Measures and Resource Bounds

In order to answer questions concerning the complexity or difficulty of computation, we must first arrive at some agreement on what exactly is meant by the term *complexity*. One is tempted to confuse this with the complexity of understanding just how some task is computed. That is, if it involves an intricate or tedious computing procedure, then it is complex. But that is a trap we shall leave for the more mathematical when they claim that a proof is difficult. We shall base our notion of difficulty upon the very practical notion of computational cost and in turn define this to be related to the amount of resources used during computation. (Simply put: if something takes a long time then it *must* be hard to do!)

Let us consider the resources used in computation. And, most important are those which seem to limit computation. In particular, we will examine *time* and *space* constraints during computation. This is very much in line with computer science practice since many problems are costly to us or placed beyond our reach due to lack of time or space - even on modern computing equipment.

We shall return to Turing machines in order to examine computational difficulty. This may seem rather arbitrary and artificial, but this choice is reasonable since most natural models of computation are not too far apart in the amounts of time and space used in computation for the same functions. (For example, consider the space used by Turing machines and programs that compute the same functions or decide membership in the same sets. They are very similar indeed!) In addition, the simplicity of the Turing machine model makes our study far less cumbersome.

All we need do is associate a time cost function and a space cost function with each machine in order to indicate exactly how much of each resource is used during computation. We shall begin with time.

Our machine model for time complexity will be the multitape Turing machine. Part of the reason for this is tradition, and part can be explained by examining the computations done by one-tape machines. For example, a two tape machine can decide whether a string is made up of n zeros followed by n ones (in shorthand we write this as 0^n1^n) in exactly 2n steps while a one tape machine might require about nlogn steps for the same task. Arguments like this make multitape machines an attractive, efficient model for computation. We shall

assume that we have a *standard enumeration* of these machines that we shall denote:

$$M_1,\ M_2,\ \ldots$$

and define a time cost function for each machine.

> **Definition** *The **time function** $T_i(n)$ is the maximum number of steps taken by multitape Turing machine M_i on any input of length n.*

These multitape machines are able to solve certain computational problems faster than their one-tape cousins. But intuitively, one might maintain that using these machines is very much in line with computation via programs since tapes are nearly the same as arrays and programs may have lots of arrays. Thus we shall claim that this is a sensible model of computation to use for our examination of complexity.

The tradeoffs gained from using many tapes instead of one or two are presented without proof in the next two theorems. First though, we need some additional mathematical notation.

> ***Definition*** *Given two recursive functions f and g, f = O(g) (pronounced: f is the **order** of g or f is **big OH** of g) if and only if there is a constant k such that $f(n) \leq k*g(n)$ for all but a finite number of n.*

This means that smaller functions are the order of larger ones. For example, x^2 is the order of 2^n or $O(2^n)$. And, if two functions are the same up to a constant, then they are of the same order. Intuitively this means that their graphs have roughly the same shape when plotted. Examine the three functions in figure 1.

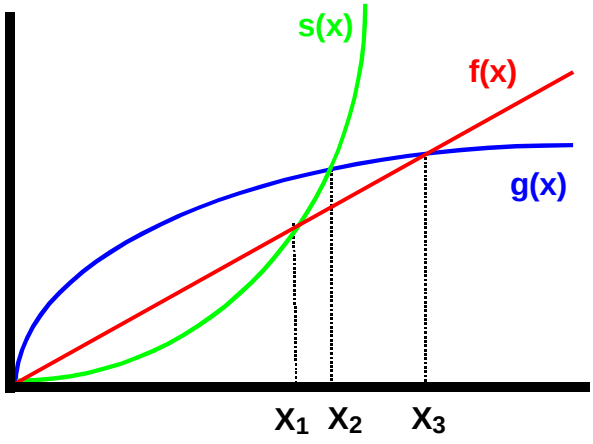

Figure 1 - Three functions

After point x_1, function s(x) is always greater than f(x) and for values of x larger than x_2, it exceeds g(x). Since s(x) grows much faster than f(x) and g(x) we know that for any constant k, after some point $s(x) > k*f(x)$ and $s(x) > k*g(x)$. Due to this, we say that f(x) = O(s(x)) and g(x) = O(s(x)). Similarly, at x_3, f(x) becomes larger than g(x) and since it remains so, we note that g(x) = O(f(x)). (The folk rule to remember here is that small, or slowly growing functions are the order of larger and faster ones.)

Let us think about this concept. Consider a linear function (such as 6x) and a quadratic (like x^2). They do not look the same (one is a line and the other is a curve) so they are not of the same order. But $5x^3 - 2x^2 - 15$ is $O(x^3)$ because it is obviously less than $6x^3$. And $\log_2(n^2)$ is $O(\log_2 n)$. While we're on the subject of logarithms, note that logs to different bases are of the same order. For example:

$$\log_e x = (\log_e 2) \log_2 x.$$

The constant here is $\log_e 2$. This will prove useful soon. Here are the theorems that were promised that indicate the relationship between one-tape Turing machines and multi-tape machines.

> **Theorem 1.** *Any computation which can be accomplished in t(n) time on a multitape Turing machine can be done in $O(t(n)^2)$ time using a one tape Turing machine.*

> **Theorem 2.** *Any computation that can be accomplished in t(n) time on a multitape Turing machine can be done in $O(t(n)\log_2(t(n))$ time using a two tape Turing machine.*

Now we need to turn our attention to the other resource that often concerns us, namely space. Our model will be a little different.

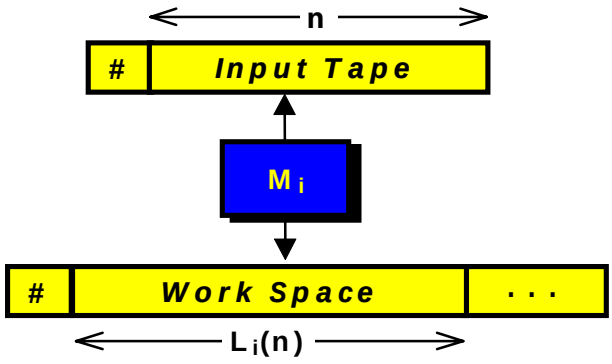

Figure 2 - Space Complexity Model

We shall not allow the Turing machine to use its input tape for computation, which means using a read-only input tape. For computation we shall give the machine one (possibly multi-track) work tape. That looks like the picture in figure 3.

The formal definition of space complexity now follows.

> **Definition.** *The **space function** $L_i(n)$ is the maximum number of work tape squares written upon by Turing machine M_i for any input of length n.*

Several additional comments about our choice of machine model are in order here. Allowing no work to be done on the input tape means that we may have space complexities that are less than the input length. This is essential since sets such as strings of the form 0^n1^n can be recognized in $\log_2 n$ space by counting the 0's and 1's in binary and then comparing the totals. Note also that since we are not concerned with the speed of computation, having several work tapes is not essential.

The machine models used in our examination of complexity have no restrictions as to number of symbols or number of tracks used per tape. This is intentional. Recalling the discussion of tracks and symbols when we studied computability we note that they are interchangeable in that a k symbol horizontal block of a tape can be written as a vertical block on k tracks. And k tracks can be represented on one track by expanding the machine's alphabet.

Thus we may read lots of symbols at once by placing them vertically in tracks instead of horizontally on a tape. We may also do lots of writes and moves on these columns of symbols. Since a similar idea is used in the proof of the next theorem, we shall present it with just the intuition used in a formal proof.

> **Theorem 3** (Linear Space Compression). *Anything that can be computed in s(n) space can also be computed in s(n)/k space for any constant k.*

> **Proof sketch**. Suppose that we had a tape with a workspace that was twelve squares long (we are not counting the endmarker) like that provided below.

#	a	b	c	d	e	f	g	h	i	j	k	l

> Suppose further that we wished to perform the same computation that we are about to do on that tape, but use a tape with a smaller workspace, for example, one that was four squares long or a third of the size of the original tape. Consider the tape shown below.

#	a	b	c	d	*
	e	f	g	h	
	i	j	k	l	

On this one, when we want to go from the square containing the 'd' to that containing the 'e' we see the right marker and then return to the left marker and switch to the middle track.

In this manner computations can always be performed upon tapes that are a fraction of the size of the original. The general algorithm for doing this is as follows for a machine M(x) that uses s(n) space for its computation.

n = the length of the input x
Lay off exactly s(n)/k squares on the work tape
Set up k rows on the work tape
Perform the computation M(x) in this space

There are two constraints that were omitted from the theorem so that it would be more readable. On is that s(n) must be at least $\log_2 n$ since we are to count up to n on the work tape before computing s(n). The other is that we must be able to lay out s(n)/k squares within that amount of tape.

We must mention another slightly picky point. In the last theorem there had to have been some sort of lower bound on space. Our machines do need at least one tape square for writing answers! So, when we talk of using s(n) space we really mean max(1, s(n)) space.

In the same vein, when we speak of t(n) time, we mean max(n+1, t(n)) time since in any nontrivial computation the machine must read its input and verify that it has indeed been read.

This brings up a topic that we shall mention and then leave to the interested reader. It is *real time computation*. By this, we mean that an answer must be presented *immediately* upon finishing the input stream. (For example, strings of the form 0^n1^n are real time recognizable on a 2-tape Turing machine.) In the sequel whenever we speak of O(n) time we mean O(kn) time or *linear time*, not real time. This is absolutely essential in our next theorem on linear time speedup. But first, some more notation.

Definition *The expression $\inf_{n\to\infty} f(n)$ denotes the limit of the greatest lower bound of f(n), f(n+1), f(n+2), ... as n goes to infinity.*

The primary use of this limit notation will be to compare time or space functions. Whenever we can say that

$$\inf_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

we know that the function f *grows faster* than the function g by more than a constant. Another way to say this is that for *every* constant k,

$$f(n) > k*g(n)$$

for all but a finite number of n. In other words, the limit (as n goes to infinity) of f(n) divided by g(n) cannot be bounded by a constant. Thus f *is not* O(g), but larger. For example, f(x) and g(x) could be x^3 and x^2, or even n and $\log_2 n$.

With that out of the way, we may present another version of the last theorem, this time with regard to time. Note that we must use our new *inf* notation to limit ourselves to machines that read their own input. This infers that if a machine does not run for at least on the order of n steps, then it is not reading its input and thus not computing anything of interest to us.

> **Theorem 4** (Linear Time Speedup). *Anything which can be computed in t(n) time can also be computed in t(n)/k time for any constant k if* $inf_{n\to\infty} t(n)/n = \infty$.

The proof of this theorem is left as an exercise since it is merely a more careful version of the linear space compression theorem. All we do is read several squares at once and then do several steps as one step. We note in passing though that if proven carefully the theorem holds for O(n) time also. (Recall that we mean linear time, not real time.)

So far, so good. A naive peek at the last two results might lead us to believe that we can compute things faster and faster if we use little tricks like doing several things at once! Of course we know that this is too good to be true. In fact, practice bears this out to some extent. We can often speed up our algorithms by a constant if we are clever. (And we did not need to do much mathematics to learn that!) Also, we know that there are *best algorithms* for much of the stuff we compute.

This brings up an interesting question. Does everything have a best or most efficient algorithm? Or at least a best algorithm up to a constant? A rather surprising result from abstract complexity theory tells us that there are some problems that have none. And, in fact, there are problems in which computation time can be cut by any amount one might wish.

> **Theorem 5** (Speedup). *For any recursive function g(n), there is a set A such that if the Turing machine M_i decides its membership then there is an equivalent machine M_k such that for all but a finite number of n: $T_i(n) \leq g(T_k(n))$.*

In other words, machine M_k runs as much faster than machine M_i as anyone might wish. Yes, we said that correctly! We can have M_k and M_i computing the same functions with $T_i(n)$ more than $[T_k(n)]^2$. Or even

$$T_i(n) \geq 2^{T_k(n)}$$

This is quite something! It is even rather strange if you think about it.

It means also that there are some problems that can never be computed in a most efficient manner. A strict interpretation of this result leads to an interesting yet rather bizarre corollary.

> **Corollary.** *There are computational problems such that given any program solving them on the world's fastest supercomputer, there is an equivalent program for a cheap programmable pocket calculator that runs faster!*

A close look at the speedup theorem tells us that the corollary is indeed true but must be read *very carefully*. Thoughts like this also lead to questions about these problems that have no best solution. We shall leave this topic with the reassuring comment that even though there are problems like that, they are so strange that most likely none will ever arise in a practical application.