CS180: Algorithms and Complexity

Professor: Raghu Meka (raghum@cs)

Plan for Today

- Logistics
- What is an algorithm?

Course goals

- **“Algorithmic thinking”**
  - Design and analysis
- **Algorithmic lens: all areas of science**
  - Applications
- **Core design principles and algorithms**

Discussion sections

Discussion sections

Attend them!

Discussion sections

Attend them!

Solutions to homework and practice problems

Grading

Grading

1. Homework: 6 x 3

Grading

1. Homework: 6 x 3

2. Weekly quizzes: 10 x 1

Grading

1. Homework: 6 x 3

2. Weekly quizzes: 10 x 1

3. Exam 1: 26

Grading

1. Homework: 6 x 3
2. Weekly quizzes: 10 x 1
3. Exam 1: 26
4. Exam 2: 22

Grading

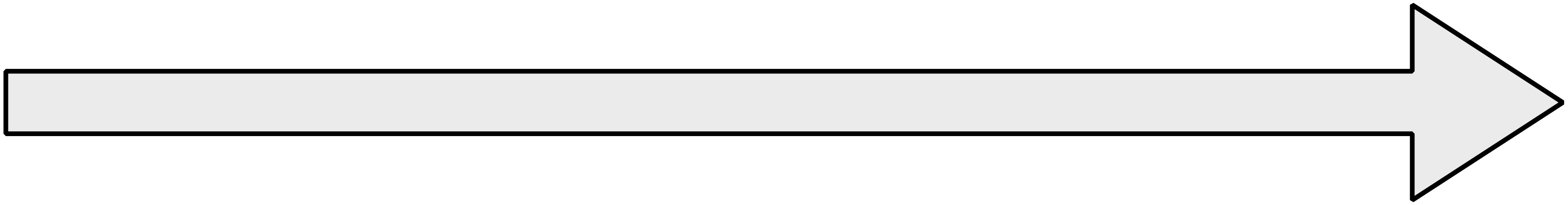
1. Homework: 6 x 3
2. Weekly quizzes: 10 x 1
3. Exam 1: 26
4. Exam 2: 22

Grading

1. Homework: 6 x 3
2. Weekly quizzes: 10 x 1
3. Exam 1: 26
4. Exam 2: 22
5. Exam 3: 24

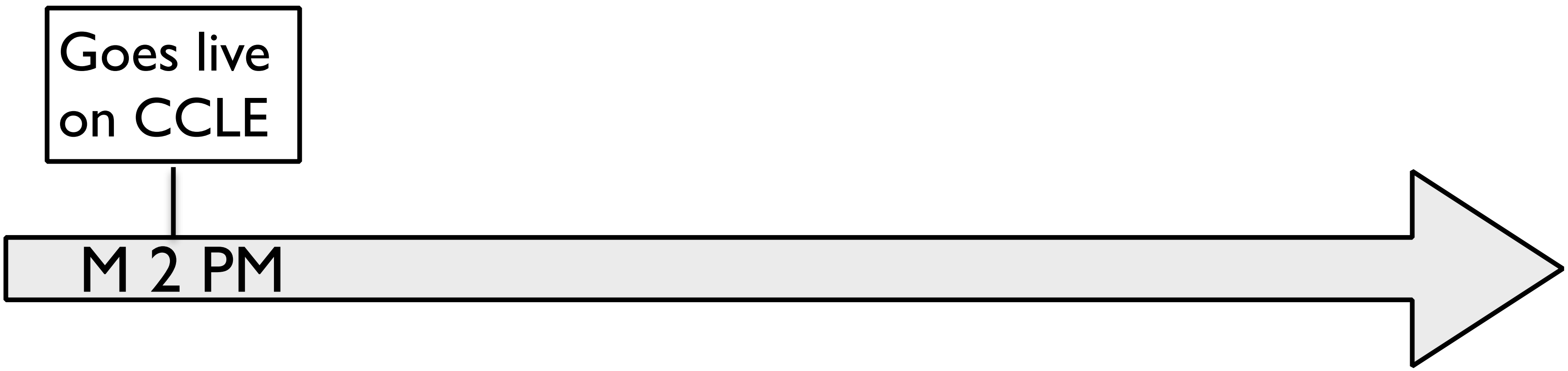
Grading: Homework

Life of a H.W



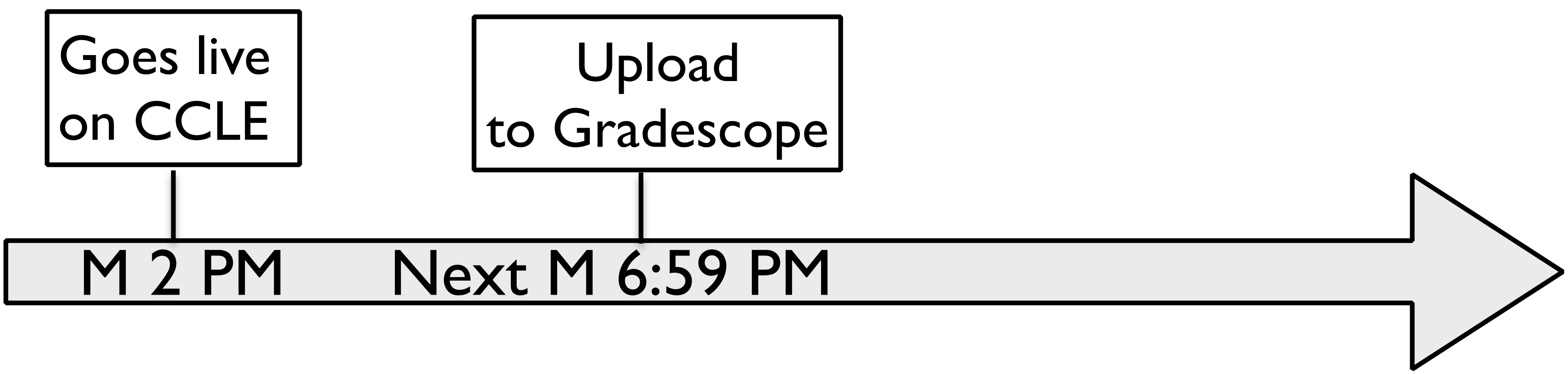
Grading: Homework

Life of a H.W



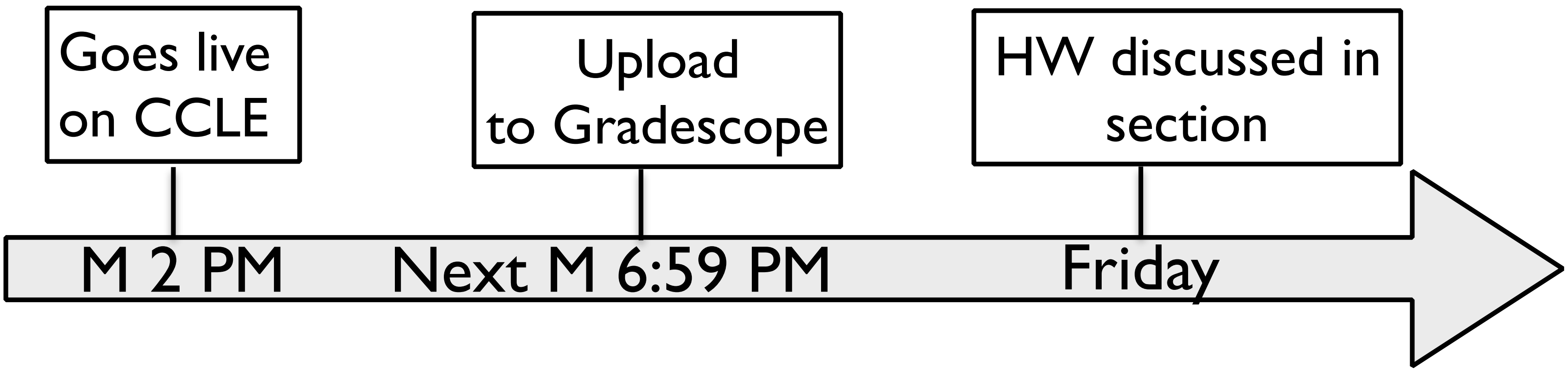
Grading: Homework

Life of a H.W



Grading: Homework

Life of a H.W



Homework rules and tips

Homework rules and tips

1. **Start early to use office hours**

Homework rules and tips

1. **Start early to use office hours**

2. **Late-submissions = 0 credit**

Homework rules and tips

1. Start early to use office hours
2. Late-submissions = 0 credit
3. Regrade: Ask on Gradescope within a week

Homework rules and tips

1. **Start early to use office hours**

2. **Late-submissions = 0 credit**

3. **Regrade: Ask on Gradescope within a week**

4. **Attempts count:**

   100% - correct; 75% - serious attempt; 50% - reasonable attempt.

Grading: Collaboration

Grading: Collaboration

1. **Collaboration encouraged:**
   Must write your own answers.

Grading: Collaboration

1. **Collaboration encouraged:**
   Must write your own answers.

2. **Form study-groups**

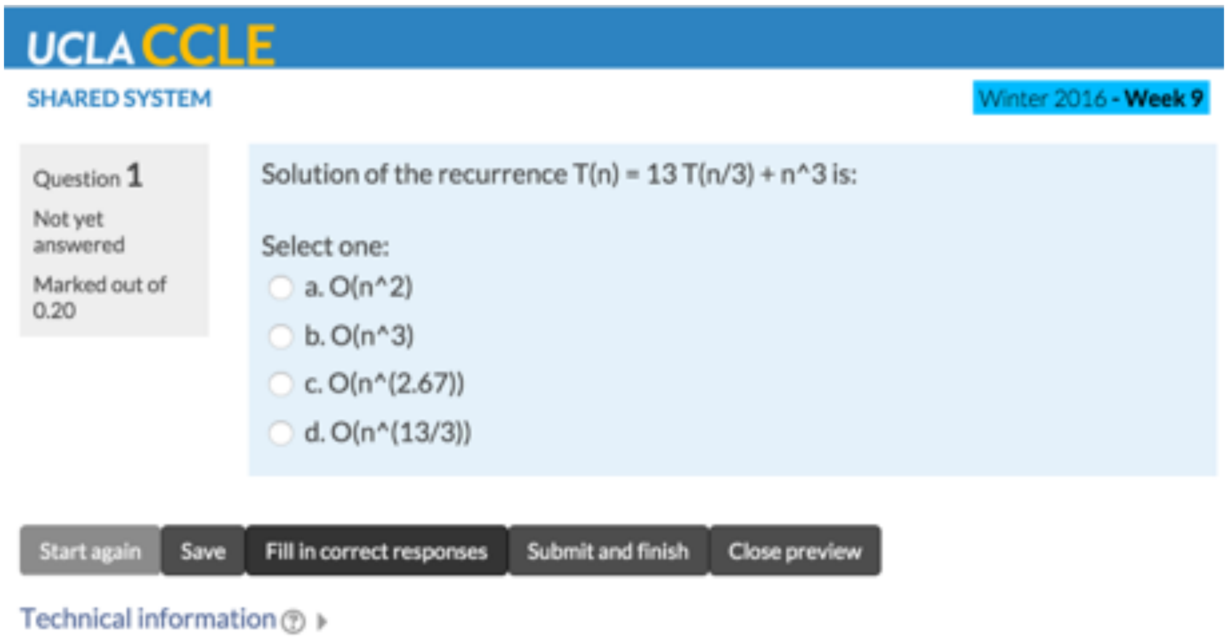
Grading: Collaboration

1. **Collaboration encouraged:**
   Must write your own answers.

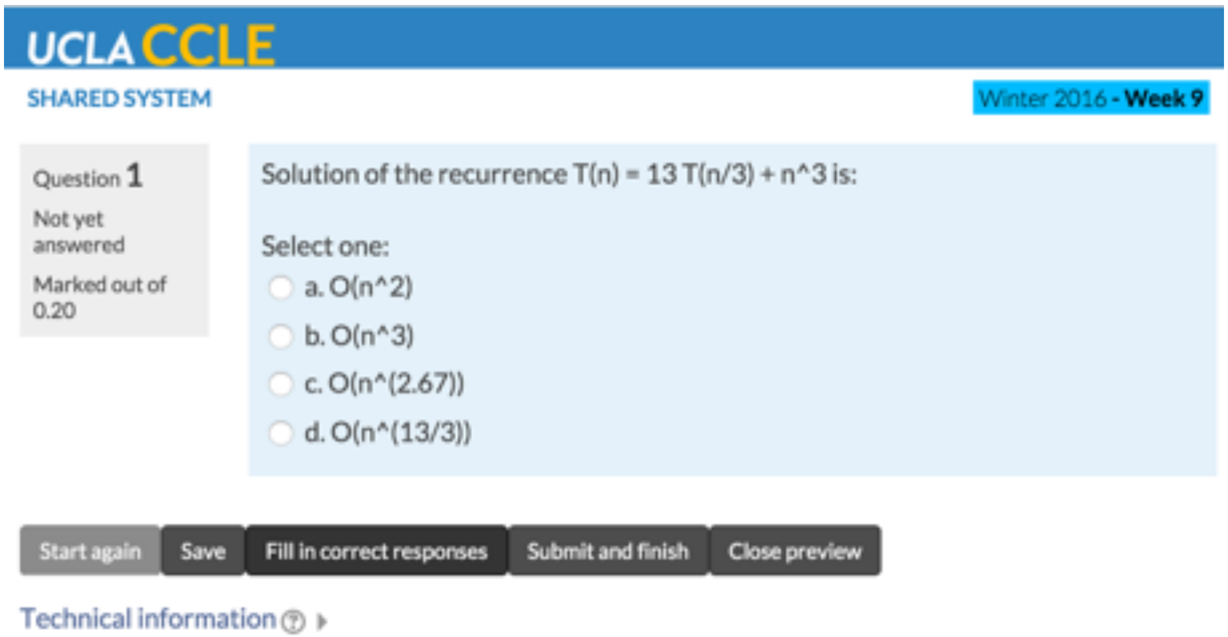
2. **Form study-groups**

3. **Attempting homework honestly**
   => Do well in exams.

Grading: Quizzes

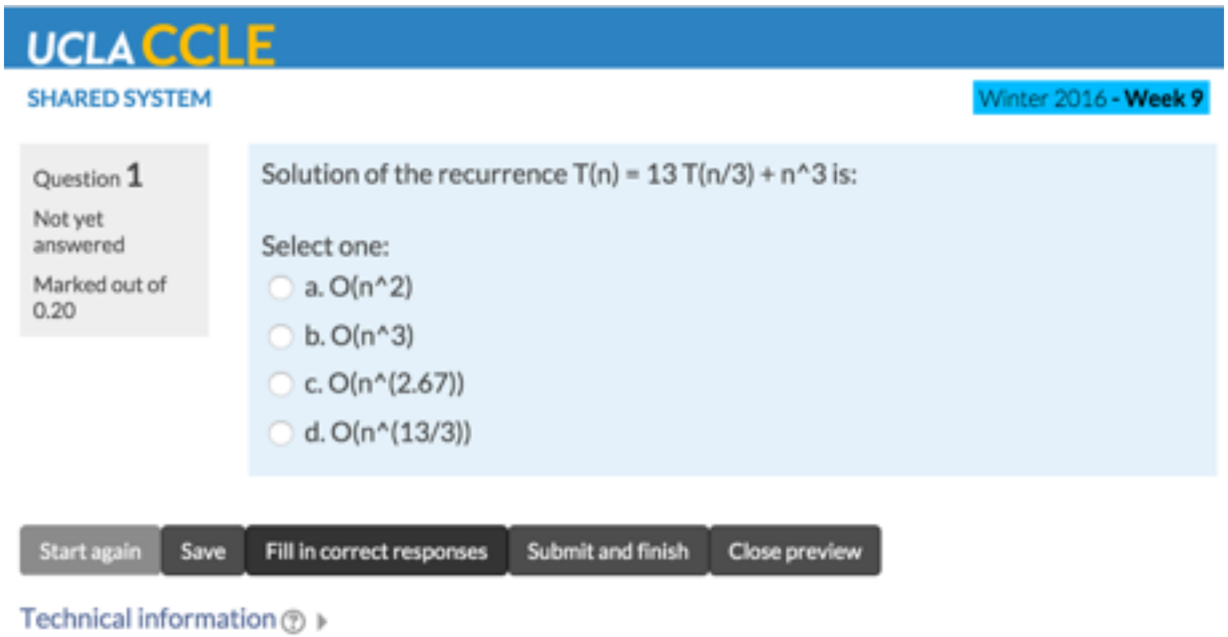


Grading: Quizzes



1. Live on CCLE from **W 6:00 PM** to **F 5:59PM**

Grading: Quizzes



1. Live on CCLE from **W 6:00 PM** to **F 5:59PM**

2. Less than five minutes; **1 point.**

Lesson plan

CS180W17

Today Monday, January 9 Week Month Agenda

Showing events after 12/1. Look for earlier events

Monday, January 9	
12:00pm	Course logistics. What is an algorithm?
Wednesday, January 11	
12:00pm	Divide and Conquer
Wednesday, January 18	
12:00pm	Divide and Conquer. Assignment 1.
Monday, January 23	
12:00pm	Graphs, connectivity. BFS
Wednesday, January 25	
12:00pm	DFS. Finding cycles. Assignment 2
Monday, January 30	
12:00pm	Greedy algorithms.
Wednesday, February 1	
12:00pm	Greedy algorithms.
Friday, February 3	
12:00pm	Exam 1

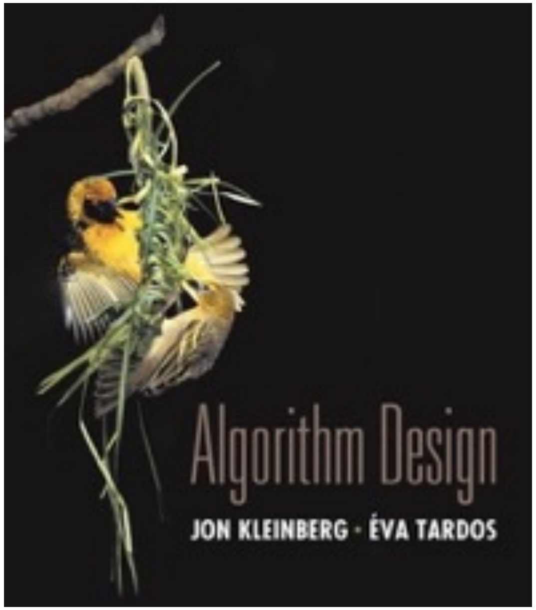
Events shown in time zone: Pacific Time Google Calendar

Detailed Calendar: www.cs180.raghumeka.org

Resources

Required

Very good: read it!



Resources

Ask questions: Don't be convinced easily.

Resources

Ask questions: Don’t be convinced easily.

(especially about the WHY!)

*

Resources

Ask questions: Don't be convinced easily.

(especially about the WHY!)

Resources

Ask questions: Don't be convinced easily.

(especially about the WHY!)

Office hours: high-latency, high-bandwidth

Resources

Ask questions: Don’t be convinced easily.

(especially about the WHY!)

Office hours: high-latency, high-bandwidth

Piazza: low-latency, low-bandwidth

Resources

Ask questions: Don't be convinced easily.

(especially about the WHY!)

Office hours: high-latency, high-bandwidth

Piazza: low-latency, low-bandwidth

Tell us about how to make the course better

Plan for Today

- Logistics
- **What is an algorithm?**

Plan for Today

- Logistics
- **What is an algorithm?**

What is an algorithm?

“ A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation. ”

Merriam-Webster

What is an algorithm?

“ A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation. ”

“ An algorithm is a finite, definite, effective procedure, with some input and some output. ” — Donald Knuth

First algorithm you ever saw/did …

First algorithm you ever saw/did …

Multiplication:

First algorithm you ever saw/did …

Multiplication:

$$\begin{array}{r} 1\ 2\ 3\ 4\ 5\ 6 \\ \times \quad\ \ 2\ 0\ 1\ 6 \end{array}$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
1\ 2\ 3\ 4\ 5\ 6 \\
\times \quad 2\ 0\ 1\ 6 \\
\hline
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{1\;2\;3\;4\;5\;6} \\
\times \quad 2\;0\;1\;\boxed{6} \\
\hline
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$\begin{array}{r} \boxed{1\ 2\ 3\ 4\ 5\ 6} \\ \times \quad 2\ 0\ 1\ \boxed{6} \\ \hline 7\ 4\ 0\ 7\ 3\ 6 \end{array}$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{1\ 2\ 3\ 4\ 5\ 6} \\
\times \quad 2\ 0\ \boxed{1}\ 6 \\
\hline
7\ 4\ 0\ 7\ 3\ 6
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rcccccc}
 & \boxed{1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6} \\
\times & 2\ \ 0\ \ \boxed{1}\ \ 6 \\
\hline
 & 7\ \ 4\ \ 0\ \ 7\ \ 3\ \ 6 \\
1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ + &
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

```
     1 2 3 4 5 6
x        2 0 1 6
----------------
     7 4 0 7 3 6
   1 2 3 4 5 6 +
```

First algorithm you ever saw/did …

Multiplication:

```
         1 2 3 4 5 6
       x     2 0 1 6
       -------------
         7 4 0 7 3 6
       1 2 3 4 5 6 +
     0 0 0 0 0 0 +
```

First algorithm you ever saw/did …

Multiplication:

```
          1 2 3 4 5 6
        x     2 0 1 6
        -------------
          7 4 0 7 3 6
        1 2 3 4 5 6 +
    0 0 0 0 0 0 +
```

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{1\ 2\ 3\ 4\ 5\ 6} \\
\times \quad 2\ 0\ 1\ 6 \\
\hline
7\ 4\ 0\ 7\ 3\ 6 \\
1\ 2\ 3\ 4\ 5\ 6\ + \\
0\ 0\ 0\ 0\ 0\ 0\ + \\
2\ 4\ 6\ 9\ 1\ 2\ +
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

			1	2	3	4	5	6
			x		2	0	1	6
			7	4	0	7	3	6
		1	2	3	4	5	6	+
	0	0	0	0	0	0	+	
2	4	6	9	1	2	+		
								6

First algorithm you ever saw/did …

Multiplication:

```
            1 2 3 4 5 6
            x     2 0 1 6
          ---------------
            7 4 0 7 3 6
          1 2 3 4 5 6 +
        0 0 0 0 0 0 +
      2 4 6 9 1 2 +
    ---------------------
                  9 6
```

First algorithm you ever saw/did …

Multiplication:

			1	2	3	4	5	6
			x		2	0	1	6
			7	4	0	7	3	6
		1	2	3	4	5	6	+
	0	0	0	0	0	0	+	
2	4	6	9	1	2	+		
					1			
						2	9	6

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{ccccccccc}
 & & & 1 & 2 & 3 & 4 & 5 & 6 \\
 & & & \times & & 2 & 0 & 1 & 6 \\
\hline
 & & & 7 & 4 & 0 & 7 & 3 & 6 \\
 & & 1 & 2 & 3 & 4 & 5 & 6 & + \\
 & 0 & 0 & 0 & 0 & 0 & 0 & + & \\
2 & 4 & 6 & 9 & 1 & 2 & + & & \\
 & & & & & 1 & & & \\
\hline
 & & & & & 7 & 2 & 9 & 6
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{ccccccccc}
 & & & \boxed{1\;2\;3\;4\;5\;6} \\
 & & & \times & & 2 & 0 & 1 & 6 \\
\hline
 & & & 7 & 4 & 0 & 7 & 3 & 6 \\
 & & 1 & 2 & 3 & 4 & 5 & 6 & + \\
 & 0 & 0 & 0 & 0 & 0 & 0 & + & \\
2 & 4 & 6 & 9 & 1 & 2 & + & & \\
 & & & & & 1 & & & \\
\hline
2 & 4 & 8 & 8 & 8 & 7 & 2 & 9 & 6
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rccccccccc}
 & & & & 1 & 2 & 3 & 4 & 5 & 6 \\
 & & & & \times & & 2 & 0 & 1 & 6 \\
\hline
 & & & & 7 & 4 & 0 & 7 & 3 & 6 \\
 & & & 1 & 2 & 3 & 4 & 5 & 6 & + \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & + & \\
 & 2 & 4 & 6 & 9 & 1 & 2 & + & & \\
 & & & & & & 1 & & & \\
\hline
\text{ANSWER} = & 2 & 4 & 8 & 8 & 8 & 7 & 2 & 9 & 6
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

First algorithm you ever saw/did …

Multiplication:

	a1	a2	a3	a4	…	an
x	b1	b2	b3	b4	…	bn

First algorithm you ever saw/did …

Multiplication:

```
     a1  a2  a3  a4  …          an
  x  b1  b2  b3  b4  …          bn
  ------------------------------------
```

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{cccccccc}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & & an} \\
x & b1 & b2 & b3 & b4 & \ldots & & \boxed{bn} \\
\hline
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rcccccc}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & an} \\
x & b1 & b2 & b3 & b4 & \ldots & \boxed{bn} \\
\hline
c1\ c2\ c3 & & & & & \ldots & c_{n+1}
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$\begin{array}{ccccccccc} & & \boxed{a1 & a2 & a3 & a4 & \ldots & & an} \\ & x & b1 & b2 & b3 & b4 & \ldots & \boxed{} & bn \\ \hline c1 & c2 & c3 & & & & \ldots & & c_{n+1} \end{array}$$

First algorithm you ever saw/did …

Multiplication:

a1 a2 a3 a4 … an

x b1 b2 b3 b4 … bn

c1 c2 c3 … c_{n+1}

… +

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rccccccc}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & & an} \\
\times & b1 & b2 & b3 & b4 & \ldots & & bn \\
\hline
c1\ c2\ c3 & & & & & & \ldots & c_{n+1} \\
 & & & & & & \ldots & +
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rlllll}
 & \boxed{a1 \quad a2 \quad a3 \quad a4 \quad \ldots \quad an} \\
\times & b1 \quad b2 \quad b3 \quad b4 \quad \ldots \quad bn \\
\hline
c1\ c2\ c3 & \ldots \quad c_{n+1} \\
 & \ldots \quad + \\
 & \ldots \quad +
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{lllllll}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & an} \\
\times & b1 & b2 & b3 & b4 & \ldots & bn \\
\hline
\end{array}
$$

c1 c2 c3 … c_{n+1}

… +

… +

⋮

n rows

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{\quad a_1 \;\; a_2 \;\; a_3 \;\; a_4 \;\; \ldots \qquad\qquad an} \\
\times \;\; b_1 \;\; b_2 \;\; b_3 \;\; b_4 \;\; \ldots \qquad\qquad bn \\
\hline
c_1 \;\; c_2 \;\; c_3 \qquad\qquad\qquad\qquad\qquad\qquad \ldots \quad c_{n+1} \\
\ldots \quad + \;\; \\
\ldots \quad + \qquad \\
\vdots \qquad\qquad\qquad\qquad \\
\hline
\end{array}
\quad \Big\updownarrow \text{ n rows}
$$

ANSWER = $\boxed{\qquad\qquad\qquad\qquad\qquad\qquad}$

First algorithm you ever saw/did …

Multiplication
INPUT: Two n-digit numbers a, b
OUTPUT: (a x b) in decimal format.

First algorithm you ever saw/did …

Multiplication
INPUT: Two n-digit numbers a, b
OUTPUT: (a x b) in decimal format.

How efficient is the previous algorithm?


EFFICIENCY
LEVEL
per cent
meter
%

What is efficiency?

What is efficiency?

Answer: Analysis of algorithms.

*

Defining Efficiency

Defining Efficiency

Runs fast on typical real-world inputs?

Defining Efficiency

Runs fast on typical real-world inputs?

PROS
Enough in practice
Easier to satisfy

Defining Efficiency

Runs fast on typical real-world inputs?

PROS	CONS
Enough in practice	Hard to quantify
Easier to satisfy	Exceptions matter!
	“Real-world” is a moving target

Efficiency

Want a general theory of efficiency that is

Efficiency

Want a general theory of efficiency that is

1. **Simple** (mathematically)

Efficiency

Want a general theory of efficiency that is

1. **Simple** (mathematically)
2. **Objective** (doesn't depend on i5 vs i7 processor)

Efficiency

Want a general theory of efficiency that is

1. **Simple** (mathematically)
2. **Objective** (doesn't depend on i5 vs i7 processor)
3. **Captures scalability** (input-sizes change)

Efficiency

Want a general theory of efficiency that is

1. **Simple** **(mathematically)**
2. **Objective** **(doesn't depend on i5 vs i7 processor)**
3. **Captures scalability** **(input-sizes change)**
4. **Predictive of practical performance**
   - "theoretically bad" algorithms should be bad in practice and vice versa (usually)

Measuring efficiency

Most important resource in computing:

Measuring efficiency

Most important resource in computing: TIME

Measuring efficiency

Most important resource in computing: TIME

TIME: # of instructions executed in a simple programming language

Measuring efficiency

Most important resource in computing: TIME

TIME: # of instructions executed in a simple programming language

- Only simple operations (+,*,-,=,if,call,…)

Measuring efficiency

Most important resource in computing: TIME

TIME: # of instructions executed in a simple programming language

- Only simple operations (+,*,-,=,if,call,…)
- Each operation takes one time step

Measuring efficiency

Most important resource in computing: TIME

TIME: # of instructions executed in a simple programming language

- Only simple operations (+,*,-,=,if,call,…)
- Each operation takes one time step
- Each memory access takes one time step

Measuring efficiency

Most important resource in computing: TIME

TIME: # of instructions executed in a simple programming language

- Only simple operations (+,*,-,=,if,call,…)
- Each operation takes one time step
- Each memory access takes one time step
- No fancy stuff built in (add these two matrices, copy this long string,…)

We left out things but...

Things we've dropped

- memory hierarchy
  - disk, caches, registers have many orders of magnitude differences in access time
- not all instructions take the same time in practice (+, ÷)
- communication
- different computers have different primitive instructions

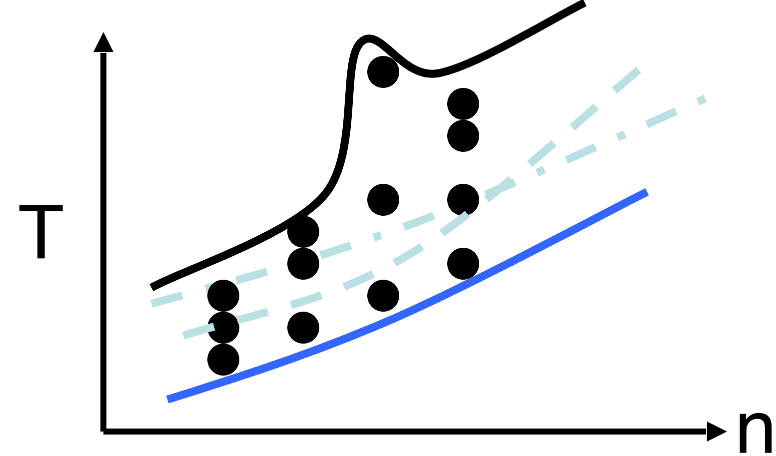
However,

- one can usually tune implementations so that the hierarchy, etc., is not a huge factor

Problem

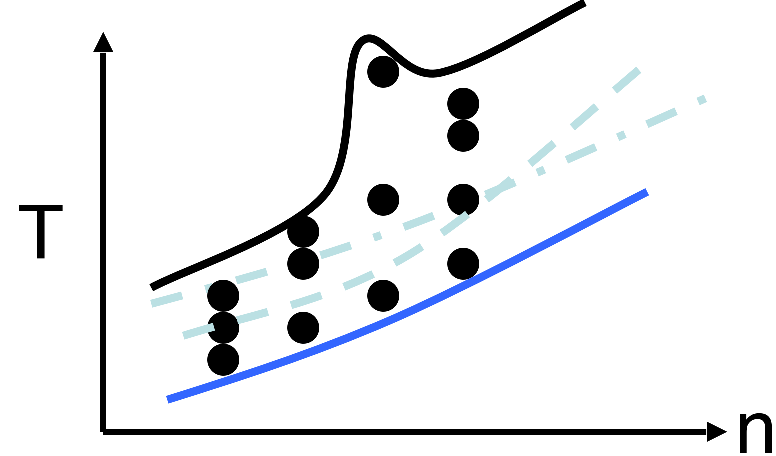
- Algorithms can have different running times on different inputs.
- Smaller inputs take less time, larger inputs take more time.

Solution



Measure performance on input size n

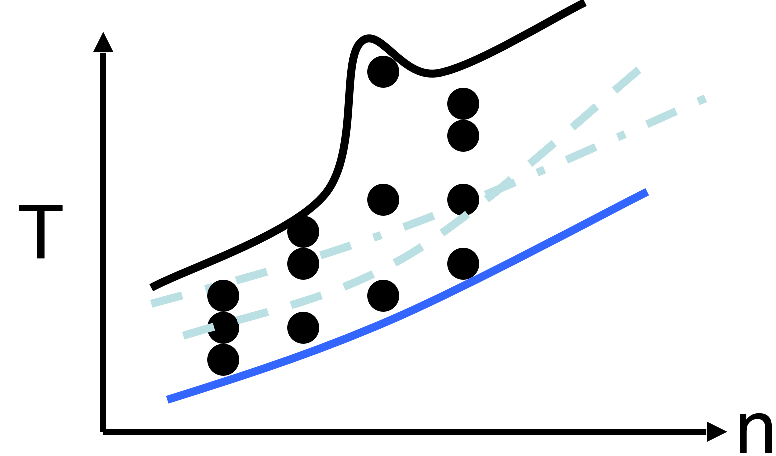
Solution



Measure performance on input size n

Average-case complexity: Average # steps algorithm takes on inputs of size n

Solution



Measure performance on input size n

Average-case complexity: Average # steps algorithm takes on inputs of size n

Worst-case complexity: Max # steps algorithm takes **over all** inputs of size n

Pros and cons:

Average-case

- Over what probability distribution? (different people may have different "average" problems)
- Analysis often hard

Worst-case

+ A fast algorithm has a comforting guarantee
+ Analysis easier than average-case
+ Useful in real-time applications (space shuttle, nuclear reactors)
- May be too pessimistic

Best-case …

Best-case …

Characterize *growth-rate* of (worst-case) run time as a function of problem size, up to a constant factor

Best-case …

Characterize *growth-rate* of (worst-case) run time as a function of problem size, up to a constant factor

Why not try to be more precise?

- Technological variations (computer, compiler, OS, …) easily 10x or more

Complexity

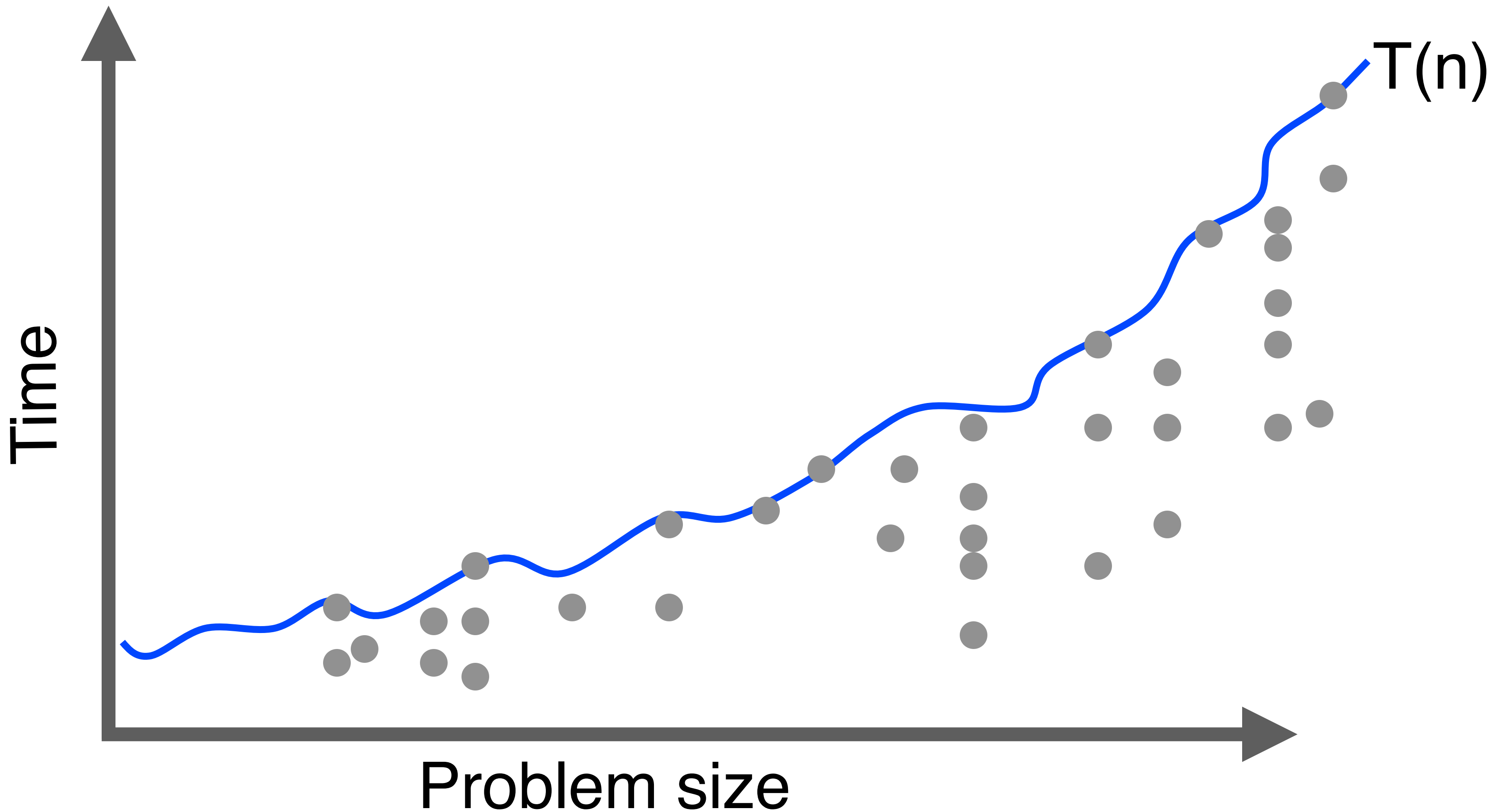
The *complexity* of an algorithm associates a number *T(n)* with each problem size *n:*

T(n) = worst-case time taken on problems of size *n*.

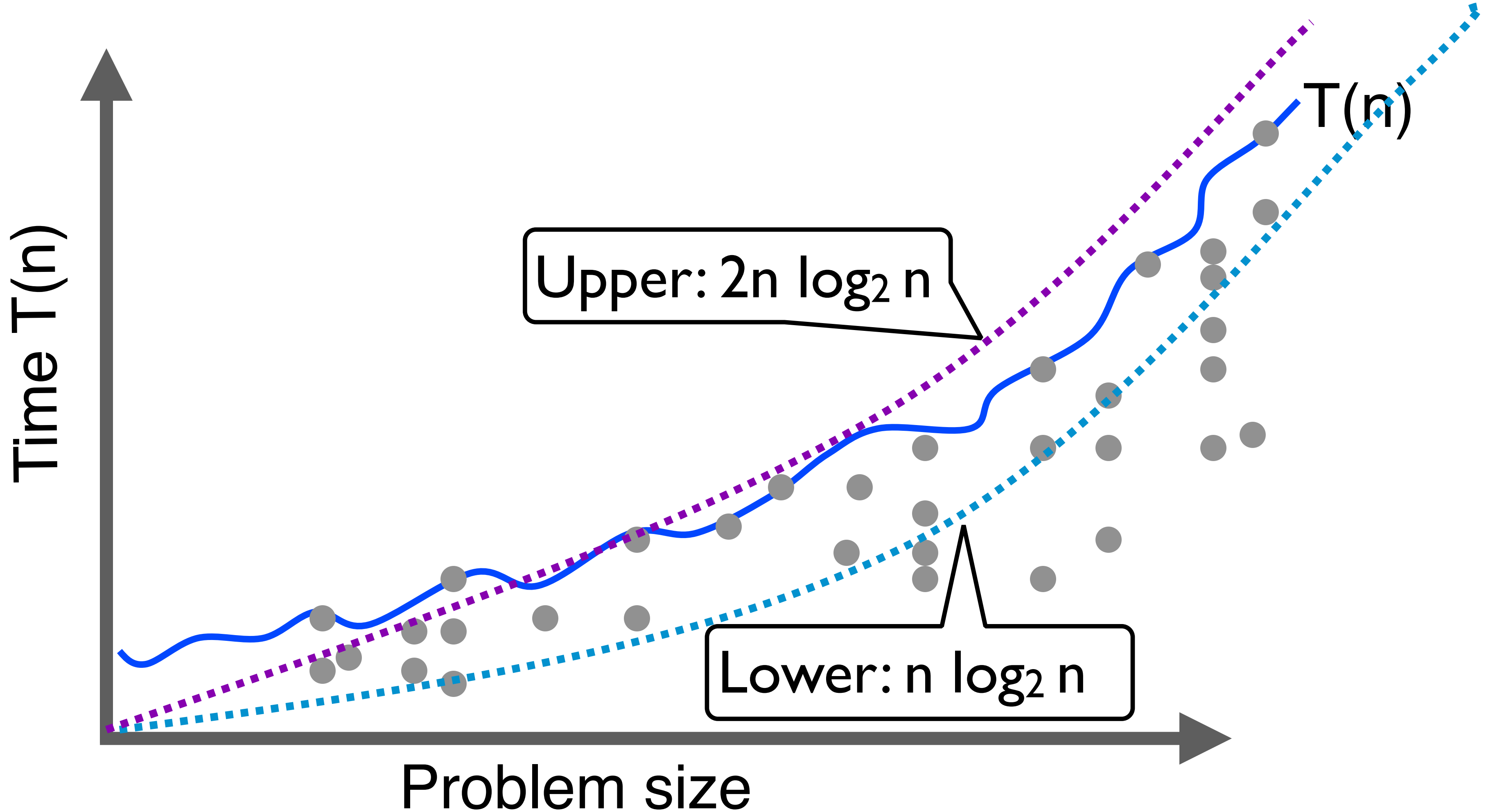
Mathematically: $T : \mathbb{N} \to \mathbb{R}^+$

i.e., *T* is a function that maps positive integers (problem sizes) to positive real numbers (number of steps).

Complexity



Complexity



Asymptotic analysis

Methodology for comparing run-times

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = O(g(n)): iff there is a constant c>0 so that
f(n) is eventually-always **at most** c g(n)

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = O(g(n)): iff there is a constant c>0 so that f(n) is eventually-always **at most** c g(n)

f(n) = Ω(g(n)): iff there is a constant c>0 so that f(n) is eventually-always **at least** c g(n)

Asymptotic analysis

Methodology for comparing run-times

Given two functions $f, g : \mathbb{N} \to \mathbb{R}^+$

f(n) = O(g(n)): iff there is a constant c>0 so that f(n) is eventually-always **at most** c g(n)

f(n) = Ω(g(n)): iff there is a constant c>0 so that f(n) is eventually-always **at least** c g(n)

f(n) = Θ(g(n)): iff both hold - there are constants c_1, c_2>0 so that eventually always $c_1 g(n) < f(n) < c_2 g(n)$

Examples

$10n^2$-$16n$+100 is $O(n^2)$

$10n^2-16n+100 < 10n^2$ for all $n > 10$

$10n^2$-$16n$+100 is $\Omega(n^2)$

$10n^2-16n+100 > n^2$ for all $n > 10$

Therefore also $10n^2-16n+100$ is $\Theta(n^2)$

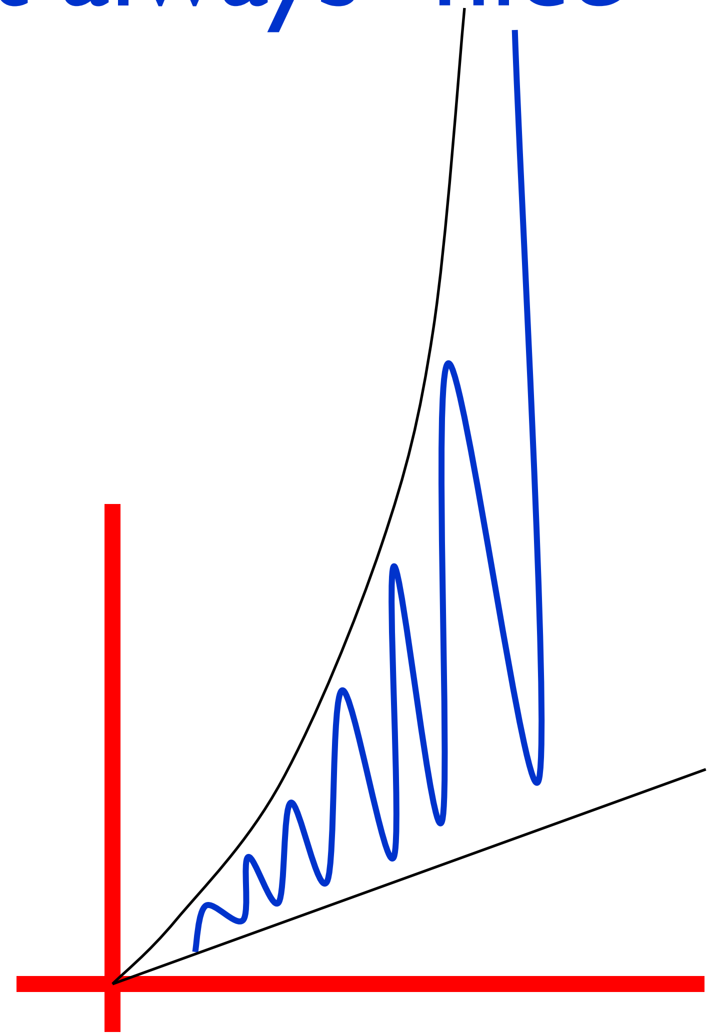
$10n^2$-$16n$+100 is not $O(n)$ also not $\Omega(n^3)$

Big-Theta, etc. not always "nice"

$$f(n) = \begin{cases} n^2, & n \text{ even} \\ n, & n \text{ odd} \end{cases}$$

f(n) is not $\Theta(n^a)$ for any *a*.

Fortunately, such nasty cases are rare



Polynomial time

P: Running time is $O(n^d)$ for some constant d independent of the input size n.

NICE SCALING: Doubling problem-size, increases time by a constant factor c (e.g., c ~ 2^d)

Contrast with exponential: Doubling problem-size can square the run-time!

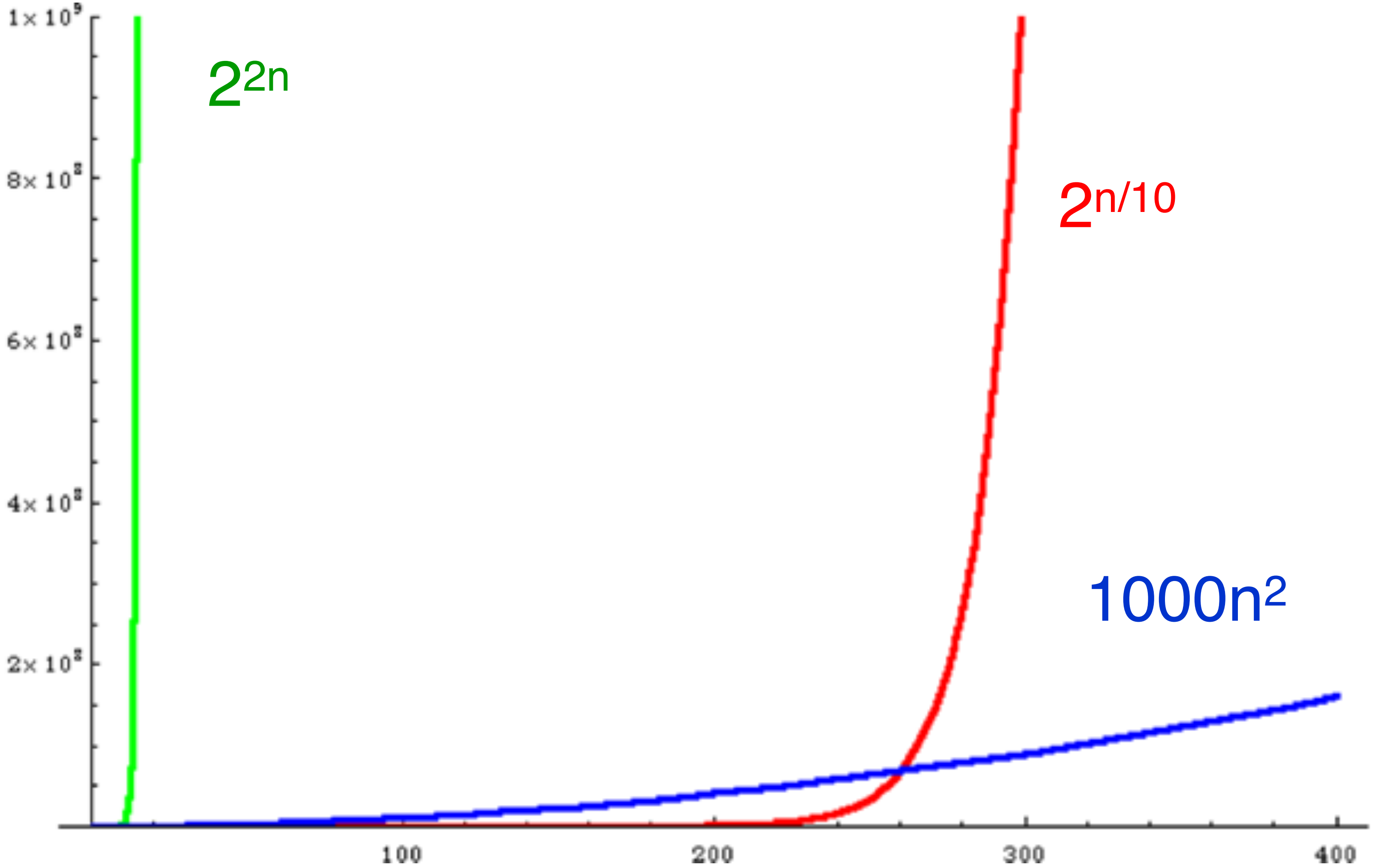
$$(\text{e.g., } T(n) = 2^{n/10} \text{ vs } T(2n) = 2^{2(n/10)} = T(n)^2)$$

Polynomial time

P: Running time is $O(n^d)$ for some constant d independent of the input size n.

Behaves well under composition: if algorithm has a polynomial running time with polynomial number of calls to a subroutine with polynomial running time, then overall running time is still polynomial.

polynomial vs exponential growth



Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

not only get very big, but do so *abruptly*, which likely yields erratic performance on small instances

another view of poly vs exp

Next year's computer will be 2x faster. If I can solve problem of size n_0 today, how large a problem can I solve in the same time next year?

Complexity	Increase	E.g. $T = 10^{12}$
$O(n)$	$n_0 \rightarrow 2n_0$	$10^{12} \rightarrow 2 \times 10^{12}$
$O(n^2)$	$n_0 \rightarrow \sqrt{2}\, n_0$	$10^{6} \rightarrow 1.4 \times 10^{6}$
$O(n^3)$	$n_0 \rightarrow \sqrt[3]{2}\, n_0$	$10^{4} \rightarrow 1.25 \times 10^{4}$
$2^{n/10}$	$n_0 \rightarrow n_0+10$	$400 \rightarrow 410$
2^n	$n_0 \rightarrow n_0+1$	$40 \rightarrow 41$

Summary

Typical initial goal for algorithm analysis is to find a

- reasonably tight ← i.e., Θ if possible
- *asymptotic* ← i.e., O or Θ
- bound on ← usually upper bound
- *worst case* running time
- as a function of problem *size*

This is rarely the last word, but often helps separate good algorithms from blatantly bad ones - so you can concentrate on the good ones!

Complexity of multiplication

Multiplication
INPUT: Two n-digit numbers a, b
OUTPUT: (a x b) in decimal format.

Complexity of multiplication

Multiplication
INPUT: Two n-digit numbers a, b
OUTPUT: (a x b) in decimal format.

How efficient is grade-school algorithm?

First algorithm you ever saw/did …

Multiplication:

First algorithm you ever saw/did …

Multiplication:

```
    a1  a2  a3  a4  …        an
x   b1  b2  b3  b4  …        bn
```

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rccccccc}
 & a1 & a2 & a3 & a4 & \ldots & & an \\
x & b1 & b2 & b3 & b4 & \ldots & & bn \\
\hline
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rcccccr}
 & a1 & a2 & a3 & a4 & \ldots & an \\
\times & b1 & b2 & b3 & b4 & \ldots & bn \\
\hline
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{\text{a1 a2 a3 a4 } \dots \text{ an}} \\
\text{x b1 b2 b3 b4 } \dots \ \boxed{\text{bn}} \\
\hline
\text{c1 c2 c3} \qquad \dots \ c_{n+1}
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rcccccccc}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & & an} \\
\times & b1 & b2 & b3 & b4 & \ldots & \boxed{} & bn \\
\hline
c1\ c2\ c3 & & & & & & \ldots & c_{n+1}
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rccccccc}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & & an} \\
\times & b1 & b2 & b3 & b4 & \ldots & \boxed{} & bn \\
\hline
c1 & c2 & c3 & & & & \ldots & c_{n+1} \\
 & & & & & & \ldots & +
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$\begin{array}{lllllll} & \boxed{a1 & a2 & a3 & a4 & \ldots & an} \\ \times & b1 & b2 & b3 & b4 & \ldots & bn \\ \hline \end{array}$$

$$c1 \; c2 \; c3 \quad \ldots \quad c_{n+1}$$

$$\ldots \quad +$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rccccccc}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & & an} \\
x & b1 & b2 & b3 & b4 & \ldots & & bn \\
\hline
c1\ c2\ c3 & & & & & & \ldots & c_{n+1} \\
 & & & & & & \ldots & + \\
 & & & & & \ldots & + &
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rcccccl}
 & \boxed{a1 & a2 & a3 & a4 & \ldots & an} \\
\times & b1 & b2 & b3 & b4 & \ldots & bn
\end{array}
$$

---

c1 c2 c3 … c_{n+1}

… +

… +

⋮

n rows

First algorithm you ever saw/did …

Multiplication:

a1 a2 a3 a4 … an

x b1 b2 b3 b4 … bn

c1 c2 c3 … c_{n+1}

… +

… +

⋮

n rows

ANSWER =

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{\quad a1 \;\; a2 \;\; a3 \;\; a4 \;\; \ldots \qquad an} \\
\times \;\; b1 \;\; b2 \;\; b3 \;\; b4 \;\; \ldots \qquad \boxed{bn} \\
\hline
c1 \;\; c2 \;\; c3 \qquad\qquad \ldots \quad c_{n+1} \\
\ldots \quad + \\
\ldots \quad + \quad \\
\vdots \qquad\qquad \\
\hline
\end{array}
$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{\quad a_1 \;\; a_2 \;\; a_3 \;\; a_4 \;\; \ldots \qquad\qquad a_n} \\
x \;\; b_1 \;\; b_2 \;\; b_3 \;\; b_4 \;\; \ldots \qquad\qquad \boxed{b_n} \\
\hline
c_1 \;\; c_2 \;\; c_3 \qquad\qquad\qquad\qquad\qquad \ldots \quad c_{n+1} \\
\ldots \quad + \\
\ldots \quad + \quad\; \\
\vdots \qquad\qquad\qquad \\
\hline
\end{array}
$$

$\longrightarrow$ n digit mults.

First algorithm you ever saw/did …

Multiplication:

$$\begin{array}{lcccccc}
 & \boxed{a1 \quad a2 \quad a3 \quad a4 \quad \ldots \quad an} \\
\times & b1 \quad b2 \quad b3 \quad b4 \quad \ldots \quad \boxed{bn} \\
\hline
c1 \; c2 \; c3 & \ldots \quad c_{n+1} \longrightarrow O(n) \text{ time.} \\
 & \ldots \quad + \\
 & \ldots \quad + \\
 & \vdots \\
\hline
\end{array}$$

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{r}
\boxed{\quad a1 \;\; a2 \;\; a3 \;\; a4 \;\; \ldots \qquad an} \\
x \;\; b1 \;\; b2 \;\; b3 \;\; b4 \;\; \ldots \qquad \boxed{bn} \\
\hline
c1 \;\; c2 \;\; c3 \qquad\qquad \ldots \quad c_{n+1} \\
\ldots \quad + \\
\ldots \quad + \quad\; \\
\vdots \qquad\qquad \\
\hline
\end{array}
$$

$\rightarrow$ O(n) time.

$\rightarrow$ O(n) time.

$\rightarrow$ O(n) time.

$\vdots$

$\rightarrow$ O(n) time.

First algorithm you ever saw/did …

Multiplication:

$$
\begin{array}{rccccccc}
 & a1 & a2 & a3 & a4 & \dots & & an \\
\times & b1 & b2 & b3 & b4 & \dots & & bn \\
\hline
c1 & c2 & c3 & & & & \dots & c_{n+1} \\
 & & & & & & \dots & + \\
 & & & & & \dots & + & \\
\hline
\end{array}
$$

$\longrightarrow$ O(n) time.

$\longrightarrow$ O(n) time.

$\longrightarrow$ O(n) time.

⋮

$\longrightarrow$ O(n) time.

First algorithm you ever saw/did …

Multiplication:

a1 a2 a3 a4 … an

x b1 b2 b3 b4 … bn

c1 c2 c3 … c_{n+1} → O(n) time.

… + → O(n) time.

… + → O(n) time.

⋮

→ O(n) time.

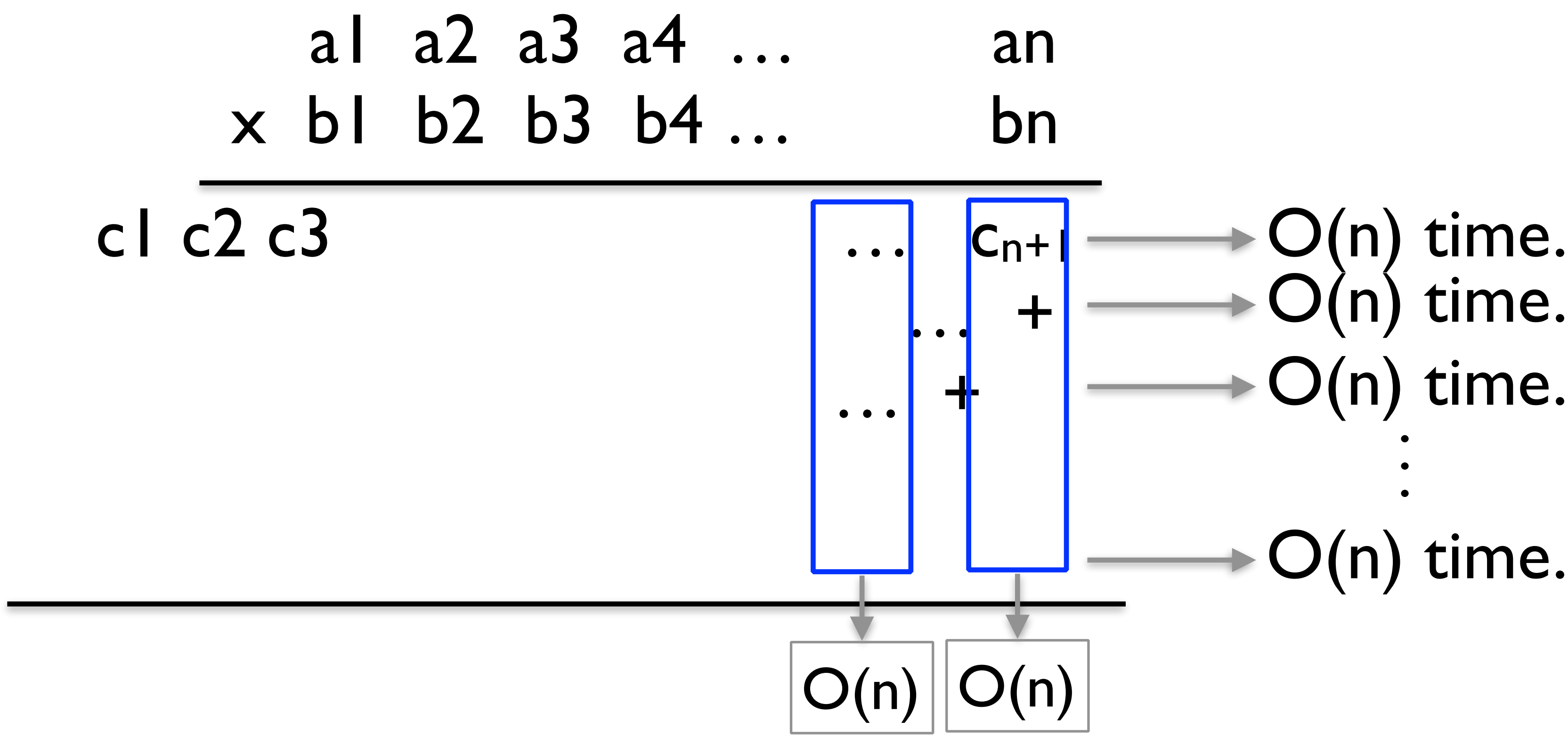
First algorithm you ever saw/did …

Multiplication:

```
          a1  a2  a3  a4  …          an
      x   b1  b2  b3  b4  …          bn
   ------------------------------------
c1 c2 c3                    …   cn+1     O(n) time.
                              …  +       O(n) time.
                            …  +         O(n) time.
                                              :
                                         O(n) time.
   ------------------------------------
                                O(n)
```

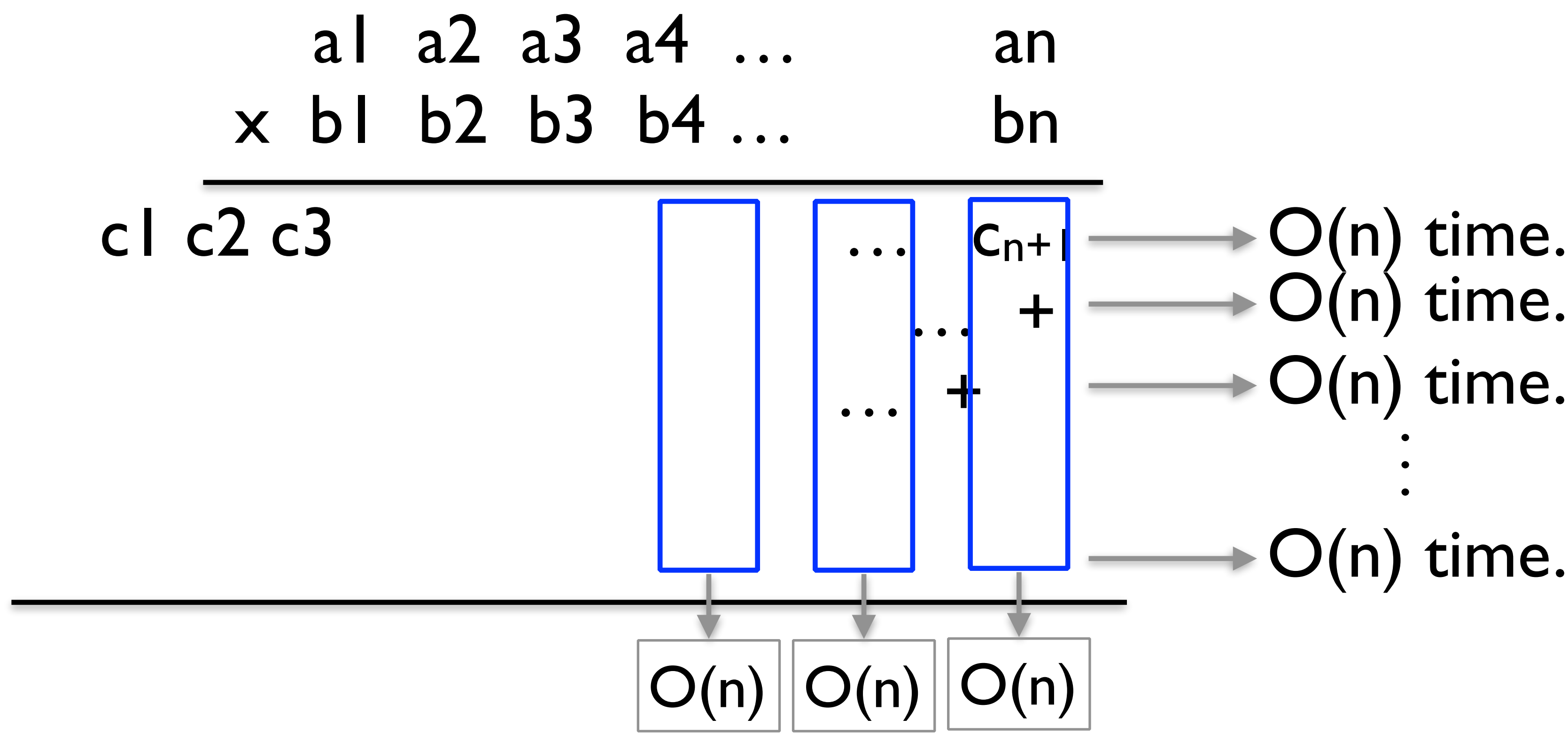
First algorithm you ever saw/did …

Multiplication:



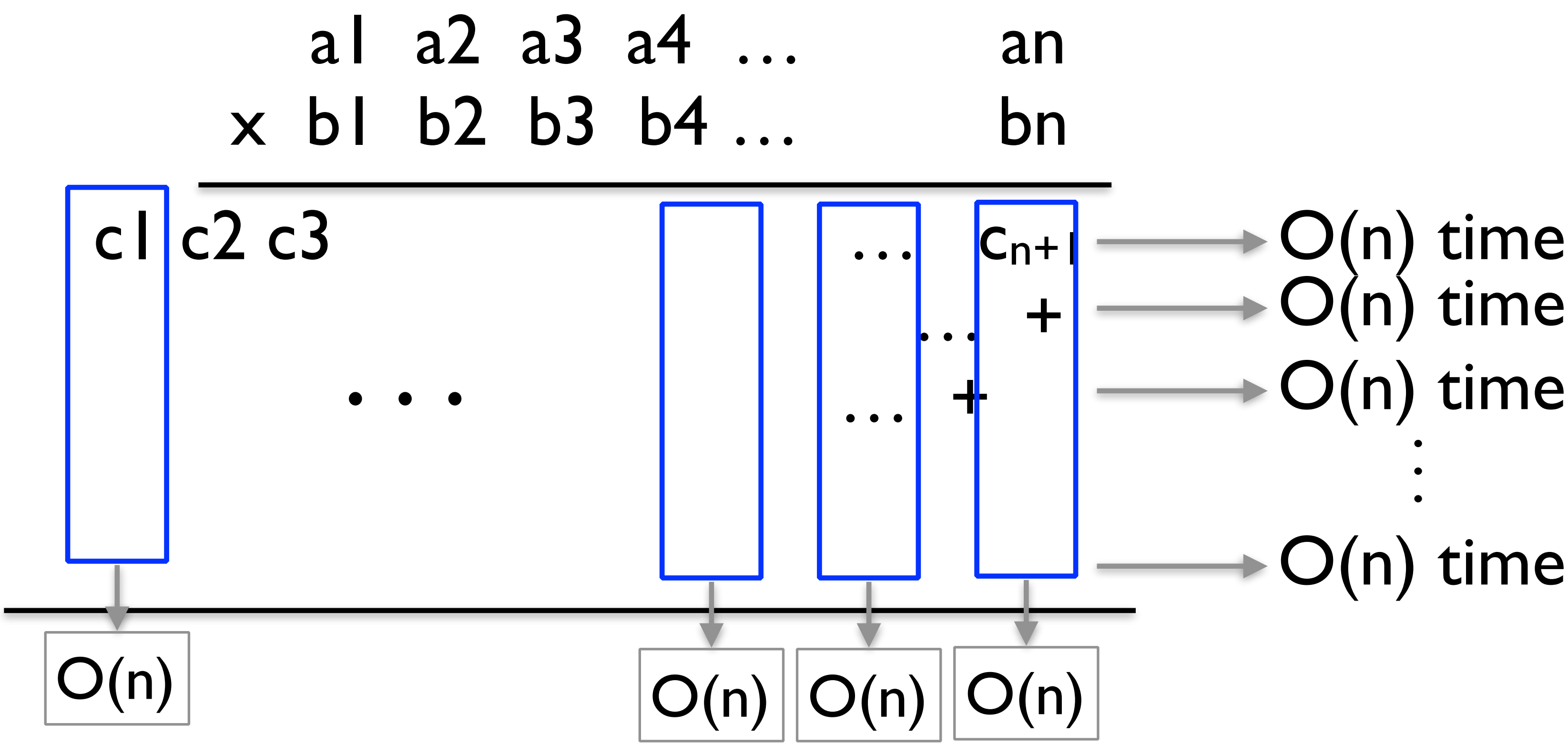
First algorithm you ever saw/did …

Multiplication:



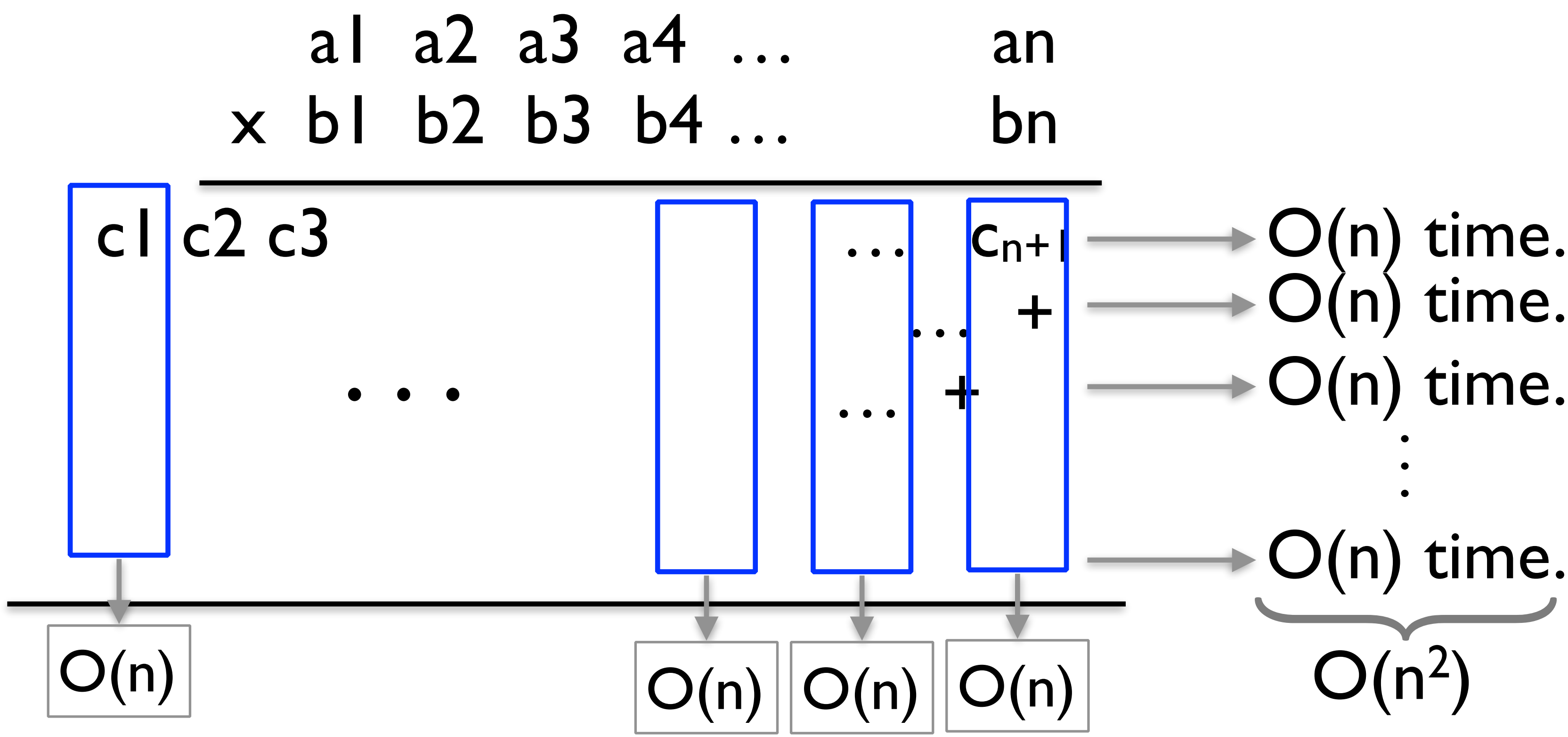
First algorithm you ever saw/did …

Multiplication:



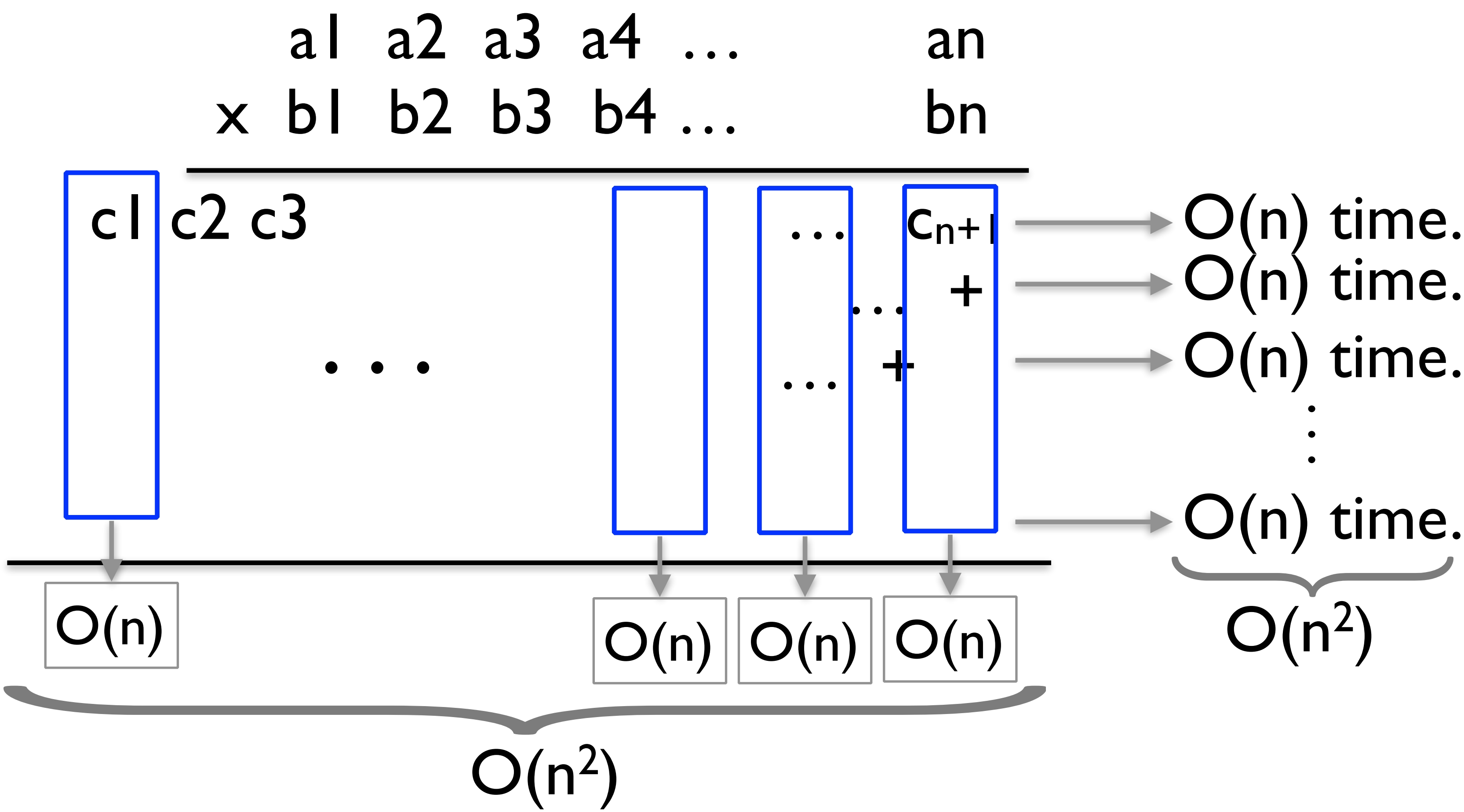
First algorithm you ever saw/did …

Multiplication:



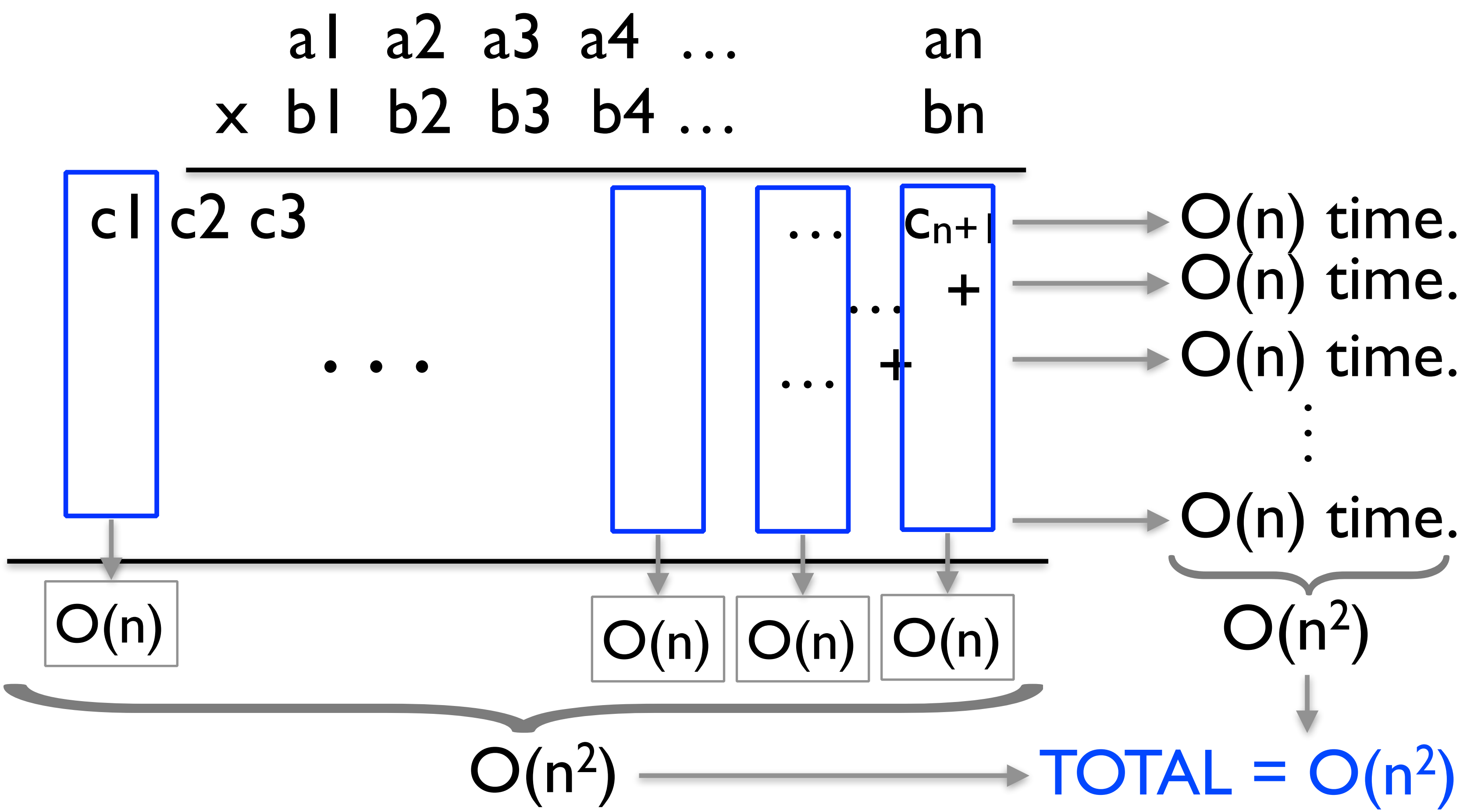
First algorithm you ever saw/did …

Multiplication:



First algorithm you ever saw/did …

Multiplication:



Complexity of multiplication

Multiplication
INPUT: Two n-digit numbers a, b
OUTPUT: (a x b) in decimal format.

Complexity of multiplication

Multiplication
INPUT: Two n-digit numbers a, b
OUTPUT: (a x b) in decimal format.

Grade-school algorithm takes $O(n^2)$ time.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Divide-and-conquer paradigm

Divide-and-conquer.

- Divide problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Most common usage.

- Divide size n into two subproblems of size n/2 in linear time.
- Solve two subproblems recursively.
- Combine two solutions into overall solution in linear time.

RUN-TIME = O(n log n)!

Sorting problem

Problem. Given a list of n elements from a totally-ordered universe, rearrange them in ascending order.

Sorting applications

Obvious applications.

- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

Sorting applications

Obvious applications.

- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

Some problems become easier once elements are sorted.

- Identify statistical outliers.
- Binary search in a database.
- Remove duplicates in a mailing list.

Sorting applications

Non-obvious applications.

- Convex hull.
- Closest pair of points.
- Interval scheduling / interval partitioning.
- Minimum spanning trees (Kruskal's algorithm).
- Scheduling

Sorting

input

A	L	G	O	R	I	T	H	M	S

Sorting

- INPUT: Array

input

A	L	G	O	R	I	T	H	M	S

Sorting

- INPUT: Array
- OUTPUT: Sorted array

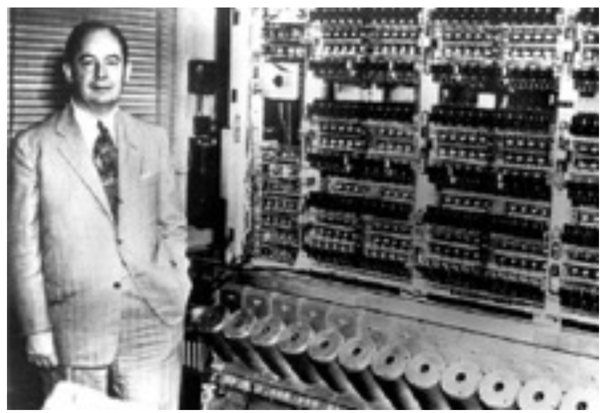
input

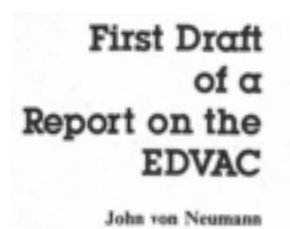
A	L	G	O	R	I	T	H	M	S

output

A	G	H	I	L	M	O	R	S	T

Why Mergesort?



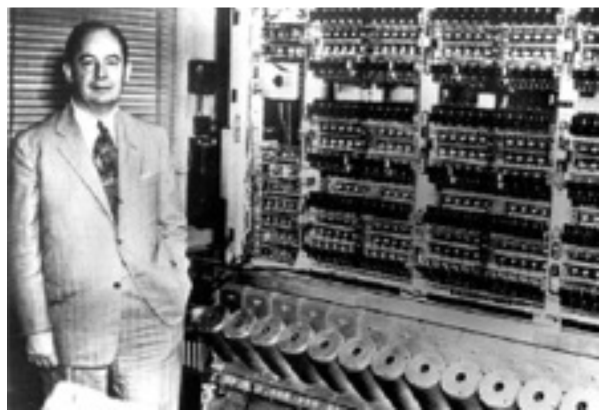


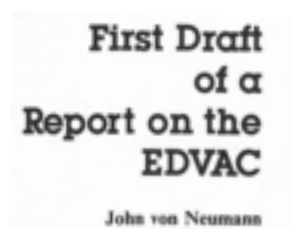
First Draft
of a
Report on the
EDVAC

John von Neumann

Why Mergesort?

- Developed in 1945 by von Neumann

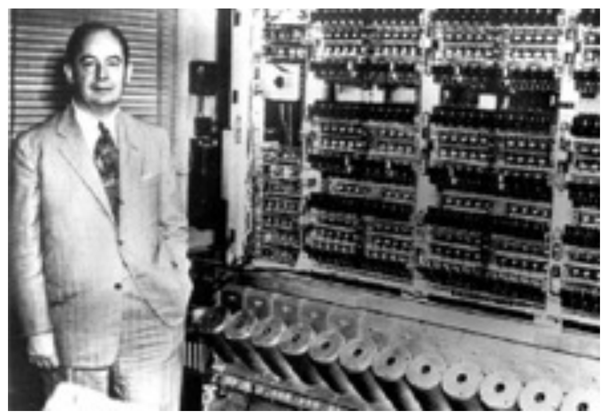


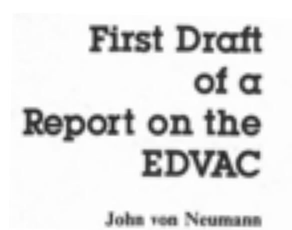

First Draft
of a
Report on the
EDVAC

John von Neumann

Why Mergesort?

- Developed in 1945 by von Neumann
- Much faster than “Selection”, “Insertion”, “Bubble”



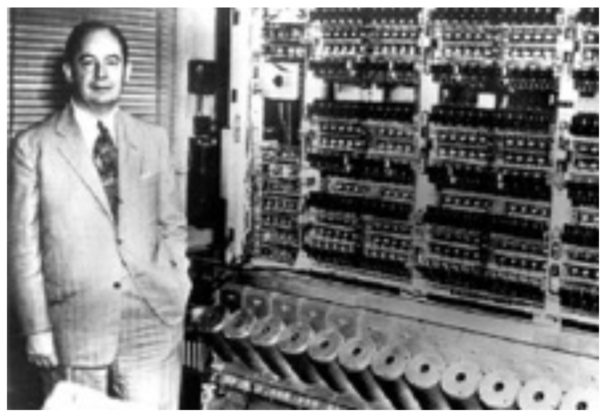


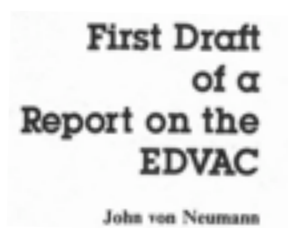
First Draft
of a
Report on the
EDVAC

John von Neumann

Why Mergesort?

- Developed in 1945 by von Neumann
- Much faster than “Selection”, “Insertion”, “Bubble”
- Sorting in Perl, Java, Python, Android:
  - Hybrid of Mergesort and others.




First Draft
of a
Report on the
EDVAC

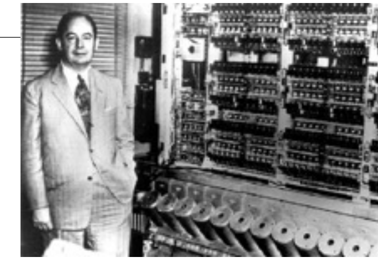
John von Neumann

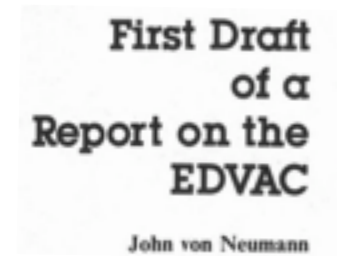
Why Mergesort?

- Developed in 1945 by von Neumann
- Much faster than “Selection”, “Insertion”, “Bubble”
- Sorting in `Perl`, `Java`, `Python`, `Android`:
  - Hybrid of Mergesort and others.

Excellent example of principles of Divide & Conquer

Mergesort

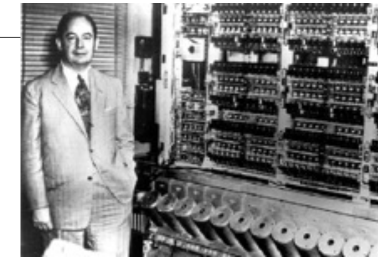


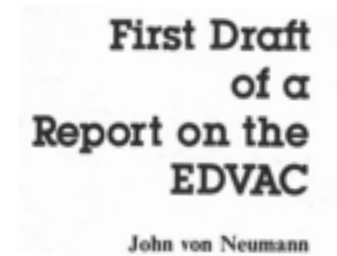

First Draft
of a
Report on the
EDVAC

John von Neumann

A	L	G	O	R	I	T	H	M	S

Mergesort





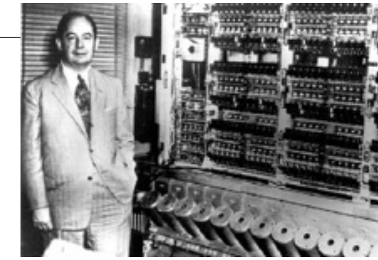
- Divide array into two halves.

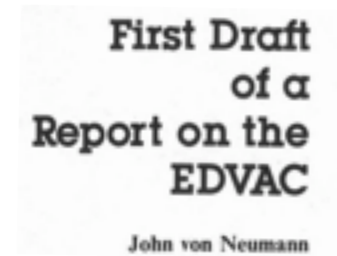
A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

Mergesort





- Divide array into two halves.
- Recursively sort left half.

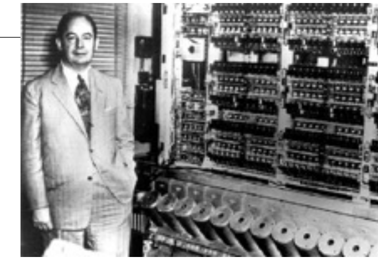
A	L	G	O	R	I	T	H	M	S

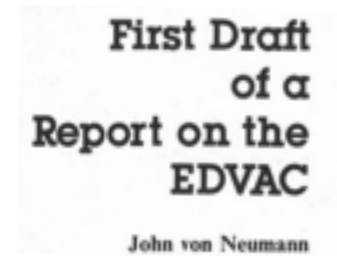
A	L	G	O	R

I	T	H	M	S

A	G	L	O	R

Mergesort





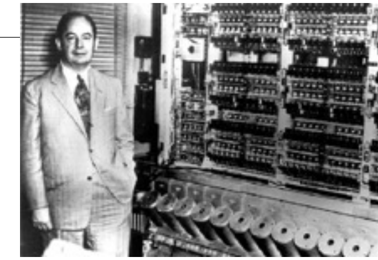
- Divide array into two halves.
- Recursively sort left half.
- Recursively sort right half.

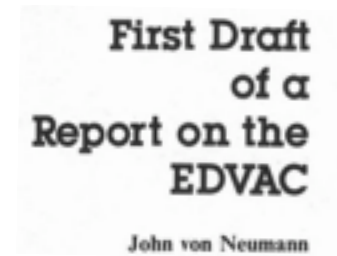
A	L	G	O	R	I	T	H	M	S

A	L	G	O	R	I	T	H	M	S

A	G	L	O	R	H	I	M	S	T

Mergesort





- Divide array into two halves.
- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

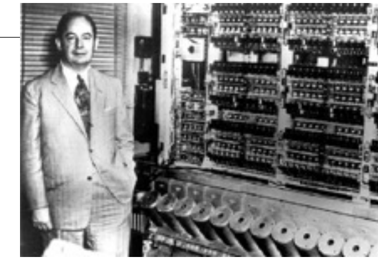
A	G	L	O	R

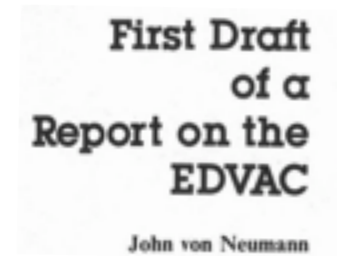
H	I	M	S	T

A	G	H	I	L	M	O	R	S	T

Mergesort

- Divide array into two halves.
- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.




First Draft of a Report on the EDVAC

John von Neumann

A	L	G	O	R	I	T	H	M	S

Running time: T(n)

A	L	G	O	R	I	T	H	M	S

divide O(1)

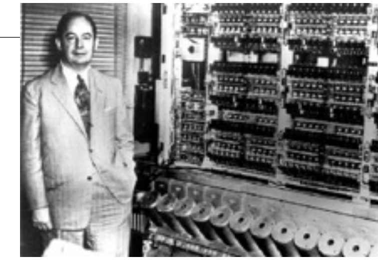
A	G	L	O	R	H	I	M	S	T

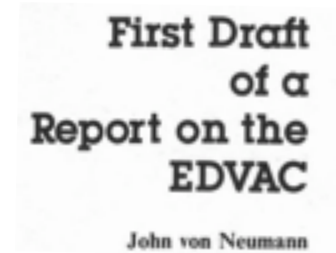
sort 2T(n/2)

A	G	H	I	L	M	O	R	S	T

merge ???

Mergesort





- Divide array into two halves.
- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.

A L G O R I T H M S	Running time: T(n)
A L G O R I T H M S	divide $O(1)$
A G L O R H I M S T	sort $2T(n/2)$
A G H I L M O R S T	merge ???

Merging

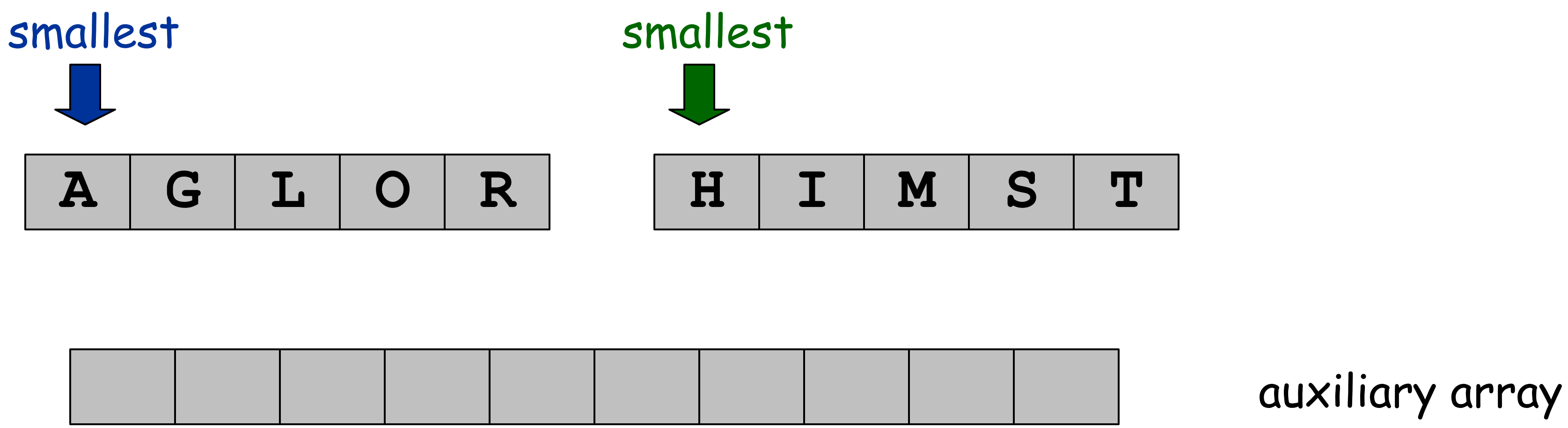
Merge

- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done

Merging

Merge

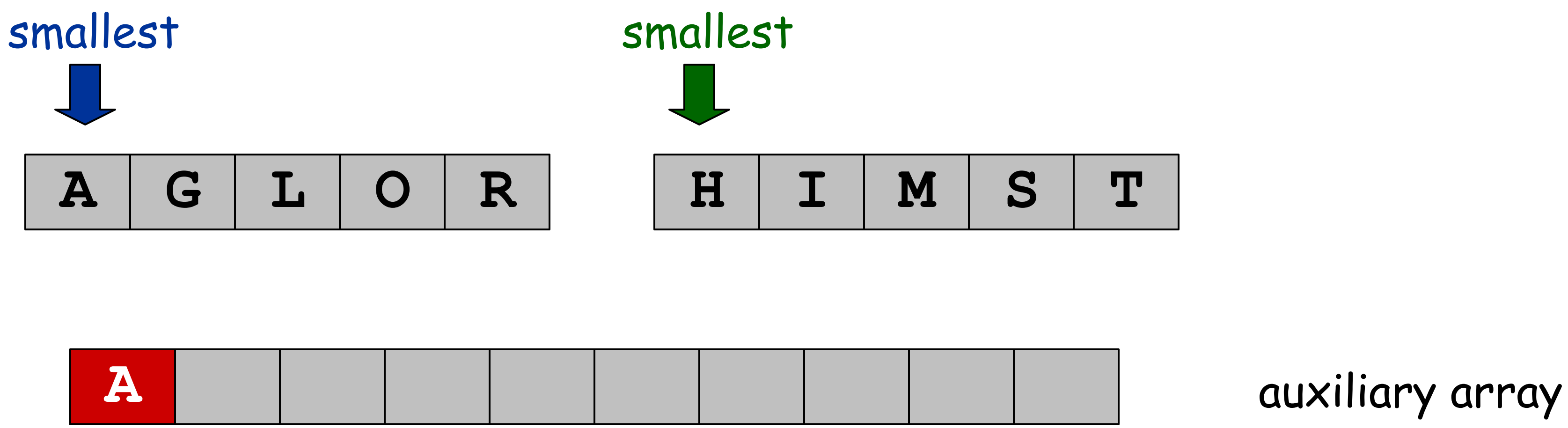
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

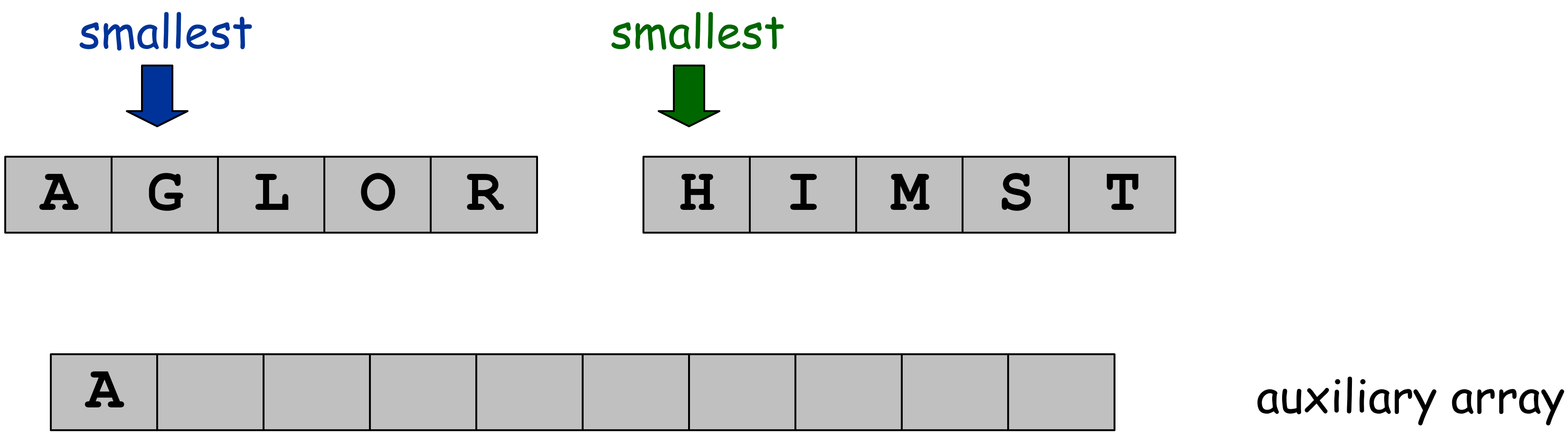
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

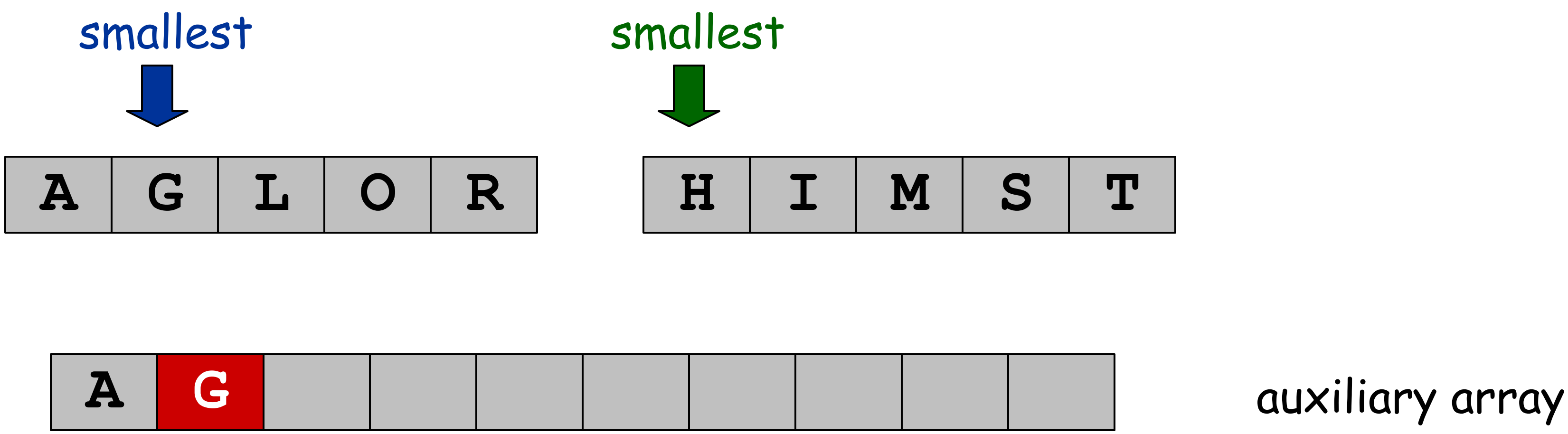
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

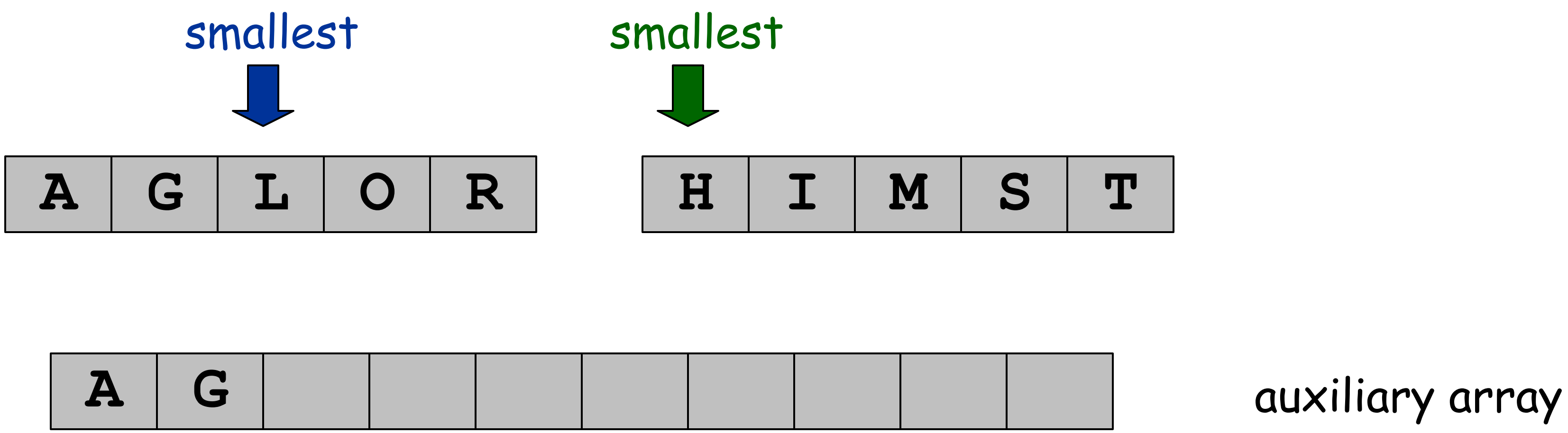
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

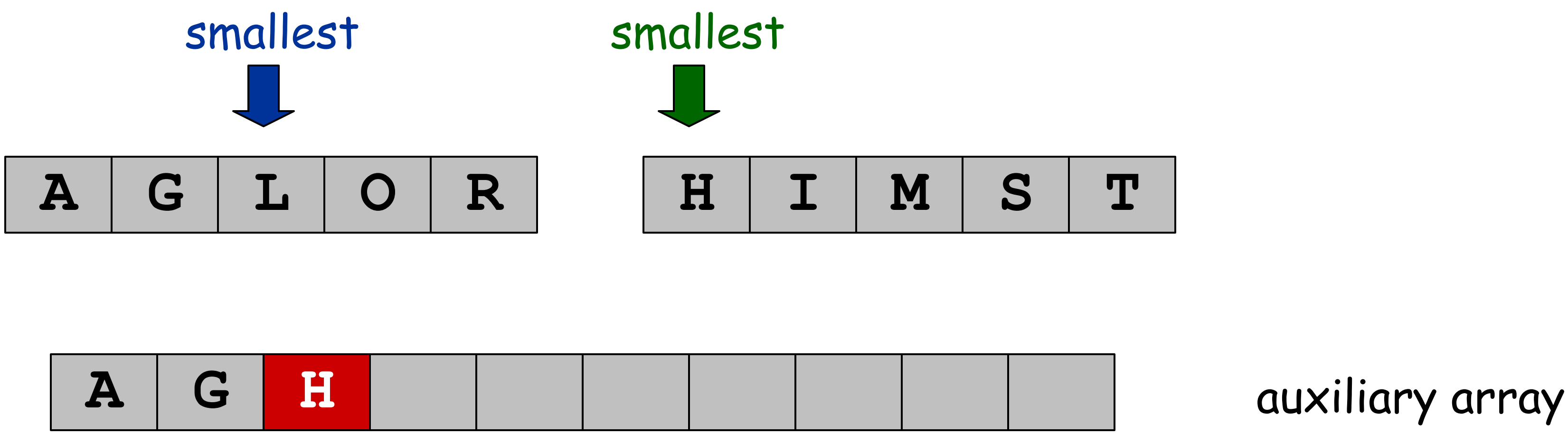
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

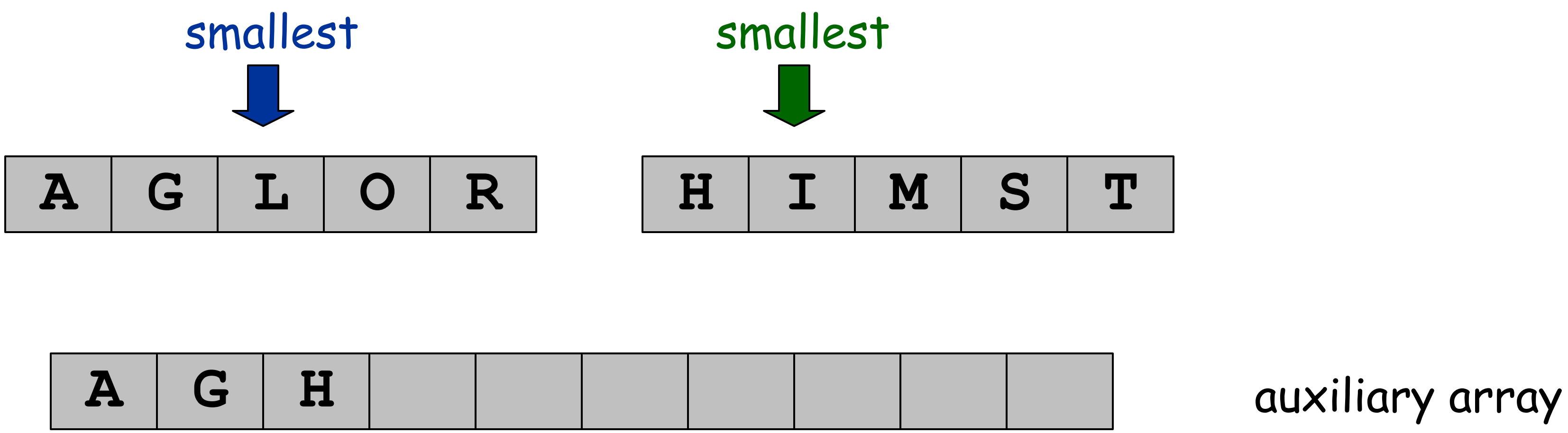
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

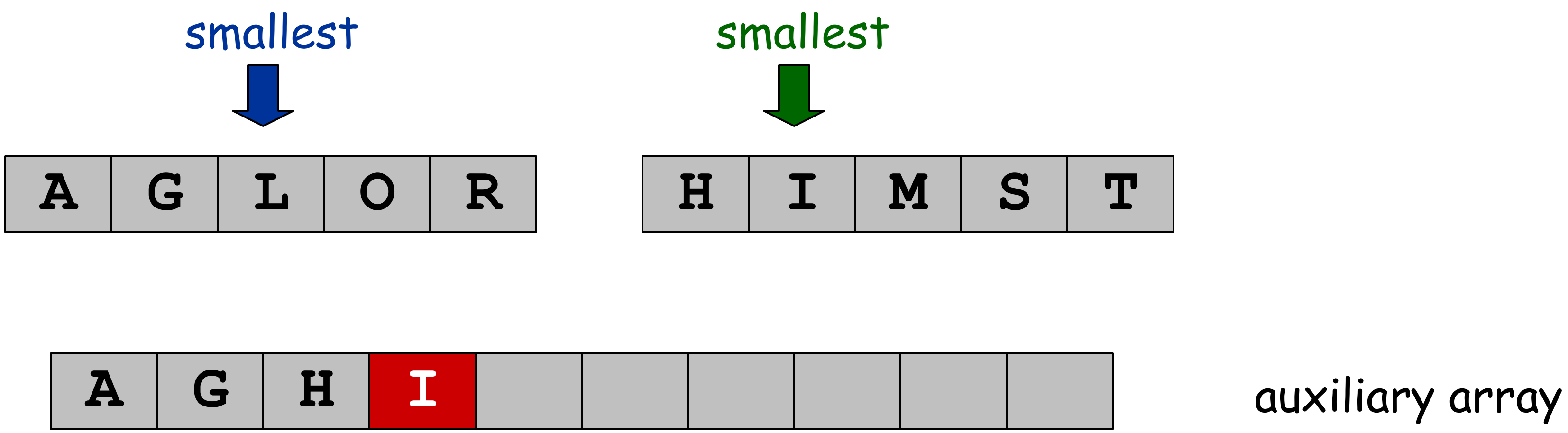
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

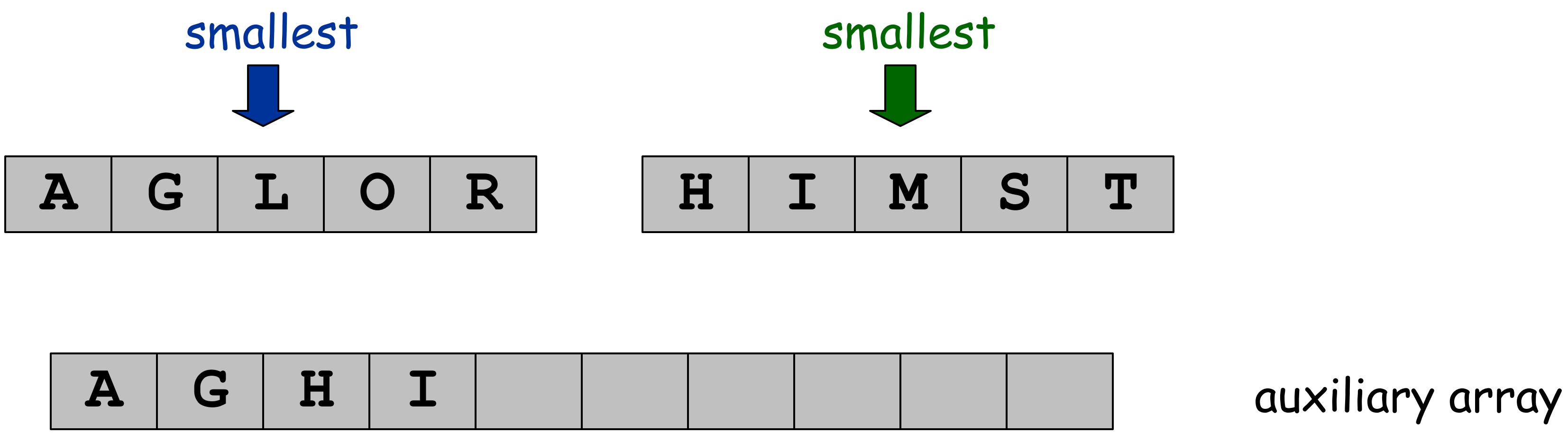
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

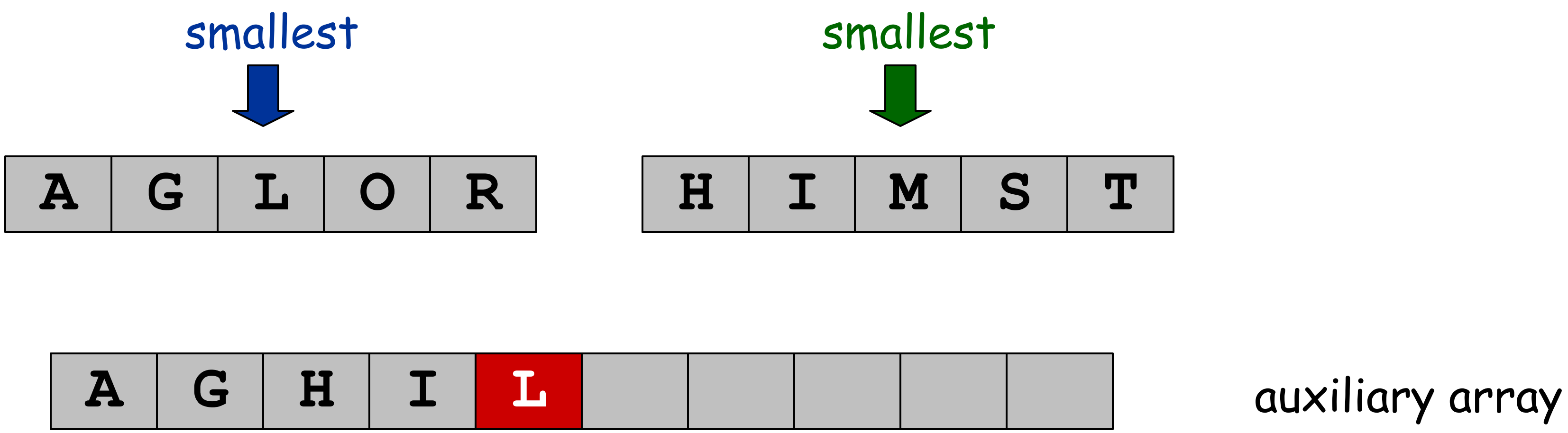
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

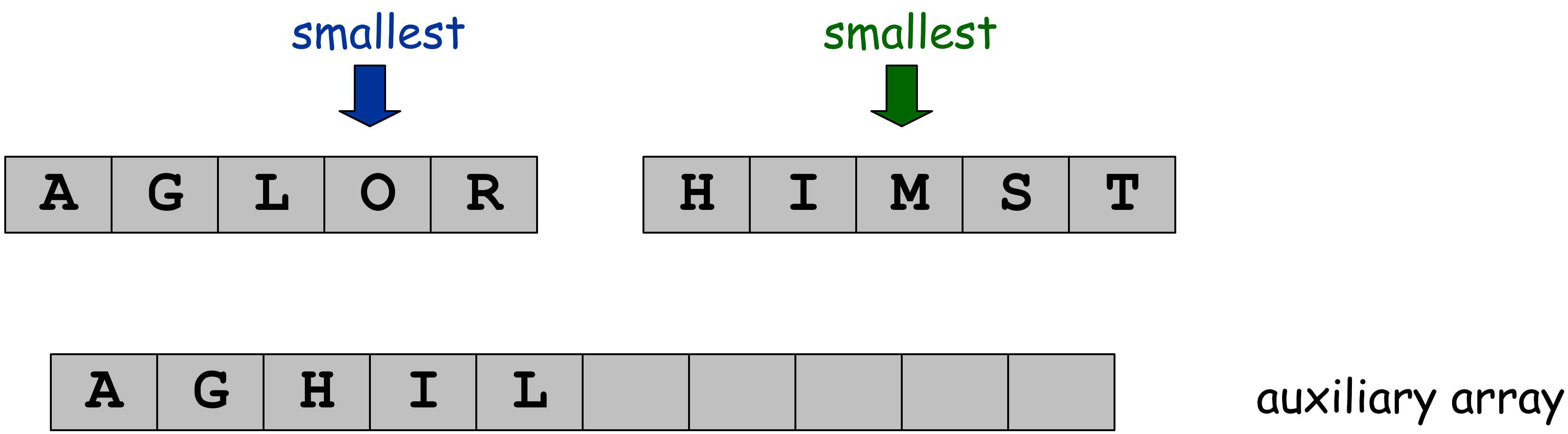
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

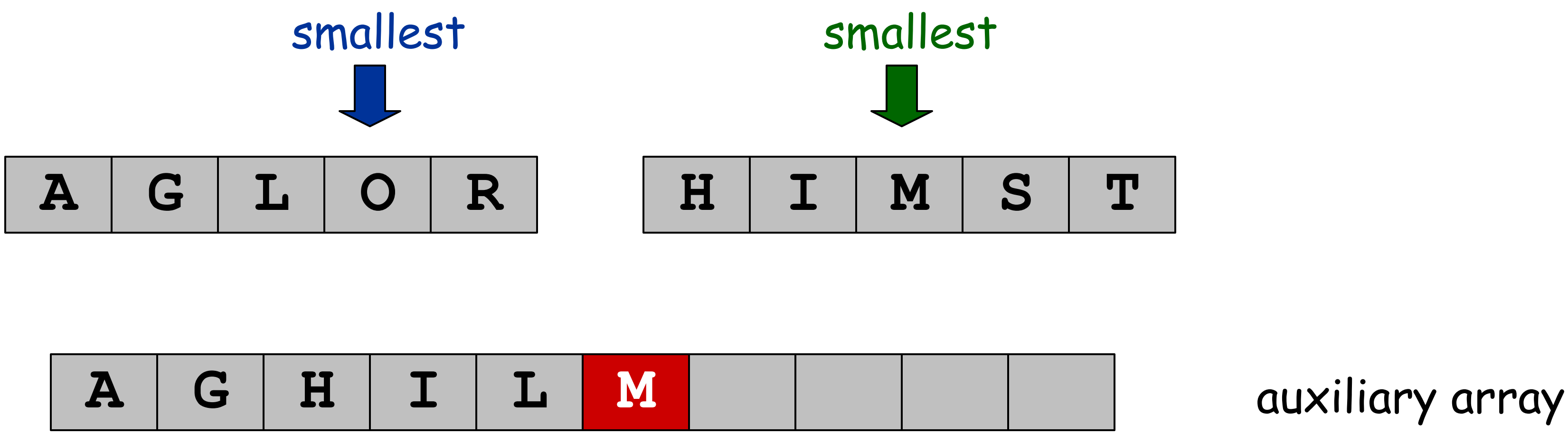
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

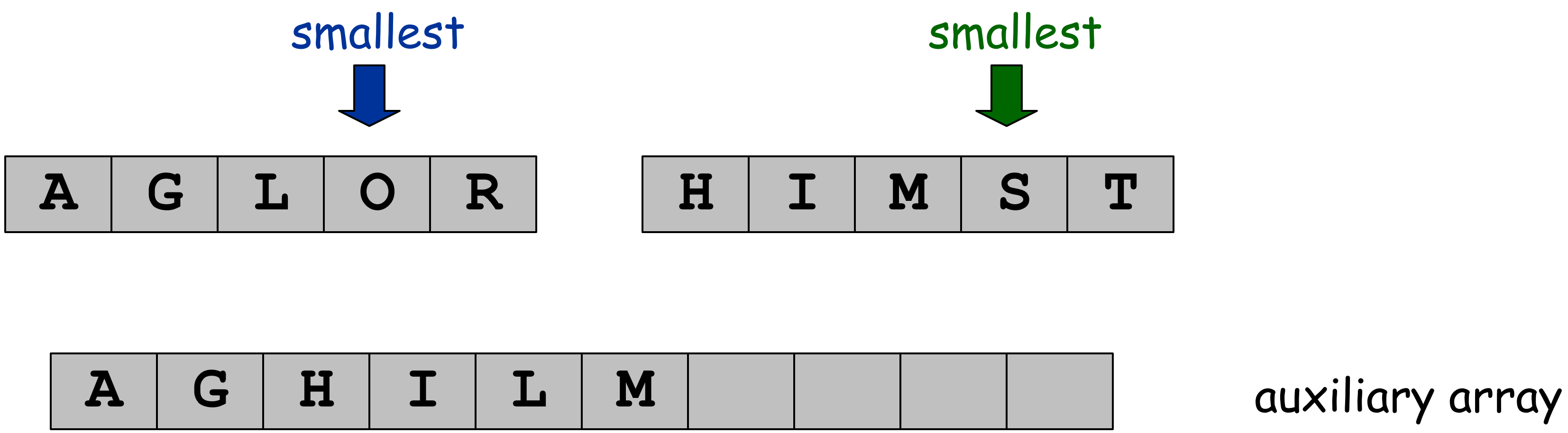
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

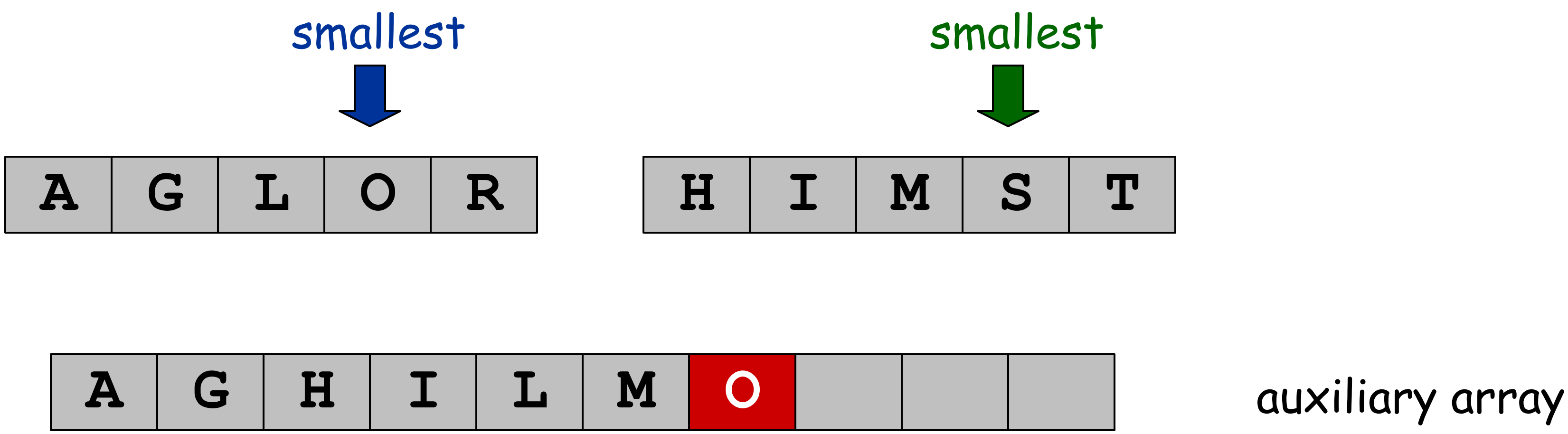
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

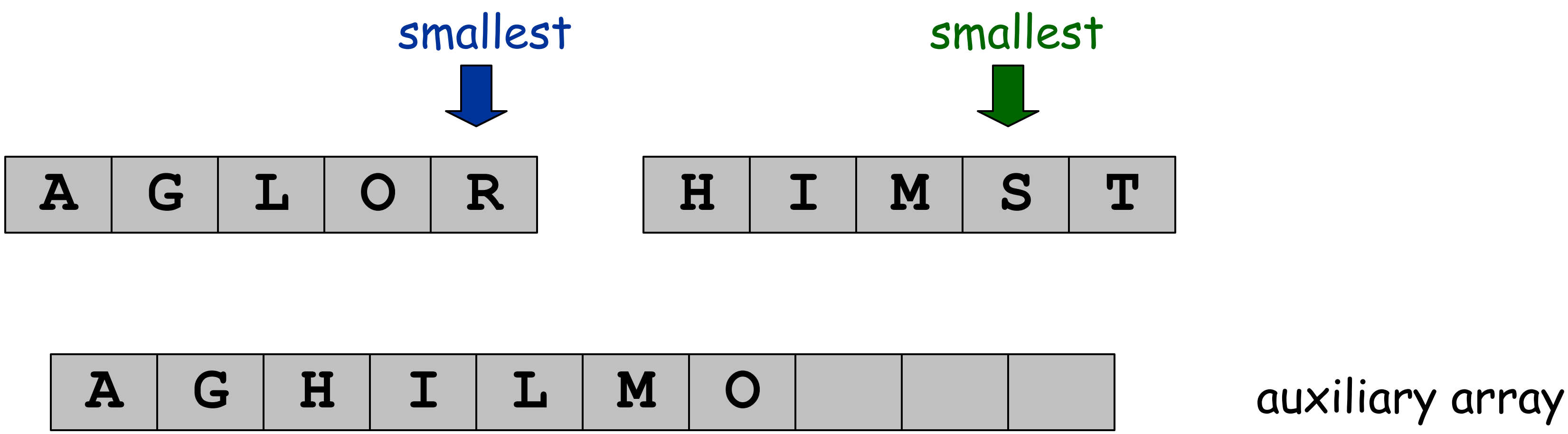
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

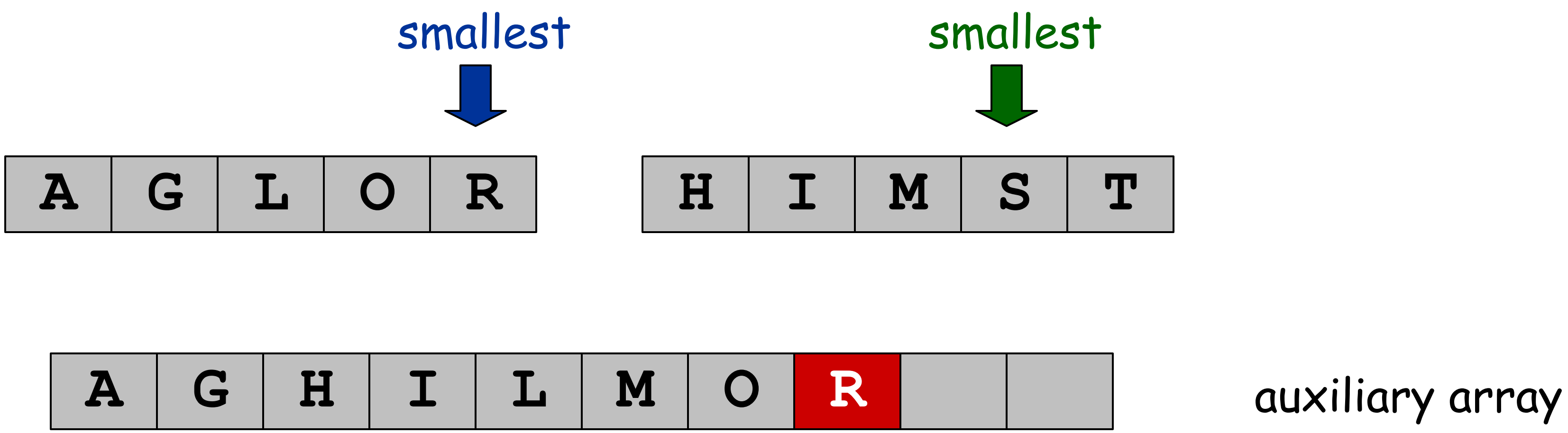
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

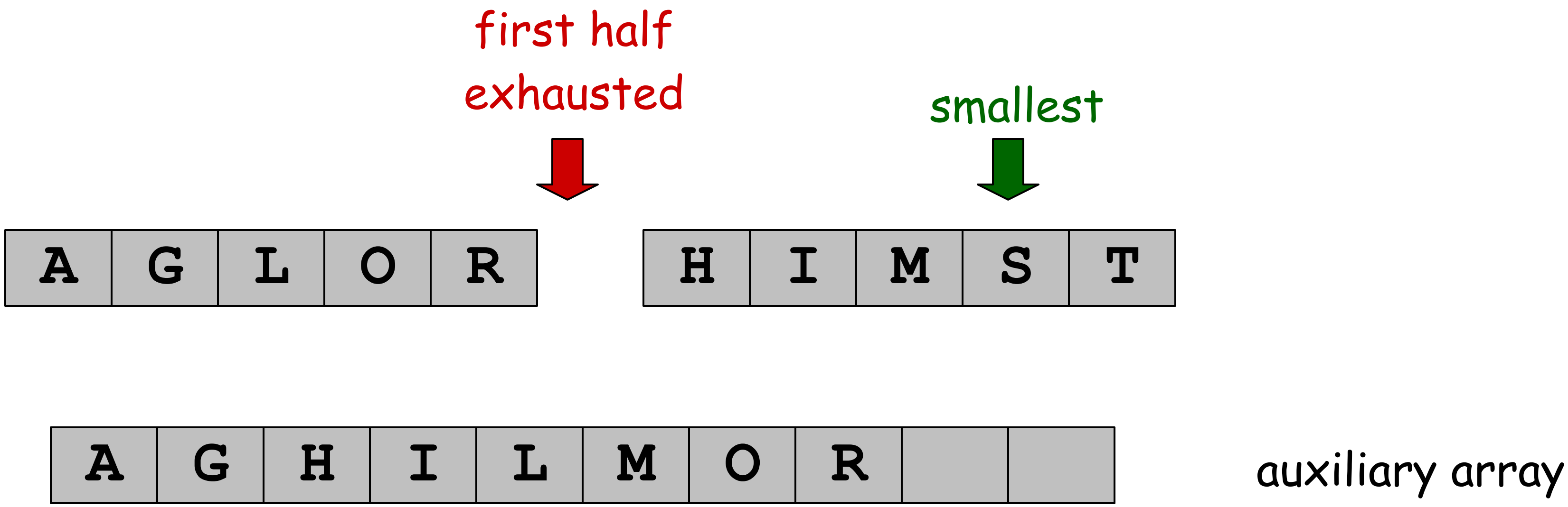
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

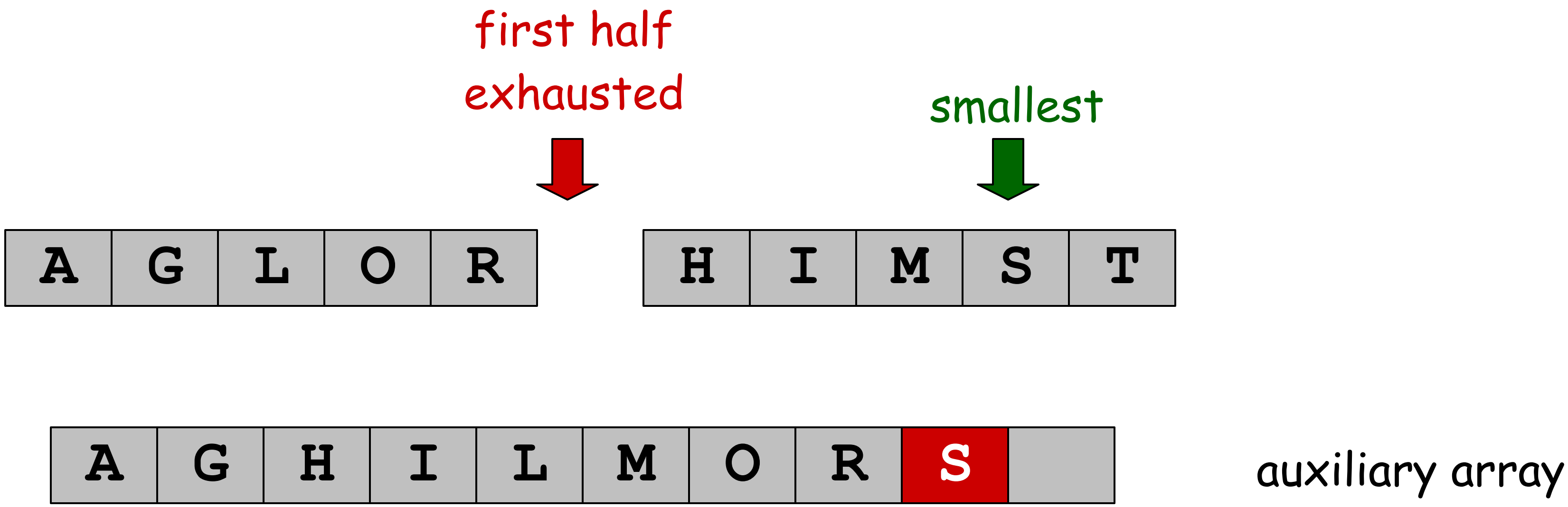
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

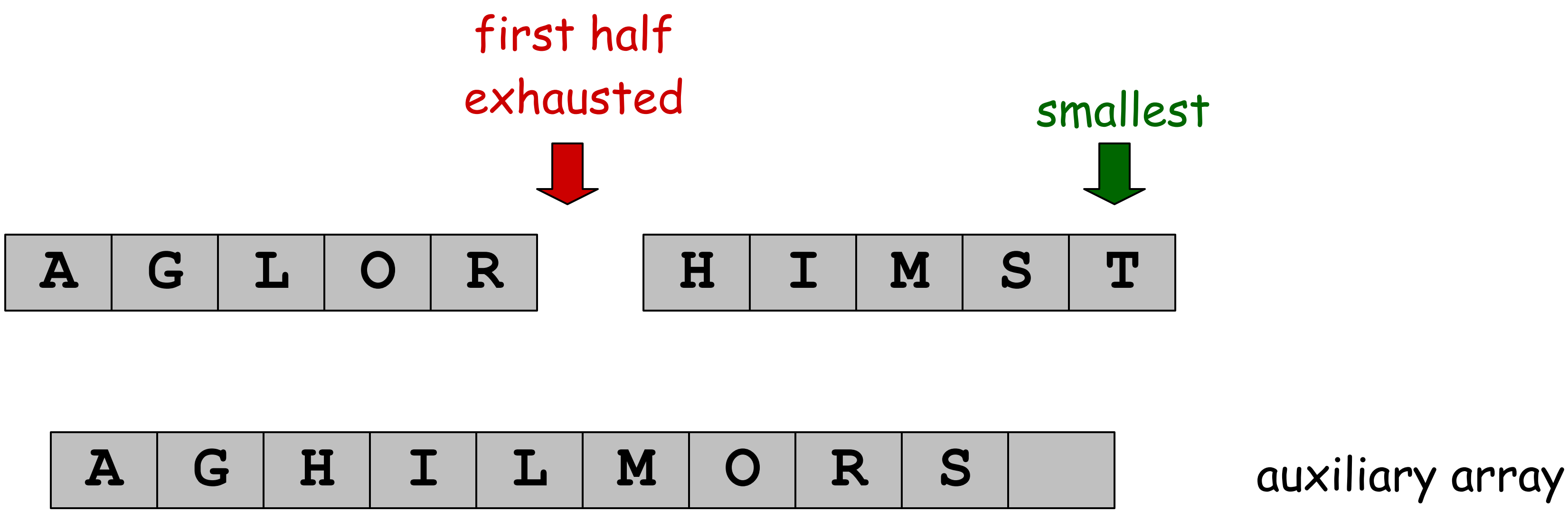
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

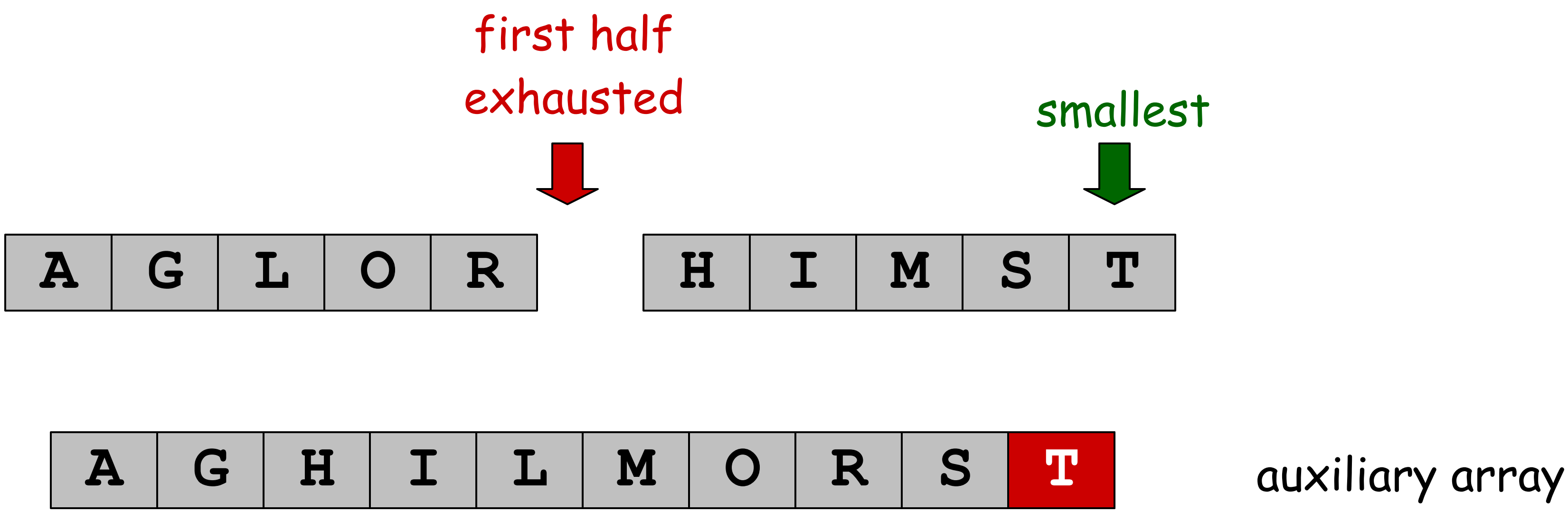
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

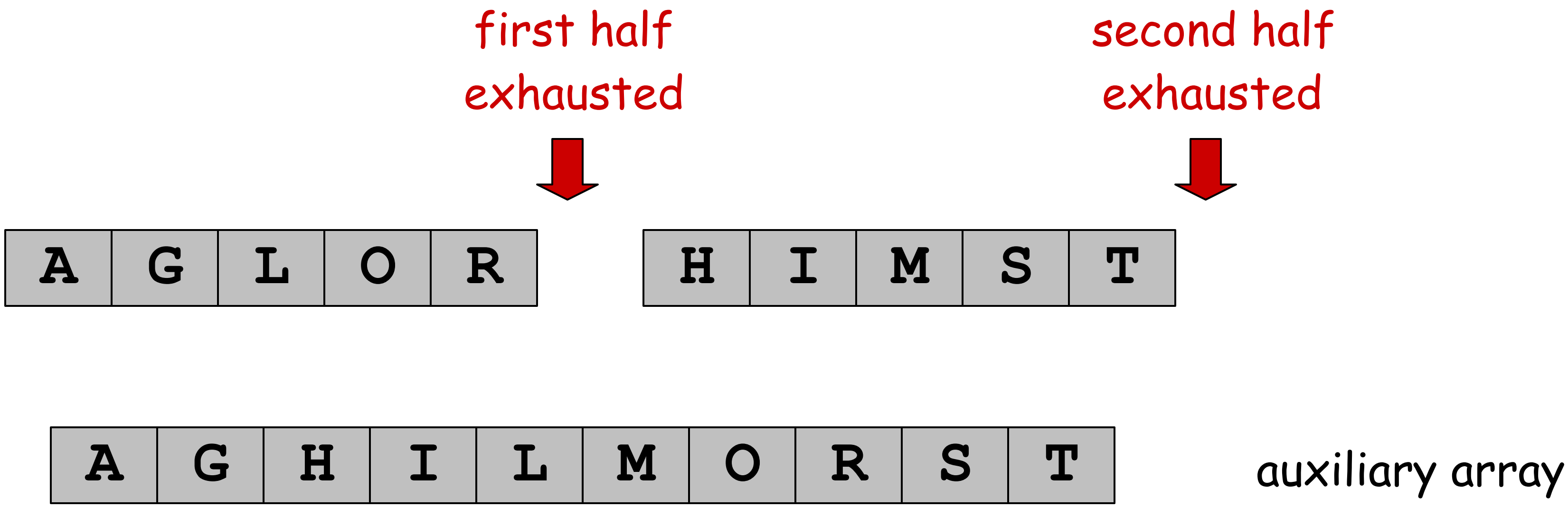
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

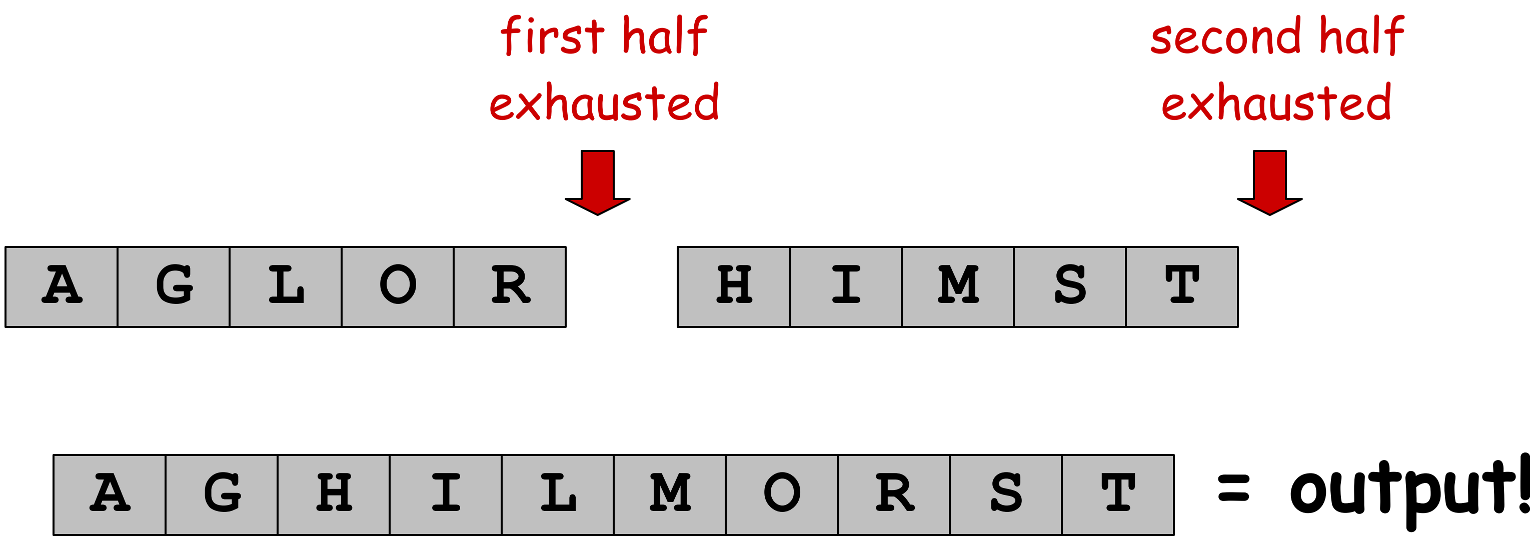
- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Merging

Merge

- Keep track of smallest element in sorted halves
- Insert smallest of two elements into new array
- Repeat until done



Is mergesort any good?

Defn: $T(n)$ = # Comparisons made by mergesort in worst-case on array with n elements.

Mergesort recurrence: $T(1) = 1$

Is mergesort any good?

Defn: T(n) = # Comparisons made by mergesort
in worst-case on array with n elements.

Mergesort recurrence: T(1) = 1

$$\boxed{T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}}$$

Is mergesort any good?

Defn: T(n) = # Comparisons made by mergesort in worst-case on array with n elements.

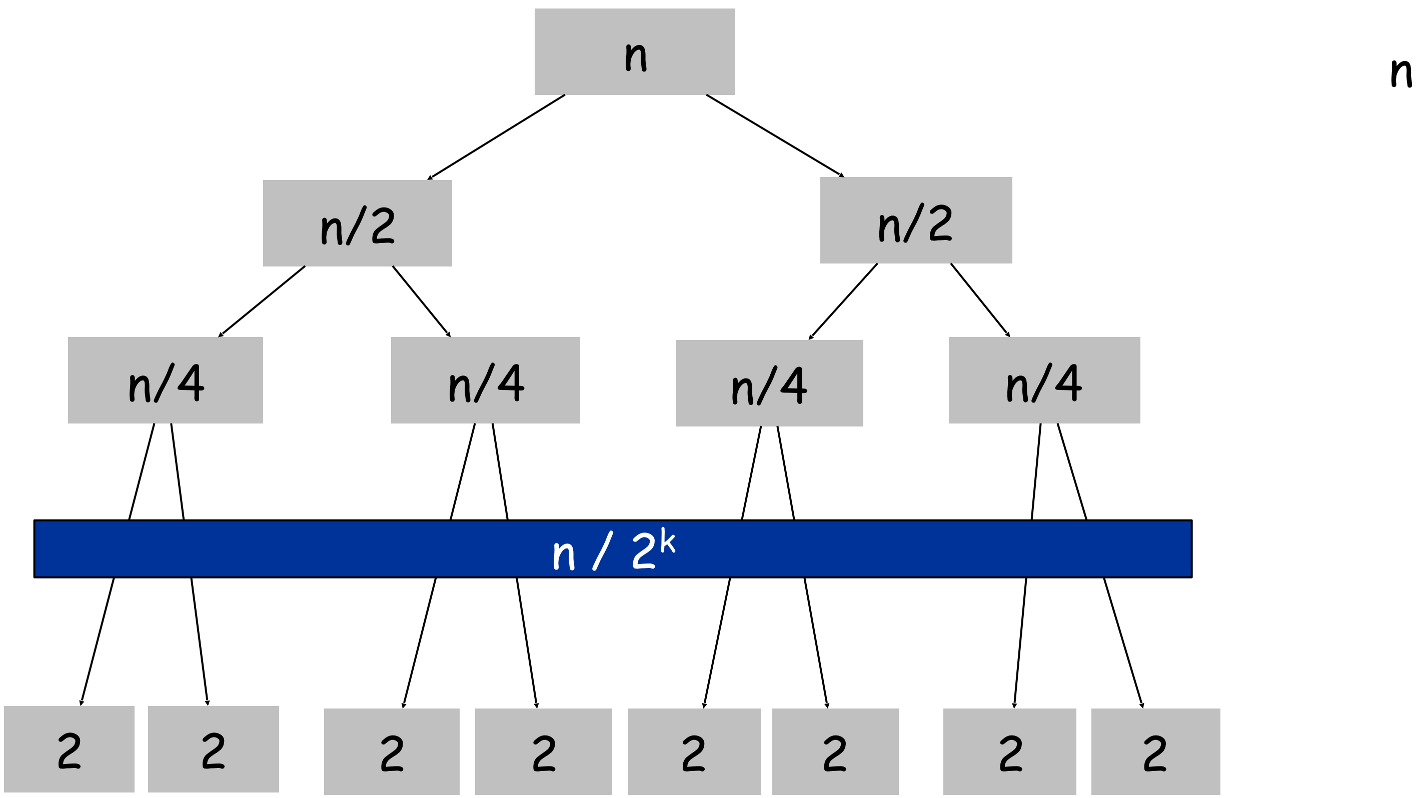
Mergesort recurrence: T(1) = 1

$$\boxed{T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}}$$

Solution: O(n log n)

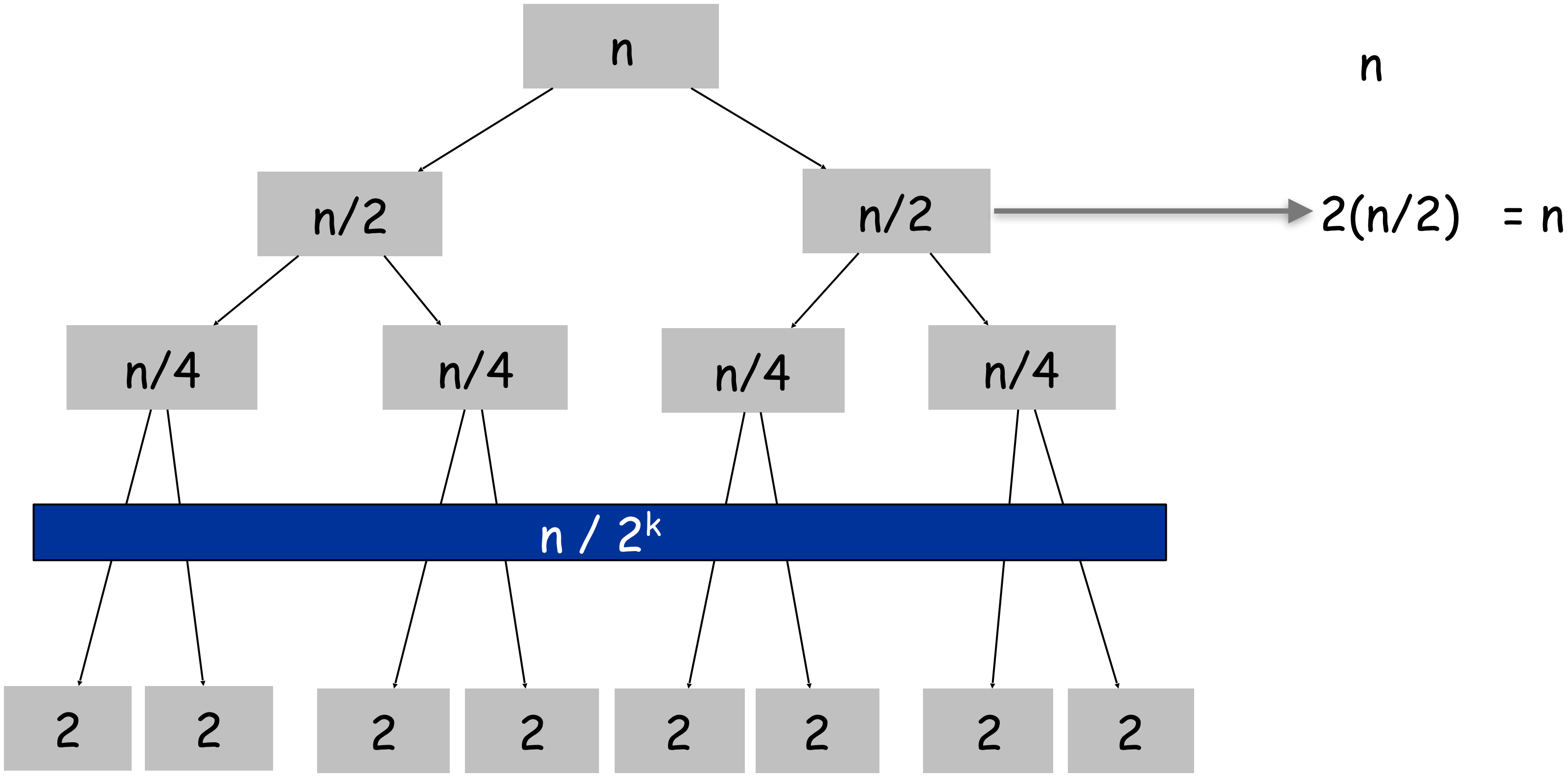
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



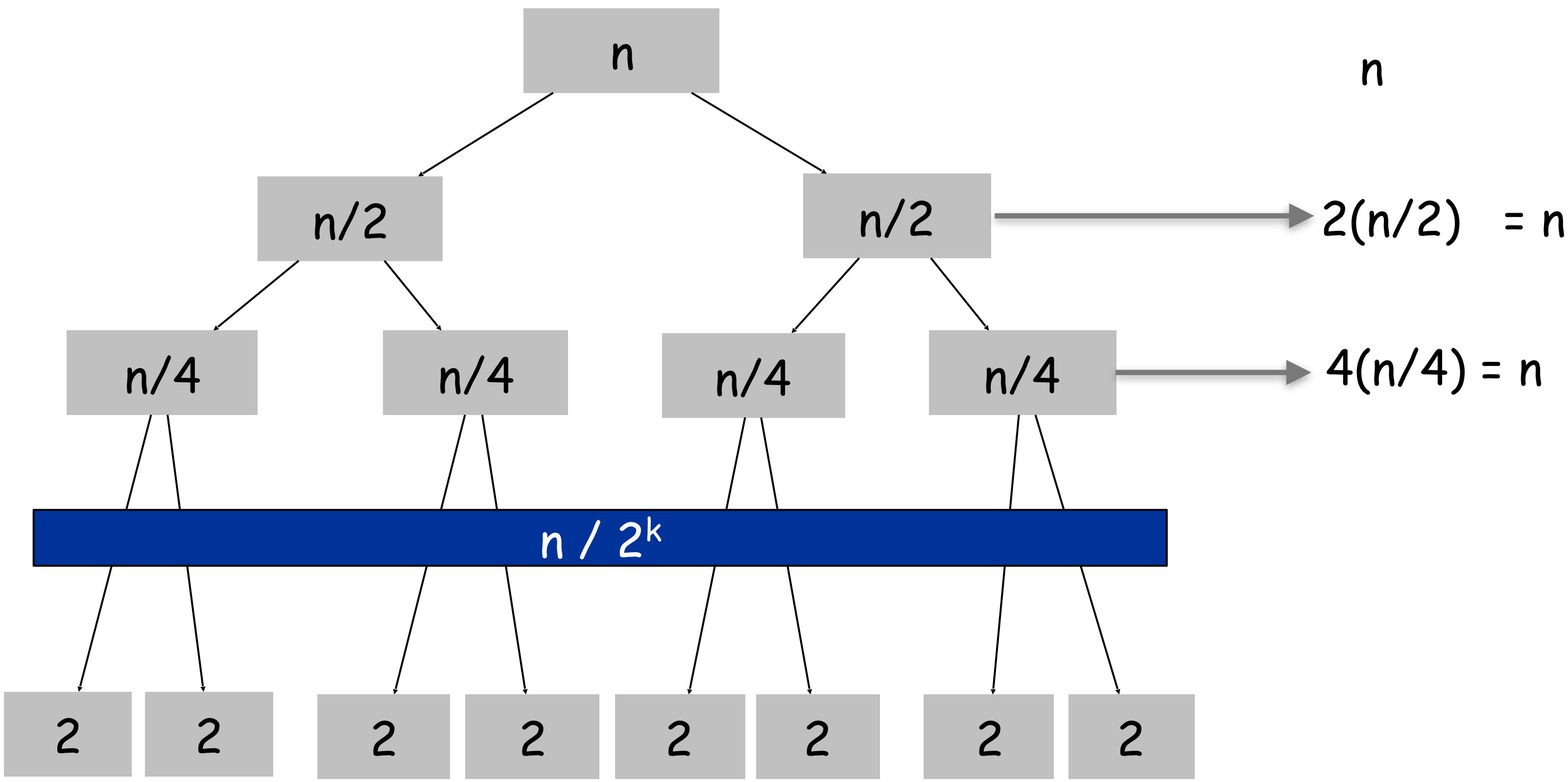
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



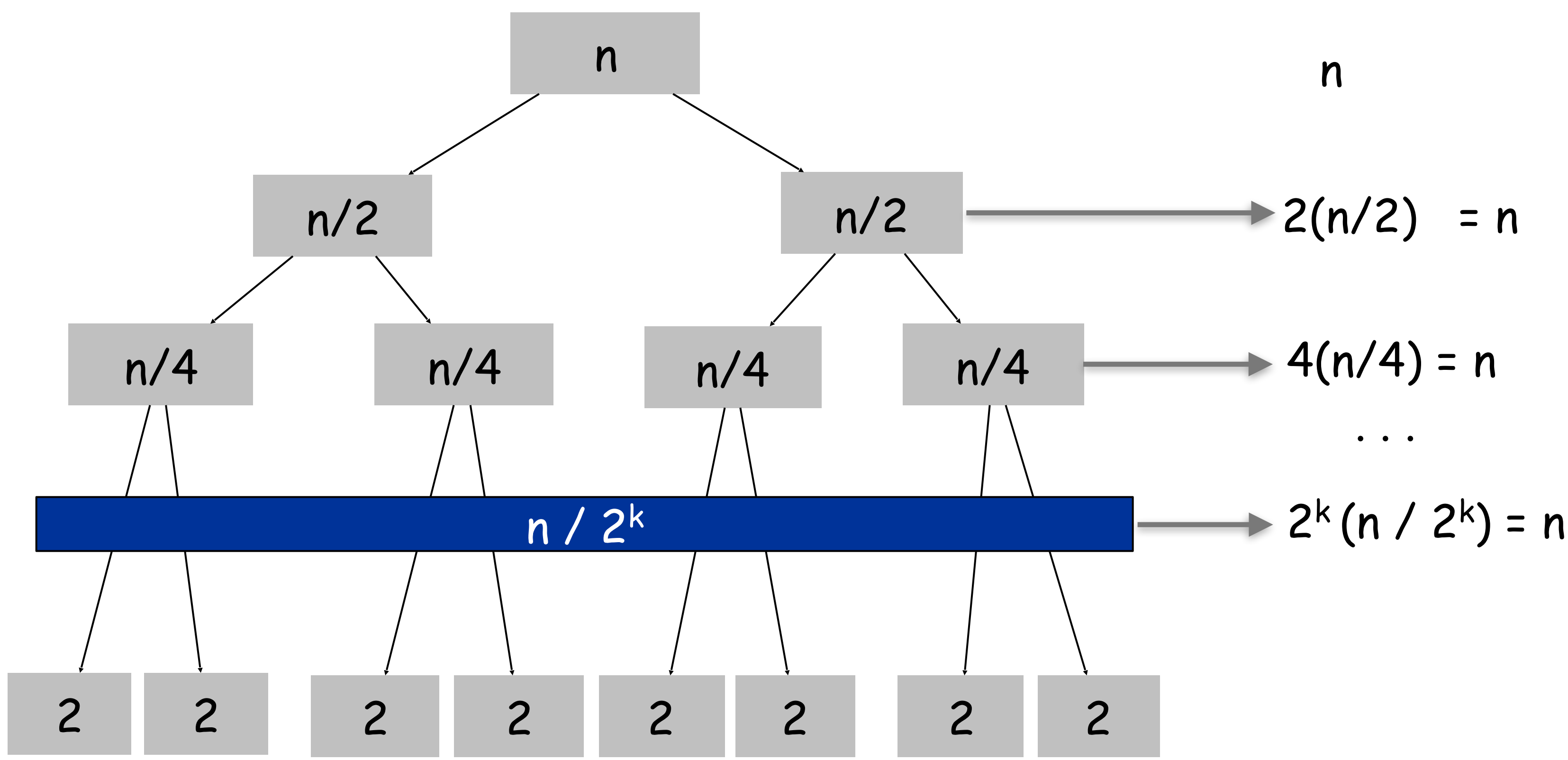
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



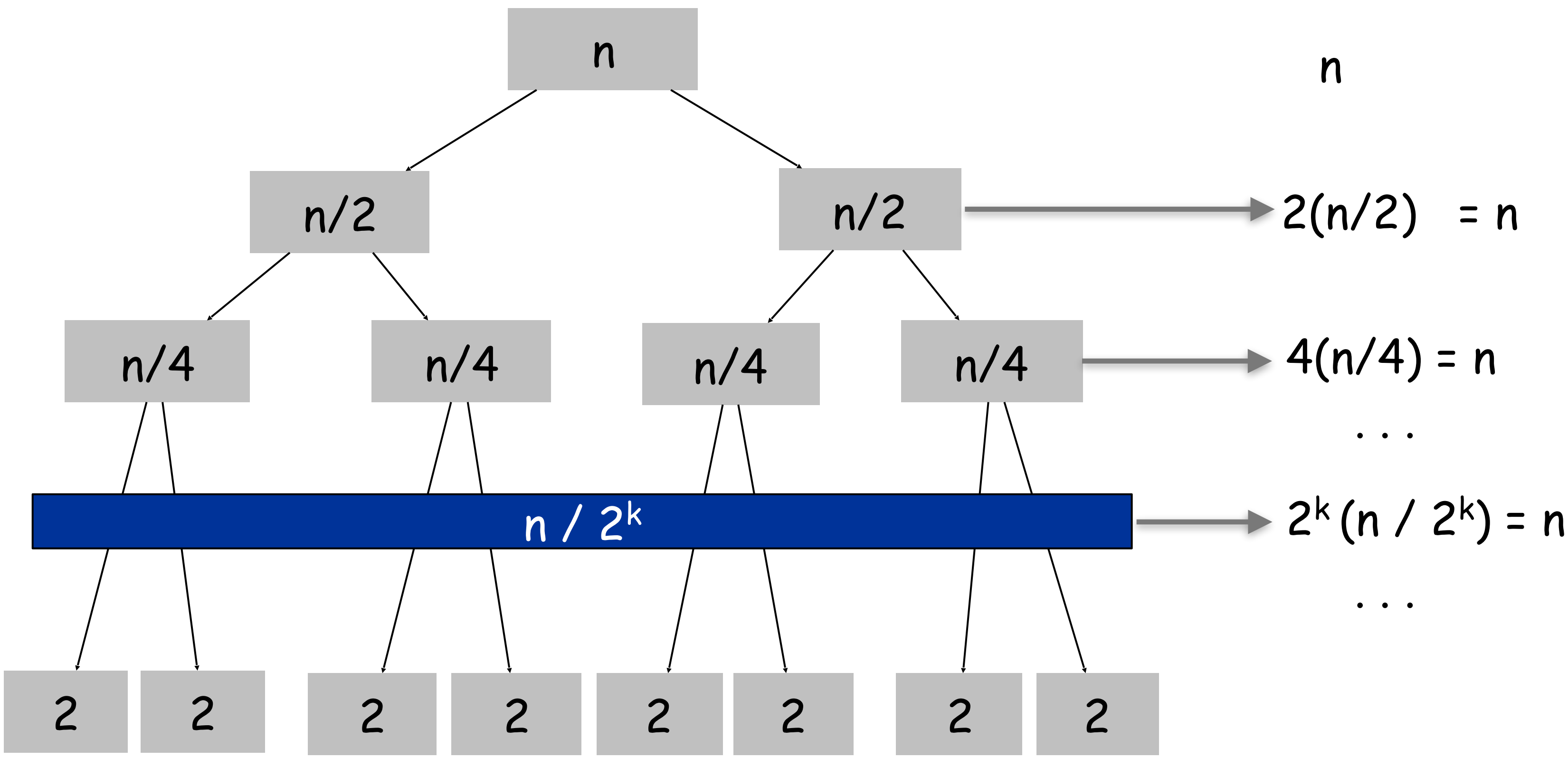
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



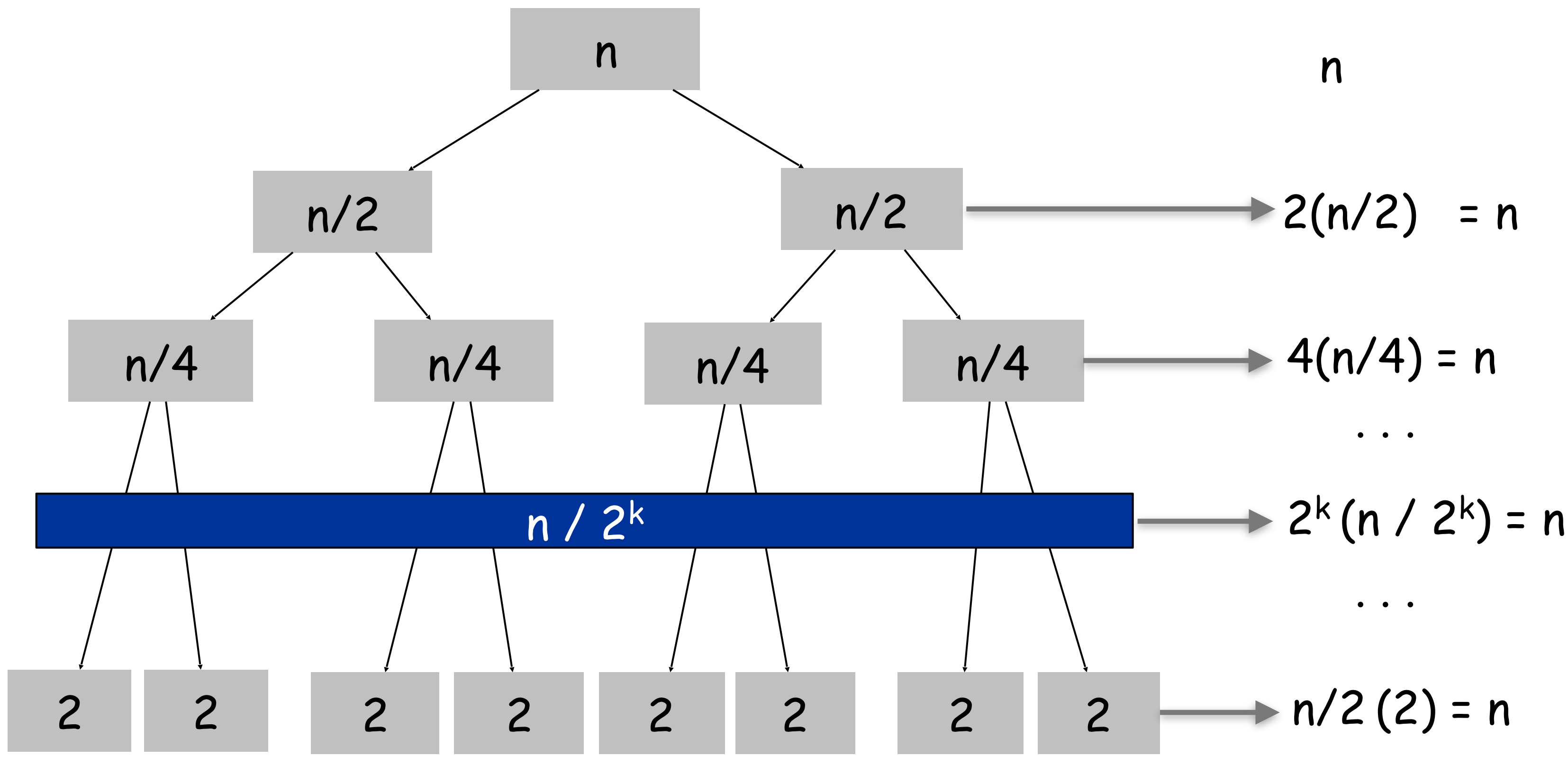
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



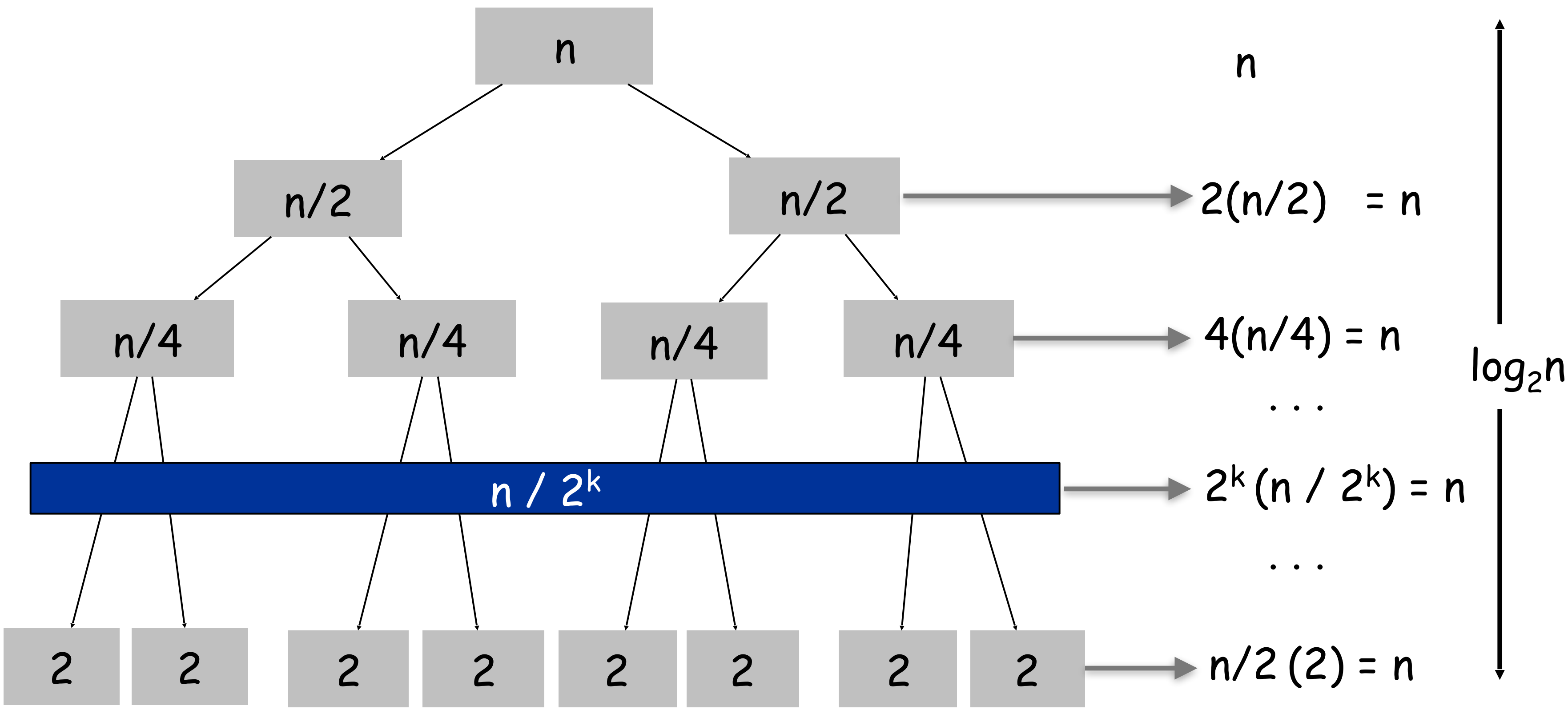
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



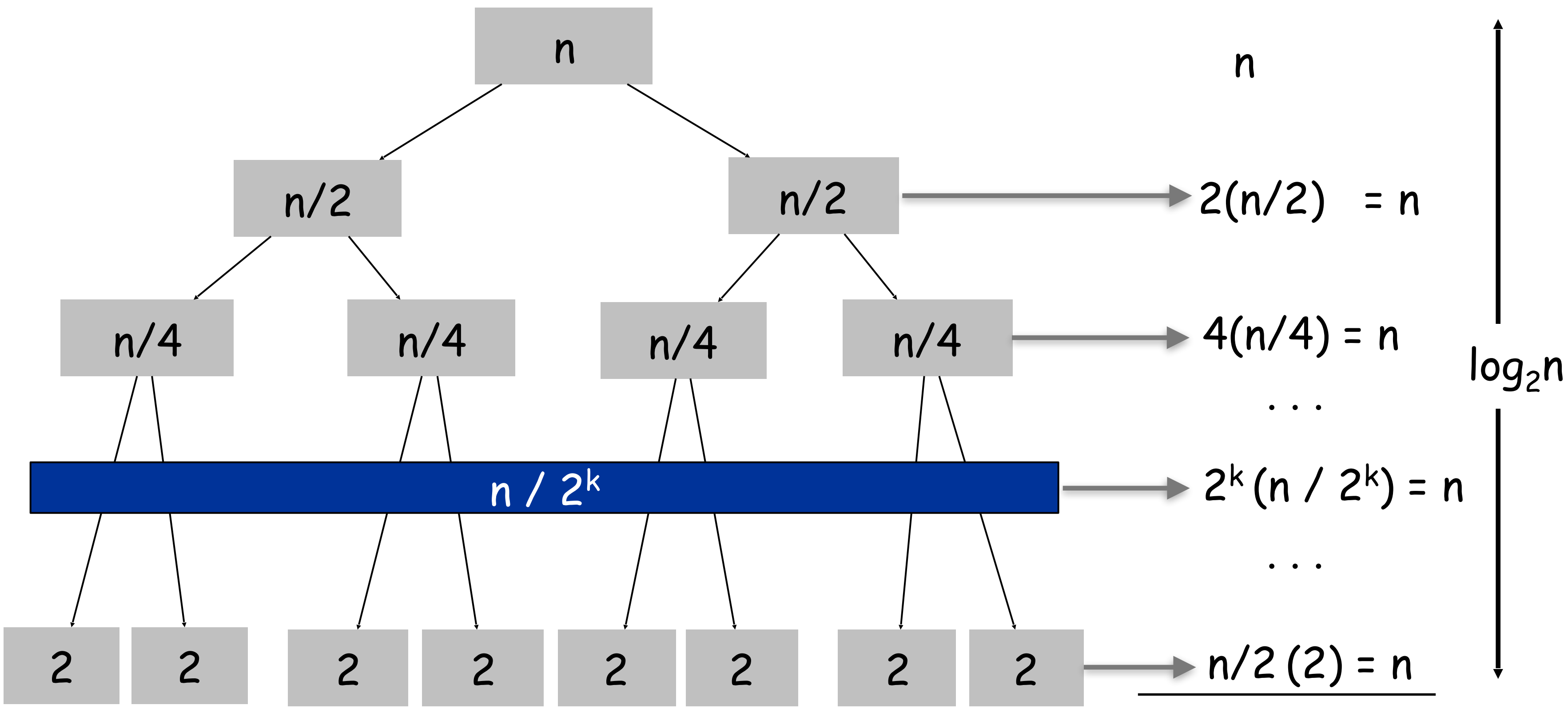
Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



Proof by recursion tree

$$T(n) = \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left-half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right-half}} + \underbrace{n}_{\text{merging}}$$



TOTAL = $n \log_2 n$

Plan for Today

Divide and Conquer

Merge-sort

Master theorem

Plan for Today

Divide and Conquer

Merge-sort

Master theorem: Generic method for solving recurrences.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) \;+\; f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.

Master method

Goal: Solve common divide-and-conquer recurrences:

$$\boxed{T(n) = a\,T\left(\frac{n}{b}\right) + f(n)}$$

Terms.

- $a \geq 1$ is the number of subproblems.
- $b > 0$ is the factor by which subproblem size decreases.
- $f(n)$ = work to divide/merge subproblems.