CMSC 351: P, NP, Etc. Part 1

Justin Wyss-Gallifent

August 5, 2025

1 Introduction

Here we're going to abstract our knowledge of time complexity to look at classifying problems. In order to do this we need to introduce the concepts of decision problems, Turing machines, etc.

2 Informalities

To start us off let's look at some informalities associated with a specific problem.

A great example to consider is the subset-sum problem. Given a list containing $n \geq 1$ integers is there a nonempty subset which sums to 0?

As n increases this problem seems to get harder and harder to solve.

Example 2.0.1. Consider these examples:

(a) $n = 1$: An example is $\{3\}$ and the answer is obviously NO. Another example is $\{0\}$ and the answer is obviously YES.

(b) $n = 2$: An example is $\{4, -4\}$ and the answer is obviously YES. Another example is $\{1, -3\}$ and the answer is obviously NO.

(c) $n = 5$: An example is $\{5, 3, -1, 6, -2\}$ and the answer is slightly less obviously YES because $\{3, -1, -2\}$ is such a subset. Another example is $\{2, 4, -3, -10\}$ and the answer is slightly less obviously NO.

(d) $n = 10$: An example is $\{1, 5, 10, 3, -6, -2, 4, 17, -9, 8\}$ and the answer is perhaps not obvious at all. It is YES.

How might you solve this algorithmically for any n? One option would be to iterate through all $2^n - 1$ different nonempty subsets and sum each one. Each sum is $\mathcal{O}(n)$ so this takes time $\mathcal{O}(n(2^n - 1))$. Can this be done more quickly?

On the other hand if a set and a subset are presented to us then it's easy for us to check that particular subset. Doing that is just $\mathcal{O}(n)$.

3 Decision Problems

3.1 What is a Decision Problem?

Definition 3.1.1. A *decision problem* is a problem which taken an input, formally an *instance*, and produces either YES or NO.

We'll often abstractly write Q for a decision problem and I for an instance and then $Q(I) = YES$ or $Q(I) = NO$ depending on whether the result is YES or NO for that instance.

Note 3.1.1. It's not uncommon to write $I \in Q$ instead of $Q(I) = YES$ and $I \notin Q$ instead of $Q(I) = NO$ in which we treat Q as the set of instances for which the answer is YES.

Example 3.1.1. Q: Given a set containing $n \geq 1$ integers is there a nonempty subset which sums to 0?
I: $\{1, 2, 4, 5, -3\}$ yields $Q(I) = YES$
I: $\{1, 2, 3, 4, -10\}$ yields $Q(I) = NO$

Example 3.1.2. Q: Given two integers numbers x and y, is the sum greater than 100?
I: $x = 50, y = 80$ yields $Q(I) = YES$
I: $x = 10, y = 10$ yields $Q(I) = NO$

Example 3.1.3. Q: Given a natural number n, does n have nontrivial factors?
I: $n = 10$ yields $Q(I) = YES$
I: $n = 5$ yields $Q(I) = NO$

Example 3.1.4. Q: Given a list of integers A, is it sorted?
I: $A = [1, 2, 3]$ yields $Q(I) = YES$
I: $A = [2, 1, 3]$ yields $Q(I) = NO$

Example 3.1.5. Q: Given a graph G, does it contain a cycle?
I: G is a complete graph on at least three vertices yields $Q(I) = YES$
I: G is a tree yields $Q(I) = NO$

3.2 Rephrasing as Decision Problems

Many types of problems which are not decision problems can be rephrased as decision problems. The rephrased problem will not be exactly the same but will carry over the same notion in what appears to be a computationally equivalent sense.

Example 3.2.1. Non-decision problem: Find a solution to a partially filled $n^2 \times n^2$ Sudoku board.

This can be rephrased as the decision problem:

Q: Given a partially filled $n^2 \times n^2$ Sudoku board, does a solution exist?

Observe that deciding if a solution exists essentially seems to be as difficult as finding one.

Example 3.2.2. Non-decision problem: For a graph G, find the shortest path between two vertices s and t.

This can be rephrased as:

Q: Given two vertices s and t and a length k, is there a path of length less than or equal to k between s and t?

Observe that deciding if there is a path of length less than or equal to k essentially seems to be as difficult as finding one.

3.3 Witnesses

Definition 3.3.1. Given a decision problem and an instance if the answer is YES then a *witness* (or *certificate* or *proof*) is some sort of evidence that the answer is YES.

We'll often use the terms *potential witness* for a some evidence which may or may not be good enough and then *valid witness* and *invalid witness.*

Note that a witness can take many forms, depending on how it might be verified.

Example 3.3.1. Q: Given a set represented as a 0-indexed list, is there a subset which adds to 0?
I: `[1,4,-3,5,-1]`
It is clear that $Q(I) = YES$ but there are several ways to give a witness:

- We could give the subset itself: `[4,-3,-1]`
- We could give the indices of the elements in the subset: `[1,2,4]`
- We could give the complement of the subset itself: `[1,5]`
- And on and on.

Note 3.3.1. If we have access to a valid witness for a decision problem and instance then we can say that the answer is YES. However if we don't have access to a valid witness or if we have an invalid witness then we cannot say that the answer is NO.

Example 3.3.2. For the decision problem: Given a set, is there a subset which adds to 0?

For the instance $\{1, 4, -3, 5, -1\}$ if we present $\{1, 4\}$ then this is an invalid witness which tells us nothing about whether $Q(I) = YES$ or $Q(I) = NO$.

4 Turing Machines

4.1 Deterministic Turing Machines

Definition 4.1.1. For us a *Deterministic Turing Machine* (DTM) is essentially your average everyday computer. Loosely speaking at any given instant it is in some state. It reads a series of instructions (a program) and each instruction updates the state to some other (or perhaps the same) state.

The term *deterministic* means that for any state and any instruction there is only one resulting state.

Example 4.1.1. For example a simple DTM might be in a state such as `00` and when it receives the instruction `1` the state is changed to `01`. For this DTM, because it is deterministic, whenever it is in the state `00` and receives the instruction `1` it will always go to the state `01`.

This "determined" behavior is the reason behind the term *deterministic*.

4.2 Nondeterministic Turing Machines

Definition 4.2.1. For us a *Nondeterministic Turing Machine* (NTM) differs from a DTM in that for any state and any instruction that instruction could (but doesn't have to) put the machine in any finite (yet unbounded) number of states simultaneously.

Example 4.2.1. For example a simple NTM might have a state such as `00` and when it receives the instruction `1` the state is changed to both `01` and `10` simultaneously.

Note 4.2.1. At this point you might argue that this is rubbish - how can a machine be in two states at once?

Let's pause here to state that while a NTM has a formal mathematical definition we should think of it here more as a thought experiment.

It is as though you had a computer which you booted up, fed it a program, and it went on to potentially branch into many different states simultaneously every time it encountered an instruction.

It cannot branch into infinitely many states simply because the machine itself only has finitely many states it could be in, even if it branched into all of them simultaneously.

4.3 TMs and Algorithms

When we discuss Turing Machines and Algorithms we'll just refer to Turing Machines with the implication that the algorithm is part of the machine. This way we don't need to talk about algorithms running on machines, rather we just talk about machines.

4.4 Comparison of DTM and NTM and ...

Theorem 4.4.1. Every DTM is a NTM.

Proof. A DTM is simply a NTM in which each state and instruction leads to exactly one state. *QED*

Note 4.4.1. A NTM is not the same as parallel computing, rather a NTM is significantly more powerful.

Note 4.4.2. A NTM is not the same as quantum computing, rather it's conjectured that they are noncomparable, meaning that each has strengths and weaknesses when compared to the other.

4.5 Behold the NTM's Power

Nondeterministic Turning Machines can handle decision problems in rather interesting ways. Consider the subset-sum problem again:

Example 4.5.1. Suppose we are given a set of n integers and we want to decide if there is a nonempty subset which sums to 0.

As far as we know a DTM would have to iterate through each nonempty subset (or at least nobody has come up with a much faster way). There are $2^n - 1$ nonempty subsets and checking the sum of each is $\mathcal{O}(n)$ resulting in a runtime of $\mathcal{O}(n(2^n - 1)) = \mathcal{O}(n2^n)$.

We can design a NTM to act differently, though. It proceeds as follows:

The NTM looks at the first element in the set and the NTM branches into two states - in one state that element is included and in the other it is not. This branching is done simultaneously and takes time $\mathcal{O}(1)$ because it is independent of the length of the list.

From each of those two states the NTM looks at the second element in the set and the NTM branches into two states - in one state that element is included and in the other it is not. Again this branching is done simultaneously and takes time $\mathcal{O}(1)$ because it is independent of the length of the list.

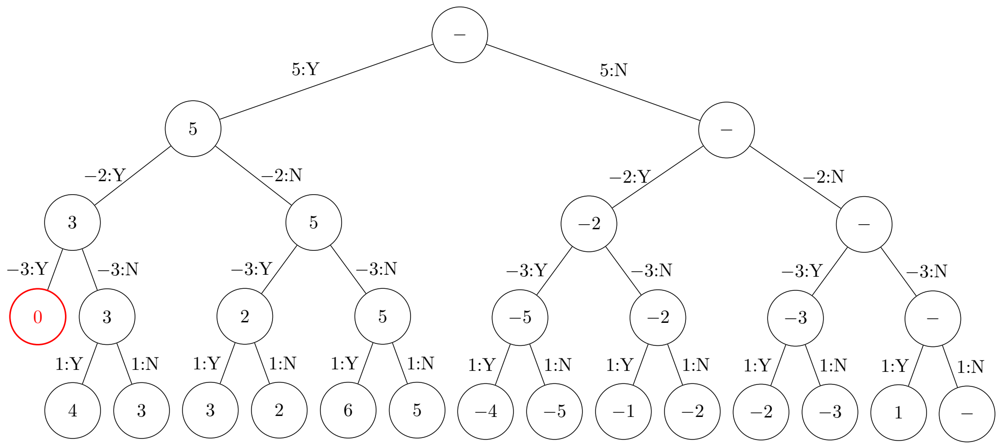
The NTM continues with each element in the set. The result can be thought of as a tree which has height at most n.

In addition to the branching at every state in which at least one number has been included the NTM calculates the sum of the elements included so far and if we ever obtain a sum of 0 we return YES.

These sums are calculated dynamically (each sum is passed to its children) and each child happens simultaneously with the children at the same level (because of the nondeterministic nature of the NTM) meaning the time required is at most $1 + 1 + ... + 1 = \mathcal{O}(n)$ since the tree has height n.

If we never return YES then the answer is NO.

Example 4.5.2. Just to perhaps clarify further here is the above example with the set $[5, -2, -3, 1]$. Each circle indicates the sum of the elements included so far, or $-$ if none are included, and the branches are labeled as to whether each successive element is included or not:



Observe that since (at least) one branch has a sum of 0 we have found a subset and in fact it only took three steps on our NTM.

5 Polytime

Before proceeding:

Definition 5.0.1. An TM runs in *polynomial time* (or *polytime*) if the asymptotic time complexity is $\mathcal{O}(n^k)$ for some integer $k \geq 0$ where n is the input size.

Example 5.0.1. All of our sorting DTMs run in polytime.

Example 5.0.2. Bogosort is a sorting algorithm which iterates through all permutations of a list and checks each to see if it is sorted, stopping when it finds the sorted permutation. Since there are $n!$ permutations this algorithm is $\Omega(n!)$ and does not run in polytime.

Example 5.0.3. There is not a polytime DTM which calculates the sum of each subset of a set with n elements.

Example 5.0.4. There is a polytime NTM which calculates the sum of each subset of a set with n elements.

Note 5.0.1. When we deal with problems that seem to have input sizes relying on multiple variables such as a graph with V vertices and E edges formally we have to decide how we are representing this input in a way in which "size" can be reduced to a single number.

6 The set P

Definition 6.0.1. The set P is the set of all decision problems Q such that $Q \in P$ iff there DTM which runs in polytime on all inputs I and such that:

- If $Q(I) = YES$ then the DTM outputs YES for input I.
- If $Q(I) = NO$ then the DTM outputs NO for input I.

Note 6.0.1. Important to note here is that the DTM runs in polytime where polytime refers to the size of the instance I. There's some nuance here because TMs work on input which consists of a string of characters which the TM understands, meaning we really should be thinking of how to encode our instance I as an string of characters for a particular TM. For example a list like `[2,0,10,7]` might be encoded for a particular TM as `10-0-1010-111`. However for all intents and purposes it's sufficient to talk about the size of I in a more traditional sense such as the number of elements in a list or the number of vertices plus the number of edges in a graph.

Example 6.0.1. Q: Given a list A of length n, is A sorted?

We can create a DTM which takes an instance (a list) A and checks whether $A[i] \leq A[i+1]$ for every $0 \leq i \leq n-2$ and returns YES if so and NO otherwise.

Since we can do this in $\mathcal{O}(n)$ we know that $Q \in P$.

Example 6.0.2. Q: Given two lists A and B of the same length n, do they contain the same values?

We can create a DTM which takes an instance (two lists A and B) sorts them and then compares them element by element. If all elements are equal, return YES, otherwise return NO.

Since sorting can certainly be done in $\mathcal{O}(n^2)$ and comparison in $\mathcal{O}(n)$ we know that $Q \in P$.

Example 6.0.3. Q: Given a graph G with V vertices and two specific vertices s and t and a distance d, is there a path of length less than or equal to d from s to t?

We can create a DTM which takes an instance (the data G, V, s, t, d), runs the shortest path algorithm using s as the starting vertex and checks if the distance from s to t is less than d and replies YES or NO accordingly.

Since the SPA runs in polynomial time and checking a single distance is $\mathcal{O}(1)$ we know that $Q \in P$.

Example 6.0.4. Q: Given a partially filled $n^2 \times n^2$ Sudoku board, is there a solution?

There is no known DTM which runs in polytime and which takes an instance (a partially filled Sudoku board) and replies YES or NO accordingly.

Thus it is unknown if $Q \in P$ or $Q \notin P$. It is suspected that $Q \notin P$.

Example 6.0.5. Q: Given a set A with n elements is there a subset which adds to 0?

There is no known DTM which runs in polytime which takes an instance (a set A) and replies YES or NO accordingly.

Thus it is unknown if $Q \in P$ or $Q \notin P$. It is suspected that $Q \notin P$.

Example 6.0.6. Q: Given a graph G with V vertices, does G contain a Hamiltonian cycle? (This is a cycle which contains each vertex exactly once.)

There is no known DTM which runs in polytime which takes any instance (a graph G) and replies YES or NO accordingly.

Thus it is unknown if $Q \in P$ or $Q \notin P$. It is suspected that $Q \notin P$.

Example 6.0.7. Q: Given a program of size n which may or may halt, does it halt?

It has been proven that there is no DTM which runs in polytime which takes any instance (a program) and replies YES or NO accordingly.

Thus $Q \notin P$.

7 The Set NP

7.1 A Reminder of P

Before proceeding a reminder of the definition of P.

Definition 7.1.1. The set P is the set of all decision problems Q such that $Q \in P$ iff there DTM which runs in polytime on all inputs I and such that:

- If $Q(I) = YES$ then the DTM outputs YES for input I.
- If $Q(I) = NO$ then the DTM outputs NO for input I.

7.2 The Original Definition of NP

And here is the original definition of NP. Note the very slight but critical difference in that we are swapping out a DTM for a NTM.

Definition 7.2.1. The set NP is the set of all decision problems Q such that $Q \in NP$ iff there NTM which runs in polytime on all inputs I and such that:

- If $Q(I) = YES$ then the NTM outputs YES for input I.
- If $Q(I) = NO$ then the NTM outputs NO for input I.

Example 7.2.1. As we've seen above, the zero-subset sum problem is in NP.

Example 7.2.2. Determining if an integer $n \geq 2$ has a nontrivial factor is in NP. For each integer $2 \leq k \leq \sqrt{n}$ we simply divide n by k (we can do these all simultaneously) and see if any remainder is 0.

7.3 The Verifier Definition of NP

Now then, let's stop to point out that this was the original definition of NP but there's an equivalent and more modern definition of NP which is based upon verification of solutions. Here is the simple version:

Definition 7.3.1. The set NP is the set of all decision problems Q such that $Q \in NP$ iff there is a polytime DTM $V(I, x)$ (called a verifier in this case) such that:

$$\forall I, \left[Q(I) = YES \longleftrightarrow \exists x, V(I, x) = YES\right]$$

Notice that this can be broken into two parts which often makes it easier to prove once we have offered up a potential verifier:

(a) For all I, if $Q(I) = YES$ then $\exists x, V(I, x) = YES$.

(b) For all I, if $Q(x) = NO$ then $\forall x, V(I, x) = NO$.

Example 7.3.1. Q: Given a list of n integers are any of them greater than 100?

We create a DTM $V(I, x)$ which takes a list (this is I) and a particular element (this is $x \in I$) and simply checks if $x > 100$, returning YES or NO accordingly.

Observe that this verifier satisfies the definition above:

(a) If $Q(I) = YES$ then I contains an element > 100. If we let x be that element then $V(I, x) = YES$ and so $\exists x, V(I, x) = YES$ is true.

(b) If $Q(I) = NO$ then I contains no elements > 100. Thus for all x we have $V(I, x) = NO$ and so $\forall x, V(I, x) = NO$ is true.

Since this DTM runs in polytime we know $Q \in NP$.

Here is the same decision problem with a different verifier:

Example 7.3.2. Q: Given a list of n integers are any of them greater than 100?

We create a DTM $V(I, x)$ which takes a list (this is I) and a particular element (this is x) and iterates through the entire list looking for an element > 100 returning YES if one is found and NO otherwise.

Observe that this verifier satisfies the definition above:

(a) If $Q(I) = YES$ then I contains an element > 100. However we can ignore that element since $V(I, x) = YES$ holds for the x which it finds and so $\exists x, V(I, x) = YES$ is true.

(b) If $Q(I) = NO$ then I contains no elements > 100. Thus for all x we have $V(I, x) = NO$ and so $\forall x, V(I, x) = NO$ is true.

Since this DTM runs in polytime we know $Q \in NP$.

Here is our subset-sum problem:

Example 7.3.3. Q: Given a set of n integers, is there a nonempty subset which adds to 0?

We create a verifier $V(I, x)$ which takes a set (this is I) and a particular subset (this is $x \subset I$) and simply checks if $\Sigma x = 0$, returning YES or NO accordingly.

Observe that this DTM satisfies the definition above:

(a) If $Q(I) = YES$ then I contains a nonempty subset which adds to 0. If we let x be that subset then $V(I, x) = YES$ and so $\exists x, V(I, x) = YES$ is true.

(b) If $Q(I) = NO$ then I contains no nonempty subsets which add to 0. Thus for all $x \subset I$ we have $V(I, x) = NO$ and so $\forall x, V(I, x) = NO$ is true.

Since this DTM runs in polytime we know $Q \in NP$.

Some other problems which can be seen to be NP in a similar way:

Example 7.3.4. Q: Given a graph G does it contain a cycle?

7.4 A Common Definition of NP

It's not infrequent to find NP defined as saying that we can find a polytime DTM $W(I, x)$ which can be tested on potential witnesses and says YES if the witness is valid and NO if it isn't.

Example 7.4.1. Q: Given a set of n integers, is there a nonempty subset which adds to 0?

We create a verifier $W(I, x)$ which takes a set (this is I) and a particular subset (this is $x \subset I$) and simply checks if $\Sigma x = 0$, returning YES or NO accordingly.

This attempted definition is appealing but not entirely correct. The reason for this is nuanced and arises from what happens when we try to formalize this approach.

A first idea might be to understand that what we're thinking is something like the following - finding a polytime DTM denoted $W(I, x)$ such that:

(a) If $W(I, x) = YES$ then x is a valid witness for I.

(b) If $W(I, x) = NO$ then x is not a valid witness for I.

Of course (a) would mean that $Q(I) = YES$, so there's no problem there.

But how about (b)? We might be tempted to say that it means that $Q(I) = NO$ but that's certainly not the case - having a single invalid witness doesn't mean that the decision problem is false for that instance. We might just have not found a valid witness.

So how else could we think of (b)? We might be tempted to simply conclude simply that x doesn't prove $Q(I)$ but this is no different from saying that x is not a valid witness for I.

Consequently a second idea might be to simply get rid of the second condition and say in summary that:

(a) If $W(I, x) = YES$ then $Q(I) = YES$.

(b) If $W(I, x) = NO$ we can conclude nothing.

However this doesn't work either since we could simply define $W(I, x) = NO$ in all cases and we would satisfy our definition.

The way to coax this approach to work is to insist that when $Q(I) = YES$ that there must be a valid witness. We will see more formally why this is the case but for now we'll call this the Common Definition:

Definition 7.4.1. The set NP is the set of all decision problems Q such that $Q \in NP$ iff there is a polytime DTM denoted $W(I, x)$ (also called a verifier in this case) such that:

(a) $\forall I, \forall x, (W(I, x) = YES \longrightarrow Q(I) = YES)$

(b) $\forall I, (Q(I) = YES \longrightarrow \exists x, W(I, x) = YES)$

In this definition (a) captures the intuition but (b) is necessary.

Example 7.4.2. Q: Given a set of n integers, is there a nonempty subset which adds to 0?

We create a verifier $W(I, x)$ which takes a set (this is I) and a particular subset (this is $x \subset I$) and simply checks if $\Sigma x = 0$, returning YES or NO accordingly. Observe that:

(a) For any set I and subset x if $\Sigma x = 0$ then clearly $Q(I) = YES$.

(b) For any set I if $Q(I) = YES$ this quite literally means there is a subset which sums to 0 and if x is that subset then of course $W(I, x) = YES$.

7.5 Equivalence of Definitions

Theorem 7.5.1. Let Q be a decision problem. Then Q satisfies the NTM Definition iff Q satisfies the Verifier Definition.

Proof. A rigorous proof that these two definitions are equivalent takes more formality than we have covered but loosely speaking we proceed as follows:

$\Longrightarrow$: Suppose Q satisfies the NTM Definition so there is a NTM which runs in polytime on all input and outputs YES iff $Q(I) = YES$.

We construct a verifier $V(I, x)$ as follows:

For a given pair I, x we follow the NTM and when a nondeterministic choice needs to be made we use x to make it. Now observe that:

- If $Q(I) = YES$ then there is a valid witness x - think of it as a computation path through the tree describing the NTM which ends with YES. For that x we will have $V(I, x) = YES$ by construction of V.
- If $Q(I) = NO$ then there are no valid witnesses - all computation paths through the tree describing the NTM end with NO. For all x we will have $V(I, x) = NO$.

Thus Q satisfies the verifier definition.

$\Longleftarrow$: Suppose Q satisfies the Verifier Definition.

We construct a (polytime) NTM as follows:

For each I we look at every potential witness of length bounded by the length of the polytime DTM applied to I and our NTM explores all potential witnesses of that length. For a given potential witness x if $V(I, x) = YES$ then our NTM says YES otherwise say NO.

Now observe that if $Q(I) = YES$ then there is some x with $V(I, x) = YES$ and our DTM will output YES as it follows x and if $Q(I) = NO$ then for all x we have $V(I, x) = NO$ and our DTM will output NO for any x it follows.

$\mathcal{QED}$

Theorem 7.5.2. Let Q be a decision problem. Then Q satisfies the Verifier Definition iff Q satisfies the Common Definition.

Proof. $\Longrightarrow$: Suppose Q satisfies the Verifier Definition. Define $W = V$ then observe:

- Suppose for some I, x we have $W(I, x) = YES$. It follows then that $V(I, x) = YES$ and so for that I we know $\exists x, V(I, x) = YES$ and so $Q(I) = YES$.
- Suppose $Q(I) = YES$. Then $\exists x, V(I, x) = YES$ and so $\exists x, W(I, x) = YES$.

Thus Q satisfies the Common Definition.

$\Longleftarrow$: Suppose Q satisfies the Common Definition. Define $V = W$ then observe for any I:

- If $Q(I) = YES$ then $\exists x, W(I, x) = YES$ and so $\exists x, V(I, x) = YES$.
- If $Q(I) = NO$ then $\forall x, W(I, x) = NO$ so $\forall x, V(I, x) = NO$.

Thus Q satisfies the Verifier Definition.

$\mathcal{QED}$

7.6 Some Problems in NP

Some other famous problems which are in NP include:

Example 7.6.1. Given a graph G and two specific vertices s and t, is there a Hamiltonian Path from s to t? A Hamiltonian Path is a path which visits every vertex exactly once.

Example 7.6.2. Given a graph G and an integer $k \geq 0$, does the graph have a k-clique? A k-clique is a subgraph of G containing k vertices and in which every two vertices are adjacent, in other words it contains the complete graph K_k as a subgraph.

Example 7.6.3. Given integers n and k, is there a factor of n which is strictly between 1 and k?

Example 7.6.4. SAT: Given a boolean expression with n variables, is there an assignment of T or F to each variable which makes the entire expression true?

Example 7.6.5. Given a list of cities and distances between them and given a maximum distance k, is there a route visiting all the cities whose total distance is less than k?

Example 7.6.6. Given a partially filled $n^2 \times n^2$ Sudoku board, is there a solution?

8 P vs NP

Theorem 8.0.1. $P \subseteq NP$

Proof. The proof uses the original definition of NP. If $Q \in P$ then we have a polytime DTM to decide Q. Since a DTM is also a NTM we know that $Q \in NP$ as well. $\mathcal{QED}$

The big question is whether $NP \subseteq P$, which would tell us that $P = NP$. Whether this is true is perhaps the biggest unsolved problem in computer science. It is strongly believed that $NP \not\subseteq P$ due to the fact that there are thousands of problems in NP for which none have been shown to be in P, despite years of attempts.

9 Problem Reduction and Equivalence

9.1 Introduction

This last section looks at what it might mean if solving one decision problem might be "as easy as" solving another.

9.2 Decision Problems and DTMs

Definition 9.2.1. For decision problems Q_1 and Q_2 We say that Q_1 is *polynomially reducible* to Q_2, written $Q_1 \leq_P Q_2$ if there is some DTM f which runs in polynomial time which converts instances for Q_1 to instances for Q_2 such that $Q_1(I) = YES$ iff $Q_2(f(I)) = YES$.

Definition 9.2.2. More informally the idea is that we want to know Q_1 and we find a way to answer Q_1 using Q_2 and that step must be polynomial. Sometimes Q_2 is called an *oracle* to give the impression that we don't care about how it works at all, we just simply ask it questions.

Consider this example for clarification:

Example 9.2.1. Suppose a function `ORACLE(n)` decides something for any integer `n` and returns `YES` or `NO`. Don't worry about what it does or how fast it does whatever it does. Now consider the following algorithm for `QUESTION(n)`:

```
function QUESTION(n)
    for i = 1 to n^2 inclusive:
        if ORACLE(i):
            return(YES)
        end if
    end for
    return(NO)
end function
```

Observe that `QUESTION` makes only n^2 calls (a polynomial number) to `ORACLE`. Thus we say that `QUESTION` is polynomially reducible to `ORACLE`. We would thus write `QUESTION` $\leq_P$ `ORACLE`.

Some notes that apply not just to this example but in general:

Note 9.2.1. We are certainly not guaranteed that `QUESTION` runs in polnomial time because we know nothing about how fast `ORACLE` runs.

Note 9.2.2. There may be other ways to answer `QUESTION` and they may be faster or slower than this.

Here is another example:

Example 9.2.2. Consider these two decision problems:

$ISHAMILTON(G)$: Given an undirected graph on n vertices, is there a Hamiltonian cycle in the graph?

$ORACLE(G)$: Given a directed graph on n vertices, is there a Hamiltonian cycle in the graph?

Given a undirected graph G the adjacency matrix for G also represents the adjacency matrix for G' where G' is obtained from G by replacing each (undirected) edge with two edges, one in each direction. This takes no time. We can then apply $ORACLE(G)$ to find a Hamiltonian cycle in G' which is also a Hamiltonian Cycle in G.

It follows that $ISHAMILTON(G) \leq_P ORACLE$.

And one more:

Example 9.2.3. Consider these two decision problems:

$ISZEROSUBSET(S)$: Given a set S of n integers is there a subset which adds to 0?

$ORACLE(S, x)$: Given a set S of n integers and one integer x in the set, is there a subset of $S - \{x\}$ which adds to $-x$?

To see this suppose we have a set S of integers. We iterate through each element x of S and for each we ask if $ORACLE(S, x) == YES$. If it is true for at least one x then we return YES for $ISZEROSUBSET(S)$ and otherwise we return NO.

It follows that $ISZEROSUBSET \leq_P ORACLE$.

9.3 Stepping Away from Decision Problems

This same idea of polynomial reduction can then apply to problems which are not decision problems.

> **Example 9.3.1.** Consider the non-decision problem:
>
> $SOLVE(B)$: Given a partially filled $n^2 \times n^2$ Sudoku board B which has a solution, find it.
>
> And the decision problem:
>
> $ORACLE(B, x, y, v)$: Given a partially filled $n^2 \times n^2$ Sudoku board B, is there a solution with the value v placed into the empty space with coordinates (x, y)?
>
> Observe that if we are given a partially filled Sudoku board we can iterate through the empty spaces and for each one we can test values 1 through n^2 using $ORACLE$. We know there's a solution so for each empty space we will find a value which works so we add this to our solution. At the end we have solved it. Note that there are at fewer than $(n^2)(n^2) = n^4$ empty spaces and we have to test at most n^2 values in each space so this is $\mathcal{O}(n^6)$.
>
> Thus we can calculate $SOLVE$ in polynomial time as a function of $ORACLE$.

9.4 Some Facts about Polynomial Reducibility

Now we can formalize some facts for decision problems Q_1 and Q_2:

- If $Q_1 \leq_P Q_2$ and $Q_2 \in P$ then $Q_1 \in P$.
- If $Q_1 \leq_P Q_2$ and $Q_1 \notin P$ then $Q_2 \notin P$.

For the mathematicians here, a nice way of thinking about this is to imagine three functions $q_1(x)$, $q_2(x)$, and $p(x)$. Suppose we know for sure that $p(x)$ is a polynomial and we know that $q_1(x) = p(q_2(x))$.

Then we can say:

- q_1 is a polynomial in terms of q_1.
- If $q_2(x)$ is a polynomial then $q_1(x)$ is a polynomial.
- If $q_1(x)$ is not a polynomial then $q_2(x)$ is not a polynomial.

Note that if $p(x)$ is not a polynomial we can say nothing.

9.5 Using Explicit Sets

First of all observe that a set A can be thought of as a decision problem Q if we treat the elements in A as instances and say that $Q(I) = YES$ if $I \in A$ and $Q(I) = NO$ if $I \notin A$.

Note 9.5.1. In reality this really works the other way around. Decision problems are really just set-membership problems.

Example 9.5.1. If $A = \{2, 5, 10\}$ then $Q(2) = YES$ and $Q(3) = NO$.

Example 9.5.2. Let A be the set of integer multiple of 3. Then $Q(6) = YES$ and $Q(5) = NO$.

Example 9.5.3. Let A be the set of partially filled sudoku boards which are solvable. Clearly there are some partially filled boards x for which $Q(x) = TRUE$ and some for which $Q(x) = FALSE$.

Definition 9.5.1. Given sets A and B we say that A is *polynomially reducible* to B if there is some function p which can be computed in polynomial time (as a function of the size of its input) and such that $x \in A$ iff $p(x) \in B$.

Note 9.5.2. The idea here is that making a decision about whether $x \in A$ or not can be, in polynomial time, altered to a question about whether $p(x) \in B$ or not.

Example 9.5.4. Let A be the set of integer multiples of 2 and let B be the set of integer multiples of 3.

To prove that A is polynomially reducible to B we define p by $p(x) = 3x/2$. Integer multiplication is a polynomial-time procedure.

Observe that:

$\Longrightarrow$: If $x \in A$ then $x = 2k$ for some $k \in \mathbb{Z}$ and then $p(x) = 3x/2 = 3(2k)/2 = 3k$ so $p(x) \in B$.

$\Longleftarrow$: If $p(x) \in B$ then $p(x) = 3k$ for some $k \in \mathbb{Z}$ and then $3x/2 = 3k$ so $x = 2k$ so $x \in A$.

Example 9.5.5. Define the sets A and B as follows:

$$A = \{s \,|\, s \text{ is a string}\}$$
$$B = \{s \,|\, s \text{ is a string ending with ``Z''}\}$$

To prove that A is polynomially reducible to B we define p by $p(s) = s + \text{``}Z''$ where $+$ is string concatenation. String concatenation is a polynomial-time procedure.

Observe that:

$\Longrightarrow$: If $s \in A$ then $p(s) = s +'' Z''$ ends with "Z'' so $p(x) \in B$.

$\Longleftarrow$: If $p(s) \in B$ then $p(s)$ ends with "Z''. Thus $s +'' Z''$ ends with "Z" and so s is a string so $s \in A$.

10 NP-Complete and NP-Hard Problems

10.1 Introduction

We'll mention NP-complete and NP-hard problems very briefly because they're important in the P vs NP discussion.

10.2 NP-Complete

Definition 10.2.1. A decision problem Q is NP-complete if it is NP and if every other problem in NP is polynomially reducible to Q.

This definition is quite significant; it essentially states that we can solve any NP problem by polynomally reducing it to Q. Thus, importantly, if we showed that just one NP-complete problem were in P then all of a sudden every single problem which is in NP would automatically be in P.

Example 10.2.1. The first problem which was proved to be NP-complete is SAT, which we mentioned earlier,

Example 10.2.2. The subset-sum problem is NP-complete.

Theorem 10.2.1. Any two NP-complete problems are polynomially equivalent.

Proof. Obvious from the definition. $\mathcal{QED}$

We mention this theorem because it is fascinating - it basically states that we can convert any NP-complete problem to any other in a polynomial amount of time. This is significant because it can seem challenging to see how to problems might be related.

For example showing that SAT and the Subset-Sum Problem are polynomially equivalent can be done by explicitly constructing functions - one which converts a boolean expression into a subset-sum problem and the other doing the reverse.

10.3 NP-Hard

Definition 10.3.1. A decision problem Q is NP-hard if every other problem in NP is polynomially reducible to Q.

Notice that the only difference between this definition and the one for NP-Complete is that NP-Hard problems don't need to be in NP themselves.

10.4 Relationships

It is clear from the definitions that every NP-complete problem is NP-hard.

However it is not the case that every NP-hard problem is NP-complete.

Example 10.4.1. The Halting Problem asks - given a program and its input (together these form the instance I) when the program is run with that particular input will it halt or continue to run forever?

This problem is known not to be in NP but every problem in NP is polynomially reducible to it.