ECS 120 Lesson 22 – Time Complexity

Oliver Kreylos

Monday, May 21th, 2001

Until now, we investigated the *computability* of problems – the question whether a certain problem can be solved by some algorithm. We defined a problem to be decidable if and only if there exists a Turing Machine that halts on every input word and that accepts exactly those words that are instances of the problem. In doing so, we neglected the fact that, even if a problem is decidable, it might take a Turing Machine a very long time to halt on some input words. Some decidable problems require any Turing Machine deciding them to take so many steps that solving them is impractical for but the most simple instances. From now on, we will focus our attention on decidable problems and analyze the running-time of Turing Machines solving them – that is, the number of computation steps a Turing Machine has to perform to accept or reject an input word of some length. Our goal is to be able to estimate how long a computation will take before actually performing it. This means, we need tools to estimate running-time from the description of an algorithm alone.

1 Running-time Analysis of Algorithms

As a running example, let us consider the language $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$. We have seen that this language is context-free, so it must be decidable. A Turing Machine deciding this language is given by the following algorithm:

Algorithm $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$ On input w,

1. Mark the beginning of the tape.
2. Scan until the end of input and reject if any zero appears after any one, or if any other characters but zeros and ones appear in the input word.

3. Return the tape head to the leftmost position.

4. Scan to the right until the first non-crossed-out character is encountered. If it is a blank, accept. If it is a one, reject.

5. Cross out the current character (a zero).

6. Scan to the right until the first character that is neither crossed out nor a zero is encountered. If it is a blank, reject.

7. Cross out the current character (a one).

8. Repeat from step 3.

If given an algorithm like the one above, we are now interested in how long a Turing Machine implementing this algorithm would run, or more precisely, how many computation steps it would perform before halting. It is apparent that the number of steps a Turing Machine performs depends on the specific input word. Therefore, every deciding Turing Machine M defines a *running-time* function $\mathrm{rt}_M\colon \Sigma^* \to \mathbf{N}$, that returns $\mathrm{rt}_M(w)$, the number of steps M performs on input word w.

1.1 Input Size

The running-time function rt_M as defined above is too difficult to compute for practical purposes – in general, the only way to evaluate it for a word w is to actually run machine M on word w. This defeats the purpose of running-time analysis: We want to be able to estimate how long a computation will take before actually starting it. The first simplification is to not evaluate the running-time for a specific input word, but for a class of input words of the same *size*. The exact definition of size depends on the particular problem:

- When sorting a set, the input size would be the number of elements in the set.
- When adding two numbers, the input size would be the total number of digits needed to write down both numbers.
- When converting an NFA to a DFA, the input size would be the number of states of the NFA.

To generalize from specific problem settings, input size in running-time analysis is usually defined by simply taking the length of the input word. Since the input word for a Turing Machine solving a problem is an encoding of a problem entity, the input word length is typically proportional to any reasonable definition of input size, given that a reasonable encoding is used:

- The length of an encoding of a set is proportional to the number of elements in the set.
- If a number is encoded in some number system, e. g., binary or decimal, the number of characters needed is exactly the number of digits.
- In our standard encoding for finite state machines, the length of the encoding is proportional to the number of states.

1.2 Types of Analysis

Using above definition of input size, the running-time of a Turing Machine M is determined by the function $t_M\colon \mathbf{N} \to \mathbf{N}$ that returns the number of computation steps performed for a word w of length $|w| = n$. There is a problem with the definition of function t_M: Even when two input words w_1 and w_2 have the same length n, machine M might perform a different number of steps on w_1 and w_2. The numbers of steps are given by $\mathrm{rt}_M(w_1)$ and $\mathrm{rt}_M(w_2)$, respectively. Which one of the values of $\mathrm{rt}_M(w)$ for all possible words w of length $|w| = n$ should be selected for function t_M? There are two choices that are most commonly used in running-time analysis:

- In *worst-case analysis*, the running-time for input size n is the maximum running-time for any word of length n: $t_M(n) := \max_{|w|=n} \mathrm{rt}_M(w)$. This is a "pessimistic" estimate: If $t_M(n) = N$, we know that M will take at most N steps for any word of length n.
- In *average-case analysis*, the running-time for input size n is the average running-time of all words of length n: $t_M(n) := (1/|\Sigma|^n) \sum_{|w|=n} \mathrm{rt}_M(w)$. This is a more practical estimate: It tells us what the expected running-time for a randomly chosen word of length n is, but there might be some word that makes the machine run much longer than the expected value. For this reason, and because average-case analysis is usually more complicated, we will concentrate on worst-case analysis.

Based on worst-case analysis, we define:

Definition 1 (Time Complexity of Turing Machines) *Let M be a deterministic Turing Machine that halts on all inputs. The* running-time *or* time complexity *of M is the function $t_M\colon \mathbf{N} \to \mathbf{N}$, where $t_M(n)$ is the maximum number of steps that M performs on any input word of length n. If $t_M(n)$ is the time complexity of M, we say that M* runs in time $t_M(n)$ *and that M* is a t_M time Turing Machine.

2 Big-O Notation

We defined above that in order to determine the running-time of a Turing Machine M on an input of size n using worst-case analysis, we have to count the number of steps M performs on any word of length n and to determine the maximum of those numbers. Since the behaviour of a Turing Machine depends on the exact input word, this counting can only be done on a word-by-word basis. This is prohibitive for anything but the smallest values of n, and even if it could be done, the resulting function t_M would probably be too complicated to be meaningful.

To overcome these problems, we will only estimate the number of steps M performs, and will not exactly compute the function t_M but only approximate it by another, simpler function. The mathematical basis for these approximations is called *big-O notation*.

Definition 2 (Big-O Notation) *If $g\colon \mathbf{N} \to \mathbf{R}^+$ is a function from the natural numbers to the positive real numbers, then $O(g)$ is the set of functions that are* bounded *by g: $O(g) := \{\, f\colon \mathbf{N} \to \mathbf{R}^+ \mid \exists c > 0 \exists n_0 \geq 0 \forall n \geq n_0 : f(n) \leq c \cdot g(n) \,\}$. To say that a function f is an element of $O(g)$, we usually write $f = O\big(g(n)\big)$.*

In other words, the above definition means that function g is an *asymptotic upper bound* for every function $f \in O(g)$: There exists a constant c, such that the value of $g(n)$, multiplied by c, is at least as big as $f(n)$, if only n is large enough. Asymptotic analysis does not describe the behaviour of a function for small values of n, and it only describes a function up to a constant factor, see Figure 1. Giving an example shows how big-O notation makes running-time estimations easier: Considering the function $f(n) = 4n^3 + 6n^2 - 5n + 17$, the definition shows that $f(n) = O(n^3)$. If n

is very large, the value of f is dominated by the highest power of n; the other, "lower" terms are not important. The factor 4 in the highest-power term can be neglected as well, because big-O notation is only exact up to a constant factor. The function $g(n) = n^3$ is a much more compact way of describing f, and for purposes of running-time analysis it is exact enough: It can still be used to estimate the running-time of a Turing Machine for larger inputs: If, for example, a Turing Machine M whose running-time is bounded by $O(n^3)$ runs for a certain (measured) time T on a problem of size k (where k is big), then it is a pretty good estimate to say that the measured time T' for a problem of m times the size will be $T' = m^3T$. Specifically, M will run eight times as long if the input size is doubled. These extrapolated running-times are very good approximations in practice.

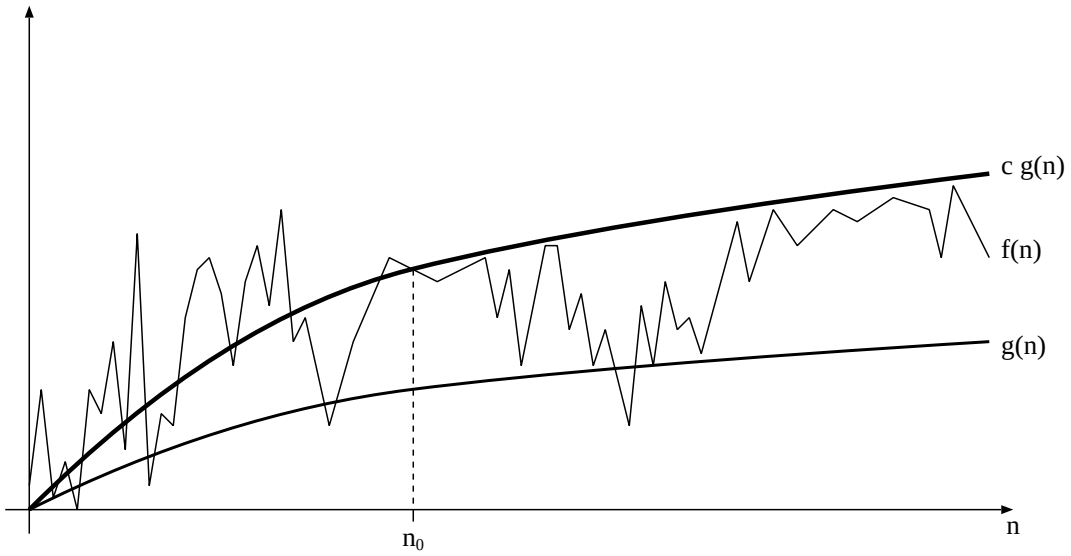

Figure 1: A function $f(n) = O\big(g(n)\big)$. The function values $f(n)$ for all $n \geq n_0$ are bounded by the function values of $c \cdot g(n)$, for some constants n_0 and c.

As examples, here are some rules about big-O notation:

1. $f(n) = O\big(f(n)\big)$ for any function f. In other words, every function is bounded by itself.

2. $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = O(n^k)$ for all $k \geq 0$ and all $a_0, a_1, \ldots, a_k \in \mathbf{R}$. In other words, every polynomial of degree k can

be bounded by the function n^k.

3. $\log_a n = O(\log_b n)$ for any bases a, b. In other words, the basis of a logarithm can be neglected in big-O notation. We typically just write $O(\log n)$ to denote a logarithm of any base.

4. $\log_b n = O(n^c)$ for any basis b and any positive exponent $c > 0$. Any logarithm function can be bound by any polynomial.

5. $n^a = O(n^b)$ for any exponents $a \leq b$. Specifically, $\sqrt{n} = O(n)$.

6. $n^k = O(a^n)$ for any exponent k and any basis $a > 0$. Any polynomial can be bound by any exponential function.

7. $a^n = b^{O(n)}$ for any bases a, b.

8. $a^n = O(n!)$ for any basis a. Any exponential function can be bound by the factorial function.

2.1 Running-Time for the Example Algorithm

We can now use big-O notation to estimate the running-time of the Turing Machine M deciding the language $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$, using the algorithm stated above. Here it is again, for convenience:

Algorithm $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$ On input w,

1. Mark the beginning of the tape.

2. Scan until the end of input and reject if any zero appears after any one, or if any other characters but zeros and ones appear in the input word.

3. Return the tape head to the leftmost position.

4. Scan to the right until the first non-crossed-out character is encountered. If it is a blank, accept. If it is a one, reject.

5. Cross out the current character (a zero).

6. Scan to the right until the first character that is neither crossed out nor a zero is encountered. If it is a blank, reject.

7. Cross out the current character (a one).

8. Repeat from step 3.

To estimate the algorithm's running-time, we look at the steps one at a time:

- Step 1 only takes a single step. We are a bit sloppy and say it takes a constant number of steps, written down as $O(1)$.
- Step 2 scans across the input. In the "worst case", the input word satisfies the requirements and the machine has to scan all the way to the first blank symbol to find out. Thus, if the length of the input word is n, this takes $O(n)$ steps.
- Step 3 moves the tape head to the left end again; this will take $O(n)$ steps.
- The number of steps performed in step 4 depend on the input word; in the worst case, all characters have been crossed out before, and the number of steps is n. We can bound all possible cases by $O(n)$.
- Step 5 takes $O(1)$ steps.
- Step 6 also takes $O(n)$ steps.
- Step 7 takes $O(1)$ steps.

We now have to consider that the steps 3 to 7 are executed in a loop. Every time the loop is executed, one zero and one one are crossed out. If the input word was in the language and n characters long, it must consist of $n/2$ zeros and ones. This means the loop, counting for a total of $O(n) + O(n) + O(1) + O(n) + O(1) = 3 \cdot O(n) + 2 \cdot O(1) = O(3n + 2) = O(n)$ steps, is executed at most $O(n/2) = O(n)$ times. Altogether, the algorithm performs $O(1) + O(n) + O(n) \cdot O(n) = O(1) + O(n) + O(n^2) = O(n^2 + n + 1) = O(n^2)$ steps.

3 Time Complexity Classes

Using our definition of worst-case asymptotic running-time, it is possible to classify all decidable languages into *complexity classes*:

Definition 3 (Complexity Class) *If* $f\colon \mathbf{N} \to \mathbf{N}$ *is a function, the* time-complexity class *of* f *is the set*

$$\mathrm{TIME}\big(f(n)\big) := \left\{ L \subset \Sigma^* \;\middle|\; L \text{ can be decided in } O\big(f(n)\big) \text{ time} \right\}.$$

In other words, $\mathrm{TIME}\big(f(n)\big)$ *is the set of all languages that can be decided by a Turing Machine whose time complexity is* $O\big(f(n)\big)$.

Our analysis of the example algorithm has shown that the language $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$ is an element of $\mathrm{TIME}(n^2)$. The next question is if there is an algorithm that can decide L in less than $O(n^2)$ steps. In fact, there is such an algorithm:

Algorithm $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$ On input w,

1. Scan the input word and reject if any zero appears to the right of a one.
2. Scan the input word and reject if the total number of ones and zeros left in the input is odd.
3. Scan across the tape and cross out every other zero, starting with the first zero, and every other one, starting with the first one.
4. If there are still zeros and ones on the tape, repeat from step 2.
5. If there are neither zeros nor ones left, accept. Otherwise, reject.

This algorithm can decide language L: In each execution of step 3, the number of zeros and ones is divided by two and the remainder is discarded. If the numbers of zeros and ones were identical before an execution, they will still be identical afterwards, and the total number of zeros and ones will be even. If, on the other hand, the numbers of zeros and ones in the input string were different, the reduced numbers will become different by exactly one after some execution of step 3. Then the total number of zeros and ones will be odd, and the machine will reject.

To analyze the running-time of the algorithm, we consider each step in turn as usual. Steps 1, 2 and 3 take $O(n)$ steps each. But, as opposed to the earlier version of the algorithm, in each iteration of the main loop the number of zeros and ones is at least cut in half. That means that the loop

can be executed at most $1 + \log_2 n$ times before no zeros or no ones are left. Altogether, the running-time is $(1 + \log_2 n) \cdot O(n) = O(\log n) \cdot O(n) = O(n \log n)$.

The found algorithm can not be further improved on single-tape Turing Machines. When switching the computation model, however, the running-time can be decreased: A two-tape Turing Machine can decide L in linear time, i. e., in time $O(n)$, by copying all zeros to the second tape, and crossing off matching numbers of zeros and ones in a single sweep of the input word using both heads.