3

Computational Complexity

3.1 Arithmetic Complexity

Many optimization techniques rely on algorithms – especially using a computer – and, as such, it is natural to consider how long an iterative process may take. As processors' speeds vary and technology improves, time is not a good candidate for a metric on "how long". Instead, one approach is to count the number of operations necessary to complete the algorithm. This is illustrated in the following example:

Consider using Gauss-Jordan elimination to solve a system of n equations in n variables as in the following 3×3 case with variables x_1, x_2, and x_3 (for a refresher on this technique, see Section 4.2).

$$\left[\begin{array}{ccc|c} -2 & 3 & 5 & 7 \\ 4 & -3 & -8 & -14 \\ 6 & 0 & -7 & -15 \end{array}\right] \xrightarrow[3R_1+R_3\to R_3]{2R_1+R_2\to R_2} \left[\begin{array}{ccc|c} -2 & 3 & 5 & 7 \\ 0 & 3 & 2 & 0 \\ 0 & 9 & 8 & 6 \end{array}\right] \tag{3.1}$$

$$\xrightarrow{-3R_2+R_3\to R_3} \left[\begin{array}{ccc|c} -2 & 3 & 5 & 7 \\ 0 & 3 & 2 & 0 \\ 0 & 0 & 2 & 6 \end{array}\right] \tag{3.2}$$

We may continue row operations to get the matrix in reduced row echelon form, but this is computationally expensive[1], so we instead back substitute:

$$2x_3 = 6, \text{ so } x_3 = 3; \tag{3.3}$$

$$3x_2 + 2(3) = 0, \text{ thus } x_2 = -2; \text{ and} \tag{3.4}$$

$$-2x_1 + 3(-2) + 5(3) = 7, \text{ hence } x_1 = 1. \tag{3.5}$$

The process of reducing the matrix but stopping short of reaching reduced row echelon form and using back substitution is usually referred to as *Gaussian elimination*.

[1] A good lesson to carry with us as we explore the topics in this text is that it is not always best for a computer to do a problem the same way you and I would solve it on paper. This matter is briefly discussed at the beginning of Section 5.

DOI: 10.1201/9780367425517-3

Counting operations at each step of the Gaussian elimination in our example on 3 variables we have:

step	multiplications	additions	process
3.1	$2(3+1)$	$2(3+1)$	elimination
3.2	$1(2+1)$	$1(2+1)$	elimination
3.3	1	0	back substitution
3.4	2	1	back substitution
3.5	3	2	back substitution

If we consider a system with n variables, then the total number of operations in Gaussian elimination, $G(n)$, is

$$
\begin{aligned}
G(n) &:= \#\text{ operations} \\
&= \#\text{ elim mult} + \#\text{ elim add} + \#\text{ back sub mult} + \#\text{ back sub add} && (3.6)\\
&= \sum_{i=1}^{n}(i-1)(i+1) + \sum_{i=1}^{n}(i-1)(i+1) + \sum_{i=1}^{n} i + \sum_{i=1}^{n}(i-1) && (3.7)\\
&= 2\sum_{i=1}^{n}(i^2-1) + 2\sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 && (3.8)\\
&= 2\frac{n(n+1)(2n+1)}{6} - 2n + \frac{n(n+1)}{2} - n && (3.9)\\
&= \frac{(4n^3+6n^2+2n) - 12n + (3n^2+3n) - 6n}{6} && (3.10)\\
&= \frac{4n^3+9n^2-13n}{6} \text{ or roughly } \frac{2n^3}{3}. && (3.11)
\end{aligned}
$$

For growing values of n we have

n	# operations	$\frac{2}{3}n^3$	% error
1	0	0.666666	33.3333
2	7	5.333333	23.8095
3	25	18.000000	28.0000
4	58	42.666666	26.4367
5	110	83.333333	24.2424
10	795	666.666666	16.1425
20	5890	5333.333333	9.4510
30	19285	18000.000000	6.6632
40	44980	42666.666666	5.1430
50	86975	83333.333333	4.1870
100	681450	666666.666666	2.1693
500	83707250	83333333.333333	0.4466
10^3	666816645000	666666666.666666	0.2241
10^4	666816645000	666666666666.666666	0.0224
10^5	666681666450000	666666666666666.666666	0.0022
10^6	666668166664500000	666666666666666666.6666	0.0002

The polynomial in 3.11 gives the number of arithmetic operations required in n variables with n unknowns. As we see in the table, as n grows large $\frac{4n^3+9n^2-13n}{6}$ is approximated nicely by $\frac{2n^3}{3}$ thus we say that the *arithmetic complexity* of Gaussian elimination is of the order $\frac{2n^3}{3}$.

It is important to note that there is much more a computer is doing than just arithmetic operations when it does a calculation. One very important process we have ignored in our example is calls to memory and these can be very expensive. Including all that is involved in memory makes a runtime assessment much more difficult and often this component is ignored. The purpose of these calculations is not to get a precise measurement of how long it will take a computer to complete an algorithm, but rather to get a rough idea of all that is involved so that we can compare the algorithm to other techniques that accomplish the same task and therfore have some means to compare which is more efficient. A thorough treatment of all this can be found in the excellent textbook [11].

3.2 Asymptotic Notation

As we saw in the example in the previous section, $\frac{2n^3}{3}$ is a very good approximation for $\frac{4n^3+9n^2-n}{6}$ as n grows large. This agrees with our intuition that as n gets big, the only term that really matters in the polynomial is the leading term. This idea is encapsulated in *asymptotic notation* (think of "asymptotic" as a synonym for "long-run behavior") and, in particular for our purposes, *big O notation.*

Definition 3.2.1 (Big O Notation)**.** *Let g be a real-valued function (though this definition also holds for complex-valued functions). Then*

$$O(g) := \{f| \text{ there exist positive constants } C, N \text{ such that } 0 \leq |f(x)| \leq C|g(x)| \text{ for all } x \geq N\}.$$

Thus $O(g)$ is a family of functions $\mathscr{F}$ for which a constant times $|g(x)|$ is eventually an upper bound for all $f \in \mathscr{F}$. More formally, f being $O(g)$ means that as long as $g(x) \neq 0$ and the limit exists, $\lim_{x\to\infty} |f(x)/g(x)| = C$ or 0 (if g is too big) where C is some positive constant.

Example 3.2.2. *Show that $\frac{4n^3+9n^2-n}{6}$ is $O(\frac{2n^3}{3})$ where $n \in \mathbf{N}$.*

Solution. For $n \geq 1$ (thus $N = 1$),

$$\begin{aligned}
&\left|\frac{4n^3 + 9n^2 - n}{6}\right| \\
&\quad \leq \left|\frac{4n^3}{6}\right| + \left|\frac{9n^2}{6}\right| + \left|\frac{n}{6}\right| \text{ by the Triangle Inequality (Theorem B.2.3)} && (3.12)\\
&\quad \leq \frac{4n^3}{6} + \frac{9n^3}{6} + \frac{n^3}{6} \text{ since } n \geq 1 && (3.13)\\
&\quad = \frac{14n^3}{6} && (3.14)\\
&\quad = \frac{7}{2} \cdot \left|\frac{2n^3}{3}\right| && (3.15)
\end{aligned}$$

establishing $\frac{4n^3+9n^2-n}{6}$ is $O(\frac{2n^3}{3})$ where $C = 7/2$. Notice that

$$\lim_{n\to\infty} \left(\frac{4n^3 + 9n^2 - n}{6}\right) / \left(\frac{2n^3}{3}\right) = 1 \tag{3.16}$$

■

Regarding our work in Example 3.2.2 one usually would not include the constant 2/3 but rather report the answer as $\frac{4n^3+9n^2-n}{6}$ is $O(n^3)$. This is, of course, because the constant does not matter in big O notation. Some Numerical Analysis texts use this problem as an example, though, and include the constant 2/3 when reporting the approximate number of [2]. We have kept the constant to be consistent with those texts and though technically correct, including the constant in the $O(\cdot)$ can viewed as bad form.

It is important to realize that Big O notation gives an asymptotic (long run) upper bound on a function. It should also be noted that when using big O notation, $O(g(x))$ is a set and, as such, one should write "$h(x) \in O(g(x))$". Note, though, that it is standard practice to abuse the notation and state "$h(x) = O(g(x))$" or "$h(x)$ is $O(g(x))$".

One further observation before some examples. We have shown that the number of arithmetic operations in performing Gaussian elimination to solve a linear system in n variables is $G(n) = \frac{4n^3+9n^2-n}{6}$ and that this function is $O(\frac{2}{3}n^2)$. Furthering our work in Example 3.2.2 by picking up in 3.15 we have

$$\begin{aligned}
\frac{7}{2}\left|\frac{2n^3}{3}\right| &= \frac{7}{3}n^3 && (3.17)\\
&< n^4 \text{ for } n \geq 3. && (3.18)
\end{aligned}$$

[2]The reason for this is that using *Cholesky Decomposition* (Chapter 5) to solve a system of linear equations is $O(\frac{n^3}{3})$; i.e. twice as fast as Gaussian elimination.

Thus, not only is $G(n) = O(\frac{2}{3}n^3)$, $G(n) = O(n^4)$. In fact,

Observation 3.2.3. *Let x be a positive real number and suppose $f(x)$ is $O(x^k)$. Then for any $l > k$, $f(x)$ is $O(x^l)$.*

We now consider a few more important examples before moving on.

Example 3.2.4. *Let $n \in \mathbf{Z}^+$. Then*

$$1 + 2 + 3 + \cdots + n \leq \overbrace{n + n + n + \cdots + n}^{n \text{ terms}} = n^2 \tag{3.19}$$

and taking $C = N = 1$ we see that the sum of the first n positive integers is $O(n^2)$.

Example 3.2.5. *Let $n \in \mathbf{Z}^+$. Then*

$$n! := n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1 \leq \overbrace{n \cdot n \cdot n \cdot \cdots \cdot n}^{n \text{ factors}} = n^n \tag{3.20}$$

and taking $C = N = 1$ we see that $n!$ is $O(n^n)$.

Example 3.2.6. *Show that for $n \in \mathbf{N}$, $f(n) = n^{k+1}$ is not $O(n^k)$ for any nonnegative integer k.*

Solution. Let us assume for contradiction that the statement is true, namely there exist positive constants C and N such that $n^{k+1} \leq Cn^k$ for all $n \geq N$. Thus for all $n \geq N$, $n \leq C$, which is absurd, since n is a natural number and therefore unbounded. Thus n^{k+1} cannot be $O(n^k)$. ■

Growth of basic "orders" of functions are shown in Figure 3.1. Note that though $n!$ is defined for nonnegative integers, Exercises 9.1 and 9.2 show how to extend the factorial function to the real numbers.

Before concluding this section, we mention that other asymptotic notation exists, for example: $o(g(x))$, $\omega(g(x))$, $\Omega(g(x))$, $\Theta(g(x))$, etc., but we do not consider them here[3].

3.3 Intractability

Some problems we will encounter will have solutions that can be reached in theory, but take too much time in practice, are said to be *intractable*. Conversely, any problem that can be solved in practice is said to be *tractable*; that is, "easily worked" or "easily handled or controlled" [16]. The bounds between

[3]The interested reader is encouraged to read the appropriate sections in [11] or [48].

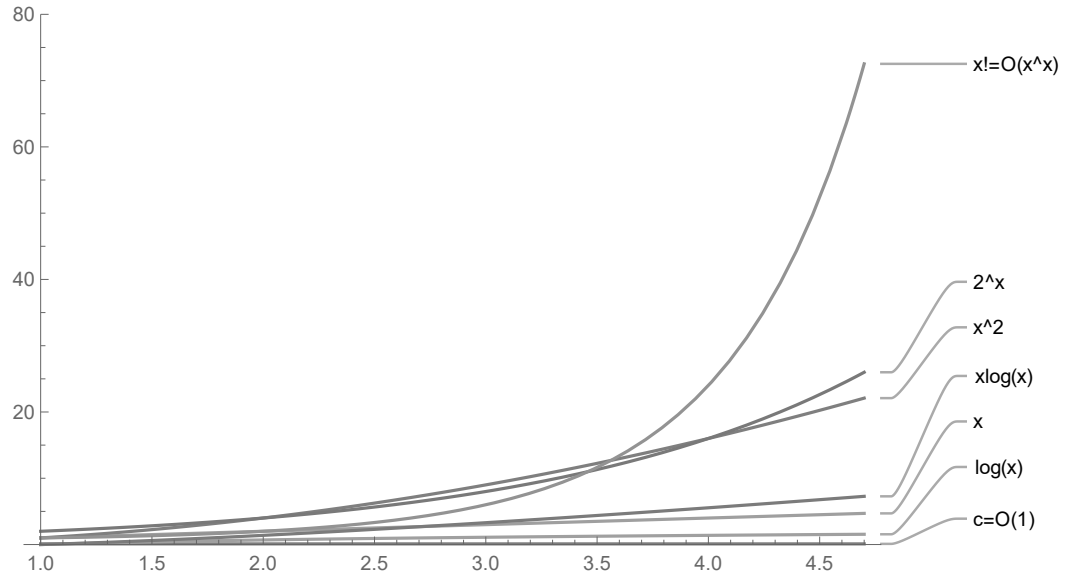

FIGURE 3.1
The growth of functions.

these two are not clearly defined and depend on the situation. Though the discipline lacks a precise definition of both tractable and intractable, their usage is standard and necessary for many situations encountered in Optimization.

For an example, let us revisit using Gauss-Jordan elimination as was considered in Section 3.1. Oak Ridge National Laboratory unveiled in 2018 its supercomputer *Summit* capable of 122.3 petaflops (122.3×10^{15}) calculations per second. Without worrying about the details, let us assume a good PC can do 100,000,000,000 $= 10^{11}$ calculations per second (this is a little generous). By our work in Section 3.1, to perform Gaussian elimination on a matrix with 10^6 rows (i.e. a system of equations with 10^6 variables) it would take Summit

$$\frac{2}{3}(10^6)^3/122.3 \times 10^{15} \approx 5.45 \text{ seconds.} \tag{3.21}$$

On a good PC this would take

$$\left(\frac{2}{3}(10^6)^3/10^{11}\right)/86400 \text{ seconds per day } \approx 77 \text{ days.} \tag{3.22}$$

Note that these calculation are not exact as we have not consider calls to memory, etc., but they do illustrate the point.

Spending 77 days to solve a problem is a nuisance, but not an insurmountable situation. Depending on the practice one would have to decide if this amount of time makes the problem intractable or not. But Gauss-Jordan elimination is a polynomial time algorithm, so to better illustrate this point let us now assume we have a program that runs in *exponential time*; say one that has as its runtime 2^n. For $n = 100$ (considerably less than 1,000,000) this

program would run on Summit for

$$\left(2^{100}/122.3 \times 10^{15}\right)/31536000 \text{ seconds per year } \approx 3.3 \text{ years!} \tag{3.23}$$

We will not even consider how long this would take on a good PC. If we consider a system with $n = 10^3$ variables, the runtime on Summit becomes 2.7×10^{276} years which is 2×10^{266} times the age of the universe.

We will close this section by noting that most computer security depends on intractability. The most used public key encryption scheme is known as RSA encryption. This encryption scheme uses a 400 digit number that is known to be the product of two primes and it works well since currently factoring algorithms for a number this large are intractable[4]. The intractability of this problem will most likely change with quantum computing.

3.4 Complexity Classes

3.4.1 Introduction

It would be helpful to have a metric by which to classify the computational difficulty of a problem. One such tool is the complexity class. A *complexity class* is a set of problems that can be solved using some model of computation (often this model of computation is given a limited amount of space, time, or some other resource to work out the problem). A *model of computation* is any method of computing an output given an input.

One of the most useful models of computation is the Turing machine. For the sake of our discussion we can consider a "Turing machine" as any algorithm that follows the following method:

0. Initialize an infinite string of "blank symbols". Each symbol can be referred to by its position, starting at position 0. This string of symbols is called the *tape* of the Turing machine.

1. Write a given finite string of symbols, s, to the tape starting at position 0. Each symbol in s must be taken from a given finite set of symbols, A, known as the *alphabet* of the Turing machine. A contains the blank symbol. We often restrict s so that it cannot contain the blank symbol.

2. Choose an arbitrary non-negative integer k and read the kth symbol of the tape. This step is referred to as "moving the head of the Turing Machine to position k".

[4]The nature of primes is also involved here. Although they are the building blocks of all integers, we know little about them, especially how many there are in (large) intervals and where they reside.

3. Change the kth symbol of the tape to any symbol in A.
4. Go to step 2 or step 5.
5. Declare either "accept" or "reject" (This is referred to as "accepting or rejecting the input string".) A Turing machine that reaches this step is said to *halt*.

When we discuss Turing machines we usually just assume step 0 has already occurred but its important to note that Turing machines in concept have an infinite amount of space on the tape to store information and that they look up information by moving the head to its position on the tape. For any position k on the tape we say that position $k-n$ is "n to the left" of position k and the position $k+n$ is "n to the right" of position k. The following algorithm is an example of a Turing Machine that we will call TM:

1. Write a given string of symbols, s, to the beginning of the tape where every symbol of s is in the set $\{0, 1\}$.
2. Move the head to the first blank after s. (We will call this position on the tape p).
3. Write the symbol 0 to position p.
4. Move the head to position 0.
5. If the symbol at the head's position is 1, move the head to position p and change the symbol at position p to a 1 if it is a 0 and to a 0 if it is a 1, then move the head back to where it was when we started this step.
6. Move the head one symbol to the right.
7. If the head is at position p, go to step 8. Otherwise go to step 5.
8. Read the symbol at position p. If it is 1, accept. If it is 0, reject.

TM is a simple Turing machine that accepts all strings that contain an odd number of 1 symbols and rejects all strings that do not. Hence, TM solves a decision problem. A *decision problem* is a problem with an answer than is either "yes" or "no". TM solves or *decides* the decision problem "Does the given string s have an odd number of 1 symbols?" When we say that TM "solves" this problem, we mean:

1. If L is the set of all strings that contain an odd number of 1 symbols, TM will accept a string s if and only if $s \in L$. (This is equivalent to saying that "L is the language of TM")
2. TM halts on every string. (It always reaches step 5 in the original given Turing machine definition).

Note that the second point is not trivial because Turing machines can get stuck in an infinite loop of moving the head and writing symbols. Now that we have a working definition of a Turing machine, we will explore how complexity classes are defined using Turing machines as their model of computation.

3.4.2 Time, Space, and Big O Notation

As previously mentioned, complexity classes usually give their chosen model of computation a limited resource. When defining complexity classes involving Turing Machines we often limit either the amount of *time* or *space* the Turing machine can use to solve the problem. *Time* refers to the amount of head-moving[5] and writing steps the Turing machine can make and *space* refers to the number of positions after the given input string on the tape the Turing machine may use.

In general, we are concerned with how Turing machines run on all inputs they can take. This presents a problem – because how much space or time a Turing machine uses to run can significantly vary based on the size of the input it is given. It is therefore usually the case that we do not place problems in complexity classes based on the absolute amount of time or space a Turing machine that solves the problems will take. Instead we allow Turing machines to use an asymptotically increasing amount of resources relative to the size of their inputs.

As we are thinking asymptotically, it is helpful to use Big O notation when defining complexity classes. Consider a Turing machine T and $h(n)$, where $h(n)$ is a function that maps the integer n to the greatest amount of read and write steps T will take on an input string of size n. If $h(n) \in O(f(n))$ we say that "T runs in $O(f)$ time".

3.4.3 The Complexity Class P

P is the set of all decision problems that can be solved in polynomial time by a Turing machine. More formally, a set of strings L is in P if and only if there exists a Turing machine whose language is L that runs in $O(n^k)$ time where k is some integer (Note that this means P is a set of languages. Not a set of Turing machines). P is sometimes thought of as the set of decision problems whose solutions are tractable. This is not the whole truth – some problems can be bounded above by large polynomials and be in P and some problems can not be asymptotically bound by a polynomial but still be solvable for reasonably sized inputs – but it is very often the case that problems in P are tractable and problems out of P with large inputs are not.

[5]In our given definition of a Turing machine we conceptualized moving the head of the Turing machine to an arbitrary position as "one step" in the algorithm. However – when counting the number of operations a Turing machine takes if a Turing machine moves from position p_1 to position p_2 the Turing machine has taken $|p_1 - p_2|$ steps.

For this reason polynomial time algorithms – especially algorithms that run in $O(n \log n)$ time or quicker – are often the goal when attempting to optimize the speed at which a decision problem is solved by a Turing machine. Though finding such algorithms for some problems is often elusive and sometimes impossible. That being said many practical problems exist in P. Determining whether or not there is a path of length n or less between two cities when given the set of roads between them, determining whether or not an item is on a list, determining whether two integers are coprime, and determining whether an integer is prime are all decision problems that exist in P. (The input strings for these problems are string representations of the integers, paths between cities, etc).

3.4.4 The Complexity Class NP

NP is a complexity class that contains all languages L for which if $s \in L$ there exists a proof that can be verified in polynomial time that s is in L[6]. More precisely, A language L is in NP if there exists a Turing machine, $TM(s, c)$[7] for which the following properties hold:

1. TM always runs in $O(n^k)$ time on any input s, c where n is the length of s and k is some integer.
2. If $s \in L$ – there exists a c such that $TM(s, c)$ accepts. (We say that c is the *certificate* of s).
3. if $s \notin L$ – no c exists such that $TM(s, c)$ accepts.

An example may be enlightening. Consider the subset-sum problem: "Given an integer n and a set of integers S, is there some subset of S that adds to n?" In this instance the certificate that can be verified in polynomial time is a subset of S that adds to n. If given S, n, and a subset of S that adds to n, a deterministic Turing machine could add the given subset and accept if and only if it adds to n. Such a subset will always exist if $(S, n) \in L$ and because we only accept if the subset of S adds to n – this Turing machine will never be "tricked" into accepting a string that is not in L (i.e. the third above premise holds). Hence, if $(S, n) \in L$ there is a proof that can be verified in polynomial time that (S, n) is in L.

NP is an interesting complexity class because many important optimization problems exist in NP. Whether or not a graph is n-colorable (see Chapter 21), determining whether or not there is a Hamiltonian cycle of weight less than n (also introduced in Chapter 21), whether or not a set of Boolean clauses

[6]NP is short for "non-deterministic polynomial" which is a description of a more generalized version of a Turing machine. For more information check out Chapter 2 of Computational complexity: a modern approach in the further reading section.

[7]Because Turing machines can take arbitrary strings as arguments – it is conventional to use the notation $TM(s_1, s_2, ...s_n)$ if drawing a distinction between different parts of the input is important.

can be satisfied, determining whether a set of flights can be scheduled without any conflicts, and determining whether k varying sized objects can be packed into n bins are all problems in NP[8]. NP is also mysterious in that it is unknown whether or not $P = NP$ – which means it is possible that there is are efficient algorithms for solving the toughest problems in NP that have not yet been discovered. Much effort has been dedicated to trying to find efficient algorithms for solving NP problems and there are a wide variety of heuristics for doing so that work well but not perfectly. For an example of a heuristic, see the *Nearest Neighbor Algorithm* or the insertion algorithms for solving the *Traveling Salesperson Problem* (Chapter 25).

3.4.5 Utility of Complexity Classes

Complexity classes are a useful tool for classifying the computational difficulty of solving a problem. It is almost universally the case that if a problem is in $EXPTIME$ (the class of problems solvable by Turing machines in exponential time) and not in P – that problem will be more difficult than one in P as inputs get large. If we are trying to determine the computational difficulty of a problem it may therefore be helpful to discover information about which complexity class it belongs to. In particular it may be useful to determine if the problem is in P, NP, or neither.

3.5 For Further Study

Parts of these texts have excellent presentations on what we have considered and can be used to deepen ones understanding of the material presented in this chapter:

- *An Introduction to Algorithms,* 3rd edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein; MIT Press (2009). (This is the go-to text for many experts when it comes to the study of algorithms.)
- *Discrete Mathematics and Its Applications,* 8th edition by Kenneth Rosen, McGraw-Hill (2019)
- *Concrete Mathematics a Foundation for Computer Science,* 2nd edition by Ronald Graham, Donald Knuth, and Oren Patashnik, Addison Wesley (1994)

[8]These problems are *NP-complete* meaning that all problems in NP are less difficult than they are. For a more thorough discussion of what that means – see chapter 2 of *Computational complexity: A Modern Approach* listed in the further study section of this chapter.

This list is not, of course, an exhaustive list of excellent sources for a general overview of optimization.

3.6 Keywords

Gauss-Jordan elimination, Gaussian elimination, arithmetic complexity, asymptotic notation, big O notation, intractability, complexity class, model of computation, Turing machine, P, NP.

3.7 Exercises

Exercise 3.1. *In this chapter, we determined that the number of arithmetic operations to perform Gaussian elimination to solve a system in n variables is* $G(n) = \frac{4n^3+9n^2-13n}{6} = O(\frac{2}{3}n^3)$. *Consider a linear system with* 412 *variables.*

i) What is the exact number of arithmetic operations to use Guassian elimination to solve this system?

ii) Calculate the value of the order $\frac{2}{3}n^3$ *for this system and determine the percent error of this approximation of* $G(n)$.

iii) What is the first value of n for which $O(\frac{2}{3}n^3)$ *has less than a* 1% *error in approximating* $G(n)$*?*

Exercise 3.2. *Show that* $x^2 + x + 1$ *is* $O(x^2)$ *but not* $O(x)$.

Exercise 3.3. *Let* n, k *be positive integers. Show* $1^k+2^k+\cdots+n^k$ *is* $O(n^{k+1})$.

Exercise 3.4. *Prove Observation 3.2.3.*

Exercise 3.5. *Construct a Turing machine whose alphabet is* $\{0, 1\}$ *that has the language of all strings that contain more 1s than 0s.*

Exercise 3.6. *Construct a Turing machine whose alphabet is the set of integers between 0 and 100 that sorts the integers in the given string s in ascending order, then accepts.*

Exercise 3.7. *Prove that if L is a language with a finite amount of strings then* $L \in P$.

Exercise 3.8. coP *is the class of decision problems for which if* $L \in P$ *then* $\bar{L} \in coP$. *Prove that* $P = coP$.

Exercise 3.9. *Prove that* $P \subseteq NP$.