Extended Introduction to Computer Science CS1001.py

Lecture 12:
Recursion, Continued

Instructors: Daniel Deutch, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Ben Bogin, Noam Parzanchevski

School of Computer Science
Tel-Aviv University
Fall Semester 2018-9
http://tau-cs1001-py.wikidot.com

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 "naïve recursion" is highly convenient for writing code may lead to highly inefficient run times. For this reason, we will also introduce a technique to improve recursive algorithms. 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):
    if n<=1:
        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:

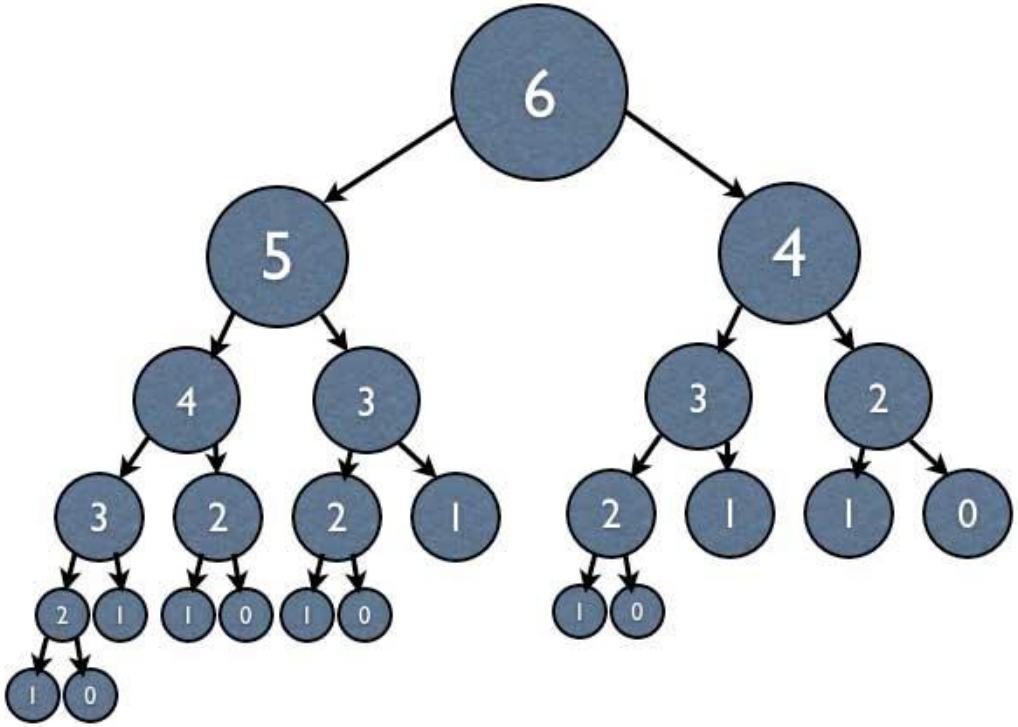
Computing Fibonacci Numbers

- 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 !!
```

Recursion Trees (reminder)

- Recursion trees are a common visual representation of a recursive process. For example, here is the recursion tree for fibonacci, for n=6:



Recursion Trees

Typically the order of calls from each node is from left to right.

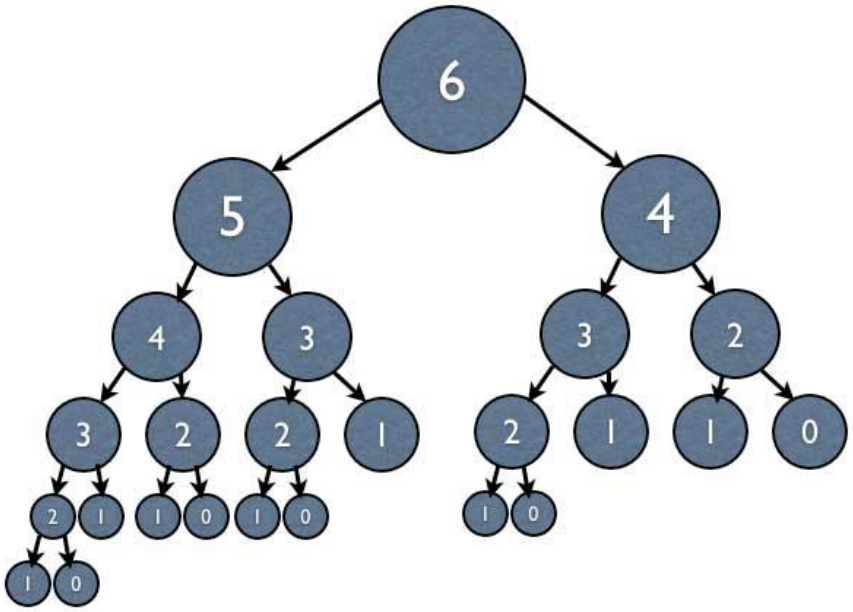
Two special types of nodes in the tree are the root of the tree (a node without a "parent"), and a leaf (a node without children). There is exactly one root, but there can be many leaves.

Recursion trees enable a better understanding of the recursive process, complexity analyses, and may help designing recursive solutions. Two important notions in this context are:

- Time complexity: the total amount of time spent in the whole tree.
- Recursion depth: the maximal length of a path from root to leaf. This is also the maximal number of recursive calls that are simultaneously open.

Recursion Trees

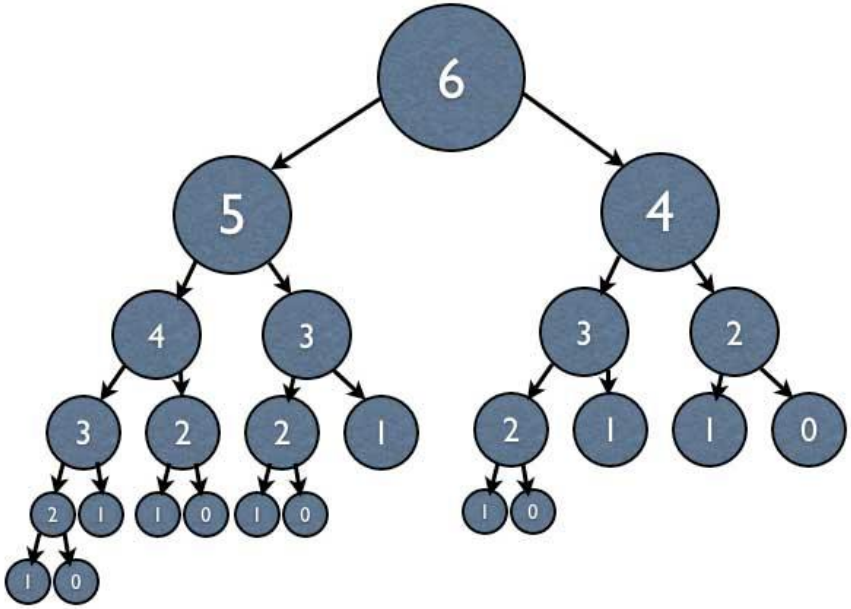
- Recursion depth – distance from root to deepest leaf
  - corresponds to maximal number of recursive calls that are open simultaneously
  - too many such calls may cause memory overload (will discuss this soon)



- depth of this tree = 5
- for input n the depth is n-1 = O(n)

Recursion Trees

- Time complexity – overall number of operations in the whole tree



- each node here takes O(1) ops
- $< 2^n$ nodes
- thus time complexity is $O(2^n)$ (not tight)
- tightest bound is $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ (without proof)

Exact Number of Calls

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

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

Count(n) 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: $count(n) = 2 \cdot Fibonacci(n) - 1$

Intuition for Improving Efficiency

- 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, this technique is often used as part of dynamic programming.

Fibonacci: Recursive Code with Memoization

- We will use a dictionary - an indexed data structure that can grow dynamically. This dictionary, which we name fib_dict, will contain the Fibonacci numbers already computed.
- We initialize the dictionary with fib_dict = {0:1, 1:1}.
- fibonacci2 is an envelope function, which calls the recursive fib2.

```
def fibonacci2(n):
    """ Envelope function for Fibonacci,
        employing memoization in a dictionary """
    fib_dict = {0:1, 1:1} # initial dictionary
    return fib2(n, fib_dict)

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

Fibonacci: Recursive Code with Memoization

- This small change implies a huge performance difference:

```
>>> elapsed("fibonacci(35)")
8.245295905878606
>>> elapsed("fibonacci2(35)")
9.053997947143898e-05


>>> elapsed("fibonacci(40)")
92.6057921749773
>>> elapsed("fibonacci2(40)")
9.909589798251517e-05


>>> elapsed("fibonacci(45)")
1031.308921394449 # 17 minutes !!
>>> elapsed("fibonacci2(45)")
0.00011015042046658152
```

Recursive Code with Memoization

- To understand how the dictionary is filled with values, let's call the recursive function directly:

```
>>> fib_dict = {0:1, 1:1}
>>> fib2(5, fib_dict)
8
>>> fib_dict
{0: 1, 1: 1, 2: 2, 3: 3, 4: 5, 5: 8}

>>> fib2(7, fib_dict)
21
>>> fib_dict
{0: 1, 1: 1, 2: 2, 3: 3, 4: 5, 5: 8, 6: 13, 7: 21}
```

- Here, computations from the first call were re-used in the second one, because the dictionary was not reset between calls.

Recursive Code with Memoization

- Here, computations from the first call were re-used in the second one, because the dictionary was not re-set between calls.

```
>>> fib_dict = {0:1, 1:1}

>>> elapsed("fib2(500, fib_dict)")
0.0004189785626902008

>>> elapsed("fib2(500, fib_dict)")
3.811481833038144e-05

>>> elapsed("fib2(1000, fib_dict)")
0.0003692017477066045

>>> elapsed("fib2(1200, fib_dict)")
0.0001581480521757328
```

No recursion at all here!
Dictionary contains result.

Recursion depth is only 500

Recursion depth is only 200

Recursion Tree of Code with Memoization

- How does the recursion tree for the code with memoization look like?
- What is its depth?
- What is the time complexity of the function?

- In class, on board.

Pushing Recursion Depth to the Limit

```
>>> fibonacci2(990)
571829406815633979529643697006273045106845980748991112071
673038743714031497887739023091610769764627307772654802298
784361803421747114571265690519449915873452164193174293407
940201977897716937097604164288130909

>>> 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?
- fib2(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" (memory environment) to the current one. 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).

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
```

Reversed Order of Calls

- As you have probably understood, Python evaluates expressions from left to right (except for when otherwise dictated by precedence of operators).
- Suppose we changed the order of calls inside fibonacci2: first we call n-2, then n-1.

```
def fib2_reverse(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]
```

- How does the recursion tree look like now? Recursion depth? Time complexity?

Reversed Order of Calls

- Memory-wise, We can now compute fibonacci up to values (about) twice as large.

```
>>> fib2_reverse(1986, {0:1,1:1})

810879226400948914184387400615591130555006766044979008432292945887556953730886091189925211916635495449333513004270234914537461284901292446908897559765539197111540858390529008105883495154321496723659980313256517069425635895034310808290391018959976093003404672302491032780932996949438063515061599695216331083525339223853907721207136862784763667719397907746593484736384718791709517060157571485740910441995359728761573 3
```

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).
  - 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

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
```

- As we mentioned already, recursive calls require maintenance operations and memory allocation ("frames"), thus tend to have a negative influence on running time, compared to the analogous iterative solution.

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 then fibb[6]. But then we make no further use of fibb[4]. It just lies, basking happily, in the memory.

- 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, prev and curr. 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 Solution: Python Code

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

```
>>> for i in range(0,7):  # sanity check
       print(fibonacci4(i))
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(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 (disregarding the size of the numbers themselves).

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 or other methods (studied in the discrete mathematics course).

```
def closed_fib(n):
    return round(((1+5**0.5)**(n+1)-(1-5**0.5)**(n+1))/(2**(n+1)*5**0.5))
```

Closed Form Formula: Code, and Danger

```
def closed_fib(n):
    return round(((1+5**0.5)**(n+1)-(1-5**0.5)**(n+1))/(2**(n+1)*5**0.5))

# sanity check
>>> for i in range(10, 60, 10):
        print(i, fibonacci4(i), closed_fib(i))

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:

Closed Form Formula: Code, and Danger

- 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.

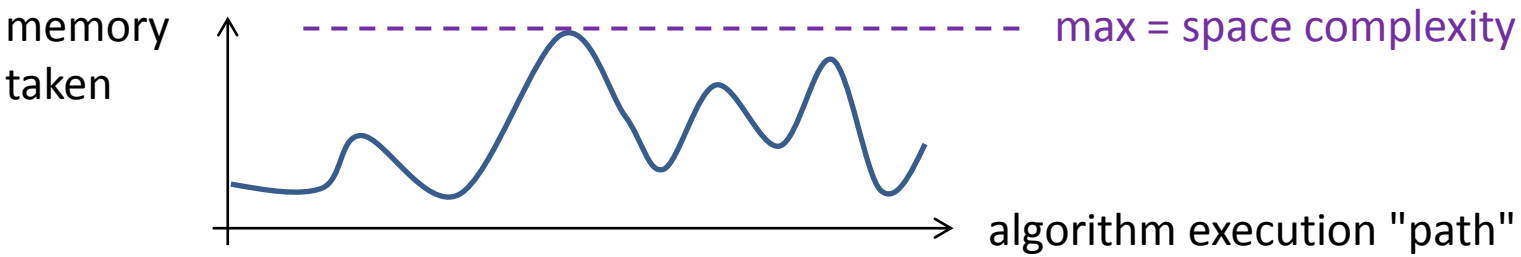
- Consider quicksort as a specific example. In any specific execution, we never call quicksort on the same set of elements more than once (think why this is true).

- 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.

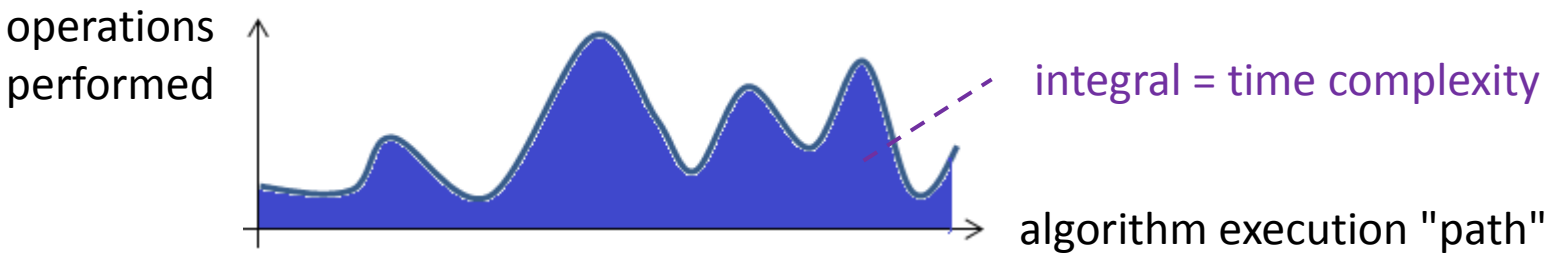
- Even if these techniques are applicable, the transformation is often not automatic, and if we deal with small instances where performance is not an issue, such optimization may be a waste of effort.

A word about space (memory) complexity

- A measure of how much memory cells the algorithm needs
  - not including memory allocated for the input of the algorithm
- This is the maximal amount of memory needed at any time point during the algorithm's execution



- Compare to time complexity, which relates to the cumulative amount of operations made along the algorithm's execution



Space (memory) Complexity

- Recursion depth has an implication on the space (memory) complexity, as each recursive call required opening a new environment in memory.

- In this course, we will not require space complexity analysis of recursive algorithms, but you are expected to analyze recursion depth and understand its affects.

- We do require understanding of space allocation requirements in basic scenarios such as:
  - copying (parts of) the input
  - list / string slicing
  - using + operator for lists (as opposed to += or lst.append)

etc.

Munch!

- The game of Munch!
- Two person games and winning strategies.
- A recursive program (in Python, of course).

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.

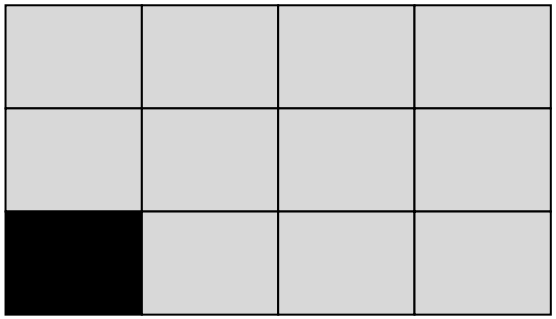
- In zero-sum games the total benefit to all players in the game, for every combination of strategies, always adds to zero (more informally, a player benefits only at the equal expense of others)

- 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.

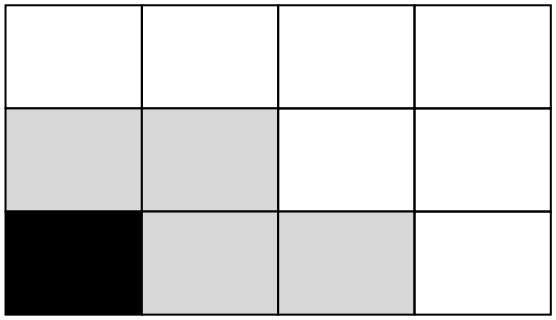


An image of a 3-by-4 chocolate bar (n=3, m=4). This configuration is compactly described by the list of heights [3,3,3,3]

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].

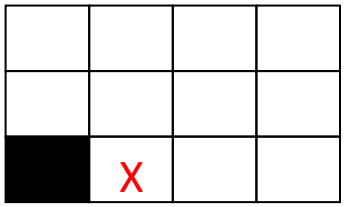
A possible Run of Munch!

[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].



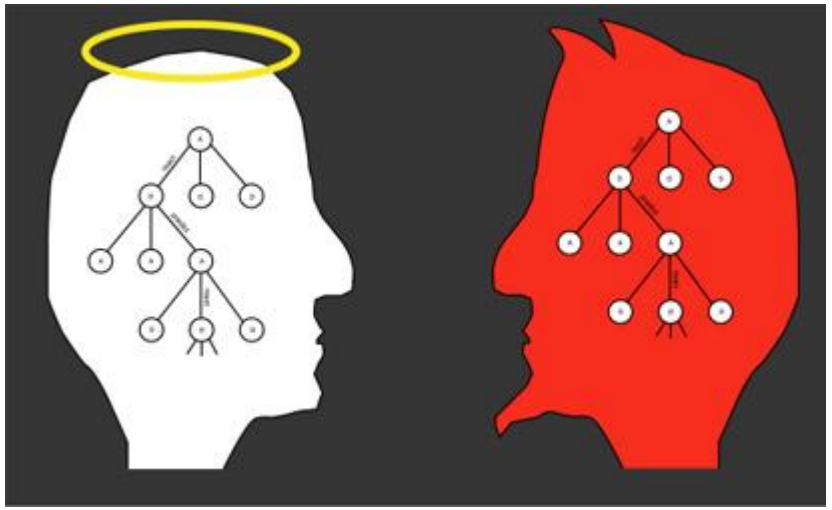
[1,0,0,0]

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].

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
```

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.

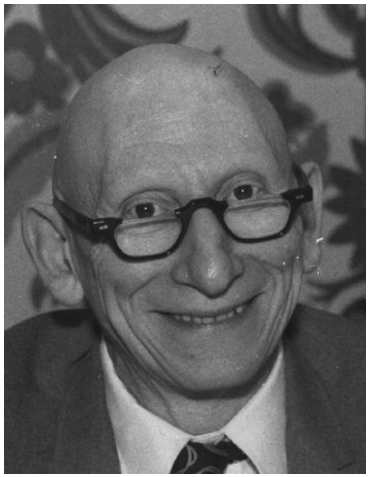
```
def ackermann(m,n):
    if m==0:
        return n+1
    elif m>0 and n==0:
        return ackermann(m-1,1)
    else:
        return ackermann(m-1, ackermann(m,n-1))
```

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

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''.