LECTURE 14

RECURSION VS ITERATION II

MCS 275 Spring 2022
Emily Dumas

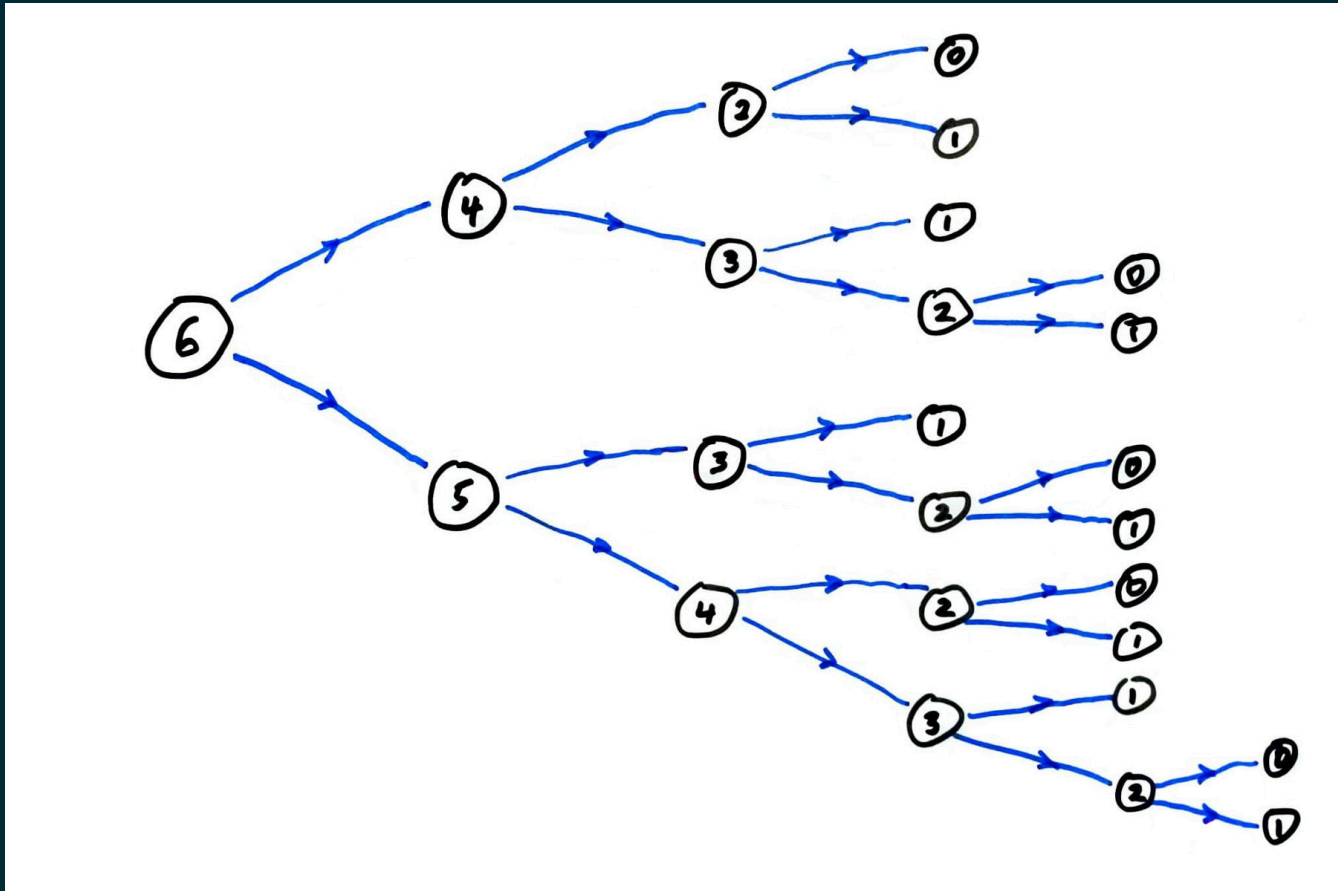# LECTURE 14: RECURSION VS ITERATION II

Course bulletins:

- Project 1 graded.

- Feedback survey open.

- Project 2 description coming Monday.

- Project 2 due 6pm CST Friday, February 25.

- Remember to check the recursion sample code.

FIBONACCI TIMING

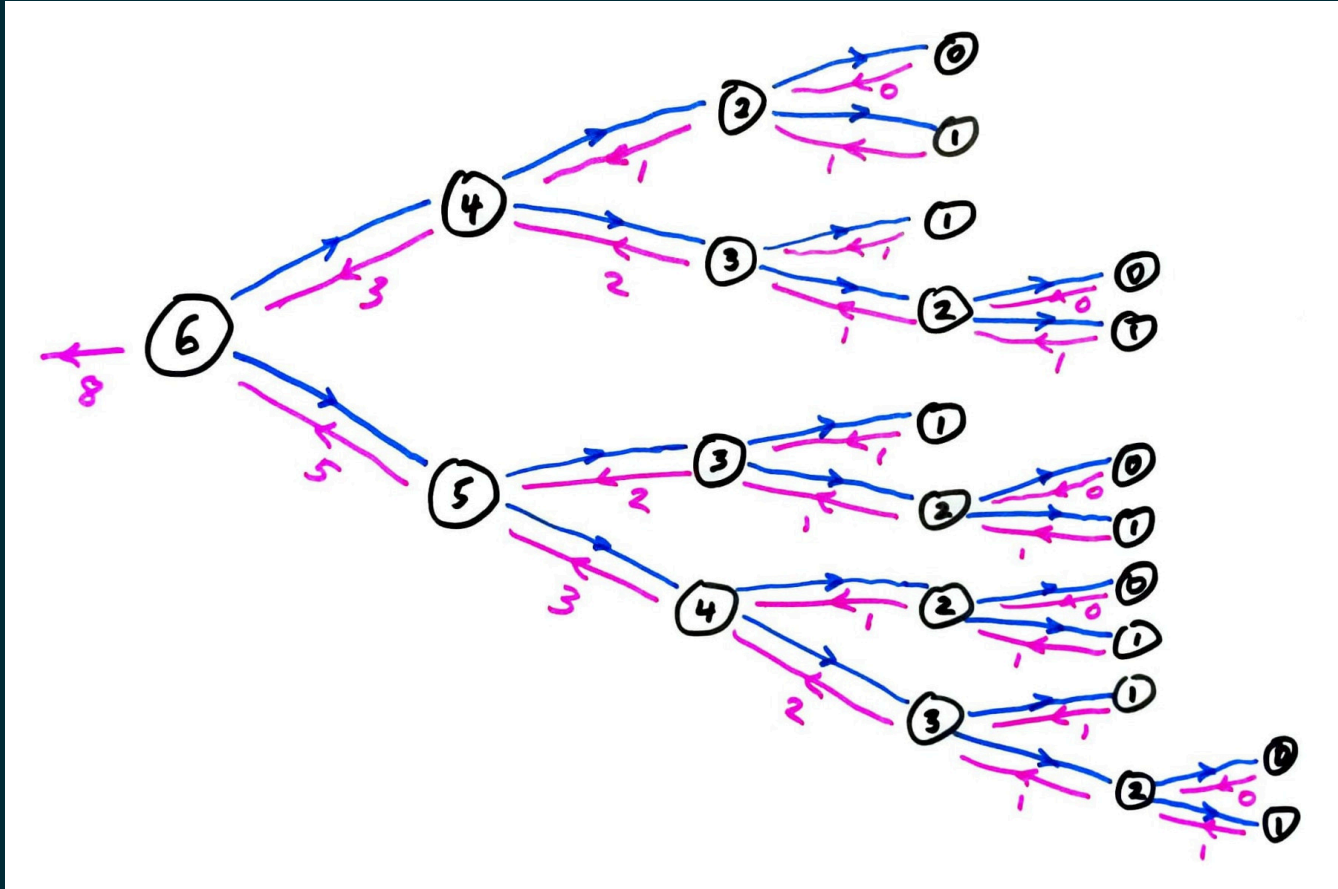|	n=35
recursive	1.9s
iterative	<0.001s

Measured on a 4.00Ghz Intel i7-6700K CPU (2015 release date) with Python 3.8.5

FIB CALL GRAPH

Most Fibonacci numbers are computed many times!

FIB CALL GRAPH

Most Fibonacci numbers are computed many times!

MEMOIZATION

`fib` computes the same terms over and over again.

Instead, let's store all previously computed results, and use the stored ones whenever possible.
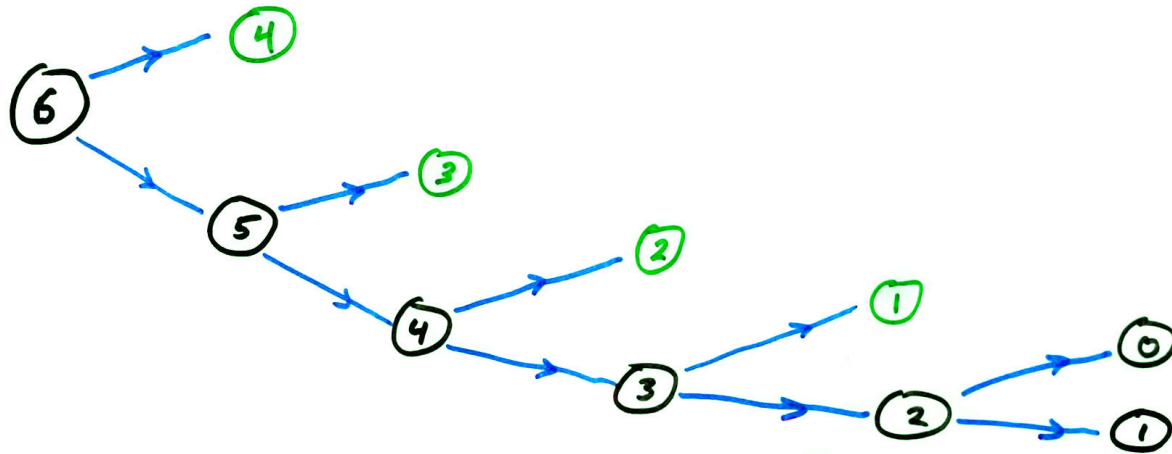
This is called **memoization**. It only works for **pure functions**, i.e. those which always produce the same return value for any given argument values.

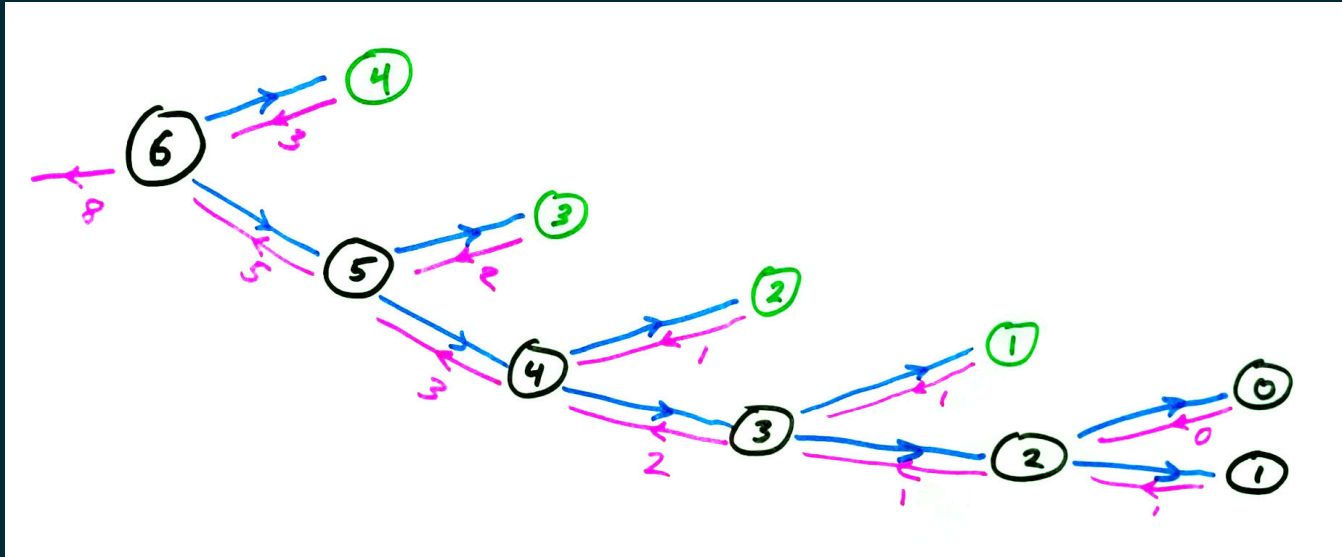`math.sin(...)` is pure; `random.random()` is not.

MEMOIZING FIB

Let's add a simple memoization feature to our recursive `fib` function.

MEMOIZED FIB CALL GRAPH

MEMOIZED FIB CALL GRAPH

FIBONACCI TIMING SUMMARY

	n=35	n=450
recursive	1.9s	> age of universe
memoized recursive	<0.001s	0.003s
iterative	<0.001s	0.001s

Measured on a 4.00Ghz Intel i7-6700K CPU (2015 release date) with Python 3.8.5

MEMOIZATION SUMMARY

Recursive functions with multiple self-calls often benefit from memoization.

Memoized version is conceptually similar to an iterative solution.

Memoization does not alleviate recursion depth limits.

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls							

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls	1						

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls	1	1					

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls	1	1	3				

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls	1	1	3	5			

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls	1	1	3	5	9		

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls	1	1	3	5	9	15	

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6
calls	1	1	3	5	9	15	25

CALL COUNTS

One way to measure the *expense* of a recursive function is to count how many times the function is called.

Let's do this for recursive `fib`.

n	0	1	2	3	4	5	6	
calls	1	1	3	5	9	15	25	
F_n	0	1	1	2	3	5	8	13

Theorem: Let $T(n)$ denote the total number of times `fib` is called to compute `fib(n)`. Then

$$T(0) = T(1) = 1$$

and

$$T(n) = T(n-1) + T(n-2) + 1.$$

Corollary: $T(n) = 2F_{n+1} - 1.$

**Proof of corollary:** Let $S(n) = 2F_{n+1} - 1.$ Then $S(0) = S(1) = 1,$ and

$$\begin{aligned} S(n) = 2F_{n+1} - 1 &= 2(F_n + F_{n-1}) - 1 \\ &= (2F_n - 1) + (2F_{n-1} - 1) + 1 \\ &= S(n-1) + S(n-2) + 1 \end{aligned}$$

Therefore S and T have the same first two terms, and follow the same recursive definition based on the two previous terms.

Corollary: Every time we increase n by 1, the naive recursive `fib` does $\approx 61.8\%$ more work.
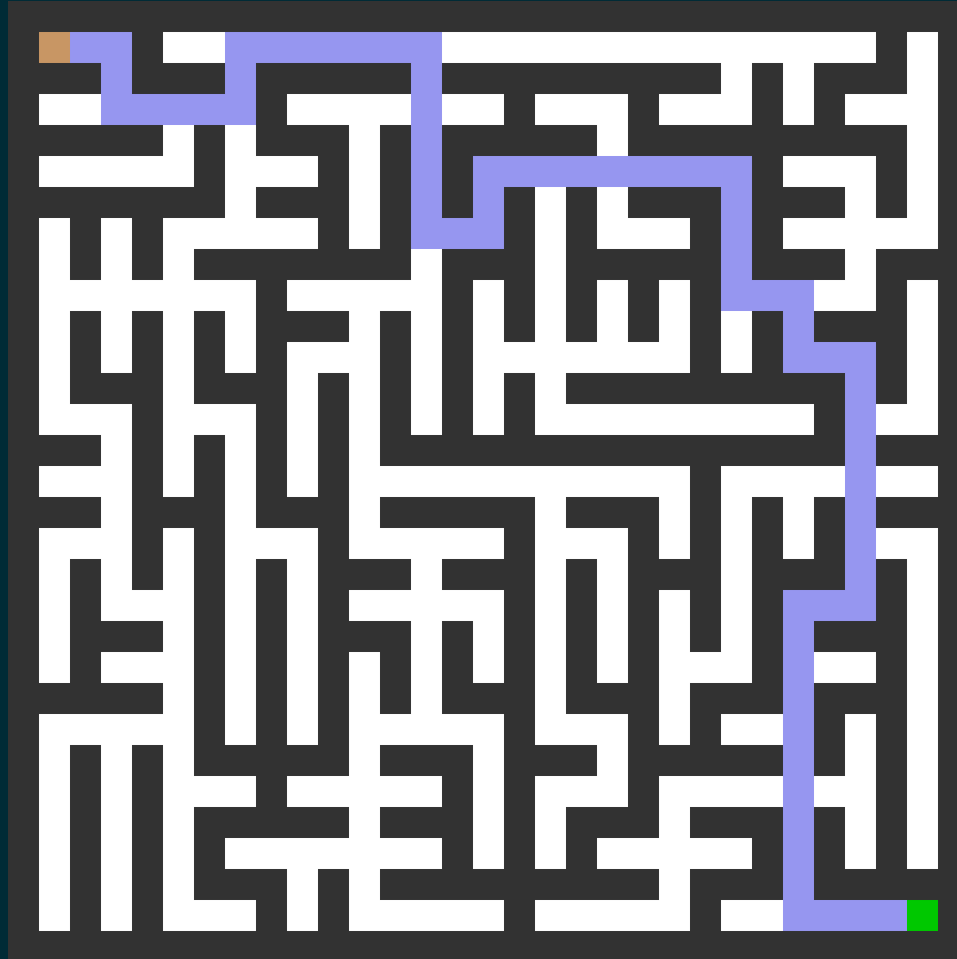
(The ratio F_{n+1}/F_n approaches $\frac{1+\sqrt{5}}{2} \approx 1.61803$.)

RECURSION WITH BACKTRACKING
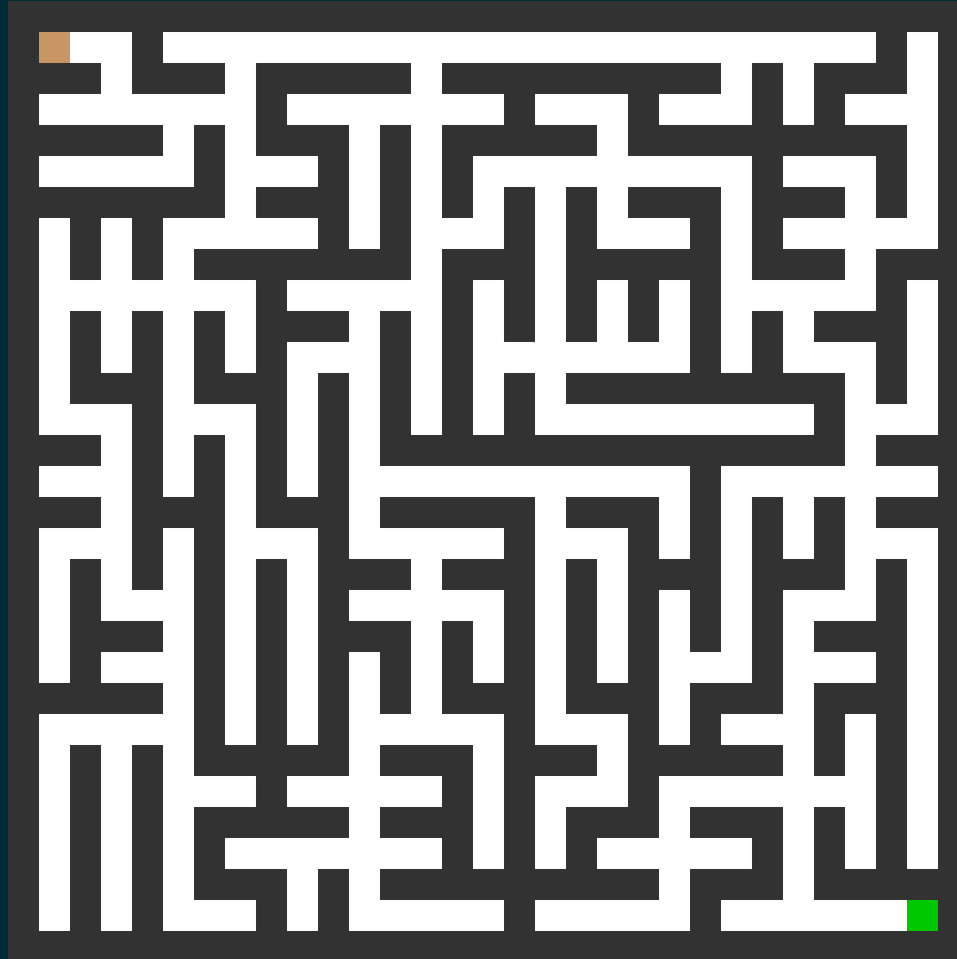
How do you solve a maze?

RECURSION WITH BACKTRACKING

How do you solve a maze?

My guess at your mental algorithm:

- Try something (move around but don't return to anywhere you've visited).
- If you reach a dead end, go back a bit and reconsider which way to go at a recent intersection.

An algorithm that formalizes this is **recursion with backtracking**.

We make a function that takes:

- The maze
- The path so far

Its goal is to add one more step to the path, never backtracking, and call itself to finish the rest of the path.

But if it hits a dead end, it needs to notice that and **backtrack**.

BACKTRACKING

Backtracking is implemented through the return value of a recursive call.

Recursive call may return:

- A solution, or
- `None`, indicating that only dead ends were found.

Algorithm `depth_first_maze_solution`:

Input: a *maze* and a *path* under consideration (partial progress toward solution).

1. If the path is a solution, just return it.

2. Otherwise, enumerate possible next steps that don't go backwards.

3. For each of the possible next steps:
   - Make a new path by adding this next step to the current one.
   - Make a recursive call to attempt to complete this path to a solution.
   - If recursive call returns a solution, we're **done**. Return it immediately.
   - (If recursive call returns `None`, continue the loop.)

4. If we get to this point, every continuation of the path is a dead end. Return `None`.

DEPTH FIRST

This method is also called a **depth first search** for a path through the maze.

Here, **depth first** means that we always add a new step to the path before considering any other changes (e.g. going back and modifying an earlier step).

MAZE COORDINATES

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)

REFERENCES

Same suggested references as Lecture 13.

REVISION HISTORY

- 2021-02-11 Initial publication