Master Theorem

CSE 5311: Design and Analysis of Algorithms

Alex Dillhoff

The University of Texas at Arlington

Introduction

Master Theorem

Recurrence trees can are useful in determining the runtime complexity of divide-and-conquer algorithms.

There are certain recurrences that can be solved using a simple formula called the **Master Theorem**.

Master Theorem

The **Master Theorem** can be used to solve any recurrence of the form

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

- n is the size of the problem,

Master Theorem

The **Master Theorem** can be used to solve any recurrence of the form

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

- n is the size of the problem,
- $a \geq 1$ is the number of subproblems,

The **Master Theorem** can be used to solve any recurrence of the form

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

- n is the size of the problem,
- $a \geq 1$ is the number of subproblems,
- $b > 1$ is the factor by which the problem size is reduced, and

Master Theorem

The **Master Theorem** can be used to solve any recurrence of the form

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

- n is the size of the problem,
- $a \geq 1$ is the number of subproblems,
- $b > 1$ is the factor by which the problem size is reduced, and
- $f(n)$ is the cost of the work done outside of the recursive calls.

Each recurrence is solved in $T(n/b)$ time.

$f(n)$ would include the cost of dividing and recombining the problem.

Master Theorem

Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n),$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ such that $a = a' + a''$.

Master Theorem

Then $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

Master Theorem

Then $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Master Theorem

Then $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq kf(n)$ for some constant $k < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Breaking it Down

$n^{\log_b a}$ is called the **watershed function**.

$f(n)$ is compared to it to determine which case applies.

Breaking it Down

$n^{\log_b a}$ is called the **watershed function**.

$f(n)$ is compared to it to determine which case applies.

- If the watershed function grows at a faster rate than $f(n)$, case 1 applies.

Breaking it Down

$n^{\log_b a}$ is called the **watershed function**.

$f(n)$ is compared to it to determine which case applies.

- If the watershed function grows at a faster rate than $f(n)$, case 1 applies.
- If they grow at the same rate, case 2 applies.

Breaking it Down

$n^{\log_b a}$ is called the **watershed function**.

$f(n)$ is compared to it to determine which case applies.

- If the watershed function grows at a faster rate than $f(n)$, case 1 applies.
- If they grow at the same rate, case 2 applies.
- If $f(n)$ grows at a faster rate, case 3 applies.

Case 1

In case 1, the watershed function should grow faster than $f(n)$ by a factor of n^ϵ for some $\epsilon > 0$.

Case 2

In case 2, technically the watershed function should grow at least the same rate as $f(n)$, if not faster.

It grows faster by a factor of $\Theta(\log^k n)$, where $k \geq 0$.

You can think of the extra $\log^k n$ as an augmentation to the watershed function to ensure that they grow at the same rate.

In most cases, $k = 0$ which results in $T(n) = \Theta(n^{\log_b a} \log n)$.

Case 3

Since case 2 allows for the watershed function to grow faster than $f(n)$, case 3 requires that it grow at least **polynomially** faster.

- $f(n)$ should grow faster by at least a factor of $\Theta(n^\epsilon)$ for some $\epsilon > 0$.
- The driving function must satisfy the regularity condition $af(n/b) \leq kf(n)$ for some constant $k < 1$ and all sufficiently large n.
- This condition ensures that the cost of the work done outside of the recursive calls is not too large.

Applying the Master Theorem

In most cases, the master method can be applied by looking at the recurrence and applying the relevant case.

If the driving and watershed functions are not immediately obvious, you can use a different method.

Example: Merge Sort

Merge Sort has a recurrence of the form $T(n) = 2T(n/2) + \Theta(n)$.

- The driving function is $f(n) = \Theta(n)$.

Example: Merge Sort

Merge Sort has a recurrence of the form $T(n) = 2T(n/2) + \Theta(n)$.

- The driving function is $f(n) = \Theta(n)$.
- The constants a and b are both 2, so the watershed function is $n^{\log^2 2}$, which is n.

Example: Merge Sort

Merge Sort has a recurrence of the form $T(n) = 2T(n/2) + \Theta(n)$.

- The driving function is $f(n) = \Theta(n)$.
- The constants a and b are both 2, so the watershed function is $n^{\log^2 2}$, which is n.
- Since $f(n)$ grows at the same rate as the watershed function, case 2 applies.

Example: Merge Sort

Merge Sort has a recurrence of the form $T(n) = 2T(n/2) + \Theta(n)$.

- The driving function is $f(n) = \Theta(n)$.
- The constants a and b are both 2, so the watershed function is $n^{\log_2 2}$, which is n.
- Since $f(n)$ grows at the same rate as the watershed function, case 2 applies.
- Therefore, $T(n) = \Theta(n \log n)$.

Example: Merge Sort

The recurrence of the divide and conquer version of matrix multiplication for square matrices is $T(n) = 8T(n/2) + \Theta(1)$

- Given $a = 8$ and $b = 2$, we can see that the complexity is inherent in the recurrence, not the driving function.

Example: Merge Sort

The recurrence of the divide and conquer version of matrix multiplication for square matrices is $T(n) = 8T(n/2) + \Theta(1)$.

- Given $a = 8$ and $b = 2$, we can see that the complexity is inherent in the recurrence, not the driving function.
- The watershed function is $n^{\log_2 8}$, which is n^3.

Example: Merge Sort

The recurrence of the divide and conquer version of matrix multiplication for square matrices is $T(n) = 8T(n/2) + \Theta(1)$.

- Given $a = 8$ and $b = 2$, we can see that the complexity is inherent in the recurrence, not the driving function.
- The watershed function is $n^{\log_2 8}$, which is n^3.
- This grows at a faster rate than $\Theta(1)$, so case 1 applies.

Example: Merge Sort

The recurrence of the divide and conquer version of matrix multiplication for square matrices is $T(n) = 8T(n/2) + \Theta(1)$.

- Given $a = 8$ and $b = 2$, we can see that the complexity is inherent in the recurrence, not the driving function.
- The watershed function is $n^{\log_2 8}$, which is n^3.
- This grows at a faster rate than $\Theta(1)$, so case 1 applies.
- Therefore, $T(n) = \Theta(n^3)$.

Class Example

$$T(n) = 9T(n/3) + n.$$

$a = 9, b = 3, f(n) = n$

$n^{\log_3 9} = n^2$

$n^{2 \cdot 1} = f(n)$

case 1 applies => $\Theta(n^2)$

Class Example

$a = 3,\ b = 4$ $\quad f(n) = n\lg n$

$n^{\log_4 3} = n^{0.793}$

Case 3

$$T(n) = 3T(n/4) + n\lg n.$$

$\Theta(n\lg n)$

$a f(n/b) \leq c f(n)$

$3(n/4)\lg(n/4) \leq (3/4)n\lg n = c f(n)$

$c = \frac{3}{4}$

Class Example

$a = 5$

$b = 2$

$f(n) = n^2$

$n^{\log_2 5}$

$$T(n) = 5T(n/2) + \Theta(n^2).$$

$n^2 < n^{\log_2 5}$

case 1 applies

$T(n) = \Theta(n^{\lg 5})$

Class Example

$a = 27$

$b = 3$

$f(n) = n^3 \lg n$

$n^{\log_3 27} = n^3$

$$T(n) = 27T(n/3) + \Theta(n^3 \lg^{k=1} n).$$

$n^3 \lg^k n$

$T(n) = \Theta(n^3 \lg^2 n)$

Class Example

$a = 5$

$b = 2$

$f(n) = n^3$

$n^{\log_2 5 + \epsilon} \leq n^3$

$$T(n) = 5T(n/2) + \Theta(n^3).$$

$T(n) = \Theta(n^3)$

Class Example

$T(n) = T(2n/3) + T(n/3) + \Theta(n)$

$a = 27$

$b = 3$

$f(n) = n^3 \lg^{-1} n$

$n^{\log_3 27} = n^3$

$$T(n) = 27T(n/3) + \Theta(n^3/\lg n).$$

$k \geq 0$

$k < 0$ $\therefore$ cannot use master method