# 11
# More About Algorithms

So far, we have had fun learning how to implement several different data structures—among them, the most used sorting and searching algorithms. The algorithm and programming world is very interesting. In this chapter, you will learn more about this world, and we are also going to discuss the next steps in case you are interested in delving deeper into this world.

We are going to cover **recursion**, which was introduced in *Chapter 8*, *Trees*. We are also going to cover **dynamic programming** and **greedy algorithms** and the famous **big O notation** (introduced in *Chapter 10*, *Sorting and Searching Algorithms*), and also see how we can have some fun with algorithms and boost our knowledge to improve our programming and problem-solving skills.

## Recursion

Recursion is a method to solve problems that consists of solving smaller portions of the same problem until you solve the original larger problem. It usually involves calling the function itself.

A method or function is recursive if it can call itself directly as follows:

```
var recursiveFunction = function(someParam){
  recursiveFunction(someParam);
};
```

A function is also called recursive if it can call itself indirectly as follows:

```
var recursiveFunction1 = function(someParam){
  recursiveFunction2(someParam);
};
```

```
var recursiveFunction2 = function(someParam){
  recursiveFunction1(someParam);
};
```

Suppose we have to execute `recursiveFunction`. What would the result be? In this case, it would be executed indefinitely. For this reason, every recursive function must have a base case, which is a condition where no recursive call is made (a stop point) to prevent infinite recursion.

# JavaScript limitation on call stack size

What happens when we forget to add a base case to stop the recursive calls of a function? It will not be executed indefinitely; the browser will throw an error, which is known as a stack overflow error.

Each browser has its own limitations, and we can use the following code to do some testing:

```
var i = 0;

function recursiveFn () {
    i++;
    recursiveFn();
}

try {
    recursiveFn();
} catch (ex) {
    alert('i = ' + i + ' error: ' + ex);
}
```

In Chrome Version 37, the function is executed 20,955 times and the browser throws the error `RangeError: Maximum call stack size exceeded`. In Firefox Version 27, the function is executed 343,429 times and the browser throws the error `InternalError: too much recursion`.

Depending on your operating system and browser, the values might be different, but will be close.

ECMAScript 6 has **tail call optimization**. If a function call is the last action inside a function (as in our example; the highlighted line), it is handled via a "jump," not via a "subroutine call." This means that our code can be executed forever in ECMAScript 6. This is why it is very important to have a base case to stop recursion.

For more information about tail call optimization, please visit `http://goo.gl/ZdTZzg`.

# The Fibonacci sequence

Let's go back to the Fibonacci problem, which was covered in *Chapter 10*, *Sorting and Searching Algorithms*. The Fibonacci sequence can be defined as follows:

- The Fibonacci of 1 or 2 is 1
- The Fibonacci of n (for n>2) is the Fibonacci of (n-1) + the Fibonacci of (n-2)

So, let's start implementing the Fibonacci function:

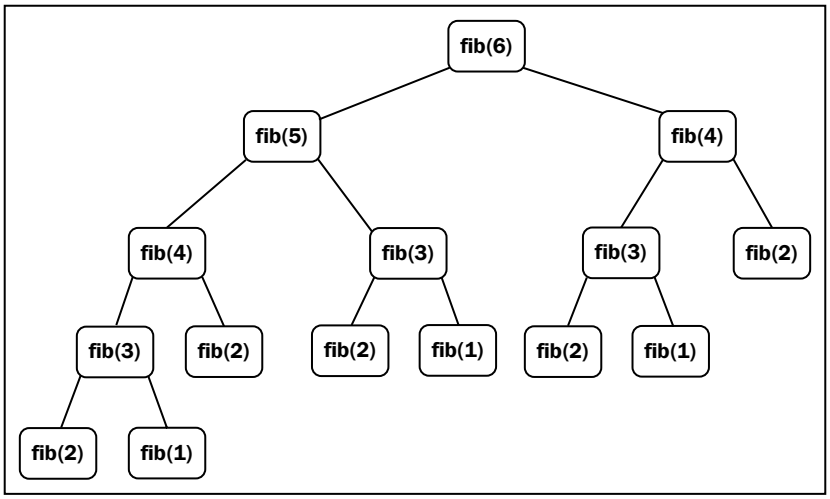
```
function fibonacci(num){
    if (num === 1 || num === 2){ //{1}
        return 1;
    }
}
```

We know the base case; the Fibonacci of 1 or 2 (`{1}`) is 1. Now, let's finish its implementation:

```
function fibonacci(num){
    if (num === 1 || num === 2){
        return 1;
    }
    return fibonacci(num - 1) + fibonacci(num - 2);
}
```

We also know that if n is greater than 2, then Fibonacci(n) is Fibonacci(n-1) + Fibonacci(n-2).

Now, we have the Fibonacci function implemented. If we try to find the Fibonacci of 6, the following will be the result of the calls that are made:




We could also implement the Fibonacci function in a non-recursive way:

```
function fib(num){
    var n1 = 1,
        n2 = 1,
        n = 1;
    for (var i = 3; i<=num; i++){
        n = n1 + n2;
        n1 = n2;
        n2 = n;
    }
    return n;
}
```

Why use recursion? Is it faster? Recursion is not faster than the normal version; it is slower. But note that recursion is clear to understand and it requires less code as well.

In ECMAScript 6, because of tail call optimization, recursion will not be slower. But in other languages, recursion code is usually slower.

So, we usually use recursion because it is easier to solve problems using it.

# Dynamic programming

**Dynamic programming** (**DP**) is an optimization technique used to solve complex problems by breaking them in to smaller sub-problems.

We covered some dynamic programming techniques earlier in this book. One problem we solved with dynamic programming was DFS in *Chapter 9*, *Graphs*.

Note that the dynamic programming approach is different from the divide and conquer approach (as we used for the merge sort and quick sort algorithms). While the divide and conquer approach will break the problem into independent sub-problems and then combine the solutions, dynamic programming breaks the problem into dependent sub-problems.

Another example is the Fibonacci problem we solved in the previous section. We broke the Fibonacci problem in to smaller problems as shown in the diagram of that section.

There are three important steps we need to follow when solving problems with DP:

1. Define the sub-problems.
2. Implement the recurrence that solves the sub-problems (and in this step, we need to follow the steps for recursion we discussed in the previous section).
3. Recognize and solve the base cases.

There are some famous problems that can be solved with dynamic programming:

- **The knapsack problem**: In this problem, given a set of items, each one with a value and volume, the goal is to determine the best collection of items out of the set in a way to maximize the total value. The constraint of the problem is that the total volume needs to the volume supported by the "knapsack" or less.
- **Longest common subsequence**: This consists of finding the longest subsequence (a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements) common to all sequences in a set of sequences.
- **Matrix chain multiplication**: In this problem, given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices (with as few operations as possible). The multiplication is not performed; the solution is finding the sequences in each of the matrices that need to be multiplied.

- **Coin change**: This consists of finding how many different ways we can make change for a particular amount of cents using a given amount of set denominations, d1… dn.
- **All-pairs shortest paths in a graph**: This consists of finding the shortest path from vertex u to vertex v for all pairs of vertices (u, v).

We are going to cover a variation of the coin change problem in the following example.

# The minimum coin change problem

The minimum coin change problem is a variation of the coin change problem. The coin change problem consists of finding in how many ways we can make change for a particular amount of cents using a given amount of set denominations, d1… dn. The minimum coin change problem consists of finding the minimum number of coins needed to make a particular amount of cents using a given amount of set denominations, d1… dn.

For example, the United States has the following denominations (coins): d1 = 1; d2 = 5; d3 = 10; d4 = 25.

If we need to make change for 36 cents, we can use 1 quarter (25), 1 dime (10), and 1 penny (1).

How do we transform this solution into an algorithm?

The min-coin change solution consists of finding the minimum number of coins for `n`. But to do that, first we need to find the solution for every `x < n`. Then, we build up the solution out of the solutions for smaller values.

Let's take a look at the algorithm:

```
function MinCoinChange(coins){
    var coins = coins; //{1}
    var cache = {};    //{2}

    this.makeChange = function(amount) {
        var me = this;
        if (!amount) { //{3}
            return [];
        }
        if (cache[amount]) { //{4}
            return cache[amount];
        }
```

```
        var min = [], newMin, newAmount;
        for (var i=0; i<coins.length; i++){ //{5}
            var coin = coins[i];
            newAmount = amount - coin;  //{6}
            if (newAmount >= 0){
                newMin = me.makeChange(newAmount); //{7}
            }
            if (
                newAmount >= 0 && //{8}
                (newMin.length < min.length-1 || !min.length)//{9}
                && (newMin.length || !newAmount) //{10}
                ){
                min = [coin].concat(newMin); //{11}
                console.log('new Min ' + min + ' for ' + amount);
            }
        }
        return (cache[amount] = min); //{12}
    };
}
```

To be more organized, we created a class that will solve the min-coin change problem given the denominations. Let's go through the algorithm step by step.

Our `MinCoinChange` class receives the `coins` parameter (`{1}`), which represents the denominations of our problem. For the US coin system, it would be `[1, 5, 10, 25]`. We can pass any denominations that we like. Also, to be more efficient and not recalculate values, we will keep `cache` (`{2}`).

Then, we have the `makeChange` method, which is also recursive and is the method that will solve the problem for us. First, if `amount` is not positive (`< 0`), then we return an empty array (`{3}`); at the end of the execution of this method, we will return an array with the amount of each coin that can be used to make the change (the minimum amount of coins). Next, we will check `cache`. If the result is already cached (`{4}`), then we simply return its value; otherwise, we execute the algorithm.

To help us, we will solve the problem based on the `coins` parameter (denominations). So, for each coin (`{5}`), we will calculate `newAmount` (`{6}`), which is going to decrease the value until we reach the minimum amount of change we can give (remember that this algorithm will calculate all `makeChange` results to `x < amount`). If `newAmount` is a valid value (positive value), then we will calculate the result for it as well (`{7}`).

At the end, we will verify whether `newAmount` is valid, whether `minValue` (the minimum amount of coins) is the best result, and whether `minValue` and `newAmount` are valid values (`{10}`). If all the verifications are positive, it means we have a better result than previously (line `{11}`—for example, for 5 cents, we can give 5 pennies or 1 nickel, 1 nickel being the best solution). At the end, we return the final result (`{12}`).

Let's test the algorithm:

```
var minCoinChange = new MinCoinChange([1, 5, 10, 25]);
console.log(minCoinChange.makeChange(36));
```

Note that if we inspect the `cache` variable, it will hold all the results for 1 to 36 cents. The result for the preceding code will be `[1, 10, 25]`.

In the source code of this book, you will find some extra lines of code that will output the steps of this algorithm. For example, if we use the denominations `[1, 3, and 4]` and execute the algorithm for the amount 6, we will produce the following output:

```
new Min 1 for 1
new Min 1,1 for 2
new Min 1,1,1 for 3
new Min 3 for 3
new Min 1,3 for 4
new Min 4 for 4
new Min 1,4 for 5
new Min 1,1,4 for 6
new Min 3,3 for 6
[3, 3]
```

So, for the amount 6, the best solution is giving 2 coins of value 3.

# Greedy algorithms

A greedy algorithm follows the problem-solving heuristic of making the locally optimal choice (the best solution at the time) at each stage with the hope of finding a global optimum (global best solution). It does not evaluate the bigger picture like a dynamic programming algorithm does.

Let's use the same problem we covered in the dynamic programming topic so we can see the difference.

# The min-coin change problem

The min-coin change problem can also be resolved with a greedy algorithm. Most of the time, the result will also be optimal, but for some denominations, the result is not going to be optimal.

Let's take a look at the algorithm:

```
function MinCoinChange(coins){
    var coins = coins; //{1}

    this.makeChange = function(amount) {
        var change = [],
            total = 0;
        for (var i=coins.length; i>=0; i--){ //{2}
            var coin = coins[i];
            while (total + coin <= amount) { //{3}
                change.push(coin);           //{4}
                total += coin;               //{5}
            }
        }
        return change;
    };
}
```

Note that the greedy version of `MinCoinChange` is much simpler than the DP one. Similar to the dynamic programming approach, we will instantiate `MinCoinChange` passing the denominations as a parameter (`{1}`).

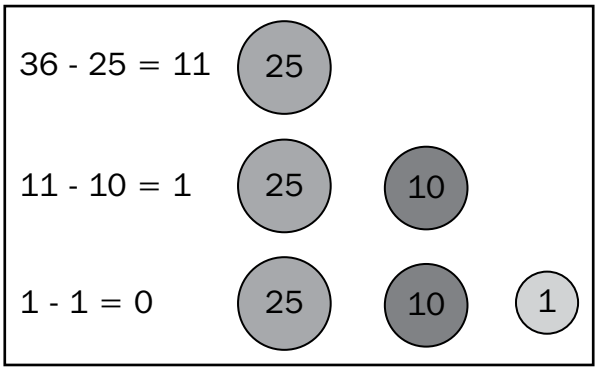
For each coin (`{2}`—starting from the biggest one to the smallest one), we will add the coin value to `total`, and `total` needs to be less than `amount` (`{3}`). We will add `coin` to the result (`{4}`) and also to `total` (`{5}`).

As you can see, the solution is very simple. We start with the coin with the greatest value and we give the change that is possible with this coin. When we cannot give more coins for the current coin value, we start giving change with the coin that has the second greatest value, and so on.

To test the code, we will use the same code we used in the DP approach:

```
var minCoinChange = new MinCoinChange([1, 5, 10, 25]);
console.log(minCoinChange.makeChange(36));
```

The result will also be `[25, 10, 1]`, the same result that we got using DP. The following diagram exemplifies how the algorithm is executed:




However, if we use the denominations `[1, 3, 4]` and execute the preceding greedy algorithm, we will get `[4, 1, 1]` as the result. If we use the dynamic programming solution, we will get `[3, 3]` as the result, which is the optimal result.

Greedy algorithms are simpler and also faster than dynamic programming algorithms. However, as we can see, it is not going to give the optimal answer all the time. But on average, it outputs an acceptable solution for the time it takes to execute.

# Big-O notation

In *Chapter 10, Sorting and Searching Algorithms*, we introduced the concept of **big-O notation**. What exactly does this mean? It is used to describe the performance or complexity of an algorithm.

When analyzing algorithms, the following classes of functions are the most commonly encountered:

| **Notation** | **Name** |
|---|---|
| *O(1)* | Constant |
| *O(log(n))* | Logarithmic |
| *O((log(n))c)* | Poly-logarithmic |
| *O(n)* | Linear |
| $O(n^2)$ | Quadratic |
| $O(n^c)$ | Polynomial |
| $O(c^n)$ | Exponential |

# Understanding big-O notation

How do we measure the efficiency of an algorithm? We usually use resources such as CPU (time) usage, memory usage, disk usage, and network usage. When talking about big-O notation, we usually consider CPU (time) usage.

Let's try to understand how big-O notation works using some examples.

## O(1)

Consider the following function:

```
function increment(num){
    return ++num;
}
```

If we try to execute the `increment(1)` function, we will have an execution time equal to *X*. If we try to execute the increment function again with a different parameter (let's say `num` is `2`), the execution time will also be *X*. The parameter does not matter, the performance of the function increment will be the same. For this reason, we can say the preceding function has a complexity of O(1) (constant).

## O(n)

Now, let's use the sequential search algorithm we implemented in *Chapter 10*, *Sorting and Searching Algorithms*, as an example:

```
function sequentialSearch(array, item){
    for (var i=0; i<array.length; i++){
        if (item === array[i]){ //{1}
            return i;
        }
    }
    return -1;
}
```

If we pass an array with 10 elements (`[1, …, 10]`) to this function and look for the element `1`, in the first attempt we will find the element we are looking for. Let's suppose the cost is 1 for each time we execute line `{1}`.

Now, let's suppose we are looking for element 11. The line `{1}` will be executed 10 times (it will iterate through all the values of the array and it will not find the value we are looking for; therefore, it will return `-1`). If the line `{1}` has a cost of 1, executing it 10 times has a cost of 10, which is 10 times more than the first example we used.

Now, suppose that the array has 1000 elements (`[1, …, 1000]`). Searching for element 1001 will result in line `{1}` being executed 1000 times (and then `return -1`).

Note that the total cost of execution of the `sequentialSearch` function depends on the number of elements of the array (size) and also depends on the value we are looking for. If the item we are looking for exists in the array, then how many times will the line `{1}` be executed? If the item we are looking for does not exist, then the line `{1}` will be executed the size-of-the-array times, which we call the worst-case scenario.

Considering the worst-case scenario of the `sequentialSearch` function, if we have an array with size 10, the cost will be 10. If we have an array with size 1000, the cost will be 1000. We can conclude that the `sequentialSearch` function has a complexity of *O(n)*, n being the size of the array (input).

To see the preceding explanation in practice, let's modify the algorithm to calculate the cost (the worst-case scenario):

```
function sequentialSearch(array, item){
    var cost = 0;
    for (var i=0; i<array.length; i++){
        cost++;
        if (item === array[i]){ //{1}
            return i;
        }
    }
    console.log('cost for sequentialSearch with input size ' + array.
length + ' is ' + cost);
    return -1;
}
```

Try executing the preceding algorithm using different input sizes so you can see the different outputs.

# O($n^2$)

For the *O($n^2$)* example, let's use the bubble sort algorithm:

```
function swap(array, index1, index2){
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
}
```

```
function bubbleSort(array){
    var length = array.length;
    for (var i=0; i<length; i++){        //{1}
        for (var j=0; j<length-1; j++ ){ //{2}
            if (array[j] > array[j+1]){
                swap(array, j, j+1);
            }
        }
    }
}
```

Consider that lines `{1}` and `{2}` have a cost of 1 each. Let's modify the algorithm to calculate the cost:

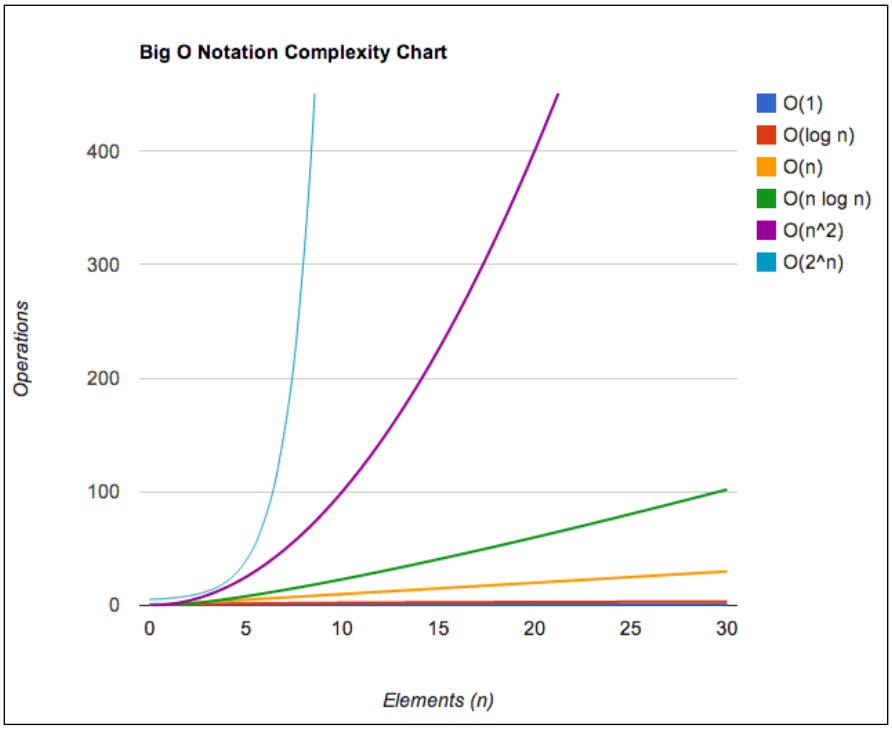
```
function bubbleSort(array){
    var length = array.length;
    var cost = 0;
    for (var i=0; i<length; i++){ //{1}
        cost++;
        for (var j=0; j<length-1; j++ ){ //{2}
            cost++;
            if (array[j] > array[j+1]){
                swap(array, j, j+1);
            }
        }
    }
    console.log('cost for bubbleSort with input size ' + length + ' is
' + cost);
}
```

If we execute `bubbleSort` for an array with size 10, the cost will be 100 ($10^2$). If we execute `bubbleSort` for an array with size 100, the cost will be 1000 ($100^2$); note that the execution will take even longer every time we increase the input size.

Note that the code from the *O(n)* complexity has only one for loop, whereas $O(n^2)$ has two nested for loops. If the algorithm has three for loops iterating through the array, it will probably be $O(n^3)$.

# Comparing complexities

We can compare the different big-O notation complexities mentioned previously using a chart as follows:




This chart was also plotted using JavaScript! You can find the source code used to plot this chart inside the `bigOChart` folder under `chapter11` of the sample code files included with this book.

In the *Appendix, Big-O Cheat Sheet*, you can find a cheat sheet that shows the complexities of the algorithms we implemented in this book.

# Having fun with algorithms

We do not learn algorithms only because we need to study them in college or because we want to become developers. We can become better professionals by boosting our problem-solving skills, using the algorithms we learned in this book as a way of solving problems.

The best way of boosting our knowledge in problem solving is practicing. And practicing does not need to be boring. In this section, we will present some websites that you can go to and start having fun with algorithms (and even earn some cash by doing so!).

Here is a list of some useful websites (some of them do not support sending the problem solution in JavaScript, but we can apply the logic we learned in this book to other programming languages as well):

- **UVa Online Judge** (`http://uva.onlinejudge.org/`): This site contains a set of problems used in several programming contests around the world, including the ACM **International Collegiate Programming Contest** (**ICPC**), sponsored by IBM (if you are still in college, try to participate in this contest, and if your team wins, you can travel around the world with all expenses paid!). This site contains hundreds of problems where we can use the algorithms learned in this book.
- **Sphere Online Judge** (`http://www.spoj.com/`): This site is similar to UVa Online Judge, but supports more languages (including JavaScript submissions).
- **Coder Byte** (`http://coderbyte.com/`): This site contains 74 problems (easy, medium, and hard difficulty) that can also be solved with JavaScript.
- **Project Euler** (`https://projecteuler.net/`): This site contains a series of mathematical/computer programming problems. All you have to do is input the answer to the problem, but we can use algorithms to find the answer for us.
- **Hacker Rank** (`https://www.hackerrank.com`): This site contains 263 challenges divided into 16 categories (we can use the algorithms that we learned in this book and much more). It also supports JavaScript, among other languages.
- **Code Chef** (`http://www.codechef.com/`): This site also contains several problems and hosts competitions online.

- **Top Coder** (`http://www.topcoder.com/`): This site organizes programming tournaments, usually sponsored by companies such as NASA, Google, Yahoo!, Amazon, and Facebook. Some contests can give you the opportunity to work with the sponsored company and some contests can give you cash prizes. The website also offers great tutorials for problem solving and algorithms.

Another nice thing about the previous websites is that they usually present a real-world problem, and we need to identify which algorithm we can use to solve that problem. It is a way of knowing that the algorithms we learned in this book are not only educational, but they can also be applied to solve real-world problems.

If you are starting a career in technology, it is highly recommended to create an account on GitHub (`https://github.com`) for free, and you can commit the source code you write to solve the problems from the previous websites. If you do not have any professional experience, GitHub can help you build a portfolio, and it can also help you get your first job!

# Summary

In this chapter, you learned more about recursion and how it can help us solve some problems using dynamic programming. You also learned about greedy algorithms.

We covered the *big-O* notation and also learned how we can calculate the complexity of an algorithm applying this concept.

We also presented some websites you can register to for free and apply all the knowledge you have acquired reading this book and even be offered your first job in IT!

Happy coding!