Principles of Computer Science II

Algorithms for BioInformatics

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 5

Pebble Game

- Game played in turns by 2 players.
- Two piles of equal number of pebbles.
- Each turn a player may either
  - take 1 pebble **from a single pile**, or
  - take 1 pebble **from both piles**.
- The player that takes the last pebble wins.

Best Strategy for Winning the Pebble Game

- Does the first player always have an advantage?
- Let's consider the most simplified version.
  - Pebbles $= 2$ – we call this the 2×2 game.
  - Is there a winning strategy?
  - What is the winning strategy?

Generaled Strategy for Winning the Pebble Game

- Can we generalize the strategy of the 2×2 game?
- What about the 3×3 game?
  - Consider different game sequences.
- Consider the $n \times n$ game.
  - Is there only one winning strategy?
  - How easy it is to describe our strategy?
  - Quality of solution.

We build a matrix for all game combinations. Four actions:

1. $\uparrow$ take one pebble from pile A.
2. $\leftarrow$ take one pebble from pile B.
3. $\nwarrow$ take one pebble from each pile.
4. * ignore game.

	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											

- The first player always loses the 2×2.
- Clearly also for 0×2, 0×4, ...
- Can we generalize for all games where each pile has an even number of pebbles?

	0	1	2	3	4	5	6	7	8	9	10
0	*		*		*		*		*		*
1											
2	*		*								
3											
4	*										
5											
6	*										
7											
8	*										
9											
10	*										

- The first player always loses the 2×2.
- Clearly also for 0×2, 0×4, ...
- Can we generalize for all games where each pile has an even number of pebbles?

	0	1	2	3	4	5	6	7	8	9	10
0	*		*		*		*		*		*
1											
2	*		*		*		*		*		*
3											
4	*		*		*		*		*		*
5											
6	*		*		*		*		*		*
7											
8	*		*		*		*		*		*
9											
10	*		*		*		*		*		*

- Only 1 option for all 0×1, 0×3, ... and 1×0, 3×0, ...

	0	1	2	3	4	5	6	7	8	9	10
0	*		*		*		*		*		*
1											
2	*		*		*		*		*		*
3											
4	*		*		*		*		*		*
5											
6	*		*		*		*		*		*
7											
8	*		*		*		*		*		*
9											
10	*		*		*		*		*		*

- Only 1 option for all 0×1, 0×3, ... and 1×0, 3×0, ...

	0	1	2	3	4	5	6	7	8	9	10
0	*	←	*	←	*	←	*	←	*	←	*
1	↑										
2	*		*		*		*		*		*
3	↑										
4	*		*		*		*		*		*
5	↑										
6	*		*		*		*		*		*
7	↑										
8	*		*		*		*		*		*
9	↑										
10	*		*		*		*		*		*

- Only 1 option for all 0×1, 0×3, ... and 1×0, 3×0, ...
- Can we generalize for other columns/rows where one pile has an odd number of pebbles and the other an even?

	0	1	2	3	4	5	6	7	8	9	10
0	*	←	*	←	*	←	*	←	*	←	*
1	↑		↑		↑		↑		↑		↑
2	*	←	*	←	*	←	*	←	*	←	*
3	↑		↑		↑		↑		↑		↑
4	*	←	*	←	*	←	*	←	*	←	*
5	↑		↑		↑		↑		↑		↑
6	*	←	*	←	*	←	*	←	*	←	*
7	↑		↑		↑		↑		↑		↑
8	*	←	*	←	*	←	*	←	*	←	*
9	↑		↑		↑		↑		↑		↑
10	*	←	*	←	*	←	*	←	*	←	*

- Only 1 option for all 0×1, 0×3, ... and 1×0, 3×0, ...
- Can we generalize for other columns/rows where one pile has an odd number of pebbles and the other an even?
- What about the other rows/columns?

	0	1	2	3	4	5	6	7	8	9	10
0	*	←	*	←	*	←	*	←	*	←	*
1	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
2	*	←	*	←	*	←	*	←	*	←	*
3	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
4	*	←	*	←	*	←	*	←	*	←	*
5	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
6	*	←	*	←	*	←	*	←	*	←	*
7	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
8	*	←	*	←	*	←	*	←	*	←	*
9	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
10	*	←	*	←	*	←	*	←	*	←	*

An algorithmic approach for winning the Pebble Game

- How can we build the matrix for any game size, e.g., 20×20
- What is the algorithm for winning the game?

An algorithmic approach for winning the Pebble Game

- How can we build the matrix for any game size, e.g., 20×20
- What is the algorithm for winning the game?
- Why should I care?

An algorithmic approach for winning the Pebble Game

- How can we build the matrix for any game size, e.g., 20×20
- What is the algorithm for winning the game?
- Why should I care?
- It is the sequence alignment problem.
- The computational idea used to solve both problems is the same.
- We need to understand how algorithms work.

Methodology of solving a computational problem

- What is the problem at hand ?
  - Identify & Understand assumptions.
  - What is our goal ?
  - Identify similar problems/solutions in the bibliography
  - What are the theoretical foundation ?
  - Can we formulate the problem in a unambiguous and precise way ?
- What is the Input that we have ?
  - Do we have enough data or should we try to collect?
  - Open data sets ?
  - Can we synthesize input data ?
- What is the expected Output ?

Solution Sketch

- Do we have a rough idea of a solution ?
- Do we have identified an approach to solving the problem ?
  - think again !
  - go through the definition – maybe we overlooked something ?
- Write down a solution sketch
  - check if it adheres to the initial assumptions
  - can you try it out with a small input ?
- Is the solution correct ? can we provide some arguments ?
- What is the performance of the solution ?
- Can we think of a more efficient solution ?

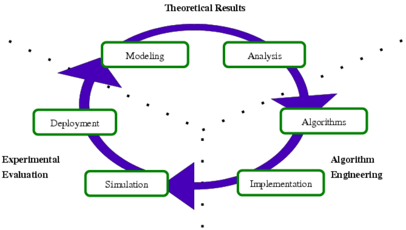
Implement the first version

- Pick your programming language of choice.
- Implement your solution
  - No need to try to make it elegant / fast.
  - Remember Donalt Knuth: There is no such thing as early optimization.
- Get some input data
  - Open datasets
  - Small size
- Limited Evaluation
  - does it work ?
  - do you need to make any modifications ?
  - are there special cases that you missed ?

Iterative approach

- Step-by-step development
  - Continuous development.
  - Agile methodology.
- Identify issues in previous version
  - Code beautification.
  - Bug fixes.
  - Performance improvements.
  - Additional functionalities.
- Implement improvements
  - Make sure code is always clean + easy to maintain.
  - Keep detailed records of changes.
  - Always keep history of source code evolution.
- Performance Evaluation
  - bigger input.
  - scalability ?

Theoretical – Practical Approach Cycle



Quality of Code

John Woods

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

Recursion Coding Style

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function a recursive function.

Termination condition:

- A recursive function has to terminate to be used in a program.
- A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.
- A base case is a case, where the problem can be solved without further recursion.

Factorial Computation: Using Iteration

```
def iterative_factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result
```

Factorial Computation: Using Recursion

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Factorial Computation

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "):
        return res

print(factorial(5))
```

Fibonacci Numbers

The Fibonacci numbers are defined by:
$F_n = F_{n-1} + F_{n-2}$
where $F_0 = 0$ and $F_1 = 1$

- 0,1,1,2,3,5,8,13,21,34,55,89, ...

Factorial Computation: Using Recursion

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Factorial Computation: Using Iteration

```
def fibi(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Measure Performance

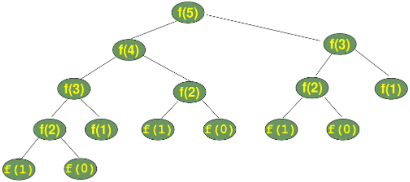
```
import time

for i in range(1,41):
        t1 = time.perf_counter()
        s = fib(i)
        t2 = time.perf_counter() - t1

        t3 = time.perf_counter()
        s = fibi(i)
        t4 = time.perf_counter() - t3

        print(f"n={i}, fib: {t2:.2f}, fibi: {t1:.2f}, percent: {t2/t4:.
```

Fibonacci Numbers



Factorial Computation: Using Recursion and Memory

```
memo = {0:0, 1:1}
def fibm(n):
    if not n in memo:
        memo[n] = fibm(n-1) + fibm(n-2)
    return memo[n]
```