

MATH 409 LECTURE 3 COMPLEXITY OF ALGORITHMS

REKHA THOMAS

Both the traveling salesman problem and the perfect matching problem are examples of (families of) **problems**. We will denote a problem by the letter P . An **instance** of a problem is a specific member of the family — one that is specified by actual data.

Example 1. (i) The problem of finding the optimal tour through all the state capitals in the United States is an instance of the TSP. In this case, $n = 50$ and the locations of the capitals and the distances between them are known quantities which gives us coordinates for the points v_1, \dots, v_{50} and actual numbers for the costs c_{ij} , $1 \leq i, j \leq 50$, $i \neq j$. (ii) The problem $\max \{5x_1 + 4x_2 : x_1 + x_2 \leq 3, x_1, x_2 \geq 0\}$ is an instance of a linear programming problem.

An **algorithm** to solve a problem P is a procedure (a sequence of steps) that inputs instances of the problem and outputs their solutions. For instance, the problem of finding the rank of a matrix can be solved using the algorithm of Gaussian elimination. The simplex method is an algorithm for solving a linear program. Note that both these problems have several algorithms that solve them.

The **size** of an instance of a problem is the length of the encoding of the instance. Roughly, it is the number of bits needed to store the instance in the computer. The set of sizes of problem instances is a subset of the set of non-negative integers, denoted as \mathbb{N} .

The efficiency of an algorithm can be measured by computing upper bounds for its **running time** on problem instances. To make this independent of the particular computer that solves the problem we need a mathematical notion of running time. To do this, we define the running time of an algorithm A as a function f_A from \mathbb{N} to \mathbb{N} such that $f_A(s)$ equals the number of **elementary operations** algorithm A takes on an instance of size s . The operations of addition, multiplication, division, subtraction and comparison (of two numbers) are examples of elementary operations. There are several models for measuring running times of algorithms. We discuss two, on an example.

Example 2. Suppose our problem P is to compute the distance between two points $p = (p_1, p_2, \dots, p_n)$ and $q = (q_1, q_2, \dots, q_n)$. An algorithm A to solve this problem is a computer code that will compute

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}.$$

An instance of this problem is given by specific data such as say $n = 2$ and $p = (3, 2)$ and $q = (5, 1)$.

(1) **The arithmetic model.** This is a simple model in which all elementary operations are assumed to take unit time. The size of an instance of the above problem P is $2n$ as the instance is specified by the $2n$ numbers that give the coordinates of p and q . In this model, we treat all numbers as being equal and count each number as contributing one **bit** to the length of encoding of the instance. The algorithm A needs the following elementary operations:

- n subtractions: to compute $(p_i - q_i), i = 1, \dots, n$
- n multiplications : to compute $(p_i - q_i)^2, i = 1, \dots, n$
- $n - 1$ additions: to sum the squares $(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2$
- one square root : to get $\sqrt{(p_1 - q_1)^2 + \dots + (p_n - q_n)^2}$

This gives a total of $2n + (n - 1) + 1 = 3n$ elementary operations. Thus we say that the running time of our algorithm A is bounded above by the polynomial $3n$ and $f_A(2n) = 3n = \frac{3}{2}(2n)$. Rewriting, we have $f_A(s) = \frac{3}{2}s$. Since $f_A(s)$ is a polynomial, A is a **polynomial-time** algorithm. In fact, its running time is a linear function of the input size and we say the algorithm is **linear** in the size of the input.

Computing the running times (complexity) of algorithms needs us to compare functions. More precisely, if there are two algorithms A_1 and A_2 for problem P , we'd like to compare the running time functions $f_{A_1}(s)$ and $f_{A_2}(s)$ to decide which algorithm is superior (if that's possible). We are also only interested in how the function f_A grows as s grows. Small instances can often be solved well by most algorithms. The real strength of an algorithm surfaces when we look at how it behaves on instances of large size. The following definitions allow us to compare functions.

Definition 3. If $f(s)$ and $g(s)$ are two positive real valued functions on \mathbb{N} , the set of non-negative integers, we say that

- (i) $f(n) = O(g(n))$ if there is a constant $k > 0$ such that $f(n) \leq k \cdot g(n)$ for all n greater than some finite n_0 ,
- (ii) $f(n) = \Omega(g(n))$ if there is a constant $k > 0$ such that $f(n) \geq k \cdot g(n)$

for all n greater than some finite n_0 ,

(iii) $f(n) = \Theta(g(n))$ if there are two constants $k, k' > 0$ such that $k' \cdot g(n) \leq f(n) \leq k \cdot g(n)$ for all n greater than some finite n_0 .

Thus if $f(n) = O(g(n))$, then the two functions may be incomparable for initial values of n (values before $n = n_0$) but “eventually” (i.e., after n_0), $f(n)$ is bounded above by a positive multiple of $g(n)$.

Example 2 continued. The function $3n$ is $O(n)$ since we can choose $k \geq 3$ which will make $3n \leq k \cdot n$ for all n . In this example, $n_0 = 0$. Thus we say that the algorithm A in the above example has running time $O(n)$. In fact the running time is also $\Omega(n)$ and $\Theta(n)$. It is also $O(n^2)$, $O(e^n)$ etc, but usually we want the smallest bound possible and so it's better to say that the running time is $O(n)$. What would be your choices for k to establish that $3n$ is also $O(n^2)$ and $O(e^n)$?

Definition 4. An algorithm A for problem P is said to be a **polynomial time** algorithm if the running time function $f_A(s)$ is a polynomial in s , where s is the size of the problem.

The complete enumeration algorithm for the TSP was $O(n!)$ which is far from being polynomial time.

The “big $O \setminus \Omega \setminus \Theta$ ” notation allows us to be approximate with calculating sizes of problems and measuring running times, in that it allows us to ignore contributions from constants and other factors that do not eventually affect the behavior of the function. For instance $3s^3 + e^s$ eventually looks like e^s and we can say that $3s^3 + e^s = O(e^s)$.

Example 2 continued. Let us recast our example above in this light. We saw that the problem size was $3n$ which is $O(n)$. Similarly the running time was $2n$ which is also $O(n)$. Thus we could simply write that the running time function of the above algorithm is $f_A(n) = O(n)$ and still get a good sense of how this running time function behaves on large instances of the problem.

Exercise 5. This exercise is all about estimating $n!$. We will use \log to denote the natural logarithm \ln so as to not confuse all the n 's that appear below.

(a) Look up the formula for $\int \log x \, dx$.

(b) Prove that $\log n! = \Theta(n \log n)$.

Hint: $\log n! = \sum_{x=1}^n \log x$. Use some simple integral calculus to bound this sum from above and below.

(c) From the computation in (b) deduce that $e \left(\frac{n}{e}\right)^n \leq n! \leq e \left(\frac{n+1}{e}\right)^{n+1}$.

- (d) What does (c) imply about the complexity of $n!$ itself? i.e., use O, Ω, Θ notation to express $n!$ in the best way you can.
- (e) Stirling's formula says that $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ and in fact, it can be shown that $n! > \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$. How does this compare to your result in (d)?