Algorithms: COMP3121/9101

Aleks Ignjatović, ignjat@cse.unsw.edu.au
office: 504 (CSE building K 17)
Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
University of New South Wales Sydney

3. 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
```

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

Basics revisited: how do we multiply two numbers?

- The primary school algorithm:

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

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

The Karatsuba trick

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

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

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

- AB can now be calculated as follows:

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

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

The Karatsuba trick

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

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

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

- AB can now be calculated as follows:

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

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

The Karatsuba trick

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

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

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

- AB can now be calculated as follows:

$$AB = A_1B_1 2^n + (A_1B_0 + A_0B_1)2^{\frac{n}{2}} + A_0B_0$$

$$= A_1B_1 2^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$$

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

The Karatsuba trick

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

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

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

- AB can now be calculated as follows:

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

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

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

The Karatsuba trick

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

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

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

- AB can now be calculated as follows:

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

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

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

$AB = A_1B_12^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)2^{\frac{n}{2}} + A_0B_0$

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 algorithm 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 $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < 0.5$$

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

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

without going through the messy calculations!

The Karatsuba trick

- How many steps does this algorithm 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 $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < 0.5$$

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

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

without going through the messy calculations!

The Karatsuba trick

- How many steps does this algorithm 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 $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3-\varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < 0.5$$

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

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

without going through the messy calculations!

The Karatsuba trick

- How many steps does this algorithm 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 $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < 0.5$$

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

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

without going through the messy calculations!

The Karatsuba trick

- How many steps does this algorithm 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

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < 0.5$$

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

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

without going through the messy calculations!

The Karatsuba trick

- How many steps does this algorithm 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 $1.5 < \log_2 3 < 1.6$ we have

$$f(n) = c\,n = O(n^{\log_2 3 - \varepsilon}) \quad \text{for any} \quad 0 < \varepsilon < 0.5$$

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

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

without going through the messy calculations!

Generalizing Karatsuba's algorithm

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

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

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,

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

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,

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

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,

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

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,

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

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.,

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

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

- 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

$$AB = \underbrace{A_2B_2}_{C_4}\, 2^{4k} + (\underbrace{A_2B_1 + A_1B_2}_{C_3})2^{3k} + (\underbrace{A_2B_0 + A_1B_1 + A_0B_2}_{C_2})2^{2k} +$$
$$+ (\underbrace{A_1B_0 + A_0B_1}_{C_1})2^{k} + \underbrace{A_0B_0}_{C_0}$$

- 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

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

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

The Karatsuba trick

$$AB = \underbrace{A_2B_2}_{C_4}\,2^{4k} + (\underbrace{A_2B_1 + A_1B_2}_{C_3})2^{3k} + (\underbrace{A_2B_0 + A_1B_1 + A_0B_2}_{C_2})2^{2k} +$$
$$+ (\underbrace{A_1B_0 + A_0B_1}_{C_1})2^{k} + \underbrace{A_0B_0}_{C_0}$$

- 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

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

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

The Karatsuba trick

$$AB = \underbrace{A_2B_2}_{C_4}\,2^{4k} + (\underbrace{A_2B_1 + A_1B_2}_{C_3})2^{3k} + (\underbrace{A_2B_0 + A_1B_1 + A_0B_2}_{C_2})2^{2k} +$$
$$+ (\underbrace{A_1B_0 + A_0B_1}_{C_1})2^{k} + \underbrace{A_0B_0}_{C_0}$$

- 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

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

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

The Karatsuba trick

$$AB = \underbrace{A_2B_2}_{C_4}\,2^{4k} + (\underbrace{A_2B_1 + A_1B_2}_{C_3})2^{3k} + (\underbrace{A_2B_0 + A_1B_1 + A_0B_2}_{C_2})2^{2k} +$$
$$+ (\underbrace{A_1B_0 + A_0B_1}_{C_1})2^{k} + \underbrace{A_0B_0}_{C_0}$$

- 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

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

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

The Karatsuba trick

$$AB = \underbrace{A_2B_2}_{C_4}\, 2^{4k} + (\underbrace{A_2B_1 + A_1B_2}_{C_3})2^{3k} + (\underbrace{A_2B_0 + A_1B_1 + A_0B_2}_{C_2})2^{2k} +$$
$$+ (\underbrace{A_1B_0 + A_0B_1}_{C_1})2^{k} + \underbrace{A_0B_0}_{C_0}$$

- 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

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

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

The Karatsuba trick

$$AB = \underbrace{A_2B_2}_{C_4}\, 2^{4k} + (\underbrace{A_2B_1 + A_1B_2}_{C_3})2^{3k} + (\underbrace{A_2B_0 + A_1B_1 + A_0B_2}_{C_2})2^{2k} +$$
$$+ (\underbrace{A_1B_0 + A_0B_1}_{C_1})2^{k} + \underbrace{A_0B_0}_{C_0}$$

- 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

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

- 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!
- Let

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

- We form the naturally corresponding polynomials:

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

- 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

- 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 the naturally corresponding polynomials:

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

- 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

- 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 the naturally corresponding polynomials:

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

- 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

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

- 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

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

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

- 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

- 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

- 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

- 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

- 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

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

- 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

- 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

- 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

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

- Simplifying the left side 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

- 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

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

- Simplifying the left side 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 produces (as an exercise solve this system by hand, using the Gaussian elimination)

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

- 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:

The Karatsuba trick: slicing into 3 pieces

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 produces (as an exercise solve this system by hand, using the Gaussian elimination)

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

- 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.
- 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:

The Karatsuba trick: slicing into 3 pieces

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 produces (as an exercise solve this system by hand, using the Gaussian elimination)

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

- 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.
- 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:

The Karatsuba trick: slicing into 3 pieces

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 produces (as an exercise solve this system by hand, using the Gaussian elimination)

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

- 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:

The Karatsuba trick: slicing into 3 pieces

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 produces (as an exercise solve this system by hand, using the Gaussian elimination)

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

- 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:

The Karatsuba trick: slicing into 3 pieces

- Solving this system of linear equations for C_0, C_1, C_2, C_3, C_4 produces (as an exercise solve this system by hand, using the Gaussian elimination)

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

- 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 MULT(A, B)

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

form polynomials $P_A(x) = A_2x^2 + A_1x + A_0; \quad P_B(x) = B_2x^2 + B_1x + B_0;$

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

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

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

form $P_C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x + C_0$; 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?
- 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 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

- 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 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

- 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

- 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 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

- 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 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

- 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

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

- 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!
- One can show that in fact slicing numbers in larger number of slices produces asymptotically faster algorithm, but its constants appearing in the asymptotic estimates explode (they involve p^p where $p+1$ is the number of slices).
- This shows that asymptotic estimates are useful only if the sizes of the constants involved are small!

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!
- One can show that in fact slicing numbers in larger number of slices produces asymptotically faster algorithm, but its constants appearing in the asymptotic estimates explode (they involve p^p where $p+1$ is the number of slices).
- This shows that asymptotic estimates are useful only if the sizes of the constants involved are small!

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!
- One can show that in fact slicing numbers in larger number of slices produces asymptotically faster algorithm, but its constants appearing in the asymptotic estimates explode (they involve p^p where $p + 1$ is the number of slices).
- This shows that asymptotic estimates are useful only if the sizes of the constants involved are small!

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!
- One can show that in fact slicing numbers in larger number of slices produces asymptotically faster algorithm, but its constants appearing in the asymptotic estimates explode (they involve p^p where $p+1$ is the number of slices).
- This shows that asymptotic estimates are useful only if the sizes of the constants involved are small!

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!
- One can show that in fact slicing numbers in larger number of slices produces asymptotically faster algorithm, but its constants appearing in the asymptotic estimates explode (they involve p^p where $p+1$ is the number of slices).

- This shows that asymptotic estimates are useful only if the sizes of the constants involved are small!

The Karatsuba trick: slicing into 3 pieces

- Recall that the original Karatsuba algorithm runs in time

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

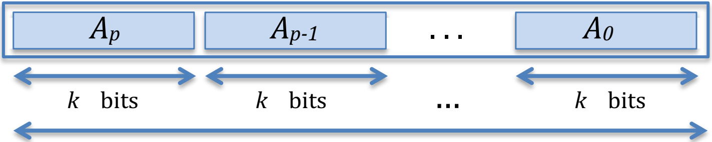
- Thus, we got a significantly faster algorithm.
- Then why not slice numbers A and B into even larger number of slices? Maybe we can get even faster algorithm?
- The answer is, in a sense, BOTH yes and no!
- One can show that in fact slicing numbers in larger number of slices produces asymptotically faster algorithm, but its constants appearing in the asymptotic estimates explode (they involve p^p where $p+1$ is the number of slices).
- This shows that asymptotic estimates are useful only if the sizes of the constants involved are small!

Generalizing Karatsuba's algorithm (not examinable)

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

- For simplicity, let us assume A and B have exactly $(p+1)k$ bits (otherwise one of the slices will have to be shorter);
- Note: p is a fixed (smallish) number, a fixed parameter of our design – $p+1$ is the number of slices we are going to make, but k depends on the input values A and B and can be arbitrarily large!
- Slice A, B into $p+1$ pieces each:

$$\begin{aligned} A &= A_p 2^{kp} + A_{p-1} 2^{k(p-1)} + \cdots + A_0 \\ B &= B_p 2^{kp} + B_{p-1} 2^{k(p-1)} + \cdots + B_0 \end{aligned}$$



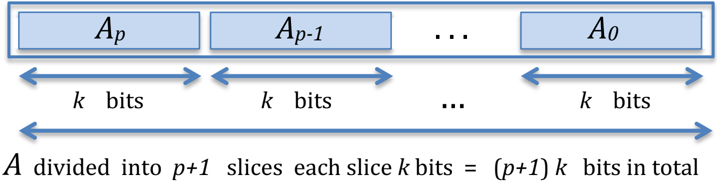
A divided into $p+1$ slices each slice k bits = $(p+1)\,k$ bits in total

Generalizing Karatsuba's algorithm (not examinable)

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

- For simplicity, let us assume A and B have exactly $(p+1)k$ bits (otherwise one of the slices will have to be shorter);
- Note: p is a fixed (smallish) number, a fixed parameter of our design – $p+1$ is the number of slices we are going to make, but k depends on the input values A and B and can be arbitrarily large!
- Slice A, B into $p+1$ pieces each:

$$\begin{aligned} A &= A_p 2^{kp} + A_{p-1} 2^{k(p-1)} + \cdots + A_0 \\ B &= B_p 2^{kp} + B_{p-1} 2^{k(p-1)} + \cdots + B_0 \end{aligned}$$

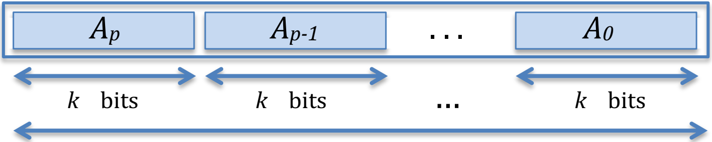


Generalizing Karatsuba's algorithm (not examinable)

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

- For simplicity, let us assume A and B have exactly $(p+1)k$ bits (otherwise one of the slices will have to be shorter);
- Note: p is a fixed (smallish) number, a fixed parameter of our design – $p+1$ is the number of slices we are going to make, but k depends on the input values A and B and can be arbitrarily large!
- Slice A, B into $p+1$ pieces each:

$$\begin{aligned} A &= A_p 2^{kp} + A_{p-1} 2^{k(p-1)} + \cdots + A_0 \\ B &= B_p 2^{kp} + B_{p-1} 2^{k(p-1)} + \cdots + B_0 \end{aligned}$$



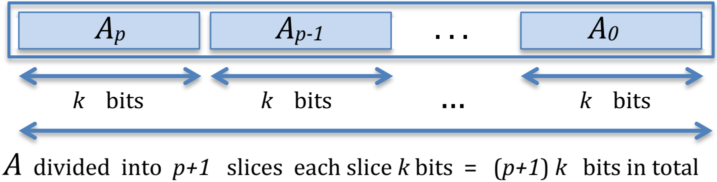
A divided into $p+1$ slices each slice k bits = $(p+1)\,k$ bits in total

Generalizing Karatsuba's algorithm (not examinable)

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

- For simplicity, let us assume A and B have exactly $(p+1)k$ bits (otherwise one of the slices will have to be shorter);
- Note: p is a fixed (smallish) number, a fixed parameter of our design – $p+1$ is the number of slices we are going to make, but k depends on the input values A and B and can be arbitrarily large!
- Slice A, B into $p+1$ pieces each:

$$\begin{aligned} A &= A_p 2^{kp} + A_{p-1} 2^{k(p-1)} + \cdots + A_0 \\ B &= B_p 2^{kp} + B_{p-1} 2^{k(p-1)} + \cdots + B_0 \end{aligned}$$

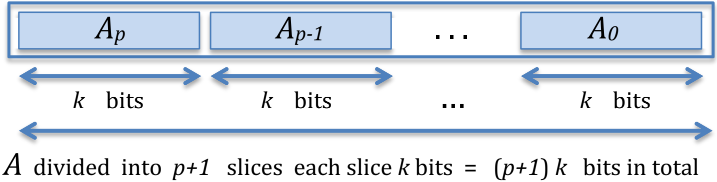


Generalizing Karatsuba's algorithm (not examinable)

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

- For simplicity, let us assume A and B have exactly $(p+1)k$ bits (otherwise one of the slices will have to be shorter);
- Note: p is a fixed (smallish) number, a fixed parameter of our design – $p+1$ is the number of slices we are going to make, but k depends on the input values A and B and can be arbitrarily large!
- Slice A, B into $p+1$ pieces each:

$$\begin{aligned} A &= A_p 2^{kp} + A_{p-1}2^{k(p-1)} + \cdots + A_0 \\ B &= B_p 2^{kp} + B_{p-1}2^{k(p-1)} + \cdots + B_0 \end{aligned}$$



Generalizing Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_p x^p + A_{p-1}x^{p-1} + \cdots + A_0 \\ P_B(x) &= B_p x^p + B_{p-1}x^{p-1} + \cdots + B_0 \end{aligned}$$

- As before, we have:
$A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

- 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 $2p$:

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

Generalizing Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_p x^p + A_{p-1} x^{p-1} + \dots + A_0 \\ P_B(x) &= B_p x^p + B_{p-1} x^{p-1} + \dots + B_0 \end{aligned}$$

- As before, we have:
  $A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

- 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 $2p$:

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

Generalizing Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_p x^p + A_{p-1}x^{p-1} + \dots + A_0 \\ P_B(x) &= B_p x^p + B_{p-1}x^{p-1} + \dots + B_0 \end{aligned}$$

- As before, we have:
$A = P_A(2^k);\ \ B = P_B(2^k);\ \ AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

- 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 $2p$:

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

Generalizing Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_p x^p + A_{p-1}x^{p-1} + \cdots + A_0 \\ P_B(x) &= B_p x^p + B_{p-1}x^{p-1} + \cdots + B_0 \end{aligned}$$

- As before, we have:
  $A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

- 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 $2p$:

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

Generalizing Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_p x^p + A_{p-1}x^{p-1} + \cdots + A_0 \\ P_B(x) &= B_p x^p + B_{p-1}x^{p-1} + \cdots + B_0 \end{aligned}$$

- As before, we have:
$A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

- 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 $2p$:

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

Generalizing Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_p x^p + A_{p-1}x^{p-1} + \cdots + A_0 \\ P_B(x) &= B_p x^p + B_{p-1}x^{p-1} + \cdots + B_0 \end{aligned}$$

- As before, we have:
  $A = P_A(2^k)$; $B = P_B(2^k)$; $AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

- 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 $2p$:

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

Generalizing Karatsuba's algorithm

- We form the naturally corresponding polynomials:

$$\begin{aligned} P_A(x) &= A_p x^p + A_{p-1}x^{p-1} + \cdots + A_0 \\ P_B(x) &= B_p x^p + B_{p-1}x^{p-1} + \cdots + B_0 \end{aligned}$$

- As before, we have:
  $A = P_A(2^k);\;\; B = P_B(2^k);\;\; AB = P_A(2^k)P_B(2^k) = (P_A(x) \cdot P_B(x))\,|_{x=2^k}$

- 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 $2p$:

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

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

$$P_A(x) = A_px^p + A_{p-1}x^{p-1} + \cdots + A_0$$
$$P_B(x) = B_px^p + B_{p-1}x^{p-1} + \cdots + B_0$$

we have

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

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

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

$$
\begin{aligned}
P_A(x) &= A_px^p + A_{p-1}x^{p-1} + \cdots + A_0 \\
P_B(x) &= B_px^p + B_{p-1}x^{p-1} + \cdots + B_0
\end{aligned}
$$

we have

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

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

Generalizing Karatsuba's algorithm

- Example:

$$(A_3x^3 + A_2x^2 + A_1x + A_0)(B_3x^3 + B_2x^2 + B_1x + B_0) =$$
$$A_3B_3x^6 + (A_2B_3 + A_3B_2)x^5 + (A_1B_3 + A_2B_2 + A_3B_1)x^4$$
$$+(A_0B_3 + A_1B_2 + A_2B_1 + A_3B_0)x^3 + (A_0B_2 + A_1B_1 + A_2B_0)x^2$$
$$+(A_0B_1 + A_1B_0)x + A_0B_0$$

- In general: for

$$P_A(x) = A_px^p + A_{p-1}x^{p-1} + \cdots + A_0$$
$$P_B(x) = B_px^p + B_{p-1}x^{p-1} + \cdots + B_0$$

we have

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

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

Generalizing Karatsuba's algorithm

- Example:

$$(A_3x^3 + A_2x^2 + A_1x + A_0)(B_3x^3 + B_2x^2 + B_1x + B_0) =$$
$$A_3B_3x^6 + (A_2B_3 + A_3B_2)x^5 + (A_1B_3 + A_2B_2 + A_3B_1)x^4$$
$$+(A_0B_3 + A_1B_2 + A_2B_1 + A_3B_0)x^3 + (A_0B_2 + A_1B_1 + A_2B_0)x^2$$
$$+(A_0B_1 + A_1B_0)x + A_0B_0$$

- In general: for

$$P_A(x) = A_px^p + A_{p-1}x^{p-1} + \cdots + A_0$$
$$P_B(x) = B_px^p + B_{p-1}x^{p-1} + \cdots + B_0$$

we have

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

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

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

$$
\begin{aligned}
P_A(x) &= A_p x^p + A_{p-1}x^{p-1} + \cdots + A_0 \\
P_B(x) &= B_p x^p + B_{p-1}x^{p-1} + \cdots + B_0
\end{aligned}
$$

we have

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

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

A VERY IMPORTANT DIGRESSION:

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

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

and if you multiply these two polynomials to obtain their product

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

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

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

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:

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

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

and if you multiply these two polynomials to obtain their product

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

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

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

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:

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

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

and if you multiply these two polynomials to obtain their product

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

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

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

is **extremely important** and is called the **LINEAR CONVOLUTION** of sequences $\vec{A}$ and $\vec{B}$ and is denoted by $\vec{C} = \vec{A} \star \vec{B}$.

AN 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).

AN 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).

AN 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).

AN 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).

AN 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).

AN 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).

AN 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).

AN 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 p is uniquely determined by its values at any $p+1$ distinct input values $x_0, x_1, \ldots, x_p$:

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

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

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix}. \tag{1}$$

- 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

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

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

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

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix}. \tag{1}$$

- 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

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

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1)), \dots, (x_p, P_A(x_p))\}$$

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

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^p \\ 1 & x_1 & x_1^2 & \dots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \dots & x_p^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix}. \tag{1}$$

- 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

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

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

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

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix}. \tag{1}$$

- 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 $P_A(x_0), P_A(x_1), \ldots, P_A(x_p)$ the coefficients $A_0, A_1, \ldots, A_p$ of the polynomial $P_A(x)$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix} \tag{2}$$

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

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

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are all distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_p)$ the coefficients $A_0, A_1, \ldots, A_p$ of the polynomial $P_A(x)$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix} \tag{2}$$

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

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

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are all distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_p)$ the coefficients $A_0, A_1, \ldots, A_p$ of the polynomial $P_A(x)$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix} \tag{2}$$

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

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

Coefficient vs value representation of polynomials - ctd.

- Thus, if all x_i are all distinct, given any values $P_A(x_0), P_A(x_1), \ldots, P_A(x_p)$ the coefficients $A_0, A_1, \ldots, A_p$ of the polynomial $P_A(x)$ are uniquely determined:

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix} \tag{2}$$

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

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

Coefficient vs value representation of polynomials- ctd.

- If we fix the inputs $x_0, x_1, \ldots, x_p$ 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**:

$$\begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix};$$

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix}.$$

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

Coefficient vs value representation of polynomials- ctd.

- If we fix the inputs $x_0, x_1, \ldots, x_p$ 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**:

$$\begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix};$$

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix}.$$

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

Coefficient vs value representation of polynomials- ctd.

- If we fix the inputs $x_0, x_1, \ldots, x_p$ 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**:

$$\begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix};$$

$$\begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^p \\ 1 & x_1 & x_1^2 & \ldots & x_1^p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_p & x_p^2 & \ldots & x_p^p \end{pmatrix}^{-1} \begin{pmatrix} P_A(x_0) \\ P_A(x_1) \\ \vdots \\ P_A(x_p) \end{pmatrix}.$$

- Thus, for fixed input values $x_0, \ldots, x_p$ 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 p,

$$P_A(x) = A_p x^p + \ldots + A_0; \qquad P_B(x) = B_p x^p + \ldots + B_0$$

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

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

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

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

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

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

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

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

Our strategy to multiply polynomials fast:

1. Given two polynomials of degree at most p,

$$P_A(x) = A_p x^p + \ldots + A_0; \qquad P_B(x) = B_p x^p + \ldots + B_0$$

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

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

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

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

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

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

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

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

Our strategy to multiply polynomials fast:

1. Given two polynomials of degree at most p,

$$P_A(x) = A_p x^p + \ldots + A_0; \qquad P_B(x) = B_p x^p + \ldots + B_0$$

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

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

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

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

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

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

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

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

Our strategy to multiply polynomials fast:

1. Given two polynomials of degree at most p,

$$P_A(x) = A_p x^p + \ldots + A_0; \qquad P_B(x) = B_p x^p + \ldots + B_0$$

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

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

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

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

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

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

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

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

Our strategy to multiply polynomials fast:

1. Given two polynomials of degree at most p,

$$P_A(x) = A_p x^p + \ldots + A_0; \qquad P_B(x) = B_p x^p + \ldots + B_0$$

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

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

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

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

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

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

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

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

Our strategy to multiply polynomials fast:

1. Given two polynomials of degree at most p,

$$P_A(x) = A_p x^p + \ldots + A_0; \qquad P_B(x) = B_p x^p + \ldots + B_0$$

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

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

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

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

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

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

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

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-p \leq m \leq p$.
- Remember that $p+1$ 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

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

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

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-p \leq m \leq p$.
- Remember that $p+1$ 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

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

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

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-p \leq m \leq p$.
- Remember that $p+1$ 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

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

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

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-p \leq m \leq p$.
- Remember that $p+1$ 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

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

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

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-p \leq m \leq p$.
- Remember that $p+1$ 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

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

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

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-p \leq m \leq p$.
- Remember that $p+1$ 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

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

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

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

Fast multiplication of polynomials - continued

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

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

- So we find the values $P_A(m)$ and $P_B(m)$ for all m such that $-p \leq m \leq p$.
- Remember that $p + 1$ 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

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

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

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

Fast multiplication of polynomials - ctd.

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

$$P_A(-p)P_B(-p),\ \ldots\ ,\ P_A(-1)P_B(-1),\quad P_A(0)P_B(0),\quad P_A(1)P_B(1),\ \ldots\ ,\ P_A(p)P_B(p)$$

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

$$P_C(-p) = P_A(-p)P_B(-p),\ \ldots\ ,\ P_C(0) = P_A(0)P_B(0),\ \ldots\ ,\ P_C(p) = P_A(p)P_B(p)$$

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

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

- We now have:

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

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

- We now have:

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

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

- We now have:

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

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

- We now have:

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

Fast multiplication of polynomials - ctd.

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

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

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

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

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

function Mult(A, B)

if $|A| = |B| < p + 1$ **then return** AB

else

obtain $p + 1$ slices $A_0, A_1, \ldots, A_p$ and $B_0, B_1, \ldots, B_p$ such that

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

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

form polynomials

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

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

for $m = -p$ to $m = p$ **do**

compute $P_A(m)$ and $P_B(m)$;

$P_C(m) \leftarrow$ Mult$(P_A(m) P_B(m))$

end for

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

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

form $P_C(x) = C_{2p} x^{2p} + \ldots + C_0$ and compute $P_C(2^k)$

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

end if

end function

How fast is our algorithm?

- it is easy to see that the values of the two polynomials we are multiplying have at most $k+s$ bits where s is a constant which depends on p but does NOT depend on k:

$$P_A(m) = A_p m^p + A_{p-1} m^{p-1} + \cdots + A_0 : \quad -p \leq m \leq p.$$

This is because each A_i is smaller than 2^k because each A_k has k bits; thus

$$|P_A(m)| < p^p(p+1) \times 2^k \quad \Rightarrow \quad \log_2 |P_A(m)| < \log_2(p^p(p+1)) + k = s + k$$

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

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

- Let $n = (p+1)k$. Then

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

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

How fast is our algorithm?

- it is easy to see that the values of the two polynomials we are multiplying have at most $k+s$ bits where s is a constant which depends on p but does NOT depend on k:

$$P_A(m) = A_p m^p + A_{p-1} m^{p-1} + \cdots + A_0 : \quad -p \leq m \leq p.$$

This is because each A_i is smaller than 2^k because each A_k has k bits; thus

$$|P_A(m)| < p^p(p+1) \times 2^k \quad \Rightarrow \quad \log_2 |P_A(m)| < \log_2(p^p(p+1)) + k = s + k$$

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

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

- Let $n = (p+1)k$. Then

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

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

How fast is our algorithm?

- it is easy to see that the values of the two polynomials we are multiplying have at most $k+s$ bits where s is a constant which depends on p but does NOT depend on k:

$$P_A(m) = A_p m^p + A_{p-1} m^{p-1} + \cdots + A_0 : \quad -p \leq m \leq p.$$

  This is because each A_i is smaller than 2^k because each A_k has k bits; thus

$$|P_A(m)| < p^p(p+1) \times 2^k \quad \Rightarrow \quad \log_2 |P_A(m)| < \log_2(p^p(p+1)) + k = s + k$$

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

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

- Let $n = (p+1)k$. Then

$$T(n) = \underbrace{(2p+1)}_{a} T\Big(\underbrace{\frac{n}{p+1}}_{b} + s\Big) + \frac{c}{p+1} n$$

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

How fast is our algorithm?

- it is easy to see that the values of the two polynomials we are multiplying have at most $k+s$ bits where s is a constant which depends on p but does NOT depend on k:

$$P_A(m) = A_p m^p + A_{p-1} m^{p-1} + \cdots + A_0 : \quad -p \leq m \leq p.$$

This is because each A_i is smaller than 2^k because each A_k has k bits; thus

$$|P_A(m)| < p^p(p+1) \times 2^k \quad \Rightarrow \quad \log_2 |P_A(m)| < \log_2(p^p(p+1)) + k = s + k$$

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

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

- Let $n = (p+1)k$. Then

$$T(n) = \underbrace{(2p+1)}_{a} T\Big(\underbrace{\frac{n}{p+1}}_{b} + s\Big) + \frac{c}{p+1} n$$

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

How fast is our algorithm?

- it is easy to see that the values of the two polynomials we are multiplying have at most $k+s$ bits where s is a constant which depends on p but does NOT depend on k:

$$P_A(m) = A_p m^p + A_{p-1}m^{p-1} + \cdots + A_0 : \quad -p \leq m \leq p.$$

  This is because each A_i is smaller than 2^k because each A_k has k bits; thus

$$|P_A(m)| < p^p(p+1) \times 2^k \quad \Rightarrow \quad \log_2 |P_A(m)| < \log_2(p^p(p+1)) + k = s + k$$

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

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

- Let $n = (p+1)k$. Then

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

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

How fast is our algorithm?

- it is easy to see that the values of the two polynomials we are multiplying have at most $k+s$ bits where s is a constant which depends on p but does NOT depend on k:

$$P_A(m) = A_p m^p + A_{p-1} m^{p-1} + \cdots + A_0 : \quad -p \leq m \leq p.$$

This is because each A_i is smaller than 2^k because each A_k has k bits; thus

$$|P_A(m)| < p^p(p+1) \times 2^k \quad \Rightarrow \quad \log_2 |P_A(m)| < \log_2(p^p(p+1)) + k = s + k$$

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

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

- Let $n = (p+1)k$. Then

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

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

How fast is our algorithm?

- it is easy to see that the values of the two polynomials we are multiplying have at most $k + s$ bits where s is a constant which depends on p but does NOT depend on k:

$$P_A(m) = A_p m^p + A_{p-1} m^{p-1} + \cdots + A_0 : \quad -p \leq m \leq p.$$

This is because each A_i is smaller than 2^k because each A_k has k bits; thus

$$|P_A(m)| < p^p(p+1) \times 2^k \quad \Rightarrow \quad \log_2 |P_A(m)| < \log_2(p^p(p+1)) + k = s + k$$

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

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

- Let $n = (p+1)k$. Then

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

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

How fast is our algorithm?

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

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

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

How fast is our algorithm?

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

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

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

How fast is our algorithm?

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

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

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

How fast is our algorithm?

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

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

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

- Note that

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

- Thus, by choosing a sufficiently large p, we can get a run time arbitrarily close to linear time!
- How large does p have to be, in order to to get an algorithm which runs in time $n^{1.1}$?

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

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

- Note that

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

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

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

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

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

- Note that

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

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

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

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

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

- Note that

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

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

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

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

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

- Note that

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

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

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

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

- 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 p at values up to p.

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

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -p \ldots p$ can **theoretically** all be done in linear time, $T(p) = c\,p$, 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!!!**

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

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

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

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -p \ldots p$ can **theoretically** all be done in linear time, $T(p) = c\,p$, 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!!!**

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

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

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -p \ldots p$ can **theoretically** all be done in linear time, $T(p) = c\,p$, 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!!!**

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

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

- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -p \ldots p$ can **theoretically** all be done in linear time, $T(p) = c\,p$, 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 $x_0, x_1, \ldots, x_p$ such that the size of x_i^p does not grow uncontrollably?

- Answer: YES; they are the complex numbers z_i lying on the unit circle, i.e., such that $|z_i| = 1$!

- This leads to something called the Fast Fourier Transform, about which you can read in the supplementary notes on Moodle (not examinable material).

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

- Answer: YES; they are the complex numbers z_i lying on the unit circle, i.e., such that $|z_i| = 1$!

- This leads to something called the Fast Fourier Transform, about which you can read in the supplementary notes on Moodle (not examinable material).

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

- Answer: YES; they are the complex numbers z_i lying on the unit circle, i.e., such that $|z_i| = 1$!

- This leads to something called the Fast Fourier Transform, about which you can read in the supplementary notes on Moodle (not examinable material).

PUZZLE!

The warden meets with 23 new prisoners when they arrive. He tells them, “You may meet today and plan a strategy. But after today, you will be in isolated cells and will have no communication with one another. In the prison there is a switch room, which contains two light switches labeled A and B, each of which can be in either the on or the off position. I am not telling you their present positions. The switches are not connected to anything. After today, from time to time whenever I feel so inclined, I will select one prisoner at random and escort him to the switch room. This prisoner will select one of the two switches and reverse its position. He must move one, but only one of the switches. He can’t move both but he can’t move none either. Then he will be led back to his cell. No one else will enter the switch room until I lead the next prisoner there, and he’ll be instructed to do the same thing. I’m going to choose prisoners at random. I may choose the same guy three times in a row, or I may jump around and come back. But, given enough time, everyone would eventually visit the switch room many times. At any time anyone of you may declare to me: “We have all visited the switch room. If it is true, then you will all be set free. If it is false, and somebody has not yet visited the switch room, you will be fed to the alligators.”

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