Maths for Computer Science Logs

Denis TRYSTRAM

Logarithms: the scaling functions

Definition

b is the base (positive real number).
$log(x)$ is defined as the inverse of the exponentiation $f(x) = b^x$:
$x = b^{log_b(x)}$

Using this definition and the basic property of the exponential, we can establish most existing properties of the log.

Logarithms: the scaling functions

Definition

b is the base (positive real number).
$log(x)$ is defined as the inverse of the exponentiation $f(x) = b^x$:
$x = b^{log_b(x)}$

Using this definition and the basic property of the exponential, we can establish most existing properties of the log.

- $log_b(x \cdot y) = log_b(x) + log_b(y)$
- $log_b(1) = 0$ is a consequence of this definition, not by convention!
- $\log_a(x) = \log_a(b) \log_b(x)$

Going further

Another useful expression of $n^{\log_a(b)}$

- $n^{\log_a(b)} = b^{\log_a(n)}$

- Draw the shape of the *log* function
- What are the links between the sum of the harmonic series and the *log*?

Geometric interpretation

draw the integral of $1/x$
give the interpretation of the multiplicative rule

Going further

- natural logarithm, base $e = 2.71828...$
- Base 2: the "natural" log of $2 = 0.69314...$
- base 10: $log(10^{-2}) = -2$

Most of them are irrational.
Example: prove that $log(2)$ is irrational, by contradiction:
Assume it is p/q, then $10^{p/q} = 2$, thus $10^p = 2^q$
impossible since the first ends by the digit 0 and the second by $2, 4, 6$ or 8
Using the same argument, $log(3)$ is irrational since the powers of 3 are odds.

Use of logs (in Computer Science)

- space needed for coding integers, base 2

Example of use:
The Master theorem

We study here Maths concepts and results that can be useful for solving problems in algorithmic.
Principle of the divide and conquer paradigm

Motivation

Divide and conquer is a powerful paradigm for designing algorithms
More precisely, for problems whose input can be decomposed

[1] Generally, the k sub-problems are solved by the same method (recursively)

Example of use: The Master theorem

We study here Maths concepts and results that can be useful for solving problems in algorithmic.
Principle of the divide and conquer paradigm

Motivation

Divide and conquer is a powerful paradigm for designing algorithms
More precisely, for problems whose input can be decomposed

Principle for a problem of size n:
If n is "small" enough, we compute the value using any existing method. Otherwise

1. Decompose the problem into k sub-problems of size n_i.
2. Solve the k sub-problems of sizes n_i $(1 \leq i \leq k)$[1]
3. Rebuild the original solution form the k partial solutions.

[1] Generally, the k sub-problems are solved by the same method (recursively)

Recursive tree

Example

Merge sort
Cost: $\Theta(n \cdot log_2(n))$

The method detailed for the analysis of the merge sort can be generalized:

Principle:

Build the tree whose **vertices** are labelled by the cost of the sub-operations and whose **edges** correspond to the partitioning of the sub-problems.

Cost analysis

n is the problem size.
The cost is obtained by solving the following recurrence equation:

- $T(1) \leq$ Cste.
- $T(n) = \sum_{1 \leq i \leq k} T(n_i) + c_1(n) + c_3(n)$

where c_1 et c_3 are the respective costs of the decomposition phase (1) and reconstruction/merge phase (3).

Simplified cost analysis

Practically, most D&C methods consider partitions into a identical sub-problems and with the same size $n_i = \frac{n}{b}$.
Thus, the cost analysis simplifies into:

- $T(1) = 1$
- $T(n) = a.T(\frac{n}{b}) + c(n)$

Simplified cost analysis

Practically, most D&C methods consider partitions into a identical sub-problems and with the same size $n_i = \frac{n}{b}$.
Thus, the cost analysis simplifies into:

- $T(1) = 1$
- $T(n) = a.T(\frac{n}{b}) + c(n)$

for merge sort, $a = b = 2$ and $c(n) = n$

Computing the cost

It is a hard question.
Some technical considerations for simplifying the computations:

- Practically, the size of the instance n does not exactly divide evenly.
- The recomposition phase is simplified asymptotically.
- The border conditions are taken in $\mathcal{O}(1)$ or $\Theta(n)$.

Master Theorem

Cost analysis

$a \geq 1$ et $b > 1$.

- $f(n) = \Theta(1)$ if $n \leq n_0$
- $f(n) = a.f(\frac{n}{b}) + c(n)$ if $n > n_0$

Here is the general formulation for solving this equation:

1. if $c(n) \in \mathcal{O}(n^{log_b a - \epsilon})$ then $f(n) \in \Theta(n^{log_b a})$
2. if $c(n) \in \Theta(n^{log_b a})$ then $f(n) \in \Theta(n^{log_b a} log(n))$
3. if $c(n) \in \Omega(n^{log_b a + \epsilon})$ and if $a.c(n/b) \leq kf(n)$ for some constant $k < 1$ then, $f(n) \in \Theta(c(n))$

where ϵ is a positive real number

Analysis of two simplified cases

Assume n is a perfect power of b (in other words, it is perfectly divisible by b until reaching 1).

- $f(1) = 1$
- $f(n) = a.f(\frac{n}{b}) + c$ (c is a positive constant)

- $f(1) = 1$
- $f(n) = a.f(\frac{n}{b}) + n$

The simplest case

Function f is a simple linear recurrence.
In this case, the solution of f function of n is given by the simplified equation below:

$$\begin{aligned} f(n) &= (1 + \log_b n) \cdot c && \text{if } a = 1 \\ &= \frac{1 - a^{\log_b n}}{1 - a} \cdot c \approx \frac{c}{1 - a} && \text{if } a < 1 \\ &= \frac{a^{\log_b n} - 1}{a - 1} \cdot c && \text{if } a > 1 \end{aligned} \tag{1}$$

proof

- Let write the expansion of the computations by replacing the successive occurrences.
- As soon as we detect a "regular" pattern, we can infer the general form...

$$\begin{aligned} f(n) &= af(n/b) + c \\ &= a\left(af(n/b^2) + c\right) + c \end{aligned}$$

proof

- Let write the expansion of the computations by replacing the successive occurrences.
- As soon as we detect a "regular" pattern, we can infer the general form...

$$\begin{aligned} f(n) &= af(n/b) + c \\ &= a\left(af(n/b^2) + c\right) + c \\ &= a^2 f(n/b^2) + (a+1)c \\ &= a^2\left(af(n/b^3) + c\right) + (a+1)c \end{aligned}$$

proof

- Let write the expansion of the computations by replacing the successive occurrences.
- As soon as we detect a "regular" pattern, we can infer the general form...

$$
\begin{aligned}
f(n) &= af(n/b) + c \\
&= a\left(af(n/b^2) + c\right) + c \\
&= a^2 f(n/b^2) + (a+1)c \\
&= a^2\left(af(n/b^3) + c\right) + (a+1)c \\
&= a^3 f(n/b^3) + (a^2 + a + 1)c \\
&\vdots \quad \vdots \\
&= \left(a^{\log_b n} + \cdots + a^2 + a + 1\right) c
\end{aligned}
\tag{2}
$$

Build a graphical representation for $a = b = 2$

Case of a linear overhead

In this case, the value of f on any argument n is given by

$$f(n) \;=\; a^{\log_b n} f(1) \;+\; \left(\sum_{i=0}^{\log_b(n)-1} (a/b)^i \right) n$$

When $a > b$, the behavior of $f(n)$ is dominated by the first term of this solution:

$$a^{\log_b n} \cdot f(1) \;=\; n^{\log_b a}$$

When $a < b$, the behavior of $f(n)$ is dominated by the second term of this solution:

$$n \cdot \sum_{i=0}^{\log_b(n)-1} (a/b)^i \;=\; \frac{\left(1 \;-\; (a/b)^{\log_b(n)}\right)}{1-(a/b)} \cdot n \;\approx\; \frac{b}{b-a} \cdot n$$

proof

We expose the algebraic pattern created by the recurrence by "unfolding" recurrence.
As before, once we discern this pattern, we jump to the general form (which can be verified via induction).

$$
\begin{aligned}
f(n) &= af(n/b) + n \\
&= a\left(af(n/b^2) + n/b\right) + n \\
&= a^2 f(n/b^2) + (an/b + n) \\
&= a^2\left(af(n/b^3) + n/b^2\right) + (a/b + 1)n \\
&= a^3 f(n/b^3) + (a^2/b^2 + a/b + 1)n \\
&\vdots \\
&= a^{\log_b n} f(1) + \left(\sum_{i=0}^{\log_b(n)-1} (a/b)^i\right) n
\end{aligned}
$$