The Master Theorem

The master theorem provides a formula for the time efficiency class of an algorithm whose running time function satisfies a specific recurrence relation, namely:

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

The most common situation where this relation comes up is the divide-and-conquer algorithms:

- b is the factor that determines how much smaller the subproblems are compared to the original problem (often 2 or 3).
- a determines how many of the subproblems must be solved ($a = 1$ when we do *decrease-by-constant-factor*).
- $f(n)$ is the **glue cost**: How much more work we need to do on top of solving the subproblems in order to solve the original problem.

  Master Theorem

  If $f(n) = \Theta\left(n^d\right)$, then compare $\log_b a$ to d (equivalently compare a to b^d):

  - If $\log_b a > d$ ($a > b^d$) then $T(n) = \Theta\left(n^{\log_b a}\right)$
  - If $\log_b a < d$ ($a < b^d$) then $T(n) = \Theta\left(n^d\right)$
  - If $\log_b a = d$ ($a = b^d$) then $T(n) = \Theta\left(n^d \log n\right)$

  Similar results hold for O and Ω.

Group work:

- If $f(n)$ is a constant function, what is d?
- In binary search what are the values of a, b, d? What does the theorem tell us in that case about the time efficiency of binary search?
- Same question for mergesort.
- Karatsuba's algorithm for multiplying two n-binary-digit integers has values $a = 3$, $b = 2$, $d = 1$. What does the Master Theorem tell us about the time efficiency of this multiplication?
  - How would you normally multiply two n-binary-digit integers? What would be the time efficiency of that algorithm?
- Imagine a problem that would normally take n^3 time to run with a brute force method. We are trying to instead use a divide-and-conquer algorithm where $b = 2$ and $d = 2$. What is the largest number a of subproblems that we can use in our algorithm, so that our algorithm would still be faster than the normal n^3 algorithm? Begin by making a table of the running time $T(n)$ for $a = 1, 2, 3, ...$

Explanation of the Master Theorem

Cost breakdown

Imagine our divide-and-conquer algorithm in levels:

- At the level 0 we have to solve 1 problem of size n.
- At the level 1 we have to solve a problems of size n/b.
- At the level 2 we have to solve a^2 problems of size n/b^2, as each of the previous a problems were split in subproblems.
- $\ldots$ and so on
- The last level is the level k when $n/b^k = 1$, so $k = \log_b n$. At that level we have to solve $a^k = a^{\log_b n} = n^{\log_b a}$ subproblems of size 1.

It clearly will take us $O(n^{\log_b a})$ time to solve those base case subproblems. But in order to solve our initial problem, we must also consider all the *glue costs* at each level, and what they would add up to.

Now let's look at the cost of that glueing:

- It costs us $f(n) = n^d$ to glue together the problems at the 0 level.
- It costs us $f(n/b) = n^d/b^d$ to glue the subproblems of each of the a problems at the 1 level, for a total of $\frac{a}{b^d}n^d$ cost for that level.
- Similarly it will cost us $\left(\frac{a}{b^d}\right)^2 n^d$ to glue the subproblems at the next level, and so on.

Let us give a name to this factor $C = \frac{a}{b^d}$. Then at each level the glue cost is multiplied by C. Therefore the total glue cost is:

$$n^d + Cn^d + C^2n^d + \cdots + C^{k-1}n^d = \left(1 + C + C^2 + \cdots + C^{k-1}\right) n^d$$

This factor C considers the balance between the fact that our problems are becoming smaller and smaller, so they cost less and less to glue by a factor of b^d, but they are also becoming more and more by a factor of a.

- When $C > 1$ ($a/b^d > 1$ or $\log_b a > d$) the number of problems increases faster than the corresponding decrease in their glue cost, resulting in increasing glue costs at every level.
- When $C < 1$ ($a/b^d < 1$ or $\log_b a < d$) the number of problems increases slower than the corresponding decrease in their glue cost, resulting in decreasing glue costs at every level.
- When $C = 1$ the two are in balance, resulting in an equal cost at each level, but we still need to do $k = \log_b n$ levels.

Base case costs vs glue costs

Before considering the different cases, let's consider the last glue cost, namely the cost on the next-to-last level. We have a cost of $C^{k-1}n^d$, but since C is a constant this is basically the same as $C^k n^d$.

Now keeping in mind that $k = \log_b n$ and that $C = \frac{a}{b^d}$ we have for the last glue cost:

$$C^k n^d = C^{\log_b n} n^d = n^{\log_b C} n^d = n^{\log_b a}$$

the second step being some log-magic and the last step being because:

$$\log_b C = \log_b \left(\frac{a}{b^d}\right) = \log_b a - d$$

What this means is that this last glue cost is at least as much as solving all the base case subproblems. Therefore in **order to estimate the total cost of $T(n)$ only the glue costs matter.**

Estimating the glue costs

Let us consider the case when $C = 1$. Then the total glue cost for all levels from the above formula is:

$$(1 + 1 + 1 + \cdots + 1)n^d = n^d k = n^d \log_b n = \Theta\left(n^d \log n\right)$$

If $C \neq 1$, then we can use the geometric sum formula:

$$1 + C + C^2 + \cdots + C^{k-1} = \frac{C^k - 1}{C - 1}$$

The total glue cost is then:

$$\frac{C^k - 1}{C - 1} n^d$$

Now, if $C < 1$, then $\frac{C^k-1}{C-1} = \Theta(1)$, and therefore the glue cost is n^d.

Lastly, if $C > 1$ then:

$$\frac{C^k - 1}{C - 1} = \Theta(C^k) = \Theta(n^{\log_b C}) = \Theta\left(n^{\log_b a - d}\right)$$

And the total glue cost is then:

$$\Theta(n^{\log_b a - d} n^d) = \Theta\left(n^{\log_b a}\right)$$

So to summarize:

- Solving all the base case subproblems costs $n^{\log_b a}$. There is no way for the algorithm to do better than that. This is in the same order as the last glue cost, and therefore to estimate the total cost we can focus on the glue costs only.

- If $\log_b a > d$, then the glue costs add up to the order of $n^{\log_b a}$, for a total running time of $\Theta(n^{\log_b a})$.
- If $\log_b a = d$, then the glue costs add up to $n^{\log_b a} \log n$, for a total running time of $\Theta(n^d \log n)$.
- If $\log_b a < d$, then the glue costs add up to n^d, for a total running time of $\Theta(n^d)$.