17. THE MASTER METHOD

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

December 2024

In this lecture, we will learn a method that provides us with a "table-like" solution for a common type of recurrences. In practice, if we encounter a recursive algorithm, we have to synthesize the recurrence relation that describes its time complexity, then see if it matches the format required by the master method. If it does, we then have to further narrow it down to one of three special cases of the method, then get the answer associated with that case. If the recurrence does not have this special form, that is when we need to apply the methods learned in the previous lecture: the recursion-tree method or the substitution method[†]. But before we can start confidently using this method, we will go through its proof, so that you can understand how and why it works.

1 The Master Theorem

Let us consider recursive algorithms that, on an input of size n, perform a recursive calls on a chunk of size $\frac{n}{b}$ of the input and do an amount of work equal to $f(n)$. I.e. we want to consider recurrences of the form:

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

$$T(1) = \Theta(1)$$

With $a \geq 1, b > 1$ and $f(n)$ a positive function.

> **Note 17.1.** As with our analysis of merge sort, we will start by only considering what happens for values of n that are powers of b; such that repeated division by b ends up at 1.
> Later we will address other values of n and see why our result still holds.

We can imagine a tree where each node has exactly a children, each doing work on a b^{th} of its size.

Generally, on each level i, there are a^i nodes; each node does $f(\frac{n}{b^i})$ work. There are $\log_b(n) + 1$ levels in total.

There are $n^{\log_b(a)}$ leaves, each doing $\Theta(1)$ work; in total, the leaves do $\Theta(n^{\log_b(a)})$ work.

All the other nodes together do $\sum_{j=0}^{\log_b(n)-1} a^j f(\frac{n}{b^j})$ work.

Thus $T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(\frac{n}{b^j})$.

We have broken the total amount of work in the tree into two parts:

- the work done by the inner nodes: this is the work needed to combine the solutions of the a subproblems
- the work done by the leaves: the important thing here is the number of leaves[‡], i.e. how many subproblems there are

[†] There are other, more advanced methods, of dealing with recurrence relations, but they are out of the scope of this course.
[‡] Because each leaf does a constant amount of work.

Let us call these two parts: $I(n) = \sum_{j=0}^{\log_b(n)-1} a^j f(\frac{n}{b^j})$ and $L(n) = \Theta(n^{\log_b(a)})$.

Our aim is to get a *tight asymptotic bound* that fits $T(n)$; i.e. a Θ notation. Because $T(n)$ is the sum of two terms, the answer will vary depending on the relationship between the growth of these two terms.

There are three possible outcomes of comparing $I(n)$ to $L(n)$; think of these outcomes symbolically as: $<$, $=$, $>$. But we will **not** be comparing numbers, but *asymptotic growths*.

The recursion-tree then has one of the following possible structures:

1. **leaf-heavy**: the leaves do more work than the inner nodes[§]

   The number of subproblems is greater than the amount of work to recombine solutions.

2. **balanced**: the work done by the leaves is the same as that done by the inner nodes.

   The number of subproblems is the same as the amount of work done to recombine solutions.

3. **root-heavy**: the root does more work than the rest of the tree[††]

   The amount of work to recombine solutions is greater than the number of subproblems.

The *emphasized* sentences are interpretations of the tree structure with regards to the nature of the recursive solution characterized by $T(n)$.

The master method offers us an answer for each of these three cases; all we need to do is identify which case applies to our current tree.

In order to prove how this works, we need to formalize the categories above.

Let $c = \log_b(a)$. Note that there are n^c leaves in the tree. The work done by a node on an input of size n is $f(n)$. The work done by the leaves of the tree starting at this node is $\Theta(n^c)$.

We have:

1. **leaf-heavy:** $f(n) \in O(n^d)$, $d < c$

   Naturally, it follows that the complexity of the entire tree is the complexity of the work done by the leaves; there are $\Theta(n^c)$ leaves, each doing a constant amount of work. So the answer is $\Theta(n^c)$.

2. **balanced:** $f(n) \in \Theta(n^c)$

   There are $\Theta(\log(n))$ levels of the tree, each doing $\Theta(n^c)$ work, so the answer is $\Theta(n^c \log(n))$.

3. **root-heavy:** $\exists k < 1$, s.t. $af\left(\frac{n}{b}\right) \leq kf(n)$

   The condition here is a bit more complex; the left-hand side of $\leq$ is the work done by the entire subtree, minus the root. On the right-hand side, we have the work done by the root, constrained by a subunitary number.

   It is often called *"the regularity condition"*.

   If the work done by the rest of the subtree is strictly less than that done by the root, then the work done by the root characterizes the whole tree; the answer is $\Theta(f(n))$.

   Note that there is an asymmetry between this case and the two previous ones; one might expect here a condition that $f(n) \in \Omega(n^d)$, $d > c$.

   This statement can indeed be *proved* given the regularity condition, but it is not sufficient on its own to guarantee that the tree is root-heavy.

   However, if $f(n) = n^t$, for some $t > 0$ then $f(n) \in \Omega(n^d)$, $d > c$ is sufficient; the regularity condition becomes a consequence of these facts (i.e., it can be proved). In practice, many useful recurrence relations have an $f(n)$ that is polynomial.

[§] Actually, for each inner node, it's true that its children do more work.

[††] Actually, each inner node does more work than its subtree.

We will now rigorously prove the answers mentioned above.

Remember that $T(n) = I(n) + L(n)$.

Notice that the work done in the leaves $L(n)$ does not feature an $f(n)$ term; so it's the same in all three cases, which is why we will focus mostly on $I(n) = \sum_{j=0}^{\log_b(n)-1} a^j f\left(\frac{n}{b^j}\right)$.

1.1 Proof for leaf-heavy trees

We can replace $f\left(\frac{n}{b^j}\right)$ with $O\left(\left(\frac{n}{b^j}\right)^d\right)$. We end up with $O\left(\sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{n}{b^j}\right)^d\right)$[‡‡].

We can factor n^d and obtain: $O\left(n^d \sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{1}{b^j}\right)^d\right)$.

Remember that d is a number smaller than $c = \log_b(a)$; thus there exists a number $\epsilon > 0$ such that $d = \log_b(a) - \epsilon$. Making this substitution in our expression we get:

$$\begin{aligned} n^d \sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{1}{b^j}\right)^{\log_b(a)-\epsilon} &= n^d \sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{b^\epsilon}{b^{\log_b(a)}}\right)^j \\ &= n^d \sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{b^\epsilon}{a}\right)^j \\ &= n^d \sum_{j=0}^{\log_b(n)-1} b^{\epsilon j}. \end{aligned}$$

We have n^d multiplied by the sum of a geometric series of ratio b^ϵ, whose solution is:

$$\frac{b^{\epsilon \log_b(n)} - 1}{b^\epsilon - 1} = \frac{n^\epsilon - 1}{b^\epsilon - 1}.$$

Replacing n^d with $n^{\log_b(a)-\epsilon}$ we get:

$$\begin{aligned} n^{\log_b(a)-\epsilon} \cdot \frac{n^\epsilon - 1}{b^\epsilon - 1} &= \frac{n^{\log_b(a)} - n^{\log_b(a)-\epsilon}}{b^\epsilon - 1} \\ &= \frac{1}{b^\epsilon - 1} n^{\log_b(a)} - \frac{1}{b^\epsilon - 1} n^{\log_b(a)-\epsilon} \end{aligned}$$

$\frac{1}{b^\epsilon - 1}$ is a constant; also notice that the term we're subtracting is in $o(n^{\log_b(a)})$ (because $\epsilon > 0$). We obtain $O\left(n^{\log_b(a)}\right)$.

So $T(n) = \Theta(n^{\log_b(a)}) + O(n^{\log_b(a)}) = \Theta(n^{\log_b(a)})$.

[‡‡] We can "pull" the O notation on the outside, because of its definition

1.2 Proof for balanced trees

We can replace $f\left(\frac{n}{b^j}\right)$ with $\Theta\left(\left(\frac{n}{b^j}\right)^c\right)$. We end up with $\Theta\left(\sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{n}{b^j}\right)^c\right)$.

As before, we factor n^c and get: $\Theta\left(n^c \sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{1}{b^j}\right)^c\right)$

$$\begin{aligned} n^{\log_b(a)} \sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{1}{b^j}\right)^{\log_b(a)} &= n^{\log_b(a)} \sum_{j=0}^{\log_b(n)-1} a^j \left(\frac{1}{a^j}\right) \\ &= n^{\log_b(a)} \sum_{j=0}^{\log_b(n)-1} 1 \\ &= n^{\log_b(a)} \log_b(n) \end{aligned}$$

So $T(n) = \Theta(n^{\log_b(a)}) + \Theta(n^{\log_b(a)} \log_b(n)) = \Theta(n^{\log_b(a)} \log(n))$.[§§]

1.3 Proof for root-heavy trees

Remember that this case requires the regularity condition: $\exists k < 1$, s.t. $af\left(\frac{n}{b}\right) \leq kf(n)$.

Iterating over this j times we get:

$$\begin{aligned} a^j f\left(\frac{n}{b^j}\right) \leq k^j f(n) = & \sum_{j=0}^{\log_b(n)-1} a^j f\left(\frac{n}{b^j}\right) \\ \leq & \sum_{j=0}^{\log_b(n)-1} k^j f(n) \\ \leq & f(n) \sum_{j=0}^{\infty} k^j \quad \text{(we do this to get rid of } k \text{ in the numerator, remember that } k < 1) \\ = & f(n) \frac{1}{1-k}. \end{aligned}$$

This means that $I(n) \in O(f(n))$. But the first term of $I(n)$ (when $j = 0$) is $f(n)$ itself, and all the other terms are positive; so $I(n) \in \Omega(f(n))$. These two statements together amount to $I(n) \in \Theta(f(n))$.

Now let us prove that the regularity condition indeed implies that $\exists \epsilon > 0$, s.t. $f(n) \in \Omega\left(n^{\log_b(a)+\epsilon}\right)$.

From the definition of Ω, we need two constants, c_0 and n_0, such that $\forall n \geq n_0$, $0 \leq f(n) \leq c_0 n^{\log_b(a)+\epsilon}$.

For now, we are working under the assumption that n is a power of b; so is n_0.

Then there is an integer s such that $n = b^s n_0$; so $s = \log_b\left(\frac{n}{n_0}\right)$.

From the regularity condition we get:

$$f(n) \geq \frac{a}{c} f\left(\frac{n}{b}\right) \geq \left(\frac{a}{c}\right)^2 f\left(\frac{n}{b^2}\right) \geq \cdots \geq \left(\frac{a}{c}\right)^s f(n_0)$$

[§§] Remember that the base of the logarithm is irrelevant, because changing it only changes the value by a constant factor.

Where:

$$\left(\frac{a}{c}\right)^s = \left(\frac{a}{c}\right)^{\log_b \frac{n}{n_0}} = \left(\frac{n}{n_0}\right)^{\log_b \frac{a}{c}} = \left(\frac{1}{n_0}\right)^{\log_b(a)-\log_b(c)} n^{\log_b(a)-\log_b(c)}$$

Going back to our inequality $f(n) \geq \left(\frac{a}{c}\right)^s f(n_0)$, we get:

$$f(n) \geq \left(\frac{1}{n_0}\right)^{\log_b(a)-\log_b(c)} n^{\log_b(a)-\log_b(c)} f(n_0)$$

Let $\epsilon = -\log_b(c)$; notice that both $\left(\frac{1}{n_0}\right)^{\log_b(a)-\log_b(c)}$ and $f(n_0)$ are constants; let's call their product c_0. We get:

$$f(n) \geq c_0 n^{\log_b(a)+\epsilon}.$$

So $f(n) \in \Omega(n^{\log_b(a)+\epsilon})$.

Going back to our recurrence relation, this means that: $T(n) = \Theta(n^{\log_b(a)}) + \Theta(f(n)) = \Theta(f(n))$.

2 Dealing with numbers that are not powers of 2

We will now extend our proof to all values of n.

We want to obtain a lower bound for:

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

and an upper bound for:

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

The two proofs are very similar, so we will only focus on the lower bound.

We obtain the following sequence describing the amount of work on each level of the recursion tree:

$$n, \left\lceil \frac{n}{b} \right\rceil, \left\lceil \frac{\left\lceil \frac{n}{b} \right\rceil}{b} \right\rceil, \ldots$$

$$n_j = \begin{cases} n & j = 0, \\ \left\lceil \frac{n_{j-1}}{b} \right\rceil & j > 0. \end{cases}$$

The amount of work done by each node is $f(n_j)$; to find the number of levels in the tree, we need to find the level j at which $f(n_j)$ is a constant; but f is an arbitrary function, so we must find when n_j is a constant. Knowing that $\lceil n \rceil \leq n + 1$, we have the following sequence of inequalities:

$$n_0 \leq n$$

$$n_1 \leq \frac{n}{b} + 1$$

$$n_2 \leq \frac{n}{b^2} + \frac{n}{b} + 1$$

$$\dots$$

$$n_j \quad \leq \quad \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} \quad < \quad \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} \quad = \quad \frac{n}{b^j} + \frac{b}{b-1}$$

For $j = \lfloor \log_b(n) \rfloor$:

$$\begin{aligned} n_{\lfloor \log_b(n) \rfloor} &< \frac{n}{b^{\lfloor \log_b(n) \rfloor}} + \frac{b}{b-1} \\ &< \frac{n}{b^{\log_b(n)-1}} + \frac{b}{b-1} \\ &= \frac{n}{\frac{n}{b}} + \frac{b}{b-1} \\ &= b + \frac{b}{b-1} \end{aligned}$$

We get a constant at level $\lfloor \log_b(n) \rfloor$: this is the height of our tree.

$$I(n) = \sum_{j=0}^{\lfloor \log_b(n) \rfloor - 1} a^j f(n_j)$$

Now we only need to redo the calculations for the three cases of the master theorem using this new definition of $I(n)$.

3 Summary – The Master Method

After we have proven all these, we can think of the results as a "cookbook"; for each new recurrence of the appropriate form, we just need to put in the work to figure out what type of strucutre our recursion tree has (leaf-heavy, balanced, recursion-heavy), then lookup in the small "table" below to find the correct answer:

1. $f(n) \in O(n^d), d < c \Rightarrow T(n) \in \Theta(n^c)$
2. $f(n) \in \Theta(n^c \log^k(n)), k \geq 0 \Rightarrow T(n) \in \Theta(n^c \log^{k+1}(n))$
3. $f(n) \in \Omega(n^d), d > c \wedge (\exists k > 1, \text{s. t.} af(\frac{n}{b}) \leq kf(n) \Rightarrow T(n) \in \Theta(f(n))$

4 References and further reading

The master theorem was first published by Jon Louis Bentley, Dorothea Haken and James B. Saxe in their 1980 article "A General Method for Solving Divide-and-Conquer Recurrences" [1].

It was popularized and named in the textbook "Introduction to Algorithms" [2] by Cormen, Leiserson, Rivest and Stein. Much of the treatment of recurrences in these notes is inspired and adapted from section 4 "Divide-and-Conquer" of the first chapter of the textbook.

Another good presentation of the theorem and its proof can be found in Tim Roughgarden's "Algorithms Illuminated – Part 1: The Basics" [3], chapter 4 "The Master Method"; the textbook serves as the basis for his algorithms course at Stanford, available online on coursera[‖].

In 1998, Mohamad Akra and Louay Bazzi developed a more generic method of determining the complexity of recurrence relations [4]. This is now known as "the Akra-Bazzi method". It can address recurrences of the form[**]:

$$T(x) = \sum_{i=1}^{k} a_i T(b_i x + h_i(x)) \text{ for } x \geq x_0 + g(x)$$

Bibliography

[1] Jon Louis Bentley, Dorothea Haken, and James B Saxe. "A general method for solving divide-and-conquer recurrences". In: *ACM SIGACT News* 12.3 (1980), pp. 36–44.

[2] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 1990.

[3] Tim Roughgarden. *Algorithms illuminated*. Soundlikeyourself publishing, 2022.

[4] Mohamad Akra and Louay Bazzi. "On the solution of linear recurrence equations". In: *Computational Optimization and Applications* 10.2 (1998), pp. 195–210.

[‖] `https://www.coursera.org/learn/algorithms-divide-conquer`

[**] This formula is just to illustrate that it covers a superset of recurrences covered by the master method; the details of what each term is and the restrictions on their relationships are not obvious from this context.