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

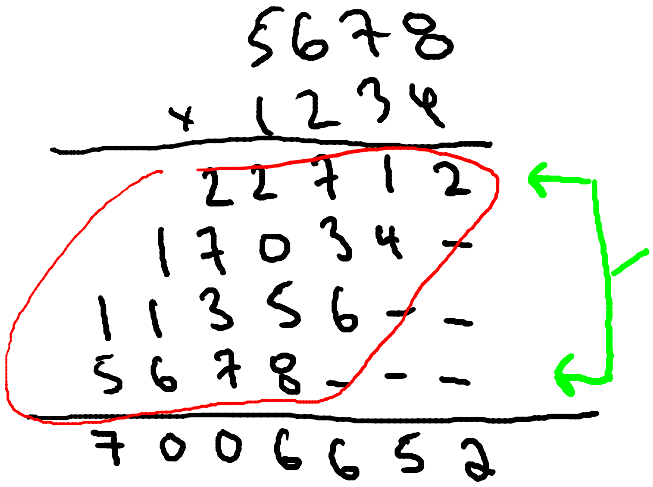
Integer Multiplication

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



Roughly n operations per row up to a constant

# of operations overall ~ constant* n^2

The Algorithm Designer's Mantra

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

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

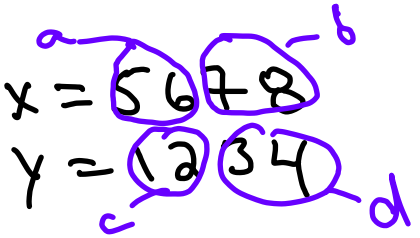
CAN WE DO BETTER ?
[than the "obvious" method]

Design and Analysis of Algorithms I

Introduction

Karatsuba Multiplication

Example



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 ③ − ② − ① $= 2840$

Step 5:

$$\begin{array}{r} 6720000 \\ 2652 \\ 284000 \\ \hline 7006652 \end{array} = (1234)(5678)$$

A Recursive Algorithm

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

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

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

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

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

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

Simple Base Case Omitted

Karatsuba Multiplication

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

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 ?

Introduction

About The Course

Design and Analysis
of Algorithms I

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

Tim Roughgarden

Topics in Sequel Course

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

Topics in Sequel Course

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

Skills You'll Learn

- Become a better programmer

Skills You'll Learn

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

Skills You'll Learn

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

Skills You'll Learn

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

Skills You'll Learn

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

Who Are You?

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

Who Are You?

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

Who Are You?

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

Who Are You?

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

Who Are You?

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

- *Excellent free reference: "*Mathematics for Computer Science", by Eric Lehman and Tom Leighton. (Easy to find on the Web.)

Supporting Materials

- All (annotated) slides available from course site.

Supporting Materials

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

Biggest influence on instructor

Freely available online

Supporting Materials

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

Assessment

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

Assessment

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

Design and Analysis
of Algorithms I

Introduction

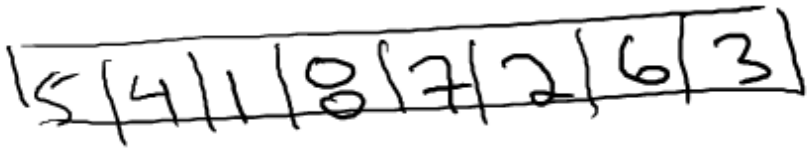
Merge Sort (Overview)

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

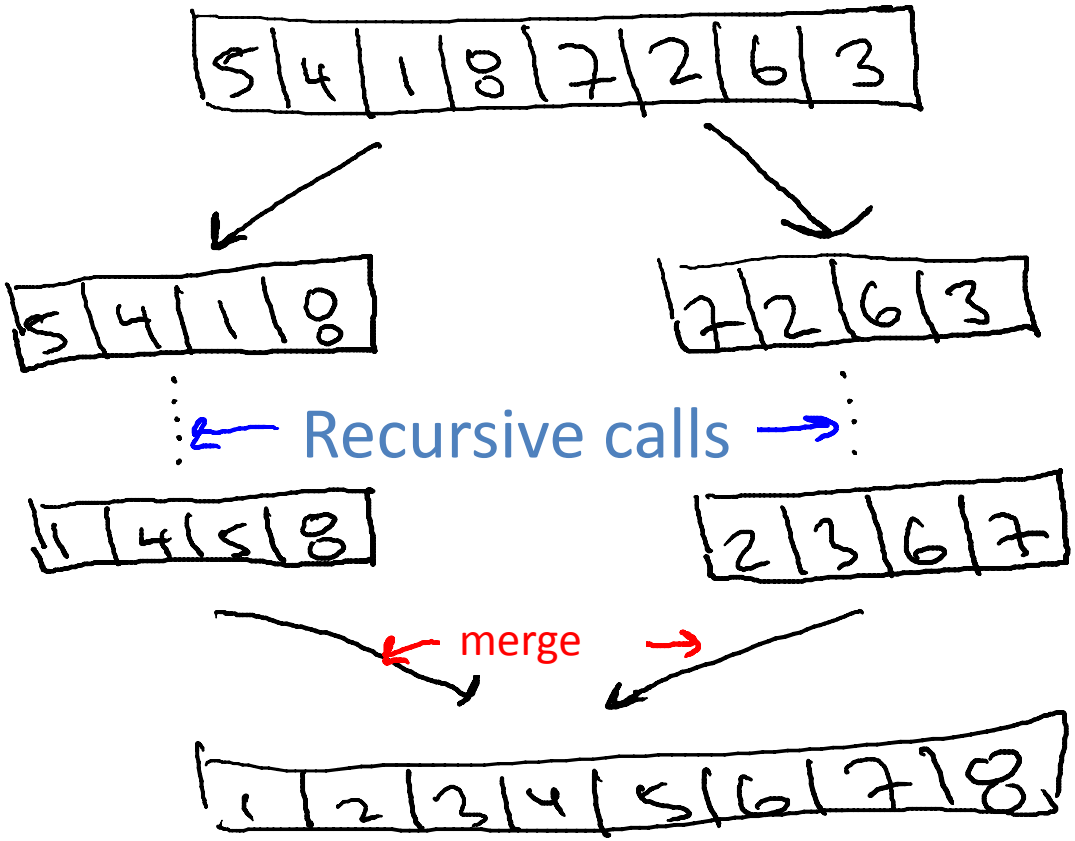
Input : array of n numbers, unsorted.



Assume Distinct

Output : Same numbers, sorted in increasing order

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 ~ # 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 $m \geq 1$)

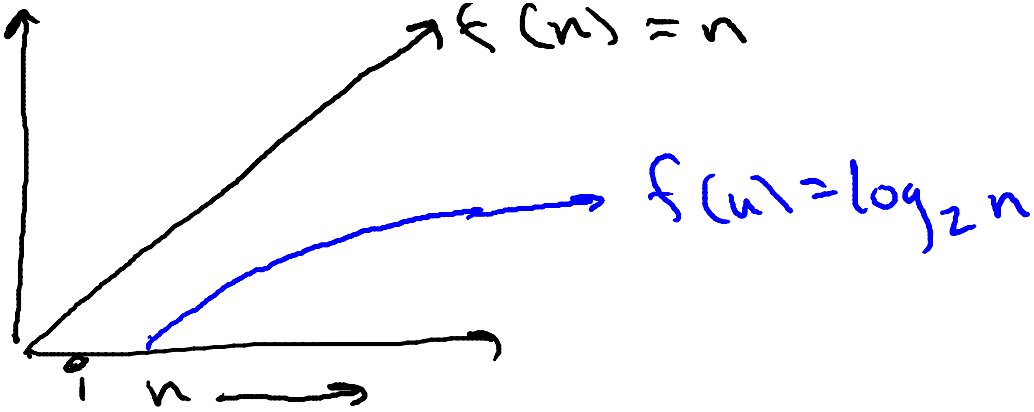
Running Time of Merge Sort

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

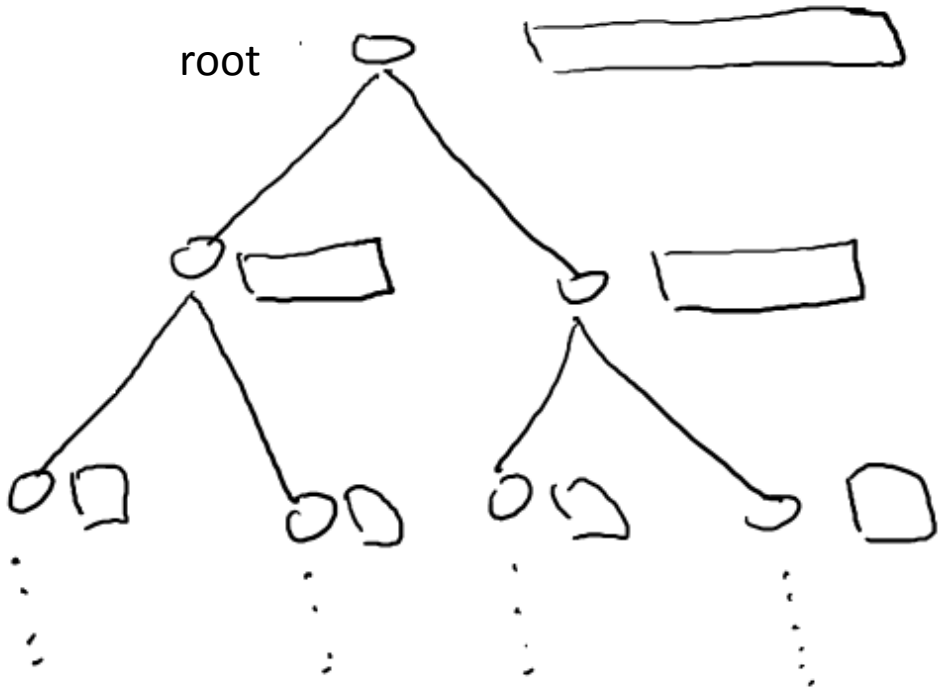
Claim: For every input array of n numbers, Merge Sort produces a sorted output array and uses at most $6n \log_2 n + 6n$ operations.

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

Level 0
[outer call to Merge Sort]

Level 1
(1st recursive calls)

Level 2

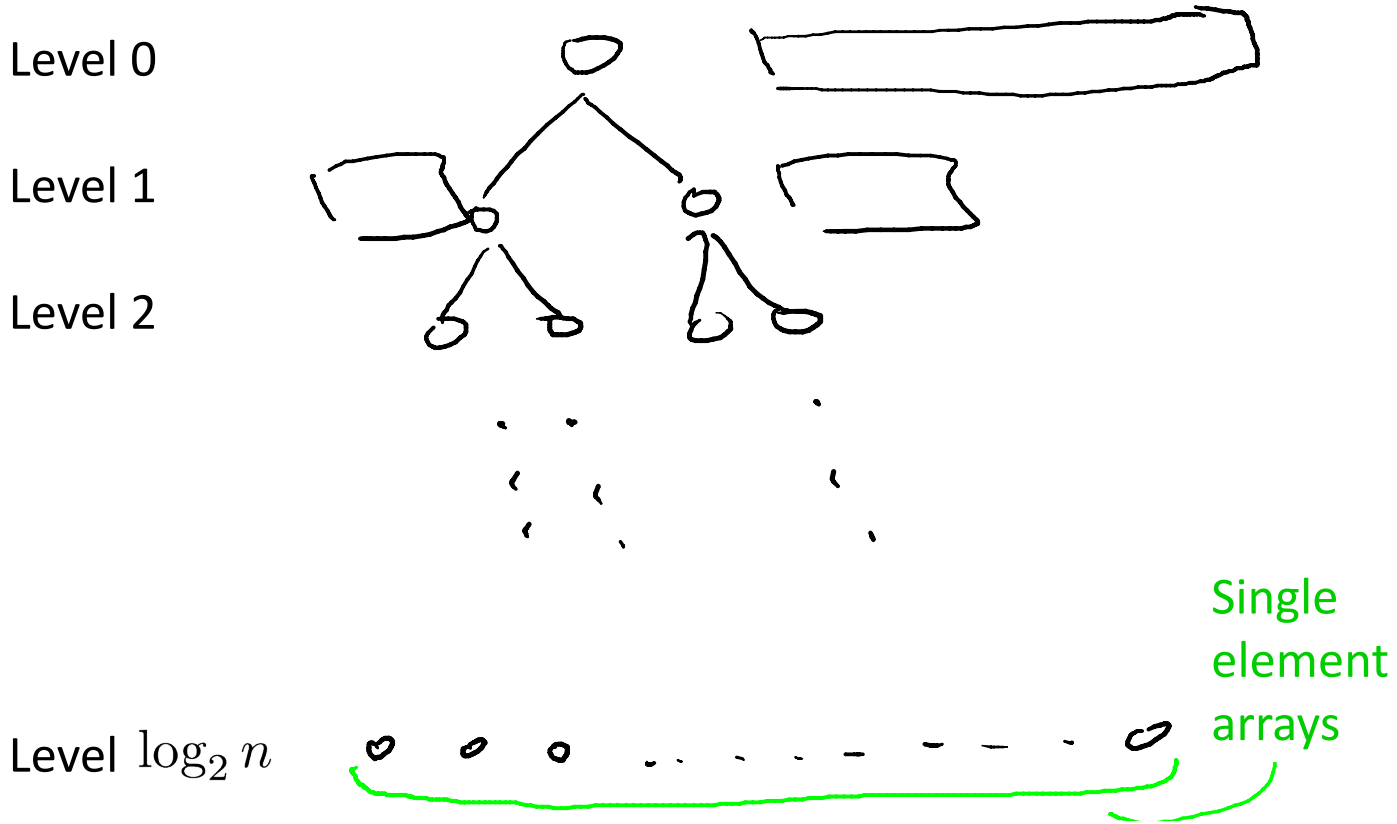


Ter
ver

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

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

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



Ter
ver

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

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

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

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

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

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

# 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

Tim Roughgarden

Running Time of Merge Sort

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

QED!

Design and Analysis
of Algorithms I

Introduction

Guiding Principles

Guiding Principle #1

"worst – case analysis" : our running time bound holds for every input of length n.
-Particularly appropriate for "general-purpose" routines

As Opposed to
--"average-case" analysis
--benchmarks

REQUIRES DOMAIN KNOWLEDGE

BONUS : worst case usually easier to analyze.

Guiding Principle #2

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

<u>Justifications</u>

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

Guiding Principle #3

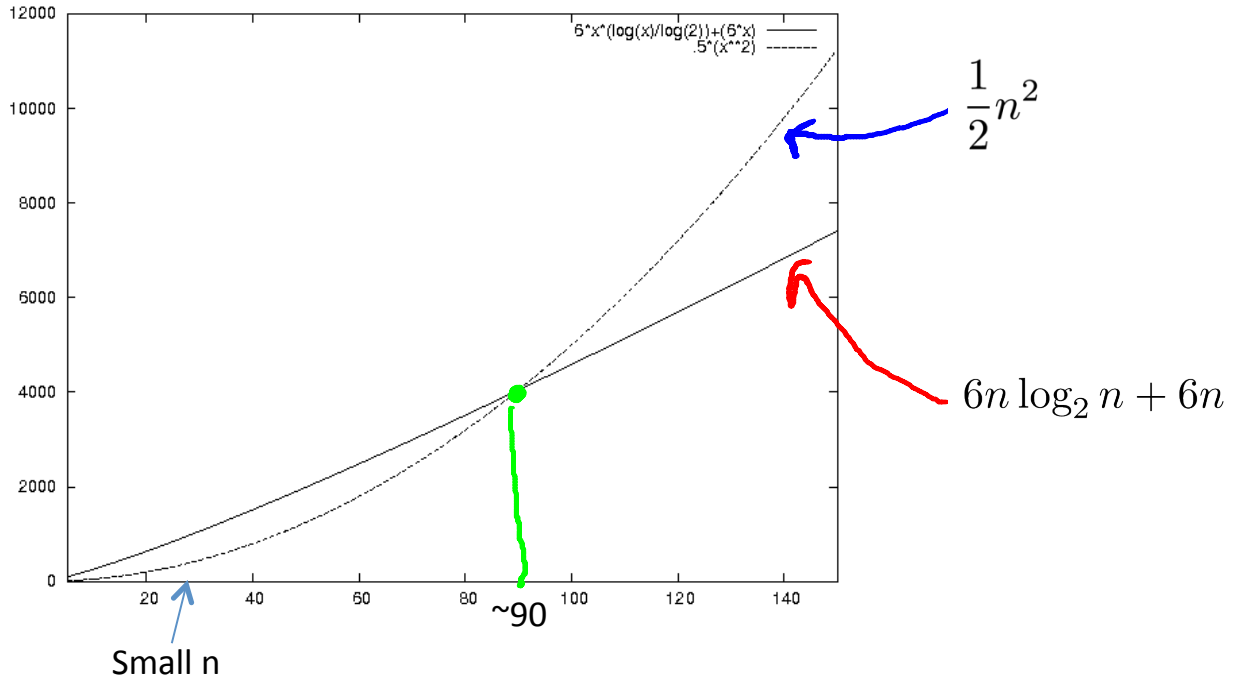
Asymptotic Analysis : focus on running time for large input sizes n

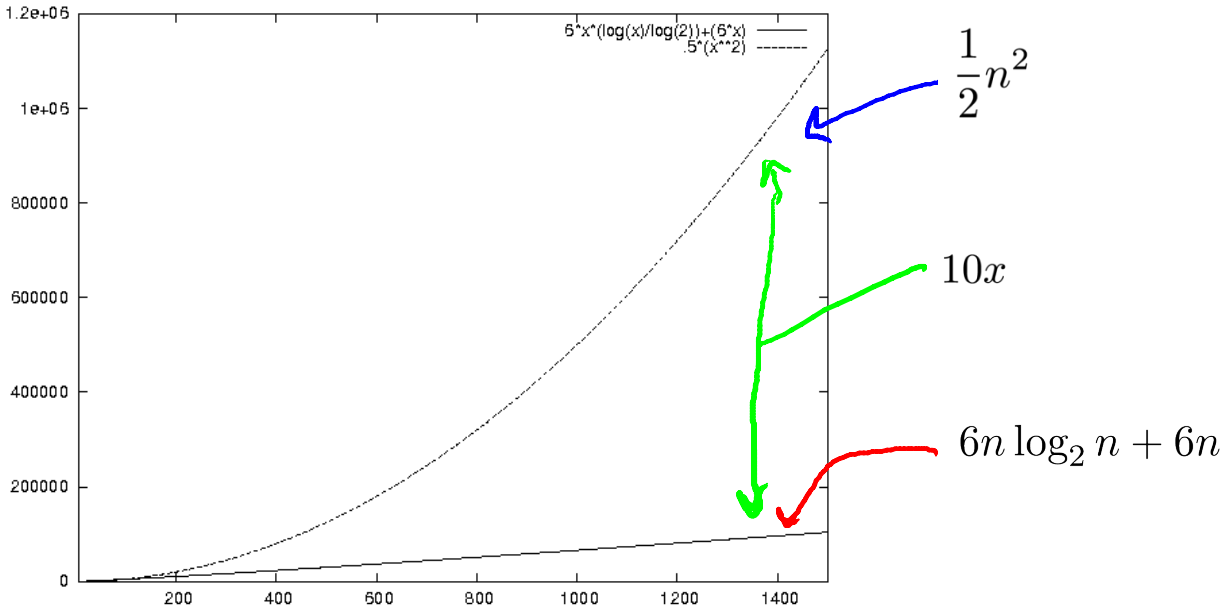
Eg : $6n \log_2 + 6n$ "better than" $\frac{1}{2}n^2$

MERGE SORT

INSERTION SORT

Justification: Only big problems are interesting!


6*x*(log(x)/log(2))+(6*x)
.5*(x**2)
12000
10000
8000
6000
4000
2000
0
20
40
60
80
100
120
140
~90
Small n
$\frac{1}{2}n^2$
$6n \log_2 n + 6n$



Tim Roughgarden

What Is a "Fast" Algorithm?

This Course : adopt these three biases as guiding principles

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

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

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) $O(1)$

B) $O(\log n)$

C) $O(n)$

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) $O(n^2)$

Asymptotic Analysis

Big-Oh: Definition

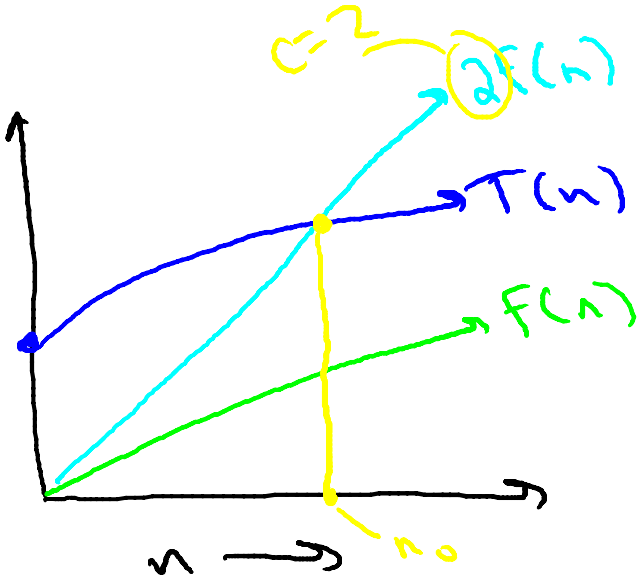
Design and Analysis of Algorithms I

Big-Oh: English Definition

Let T(n) = function on n = 1,2,3,…
[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



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

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

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

For all $n \geq n_0$

Warning : c, n_0 cannot depend on n

Asymptotic Analysis

Big-Oh: Basic Examples

Design and Analysis of Algorithms I

Example #1

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

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

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

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

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

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

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

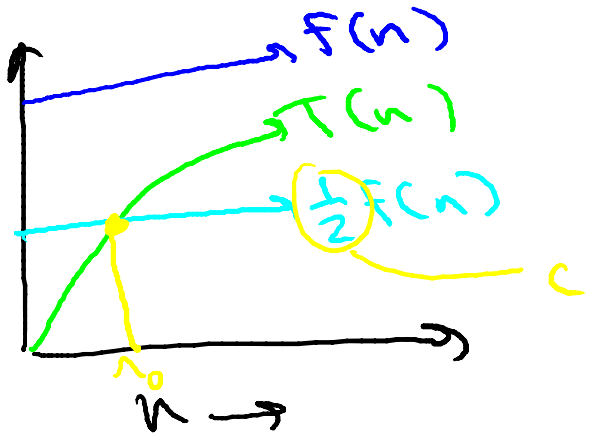
Big-Oh: Relatives (Omega & Theta)

Omega Notation

<u>Definition</u> : $T(n) = \Omega(f(n))$
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



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

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.)

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

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 compter science and mathematics journals, adopt the O, Ω, and Θ notations as defined above, unless a better alternative can be found reasonably soon".

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

Asymptotic Analysis

Additional Examples

Design and Analysis of Algorithms I

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]

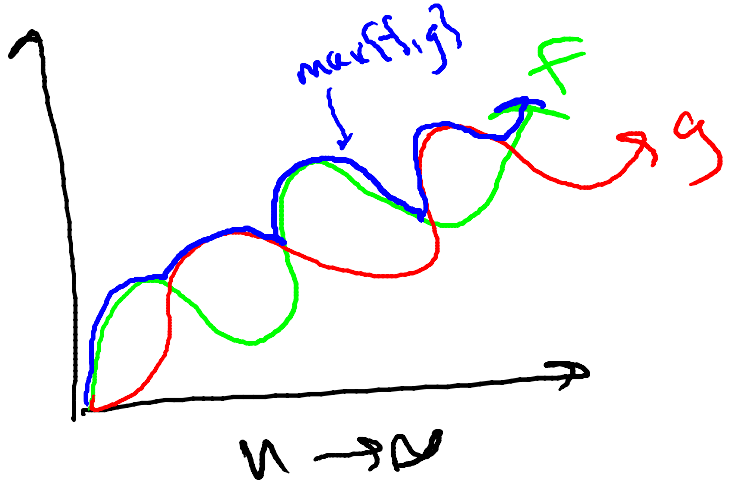
$$2^{9n} \leq c \quad \forall n \geq n_0$$

Which is certainly false. Q.E.D

Example #3

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

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



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 $\frac{1}{2} * (f(n) + g(n)) \leq max\{f(n), g(n)\} \leq f(n) + g(n) \quad \forall n \geq 1$

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

Q.E.D

Tim Roughgarden

Design and Analysis of Algorithms I

Divide and Conquer

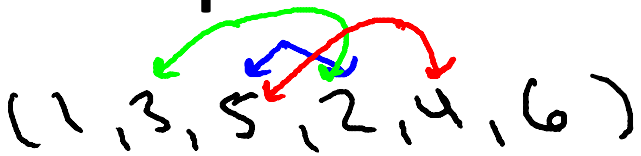
Counting Inversions I

The Problem

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

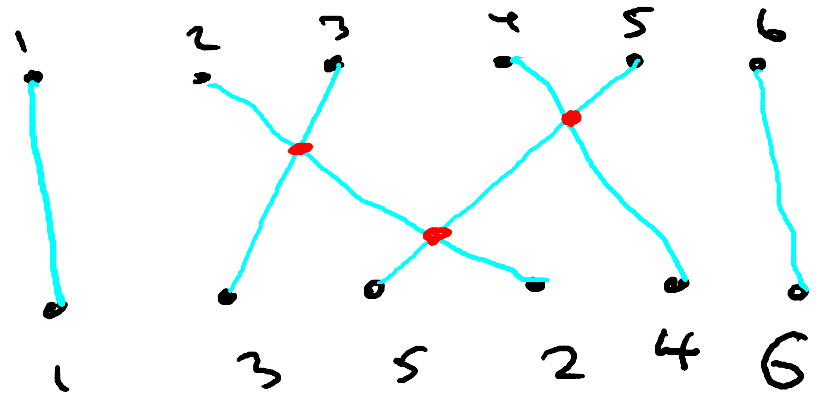
Output : number of inversions = number of pairs (i,j) of array indices with i<j and A[i] > A[j]

Examples and Motivation

Example 

Inversions :

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



Motivation : numerical similarity measure between two ranked lists eg: for collaborative filtering

Ter
ver

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

- 15
- 21
- 36
- 64

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

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<=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(nlog(n)) time [just like Merge Sort]

Tim Roughgarden

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(nlog(n)) time [just like Merge Sort]

Pseudocode for Merge:

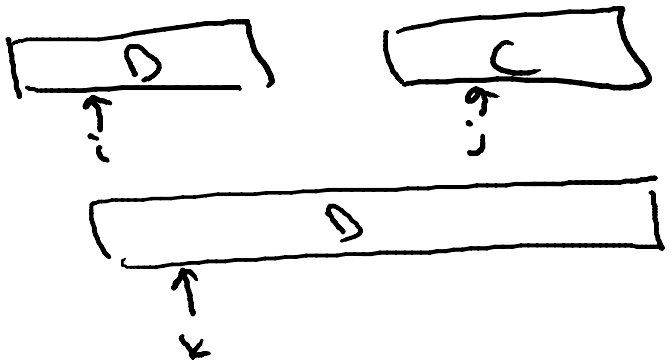
D = output [length = n]

B = 1st sorted array [n/2]

C = 2nd sorted array [n/2]

i = 1

j = 1



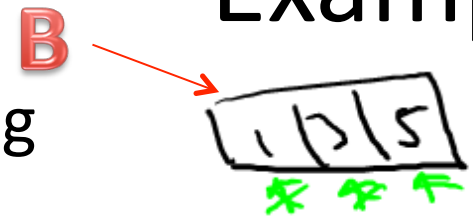

```
for k = 1 to n
        if B(i) < C(j)
                D(k) = B(i)
                i++
        else [C(j) < B(i)]
                D(k) = C(j)
                j++
end
```

(ignores end cases)

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

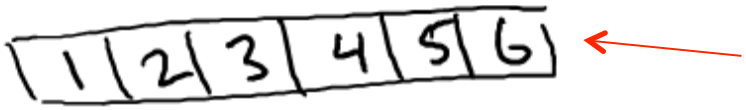
- B has the smallest element of A, C the second-smallest, B, the third-smallest, and so on.
- All elements of B are less than all elements of C.
- All elements of B are greater than all elements of C.
- There is not enough information to answer this question.

Example

Consider merging 

and 

.

Output : 

⇒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 2nnd 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

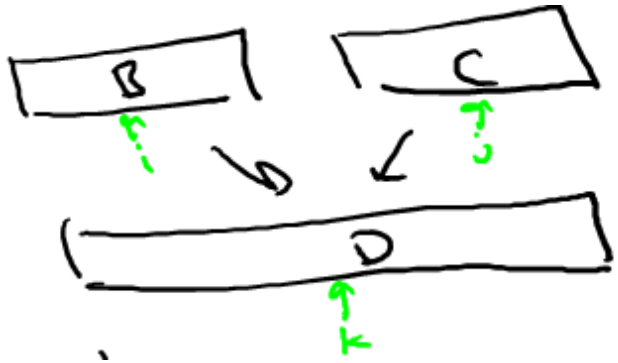
⇒ no inversions involving x and y

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

merge running total

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

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

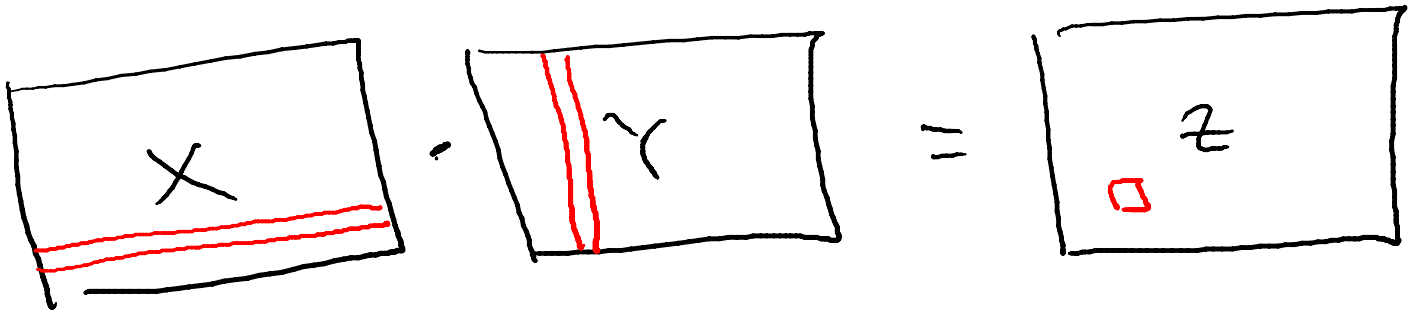
Tim Roughgarden

Design and Analysis of Algorithms I

Divide and Conquer

Matrix Multiplication

Matrix Multiplication



(all n X n matrices)

Where z_{ij} = (ith row of X) . (jth column of Y)

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

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

Tim Roughgarden

Example (n=2)

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

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

$\theta(n)$

Ter
ver

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)

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

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

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

Proof: $AE + \cancel{AH} + \cancel{DE} + \cancel{DH} + \cancel{DG} - \cancel{DE} - \cancel{AH} - \cancel{BH} + \cancel{BG} + \cancel{BH} - \cancel{DG} - \cancel{DH} = AE + BG$ Q.E.D

Question : where did this come from ? (remains open!)

Tim Roughgarden

Divide and Conquer

Closest Pair I

Design and Analysis of Algorithms I

The Closest Pair Problem

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

Tim Roughgarden

Initial Observations

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

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

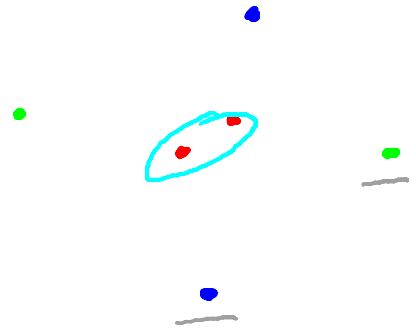
1-D Version of Closest Pair :

1. Sort points (O(nlog(n)) time)
2. Return closest pair of adjacent points (O(n) time)

Goal : O(nlog(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(nlog(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 = left half of P, R = right half of P. Form

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

2. (p_1,q_1) = ClosestPair(Q_x,Q_y)
3. (p_2,q_2) = ClosestPair(R_x,R_y)
4. (p_3,q_3) = 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)

1. Let Q = left half of P, R = right half of P. Form

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

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

BASE CASE OMITTED

WILL DESCRIBE NEXT

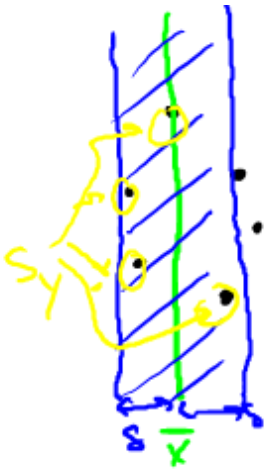
Requirements

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

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
    Let p,q = i^{th} , $(i+j)^{th}$ points of S_y
    If d(p,q) < best
      best pair = (p,q), best = d(p,q)

O(1) time

O(n) time

At end return
best pair

Tim Roughgarden

Correctness Claim

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

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

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

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

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

Corollary2 ClosestPair is correct, and runs in O(nlog(n)) time.

Assuming claim is true!

Divide and Conquer

Closest Pair II

Design and Analysis of Algorithms I

Correctness Claim

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

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

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

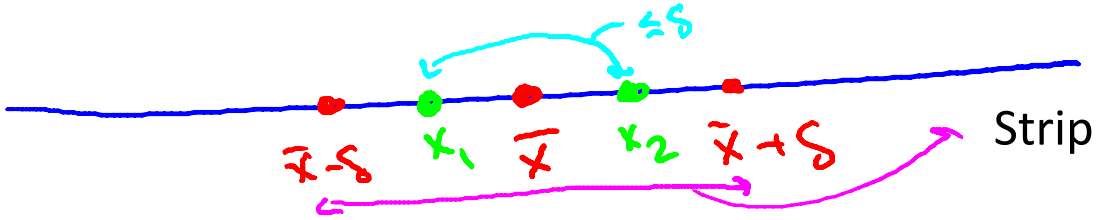
(B) p and q are at most 7 positions apart in S_y.

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



Strip that defines S_y

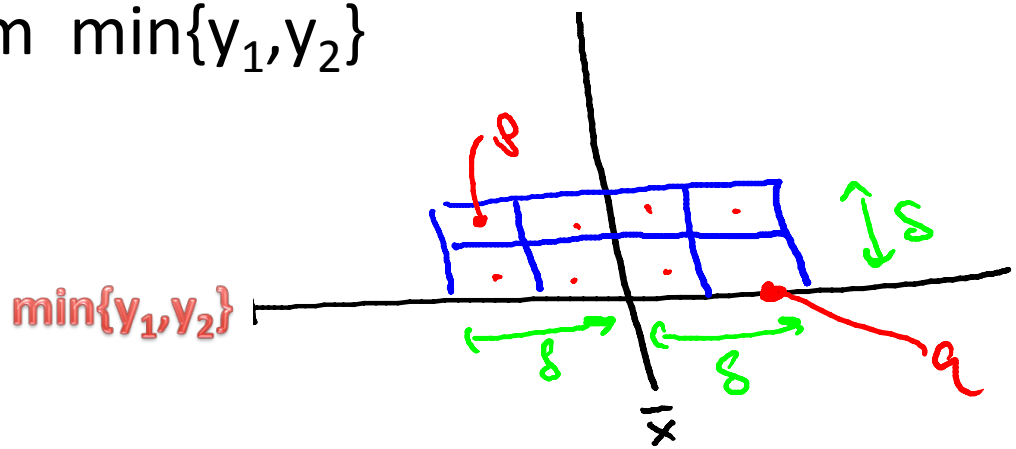
Note : $p \in Q => x_1 \leq \bar{x}\ and\ q \in R => x_2 \geq \bar{x}.$

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

Proof of Correctness Claim (B)

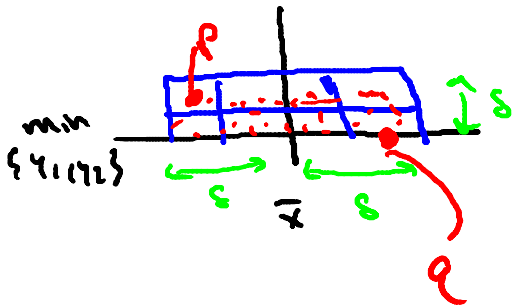
(B) : p = (x_1,y_1) and q = (x_2,y_2) are at most 7 positions apart in S_y

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



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.

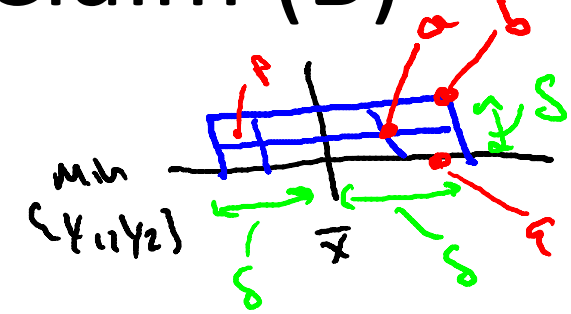


Proof : First, recall y-coordinates of p,q differ by $< \delta$

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)

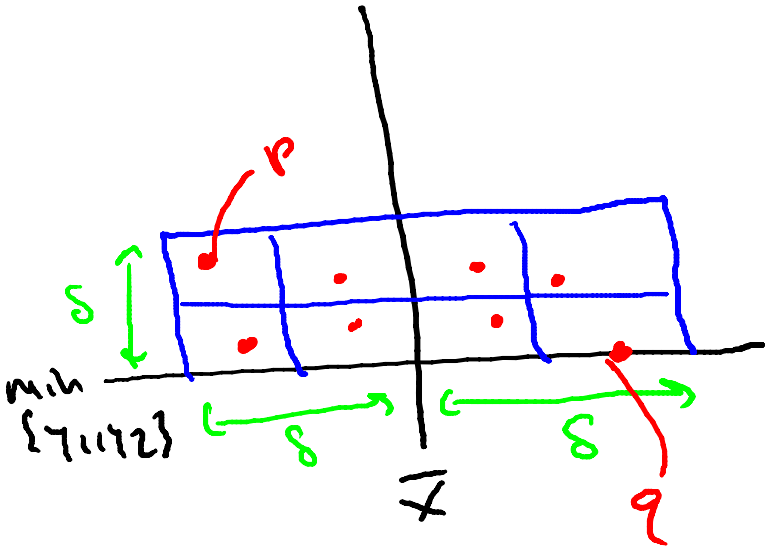
Q.E.D

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