20190115, 2019016, 20190118

Before we can talk intelligently about different algorithmic paradigms, we need to talk about different classes of problems.

When computers first became widely available, people all over the world started to create algorithms to solve computational problems. It was soon recognized that the problems they were working on fell into two loose categories: ones that were **easy** to solve with fast algorithms, and ones for which it seemed to be **impossible** to find any efficient algorithm at all.

Researchers looked for some way to formalize these concepts. The concept of computational complexity was applied, and it turned out that the "easy" problems can all be solved in polynomial time – ie they have algorithms that run in O(n^k) time for some constant k. None of the hard problems have polynomial time algorithms at all, even for very large values of k.

The study of the difficulty of problems experienced an enormous breakthrough in the early 1970's. **Cook** and **Levin**, working independently and virtually simultaneously, proved a theorem that revolutionized our understanding of the relationship between easy problems and hard problems, and also provided us with an immensely powerful tool for identifying computationally hard problems. But we need to work up to that ...

Definition: a problem X is a **decision problem** if

- the answer to any instance of X is either Yes or No
- the answer to any instance of X is completely determined by the details of the instance

For example, let X = "Does the set S contain the value 3?" where S is a well-

defined set such as {1, 10, 7, 3, 8}. X is a decision problem.

Another example: let X = "Does program P enter an infinite loop when the input is I?". This is the well-known Halting Problem, and it is a decision problem (even though we know there is no algorithm that can solve it for all possible programs and inputs).

Another example: let X = "Will this coin land Heads-up the next time it is tossed after this question is asked?". Here the answer is either Yes or No, but the answer is **not** determined by the details of the instance - tossing the coin is a random event and its outcome cannot be predicted with 100% accuracy (unless the coin is the same on both sides, or unless the universe is completely deterministic). In this case, X is not a decision problem.

Definition: The class **P** is the set of all decision problems that can be solved (i.e. the complete details of the solution, if there is one, can be found) using a "real" computer in O(n^k) time, where k is constant for each problem.

For example, the problem "Given a graph on n vertices, are there three vertices that are all mutually adjacent?" can be solved by an algorithm that examines each of the $\binom{n}{3}$ possible solutions. Since such an algorithm clearly runs in O(n^3) time, this problem is in the class **P**.

Definition: The class **NP**: This class is a bit more complex than **P**. First, we distinguish between **solving** a problem - actually finding the answer - and **verifying** a solution - checking the details to make sure they are correct. For example, if the problem is "Given a set of integers, is the number 17 in the set?" the answer might be "Yes, it is in position 5 in the set". To verify this we would check the appropriate value to see if it really is 17.

Second, we imagine a type of computer, called a **Non-Deterministic Turing Machine** (this is where the **N** in **NP** comes from) which has the magic ability to guess the right answer to any decision problem, and which will provide us with the details if the answer is **Yes**. Once the NDTM has performed this magic trick, it relapses back into being a normal computer.

Now we can define **NP**. **NP** is the set of all decision problems that can be solved by a NDTM which **verifies all Yes** answers in O(n^k) time, where k is constant for each problem.

In other words, **NP** is the set of all decision problems for which if someone tells you the answer, if the answer is No you don't have to do anything, but if the answer is Yes and they give you the details of the answer, you can verify the correctness of that answer in polynomial time.

The first response when one hears about the class **NP** is often "What is the point of talking about problems that are solved on imaginary magical computers?"

We'll answer that question in two steps. First, it is important to see that **P** is contained in **NP**. If we have an algorithm that correctly solves a problem and provides details of the solution, then we could use that same algorithm to verify the correctness of the Yes/No solution "guessed" by an NDTM.

Second, it fairly quickly became apparent that most of those “really hard” problems I have alluded to – the ones that nobody could find good algorithms for – actually do have the property that “Yes” answers can be verified quite easily. So the NDTM is a useful *conceptual* type of computer for discussing the solution of these problems.

Remember that the goal of this line of research was to find a practical way to distinguish between easy and hard problems. So far all we have is an imaginary magical computer – not very useful. But stay with me – this all leads to a very

practical and essential tool in modern computer science.

The question to be addressed is, are there any problems in **NP** that are not in **P**? In other words, are there any problems that can be verified using an NDTM that cannot be solved using a "real" computer? It would seem that there absolutely must be some - after all, the NDTM can magically guess right answers to everything, and all it has to do is verify the Yes answers.

If there are no such problems, then the two classes **P** and **NP** are equal – which virtually nobody believes – because if **P = NP** then real computers are just as powerful as magic computers.

To address the problem "Does **P = NP**?" people started trying to find the most difficult problems in **NP** - these are surely the best candidates for problems that in **NP** but not in **P** - and by the same token, if we can prove that the most difficult problems in **NP** are also in **P**, then **P = NP**.

But how can we identify the hardest problems in **NP**? Our measure of problem difficulty so far has been the computational complexity of the best algorithm for solving the problem ... but for the problems we are interested in, we have **no idea** what the best algorithm is. We are in the uncomfortable position of trying to compare the difficulty of problems that we don't know how to solve.

It turns out we can do this in a clever way. We imagine two problems X and Y. We would like to show that X is "easier" than Y (or, more precisely, that X is "not harder" than Y). We do this indirectly by showing that **IF** we could find an efficient algorithm for Y, this would immediately give us an efficient algorithm for X.

To show this relationship between X and Y, we demonstrate that any instance of X (i.e. any specific set of values or objects that X applies to) can be transformed **in polynomial time** into an instance of Y (i.e. a specific set of values or objects that Y applies to) in an **answer-preserving** way. That is, if the answer to X on that specific set of values is **YES**, then the answer to Y on the transformed set of values is also **YES** (and similarly for **NO**).

Here's a simple example:

X: Given a set S of n integers, does S contain the value 4?

Y: Given a set S of n integers, and a specific integer k, does S contain the value k?

Obviously, both of these problems are so simple that we can immediately see good algorithms for solving them. Ignore that for the moment - we are focusing on the relationship between the problems.

Suppose we are given an instance of X (a specific set of integers), and suppose we have no idea how to solve it. We ask "If we knew how to solve Y, how could we use that knowledge to solve X?"

Well, if we know how to solve Y for any set S and any integer k, we can create an algorithm **solve_Y(S,k)** for this problem. Then we could create an algorithm for X like this:

```
def solve_X(S):
    return solve_Y(S,4)
```

This algorithm for problem X adds virtually nothing to the time requirement (putting it formally, the instance of X is transformed into an instance of Y in constant time, simply by assigning k the value 4), and it is answer-preserving: the call to solve_Y() returns "Yes" iff the correct answer to the instance of X is "Yes".

Thus we can say that **if** we could solve Y efficiently, **then** we could also solve X efficiently.

It is also possible to turn this around: if we could solve X efficiently, we could also solve Y. That is, any instance of Y (search for k) can be easily transformed into an instance of X (search for 4). In class we discussed methods for doing this – it's a beneficial exercise which I recommend for anyone who was not in class when we discussed this.

Here's another, slightly more complex example:

X: Given a set of n integers, are there more positive than negative integers in the set? (NB: X is a decision problem, and it is in NP)

Y: Given a set of n integers, is the sum of the set positive? (Y is also a decision problem, and Y is also in NP)

(Once again, it is obvious that we can solve both of these problems easily. I have chosen simple problems for these examples so that we can focus on the transformation process.)

I posed this as an exercise in class – if you have not yet done so, spend some time thinking about this before you read on ...

Solving X requires counting, but solving Y involves adding ... the key insight is that counting is equivalent to adding 1's. So if we transform the instance of X into an instance of Y by replacing every positive integer by 1, and every negative integer by -1, then solving Y on the transformed set will give us a YES answer if and only if the answer to X on the original set is YES. So our transformation is answer-preserving, as required ... but is it a polynomial-time transformation? Yes it is, because all we need to do is make a single, constant-time change to each element of the set - the entire transformation requires O(n) time.

Thus if we can find an efficient algorithm for Y, we will also have an efficient algorithm for X.

Our term for this kind of transformation is **reduction**. We say **X reduces to Y**. This is confusing to many people because an intuitive interpretation of the word "reduce" often suggests "simplify". Here, the reduction goes from the "easier" problem (X) to the "harder" or "more general" problem (Y). It is useful to remind ourselves exactly what we mean by **X reduces to Y: if we could solve Y, then we could also solve X**.

Reduction is NOT about:
- showing that there is an efficient algorithm for Y
- showing that there is an efficient algorithm for X
- showing that X is difficult or easy
- showing that Y is difficult or easy

Reduction IS about:
- showing that **IF** we could solve Y in polynomial time, **THEN** we could also solve X in polynomial time by transforming instances of X into instances of Y.

The standard notation for reduction is $\propto$.

For our problems above, we write X $\propto$ Y

The next step in the argument is to observe that reduction is **transitive**. If X $\propto$ Y, and Y $\propto$ Z, then X $\propto$ Z - the transformation now takes two steps, but it is still polynomial-time, and (this is crucial) it is still answer-preserving.

Now we can imagine long chains of problems linked by reduction. The first problems in each chain would be quite easy, and as we move along the chains the problems get harder and harder. The problems we are searching for, the most difficult problems in NP, will be at the far ends of the chains.

In class the question was asked "Do these chains go on forever, or do they reach an end?" My (very satisfying) answer was "Both. The chains do go on forever, and they also reach an end." What I mean by this is that there is always another problem that we can reduce to, but the *difficulty* of these problems (as long as we stay in the class **NP**) reaches a maximum level and stays there.

Finally, we can get back to Cook and Levin. To describe their amazing discovery and its importance, we need one or two more definitions.

SAT: SAT is a problem in NP, defined as follows: Let E be a Boolean expression with n **literals** (a literal is just a Boolean variable, possibly negated). Each literal may occur more than once in E. Is there a way to assign True and False to the literals in E so that E is true?

Example: Let $E = (x_1 \wedge \neg x_2) \vee \neg x_1$ If we let $x_1 = True$ and $x_2 = False$, E evaluates to True, so E is satisfiable.

Example: Let $E = (x_1 \wedge \neg x_2) \wedge (x_2 \vee \neg x_1)$. E is not satisfiable ... you can verify this for yourself.

And now at last the jewel in the crown – the most important result in the history of the study of algorithms (which is the most important part of computer science ... but maybe I'm biased):

The Cook-Levin Theorem:

Let X be any problem in NP. Then X reduces to SAT.

This means if we could solve SAT in polynomial time then we could solve every problem in **NP** in polynomial time too. That would have a couple of interesting implications. For one thing, it would mean that all of the decision problems that have defeated everyone who has attempted to find good algorithms for them for the last 50 years, actually do have polynomial time algorithms. For another thing, it would mean **P = NP** ... which means that our normal, real-world computers have just as much power as magical, guess-the-right-answer-every-time Non-deterministic computers. Most people don't believe either of these things ... and so most people believe that SAT simply cannot be solved in polynomial time.

Well, so what? The number of times in a day I am confronted with a Boolean expression and asked to determine if it is satisfiable ... is small. I can probably live the rest of my life without needing to solve an instance of this problem.

But SAT is not alone! Immediately after Cook and Levin opened the gate by proving that all problems in NP reduce to SAT, other researchers (notably, Karp) used the Cook-Levin Theorem to prove that there are infinitely many other problems that have the same property ... and many of those problems are highly practical, real-world, every day problems.

Definition:

A problem X in **NP** is called **NP-Complete** if all problems in **NP** reduce to X (ie if finding a polynomial-time algorithm for X would show **P = NP**).

At this point we have only identified one NP-Complete problem: SAT. It is time to look at how we can find others. Fortunately, the mechanism for showing that a problem is NP-Complete is something we already know: reduction.

Now we can finally get to the practical use of all this theorizing about magical computers, polynomial-time transformations, and so on. When we are presented with a new problem to be solved, our first task as algorithm designers **must be** to ask "Will we ever be able to find a good (ie polynomial-time) algorithm for this problem?" If the problem is NP-Complete, the answer (realistically) is No ... because finding such an algorithm would prove P = NP, and we don't believe that is true. So if our new problem is NP-Complete, we should not waste our time looking for a polynomial-time algorithm to solve it. We have to choose between finding a fast algorithm that sometimes gives the wrong answer, or an algorithm that always gives the right answer but sometimes (or always) takes a very long time.

Determining that a new problem is NP-Complete can save us from wasting a ton of time searching for a polynomial-time algorithm. That's the practical value of this excursion into complexity theory.

So if we are given problem X, how do we prove X is NP-Complete? We could do what Cook and Levin did: work out a complete template for reducing every problem in **NP** to X in polynomial time ... but that's a lot of work. There's a much easier way. All we need to do is

1. Show X is in **NP** - this is usually the easy part
2. Find some **known NP-Complete** problem Y for which we can show $Y \propto X$ ... and then we are done!

Why does this suffice? Because Y is NP-Complete, we already know every problem in **NP** reduces to Y. Now we have shown Y reduces to X. Therefore by transitivity, every problem in **NP** reduces to X ... and that is equivalent to saying X is NP-Complete.

As an example of this ...

Let's look at two very popular problems ...

Partition: Given a set S of n positive integers, can S be partitioned (ie divided) into two subsets S_1 and S_2 with the sum of the elements of S_1 equal to the sum of the elements of S_2?

Note that Partition is in NP.

Subset Sum: Given a set S of positive integers and a positive integer k, does S contain a subset that sums to k?

Subset Sum is also in NP.

Suppose we know that Partition is NP-Complete (in fact, it is!). We can use this knowledge to show that Subset Sum is also NP-Complete. The reduction from Partition to Subset Sum is trivial because Subset Sum is a generalization of Partition. In Partition we are looking for a subset that sums to a specific value: half the total sum of S. If such a subset exists, the answer to Partition for this set S is "Yes". But to transform this to an instance of Subset Sum, all we need to do is let the target value k = half the total sum of S. (This is exactly the same approach as we used to reduce the "Does a set contain the value 4?" problem to the "Does a set contain the value k?" problem.) This reduction takes constant time and it is answer-preserving ... so we know Partition reduces to Subset Sum. And from that we know Subset Sum is also NP-Complete. Make sure you see why we know this!

This very simple proof that Subset Sum is NP-Complete (based on the assertion that Partition is NP-Complete) illustrates a very powerful truth: if a special case of a problem is known to be NP-Complete, then the general case is also NP-Complete.

Be careful with this. If we know that the general case of a problem is NP-Complete, it is not safe to assume that all special cases are also NP-Complete.

For example, consider this (ridiculously simple) problem: Let E be a Boolean expression containing no more than four literals. Can E be satisfied? This is obviously a special case of the SAT problem, and it is equally obvious that instances of this problem can be solved by trying all combinations of True and False for the literals – there can be no more than 16 combinations to try.

Here we know that the general case (SAT) is NP-Complete, but the special case we just defined is not NP-Complete.

However, in the case of Partition and Subset Sum, we **can** show that the general case reduces to the special case.

This is what we will do on 190122.