CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Logarithms in CS

Department of Computer Science
University of Maryland, College Park

Understanding Logarithms

- The **logarithm** is the inverse of exponentiation.
- If $b^x = y$, then $\log_b(y) = x$
- Example:
  - $2^3 = 8 \rightarrow \log_2(8) = 3$
  - $10^4 = 10{,}000 \rightarrow \log_{10}(10{,}000) = 4$
- In words: **"The log tells us how many times to multiply the base by itself to get a number."**
- **Important Properties** (In the Realm of Real Numbers):
  - The **argument** (input to the log) must be **positive** ($x > 0$). Logarithms are **not defined for zero or negative numbers** in the real number system.
  - The **base** of a logarithm **must be positive and not equal to 1**:
  - The **answer** (output of the log) can be **any real number**, but:
  - **For $x > 1$,** $\log_b(x)$ is **positive**.
  - **For $x = 1$,** $\log_b(1) = 0$ because $b^0 = 1$
  - **For $0 < x < 1$,** $\log_b(x)$ is **negative.**
- In computer science, we almost always use base-2 logarithms ($\log_2(X)$ which is CS is known as lg(x)) because of binary representation and the halving strategy used in many algorithms. We also assume the input x is an integer ≥1, as it typically represents the problem size.

Know These Powers of 2—Or Struggle Forever as a Computer Scientist!

Yes, you do have to memorize this table ☹

x	2^n Value	$\log_2(x)$
2^0	1	0
2^1	2	1
2^2	4	2
2^3	8	3
2^4	16	4
2^5	32	5
2^6	64	6
2^7	128	7
2^8	256	8
2^9	512	9
2^{10}	1,024	10
2^{11}	2,048	11
2^{12}	4,096	12
2^{13}	8,192	13
2^{14}	16,384	14
2^{15}	32,768	15
2^{16}	65,536	16

Logarithm Identities (Useful for CS)

- **Product Rule**: $\log_b(xy) = \log_b x + \log_b y$
- Example: $\log_2(16\times4) = \log_2(16) + \log_2(4) = 4 + 2 = 6$

- **Quotient Rule**: : $\log_b(x/y) = \log_b x - \log_b y$
- Example: $\log_2(32/4) = \log_2(32) - \log_2(4) = 5 - 2 = 3$

- **Power Rule**: : $\log_b(x^k) = k\log_b x$
- Example: $\log_2(8^3) = 3\log_2(8) = 3*3 = 9$

Logarithmic Reduction – "Halving Until 1"

- **Why does binary search take O(log n) time?**
  Binary search works by dividing a sorted list in half at each step.
- If the list has **n** elements:
  - After 1 step: at most **n/2** elements remain
  - After 2 steps: at most **n/4** elements remain.
  - After **k** steps: at most **n / 2^k** elements remain.
- The process stops when only one element remains:

$$\frac{n}{2^k} = 1$$

- Solving for **k**:

$$n = 2^k$$
$$k = \lg(n)$$

- Reminder: The **ceiling** of a number is the smallest integer greater than or equal to that number. Take the ceiling of lg(n) if n is not a power of 2 to get whole number of steps.

- Thus, the number of steps required is at most **O(log n)**, making binary search highly efficient compared to linear search, which takes **O(n)** time.

Logarithms in Big-O Notation

- **Logarithms appear when we repeatedly divide a problem into smaller parts**.

- Common logarithmic complexities:
- **Binary search**: O(log(n))
- **Balanced search trees (BST, AVL, Red-Black trees)**: : O(log(n))
- **Heap operations (insert, delete-min)**: : O(log(n))
- **Why ignore the base in Big-O?**
- Change-of-base formula: $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$
- Example: $\log_{10}(n) = \frac{\log_2(n)}{\log_2(10)}$
- Since$1/\log_2(10)$is a **constant**, switching bases **only changes the constant factor**.

- Since **log bases differ by a constant**, Big-O treats them the same:
  - $O(\log_2(n))$, $O(\log_{10}(n))$, O(ln(n)) are all equivalent.

Logarithmic algorithms grow very slowly

- If an algorithm runs in $O(\log_2(n))$ how much slower is it if the input doubles?
- If the input size is n, the running time is $O(\log_2(n))$
- If the input **doubles** (2n), the new running time is: $O(\log_2(2n))$
- Reminder:

Product Rule: $\log_b(xy) = \log_b x + \log_b y$

$$\log_2(2n) = \log_2 2 + \log_2 n = 1 + \log_2 n$$

- **This means the running time increases by only 1 extra step!**
- **Key takeaway:** Logarithmic algorithms grow very slowly compared to linear or exponential ones.