

## 6.2 Master Theorem

Note that the cases do not cover all possibilities.

### Lemma 1

Let  $a \geq 1, b \geq 1$  and  $\epsilon > 0$  denote constants. Consider the recurrence

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

#### Case 1.

If  $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$  then  $T(n) = \Theta(n^{\log_b a})$ .

#### Case 2.

If  $f(n) = \Theta(n^{\log_b(a)} \log^k n)$  then  $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ ,  $k \geq 0$ .

#### Case 3.

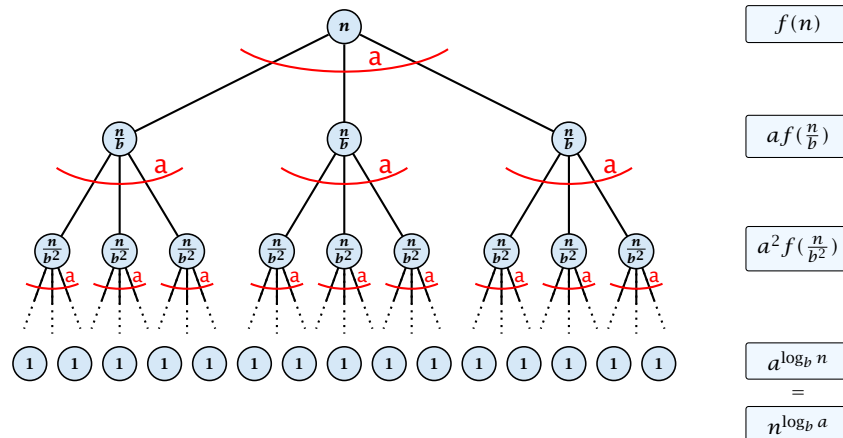
If  $f(n) = \Omega(n^{\log_b(a)+\epsilon})$  and for sufficiently large  $n$   $af\left(\frac{n}{b}\right) \leq cf(n)$  for some constant  $c < 1$  then  $T(n) = \Theta(f(n))$ .

## 6.2 Master Theorem

We prove the Master Theorem for the case that  $n$  is of the form  $b^\ell$ , and we assume that the non-recursive case occurs for problem size 1 and incurs cost 1.

## The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



## 6.2 Master Theorem

This gives

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

Case 1. Now suppose that  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

$$\begin{aligned} b^{-i(\log_b a - \epsilon)} &= b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i} \\ &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i \\ \sum_{i=0}^k q^i &= \frac{q^{k+1} - 1}{q - 1} \\ &= cn^{\log_b a - \epsilon} (b^{\epsilon \log_b n} - 1) / (b^\epsilon - 1) \\ &= cn^{\log_b a - \epsilon} (n^\epsilon - 1) / (b^\epsilon - 1) \\ &= \frac{c}{b^\epsilon - 1} n^{\log_b a} (n^\epsilon - 1) / (n^\epsilon) \end{aligned}$$

Hence,

$$T(n) \leq \left(\frac{c}{b^\epsilon - 1} + 1\right) n^{\log_b a} \Rightarrow T(n) = \mathcal{O}(n^{\log_b a}).$$

Case 2. Now suppose that  $f(n) \leq cn^{\log_b a}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} = cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 = cn^{\log_b a} \log_b n$$

Hence,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n) \Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log n).$$

Case 2. Now suppose that  $f(n) \geq cn^{\log_b a}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} = cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 = cn^{\log_b a} \log_b n$$

Hence,

$$T(n) = \Omega(n^{\log_b a} \log_b n) \Rightarrow T(n) = \Omega(n^{\log_b a} \log n).$$

Case 2. Now suppose that  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b \left(\frac{n}{b^i}\right)\right)^k \\ n = b^\ell \Rightarrow \ell = \log_b n &= cn^{\log_b a} \sum_{i=0}^{\ell - 1} \left(\log_b \left(\frac{b^\ell}{b^i}\right)\right)^k \\ &= cn^{\log_b a} \sum_{i=0}^{\ell - 1} (\ell - i)^k \\ &= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \approx \frac{1}{k} \ell^{k+1} \\ &\approx \frac{c}{k} n^{\log_b a} \ell^{k+1} \Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n). \end{aligned}$$

Case 3. Now suppose that  $f(n) \geq dn^{\log_b a + \epsilon}$ , and that for sufficiently large  $n$ :  $af(n/b) \leq cf(n)$ , for  $c < 1$ .

From this we get  $a^i f(n/b^i) \leq c^i f(n)$ , where we assume that  $n/b^{i-1} \geq n_0$  is still sufficiently large.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq \sum_{i=0}^{\log_b n - 1} c^i f(n) + \mathcal{O}(n^{\log_b a})$$

$$q < 1 : \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \leq \frac{1}{1-q} \leq \frac{1}{1-c} \leq \frac{1}{1-c} f(n) + \mathcal{O}(n^{\log_b a})$$

Hence,

$$T(n) \leq \mathcal{O}(f(n)) \quad \Rightarrow T(n) = \Theta(f(n)).$$

Where did we use  $f(n) \geq \Omega(n^{\log_b a + \epsilon})$ ?

## Example: Multiplying Two Integers

Suppose we want to multiply two  $n$ -bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers  $A$  and  $B$ :

$$\begin{array}{r} 110110101 \quad A \\ + 100010011 \quad B \\ \hline 10111001000 \end{array}$$

This gives that two  $n$ -bit integers can be added in time  $\mathcal{O}(n)$ .

## Example: Multiplying Two Integers

Suppose that we want to multiply an  $n$ -bit integer  $A$  and an  $m$ -bit integer  $B$  ( $m \leq n$ ).

$$\begin{array}{r} 10001 \times 1011 \\ \hline \phantom{10001} 10001 \\ \phantom{10001} 100010 \\ \phantom{10001} 0000000 \\ \phantom{10001} 10001000 \\ \hline 10111011 \end{array}$$

- This is also known as the "school method" for multiplying integers.
- Note that the intermediate numbers that are generated can have at most  $m + n \leq 2n$  bits.

Time requirement:

- ▶ Computing intermediate results:  $\mathcal{O}(nm)$ .
- ▶ Adding  $m$  numbers of length  $\leq 2n$ :  $\mathcal{O}((m+n)m) = \mathcal{O}(nm)$ .

## Example: Multiplying Two Integers

A recursive approach:

Suppose that integers  $A$  and  $B$  are of length  $n = 2^k$ , for some  $k$ .

$$\boxed{B_1} \boxed{B_0} \times \boxed{A_1} \boxed{A_0}$$

Then it holds that

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \quad \text{and} \quad B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Hence,

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

## Example: Multiplying Two Integers

### Algorithm 3 mult( $A, B$ )

1: <b>if</b> $ A  =  B  = 1$ <b>then</b>	$\mathcal{O}(1)$
2: <b>return</b> $a_0 \cdot b_0$	$\mathcal{O}(1)$
3: split $A$ into $A_0$ and $A_1$	$\mathcal{O}(n)$
4: split $B$ into $B_0$ and $B_1$	$\mathcal{O}(n)$
5: $Z_2 \leftarrow \text{mult}(A_1, B_1)$	$T(\frac{n}{2})$
6: $Z_1 \leftarrow \text{mult}(A_1, B_0) + \text{mult}(A_0, B_1)$	$2T(\frac{n}{2}) + \mathcal{O}(n)$
7: $Z_0 \leftarrow \text{mult}(A_0, B_0)$	$T(\frac{n}{2})$
8: <b>return</b> $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$	$\mathcal{O}(n)$

We get the following recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

## Example: Multiplying Two Integers

**Master Theorem:** Recurrence:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$       $T(n) = \mathcal{O}(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$     $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- ▶ Case 3:  $f(n) = \Omega(n^{\log_b a + \epsilon})$       $T(n) = \Theta(f(n))$

In our case  $a = 4$ ,  $b = 2$ , and  $f(n) = \mathcal{O}(n)$ . Hence, we are in Case 1, since  $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$ .

We get a running time of  $\mathcal{O}(n^2)$  for our algorithm.

⇒ Not better than the “school method”.

## Example: Multiplying Two Integers

We can use the following identity to compute  $Z_1$ :

$$\begin{aligned} Z_1 &= A_1 B_0 + A_0 B_1 && = Z_2 && = Z_0 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \underbrace{A_1 B_1}_{Z_2} - \underbrace{A_0 B_0}_{Z_0} \end{aligned}$$

Hence,

### Algorithm 4 mult( $A, B$ )

1: <b>if</b> $ A  =  B  = 1$ <b>then</b>	$\mathcal{O}(1)$
2: <b>return</b> $a_0 \cdot b_0$	$\mathcal{O}(1)$
3: split $A$ into $A_0$ and $A_1$	$\mathcal{O}(n)$
4: split $B$ into $B_0$ and $B_1$	$\mathcal{O}(n)$
5: $Z_2 \leftarrow \text{mult}(A_1, B_1)$	$T(\frac{n}{2})$
6: $Z_0 \leftarrow \text{mult}(A_0, B_0)$	$T(\frac{n}{2})$
7: $Z_1 \leftarrow \text{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$	$T(\frac{n}{2}) + \mathcal{O}(n)$
8: <b>return</b> $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$	$\mathcal{O}(n)$

A more precise (correct) analysis would say that computing  $Z_1$  needs time  $T(\frac{n}{2} + 1) + \mathcal{O}(n)$ .

## Example: Multiplying Two Integers

We get the following recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

**Master Theorem:** Recurrence:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$       $T(n) = \mathcal{O}(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$     $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- ▶ Case 3:  $f(n) = \Omega(n^{\log_b a + \epsilon})$       $T(n) = \Theta(f(n))$

Again we are in Case 1. We get a running time of  $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ .

A huge improvement over the “school method”.