



Design and Analysis
of Algorithms I

Introduction

Why Study Algorithms?

Why Study Algorithms?

- important for all other branches of computer science

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
 - “Everyone knows Moore’s Law – a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years....in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed.”
 - Excerpt from *Report to the President and Congress: Designing a Digital Future*, December 2010 (page 71).

Why Study Algorithms?

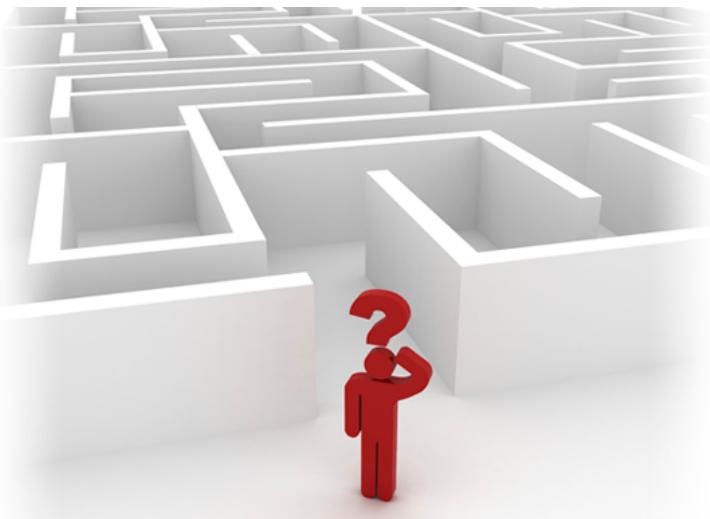
- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel “lens” on processes outside of computer science and technology
 - quantum mechanics, economic markets, evolution

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel “lens” on processes outside of computer science and technology
- challenging (i.e., good for the brain!)

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel “lens” on processes outside of computer science and technology
- challenging (i.e., good for the brain!)
- fun



Design and Analysis
of Algorithms I

Introduction

Integer Multiplication

Integer Multiplication

Input : 2 n-digit numbers x and y

Output : product $x*y$

“Primitive Operation” - add or multiply 2 single-digit numbers

The Grade-School Algorithm

A handwritten multiplication problem is shown:

$$\begin{array}{r} 5678 \\ \times 1234 \\ \hline 22712 \\ 17034 \\ 11356 \\ \hline 7006652 \end{array}$$

The result is circled in red. A green bracket on the right side of the circled area indicates that there are roughly n operations per row up to a constant. A green arrow points from the text to the bracket.

of operations overall \sim constant* n^2

The Algorithm Designer's Mantra

“Perhaps the most important principle for the good algorithm designer is to refuse to be content.”

-Aho, Hopcroft, and Ullman, *The Design and Analysis of Computer Algorithms*, 1974

CAN WE DO BETTER ?
[than the “obvious” method]



Design and Analysis
of Algorithms I

Introduction

Karatsuba Multiplication

Example

$$x = \overbrace{5}^a \overbrace{6}^b \overbrace{7}^c \overbrace{8}^d$$
$$y = \overbrace{1}^e \overbrace{2}^f \overbrace{3}^g \overbrace{4}^h$$

Step 1: compute $a \cdot c = 672$

Step 2: compute $b \cdot d = 2652$

Step 3: compute $(a+b)(c+d) = 134 \cdot 46 = 6164$

Step 4: compute $\textcircled{3} - \textcircled{2} - \textcircled{1} = 2840$

Step 5:

$$\begin{array}{r} 6720000 \\ 2652 \\ \hline 2840000 \\ \hline 7006652 \end{array} = ((1234)(5678))$$

A Recursive Algorithm

Write $x = 10^{n/2}a + b$ and $y = 10^{n/2}c + d$

Where a, b, c, d are $n/2$ -digit numbers.

[example: $a=56, b=78, c=12, d=34$]

$$\begin{aligned}\text{Then } x \cdot y &= (10^{n/2}a + b)(10^{n/2}c + d) \\ &= (10^n ac + 10^{n/2}(ad + bc) + bd\end{aligned}\quad (*)$$

Idea : recursively compute ac, ad, bc, bd , then
compute $(*)$ in the obvious way

Simple Base Case
Omitted

Karatsuba Multiplication

$$x \cdot y = (10^n ac + 10^{n/2}(ad + bc) + bd$$

1. Recursively compute ac
2. Recursively compute bd
3. Recursively compute $(a+b)(c+d) = ac+bd+ad+bc$

Gauss' Trick : $(3) - (1) - (2) = ad + bc$

Upshot : Only need 3 recursive multiplications (and some additions)

Q : which is the fastest algorithm ?



Design and Analysis
of Algorithms I

Introduction

About The Course

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
- Primitives for reasoning about graphs
- Use and implementation of data structures

Course Topics

- Vocabulary for design and analysis of algorithms
 - E.g., “Big-Oh” notation
 - “sweet spot” for high-level reasoning about algorithms

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
 - Will apply to: Integer multiplication, sorting, matrix multiplication, closest pair
 - General analysis methods (“Master Method/Theorem”)

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
 - Will apply to: QuickSort, primality testing, graph partitioning, hashing.

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
- Primitives for reasoning about graphs
 - Connectivity information, shortest paths, structure of information and social networks.

Course Topics

- Vocabulary for design and analysis of algorithms
- Divide and conquer algorithm design paradigm
- Randomization in algorithm design
- Primitives for reasoning about graphs
- Use and implementation of data structures
 - Heaps, balanced binary search trees, hashing and some variants (e.g., bloom filters)

Topics in Sequel Course

- Greedy algorithm design paradigm

Topics in Sequel Course

- Greedy algorithm design paradigm
- Dynamic programming algorithm design paradigm

Topics in Sequel Course

- Greedy algorithm design paradigm
- Dynamic programming algorithm design paradigm
- NP-complete problems and what to do about them

Topics in Sequel Course

- Greedy algorithm design paradigm
- Dynamic programming algorithm design paradigm
- NP-complete problems and what to do about them
- Fast heuristics with provable guarantees
- Fast exact algorithms for special cases
- Exact algorithms that beat brute-force search

Skills You'll Learn

- Become a better programmer

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills
- Start “thinking algorithmically”

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills
- Start “thinking algorithmically”
- Literacy with computer science’s “greatest hits”

Skills You'll Learn

- Become a better programmer
- Sharpen your mathematical and analytical skills
- Start “thinking algorithmically”
- Literacy with computer science’s “greatest hits”
- Ace your technical interviews

Who Are You?

- It doesn't really matter. (It's a free course, after all.)

Tim Roughgarden

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.
- Doesn't matter which language(s) you know.
 - But you should be capable of translating high-level algorithm descriptions into working programs in *some* programming language.

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.
- Doesn't matter which language(s) you know.
- Some (perhaps rusty) mathematical experience.
 - Basic discrete math, proofs by induction, etc.

Who Are You?

- It doesn't really matter. (It's a free course, after all.)
- Ideally, you know some programming.
- Doesn't matter which language(s) you know.
- Some (perhaps rusty) mathematical experience.
 - Basic discrete math, proofs by induction, etc.
- *Excellent free reference:* “Mathematics for Computer Science”, by Eric Lehman and Tom Leighton. (Easy to find on the Web.)

Supporting Materials

- All (annotated) slides available from course site.

Supporting Materials

- All (annotated) slides available from course site.
- No required textbook. A few of the many good ones:
 - Kleinberg/Tardos, *Algorithm Design*, 2005.
 - Dasgupta/Papadimitriou/Vazirani, *Algorithms*, 2006.
 - Cormen/Leiserson/Rivest/Stein, *Introduction to Algorithms*, 2009 (3rd edition).
 - Mehlhorn/Sanders, *Data Structures and Algorithms: The Basic Toolbox*, 2008.

 
Freely available online

Supporting Materials

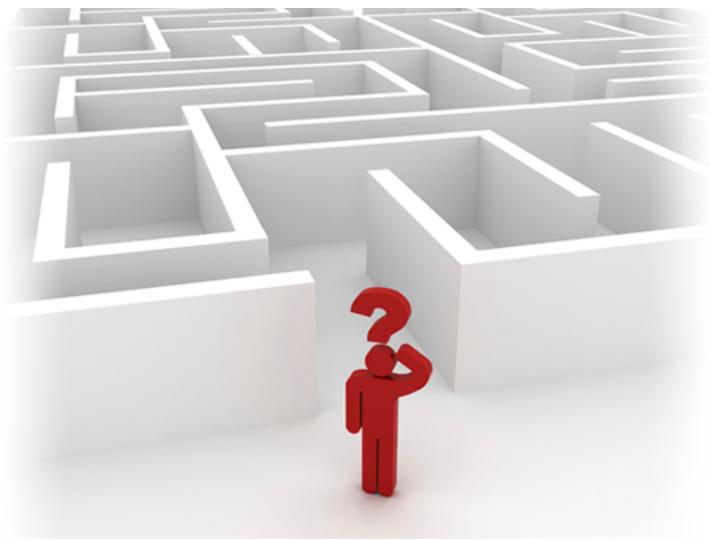
- All (annotated) slides available from course site.
- No required textbook. A few of the many good ones:
 - Kleinberg/Tardos, *Algorithm Design*, 2005.
 - Dasgupta/Papadimitriou/Vazirani, *Algorithms*, 2006.
 - Cormen/Leiserson/Rivest/Stein, *Introduction to Algorithms*, 2009 (3rd edition).
 - Mehlhorn/Sanders, *Data Structures and Algorithms: The Basic Toolbox*, 2008.
- No specific development environment required.
 - But you should be able to write and execute programs.

Assessment

- No grades per se. (Details on a certificate of accomplishment TBA.)
- Weekly homeworks.
 - Test understand of material
 - Synchronize students, greatly helps discussion forum
 - Intellectual challenge

Assessment

- No grades per se. (Details on a certificate of accomplishment TBA.)
- Weekly homeworks.
- Assessment tools currently just a “1.0” technology.
 - We’ll do our best!
- Will sometimes propose harder “challenge problems”
 - Will not be graded; discuss solutions via course forum



Design and Analysis
of Algorithms I

Introduction

Merge Sort (Overview)

Why Study Merge Sort?

- Good introduction to divide & conquer
 - Improves over Selection, Insertion, Bubble sorts
- Calibrate your preparation
- Motivates guiding principles for algorithm analysis (worst-case and asymptotic analysis)
- Analysis generalizes to “Master Method”

The Sorting Problem

Input : array of n numbers, unsorted.

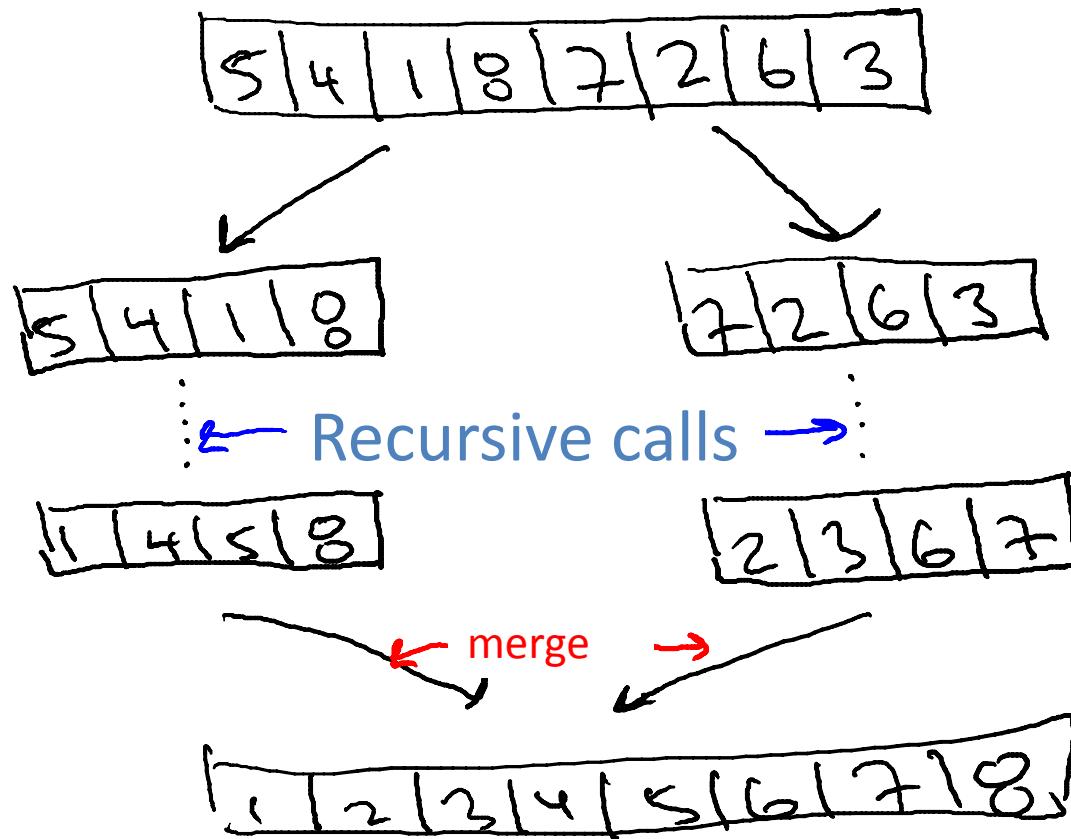
15|4|11|8|7|2|6|3|

Assume
Distinct

Output : Same numbers, sorted in increasing order

1|2|3|4|5|6|7|8|

Merge Sort: Example



Tim Roughgarden



Design and Analysis
of Algorithms I

Introduction

Merge Sort

(Pseudocode)

Tim

Merge Sort: Pseudocode

```
-- recursively sort 1st half of the input array  
-- recursively sort 2nd half of the input array  
-- merge two sorted sublists into one  
[ignores base cases]
```

Pseudocode for Merge:

C = output [length = n]

A = 1st sorted array [n/2]

B = 2nd sorted array [n/2]

i = 1

j = 1

for k = 1 to n

if A(i) < B(j)

C(k) = A(i)

i++

else [B(j) < A(i)]

C(k) = B(j)

j++

end

(ignores end cases)

Merge Sort Running Time?

Key Question : running time of Merge Sort on array of n numbers ?

[running time \sim # of lines of code executed]

Pseudocode for Merge:

$C = \text{output} [\text{length} = n]$

$A = 1^{\text{st}}$ sorted array $[n/2]$

$B = 2^{\text{nd}}$ sorted array $[n/2]$

$i = 1$

$j = 1$

} 2 operations

```
for k = 1 to n ✓  
    if A(i) < B(j) ✓  
        C(k) = A(i) -  
        i++ -  
    else [B(j) < A(i)]  
        C(k) = B(j) -  
        j++ -  
end  
(ignores end cases)
```

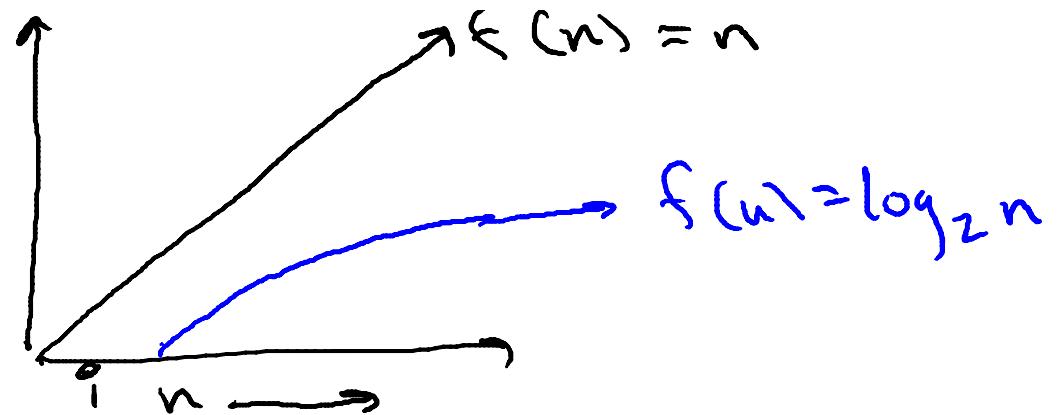
Running Time of Merge

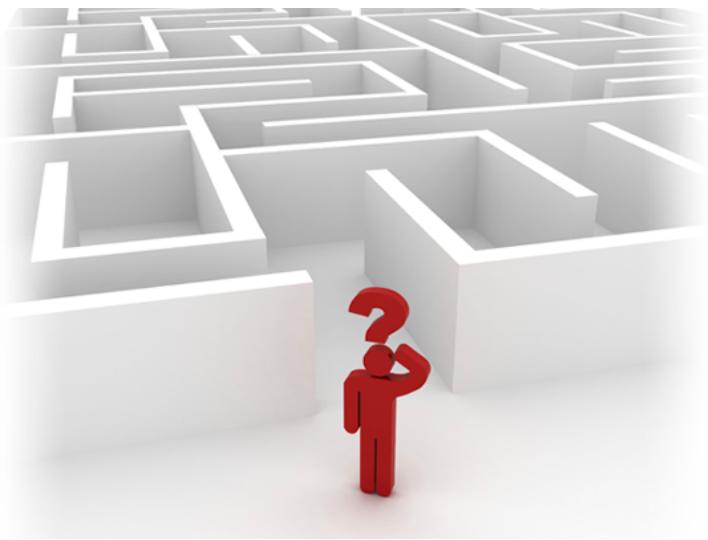
Upshot : running time of Merge on array of m numbers is $\leq 4m + 2$
 $\leq 6m$ (Since $m \geq 1$)

Running Time of Merge Sort

Claim : Merge Sort requires
 $\leq 6n \log_2 n + 6n$ operations
to sort n numbers.

Recall : $= \log_2 n$ is the #
of times you divide by 2
until you get down to 1





Design and Analysis
of Algorithms I

Introduction

Merge Sort

(Analysis)

Tim

Running Time of Merge Sort

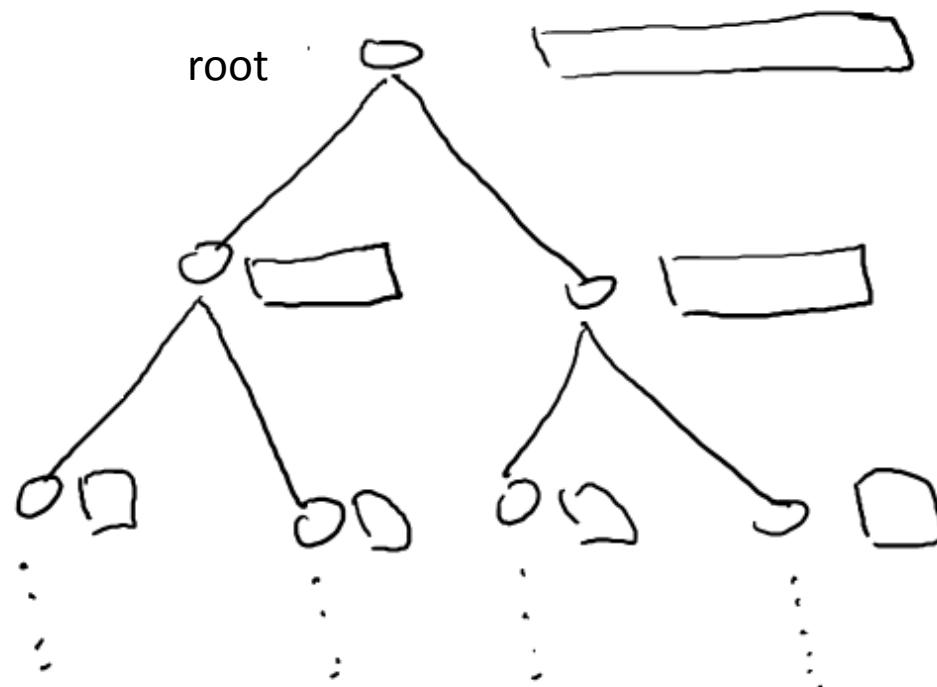
Claim: For every input array of n numbers, Merge Sort produces a sorted output array and uses at most $6n \log_2 n + 6n$ operations.

Proof of claim (assuming $n = \text{power of } 2$):

Level 0
[outer call to
Merge Sort]

Level 1
(1st recursive
calls)

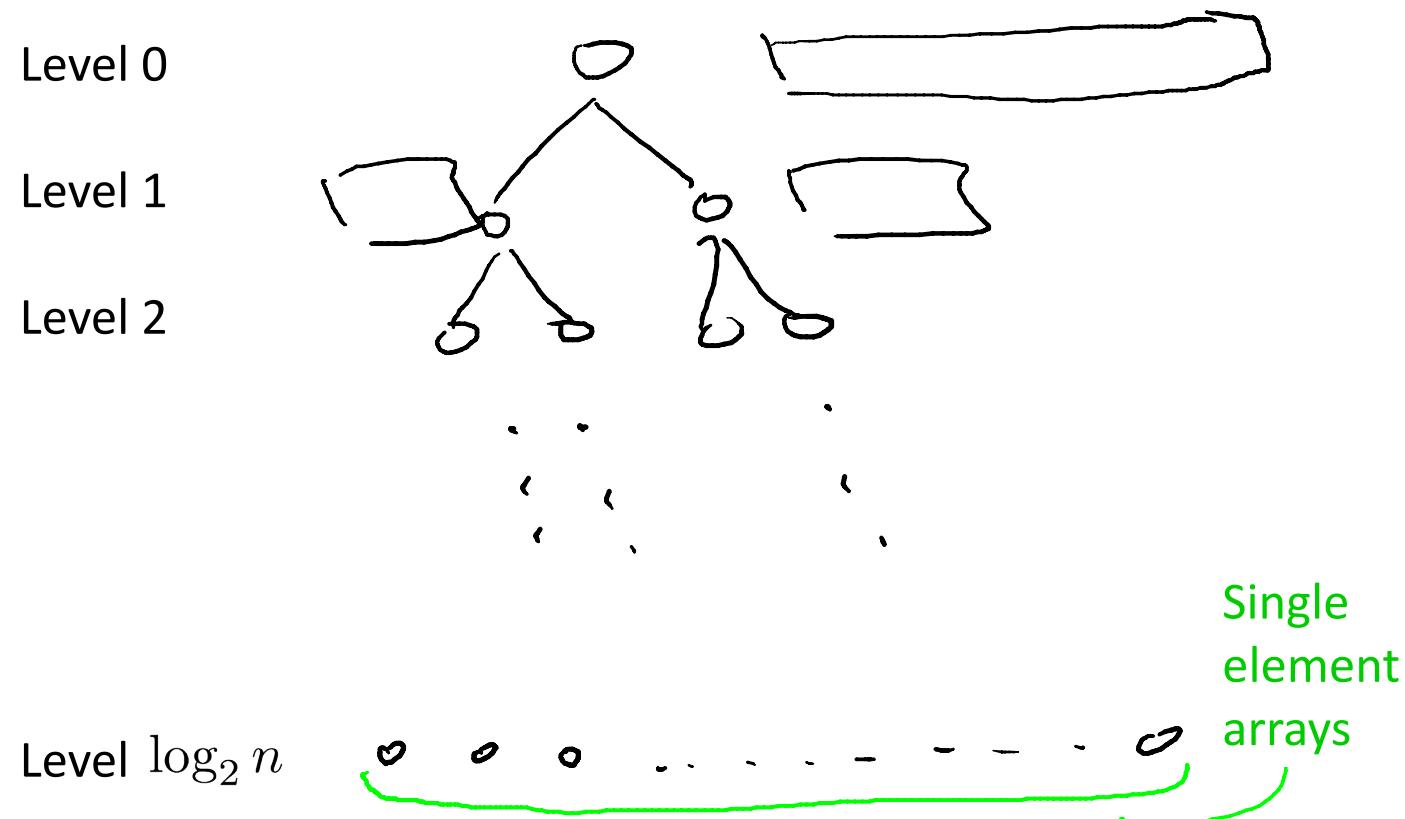
Level 2



Roughly how many levels does this recursion tree have (as a function of n , the length of the input array)?

- A constant number (independent of n).
- $\log_2 n$ $(\log_2 n + 1)$ to be exact!
- \sqrt{n}
- n

Proof of claim (assuming $n = \text{power of } 2$):



Tim Roughgarden

What is the pattern ? Fill in the blanks in the following statement: at each level $j = 0, 1, 2, \dots, \log_2 n$, there are <blank> subproblems, each of size <blank>.

- 2^j and 2^j , respectively
- $n/2^j$ and $n/2^j$, respectively
- 2^j and $n/2^j$, respectively
- $n/2^j$ and 2^j , respectively

Proof of claim (assuming $n = \text{power of 2}$) :

At each level $j=0,1,2,\dots, \log_2 n$,

Total # of operations at level $j = 0,1,2,\dots, \log_2 n$

$$\leq 2^j * 6\left(\frac{n}{2^j}\right) = 6n$$

of level-j
subproblems

Size of level-j
subproblem

Work per level – j
subproblem

Total

$$6n(\log_2 n + 1)$$

Work
per level

of
levels

Running Time of Merge Sort

Claim: For every input array of n numbers, Merge Sort produces a sorted output array and uses at most $6n \log_2 n + 6n$ operations.

QED!



Design and Analysis
of Algorithms I

Introduction --- Guiding Principles

Guiding Principle #1

“worst – case analysis” : our running time bound holds for every input of length n .

-Particularly appropriate for “general-purpose” routines

As Opposed to

--“average-case” analysis
--benchmarks

REQUIRES DOMAIN
KNOWLEDGE

BONUS : worst case usually easier to analyze.

Guiding Principle #2

Won't pay much attention to constant factors,
lower-order terms

Justifications

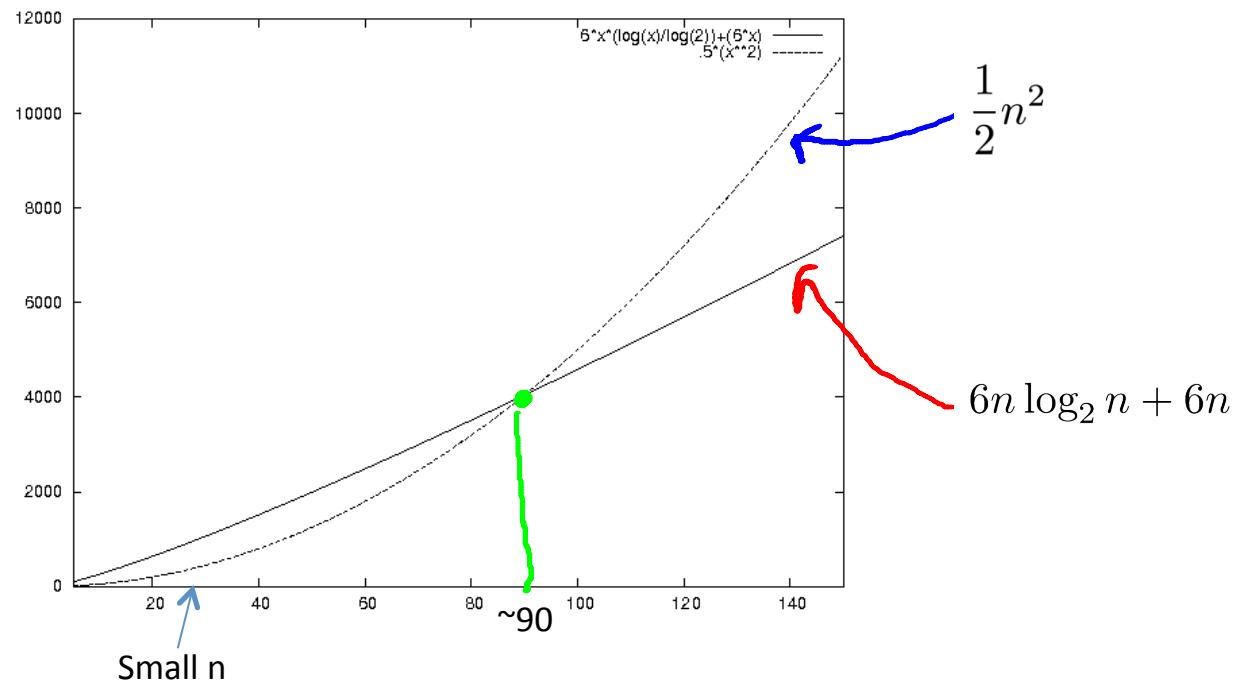
1. Way easier
2. Constants depend on architecture / compiler /
programmer anyways
3. Lose very little predictive power
(as we'll see)

Guiding Principle #3

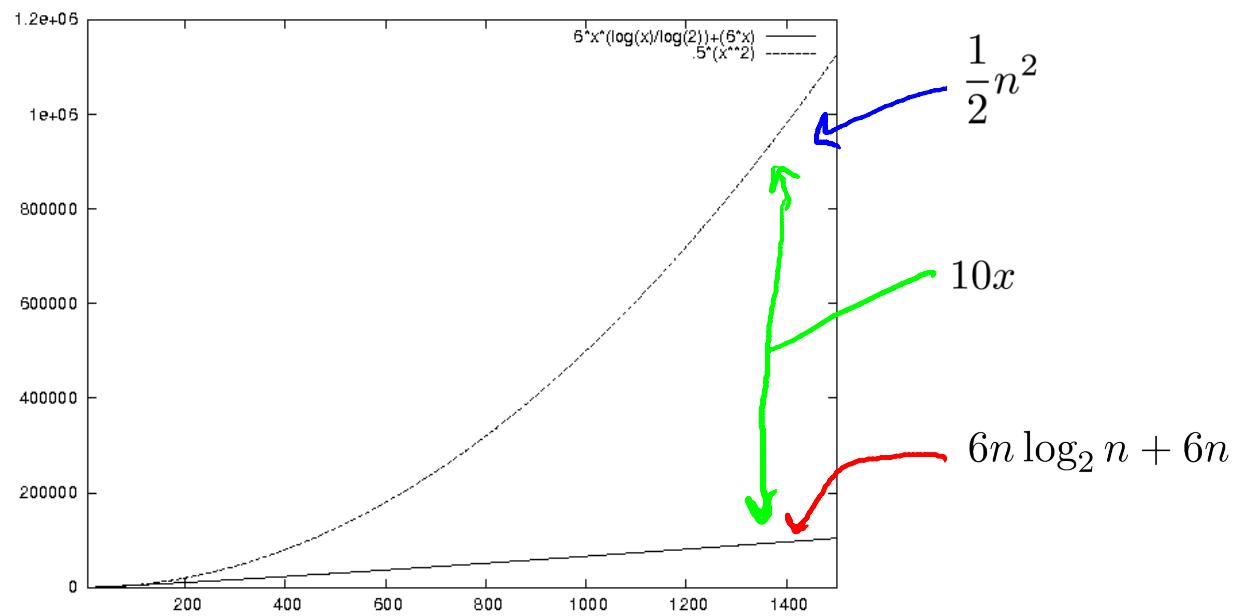
Asymptotic Analysis : focus on running time for large input sizes n

Eg : $\underbrace{6n \log_2 n + 6n}_{\text{MERGE SORT}}$ “better than” $\underbrace{\frac{1}{2}n^2}_{\text{INSERTION SORT}}$

Justification: Only big problems are interesting!



Tim Roughgarden



Tim Roughgarden

What Is a “Fast” Algorithm?

This Course : adopt these three biases as guiding principles

fast \approx worst-case running time
algorithm grows slowly with input size

Usually : want as close to linear ($O(n)$) as possible