Computational Problems

Last Updated: January 27th, 2022

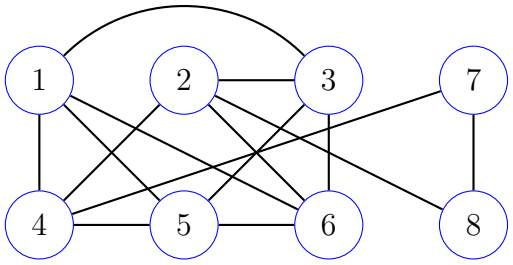
1 Introduction

Informally, when we think of a problem, we think of a situation that is in need of a solution. In computer science we think of a problem as not just one situation, but rather a collection of situations that share a common underlying theme. Each situation is referred to as a **problem instance**, and represents a concrete example of the general problem.

Example 1.1. Consider the problem called `Prime`, where a problem instance is a positive integer $n \geq 2$, and the solution we seek is an answer, `yes` or `no`, to the question of whether n is a prime number. For example, the solution to 11 is `yes`, while the solution to 36 is `no`. `Prime` is an example of what is called a **decision problem** because the solution to each instance is a Boolean value: $1 =$ `yes`, $0 =$ `no`. For a decision problem, a **positive instance** (respectively, **negative instance**) is any instance for which the solution is 1 (respectively, 0). Here, 11 is a positive instance, while 36 is a negative instance. □

As we'll see in Chapter 3 and subsequent chapters, decision problems are fundamental to the theory of computing.

Example 1.2. Consider the problem called `Clique`, where a problem instance is a simple graph $G = (V, E)$, and the solution we seek is a subset $C \subseteq V$ of vertices of maximum size such that, for every $u \in C$ and $v \in C$, $(u, v) \in E$. In other words, for every pair of vertices in C there is an edge $e \in E$ that is incident with u and v. `Clique` is called an **optimization problem**, since it calls for finding a structure (in this case a subset) that optimizes (in this case maximizes) an objective function (the objective is to make the set as large as possible) subject to the constraint that every pair of set members must form an edge in G. The graph below is an instance of `Clique`. Provide a solution to this problem instance.



Solution.

Example 1.3. Consider the problem called `Sort`, where a problem instance is an array a of integers, and the solution we seek is another array b whose members are the same members of a, but in sorted order. For example, an instance of sort is $(-4, 28, -11, 33, 18, -12, -15)$, and its solution is

$$(-15, -12, -11, -4, 18, 28, 33).$$

□

2 Problem size parameters

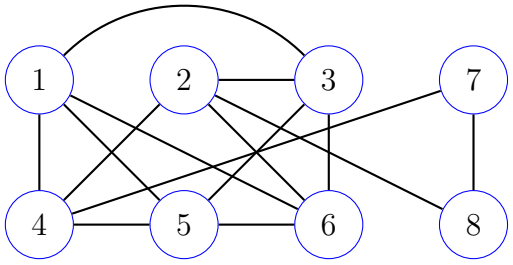
Of course, problems of all kinds are what drive computer science, especially the theory of computing. Given a problem L the first stop for L on the theory train is in the land of computability where we make sure that it can be solved by a computer. If the answer is "no", then computability theory is responsible for providing a proof of this claim. This can seem a very challenging task, especially since there are an infinite number of possible programs that can be written, and how can we be sure that none of them can solve L? On the other hand, if the answer is "yes", then L is sent on its way with a souvenir algorithm and continues on its journey to the land of computational complexity where it attempts to find its proper place in society amongst different classes of problems. It's here that the intrinsic complexity of L gets studied, meaning that one attempts to determine functions $s(n)$ and $t(n)$ that represent tight lower bounds on the amount of memory and time needed for an algorithm to solve the problem, respectively. In other words, if `P` is a computer program that implements some algorithm for solving L, then `P` would require $\Omega(s(n))$ amount of memory and $\Omega(t(n))$ amount of time to solve a problem instance having size n. Here, n is what is referred to as a **size parameter** for L, i.e. a parameter that is associated with the **size** of a given problem instance, i.e. the number of bits that are needed to represent the instance. In complexity theory, for the sake of simplicity, we work with size parameters that represent the size of a problem instance rather than the actual instance size.

Example 2.1. Consider an algorithm that sorts an array of integers. Suppose each integer can be represented using k bits. Then the size of a problem instance is equal to k times the number of integers to be sorted. Let n be a parameter that represents the number of integers in the array. Then the problem size is equal to nk, and the memory and time functions can be written as $s(n, k)$ and $t(n, k)$. Now, if the algorithm is independent of the number of bits used to represent each integer, then we may drop the k parameter. In this case we have $s(n)$ and $t(n)$ is written. For example, the Quicksort algorithm assumes an O(1) comparison operation for integers that is independent of the number of bits representing each integer. In this case n suffices as size parameter. On the other hand, the Radixsort algorithm requires to each bit of each integer, in which case n and k are appropriate size parameters. Therefore, choosing appropriate size parameters for a problem may be algorithm dependent.

Example 2.2. Recall that a graph is a pair of sets $G = (V, E)$, where V is the vertex set, and E is the edge set whose members have the form (u, v), where $u, v \in V$. A **graph algorithm** takes as input a graph, and computes some property of the graph. The most commonly used size parameters for describing the memory and time requirements of such an algorithm are $n = |V|$, called the **order** of G, and $m = |E|$, called the **graph size** of G. Also, we usually do not include size parameters for representing the memory required to store a single vertex or edge, since graph-algorithm steps are usually independent of the data stored in each vertex and edge. In other words, regardless of whether each vertex stores an integer, or an `Employee` data structure, the algorithm steps, and hence the big-O growth of the memory and running time, remain the same.

Example 2.3. Consider an algorithm that takes as input a positive integer p, and determines whether or not p is prime. Since a positive integer p can be repreented using $\lfloor \log p \rfloor + 1$ bits, we use size parameter $m = \log p$ to represent the problem size.

Example 2.4. The `Vertex Cover (VC)` decision problem is the problem of deciding if a simple graph $G = (V, E)$ has a vertex cover of size $k \geq 0$, for some integer k. In other words does G have a subset C of k vertices for which every edge $e \in E$ is incident with at least one vertex in C? Show that $(G, k = 5)$ is a positive instance of `VC` and provide appropriate size parameters for `VC`.



Solution.

3 Algorithms and their Analysis

Given computational problem A, an **algorithm** that solves A is a description of a step-by-step process whose execution on an instance x of A has the effect of realizing a solution for x. Before accepting an algorithm as providing a valid mean for solving A and using it in practice, we must first provide sufficient analysis of the algorithm. The analysis has the goal of establishing the following properties of the algorithm.

Correctness It must be established that the algorithm performs as advertised. In other words, for each problem instance x of A the algorithm produces a correct solution to x.

Complexity Bounds must be provided on either the amount of time required to execute the algorithm, or the amount of space (i.e. memory) used by the algorithm as a function of the size of an input instance.

Implementation Appropriate data structures must be identified that allow for the algorithm to achieve a desired time or space complexity.

The correctness of an algorithm is sometimes immediate from its description, while the correctness of others may require clever mathematical proofs.

As for complexity, in this course we are primarily concerned with the big-O growth of the worst-case **running time** of an algorithm. Of course, the running time T will be a function of the size parameters for the problem, and will have a big-O growth that is proportional to the number of algorithm steps that is required to execute the algorithm on some instance. Recall that we define the **size** of a problem instance as the minimum number of bits needed to represent the instance.

Note that we use the growth terminology from the big-O lecture to describe the running time of an algorithm. For example, if an algorithm has running time $T(n) = \mathrm{O}(n)$, then the algorithm is said to have a linear running time. In general we may write

$$\Omega(f(n)) \leq T(n) \leq O(g(n)),$$

where $f(n)$ represents the **best-case** running time, and $g(n)$ denotes the worst-case. In case $f(n) = \Theta(g(n))$, then we may write $T(n) = \Theta(g(n))$, meaning that the algorithm always requires on the order of $g(n)$ steps, regardless of the size-n input.

Another time complexity measure of interest is the **average-case running time** T_{ave}, which is obtained by taking the average of the running times for inputs of a given size.

Finally, the runnng time of an algorithm is dependent on its implementation. For example, a graph algorithm may have running time $T(m, n) = \mathrm{O}(m^2)$ using one implementation, and $T(m, n) = \mathrm{O}(m \log n)$ using another. For this reason complexity analysis is inseparable from implementation analysis, and it often requires the use of both basic and advanced data structures for achieving a desired running time. On the other hand, correctness analysis is usually independent of implementation.

Exercises

1. An instance of the `Perfect` decision problem is an integer $n \geq 1$, and the problem is to decide if n is the sum of each of its proper divisors. For example, 6 is perfect since $6 = 3 + 2 + 1$. i) Determine whether 36 is perfect. ii) Provide one or more size parameters for `Perfect` to properly represent the size of a `Perfect` instance.

2. Suppose an algorithm for deciding `Perfect` requires $O(n^2)$ steps, where n is the problem instance. Use your answer to the previous exercise and the big-O growth terminology provided in this chapter to describe the algorithm's running time.

3. A **permutation** of the numbers $1, \ldots, n$ is an ordering of these numbers. It may also be thought of as a one-to-one correspondence p from the set $\{1, \ldots, n\}$ to the set $\{1, \ldots, n\}$, where $p(i)$ equals the value at position i. For example, the permutation

$$p = (4 \;\; 5 \;\; 3 \;\; 1 \;\; 2)$$

may be viewed as the one-to-one correspondence from $\{1, 2, 3, 4, 5\}$ to itself in which $p(1) = 4$, $p(2) = 5$, $p(3) = 3$, $p(4) = 1$, and $p(5) = 2$. Then if p and q are two permutations of $1, \ldots, n$, then we may define the multiplication of p with q, written $p \circ q$ as the composite function $(p \circ q)(i) = p(q(i))$. Since the composite of two one-to-one correspondences is also a one-to-one correspondence, it follows that $p \circ q$ results in another permutation. Using p defined above, and

$$q = (1 \;\; 3 \;\; 5 \;\; 4 \;\; 2),$$

compute $p \circ q$.

4. Prove that permutation multiplication is associative, i.e.

$$p \circ (q \circ r) = (p \circ q) \circ r.$$

Hint: apply both sides (i.e. functions) to some input i, $1 \leq i \leq n$, and show that both sides produce the same output.

5. Since the previous two exercises define what it means to multiply two permutations, and that multiplication is associative, we may thus raise a permutation to some power. For example, if p is a permutation, then $p^3 = p \circ p \circ p$ is also a permutation. Compute p^3 for the permutation defined in Exercise 3.

6. An instance of decision problem `Perm Power` takes as input two permutations p and q of $1, \ldots, n$, and a nonnegative integer t. The problem is to decide if $q = p^t$. i) Determine whether $(p, q, 4)$ is a positive instance, where

$$p = (3 \;\; 1 \;\; 4 \;\; 5 \;\; 2)$$

and

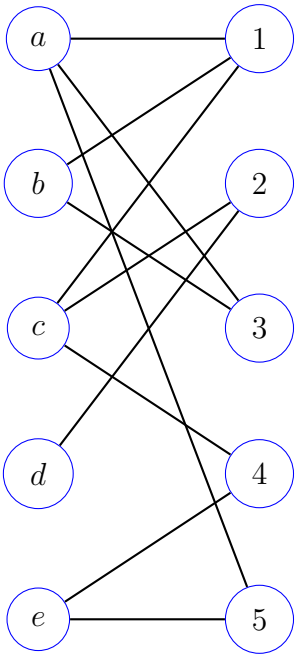
$$q = (2 \;\; 5 \;\; 1 \;\; 3 \;\; 4).$$

ii) Provide one or more size parameters for `Perm Power` to properly represent the size of a `Perm Power` instance.

7. Suppose an algorithm for deciding `Perm Power` requires $O(n \log t)$ steps. Use your answer to the previous exercise and the big-O growth terminology provided in this chapter to describe the algorithm's running time.

8. Consider the optimization problem called `Coloring` where an input instance is a simple graph $G = (V, E)$, and a solution is a minimum (in size) set of colors that can be used to color each vertex in V in such a way that no two adjacent vertices are assigned the same color. i) Find a minimum set of colors for the graph provided in Example 2.4. ii) Provide one or more size parameters for `Coloring` to properly represent the size of a `Coloring` instance.

9. An instance of the `Perfect Matching` decsion problem is a bipartite graph $G = (V_1, V_2, E)$, where $|V_1| = |V_2| = n$. The problem is to decide if G has a **matching** of size n, in other words, a set of n edges, no two of which are incident with the same vertex. i) Decide whether the bipartite graph provided below is a positive instance of `Perfect Matching`. ii) Provide one or more size parameters for `Perfect Matching` to properly represent the size of a `Perfect Matching` instance.



10. The `3-Dimensional Matching (3DM)` decision problem takes as input three sets A, B, and C, each having size n, along with a set S of triples of the form (a, b, c) where $a \in A$, $b \in B$, and $c \in C$. We assume that $|S| = m \geq n$. The problem is to decide if there exists a subset of n triples (called a *matching*) from S for which each member from $A \cup B \cup C$ belongs to exactly one of the triples. i) Decide if (A, B, C, S) is a positive instance of `3DM`, where $A = \{a, b, c\}$, $B = \{1, 2, 3\}$, $C = \{x, y, z\}$, and

$$S = \{(b, 2, y), (b, 1, z), (a, 3, z), (c, 2, y), (a, 2, y), (a, 3, y), (c, 3, x), (c, 1, z), (b, 1, x)\}.$$

ii) Provide one or more size parameters for `3DM` to properly represent the size of a `3DM` instance.

Exercise Solutions

1. i) The divisors of 36 are 1,2,3,4,6,9,12, and 18.

$$1 + 2 + 3 + 4 + 6 + 9 + 12 + 18 = 55 \neq 36,$$

and so 36 is a negative instance of `Perfect`. ii) Since the input to perfect is a single integer n, an appropriate size parameter is $m = \log n$, since this rougnly the number of bits needed to represent n.

2. The algorithm has exponential running time since $n^2 = (2^{\log n})^2 = 4^{\log n}$ which grows exponentially with respect to input parameter $\log n$.

3. We have

$$p \circ q = (4 \;\; 3 \;\; 2 \;\; 1 \;\; 5).$$

For example,

$$(p \circ q)(3) = p(q(3)) = p(5) = 2,$$

and so the third number in the permutation is 2.

4. For arbitrary i, we have

$$(p \circ (q \circ r))(i) = p((q \circ r)(i)) = p(q(r(i))).$$

Similarly,

$$((p \circ q) \circ r)(i) = (p \circ q)(r(i)) = p(q(r(i))).$$

5. We have

$$p^2 = (1 \;\; 2 \;\; 3 \;\; 4 \;\; 5),$$

which is the **identity permutation**. Therefore, $p^3 = p \circ p^2 = p$.

6. i) We have

$$p^2 = (4 \;\; 3 \;\; 5 \;\; 2 \;\; 1),$$

and

$$p^4 = p^2 \circ p^2 = (2 \;\; 5 \;\; 1 \;\; 3 \;\; 4) = q.$$

Therefore $(p, q, 4)$ is a positive instance of `Perm Power`. ii) The size parameters are n, the size of each permutation, and $\log t$, the size of input t.

7. The algorithm has quadratic running time since, if we substitute $m = \log t$, we get $O(nm)$ which is quadratic.

8. i) The least number of colors needed is four (why not three?): red (r), blue (b), green (g), and yellow (y). We have the following coloring of vertices.

Vetex	**Color**
1	r
2	r
3	g
4	g
5	b
6	y
7	r
8	b

ii) Size parameters: $m = |E|$, $n = |V|$ since an instance of `Coloring` is a graph.

9. i) Positive instance with perfect matching: $\{(a, 1), (b, 3), (c, 4), (d, 2), (e, 5)\}$ ii) Size parameters: $m = |E|$, $n = |V|$ since an instance of `Perfect Matching` is a graph.

10. i) Positive instance with 3DM $\{(a, 3, z), (b, 1, x), (c, 2, y)\}$. ii) Size parameters: $m = |S|$ and $n = |A| = |B| = |C|$.