Divide-&-Conquer – Symbolic Polynomial and Large Integer Multiplication

Textbook Reading:

- Section 7.1 Divide-and-Conquer Paradigm, pp. 284-286.
- Section 7.2 Symbolic Polynomial Multiplication, pp. 286-291.
- Section 7.3 Multiplication of Large Integers, pp. 291-292.

Divide-and-Conquer Paradigm

procedure *Divide_and_Conquer*(I,J) **recursive**

Input: I (an input to the given problem)

Output: J (a solution to the given problem corresponding to input I)

```
    if I ∈ Known then
            assign the a priori or ad hoc solution for I to J
    else
            Divide(I,I_1,...,I_m)      //m may depend on the input I
            for i ← 1 to m do
                    Divide_and_Conquer(I_i,J_i)
            endfor
            Combine(J_1,...,J_m,J)
    endif
end Divide_and_Conquer
```

Examples of Divide-&-Conquer Algorithms we've already seen

- Binary Search
- Interpolation Search
- Mergesort
- Quicksort

Symbolic Multiplication of Polynomials

$P(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0 \Leftrightarrow [a_0 , a_1 , \ \ldots a_{n-1}]$
$Q(x) = b_{n-1}x^{n-1} + \ldots + b_1x + b_0 \Leftrightarrow [b_0 , b_1 , \ \ldots b_{n-1}]$
$R(x) = P(x)Q(x) = c_{2n-2}x^{2n-2} + \ldots + c_1x + c_0 \Leftrightarrow [c_0 , c_1 , \ \ldots c_{2n-2}]$

$R(x) = (a_0 + a_1x + \ldots + a_{n-1}x^{n-1})\ (b_0 + b_1x + \ldots + b_{n-1}x^{n-1})$

$(a_0 * b_0) + (a_0 * b_1 + a_1 * b_0)\ x + (a_0 * b_2 + a_1 * b_1 + a_2 * b_0)\ x^2$
$+ (a_0 * b_3 + a_1 * b_2 + a_2 * b_1 + a_3 * b_0)\ x^3 + \ldots$
$+ (a_0 * b_{n-1} + a_1 * b_{n-2} + a_2 * b_{n-3} + \ldots + a_{n-1} b_0)\ x^{n-1}$
$+ (a_1 * b_{n-1} + a_2 * b_{n-2} + a_3 * b_{n-3} + \ldots + a_{n-1} b_1)\ x^n$
$+ \ldots. + (a_{n-2} * b_{n-1} + a_{n-1} * b_{n-2})\ x^{2n-3} + (a_{n-1} * b_{n-1})\ x^{2n-2}$

so that

$c_0 = a_0 * b_0\ ,\ \ c_1 = a_0 * b_1 + a_1 * b_0\ ,\ \ c_2 = a_0 * b_2 + a_1 * b_1 + a_2 * b_0,$
$c_3 = a_0 * b_3 + a_1 * b_2 + a_2 * b_1 + a_3 * b_0\ ,\ c_4 = \ldots\ ,$
$c_{n-1} = a_0 * b_{n-1} + a_1 * b_{n-2} + a_2 * b_{n-3} + \ldots + a_{n-1} b_0$
$c_n = a_1 * b_{n-1} + a_2 * b_{n-2} + a_3 * b_{n-3} + \ldots + a_{n-1} b_1\ , \ldots,$
$c_{2n-3} = a_{n-2} * b_{n-1} + a_{n-1} * b_{n-2}\ ,\ \ c_{2n-2} = a_{n-1} * b_{n-1}$

Straightforward algorithm

The sum for c_i is called a *convolution*. The naïve algorithm is to simply compute all convolutions. The number of multiplications involved is

$$1 + 2 + \ldots + n - 2 + n - 1 + n + n - 1 + n - 2 + \ldots + 2 + 1$$
$$= 2*(1 + 2 + \ldots + n - 1) + n$$
$$= 2 * ((n - 1)*n/2) + n = (n - 1)n + n = n^2.$$

Thus, the worst-case, average, and best-case complexities are all given by

$$W(n) = A(n) = B(n) = n^2.$$

Divide Step

Setting $d = \lceil n/2 \rceil$, we divide the set of coefficients of the polynomials in half

$P_1(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \ldots + a_1x + a_0 \leftrightarrow [a_0, a_1, \ldots a_{d-1}]$

$P_2(x) = a_{n-1}x^{n-d-1} + a_{n-2}x^{n-d-2} + \ldots + a_{d+1}x + a_d \leftrightarrow [a_d, a_{\mathrm{d}+1}, \ldots a_{n-1}]$

$Q_1(x) = b_{d-1}x^{d-1} + b_{d-2}x^{d-2} + \ldots + b_1x + b_0 \leftrightarrow [b_0, b_1, \ldots b_{d-1}]$

$Q_2(x) = b_{n-1}x^{n-d-1} + b_{n-2}x^{n-d-2} + \ldots + b_{d+1}x + b_d \leftrightarrow [b_d, b_{\mathrm{d}+1}, \ldots b_{n-1}]$

Then,

$P(x) = P_1(x) + x^d P_2(x)$, $Q(x) = Q_1(x) + x^d Q_2(x)$, so that

$$R(x) = P(x)Q(x) = (P_1(x) + x^d P_2(x))\,(Q_1(x) + x^d Q_2(x))$$

$$= P_1(x)\,Q_1(x) + x^d\,(P_1(x)Q_2(x) + P_2(x)Q_1(x)) + x^{2d}\,P_2(x)Q_2(x)$$

Combine Step

$R(x) = P(x)Q(x) = (P_1(x) + x^d P_2(x)) (Q_1(x) + x^d Q_2(x))$

$= P_1(x) Q_1(x) + x^d (P_1(x)Q_2(x) + P_2(x)Q_1(x)) + x^{2d}P_2(x)Q_2(x)$

This involves 4 multiplications of polynomials of half the size. Note that multiplying by x^d and x^{2d} does not involve any coefficient multiplications.

Recursion Relation for Worst-Case Complexity

The combine involves 4 recursive calls with polynomials of half the size. The initial condition involves multiplying constant polynomials of size 1, i.e., $a_0 * b_0$. Thus, assuming $n = 2^k$

$$W(n) = 4W(n/2), \text{ initial condition } W(1) = 1.$$

Using Repeated Substitution to Solve

$$
\begin{aligned}
W(n) = 4W(n/2) = 4(4W(n/4)) &= 4^2W(n/4) \\
&= 4^2(4W(n/8) = 4^3W(n/8) \\
&\;\;\vdots \\
&= 4^kW(n/2^k) \\
&= 4^kW(1) \\
&= 4^k = (2^k)^2 = n^2
\end{aligned}
$$

Bad Performance

Our Divide-and-Conquer has worst-case complexity

$$W(n) = n^2 .$$

But this is no better than the straightforward algorithm, and even worse because we have the overhead of recursion!

Solution: We need a better combine operation.

Improved Combine Operation

Observe that

$$\boldsymbol{P_1(x)Q_2(x) + P_2(x)Q_1(x) = (P_1(x) + P_2(x))(Q_1(x) + Q_2(x)) - P_1(x)\, Q_1(x) - P_2(x)\, Q_2(x),}$$

Substituting we obtain

$R(x) = P_1(x)\, Q_1(x) + x^d\, (P_1(x)Q_2(x) + P_2(x)Q_1(x)) + x^{2d}P_2(x)Q_2(x)$

$= P_1(x)\, Q_1(x) + x^d\, ((P_1(x) + P_2(x))(Q_1(x) + Q_2(x)) - P_1(x)Q_1(x) - P_2(x)Q_2(x)) + x^{2d}P_2(x)Q_2(x)$

which involves 5 multiplications instead of 4, but only 3 distinct multiplications.

Using this formula, $R(x)$ can be computed using only 3 **distinct** multiplications of polynomials of half the size, i.e., only 3 recursive calls in the Divide-and-Conquer algorithm based on this combine operations.

Pseudocode for Divide-&-Conquer Algorithm

function *PolyMult1*(P,Q,n) **recursive**

Input: $P(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$, $Q(x) = b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ (polynomials)

n (a positive integer)

Output: $P(x)Q(x)$ (the product polynomial)

if $n = 1$ **then**

  return(a_0b_0)

else

  $d \leftarrow \lceil n/2 \rceil$

  Split($P(x)$,$P_1(x)$,$P_2(x)$)

  Split($Q(x)$,$Q_1(x)$,$Q_2(x)$)

  $R(x) \leftarrow$ *PolyMult1*($P_2(x)$,$Q_2(x)$,d)

  $S(x) \leftarrow$ *PolyMult1*($P_1(x) + P_2(x)$,$Q_1(x) + Q_2(x)$,d)

  $T(x) \leftarrow$ *PolyMult1*($P_1(x)$,$Q_1(x)$,d)

  return($x^{2d}R(x) + x^d(S(x) - R(x) - T(x)) + T(x)$)

endif

end *PolyMult1*

Recursion Relation for Worst-Case Complexity

Algorithm involves 3 recursive calls with polynomials of half the size. The initial condition involves multiplying constant polynomials of size 1, i.e., $a_0 * b_0$. Thus, assuming $n = 2^k$

$$W(n) = 3W(n/2), \text{ initial condition } W(1) = 1.$$

Using Repeated Substitution to Solve

$$
\begin{aligned}
W(n) = 3W(n/2) = 3(3W(n/4)) &= 3^2 W(n/4) \\
&= 3^2(3W(n/8) = 3^3 W(n/8) \\
&\;\;\vdots \\
&= 3^{\mathrm{k}} W(n/2^k) \\
&= 3^{\mathrm{k}} W(1) \\
&= 3^{\mathrm{k}} \\
&=((2^{\log_2 3})^k \\
&=(2^k)^{\log_2 3} = n^{\log_2 3}
\end{aligned}
$$

Complexity of Improved Divide-&-Conquer Algorithm

The worst-case complexity is

$$W(n) = n^{\log_2 3} \approx n^{1.5849}$$

This is a significant improvement over quadratic complexity of the straightforward algorithm.

Multiplication of Polynomials of Different Input Sizes

In practice, we often encounter the problem of multiplying two polynomials $P(x)$ and $Q(x)$ of different input sizes m and n, respectively.

If $m < n$, then the straightforward solution is to augment $P(x)$ with $n - m$ leading zeros.

This would be quite inefficient if n is significantly larger than m.

Better solution

Partition $Q(x)$ into blocks of size m. For convenience, we assume n is a multiple of m, that is, $n = km$ for some positive integer k. We let $Q_i(x)$ be the polynomial of degree m given by

$$Q_i(x) = b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + \cdots + b_{(i-1)m+1}x + b_{(i-1)m}, \quad i \in \{1, \ldots, k\}.$$

Then

$$Q(x) = Q_k(x)x^{m(k-1)} + Q_{k-1}(x)x^{m(k-2)} + \cdots + Q_2(x)x^m + Q_1(x).$$

so that

$$P(x)Q(x) = P(x)Q_k(x)x^{m(k-1)} + P(x)Q_{k-1}(x)x^{m(k-2)} + \cdots + P(x)Q_1(x).$$

Pseudocode for efficient algorithms for multiplying polynomial of possibly different sizes

function *PolyMult2*($P(x)$,$Q(x)$,m,n)
Input: $P(x) = a_{m-1}x^{m-1} + ... + a_1x + a_0$, $Q(x) = b_{n-1}x^{n-1} + ... + b_1x + b_0$
n, m (positive integers) //$n = km$ for some integer k
Output: $P(x)Q(x)$ (the product polynomial)

ProdPoly(x) ← 0 {initialize all coefficients of *ProdPoly*(x) to be 0}
for i ← 1 **to** k **do**
$Q_i(x) \leftarrow b_{im-1}x^{m-1} + b_{im-2}x^{m-2} + ... + b_{i\,m-m+1}\,x + b_{\mathrm{i}m-m}$
endfor
for i ← 1 **to** k **do**
ProdPoly(x) ← *ProdPoly*(x) + $x^{(i-1)m}$*PolyMult1*($P(x)$,$Q_i(x)$,m)
endfor
return(*ProdPoly*(x))

end *PolyMult2*

Complexity Analysis

The complexity of *PolyMult1* for multiplying two polynomials of size *m* is $\Theta((m^{\log_2 3})$. Since *PolyMult2* invokes *PolyMult1* a total of *k* = *n*/*m* times, each time with input polynomials of size *m*, it follows that the complexity of *PolyMult2* is

$$\Theta(km^{\log_2 3}) = \Theta\left(\frac{nm^{\log_2 3}}{m}\right) = \Theta\left(nm^{\log_2 (3/2)}\right)$$

Large Integer Multiplication

Consider two n-digit decimal integers

$$U = u_{n-1}u_{n-2} \ldots u_1u_0, \;\; V = v_{n-1}v_{n-2} \ldots v_1v_0$$

The algorithm you learned in school for multiplying these integers involves n^2 digit multiplications.

Large Integer Multiplication cont'd

Observe that for $x = 10$

$$U = u_{n-1}x^{n-1} + \ldots + u_1 x + u_0,$$
$$V = v_{n-1}x^{n-1} + \ldots + v_1 x + v_0$$

Thus, integer multiplication can be implemented in an analogous way to polynomial multiplication, yielding an algorithm for integer multiplication that has worst-case complexity

$$W(n) = n^{\log_2 3} \approx n^{1.5849}$$

Pseudocode for Large Integer Multiplication

function *PolyMult1*(P,Q,n) **recursive**

Input: $U = u_{n-1}u_{n-2} \dots u_1u_0$, $V = v_{n-1}v_{n-2} \dots v_1v_0$ (n-digit decimal integers)

n (a positive integer)

Output: UV (the product)

if $n = 1$ **then**

return($u_0 * v_0$)

else

$d \leftarrow \lceil n/2 \rceil$

$Split(U,U_1,U_2)$

$Split(V,V_1,V_2)$

$R(x) \leftarrow PolyMult1(U_2,V_2,d)$

$S(x) \leftarrow PolyMult1(U_1 + U_2,V_1 + V_2,d)$

$T(x) \leftarrow PolyMult1(U_1,V_1,d)$

return $(10^{2d}R(x) + 10^{d}(S(x) - R(x) - T(x)) + T(x))$

endif

end *PolyMult1*

An Even Faster Algorithm

- We presented an $O(n^{\log_2 3})$ algorithm for symbolic polynomial multiplication and large integer multiplication.
- When we cover the Discrete Fourier Transform, we will see that the Fast-Fourier Transform (*FFT*), another Divide-&-Conquer algorithm, can be applied to solve both of these important problems in time $O(n \log n)$, an amazing result!

Teacher: Why are you doing your multiplication on the floor?

Student: You told me not to use tables.