4. INTEGER MULTIPLICATION II

Raveen de Silva, r.desilva@unsw.edu.au
office: K17 202
Course Admin: Anahita Namvar, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 3, 2021

Table of Contents

Basics revisited: how do we multiply two numbers?

- The primary school algorithm:

```
        X X X X  <- first input integer
      * X X X X  <- second input integer
        -------
        X X X X  \
      X X X X     \ O(n^2) intermediate operations:
    X X X X       / O(n^2) elementary  multiplications
  X X X X        /    + O(n^2) elementary additions
---------------
X X X X X X X X  <-  result of length 2n
```

- Can we do it faster than in n^2 many steps??

The Karatsuba trick

- Take the two input numbers A and B, and split them into two halves:

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad \underbrace{XX\ldots X}_{\frac{n}{2}}\underbrace{XX\ldots X}_{\frac{n}{2}}.$$
$$B = B_1 2^{\frac{n}{2}} + B_0$$

- AB can now be calculated as follows:

$$\begin{aligned} AB &= A_1 B_1 2^n + (A_1 B_0 + A_0 B_1) 2^{\frac{n}{2}} + A_0 B_0 \\ &= A_1 B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0) 2^{\frac{n}{2}} + A_0 B_0. \end{aligned}$$

The Karatsuba trick

- We have saved one multiplication, now we have only three: A_0B_0, A_1B_1 and $(A_1 + A_0)(B_1 + B_0)$.

- The runtime satisfies

$$T(n) = 3T\left(\frac{n}{2}\right) + c\,n,$$

and the Master Theorem gives

$$T(n) = \Theta\left(n^{\log_2 3}\right) = O\left(n^{1.585}\right).$$

Table of Contents

Generalising Karatsuba's algorithm

- Can we do better if we break the numbers in more than two pieces?
- Let's try breaking the numbers A, B into 3 pieces; then with $k = n/3$ we obtain

$$A = \underbrace{XXX\ldots XX}_{k \text{ bits of } A_2}\ \underbrace{XXX\ldots XX}_{k \text{ bits of } A_1}\ \underbrace{XXX\ldots XX}_{k \text{ bits of } A_0}$$

i.e.,

$$A = A_2\, 2^{2k} + A_1\, 2^k + A_0$$
$$B = B_2\, 2^{2k} + B_1\, 2^k + B_0$$

Generalising Karatsuba's algorithm

Multiplying out, we get

$$
\begin{aligned}
AB = & \underbrace{A_2 B_2}_{C_4} \, 2^{4k} \\
& + \underbrace{(A_2 B_1 + A_1 B_2)}_{C_3} \, 2^{3k} \\
& + \underbrace{(A_2 B_0 + A_1 B_1 + A_0 B_2)}_{C_2} \, 2^{2k} \\
& + \underbrace{(A_1 B_0 + A_0 B_1)}_{C_1} \, 2^{k} \\
& + \underbrace{A_0 B_0}_{C_0},
\end{aligned}
$$

Generalising Karatsuba's algorithm

Question

Can we get these five coefficients with less than 9 multiplications? Best case: 5 multiplications?

- Should we perhaps look at

$$\begin{aligned}(A_2 + A_1 + A_0)(B_2 + B_1 + B_0) = {} & A_0B_0 + A_1B_0 + A_2B_0 \\ & + A_0B_1 + A_1B_1 + A_2B_1 \\ & + A_0B_2 + A_1B_2 + A_2B_2?\end{aligned}$$

- Not clear at all how to get all five coefficients with 5 multiplications only ...

The Karatsuba trick: slicing into 3 pieces

- We now look for a method for getting these coefficients without any guesswork!

$$
\begin{aligned}
C_4 &= A_2 B_2 \\
C_3 &= A_2 B_1 + A_1 B_2 \\
C_2 &= A_2 B_0 + A_1 B_1 + A_0 B_2 \\
C_1 &= A_1 B_0 + A_0 B_1 \\
C_0 &= A_0 B_0
\end{aligned}
$$

- This should remind us of multiplying polynomials.

The Karatsuba trick: slicing into 3 pieces

- Let

$$A = A_2\, 2^{2k} + A_1\, 2^k + A_0$$
$$B = B_2\, 2^{2k} + B_1\, 2^k + B_0.$$

- We form the naturally corresponding polynomials:

$$P_A(x) = A_2\, x^2 + A_1\, x + A_0$$
$$P_B(x) = B_2\, x^2 + B_1\, x + B_0,$$

  where $A = P_A(2^k)$ and $B = P_B(2^k)$.
- Then the coefficients of $P_C(x) = P_A(x)P_B(x)$ are exactly the coefficients we want!

The Karatsuba trick: slicing into 3 pieces

- Let

$$\begin{aligned} P_C(x) &= P_A(x)P_B(x) \\ &= C_4\,x^4 + C_3\,x^3 + C_2\,x^2 + C_1\,x + C_0, \end{aligned}$$

  so that

$$\begin{aligned} A \cdot B &= P_A(2^k)P_B(2^k) \\ &= P_C(2^k) \\ &= C_4\,2^{4k} + C_3\,2^{3k} + C_2\,2^{2k} + C_1\,2^k + C_0. \end{aligned}$$

- Note that the right hand side involves only shifts and additions.

The Karatsuba trick: slicing into 3 pieces

Question

How do we find $P_C(x)$ without multiplying out in full?

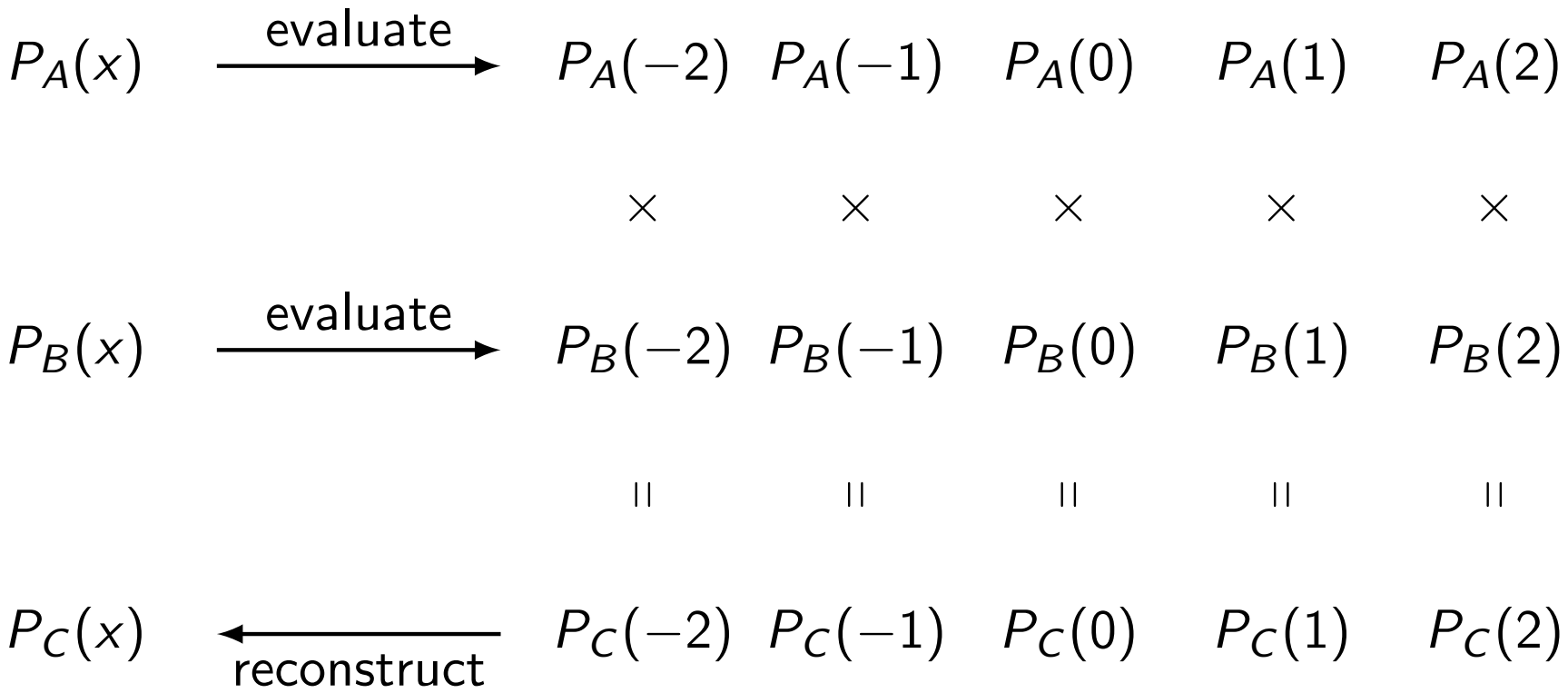
Answer

Find the value of $P_C(x)$ at several points, using

$$P_C(x_0) = P_A(x_0)P_B(x_0).$$

- Since the product polynomial $P_C(x) = P_A(x)P_B(x)$ is of degree 4 we need 5 values to **uniquely determine** $P_C(x)$. For simplicity, we choose -2, -1, 0, 1 and 2.

The Karatsuba trick: slicing into 3 pieces



The Karatsuba trick: slicing into 3 pieces

- First, let's examine the "evaluate" step. For $P_A(x) = A_2x^2 + A_1x + A_0$ we have

$$\begin{aligned}
P_A(-2) &= A_2(-2)^2 + A_1(-2) + A_0 &&= 4A_2 - 2A_1 + A_0 \\
P_A(-1) &= A_2(-1)^2 + A_1(-1) + A_0 &&= A_2 - A_1 + A_0 \\
P_A(0) &= A_20^2 + A_10 + A_0 &&= A_0 \\
P_A(1) &= A_21^2 + A_11 + A_0 &&= A_2 + A_1 + A_0 \\
P_A(2) &= A_22^2 + A_12 + A_0 &&= 4A_2 + 2A_1 + A_0,
\end{aligned}$$

  and we have similar expressions from $P_B(x)$.

- Note that these evaluations involve only a constant number of additions.

The Karatsuba trick: slicing into 3 pieces

- The next step is to multiply values of P_A and P_B to calculate values of P_C.

$$
\begin{aligned}
P_C(-2) &= P_A(-2)P_B(-2) \\
P_C(-1) &= P_A(-1)P_B(-1) \\
P_C(0) &= P_A(0)P_B(0) \\
P_C(1) &= P_A(1)P_B(1) \\
P_C(2) &= P_A(2)P_B(2)
\end{aligned}
$$

- At this stage we require only five multiplications of large (i.e. $n/3$-bit) numbers.

The Karatsuba trick: slicing into 3 pieces

- Perhaps the least straightforward step is "reconstruct". How do we go from values of P_C to its coefficients? Expanding out, we have

$$
\begin{aligned}
P_C(-2) &= C_4(-2)^4 + C_3(-2)^3 + C_2(-2)^2 + C_1(-2) + C_0 \\
P_C(-1) &= C_4(-1)^4 + C_3(-1)^3 + C_2(-1)^2 + C_1(-1) + C_0 \\
P_C(0) &= C_4 0^4 + C_3 0^3 + C_2 0^2 + C_1 \cdot 0 + C_0 \\
P_C(1) &= C_4 1^4 + C_3 1^3 + C_2 1^2 + C_1 \cdot 1 + C_0) \\
P_C(2) &= C_4 2^4 + C_3 2^3 + C_2 2^2 + C_1 \cdot 2 + C_0.
\end{aligned}
$$

- This is a system of five linear equations in five variables. We know how to solve these!

The Karatsuba trick: slicing into 3 pieces

- Using Gaussian elimination, we obtain

$$
\begin{aligned}
C_0 &= P_C(0) \\
C_1 &= \frac{1}{12}\left[P_C(-2) - 8P_C(-1) + 8P_C(1) - P_C(2)\right] \\
C_2 &= \frac{1}{24}\left[-P_C(-2) + 16P_C(-1) - 30P_C(0) + 16P_C(1) - P_C(2)\right] \\
C_3 &= \frac{1}{12}\left[-P_C(-2) + 2P_C(-1) - 2P_C(1) + P_C(2)\right] \\
C_4 &= \frac{1}{24}\left[P_C(-2) - 4P_C(-1) + 6P_C(0) - 4P_C(1) + P_C(2)\right].
\end{aligned}
$$

- Note that these expressions do not involve any multiplications of TWO large numbers and thus can be done in linear time (i.e. $O(n)$, where n is the number of bits).

The Karatsuba trick: slicing into 3 pieces

- With the coefficients C_4, C_3, C_2, C_1, C_0 obtained, we can now form the polynomial

$$P_C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0.$$

- We can then compute

$$P_C(2^k) = C_4 2^{4k} + C_3 2^{3k} + C_2 2^{2k} + C_1 2^k + C_0$$

  in linear time, using only bitwise shifts of the coefficients and a constant number of additions.
- Thus we have obtained $A \cdot B = P_A(2^k)P_B(2^k) = P_C(2^k)$ with only 5 multiplications!

The Karatsuba trick: slicing into 3 pieces

1: **function** $\textsc{Mult}(A, B)$

2: obtain A_2, A_1, A_0 and B_2, B_1, B_0 such that

$$A = A_2\ 2^{2\,k} + A_1\ 2^k + A_0; \qquad B = B_2\ 2^{2\,k} + B_1\ 2^k + B_0$$

3: form polynomials

$$P_A(x) = A_2 x^2 + A_1 x + A_0; \qquad P_B(x) = B_2 x^2 + B_1 x + B_0$$

The Karatsuba trick: slicing into 3 pieces

4: evaluate

$$
\begin{aligned}
P_A(-2) &\leftarrow 4A_2 - 2A_1 + A_0 & P_B(-2) &\leftarrow 4B_2 - 2B_1 + B_0;\\
P_A(-1) &\leftarrow A_2 - A_1 + A_0 & P_B(-1) &\leftarrow B_2 - B_1 + B_0;\\
P_A(0) &\leftarrow A_0 & P_B(0) &\leftarrow B_0;\\
P_A(1) &\leftarrow A_2 + A_1 + A_0 & P_B(1) &\leftarrow B_2 + B_1 + B_0;\\
P_A(2) &\leftarrow 4A_2 + 2A_1 + A_0 & P_B(2) &\leftarrow 4B_2 + 2B_1 + B_0.
\end{aligned}
$$

The Karatsuba trick: slicing into 3 pieces

5: multiply

$$
\begin{aligned}
P_C(-2) &\leftarrow \textsc{Mult}(P_A(-2), P_B(-2));\\
P_C(-1) &\leftarrow \textsc{Mult}(P_A(-1), P_B(-1));\\
P_C(0) &\leftarrow \textsc{Mult}(P_A(0), P_B(0));\\
P_C(1) &\leftarrow \textsc{Mult}(P_A(1), P_B(1));\\
P_C(2) &\leftarrow \textsc{Mult}(P_A(2), P_B(2)).
\end{aligned}
$$

The Karatsuba trick: slicing into 3 pieces

6: reconstruct

$$C_0 \leftarrow P_C(0);$$

$$C_1 \leftarrow \frac{1}{12}\left[P_C(-2) - 8P_C(-1) + 8P_C(1) - P_C(2)\right];$$

$$C_2 \leftarrow \frac{1}{24}\left[-P_C(-2) + 16P_C(-1) - 30P_C(0) + 16P_C(1) - P_C(2)\right];$$

$$C_3 \leftarrow \frac{1}{12}\left[-P_C(-2) + 2P_C(-1) - 2P_C(1) + P_C(2)\right];$$

$$C_4 \leftarrow \frac{1}{24}\left[P_C(-2) - 4P_C(-1) + 6P_C(0) - 4P_C(1) + P_C(2)\right].$$

The Karatsuba trick: slicing into 3 pieces

7: form

$$P_C(x) = C_4\,x^4 + C_3\,x^3 + C_2\,x^2 + C_1\,x + C_0$$

8: compute

$$P_C(2^k) = C_4\,2^{4k} + C_3\,2^{3k} + C_2\,2^{2k} + C_1\,2^k + C_0$$

9: **return** $P_C(2^k) = A \cdot B$.

10: **end function**

The Karatsuba trick: slicing into 3 pieces

- How fast is this algorithm?
- We have replaced a multiplication of two n bit numbers with 5 multiplications of $n/3$ bit numbers with an overhead of additions, shifts and so on, all doable in linear time $c\,n$, and thus

$$T(n) = 5\,T\left(\frac{n}{3}\right) + c\,n.$$

- We now apply the Master Theorem.
- The critical exponent is $c^* = \log_3 5 \approx 1.465$, so the critical polynomial is $n^{\log_3 5}$.
- Clearly $f(n) = O\left(n^{\log_3 5 - 0.1}\right)$, so the first case of the Master Theorem applies and we get $T(n) = \Theta\left(n^{\log_3 5}\right) = O(n^{1.47})$.

The Karatsuba trick: slicing into 3 pieces

- Recall that the original Karatsuba algorithm runs in time

$$n^{\log_2 3} \approx n^{1.58} > n^{1.47}.$$

- Thus, we got a significantly faster algorithm.

Question

Why not slice numbers A and B into even larger number of slices? Maybe we can get an even faster algorithm?

Answer

In a sense, BOTH yes and no. Let's see what happens if we slice numbers into $p+1$ many (approximately) equal slices, where $p = 1, 2, 3, \ldots...$

Generalising Karatsuba's algorithm

The general case - slicing the input numbers A, B into $p+1$ many slices

- For simplicity, let us assume A and B have exactly $n = (p+1)k$ bits (otherwise pad out the most significant slice with zeros).
- Note: p is a fixed parameter of our design – $p+1$ is the number of slices we are going to make, but k depends on the input values A and B and can be arbitrarily large!
- The runtime of our algorithm will be a function of both p and k, but the asymptotics will be entirely in terms of k, i.e. in terms of n.

Generalising Karatsuba's algorithm

- We begin by slicing A and B into $p+1$ pieces each:

$$A = \underbrace{XX\ldots X}_{k \text{ bits of } A_p}\ \underbrace{XX\ldots X}_{k \text{ bits of } A_{p-1}}\ \ldots \underbrace{XX\ldots X}_{k \text{ bits of } A_1}\underbrace{XX\ldots X}_{k \text{ bits of } A_0}$$

  i.e.,

$$A = A_p\, 2^{pk} + A_{p-1}\, 2^{(p-1)k} + \ldots + A_1\, 2^k + A_0$$
$$B = B_p\, 2^{pk} + B_{p-1}\, 2^{(p-1)k} + \ldots + B_1\, 2^k + B_0.$$

- Then

$$AB = \sum_{t=0}^{2p} \left(\sum_{i+j=t} A_i B_j \right) 2^{tk} = \sum_{t=0}^{2p} C_t 2^{tk}.$$

Generalising Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$P_A(x) = A_p\, x^p + A_{p-1}\, x^{p-1} + \ldots + A_1\, x + A_0$$
$$P_B(x) = B_p\, x^p + B_{p-1}\, x^{p-1} + \ldots + B_1\, x + B_0,$$

  and let

$$P_C(x) = P_A(x) \cdot P_B(x) = \sum_{t=0}^{2p} \left(\sum_{i+j=t} A_i B_j \right) x^t = \sum_{t=0}^{2p} C_t x^t.$$

- As before, we have

$$A = P_A(2^k) \text{ and } B = P_B(2^k),$$

  so

$$AB = P_A(2^k) \cdot P_B(2^k) = P_C(2^k).$$

Generalising Karatsuba's algorithm

- We will again adopt the following strategy: first multiply $P_A(x)$ and $P_B(x)$ to obtain $P_C(x)$, then evaluate $P_C(2^k)$.
- We saw that the coefficients of $P_C(x)$ are given by

$$C_t = \sum_{i+j=t} A_i B_j,$$

  requiring $(p+1)^2$ multiplications in total.
- However, $P_C(x)$ is of degree $2p$, so we can instead evaluate

$$P_C(x_0) = P_A(x_0) \cdot P_B(x_0)$$

  at $2p+1$ points (say $-p, \ldots, p$) and reconstruct the polynomial from these values.

An important digression: convolution

- This is in fact a very common problem, fundamental to many fields of engineering!
- Define vectors $\vec{v} = (v_0, v_1, \ldots, v_r)$ and $\vec{w} = (w_0, w_1, \ldots, w_s)$. Then let $\vec{u} = (u_0, u_1, \ldots, u_{r+s})$ such that

$$u_t = \sum_{i+j=t} v_i w_j.$$

  $\vec{u}$ is said to be the *linear convolution* of $\vec{v}$ and $\vec{w}$, denoted $\vec{u} = \vec{v} \star \vec{w}$.
- If we form polynomials with coefficients from $\vec{v}$ and $\vec{w}$, their product has coefficients given by $\vec{u}$.

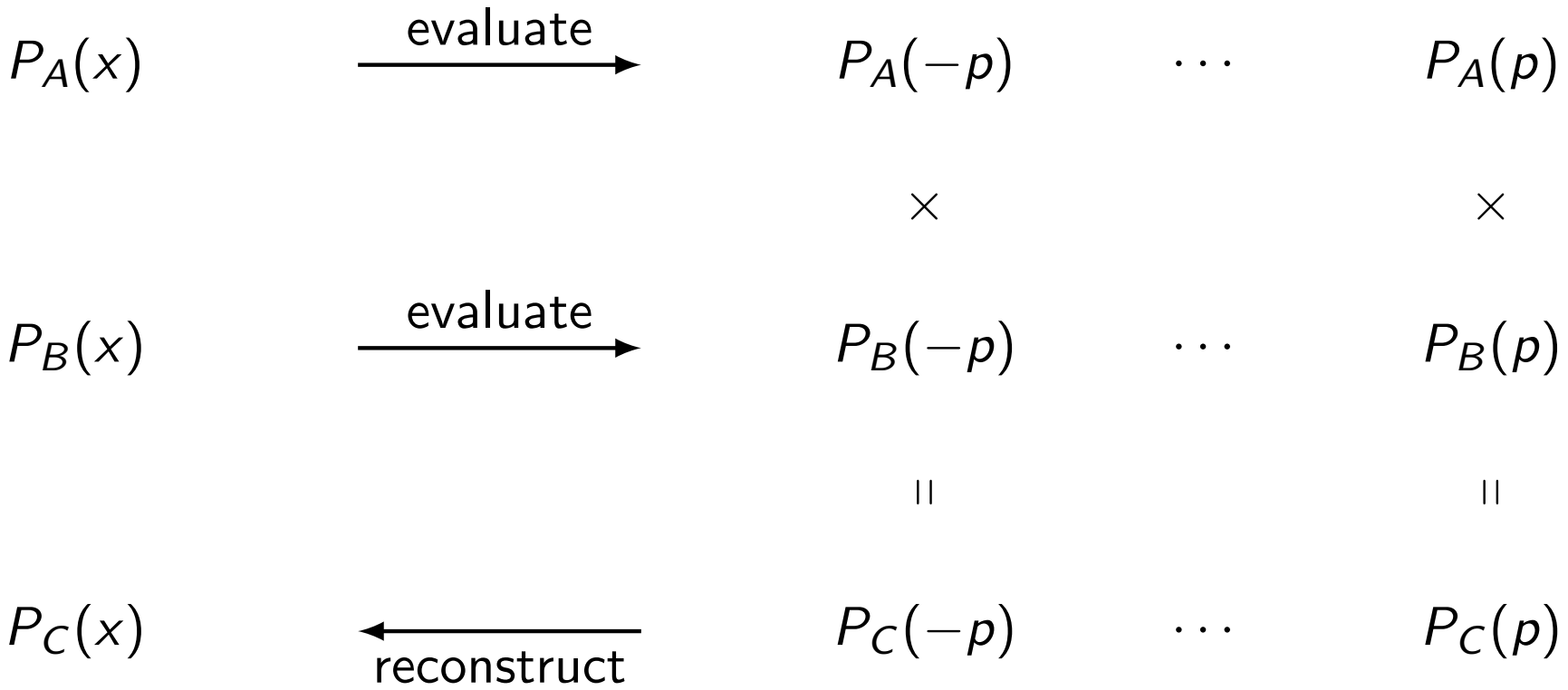
An important digression: convolution

- For example, if you have an audio signal and you want to emphasise the bass sounds, you would pass the sequence of discrete samples of the signal through a digital filter which amplifies the low frequencies more than the medium and the high audio frequencies.
- This is accomplished by computing the linear convolution of the sequence of discrete samples of the signal with a sequence of values which correspond to that filter, called *the impulse response* of the filter.
- This means that the samples of the output sound are simply the coefficients of the product of two polynomials:
  1. polynomial $P_A(x)$ whose coefficients A_i are the samples of the input signal;
  2. polynomial $P_B(x)$ whose coefficients B_k are the samples of the so called impulse response of the filter (they depend on what kind of filtering you want to do).

An important digression: convolution

- Convolutions are the bread-and-butter of signal processing, and for that reason it is **extremely important** to find fast ways of multiplying two polynomials of possibly very large degrees.
- In signal processing these degrees can be greater than 1000.
- This is the main reason for us to study methods of fast computation of convolutions (aside of finding products of large integers, which is what we are doing at the moment).

Generalising Karatsuba's algorithm



Generalising Karatsuba's algorithm

- To evaluate $P_A(x_0)$, we compute

$$P_A(x_0) = A_0 + A_1 x_0 + A_2 x_0^2 + \ldots + A_p x_0^p.$$

- This can be rewritten as a matrix-vector product as follows:

$$P_A(x_0) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix}.$$

Generalising Karatsuba's algorithm

- Then we can let

$$M_p = \begin{pmatrix} 1 & -p & (-p)^2 & \cdots & (-p)^p \\ 1 & -(p-1) & (-(p-1))^2 & \cdots & (-(p-1))^p \\ 1 & -(p-2) & (-(p-2))^2 & \cdots & (-(p-2))^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & p & p^2 & \cdots & p^p \end{pmatrix},$$

  which is a *Vandermonde matrix*.

- Since p is a constant, we will assume that this matrix is *precomputed*, so it will contribute only a *constant* to the runtime of our algorithm.

Generalising Karatsuba's algorithm

- Then we have

$$\begin{pmatrix} P_A(-p) \\ P_A(-(p-1)) \\ P_A(-(p-2)) \\ \vdots \\ P_A(p) \end{pmatrix} = M_p \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix}.$$

- This matrix-vector product requires $O(p^2)$ multiplications of a k-bit number A_i by an entry of M_p. However, M_p is a constant matrix of integers, so each of these multiplications takes $\Theta(k)$ time. The same is true for $P_B(x)$.
- Therefore, the 'evaluate' step takes *linear time*.

Generalising Karatsuba's algorithm

- The next step is to multiply. We compute

$$\begin{aligned} P_C(-p) &= P_A(-p)\,P_B(-p) \\ P_C(-(p-1)) &= P_A(-(p-1))\,P_B(-(p-1)) \\ P_C(-(p-2)) &= P_A(-(p-2))\,P_B(-(p-2)) \\ \cdots &= \cdots \\ P_C(p) &= P_A(p)\,P_B(p). \end{aligned}$$

- This requires $2p+1$ multiplications of large numbers.

Generalising Karatsuba's algorithm

- The 'reconstruct' step requires us to go from $2p+1$ values of $P_C(x)$ to the coefficients of the polynomial.
- Again we recall that

$$P_C(x_0) = C_0 + C_1 x_0 + C_2 x_0^2 + \ldots + C_{2p} x_0^{2p}$$

$$= \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{2p} \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \vdots \\ C_{2p} \end{pmatrix}.$$

Generalising Karatsuba's algorithm

- Let

$$M_{2p} = \begin{pmatrix} 1 & -p & (-p)^2 & \cdots & (-p)^{2p} \\ 1 & -(p-1) & (-(p-1))^2 & \cdots & (-(p-1))^{2p} \\ 1 & -(p-2) & (-(p-2))^2 & \cdots & (-(p-2))^{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & p & p^2 & \cdots & p^{2p} \end{pmatrix}.$$

Theorem

A square Vandermonde matrix with distinct rows is invertible.

- We will assume that M_{2p} and its inverse are precomputed.

Generalising Karatsuba's algorithm

- We have

$$\begin{pmatrix} P_C(-p) \\ P_C(-(p-1)) \\ P_C(-(p-2)) \\ \vdots \\ P_C(p) \end{pmatrix} = M_{2p} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \vdots \\ C_{2p} \end{pmatrix}. \qquad (1)$$

Generalising Karatsuba's algorithm

- Then, to find the coefficients C_t, we left multiply both sides of (1) by M_{2p}^{-1} to obtain

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \vdots \\ C_{2p} \end{pmatrix} = M_{2p}^{-1} \begin{pmatrix} P_C(-p) \\ P_C(-(p-1)) \\ P_C(-(p-2)) \\ \vdots \\ P_C(p) \end{pmatrix}.$$

- Much like the 'evaluate' step, this requires $O(p^2)$ multiplications of a constant by a large number, which is *linear time*.

Generalising Karatsuba's algorithm

1: **function** $\textsc{Mult}(A, B)$
2: **if** $|A| = |B| < p + 1$ **then return** AB
3: **else**
4: obtain $A_0, A_1, \ldots, A_p$ and $B_0, B_1, \ldots, B_p$ such that

$$A = A_p 2^{p\,k} + A_{p-1} 2^{(p-1)\,k} + \ldots + A_0$$

$$B = B_p 2^{p\,k} + B_{p-1} 2^{(p-1)\,k} + \ldots + B_0$$

5: form polynomials

$$P_A(x) = A_p x^p + A_{p-1} x^{(p-1)} + \ldots + A_0$$

$$P_B(x) = B_p x^p + B_{p-1} x^{(p-1)} + \ldots + B_0$$

Generalising Karatsuba's algorithm

6: **for** $m = -p$ to $m = p$ **do**
7: compute $P_A(m)$ and $P_B(m)$;
8: $P_C(m) \leftarrow \textsc{Mult}(P_A(m), P_B(m))$
9: **end for**

Generalising Karatsuba's algorithm

10: compute $C_0,\ C_1, \ldots\ C_{2p}$ via

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \vdots \\ C_{2p-1} \\ C_{2p} \end{pmatrix} = M_{2p}^{-1} \begin{pmatrix} P_C(-p) \\ P_C(-(p-1)) \\ P_C(-(p-2)) \\ \vdots \\ P_C(p-1) \\ P_C(p) \end{pmatrix}.$$

Generalising Karatsuba's algorithm

form $P_C(x) = C_{2p}x^{2p} + \ldots + C_0$ and compute $P_C(2^k)$
return $P_C(2^k) = A \cdot B$
end if
end function

How fast is our algorithm?

- As we discussed, the 'evaluate' and 'reconstruct' steps take linear time. However, we must be a little more careful with the multiplication step.
- How large are the function values

$$P_A(x) = A_p x^p + A_{p-1} x^{p-1} + \cdots + A_0?$$

  - Each A_i is a k-bit number.
  - Each term x^i has absolute value at most p^p.
  - There are $p + 1$ such terms.
- Therefore

$$|P_A(x)| < (p+1)p^p 2^k.$$

How fast is our algorithm?

- Since $|P_A(x)| < (p+1)p^p 2^k$, it follows that

$$\log_2|P_A(x)| < \log_2((p+1)p^p) + k.$$

- Letting

$$s = \lfloor \log_2((p+1)p^p) \rfloor,$$

we see that the function values are $(k+s)$-bit numbers, where s is a *constant*.

How fast is our algorithm?

- Thus, we have reduced a multiplication of two n-digit numbers to $2p+1$ multiplications of $(k+s)$-digit numbers plus a linear overhead (of additions, splitting the numbers, etc.)
- So we get the following recurrence for the complexity of $\textsc{Mult}(A, B)$:

$$T(n) = (2p+1)\,T\left(\frac{n}{p+1} + s\right) + \Theta(n).$$

- Since s is constant, its impact can be neglected.

How fast is our algorithm?

- So we have $a = 2p + 1$, $b = p + 1$ and $f(n) = \Theta(n)$. Let's use the Master Theorem.
- The critical exponent is

$$c^* = \log_b a = \log_{p+1}(2p + 1) > 1.$$

- Then there exists a small $\varepsilon > 0$ such that $f(n) = O\left(n^{c^* - \varepsilon}\right)$, so Case 1 of the Master Theorem applies.
- We conclude that

$$T(n) = \Theta\left(n^{c^*}\right) = \Theta\left(n^{\log_{p+1}(2p+1)}\right).$$

How fast is our algorithm?

- How small can $\log_{p+1}(2p+1)$ get? Note that

$$\begin{aligned}\log_{p+1}(2p+1) &< \log_{p+1} 2(p+1) \\ &= \log_{p+1} 2 + \log_{p+1}(p+1) \\ &= 1 + \log_{p+1} 2 \\ &= 1 + \frac{1}{\log_2(p+1)},\end{aligned}$$

  which can be made arbitrarily close to 1 by choosing a sufficiently large p.

- Therefore using a large enough number of slices allows us to get a run time arbitrarily close to linear time!

How fast is our algorithm?

Question

How many slices are needed for an $O(n^{1.1})$ time algorithm?

Answer

We want $c^* \leq 1.1$, so we can solve

$$\begin{aligned} 1 + \frac{1}{\log_2(p+1)} &= 1.1 \\ \log_2(p+1) &= 10 \\ p + 1 &= 2^{10} \end{aligned}$$

to see that 1024 slices should be used.

How fast is our algorithm, really?

- Unfortunately, there are some problems when we use large p.
- In our algorithm, we evaluate polynomials $P_A(x)$ and $P_B(x)$ both of degree p at values $x = -p, \ldots, p$.
- For $p = 1023$, evaluating $P_A(p) = A_p p^p + \ldots + A_0$ involves multiplication of A_p with $p^p = 1023^{1023} > 10^{3000}$.
- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -p, \ldots, p$ can all be done in **asymptotically** linear time, the constant factor on this is absolutely **humongous**.

How fast is our algorithm, really?

- Consequently, slicing the input numbers into more than just a few slices results in a hopelessly slow algorithm, despite the fact that the asymptotic bounds improve as we increase the number of slices!

- It is for this reason that, for example, Python implements multiplication of large numbers using only two slices, i.e. the original Karatsuba algorithm.

- The moral is: **In practice, asymptotic estimates are useless unless the size of the constants hidden by the big O notation are estimated and found to be reasonably small!!!**

Preview

- **Crucial question:** Are there numbers $x_0, x_1, \ldots, x_p$ such that the size of x_i^p does not grow uncontrollably?
- Answer: YES; they are the complex numbers z_i lying on the unit circle, i.e., such that $|z_i| = 1$!
- This motivates us to consider values of polynomials at inputs which are equally spaced complex numbers all lying on the unit circle.
- The sequence of such values is called **the discrete Fourier transform (DFT)** of the sequence of the coefficients of the polynomial being evaluated.

Preview

- We will present a very fast algorithm for computing these values, called **the Fast Fourier Transform**, abbreviated as **FFT**.
- The Fast Fourier Transform is **the most executed algorithm today** and is thus arguably **the most important algorithm of all**.
- Every mobile phone performs thousands of FFT runs each second, for example to compress your speech signal or to compress images taken by your camera, to mention just a few uses of the FFT.

Table of Contents

PUZZLE!

The warden meets with 23 new prisoners when they arrive. He tells them, "You may meet today and plan a strategy. But after today, you will be in isolated cells and will have no communication with one another. In the prison there is a switch room, which contains two light switches labeled A and B, each of which can be in either the on or the off position. I am not telling you their present positions. The switches are not connected to anything.

"After today, from time to time whenever I feel so inclined, I will select one prisoner at random and escort him to the switch room. This prisoner will select one of the two switches and reverse its position. He must move one, but only one of the switches. He can't move both but he can't move none either. Then he will be led back to his cell.

PUZZLE!

"No one else will enter the switch room until I lead the next prisoner there, and he'll be instructed to do the same thing. I'm going to choose prisoners at random. I may choose the same guy three times in a row, or I may jump around and come back. "But, given enough time, everyone would eventually visit the switch room many times. At any time anyone of you may declare to me:

We have all visited the switch room.

"If it is true, then you will all be set free. If it is false, and somebody has not yet visited the switch room, you will be fed to the alligators."

What is the strategy the prisoners can devise to gain their freedom?

That's All, Folks!!