History of CS

The word computer was first used for people who could perform calculations or computations back in 17-th century. In the end of 19-th century, people realized that machines could perform same computations without errors and by far faster than any group of human computers.

One of most famous mathematicians and a founder of Computer Science is Alan Turing. Even though, he did not invent the first computer he had established fundamental theories of computing and computation and without it we would not have the computers we use today. He is more known as an inventor of Turing machine, which I’ll cover later. Also Alan Turing have designed an algorithm as well a machine that uses it in order to decipher Enigma code, which was used by Nazis to communicate during World War II thus shortening the war by at least 2 years. Turing is also credited as a Father of Computer Science. Since 1966, the Turing Award has been given annually by the Association for Computing Machinery to a person for technical contributions to the computing community. It is widely considered to be the computing world's highest honor, equivalent to the Nobel Prize.

**Turing Machines**

*First published Thu Sep 14, 1995; substantive revision Tue Jun 26, 2012*

Turing machines, first described by [Alan Turing](http://plato.stanford.edu/entries/turing/) in (Turing 1937), are simple abstract computational devices intended to help investigate the extent and limitations of what can be computed.

Turing was interested in the question of what it means for a task to be computable, which is one of the foundational questions in the [philosophy of computer science](http://plato.stanford.edu/entries/computer-science/). Intuitively a task is computable if it is possible to specify a sequence of instructions which will result in the completion of the task when they are carried out by some machine. Such a set of instructions is called an *effective procedure*, or *algorithm*, for the task. The problem with this intuition is that what counts as an effective procedure may depend on the capabilities of the machine used to carry out the instructions. In principle, devices with different capabilities may be able to complete different instruction sets, and therefore may result in different classes of computable tasks (see the entry on [computability and complexity](http://plato.stanford.edu/entries/computability/)).

Turing proposed a class of devices that came to be known as Turing machines. These devices lead to a formal notion of computation that we will call *Turing-computability*. A task is Turing computable if it can be carried out by some Turing machine.

The proposition that Turing's notion captures exactly the intuitive idea of effective procedure is called the [Church-Turing thesis](http://plato.stanford.edu/entries/church-turing/). This proposition is not provable, since it is a claim about the relationship between a formal concept and intuition. The thesis would be refuted by an intuitively acceptable algorithm for a task that is not Turing-computable, and no such counterexample has been found. Other independently defined notions of computability based on alternative foundations, such as [recursive functions](http://plato.stanford.edu/entries/recursive-functions/) and abacus machines have been shown to be equivalent to Turing-computability. These two facts indicate that there is at least something natural about this notion of computability.

Turing machines are not physical objects but mathematical ones. We require neither soldering irons nor silicon chips to build one. The architecture is simply described, and the actions that may be carried out by the machine are simple and unambiguously specified. Turing recognized that it is not necessary to talk about *how* the machine carries out its actions, but merely to take as given the twin ideas that the machine can carry out the specified actions, and that those actions may be uniquely described.

## 2. Describing Turing Machines

Every Turing machine has the same machinery. What makes one Turing machine perform one task and another a different task is the table of transition rules that make up the machine's program, and a specified initial state for the machine. We will assume throughout that a machine starts in the lowest numbered of its states.

We can describe a Turing machine, therefore, by specifying only the 4-tuples that make up its program. Here are the tuples describing a simple machine.

|  |
| --- |
| ⟨ *s*0, 1, *s*0, » ⟩ |
| ⟨ *s*0, 0, *s*1, 1 ⟩ |
| ⟨ *s*1, 1, *s*1, « ⟩ |
| ⟨ *s*1, 0, *s*2, » ⟩ |

This machine has three states, numbered *s*0, *s*1 and *s*2. The first two instructions describe what happens in state *s*0. There are two possibilities, either the machine is scanning a ‘1’, in which case the head moves to the right and stays in state *s*0. The machine leaves state *s*0 and enters *s*1 if it is scanning a ‘0’. It writes a ‘1’ on that transition. The second two instructions describe what happens in state *s*1, namely if it is scanning a ‘1’ the machine moves the head to the left staying in state *s*1. If it is scanning a ‘0’, the head moves to the right and the machine moves into state *s*2. Since there are no instructions for state *s*2, the machine halts if it reaches that state.

## 3. Varieties of Turing Machines

We have presented here one of the most common formulations of Turing's basic idea. There are a number of variations to the formulation that turn out to be equivalent to this one, and different authors present Turing machines using any of these. Since they are all provably equivalent to one another we can consider any of the formulations as being the definition of Turing machine as we find convenient.

Formulation F1 and formulation F2 are equivalent if for every machine described in formulation F1 there is machine a described in F2 which has the same input-output behavior, and vice versa, i.e., when started on the same tape at the same cell, will terminate with the same tape on the same cell.

**Two-way infinite tapes**

In our original formulation we specified that the tape had an end, at the left say, and stretched infinitely far to the right. Relaxing this stipulation to allow the tape to stretch infinitely far to right and left results in a new formulation of Turing machines. You might expect that the additional flexibility of having a two-way infinite tape would increase the number of functions that could be computed, but it does not. If there is a machine with a two-way infinite tape for computing some function, there there is machine with a one-way infinite tape that will compute that same function.

**Arbitrary numbers of read-write heads**

Modifying the definition of a Turing machine so that the machine has several read-write heads does not alter the notion of Turing-computability.

**Multiple tapes**

Instead of a single infinite tape, we could consider machines possessing many such tapes. The formulation of such a machine would have to allow the tuples to specify which tape is to be scanned, where the new symbol is to be written, and which tape head is to move. Again this formulation is equivalent to the original.

**Two-dimensional tapes**

Instead of a one-dimensional infinite tape, we could consider a two-dimensional “tape”, which stretches infinitely far up and down as well as left and right. We would add to the formulation that a machine transition can cause the read-write head to move up or down one cell in addition to being able to move left and right. Again this formulation is equivalent to the original.

**Arbitrary movement of the head**

Modifying the definition of a Turing machine so that the read-write head may move an arbitrary number of cells at any given transition does not alter the notion of Turing-computability.

**Arbitrary finite alphabet**

In our original formulation we allowed the use of only two symbols on the tape. In fact we do not increase the power of Turing machines by allowing the use of any finite alphabet of symbols.

**5-tuple formulation**

A common way to describe Turing machines is to allow the machine to both write and move its head in the same transition. This formulation requires the 4-tuples of the original formulation to be replaced by 5-tuples

⟨ State0, Symbol, Statenew, Symbolnew, Move ⟩

where Symbolnew is the symbol written, and Move is one of « and ».

Again, this additional freedom does not result in a new definition of Turing-computable. For every one of the new machines there is one of the old machines with the same properties.

**Non-deterministic Turing machines**

An apparently more radical reformulation of the notion of Turing machine allows the machine to explore alternatives computations in parallel. In the original formulation we said that if the machine specified multiple transitions for a given state/symbol pair, and the machine was in such a state then it would halt. In this reformulation, all transitions are taken, and all the resulting computations are continued in parallel. One way to visualize this is that the machine spawns an exact copy of itself and the tape for each alternative available transition, and each machine continues the computation. If any of the machines terminates successfully, then the entire computation terminates and inherits that machine's resulting tape. Notice the word successfully in the preceding sentence. In this formulation, some states are designated as accepting states and when the machine terminates in one of these states, then the computation is successful, otherwise the computation is unsuccessful and any other machines continue in their search for a successful outcome.

The addition of non-determinism to Turing machines does not alter the definition of Turing-computable.

Turing's original formulation of Turing Machines used the 5-tuple representation of machines. Post introduced the 4-tuple representation, and the use of a two-way infinite tape.

**A more complex machine**

In addition to performing numerical functions using unary representation for numbers, we can perform tasks such as copying blocks of symbols, erasing blocks of symbols and so on. Here is an example of a Turing machine which when started in standard configuration on a tape containing a single block of ‘1’s, halts on a tape containing two copies of that block of ‘1’s, with the blocks separated by a single ‘0’. It uses an alphabet consisting of the symbols ‘0’, ‘1’ and ‘A’.

## 4. What Can Be Computed

Turing machines are very powerful. For a very large number of computational problems, it is possible to build a Turing machine that will be able to perform that computation. We have seen that it is possible to design Turing machines for arithmetic on the natural numbers, for example.

**Computable Numbers**

Turing's original paper concerned computable numbers. A number is Turing-computable if there exists a Turing machine which starting from a blank tape computes an arbitrarily precise approximation to that number. All of the algebraic numbers (roots of polynomials with algebraic coefficients) and many transcendental mathematical constants, such as e and π are Turing-computable.

**Computable Functions**

As we have seen, Turing machines can do more than write down numbers. Among other things they can compute numeric functions, such as the machine for addition (presented in [Figure 2](http://plato.stanford.edu/entries/turing-machine/#Addition)) multiplication, proper subtraction, exponentiation, factorial and so on.

The characteristic function of a predicate is a function which has the value TRUE or FALSE when given appropriate arguments. An example would be the predicate ‘IsPrime’, whose characteristic function is TRUE when given a prime number, 2, 3, 5 etc and FALSE otherwise, for example when the argument is 4, 9, or 12. By adopting a convention for representing TRUE and FALSE, perhaps that TRUE is represented as a sequence of two ‘1’s and FALSE as one ‘1’, we can design Turing-machines to compute the characteristic functions of computable predicates. For example, we can design a Turing machine which when started on a tape representing a number terminates with TRUE on the tape if and only if the argument is a prime number. The results of such functions can be combined using the using the boolean functions: AND, NOT, OR, IF-THEN-ELSE, each of which is Turing-computable.

In fact the Turing-computable functions are just the recursive functions, described below.

**Universal Turing Machines**

The most striking positive result concerning the capabilities of Turing machines is the existence of Universal Turing Machines (UTM). When started on a tape containing the encoding of another Turing machine, call it T, followed by the input to T, a UTM produces the same result as T would when started on that input. Essentially a UTM can simulate the behavior of any Turing machine (including itself).

One way to think of a UTM is as a programmable computer. When a UTM is given a program (a description of another machine), it makes itself behave as if it were that machine while processing the input.

Note again, our identification of input-output equivalence with “behaving identically”. A machine T working on input t is likely to execute far fewer transitions that a UTM simulating T working on t, but for our purposes this fact is irrelevant.

In order to design such a machine, it is first necessary to define a way of representing a Turing machine on the tape for the UTM to process. To do this we will recall that Turing machines are formally represented as a collection of 4-tuples. We will first design an encoding for individual tuples, and then for sequences of tuples.

## 5. What Cannot Be Computed

In the previous section we described a way of encoding Turing machines for input to a Universal Turing Machine. The encoding of a Turing machine is a sequence of ‘0’s and ‘1’ and any such sequence can be interpreted as a natural number. We can think of the encoding of a Turing machine as being a natural number which is the serial number of that machine. Because of the way the encoding works, each Turing machine will have a distinct serial number. Since all of the serial numbers are natural numbers, the number of distinct Turing machines is countably infinite.

On the other hand, the number of functions on the natural numbers is uncountable. There are (uncountably) more functions on the natural numbers than there are Turing machines, which shows that there are uncomputable functions, functions whose results cannot be computed by any Turing machine, because there are simply not enough Turing machines to compute the functions.

This proof by counting is somewhat unsatisfactory, since it tells us that there are uncomputable functions, but provides us with no examples. Here we give two examples of uncomputable functions.

### 5.1 The Busy Beaver

Imagine a Turing machine that is started on a completely blank tape, and eventually halts. If the machine leaves n ones on the tape when it halts, we will say that the productivity of this machine is n. We will say that the productivity of any machine that does not halt is 0. Productivity is a function from Turing machine descriptions (natural numbers) to natural numbers. We will write p(T)=n to indicate that the productivity of machine T is n.

Among the Turing machines that have a particular number of states, there is a maximum productivity that a Turing machine with that number of states can have. This too is a function from natural numbers (the number of states) to natural numbers (the maximum productivity of a machine with that number of states). We will write this function as BB(k)=n to indicate the maximum productivity of a k-state Turing machine is n. There may be multiple different k state machines with the maximum productivity n. We call any of these machines a Busy Beaver for k.

There is no Turing machine which will compute the function BB(k), i.e., which when started in standard configuration on a tape with k ‘1’s will halt in standard configuration on a tape with BB(k) ‘1’s. This example is due to Tibor Radó (Radó 1962).

The proof that there is no such function proceeds by assuming that there is such a machine, i.e. that there is a machine which starts in standard configuration with k ‘1’s on the tape, and halts in standard configuration with BB(k) ‘1’s on the tape. We will call this machine B and assume that it has k states.

There is an n-state machine which writes n ‘1’s on an initially blank tape (exercise for the reader). We can construct a new machine which connects the halting state of this machine to the start state of B and then connecting the halting state of B to the start state of another copy of B. So the first machine writes n ‘1’ and then the first copy of B computes BB(n), but then the second copy of B takes over and computes BB(BB(n)). The total number of states in our machine is n+2k. Our machine may be a Busy Beaver for n+2k, but it is certainly no more productive than such a machine. So (if the Busy Beaver machine exists)

BB(n+2k) ≥ BB(BB(n)), for any n.

It is easy to show that the productivity of Turing machines increases as states are added, i.e.,

if i < j, then BB(i) < BB(j)

(another exercise). Consequently (if the Busy Beaver machine exists)

n+2k ≥  BB(n), for any n.

Since this is true for any n, it is true for n+11, yielding:

n+11+2k ≥ BB(n+11), for any n.

But it is easy to show that BB(n+11) ≥ 2n (another exercise, but show that there is an eleven state machine for doubling the number of ‘1’ on the tape, and compose such a machine with the n-state machine for writing n ‘1’s). Combining this fact with the previous inequality we have:

n+11+2k ≥  BB(n+11) ≥ 2n, for any n.

from which by subtracting n from both sides we have 11+2k ≥ n, for any n, if the Busy Beaver exists, which is a contradiction.

Even though the productivity function is uncomputable, there is considerable interest in the search for Busy Beaver Turing machines (most productive machines with a given number of states). Some candidates can be found by following links in the [Other Internet resources](http://plato.stanford.edu/entries/turing-machine/#Oth) section of this article.

### 5.2 The Halting Problem

It would be very useful to be able to examine the description of a Turing machine and determine whether it halts on a given input. This problem is called the Halting problem and is, regrettably, uncomputable. That is, no Turing machine exists which computes the function h(t,n) which is defined to be TRUE if machine t halts on input n and FALSE otherwise.

To see the uncomputability of the halting function, imagine that such a machine H exists, and consider a new machine built by composing the copying machine of [Figure 4](http://plato.stanford.edu/entries/turing-machine/#Copycat) with H by joining the halt state of the copier to the start state of H. Such a machine, when started on a tape with n ‘1’s determines whether the machine whose code is n halts when given input n, i.e., it computes M(n) = h(n,n).

Now lets add another little machine to the halt state of H. This machine goes into an infinite sequence of transitions if the tape contains TRUE when it starts, and halts if the tape contains FALSE (its an exercise for the reader to construct this machine, assume that TRUE is represented by ‘11’, and FALSE by ‘1’).

This composed machine, call it M, halts if the machine with the input code n does not halt on an initial tape containing n (because if machine n does not halt on n, the halting machine will leave TRUE on the tape, and M will then go into its infinite sequence), and vice versa.

To see that this is impossible, consider the code for M itself. What happens when M is started on a tape containing Ms code? Assume that M halts on M, then by the definition of the machine M it does not halt. But equally, if it does not halt on M the definition of M says that it should halt.

This is a contradiction, and the Halting machine cannot exist. The fact that the halting problem is not Turing-computable was first proved by Turing in (Turing 1937). Of course this result applies to real programs too. There is no computer program which can examine the code for a program and determine whether that program halts.