

Extended Introduction to Computer Science

CS1001.py

Lecture 13a: Recursion (4) - Memoization, Munch!

Instructors: Jonathan Berant, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Noam Parzanchevski, Ben Bogin

School of Computer Science

Tel-Aviv University

Spring Semester 2019

<http://tau-cs1001-py.wikidot.com>

Memoization (Reminder)

- We saw a recursive but inefficient method for computing the **n'th** Fibonacci number:

```
def fibonacci(n):  
    if n<=1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

```
>>> elapsed("fibonacci(45)")  
432.662887 # over 7 minutes !!
```

Memoization (Reminder)

- We improved this with memoization: trading time for space by storing values that are computed many times:

```
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]
```

```
>>> elapsed("fibonacci2(45)")  
0.00011015042046658152
```

Memoization (Reminder)

- We further improved the maximal Fibonacci number we can compute given a fixed recursion depth, by better controlling the recursion tree:

```
def fib2_rev(n, fib_dict):  
    if n not in fib_dict:  
        res = fib2_rev(n-2, fib_dict) + \  
              fib2_rev(n-1, fib_dict)  
        fib_dict[n] = res  
    return fib_dict[n]
```

```
>>> fib2_rev(1001, {0:1,1:1})
```

```
11379692539836027225752378255222417557274593035373051314  
50866341766910925361459854701461293346418669027836730423  
22088625863396052888690096969577173696370562180400527049  
497109023054114771394568040040412172632376
```

Memoization – iterative version (Reminder)

```
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]
```

```
>>> elapsed("fibonacci2(2000)")  
0.003454221497536104  
>>> elapsed("fibonacci3(2000)")  
0.0008148609599825107
```

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 $\text{fibb}[4] = \text{fibb}[2] + \text{fibb}[3]$. This entry will subsequently be used to determine $\text{fibb}[5]$ and then $\text{fibb}[6]$. But then we make no further use of $\text{fibb}[4]$. 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, `previous` and `current`. Every iteration, those will be updated. Normally, we would need a third variable `next` for keeping a value temporarily. However Python supports the "simultaneous" assignment of multiple variables (first the right hand side is evaluated, then the left hand side is assigned).

Iterative Fibonacci Solution: Python Code

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

```
>>> for i in range(0,7): # 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)
6.1507589999999999
>>> 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.

Recursive Formulae of Algorithms Seen in our Course

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

Munch!

- 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

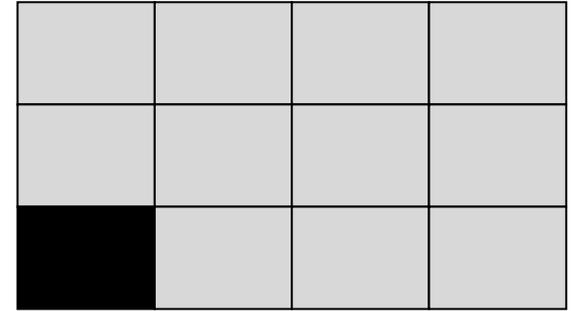
From Wikipedia:

- **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 choose 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**.

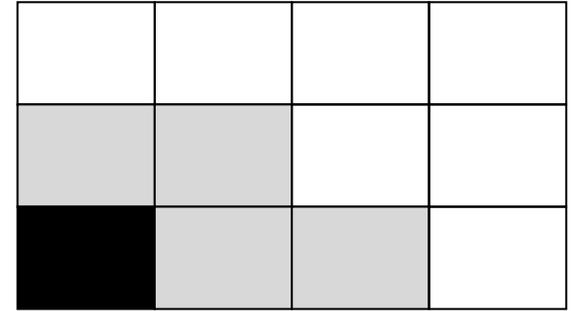


An image of a 3-by-4 chocolate bar ($n=3$, $m=4$). This configuration is compactly described by the list of heights $[3,3,3,3]$

Munch! (example cont.)

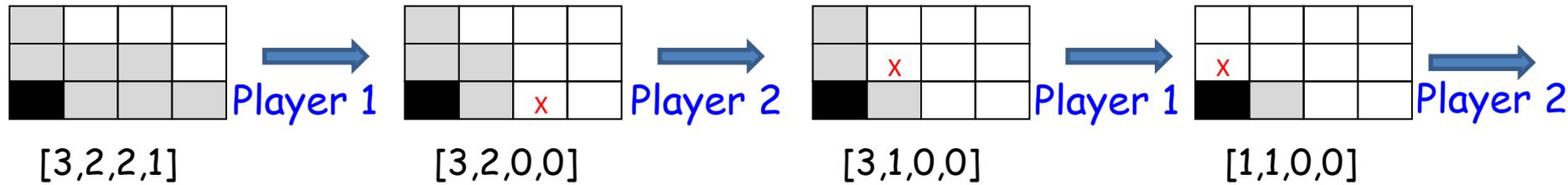
Munch! is a **two player**, full information **game**. The game starts with a chocolate bar with n rows and m columns. Players alternate taking moves, where they choose a chocolate square that was not eaten yet, and **munch** all existing squares to the right and above the chosen square (including the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



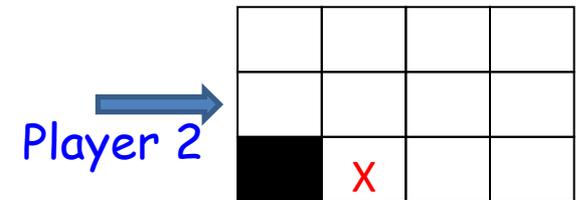
Suppose the game has reached the configuration on the left, [3,2,2,1], and it is now the turn of player 1 to move.

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].

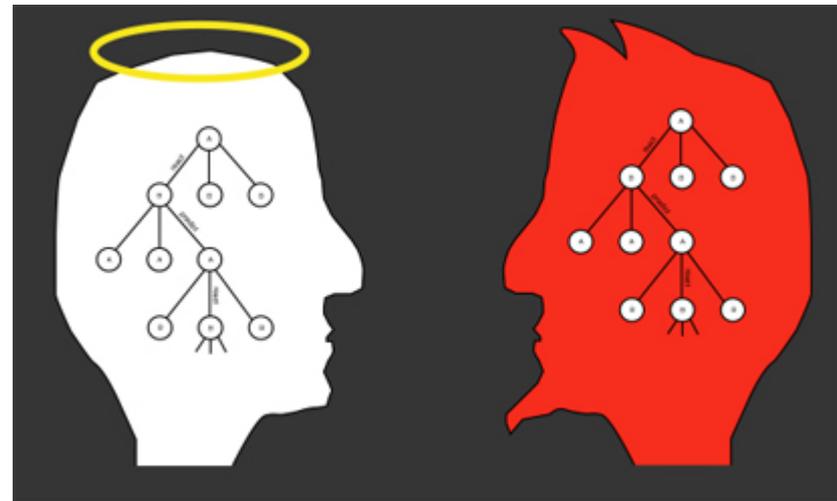


Player 1 must now munch the poisoned lower left corner, and consequently **loses the game (in great pain and torment)**. [1,0,0,0]

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**.



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 $[0,0,\dots,0]$ (alternatively $[1,0,\dots,0]$ is losing).

The Initial **Munch!** Configuration is **Winning**

- We will show (on the board) that the initial configuration $[n, n, \dots, n]$ 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 $[n, n, \dots, n]$ is indeed a **winning configuration**.
- Another sanity check is that in an n -by- n bar, the configuration $[n, 1, \dots, 1]$ 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 $m \geq 4$ 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.

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".