COMP3121: Algorithms & Programming Techniques
Summary notes – Week 2

Gerald Huang

Updated: June 14, 2020

Contents

1 Lecture 3 – Recurrences

Content covered here can be found in the Lecture 3A and Lecture 3B recordings.

Date: June 14, 2020

1.1 Asymptotic notation

- **Big Oh notation**: $f(n) = O(g(n))$ is an abbreviation for

  There exist positive constants c and n_0 such that

  $$0 \le f(n) \le cg(n), \quad \text{for all } n \le n_0.$$

  - We say that $g(n)$ is an *asymptotic upper bound for $f(n)$*.
  - $f(n) = O(g(n))$ means that $f(n)$ does **not** grow substantially faster than $g(n)$.
  - Assume that $c > 1$.
- **Omega notation**: $f(n) = \Omega(g(n))$ is an abbreviation for

  There exist positive constants c and *n_0 such that*

  $$0 \le cg(n) \le f(n) \quad \text{for all } n \le n_0.$$

  - We say that $g(n)$ is an *asymptotic lower bound for $f(n)$*.
  - $f(n) = \Omega(g(n))$ means that $f(n)$ grows **at least** as fast as $g(n)$.
  - Assume that $c > 0$.
- **Theta notation**: $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$; that is $f(n)$ and $g(n)$ have the **same asymptotic growth rate**.

- **Examples**:
  - $5n = O(n^2)$ since $5n < n^2$ for $n > 5$. So pick $c = 1$ and $n_0 = 5$.
  - $2n^2 = O(n^2)$ since $2n^2 < cn^2$ for all $n > n_0$ where $c = 3$ and $n_0 = 1$.
  - $n^3 = \Omega(n^2)$ since $n^2 < n^3$ for all $n > 1$ where $c = 1$.
  - $n^3 = \Omega(10n^3 + n^2)$ since $c(10n^3 + n^2) < n^3$ for $c = 1/20$ and for all $n > 1$.

1.2 Recurrences

Let $a \ge 1$ be an integer and $b > 1$ be a real number. Assume that a divide and conquer algorithm

- reduces a problem size n to a many problems of smaller size n/b.
- the overhead cost of splitting up/combining the solutions for size n/b into a solution for size n is $f(n)$.

Then the time complexity of such algorithm satisfies

$$T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + f(n) \approx aT\left(\frac{n}{b}\right) + f(n).$$

Note we only need to find

1. the **growth rate** of the solution (its asymptotic behaviour).
2. the (approximate **sizes of constants** involved.

1.3 Master theorem

Master Theorem

Let $a \geq 1$ be an integer and $b > 1$ be real. Also let $f(n) > 0$ be a non decreasing function and $T(n)$ be a solution of the recurrence $T(n) = aT(n/b) + f(n)$.
Then

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ **and** for some $c < 1$ and some n_0,
$$af(n/b) \leq cf(n)$$
holds for all $n > n_0$, then $T(n) = \Theta(f(n))$.
4. Otherwise, the Master Theorem is **not** applicable.

- **Examples**:
  - Let $T(n) = 4T(n/2) + n$. Then $n^{\log_b a} = n^{\log_2 4} = n^2$. Thus $f(n) = n = O(n^{2-\epsilon})$ for any $\epsilon < 1$. Thus $T(n) = \Theta(n^2)$.
  - Let $T(n) = 2T(n/2) + cn$. Then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Thus, $f(n) = cn = \Theta(n) = \Theta(n^{\log_2 2})$. So $T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$.
  - Let $T(n) = 3T(n/4) + n$. Then $n^{\log_b a} = n^{\log_4 3} < n^{0.8}$. Thus $f(n_) = n = \Omega(n^{0.8+\epsilon})$ for any $\epsilon < 0.2$. Also $af(n/b) = 3f(n/4) = 3/4n < cn = cf(n)$ for $c = 0.8 < 1$.
  - Let $T(n) = 2T(n/2) + n \log_2 n$. Then $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Thus $f(n) = n \log_2 n = \Omega(n)$. However, $f(n) = n \log_2 n \neq \Omega(n^{1+\epsilon})$ no matter how small $\epsilon > 0$. Thus, the Master Theorem **does not apply**!

Proof. Since
$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{1}$$
implies (by applying it to n/b in place of n)
$$T\left(\frac{n}{b}\right) = aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right) \tag{2}$$
and (by applying (1) to n/b^2 in place of n)
$$T\left(\frac{n}{b^2}\right) = aT\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right) \tag{3}$$

and so on, we get

$$\overbrace{T(n) = a\underbrace{T\left(\frac{n}{b}\right)}_{(2)} + f(n)}^{(1)} = a\left(aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n)$$

$$= a^2\underbrace{T\left(\frac{n}{b^2}\right)}_{(3)} + af\left(\frac{n}{b}\right) + f(n) = a^2\underbrace{\left(aT\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right)\right)}_{(3)} + af\left(\frac{n}{b}\right) + f(n)$$

$$= a^3T\left(\frac{n}{b^3}\right) + a^2f\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) = \ldots$$

Continuing in this way $\log_b n - 1$ many times, we get

$$\begin{aligned}
T(n) &= a^3T\left(\frac{n}{b^3}\right) + a^2f\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \\
&= \ldots \\
&= a^{\lfloor \log_b n \rfloor}T\left(\frac{n}{b^{\lfloor \log_b n \rfloor}}\right) + a^{\lfloor \log_b n \rfloor - 1}f\left(\frac{n}{b^{\lfloor \log_b n \rfloor - 1}}\right) + \cdots + af\left(\frac{n}{b}\right) + f(n) \\
&\approx a^{\log_b n}T\left(\frac{n}{b^{\log_b n}}\right) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right).
\end{aligned}$$

Use the fact that $a^{\log_b n} = n^{\log_b a}$ to get

$$T(n) \approx n^{\log_b a}T(1) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right). \tag{4}$$

We now consider the different cases of the Master Theorem.

- **Case 1**: $f(m) = O(m^{\log_b a - \epsilon})$. Then

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i O\left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}.$$

  Since $n^{\log_b a - \epsilon}$ is independent of i, then we can move it outside of the sum. So we have

$$\begin{aligned}
\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) &= O\left(\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}\right) = O\left(n^{\log_b a - \epsilon}\sum_{i=0}^{\lfloor \log_b n \rfloor - 1}\left(\frac{a^i}{(b^i)^{\log_b a - \epsilon}}\right)\right) \\
&= O\left(n^{\log_b a - \epsilon}\sum_{i=0}^{\lfloor \log_b n \rfloor - 1}\left(\frac{a^i}{(b^{\log_b a - \epsilon})^i}\right)\right).
\end{aligned}$$

  Use the fact that $b^{\log_b a - \epsilon} = b^{\log_b a} \times b^{-\epsilon} = ab^{-\epsilon}$ to get

$$\begin{aligned}
\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) &= O\left(n^{\log_b a - \epsilon}\sum_{i=0}^{\lfloor \log_b n \rfloor - 1}\left(\frac{a}{a \times b^{-\epsilon}}\right)^i\right) \\
&= O\left(n^{\log_b a - \epsilon}\underbrace{\sum_{i=0}^{\lfloor \log_b n \rfloor - 1}\left(b^{\epsilon}\right)^i}_{\text{geometric series}}\right)
\end{aligned}$$

The sum becomes a geometric series with ratio $r = b^\epsilon$. Using the result

$$\sum_{i=0}^{m} a^i = \frac{a^{m+1} - 1}{a - 1},$$

the sum simplifies to $\displaystyle\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(b^\epsilon\right)^i = \frac{(b^\epsilon)^{\lfloor \log_b n \rfloor} - 1}{b^\epsilon - 1}$.

So we have

$$\begin{aligned}
\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) &= O\left(n^{\log_b a - \epsilon}\left[\frac{(b^\epsilon)^{\lfloor \log_b n \rfloor} - 1}{b^\epsilon - 1}\right]\right) \\
&= O\left(n^{\log_b a - \epsilon}\left[\frac{(b^{\lfloor \log_b n \rfloor})^\epsilon - 1}{b^\epsilon - 1}\right]\right) \\
&= O\left(n^{\log_b a - \epsilon}\left[\frac{n^\epsilon - 1}{b^\epsilon - 1}\right]\right) \\
&= O\left(\frac{n^{\log_b a} - n^{\log_b a - \epsilon}}{b^\epsilon - 1}\right).
\end{aligned}$$

Since $b^\epsilon - 1$ is a constant, then we can simply discard it. Also $n^{\log_b a - \epsilon}$ is smaller than $n^{\log_b a}$, so the sum ends up running in $O(n^{\log_b a})$; that is,

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} = O(n^{\log_b a}).$$

So $T(n) \approx n^{\log_b a} T(1) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$.

- **Case 2**: $f(m) = \Theta(m^{\log_b a})$. Then

$$\begin{aligned}
\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) &= \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \Theta\left(\frac{n}{b^i}\right)^{\log_b a} \\
&= \Theta\left(\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}\right) \\
&= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} \left(\frac{a}{b^{\log_b a}}\right)^i\right) \\
&= \Theta\left(n^{\log_b a} \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} 1\right) \\
&= \Theta\left(n^{\log_b a} \lfloor \log_b n \rfloor\right).
\end{aligned}$$

Thus,

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) = \Theta\left(n^{\log_b a} \log_2 n\right).$$

So $T(n) \approx n^{\log_b a} T(1) + O(n^{\log_b a} \log_2 n) = \Theta(n^{\log_b a} \log_2 n)$.

- **Case 3**: $f(m) = \Omega(m^{\log_b a+\epsilon})$ and $af(n/b) \le cf(n)$ for some $0 < c < 1$. By substitution,

$$
\begin{aligned}
f\left(\frac{n}{b}\right) &\le \frac{c}{a} f(n) \\
f\left(\frac{n}{b^2}\right) &\le \frac{c}{a} f\left(\frac{n}{b}\right) \\
f\left(\frac{n}{b^3}\right) &\le \frac{c}{a} f\left(\frac{n}{b^2}\right) \\
&\vdots \\
f\left(\frac{n}{b^i}\right) &\le \frac{c}{a} f\left(\frac{n}{b^{i-1}}\right).
\end{aligned}
$$

  Chaining these inequalities, we get

$$f\left(\frac{n}{b^i}\right) \le \frac{c^i}{a^i} f(n).$$

  Thus,

$$\sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right) \le \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i \left(\frac{c^i}{a^i} f(n)\right) < f(n) \sum_{i=0}^{\infty} c^i = \frac{f(n)}{1-c}.$$

  Since we had

$$T(n) \approx n^{\log_b a} T(1) + \sum_{i=0}^{\lfloor \log_b n \rfloor - 1} a^i f\left(\frac{n}{b^i}\right)$$

  and since $f(n) = \Omega(n^{\log_b a+\epsilon})$, we get

$$T(n) < n^{\log_b a} T(1) + O(f(n)) = O(f(n))$$

  but we also have

$$T(n) = aT(n/b) + f(n) > f(n).$$

  Thus, $T(n) = \Theta(f(n))$.

□

Extra reading:
- *Computing Master Theorem*

A final puzzle

Five pirates have to split 100 bars of gold. They all line up and proceed as follows:
1. *The first pirate in line gets to propose a way to split up the gold.*
2. *The pirates, including the one who proposed, vote on whether to accept the proposal. If the proposal is rejected, the pirate who made the proposal is killed.*
3. *The next pirate in line makes his proposal, and the four pirates vote again. If the vote is tied, the the proposing priate is still killed; only majority can accept a proposal. The process continues until a proposal is accepted or there is only one pirate left. Assume that every pirate*
   - *above all wants to live*
   - *given that he will be alive he wants to get as much gold as possible.*
   - *given maximal possible amount of gold, he wants to see any other pirate*

Extra reading:

- *Wikipedia article*
- *Pirate's game solution*

2 Lecture 4 – Integer multiplication

Content covered here can be found in the Lecture 4A and Lecture 4B recordings.

Date: June 14, 2020

2.1 Karatsuba trick

From lecture 2, the recurrence of the Karatsuba trick was $T(n) = 3T(n/2) + cn$. So $a = 3$, $b = 2$, $f(n) = cn$ and $n^{\log_b a} = n^{\log_2 3}$.

Since $1.5 < \log_2 3 < 1.6$, we have

$$f(n) = cn = O(n^{\log_2 3-\epsilon}) \quad \text{for any } 0 < \epsilon < 0.5.$$

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

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

2.1.1 Karatsuba's algorithm for 3 slices

Break the numbers A, B into three pieces. Then with $k = n/3$, we obtain

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

In other words,

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

- ie A_2 is shifted $2k$ binary bits to the left, A_1 is shifted k binary bits to the left and add A_0.
- Repeat the same for B:
$$B = B_2 2^{2k} + B_1 2^k + B_0.$$
- So
$$AB = A_2B_2 2^{4k} + (A_2B_1 + A_1B_2)2^{3k} + (A_2B_0 + A_1B_1 + A_0B_2)2^{2k} + (A_1B_0 + A_0B_1)2^k + A_0B_0.$$
  - Notice that there are five sums to deconstruct AB.
  - Rather than computing the nine individual products, use five multiplications if possible.

Define

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

- We form naturally corresponding polynomials:

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

Note that

$$
\begin{aligned}
A &= A_2 (2^k)^2 + A_1 2^k + A_0 = P_A(2^k). \\
B &= B_2 (2^k)^2 + B_1 2^k + B_0 = P_B(2^k).
\end{aligned}
$$

If we manage to compute 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 five multiplications, we 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 five values to **uniquely determine** $P_C(x)$.

Choose the smallest possible five integer values (by absolute value). Thus, we compute

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

For $P_A(x)$, we have

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

For $P_B(x)$, we have

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

We can now obtain $P_C(-2), P_C(-1), P_C(0), P_C(1), P_C(2)$ with only five multiplications of large numbers

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

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

$$\begin{aligned} P_C(0) &= P_A(0)P_B(0) = A_0B_0 \\ P_C(1) &= P_A(1)P_B(1) \\ &= (A_0 + A_1 + A_2)(B_0 + B_1 + B_2) \end{aligned}$$

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

Simplifying everything, we obtain

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

Solve the system of linear equations for C_0, C_1, C_2, C_3, C_4, we obtain

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

- Note that these expressions do not involve any multiplications of two large numbers and thus can be done in linear time.
- 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 $P_C(2^k)$ in linear time because computing $P_C(2^k)$ involves only binary shifts of the coefficients plus $O(k)$ additions.

- Thus we have obtained $A \cdot B$ with only five multiplications.

- We have replaced a multiplication of two n bit numbers with five multiplications of $n/3$ bit numbers with an overhead of additions, shifts, all doable in linear time cn. Thus,

$$T(n) = 5T(n/3) + cn.$$

Applying the Master Theorem, we have $a = 5$, $b = 3$, so consider $n^{\log_b a} = n^{\log_3 5} \approx n^{1.465\ldots}$. So we get $T(n) = O(n^{\log_3 5}) < O(n^{1.47})$.

Recall that the original Karatsuba algorithm runs in time

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

Thus, we got a significantly faster algorithm.

2.1.2 Generalising Karatsuba's algorithm

Slice the input numbers A, B into $n + 1$ many slices. For simplicity, let A, B have $(n + 1)k$ bits (k can be arbitrarily large but n is **fixed**).

Slice A, B into $n + 1$ pieces each

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

- Form the naturally corresponding polynomials

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

- As before, we have

$$A = P_A(2^k), \quad B = P_B(2^k), \quad AB = P_A(2^k) P_B(2^k) = (P_A(x) \cdot P_B(x))\Big|_{x=2^k}.$$

- Since $AB = (P_A(x) \cdot P_B(x))\Big|_{x=2^k}$, adopt the strategy
  - Figure out how to multiply polynomials fast to obtain

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

  - Evaluate $P_C(2^k)$.

- Note that $P_C(x) = P_A(x) \cdot P_B(x)$ is of degree $2n$

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

- We have

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

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

An important digression!

Let $\mathbf{A} = (A_0, A_1, \ldots, A_{n-1}, A_n)$ and $\mathbf{B} = (B_0, B_1, \ldots, B_{m-1}, B_m)$, be two sequences. Forming the two corresponding polynomials

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

and multiplying these two polynomials obtains their product

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

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

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

is *extremely important* and is called the **linear convolution** of the sequences $\mathbf{A}$ and $\mathbf{B}$ and is denoted by $\mathbf{C} = \mathbf{A} * \mathbf{B}$.

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

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

- For $P_A(x) = A_n x^n + \cdots + A_0$, these values can be obtained via a matrix multiplication.

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

- Such a matrix is called the *Vandermonde matrix* and if all x_i are distinct, then the matrix is invertible.
- If all x_i are distinct, given any values $P_A(x_k)$ the coefficients $A_0, A_1, \ldots, A_n$ of the polynomial are uniquely determined.

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

- Use (2) to go from polynomial to constant form and (1) to go from constant to polynomial form.
- Thus, for fixed input values $x_0, \ldots, x_n$ this switch between the two kinds of representations is done in **linear time**!
- **Strategy**:

1. Given two polynomials of degree at most n,

$$P_A(x) = A_n x^n + \cdots + A_0, \quad P_B(x) = B_n x^n + \cdots + B_0,$$

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

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

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

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

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

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

- Use $2n+1$ smallest possible integer values

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

- We find the values $P_A(m)$ and $P_B(m)$ for all m such that $-n \le m \le n$.
- 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

$$d \cdot A = \underbrace{A + A + \cdots + A}_{d}.$$

- Thus, all the values in $P_A(m)$ and $P_B(m)$ can be found in time linear in the number of bits of the input numbers.
- We now perform $2n+1$ multiplications of large numbers to obtain

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

- For $P_C(x) = P_A(x)P_B(x)$, these products are $2n+1$ many values of $P_C(x)$.
- Let $C_0, C_1, \ldots, C_{2n}$ be the coefficients of the product polynomial $C(x)$. We now have

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

  - This is just a system of linear equations that can be solved for $C_0, C_1, \ldots, C_{2n}$.
  - Apply the inverse Vandermonde matrix as described earlier.
  - The inverse matrix also involves only constant depending on n only.
  - Thus the coefficients C_i can be obtained in linear time.

- We get the recurrence for the complexity

$$T((n+1)k) = (2n+1)T(k+s) + ck.$$

  Let $N = (n+1)k$. Then

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

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

$$T(N) = (2n+1)T\left(\frac{N}{n+1}\right) + c \cdot N.$$

  - Apply the Master theorem, we have $a = 2n+1$, $b = n+1$, $f(N) = c \cdot N$.
  - Since $\log_b a = \log_{n+1}(2n+1) > 1$, we can choose a small ϵ such that also $\log_b a - \epsilon > 1$.
  - Consequently, for such an ϵ, we have

$$f(N) = c/(n+1)N = O(N^{\log_b a - \epsilon}).$$

  - Thus, the first case of the Master theorem applies. So we get

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

    * Note that

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

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

- But we would have to evaluate polynomials $P_A(x)$ and $P_B(x)$ both of degree n at values up to n. For example if $n = 2^{10}$, then it involves $n^n = (2^{10})^{10} \approx 1.27 \times 10^{3079}$ calculations.
- Thus, while evaluations of $P_A(x)$ and $P_B(x)$ for $x = -n \dots n$ can *theoretically* be done in linear time, $T(n) = cn$, the constant c is huge!

Moral: 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!

- Theoretically fast algorithm but huge overhead results in a slow algorithm!