Craig Silverstein
Gates 470
Phone: (72)3-1787
Office Hours: Wed 4:30–5:30, Fri 1–3, Gates 470
Section: Monday 2:15–3:15, 380-380W

Computer Science 154
School of Engineering
Stanford University

Section Handout #7

Time

Time complexity is a concept central to all aspects of Computer Science. Everybody wants to know how fast their program runs. But with so many different types of hardware and chip designs (to say nothing of parallel computation), what does it mean to time a program? We need a common base. That common base is turing machines.

So how long does it take a TM to run on an input? By definition, it's the number of transitions the TM makes before halting (we deal only with TMs that halt on all inputs). Of course, usually different inputs will require different amounts of time (is 3 prime? How about 26578754690563?) so we express how long an algorithm takes to run in terms of how long the input is (we'd get the most information if we compiled a table on how long our algorithm ran on *every possible* input, but that would take a bit more time than most of us have free. It turns out that figuring it out for every possible input size is often easier).

Thus, the time measure we use is not seconds but number of steps on a Turing Machine. And when we say "algorithm," all we mean is a program for a... Turing Machine! And when we say size of the input, we mean the number of tape square the input takes up on a potato. No, I'm just kidding. It's really the number of tape squares taken up on a Turing Machine. This is the great breakthrough of Turing Machines, that they give us a mathematical framework for talking about how long algorithms take.

But we can't talk about how long algorithms take in general; we talk about how long a *specific* algorithm takes using a *specific* Turing Machine (TMs are much faster if you allow an alphabet of size 100 as opposed to one of size 2. It also matters how you represent the input: if you represent it in unary, it takes up more space than if it's represented in binary. These are all things that need to be specified when talking about time complexity). Thus it doesn't make sense to ask: How long does it take to figure out if a number n is prime? Instead we ask: I have this TM program that accepts if the input is prime and rejects (by going into q_r) if it's not. How long does this program take to execute?

Now we don't care about any aspect of the input but its size; the idea is we try our TM program on all inputs of size 5, say, and figure out the longest it takes (it takes 4 steps on half the inputs, 7 on the other half, say). That is the time complexity for that size input (we would say our TM takes time 7 on inputs of size 5, though in some cases that's an overestimate). We then try to find a formula that encompasses all the sizes: if our TM takes time 1 on size 1 inputs, time 2 on size 2, time 4 on size 3, time 6 on size 4, time 8 on time 5, etc. we say the TM takes time 2n-1 on inputs of size n. Again, we are overestimating in some cases, but we can't say 2n-2 because then it's wrong for size 1.

So, for a complicated problem, we might get a complicated formula: $n^{2.5}+n\log n-4$. We throw out all but the highest power of n, since everything else is dwarfed by this huge term. Height goes approximately:

$$5^n, 4^n, \ldots, 2^n, n^5, n^4, \ldots, n^2, n^{1.5}, n\log n, n, \sqrt{n}, \log n$$

(of course, we can go higher than 5 as well). We then throw out all the constants associated with the highest power. The reason for this is a bit more subtle: given a TM that runs in time $5n^2$, say, we can construct another machine that runs in time just n^2 by encoding five squares of our TM in just one square of our new TM (using a much bigger alphabet $\Sigma \times \Sigma \times \Sigma \times \Sigma \times \Sigma$). We're left with just a simple term involving n, so we stick it inside an $O()$ to make it feel more important. TM takes time in $O(n^2)$, $O(n \log n)$, etc.

Polynomial time

In general, the way we find the function is to do a coarse analysis of what the TM has to do on a given input. For instance, a TM that accepts all inputs would take time 1, for the transition from q_0 to q_a. A TM that accepts all inputs of even length would take time $n + 1$ — n to reaad the input, and 1 to go from the space ending the input to q_a or q_r. A TM that checks to see if any two characters of the inputs were the same would have the following analysis:

For each square i of the input:

1. Look at what character, call it σ, is on square i.
2. Go to the beginning of the tape.
3. Go through each square and see if it has σ.
4. If you see two σ's, then you can go to q_a, else you go back to square $i + 1$ and continue.

It takes us 0 moves to look at the character on the square (we can combine it with a move), at most n moves to go the beginning of the tape, n moves to go through each square, and then $n - 1$ moves to get to square $i + 1$. In the worse case (where the last two characters you look at match), we could have to do this $n - 1$ times. Total time is: $(n + n + n - 1) * (n - 1) = 3n^2 - 4n + 1$. We say this algorithm is in $O(n^2)$, since n^2 is the fastest-growing term in this function. Actually, we can find a somewhat better function (it really only takes time i to go to the beginning of the tape from square i, and then time $n - 1 + 1$ to get from the end to square $i + 1$. This gives us $f(n) = 2n^2 + 2n + 1$. This is still $O(n^2)$; we don't worry about differences of constants).

Different types of TMs will give different bounds. Two-tape machines can save time because you can store numbers on the second tape instead of having to shuttle to the end of your tape and storing numbers there. Non-deterministic machines are much faster because they can guess the correct series of transitions instead of having to try every series. Differences between one-tape and two-tape machines are only polynomial: the difference between x^2 and x^4 say. But differences between DTMs and NDTMs are (we think) exponential: x vs. 2^x. This gives us the following classes of languages:

A language L is in **P** if we can find a Deterministic Turing Machine M that decides $\mathcal{L}$ that runs in polynomial time (that is, there is an f such that $f(n)$ is a polynomial and $TIME_M(w) < f(|w|)$ for all w).

(A language L is in **NP** if we can find a Non-deterministic Turing Machine M that decides $\mathcal{L}$ that runs in polynomial time.)

Hints on Homework 6

For problem 1, I like to think of each line of code as a state. Then part 1 is, "is some state not reached?" and part 2 is, "is a *specific* state not reached?"

For problem 2, remember: $O(x)$ means "runs in cx time steps **or fewer**." $\Omega(x)$ means "runs in cx time steps **or more**." $\theta(x)$ means "runs in **exactly** cx time steps." (or, more accurately, in at least $c_1 x$ and at most $c_2 x$ time steps on all inputs, where c_1 and c_2 do not depend on the input size).