CS301: Computability and Complexity Theory (CC)

Lecture 9: Class P

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

December 8, 2023

Table of contents

Section 1

Previously on CS301

Measuring Complexity

Definition

Let f and g be functions $f, g : N \to R^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** for $f(n)$, or more precisely, that $g(n)$ is an **asymptotic upper bound** for $f(n)$, to emphasize that we are suppressing constant factors.

Measuring Complexity

- Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor
- You may think of O as representing a suppressed constant
- In practice, most functions f that you are likely to encounter have an obvious highest order term h. In that case, write $f(n) = O(g(n))$, where g is h without its coefficient

Example I

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$. Then, selecting the highest order term $5n^3$ and disregarding its coefficient 5 gives $f_1(n) = O(n^3)$.

Let's verify that this result satisfies the formal definition. We do so by letting $c = 6$ and $n_0 = 10$. Then, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for every $n \geq 10$.

In addition, $f_1(n) = O(n^4)$ because n^4 is larger than n^3 and so is still an asymptotic upper bound on f_1. However, $f_1(n)$ is not $O(n^2)$. Regardless of the values we assign to c and n_0, the definition remains unsatisfied in this case.

Measuring Complexity

Big-O notation has a companion called **small-o notation**. Big-O notation says that one function is asymptotically *no more* than another. To say that one function is asymptotically *less than* another, we use small-o notation.

Definition

Let f and g be functions $f, g : N \to R^+$. Sat that $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

in other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a numbr n_0 exists, where $f(n) < cg(n)$ for all $n \geq n_0$

The difference between the big-O and small-o notations is analogous to the difference between $\leq$ and $<$.

Section 2

Context setup

Context setup

Corresponding to Sipser 7.1 & 7.2

Context setup

- We start by learning how to analyze alghoritms
- We examine how the choice of computational model can affect the time complexity of languages
- We begin the study of **P**

Section 3

Analyzing Algorithms

Analyzing Algorithms

Let get back to $A = \{0^k 1^k | k \geq 0\}$. The algorithm is:

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 if both 0s and 1s remain on the tape:
3.     Scan across the tape, crossing off a single 0 and a single 1
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s
   have been crossed off, reject. Otherwise, if neither 0s nor 1s remain on the tape, accept
```

To analyze M_1 , we consider each of its four stages separately

Analyzing Algorithms

- In first stage the machine scans across the tape to verify that the input is of the form 0^*1^*. Performing this scan uses n steps. As we mentioned earlier, we typically use n to represent the length of the input. Repositioning the head at the left-hand end of the tape uses another n steps. So the total used in this stage is $2n$ steps. In big-O notation, we say that this stage uses $O(n)$ steps
- In stages 2 and 3, the machine repeatedly scans the tape and crosses off a 0 and 1 on each scan. Each scan uses $O(n)$ steps. Because each scan crosses off two symbols, at most $n/2$ scans can occur. So the total time taken by stages 2 and 3 is $(n/2)O(n) = O(n^2)$ steps
- In stage 4, the machine makes a single scan to decide whether to accept or reject. The time taken in this stage is at most $O(n)$
- Thus, the total time of M_1 on an input of length n is $O(n) + O(n^2) + O(n)$. In other words, its running time is $O(n^2)$, which completes the time analysis of this machine

Analyzing Algorithms

Next we set up some notation for classifying languages according to their time requirements

Definition

Let $t : N \rightarrow R^+$ be a function. Define the time complexity class, $TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time TM

The language $A = \{0^k 1^k | k \geq 0\}$ is in $TIME(n^2)$ because M_1 decides A in $O(n^2)$ and $TIME(n^2)$ contains all languages that can be decided in $O(n^2)$ time.

Analyzing Algorithms

Is there a machine that decides A asymptotically more quickly?

In other words, is $A \in TIME(t(n))$ for $t(n) = o(n^2)$? We can improve the running time by crossing off two 0s and two 1s on every scan instead of just one because doing so cuts the number of scans by half. But that improves the running time only by a factor of 2 and doesn't affect the asymptotic running time.

The following machine, M_2 , uses a different method to decide A asymptotically faster. It shows that $A \in TIME(nlog(n))$

Analyzing Algorithms

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 0s and some 1s remain on the tape:
3.     Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, reject
4.     Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1
5. If no 0s and no 1s remain on the tape, accept. Otherwise, reject
```

Before analyzing M_2, let's verify that it actually decides A. On every scan performed in stage 4, the total number of 0s remaining is cut in half and any remainder is discarded. Thus, if we started with 13 0s, after stage 4 is executed a single time, only 6 0s remain. After subsequent executions of this stage, 3, then 1, and then 0 remain. This stage has the same effect on the number of 1s.

Analyzing Algorithms

To analyze the running time of M_2, we first observe that every stage takes $O(n)$ time. We then determine the number of times that each is executed. Stages 1 and 5 are executed once, taking a total of $O(n)$ time. Stage 4 crosses off at least half the 0s and 1s each time it is executed, so at most $1 + log_2 n$ iterations of the repeat loop occur before all get crossed off. Thus the total time of stages 2, 3, and 4 is $(1 + log_2 n)O(n)$, or $O(nlogn)$. The running time of M_2 is $O(n) + O(nlogn) = O(nlogn)$.

We can decide the language A in $O(n)$ time (also called *linear time*) if the TM has a second tape. The following two-tape TM M_3 decides A in linear time. Machine M_3 operates differently from the previous machines for A. It simply copies the 0s to its second tape and then matches them against the 1s.

Analyzing Algorithms

M_3 = On input string w:

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

This machine is simple to analyze. Each of the four stages uses $O(n)$ steps, so the total running time is $O(n)$ and thus is linear. Note that this running time is the best possible because n steps are necessary just to read the input.

Analyzing Algorithms

- In computability theory, the Church–Turing thesis implies that all reasonable models of computation are equivalent—that is, they all decide the same class of languages
- In complexity theory, the choice of model affects the time complexity of languages
- Languages that are decidable in, say, linear time on one model aren't necessarily decidable in linear time on another
- In complexity theory, we classify computational problems according to their time complexity. But with which model do we measure time? The same language may have different time requirements on different models.
- Fortunately, time requirements don't differ greatly for typical deterministic models. So, if our classification system isn't very sensitive to relatively small differences in complexity, the choice of deterministic model isn't crucial

Section 4

Complexity Relationships

Complexity Relationships

Next, we examine how the choice of computational model can affect the time complexity of languages. We consider three models: the single-tape TM; the multitape TM; and the nondeterministic TM.

Theorem

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

Proof idea: The idea behind the proof of this theorem is quite simple. We showed how to convert any multitape TM into a single-tape TM that simulates it. Now we analyze that simulation to determine how much additional time it requires. We show that simulating each step of the multitape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps.

Complexity Replationships

Proof.

Let M be a k-tape TM that runs in $t(n)$ time. We construct a single-tape TM S that runs in $O(t^2(n))$ time. Machine S operates by simulating $M.S$ uses its single tape to represent the contents on all k of M's tapes. The tapes are stored consecutively, with the positions of M's heads marked on the appropriate squares.

Initially, S puts its tape into the format that represents all the tapes of M and then simulates M's steps. To simulate one step, S scans all the information stored on its tape to determine the symbols under M's tape heads. Then S makes another pass over its tape to update the tape contents and head positions. If one of M's heads moves rightward onto the previously unread portion of its tape, S must increase the amount of space allocated to this tape. It does so by shifting a portion of its own tape one cell to the right.

Complexity Replationships

Proof.

Now we analyze this simulation. For each step of M , machine S makes two passes over the active portion of its tape. The first obtains the information necessary to determine the next move and the second carries it out. The length of the active portion of S's tape determines how long S takes to scan it, so we must determine an upper bound on this length. To do so, we take the sum of the lengths of the active portions of M's k tapes. Each of these active portions has length at most $t(n)$ because M uses $t(n)$ tape cells in $t(n)$ steps if the head moves rightward at every step, and even fewer if a head ever moves leftward. Thus, a scan of the active portion of S's tape uses $O(t(n))$ steps.
To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts. Each uses $O(t(n))$ time, so the total time for S to simulate one of M 's steps is $O(t(n))$.

Complexity Replationships

Proof.

Now we bound the total time used by the simulation. The initial stage, where S puts its tape into the proper format, uses $O(n)$ steps. Afterward, S simulates each of the $t(n)$ steps of M, using $O(t(n))$ steps, so this part of the simulation uses $t(n) \times O(t(n)) = O(t^2(n))$ steps.
Therefore, the entire simulation of M uses $O(n) + O(t^2(n))$ steps.
We have assumed that $t(n) \geq n$ (a reasonable assumption because M could not even read the entire input in less time). Therefore, the running time of S is $O(t^2(n))$ and the proof is complete. $\square$

Complexity Relationships

Next, we consider the analogous theorem for nondeterministic single-tape TMs. We show that any language that is decidable on such a machine is decidable on a deterministic single-tape TM that requires significantly more time. Before doing so, we must define the running time of a nondeterministic TM. Recall that a nondeterministic TM is a decider if all its computation branches halt on all inputs.

Definition

Let N be a nondeterministic TM that is a decider. The running time of N is the function $f : N \rightarrow N$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n

The definition of the running time of a nondeterministic TM is not intended to correspond to any real-world computing device. Rather, it is a useful mathematical definition that assists in characterizing the complexity of an important class of computational problems, as we demonstrate next.

Complexity Relationships

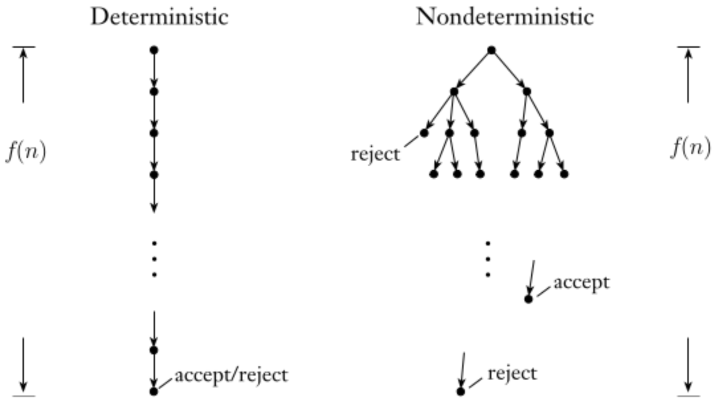

Figure: Measuring deterministic and nondeterministic time

Complexity Relationships

Theorem

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

Proof.

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N as in a previous proof by searching N's nondeterministic computation tree. Now we analyze that simulation.

On an input of length n, every branch of N's nondeterministic computation tree has a length of at most $t(n)$. 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. Thus, the total number of leaves in the tree is at most $b^{t(n)}$

Complexity Relationships

Proof.

The simulation proceeds by exploring this tree breadth first. In other words, it visits all nodes at depth d before going on to any of the nodes at depth $d + 1$. The algorithm inefficiently starts at the root and travels down to a node whenever it visits that node. But eliminating this inefficiency doesn't alter the statement of the current theorem, so we leave it as is. The total number of nodes in the tree is less than twice the maximum number of leaves, so we bound it by $O(b^{t(n)})$. The time it takes to start from the root and travel down to a node is $O(t(n))$. Therefore, the running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.
TM D has three tapes. Converting to a single-tape TM at most squares the running time thus, the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$ and the theorem is proved □

Section 5

The Class P

The Class P

Theorems from *Complexity Relationships* illustrate an important distinction:

- On the one hand, we demonstrated at most a square or polynomial difference between the time complexity of problems measured on deterministic single-tape and multitape TMs
- On the other hand, we showed at most an exponential difference between the time complexity of problems on deterministic and nondeterministic TMs

Polynomial Time

- For our purposes, polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large
- Why this separation between polynomials and exponentials rather than between some other classes of functions?
- First, note the dramatic difference between the growth rate of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n
- For $n = 1000$, the size of a reasonable input to an algorithm. In that case, n^3 is 1 billion, a large but manageable number, whereas 2^n is a number much larger than the number of atoms in the universe
- Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful

Polynomial Time

- Exponential time algorithms typically arise when we solve problems by exhaustively searching through a space of solutions, called **brute-force search**
- For example, one way to factor a number into its constituent primes is to search through all potential divisors. The size of the search space is exponential, so this search uses exponential time
- Sometimes brute-force search may be avoided through a deeper understanding of a problem, which may reveal a polynomial time algorithm of greater utility
- All reasonable deterministic computational models are **polynomially equivalent**. That is, any one of them can simulate another with only a polynomial increase in running time
- When we say that all reasonable deterministic models are polynomially equivalent, we do not attempt to define **reasonable**. However, we have in mind a notion broad enough to include models that closely approximate running times on actual computers

Polynomial Time

- From here on we focus on aspects of time complexity theory that are unaffected by polynomial differences in running time
- Ignoring these differences allows us to develop a theory that doesn't depend on the selection of a particular model of computation
- **You** may feel that disregarding polynomial differences in running time is absurd. Real programmers certainly care about such differences and work hard just to make their programs run twice as quickly
- However, we disregarded constant factors a while back when we introduced asymptotic notation. Now we propose to disregard the much greater polynomial differences, such as that between time n and time n^3
- Our decision to disregard polynomial differences doesn't imply that we consider such differences unimportant

Polynomial Time

Now we come to an important definition in complexity theory

Definition

P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM. In other words

$$P = \bigcup_k TIME(n^k)$$

The class P plays a central role in our theory and is important because

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape TM (robustnes)
- P roughly corresponds to the class of problems that are realistically solvable on a computer (relevance)

Examples

When we analyze an algorithm to show that it runs in polynomial time, we need to do two things:

1. First, we have to give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when it runs on an input of length n
2. Then, we have to examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model. We choose the stages when we describe the algorithm to make this second part of the analysis easy to do

When both tasks have been completed, we can conclude that the algorithm runs in polynomial time because we have demonstrated that it runs for a polynomial number of stages, each of which can be done in polynomial time, and the composition of polynomials is a polynomial

Examples

- One point that requires attention is the encoding method used for problems
- We continue to use the angle-bracket notation $< . >$ to indicate a reasonable encoding of one or more objects into a string, without specifying any particular encoding method
- A reasonable method is one that allows for polynomial time encoding and decoding of objects into natural internal representations or into other reasonable encodings
- For example, many computational problems we study next contain encodings of graphs
- One reasonable encoding of a graph is a list of its nodes and edges. Another is the adjacency matrix, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not
- When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation
- In reasonable graph representations, the size of the representation is a polynomial in the number of nodes
- Thus, if we analyze an algorithm and show that its running time is polynomial (or exponential) in the number of nodes, we know that it is polynomial (or exponential) in the size of the input

Example I

The first example concerns directed graphs. A directed graph G contains nodes s and t. The *PATH* problem is to determine whether a directed path exists from s to t. Let

$$PATH = \{< G, s, t > \mid G \textit{ is a directed graph that has a directed path from s to t}\}$$

Theorem

$PATH \in P$

Example I

Proof idea: We prove this theorem by presenting a polynomial time algorithm that decides *PATH*. Before describing that algorithm, let's observe that a brute-force algorithm for this problem isn't fast enough.
A brute-force algorithm for *PATH* proceeds by examining all potential paths in G and determining whether any is a directed path from s to t. A potential path is a sequence of nodes in G having a length of at most m, where m is the number of nodes in G. (If any directed path exists from s to t, one having a length of at most m exists because repeating a node never is necessary) But the number of such potential paths is roughly m^m, which is exponential in the number of nodes in G. Therefore, this brute-force algorithm uses exponential time.
To get a polynomial time algorithm for *PATH*, we must do something that avoids brute force. One way is to use a graph-searching method such as breadth-first search. Here, we successively mark all nodes in G that are reachable from s by directed paths of length 1, then 2, then 3, through m. Bounding the running time of this strategy by a polynomial is easy.

Examaple I

Proof.

A polynomial time algorithm M for PATH operates as follows

$M =$ On input $< G, s, t >$, where G is a directed graph with nodes s and t:

1. Place a mark on node s
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b
4. If t is marked, *accept* . Otherwise, *reject*.

Example I

Proof.

Now we analyze this algorithm to show that it runs in polynomial time. Obviously, stages 1 and 4 are executed only once. Stage 3 runs at most m times because each time except the last it marks an additional node in G. Thus, the total number of stages used is at most $1 + 1 + m$, giving a polynomial in the size of G

Stages 1 and 4 of M are easily implemented in polynomial time on any reasonable deterministic model. Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which also is easily implemented in polynomial time. Hence M is a polynomial time algorithm for *PATH* $\square$