Algorithms Specialization

Go To Course

About Syllabus Reviews Instructors Enrollment Options FAQ

About this Course

4.8 2,110 ratings • 409 reviews

The primary topics in this part of the specialization are: asymptotic ("Big-oh") notation, sorting and searching, divide and conquer (master method, integer and matrix multiplication, closest pair), and randomized algorithms (QuickSort, contraction algorithm for min cuts).

SKILLS YOU WILL GAIN

Algorithms Randomized Algorithm Sorting Algorithm Divide And Conquer Algorithms

Course 1 of 4 in the
Algorithms Specialization

100% online
Start instantly and learn at your own schedule.

Flexible deadlines
Reset deadlines in accordance to your schedule.

Intermediate Level

Approx. 21 hours to complete
Suggested: 4 weeks of study, 4-8 hours/week

English
Subtitles: English

Syllabus - What you will learn from this course

WEEK 1

3 hours to complete

Week 1

Introduction; "big-oh" notation and asymptotic analysis.

13 videos (Total 130 min), 3 readings, 2 quizzes SEE LESS

13 videos

Why Study Algorithms? 4m

Integer Multiplication 8m

Karatsuba Multiplication 12m

About the Course 17m

Merge Sort: Motivation and Example 8m

Merge Sort: Pseudocode 12m

Merge Sort: Analysis 9m

Guiding Principles for Analysis of Algorithms 15m

The Gist 14m

Big-Oh Notation 4m

Basic Examples 7m

Big Omega and Theta 7m

Additional Examples [Review - Optional] 7m

3 readings

Welcome and Week 1 Overview 10m

Overview, Resources, and Policies 10m

Lecture slides 10m

2 practice exercises

Problem Set #1 10m

Programming Assignment #1 2m

WEEK
2

3 hours to complete

Week 2

Divide-and-conquer basics; the master method for analyzing divide and conquer algorithms.

11 videos (Total 170 min), 2 readings, 2 quizzes SEE LESS

11 videos

O(n log n) Algorithm for Counting Inversions I 12m

O(n log n) Algorithm for Counting Inversions II 16m

Strassen's Subcubic Matrix Multiplication Algorithm 22m

O(n log n) Algorithm for Closest Pair I [Advanced - Optional] 31m

O(n log n) Algorithm for Closest Pair II [Advanced - Optional] 18m

Motivation 7m

Formal Statement 10m

Examples 13m

Proof I 9m

Interpretation of the 3 Cases 10m

Proof II 16m

2 readings

Week 2 Overview 10m

Optional Theory Problems (Batch #1) 10m

2 practice exercises

Problem Set #2 10m

Programming Assignment #2 2m

WEEK
3

3 hours to complete

Week 3

The QuickSort algorithm and its analysis; probability review

9 videos (Total 156 min), 1 reading, 2 quizzes SEE LESS

9 videos

QuickSort: Overview 12m

Partitioning Around a Pivot 24m

Correctness of Quicksort [Review - Optional] 10m

Choosing a Good Pivot 22m

Analysis I: A Decomposition Principle 21m

Analysis II: The Key Insight 11m

Analysis III: Final Calculations 8m

Probability Review I 25m

Probability Review II 17m

1 reading

Week 3 Overview 10m

2 practice exercises

Problem Set #3 10m

Programming Assignment #3 6m

WEEK
4

4 hours to complete

Week 4

Linear-time selection; graphs, cuts, and the contraction algorithm.

11 videos (Total 184 min), 3 readings, 3 quizzes SEE LESS

11 videos

Randomized Selection - Algorithm 21m

Randomized Selection - Analysis 20m

Deterministic Selection - Algorithm [Advanced - Optional] 16m

Deterministic Selection - Analysis I [Advanced - Optional] 22m

Deterministic Selection - Analysis II [Advanced - Optional] 12m

Omega(n log n) Lower Bound for Comparison-Based Sorting [Advanced - Optional] 13m

Graphs and Minimum Cuts 15m

Graph Representations 14m

Random Contraction Algorithm 8m

Analysis of Contraction Algorithm 30m

Counting Minimum Cuts 7m

3 readings

Week 4 Overview 10m

Optional Theory Problems (Batch #2) 10m

Info and FAQ for final exam 10m

3 practice exercises

Problem Set #4 10m

Programming Assignment #4 2m

Final Exam 20m

4.8 ★★★★★

409 Reviews >

24% started a new career after completing these courses

83% got a tangible career benefit from this course

20% got a pay increase or promotion

Top Reviews

★★★★★ By KS • SEP 14TH 2018

Well researched. Topics covered well, with walkthrough for example cases for each new introduced algorithm. Great experience, learned a lot of important algorithms and algorithmic thinking practices.

★★★★★ By CV • JUN 11TH 2017

A really exciting and challenging course. Loved the way the instructor explained everything with so much detail and precision. Definitely looking forward to the next course in the specialization.

Instructor

Tim Roughgarden

Professor

Computer Science

About Stanford University

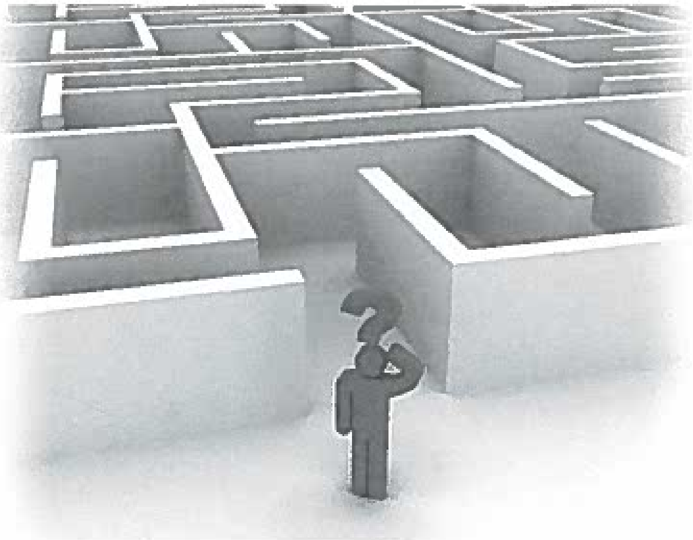
The Leland Stanford Junior University, commonly referred to as Stanford University or Stanford, is an American private research university located in Stanford, California on an
8,180-acre (3,310 ha) campus near Palo Alto, California, United States.

About the Algorithms Specialization

Algorithms are the heart of computer science, and the subject has countless practical applications as
well as intellectual depth. This specialization is an introduction to algorithms for learners with at least
a little programming experience. The specialization is rigorous but emphasizes the big picture and
conceptual understanding over low-level implementation and mathematical details. After co... MORE

Week 1

① July 11, 18



Introduction

Why Study Algorithms?

Design and Analysis of Algorithms I

What is an algo? set of well-defined rules for solving a computational prob.

why — 1

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

Tim Roughgarden

why_2

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

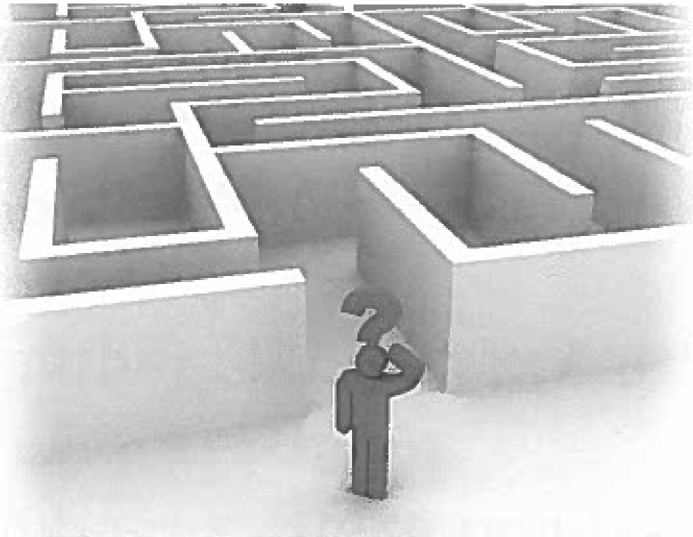
Why Study Algorithms?

- 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

July 11, 18 2



Design and Analysis
of Algorithms I

Introduction

Integer Multiplication

Integer - 1

The amount of needed to carry a task [illegible].

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

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

Roughly n operations per row up to a constant

# of operations overall ~ constant* n^2

The Algorithm Designer's Mantra

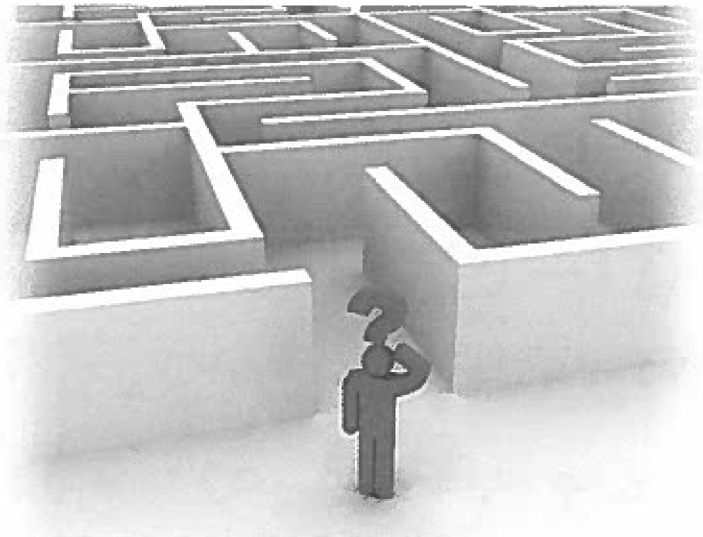
"Perhaps the most important principle for the good algorithm designer is to refuse to be content."

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

CAN WE DO BETTER ?
[than the "obvious" method]

there are better algo for integer multiplications.

July 12, 18 (3)



Design and Analysis of Algorithms I

Introduction

Karatsuba Multiplication

... we divide each no. into two parts

Example

$x = \underbrace{56}_{a}\underbrace{78}_{b}$

$y = \underbrace{12}_{c}\underbrace{34}_{d}$

multiply a.c

Step1: compute $a \cdot c = 672$

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

Step3: compute $(a+b)(c+d) = 134 \cdot 46 = 6164$

Step 4: compute $\underset{\text{step}}{3} - \underset{\text{step 2}}{2} - \underset{\text{step 1}}{1} = 2840$

Step 5:

step 1	6720000	pad step 1 with 4 zeros
step 2	2652	
step 4	284000	pad it with 2 zero
	7006652 $= (1234)(5678)$	

means an algorithm which invoke themselves as a subroutine with smaller input.

A Recursive Algorithm:

Write $\overset{5678}{x} = 10^{n/2}\overset{56}{a} + \overset{78}{b}$ and $y = 10^{n/2}\overset{12}{c} + \overset{34}{d}$ ← any n-digit number can be decomposed in terms of two integer nos.

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

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

Then $x.y = (10^{n/2}a + b)(10^{n/2}c + d)$

$$= (10^n ac + 10^{n/2}(ad + bc) + bd \quad (*)$$

assuming n (no. of digits) is even. If odd then one part has a different no. for e.g. 12345 → 123 45

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

Simple Base Case Omitted

What would be the base case? two 1-digit no.

Karatsuba Multiplication

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

only 3 quantities for recursive call

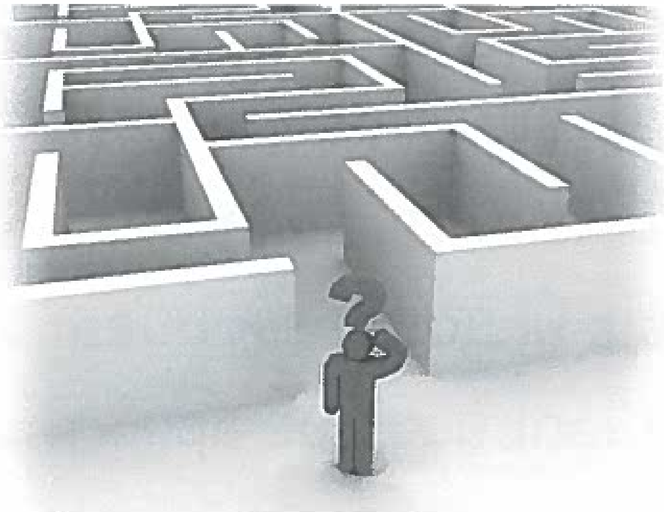
1. Recursively compute ac
2. Recursively compute bd
3. Recursively compute (a+b)(c+d) = ac+bd+ad+bc

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

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

Q : which is the fastest algorithm ? we need to wait for now

July 13, 18 4



Introduction

About The Course

Design and Analysis of Algorithms I

mostly an undergrad course

Intro 1

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 (performance metric like running time
    (granularity)
  - "sweet spot" for high-level reasoning about algorithms

the key point is to ignore constant factor and lower order terms, how it scales with large input sizes.

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")

break the problem into smaller problems which then gets solved recursively, and then somehow quickly combine the solutions to the subproblems into one for the original problem that you actually care about.

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

  vectors / arrays / lists / stacks / queues

  - 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.)

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.

You won't understand an algorithm until you coded up.

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. Free ✓
  - Cormen/Leiserson/Rivest/Stein, *Introduction to Algorithms*, 2009 (3rd edition).
  - Mehlhorn/Sanders, *Data Structures and Algorithms: The Basic Toolbox*, 2008.

Biggest influence on instructor

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

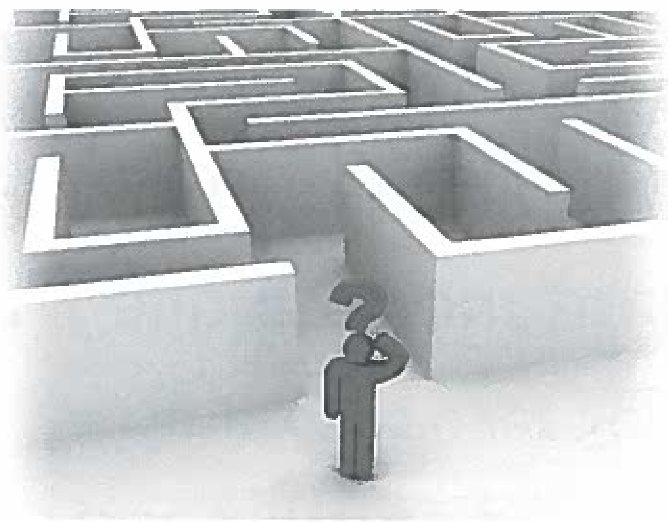
- No grades per se. (Details on a certificate of accomplishment TBA.)
- Weekly homeworks.
  - Test understand of material
  - Synchronize students, greatly helps discussion forum
  - Intellectual challenge

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

That is a very old algo. Von Neumann 1945

July 15, 2018 5



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

other avail. algo.

need them

- 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.

5	4	1	8	7	2	6	3

Assume **Distinct** numbers for now.

Output : Same numbers, sorted in increasing order / decreasing order.

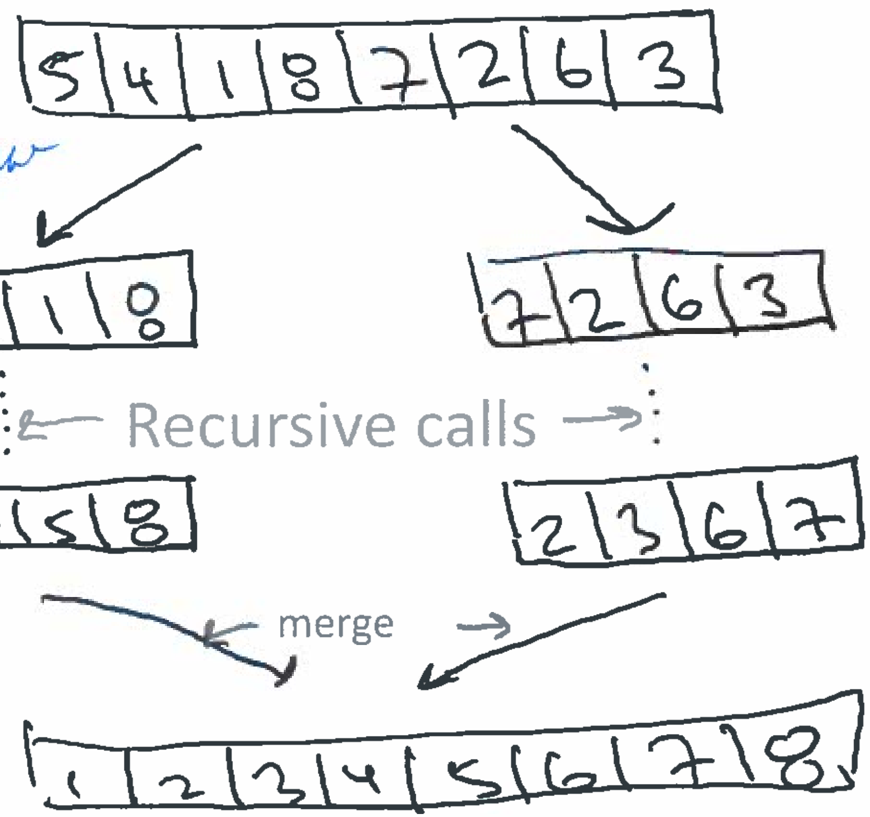
1	2	3	4	5	6	7	8

Tim Roughgarden

recursive calls

Merge Sort: Example

copy to new vectors



Step that is called merge

Tim Roughgarden

*A recursive algo. needs a base case.

Recursive Definition

Recursive Algorithm

*Anything recursive can be implemented iteratively, but recurs is a lot easier to read in most cases

Subjects to be Learned

- solving problem with recursive algorithm
- computing function with recursive algorithm
- Checking set membership with recursive algorithm

Contents

A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

If a set or a function is defined recursively, then a recursive algorithm to compute its members or values mirrors the definition. Initial steps of the recursive algorithm correspond to the basis clause of the recursive definition and they identify the basis elements. They are then followed by steps corresponding to the inductive clause, which reduce the computation for an element of one generation to that of elements of the immediately preceding generation.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Example 1: Algorithm for finding the k-th even natural number
Note here that this can be solved very easily by simply outputting $2*(k - 1)$ for a given k . The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: **Even**(positive integer k)
Input: k , a positive integer
Output: k-th even natural number (the first even being 0)

Algorithm:
if $k = 1$, **then** return 0;
else return **Even**(k-1) + 2 .

Here the computation of **Even**(k) is reduced to that of **Even** for a smaller input value, that is **Even**(k-1). **Even**(k) eventually becomes **Even**(1) which is 0 by the first line. For example, to compute **Even**(3), **Algorithm Even**(k) is called with $k = 2$. In the computation of **Even**(2), **Algorithm Even**(k) is called with $k = 1$. Since **Even**(1) = 0, 0 is returned for the computation of **Even**(2), and **Even**(2) = **Even**(1) + $2 = 2$ is obtained. This value 2 for **Even**(2) is now returned to the computation of **Even**(3), and **Even**(3) = **Even**(2) + $2 = 4$ is obtained.
As can be seen by comparing this algorithm with the recursive definition of the set of nonnegative even numbers, the first line of the algorithm corresponds to the basis clause of the definition, and the second line corresponds to the inductive clause.

By way of comparison, let us see how the same problem can be solved by an iterative algorithm.

Algorithm 1-a: Even(positive integer *k*)
Input: *k*, a positive integer
Output: *k*-th even natural number (the first even being *0*)

Algorithm:
```
int  i, even;
i := 1;
even := 0;
while( i < k ) {
        even := even + 2;
        i := i + 1;
}
return even .
```

Example 2: Algorithm for computing the *k*-th power of *2*

Algorithm 2 Power_of_2(natural number **k**)
Input: *k* , a natural number
Output: *k*-th power of *2*

Algorithm:
```
if k = 0, then return 1;
else return 2*Power_of_2(k - 1) .
```

By way of comparison, let us see how the same problem can be solved by an iterative algorithm.

Algorithm 2-a Power_of_2(natural number **k**)
Input: *k* , a natural number
Output: *k*-th power of *2*

Algorithm:
```
int  i, power;
i := 0;
power := 1;
while( i < k ) {
        power := power * 2;
        i := i + 1;
}
return power .
```

The next example does not have any corresponding recursive definition. It shows a recursive way of solving a problem.

Example 3: Recursive Algorithm for Sequential Search

Algorithm 3 SeqSearch(*L*, *i*, *j*, *x*)

Input: *L* is an array, *i* and *j* are positive integers, $i \leq j$, and *x* is the key to be searched for in *L*.

Output: If *x* is in *L* between indexes *i* and *j*, then output its index, else output *0*.

Algorithm:

if $i \leq j$ **, then**
{
  if $L(i) = x$, **then** return i **;**
  else return **SeqSearch(**L, $i+1$, j, x**)**
}
else return 0.

Recursive algorithms can also be used to test objects for membership in a set.

Example 4: Algorithm for testing whether or not a number x is a natural number

Algorithm 4 Natural(a number x)
Input: A number x
Output: "**Yes**" if x is a natural number, else "**No**"

Algorithm:
if $x < 0$, **then** return "**No**"
else
  if $x = 0$, **then** return "**Yes**"
  else return **Natural(** $x - 1$ **)**

Example 5: Algorithm for testing whether or not an expression w is a proposition(propositional form)

Algorithm 5 Proposition(a string w **)**

Input: A string w
Output: "**Yes**" if w is a proposition, else "**No**"

Algorithm:
if w is 1(true), 0(false), **or** a propositional variable, **then** return "**Yes**"
else if $w = \sim w_1$, **then** return **Proposition**(w_1)
  else
   if ($w = w_1 \vee w_2$ **or** $w_1 \wedge w_2$ **or** $w_1 \rightarrow w_2$ **or** $w_1 \leftrightarrow w_2$ **) and**
    Proposition(w_1) = **Yes and Proposition**(w_2) = **Yes**
   then return **Yes**
   else return **No**
end

Test Your Understanding of Recursive Algorithm

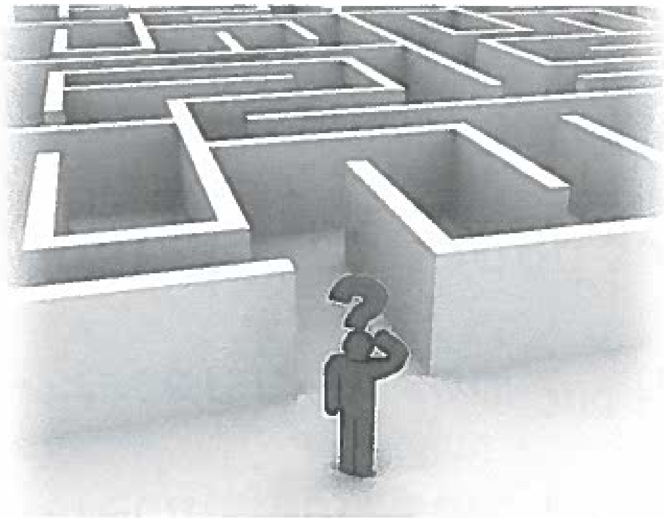
Indicate which of the following statements are correct and which are not.
Click Yes or No , then Submit.

[Next – First Principle of Mathematical Induction](#)

[Back to Schedule](#)
[Back to Table of Contents](#)

July 15, 18

6



Design and Analysis
of Algorithms I

Introduction

Merge Sort (Pseudocode)

// informal high level discription of the operation principle of a computer prog.

Tim

Pseudo — 1

· base case: when data is small you dont do any recursion and you return a trivial answer.

· In sorting base case:

Merge Sort: Pseudocode

-- recursively sort 1st half of the input array — If no. of elements are odd then one set gets one more element

-- recursively sort 2nd half of the input array

-- merge two sorted sublists into one → The difficult part

[ignores base cases] only one or two elements → return with no recursion.

emp

C

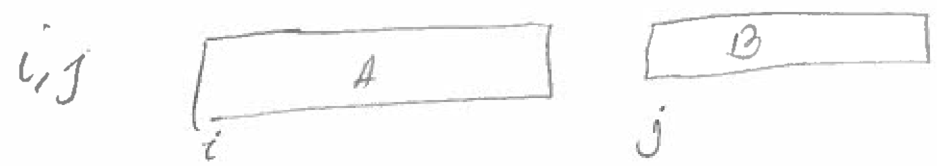
You populate the output array in sorted order by traversing pointers (or traversing through the sorted arrays in parallel.

A, B

Pseudocode for Merge:

The smallest array of output is either the smallest of A or B.

```
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)

great.

Tim Roughgarden

Pseudo - 2

running time? running algo in a debugger, anytime you press 'enter' you advance one line ⇒ running time ≡ no. enters/lines

Merge Sort Running Time?

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

[running time ~ # of lines of code executed]

Rather than looking at the total running time, we find the total of run time operation one merge only

Tim Roughgarden

• Imp: small ambiguity in counting # lines does not really matter

Pseudocode for Merge:

C = output [length = n]

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

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

1 i = 1

2 j = 1

2 operations

2 + 4 = 6

```
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
```

we operate inside the loop n X.

1, 2, 3: 3 operations, either this or that

4: k = k+1

(ignores end cases)

in each loop 4 iterations

Tim Roughgarden

Pseudo 3

Running Time of Merge

Upshot : running time of Merge on array of m numbers is $\leq 4m + 2$

$\leq 6m$ (Since $m \geq 1$) ✓

the same as "n"

(bcs we may merge smaller subproblems)

to make it easier (sloppy.

Tim Roughgarden

Running Time of Merge Sort

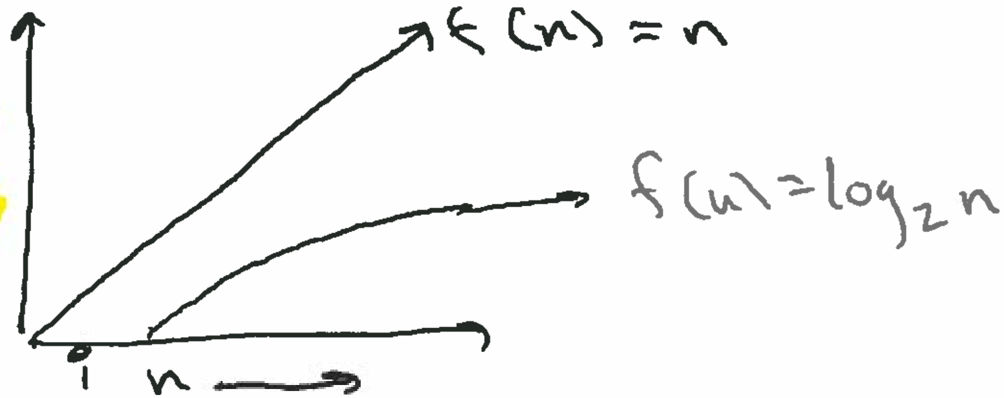
Claim : Merge Sort requires $\leq 6n\log_2 n + 6n$ operations to sort n numbers.

"As n grows the difference is large. much faster"

$x = \lg_2 n \Rightarrow 2^x = n$

imp

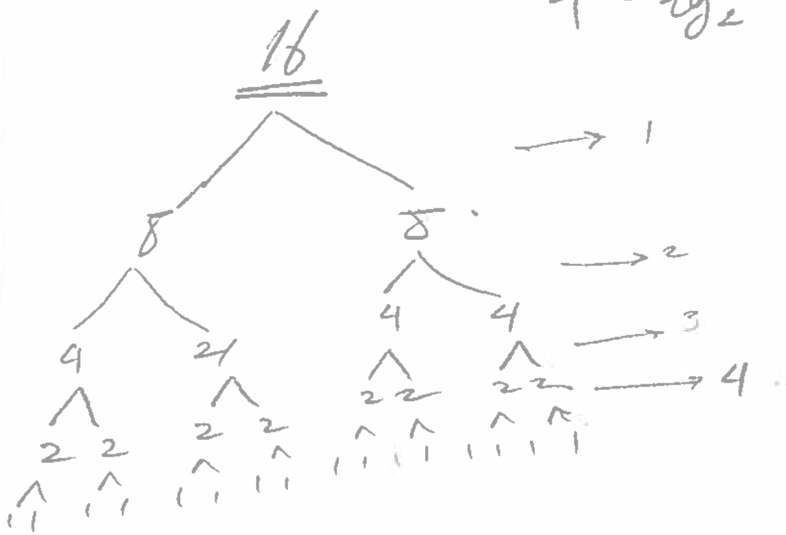
Recall : $= \log_2 n$ is the # of times you divide by 2 until you get down to 1



(See next page)

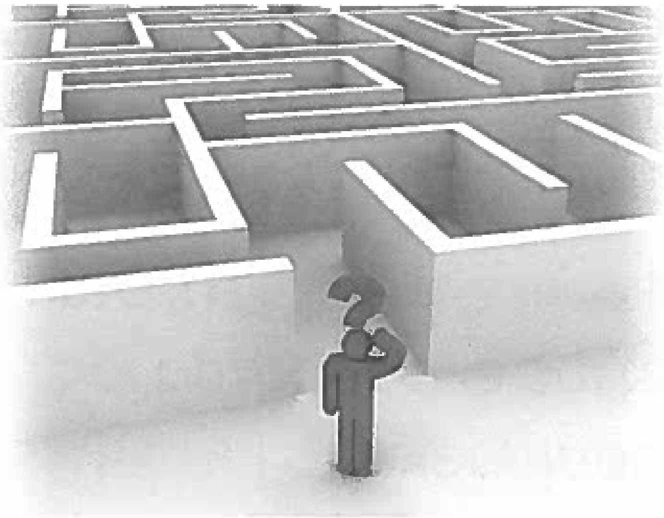
This is more efficient in comparison to "n^2"

$4 = \log_2 16$



July 15, 18

7



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.

recursive call

Proof of claim (assuming n = power of 2):

levels of recursion = divide n by 2 to get 1

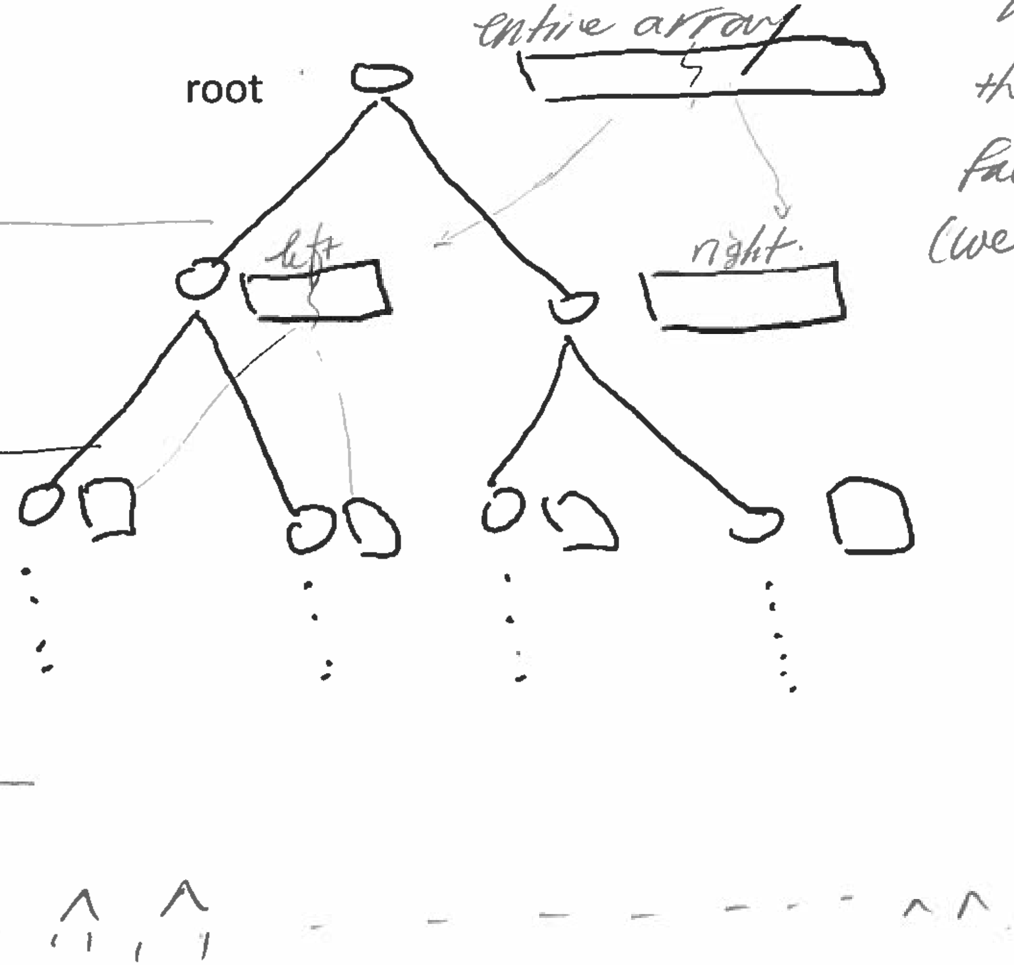
Level 0 = [outer call to Merge Sort]

Level 1 (1^{st} recursive calls)

Level 2

= level $\lg_2 n$

array of size "0" or 1



Tree structure help to count up the algo and facilitate the analysis (we count the work level by level.

total no of levels ($\lg_2 n + 1$)

Analysis 2

Ter
ver

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

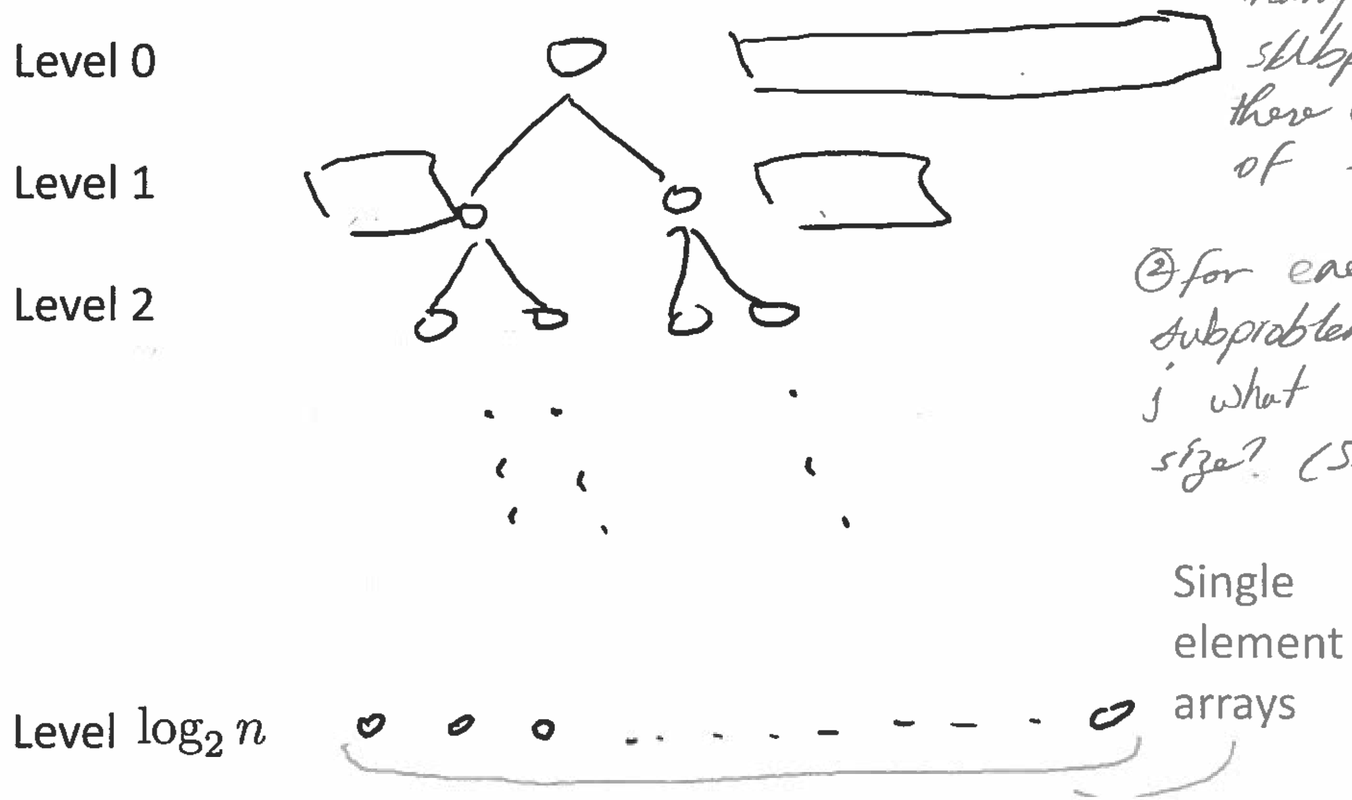
- ○ A constant number (independent of n).
- ○ $\log_2 n$ $(\log_2 n + 1)$ to be exact!
- ○ $\sqrt{n}$
- ○ n

The motivation for writing down/organizing the work performed by merge sort in this way is allows us to work level by level.

Proof of claim (assuming n = power of 2):

two questions.
① At level j, how many distinct subproblems are there (as a func of level j)? 2^j

② for each distinct subproblems at level j what is the input size? (size of array) $\frac{n}{2^j}$



$\frac{n}{2^j}$

Analysis 3

Te
ve

What is the pattern ? Fill in the blanks in the following statement: at each level j = 0,1,2,.., $\log_2 n$, 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*

we count the work level by level.

Proof of claim (assuming n = power of 2) :

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

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

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

6m we had before for merging.

independent of levels

# 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

great.

Tim Roughgarden

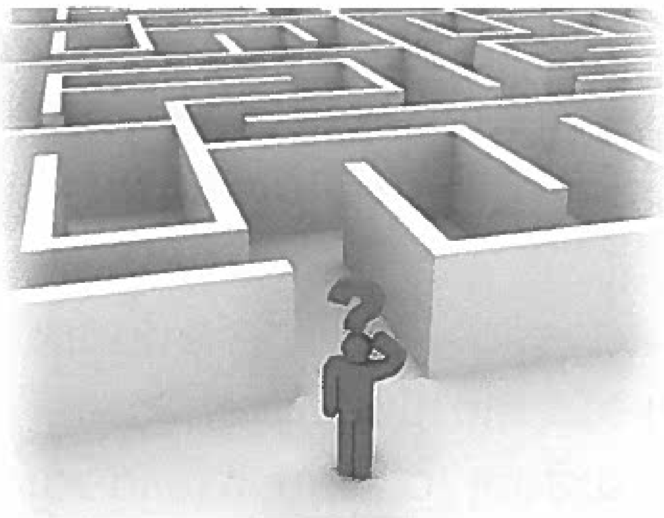
Analysis 4

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!

July 15, 18 8



Design and Analysis of Algorithms I

Introduction

Guiding Principles

[illegible] 1

Three assumptions we had for that analysis

Guiding Principle #1

1 "worst – case analysis" : our running time bound holds for every input of length n. no assumption on what the input is:

-Particularly appropriate for "general-purpose" routines

Other methods of analysis

As Opposed to

2 --"average-case" analysis

3 --benchmarks

REQUIRES DOMAIN KNOWLEDGE

we won't discuss them here.

BONUS : worst case usually easier to analyze. no assumptions good for general purposes.

Tim Roughgarden

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 e.g. how to count # of lines in a loop. imp

Inappropriate to consider.

3. Lose very little predictive power (as we'll see)

guide 2

Guiding Principle #3

means

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

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

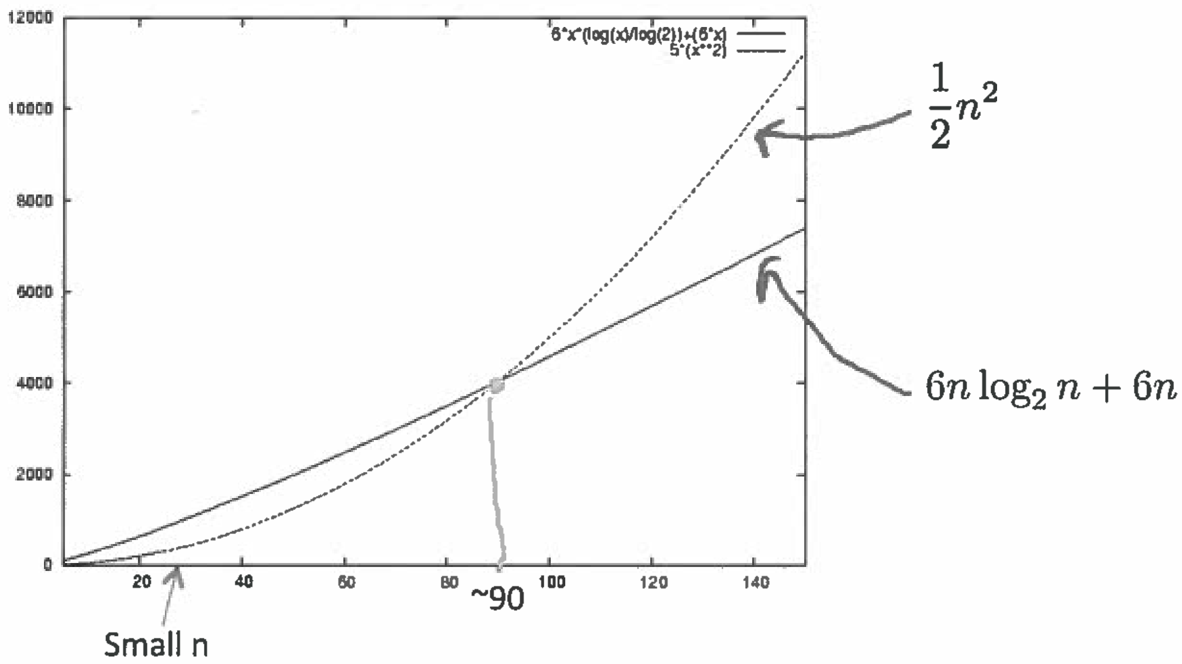
for small n ② < ①

large n ① < ②

Justification: Only big problems are interesting!

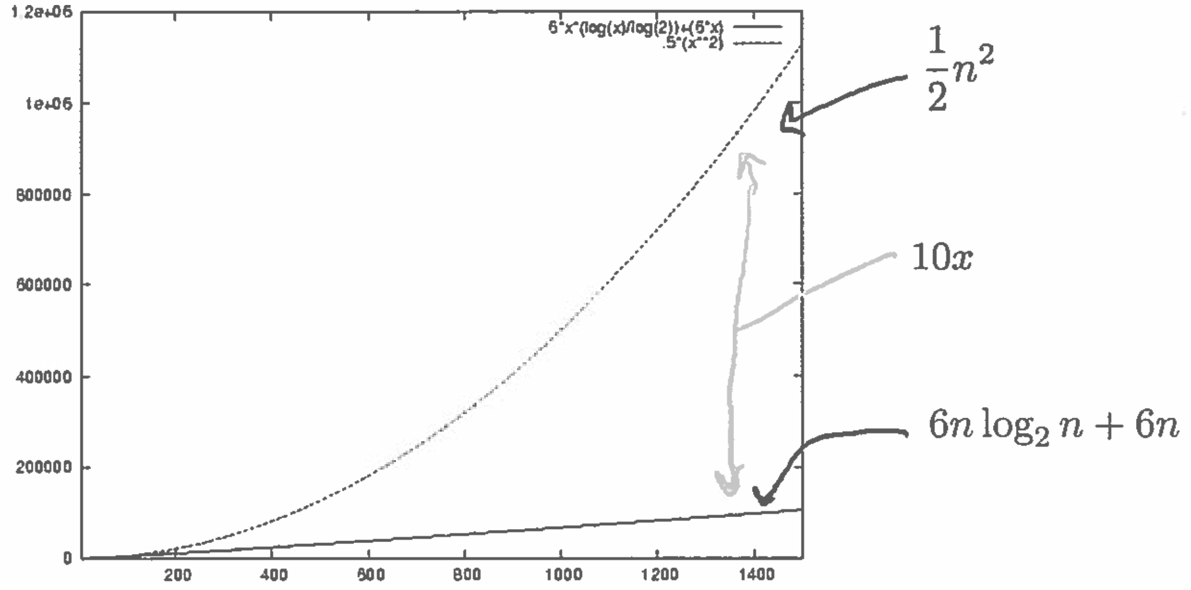
Tim Roughgarden

We can switch btw algs: for smaller size we use ---
for large size we use --



Tim Roughgarden

guide-3



Tim Roughgarden

What Is a "Fast" Algorithm?

This Course : adopt these three biases as guiding principles

imp

Def. of a

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

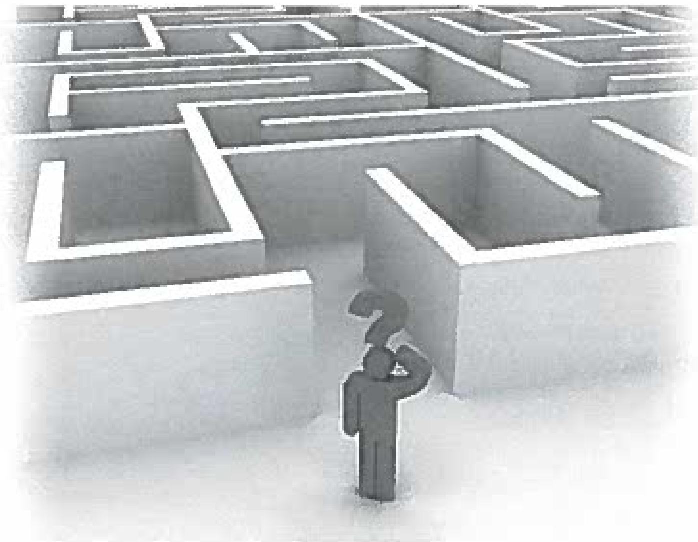
$\Downarrow$ means linear time (best case scenario)

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

Tim Roughgarden

guide 4

July 16, 2018



Design and Analysis
of Algorithms I

Asymptotic Analysis

The Gist

the essence of something

gist-1

Asymptotic analysis: discusses the high level performance of computer algo.

Motivation

Importance: Vocabulary for the design and analysis of algorithms (e.g. "big-Oh" notation).

means

- "Sweet spot" for high-level reasoning about algorithms.
- Coarse enough to suppress architecture/language/compiler-dependent details. ← we ignore this
- Sharp enough to make useful comparisons between different algorithms, especially on large inputs (e.g. sorting or integer multiplication). but include this

Asymptotic Analysis

imp

High-level idea: Suppress constant factors and lower-order terms 7 words

e.g. compiler/language ← too system-dependent

irrelevant for large inputs ⇐ great

Example: Equate $6n\log_2 n + 6n$ with just $n\log n$.

constant lower order

Terminology: Running time is $O(n\log n)$

how to pronounce it ["big-Oh" of $n\log n$]

where n = input size (e.g. length of input array).

gist-2

lines of code executed

good examples.

Example: One Loop

Problem: **Does array A contain the integer t?** Given A (array of length n) and t (an integer).

Algorithm 1

```
for i = 1 to n do
    if A[i] == t then
        Return TRUE
Return FALSE
```

Question: **What is the running time?**

A) $O(1)$ C) $O(n)$

B) $O(\log n)$ D) $O(n^2)$

- we consider the first worst case scenario.
- we want to know
- how many lines of code executed
- Where t exists in A?

Example: Two Loops

(the same as before, even though operations are not the same).

Given A, B (arrays of length n) and t (an integer). [Does A or B contain t?]

Algorithm 2

```
for i = 1 to n do
    if A[i] == t then
        Return TRUE
for i = 1 to n do
    if B[i] == t then
        Return TRUE
Return FALSE
```

Question: **What is the running time?**

A) $O(1)$ C) $O(n)$

B) $O(\log n)$ D) $O(n^2)$

Example: Two Nested Loops

Problem: Do arrays A, B have a number in common? Given arrays A, B of length n.

Algorithm 3

```
for i = 1 to n do
    for j = 1 to n do
        if A[i] == B[j] then
            Return TRUE
Return FALSE
```

Question: What is the running time?

A) $O(1)$ C) $O(n)$

B) $O(\log n)$ D) $O(n^2)$

very good

two loops:

$O(n^2)$.

Example: Two Nested Loops (II)

Problem: **Does array A have duplicate entries?** Given arrays A of length n.

Algorithm 4

```
1: for i = 1 to n do
2:    for j = i+1 to n do
3:       if A[i] == A[j] then
4:          Return TRUE
5: Return FALSE
```

if we start j from 1 then in fact we are comparing the elements twice

Question: **What is the running time?**

A) $O(1)$ C) $O(n)$

B) $O(\log n)$ D) $O(n^2)$

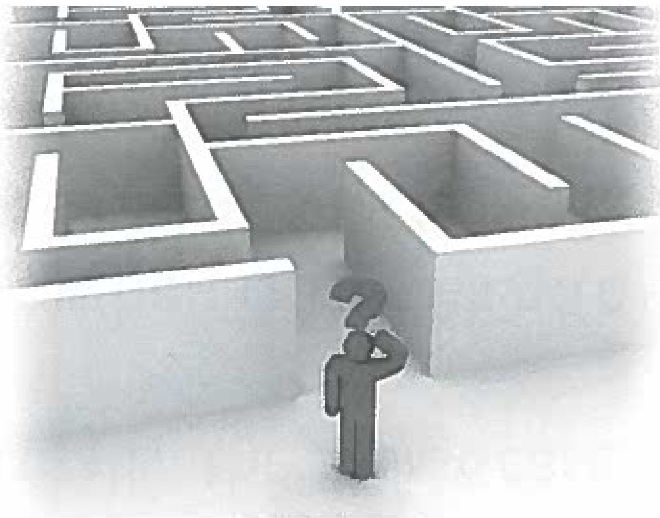
the difference with the previous ex. instead of counting twice, we count once $\Rightarrow$ a constant factor of $\frac{1}{2}$ $\Rightarrow$ still $O(n^2)$.

gist_4

july 16, 2018 10

Asymptotic Analysis

Big-Oh: Definition



Design and Analysis
of Algorithms I

B – 1

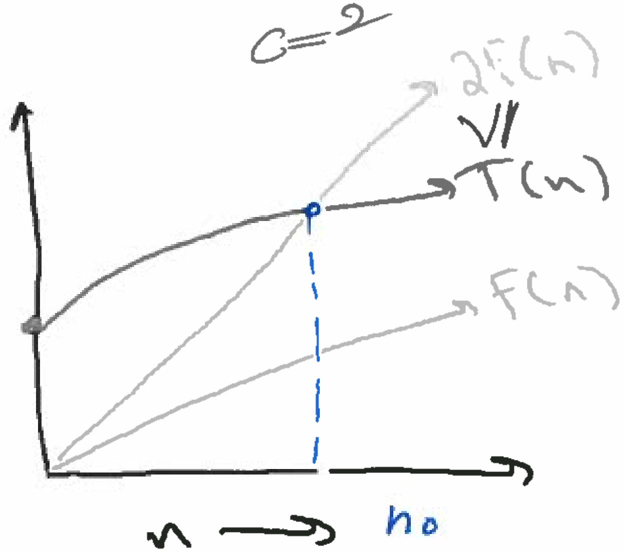
Big-Oh: English Definition

Let $T(n)$ = function on $n = 1,2,3,\ldots$
[usually, the worst-case running time of an algorithm]

Q : When is $T(n) = O(f(n))$? iff

A : if eventually (for all sufficiently large n), $T(n)$ is bounded above by a constant multiple of $f(n)$

Big-Oh: Formal Definition



Picture $T(n) = O(f(n))$

Formal Definition : T(n) = O(f(n)) if and only if there exist constants $c, n_0 > 0$ such that

$$T(n) \leq c \cdot f(n)$$

For all $n \geq n_0$

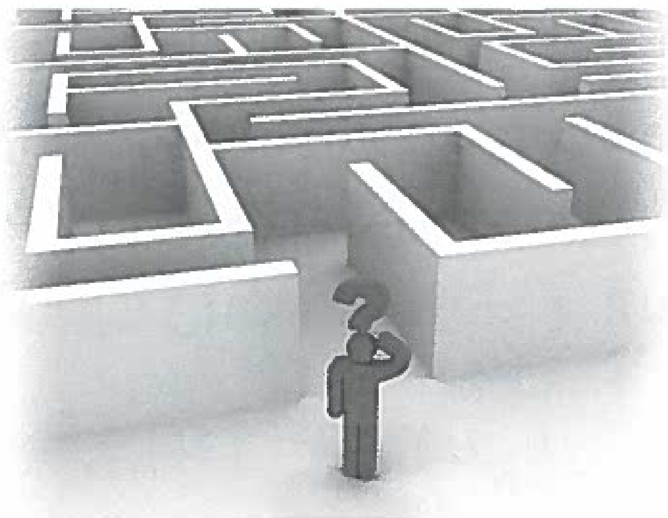
Warning : c, n_0 cannot depend on n

0-2

July 16, 2018 11

Asymptotic Analysis

Big-Oh: Basic Examples



Design and Analysis of Algorithms I

B.E_1

To prove it, we use reverse engineering to find c;

Example #1

Claim : if $T(n) = \underline{a_k n^k} + \ldots + a_1 n + a_0$ then

dominants the growth ✓

$$T(n) = O(\underbrace{n^k}_{f(n)})$$

Proof : Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + .. + |a_1| + |a_0|$

Need to show that $\forall n \geq 1, T(n) \leq c \cdot n^k$

We have, for every $n \geq 1$,

① we apply the abs value the summation increases →

② $n \leq n^k \quad \forall n \geq 1 \Rightarrow$

$$\boxed{T(n) \leq |a_k|n^k + \ldots + |a_1|n + |a_0|}$$

$$\boxed{\leq |a_k|n^k + \ldots + |a_1|n^k + |a_0|n^k} = \underbrace{(|a_k| + \cdots + |a_1|)}_{c}n^k$$

$$= c \cdot n^k$$

Tim Roughgarden

$$\boxed{T(n) \leq c\,(f(n))}$$

✓

Proof by contradiction

Example #2

Claim : for every $k \geq 1$, n^k is not $O(n^{k-1}) \equiv n^k \not\leq c \cdot (n^{k-1})$

Proof : by contradiction. Suppose $n^k = O(n^{k-1})$
Then there exist constants c, n_0 such that

we want to show

$$n^k \leq c \cdot n^{k-1} \quad \forall n \geq n_0$$

But then [cancelling n^{k-1} from both sides]:

All constant integers are bounded by a constant $c \equiv$ $n \leq c \quad \forall n \geq n_0$ we cannot find such "c".

wrong

Which is clearly False [contradiction].

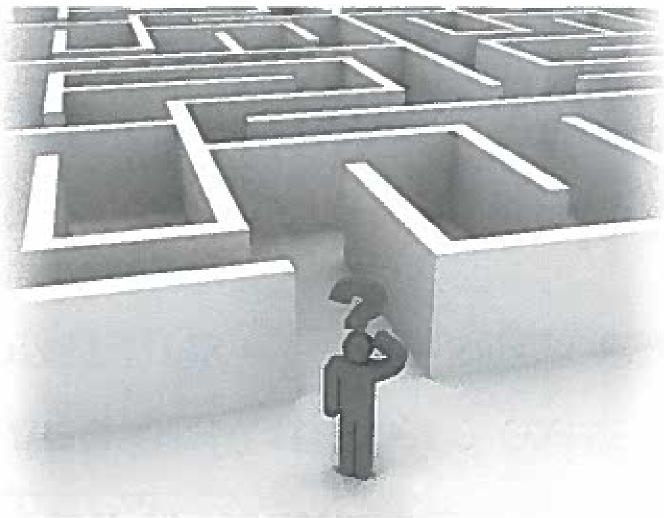
Tim Roughgarden

B.E.-2

July 16, 2018 (12)

Asymptotic Analysis

Big-Oh: Relatives (Omega & Theta)



Design and Analysis of Algorithms I

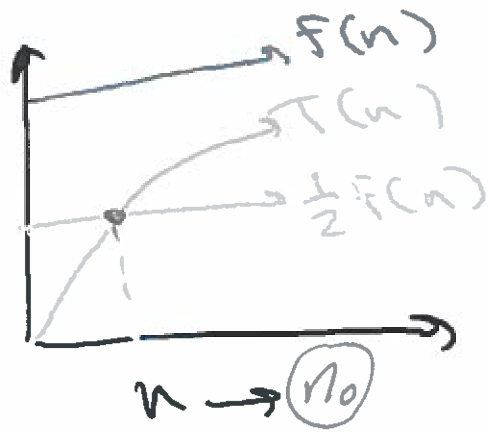
5 – 1

Omega Notation

Definition : $T(n) = \Omega(f(n))$
If and only if there exist constants c, n_0 such that

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

Picture



$$T(n) = \Omega(f(n))$$

Sloppy ≡ using O notation instead of Θ notation.
We usually care about the upper bound.

Theta Notation

<u>Definition</u> : $T(n) = \theta(f(n))$ if and only if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

<u>Equivalent</u> : there exist constants c_1, c_2, n_0 such that

$$c_1 f(n) \leq T(n) \leq c_2 f(n)$$

sandwich btw two constants

$$\forall n \geq n_0$$

Summary : $T(n) = O(f(n))$ if $T(n) \leq c\,(f(n))$ $\exists\, n_0, c_1$
$T(n) = \Omega(f(n))$ if $T(n) \geq c\, f(n)$ $\exists\, n_0, c_2$

Tim Roughgarden

Ω - 2

- $\frac{1}{2}n^2 + 3n \geq c\,n$ $\quad n_0 = 1,\ c = 1 \quad \Omega(n)$
- $\frac{1}{2}n^2 + 3n \leq c\,n^3$ $\quad n_0 = 1 \quad c = 4$
- $\frac{1}{2}n^2 + 3n \geq c\,n^2 \Rightarrow (\frac{1}{2} - c)n^2 \geq -3n$ $\quad n = 1 \Rightarrow \frac{1}{2} - c \geq -3 \quad -c \geq -3\frac{1}{2} \geq -4.$

$c = 4$

Let $T(n) = \dfrac{1}{2}n^2 + 3n$. Which of the following statements are true ? (Check all that apply.)

- ☐ $T(n) = O(n)$.
- ☐ $T(n) = \Omega(n)$. $[n_0 = 1, c = \frac{1}{2}]$

not a good lower bound, but legitimate

- ☐ $T(n) = \Theta(n^2)$. $[n_0 = 1, c_1 = 1/2, c_2 = 4]$
- ☐ $T(n) = O(n^3)$. $[n_0 = 1, c = 4]$

not a good upper bound, but legitimate

Little-Oh Notation

we won't use it much

<u>Definition</u> : $T(n) = o(f(n))$ if and only if for all constants c>0, there exists a constant n_0 such that

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

difference with big-O.
(here we need to show it for all constant c).

You should find n_0 based on c

<u>Exercise</u> : $\forall k \geq 1, n^{k-1} = o(n^k)$

Solution, we want to show that $n^{k-1} \leq c\, n^k \quad \forall c$:

$1 \leq c n_0$ this is correct for $\forall c$, if $n_0 = 1$

$\Rightarrow \boxed{n_0 = \left(\frac{1}{c}\right)}$ rounded up to the nearest int

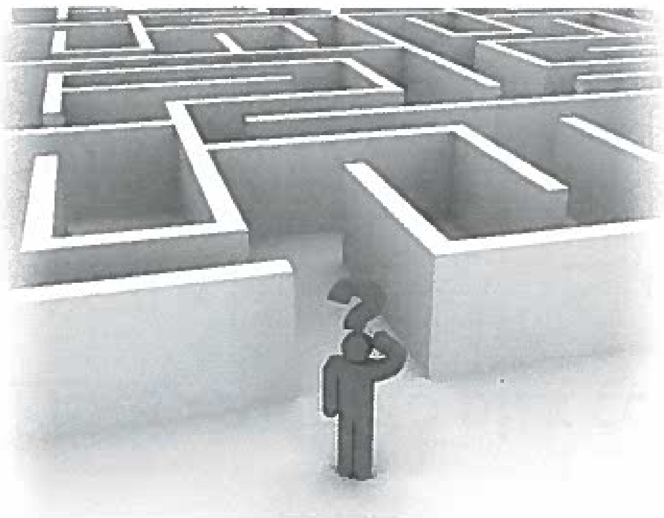
~3

Where Does Notation Come From?

- "On the basis of the issues discussed here, I propose that members of SIGACT, and editors of compter science and mathematics journals, adopt the O, Ω, and Θ notations as defined above, unless a better alternative can be found reasonably soon".

-D. E. Knuth, "Big Omicron and Big Omega and Big Theta", SIGACT News, 1976. Reprinted in "Selected Papers on Analysis of Algorithms."

July 16, 2018 13



Design and Analysis
of Algorithms I

Asymptotic Analysis

Additional Examples

Ex_1

Example #1

Claim : $2^{n+10} = O(2^n)$

we reverse engineer to find c

Proof : need to pick constants c, n_0 such that

$$(*) \quad 2^{n+10} \leq c \cdot 2^n \quad n \geq n_0$$

simple

Note : $2^{n+10} = 2^{10} \times 2^n = (1024) \times 2^n$

So if we choose $c = 1024, n_0 = 1$ then (*) holds.

Q.E.D

Example #2

<u>Claim</u> : $2^{10n} \neq O(2^n)$

<u>Proof</u> : by contradiction. If $2^{10n} = O(2^n)$ then there exist constants $c, n_0 > 0$ such that

$$2^{10n} \leq c \cdot 2^n \quad n \geq n_0$$

But then [cancelling 2^n]

$$2^{9n} \leq c \quad \forall n \geq n_0$$

Which is certainly false. <u>Q.E.D</u>

Tim Roughgarden

EX-2

Imp: There is no difference btw taking the pointwise max. of two nonnegative functions and taking their sum.

Example #3

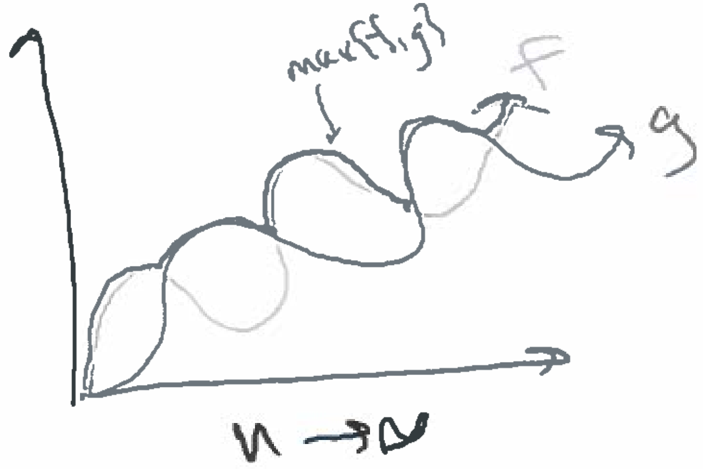
<u>Claim</u> : for every pair of (positive) functions f(n), g(n),
≥ 0 ≥ 0
$n \geq n_0$

no matter

$$max\{f, g\} = \theta(f(n) + g(n))$$

$c_1 = 1$ f(n)+ g(n)

c_2 min(f(n),



Tim Roughgarden

Example #3 (continued)

Proof : $max\{f,g\} = \theta(f(n)+g(n))$

For every n, we have

always true $$max\{f(n),g(n)\} \le f(n)+g(n) \Rightarrow c_2 = 1$$

And

$$2 * max\{f(n),g(n)\} \ge f(n)+g(n)$$

Thus $c_1 = \frac{1}{2} * (f(n)+g(n)) \le max\{f(n),g(n)\} \le f(n)+g(n) \quad \forall n \ge 1$

$=> max\{f,g\} = \theta(f(n)+g(n))$ [where $n_0 = 1, c_1 = 1/2, c_2 = 1$]

Q.E.D

Tim Roughgarden

EX-3