1 Quicksort Algorithm

1.1 Quicksort Overview

- Prevalent in practice
- $O(n \log n)$ time "on average", works in place, minimal extra memory needed

1.1.1 The Sorting Problem

Input: array of n numbers, unsorted

Ouput: Same numbers, sorted in increasing order

Assume: all array entries distinct

1.1.2 Partitioning around a pivot

Key idea: partition array around a pivot element

- pick element of array
- rearrange array so that:
  - left of pivot $\Rightarrow$less than pivot
  - right of pivot $\Rightarrow$greater than pivot

Note: puts pivot in its "rightful position"

1.1.3 Two cool factors about partition

Linear ($O(n)$) time, no extra memory

Reduces problem size

1.1.4 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
```

1.2 Partitioning Around a Pivot

1.2.1 The Easy Way Out

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

[3:00]

1.2.2 In-Place Implementation

Assume: pivot = 1st element of array

[if not, swap pivot $\longleftrightarrow$1st element as preprocessing step]

High-level Idea: [5:30]

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

1.2.3 Partition Example

[7:20-16:00]

1.2.4 Pseudocode for Partition

```
Partition(A, l, r)
        p := A[l]          #pivot, first entry in array
        i := l + 1         # just to right of pivot
        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]
```

1.2.5 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)

1.2.6 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

Consequence: at end of for loop, have

$\mathbf{p}$	$< \mathbf{p}$	$^{(i)}$	$> \mathbf{p}$	$^{(j)}$

after final swap, array partitioned around pivot.

1.3 Choosing a Good Pivot

1.3.1 The Importance of the Pivot

Q: Running time of QuickSort?

A: Depends on the quality of the pivot.

The running time of QuickSort on an already sorted array if the pivot is always the first element is $O(n^2)$.

Reason: unbalanced split

$<$ **p** {empty}	p	$>$ **p**{length n-1 (still sorted), recurse on these}

Running time: $\geq n + (n-1) + (n-2) + \ldots + 1 = O(n^2)$

Suppose we run QuickSort on some input, and, magically, every recursive call chooses the median element of its subarray as its pivot, the running time in this case is $\Theta(n \log n)$.

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

Then: $T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$

$\frac{n}{2}$ because pivot = median

$\Theta(n)$ choose pivot, partition

$\Rightarrow T(n) = \Theta(n \log n)$

1.3.2 Random Pivots

Key question: how to choose pivots?

Big Idea: Random Pivots

That is: 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. [non-trivial exercise: prove via recursion tree]
2. Half of elements give a 25-75 split or better

Does this really work?

1.3.3 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]

"Average" is over random choices made by the algorithm.