



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?

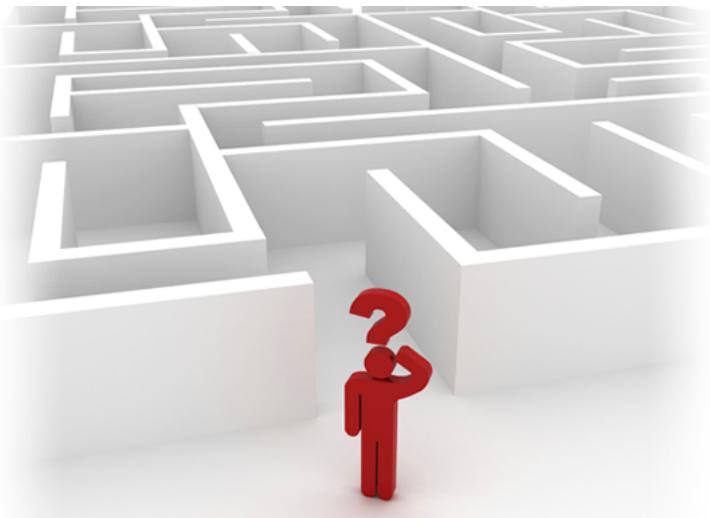
- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel “lens” on processes outside of computer science and technology
 - quantum mechanics, economic markets, evolution

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 arrow points from the text "Roughly n operations per row up to a constant" to the circled area.

Roughly n operations per row up to a constant

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

$x = \underline{\underline{5}} \underline{\underline{6}} \underline{\underline{7}} \underline{\underline{8}}$
 $y = \underline{\underline{1}} \underline{\underline{2}} \underline{\underline{3}} \underline{\underline{4}}$

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:

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

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

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

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

Simple Base Case
Omitted

Karatsuba Multiplication

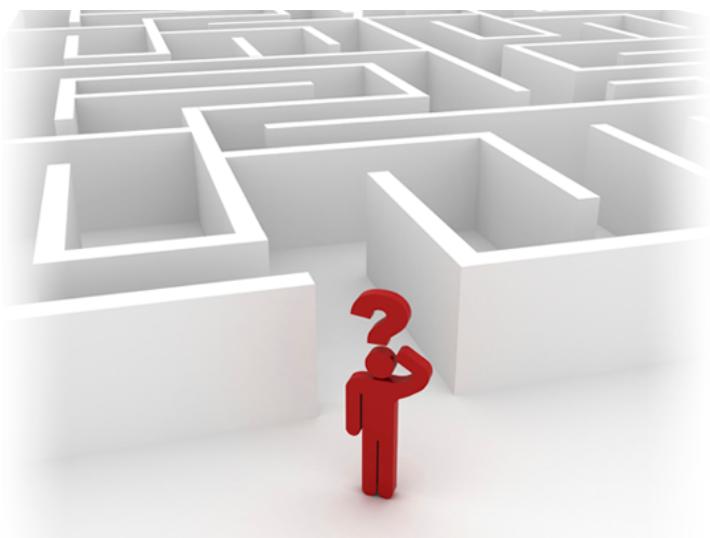
$$x \cdot y = (10^n ac + 10^{n/2}(ad + bc) + bd$$

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

Tim Roughgarden

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”

Tim Roughgarden

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.

Biggest influence
on instructor

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

Tim Roughgarden

Supporting Materials

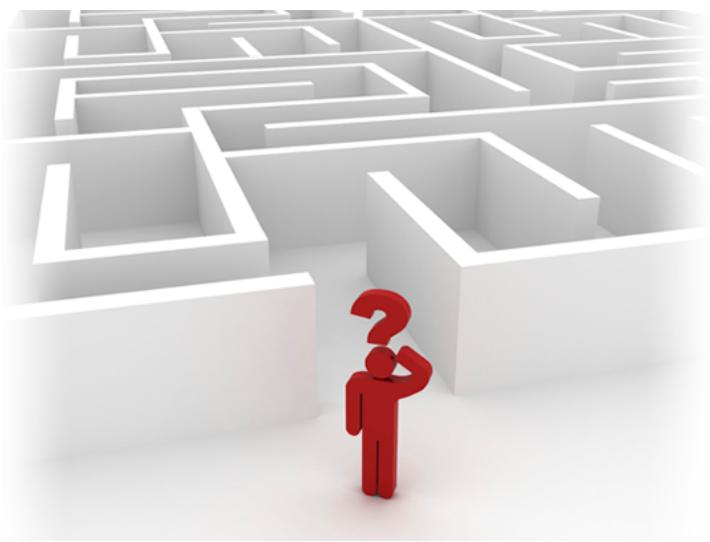
- All (annotated) slides available from course site.
- No required textbook. A few of the many good ones:
 - Kleinberg/Tardos, *Algorithm Design*, 2005.
 - Dasgupta/Papadimitriou/Vazirani, *Algorithms*, 2006.
 - Cormen/Leiserson/Rivest/Stein, *Introduction to Algorithms*, 2009 (3rd edition).
 - Mehlhorn/Sanders,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

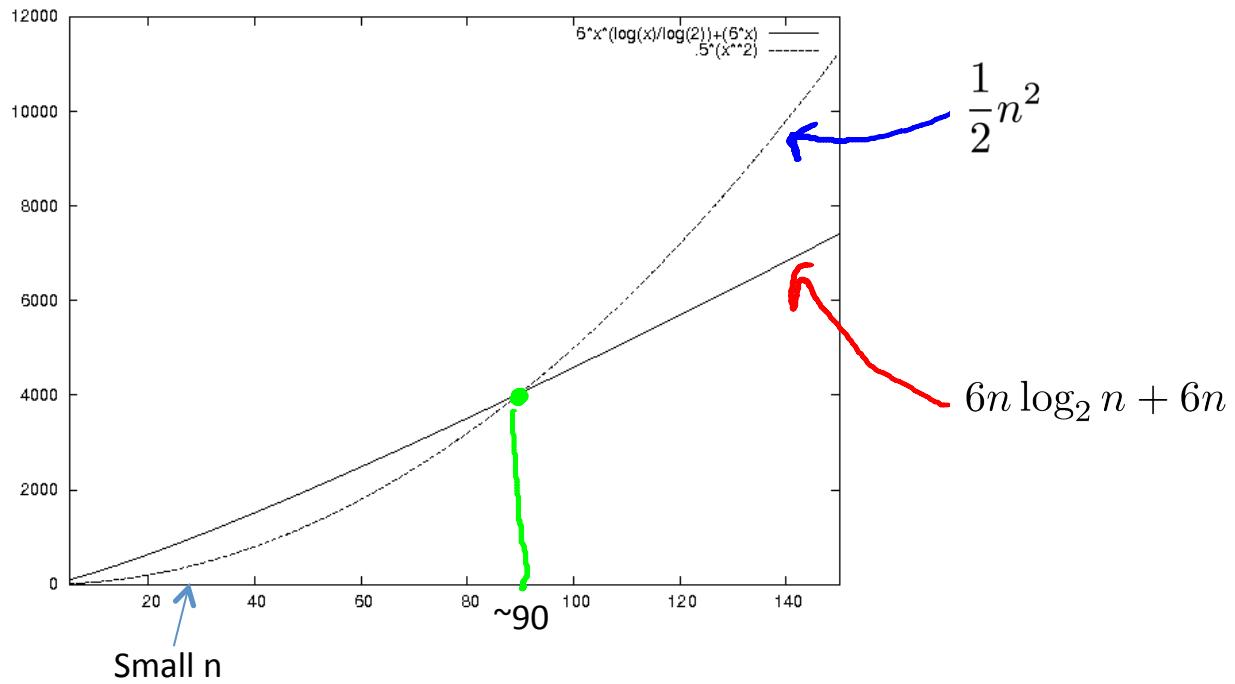
1. Way easier
2. Constants depend on architecture / compiler /
programmer anyways
3. Lose very little predictive power
(as we'll see)

Guiding Principle #3

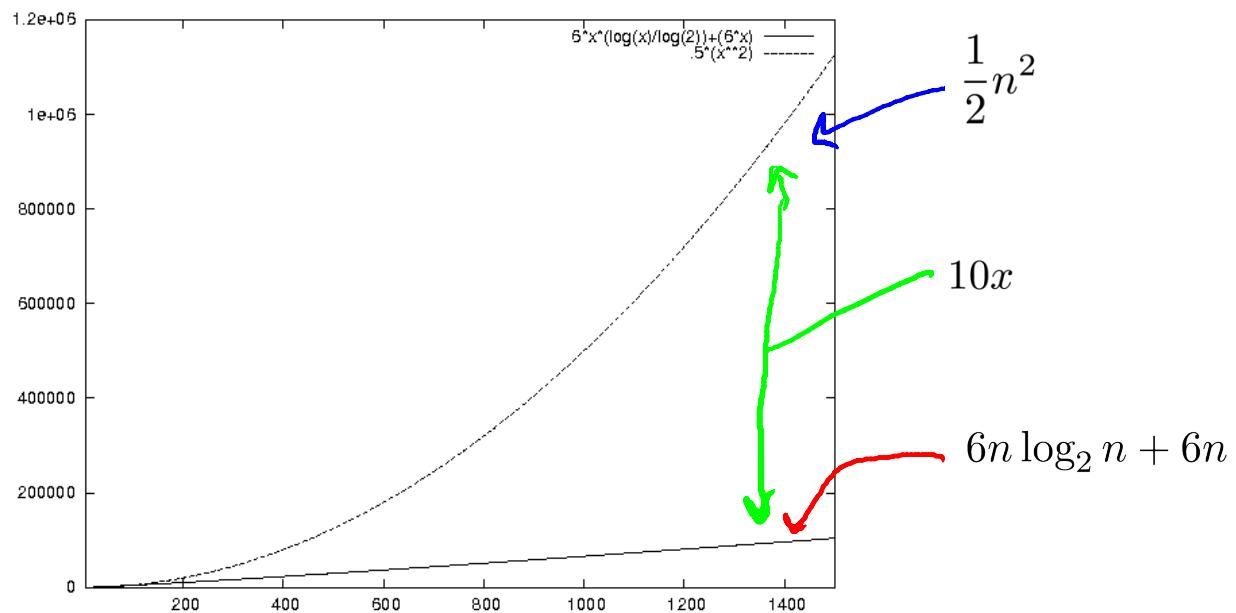
Asymptotic Analysis : focus on running time for large input sizes n

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



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



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.

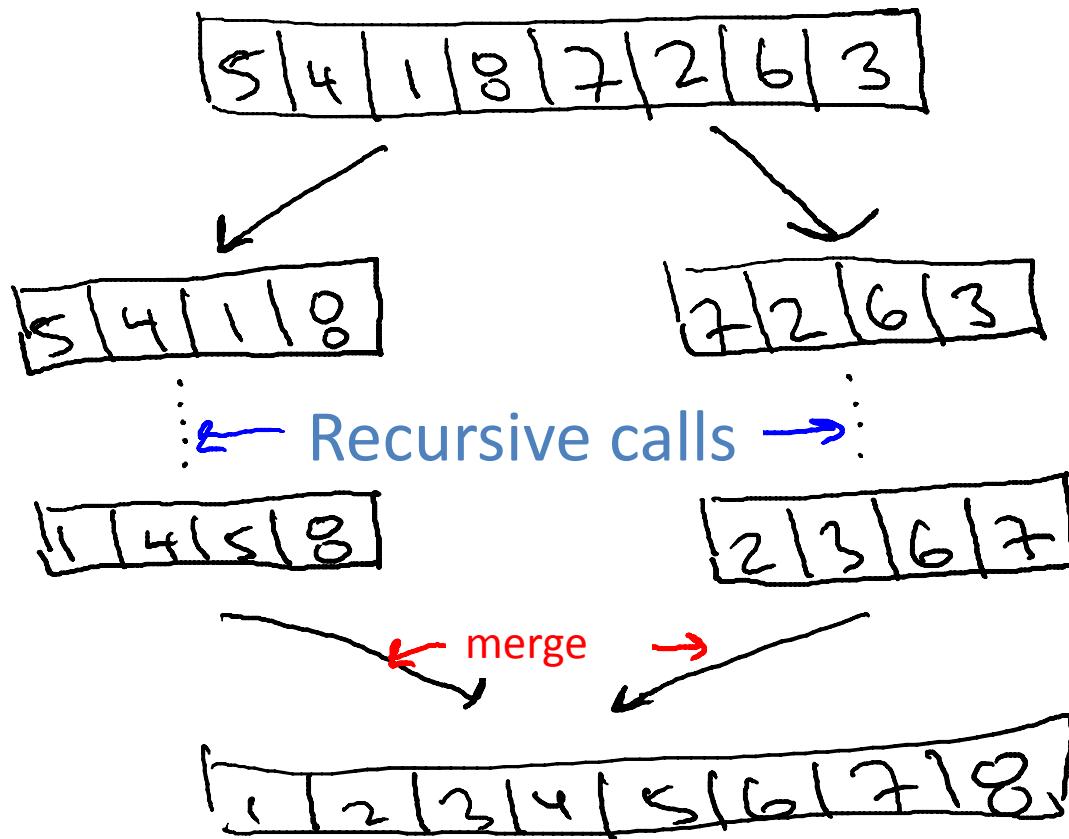
15|4|11|8|7|2|6|3|

Assume
Distinct

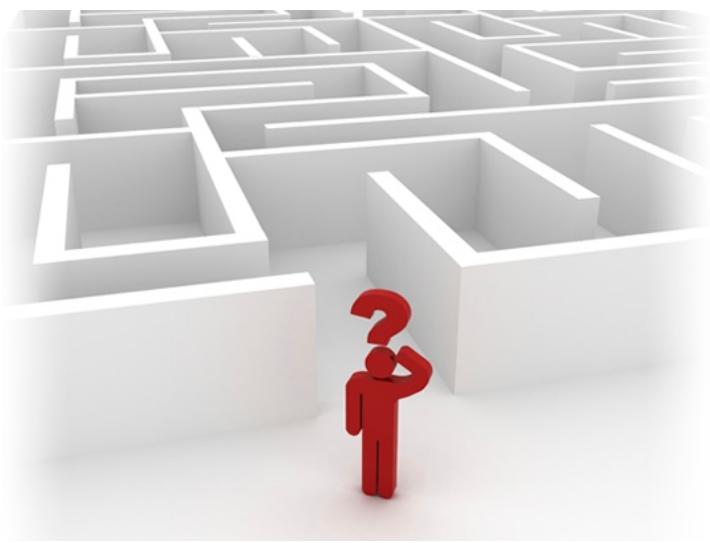
Output : Same numbers, sorted in increasing order

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 = output [length = n]

A = 1st sorted array [n/2]

B = 2nd 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)

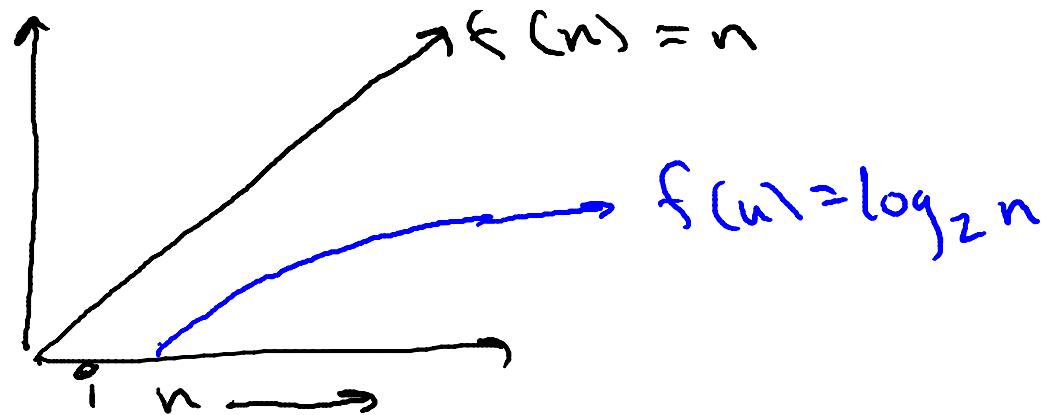
Running Time of Merge

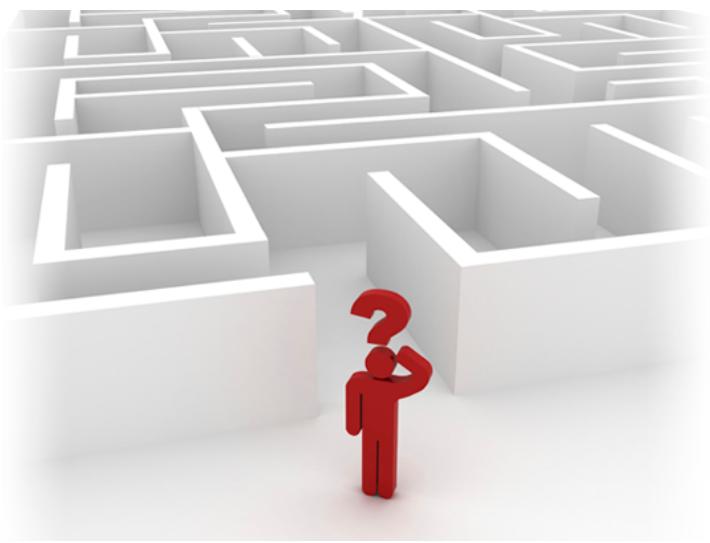
Upshot : running time of Merge on array of m numbers is $\leq 4m + 2$
 $\leq 6m$ (Since $m \geq 1$)

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

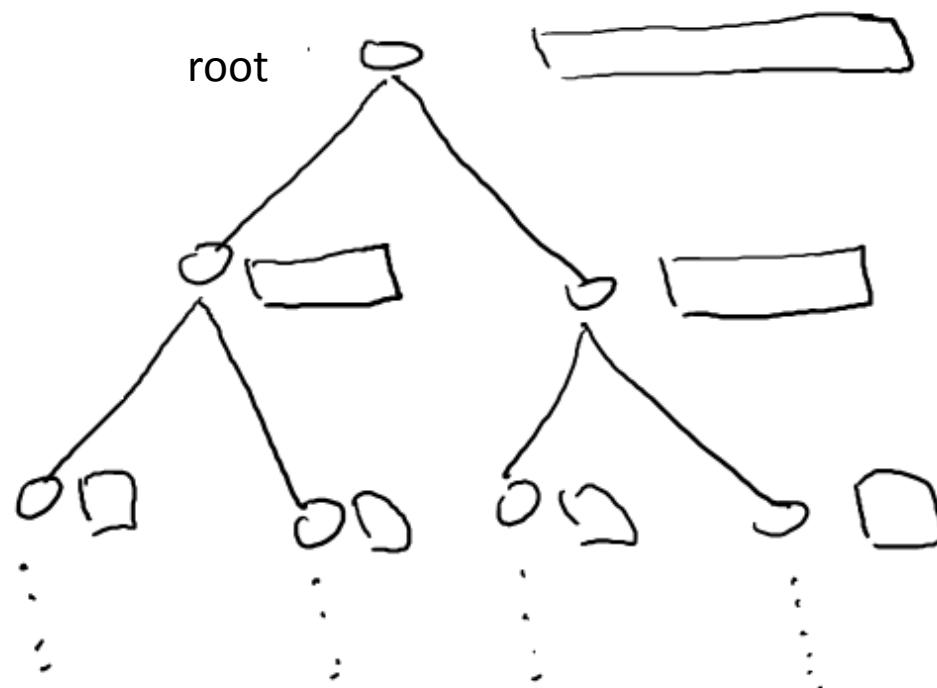
Claim: For every input array of n numbers, Merge Sort produces a sorted output array and uses at most $6n \log_2 n + 6n$ operations.

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

Level 0
[outer call to
Merge Sort]

Level 1
(1st recursive
calls)

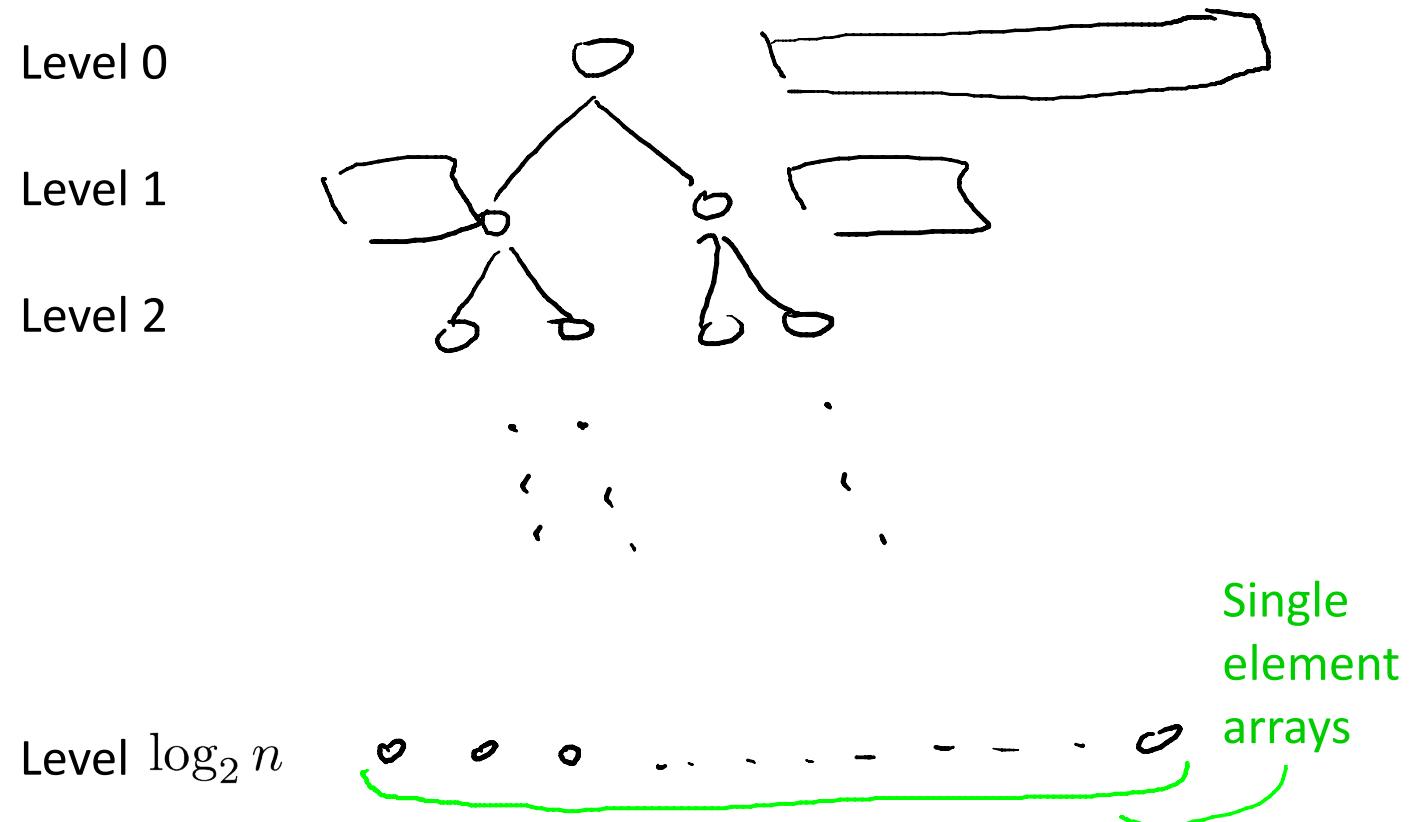
Level 2



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!



Asymptotic Analysis

The Gist

Design and Analysis
of Algorithms I

Motivation

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

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

Asymptotic Analysis

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

too system-dependent

irrelevant for large inputs

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

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

[“big-Oh” of $n \log n$]

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

Example: One Loop

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

Algorithm 1

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

Question: What is the running time?

- A) $O(1)$
- C) $O(n)$
- B) $O(\log n)$
- D) $O(n^2)$

Example: Two Loops

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

Algorithm 2

```
1: for  $i = 1$  to  $n$  do
2:   if  $A[i] == t$  then
3:     Return TRUE
4: for  $i = 1$  to  $n$  do
5:   if  $B[i] == t$  then
6:     Return TRUE
7: 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

```
1: for  $i = 1$  to  $n$  do
2:   for  $j = 1$  to  $n$  do
3:     if  $A[i] == B[j]$  then
4:       Return TRUE
5: 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 (II)

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

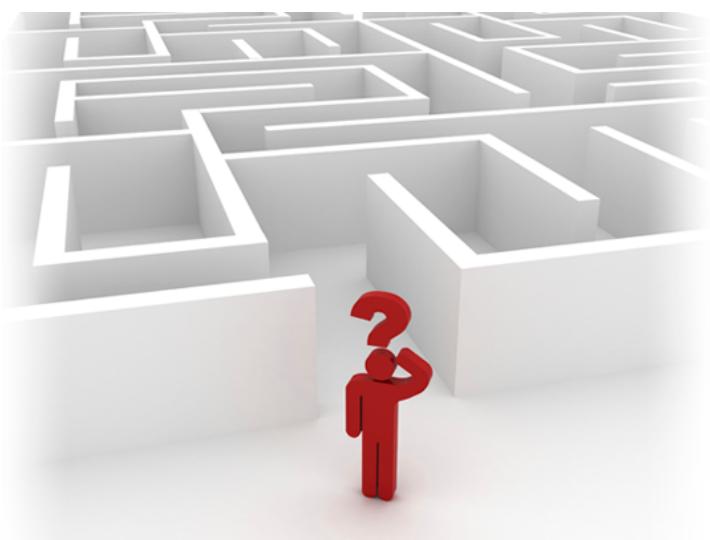
Algorithm 4

```
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
```

Question: What is the running time?

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

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



Design and Analysis
of Algorithms I

Asymptotic Analysis

Big-Oh: Definition

Big-Oh: English Definition

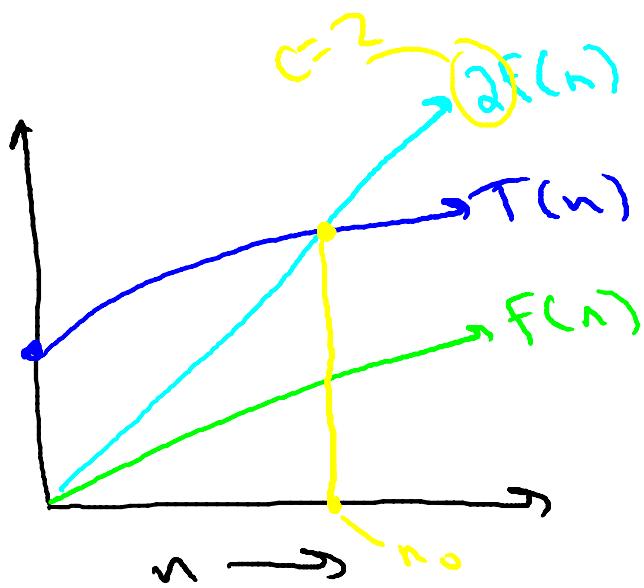
Let $T(n) = \text{function on } n = 1, 2, 3, \dots$

[usually, the worst-case running time of an algorithm]

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

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

Big-Oh: Formal Definition



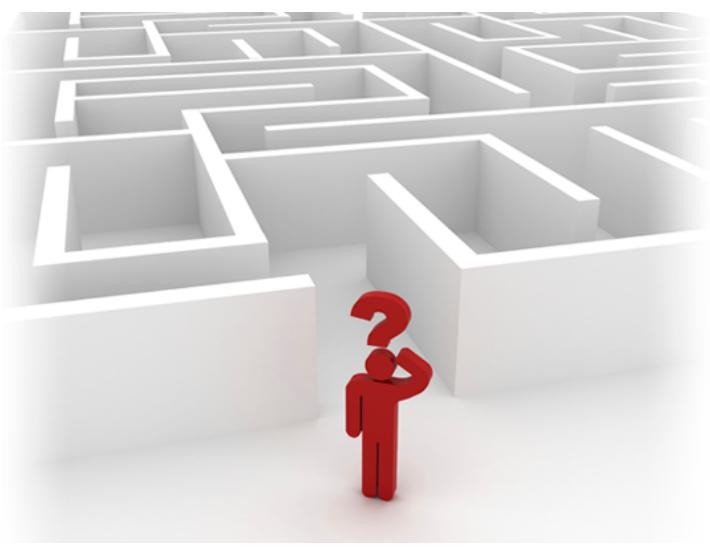
Formal Definition : $T(n) = O(f(n))$ if and only if there exist constants $c, n_0 > 0$ such that

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

For all $n \geq n_0$

Warning : c, n_0 cannot depend on n

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



Design and Analysis
of Algorithms I

Asymptotic Analysis

Big-Oh: Basic Examples

Example #1

Claim : if $T(n) = a_k n^k + \dots + a_1 n + a_0$ then

$$T(n) = O(n^k)$$

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

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

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

$$\begin{aligned} T(n) &\leq |a_k|n^k + \dots + |a_1|n + |a_0| \\ &\leq |a_k|n^k + \dots + |a_1|n^k + |a_0|n^k \\ &= c \cdot n^k \end{aligned}$$

Example #2

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

Proof : by contradiction. Suppose $n^k = O(n^{k-1})$

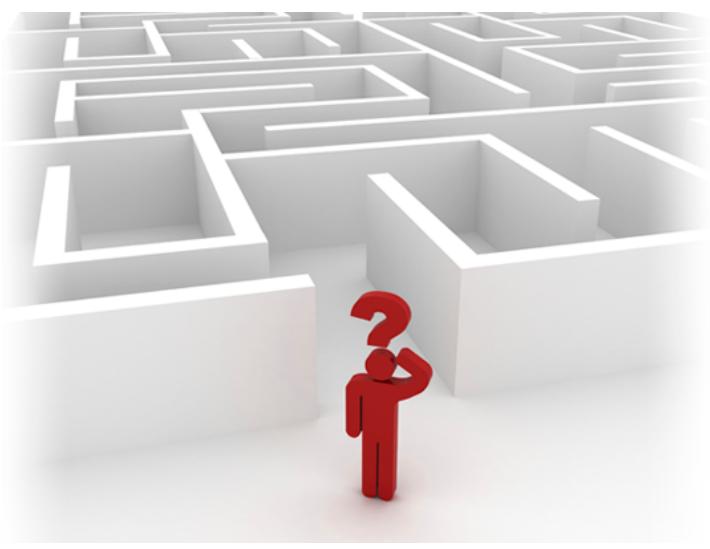
Then there exist constants c, n_0 such that

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

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

$$n \leq c \quad \forall n \geq n_0$$

Which is clearly False [contradiction].



Design and Analysis
of Algorithms I

Asymptotic Analysis

Big-Oh: Relatives (Omega & Theta)

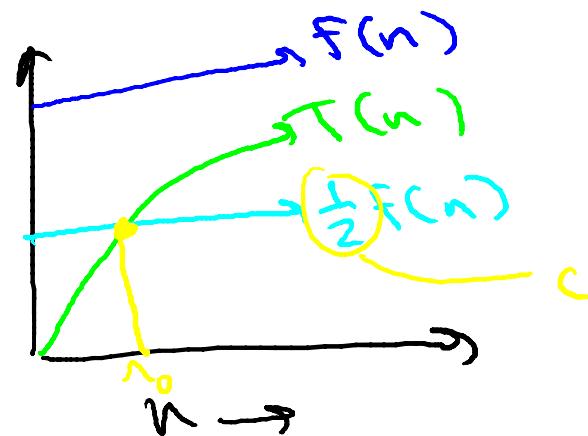
Omega Notation

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



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

Tim Roughgarden

Theta Notation

Definition : $T(n) = \theta(f(n))$ if and only if

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

Equivalent : there exist constants c_1, c_2, n_0 such that

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

$$\forall n \geq n_0$$

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

$T(n) = O(n)$.

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

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

 $T(n) = O(n^3)$. $[n_0 = 1, c = 4]$

Little-Oh Notation

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

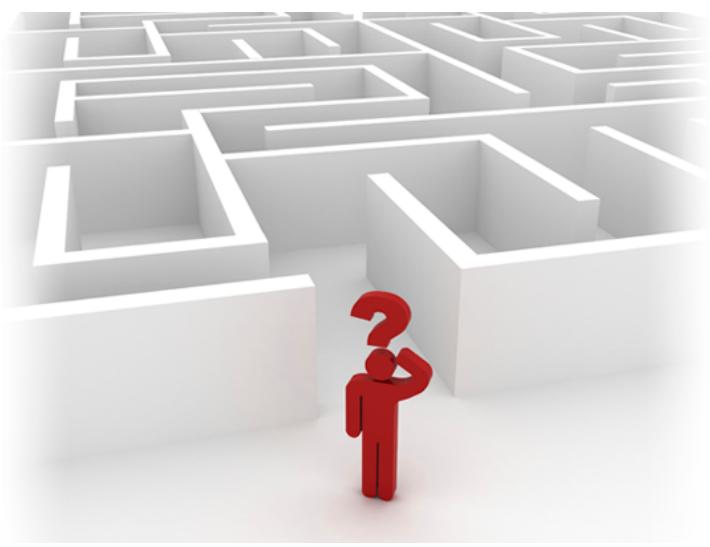
$$T(n) \leq c \cdot f(n) \quad \forall n \geq n_0$$

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

Where Does Notation Come From?

“On the basis of the issues discussed here, I propose that members of SIGACT, and editors of computer 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.”



Design and Analysis
of Algorithms I

Asymptotic Analysis

Additional Examples

Example #1

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

Proof: need to pick constants c, n_0 such that

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

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

Claim: $2^{10n} \neq O(2^n)$

Proof: 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]

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

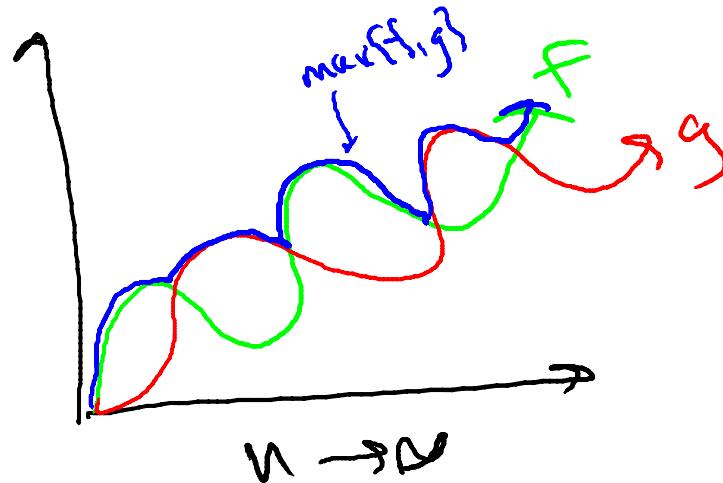
Which is certainly false.

Q.E.D

Example #3

Claim : for every pair of (positive) functions $f(n), g(n)$,

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



Tim Roughgarden

Example #3 (continued)

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

For every n , we have

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

And

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

Thus $\frac{1}{2} * (f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq f(n) + g(n) \quad \forall n \geq 1$
 $\Rightarrow \max\{f, g\} = \theta(f(n) + g(n)) \quad [\text{where } n_0 = 1, c_1 = 1/2, c_2 = 1]$

Q.E.D

Tim Roughgarden



Design and Analysis
of Algorithms I

Divide and Conquer

Closest Pair I

The Closest Pair Problem

Input : a set $P = \{p_1, \dots, p_n\}$ of n points in the plane \mathbb{R}^2 .

Notation : $d(p_i, p_j)$ = Euclidean distance

So if $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

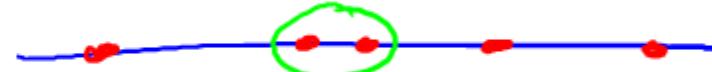
Output : a pair $p^*, q^* \in P$ of distinct points that minimize $d(p, q)$ over p, q in the set P

Initial Observations

Assumption : (for convenience) all points have distinct x-coordinates, distinct y-coordinates.

Brute-force search : takes $\theta(n^2)$ time.

1-D Version of Closest Pair :

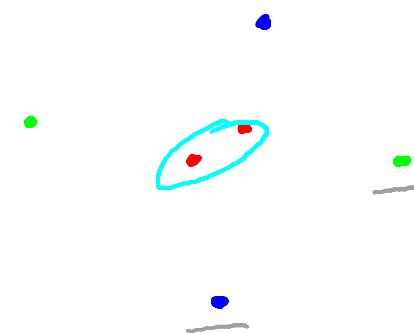


1. Sort points ($O(n \log(n))$ time)
2. Return closest pair of adjacent points ($O(n)$ time)

Goal : $O(n \log(n))$ time algorithm for 2-D version.

High-Level Approach

1. Make copies of points sorted by x-coordinate (P_x) and by y-coordinate (P_y)
[$O(n \log(n))$ time]



(but this is not enough!)

2. Use Divide+Conquer

The Divide and Conquer Paradigm

1. DIVIDE into smaller subproblems.
2. CONQUER subproblems recursively.
3. COMBINE solutions of subproblems into one for the original problem.

ClosestPair(P_x, P_y)

BASE CASE
OMITTED

1. Let $Q = \text{left half of } P$, $R = \text{right half of } P$. Form

Q_x, Q_y, R_x, R_y [takes $O(n)$ time]

2. $(p_1, q_1) = \text{ClosestPair}(Q_x, Q_y)$
3. $(p_2, q_2) = \text{ClosestPair}(R_x, R_y)$
4. $(p_3, q_3) = \text{ClosestSplitPair}(P_x, P_y)$
5. Return best of $(p_1, q_1), (p_2, q_2), (p_3, q_3)$

Suppose we can correctly implement the ClosestSplitPair subroutine in $O(n)$ time. What will be the overall running time of the Closest Pair algorithm ? (Choose the smallest upper bound that applies.)

- $O(n)$
- $O(n \log n)$
- $O(n(\log n)^2)$
- $O(n^2)$

Key Idea : only need to bother computing the closest split pair in “unlucky case” where its distance is less than $d(p_1, q_1)$ and $d(p_2, q_2)$.

Result of 1st recursive call

Result of 2nd recursive call

ClosestPair(P_x, P_y)

- Let $Q = \text{left half of } P$, $R = \text{right half of } P$. Form

BASE CASE
OMITTED

Q_x, Q_y, R_x, R_y [takes $O(n)$ time]

- $(p_1, q_1) = \text{ClosestPair}(Q_x, Q_y)$

- $(p_2, q_2) = \text{ClosestPair}(R_x, R_y)$

- Let $\delta = \min\{d(p_1, q_1), d(p_2, q_2)\}$

- $(p_3, q_3) = \text{ClosestSplitPair}(P_x, P_y, \delta)$

- Return best of $(p_1, q_1), (p_2, q_2), (p_3, q_3)$

Requirements
 1. $O(n)$ time
 2. Correct whenever closest pair of P is a split pair

WILL DESCRIBE NEXT

ClosestSplitPair(P_x, P_y, δ)

Let \bar{x} = biggest x-coordinate in left of P . (O(1) time)

Let S_y = points of P with x-coordinate in
Sorted by y-coordinate (O(n) time)

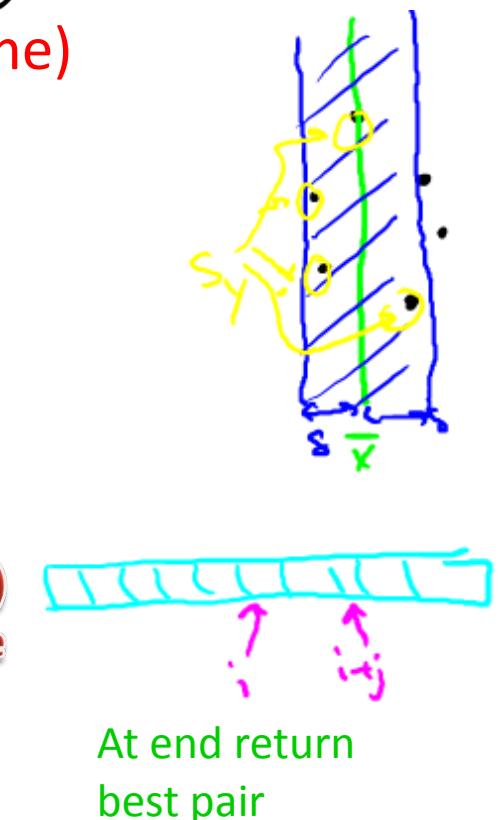
Initialize best = δ , best pair = NULL

For $i = 1$ to $|S_y| - 7$

 For $j = 1$ to 7

O(1)
time Let $p, q = i^{\text{th}}, (i+j)^{\text{th}}$ points of S_y
 If $d(p, q) < \text{best}$

 best pair = (p, q) , best = $d(p, q)$



Tim Roughgarden

Correctness Claim

Claim : Let $p \in Q, q \in R$ be a split pair with $d(p, q) < \delta$

Then: (A) p and q are members of S_y

(B) p and q are at most 7 positions apart in S_y .

$$\min\{d(p_1, q_1), d(p_2, q_2)\}$$



Corollary1 : If the closest pair of P is a split pair, then the ClosestSplitPair finds it.

Corollary2 ClosestPair is correct, and runs in $O(n \log(n))$ time.

Assuming
claim is true!



Design and Analysis
of Algorithms I

Divide and Conquer

Closest Pair II

Correctness Claim

Claim : Let $p \in Q, q \in R$ be a split pair with $d(p, q) < \delta$

Then: (A) p and q are members of S_y
(B) p and q are at most 7 positions apart in S_y .

$$\min\{d(p_1, q_1), d(p_2, q_2)\}$$

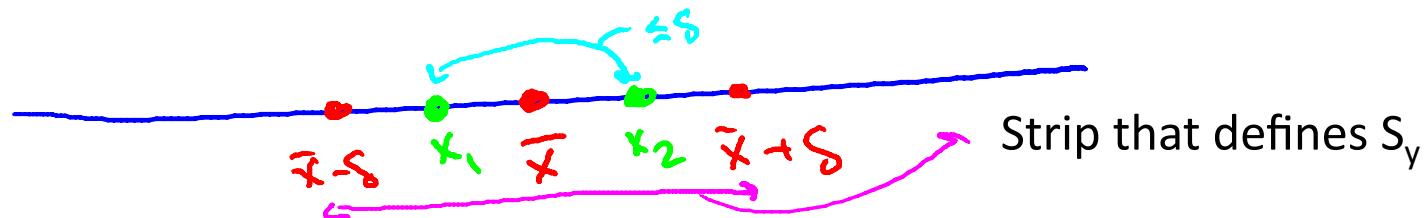


Proof of Correctness Claim (A)

Let $p = (x_1, y_1) \in Q$, $q = (x_2, y_2) \in R$, $d(p, q) \leq \delta$

Note : Since $d(p, q) \leq \delta$, $|x_1 - x_2| \leq \delta$ and $|y_1 - y_2| \leq \delta$

Proof of (A) [p and q are members of S_y i.e. $x_1, x_2 \in [\bar{x} - \delta, \bar{x} + \delta]$]



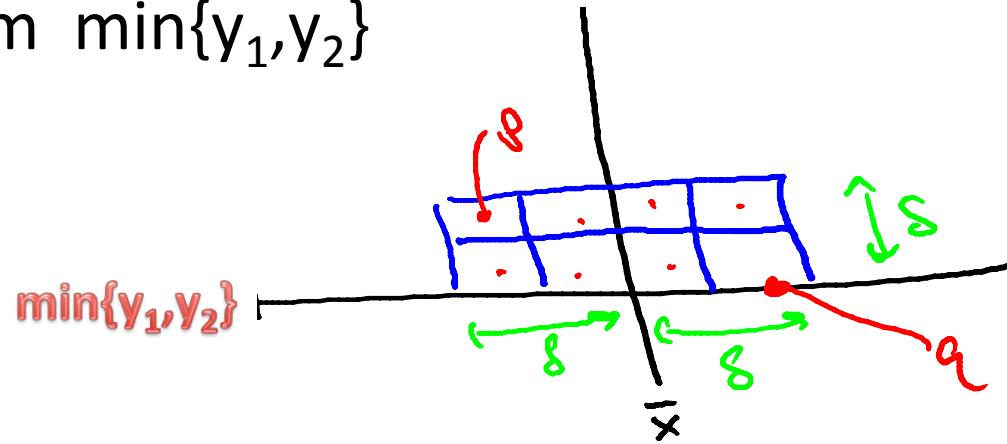
Note : $p \in Q \Rightarrow x_1 \leq \bar{x}$ and $q \in R \Rightarrow x_2 \geq \bar{x}$.

$$\Rightarrow x_1, x_2 \in [\bar{x} - \delta, \bar{x} + \delta]$$

Proof of Correctness Claim (B)

(B) : $p = (x_1, y_1)$ and $q = (x_2, y_2)$ are at most 7 positions apart in S_y

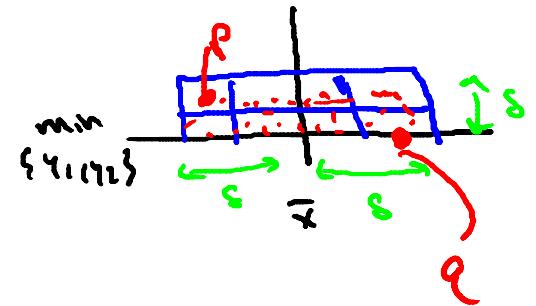
Key Picture : draw $\delta/2 \times \delta/2$ boxes with center \bar{x} and bottom $\min\{y_1, y_2\}$



Tim Roughgarden

Proof of Correctness Claim (B)

Lemma 1 : all points of S_y with y-coordinate between those of p and q, inclusive, lie in one of these 8 boxes.



Proof : First, recall y-coordinates of p,q differ by $< \delta$

Second, by definition of S_y , all have
x-coordinates between $\bar{x} - \delta$ and $\bar{x} + \delta$

Q.E.D

Proof of Correctness Claim (B)

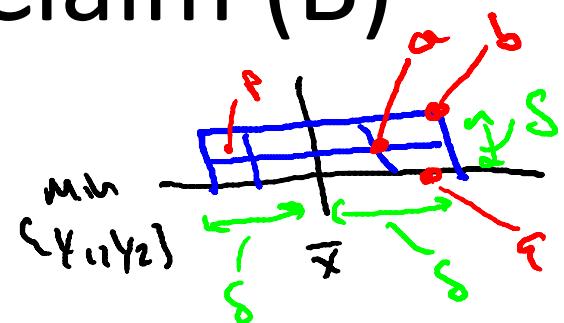
Lemma 2 : At most one point
of P in each box.

Proof : by contradiction

Suppose a,b lie in the same box. Then :

- I. a,b are either both in Q or both in R
- II. $d(a, b) \leq \frac{\delta}{2} \cdot \sqrt{2} \leq \delta$

But (i) and (ii) contradict the definition of δ
(as smallest distance between pairs of points
in Q or in R)



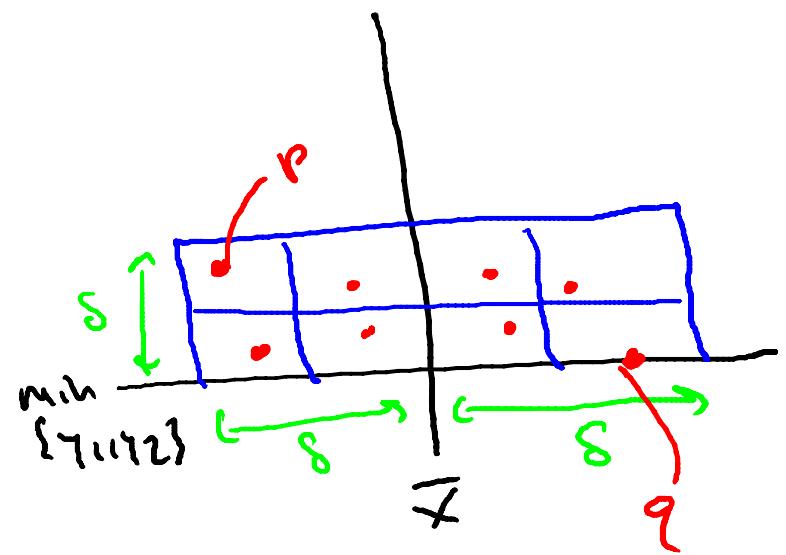
Q.E.D

Tim Roughgarden

Final Wrap-Up

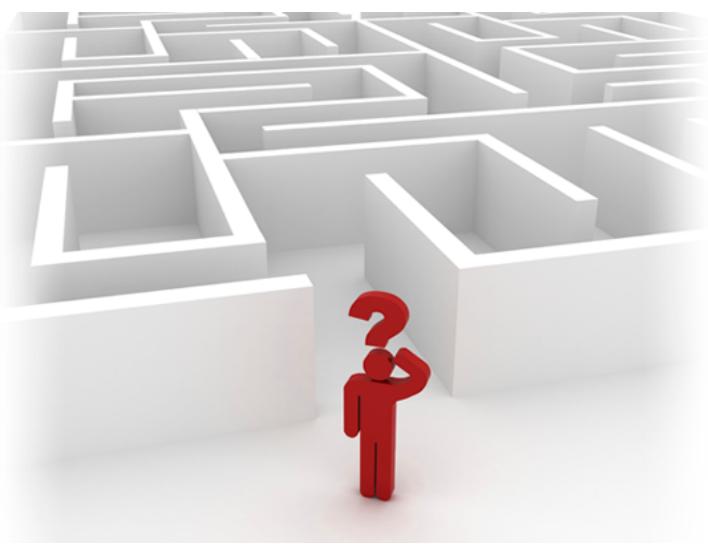
Lemmas 1 and 2 => at most 8
points in this picture
(including p and q)

=> Positions of p,q in S_y differ
by at most 7



Q.E.D

Tim Roughgarden



Design and Analysis
of Algorithms I

Divide and Conquer

Counting Inversions I

The Problem

Input : array A containing the numbers 1,2,3,..,n in some arbitrary order

Output : number of inversions = number of pairs (i,j) of array indices with $i < j$ and $A[i] > A[j]$

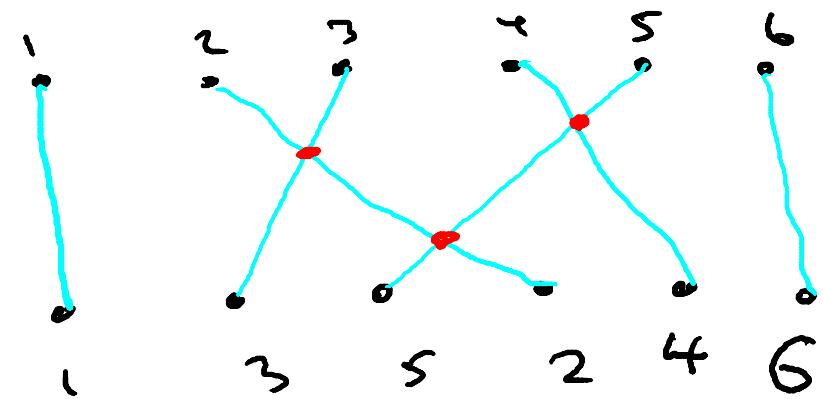
Examples and Motivation

Example

(1, 3, 5, 2, 4, 6)

Inversions :

(3,2), (5,2), (5,4)



Motivation : numerical similarity measure

between two ranked lists eg: for collaborative filtering

What is the largest-possible number of inversions that a 6-element array can have?

- 15 In general, $\binom{n}{2} = n(n - 1)/2$
- 21
- 36
- 64

High-Level Approach

Brute-force : $\theta(n^2)$ time

Can we do better ? Yes!

KEY IDEA # 1 : Divide + Conquer

Call an inversion (i,j) [with $i < j$]

Left : if $i, j < n/2$

Right : if $i, j > n/2$

Split : if $i \leq n/2 < j$

Note : can compute these
recursively

need separate subroutine for
these

High-Level Algorithm

Count (array A, length n)

 if n=1, return 0

 else

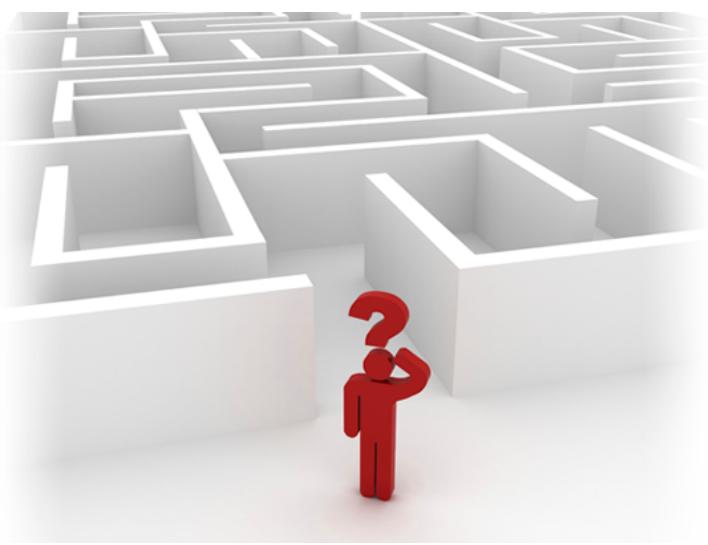
 X = Count (1st half of A, n/2)

 Y = Count (2nd half of A, n/2)

 Z = CountSplitInv(A,n) ← CURRENTLY UNIMPLEMENTED

 return x+y+z

Goal : implement CountSplitInv in linear ($O(n)$) time then
count will run in $O(n \log(n))$ time [just like Merge Sort]



Design and Analysis
of Algorithms I

Divide and Conquer

Counting Inversions II

Piggybacking on Merge Sort

KEY IDEA # 2 : have recursive calls both count inversions and sort.
[i.e. , piggy back on Merge Sort]

Motivation : Merge subroutine naturally uncovers split inversions [as we'll see]

High-Level Algorithm (revised)

Sort-and-Count (array A, length n)

if $n=1$, return 0
else

Sorted version of 1st half → (B,X) = Sort-and-Count(1st half of A, $n/2$)
Sorted version of 2nd half → (C,Y) = Sort-and-Count(2nd half of A, $n/2$)
Sorted version of A → (D,Z) = CountSplitInv(A,n) ← CURRENTLY UNIMPLEMENTED
return X+Y+Z

Goal : implement CountSplitInv in linear ($O(n)$) time
=> then Count will run in $O(n\log(n))$ time [just like Merge Sort]

Pseudocode for Merge:

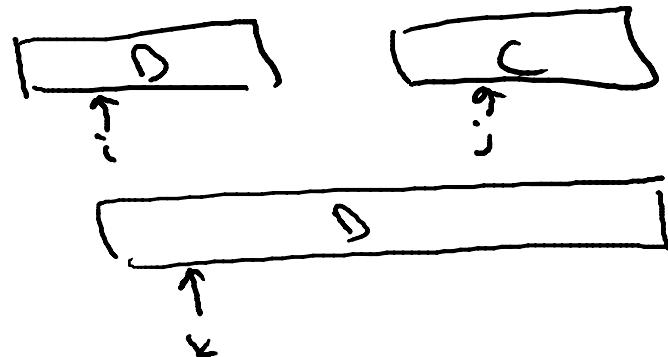
D = output [length = n]

B = 1st sorted array [n/2]

C = 2nd sorted array [n/2]

i = 1

j = 1

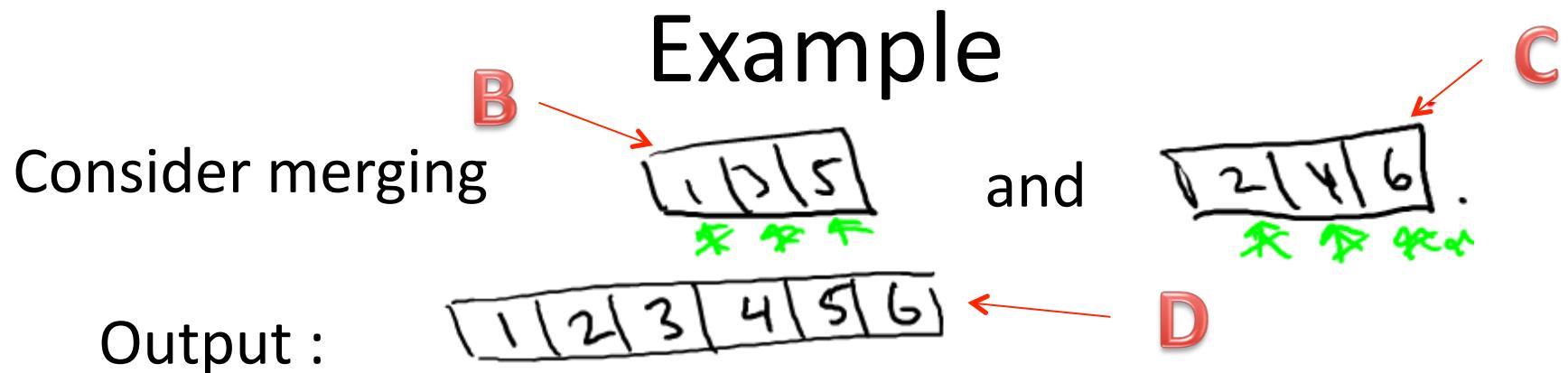


```
for k = 1 to n
    if B(i) < C(j)
        D(k) = B(i)
        i++
    else [C(j) < B(i)]
        D(k) = C(j)
        j++
end
(ignores end cases)
```

Tim Roughgarden

Suppose the input array A has no split inversions. What is the relationship between the sorted subarrays B and C?

- B has the smallest element of A, C the second-smallest, B, the third-smallest, and so on.
- All elements of B are less than all elements of C.
- All elements of B are greater than all elements of C.
- There is not enough information to answer this question.



- ⇒ When 2 copied to output, discover the split inversions (3,2) and (5,2)
- ⇒ when 4 copied to output, discover (5,4)

General Claim

Claim the split inversions involving an element y of the 2nd array C are precisely the numbers left in the 1st array B when y is copied to the output D .

Proof : Let x be an element of the 1st array B .

1. if x copied to output D before y , then $x < y$
⇒ no inversions involving x and y
2. If y copied to output D before x , then $y < x$
=> X and y are a (split) inversion.

Q.E.D

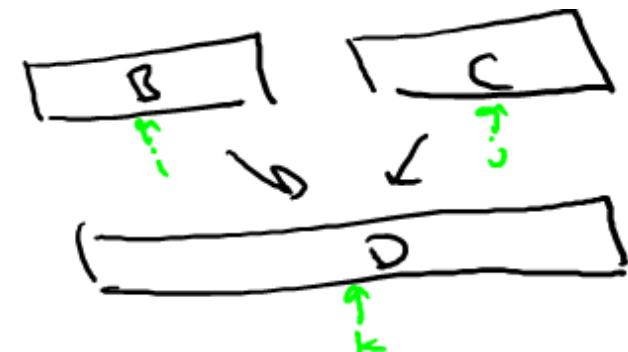
Merge_and_CountSplitInv

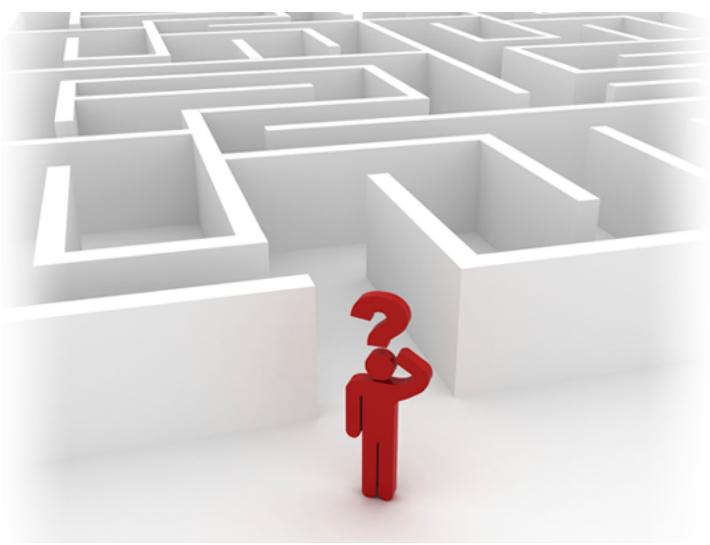
-- while merging the two sorted subarrays, keep running total of number of split inversions

-- when element of 2nd array C gets copied to output D, increment total by number of elements remaining in 1st array B

Run time of subroutine : $O(n) + O(n) = O(n)$

=> Sort_and_Count runs in $O(n \log(n))$ time [just like Merge Sort]

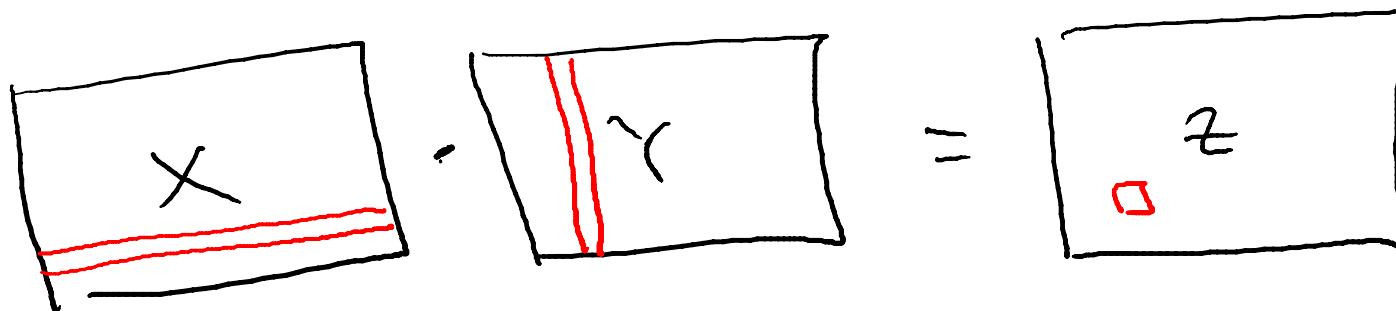




Design and Analysis
of Algorithms I

Divide and Conquer Matrix Multiplication

Matrix Multiplication



(all $n \times n$ matrices)

Where $z_{ij} = (\text{i}^{\text{th}} \text{ row of } X) \cdot (\text{j}^{\text{th}} \text{ column of } Y)$

$$= \sum_{k=1}^n X_{ik} \cdot Y_{kj}$$

Note : input size
 $= \theta(n^2)$

Example (n=2)

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

$$z_{ij} = \sum_{k=1}^n X_{ik} \cdot Y_{kj}$$

$$\theta(n)$$

What is the asymptotic running time of the straightforward iterative algorithm for matrix multiplication?

$\theta(n \log n)$

$\theta(n^2)$

 $\theta(n^3)$

$\theta(n^4)$

The Divide and Conquer Paradigm

1. DIVIDE into smaller subproblems
2. CONQUER subproblems recursively.
3. COMBINE solutions of subproblems into one for the original problem.

Applying Divide and Conquer

Idea :

Write $X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ and $Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$

[where A through H are all $n/2$ by $n/2$ matrices]

Then : (you check)

$$X \cdot Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Tim Roughgarden

Recursive Algorithm #1

Step 1 : recursively compute the 8 necessary products.

Step 2 : do the necessary additions ($\theta(n^2)$ time)

Fact : runtime is $\theta(n^3)$ [follows from the master method]

Strassen's Algorithm (1969)

Step 1 : recursively compute only 7 (cleverly chosen) products

Step 2 : do the necessary (clever) additions + subtractions
(still $\theta(n^2)$ time)

Fact : better than cubic time!

[see Master Method lecture]

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

$$Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

The Details

The Seven Products : $P_1 = A(F-H)$, $P_2 = (A+B)H$,
 $P_3 = (C+D)E$, $P_4 = D(G-E)$, $P_5 = (A+D)(E+F)$,
 $P_6 = (B-D)(G+H)$, $P_7 = (A-C)(E+F)$

Claim : $X \cdot Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 \\ P_3 + P_4 \end{pmatrix} \begin{pmatrix} P_1 + P_2 \\ P_1 + P_5 - P_3 - P_7 \end{pmatrix}$

Proof: ~~$AE + AH + DE + DH + DG - DE - AH - BH$~~ Q.E.D
 ~~$+ BG + BH - DG - DH$~~ $= AE + BG$ (remains)

Question : where did this come from? open!



Design and Analysis
of Algorithms I

Master Method

Motivation

Integer Multiplication Revisited

Motivation : potentially useful algorithmic ideas often need mathematical analysis to evaluate

Recall : grade-school multiplication algorithm uses $\theta(n^2)$ operation to multiply two n-digit numbers

A Recursive Algorithm

Recursive approach

Write $x = 10^{n/2}a + b$ $y = 10^{n/2}c + d$
[where a,b,c,d are n/2 – digit numbers]

So :

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

Algorithm#1 : recursively compute ac,ad,bc,bd,
then compute (*) in the obvious way.

A Recursive Algorithm

$T(n)$ = maximum number of operations this algorithm needs to multiply two n -digit numbers

Recurrence : express $T(n)$ in terms of running time of recursive calls.

Base Case : $T(1) \leq$ a constant.

For all $n > 1$: $T(n) \leq 4T(n/2) + O(n)$

Work done by recursive calls

Work done here

A Better Recursive Algorithm

Algorithm #2 (Gauss) : recursively compute ac , bd ,
 $(a+b)(c+d)^{(3)}$ [recall $ad+bc = (3) - (1) - (2)$]

New Recurrence :

Base Case : $T(1) \leq$ a constant

Which recurrence best describes the running time of Gauss's algorithm for integer multiplication?

$T(n) \leq 2T(n/2) + O(n^2)$

 $3T(n/2) + O(n)$

$4T(n/2) + O(n)$

$4T(n/2) + O(n^2)$

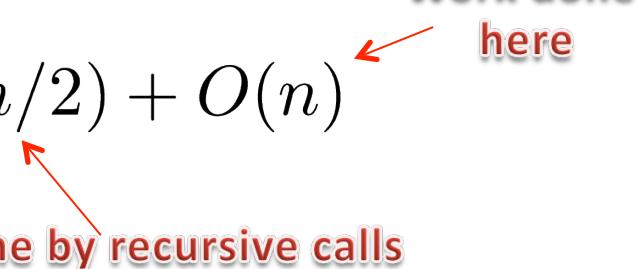
A Better Recursive Algorithm

Algorithm #2 (Gauss) : recursively compute ac , bd ,
 $(a+b)(c+d)^{(3)}$ [recall $ad+bc = (3) - (1) - (2)$]

New Recurrence :

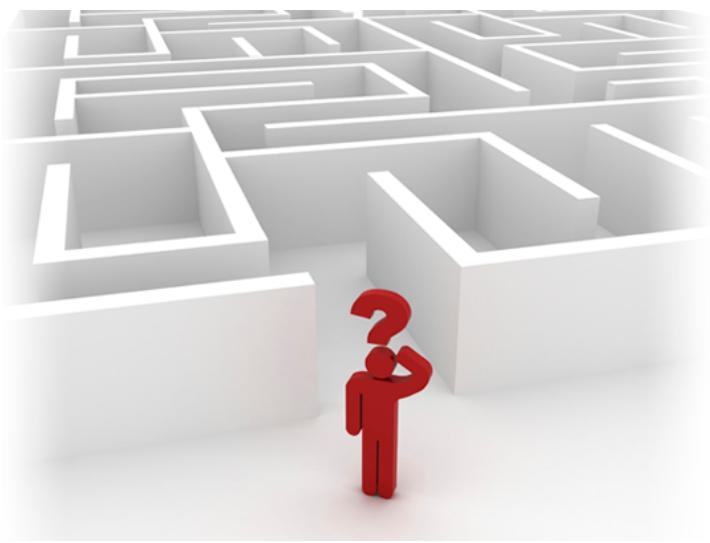
Base Case : $T(1) \leq$ a constant

For all $n > 1$: $T(n) \leq 3T(n/2) + O(n)$



Work done here

Work done by recursive calls



Design and Analysis
of Algorithms I

Master Method

The Precise Statement

The Master Method

Cool Feature : a “black box” for solving
recurrences.

Assumption : all subproblems have
equal size.

Recurrence Format

1. Base Case : $T(n) \leq$ a constant for all sufficiently small n
2. For all larger n :

$$T(n) \leq aT(n/b) + O(n^d)$$

where

a = number of recursive calls (≥ 1)

b = input size shrinkage factor (> 1)

d = exponent in running time of “combine step” (≥ 0)

[a, b, d independent of n]

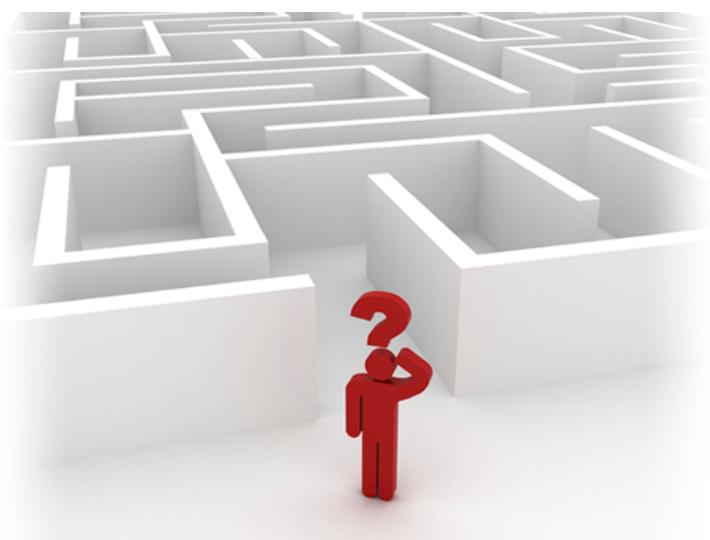
The Master Method

-

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad (\text{Case 1}) \\ O(n^d) & \text{if } a < b^d \quad (\text{Case 2}) \\ O(n^{\log_b a}) & \text{if } a > b^d \quad (\text{Case 3}) \end{cases}$$

Base doesn't matter (only changes leading constants)

Base matters



Design and Analysis
of Algorithms I

Master Method

Examples

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad (\text{Case 1}) \\ O(n^d) & \text{if } a < b^d \quad (\text{Case 2}) \\ O(n^{\log_b a}) & \text{if } a > b^d \quad (\text{Case 3}) \end{cases}$$

Example #1

Merge Sort

$$\left. \begin{array}{l} a = 2 \\ b = 2 \\ d = 1 \end{array} \right\} \quad b^d = a \Rightarrow \text{Case 1}$$

$$T(n) = O(n^d \log n) = O(n \log n)$$

Where are the respective values of a, b, d for a binary search of a sorted array, and which case of the Master Method does this correspond to?

- 1, 2, 0 [Case 1] $a = b^d \Rightarrow T(n) = O(n^d \log n) = O(\log n)$
- 1, 2, 1 [Case 2]
- 2, 2, 0 [Case 3]
- 2, 2, 1 [Case 1]

Example #3

Integer Multiplication Algorithm # 1

$$a = 4$$

$$b = 2$$

$$d = 1$$

$$b^d = 2 < a \text{ (Case 3)}$$

$$\Rightarrow T(n) = O(n^{\log_b a}) = O(n^{\log_2 4})$$

$$= O(n^2)$$

Same as grade-school
algorithm

Where are the respective values of a, b, d for Gauss's recursive integer multiplication algorithm, and which case of the Master Method does this correspond to?

- 2, 2, 1 [Case 1]
- 3, 2, 1 [Case 1]
- 3, 2, 1 [Case 2]
- 3, 2, 1 [Case 3] Better than
the grade-
school
algorithm!!!

$$a = 3, \quad b^d = 2 \quad a > b^d \quad (\text{Case 3})$$
$$\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

Example #5

Strassen's Matrix Multiplication Algorithm

$$a = 7$$

$$\left. \begin{array}{l} b = 2 \\ d = 2 \end{array} \right\} b^d = 4 < a \quad (\text{Case 3})$$

$$\Rightarrow T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

\Rightarrow beats the naïve iterative algorithm !

Tim Roughgarden

Example #6

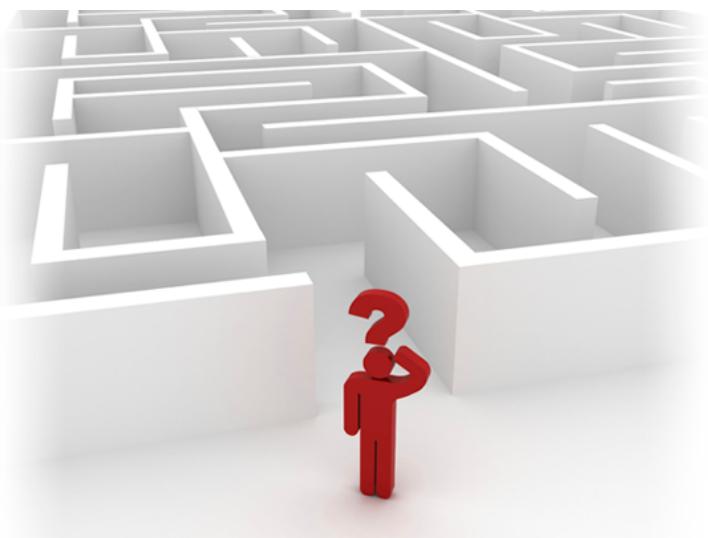
Fictitious Recurrence

$$T(n) \leq 2T(n/2) + O(n^2)$$

$$\Rightarrow a = 2$$

$$\begin{aligned} \Rightarrow b &= 2 \\ \Rightarrow d &= 2 \end{aligned} \quad \left. \right\} b^d = 4 > a \quad (\textit{Case 2})$$

$$\Rightarrow T(n) = O(n^2)$$



Design and Analysis
of Algorithms I

Master Method

Proof (Part I)

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad (\text{Case 1}) \\ O(n^d) & \text{if } a < b^d \quad (\text{Case 2}) \\ O(n^{\log_b a}) & \text{if } a > b^d \quad (\text{Case 3}) \end{cases}$$

Preamble

Assume : recurrence is

- I. $T(1) \leq c$ (For some constant c)
- II. $T(n) \leq aT(n/b) + cn^d$

And n is a power of b.

(general case is similar, but more tedious)

Idea : generalize MergeSort analysis.

(i.e., use a recursion tree)

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

of times you can divide n by b
before reaching 1

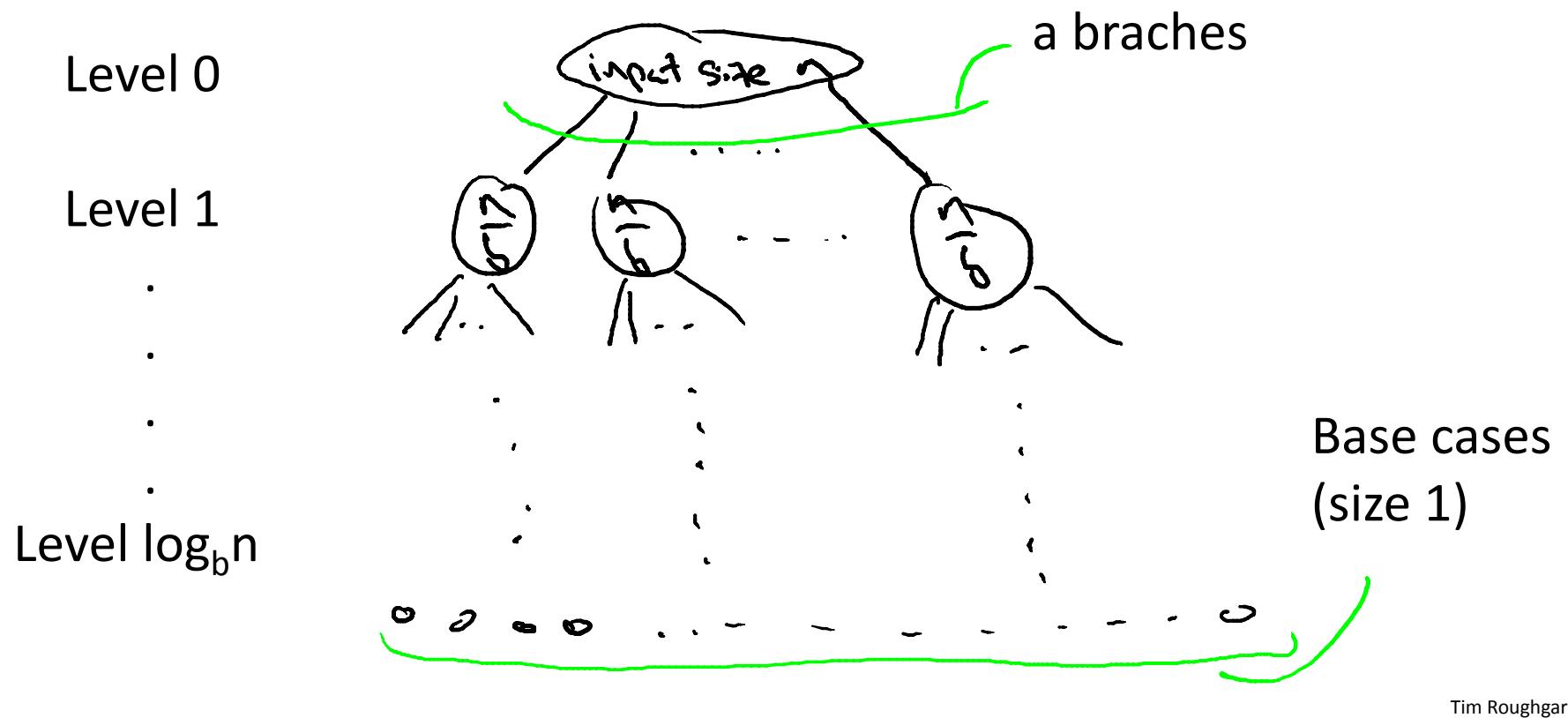
a^j and n/a^j , respectively.

a^j and n/b^j , respectively.

b^j and n/a^j , respectively.

b^j and n/b^j , respectively.

The Recursion Tree



Work at a Single Level

Total work at level j [ignoring work in recursive calls]

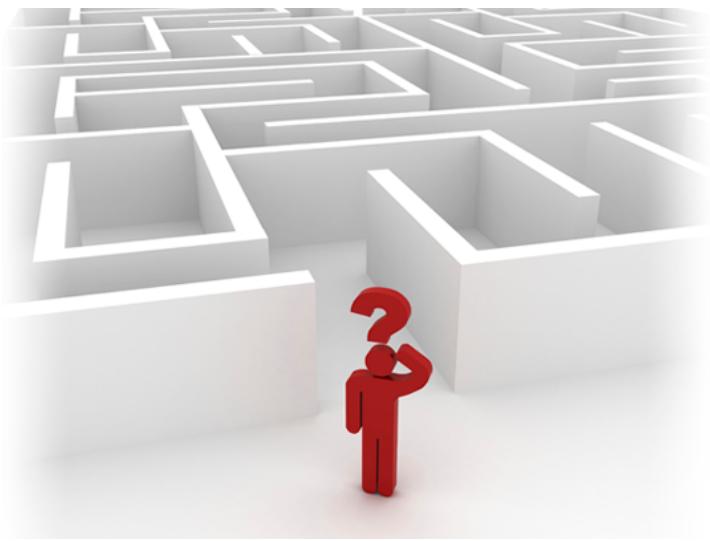
$$\leq \underbrace{a^j}_{\text{\# of level-}j \text{ subproblems}} \cdot c \cdot \underbrace{\left(\frac{n}{b^j}\right)^d}_{\text{Size of each level-}j \text{ subproblem}} = cn^d \cdot \left(\frac{a}{b^d}\right)^j$$

The diagram illustrates the formula for total work at level j . A red circle highlights the term $\left(\frac{n}{b^j}\right)^d$, which is further broken down into its components: n (blue oval) and b^j (blue bracket). An arrow points from this highlighted term to the text "Work per level- j subproblem". Another blue bracket underlines the entire term $\left(\frac{n}{b^j}\right)^d$, with an arrow pointing to the text "Size of each level- j subproblem".

Total Work

Summing over all levels $j = 0, 1, 2, \dots, \log_b n$:

$$\text{Total work} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \quad (*)$$



Design and Analysis
of Algorithms I

Master Method

Intuition for the 3 Cases

How To Think About (*)

Our upper bound on the work at level j :

$$cn^d \times \left(\frac{a}{b^d}\right)^j$$

Interpretation

a = rate of subproblem proliferation (RSP)

b^d = rate of work shrinkage (RWS)

(per subproblem)

Which of the following statements are true?
(Check all that apply.)

- If $RSP < RWS$, then the amount of work is decreasing with the recursion level j .
- If $RSP > RWS$, then the amount of work is increasing with the recursion level j .
- No conclusions can be drawn about how the amount of work varies with the recursion level j unless RSP and RWS are equal.
- If RSP and RWS are equal, then the amount of work is the same at every recursion level j .

Intuition for the 3 Cases

Upper bound for level j : $cn^d \times \left(\frac{a}{b^d}\right)^j$

1. $RSP = RWS \Rightarrow$ Same amount of work each level (like Merge Sort) [expect $O(n^d \log(n))$]
2. $RSP < RWS \Rightarrow$ less work each level \Rightarrow most work at the root [might expect $O(n^d)$]
3. $RSP > RWS \Rightarrow$ more work each level \Rightarrow most work at the leaves [might expect $O(\# \text{ leaves})$]



Design and Analysis
of Algorithms I

Master Method

Proof (Part II)

The Story So Far/Case 1

Total work: $\leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$ (*)

If $a = b^d$, then

$$(*) = cn^d (\log_b n + 1)$$

$$= O(n^d \log n)$$

[end Case 1]

$= 1$ for
all j

$= (\log_b n + 1)$

Basic Sums Fact

For $r \neq 1$, we have

$$1 + r + r^2 + r^3 + \dots + r^k = \frac{r^{k+1} - 1}{r - 1}$$

Proof : by induction (you check)

Upshot:

1. If $r < 1$ is constant, RHS is $\leq \frac{1}{1-r} = \text{a constant}$
I.e., 1st term of sum dominates
2. If $r > 1$ is constant, RHS is $\leq r^k \cdot \left(1 + \frac{1}{r-1}\right)$
I.e., last term of sum dominates

Independent of k

Case 2

$$\text{Total work: } \leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \quad (*)$$

If $a < b^d$ [RSP < RWS]

$$= O(n^d)$$

$$\sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j := r$$

<= a constant
(independent of n)
[by basic sums fact]

[total work dominated by top level]

Case 3

$$\text{Total work: } \leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \quad (*)$$

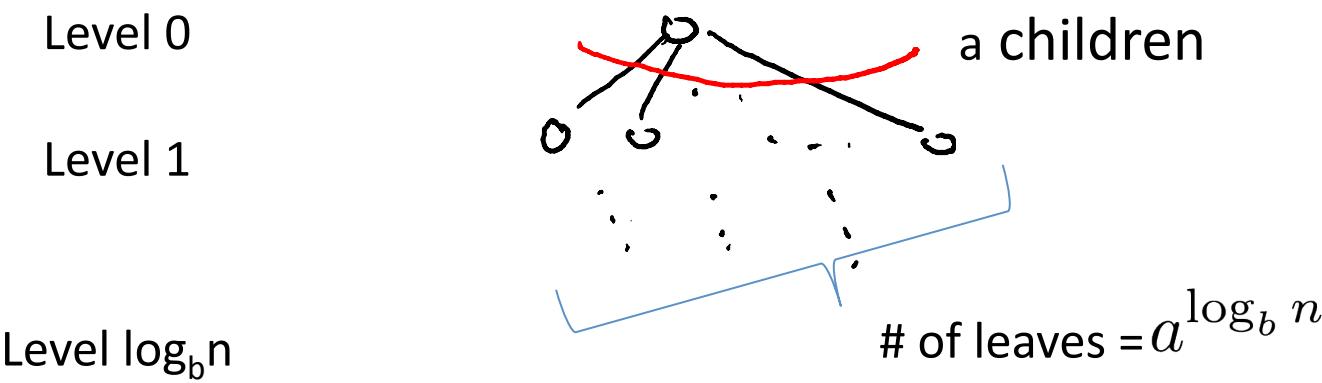
If $a > b^d$ [$RSP > RWS$]

$$(*) = O(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n})$$

≤ constant *
largest term

$$\text{Note : } b^{-d \log_b n} = (b^{\log_b n})^{-d} = n^{-d}$$

$$\text{So : } (*) = O(a^{\log_b n})$$



Which of the following quantities is equal to $a^{\log_b n}$?

- The number of levels of the recursion tree.
- The number of nodes of the recursion tree.
- The number of edges of the recursion tree.
- The number of leaves of the recursion tree.

Case 3 continued

$$\text{Total work: } \leq cn^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \quad (*)$$

$$So : (*) = O(a^{\log_b n}) = O(\# \text{ leaves})$$

Note : $a^{\log_b n} = n^{\log_b a}$

More intuitive

Simpler to apply

$$[Since (\log_b n)(\log_b a) = (\log_b a)(\log_b n)]$$

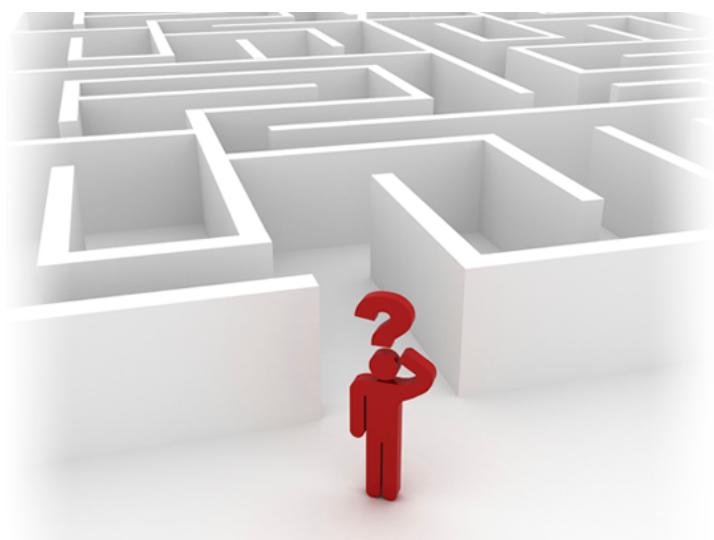
[End Case 3]

The Master Method

If $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \text{ (Case 1)} \\ O(n^d) & \text{if } a < b^d \text{ (Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (Case 3)} \end{cases}$$



Design and Analysis
of Algorithms I

QuickSort

Proof of Correctness

Induction Review

Let $P(n)$ = assertion parameterized by positive integers n .

For us : $P(n)$ is “Quick Sort correctly sorts every input array of length n ”

How to prove $P(n)$ for all $n \geq 1$ by induction :

1. [base case] directly prove that $P(1)$ holds.
2. [inductive step] for every $n \geq 2$, prove that:
If $P(k)$ holds for all $k < n$, then $P(n)$ holds as well.

INDUCTIVE
HYPOTHESIS

Correctness of QuickSort

$P(n)$ = “ QuickSort correctly sorts every input array of length n ”

Claim : $P(n)$ holds for every $n \geq 1$ [no matter how pivot is chosen]

Proof by induction :

1. [base case] every input array of length 1 is already sorted.
Quick Sort returns the input array which is correct (so $P(1)$ holds)
2. [inductive step] Fix $n \geq 2$. Fix some input array of length n .

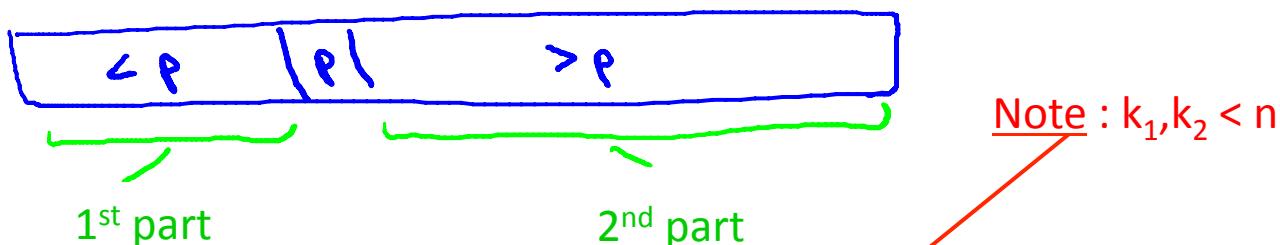
Need to show : if $P(k)$ holds for all $k < n$, then $P(n)$ holds as well.

INDUCTIVE STEP

Tim Roughgarden

Correctness of QuickSort (con'd)

Recall : QuickSort first partitions A around some pivot p.



Note : pivot winds up in the correct position.

Let k_1, k_2 = lengths of 1st, 2nd parts of partitioned array.

By inductive hypothesis : 1st, 2nd parts get sorted correctly by recursive calls. So after recursive calls, entire array correctly sorted.



QuickSort

Overview

Design and Analysis
of Algorithms I

QuickSort

- Definitely a “greatest hit” algorithm
- Prevalent in practice
- Beautiful analysis
- $O(n \log n)$ time “on average”, works in place
 - i.e., minimal extra memory needed
- See course site for optional lecture notes

The Sorting Problem

Input : array of n numbers, unsorted

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

Output : Same numbers, sorted in increasing order

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

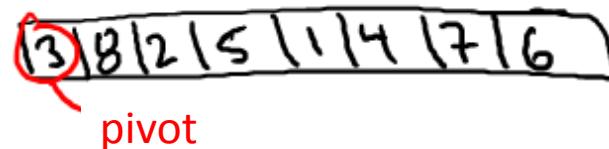
Assume : all array entries distinct.

Exercise : extend QuickSort to handle duplicate entries

Partitioning Around a Pivot

Key Idea : partition array around a pivot element.

-Pick element of array



-Rearrange array so that

- Left of pivot => less than pivot

- Right of pivot => greater than pivot



Note : puts pivot in its “rightful position”.

Two Cool Facts About Partition

1. Linear $O(n)$ time, no extra memory
[see next video]
2. Reduces problem size

QuickSort: High-Level Description

[Hoare circa 1961]

QuickSort (array A, length n)

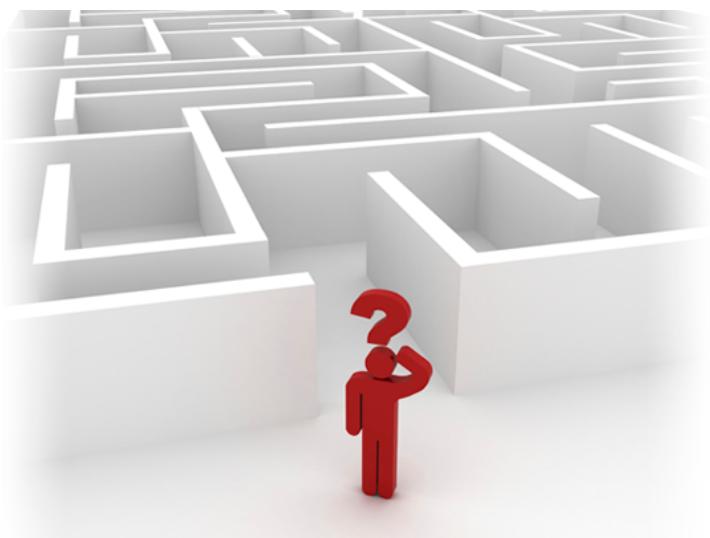
- If $n=1$ return
- $p = \text{ChoosePivot}(A, n)$
- Partition A around p
- Recursively sort 1st part
- Recursively sort 2nd part

[currently unimplemented]



Outline of QuickSort Videos

- The Partition subroutine
- Correctness proof [optional]
- Choosing a good pivot
- Randomized QuickSort
- Analysis
 - A Decomposition Principle
 - The Key Insight
 - Final Calculations



Design and Analysis
of Algorithms I

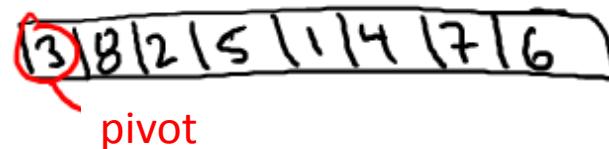
QuickSort

The Partition Subroutine

Partitioning Around a Pivot

Key Idea : partition array around a pivot element.

-Pick element of array



-Rearrange array so that

- Left of pivot => less than pivot

- Right of pivot => greater than pivot



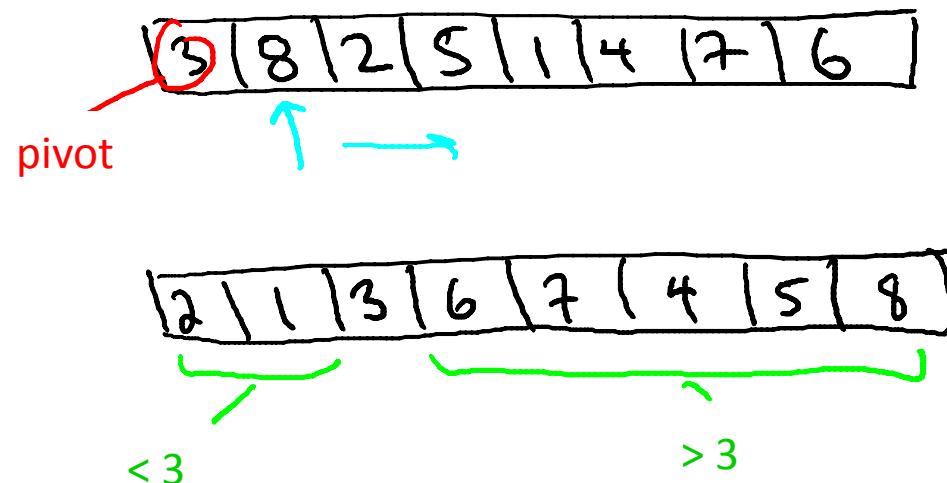
Note : puts pivot in its “rightful position”.

Two Cool Facts About Partition

1. Linear $O(n)$ time, no extra memory
[see next video]
2. Reduces problem size

The Easy Way Out

Note : Using $O(n)$ extra memory, easy to partition around pivot in $O(n)$ time.

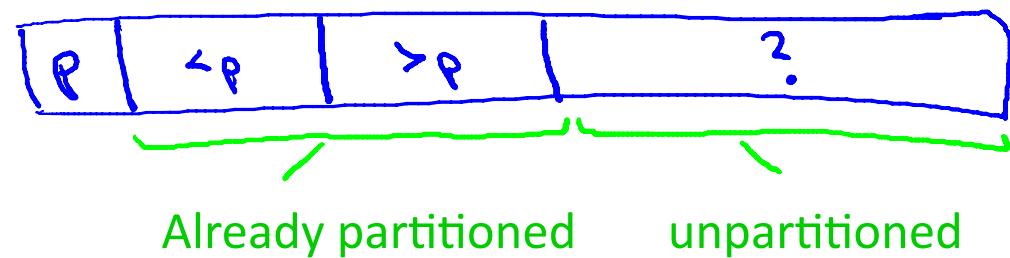


In-Place Implementation

Assume : pivot = 1st element of array

[if not, swap pivot <--> 1st element as preprocessing step]

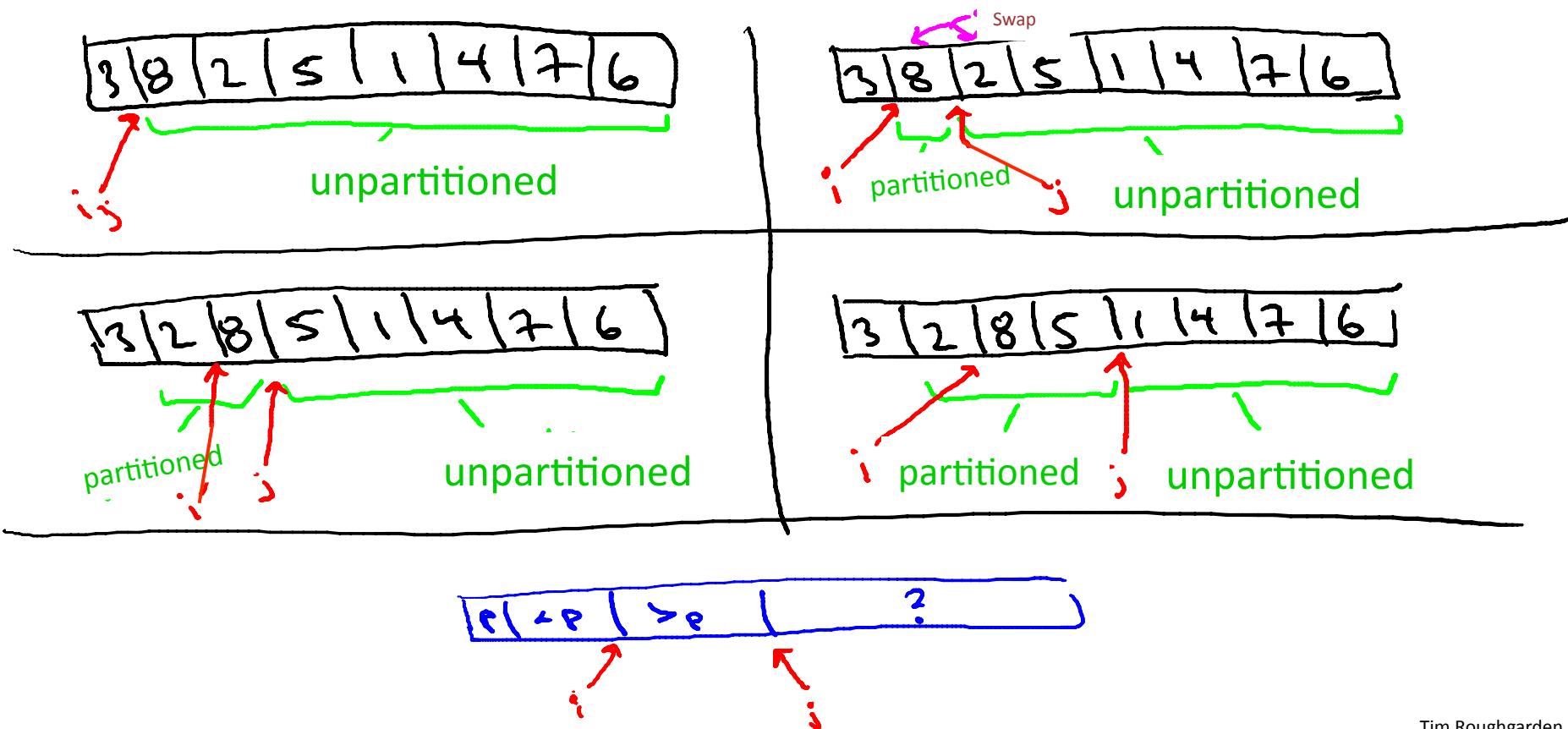
High – Level Idea :



-Single scan through array

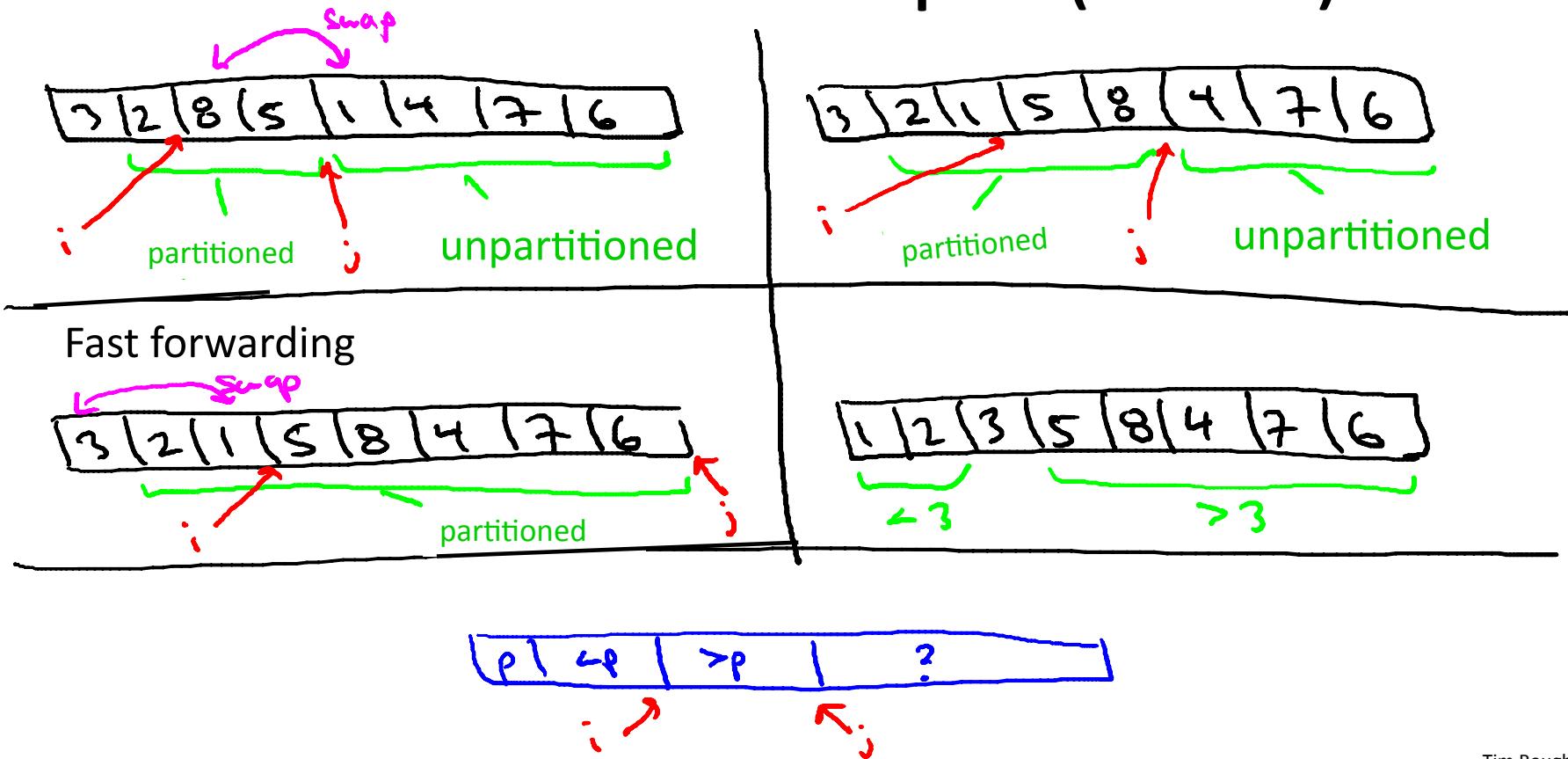
- invariant : everything looked at so far is partitioned

Partition Example



Tim Roughgarden

Partition Example (con'd)



Tim Roughgarden

Pseudocode for Partition

Partition (A, l, r)

[input corresponds to $A[l \dots r]$]

- $p := A[l]$

- $i := l+1$

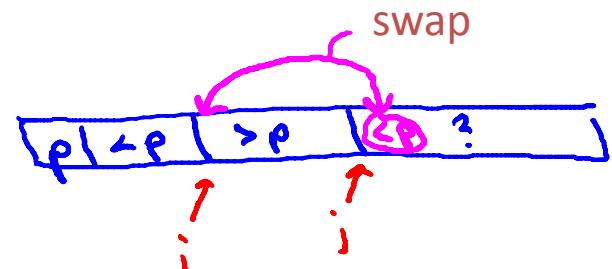
- for $j = l+1$ to r

 - if $A[j] < p$ [if $A[j] > p$, do nothing]

 - swap $A[j]$ and $A[i]$

 - $i := i+1$

- swap $A[l]$ and $A[i-1]$



Tim Roughgarden

Running Time

Running time = $O(n)$, where $n = r - l + 1$ is the length of the input (sub) array.

Reason : $O(1)$ work per array entry.

Also : clearly works in place (repeated swaps)

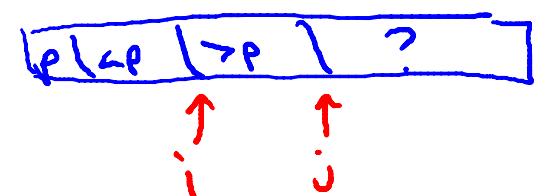
Correctness

Claim : the for loop maintains the invariants :

1. $A[i+1], \dots, A[i-1]$ are all less than the pivot
2. $A[i], \dots, A[j-1]$ are all greater than pivot.

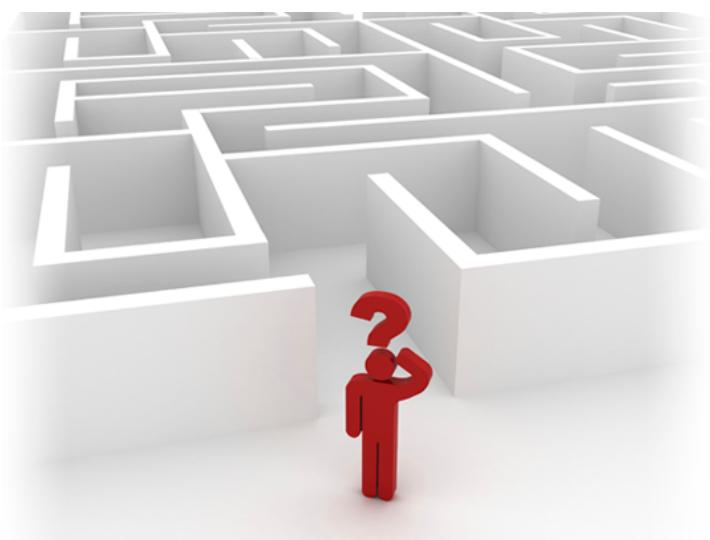
[Exercise : check this, by induction.]

Consequence : at end of for loop, have:



=> after final swap, array partitioned around pivot.

Q.E.D



Design and Analysis
of Algorithms I

QuickSort

Choosing a Good Pivot

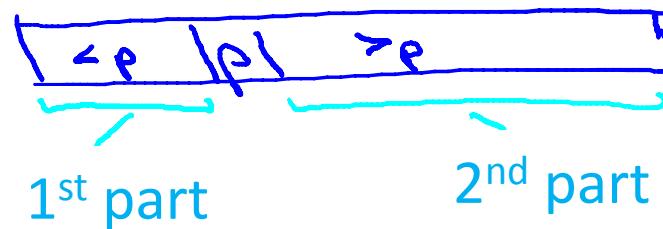
QuickSort: High-Level Description

[Hoare circa 1961]

QuickSort (array A, length n)

- If $n=1$ return
- $p = \text{ChoosePivot}(A, n)$
- Partition A around p
- Recursively sort 1st part
- Recursively sort 2nd part

[currently unimplemented]



The Importance of the Pivot

Q : running time of QuickSort ?

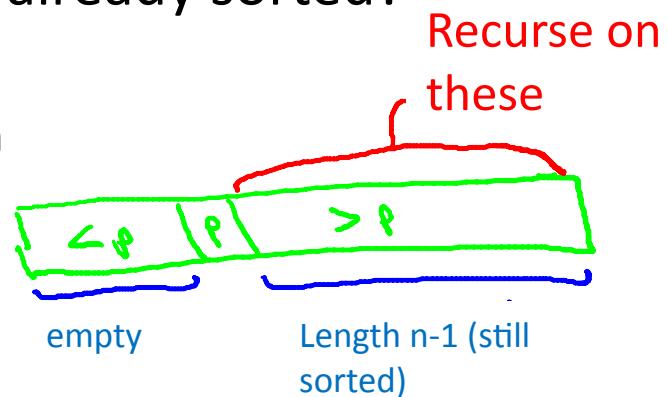
A : depends on the quality of the pivot.

Suppose we implement QuickSort so that ChoosePivot always selects the first element of the array. What is the running time of this algorithm on an input array that is already sorted?

- Not enough information to answer question
- $\theta(n)$
- $\theta(n \log n)$
- $\theta(n^2)$

1st $n/2$ terms are all at least $n/2$

Reason :



Runtime :

$$\begin{aligned} &\geq n + (n - 1) + (n - 2) + \dots + 1 \\ &= \theta(n^2) \end{aligned}$$

Suppose we run QuickSort on some input, and, magically, every recursive call chooses the median element of its subarray as its pivot. What's the running time in this case?

Not enough information to answer question

$\theta(n)$

Reason : Let $T(n) =$ running time on arrays of size n .

$\theta(n \log n)$

$\theta(n^2)$

Then : $T(n) \leq 2T(n/2) + \theta(n)$
 $\Rightarrow T(n) = \theta(n \log n)$ [like MergeSort]

Because pivot = median
choosePivot
partition

Random Pivots

Key Question : how to choose pivots ? BIG IDEA : RANDOM PIVOTS!

That is : in every recursive call, choose the pivot randomly.
(each element equally likely)

Hope : a random pivot is “pretty good” “often enough”.

Intuition : 1.) if always get a 25-75 split, good enough for $O(n \log(n))$ running time. [this is a non-trivial exercise : prove via recursion tree]
2.) half of elements give a 25-75 split or better

Q : does this really work ?

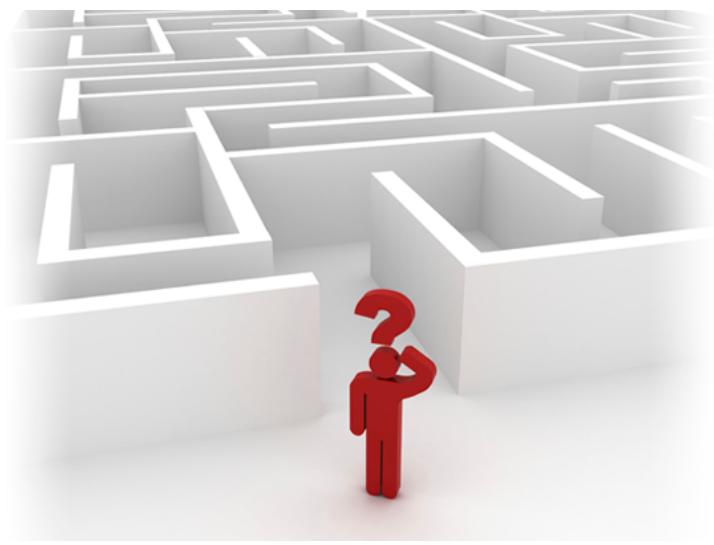
Tim Roughgarden

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$.

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm
(i.e., the pivot choices)



Design and Analysis
of Algorithms I

QuickSort

Analysis I: A Decomposition Principle

Necessary Background

Assumption: you know and remember (finite) sample spaces, random variables, expectation, linearity of expectation. For review:

- Probability Review I (video)
- Lehman-Leighton notes (free PDF)
- Wikibook on Discrete Probability

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$.

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm
(i.e., the pivot choices)

Preliminaries

Fix input array A of length n

Sample Space Ω = all possible outcomes of random choices in QuickSort (i.e., pivot sequences)

Key Random Variable : for $\sigma \in \Omega$

$C(\sigma)$ = # of comparisons between two input elements made by QuickSort (given random choices σ)

Lemma: running time of QuickSort dominated by comparisons.

Remaining goal : $E[C] = O(n \log(n))$

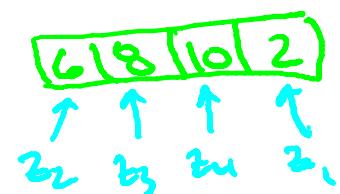
There exist constant c s.t. for all $\sigma \in \Omega$, $RT(\sigma) \leq c \cdot C(\sigma)$
(see notes)

Tim Roughgarden

Building Blocks

Note can't apply Master Method [random, unbalanced subproblems]

[A = final input array]



Notation : z_i = i^{th} smallest element of A

For $\sigma \in \Omega$, indices $i < j$

$X_{ij}(\sigma) = \# \text{ of times } z_i, z_j \text{ get compared in QuickSort with pivot sequence } \sigma$

Fix two elements of the input array. How many times can these two elements get compared with each other during the execution of QuickSort?

1

0 or 1

0, 1, or 2

Any integer between 0 and $n - 1$

Reason : two elements compared only when one is the pivot, which is excluded from future recursive calls.

Thus : each X_{ij} is an “indicator” (i.e., 0-1) random variable

A Decomposition Approach

So : $C(\sigma) = \# \text{ of comparisons between input elements}$

$X_{ij}(\sigma) = \# \text{ of comparisons between } z_i \text{ and } z_j$

Thus : $\forall \sigma, C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}(\sigma)$

By Linearity of Expectation : $E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}]$

Since $E[X_{ij}] = 0 \cdot Pr[X_{ij} = 0] + 1 \cdot Pr[X_{ij} = 1] = Pr[X_{ij} = 1]$

Thus : $E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n Pr[z_i, z_j \text{ get compared}] \quad (*)$

Next video

A General Decomposition Principle

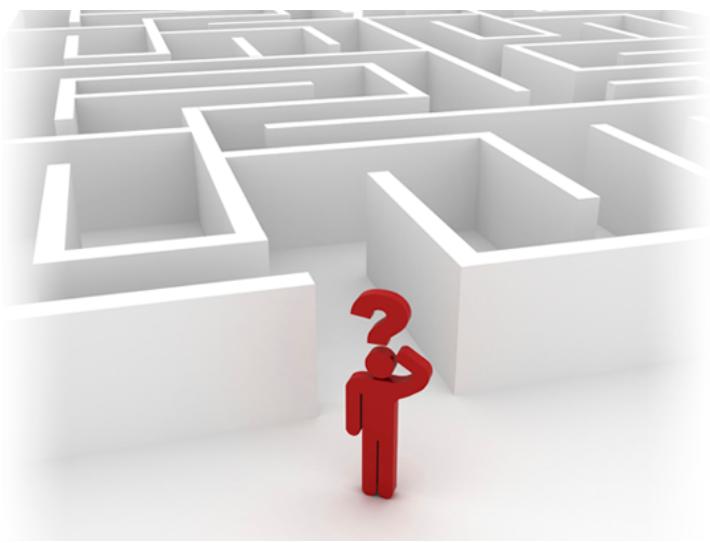
1. Identify random variable Y that you really care about
2. Express Y as sum of indicator random variables :

$$Y = \sum_{l=1}^m X_e$$

3. Apply Linearity of expectation :

$$E[Y] = \sum_{l=1}^m Pr[X_e = 1]$$

“just” need to understand these!



Design and Analysis
of Algorithms I

QuickSort

Analysis II: The Key Insight

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$.

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm
(i.e., the pivot choices)

The Story So Far

$C(\sigma)$ = # of comparisons between input elements

$X_{ij}(\sigma)$ = # of comparisons between z_i and z_j

$i^{\text{th}}, j^{\text{th}}$ smallest entries in array

Recall : $E[C] = \sum_{i=1}^{n-1} \sum_{k=i+1}^n Pr[X_{ij} = 1]$

$= Pr[z_i z_j \text{ get compared}]$

Key Claim : for all $i < j$, $Pr[z_i, z_j \text{ get compared}] = 2/(j-i+1)$

Proof of Key Claim

Fix z_i, z_j with $i < j$

Consider the set $z_i, z_{i+1}, \dots, z_{j-1}, z_j$

$$\Pr[z_i, z_j \text{ get compared}] = \frac{2}{(j-i+1)}$$

Inductively : as long as none of these are chosen as a pivot, all are passed to the same recursive call.

Consider the first among $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ that gets chosen as a pivot.

1. If z_i or z_j gets chosen first, then z_i and z_j get compared
2. If one of z_{i+1}, \dots, z_{j-1} gets chosen first then z_i and z_j are never compared [split into different recursive calls]

KEY
INSIGHT

Tim Roughgarden

Proof of Key Claim (con'd)

1. z_i or z_j gets chosen first \Rightarrow they get compared
2. one of z_{i+1}, \dots, z_{j-1} gets chosen first $\Rightarrow z_i, z_j$ never compared

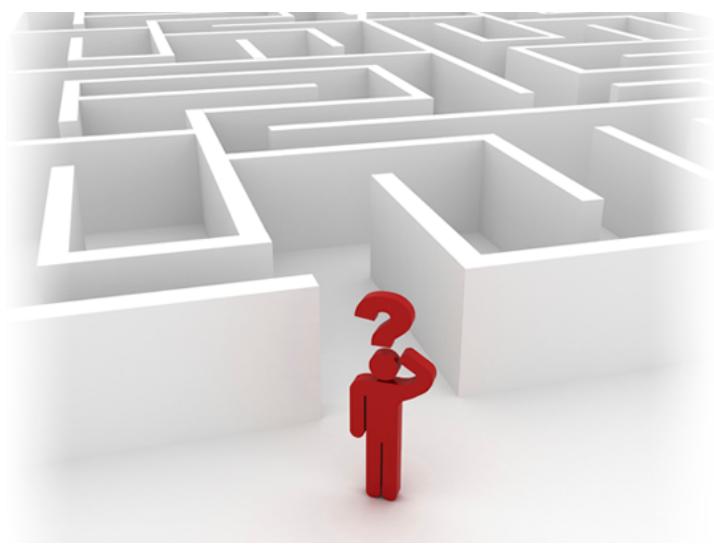
Note : Since pivots always chosen uniformly at random, each of $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ is equally likely to be the first

$$\Rightarrow \Pr[z_i, z_j \text{ get compared}] = \frac{2}{(j-i+1)}$$

Choices that lead to
 case (1) Total # of choices

$$\text{So: } E[C] = \sum_{i=1}^{n-1} \sum_{j=1}^n \frac{2}{j - i + 1}$$

[Still need to show
this is $O(n \log n)$]



Design and Analysis
of Algorithms I

QuickSort

Analysis I: A Decomposition Principle

Necessary Background

Assumption: you know and remember (finite) sample spaces, random variables, expectation, linearity of expectation. For review:

- Probability Review I (video)
- Lehman-Leighton notes (free PDF)
- Wikibook on Discrete Probability

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$.

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm
(i.e., the pivot choices)

Preliminaries

Fix input array A of length n

Sample Space Ω = all possible outcomes of random choices in QuickSort (i.e., pivot sequences)

Key Random Variable : for $\sigma \in \Omega$

$C(\sigma)$ = # of comparisons between two input elements made by QuickSort (given random choices σ)

Lemma: running time of QuickSort dominated by comparisons.

Remaining goal : $E[C] = O(n \log(n))$

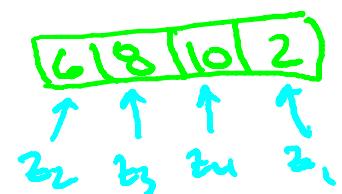
There exist constant c s.t. for all $\sigma \in \Omega$, $RT(\sigma) \leq c \cdot C(\sigma)$
(see notes)

Tim Roughgarden

Building Blocks

Note can't apply Master Method [random, unbalanced subproblems]

[A = final input array]



Notation : z_i = i^{th} smallest element of A

For $\sigma \in \Omega$, indices $i < j$

$X_{ij}(\sigma)$ = # of times z_i, z_j get compared in
QuickSort with pivot sequence σ

Fix two elements of the input array. How many times can these two elements get compared with each other during the execution of QuickSort?

1

0 or 1

0, 1, or 2

Any integer between 0 and $n - 1$

Reason : two elements compared only when one is the pivot, which is excluded from future recursive calls.

Thus : each X_{ij} is an “indicator” (i.e., 0-1) random variable

A Decomposition Approach

So : $C(\sigma)$ = # of comparisons between input elements

$X_{ij}(\sigma)$ = # of comparisons between z_i and z_j

Thus : $\forall \sigma, C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}(\sigma)$

By Linearity of Expectation : $E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}]$

Since $E[X_{ij}] = 0 \cdot Pr[X_{ij} = 0] + 1 \cdot Pr[X_{ij} = 1] = Pr[X_{ij} = 1]$

Thus : $E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n Pr[z_i, z_j \text{ get compared}] \quad (*)$

Next video

A General Decomposition Principle

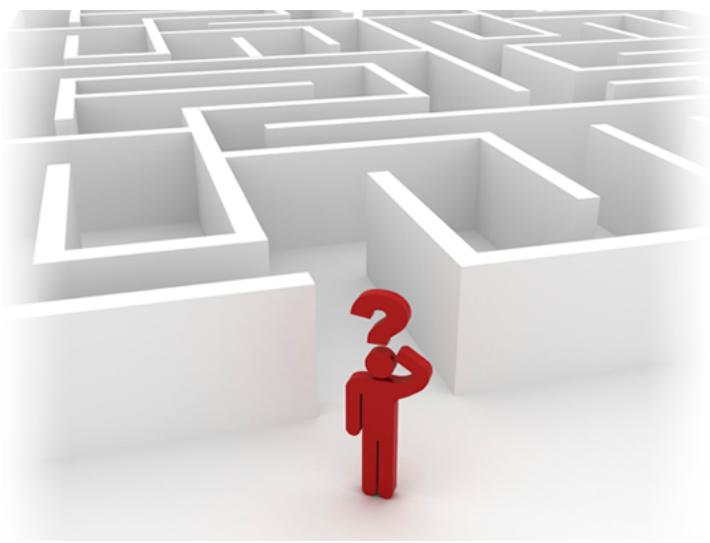
1. Identify random variable Y that you really care about
2. Express Y as sum of indicator random variables :

$$Y = \sum_{l=1}^m X_e$$

3. Apply Linearity of expectation :

$$E[Y] = \sum_{l=1}^m Pr[X_e = 1]$$

“just” need to understand these!



Design and Analysis
of Algorithms I

QuickSort

Analysis II: The Key Insight

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$.

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm
(i.e., the pivot choices)

The Story So Far

$C(\sigma)$ = # of comparisons between input elements

$X_{ij}(\sigma)$ = # of comparisons between z_i and z_j

$i^{\text{th}}, j^{\text{th}}$ smallest entries in array

Recall : $E[C] = \sum_{i=1}^{n-1} \sum_{k=i+1}^n Pr[X_{ij} = 1]$

$= Pr[z_i z_j \text{ get compared}]$

Key Claim : for all $i < j$, $Pr[z_i, z_j \text{ get compared}] = 2/(j-i+1)$

Proof of Key Claim

Fix z_i, z_j with $i < j$

Consider the set $z_i, z_{i+1}, \dots, z_{j-1}, z_j$

$$\Pr[z_i, z_j \text{ get compared}] = \frac{2}{(j-i+1)}$$

Inductively : as long as none of these are chosen as a pivot, all are passed to the same recursive call.

Consider the first among $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ that gets chosen as a pivot.

1. If z_i or z_j gets chosen first, then z_i and z_j get compared
2. If one of z_{i+1}, \dots, z_{j-1} gets chosen first then z_i and z_j are never compared [split into different recursive calls]

KEY
INSIGHT

Tim Roughgarden

Proof of Key Claim (con'd)

1. z_i or z_j gets chosen first \Rightarrow they get compared
2. one of z_{i+1}, \dots, z_{j-1} gets chosen first $\Rightarrow z_i, z_j$ never compared

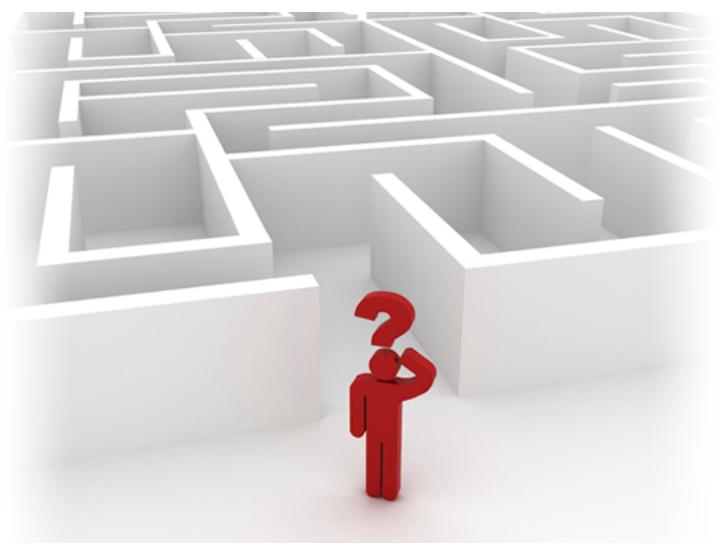
Note : Since pivots always chosen uniformly at random, each of $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ is equally likely to be the first

$$\Rightarrow \Pr[z_i, z_j \text{ get compared}] = \frac{2}{(j-i+1)}$$

Choices that lead to
 case (1) Total # of choices

So : $E[C] = \sum_{i=1}^{n-1} \sum_{j=1}^n \frac{2}{j - i + 1}$

[Still need to show
this is $O(n \log n)$]



Design and Analysis
of Algorithms I

QuickSort

Analysis III: Final Calculations

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm (i.e., pivot choices)

The Story So Far

$$E[C] = 2 \sum_{i=1}^{n-1} \sum_{j=1}^n \frac{1}{j - i + 1}$$

<= n choices
for i
 $\theta(n^2)$ terms
How big can this be ?

(*)

Note : for each fixed i , the inner sum is

$$\sum_{j=i+1}^n \frac{1}{j - i + 1} = 1/2 + 1/3 + \dots$$

$$So \quad E[C] \leq 2 \cdot n \cdot \sum_{k=2}^n \frac{1}{k}$$

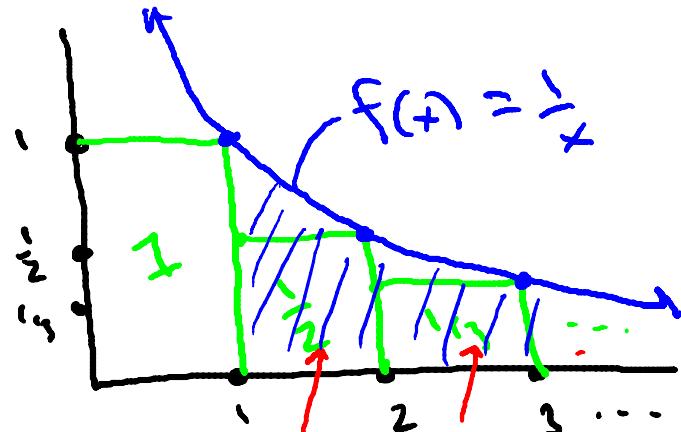
Claim : this is $<= \ln(n)$

Completing the Proof

$$E[C] \leq 2 \cdot n \cdot \sum_{k=2}^n \frac{1}{k}$$

Claim $\sum_{k=2}^n \frac{1}{k} \leq \ln n$

Proof of Claim



So :
 $E[C] \leq 2n \ln n$
Q.E.D.

$$So \quad \sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx$$

$$= \ln x \Big|_1^n$$

$$= \ln n - \ln 1$$

$$= \ln n \quad \text{Q.E.D. (CLAIM)}$$



Probability Review

Part I

Design and Analysis
of Algorithms I

Topics Covered

- Sample spaces
- Events
- Random variables
- Expectation
- Linearity of Expectation

See also:

- Lehman-Leighton notes (free PDF)
- Wikibook on Discrete Probability

Concept #1 – Sample Spaces

Sample Space Ω : “all possible outcomes”

[in algorithms, Ω is usually finite]

Also : each outcome $i \in \Omega$ has a probability $p(i) \geq 0$

Constraint : $\sum_{i \in \Omega} p(i) = 1$

Example #1 : Rolling 2 dice. $\Omega = \{(1,1), (2,1), (3,1), \dots, (5,6), (6,6)\}$

Example #2 : Choosing a random pivot in outer QuickSort call.

$\Omega = \{1, 2, 3, \dots, n\}$ (index of pivot) and $p(i) = 1/n$ for all $i \in \Omega$

Concept #2 – Events

An event is a subset $S \subseteq \Omega$

The probability of an event S is $\sum_{i \in S} p(i)$

Consider the event (i.e., the subset of outcomes for which) “the sum of the two dice is 7”. What is the probability of this event?

$$S = \{(1,6), (2,5), (3,4), (4,3), (5,2), (6,1)\}$$

1/36

1/12

$$\Pr[S] = 6/36 = 1/6$$

1/6

1/2

Consider the event (i.e., the subset of outcomes for which) “the chosen pivot gives a 25-75 split of better”. What is the probability of this event?

$1/n$

$S = \{(n/4+1)^{\text{th}} \text{ smallest element}, \dots, (3n/4)^{\text{th}} \text{ smallest element}\}$

$1/4$

$\Pr[S] = (n/2)/n = 1/2$

$1/2$

$3/4$

Concept #2 – Events

An event is a subset

The probability of an event S is

Ex#1 : sum of dice = 7. $S = \{(1,1), (2,1), (3,1), \dots, (5,6), (6,6)\}$

$$\Pr[S] = 6/36 = 1/6$$

Ex#2 : pivot gives 25-75 split or better.

$S = \{(\lfloor n/4 + 1 \rfloor)^{\text{th}} \text{ smallest element}, \dots, (\lfloor 3n/4 \rfloor)^{\text{th}} \text{ smallest element}\}$

$$\Pr[S] = (n/2)/n = 1/2$$

Concept #3 - Random Variables

A Random Variable X is a real-valued function

$$X : \Omega \rightarrow \mathbb{R}$$

Ex#1 : Sum of the two dice

Ex#2 : Size of subarray passed to 1st recursive call.

Concept #4 - Expectation

Let $X : \Omega \rightarrow \mathbb{R}$ be a random variable.

The expectation $E[X]$ of X = average value of X

$$= \sum_{i \in \Omega} X(i) \cdot p(i)$$

What is the expectation of the sum of two dice?

6.5

7

7.5

8

Which of the following is closest to the expectation of the size of the subarray passed to the first recursive call in QuickSort?

Let X = subarray size

$n/4$

$n/3$

$n/2$

$3n/4$

$$\begin{aligned} \text{Then } E[X] &= (1/n)*0 + (1/n)*2 + \dots + (1/n)*(n-1) \\ &= (n-1)/2 \end{aligned}$$

Concept #4 - Expectation

Let $X : \Omega \rightarrow \mathbb{R}$ be a random variable.

The expectation $E[X]$ of X = average value of X

$$= \sum_{i \in \Omega} X(i) \cdot p(i)$$

Ex#1 : Sum of the two dice, $E[X] = 7$

Ex#2 : Size of subarray passed to 1st recursive call.

$$E[X] = (n-1)/2$$

Concept #5 – Linearity of Expectation

Claim [LIN EXP] : Let X_1, \dots, X_n be random variables defined on Ω . Then :

$$E\left[\sum_{j=1}^n X_j\right] = \sum_{j=1}^n E[X_j]$$

Ex#1 : if X_1, X_2 = the two dice, then

$$E[X_j] = (1/6)(1+2+3+4+5+6) = 3.5$$

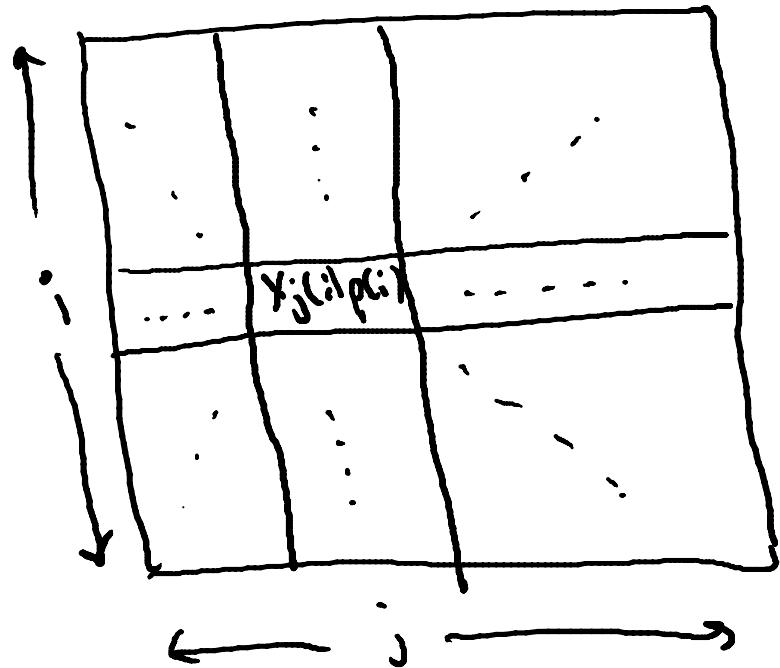
CRUCIALLY:
HOLDS EVEN WHEN
 X_j 's ARE NOT
INDEPENDENT!
[WOULD FAIL IF
REPLACE SUMS WITH
PRODUCTS]

By LIN EXP : $E[X_1 + X_2] = E[X_1] + E[X_2] = 3.5 + 3.5 = 7$

Linearity of Expectation (Proof)

$$\begin{aligned}\sum_{j=1}^n E[X_j] &= \sum_{j=1}^n \sum_{i \in \Omega} X_j(i)p(i) \\&= \sum_{i \in \Omega} \sum_{j=1}^n X_j(i)p(i) \\&= \sum_{i \in \Omega} p(i) \sum_{j=1}^n X_j(i) \\&= E\left[\sum_{j=1}^n X_j\right]\end{aligned}$$

Q.E.D.



Example: Load Balancing

Problem : need to assign n processes to n servers.

Proposed Solution : assign each process to a random server

Question : what is the expected number of processes assigned to a server ?

Load Balancing Solution

Sample Space Ω = all n^n assignments of processes to servers, each equally likely.

Let Y = total number of processes assigned to the first server.

Goal : compute $E[Y]$

Let $X_j = \begin{cases} 1 & \text{if } j\text{th process assigned to first server} \\ 0 & \text{otherwise} \end{cases}$

“indicator random variable”

$$\text{Note } Y = \sum_{j=1}^n X_j$$

Load Balancing Solution (con'd)

We have

$$\begin{aligned}
 E[Y] &= E\left[\sum_{j=1}^n X_j\right] \\
 &= \sum_{j=1}^n E[X_j] \\
 &= \sum_{j=1}^n (Pr[X_j = 0] \cdot 0 + Pr[X_j = 1] \cdot 1) \\
 &= \sum_{j=1}^n \frac{1}{n} = 1
 \end{aligned}$$

0
 $\underbrace{\Pr[X_j = 1]}$
 $\equiv 1/n$ (servers chosen
uniformly at random)



Design and Analysis
of Algorithms I

Probability Review

Part II

Topics Covered

- Conditional probability
- Independence of events and random variables

See also:

- Lehman-Leighton notes (free PDF)
- Wikibook on Discrete Probability

Concept #1 – Sample Spaces

Sample Space Ω : “all possible outcomes”

[in algorithms, Ω is usually finite]

Also : each outcome $i \in \Omega$ has a probability $p(i) \geq 0$

Constraint : $\sum_{i \in \Omega} p(i) = 1$

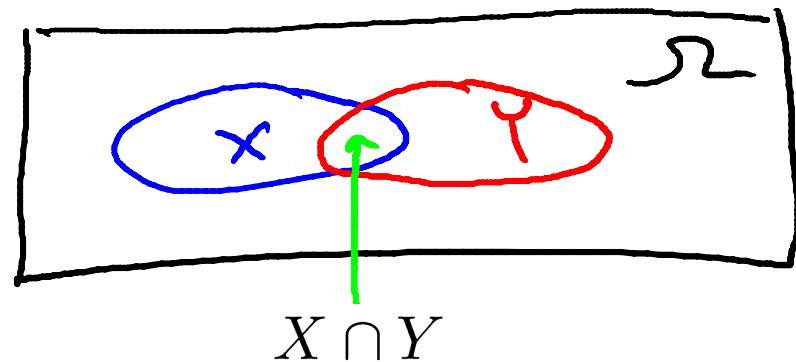
An event is a subset $S \subseteq \Omega$

The probability of an event S is $\sum_{i \in S} p(i)$

Tim Roughgarden

Concept #6 – Conditional Probability

Let $X, Y \subseteq \Omega$ be events.



Then $Pr[X|Y] = \frac{Pr[X \cap Y]}{Pr[Y]}$
("X given Y")

Suppose you roll two fair dice. What is the probability that at least one die is a 1, given that the sum of the two dice is 7?

- $1/36$
- $1/6$
- $1/3$
- $1/2$

$X = \text{at least one die is a } 1$

$Y = \text{sum of two dice} = 7$

$$= \{(1,6), (2,5), (3,4), (4,3), (5,2), (6,1)\}$$

$$\Rightarrow X \cap Y = \{(1, 6), (6, 1)\}$$

$$Pr[X|Y] = \frac{Pr[X \cap Y]}{Pr[Y]} = \frac{(2/36)}{(6/36)} = \frac{1}{3}$$

Concept #7 – Independence (of Events)

Definition : Events $X, Y \subseteq \Omega$ are independent if (and only if) $Pr[X \cap Y] = Pr[X] \cdot Pr[Y]$

You check : this holds if and only if $Pr[X | Y] = Pr[X]$
 $\iff Pr[Y | X] = Pr[Y]$

WARNING : can be a very subtle concept.
(intuition is often incorrect!)

Independence (of Random Variables)

Definition : random variables A, B (both defined on Ω)
 are independent if and only if the events $\Pr[A=1]$, $\Pr[B=b]$ are
 independent for all a,b. [$\Leftrightarrow \Pr[A = a \text{ and } B = b] = \Pr[A=a] * \Pr[B=b]$]

Claim : if A,B are independent, then $E[AB] = E[A]*E[B]$

Proof :

$$\begin{aligned}
 E[AB] &= \sum_{a,b} (a \cdot b) \cdot \Pr[A = a \text{ and } B = b] \\
 &= \sum_{a,b} (a \cdot b) \cdot \Pr[A = a] \cdot \Pr[B = b] \quad (\text{Since A,B independent}) \\
 &= (\sum_a a \cdot \Pr[A = a]) (\sum_b b \cdot \Pr[B = b])
 \end{aligned}$$

Q.E.D.

Example

Let $X_1, X_2 \in \{0, 1\}$ be random, and $X_3 = X_1 \oplus \overset{\text{XOR}}{X_2}$

formally : $\Omega = \{000, 101, 011, 110\}$, each equally likely.

Claim : X_1 and X_3 are independent random variables (you check)

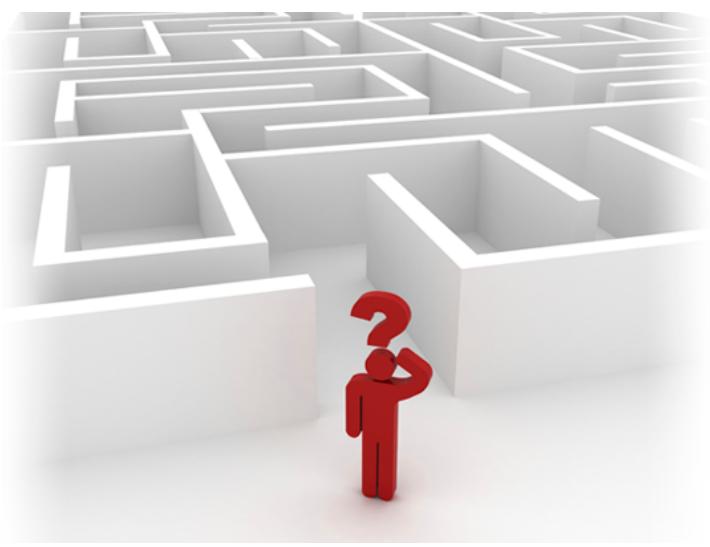
Claim : $X_1 X_3$ and X_2 are not independent random variables.

Proof : suffices to show that

$$E[X_1 X_2 X_3] \neq E[X_1 X_3] E[X_2]$$

$\stackrel{=0}{\Rightarrow}$ $\stackrel{=1/2}{\Rightarrow}$ Since X_1 and X_3 independent

$$= E[X_1] E[X_3] = 1/4$$



Design and Analysis
of Algorithms I

Linear-Time Selection

Deterministic
Selection (Algorithm)

The Problem

Input : array A with n **distinct** numbers and a number

For simplicity

Output : i^{th} order statistic (i.e., i^{th} smallest element of input array)

Example : median.

($i = (n+1)/2$ for n odd,
 $i = n/2$ for n even)

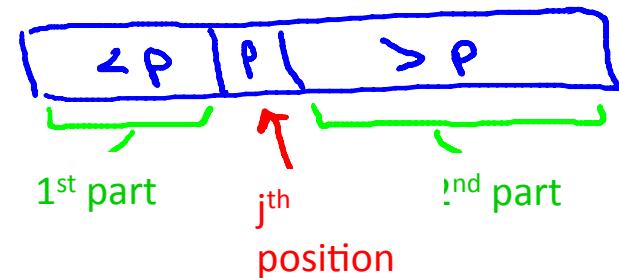


3rd order statistic

Randomized Selection

Rselect (array A, length n, order statistic i)

- 0) if $n = 1$ return $A[1]$
- 1) Choose pivot p from A uniformly at random
- 2) Partition A around p
let $j = \text{new index of } p$
- 3) If $j = i$, return p
- 4) If $j > i$, return Rselect(1^{st} part of A , $j-1$, i)
- 5) [if $j < i$] return Rselect (2^{nd} part of A , $n-j$, $i-j$)



Guaranteeing a Good Pivot

Recall : “best” pivot = the median ! (seems circular!)

Goal : find pivot guaranteed to be pretty good.

Key Idea : use “median of medians”!

A Deterministic ChoosePivot

ChoosePivot(A,n)

- logically break A into $n/5$ groups of size 5 each
- sort each group (e.g., using Merge Sort)
- copy $n/5$ medians (i.e., middle element of each sorted group)
into new array C
- recursively compute median of C (!)
- return this as pivot

The DSelect Algorithm

DSelect(array A, length n, order statistic i)

1. Break A into groups of 5, sort each group
2. C = the $n/5$ “middle elements”
3. $p = \text{DSelect}(C, n/5, n/10)$ [recursively computes median of C]
4. Partition A around p
5. If $j = i$ return p
6. If $j < i$ return DSelect(1^{st} part of A, $j-1$, i)
7. [else if $j > i$] return DSelect(2^{nd} part of A, $n-j$, $i-j$)

ChoosePivot

Same as
before

How many recursive calls does DSelect make?

0

1

2

3

Running Time of DSelect

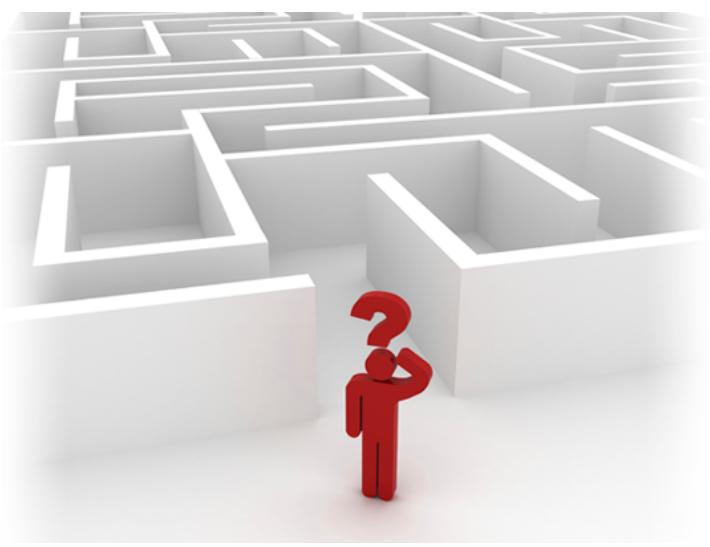
Dselect Theorem : for every input array of length n ,
Dselect runs in $O(n)$ time.

Warning : not as good as Rselect in practice

- 1) Worse constraints
- 2) not-in-place

History : from 1973

Blum – Floyd – Pratt – Rivest – Tarjan
(‘95) (‘78) (‘02) (‘86)



Design and Analysis
of Algorithms I

Linear-Time Selection

Deterministic Selection (Analysis)

The DSelect Algorithm

DSelect(array A, length n, order statistic i)

1. Break A into groups of 5, sort each group

2. C = the $n/5$ “middle elements”

3. $p = \text{DSelect}(C, n/5, n/10)$ [recursively computes median of C]

4. Partition A around p

5. If $j = i$ return p

6. If $j < i$ return DSelect(1^{st} part of A, $j-1$, i)

7. [else if $j > i$] return DSelect(2^{nd} part of A, $n-j$, $i-j$)

Choose
Pivot

Same as
before

What is the asymptotic running time of step 1 of the DSelect algorithm?

- $\theta(1)$
- $\theta(\log n)$
- $\theta(n)$
- $\theta(n \log n)$

Note : sorting an array with 5 elements takes
 ≤ 120 operations

[why 120 ? Take $m = 5$ in our $6m(\log_2 m + 1)$ bound for Merge Sort]

$$6 * 5 * (\log_2 5 + 1) \leq 120$$

≤ 3

of gaps ops per group

So : $\leq (n/5) * 120 = 24n = O(n)$ for all groups

The DSelect Algorithm

DSelect(array A, length n, order statistic i) $\theta(n)$

1. Break A into groups of 5, sort each group $\theta(n)$
2. C = the $n/5$ “middle elements” $\theta(n)$
3. p = DSelect(C, $n/5$, $n/10$) [recursively computes median of C]
4. Partition A around p $T\left(\frac{n}{5}\right)$
5. If $j = i$ return p $\theta(n)$
6. If $j < i$ return DSelect(1st part of A, $j-1$, i) $T(?)$
7. [else if $j > i$] return DSelect(2nd part of A, $n-j$, $i-j$)

Rough Recurrence

Let $T(n)$ = maximum running time of Dselect on an input array of length n .

There is a constant $c \geq 1$ such that :

1. $T(1) = 1$
2. $T(n) \leq c*n + T(n/5) + T(?)$

sorting the groups
partition

recursive
call in line 3

recursive call in
line 6 or 7

The Key Lemma

Key Lemma : 2nd recursive call (in line 6 or 7) guaranteed to be on an array of size $\leq 7n/10$ (roughly)

Upshot : can replace “?” by “7n/10”

Rough Proof : Let $k = n/5 = \# \text{ of groups}$
Let $x_i = i^{\text{th}}$ smallest of the k “middle elements”
[So pivot = $x_{k/2}$]

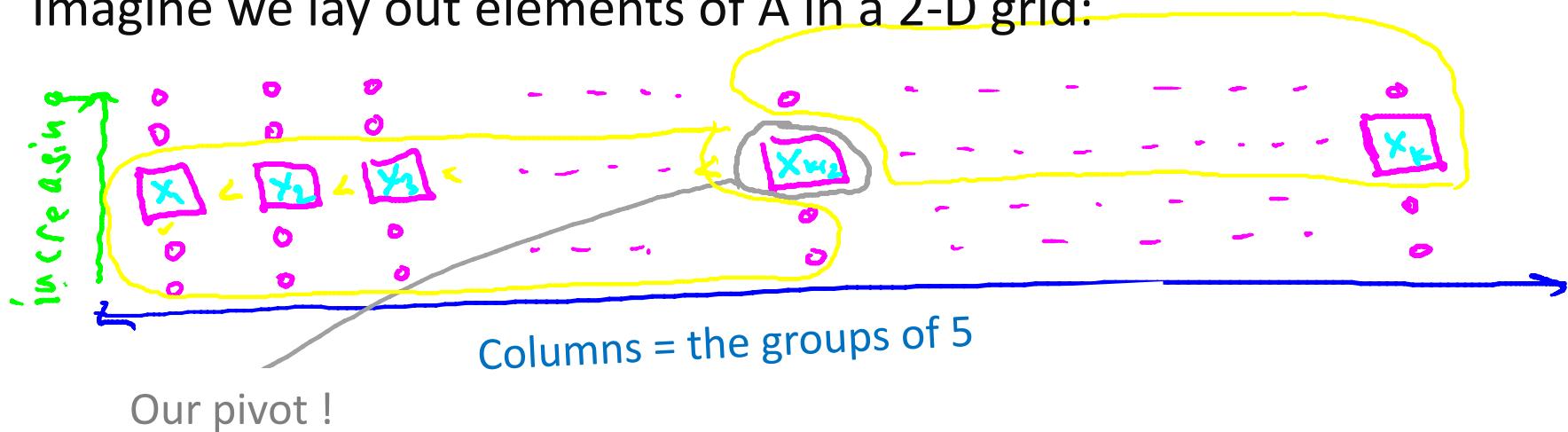
Goal : $\geq 30\%$ of input array smaller than $x_{k/2}$,
 $\geq 30\%$ is bigger

Tim Roughgarden

Rough Proof of Key Lemma

Thought Experiment :

Imagine we lay out elements of A in a 2-D grid:

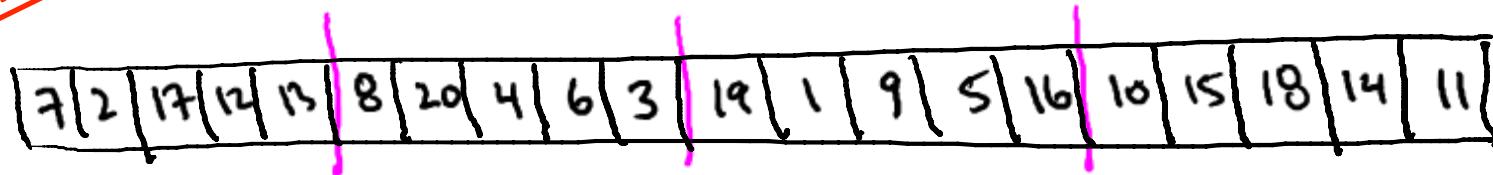


Key point : $x_{k/2}$ bigger than 3 out of 5 (60%) of the elements in
~ 50% of the groups

=> bigger than 30% of A (similarly, smaller than 30% of A)

Example

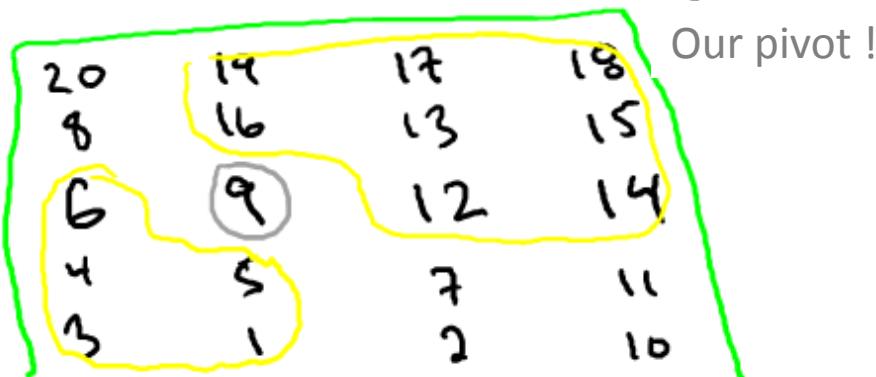
Input



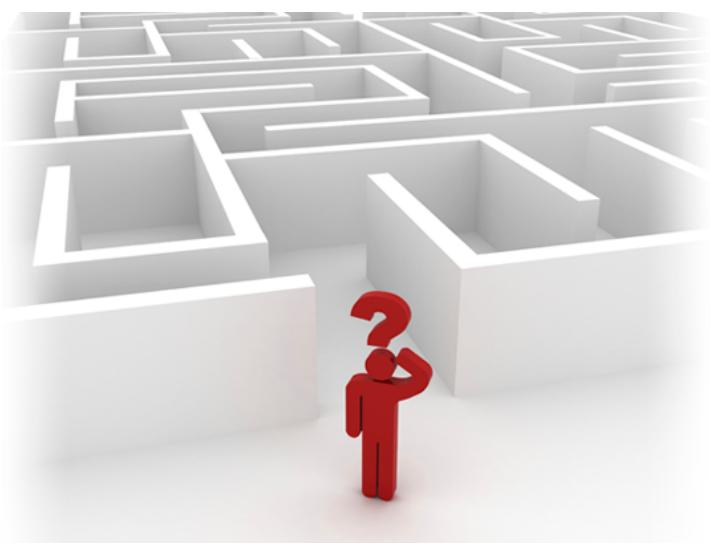
After
sorting
groups
of 5



The
grid :



Tim Roughgarden



Design and Analysis
of Algorithms I

Linear-Time Selection

Deterministic
Selection (Analysis II)

Rough Recurrence (Revisited)

Let $T(n)$ = maximum running time of Dselect on an input array of length n .

There is a constant $c \geq 1$ such that :

1. $T(1) = 1$
2. $T(n) \leq c*n + T(n/5) + T(\text{?})$

$\leq 7n/10$ by
Key Lemma

sorting the groups recursive recursive call in
partition call in line 3 line 6 or 7

Rough Recurrence (Revisited)

$$T(1) = 1, T(n) \leq cn + T(n/5) + T(7n/10)$$

Constant $c \geq 1$

Note : different-sized subproblems => can't use Master Method!

Strategy : “hope and check”

Hope : there is some constant a [independent of n]

Such that $T(n) \leq an$ for all $n \geq 1$

[if true, then $T(n) = O(n)$ and algorithm is linear time]

Analysis of Rough Recurrence

Claim : Let $a = 10c$

Then $T(n) \leq an$ for all $n \geq 1$

=> Dselect runs in
 $O(n)$ time

$$T(1) = 1 ; T(n) \leq cn + T(n/5) + T(7n/10)$$

Constant $c \geq 1$

Proof : by induction on n

Base case : $T(1) = 1 \leq a*1$ (since $a \geq 1$)

Inductive Step : $[n > 1]$

Inductive Hypothesis : $T(k) \leq ak \forall k < n$

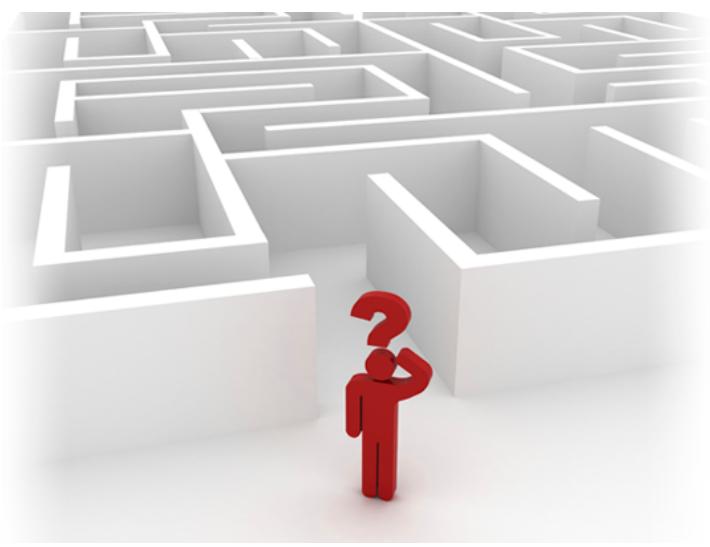
We have $T(n) \leq cn + T(n/5) + T(7n/10)$

$$\begin{aligned} &\stackrel{\text{GIVEN}}{\leq} cn + a(n/5) + a(7n/10) \\ &\stackrel{\text{IND HYP}}{=} n(c + 9a/10) = an \end{aligned}$$

Choice of a

Q.E.D.

Tim Roughgarden



Design and Analysis
of Algorithms I

Linear-Time Selection

Randomized Selection (Algorithm)

Prerequisites

Watch this after:

- QuickSort - Partitioning around a pivot
- QuickSort – Choosing a good pivot
- Probability Review, Part I

The Problem

Input : array A with n **distinct** numbers and a number

For simplicity

Output : i^{th} order statistic (i.e., i^{th} smallest element of input array)

Example : median.

($i = (n+1)/2$ for n odd,
 $i = n/2$ for n even)



3rd order statistic

Reduction to Sorting

O(nlog(n)) algorithm

- 1) Apply MergeSort
- 2) return i^{th} element of sorted array

Fact : can't sort any faster [see optional video]

Next : $O(n)$ time (randomized) by modifying Quick Sort.

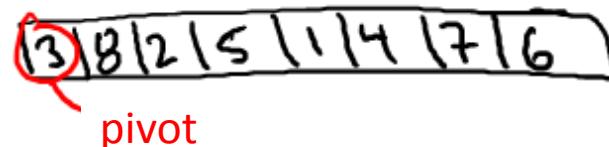
Optional Video : $O(n)$ time deterministic algorithm.

-- pivot = “median of medians” (warning : not practical)

Partitioning Around a Pivot

Key Idea : partition array around a pivot element.

-Pick element of array



-Rearrange array so that

- Left of pivot => less than pivot

- Right of pivot => greater than pivot



Note : puts pivot in its “rightful position”.

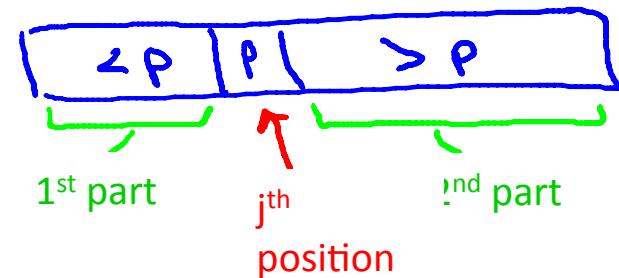
Suppose we are looking for the 5th order statistic in an input array of length 10. We partition the array, and the pivot winds up in the third position of the partitioned array. On which side of the pivot do we recurse, and what order statistic should we look for?

- The 3rd order statistic on the left side of the pivot.
- The 2nd order statistic on the right side of the pivot.
- The 5th order statistic on the right side of the pivot.
- Not enough information to answer question – we might need to recurse on the left or the right side of the pivot.

Randomized Selection

Rselect (array A, length n, order statistic i)

- 0) if $n = 1$ return $A[1]$
- 1) Choose pivot p from A uniformly at random
- 2) Partition A around p
let $j = \text{new index of } p$
- 3) If $j = i$, return p
- 4) If $j > i$, return Rselect(1^{st} part of A , $j-1$, i)
- 5) [if $j < i$] return Rselect (2^{nd} part of A , $n-j$, $i-j$)



Properties of RSelect

Claim : Rselect is correct (guaranteed to output ith order statistic)

Proof : by induction. [like in optional QuickSort video]

Running Time ? : depends on “quality” of the chosen pivots.

What is the running time of the RSelect algorithm if pivots are always chosen in the worst possible way?

- $\theta(n)$
- $\theta(n \log n)$
- $\theta(n^2)$
- $\theta(2^n)$

Example :

-- suppose $i = n/2$
-- suppose choose pivot = minimum
every time
 $\Rightarrow \Omega(n)$ time in each of $\Omega(n)$ recursive calls

Running Time of RSelect?

Running Time ? : depends on which pivots get chosen.
(could be as bad as $\theta(n^2)$)

Key : find pivot giving “balanced” split.

Best pivot: the median ! (but this is circular)

⇒ Would get recurrence $T(n) \leq T(n/2) + O(n)$

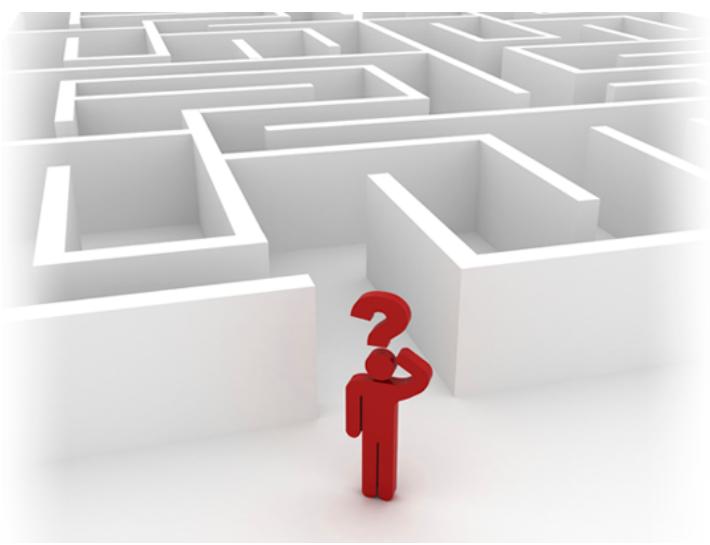
⇒ $T(n) = O(n)$ [case 2 of Master Method]

Hope : random pivot is “pretty good” “often enough”

Running Time of RSelect

Rselect Theorem : for every input array of length n , the average running time of Rselect is $O(n)$

- holds for every input [no assumptions on data]
- “average” is over random pivot choices made by the algorithm



Design and Analysis
of Algorithms I

Linear-Time Selection

Randomized Selection (Analysis)

Running Time of RSelect

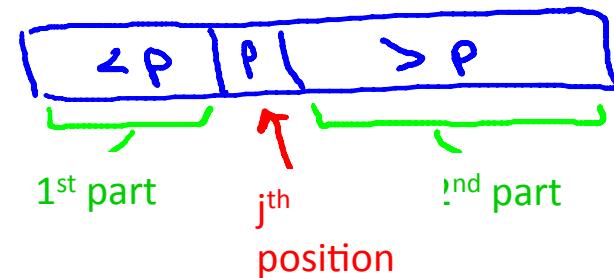
Rselect Theorem : for every input array of length n , the average running time of Rselect is $O(n)$

- holds for every input [no assumptions on data]
- “average” is over random pivot choices made by the algorithm

Randomized Selection

Rselect (array A, length n, order statistic i)

- 0) if $n = 1$ return $A[1]$
- 1) Choose pivot p from A uniformly at random
- 2) Partition A around p
let j = new index of p
- 3) If $j = i$, return p
- 4) If $j > i$, return Rselect(1^{st} part of A , $j-1$, i)
- 5) [if $j < i$] return Rselect (2^{nd} part of A , $n-j$, $i-j$)



Proof I: Tracking Progress via Phases

Note : Rselect uses $\leq cn$ operations outside of recursive call [for some constant $c > 0$] [from partitioning]

Notation : Rselect is in phase j if current array size between $(\frac{3}{4})^{j+1} \cdot n$ and $(\frac{3}{4})^j \cdot n$

- X_j = number of recursive calls during phase j

$$\text{Note : running time of RSelect} \leq \sum_{\text{phases } j} X_j \cdot \underbrace{c \cdot (\frac{3}{4})^j \cdot n}_{\begin{array}{l} \text{\# of phase } j \text{ subproblems} \\ \text{Work per phase } j \text{ subproblem} \end{array}}$$

=<= array size during phase j

Tim Roughgarden

Proof II: Reduction to Coin Flipping

$X_j = \# \text{ of recursive calls during phase } j$ → Size between $(\frac{3}{4})^{j+1} \cdot n$ and $(\frac{3}{4})^j \cdot n$

Note : if Rselect chooses a pivot giving a 25 – 75 split (or better) then current phase ends !
(new subarray length at most 75 % of old length)



Recall : probability of 25-75 split or better is 50%

So : $E[X_j] \leq$ expected number of times you need to flip a fair coin
to get one “heads”
(heads ~ good pivot, tails ~ bad pivot)

Tim Roughgarden

Proof III: Coin Flipping Analysis

Let N = number of coin flips until you get heads.
(a “geometric random variable”)

Note : $E[N] = 1 + (1/2)*E[N]$

1st coin flip Probability of tails # of further coin flips needed in this case

Solution : $E[N] = 2$ (Recall $E[X_j] \leq E[N]$)

Putting It All Together

Expected
running time of
RSelect

$$\leq E[cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^j X_j] \quad (*)$$

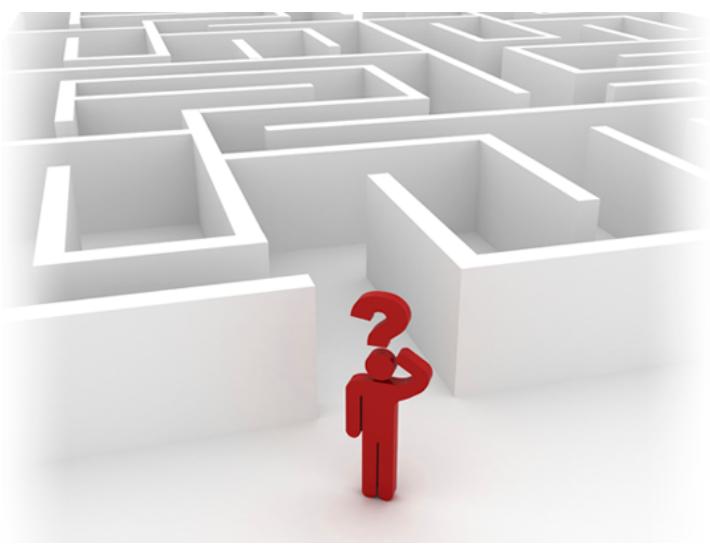
$$= cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^j E[X_j] \quad [\text{LIN EXP}]$$

= E[# of coin flips N] = 2

$$\leq 2cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^j$$

geometric sum,
 $\leq 1/(1-3/4) = 4$

$$\leq 8cn = O(n) \quad \text{Q.E.D.}$$



Design and Analysis
of Algorithms I

Linear-Time Selection

An $\Omega(n \log n)$
Sorting Lower Bound

A Sorting Lower Bound

Theorem : every “comparison-based” sorting algorithm has worst-case running time $\Omega(n \log n)$

[assume deterministic, but lower bound extends to randomized]

Comparison-Based Sort : accesses input array elements only via comparisons ~ “general purpose sorting method”

Examples : Merge Sort, Quick Sort, Heap Sort

Non Examples : Bucket Sort, Counting Sort, Radix Sort

Good for data from distributions → good for small integers → good for medium-size integers

Tim Roughgarden

Proof Idea

Fix a comparison-based sorting method and an array length n

⇒ Consider input arrays containing $\{1, 2, 3, \dots, n\}$ in some order.

⇒ $n!$ such inputs

Suppose algorithm always makes $\leq k$ comparisons to correctly sort these $n!$ inputs.

=> Across all $n!$ possible inputs, algorithm exhibits $\leq 2^k$ distinct executions → i.e., resolution of the comparisons

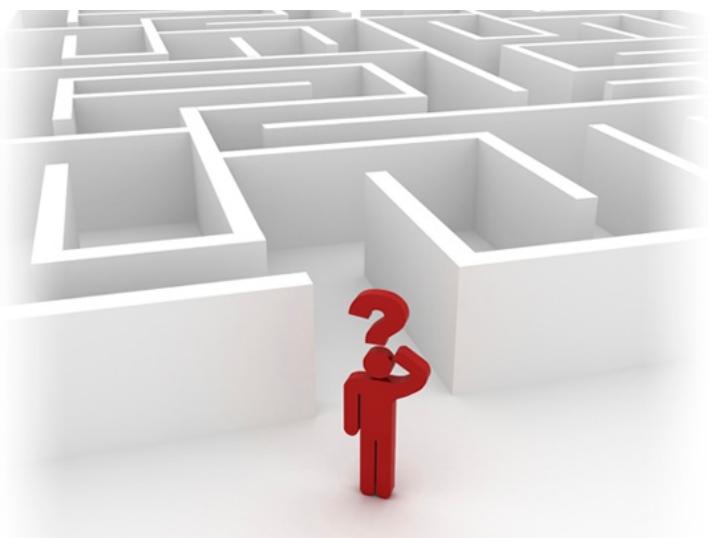
Tim Roughgarden

Proof Idea (con'd)

By the Pigeonhole Principle : if $2^k < n!$, execute identically on two distinct inputs => must get one of them incorrect.

So : Since method is correct,

$$\begin{aligned} 2^k &\geq n! \\ &\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \\ \Rightarrow k &\geq \frac{n}{2} \cdot \log_2 \frac{n}{2} = \Omega(n \log n) \end{aligned}$$



Contraction Algorithm

The Algorithm

Design and Analysis
of Algorithms I

The Minimum Cut Problem

- INPUT: An undirected graph $G = (V, E)$.
[Parallel  edges allowed]
[See other video for representation of the input]
- GOAL: Compute a cut with fewest number of crossing edges. (a min cut)

Random Contraction Algorithm

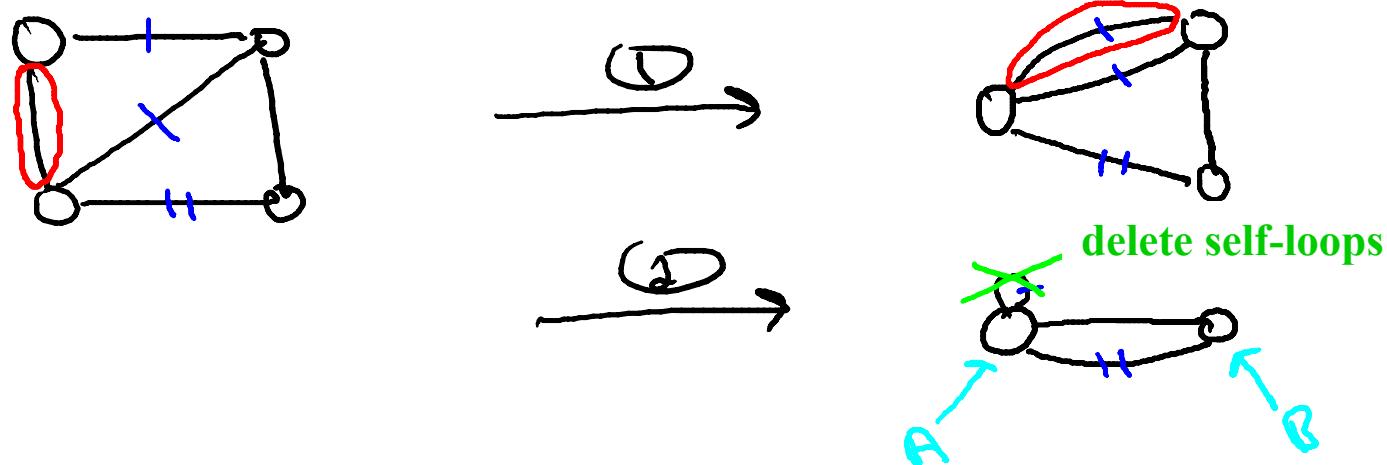
[due to Karger, early 90s]

While there are more than 2 vertices:

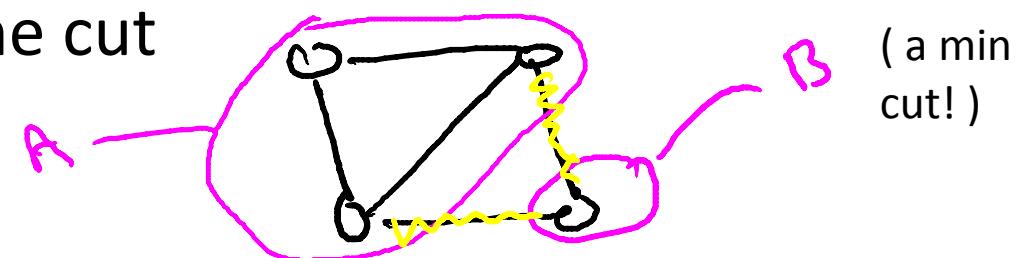
- pick a remaining edge (u,v) uniformly at random
- merge (or “contract”) u and v into a single vertex
- remove self-loops

return cut represented by final 2 vertices.

Example

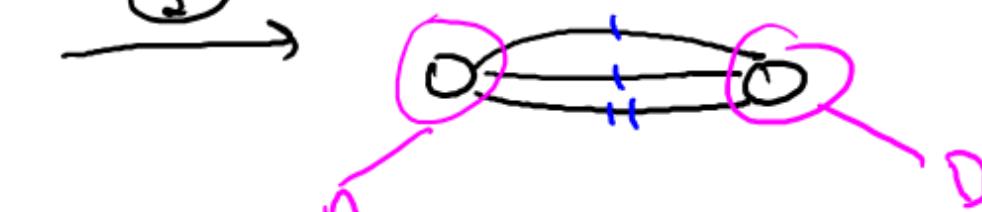
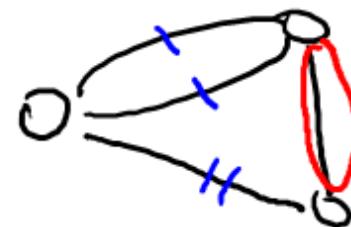
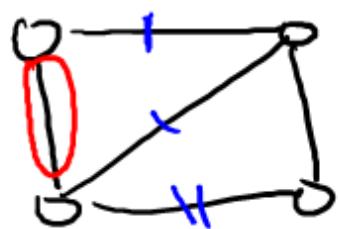


=> Corresponds to the cut

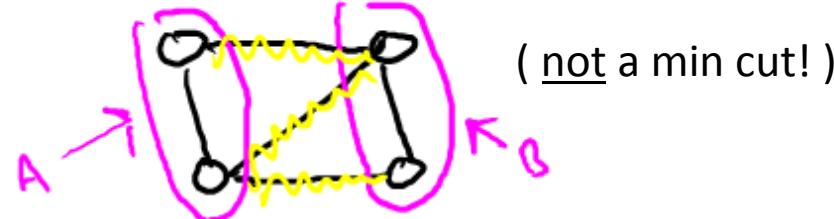


Tim Roughgarden

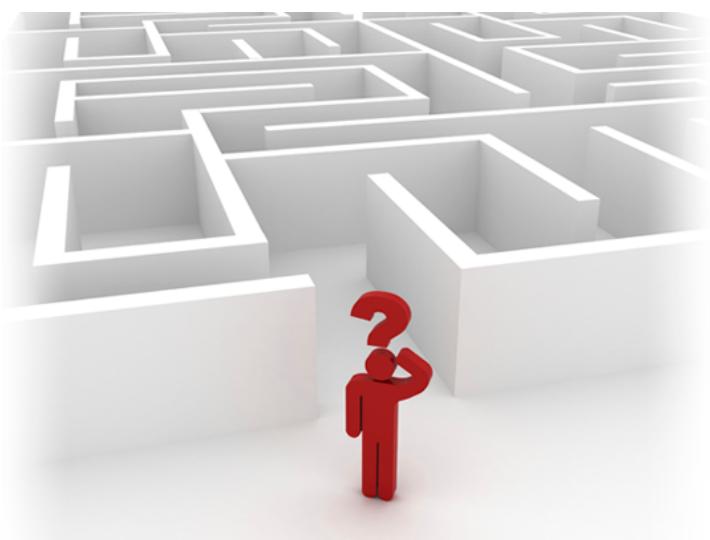
Example (con'd)



- Corresponds to the cut



Tim Roughgarden



Contraction Algorithm

The Analysis

Design and Analysis
of Algorithms I

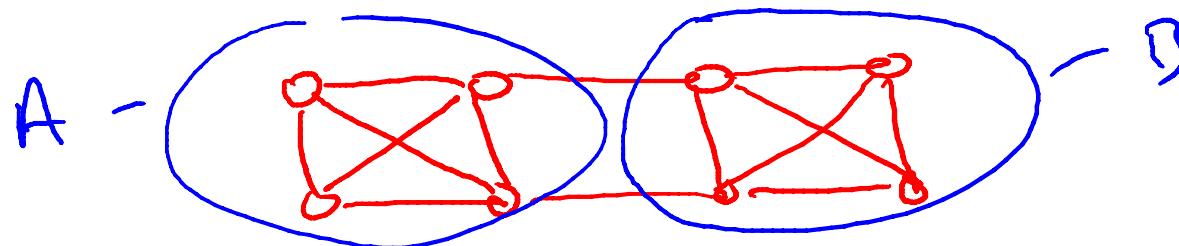
The Minimum Cut Problem

Input: An undirected graph $G = (V, E)$.

[parallel edges  allowed]

[See other video for representation of input]

Goal: Compute a cut with fewest number of crossing edges.
(a min cut)



Random Contraction Algorithm

[due to Karger, early 90s]

While there are more than 2 vertices:

- pick a remaining edge (u,v) uniformly at random
- merge (or “contract”) u and v into a single vertex
- remove self-loops

return cut represented by final 2 vertices.

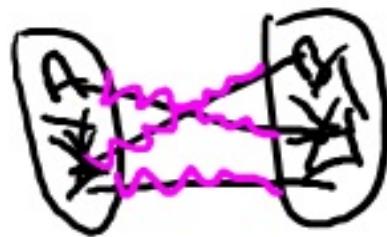
The Setup

Question: what is the probability of success?

Fix a graph $G = (V, E)$ with n vertices, m edges.

Fix a minimum cut (A, B) .

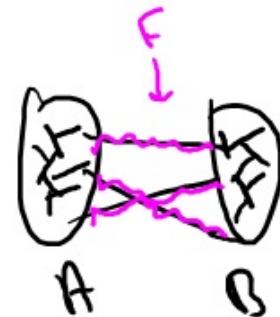
Let $k = \#$ of edges crossing (A, B) . (Call these edges F)



Tim Roughgarden

What Could Go Wrong?

1. Suppose an edge of F is contracted at some point
⇒ algorithm will not output (A, B).



2. Suppose only edges inside A or inside B get contracted ⇒ algorithm will output (A, B).

Thus: $\Pr [\text{output is } (A, B)] = \Pr [\text{never contracts an edge of } F]$

Let S_i = event that an edge of F contracted in iteration i.

Goal: Compute $\Pr[\neg S_1 \wedge \neg S_2 \wedge \neg S_3 \wedge \dots \wedge \neg S_{n-2}]$

What is the probability that an edge crossing the minimum cut (A, B) is chosen in the first iteration (as a function of the number of vertices n , the number of edges m , and the number k of crossing edges)?

- k/n
- k/m
- k/n^2
- n/m

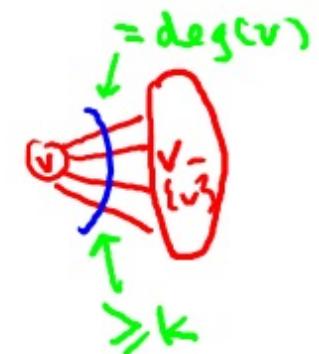
$$\Pr[S_1] = \frac{\text{\# of crossing edges}}{\text{\# of edges}} = \frac{k}{m}$$

The First Iteration

Key Observation: degree of each vertex is at least k
 # of incident edges

Reason: each vertex v defines a cut $(\{v\}, V - \{v\})$.

Since $\sum_v \underbrace{\text{degree}(v)}_{\geq kn} = 2m$, we have $m \geq \frac{kn}{2}$



Since $\Pr[S_1] = \frac{k}{m}$, $\Pr[S_1] \leq \frac{2}{n}$

The Second Iteration

$$\begin{aligned}
 \text{Recall: } \Pr[\neg S_1 \wedge \neg S_2] &= \Pr[\neg S_2 | \neg S_1] \cdot \Pr[\neg S_1] \\
 &= 1 - \frac{k}{\# \text{ of remaining edge}} \geq \left(1 - \frac{2}{n}\right)
 \end{aligned}$$

what is this?

Note: all nodes in contracted graph define cuts in G
 (with at least k crossing edges).

➤ all degrees in contracted graph are at least k

So: # of remaining edges $\geq \frac{1}{2}k(n-1)$

$$\text{So } \Pr[\neg S_2 | \neg S_1] \geq 1 - \frac{2}{(n-1)}$$

All Iterations

In general:

$$\begin{aligned}
 & \Pr[\neg S_1 \wedge \neg S_2 \wedge \neg S_3 \wedge \dots \wedge \neg S_{n-2}] \\
 &= \underbrace{\Pr[\neg S_1]}_{\text{Pr}} \underbrace{\Pr[\neg S_2 | \neg S_1]}_{\text{Pr}} \Pr[\neg S_3 | \neg S_2 \wedge \neg S_1] \dots \Pr[\neg S_{n-2} | \neg S_1 \wedge \dots \wedge \neg S_{n-3}] \\
 &\geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \dots \left(1 - \frac{2}{n-(n-4)}\right) \left(1 - \frac{2}{n-(n-3)}\right) \\
 &= \cancel{\frac{n-2}{n}} \cdot \cancel{\frac{n-3}{n-1}} \cdot \cancel{\frac{n-4}{n-2}} \dots \cancel{\frac{2}{4}} \cdot \cancel{\frac{1}{3}} = \frac{2}{n(n-1)} \geq \frac{1}{n^2}
 \end{aligned}$$

Problem: low success probability! (But: non trivial)

recall $\simeq 2^n$ cuts!

Repeated Trials

Solution: run the basic algorithm a large number N times, remember the smallest cut found.

Question: how many trials needed? 

Let T_i = event that the cut (A, B) is found on the i^{th} try.

➤ by definition, different T_i 's are independent

So: $\Pr[\text{all } N \text{ trials fail}] = \Pr[\neg T_1 \wedge \neg T_2 \wedge \dots \wedge \neg T_N]$

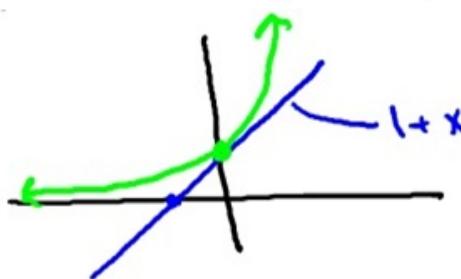
$$= \prod_{i=1}^N \Pr[\neg T_i] \leq \left(1 - \frac{1}{n^2}\right)^N$$

By independence!

Repeated Trials (con'd)

Calculus fact: \forall real numbers x , $1+x \leq e^x$

$$\Pr[\text{all trials fail}] \leq (1 - \frac{1}{n^2})^N$$

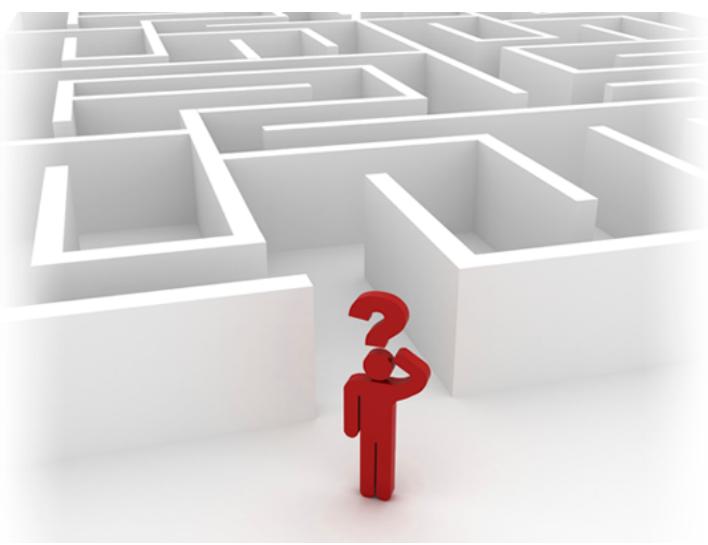


So: if we take $N = n^2$, $\Pr[\text{all fail}] \leq (e^{-\frac{1}{n^2}})^{n^2} = \frac{1}{e}$

If we take $N = n^2 \ln n$, $\Pr[\text{all fail}] \leq (\frac{1}{e})^{\ln n} = \frac{1}{n}$

Running time: polynomial in n and m but slow ($\Omega(n^2m)$)

But: can get big speed ups (to roughly $O(n^2)$) with more ideas.



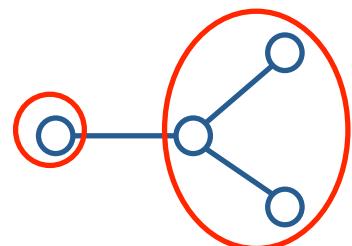
Design and Analysis
of Algorithms I

Contraction Algorithm

Counting Mininum Cuts

The Number of Minimum Cuts

NOTE: A graph can have multiple min cuts.
[e.g., a tree with n vertices has $(n-1)$ minimum cuts]



QUESTION: What's the largest number of min cuts that a graph with n vertices can have?

ANSWER:

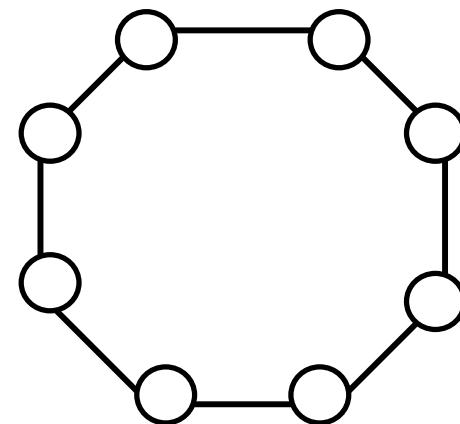
$$\binom{n}{2} = \frac{n(n - 1)}{2}$$

The Lower Bound

Consider the n-cycle.

NOTE: Each pair of the n edges defines
a distinct minimum cut
(with two crossing edges).

➤ has $\geq \binom{n}{2}$ min cuts



The Upper Bound

Let $(A_1, B_1), (A_2, B_2), \dots, (A_t, B_t)$ be the min cuts of a graph with n vertices.

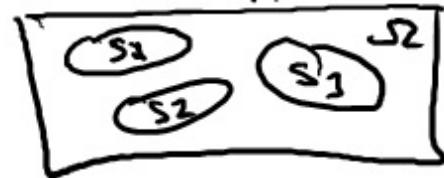
By the Contraction Algorithm analysis (without repeated trials):

$$\Pr[\underbrace{\text{output} = (A_i, B_i)}_{S_i}] \geq \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \quad \forall i = 1, 2, \dots, t$$

Note: S_i 's are disjoint events (i.e., only one can happen)

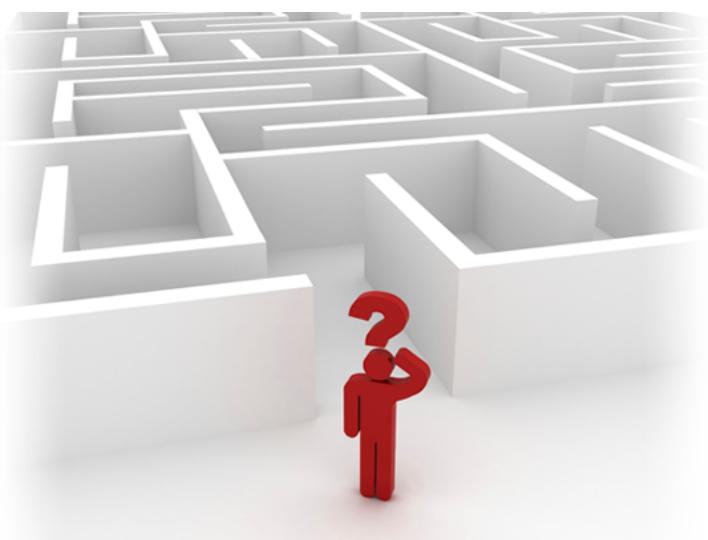
➤ their probabilities sum to at most 1

Thus: $\frac{t}{\binom{n}{2}} \leq 1 \Rightarrow t \leq \binom{n}{2}$



QED !

Tim Roughgarden



Contraction Algorithm

Overview

Design and Analysis
of Algorithms I

Goals for These Lectures

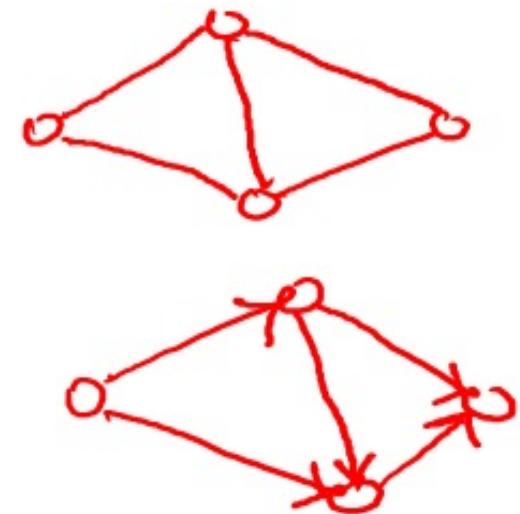
- Further practice with randomized algorithms
 - In a new application domain (graphs)
- Introduction to graphs and graph algorithms

Also: “only” 20 years ago!

Graphs

Two ingredients

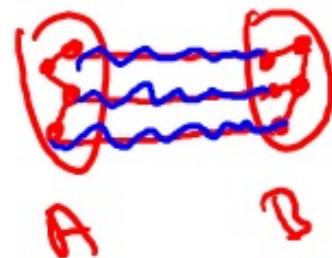
- Vertices aka nodes (V)
- Edges (E) = pairs of vertices
 - can be undirected [unordered pair] or directed [ordered pair] (aka arcs)



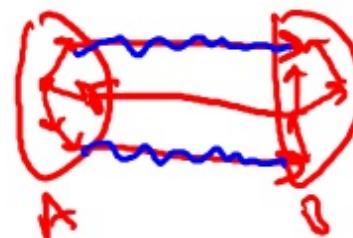
Examples: road networks, the Web, social networks, precedence constraints, etc.

Cuts of Graphs

Definition: a cut of a graph (V, E) is a partition of V into two non-empty sets A and B .



[undirected]



[directed]

Definition: the crossing edges of a cut (A, B) are those with:

- the one endpoint in each of (A, B) [undirected]
- tail in A , head in B [directed]

Roughly how many cuts does a graph with n vertices have?

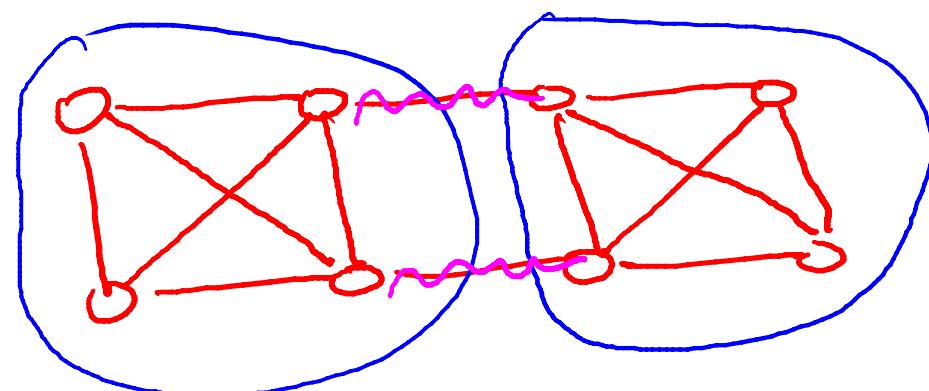
- n
- n^2
- 2^n
- n^n

The Minimum Cut Problem

- INPUT: An undirected graph $G = (V, E)$.
[Parallel  edges allowed]
[See other video for representation of the input]
- GOAL: Compute a cut with fewest number of crossing edges. (a min cut)

What is the number of edges crossing a minimum cut in the graph shown below?

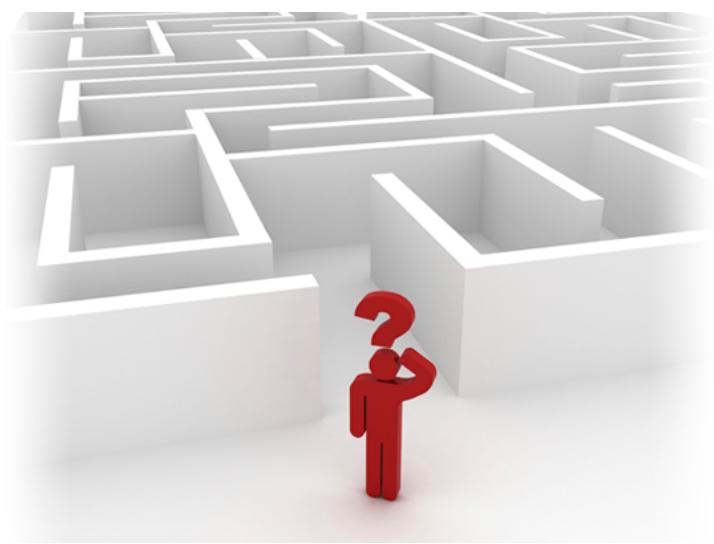
- 1
- 2
- 3
- 4



A Few Applications

- identify network bottlenecks / weaknesses
- community detection in social networks
- image segmentation
 - input = graph of pixels
 - use edge weights
 - [(u,v) has large weight \Leftrightarrow “expect” u,v to come from some object]

hope: repeated min cuts identifies the primary objects in picture.



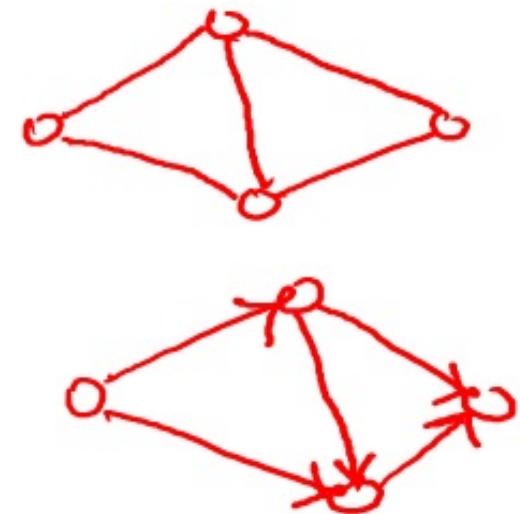
Design and Analysis
of Algorithms I

Graph Algorithms Representing Graphs

Graphs

Two ingredients

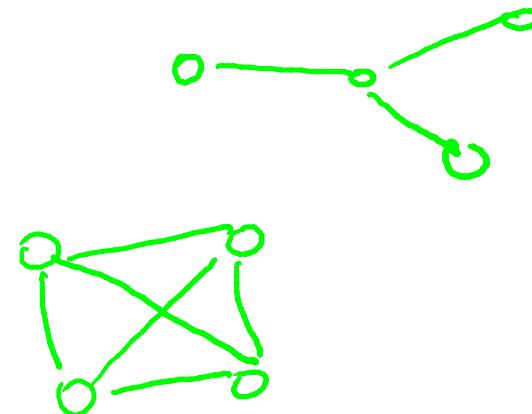
- Vertices aka nodes (V)
- Edges (E) = pairs of vertices
 - can be undirected [unordered pair] or directed [ordered pair] (aka arcs)



Examples: road networks, the Web, social networks, precedence constraints, etc.

Consider an undirected graph that has n vertices, no parallel edges, and is connected (i.e., “in one piece”). What is the minimum and maximum number of edges that the graph could have, respectively ?

- $n - 1$ and $n(n - 1)/2$
- $n - 1$ and n^2
- n and 2^n
- n and n^n



Sparse vs. Dense Graphs

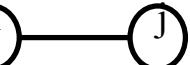
Let $\underline{n} = \#$ of vertices, $\underline{m} = \#$ of edges.

In most (but not all) applications, m is $\Omega(n)$ and $O(n^2)$

- in a “sparse” graph, m is or is close to $O(n)$
- in a “dense” graph, m is closer to $\theta(n^2)$

The Adjacency Matrix

Represent G by a $n \times n$ 0-1 matrix A where

$$A_{ij} = 1 \Leftrightarrow G \text{ has an } i-j \text{ edge}$$


Variants

- $A_{ij} = \# \text{ of } i-j \text{ edges}$ (if parallel edges)
- $A_{ij} = \text{weight of } i-j \text{ edge}$ (if any)
- $A_{ij} = \begin{cases} +1 & \text{if } \text{○} \rightarrow \text{○} \\ -1 & \text{if } \text{○} \leftarrow \text{○} \end{cases}$

How much space does an adjacency matrix require, as a function of the number n of vertices and the number m of edges?

- $\theta(n)$
- $\theta(m)$
- $\theta(m + n)$
- $\theta(n^2)$

Adjacency Lists

Ingredients

- array (or list) of vertices
- array (or list) of edges
- each edge points to its endpoints
- each vertex points to edges incident on it

How much space does an adjacency list representation require, as a function of the number n of vertices and the number m of edges?

$\theta(n)$

$\theta(m)$

$\theta(m + n)$

$\theta(n^2)$

Adjacency Lists

Ingredients

- array (or list) of vertices
- array (or list) of edges
- each edge points to its endpoints
- each vertex points to edges incident on it

one-to-one

correspondence !

Space

$\theta(n)$

$\theta(m)$

$\theta(m)$

$\theta(m)$

$\theta(m + n)$

[or $\theta(\max\{m, n\})$]

Question: which is better?

Answer: depends on graph density and operations needed.

This course: focus on adjacency lists.



Graph Primitives

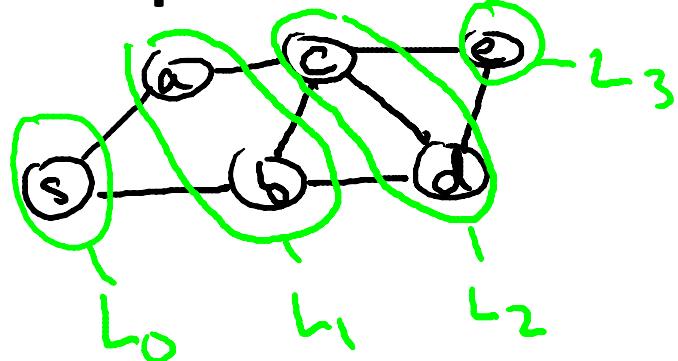
Breadth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Breadth-First Search (BFS)

- explore nodes in “layers”
- can compute shortest paths
- connected components of undirected graph



Run time : $O(m+n)$ [linear time]

The Code

BFS (graph G, start vertex s)

[all nodes initially unexplored]

-- mark s as explored

-- let Q = queue data structure (FIFO), initialized with s

-- while $Q \neq \emptyset$:

-- remove the first node of Q, call it v

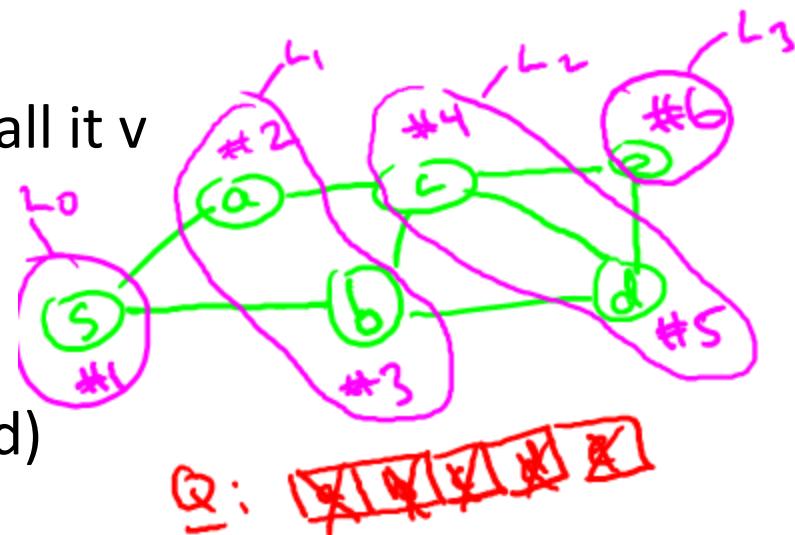
-- for each edge(v,w) :

-- if w unexplored

--mark w as explored

-- add w to Q (at the end)

O(1)
time



Basic BFS Properties

Claim #1 : at the end of BFS, v explored \iff
 G has a path from s to v .

Reason : special case of the generic algorithm

Claim #2 : running time of main while loop
= $O(n_s + m_s)$, where n_s = # of nodes reachable from s
 m_s = # of edges reachable from s

Reason : by inspection of code.

Application: Shortest Paths

Goal : compute $\text{dist}(v)$, the fewest # of edges on path from s to v .

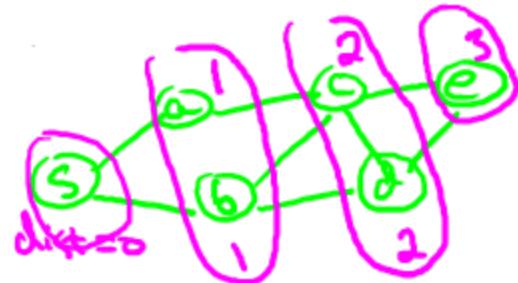
Extra code : initialize $\text{dist}(v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$

-When considering edge (v,w) :

- if w unexplored, then set $\text{dist}(w) = \text{dist}(v) + 1$

Claim : at termination $\text{dist}(v) = i \iff v$ in i th layer
(i.e., shortest s - v path has i edges)

Proof Idea : every layer i node w is added to Q by a layer $(i-1)$ node v via the edge (v,w)



Application: Undirected Connectivity

Let $G = (V, E)$ be an undirected graph.

Connected components = the “pieces” of G .

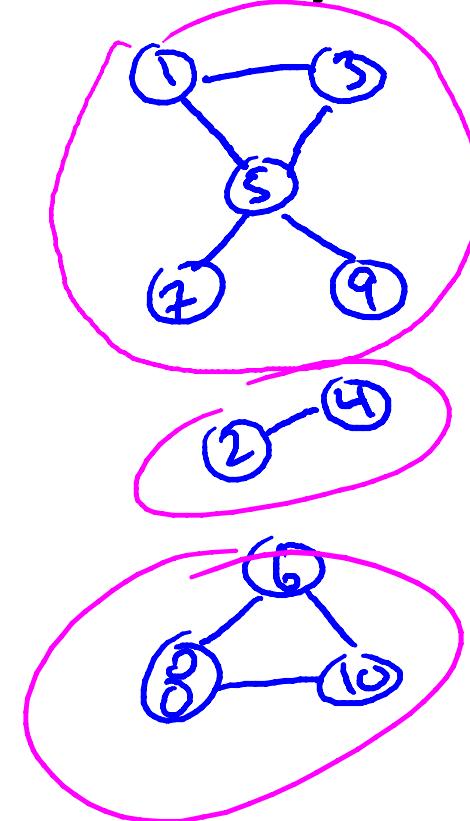
Formal Definition : equivalence classes of
the relation $u \leftrightarrow v \iff$ there exists $u-v$ path
in G . [check: \leftrightarrow is an equivalence relation]

Goal : compute all connected components

Why? - check if network is disconnected

- graph visualisation

- clustering



Connected Components via BFS

To compute all components : (undirected case)

-- initialize all nodes as unexplored $O(n)$

[assume labelled 1 to n]

-- for $i = 1$ to n $O(n)$

-- if i not yet explored [in some previous BFS]

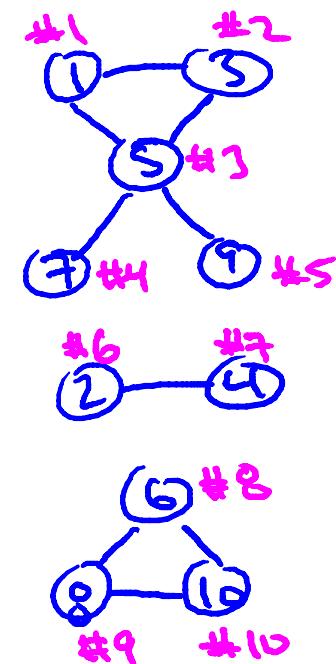
-- $\text{BFS}(G, i)$ [discovers precisely i 's connected component]

Note : finds every connected component.

Running time : $O(m+n)$

$O(1)$ per node

$O(1)$ per edge in each BFS





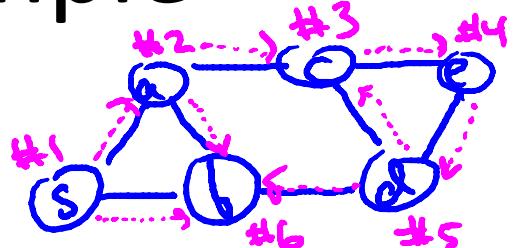
Graph Primitives

Depth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Depth-First Search (DFS) : explore aggressively, only backtrack when necessary.



- also computes a topological ordering of a directed acyclic graph
- and strongly connected components of directed graphs

Run Time : $O(m+n)$

The Code

Exercise : mimic BFS code, use a stack instead of a queue [+
some other minor modifications]

Recursive version : DFS(graph G, start vertex s)

- mark s as explored
- for every edge (s,v) :
 - if v unexplored
 - $\text{DFS}(G,v)$

Basic DFS Properties

Claim #1 : at the end of the algorithm, v marked as explored
 \Leftrightarrow there exists a path from s to v in G.

Reason : particular instantiation of generic search procedure

Claim #2 : running time is $O(n_s + m_s)$,
where $n_s = \#$ of nodes reachable from s
 $m_s = \#$ of edges reachable from s

Reason : looks at each node in the connected component of s
at most once, each edge at most twice.

Application: Topological Sort

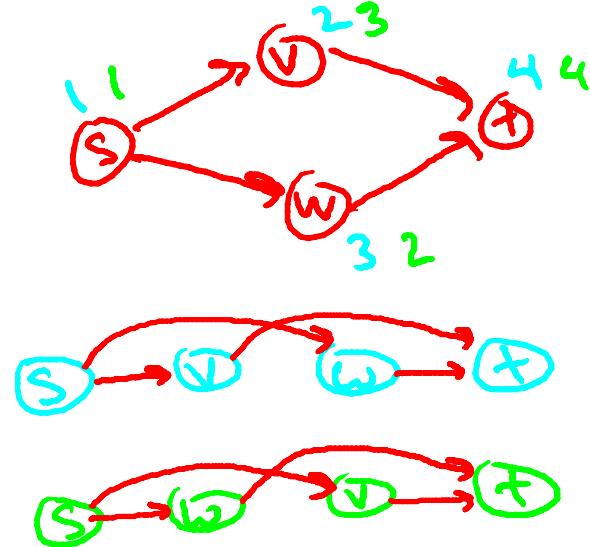
Definition : A topological ordering of a directed graph G is a labeling f of G 's nodes such that:

1. The $f(v)$'s are the set $\{1, 2, \dots, n\}$
2. $(u, v) \in G \Rightarrow f(u) < f(v)$

Motivation : sequence tasks while respecting all precedence constraints.

Note : G has directed cycle \Rightarrow no topological ordering

Theorem : no directed cycle \Rightarrow can compute topological ordering in $O(m+n)$ time.



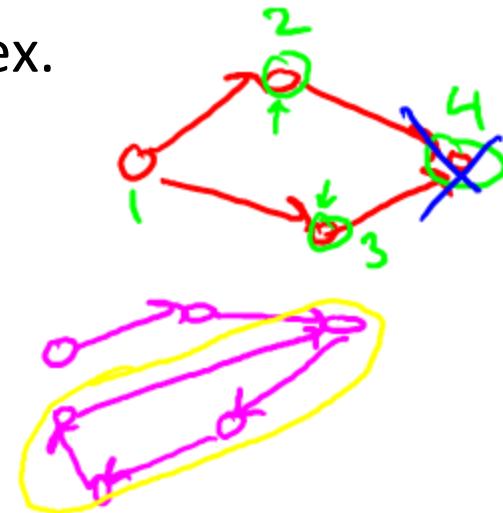
Straightforward Solution

Note : every directed acyclic graph has a sink vertex.

Reason : if not, can keep following outgoing arcs to produce a directed cycle.

To compute topological ordering :

- let v be a sink vertex of G
- set $f(v) = n$
- recurse on $G - \{v\}$



Why does it work? : when v is assigned to position i , all outgoing arcs already deleted => all lead to later vertices in ordering.

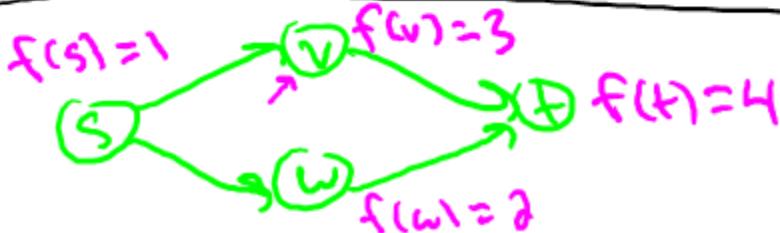
Topological Sort via DFS (Slick)

DFS-Loop (graph G)

- mark all nodes unexplored
- current-label = n [to keep track of ordering]
- for each vertex
 - if v not yet explored [in previous DFS call]
 - DFS(G,v)

DFS(graph G, start vertex s)

- for every edge (s,v)
 - if v not yet explored
 - mark v explored
 - DFS(G,v)
- set $f(s) = \text{current_label}$
- $\text{current_label} = \text{current_label} - 1$



Topological Sort via DFS (con'd)

Running Time : $O(m+n)$.

Reason : $O(1)$ time per node, $O(1)$ time per edge.

Correctness : need to show that if (u,v) is an edge,
then $f(u) < f(v)$



(since no
directed cycles)

Case 1 : u visited by DFS before $v \Rightarrow$ recursive call
corresponding to v finishes before that of u (since DFS).
 $\Rightarrow f(v) > f(u)$

Case 2 : v visited before $u \Rightarrow v$'s recursive call finishes before
 u 's even starts. $\Rightarrow f(v) > f(u)$

Q.E.D.



Design and Analysis
of Algorithms I

Graph Primitives

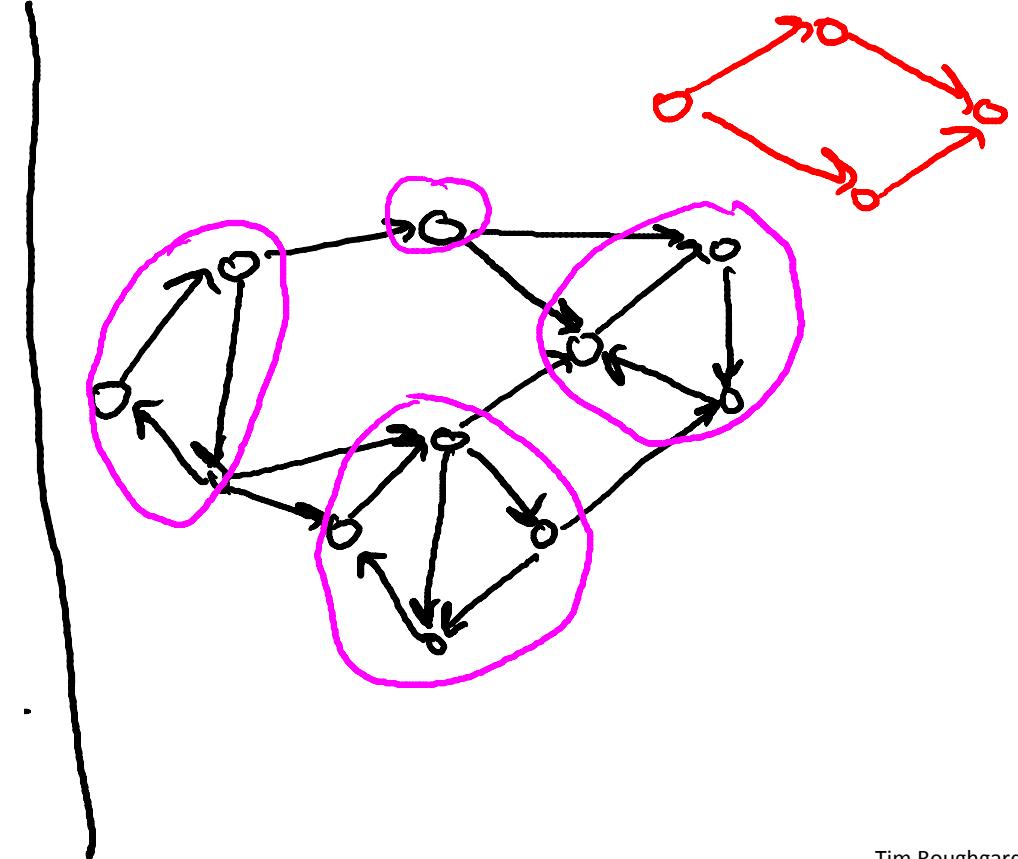
An $O(m+n)$ Algorithm
for Computing Strong
Components

Strongly Connected Components

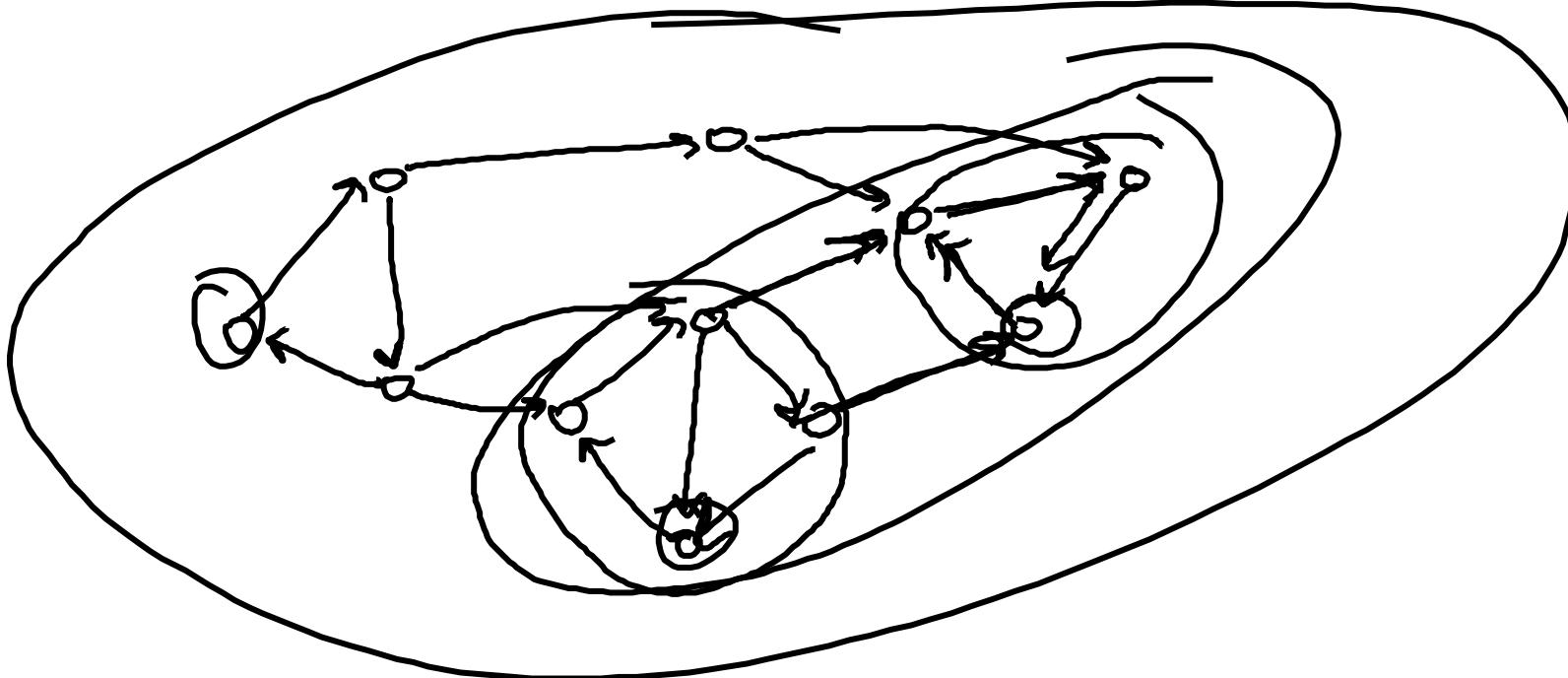
Formal Definition : the strongly connected components (SCCs) of a directed graph G are the equivalence classes of the relation

$u \leftrightarrow v \iff$ there exists a path $u \rightarrow v$ and a path $v \rightarrow u$ in G

You check : \leftrightarrow is an equivalence relation



Why Depth-First Search?



Kosaraju's Two-Pass Algorithm

Theorem : can compute SCCs in $O(m+n)$ time.

Algorithm : (given directed graph G)

1. Let $\text{Grev} = G$ with all arcs reversed
2. Run DFS-Loop on Grev ← Goal : compute “magical ordering” of nodes
Let $f(v) = \text{“finishing time” of each } v \text{ in } V$ Goal : discover the SCCs
1. Run DFS-Loop on G ← one-by-one
processing nodes in decreasing order of finishing times
[SCCs = nodes with the same “leader”]

DFS-Loop

DFS-Loop (graph G)

Global variable $t = 0$

For finishing
times in 1st
pass

[# of nodes processed so far]

For leaders
in 2nd pass

Global variable $s = \text{NULL}$

[current source vertex]

Assume nodes labeled 1 to n

For $i = n$ down to 1

if i not yet explored

$s := i$

$\text{DFS}(G, i)$

DFS (graph G, node i)

-- mark i as explored

For rest of
DFS-Loop

-- set $\text{leader}(i) := \text{node } s$

-- for each arc (i, j) in G :

-- if j not yet explored

-- $\text{DFS}(G, j)$

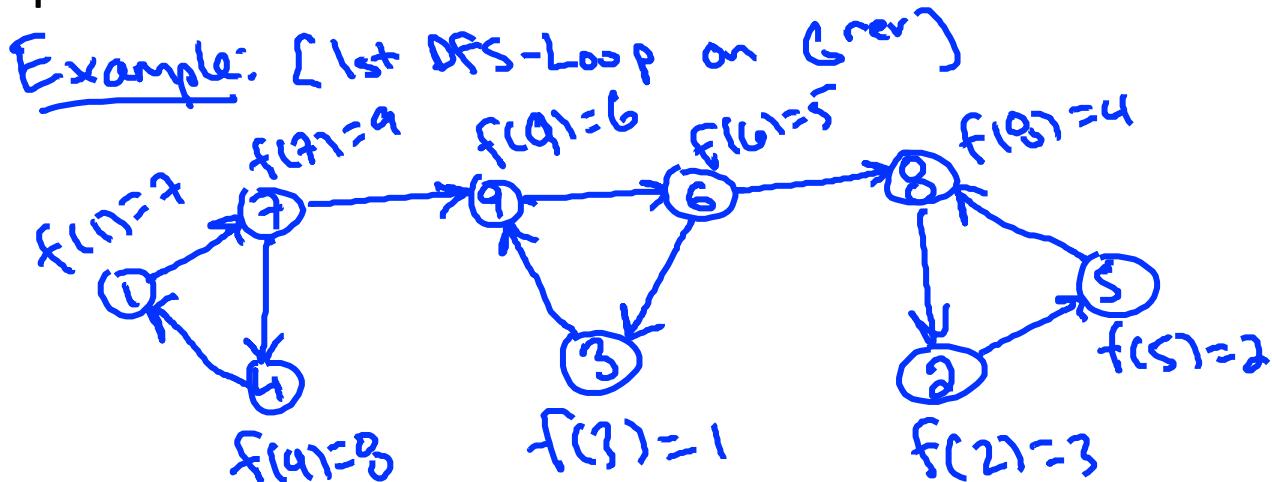
-- $t++$

-- set $f(i) := t$

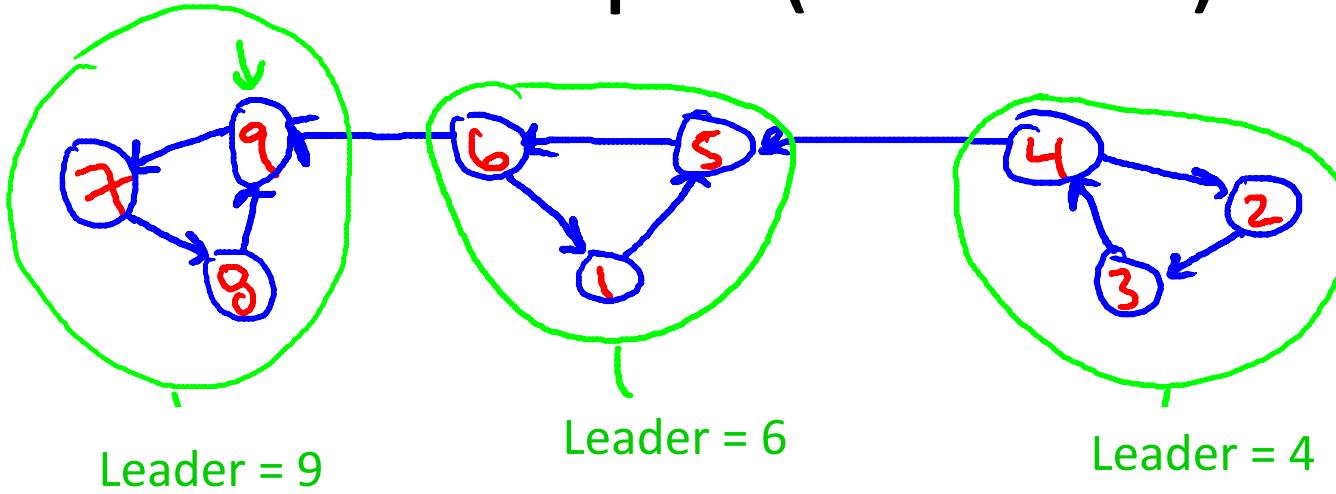
i's finishing
time

Only one of the following is a possible set of finishing times for the nodes 1,2,3,...,9, respectively, when the DFS-Loop subroutine is executed on the graph below. Which is it?

- 9,8,7,6,5,4,3,2,1
- 1,7,4,9,6,3,8,2,5
- 1,7,9,6,8,2,5,3,4
- 7,3,1,8,2,5,9,4,6



Example (2nd Pass)



Running Time : $2 * \text{DFS} = O(m+n)$

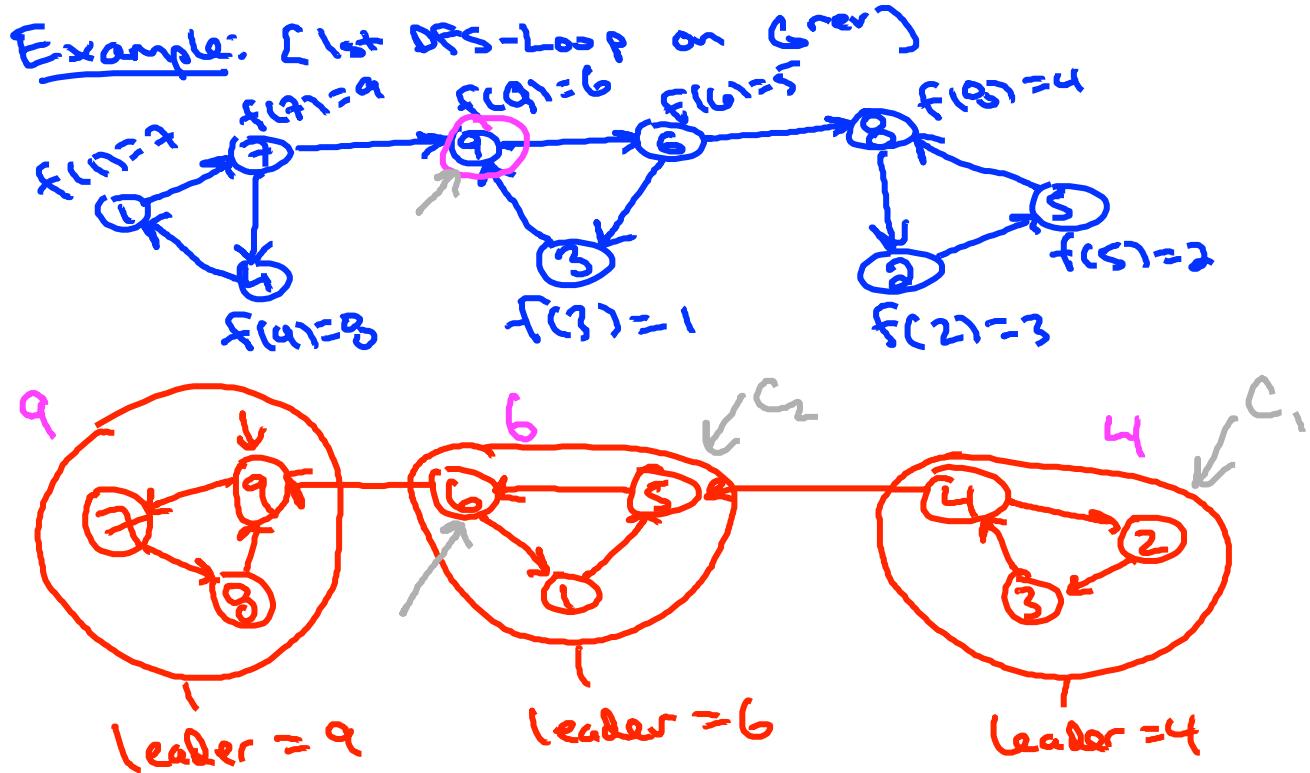


Design and Analysis
of Algorithms I

Graph Primitives

Correctness of
Kosaraju's Algorithm

Example Recap

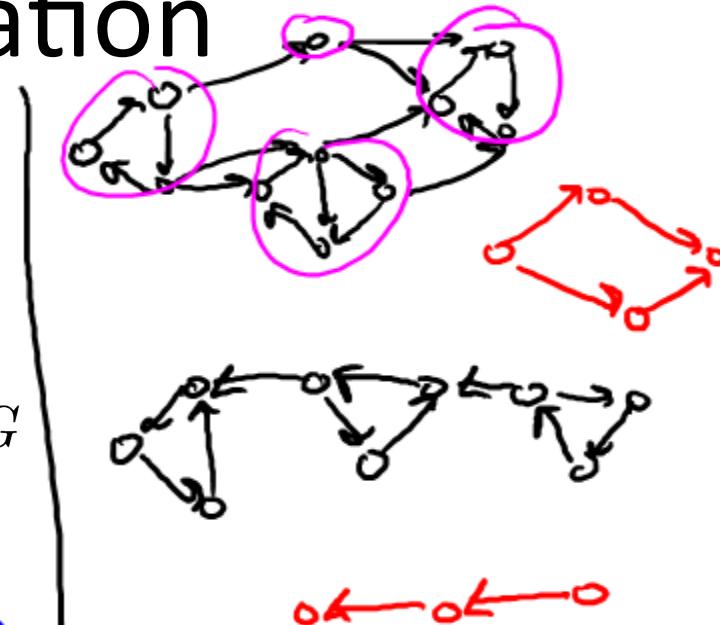
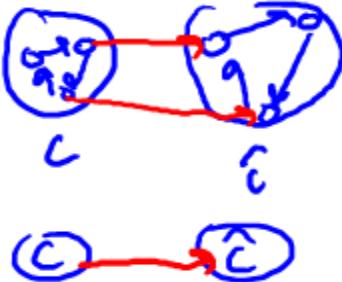


Observation

Claim : the SCCs of a directed graph G induce an acyclic “meta-graph”:

- meta-nodes = the SCCs C_1, \dots, C_k of G
- \exists arc $C \rightarrow \hat{C} \iff \exists$ arc $\square(i, j) \in G$
with $i \in C, j \in \hat{C}$

Why acyclic ? : a cycle of SCCs would collapse into one.

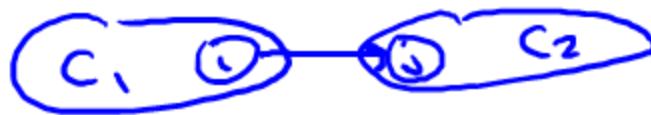


What how are the SCC of the original graph G and its reversal $G^{\uparrow rev}$ related?

- In general, they are unrelated.
- Every SCC of G is contained in an SCC of $G^{\uparrow rev}$, but the converse need not hold.
- Every SCC of $G^{\uparrow rev}$ is contained in an SCC of G , but the converse need not hold.
- They are exactly the same.

Key Lemma

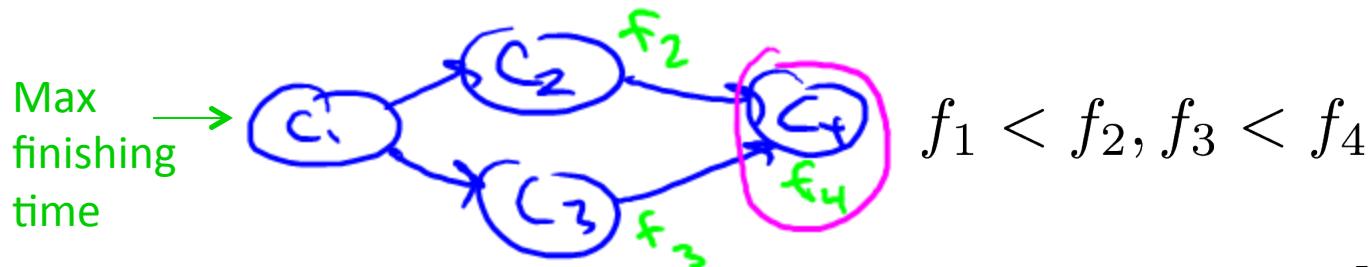
Lemma : consider two “adjacent” SCCs in G:



Let $f(v)$ = finishing times of DFS-Loop in Grev

Then : $\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v)$

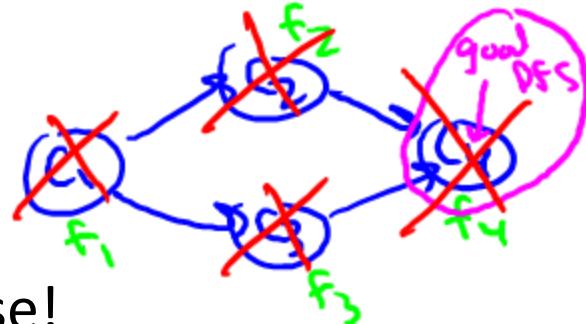
Corollary : maximum f-value of G must lie in a “sink SCC”



Correctness Intuition

(see notes for formal proof)

By Corollary : 2nd pass of DFS-Loop begins somewhere in a sink SCC C^* .



⇒ First call to DFS discovers C^* and nothing else!

⇒ Rest of DFS-Loop like recursing on G with C^* deleted

[starts in a sink node of $G - C^*$]

⇒ successive calls to $\text{DFS}(G, i)$ “peel off” the SCCs one by one

[in reverse topological order of the “meta-graph” of SCCs]

Proof of Key Lemma

In Grev:



Let $v = 1^{\text{st}}$ node of $C_1 \cup C_2$

Still SCCs (of Grev)
[by Quiz]

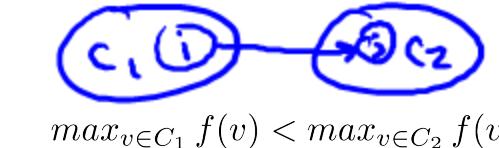
reached by 1st pass of DFS-Loop (on Grev)

Case 1 [$v \in C_1$] : all of C_1 explored before C_2 ever reached.

Reason : no paths from C_1 to C_2 (since meta-graph is acyclic)

\Rightarrow All f-values in C_1 less than all f-values in C_2

Case 2 [$v \in C_2$] : $\text{DFS}(\text{Grev}, v)$ won't finish until all of $C_1 \cup C_2$ completely explored $\Rightarrow f(v) > f(w)$ for all w in C_1



Q.E.D.



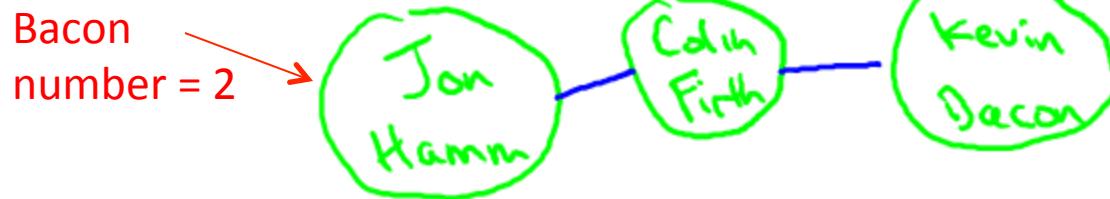
Design and Analysis
of Algorithms I

Graph Primitives

Introduction to Graph Search

A Few Motivations

1. Check if a network is connected (can get to anywhere from anywhere else)



2. Driving directions
3. Formulate a plan [e.g., how to fill in a Sudoku puzzle]
-- nodes = a partially completed puzzle -- arcs = filling in one new sequence
4. Compute the “pieces” (or “components”) of a graph
-- clustering, structure of the Web graph, etc.

Generic Graph Search

- Goals : 1) find everything findable from a given start vertex
2) don't explore anything twice

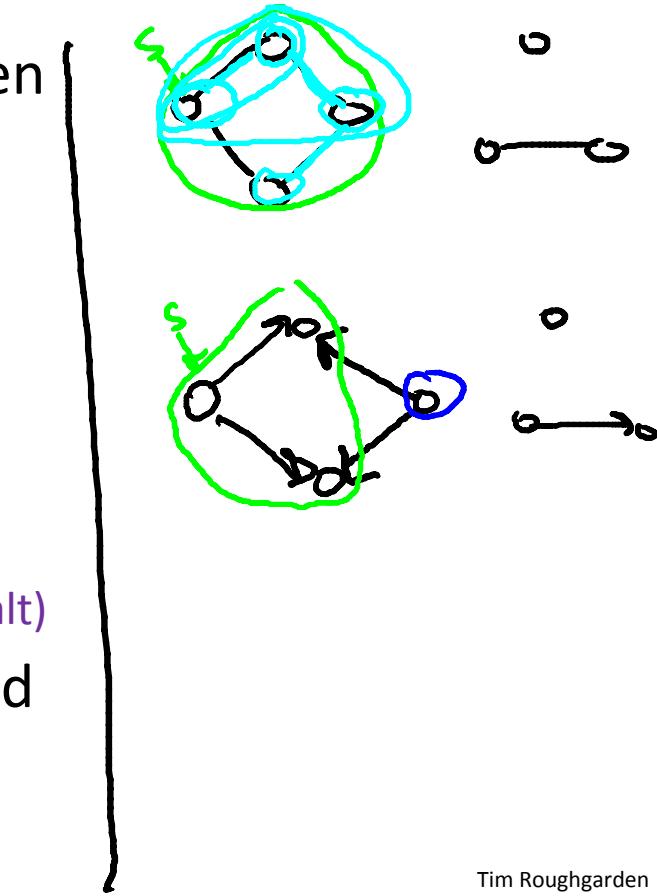
Goal:
 $O(m+n)$ time

Generic Algorithm (given graph G , vertex s)

-- initially s explored, all other vertices unexplored

-- while possible : (if none, halt)

- choose an edge (u,v) with u explored and v unexplored
- mark v explored



Generic Graph Search (con'd)

Claim : at end of the algorithm, v explored \Leftrightarrow G has a path from (G undirected or directed) s to v

Proof : (\Rightarrow) easy induction on number of iterations (you check)

(\Leftarrow) By contradiction. Suppose G has a path P from s to v:



But v unexplored at end of the algorithm. Then there exists an edge (u,x) in P with u explored and x unexplored.

But then algorithm would not have terminated, contradiction.

Q.E.D.

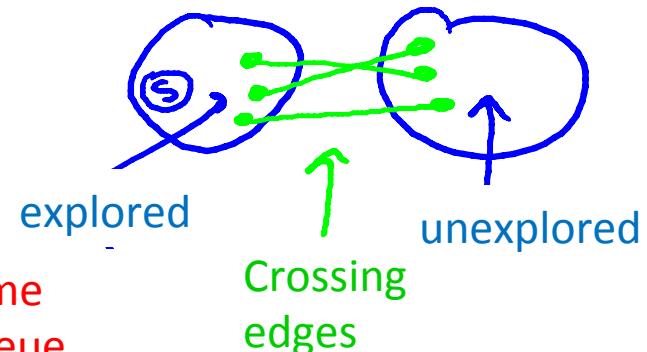
BFS vs. DFS

Note : how to choose among the possibly many “frontier” edges ?

Breadth-First Search (BFS)

- explored nodes in “layers”
- can compute shortest paths
- can compute connected components of an undirected graph

$O(m+n)$ time
using a queue
(FIFO)



Depth-First Search (DFS)

$O(m+n)$ time using a stack (LIFO)
(or via recursion)

- explore aggressively like a maze, backtrack only when necessary
- compute topological ordering of a directed acyclic graph
- compute connected components in directed graphs



Graph Primitives

Structure of the Web

Design and Analysis
of Algorithms I

The Web graph

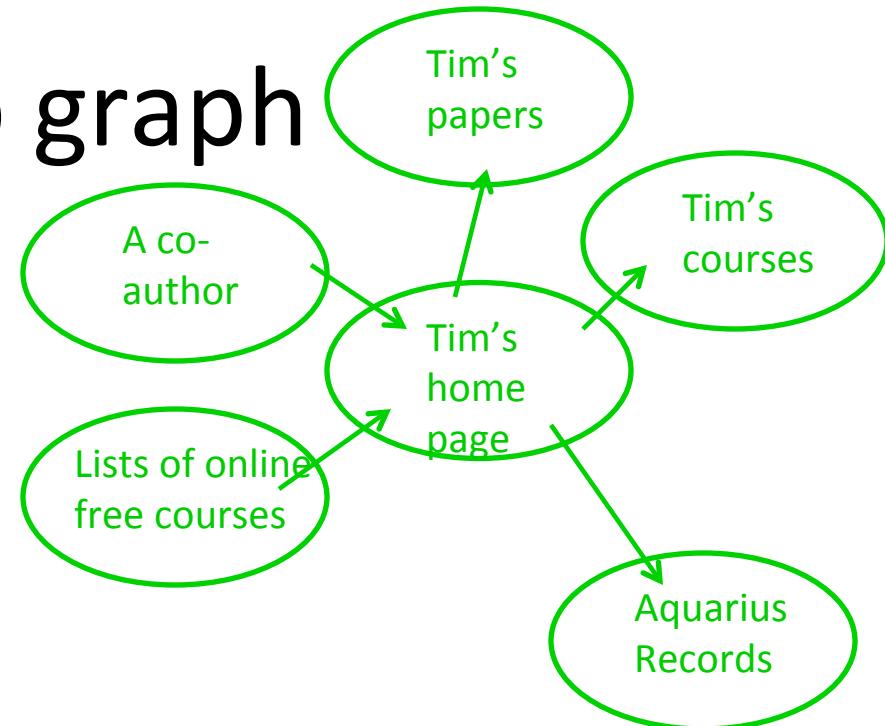
- Vertices = Web pages
- (directed) edges = hyperlinks

Question : what does the web graph
look like ?

(assume you've already “crawled” it)

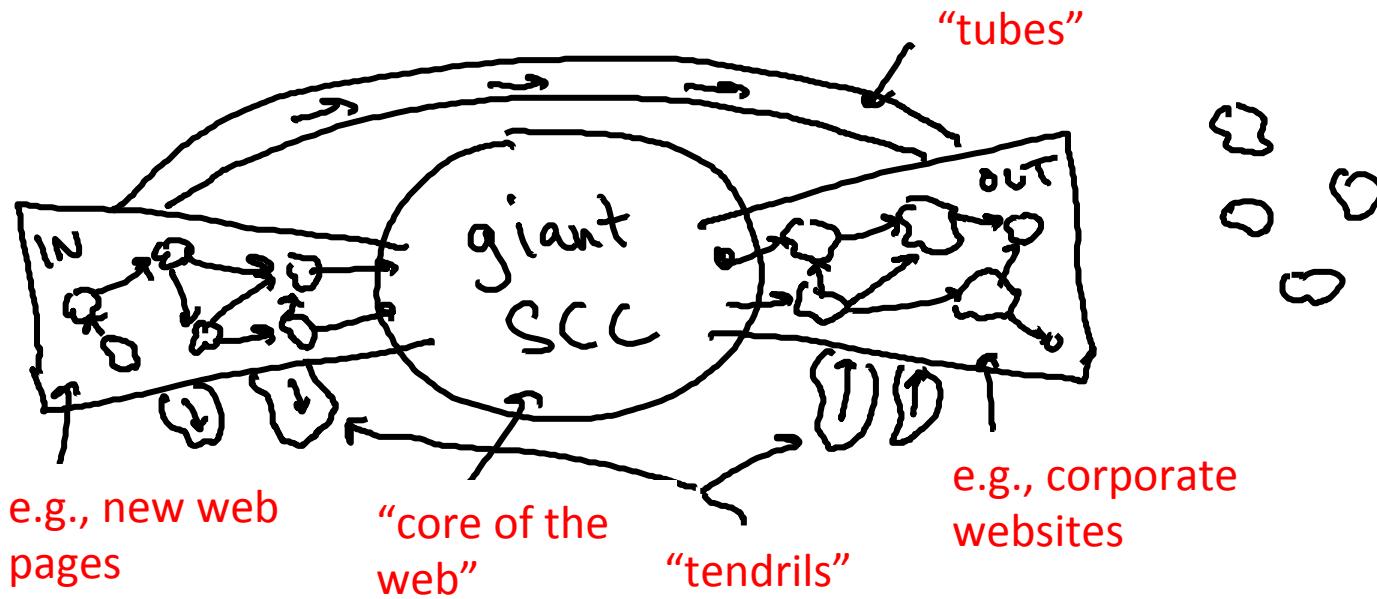
Size : ~ 200 million nodes, ~ 1 billion edges

Reference : [Broder et al WWW 2000]
computed the SCCs of the Web graph.



(pre map-reduce/hadoop)

The Bow Tie

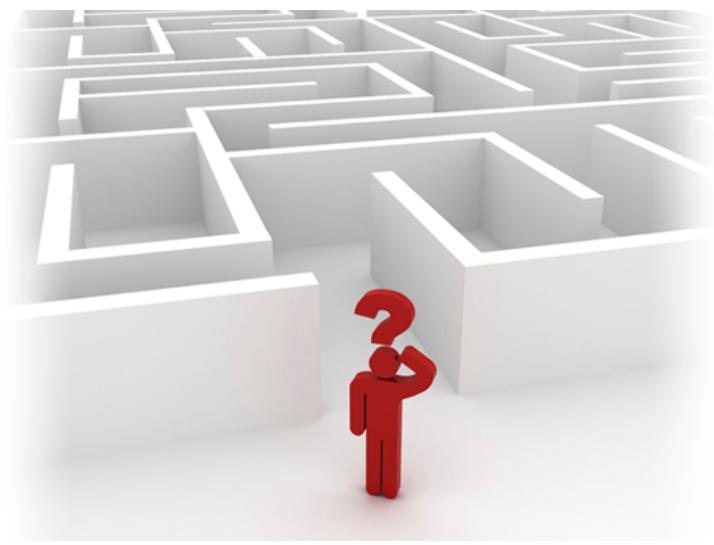


Main Findings

1. All 4 parts (giant, IN, OUT, tubes + tendrils) have roughly the same size
2. Within CORE, very well connected (has the “small world” property) [Milgram]
3. Outside, surprisingly poorly connected

Modern Web Research

1. **Temporal aspects** – how is the web graph evolving over time ?
 2. **Informational aspects** – how does new information propagate throughout the Web (or blogosphere, or Twitter, etc.)
 3. **Finer-grained structure** – how to define and compute “communities” in information and social networks ?
- Recommended Reading :** Easley + Kleinberg, “Networks, Crowds, & Markets”



Design and Analysis
of Algorithms I

QuickSort

Analysis I: A Decomposition Principle

Necessary Background

Assumption: you know and remember (finite) sample spaces, random variables, expectation, linearity of expectation. For review:

- Probability Review I (video)
- Lehman-Leighton notes (free PDF)
- Wikibook on Discrete Probability

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$.

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm
(i.e., the pivot choices)

Preliminaries

Fix input array A of length n

Sample Space Ω = all possible outcomes of random choices in QuickSort (i.e., pivot sequences)

Key Random Variable : for $\sigma \in \Omega$

$C(\sigma)$ = # of comparisons between two input elements made by QuickSort (given random choices σ)

Lemma: running time of QuickSort dominated by comparisons.

Remaining goal : $E[C] = O(n \log(n))$

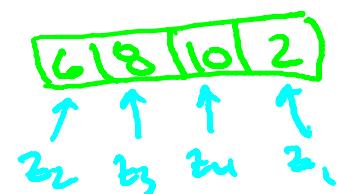
There exist constant c s.t. for all $\sigma \in \Omega$, $RT(\sigma) \leq c \cdot C(\sigma)$
(see notes)

Tim Roughgarden

Building Blocks

Note can't apply Master Method [random, unbalanced subproblems]

[A = final input array]



Notation : z_i = i^{th} smallest element of A

For $\sigma \in \Omega$, indices $i < j$

$X_{ij}(\sigma)$ = # of times z_i, z_j get compared in
QuickSort with pivot sequence σ

Fix two elements of the input array. How many times can these two elements get compared with each other during the execution of QuickSort?

1

0 or 1

0, 1, or 2

Any integer between 0 and $n - 1$

Reason : two elements compared only when one is the pivot, which is excluded from future recursive calls.

Thus : each X_{ij} is an “indicator” (i.e., 0-1) random variable

A Decomposition Approach

So : $C(\sigma) = \# \text{ of comparisons between input elements}$

$X_{ij}(\sigma) = \# \text{ of comparisons between } z_i \text{ and } z_j$

Thus : $\forall \sigma, C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}(\sigma)$

By Linearity of Expectation : $E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}]$

Since $E[X_{ij}] = 0 \cdot Pr[X_{ij} = 0] + 1 \cdot Pr[X_{ij} = 1] = Pr[X_{ij} = 1]$

Thus : $E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n Pr[z_i, z_j \text{ get compared}] \quad (*)$

Next video

A General Decomposition Principle

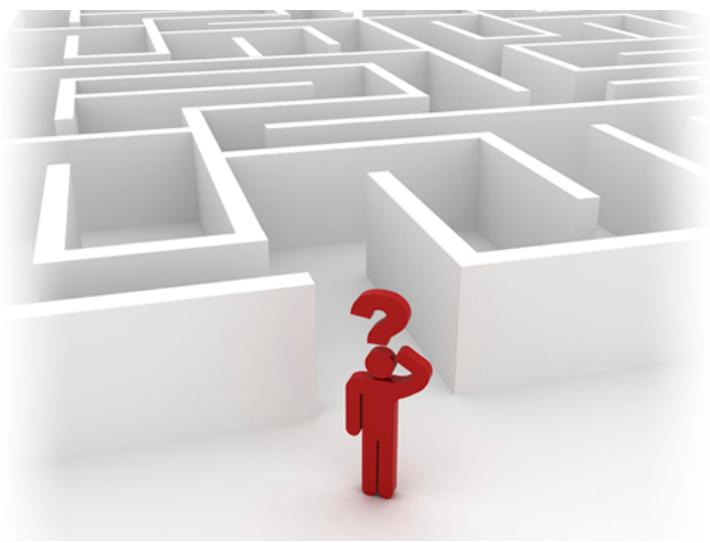
1. Identify random variable Y that you really care about
2. Express Y as sum of indicator random variables :

$$Y = \sum_{l=1}^m X_e$$

3. Apply Linearity of expectation :

$$E[Y] = \sum_{l=1}^m Pr[X_e = 1]$$

“just” need to understand these!



Design and Analysis
of Algorithms I

QuickSort

Analysis II: The Key Insight

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$.

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm
(i.e., the pivot choices)

The Story So Far

$C(\sigma)$ = # of comparisons between input elements

$X_{ij}(\sigma)$ = # of comparisons between z_i and z_j

$i^{\text{th}}, j^{\text{th}}$ smallest entries in array

Recall : $E[C] = \sum_{i=1}^{n-1} \sum_{k=i+1}^n Pr[X_{ij} = 1]$

$= Pr[z_i z_j \text{ get compared}]$

Key Claim : for all $i < j$, $Pr[z_i, z_j \text{ get compared}] = 2/(j-i+1)$

Proof of Key Claim

Fix z_i, z_j with $i < j$

Consider the set $z_i, z_{i+1}, \dots, z_{j-1}, z_j$

$$\Pr[z_i, z_j \text{ get compared}] = \frac{2}{(j-i+1)}$$

Inductively : as long as none of these are chosen as a pivot, all are passed to the same recursive call.

Consider the first among $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ that gets chosen as a pivot.

1. If z_i or z_j gets chosen first, then z_i and z_j get compared
2. If one of z_{i+1}, \dots, z_{j-1} gets chosen first then z_i and z_j are never compared [split into different recursive calls]

KEY
INSIGHT

Tim Roughgarden

Proof of Key Claim (con'd)

1. z_i or z_j gets chosen first \Rightarrow they get compared
2. one of z_{i+1}, \dots, z_{j-1} gets chosen first $\Rightarrow z_i, z_j$ never compared

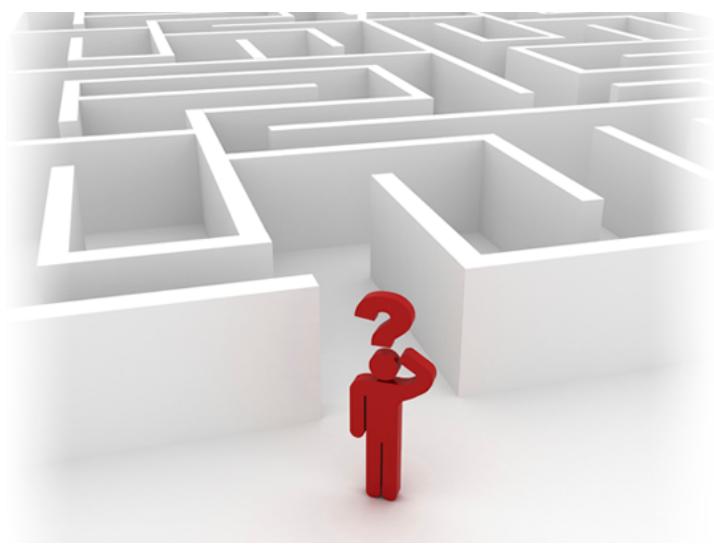
Note : Since pivots always chosen uniformly at random, each of $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ is equally likely to be the first

$$\Rightarrow \Pr[z_i, z_j \text{ get compared}] = \frac{2}{(j-i+1)}$$

Choices that lead to
 case (1) Total # of choices

$$\text{So: } E[C] = \sum_{i=1}^{n-1} \sum_{j=1}^n \frac{2}{j - i + 1}$$

[Still need to show
this is $O(n \log n)$]



Design and Analysis
of Algorithms I

QuickSort

Analysis III: Final Calculations

Average Running Time of QuickSort

QuickSort Theorem : for every input array of length n , the average running time of QuickSort (with random pivots) is $O(n \log(n))$

Note : holds for every input. [no assumptions on the data]

- recall our guiding principles !
- “average” is over random choices made by the algorithm (i.e., pivot choices)

The Story So Far

$$E[C] = 2 \sum_{i=1}^{n-1} \sum_{j=1}^n \frac{1}{j - i + 1}$$

<= n choices
for i
 $\theta(n^2)$ terms
How big can this be ?

(*)

Note : for each fixed i , the inner sum is

$$\sum_{j=i+1}^n \frac{1}{j - i + 1} = 1/2 + 1/3 + \dots$$

$$So \quad E[C] \leq 2 \cdot n \cdot \sum_{k=2}^n \frac{1}{k}$$

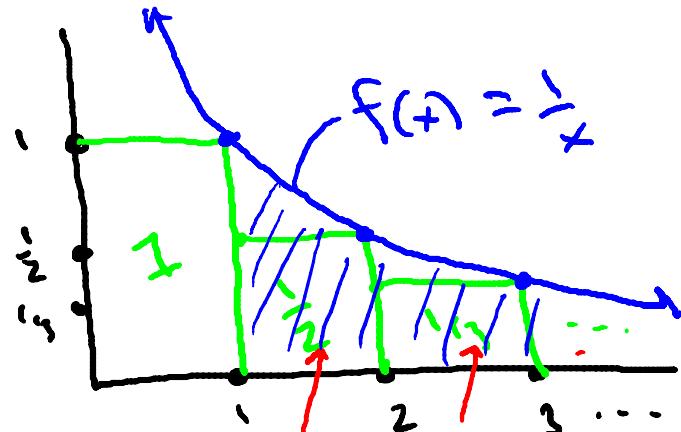
Claim : this is $<= \ln(n)$

Completing the Proof

$$E[C] \leq 2 \cdot n \cdot \sum_{k=2}^n \frac{1}{k}$$

Claim $\sum_{k=2}^n \frac{1}{k} \leq \ln n$

Proof of Claim



So :
 $E[C] \leq 2n \ln n$
Q.E.D.

$$So \quad \sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx$$

$$= \ln x \Big|_1^n$$

$$= \ln n - \ln 1$$

$$= \ln n \quad \text{Q.E.D. (CLAIM)}$$



Probability Review

Part I

Design and Analysis
of Algorithms I

Topics Covered

- Sample spaces
- Events
- Random variables
- Expectation
- Linearity of Expectation

See also:

- Lehman-Leighton notes (free PDF)
- Wikibook on Discrete Probability

Concept #1 – Sample Spaces

Sample Space Ω : “all possible outcomes”
[in algorithms, Ω is usually finite]

Also : each outcome $i \in \Omega$ has a probability $p(i) \geq 0$

Constraint : $\sum_{i \in \Omega} p(i) = 1$

Example #1 : Rolling 2 dice. $\Omega = \{(1,1), (2,1), (3,1), \dots, (5,6), (6,6)\}$

Example #2 : Choosing a random pivot in outer QuickSort call.

$\Omega = \{1, 2, 3, \dots, n\}$ (index of pivot) and $p(i) = 1/n$ for all $i \in \Omega$

Concept #2 – Events

An event is a subset $S \subseteq \Omega$

The probability of an event S is $\sum_{i \in S} p(i)$

Consider the event (i.e., the subset of outcomes for which) “the sum of the two dice is 7”. What is the probability of this event?

$$S = \{(1,6), (2,5), (3,4), (4,3), (5,2), (6,1)\}$$

1/36

1/12

$$\Pr[S] = 6/36 = 1/6$$

1/6

1/2

Consider the event (i.e., the subset of outcomes for which) “the chosen pivot gives a 25-75 split of better”. What is the probability of this event?

$1/n$

$S = \{(n/4+1)^{\text{th}} \text{ smallest element}, \dots, (3n/4)^{\text{th}} \text{ smallest element}\}$

$1/4$

$\Pr[S] = (n/2)/n = 1/2$

$1/2$

$3/4$

Concept #2 – Events

An event is a subset

The probability of an event S is

Ex#1 : sum of dice = 7. $S = \{(1,1), (2,1), (3,1), \dots, (5,6), (6,6)\}$

$$\Pr[S] = 6/36 = 1/6$$

Ex#2 : pivot gives 25-75 split or better.

$S = \{(\lfloor n/4 + 1 \rfloor)^{\text{th}} \text{ smallest element}, \dots, (\lfloor 3n/4 \rfloor)^{\text{th}} \text{ smallest element}\}$

$$\Pr[S] = (n/2)/n = 1/2$$

Concept #3 - Random Variables

A Random Variable X is a real-valued function

$$X : \Omega \rightarrow \mathbb{R}$$

Ex#1 : Sum of the two dice

Ex#2 : Size of subarray passed to 1st recursive call.

Concept #4 - Expectation

Let $X : \Omega \rightarrow \mathbb{R}$ be a random variable.

The expectation $E[X]$ of X = average value of X

$$= \sum_{i \in \Omega} X(i) \cdot p(i)$$

What is the expectation of the sum of two dice?

6.5

7

7.5

8

Which of the following is closest to the expectation of the size of the subarray passed to the first recursive call in QuickSort?

Let X = subarray size

$n/4$

$n/3$

$n/2$

$3n/4$

$$\begin{aligned} \text{Then } E[X] &= (1/n)*0 + (1/n)*2 + \dots + (1/n)*(n-1) \\ &= (n-1)/2 \end{aligned}$$

Concept #4 - Expectation

Let $X : \Omega \rightarrow \mathbb{R}$ be a random variable.

The expectation $E[X]$ of X = average value of X

$$= \sum_{i \in \Omega} X(i) \cdot p(i)$$

Ex#1 : Sum of the two dice, $E[X] = 7$

Ex#2 : Size of subarray passed to 1st recursive call.

$$E[X] = (n-1)/2$$

Concept #5 – Linearity of Expectation

Claim [LIN EXP] : Let X_1, \dots, X_n be random variables defined on Ω . Then :

$$E\left[\sum_{j=1}^n X_j\right] = \sum_{j=1}^n E[X_j]$$

Ex#1 : if X_1, X_2 = the two dice, then

$$E[X_j] = (1/6)(1+2+3+4+5+6) = 3.5$$

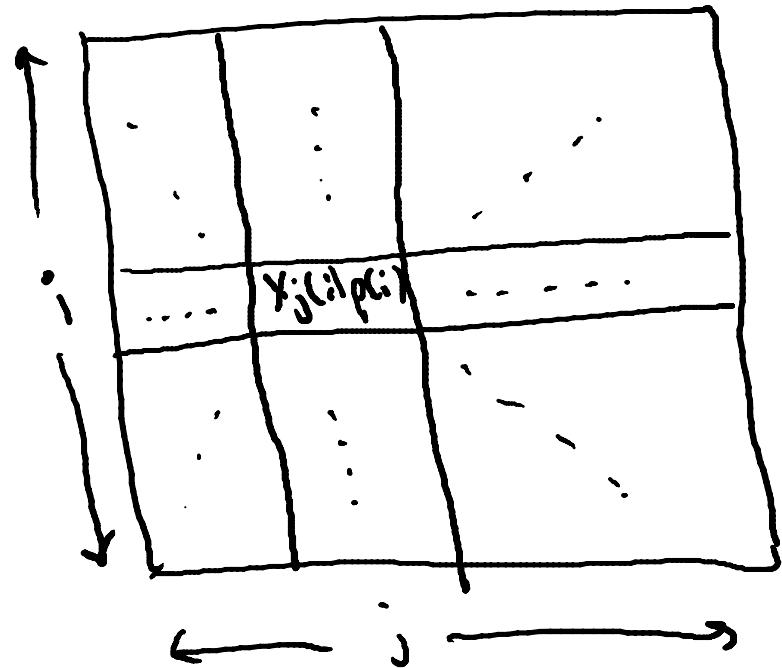
CRUCIALLY:
HOLDS EVEN WHEN
 X_j 's ARE NOT
INDEPENDENT!
[WOULD FAIL IF
REPLACE SUMS WITH
PRODUCTS]

By LIN EXP : $E[X_1 + X_2] = E[X_1] + E[X_2] = 3.5 + 3.5 = 7$

Linearity of Expectation (Proof)

$$\begin{aligned}\sum_{j=1}^n E[X_j] &= \sum_{j=1}^n \sum_{i \in \Omega} X_j(i)p(i) \\&= \sum_{i \in \Omega} \sum_{j=1}^n X_j(i)p(i) \\&= \sum_{i \in \Omega} p(i) \sum_{j=1}^n X_j(i) \\&= E\left[\sum_{j=1}^n X_j\right]\end{aligned}$$

Q.E.D.



Example: Load Balancing

Problem : need to assign n processes to n servers.

Proposed Solution : assign each process to a random server

Question : what is the expected number of processes assigned to a server ?

Load Balancing Solution

Sample Space Ω = all n^n assignments of processes to servers, each equally likely.

Let Y = total number of processes assigned to the first server.

Goal : compute $E[Y]$

Let $X_j = \begin{cases} 1 & \text{if } j\text{th process assigned to first server} \\ 0 & \text{otherwise} \end{cases}$

“indicator random variable”

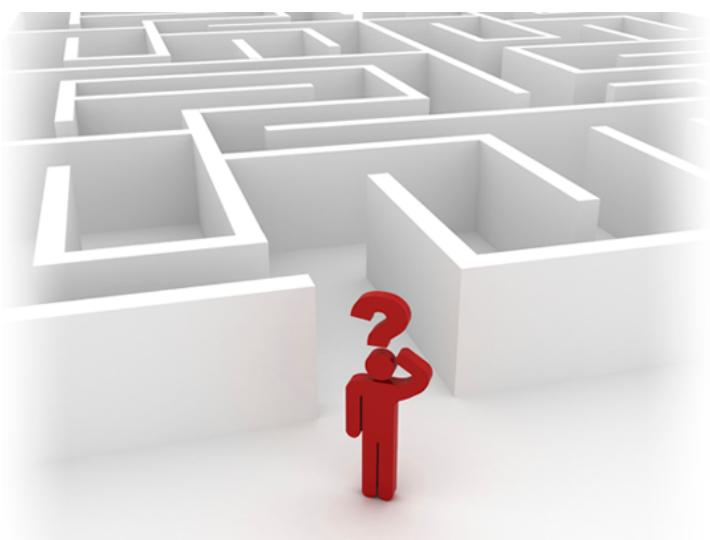
$$\text{Note } Y = \sum_{j=1}^n X_j$$

Load Balancing Solution (con'd)

We have

$$\begin{aligned}
 E[Y] &= E\left[\sum_{j=1}^n X_j\right] \\
 &= \sum_{j=1}^n E[X_j] \\
 &= \sum_{j=1}^n (Pr[X_j = 0] \cdot 0 + Pr[X_j = 1] \cdot 1) \\
 &= \sum_{j=1}^n \frac{1}{n} = 1
 \end{aligned}$$

0
 $\underbrace{\Pr[X_j = 1]}$
 $\equiv 1/n$ (servers chosen
uniformly at random)



Design and Analysis
of Algorithms I

Probability Review

Part II

Topics Covered

- Conditional probability
- Independence of events and random variables

See also:

- Lehman-Leighton notes (free PDF)
- Wikibook on Discrete Probability

Concept #1 – Sample Spaces

Sample Space Ω : “all possible outcomes”

[in algorithms, Ω is usually finite]

Also : each outcome $i \in \Omega$ has a probability $p(i) \geq 0$

Constraint : $\sum_{i \in \Omega} p(i) = 1$

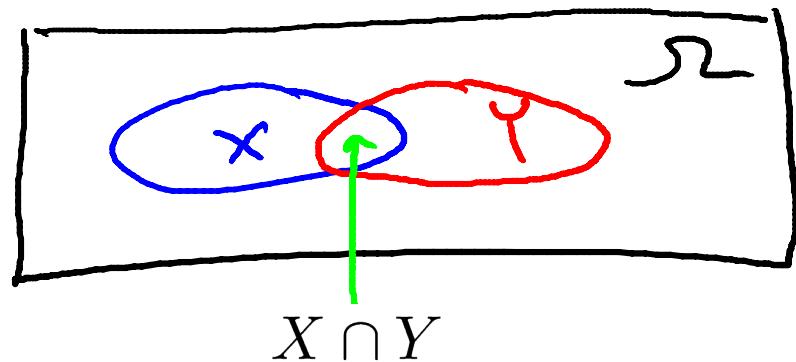
An event is a subset $S \subseteq \Omega$

The probability of an event S is $\sum_{i \in S} p(i)$

Tim Roughgarden

Concept #6 – Conditional Probability

Let $X, Y \subseteq \Omega$ be events.



Then $Pr[X|Y] = \frac{Pr[X \cap Y]}{Pr[Y]}$
("X given Y")

Suppose you roll two fair dice. What is the probability that at least one die is a 1, given that the sum of the two dice is 7?

- $1/36$
- $1/6$
- $1/3$
- $1/2$

$X = \text{at least one die is a } 1$

$Y = \text{sum of two dice} = 7$

$$= \{(1,6), (2,5), (3,4), (4,3), (5,2), (6,1)\}$$

$$\Rightarrow X \cap Y = \{(1, 6), (6, 1)\}$$

$$Pr[X|Y] = \frac{Pr[X \cap Y]}{Pr[Y]} = \frac{(2/36)}{(6/36)} = \frac{1}{3}$$

Concept #7 – Independence (of Events)

Definition : Events $X, Y \subseteq \Omega$ are independent if (and only if) $Pr[X \cap Y] = Pr[X] \cdot Pr[Y]$

You check : this holds if and only if $Pr[X | Y] = Pr[X]$
 $\iff Pr[Y | X] = Pr[Y]$

WARNING : can be a very subtle concept.
(intuition is often incorrect!)

Independence (of Random Variables)

Definition : random variables A, B (both defined on Ω)
 are independent if and only if the events $\Pr[A=1]$, $\Pr[B=b]$ are
 independent for all a,b. [$\Leftrightarrow \Pr[A = a \text{ and } B = b] = \Pr[A=a] * \Pr[B=b]$]

Claim : if A,B are independent, then $E[AB] = E[A]*E[B]$

Proof :

$$\begin{aligned}
 E[AB] &= \sum_{a,b} (a \cdot b) \cdot \Pr[A = a \text{ and } B = b] \\
 &= \sum_{a,b} (a \cdot b) \cdot \Pr[A = a] \cdot \Pr[B = b] \quad (\text{Since A,B independent}) \\
 &= (\sum_a a \cdot \Pr[A = a]) (\sum_b b \cdot \Pr[B = b])
 \end{aligned}$$

Q.E.D.

Example

Let $X_1, X_2 \in \{0, 1\}$ be random, and $X_3 = X_1 \oplus \overset{\text{XOR}}{X_2}$

formally : $\Omega = \{000, 101, 011, 110\}$, each equally likely.

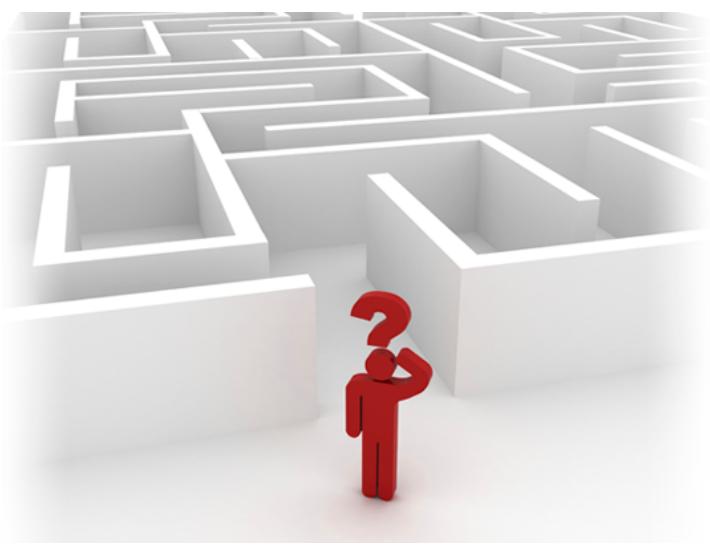
Claim : X_1 and X_3 are independent random variables (you check)

Claim : $X_1 X_3$ and X_2 are not independent random variables.

Proof : suffices to show that

$$E[X_1 X_2 X_3] \neq E[X_1 X_3] E[X_2]$$

$\Rightarrow 0$
 $= E[X_1] E[X_3] = 1/4$
Since X_1 and X_3 independent



Design and Analysis
of Algorithms I

Linear-Time Selection

Deterministic
Selection (Algorithm)

The Problem

Input : array A with n **distinct** numbers and a number

For simplicity

Output : i^{th} order statistic (i.e., i^{th} smallest element of input array)

Example : median.

($i = (n+1)/2$ for n odd,
 $i = n/2$ for n even)

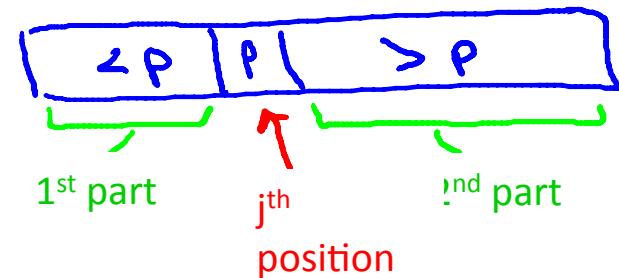


3rd order statistic

Randomized Selection

Rselect (array A, length n, order statistic i)

- 0) if $n = 1$ return $A[1]$
- 1) Choose pivot p from A uniformly at random
- 2) Partition A around p
let $j = \text{new index of } p$
- 3) If $j = i$, return p
- 4) If $j > i$, return Rselect(1^{st} part of A , $j-1$, i)
- 5) [if $j < i$] return Rselect (2^{nd} part of A , $n-j$, $i-j$)



Guaranteeing a Good Pivot

Recall : “best” pivot = the median ! (seems circular!)

Goal : find pivot guaranteed to be pretty good.

Key Idea : use “median of medians”!

A Deterministic ChoosePivot

ChoosePivot(A,n)

- logically break A into $n/5$ groups of size 5 each
- sort each group (e.g., using Merge Sort)
- copy $n/5$ medians (i.e., middle element of each sorted group)
into new array C
- recursively compute median of C (!)
- return this as pivot

The DSelect Algorithm

DSelect(array A, length n, order statistic i)

1. Break A into groups of 5, sort each group
2. C = the $n/5$ “middle elements”
3. $p = \text{DSelect}(C, n/5, n/10)$ [recursively computes median of C]
4. Partition A around p
5. If $j = i$ return p
6. If $j < i$ return DSelect(1^{st} part of A, $j-1$, i)
7. [else if $j > i$] return DSelect(2^{nd} part of A, $n-j$, $i-j$)

ChoosePivot

Same as
before

How many recursive calls does DSelect make?

0

1

2

3

Running Time of DSelect

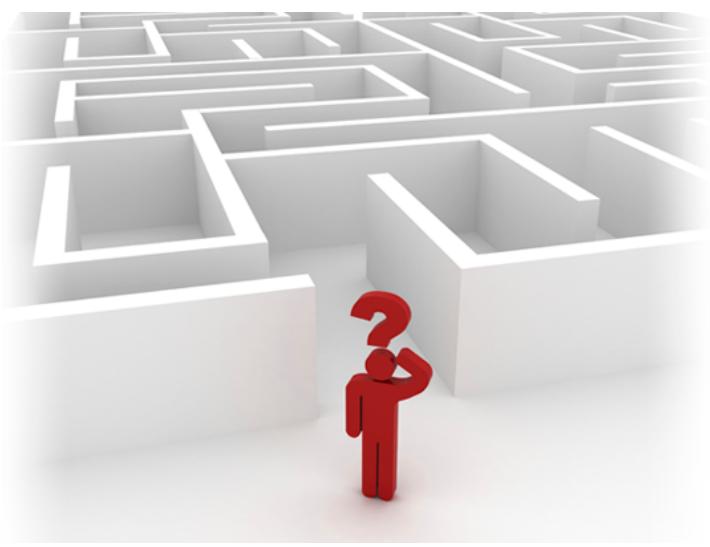
Dselect Theorem : for every input array of length n ,
Dselect runs in $O(n)$ time.

Warning : not as good as Rselect in practice

- 1) Worse constraints
- 2) not-in-place

History : from 1973

Blum – Floyd – Pratt – Rivest – Tarjan
(‘95) (‘78) (‘02) (‘86)



Design and Analysis
of Algorithms I

Linear-Time Selection

Deterministic Selection (Analysis)

The DSelect Algorithm

DSelect(array A, length n, order statistic i)

1. Break A into groups of 5, sort each group

2. C = the $n/5$ “middle elements”

3. $p = \text{DSelect}(C, n/5, n/10)$ [recursively computes median of C]

4. Partition A around p

5. If $j = i$ return p

6. If $j < i$ return DSelect(1^{st} part of A, $j-1$, i)

7. [else if $j > i$] return DSelect(2^{nd} part of A, $n-j$, $i-j$)

Choose
Pivot

Same as
before

What is the asymptotic running time of step 1 of the DSelect algorithm?

- $\theta(1)$
- $\theta(\log n)$
- $\theta(n)$
- $\theta(n \log n)$

Note : sorting an array with 5 elements takes
 ≤ 120 operations

[why 120 ? Take $m = 5$ in our $6m(\log_2 m + 1)$ bound for Merge Sort]

$$6 * 5 * (\log_2 5 + 1) \leq 120$$

≤ 3

of gaps ops per group

So : $\leq (n/5) * 120 = 24n = O(n)$ for all groups

The DSelect Algorithm

DSelect(array A, length n, order statistic i) $\theta(n)$

1. Break A into groups of 5, sort each group $\theta(n)$
2. C = the $n/5$ “middle elements” $\theta(n)$
3. p = DSelect(C, $n/5$, $n/10$) [recursively computes median of C]
4. Partition A around p $T\left(\frac{n}{5}\right)$
5. If $j = i$ return p $\theta(n)$
6. If $j < i$ return DSelect(1st part of A, $j-1$, i) $T(?)$
7. [else if $j > i$] return DSelect(2nd part of A, $n-j$, $i-j$)

Rough Recurrence

Let $T(n)$ = maximum running time of Dselect on an input array of length n .

There is a constant $c \geq 1$ such that :

1. $T(1) = 1$
2. $T(n) \leq c*n + T(n/5) + T(?)$

sorting the groups
partition

recursive
call in line 3

recursive call in
line 6 or 7

The Key Lemma

Key Lemma : 2nd recursive call (in line 6 or 7) guaranteed to be on an array of size $\leq 7n/10$ (roughly)

Upshot : can replace “?” by “7n/10”

Rough Proof : Let $k = n/5 = \# \text{ of groups}$
Let $x_i = i^{\text{th}}$ smallest of the k “middle elements”
[So pivot = $x_{k/2}$]

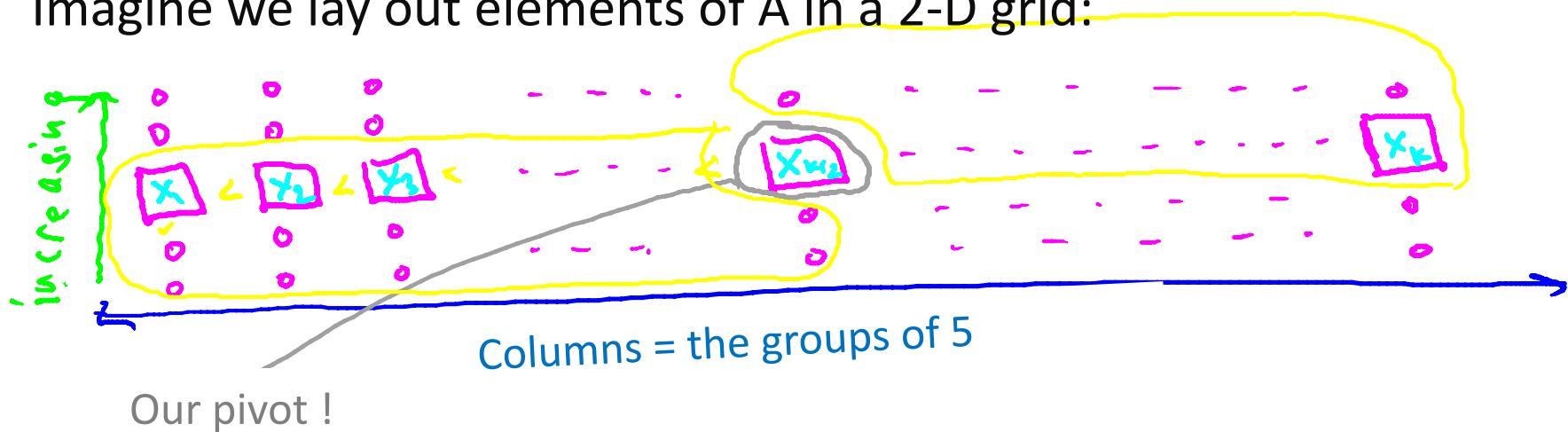
Goal : $\geq 30\%$ of input array smaller than $x_{k/2}$,
 $\geq 30\%$ is bigger

Tim Roughgarden

Rough Proof of Key Lemma

Thought Experiment :

Imagine we lay out elements of A in a 2-D grid:



Key point : $x_{k/2}$ bigger than 3 out of 5 (60%) of the elements in
~ 50% of the groups

=> bigger than 30% of A (similarly, smaller than 30% of A)

Example

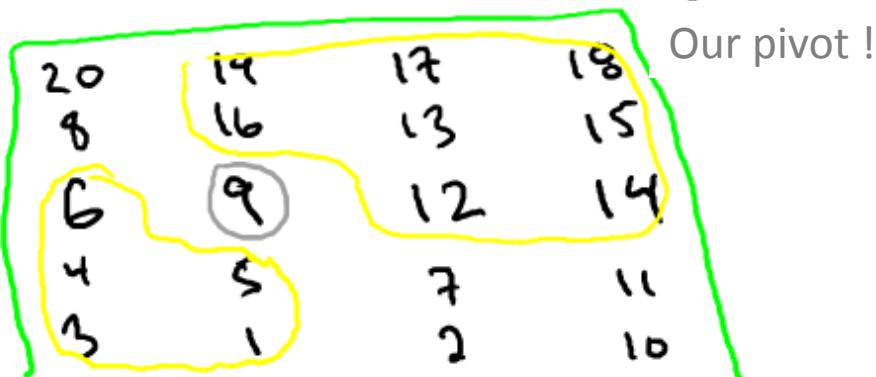
Input

7	2	17	12	13	8	20	4	6	3	19	1	9	5	16	10	15	18	14	11
---	---	----	----	----	---	----	---	---	---	----	---	---	---	----	----	----	----	----	----

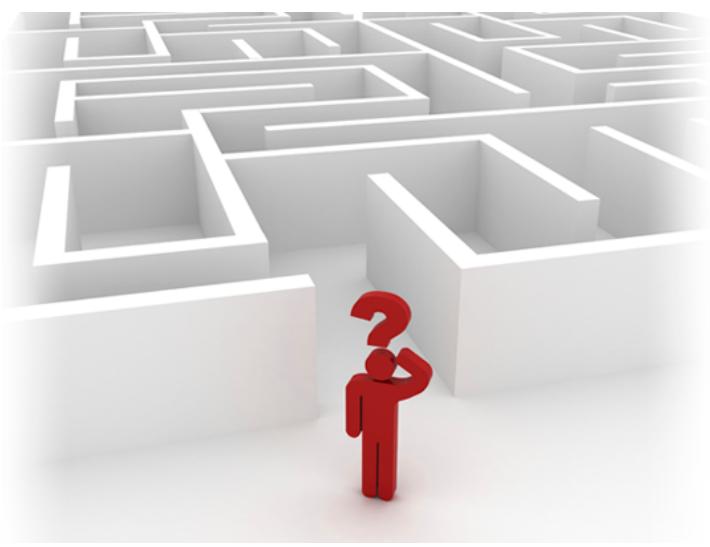
After
sorting
groups
of 5

2	7	12	13	17	3	4	6	8	20	1	5	9	16	19	10	11	14	15	18
---	---	----	----	----	---	---	---	---	----	---	---	---	----	----	----	----	----	----	----

The
grid :



Tim Roughgarden



Design and Analysis
of Algorithms I

Linear-Time Selection

Deterministic
Selection (Analysis II)

Rough Recurrence (Revisited)

Let $T(n)$ = maximum running time of Dselect on an input array of length n .

There is a constant $c \geq 1$ such that :

1. $T(1) = 1$
2. $T(n) \leq c*n + T(n/5) + T(\text{?})$

$\leq 7n/10$ by
Key Lemma

sorting the groups recursive recursive call in
partition call in line 3 line 6 or 7

Rough Recurrence (Revisited)

$$T(1) = 1, T(n) \leq cn + T(n/5) + T(7n/10)$$

Constant $c \geq 1$

Note : different-sized subproblems => can't use Master Method!

Strategy : “hope and check”

Hope : there is some constant a [independent of n]

Such that $T(n) \leq an$ for all $n \geq 1$

[if true, then $T(n) = O(n)$ and algorithm is linear time]

Analysis of Rough Recurrence

Claim : Let $a = 10c$

Then $T(n) \leq an$ for all $n \geq 1$

=> Dselect runs in
 $O(n)$ time

$$T(1) = 1 ; T(n) \leq cn + T(n/5) + T(7n/10)$$

Constant $c \geq 1$

Proof : by induction on n

Base case : $T(1) = 1 \leq a*1$ (since $a \geq 1$)

Inductive Step : $[n > 1]$

Inductive Hypothesis : $T(k) \leq ak \forall k < n$

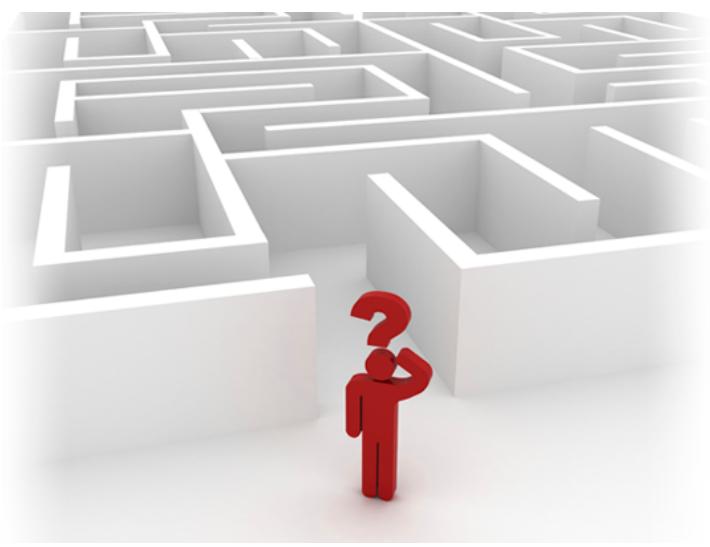
We have $T(n) \leq cn + T(n/5) + T(7n/10)$

$$\begin{aligned} &\stackrel{\text{GIVEN}}{\leq} cn + a(n/5) + a(7n/10) \\ &\stackrel{\text{IND HYP}}{=} n(c + 9a/10) = an \end{aligned}$$

Choice of a

Q.E.D.

Tim Roughgarden



Design and Analysis
of Algorithms I

Linear-Time Selection

Randomized Selection (Algorithm)

Prerequisites

Watch this after:

- QuickSort - Partitioning around a pivot
- QuickSort – Choosing a good pivot
- Probability Review, Part I

The Problem

Input : array A with n **distinct** numbers and a number

For simplicity

Output : i^{th} order statistic (i.e., i^{th} smallest element of input array)

Example : median.

($i = (n+1)/2$ for n odd,
 $i = n/2$ for n even)



3rd order statistic

Reduction to Sorting

O(nlog(n)) algorithm

- 1) Apply MergeSort
- 2) return i^{th} element of sorted array

Fact : can't sort any faster [see optional video]

Next : $O(n)$ time (randomized) by modifying Quick Sort.

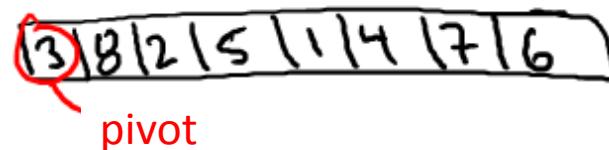
Optional Video : $O(n)$ time deterministic algorithm.

-- pivot = “median of medians” (warning : not practical)

Partitioning Around a Pivot

Key Idea : partition array around a pivot element.

-Pick element of array



-Rearrange array so that

- Left of pivot => less than pivot

- Right of pivot => greater than pivot



Note : puts pivot in its “rightful position”.

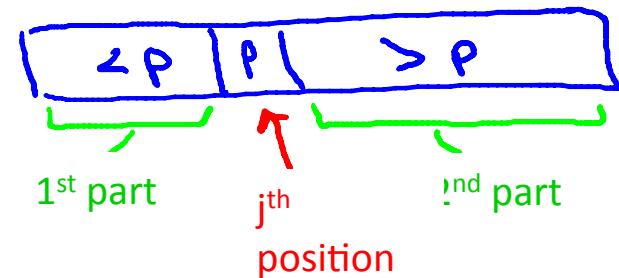
Suppose we are looking for the 5th order statistic in an input array of length 10. We partition the array, and the pivot winds up in the third position of the partitioned array. On which side of the pivot do we recurse, and what order statistic should we look for?

- The 3rd order statistic on the left side of the pivot.
- The 2nd order statistic on the right side of the pivot.
- The 5th order statistic on the right side of the pivot.
- Not enough information to answer question – we might need to recurse on the left or the right side of the pivot.

Randomized Selection

Rselect (array A, length n, order statistic i)

- 0) if $n = 1$ return $A[1]$
- 1) Choose pivot p from A uniformly at random
- 2) Partition A around p
let $j = \text{new index of } p$
- 3) If $j = i$, return p
- 4) If $j > i$, return Rselect(1^{st} part of A , $j-1$, i)
- 5) [if $j < i$] return Rselect (2^{nd} part of A , $n-j$, $i-j$)



Properties of RSelect

Claim : Rselect is correct (guaranteed to output ith order statistic)

Proof : by induction. [like in optional QuickSort video]

Running Time ? : depends on “quality” of the chosen pivots.

What is the running time of the RSelect algorithm if pivots are always chosen in the worst possible way?

- $\theta(n)$
- $\theta(n \log n)$
- $\theta(n^2)$
- $\theta(2^n)$

Example :

-- suppose $i = n/2$
-- suppose choose pivot = minimum
every time
 $\Rightarrow \Omega(n)$ time in each of $\Omega(n)$ recursive calls

Running Time of RSelect?

Running Time ? : depends on which pivots get chosen.
(could be as bad as $\theta(n^2)$)

Key : find pivot giving “balanced” split.

Best pivot: the median ! (but this is circular)

⇒ Would get recurrence $T(n) \leq T(n/2) + O(n)$

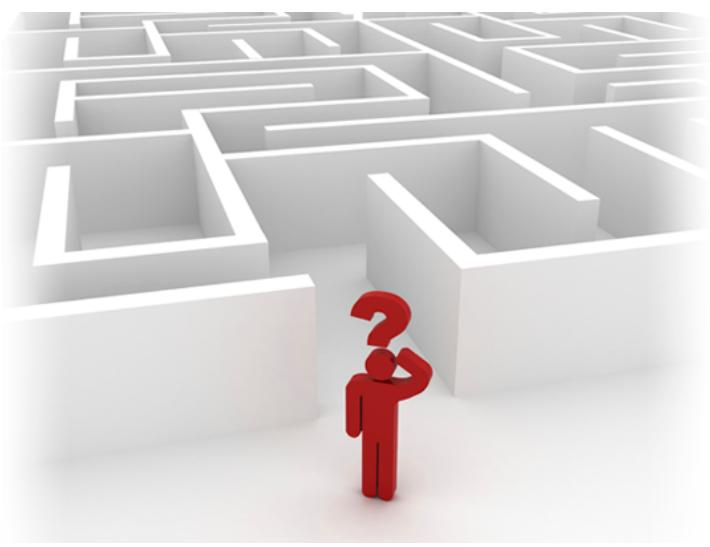
⇒ $T(n) = O(n)$ [case 2 of Master Method]

Hope : random pivot is “pretty good” “often enough”

Running Time of RSelect

Rselect Theorem : for every input array of length n , the average running time of Rselect is $O(n)$

- holds for every input [no assumptions on data]
- “average” is over random pivot choices made by the algorithm



Design and Analysis
of Algorithms I

Linear-Time Selection

Randomized Selection (Analysis)

Running Time of RSelect

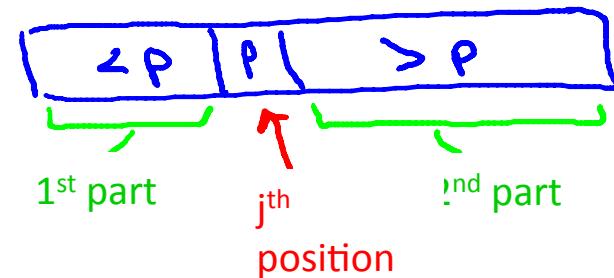
Rselect Theorem : for every input array of length n , the average running time of Rselect is $O(n)$

- holds for every input [no assumptions on data]
- “average” is over random pivot choices made by the algorithm

Randomized Selection

Rselect (array A, length n, order statistic i)

- 0) if $n = 1$ return $A[1]$
- 1) Choose pivot p from A uniformly at random
- 2) Partition A around p
let j = new index of p
- 3) If $j = i$, return p
- 4) If $j > i$, return Rselect(1^{st} part of A , $j-1$, i)
- 5) [if $j < i$] return Rselect (2^{nd} part of A , $n-j$, $i-j$)



Proof I: Tracking Progress via Phases

Note : Rselect uses $\leq cn$ operations outside of recursive call [for some constant $c > 0$] [from partitioning]

Notation : Rselect is in phase j if current array size between $(\frac{3}{4})^{j+1} \cdot n$ and $(\frac{3}{4})^j \cdot n$

- X_j = number of recursive calls during phase j

$$\text{Note : running time of RSelect} \leq \sum_{\text{phases } j} X_j \cdot \underbrace{c \cdot (\frac{3}{4})^j \cdot n}_{\begin{array}{l} \text{\# of phase } j \text{ subproblems} \\ \text{Work per phase } j \text{ subproblem} \end{array}}$$

=<= array size during phase j

Tim Roughgarden

Proof II: Reduction to Coin Flipping

$X_j = \# \text{ of recursive calls during phase } j$ → Size between $(\frac{3}{4})^{j+1} \cdot n$ and $(\frac{3}{4})^j \cdot n$

Note : if Rselect chooses a pivot giving a 25 – 75 split (or better) then current phase ends !
(new subarray length at most 75 % of old length)



Recall : probability of 25-75 split or better is 50%

So : $E[X_j] \leq$ expected number of times you need to flip a fair coin
to get one “heads”
(heads ~ good pivot, tails ~ bad pivot)

Tim Roughgarden

Proof III: Coin Flipping Analysis

Let N = number of coin flips until you get heads.
(a “geometric random variable”)

Note : $E[N] = 1 + (1/2)*E[N]$

1st coin flip Probability of tails # of further coin flips needed in this case

Solution : $E[N] = 2$ (Recall $E[X_j] \leq E[N]$)

Putting It All Together

Expected
running time of
RSelect

$$\leq E[cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^j X_j] \quad (*)$$

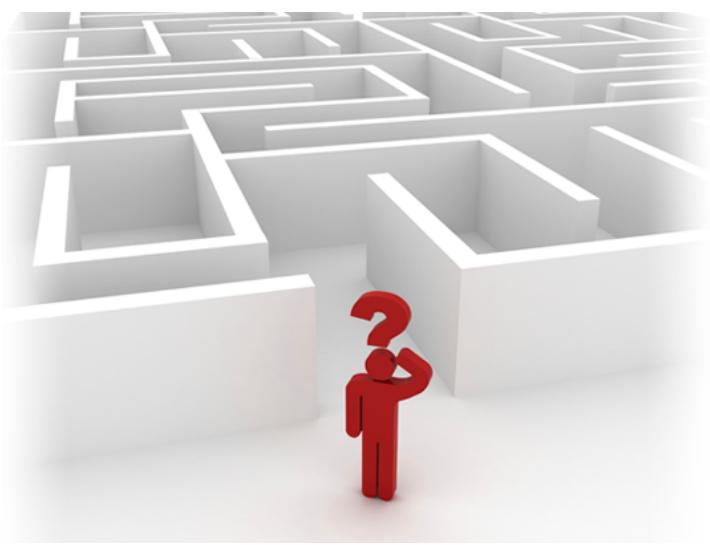
$$= cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^j E[X_j] \quad [\text{LIN EXP}]$$

= E[# of coin flips N] = 2

$$\leq 2cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^j$$

geometric sum,
 $\leq 1/(1-3/4) = 4$

$$\leq 8cn = O(n) \quad \text{Q.E.D.}$$



Design and Analysis
of Algorithms I

Linear-Time Selection

An $\Omega(n \log n)$
Sorting Lower Bound

A Sorting Lower Bound

Theorem : every “comparison-based” sorting algorithm has worst-case running time $\Omega(n \log n)$

[assume deterministic, but lower bound extends to randomized]

Comparison-Based Sort : accesses input array elements only via comparisons ~ “general purpose sorting method”

Examples : Merge Sort, Quick Sort, Heap Sort

Non Examples : Bucket Sort, Counting Sort, Radix Sort

Good for data from distributions → good for small integers → good for medium-size integers

Tim Roughgarden

Proof Idea

Fix a comparison-based sorting method and an array length n

⇒ Consider input arrays containing $\{1, 2, 3, \dots, n\}$ in some order.

⇒ $n!$ such inputs

Suppose algorithm always makes $\leq k$ comparisons to correctly sort these $n!$ inputs.

=> Across all $n!$ possible inputs, algorithm exhibits $\leq 2^k$ distinct executions → i.e., resolution of the comparisons

Tim Roughgarden

Proof Idea (con'd)

By the Pigeonhole Principle : if $2^k < n!$, execute identically on two distinct inputs => must get one of them incorrect.

So : Since method is correct,

$$\begin{aligned} 2^k &\geq n! \\ &\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \\ \Rightarrow k &\geq \frac{n}{2} \cdot \log_2 \frac{n}{2} = \Omega(n \log n) \end{aligned}$$



Contraction Algorithm

The Algorithm

Design and Analysis
of Algorithms I

The Minimum Cut Problem

- INPUT: An undirected graph $G = (V, E)$.
[Parallel  edges allowed]
[See other video for representation of the input]
- GOAL: Compute a cut with fewest number of crossing edges. (a min cut)

Random Contraction Algorithm

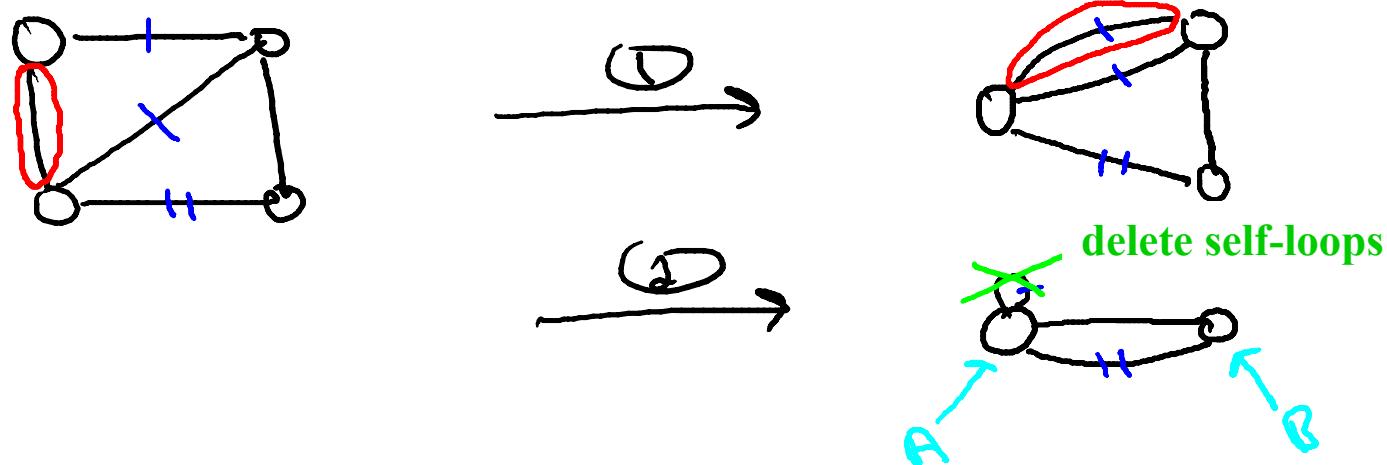
[due to Karger, early 90s]

While there are more than 2 vertices:

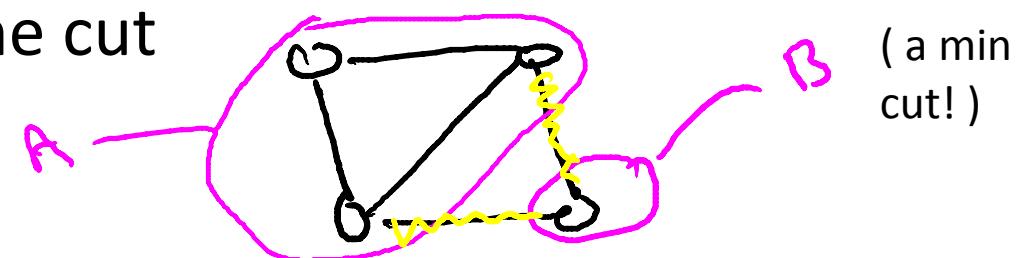
- pick a remaining edge (u,v) uniformly at random
- merge (or “contract”) u and v into a single vertex
- remove self-loops

return cut represented by final 2 vertices.

Example

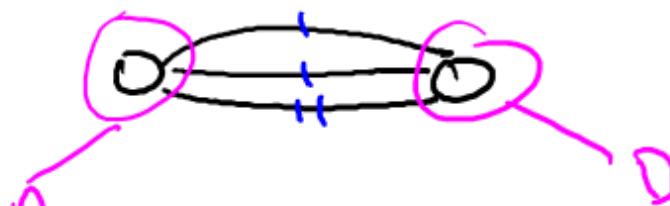
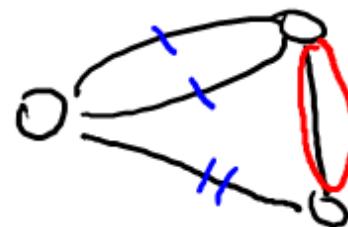
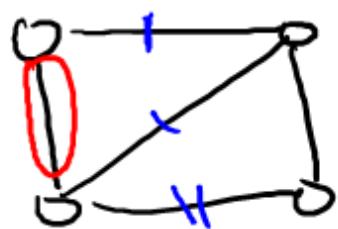


=> Corresponds to the cut



Tim Roughgarden

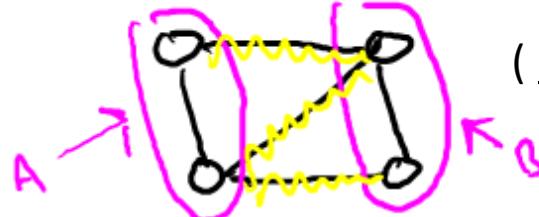
Example (con'd)



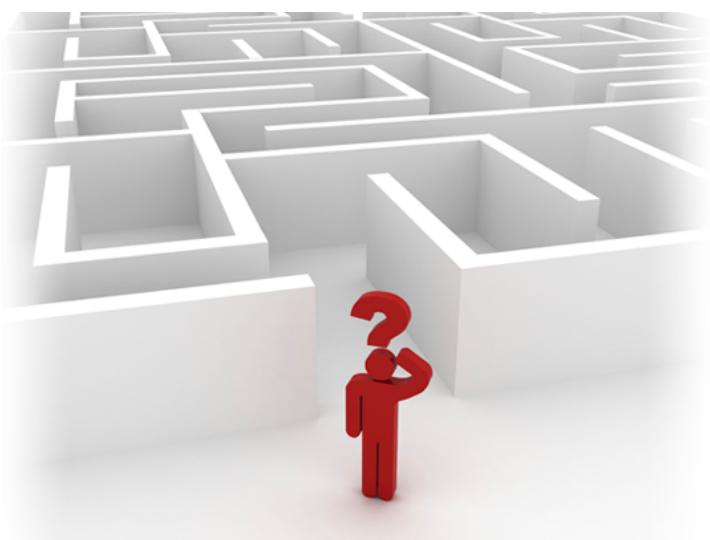
A

**KEY
QUESTION:**
What is the probability of success?

(not a min cut!)



- Corresponds to the cut



Contraction Algorithm

The Analysis

Design and Analysis
of Algorithms I

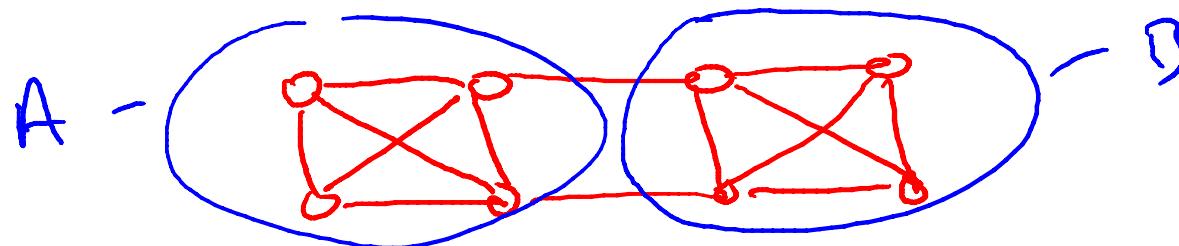
The Minimum Cut Problem

Input: An undirected graph $G = (V, E)$.

[parallel edges  allowed]

[See other video for representation of input]

Goal: Compute a cut with fewest number of crossing edges.
(a min cut)



Random Contraction Algorithm

[due to Karger, early 90s]

While there are more than 2 vertices:

- pick a remaining edge (u,v) uniformly at random
- merge (or “contract”) u and v into a single vertex
- remove self-loops

return cut represented by final 2 vertices.

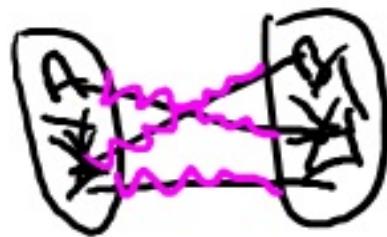
The Setup

Question: what is the probability of success?

Fix a graph $G = (V, E)$ with n vertices, m edges.

Fix a minimum cut (A, B) .

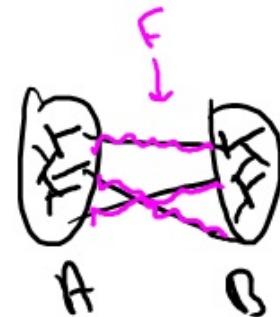
Let $k = \#$ of edges crossing (A, B) . (Call these edges F)



Tim Roughgarden

What Could Go Wrong?

1. Suppose an edge of F is contracted at some point
⇒ algorithm will not output (A, B).
2. Suppose only edges inside A or inside B get contracted ⇒ algorithm will output (A, B).



Thus: $\Pr [\text{output is } (A, B)] = \Pr [\text{never contracts an edge of } F]$

Let S_i = event that an edge of F contracted in iteration i.

Goal: Compute $\Pr[\neg S_1 \wedge \neg S_2 \wedge \neg S_3 \wedge \dots \wedge \neg S_{n-2}]$

What is the probability that an edge crossing the minimum cut (A, B) is chosen in the first iteration (as a function of the number of vertices n , the number of edges m , and the number k of crossing edges)?

- k/n
- k/m
- k/n^2
- n/m

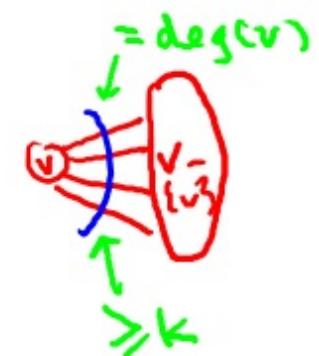
$$\Pr[S_1] = \frac{\text{\# of crossing edges}}{\text{\# of edges}} = \frac{k}{m}$$

The First Iteration

Key Observation: degree of each vertex is at least k
of incident edges

Reason: each vertex v defines a cut $(\{v\}, V - \{v\})$.

Since $\sum_v \underbrace{\text{degree}(v)}_{\geq kn} = 2m$, we have $m \geq \frac{kn}{2}$



Since $\Pr[S_1] = \frac{k}{m}$, $\Pr[S_1] \leq \frac{2}{n}$

The Second Iteration

$$\begin{aligned}
 \text{Recall: } \Pr[\neg S_1 \wedge \neg S_2] &= \Pr[\neg S_2 | \neg S_1] \cdot \Pr[\neg S_1] \\
 &= 1 - \frac{k}{\# \text{ of remaining edge}} \geq \left(1 - \frac{2}{n}\right)
 \end{aligned}$$

what is this?

Note: all nodes in contracted graph define cuts in G
 (with at least k crossing edges).

➤ all degrees in contracted graph are at least k

So: # of remaining edges $\geq \frac{1}{2}k(n-1)$

$$\text{So } \Pr[\neg S_2 | \neg S_1] \geq 1 - \frac{2}{(n-1)}$$

All Iterations

In general:

$$\begin{aligned}
 & \Pr[\neg S_1 \wedge \neg S_2 \wedge \neg S_3 \wedge \dots \wedge \neg S_{n-2}] \\
 &= \underbrace{\Pr[\neg S_1]}_{\text{Pr}} \underbrace{\Pr[\neg S_2 | \neg S_1]}_{\text{Pr}} \Pr[\neg S_3 | \neg S_2 \wedge \neg S_1] \dots \Pr[\neg S_{n-2} | \neg S_1 \wedge \dots \wedge \neg S_{n-3}] \\
 &\geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \dots \left(1 - \frac{2}{n-(n-4)}\right) \left(1 - \frac{2}{n-(n-3)}\right) \\
 &= \cancel{\frac{n-2}{n}} \cdot \cancel{\frac{n-3}{n-1}} \cdot \cancel{\frac{n-4}{n-2}} \dots \cancel{\frac{2}{4}} \cdot \cancel{\frac{1}{3}} = \frac{2}{n(n-1)} \geq \frac{1}{n^2}
 \end{aligned}$$

Problem: low success probability! (But: non trivial)

recall $\simeq 2^n$ cuts!

Repeated Trials

Solution: run the basic algorithm a large number N times, remember the smallest cut found.

Question: how many trials needed? 

Let T_i = event that the cut (A, B) is found on the i^{th} try.

➤ by definition, different T_i 's are independent

So: $\Pr[\text{all } N \text{ trials fail}] = \Pr[\neg T_1 \wedge \neg T_2 \wedge \dots \wedge \neg T_N]$

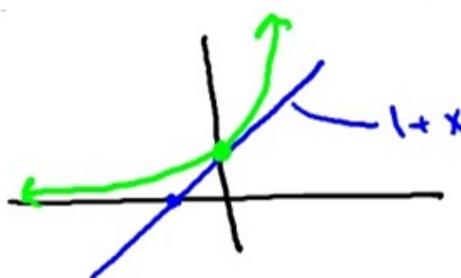
$$= \prod_{i=1}^N \Pr[\neg T_i] \leq \left(1 - \frac{1}{n^2}\right)^N$$

By independence!

Repeated Trials (con'd)

Calculus fact: \forall real numbers x , $1+x \leq e^x$

$$\Pr[\text{all trials fail}] \leq (1 - \frac{1}{n^2})^N$$

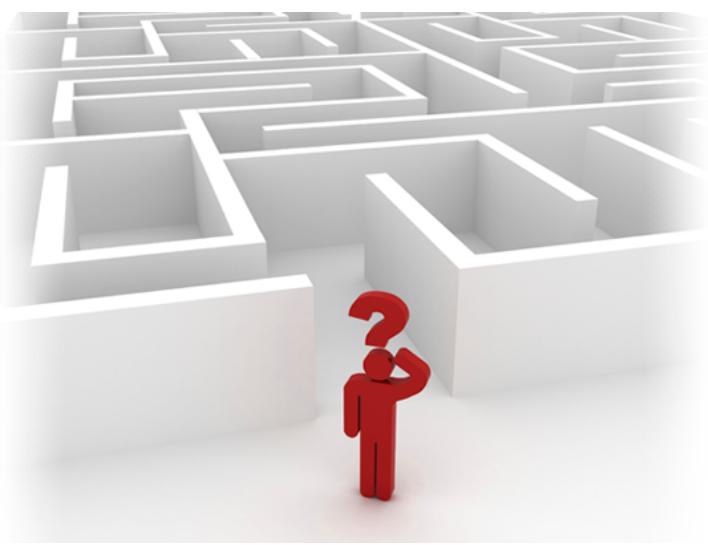


So: if we take $N = n^2$, $\Pr[\text{all fail}] \leq (e^{-\frac{1}{n^2}})^{n^2} = \frac{1}{e}$

If we take $N = n^2 \ln n$, $\Pr[\text{all fail}] \leq (\frac{1}{e})^{\ln n} = \frac{1}{n}$

Running time: polynomial in n and m but slow ($\Omega(n^2m)$)

But: can get big speed ups (to roughly $O(n^2)$) with more ideas.



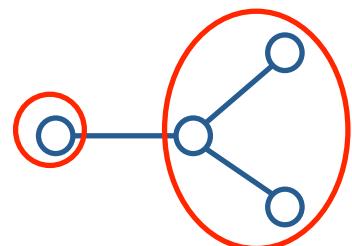
Design and Analysis
of Algorithms I

Contraction Algorithm

Counting Mininum Cuts

The Number of Minimum Cuts

NOTE: A graph can have multiple min cuts.
[e.g., a tree with n vertices has $(n-1)$ minimum cuts]



QUESTION: What's the largest number of min cuts that a graph with n vertices can have?

ANSWER:

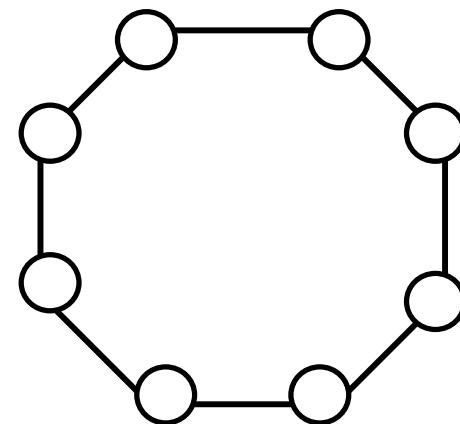
$$\binom{n}{2} = \frac{n(n - 1)}{2}$$

The Lower Bound

Consider the n-cycle.

NOTE: Each pair of the n edges defines
a distinct minimum cut
(with two crossing edges).

➤ has $\geq \binom{n}{2}$ min cuts



The Upper Bound

Let $(A_1, B_1), (A_2, B_2), \dots, (A_t, B_t)$ be the min cuts of a graph with n vertices.

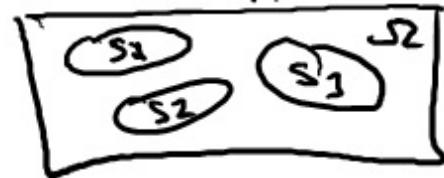
By the Contraction Algorithm analysis (without repeated trials):

$$\Pr[\underbrace{\text{output} = (A_i, B_i)}_{S_i}] \geq \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \quad \forall i = 1, 2, \dots, t$$

Note: S_i 's are disjoint events (i.e., only one can happen)

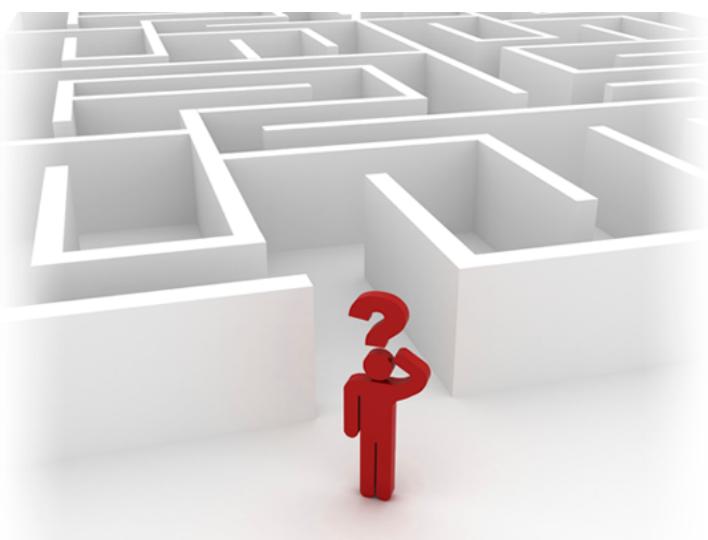
➤ their probabilities sum to at most 1

Thus: $\frac{t}{\binom{n}{2}} \leq 1 \Rightarrow t \leq \binom{n}{2}$



QED !

Tim Roughgarden



Contraction Algorithm

Overview

Design and Analysis
of Algorithms I

Goals for These Lectures

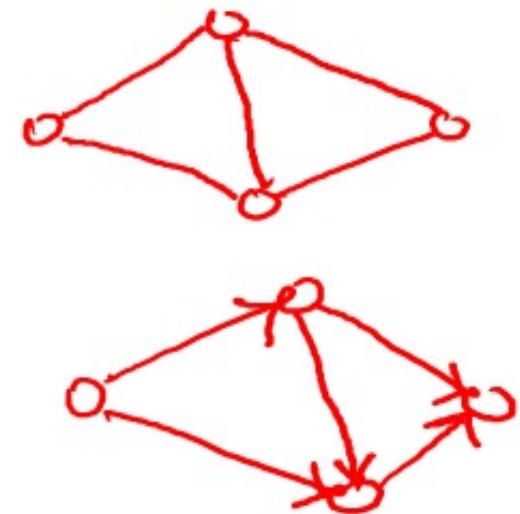
- Further practice with randomized algorithms
 - In a new application domain (graphs)
- Introduction to graphs and graph algorithms

Also: “only” 20 years ago!

Graphs

Two ingredients

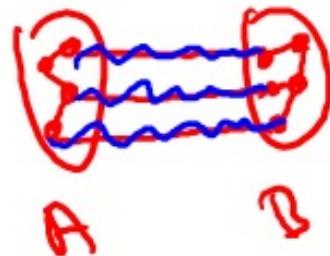
- Vertices aka nodes (V)
- Edges (E) = pairs of vertices
 - can be undirected [unordered pair] or directed [ordered pair] (aka arcs)



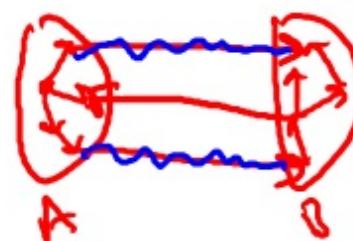
Examples: road networks, the Web, social networks, precedence constraints, etc.

Cuts of Graphs

Definition: a cut of a graph (V, E) is a partition of V into two non-empty sets A and B .



[undirected]



[directed]

Definition: the crossing edges of a cut (A, B) are those with:

- the one endpoint in each of (A, B) [undirected]
- tail in A , head in B [directed]

Roughly how many cuts does a graph with n vertices have?

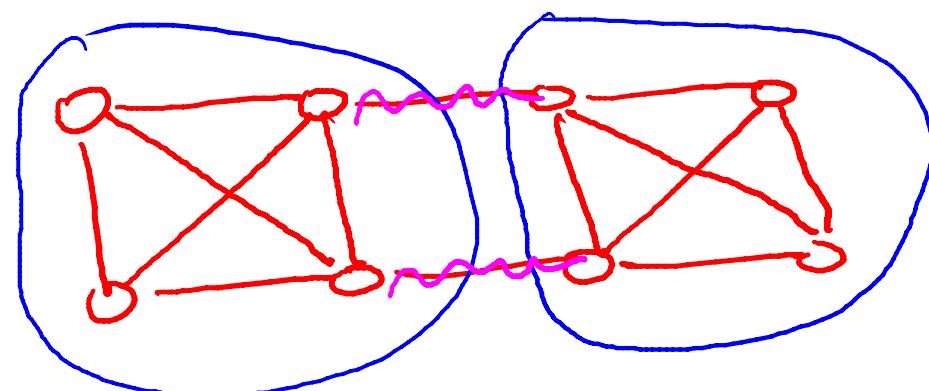
- n
- n^2
- 2^n
- n^n

The Minimum Cut Problem

- INPUT: An undirected graph $G = (V, E)$.
[Parallel  edges allowed]
[See other video for representation of the input]
- GOAL: Compute a cut with fewest number of crossing edges. (a min cut)

What is the number of edges crossing a minimum cut in the graph shown below?

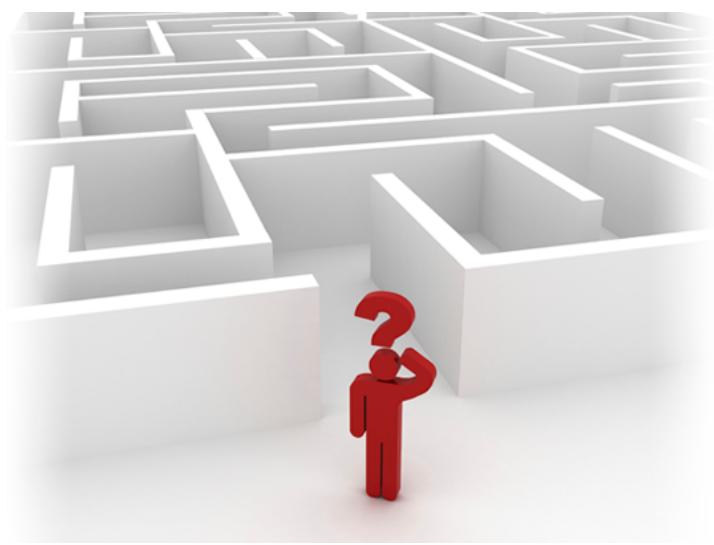
- 1
- 2
- 3
- 4



A Few Applications

- identify network bottlenecks / weaknesses
- community detection in social networks
- image segmentation
 - input = graph of pixels
 - use edge weights
 - [(u,v) has large weight \Leftrightarrow “expect” u,v to come from some object]

hope: repeated min cuts identifies the primary objects in picture.



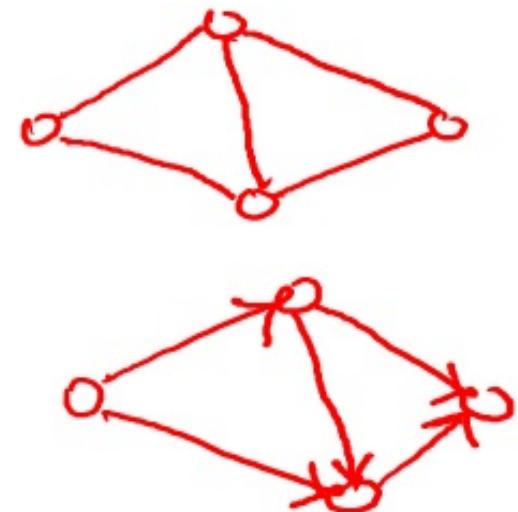
Design and Analysis
of Algorithms I

Graph Algorithms Representing Graphs

Graphs

Two ingredients

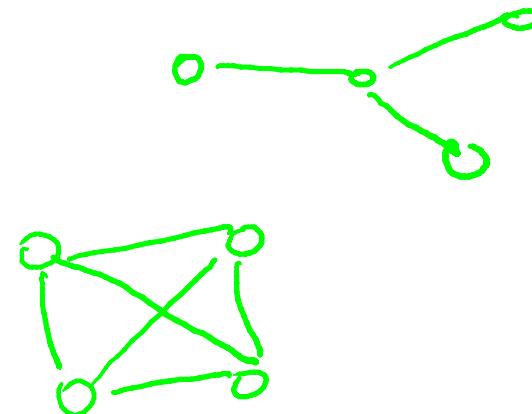
- Vertices aka nodes (V)
- Edges (E) = pairs of vertices
 - can be undirected [unordered pair] or directed [ordered pair] (aka arcs)



Examples: road networks, the Web, social networks, precedence constraints, etc.

Consider an undirected graph that has n vertices, no parallel edges, and is connected (i.e., “in one piece”). What is the minimum and maximum number of edges that the graph could have, respectively ?

- $n - 1$ and $n(n - 1)/2$
- $n - 1$ and n^2
- n and 2^n
- n and n^n



Sparse vs. Dense Graphs

Let $\underline{n} = \#$ of vertices, $\underline{m} = \#$ of edges.

In most (but not all) applications, m is $\Omega(n)$ and $O(n^2)$

- in a “sparse” graph, m is or is close to $O(n)$
- in a “dense” graph, m is closer to $\theta(n^2)$

The Adjacency Matrix

Represent G by a $n \times n$ 0-1 matrix A where

$$A_{ij} = 1 \Leftrightarrow G \text{ has an } i-j \text{ edge}$$


Variants

- $A_{ij} = \# \text{ of } i-j \text{ edges}$ (if parallel edges)
- $A_{ij} = \text{weight of } i-j \text{ edge}$ (if any)
- $A_{ij} = \begin{cases} +1 & \text{if } \text{○} \rightarrow \text{○} \\ -1 & \text{if } \text{○} \leftarrow \text{○} \end{cases}$

How much space does an adjacency matrix require, as a function of the number n of vertices and the number m of edges?

- $\theta(n)$
- $\theta(m)$
- $\theta(m + n)$
- $\theta(n^2)$

Adjacency Lists

Ingredients

- array (or list) of vertices
- array (or list) of edges
- each edge points to its endpoints
- each vertex points to edges incident on it

How much space does an adjacency list representation require, as a function of the number n of vertices and the number m of edges?

$\theta(n)$

$\theta(m)$

$\theta(m + n)$

$\theta(n^2)$

Adjacency Lists

Ingredients

- array (or list) of vertices
- array (or list) of edges
- each edge points to its endpoints
- each vertex points to edges incident on it

one-to-one

correspondence !

Space

$\theta(n)$

$\theta(m)$

$\theta(m)$

$\theta(m)$

$\theta(m + n)$

[or $\theta(\max\{m, n\})$]

Question: which is better?

Answer: depends on graph density and operations needed.

This course: focus on adjacency lists.



Graph Primitives

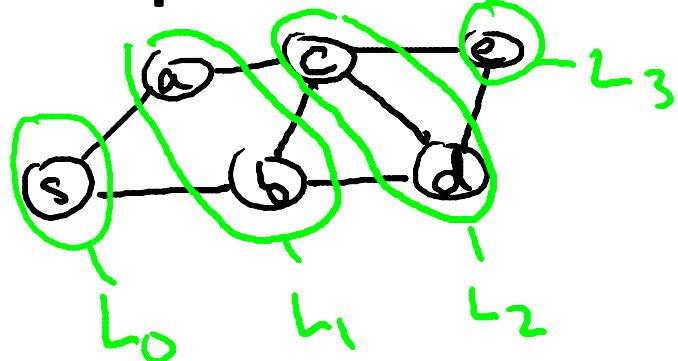
Breadth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Breadth-First Search (BFS)

- explore nodes in “layers”
- can compute shortest paths
- connected components of undirected graph



Run time : $O(m+n)$ [linear time]

The Code

BFS (graph G, start vertex s)

[all nodes initially unexplored]

-- mark s as explored

-- let Q = queue data structure (FIFO), initialized with s

-- while $Q \neq \emptyset$:

-- remove the first node of Q, call it v

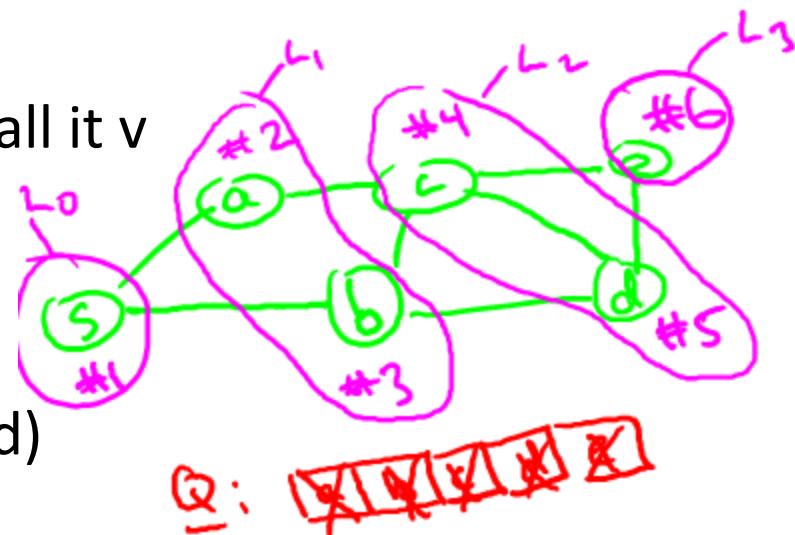
-- for each edge(v,w) :

-- if w unexplored

--mark w as explored

-- add w to Q (at the end)

O(1)
time



Basic BFS Properties

Claim #1 : at the end of BFS, v explored \iff
 G has a path from s to v .

Reason : special case of the generic algorithm

Claim #2 : running time of main while loop
= $O(n_s + m_s)$, where n_s = # of nodes reachable from s
 m_s = # of edges reachable from s

Reason : by inspection of code.

Application: Shortest Paths

Goal : compute $\text{dist}(v)$, the fewest # of edges on path from s to v .

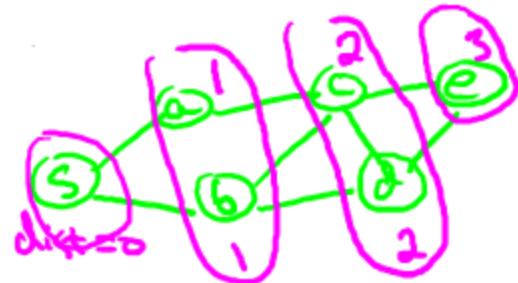
Extra code : initialize $\text{dist}(v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$

-When considering edge (v,w) :

- if w unexplored, then set $\text{dist}(w) = \text{dist}(v) + 1$

Claim : at termination $\text{dist}(v) = i \iff v$ in i th layer
(i.e., shortest s - v path has i edges)

Proof Idea : every layer i node w is added to Q by a layer $(i-1)$ node v via the edge (v,w)



Application: Undirected Connectivity

Let $G = (V, E)$ be an undirected graph.

Connected components = the “pieces” of G .

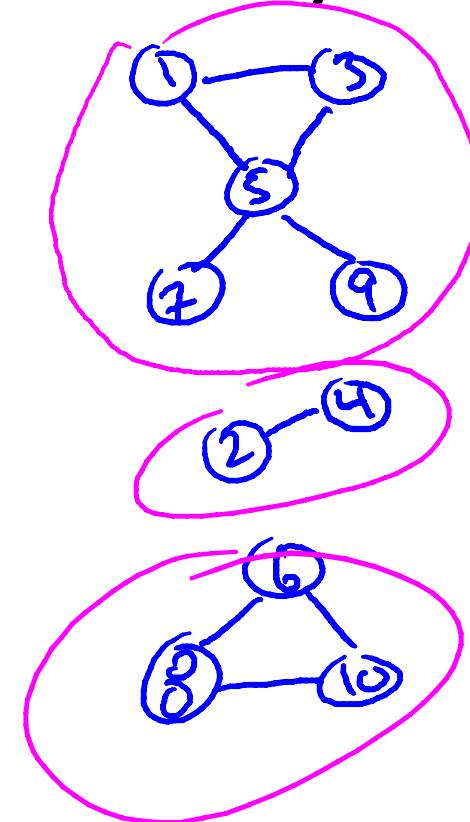
Formal Definition : equivalence classes of
the relation $u \leftrightarrow v \iff$ there exists $u-v$ path
in G . [check: \leftrightarrow is an equivalence relation]

Goal : compute all connected components

Why? - check if network is disconnected

- graph visualisation

- clustering



Connected Components via BFS

To compute all components : (undirected case)

-- initialize all nodes as unexplored $O(n)$

[assume labelled 1 to n]

-- for $i = 1$ to n $O(n)$

-- if i not yet explored [in some previous BFS]

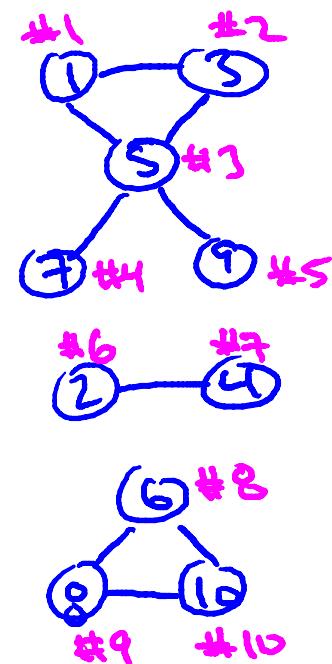
-- $\text{BFS}(G, i)$ [discovers precisely i 's connected component]

Note : finds every connected component.

Running time : $O(m+n)$

$O(1)$ per node

$O(1)$ per edge in each BFS





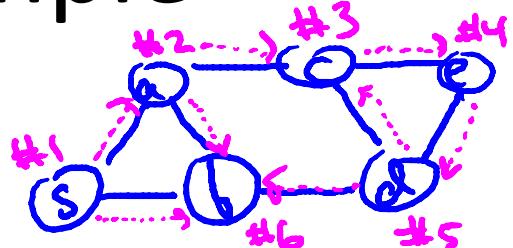
Graph Primitives

Depth-First Search

Design and Analysis
of Algorithms I

Overview and Example

Depth-First Search (DFS) : explore aggressively, only backtrack when necessary.



- also computes a topological ordering of a directed acyclic graph
- and strongly connected components of directed graphs

Run Time : $O(m+n)$

The Code

Exercise : mimic BFS code, use a stack instead of a queue [+
some other minor modifications]

Recursive version : DFS(graph G, start vertex s)

- mark s as explored
- for every edge (s,v) :
 - if v unexplored
 - $\text{DFS}(G,v)$

Basic DFS Properties

Claim #1 : at the end of the algorithm, v marked as explored
 \Leftrightarrow there exists a path from s to v in G.

Reason : particular instantiation of generic search procedure

Claim #2 : running time is $O(n_s + m_s)$,
where $n_s = \#$ of nodes reachable from s
 $m_s = \#$ of edges reachable from s

Reason : looks at each node in the connected component of s
at most once, each edge at most twice.

Application: Topological Sort

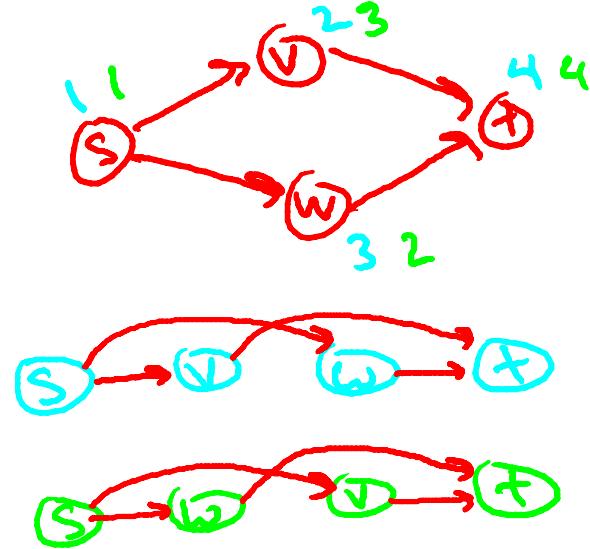
Definition : A topological ordering of a directed graph G is a labeling f of G 's nodes such that:

1. The $f(v)$'s are the set $\{1, 2, \dots, n\}$
2. $(u, v) \in G \Rightarrow f(u) < f(v)$

Motivation : sequence tasks while respecting all precedence constraints.

Note : G has directed cycle \Rightarrow no topological ordering

Theorem : no directed cycle \Rightarrow can compute topological ordering in $O(m+n)$ time.



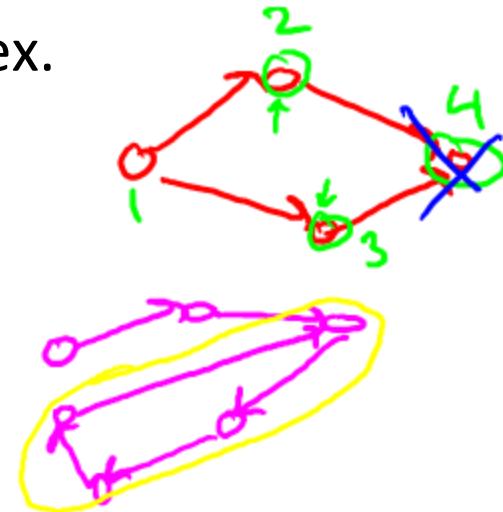
Straightforward Solution

Note : every directed acyclic graph has a sink vertex.

Reason : if not, can keep following outgoing arcs to produce a directed cycle.

To compute topological ordering :

- let v be a sink vertex of G
- set $f(v) = n$
- recurse on $G - \{v\}$



Why does it work? : when v is assigned to position i , all outgoing arcs already deleted => all lead to later vertices in ordering.

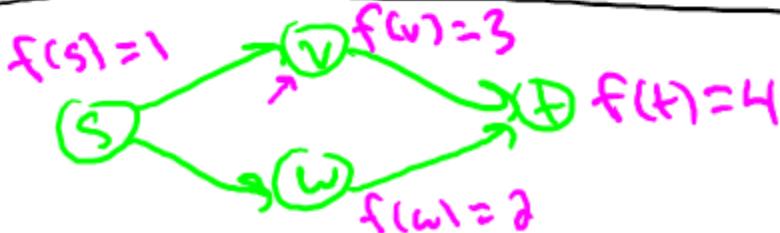
Topological Sort via DFS (Slick)

DFS-Loop (graph G)

- mark all nodes unexplored
- current-label = n [to keep track of ordering]
- for each vertex
 - if v not yet explored [in previous DFS call]
 - DFS(G,v)

DFS(graph G, start vertex s)

- for every edge (s,v)
 - if v not yet explored
 - mark v explored
 - DFS(G,v)
- set $f(s) = \text{current_label}$
- $\text{current_label} = \text{current_label} - 1$



Topological Sort via DFS (con'd)

Running Time : $O(m+n)$.

Reason : $O(1)$ time per node, $O(1)$ time per edge.

Correctness : need to show that if (u,v) is an edge,
then $f(u) < f(v)$



(since no
directed cycles)

Case 1 : u visited by DFS before $v \Rightarrow$ recursive call
corresponding to v finishes before that of u (since DFS).
 $\Rightarrow f(v) > f(u)$

Case 2 : v visited before $u \Rightarrow v$'s recursive call finishes before
 u 's even starts. $\Rightarrow f(v) > f(u)$

Q.E.D.



Design and Analysis
of Algorithms I

Graph Primitives

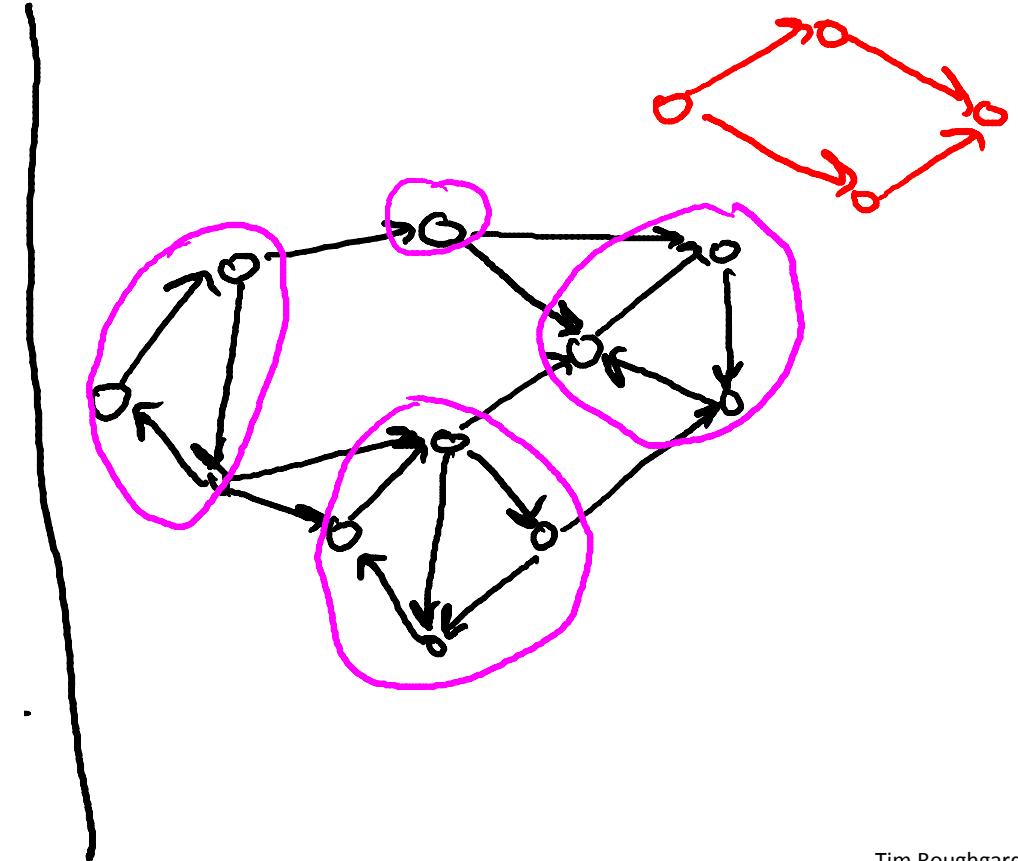
An $O(m+n)$ Algorithm
for Computing Strong
Components

Strongly Connected Components

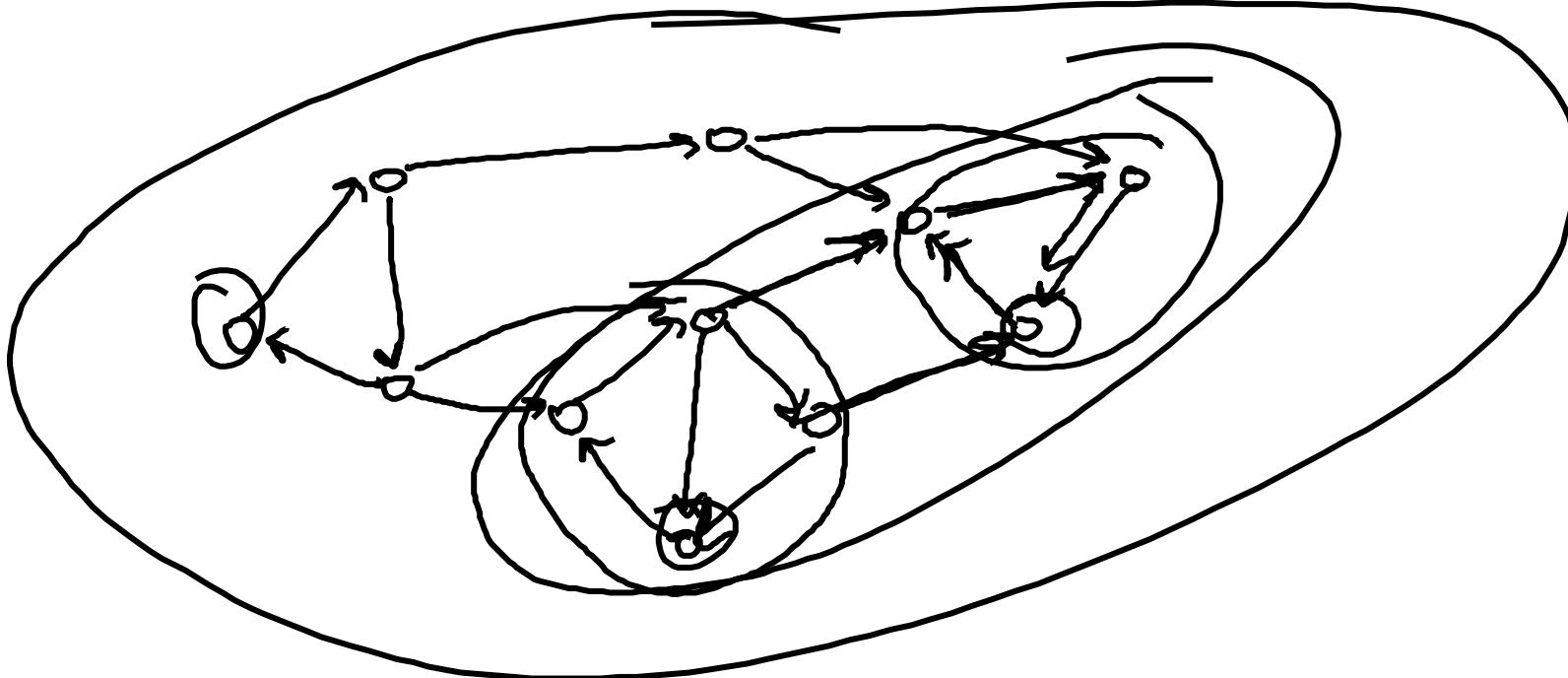
Formal Definition : the strongly connected components (SCCs) of a directed graph G are the equivalence classes of the relation

$u \leftrightarrow v \iff$ there exists a path $u \rightarrow v$ and a path $v \rightarrow u$ in G

You check : \leftrightarrow is an equivalence relation



Why Depth-First Search?



Kosaraju's Two-Pass Algorithm

Theorem : can compute SCCs in $O(m+n)$ time.

Algorithm : (given directed graph G)

1. Let $\text{Grev} = G$ with all arcs reversed
2. Run DFS-Loop on Grev ← Goal : compute “magical ordering” of nodes
Let $f(v) = \text{“finishing time” of each } v \text{ in } V$ Goal : discover the SCCs
1. Run DFS-Loop on G ← one-by-one
processing nodes in decreasing order of finishing times
[SCCs = nodes with the same “leader”]

DFS-Loop

DFS-Loop (graph G)

Global variable $t = 0$

For finishing
times in 1st
pass

[# of nodes processed so far]

For leaders
in 2nd pass

Global variable $s = \text{NULL}$

[current source vertex]

Assume nodes labeled 1 to n

For $i = n$ down to 1

if i not yet explored

$s := i$

$\text{DFS}(G, i)$

DFS (graph G, node i)

-- mark i as explored

For rest of
DFS-Loop

-- set $\text{leader}(i) := \text{node } s$

-- for each arc (i, j) in G :

-- if j not yet explored

-- $\text{DFS}(G, j)$

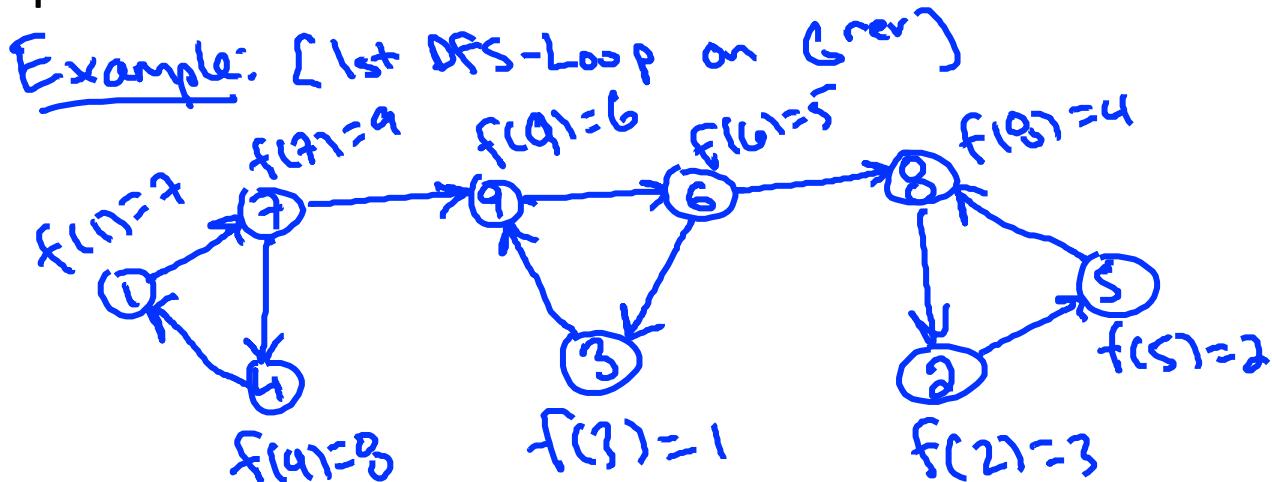
-- $t++$

-- set $f(i) := t$

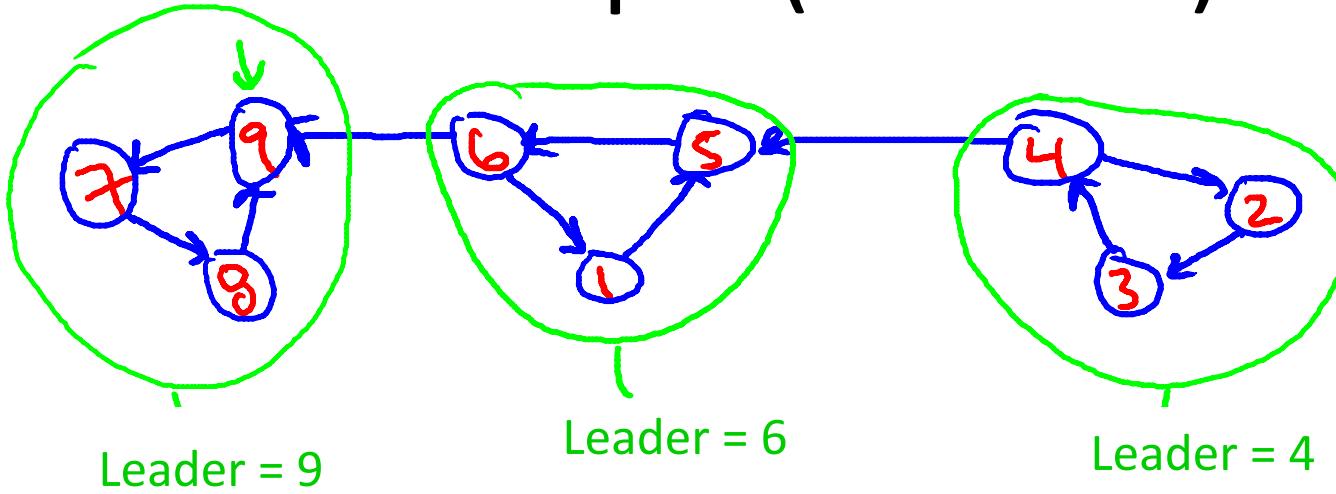
i's finishing
time

Only one of the following is a possible set of finishing times for the nodes 1,2,3,...,9, respectively, when the DFS-Loop subroutine is executed on the graph below. Which is it?

- 9,8,7,6,5,4,3,2,1
- 1,7,4,9,6,3,8,2,5
- 1,7,9,6,8,2,5,3,4
- 7,3,1,8,2,5,9,4,6



Example (2nd Pass)



Running Time : $2 * \text{DFS} = O(m+n)$

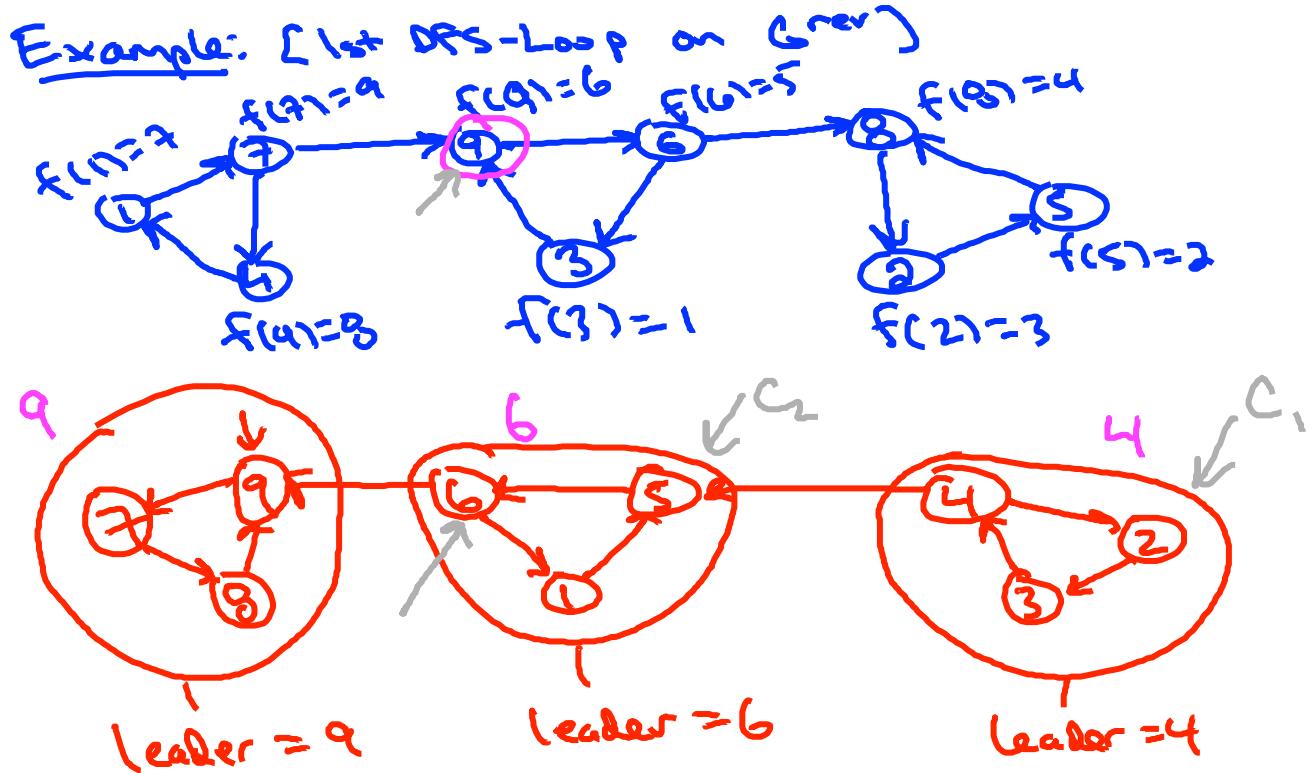


Design and Analysis
of Algorithms I

Graph Primitives

Correctness of
Kosaraju's Algorithm

Example Recap

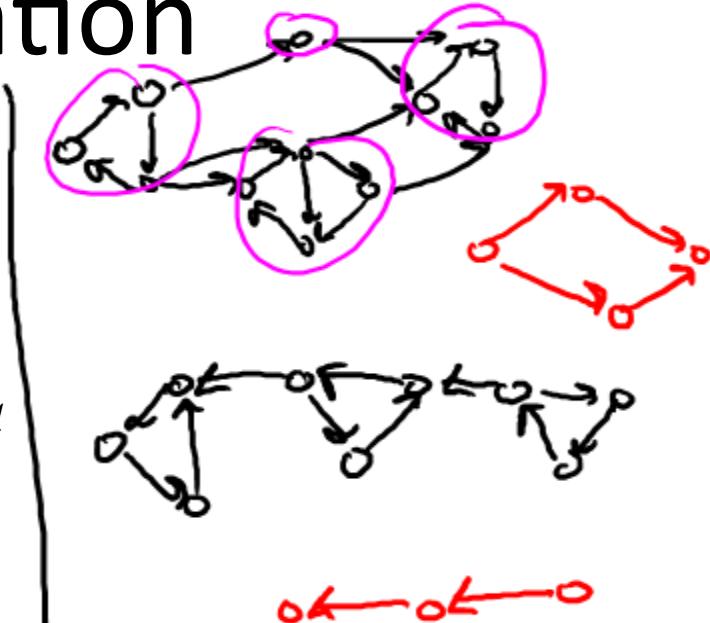
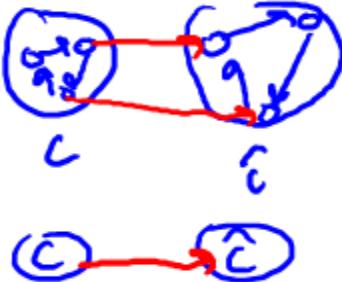


Observation

Claim : the SCCs of a directed graph G induce an acyclic “meta-graph”:

- meta-nodes = the SCCs C_1, \dots, C_k of G
- \exists arc $C \rightarrow \hat{C} \iff \exists$ arc $\square(i, j) \in G$
with $i \in C, j \in \hat{C}$

Why acyclic ? : a cycle of SCCs would collapse into one.

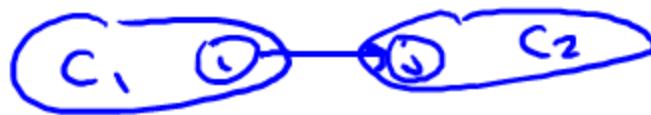


What how are the SCC of the original graph G and its reversal $G^{\uparrow rev}$ related?

- In general, they are unrelated.
- Every SCC of G is contained in an SCC of $G^{\uparrow rev}$, but the converse need not hold.
- Every SCC of $G^{\uparrow rev}$ is contained in an SCC of G , but the converse need not hold.
- They are exactly the same.

Key Lemma

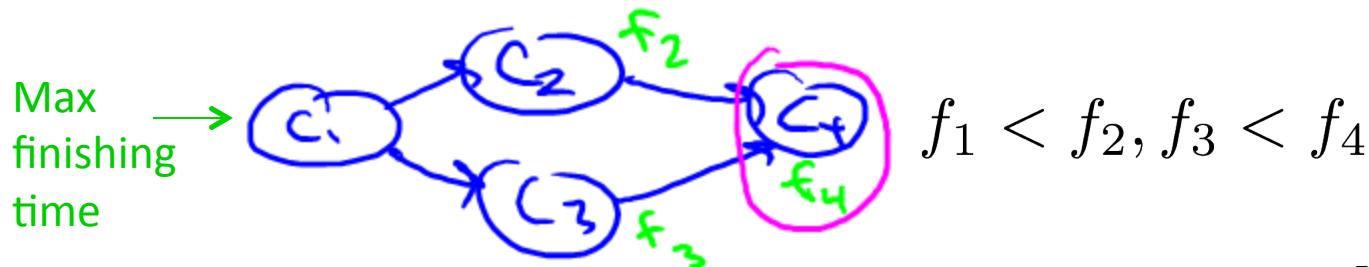
Lemma : consider two “adjacent” SCCs in G:



Let $f(v)$ = finishing times of DFS-Loop in Grev

Then : $\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v)$

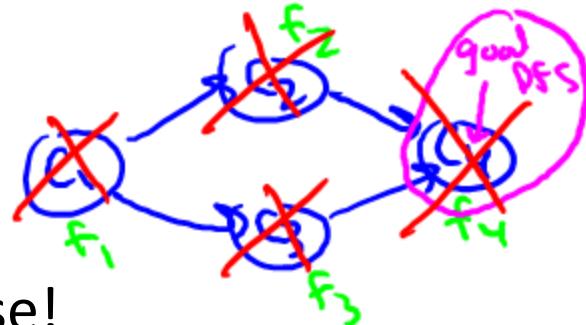
Corollary : maximum f-value of G must lie in a “sink SCC”



Correctness Intuition

(see notes for formal proof)

By Corollary : 2nd pass of DFS-Loop begins somewhere in a sink SCC C^* .



⇒ First call to DFS discovers C^* and nothing else!

⇒ Rest of DFS-Loop like recursing on G with C^* deleted

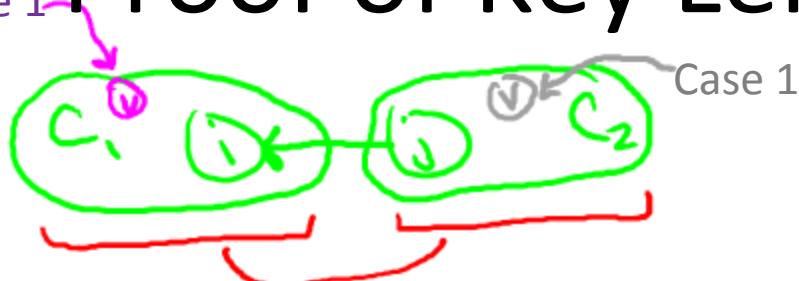
[starts in a sink node of $G - C^*$]

⇒ successive calls to $\text{DFS}(G, i)$ “peel off” the SCCs one by one

[in reverse topological order of the “meta-graph” of SCCs]

Proof of Key Lemma

In Grev:



Let $v = 1^{\text{st}}$ node of $C_1 \cup C_2$

Still SCCs (of Grev)
[by Quiz]

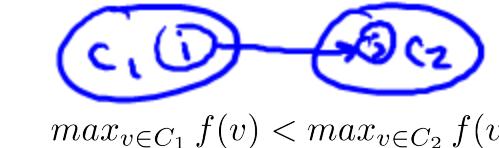
reached by 1st pass of DFS-Loop (on Grev)

Case 1 [$v \in C_1$] : all of C_1 explored before C_2 ever reached.

Reason : no paths from C_1 to C_2 (since meta-graph is acyclic)

\Rightarrow All f-values in C_1 less than all f-values in C_2

Case 2 [$v \in C_2$] : $\text{DFS}(\text{Grev}, v)$ won't finish until all of $C_1 \cup C_2$ completely explored $\Rightarrow f(v) > f(w)$ for all w in C_1



Q.E.D.



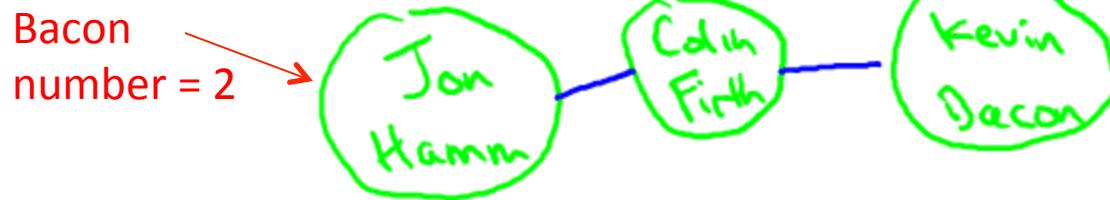
Design and Analysis
of Algorithms I

Graph Primitives

Introduction to Graph Search

A Few Motivations

1. Check if a network is connected (can get to anywhere from anywhere else)



2. Driving directions
3. Formulate a plan [e.g., how to fill in a Sudoku puzzle]
-- nodes = a partially completed puzzle -- arcs = filling in one new sequence
4. Compute the “pieces” (or “components”) of a graph
-- clustering, structure of the Web graph, etc.

Generic Graph Search

- Goals : 1) find everything findable from a given start vertex
2) don't explore anything twice

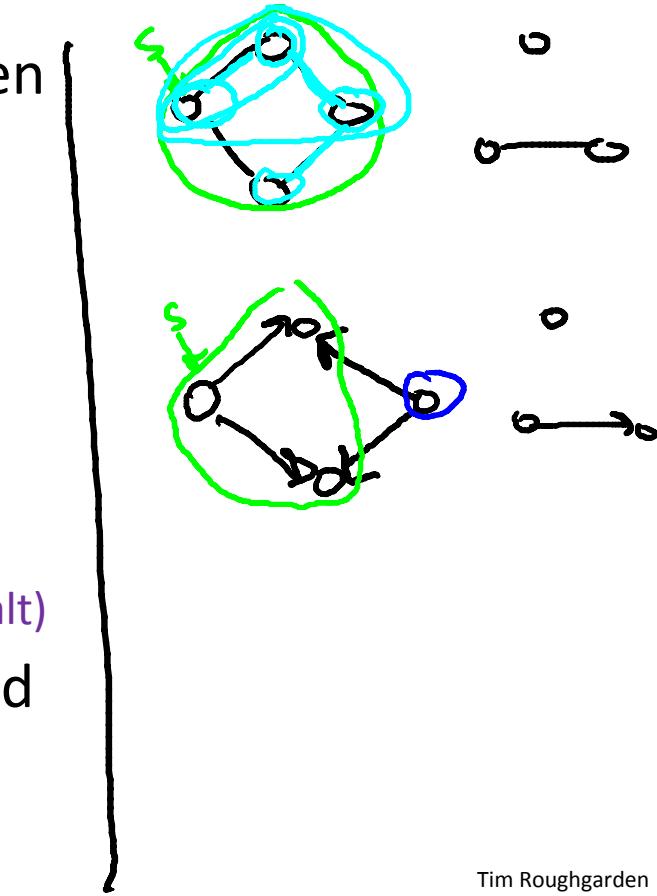
Goal:
 $O(m+n)$ time

Generic Algorithm (given graph G , vertex s)

-- initially s explored, all other vertices unexplored

-- while possible : (if none, halt)

- choose an edge (u,v) with u explored and v unexplored
- mark v explored



Generic Graph Search (con'd)

Claim : at end of the algorithm, v explored \Leftrightarrow G has a path from (G undirected or directed) s to v

Proof : (\Rightarrow) easy induction on number of iterations (you check)

(\Leftarrow) By contradiction. Suppose G has a path P from s to v:



But v unexplored at end of the algorithm. Then there exists an edge (u,x) in P with u explored and x unexplored.

But then algorithm would not have terminated, contradiction.

Q.E.D.

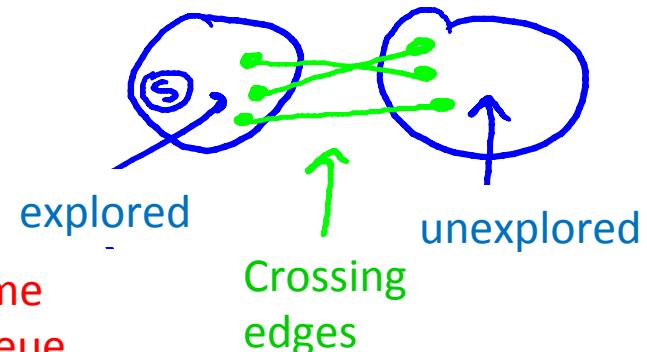
BFS vs. DFS

Note : how to choose among the possibly many “frontier” edges ?

Breadth-First Search (BFS)

- explored nodes in “layers”
- can compute shortest paths
- can compute connected components of an undirected graph

$O(m+n)$ time
using a queue
(FIFO)



Depth-First Search (DFS)

$O(m+n)$ time using a stack (LIFO)
(or via recursion)

- explore aggressively like a maze, backtrack only when necessary
- compute topological ordering of a directed acyclic graph
- compute connected components in directed graphs



Graph Primitives

Structure of the Web

Design and Analysis
of Algorithms I

The Web graph

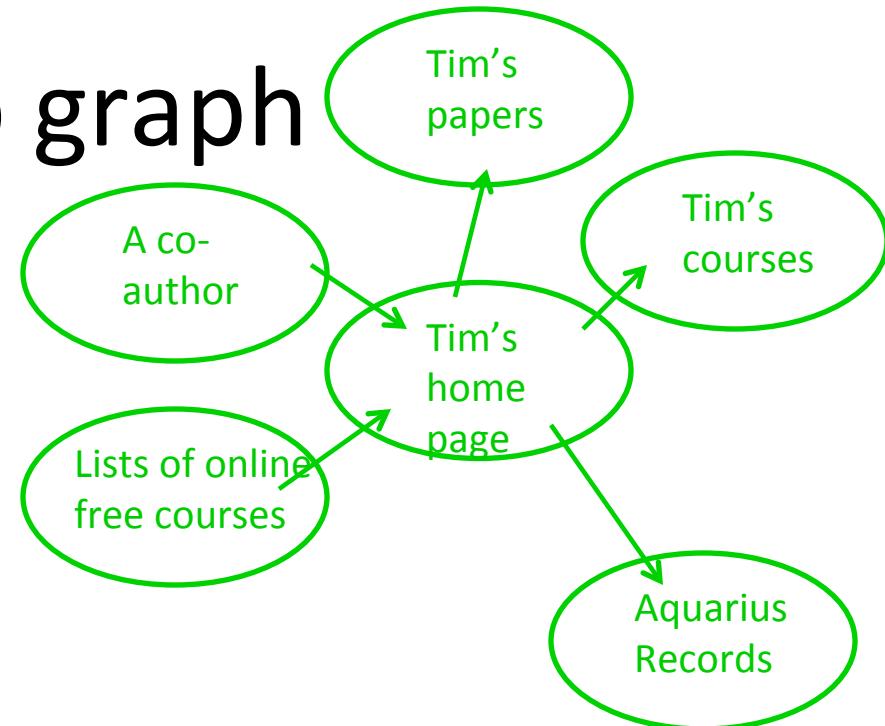
- Vertices = Web pages
- (directed) edges = hyperlinks

Question : what does the web graph
look like ?

(assume you've already “crawled” it)

Size : ~ 200 million nodes, ~ 1 billion edges

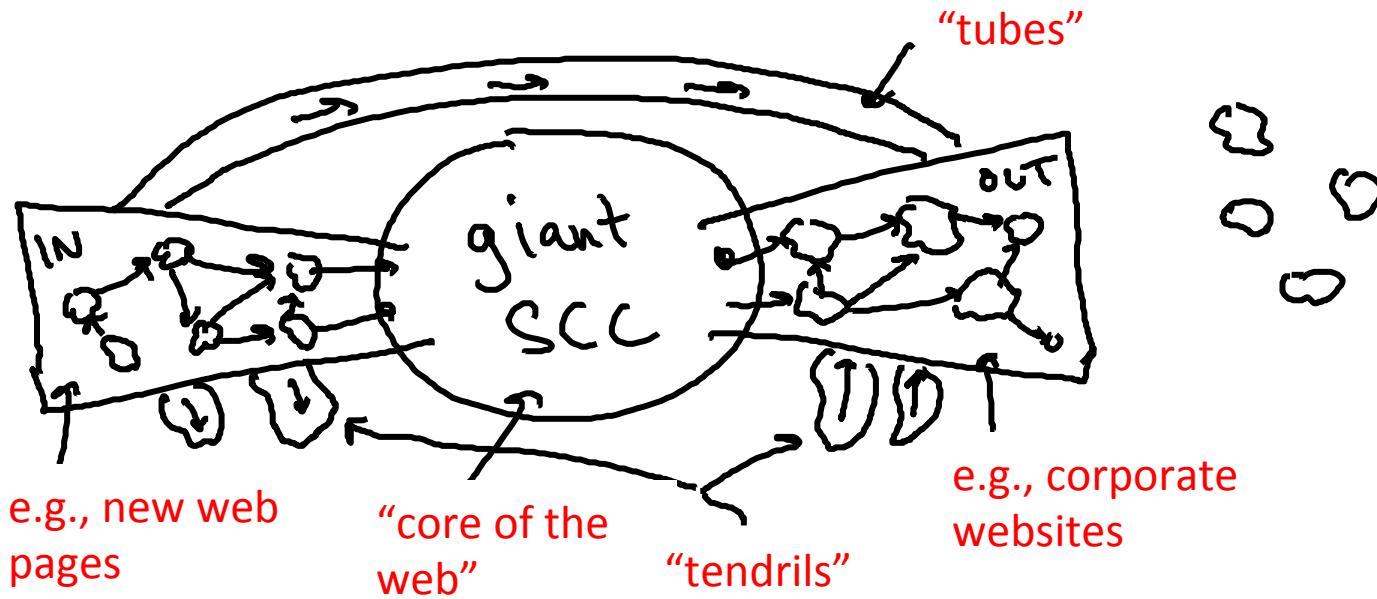
Reference : [Broder et al WWW 2000]
computed the SCCs of the Web graph.



ETC.
ETC.

(pre map-reduce/hadoop)

The Bow Tie

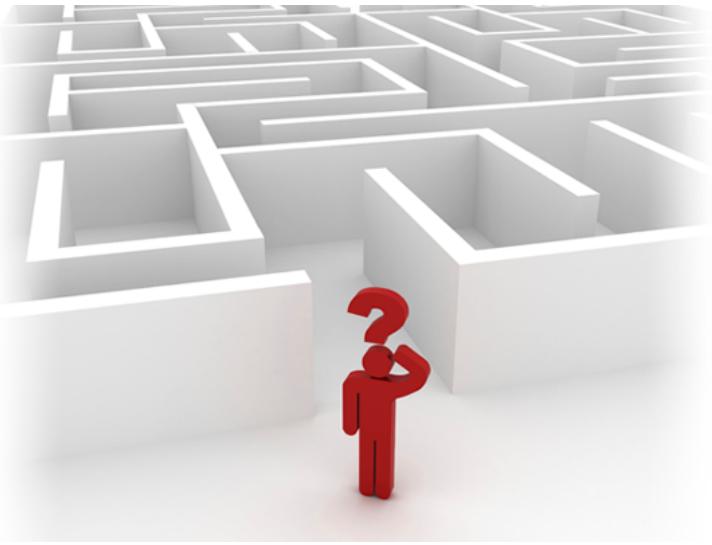


Main Findings

1. All 4 parts (giant, IN, OUT, tubes + tendrils) have roughly the same size
2. Within CORE, very well connected (has the “small world” property) [Milgram]
3. Outside, surprisingly poorly connected

Modern Web Research

1. **Temporal aspects** – how is the web graph evolving over time ?
 2. **Informational aspects** – how does new information propagate throughout the Web (or blogosphere, or Twitter, etc.)
 3. **Finer-grained structure** – how to define and compute “communities” in information and social networks ?
- Recommended Reading :** Easley + Kleinberg, “Networks, Crowds, & Markets”



Design and Analysis
of Algorithms I

Graph Primitives

Dijkstra's Algorithm: The Basics

Single-Source Shortest Paths

Input: directed graph $G=(V, E)$. ($m=|E|$, $n=|V|$)

- each edge has non negative length l_e
- source vertex s

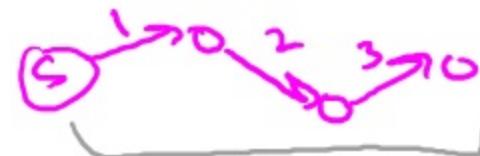
Output: for each $v \in V$, compute

$L(v) :=$ length of a shortest $s-v$ path in G

Assumption:

1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $l_e \geq 0 \quad \forall e \in E$

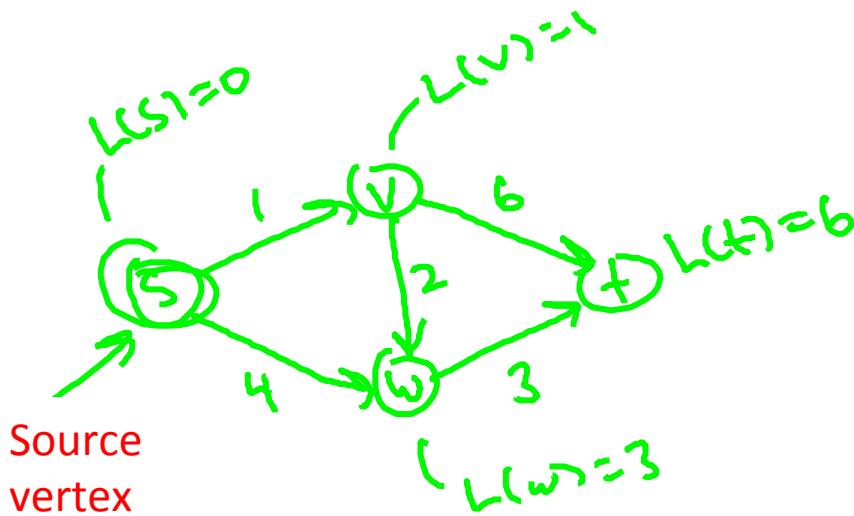
Length of path
= sum of edge lengths



Path length = 6

One of the following is the list of shortest-path distances for the nodes s, v, w, t , respectively. Which is it?

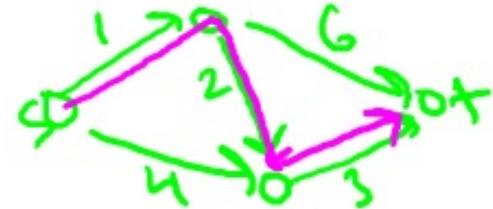
- 0,1,2,3
- 0,1,4,7
- 0,1,4,6
- 0,1,3,6



Why Another Shortest-Path Algorithm?

Question: doesn't BFS already compute shortest paths in linear time?

Answer: yes, IF $l_e = 1$ for every edge e .



Question: why not just replace each edge e by directed path of l_e unit length edges:



Answer: blows up graph too much

Solution: Dijkstra's shortest path algorithm.

This array
only to help
explanation!

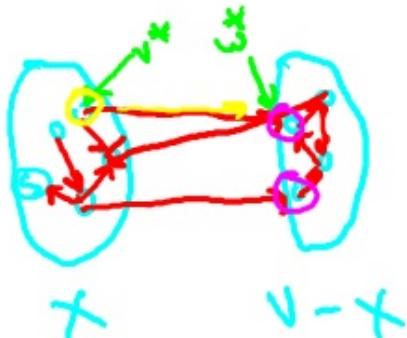
Dijkstra's Algorithm

Initialize:

- $X = [s]$ [vertices processed so far]
- $A[s] = 0$ [computed shortest path distances]
- $B[s] = \text{empty path}$ [computed shortest paths]

Main Loop

- while $X \neq V$:

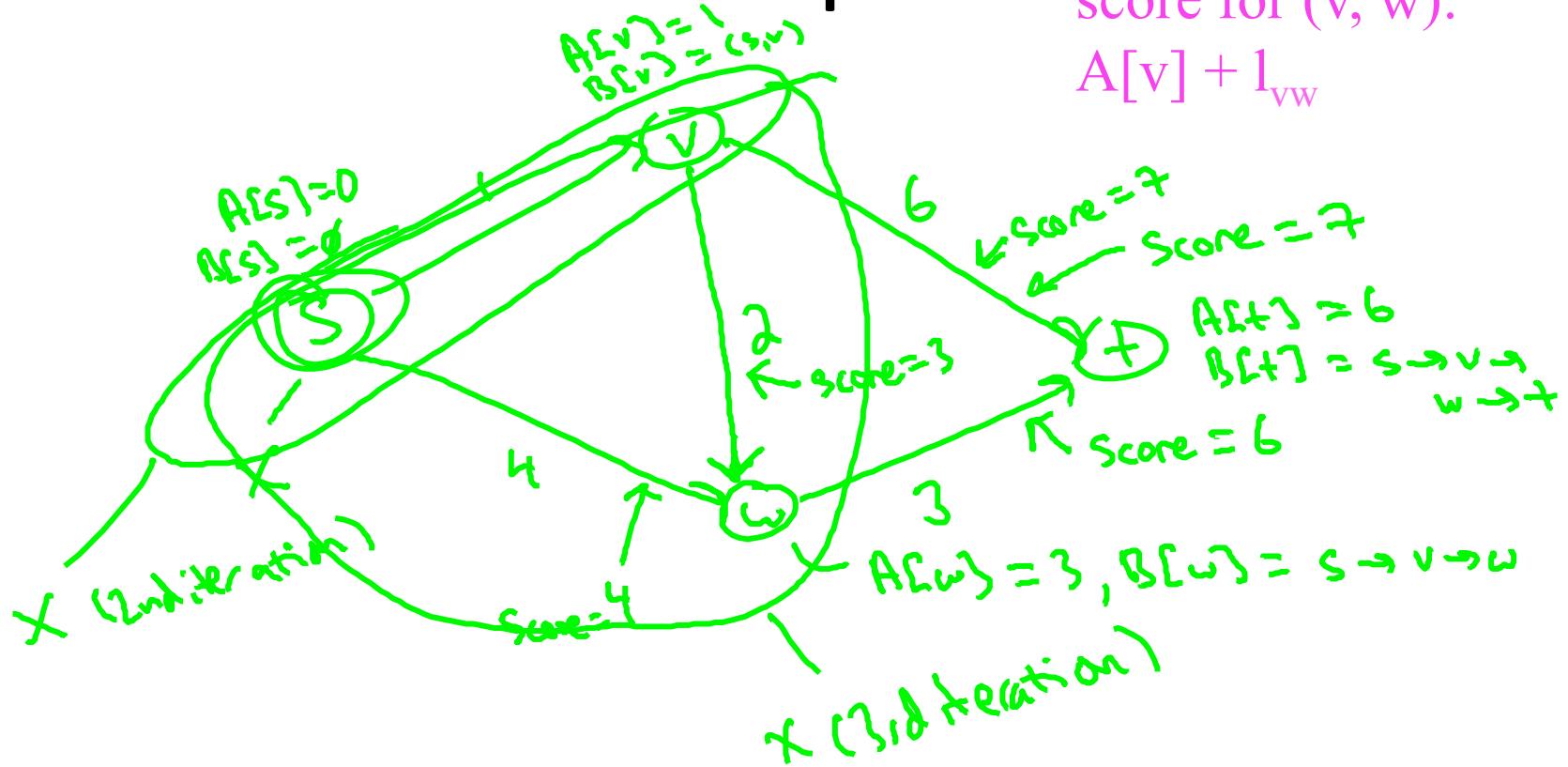


Main Loop cont'd:

- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$
- [call it (v^*, w^*)] Already computed in earlier iteration
- add w^* to X
 - set $A[w^*] := A[v^*] + l_{v^*w^*}$
 - set $B[w^*] := B[v^*] \cup (v^*, w^*)$

Example

Dijkstra's greedy
score for (v, w) :
 $A[v] + l_{vw}$

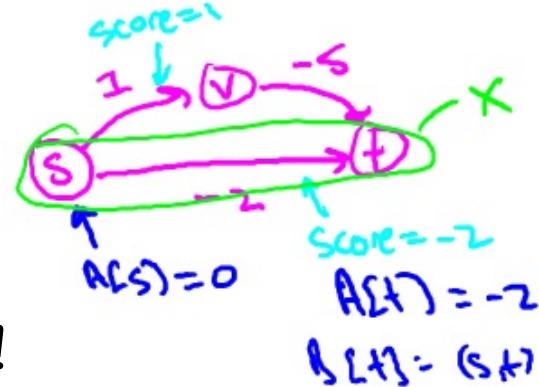


Non-Example

Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)

Problem: doesn't preserve shortest paths !

Also: Dijkstra's algorithm incorrect on this graph !
(computes shortest s-t distance to be -2 rather than -4)





Design and Analysis
of Algorithms I

Graph Primitives

Dijkstra's Algorithm: Why It Works

This array
only to help
explanation!

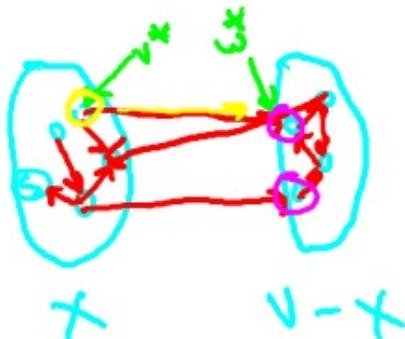
Dijkstra's Algorithm

Initialize:

- $X = [s]$ [vertices processed so far]
- $A[s] = 0$ [computed shortest path distances]
- $B[s] = \text{empty path}$ [computed shortest paths]

Main Loop

- while $X \neq V$:



-need to grow
x by one node

Main Loop cont'd:

- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$
- [call it (v^*, w^*)]
- Already computed in earlier iteration
- add w^* to X
 - set $A[w^*] := A[v^*] + l_{v^*w^*}$
 - set $B[w^*] := B[v^*] \cup (v^*, w^*)$

Correctness Claim

Theorem [Dijkstra] For every directed graph with nonnegative edge lengths, Dijkstra's algorithm correctly computes all shortest-path distances.

$$[i.e., A[v] = L(v) \quad \forall v \in V]$$

what algorithm
computes

True shortest
distance from s to v

Proof: by induction on the number of iterations.

Base Case: $A[s] = L[s] = 0$ (correct)

Proof

Inductive Step:

Inductive Hypothesis: all previous iterations correct (i.e., $A[v] = L(v)$ and $B[v]$ is a true shortest s-v path in G , for all v already in X).

In current iteration:

We pick an edge (v^*, w^*) and we add w^* to X .

We set $B[w^*] = B[v^*] \cup (v^*, w^*)$

has length $L(v^*) + l_{v^*w^*}$

has length $L(v^*)$

$L(v^*)$ by I.H

Also: $A[w^*] = A[v^*] + l_{v^*w^*} = L(v^*) + l_{v^*w^*}$



Proof (con'd)

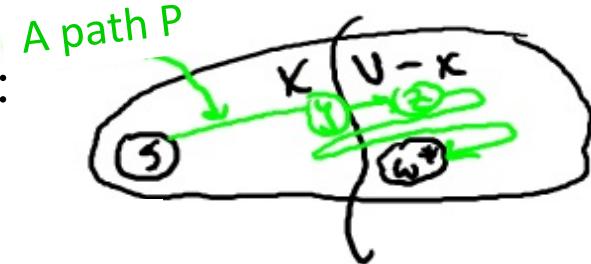
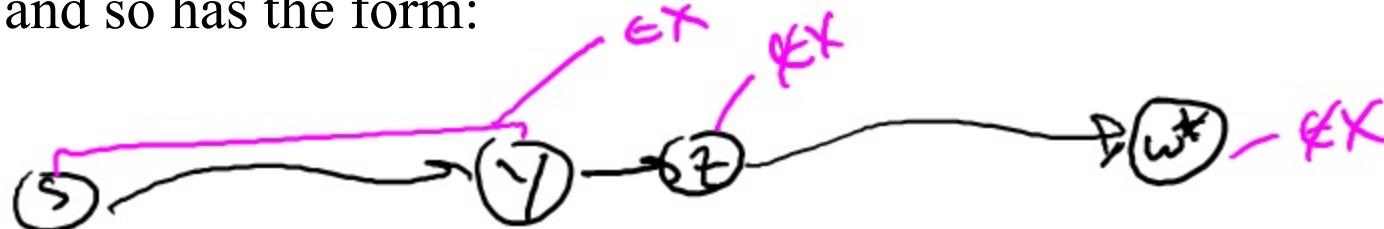
Upshot: in current iteration, we set:

1. $A[w^*] = L(v^*) + l_{v^*w^*}$
2. $B[w^*] = \text{an } s \rightarrow w^* \text{ path with length } (L(v^*) + l_{v^*w^*})$

To finish proof: need to show that *every* $s-w^*$ path has length $\geq L(v^*) + l_{v^*w^*}$ (if so, our path is the shortest!)

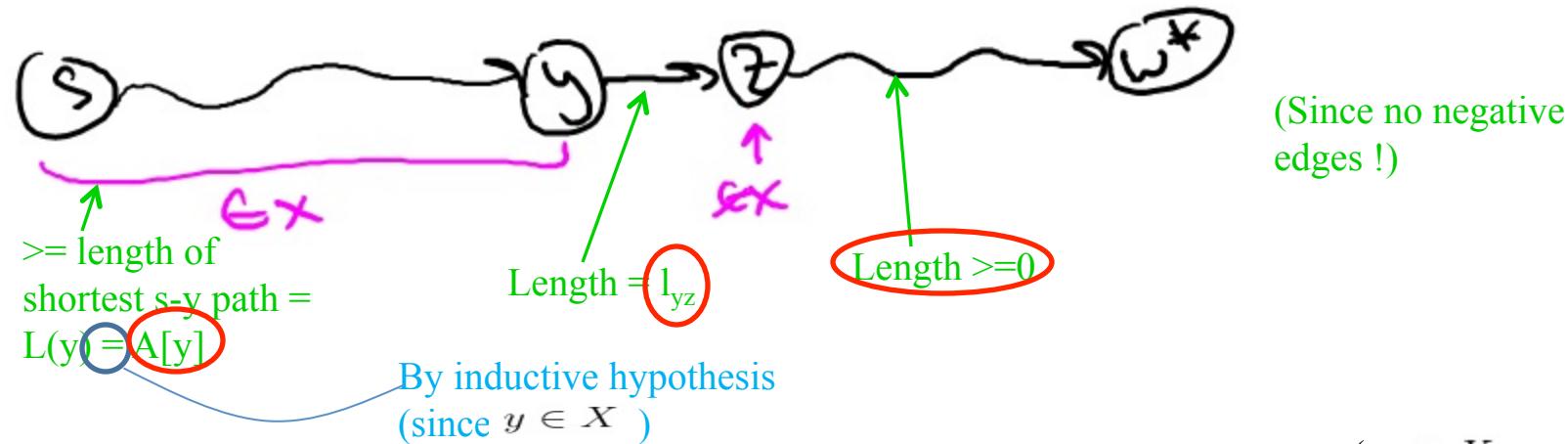
So: Let $P = \text{any } s \rightarrow w^* \text{ path. Must "cross the frontier":}$

and so has the form:



Proof (con'd)

So: every $s \rightarrow w^*$ path P has to have the form



Total length of path P : at least $A[y] + C_{yz}$ ↑ length of our path !
 -> by Dijkstra's greedy criterion, $A[v^*] + l_{v^*w^*} \leq A[y] + l_{yz} \leq$ length of P

$$(y \in X \\ z \notin X)$$

Q.E.D.



Design and Analysis
of Algorithms I

Graph Primitives

Dijkstra's Algorithm: Fast Implementation

Single-Source Shortest Paths

Input: directed graph $G=(V, E)$. ($m=|E|$, $n=|V|$)

- each edge has non negative length l_e
- source vertex s

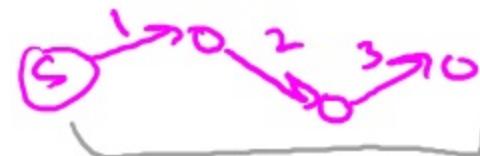
Output: for each $v \in V$, compute

$L(v) :=$ length of a shortest $s-v$ path in G

Assumption:

1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $l_e \geq 0 \quad \forall e \in E$

Length of path
= sum of edge lengths



Path length = 6

This array
only to help
explanation!

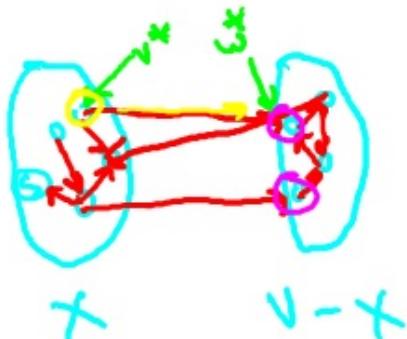
Dijkstra's Algorithm

Initialize:

- $X = [s]$ [vertices processed so far]
- $A[s] = 0$ [computed shortest path distances]
- ~~$B[s] = \text{empty path}$ [computed shortest paths]~~

Main Loop

- while $X \neq V$:



-need to grow
 X by one node

Main Loop cont'd:

- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$
- [call it (v^*, w^*)]
- Already computed in earlier iteration
- add w^* to X
 - set $A[w^*] := A[v^*] + l_{v^*w^*}$
 - set ~~$B[w^*] := B[v^*] \cup (v^*, w^*)$~~

Which of the following running times seems to best describe a “naïve” implementation of Dijkstra’s algorithm?

- $\theta(m+n)$
- $\theta(m \log n)$
- $\theta(n^{12})$
- $\theta(mn)$

- $(n-1)$ iterations of while loop
- $\theta(m)$ work per iteration
 - [$\theta(1)$ work per edge]

CAN WE DO BETTER?

Heap Operations

Recall: raison d'être of heap = perform Insert, Extract-Min in $O(\log n)$ time.
[rest of video assumes familiarity with heaps]

Height $\sim \log_2 n$

- conceptually, a perfectly balanced binary tree
- Heap property: at every node, key \leq children's keys
- extract-min by swapping up last leaf, bubbling down
- insert via bubbling up



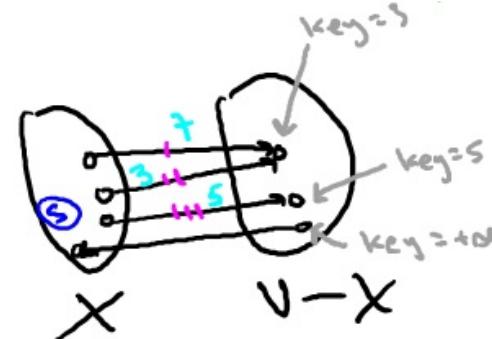
Also: will need ability to delete from middle of heap. (bubble up or down as needed)

Two Invariants

Invariant #1: elements in heap = vertices of $V-X$.

Invariant #2: for $v \notin X$
 $\text{Key}[v] = \text{smallest Dijkstra greedy score of an edge } (u, v) \text{ in } E \text{ with } v \in X$

(of $+\infty$ if no such edges exist)



Point: by invariants, Extract-Min yields correct vertex w^* to add to X next.

(and we set $A[w^*]$ to $\text{key}[w^*]$)

Maintaining the Invariants

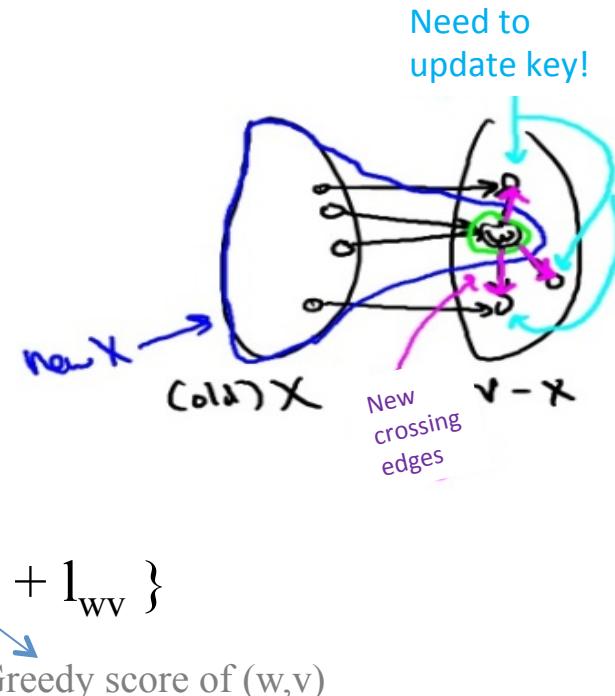
To maintain Invariant #2: [i.e., that $\forall v \notin X$

$\text{Key}[v] = \text{smallest Dijkstra greedy score of edge } (u,v) \text{ with } u \text{ in } X$]

When w extracted from heap (i.e., added to X)

- for each edge (w,v) in E:
 - if $v \in V-X$ (i.e., in heap)
 - delete v from heap
 - recompute $\text{key}[v] = \min \{\text{key}[v], A[w] + l_{wv} \}$
 - re-Insert v into heap

Key update



Running Time Analysis

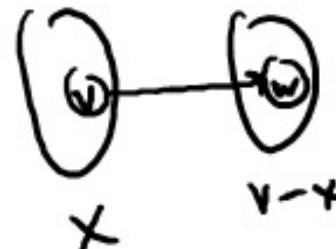
You check: dominated by heap operations. ($O(\log(n))$ each)

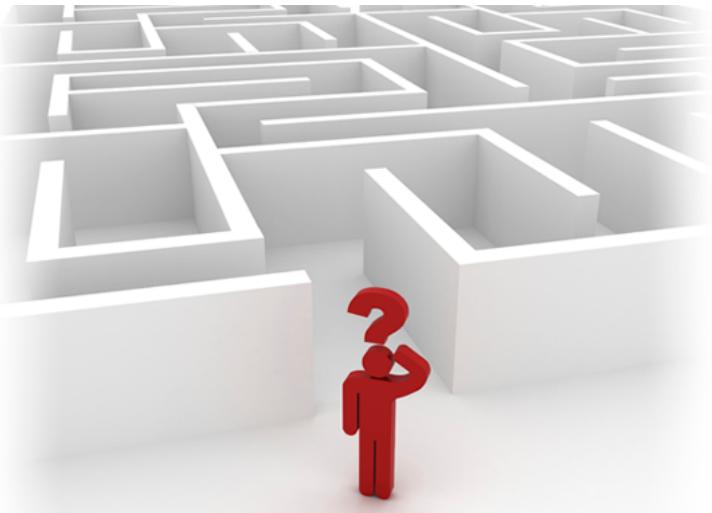
- $(n-1)$ Extract mins
- each edge (v,w) triggers at most one Delete/Insert combo
(if v added to X first)

So: # of heap operations in $O(n+m) = O(m)$

So: running time = $O(m \log(n))$ (like sorting)

Since graph is
weakly connected





Design and Analysis
of Algorithms I

Data Structures

Heaps and Their Applications

Heap: Supported Operations

- A container for objects that have keys
- Employer records, network edges, events, etc.

Insert: add a new object to a heap.

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

Equally well,
EXTRACT MAX

Extract-Min: remove an object in heap with a minimum key value. [ties broken arbitrarily]

Running time : $O(\log n)$ [$n = \#$ of objects in heap]

Also : **HEAPIFY** ($\binom{n \text{ batched Inserts}}{\text{in } O(n) \text{ time}}$), **DELETE**($O(\log(n))$ time)

Application: Sorting

Canonical use of heap : fast way to do repeated minimum computations.

Example : SelectionSort $\sim \theta(n)$ linear scans, $\theta(n^2)$ runtime on array of length n

Heap Sort : 1.) insert all n array elements into a heap
2.) Extract-Min to pluck out elements in sorted order

Running Time = $2n$ heap operations = $O(n \log(n))$ time.

=> optimal for a “comparison-based” sorting algorithm!

Application: Event Manager

“Priority Queue” – synonym for a heap.

Example : simulation (e.g., for a video game)

- Objects = event records [Action/update to occur at given time in the future]
- Key = time event scheduled to occur
- Extract-Min => yields the next scheduled event

Application: Median Maintenance

I give you : a sequence x_1, \dots, x_n of numbers, one-by-one.

You tell me : at each time step i , the median of $\{x_1, \dots, x_i\}$.

Constraint : use $O(\log(i))$ time at each step i .

Solution : maintain heaps H_{Low} : supports Extract Max

H_{High} : supports Extract Min

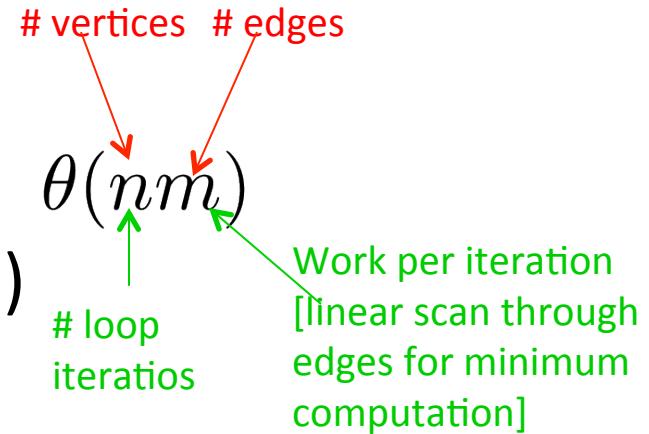
Key Idea : maintain invariant that $\sim i/2$ smallest (largest) elements in
 $H_{\text{Low}} (H_{\text{High}})$

You Check : 1.) can maintain invariant with $O(\log(i))$ work
2.) given invariant, can compute median in $O(\log(i))$ work

Application: Speeding Up Dijkstra

Dijkstra's Shortest-Path Algorithm

- Naïve implementation => runtime = $O(nm)$
- with heaps => runtime = $O(m \log(n))$





Design and Analysis
of Algorithms I

Data Structures

Heaps: Some Implementation Details

Heap: Supported Operations

- A container for objects that have keys
- Employer records, network edges, events, etc.

Insert: add a new object to a heap.

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

Equally well,
EXTRACT MAX

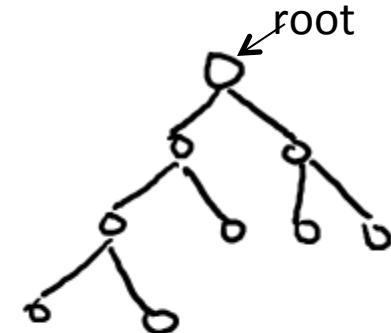
Extract-Min: remove an object in heap with a minimum key value. [ties broken arbitrarily]

Running time : $O(\log n)$ [$n = \#$ of objects in heap]

Also : **HEAPIFY** ($\binom{n \text{ batched Inserts}}{\text{in } O(n) \text{ time}}$), **DELETE**($O(\log(n))$ time)

The Heap Property

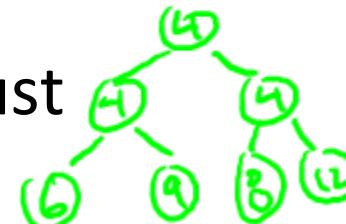
Conceptually : think of a heap as a tree.
-rooted, binary, as complete as possible



Heap Property: at every node x ,
 $\text{Key}[x] \leq$ all keys of x 's children

Consequence : object at root must have minimum key value (6)

alternatively



A heap

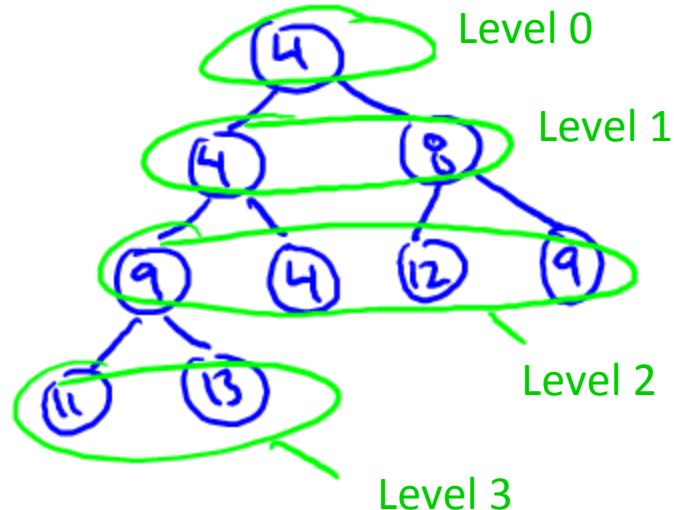


Array Implementation



Note : $\text{parent}(i) = i/2$ if i even
 $= [i/2]$ if i odd

i.e., round down



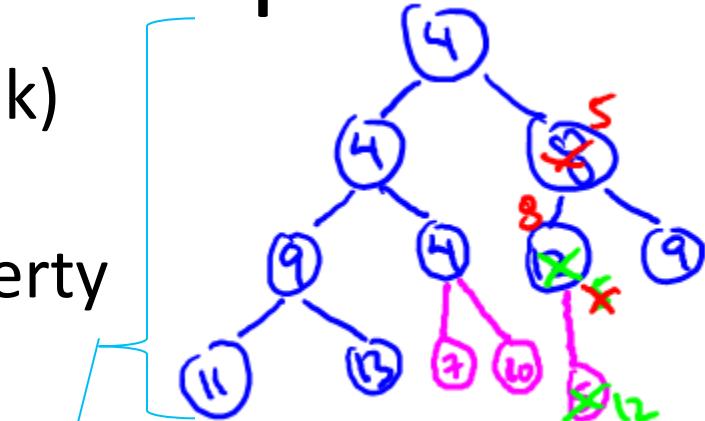
and children of i are $2i, 2i+1$

Insert and Bubble-Up

Implementation of Insert (given key k)

Step 1: stick k at end of last level.

Step 2 : Bubble-Up k until heap property is restored (i.e., key of k's parent is $\leq k$)



$\sim \log_2 n$ levels ($n = \#$ of items in heap)

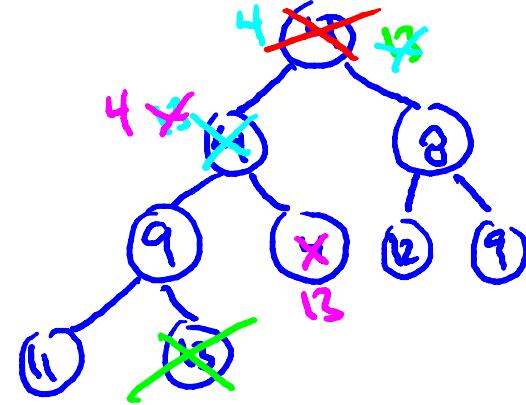
Check : 1.) bubbling up process must stop, with heap property restored
2.) runtime = $O(\log(n))$

Extract-Min and Bubble-Down

Implementation of Extract-Min

1. Delete root
2. Move last leaf to be new root.
3. Iteratively Bubble-Down until heap property has been restored

[always swap with smaller child!]



Check : 1.) only Bubble-Down once per level, halt with a heap
2.) run time = $O(\log(n))$



Data Structures

Introduction

Design and Analysis
of Algorithms I

Data Structures

Point : organize data so that it can be accessed quickly and usefully.

Examples : lists, stacks, queues, heaps, search trees, hashtables, bloom filters, union-find, etc.

Why so Many ? : different data structures support different sets of operations => suitable for different types of tasks.

Rule of Thumb : choose the “minimal” data structure that supports all the operations that you need.

Taking It To The Next Level

Level 0

- “what's a data structure ?”

Level 1

- cocktail party-level literacy

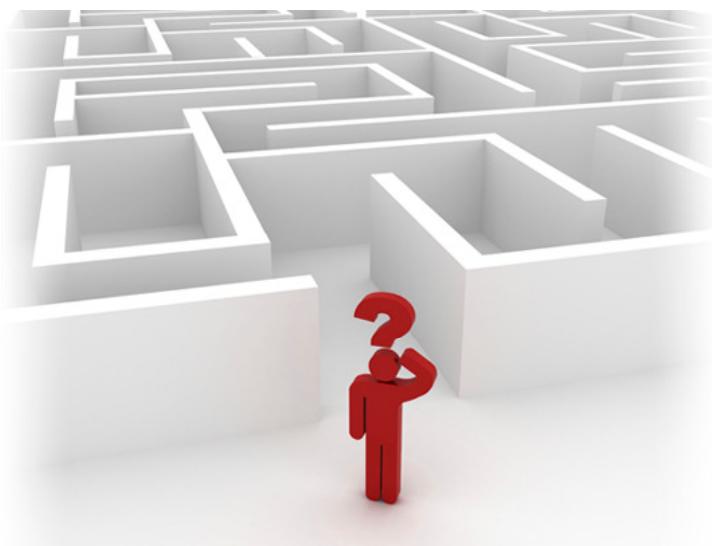
Level 2

- “this problem calls out for a heap”

Level 3

- “I only use data structures that I create myself”





Design and Analysis
of Algorithms I

Data Structures

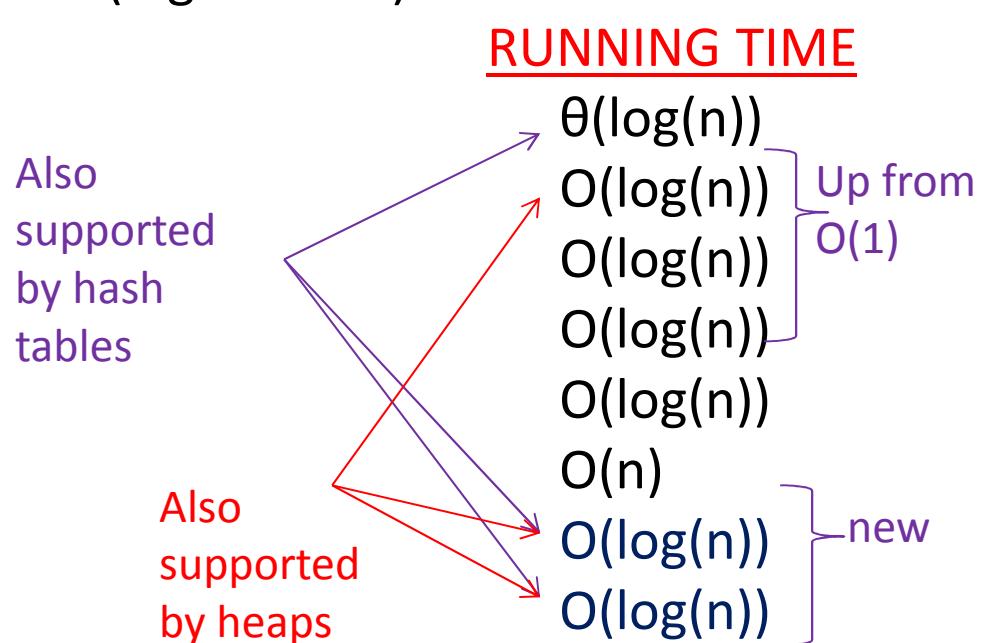
Binary Search
Tree Basics

Balanced Search Trees: Supported Operations

Raison d'etre : like sorted array + fast (logarithmic) inserts + deletes !

OPERATIONS

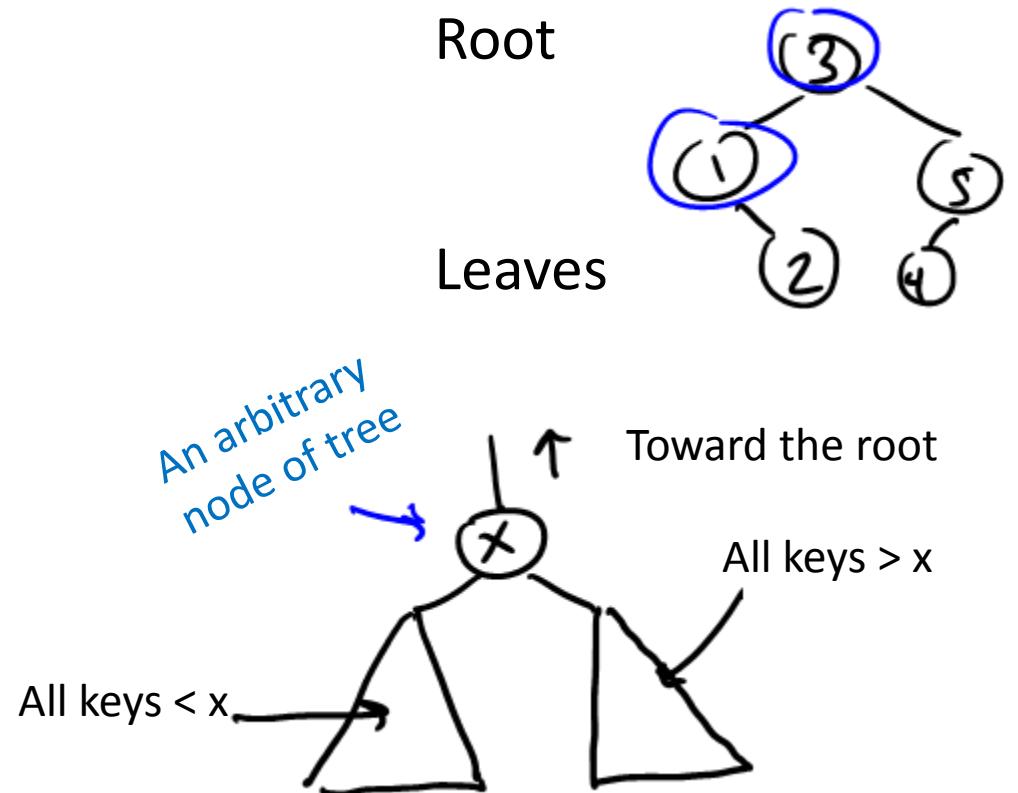
SEARCH
SELECT
MIN/MAX
PRED/SUCC
RANK
OUTPUT IN SORTED ORDER
INSERT
DELETE



Binary Search Tree Structure

- exactly one node per key
- most basic version :
 - each node has
 - left child pointer
 - right child pointer
 - parent pointer

SEARCH TREE PROPERTY :
(should hold at every node of the search tree)



Tim Roughgarden

The Height of a BST

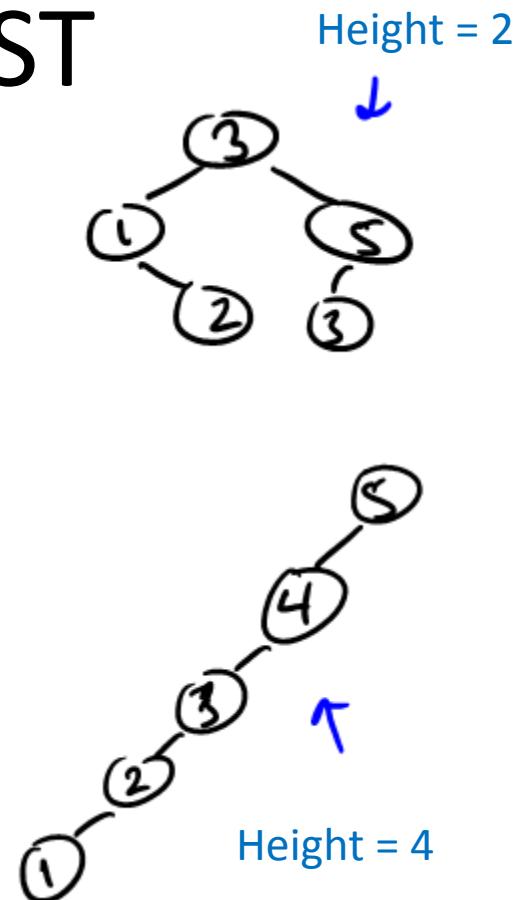
Note : many possible trees for a set of keys.

Note : height could be anywhere from $\sim \log_2 n$ to $\sim n$

(aka depth) longest root-leaf path

Worst case, a chain

Best case, perfectly balanced



Tim Roughgarden

Searching and Inserting

To Search for key k in tree T

- start at the root
- traverse left / right child pointers as needed

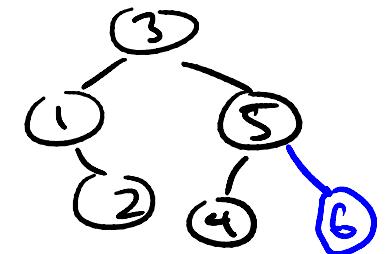
If $k <$ key at current node If $k >$ key at current node

- return node with key k or NULL, as appropriate

To Insert a new key k into a tree T

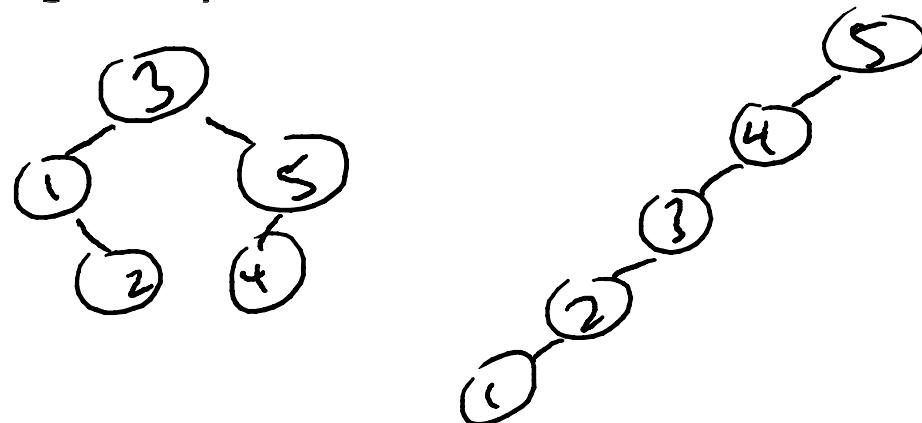
- search for k (unsuccessfully)
- rewire final NULL ptr to point to new node with key k

Exercise :
preserves
search tree
property!



The worst-case running time of Search (or Insert) operation in a binary search tree containing n keys is...?

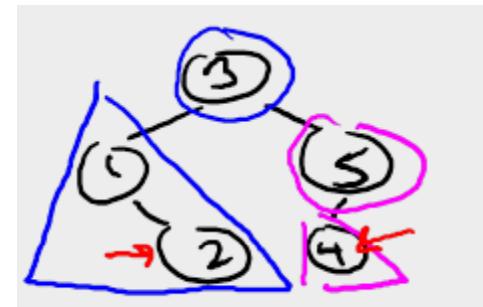
- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(\text{height})$
- $\theta(n)$



Min, Max, Pred, And Succ

To compute the minimum (maximum) key of a tree

- Start at root
- Follow left child pointers (right ptrs, for maximum) until you can't anymore (return last key found)



To compute the predecessor of key k

- Easy case : If k's left subtree nonempty, return max key in left subtree
- Otherwise : follow parent pointers until you get to a key less than k.

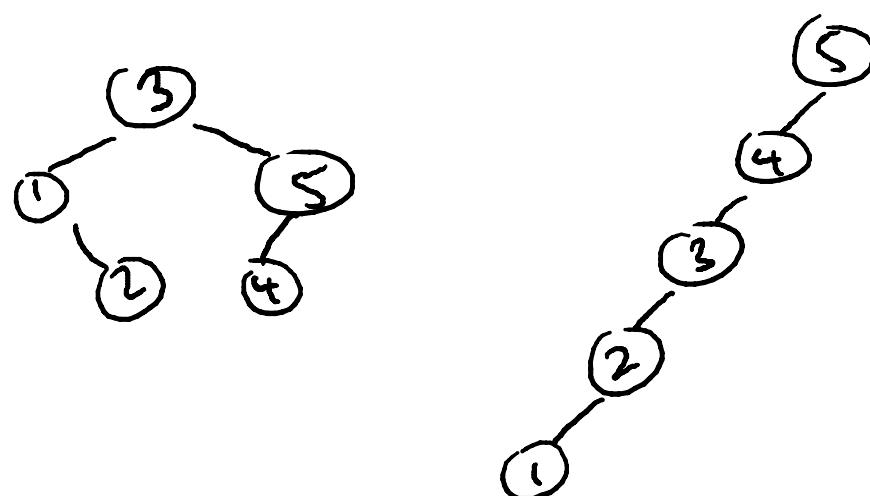
Happens first time you "turn left"

Exercise :
prove this
works

Tim Roughgarden

The worst-case running time of the Max operation in a binary search tree containing n keys is...?

- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(\text{height})$
- $\theta(n)$



In-Order Traversal

TO PRINT OUT KEYS IN INCREASING ORDER

-Let r = root of search tree, with subtrees TL and TR

- recurse on TL

[by recursion (induction) prints out keys of TL
in increasing order]

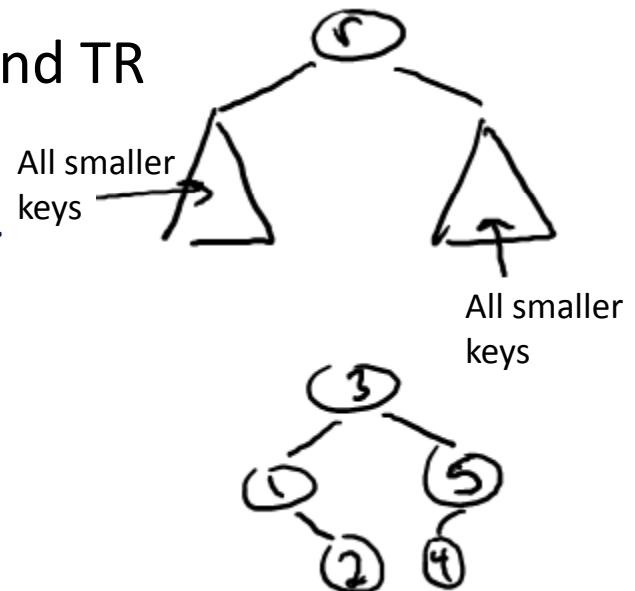
-Print out r 's key

RUNNING TIME

$O(1)$ time, n recursive
calls $\Rightarrow O(n)$ total

-Recurse on TR

[prints out keys of TR in increasing order]



Deletion

TO DELETE A KEY K FROM A SEARCH TREE

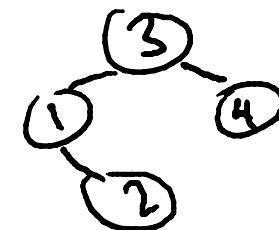
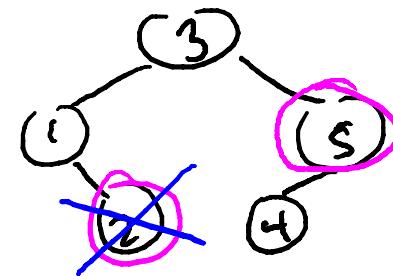
- SEARCH for k

EASY CASE (k's node has no children)

-Just delete k's node from tree, done

MEDIUM CASE (k's node has one child)

(unique child assumes position
previously held by k's node)



Deletion (con'd)

DIFFICULT CASE (k's node has 2 children)

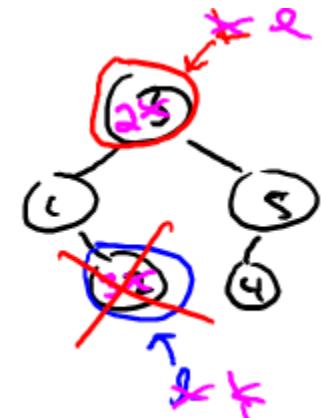
-Compute k's predecessor l

[i.e., traverse k's (non-NULL) left child ptr, then right child ptrs until no longer possible]

- SWAP k and l !

NOTE : in it's new position, k has no right child !

=> easy to delete or splice out k's new node



RUNNING

TIME :

$\Theta(\text{height})$

Exercise : at end, have a valid search tree !

Select and Rank

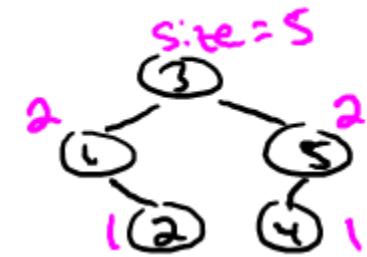
Idea : store a little bit of extra info at each tree node about the tree itself (i.e., not about the data)

Example Augmentation : $\text{size}(x) = \# \text{ of tree nodes in subtree rooted at } x$.

Note : if x has children y and z ,
then $\text{size}(y) + \text{size}(z) + 1$

↑ ↑ ↑
Population in Right subtree x itself
left subtree

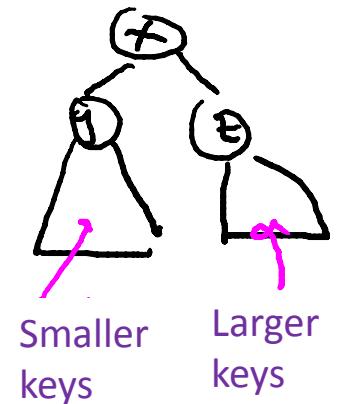
Also : easy to keep sizes up-to-date during an Insertion or Deletion (you check!)



Select and Rank (con'd)

HOW TO SELECT i^{th} ORDER STATISTIC FROM
AUGMENTED SEARCH TREE (with subtree sizes)

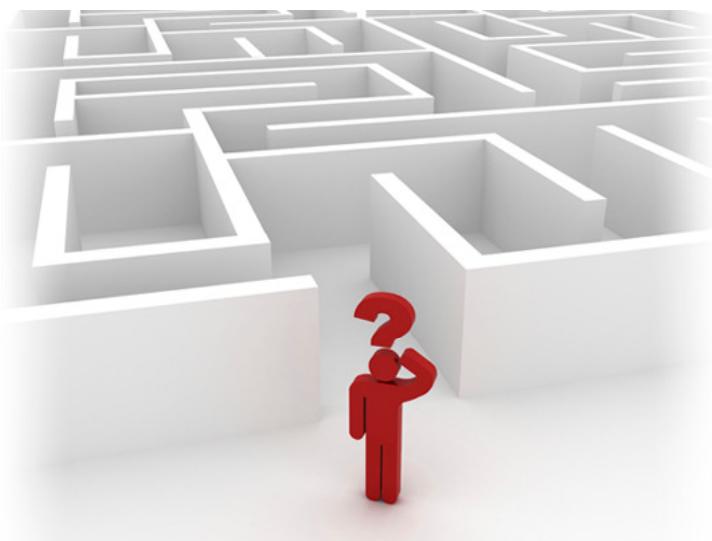
- start at root x , with children y and z
- let $a = \text{size}(y)$ [$a = 0$ if x has no left child]
- if $a = i-1$, return x 's key
- if $a \geq i$, recursively compute i^{th} order statistic of search tree rooted at y
- if $a < i-1$ recursively compute $(i-a-1)^{\text{th}}$ order statistic of search tree rooted at z



RUNNING TIME = $\Theta(\text{height})$.

[EXERCISE : how to implement RANK ?

Tim Roughgarden

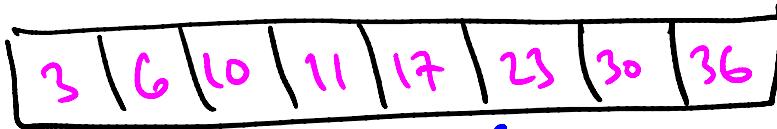


Design and Analysis
of Algorithms I

Data Structures

Balanced Search
Trees: Supported
Operations

Sorted Arrays: Supported Operations



OPERATIONS

SEARCH

SELECT (given order statistic I)

MIN/MAX

PRED/SUCC (given pointer to a key)

RANK (i.e., # of keys less than or equal to
a given value)

OUTPUT IN SORTED ORDER

BUT WHAT ABOUT
INSERTIONS + DELETIONS ?
(would take $\Theta(n)$ time)

RUNNING TIME

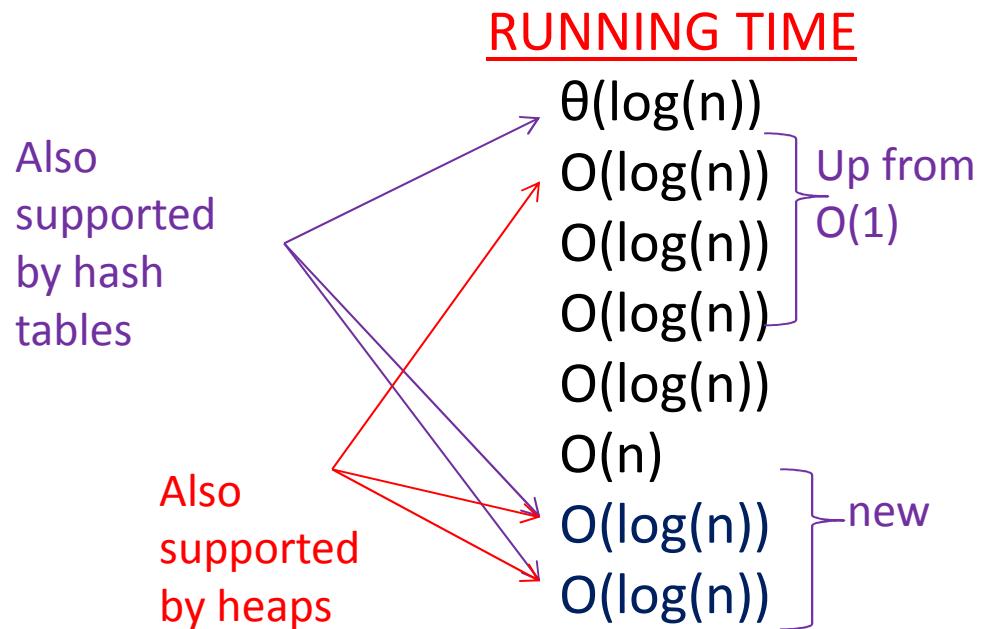
 $\Theta(\log(n))$ $O(1)$ $O(1)$ $O(1)$ $\Theta(\log(n))$ $O(n)$

Balanced Search Trees: Supported Operations

Raison d'etre : like sorted array + fast (logarithmic) inserts + deletes !

OPERATIONS

SEARCH
SELECT
MIN/MAX
PRED/SUCC
RANK
OUTPUT IN SORTED ORDER
INSERT
DELETE



Tim Roughgarden



Design and Analysis
of Algorithms I

Data Structures

Insertion In A Red-Black Tree

High-Level Plan

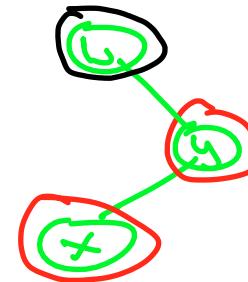
Idea for Insert / Delete: proceed as in a normal binary search tree, then recolor and/or perform rotations until invariants are restored.

Insert(x): ① insert x as usual
(makes x a leaf)

② try coloring x red

③ if x's parent y is black, done.

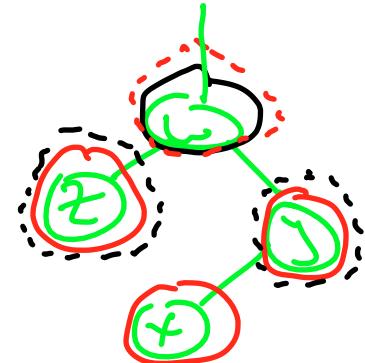
④ else y is red. \Rightarrow y has a black parent w



Insertion

Case 1

Case 1: the other child z of x' 's grand parent w is also red.



\Rightarrow recolor y, z black and w red

[key point: does not break invariant ④]

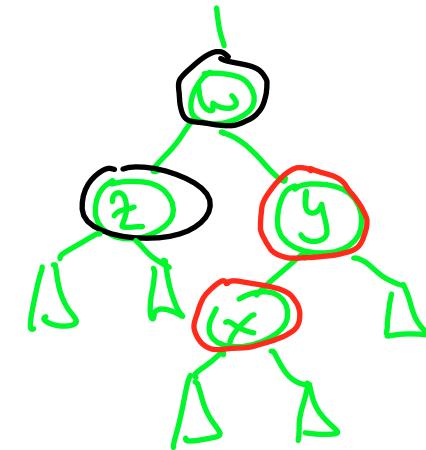
\Rightarrow either restores invariant ③ or propagates the double red upward

\Rightarrow can only happen $O(\log n)$ times

If you reach the root, recolor it black \Rightarrow preserves invariant ④

Case 2

Case 2: let x, y be the current double-red, & the deeper node.
let $w = x$'s grand parent.
Suppose w 's other child ($\neq y$) is NULL
or is a black node z .



Exercise / Case analysis (details omitted): Can eliminate double-red [\Rightarrow all invariants satisfied] in $O(1)$ time via 2-3 rotations + recolorings.



Design and Analysis
of Algorithms I

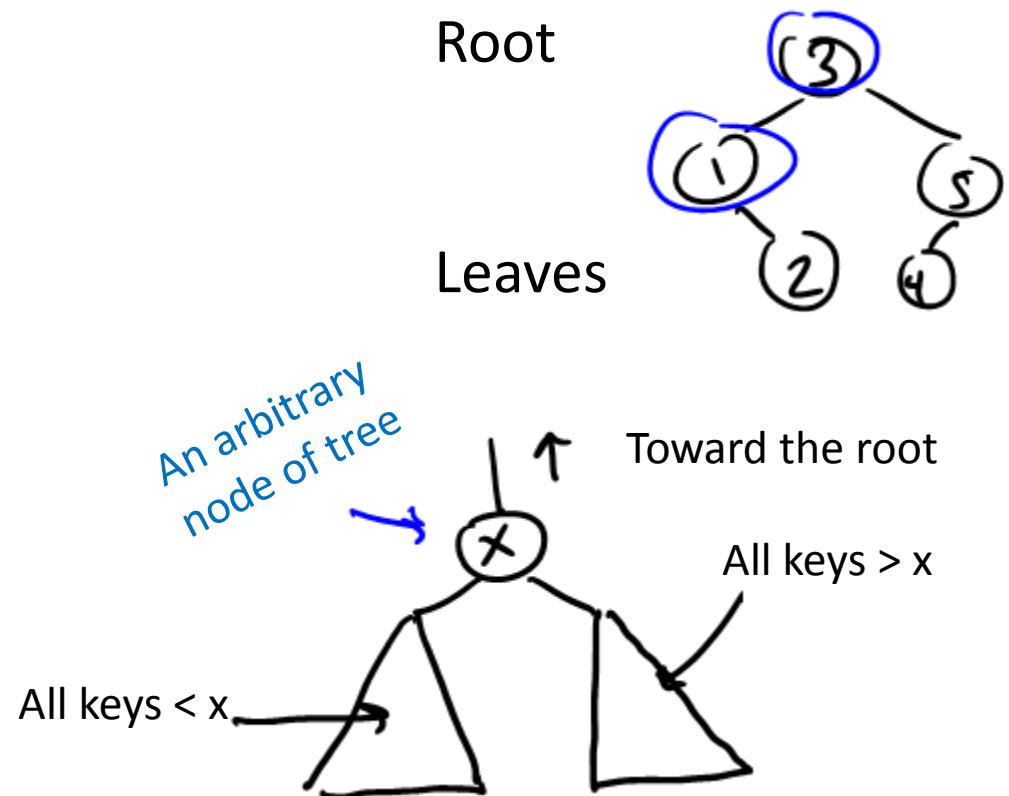
Data Structures

Red-Black Trees

Binary Search Tree Structure

- exactly one node per key
- most basic version :
 - each node has
 - left child pointer
 - right child pointer
 - parent pointer

SEARCH TREE PROPERTY :
(should hold at every node of the search tree)



The Height of a BST

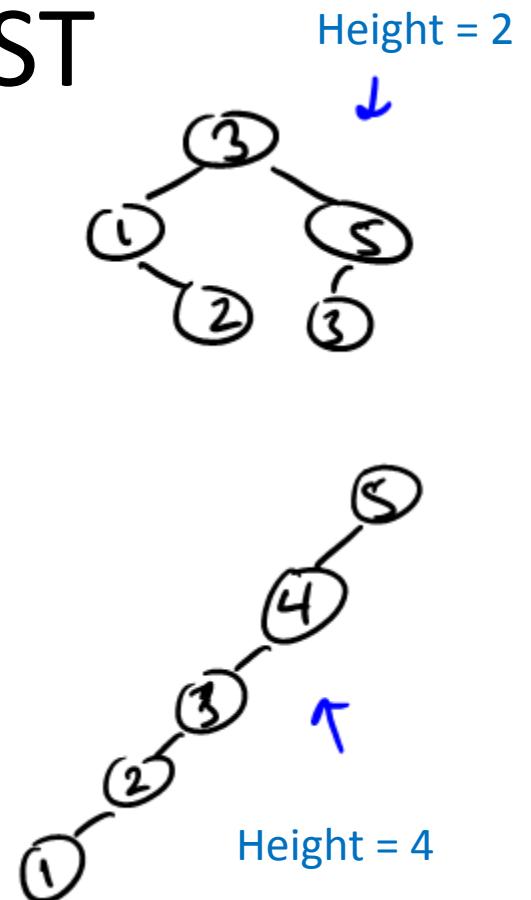
Note : many possible trees for a set of keys.

Note : height could be anywhere from $\sim \log_2 n$ to $\sim n$

(aka depth) longest root-leaf path

Worst case, a chain

Best case, perfectly balanced



Balanced Search Trees

Idea : ensure that height is always $O(\log(n))$ [best possible]

⇒ Search / Insert / Delete / Min / Max / Pred / Succ will then run in $O(\log(n))$ time [$n = \#$ of keys in tree]

Example : red-black trees [Bayes '72, Guibas-Sedgewick '78]

[see also AUL trees, splay trees, B trees]

Red-Black Invariants

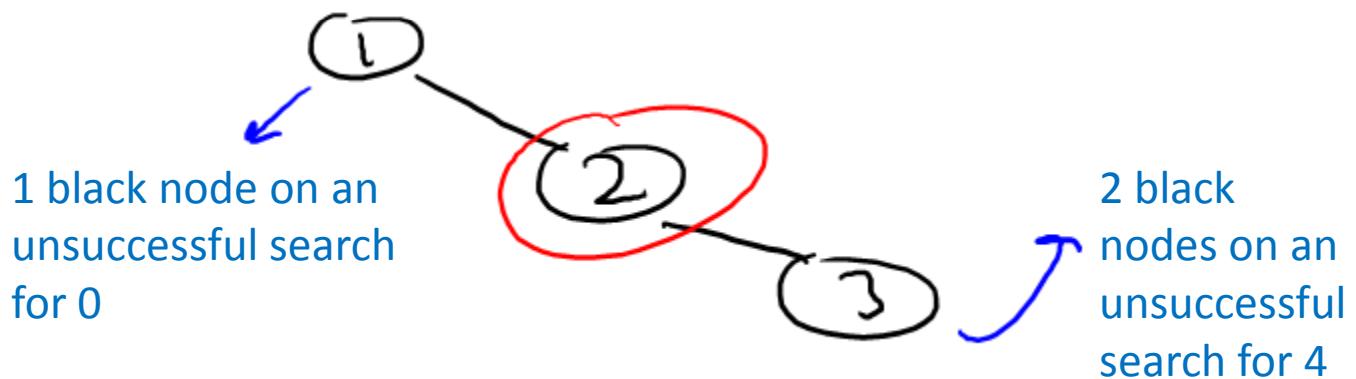
1. Each node red or black
2. Root is black
3. No 2 reds in a row
[red node => only black children]
4. Every root-NULL path has same number of black nodes

Like in an
unsuccessful search

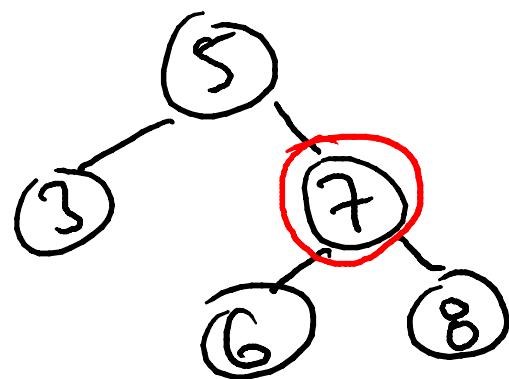
Example #1

Claim : a chain of length 3 cannot be a red-black tree

Proof



Example #2

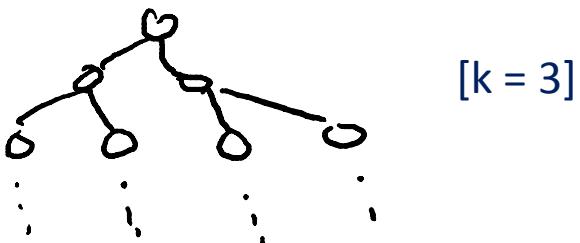


Height Guarantee

Claim : every red-black tree with n nodes has height $\leq 2 \log_2(n + 1)$

Proof : Observation : if every root-NUL path has $\geq k$ nodes, then tree includes (at the top) a perfectly balanced search tree of depth $k-1$.

=> Size n of the tree
must Be at least $2^k - 1$



Height Guarantee (con'd)

Story so far : size $n \geq 2^k - 1$, where $k = \text{minimum } \# \text{ of nodes on root - NULL path}$

$$\Rightarrow k \leq \log_2(n + 1)$$

Thus : in a red-black tree with n nodes, there is a root-NULL path with at most $\log_2(n + 1)$ black nodes.

By 4th Invariant : every root-NULL path has $\leq \log_2(n + 1)$ black nodes

By 3rd Invariant : every root-NULL path has $\leq 2 \log_2(n + 1)$ total nodes.

Q.E.D.

Which of the search tree operations have to be re-implemented so that the Red-Black invariants are maintained?

- Search
- Delete
- Insert and Delete
- None of the above



Design and Analysis
of Algorithms I

Data Structures

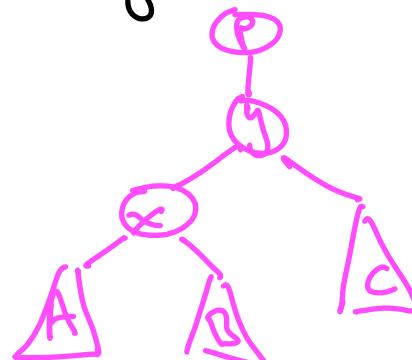
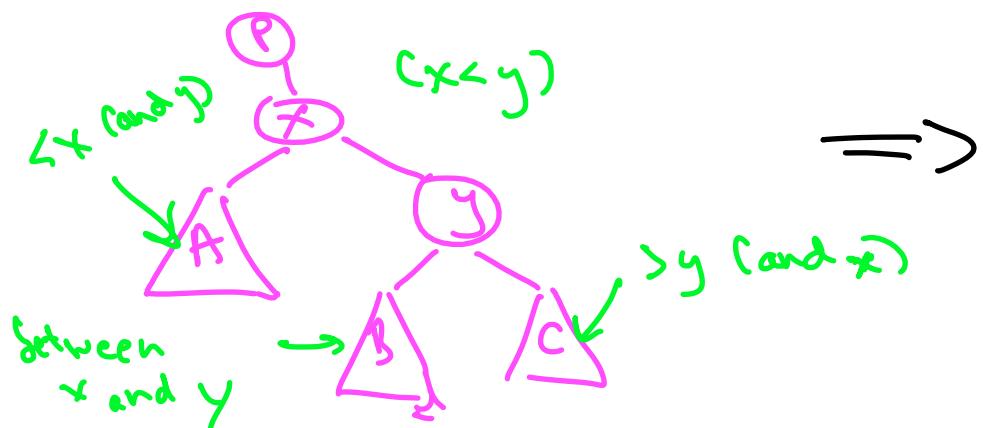
Rotations

Left Rotations

key primitive: rotations. (common to all balanced search tree implementations - red-black, AVL, B+, etc.)

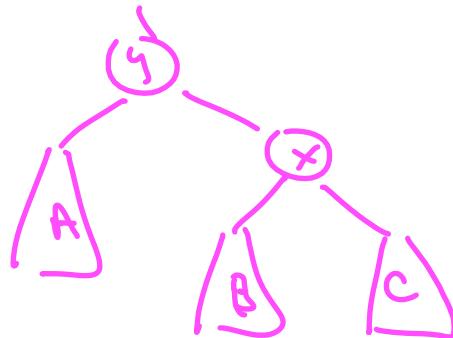
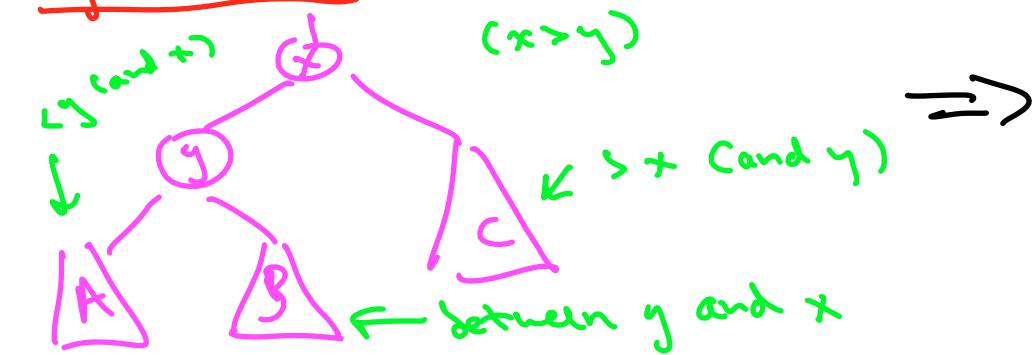
Idea: locally rebalance subtrees at a node in our tree.

Left rotation: (of a parent x and right child y)



Right Rotations

Right rotation:



Nice properties: Search tree property maintain, can implement in $O(1)$ time.



Design and Analysis
of Algorithms I

Data Structures

Hash Tables and Applications

Hash Table: Supported Operations

Purpose : maintain a (possibly evolving) set of stuff.
(transactions, people + associated data, IP addresses, etc.)

Insert : add new record

Using a “key”

Delete : delete existing record

AMAZING
GUARANTEE

Lookup : check for a particular record
(a “dictionary”)

All operations in
 $O(1)$ time ! *

* 1. properly implemented 2. non-pathological data

Application: De-Duplication

Given : a “stream” of objects.

- Linear scan through a huge file
- Or, objects arriving in real time

Goal : remove duplicates (i.e., keep track of unique objects)

- e.g., report unique visitors to web site
- avoid duplicates in search results

Solution : when new object x arrives

- lookup x in hash table H
- if not found, Insert x into H

Application: The 2-SUM Problem

Input : unsorted array A of n integers. Target sum t.

Goal : determine whether or not there are two numbers x,y in A with

$$x + y = t$$

Naïve Solution : $\theta(n^2)$ time via exhaustive search

Better : 1.) sort A ($\theta(n \log n)$ time) 2.) for each x in A, look for

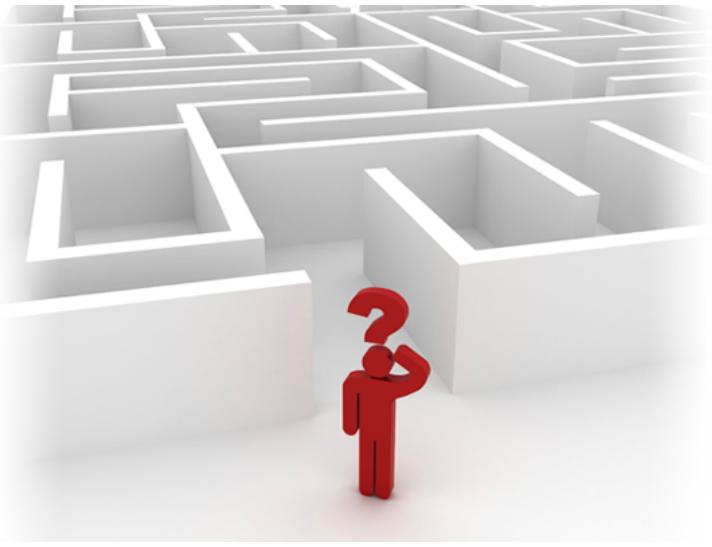
$\theta(n)$ time $\theta(n \log n)$  t-x in A via binary search

Amazing : 1.) insert elements of A
 into hash table H

2.) for each x in A,
 Lookup t-x  $\theta(n)$ time

Further Immediate Applications

- Historical application : symbol tables in compilers
- Blocking network traffic
- Search algorithms (e.g., game tree exploration)
 - Use hash table to avoid exploring any configuration (e.g., arrangement of chess pieces) more than once
- etc.



Design and Analysis
of Algorithms I

Data Structures

Hash Tables: Some Implementation Details

Hash Table: Supported Operations

Purpose : maintain a (possibly evolving) set of stuff.
(transactions, people + associated data, IP addresses, etc.)

Insert : add new record

Using a “key”

Delete : delete existing record

AMAZING
GUARANTEE

Lookup : check for a particular record
(a “dictionary”)

All operations in
 $O(1)$ time ! *

* 1. properly implemented 2. non-pathological data

High-Level Idea

Setup : universe U [e.g., all IP addresses, all names, all chessboard configurations, etc.]
[generally, REALLY BIG]

Goal : want to maintain evolving set $S \subseteq U$
[generally, of reasonable size]

Solution : 1.) pick $n = \#$ of “buckets” with
(for simplicity assume $|S|$ doesn’t vary much)

- 2.) choose a hash function $h : U \rightarrow \{0, 1, 2, \dots, n - 1\}$
- 3.) use array A of length n , store x in $A[h(x)]$

Naïve Solutions

1. Array-based solution
[indexed by u]
- $O(1)$ operations
but $\theta(|U|)$ space
2. List-based solution
- $\theta(|S|)$ space but
 $\theta(|S|)$ Lookup

Consider n people with random birthdays (i.e., with each day of the year equally likely). How large does n need to be before there is at least a 50% chance that two people have the same birthday?

- 23 50 %
- 57 99 %
- 184 99.99....%
- 367 100%

BIRTHDAY “PARADOX”

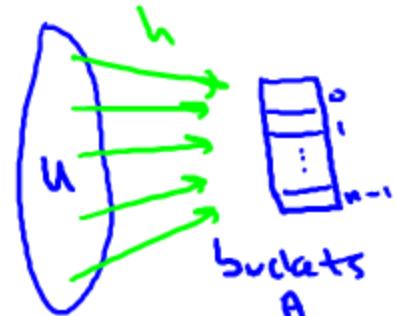
Resolving Collisions

Collision: distinct $x, y \in U$ such that $h(x) = h(y)$

Solution # 1 : (separate) chaining

- keep linked list in each bucket
- given a key/object x , perform Insert/Delete/Lookup in the list in $A[h(x)]$

Linked list for x Bucket for x



Solution #2 : open addressing. (only one object per bucket)

- Hash function now specifies probe sequence $h_1(x), h_2(x), \dots$
(keep trying till find open slot)
- Examples : linear probing (look consecutively), double hashing

Use 2 hash functions

What Makes a Good Hash Function?

Note : in hash table with chaining, Insert is $\theta(1)$
 $\theta(\text{list length})$ for Insert/Delete.

Insert new object x at
front of list in $A[h(x)]$

could be anywhere from m/n to m for m objects

Equal-length lists

Point : performance depends on the choice of hash function!
(analogous situation with open addressing)

All
objects in
same
bucket

Properties of a “Good” Hash function

1. Should lead to good performance => i.e., should “spread data out” (gold standard – completely random hashing)
2. Should be easy to store/ very fast to evaluate.

Bad Hash Functions

Example : keys = phone numbers (10-digits).

$$|u| = 10^{10}$$

-Terrible hash function : $h(x) = 1^{\text{st}} \ 3 \text{ digits of } x$

$$\text{choose } n = 10^3$$

(i.e., area code)

- mediocre hash function : $h(x) = \text{last 3 digits of } x$

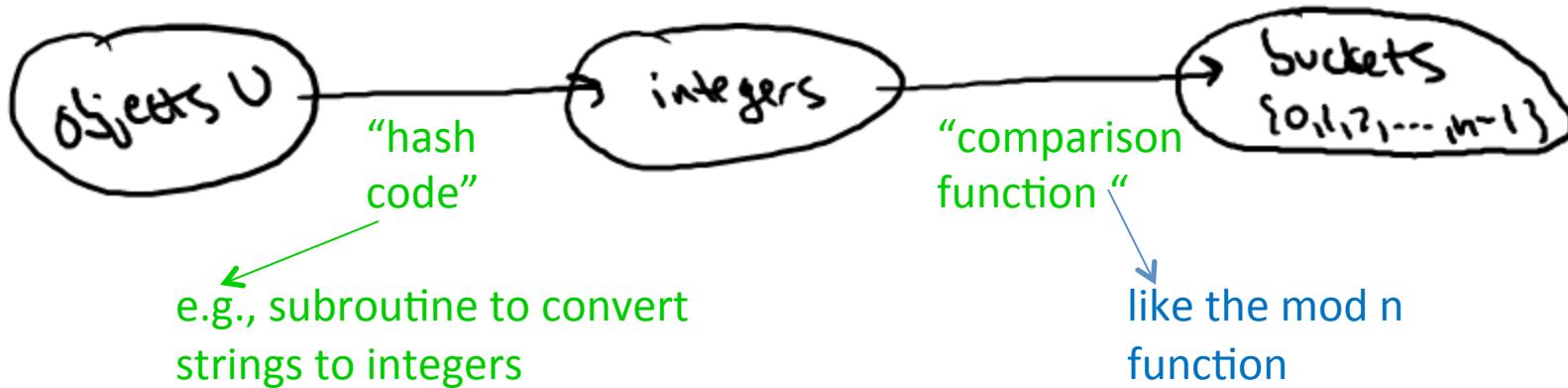
[still vulnerable to patterns in last 3 digits]

Example : keys = memory locations. (will be multiples of a power of 2)

-Bad hash function : $h(x) = x \bmod 1000$ (again $n = 10^3$)

=> All odd buckets guaranteed to be empty.

Quick-and-Dirty Hash Functions



How to choose $n = \#$ of buckets

1. Choose n to be a prime (within constant factor of # of objects in table)
2. Not too close to a power of 2
3. Not too close to a power of 10



Design and Analysis
of Algorithms I

Data Structures

Universal Hash Functions: Definition and Example

Overview of Universal Hashing

Next : details on randomized solution (in 3 parts).

Part 1 : proposed definition of a “good random hash function”.
 (“universal family of hash functions”)

Part 3 : concrete example of simple + practical such functions

Part 4 : justifications of definition : “good functions” lead to “good performance”

Universal Hash Functions

Definition : Let H be a set of hash functions from U to $\{0,1,2,\dots,n-1\}$

H is universal if and only if :

for all x,y in U (with $x \neq y$)

$$Pr_{h \in H}[x, y \text{ collide}] \leq \frac{1}{n} \quad (n = \# \text{ of buckets})$$

ie., $h(x) = h(y)$

When h is chosen uniformly at random from H .
(i.e., collision probability as small as with “gold standard” of perfectly random hashing)

Consider a hash function family H , where each hash function of H maps elements from a universe U to one of n buckets. Suppose H has the following property: for every bucket i and key k , a $1/n$ fraction of the hash functions in H map k to i . Is H universal ?

- Yes, always.
- No, never.
- Maybe yes, maybe no (depends on the H).
Only if the hash table is implemented using chaining.

Yes : Take $H = \text{all functions from } U \text{ to } \{0,1,2,\dots,n-1\}$

No : Take $H = \text{the set of } n \text{ different constant functions}$

Example: Hashing IP Addresses

Let $U = \text{IP addresses (of the form } (x_1, x_2, x_3, x_4), \text{ with each } x_i \in \{0, 1, 2, \dots, 255\}\}$

Let $n = \text{a prime (e.g., small multiple of # of objects in HT)}$

Construction : Define one hash function h_a per 4-tuple $a = (a_1, a_2, a_3, a_4)$ with each $a_i \in \{0, 1, 2, 3, \dots, n - 1\}$

Define : $h_a : \text{IP addrs} \rightarrow \text{buckets by } n^4 \text{ such functions}$

$$h_a(x_1, x_2, x_3, x_4) = \begin{pmatrix} a_1x_1 + a_2x_2 + \\ a_3x_3 + a_4x_4 \end{pmatrix} \text{ mod } n$$

A Universal Hash Function

Define : $H = \{h_a | a_1, a_2, a_3, a_4 \in \{0, 1, 2, \dots, n - 1\}\}$

$$h_a(x_1, x_2, x_3, x_4) = \begin{pmatrix} a_1x_1 + a_2x_2 + \\ a_3x_3 + a_4x_4 \end{pmatrix} \text{ mod } n$$

Theorem: This family is universal

Proof (Part I)

Consider distinct IP addresses $(x_1, x_2, x_3, x_4), (y_1, y_2, y_3, y_4)$.

Assume : $x_4 \neq y_4$

Question : collision probability ?

(i.e., $\text{Prob}_{h_a \in H} [h_a(x_1, \dots, x_4) = h_a(y_1, \dots, y_4)]$)

Note : collision \Leftrightarrow

$$a_1x_1 + a_2 + x_2 + a_3 + x_3 + a_4x_4 = a_1y_1 + a_2 + y_2 + a_3 + y_3 + a_4 + y_4 \pmod{n}$$

$$\Leftrightarrow a_4(x_4 - y_4) = \sum_{i=1}^3 a_i(y_i - x_i) \pmod{n}$$

Next : condition on random choice of a_1, a_2, a_3 . (a_4 still random)

Proof (Part II)

The Story So Far : with a_1, a_2, a_3 fixed arbitrarily, how many choices of a_4 satisfy

$$a_4(x_4 - y_4) = \sum_{i=1}^3 a_i(y_i - x_i) \pmod{n}$$

Still random

$\Leftrightarrow x, y$ collide under h_a

Some fixed number in
 $\{0, 1, 2, \dots, n-1\}$

Key Claim : left-hand side equally likely to be any of $\{0, 1, 2, \dots, n-1\}$

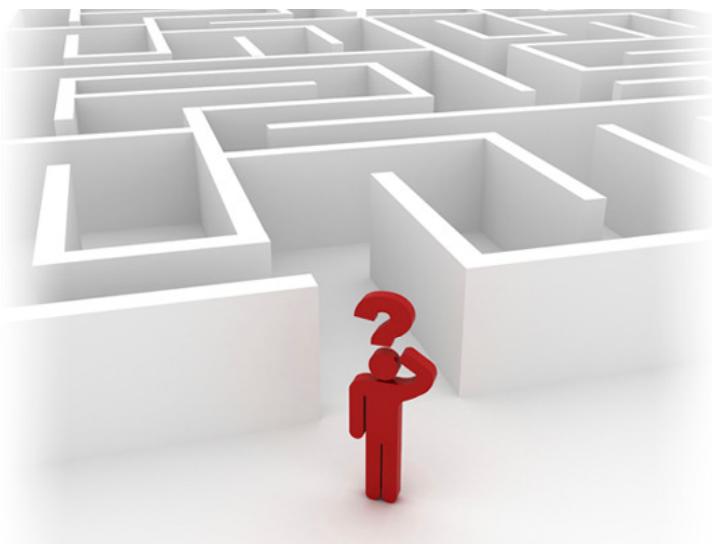
Reason : $x_4 \neq y_4$ ($x_4 - y_4 \neq 0 \pmod{n}$)
 n is prime, a_4 uniform at random

[addendum : make sure n bigger than the maximum value of an a_i]

\rightarrow Implies $\text{Prob}[h_a(x) = h_a(y)] = 1/n$

“Proof” by example : $n = 7$, $x_4 - y_4 = 2$ or $3 \pmod{n}$

Q.E.D.



Design and Analysis
of Algorithms I

Data Structures

Universal Hash Functions: Motivation

Hash Table: Supported Operations

Purpose : maintain a (possibly evolving) set of stuff.
(transactions, people + associated data, IP addresses, etc.)

Insert : add new record

Delete : delete existing record
easier/more common with chaining than open addressing

Lookup : check for a particular record
(a “dictionary”)

Using a “key”

AMAZING
GUARANTEE

All operations in
 $O(1)$ time ! *

* 1. properly implemented 2. non-pathological data

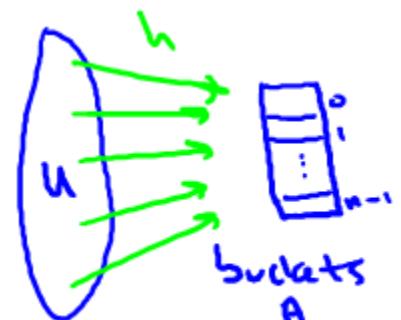
Resolving Collisions

Collision : distinct $x, y \in U$ such that $h(x) = h(y)$.

Solution#1: (separate) chaining.

- keep linked list in each bucket
- given a key/object x , perform Insert/Delete/Lookup in the list in $A[h(x)]$

bucket for x
linked list for x



Solution#2 : open addressing. (only one object per bucket)

- hash function now specifies probe sequence $h_1(x), h_2(x), \dots$
(keep trying till find open slot)

use 2 hash functions

- examples : linear probing (look consecutively), double hashing



The Load of a Hash Table

Definition : the load factor of a hash table is

$$\alpha := \frac{\text{\# of objects in hash table}}{\text{\# of buckets of hash table}}$$

Which hash table implementation strategy is feasible for load factors larger than 1?

- Both chaining and open addressing
- Neither chaining nor open addressing
- Only chaining
- Only open addressing

The Load of a Hash Table

Definition : the load factor of a hash table is

$$\alpha := \frac{\text{# of objects in hash table}}{\text{# of buckets of hash table}}$$

Note : 1.) $\alpha = O(1)$ is necessary condition for operations to run in constant time.
2.) with open addressing, need $\alpha \ll 1$.

Upshot#1 : good HT performance, need to control load.

Tim Roughgarden

Pathological Data Sets

Upshot#2 : for good HT performance, need a good hash function.

Ideal : user super-clever hash function guaranteed
to spread every data set out evenly.

Problem : DOES NOT EXIST! (for every hash function, there is a
pathological data set)

Reason : fix a hash function $h : U \rightarrow \{0,1,2,\dots,n-1\}$

\Rightarrow a la Pigeonhole Principle, there exist bucket i such that at least
 $|U|/n$ elements of U hash to i under h .



\Rightarrow if data set drawn only from these,
everything collides !

Tim Roughgarden

Pathological Data in the Real World

Preference : Crosby and Wallach, USENIX 2003.

Main Point : can paralyze several real-world systems (e.g., network intrusion detection) by exploiting badly designed hash functions.

- open source
- overly simplistic hash function

(easy to reverse engineer a pathological data set)

Solutions

1. Use a cryptographic hash function (e.g., SHA-2)
-- infeasible to reverse engineer a pathological data set

2. Use randomization. ←In next 2 videos
-- design a family H of hash functions such that for all
data sets S , “almost all” functions $h \in H$ spread S
out “pretty evenly”.
(compare to QuickSort guarantee)

Overview of Universal Hashing

Next : details on randomized solution (in 3 parts).

Part 1 : proposed definition of a “good random hash function”.
 (“universal family of hash functions”)

Part 3 : concrete example of simple + practical such functions

Part 4 : justifications of definition : “good functions” lead to “good performance”



Design and Analysis
of Algorithms I

Data Structures

Universal Hash
Functions: Performance
Guarantees (Chaining)

Overview of Universal Hashing

Next : details on randomized solution (in 3 parts).

Part 1 : proposed definition of a “good random hash function”.
 (“universal family of hash functions”)

Part 3 : concrete example of simple + practical such functions

Part 4 : justifications of definition : “good functions” lead to “good performance”

Universal Hash Functions

Definition : Let H be a set of hash function from U to $\{0,1,2,\dots,n-1\}$

H is universal if and only if :

For all $x, y \in U$ (with $x \neq y$)

$$Pr_{h \in H}[x, y \text{ collide}; h(x) = h(y)] \leq 1/n \quad (\text{n} = \# \text{ of buckets})$$

When h is chosen uniformly at random at random from H .
(i.e., collision probability as small as with “gold standard” of perfectly random hashing)

Chaining: Constant-Time Guarantee

Scenario : hash table implemented with chaining. Hash function h chosen uniformly at random from universal family H .

Theorem : [Carter-Wegman 1979]

All operations run in $O(1)$ time. (for every data set S)

Caveats : 1.) in expectation over the random choice of the hash function h . ($h = \# \text{ of buckets}$)

2.) assumes $|S| = O(n)$ [i.e., load $\alpha = \frac{|S|}{n} = O(1)$]

3.) assumes takes $O(1)$ time to evaluate hash function

Tim Roughgarden

Proof (Part I)

Will analyze an unsuccessful Lookup
(other operations only faster).

So : Let S = data set with $|S| = O(n)$ # of buckets
Consider Lookup for $x \notin S$ (arbitrary data set S)

Running Time : $O(1) + O(\text{list length in } A[h(x)])$

Compute
 $h(x)$

Traverse
list

L

A random variable,
depends on hash
function h

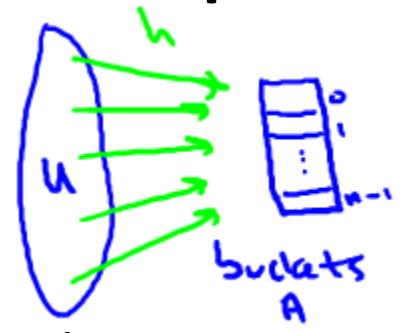
A General Decomposition Principle

Collision : distinct $x, y \in U$ such that $h(x) = h(y)$.

Solution#1: (separate) chaining.

- keep linked list in each bucket
- given a key/object x , perform Insert/Delete/Lookup in the list in $A[h(x)]$

bucket for x
linked list for x



Solution#2 : open addressing. (only one object per bucket)

- hash function now specifies probe sequence $h_1(x), h_2(x), \dots$
(keep trying till find open slot)

use 2 hash functions

- examples : linear probing (look consecutively), double hashing

Tim Roughgarden



Proof (Part II)

Let $L = \text{list length in } A[h(x)]$.

For $y \in S$ (so, $y \neq x$) define $z_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{otherwise} \end{cases}$

Note : $L = \sum_{y \in S} A_y$

So : $E[L] = \sum_{y \in S} E[Z_y]$

Recall

$$z_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{otherwise} \end{cases}$$

What does $E[z_y]$ evaluate to?

$$E[z_y] = 0 \cdot Pr[z_y = 0] + 1 \cdot Pr[z_y = 1]$$

- $\Pr[h(y) = 0]$
- $\Pr[h(y) \neq x]$
- $\Pr[h(y) = h(x)]$
- $\Pr[h(y) \neq h(x)]$

Proof (Part II)

Let $L = \text{list length in } A[h(x)]$.

For $y \in S$ (so, $y \neq x$) define $z_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{otherwise} \end{cases}$

Note : $L = \sum_{y \in S} A_y$

So : $E[L] = \sum_{y \in S} E[Z_y] = \sum_{y \in S} Pr[h(y) = h(x)]$

Which of the following is the smallest valid upper bound on
 $\Pr[h(y) = h(x)]$?

- $1/n^2$
- $1/n$
- $1/2$
- $1 - 1/n$

By definition of a universal family
of hash functions

Proof (Part II)

Let $L = \text{list length in } A[h(x)]$.

For $y \in S$ (so, $y \neq x$) define $z_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{otherwise} \end{cases}$

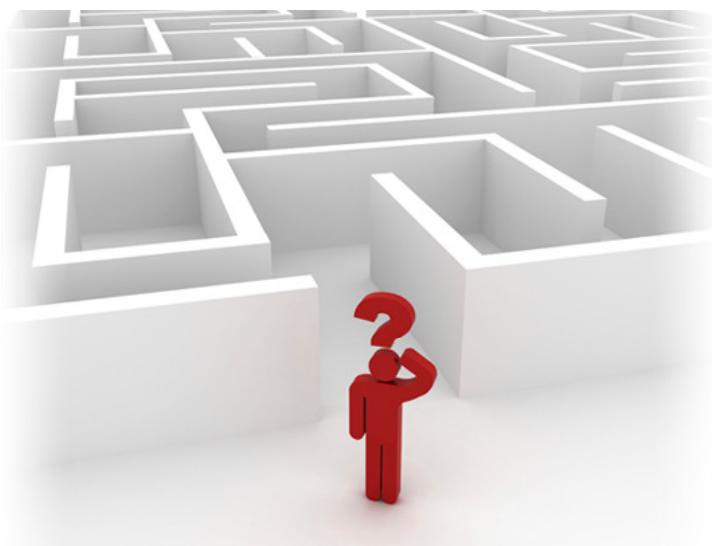
Note : $L = \sum_{y \in S} A_y$

So : $E[L] = \sum_{y \in S} E[Z_y] = \sum_{y \in S} \Pr[h(y) = h(x)] \leq \frac{1}{n}$

Since H is universal $\rightarrow \leq \sum_{y \in S} \frac{1}{n}$

$$= \frac{|S|}{n} = \text{load } \alpha = O(1) \quad \text{Provided } |S| = O(n)$$

Q.E.D.
Tim Roughgarden



Design and Analysis
of Algorithms I

Data Structures

Performance Guarantees
(Open Addressing)

Open Addressing

Recall : one object per slot, hash function produces a probe sequence for each possible key x .

Fact : difficult to analyze rigorously.

Heuristic assumption : (for a quick & dirty idealized analysis only) all $n!$ probe sequences equally.

Heuristic Analysis

Observation : under heuristic assumption, expected
Insertion time is $\sim \frac{1}{1-\alpha}$, where α = load

Proof : A random probe finds an empty slot with
probability $1 - \alpha$

So : Insertion time \sim the number N of coin flips to get
“heads”, where $\text{Pr}[\text{“heads”}] = 1 - \alpha$

Let N denote the number of coin flips need to get “heads”, with a coin whose probability of “heads” is $1 - \alpha$. What is $E[N]$?

- $1/(1 - \alpha)$
- $1/\alpha$
- $1 - \alpha$
- α

Heuristic Analysis

Observation : under heuristic assumption, expected Insertion time is $\sim \frac{1}{1-\alpha}$, where α = load

Proof : A random probe finds an empty slot with probability $1 - \alpha$

So : Insertion time \sim the number N of coin flips to get “heads”, where $\Pr[\text{“heads”}] = 1 - \alpha$

Note : $E[N] = 1 + \alpha \cdot E[N]$

Solution : $E[N] = \frac{1}{1 - \alpha}$

Linear Probing

Note : heuristic assumption completely false.

Assume instead : initial probes uniform at random independent for different keys. (“less false”)

Theorem : [Knuth 1962] under above assumption, expected Insertion time is

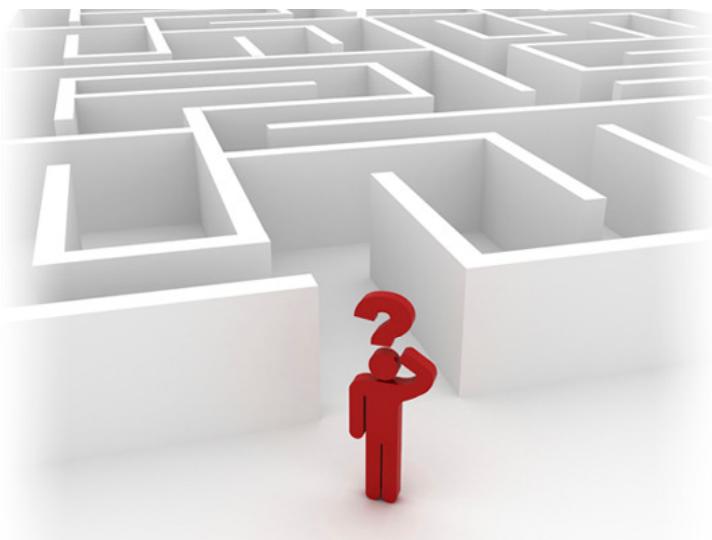
$$= \frac{1}{(1 - \alpha)^2}, \text{ where } \alpha = \text{load}$$

The Allure of Algorithms

“I first formulated the following derivation in 1962... Ever since that day, the analysis of algorithms has in fact been one of the major themes in my life.”

-D. E. Knuth, *The Art of Computer Programming, Volume 3.* (3rd ed., P. 536)

Tim Roughgarden



Design and Analysis
of Algorithms I

Data Structures

Bloom Filters

Bloom Filters: Supported Operations

Raison D'être: fast Inserts and Lookups.

Comparison to Hash Tables:

Pros: more space efficient.

Cons:

- 1) can't store an associated object
- 2) No deletions
- 3) Small false positive probability

(i.e., might say x has been inserted even though it hasn't been)

Bloom Filters: Applications

Original: early spellcheckers.

Canonical: list of forbidden passwords

Modern: network routers.

- Limited memory, need to be super-fast

Bloom Filter: Under the Hood

Ingredients: 1) array of n bits ($So \frac{n}{|S|} = \# \text{ of bits per object in data set } S$)

2) k hash functions h_1, \dots, h_k ($k = \text{small constant}$)

Insert(x): for $i = 1, 2, \dots, k$ (whether or not bit already set to 1)
set $A[h_i(x)] = 1$

Lookup(x): return TRUE $\Leftrightarrow A[h_i(x)] = 1$ for every $i = 1, 2, \dots, k$.

Note: no false negatives. (if x was inserted, $\text{Lookup}(x)$ guaranteed to succeed)

But: false positive if all $k - h_i(x)$'s already set to 1 by other insertions.

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i 's and x 's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A , the probability it's been set to 1 is (under above assumption):

Under the heuristic assumption, what is the probability that a given bit of the bloom filter (the first bit, say) has been set to 1 after the data set S has been inserted?

- $(1 - 1/n)^{k|S|}$ prob 1st bit = 0
- $1 - (1 - 1/n)^{k|S|}$ prob 1st bit = 1
- $(1/n)^{|S|}$
- $(1 - 1/n)^{|S|}$

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

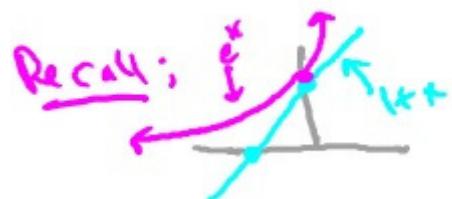
Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i 's and x 's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A , the probability it's been set to 1 is (under above assumption):

$$1 - (1 - \frac{1}{n})^{k|S|} \leq 1 - e^{-\frac{k|S|}{n}} = 1 - e^{-\frac{k}{b}}$$

$b = \# \text{ of}$
 bits per
 object
 $(n/|S|)$



Heuristic Analysis (con'd)

Story so far: probability a given bit is 1 is $\leq 1 - e^{\frac{-k}{b}}$

So: under assumption, for x not in S , false positive probability is $\leq [1 - e^{\frac{-k}{b}}]^k$
where $b = \#$ of bits per object.

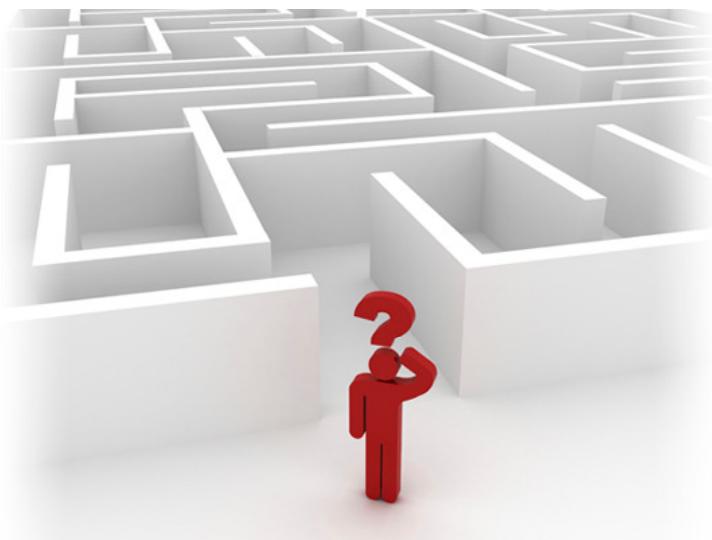
How to set k ?: for fixed b , ϵ is minimized by setting

$$\text{Plugging back in: } \epsilon \approx \left(\frac{1}{2}\right)^{(ln 2)b} \quad \text{or} \quad b \approx 1.44 \log_2 \frac{1}{\epsilon}$$

(exponentially
small in b)

$$k \approx (ln 2) \cdot b \approx 0.693$$

Ex: with $b = 8$, choose $k = 5$ or 6 , error probability only approximately 2%.



Design and Analysis
of Algorithms I

Data Structures

Bloom Filters

Bloom Filters: Supported Operations

Raison D'être: fast Inserts and Lookups.

Comparison to Hash Tables:

Pros: more space efficient.

Cons:

- 1) can't store an associated object
- 2) No deletions
- 3) Small false positive probability

(i.e., might say x has been inserted even though it hasn't been)

Bloom Filters: Applications

Original: early spellcheckers.

Canonical: list of forbidden passwords

Modern: network routers.

- Limited memory, need to be super-fast

Bloom Filter: Under the Hood

Ingredients: 1) array of n bits ($So \frac{n}{|S|} = \# \text{ of bits per object in data set } S$)

2) k hash functions h_1, \dots, h_k ($k = \text{small constant}$)

Insert(x): for $i = 1, 2, \dots, k$ (whether or not bit already set ot 1)
set $A[h_i(x)] = 1$

Lookup(x): return TRUE $\Leftrightarrow A[h_i(x)] = 1$ for every $i = 1, 2, \dots, k$.

Note: no false negatives. (if x was inserted, $\text{Lookup}(x)$ guaranteed to succeed)

But: false positive if all $k - h_i(x)$'s already set to 1 by other insertions.

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i 's and x 's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A , the probability it's been set to 1 is (under above assumption):

Under the heuristic assumption, what is the probability that a given bit of the bloom filter (the first bit, say) has been set to 1 after the data set S has been inserted?

- $(1 - 1/n)^{k|S|}$ prob 1st bit = 0
- $1 - (1 - 1/n)^{k|S|}$ prob 1st bit = 1
- $(1/n)^{|S|}$
- $(1 - 1/n)^{|S|}$

Heuristic Analysis

Intuition: should be a trade-off between space and error (false positive) probability.

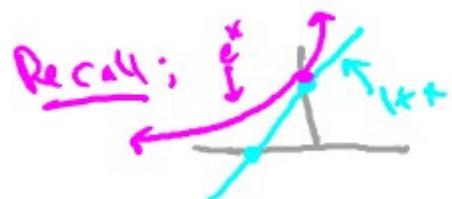
Assume: [not justified] all $h_i(x)$'s uniformly random and independent (across different i 's and x 's).

Setup: n bits, insert data set S into bloom filter.

Note: for each bit of A , the probability it's been set to 1 is (under above assumption):

$$1 - (1 - \frac{1}{n})^{k|S|} \leq 1 - e^{-\frac{k|S|}{n}} = 1 - e^{-\frac{k}{b}}$$

$b = \# \text{ of}$
 bits per
 object
 $(n/|S|)$



Heuristic Analysis (con'd)

Story so far: probability a given bit is 1 is $\leq 1 - e^{\frac{-k}{b}}$

So: under assumption, for x not in S , false positive probability is $\leq [1 - e^{\frac{-k}{b}}]^k$
where $b = \#$ of bits per object.

How to set k ?: for fixed b , ϵ is minimized by setting

$$\text{Plugging back in: } \epsilon \approx \left(\frac{1}{2}\right)^{(ln 2)b} \quad \text{or} \quad b \approx 1.44 \log_2 \frac{1}{\epsilon}$$

(exponentially
small in b)

$$k \approx (ln 2) \cdot b \approx 0.693$$

Ex: with $b = 8$, choose $k = 5$ or 6 , error probability only approximately 2%.