Extended Introduction to Computer Science CS1001.py

Lecture 13: Recursion (4) - Munch!

Instructors: Elhanan Borenstein, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Noam Parzanchevsky, Ori Zviran

School of Computer Science
Tel-Aviv University
Spring Semester 2019-20
http://tau-cs1001-py.wikidot.com

Lectures 10-12: Overview

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

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)

Lectures 13a: Plan

- The game of Munch!
  - Two person games and winning strategies.
  - A recursive program (in Python, of course).
  - 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 perfect or full information game – when all players know the moves previously made by all other players

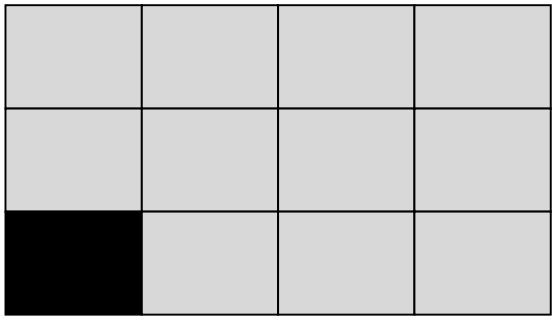
- In zero-sum games the total benefit to all players in the game, for every combination of strategies, always adds to zero (more informally, a player benefits only at the equal expense of others)

- 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.

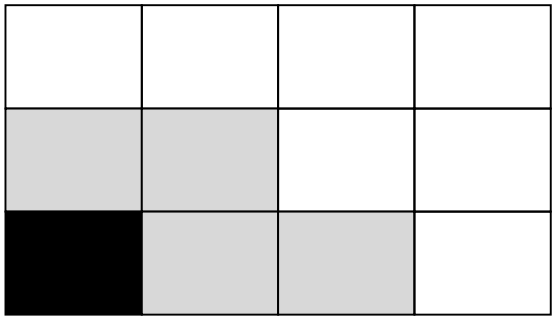


An image of a 3-by-4 chocolate bar (n=3, m=4). This configuration is compactly described by the list of heights [3,3,3,3]

Munch! (example cont.)

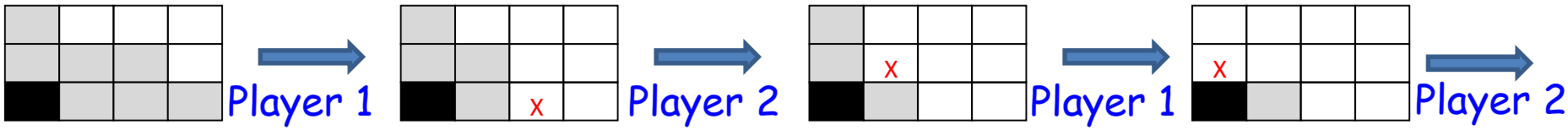
Munch! is a two player, full information game. The game starts with a chocolate bar with n rows and m columns. Players alternate taking moves, where they chose a chocolate square that was not eaten yet, and munch all existing squares to the right and above the chosen square (including the chosen square).

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!



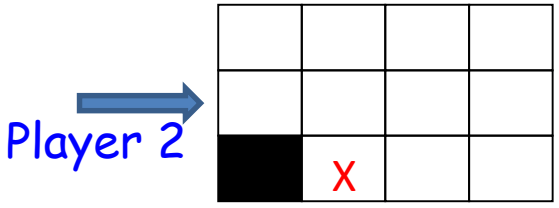
[3,2,2,1] [3,2,0,0] [3,1,0,0] [1,1,0,0]

Suppose the game has reached the configuration on the left, [3,2,2,1], and it is now the turn of player 1 to move.
Player 1 munches the square marked with X, so the configuration becomes [3,2,0,0].

Player 2 munches the top rightmost existing square, so the configuration becomes [3,1,0,0].

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

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



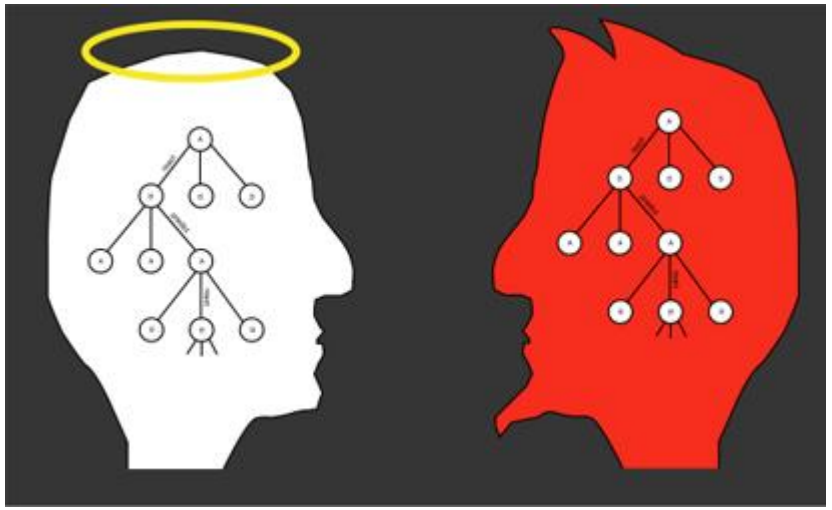
[1,0,0,0]

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 $\leq n \cdot m$ 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 win-able position.
- This defines a recursion, whose base case is the winning configuration [0,0,…,0] (alternatively [1,0,…,0] is losing).

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
```

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?

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
```

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.

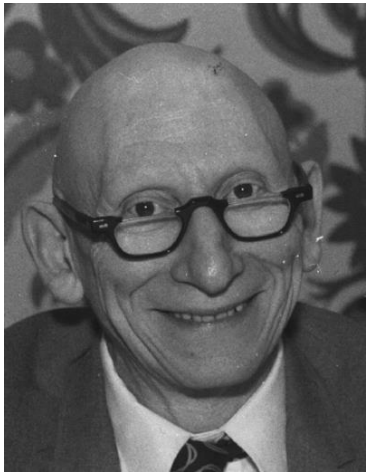
```
def ackermann(m,n):
    if m==0:
        return n+1
    elif m>0 and n==0:
        return ackermann(m-1,1)
    else:
        return ackermann(m-1, ackermann(m,n-1))
```

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''.