Reducibilities and Completeness.

We now have a framework in which to study the recursive functions in relation to one of the major considerations of computer science: computational complexity. We know that there is a hierarchy of classes but we do not know very much about the classes other than this. In this section we shall present a tool for refining our rough framework of complexity classes.

Let us return to an old mathematical technique we saw in chapter two. There we mapped problems into each other and used this to be make statements about unsolvability. We shall do the same at the subrecursive level except that we shall use mappings to help us determine the complexity of problems. And even use the same kind of mappings that were used before. Here is the definition of reducibility one more time.

> **Definition.** *The set A is* ***many-one reducible*** *to the set B (written* $A \leq_m B$*) if and only if there is a recursive function g(x) such that for all x:* $x \in A$ *if and only if* $g(x) \in B$.

With the function g(x) we have mapped all of the members of A into the set B. Integers not in A get mapped into B 's complement. Symbolicly:

$$g(A) \subseteq B \quad \text{and} \quad g(\overline{A}) \subseteq \overline{B}$$

Note that the mapping is *into* and that several members of A may be mapped into the same element of B by g(x).

An important observation is that we have restated the membership problem for A in terms of membership in B. This provides a new way to decide membership in A. Let's have a look. If the Turing machine M_b decides membership in B and M_g computes g(x) then membership in A can be decided by:

$$M_a(x) = M_b(M_g(x)).$$

(In other words, compute g(x) and then check to see if it is in B.)

Now for the complexity of all this. It is rather straightforward. Just the combined complexity of computing g(x) and then membership in B. In fact, the time and space complexities for deciding membership in A are:

$$L_a(n) = \text{maximum}[L_g(n), L_b(\,|g(x)|\,)]$$
$$T_a(n) = T_g(n) + T_b(\,|g(x)|\,)$$

Thinking a little about the length of g(x) and its computation

we easily figure out that the number of digits in g(x) is bounded by the computation space and time used and thus:

$$|g(x)| \leq L_g(n) \leq T_g(n)$$

for x of length n. This makes the complexity of the decision problem for A:

$$L_a(n) \leq \text{maximum}[L_g(n), L_b(L_g(n))]$$
$$T_a(n) \leq T_g(n) + T_b(L_g(n))$$

An essential aside on space below O(n) is needed here. If $L_b(n)$ is less than O(n) and |g(x)| is O(n) or greater we can often compute symbols of g(x) one at a time as needed and then feed them to M_b for its computation. Thus we need not write down all of g(x) - just the portion required at the moment by M_b. We do however need to keep track of where the read head of M_b is on g(x). This will use $\log_2|g(x)|$ space. If this is the case, then

$$L_a(n) \leq \text{maximum}[L_g(n), L_b(|g(x)|), \log_2|g(x)|]$$

Mapping functions which are almost straight character by character translations are exactly what is needed. As are many $\log_2 n$ space translations.

What we would like to do is to be able to say something about the complexity of deciding membership in A in terms of B's complexity and not have to worry about the complexity of computing g(x). It would be perfect if $A \leq_m B$ meant that A is no more complex than B. This means that computing g(x) must be less complex than the decision problem for B. In other words, the mapping g(x) must preserve | not influence) complexity. In formal terms:

> **Definition.** *Let $A \leq_m B$ via g(x). The recursive function g(x) is **complexity preserving** with respect to **space** if and only if there is a Turing machine M_b which decides membership in B and a Turing machine M_g which computes g(x) such that:*
>
> $$maximum[L_g(n), L_b(L_g(n))] = O(L_b(n)).$$

> **Definition.** *Let $A \leq_m B$ via g(x). The recursive function g(x) is **complexity preserving** with respect to **time** if and only if there is a Turing machine M_b which decides membership in B and a Turing machine M_g which computes g(x) such that:*
>
> $$T_g(n) + T_b(L_g(n)) = O(T_b(n)).$$

These complexity preserving mappings now can be used to demonstrate the relative complexities of decision problems for sets. In fact, we can often pinpoint a set's complexity by the use of complexity preserving reducibilities. The next two theorems explain this.

Theorem 1. *If $A \leq_m B$ via a complexity preserving mapping and B is in DSPACE(s(n)) then A is in DSPACE(s(n)) also.*

Proof. Follows immediately from the above discussion. That is, if $M_a(x) = M_b(g(x))$ where g(x) is a complexity preserving mapping from A to B, then $L_a(n) = O(L_b(n))$.

Theorem 2. *If $A \leq_m B$ via a complexity preserving mapping and the best algorithm for deciding membership in A requires O(s(n)) space then the decision problem for B cannot be computed in less than O(s(n)) space.*

Proof. Suppose that the membership problem for B can be computed in less than O(s(n)) space. Then by theorem 1, membership in A can be computed in less. This is a contradiction.

Now, just what have we accomplished? We have provided a new method for finding upper and lower bounds for the complexity of decision problems. If $A \leq_m B$ via a complexity preserving mapping, then the complexity of A's decision problem is the lower bound for B's and the complexity of B's decision problem is the upper bound for A's. Neat. And it is true for time too.

An easy example might be welcome. We can map the set of strings of the form $0^n\#1^n$ into the set of strings of the form w#w (where w is a string of 0's and 1's). The mapping is just a character by character transformations which maps both 0's and 1's into 0's and maps the marker (#) into itself. Thus we have shown that $0^n\#1^n$ is no more difficult to recognize than w#w.

Now that we have provided techniques for establishing upper and lower complexity bounds for sets in terms of other sets, let us try the same thing with classes. In other words, why not bound the complexity for an entire class of sets all at once?

Definition. *The set A is **hard** for a class of sets if and only if every set in the class is many-one reducible to A.*

What we have is a way to put an upper bound on the complexity for an entire class. For example if we could show that deciding w#w is hard for the context free languages then we would know that they are all in $DSPACE(\log_2 n)$ or $DTIME(n)$. Too bad that we cannot! When we combine hardness with complexity preserving reducibilities and theorem 1, out comes an easy corollary.

Corollary. *If $A \in DSPACE(s(n))$ is hard for $DSPACE(r(n))$ via complexity preserving mappings then $DSPACE(r(n)) \subseteq DSPACE(s(n))$.*

And, to continue, what if a set is hard for the class which contains it? This means that the hard set is indeed the most difficult set to recognize in the class. We have a special name for that.

> **Definition.** *A set is **complete** for a class if and only if it is a member of the class and hard for the class.*

Think about this concept for a bit. Under complexity preserving mappings complete sets are the most complex sets in the classes for which they are complete. In a way they represent the class. If there are two classes with complete sets, then comparing the complete sets tells us a lot about the classes. So, why don't we name the classes according to their most complex sets?

> **Definition.** *The class of sets no more complex (with respect to space) than A **DSPACE(A)** contains all sets B such that for each Turing machine which decides membership in A there is some machine which decides membership in B in no more space.*

Of course we can do the same for time and nondeterministic resources also. We must note that we defined the complexity relationship between A and B very carefully. This is because sets with speedup may be used to name or denote classes as well as those with best algorithms. Thus we needed to state that *for each algorithm for A there is one for B which is no worse,* so if the set B has speedup, then it can still be in *DSPACE(A)* even if A has speedup too. Now for a quick look at what happens when we name a class after its complete set.

> **Theorem 3.** *If the set A is DSPACE(s(n))-complete via complexity preserving reducibilities then DSPACE(s(n)) = DSPACE(A).*
>
> **Proof.** Theorem 2 assures us that every set reducible to A is no more difficult to decide membership in than A. Thus *DSPACE(S(N))* 5 *DSPACE(A).* And, since A is a member of *DSPACE(S(N))* then all members of *DSPACE(A)* must be also since they require less time to decide membership in than A.

That was not unexpected. Seems like we set it up that way! We now go a bit further with our next question. Just what kinds of complexity classes have complete sets? Do all of them? Let us start by asking about the functions which are resource bounds for classes with complete sets and go on from there.

> **Theorem 4.** *Every space complexity class which has a complete set via complexity preserving reducibilities is some DSPACE(s(n)) where s(n) is a space function.*

Proof. Let *DSPACE(r(n))* have the complete set A. Since A is a member of *DSPACE(r(n))*, there must be a Turing machine M_a which decides membership in A in O(r(n)) space. Thus $L_a(n) = O(r(n))$. Which makes *DSPACE*($L_a(n)$) 5 *DSPACE(r(n))*. Theorem 1 assures us that *DSPACE(r(n))* 5 *DSPACE*($L_a(n)$ since all of the the members of *DSPACE(r(n))* are reducible to A.

Theorem 5. *If there is a best algorithm for deciding membership in the set A then there is a space function s(n) such that DSPACE(A) = DSPACE(s(n)).*

Proof. Almost trivial. The space function s(n) which we need to name the complexity class is just the space function for the Turing machine which computes the most efficient algorithm for deciding membership in A.

Now we know that space functions can be used to name some of our favorite complexity classes: those with complete sets, and those named by sets with best decision procedures. Let us turn to the classes named by sets which have no best decision procedure. These, as we recall were strange. They have speedup and can be computed by sequences of algorithms which run faster and faster.

Theorem 6. *If the set S has speedup then there is no recursive function s(n) such that DSPACE(s(n)) = DSPACE(S).*

All of the results mentioned in the last series of theorems (3 - 6) are true for time and both kinds of nondeterministic class. Well, except for theorem 6. For time the speedup must be at least $n\log_2 n$. But that is not too bad.

We would like to have a theorem which states that all classes with space or time functions as their resource bounds have complete sets. This *is* true for a few of these classes, namely those with polynomial bounds. For now, let us leave this section with the reassuring that complete sets can define classes and none of them have speedup.