Ch. 7 – Computational Complexity

§1. Arithmetical Complexity

In Ch. 5 (Turing Machines) & Ch. 6 (Recursive functions and relations) we introduced two models of computation. We also found that there were relations (decision problems) that were not "algorithmically" decidable and that there were functions which were not "algorithmically" computable. To specify what is an algorithm and what algorithmically means – we can use either Turing Machines or recursive functions. The halting problem was not Turing-decidable because we cannot expect a single (finite) TM to be able to embody all the complexity of all TMs. The same was basically true of the fact that the Busy-beaver was Turing computable – we cannot expect a single TM to embody the complexity of all TMs which halt on the blank tape and produces a certain number of 1's.

Now there are many decision problems which are Turing-decidable (such as whether or not a given number is prime) – but some of them might take an unreasonable (infeasible) amount of time to give a decision (YES/NO answer) even when the input is reasonably small. So we shall endeavor to measure the complexity (how long it will take as a function of the size of the input) of the fastest algorithm that can solve a given decision problem. Now we do not always have to find the fastest algorithm – as soon as we find one that is good/fast enough we can then say the complexity is $\leq$ such & such.

<u>Ex. 1</u> Let us try to estimate how many basic operations (steps) are needed to find the SUM of two positive numbers n_1 and n_2. Let's start with how to find the sum of two 10 digit numbers

```
      ..0 0 1 1-  ← CARRIES  (first carry is always 0)
   9,012,654,378
+  7,891,230,546
   ----------------
      ...4,924  ← ANSWER  (in this case there will be 11 digits)
```

If we consider adding two digits & a carry as a basic step & writing down a carry or a digit of the answer as one basic step also — then we will need:

10 steps for adding two digits and a carry
10+1 steps for writing down 11 carries
10+1 steps for writing down 11 possible digits in the answer.

So we will need $3(10)+2$ basic steps. You might even want to add 10+10 steps for writing down the two inputs. So it will take at most $5(10)+2$ steps.

Now if we had 100 digit numbers as inputs, then it is easy to see that it will take at most $5(100)+2$ basic steps. And if we had to add two k digit numbers then it will take at most $5k+2 \leq 6k$ for $k \geq 2$, steps to add them. So we can take k as the size of the input. Of course, we might not always want to add two positive integers with the same number of digits — so, in general, we can take k as the number of digits in the larger of the two numbers. And we are guaranteed to take less than $6k$ steps. Now we will write this as $O(k)$ steps where $O(k)$ means

that the number of steps required is $\leq ck$ for all $k \geq k_0$, where k_0 & c are constants. Hence addition can be done in at most $O(k)$ basic steps. By the way, the relation between a positive number n, and its number of digits k (in base 10) is given by $k = \lfloor \log_{10}(n) \rfloor + 1$.

For example the number of digits in 786 is given by

$$k = \lfloor \log_{10}(786) \rfloor + 1 = \lfloor 2.895\cdots \rfloor + 1 = 2 + 1 = 3.$$

Also number of digits in 1000 is $\lfloor \log_{10}(1000) \rfloor + 1 = 3 + 1 = 4$.

Ex.2 Now let us estimate how many basic operations (steps) are needed to find the PRODUCT of two k digit numbers.

Let us see how we can multiply two 10 digits numbers

```
      9,012,654,378
  X   7,891,230,546
      ---------------
            2 2 4 4 -   CARRIES
        - - - 6,268 (MULT by 6)
            1 3 3 - -   CARRIES
        - - - 5,120 (PUT A "0" & MULT by 4)
```

Add these two

```
  1 0 0 0 -   CAR
- - - 1,388
```

Let us count multiplying two digits & adding a carry as a basic step. Then we will need

10 steps for multiplying by 6 + 2(10+1) more for the carries & answer.

10 steps for multiplying by 4 + 2(10+2) more for carries & answer

at most 6(10+1) steps for adding these numbers.

And we have to do this for each of the numbers 5, 0, 3, 2, ..., 8, 7. So we will need 10 multiplications of a 10-digit by a digit - and

9 additions of two numbers with at most 20 digits (Remember the number of digits increase by 1 as we go down). So we will need at most

$$\underbrace{10(5.10)}_{\text{for the multiplications}} + \underbrace{10(6.20)}_{\text{for the additions}} = 17.(10)^2.$$

In general, it is not difficult to see that to multiply two 100-digit numbers, we will need 100 multiplications of 100 digit number by a single digit and 99 additions of two numbers with at most 200-digits. This will take at most $17.(100)^2$ basic steps. So we can multiply two k-digit numbers in at most $17k^2 = O(k^2)$ steps.

Now a modern computer does not work like our Turing machines or like us human beings. It can add and multiply almost instantaneously. So if we are dealing with modern computers, we should consider multiplication and addition (as well as subtraction and division giving quotient & remainder) as a basic step. The minimum of steps that it takes to solve a given problem will then be called its Arithmetic Complexity. But this is not a precise enough notion that can be studied much further. We will only look at exponentiation.

<u>Ex.3</u> Suppose we are asked how many multiplications and additions, it will take to compute $(12)^{530}$. Now this looks very easy — all we have to do is to just multiply 12 by itself 530 times. But there are problems.

First of all $(12)^{530} > (10)^{530}$ which is a "1" followed by 530 zeros. This is one mighty big number – the estimated number of particles in the visible universe is less than 10^{100}. So we can't even write down $(12)^{530}$ if we use one particle to store a digit!

That is why we will only consider decision-problems in the remaining sections of this chapter. Our answers will be just 0 or 1, NO or yes, red light or green light! Now let us change the problem and ask what is $(12)^{530} \pmod 7$. Now this does not sound too hard – all we have to do is work with small numbers, less than 7

$$(12)^2 \equiv (12 \bmod 7)(12 \bmod 7) \equiv 5(5) \equiv 4 \pmod 7$$
$$(12)^3 \equiv (12 \bmod 7)^2 \cdot (12 \bmod 7)^1 \equiv 4(5) \equiv 6 \pmod 7$$
$$(12)^4 \equiv (12 \bmod 7)^3 \cdot (12 \bmod 7) \equiv 6(5) \equiv 2 \pmod 7 \ \ldots$$
$$\vdots$$
$$(12)^{530} \equiv (12 \bmod 7)^{529} (12 \bmod 7) \equiv \ldots \quad \text{only 529 steps!}$$

But we can do this much faster.

$$(12)^2 \equiv 4 \pmod 7 \qquad (12)^4 \equiv 4(4) \equiv 2 \pmod 7$$
$$(12)^8 \equiv 2(2) \equiv 4 \bmod 7 \qquad (12)^{16} \equiv 4(4) \equiv 2 \pmod 7$$

And it doesn't take a genius to see that $(12)^{512} \equiv 4 \pmod 7$

So $(12)^{530} \equiv 12^{512} \cdot 12^{16} \cdot 12^{2} \equiv 4 \cdot 2 \cdot 4 \equiv 4 \pmod 7$

Even if we had to use a computer, it would have just takes 9 steps to reach from $(12)^1$, $(12)^2$, $(12)^4$, $(12)^8$, $(12)^{16}$, $(12)^{32}$, $(12)^{64}$, $(12)^{128}$, $(12)^{256}$, to $(12)^{512}$. That's good programming!

§2. Time Complexity from the Turing Machine model

Recall that a decision problem was a problem for which there is a denumerable number of inputs and always, a YES/NO answer. A typical decision problem would be "Is n prime?". The input is usually the base-10 represent of n and the output is YES or NO.

Ex. 1 Let $L = \{a^k b^k : k \geq 0\}$ and $w \in \{a,b\}^*$. Then the problem: "Is $w \in L$?" is a decision problem. This decision problem is algorithmically solvable because we had found a DTM, M, which will halt in an accepting state if $w \in L$; and which will halt in a non-accepting state if $w \notin L$. This DTM is called a decider because it always halt on any input and it gives the correct answer.

The max. number of steps, $f_M(n)$, that this DTM M takes on inputs of size n can be called the time-complexity of this DTM. Let us estimate how many steps this DTM takes to decide whether not a string w of length n is in L.

...	⊔	a	a	a	...	a	b	b	b	...	b	⊔	⊔	...	

First M scans the tape and replaces each "a" with an "x" and then checks for the first "b" and replaces it by a "y". This will take at most n steps. Then it comes back to the rightmost x and replaces the next "a" by an x and looks for the first b, skipping any other a's or y's. This

will take at most $2n$ more steps. If $w \in L$, then M will replace each "a" by an "x", each "b" by a "y", and halt after seeing the blank symbol. Since there will be $\frac{n}{2}$ a's (remember input $a^k b^k$ has size $2k=n$), M will go up and down the tape $\frac{n}{2} \cdot 2n$ times at most. So M will take about n^2 steps to say YES. If no. of a's $\neq$ no. of b's or if there is an "a" after a "b" or a "b" before an "a" — then M will hat in fewer steps. So we can say that $f_M(n) = O(n^2)$. We can then define the TIME complexity of the problem of testing for membership in L by TIME COMPLEXITY $= \min\{f_M(n) : M$ is a DECIDER DTM which correctly checks if $w \in L\}$. We will however do something slightly different.

Suppose we want to represent a number n. Then we have many choices. Base-10 is what we use in ordinary mathematics and Base-2 is very much liked in Computer science. We can also use base-1, which our primitive ansestors used. Now the base-1 reprentation has length n — because we use n 1's to represent n. The base-10 representation has length $\lfloor \log_{10}(n) \rfloor + 1$ and the base-2 representation has length $\lfloor \log_2(n) \rfloor + 1$. So if we want to represent 989,878 it will take 6 digits in base-10 and 20 digits in base 2 — but it will take 989,878 which is really a million digits in base 1. Now any base <u>except 1</u> is good and their representation lengths differ only by a constant. So for example, length of base-2 repr. $= \log_2(10) \cdot$ length of base-10 repr. Note $\log_2(10) \simeq 3.2193$

So when we wish to input a number we can use any base – except base-1. A similar thing is true about Turing Machines – and there is a corresponding theorem.

<u>Def.</u> A multitape DTM is defined in the same way as a regular one-tape DTM. Each of the tapes in multi-tape TM will have its own head and, by convention, the input is always entered onto the first tape. The transitions now will have an extra component (instead of just moving RIGHT or LEFT of the head) – which tells it on which tape to write & move RIGHT or left of the head and then act on what it reads.

<u>Theorem</u>: Let $f_{M_2}(n)$ be the time-complexity of a 2-tape DTM M_2. Then we can find a DTM M_1 which is equivant to M_2 and has time complexity $O\left([f_{M_2}(n)]^2\right)$ provided $f_{M_2}(n) \geq n$

Moreover, this is basically true for k-tape TM M_k (for $k \geq 2$). So basically, for a given algorithm, all the k-tape DTMs will have the similar time complexity while a 1-tape DTM will be much slower. So this is why we give the following definition

<u>Def.</u> The TIME COMPLEXITY $f_M(n)$ of a given decider 2-tape DTM M is the <u>largest</u> number of steps it takes for it to correctly process inputs of size n.

The TIME COMPLEXITY, $f_Q(n)$ of a given decision problem (question) Q is defined by

$$f_Q(n) = \min\{f_M(n) : M \text{ is a decider 2-tape DTM which correctly solves the problem } Q.\}$$

So now we know two things — the <u>time complexity</u>, $f_M(n)$, <u>of a given DTM</u> which solves a particular decision problem Q — and the <u>time complexity</u>, $f_Q(n)$, of a given decision problem. Now it is very difficult to find the exact value of $f_M(n)$ because it is not all easy to count the exact no. of steps that are needed to process all inputs of size n. We are usually able to find an upper bound an usually write that in big-O notation to make things even simpler. As for $f_Q(n)$ — that is extremely difficult. We don't even know what is the fastest way to determine whether or not a number n is prime. But we have several PRIMALITY TESTING algorithm and can give the complexities of the algorithms in big-O notation. One last thing — we don't always go back to DTMs when we get a problem like PRIMALITY TESTING — because, we can (in principle) convert any algorithm (in ordinary mathematics) into a DTM or a NTM.

§3. <u>P-type, NP-type, and NP-complete Decision Problems.</u>

Our aim in this final section is to look at two important class of decision problems, P-type & NP-type.

<u>Def.</u> Let $DT(n^k)$ be the class of all decision problems that can be solved by a 2-tape decider DTM in $O(n^k)$ steps. We define the class $\mathcal{P}$ of decision problems by $\mathcal{P} = \bigcup_{k \in \mathbb{N}^+} DT(n^k)$. A problem in $\mathcal{P}$ can be solved in $f(n)$ steps, where $f(n)$ is a polynomial in n and is therefore called a <u>polynomial time problem</u>

Examples of Decision-problems in $\mathcal{P}$.

P1: Let G be a directed graph and u & v be vertices in G. Is there a directed path from u to v in G? The input is $\langle G, u, v\rangle$ and we can the adjacency matrix of G and u & v as the input. The size of the input will be $n.n+2$ if $\{1,2,3,\dots,n\}$ are the vertices of G. This problem can be solved in $O(n^3)$ steps – so P1 is polynomial time. Of course, if we have resort to a decider DTM, both the input and the time the DTM will be longer – but the complexity will still be polynomial, may be a higher degree polynomial in n – but a polynomial all the same.

P2: Let L be a context-free language based on T, and $w \in T^*$. Is $w \in L$?
The input here will w and the size of the input will be $|w|$ = length of w. There will always be a TM which can determine whether or not $w \in L$ in $O(n^3)$ steps.

P3 Let A be an $n \times n$ integer matrix. Is A invertible? Here the input is A and the length of the input can be considered as $n^2.k$ where k = the largest number of digits in the n^2 entries of A. By using row operations, we can quickly tell whether or not A is invertible. All we have to do is to reduce A into an upper triangle matrix U and A will be invertible $\Leftrightarrow$ all the diagonal entries of U are non-zero. Again if we resort to a decider DTM, we will find that it can solve the problem in polynomial time.

There are many other decision problems that can be solved in polynomial times. The problems in P are called feasibly solvable problems – because our modern computers can solve such problems with input size of 1000 in a reasonably short amount of time – usually within a few hours. Decision problems which have time complexity $O(2^n)$ or $O(n!)$ are considered unfeasible – because even with input size as small as 50, our modern computers will take centuries to solve them. There is a class of problems which can be checked in polynomial time if evidence (a certificate) is somehow produced.

Ex.1 Consider the problem of finding the prime factorization of a number n. If we are given a proposed prime-factorization, then we can quickly check in polynomials whether or not the proposed prime-factorization is really correct. These types of problems are usually referred to as NP-type problems. We currently do not know any algorithm which can find the prime factorization of a number n. The input size will, of course, be the no. of digits in n. (By the way this is not a decision-problem – but it can be converted into a decision problem.

Ex.2 Another problem is to determine whether or not a given number n is COMPOSITE. This is a decision problem. In the 16th century, Fermat thought that $2^{2^5}+1 = 4,294,967,297$ was not composite, but no one could verify that.

Then 92 years later Euler show that $(641)(6{,}700{,}417) = 4{,}294{,}967{,}297$. So Fermat was wrong, after all – despite his brilliance. (For a long time it was thought that the COMPOSITE decision problem could not be solved in polynomial time). If n is composite then this can be verified in polynomial time. So COMPOSITE is NP and non-composite = PRIME is co-NP. But in 2004, it was shown that PRIME is in P. So both

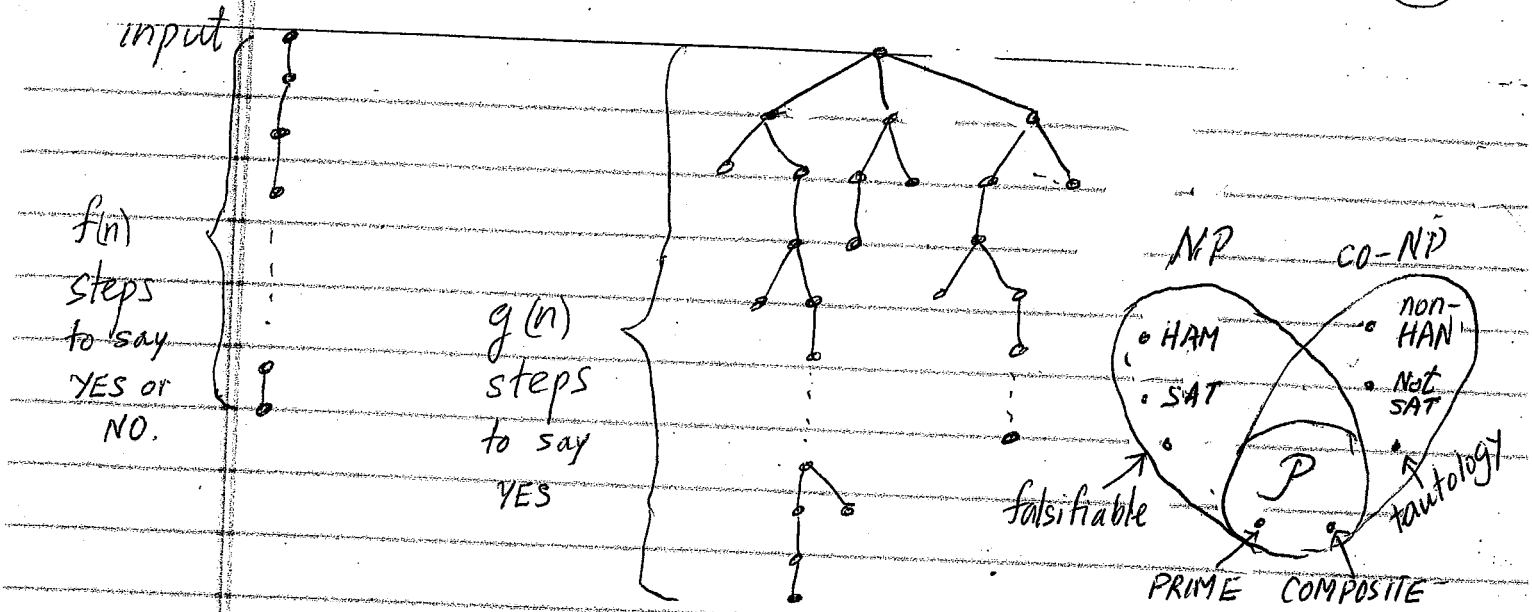
We will now give a precise definition of what it means for a decision problem to be in the class NP.

<u>Def.</u> Let $NT(n^k)$ be the class of all decision problems that can be solved by a decider NTM (Non-deterministic TM) in $O(n^k)$ steps. We define the class NP by

$$NP = \bigcup_{k \in \mathbb{N}^+} NT(n^k)$$

A decision problem in NP can be checked in $g(n)$ steps by a decider NTM, where $g(n)$ is a polynomial in n = input size.

Now any decider NTM is equivalent to a decider DTM – but if the NTM takes $g(n)$ steps, the equivalent DTM will take $r^{g(n)}$ steps where $r > 1$. So the DTM will be much slower. Remember a decider NTM halts on all inputs – but, unlike a DTM, it branches out and works in parallel computation. The time it takes a decider NTM M to solve a problem with input size n is the largest number of steps in all its branches for the input n. We will illustrate this on the next page. The decider NTM must take polynomial time if the answer is YES – but it can take exponential time, $2^{p(n)}$, to say NO.



Now we have just given the exact definition of what is an NP-type decision problem — but it is simple to think of NP-type problems as decision problem for a witness (evidence or a certificate) can be produced out of the blue — and then be verified in polynomial time.

Examples of NP-type decision problems.

<u>N1</u>. Let G be a directed graph. A directed <u>Hamilton-path</u> in G is a directed path which includes all the vertices of G exactly once. How can we tell if a given digraph G has a directed Hamilton path? Well, we could start each vertex and find all possible directed paths and see if any of these directed paths contain all the vertices. (By the way, we are not allowed to repeat a vertex in a path.) Now we can do this — but it will take exponential time in $n = |V(G)|$. But if someone gives us G and a certificate H (which is a path) we can verify that H is indeed a directed Hamilton path in polynomial time. So the Hamilton-path problem is in NP.

N2. The subset-sum Decision Problem

Suppose we are given a set S of n integers and a target integer, t. The problem is to determine whether or not S has a subset A whose sum is t. The input will be $\langle S, t \rangle$ and this will be of size $O(n \log_{10} 10)$. Now if we are given a subset A of S, we can easily verify if the sum of the integers in A is equal to t, in polynomial time. So this is another problem in NP.

Now you might ask, what if there is no subset of S whose sum is equal to t? Well, we can let the NTM (Non-determinstic TM) behave for this case like a DTM & check all the 2^n subsets of S to see if any sum up to t – and if none has the sum t, then our answer will be NO. And all this checking can take more than polynomial-time because we are allowed to take long for NO. We often talk of certificates or witness – because we usually do not want to deal with the NTMs. Remember a NTM is a theoretical concept – there is no such thing as a real-life NTM.

We know that any decider NTM can be converted into a decider DTM – just like how any NFA can be converted into a DFA. But the DFA will usually have a lot more states than the corresponding NFA. The same thing is true for the DTM – it will take much more time to solve a decision problem than the corresponding NTM. The question is whether it will take polynomial time more than the NTM. This is asking if $P = NP$.

We do not know as yet if $P = NP$ – but most mathematicians (including myself) think that $P \subsetneq NP$. This means that we think that there are decision problems for which we can check the answer in polynomial time – but which we cannot completely solve in polynomial time. There is some indications (but no proof) that we might be right.

N3. The CNF-Satisfiability problem. (CNF-SAT)
Given an proposition in Conjunctive Normal Form (CNF) with n variables, is there a truth assignment to the variables that satisfies the proposition? A proposition is CNF if it is of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ where each C_i is of the form $A_1 \vee A_2 \vee A_3 \vee \cdots \vee A_i$ wher each $A_i = \bar{x}_j$ or x_j. For example $(x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3 \vee \bar{x}_1) \wedge (\bar{x}_4 \vee x_2)$ is in CNF form. Now this is clearly a decision problem and any proposed truth assignment can be verified in polynomial time. Hence CNF-SAT is in NP. Now the amazing thing is that <u>any</u> NP problem can be reduced to a polynomial-size input CNF-SAT problem in polynomial time. So if we can solve CNF-SAT in polynomial time, then we will be able to solve any NP problem in polynomial time – and so get $P = NP$. Unfortunately, we do not know how to solve CNF-SAT in polynomial time. The CNF-SAT problem is called <u>NP-complete</u> because any NP problem can be reduce to it in polynomial time. There are many other NP-complete problems. We can define space-complexity as the amount of space needed. We can show that $P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$ but this will need a full course in complexity Theory.

END