Scribe Notes for *Algorithmic Number Theory*
Class 6—May 26, 1998

Scribes: Lynn Jones, Nick Loehr, and Hussein Suleman

Abstract

This class continued the review of complexity theory. After reviewing Homework Assignment 1, we defined the notions of many-to-one reductions, polynomial time reductions, and NP-completeness. We discussed various complexity classes relating to time complexity, space complexity, and randomized complexity, e.g., P, NP, $EXPTIME$, $PSPACE$, RP, BPP, and ZPP. At the end of class, we started to discuss parallel complexity and Boolean circuits.

1 Homework Assignment 1

We started class by going over solutions to portions of the first homework assignment. Nick presented the solution to problem 2, and Jeremy presented the solution to problem 3.

2 Reductions

Reductions are mechanisms that allow us to transform and compare instances of problems in terms of computability and complexity. Informally, a reduction is a transformation that lets us solve one problem using the solution to a known problem.

Definition 2.1. (Definition 3.4.1 in the text)
Suppose we have languages $L_1, L_2 \subset \Sigma^*$. A *reduction* from L_1 to L_2 is a computable function $f : \Sigma^* \to \Sigma^*$ such that $x \in L_1$ if and only if $f(x) \in L_2$. This function maps representations of L_1 onto representations of L_2.

Thus, being able to solve the language recognition problem for L_2 allows us to solve the recognition problem for L_1. This relationship is usually denoted

$$L_1 \leq L_2.$$

We can interpret this as stating that the problem posed by L_1 is no harder than that of L_2.

In complexity theoretic frameworks, it is desirable to have reduction functions that are computable in polynomial time. When this is the case, we denote the relationship between the two languages as

$$L_1 \leq_m^P L_2.$$

Here P signifies that the reduction is computable in polynomial time, and m signifies that it is a many-to-one reduction. If we have a polynomial time algorithm for L_2, then we obtain a polynomial time algorithm for L_1 since the reduction is computable in polynomial time, and the composition of polynomials is a polynomial.

3 NP-completeness

Definition 3.1. (Definition 3.4.2 in the text)
A language L is said to be *NP-complete* if:

1. $L \in NP$.
2. For every $M \in NP$, $M \leq_m^P L$.

Intuitively, the NP-complete languages constitute the hardest languages in NP, in the sense that a polynomial time algorithm for recognizing any NP-complete language could be combined with a polynomial time reduction to give a polynomial time algorithm recognizing an arbitrary language in NP. Examples of NP-complete problems include satisfiability, the traveling salesman problem, and the Hamiltonian circuit problem.

4 Randomized Complexity

It is of interest to consider polynomial time algorithms that are provided with an unbiased source of random bits. A randomized algorithm uses a random "guess" consisting of r random bits. Essentially, the algorithm selects a random integer uniformly from the set $\{0, 1, 2, \ldots, 2^r - 1\}$. Of course, we still require that r, the length of the guess in bits, be bounded by $p(\lg x)$ for some polynomial p.

Table 1 lists three complexity classes associated with randomized algorithms. In the table, L represents the language being recognized and x represents an input string.

Table 1: Complexity Classes for Randomized Algorithms.

Complexity Class	Errors	Algorithm Name	Description
RP	one-sided	Monte Carlo	Algorithm always rejects when $x \notin L$; it accepts with probability $\geq 1/2$ when $x \in L$.
BPP	two-sided	Atlantic City	If $x \notin L$, algorithm rejects with probability $\geq 3/4$; if $x \in L$, algorithm accepts with probability $\geq 3/4$.
ZPP	none	Las Vegas	Algorithm runs until it finishes with answer; algorithm has *expected* polynomial running time.

Note the distinction between *nondeterministic* algorithms on the one hand and *randomized* algorithms on the other. Each possible execution path in a nondeterministic algorithm to recognize L takes an instance x and a guess y and either accepts or rejects x. If x is indeed in L, there may be only one "right" guess y that causes the algorithm to accept. In contrast, if there are "many" different valid guesses that will lead the algorithm to accept, a randomized algorithm to recognize L may succeed. Note that the probability that a randomized algorithm will succeed or fail should depend only on the random bits it receives, not on the particular instance x.

The success probabilities appearing in Table 1 are somewhat arbitrary, since the probability of success for any randomized algorithm may be improved by running the algorithm multiple times and taking a majority vote.

5 Space Complexity

In analyzing complexity of a class of algorithms, we also consider the physical space required to carry out the computation. This space, whether measured as cells of some Turing Machine tape or registers in RAM, is the number of units needed to represent the encoding of the state of execution of an algorithm on a particular-sized instance.

Think of memory registers $R_1, R_2, \ldots$, holding integers $x_1, x_2, \ldots$, in some random access machine. Assume that for a particular input of size n, that R_m is the highest numbered cell used at any time by algorithm A on any input of size n. Then we "charge" A for using cells R_1 through R_m (even if some cells in between are not used).

Consider cell R_i. Let x_i be the largest integer (in absolute value) that is ever stored in R_i during execution of algorithm A on any input of size n. Thus, the encoding of x_i is larger than the encoding of any other value that will be stored in register R_i. Then we "charge" A a cost $\lg(x_i)$ for each cell R_i.

Using these conventions, we define the space consumption $S(n)$ for algorithm A on inputs of length n to be

$$S(n) = \sum_{i=1}^{m} \lg(x_i)$$

Just as we asked if a language can be recognized in polynomial time, we also want to know if it can be recognized using polynomial space; i.e., is $S(n) \leq p(n)$ for some polynomial p? If so, then algorithm A has polynomial space complexity.

Previous language classes we discussed were categorized by their time complexity. We now introduce the complexity class, *PSPACE*, the set of languages that have algorithms that can execute in polynomial space. This class contains all of the time complexity classes described so far, and in fact, even some exponential time complexity problems can be solved in polynomial space. Just as we have NP-complete problems, we also have $PSPACE$-complete problems that include, for example, determining the winner of two-player games like chess (suitably generalized to $n \times n$ boards, of course).

6 Summary of Complexity Classes

Complexity classes provide general categorizations for the complexity of algorithms. Figure 1 shows the conjectured relationships among some common complexity classes pertinent to this course. In what follows, we assume $P \neq NP$.

- P is the set of all problems with polynomial time deterministic algorithms. The problems are usually formulated as decision problems or language recognition problems. Problems within this category are called *tractable* since feasible algorithms exist to solve them.

- NP is the set of all problems with polynomial time nondeterministic algorithms. The subclass NP-complete refers to those problems within NP which are as hard as any other in NP. Since no efficient algorithms for solving these problems are known, they are termed *intractable*. Examples of languages in NP which are not known to be in P are PRIMES, COMPOSITES and GRAPH-ISOMORPHISM.

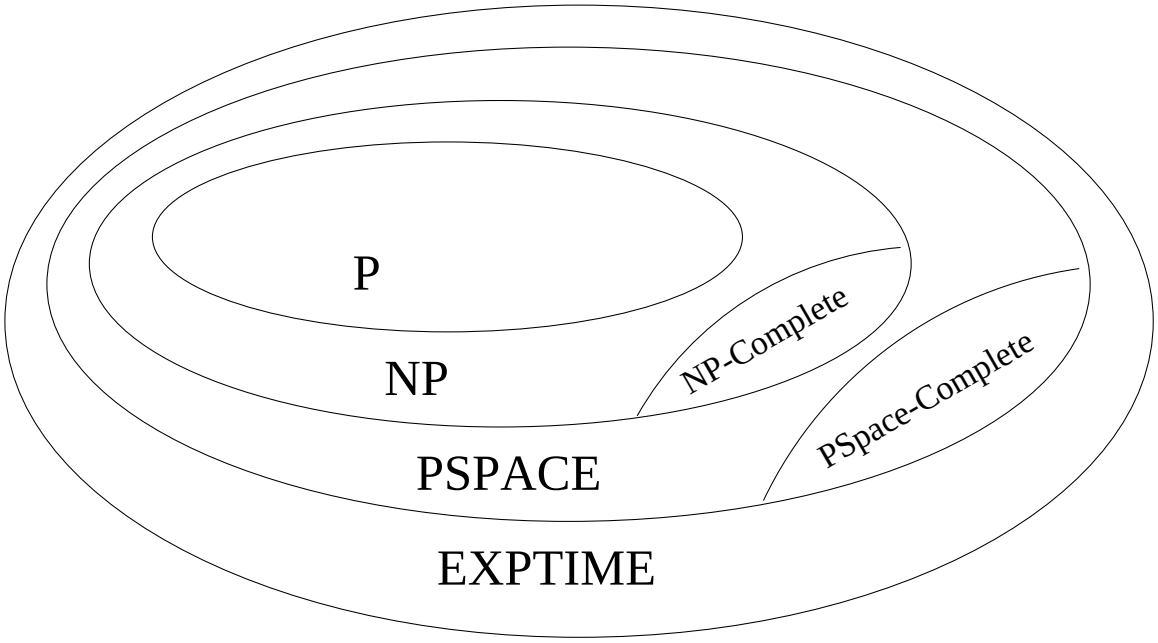

Figure 1: Relationships among complexity classes

- *PSPACE* is the set of all languages that can be recognized with algorithms requiring space of polynomial order. The subclass of *PSPACE*-complete problems consists of those problems within *PSPACE* that are as hard as any other in *PSPACE*. Some game theory problems fall in this category. Note that *PSPACE* can contain within it problems with exponential time complexity but polynomial space complexity.
- *EXPTIME* is the set of all languages that can be recognized with algorithms requiring at most exponential time complexity.

7 Parallel Complexity

We want to use multiple, independently operating processors together to solve a problem, with the hope of speeding up execution. A simple model of parallel processing is the *Boolean circuit* or combinational circuit consisting of AND ($\wedge$), OR ($\vee$) and NOT ($\sim$) gates.

Example 7.1. The addition of two binary numbers can be represented by Boolean formulae.

$$\begin{array}{ccc} c_1 & c_0 & \\ & a_1 & a_0 \\ + & b_1 & b_0 \\ \hline s_2 & s_1 & s_0 \end{array}$$

The carry bits, c_i and the result bits, s_i can be determined by these formulae:

$$
\begin{aligned}
s_0 &= (a_0 \wedge \sim b_0) \vee (\sim a_0 \wedge b_0) \\
c_0 &= a_0 \wedge b_0 \\
s_1 &= (a_1 \wedge b_1 \wedge c_0) \vee (a_1 \wedge \sim b_1 \wedge \sim c_0) \vee (\sim a_1 \wedge b_1 \wedge \sim c_0) \vee (\sim a_1 \wedge \sim b_1 \wedge c_0) \\
c_1 &= (a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0) \\
s_2 &= c_1
\end{aligned}
$$

Note that the formula for s_1 is essentially the parity of the operands. Note too that c_0 is also TRUE if all three of the operands are TRUE, but if all three are, then certainly two of them are.

8 Next Class

Our next class will continue with this example, complete the discussion of complexity theory, and begin the discussion of greatest common divisors.

References

[1] E. Bach and J. Shallit, *Algorithmic Number Theory*, The MIT Press, Cambridge, Massachusetts, 1996.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.