Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Fall 2017

Instructor: Dr. Sateesh Mane

October 4, 2017

1 Lecture 1

This is a collection of "useful algorithms" and lessons in organizing calculations to save on time and/or memory.

1.1 GCD greatest common divisor

For two positive integers a and b, there is a simple and efficient algorithm to compute their greatest common divisor (gcd). It dates from classical Greece and we call it "Euclid's algorithm" although Euclid himself never claimed to invent it. *But he wrote it down, and his work has survived.* It is possible that other ancient civilizations also knew it (China, India, Egypt, etc.?) but we do not know. Without loss of generality, suppose $a > b$ (and $b > 0$), else just swap a and b. Then we proceed recursively.

- Calculate $c = a \% b$ (the remainder after integer division).
- If $c == 0$, then gcd $= b$ (because b divides a).
- If $c == 1$, then gcd $= 1$.
- Else proceed recursively and compute gcd(b, c).

The process must terminate because the integers are positive and always get smaller and eventually must reach 0 or 1.

- Comment: if b is zero, the Excel returns the gcd$(a, 0)$ equals a. This makes sense: if b is zero, then a divides a and it also divides 0 (the remainder is zero in both cases). A special case to note.

1.2 Conversion decimal to binary or hex, etc.

Computers store numbers internally as binary digits (or hexadecimal). But they display the numbers on screen in decimal format. We also input numbers (in online forms, etc.) in decimal format. Hence conversion of digits from decimal to/from binary or hex is a frequently performed operation. Hence it is essential to do it efficiently. For simplicity we consider only positive numbers.

1.2.1 Decimal integer to binary

We begin with integers. We shall consider fractions later. Proceed recursively:

- Divide the number by 2 (integer division) and store the remainder.
- Divide the quotient by 2 (integer division) and store the remainder.
- Repeat until the quotient reaches zero. This must happen and the algorithm will terminate.
- Read the remainders in reverse order, that is the binary representation of the decimal number.
- Example: $a = 19$

19	
9	1
4	1
2	0
1	0
0	1

- Hence $19_{\text{dec}} = 10011_{\text{bin}}$.

1.2.2 Decimal integer to hex

The fundamental algorithm is exactly the same, but we perform integer division by 16 (not 2). Proceed recursively:

- Example: $a = 30$

30	
1	14
0	1

- The new twist here is that the remainder takes values from 0 through 15. Hence we need a lookup array for hex digits $\texttt{hex\_digits} = \{\texttt{0},\texttt{1},\texttt{2},\texttt{3},\texttt{4},\texttt{5},\texttt{6},\texttt{7},\texttt{8},\texttt{9},\texttt{A},\texttt{B},\texttt{C},\texttt{D},\texttt{E},\texttt{F}\}$. (Hence the output is alphanumeric.) In the above example, the lookup array says $14 \to E$ so

$$30_{\text{dec}} = 1E_{\text{hex}} .$$

1.2.3 Decimal integer to base b

Obviously the same algorithm works for any base $b \geq 2$. If $b > 10$ then we require a lookup table to output alphanumeric digits, but that is not a problem.

1.2.4 Decimal fraction to binary

The algorithm is essentially the same as above, but now we multiply by 2 at each step. Also, we keep the fraction and record the integer part in an array. It is simplest to explain with some examples. For ease of reading, it is convenient to store the integers on the left

- Example: $a = \frac{1}{4}$

	$\frac{1}{4}$
0	$\frac{1}{2}$
1	0

- For fractions, we read the array from the top down. Hence $(\frac{1}{4})_{\text{dec}} = 0.01_{\text{bin}}$.
- Example: $a = \frac{1}{3}$

	$\frac{1}{3}$
0	$\frac{2}{3}$
1	$\frac{1}{3}$
0	$\frac{2}{3}$
1	$\frac{1}{3}$
$\vdots$	

- **A problem with this algorithm is that the decimal or binary representation of a fraction may not terminate.**
- For example $(\frac{1}{3})_{\text{dec}} = 0.333\ldots$ and from the above we see that $(\frac{1}{3})_{\text{bin}} = 0.010101\ldots$
- Hence we need to impose a cutoff on the maximum number of digits to output.

1.2.5 Decimal fraction to hex

Obviously we follow the same procedure but we multiply by 16 instead of 2.

- Example: $a = \frac{1}{4}$

	$\frac{1}{4}$
4	0

- Hence $(\frac{1}{4})_{\text{dec}} = 0.4_{\text{hex}}$ $(= \frac{4}{16})$.

- A more complicated example is $a = 0.11$

	0.11
1	0.76
12	0.16
2	0.56
8	0.96
15	0.36
5	0.76
12	0.16
$\vdots$	

- The expansion repeats and continues indefinitely. We require a lookup table. Hence

$$0.11_{\text{dec}} = (0.1C28F5C28F5\ldots)_{\text{hex}}\,.$$

1.3 Karatsuba multiplication

- Suppose we wish to multiply two numbers (integers), which both have n digits (to keep the analysis simple).
- For many years it was believed that n^2 multiplications of digits was required.
- But in 1960 Karatsuba (who was 23 years old), found a faster way.
- Let the numbers be x and y, and let the base (of the number system) be B. Choose some integer m (the value is up to you). Then we can write
$$x = x_1 B^m + x_0\,, \qquad y = y_1 B^m + y_0\,.$$
Then the product is
$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0) = x_1 y_1 B^{2m} + (x_1 y_0 + x_0 y_1) B^m + x_0 y_0\,.$$
This requires four multiplications.
- But Karatsuba observed that we can do it in only three multiplications as follows. Compute $z_0 = x_0 y_0$ and $z_2 = x_1 y_1$ and save the values. Then
$$z_1 = x_1 y_0 + x_0 y_1 = (x_1 + x_0)(y_1 + y_0) - z_0 - z_2\,.$$
The last step requires only *one* multiplication, because z_0 and z_2 have been precomputed.
- Karatsuba's algorithm requires only three multiplications, at the cost of a few extra additions.
- For large n, the computational complexity of Karatsuba's algorithm is of order $n^{\log_2 3} \simeq n^{1.585}$, which is faster than $O(n^2)$.
- You can read more about Karatsuba's algorithm online.
- **Important lesson: There is room for improvement even for basic tasks.**

Did I forget to lecture this in class?

1.4 Fibonacci numbers

- Leonardo of Pisa (in Italy) was better known as Fibonacci.
- The Fibonacci numbers are defined via the recurrence relation

$$F_n = F_{n-1} + F_{n-2}\,.$$

- The initial values are $F_1 = 1$ and $F_2 = 1$.
- This obviously lends itself to recursion. A very simple recursive algorithm to calculate F_n can be coded in only a few lines as follows:

```
int Fib_recursive(int n)
{
   if ((n == 1) || (n == 2)) return 1;
   return (Fib_recursive(n-1) + Fib_recursive(n-2));
}
```

- ***** This is terrible *****
- It is wasteful of both memory and time.
- To illustrate, consider how the above algorithm woul compute F_6:

$$\begin{aligned} F[6] &= F[5] + F[4] \\ &= (F[4] + F[3]) + (F[3] + F[2]) \\ &= ((F[3] + F[2]) + (F[2] + F[1])) + ((F[2] + F[1]) + F[2]) \\ &= (((F[2] + F[1]) + F[2]) + (F[2] + F[1])) + ((F[2] + F[1]) + F[2])\,. \end{aligned}$$

- Observe how much duplication of calculation there is and how much waste of memory.
- The computational complexity rapidly gets wose for larger values of n.

- It is better to allocate an array internally and to compute the numbers from $i = 3$ up to n. The code fragment is as follows. (An array F of sufficient size must be allocated. Relevant variables must be declared and initialized.)

  ```
  int i;
  F[1] = 1;
  F[2] = 1;
  for (i = 3; i <= n; ++i) {
     F[i] = F[i-1] + F[i-2];
  }
  int result = F[n];

  (relase allocated memory)

  return result;
  ```

- Notice that there is no duplication of calculations.
- The computation time is $O(n)$ for arbitrary n.
- **Important lesson: The short simple algorithm is not always the best.**