Algorithms, Data Structures, and Problem Solving

Masoumeh Taromirad

Hamlstad University

DA4002, Fall 2016

Motivating Example

how does problem size influence an algorithm?

for example insertion sort

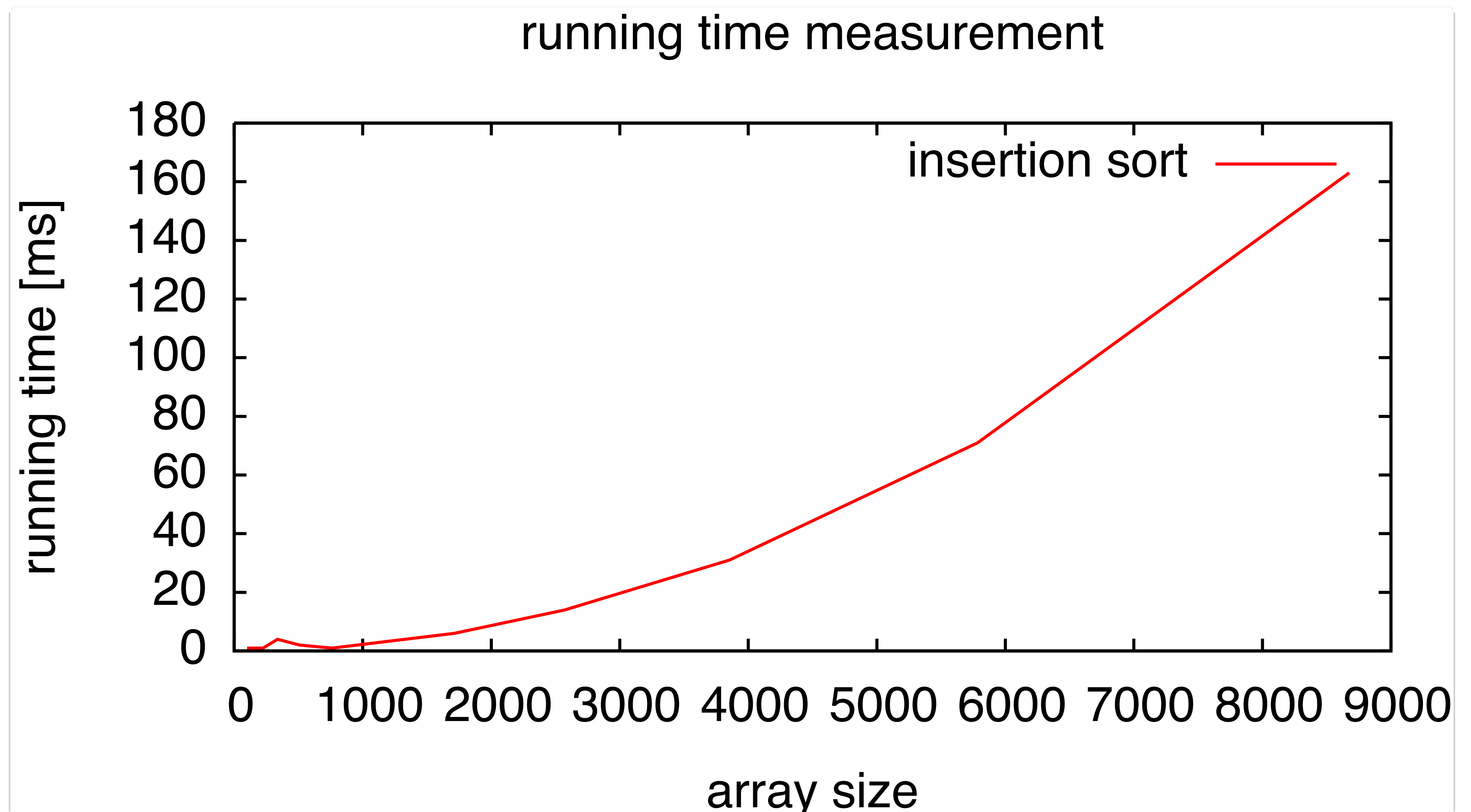
```
for (len = 100; len < 10000; len *= 2) {
 int * data = random_number_array (len);
 clock_t t0, t1;
 t0 = clock ();
 insertion_sort (data);
 t1 = clock ();
 printf ("%d\t%lu", len, (t1 - t0));
 free (data);
}
```

Motivating Example

how does problem size influence an algorithm?

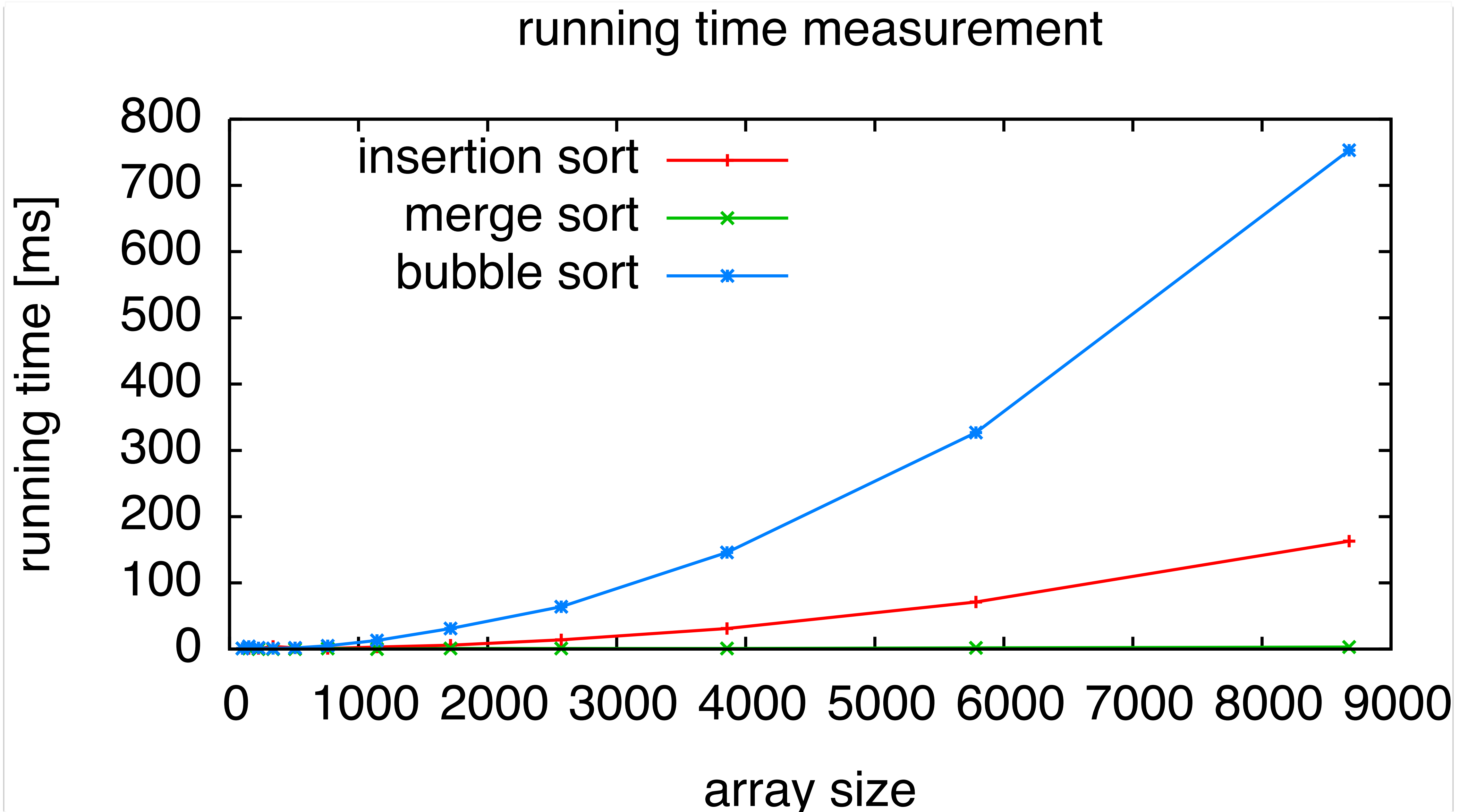
for example insertion sort

len	*t1-t0*
100	1
150	1
225	1
338	2
507	0
761	17
1142	3
1713	6
2570	14
3855	31
5783	71
8675	166



Motivating Example

do the same for other sorting algorithms...

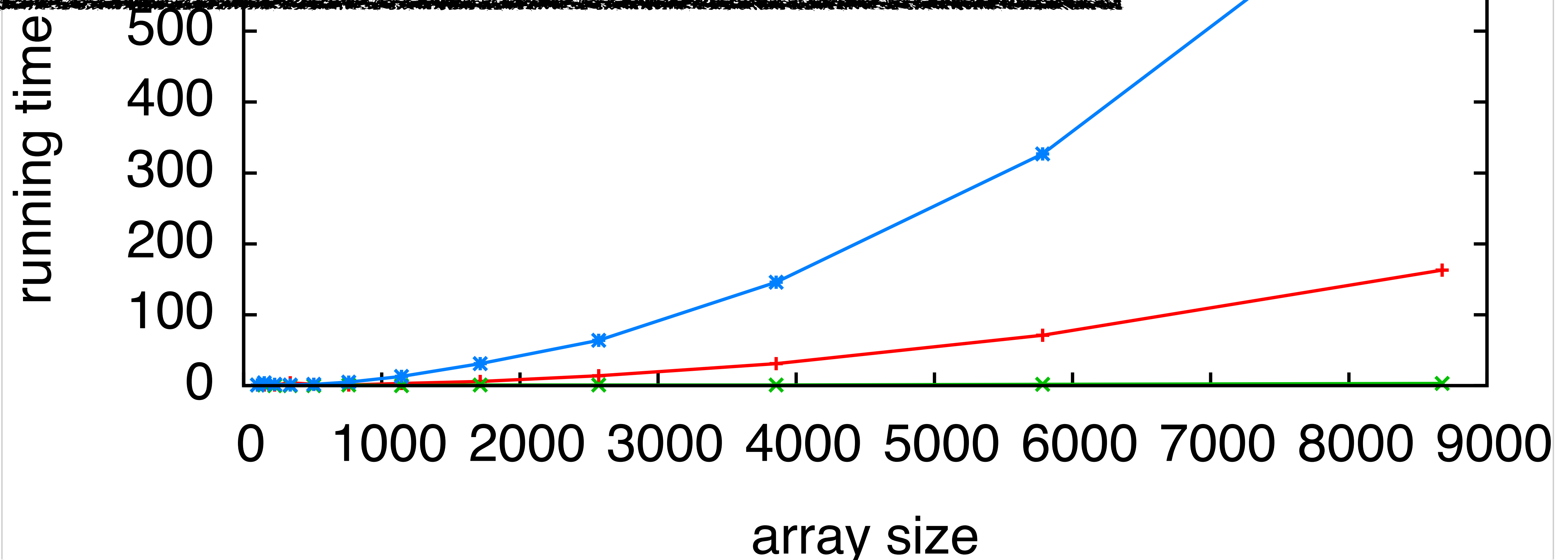


Motivating Example

do the same for other sorting algorithms...

complexity analysis:

classify algorithms based on the ***shape*** of their time-growth



Complexity Analysis

- estimating performance as a function of problem size
  - running time
  - memory requirement
- not interested in precise predictions
  - upper-bound for the worst-case growth rate

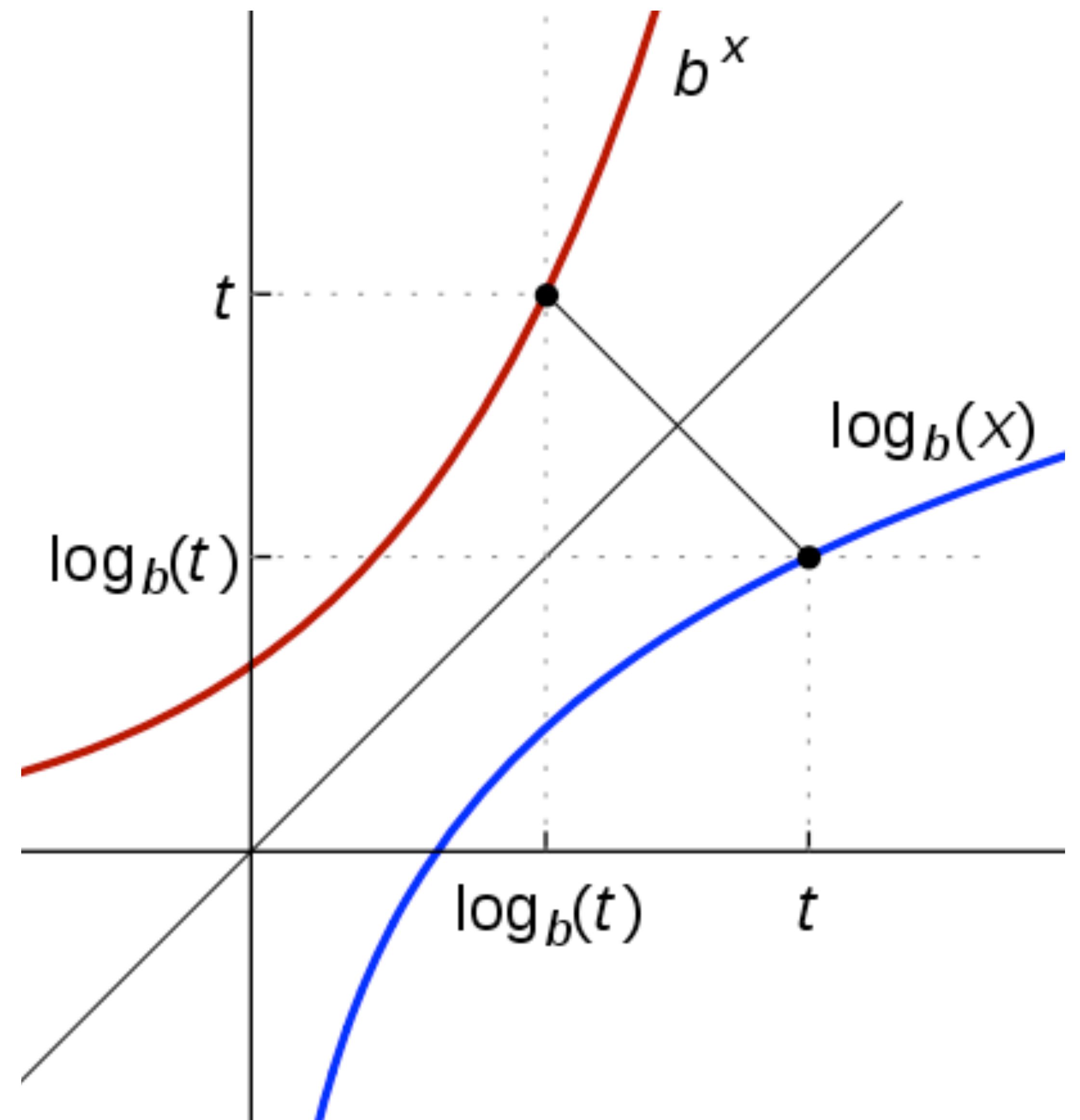
Complexity Analysis

- estimating performance as a function of problem size
  - running time
  - *we'll ignore memory in this course*
- not interested in precise predictions
  - upper-bound for the worst-case growth rate

Foundations of Computational Complexity

Math Refresher

polynomials



logarithms

Polynomials

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

$$= a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

in complexity analysis, we can simplify:

$$p(x) \approx a_n x^n \quad \forall x > x_{\text{large}}$$

Logarithms

logarithms **invert** an exponential:

what is the value of **y**?

$$x = b^y \Leftrightarrow y = \log_b(x)$$

we will use b=2 and simply write

$$logN = log_2(N)$$

(Why do we need logarithms?)

- example: binary search
  - how long will the loop run?
  - each step cuts the problem size in half

$$P = 2^R \Leftrightarrow R = \log P$$

problem size

number of times it can be cut in half

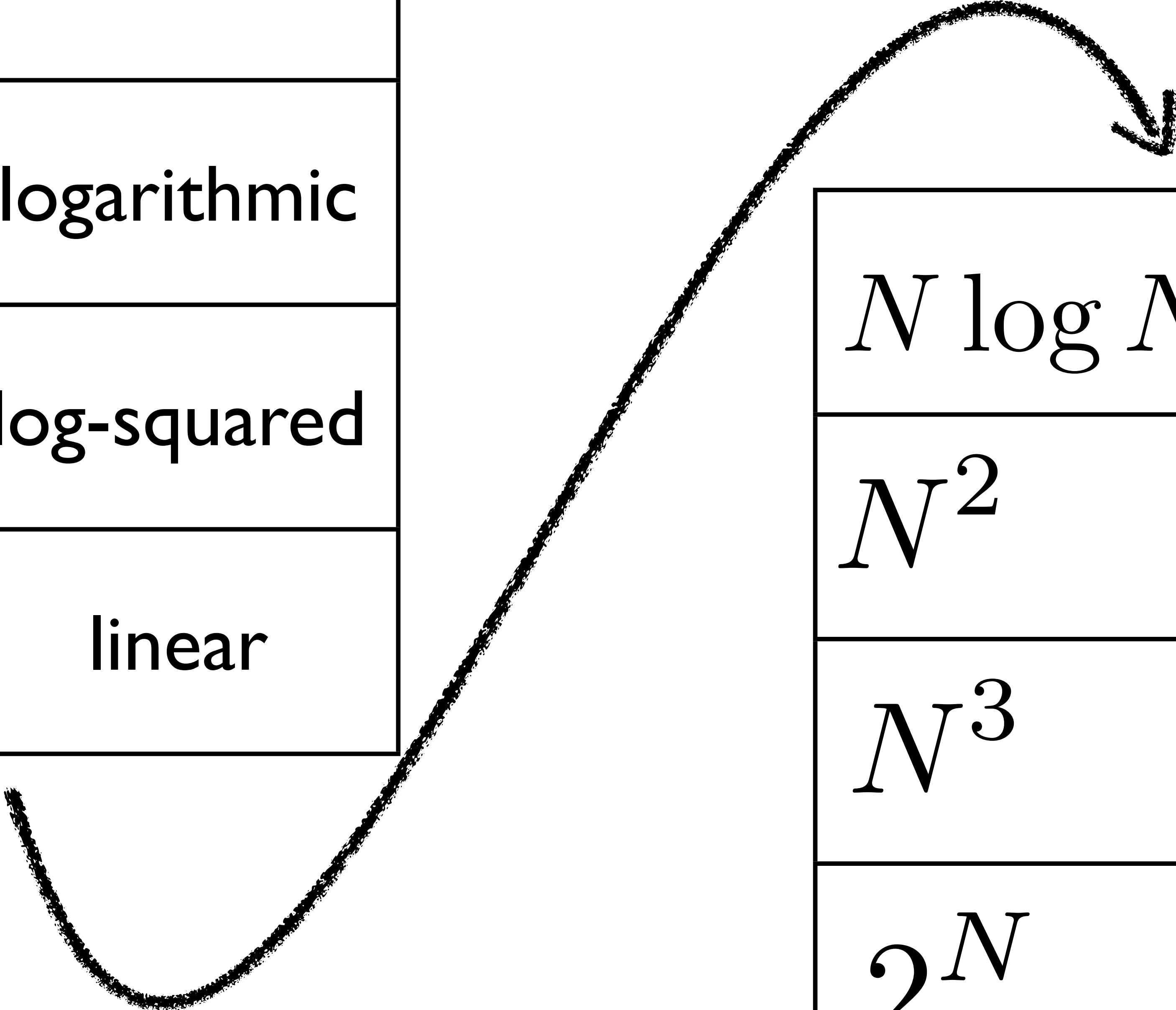
- example: binary trees
  - how high is a tree that fits N nodes?

$$N = 2^{H+1} - 1 \Leftrightarrow \underline{H = \log(N+1) - 1}$$

Commonly Encountered Functions

slowest-growing

c	constant
$\log N$	logarithmic
$\log^2 N$	log-squared
N	linear



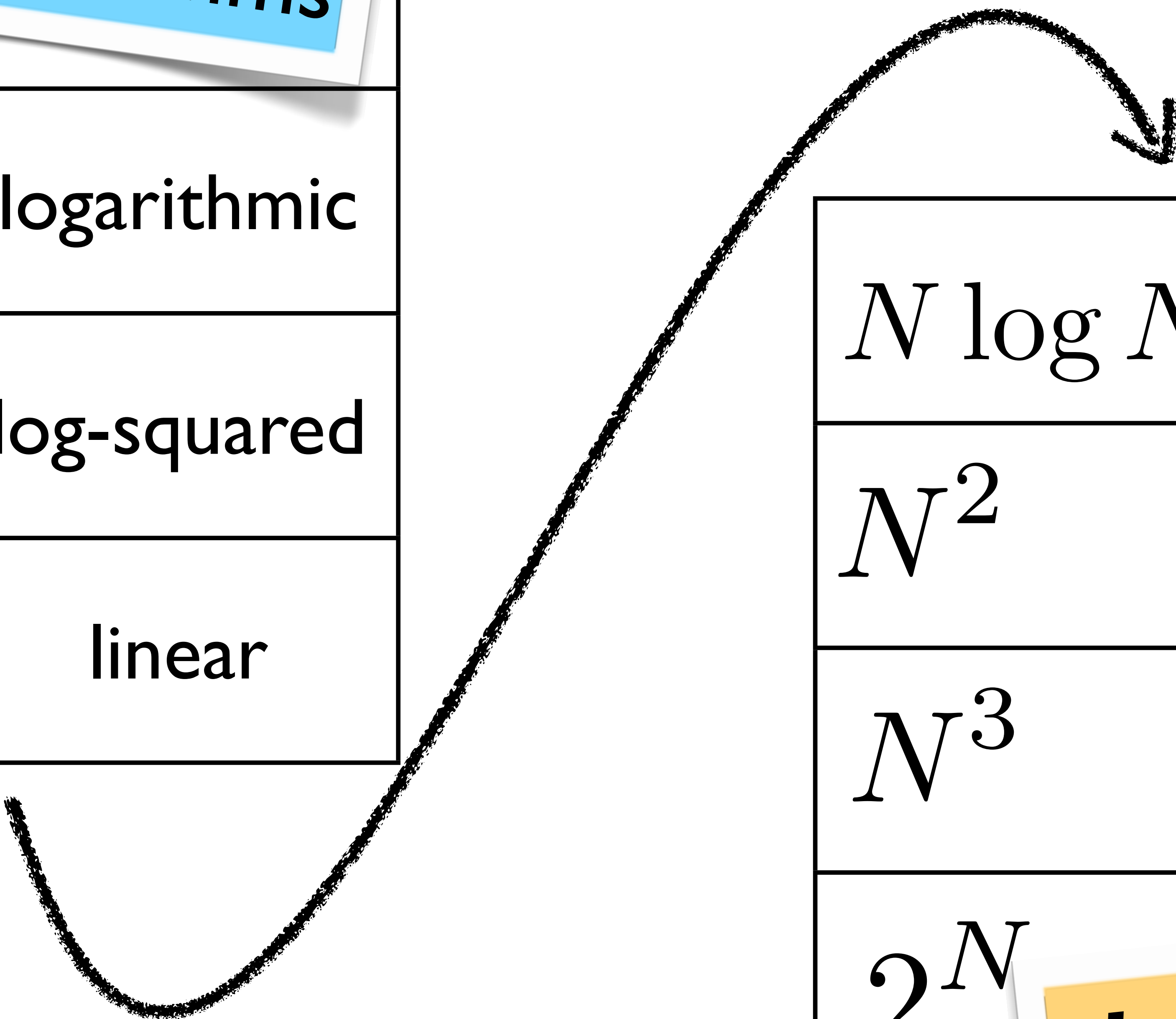
$N \log N$	N log N
N^2	quadratic
N^3	cubic
2^N	exponential

fastest-growing

Commonly Encountered Functions

fast algorithms

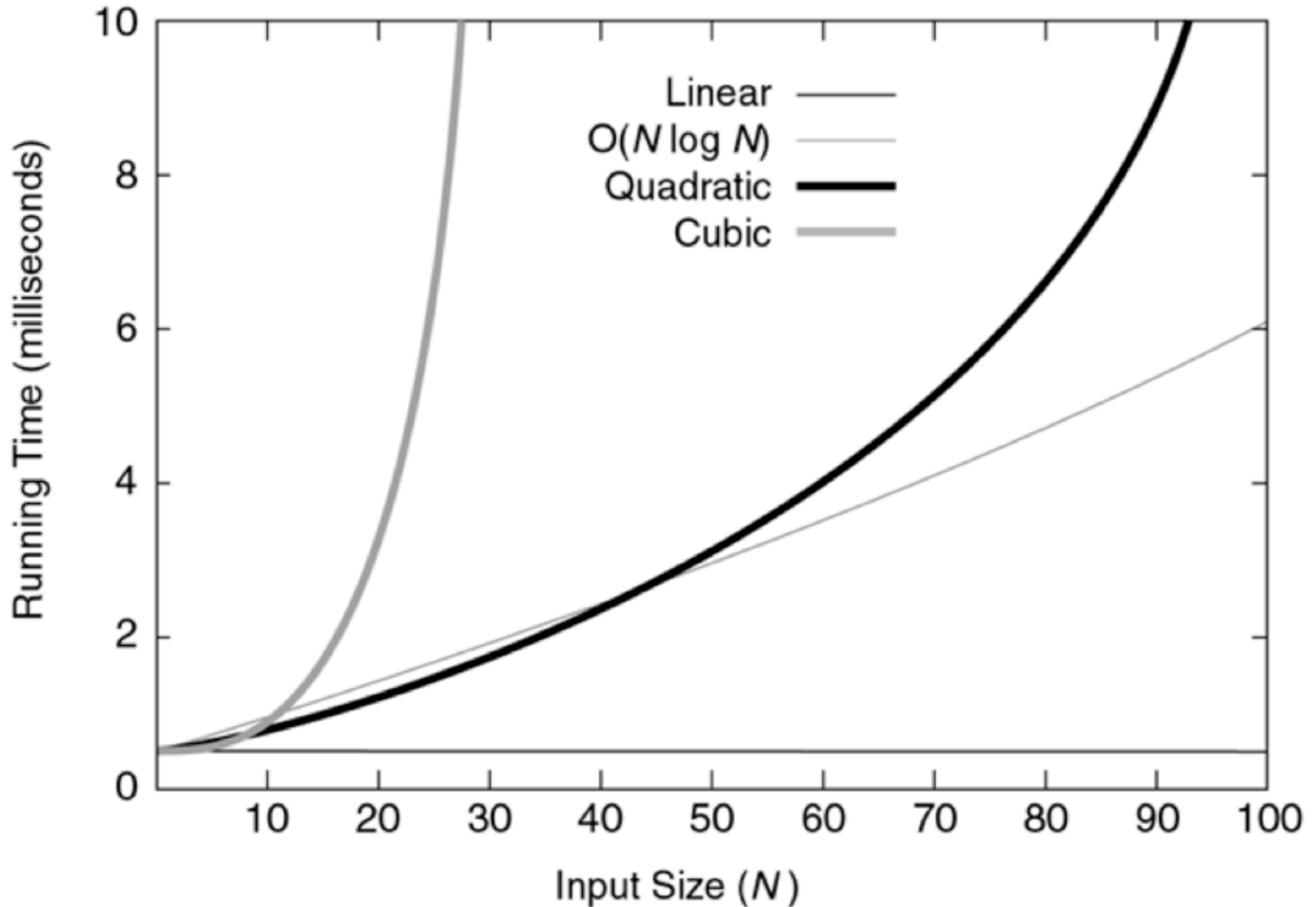
c	
$\log N$	logarithmic
$\log^2 N$	log-squared
N	linear



$N \log N$	N log N
N^2	quadratic
N^3	cubic
2^N	

slow algorithms

Growth Rates



Upper Bound: "Big-Oh" Notation

definition

"Big-Oh" indicates how the upper bound on execution time changes with problem size

formally, it is written **O(F(N))** and defined mathematically like this:

$$T(N) \in O(F(N))$$

$$\Leftrightarrow$$

$$\exists\, c, N_0 > 0 \Rightarrow T(N) \leq cF(N)\ \forall\, N \geq N_0$$

Upper Bound: "Big-Oh" Notation

execution time T in function of problem size N

complexity class [example: O(log N)]

it's an **upper bound**

valid for sufficiently **large** problems

$$T(N) \in O(F(N))$$

$$\Leftrightarrow$$

$$\exists\ c, N_0 > 0 \Rightarrow T(N) \leq cF(N)\ \forall\ N \geq N_0$$

Big-Oh Simplification Rules

- only keep the fastest-growing additive terms
- remove constants
- examples:

$$O(17.9 + 3.5 \times N + 0.1 \times N^2) = O(N^2)$$

$$O(10000 \times N^3 + 0.00001 \times 2^N) = O(2^N)$$

Big-Oh Intuition

when the problem size N grows to N'
the execution time T grows to T'
according to the term "inside" the Big-Oh

Big-Oh	$\boldsymbol{N' = 2N}$	$\boldsymbol{N' = 10N}$
c	$T' = T$	$T' = T$
$\log N$	$T' = T + c$	$T' = T + 3.32c$
$\log^2 N$	$T' = T + (1 + 2\log N)c$	$T' = T + 3.32(1 + 2\log N)c$
N	$T' = 2T$	$T' = 10T$
$N \log N$	$T' = 2(Nc + T)$	$T' = 10(3.32Nc + t)$
N^2	$T' = 4T$	$T' = 100T$
N^3	$T' = 8T$	$T' = 1000T$
2^N	$T' = \sqrt{c}T^2$	$T' = \sqrt[10]{c}T^{10}$

Complexity of Iterative Algorithms

Example: Linear Search

```
for (ii = 0; ii < length; ++ii) {
 if (array[ii] == x) {
  return ii;
 }
}
return -1;
```

Example: Linear Search

N = array length

```
for (ii = 0; ii < length; ++ii) {
  if (array[ii] == x) {
    return ii;
  }
}
return -1;
```

worst case A

x is the last element: N iterations

worst case B

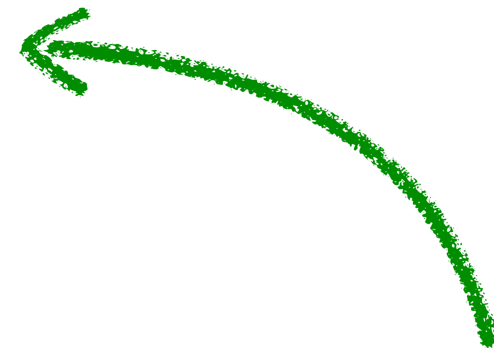
x is not in the array: N iterations

average case

x lies in the middle: N/2 iterations

Example: Linear Search

***"Big-Oh"* complexity is O(N)**



worst case A

  x is the last element: N iterations

worst case B

  x is not in the array: N iterations

average case

  x lies in the middle: N/2 iterations

Example: Nested Loops

```
for (ii = 0; ii < N; ++ii)
  for (jj = 0; jj < N; ++jj)
    ++sum;
```

Example: Binary Search

```
int low = 0;
int high = length - 1;
while (low <= high) {
  int mid = (low + high) / 2;
  if (array[mid] < x) {
    low = mid + 1;
  }
  else if (array[mid] > x) {
    high = mid - 1;
  }
  else {
    return mid;
  }
}
return -1;
```

Example: Binary Search

```
int low = 0;
int high = length - 1;
while (low <= high) {
  int mid = (low + high) / 2;
  if (array[mid] < x) {
    low = mid + 1;
  }
  else if (array[mid] > x) {
    high = mid - 1;
  }
  else {
    return mid;
  }
}
return -1;
```

cut the problem in **half**

how often do we have to cut?

the opposite question is easier:

*if we cut **x** times, then how long can the array be at most?*

Example: Binary Search

```
int low = 0;
int high = length - 1;
while (low <= high) {
   int mid = (low + high) / 2;
   if (array[mid] < x) {
      low = mid + 1;
```

cut the problem in **half**

how often do we have to cut?

the opposite question is easier:

*if we cut **x** times, then how long can the array be?*

$$N_{\max} = 2^x$$

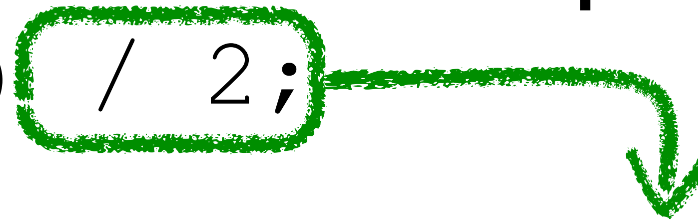
...to invert the exponential, we use the logarithm:

$$x \leq \log N$$

Example: Binary Search

```
int low = 0;
int high = length - 1;
while (low <= high) {
  int mid = (low + high) / 2;
  if (array[mid] < x) {
    low = mid + 1;
  }
  else if (array[mid] > x) {
    high = mid - 1;
  }
  else {
    return mid;
  }
}
return -1;
```

cut the problem in **half**



O(log N)

Example: Sequence of For Loops

```
for (ii = 0; ii < N; ++ii)
   sum++;
for (jj = 1; jj <= N; jj *= 2)
   sum++;
```

Insertion Sort Group Activity

extracting execution time formulas from source code

Merge Sort Group Activity

review function call mechanism, and
analyze recursive execution time expressions

Complexity of Recursive Algorithms

Merge sort is a “Divide and Conquer” algorithm.
There is a nice general formula for those.

Complexity Analysis of Divide & Conquer Algorithms

- theoretical runtime expressions are **recursive**
  - rather tricky in general
  - but there is a recipe that can be used in most situations encountered in practice
- Wikipedia calls it "Master Theorem"
  http://en.wikipedia.org/wiki/Master_theorem

Complexity Analysis of Divide & Conquer Algorithms

```
procedure someDnC( N ) :
    if N < 1 then exit

    Do work of amount f(N)

    someDnC(N/B)
    ...A times...
    someDnC(N/B)
 end procedure
```

overhead
(divide + reassamble)

recursion
(smaller sub-problems)

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + f(N)$$

Apply this to Merge Sort

recurse **twice** with N'=**N/2**

```
void msort (char ** strv, char ** tmp,
            int begin, int end)
{
  int length, middle;
  length = end - begin;
  if (length <= 1)
    return;
  middle = begin + length / 2;
  msort (strv, tmp, begin, middle);
  msort (strv, tmp, middle, end);
  merge (strv, tmp, begin, middle, end);
}
```

overhead:

divide: O(c)

merge: O(N)

total: **O(N)**

Apply this to Merge Sort

recurse **twice**
with N'=**N/2**

```
procedure someDnC( N ) :
    if N < 1 then exit

    Do work of amount f(N)

    someDnC(N/B)
    ...A times...
    someDnC(N/B)
 end procedure
```

overhead:
divide: O(c)
merge: O(N)
total: **O(N)**

$$A = 2$$

$$B = 2$$

$$f(N) \in O(N)$$

$$\Rightarrow T(N) = 2T\left(\frac{N}{2}\right) + cN$$

Apply this to Merge Sort

- T(N) is defined the **recursively**

$$T(N) = 2T\left(\frac{N}{2}\right) + cN$$

- *illustrate this case on the whiteboard...*

Apply this to Merge Sort

- T(N) is defined the **recursively**

$$T(N) = 2T\left(\frac{N}{2}\right) + cN$$

- *illustrate this case on the whiteboard...*

$$T(N) \in O(N \log N)$$

General values for A, B, and f(N)

- conditions and assumptions:

$$A \geq 1$$

$$B > 1$$

$$f(N) \in O(N^k)$$

$$T(1) = 1$$

$$N = B^M \Rightarrow \begin{cases} \frac{N}{B} = B^{M-1} \\ N^k = \left(B^M\right)^k = \left(B^k\right)^M \end{cases}$$

- resulting general form:

$$T(N) = AT\left(\frac{N}{B}\right) + O(N^k)$$

General values for A, B, and f(N)

- conditions and assumptions:

$$A \geq 1$$

$$B > 1$$

$$f(N) \in O(N^k)$$

$$T(1) = 1$$

$$N = B^M \Rightarrow \begin{cases} \frac{N}{B} = B^{M-1} \\ N^k = \left(B^M\right)^k = \left(B^k\right)^M \end{cases}$$

- resulting general form:

sloppy notation

(mixes function with bound)

$$T(N) = AT\left(\frac{N}{B}\right) + O(N^k)$$

“Master Theorem”

- given a runtime estimate of the recursive form:

$$T(N) = AT\left(\frac{N}{B}\right) + O(N^k)$$

- its Big-Oh complexity is given by:

$$T(N) \in \begin{cases} O\left(N^{\log_B A}\right) & \Leftarrow A > B^k \\ O\left(N^k \log N\right) & \Leftarrow A = B^k \\ O\left(N^k\right) & \Leftarrow A < B^k \end{cases}$$

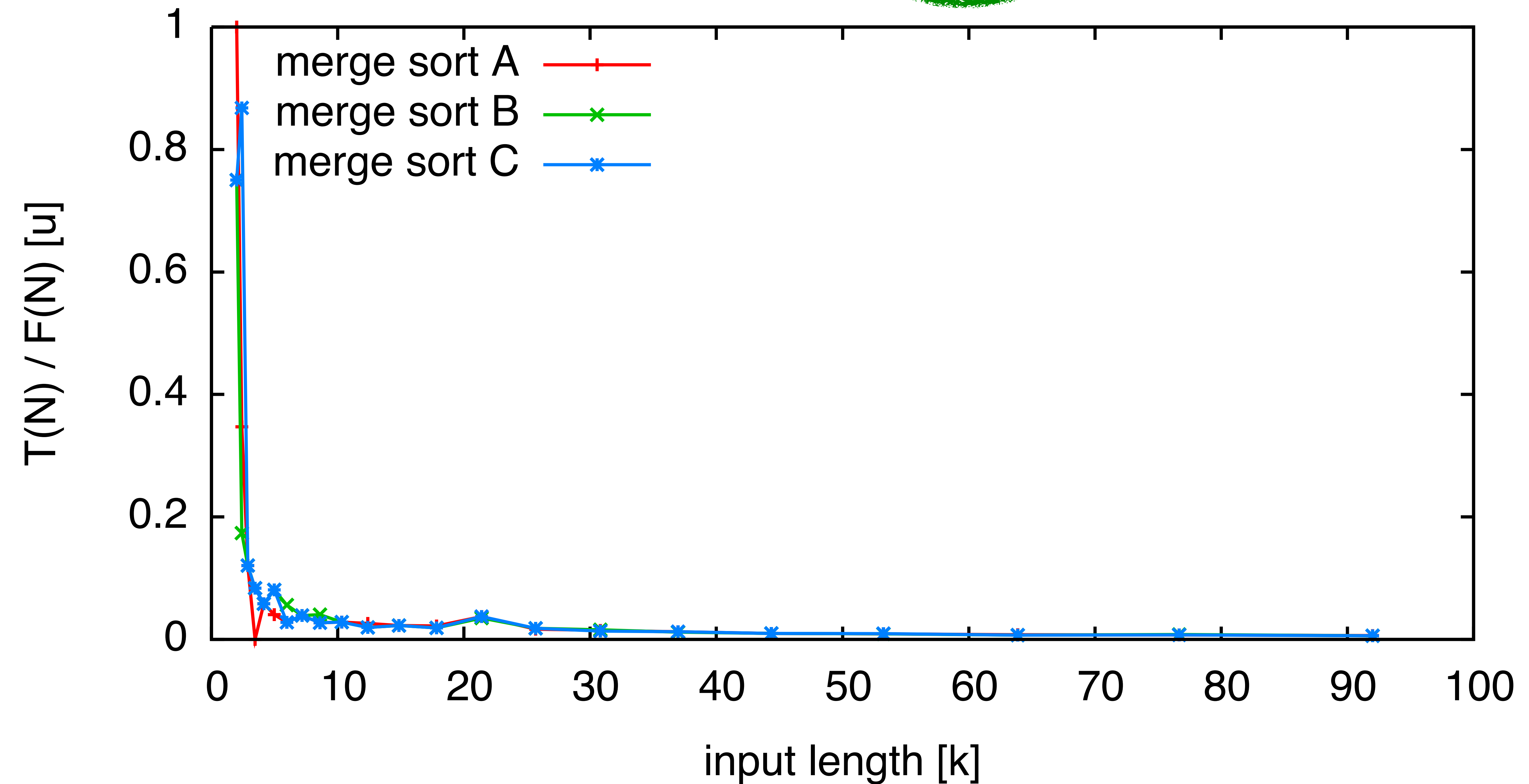
Matching Theory to Reality

is $T(N) \in O(F(N))$ true?

- check whether T(N)/F(N) converges
  - if it goes to zero,
    F(N) grows too fast
  - if it goes to infinity,
    F(N) grows too slowly
  - if it converges to a constant,
    we found the correct answer

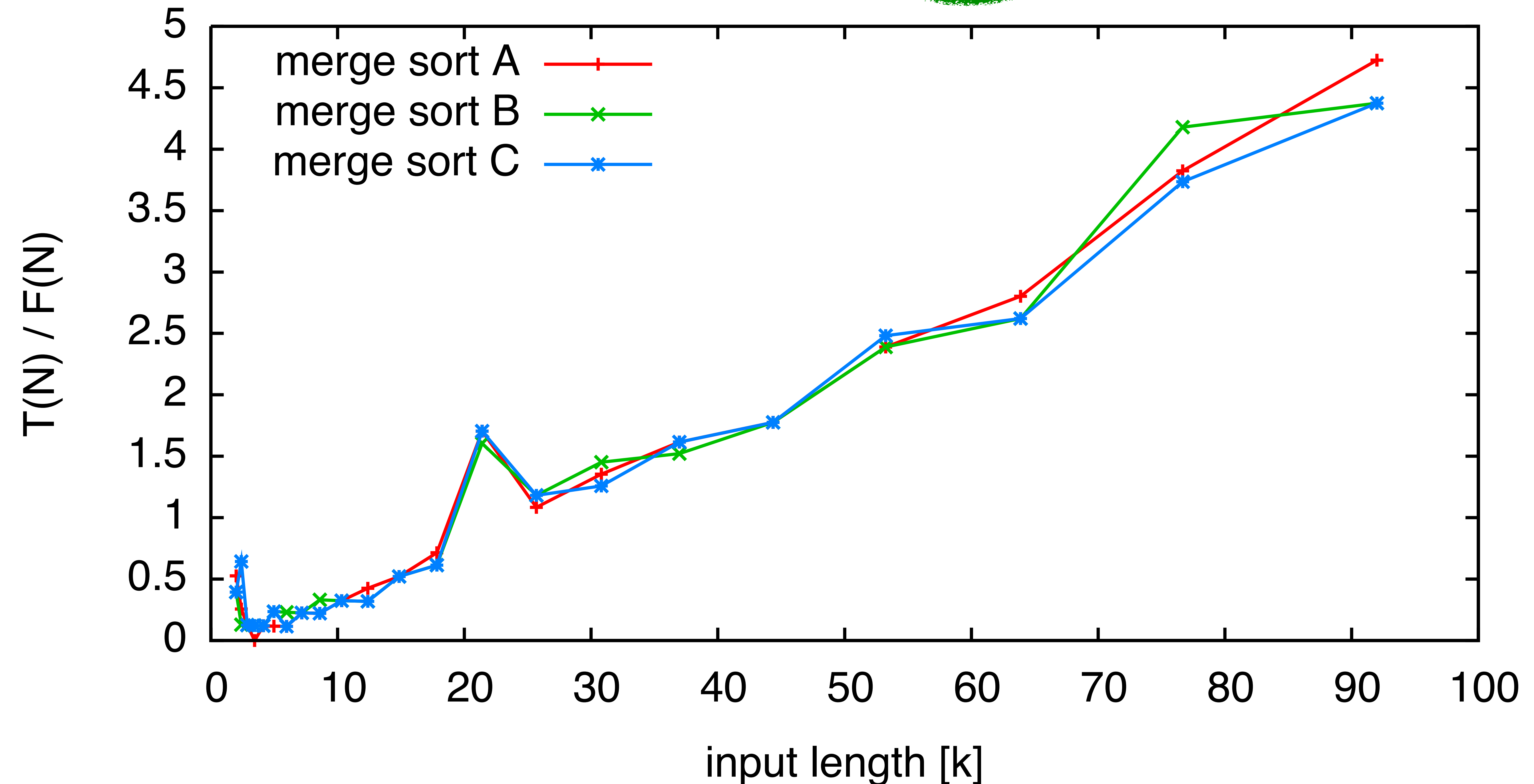
*example: is merge sort O(N*N)? or O(log N)? or O(N log N)?*

Matching Theory to Reality

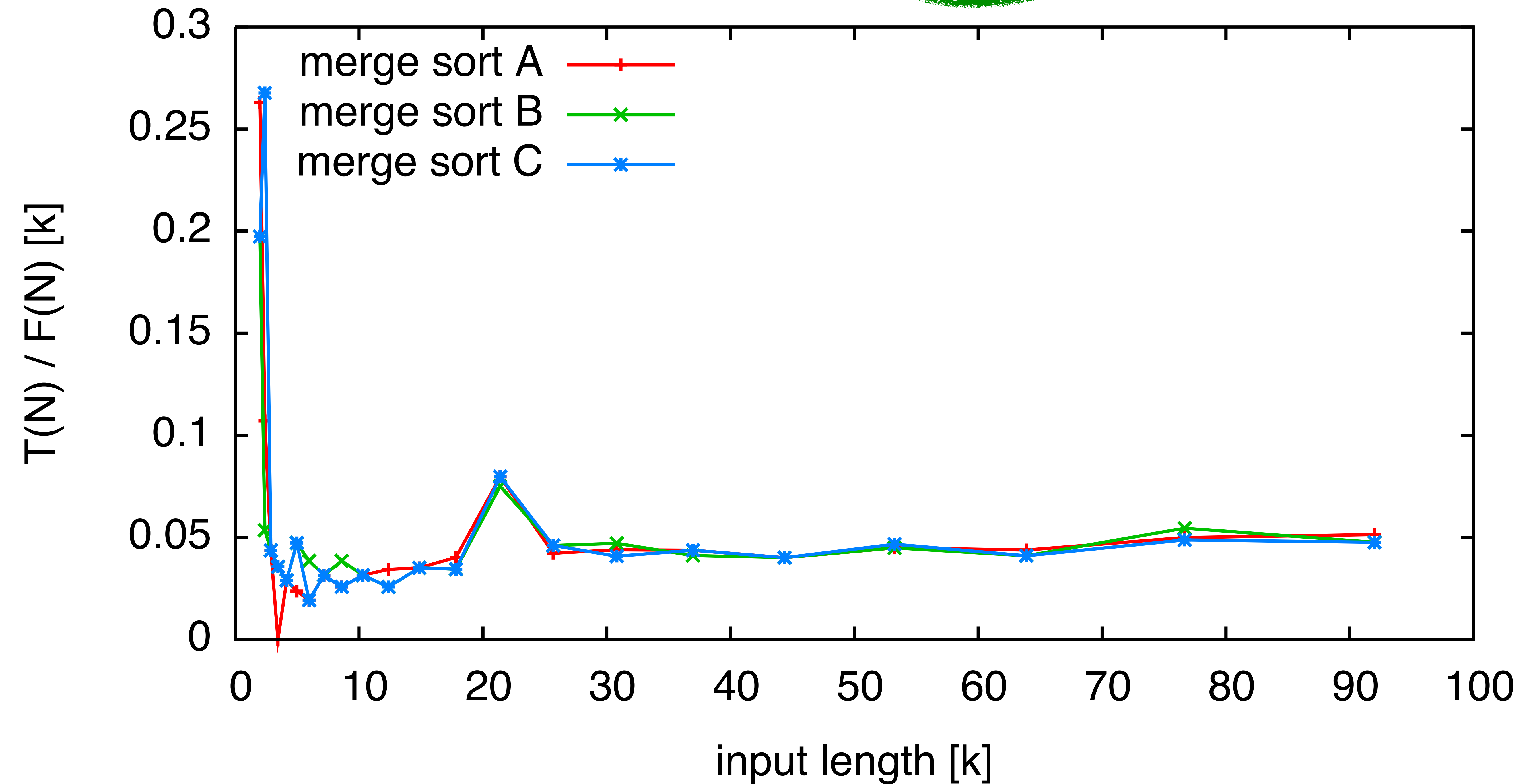


Matching Theory to Reality

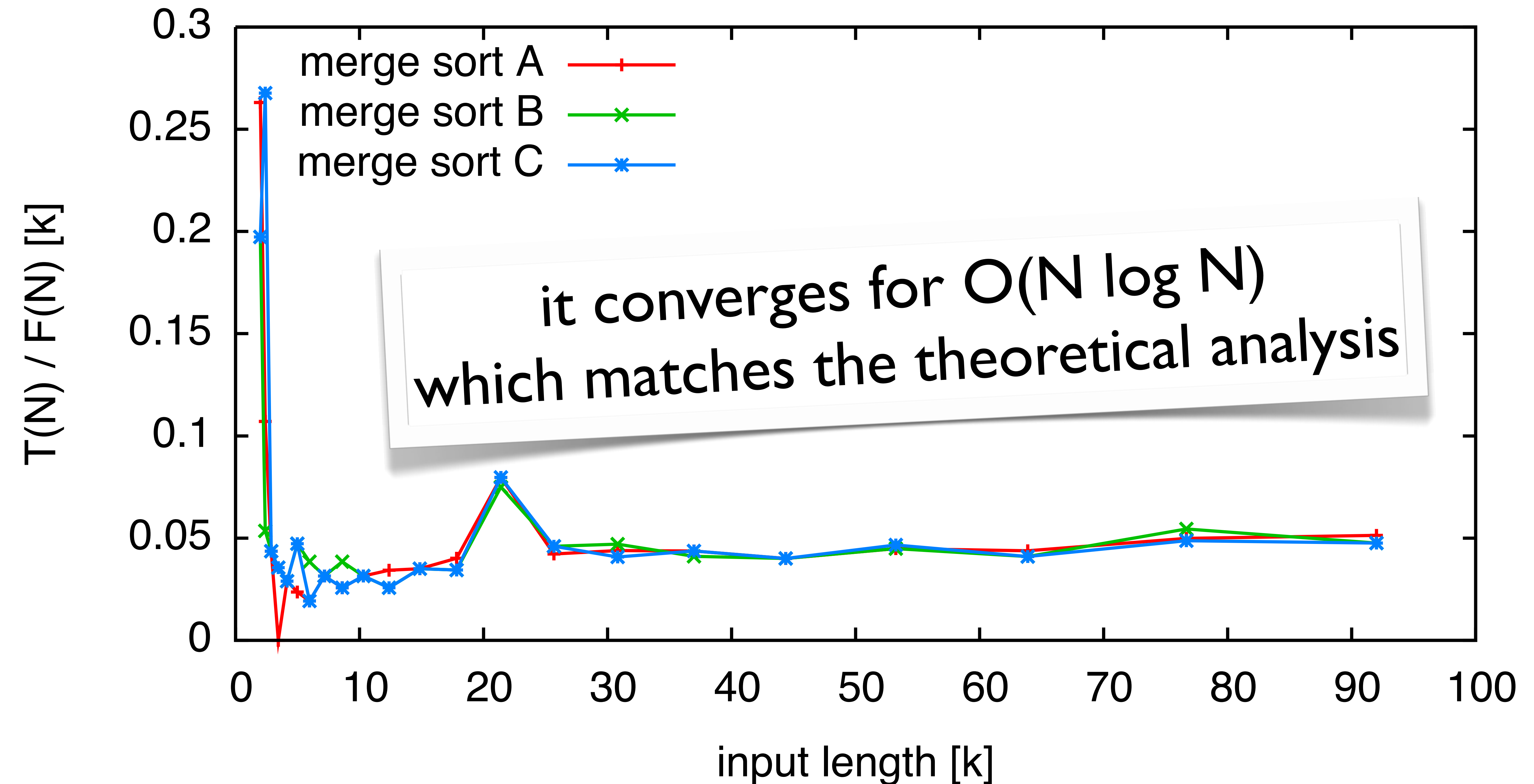
sorting time O(log N) is it O(log N)?



Matching Theory to Reality



Matching Theory to Reality



Take-Home Message

- find the shape of an upper bound on computation time (Big-Oh notation)
- relatively easy for most iterative cases
- general formula for divide & conquer
- Big-Oh needs to be complemented by empirical data in practice