

Example computation

Consider the decidable language $A = \{0^n 1^n \mid n \geq 0\}$ and the following TM M_1 deciding A :

$M_1 =$ "On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1
2. Repeat as long as both 0-s and 1-s remain of the tape:
 - (a) Scan across the tape, crossing off a single 0 and a single 1
3. If 0-s still remain after all 1-s have been crossed off, or if 1-s still remain after all 0-s have been crossed off *reject*. Otherwise, if neither 0-s nor 1-s remain of the tape, *accept*.

22c:131 Limits of Computation

Hantao Zhang

<http://www.cs.uiowa.edu/~hzhang/c131>

The University of Iowa
Department of Computer Science

Time Complexity

Simplifying Conventions

1. The running time of an algorithm is a function of the length of the string representing the input on the algorithm
2. In *worst-case-analysis* we consider the longest running time of all inputs of a particular length
3. In *average-case analysis* we consider the average of all the running times of inputs of a particular length

Analyzing an Algorithm

How much time does take M_1 to decide A ?

It is

- Number of steps the TM has (this may depends on several parameters).

Big-O and small-O notation

- The exact running time of an algorithm may be a complex-expression. Therefore usually this is just estimated.
- One convenient form of the estimation is so called *asymptotic analysis* which determines the running time of the algorithm on large inputs.
- In the asymptotic analysis one may consider only the highest order term of the expression for the running time of the algorithm, disregarding both the coefficient of that term and any lower order terms.
- This is valid because the value of the highest order term dominates the value of other terms on large inputs.

Time Complexity of a TM

Let M be a deterministic Turing machine that halts on all inputs. The *running time* or *time complexity* of M is a function $f : \mathcal{N} \rightarrow \mathcal{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of length n

- If $f(n)$ is the running time of M , we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine.
- Customarily n represents the length of the input

Formally

Let f and g be functions, $f, g : \mathcal{N} \rightarrow \mathcal{R}$. We say that $f(n) = \mathcal{O}(g(n))$ if positive integers c and n_0 exist such that $\forall n \geq n_0, f(n) \leq cg(n)$

When $f(n) = \mathcal{O}(g(n))$ we say that $g(n)$ is an *upper bound* for $f(n)$; more precisely, $g(n)$ is an *asymptotic upper bound* for $f(n)$, which emphasizes the suppression of constant factors.

Examples

Consider the function $f(n) = 6n^3 + 2n^2 + 10n + 100$.

- Disregarding the coefficient 6, we say that f is asymptotically at most n^3
- The asymptotic notation, or big-O notation for describing the estimation defined by $f(n)$ is $f(n) = \mathcal{O}(n^3)$

Examples

1. When we use logarithms, because $\log_b n = \log_2 n / \log_2 b$, the base needs not be specified. Thus, if

$$f_2(n) = 3n \log_2(n) + 5n \log_2(\log_2(n)) + 2$$

we have $f_2(n) = \mathcal{O}(n \log n)$

Intuitively:

- $f(n) = \mathcal{O}(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor.
- One may think of \mathcal{O} as representing a constant
- In practice most functions f encountered in algorithm analysis have an obvious highest order term $h(n)$. In that case $f(n) = \mathcal{O}(h(n))$

Polynomial and Exponential Bounds

- Bounds of the form n^c for $c > 0$ are called polynomial
- Bounds of the form a^{cn^δ} , for $a > 1$, $c, \delta > 0$, are called exponential bounds

Expressions of big-O

- Consider $f(n) = \mathcal{O}(n^2) + n$, where each occurrence of \mathcal{O} represents a different suppressed constant. Because $\mathcal{O}(n^2)$ dominates $\mathcal{O}(n)$, $f(n) = \mathcal{O}(n^2)$
- When \mathcal{O} occurs in the exponent, as in $f(n) = 2^{\mathcal{O}(n)}$, the same idea applies, i.e., $f(n) = \mathcal{O}(2^{cn})$ for some constant c . However, if we also have $g(n) = \mathcal{O}(n^2)$, it cannot be concluded that $f(n) = \mathcal{O}(g(n))$.
- For expressions of the form $f(n) = 2^{\mathcal{O}(\log n)}$, using the identity $n = 2^{\log_2 n}$, i.e., $n^c = 2^{c \log_2 n}$ we can see that $2^{\mathcal{O}(\log n)}$ represents an upper bound of n^c for some c .
- Because $\mathcal{O}(1)$ represents a value that is never more than a constant, $n^{\mathcal{O}(1)}$ represents the value n^c for some c .

Examples

Check that:

1. $\sqrt{n} = o(n)$
2. $n = o(n \log(\log(n)))$
3. $n \log(\log(n)) = o(n \log(n))$
4. $n \log(n) = o(n^2)$
5. $n^2 = o(n^3)$
6. $f(n)$ is never $o(f(n))$

Small-O notation

- Big-O notation says that a function is asymptotically *no more than* another function
- Small-O notation says that a function is asymptotically *less than* another function
- Formally: for $f, g : \mathcal{N} \rightarrow \mathcal{R}$, we say that $f(n) = o(g(n))$ if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$. That is, $f(n) = o(g(n))$ means that for **any real** $c > 0$ there exists n_0 such that $f(n) < cg(n)$ for all $n > n_0$

On input of length n :

Consider each of the three stages separately:

1. In stage 1 the machine scans the tape in n steps to verify that the input is of the form 0^*1^* ; repositioning the tape at the left end uses another n steps. So, the total number of steps is $2n$, i.e., $\mathcal{O}(n)$.
2. Each scan of the tape in stage 2 and 2(a) is performed in $\mathcal{O}(n)$. Because each scan crosses off a 0 and a 1, at most $n/2$ scans occur. I.e., the total number of steps is $(n/2)\mathcal{O}(n) = \mathcal{O}(n^2)$
3. In stage 3 the machine makes a single scan to decide whether to accept or reject taking $\mathcal{O}(n)$ steps

Total number of steps is $\mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$

Analyzing Algorithms

Consider the TM algorithm M_1 that decides the language $A = \{0^n 1^n \mid n \geq 0\}$:

$M_1 =$ "On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1
2. Repeat as long as both 0-s and 1-s remain of the tape:
 - (a) Scan across the tape, crossing off a single 0 and a single 1
3. If 0-s still remain after all 1-s have been crossed off, or is 1-s still remain after all 0-s have been crossed off *reject*. Otherwise, if neither 0-s nor 1-s remain of the tape, *accept*.

M_2 a $\text{TIME}(n \log n)$ TM

M_2 = "On input string w :

1. Scan across the tape and *reject* if a 0 is found to the right of a 1
2. Repeat as long as some 0's and some 1's remain on the tape:
 - (a) Scan across the tape, checking whether the total number of 0's and 1's remaining on the tape is even or odd. If it is odd *reject*.
 - (b) Scan again across the tape, crossing every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1
3. If no 0's and no 1's remain of the tape, *accept*; otherwise *reject*

Notations

Let $t : \mathcal{N} \rightarrow \mathcal{N}$ be a function. The time complexity class $\text{TIME}(t(n))$ is the collection of languages that are decidable by an $\mathcal{O}(t(n))$ time TM.

- Since language $A = \{0^n 1^n \mid n \geq 0\}$ is decided by M_1 in $\mathcal{O}(n^2)$ steps, $A \in \text{TIME}(n^2)$.
- $\text{TIME}(n^2)$ contains all languages decidable in $\mathcal{O}(n^2)$ time.
- Is there a TM that decides A asymptotically faster? I.e., is A in $\text{TIME}(t(n))$ for $t(n) = o(n^2)$? One can cross 2 0's and 2 1's in stage 2 which cuts the number of scans by half but running time does not change.

Deciding A in $\mathcal{O}(n)$

The following two-tape TM M_3 decides A in linear time, i.e., in $\mathcal{O}(n)$ time

M_3 = "On input string w on tape 1:

1. Scan across the tape 1 and reject if a 0 is found to the right of a 1
2. Scan across the 0's on tape 1 until the first 1; at the same time copy the 0's on tape 2.
3. Scan across the 1's on tape 1 until the end of the tape is discovered. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0's are crossed off before all 1 are read *reject*
4. If all 0's have been crossed off *accept*; if any 0's remain *reject*

Analysis

1. Stage 1 takes $\mathcal{O}(n)$ steps
2. Stage 2 takes $(1 + \log_2 n)\mathcal{O}(n) = \mathcal{O}(n \log_2 n)$ steps
3. Stage 3 takes $\mathcal{O}(n)$ steps

Total number of steps is asymptotically $\mathcal{O}(n \log_2 n)$, that is, $A \in \text{TIME}(n \log n)$.

Note: any language that can be decided in $o(n \log n)$ on a single-tape TM is regular; since A is not regular no faster algorithm performed by a single-tape TM exists to decide it.

Observations

There is an important difference between complexity theory and computability theory:

- The Church-Turing thesis implies that all reasonable models of computation are equivalent (i.e., for each model C there is an equivalent model C').
- In complexity theory the choice of model affects the time complexity of the language decided by that model (i.e., languages decidable by model C in linear time are not necessarily decidable in linear time by the equivalent model C').

Conclusions

- M_1 using one tape decides A in $\mathcal{O}(n^2)$
- M_2 using one tape decides A in $\mathcal{O}(n \log n)$
- M_3 using two tapes decides A in $\mathcal{O}(n)$

Conclusion: time complexity of A on one-tape TM is $\mathcal{O}(n \log n)$; time complexity of A on two-tape TM is $\mathcal{O}(n)$

Theorem 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $\mathcal{O}(t^2(n))$ time single-tape Turing machine.

Proof idea: We know how to convert a multitape TM M into a single-tape TM S that simulates M . We show that each step of M can be simulated by S in time $\mathcal{O}(t(n))$. Since there are $\mathcal{O}(t(n))$ steps performed by M there will be $\mathcal{O}(t^2(n))$ steps performed by S

Complexity Relations

Complexity relations among models show how choice of computational model can affect the time complexity of languages. For that we consider three computation models:

- Single-tape Turing machine
- Multiple-tape Turing machine
- Nondeterministic Turing machine

Analyzing S

- For each step of M , S makes two passes over the active portion of its tape: first to obtain the info necessary to determine the next move and second to carry out the move.
- The upper-bound of the length of the active portion of S 's tape is the sum of the length of active portions of M 's k tapes, which are bounded by $t(n)$, i.e., upper-bound of the active portion of S 's tape is $\mathcal{O}(t(n))$.

Proof of Theorem 7.8

For M a k -tape TM that runs in $t(n)$ time, the single-tape TM S operates as follows:

1. Initially S puts its tape into the format that represents all the k -tapes of M and then simulates M
2. To simulate one step of M , S scan all the info stored on its tape to determine the symbol under M 's heads.
3. Then S makes another pass over its tape to update the tape contents and head positions.
4. If one on M 's heads moves rightward onto a previously unread position on its tape, S must increase the amount of space by shifting a portion of its tape one cell to the right

Corollary

Let $t(n)$ and $s(n)$ be functions, where $t(n), s(n) \geq n$. If a multitape Turing machine takes $t(n)$ time and $s(n)$ space for any input of length n , then there is an equivalent $\mathcal{O}(t(n)s(n))$ time single-tape Turing machine.

Simulation time

- To simulate each of M 's steps, S performs two scans, each using $\mathcal{O}(t(n))$ time, i.e., one step of M is simulated by S in $\mathcal{O}(t(n))$ time.
- Since by assumption M perform $t(n)$ steps, total time taken by S to simulate M is $\mathcal{O}(t(n))$ taken by the initial step plus $t(n) \mathcal{O}(t(n))$ to perform the simulation, i.e., time complexity of S is $\mathcal{O}(t^2(n))$

Deterministic and nondeterministic TM

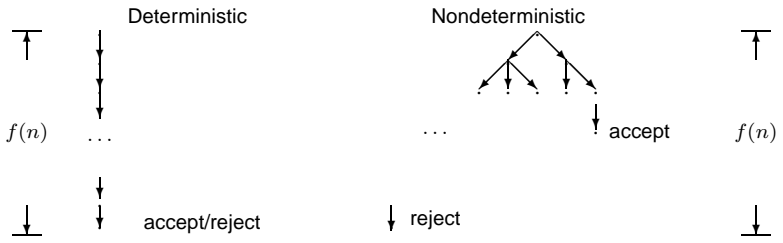


Figure 1: Measuring deterministic and nondeterministic time

Nondeterministic TM

Let N be a nondeterministic TM that is a decider. The running time of N is the function $f : \mathcal{N} \rightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N makes on any branch of its computation, on any input of length n , as shown in Figure 1

Theorem 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{\mathcal{O}(t(n))}$ time deterministic single-tape Turing machine.

Proof idea: Let N be a nondeterministic TM running in time $t(n)$. We construct a deterministic TM D that simulates N by searching N 's nondeterministic computation tree

Note

- Definition of the running time of a nondeterministic TM is not intended to correspond to any real-world computing device.
- The running time of a nondeterministic TM is a useful abstraction that assists in characterizing the complexity of an important class of computational problems.

The Simulation

1. Visit all nodes at depth d of the computation tree before going to the depth $d + 1$, starting with the root; total number of nodes is less than twice the number of leaves, i.e., this is bound by $\mathcal{O}(b^{t(n)})$.
2. Time for starting from the root and traveling down to a node is $\mathcal{O}(t(n))$
3. Therefore the running time of D is $\mathcal{O}(b^{t(n)}) = 2^{\mathcal{O}(t(n))}$.

Note: TM D has three tapes. Converting D to a single-tape, by previous theorem, the running time at most squares.

Thus we have: $(2^{\mathcal{O}(t(n))})^2 = 2^{\mathcal{O}(2t(n))} = 2^{\mathcal{O}(t(n))}$.

Proof

Computation tree:

1. On input of length n , every branch of N 's nondeterministic computation tree has a length at most $t(n)$
2. Every node in the tree can have at most b children where b is the maximum number of legal choices given by N 's transition function
3. Hence the total number of leaves in the tree is at most $b^{t(n)}$.

Summary

Let $t : \mathcal{N} \rightarrow \mathcal{N}$ be a function.

The time complexity class $\text{TIME}(t(n))$ is the collection of languages that are decidable by an $\mathcal{O}(t(n))$ time TM.

The time complexity class $\text{NTIME}(t(n))$ is the collection of languages that are decidable by an $\mathcal{O}(t(n))$ time NTM.

$\text{TIME}(t(n)) \subset \text{NTIME}(t(n)) \subset \text{TIME}(2^{ct(n)})$ for some constant c .