Learning Objectives: Recursion

Learners will be able to...

- **Define recursion**
- **Identify the base case**
- **Identify the recursive pattern**
- **Identify when to use recursion**

Recursion in Data Structures

What is Recursion?

Recursion is a programming concept where a function calls itself inside its own definition. Recursion refers to the method where the solution to a problem depends on solutions to smaller instances of the same problem. This may seem like a strange idea, but it can be used to solve problems that would be difficult or impossible to solve with a traditional, iterative approach.

To understand recursion, it's helpful to think of a problem that can be broken down into smaller and smaller subproblems (these are sometimes referred to as self-similar problems). In data structures and algorithms, recursion plays an essential role in breaking down complex problems into more manageable parts.

When to Use Recursion

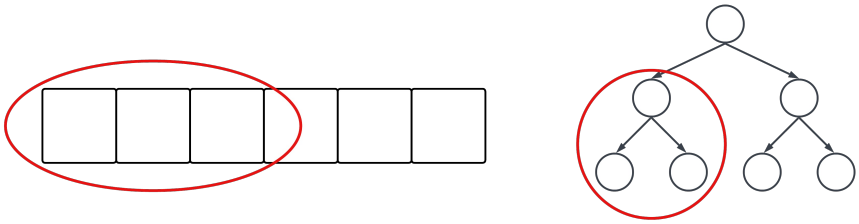
Recursion is especially potent in scenarios where:

1. **Natural Hierarchy Exists**: For instance, in tasks like traversing a file directory.
2. **Problem Can Be Broken Down**: Such as sorting, where a large list can be split into smaller ones.
3. **Solution Requires Multiple Steps**: For example, calculating factorial values or Fibonacci numbers.

Applying recursion requires a mental shift because it can initially seem counterintuitive. Instead of seeing the solution in a linear, step-by-step manner, one needs to trust that smaller versions of the problem will be solved, culminating in the solution to the original issue. The smallest problem we know how to solve in recursion is the base case.

Recursive Algorithms

Recursion is fundamental when working with data structures and algorithms. There are many data structures that are self-similar. These can be linear data or non-linear structures. If you look at the first half of an array, we can think of this half as being its own, smaller array. The same thing can be done to a tree. A subset of a tree can be thought of being its own, smaller tree.



Because of the inherent, self-similar structure of some of the data structures, common algorithms often traverse these structures in a recursive manner. This could be for searching, for sorting, or for something entirely different. Regardless, recursion and data structures and algorithms go hand in hand.

Factorial

Calculating Factorial Recursively

The mathematical concept of factorial ($n!$) is the product of all consecutive, positive integers less than or equal to n. In other words:

$$n! = n \times (n-1) \times (n-2) ... \times 3 \times 2 \times 1$$

Let's look at the concrete example of $5!$. This can be expressed as $5 \times 4 \times 3 \times 2 \times 1$. If we look closely at a subset of this, we can see that $4 \times 3 \times 2 \times 1$ is actually $4!$. So we can say that $5! = 5 \times 4!$. If we continue with this logic, then $4! = 4 \times 3!$. We can continue breaking down this larger problem down into smaller, self-similar subproblems. Mouse over the image below to see the recursive structure of $5!$.

Because $5!$ is self-similar, recursion can be used to calculate the answer. Copy the following code into the text editor on your left and click `TRY IT` to test the code. You should see `120` as the result.

```java
public class Factorial {

  public static int factorial(int n) {
    if (n == 1) { //base case
      return 1;
    } else { //recursive step
      return n * factorial(n - 1);
    }
  }

  public static void main(String[] args) {
    System.out.println(factorial(5));
  }
}
```

Recursion is an abstract and difficult topic, so it might be a bit hard to follow what is going on here. When `n` is 5, it starts a multiplication problem of `5 * factorial(4)`. The method runs again and the multiplication problem becomes `5 * 4 * Factorial(3)`. This continues until `n` is 1. It returns the value `1`, and Java solves the multiplication problem `5 * 4 * 3 * 2 * 1`. The video below should help explain how `5!` is calculated recursively.

The Base Case

Each recursive method has two parts: the recursive case (where the method calls itself with a different parameter) and the base case (where the method stops calling itself and sometimes returns a value).

The base case is the most important part of a recursive method. Without it, the method will never stop calling itself. Like an infinite loop, Java will stop the program with an error. Replace the `factorial` method in your code with the one below and see what happens.

```
// This recursive method returns an error. THE BASE CASE IS
      MISSING.
public static int factorial(int n) {
  return n * factorial(n - 1);
}
```

When creating a recursive algorithm, always start with the base case. That is, determine when your algorithm should stop running. Each time the method is called recursively, the program should get one step closer to the base case.

challenge

Try this variation:

If you were to call the `factorial` method with `0` as the argument, you will see an error since the method would never stop calling itself.

```
System.out.println(factorial(0));
```

Modify the `factorial` method so that it will return `1` if it is passed any non-positive integer.

▶ **Solution**

A Deep Dive with Merge Sort

Recursive Merge Sort

Let's take another look at the merge sort, which we briefly saw in the previous module. A merge sort lends itself nicely to recursion because of the way it tackles the sorting problem. Instead of trying to sort an entire list (which is too big), a merge sort divides the array in half and tries to sort the smaller arrays.

The algorithm keeps dividing the arrays until they are small enough to be sorted. These sorted arrays are combined with other arrays and sorted once more. A merge sort keeps combining all of these sorted arrays until there is only one array left, which is completely sorted.

Dividing a big problem into smaller, easier to solve problems is a hallmark of recursion. Here is a recursive implementation of a merge sort:

```java
public class MergeSort {

    private static void merge(int arr[], int left, int
        mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int leftArray[] = new int[n1];
        int rightArray[] = new int[n2];

        for (int i = 0; i < n1; ++i)
            leftArray[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            rightArray[j] = arr[mid + 1 + j];

        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (leftArray[i] <= rightArray[j]) {
                arr[k] = leftArray[i];
                i++;
            } else {
                arr[k] = rightArray[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = leftArray[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = rightArray[j];
            j++;
            k++;
        }
    }

    public static void sort(int arr[], int left, int
        right) {
        if (left < right) {
            int mid = (left + right) / 2;
            sort(arr, left, mid);
            sort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }
```

In particular, let's focus on the `sort` method where recursion is actually used. Unlike the factorial example, we do not see two branches in our conditional. It is much harder to differentiate between the base case and the recursive case because we have only one branch in our conditional. The only branch is the recursive case. As long as `left` is less than `right`, the method will keep recursing. When `left` is no longer less than `right`, the method skips the recursive calls and simply ends. This is the base case.

```
void sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        sort(arr, left, mid);         // Recursive call
on left half
        sort(arr, mid + 1, right);    // Recursive call
on right half
        merge(arr, left, mid, right);
    }
}
```

The algorithm continually splits the array in half; `left` to `mid` is one half, while `mid + 1` to `right` is the other half. This splitting continues until each sub-array has a single element. In an array with a length of one, `left` and `right` will have the same value, which means `left` is no longer less than `right`. These sublists are then merged together, in a recursive manner, to produce a sorted list.

Formative Assessment 1

Formative Assessment 2