The Master Theorem

- Preliminaries (supporting concepts)
- The theorem
- The proof
- The theorem - simplified version
- Examples
- Moving further - Akra-Bazzi (just definition)

Preliminaries

Assumptions

All functions appearing in the definitions and proofs are assumed to be real and asymptotically positive (domain and codomain are R, and for all sufficiently large elements from the domain, all the function values are positive). The formal definition:

$$\texttt{Given f : R} \rightarrow \texttt{R, } \exists\, n_0 \texttt{ so that, } \forall \texttt{n} > n_0\texttt{, f(n) > 0}$$

Sometimes I will use the abbreviation a.p. to emphasize a function is asymptotically positive, especially when this property will be required for a proof.

Complexity Measurements

Algorithm complexity is a measure of the running time of an algorithm. A few ways to measure this complexity:
- Big O (read Big Oh)
- Ω (read Omega)
- Θ (read Theta)
- Asymptotic equality of functions ($\sim$)

Big O

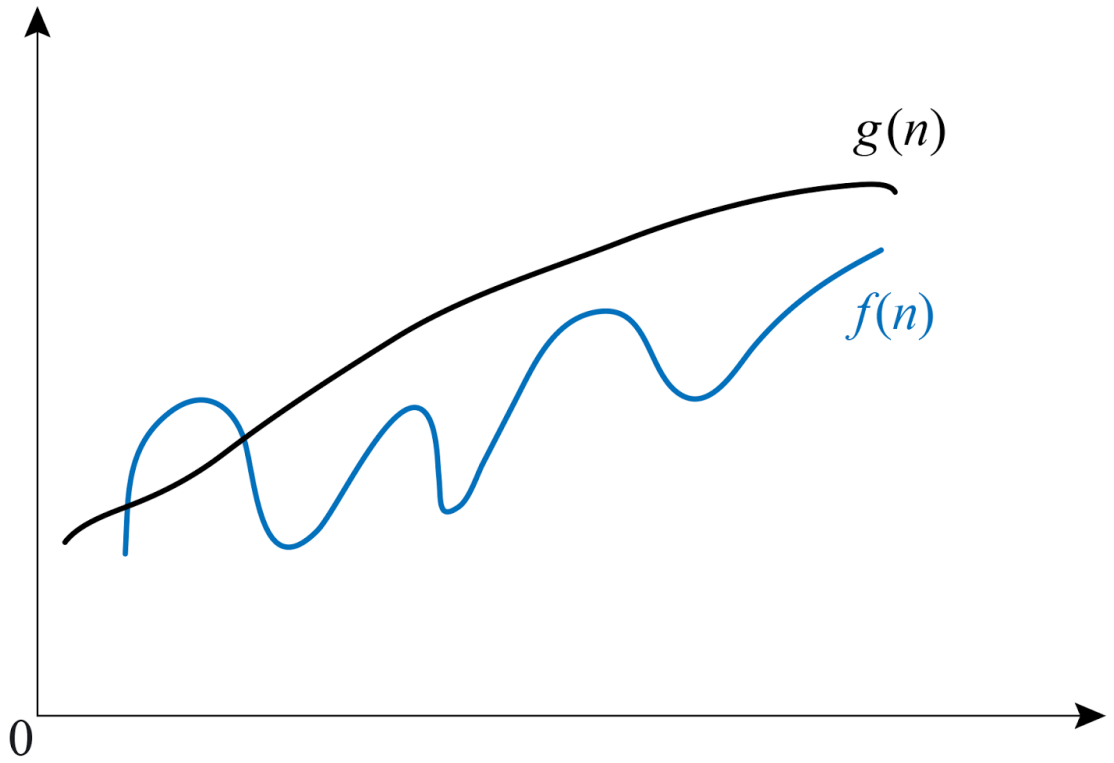
Given a function, finding its corresponding Big O is basically searching for the upper bound. The formal definition:

Given $g : R \to R$ asymptotically positive, O(g(x)) is this *set of functions* such that:

$$O(g(x)) = \{f : R \to R \text{ a.p.} \mid \exists\, c > 0,\ x_0 > 0 \text{ constants so that, } \forall x \geq x_0,\ \ 0 \leq f(x) \leq cg(x)\}$$

We say that $f(x)$ is $O(g(x))$ if $f(x) \in O(g(x))$. The simplified and ubiquitous notation is $f(x) = O(g(x))$. Plus, in practice we are given a specific f and we're searching for g. So we say/write:

$$f(x) = O(g(x)) \text{ if } \exists\, c > 0,\ x_0 > 0 \text{ constants such that, } \forall x \geq x_0,\ \ 0 \leq f(x) \leq cg(x).$$



$f(n) = O(g(n))$

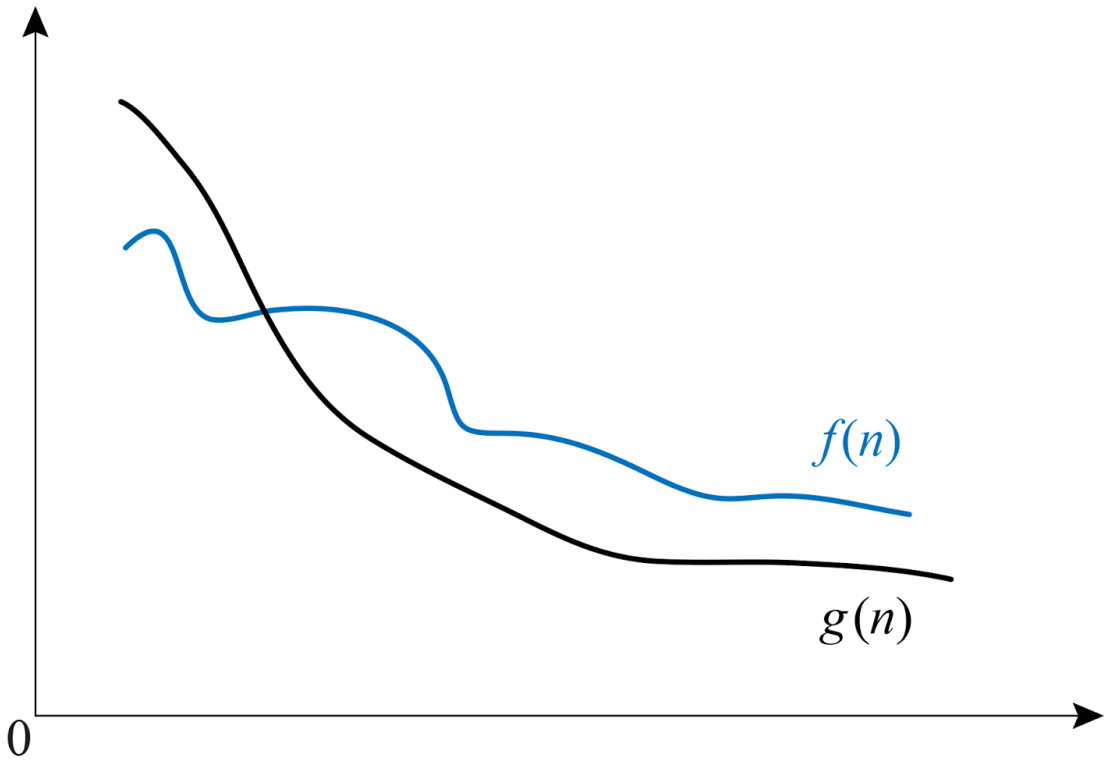
Ω

Formal definition: given g real function, a.p.,

$$\Omega(g(x)) = \{f : R \to R \text{ a.p.} \mid \exists\, c > 0,\ x_0 > 0 \text{ constants so that, } \forall x \geq x_0,\ 0 \leq cg(x) \leq f(x)\}$$

Again we simplify notations and we use this definition (which implies g is the lower bound):

$$f(x) = \Omega(g(x)) \text{ if } \exists\, c > 0,\ x_0 > 0 \text{ constants so that, } \forall x \geq x_0,\ 0 \leq cg(x) \leq f(x).$$



$f(n) = \Omega(g(n))$

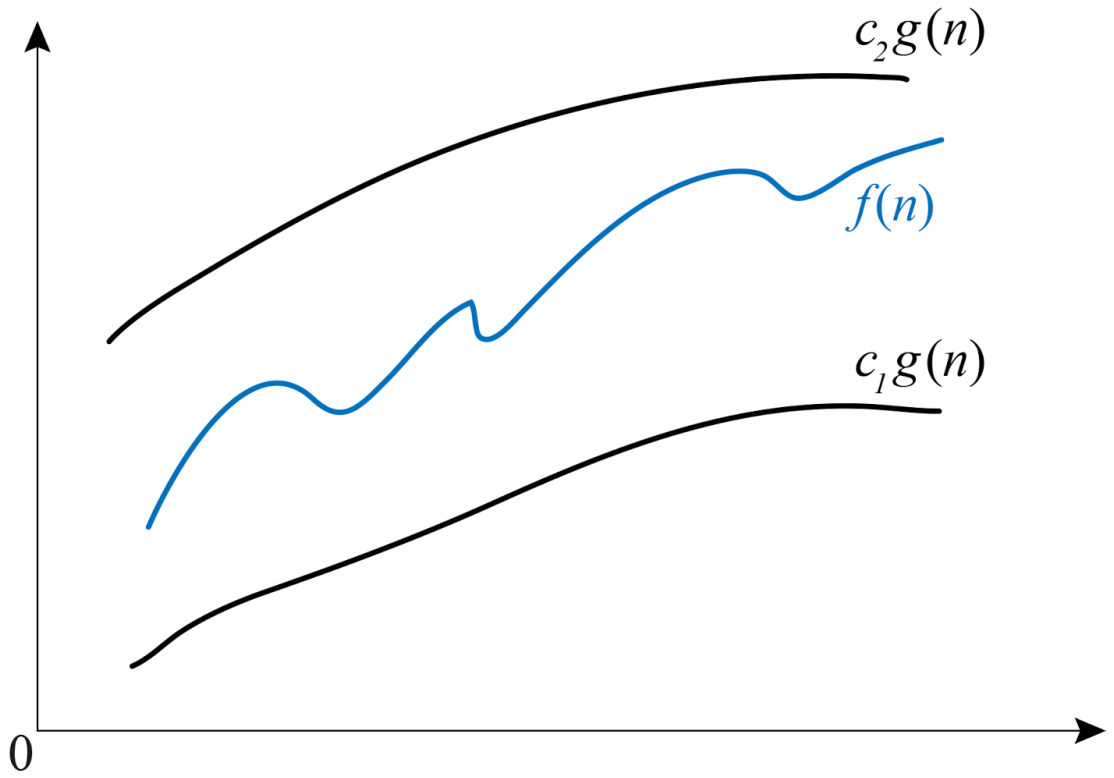
Θ

Big O and Ω do not accurately capture growth rate. We use Θ and ~ for capturing growth rate.

$$\Theta(g(x)) = \{f : R \to R \text{ a.p.} \mid \exists\ c_1 > 0, c_2 > 0, x_0 > 0 \text{ constants such that,}$$
$$\forall x \geq x_0, \quad 0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x)\ \}$$

Simplified version:

$$f(x) = \Theta(g(x)) \text{ if } \exists\ c_1 > 0,\ c_2 > 0,\ x_0 > 0 \text{ constants such that,}$$
$$\forall x \geq x_0,\ 0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x)\ .$$



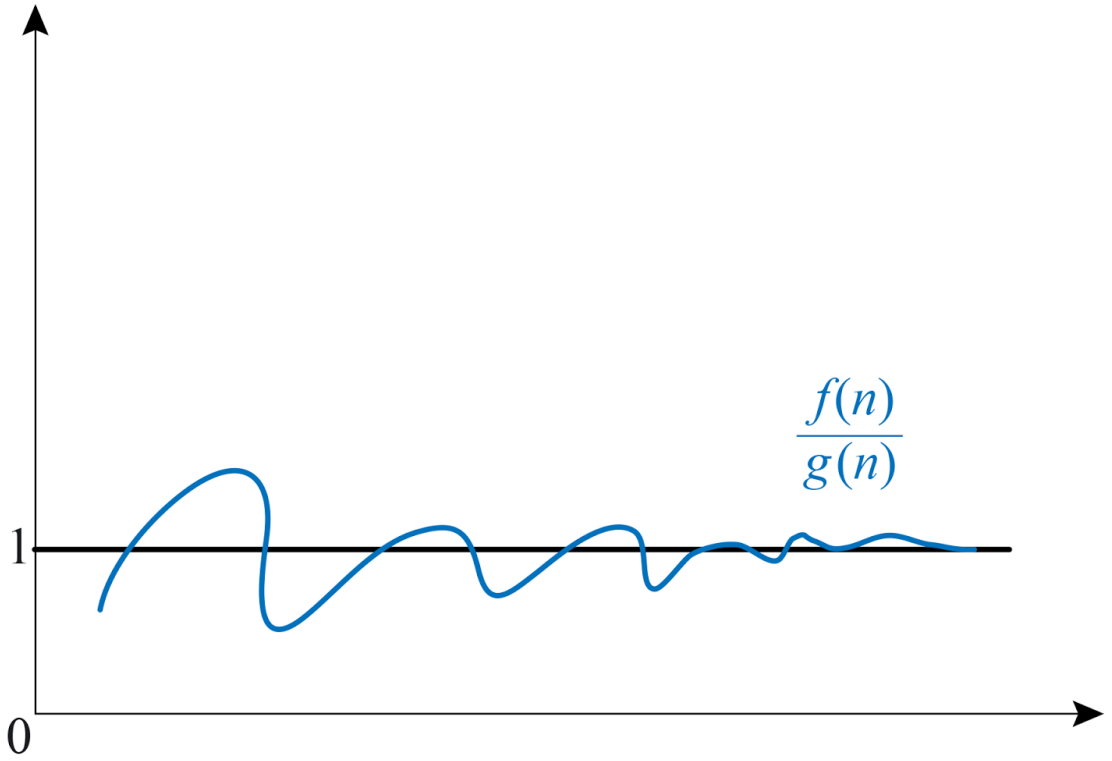
$f(n) = \Theta(g(n))$

Theorem: $f(x) = \Theta(g(x))$ *iif* $f(x) = O(g(x))$ *and* $f(x) = \Omega(g(x))$.

f~g

Definition: two real and a.p. functions f, g are said to be asymptotically equal if

$$\forall \varepsilon > 0,\ \exists\, x_0 > 0 \ so\ that\ \left|\frac{f(x)}{g(x)} - 1\right| < \varepsilon,\ \forall x > x_0$$



f~g

Divide And Conquer

Definition: programming technique where the problem is divided into subproblems whose results are combined to give the solution. The algorithms consist in three steps called: divide (where we split the problem into subproblems), conquer (we get back the results of the subproblems) and combine/merge (we put together the results of the subproblems).

Divide and conquer algorithms are usually described by a recurrence relation of this form:

$$T(n) = a_i T(\tfrac{n}{b_i}) \ + \ldots \ + a_1 T(\tfrac{n}{b_1}) + f(n)$$

Here we divide the problem into i subproblems, each one executed a_i times, each with an entry data with size $\frac{n}{b_i}$, and it costs f(n) to combine the results of these subproblems. Note I am not mentioning the base case in the recurrence relation (when the problem can not be divided anymore).

Example: the recurrence relation for merge sort is: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ and the base cases are constant times (sorting an array of one or two elements).

The Master Theorem
(the complicated version!)

Let $a \geq 1$ and $b > 1$ constants, f an asymptotically positive function, and T be a recurrence defined on non negative integers with $T(n) = aT(\frac{n}{b}) + f(n)$ and with base cases of $\Theta(1)$. Then, $T(n)$ has the following asymptotic bounds:

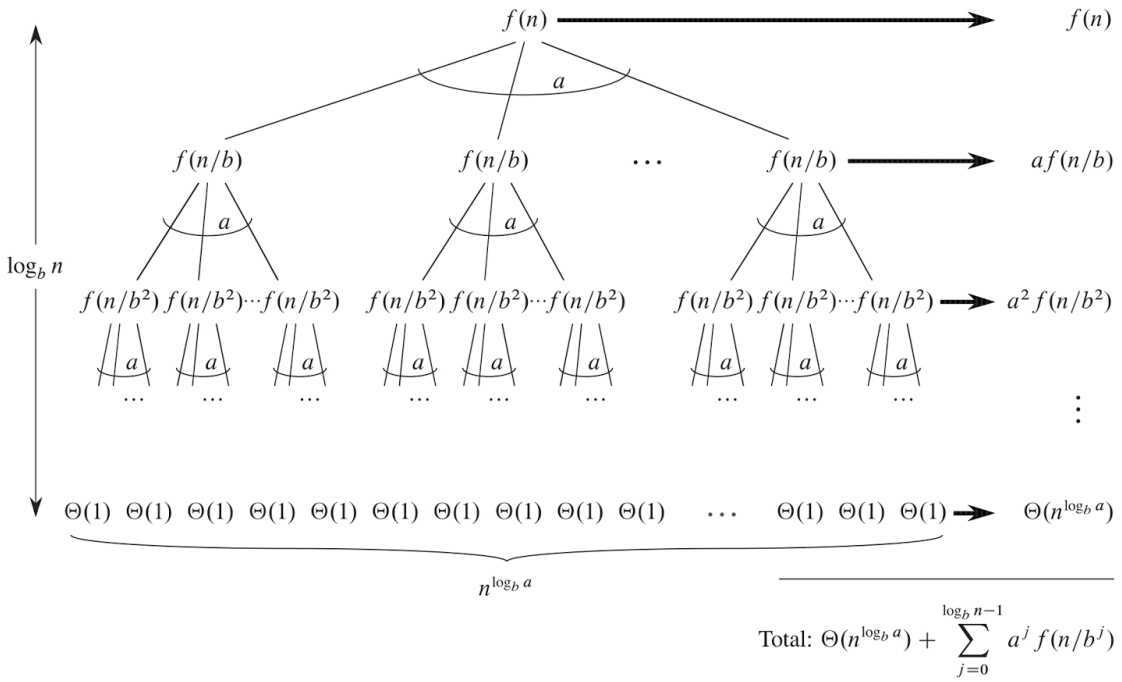
- $If\ \exists\ \varepsilon > 0\ such\ that\ f(n) = O(n^{log_b a - \varepsilon}),\ then\ T(n) = \Theta(n^{log_b a})$
- $If\ f(n) = \Theta(n^{log_b a}),\ then\ T(n) = \Theta(lg(n) * n^{log_b a})$
- $If\ \exists\ \varepsilon > 0, 0 < c < 1,\ constants, such\ that\ f(n) = \Omega(n^{log_b a + \varepsilon})\ and\ af(\frac{n}{b}) < cf(n)\ for\ all\ sufficiently\ large\ n,\quad then\ T(n) = \Theta(f(n))$

The theorem still holds if we replace $\frac{n}{b}$ with $\left[\frac{n}{b}\right]$ (where this is either the floor or ceiling function)

Few observations:

- The theorem does not cover all possible cases (for case 1, when f is asymptotically smaller but not polynomially smaller than $n^{log_b a}$, or in case 3, when f is bigger but not polynomially bigger and or the regularity condition is not met)
- There is the 'battle' between f(n) and $n^{log_b a}$... and what does $n^{log_b a}$ represent in the recursion tree?

Proof with simplifying assumption that n is a power of b.



We notice $a^{log_b n} = n^{log_b a}$ is the number of leaves, $log_b n$ is the depth of the tree and we can write

$$T(n) = a^{log_b n} * \Theta(1) + \sum_{j=0}^{log_b n - 1} a^i f(\tfrac{n}{b^i}) = n^{log_b a} * \Theta(1) + \sum_{j=0}^{log_b n - 1} a^i f(\tfrac{n}{b^i}) = \Theta(n^{log_b a}) + \sum_{j=0}^{log_b n - 1} a^i f(\tfrac{n}{b^i})$$

We observe the first term is the cost of the leaves and the second is the cost of levels in the tree (dividing and combining the results of the subproblems). So next we have to determine the asymptotic bounds for the second term of the equality.

Lemma. Let f(n) a.p. be a function defined over exact powers of n, g(n) be a function defined on integer powers of b,

$a \geq 1, b > 1$ *and* $g(n) = \sum_{j=0}^{log_b n - 1} a^i f(\frac{n}{b^i})$ .Then, the following asymptotic bounds hold for g:

- *If* $\exists\ \varepsilon > 0$ *so that* $f(n) = O(n^{log_b a - \varepsilon})$, *then* $g(n) = O(n^{log_b a})$
- *If* $f(n) = \Theta(n^{log_b a})$, *then* $g(n) = \Theta(lg(n) * n^{log_b a})$
- *If* $\exists\ 0 < c < 1$, *so that* $af(\frac{n}{b}) \leq cf(n)$ *for all sufficiently large* n, *then* $g(n) = \Theta(f(n))$

Proof

Case 1.

$$f(n) = O(n^{log_b a - \varepsilon}) \Rightarrow f(\frac{n}{b^i}) = O((\frac{n}{b^i})^{log_b a - \varepsilon}) \Rightarrow g(n) = O(\sum_{i=0}^{log_b n - 1} a^i (\frac{n}{b^i})^{log_b a - \varepsilon})$$

$$\sum_{i=0}^{log_b n - 1} a^i (\frac{n}{b^i})^{log_b a - \varepsilon} = \sum_{i=0}^{log_b n - 1} a^i \frac{n^{log_b a - \varepsilon}}{b^{i(log_b a - \varepsilon)}} = n^{log_b a - \varepsilon} \sum_{i=0}^{log_b n - 1} (\frac{ab^\varepsilon}{b^{log_b a}})^i = n^{log_b a - \varepsilon} \sum_{i=0}^{log_b n - 1} (b^\varepsilon)^i = n^{log_b a - \varepsilon} \left(\frac{b^{\varepsilon log_b n} - 1}{b^\varepsilon - 1}\right) = n^{log_b a - \varepsilon} \left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right)$$

Because b *and* ε are constants, $O\left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right) = O(n^\varepsilon)$ and O is the upper bound and all factors are positive,

$O(n^{log_b a - \varepsilon} \left(\frac{n^\varepsilon - 1}{b^\varepsilon - 1}\right)) = O(n^{log_b a})$, or $O(g(n)) = O(n^{log_b a})$

Case 2.

$$f(n) = \Theta(n^{log_b a}) \Rightarrow f(\frac{n}{b^i}) = \Theta((\frac{n}{b^i})^{log_b a}) \Rightarrow g(n) = \Theta(\sum_{i=0}^{log_b n - 1} a^i (\frac{n}{b^i})^{log_b a})$$

$$\sum_{i=0}^{log_b n - 1} a^i (\frac{n}{b^i})^{log_b a} = n^{log_b a} \sum_{i=0}^{log_b n - 1} (\frac{a}{b^{log_b a}})^i = n^{log_b a} \sum_{i=0}^{log_b n - 1} 1^i = n^{log_b a} log_b n$$

From these two we deduce $g(n) = \Theta(n^{log_b a} log_b n) = \Theta(n^{log_b a} lg(n))$

Case 3.

From the definition of g and because all summation terms are positive, we deduce $g(n) = \Omega(f(n))$.

Also, $af(\frac{n}{b}) \le cf(n) \Rightarrow f(\frac{n}{b}) \le \frac{c}{a}f(n) \Rightarrow f(\frac{n}{b^2}) \le \frac{c}{a}f(\frac{n}{b}) \le (\frac{c}{a})^2 f(n) \ldots. \Rightarrow f(\frac{n}{b^i}) \le (\frac{c}{a})^i f(n)$ *for sufficiently large* $\frac{n}{b^i}$

or $a^i f(\frac{n}{b^i}) \le c^i f(n)$ *for all sufficiently large* $\frac{n}{b^i}$

Then, considering all small enough terms as $O(1)$,

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^i f(\frac{n}{b^i}) \le \sum_{j=0}^{\log_b n - 1} c^j f(n) + O(1) \le f(n) \sum_{i=0}^{\infty} c^i + O(1) = f(n)\frac{1}{1-c} + O(1) = O(f(n))$$ (*because c is constant and* $0 < c < 1$)

Because $g(n) = \Omega(f(n))$ *and* $g(n) = O(f(n))$, *we can conclude* $g(n) = \Theta(f(n))$ (*cf. previously mentioned theorem*)

Going back to the master theorem,

- for case 1, $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$
- for case 2, $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} lg(n)) = \Theta(n^{\log_b a} lg(n))$
- for case 3, $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$, *because* $f(n) = \Omega(n^{\log_b a + \varepsilon})$

Note that this proof is not complete as we covered just the case n is an exact power of b. Extending the proof to cover all cases is beyond the scope of this presentation.

The Master Method (The simpler version)

This is the definition of The Master Method given by Tim Roughgarden in his course Algorithms: Design and Analysis, Part 1, on Coursera. This one is a lot easier to use in practice, but covers less cases than the 'complete' version.

Given a recurrence of this form, $T(n) \leq aT(\frac{n}{b}) + O(n^d)$, we can conclude:

- $T(n) = O(n^d log n),\ if\ a = b^d$
- $T(n) = O(n^{log_b a}),\ if\ a > b^d$
- $T(n) = O(n^d),\ if\ a < b^d$

The proof is similar to the one for the complicated version.

Examples

- Merge sort: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ (combining the sorted arrays is a linear time algorithm).
  Then $T(n) = \Theta(nlgn)$ (case 2, $n^{log_b a} = n^{log_2 2} = n = \Theta(n)$)

- Binary search: $T(n) = T(\frac{n}{2}) + \Theta(1)$. Then $T(n) = \Theta(lgn)$ (again case 2).

- This recurrence: $T(n) = 2T(\frac{n}{2}) + nlgn$. What case could we apply? Hint: $\lim_{n \to \infty} \frac{logn}{n} = 0$

- Matrix multiplication
  - Division (8 multiplications + 4 additions)
    - Algorithm:

      $\begin{bmatrix} C_{11} C_{12} \end{bmatrix} = \begin{bmatrix} A_{11} A_{12} \end{bmatrix} + [B_{11} B_{12}]$
      $\begin{bmatrix} C_{21} C_{22} \end{bmatrix} \quad \begin{bmatrix} A_{21} A_{22} \end{bmatrix} \quad \begin{bmatrix} B_{21} B_{22} \end{bmatrix}$, where

      $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
      $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
      $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
      $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

    - Recurrence formula:

      $T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$, $f(n) = \Theta(n^2)$, $n^{log_a b} = n^3$, *so we use Case* 1 : $T(n) = \Theta(n^3)$

  - Strassen (7 multiplications + 18 additions and subtractions)
    - Using the matrices from above, let:

      $P = (A_{11} + A_{22})(B_{11} + B_{22})$
      $Q = (A_{21} + A_{22})B_{11}$
      $R = A_{11}(B_{12} - B_{22})$
      $S = A_{22}(B_{21} - B_{11})$
      $T = (A_{11} + A_{12})B_{22}$
      $U = (A_{21} - A_{11})(B_{11} + B_{12})$
      $V = (A_{11} - A_{22})(B_{21} + B_{22})$

      and

      $C_{11} = P + S - T + V$
      $C_{12} = R + T$
      $C_{21} = Q + S$
      $C_{22} = P + R - Q + U$

    - Recurrence formula:

      $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$, $f(n) = \Theta(n^2)$, $n^{log_a b} = n^{log_2 7}$, *Case* 1 : $T(n) = \Theta(n^{2.81})$

Akra - Bazzi method
(a.k.a what to do when all else fails!)

Let T be a recurrence of this form: $T(x) = \left\{ \sum_{i=1}^{k} a_i T(b_i x) + f(x),\ x > x_0 \right\}$

where:
- $k \geq 1$ integer constant, $x \geq 1$
- x_0 constant such that $x_0 \geq \frac{1}{b_i}$ and $x_0 \geq \frac{1}{1-b_i}$ for all $i = \overline{1,k}$
- $T(x) = \Theta(1), 1 \leq x \leq x_0$ (base cases)
- $a_i > 0$ for all $i = \overline{1,k}$
- $0 < b_i < 1$ for all $i = \overline{1,k}$
- f(x) is a nonnegative function that satisfies the polynomial-growth conditions: $\exists c_1, c_2 > 0$ constants such that, for all $x \geq 1$ and for all $i = \overline{1,k}$, and for all u such that $b_i x \leq u \leq x$, we have $c_1 f(x) \leq f(u) \leq c_2 f(x)$

Let p be the unique real number such that $\sum_{i=1}^{k} a_i b_i^p = 1$ (this value always exists).

Then, $T(x) = \Theta(x^p(1 + \int_{u=1}^{x} \frac{f(u)}{u^{p+1}}))$

The difficulty is finding p. Binary search works, choosing an acceptable approximation of ε and the desired number of decimals. Another solution for finding p is applying the Newton Raphson method.