Slide 2

From my experience, grasping the idea of Recursion only comes through practice and repetition, so we will be coding several examples to practice our recursion skills

Slide 3

Stack’s are used in almost every modern programming language for “control flow”. Control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

Slide 5

We’ve got three functions,

1. a function “multiply” which multiplies two numbers
2. “Square” which calls multiply with the same number twice
3. “printSquare”, which prints the square of a number by calling square and then printing that value via console.log

Slide 6

First, main() function signifies the file itself so we push that onto the call stack.

For the sake of this example, please ignore main(), technically in lower level programming languages there is a main function that is called before other functions or “subroutines” are executed. So you can think of this like initializing our program

Slide 7

then we have some function definitions (they are just like defining the state of the world) and then we use one of our function definitions by calling printSquare(4)

so printSquare is a function call so we push that onto the stack

Slide 8

then we immediately call square(n) so we push that onto the stack

Slide 9

then we immediately call multiply(n, n) so we push that onto the stack

Slide 10

Multiply is the first function where we actually reach the “return” statement

so in multiply we return a \* b and pop off of the stack

(whenever you hit a return statement in JavaScript, the JS engine is popping something off the stack)

Slide 11

Square returns

Slide 12

Then we get back to our printSquare function, and console.log is added to the stack

Slide 13

After we the console.log finishes, printSquare function is over and the stack is cleared

Slide 17

base case: this is a statement, usually within a conditional clause like if, that stops the recursion. If you don't have a base case, the recursion will go on infinitely and your program will crash.

When building your recursive case (the code that will be repeated), one of the rules of thumb is to ensure that the arguments you use for the recursion will lead to your base case.

Slide 18

Some problems are better expressed recursively. This example seems more logical to be written iteratively (with a loop), but I wanted to illustrate the fact that every time you use an iterative approach, you can use recursion. In fact, many programming languages don't have loops like for or while, and use recursion for everything.

Slide 23

We are going to simulate what the call stack would look like. The # in red represents the recursive level we are at. The red arrow represents what line of code is being executed at any given time

Slide 32

See problems.js for all of these code snippets so far

Try this code yourself in a visualization of the JS runtime <http://latentflip.com/loupe/?code=ZnVuY3Rpb24gZmFjdG9yaWFsKG51bSl7DQogIGlmKG51bSA8IDApew0KICAgIHJldHVybiAtMTsNCiAgfWVsc2UgaWYobnVtID09PSAwKXsNCiAgICByZXR1cm4gMTsNCiAgfWVsc2V7DQogICAgcmV0dXJuIChudW0gKiBmYWN0b3JpYWwobnVtIC0gMSkpOw0KICB9DQp9DQoNCmZhY3RvcmlhbCgzKTs%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D>

Slide 33

Each dot can be considered a “node”.

“leaf” nodes are the lowest most nodes on any branch of the tree

The highest node is considered the “root”, like the root of a tree

Slide 34

Imagine of we tried to attempt this problem with just iteration, how could we anticipate the infinite number of levels down that each node of a tree could have, all the different branches, it would be nearly impossible to attempt this problem without recursion.

Using recursion however, we will realize that this problem isn’t so difficult after all.

Slide 38

See treeCountLeaves example