

CPSC 259: Data Structures and Algorithms for Electrical Engineers

Algorithm Analysis

Textbook References:

- (a) Thareja (first edition) 4.6-4.7
- (b) Thareja (second edition): 2.8 – 2.12

Learning Goals

- Justify which operation(s) we should measure in an algorithm/program in order to estimate its “efficiency”.
- Define the “input size” n for various problems, and determine the effect (in terms of performance) that increasing the value of n has on an algorithm.
- Given a fragment of code, write a formula which measures the number of steps executed, as a function of n .
- Define the notion of Big-O complexity, and explain pictorially what it represents.
- Compute the worst-case asymptotic complexity of an algorithm in terms of its input size n , and express it in Big-O notation.

Learning Goals (cont)

- Compute an appropriate Big-O estimate for a given function $T(n)$.
- Discuss the pros and cons of using best-, worst-, and average-case analysis, when determining the complexity of an algorithm.
- Describe why best-case analysis is rarely relevant and how worst-case analysis may never be encountered in practice.
- Given two or more algorithms, rank them in terms of their time and space complexity.
- [Future units] Give an example of an algorithm/problem for which average-case analysis is more appropriate than worst-case analysis.

Introduction

- What is Algorithm?
 - a clearly specified **set of simple instructions** to be followed to solve a problem
 - Takes a set of values, as input and
 - produces a value, or set of values, as output
 - May be specified
 - In English
 - As a computer program
 - As a pseudo-code
- Data structures
 - Methods of organizing data
- Program = algorithms + data structures

Introduction

- Why need algorithm analysis ?
 - writing a working program is not good enough
 - The program may be inefficient!
 - If the program is run on a large data set, then the running time becomes an issue

Example: Selection Problem

- Given a list of N numbers, determine the k th largest, where $k \leq N$.
- Algorithm 1:
 - (1) Read N numbers into an array
 - (2) Sort the array in decreasing order by some simple algorithm
 - (3) Return the element in position k

Example: Selection Problem...

- Algorithm 2:
 - (1) Read the first k elements into an array and sort them in decreasing order
 - (2) Each remaining element is read one by one
 - If smaller than the k th element, then it is ignored
 - Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.
 - (3) The element in the k th position is returned as the answer.

Example: Selection Problem...

- Which algorithm is better when
 - $N = 100$ and $k = 100$?
 - $N = 100$ and $k = 1$?
- What happens when $N = 1,000,000$ and $k = 500,000$?
- There exist better algorithms

Algorithm Analysis

- We only analyze *correct* algorithms
- An algorithm is correct
 - If, for every input instance, it halts with the correct output
- Incorrect algorithms
 - Might not halt at all on some input instances
 - Might halt with other than the desired answer
- Analyzing an algorithm
 - Predicting the resources that the algorithm requires
 - Resources include
 - Memory
 - Communication bandwidth
 - Computational time (usually most important)

Algorithm Analysis...

- Factors affecting the running time
 - computer
 - compiler
 - algorithm used
 - input to the algorithm
 - The content of the input affects the running time
 - typically, the *input size* (number of items in the input) is the main consideration
 - E.g. sorting problem \Rightarrow the number of items to be sorted
 - E.g. multiply two matrices together \Rightarrow the total number of elements in the two matrices
- Machine model assumed
 - Instructions are executed one after another, with no concurrent operations \Rightarrow Not parallel computers

Efficiency

- Complexity theory addresses the issue of how *efficient* an algorithm is, and in particular, how well an algorithm *scales* as the problem size increases.
- Some **measure of efficiency** is needed to compare one algorithm to another (assuming that both algorithms are correct and produce the same answers). Suggest some ways of how to measure efficiency.
 - Time (How long does it take to run?)
 - Space (How much memory does it take?)
 - Other attributes?
 - Expensive operations, e.g. I/O
 - Elegance, Cleverness
 - Energy, Power
 - Ease of programming, legal issues, etc.

Analyzing Runtime

```
old2 = 1;  
old1 = 1;  
for (i=3; i<n; i++) {  
    result = old2+old1;  
    old1 = old2;  
    old2 = result;  
}
```

How long does this take?

Analyzing Runtime

```
old2 = 1;  
old1 = 1;  
for(i=3; i<n; i++){  
    result = old2+old1;  
    old1 = old2;  
    old2 = result;  
}
```

Wouldn't it be nice if it didn't depend on so many things?

How long does this take?

IT DEPENDS

- What is n ?
- What machine?
- What language?
- What compiler?
- How was it programmed?

Number of Operations

- Let us focus on one complexity measure: the **number of operations** performed by the algorithm on an input of a given **size**.
- What is meant by “number of operations”?
 - # instructions executed
 - # comparisons
- Is the “number of operations” a precise indicator of an algorithm’s running time (time complexity)? Compare a “shift register” instruction to a “move character” instruction, in assembly language.
 - No, some operations are more costly than others
- Is it a fair indicator?
 - Good enough

Analyzing Runtime

```
old2 = 1
old1 = 1
for(i=3; i<n; i++){
    result = old2+old1
    old1 = old2
    old2 = result
}
```

How many operations does this take?

IT DEPENDS

- What is n ?

- Running time is a function of n such as $T(n)$
- This is really nice because the runtime analysis doesn't depend on **hardware** or **subjective conditions** anymore

Input Size

- What is meant by the input size n ? Provide some application-specific examples.
 - Dictionary:
 - # words
 - Restaurant:
 - # customers or # food choices or # employees
 - Airline:
 - # flights or # luggage or # costumers
- We want to express the number of operations performed as a function of the input size n .

Run Time as a Function of **Size of** Input

- But, **which** input?
 - Different inputs of same size have different run times

E.g., what is run time of linear search in a list?

- If the item is the first in the list?
- If it's the last one?
- If it's not in the list at all?

What should we report?

Which Run Time?

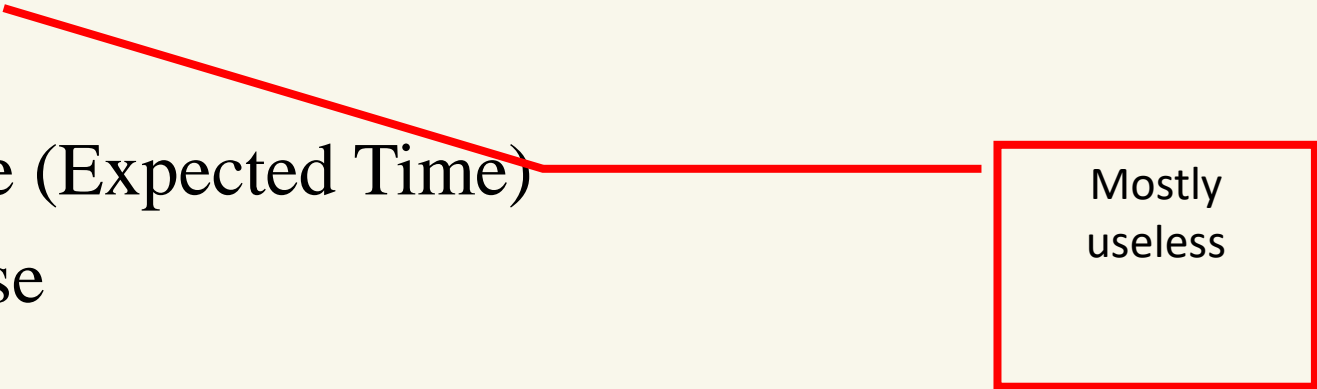
There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.

Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.

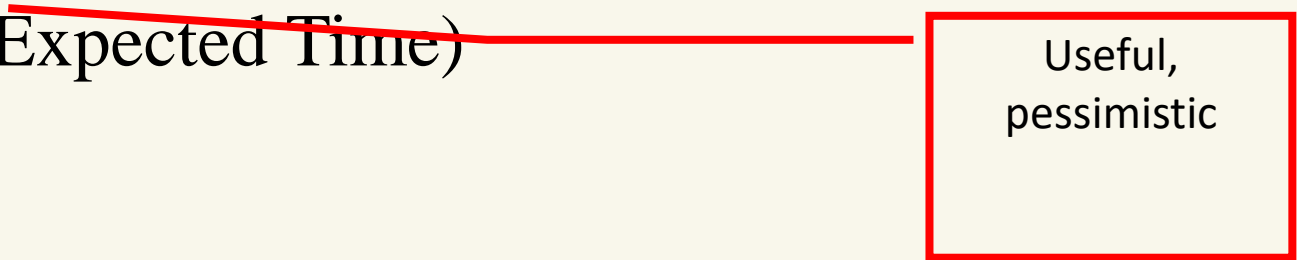


Mostly
useless

Which Run Time?


There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.



Useful,
pessimistic

Which Run Time?



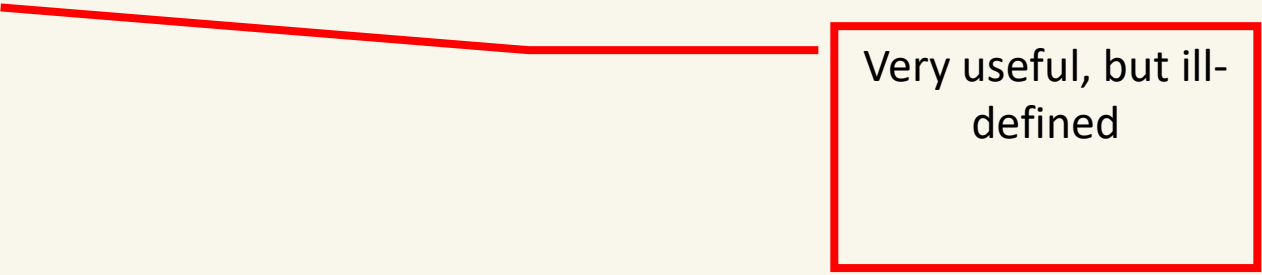
Useful, hard to
do right

- Average Case (Expected Time)
 - Requires a notion of an "average" input to an algorithm, which uses a probability distribution over possible inputs.
 - Allows discriminating among algorithms with the same worst case complexity
 - Classic example: Insertion Sort vs QuickSort

Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- etc.



Very useful, but ill-defined

Scalability!

- What's more important?
 - At $n=5$, plain recursion version is faster.
 - At $n=3500$, complex version is faster.
- Computer science is about solving problems people couldn't solve before. Therefore, the emphasis is almost always on solving the big versions of problems.
- (In computer systems, they always talk about “scalability”, which is the ability of a solution to work when things get really big.)

Asymptotic Analysis

- Asymptotic analysis is analyzing what happens to the run time (or other performance metric) as the input size n goes to infinity.
 - The word comes from “asymptote”, which is where you look at the limiting behavior of a function as something goes to infinity.
- This gives a solid mathematical way to capture the intuition of emphasizing scalable performance.
- It also makes the analysis a lot simpler!

Big-O (Big-Oh) Notation

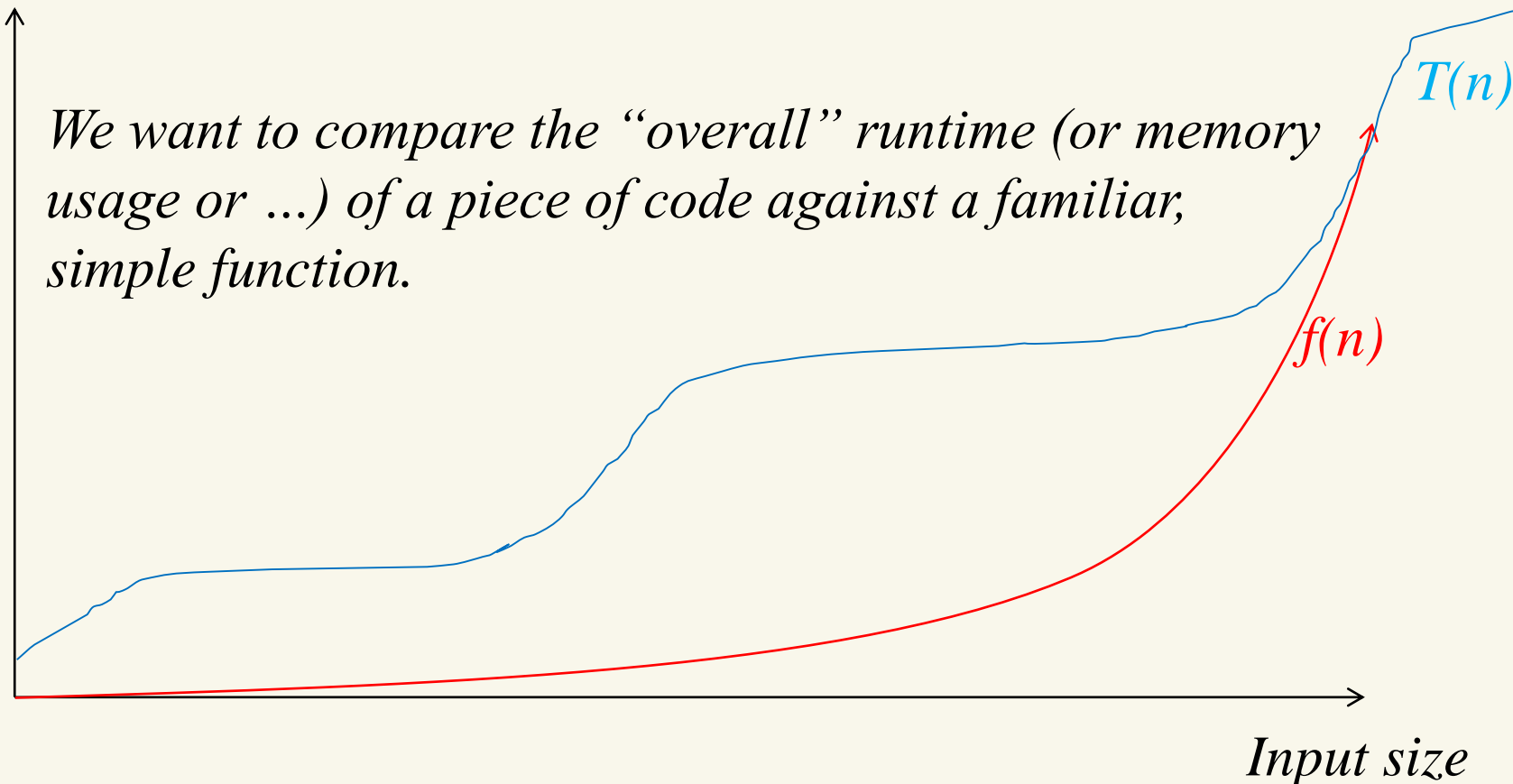
- Let $T(n)$ and $f(n)$ be functions mapping $\mathbb{Z}^+ \rightarrow \mathbb{R}^+$.

Positive integers

Positive real numbers

We want to compare the “overall” runtime (or memory usage or ...) of a piece of code against a familiar, simple function.

Time
(or anything
else we can
measure)



Big-O Notation

There exists

$T(n) \in O(f(n))$ if $\exists c$ and n_0 such that

$$T(n) \leq c f(n) \quad \forall n \geq n_0$$

For all

$c f(n)$

$T(n)$

$f(n)$

The function $f(n)$ is, asymptotically “greater than or equal to” the function $T(n)$ if in the “long run”, $f(n)$ (multiplied by a suitable constant) upper-bounds $T(n)$.

Time
(or anything
else we can
measure)

Input size

n_0

Big-O Notation

There exists

$T(n) \in O(f(n))$ if $\exists c$ and n_0 such that

$$T(n) \leq c f(n) \quad \forall n \geq n_0$$

For all

$c f(n)$

$T(n)$

$f(n)$

n_0

We **do** want the comparison to be valid for all sufficiently large inputs... but we're willing to ignore behaviour on small examples. (Looking for a “steady state”.)

Time
(or anything
else we can
measure)

Input size

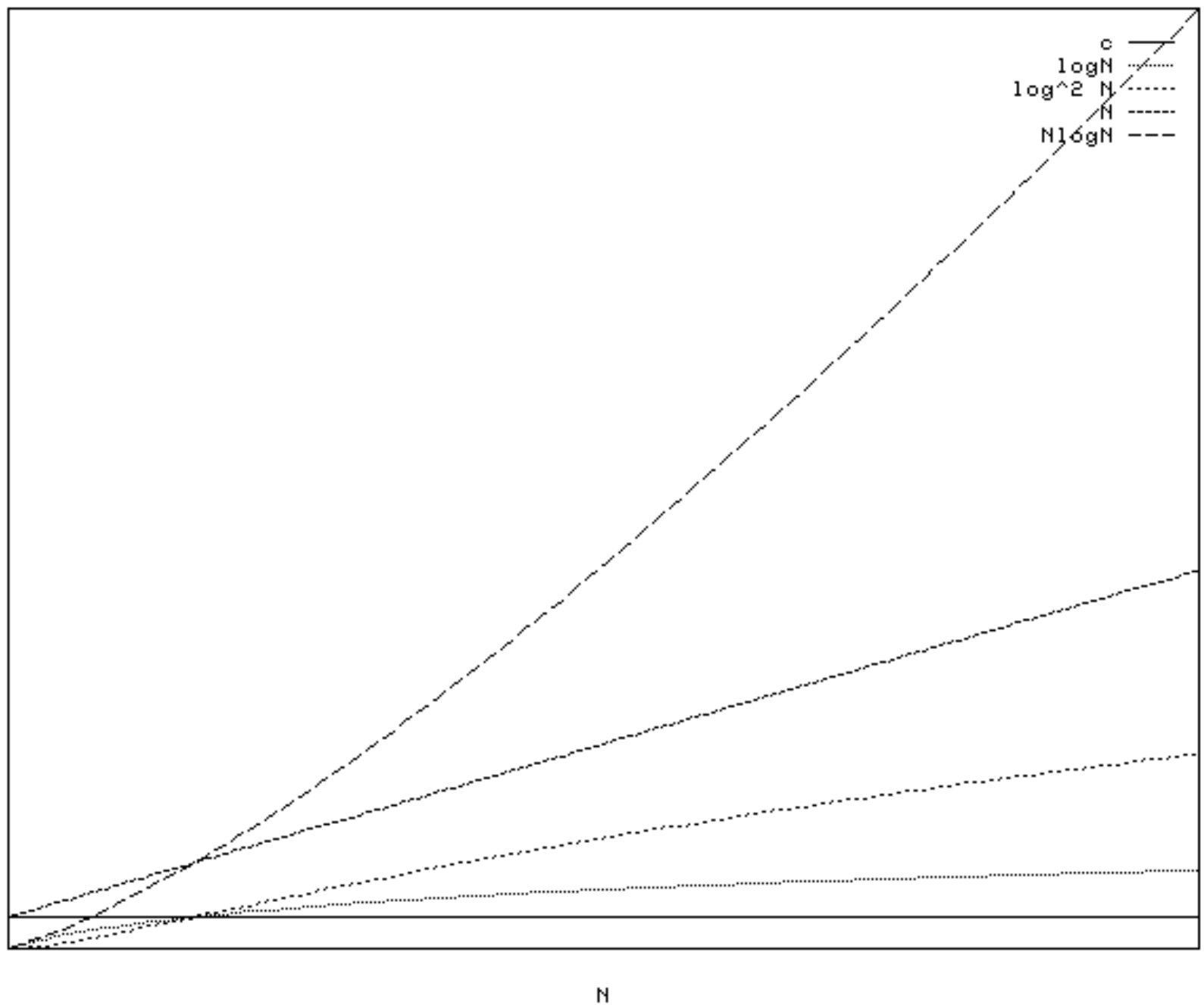
Big-O Notation (cont.)

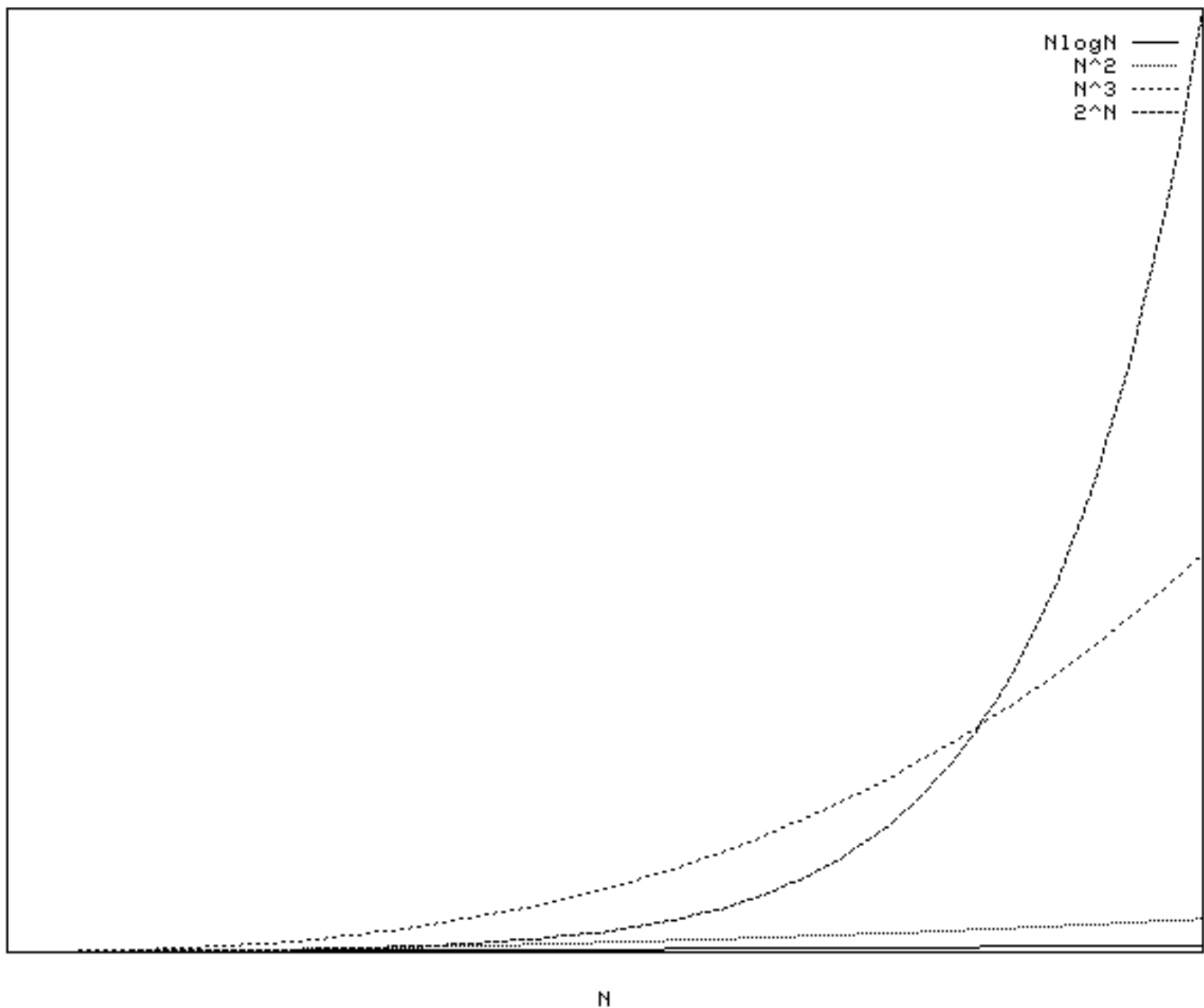
- Using Big-O notation, we might say that Algorithm A “runs in time Big-O of $n \log n$ ”, or that Algorithm B “is an order n -squared algorithm”. We mean that the number of operations, as a function of the input size n , is $O(n \log n)$ or $O(n^2)$ for these cases, respectively.
- Constants don't matter in Big-O notation because we're interested in the **asymptotic behavior** as n grows arbitrarily large; but, be aware that large constants can be very significant in an actual implementation of the algorithm.

Typical Growth Rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates





Rates of Growth

- Suppose a computer executes 10^{12} ops per second:

n =	10	100	1,000	10,000	10^{12}
n	$10^{-11}s$	$10^{-10}s$	$10^{-9}s$	$10^{-8}s$	1s
n lg n	$10^{-11}s$	$10^{-9}s$	$10^{-8}s$	$10^{-7}s$	40s
n^2	$10^{-10}s$	$10^{-8}s$	$10^{-6}s$	$10^{-4}s$	$10^{12}s$
n^3	$10^{-9}s$	$10^{-6}s$	$10^{-3}s$	1s	$10^{24}s$
2^n	$10^{-9}s$	$10^{18}s$	$10^{289}s$		

$10^4s = 2.8 \text{ hrs}$

$10^{18}s = 30 \text{ billion years}$