# Why does the master theorem work?

Ajitesh Dasaratha

October 2025

There is a part of your cheatsheet that says:

Suppose you have a recurrence of the form

$$T(n) = rT\left(\tfrac{n}{c}\right) + f(n).$$

The *master theorem* gives a good asymptotic estimate of the recurrence. If the work at each level is:

Decreasing: $rf\left(\frac{n}{c}\right) = \kappa f(n)$ where $\kappa < 1$, $T(n) = O(f(n))$

Equal: $rf\left(\frac{n}{c}\right) = f(n)$, $T(n) = O(f(n) \cdot \log n)$

Increasing: $rf\left(\frac{n}{c}\right) = Kf(n)$ where $K > 1$, $T(n) = O\left(n^{\log_c r}\right)$

How does this work?

One fact that'll get used in all of these proofs is that the number of levels in the tree is roughly $\log_c n$. Why? Let the tree have $l$ levels. At each level, the problem size is divided by a factor of $c$ from the previous level, so at the root you have problem size $n$, one level down it's $n/c$, then $n/c^2$ and so on. Following this logic, the problem size at the last level is $n/c^l$. We also know that at the last level, our problem size is a very small number, which for simplicity we'll say is 1.

So,

$$1 \approx \frac{n}{c^l} \implies c^l \approx n \implies l \approx \log_c n$$

Now we move on to proving the three cases.

**Equal work:** The total work done is *work done at each level* $\times$ *number of levels* $=$ $f(n) \times \log_c n$, but in Big O that's just $O(f(n) \cdot \log n)$

**Decreasing work:** Let some $u < 1$ be an *upper bound* on how much work gets passed on to the next level. So if at level $i$ you have $w$ work, at level $i+i$ you have at most $uw < w$ work. Then, at the root level you have $f(n)$ work, on the next at most $uf(n)$, then $u^2 f(n)$, and so on. If we pretend the tree has infinite levels, then using the geometric series formula the total work is

$$f(n) + uf(n) + u^2 f(n) + ... = \frac{f(n)}{1-u} = O(f(n))$$

Since the total work done is less than what this infinite sum suggests, $O(f(n))$ is a valid upper bound. And since $f(n)$ work is done at the root level, we also know the runtime is at least $O(f(n))$, so this is the tightest bound we can get.

*Before you get into the last case, note that I will be using the identity $a^{\log_b c} = c^{\log_b a}$ as fact. If you're curious why this holds, then like all important questions in life, someone has answered it here on Reddit.*

**Increasing work:** This part of the master theorem works only if $f(n)$ grows slower than $n^{\log_c r}$. First consider functions of the form $f(n) = n^p$, $0 < p \leq \log_c r$. The recurrence is then $T(n) = rT(n/c) + n^p$. The amount of work one level below the root is:

$$rT(n/c) = r(rT(\frac{n}{c^2}) + \frac{n^p}{c^p}) = r^2 T(n/c) + r\frac{n^p}{c^p}$$

The work done at the root is $n^p$, and at the next level its $\frac{r}{c^p} \times n^p$. So the work gets multiplied by a factor of $\frac{r}{c^p}$ at every level, and there are $l = \log_c n$ levels. Using geometric sum, total work is then:

$$n^p(1 + \frac{r}{c^p}\frac{r^2}{c^{2p}} + ... + \frac{r^{l-1}}{c^{(l-1)p}}) = n^p \cdot \frac{\left(\frac{r}{c^p}\right)^l - 1}{\frac{r}{c^p} - 1}$$

To simplify,

$$\left(\frac{r}{c^p}\right)^{\log_c n} = n^{\log_c \frac{r}{c^p}} = n^{\log_c r - \log_c c^p} = n^{\log_c r - p}$$

So total work is

$$n^p \cdot \frac{\left(\frac{r}{c^p}\right)^l - 1}{\frac{r}{c^p} - 1} = n^p \cdot \frac{n^{\log_c r - p} - 1}{\frac{r}{c^p} - 1} = \frac{n^{\log_c r} - n^p}{\frac{r}{c^p} - 1}$$

Since the bottom is constant, and $n^{\log_c r}$ grows faster than $n^p$, we finally get

$$\frac{n^{\log_c r} - n^p}{\frac{r}{c^p} - 1} = O(n^{\log_c r} - n^p) = O(n^{\log_c r})$$

So we have shown that for any $f(n) = n^p,\ p < \log_c r,\ T(n) = O(n^{\log_c r})$. If $f(n)$ grows slower than *any* such polynomial $n^p$, then we can be sure that $O(n^{\log_c r})$ is a still valid upper bound to the running time, since we are passing down *less* work at every level. But how do we know $O(n^{\log_c r})$ is the *tighest* upper bound?

Now imagine if we had a function that was growing slower than $n^p$. In fact, let's assume $f(n) = 0$ just to prove a point. In this case, the total work is proportional to the number of leaves in this tree, since no work is done at any of the levels except the last level. At that level, we a very small subproblem, so each takes a constant amount of time to solve. How many leaves are there? The tree has $\log_c n$ levels, and each level multiplies the number of leaves by $r$, since that's how many subproblems we divide each problem into. So there are $r^{\textit{number of levels in tree}} = r^{\log_c n} = n^{\log_c r}$. Since each takes a constant time to solve, the Big O runtime is still $O(n^{\log_c r})$.

We have proved that whether the work at each level is 0 or $n^p,\ p < \log_c r$, the Big O runtime is $O(n^{\log_c r})$ in both cases. What if a function grows at an intermediate rate? Well, since it's growth rate is bounded both above and below by $O(n^{\log_c r})$, we know that the tighest bound we can establish for its runtime is $O(n^{\log_c r})$.