Chapter 13

Measuring and Classifying Complexity

13.1. (a) Suppose $f(n) \leq C_0 h(n)$ for $n \geq n_0$ and $g(n) \leq C_1 k(n)$ for $n \geq n_1$. Then $f(n) + g(n) \leq C(h(n) + k(n))$ for $n \geq N$, where $C = \max(C_0, C_1)$ and $N = \max(n_0, n_1)$; also, for the same N, $f(n) * g(n) \leq D(h(n) * k(n))$ for $n \geq N$, where $D = C_0 * C_1$.

(c) This statement follows from the inequalities $\max(f, g) \leq f + g \leq 2\max(f, g)$.

13.2.

	f_1	f_2	f_3	f_4
f_1		yes	yes	yes
f_2	yes		yes	yes
f_3	no	no		no
f_4	no	no	yes	

For example, $f_3 \neq O(f_1)$: if $n > 1$, $f_3(n)/f_1(n) = n/(2\log n)$, and this ratio is unbounded. $f_4 = O(f_3)$: if $n > 1$,

$$f_4(n)/f_3(n) = \begin{cases} 2\log n/n & \text{if } n \text{ is even} \\ \log^2 n/n & \text{if } n \text{ is odd} \end{cases}$$

and in either case the ratio is bounded (in fact, approaches 0).

13.3. If $f(n) \leq Cp(n)$ for every $n \geq N_0$, then $f(n) \leq Cp(n) + M$ for every n, where $M = \max\{f(n) | n < N_0\}$.

13.4. It is clear that $n! < n^n$ if $n > 1$, and since $n^n = e^{n\log n}$ and $2^{2^n} = e^{2^n \log 2}$, $n^n < 2^{2^n}$. Thus it is sufficient to show that the growth rate of $n!$ is greater than exponential. But this is reasonably clear: if $n > a^2$, for example,

$$\begin{aligned} n!/a^n &= \{(1/a)(2/a)\ldots(a/a)\} \\ &\quad *\{((a+1)/a)((a+2)/a)\ldots((a^2-a)/a)\} \\ &\quad *\{((a^2-a+1)/a)((a^2-a+2)/a)\ldots(a^2/a)\} \\ &\quad *\{((a^2+1)/a)\ldots(n/a)\} \\ &= A * B * C * D \end{aligned}$$

The product $A * C$ is at least 1, $B > 1$, and it is obvious that D gets arbitrarily large as n gets larger.

13.5. (a) $2^{\sqrt{n}} = e^{\sqrt{n}\log 2}$ and $n^{\log n} = e^{\log^2 n}$. Since $\sqrt{n}\log 2 > \log^2 n$ for all large n, $2^{\sqrt{n}} > n^{\log n}$. It is sufficient to show that the growth rate of $n^{\log n}$ is greater than polynomial and that of $2^{\sqrt{n}}$ is less than exponential. The first is clear, since $a^n = e^{a\log n}$ and $\log^2 n$ grows faster than $a\log n$; the second is just as easy, and follows from comparing $\sqrt{n}\log 2$

to $n \log a$ for any fixed a.

(b) The function $\sqrt{n}$ has a larger growth rate than $\log^2(n)$. It follows that $2^{\sqrt{n}}$ has a larger growth rate than $n^{\log n}$.

13.6. $(\log n)^{\log n} = e^{\log n \log\log n}$. Since $\log\log n$ eventually gets large, this function has a larger growth rate than $e^{k \log n}$, for any k, and therefore has a larger growth rate than any polynomial. However, since $\log n \log\log n$ has a smaller grown rate than cn, for any positive constant c, this function has a smaller growth rate than any exponential function.

13.7. We wish to show that for any fixed k, and any $a > 1$, $n^k = O(a^n)$. (From this fact the theorem will follow, just as it did in the proof given in the chapter.)

Choose N_0 so that $((N_0+1)/N_0)^k \leq a$. Then if $n = N_0 + m$,

$$\begin{aligned} n^k &= (N_0+m)^k \\ &= N_0^k((N_0+1)/N_0)^k \ldots ((N_0+m)/(N_0+m-1))^k \\ &\leq N_0^k a^m = C a^{N_0+m} = C a^n \end{aligned}$$

where C is the constant N_0^k / a^{N_0}.

13.8. For simplicity, suppose the input string is $a^n = a^{2k+1}$. (Note that for input strings of odd length, the number of moves depends only on the length, not on the symbols themselves.) It is easy to check that $4k+3$ moves are required to transform the tape contents $\Delta \underline{a} a \ldots a$ to tape contents $\Delta A \underline{a} a \ldots a A$; $4(k-1)+3$ moves to transform this to $\Delta A A \underline{a} a \ldots a A A$; $\ldots$; and $4(1)+3$ to go to $\Delta A \ldots A \underline{a} A \ldots A$. Adding the single move at the beginning and the two additional moves before the machine crashes, we obtain

$$\begin{aligned} 3 + 4\sum_{i=1}^{k} i + 3k &= 3 + 4k(k+1)/2 + 3k \\ &= 3 + 4((n-1)/2)((n+1)/2)/2 + 3(n-1)/2 \\ &= 3 + (n^2-1)/2 + 3n/2 - 3/2 \\ &= (n^2+3n+2)/2 \end{aligned}$$

13.9(a) Suppose the input is $a^n = a^{2k}$. $4k+1$ moves are needed to go from $\Delta \underline{a} a^{2k-1}$ to $\Delta\Delta \underline{a} a^{2k-3}$; $4(k-1)+1$ to go to $\Delta\Delta\Delta \underline{a} a^{2k-5}$; $\ldots$; and $4(1)+1$ to go to $\Delta^{k+1} \underline{\Delta}$. Add one move at the beginning and at the end, and the result is

$$\begin{aligned} 2 + \sum_{i=1}^{k}(4i+1) &= 2 + 4k(k+1)/2 + k \\ &= 2k^2 + 3k + 2 \\ &= (n^2+3n+4)/2 \end{aligned}$$

Moreover, it can be checked that the same formula holds when n is odd.

(*b*) Suppose that the input is a^n. $2n+3$ moves are required to get from tape contents $\Delta\underline{a}a^{n-1}$ to tape contents $\Delta A\underline{a}a^{n-2}\Delta a$. Each additional such pass requires the same number of moves. The next-to-last ends with $\Delta A^{n-1}\underline{a}\Delta a^{n-1}$, and the last ends with $\Delta A^n\underline{\Delta}a^n$. $n+2$ moves are then required to finish up. Adding the one move at the beginning, we obtain

$$n(2n+3)+n+3 = 2n^2+4n+3$$

13.11. Saying that the problem is solvable is the same as saying that the language L of strings that represent instances of the problem for which the answer is yes is recursive. Let T be any TM computing χ_L, and let τ_T be its time complexity. Let us now consider a different encoding of instances. Take a symbol \$ not used in the original encodings; if x was the original encoding of an instance, then the new encoding will be $x\$^m$, where $m = \tau_T(|x|)$. Let L_1 be the new language corresponding to L. Then we may modify T to form a new TM T_1 computing χ_{L_1} as follows: T_1 simply erases all the \$'s and executes T. The time complexity of T_1 satisfies the inequality

$$\begin{aligned}\tau_{T_1}(|x|+m) &\leq 2(|x|+m+1)+\tau_T(|x|)\\ &\leq 2(|x|+m+1)+m\\ &\leq 3(|x|+m)+2\end{aligned}$$

which implies that the time complexity of T_1 is at most linear.

13.14. For a particular f, Theorem 13.3 says that the property of being in $Time(f)$ is a nontrivial language property. Therefore, by Rice's theorem, the problem is unsolvable.

13.18. Let T_1 and T_2 be TMs recognizing L_1 and L_2 in times f_1 and f_2, respectively. If we consider a 2-tape TM that executes T_1 and T_2 simultaneously, one on each tape, then aside from the time required to copy the input onto tape 2, the time required to recognize either the union or the intersection is no more than the maximum of the two functions. We can take both g and h to be the function $2n+2+\max(f_1(n), f_2(n))$.

13.19. In both cases, we can construct a TM so that at each step, every work tape contains an integer, in binary notation, no larger than the length of the input. (This is where the log term comes from: the number of digits required to represent n in binary.)

For palindromes, we could use three work tapes and proceed as follows. Begin by placing the integer 1 on tape 1 and the integer n on tape 2. (On the kth pass, these tapes have the numbers k and $n-k$, respectively.) On each pass, start at the position on the input tape specified by the first work tape, use the third tape to count from that point to the number on the second tape, and check that those two symbols of the input are the same. Then increment the number on the first tape, decrement the number on the second tape, erase the third tape, and start the next pass.

For balanced strings of parentheses, we can get by with one work tape. Process the input left-to-right, keeping track on the work tape of the difference between the numbers of right and left parentheses. This number is initially 0, is incremented each time a left

parenthesis is read and decremented each time a right parenthesis is read. If the number never goes below 0 and ends up at 0, the string is balanced.

13.20. The statement $f = o(g)$ does not follow. If we allow the functions to be nondecreasing, instead of strictly increasing, there is a straightforward way to construct a counterexample. Let $g(n) = n$ for every n. Let $f(0) = 0$, $f(1) = f(2)) = 1$, $f(3) = \ldots = f(9) = 3$, $f(10) = \ldots f(40) = 10$, $f(41) = \ldots = f(205) = 41$, etc. The pattern is that on the first interval where f is constant and nonzero, the ratio f/g decreases from 1 to 1/2, on the second such interval f/g decreases from 1 to 1/3, on the next such interval f/g decreases from 1 to 1/4, etc. Then $f = O(g)$, since $f(n) \leq g(n)$ for every n; $g \neq O(f)$, because for each $n > 0$ there is an i for which $g(i) = nf(i)$; and $f \neq o(g)$, because there are infinitely many n's for which $f(n) = g(n)$. Constructing a counterexample is a little harder if f and g are required to be increasing, since f is not allowed to be constant on a long interval. We can use the same general idea if we take $g(n) = 2^n$. $f(n)$ is still defined to be $g(n)$ at infinitely many different n's; instead of being constant on an interval, f will increase by 1 at each step; and the ith interval will be long enough so that the ratio f/g will decrease from 1 to some number less than or equal to $1/i$.

13.21. No. Consider the functions f and g defined as follows. $f(0) = g(0) = 1$; $f(1) = f(0) + 1$; $g(1) = 2f(1)$; $g(2) = g(1) + 1$; $f(2) = 3g(2)$; $f(3) = f(2) + 1$; $g(3) = 4f(3)$; ...

Then for any $n > 1$ there are values of j and k so that $f(j)/g(j) > n$ and $g(k)/f(k) > n$. It follows that $f \neq O(g)$ and $g \neq O(f)$.

13.22. One approach is to have the TM copy the input onto the second tape and return the tape heads to the beginning; have tape head 1 move two symbols for every one tape head 2 moves (rejecting if the length is not even), so that tape head 2 is now in the middle of the input; move tape head 1 back to the beginning; then compare the first half of tape 1 to the second half of tape 2, symbol by symbol.

13.23. For simplicity, we consider the case in which there is only one string x for which T makes more than $f(|x|)$ moves on input x. The general case follows by induction. Let $|x| = n$. We would like to modify T to obtain a TM T_1 so that T_1 recognizes L and $\tau_{T_1} \leq f$. One approach would be to have T_1 simply behave like a finite automaton reading the input, until it determines whether the input string is x; if it is, it erases the tape and leaves the appropriate output (1 if $x \in L$, 0 otherwise), and if it is not, it returns to the beginning of the tape and proceeds to execute T. The problem with this is that in the second case, it might require $f(m) + 2n + 2$ moves to process a string of length m, whereas we want the maximum to be $f(m)$—or more precisely, $\max(f(m), 2m + 2)$. We can use the same idea, though, just being more careful.

By using a larger alphabet, we can have T_1 start moving its tape head to the right, replacing the symbol σ on tape square i by a new symbol (σ, i), for $0 \leq i \leq n$. If it reaches the end of the input by this time (that is, if the input string is *any one* of the strings of length n or less), then T_1 erases the tape, leaves the appropriate output, and halts. Otherwise, the input string has length at least $n + 1$. In this case, there are exactly

s^n possibilities for the prefix of length n, where s is the size of the input alphabet. Let us consider a specific prefix α of length n. For an input string starting with prefix α, there are two things T might do: crash before ever moving its tape head to square $n+1$; or eventually move its tape head to square $n+1$, having first replaced the symbol on square i by σ_i for each i with $0 \leq i \leq n$. If T crashes before it reaches square $n+1$, T_1 then proceeds to erase the tape and print the output 0. Otherwise, T_1 proceeds with exactly the same computation T would make, starting just after the move on which T reaches square $n+1$ for the first time, *except* that in the subsequent computation, T_1 treats the symbol (σ, i) exactly the same as it would the symbol σ_i when it reads it. In other words, T_1 has effectively bypassed the preliminary steps of T, during which T has not yet reached square $n+1$, replacing these steps by the first $n+1$ moves of T_1. It has not lost any information by doing this, because of the way it has marked the first $n+1$ squares of the tape. The result is that T_1 is able to complete the computation in no more steps than T would have required. In particular, $\tau_{T_1}(m) \leq \max(f(m), 2m+2)$ for each m.

13.25. See the solution to Exercise 9.44. Although that exercise involves a 2-tape TM, the same argument applies to an arbitrary multitape TM.

13.26. As in the proof of Theorem 14.3, let

$$L = \{w \mid w = e(T'), \text{ where } T' \text{ crashes on } w \text{ within } f(|w|) \text{ moves}\}$$

Let T be any TM accepting L. Then there are infinitely other TMs that also accept L and have the same time complexity as T. (Simply modify T by adding any number of moves that are never actually executed.) Let T_1 be any such TM, and let $w = e(T_1)$. If T halts on w, then $w \in L$, since T accepts L. This means that T_1 crashes on input w. But this is a contradiction, since T_1 halts on the same strings as T. On the other hand, if T crashes on w within $f(|w|)$ moves, then the same is true of T_1, and thus $w = e(T_1) \in L$ by definition of L. This is also a contradiction, since T accepts L. The only possibility left is that T crashes on w and that it requires more than $f(|w|)$ moves for this to happen. Since T_1 can be one of infinitely many different TMs, we may conclude that $\tau_T(n) > f(n)$ for infinitely many n.

13.27. Let T be a TM computing f. Then let T_1 be a TM that, on any input x, changes it to $1^{|x|}$ and executes T on that string. Then the function τ_{T_1} is a step-counting function, since T_1 makes the same number of moves for every input string of a given length. Moreover, $\tau_{T_1}(n) > f(n)$, because T_1 requires at least $f(n)$ moves to write the output string $1^{f(n)}$.