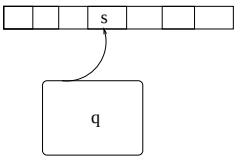
Turing Machines and Algorithms

In Computer Science, we develop algorithms for problems we encounter. In your studies so far, you have probably never *rigorously* defined what is meant by an "algorithm". This lack of a formal definition usually doesn't matter because most of the procedures we develop are obviously algorithms. Informally, an algorithm is a sequence of steps which, when followed, solve the problem or perform the action desired. However, once we want to start arguing that algorithms (for a particular problem) *don't* exist, we have to be completely rigorous about what we mean by an algorithm.

In 1936, Alan Turing gave the first definition of an algorithm, in the form of (what we call today) a Turing machine. He wanted a definition that was simple, clear, and sufficiently general.

First, we introduce some notation.
Let Σ be a finite alphabet of symbols, By Σ^* we mean the set of all finite strings (including the empty string) consisting of elements of Σ. By a *language* over Σ we mean a subset $L \subseteq \Sigma^*$. For a string $x \in \Sigma^*$, we denote the length of x by $|x|$.

Turing machines



A Turing machine consists of a two-way infinite tape and a finite state control. The tape is divided up into squares, each of which holds a symbol from a finite tape alphabet that includes the blank symbol $\not b$.

The machine has a read/write head that is connected to the control, and that that scans squares on the tape. Depending on the state of the control and symbol scanned, it makes a move, consisting of

- printing a symbol
- moving the head left or right one square
- assuming a new state

The tape will initially be completely blank, except for an input string over the finite input alphabet Σ; the head will initially be pointing to the leftmost symbol of the input.

A Turing machine M is specified by giving the following:

- Σ (a finite input alphabet).

- Γ (a finite tape alphabet). $\Sigma \subseteq \Gamma$. $\not b \in \Gamma - \Sigma$.
- Q (a finite set of states). There are 3 special states:
  - q_0 (the initial state)
  - q_{accept} (the state in which M halts and accepts)
  - q_{reject} (the state in which M halts and rejects)
- $\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ (the transition function)
  (If $\delta(q, s) = (q', s', h)$, this means that if q is the current state and s is the symbol being scanned, then q' is the new state, s' is the symbol printed, and h is either L or R, corresponding to a left or right move by one square.

The Turing machine M works as follows on input string $x \in \Sigma^*$.
Initially x appears on the tape, surrounded on both sides by infinitely many blanks, and with the head pointing to the leftmost symbol of x. (If x is the empty string, then the tape is completely blank and the head is pointing to some square.) The control is initially in state q_0.
M moves according to the transition function δ.
M may run forever, but if it halts, then it halts either in state q_{accept} or q_{reject}. (For the moment, we will only be interested in whether M accepts or rejects; later, we will explain what it means for M to output a string.)

Definition 1 *M accepts a string $x \in \Sigma^*$ if M with input x eventually halts in state q_{accept}. We write $\mathcal{L}(M) = \{x \in \Sigma^* | M \text{ accepts } x\}$, and we refer to $\mathcal{L}(M)$ as the language accepted by M.*

(Notice that we are abusing notation, since we refer both to a Turing machine accepting a string, and to a Turing machine accepting a language, namely the set of strings it accepts.)

Example 1 PAL = the set of even length palindromes =
$\{yy^r | y \in \{0,1\}^*\}$, where y^r means y spelled backwards.

We will design a Turing machine M that accepts the language PAL$\subseteq \{0,1\}^*$. M will have input alphabet $\Sigma = \{0,1\}$, and tape alphabet $\Gamma = \{0,1,\not b\}$. (Usually it is convenient to let Γ have a number of extra symbols in it, but we don't need to for this simple example.) We will have state set $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$.

If, in state q_0, M reads 0, then that symbol is replaced by a blank, and the machine enters state q_1; the role of q_1 is to go to the right until the first blank, remembering that the symbol 0 has most recently been erased; upon finding that blank, M enters state q_2 and goes left one square looking for symbol 0; if 0 is not there then we *reject*, but if 0 is there, then we erase it and enter state q_3 and go left until the first blank; we then go right one square, enter state q_0, and continue as before.

If we read 1 in state q_0, then we operate in a manner similar to that above, using states q_4 and q_5 instead of q_1 and q_2.

If we read $\not b$ in state q_0, this means that all the characters of the input have been checked, and so we accept.

Formally, the transition function δ is as follows. (Note that when we accept or reject, it doesn't matter what we print or what direction we move in; we arbitrarily choose to print $\not b$ and move right in these cases.)

State q	*Symbol* s	*Action* $\delta(q,s)$
q_0	0	$(q_1, \not b, R)$
q_1	0	$(q_1, 0, R)$
q_1	1	$(q_1, 1, R)$
q_1	$\not b$	$(q_2, \not b, L)$
q_2	1	$(q_{reject}, \not b, R)$
q_2	$\not b$	$(q_{reject}, \not b, R)$
q_2	0	$(q_3, \not b, L)$
q_3	0	$(q_3, 0, L)$
q_3	1	$(q_3, 1, L)$
q_3	$\not b$	$(q_0, \not b, R)$
q_0	1	$(q_4, \not b, R)$
q_4	0	$(q_4, 0, R)$
q_4	1	$(q_4, 1, R)$
q_4	$\not b$	$(q_5, \not b, L)$
q_5	0	$(q_{reject}, \not b, R)$
q_5	$\not b$	$(q_{reject}, \not b, R)$
q_5	1	$(q_3, \not b, L)$
q_0	$\not b$	$(q_{accept}, \not b, R)$

We now define the notion of worst case time complexity of Turing machines.

Let M be a Turing machine over input alphabet Σ. For each $x \in \Sigma^*$, let $t_M(x)$ be the number of steps required by M to *halt* (i.e., terminate in one of the two final states) on input x. (Each step is an execution of one instruction of the machine, and we define $t_M(x) = \infty$ if M never halts on input x.)

Definition 2 (Worst case time complexity of M) *The worst case time complexity of M is the function $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined by*

$$T_M(n) = \max\{t_M(x) \mid x \in \Sigma^*, |x| = n\}.$$

Let us return to the Turing machine M from our palindrome example, and try to estimate $T_M(n)$. Calculating $T_M(n)$ exactly is difficult, but we can find reasonable upper and lower bounds for it. To compute an upper bound, consider an arbitrary input x of length n; M

will make less than or equal to $n+1$ passes over x, each taking less than or equal to $n+1$ steps, so $t_M(x) \leq (n+1)^2$, so $T_M(n) \in O(n^2)$. For a lower bound, consider a palindrome of length n (where n may or may not be even). M will go to the right for $n+1$ steps, then left for n steps, etc., for a total of $(n+1)+(n)+\cdots+1 \in \Omega(n^2)$ steps.
So $T_M(n)$ is in $\Theta(n^2)$.

(Notice that M performs much faster on some strings than on others. For example, if $x = 10000000$, then $t_M(x) = 10$. In general, if the first and the last symbols of x differ, then $t_M(x) = n+2$.)

When we talk about Turing machines accepting a language, we implicitly are only concerned, for each input, with whether or not the input is accepted, that is, with whether we say "yes" or "no" on the input.

Often, however, we want our algorithm to output a *string*, so we need a convention whereby a Turing machine, when it halts, can be viewed as outputting a string. (Note that when we view a Turing machine as outputting a string, we will not care if the final state is accepting or not.) We will let the the output alphabet be the same as the input alphabet Σ. If and when machine M halts, we define the *output* to be the string $y \in \Sigma^*$ beginning at the head position, and continuing to the right up to but not including the first symbol not in Σ; since only a finite portion of the tape is non-blank and since $\not\flat \notin \Sigma$, such a symbol will exist.

We say that M computes $f : \Sigma^* \to \Sigma^*$ if for every $x \in \Sigma^*$, M computing on input x eventually halts with output $f(x)$.

Before continuing, we should reflect on the reasonableness of our definitions. After all, Turing machines are pretty inconvenient to program, and the programs we get often seem inefficient. For this reason we will introduce a "suped-up" version of a Turing machine, called a "multi-tape Turing machine"; these machines are easier to program and are more efficient than normal Turing machines, but their power is just the same as normal Turing machines, and there is not a *significant* change in efficiency.

A multitape Turing machine M uses k tapes, for some constant $k \geq 1$. The first tape is the input-output tape, according to the same conventions as with a normal Turing machine; the other $k-1$ tapes are initially blank. The Finite State Control starts in state q_0 and has k heads, one reading/writing each tape. The move M makes depends on the state and on the k symbols being read; a move consists of writing on each tape, moving each head, and going to a new state. More formally the transition function is:

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times (\Gamma^k) \to Q \times (\Gamma \times \{L, R\})^k$$

As an example, consider (a description of) a 2-tape machine M for accepting PAL:
M begins by checking that the input x is of even length, and if not rejects. Then M copies x onto the second tape, and positions the head on tape 1 to the left of x and positions the head on tape 2 to the right of x. Then M moves one head to the right while moving the other to the left, checking that the symbols match; if they all match, M accepts, otherwise

M rejects.
For this machine we have linear (worst-case) running time: $T_M(n) \in \Theta(n)$.

It turns out that we can always simulate a multitape Turing machine by a normal Turing machine, with the running time at worst squaring.

Theorem 1 *For every multitape Turing machine M there is a normal Turing machine M' such that $T_{M'}(n) \in O((T_M(n))^2)$.*

Proof Outline:
We want to simulate a k-tape machine M by a 1-tape machine M'. One way of doing this is as follows. A portion of the tape of M' will contain all the information about the squares of the tapes of M that have been visited. We will think of (this part of) the tape of M' as containing k (vertical) "tracks", where the ith track contains information about the ith tape of M, including the position of the head; a tape symbol of M' will therefore actually be a k-tuple of symbols. If on a square of M' the ith track has symbol a, this will mean that the corresponding square of tape i of M contains a; if on a square of M' the ith track has symbol a^*, this will mean that the corresponding square of tape i of M contains a and has the head pointing to it. If M has visited s tape squares so far, then M' will take time $O(s)$ to simulate the next step of M.

So on an input of length n, M' will have to simulate $T_M(n)$ steps, taking time $O(T_M(n))$ on each step, for a total time of $O((T_M(n))^2)$. □

Nondeterministic Turing machines

As a final justification of our definitions, let us consider a new model of Turing machine in which we give the Turing machine an ability that no modern computer has - the ability to always "know" the next proper step in the algorithm.

Instead of having a transition function deterministically indicate what the next move of the Turing machine is based on the current state and tape symbol, we will have the transition function present our Turing Machine with a finite set of possible moves and we will assume that the machine has the ability to always choose the best possible move from the choices presented.

The model for a nondeterminsitic Turing machine is similar to the deterministic Turing machine except that the transition function now maps to a set of moves instead of a single move.

- $\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$

Where $\mathcal{P}$ represents the *power set.*

A nondeterministic Turing machine M accepts an string x if there exists any sequence of moves of M on input x such that M enters the state q_{accept}.

Another way to think of a nondeterministic Turing machine is of parallel executions of a deterministic machine. When the machine faces a choice of moves, we assume the machine splits into multiple copies of itself and it follows all of the possible choices in parallel. This splitting repeats every time a choice is faced. If any one of the parallel copies ends up in the accepting state, we consider the machine to have accepted the input, and we discard all the other copies of the machine.

Even though it appears that nondeterminism should give the Turing machine a huge computational advantage, it turns out that everything which can be computed by a nondeterministic Turing machine can also be computed by a deterministic Turing machine.

Theorem 2 *For every nondeterministic Turing machine N there is a deterministic Turing machine M which accepts the same language.*

Proof Outline:
We will create M as a 3-tape deterministic Turing machine and use the above theorem that there exists a single tape Turing machine equivalent to any multi-tape Turing machine.

For any state and current symbol of N, there are a finite number of possible moves from which N must choose. Let r be the maximum number of choices N will face at any step, and for each state and symbol, we will arbitrarily number the choices of δ from $1, \ldots, r$.

Any accepting execution of N will have a unique identifying sequence of the numbers $1, \ldots, r$ which corresponds to the choices N made in its execution. Therefore, M will generate all possible sequences of the numbers $1, \ldots, r$ in a systematic way and use these to guide a simulation of the execution of N. Since the sequence corresponding to the accepting execution of N is finite, M will eventually generate that sequence.

On the first tape of M, we will store the input to N.

On the second tape of M, M will generate all sequences of the numbers $1, \ldots, r$ in an orderly fashion - ordered shortest to longest, and sequences of equal length ordered in numerical order.

On the third tape of M, M will simulate the execution of N. For each sequence on the second tape, M will copy the input string from the first string to the third string and then simulate N on the third tape. M will use the current sequence from the second tape to direct it whenever it has a choice to make. If the simulation of N ever enters N's accepting state, then M will halt and accept the input. If the simulation of N either rejects or fails to halt by the end of the current sequence or if the current sequence contains a value which does not match a legal choice, M will move to the next sequence on the second tape and restart the simulation.

If N accepts its input, so will M because M will eventually generate the sequence which corresponds to the execution of N. Likewise, if N does not have an accepting execution, M also will not accept the input. □

Note that the above proof does not handle termination. For example, suppose for a nondeterministic Turing machine N, every possible choice of the execution leads to the state q_{reject}. Thus, N will eventually halt. However, in the above proof the deterministic Turing machine M will not halt unless the state q_{accept} is reached. Instead, the second tape will continue generating an infinite series of number sequences. Can you think of a way to modify the proof so that M is guaranteed to halt if N halts?

Although nondeterminism does not add to the computational power of a Turing machine, it is still of interest when we look at issues of efficiency. Let us calculate the running time of M. Each simulation of N by M will take at least $T_N(n)$ steps. How many times will N be simulated? There are at most r choices at each step of the execution of N so M will have to simulate N at most $r^{T_N(n)}$ times. Thus, $T_M(n) \in O(T_N(n) \times r^{T_N(n)})$.

Obviously, the proof above presented a naive implementation of a nondeterministic algorithm by a deterministic Turing machine, and we could be more clever in our implementation and reduce the running time of M. However, there is no known way to do the simulation with less than an exponential slowdown.

A better understanding of how nondeterminism affects computation may help us to understand when an efficient algorithm does and does not exist for a specific problem. It may be that some problems are theoretically solvable but will require so many computational resources so as to be practically unsolvable. We will see more of nondeterminism later!