Algorithms
&
Complexity Analysis

Algorithms

- An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.
- Al-Khwārizmī, the Persian astronomer and mathematician
- Arabic treatise, *On Calculation with Hindu Numerals*, in 825 AD
- Translated to Latin in the 12th century as

  Algoritmi de numero Indorum[1],
- Probably meaning "Algoritmi on the numbers of the Indians“
- "Algoritmi" was the translator's rendition of the author's name.
- On misinterpretation *Algoritmi* as a Latin plural became

  "algorithm“

  to mean "calculation method".

Complexity

- What is an "efficient" program?
- How can we measure efficiency?
- The Big O, Big Theta and Big Omega Notation
- Asymptotic Analysis

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size. Generally represented by n, indicating the number of elements to be processed.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use System time to get an accurate measure of the actual running time
- Plot the results

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time and memory space as a function of the input size, n.
- Can take into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
- Preferred notation for specifying programs in an abstract manner

Complexity Analysis of Algorithms

- Two main factors that should be studied to analyze a program’s efficiency are :
  - The time required to execute
    - Time Complexity
  - Amount of computer memory consumed
    - Space complexity

Counting Primitive Operations

Algorithm ***arrayMax***(***A***, ***n***)	# operations
currentMax ← ***A***[0]	1
for ***i*** ← 1 **to** ***n - 1*** **do**	***n*** - 1
if ***A***[*i*] > ***currentMax*** **then**	***n*** - 1
currentMax ← ***A***[***i***]	***n*** - 1
increment counter ***i***	***n*** - 1
return ***currentMax***	1

Time complexity $T(n) = 4n - 2$

Space complexity $S(n) = n+2$

Asymptotic Analysis of f(n)

- Studying function behavior as n takes large values.
- Mathematically, view the patterns in the values of f(n) , as n→ infinity.
- Does not predict or consider exact values of the functions.

How 4n-2 Grows?

- Estimated running time for different values of n:
- n = 10 => 38 steps
- n = 100 => 398 steps
- n = 1,000 => 3998steps
- n = 1,000,000 => 3,999,998 steps
- As n grows, the number of steps grows in *linear* proportion to n for this ***arrayMax***(***A***, ***n***)
- This makes sense since T(n) = 4n-2 is a linear function in n.

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $\boldsymbol{T}(\boldsymbol{n})$ by a constant factor, but
  - Does not alter the growth rate of $\boldsymbol{T}(\boldsymbol{n})$
- The linear growth rate of the running time $\boldsymbol{T}(\boldsymbol{n})$ is an intrinsic property of algorithm ***arrayMax***

Asymptotic analysis

- Linear functions: $f(n) = 5n + 20$
  - For $n > 20$, $f(n) = 5n + 20 <= 5n + n = 6n$
  - As $n \rightarrow$ infinity, $f(n)$ has an upper bound function $g(n)=6n$
- Quadratic functions: $f(n) = 8n^2 + 6n + 3$

$$
\begin{aligned}
f(n) &= 8n^2 + 6n + 3 \\
&< 8n^2 + 6n + n \qquad \text{for } n > 3 \\
&= 8n^2 + 7n \\
&< 8n^2 + n * n \qquad \text{for } n > 7 \\
&= 9n^2
\end{aligned}
$$

  - As $n \rightarrow$ infinity, $f(n)$ has an upper bound function $g(n)=9n^2$

Big Oh notation

- **Definition**: $f(n) = O(g(n))$
- Read as "f of n equals big oh of g of n"
- If and only if, there exist two positive constants c1 and m1 such that

$$f(n) \le c1\ g(n), \qquad \text{for all } n \ge m1$$

- The upper bound function is $c1\ g(n)$
- Since, $f(n) = 5n + 20 \le 6n$, for $n>20$ we conclude $f(n) = O(n)$

Omega notation

- **Definition**: $f(n) = \Omega(g(n))$
- Read as "f of n equals Omega of g of n"
- If and only if, there exist two positive constants c2 and m2 such that

$$f(n) \geq c2\, g(n), \qquad \text{for all } n \geq m2$$

- The lower bound function is $c2\, g(n)$
- Since, $f(n) = 5n + 20 \geq 4n$, for n>20 we conclude $f(n) = \Omega(n)$

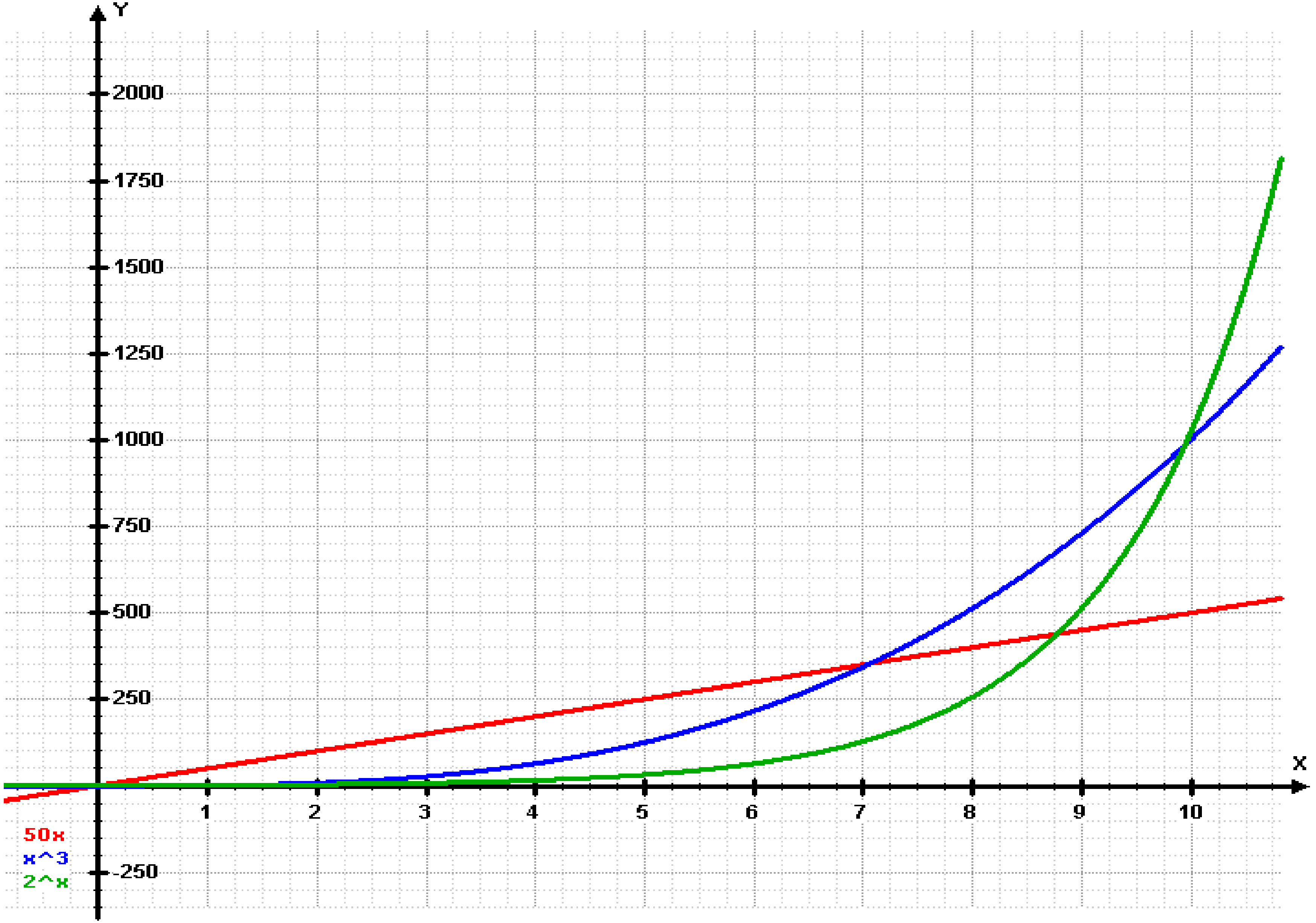
Theta notation

- **Definition**: $f(n) = \Theta(g(n))$
- Read as "f of n equals theta of g of n"
- If and only if, there exist positive constants c1, c2 and m such that

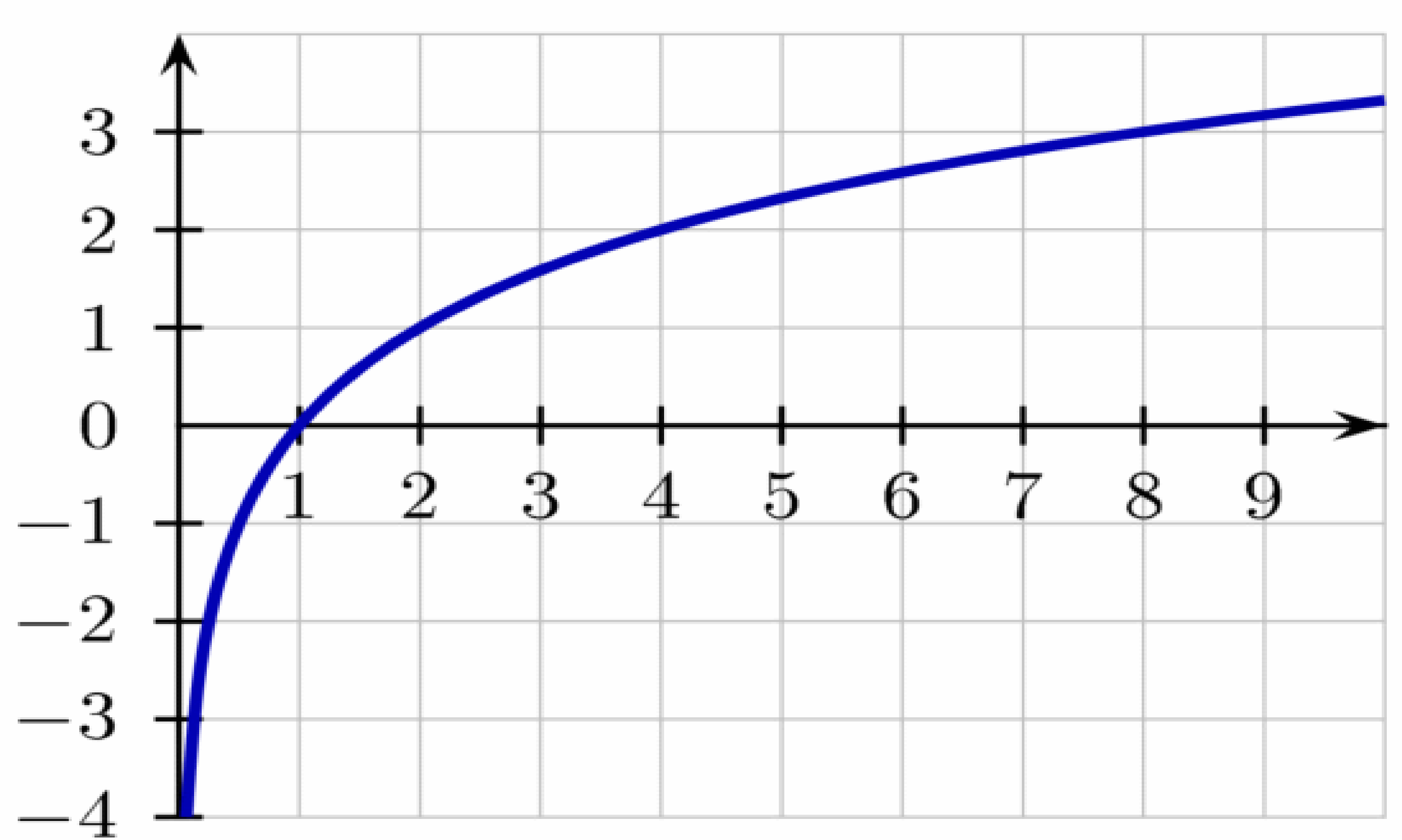
$c1\ g(n) <= f(n) <= c2\ g(n),$ for all $n >= m$

- Indicates f(n) and g(n) grow at the same rate asymptotically
- Since, $4n <= f(n) = 5n + 20 <= 6n$, for $n>20$ we conclude $f(n) = \Theta(n)$

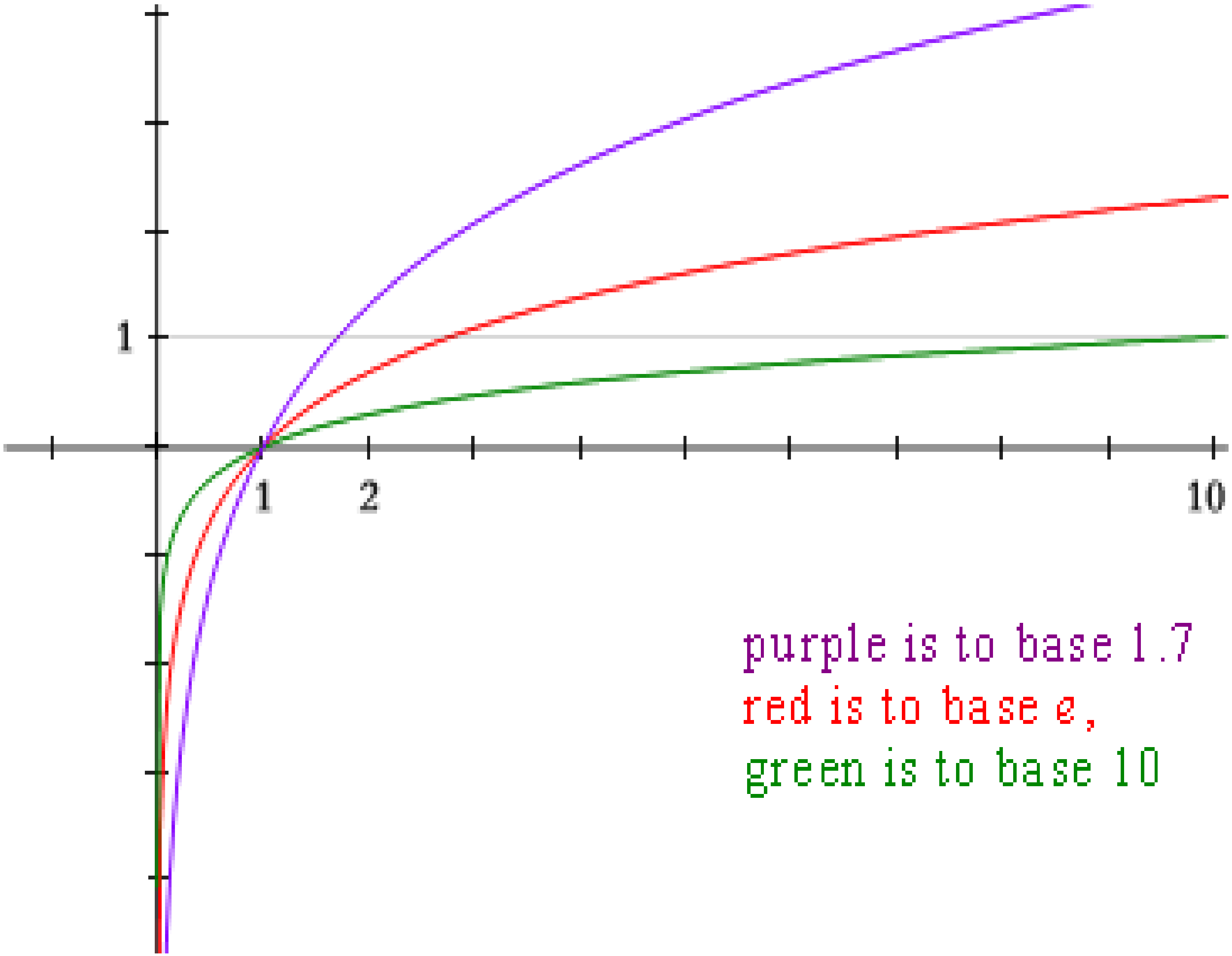
Exponential growth

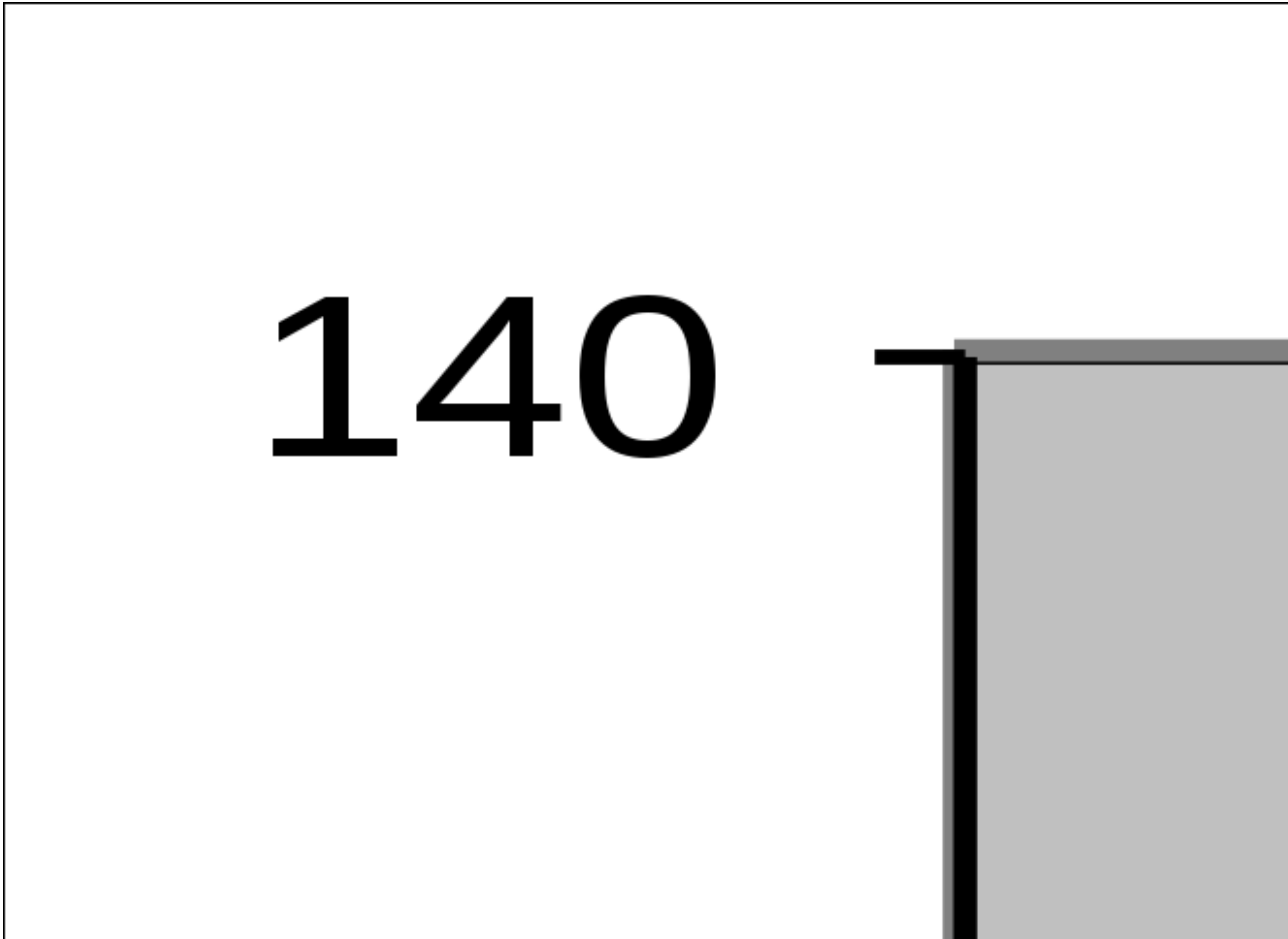


Logarithmic growth



Logarithmic growth rates




140