Design and Analysis of Algorithms I

Divide and Conquer

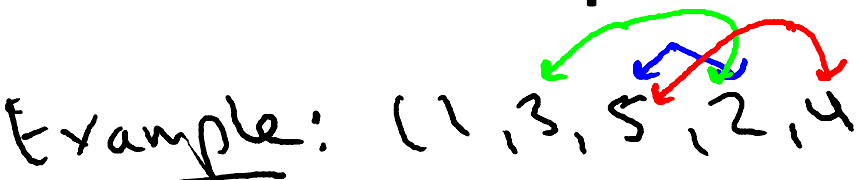
Counting Inversions I

The Problem

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

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

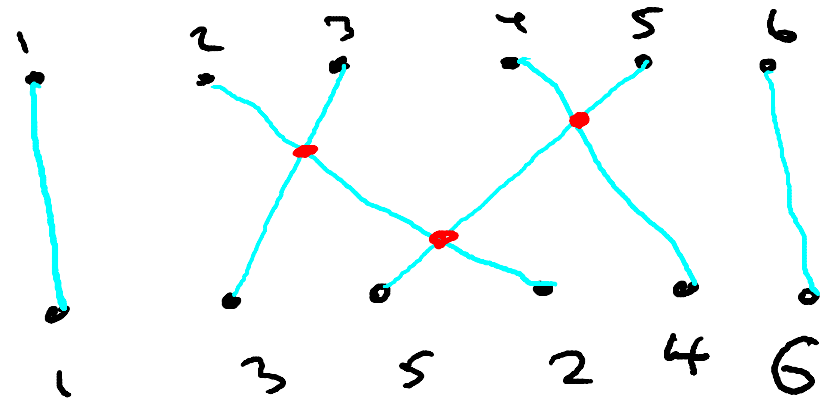
Examples and Motivation



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

Inversions:

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



Motivation: Numerical similarity measure between two ranked lists. (e.g., for "collaborative filtering")

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

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

High-Level Approach

Brute-force: $O(n^2)$ time.

Can we do better? YES!

KEY IDEA #1: Divide + Conquer.

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

- left if $i,j \leq n/2$
- right if $i,j > n/2$
- split if $i \leq \frac{n}{2} < j$

Note: can compute left & right 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 $\Rightarrow$ then Count will run in $O(n \log n)$ time [just like MergeSort].

Design and Analysis of Algorithms I

Divide and Conquer

Counting Inversions II

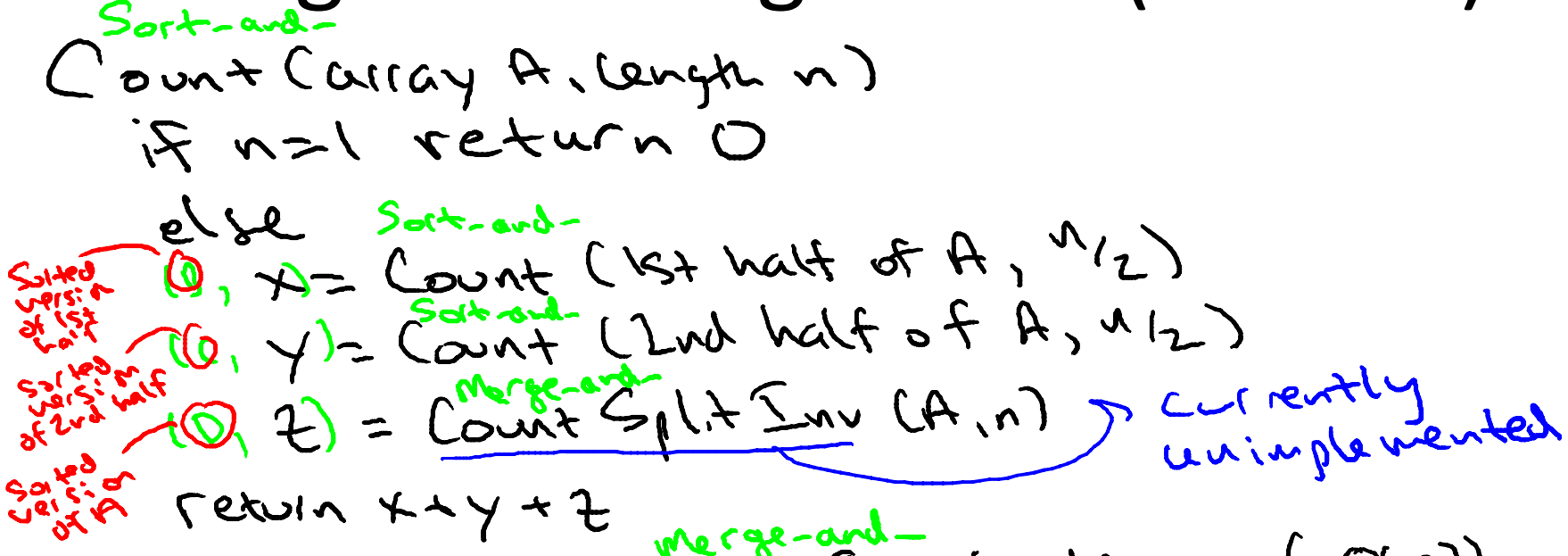
Piggybacking on Merge Sort

KEY IDEA #2: have recursive calls both count inversions and sort.

[i.e., piggyback on Merge Sort]

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

High-Level Algorithm (revised)



Goal: implement Merge-and-CountSplitInv in linear ($O(n)$) time $\Rightarrow$ then Sort-and-Count will run in $O(n \log n)$ time [just like merge sort].

Pseudocode for Merge:

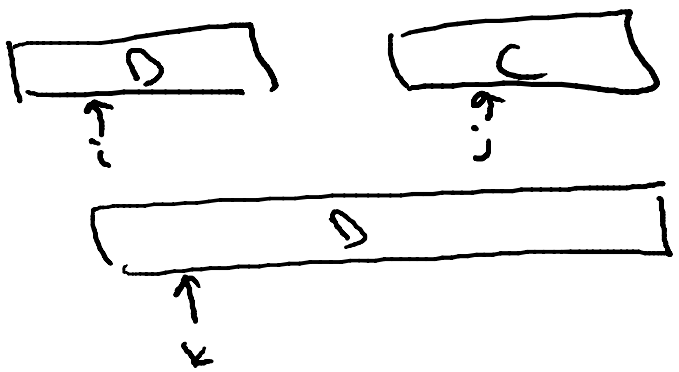
D = output [length = n]

B = 1st sorted array [n/2]

C = 2nd sorted array [n/2]

i = 1

j = 1



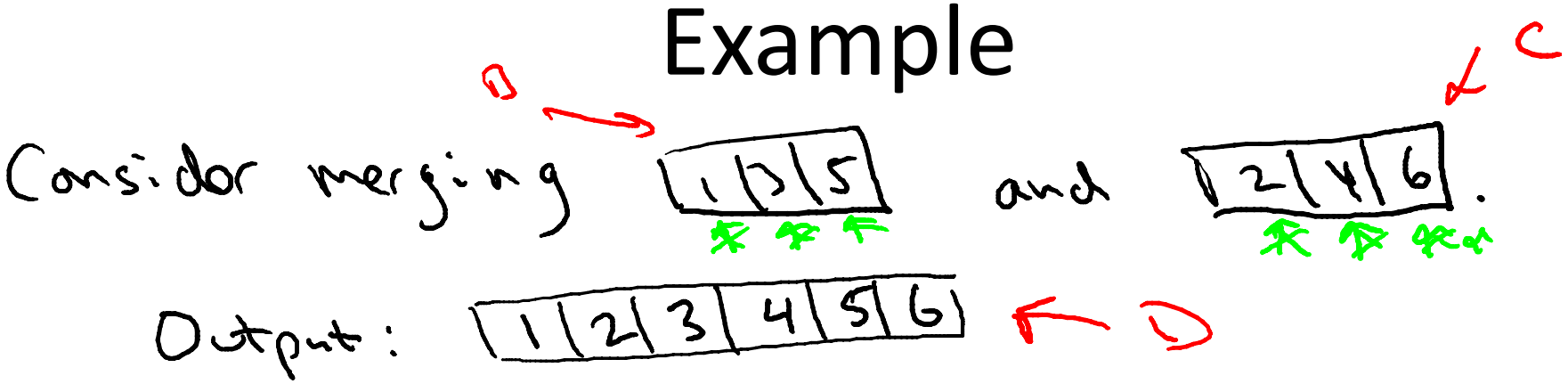

```
for k = 1 to n
        if B(i) < C(j)
                D(k) = B(i)
                i++
        else [C(j) < B(i)]
                D(k) = C(j)
                j++
end
```

(ignores end cases)

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



=> 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.

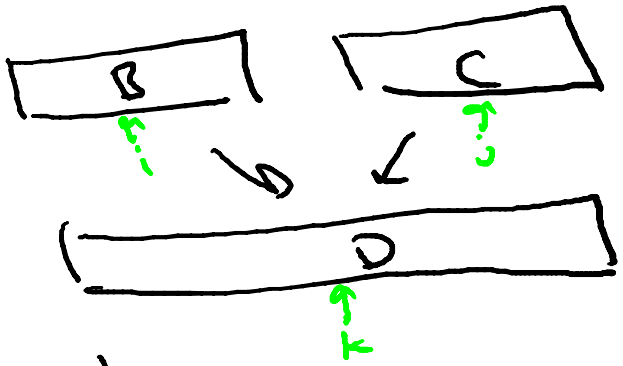
(1) if x copied to output D before y, then $x < y$
$\Rightarrow$ no inversion involving x, y

(2) if y copied to output D before x, then $y < x$
$\Rightarrow$ x, y are a (split) inversion

QED!

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 : $O(n)$ (Merge) $+ O(n)$ (running total) $= O(n)$

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

Design and Analysis
of Algorithms I

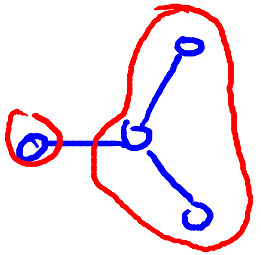
Contraction Algorithm

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?

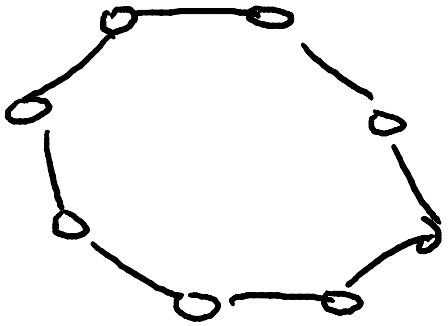
ANSWER: $\binom{n}{2} = \frac{n(n-1)}{2}$

The Lower Bound

Consider the n-cycle.

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

$\Rightarrow$ has $\geq \binom{n}{2}$ min cuts



($n=8$)

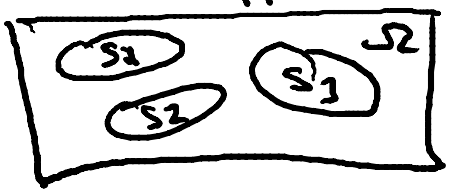
The Upper Bound

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

By the Contraction Algorithm analysis (without repeated trials):

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

Note: S_i's are disjoint events. (i.e., only one can happen)
$\Rightarrow$ their probabilities sum to at most 1



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

QED!

Design and Analysis
of Algorithms I

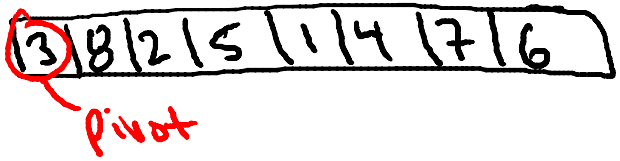
QuickSort

The Partition Subroutine

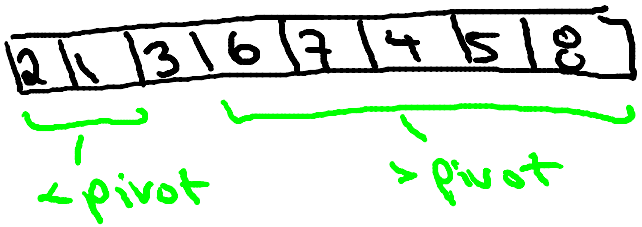
Partitioning Around a Pivot

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".

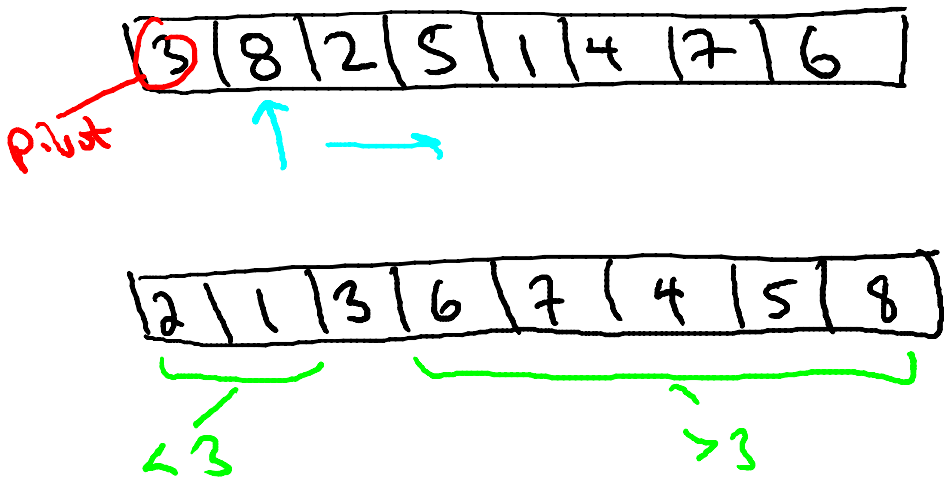
Two Cool Facts About Partition

(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

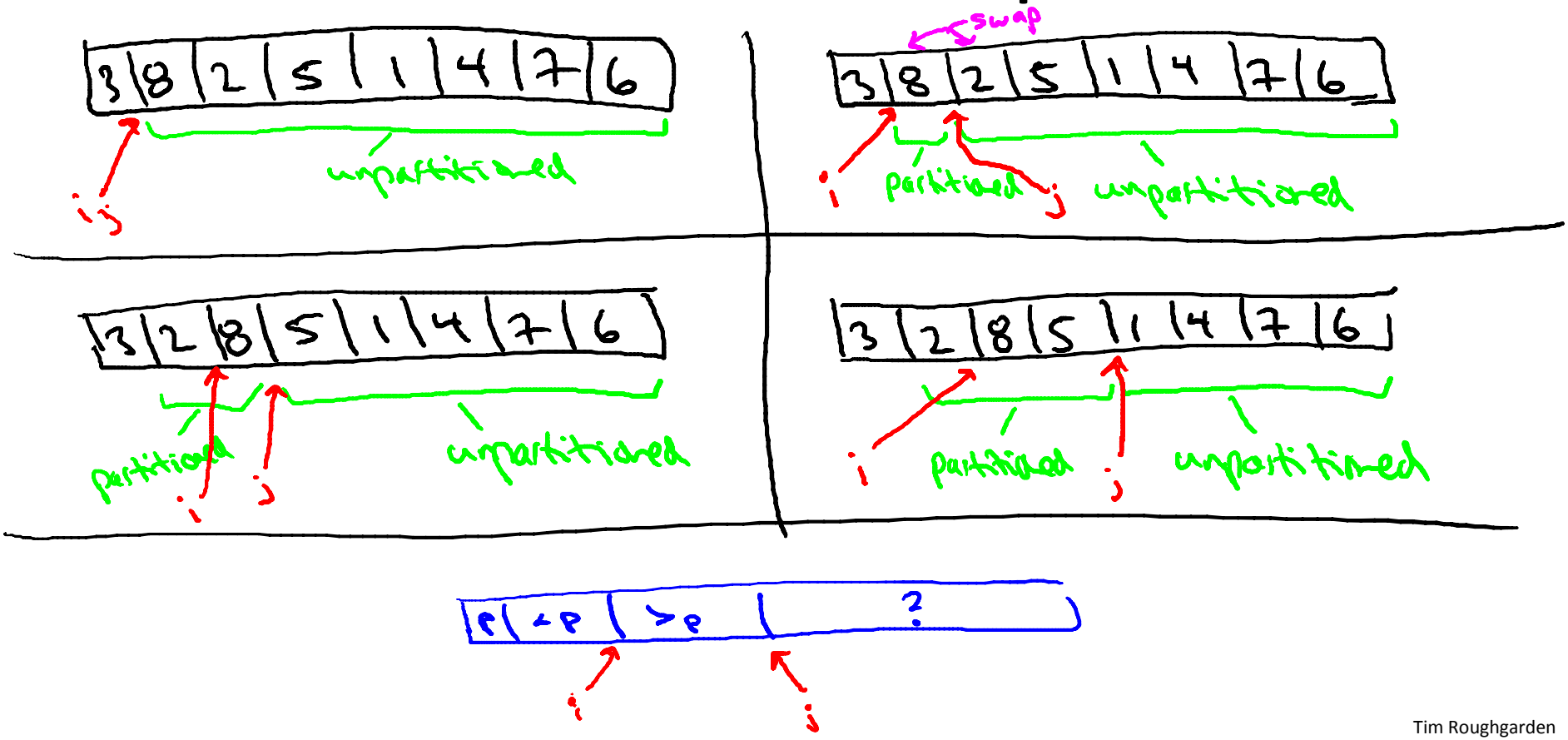
Assume: pivot = 1st element of array.

[if not, swap pivot ↔ 1st element as preprocessing step]

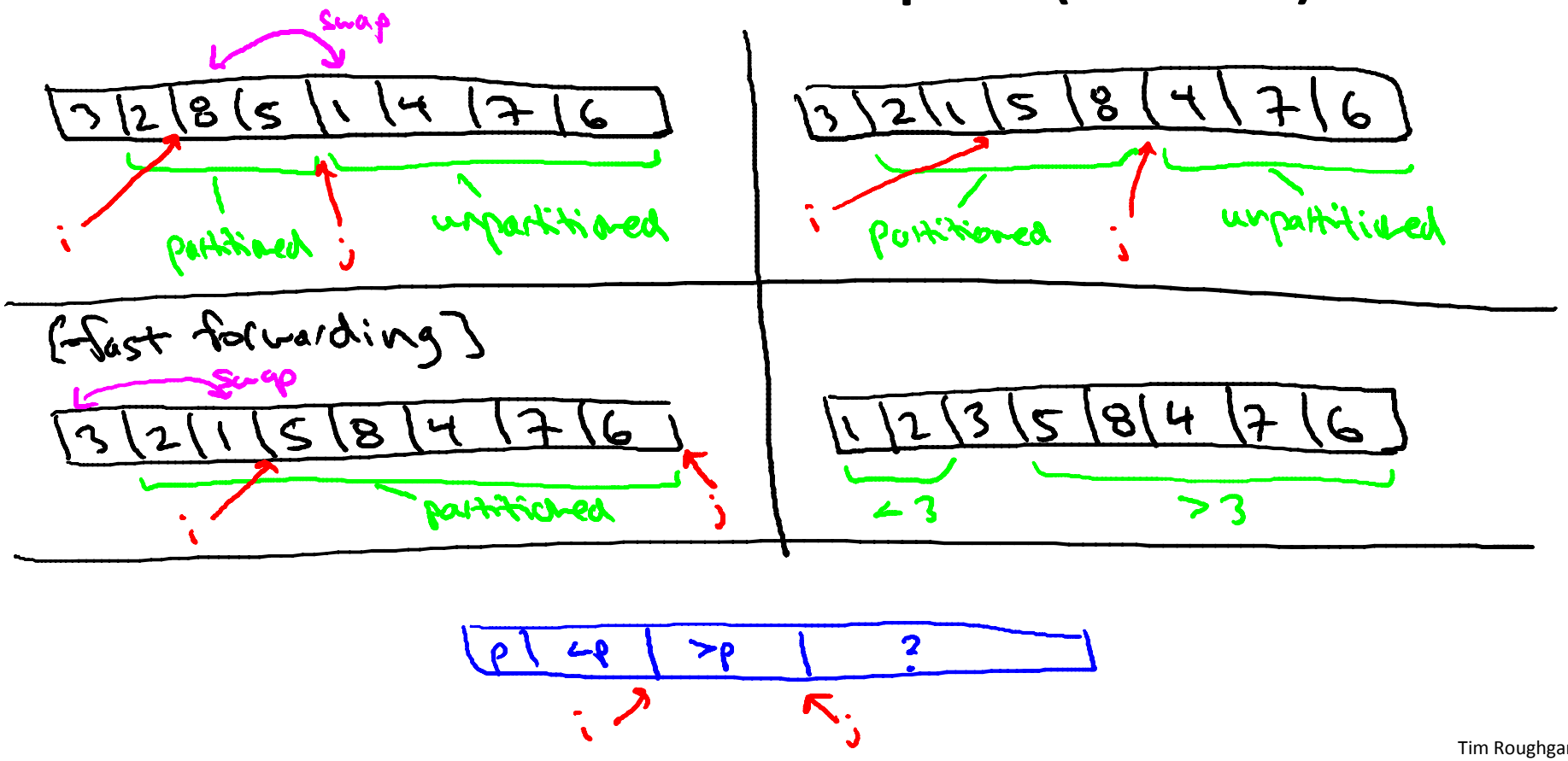
High-level Idea:

- single scan through array
- invariant: everything looked at so far is partitioned

Partition Example



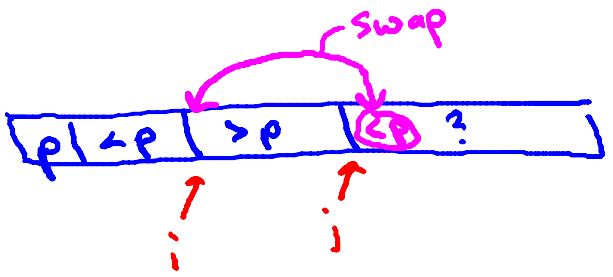
Partition Example (con'd)



Pseudocode for Partition

Partition (A, l, r) [input = A[l ... 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]



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.

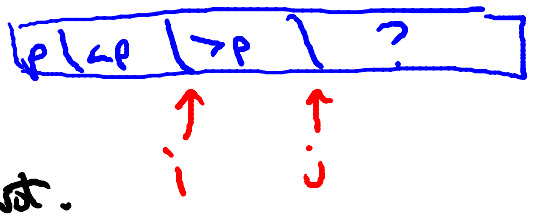
Also: clearly works in place (repeated swaps).

Correctness

Claim: the for loop maintains the invariants:

① A[l+1], ..., A[i-1] are all less than the pivot

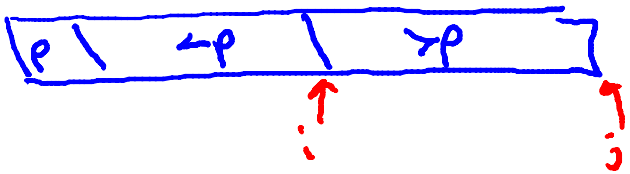
② A[i], ..., A[j-1] are all greater than pivot.



[Exercise: check this, by induction.]

Consequence: at end of for loop, have

=> after final swap, array partitioned around pivot. QED!



Data Structures

Hash Tables and Applications

Design and Analysis of Algorithms I

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

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

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 $x + y = t$

Naive solution: $O(n^2)$ time via exhaustive search.

Better: (1) Sort A ($O(n \log n)$ time) (2) for each x in A, look for $t - x$ in A via binary search

$O(n \log n)$ → $O(n \log n)$ time

Amazing: (1) insert elements of A into hash table H (2) for each x in A, Lookup $t - x$ in H → $O(n)^*$ time

$O(n)^*$ 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.

Data Structures

Hash Tables: Some Implementation Details

Design and Analysis of Algorithms I

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

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

High-Level Idea

Setup: universe U (e.g., all IP addresses, all names, all chess board 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 $n \approx |S|$ (for simplicity assume $|S|$ doesn't vary too much)

② choose a hash function $h: U \to \{0, 1, 2, \ldots, n-1\}$

③ use array A of length n, store x in $A[h(x)]$

Naive 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

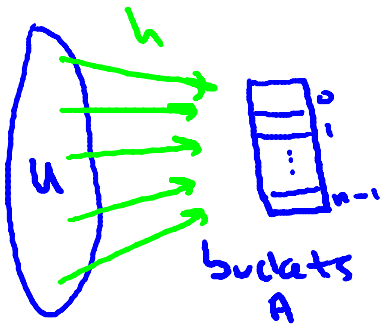
Tim Roughgarden

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"

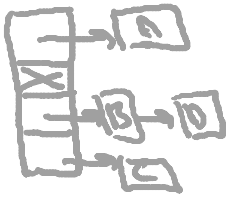
Resolving Collisions



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 $A[h(x)]$ (bucket for x; linked list for x)



Solution #2: open addressing. (only one object per bucket)

- hash function now specifies probe sequence $h_1(x), h_2(x), \ldots$ (keep trying til 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 $O(1)$ [insert new object x at front of list in $A[h(x)]$]

$O(\text{list length})$ for Insert / Delete.

→ could be anywhere from $\frac{m}{n}$ to m for m objects

$\frac{m}{n}$ → equal-length lists

m → all objects in same bucket

Point: performance depends on the choice of hash function!

(analogous situation with open addressing)

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: $h(x)$ = last 3 digits of x (i.e., area code) Choose $n = 10^3$
- mediocre hash function: $h(x)$ = last 3 digits of x

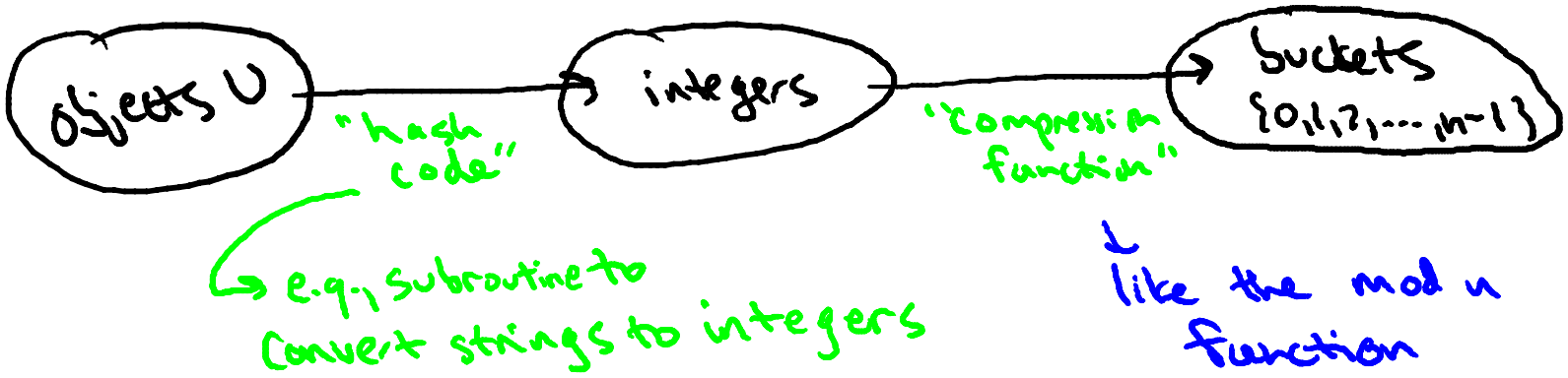
  [still vulnerable to patterns in last 3 digits]

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

Final Exam

Question 1

Recall the Partition subroutine that we used in both QuickSort and RSelect. Suppose that the following array has just been partitioned around some pivot element: 3, 1, 2, 4, 5, 8, 7, 6, 9

Which of these elements could have been the pivot element? (Hint: There could be more than one possibility!)

- [ ] 3
- [ ] 9
- [ ] 2
- [ ] 4
- [ ] 5

Question 2

Here is an array of ten integers: 5 3 8 9 1 7 0 2 6 4

Suppose we run MergeSort on this array. What is the number in the 7th position of the partially sorted array after the outermost two recursive calls have completed (i.e., just before the very last Merge step)? (When we say "7th" position, we're counting positions starting at 1; for example, the input array has a "0" in its 7th position.)

- ( ) 4
- ( ) 3
- ( ) 1
- ( ) 2

Question 3

What is the asymptotic worst-case running time of MergeSort, as a function of the input array length n?

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

Question 4

Consider a directed graph $G=(V,E)$ with non-negative edge lengths and two vertices s and t of V. Let P denote a shortest path from s to t in G. If we add 10 to the length of every edge in the graph, then:

- [ ] P might or might not remain a shortest $s-t$ path (depending on the graph).
- [ ] P definitely does not remain a shortest $s-t$ path.
- [ ] If P has only one edge, then P definitely remains a shortest $s-t$ path.
- [ ] P definitely remains a shortest $s-t$ path.

Question 5

What is the running time of depth-first search, as a function of n and m, if the input graph $G=(V,E)$ is represented by an adjacency matrix (i.e., NOT an adjacency list), where as usual $n=|V|$ and $m=|E|$?

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

Question 6

What is the asymptotic running time of the Insert and Extract-Min operations, respectively, for a heap with n objects?

- $\Theta(1)$ and $\Theta(\log n)$
- $\Theta(\log n)$ and $\Theta(1)$
- $\Theta(n)$ and $\Theta(1)$
- $\Theta(\log n)$ and $\Theta(\log n)$

Question 7

On adding one extra edge to a directed graph G, the number of strongly connected components...?

- ...cannot change
- ...cannot decrease
- ...cannot decrease by more than 1
- ...might or might not remain the same (depending on the graph).

Question 8

What is the asymptotic running time of Randomized QuickSort on arrays of length n, in expectation (over the choice of random pivots) and in the worst case, respectively?

- $\Theta(n\log n)$ [expected] and $\Theta(n\log n)$ [worst case]
- $\Theta(n^2)$ [expected] and $\Theta(n^2)$ [worst case]
- $\Theta(n)$ [expected] and $\Theta(n\log n)$ [worst case]
- $\Theta(n\log n)$ [expected] and $\Theta(n^2)$ [worst case]

Question 9

Let f and g be two increasing functions, defined on the natural numbers, with $f(1), g(1) \geq 1$. Assume that $f(n) = O(g(n))$. Is $2^{f(n)} = O(2^{g(n)})$? (Multiple answers may be correct.)

- Always
- Yes if $f(n) \leq g(n)$ for all sufficiently large n
- Never
- Maybe, maybe not (depends on the functions f and g).

Question 10

Let $0 < \alpha < .5$ be some constant. Consider running the Partition subroutine on an array with no duplicate elements and with the pivot element chosen uniformly at random (as in QuickSort and RSelect). What is the probability that, after partitioning, both subarrays (elements to the left of the pivot, and elements to the right of the pivot) have size at least α times that of the original array?

- $1-\alpha$
- $2-2\alpha$
- α
- $1-2\alpha$

Question 11

Which of the following statements hold? (As usual n and m denote the number of vertices and edges, respectively, of a graph.)

- Depth-first search can be used to compute the strongly connected components of a directed graph in $O(m+n)$ time.
- Breadth-first search can be used to compute the connected components of an undirected graph in $O(m+n)$ time.
- Depth-first search can be used to compute a topological ordering of a directed acyclic graph in $O(m+n)$ time.
- Breadth-first search can be used to compute shortest paths in $O(m+n)$ time (when every edge has unit length).

Question 12

When does a directed graph have a unique topological ordering?

- Whenever is has a unique cycle
- Whenever it is a complete directed graph
- None of the other options
- Whenever it is directed acyclic

Question 13

Suppose that a randomized algorithm succeeds (e.g., correctly computes the minimum cut of a graph) with probability p (with $0<p<1$). Let ϵ be a small positive number (less than 1). How many independent times do you need to run the algorithm to ensure that, with probability at least $1-\epsilon$, at least one trial succeeds?

- $\log(1-p)\log\epsilon$
- $\log\epsilon\log(p)$
- $\log(p)\log\epsilon$
- $\log\epsilon\log(1-p)$

Question 14

Suppose you implement the operations Insert and Extract-Min using a *sorted* array (e.g., from biggest to smallest). What is the worst-case running time of Insert and Extract-Min, respectively? (Assume that you have a large enough array to accommodate the Insertions that you face.)

- $\Theta(\log n)$ and $\Theta(1)$
- $\Theta(1)$ and $\Theta(n)$
- $\Theta(n)$ and $\Theta(n)$
- $\Theta(n)$ and $\Theta(1)$

Question 15

Which of the following patterns in a computer program suggests that a heap data structure could provide a significant speed-up (check all that apply)?

- [ ] Repeated lookups
- [ ] Repeated maximum computations
- [ ] None of these
- [ ] Repeated minimum computations

Question 16

Which of the following patterns in a computer program suggests that a hash table could provide a significant speed-up (check all that apply)?

- [ ] Repeated lookups
- [ ] Repeated maximum computations
- [ ] None of these
- [ ] Repeated minimum computations

Question 17

Which of the following statements about Dijkstra's shortest-path algorithm are true for input graphs that might have some negative edge lengths?

- [ ] It is guaranteed to correctly compute shortest-path distances (from a given source vertex to all other vertices).
- [ ] It may or may not terminate (depending on the graph).
- [ ] It may or may not correctly compute shortest-path distances (from a given source vertex to all other vertices), depending on the graph.
- [ ] It is guaranteed to terminate.

Question 18

Suppose you are given k sorted arrays, each with n elements, and you want to combine them into a single array of kn elements. Consider the following approach. Divide the k arrays into $k/2$ pairs of arrays, and use the Merge subroutine taught in the MergeSort lectures to combine each pair. Now you are left with $k/2$ sorted arrays, each with $2n$ elements. Repeat this approach until you have a single sorted array with kn elements. What is the running time of this procedure, as a function of k and n?

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

Question 19

Running time of Strassen's matrix multiplication algorithm: Suppose that the running time of an algorithm is governed by the recurrence $T(n)=7*T(n/2)+n^2$. What's the overall asymptotic running time (i.e., the value of $T(n)$)?

- $\theta(n^2\log n)$
- $\theta(n_{\log 2 \log 7})$

- $\theta(n^2)$
- $\theta(n^{\log_2(7)})$

Question 20

Recall the Master Method and its three parameters a,b,d. Which of the following is the best interpretation of b^d, in the context of divide-and-conquer algorithms?

- The rate at which the number of subproblems is growing (per level of recursion).
- The rate at which the subproblem size is shrinking (per level of recursion).
- The rate at which the total work is growing (per level of recursion).
- The rate at which the work-per-subproblem is shrinking (per level of recursion).

Submit Answers Save Answers