Two Problems with Simple Sorts

- They might compare every pair of elements
  - learn only one piece of information/comparison
  - contrast with binary search: learns N/2 pieces of information with first comparison
- They often move elements one place at a time (bubble and insertion)
  - even if the element is “far” out of place
  - contrast with selection sort: moves each element exactly to its rightful place
- Faster sorts attack these two problems

Quicksort: Background

- ‘Easy’ to implement
- Works well with variety of input data
- Consumes few resources (memory)
Quicksort: Divide and Conquer

- **Base case:**
  - arrays of length 0 or 1 are trivially sorted

- **Inductive step:**
  - guess an element elt to partition array
  - form array of [LHS]elt[RHS] (divide)
    - for all i in LHS, i <= elt
    - for all j in RHS, j >= elt
  - recursively sort [LHS] and [RHS] (conquer)

Quicksort with Simple Partition

Algorithm QuickSort(a[], left, right)

- **Input:** array a of distinct elements, integers left and right
- **Output:** sorted array a

1. if left >= right then return pivot <- Partition(a, left, right)
2. QuickSort(a, left, pivot-1)
3. QuickSort(a, pivot+1, right)

If base case, return
Else divide (partition and find pivot)
And conquer (recursively QuickSort)

*Note that pivot is not part of either recursive call*

How to Form [LHS]elt[RHS]?*

- **Divide and conquer algorithm**
  - ideal division: equal-sized LHS, RHS

- **Ideal division is the median**
  - unfortunately, to find median, need to sort!

- **Simple choice:** just pick any element
  - if array is random, as likely good as bad
  - however, not guaranteed to be a good pick
Simple Partition

Algorithm Partition(a, left, right)
Input: array a of distinct elements, integers left and right
Output: integer
p ← a[right], lhs ← left, rhs ← right-1
while lhs ≤ rhs do
  while lhs ≤ rhs and a[lhs] ≤ p do
    lhs = lhs + 1
  while rhs ≥ lhs and a[rhs] ≥ p do
    rhs = rhs -1
  if lhs < rhs then
    swap(a[lhs], a[rhs])
swap(a[lhs], a[right])
return lhs

Choose a[right] as pivot
Do until cross:
– scan from left, looking for >= pivot
– scan from right, looking for <= pivot
– swap them
Move pivot to 'middle'

Analysis

■ Cost of partitioning N elements: O(N)
■ Worst case: pivot always leaves one side empty
  – T(N) = N + T(N-1) + T(1)
  – T(N) = N + T(N-1) [since T(1) is zero]
  – T(N) ~ N/2 = O(N)
■ Best case: pivot divides things equally
  – T(N) = N + T(N/2) + T(N/2)
  – T(N) = N + 2T(N/2)
  – T(N) = N log N = O(N log N) [with brute-force or telescoping]
■ Average case: tricky
  – 2N ln N ~ 1.39 N log N = O(N log N)

Quicksort

Advantages
■ on average, n log n time to sort n items
■ short inner loop
  – Partition costs O(n)
■ efficient memory usage
■ thoroughly analyzed and understood
Quicksort

Disadvantages
- worst case, $n^2$ time to sort $n$ items
- not stable
  - and surprisingly difficult to make stable
- fragile
  - simple implementation mistakes are costly

Improving Splits
- Key to performance: a "good" split
  - any single choice could always be worst one
  - too expensive to actually compute best one (median)
- Rather than compute median, sample it
  - simple way: pick three elements, take their median
  - very likely to give you better performance
- Sampling is a very powerful technique!

Other Improvements
- Reduce the cost of "little" sorts
  - divide and conquer: most sorts are little
  - insertion sort is faster than quicksort on small arrays
  - can bail out of quicksort when
    - right-left is smaller than some $k$
  - can also leave small arrays unsorted
    - use a single (fast!) insertion sort pass at the end
Summary: Quicksort

- On average, \( n \log n \) sort
- Efficiency based upon selection of pivot
  - randomly choose last key in partition, or
  - sample three keys, or ...
- Other methods of tuning
  - use other sort when partition is ‘small’