Algorithms: COMP3121/3821/9101/9801

Aleks Ignjatović

School of Computer Science and Engineering
University of New South Wales

TOPIC 2: 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:

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

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

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

The Karatsuba trick

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

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

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

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

The Karatsuba trick

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

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

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

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

- 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:

$$\begin{aligned} AB &= A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0 \\ &= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0 \end{aligned}$$

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

  if $|A| = |B| = 1$ **then return** AB

  else

    $A_1 \leftarrow \text{MoreSignificantPart}(A)$;

    $A_0 \leftarrow \text{LessSignificantPart}(A)$;

    $B_1 \leftarrow \text{MoreSignificantPart}(B)$;

    $B_0 \leftarrow \text{LessSignificantPart}(B)$;

    $U \leftarrow A_0 + A_1$;

    $V \leftarrow B_0 + B_1$;

    $X \leftarrow \textsc{Mult}(\mathrm{A}_0, \mathrm{B}_0)$;

    $W \leftarrow \textsc{Mult}(\mathrm{A}_1, \mathrm{B}_1)$;

    $Y \leftarrow \textsc{Mult}(\mathrm{U}, \mathrm{V})$;

    return $W\,2^n + (Y - X - W)\,2^{n/2} + X$

  end if

end function

The Karatsuba trick

- How many steps does this take? (remember, addition is in linear time!)

The Karatsuba trick

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

The Karatsuba trick

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

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

- since $\quad 1.5 < \log_2 3 < 1.6$ 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 steps does this take? (remember, addition is in linear time!)
- Recurrence: $$T(n) = 3\,T\left(\frac{n}{2}\right) + c\,n$$

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

- since $\quad 1.5 < \log_2 3 < 1.6$ 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 steps does this take? (remember, 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 $\quad 1.5 < \log_2 3 < 1.6$ 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; then with $k = n/3$ we obtain

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; then with $k = n/3$ we obtain

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

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; 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.,

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

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; 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.,

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

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

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; then with $k = n/3$ we obtain

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

i.e.,

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

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

- So,

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

The Karatsuba trick

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

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:

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

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

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

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

- Not clear at all 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!

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:

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

- Note that

$$A = A_2\, (2^k)^2 + A_1\, 2^k + A_0 = P_A(2^k);$$
$$B = B_2\, (2^k)^2 + B_1\, 2^k + B_0 = P_B(2^k).$$

The Karatsuba trick: slicing into 3 pieces

- If we manage to compute somehow the product polynomial

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

The Karatsuba trick: slicing into 3 pieces

- If we manage to compute somehow the product polynomial

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

with only 5 multiplications, we can then obtain the product of numbers A and B simply as

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

The Karatsuba trick: slicing into 3 pieces

- If we manage to compute somehow the product polynomial

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

  with only 5 multiplications, we can then obtain the product of numbers A and B simply as

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

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

The Karatsuba trick: slicing into 3 pieces

- If we manage to compute somehow the product polynomial

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

  with only 5 multiplications, we can then obtain the product of numbers A and B simply as

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

- Note that the right hand side involves only shifts and additions.
- Since the product polynomial $P_C(x) = P_A(x)P_B(x)$ is of degree 4 we need 5 values to **uniquely determine** $P_C(x)$.

The Karatsuba trick: slicing into 3 pieces

- If we manage to compute somehow the product polynomial

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

with only 5 multiplications, we can then obtain the product of numbers A and B simply as

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

- Note that the right hand side involves only shifts and additions.
- Since the product polynomial $P_C(x) = P_A(x)P_B(x)$ is of degree 4 we need 5 values to **uniquely determine** $P_C(x)$.
- We choose **the smallest possible** 5 **integer values** (smallest by their absolute value),

The Karatsuba trick: slicing into 3 pieces

- If we manage to compute somehow the product polynomial

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

  with only 5 multiplications, we can then obtain the product of numbers A and B simply as

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

- Note that the right hand side involves only shifts and additions.
- Since the product polynomial $P_C(x) = P_A(x)P_B(x)$ is of degree 4 we need 5 values to **uniquely determine** $P_C(x)$.
- We choose **the smallest possible** 5 **integer values** (smallest by their absolute value), i.e., -2, -1, 0, 1, 2.

The Karatsuba trick: slicing into 3 pieces

- If we manage to compute somehow the product polynomial

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

  with only 5 multiplications, we can then obtain the product of numbers A and B simply as

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

- Note that the right hand side involves only shifts and additions.
- Since the product polynomial $P_C(x) = P_A(x)P_B(x)$ is of degree 4 we need 5 values to **uniquely determine** $P_C(x)$.
- We choose **the smallest possible** 5 **integer values** (smallest by their absolute value), 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

- 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

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

- These evaluations involve only additions because $2A = A + A$; $4A = 2A + 2A$.

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 obtain $P_C(-2), P_C(-1), P_C(0), P_C(1), P_C(2)$ with only 5 multiplications of large numbers:

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

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

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

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

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

The Karatsuba trick: slicing into 3 pieces

- Thus, if we represent the product $C(x) = P_A(x)P_B(x)$ in the coefficient form as $C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0$ we get

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

The Karatsuba trick: slicing into 3 pieces

- Thus, if we represent the product $C(x) = P_A(x)P_B(x)$ in the coefficient form as $C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0$ we get

$$\begin{aligned}
C_4(-2)^4 + C_3(-2)^3 + C_2(-2)^2 + C_1(-2) + C_0 &= P_C(-2) = P_A(-2)P_B(-2)\\
C_4(-1)^4 + C_3(-1)^3 + C_2(-1)^2 + C_1(-1) + C_0 &= P_C(-1) = P_A(-1)P_B(-1)\\
C_40^4 + C_30^3 + C_20^2 + C_1 \cdot 0 + C_0 &= P_C(0) = P_A(0)P_B(0)\\
C_41^4 + C_31^3 + C_21^2 + C_1 \cdot 1 + C_0 &= P_C(1) = P_A(1)P_B(1)\\
C_42^4 + C_32^3 + C_22^2 + C_1 \cdot 2 + C_0 &= P_C(2) = P_A(2)P_B(2).
\end{aligned}$$

- Doing the simplifications we obtain

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

The Karatsuba trick: slicing into 3 pieces

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 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

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 we obtain

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

- Note that these expressions do not involve any multiplications of TWO large numbers and thus can be done in linear time.

The Karatsuba trick: slicing into 3 pieces

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 we obtain

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

- Note that these expressions do not involve any multiplications of TWO large numbers and thus can be done in linear time.
- 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

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 we obtain

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

- Note that these expressions do not involve any multiplications of TWO large numbers and thus can be done in linear time.
- With the coefficients C_0, C_1, C_2, C_3, C_4 obtained, we can now form the polynomial $P_C(x) = C_0 + C_1x + C_2x^2 + C_3x^3 + C_4x^4$.
- 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

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 we obtain

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

- Note that these expressions do not involve any multiplications of TWO large numbers and thus can be done in linear time.
- 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.
- 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

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 we obtain

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

- Note that these expressions do not involve any multiplications of TWO large numbers and thus can be done in linear time.
- 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.
- Thus we have obtained $A \cdot B = P_A(2^k) P_B(2^k) = P_C(2^k)$ with only 5 multiplications!
- 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 $P_A(x) = A_2 x^2 + A_1 x + A_0$; $\quad P_B(x) = B_2 x^2 + B_1 x + B_0$;

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

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

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

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 $P_C(2^k) = A \cdot B$.

end function

The Karatsuba trick: slicing into 3 pieces

- How fast is this algorithm?

- 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:

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 $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465...}$

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 $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465...}$
- Clearly, the first case of the MT applies and we get $T(n) = O(n^{\log_3 5}) < O(n^{1.47})$.

The Karatsuba trick: slicing into 3 pieces

- Recall that the original Karatsuba algorithm runs in time

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

- 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

- 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 A and B into even larger number of slices? Maybe we can get even faster algorithm?

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.
- Then why not slice numbers A and B 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 the input numbers A, B into $n + 1$ many slices

The general case - slicing the 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 the 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:

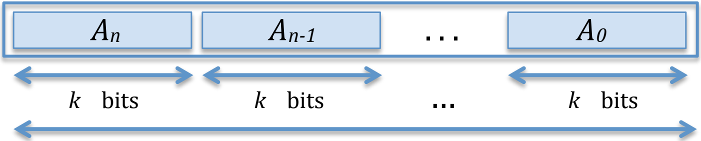
$$\begin{aligned} A &= A_n 2^{kn} + A_{n-1} 2^{k(n-1)} + \cdots + A_0 \\ B &= B_n 2^{kn} + B_{n-1} 2^{k(n-1)} + \cdots + B_0 \end{aligned}$$

Generalizing Karatsuba's algorithm

The general case - slicing the 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:

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



A divided into $n+1$ slices each slice k bits = $(n+1)\,k$ bits in total

- We form the naturally corresponding polynomials:

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

Generalizing Karatsuba's algorithm

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$

Generalizing Karatsuba's algorithm

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$
- $A = P_A(2^k);\;\; B = P_B(2^k);\;\; AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

Generalizing Karatsuba's algorithm

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$
- $A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$
- Since

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

Generalizing Karatsuba's algorithm

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$
- $A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$
- Since
$$AB = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$
we adopt the following strategy:

Generalizing Karatsuba's algorithm

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$
- $A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$
- Since
$$AB = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$
  we adopt the following strategy:
  - we will first figure out how to multiply polynomials fast to obtain
$$P_C(x) = P_A(x) \cdot P_B(x);$$

Generalizing Karatsuba's algorithm

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$
- $A = P_A(2^k);\;\; B = P_B(2^k);\;\; AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$
- Since
$$AB = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$
we adopt the following strategy:
  - we will first figure out how to multiply polynomials fast to obtain
$$P_C(x) = P_A(x) \cdot P_B(x);$$
  - then we evaluate $P_C(2^k)$.

Generalizing Karatsuba's algorithm

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$
- $A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$
- Since
$$AB = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$
we adopt the following strategy:
    - we will first figure out how to multiply polynomials fast to obtain
$$P_C(x) = P_A(x) \cdot P_B(x);$$
    - then we 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

- Since the coefficients A_i, B_i have k bits, they satisfy $A_i, B_i < 2^k$
- $A = P_A(2^k);\ \ B = P_B(2^k);\ \ AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$
- Since
$$AB = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$$
we adopt the following strategy:
    - we will first figure out how to multiply polynomials fast to obtain
    $$P_C(x) = P_A(x) \cdot P_B(x);$$
    - then we 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:

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

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

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

we have

Generalizing Karatsuba's algorithm

- Example:

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

- In general: for

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

we have

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

Generalizing Karatsuba's algorithm

- Example:

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

- In general: for

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

we have

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

- We need to find the coefficients $C_j = \sum_{i+k=j} A_iB_k$ without performing $(n+1)^2$ many multiplications necessary to get all products of the form A_iB_k.

A VERY IMPORTANT DIGRESSION:

If you have two sequences $\vec{A} = (A_0, A_1, \ldots, A_{n-1}, A_n)$ and $\vec{B} = (B_0, B_1, \ldots, B_{m-1}, B_m)$, and if you form the two corresponding polynomials

$$P_A = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_1 x + A_0$$
$$P_B = B_m x^m + B_{m-1} x^{m-1} + \ldots + B_1 x + B_0$$

A VERY IMPORTANT DIGRESSION:

If you have two sequences $\vec{A} = (A_0, A_1, \ldots, A_{n-1}, A_n)$ and $\vec{B} = (B_0, B_1, \ldots, B_{m-1}, B_m)$, and if you form the two corresponding polynomials

$$P_A = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_1 x + A_0$$
$$P_B = B_m x^m + B_{m-1} x^{m-1} + \ldots + B_1 x + B_0$$

and if you multiply these two polynomials to obtain their product

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{m+n} \left(\sum_{i+k=j} A_i B_k \right) x^j = \sum_{j=0}^{n+m} C_j \, x^j$$

A VERY IMPORTANT DIGRESSION:

If you have two sequences $\vec{A} = (A_0, A_1, \ldots, A_{n-1}, A_n)$ and $\vec{B} = (B_0, B_1, \ldots, B_{m-1}, B_m)$, and if you form the two corresponding polynomials

$$P_A = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_1 x + A_0$$
$$P_B = B_m x^m + B_{m-1} x^{m-1} + \ldots + B_1 x + B_0$$

and if you multiply these two polynomials to obtain their product

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{m+n} \left(\sum_{i+k=j} A_i B_k \right) x^j = \sum_{j=0}^{n+m} C_j \, x^j$$

then the sequence $\vec{C} = (C_0, C_1, \ldots, C_{n+m})$ of the coefficients of the product polynomial, with these coefficients given by

$$C_j = \sum_{i+k=j} A_i B_k, \quad \text{for} \quad 0 \leq j \leq n+m,$$

is **EXTREMELY IMPORTANT** and is called **the linear convolution** of sequences $\vec{A}$ and $\vec{B}$ and is denoted by $\vec{C} = \vec{A} \star \vec{B}$.

A VERY IMPORTANT DIGRESSION:

- 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.

A VERY IMPORTANT DIGRESSION:

- 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.

A VERY IMPORTANT DIGRESSION:

- 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:

A VERY IMPORTANT DIGRESSION:

- 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;

A VERY IMPORTANT DIGRESSION:

- 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 of what kind of filtering you want to do).

A VERY IMPORTANT DIGRESSION:

- 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 of what kind of filtering you want to do).
- Convolutions are 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.

A VERY IMPORTANT DIGRESSION:

- 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 of what kind of filtering you want to do).
- Convolutions are 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.

A VERY IMPORTANT DIGRESSION:

- 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 of what kind of filtering you want to do).
- Convolutions are 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).

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 $x_0, x_1, \ldots, x_n$:

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

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 $x_0, x_1, \ldots, x_n$:

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

- For $P_A(x) = A_n x^n + A_{n-1}x^{n-1} + \ldots + A_0$, these values can be obtained via a matrix multiplication:

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

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 $x_0, x_1, \ldots, x_n$:

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

- For $P_A(x) = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_0$, these values can be obtained via a matrix multiplication:

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

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

Coefficient vs value representation of polynomials

- Every polynomial $P_A(x)$ of degree n is uniquely determined by its values at any $n+1$ distinct input values $x_0, x_1, \ldots, x_n$:

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

- For $P_A(x) = A_n x^n + A_{n-1} x^{n-1} + \ldots + A_0$, these values can be obtained via a matrix multiplication:

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

- It can be shown that, if x_i are all distinct then this matrix is invertible.
- Such a matrix is called *the Vandermonde matrix*.

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are all distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ of the polynomial $P_A(x)$ 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 all distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ of the polynomial $P_A(x)$ 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 all distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ of the polynomial $P_A(x)$ 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 all distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_n)$ the coefficients $A_0, A_1, \ldots, A_n$ of the polynomial $P_A(x)$ 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.

- If we fix the inputs $x_0, x_1, \ldots, x_n$ then commuting between a representation of a polynomial $P_A(x)$ via its coefficients and a representation via its values at these points 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.

- If we fix the inputs $x_0, x_1, \dots, x_n$ then commuting between a representation of a polynomial $P_A(x)$ via its coefficients and a representation via its values at these points is done via the following two matrix multiplications, with matrices made up from **constants**:

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

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

Coefficient vs value representation of polynomials- ctd.

- If we fix the inputs $x_0, x_1, \ldots, x_n$ then commuting between a representation of a polynomial $P_A(x)$ via its coefficients and a representation via its values at these points 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:

1. 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:

1. Given two polynomials of degree at most n,

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

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:

1. Given two polynomials of degree at most n,

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

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

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

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

- **Note:** since the product of the two polynomials will be of degree $2n$ we need the values of $P_A(x)$ and $P_B(x)$ at $2n+1$ points, rather than just $n+1$ points!

Our strategy to multiply polynomials fast:

1. Given two polynomials of degree at most n,

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

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

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

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

   - **Note:** since the product of the two polynomials will be of degree $2n$ we need the values of $P_A(x)$ and $P_B(x)$ at $2n+1$ points, rather than just $n+1$ points!

2. Multiply these two polynomials point-wise, using $2n+1$ multiplications only.

Our strategy to multiply polynomials fast:

1. Given two polynomials of degree at most n,

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

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

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

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

   - **Note:** since the product of the two polynomials will be of degree $2n$ we need the values of $P_A(x)$ and $P_B(x)$ at $2n+1$ points, rather than just $n+1$ points!

2. Multiply these two polynomials point-wise, 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:

1. Given two polynomials of degree at most n,

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

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

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

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

- **Note:** since the product of the two polynomials will be of degree $2n$ we need the values of $P_A(x)$ and $P_B(x)$ at $2n+1$ points, rather than just $n+1$ points!

2. Multiply these two polynomials point-wise, 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 - continued

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

Fast multiplication of polynomials - continued

- 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 - continued

- 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 - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-n \leq m \leq n$.
- Remember that $n+1$ is the number of slices we split the input numbers A, B.

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-n \leq m \leq n$.
- Remember that $n+1$ is the number of slices we split the input numbers A, B.
- Multiplication of a large number with k bits by a constant integer d can be done in time linear in k because it is reducible to $d-1$ additions:

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-n \leq m \leq n$.
- Remember that $n+1$ is the number of slices we split the input numbers A, B.
- Multiplication of a large number with k bits by a constant integer d can be done in time linear in k 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.

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

$$P_A(-n)P_B(-n), \ \dots\ , \ P_A(-1)P_B(-1), \quad P_A(0)P_B(0), \quad P_A(1)P_B(1), \ \dots\ , \ P_A(n)P_B(n)$$

Fast multiplication of polynomials - ctd.

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

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

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

$$P_C(-n) = P_A(-n)P_B(-n),\ \dots\ ,\ P_C(0) = P_A(0)P_B(0),\ \dots\ ,\ P_C(n) = P_A(n)P_B(n)$$

Fast multiplication of polynomials - ctd.

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

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

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

$$P_C(-n) = P_A(-n)P_B(-n),\ \dots\ ,\ P_C(0) = P_A(0)P_B(0),\ \dots\ ,\ P_C(n) = P_A(n)P_B(n)$$

- Let $C_0, C_1, \dots, C_{2n}$ be the coefficients of the product polynomial $C(x)$, i.e., let

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

Fast multiplication of polynomials - ctd.

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

$$P_A(-n)P_B(-n),\ \dots\ ,\ P_A(-1)P_B(-1),\quad P_A(0)P_B(0),\quad P_A(1)P_B(1),\ \dots\ ,\ P_A(n)P_B(n)$$

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

$$P_C(-n) = P_A(-n)P_B(-n),\ \dots\ ,\ P_C(0) = P_A(0)P_B(0),\ \dots\ ,\ P_C(n) = P_A(n)P_B(n)$$

- Let $C_0, C_1, \dots, C_{2n}$ be the coefficients of the product polynomial $C(x)$, i.e., let

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

- We now have:

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

Fast multiplication of polynomials - ctd.

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

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

Fast multiplication of polynomials - ctd.

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

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

- i.e., we can obtain $C_0, C_1, \ldots, C_{2n}$ as

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

Fast multiplication of polynomials - ctd.

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

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

- i.e., we can obtain $C_0, C_1, \ldots, C_{2n}$ as

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

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

Fast multiplication of polynomials - ctd.

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

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

- i.e., we can obtain $C_0, C_1, \ldots, C_{2n}$ as

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

- But the inverse matrix also involves only constants depending on n only;
- Thus the coefficients C_i can be obtained in linear time.

Fast multiplication of polynomials - ctd.

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

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

- i.e., we can obtain $C_0, C_1, \ldots, C_{2n}$ as

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

- But the inverse matrix also involves only constants depending on n only;
- Thus the coefficients C_i can be obtained in linear time.
- So here is the algorithm we have just described:

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

if $|A| = |B| = 1$ **then return** AB

else

obtain $n+1$ slices $A_0, A_1, \ldots, A_n$ 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 $P_B(m)$;

$P_C(m) \leftarrow \textsc{Mult}(n, P_A(m) P_B(m))$

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 $P_C(2^k) = A \cdot B$

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 at most

$$\begin{aligned} \log_2\left((n+1)2^k n^n\right) &= k + \log_2\left((n+1) \cdot n^n\right) \\ &= k + \log_2(n+1) + n \log_2 n \end{aligned}$$

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, $-n \leq m \leq n$:

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

- 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 at most

$$\begin{aligned}\log_2\left((n+1)2^k n^n\right) &= k + \log_2\left((n+1) \cdot n^n\right) \\ &= k + \log_2(n+1) + n \log_2 n\end{aligned}$$

- But $s = \log_2(n+1) + n \log_2 n$ 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 at most

$$\begin{aligned} \log_2\left((n+1)2^k n^n\right) &= k + \log_2\left((n+1) \cdot n^n\right) \\ &= k + \log_2(n+1) + n \log_2 n \end{aligned}$$

- But $s = \log_2(n+1) + n \log_2 n$ 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

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \underbrace{\left(\sum_{i+m=j} A_i B_m \right)}_{C_j} x^j = \sum_{j=0}^{2n} C_j \, x^j, \quad \text{where} \quad C_j = \sum_{i+m=j} A_i B_m$$

Generalizing Karatsuba's algorithm

- Recall that

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \underbrace{\left(\sum_{i+m=j} A_i B_m \right)}_{C_j} x^j = \sum_{j=0}^{2n} C_j \, x^j, \quad \text{where} \quad C_j = \sum_{i+m=j} A_i B_m$$

- How big are C_j? Since $A_i, B_m < 2^k$, we have $A_i B_m < 2^{2k}$ and

$$C_j = \sum_{i+k=j} A_i B_m \leq (n+1) 2^{2k}$$

Generalizing Karatsuba's algorithm

- Recall that

$$P_A(x) \cdot P_B(x) = \sum_{j=0}^{2n} \underbrace{\left(\sum_{i+m=j} A_i B_m \right)}_{C_j} x^j = \sum_{j=0}^{2n} C_j \, x^j, \quad \text{where} \quad C_j = \sum_{i+m=j} A_i B_m$$

- How big are C_j? Since $A_i, B_m < 2^k$, we have $A_i B_m < 2^{2k}$ and

$$C_j = \sum_{i+k=j} A_i B_m \leq (n+1) 2^{2k}$$

- 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?

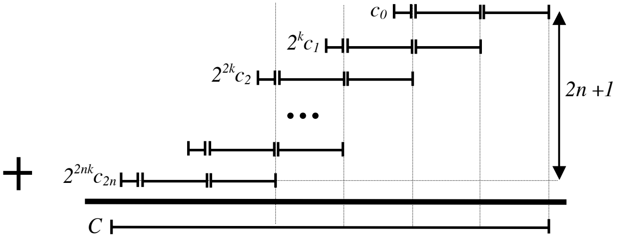
- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$
- C is obtained as a sum of binary shifts of $2n+1$ numbers C_j;

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 at most $2k + \lceil \log_2(n+1) \rceil$ bits;

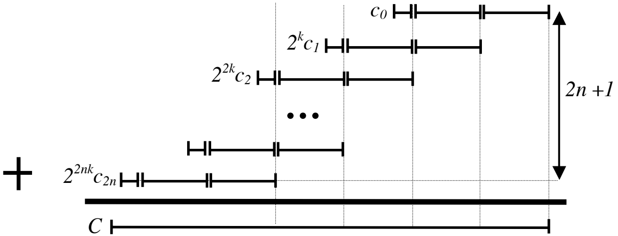
How fast is our algorithm?

- 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 at most $2k + \lceil \log_2(n+1) \rceil$ bits;



How fast is our algorithm?

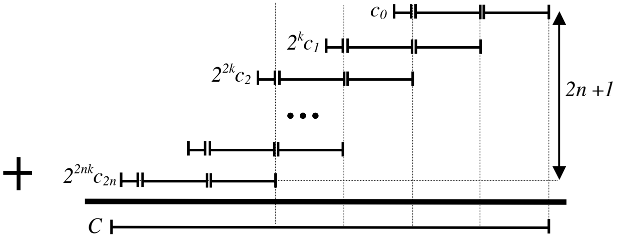
- We compute $C = A\,B$ as $C = P_C(2^k) = \sum_{j=0}^{2n} C_j 2^{kj}$
- C is obtained as a sum of binary shifts of $2n+1$ numbers C_j;
- each C_j has at most $2k + \lceil \log_2(n+1) \rceil$ bits;



- 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 at most $2k + \lceil \log_2(n+1) \rceil$ 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

$$T(N) = \underbrace{(2n+1)}_{a}\, T\Big(\underbrace{\frac{N}{n+1}}_{b} + s\Big) + d\,N$$

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

$$T(N) = \underbrace{(2n+1)}_{a}\, T\Big(\frac{N}{\underbrace{n+1}_{b}} + s\Big) + d\,N$$

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

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

$$T(N) = \underbrace{(2n+1)}_{a}\,T\Big(\frac{N}{\underbrace{n+1}_{b}} + s\Big) + d\,N$$

- Since s is constant, its impact can be neglected.
- Since $\log_b a = \log_{n+1}(2n+1) > 1$, we can choose a small ε such that also $\log_b a - \varepsilon > 1$.

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

$$T(N) = \underbrace{(2n+1)}_{a}\, T\Big(\frac{N}{\underbrace{n+1}_{b}} + s\Big) + d\,N$$

- Since s is constant, its impact can be neglected.
- Since $\log_b a = \log_{n+1}(2n+1) > 1$, we can choose a small ε such that also $\log_b a - \varepsilon > 1$.
- Consequently, for such an ε we would have $\quad f(N) = d\,N = O\left(N^{\log_b a - \varepsilon}\right)$.

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

$$T(N) = \underbrace{(2n+1)}_{a}\, T\Big(\frac{N}{\underbrace{n+1}_{b}} + s\Big) + d\,N$$

- Since s is constant, its impact can be neglected.
- Since $\log_b a = \log_{n+1}(2n+1) > 1$, we can choose a small ε such that also $\log_b a - \varepsilon > 1$.
- Consequently, for such an ε we would have $f(N) = d\,N = O\left(N^{\log_b a - \varepsilon}\right)$.
- Thus, with $a = 2n+1$ and $b = n+1$ the first case of the Master Theorem applies;

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

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

- Since s is constant, its impact can be neglected.
- Since $\log_b a = \log_{n+1}(2n+1) > 1$, we can choose a small ε such that also $\log_b a - \varepsilon > 1$.
- Consequently, for such an ε we would have $f(N) = d\,N = O\left(N^{\log_b a - \varepsilon}\right)$.
- Thus, with $a = 2n+1$ and $b = n+1$ the first case of the Master Theorem applies;
- so we get:

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

- Note that

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

- 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 the input numbers into $2^{10} = 1024$ pieces!!

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ both of degree n at values up to n.

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ both of degree n at values up to n.

- However, $n = 2^{10} - 1$, so evaluating $P_A(n) = A_n n^n + \ldots + A_0$ involves multiplication of A_n with $n^n = (2^{10} - 1)^{2^{10}-1} \approx 1.27 \times 10^{3079}$.

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ both of degree n at values up to n.

- However, $n = 2^{10} - 1$, so evaluating $P_A(n) = A_n n^n + \ldots + A_0$ involves multiplication of A_n with $n^n = (2^{10} - 1)^{2^{10}-1} \approx 1.27 \times 10^{3079}$.

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -n \ldots n$ can **theoretically** all be done in linear time, $T(n) = c\,n$, the constant c is absolutely **humongous**.

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ both of degree n at values up to n.
- However, $n = 2^{10} - 1$, so evaluating $P_A(n) = A_n n^n + \ldots + A_0$ involves multiplication of A_n with $n^n = (2^{10} - 1)^{2^{10}-1} \approx 1.27 \times 10^{3079}$.
- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -n \ldots n$ can **theoretically** all be done in linear time, $T(n) = c\,n$, the constant c is absolutely **humongous**.
- Consequently, slicing the input numbers in 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!

- We would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ both of degree n at values up to n.
- However, $n = 2^{10} - 1$, so evaluating $P_A(n) = A_n n^n + \ldots + A_0$ involves multiplication of A_n with $n^n = (2^{10} - 1)^{2^{10}-1} \approx 1.27 \times 10^{3079}$.
- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -n \ldots n$ can **theoretically** all be done in linear time, $T(n) = c\,n$, the constant c is absolutely **humongous**.
- Consequently, slicing the input numbers in 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!
- The moral is: **In practice, asymptotic estimates are useless if the size of the constants hidden by the O-notation are not estimated and found to be reasonably small!!!**

- **Crucial question:** Are there numbers $x_0, x_1, \ldots, x_n$ such that the size of x_i^n does not grow uncontrollably?

- **Crucial question:** 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 $|z_i| = 1$!

- **Crucial question:** 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 $|z_i| = 1$!
- This motivates us to consider values of polynomials at inputs which are equally spaced complex numbers all lying on the unit circle.

- **Crucial question:** 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 $|z_i| = 1$!
- 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.

- **Crucial question:** 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 $|z_i| = 1$!
- 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.
- We will present a very fast algorithm for computing these values, called **the Fast Fourier Transform**, abbreviated as **FFT**.

- **Crucial question:** 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 $|z_i| = 1$!
- 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.
- 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**.

- **Crucial question:** 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 $|z_i| = 1$!
- 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.
- 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.

- **Crucial question:** 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 $|z_i| = 1$!
- 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.
- 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.
- After we study the FFT we will have a guest lecture by a Dolby engineer to demonstrate to you some cool applications of FFT.