Complexity Classes

All of our computing devices now possess two additional attributes: time and space bounds. We shall take advantage of this and classify all of the recursive sets based upon their computational complexity. This allows us to examine these collections with respect to resource limitations. This, in turn, might lead to the discovery of special properties common to some groups of problems that influence their complexity. In this manner we may learn more about the intrinsic nature of computation. We begin, as usual, with the formal definitions.

> **Definition.** *The class of all sets computable in time t(n) for some recursive function t(n),* ***DTIME(t(n))*** *contains every set whose membership problem can be decided by a Turing machine which halts within O(t(n)) steps on any input of length n.*

> **Definition.** *The class of all sets computable in space s(n) for some recursive function s(n),* ***DSPACE(s(n))*** *contains every set whose membership problem can be decided by a Turing machine which uses at most O(s(n)) tape squares on any input of length n.*

We must note that we defined the classes with bounds of *order* t(n) and *order* s(n) for a reason. This is because of the linear space compression and time speedup theorems presented in the section on measures. Being able to use order notation brings benefits along with it. We no longer have to mention constants. We may just say n^2 time, rather than $3n^2 + 2n - 17$ time. And the bases of our logarithms need appear no longer. We may now speak of nlogn time or logn space. This is quite convenient!

Some of the automata theoretic classes examined in the study of automata fit nicely into this scheme of complexity classes. The smallest space class, DSPACE(1), is the class of regular sets and DSPACE(n) is the class of sets accepted by deterministic linear bounded automata. And, remember that the smallest time class, DTIME(n), is the class of sets decidable in *linear* time, not real time.

Our first theorem, which follows almost immediately from the definitions of complexity classes, assures us that we shall be able to find all of the recursive sets within our new framework. It also provides the first characterization of the class of recursive sets we have seen.

> **Theorem 1.** *The union of all the DTIME(t(n)) classes or all of the DSPACE(s(n)) classes is exactly the class of recursive sets.*

Proof. This is quite stratighforward. From the original definitions, we know that membership in any recursive set can be decided by some Turing machine that halts for every input. The time and space functions for these machines name the complexity classes that contain these sets. In other words, if M_a decides membership in the recursive set A, then A is obviously a member of DTIME($T_a(n)$) and DSPACE($L_a(n)$).

On the other hand, if a set is in some complexity class then there must be some Turing machine that decides its membership within some recursive time or space bound. Thus a machine which always halts decides membership in the set. This makes all of the sets within a complexity class recursive.

We now introduce another concept in computation: *nondeterminism*. It might seem a bit strange at first, but examining it in the context of complexity is going to provide us with some very important intuition concerning the complexity of computation. We shall provide two definitions of this phenomenon.

The first is the historical definition. Early in the study of theoretical computing machines, the following question was posed.

Suppose a Turing machine is allowed several choices of action for an input-symbol pair. Does this increase its computational power?

Here is a simple example. Consider the following Turing machine instruction.

0	**1**	**right**	**next**
1	**1**	**left**	**same**
1	**0**	**halt**	
b	**1**	**left**	**I75**

When the machine reads a one, it may either print a one and move left or print a zero and halt. This is a choice. And the machine may choose either of the actions in the instruction. If it is possible to reach a halting configuration, then the machine accepts.

We need to examine nondeterministic computation in more detail. Suppose that the above instruction is I35 and the machine containing it is in configuration: #0110(I35)110 reading a one. At this point the instruction allows it to make either choice and thus enter one of two different configurations. This is pictured below as figure 1.

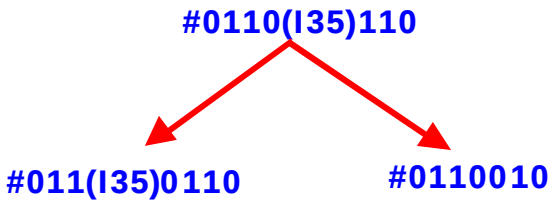

Figure 1 - A Computational Choice

We could now think of a computation for one of these new machines, not as a mystical, magic sequence of configurations, but as a tree of configurations that the machine could pass through during its computation. Then we consider paths through the computation tree as possible computations for the machine. Then if there is a path from the initial configuration to a halting configuration, we say that the machine halts. A more intuitive view of set acceptance may be defined as follows.

> **Definition.** *A nondeterministic Turing machine accepts the input x if and only if there is a path in its computation tree that leads from the initial configuration to a halting configuration.*

Here are the definitions of nondeterministic classes.

> **Definition.** *For a recursive function t(n) is* ***NTIME(t(n))*** *is the class of sets whose members can be accepted by nondeterministic Turing machines that halt within O(t(n)) steps for every input of length n.*

> **Definition.** *For a recursive function s(n),* ***NSPACE(s(n))*** *is the class of sets whose members can be accepted by nondeterministic Turing machines that use at most O(s(n)) tape squares for any input of length n.*

(NB. We shall see NSPACE(n) in the context of formal languages. It is the family of context sensitive languages or sets accepted by nondeterministic linear bounded automata.)

Now that we have a new group of computational devices, the first question to ask is whether or not they allow us to compute anything new. Our next theorem assures us that we still have the recursive sets. It is given with a brief proof sketch since the details will be covered in other results below.

> **Theorem 2.** *The union of all the NTIME(t(n)) classes or all of the NSPACE(s(n)) classes is exactly the class of recursive sets.*

Proof Sketch. There are two parts to this. First we maintain that since the recursive sets are the union of the DTIME or DSPACE classes, then they are all contained in the union of the NTIME or NSPACE classes.

Next we need to show that any set accepted by a nondeterministic Turing machine has a decidable membership problem. Suppose that a set is accepted by a t(n)-time nondeterministic machine. Now recall that the machine accepts if and only if there is a path in its computation tree that leads to a halting configuration. Thus all one needs to do is to generate the computation tree to a depth of t(n) and check for halting configurations.

Now let us examine nondeterministic acceptance from another viewpoint. A path through the computation tree could be represented by a sequence of rows in the instructions that the machine executes. Now consider the following algorithm that receives an input and a path through the computation tree of some nondeterministic Turing machine M_m.

```
Verify(m, x, p)
Pre:  p[] = sequence of rows in instructions to be
      executed by Mm as it processes input x
Post: halts if p is a path through the computation tree

i = 1;
config = (I1)#x;
while config is not a halting configuration and i≤ k do
  i = i + 1;
  if row p(i) in config's instruction can be executed by Mm
     then set config to the new configuration
     else loop forever
if config is a halting configuration then halt(true)
```

This algorithm verifies that p is indeed a path through the computation tree of M_m and if it leads to a halting configuration, the algorithm halts and accepts. Otherwise it either loops forever or terminates without halting. In addition, the algorithm is *deterministic*. There are <u>*no*</u> choices to follow during execution.

Now let us examine paths through the computation tree. Those that lead to halting configurations show us that the input is a member of the set accepted by the machine. We shall say that these paths *certify* that the input is a member of the set and call the path a *certificate of authenticity* for the input. This provides a clue to what nondeterministic operation is really about.

Certificates do not always have to be paths through computation trees. Examine the following algorithm for accepting nonprimes (composite numbers) in a nondeterministic manner.

```
NonPrime(x)
nondeterministically determine integers y and z;
if y*z = x then halt(true)
```

Here the certificate of authenticity is the pair <y, z> since it demonstrates that x is not a prime number. We could write a completely deterministic algorithm which when given the triple <x, y, z> as input, compares y*z to x and certifies that x is not prime if x = y∗z.

This leads to our second definition of nondeterministic operation. We say that the following deterministic Turing machine M uses certificates to *verify membership in the set A.*

M(x, c) halts if and only if c provides a proof of $x \in A$

The nondeterministic portion of the computation is finding the certificate and we need not worry about that. Here are our definitions in terms of verification.

> **Definition.** *The class of all sets nondeterministicly acceptable in time t(n) for a recursive function t(n),* ***NTIME(t(n))*** *contains all of the sets whose members can be verified by a Turing machine in at most O(t(n)) steps for any input of length n and certificate of length $\leq t(n)$.*

Note that certificates must be shorter in length than t(n) for the machine to be able to read them and use them to verify that the input is in the set.

We should also recall that nondeterministic Turing machines and machines which verify from certificates do not decide membership in sets, but accept them. This is an important point and we shall come back to it again.

At this point we sadly note that the above wonderfully intuitive definition of nondeterministic acceptance by time-bounded machines does not extend as easily to space since there seems to be no way to generate certificates in the worktape space provided.

We mentioned earlier that there is an important distinction between the two kinds of classes. In fact, important enough to repeat. *Nondeterministic machines accept sets, while deterministic machines decide membership in sets.* This is somewhat reminiscent of the difference between recursive and recursively enumerable sets and there are some parallels. At present the

differences between the two kinds of classes is not well understood. In fact, it is not known whether these methods of computation are equivalent. We do know that

$$\begin{aligned} &DSPACE(1) = NSPACE(1) \\ &DSPACE(s(n)) \subseteq NSPACE(s(n)) \\ &DTIME(t(n)) \subseteq NTIME(t(n)) \end{aligned}$$

for every recursive s(n) and t(n). Whether DSPACE(s(n)) = NSPACE(s(n)) or whether DTIME(t(n)) = NTIME(t(n)) remain famous open problems. The best that anyone has achieved so far is the following result that is presented here without proof.

Theorem 3. *If $s(n) \geq \log_2 n$ is a space function, then $NSPACE(s(n)) \subseteq DSPACE(s(n)^2)$.*

Our next observation about complexity classes follows easily from the linear space compression and speedup theorems. Since time and space use can be made more efficient by a constant factor, we may state that:

$$\begin{aligned} &DTIME(t(n)) = DTIME(k*t(n)) \\ &NTIME(t(n)) = NTIME(k*t(n)) \\ &DSPACE(s(n)) = DSPACE(k*s(n)) \\ &NSPACE(s(n)) = NSPACE(k*s(n)) \end{aligned}$$

for every recursive s(n) and t(n), and constant k. (Remember that t(n) means max(n+1, t(n)) and that s(n) means max(1, s(n)) in each case.)

While we are comparing complexity classes it would be nice to talk about the relationship between space and time. Unfortunately not much is known here either. About all we can say is rather obvious. Since it takes one unit of time to write upon one tape square we know that:

$$TIME(t(n)) \subseteq SPACE(t(n))$$

because a machine cannot use more than t(n) tape squares if it runs for t(n) steps. Going the other way is not so tidy. We can count the maximum number of steps a machine may go through before falling into an infinite loop on s(n) tape and decide that for some constant c:

$$SPACE(s(n)) \subseteq TIME(2^{cs(n)})$$

for both deterministic and nondeterministic complexity classes. And, in fact, this counting of steps is the subject of our very next theorem.)

Theorem 4. *If an s(n) tape bounded Turing machine halts on an input of length n then it will halt within $O(2^{cs(n)})$ steps for some constant c.*

Proof Sketch. Consider a Turing machine that uses O(s(n)) tape. There is an equivalent machine M_i that uses two worktape symbols and also needs no more than O(s(n)) tape. This means that there is a constant k such that M_i never uses more than k∗s(n) tape squares on inputs of length n.

We now recall that a machine configuration consists of:

a) the instruction being executed,
b) the position of the head on the input tape,
c) the position of the head on the work tape, and
d) a work tape configuration.

We also know that if a machine repeats a configuration then it will run forever. So, we almost have our proof.

All we need do is count machine configurations. There are $|M_i|$ instructions, n+2 input tape squares, k∗s(n) work tape squares, and $2^{ks(n)}$ work tape configurations. Multiplying these together provides the theorem's bound.

One result of this step counting is a result relating nondeterministic and deterministic time. Unfortunately it is nowhere near as sharp as theorem 3, the best relationship between deterministic and nondeterministic space. Part of the reason is that our simulation techniques for time are not as good as those for space.

Corollary. *$NTIME(t(n)) \subseteq DTIME(2^{ct(n)^2})$*

Proof. $NTIME(t(n)) \subseteq NSPACE(t(n)) \subseteq DSPACE(t(n)^2) \subseteq DTIME(2^{ct(n)^2})$ because of theorems 3 and 4. (We could have proven this from scratch by simulating a nondeterministic machine in a deterministic manner, but the temptation to use our last two results was just too tempting!)

Our first theorem in this section stated that the union of all the complexity classes results in the collection of all of the recursive sets. An obvious question is whether one class can provide the entire family of recursive sets. The next result denies this.

Theorem 5. *For any recursive function s(n), there is a recursive set that is not a member of DSPACE(s(n)).*

Proof. The technique we shall use is diagonalization over *DSPACE(s(n))*. We shall examine every Turing machine that operates in s(n) space and define a set that cannot be decided by any of them.

First, we must talk about the machines that operate in O(s(n)) space. For each there is an equivalent machine which has one track and uses the alphabet {0,1,*b*}. This binary alphabet, one track Turing machine also operates in O(s(n)) space. (Recall the result on using a binary alphabet to simulate machines with large alphabets that used blocks of standard size to represent symbols.) Let's now take an enumeration M_1, M_2, ... of these one track, binary machines and consider the following algorithm.

```
Examine(i, k, x)
Pre: n = length of x

lay out k*s(n) tape squares on the work tape;
run Mi(x) within the laid off tape area;
if Mi(x) rejects then accept else reject
```

This is merely a simulation of the binary Turing machine M_i on input x using k∗s(n) tape. And, the simulation lasts until we know whether or not the machine will halt. Theorem 4 tells us that we only need wait some constant times $2^{cs(n)}$ steps. This is easy to count to on a track of a tape of length k∗s(n). Thus the procedure above is recursive and acts differently than M_i on input x if $L_i(n) \leq k*s(n)$.

Our strategy is going to be to feed the Examine routine all combinations of k and i in hopes that we shall eventually knock out all s(n) tape bounded Turing machines.

Thus we need a sequence of pairs <i, k> such that each pair occurs in our sequence infinitely often. Such sequences abound. A standard is:

<1,1>, <1,2>, <2,1>, <1,3>, <2,2>, <3,1>, ...

For each input x we take the x^{th} pair in the sequence. The decision procedure for the set we claim is not s(n) space computable is now:

```
Diagonal(x)
select the x-th pair <i, k> from the sequence;
Examine(i, k, x)
```

Two things need to be verified. First, we need to show that the above decision procedure can be carried out by some Turing machine. We note that M_i comes from an enumeration of two work tape symbol machines and then appeal to Church's thesis for the actual machine construction for the decision procedure. Next we need to prove that this procedure cannot be carried out by an s(n) space bounded Turing machine.

Suppose that the Diagonal procedure is indeed s(n) space computable. Then there is some two worktape symbol, s(n) space bounded Turing machine M_i which computes the above Diagonal procedure. And there is a constant k such that for all but a finite number of inputs, M_i uses no more than k∗s(n) tape squares on inputs of length n. In particular, there is an x such that <j, k> is the x^{th} pair in our sequence of pairs and the computation of $M_i(x)$ requires no more than k∗s(n) tape. (In fact there are infinitely many of these x since the pair <j, k> appears infinitely often in the sequence.) In this case

$$M_j(x) \neq \text{Examine}(j, k, x) = \text{Diagonal}(x)$$

which is a contradiction. Thus M_i cannot be an s(n) bounded machine and the set defined by our Diagonal procedure cannot be a member of DSPACE(s(n)).

It should come as no surprise that the same result holds for nondeterministic space as well as time classes. Thus we do have a hierarchy of classes since none of them can hold all of the recursive sets. This seems in line with intuition since we believe that we can compute bigger and better things with more resources at our disposal.

Our next results explore the amount of space or time needed to compute new things for classes with resource bounds that are tape or time functions. (Recall that *tape* or *time* functions are bounds for actual Turing machines.) We consider these resource bounds to be *well behaved* and note in passing that there are strange functions about which are not tape or time functions.

Theorem 6 (Space Hierarchy). *If r(n) and s(n) are both at least O(n), s(n) is a space function, and* $\inf_{n\to\infty} r(n)/s(n) = 0$, *then* $DSPACE(s(n)) - DSPACE(r(n)) \neq \varnothing$.

Proof Sketch. The proof is very similar to that of the last theorem. All that is needed is to change the space laid off in the Examine routine to s(n). Since s(n) grows faster than any constant times r(n), the diagonalization proceeds as scheduled. One note. What makes this simulation and diagonalization possible is that s(n) is a space function.

This allows us to lay off s(n) tape squares in s(n) space. Thus the diagonalization does produce a decision procedure for a set which is s(n) space decidable but not r(n) space decidable.

The major reason the simulation worked was that we were able to lay out s(n) tape squares. This is because we could compute s(n) by taking the machine it was a tape function for and run it on all inputs of length n to find the longest one. This requires O(n) space. If s(n) is even *more well behaved* we can do better.

Definition. *A recursive function s(n) is* ***efficiently space computable*** *if and only if it can be computed within s(n) space.*

If s(n) is efficiently space computable, then the space hierarchy theorem is true for s(n) down to $O(\log_2 n)$ because we can lay out the required space for the simulation and keep track of which input symbol is being read.

Many functions are efficiently space computable, including such all time favorites such as $\log_2 n$ and $(\log_2 n)^k$. An exercise dealing with efficient space computability will be to prove that all space functions that are at least O(n) are efficiently space computable.

Combining the space hierarchy theorem with the linear space compression theorem provides some good news at last. If two functions differ only by a constant, then they bound the same class. But if one is larger than the other by more than a constant then one class is properly contained in the other.

Sadly the result for time is not as sharp. We shall need one of our functions to always be efficiently time computable and do our simulation with two tapes. Here is the theorem.

Theorem 7 (Time Hierarchy). *If r(n) and t(n) are both at least O(n), t(n) is efficiently time computable, and*

$$\inf_{n \to \infty} \frac{r(n)\log_2 r(n)}{t(n)} = 0$$

then DTIME(t(n)) - DTIME(r(n)) ≠ ∅.