Lecture 20

Resource-Bounded Computations

In this lecture, we will investigate the resource requirements of Turing machines. So far, we've allowed our machines to use an arbitrarily large amount of time (and an arbitrarily large amount of memory tape) to complete their computations. However, in practice we care about algorithms that run *efficiently*, terminating in a reasonable amount of time, and using a reasonable amount of memory.

20.1 Defining time and space usage

We will start by defining the computation time of a Turing machine.

Definition 20.1 (Computation time on a string)**.** *Let M be a Turing machine and let w be a string. The* computation time *of M on w is the number of steps M takes before halting when run on w. If $M(w)$ does not halt, we say the computation time of M on w is infinite.*

In computer science, we usually care about how the computation time grows with the input size. That is, we don't just care about the amount of time M will take to run on a single input w, but instead on how the running time of M grows as a function of the input size. We will focus on worst-case running time, because we would like to make guarantees like "M will always run for at most $10n^2$ steps on inputs of size at most n". The "always" part of such a statement requires us to look at worst-case inputs.

Definition 20.2 (Running time of a TM)**.** *Let M be a Turing machine with input alphabet Σ. Let $T\colon \Sigma^* \to \mathbb{N} \cup \{\infty\}$ be such that $T(w)$ is the computation time of M on w for each $w \in \Sigma^*$. Define the running time of M to be the function $t\colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined by*

$$t(n) = \max\{T(w) : w \in \Sigma^*, |w| \leq n\}.$$

Note that the way we defined it, $t(n)$ is a non-decreasing function, because as n gets larger $t(n)$ takes a maximum over a larger and larger set. Also note that we set $t(n) = \infty$ if M does not halt on some string of length at most n. Typically, we will restrict to Turing machines that always halt, so the function $t(n)$ will output natural numbers (and not ∞).

We can also define the space (i.e. memory) used by a Turing machine. This definition is slightly more tedious as we need to ensure that we correctly keep track of the position of the tape head on the tape; we can then define the amount of memory used to be the number of unique cells of the tape the TM ever stepped on (excluding the initial cell).

Definition 20.3 (Space used by a TM on a string)**.** *Let M be a Turing machine and let w be a string. The* space *used by M on w is defined as follows. At each configuration that M goes through when run on w, we will keep track of the position $p \in \mathbb{N}$ of the head on the tape, relative to the starting position. That is, at the beginning, we will have $p = 0$, and after each step of M that uses a $\leftarrow$ transition, we decrease p by 1, while after each step of M that uses a $\rightarrow$ transition, we increase p by 1. Let $p_0, p_1, p_2, \ldots$ be the sequence of such positions corresponding to the configurations of M when run on w. Let $p_{\max}$ be the maximum value of p_i that occurs in this sequence (or ∞ if there is no maximum), and let $p_{\min}$ be the minimum value of p_i that occurs in this sequence (or $-\infty$ if there is no minimum). Then we define the space used by M on w as $p_{\max} - p_{\min}$.*

Definition 20.4 (Space function of a TM)**.** *Let M be a Turing machine with input alphabet Σ. Let $S : \Sigma^* \to \mathbb{N} \cup \{\infty\}$ be such that $S(w)$ is the space used by M on w for each $w \in \Sigma^*$. Define the space function of M to be the function $s\colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined by*

$$s(n) = \max\{S(w) : w \in \Sigma^*, |w| \leq n\}.$$

Note that in order for a Turing machine M to see a cell of the tape, it has to take a step. From this, it's not hard to see that the number of unique cells of the tape seen by M when run on w is at most the number of steps taken by M when run on w. From this it follows that the space function is at most the running time: $s(n) \leq t(n)$ for all $n \in \mathbb{N}$ (when both are defined with respect to the same Turing machine M).

20.2 Review of big-O notation

For any function $f\colon \mathbb{N} \to \mathbb{N}$, we will define the class $\mathsf{TIME}(f)$, which consists of all languages which can be decided in $O(f(n))$ time, and also the class $\mathsf{SPACE}(f)$, which consists of all languages which can be decided in $O(f(n))$ space.

Before we do so, let's recall some big-O notation. For two functions $f\colon \mathbb{N} \to \mathbb{N}$ and $g\colon \mathbb{N} \to \mathbb{N}$, we write $f(n) = O(g(n))$ if there are constants $a, b > 0$ such that $f(n) \leq ag(n) + b$ for all $n \in \mathbb{N}$. Note that the notation $f(n) = O(g(n))$ is somewhat misleading, especially the "=" part. What we really mean by $f(n) = O(g(n))$ is something closer to $f(n) \leq O(g(n))$. In particular, this is not symmetric; you cannot write $O(g(n)) = f(n)$.

The big-O notation lets us to ignore constant factors by saying "f grows slower than or equal to g, except possibly for constant factors". For the reverse direction, if we want to say that f grows at least as fast as g, we would write $f(n) = \Omega(g(n))$. In essence, when you see "$= O(\cdot)$" you should think "$\leq$", and when you see "$= \Omega(\cdot)$" you should think "$\geq$". Formally, we can define $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.

If we have both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, we write $f(n) = \Theta(g(n))$. This makes "$\Theta(\cdot)$" correspond to "=" if we think of big-O as corresponding to "$\leq$". Finally, we can also define a strict-inequality version of big-O. This is called little-o notation. We say $f(n) = o(g(n))$ if for all $C > 0$, there is some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, we have $f(n) \leq g(n)/C$. In other words, $f(n)$ grows strictly slower than $g(n)/C$ for any constant C, no matter how large. For the reverse direction, we can write $f(n) = \omega(g(n))$ if we want to say that $f(n)$ grows strictly larger than $g(n)$ (this is equivalent to writing $g(n) = o(f(n))$). When you see "$= o(\cdot)$", you should think "$<$", and when you see "$= \omega(\cdot)$", you should think "$>$".

Big-O notation can be especially confusing for functions that are not monotonic (that is, functions that zigzag up and down) or for functions which take values of 0 or below. We will not deal with these cases; generally, you may assume that all functions take values strictly larger than 0 and that they are all (weakly) increasing unless specified otherwise.

20.3 TIME and SPACE classes

We will now define the classes $\mathsf{TIME}(f)$ and $\mathsf{SPACE}(f)$. We will assume a fixed, universal alphabet for all the languages we deal with, such as $\Sigma = \{0, 1\}$.

Definition 20.5. *Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. We define $\mathsf{TIME}(f)$ to be the class of all languages that are decidable by a Turing machine M such that its running time t satisfies $t(n) = O(f(n))$.*

We define $\mathsf{SPACE}(f)$ to be the class of all languages that are decidable by a Turing machine M such that its space function s satisfies $s(n) = O(f(n))$.

Note that since $s(n) \leq t(n)$ when both are with respect to the same Turing machine M, we know that any language which can be decided in $O(f(n))$ time can also be decided in $O(f(n))$ space (using the same Turing machine). This means that any language in $\mathsf{TIME}(f)$ is also in $\mathsf{SPACE}(f)$, so we have $\mathsf{TIME}(f) \subseteq \mathsf{SPACE}(f)$ for all $f\colon \mathbb{N} \to \mathbb{N}$.

We will often use notation like $\mathsf{TIME}(n\sqrt{n})$, where the function $n\sqrt{n}$ is not integer-valued. We could instead have written something like $\mathsf{TIME}(\lceil n\sqrt{n}\rceil)$ to get an integer-valued function, but this clutters notation for no reason (rounding up, or rounding down, only changes the function value by up to 1, which does not matter inside big-O notation and hence does not matter for the classes $\mathsf{TIME}(f)$ and $\mathsf{SPACE}(f)$).

Note that the classes $\mathsf{TIME}(f)$ and $\mathsf{SPACE}(f)$ depend on the exact definition of a Turing machine. In fact, even switching between 1-tape Turing machines and 2-tape Turing machines can change the minimum running time it takes to decide a language, and in fact this minimum running time can change by more than a constant factor: this means that the definitions of $\mathsf{TIME}(f)$ and $\mathsf{SPACE}(f)$ are not robust to even small changes in the definition of a Turing machine.

This seems bad, because the reason we were using Turing machines in the first place is because we argued that the precise model of computation was not important (all sufficiently powerful models of computation were equivalent) and therefore we were free to pick the simplest one. This was true when we were dealing with *computability*, that is, what's possible to compute or decide given unlimited time and space, but it is not necessarily true for resource-bounded computations.

Luckily, it turns out that all the conversions we've previously discussed (between 1-tape Turing machines and multi-tape Turing machines, between different sizes of the tape alphabet, and between Turing machines and assembly languages) can only affect the time and space complexity by at most a *polynomial factor*. In fact, many of these conversions lose at most a quadratic factor. That is, if you can decide a language A on a 2-tape Turing machine in $f(n)$ time, then you'll be able to decide it on a 1-tape Turing machine in at most $O(f(n)^2)$ time (simply by using the 1-tape Turing machine to simulate a 2-tape Turing machine, as we've previously seen). The same thing happens with other conversions, and also when we measure space instead of time.

This means that while $\mathsf{TIME}(f)$ and $\mathsf{SPACE}(f)$ are not robust to changes in the model of computation, they are only "slightly" non-robust; they become robust to such changes if we ignore polynomial factors (instead of just ignoring constant factors as in the big-O notation). To create robust complexity classes, which do not depend on the details of the definition of a Turing machine (e.g. 1-tape versus 2-tape), we can define polynomial and exponential time and space classes, as follows.

Definition 20.6. *Define the following complexity classes.*

$$\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathsf{TIME}(n^k)$$
$$\mathsf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{SPACE}(n^k)$$
$$\mathsf{EXP} = \bigcup_{k\in\mathbb{N}} \mathsf{TIME}(2^{n^k})$$
$$\mathsf{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{SPACE}(2^{n^k}).$$

Note that P contains all languages that can be decided in time $O(n^k)$ for some k; this is called *polynomial time*. Whether a language is decidable in polynomial time does not depend on the model of computation: it does not depend on whether we use 1-tape Turing machines or 2-tape Turing machines or assembly language or Python programs.

Similarly, the class PSPACE contains all languages that can be decided in $O(n^k)$ space for some k, and this is called *polynomial space*. This is also robust to changes in the model of computation.

The classes EXP and $\mathsf{EXPSPACE}$ correspond to exponential time and space respectively, but the way we define exponential time and space is a bit strange; the function 2^{n^2} would count as exponential time, for example. Sometimes, we want to talk about only exponential functions of the form c^n for some constant $c > 1$. Languages that can be decided in time $O(c^n)$ for some $c > 1$ constitute the class E. Note that $\mathsf{E} \subseteq \mathsf{EXP}$; the class EXP contains larger complexities, such as 2^{n^2}, but also complexities like 3^n since 3^n grows slower than 2^{n^2}. All of these classes (including E) are robust to changes in the computational model.

Clearly, we have

$$\mathsf{P} \subseteq \mathsf{E} \subseteq \mathsf{EXP}$$

and

$$\mathsf{PSPACE} \subseteq \mathsf{EXPSPACE}.$$

Also, since space classes are larger than or equal to the corresponding time classes, we have

$$\mathsf{P} \subseteq \mathsf{PSPACE}, \qquad \mathsf{EXP} \subseteq \mathsf{EXPSPACE}.$$

As it turns out, we actually also have a relationship between PSPACE and EXP.

Theorem 20.7. $\mathsf{PSPACE} \subseteq \mathsf{EXP}$.

Proof. Let A be a language in PSPACE. Then there is some $k \in \mathbb{N}$ such that $A \in \mathsf{SPACE}(n^k)$. This means there is some Turing machine M which decides A and with the property that its space function s satisfies $s(n) = O(n^k)$. In other words, there are constants $a, b > 0$ such that $s(n) \le an^k + b$ for all $n \in \mathbb{N}$. This means that for any string w, when M runs on w, its tape head only visits at most $a|w|^k + b$ different cells.

Let Γ be the tape alphabet of M, and let Q be the set of internal states of M. Then the total number of unique configurations that the run of $M(w)$ can reach is at most $|\Gamma|^{a|w|^k+b} \cdot |Q| \cdot (a|w|^k + b)$. This is because there are $|\Gamma|$ options for each cell of the tape and at most $a|w|^k + b$ cells that $M(w)$ can visit, for a total of $|\Gamma|^{a|w|^k+b}$ arrangements of the relevant part of the tape; but also, there are at most $|Q|$ internal states that M can be in, and at most $a|w|^k + b$ positions that its tape head can occupy.

Note that if the run of $M(w)$ was ever in the exact same configuration two different times, then it must enter an infinite loop; that's because the behavior of a Turing machine is completely determined by its configuration, so if configuration c yields configuration c, it will just keep reaching the same configuration again and again. Since we know that $M(w)$ halts (since M decides A), this cannot happen, so $M(w)$ never reaches the same configuration twice. Since its run reaches at most $|\Gamma|^{a|w|^k+b} \cdot |Q| \cdot (a|w|^k + b)$ total unique configurations, the run must take at most $|\Gamma|^{a|w|^k+b} \cdot |Q| \cdot (a|w|^k + b)$ steps before halting.

We now have a bound on the running time of $M(w)$. Note that a, b, k, $|Q|$, and $|\Gamma|$ are constants that are independent of $|w|$. If we observe that $|\Gamma|^{an^k+b} \cdot |Q| \cdot (an^k + b) = O(2^{n^{k+1}})$, we conclude that the running time t of M must satisfy $t(n) = O(2^{n^{k+1}})$. Hence $A \in \mathsf{TIME}(2^{n^{k+1}})$, so $A \in \mathsf{EXP}$. □

This means we now have a sequence of subsequently larger complexity classes:

$$\mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{EXPSPACE}.$$

We could continue the chain, with EXPSPACE being a subset of EXPEXP (the class of languages decidable in doubly-exponential time, like $O(2^{2^{n^3}})$).

20.4 Padding arguments

We don't know whether P is equal to PSPACE, nor whether PSPACE is equal to EXP, nor whether EXP is equal to EXPSPACE. We conjecture they are all different, so that each is strictly contained within the last, but proving this seems to be beyond our current abilities.

In the next lecture, we will see that P is not equal to EXP and that PSPACE is not equal to EXPSPACE. In other words, we can at least prove that given exponential time, it's possible to solve more problems (i.e. decide more languages) than given only polynomial time, and similarly for exponential versus polynomial space.

There is one more thing we know how to show: we can show that if $\mathsf{P} = \mathsf{PSPACE}$, then $\mathsf{EXP} = \mathsf{EXPSPACE}$. The reverse direction is not known. This proof of this theorem uses a detail in our definition of these complexity classes: we defined all of the running times in terms of the *size of the input string*. By manipulating the input string to make it artificially longer, we can get a Turing machine that seems to run in "less time" – or rather, it runs in the same amount of time, but it is less compared to the input size than before, because we increased the input size. This will be the key to the proof.

Theorem 20.8. *Suppose that* $\mathsf{P} = \mathsf{PSPACE}$. *Then* $\mathsf{EXP} = \mathsf{EXPSPACE}$.

Proof. Suppose that $\mathsf{P} = \mathsf{PSPACE}$. We already know that $\mathsf{EXP} \subseteq \mathsf{EXPSPACE}$, so all we need to do is to show that $\mathsf{EXPSPACE} \subseteq \mathsf{EXP}$. To this end, let A be a language in EXPSPACE. Then $A \in \mathsf{SPACE}(2^{n^k})$ for some $k \in \mathbb{N}$. Let M be a Turing machine that decides A and whose space function s satisfies $s(n) = O(2^{n^k})$. The problem is that M might take more than exponential *time*; we want to find a way to decide A in "merely" exponential time.

Consider the language $B = \{w0^{2^{|w|^k}} : w \in A\}$. We can decide this language using a Turing machine M_B, as follows. Given an input x, the machine M_B will start by computing the length of the string, and seeing if this length is of the form $n \cdot 2^{n^k}$ for some n; if so, it will also check that all but the first n characters are 0. This can be done in polynomial time in the input length $|x|$ (the value of k will be hard-coded into M_B). If this formatting check fails, M_B can safely reject, since the input string will not have the form $w0^{2^{|w|^k}}$ for any w.

If the input has the desired form $w0^{2^{|w|^k}}$, then M_B will simulate M on input w. This will potentially take a lot of time, but it will use only $s(|w|) = O(2^{|w|^k})$ space. In the end, M_B will accept if $M(w)$ accepts and reject otherwise.

Note that M_B decides B, and that it does so in polynomial space; this is because the first part of the computation, when we verify if the input x has the right format, can be done in polynomial time in $|x|$ (and hence also in polynomial space), and the second part runs $M(w)$, which takes exponential space but is run on a string w which is much, much smaller than the input size $|x|$ to M_B. Hence, in terms of $|x|$, everything took only polynomial space (in fact, we can actually get this to be linear space, so $B \in \mathsf{SPACE}(n)$).

It follows that $B \in \mathsf{PSPACE}$. Since we are assuming that $\mathsf{P} = \mathsf{PSPACE}$, it also follows that $B \in \mathsf{P}$. This means that there is some Turing machine M'_B that decides B in polynomial time, not just in polynomial space.

Finally, we will use M'_B to construct a machine M' that decides A in exponential time. Given input w, the machine M' will start by padding w with a 1 followed by a lot of 0s, to get the string $w0^{2^{|w|^k}}$ (the value of k will be hard-coded into M'). This will take at most exponential time in $|w|$ to construct. Then, M' will simulate M'_B on the resulting string $w0^{2^{|w|^k}}$. This will take polynomial time in the size of $w0^{2^{|w|^k}}$, which is exponential time in $|w|$. Finally, M' will accept if M'_B accepts, and M' will reject if M'_B rejects.

Since M'_B decides B, it is not hard to see that M' must decide A. Moreover, M' runs in only exponential time. Hence $A \in \mathsf{EXP}$, as desired. $\square$

20.5 A note on the Church-Turing thesis

In the discussion of Turing machines and other models of computation, we mentioned the "Church-Turing thesis". This was the conclusion that all (sufficiently rich) models of computation are equivalent. It is named after mathematicians Alonzo Church and Alan Turing, who independently defined different models of computation, and then realized their models are equivalent. Note this is a "thesis," not a theorem, meaning it is not something formally proven, but just an informal conclusion.

The Church-Turing thesis deals with equivalence between computational models in terms of what is computable. But are all computational models equivalent in terms of what is *efficiently* computable? In other words, do we get the same class P when we define it in terms of different models of computation? The general answer seems to be "yes", at least for deterministic models of computation, and this is referred to as the "Extended Church-Turing thesis". This was not actually formulated by Church and Turing, but only much later.

Note that there are some more significant challenges to the extended Church-Turing thesis than to the original. In particular, it seems possible that what's efficiently-computable in real life – that is, the model allowed by the laws of physics – might be a class of languages which is larger than P. The main challenger is the notion of quantum computation, which seems to be allowed by the laws of physics as we currently understand them. A quantum computer can efficiently decide some languages that do not seem to be in P (though we cannot prove that they are not in P). The class of languages decidable by a polynomial-time quantum computer is called BQP. We don't know for sure whether P is equal to BQP, but we suspect they are not equal, because BQP contains problems for which we do not know any polynomial-time classical algorithms. In that sense, the extended Church-Turing thesis might be wrong.

Still, the class P is interesting regardless, because it captures what you can do on deterministic, classical devices, such as your laptop.

20.6 A note on languages versus other tasks

So far, we've been talking about efficient computation only in the context of deciding languages. You might wonder: what if we want to perform some other computation, like finding the shortest path in a graph, or factoring a number? These computational tasks don't sound like they have anything to do with deciding languages.

Well, you can think of languages as specifying a "yes/no" task: a task in which you are given a string as input, and must answer yes or no. For example, given a number, determine whether it is prime. Such yes/no tasks correspond to languages, because we can just take the language of all input strings for which the answer is "yes". If we do so, we'll get a language A with the property that deciding A is the same as solving the yes/no problem in question (in the case of testing if a number is prime, we'll let $A = \{p \in \mathbb{N} : p \text{ is prime}\}$, and deciding A becomes equivalent to determining whether a given number is prime).

OK, but you might wonder: what about tasks that are not yes/no questions? For example, what about finding the shortest path in a graph? Or, perhaps a simpler example, what about finding the prime factorization of a number?

It turns out that many such problems are secretly equivalent to yes/no tasks, at least if we do not care about polynomial factors in the space or time complexity (and in this course, we generally don't care about such factors). For example, consider the task of finding the prime factorization of a given number n. Now consider the language

$$A = \{\langle n, t\rangle : n, t \in \mathbb{N}, n \text{ has a prime factor that is at most t}\}.$$

We will prove the following theorem.

Theorem 20.9. *If we had a polynomial-time algorithm for factoring a number n, then we could decide A in polynomial time, and if we could decide A in polynomial time, then we would get a polynomial-time algorithm for factoing a number n.*

(Note that by a polynomial-time algorithm for factoring n, we mean polynomial time in the length of the input string $\langle n\rangle$, whose length is actually $O(\log n)$; so we are really talking about running time of $O((\log n)^k)$ for a constant $k \in \mathbb{N}$.)

Proof. If we had a polynomial-time algorithm for factoring a number n, then we could clearly decide A: given (n, t), we would factor n, find its smallest prime factor, and check if it is at most t.

For the reverse direction, suppose we could decide A in polynomial time using some machine M_A; we construct a machine M that factors a given number n in polynomial time. The way M will work is as follows: on input n, it will repeatedly call $M_A(\langle n, t\rangle)$ for different values of t. In fact, it will use binary search to find the smallest value of t for which $M_A(\langle n, t\rangle)$ accepts. This smallest value must be the smallest prime factor of n itself. Then M will divide n by t, get a result n_2, and repeat this procedure to find another prime factor of n_2, and so on until all prime factors are found.

Note that the binary search on t only uses $O(\log n)$ calls to M_A, and each call runs in polynomial time in the input size $|\langle n, t\rangle|$, which is at most $O((\log n)^k)$; hence M finds one factor of n in polynomial time. Dividing n by a prime factor can be done in polynomial time using long division. Finally, this process needs to repeat once for each factor of n, but each factor is at least 2, so there can be at most $\log_2 n$ of them. This means the whole process of finding the smallest prime factor repeats only $O(\log n)$ times, so the total running time is polynomial in $\log n$ (and hence in the input size $|\langle n\rangle|$). □

By the way, it is not known whether there is a polynomial-time algorithm for factoring, and therefore it is also not known whether the language A above is decidable in polynomial time. That is to say, we do not know whether A is in P. (Interestingly, we do know a quantum algorithm for factoring a number, which implies that $A \in \mathsf{BQP}$.)

The point of the above theorem was to show that a task that does not look anything like a yes/no question, such as factoring, might still be secretly equivalent to a yes/no question, at least if we do not care about polynomial factors (in this case, factoring is equivalent to the question of whether a string is in the language A). Of course, this was only the special case of factoring. But it turns out that many other computational tasks can also be converted to languages, in a manner similar to how factoring a number was equivalent (up to polynomial factors) to deciding a certain language. There are exceptions to this, but they are not too common. So for this course, we will focus on the study of languages, with the idea that if we understood the complexity of languages really well, it would also let us understand the complexity of other computational tasks, even if they are not yes/no questions.