CS3510 L3 Arithmetic

Today we will begin discussing algorithms which involve numbers. Like any other object, numbers have an encoding, and we discuss runtimes of algorithms with respect to the sizes of the inputs. We usually denote a number by x and/or y and say it takes n bits. For some number x, what is its reasonable encoding size? It takes $\log(x) = n$ bits. Using n bits, you can represent numbers as large as $2^n - 1$, so to represent a number x takes approximately $\log(x)$ bits. Why don't we care about the base? Recall the change of base formula: $\log_b(n) = \log_a(n) / \log_a b$. The only difference between $\log_2 n$, $\log_{10} n$ and more 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 do.

What is the complexity of addition. A fairly simple atomic operation. Given two n bit numbers x, y. Lets do the grade school way of adding with carry

```
  1 1 1
  1000101
+ 1110111
----------
 10111100
```

for each bit of x, y, we do either 2 or 3 things. that means we take $\leq 3n + 1$ steps which is $O(n)$. Can we do better? No! It takes atleast $\Omega(n)$ to read the input and write down the answer. This matching upper and lower bound gives us an $\Theta(n)$ for addition. Even if computers can add 32 bit numbers in single steps, to add billion bit numbers will still take a linear amount of instructions.

It is natural for us to consider multiplication next. First the grade school way. for $y = y_n \dots y_1$ as bits, notice $xy = x(y_n 2^{n-1} + y_{n-1} 2^{n-2} + \dots y_2 2 + y_1) = x y_n 2^{n-1} + x y_{n-1} 2^{n-2} + \dots$ As each y_i is a bit, we just add together up to n shifts of x. Each addition takes $O(n)$ and there are up to n of them giving an algorithm of $O(n^2)$. The example from the book of 13×11:

```
  1 1 0 1      13
  1 0 1 1      11
-------------
  1 0 0 1
  1 1 0 1
0 0 0 0
1 1 0 1
-------------
1 0 0 0 1 1 1 1     143
```

Can we do better? Al Khwarizmi noticed the following recursive algorithm:

```
def mult(x, y)
    if y = 0 return 0
    z = mult(x, ⌊y/2⌋)
    if y even:
        return 2z
    if y is odd:
        return 2z + x.
```

First lets prove correctness by induction on y. base case is good. If y is even, then $z = x \cdot y/2$, so $2z = xy$ as desired. If y is odd, then $z = x \cdot (y-1)/2$. so $2z + x = x(y-1) + x = xy$. We don't need the master theorem to analyze this. Each recursive call shifts y by 1 bit, so there are n recursive call. Each call in worst case performs one n bit addition, so we see that our algorithm is $O(n^2)$.

Lets do a third algorithm for multiplication for x, y n bit numbers, we may represent them as an upper and lower half. $x = [x_L][x_R] = 2^{n/2} x_L + x_R$

$y = [y_L][y_R] = 2^{n/2} y_L + y_R$.

computing x from x_L, x_R and vice versa is quick. This division of the numbers into halves gives us a div conquer algorithm. Notice that

$$xy = (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) =$$

$$2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

```
def mult(x, y)
    compute n; if n = 1 return xy
    x_L = x >> (n/2)        x_L = (x >> (n/2)) << (n/2)
    x_R = x % 2^(n/2)
    y_L = y >> (n/2)
    y_R = y % 2^(n/2)
    LL = mult(x_L, y_L)
    LR = mult(x_L, y_R)
    RL = mult(x_R, y_L)
    RR = mult(x_R, y_R)
    return (LL << n) + ((LR + RL) << (n/2)) + RR
```

What is this runtime? $T(n) = 4T(n/2) + O(n)$

$\log_2 4 = 2 > 1 = d$ so $T(n) = O(n^2)$

Btw, instead of comparing d ☐ $\log_b a$, compare b^d ☐ a

We have now seen three algorithms for ~~positive~~ multiplication which cannot do better than $O(n^2)$. We may conjecture that multiplication requires $\Omega(n^2)$. Kolmogorov thought so, and set out to prove it in a very long conference. On an early day, a young Karatsuba noticed the following:

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

This gives an algorithm with one less recursive call, at the cost of being smarter with your other ones!

```
def mult(x,y):
    compute n
    if n=1 return xy
    x_L =
    x_R =
    y_L =
    y_R =
    LL = mult(x_L, y_L)
    RR = mult(x_R, y_R)
    MM = mult(x_L + x_R, y_L + y_R)
    return (LL << n) + ((MM - LL - RR) << (n/2)) + RR
```

Our runtime has improved to $T(n) = 3T(n/2) + O(n)$.

$\log_2 3 \approx 1.59 \Rightarrow d = 1$ so $O(n^{\log_b a}) = O(n^{1.59}) = T(n)$.

We ~~broke~~ broke the quadratic lower bound! We are unaware of any ~~conjectured~~ lower bound on multiplication. It is conjectured to be $\Omega(n \log n)$. Only recently (in 2020) was an $O(n \log n)$ time algorithm found.

5

Lets consider two algorithms for exponentiation. Suppose we want to compute A^x where A a matrix or number and x a number. We can do like $x = x_n \ldots x_1$ and

$$A^x = A^{2^{n-1}x_n + 2^{n-2}x_{n-1} + \ldots} = A^{2^{n-1}x_n} A^{2^{n-2}x_{n-1}} A^{2} \ldots$$

for example, $A^{17} = A^{16+1} = A^{16} A$, where A^{16} can be calculated by repeated squaring.

```
def exp(A, x)
    ans = 1
    temp = A
    for i in [1..n]
        if x_i = 1
            ans *= temp
        temp = (temp)^2
    return ans.
```

Although you only perform only n multiplications, repeated squaring of A grows very fast. I want you to compute the runtime of this one we can give an easier to analyze algorithm for modular exponentiation

```
def modexp(A, x, N)          // returns A^x mod N
    if x=0 return 1
    z = modexp(A, ⌊x/2⌋, N)
    if x is even
        return z^2 mod N
    else
        return A*z^2 mod N
```

if x is even, then $(A^{x/2})^2 = A^x$

if x is odd, then $(A^{(x-1)/2})^2 * A = A^{x-1} A = A^x$.

for x is n bits, there are n recursive calls. Each call does a bit multiplication so worst case is $O(n^3)$.

Lets conclude today with matrix multiplication. The parameter is n, and we are concerned with the number of operations to compute $C = AB$ where A, B are $n \times n$ matrices. The classic way is for each C_{ij} of C to compute $C_{ij} = \sum_{k=0}^{n} A_{ik} B_{kj}$. Each element takes a linear number of steps and there are n^2 elements so this gives us an $O(n^3)$ time.

Lets try a divconquer strategy.

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

A $\quad$ B $\quad$ C

computing C takes 8 calls, each size $n/2$ several n^2 sized additions gives us $T(n) = 8T(n/2) + O(n^2)$ by MT is $O(n^{\log_2 8}) = O(n^3)$. unimpressive

$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$ $\quad O(n^{\log_2 7}) \approx$
$M_2 = (A_{21} + A_{22})B_{11}$
$M_3 = A_{11}(B_{12} - B_{22})$ $\quad O(n^{2.81})$
$M_4 = A_{22}(B_{21} - B_{11})$
$M_5 = (A_{11} + A_{12})B_{22}$ $\quad$ subcubic!
$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

$O(n^{2.37287}) \rightarrow O(n^{2.37286})$ took 30 pages