A 2-tape decider for $\{0^k 1^k \mid k \geq 0\}$

$M_3 =$ "On input string w:

1. Scan across tape 1; if a 0 found to the right of a 1, *reject*.
2. Scan across the 0s on tape 1, copying each to tape 2.
3. Continue scanning across tape 1. For each 1 read on tape 1, move right along tape 2.
4. If the 0s run out before the 1s, or the 1s run out before the 0s, *reject*; otherwise, *accept*."

This is easy to analyze exactly: It first uses n steps to scan tape 1 in stage 1 and then another n steps to return to the beginning of the tape. Then the remaining stages require just one more scan from left to right, involving n steps. Thus this algorithm runs in linear ($O(n)$) time.

Not surprisingly, being able to take advantage of more tapes gives a more efficient algorithm. (In fact, this is an asymptotically optimal running time for deciding this language since any decider for this language must use $O(n)$ time just to read the entire input string.)

Multi-Tape TM Running Times vs. Single-Tape TM Running Times

Theorem. Suppose a multi-tape TM M has running time $t(n) \geq n$ on any input of size n. Then a single-tape TM S simulating M has running time $O(t^2(n))$.

Proof.

- We assume S uses the simulation strategy described earlier in the course.
- To determine an upper bound on the length of working tape used by S, assume that every move on all k of M's tapes is to the right.
- This uses $kO(t(n))$ cells across all k tapes. Since k is a constant independent of n, the number of cells used on all tapes is $kO(t(n)) = O(t(n))$.
- Thus $O(t(n))$ cells are used on S's tape, plus a few more for the separators, but there are only $O(k) = O(1)$ of these, and $O(t(n)) + O(1) = O(t(n))$.
- For every simulated move, S scans its tape twice, making $O(t(n)) + O(t(n)) = O(t(n))$ moves to do this.
- Thus each of M's $O(t(n))$ moves takes $O(t(n))$ moves to simulate in S.
- There is also an initial setup stage that S performs to place a separator surrounding each of its simulated tapes as well as a single blank on each "virtual tape" except the first. This requires $O(n + 2k) = O(n)$ moves.
- Therefore the entire simulation takes $O(t(n))O(t(n)) + O(n) = O(t^2(n)) + O(n)$ moves in S.
- Since we're assuming $t(n) \geq n$, this implies $t^2(n) \geq t(n) \geq n$, so the $O(n)$ term can be ignored, yielding $O(t^2(n))$ as the asymptotic upper bound on the running time of S.

This result says that going from a multi-tape to a single-tape model causes, at worst, a quadratic slowdown in the running time.

The Class P

Definition. A *polynomial-time* (*polytime*, for short) algorithm is one whose running time $f(n)$ satisfies $f(n) = O(n^k)$ for some $k \geq 0$.

Note: This does not mean the running time $f(n)$ *is* a polynomial, just that $f(n)$ *is bounded above by* a polynomial.

E.g., algorithms having these time complexities all qualify as polytime algorithms by this definition:

- n^5 $(= O(n^5))$
- $n \log n$ $(= O(n^2))$
- $\sqrt{n}$ $(= O(n))$

A Really Important Definition:

$$\mathrm{P} = \{L \mid L \text{ is a language having a (deterministic) polytime decider}\}$$

Note that we have not specified how many tapes the decider is allowed to have. The following result shows that it doesn't really matter:

Theorem. If L can be decided in polynomial time on a multi-tape TM, then it can be decided in polynomial time on a single-tape TM.

Proof. Since we have just shown that any $t(n)$-time algorithm on a multi-tape TM can be simulated in $O(t^2(n))$ time on a single-tape TM, any $O(n^k)$-time multi-tape decider can be simulated in $O(n^{2k})$ time on a single-tape TM.

Time Complexity of a Nondeterministic TM

Definition. Let N be an NTM that is a decider. Recall that this means all of its computation branches halt on all inputs. The *running time* of N is $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps N uses on any branch of its computation tree on any input of length n.

Note that this definition is consistent with the idea that all branches of a nondeterministic machine's computation tree should be considered to run in parallel. Thus no time penalty is imposed on the "branchiness" of the computation tree, only on its height.

Nondeterministic TM vs. Deterministic TM

Theorem. Suppose a nondeterministic TM N has running time $t(n) \geq n$ on any input of size n. Then a single-tape deterministic TM D simulating N has running time $2^{O(t(n))}$.

Proof. On any input of length n, every branch of N's computation tree has length $\leq t(n)$. Every node has at most b children, where b is the maximum number of choices given by N's transition function. Consider a 3-tape (deterministic) TM M simulating N as described earlier in this course. Recall that this simulation uses a breadth-first strategy and proceeds by restarting at the root of the tree each time and traversing down one level deeper than on the previous iteration. To get an upper bound on the total number of steps taken by M at each stage of the simulation, assume that every node has exactly b children and every computation branch has length equal to $t(n)$.

For such a tree the number of steps taken to explore the tree in any iteration down to a given level is equal to the total number of non-root nodes down to that level. The number of nodes at level l is b^l, so the total number of non-root nodes down to level l is

$$b + b^2 + b^3 + \ldots + b^l.$$

Thus the total number of steps taken for all the iterations in such a tree is

$$\begin{aligned}
&b+ \\
&b + b^2+ \\
&b + b^2 + b^3+ \\
&\ldots \\
&b + b^2 + b^3 + \ldots + b^{t(n)},
\end{aligned}$$

where the overall summation has been arranged so that each row represents the total number of steps taken to get from the root down to all nodes at a given level. The entire summation then represents the total number of steps taken by the time the simulation gets to all the leaves at level $t(n)$.

We can obviously get an upper bound on this by replacing every row in the above arrangement of the summation by the last row, yielding

$$\begin{aligned}
&b + b^2 + b^3 + \ldots + b^{t(n)}+ \\
&b + b^2 + b^3 + \ldots + b^{t(n)}+ \\
&b + b^2 + b^3 + \ldots + b^{t(n)}+ \\
&\ldots \\
&b + b^2 + b^3 + \ldots + b^{t(n)},
\end{aligned}$$

which has $t(n)$ rows. In turn, we can clearly get an upper bound for this by replacing every term (in every row) by $b^{t(n)}$, and the result looks like[1]

[1] While it might appear that this would give a much too loose bound, it's clear from looking at the final result that it yields exactly the same asymptotic bound as if the entire sum were replaced by the single term $b^{t(n)}$. While this may seem surprising, the reason for this is that in such a tree, the number of leaves dominates the total number of nodes, even when all earlier nodes in the tree are counted multiple times as in the analysis here.

Nondeterministic TM vs. Deterministic TM (Continued)

$$
\begin{array}{l}
b^{t(n)} + b^{t(n)} + b^{t(n)} + \ldots + b^{t(n)} + \\
b^{t(n)} + b^{t(n)} + b^{t(n)} + \ldots + b^{t(n)} + \\
b^{t(n)} + b^{t(n)} + b^{t(n)} + \ldots + b^{t(n)} + \\
\ldots \\
b^{t(n)} + b^{t(n)} + b^{t(n)} + \ldots + b^{t(n)},
\end{array}
$$

where each row has $t(n)$ terms and there are $t(n)$ rows, making a grand total of $t^2(n)$ identical $b^{t(n)}$ terms. Thus this upper bound is

$$
\begin{aligned}
t^2(n) b^{t(n)} &= 2^{2\log_2 t(n)} 2^{(\log_2 b) t(n)} \\
&= 2^{2\log_2 t(n) + (\log_2 b) t(n)} \\
&= 2^{O(t(n))}
\end{aligned}
$$

since $\log_2 t(n) = O(t(n))$ and $\log_2 b$ is a constant.

This is the asymptotic upper bound on the running time of the 3-tape deterministic TM M simulating the NTM N. This 3-tape machine can be simulated in turn by the single-tape (deterministic) TM D in time $(2^{O(t(n))})^2 = 2^{2O(t(n))} = 2^{O(t(n))}$.

This result says that going from a nondeterministic model to a deterministic model causes, at worst, an exponential slowdown. (While this theorem only considers upper bounds, it is not hard to give NTM examples where the deterministic TM simulation does, indeed, have exponential running time.)

Multi-Tape NTM vs. Single-Tape TM

Theorem. Suppose a multi-tape nondeterministic TM N has running time $t(n) \geq n$ on any input of size n. Then a single-tape deterministic TM D simulating N has running time $2^{O(t(n))}$.

Proof. Suppose N has k tapes. We first simulate N on a $(k+2)$-tape deterministic TM M. The analysis for this case is identical to that in the proof of the previous theorem, in which we assumed N had only one tape. The result is that the running time of M is $2^{O(t(n))}$ once again. When we simulate M using the single-tape machine D, the running time of D is then $(2^{O(t(n))})^2 = 2^{O(t(n))}$, just as before.

The Class NP

Definition. A *nondeterministic polynomial-time* (*nondeterministic polytime*, for short) algorithm is one whose running time $f(n)$ on an NTM satisfies $f(n) = O(n^k)$ for some $k \geq 0$.

Another Really Important Definition:

$$\text{NP} = \{L \mid L \text{ is a language having a nondeterministic polytime decider}\}$$

Note that we have not specified how many tapes the nondeterministic decider is allowed to have. The following results show that it doesn't really matter:

Theorem. Any multi-tape NTM can be simulated with at most quadratic slowdown on a single-tape NTM.

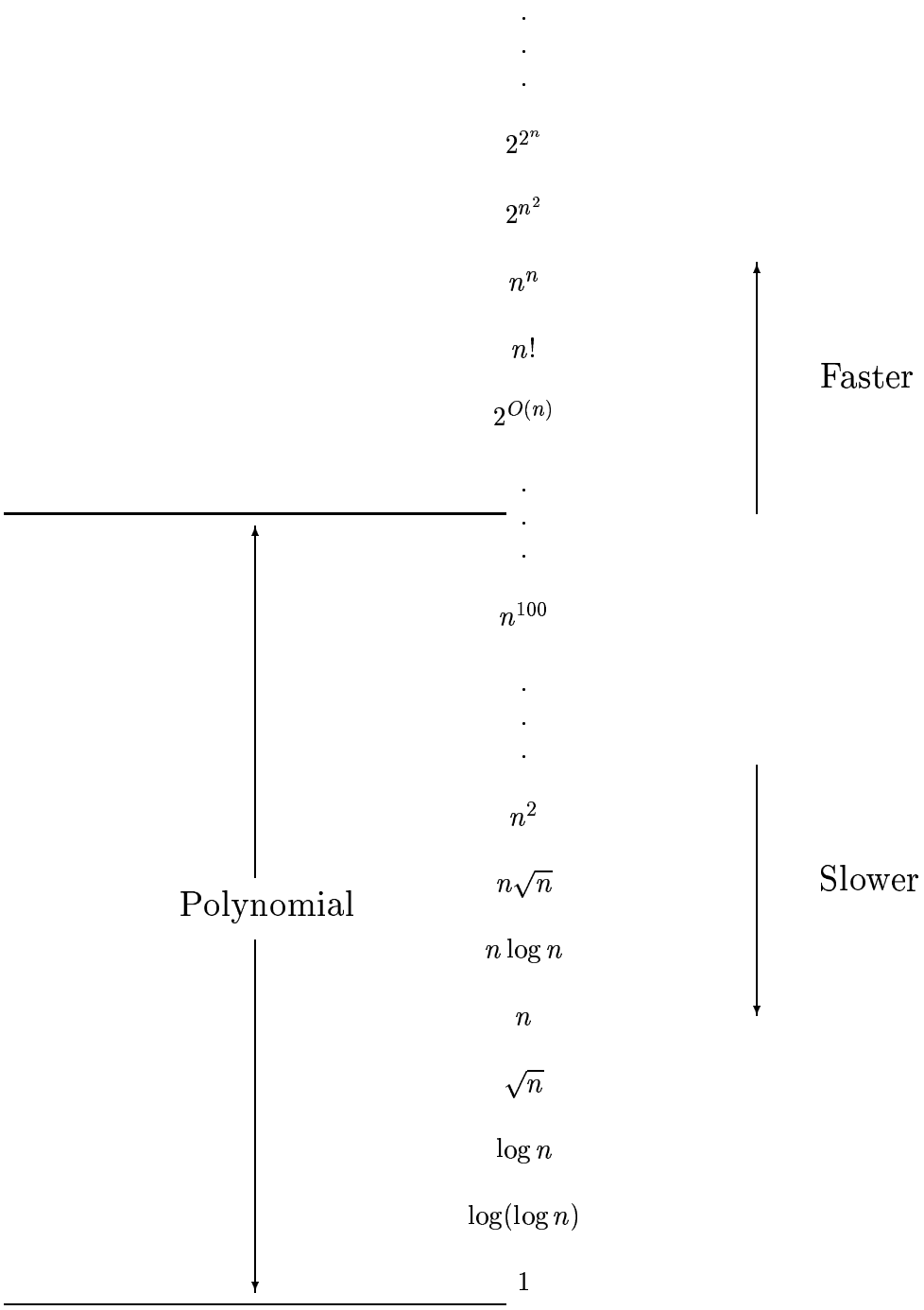
We don't bother to give a complete proof here, but we note that this essentially follows from all the earlier results because every branch of the nondeterminism can be simulated on a single-tape machine using the simulation strategy described for deterministic machines and because the simulation of each such branch suffers at most a quadratic slowdown on the single-tape machine.

Corollary. If L can be decided in polynomial time on a multi-tape NTM, then it can be decided in polynomial time on a single-tape NTM.

Note that since deterministic TMs are just a special case of nondeterministic TMs, any language in P automatically belongs to NP as well. That is,

$$\text{P} \subseteq \text{NP}.$$

Rate of Growth of Various Functions



Why P and NP Are Important

Importance of P:

- The definition of P is robust against many variations:
  - the details of the machine model (e.g., single-tape TM, multi-tape TM, even more realistic models using RAM, multiprocessor architectures, or other features of modern computers)
  - reasonable variations in how input is encoded
  - reasonable variations in how input size is measured
- If no polytime algorithm exists for a problem, there's no hope of solving it exactly for realistic-size instances.
- When a polytime algorithm exists for a problem, it's because of some insight into the problem. Brute-force approaches to combinatorial problems never yield polytime algorithms.

Importance of NP:

- The definition of NP is robust against the same variations listed above.
- Many problems of practical interest are in NP.
- The simulation of an NTM with a TM only yields algorithms with at least exponential running time, in general. Since such algorithms are not practical for realistic-size problems, finding practical algorithms for such problems is not straightforward and is therefore highly challenging.
- NP-completeness and the open $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ question.

For either case, the fact that polynomials are closed under various operations like

- addition
- multiplication
- composition

implies that algorithms built in various ways out of polytime algorithms will themselves be polytime. This applies to such constructs as

- running a sequence of polytime algorithms
- iterating a polytime algorithm a polynomial number of times
- using the output of one polytime algorithm as input to another

Measuring Input Size

There are a variety of ways to encode objects as strings for input to deciders. If the length of the input string depends on our choice of encoding, how does the running time function change?

The answer is that, while it may change, if an algorithm is polytime for any "reasonable" choice of encoding, it will remain a polytime algorithm when the input is re-encoded in any other way.

Example of re-encoding: changing the alphabet (as long as the alphabets involved have at least 2 symbols).

- E.g., think of using an alphabet consisting of all alphanumeric characters on a standard computer keyboard, then think of how strings of these symbols can be re-encoded into the alphabet $\{0, 1\}$ using ASCII codes.
- This may lead to rescaling of the size of the input, but converting between different such encodings can be done in polynomial time, so the overall effect is that a polytime algorithm when one alphabet is used gives rise to a polytime algorithm when any other alphabet is used.

Example of an "unreasonable" encoding: unary encoding of numbers.

- It's easy to describe an $O(k^2)$-time 2-tape TM algorithm to decide whether a number k is prime using unary encoding of k.
- But there's an exponentially more economical encoding of numbers using radix-b representation for $b > 1$ (e.g., binary or decimal).
- When one of these is used, the number k being tested can be encoded as a string of length essentially equal to $n = \log_b k$. In terms of this input size, this $O(k^2)$-time algorithm is actually a $O(b^{2n})$-time algorithm and therefore *not* a polytime algorithm.

Another example: It is often convenient not to have to identify input size precisely in terms of length of the input string. Consider languages representing decision problems on graphs. Possible ways to measure the size of a graph $G = (V, E)$:

- number of nodes, $|V|$
- number of edges, $|E|$
- length of its encoding as a list of nodes and a list of edges
- length of its encoding as an adjacency matrix ((i, j) entry is 1 if there's an edge from node i to node j and 0 if there's not)

These are polynomially related to each other:

- $|E| = O(|V|^2)$
- length of encoding as a list of nodes and edges $= O(|V|^2 \log |V|)$
- length of encoding as adjacency matrix $= O(|V|^2)$

An algorithm is polytime in input length using any reasonable encoding of the input iff it's polytime in any other way of measuring input size that is polynomially related to this input length.