CS 3510 Algorithms 8/29/2023

Lecture 3: Arithmetic

Lecturer: Abrahim Ladha *Scribe(s): Saahir Dhanani*

Arithmetic involves some basic operations on numbers, such as addition, subtraction, multiplication, exponentiation, and so on. We will formalize these operations by discussing some algorithms for arithmetic.

1 Representations of Numbers

In order to discuss algorithms that operate on numbers, we must discuss how numbers are represented in this context. Like any other object, numbers have an encoding. On computers, this encoding is a sequence of bits. In the context of algorithms, we are concerned with the runtime in terms of the *size* of the input, not the input itself. Given n bits, the largest number we can represent is $2^n - 1$. In other words, given a number, x, it takes $n = \log x$ bits to represent the number on a computer. Note that here we don't particularly care about the base. Recall the change of base formula

$$\log_b n = \frac{\log_a n}{\log_a b}$$

The only difference between $\log_{10}$ and $\log_2$ is multiplication by a constant. All logs have the same asymptotics, but are scaled differently. We only use base 10 because we have 10 fingers. We will use base 2 most commonly because computers also do.

2 Algorithms for Addition

The first arithmetic operation you learn is always addition.

1. Basic Addition Algorithm:
   This is the addition algorithm you learned in grade school
   Input: Two n-bit numbers: x, y
   Output: The sum $x + y$
   Example: input: $x = 1000101$ and $y = 1110111$

$$\begin{array}{rr} & 1\,000\,101(69) \\ + & 1\,110\,111(119) \\ \hline & 10\,111\,100(188) \end{array}$$

   For each bit, we loop right to left and do either 2 or 3 steps, depending on if we have to carry or not. This means that it takes $\leq 3n + 1$ steps to complete this algorithm, which is $\mathcal{O}(n)$.

Can we do better? It is not possible, since it takes $\Omega(n)$ time to even read the input and write the down the answer. Since we have a lower and upper bound, we know that this algorithm takes $\Theta(n)$ time. Even though computers may be able to add 32-bit numbers in one step, constant time, adding larger numbers will be a function of the number of 32 bit blocks its represented as, so it really is asymptotically linear time anyway.

3 Algorithms for Multiplication

The next obvious step after addition is moving on to multiplication. We give four multiplication algorithms.

1. Basic Multiplication Algorithm:
   This is the kind of multiplication you would've learned in grade school.
   Input: Two n-bit numbers: x, y
   Output: The product xy
   Example: input: $x = 1101$ and $y = 1011$

$$\begin{array}{r}
1101 \\
\times\ 1011 \\
\hline
1101 \\
1101 \\
0000 \\
1101 \\
\hline
10001111
\end{array}$$

   Note that, if we consider the bits of y as $y_n ... y_1$, then

$$xy = x(y_n 2^{(n-1)} + y_{n-1} 2^{(n-2)} + ... + y_2 2^1 + y_1 2^0) = x y_n 2^{(n-1)} + x y_{n-1} 2^{(n-2)} + ... + x y_2 2 + x y_1$$

   As each y_i is a bit, we just add together up to n shifts of x. Think of y_i as a bit selecting whether we add that shift or not. Since there are n possible additions, and each one takes $\mathcal{O}(n)$ time, the total runtime for this multiplication algorithm is $\mathcal{O}(n^2)$.

   Is it possible to do better than this? Al-Khwarizmi noticed that there was a recursive algorithm for multiplication.

2. Recursive Multiplication Algorithm:
   Input: Two n-bit numbers: x, y
   Output: The product xy
   Pseudocode :

```
def mult(x, y):
    ## Base Case
    if y == 0: return 0
    # Split problem in half, recursively call mult
    z = mult(x, floor(y/2))
    if y is even:
        return 2*z
    if y is odd:
        return 2*z + x
```

   Let us now prove the correctness of this algorithm: First, the base case is good because anything times 0 is still 0. Next, if y is even, then $z = x\frac{y}{2}$, which means that $2z = xy$, as desired. Otherwise, if y is odd, then $z = x\frac{y-1}{2}$, which means that $2z + x = x(y-1) + x = xy$. The running time of the algorithm can be analyzed without the use of the master theorem. Notice that each recursive call shifts y by 1 bit (recall that shifting the bits left/right is the same as multiplying/dividing by powers of two). Since there are n bits, there will be n recursive calls. In the worst case, each call performs an n bit addition. Therefore, with n calls, each taking $\mathcal{O}(n)$ time, the overall running time is $\mathcal{O}(n^2)$ This running time isn't any better than that of the previous algorithm! Is it possible to do better than this? Lets try a divide and conquer approach.

3. Divide and Conquer Multiplication Algorithm:
   Input: Two n-bit numbers: x, y
   Output: The product xy

   Here, we will represent x and y slightly differently, by representing them with an upper and lower half (left and right half).

$$x = \begin{array}{|c|c|}\hline x_L & x_R \\ \hline\end{array} = (2^{n/2})x_L + x_R$$
$$y = \begin{array}{|c|c|}\hline y_L & y_R \\ \hline\end{array} = (2^{n/2})y_L + y_R$$

   Note that it is quite easy to compute x_L and x_R given x and vice versa since we can just shift the bits to get the desired part. If x, y each have n bits, then x_L, x_R, y_L, y_R each have $n/2$ bits. With this new representation, we can rewrite the computation of

xy as follows:

$$\begin{aligned}
xy &= \\
((2^{n/2})x_L + x_R)((2^{n/2})y_L + y_R) &= \\
= 2^n x_L y_L + 2^{n/2} x_L y_R + 2^{n/2} x_R y_L + x_R y_R &= \\
= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R &
\end{aligned}$$

Thus, we have broken down the computation of xy into a few smaller subproblems. Four multiplications, each of numbers of size $n/2$. We can now translate this equation into an algorithm

Pseudocode :

```
def mult(x, y):
    compute n
    ## Base case:
    if n == 1: return x * y
    xL = x >> (n/2)
    xR = x % (2^(n/2)) #Note, more than one way to do this
    yL = y >> (n/2)1
    yR = y % (2^(n/2))
    LL = mult(xL, yL)
    LR = mult(xL, yR)
    RL = mult(xR, yL)
    RR = mult(xR, yR)
    return (LL << n) + ((LR + RL) << (n/2)) + RR
```

Each component here is drawn directly from the equation above. Since this is a divide and conquer algorithm, we can use the master theorem to derive the runtime. To do this, we need to write out the recurrence: $T(n) = aT(n/b) + O(n^d)$. In this case, $a = 4$ since there are 4 recursive calls to mult, $b = 2$ since the size of the input to each call of mult is halving, and $d = 1$ since we are doing a linear amount of work per call. This gives us the following recurrence: $T(n) = 4T(n/2) + O(n)$. Now, we can compare d with $\log_b a$: $1 < \log_2 4$ which means that the time complexity is $\mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$.

4. We got the same time complexity again! At this point, after three attempts, one might think that we cannot do better than $\mathcal{O}(n^2)$. Kolmogorov conjectured that that multiplication has a lower bound of $\Omega(n^2)$, and set out to prove this at a large conference. Then Karatsuba came along and realized he could reduce the number of subproblems that result from xy. He did this by noticing that:

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$$

At first glance, this may seem like an increase in the number of multiplications, but recall that we have already computed $x_L y_L$ and $x_R y_R$ since they are used in other parts of the computation for xy. Now, using this revised component, we can create a new algorithm with less recursive calls!

Karatsuba Algorithm (A Smarter Divide and Conquer Multiplication Algorithm):
Pseudocode :

```
def mult(x, y):
    compute n
    ## Base case:
    if n == 1: return x * y
    xL = x >> (n/2)
    xR = x % (2^(n/2))
    yL = y >> (n/2)
    yR = y % (2^(n/2))
    LL = mult(xL, yL)
    RR = mult(xR, yR)
    MM = mult(xL + xR, yL + yR)
    return (LL << n) + ((MM - LL - RR)<< (n/2)) + RR
```

Once again, we can use the master theorem to do an analysis of the runtime, similar to the previous section. In this case, $a = 3$ since there are 3 recursive calls to mult, $b = 2$ since the size of the input to each call of mult is halving, and $d = 1$ since we are doing a linear amount of work per call. Note that the work done per call also includes computing the parameters for the recursive calls. This gives us the following recurrence: $T(n) = 3T(n/2) + O(n)$. Now, we can compare d with $\log_b a$: $1 < \log_2 3$ which means that the time complexity is $\mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.59})$.

If you generalize this Karatsuba style divconquer, you get the Toom-Cook algorithm. We've successfully broken the quadratic lower bound! We are unaware of any nontrivial lower bounds on multiplication. It is conjectured to be $\Omega(n \log n)$. Only recently (2020) was there an $\mathcal{O}(n \log n)$ algorithm found.

4 Algorithms for Exponentiation

Let's consider two algorithms for exponentiation.

Suppose we want to compute A^x where A could be a matrix or a number and x is a number. If we consider the bits of x as $x = x_n ... x_1$, then

$$A^x = A^{2^{n-1}x_n + 2^{n-2}x_{n-1} + ... + A^{2^1 x_2} + A^{2^0 x_1}} =$$
$$A^{2^{n-1}x_n} A^{2^{n-2}x_{n-1}} ... A^{2^1 x_2} A^{2^0 x_1}$$

For example, $A^{17} = A^{16+1} = A^{16}A$, where A^{16} can be calculated by repeatedly squaring. This leads us into the following algorithm:Basic Exponentiation Algorithm:
Pseudocode :

1.
```
def exp(A, x):
    ans = 1
    temp = A
    for i in [1...n]:
        if x_i == 1:
            ans *= temp
        temp *= temp
    return ans
```

   There are only n multiplications, however, repeated squaring of A grows very fast. We would like the multiplication to be of n-bit numbers, and not something growing large. Computing the run time of modular exponentiation is easier. Here, lets restrict ourselves to assuming A is a number.

2. Modular Exponentiation Algorithm:
   input: A, x, N where N is the modulus.
   output: $A^x \mod N$

   Pseudocode :

```
def modexp(A, x, N):
    if x == 0: return 1
    z = modexp(A, floor(x/2), N)
    if x is even:
        return z*z mod N
    else:
        return A * z*z mod N
```

   Let us now prove the correctness of this algorithm: First, the base case is good because anything to the power 0 is 1. Next, if x is even, then $(A^{x/2})^2 = A^x$, as desired. Otherwise, if x is odd, then $(A^{(x-1)/2})^2 A = A^{x-1}A = A^x$. The running time of the algorithm can be analyzed without the use of the master theorem. Notice that each recursive call shifts x by 1 bit (recall that shifting the bits left/right is the same as multiplying/dividing by powers of two). Since there are n bits of x, there will be n recursive calls. Each call performs an n bit multiplication in the worst case. Therefore, with n calls, each taking $\mathcal{O}(n^2)$ time, the overall running time is $\mathcal{O}(n^3)$. You can of course use Karatsuba multiplication to get a better bound of $\mathcal{O}(n^{2.59})$ but that doesn't make the $\mathcal{O}(n^3)$ wrong, just worse.

5 Algorithms for Matrix Multiplication

Let's conclude with algorithms for matrix multiplication. Here, we want to compute $C = AB$, where A and B are both matrices of size $n \times n$. In this model, we assume that multiplications and additions take unit time, and we are concerned more with the number of operations. The matricies may have n^2 elements each, but we parametrize the problem by their dimension, as a function of just n.

1.

2. Basic Matrix Multiplication Algorithm:
Input: Two $n \times n$ matrices, A and B
Output: C, where $C = AB$
The classic way to compute a matrix multiplication is by computing, for every element $C_{i,j}$ of C,
$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} B_{k,j}$$
The computation for each element of C takes a linear number of steps, and there n^2 elements, so this gives us a total runtime of $\mathcal{O}(n^3)$. Lets try a different approach.

3. Basic Divide and Conquer Matrix Multiplication Algorithm:

Consider splitting A and B into smaller submatrices:
$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Then
$$C = AB = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

We can see that computing C takes can be done with 8 subproblems, where the matrices in each call are of size $n/2$ by $n/2$. Once again, we can use the master theorem to do an analysis of the runtime. In this case, $a = 8$ since there are 8 recursive calls, $b = 2$ since the size of the input to each recursive call is halving, and $d = 2$ since there are multiple n^2 sized additions happening during the combination step. Each addition takes unit time, but there are additions for each element of the matrix, giving quadratically many additions. This gives us the following recurrence: $T(n) = 8T(n/2) + O(n^2)$. Using the master theorem, we can tell that the time complexity is $\mathcal{O}(n^{\log_2 8}) = \mathcal{O}(n^3)$. However, we can definitely definitely improve upon this runtime by being clever.

4. Better Divide and Conquer Matrix Multiplication Algorithm:
We will employ the same idea that was used to improve the first divide and conquer algorithm for multiplication: reduce the number of subproblems!

$$\begin{aligned}
M1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M2 &= (A_{21} + A_{22})(B_{11}) \\
M3 &= (A_{11})(B_{12} - B_{22}) \\
M4 &= (A_{22})(B_{21} - B_{11}) \\
M5 &= (A_{11} + A_{12})(B_{22}) \\
M6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

Then

$$C = AB = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

Using the master theorem, the runtime of this new approach is $\mathcal{O}(n^{\log_2 7})$ which is approximately $\mathcal{O}(n^{2.81})$. The current best known algorithm for matrix multiplication has a runtime of $\mathcal{O}(n^{2.37286})$. It is an open problem, and an active area of research to improve on this.