LECTURE 11

RECURSION VS ITERATION II

MCS 275 Spring 2023
David Dumas

LECTURE 11: RECURSION VS ITERATION II

Reminders and announcements:

- Project 1 due Friday at 6pm
- Project 2 description coming by Monday
- Remember to check the recursion sample code.

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 my old office computer (2015 Intel i7-6700K) with Python 3.8.5

CALL COUNTS

Another way to measure the cost 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

Another way to measure the cost 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

Another way to measure the cost 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

Another way to measure the cost 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

Another way to measure the cost 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

Another way to measure the cost 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

Another way to measure the cost 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

Another way to measure the cost 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

Another way to measure the cost 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: Both sequences $T(n)$ and $2F_{n+1} - 1$ have the same first two terms, and obey the same recursion relation. Induction.

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

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

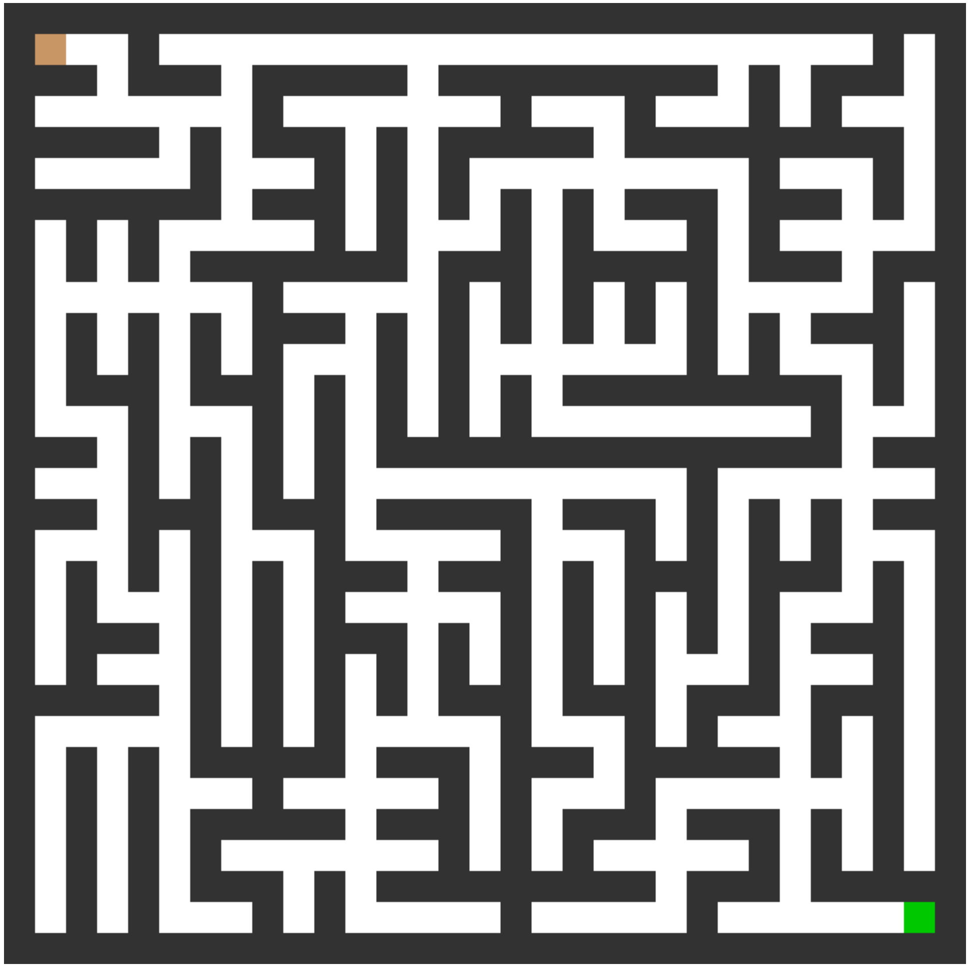
$$\begin{aligned} S(n) &= 2F_{n+1} - 1 = 2(F_n + F_{n-1}) - 1 \\ &= (2F_n - 1) + (2F_{n-1} - 1) + 1 \\ &= S(n-1) + S(n-2) + 1 \end{aligned}$$

Therefore S and T have the same first two terms, and follow the same recursive definition based on the two previous terms. By induction, the set of n such that $T(n) = 2F_{n+1} - 1$ is all of $\mathbb{N}$.

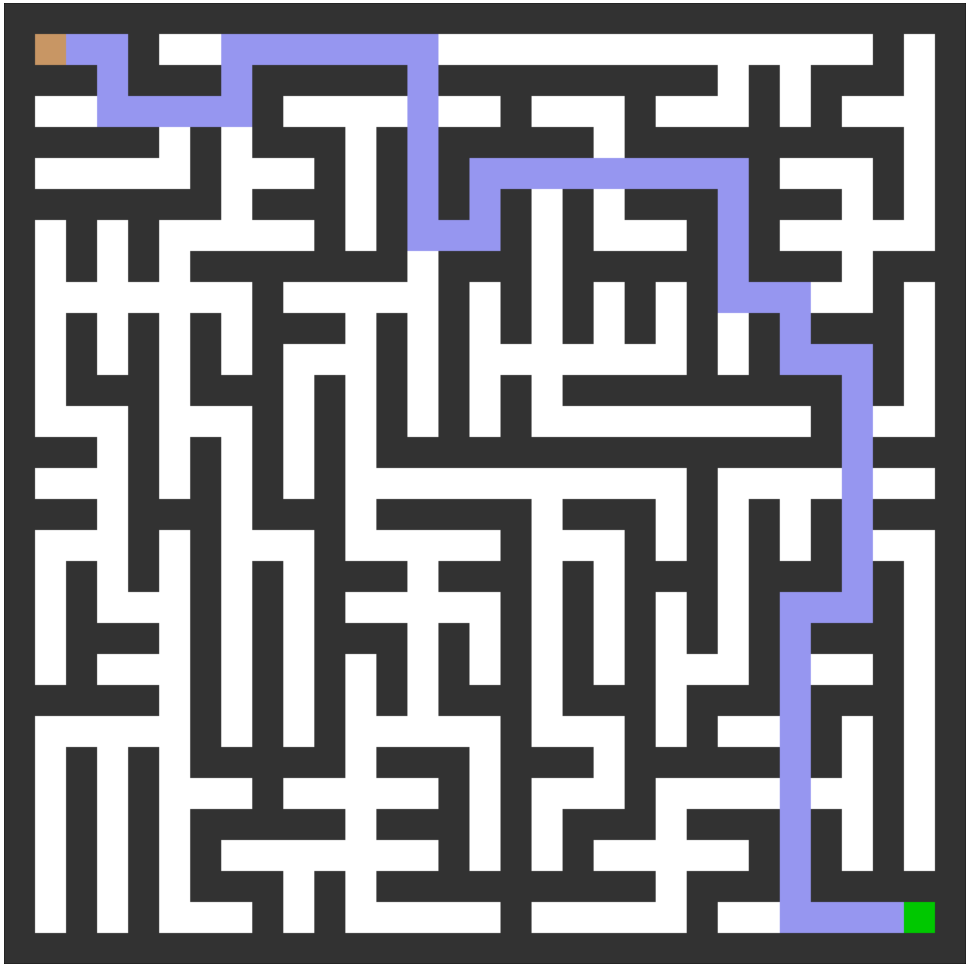
Corollary: Every time we increase n by `1`, the naive recursive `fib` does $\approx 61.8\%$ more work.

(The ratio F_{n+1}/F_n approaches $\frac{1+\sqrt{5}}{2} \approx 1.61803$.)

RECURSION WITH BACKTRACKING



RECURSION WITH BACKTRACKING



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).

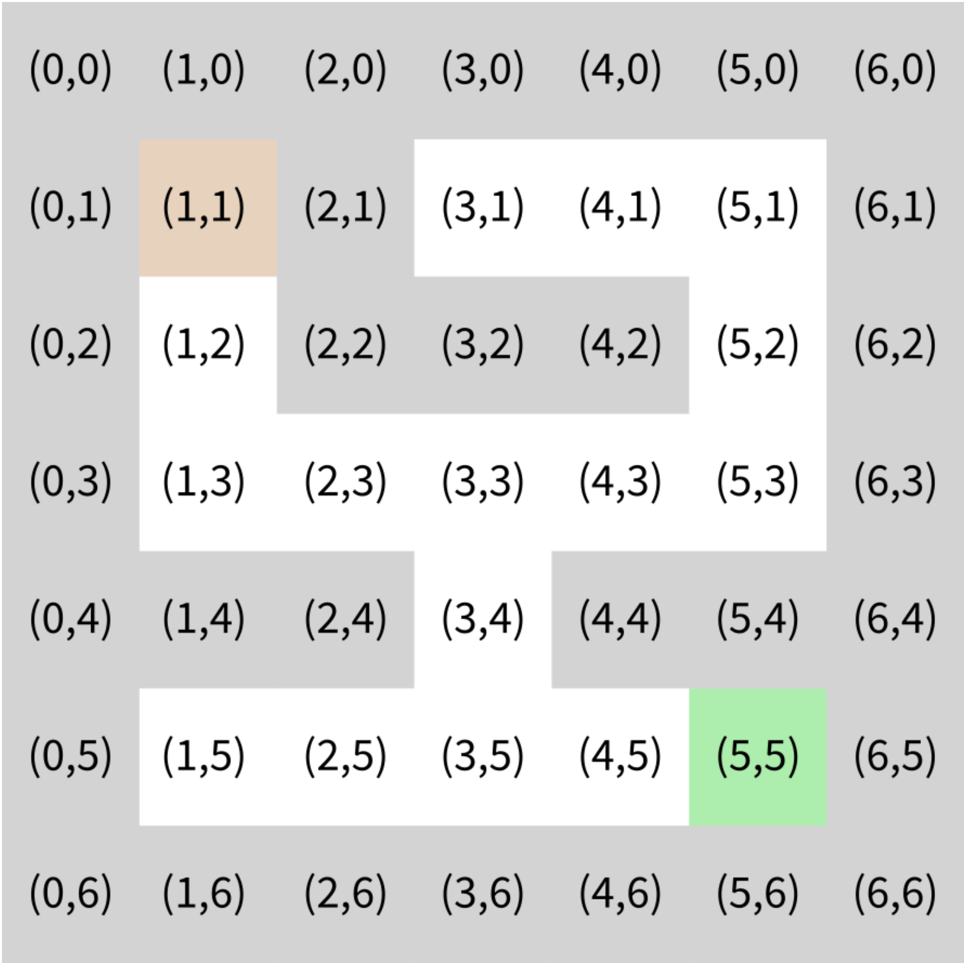
1. If the path is a solution, just return it.
2. Otherwise, enumerate possible next steps that don't go backwards.
3. For each of the possible next steps:
    - Make a new path by adding this next step to the current one.
    - Make a recursive call to attempt to complete this path to a solution.
    - 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



MAZE API

Let's explore the `Maze` class from `maze.py`.

REFERENCES

Same suggested references as Lecture 10.

REVISION HISTORY

- 2022-02-11 Last year's lecture on this topic finalized
- 2023-02-08 Updated version for spring 2023