@ChiragG28
Javascript Basics Part 3 :

Consider this puzzle: by starting from the number 1 and repeatedly either adding 5 or multiplying by 3, an infinite set of numbers can be produced.

How would you write a function that, given a number, tries to find a sequence of such additions and multiplications that produces that number ?

- For example : the number 13 could be reached by first multiplying by 3 and then adding 5 twice, whereas the number 15 cannot be reached at all.

Here is a recursive solution:

```
function findSolution(target)
{
     function find(current, history)
     {
          if (current == target)
          {
               return history;
          }
          else if (current > target)
          {
               return null;
          }
          else
          {
               return find(current + 5, `(${history} + 5)`) ||
               find(current * 3, `(${history} * 3)`);
          }
     }
     return find(1, "1");
}

console.log(findSolution(24));
```

```
// → (((1 * 3) + 5) * 3)
```

The function performs one of three actions.

If the current number is the target number, the current history is a way to reach that target, so it is returned.

If the current number is greater than the target, there's no sense in further exploring this branch because both adding and multiplying will only make the number bigger, so it returns null.

Finally, if we're still below the target number, the function tries both possible paths that start from the current number by calling itself twice, once for addition and once for multiplication.

If the first call returns something that is not null , it is returned. Otherwise, the second call is returned, regardless of whether it produces a string or null .

To better understand how this function produces the effect we're looking for, let's look at all the calls to find that are made when searching for a solution for the number 13 .

```
find(1, "1")
     find(6, "(1 + 5)")
          find(11, "((1 + 5) + 5)")
               find(16, "(((1 + 5) + 5) + 5)")
               too big
               find(33, "(((1 + 5) + 5) * 3)")
               too big
          find(18, "((1 + 5) * 3)")
               too big
     find(3, "(1 * 3)")
          find(8, "((1 * 3) + 5)")
               find(13, "(((1 * 3) + 5) + 5)")
     Found!
```

The indentation indicates the depth of the call stack.

The first time find is called, it starts by calling itself to explore the solution that starts with (1 + 5) .

That call will further recurse to explore every continued solution that yields a number less than or equal to the target number.

Since it doesn’t find one that hits the target, it returns null back to the first call. There the || operator causes the call that explores (1 * 3) to happen.

This search has more luck its first recursive call, through yet another recursive call, hits upon the target number.

That innermost call returns a string, and each of the || operators in the intermediate calls passes that string along, ultimately returning the
solution.