Data Structures
Spring 2009

Kishore Kothapalli

Chapter 1

Introduction

1.1 A Gentle Introduction

This course tries to understand the techniques of information storage and processing with applications to several problems of interest. We argue that information processing cannot be viewed as independently from information storage, efficient information processing requires efficient information storage. An analogy for this is as follows. Consider an office desk with books and papers scattered all around the desk. To locate any particular item, it would generally be very difficult. Instead, if the items were organized into files, it would be easy (hopefully) to locate the required items. In a similar fashion, data that is properly organized lends itself to easy manipulation. The right kind of storage of course does depend on the kind of operations one is interested in. As an analogy consider searching for a telephone number by name. Not so long ago, people used a printed telephone directory to do this operation. Imagine a reasonably sized city with about half a million telephone users. If we are searching for the telephone number of say Mr S. V. Rao, then it is not an easy task to pore over half a million records. You will immediately notice that a telephone directory is arranged in a sorted order so that names starting with A appear first followed by names starting with B, and so on. So, we start searching for S. V. Rao starting from more than half-way through the directory. It is not too difficult in this case to search for the required number. Similar examples abound in daily life, some of which we shall formalize in this course.

The above paragraph suggests that one should consider the information and the intended manipulations before choosing how to organize the information. Information organization is often referred to as data structures. This course is about the study of popular data structures and their advantages to information processing. We start with number systems to motivate using a real life example. We then cover the array based data structures including implementing stacks and queues using arrays. Stacks and queues have important applications including table calculators, for example. Given the limitations of array based data structures, we then cover pointer based data structures such as linked lists with applications to polynomials and matrices. We then cover specialized data structures such as hash tables and trees. This is followed by a treatment of graphs and operations on graphs using suitable data structures. Advanced concepts in data structures such as amortized tables, data structures for strings, and querying based data structures shall be covered during the final weeks of the semester.

Emphasis will also be on the formal analysis of the operations for standard parameters such as time and space apart from correctness of the proposed mechanisms. A small introduction to the tools required for the analysis shall be presented in the class. A full list of topics and the tentative schedule is available at the course web site at http://cstar.iiit.ac.in/ kkishore/.

An important component of this course is the importance attached to porlbem solving and programming. Problems shall be solved using the suggested data structures and then implemented using the C programming

language. The weightage attached to various components is available at the course web site.

1.2 Number Systems

Since ancient times, the ability to represent numbers and perform arithmetic operations on numbers played an important role. Various ancient civilizations have used different styles of number formats including the Hindu/Arabic system and the Roman system. We are also aware of the unary system and the binary system the latter due to its use in modern day digital computers. Let us briefly recall these systems before considering operations on numbers in these systems.

1.2.1 The Hindu/Arabic System

This is also known as the decimal system and is presently in vogue in written communication. The system used 10 digits 0, 1,$\cdots$, 9. For instance three hundred and sixty seven is written as 367. This is a positional number system with the rightmost position having a weight of 1, the next position a weight of 10, and so on.

1.2.2 The Roman System

This is a system using the symbols I, V, X, L, C, and M. This is not a positional system. As an example, 24 is written as XXIV.

1.2.3 The Unary and the Binary System

In the former, the only symbol used is I and a number is represented by writing a number of I's equal to its value. For example, 7 is written as IIIIIII. It can be noticed that this system is too laborious for large numbers such as those beyond 20 or so.

In the latter, two symbols 0 and 1 are used. This is a positional system with weights of 1, 2, 4, and so on. For instance, 55 is written as 110111. The latter is popular because of its use in the present generation computers.

1.2.4 Operations on Numbers

Let us consider the basic arithmetic operations such as addition and multiplication. In the case of addition using the Hindu system, we have the notion of a carry so that if the sum of any two digits in a position with weight of 10^k will create a carry of 1 with a weight of 10^{k+1}. The same is concept holds true for positional systems such as the binary system, with the only difference being the weights. In the unary system, the addition of numbers x and y will result in a number with $x + y$ I's.

Addition in the Roman system is much more complicated. To add two numbers in the Roman system, one starts by writing the numbers side by side. For example, to add 32 and 53, write XXXII and LIII side by side as XXXII LIII. Then, arrange these letters so that the numerals are in decreasing order. We get: LXXXIIIII. The five Is can be simplified by writing them as V. So we get LXXXV. This process is continued till no simplification is possible. Care has to be taken when adding numbers such as 64, written as LXIV.

Similar observations hold in the case of multiplication as multiplication can be treated as repeated addition in its basic form. So one way to multiply Roman numbers is to multiply the larger number by the smaller number. For example, to multiply 123 by 52, we start by writing the numbers as CXXIII and LII. Now, multiply the larger number CXIII by L. This is done by first multiplying by 10, which means promoting the symbols and then duplicating each symbol five times. Simplification can be done to reduce the number to

the standard form. Then, multiplication by 2 is simply duplicating each symbol twice. Simplifying, we get the answer.

⟦***HW: Implement addition and multiplication in the above number systems***⟧.

The above reading suggests that the method of number representation does have a bearing on the ease of the operation. Hence, even with simple operations such as addition and multiplication, a better way to organize data helps in performing the operation efficiently. In the rest of this chapter, we will use the decimal/binary system. We will carry forward this lesson to other operations.

1.3 Further Operations on Numbers

The previous section outlined the need for a good representation for performing an operation. In this section, we focus on the efficiency of the operation itself. Consider for example the problem of computing the greatest common divisor of two positive integers. Recall that the greatest common divisor of two positive integers m and n is the integers p such that p divides both m and n and is the largest of all such common divisors of m and n.

Given two integers a and b, the largest integer that divides both a and b is called the greatest common divisor of a and b and is denoted $\gcd(a, b)$. One can find the $\gcd$ of a pair of numbers by exhaustively listing the divisors of each of the numbers and then picking the largest common element in these sets. For example, if $a = 24$ and $b = 42$, the set of divisors of 24 is $\{1, 2, 3, 4, 6, 8, 12, 24\}$ and the set of divisors of 42 is $\{1, 2, 3, 6, 7, 14, 21\}$. The common divisors are $1, 2, 3, 6$. Hence, the greatest common divisor of 24 and 42 is 6.

While the above example is easy to follow for small numbers, it becomes extremely impractical for large numbers to list all their divisors. Another attack is to use the fundamental theorem of arithmetic and write $a = p_1^{a_1} \cdot p_2^{a_2} \cdot \cdots \cdot p_k^{a_k}$ and $b = p_1^{b_1} \cdot p_2^{b_2} \cdots p_r^{b_r}$ for primes $p_1, \cdots, p_k$ and integers $a_1, \cdots, a_k$ and $b_1, \cdots, b_r$. Then, it holds that $\gcd(a, b) = p_1^{\min\{a_1,b_1\}} \cdot p_2^{\min\{a_2,b_2\}} \cdots p_k^{\min\{a_k,b_k\}}$.

However, both the above methods require one to factorize a and b. Integer factorization is a computationally intractable problem. Hence, both the methods fail to work well in practice. In what is these days considered a breakthrough achievement of ancient mathematics, the Greek mathematician Euclid gave an efficient algorithm to compute the $\gcd$ of two integers. We will now study Euclid's algorithm.

1.4 Euclid's Algorithm

The algorithm of Euclid is based on the following important lemma.

Lemma 1.4.1 *Let a, b be two positive integers. Let q and r be integers such that $a = b \cdot q + r$. Then,*

$$\gcd(a, b) = \gcd(b, r).$$

Proof. We argue that the common divisors of a and b are also common divisors of b and r. To this end, we shall show that every common divisor of a and b also divides r, and vice-versa.

Let d divide both a and b. Then, d also divides $a - bq = r$. Similarly, let e divide both b and r. Then, e divides also $bq + r = a$. □

The above lemma leads us to the following algorithm for finding the $\gcd$ of two integers a, b.

Algorithm GCD-Euclid(a, b)
$x := a$, $y := b$;

while $(y \neq 0)$
$r := x \bmod y$;
$x := y$;
$y := r$;
end-while
End-Algorithm.

TODO: Running time of Euclid's algorithm: Related to Fibinacci numbers, $O(\log b)$ if $b > a$ and $O(\log a)$ if $a > b$.

The above example suggests that efficient processing of information is in general essential. One can see the difference in the runtime of the naive algorithm to that of Euclid's algorithm on a present day computer. The difference is remarkable for especially large numbers. Hence, one seeks efficient information processing strategies and efficient information storage strategies. In the following we'll study one more kind of operations on integers.

1.5 Modular Arithmetic

Of particular interest in some settings is operations on integers modulo a given positie integer. This is called as modular arithmetic and is known to possess several important properties and applications. Examples include the RSA cryptosystem and the Chinese remainder theorem, among others. Let us formally define the concept before we explore operations in modulo arithmetic.

Definition 1.5.1 *Let m, a, and b be positive integers. Then we say that a is congruent to b modulo m, written $a \equiv b(\bmod m)$, if and only if m divides $a - b$. We also use the notation $a \not\equiv b \bmod m)$ if m does not divide $a - b$.*

Let us denote the remainder obtained by dividing a by m as $a \bmod m$. It follows from the above definition that $a \equiv b(\bmod m)$ if and only if $a \bmod m = b \bmod m$. For example, $13 \equiv 38(\bmod 5)$ since $13 \bmod 5 = 38 \bmod 5 = 3$. In addition, the following is easy to see.

Theorem 1.5.2 *Let m be a positive integer. Then integers a and b are positive integers if and only if there exists an integer k such that $a = b + km$.*

Further properties of the above definition are the following. In the following, we let m be a positive integer.

- If $a \equiv b(\bmod m)$ and $c \equiv d(\bmod m)$, then $a + c = b + d(\bmod m)$ and $ac \equiv bd(\bmod m)$.
- Modular addition can be defined as follows. $(a + b) \bmod m = ((a \bmod m) + (b \bmod m) \bmod m)$. For example, if $a = 20$ and $b = 17$ and $m = 32$, then $a + b \equiv 5 \bmod m$.
- Modular multiplication can be defined as follows. $ab(\bmod m) = ((a \bmod m)(b \bmod m) \bmod m)$. As an example, if $a = 12$ and $b = 8$ and $m = 20$, then $ab \equiv 16(\bmod m)$.

Apart from modular addition and multiplication, another important operation is that of modular exponentiation. Modular exponentiation has applications to several cryptographic systems. In general, given integers b, n, and m, the operation involves finding $b^n(\bmod m)$. Thus, we seek the reaminder of b^n divided by m.

A straight-forward way to compute $b^n(\mathrm{mod}m)$ is to raise b to its nth power and then take the modulo with respect to m. However, this quickly becomes impractical even for moderate values of b, n, and m. It will be a programming excercise to implement this naive solution.

A better solution can be obtained by observing the following. Notice that for $k > 1$, as soon as b^k exceedes m, one can simply take the modulo with respect to m before proceeding further. This reduces the scale of the numbers we have to encounter during this computation. This solution does not however save the number of computations.

A further improvement can be obtained by noticing the following. Let $n = (n_{r-1}n_{r-2}\cdots n_1n_0)_2$. Then,

$$b^n = b^{(n_{r-1}n_{r-2}\cdots n_1n_0)_2} = b^{n_{r-1}\cdot 2^{r-1}+n_{r-2}\cdot 2^{r-2}+\cdots+n_1\cdot 2+n_0\cdot 1} = b^{n_{r-1}\cdot 2^{r-1}} \cdot b^{n_{r-2}\cdot 2^{r-2}} \cdot \ldots b^{n_1\cdot 2} \cdot b^{n_0\cdot 1}.$$

Moreover, if $n_i = 0$ for any $0 \leq i \leq r-1$, then $b^{n_i\cdot 2^i} = 1$. Hence, it has no effect on $b^n(\mathrm{mod}m)$. Thus, we need to evaluate terms of the form $b^{n_i 2^i}$ whenever $n_i = 1$. For example, if $b = 4$, $n = 9 = (1001)_2$ and $m = 11$, we compute $4^{(1001)_2}(\mathrm{mod}11) = 4^{1\cdot 2^4} \bmod 11 \cdot 4^{1\cdot 1}(\mathrm{mod}11)$. A final improvement can be made if we notice that to compute $b^{2^r}(\mathrm{mod}m)$ we can simply compute $b(\mathrm{mod}m)$, and $b^2 \equiv (b \cdot b)(\mathrm{mod}m)$, $b^4 \equiv (b^2(\mathrm{mod}m) \cdot (b^2(\mathrm{mod}m)$ and so on for r iterations. The above ideas lead us to the following algorithm for modular exponentiaion.

Algorithm ModularExponentiaion(b, n, m)
  let $n = n_{r-1}n_{r-2}\cdots r_1r_0$;
  answer $= 1$;
  temp = $b \bmod m$;
  for $i = 0$ to $r-1$ do
  begin
    if $n_i == 1$ then
      answer = (answer · temp) $(\mathrm{mod}m)$;
  end
temp = (temp · temp) $(\mathrm{mod}m)$;
return answer;
End-Algorithm.

A detailed example follows. TODO.

1.6 Large Integers

The above techniques are useful when the numbers used can be be stored in a machine word. In that case, one can use machine instructions such as ADD, MULTIPLY to perform additions and multiplcation of integers. However, there are important applications which require integers whose size exceeds that of machine words. Adding and multiplying such large integers requires special care. TO DO.

1.7 Lab, Tutorial, and Homework

The tutorial shall discuss algorithms for adding and multiplying Roman numbers. Additionally, modular inverse of integers to be discussed.

The lab would implement the discussed algorithms, at least the basic ones such as addition of Roman numerals, and Euclid's GCD algorithm.

Homework problems:

- Roman number multiplication
- Modular inverse
- Modular Exponentiation, hence implement RSA for small numbers.
- Timing profile of Roman number multiplication and decimal system multiplication.

Acknowledgements

Pranav initiated the discussion on number representation as an elementary data structure.