Lecture 15: Time Complexity Classes

Ryan Bernstein

1 Introductory Remarks

- Assignment 4 is posted now. This assignment is *optional*. If you'd like to boost your homework grade (or if you just want the practice), you're welcome to submit it, at which point both the numerator and denominator of your homework score will be updated accordingly. If you don't feel like doing it, though, you can keep your current homework grade.
  - Assignment 4 is due next Thursday (June 2nd)
- This is week 9. We're nearing the end of the term, and we're also nearing the end of the material. We have two meetings this week and two meetings next week. It's possible that we'll reach the end of the material covered on the final exam either today or Thursday.

  I'd intended to devote next Thursday to a practice exam, as we did for the midterm. This means that there may be one or two lectures unallocated. We have a few choices for things we could do during this time.
  - We could just take the day off, as I'm sure you've got stuff going on in other classes
  - We could delve further into some interesting topics that won't be covered on the exam, including:
    - Techniques used to find approximate solutions to problems for which *optimal* solutions are prohibitively expensive to compute (we'll see examples of such problems today), including constraint relaxation and randomness
    - Deterministic context-free grammars and their use in programming languages, which is basically the only time that context-free grammars are interesting or useful at all
    - Space complexity
  - If there are any topics from any point in the term that you feel were not covered in sufficient detail (or that you're just feeling rusty on), we could review these topics

  Regardless of what you choose, I'll inform you when we reach the end of the material on which you'll be tested, and attendance at any subsequent lectures will be strictly optional.

1.1 Recapitulation

Last time, we discussed the runtime of Turing machines. We measure the runtime of a Turing machine in terms of the total number of transitions that the machine must follow. We analyze runtime as a function of the size of the input string w on which a machine operates. These functions analyze "worst-case" running

time — that is, if the runtime of a machine is $f(n)$, then $f(n)$ is the maximal number of steps used on *any* input of length n.

We also analyze running time in terms of "Big-O" asymptotic notation. If f and g are both functions from $\mathbb{N} \to \mathbb{N}$, then $f(n) = O(g(n)$ if for some integers n_0 and c, $n > n_0 \to f(n) \leq c \cdot g(n)$. Intuitively, this means that a graph of g (possibly scaled by some constant factor c will eventually overtake and surpass a graph of f.

This allows us to disregard details that we don't want to deal with, including constant factors, lower-order polynomial terms, and logarithmic bases.

We also ended with the following worksheet exercise, which we didn't have time to discuss. So here it is again.

Worksheet Exercise Consider the following two-tape Turing machine that also decides $\{0^n1^n \mid n \geq 0\}$:

$M =$ "On input w:

1. If the input is not of the form 0^*1^*, Reject
2. For each zero in the input, write a one to tape 2
3. Return tape 2's head to the left
4. For each one in the input, ensure that we see a one on tape 2
   - If either tape sees a blank before the other, Reject
   - If both tapes read blanks at the same time, Accept"

Find the time complexity of M.

2 Time Complexity Classes

We can now partition the set of Turing-decidable languages into time complexity classes. We define complexity classes as follows:

$TIME(f(n)) = \{L \mid L$ is a language decidable by an $O(f(n))$-time single-tape deterministic Turing machine$\}$

Note that this is a set of languages, not a set of Turing machines. This means that to show that a language is *outside* a complexity class, it's not sufficient to show that a Turing machine exists that runs in more than $f(n)$ steps. Rather, we'd need to show that no $O(f(n))$-time Turing machine could possibly exist to decide that language, which is a much trickier problem. As we'll see, we don't really have a great way to do this.

We'll also introduce another way of looking at time complexity, this time in terms of nondeterministic machines:

$NTIME(f(n)) = \{L \mid L$ is a language decidable by an $O(f(n))$-time single-tape nondeterministic Turing machine$\}$

2.1 Important Complexity Classes

There are a few larger complexity classes that are very noteworthy and important. Unfortunately, this is probably going to be the most confusing lecture of the term. It's not because these concepts are particularly difficult to understand, but instead because computer scientists just suck at naming things. We're going to introduce several complexity classes that are named very similarly, but mean very different things. Sorry.

The first two complexity classes we're going to discuss are called P and NP.

2.1.1 P

P is pretty self-explanatory, although since we're computer scientists, we do have a way to write it that makes it look scary:

$$P = \bigcup_{k \geq 0} (TIME(n^k))$$

What does this mean, in English? The answer is actually pretty simple: the giant U is a union operator, so this means that P is the union of all complexity classes of the form n^k for any $k \geq 0$. It would probably be clearer to write $P = \{L \mid L$ is a language decidable by an $O(n^k)$-time single-tape deterministic Turing machine for som but I didn't write this book.

The problem that we looked at on the worksheet today is an example of a problem in P. During the last lecture, we showed that the language $\{0^n 1^n \mid n \geq 0\}$ was decidable by an $O(n^2)$-time Turing machine. Since n^2 is a polynomial, we say that this problem is in P, the class of problems that can be solved by a deterministic Turing machine in polynomial time.

2.1.2 NP

NP is similar. The P once again stands for "polynomial". The N stands for "nondeterministic". Given that, you can probably guess where we're going with this, but:

$$NP = \bigcup_{k \geq 0} (NTIME(n^k))$$

In other words, NP is the class of all problems solvable by a nondeterministic Turing machine in a polynomial number of steps.

Since we can consider a deterministic machine as a nondeterministic machine that never branches, we can conclude that $P \subseteq NP$. We'll look at this relationship in greater detail later.

Example Let HAMPATH $= \{\langle G \rangle \mid G$ is a graph for which a Hamiltonian path exists$\}$. (A Hamiltonian path is a path that visits every node in the graph exactly once, without repetition.) Show that HAMPATH is in NP.

We can construct a nondeterministic Turing machine N that decides HAMPATH pretty easily. $N =$ "On input w:

1. Nondeterministically generate all $|V|!$ permutations of all $|V|$ vertices. (Note that each branch can do this in $O(|V|)$ time.) For each branch:
   (a) For each consecutive $v_i v_j$ in the generated path:
      - If E does not contain an edge from v_i to v_j, REJECT
   (b) ACCEPT"

3 Solvers and Verifiers

There's one issue with our definition of NP, which we touched on during the last lecture. The problem is that nondeterministic machines aren't really a realistic model of computation. Even if we create machines that run in parallel, we have some fixed-number of cores. My desktop can run four threads in parallel — eight if you count hyperthreading as parallelism — but when we build nondeterministic machines, we're counting on the ability to execute an unlimited number of branches simultaneously.

Consider the problem of cracking encryption keys that we discussed last time. We said that if I were to try all possible 2048-bit RSA encryption keys in sequence, a computer that could try 10,000 keys per second would take 10^{606} years to exhaust the keyspace. A nondeterministic machine, on the other hand, could simply *try all of the keys at once*, solving the problem near-instantly.

It makes sense, then, to define NP in terms of deterministic machines, which more closely model the things of which computers are actually capable. To find a way of doing this, we can again consider the problem of encryption. While it may take 10^{606} years to try every key in the keyspace, it certainly does not take this long to actually *use* one of these keys.

We can redefine NP as follows:

$$NP = \{L \mid L \text{ is a problem that can be verified in polynomial time by a deterministic Turing machine}\}$$

Verifiers are similar to solvers, but they take an additional input called a *certificate*. This certificate is a candidate solution. In the HAMPATH example we discussed before, we're trying to decide whether or not a Hamiltonian path exists in the graph G. A verifier for HAMPATH would then take a certificate that was a permutation of all $|V|$ nodes. If this permutation was a Hamiltonian path in G, the verifier ACCEPTS; otherwise, it REJECTS.

Example Construct a poly-time deterministic verifier to show that HAMPATH $\in NP$.

c is a permutation of all of the vertices in $|V|$.

$V =$ "On input $\langle G, c\rangle$:

1. For each consecutive $v_i v_j$ in the generated path: $O(|V|)$
   - If E does not contain an edge from v_i to v_j, REJECT $O(|E|)$
2. ACCEPT" $O(1)$

Example Let KNAPSACK = $\{\langle V, W, x, k\rangle \mid V$ is a set of item values, W is a set of corresponding item weights, and some combination of items has a value of at least x with a weight that does not exceed $k\}$. Construct a poly-time deterministic verifier to show that KNAPSACK $\in NP$.

c is a set of item indices. We'll let N be the number of items ($|V| = |W| = N$).

$V =$ "On input $\langle V, W, x, k, c\rangle$:

1. For each $i \in c$: $O(N)$
   (a) Add W_i to a weight accumulator w $O(n)$
   (b) If $w > k$, REJECT $O(n)$
   (c) Add V_i to a value accumulator v $O(n)$
2. If $v \geq x$, ACCEPT" $O(n)$

Note that since we've added accumulators, we mark updates to these accumulators as $O(n)$ since we have no random access and are probably storing them after the input. We could also simply use a multi-tape Turing machine and write these on another tape. Since every $f(n)$-time multi-tape machine can be emulated by a $O(f^2(n))$-time single-tape machine, this doesn't change the fact that the machine runs in polynomial time.

Worksheet Example Let SUBSETSUM = $\{\langle S, v\rangle \mid S$ is a set of integers with a subset that sums to $v\}$. Construct a deterministic verifier that shows that SUBSETSUM $\in NP$.

4 Poly-Time Reducibility

We've now introduced a difficulty spectrum, with problems in NP being conceptually "harder" than problems in P. We can determine the relative positions of problems on this spectrum using reductions.

We say that a decision problem A is *poly-time reducible* to a decision problem B — written $A \leq_p B$ — if there exists a computable function F such that:

1. $\forall s(s \in A \rightarrow F(s) \in B)$
2. $\forall s(s \notin A \rightarrow F(s) \notin B)$
3. F runs in polynomial time.

Informally, we'd introduced mapping reductions as meaning that a solution to B could be used to solve A. Now, we take this one step further and place bounds on the runtime of a solver for A.

Theorem If $A \leq_p B$ and $B \in P$, then $A \in P$.

Assume that $B \in P$. Then there exists a machine M_B that decides B in polynomial time. If $A \leq_p B$, then we can convert a candidate solution for A into a candidate solution for B in polynomial time. We can therefore construct a poly-time solver for A as follows:

$M =$ "On input w:

1. Compute $F(w)$

2. Run M_B on $F(w)$.
    - If M_B accepts $F(w)$, ACCEPT w
    - If M_B rejects $F(w)$, REJECT w"

We also introduced mapping reductions as meaning that A was *not harder* than B. We can consider poly-time reductions the same way, although our concept of "difficulty" is now time-sensitive.

5 NP-Hardness

We can now use the idea of poly-time reducibility to relate problems to each other, and this allows us to introduce a more meaningful class of problems, called NP-hard. Because computer scientists suck at naming things, NP-hard is not actually part of NP. Rather, NP-hard is the set of all problems to which *all* problems in NP are poly-time reducible.

6 P vs. NP

We've been speaking extremely vaguely about the differences between these problems and problem classes. All that poly-time reductions allow us to say is that one problem is not harder than another. However, it's very difficult to place any given problem on our difficulty spectrum. It's possible that poly-time solutions to problems that we place in NP or call NP-hard exist, but have yet to be found.

We can actually extend this to our notion of difficulty classes. We say that every problem in NP is poly-time reducible to any given NP-hard problem. This means that if a poly-time solution was found for any NP-hard problem, *every problem in NP would be solvable in polynomial time.* Since we say that P is the class of all problems solvable in polynomial time, this would mean that P and NP were the same.

Why is this significant? We have another, more informal definition for P. If a problem is solvable in polynomial time, we say that these problems are solvable efficiently, or in a reasonable amount of time. We generally think of something like bubble sort — with it's $O(n^2)$ running-time — as being inefficient. But this inefficiency pales in comparison to deterministically solving a problem in NP. Again, this is the difference between the "do I have time to make a cup of coffee before this finishes?" kind of inefficiency and the "will life on Earth still exist when this finishes" type of inefficiency.

Some of these problems in NP are also very important. This includes prime number factorization, which could be used to break some encryption schemes. There are also problems such as optimal protein folding, which could probably do some cool medical stuff that I am in no way qualified to talk about. Fast solutions to these problems have the potential to be world-changing.

Of course, this idea of "efficiency" is somewhat overblown. Polynomial or even constant-time solutions still have the potential to take unreasonably long to solve. If we could guarantee that RSA-key lenghts would *never* grow beyond 2048 bits, for example, a brute-force attack on RSA keys technically runs in constant-time. But faster solutions to these problems *could* have the potential to significantly advance technology.

The relationship between P and NP is currently unknown, which means that we've actually reached an open problem at the very edge of computer science. So that's cool.