Computer Science 1001.py, Lecture 13a
Hanoi Towers' Monster
Tail Recursion
Ackermann Function
Munch!

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Amir Gilad, Michal Kleinbort

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University, Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

Lecture 12 Highlights

- Recursion is a powerful control mechanism.
- In many cases, it is easy and natural to code recursively.
- Code often directly follows recursive definitions.
- Recursive definitions with simple, highly concise code can hide terrible (exponential time) performance.
- In some cases, techniques like memorizing/dynamic programming lead to more efficient code.

- Recursion depth limit.
- Towers of Hanoi.

Lecture 13: Plan

More recursion:

- Code for `Hanoi Towers`.

- The `Hanoi Towers` Monster: Finding individual moves.
- The `Hanoi Towers` Monster: Binary search to the rescue.
- Tail recursion (for reference only).
- Ackermann function: Recursive definitions with simple, highly compact code can describe super fast growing functions.
- Munch!

Recursive Hanoi Towers Code

The following code prints the individual moves required to relocate a stack of n discs, initially on rod `start`, to rod `target`, employing intermediate rod `via`.

```
def HanoiTowers(start,via,target,n):
    """ prints a list of discs moves, such that a stack of n
    discs is moved from rod "start" to rod "target", employing
    intermediate rod "via" """
    if n==0:
        return None
    else:
        HanoiTowers(start,target,via,n-1)
        print("disk", n, "from", start, "to", target)
        HanoiTowers(via,start,target,n-1)
```

We have set `n == 0` as the base case of the recursion (in that case, `None` is returned, and the recursion stops). Every positive value of `n` results in two recursive calls with `n-1`, and one print (corresponding to the move of the bottom disc at the current recursion level).

Recursive Hanoi Towers Code: Executions

We test the code on some small cases, which can be verified manually.

```
>>> HanoiTowers("A", "B", "C", 1)
disk 1 from A to C

>>> HanoiTowers("A", "B", "C", 2)
disk 1 from A to B
disk 2 from A to C
disk 1 from B to C

>>> HanoiTowers("A", "B", "C", 3)
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
```

And Now For Something Completely Different: Monsters, Nightmares, and the Towers of Hanoi

(figure from http://unrealitymag.com/index.php/2010/06/25/scary-monster-pictures/)

Towers of Hanoi Nightmare

Suppose your partner* wakes you in the middle of the night after having a terrible dream. A monster demanded to know what is the $3^{97}+19$ move in an $n=200$ disk Towers of Hanoi puzzle, or else . . . .

Having seen and even understood the material in last class, you quickly explain to your frightened partner that either expanding all $H(200)=2^{200}-1$ moves, or even just the first $3^{97}+19$, is out of computational reach in any conceivable future, and the monster should try its luck elsewhere.

Your partner is not convinced.

You eventually wake up and realize that your best strategy for sleeping the rest of the night is to solve this new problem. The first step towards taming the monster is to give the new problem a name: `hanoi_move(start,via,target,n,k)`.

---

*we are very careful to be gender and sexual inclination neutral here

Feeding the Towers of Hanoi Monster

The solution to `HanoiTowers(start,via,target,n)` consisted of three (unequal) parts, requiring $2^n - 1$ steps altogether:

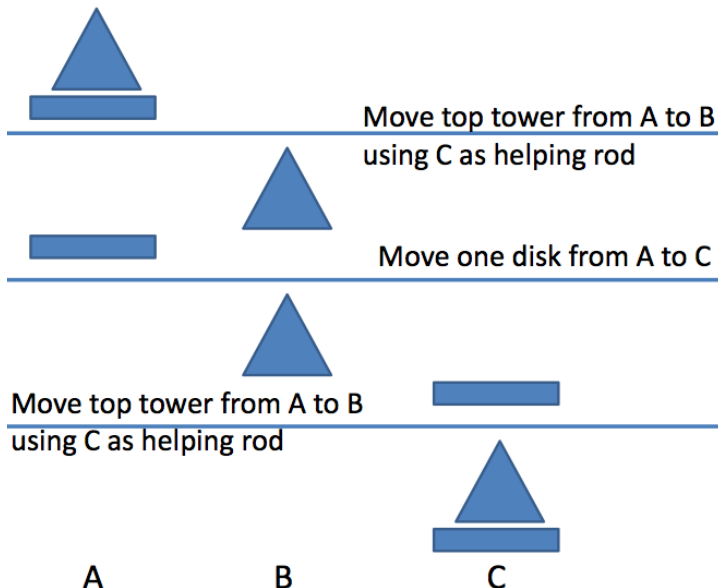
In the first part, which takes $2^{n-1} - 1$ steps, we move $n - 1$ disks.
In the second part, which takes exactly one step, we move disk number n.
In the last part, which again takes $2^{n-1} - 1$ steps, we move $n - 1$ disks.

The monster wants to know step number k of $2^n - 1$ steps. If $k = 2^{n-1}$, this is the "middle step": `move disk n from start to target`.

But what do we do if $k \neq 2^{n-1}$?

Well, we think recursively:

Visualizing n Disk Towers of Hanoi (reminder)



Feeding the Towers of Hanoi Monster, cont.

The solution to `HanoiTowers(start,via,target,n)` consisted of three (unequal) parts, requiring $2^n - 1$ steps altogether:
But what do we do if $k \neq 2^{n-1}$?

Well, we think recursively:

If $k < 2^{n-1}$ then this is move number k in an $n - 1$ disks problem (part one of the n disks problem).

If $k > 2^{n-1}$ then this is move number $k - 2^{n-1}$ in an $n - 1$ disks problem (part three of the n disks problem).

We should keep in mind that the roles of the rods in these smaller subproblems will be permuted.

Recursive Monster Code

```
def hanoi_move(start,via,target,n,k):
    """ finds the k-th move in an Hanoi Towers instance
    with n discs """
    if n <=  0:
        return "zero or fewer disks"
    elif k <=  0 or k >=  2**n or type(k)!=int:
        return "number of moves is illegal"
    elif k == 2**(n-1):
        return str.format("disk {} from {} to {}",
                          n, start, target)
    elif k < 2**(n-1):
        return hanoi_move(start,target,via,n-1,k)
    else:
        return hanoi_move(via,start,target,n-1,k-2**(n-1))
```

Recursive Monster Code: Executions

We first test it on some small cases, which can be verified by running the `HanoiTowers` program. Once we are satisfied with this, we solve the monster's question (and get back to sleep).

```
>>> hanoi_move("A","B","C",1,1)
'disk 1 from A to C'
>>> hanoi_move("A","B","C",2,1)
'disk 1 from A to B'
>>> hanoi_move("A","B","C",2,2)
'disk 2 from A to C'
>>> hanoi_move("A","B","C",3,7)
'disk 1 from A to C'
>>> hanoi_move("A","B","C",4,8)
'disk 4 from A to C'
>>> hanoi_move("A","B","C",200,3**97+19)
'disk 2 from B to A'     # saved!
```

Recursive Monster Solution and Binary Search

The recursive `hanoi_move(start,via,target,n,k)` makes at most one recursive call.

The way it "homes" on the right move employs the already familiar paradigm of binary search: It first determines if move number k is exactly the middle move in the n disk problem. If it is, then by the nature of the problem it is easy to exactly determine the move.

If not, it determines if the move is in the first half of the moves' sequence ($k < 2^{n-1}$) or in the second half ($k > 2^{n-1}$), and makes a recursive call with the correct permutation of rods.

The execution length is linear in n (and not in $2^n - 1$, the length of the complete sequence of moves).

Reflections on Binary Search

We have already seen binary search and realized it is widely applicable (not only when monsters confront you at night). We can use binary search when we look for an item in a huge space, in cases where that space is structured so we could tell if the item is

1. right at the middle,
2. in the top half of the space,
3. or in the lower half of the space.

In case (1), we solve the search problem in the current step. In cases (2) and (3), we deal with a search problem in a space of half the size.

In general, this process will thus converge in a number of steps which is $\log_2$ of the size of the initial search space. This makes a huge difference. Compare the performance to going linearly over the original space of $2^n - 1$ moves, item by item.

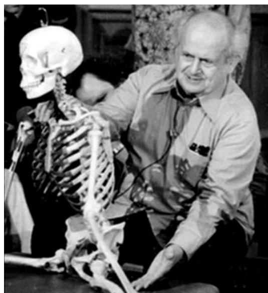
Binary Search (and the Feldenkrais Method, Again)

Quoting (again) Dr. Moshé Feldenkrais (1904-1984), his method "...makes the impossible possible, the possible comfortable, and the comfortable pleasant."

Let us borrow Dr. Feldenkrais' description to binary search.

In the case of searching an ordered list, we could classify binary search as making the possible comfortable.

Here, the search space is so huge that binary search is definitely making the impossible possible.



Replacing Iteration with Recursion

Due to timing constraints, all this section is “for reference only”.

Replacing Iteration with Recursion

Suppose you wanted to compute $\sum_{k=1}^{n} k$, which in Python is simply `sum(range(1,n+1))`. However, for some reason, the lunatic course staff insists that you do not use `for`, `while`, or `sum`.

What would you do?

Well, luckily you have just learned about recursion.

```
def recursive_sum(n):
    """ computes 1+2+...+n recursively """
    if n <= 1:
        return 1
    else:
        return n + recursive_sum(n-1)
```

This solution is fine, but much less efficient than the iterative solution. Furthermore, as soon as n approaches, say, 1000, you will hit Python's recursion limit and crash. Of course you can set a higher recursion limit, but even $n = 1000000$ won't be feasible, while it is a piece of cake for `sum`.

Tail Recursion

We modify the code such that the recursive call occurs exactly at the end of the function body, and furthermore it is a pure recursive call (the result of the recursive call is returned AS IS as the result of the current call, and is not mixed with other operations such as the + above). To this end, we introduce an auxiliary argument, `partial`.

```
def tail_sum(n,partial=0):
    """ computes partial+(1+2+...+n) with tail recursion """
    if n <=  0:
        return partial
    else:
        return tail_sum(n-1,partial+n)
```

And then

```
def my_sum(n):
        """ computes 0+1+2+...+n with tail recursion """
        return tail_sum(n)
```

Tail Recursion, a Few Execution

```
# note: the argument n better not be too large

>>> tail_sum(10)
55
>>> tail_sum(10,-5)
50

>>> tail_sum(0)
0

>>> my_sum(10)
55
```

tail_sum: Manual Removal of Recursion

```python
def tail_sum(n,partial=0):
    """ computes partial+(1+2+...+n) with tail recursion """
    if n <=  0:
        return partial
    else:
        return tail_sum(n-1,partial+n)
```

By carefully analyzing the code, we observe that the tail recursion can be replaced by a simple `while` iteration, where `n` is the variable controlling the iteration.

```python
def iter_sum(n,partial=0):
    """ computes partial+(1+2+...+n) with iteration """
    while n >=  0:
        partial = partial + n
        n = n-1
    return partial
```

Tail Recursion: Automatic Removal of Recursion

```
def tail_sum(n,partial=0):
    """ computes partial+(1+2+...+n) with tail recursion """
    if n <=  0:
        return partial
    else:
        return tail_sum(n-1,partial+n)
```

A smart compiler can automatically convert this code into a more efficient one, employing no recursive calls at all: When reaching the tail, no new function frame is built, Instead, the function is “freshly invoked” (no history kept) with the parameters `n-1,partial+n`.

Recursion of that form is called tail recursion. This optimization is frequently used in functional languages (including Scheme) where recursion is the main control mechanism.

Tail Recursion: A Graphical Illustration [†]

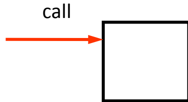
[†] designed by Prof. Amiram Yehudai

graphical illustration: recursive process

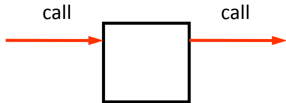
graphical illustration: recursive process

call

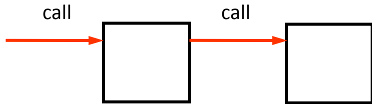
graphical illustration: recursive process



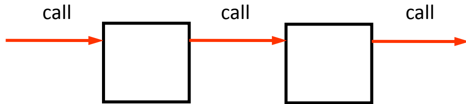
graphical illustration: recursive process



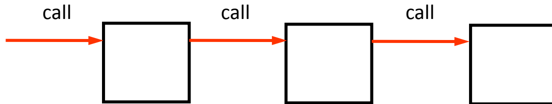
graphical illustration: recursive process



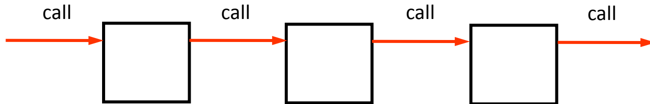
graphical illustration: recursive process



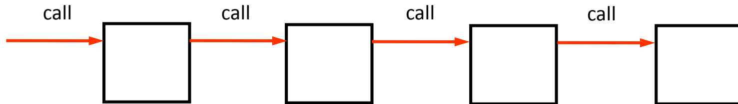
graphical illustration: recursive process



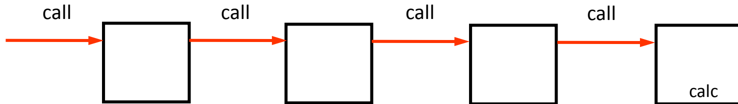
graphical illustration: recursive process



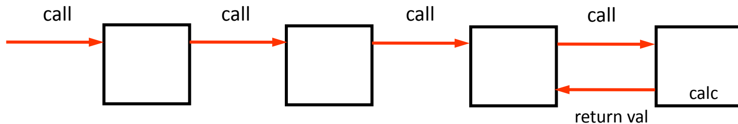
graphical illustration: recursive process



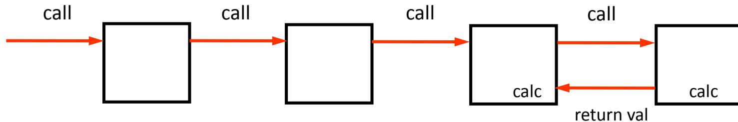
graphical illustration: recursive process



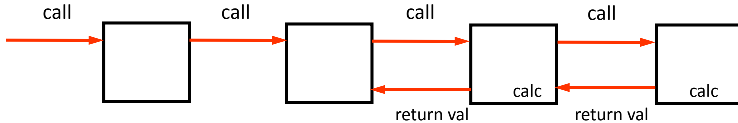
graphical illustration: recursive process



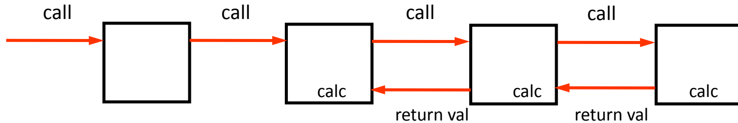
graphical illustration: recursive process



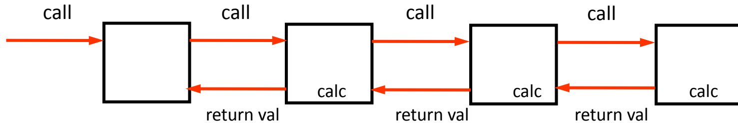
graphical illustration: recursive process



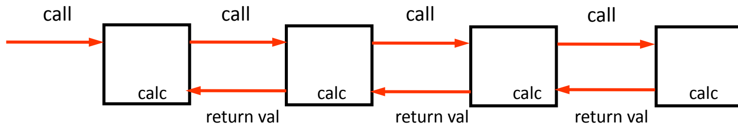
graphical illustration: recursive process



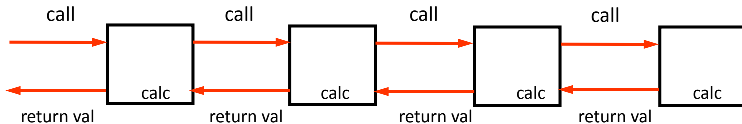
graphical illustration: recursive process



graphical illustration: recursive process

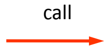


graphical illustration: recursive process

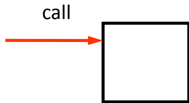


tail recursive process

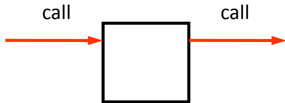
tail recursive process



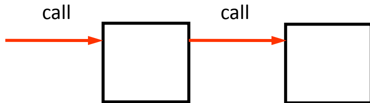
tail recursive process



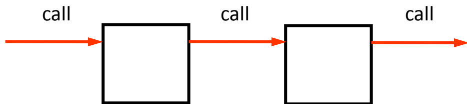
tail recursive process



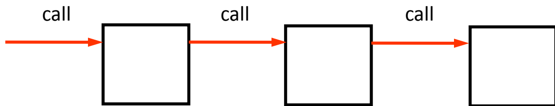
tail recursive process



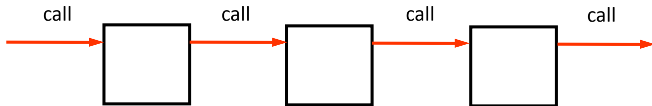
tail recursive process



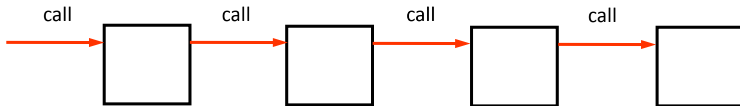
tail recursive process



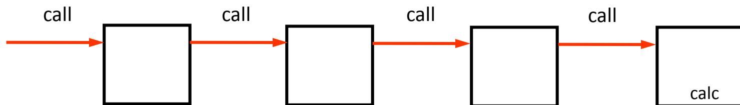
tail recursive process



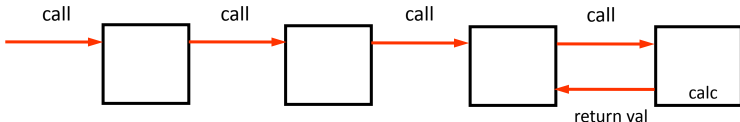
tail recursive process



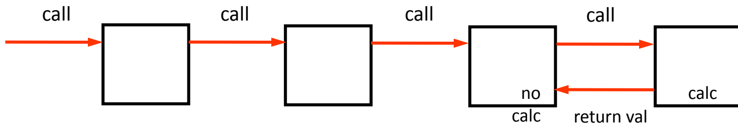
tail recursive process



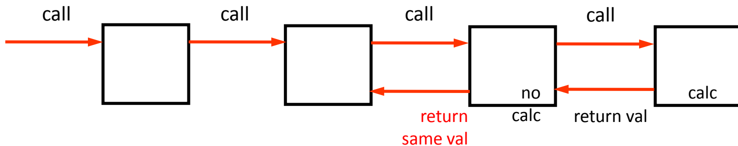
tail recursive process



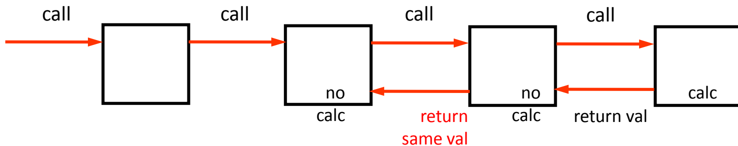
tail recursive process



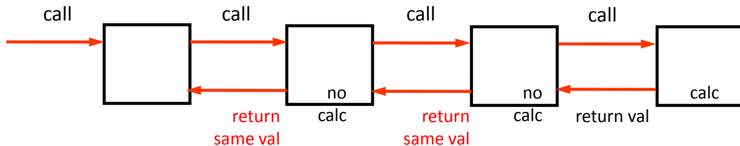
tail recursive process



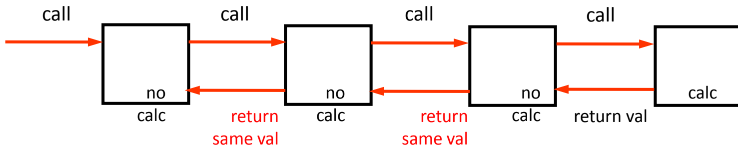
tail recursive process



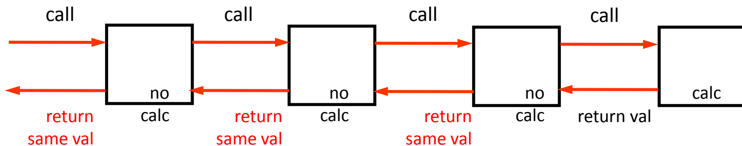
tail recursive process



tail recursive process



tail recursive process

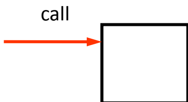


converting a tail recursive to an iterative process

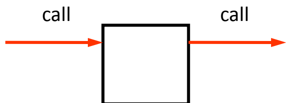
converting a tail recursive to an iterative process

call

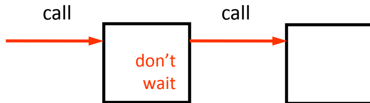
converting a tail recursive to an iterative process



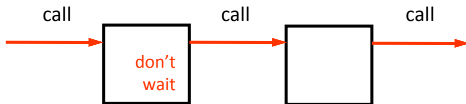
converting a tail recursive to an iterative process



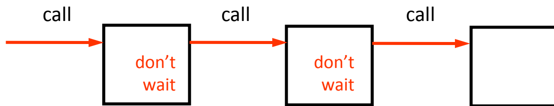
converting a tail recursive to an iterative process



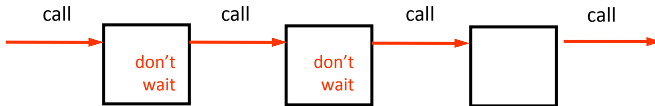
converting a tail recursive to an iterative process



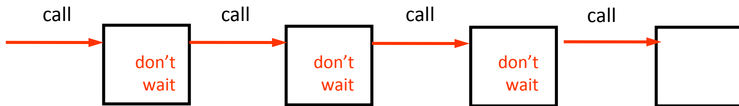
converting a tail recursive to an iterative process



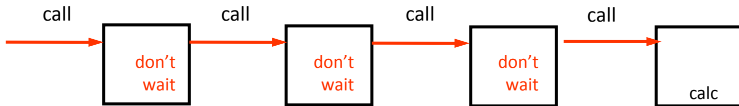
converting a tail recursive to an iterative process



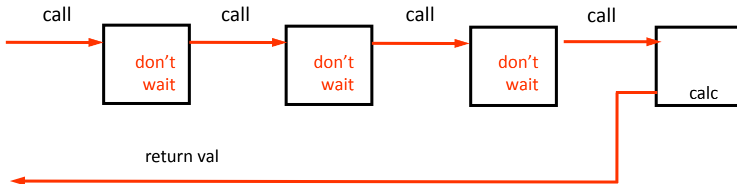
converting a tail recursive to an iterative process



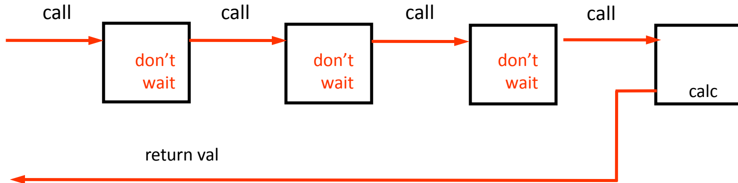
converting a tail recursive to an iterative process



converting a tail recursive to an iterative process



converting a tail recursive to an iterative process



In functional languages, such as scheme, a tail recursive function is converted to an iterative one.

Recursive Monster Code (revisited)

```
def hanoi_move(start,via,target,n,k):
    """ finds the k-th move in an Hanoi Towers instance
    with n discs """
    if n <=  0:
        return "zero or fewer disks"
    elif k <=  0 or k >=  2**n or type(k)!=int:
        return "number of moves is illegal"
    elif k == 2**(n-1):
        return str.format("disk {} from {} to {}",
                          n, start, target)
    elif k < 2**(n-1):
        return hanoi_move(start,target,via,n-1,k)
    else:
        return hanoi_move(via,start,target,n-1,k-2**(n-1))
```

We observe that the code makes a single recursive call, and furthermore the retuned value is just the value of the recursive call, with no additional computation.

Thus, the recursion in our Hanoi monster code is an instance of tail recursion, and iteration can thus replace recursion completely.

Hanoi Monster Code: from Recursion to Loop

```
if k == 2**(n-1):
    return str.format("disk {} from {} to {}",
                      n, start, target)
elif k < 2**(n-1):
    start,via,target = start,target,via # permute rods
    n,k = n-1,k    # decrease n, keep k unchanged
elif k > 2**(n-1):
    start,via,target = via,start,target # permute rods
    n,k = n-1, k-2**(n-1) # decrease n and k
```

Replace all recursive calls by assignments to the parameters' values.

Hanoi Monster Code: from Recursion to Loop

```
while True:
    if k == 2**(n-1):
        return str.format("disk {} from {} to {}",
                           n, start, target)
    elif k < 2**(n-1):
        start,via,target = start,target,via # permute rods
        n,k = n-1,k   # decrease n, keep k unchanged
    elif k > 2**(n-1):
        start,via,target = via,start,target # permute rods
        n,k = n-1, k-2**(n-1) # decrease n and k
```

And enclose this part in a `while True` loop.

Hanoi Monster Code: Iterative Version

```
def hanoi_move_tail(start,via,target,n,k):
    """ finds the k-th move in an Hanoi Towers instance
    with n discs, removed tail recursion """
    if n <=  0:
        return "zero or fewer disks"
    elif k <=  0 or k >=  2**n or type(k)!=int:
        return "number of moves is illegal"
    else:
        while True:
            if k == 2**(n-1):
                return str.format("disk {} from {} to {}",
                                  n, start, target)
            elif k < 2**(n-1):
                start,via,target = start,target,via # permute rods
                n,k = n-1,k   # decrease n, keep k unchanged
            elif k > 2**(n-1):
                start,via,target = via,start,target # permute rods
                n,k = n-1, k-2**(n-1) # decrease n and k
```

Hanoi Monster: Recursive vs. Iterative

We will check the correctness of the iterative code by

- Running it on a small instance where moves can be verified directly.
- Running it on larger instances and verifying that its results are identical to the results of the recursive algorithm.

This is not a proof of correctness, but at least it reinforces our belief in this code.

We maintain that the recursive solution is easier to come up with, and it is somewhat easier to verify its correctness.

Yet the iterative solution is more efficient, and can handle a large number of disks (say $n = 10^6$), where the recursion solution fails.

Recursive vs. Iterative: A Small Sanity Check

```
>>> for elem in HanoiTowers("A","B","C",4):
        print(elem)
disk 1 from A to B
disk 2 from A to C
disk 1 from B to C
disk 3 from A to B
disk 1 from C to A
disk 2 from C to B
disk 1 from A to B
disk 4 from A to C
disk 1 from B to C
disk 2 from B to A
disk 1 from C to A
disk 3 from B to C
disk 1 from A to B
disk 2 from A to C
disk 1 from B to C
>>> hanoi_move("A","B","C",4,5)
'disk 1 from C to A'
>>> hanoi_move_tail("A","B","C",4,5)
'disk 1 from C to A'
>>> hanoi_move("A","B","C",4,9)
'disk 1 from B to C'
>>> hanoi_move_tail("A","B","C",4,9)
'disk 1 from B to C'
```

Recursive vs. Iterative: More Sanity Checks

```
>>> hanoi_move_tail("A","B","C",400,5**90)
'disk 1 from A to B'
>>> hanoi_move("A","B","C",400,5**90)
'disk 1 from A to B'

>> hanoi_move_tail("A","B","C",1000,5**201+17)
'disk 2 from B to A'
>>> hanoi_move("A","B","C",1000,5**201+17)
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    hanoi_move("A","B","C",1000,5**201+17)
  File "/Users/admin/Dropbox/InttroCS2012/Code2014/intro12/hanoi_it
    return hanoi_move(start,target,via,n-1,k)
*
*
*
    return hanoi_move(start,target,via,n-1,k)
  File "/Users/admin/Dropbox/InttroCS2012/Code2014/intro12/hanoi_it
    elif k <=  0 or k >=  2**n or type(k)!=int:
RuntimeError: maximum recursion depth exceeded in comparison

>>> hanoi_move_tail("A","B","C",10**4,5**900+97)
'disk 2 from A to C'
```

Recursive vs. Iterative: Timing

```
>>> elapsed("""hanoi_move("A","B","C",900,2**900-1)""")
0.0072909999999950514
>>> elapsed("""hanoi_move_tail("A","B","C",900,2**900-1)""")
0.0050880000000006476

>>> elapsed("""hanoi_move("A","B","C",900,5**90+97)""")
0.00791399999999533
>>> elapsed("""hanoi_move_tail("A","B","C",900,5**90+97)""")
0.00357100000000879
```

The iterative solution is just under two times faster than the recursive one. This is not that impressive!.

Iterative solutions written by CS1001.py students ET and RT (Erez Timnat and Roi Tagar) in 2011 (code given in Appendix, for reference only) attained speed up factors of 1000 and even more. However, they required a deeper understanding of the algorithm's dynamics, and were much much harder to comprehend.

Recursive vs. Iterative: Summary

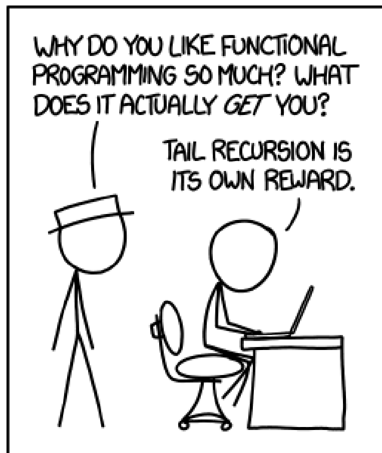
As we have seen, conversion from tail recursion to iteration is easy, and can even be done automatically.

Converting non-tail recursion to iteration (or to tail recursion) is harder. Sometimes it may be accomplished by introducing additional parameters to pass the state of the computation (as in `tail_sum`).

It is more complicated for non-linear recursion (when a function calls itself more than once).

In principle, it is always possible to convert recursion to iteration by simulating the control, using a special data structure to keep track of frames' stack. But this would produce a very complicated algorithm.

XKCD View on Tail Recursion[‡]



[‡]thanks to Prof. Amiram Yehudai for pointing this out

Recursion in Other Programming Languages

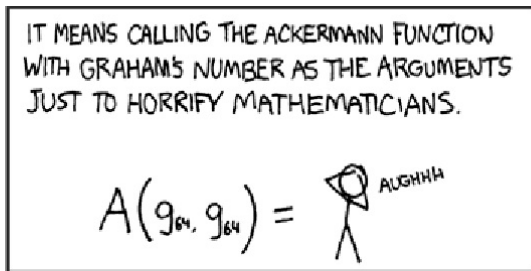
Python, C, Java, and most other programming languages employ recursion as well as a variety of other flow control mechanisms.

By way of contrast, all LISP dialects (including Scheme) use recursion as their major control mechanism. We saw that recursion is often not the most efficient implementation mechanism.

Taken together with the central role of eval in LISP, this may have prompted the following statement, attributed to Alan Perlis of Yale University (1922-1990): "LISP programmers know the value of everything, and the cost of nothing".

In fact, the origin of this quote goes back to Oscar Wilde. In The Picture of Dorian Gray (1891), Lord Darlington defines a cynic as "a man who knows the price of everything and the value of nothing".

And Now For Something Completely Different: the Ackermann Function



(taken from http://xkcd.com/207/)

Not for the Soft At Heart: the Ackermann Function

This recursive function, invented by the German mathematician Wilhelm Friedrich Ackermann (1896–1962), is defined as following:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \ . \end{cases}$$

This is a total recursive function, namely it is defined for all arguments (pairs of non negative integers), and is computable (it is easy to write Python code for it). However, it is so called non primitive recursive function, and one manifestation of this is its huge rate of growth.

You will meet the inverse of the Ackermann function in the data structures course as an example of a function that grows to infinity very very slowly.

Ackermann function: Python Code

Writing down Python code for the Ackermann function is easy – just follow the definition.

```
def ackermann(m,n):
    # Ackermann function
    if m == 0:
        return n+1
    elif m > 0 and n == 0:
        return ackermann(m-1,1)
    else:
        return ackermann(m-1,ackermann(m,n-1))
```

However, running it with $m \geq 4$ and any positive n causes run time errors, due to exceeding Python's maximum recursion depth. Even `ackermann(4,1)` causes such a outcome.

Ackermann function and Python Recursion Depth

However, running it with $m \geq 4$ and any positive n causes run time errors, due to exceeding Python's maximum recursion depth. Even `ackermann(4,1)` causes such outcome.

Not deterred by this, we increase the recursion depth limit, say to 100,000.

```
>>> sys.setrecursionlimit(100000)
```

Even with such larger recursion depth limit, the code crashed on an 8GB RAM machine running either MAC OSX 10.6.8 or Windows 7, reporting "segmentation fault".

Only on a Linux machine did `ackermann(4,1)` ran to completion (returning just `65,533`, by the way).

Adventurous? `ackermann(4,2)` will exceed any recursion depth that Python will accept, and cause your execution to crash.

This is what happens with such rapidly growing functions.

Appendix: Iterative Solutions to the Monster Problem (reference only)

(timings employ the `timeit` rather than the `clock` module.)

Recursive Monster Code

```
def hanoi_move(start,via,target,n,k):
    """ finds the k-th move in Hanoi Towers instance
    with n disks """
    if n <=  0:
        return "zero or fewer disks"
    elif k <=  0 or k >=  2**n or type(k)!=int:
        return "number of moves is illegal"
    elif k == 2**(n-1):
        return str.format("disk {} from {} to {}",n,start,target)
    elif k < 2**(n-1):
        return hanoi_move(start,target,via,n-1,k)
    else:
        return hanoi_move(via,start,target,n-1,k-2**(n-1))
```

During the class in Fall 2011, I was asked if we can eliminate recursion and use iteration instead. At this time I was not yet aware of the tail recursion removal option, so I mumbled something to the extent that I am not sure, it may be possible, and even if we can, it will be hard to beat the simplicity and elegance of the recursive solution.

Hanoi Monster Solution: Iterative Solution

All these claims were valid (especially the "I am not sure"). Yet, riding my bike home, I thought that if k is very simple (*e.g.* a power of two), a non recursive solution should be possible. I did not push this any further, though. When I got home, I found this message in my mailbox:

```
From:  Erez Timnat <ereztimn@mail.tau.ac.il>
Subject:  Hanoi Towers
Date:  November 30, 2011 7:00:22 PM GMT+02:00
To:  benny chor <benny@cs.tau.ac.il>

I enjoyed your lecture today on hanoi towers, and so I
wondered if I could implement an iterative solution to
hanoi_move.  So I did.  It uses the binary representation of
k, and I think it's quite cool.
So - I'm attaching hereby my implementation, and I hope
you'll find it useful/interesting/amusing.

Thanks,

Erez
```

Hanoi Monster Solution: Erez Timnat's Iterative Solution

```
def nbinary(n,k):
    """ computes the binary representation of k, and pads it
    with leading zeroes if the result is shorter than n bits """
    k=bin(k)
    k=[k[i] for i in range(2,len(k))]
    return (n-len(k))*['0']+k

def erez_iter_hanoi_move(start,via,target,n,k):
    """ iterative solution to the hanoi move problem """
    if n <=  0:
        return "zero or fewer disks"
    elif k <=  0 or k >=  2**n or type(k)!=int:
        return "number of moves is illegal"
    k=nbinary(n,k)
    for i in range(n):
        if k[i] == '1':
            disk=n-i
            disk_start,disk_target=start,target
            start,via=via,start
        else:   #k[i] == '0'
            via,target=target,via
    return str.format("disk {} from {} to {}", \
          disk, disk_start, disk_target)
```

Hanoi Monster: Recursive vs. Iterative

We will not explain the code in class.
We do recommend you try figuring it out yourself.
Writing your own iterative Hanoi Towers code may be a good idea.

We will check the correctness of Erez' iterative code by

- Running it on a small instance where moves can be verified directly.
- Running it on larger instances and verifying that its results are identical to the results of the recursive algorithm.

This is not a proof of correctness, but at least it reinforces our belief in this code.

We maintain that the recursive solution is easier to come up with, and it is much easier to verify its correctness.

Yet the iterative solution is way more efficient, and can handle a large number of disks (say $n = 10^6$), where the recursion solution fails.

Recursive vs. Iterative: A Small Sanity Check

```
>>> for elem in HanoiTowers("A","B","C",4):
    print(elem)
disk 1 from A to B
disk 2 from A to C
disk 1 from B to C
disk 3 from A to B
disk 1 from C to A
disk 2 from C to B
disk 1 from A to B
disk 4 from A to C
disk 1 from B to C
disk 2 from B to A
disk 1 from C to A
disk 3 from B to C
disk 1 from A to B
disk 2 from A to C
disk 1 from B to C
>>> hanoi_move("A","B","C",4,5)
'disk 1 from C to A'
>>> erez_iter_hanoi_move("A","B","C",4,5)
'disk 1 from C to A'
>>> hanoi_move("A","B","C",4,9)
'disk 1 from B to C'
>>> erez_iter_hanoi_move("A","B","C",4,9)
'disk 1 from B to C'
```

Recursive vs. Iterative: More Sanity Checks

```
>>> erez_iter_hanoi_move("A","B","C",400,5**90)
'disk 1 from A to B'
>>> hanoi_move("A","B","C",400,5**90)
'disk 1 from A to B'

>>> erez_iter_hanoi_move("A","B","C",8000,5**900+75)
'disk 3 from A to B'
>>> hanoi_move("A","B","C",8000,5**900+75)
'disk 3 from A to B'

>>> erez_iter_hanoi_move("A","B","C",23000,5**1900+76)
'disk 1 from C to A'
>>> hanoi_move("A","B","C",23000,5**1900+76)

  File "/Users/admin/Documents/InttroCS2011/Code/intro11/lecture11_
    elif k <=  0 or k >=  2**n or type(k)!=int:
RuntimeError: maximum recursion depth exceeded in comparison

>>> erez_iter_hanoi_move("A","B","C",10**7,5**9000+97)
                    # ten million disks
'disk 2 from A to C'
```

Recursive vs. Iterative: Timing

```
>>> import timeit
>>> def recurse():
    return hanoi_move("A","B","C",8500,5**900+7**800)
>>> def iterate():
    return erez_iter_hanoi_move("A","B","C",8500,5**900+7**800)

>>> timeit.timeit(recurse,number=10)
9.107219934463501
>>> timeit.timeit(iterate,number=10)
0.023094892501831055
```

The iterative solution is `9.1/0.023 = 394.3` times faster than the recursive one!

The poor recursive solution is elegant indeed, but deep inside it stacks 8,500 frames on top of each other, and this takes time.

Hanoi Monster Solution: Roi Tagar's Iterative Solution

After this presentation was already in press (this term used to mean "publication still being printed, and not yet available for sale". In our context, however, this means it was too late for me to verify and update the presentation before class), I got a second e-mail from a student in the class, suggesting a different iterative solution.

The student, Roi Tagar, observes that both Erez' and his solutions employ repeated patterns in the sequences of moves. The patterns depend on the parity of n, the total number of disks, and on the identity of the disk.

Both algorithms look for 1s in the binary representation of k. One difference is that the solution of Erez employs strings for this task, while Roi's solution employs integers.

Roi Tagar's Iterative Solution: Helping Code

```
def format_move(disk, move):
    return str.format('disk {} from {} to {}', \
    disk, move[0], move[1])

def compose_moves(start, via, target):
    '''
    Generates a 'moves' list, that matches the pattern
    The result for "A", "B", "C"
    moves=['AB', 'BC', 'CA', 'AC', 'CB', 'BA']
    '''
    return [(start,via), (via,target), (target,start),\
            (start,target), (target,via), (via,start)]
```

Roy Tagar's Iterative Solution: Main Code

```
def roi_iter_hanoi_move(start, via, target,n,k):
    ''' this solution uses a repeating pattern found in the
    problem's solutions. It satisfies 2 conditions:
    1. disk 1 for even n goes by the pattern:
            start->via, via->target, target->start
       disk 1 for odd n goes by the pattern:
            start->target, target->via, via->start
    2. disk a for n disks acts like disk a-1 for n-1 disks  '''

    moves = compose_moves(start, via, target)
    if k == 2**n - 1:
        return format_move(n, (start, target))
# seeking the LSB that equals 1 in the binary representation of k
    times = 0
    while k % 2 == 0:
        times+=1
        k//=2
    # now k%2 == 1
    disk = 1 + times
    move = moves[((k-1)//2)%3 + 3*((n-times)%2)]

    return format_move(disk, move)
```

Roi Tagar's Iterative Solution: Sanity Check and Timing

```
>>> roi_iter_hanoi_move("A","B","C",400,5**90)
'disk 1 from A to B'
>>> roi_iter_hanoi_move("A","B","C",8000,5**900+75)
'disk 3 from A to B'
>>> roi_iter_hanoi_move("A","B","C",23000,5**1900+76)
'disk 1 from C to A'
>>> roi_iter_hanoi_move("A","B","C",10**7,5**9000+97)
'disk 2 from A to C'

>>> def roi_iter():
    return roi_iter_hanoi_move("A", "B", "C", 8500, 5**900+7**800)

>>> timeit.timeit(roi_iter, number=10)
0.0006730556488037109
```

Hanoi Monster Solution: RT vs. ET Iterative Solutions

In addition to using integers vs. strings, another difference is that RT's code does not perform `n` iterations in every inner loop. Instead, it halts when the least significant bit that equals `1` is detected.

As we just saw, these seemingly small changes imply a factor 34 improvement for the last specific input, when compared to ET's iterative solution.

Roi notes that lower-level languages, using knowledge about the system architecture for large integer representations, might have more efficient solutions for this problem, (*e.g.* a 32 bit processor could do a 32 bit scan in one clock cycle).

Both iterative algorithms required a deeper understanding of the structure of the problem than what was needed to design the recursive code. In both cases, this better understanding resulted in vastly improved performances. As is often the case, the iterative codes are somewhat more complex than the recursive one.

Merge Sort with Transylvanian-Saxon (German) folk dance

https://www.youtube.com/watch?v=XaqR3G_NVoo