Recursion

Ermin Hodžić

May 2023

Requirements. The minimum requirement for understanding the presented material is to know the basics of programming, including how to write and call custom functions. Experience with mathematical notation and proofs by induction is going to make the first three sections more clear and easier to understand, even though an attempt was made to introduce and fully explain mathematical induction in Section 3 so that hopefully even a beginner may understand it. Even though this material does not focus on the running time analysis of algorithms, some understanding of the big-Oh notation will help, even if not necessary.

How the material is organized. This manuscript attempts to engage readers of various levels of expertise in the topic of recursion. Effort has been made to write it so that even complete beginners to recursion (but not to programming) should be able to understand a good part of the presented material and get a good grasp at what recursion is and how it works. However, there is no lack of explanation of advanced concepts everywhere in the manuscript for the more experienced reader, and there is some overlap with other topics in computer science that recursion naturally touches upon. Going in-depth on such topics is out of the scope of this manuscript, and where such topics are introduced, it is to help with the explanation of some aspect of recursion, or to demonstrate a possible use case.

This text was envisioned as a script for a live lecture series, and contains some repetition in the segments that are related. The reader will also sometimes find mentions of concepts and techniques that will not be immediately explained, as they rely on additional material that has not yet been mentioned. However, a great deal of care has been invested into making sure that such "teased" topics are eventually explained.

Goal of the manuscript. The goal is to provide a comprehensive review of recursion – what the main idea is, why it works, how to use it to solve problems and integrate it into the reader's toolset. Some advanced algorithmic techniques and data structures heavily rely on recursive code, or require "recursive thinking" in order to solve problems. Examples of such techniques are divide & conquer and dynamic programming, foundation for both of them being recursion, as well as various graph-traversal-based algorithms such as computing strongly-connected components, discovering articulation points, and topologically sorting directed acyclic graphs. Additionally, data structures such as binary search trees, interval trees and segment trees are naturally recursive structures. Without a good understanding of recursion, one might find it much more difficult to understand and learn the above-mentioned topics.

Contents

1 Introduction **3**

2 Structure of Recursion **5**

3 Mathematical Induction **5**

4 Examples **7**
4.1 Factorials 8
4.2 Numeric Array Sum 8
4.3 The Maximum Element 8
4.4 Number of Trailing 0-digits of a Factorial 9
4.5 Array Sorting 10

5 Problems With Using Recursion **11**

6 Examples of Useful Recursion **13**
6.1 Quicksort 13
6.2 Merge Sort 14
6.3 Exponentiation 16
6.3.1 Fast Fibonacci 17

7 Removing Recursion **18**

1 Introduction

When writing code, one tool at our disposal are functions (sometimes also called methods, or procedures), which are pieces of code that perform a particular task, are given a unique name by which they can be called from other parts of code, and to which we can pass certain parameters that they need in order to perform their tasks. For a simple example, suppose that we are coding a program which is processing a large amount of financial data, and we often find ourselves needing to compute sums of real numbers contained in various arrays. Instead of writing loops that compute the needed sums every time, we can make things easier for ourselves by writing a single function that performs the task, and simply call it whenever we need to, passing the appropriate array as an argument (sometimes also called parameter) to the function. Such a function could perhaps look like this:

Algorithm: sumAllElements(A)

Input: An array of numbers A
Output: The sum of all the numbers in A

```
1 r = 0
2 for (a_i in A) do
3     r += a_i
4 return r
```

From the body (code) of one function, we can call any other function if we need to or want to. For example, if we needed to convert numbers in an array into fractions that they represent relative to the sum of all the numbers of the array, we could achieve that with the help of our already made *sumAllElements* function, like this:

Algorithm: convertToFractions(A)

Input: An array of numbers A
Result: Transforms each element of A into a percentage it represents relative to the sum of all elements

```
1 sum = sumAllElements(A)
2 for (a_i in A) do
3     a_i /= sum
```

Someone might now be wondering “A function can call ANY function? Does it have to be another function? What happens if a function calls itself?” A function can indeed call itself, though it needs to be carefully designed in order to do anything actually useful and not just end up in an infinite loop where it keeps calling itself over and over again and doing nothing else. For example, if we added a call to itself anywhere in the body of our function *sumAllElements*, without changing anything else, we would end up with an entirely useless function which causes our program to become unresponsive. At this point one might be wondering what the purpose of such functionality (a function being able to call itself) is, and how that could be useful at all. Rest easy, because the ability of a function to call itself can indeed be very useful, and some advanced algorithmic techniques even rely on it.

In computer science, a special kind of function which calls itself during its execution in order to solve a problem is called a *recursive* function. Alternatively, one may sometimes see the whole notion called simply *recursion*. To begin to understand why recursion makes sense, it is important to note that a recursive function never calls itself with the exact same parameters that it was itself called with. The code of the function is written once and it does not change during the execution of a program, so if we call the function multiple times with the exact same parameters, it will do exactly the same thing every time. Thus, a function that recursively calls itself with the exact same parameters would end up stuck in an infinite loop. Therefore, the parameters of recursive function calls need to be different from those passed to the parent call. To illustrate this, let us take a look at an example of recursion in mathematics.

In mathematics, the most famous introductory example of recursion are the *Fibonacci numbers*. The n-th Fibonacci number is recursively defined as the sum of two preceding Fibonacci numbers, *i.e.* $F_n = F_{n-1} + F_{n-2}$, with base defined values for the first two Fibonacci numbers $F_0 = F_1 = 1$. To illustrate how recursion works with the example of Fibonacci numbers, let us consider how we would approach the problem of calculating the n-th Fibonacci number.

Problem statement: Given a positive integer n, calculate F_n.

To move away from the abstract and be better able to see the pattern here, let us assume that n is some fixed number, for example 7, and our goal is to calculate F_7. Let us attempt to solve the problem by blindly following the recursive formula and seeing where it gets us. The recursive formula for F_n tells us that $F_7 = F_6 + F_5$. At first glance, this only seems to multiply our problem further – now we need to compute *two* Fibonacci numbers in order to get the result, and the only tool we have is a recursive formula that seems to create more problems to be solved. However, notice that the parameters of the two new problems are smaller than those of the original problem. Indeed, instead of needing to compute F_7, we now "only" need to compute F_6 and F_5. Does that help us? Maybe. Let us keep applying the recursive formula and observe what happens.

$$F_7 = F_6 + F_5 \quad (F_6 \text{ and } F_5 \text{ are new})$$
$$F_6 = F_5 + F_4 \quad \text{and} \quad F_5 = F_4 + F_3 \quad (F_4 \text{ and } F_3 \text{ are new})$$

The amount of different Fibonacci numbers is rapidly increasing – we currently have equations for F_7, F_6 and F_5, and need to calculate F_4 and F_3. Let us keep going and expanding every newly-encountered Fibonacci number with the same formula.

$$F_7 = F_6 + F_5 \quad (F_6 \text{ and } F_5 \text{ are new})$$
$$F_6 = F_5 + F_4 \quad \text{and} \quad F_5 = F_4 + F_3 \quad (F_4 \text{ and } F_3 \text{ are new})$$
$$F_4 = F_3 + F_2 \quad \text{and} \quad F_3 = F_2 + F_1 \quad (F_2 \text{ and } F_1 \text{ are new})$$
$$F_2 = F_1 + F_0 \quad \text{and} \quad F_1 = F_0 + \ldots?$$

Hold on a second! Looking back at the definition of the Fibonacci numbers, there was a part that says, direct quote: "with base defined values for the first two Fibonacci numbers $F_0 = F_1 = 1$". We have explicit numerical values for F_1 and F_0 that we already know! This means that we then also know $F_2 = F_1 + F_0 = 1 + 1 = 2$! And then since we now know the exact values of F_0, F_1 and F_2, we can then calculate the exact value of F_3, as well! Following this same pattern back from F_0 and F_1 towards F_7, we have:

$$F_0 = 1 \quad \text{and} \quad F_1 = 1$$
$$F_2 = F_1 + F_0 = 1 + 1 = 2$$
$$F_3 = F_2 + F_1 = 2 + 1 = 3$$
$$F_4 = F_3 + F_2 = 3 + 2 = 5$$
$$F_5 = F_4 + F_3 = 5 + 3 = 8$$
$$F_6 = F_5 + F_4 = 8 + 5 = 13$$
$$F_7 = F_6 + F_5 = 13 + 8 = 21$$

And there we have it, $F_7 = 21$ has been successfully computed, and we did it all using *recursion*. Let us summarize the algorithm that we just used, in words. We had a number n and we needed to compute F_n. To do that, we were given a recursive formula $F_n = F_{n-1} + F_{n-2}$, and we were also given two base cases of already known values for $F_1 = 1$ and $F_0 = 1$. In order to calculate F_n, we kept reducing the problem of calculating F_n to two problems of calculating F_{n-1} and F_{n-2}, over and over again, until reaching the point of recursion where the parameters of the problem were $n = 1$ and $n = 0$, for which we already have values $F_1 = 1$ and $F_0 = 1$. Once the recursion reached this point of being able to directly compute the result of the problem based on the given base cases, without the need to further apply recursion, then we followed the steps of recursion back. At every step back, we were filling in values that were previously unknown, allowing each recursive problem to be solved and report its result back to the parent-problem, for increasingly larger values of the parameter n. Eventually, the recursion backtracks all the way back to the first call that was tasked with computing F_n, now with results for F_{n-1} and F_{n-2} known and stored as numbers, allowing us to sum those two up and return their sum as the final result of F_n.

Given everything mentioned so far, we can directly write the following recursive function that calculates the n-th Fibonacci number:

This function is guaranteed to not get stuck in an infinite loop when called with a non-negative value of n, because the parameters of the recursive calls are lower than n by at least 1, and never more than 2. As the recursive calls get

```
Algorithm: Fib(n)
  Input: A positive integer n
  Output: The value of the n-th Fibonacci number, F_n
1 if (n == 0 or n == 1) then
2     return 1
3 else
4     return Fib(n - 1) + Fib(n - 2)
```

chained one after another in order to compute the result, every such call lowering the value of their parameters by 1 and 2, at some point the parameters will shrink down to 1 and 0, guaranteeing that we hit our base cases. We also cannot ever "overshoot" and go below 0 because the lowest non-base case is when $n = 2$, and $Fib(n-2)$ will then result in calling $Fib(0)$. Just as a side comment, notice that if we had some other number G_n defined as $G_n = G_{n-1} + G_{n-2} + G_{n-3}$, we would need to have 3 base cases, since the parameter in recursive calls can get reduced by up to 3.

When witnessing recursion for the first time, things can look very suspicious and it might not be immediately clear why code such as the one written above is correct, or how it is possible to simply assume that recursive calls of this function will correctly compute values of F_{n-1} and F_{n-2}. The answer to all the listed questions is *mathematical induction*, which will be described in the next two chapters.

2 Structure of Recursion

In Introduction, we saw an example of recursion (both through a mathematical definition and through a recursive function) used to calculate Fibonacci numbers, which hints at what might be two main parts that are required for a complete definition of a recursive algorithm – the *base of recursion* and the *recursive relation*.

Base of recursion (often simply called the **base case**) represents the basic cases for which there are already-defined values and which do not need to be computed (at least not recursively). In the specific example of Fibonacci numbers, the basis of recursion are values $F_0 = 1$ and $F_1 = 1$.

Recursive relation represents the description of the solution through recursive calls of the same function (thereby meaning that it solves the exact same kind of problem), used to calculate values that are not covered by the base of recursion. In the specific example of Fibonacci numbers, the recursive relation is given by the formula $F_n = F_{n-1} + F_{n-2}$. We say that a recursive call of the same algorithm on a subset of the input data (or on smaller values of input parameters) solves a *subproblem* of the given problem. Therefore, the recursive relation represents the way of solving the problem through solving subproblems and then combining them into the final result for the complete problem.

Given a pair of the recursive basis and the recursive relation, we call it the *recursive formula*. For the specific problem of calculating the n-th Fibonacci number, the recursive formula is (comparing this formula to the recursive code of the function *Fib* given above, we see that the code is simply a direct implementation of the formula):

$$F_n = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ F_{n-1} + F_{n-2}, & \text{otherwise} \end{cases}$$

Having described the structure of recursion, we will examine why it actually works in the next chapter.

3 Mathematical Induction

Those familiar with the method for proving claims in mathematics that is called *mathematical induction* may notice that the recursive formula is actually identical to mathematical induction. Before we go back to recursion, let us briefly recap what mathematical induction is and what it is useful for.

Mathematical induction is one of multiple ways of proving claims in mathematics, used for natural numbers or objects that can be mapped to unique natural numbers in some manner. Proof of correctness via mathematical induction consists of proving correctness of the following two of its components:

1. **Basis:** proof that the claim is true for some small natural number, such as 1.

2. **Inductive step:** proof that the claim is true for a number n that falls outside of the base case, under the assumption that it is already true for $n-1$.

Thus we have two things to prove (the base case and the inductive step), and we are allowed to make one assumption – that the claim is already true for a number $n-1$. The reason that we are allowed to make the assumption that the claim is true for $n-1$ and use that to help us prove that it is also true for n, is that this follows from the correctness of the base case. Let us analyze why. If the basis is correct, then we know that the claim is true for the number $n=1$. If someone gave us a correct proof of the inductive step, but one that relies on the claim being true for $n-1$, then we could apply that proof to show that the claim is true for $n=2$. Why? Because when $n=2$, then $n-1=1$, and we already know that the claim is true for $n=1$ because it is covered by the base case. Thus, the claim being true for the base case means that the assumption that it is true for $n-1$ holds when $n=2$, and thus the claim is true for $n=2$. Now, knowing that the claim is true for $n=2$, that means that the assumption that the claim is true for $n-1$ holds when $n=3$, and thus the claim holds for $n=3$. You can see that we can apply this same reasoning indefinitely, step by step. This is why in the proof for the inductive step, we can safely assume that the claim is true for $n-1$ (or really any number between n and the base case), as long as the base case is correct, and as long as that assumption leads to a correct proof for the inductive step.

The above shows the importance of the basis of induction – the correctness of the claim for every number that is bigger than the basis relies on the claim being true for the basis. That is, the claim is correct for n because we assume that it is correct for $n-1$ (assuming, of course, that the inductive step itself has been proven correct). For $n-1$ it is correct because we assume that it is correct for $n-2$, and so on all the way down to $n=2$, where we no longer need to assume but we instead *know* that the claim is indeed true for $n-1=1$ because that case is trivially verifiable. This methodology of backtracking the inductive step all the way down to the base case is actually identical to recursion that we saw in the example of Fibonacci numbers!

In the inductive step, it is sometimes useful to assume that the claim is true for not only $n-1$ but also for all numbers smaller than n. These two assumptions can be used interchangeably in mathematical induction because if the claim is true for $n-1$, then we know that it is also true for $n-2$ and $n-3$ and so on. Other way round, if the claim is true for all numbers smaller than n, then it is also true for $n-1$.

Now let us look at a few examples of proving claims via mathematical induction.

Claim 1. *Every natural number greater than* 1 *is either a prime number or a product of one or multiple prime number factors.*

Proof. **Base case.** For $n=2$, the claim is trivially true because 2 is a prime number.

Inductive step. Let us assume that the claim is true for all numbers smaller than n, and let us now prove that it leads to the claim being also true for n. Number n has to be either a prime or a composite number. In the case that it is a prime number, then the claim is obviously true. In the case that it is not a prime number, it is a composite number and can be written as a product of two numbers a and b which are both greater than 1 and smaller than n.

$$n = a \cdot b \quad (n > a, b > 1)$$

Since a and b are smaller than n, the claim is true for them (based on the assumption of the inductive step), *i.e.* a and b are either prime numbers or can each be written as a product that has prime numbers in it. Regardless, since n is a product of a and b, it follows that n is a product of prime factors of a and b, and hence itself a product of prime factors. Therefore, the claim is true for n in the composite case as well. □

To demonstrate the above proof, let us test it on the first few natural numbers.

- **n = 2** The claim is true because 2 is a prime.
- **n = 3** The claim is true because 3 is a prime.
- **n = 4** The claim must be true because $4 = 2 \cdot 2$, and we know that the claim is true for number 2.
- **n = 5** The claim is true because 5 is a prime.
- **n = 6** The claim must be true because $6 = 2 \cdot 3$, and we know that the claim is true for numbers 2 and 3.

Continuing in this fashion, we can see through specific examples that the claim is true for every natural number that we try.

Claim 2. *The sum of all natural numbers from* 1 *to n is equal to* $\frac{n\cdot(n+1)}{2}$.

Proof. **Base case.** For $n = 1$, the claim is true since $\frac{1\cdot(1+1)}{2} = \frac{1\cdot 2}{2} = 1$, which is indeed the sum of all natural numbers from 1 to 1.

Inductive step. Let us assume that the claim is true for all natural numbers smaller than n. The sum of all numbers from 1 to n can be written as the sum of numbers from 1 to $n-1$ with n added to it, *i.e.*

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$$

Based on the assumption that the claim is true for all numbers smaller than n, we have:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)\cdot((n-1)+1)}{2} = \frac{(n-1)\cdot n}{2}$$

$$\implies n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)\cdot n}{2}$$

$$n + \frac{(n-1)\cdot n}{2} = \frac{2n + (n-1)\cdot n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n\cdot(n+1)}{2}$$

The last expression is identical to the expression in our claim, therefore proving the correctness for n. □

Let us now go back to the example of the recursive function which calculates the n-th Fibonacci number.

Algorithm: Fib(n)

Input: A positive integer n
Output: The value of the n-th Fibonacci number, F_n

```
1 if (n == 0 or n == 1) then
2     return 1
3 else
4     return Fib(n - 1) + Fib(n - 2)
```

The basis of recursion is the *if* block on lines 1–2, and the recursive relation is implemented in the *else* block on lines 3–4. Using mathematical induction as the method of proof, we conclude that line 4 returns the correct result outside of the base case (since it is identical to the definition of F_n in this simple example of ours), and we know that line 2 returns the correct result for the base case. We therefore conclude that our recursive algorithm is correct.

We will encounter more complex examples in further chapters when we use mathematical induction to prove algorithm correctness (not only for recursion).

4 Examples

In this section, we will go through some easy examples of application of recursion to algorithmic problem solving. The use of recursion is not actually necessary for the given examples; quite the contrary, most of them are solved very easily and elegantly through the use of simple loops. However, we will stubbornly use recursion to solve these simple problems because the aim of this section is to train the reader to think about solving problems in a recursive manner. Later on, we will explore problems for which solutions via recursion come very naturally.

The ability to easily construct recursive formulae to solve a particular given problem is necessary to understand algorithmic techniques of *divide & conquer* and *dynamic programming*. These techniques are used to implement some of the most efficient algorithms that are commonly used, and they are most easily and best understood in a recursive form.

4.1 Factorials

n-factorial (denoted $n!$) is defined as the product of all natural numbers from 1 to n, *i.e.* $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$. There is a special case for 0-factorial, which is defined as $0! = 1$ (the reason for it is that the definition of factorials comes from combinatorics, related to the number of permutations; specifically there is exactly one way to order 0 elements). To apply recursion to this problem, notice that the product of the first n natural numbers $n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$ can be split. Specifically, we can separately compute the product $(n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$, and then multiply the result of that with n in order to obtain $n!$. Recursively, n-factorial can be defined as the product of the number n and the $(n-1)$-th factorial, *i.e.* $n! = n \cdot (n-1)!$, with the base case of $0! = 1$. The recursive formula and the recursive function are then:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

Algorithm: Factorial(n)

```
    Input: A positive integer n
    Output: The value of n!
1   if (n == 0) then
2       return 1
3   else
4       return n * Factorial(n - 1);
```

Correctness of the algorithm is obvious (a trivial case of mathematical induction that follows straight from the definition).

4.2 Numeric Array Sum

The task is to compute the sum of numbers in an array $S = (s_1, s_2, \ldots, s_n)$. Similarly to the example of computing the factorial, we can split S into $(s_1, \ldots, s_{n-1})$ and s_n. Recursive relation then represents the sum of all elements of S as the sum of two numbers – the last element s_n and the sum of the first $n-1$ elements. The base case arises when S consists only of a single element, in which case the sum is equal to the single element itself. Showing correctness is trivial in this example as well.

$$\sum_{i=1}^{n} s_i = \begin{cases} s_1, & \text{if } n = 1 \\ s_n + \sum_{i=1}^{n-1} s_i, & \text{otherwise} \end{cases}$$

Algorithm: arraySum(S)

```
    Input: An array of numbers S = (s_1, ..., s_n)
    Output: The sum of all numbers in S
1   if (n == 1) then
2       return s_1
3   else
4       return arraySum((s_1, ..., s_{n-1})) + s_n
```

4.3 The Maximum Element

The task is to find the value of the maximum element in an array $S = (s_1, s_2, \ldots, s_n)$. Recursive relation represents the solution as the maximum of two values – the last element s_n and the maximum value from among the first $n-1$ elements.

$$\text{setMax}(S) = \begin{cases} s_1, & \text{if } n = 1 \\ \max(s_n,\ \text{setMax}((s_1, s_2, \ldots, s_{n-1}))), & \text{otherwise} \end{cases}$$

Algorithm: setMax(S)

Input: An array of numbers $S = (s_1, s_2, ..., s_n)$
Output: The maximum element of the array S

```
1 if (n == 1) then
2     return s_1
3 else
4     return max( setMax((s_1, ..., s_{n-1})), s_n )
```

4.4 Number of Trailing 0-digits of a Factorial

Let us now try a less trivial task – the aim is to count the number of 0 digits at the end of the value of n-factorial. In order to gain some intuition about how to solve the problem, let us analyze what the result of this problem looks like in the general case. Every natural number x can be written as the product of some power of number 10 and the remainder, *i.e.* $x = 10^a \cdot \frac{x}{10^a}$ (this is true for every non-zero number a). For example, $1250 = 125 \cdot 10^1$, $10000 = 1 \cdot 10^4$, $7500 = 750 \cdot 10^1$ and $123 = 123 \cdot 10^0$. The reason that we are focusing on the number 10 is that the number of 0 digits at the end of a number x is equal to the largest power of 10 (let us denote that by A) that we can "pull out of" x as a factor such that $\frac{x}{10^A}$ remains an integer.

However, it is not enough to just find the largest power of 10 contained as a factor for all numbers from 1 to n! In the example of $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$, there is a 0 digit at the end even though none of the numbers have 10 as their factor. However, some numbers do have 2 and 5 as their factors, which make 10 when multiplied together. Therefore, for every trailing 0 digit of $n!$, there must exist a pair of numbers 2 and 5 as its factors. In fact, since the number 5 is greater than 2, then the product of all numbers from 1 to n has to contain more *twos* than *fives*. Therefore, it suffices to count *fives*, as we can be certain that each of them will have an available factor 2 to be multiplied with.

Recursive relation of this problem represents the solution as the sum of the number of factors 5 of n and the number of factors 5 of $(n-1)!$. To find the number of factors 5 in a given number, we write an additional function *numFives*, which we also implement through recursion.

$$\text{FactFives}(n) = \begin{cases} 0, & \text{if } n = 0 \\ \text{FactFives}(n-1) + \text{NumFives}(n), & \text{otherwise} \end{cases}$$

Algorithm: FactFives(n)

Input: A non-negative integer n
Output: The number of trailing zeroes in n-factorial

```
1 if (n == 0) then
2     return 0
3 else
4     return FactFives(n − 1) + NumFives(n)
```

$$\text{NumFives}(n) = \begin{cases} 0, & \text{if } 5 \nmid n \\ 1 + \text{NumFives}(n/5), & \text{otherwise} \end{cases}$$

Algorithm: NumFives(n)

Input: A non-negative integer n
Output: The number of times that n can be divided by 5

```
1 if (n % 5 != 0) then
2     return 0
3 else
4     return 1 + NumFives(n/5)
```

4.5 Array Sorting

We are given an array $S = (s_1, s_2, ..., s_n)$ whose elements we need to reorder in a way that every consecutive element is greater than or equal to the previous, *i.e.* find a permutation of S, $(s_{i_1}, s_{i_2}, ..., s_{i_n})$ $i_j \in \{1, 2, ..., n\}$, such that $s_{i_k} \leq s_{i_{k+1}} \forall k$. We will solve this task through a recursive implementation of *selection sort*, which iteratively identifies the smallest element from the unsorted portion of the array and moves it to the end of the sorted portion. A demonstration of the first few iterations of *selection sort* on a sample array is given below.

Array S:	$S = (5, 3, 2, 7, 0, 2, 9, 13, 1)$
The minimum unsorted element is 0.	$S = (\ \vert\ 5, 3, 2, 7, \underline{\mathbf{0}}, 2, 9, 13, 1)$
We move it into the sorted portion.	$S = (0\ \vert\ 5, 3, 2, 7, 2, 9, 13, 1)$
The minimum unsorted element is 1.	$S = (0\ \vert\ 5, 3, 2, 7, 2, 9, 13, \underline{\mathbf{1}})$
We move it into the sorted portion.	$S = (0, 1\ \vert\ 5, 3, 2, 7, 2, 9, 13)$
The minimum unsorted element is 2.	$S = (0, 1\ \vert\ 5, 3, \underline{\mathbf{2}}, 7, 2, 9, 13)$
We move it into the sorted portion.	$S = (0, 1, 2\ \vert\ 5, 3, 7, 2, 9, 13)$
The minimum unsorted element is 2.	$S = (0, 1, 2\ \vert\ 5, 3, 7, \underline{\mathbf{2}}, 9, 13)$
We move it into the sorted portion.	$S = (0, 1, 2, 2\ \vert\ 5, 3, 7, 9, 13)$
The minimum unsorted element is 3.	$S = (0, 1, 2, 2\ \vert\ 5, \underline{\mathbf{3}}, 7, 9, 13)$
We move it into the sorted portion.	$S = (0, 1, 2, 2, 3\ \vert\ 5, 7, 9, 13)$
etc.	

Recursive relation for *selection sort* returns an ordered pair; the first position of which is occupied by the minimum element of the array, and the second position occupied by the sorted remainder of the array.

$$\text{selectionSort}(S) = \begin{cases} s_1, & \text{if } |S| = 1 \\ (\text{setMin}(S), \text{selectionSort}(S - \{\text{setMin}(S)\})), & \text{otherwise} \end{cases}$$

In the function which implements the recursive formula, we use an auxiliary function *setMinIdx* which returns the index of the minimum element in a given array. Once we have the index, we put the element with that index to the beginning of the array with a position swap. Then we recursively call the function on the unsorted remainder of the array.

Algorithm: selectionSort(S)

Input: An array of numbers $S = (s_1, ..., s_n)$
Result: Elements of S are rearranged in a sorted order, such that $\forall i : s_i \leq s_{i+1}$

```
1 if (n > 1) then
2     idx = setMinIdx(S)
3     swapPosition(s_1, s_idx)
4     selectionSort((s_2, ..., s_n));
```

The recursive implementation of *setMinIdx* is very similar to the recursive implementation of *setMax* (requiring the change in order to return the index, rather than the value of the maximum element), so we will skip that.

It is important to note that our current implementation of *selection sort* which moves the minimum element to the front via a position swap results in *unstable sorting* because it can change the relative position of elements that have the same values of their sort keys. By shifting the minimum element to the front through continuous swaps with the element to the left of it, we preserve the relative order of all the elements in the unsorted portion and obtain a *stable sort*. We will implement such a function recursively as well; let us call it *shiftToFront*. As its parameter, it takes the index of the element that we wish to shift to the front.

Algorithm: shiftToFront(S, idx)

Input: An array of numbers $S = (s_1, ..., s_n)$ and an index idx
Result: Elements of S are shifted so that the element that was initially at index idx is now at the front of the array

```
1 if (idx > 1) then
2     swapPosition(s_{idx-1}, s_{idx})
3     shiftToFront(S, idx − 1)
```

Algorithm: stableSelectionSort(S)

Input: An array of numbers $S = (s_1, ..., s_n)$
Result: Elements of S are rearranged in a stable sorted order, such that $\forall i : s_i \leq s_{i+1}$

```
1 if (n > 1) then
2     idx = setMinIdx(S)
3     shiftToFront(S, idx)
4     stableSelectionSort((s_2, ..., s_n));
```

5 Problems With Using Recursion

Examples of problems that we solved in previous sections do not actually require recursion. In fact, it would be easier (less lines of code) to solve them using iterative algorithms. *Selection sort* and *FactFives* can be implemented with two nested loops, while the rest of the sample problems can be easily solved with just a single loop.

Aside from often resulting in more lines of code (not always the case), an important problem with recursion is that it uses additional memory resources and slows down the execution of the code. Every time a function is called, new variables get assigned and placed on the memory stack – the function's parameters and local variables. Recursive function calls are no exception to that rule. As an example, in the case of the recursive function which calculates n-factorial, there will be n different copies of the variable n by the time the recursion reaches the base case; and in the case of the sum of elements of an array, there will be n different copies of both the variable n and the pointer to the array S.

In the case of the recursive function that calculates the n-th Fibonacci number (from the Introduction), things look even worse because the function is recursively called twice; additionally, the first recursive call later recursively calls the function with the same parameter as the second recursive call. This is perhaps best understood on a graphic example

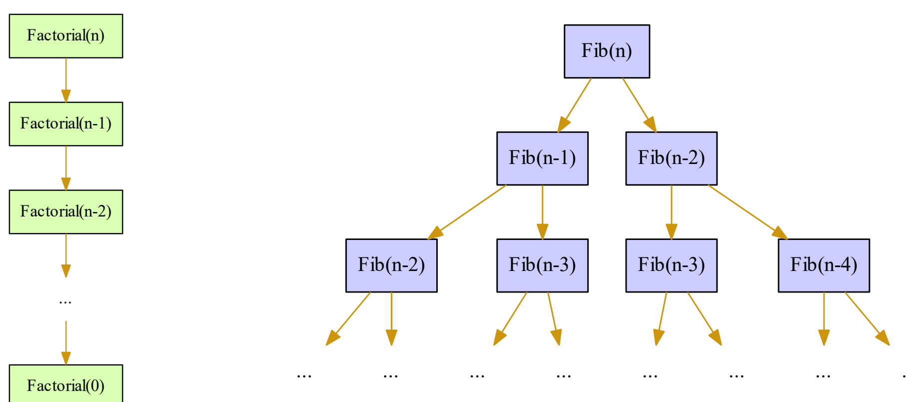


Figure 1: Trees of recursive function calls of *Factorial* and *Fib*.

such as on Figure 1, which shows recursive trees[1] of functions *Factorial* and *Fib*. It has the first call as its root node, and each node has an outgoing edge towards its children, one for each recursive call it makes.

The **left side** of the Figure depicts recursive calls of the function *Factorial*. The root represents the call with parameter n. Since the function *Factorial* makes only a single recursive call, the root has only one child – the node with parameter $n-1$. In turn, the only child of that node is the node with parameter $n-2$ *etc*. In the end, we reach the base case which has the parameter 0. The base case returns as its result the value 1 into node *Factorial*(1); that node returns as its result the value of $1 \cdot 1$ into node *Factorial*(2); that node returns as its value $1 \cdot 1 \cdot 2$ into node *Factorial*(3) *etc*. The recursive tree of the function *Factorial* has a linear structure and it is not possible to create an asymptotically faster iterative algorithm.

The **right side** of the figure depicts recursive calls of the function *Fib*. There we immediately notice that *Fib* often repeats its calls with the same parameter values, *i.e.* we calculate the same number F_n (for various values of n) multiple times. We can see that just in two first tree levels below the root, we call function *Fib* with parameter $n-3$ twice, while *Fib*$(n-2)$ also calls it once again in the level below. Parameter $n-4$ is used in a total of five recursive calls on levels 3 and 4. In fact, due to these repetitions, there will be exponentially many recursive calls of *Fib*. We revisit this specific problem of a large overlap of recursive subproblems in the chapter on *dynamic programming*, which is an algorithmic technique whose one characteristic is that repetitions of identical work are eliminated.

In the case of Fibonacci numbers, we can actually very easily write an iterative version which calculates the n-th Fibonacci numbers in linear time and constant space.

Algorithm: FibIterative(n)

Input: A non-negative integer n
Output: The n-th Fibonacci number

```
1 last = 1
2 beforeLast = 1
3 for (i = 2...n) do
4     temp = last + beforeLast
5     beforeLast = last
6     last = temp
7 return last
```

Given everything written so far where we focused entirely on the drawbacks of recursion, one would be right to ask whether we should ever even use recursion. In fact, every recursive algorithm can be written without recursion, using loops and the *stack* data structure to "unwind the recursion" by simulating the memory stack. Nevertheless, that almost always results in ugly code which is harder to debug, maintain and update.

Probably the strongest argument for using recursion is that, for some problems, recursion represents the most natural approach to their solving. The best examples of that are algorithms that use the *divide & conquer* approach, such as *quick sort* or *merge sort*; or functions that operate on tree structures (binary trees, segment trees *etc*) and general graphs. Even though it is possible to write iterative versions of such algorithms in some cases without using stacks, it is far more complicated to do so compared to writing simple recursive functions. Nowadays, it is also generally unnecessary since contemporary compilers can remove some forms of recursion (transform it into iterative form with stacks) automatically in their optimized compilation mode. Thus, we get both the beauty and simplicity of writing recursive functions (where applicable) and the optimized running time thanks to modern compilers.

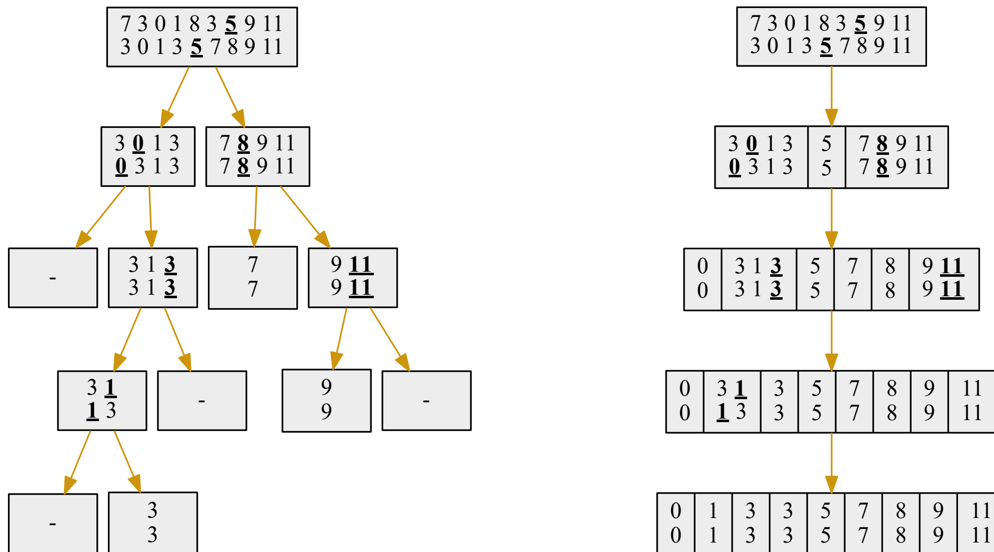
The earlier example of calculating the n-th Fibonacci number represents an extreme case of inefficiency of recursion, but the reason for it is not inherently in recursion as an algorithmic technique, but rather naive direct implementation of the algorithm from the mathematical definition of Fibonacci numbers. It is thus somewhat ironic that the fastest known solution to the problem of calculating the n-th Fibonacci number (a problem that is used as a notorious example of the inefficiency of recursion) actually has a recursive form – through raising a matrix to the n-th power in $O(\log n)$ time via techniques of linear algebra. Exponentiation (of either numbers or matrices) is a problem which is asymptotically most efficiently solved through the *divide & conquer* technique, which is based on recursive thinking. More on this later.

[1] A recursive tree is a structure for graphical representation of calls of recursive functions

6 Examples of Useful Recursion

After presenting silly problem examples where recursion does not naturally fit, and scaring everyone with worst-case scenarios where recursion leads to exponential running time, it is time to give a few examples of problems for which recursion comes naturally, and efficiently solves them.

6.1 Quicksort



(a) Forward pass of the recursion – selecting pivots and partitioning the arrays. Calls on empty partitions are indicated by nodes labeled with "-". Pivots are not included in recursive calls.

(b) Status of the whole array at each level of recursion. The last level of recursion results in a fully sorted array.

Figure 2: An example of the execution of Quicksort on an unsorted array $(7, 3, 0, 1, 8, 3, 5, 9, 11)$. The left side shows the recursion tree (nodes where the array has only 1 element are not shown), and the right side shows the state of the whole array at each level of recursion. Randomly picked elements that are used for partitioning are underlined and in bold.

We have previously seen an example of a recursive implementation of Selection sort, which has an asymptotic running time of $\Theta(n^2)$. Here, we will look at Quicksort, which has an expected running time of $\Theta(n \log n)$, even if its worst-case running time is $O(n^2)$. We will not be proving its expected running time here, so if you are interested in that, look up material dedicated to divide & conquer algorithms.

The idea of Quicksort is very simple – pick an element of the array at random, and divide the rest of the array into two parts, one containing all elements smaller than or equal to the picked element, and the other part containing all elements larger than the picked element. We then rearrange the elements of the original array so that all elements from the first part are at the beginning, followed by the picked element, and after which we place the elements from the second part. While this process does not result in a sorted array, it results in the picked element being placed precisely where it would be if the array was entirely sorted. After all, all the elements preceding it are smaller than it or equal to it, and all elements following it are larger than it, so this must be its true sorted position.

The partitioning process that we described so far has nothing to do with recursion. However, we can now use recursion to repeat the same process on the two parts of the array that we created, with the base case being when the

array consists of only a single element, and this will eventually result in a fully sorted array. Let us examine why this works by looking at the facts of the partitioning process:

1. An element is picked at random and placed at its correct sorted position
2. We are left with two parts of the array which are always smaller than the current array size by at least 1, since at the very least we took out the picked element

Property (2) guarantees that the recursive subproblems are always smaller and will thus eventually become small enough to be considered a base case, while the property (1) guarantees that we will eventually end up with a fully sorted array if we keep repeating this process on all the unsorted parts. Figure 2 shows an example of the execution of Quicksort on an unsorted array $(7, 3, 0, 1, 8, 3, 5, 9, 11)$. The left side of the figure shows the recursion tree (nodes where the array has only 1 element are omitted), and the right side shows the state of the whole array at each level of recursion. Randomly picked elements that are used for partitioning are underlined and in bold. Each node of the recursion tree in the figure consists of two boxes; the upper one representing the array as given to the current recursion call, and the lower one representing the state of the given array after the partitioning process has been completed.

If we denote the chosen element used to partition the array S as p, the portion of the array that contains elements smaller than or equal to it as P_1, and the portion of the array that contains elements larger than it as P_2, the following would be the recursive formula for Quicksort:

$$\text{QuickSort}(S) = \begin{cases} (s_1), & \text{if } |S| = 1 \\ (\text{QuickSort}(P_1), p, \text{QuickSort}(P_2)), & \text{otherwise} \end{cases}$$

A recursive function which implements Quicksort (as an unstable sort) would look as follows:

Algorithm: QuickSort(S)

Input: An array of numbers $S = (s_1, ..., s_n)$
Result: Elements of S are rearranged in a stable sorted order, such that $\forall i : s_i \leq s_{i+1}$

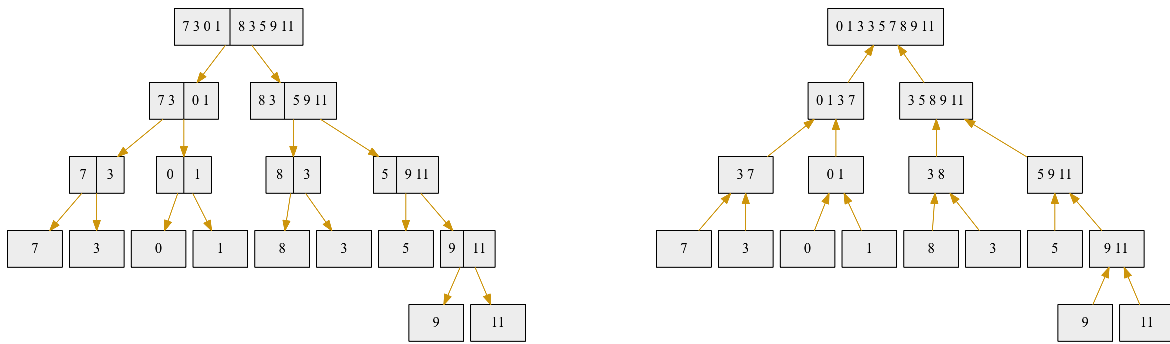
```
1  if (n > 1) then
       /* Partition                                         */
2      randIdx = randomInteger(1, n)
3      swapPosition(s_1, s_randIdx)
4      numSmaller = 0
5      for (i = 2...n) do
6          if (s_i < s_1) then
7              swapPosition(s_i, s_{2+numSmaller})
8              numSmaller++
9      swapPosition(s_1, s_{1+numSmaller})
       /* Do recursion                                      */
10     QuickSort({s_1, ..., s_numSmaller})
11     QuickSort({s_{numSmaller+2}, ..., s_n})
```

As we can see, most of the written code is for handling the partition process, while the recursion itself is quite simple and elegant. In fact, there is no way to write Quicksort without recursion other than directly simulating recursion via stacks and while loops, but such code is neither simpler nor easier to understand and debug, and there is no benefit to the asymptotic running time.

6.2 Merge Sort

Now we are going to look into another recursive sorting algorithm, Merge Sort, which has a running time of $\Theta(n \log n)$. Similar to Quicksort, it comes from the divide & conquer family of algorithms, where the general process is to split the problem into two parts without overlap, solve them recursively, and then typically to combine the solutions to the two subproblems into a solution to the overall problem through a post-processing step[2]. While Merge Sort is neither

[2]Quicksort is out of the ordinary here, as it has no post-processing step. Instead, Quicksort has a pre-processing step in the form of the partitioning of the array, which makes it unnecessary to do any post-processing



(a) Forward pass of the recursion – splitting the problems in half

(b) Backward pass of the recursion – merging the sorted halves into the fully sorted array

Figure 3: An example of the execution of Merge Sort on an unsorted array $(7, 3, 0, 1, 8, 3, 5, 9, 11)$. The left side shows the recursion tree during the splitting procedure, with all the resulting recursive calls that are made. The right side then shows the merging of the calls as the recursion returns from the leaves of the tree all the way back up to the root call.

faster than Quicksort in the expected case, nor practically faster than another $\Theta(n \log n)$ sorting algorithm known as Heapsort[3], we are going to study it here because it represents a different kind of recursion compared to Quicksort. It will become clear why we care about this distinction in the following Section.

The idea of Merge Sort is very simple – simply split the array of size n into two equal halves and sort them independently of each other using recursion. As a result of that, we end up with two sorted arrays of size $\frac{n}{2}$. Merging these two arrays so that it produces a fully sorted array turns out to be rather trivial. A way to create a sorted array from them, is to always select the smallest element from both of the subarrays, and place it as the last element of the fully sorted array until we deplete both subarrays of all their elements. The extraction of the minimum element is trivial when the two smallest remaining elements are always at the front of the two subarrays, since they are each fully sorted, and a simple comparison between the two front elements suffices. The recursive formula and the recursive function are given by:

$$\text{MergeSort}(S) = \begin{cases} s_1, & \text{if } |S| = 1 \\ \text{merge}(\text{MergeSort}(S[1..\frac{|S|}{2}]), \text{MergeSort}(S[\frac{|S|}{2} + 1..|S|])), & \text{otherwise} \end{cases}$$

Figure 3 shows an example of the execution of Merge Sort. Once the recursion reaches a leaf (an array that is just a single element), it starts to return the results back to the parent call, at which point merging begins. Each node merges the results obtained by its two children nodes and returns it back to its own parent until the recursion returns all the way back to the root call on the whole array. Since we always split the array right down the middle in each recursive call, there are exactly $\Theta(\log n)$ levels of recursion. From the recursion tree, we can see that on each level of recursion there is partial merging done on all n elements of the array (the only exception to this is the very last level when n is not a power of 2, on which there is no merging being done). The total number of times that a minimum element is selected on a recursion level is thus n, and thus each level of recursion spends a total of $\Theta(n)$ time to complete all the merges. The total running time is therefore $\Theta(n \log n)$.

[3] Heapsort is essentially a faster Selection Sort, with the difference that the extraction of the minimum element is not done via a for loop in $\Theta(n)$ time, but rather using a *Heap* data structure which allows that to be done in $O(\log n)$ time

```
Algorithm: MergeSort(S)
  Input: An array of numbers S = (s_1, ..., s_n)
  Result: Elements of S are rearranged in a stable sorted order, such that ∀i : s_i ≤ s_{i+1}
1  if (n > 1) then
      /* Do recursion                                                    */
2     mid = n/2
3     MergeSort((s_1, ..., s_mid))
4     MergeSort((s_mid+1, ..., s_n))
      /* Merging                                                         */
5     temp = new array of size n
6     i = 1, j = mid + 1, k = 1
7     while (i ≤ mid and j ≤ n) do
8        if (s_i < s_j) then
9           temp_k = s_i
10          i++, k++
11       else
12          temp_k = s_j
13          j++, k++
14    while (i ≤ mid) do
15       temp_k = s_i
16       i++, k++
17    while (j ≤ n) do
18       temp_k = s_j
19       j++, k++
20    for (i = 1...n) do
21       s_i = temp_i
22    delete temp
```

6.3 Exponentiation

Exponentiation is the problem of raising a number n to the power of k (to keep matters simple, we will assume that k is a non-negative integer). A straightforward way to solve this problem with a for loop could be something like this:

```
Algorithm: linearExponentiation(n, k)
  Input: Two non-negative integers n and k
  Output: The result of raising n to the power of k
1  r = 1
2  for (i = 1...k) do
3     r *= n
4  return r
```

This algorithm has running time of $\Theta(k)$. Can we do better? Turns out that, yes, we indeed can, because squaring a number is the same as doubling its exponent, i.e. $(n^k)^2 = n^{2k}$. Thinking about this in reverse, if k is an even number, then if we compute $n^{\frac{k}{2}}$, and multiply that result with itself, we will obtain the solution to n^k. As for the case when k is an odd number, we will simply reduce it to computing n^{k-1} and then multiply the result of that with n to obtain n^k.

The recursive formula for this algorithm is then given by:

$$\text{fastExponentiation}(S) = \begin{cases} 1, & \text{if } k = 0 \\ n \cdot \text{fastExponentiation}(n, k-1), & \text{if } k \text{ is an odd number} \\ (\text{fastExponentiation(n, k/2)})^2, & \text{otherwise} \end{cases}$$

And the resulting recursive function is:

Algorithm: fastExponentiation(n, k)

Input: Two non-negative integers n and k
Output: The result of raising n to the power of k

```
1 if (k == 0) then
2     return 1
3 else if (k % 2 == 1) then
4     return n * fastExponentiation(n, k − 1)
5 else
6     r = fastExponentiation(n, k/2)
7     return r * r
```

For the running time analysis, the best case scenario would be if k was a power of 2, so it can keep getting halved all the way down to 1, in which case the running time is $\Theta(\log k)$. This cannot happen if k is an odd number, but note that if k is an odd number, then $k-1$ is an even number. Thus, every time after we reduce the exponent by merely 1, we can instantly halve it right after, and so the total number of times that k can be odd in the worst case is equal to the number of times that we could halve the number. We can then conclude that the total running time is then $\Theta(\log k)$.

6.3.1 Fast Fibonacci

Having just seen how to perform logarithmic-time exponentiation, let us revisit the very first recursive problem introduced in this material, which is the problem of computing the n-th Fibonacci number F_n. At the end of Section 5, it was mentioned that there is a logarithmic running-time solution to computing F_n by using matrix equations. So what was that about?

If we wish to present the relation $F_n = F_{n-1} + F_{n-2}$ in the form of a product of matrices, we can have something like this:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

This is not the only way to do it; we could have just as easily used the following equation:

$$\begin{bmatrix} F_n \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

However, notice that the first equation allows expanding the product in a "recursive" manner, while the second one does not. This is because both the matrix on the left hand side and the right matrix on the right hand side of the first equation have the same form, with the only difference being that the indices on the right hand side are lower by 1. Thus, we can expand it like this:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix}$$

At this point, it is likely that you can see where this is going. We can go all the way until we end up with the following:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \cdots \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

So we can compute F_n by simply raising a 2×2 matrix to the power of $n-1$, and adding up the two numbers in the first row. How lucky we are that we just learnt how to do efficient exponentiation! The algorithm to achieve matrix exponentiation is identical to the one for the exponentiation of real numbers, the only difference being that the base case returns an identity matrix I_2 instead of the number 1, and the multiplication in the return statements has to follow the rules for the multiplication of matrices. Since the multiplication of two 2×2 matrices is done by performing 8 scalar multiplications and 4 additions, which takes $O(1)$ time, the overall running time is still just $\Theta(\log n)$.

7 Removing Recursion

In this Section, we will look into how to rewrite a recursive algorithm so that it does not perform recursive calls. While this should generally not be a concern in most use cases if you are writing in a modern programming language, there may be some domain-specific restrictions that either prevent you from using a programming language that supports recursion, or the hardware resource constraints for your problem are so strict that every byte counts and recursion is simply not an option. If you find yourself in such a situation, you will need to know how to remove recursion. The specific way to perform such a task will greatly depend on which recursive algorithm you are trying to rewrite, and how and where within the function the recursive calls are made, so we are going to go through a number of examples of different kinds of recursion and how to deal with it.

First, let us revise how function calls work in general. When your application makes a function call, the execution of the current function is suspended (assuming no threading or parallel computing) until the called function returns. Parameters for the called function are placed on the memory stack, variables are created from those parameters, and instructions are then executed from the start of the called function. Once the called function finishes, it places its return value on the memory stack, variable assigned to capture that return value in the parent function is assigned the returned value from the stack, and the execution continues from the next instruction in the parent function.

Now let us begin with the simplest case of recursion in terms of how easy it is to remove it – *tail recursion*. Tail recursion is the term we use to describe a recursive function whose very last action is a recursive call to itself, and there is no post-processing. The best example of a natural recursive algorithm that also represents tail recursion is Quicksort, in which all the processing is done at the start during the partitioning stage and the two recursive calls are done at the very end. As such, a function call of Quicksort does not need to know what the results of the execution of its children's function calls are. It already did its part of the job, and now it's up to its children calls to do theirs and finish the job. Thus, to simulate this tail recursion, we simply use a stack data structure which we fill with all intervals of the array that still need to be processed by our QuickSort code. The code that achieves this is rather straightforward:

Algorithm: QuickSortWithoutRecursion(S)

Input: An array of numbers $S = (s_1, ..., s_n)$
Result: Elements of S are rearranged in a stable sorted order, such that $\forall i : s_i \leq s_{i+1}$

```
1   st = stack of pairs of integers
2   st.push( (1, n) ) ;                              // Insert initial array bounds
3   while (st not empty) do
4       (a, b) = st.top
5       st.removeTop
6       if (a < b) then
            /* Partition the array within current bounds                     */
7           randIdx = randomInteger(a, b)
8           swapPosition(s_a, s_randIdx)
9           numSmaller = 0
10          for (i = a + 1...b) do
11              if (s_i < s_a) then
12                  swapPosition(s_i, s_{a+1+numSmaller})
13                  numSmaller++
14          swapPosition(s_a, s_{a+numSmaller})
            /* Push new bounds onto the stack                                */
15          st.push( (a, a + numSmaller - 1) )
16          st.push( (a + numSmaller + 1, b) )
```

Another special case of easy-to-remove recursion is "linear" recursion, in which there is only a single recursive call, and the post-processing procedure is very simple. An example of this would be the recursive function that computes n-factorial. The last action of this kind of recursive function is not the recursive call itself, but the multiplication of the recursive call's result with n, and as such it does not represent tail recursion. However, we can easily transform this kind of recursion into tail recursion by passing the current result to its children as a parameter. As a result, it is not the first call that computes the result after all its descendants return their results, but rather it is the last call that computes

the final result. To illustrate this, let us look at a tail-recursive version of the n-factorial function:

```
Algorithm: FactorialTailRecursion(n, resultSoFar)
  Input: A non-negative integer n
  Output: The value of n-factorial
1 if (n == 0) then
2     return resultSoFar
3 else
4     return FactorialTailRecursion(n − 1, n * resultSoFar)
```

The result is obtained by calling $FactorialTailRecursion(n, 1)$. We see that the current product is kept in $resultSoFar$, which is passed on all the way down to the call that has $n = 0$, at which point the final result is simply returned all the way back to the root call, without any post-processing in parent calls. Since this code now represents a case of linear tail recursion, we can remove recursion similar to how we did it for Quicksort, with the exception that we now also need to actually return the final result, something that Quicksort did not require.

```
Algorithm: FactorialWithoutRecursion(n)
  Input: A non-negative integer n
  Output: The value of n-factorial
1 st = stack of pairs of integers
2 st.push( (n, 1) )
3 while (st not empty) do
4     (m, resultSoFar) = st.top
5     st.removeTop
6     if (m == n) then
7         return resultSoFar
8     else
9         st.push( (m − 1, m * resultSoFar) )
```

Finally, we now look at examples of the general case of recursion (with increasing complexity) where the recursive call is not the last action. An example of this kind of recursion would be the Merge Sort algorithm that we examined in the previous section. The last action in Merge Sort is the merging of the two sorted lists, after children calls have returned. However, before going into the more complicated Merge Sort, let us first look at a much simpler example of the standard recursive algorithm for n-factorial:

```
Algorithm: Factorial(n)
  Input: A non-negative integer n
  Output: The value of n-factorial
1 if (n == 0) then
2     return 1
3 else
4     return n * Factorial(n − 1)
```

To remove recursion from this function, we are going to utilize two additional things – a return value stack, and a concept of breakpoints within the function code. As the name indicates, we will use the return value stack to hold the values returned by recursive calls that parent calls will need for post-processing. Breakpoints are going to be used to help our program decide from which part of the function it should be executing code. To better understand why we need this, note that the body of the function is split into two parts – the part before the recursive call, and the part after the recursive call. The part after the recursive call cannot be executed until the recursive call has finished, at which point we can take the return value from the stack, and then continue its execution. This is illustrated in the following modification of the recursive function:

```
Algorithm: Factorial(n)
   Input: A non-negative integer n
   Output: The value of n-factorial
 1 if (n == 0) then
 2     return 1
 3 else
       /* Part 1 - before and including the recursive call                    */
 4     r = Factorial(n − 1)
       /* Part 2 - after recursion                                            */
 5     return n * r
```

Recursion-free code that simulates this would then be as follows:

```
Algorithm: FactorialWithoutRecursion(n)
   Input: A non-negative integer n
   Output: The value of n-factorial
 1 if (n == 0) then
 2     return 1
 3 st_p = stack of pairs of integers to store the n parameter and the breakpoint number
 4 st_r = stack of return values
 5 st_p.push( (n, 1) )
 6 while (st_p not empty) do
       /* Get the current parameters                                          */
 7     (m, bp) = st_p.top
 8     st_p.removeTop
 9     if (m == 0) then
10         st_r.push(1)
11     else
           /* Part 1 - before and including the recursive call                */
12         if (bp == 1) then
13             st_p.push( (m, 2) ) ;      // First push the call to go into the second breakpoint
                after "recursion"
14             st_p.push( (m − 1, 1) ) ;           // Then, over it, place the "recursive" call
           /* Part 2 - after recursion                                        */
15         else if (bp == 2) then
16             r = st_r.top st_r.removeTop if (m == n) then
17                 return m * r
18             else
19                 st_r.push( m * r )
```

Now that we have a general template for handling general-case recursion, we are going to do the same for Merge Sort. Note that we do not need to use a return value stack for Merge Sort, as it does not return any value.

```
Algorithm: MergeSortWithoutRecursion(S)
   Input: An array of numbers S = (s_1, ..., s_n)
   Result: Elements of S are rearranged in a stable sorted order, such that ∀i : s_i ≤ s_{i+1}
 1 temp = new array of size n
 2 st = stack of triplets of array bounds and the breakpoint number
 3 st.push( (1, n, 1) )
 4 while (st not empty) do
       /* Get the current parameters                                          */
 5     (a, b, bp) = st.top
 6     st.removeTop
 7     if (a < b) then
 8         mid = (a + b)/2
           /* Part 1 - Recursion                                               */
 9         if (bp == 1) then
               /* Before going into recursion, create a call for the second breakpoint  */
10             st.push( (a, b, 2) )
11             st.push( (a, mid, 1) )
12             st.push( (mid + 1, b, 1) )
           /* Part 2 - Merging                                                 */
13         else if (bp == 2) then
14             i = a, j = mid + 1, k = 1
15             while (i ≤ mid and j ≤ b) do
16                 if (s_i < s_j) then
17                     temp_k = s_i
18                     i++, k++
19                 else
20                     temp_k = s_j
21                     j++, k++
22             while (i ≤ mid) do
23                 temp_k = s_i
24                 i++, k++
25             while (j ≤ b) do
26                 temp_k = s_j
27                 j++, k++
28             for (i = a...b) do
29                 s_i = temp_{i-a+1}
30 delete temp
```

The examples provided so far should cover everything you need to know about how to rewrite your code to not use recursion. In case that your code is somehow even more complex than Merge Sort, perhaps by having multiple disjoint recursive calls between which there is some computing being done, you would simply create more breakpoints, so that you can control where the execution of your function continues after each stack reading.

Having learnt how to do it, a valid question is whether you *should* invest effort into removing recursion from your codebase. The answer for most people is going to be "no". If performance matters in your work so much, keep in mind that contemporary compilers are quite good in turning most of the recursive functions you write into tail recursion (if possible), and then optimizing it to not use recursion by using something similar to the above-described processes. Additionally, code being clean, intuitive, easy to understand, and easy to update and modify, is very important for longer-term code maintenance. All in all, advice is to keep using recursion where it naturally fits best.