Time Complexity

Definition. Let M be a TM that halts on all inputs. The *running time* (or *time complexity*) of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$ such that $f(n)$ is the maximum number of steps that M uses on any input string of length n.

Note: Since $f(n)$ is the maximum for *any* input of length n, this is sometimes also called the *worst-case* running time of M.

Example: Consider the following decider for $\{0^k1^k \mid k \geq 0\}$.

$M_1 =$ "On input string w:

1. Scan across the tape; if a 0 found to the right of a 1, *reject*.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If there are extra 0s left or extra 1s left, *reject*; otherwise, *accept*."

Exact Analysis of the Running Time of M_1 (Almost)

To determine f for this algorithm we need to be able to identify what input of length n gives rise to the maximum number of steps and then examine the running time for this particular input, and we need to do this for arbitrary n. While this is doable for this particular algorithm, we won't go to this trouble, especially since our ultimate goal is to be able to perform an *asymptotic analysis*, which is generally simpler.

Therefore we start with the following observations:

- Any input not of the form 0^*1^* will be rejected in stage 1 in no more than n steps.
- Therefore the only input causing the remaining stages to run is input of the form 0^*1^*, so only input of this form can give rise to the worst-case running time of this algorithm.

At this point, rather than determine the running time for all possible input strings of the form 0^i1^j, we focus solely on input of the form 0^k1^k, which will be accepted. For such input:

- Stage 1 will use n steps to examine every symbol in the input, and then another n steps are required to get back to the beginning of the tape to prepare for the subsequent stages. Thus this involves a total of $2n$ steps.
- During each iteration of the loop, stage 3 involves crossing off a 0 and locating its corresponding 1 to cross off. This takes $n/2$ steps for the rightward sweep and another $n/2 + 1$ steps to go back to the left to locate the next un-crossed-off 0, for a total of $n + 1$ steps.
- The loop iterates once for each 0, which means it iterates $n/2$ times. Thus the entire loop, stages 2 and 3, takes $(n/2) * (n + 1) = n(n + 1)/2$ steps. Note that the leftward pass during the last iteration can determine that there are no more 0s, which will then terminate the loop.
- Finally, stage 4 will take $n/2 + 1$ steps to sweep to the right to locate the blank following the end of the string, and during this sweep it will be discovered that there are no remaining 1s.

Therefore the overall running time for M_1 on an accepted string of length n is

$$2n + \frac{1}{2}n(n+1) + \frac{1}{2}n + 1 = \frac{1}{2}n^2 + 3n + 1.$$

Asymptotic Analysis and Notation

Definition. Given functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$, we say that

$$f(n) = O(g(n))$$

if there exist positive numbers c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Another way to express this is to say that $g(n)$ is an *asymptotic upper bound* for $f(n)$.

The idea is that, for sufficiently large values of n, the function g (when scaled by the constant c) always gives values no smaller than f does.

In practice: Starting with f, keep only its dominant (i.e., highest-order or, more generally, fastest-growing) term and ignore its coefficient, and use this for g.

Common shorthand: Replace f and g in this notation by the expressions in their dependent variable n that define them. For example, instead of referring to the function f defined by $f(n) = 4n^3 + 25n^2 - 6n + 7$, we simply say: the function $4n^3 + 25n^2 - 6n + 7$.

Examples:

- $4n^3 + 25n^2 - 6n + 7 = O(n^3)$
- $5n + 7n \log_2 n = O(n \log_2 n)$
- $2^n + 5n^{100} = O(2^n)$

Asymptotic Analysis and Notation (Continued)

Consider $f(n) = 7n^2 + 5n + 27$.

According to the informal description just given, we can write $f(n) = O(n^2)$.

Is this justified by the definition?

Want $c, n_0 > 0$ such that $7n^2 + 5n + 27 \leq cn^2$.

$c = 7$ (or anything less) won't work, but any $c > 7$ will. Choose $c = 8$.

Want $8n^2 \geq 7n^2 + 5n + 27$, or $n^2 \geq 5n + 27$, for sufficiently large n. Plot n^2 and $5n + 27$ on a graph and see that they cross when n is somewhere between 8 and 9. Thus if we choose $n_0 = 9$, this guarantees that $8n^2 \geq 7n^2 + 5n + 27$ for all $n \geq n_0$.

Underlying idea: eventually (i.e., for large enough n), highest-order or fast-growing term dominates.

Application to time complexity: want to examine how well an algorithm scales for large inputs, and determining its running time in $O(g(n))$ form captures the essence of this.

Getting Familiar with Asymptotic Notation

- $O(1)$ means bounded by a constant
- $O(n)$ means bounded by a linear function
- $O(n^2)$ means bounded by a quadratic function
- etc.

Some other facts: Let a_1 and a_2 be positive constants, and suppose $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Then

- $a_1 f_1(n) + a_2 f_2(n) = O(g_1(n) + g_2(n))$
- $f_1(n) f_2(n) = O(g_1(n) g_2(n))$

These justify certain algebraic manipulations involving O-notation as in this example:

$$O(n)O(n) + 5nO(1) + O(3n^2) = O(n^2) + O(n) + O(n^2) = O(n^2).$$

Asymptotic Time Complexity Analysis

Earlier we performed (almost) an exact analysis of the following decider for $\{0^k1^k \mid k \geq 0\}$:

$M_1 =$ "On input string w:
1. Scan across the tape; if a 0 found to the right of a 1, *reject*.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If there are extra 0s left or extra 1s left, *reject*; otherwise, *accept*."

Now we perform an asymptotic analysis of its time complexity instead.

- Stage 1 uses $O(n)$ steps to sweep right, then $O(n)$ steps to get back to the beginning of the tape, for a total of $O(n) + O(n) = O(n)$ steps.
- During each iteration of the loop, stage 3 uses $O(n)$ steps to move from each 0 to its corresponding 1, and another $O(n)$ steps to move back to the next un-crossed-out 0, if any. Thus stage 3 uses $O(n) + O(n) = O(n)$ steps.
- There are $O(n)$ 0s in the string, so the number of iterations of the loop is $O(n)$, making the overall running time for stages 2 and 3 $O(n)O(n) = O(n^2)$.
- Finally, stage 4 takes $O(n)$ steps.

Therefore the overall running time of this algorithm is $O(n) + O(n^2) + O(n) = O(n^2)$.

We say this is an $O(n^2)$-*time* (or *quadratic-time*) algorithm.

From now on, this is the only form of time complexity analysis we consider.

Can you think of a language that has a linear-time decider? a constant-time decider?

Asymptotic Notation with Logarithms and Exponential Functions

Since

$$\log_a n = (\log_a b) \log_b n$$

for any $a, b > 0$, base-a logarithms and base-b logarithms differ from each other by the constant factor $\log_a b$.

Thus, wherever a $\log_b n$ factor might appear in asymptotic notation, it doesn't matter what base b is used since constant factors are suppressed.

Standard convention: just write $\log n$, without explicitly identifying what base is used.

Examples:

- The decimal representation of an integer k uses $O(\log k)$ digits.
- The binary representation of an integer k uses $O(\log k)$ bits.

Clearly, if we wanted an exact count of the number of decimal digits for k we'd use base-10 logarithms, while base-2 logarithms are the right choice for the number of bits.

Suppose $f(n) = g(n)b^{h(n)}$ for some $b > 0$ and $g(n) = O(b^{h(n)})$. (The second condition just means that the $g(n)$ factor grows no faster than the exponential $b^{h(n)}$ factor.)

Then it is not hard to show that $f(n) = 2^{O(h(n))}$. One consequence of this is that exponentiation involving any base can be replaced by exponentiation using base 2 in any expression involving asymptotic notation.

Asymptotic Analysis of Another $\{0^k1^k \mid k \geq 0\}$ Decider

$M_2 =$ "On input string w:

1. Scan across the tape; if a 0 found to the right of a 1, *reject*.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking the parity of the number of 0s and 1s remaining. If it's odd, *reject*.
4. Scan across the tape, crossing off every other 0 and every other 1.
5. If there are extra 0s left or extra 1s left, *reject*; otherwise, *accept*."

Unlike the earlier algorithm, it's certainly not obvious that this algorithm does what it's supposed to do. See the discussion on p. 252 of Sipser for an argument justifying its correctness. Here we just examine its asymptotic running time:

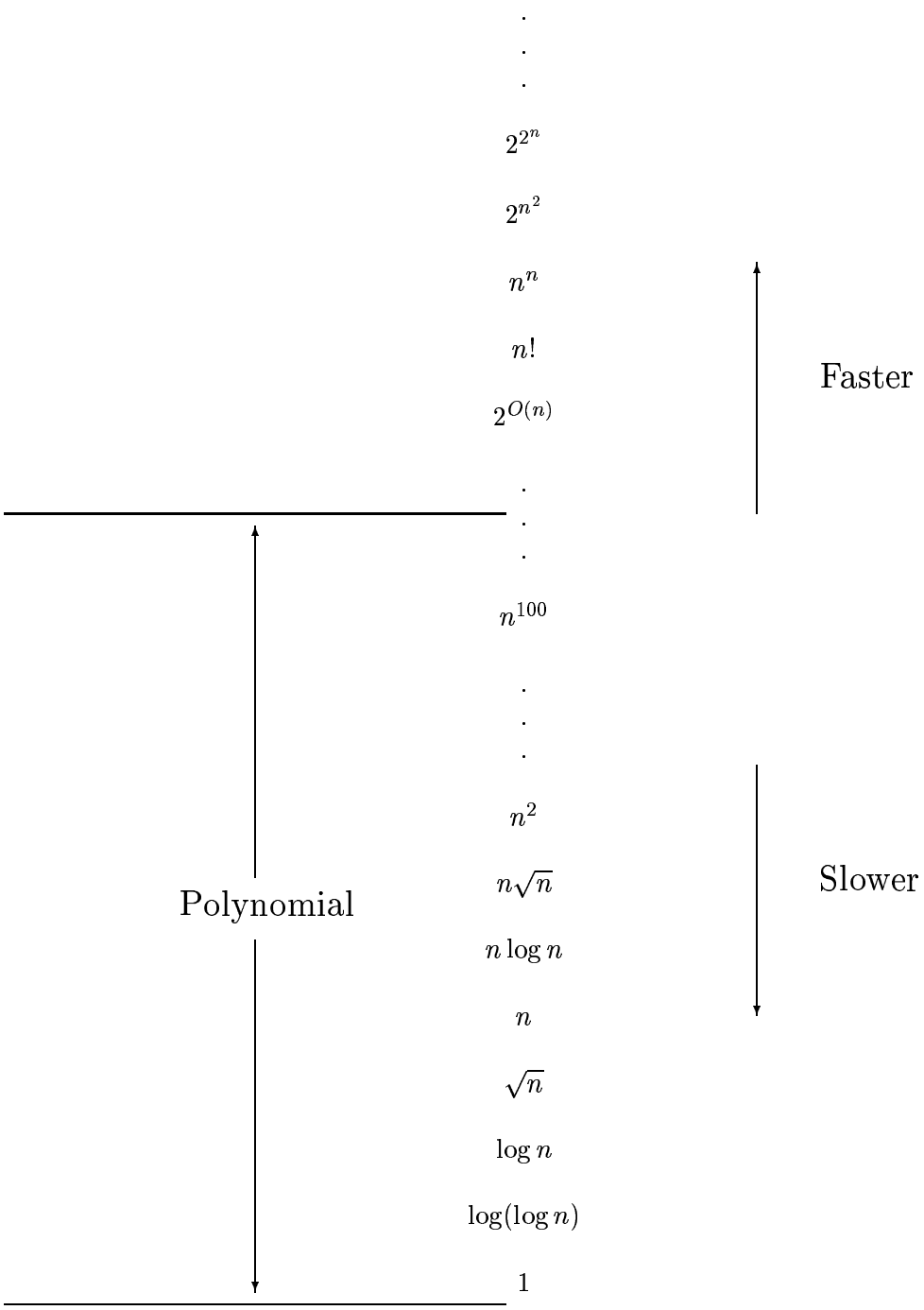
- As before, stage 1 uses $O(n)$ steps.
- Stage 3 clearly uses $O(n)$ steps, as does stage 4, so the body of the loop uses $O(n)$ steps, just as before.
- What's different is how many iterations of the loop occur. Since essentially half of the unmarked cells get marked on each iteration, the number of iterations is $O(\log_2 n) = O(\log n)$; input twice as long only requires one more iteration, for example. Thus stages 2-4 take $O(n \log n)$ steps.
- And finally, stage 5 requires $O(n)$ steps.

Therefore the entire algorithm has asymptotic time complexity

$$O(n) + O(n \log n) + O(n) = O(n \log n).$$

Since the rate of growth of $n \log n$ is strictly slower than n^2, M_2 is an asymptotically more efficient decider for this language than M_1.

Rate of Growth of Various Functions



Benefits of Asymptotic Analysis

- Can use approximate analysis – often much simpler to compute.
- Not highly sensitive to certain details of the computation model used:
  - What if some steps are faster than others on a particular computer?
  - What if the relative speed of certain steps vary across computers?
  - What if a different size alphabet is used?
- Identifies dominant contributions to the running time.
- Addresses in a concise but informative way the issue of how running time scales with ever-increasing input size.