G *Analysis of Algorithms and Complexity Classes*

G.1 Running Time

Yet another way to study an algorithm is to compute the running time of the algorithm purely as a function of the length of the input. *Worst-case analysis* considers the longest running time of all inputs of a particular length. *Average-time analysis* considers the average of all the running times of inputs of a particular length. The worst-case analysis is typically referred to as the *running time* of an algorithm.

Finding an expression for the exact running time is often difficult, but in most cases close estimations are possible. *Asymptotic analysis* provides the means of analyzing the running time of the algorithms for large inputs. As the lengths of the inputs become large, the high-order terms dominate the value of the expression and the low-order terms have little or no effect. Hence, a close approximation can be obtained only by considering the highest-order term in an expression. For example, the highest-order term of the function $f(n) = 2n^5 + 100n^3 + 27n + 2003$ is $2n^5$. As n becomes large, disregarding the coefficient 2, the function $f(n)$ behaves like n^5 and it is said that $f(n)$ is asymptotically at most n^5. The following are some common definitions that are useful in analyzing the asymptotic behavior of functions and hence algorithms.

DEFINITION G.1.1 *Let f and g be two functions $f, g : \mathbb{N} \to \mathbb{R}^+$.*

- *The function g is an* asymptotically upper bound *for f, denoted $f(n) \in O(g(n))$ and read f is big-O of g, if there* exists *a constant $c > 0$ and $n_0 \in \mathbb{N}$ such that $\forall n \geq n_0$,*

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

- *The function g is an* asymptotically strict upper bound *for f, denoted $f(n) \in o(g(n))$ and read f is small-o of g, if for* every *constant $c > 0$ and $n_0 \in \mathbb{N}$ such that $\forall n \geq n_0$,*

  $f(n) < cg(n)$.

- *The function g is an* asymptotically lower bound *for f, denoted $f(n) \in \Omega(g(n))$ and read f is big-Omega of g, if there* exists *a constant $c > 0$ and $n_0 \in \mathbb{N}$ such that $\forall n \geq n_0$,*

  $cf(n) \geq g(n)$.

- *The function g is an* asymptotically strict lower bound *for f, denoted $f(n) \in \omega(g(n))$ and read f is small-omega of g, if for* every *constant $c > 0$ and $n_0 \in \mathbb{N}$ such that $\forall n \geq n_0$,*

  $f(n) > cg(n)$.

- *The function g is* asymptotically equal *to f, denoted $f(n) \in \Theta(g(n))$ and read f is theta of g, if*

  $f(n) \in O(g(n)) \quad and \quad f(n) \in \Omega(g(n))$.

Considering only the highest terms and disregarding constant factors, the big-O notation says that the function f is no more than the function g. The big-O notation is thought of as suppressing a constant factor. For example, $f(n) = 5n^4 + 7n^3 - 4n^2 \in O\left(n^4\right)$, $f(n) = n^{\log n} + n^{100} \in O\left(n^{\log n}\right)$, etc. When f is strictly less than g, the small-o notation is used. The small o-notation indicates that the function g grows much faster than the function f. For example, $f(n) = \log^{100} n \in o\left(n^{1/100}\right)$, $f(n) = n^{40} \in o(2^n)$, etc. The notations Ω and ω express the opposite of O and o notations, respectively. Thus, the big-Omega notation indicates that f grows no slower than g, and the small-omega notation indicates that f grows faster than g. When the functions f and g grow at the same rate, the Θ notation is used. For example, $f(n) = 3n^5 + n^4 \in \Theta\left(n^5\right)$.

When describing the running time of different algorithms, certain terms come up frequently. The running times of common algorithms such as matrix multiplication, sorting, shortest path, etc., are $O(n^c)$, where c is some positive constant. In such cases, it is said that the running time is polynomial in the length of the input n. Other algorithms, such as satisfiability of Boolean expressions, Hamiltonian paths, decomposition of integers into prime factors, etc., are $O(2^{n^c})$, where c is some positive constant. Such algorithms are said to be running in exponential time. The

following table summarizes some of the common characterizations of the running time of algorithms (c is some positive constant).

	Running time
constant	$O(1)$
logarithmic	$O(\log n)$
polylogarithmic	$O(\log^c n)$
linear	$O(n)$
polynomial	$O(n^c)$
quasipolynomial	$O\left(n^{\log^c n}\right)$
exponential	$O\left(2^{n^c}\right)$
doubly expnonential	$O\left(2^{2^{n^c}}\right)$

G.2 Complexity Theory

The goal of *complexity theory* is to characterize the amount of resources needed for the computation of specific problems. Common resources include sequential time, sequential space, number of gates in Boolean circuits, parallel time in a multiprocessor machine, etc. The exact complexity of a problem is determined by the amount of resources that is both sufficient and necessary for its solution. Sufficiency implies an upper bound on the amount of resources needed to solve the problem for every instance of the input. Necessity implies a lower bound, i.e., for some instance of the input, at least a certain amount of resources is required to solve the problem.

The amount of resources that is needed to solve a problem allows for an elegant classification of problems according to their computational complexity. Researchers have developed the notion of *complexity classes*, where a complexity class is defined by specifying (a) the type of computation model M, (b) the resource R which is measured in this model, and (c) an upper bound U on this resource. A complexity class, then, consists of all problems requiring at most an amount U of resource R for their solution in the model M. Thus, the *complexity of a problem* is determined by finding to which complexity classes it belongs (by providing upper bounds on the resource) and to which complexity classes it does not belong (by providing lower bounds). To define complexity classes more precisely, we will need to make use of definitions of alphabets, strings, and languages.

Input Representation

The amount of the resource used in a complexity class is expressed in terms of the length of the input. It is not clear, however, how to define the length of the input since it can be of different types and values, i.e., integers, names, graphs, matrices, etc. It is convenient to have a unique and clear definition of the length of the input. To this end, researchers have proposed the encoding of inputs as strings over a set of symbols and have defined the length of the input as the number of symbols of the encoding string.

DEFINITION G.2.1 *An alphabet, usually denoted by* Σ*, is any finite set of symbols.*

DEFINITION G.2.2 *A string* s *over an alphabet* Σ *is a sequence of symbols from* Σ*. The length of a string* s*, denoted* $|s|$*, is equal to the number of its symbols. The set of all strings over the alphabet* Σ *is denoted by* Σ^**.*

The encoding of an input a is denoted by $\text{enc}(a)$. To illustrate, let $\Sigma = \{0, 1\}$. Then, integers can be encoded in standard binary form, e.g., the encodings of 5 and 35 are 101 and 100001 of lengths 3 and 6, respectively. The encoding of a graph $G = (V, E)$, where $V = \{v_1, \ldots, v_n\} \subset \mathbb{N}$ and $E = \{(v'_1, v''_1), \ldots, (v'_m, v''_m)\} \subseteq V \times V$, can be obtained by concatenating the encodings of its vertex set and its edge set. The vertex set and the edge set can be encoded by concatenating the encodings of the vertices and of the edges, respectively. Special markers can be used to indicate the ending of a vertex and edge encoding. Thus, $\text{enc}(G) = \text{enc}(v_1) \circ \text{enc}(*) \circ \cdots \circ \text{enc}(v_n) \circ \text{enc}(*) \circ \text{enc}(+) \circ \text{enc}(v'_1) \circ \text{enc}(*) \circ \text{enc}(v''_1) \circ \text{enc}(*) \circ \cdots \circ \text{enc}(v'_m) \circ \text{enc}(*) \circ \text{enc}(v''_m) \circ \text{enc}(*)$, where $\text{enc}(*)$ and $\text{enc}(+)$ are the encodings of special markers used to separate vertices and indicate the start of the edge encodings, respectively, and $\circ$ denotes concatenation.

Problem Abstraction

A problem can be thought of as mapping an input instance to a solution. In many cases, we are interested in problems whose solution is either "yes" or "no." Such problems are known as *decision problems*. For example, the graph-coloring problem asks whether it is possible to color the vertices of a graph $G = (V, E)$ using k different colors such that no two vertices connected by an edge have the same color. In many other cases, we are interested in finding the best solution according to some criteria. Such problems are known as *optimization problems*. To continue our example, we may be interested in determining the minimum number of colors needed to color a graph. Generally, an optimization problem can be cast as a decision problem by imposing an upper bound. In our example, we can determine the minimum number of colors needed to color a graph $G = (V, E)$ by invoking the corresponding decision problem with $k = 1, \ldots, |V|$ until the answer to the decision problem is "yes."

In the rest of the section, we restrict our attention to decision problems since their definition is more amendable to complexity analysis and since other problems can be cast as decision problems.

Languages

Languages provide a convenient framework for expressing decision problems.

DEFINITION G.2.3 *A language L over an alphabet Σ is a set of strings over the alphabet Σ, i.e., $L \subseteq \Sigma^*$.*

The language defined by a decision problem includes all the input instances whose solution is "yes." For example, the graph-coloring problem defines the language whose elements are all the encodings of graphs that can be colored using k colors.

Acceptance of Languages

An algorithm A accepts a string $s \in \Sigma^*$ if the output of the algorithm $A(s)$ is "yes." The string s is rejected by the algorithm if its output $A(s)$ is "no." The language L accepted by an algorithm A is the set of strings accepted by the algorithm, i.e.,

$L = \{s : s \in \Sigma^* \text{ and } A(s) = \text{"yes"}\}$.

Note that even if L is the language accepted by the algorithm A, given some input string $s \notin L$, the algorithm will not necessarily reject s. It may never be able to determine that $s \notin L$ and thus loop forever. Language L is *decided* by an algorithm A if for every string $s \in \Sigma^*$, A accepts s if $s \in L$ and A rejects s if $s \notin L$. If L is decided by A, it guarantees that on any input string the algorithm will terminate.

DEFINITION G.2.4 *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. An algorithm A* decides a language *L over some alphabet Σ in time $O(t(n))$ if for every string s of length n over Σ, the algorithm A in $O(t(n))$ steps accepts s if $s \in L$ or rejects s if $s \notin L$. Language L is decided in time $O(t(n))$.*

Complexity Classes

We are now ready to define some of the most important complexity classes. We start with the definition of the polynomial-time complexity class.

DEFINITION G.2.5 *A language L is in* P *if there exists a polynomial-time algorithm A that decides L.*

The complexity class P encompasses a wide variety of problems such as sorting, shortest path, Fourier transform, etc. Roughly speaking, P corresponds to all the problems that admit an efficient algorithm. Generally, we think of problems that are solvable by polynomial time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

Indeed, for many problems there are no polynomial-time algorithms. For example, deciding whether or not a graph $G = (V, E)$ can be colored with three colors is not known to be in P. These problems can be solved by brute-force algorithms in exponential time.

DEFINITION G.2.6 *A language L is in* EXPTIME *if there exists an exponential-time algorithm A that decides L.*

Interestingly enough, many of these hard problems have a feature that is called polynomial-time verifiability. That is, although currently it is not possible to solve these problems in polynomial time, if a candidate solution to the problem, called a certificate, is given, the correctness of the solution can be verified in polynomial time. For example, a certificate for the graph-coloring problem with three colors would be a mapping that for each vertex indicates its color. The correctness can be verified in polynomial time by examining all the edges and checking for each edge that the colors of its two vertices are different. This observation is captured by the following definition.

DEFINITION G.2.7 *A language L is in* NP *if there exists a polynomial-time verifier algorithm A and a constant c such that for every string s there exists a certificate y of length $O(|s|^c)$ such that $A(s, y) =$ "yes" if $s \in L$ and $A(s, y) =$ "no" if $s \notin L$.*

It is clear that P $\subseteq$ NP since any language that can be decided in polynomial time can also be decided without the need of a certificate. The most fundamental question in complexity theory is whether P $\subset$ NP or P $=$ NP. After many years of extensive research the question remains unanswered. An important step was made in the 70s when Cook and Levin related the complexity of certain NP problems to the complexity of all NP problems. They were able to prove that if a polynomial-time algorithm existed for one of these problems, then a polynomial-time algorithm could be constructed for any NP problem. These special problems form an important complexity class known as NP-complete.

DEFINITION G.2.8 *A language L_1 is* polynomial time reducible *to a language L_2, denoted $L_1 \leq_p L_2$, if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$, such that for every $s \in \Sigma^*$,*

$$s \in L_1 \iff f(s) \in L_2.$$

f is called the reduction function and the algorithm F that computes f is called the reduction algorithm.

If a language L_1 is reducible to a language L_2 via some polynomial-time computable function f, and if L_2 has a polynomial-time algorithm A_2, then we can construct a polynomial-time algorithm A_1 for L_1. Given some input string s, algorithm A_1 invokes F to compute $f(s)$ and then invokes A_2 on $f(s)$ and gives the same answer as A_2. Thus, via reductions, the solution of one problem can be used to solve other problems.

DEFINITION G.2.9 *A language L is in* NP-complete *if*

1. $L \in$ NP*, and*

2. *if $L' \in$* NP*, then $L' \leq_p L$.*

If L satisfies the second condition, but not necessarily the first condition, then L is NP-hard.

It is clear now that if an NP-complete problem has a polynomial-time algorithm, then via reductions it is possible to construct a polynomial-time algorithm for any problem in NP. This would imply that P = NP.

In addition to time, another common resource of interest is space. Using the same framework, complexity classes can be defined based on the amount of space the algorithms use to solve problems.

DEFINITION G.2.10 *Let $t : \mathbb{N} \to \mathbb{N}$ be a function. An algorithm A decides a language L over some alphabet Σ in space $O(t(n))$ if for every string s of length n over Σ, the algorithm A using at most $O(t(n))$ space accepts s if $s \in L$ or rejects s if $s \notin L$. The language L is decided in space $O(t(n))$.*

DEFINITION G.2.11 *A language L is in* PSPACE *if there exists a polynomial-space algorithm A that decides L.*

DEFINITION G.2.12 *A language L is in* PSPACE-complete *if*

1. $L \in$ PSPACE*, and*

2. *if $L' \in$* PSPACE*, then $L' \leq_p L$.*

If L satisfies the second condition, but not necessarily the first condition, then L is PSPACE-hard.

It can be easily shown that the relationship between the different complexity classes that have been defined in this section is as follows:

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

G.3 Completeness

When describing robotics algorithms in this book, several notions of "completeness" are used.

DEFINITION G.3.1 *An algorithm A is* complete *if in a finite amount of time, A always finds a solution if a solution exists or otherwise A determines that a solution does not exist.*

DEFINITION G.3.2 *An algorithm A is* resolution complete *if in a finite amount of time and for some small resolution step $\epsilon > 0$, A always finds a solution if a solution exists or otherwise A determines that a solution does not exist.*

DEFINITION G.3.3 *An algorithm A is* probabilistically complete *if the probability of finding a solution, if a solution exists, converges to* 1*, when the running time approaches infinity.*

Complete algorithms include many common algorithms such as A^*, shortest-path, scheduling problems, etc. Resolution complete algorithms have to approximate a continuous measure by discretizing it at small steps. Ray tracing from graphics algorithms and sampling-based planning algorithms that use a grid representation of the configuration space are examples of resolution complete algorithms. Probabilistic completeness guarantees that given enough time, a solution will be found (if a solution exists). If a solution does not exist, the algorithm may not be able to necessarily detect this fact and thus runs forever. In practice, probabilistic algorithms terminate and declare failure if an upper bound on the amount of time the algorithm could use has passed and a solution has not been found. The basic Probabilistic RoadMap planner (`PRM`) is an example of a probabilistically complete algorithm. Such an algorithm trades completeness for efficiency; in many cases, for the same problem, a probabilistically complete algorithm will find a solution faster (if a solution exists) than a complete algorithm.