

1 Time and Space on Turing Machines

As discussed in the last lecture, the Turing machine is a convenient model to study partly because the resources of time and space have elegant formulations with respect to it. These resources correspond respectively to running time and memory usage of computer programs.

A Turing machine M takes time T on input x if the computation of M on input x has length $T + 1$. Recall that the computation of M is the sequence of configurations $C_0, C_1 \dots C_T$ such that for each $i, 1 \leq i \leq T$, C_i follows from C_{i-1} by one application of the transition function.

Given a function $T : \mathbb{N} \rightarrow \mathbb{N}$, a Turing machine M is said to run in time T if for each n , M takes time at most $T(n)$ on any input x of length n . Thus T is an upper bound on the *worst-case* time complexity of M - it is at least the maximum over all inputs x of length n of the time taken by M on x . This is not the only reasonable measure of time complexity - we could also consider, say, the *average-case* time complexity, which is some average of times taken on individual inputs. The study of worst-case complexity corresponds to the desire to solve a problem correctly and efficiently on *all* inputs. Worst-case complexity has the advantage of being a very robust notion - it does not vary significantly with the computational model on which it is defined.

A Turing machine M takes space S on input x if the number of different tape cells accessed during the computation of M , excluding cells of the input tape, is S . Given a function $S : \mathbb{N} \rightarrow \mathbb{N}$, a Turing machine M is said to run in space S if it takes space at most $S(n)$ on any input of length n .

What kinds of time and space bounds are we interested in? A non-trivial time bound is at least linear, since a Turing machine needs linear time just to read its input (At most one new input bit can be read in each computation step). This is not a constraint for space, since cells on the input tape are not counted when measuring space. Even constant space bounds make sense, and in fact the class of languages decidable using zero space is exactly the regular languages. However, we will mainly be interested in space bounds that are at least $\log(n)$, since $\log(n)$ space is required just to maintain a pointer to the input on a read/write tape. Maintaining a pointer to the input is a minimum requirement for most non-trivial computational tasks.

Now we are ready to define time and space bounded complexity classes. $\text{DTIME}(T(n))$ is the class of languages decided by Turing machines running in time $O(T(n))$. $\text{DSpace}(S(n))$ is the class of languages decided by Turing machines running in space $O(S(n))$.

2 Analyzing Time and Space: Examples

We discuss a couple of examples to illustrate the notions in the lecture so far.

The first example is the PARITY problem. The PARITY language consists of all strings with an odd number of ones. We give an informal description of a Turing machine which solves PARITY efficiently. The TM reads its input from

left to right one bit at a time, and maintains in its state the parity of the bits read so far. Suppose the machine is in state q_1 if the parity of the bits so far is 1 (i.e., the string read so far contains an odd number of ones), and in state q_2 otherwise. Say the machine reads a new bit x_i . If x_i is 0, the machine remains in the same state as before. If x_i is 1, the machine switches its state, going to state q_2 if it was in state q_1 and going to state q_1 if it was in state q_2 . It should be clear that this preserves the invariant that the state reflects the parity of bits read so far. When the machine comes to the end of its input, it accepts if it is in state q_1 , and rejects otherwise.

Let us analyze the time and space taken by this Turing machine. The number of computation steps is just the length of the input, since the machine reads one input bit at each stage and halts when it reaches the end of the input. The space required is zero, since the machine does not require any read/write tapes. Thus the machine is highly efficient both in terms of time and space - the time taken is the minimum required for deciding a non-trivial language, and the space taken is the minimum possible.

In fact, a more general result holds: any regular language is accepted by a Turing machine taking linear time and zero space. Since PARITY is regular, the result above is a consequence. This is not surprising as regular languages are defined by finite automata, which are essentially Turing machines without read/write tapes.

Next we consider the example of the DUPLICATION problem. The language DUPLICATION consists of all strings $w?w$, where $w \in \{0,1\}^*$ and '?' is a separate symbol.

We design a Turing machine to solve DUPLICATION. The Turing machine reads the input from left to right until it encounters the '?' symbol. As it reads the input, it copies the input into one of its read/write tapes. If the Turing machine never reads a '?' symbol, it rejects. If it does read a '?' symbol, then it stops copying the input onto the read/write tape. Instead, it moves the input head of the read/write tape back to the leftmost position, and then compares the rest of the input symbol by symbol with what is written on the read/write tape. If there is an exact correspondence, the machine accepts, else it rejects.

This machine takes linear time as well, though here the time is not exactly n but rather $\lceil 3n/2 + 1 \rceil$. But in addition, the machine takes linear space, since it accesses a linear number of cells on the read/write tape in the worst case. Thus, the machine is efficient in time but not very efficient in space.

In fact, there is a different Turing machine solving this problem which takes quadratic ($O(n^2)$) time and $O(\log(n))$ space. This illustrates a common phenomenon in computational complexity: there are often *tradeoffs* between resources. Trying to make do with less of a given resource might cause a greater use to be made of a different resource. For the DUPLICATION problem, it can be shown that the product of the time and space required to solve it has to be at least quadratic.

Exercise: Construct a Turing machine solving DUPLICATION which takes time $O(n^2)$ and space $O(\log(n))$.

3 Programs as Data, and Efficient Universal Machines

A Turing machine is a finite object, and hence can be encoded as data and fed as input to another Turing machine. This phenomenon gives enormous flexibility to computation, and permits the existence of general-purpose computers, rather than insisting on a different computer for each application. General-purpose computers are formalized as a *universal Turing machine*. When given as input a pair $\langle M, x \rangle$, where M is the encoding of a Turing machine and x is an input to the machine encoded by M , a universal machine U outputs $M(x)$.

We deal with two issues in this section: first we describe a couple of properties that natural encodings of Turing machines satisfy, and then we mention how universal machines can be made time and space efficient.

We require two properties of Turing machine encodings:

1. Every string over $\{0, 1\}$ encodes *some* Turing machine. This can be easily ensured by assuming that any string that is not a “well-formed encoding” of a Turing machine encodes the TM with one state.
2. Every Turing machine is encoded by an infinite number of strings. This can be ensured by “padding” the encoding in some way, and defining the padded string to encode the same Turing machine as the original string. For example, let $E : \{0, 1\}^* \rightarrow \mathcal{M}$ be an encoding, where \mathcal{M} is the class of Turing machines. We can define a new encoding E' as follows: for all strings y , $E'(y1) = E(y)$ and $E'(y01^n) = E'(y1)$ for all y and n . This is not as artificial as it might sound - the same kind of phenomenon is present in standard programming language, where the comments feature allows one to “pad” programs without affecting their functionality.

Since we are concerned with efficient computation in this course, we would like our universal Turing machines to also be efficient. The standard construction of universal Turing machines also gives universal machines that are reasonably time-efficient and space-efficient, where “reasonably” means that there is at most a polynomial slowdown compared to the machine being simulated. We will require somewhat more optimized results.

Theorem 1 [Hennie-Stearns] *There is a universal Turing machine U such that for any machine M running in time T on input x , U runs in time at most $c_M T \log(T)$ on input $\langle M, x \rangle$ and outputs $M(x)$, where c_M is a constant depending only on the size of M 's state set, input alphabet and tape alphabet.*

The proof of the Hennie-Stearns result is given in a supplementary note on the course web page.

Theorem 2 [Stearns-Hartmanis-Lewis] *There is a universal Turing machine U' such that for any machine M running in space S on input x , U' runs in space at most $d_M S$ on input $\langle M, x \rangle$ and outputs $M(x)$, where d_M is a*

constant depending only on the size of M 's state set, input alphabet and tape alphabet.

The proof of this result is not hard - it just involves being careful with extra space usage when simulating M .

Exercise: Prove Theorem 2.

4 The Hierarchy Question

A natural question to ask about resources like time and space is: can we solve more problems given more of a resource? A result to this effect is known as a hierarchy theorem.

We can ask if in general, for time bounds t and T where the latter grows faster than the former, $\text{DTIME}(t)$ is strictly contained in $\text{DTIME}(T)$. This is false - a result due to Manuel Blum says there are arbitrarily large complexity gaps for certain peculiar complexity bounds, e.g. there is a growth rate t such that $\text{DTIME}(t) = \text{DTIME}(\log(t))$. However, the time bound t in this example is pathological - it does not occur as the time bound for any natural problem. We can pose the question in a more restricted setting, where we focus our attention on time bounds or space bounds that are "natural" in some sense.

We use the notion of *constructibility* - a resource bound is constructible if it can be computed "efficiently", where efficiency is measured with respect to that very resource bound.

A function $t : \mathbb{N} \rightarrow \mathbb{N}$ is time-constructible if there is a transducer which outputs $t(n)$ on any input of length n , within time $O(t(n))$. Most time bounds commonly encountered in algorithm design and analysis are time-constructible, eg. $n, n \log(n), n^2, n^{\log(n)}, 2^n, n^n$ etc.

Analogously, a function $s : \mathbb{N} \rightarrow \mathbb{N}$ is space-constructible if there is a transducer which outputs $s(n)$ on any input of length n , and operates within space $O(s(n))$. Most commonly encountered space bounds are space-constructible, eg., $\log(n), \log^2(n), n, n^2, 2^n$ etc.

Hierarchy theorems do hold for constructible time and space bounds. We prove this in the next lecture.