CHAPTER 3

Turing Machines

> "The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer."
>
> Alan M. Turing (1912-1954)

Imagine that you could reduce all known computations, anything you know a computer can do and even arithmetic, down to just a few fundamental operations of reading, writing and moving symbols around on pieces of paper and nothing more. It would be simple enough to even solve any computation by hand given enough patience. With such a simple machine, we could not only study computation and prove its properties, but also help design a physical realisation of such a computation machine. Alan Turing [Turing, 1937] introduced just such a machine in order to study the "Entscheidungsproblem" (or the decision problem) and in the process developed a theory of computation. These machines are known as Turing machines in his honour and eventually led to the design the first digital computers by John von Neumann and others.

3.1 Computers

But what are computers? We associate computers with digital devices and electronic hardware because this is the current generation of computing we use today. However, the electronic computer is a fairly recent invention with the Colossus constructed in 1944 at Manchester, England and the ENIAC constructed in 1945 in the United States. Before then, a computer was a specialised human being that computed results for scientists and engineers by hand! In fact, the computations for the Manhattan project that eventually devised the Hydrogen bomb used to end World War II utilised women computers extensively. The design of the workflow for these human computers were sophisticated enough to detect errors and one of the major problems Nobel laureate Richard Feynman had to solve for this project to ensure calculations could be done in time [Feynman, 2005]. We will use this idea of a human computer to establish if and how we

can transition to an automatic computing engine (ACE) as Turing envisioned it[1] or a computer as we know it today.

Imagine that you have a hypothetical "computer" that is defined by everything inside a closed off room. This room has two small letterboxes, one for the input and another for the output. From the outside, it is not possible to see the inner workings of the room and the only way to interact with this "computer" is via letters through these letter receptacles.

Let our room initially house a human and that person computes whatever is requested via the input letters. Immediately you would recognise that the "language" used to communicate with the "computer" is very important and has to be predefined in order for the device to understand the input and for us to understand its output. More on this definitions later when we construct our ACEs. Without loss of generality, we may assume that the device is only used for arithmetic calculations, so that a simple computation 2+3 provided to the "computer" would be easily calculated by the person in the room and return the result 5. The person inside the room would use his or her knowledge of arithmetic to compute the result.

Now let us consider providing an instruction booklet to the human that describes the method for all types of supported arithmetic step-by-step, such as addition in our case above. The human is told to only compute the result following this booklet. The human would examine the input and based on its contents, select the appropriate instructions, modify the symbols until a solution is obtained. From this scenario, a two practical points arise:

1. The language and the representation of the inputs now need to match the instructions more formally, since the they must operate on valid known inputs to function properly.

2. It is clear that the human would require working paper and pencil/pen to keep an accounting of which part of the input is currently being processed, where in the booklet one is currently located and the value(s) of the intermediate result.

In other words, our "computer" requires a formal language and working paper or some memory in addition to the instruction booklet.

If we examine our 2+3 example using this booklet approach, we can define a unary representation for numbers and instructions depending on the arithmetic operators supported and cells to write our representation. A unary representation is used in cartoons and movies, where the protagonist commonly counts the number days in captivity via a series if ones. In our example, the input would look like We have chosen

1	1	+	1	1	1

to use the '+' symbol, but it could have been any symbol as long as it corresponds to the instructions. To obtain our sum, observe that we simply need to move all the three 1's after the + symbol one cell to the left, overwriting the '+' symbol and return the result as this will be the unary representation of the answer 5.

1	1	1	1	1	_

Thus, the instructions to a human would be along the lines of the following for addition:

[1] We use the term ACE to refer to the concept of an automated computer and loosely on the ACE computer specifications designed initially by Turing.

1. Start at the first element.
2. Travel right, denoted by the symbol R until the operator symbol is located in the input.
3. If the operation is addition, move all the 1's after the + symbol one cell to the left overwriting the + symbol.
4. Return the result.

Note that the act of moving symbols across the cells will involve the human using the working paper to keep track of the intermediate results. The act of executing the instructions above, the human will need to keep track of the state of the computation so far, i.e. moving cells, returning the result etc. Finally note that the human will need to look at and place their attention or awareness on one cell at a time to do the moving of cells.

It is important to point out that the instructions we are talking about is what we call a computer program (or instruction sets) in modern terminology. This program represents an algorithm or a collection of algorithms that can solve the problem for which we seek a solution. Also, not only is the input is a series of symbols, but the instructions above can also be represented as symbols resulting in the computation being reduced to nothing but "codes" and a manipulation of this code. Therefore, every instruction in this program or algorithm can be seen as a decision that is represented as a code. Turing envisioned all mathematics in this way, so that any problem or theorem is a unique code that can be manipulated to prove important results, such as the halting problem that will be discussed later.

We are nearly done creating our ACE. In our "computer" above, the human is not doing very much other than the mechanical tasks required by the detailed instructions provided. Thus, we now we replace the human with an (electro)mechanical device that will keep track of the awareness of the computation and one that can read/write on cells just like the human did. This device is very simple to construct as it simply read/writes cells and only follows the instructions provided as codes. For convenience of notation and description, we shall call this device the read/write head and the group of all cells as the tape or array. Though this is terminology that dates back to Turing's time of ticker tape machines, we can associate it with sequential access memory such as hard disks or magnetic tape drives. We may also refer to it as an array for a more modern interpretation because it is possible to represent one dimensional (1D), 2D, 3D and even n-dimensional arrays via 1D C-style arrays quite easily and therefore possible to take this representation into account in the instructions.

Now we have completed our ACE. By construction, it is the simplest computational machine capable of computing everything. It was Turing's work that first introduced this concept of the stored computer program in such a formal way. It also makes the bold assertion that *any* computation that a human can do can be reduced down to algorithms! We shall discuss these issues at the end of the chapter. For now, we will examine how Turing used these machines to prove that there exist programs that may never halt or arrive at a fixed solution.

3.2 The Halting Problem

We noted in the previous section that our Turing machine uses the concept of a stored program to compute. The machine repeatedly executes the instructions in this program until a solution is reached. But what if a

solution is never reached? How do we know that a solution is even possible for a given computation? Does there always exist a solution to every problem? In other words using the terminology of Turing machines, can we determine if a problem or computation will ever complete or halt? Just as for any other problem, we have already shown that we can create a Turing machine to compute a solution of this problem for us.

Let us create a Turing machine that computes whether or not a given problem is guaranteed to halt or not. This machine has an important property however, since its input is another Turing machine. These types of general machines are called universal Turing machines and Turing [1937] introduced them to prove the halting problem. The input of this machine is a Turing machine and the output is whether or not the computation will halt or not. We note that this machine T_H will halt if and only if the input machine halts, producing the output True. If the machine does not halt, we will need to let it loop until it is finished. Using the same argument above, we can run this universal Turing machine T_H inside another Turing machine T_D to detect and decide that the original Turing machine, the one input to T_H, is looping and tell us that it will not halt by returning True if T_H is looping.

We may test this setup with any normal Turing machine and it seems all is fine and well. That is until you realise that the input to T_H must be all Turing machines for our assertion that every program will halt, including our Turing machine T_D. Setting T_D as the input to itself causes a contradiction to occur. When T_D is supposed to halt it will loop, but at the same time when T_D will loop it is supposed to halt. Clearly, it cannot be in both states at the same time. This self-referential idea causes many problems in mathematical logic, including the question introduced by Bertrand Russell asking whether $\mathcal{R}$, the set of all sets that do not contain themselves, contains $\mathcal{R}$ itself? If $\mathcal{R}$ is a set of all sets that that does not contain itself, it must in $\mathcal{R}$, which is a contradiction.

The consequence of this theorem is that there will be particular computations, where it will be impossible to know if solutions exist and that programs will halt. This extends the incompleteness theorems Gödel [1931] to decidability, so that not only are there potentially theorems that cannot be proven but there will be problems that are not decidable either.

Till now, we have discussed Turing machines and the ideas behind their use in proving theorems. In the remaining sections of this chapter, we will construct these machines and design programs for them. In order to do this, we first investigate simpler machines that are possible, which do not compute everything, but are still useful for very simple specialised devices. These machines are devices that do not have any memory.

3.3 Finite State Machines

Consider having to design a device or system that only needs to decide whether to accept an input or reject it. Equivalently, you would like a system that produces a yes/on or no/off decision. In such a scenario, a machine that has a finite number of states and one that processes the input one element at a time is appropriate. This type of device that has no memory (in the sense of a tape or array) and is called a Finite State Machine (FSM).

3.3.1 State-based Computation

One of the simplest FSMs is the well known on/off switch that is capable of turning things on or off. We can represent a FSM as a state diagram using a directed graph that represents transitions between states

as edges such as that of figure 3.1. In this diagram, we can keep track of how the computation progresses

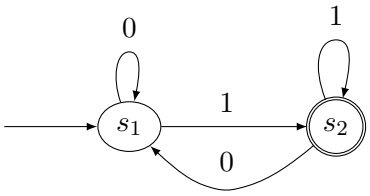

Figure 3.1: A graph representation of the FSM that is a simple on/off switch.

by following the arrows in the diagram depending on the input elements encountered one at a time. We start at an initial state s_1 and only end successfully if we end up in the accept state s_2, indicated by the double circle. For example, a string $w = 1101$, we can express the resulting computation as a series of state transitions:

1. Start computation in state s_1
2. Read first character in w, transition from s_1 to s_2 because it is a 1
3. Read next character in w, transition from s_2 to s_2 as it is 1
4. Read next character in w, transition from s_2 to s_1 as it is 0
5. Read next character in w, transition from s_1 to s_2 as it is 1
6. Reading is complete and we are in the accept state s_2
7. Accept the computation

When in the accept state, we have completed our computation successfully by processing the entire input string and therefore stop the machine. If we however end up in any other state, we reject the computation as a failure. Writing the above list of transitions as sentences is cumbersome, so there are more efficient representations such as the state diagram of figure 3.1 or as a transition table below.

	0	1
s_1	s_1	s_2
s_2	s_1	s_2

Table 3.1: Transition table for the FSM in figure 3.1.

We may also write the transitions symbolically also using the notation δ to represent a transition function, where he δ symbol is usually used to represent change in many areas of mathematics. The arguments to $\delta(s, a)$ would be the character being processed and the current state and it would return the new state. To summarise, the FSM in diagram 3.1 as transitions are

$$\delta(s_1, 0) = s_1, \qquad \delta(s_1, 1) = s_2 \tag{3.1}$$
$$\delta(s_2, 0) = s_1, \qquad \delta(s_2, 1) = s_2. \tag{3.2}$$

Although the transition table above defines how the computation will progresses for the string w, we should ask for what inputs are accepted by the machine in general and which ones are rejected. In this machine, it is clear that any binary string ending in 1 will be accepted, but this is not always so easy to determine for more complex machines. We will describe the significance of strings that are accepted by FSMs in the next section.

Formally, each FSM firstly consists of pre-defined set of states $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ for number of states n, which is used during the computation. This includes three important states that all FSMs have: an initial state s_0, an accept state s_{yes} and a reject state s_{no}. Secondly, we need the set of transitions or instructions δ for the FSM that defines what state the machine should go into when in a particular state and it reads a certain character from the input string. In notation, we mean that $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, i.e. state paired with character to another state. Lastly, we need to formally define the set of strings acceptable by FSMs and we discuss this in the next section.

3.3.2 Regular Languages

As we have ascertained in the previous section, the language of any machine input needs to be carefully defined to match the instructions and the FSM is no exception. First, we need to define the symbols that we will use for the input and the computation to apply on. The set of all possible unique symbols or characters is called the alphabet $\mathcal{A}$. The input is then a string of symbols concatenated together from the alphabet. For example, a binary FSM would have the alphabet $\mathcal{A} = \{0, 1\}$ and one of the possible strings as $w = 1101$ as we have seen before. We can then define the set of all finite strings of $\mathcal{A}$ as $\mathcal{A}^*$. The set $\mathcal{A}^*$ is an infinite set of all possible strings via the alphabet $\mathcal{A}$, each of finite length. An example of $\mathcal{A}^*$ for the alphabet $\mathcal{A} = 0, 1$ would be

$$\mathcal{A}^* = \{\epsilon, 0, 00, 1, 11, 01, \ldots\}, \tag{3.3}$$

where ϵ indicates the empty string, i.e. a string of length zero.

In order to make our notation more compact, it is possible to generalise our transition function to accept strings w not just characters of the alphabet, so that we can write $\delta^*(s, w)$. We can then define the transitions recursively by repeatedly breaking down the string w to the string $w = ua$, where u is a substring of w including all characters except a, which is the last character of w so that $a \in \mathcal{A}$, i.e. one of the unique characters of the alphabet. This gives us the notation $\delta(\delta^*(s, u), a)$. The notation is based around the last character of the string because it will decide whether the machine's final state will be an accept state or not. Formally, we can define the function recursively as

$$\delta^*(s, w) = \begin{cases} s, & \text{if } w = \epsilon \\ \delta(\delta^*(s, u), a), & \text{if } w = ua \text{ for } u \in \mathcal{A}^* \text{ and } a \in \mathcal{A} \end{cases} \tag{3.4}$$

We can then repeat our "breakdown" of the substring u in this way until the empty string remains, in which case do nothing more and return. This is equivalent to following the state diagram until the entire input string is processed and the accept state is reached.

If we define the set of final acceptable states as $\mathcal{S}^{yes}$, then the language of the machine M can then be defined as

$$L(M) = \{w \in \mathcal{A}^* \mid \delta^*(s_1, w) \in \mathcal{S}^{yes}\}. \tag{3.5}$$

Any language which is accepted by a FSM is known as a regular language. Conversely, a FSMs can only recognise a language if it is regular.

We may now put all these aspects together to formally define FSMs. A machine $M = (\mathcal{S}, \mathcal{A}, \delta, s_0, \mathcal{S}^{yes})$ is a FSM if it accepts an input string $w \in \mathcal{A}^*$ of length n from a series of states $r_0, r_1, \ldots, r_n$ in $\mathcal{S}$ such that

1. $r_0 = s_0$, i.e. starting from the initial state,
2. $\delta(r_i, a_i) = r_{i+1}$ for $0 \leqslant i < n - 1$ and $w = a_0 a_1 \ldots a_n$,
3. $r_n \in \mathcal{S}^{yes}$, i.e. ending up in an accept state.

In other words, starting from a state r_0, which is the initial state s_0, we process each character of w to the next state up to the last state. The last state must be an accept state for the FSM M to recognise the language L formed from the alphabet $\mathcal{A}$ and transitions δ.

3.3.3 Regular Operations

Having defined regular languages, it is important to consider the types of operations between languages allowed that preserve the regularity of the resulting language. In this text we will explore the result of combining two regular languages, namely their union and intersection. To determine if the result of combining two languages is regular, we may utilise the corresponding FSMs M_1 and M_2 of these languages to process an example input w via both machines simultaneously and establish if a new FSM M_3 is definable from the resulting operation.

3 Theorem (Regular Operations)
The operation of a union of two languages $L_1 \cup L_2$ with the same alphabet $\mathcal{A}$ is also a regular language L_3.

Proof: *Consider the corresponding FSMs M_1 and M_2 of the two languages having the states $\mathcal{S}_1 = \{s_1, s_2\}$ and $\mathcal{S}_2 = \{r_1, r_2\}$. Simulating a new machine M_3 that runs these two machines simultaneously implies that the machine M_3 is in a pair of states from both $\mathcal{S}_1$ and $\mathcal{S}_2$ at the same time at any one given moment. Given that the cardinality of $\mathcal{S}_1$ and $\mathcal{S}_2$ is both 2, the states of M_3 can only have the following four possible pair of states*

$$\mathcal{S}_3 = \{(s_1, r_1), (s_2, r_1), (s_1, r_2), (s_2, r_2)\} \tag{3.6}$$

But the pairs of states in above set $\mathcal{S}_3$ are themselves a unique combination of pairs and can mathematically substituted with different dummy variables q_1, q_2, q_3 and q_4. Since the alphabet of the two languages is the same, the alphabet and set of strings $\mathcal{A}^*$ of M_3 must be the same as well.

We can apply the same logic to the transitions functions of both machines δ_1 and δ_1 to create a new function δ_3 that describes the paired transition of both machines as the input is processed from states q_i to q_j. For example given a character a from input w, we could have $\delta_3((s_1, r_1), a) = (s_1, r_2)$ in our example using $\mathcal{S}_1$ and $\mathcal{S}_2$ above assuming that $\delta_1(s_1, a) = s_1$ and $\delta_2(r_1, a) = r_2$.

Lastly, we may define our start and final states as suggest a combination of M_1 and M_2 states, since we are running both states at once. Thus $\mathcal{S}_3^{yes} = \mathcal{S}_1^{yes} \times \mathcal{S}_2^{yes}$ and likewise for the initial state. Therefore, if we extend our sets $\mathcal{S}_1$ and $\mathcal{S}_2$ to be set of states for any arbitrary FSMs, we can see that we will always be able to define a new machine M_3 that fulfils the criteria for a FSM and hence is also a regular language. ◀

A similar argument to theorem 3 on the previous page can be made for the intersection of languages. In the next section, we explore how parallel and branched processing can be done via FSMs.

3.3.4 Non-deterministic FSMs

In our example of figure 3.1 on page 21, what would we do if there were multiple transition edges from one state to another for the same character encountered? We would have two possible transitions from a state to another and how do we decide which one to take? In our previous FSM had no choices between edges to follow because there were only single edges between states and the computation is unambiguous, i.e. $\mathcal{S} \times \mathcal{A} \to \mathcal{S}$. We call these machines deterministic FSMs and machines with multiple edges or transitions (i.e. branches) between states for the same character are known as non-deterministic FSMs.

Computations involving non-deterministic FSMs follow all possible paths equally, so that when multiple edges are encountered, parallel processes are spawned for each edge. In other words, a non-deterministic FSM is composed of multiple deterministic FSMs with the all possible computations conducted depending on the input string. Any computation process that eventually accepts the input results in the non-deterministic FSM accepting the input also. It may seem that non-deterministic FSMs are more powerful, but they can only do what FSMs can do, though it may be easier to handle differing inputs. We need to add more elements to FSMs to make them more powerful.

3.4 Turing Machines

We saw that FSMs have an alphabet $\mathcal{A}$, associated states $\mathcal{S}$ and transition rules $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. It does not however, have memory for working as a human would need to do arbitrary computations. This would also require keeping track of the "awareness" within that memory. Thus, we define a more powerful machine that has a tape for memory and a read/write head for keeping track of awareness, which is known as a Turing machine.

3.4.1 Memory

The tape is a 1D array that can be thought of as sequential access memory. The read/write head moves left or right along the array and points to a cell in the tape that corresponds to the current computation taking place. The tape may hold the input and output strings, any intermediate results or working and even the program as we shall see later in this section. Figure 3.2 on the next page shows an illustration of the tape and head.

For theoretical reasons, we may consider the tape to be infinite in length. Unlike FSMs, the head also permits us to back along the input and mark on the tape. In fact, we are permitted to write any symbol we

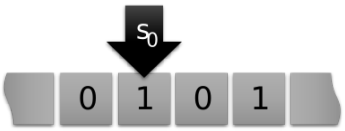

Figure 3.2: An illustration of a Turing machine with its tape and read/write head. The tape extends to infinity on both sides.

like as long as we define a tape alphabet $\mathcal{A}_T$ that is a superset of symbols permitted on the tape with the input alphabet $\mathcal{A}_i$. For example, for the binary alphabet $\mathcal{A}_i = \{0, 1\}$, we could have the tape alphabet $\mathcal{A}_T = \{0, 1, \star\}$, where we use the $\star$ symbol to represent intermediate results or working. Thus, we can write the working or intermediate results, as well as the input on the tape with unrestricted access. This allows us to conduct more powerful operations via more powerful transition functions.

3.4.2 Programs

With memory available for working and ability to read/write and move the head, we have more degrees of freedom with our transitions. Indeed, now our transitions have the form $\delta : \mathcal{S} \times \mathcal{A}_T \to \mathcal{S} \times \mathcal{A}_T \times \{L, R\}$, where L and R represent the left and right movement of the head.

Finally, the Turing machine also has an initial state s_0 like an FSM, but only has single accept and reject states, s_a and s_r respectively, because we have the ability to complete the computation instantly without having to process the entire input strin since we are free to move back and forth along the input. An example of such a computation that halts immediately is a Turing machine that checks the input contains at least a single 1 in the input string. As you can imagine, once a single 1 is encountered, regardless the length of the string and how much of the string has been processed, further computation is not required and the machine should accept and halt. Thus, we can formally define the concept of a Turing machine.

1 **Definition (Turing Machine)**
A Turing machine is a seven tuple $T = (\mathcal{S}, \mathcal{A}_i, \mathcal{A}_T, \delta, s_0, s_a, s_r)$, where the sets $\mathcal{S}, \mathcal{A}_i$ and $\mathcal{A}_T$ are all finite and

1. *$\mathcal{S}$ is the set of all ppossible states,*
2. *$\mathcal{A}_i$ is the input alphabet without the blank symbol,*
3. *$\mathcal{A}_T$ is the tape alphabet which includes the blank symbol,*
4. *$\delta : \mathcal{S} \times \mathcal{A}_T \to \mathcal{S} \times \mathcal{A}_T \times \{L, R\}$ is the transition function,*
5. *$s_0 \in \mathcal{S}$ is the initial state,*
6. *$s_a \in \mathcal{S}$ is the accept state,*
7. *$s_r \in \mathcal{S}$ is the reject state with $s_r \neq s_a$.*

The Turing machine would function as the following:

1. We assume that the tape is infinite in both directions.

2. Begin with the input placed in the middle of the tape with the head in the left most cell of the input. The rest of the tape is considered empty or blank.

3. The input alphabet $\mathcal{A}_i$ does not have the blank symbol, so it is safe to assume that the end of the input is marked with a blank symbol.

4. The computation begins by using the tape alphabet $\mathcal{A}_T$ and following the transition function until either the accept or reject state is reached.

5. When either of s_a or s_r is reached, the machine halts. Otherwise, the machine continues forever without halting.

Some implementations of Turing machine tape is one-sided, so that it is only infinite to the right. Turing [1937] does not mention how the tape should be implemented and indeed it makes very little difference to the computing power of the machine. In this one-sided scenario however, if the head tries to move off the tape, it merely stops at the closest cell of the tape and does not move.

3.4.3 Configurations

To track the progress and to debug Turing machine processes, it is often useful to include a notation for the current "configuration" of the machine. This is most commonly done by showing the current state of the tape that is in use and marking the current head position with the state name. Figure 3.3 shows an example of the configuration of a Turing machine in the state s_7. This notation allows us to write a plain

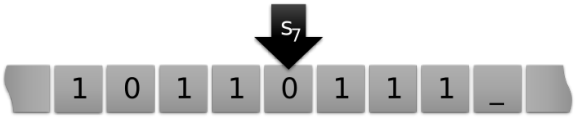

Figure 3.3: An example of a Turing machine configuration showing the current state, tape contents and read/write head position. This configuration can be written in plain text as $1011s_70111$.

text description of the machine configuration easily for each step and follow the computation especially for debugging purposes.

We will close this section by showing an example of Turing machines that uses most of the concepts above. Consider a Turing machine whose task is to determine if the left side of the input is the same as the right side, where the hash symbol $\#$ is used as the separator. For example, the input string $011\#011$ should be accepted by the Turing machine. The algorithm of the Turing machine would be something like:

1. Check if the first character of the input is a hash, if it is then accept and halt,

2. Otherwise, cross of the first character with a symbol for crossing, say $\star$,

3. Enter state designated for the character and move right until the separator is found, e.g. with s_1 for 0 and s_2 for 1, then $\star11\#s_1011$,

4. If the first character encountered after the hash and any crosses matches the state of the encountered character, cross off and continue, else reject and halt,
5. Move left until the first cross is encountered after the hash and go back to initial state, e.g. $\star s_0 11 \# \star 11$,
6. Repeat until no non-crossed characters are left, e.g. $\star\star\star s_0 \# \star\star\star$,
7. If only crosses are left either side of the hash, then accept and halt.

In the next section, we will look at ways to try increase the power of the Turing machines if possible and explore the most universal of all Turing machines.

3.5 Universal Turing Machines

3.6 Hypercomputation

Bibliography

Berlekamp, E., J.Conway, R.Guy, 1982. Winning Ways for your Mathematical Plays. Vol. 2.

Church, A., 1932. A set of postulates for the foundation of logic. Annals of Mathematics 33 (2), 346–366.
URL https://doi.org/10.2307/1968337

Diophantus, 100. Arithmetica. Springer-Verlag Berlin and Heidelberg GmbH & Co. KG (December 31, 1982).

Euclid, 300BCE. The Elements.

Euler, L., 1763. Theoremata Arithmetica Nova Methodo Demonstrata. Novi Commentarii Academiae Scientiarum Petropolitanae 8, 74–104.

Feynman, R., 2005. The Pleasure of Finding Things Out: The Best Short Works of Richard P. Feynman. Helix Books.
URL https://www.amazon.com/Pleasure-Finding-Things-Out-Richard-ebook/dp/B005QBLHB4

Gardner, M., 1970. Mathematical games: The fantastic combinations of john conway□s new solitaire game □life□. Scientific American (223), 120–123.

Gauss, C. F., 1801. Disquisitiones Arithmeticae. Yale Univeristy Press.

Gödel, K., Dec 1931. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. Monatshefte für Mathematik und Physik 38 (1), 173–198.
URL https://doi.org/10.1007/BF01700692

Klein, F., Mar 1893. Vergleichende betrachtungen über neuere geometrische forschungen. Mathematische Annalen 43 (1), 63–100.
URL https://doi.org/10.1007/BF01446615

Langton, C. G., 1986. Studying artificial life with cellular automata. Physica D: Nonlinear Phenomena 22 (1), 120 – 149, proceedings of the Fifth Annual International Conference.
URL https://doi.org/10.1016/0167-2789(86)90237-X

Sipser, M., 2013. Introduction to the theory of computation / Michael Sipser., 3rd Edition. Cengage Learning, Andover.

Turing, A. M., 1937. On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society s2-42 (1), 230–265.
URL https://doi.org/10.1112/plms/s2-42.1.230