Algorithms: COMP3121/3821/9101/9801

Aleks Ignjatović

School of Computer Science and Engineering
University of New South Wales

LECTURE 3: FAST LARGE INTEGER MULTIPLICATION

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
```

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:

The Karatsuba trick

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

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad A = \overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_1}\overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_0}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

The Karatsuba trick

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

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad A = \overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_1}\overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_0}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- $A_1 = \text{MoreSignificantPart}(A); \quad A_0 = \text{LessSignificantPart}(A);$

The Karatsuba trick

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

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad A = \overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_1}\overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_0}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- $A_1 = \text{MoreSignificantPart}(A); \quad A_0 = \text{LessSignificantPart}(A);$

- AB can now be calculated as follows:

The Karatsuba trick

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

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad A = \overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_1}\overbrace{\underbrace{XX\ldots X}_{n/2\ bits}}^{A_0}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- $A_1 = \text{MoreSignificantPart}(A); \quad A_0 = \text{LessSignificantPart}(A);$

- AB can now be calculated as follows:

$$AB = A_1 B_1 2^n + (A_1 B_0 + A_0 B_1) 2^{\frac{n}{2}} + A_0 B_0$$

The Karatsuba trick

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

$$A = A_1 2^{\frac{n}{2}} + A_0 \qquad A = \overbrace{\underbrace{XX\dots X}_{n/2\ bits}}^{A_1}\overbrace{\underbrace{XX\dots X}_{n/2\ bits}}^{A_0}$$

$$B = B_1 2^{\frac{n}{2}} + B_0$$

- $A_1 = \text{MoreSignificantPart}(A); \quad A_0 = \text{LessSignificantPart}(A);$
- AB can now be calculated as follows:

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

function $\textsc{Mult}(A, B)$
    if $|A| = |B| = 1$ **then return** AB
    else
        $A_1 \leftarrow \mathrm{MoreSignificantPart}(A);$
        $A_0 \leftarrow \mathrm{LessSignificantPart}(A);$
        $B_1 \leftarrow \mathrm{MoreSignificantPart}(B);$
        $B_0 \leftarrow \mathrm{LessSignificantPart}(B);$
        $U \leftarrow A_0 + A_1;$
        $V \leftarrow B_0 + B_1;$
        $X \leftarrow \textsc{Mult}(\mathrm{A}_0, \mathrm{B}_0);$
        $W \leftarrow \textsc{Mult}(\mathrm{A}_1, \mathrm{B}_1);$
        $Y \leftarrow \textsc{Mult}(\mathrm{U}, \mathrm{V});$
        return $W\,2^n + (Y - X - W)\,2^{n/2} + X$
    end if
end function

The Karatsuba trick

- How many multiplications does this take? (addition is in linear time!)

The Karatsuba trick

- How many multiplications does this take? (addition is in linear time!)
- Recurrence: $$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

The Karatsuba trick

- How many multiplications does this take? (addition is in linear time!)
- Recurrence: $$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

$$a = 3; \quad b = 2; \quad f(n) = c\,n; \quad n^{\log_b a} = n^{\log_2 3}$$

The Karatsuba trick

- How many multiplications does this take? (addition is in linear time!)
- Recurrence: $$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

$$a = 3; \quad b = 2; \quad f(n) = c\,n; \quad n^{\log_b a} = n^{\log_2 3}$$

- since $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < \log_2 3 - 1$$

The Karatsuba trick

- How many multiplications does this take? (addition is in linear time!)
- Recurrence: $$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

$$a = 3; \quad b = 2; \quad f(n) = c\,n; \quad n^{\log_b a} = n^{\log_2 3}$$

- since $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < \log_2 3 - 1$$

- Thus, the first case of the Master Theorem applies.

The Karatsuba trick

- How many multiplications does this take? (addition is in linear time!)
- Recurrence: $$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

$$a = 3; \quad b = 2; \quad f(n) = c\,n; \quad n^{\log_b a} = n^{\log_2 3}$$

- since $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < \log_2 3 - 1$$

- Thus, the first case of the Master Theorem applies.
- Consequently,

$$T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

without going through the messy calculations!

Generalizing Karatsuba's algorithm

Can we do better if we break the numbers in more than two pieces?

Generalizing Karatsuba's algorithm

Can we do better if we break the numbers in more than two pieces?

Lets try breaking the numbers A, B into 3 pieces:

Generalizing Karatsuba's algorithm

Can we do better if we break the numbers in more than two pieces?

Lets try breaking the numbers A, B into 3 pieces:

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

Generalizing Karatsuba's algorithm

Can we do better if we break the numbers in more than two pieces?

Lets try breaking the numbers A, B into 3 pieces:

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

$$A = \quad A_2 2^{2k} \quad + \quad A_1 2^k \quad + \quad A_0$$

Generalizing Karatsuba's algorithm

Can we do better if we break the numbers in more than two pieces?

Lets try breaking the numbers A, B into 3 pieces:

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

$$A = \quad A_2 2^{2k} \quad + \quad A_1 2^k \quad + \quad A_0$$

$$B = \quad B_2 2^{2k} \quad + \quad B_1 2^k \quad + \quad B_0$$

Generalizing Karatsuba's algorithm

Can we do better if we break the numbers in more than two pieces?

Lets try breaking the numbers A, B into 3 pieces:

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

$$A = \quad A_2 2^{2k} \quad + \quad A_1 2^k \quad + \quad A_0$$

$$B = \quad B_2 2^{2k} \quad + \quad B_1 2^k \quad + \quad B_0$$

$$AB = A_2B_2\,2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + \\ + (A_1B_0 + A_0B_1)2^k + A_0B_0$$

The Karatsuba trick

$$AB = A_2B_2\,2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + \\ + (A_1B_0 + A_0B_1)2^{k} + A_0B_0$$

The Karatsuba trick

$$AB = A_2B_2\,2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + \\ + (A_1B_0 + A_0B_1)2^{k} + A_0B_0$$

- we need only 5 coefficients:

The Karatsuba trick

$$AB = A_2B_2\,2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + \\ + (A_1B_0 + A_0B_1)2^{k} + A_0B_0$$

- we need only 5 coefficients:

$$\begin{aligned} C_4 &= A_2B_2 \\ C_3 &= A_2B_1 + A_1B_2 \\ C_2 &= A_2B_0 + A_1B_1 + A_0B_2 \\ C_1 &= A_1B_0 + A_0B_1 \\ C_0 &= A_0B_0 \end{aligned}$$

The Karatsuba trick

$$AB = A_2B_2\,2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + \\ + (A_1B_0 + A_0B_1)2^{k} + A_0B_0$$

- we need only 5 coefficients:

$$\begin{aligned} C_4 &= A_2B_2 \\ C_3 &= A_2B_1 + A_1B_2 \\ C_2 &= A_2B_0 + A_1B_1 + A_0B_2 \\ C_1 &= A_1B_0 + A_0B_1 \\ C_0 &= A_0B_0 \end{aligned}$$

- Can we get these with 5 multiplications only?

The Karatsuba trick

$$AB = A_2B_2\,2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + \\ + (A_1B_0 + A_0B_1)2^{k} + A_0B_0$$

- we need only 5 coefficients:

$$\begin{aligned} C_4 &= A_2B_2 \\ C_3 &= A_2B_1 + A_1B_2 \\ C_2 &= A_2B_0 + A_1B_1 + A_0B_2 \\ C_1 &= A_1B_0 + A_0B_1 \\ C_0 &= A_0B_0 \end{aligned}$$

- Can we get these with 5 multiplications only?
- Should we perhaps look at

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

The Karatsuba trick

$$AB = A_2B_2\,2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + \\ + (A_1B_0 + A_0B_1)2^{k} + A_0B_0$$

- we need only 5 coefficients:

$$\begin{aligned} C_4 &= A_2B_2 \\ C_3 &= A_2B_1 + A_1B_2 \\ C_2 &= A_2B_0 + A_1B_1 + A_0B_2 \\ C_1 &= A_1B_0 + A_0B_1 \\ C_0 &= A_0B_0 \end{aligned}$$

- Can we get these with 5 multiplications only?
- Should we perhaps look at

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

- Not clear how to get $C_0 - C_4$ with 5 multiplications only ...

The Karatsuba trick: slicing into 3 pieces

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

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

The Karatsuba trick: slicing into 3 pieces

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

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

The Karatsuba trick: slicing into 3 pieces

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

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

- Their product $P_C(x) = P_A(x)P_B(x)$ is of degree 4;

The Karatsuba trick: slicing into 3 pieces

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

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

- Their product $P_C(x) = P_A(x)P_B(x)$ is of degree 4;
- we need 5 values to uniquely determine the product.

The Karatsuba trick: slicing into 3 pieces

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

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

- Their product $P_C(x) = P_A(x)P_B(x)$ is of degree 4;
- we need 5 values to uniquely determine the product.
- We choose **the smallest possible** 5 **integer values**, i.e., -2, -1, 0, 1, 2.

The Karatsuba trick: slicing into 3 pieces

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

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

- Their product $P_C(x) = P_A(x)P_B(x)$ is of degree 4;
- we need 5 values to uniquely determine the product.
- We choose **the smallest possible** 5 **integer values**, i.e., -2, -1, 0, 1, 2.
- Thus, we compute
  $$P_A(-2), P_A(-1), P_A(0), P_A(1), P_A(2)$$
  $$P_B(-2), P_B(-1), P_B(0), P_B(1), P_B(2)$$

The Karatsuba trick: slicing into 3 pieces

- For $P_A(x) = A_2x^2 + A_1x + A_0$ we have

$$
\begin{aligned}
P_A(-2) &= A_2(-2)^2 + A_1(-2) + A_0 = & 4A_2 - 2A_1 + A_0 \\
P_A(-1) &= A_2(-1)^2 + A_1(-1) + A_0 = & A_2 - A_1 + A_0 \\
P_A(0) &= A_2 0^2 + A_1 0 + A_0 = & A_0 \\
P_A(1) &= A_2 1^2 + A_1 1 + A_0 = & A_2 + A_1 + A_0 \\
P_A(2) &= A_2 2^2 + A_1 2 + A_0 = & 4A_2 + 2A_1 + A_0
\end{aligned}
$$

The Karatsuba trick: slicing into 3 pieces

- For $P_A(x) = A_2x^2 + A_1x + A_0$ we have

$$
\begin{aligned}
P_A(-2) &= A_2(-2)^2 + A_1(-2) + A_0 = & 4A_2 - 2A_1 + A_0 \\
P_A(-1) &= A_2(-1)^2 + A_1(-1) + A_0 = & A_2 - A_1 + A_0 \\
P_A(0) &= A_2 0^2 + A_1 0 + A_0 = & A_0 \\
P_A(1) &= A_2 1^2 + A_1 1 + A_0 = & A_2 + A_1 + A_0 \\
P_A(2) &= A_2 2^2 + A_1 2 + A_0 = & 4A_2 + 2A_1 + A_0
\end{aligned}
$$

- Similarly, for $P_B(x) = B_2x^2 + B_1x + B_0$ we have

$$
\begin{aligned}
P_B(-2) &= B_2(-2)^2 + B_1(-2) + B_0 = & 4B_2 - 2B_1 + B_0 \\
P_B(-1) &= B_2(-1)^2 + B_1(-1) + B_0 = & B_2 - B_1 + B_0 \\
P_B(0) &= B_2 0^2 + B_1 0 + B_0 = & B_0 \\
P_B(1) &= B_2 1^2 + B_1 1 + B_0 = & B_2 + B_1 + B_0 \\
P_B(2) &= B_2 2^2 + B_1 2 + B_0 = & 4B_2 + 2B_1 + B_0
\end{aligned}
$$

The Karatsuba trick: slicing into 3 pieces

- Having obtained $P_A(-2), P_A(-1), P_A(0), P_A(1), P_A(2)$ and $P_B(-2), P_B(-1), P_B(0), P_B(1), P_B(2)$ we can now compute

$$\begin{aligned} P_C(-2) &= P_A(-2)P_B(-2) \\ &= (A_0 - 2A_1 + 4A_2)(B_0 - 2B_1 + 4B_2) \end{aligned}$$

$$\begin{aligned} P_C(-1) &= P_A(-1)P_B(-1) \\ &= (A_0 - A_1 + A_2)(B_0 - B_1 + B_2) \end{aligned}$$

$$\begin{aligned} P_C(0) &= P_A(0)P_B(0) \\ &= A_0 B_0 \end{aligned}$$

$$\begin{aligned} P_C(1) &= P_A(1)P_B(1) \\ &= (A_0 + A_1 + A_2)(B_0 + B_1 + B_2) \end{aligned}$$

$$\begin{aligned} P_C(2) &= P_A(2)P_B(2) \\ &= (A_0 + 2A_1 + 4A_2)(B_0 + 2B_1 + 4B_2) \end{aligned}$$

Thus, it takes 5 **large integer multiplications** to obtain $P_C(-2), P_C(-1), P_C(0), P_C(1), P_C(2)$

The Karatsuba trick: slicing into 3 pieces

- A polynomial $P_C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0$ is uniquely determined by its values at 5 distinct inputs x_1, x_2, x_3, x_4, x_5 and we have in a matrix form

$$\begin{pmatrix} 1 & x_1 & (x_1)^2 & (x_1)^3 & (x_1)^4 \\ 1 & x_2 & (x_2)^2 & (x_2)^3 & (x_2)^4 \\ 1 & x_3 & (x_3)^2 & (x_3)^3 & (x_3)^4 \\ 1 & x_4 & (x_4)^2 & (x_4)^3 & (x_4)^4 \\ 1 & x_5 & (x_5)^2 & (x_5)^3 & (x_5)^4 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} C_0 + C_1x_1 + C_2(x_1)^2 + C_3(x_1)^3 + C_4(x_1)^4 \\ C_0 + C_1x_2 + C_2(x_2)^2 + C_3(x_2)^3 + C_4(x_2)^4 \\ C_0 + C_1x_3 + C_2(x_3)^2 + C_3(x_3)^3 + C_4(x_3)^4 \\ C_0 + C_1x_4 + C_2(x_4)^2 + C_3(x_4)^3 + C_4(x_4)^4 \\ C_0 + C_1x_5 + C_2(x_5)^2 + C_3(x_5)^3 + C_4(x_5)^4 \end{pmatrix}$$

$$= \begin{pmatrix} P_C(x_1) \\ P_C(x_2) \\ P_C(x_3) \\ P_C(x_4) \\ P_C(x_5) \end{pmatrix}$$

The Karatsuba trick: slicing into 3 pieces

- For the product $P_C(x) = P_A(x)P_B(x)$ we have

$$\begin{pmatrix} 1 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 1 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ 1 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1 & 1^1 & 1^2 & 1^3 & 1^4 \\ 1 & 2^1 & 2^2 & 2^3 & 2^4 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix}$$

$$= \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

The Karatsuba trick: slicing into 3 pieces

- Since

$$\begin{pmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

The Karatsuba trick: slicing into 3 pieces

- Since

$$\begin{pmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

we have

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{pmatrix}^{-1} \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

The Karatsuba trick: slicing into 3 pieces

- Since

$$\begin{pmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

we have

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{pmatrix}^{-1} \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

i.e.,

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ \frac{1}{12} & -\frac{2}{3} & 0 & \frac{2}{3} & -\frac{1}{12} \\ -\frac{1}{24} & \frac{2}{3} & -\frac{5}{4} & \frac{2}{3} & -\frac{1}{24} \\ -\frac{1}{12} & \frac{1}{6} & 0 & -\frac{1}{6} & \frac{1}{12} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \end{pmatrix} \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

The Karatsuba trick: slicing into 3 pieces

- From

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ \frac{1}{12} & -\frac{2}{3} & 0 & \frac{2}{3} & -\frac{1}{12} \\ -\frac{1}{24} & \frac{2}{3} & -\frac{5}{4} & \frac{2}{3} & -\frac{1}{24} \\ -\frac{1}{12} & \frac{1}{6} & 0 & -\frac{1}{6} & \frac{1}{12} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \end{pmatrix} \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

The Karatsuba trick: slicing into 3 pieces

- From

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ \frac{1}{12} & -\frac{2}{3} & 0 & \frac{2}{3} & -\frac{1}{12} \\ -\frac{1}{24} & \frac{2}{3} & -\frac{5}{4} & \frac{2}{3} & -\frac{1}{24} \\ -\frac{1}{12} & \frac{1}{6} & 0 & -\frac{1}{6} & \frac{1}{12} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \end{pmatrix} \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

- If we do the multiplications we obtain

$$\begin{aligned}
C_0 &= P_C(0) \\
C_1 &= \frac{P_C(-2)}{12} - \frac{2P_C(-1)}{3} + \frac{2P_C(1)}{3} - \frac{P_C(2)}{12} \\
C_2 &= -\frac{P_C(-2)}{24} + \frac{2P_C(-1)}{3} - \frac{5P_C(0)}{4} + \frac{2P_C(1)}{3} - \frac{P_C(2)}{24} \\
C_3 &= -\frac{P_C(-2)}{12} + \frac{P_C(-1)}{6} - \frac{P_C(1)}{6} + \frac{P_C(2)}{12} \\
C_4 &= \frac{P_C(-2)}{24} - \frac{P_C(-1)}{6} + \frac{P_C(0)}{4} - \frac{P_C(1)}{6} + \frac{P_C(2)}{24}
\end{aligned}$$

The Karatsuba trick: slicing into 3 pieces

- From

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ \frac{1}{12} & -\frac{2}{3} & 0 & \frac{2}{3} & -\frac{1}{12} \\ -\frac{1}{24} & \frac{2}{3} & -\frac{5}{4} & \frac{2}{3} & -\frac{1}{24} \\ -\frac{1}{12} & \frac{1}{6} & 0 & -\frac{1}{6} & \frac{1}{12} \\ \frac{1}{24} & -\frac{1}{6} & \frac{1}{4} & -\frac{1}{6} & \frac{1}{24} \end{pmatrix} \begin{pmatrix} P_C(-2) \\ P_C(-1) \\ P_C(0) \\ P_C(1) \\ P_C(2) \end{pmatrix}$$

- If we do the multiplications we obtain

$$\begin{aligned}
C_0 &= P_C(0) \\
C_1 &= \frac{P_C(-2)}{12} - \frac{2P_C(-1)}{3} + \frac{2P_C(1)}{3} - \frac{P_C(2)}{12} \\
C_2 &= -\frac{P_C(-2)}{24} + \frac{2P_C(-1)}{3} - \frac{5P_C(0)}{4} + \frac{2P_C(1)}{3} - \frac{P_C(2)}{24} \\
C_3 &= -\frac{P_C(-2)}{12} + \frac{P_C(-1)}{6} - \frac{P_C(1)}{6} + \frac{P_C(2)}{12} \\
C_4 &= \frac{P_C(-2)}{24} - \frac{P_C(-1)}{6} + \frac{P_C(0)}{4} - \frac{P_C(1)}{6} + \frac{P_C(2)}{24}
\end{aligned}$$

- Thus, from the five values $P_C(-2)$, $P_C(-1)$, $P_C(0)$, $P_C(1)$, $P_C(2)$ of $P_C(x) = P_A(x)P_B(x)$
we get the five coefficients C_0, C_1, C_2, C_3, C_4 of $P_C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0$ in **linear time**.

The Karatsuba trick: slicing into 3 pieces

- With the coefficients C_0, C_1, C_2, C_3, C_4 obtained, we can now form the polynomial
$$P_C(x) = C_0 + C_1 x + C_2 x^2 + C_3 x^3 + C_4 x^4$$

The Karatsuba trick: slicing into 3 pieces

- With the coefficients C_0, C_1, C_2, C_3, C_4 obtained, we can now form the polynomial
$$P_C(x) = C_0 + C_1 x + C_2 x^2 + C_3 x^3 + C_4 x^4$$

- We can now compute

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

in linear time, because computing $P_C(2^k)$ involves only binary shifts of the coefficients plus $O(k)$ additions.

The Karatsuba trick: slicing into 3 pieces

- With the coefficients C_0, C_1, C_2, C_3, C_4 obtained, we can now form the polynomial
$$P_C(x) = C_0 + C_1 x + C_2 x^2 + C_3 x^3 + C_4 x^4$$

- We can now compute
$$P_C(2^k) = C_0 + C_1 2^k + C_2 2^{2k} + C_3 2^{3k} + C_4 2^{4k}$$
in linear time, because computing $P_C(2^k)$ involves only binary shifts of the coefficients plus $O(k)$ additions.

- Here is the complete algorithm:

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

obtain A_0, A_1, A_2 and B_0, B_1, B_2 such that $A = A_2\, 2^{2\,k} + A_1\, 2^k + A_0$; $\quad B = B_2\, 2^{2\,k} + B_1\, 2^k + B_0$;

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

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

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

$$
\begin{aligned}
C_0 &\leftarrow P_C(0); \qquad C_1 \leftarrow \frac{P_C(-2)}{12} - \frac{2P_C(-1)}{3} + \frac{2P_C(1)}{3} - \frac{P_C(2)}{12}\\
C_2 &\leftarrow -\frac{P_C(-2)}{24} + \frac{2P_C(-1)}{3} - \frac{5P_C(0)}{4} + \frac{2P_C(1)}{3} - \frac{P_C(2)}{24}\\
C_3 &\leftarrow -\frac{P_C(-2)}{12} + \frac{P_C(-1)}{6} - \frac{P_C(1)}{6} + \frac{P_C(2)}{12}\\
C_4 &\leftarrow \frac{P_C(-2)}{24} - \frac{P_C(-1)}{6} + \frac{P_C(0)}{4} - \frac{P_C(1)}{6} + \frac{P_C(2)}{24}
\end{aligned}
$$

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

return $P_C(2^k)$

end function

The Karatsuba trick: slicing into 3 pieces

- How fast is this algorithm?

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 the similar, all doable in linear time $c\,n$;

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 the similar, all doable in linear time $c\,n$; thus,

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

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 the similar, all doable in linear time $c\,n$; thus,

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

- We now apply the Master Theorem:
we have $a = 5,\ \ b = 3$, so we consider $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465\ldots}$

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 the similar, all doable in linear time $c\,n$; thus,

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

- We now apply the Master Theorem:
we have $a = 5,\ \ b = 3$, so we consider $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465\ldots}$

- Clearly the first case applies and we get $T(n) = O(n^{1.47})$;

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 the similar, all doable in linear time $c\,n$; thus,

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

- We now apply the Master Theorem:
we have $a = 5,\;\; b = 3$, so we consider $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465\ldots}$

- Clearly the first case applies and we get $T(n) = O(n^{1.47})$;

- Recall that the original Karatsuba algorithm runs in time $n^{\log_2 3} \approx n^{1.58} > n^{1.47}$.

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 the similar, all doable in linear time $c\,n$; thus,

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

- We now apply the Master Theorem:
we have $a = 5, \;\; b = 3$, so we consider $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465\ldots}$

- Clearly the first case applies and we get $T(n) = O(n^{1.47})$;

- 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.

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 the similar, all doable in linear time $c\,n$; thus,

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

- We now apply the Master Theorem:
we have $a = 5, \;\; b = 3$, so we consider $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465\ldots}$

- Clearly the first case applies and we get $T(n) = O(n^{1.47})$;

- 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.

- Then why not slice numbers into even larger number of slices? Maybe we can get even faster algorithm?

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 the similar, all doable in linear time $c\,n$; thus,

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

- We now apply the Master Theorem:
we have $a = 5,\ \ b = 3$, so we consider $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465\ldots}$

- Clearly the first case applies and we get $T(n) = O(n^{1.47})$;

- 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.

- Then why not slice numbers into even larger number of slices? Maybe we can get even faster algorithm?

- The answer is, in a sense, BOTH yes and no, so lets see what happens if we slice numbers into $n + 1$ many equal slices...

Generalizing Karatsuba's algorithm

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

Generalizing Karatsuba's algorithm

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

- For simplicity let A, B have $(n+1)k$ bits; (k can be arbitrarily large)

Generalizing Karatsuba's algorithm

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

- For simplicity let A, B have $(n+1)k$ bits; (k can be arbitrarily large)
- Slice A, B into $n+1$ pieces each:

$$\begin{aligned} A &= A_n 2^{kn} + A_{n-1}2^{k(n-1)} + \cdots + A_0 \\ B &= B_n 2^{kn} + B_{n-1}2^{k(n-1)} + \cdots + B_0 \end{aligned}$$

Generalizing Karatsuba's algorithm

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

- For simplicity let A, B have $(n+1)k$ bits; (k can be arbitrarily large)
- Slice A, B into $n+1$ pieces each:

$$\begin{aligned} A = \quad & A_n 2^{kn} \quad + \quad A_{n-1} 2^{k(n-1)} + \quad \cdots + \quad A_0 \\ B = \quad & B_n 2^{kn} \quad + \quad B_{n-1} 2^{k(n-1)} + \quad \cdots + \quad B_0 \end{aligned}$$

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) = \quad & A_n x^n \quad + \quad A_{n-1} x^{n-1} + \quad \cdots + \quad A_0 \\ P_B(x) = \quad & B_n x^n \quad + \quad B_{n-1} x^{n-1} + \quad \cdots + \quad B_0 \end{aligned}$$

Generalizing Karatsuba's algorithm

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

- For simplicity let A, B have $(n+1)k$ bits; (k can be arbitrarily large)
- Slice A, B into $n+1$ pieces each:

$$\begin{aligned} A = \quad & A_n 2^{kn} & + \quad & A_{n-1} 2^{k(n-1)} + & \cdots + \quad & A_0 \\ B = \quad & B_n 2^{kn} & + \quad & B_{n-1} 2^{k(n-1)} + & \cdots + \quad & B_0 \end{aligned}$$

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) = \quad & A_n x^n & + \quad & A_{n-1} x^{n-1} + & \cdots + \quad & A_0 \\ P_B(x) = \quad & B_n x^n & + \quad & B_{n-1} x^{n-1} + & \cdots + \quad & B_0 \end{aligned}$$

  - coefficients satisfy $A_i, B_i < 2^k$

Generalizing Karatsuba's algorithm

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

- For simplicity let A, B have $(n+1)k$ bits; (k can be arbitrarily large)
- Slice A, B into $n+1$ pieces each:

$$\begin{aligned} A &= A_n 2^{kn} + A_{n-1} 2^{k(n-1)} + \cdots + A_0 \\ B &= B_n 2^{kn} + B_{n-1} 2^{k(n-1)} + \cdots + B_0 \end{aligned}$$

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_n x^n + A_{n-1} x^{n-1} + \cdots + A_0 \\ P_B(x) &= B_n x^n + B_{n-1} x^{n-1} + \cdots + B_0 \end{aligned}$$

  - coefficients satisfy $A_i, B_i < 2^k$
  - $A = P_A(2^k)$; $\quad B = P_B(2^k)$

Generalizing Karatsuba's algorithm

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

- For simplicity let A, B have $(n+1)k$ bits; (k can be arbitrarily large)
- Slice A, B into $n+1$ pieces each:

$$\begin{aligned} A &= A_n 2^{kn} + A_{n-1} 2^{k(n-1)} + \cdots + A_0 \\ B &= B_n 2^{kn} + B_{n-1} 2^{k(n-1)} + \cdots + B_0 \end{aligned}$$

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_n x^n + A_{n-1} x^{n-1} + \cdots + A_0 \\ P_B(x) &= B_n x^n + B_{n-1} x^{n-1} + \cdots + B_0 \end{aligned}$$

  - coefficients satisfy $A_i, B_i < 2^k$
  - $A = P_A(2^k); \quad B = P_B(2^k)$
  - $AB = P_A(2^n) P_B(2^n) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

Generalizing Karatsuba's algorithm

$$AB = P_A(2^n)P_B(2^n) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$

Generalizing Karatsuba's algorithm

$$AB = P_A(2^n)P_B(2^n) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$

Strategy:

Generalizing Karatsuba's algorithm

$$AB = P_A(2^n)P_B(2^n) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$

Strategy:

- figure out how to multiply polynomials fast to obtain

$$P_C(x) = P_A(x) \cdot P_B(x);$$

Generalizing Karatsuba's algorithm

$$AB = P_A(2^n)P_B(2^n) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$

Strategy:

- figure out how to multiply polynomials fast to obtain

$$P_C(x) = P_A(x) \cdot P_B(x);$$

- evaluate $P_C(2^k)$.

Generalizing Karatsuba's algorithm

$$AB = P_A(2^n)P_B(2^n) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$

Strategy:

- figure out how to multiply polynomials fast to obtain

$$P_C(x) = P_A(x) \cdot P_B(x);$$

- evaluate $P_C(2^k)$.
- Note that $P_C(x) = P_A(x) \cdot P_B(x)$ is of degree $2n$:

Generalizing Karatsuba's algorithm

$$AB = P_A(2^n)P_B(2^n) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$

Strategy:

- figure out how to multiply polynomials fast to obtain

$$P_C(x) = P_A(x) \cdot P_B(x);$$

- evaluate $P_C(2^k)$.
- Note that $P_C(x) = P_A(x) \cdot P_B(x)$ is of degree $2n$:

$$P_C(x) = \sum_{j=0}^{2n} C_j x^j$$

Generalizing Karatsuba's algorithm

Example:

$$
\begin{aligned}
&(a_3x^3 + a_2x^2 + a_1x + a_0)(b_3x^3 + b_2x^2 + b_1x + b_0) = \\
&\quad a_3b_3x^6 + (a_2b_3 + a_3b_2)x^5 + (a_1b_3 + a_2b_2 + a_3b_1)x^4 \\
&\qquad +(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)x^3 + (a_0b_2 + a_1b_1 + a_2b_0)x^2 \\
&\qquad\qquad +(a_0b_1 + a_1b_0)x + a_0b_0
\end{aligned}
$$

Generalizing Karatsuba's algorithm

Example:

$$
\begin{aligned}
(a_3x^3 + a_2x^2 + a_1x + a_0)(b_3x^3 + b_2x^2 + b_1x + b_0) = \\
a_3b_3x^6 + (a_2b_3 + a_3b_2)x^5 + (a_1b_3 + a_2b_2 + a_3b_1)x^4 \\
+(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)x^3 + (a_0b_2 + a_1b_1 + a_2b_0)x^2 \\
+(a_0b_1 + a_1b_0)x + a_0b_0
\end{aligned}
$$

In general: for

$$
\begin{aligned}
P_A(x) &= A_nx^n + A_{n-1}x^{n-1} + \cdots + A_0 \\
P_B(x) &= B_nx^n + B_{n-1}x^{n-1} + \cdots + B_0
\end{aligned}
$$

Generalizing Karatsuba's algorithm

Example:

$$
\begin{aligned}
(a_3x^3 + a_2x^2 + a_1x + a_0)(b_3x^3 + b_2x^2 + b_1x + b_0) = \\
a_3b_3x^6 + (a_2b_3 + a_3b_2)x^5 + (a_1b_3 + a_2b_2 + a_3b_1)x^4 \\
+(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)x^3 + (a_0b_2 + a_1b_1 + a_2b_0)x^2 \\
+(a_0b_1 + a_1b_0)x + a_0b_0
\end{aligned}
$$

In general: for

$$
\begin{aligned}
P_A(x) &= A_nx^n + A_{n-1}x^{n-1} + \cdots + A_0 \\
P_B(x) &= B_nx^n + B_{n-1}x^{n-1} + \cdots + B_0
\end{aligned}
$$

setting $A_i = 0$ and $B_i = 0$ for $n < i \leq 2n$ we have

Generalizing Karatsuba's algorithm

Example:

$$\begin{aligned}
&(a_3x^3 + a_2x^2 + a_1x + a_0)(b_3x^3 + b_2x^2 + b_1x + b_0) = \\
&\quad a_3b_3x^6 + (a_2b_3 + a_3b_2)x^5 + (a_1b_3 + a_2b_2 + a_3b_1)x^4 \\
&\quad\quad +(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)x^3 + (a_0b_2 + a_1b_1 + a_2b_0)x^2 \\
&\quad\quad\quad +(a_0b_1 + a_1b_0)x + a_0b_0
\end{aligned}$$

In general: for

$$P_A(x) = A_nx^n + A_{n-1}x^{n-1} + \cdots + A_0$$
$$P_B(x) = B_nx^n + B_{n-1}x^{n-1} + \cdots + B_0$$

setting $A_i = 0$ and $B_i = 0$ for $n < i \leq 2n$ we have

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \left(\sum_{i=0}^{j} A_iB_{j-i} \right) x^j = \sum_{j=0}^{2n} C_j\, x^j$$

We need to find the coefficients $C_j = \sum_{i=0}^{j} A_iB_{j-i}$ without performing $(n+1)^2$ many multiplications

Coefficient vs value representation of polynomials

- Every polynomial $P_A(x)$ of degree n is uniquely determined by its values at any $n+1$ distinct input values for x:

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_n, P_A(x_n))\}$$

Coefficient vs value representation of polynomials

- Every polynomial $P_A(x)$ of degree n is uniquely determined by its values at any $n+1$ distinct input values for x:

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_n, P_A(x_n))\}$$

- If $P_A(x) = A_n x^n + A_{n-1}x^{n-1} + \ldots + A_0$, we can write in matrix form:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix}. \tag{1}$$

Coefficient vs value representation of polynomials

- Every polynomial $P_A(x)$ of degree n is uniquely determined by its values at any $n+1$ distinct input values for x:

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_n, P_A(x_n))\}$$

- If $P_A(x) = A_n x^n + A_{n-1}x^{n-1} + \ldots + A_0$, we can write in matrix form:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix}. \tag{1}$$

- It can be shown that, if x_i are all distinct then this matrix is invertible.

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix} \tag{2}$$

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix} \tag{2}$$

- Equations (1) and (2) show how we can commute between:

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix} \tag{2}$$

- Equations (1) and (2) show how we can commute between:
  1. a representation of a polynomial $P_A(x)$ via its coefficients $A_n, A_{n-1}, \ldots, A_0$, i.e. $P_A(x) = A_n x^n + \ldots + A_1 x + A_0$

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix} \tag{2}$$

- Equations (1) and (2) show how we can commute between:
  1. a representation of a polynomial $P_A(x)$ via its coefficients $A_n, A_{n-1}, \ldots, A_0$, i.e. $P_A(x) = A_n x^n + \ldots + A_1 x + A_0$
  2. a representation of a polynomial $P_A(x)$ via its values

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_n, P_A(x_n))\}$$

Coefficient vs value representation of polynomials- ctd.

Commuting between a representation of a polynomial $P_A(x)$ via its coefficients $A_n, A_{n-1}, \ldots, A_0$, i.e. $P_A(x) = A_n x^n + \ldots + A_1 x + A_0$ and a representation of a polynomial $P_A(x)$ via its values

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_n, P_A(x_n))\}$$

is done via the following two matrix multiplications, with matrices made up from constants:

$$\begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix}.$$

Coefficient vs value representation of polynomials- ctd.

Commuting between a representation of a polynomial $P_A(x)$ via its coefficients $A_n, A_{n-1}, \ldots, A_0$, i.e. $P_A(x) = A_n x^n + \ldots + A_1 x + A_0$ and a representation of a polynomial $P_A(x)$ via its values

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_n, P_A(x_n))\}$$

is done via the following two matrix multiplications, with matrices made up from constants:

$$\begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix}.$$

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_n) \end{pmatrix}$$

Thus, for fixed input values $x_0, \ldots, x_n$ this switch between the two kinds of representations is done in **linear time**!

Our strategy to multiply polynomials fast:

- Given two polynomials of degree at most n,

$$P_A(x) = A_n x^n + \ldots + A_0; \qquad P_B(x) = B_n x^n + \ldots + B_0$$

Our strategy to multiply polynomials fast:

- Given two polynomials of degree at most n,

$$P_A(x) = A_n x^n + \ldots + A_0; \qquad P_B(x) = B_n x^n + \ldots + B_0$$

1. convert them into value representation at $2n+1$ distinct points $x_0, x_1, \ldots, x_{2n}$:

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_{2n}, P_A(x_{2n}))\}$$

$$P_B(x) \leftrightarrow \{(x_0, P_B(x_0)), (x_1, P_B(x_1)), \ldots, (x_{2n}, P_B(x_{2n}))\}$$

Our strategy to multiply polynomials fast:

- Given two polynomials of degree at most n,

$$P_A(x) = A_n x^n + \ldots + A_0; \qquad P_B(x) = B_n x^n + \ldots + B_0$$

1. convert them into value representation at $2n+1$ distinct points $x_0, x_1, \ldots, x_{2n}$:

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_{2n}, P_A(x_{2n}))\}$$

$$P_B(x) \leftrightarrow \{(x_0, P_B(x_0)), (x_1, P_B(x_1)), \ldots, (x_{2n}, P_B(x_{2n}))\}$$

2. multiply them point by point using $2n+1$ multiplications only:

Our strategy to multiply polynomials fast:

- Given two polynomials of degree at most n,

$$P_A(x) = A_n x^n + \ldots + A_0; \qquad P_B(x) = B_n x^n + \ldots + B_0$$

1. convert them into value representation at $2n+1$ distinct points $x_0, x_1, \ldots, x_{2n}$:

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_{2n}, P_A(x_{2n}))\}$$

$$P_B(x) \leftrightarrow \{(x_0, P_B(x_0)), (x_1, P_B(x_1)), \ldots, (x_{2n}, P_B(x_{2n}))\}$$

2. multiply them point by point using $2n+1$ multiplications only:

$$P_A(x)P_B(x) \leftrightarrow \{(x_0, \underbrace{P_A(x_0)P_B(x_0)}_{P_C(x_0)}), (x_1, \underbrace{P_A(x_1)P_B(x_1)}_{P_C(x_1)}), \ldots, (x_{2n}, \underbrace{P_A(x_{2n})P_B(x_{2n})}_{P_C(x_{2n})})\}$$

Our strategy to multiply polynomials fast:

- Given two polynomials of degree at most n,

$$P_A(x) = A_n x^n + \ldots + A_0; \qquad P_B(x) = B_n x^n + \ldots + B_0$$

1. convert them into value representation at $2n+1$ distinct points $x_0, x_1, \ldots, x_{2n}$:

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \ldots, (x_{2n}, P_A(x_{2n}))\}$$

$$P_B(x) \leftrightarrow \{(x_0, P_B(x_0)), (x_1, P_B(x_1)), \ldots, (x_{2n}, P_B(x_{2n}))\}$$

2. multiply them point by point using $2n+1$ multiplications only:

$$P_A(x)P_B(x) \leftrightarrow \{(x_0, \underbrace{P_A(x_0)P_B(x_0)}_{P_C(x_0)}), (x_1, \underbrace{P_A(x_1)P_B(x_1)}_{P_C(x_1)}), \ldots, (x_{2n}, \underbrace{P_A(x_{2n})P_B(x_{2n})}_{P_C(x_{2n})})\}$$

3. Convert such value representation of $P_C(x) = P_A(x)P_B(x)$ back to coefficient form

$$P_C(x) = C_{2n}x^{2n} + C_{2n-1}x^{2n-1} + \ldots + C_1 x + C_0;$$

Fast multiplication of polynomials - ctd.

- What values should we choose for $x_0, x_1, \ldots, x_{2n}$??

Fast multiplication of polynomials - ctd.

- What values should we choose for $x_0, x_1, \ldots, x_{2n}$??
- Key idea: use $2n+1$ smallest possible integer values!

Fast multiplication of polynomials - ctd.

- What values should we choose for $x_0, x_1, \ldots, x_{2n}$??
- Key idea: use $2n+1$ smallest possible integer values!

$$\{-n,\ -(n-1),\ \ldots\ ,\ -1,\ 0,\ 1,\ \ldots\ ,\ n-1,\ n\}$$

Fast multiplication of polynomials - ctd.

- What values should we choose for $x_0, x_1, \dots, x_{2n}$??
- Key idea: use $2n+1$ smallest possible integer values!

$$\{-n,\ -(n-1),\ \dots\ ,\ -1,\ 0,\ 1,\ \dots\ ,\ n-1,\ n\}$$

- Thus, we want to find the values

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \dots + A_0 : \quad -n \le m \le n,$$

$$P_B(m) = B_n m^n + B_{n-1} m^{n-1} + \dots + B_0 : \quad -n \le m \le n.$$

Fast multiplication of polynomials - ctd.

- What values should we choose for $x_0, x_1, \ldots, x_{2n}$??
- Key idea: use $2n+1$ smallest possible integer values!

$$\{-n,\ -(n-1),\ \ldots\ ,\ -1,\ 0,\ 1,\ \ldots\ ,\ n-1,\ n\}$$

- Thus, we want to find the values

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \cdots + A_0 : \quad -n \leq m \leq n,$$

$$P_B(m) = B_n m^n + B_{n-1} m^{n-1} + \cdots + B_0 : \quad -n \leq m \leq n.$$

- Recall: $n+1$ is the number of slices we split the input numbers A, B.

Fast multiplication of polynomials - ctd.

- What values should we choose for $x_0, x_1, \ldots, x_{2n}$??
- Key idea: use $2n+1$ smallest possible integer values!

$$\{-n,\ -(n-1),\ \ldots\ ,\ -1,\ 0,\ 1,\ \ldots\ ,\ n-1,\ n\}$$

- Thus, we want to find the values

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \cdots + A_0 : \quad -n \leq m \leq n,$$

$$P_B(m) = B_n m^n + B_{n-1} m^{n-1} + \cdots + B_0 : \quad -n \leq m \leq n.$$

- Recall: $n+1$ is the number of slices we split the input numbers A, B.

- Thus, n is constant; only coefficients A_i and B_i are large - they are the slices of A and B.

Fast multiplication of polynomials - ctd.

- What values should we choose for $x_0, x_1, \dots, x_{2n}$??
- Key idea: use $2n+1$ smallest possible integer values!

$$\{-n,\ -(n-1),\ \dots\ ,\ -1,\ 0,\ 1,\ \dots\ ,\ n-1,\ n\}$$

- Thus, we want to find the values

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \dots + A_0 : \quad -n \le m \le n,$$

$$P_B(m) = B_n m^n + B_{n-1} m^{n-1} + \dots + B_0 : \quad -n \le m \le n.$$

- Recall: $n+1$ is the number of slices we split the input numbers A, B.

- Thus, n is constant; only coefficients A_i and B_i are large - they are the slices of A and B.
- Values:

$$(-m)^n,\ (-m)^{n-1},\ \cdots\ ,\ (-m)^2,\ -m,\ m,\ m^2,\ \dots\ m^n$$

for $0 \le m \le n$ are all constants which do not depend on A or B!

Fast multiplication of polynomials - ctd.

- Multiplication of a large number with k bits by a constant d can be done in linear time because it is reducible to $d-1$ additions:

$$d \cdot A = \underbrace{A + A + \ldots + A}_{d}$$

Fast multiplication of polynomials - ctd.

- Multiplication of a large number with k bits by a constant d can be done in linear time because it is reducible to $d-1$ additions:

$$d \cdot A = \underbrace{A + A + \ldots + A}_{d}$$

- Thus, all the values

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \cdots + A_0 : \quad -n \leq m \leq n,$$

$$P_B(m) = B_n m^n + B_{n-1} m^{n-1} + \cdots + B_0 : \quad -n \leq m \leq n.$$

can be found in time linear in the number of bits of the input numbers!

Fast multiplication of polynomials - ctd.

- Multiplication of a large number with k bits by a constant d can be done in linear time because it is reducible to $d-1$ additions:

$$d \cdot A = \underbrace{A + A + \ldots + A}_{d}$$

- Thus, all the values

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \cdots + A_0 : \quad -n \leq m \leq n,$$

$$P_B(m) = B_n m^n + B_{n-1} m^{n-1} + \cdots + B_0 : \quad -n \leq m \leq n.$$

can be found in time linear in the number of bits of the input numbers!

- We now perform $2n-1$ multiplications of large numbers to obtain

$$P_A(-n)P_B(-n),\ \ldots\ ,\ P_A(-1)P_B(-1),\ P_A(0)P_B(0),\ P_A(1)P_B(1),\ \ldots\ ,\ P_A(n)P_B(n)$$

Fast multiplication of polynomials - ctd.

- Multiplication of a large number with k bits by a constant d can be done in linear time because it is reducible to $d-1$ additions:

$$d \cdot A = \underbrace{A + A + \ldots + A}_{d}$$

- Thus, all the values

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \cdots + A_0 : \quad -n \leq m \leq n,$$

$$P_B(m) = B_n m^n + B_{n-1} m^{n-1} + \cdots + B_0 : \quad -n \leq m \leq n.$$

can be found in time linear in the number of bits of the input numbers!

- We now perform $2n-1$ multiplications of large numbers to obtain

$$P_A(-n)P_B(-n), \ \ldots \ , \ P_A(-1)P_B(-1), \ P_A(0)P_B(0), \ P_A(1)P_B(1), \ \ldots \ , \ P_A(n)P_B(n)$$

- For $P_C(x) = P_A(x)P_B(x)$ these are $2n+1$ many values of $P_C(x)$:

$$P_C(-n), \ \ldots \ , \ P_C(-1), \ P_C(0), \ P_C(1), \ \ldots \ , \ P_C(n)$$

Fast multiplication of polynomials - ctd.

$$\begin{aligned}
P_C(-n) &= P_A(-n) \cdot P_B(-n) \\
P_C(-n+1) &= P_A(-n+1) \cdot P_B(-n+1) \\
&\dots \\
P_C(0) &= P_A(0) \cdot P_B(0) \\
&\dots \\
P_C(n-1) &= P_A(n-1) \cdot P_B(n-1) \\
P_C(n) &= P_A(n) \cdot P_B(n)
\end{aligned}$$

Fast multiplication of polynomials - ctd.

$$\begin{aligned}
P_C(-n) &= P_A(-n) \cdot P_B(-n) \\
P_C(-n+1) &= P_A(-n+1) \cdot P_B(-n+1) \\
&\dots \\
P_C(0) &= P_A(0) \cdot P_B(0) \\
&\dots \\
P_C(n-1) &= P_A(n-1) \cdot P_B(n-1) \\
P_C(n) &= P_A(n) \cdot P_B(n)
\end{aligned}$$

Since $P_C(x) = C_{2n}x^{2n} + C_{2n-1}x^{2n-1} + \dots + C_0$, we have:

Fast multiplication of polynomials - ctd.

$$
\begin{aligned}
P_C(-n) &= P_A(-n) \cdot P_B(-n) \\
P_C(-n+1) &= P_A(-n+1) \cdot P_B(-n+1) \\
&\dots \\
P_C(0) &= P_A(0) \cdot P_B(0) \\
&\dots \\
P_C(n-1) &= P_A(n-1) \cdot P_B(n-1) \\
P_C(n) &= P_A(n) \cdot P_B(n)
\end{aligned}
$$

Since $P_C(x) = C_{2n}x^{2n} + C_{2n-1}x^{2n-1} + \cdots + C_0$, we have:

$$
\begin{aligned}
C_{2n}(-n)^{2n} + C_{2n-1}(-n)^{2n-1} + \cdots + C_0 &= P_C(-n) \\
C_{2n}(-(n-1))^{2n} + C_{2n-1}(-(n-1))^{2n-1} + \cdots + C_0 &= P_C(-(n-1)) \\
&\dots \\
C_{2n}(n-1)^{2n} + C_{2n-1}(n-1)^{2n-1} + \cdots + C_0 &= P_C(n-1) \\
C_{2n}n^{2n} + C_{2n-1}n^{2n-1} + \cdots + C_0 &= P_C(n)
\end{aligned}
$$

This is just a system of linear equations, that can be solved for $C_0, C_1, \ldots, C_{2n}$:

Fast multiplication of polynomials - ctd.

$$\begin{pmatrix} 1 & -n & (-n)^2 & \dots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \dots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \dots & n^{2n} \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix},$$

Fast multiplication of polynomials - ctd.

$$\begin{pmatrix} 1 & -n & (-n)^2 & \ldots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \ldots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \ldots & n^{2n} \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix},$$

i.e.,

$$\begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} 1 & -n & (-n)^2 & \ldots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \ldots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \ldots & n^{2n} \end{pmatrix}^{-1} \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix}.$$

Fast multiplication of polynomials - ctd.

$$\begin{pmatrix} 1 & -n & (-n)^2 & \dots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \dots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \dots & n^{2n} \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix},$$

i.e.,

$$\begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} 1 & -n & (-n)^2 & \dots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \dots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \dots & n^{2n} \end{pmatrix}^{-1} \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix}.$$

But the inverse matrix also involves only constants depending on n only;

Fast multiplication of polynomials - ctd.

$$\begin{pmatrix} 1 & -n & (-n)^2 & \dots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \dots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \dots & n^{2n} \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix},$$

i.e.,

$$\begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} 1 & -n & (-n)^2 & \dots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \dots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \dots & n^{2n} \end{pmatrix}^{-1} \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix}.$$

But the inverse matrix also involves only constants depending on n only;

Thus the coefficients C_i can be obtained in linear time.

So herre is the algorithm we have just described:

function MULT(n, A, B)
  if $|A| = |B| = 1$ **then return** AB
  else
    obtain $A_0, A_1, \ldots, A_n$ and $B_0, B_1, \ldots, B_n$ such that

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

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

    form polynomials

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

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

    for $m = -n$ to $m = n$ **do**
      compute $P_A(m)$ and $P_B(m)$;
      $P_C(m) \leftarrow$ MULT($n, P_A(m) P_B(m)$)
    end for
    compute $C_0,\ C_1, \ldots\ C_{2n}$ via

$$\begin{pmatrix} C_0 \\ C_1 \\ \vdots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} 1 & -n & (-n)^2 & \ldots & (-n)^{2n} \\ 1 & -(n-1) & (-(n-1))^2 & \ldots & (-(n-1))^{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & n & n^2 & \ldots & n^{2n} \end{pmatrix}^{-1} \begin{pmatrix} P_C(-n) \\ P_C(-(n-1)) \\ \vdots \\ P_C(n) \end{pmatrix}.$$

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

How fast is our algorithm?

- For each m such that $-n \leq m \leq n$, the value of $P_A(m)$ is a sum of $n+1$ values;
- Each value is a product of a k-bit number A_j with a constant m^j, $-n \leq m \leq n$:

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \ldots A_0$$

How fast is our algorithm?

- For each m such that $-n \leq m \leq n$, the value of $P_A(m)$ is a sum of $n+1$ values;
- Each value is a product of a k-bit number A_j with a constant m^j, $-n \leq m \leq n$:

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \ldots A_0$$

- So $|P_A(m)| \leq (n+1) \cdot 2^k \cdot n^n$.

How fast is our algorithm?

- For each m such that $-n \leq m \leq n$, the value of $P_A(m)$ is a sum of $n+1$ values;
- Each value is a product of a k-bit number A_j with a constant m^j, $-n \leq m \leq n$:

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \ldots A_0$$

- So $|P_A(m)| \leq (n+1) \cdot 2^k \cdot n^n$.
- The number of bits of each $P_A(m)$ for $-n \leq m \leq n$ is

$$\log_2\left((n+1)2^k n^n\right) = k + \log_2\left((n+1) \cdot n^n\right)$$

How fast is our algorithm?

- For each m such that $-n \leq m \leq n$, the value of $P_A(m)$ is a sum of $n+1$ values;
- Each value is a product of a k-bit number A_j with a constant m^j, $-n \leq m \leq n$:

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \ldots A_0$$

- So $|P_A(m)| \leq (n+1) \cdot 2^k \cdot n^n$.
- The number of bits of each $P_A(m)$ for $-n \leq m \leq n$ is

$$\log_2 \left((n+1)2^k n^n\right) = k + \log_2 \left((n+1) \cdot n^n\right)$$

- $\log((n+1) \cdot n^n)$ is constant;

How fast is our algorithm?

- For each m such that $-n \leq m \leq n$, the value of $P_A(m)$ is a sum of $n+1$ values;
- Each value is a product of a k-bit number A_j with a constant m^j, $-n \leq m \leq n$:

$$P_A(m) = A_n m^n + A_{n-1} m^{n-1} + \ldots A_0$$

- So $|P_A(m)| \leq (n+1) \cdot 2^k \cdot n^n$.
- The number of bits of each $P_A(m)$ for $-n \leq m \leq n$ is

$$\log_2\left((n+1)2^k n^n\right) = k + \log_2\left((n+1) \cdot n^n\right)$$

- $\log((n+1) \cdot n^n)$ is constant; Thus,
- **Each $P_A(m)$ has $k+s$ bits, where s is a constant independent of k.**

Generalizing Karatsuba's algorithm

- Recall that

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \underbrace{\left(\sum_{i=0}^{j} A_i B_{j-i} \right)} x^j = \sum_{j=0}^{2n} \underbrace{C_j} \, x^j \qquad \text{where} \qquad C_j = \sum_{i=0}^{j} A_i B_{j-i}$$

Generalizing Karatsuba's algorithm

- Recall that

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \underbrace{\left(\sum_{i=0}^{j} A_i B_{j-i} \right)} x^j = \sum_{j=0}^{2n} \underbrace{C_j}\ x^j \qquad \text{where} \qquad C_j = \sum_{i=0}^{j} A_i B_{j-i}$$

- How big are C_j? Since $A_i, B_i < 2^k$, we have $A_i B_{j-i} < 2^{2k}$ and

$$C_j = \sum_{i=0}^{j} A_i B_{j-i} < j 2^{2k} \le (n+1) 2^{2k}$$

Generalizing Karatsuba's algorithm

- Recall that

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \underbrace{\left(\sum_{i=0}^{j} A_i B_{j-i} \right)} x^j = \sum_{j=0}^{2n} \underbrace{C_j}\, x^j \qquad \text{where} \qquad C_j = \sum_{i=0}^{j} A_i B_{j-i}$$

- How big are C_j? Since $A_i, B_i < 2^k$, we have $A_i B_{j-i} < 2^{2k}$ and

$$C_j = \sum_{i=0}^{j} A_i B_{j-i} < j 2^{2k} \leq (n+1) 2^{2k}$$

- Thus, C_j **has at most** $2k + \log_2(n+1)$ **many bits**

Generalizing Karatsuba's algorithm

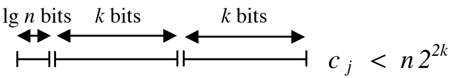
- Recall that

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \underbrace{\left(\sum_{i=0}^{j} A_i B_{j-i} \right)}_{} x^j = \sum_{j=0}^{2n} \underbrace{C_j}_{} \; x^j \qquad \text{where} \qquad C_j = \sum_{i=0}^{j} A_i B_{j-i}$$

- How big are C_j? Since $A_i, B_i < 2^k$, we have $A_i B_{j-i} < 2^{2k}$ and

$$C_j = \sum_{i=0}^{j} A_i B_{j-i} < j 2^{2k} \leq (n+1) 2^{2k}$$

- Thus, C_j **has at most** $2k + \log_2(n+1)$ **many bits**



How fast is our algorithm?

- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$

How fast is our algorithm?

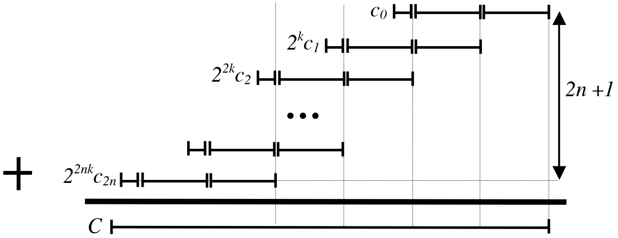
- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$
- C is obtained as a sum of binary shifts of $2n+1$ numbers C_j;

How fast is our algorithm?

- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$
- C is obtained as a sum of binary shifts of $2n + 1$ numbers C_j;
- each C_j has $2k + \lceil \log_2(n+1) \rceil$ bits;

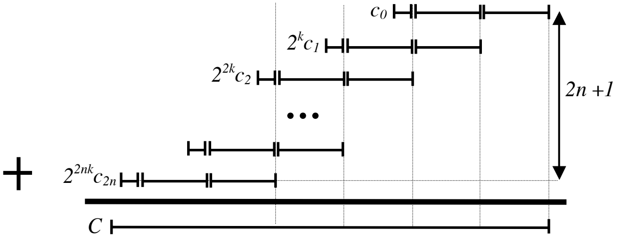
How fast is our algorithm?

- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$
- C is obtained as a sum of binary shifts of $2n+1$ numbers C_j;
- each C_j has $2k + \lceil \log_2(n+1) \rceil$ bits;



How fast is our algorithm?

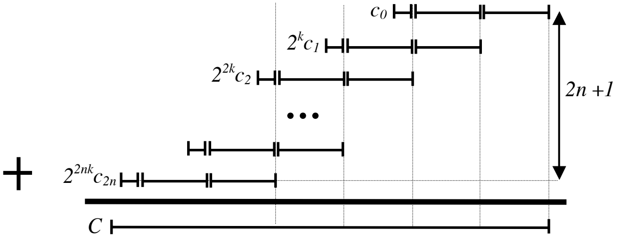
- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$
- C is obtained as a sum of binary shifts of $2n+1$ numbers C_j;
- each C_j has $2k + \lceil \log_2(n+1) \rceil$ bits;



- n is constant: it is the number of pieces we slice input numbers;

How fast is our algorithm?

- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$
- C is obtained as a sum of binary shifts of $2n+1$ numbers C_j;
- each C_j has $2k + \lceil \log_2(n+1) \rceil$ bits;



- n is constant: it is the number of pieces we slice input numbers;
- Thus, **evaluation of $P_C(2^k)$ takes $O(k)$ many steps.**

How fast is our algorithm?

- We have reduced a multiplication of two $k(n+1)$ digit numbers to $2n+1$ multiplications of $k+s$ digit numbers plus a linear overhead (of additions splitting the numbers atc.)

How fast is our algorithm?

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

How fast is our algorithm?

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

$$T((n+1)k) = (2n+1)T(k+s) + c\,k$$

How fast is our algorithm?

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

$$T((n+1)k) = (2n+1)T(k+s) + c\,k$$

- Let $N = (n+1)k$. Then for another constant d

$$T(N) = (2n+1)\,T\left(\frac{N}{n+1} + s\right) + d\,N$$

How fast is our algorithm?

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

$$T((n+1)k) = (2n+1)T(k+s) + c\,k$$

- Let $N = (n+1)k$. Then for another constant d

$$T(N) = (2n+1)\,T\left(\frac{N}{n+1} + s\right) + d\,N$$

- Since s is constant, its impact can be neglected.
- Since $\log_b a = \log_{n+1}(2n+1) > 1$,

$$f(N) = c\,N = O\left(N^{\log_{n+1}(2n+1)-\varepsilon}\right)$$

How fast is our algorithm?

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

$$T((n+1)k) = (2n+1)T(k+s) + c\,k$$

- Let $N = (n+1)k$. Then for another constant d

$$T(N) = (2n+1)\,T\left(\frac{N}{n+1} + s\right) + d\,N$$

- Since s is constant, its impact can be neglected.
- Since $\log_b a = \log_{n+1}(2n+1) > 1$,

$$f(N) = c\,N = O\left(N^{\log_{n+1}(2n+1)-\varepsilon}\right)$$

- Thus, with $a = 2n+1$ and $b = n+1$ the first case of the Master Theorem applies;
- we get:

$$T(N) = \Theta\left(N^{\log_b a}\right) = \Theta\left(N^{\log_{n+1}(2n+1)}\right)$$

- Note that

$$N^{\log_{n+1}(2n+1)} < N^{\log_{n+1} 2(n+1)} = N^{\log_{n+1} 2+\log_{n+1}(n+1)}$$

$$= N^{1+\log_{n+1} 2} = N^{1+\frac{1}{\log_2(n+1)}}$$

- Note that

$$N^{\log_{n+1}(2n+1)} < N^{\log_{n+1} 2(n+1)} = N^{\log_{n+1} 2+\log_{n+1}(n+1)}$$
$$= N^{1+\log_{n+1} 2} = N^{1+\frac{1}{\log_2(n+1)}}$$

- Thus, by choosing a sufficiently large n, we can get a run time arbitrarily close to linear time!

- Note that

$$N^{\log_{n+1}(2n+1)} < N^{\log_{n+1}2(n+1)} = N^{\log_{n+1}2+\log_{n+1}(n+1)}$$
$$= N^{1+\log_{n+1}2} = N^{1+\frac{1}{\log_2(n+1)}}$$

- Thus, by choosing a sufficiently large n, we can get a run time arbitrarily close to linear time!

- How large does n have to be, in order to to get an algorithm which runs in time $N^{1.1}$?

- Note that

$$N^{\log_{n+1}(2n+1)} < N^{\log_{n+1} 2(n+1)} = N^{\log_{n+1} 2+\log_{n+1}(n+1)}$$
$$= N^{1+\log_{n+1} 2} = N^{1+\frac{1}{\log_2(n+1)}}$$

- Thus, by choosing a sufficiently large n, we can get a run time arbitrarily close to linear time!

- How large does n have to be, in order to to get an algorithm which runs in time $N^{1.1}$?

$$N^{1.1} = N^{1+\frac{1}{\log_2(n+1)}} \quad \rightarrow \quad \frac{1}{\log_2(n+1)} = \frac{1}{10} \quad \rightarrow \quad n+1 = 2^{10}$$

- Note that

$$N^{\log_{n+1}(2n+1)} < N^{\log_{n+1} 2(n+1)} = N^{\log_{n+1} 2+\log_{n+1}(n+1)}$$
$$= N^{1+\log_{n+1} 2} = N^{1+\frac{1}{\log_2(n+1)}}$$

- Thus, by choosing a sufficiently large n, we can get a run time arbitrarily close to linear time!

- How large does n have to be, in order to to get an algorithm which runs in time $N^{1.1}$?

$$N^{1.1} = N^{1+\frac{1}{\log_2(n+1)}} \quad \rightarrow \quad \frac{1}{\log_2(n+1)} = \frac{1}{10} \quad \rightarrow \quad n+1 = 2^{10}$$

- Thus, we would have to slice input numbers into $2^{10} = 1024$ pieces.

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ at values up to $n = 2^{10} - 1$, which involves computing

$$n^n = (2^{10} - 1)^{2^{10}-1} = 1.27 \times 10^{3079}$$

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ at values up to $n = 2^{10} - 1$, which involves computing

$$n^n = (2^{10} - 1)^{2^{10}-1} = 1.27 \times 10^{3079}$$

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ can be all done in linear time $T(n) = c\,n$, the constant c is huge;

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ at values up to $n = 2^{10} - 1$, which involves computing

$$n^n = (2^{10} - 1)^{2^{10}-1} = 1.27 \times 10^{3079}$$

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ can be all done in linear time $T(n) = c\,n$, the constant c is huge;

- the algorithm is not practical at all!

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ at values up to $n = 2^{10} - 1$, which involves computing

$$n^n = (2^{10} - 1)^{2^{10}-1} = 1.27 \times 10^{3079}$$

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ can be all done in linear time $T(n) = c\,n$, the constant c is huge;
- the algorithm is not practical at all!
- The moral is: **asymptotic estimates are useless in practice if the size of the constants hidden by the O-notation are not estimated and found to be reasonably small!!!**

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ at values up to $n = 2^{10} - 1$, which involves computing

$$n^n = (2^{10} - 1)^{2^{10}-1} = 1.27 \times 10^{3079}$$

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ can be all done in linear time $T(n) = c\,n$, the constant c is huge;

- the algorithm is not practical at all!

- The moral is: **asymptotic estimates are useless in practice if the size of the constants hidden by the O-notation are not estimated and found to be reasonably small!!!**

- Can we avoid explosion in the size of x^n needed for evaluation of a polynomial of degree n? Are there numbers $x_0, x_1, \ldots, x_n$ such that the size of x_i^n does not grow uncontrollably?

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ at values up to $n = 2^{10} - 1$, which involves computing

$$n^n = (2^{10} - 1)^{2^{10}-1} = 1.27 \times 10^{3079}$$

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ can be all done in linear time $T(n) = c\,n$, the constant c is huge;
- the algorithm is not practical at all!
- The moral is: **asymptotic estimates are useless in practice if the size of the constants hidden by the O-notation are not estimated and found to be reasonably small!!!**
- Can we avoid explosion in the size of x^n needed for evaluation of a polynomial of degree n? Are there numbers $x_0, x_1, \ldots, x_n$ such that the size of x_i^n does not grow uncontrollably?
- Answer: YES; they are the complex numbers z_i lying on the unit circle, i.e., such that $|z_i| = 1$!

Digression: Convolution

- Let $a = \langle A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$ and $b = \langle B_n, B_{n-1}, \ldots B_1, B_0 \rangle$ be two sequences;
- pad them with n $0's$ to length $2n+1$:

$$\langle \underbrace{0, 0, \ldots, 0}_{n}, A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$$

Digression: Convolution

- Let $a = \langle A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$ and $b = \langle B_n, B_{n-1}, \ldots B_1, B_0 \rangle$ be two sequences;
- pad them with n $0's$ to length $2n+1$:

$$\langle \underbrace{0, 0, \ldots, 0}_{n}, A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$$

$$\langle \underbrace{0, 0, \ldots, 0}_{n}, B_n, B_{n-1}, \ldots, B_1, B_0 \rangle$$

Digression: Convolution

- Let $a = \langle A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$ and $b = \langle B_n, B_{n-1}, \ldots B_1, B_0 \rangle$ be two sequences;
- pad them with n $0's$ to length $2n + 1$:

$$\langle \underbrace{0, 0, \ldots, 0}_{n}, A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$$

$$\langle \underbrace{0, 0, \ldots, 0}_{n}, B_n, B_{n-1}, \ldots, B_1, B_0 \rangle$$

i.e., we set $A_i = B_i = 0$ for $n < i \leq 2n$.

Digression: Convolution

- Let $a = \langle A_n, A_{n-1}, \dots, A_1, A_0 \rangle$ and $b = \langle B_n, B_{n-1}, \dots B_1, B_0 \rangle$ be two sequences;
- pad them with n $0's$ to length $2n+1$:

$$\langle \underbrace{0, 0, \dots, 0}_{n}, A_n, A_{n-1}, \dots, A_1, A_0 \rangle$$

$$\langle \underbrace{0, 0, \dots, 0}_{n}, B_n, B_{n-1}, \dots, B_1, B_0 \rangle$$

i.e., we set $A_i = B_i = 0$ for $n < i \leq 2n$.

Then the sequence $$a * b = \left\{ \sum_{i=0}^{j} A_i B_{j-i} \right\}_{j=0}^{2n}$$

is called *the (linear) convolution* of the sequences A and B.

Digression: Convolution

- Let $a = \langle A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$ and $b = \langle B_n, B_{n-1}, \ldots B_1, B_0 \rangle$ be two sequences;
- pad them with n $0's$ to length $2n + 1$:

$$\langle \underbrace{0, 0, \ldots, 0}_{n}, A_n, A_{n-1}, \ldots, A_1, A_0 \rangle$$

$$\langle \underbrace{0, 0, \ldots, 0}_{n}, B_n, B_{n-1}, \ldots, B_1, B_0 \rangle$$

i.e., we set $A_i = B_i = 0$ for $n < i \leq 2n$.

Then the sequence $$a * b = \left\{ \sum_{i=0}^{j} A_i B_{j-i} \right\}_{j=0}^{2n}$$

is called *the (linear) convolution* of the sequences A and B.

$$a * b = \langle A_n B_n,\ A_{n-1} B_n + A_n B_{n-1},\ A_{n-2} B_n + A_{n-1} B_{n-1} + A_n B_{n-2},$$
$$\ldots, A_2 B_0 + A_1 B_1 + A_0 B_2,\ A_1 B_0 + A_0 B_1,\ A_0 B_0 \rangle$$