Extended Introduction to Computer Science CS1001.py

Lecture 13: Recursion (4) - Hanoi Towers, Munch!

Instructors: Daniel Deutch, Amir Rubinstein,

Teaching Assistants: Amir Gilad, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Winter Semester, 2017-18
http://tau-cs1001-py.wikidot.com

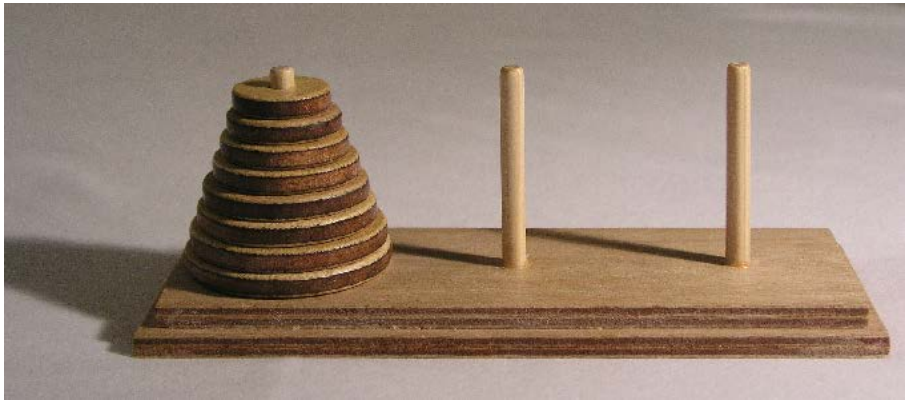
Lectures 13: Plan

- Recursion: basic examples and definition
  - Fibonacci
  - factorial
- Binary search - revisited
- Sorting
  - Quick-Sort
  - Merge-Sort
- Improving recursion with memoization
- Towers of Hanoi
- Munch!

today

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)

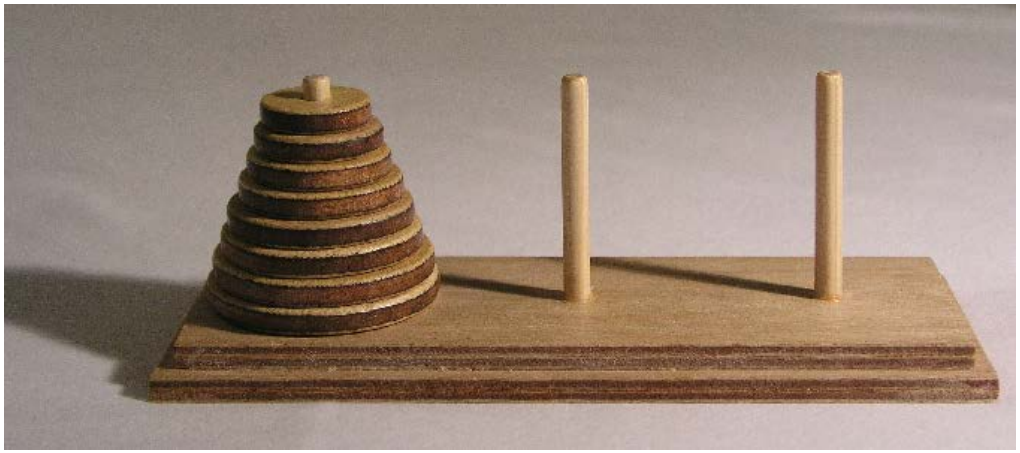
Towers of Hanoi - origin

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 - Description

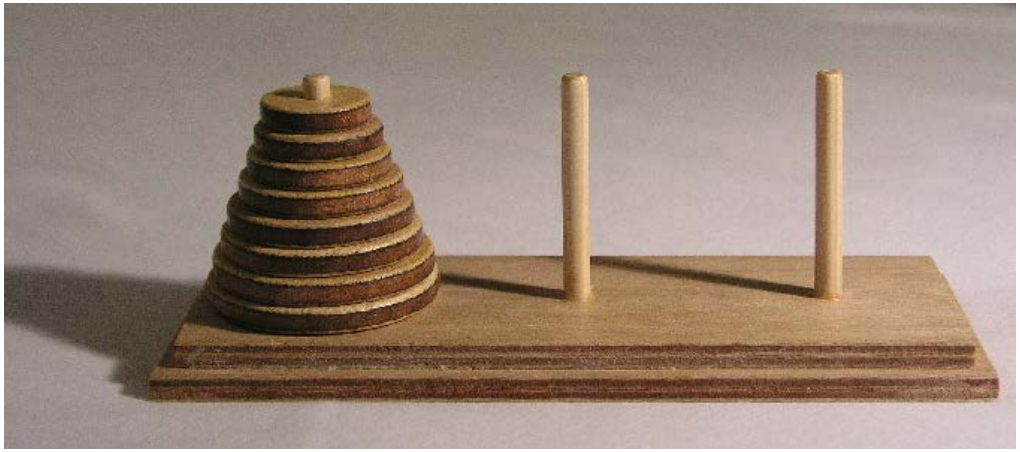
- There are three rods, named A, B, C, and n disks of different sizes which can be placed onto any rod.
- The puzzle starts with all n disks in a stack in ascending order of size on one rod, say A, so that the smallest is at the top (see figure).



(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:

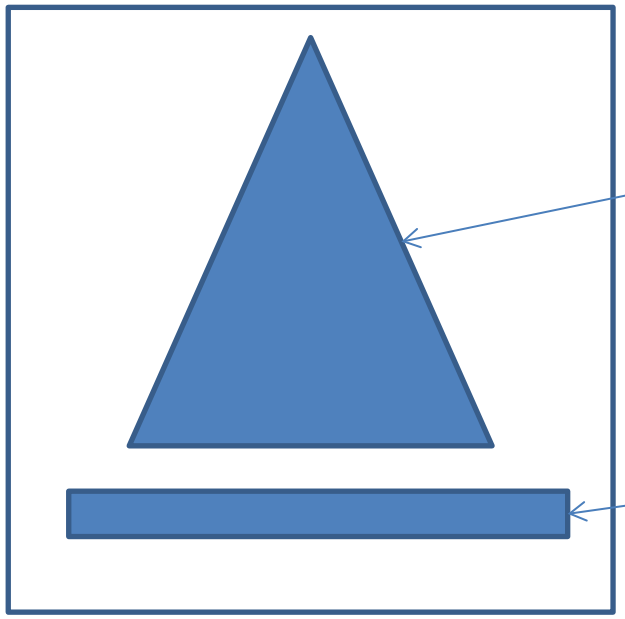
it is either empty,

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:



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]

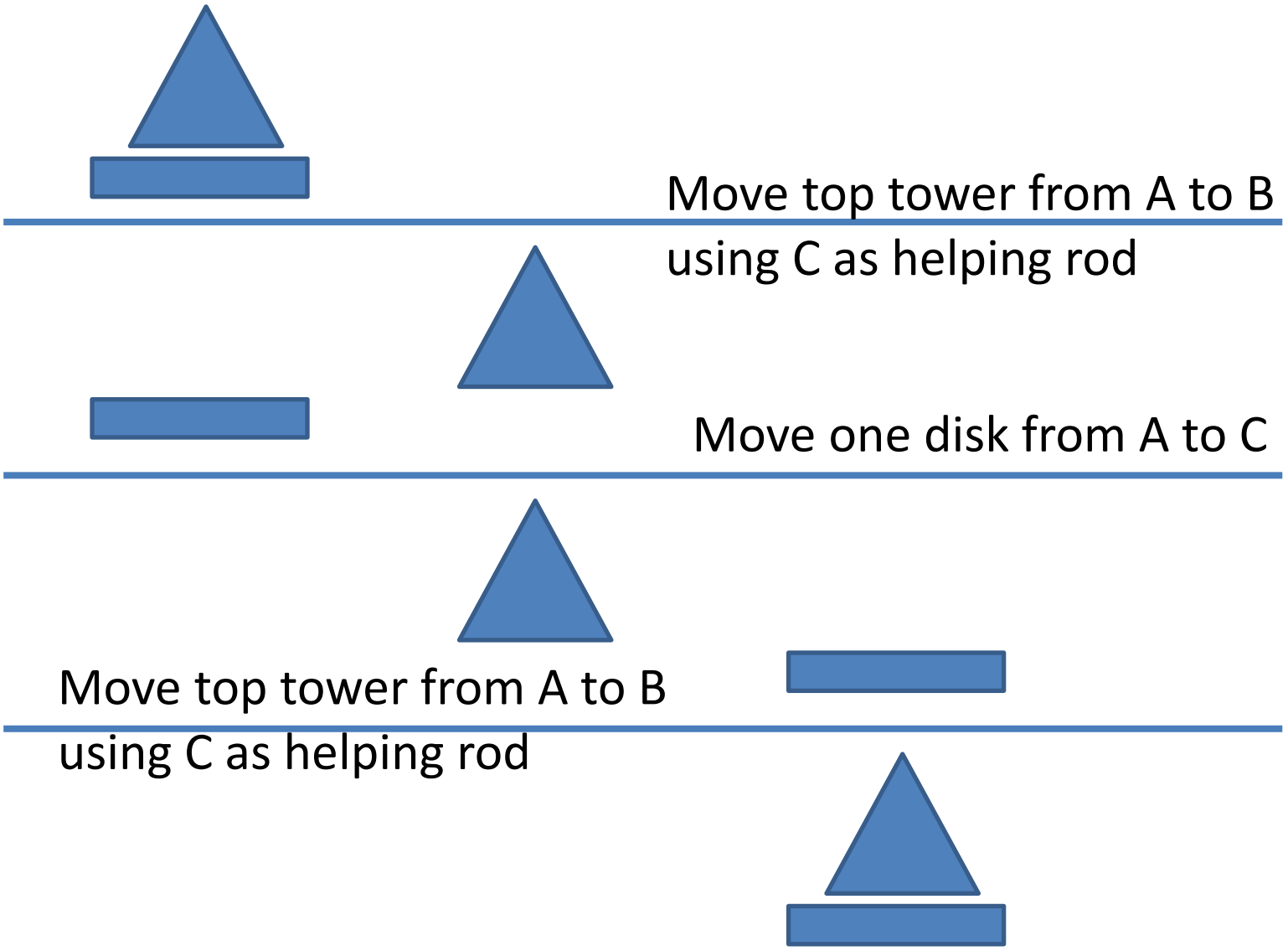
Towers of Hanoi: recursive algorithm

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: Python Code

- We write a function of four arguments,
  `HanoiTowers(start, via, target, n)`.
- The first three arguments are the three rods' "names", such as "A", "B", and "C". The last argument, `n`, is the number of discs.
- The function prints the moves, and does not return anything.

```
def HanoiTowers(start, via, target, n):
    if n>0:
        HanoiTowers(start, target, via, n-1)
        print("disk", n, "from", start, "to", target)
        HanoiTowers(via, start, target, n-1)
```

- Question: what is the base case here (it appears indirectly in the code).

Towers of Hanoi: Python Code

- 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).

```
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:
        HanoiTowers(start, target, via, n-1)
        print("disk", n, "from", start, "to", target)
        HanoiTowers(via, start, target, n-1)
```

Towers of Hanoi: Running the Code

```
>>> 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
```

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 recursion trees or 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, we can argue that $H(n) = 2^n - 1$ moves are required from any solution strategy. (Of course, more inefficient strategies do exist).
- Proof idea: will be explained in class.

Towers of Hanoi Nightmare

Suppose a monster demanded to know what the $3^{97} + 19$ 'th move in an n = 200 disk Towers of Hanoi puzzle is, or else . . . .

Having seen and even understood the material, you realize 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.

You eventually decide 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)
```

Towers of Hanoi Nightmare

To compute the k-th move in the an n disk Tower of Hanoi puzzle, we recall the solution of the Tower of Hanoi puzzle, and think recursively:

The solution to `HanoiTowers(start,via,target,n)` takes $2^n - 1$ steps altogether (so $1 \leq k \leq 2^n - 1$), and consists of three (unequal) parts.

- In the first part, which takes $2^{n-1} - 1$ steps, we move n-1 disks. If $1 \leq k \leq 2^{n-1} - 1$ the move we are looking for is within this part.

- In the second part, which takes exactly one step , we move disk number n. If $k = 2^{n-1}$ this is the move we want.

- In the last part, which again takes $2^{n-1} - 1$ steps, we again move n-1 disks. If $2^{n-1} + 1 \leq k \leq 2^n - 1$ the move is within this part, and is the $k - 2^{n-1}$ 'th move of this part.

Hanoi Monster - Code

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

Note the roles of the rods, as in the HanoiTowers function.

Recursive Monster Code: Executions

We first test it on some small cases, which can be verified by running the HanoiTowers program.

```
>>> 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'
```

Once we are satisfied with this, we solve the monster's question.

```
>>> 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 sequence of moves).

Binary Search

- We have already seen binary search and realized it is widely applicable (not only when monsters confront you). 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.

- The binary search idea is also known as lion in the desert idea.

And Now to Something Completely Different:

Munch!

Plan

- The game of Munch!
- Two person games and winning strategies.
- A recursive program (in Python, of course).
- A warning regarding running time.
- An existential proof that the first player has a winning strategy.

Game Theory

From Wikipedia:

- Game theory is the study of mathematical models of conflict and cooperation between intelligent rational decision-makers

- A game is one of perfect or full information if all players know the moves previously made by all other players.

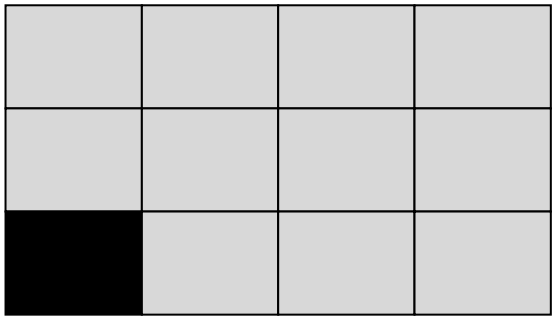
- In zero-sum games the total benefit to all players in the game, for every combination of strategies, always adds to zero (more informally, a player benefits only at the equal expense of others)

- Games, as studied by economists and real-world game players, are generally finished in finitely many moves

Munch!

Munch! is a two player, full information game. The game starts with a chocolate bar with n rows and m columns. Players alternate taking moves, where they chose a chocolate square that was not eaten yet, and munch all existing squares to the right and above the chosen square (including the chosen square).

The game ends when one of the players chooses and munches the lower left square. It so happens that the lower left corner is poisoned, so the player who made that move dies immediately, and consequently loses the game.

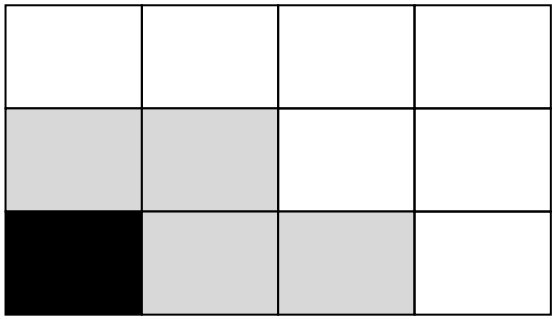


An image of a 3-by-4 chocolate bar (n=3, m=4). This configuration is compactly described by the list of heights [3,3,3,3]

Munch! (example cont.)

Munch! is a two player, full information game. The game starts with a chocolate bar with n rows and m columns. Players alternate taking moves, where they chose a chocolate square that was not eaten yet, and munch all existing squares to the right and above the chosen square (including the chosen square).

The game ends when one of the players chooses and munches the lower left square. It so happens that the lower left corner is poisoned, so the player who made that move dies immediately, and consequently loses the game.



An image of a possible configurations in the game. The white squares were already eaten. The configuration is described by the list of heights [2,2,1,0].

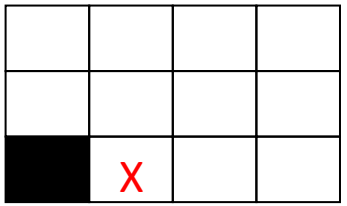
A possible Run of Munch!

[3,2,2,1] [3,2,0,0] [3,1,0,0] [1,1,0,0]

Suppose the game has reached the configuration on the left, [2,2,2,1], and it is now the turn of player 1 to move.
Player 1 munches the square marked with x, so the configuration becomes [2,2,0,0].

Player 2 munches the top rightmost existing square, so the configuration becomes [2,1,0,0].

Player 1 move leads to [1,1,0,0].
Player 2 move leads to [1,0,0,0].



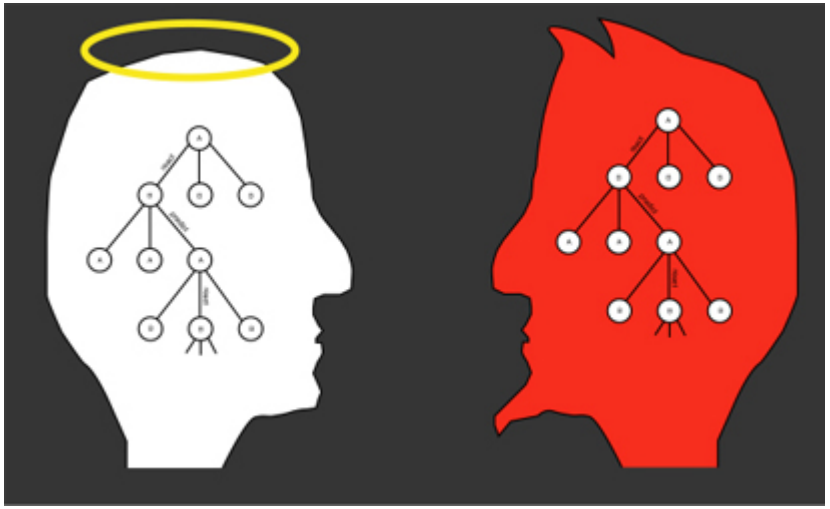
[1,0,0,0]

Player 1 must now munch the poisoned lower left corner, and consequently loses the game (in great pain and torment).

Two Player Full Information Games

A theorem from game theory states that in a finite, full information, two player, zero sum, deterministic game, either the first player or the second player has a winning strategy.

Unfortunately, finding such winning strategy is often computationally infeasible.



http://about.quickienomics.com/wp-content/uploads/2011/10/game-theory.jpg

Munch!: Winning and Losing Configurations

- Every configuration has fewer than n times m legal continuing configurations.
- A given configuration *C* is winning if it has (at least one) legal losing continuation *C'*. The player whose turn it is in *C* is rational, and thus will choose *C'* for its continuation, putting the opponent in a losing position
- A given configuration *C* is losing if all its legal continuations are winning. No matter what the player whose turn it is in *C* will choose, the continuation *C'* puts the opponent in a winning position.
- This defines a recursion, whose base case is the winning configuration [0,0,…,0].

The Initial Munch! Configuration is Winning

- We will show (on the board) that the initial configuration [n,n,…,n] of an n-by-m chocolate bar is a winning configuration for all n-by-m size chocolate bars (provided the bar has at least 2 squares).

- This implies that player 1 has a winning strategy.

- Interestingly, our proof is purely existential. We show such winning strategy exists, but do not have a clue on what it is (e.g. what should player 1 munch so that the second configuration will be a losing one?).

Munch! Code (recursive)

```
def win(n, m, hlst, show=False):
    ''' determines if in a given configuration, represented by hlst,
    in an n-by-m board, the player who makes the current move has a
    winning strategy. If show is True and the configuration is a win,
    the chosen new configuration is printed.'''
    assert n>0 and m>0 and min(hlst)>=0 and max(hlst)<=n and \
               len(hlst)==m
    if sum(hlst)==0:  # base case: winning configuration
        return True
    for i in range(m):  # for every column, i
        for j in range(hlst[i]): # for every possible move, (i,j)
            move_hlst = [n]*i+[j]*(m-i)
            # full height up to i, height j onwards
            new_hlst = [min(hlst[i], move_hlst[i]) for i in range(m)]
            # munching
            if not win(n, m, new_hlst):
                if show:
                    print(new_hlst)
                return True
    return False
```

Running the Munch! code

```
>>> win(5,3,[5,5,5],show=True)
[5, 5, 3]
True
>>> win(5,3,[5,5,3],show=True)
False
>>> win(5,3,[5,5,2],show=True)
[5, 3, 2]
True
>>> win(5,3,[5,5,1],show=True)
[2, 2, 1]
True
>>> win(5,5,[5,5,5,5,5],True)
[5, 1, 1, 1, 1]
True
>>> win(6,6,[6,1,1,1,1,1],show=True)
False
```

Implementing Munch! in Python

- A good sanity check for your code is verifying that [n,n,…,n] is indeed a winning configuration.
- Another sanity check is that in an n-by-n bar, the configuration [n,1,…,1] is a losing configuration (why?)
- This recursive implementation will be able to handle only very small values of n,m (in, say, one minute).
  - ❑ Can memoization help here?

Recursive Formulae of Algorithms Seen in our Course

דוגמא	פעולות מעבר לרקורסיה	קריאות רקורסיביות	נוסחת נסיגה	סיבוכיות
max1 (מהתרגול), עצרת	1	N-1	T(N)=1+T(N-1)	O(N)
חיפוש בינארי	1	N/2	T(N)=1+T(N/2)	O(log N)
Quicksort (worst case)	N	N-1	T(N)=N+ T(N-1)	$O(N^2)$
Mergesort Quicksort (best case)	N	N/2 ,N/2	T(N)=N+2T(N/2)	O(N log N)
חיפוש בינארי עם slicing	N	N/2	T(N)=N+T(N/2)	O(N)
max2 (מהתרגול)	1	N/2 ,N/2	T(N)=1+2T(N/2)	O(N)
האנוי	1	N-1, N-1	T(N)=1+2T(N-1)	$O(2^N)$
פיבונאצ'י	1	N-1, N-2	T(N)=1+T(N-1)+T(N-2)	$O(2^N)$ (לא הדוק)

Last words (not for the Soft At Heart): the Ackermann Function (for reference only)

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 what is known as a 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 (for reference only)

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

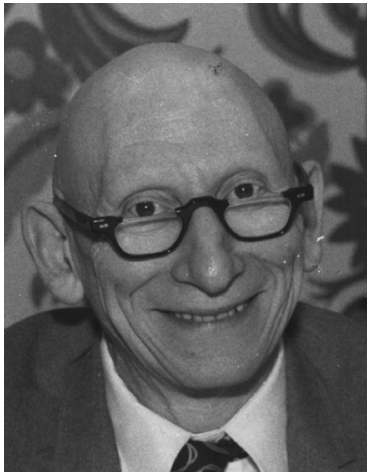
```
def ackermann(m,n):
    if m==0:
       return n+1
    elif m>0 and n==0:
       return ackermann(m-1,1)
    else:
       return ackermann(m-1, ackermann(m,n-1))
```

However, running it with m ≥ 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

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.



Picture from a web Page by Paolo Alessandrini

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''.