QuickSort

Overview

Design and Analysis
of Algorithms I

QuickSort

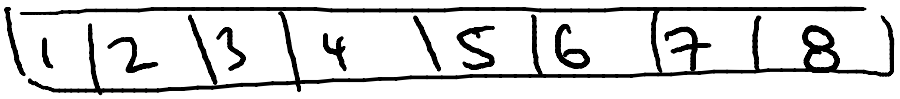
- Definitely a "greatest hit" algorithm
- Prevalent in practice
- Beautiful analysis
- $O(n \log n)$ time "on average", works in place
  - i.e., minimal extra memory needed
- See course site for optional lecture notes

Tim Roughgarden

The Sorting Problem

Input : array of n numbers, unsorted

Output : Same numbers, sorted in increasing order



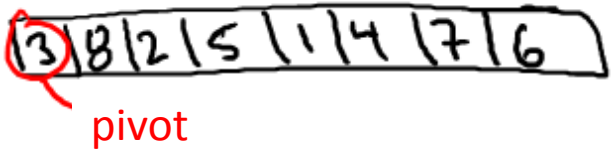
Assume : all array entries distinct.

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]

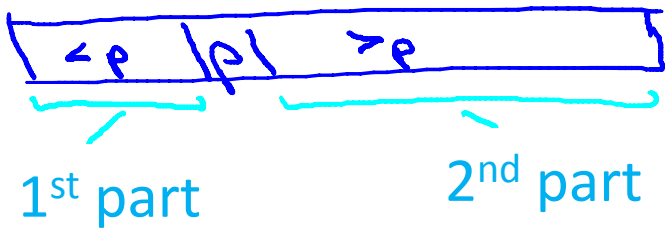
2. Reduces problem size

QuickSort: High-Level Description

[Hoare circa 1961]

QuickSort (array A, length n)
-If n=1 return
-p = 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

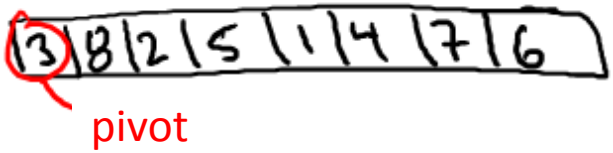
QuickSort

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”.

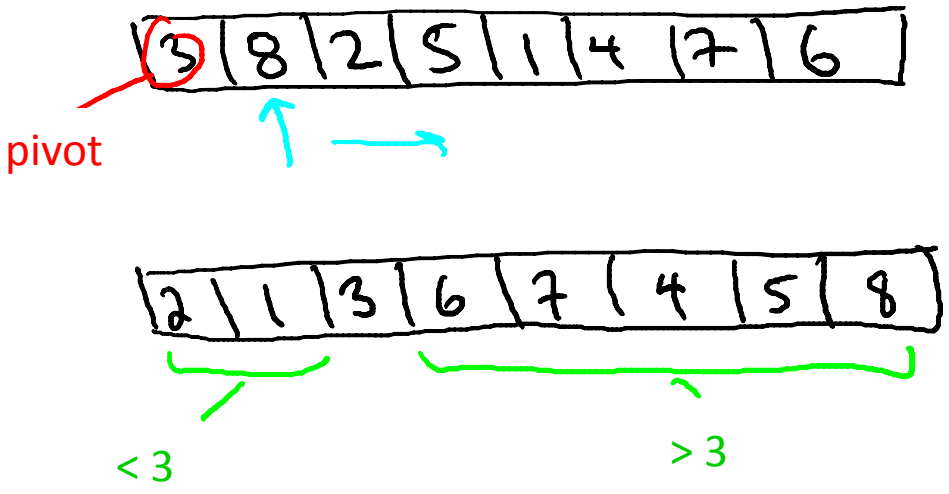
Two Cool Facts About Partition

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.

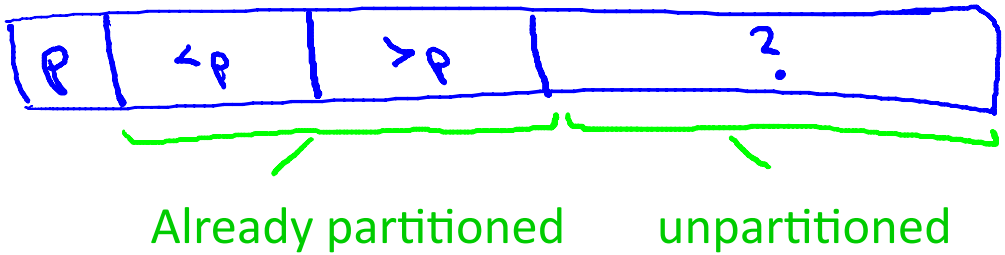


In-Place Implementation

Assume : pivot = 1st element of array

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

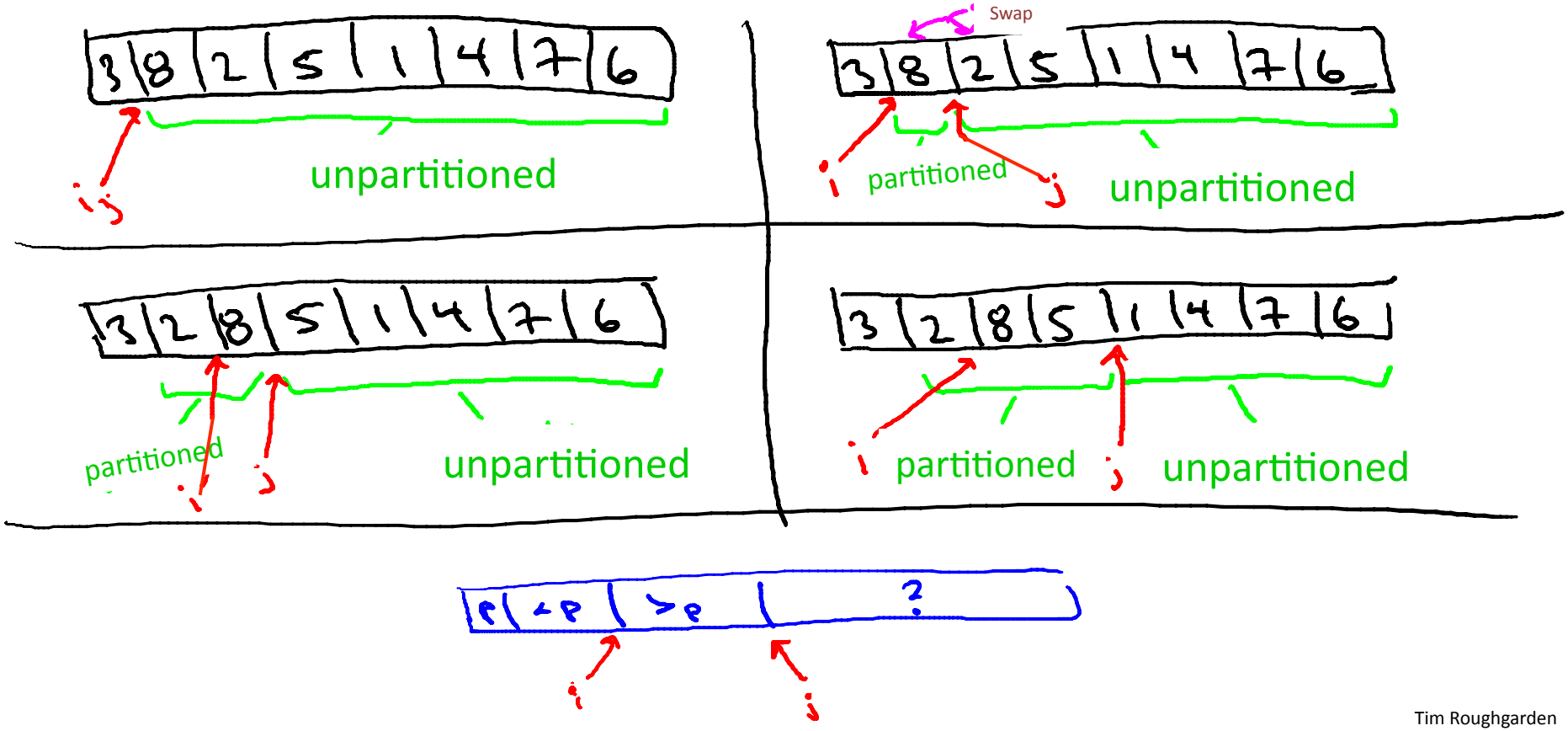
High – Level Idea :



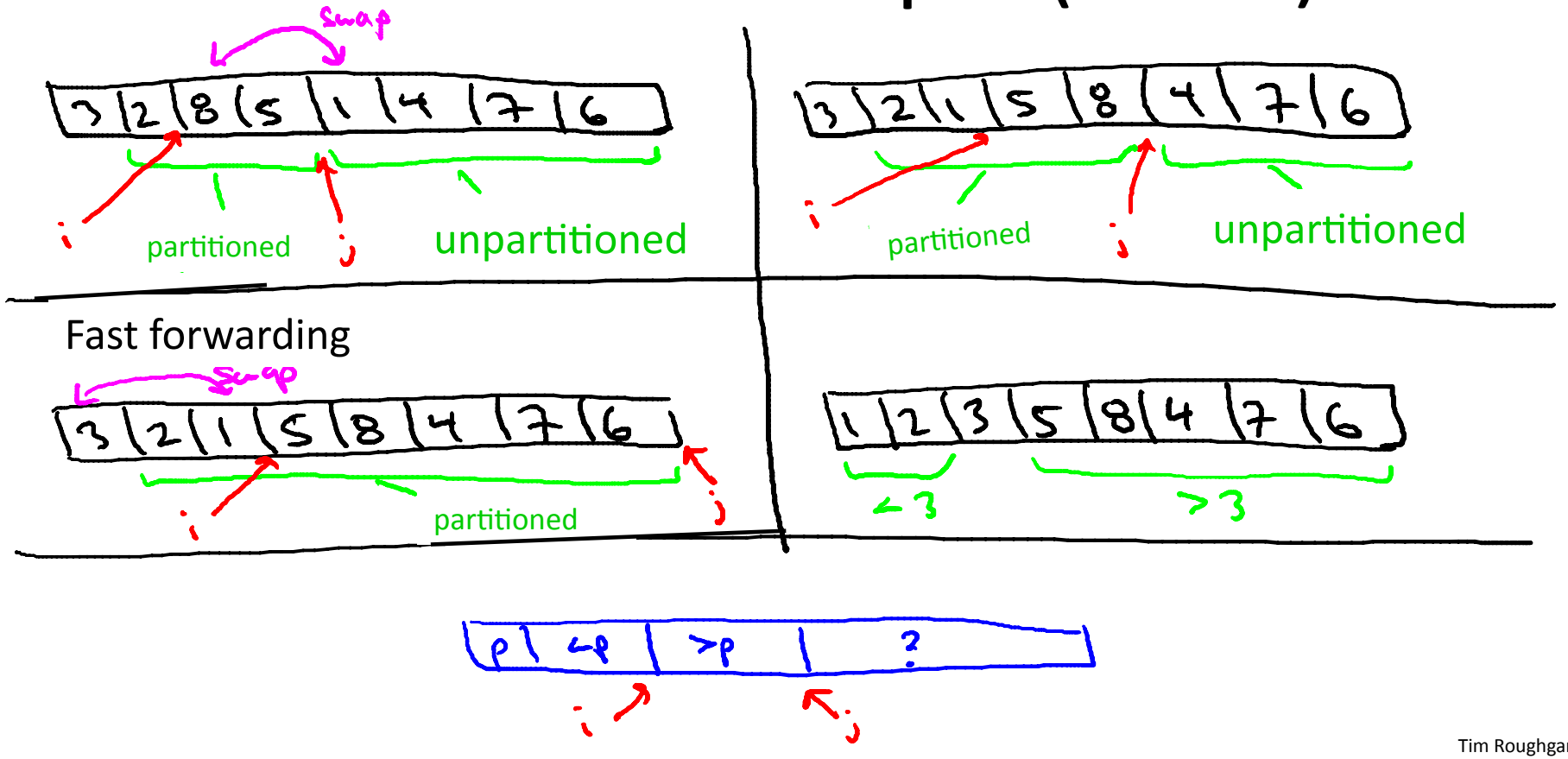
-Single scan through array

- invariant : everything looked at so far is partitioned

Partition Example



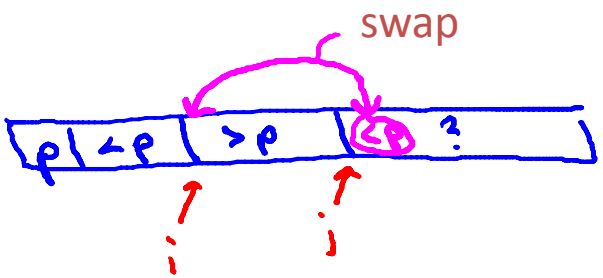
Partition Example (con'd)



Tim Roughgarden

Pseudocode for Partition

```
Partition (A,l,r)                    [ input corresponds to 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

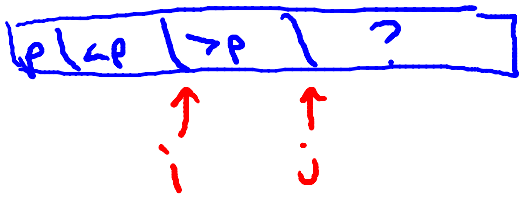
Running time = O(n), where n = r – l + 1 is the length of the input (sub) array.

Reason : O(1) work per array entry.

Also : clearly works in place (repeated swaps)

Correctness

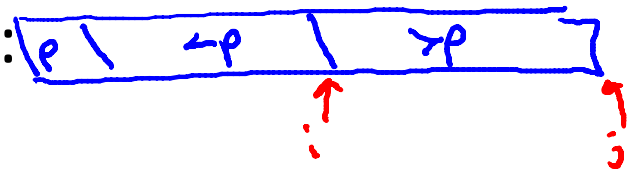
Claim : the for loop maintains the invariants :



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.

Q.E.D

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 >= 1 by induction :

1. [base case] directly prove that P(1) holds.
2. [inductive step] for every n>=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 >= 1 [no matter how pivot is chosen]

Proof by induction :

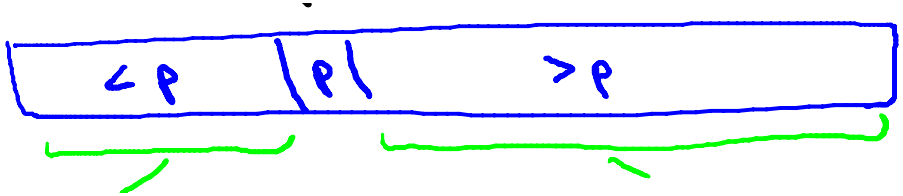
1. [base case] every input array of length 1 is already sorted.
   Quick Sort returns the input array which is correct (so P(1) holds)
2. [inductive step] Fix n>=2. Fix some input array of length n.

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

INDUCTIVE STEP

Correctness of QuickSort (con'd)

Recall : QuickSort first partitions A around some pivot p.



Note : $k_1, k_2 < n$

Note : pivot winds up in the correct position.

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

Using $P(k_1)$, $P(k_2)$

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

Tim Roughgarden

Design and Analysis
of Algorithms I

QuickSort

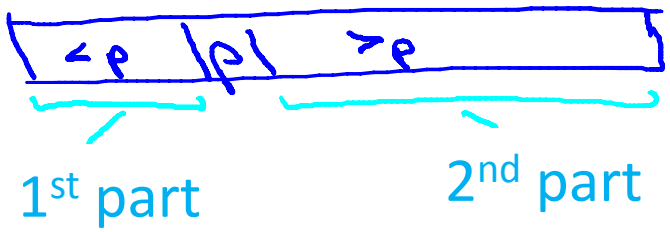
Choosing a Good Pivot

QuickSort: High-Level Description

[Hoare circa 1961]

QuickSort (array A, length n)
-If n=1 return
-p = ChoosePivot(A,n)
-Partition A around p
-Recursively sort 1st part
-Recursively sort 2nd part

[currently unimplemented]



The Importance of the Pivot

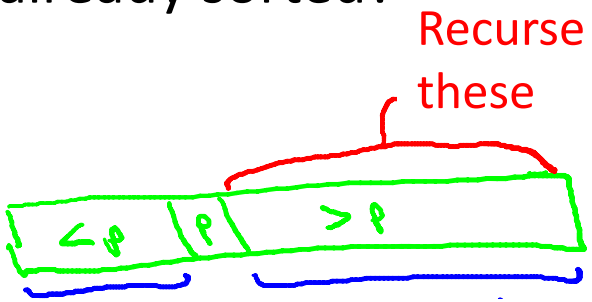
Q : running time of QuickSort ?

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

Reason :



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

Runtime : $\geq n + (n-1) + (n-2) + \ldots + 1$

$= \theta(n^2)$

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

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

Because pivot = median
choosePivot
partition

Then : $T(n) \leq 2T(n/2) + \theta(n)$

$=> T(n) = \theta(n \log n) \quad [like \quad MergeSort]$

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(nlog(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 ?

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(nlog(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)