Algorithms and Problems

The aim of complexity theory is to understand what makes certain problems difficult to solve algorithmically. When we say that a problem is difficult, we mean not that it hard to come up with an algorithm for solving a problem, but that any algorithm we can devise is inefficient, requiring inordinate amount of resources such as time and memory space. In this course, we aim at a theoretical understanding of why some problems appear to be inherently difficult.

The course builds upon **Data Structures and Algorithms**, where we saw how one measures the complexity of algorithms, by asymptotic measures of the number of steps the algorithms takes. It also builds on **Computation Theory**, where we saw a formal, mathematical model of the concept of *algorithm*, which allows us to show that some problems (such as the Halting Problem) are not solvable at all, algorithmically. We now wish to look at problems which are solvable (in the sense of computation theory) in that there is an algorithm, but they are not practically solvable because any algorithm is extremely inefficient. To start to make these notions precise, we look at a specific, familiar problem.

Sorting

Consider the statement.

> Insertion Sort runs in time $O(n^2)$.

This is shorthand for the following statement:

> If we count the number of steps performed by the Insertion Sort algorithm on an input of size n, taking the largest such number, from among all inputs of that size, then the function of n so defined is *eventually* bounded by a *constant multiple* of n^2.

More formally, we define the notation O, Ω and θ as follows:

Definition

For functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$, we say that:

- $f = O(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \leq cg(n)$;
- $f = \Omega(g)$, if there is an $n_0 \in \mathbb{N}$ and a constant c such that for all $n > n_0$, $f(n) \geq cg(n)$.
- $f = \theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

This allows us to compare algorithms. Thus, if we consider, in conjunction with the statement above about Insertion Sort, the statement

> Merge Sort is an $O(n \log n)$ algorithm.

we know that Merge Sort is an asymptotically faster algorithm than Insertion Sort. Whatever other constant factors might be involved, any implementation of the former will be faster than any implementation of the latter, for sufficiently large lists.

However, the question we are interested in is: what is the complexity of the sorting problem? That is, what is the running time complexity of the fastest possible algorithm for sorting a list? The analysis of Merge Sort tells us that this is no worse than $O(n \log n)$. In general, the complexity of a particular algorithm establishes an *upper bound* on the complexity of the problem that the algorithm solves. To establish a *lower bound* we need to show that no possible algorithm, including those as yet undreamed of, can do better. This is a much harder task, in principle, and no good general purpose methods are known.

The case of the sorting problem is an exception, in that we can actually prove a lower bound of $\Omega(n \log n)$, showing that Merge Sort is asymptotically optimal.

Lower Bound. We will now establish the lower bound on the complexity of the sorting problem. That is, we aim to show that, for any algorithm A which sorts a list of numbers, the worst case running time of A on a list of n numbers is $\Omega(n \log n)$. We make no assumptions about the algorithm A other than it sorts a list of numbers, and makes its decisions on the basis of comparisons between numbers.

Suppose the algorithm sorts a list of n numbers $a_1, \ldots, a_n$. We can assume, without loss of generality, that the numbers are all distinct. Thus, whenever two numbers a_i and a_j are compared, the outcome is exactly one of the two possibilities: $a_i < a_j$ or $a_j < a_i$. When sorting the list of numbers $a_1, \ldots, a_n$, the algorithm A must reach its first decision point, which involves the comparison of two numbers a_i and a_j. Based on the result of this comparison, the algorithm will take one of two different branches, each of which may eventually lead to another comparison, and so on. Thus, the entire computation of the algorithm A can be represented as a tree (see Figure 1).

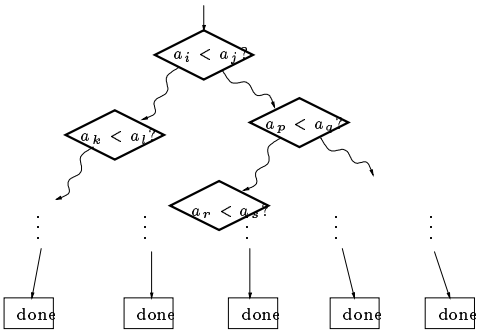

Figure 1: The computation tree for a sorting algorithm

The leaves of the tree represent a completed computation. How many leaves are there? We know that there are $n!$ different ways that the initial collection of n numbers could have been presented to the algorithm. Each of these must lead to a different branch through the computation tree. If this were not the case, we could find two different orderings of the same list, on which the algorithm performed exactly the same actions. Thus, the algorithm would incorrectly sort at least one of them. We conclude that the computation tree must have at least $n!$ leaves.

A binary tree with $n!$ leaves must have height at least $\log_2(n!)$. However, $\log(n!) = \theta(n \log n)$, establishing the result we wished to show. To see that $\log(n!) = \theta(n \log n)$, note the following inequalities:

$$\begin{aligned} \log(n!) = \log(n \cdot (n-1) \cdots 1) &= \log(n) + \log(n-1) + \cdots + \log(1) \\ &< \log(n) + \log(n) + \cdots + \log(n) = n \log n \end{aligned}$$

and

$$\begin{aligned} \log(n!) = \log(n \cdot (n-1) \cdots 1) &> \log(n) + \log(n-1) + \cdots + \log(n/2) \\ &> \log(n/2) + \log(n/2) + \cdots + \log(n/2) = (n/2) \log(n/2) \end{aligned}$$

The Travelling Salesman Problem The travelling salesman problem is defined as follows:

> Given a set V of vertices, along with a cost matrix, which is defined as a function $c : V \times V \to \mathbb{N}$, giving for each pair of vertices a positive integer cost, the problem is to find an ordering of the vertices $v_1, \ldots, v_n$ for which the total cost:
>
> $$c(v_n, v_1) + \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$
>
> is the smallest possible.

An obvious algorithm to solve the above problem is to try all possible orderings of the set of vertices, and compute the total cost for each one, finally choosing the one with the lowest cost. Since there are are $n!$ possible orderings, the running time of the algorithm is at least $n!$. There are somewhat better algorithms known, but they still have exponential time complexity.

In order to prove a lower bound on the complexity of the travelling salesman problem, we can carry out an analysis similar to the one given above for the sorting problem. That is, we can consider the computation tree of an arbitrary algorithm for solving this problem. Once again we see that there must be at least one branch in the computation tree corresponding to every possible ordering of the vertices. Thus, the computation tree must have at least $n!$ leaves and therefore the running time of the algorithm is $\Omega(n \log n)$. However, this is a far cry from the best known upper bound, which is $O(n^2 2^n)$. The gap between these two bounds sums up our state of knowledge (or lack of it) on the complexity of the travelling salesman problem. Indeed, it is emblematic of our state of knowledge of the complexity of many combinatorial problems.

Formalising Algorithms

As we have seen, one of the aims of complexity theory is to establish lower bounds on the complexity of problems, not just specific algorithms. To do this, we have to be able to say something about all algorithms for solving a particular problem. In order to say something

about all algorithms, it is useful to have a precise formal model of an algorithm. Fortunately, we have one at hand, introduced in Computation Theory for exactly this purpose. It is the Turing machine.

One important feature of the Turing machine as a formalisation of the notion of an algorithm is that it is simple. Algorithms are composed of very few possible moves or instructions. While this simplicity means it is not a formalism we would actually like to use to express algorithms, it does makes proofs about *all* algorithms easier, as there are fewer cases to consider. By equivalence results we can prove, we can then be confident that the proofs apply equally well to other models of computation we can construct.

Turing Machines

For our purposes, a Turing Machine consists of:

- K — a finite set of states;
- Σ — a finite set of symbols, disjoint from K;
- $s \in K$ — an initial state;
- $\delta : (K \times \Sigma) \to K \cup \{\texttt{acc}, \texttt{rej}\} \times \Sigma \times \{L, R, S\}$

  A transition function that specifies, for each state and symbol a next state (or accept `acc` or reject `rej`), a symbol to overwrite the current symbol, and a direction for the tape head to move (L – left, R – right, or S - stationary)

Informally, this is usually pictured as a box containing a finite state control, which can be in any of the states in K, a tape (infinite in one direction), containing a finite string of symbols from Σ, and a read/write head pointing at a position in the tape. A complete snapshot of the machine at a moment in time can be captured if we know three things—the state of the control, the contents of the tape, and the position on the tape at which the head is pointing. We choose to represent this information as the following triple:

Definition

A *configuration* is a triple (q, w, u), where $q \in K$ and $w, u \in \Sigma^\star$

The intuition is that (q, w, u) represents a machine in state q with the string wu on its tape, and the head pointing at the last symbol in w. The configuration of a machine determines its future behaviour.

Computation We think of a Turing machine as performing a computation by proceeding through a series of configurations. The transition from one configuration to the next is specified by the transition function δ. Formally,

> Given a machine $M = (K, \Sigma, s, \delta)$ we say that a configuration (q, w, u) *yields in one step* (q', w', u'), written
>
> $$(q, w, u) \to_M (q', w', u')$$
>
> if

- $w = va$;
- $\delta(q, a) = (q', b, D)$; and
- either $D = L$ and $w' = v$ $u' = bu$
  or $D = S$ and $w' = vb$ and $u' = u$
  or $D = R$ and $w' = vbc$ and $u' = x$, where $u = cx$.

The relation $\rightarrow_M^\star$ is the reflexive and transitive closure of $\rightarrow_M$. That is, for any two configurations c and c', we have $c \rightarrow_M^\star c'$ if the machine M can go from configuration c to configuration c' in 0 or more steps.

A *computation* of the machine M is a sequence of configurations $c_1, \ldots, c_n$ such that $c_i \rightarrow_M c_{i+1}$ for each i.

Each machine M defines a language $L(M) \subseteq \Sigma^\star$ which it accepts. This language is defined by:

$$L(M) = \{x \mid (s, \triangleright, x) \rightarrow_M^\star (\mathtt{acc}, w, u) \text{ for some } w \text{ and } u\}$$

Here, we use $\triangleright$ as a special symbol to denote the left end of the machine tape. This should be thought of as a symbol which cannot be overwritten by any other symbol, and such that when the head is reading $\triangleright$, it cannot move to further to the left.

So, $L(M)$ is the set of strings x such that if the machine M is started with the string x on the input tape, it will eventually reach the accepting state `acc`. Note that the strings excluded from $L(M)$ include those which cause the machine to reach the rejecting state as well as those which cause it to run forever. Sometimes it is useful to distinguish between these two cases. We will use the following definition:

> A machine M is said to *halt on input* x if the set of configurations (q, w, u) such that $(s, \triangleright, x) \rightarrow_M^\star (q, w, u)$ is finite.

We recall a few further definitions regarding languages that are accepted by some machine.

Definition

- A language $L \subseteq \Sigma^\star$ is *recursively enumerable* if it is $L(M)$ for some M.
- A language L is *decidable* if it is $L(M)$ for some machine M which *halts on every input.*
- A language L is *semi-decidable* if it is recursively enumerable but not decidable.
- A function $f : \Sigma^\star \rightarrow \Sigma^\star$ is *computable*, if there is a machine M, such that for all x, $(s, \triangleright, x) \rightarrow_M^\star (\mathtt{acc}, f(x), \varepsilon)$

where ε denotes the empty string.

An example of a language that is recursively enumerable but not decidable is the *Halting Problem* H. To be precise, we fix a representation of Turing machines as strings over the alphabet $\{0, 1\}$ (known as a Gödel numbering). If $[M]$ denotes the string representing a machine M, then Halting Problem H is defined as the language:

$$H = \{[M], x \mid M \text{ halts on input } x\}.$$

An example of a language that is not recursively enumerable is the complement of H, denoted $\bar{H}$:

$$\bar{H} = \{[M], x \mid M \text{ does not halt on input } x\}.$$

Example Consider the machine with δ given by:

	$\triangleright$	0	1	$\sqcup$
s	$s, \triangleright, R$	$s, 0, R$	$s, 1, R$	$q, \sqcup, L$
q	acc, $\triangleright, R$	$q, \sqcup, L$	rej, $\sqcup, R$	$q, \sqcup, L$

This machine will accept any string that contains only 0s before the first blank (but only after replacing them all by blanks). To see this, note that the machine, starting in state s, moves to the right, leaving all 1s and 0s unchanged, until it encounters the first blank. It then goes into state q. This causes it to move left, replacing all 0s with blanks. If it encounters a 1 on the way, it rejects the input. However, if it reaches the left end of the tape, it accepts.

Our formal treatment of Turing machines can be extended easily to machines that have multiple tapes. We only give a brief indication here of how it would be done. For instance, if we have a machine with k tapes, we would specify it by:

- K, Σ, s; and
- $\delta : (K \times \Sigma^k) \to K \cup \{a, r\} \times (\Sigma \times \{L, R, S\})^k$

That is, a transition is determined by the state and the k symbols that are under the k tape heads. Moreover, the transition determines the new state, the k symbols to be written on the tapes, and a direction for each tape head to move.

Similarly, to specify a configuration, we would need to specify the state, the contents of all k tapes, and the positions of all k tape heads. This can be captured by a $2k+1$ tuple as follows:

$$(q, w_1, u_1, \ldots, w_k, u_k).$$

It is known that any language that is accepted by a k tape Turing machine is also accepted by a one tape Turing machine.

Complexity

For any function $f : \mathbb{N} \to \mathbb{N}$ we say that a language L is in $\mathsf{TIME}(f(n))$ if there is a machine (possibly with multiple tapes) $M = (K, \Sigma, s, \delta)$, such that:

- $L = L(M)$; and
- for each $x \in L$ with n symbols, there is a computation of M, of length at most $f(n)$ starting with $(s, \triangleright, x, \triangleright, \varepsilon, \ldots, \triangleright, \varepsilon)$ (that is to say, in the start state, with x on the first tape, and the empty string on all other tapes) and ending in an accepting configuration.

In short, $\mathsf{TIME}(f(n))$ is the set of all languages accepted by some machine with running time $f(n)$.

Similarly, we define $\mathsf{SPACE}(f(n))$ to be the languages accepted by a machine which uses at most $f(n)$ tape cells on inputs of length n. In defining space complexity, we assume a machine M, which has a read-only input tape, and separate work tapes. We only count cells on the work tapes towards the complexity.

In general, not only can a single tape Turing machine simulate any other model of computation (multi-tape Turing machines, Random Access Machines, Java programs, the lambda calculus, etc.), but it can do so efficiently. That is, the simulation can be carried out by a Turing machine with only a polynomial factor increase in time and space complexity. This leads one to what is sometimes called the strong form of the Church-Turing thesis:

> Any two reasonable models of computation are polynomially equivalent, that is each can simulate the other within a polynomial factor of complexity.

There are, however, unreasonable models of computation, where it is not clear that a polynomial time simulation is possible. One such, for which we would dearly like to know whether or not it can be simulated with a polynomial time factor is the *nondeterministic* Turing machine.

Nondeterminism If, in the definition of a Turing machine, we relax the condition on δ being a function and instead allow an arbitrary relation, we obtain a *nondeterministic Turing machine.*

$$\delta \subseteq (K \times \Sigma) \times (K \times \Sigma \times \{R, L, S\}).$$

The notion of a configuration is unchanged. The state q, and the contents of the tape wu, with the position of the head given by the last symbol in w still give a snapshot of the machine. However, it is no longer the case that the configuration completely determines the future behaviour of the machine. More precisely, the yields relation $\rightarrow_M$ between configurations is no longer functional. For a given configuration (q, w, u), there may be more than one configuration (q', w', u') such that $(q, w, u) \rightarrow_M (q', w', u')$.

We still define the language accepted by M by:

$$L(M) = \{x \mid (s, \triangleright, x) \rightarrow_M^\star (\mathtt{acc}, w, u) \text{ for some } w \text{ and } u.\}$$

That is, it is the set of strings x for which there is some computation going from the starting configuration to an accepting configuration. It may, however, be the case that for some x in the language, there are other computations that lead to rejection, or that do not halt at all.

The computation of a nondeterministic machine, starting on a string x can be pictured as a tree of successive configurations.

We say that the machine accepts the string x if there is any path in the tree that leads to an accepting configuration. Conversely, the machine does not accept x if all paths either lead to a rejecting state or are infinite.

A deterministic Turing machine can simulate a nondeterministic one, essentially by carrying out a breadth-first search of the computation tree, until an accepting configuration is found. Thus, for any nondeterministic machine M, there is a deterministic machine which

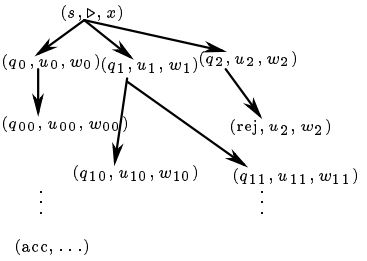

Figure 2: Nondeterministic computation

accepts exactly the language $L(M)$. However, it is not clear that this simulation can be carried out in polynomial time. To say that a nondeterministic machine runs in polynomial time is to say that the height of the computation tree on input string x is bounded by a polynomial $p(|x|)$, in $|x|$—the length of x. However, the time required by a deterministic algorithm to carry out a breadth first search of a tree of height $p(|x|)$ is $O(2^{p(|x|)})$.

Decidability and Complexity It is fairly straightforward to note that every decidable language has a time complexity in the sense that, if L is decidable, there is a *computable* function f such that $L \in \mathsf{TIME}(f(n))$. To see this, take a machine M such that $L = L(M)$ and such that M halts on all inputs, and take f to be the function that maps n to the maximum number of steps taken by M on any string $x \in L$ of length n. To see that f is computable, note that we can construct an algorithm which given an input number n, simulates M on all possible strings of length n. Since we know that M halts on all inputs, this process eventually terminates, and we can calculate the maximum number of steps taken by M on any of the inputs.

We could well ask if this might be true for a semi-decidable language. That is, if L is semi-decidable, it is recursively enumerable but not decidable. So, there is a Turing machine M such that $L = L(M)$, though M might not halt on inputs not in L. So, there is still a well-defined function f which maps n to the maximum over all strings x in L of length n of the number of steps taken by M to accept x. However, we can show that f cannot be a computable function. We can say more: there is no computable function g such that $f = O(g)$. To see why this is the case, suppose there were a computable function f such that for every $x \in L$ of length n, M accepts x in at most $f(n)$ steps. We can then construct a machine M' that accepts L and always halts, as follows. M', on input x, takes the length n of x and computes $f(n)$. M' then simulates M on input x for $f(n)$ steps, counting them along the way. If the simulation results in acceptance, M' accepts the input. However, if the simulation results in rejection, *or* the computation has not been completed in $f(n)$ steps, x is rejected. Thus, M' halts on all inputs, and accepts exactly the strings that M accepts, which is a contradiction, since we assumed that L is not decidable.

In other words, we have just shown that, for any semi-decidable language L, the running time of a machine M accepting L cannot be bounded above by any computable function.