At the beginning of its computation, $\mathcal{U}$ will contain only the input $\langle \mathcal{M}, w\rangle$ on its first tape, and its other two tapes will be blank.

$\cdots$	␣	$\langle$	$\cdots$	$\mathcal{M}$	$\cdots$	,	$\cdots$	w	$\cdots$	$\rangle$	␣	$\cdots$
$\cdots$	␣	␣	$\cdots$	␣	$\cdots$	␣	$\cdots$	␣	$\cdots$	␣	␣	$\cdots$
$\cdots$	␣	␣	$\cdots$	␣	$\cdots$	␣	$\cdots$	␣	$\cdots$	␣	␣	$\cdots$

Now, we can describe how $\mathcal{U}$ simulates the computation of $\mathcal{M}$ on w:

1. Initialize the tapes in the following way:
   (a) Transfer the encoding of $\mathcal{M}$ from the first tape to the second tape by writing $\langle \mathcal{M}\rangle$ to the second tape and erasing it from the first tape.
   (b) Read the encoding of $\mathcal{M}$ on the second tape to determine the number of states in $\mathcal{M}$. If $\mathcal{M}$ contains k states, then write $\lceil \log_2(k)\rceil$ copies of `0` to the third tape.[5]

   After initialization, the tape will look like the following:

$\cdots$	␣	␣	$\cdots$	␣	$\cdots$	$\langle$	$\cdots$	w	$\cdots$	$\rangle$	␣	$\cdots$
$\cdots$	␣	$\langle$	$\cdots$	$\mathcal{M}$	$\cdots$	$\rangle$	$\cdots$	␣	$\cdots$	␣	␣	$\cdots$
$\cdots$	␣	0	$\cdots$	0	$\cdots$	0	$\cdots$	␣	$\cdots$	␣	␣	$\cdots$

2. Move the input head of the first tape to the first symbol of w. Move the input head of the second tape to the first symbol of $\langle \mathcal{M}\rangle$. Move the input head of the third tape to the first symbol of the sequence of `0`s.
3. Repeat the following steps until $\mathcal{M}$ halts:
   (a) For each computation step of $\mathcal{M}$, scan the second tape to find a transition that matches the current input symbol and state written on the first and third tapes, respectively.
   (b) Modify the contents of the first and third tapes to reflect the transition that was just taken.
   (c) If no transition exists for the current state/symbol pair, then halt and go to step 4.
4. Transition to q_{accept} if $\mathcal{M}$ transitioned to its accepting state. Transition to q_{reject} if $\mathcal{M}$ transitioned to its rejecting state. $\square$

Let's now verify that this construction does, in fact, satisfy the three criteria we laid out earlier. First, we require that $\mathcal{U}$ halts its computation on input $\langle \mathcal{M}, w\rangle$ if and only if $\mathcal{M}$ halts its computation on input w. Since $\mathcal{U}$ uses the description of $\mathcal{M}$ to see what it would do after reading each symbol of w, $\mathcal{U}$ behaves in a manner identical to $\mathcal{M}$, and so $\mathcal{U}$ halts on its input if and only if $\mathcal{M}$ also halts on its input. Second, we require that $\mathcal{U}$ accepts if and only if $\mathcal{M}$ accepts, and we see immediately that this happens by Step 4 of the simulation procedure. In fact, $\mathcal{U}$ can't reach its accepting state unless $\mathcal{M}$ has done the same. Third, we require the same behaviour for rejecting inputs, and the argument in this case is nearly identical to the one we had for accepting inputs. Therefore, the universal Turing machine $\mathcal{U}$ satisfies all of our criteria and performs a correct simulation of any other Turing machine we provide as input!

3 The Church–Turing Thesis

In the early days of computer science, long before physical computers as we know them existed, mathematicians and logicians wanted to know whether it was possible to use mechanical procedures to solve mathematical problems. Perhaps the most well-known of these problems at the time was the *Entscheidungsproblem*, which is German for "decision problem". The Entscheidungsproblem asks whether there

[5]Remember, the current state is being maintained as a binary number, so we need a logarithmic amount of bits to represent a given state's number k.

exists a general procedure to decide whether a given statement is valid and provable using a predetermined system of logic. The Entscheidungsproblem was formalized by the German mathematician David Hilbert in 1928, although the seeds of the idea go all the way back to 1900, when Hilbert presented his famous set of problems at the International Congress of Mathematicians.

The problem back then was that there was no universally-agreed upon definition of a "procedure" that could decide the problem. The most appropriate definition was eventually taken to be that of an *effective method.* If we're given a class of problems, then a method for that class of problems is called effective if

1. the method consists of a finite number of exact instructions; and
2. the method always terminates and produces a correct answer when it is applied to a problem from its class.

In principle, an effective method is one that a human can perform on paper in a purely mechanical manner; it requires no creative thought or insight to arrive at an answer. It is computation in its purest form. If we view the class of problems in a way that allows us to sort instances of the problem into "yes" and "no" outcomes, then we essentially have a function that takes an input and returns either "yes" or "no". Then, any function with an associated effective method to solve it is called an *effectively calculable* function.

Throughout the 1930s, mathematicians and logicians focused on developing effective methods for solving the Entscheidungsproblem. In 1931, the Austrian-born logician Kurt Gödel published his famous paper introducing his incompleteness theorems. In this paper, Gödel introduced the notion of *recursive functions*, which was his approach to defining effective calculability.

The first major breakthrough with the Entscheidungsproblem came in a series of papers by the American mathematician Alonzo Church, wherein he introduced the notion of the *lambda calculus.* The lambda calculus is a system of mathematical logic that allows us to express computations in terms of function applications. Church proposed that the class of effectively calculable functions should correspond to the class of functions that can be defined in the lambda calculus. Indeed, along with Stephen Kleene and J. Barkley Rosser, Church showed that the class of lambda-definable functions corresponded exactly to the class of recursive functions. In 1936, Church then proved that the Entscheidungsproblem was unsolvable.

The next breakthrough came, again, in 1936 with a presentation by Alan Turing to the London Mathematical Society. Like Church, Turing showed that the Entscheidungsproblem was unsolvable, but he used a different formalization: a machine model, which later came to be known as our familiar *Turing machine.* Turing was aware of Church's work, and he added as an appendix to his paper a proof sketch showing that his machine formalization was equivalent to Church's lambda calculus. Turing would go on to complete his PhD under the supervision of Church just a couple of years later.

Despite this flurry of results throughout the 1930s, it would take until 1952 for someone to formally define the notion of effective calculability. Stephen Kleene, in his book *Introduction to Metamathematics*, introduces what he calls *Church's thesis*:

> "Every effectively calculable function (effectively decidable predicate) is general recursive."

In other terms, this thesis states that any problem for which there exists a procedure that returns an answer on each input belonging to the problem's language is semidecidable.

Kleene later introduces in the same book what he calls *Turing's thesis*:

> "[. . .] that every function which would naturally be regarded as computable under [Turing's] definition, i.e. by one of his machines, is equivalent to Church's thesis [. . .]"

This thesis encapsulates the spirit of the appendix Turing provided in his paper, where he showed that his machine formalization was equivalent to the lambda calculus formalization given by Church: anything that a Turing machine can do is effectively calculable, and therefore semidecidable.

Taken together, these two statements give us the *Church–Turing thesis*: a connection between effectively calculable procedures and Turing machines. Note that, since this result is more definitional rather than a statement that we can formally prove, we refer to it as a "thesis" instead of a "theorem".

In modern language, we can state the thesis as follows.

Church–Turing thesis. *Any function that can be computed by an algorithm can also be computed on a Turing machine.*

In recent times, the Church–Turing thesis has allowed researchers to prove that all sorts of formal models are capable of behaving like a machine running an algorithm. If some model of computation or some system of rules can be used in a way that allows it to simulate the computation of any Turing machine, then we say that model or system is *Turing-complete*.

We've already seen one example of something that is Turing-complete—the universal Turing machine. But since that is itself a kind of Turing machine, we shouldn't be too surprised. Instead, there are many more (and much weirder) examples of Turing-complete things in our daily lives:

- Most general-purpose programming languages, and some specialized languages (like LaTeX, the type-setting system used to create these lecture notes!)
- Microsoft Excel[6] and Microsoft PowerPoint[7]
- Conway's Game of Life[8] and other cellular automata[9]
- Enzyme-based DNA computers[10]
- The cells of the human heart[11]
- The Dwarf Fortress[12], Minecraft[13], and Minesweeper[14] video games
- The Magic: The Gathering card game[15]
- The x86 assembler instruction `mov`, by itself[16]

You might now reasonably ask whether the computers we use every day are Turing-complete. The answer, interestingly, is no! This comes down to one simple reason: nobody has yet figured out how to equip a real-world computer with an infinite amount of memory. Thus, when we speak about something being Turing-complete, we often set aside the limitation of finite memory and focus on the computational power of the thing itself.

[6] A. Gordon. LAMBDA: The ultimate Excel worksheet function, 2021. Accessible at `https://www.microsoft.com/en-us/research/blog/lambda-the-ultimatae-excel-worksheet-function/`.

[7] T. Wildenhain. On the Turing completeness of MS PowerPoint. In Proc. of SIGBOVIK 2017, pages 102–106. 2017.

[8] E. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays*, volume 4. A. K. Peters Ltd., Wellesley, second edition, 2004.

[9] M. Cook. Universality in elementary cellular automata. *Complex Systems*, 15:1–40, 2004.

[10] E. Shapiro. A mechanical Turing machine: Blueprint for a biomolecular computer. *Interface Focus*, 2:497–503, 2012.

[11] S. Scarle. Implications of the Turing completeness of reaction-diffusion models, informed by GPGPU simulations on an XBox 360: Cardiac arrhythmias, re-entry and the Halting problem. *Computational Biology and Chemistry*, 33(4):253–260, 2009.

[12] Jong89. DF Map Archive: Razorlength - 1036 Early Winter, 2009. Accessible at `https://mkv25.net/dfma/map-8269`.

[13] Bytejacker. Minecraft - Notch Interview!, 2010. Accessible at `https://www.youtube.com/watch?v=rqUDam_KJno`.

[14] R. Kaye. Infinite versions of Minesweeper are Turing complete. University of Birmingham School of Mathematics preprint number 2000/15, 2000. Accessible at `https://web.mat.bham.ac.uk/R.W.Kaye/minesw/infmsw.pdf`.

[15] A. Churchill, S, Biderman, and A. Herrick. Magic: The Gathering is Turing complete. In Proc. of FUN 2021, pages 9:1–9:19. 2021.

[16] S. Dolan. `mov` is Turing-complete, 2013. Accessible at `https://drwho.virtadpt.net/files/mov.pdf`.