Master Theorem

In this tutorial, you will learn what master theorem is and how it is used for solving recurrence relations.

The master method is a formula for solving recurrence relations of the form:

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

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed

     to have the same size.

f(n) = cost of the work done outside the recursive call,

      which includes the cost of dividing the problem and

      cost of merging the solutions


Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.
```

An asymptotically positive function means that for a sufficiently large value of `n`, we have `f(n) > 0`.

> The master theorem is used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.

Master Theorem

If $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function, then the time complexity of a recursive relation is given by

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

where, T(n) has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$, then $T(n) = \Theta(f(n))$.

$\epsilon > 0$ is a constant.

Each of the above conditions can be interpreted as:

1. If the cost of solving the sub-problems at each level increases by a certain factor, the value of $f(n)$ will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level ie. $n^{\log_b a}$
2. If the cost of solving the sub-problem at each level is nearly equal, then the value of $f(n)$ will be $n^{\log_b a}$. Thus, the time complexity will be $f(n)$ times the total number of levels ie. $n^{\log_b a} * \log n$
3. If the cost of solving the subproblems at each level decreases by a certain factor, the value of $f(n)$ will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of $f(n)$.

Solved Example of Master Theorem

$$T(n) = 3T(n/2) + n2$$

Here,

$a = 3$

$n/b = n/2$

$f(n) = n^2$

$\log_b a = \log_2 3 \approx 1.58 < 2$

ie. $f(n) < n^{\log_b a+\epsilon}$, where, ϵ is a constant.

Case 3 implies here.

Thus, $T(n) = f(n) = \Theta(n^2)$

Master Theorem Limitations

The master theorem cannot be used if:

- $T(n)$ is not monotone. eg. $T(n) = \sin n$
- $f(n)$ is not a polynomial. eg. $f(n) = 2^n$
- a is not a constant. eg. $a = 2n$
- $a < 1$

Master theorem (analysis of algorithms)

In the analysis of algorithms, the **master theorem for divide-and-conquer recurrences** provides an asymptotic analysis (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms. The approach was first presented by Jon Bentley, Dorothea Blostein (née Haken), and James B. Saxe in 1980, where it was described as a "unifying method" for solving such recurrences.[1] The name "master theorem" was popularized by the widely-used algorithms textbook *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein.

Not all recurrence relations can be solved with the use of this theorem; its generalizations include the Akra–Bazzi method.

Introduction

Consider a problem that can be solved using a recursive algorithm such as the following:

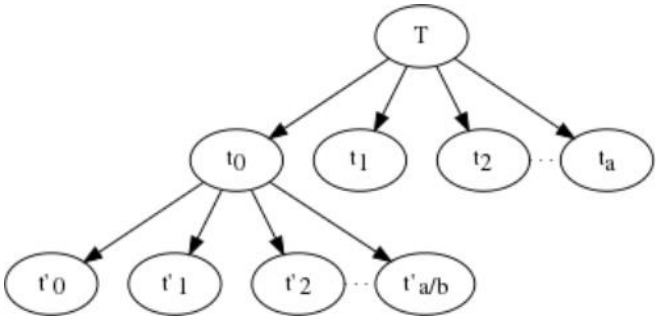
```
procedure p(input x of size n):

    if n < some constant k:

        Solve x directly without recursion

    else:
```

The above algorithm divides the problem into a number of subproblems recursively, each subproblem being of size n/b. Its solution tree has a node for each recursive call, with the children of that node being the other calls made from that call. The leaves of the tree are the base cases of the recursion, the subproblems (of size less than k) that do not recurse. The above example would have a child nodes at each non-leaf node. Each node does an amount of work that corresponds to the size of the subproblem n passed to that instance of the recursive call and given by $f(n)$. The total amount of work done by the entire algorithm is the sum of the work performed by all the nodes in the tree.



Solution tree.

The runtime of an algorithm such as the p above on an input of size n, usually denoted $T(n)$, can be expressed by the recurrence relation

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),$$

where $f(n)$ is the time to create the subproblems and combine their results in the above procedure. This equation can be successively substituted into itself and expanded to obtain an expression for the total amount of work done.[2] The master theorem allows many recurrence relations of this form to be converted to Θ-notation directly, without doing an expansion of the recursive relation.

Generic form

The master theorem always yields asymptotically tight bounds to recurrences from divide and conquer algorithms that partition an input into smaller subproblems of equal sizes, solve the subproblems recursively, and then combine the subproblem solutions to give a solution to the original problem. The time for such an algorithm can be expressed by adding the work that they perform at the top level of their recursion (to divide the problems into subproblems and then combine the subproblem solutions) together with the time made in the recursive calls of the algorithm. If $T(n)$ denotes the total time for the algorithm on an input of size n, and $f(n)$ denotes the amount of time taken at the top level of the recurrence then the time can be expressed by a recurrence relation that takes the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Here n is the size of an input problem, a is the number of subproblems in the recursion, and b is the factor by which the subproblem size is reduced in each recursive call (b>1). Crucially, a and must not depend on n. The theorem below also assumes that, as a base case for the recurrence, $T(n) = \Theta(1)$ when n is less than some bound $\kappa > 0$, the smallest input size that will lead to a recursive call.

Recurrences of this form often satisfy one of the three following regimes, based on how the work to split/recombine the problem $f(n)$ relates to the *critical exponent* $c_{\text{crit}} = \log_b a$. (The table below uses standard big O notation).

$$c_{\text{crit}} = \log_b a = \log(\#\text{subproblems}) / \log(\text{relative subproblem size})$$

Case	Description	Condition on $f(n)$ in relation to c_{crit}, i.e. $\log_b a$	Master Theorem bound	Notational examples
1	Work to split/recombine a problem is dwarfed by subproblems. i.e. the recursion tree is leaf-heavy	When $f(n) = O(n^c)$ where $c < c_{\text{crit}}$ (upper-bounded by a lesser exponent polynomial)	... then $T(n) = \Theta\left(n^{c_{\text{crit}}}\right)$ (The splitting term does not appear; the recursive tree structure dominates.)	If $b = a^2$ and $f(n) = O(n^{1/2-\epsilon})$, then $T(n) = \Theta(n^{1/2})$.
2	Work to split/recombine a problem is comparable to subproblems.	When $f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for a $k \geq 0$ (rangebound by the critical-exponent polynomial, times zero or more optional $\log$s)	... then $T(n) = \Theta\left(n^{c_{\text{crit}}} \log^{k+1} n\right)$ (The bound is the splitting term, where the log is augmented by a single power.)	If $b = a^2$ and $f(n) = \Theta(n^{1/2})$, then $T(n) = \Theta(n^{1/2} \log n)$. If $b = a^2$ and $f(n) = \Theta(n^{1/2} \log n)$, then $T(n) = \Theta(n^{1/2} \log^2 n)$.
3	Work to split/recombine a problem dominates subproblems. i.e. the recursion tree is root-heavy.	When $f(n) = \Omega(n^c)$ where $c > c_{\text{crit}}$ (lower-bounded by a greater-exponent polynomial)	... this doesn't necessarily yield anything. Furthermore, if $af\left(\frac{n}{b}\right) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n (often called the *regularity condition*) then the total is dominated by the splitting term $f(n)$: $T(n) = \Theta(f(n))$	If $b = a^2$ and $f(n) = \Omega(n^{1/2+\epsilon})$ and the regularity condition holds, then $T(n) = \Theta(f(n))$.

A useful extension of Case 2 handles all values of k[3]

$f(n)$

c_{crit}

$\log_b a$

Case	Condition on $f(n)$ in relation to $n^{c_{\text{crit}}}$, i.e.	Master Theorem bound	Notational examples
2a	$f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for any $k > -1$ When $b = a^2$	... then $T(n) = \Theta(n^{c_{\text{crit}}} \log^{k+1} n)$ (The bound is the splitting term, where the log is augmented by a single power.)	$f(n) = \Theta(n^{1/2} / \log^{1/2} n)$ If $T(n) = \Theta(n^{1/2} \log^{1/2} n)$ and $k = -\frac{1}{2}$, then $T(n) = \Theta(n^{1/2} \log \log n)$.
2b	$f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for $k = -1$ When $b = a^2$	... then $T(n) = \Theta(n^{c_{\text{crit}}} \log \log n)$ (The bound is the splitting term, where the log reciprocal is replaced by an iterated log.)	$f(n) = \Theta(n^{1/2} / \log n)$ If $T(n) = \Theta(n^{1/2} \log \log n)$ and $k = -1$, then $T(n) = \Theta(n^{1/2} \log \log \log n)$.
2c	$f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for any $k < -1$ When $b = a^2$	... then $T(n) = \Theta(n^{c_{\text{crit}}})$ (The bound is the splitting term, where the log disappears.)	If $f(n) = \Theta(n^{1/2} / \log^2 n)$ and $T(n) = \Theta(n^{1/2})$, then $k = -2$.

Examples

Case 1 example

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

As one can see from the formula above:

$a = 8, b = 2, f(n) = 1000n^2$, so $f(n) = O(n^c)$ $c = 2$

Next, we see if we satisfy the case 1 condition:

$a = \log_b 8 = \log_2 8 \geq c$.

It follows from the first case of the master theorem that

$$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^3\right)$$

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 1001n^3 - 1000n^2$, assuming $T(1) = 1$).

Case 2 example

$$T(n) = 2T\left(\frac{n}{2}\right) + 10n$$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ c = 1,\ f(n) = 10n$$

$$f(n) = \Theta\left(n^c \log^k n\right) \qquad c = 1, k = 0$$

Next, we see if we satisfy the case 2 condition:

$a = \log_b 2 = \log_2 2 = 1$, and therefore, c and $\log_b a$ are equal

So it follows from the second case of the master theorem:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right) = \Theta\left(n^1 \log^1 n\right) = \Theta(n \log n)$$

Thus the given recurrence relation $T(n) = \Theta(n \log n)$ was solved.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = n + 10n \log_2 n$, assuming $T(1) = 1$).

Case 3 example

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ f(n) = n^2$$

$$f(n) = \Omega\left(n^c\right) \qquad c = 2$$

Next, we see if we satisfy the case 3 condition:

$$+ \frac{}{\log n}$$

$a = \log_b 2 = \log_2$, and therefore, yes, $a \quad \log_b$

The regularity condition also holds:

$2 \left(\frac{n^2}{4}\right) \leq kn^2$, choosing $k = 1/2$

So it follows from the third case of the master theorem:

$T(n) = \Theta(f(n)) = \Theta\left(n^2\right).$

Thus the given recurrence relation $T(n)$ was in $\Theta(n^2)$, that complies with the $f(n)$ of the original formula.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 2n^2$, assuming $T(1)$.) $- n$

Inadmissible equations

The following equations cannot be solved using the master theorem:[4]

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$

  a is not a constant; the number of subproblems should be fixed

- $T(n) = 2T\left(\frac{n}{2}\right) \quad n$

  non-polynomial difference between $f(n)$ and $n^{\log_b a}$ (see below; extended version applies)

- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$

  $a < 1$ cannot have less than one sub problem

- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

$f(n)$, which is the combination time, is not positive

- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

  case 3 but regularity violation.

In the second inadmissible example above, the difference between $f(n)$ and $n^{\log_b a}$ can be expressed with the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n/\log n}{n^{\log_2 2}} = \frac{1}{\log n}$. It is clear that $\epsilon > 0$ for

$$= \Theta(n \log \log n)$$

Application to common algorithms

$a = 1, b = 2, c = 0, k = 0$

Algorithm	Recurrence relationship	Run time	Comment
Binary search	$T(n) = T\left(\frac{n}{2}\right) + O(1)$	$O(\log n)$	Apply Master theorem case , where $a = 2, b = 2, c = 0$ [5]
Binary tree traversal	$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$	$O(n)$	Apply Master theorem case where [5]
Optimal sorted matrix search	$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n)$	$O(n)$	Apply the Akra–Bazzi theorem for and to get
Merge sort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$	Apply Master theorem case , where $a = 2, b = 2, c = 1, k = 0$