ACADEMY

MODULE 5: ALGORITHMS

M.Sc. in Artificial Intelligence
IDEA Academy, Malta

Luke Collins[*]
luke.collins.mt

3 Nov 2022

Contents

[*]Department of Mathematics, Faculty of Science, University of Malta.

0. Introduction

My name is Luke Collins, and I will be teaching you this course on algorithms. The goal of this course is to endow you with the skills to distinguish between good and bad algorithms, to familiarise you with some classical algorithms on sorting and searching, graphs, path-finding, and network flow, and to teach you some standard techniques in algorithm design. The official reference book for the course is [1].

Just as with your previous modules, there will be a small discussion component in the middle of the course, and a programming assignment at the end, where you will apply what you've learned.

If you have any questions about the course, don't hesitate to contact me via email on `luke@collins.mt`, or talk to me after one of my lectures.

What is an Algorithm?

Lecture 1
22 Feb 2022

An algorithm is a precisely defined computational procedure that takes a number of *inputs* (possibly zero), and produces some desired *output*. It can be thought of as a function (in the mathematical sense) mapping inputs from some *input space* to the corresponding outputs in some *output space*.

Typically, algorithms are devised to solve a computational problem. A classical example is the problem of sorting a given list of integers $x_1, \ldots, x_n$ into non-decreasing ($\leqslant$) order—this is an example with which we shall become intimately familiar. Another example is the well-known *Travelling Salesman Problem* (TSP): given a list of cities, together with the distances between them, the goal is to determine the optimal route one should take in order to visit each city precisely once.

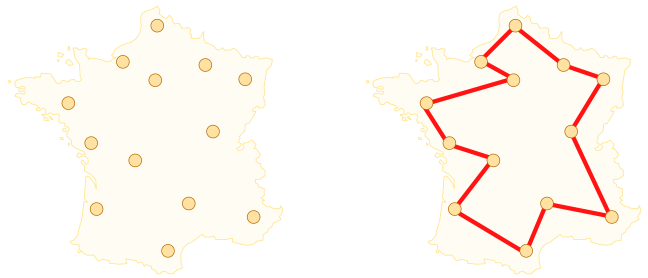


Figure 1: An instance of the TSP in France, and its solution

At this point, you might be thinking: "what is the difference between an algorithm and a computer program?", and this is a legitimate question. A computer program is usually written in a specific language (such as C or Python), whereas an algorithm is something which is language agnostic: it refers to the sequence of steps we perform—the idea behind the code.

In fact, we often write algorithms using *pseudocode*, which is something resembling a programming language, but with no strict syntax.

```
int main(){
  unsigned x=0;
   for(;x<10;)
      x++;
   return 0;
}
```

```
x=0
while x<10:
    x+=1
```

```
$x \leftarrow 0$
**while** $x < 10$ **do**
    $x = x + 1$
```

Figure 2: Two programs, one written in C, the other in Python, which carry out the logic in the pseudocode on the right

Not all Algorithms are Created Equal

The two problems we mentioned earlier are wildly different from each other: the former is quite well-understood, and there are many "good" algorithms which can sort a list of integers. The latter, on the other hand, is a very difficult problem for which no "good" algorithm exists (yet).

Shortly, we will give a more precise definition of what makes an algorithm "good", but the essential idea is that an algorithm is good if it isn't too slow. More precisely, if the "size" of the input gets large, then the time the algorithm takes to solve the problem shouldn't scale too horribly. Now again, since different problems involve different kinds of inputs, the term "size of the input" takes on different meanings, depending on the algorithm in question; in the case of sorting, we usually interpret it to mean how long the list of integers is, whereas in the TSP, we typically use the number of cities.

N (Length of List)	2	10	100	1000	10^6	10^9
Time	0.01 ns	0.01 ns	10 ns	0.0002 s	0.4 s	10 minutes

N (Cities)	2	10	12	20	30	40	50
Time	0.01 ns	3 662 ns	0.02 s	15 s	9.6 h	728 days	3192 years

Table 1: Time taken by Merge Sort and Held–Karp algorithm (TSP)

Table 1 gives some running times for merge sort, a sorting algorithm we will cover, as well as the Held–Karp algorithm for TSP, which is essentially the best algorithm we currently have for the problem.

The reason they differ so drastically is because we can show, using mathematical techniques, that the time taken by merge sort is roughly $c_1 N \log N$, where c_1 is a fixed constant, and N is the length of the list. By contrast, the time taken by the Held–Karp algorithm is around $c_2 N^2 2^N$, where N is the number of cities.

In this course, we will learn how to obtain estimates like these for the runtime in terms of N. This is called finding the *time complexity* of the algorithm.

You should have a good understanding of how different mathematical functions grow as their input grows—there is what is typically referred to as an *asymptotic hierarchy*:

$$\log N \lll N^{0.1} \lll \sqrt{N} \lll N \lll N\log N \lll N\sqrt{N} \lll N^2 \lll N^3 \lll 2^N \lll 3^N \lll N!$$

The notation $\lll$ here will be defined precisely later on, but you can read it as "grows more slowly than". Obviously this is not a complete list of possible functions in this hierarchy: you should, for example, understand where $N^{2/3}$, $N^2 \log N$ and $N^3 + 7\log N$ go if we were to insert them above.

The following exercise should help your understanding of growth of functions.

Exercise 0.1. For each of the functions $f(N)$ in the following table, determine how large N can be so that an algorithm taking $f(N)$ microseconds doesn't exceed the time stipulated in the column.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\log N$							
$\sqrt{N}$							
N							
$N\log N$							
N^2							
N^3							
2^N							
$N!$							

[Assume $\log = \log_2$.]

How to Think About this Course

The best analogy for a course on algorithms is that of a cooking class—it doesn't really matter what recipes you end up preparing, the important things you take away from the course are the techniques and ability to tell a good algorithm (or recipe) from a bad one.

Having said that, just as with cooking, there are some classic algorithms everyone who has taken such a course ought to know. (Just as everyone who has taken a cooking class ought to know how to make scrambled eggs). An algorithm is quite analogous to a recipe: both are procedures defined to transform a set of inputs (eggs, flour, etc.) into an output (a cake) by following a sequence of steps (the recipe).

1. Complexity

When we determine the time complexity of an algorithm, we mainly want to determine the number of "steps" that the algorithm will perform, where a step is usually understood to mean a basic statement (such as a variable declaration, print statement or simple expression evaluation) which should take the computer the same amount of time to execute.

Thus our unit of time in terms of complexity calculations is "steps" rather than something like seconds, since it is reasonable to assume that the actual time taken (in seconds) is proportional to the number of steps; and as we've also mentioned, the important thing isn't really the time taken, but more how the algorithm's complexity scales with respect to the size of the input (i.e., we care about what the running time is proportional to, rather than the explicit proportionality constant).

Example 1.1. Take the following trivial example.

```
1: $x \leftarrow 3$
2: for $i = 1, \dots, N$ do
3:     print($i$)
4:     if $i = x$ then
5:         print("$i$ and $x$ are the same")
```

The time complexity of this algorithm depends on N, so it's going to be some function $T(N)$, of N. Let's count the steps:

- Line 1 simply assigns 3 to x, this costs 1 step.
- Line 2 initialises the variable i, which costs 1 step. Implicitly, the loop construct will increment i each time the block 3–5 is executed, so that adds another $N-1$ steps. In addition to this, the block 3–5 will itself be run N times; but we need to determine how costly that block is (the block isn't a "basic" statement, we need to determine its complexity).
  - Line 3 simply prints the current value of i, this costs 1 step.
  - Line 4 checks whether i and x are the same, essentially evaluating the boolean expression `i==x`, this costs a single step.
  - Line 5 is technically also a block made up of a single print statement, but this is only run in the case that i is the same as x. Since x is just 3, this costs 1 step in the iteration with $i = 3$, and 0 steps in all the other $N-1$ iterations.

Thus the total number of steps for this algorithm is

$$T(N) = 1 + 1 + (N-1) + N(1+1) + 1(1) + (N-1)(0) = 3N + 2.$$

Since this is a linear expression, we call this a *linear-time* algorithm.

Example 1.2 (Matrix Multiplication). Recall that given two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$, the entry in the ijth column of their product $\mathbf{AB}$ is

$$\mathbf{AB}[i,j] = \sum_{k=1}^{N} \mathbf{A}[i,k]\mathbf{B}[k,j],$$

where the notation $\mathbf{X}[i,j]$ denotes the ijth entry of the matrix $\mathbf{X}$.

In pseudocode, the algorithm is the following.

```
   **Input**: two $N \times N$ arrays $\mathbf{A}$ and $\mathbf{B}$
1: $\mathbf{AB} \leftarrow N \times N$ array full of zeros
2: for $i = 1, \ldots, N$ do
3:     for $j = 1, \ldots, N$ do
4:         for $k = 1, \ldots, N$ do
5:             $\mathbf{AB}[i,j] \leftarrow \mathbf{AB}[i,j] + \mathbf{A}[i,k] \times \mathbf{B}[k,j]$
```

To determine the number of steps, the cost of line 1 is debatable: even though it seems like a simple initialisation, in memory, initialising an $n \times n$ array of zeros is equivalent to initialising N^2 variables and assigning them the value zero, so we will consider line 1 to cost N^2 steps.

Since there is no conditional logic on the loops in line 2–4, it is clear that each loop will initialise its counter, and increment it every iteration. Line 5 itself is just a simple assignment, we can consider it to cost 1 step. Thus the total number of steps is

$$\begin{aligned} T(N) &= N^2 + 1 + (N-1) + N(1 + (N-1) + N(1 + (N-1) + 1)) \\ &= N^3 + 3N^2 + N, \end{aligned}$$

so this algorithm has *cubic* time complexity.

Exercise 1.3. Show that the algorithm below performs $\frac{1}{2}(N^2 + 4N + 4)$ steps if N is even, and $\frac{1}{2}(N^2 + 3N + 4)$ if N is odd.

```
1: $x \leftarrow 0$
2: for $i = 1, \ldots, N$ do
3:     if $i$ is even then
4:         for $j = 1, \ldots, N$ do
5:             $x \leftarrow x + j$
6:     else
7:         $x \leftarrow x + i$
8: print($x$)
```

In general, it can be quite tedious to be very precise about the exact number of steps. The main thing we care about is the largest term, since when N is big, this is what dominates the expression.

Asymptotic Notation

Suppose we compute the number of steps of an algorithm to be

$$T(N) = 2N^3 - 7N^2 + N - 17.$$

What we care about in terms of time complexity is the most significant term, since when N is large, that dominates all the other terms (so they are less important). So in this case, the most important term is $2N^3$.

Moreover, the constant 2 isn't that important either, since we mainly care about the rate of growth—we'd describe an algorithm with this runtime as an algorithm with "cubic" (N^3) time complexity.

We borrow the following notation from mathematics:

Definition 1.4 (Big-Oh)**.** Let f and g be two positive-valued functions. If there exists a constant $C > 0$ such that, eventually,[1]

$$f(N) \leqslant C \cdot g(N)$$

for all N, then we say that $f(N)$ is *asymptotically dominated* by $g(N)$, denoted by $f(N) \ll g(N)$ or $f(N) = O(g(N))$; the latter is typically read "f is big-Oh of g".[2]

Example 1.5. Let's mathematically prove that $T(N) = 2N^3 - 7N^2 + N - 17$ is $O(N^3)$. Indeed,

$$\begin{aligned} T(N) &= 2N^3 - 7N^2 + N - 17 \\ &\leqslant 2N^3 \qquad\quad + N^3 && \text{(if } N \geqslant 1\text{)} \\ &= 3N^3, \end{aligned}$$

so we take $C = 3$ in the definition, and $N \geqslant 1$ (for the "eventually" part). □

Examples 1.6. Here are some more examples of the notation.

- $N^2 + 3N + 2 = O(N^2)$
- $5N^2 = O(N^2)$
- $5N^2 = O(N^3)$
- $N^2 = O(N^2 + N + 1)$
- $10 \ll 1$
- $\sqrt{N} \ll N^{0.6}$
- A non-example: $N^3 - 5 = O(N^2)$ is *not* true, nor is $N^2 \ll N \log N$.

[1] i.e., for all N larger than some fixed constant.

[2] The sign "=" is not meant to express "is equal to" in its normal mathematical sense, but rather a more colloquial "is".

Two related notations are big-Ω and big-Θ.

Definitions 1.7 (big-Ω and big-Θ)**.** Let f and g be positive valued functions. Then:

(i) f *asymptotically dominates* g, denoted $f(N) \gg g(N)$ or

$$f(N) = \Omega(g(N))$$

(read "f is big-Omega of g"), if $g(N) = O(f(N))$.

(ii) f and g are *asymptotically equivalent*, written $f(N) \asymp g(N)$ or

$$f(N) = \Theta(g(N))$$

(read "f is big-Theta of g") if both $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$.

Example 1.8. $N^2 + \sqrt{N} = O(N^3)$, $N^2 + \sqrt{N} = \Omega(N)$ and $N^2 + \sqrt{N} = \Theta(N^2)$.

Using this notation, we can say that our matrix multiplication algorithm ran in $\Theta(N^3)$ time, and that the algorithm from example 1.1 has time complexity $O(N)$.

Example 1.9. An algorithm with time complexity $\Theta(N \log N)$ time takes 1 second to finish execution when $N = 1\,000$. How long does it take, approximately, when $N = 1\,000\,000$?

Since the algorithm is $\Theta(N \log N)$, then the running time is approximately $c \cdot N \log N$ for some constant c. Since when $N = 1\,000$, this is 1 second, then $1 = c \cdot 1\,000 \log(1\,000)$, which means that $c = \frac{1}{1\,000 \log(1\,000)}$, so in general an approximation for the running time is

$$\frac{1}{1\,000 \log(1\,000)} N \log N \text{ seconds},$$

and this is increasingly accurate as N gets large.

Thus when $N = 1\,000\,000$, the time taken is approximately

$$\frac{1}{1\,000 \log(1\,000)} \cdot 1\,000\,000 \log(1\,000\,000) = 2\,000 \text{ seconds},$$

or around 33 minutes.

Exercise 1.10.

1. Prove that $\log N$ is $O(N^\epsilon)$ for any $\epsilon > 0$.
   [Hint: $\log N = \frac{1}{\epsilon} \int_1^{n^\epsilon} \frac{dt}{t} < \frac{1}{\epsilon} \int_1^{n^\epsilon} dt$.]
2. Prove that $f(N) = N^2 + N\sqrt{N} + \log N$ is $\Theta(N^2)$.
3. It takes 15 seconds to solve the TSP on 20 cities using the Held–Karp algorithm, which is $\Theta(N^2 2^N)$. Approximately how long will it take to do 50 cities? Express your answer in years.

Induction

Suppose an algorithm makes use of two nested loops, and suppose the inner loop uses the counter of the outer loop. Here is an example:

```
1: $x \leftarrow 0$
2: for $i = 1, \ldots, N$ do
3:     for $j = i, \ldots, N$ do
4:         $x \leftarrow x + 1$
5: print($x$)
```

Notice that on line 3, the for-loop starts from i, which varies with the outer loop. In this case, the outer loop clearly runs N times, but the number of times the inner loop runs depends on the outer loop. If we ignore the implicit assignment and increment steps in the for-loops, counting solely the number of times line 4 ends up being executed, we see that it runs

$$\sum_{i=1}^{N}\sum_{j=i}^{N} 1 = N + (N-1) + (N-2) + \cdots + 2 + 1$$

times. This is the well known *Nth triangular number*, and has a nice closed form, namely,

$$\frac{N(N+1)}{2},$$

i.e., $\frac{1}{2}N^2 + \frac{1}{2}N$, which means the complexity is still $\Theta(N^2)$. In other words, even though it seems like we are executing line 4 fewer times, in terms of complexity, it's still just as bad as when j starts from 1 each time (since this is obviously $\Theta(N^2)$).

Why is this formula for $1 + \cdots + N$ true? We can prove it (and many other important formulae) using the technique of induction.

Probably, all mathematical proofs you have encountered so far have been deductive, where one step is deduced from the other (Step 1 $\Rightarrow$ Step 2 $\Rightarrow \cdots \Rightarrow$ Step n). A relatively newer concept (for which we should thank Pascal) is mathematical proof by induction. Induction is a powerful tool we use to prove statements for all natural numbers $(1, 2, 3, \ldots)$, or for any discrete structure in general.

The idea is based on what we often call the *domino effect*. Consider an (infinite) line of dominoes. We wish to prove that all the dominoes will fall, and we do this as follows:

(1) Prove that the first one will fall.

(2) Prove that if the $(N-1)$st domino falls, then the Nth domino falls also.

Suppose we manage to prove both (1) and (2). By (1), we know that the first domino falls. But then by (2), the second one falls also. But now we can use (2) again to conclude that the third domino falls. And the fourth. And the fifth. This can continue indefinitely, thus proving that all the dominoes fall.

Let us transfer this to a mathematical context. Suppose we wish to prove that a statement S_N is true for every natural number N. We can do this by:

(1) Proving that S_N is true for the case $N = 1$, i.e., proving that S_1 is true. This is called the **base case**.

(2) Assume that S_{N-1} is true (we call this the **inductive hypothesis**). Based on this assumption, prove that S_N is true. This is called the **inductive step**.

(2) allows us to transform the proof of the case when $N = 1$ into a proof for when $N = 2$. We can then use (2) again to establish that the result is true when $N = 3$, and so on. I like to think of (2) as a "machine" which transforms a proof of the statement for a specific case, into a proof of the statement for the next case.

Example 1.11. Let's prove the formula $1 + \cdots + N = \frac{1}{2}N(N+1)$.

Base case: For $N = 1$, we have $1 = \frac{1}{2}(1+1)$, so the statement holds.

Now, assume the result holds for $N - 1$, i.e., that

$$1 + 2 + \cdots + (N-1) = \frac{N-1}{2}((N-1)+1) = \tfrac{1}{2}N(N-1).$$

Can we use this to show that it is true for N?

$$\begin{aligned} 1 + 2 + \cdots + (N-1) + N &= \frac{N}{2}(N-1) + N \quad \text{(by induction hypothesis)} \\ &= \frac{N}{2}(N-1+2) \\ &= \frac{N}{2}(N+1) \end{aligned}$$

i.e., the statement holds for N. This completes the proof. □

Example 1.12. We show that the number $4^N + 2$ is always divisible by 3 for every N.

For the base case, when $N = 1$ we have $4^1 + 2 = 6 = 3(2)$, so the statement holds.

Next for the inductive step, we have

$$\begin{aligned} 4^N + 2 &= 4(4^{N-1}) + 2 \\ &= 4(4^{N-1}) + 4 - 2 \\ &= 4(4^{N-1} + 1) - 2 \\ &= 4(4^{N-1} + 2 - 1) - 2 \\ &= 4(3a - 1) - 2 \quad \text{(by induction hypothesis)} \end{aligned}$$

$$\begin{aligned}
&= 12a - 4 - 2 \\
&= 12a - 6 \\
&= 3(4a - 2) \\
&= 3b \qquad \text{where } b = (4a-2)\text{, which is an integer.}
\end{aligned}$$ □

Example 1.13. How about an example with matrices. We prove that if

$$\mathbf{A} = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix},$$

then the matrix power $\mathbf{A}^N$ is given by

$$\mathbf{A}^N = \begin{pmatrix} 3^N & 3^N - 2^N \\ 0 & 2^N \end{pmatrix}.$$

For the base case, we have

$$\mathbf{A}^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 3^0 & 3^0 - 2^0 \\ 0 & 2^0 \end{pmatrix},$$

so the statement holds when $N = 0$. Now

$$\begin{aligned}
\mathbf{A}^N &= \mathbf{A}^{N-1}\mathbf{A} \\
&= \begin{pmatrix} 3^{N-1} & 3^{N-1} - 2^{N-1} \\ 0 & 2^{N-1} \end{pmatrix}\begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix} \qquad \text{(by the IH)} \\
&= \begin{pmatrix} 3^{N-1}(3) + 0 & 3^{N-1} + 2(3^{N-1} - 2^{N-1}) \\ 0 & 2^{N-1}(2) \end{pmatrix} \\
&= \begin{pmatrix} 3^N & 3^{N-1} + 2(3^{N-1}) - 2^N \\ 0 & 2^N \end{pmatrix} \\
&= \begin{pmatrix} 3^N & 3^{N-1}(1+2) - 2^N \\ 0 & 2^N \end{pmatrix} = \begin{pmatrix} 3^N & 3^N - 2^N \\ 0 & 2^N \end{pmatrix},
\end{aligned}$$

as required. □

Remark 1.14. The general strategy of proof by induction is to try and "break down" the objects we are working with in the N case into the $N-1$ case, plus something else which elevates it to the N case (e.g., the matrix $\mathbf{A}^N$ into $\mathbf{A}^{N-1}\mathbf{A}^1$). This allows one to apply the inductive hypothesis.

Thus, one deduces that induction would not be very useful when it is very difficult, or impossible, to relate the case of $N-1$ to that of N. When results are about the natural numbers themselves, often one can use induction, because we have an explicit way to relate $N-1$ and N (namely $N = (N-1)+1$).

Exercise 1.15.
1. Prove that the sum of the first N odd numbers is N^2.
2. Show that $(1+2+\cdots+N)^2 = 1^3+2^3+\cdots+N^3$.
3. ♨ Prove that 5^N can always be expressed as a sum of two squares.

Recursion

A very important result, which goes hand-in-hand with induction, is the idea of definition by recursion. This idea is essential in mathematics and computer science, the idea of a function "using itself" in a definition.

Let us motivate this with an example, before we state the theorem. The factorial of a natural number N, written $N!$, is the product of all numbers from 1 to N, i.e.,

$$N! = N\cdot(N-1)\cdots 2\cdot 1.$$

An alternative way to define this is the following: suppose we want to find 5!, *but we already know what* 4! *is*. How can we use this information? Well if we already know what 4! is, we can just multiply this by 5 and we get 5!—in general, we have that $N! = N\cdot(N-1)!$. We could use this to define factorial differently—let $\mathrm{fac}(N)$ denote our "newly" defined factorial.

If we define $\mathrm{fac}(N) = N\cdot\mathrm{fac}(N-1)$, this almost works. Indeed, if we try to evaluate 5! this way, we do

$$\begin{aligned}\mathrm{fac}(5) &= 5\cdot\mathrm{fac}(4)\\ &= 5\cdot 4\cdot\mathrm{fac}(3)\\ &= 5\cdot 4\cdot 3\cdot\mathrm{fac}(2)\\ &= 5\cdot 4\cdot 3\cdot 2\cdot\mathrm{fac}(1)\\ &= 5\cdot 4\cdot 3\cdot 2\cdot 1\cdot\mathrm{fac}(0) \qquad (*)\\ &= 5\cdot 4\cdot 3\cdot 2\cdot 1\cdot 0\cdot\mathrm{fac}(-1)\\ &\vdots\end{aligned}$$

Something seems to have gone wrong here. The "unrolling" of the fac's seems to have worked, because we did get the product $5\cdots 1$ appearing, but this will keep on going forever. We need some mechanism which stops this infinite expansion. If we instead say $\mathrm{fac}(0) = 1$, then we would actually stop at the line $(*)$. This is called the *base case*. Familiar?

So we define

$$\mathrm{fac}(N) = \begin{cases} 1 & \text{if } N = 0\\ N\cdot\mathrm{fac}(N-1) & \text{otherwise.}\end{cases}$$

And this works! Apart from being nicely succinct (without having to use informal notions such as $\cdots$), this gives us a way to write an elegant algorithm for computing $N!$:

```
function fac($N$)
    if $N = 0$ then
        return 1
    else
        return $N * \text{fac}(N-1)$
```

But is this always allowed? Can we define a random function, say,

$$f(N) = \begin{cases} 1 & \text{if } N = 0 \\ N\,f(N+1) & \text{otherwise?} \end{cases}$$

Try to work out $f(5)$. It's not hard to see why this definition is problematic. Thus, we need to see precisely *when* we are allowed to do this. Indeed, notice that fac(N) made use of N, and of *the value of* fac($N-1$). In other words, in defining $f(N)$, if we only allow $f(N-1)$ to appear (and possibly N), then the definition should be fine.

This is what the following theorem guarantees.

Theorem 1.16 (Definition by Recursion)**.** *Suppose X is a set, let $x \in X$ and let $g\colon \mathbb{N}\times X \to X$ be a total function. Then there exists a unique total function $f : \mathbb{N} \to X$ such that*

$$f(N) = \begin{cases} x & \textit{if } N = 0 \\ g(N, f(N-1)) & \textit{otherwise.} \end{cases}$$

All this theorem is telling us is that we are allowed to have $f(N-1)$ appearing in the definition of f (and only that), and this will determine a unique, valid function from $\mathbb{N}$ to X. We will not prove it here, since it is a bit technical (although not difficult). You probably guessed it, but the proof is by induction!

For $N!$, the function $g\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ we need to take $g(N,t) = Nt$.

This theorem can be generalised so that we have any number of base cases.

Theorem 1.17 (General Definition by Recursion)**.** *Let X be a set, let $x_0, \ldots, x_r \in X$, and let $g\colon X^{r+1}\times\mathbb{N} \to X$ be a function. Then there exists a unique function $f : \mathbb{N} \to X$ such that*

$$f(N) = \begin{cases} x_N & \textit{if } 0 \leqslant N \leqslant r, \\ g(f(N-r), f(N-r+1), \ldots, f(N-2), f(N-1), N) & \textit{otherwise.} \end{cases}$$

This looks a bit complicated, so let us give an example to clarify. The Fibonacci sequence, named after Leonardo Bonacci of Pisa (1170–1270 A.D.), is defined as follows.

$$\operatorname{fib}(N) = \begin{cases} 0 & \text{if } N = 0 \\ 1 & \text{if } N = 1 \\ \operatorname{fib}(N-1) + \operatorname{fib}(N-2) & \text{otherwise.} \end{cases}$$

Here we have $r = 1$ in theorem 1.17 with $X = \mathbb{N}$, so our definition must match the form

$$f(N) = \begin{cases} x_0 & \text{if } N = 0 \\ x_1 & \text{if } N = 1 \\ g(f(N-1), f(N-2), N) & \text{otherwise} \end{cases}$$

if we are allowed to define it. The base cases are clearly compatible. But what about the last case? Is it a function of the form $g(f(N-2), f(N-1), N)$? Yes! It simply does not make use of the variable N, but it makes use of "what it's allowed to", namely the values of fib() for the previous two natural numbers.

Remark 1.18. If you are interested in the formal details of induction and recursion (and they are very interesting), it turns out you can do induction and recursion on any set which can be ordered (i.e., a ranked poset), not just the natural numbers. Such structures include programming languages: you can prove things about programming languages using induction (e.g. type safety, normalisation, etc.).

But this falls out of the scope of this course. However, if you are interested, for the mathematical side, I suggest looking at the Recursion Theorem in chapter 3 of Hrbáček and Jech's *Introduction to Set Theory*. For the computer science side, I suggest Pierce's *Types and Programming Languages*.

Exercise 1.19. 1. A *derangement* of the numbers $1, 2, \ldots, N$ is an arrangement of the numbers such that no number i is in the ith position. For example, 4321 and 21534 are derangements, but 54321 is not, because 3 appears in the 3rd position.

The number of derangements of the numbers $1, \ldots, N$ is denoted by $!N$.

(a) Explain why

$$!N = \begin{cases} 1 & \text{if } N = 0 \\ 0 & \text{if } N = 1 \\ (N-1)[!(N-1) + !(N-2)] & \text{otherwise,} \end{cases}$$

using combinatorial reasoning.

(b) Prove, using induction, that $!N = N!(\frac{1}{2!} - \frac{1}{3!} + \cdots + (-1)^N \frac{1}{N!})$.

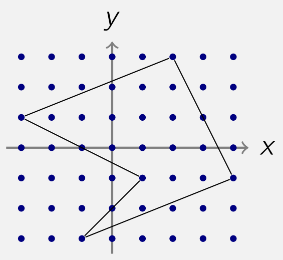
(c) Deduce that $!N \approx N!/e$.

(d) 15 people in an office organise a "Secret Santa" for Christmas, where they each write their names on a piece of paper, place them in a hat, and subsequently each person picks one from the hat.

If someone gets their own name, they have to do the whole process again. What is the probability that they have to do the process N times?

2. (Pick's Theorem) Consider the points in the plane with integral coordinates, $\mathbb{Z}^2$. A *simple polygon* is a polygon whose edges do not intersect each other.

Show that the area of a simple polygon with vertices at integral coordinates is $\frac{1}{2}(2i + b - 2)$, where i is the number of points contained entirely within the polygon, and b is the number of points that lie on the boundary of the polygon (i.e. on some edge.)



In the example above, there are $i = 15$ points inside the polygon, and $b = 8$ on the boundary. Thus the area, by Pick's theorem, is $A = \frac{1}{2}(30 + 8 - 2) = 18$ square units.

3. (Catalan Numbers) A sequence of open and closed brackets is said to be *balanced* if each open bracket can be matched with a closed bracket. For example,

`((()))` `()()()` `()(()())(())`

are balanced, whereas `)(`, `())(` and `(()))(()` are not. Given N pairs of open/closed brackets, the *Catalan number* C_N is the number of balanced arrangements of the $2N$ brackets. For example, $C_3 = 5$, since the only possibilities are `()()()`, `(())()`, `()(())`, `(()())`. and `((()))`.

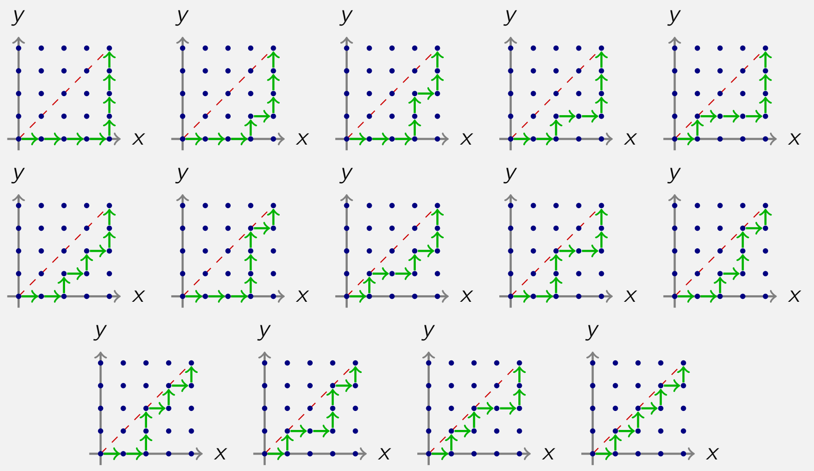
(a) Show that in general, $C_{N+1} = C_N C_0 + C_{N-1} C_1 + \cdots + C_0 C_N$, where $C_0 = 1$ by convention.

(b) Hence show that $C_N = \frac{1}{N+1}\binom{2N}{N}$.

(c) Show that the coefficient of x^N in the Maclaurin series expansion of $f(x) = 2/(1+\sqrt{1-4x})$ is C_N.

(d) Show that C_N is the number of *monotonic paths* from $(0,0)$ to (N,N) in $\mathbb{N}^2$ which do not go above the diagonal. A monotonic path is one which goes only rightwards or upwards.

Here is an illustration for the case $N = 4$:



All possible monotonic paths from $(0,0)$ to $(4,4)$, so $C_4 = 14$.

Now, this chapter is supposed to be about complexity, and we seem to have gotten a little side tracked. Let's determine the time complexity of the following algorithm which computes the Nth Fibonacci number, F_N, for $N \geqslant 1$.

```
function fib($N$)
    if $N < 3$ then
        return 1
    else
        return fib($N-1$) + fib($N-2$)
```

Each function call to fib() (excluding subsequent calls) does $\Theta(1)$ work, namely, a comparison and an addition, so a good estimate for the number of steps is to count the number of times fib() ends up being called.

By induction, we can show that this number is $2F_N - 1$.

So the algorithm has time complexity $(2F_N - 1)\cdot\Theta(1) = \Theta(F_N)$. But is this good, or bad? Can we get a mathematical expression for F_N to understand where it fits into the asymptotic hierarchy of functions? The answer is yes, since we can find a formula for functions which satisfy a relation of the kind $f(N) = A\,f(N-1) + B\,f(N-2) + C$.

Theorem 1.20 (Stable Second Order Linear Recurrence Relations)**.** *Suppose that for all relevant N, a function f satisfies the relation*

$$f(N) = A\,f(N-1) + B\,f(N-2) + C,$$

where A, B and C are real numbers. If $A + B \neq 1$, then for all N, f is defined by a formula of the form

$$f(N) = \frac{C}{1-A-B} + c_1\,\alpha^N + c_2\,\beta^N,$$

where c_1 and c_2 are constants, and α and β are the roots of $x^2 = Ax + B$.

We can prove this by induction, but in the interest of brevity we will omit the proof.

For the Fibonacci numbers, we have $A = B = 1$ and $C = 0$, so an explicit formula for them has the form

$$F_N = c_1\,\alpha^N + c_2\,\beta^N$$

where α and β are the roots of $x^2 = x + 1$. The roots are easily found to be $(1 \pm \sqrt{5})/2$, and these values are usually denoted by φ and $1/\varphi$,[3] since it turns out that $(1+\sqrt{5})/2 = 1/((1-\sqrt{5})/2)$. So we have that

$$F_N = c_1\varphi^N + \frac{c_2}{\varphi^N}$$

where $\varphi = (1+\sqrt{5})/2$, and c_1, c_2 are constants. We don't really need to find these constants, since it's already clear that $F_N = \Theta(\varphi^N)$ from the above, but we can determine them since we know that $F_0 = 0$, which means that

$$0 = c_1 + c_2 \implies c_2 = -c_1,$$

so the formula is

$$F_N = c_1\varphi^N - \frac{c_1}{\varphi^N}.$$

Now to determine c_1, we use the fact that $F_1 = 1$, so

$$1 = c_1\left(\frac{1+\sqrt{5}}{2}\right) - \frac{c_1}{\frac{1+\sqrt{5}}{2}},$$

which we can solve to obtain that $c_1 = 1/\sqrt{5}$, which means that

$$F_N = \frac{\varphi^N - (1/\varphi)^N}{\sqrt{5}},$$

which is a nice explicit formula for F_N.

[3] φ is called the *Golden ratio*, this number has an ancient history.

	Constant time	$\Theta(1)$
	Fractional Power	$\Theta(N^c)$, $c \in (0, 1)$
	Linear	$\Theta(N)$
"Good"	Log-linear	$\Theta(N \log N)$
	Quadratic	$\Theta(N^2)$
	$\vdots$	$\vdots$
	Polynomial	$\Theta(N^k)$
	$\vdots$	$\vdots$
	$\vdots$	$\vdots$
"Bad"	Exponential	$\Theta(c^N)$, $c > 1$
	Factorial	$\Theta(N!)$
	$\vdots$	$\vdots$

Table 2: Different time complexities: the good, the bad & the ugly

So we managed to find a nice mathematical formula for the Nth Fibonacci number, but the bad news is that it is an exponential function. In particular, since $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$, then

$$\left(\tfrac{3}{2}\right)^N \lll \varphi^N \lll 2^N,$$

so it's a bad algorithm.

In general, the cut-off point for what we consider a "bad" algorithm, in terms of its time complexity, is anything with complexity larger than polynomial (see table 2). As we've pointed out before, it's impossible to give an exhaustive table of time complexities, but you should know how to compare any two functions. For instance, which is better, $N \log(\log N)$ or $\sqrt{N}\, 2^{\log N}$?

Before we conclude this chapter, let's remark on how we can improve the algorithm for computing Fibonacci numbers: utilising a technique known as *memoisation*.

Notice that when computing, say, F_{50} using this algorithm, many redundant calls are being made. For instance, we will wastefully compute F_{48} twice, and in turn this will produce two evaluation trees which are identical and needlessly repeat computations. Similarly we end up working out F_{47} three times, F_{46} four times, and so on. Refer to figure 3.

What we can do is *cache* the value once it has been computed, so we never work out a Fibonacci number if it's already been worked out.

Here is the updated algorithm which does the caching:

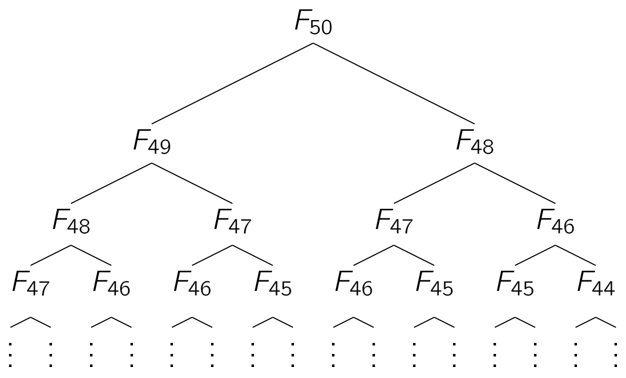


Figure 3: Recursive function calls for the $\Theta(\varphi^N)$ Fibonacci algorithm

```
fibs_done ← array full of $N$ zeros
function fib($N$)
    if $N < 3$ then
        return 1
    else if fibs_done[$N$] $\neq 0$ then
        return fibs_done[$N$]
    else
        ans ← fib($N - 1$) + fib($N - 2$)
        fibs_done[$N$] ← ans
        return ans
```

This time, the time complexity is drastically improved to $\Theta(N)$, since we only make a recursive function call once per Fibonacci call! Look at the two recursive call trees in figures 3 and 4 to get a better understanding.

So as a takeaway from this example, when designing an algorithm, always consider how you can speed things up by saving previously computed terms. This ties into a broader principle of algorithm design, known as *dynamic programming*.

We've focused mainly on time complexity so far, but there is another kind of complexity we haven't mentioned: *space complexity*, i.e., how much space we need to store data while the algorithm is running. It turns out that both implementations require $O(N)$ space, the former due to the size of the recursion stack, the latter more clearly requires $O(N)$ space since we need to store N values.

There is a way to implement the Fibonacci algorithm which avoids an $O(N)$ space complexity. The idea is to keep only the previous two terms, and work iteratively, rather than recursively.

This method is quite natural to formulate, but had we started with it, we would have deprived ourselves of the exploration of induction and recursion, which wouldn't have

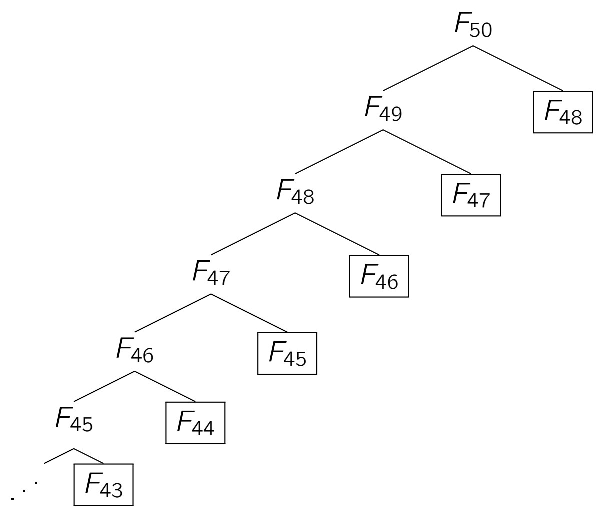


Figure 4: Recursive function calls for the memoised Fibonacci algorithm. The boxed function calls are $\Theta(1)$ since we just need to fetch them from the cache.

been very fun. Here it is:

```
function fib($N$)
    previous ← 0
    current ← 1
    for $N - 1$ times do
        new ← previous + current
        previous ← current
        current ← new
    return current
```

Clearly (assuming that simple things like addition and assignment are $\Theta(1)$), this takes $\Theta(N)$ time and $\Theta(1)$ space. We will revisit the ideas of dynamic programming at a later stage in the course, when we solve the classical problem of finding a longest common subsequence between two strings.

Exercise 1.21. The binomial coefficient $\binom{n}{k}$ (a.k.a., Pacal's triangle) can be defined recursively by

$$\text{binom}(n, k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k \\ \text{binom}(n-1, k-1) + \text{binom}(n-1, k) & \text{otherwise.} \end{cases}$$

Prove by induction that the recursive algorithm inferred by this definition calls the function binom() $2\binom{n}{k} - 1$ times. Write a better algorithm using memoisation, and an iterative one which uses the fact $\binom{n}{k} = n(n-1)\cdots(n-k+1)/k!$.