether for computing F_n).

Fibonacci Numbers: Iterative (Non Recursive) Solution, cont.

This time, we will maintain a list data structure, denoted `fibb`. Its elements will be `fibb[0],fibb[1],fibb[2],...,fibb[n]` ($n+1$ elements altogether for computing F_n).

Upon generating the list, all its values are set to 0.

Next, we initialize the values `fibb[0] = fibb[1] = 1`.

And then we simply iterate, determine the value of the k-th element, `fibb[k]`, after `fibb[k-2]`, and `fibb[k-1]` were already determined.

No recursion implies no nested function calls, hence reduced overhead (and no need to confront Python's recursion depth limit :-).

Iterative Fibonacci Solution: Python Code

```
def fibonacci3(n):
    """ iterative Fibonacci, employing
    memoization in a list """
    if n<2:
        return 1
    else:
        fibb = [0 for i in range(n+1)]
        fibb[0] = fibb[1]=1  # initialize
        for k in range(2,n+1):
            fibb[k] = fibb[k-1] + fibb[k-2] # update next element
        return fibb[n]
```

Recursive vs. Iterative: Timing

Let us now do some performance comparisons:
`fibonacci2` vs. `fibonacci3`:

```
>>> import sys
>>> sys.setrecursionlimit(20000)
>>> elapsed("fibonacci2(2000)")
0.003454221497536104
>>> elapsed("fibonacci3(2000)")
0.0008148609599825107
>>> elapsed("fibonacci2(2000)")
5.917295926849419 6e-05
>>> elapsed("fibonacci3(2000)")
0.0008156828066319122
>>>
```

As we saw, `fibonacci2` runs faster the second time (when `fib_dict` has already been computed).

Finally: Iterative Fibonacci Solution Using $O(1)$ Memory

No, we are not satisfied yet.

Think about the algorithm's execution flow. Suppose we have just executed the assignment `fibb[4]=fibb[2]+fibb[3]`. This entry will subsequently be used to determine `fibb[5]` and `fibb[6]`. But then we make no further use of `fibb[4]`. It just lies, basking happily, in the memory (much like good old Agama stellio).

The following observation holds in "real life" as well as in the "computational world": Time and space (memory, at least a computer's memory) are important resources that have a fundamental difference:
Time cannot be re-used, while memory (space) can be.

Iterative Fibonacci Reusing Memory

At any point in the computation, we can maintain just two values, `fibb[k-2]` and `fibb[k-1]`. We use them to compute `fibb[k]`, and then reclaim the space used by `fibb[k-2]` to store `fibb[k-1]` in it.

In practice, we will maintain two variables, `previous` and `current`. Every iteration, those will be updated. Normally, we would need a third variable `next` for keeping a value temporarily. However Python supports the "simultaneous" assignment of multiple variables (first the right hand side is evaluated, then the left hand side is assigned).

Iterative Fibonacci Reusing Memory: Code

```
def fibonacci4(n):
    """ fibonacci in O(1) memory """
    if n<2:
        return 1 # base case
    else:
        previous = 1
        current = 1
        for i in range(n-1): # n-1 iterations (count carefully)
            current, previous = previous+current, current
                    # simultaneous assignment
        return current

>>> for i in range(0,7):
                print(fibonacci4(i))
            # sanity check
1
1
2
3
5
8
13
```

Iterative Fibonacci Code, Reusing Memory: Performance

Reusing memory can surely help if memory consumption is an issue. Does it help with runtime as well?

```
>>> elapsed("fibonacci3(10000)",number=100)
0.7590410000000001
>>> elapsed("fibonacci4(10000)",number=100)
0.3688609999999999
>>> elapsed("fibonacci3(30000)",number=100)
4.650388
>>> elapsed("fibonacci4(30000)",number=100)
2.024259
>>> elapsed("fibonacci3(100000)",number=10)
6.150758999999999
>>> elapsed("fibonacci4(100000)",number=10)
1.8084930000000004
```

We see that there is about 50–70% saving in time. Not dramatic, but significant in certain circumstances.

The difference has to do with different speed of access to different level cache in the computer memory. The `fibonacci4` function uses $O(1)$ memory vs. the $O(n)$ memory usage of `fibonacci3`.

Closed Form Formula

And to really conclude our Fibonacci excursion, we note that there is a closed form formula for the n-th Fibonacci number,

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}} \ .$$

You can verify this by induction. You will even be able to derive it yourself, using generating functions (studied in the discrete mathematics course).

Closed Form Formula: Code, and Danger

```
def closed_fib(n):
    """ code for closed form Fibonacci number """
    return round(((1+5**0.5)**n-(1-5**0.5)**n)/(2**n*5**0.5))
>>> for i in range(1,6):
      print(i*10, fibonacci4(i*10), closed_fib(i*10))
         # sanity check
10 89 89
20 10946 10946
30 1346269 1346269
40 165580141 165580141
50 20365011074 20365011074
```

However, being aware that floating point arithmetic in Python (and other programming languages) has finite precision, we are not convinced, and push for larger values:

```
>>> for i in range(40,90):
            if fibonacci4(i) != closed_fib(i):
              print(i, fibonacci4(i), closed_fib(i))
              break

70 308061521170129 308061521170130
```

Bingo!

Reflections: Memoization, Iteration, Memory Reuse

In the Fibonacci numbers example, all the techniques above proved relevant and worthwhile performance wise. These techniques won't always be applicable for every recursive implementation of a function.

Consider `quicksort` as a specific example. In any specific execution, we never call quicksort on the same set of elements more than once (think why this is true).
So memoization is not applicable to quicksort. And replacing recursion by iteration, even if applicable, may not be worth the trouble and surely will result in less elegant and possibly more error prone code.

Even if these techniques are applicable, the transformation is often not automatic, and if we deal with small instances where performance is not an issue, such optimization may be a waste of effort.