CS 4102, Algorithms: More Sorting

- Let’s finish some sorting problems
- Chapter 6 in the textbook
- Insertion Sort, Quicksort
- Lower bound for sorting using key comparisons

Reminder: Common Forms of Recurrence Equations

- Remember these?
  - Divide and conquer: Like **Mergesort**
    $T(n) = bT(n/c) + f(n)$
    - Solve directly or apply master theorem
  - Chip and conquer:
    $T(n) = T(n-c) + f(n)$
    - Note: One subproblem of lesser cost!
    - **Insertion sort** will be like this.
  - Chip and Be Conquered:
    $T(n) = b\, T(n-c) + f(n)$ where $b > 1$
    - Like Towers of Hanoi
    - Exponential! See recursion tree argument on p. 140

Insertion Sort

- The strategy:
  1. First section of list is sorted (say i-1 items)
  2. Increase this partial solution by…
  3. Shifting down next item beyond sorted section (i.e. the ith item) down to its proper place in sorted section. (Must shift items up to make room.)
  4. Since one item alone is already sorted, we can put steps 1-3 in a loop going from the 2nd to the last item.
- Note: Example of general strategy:
  Extend a partial solution by increasing its size by one. (Possible name: *decrease and conquer*)

Insertion Sort: Pseudocode from text

```
insertion_sort(a) {
    n = a.last
    for i = 2 to n {
        val = a[i]      // save a[i] so it can be inserted
        j = i – 1       // into the correct place
        // if val < a[j],move a[j] right to make room for a[i]
        while (j ≥ 1 && val < a[j]) {
            a[j + 1] = a[j]
            j = j - 1
        }
        a[j + 1] = val // insert val
    }
}
```

Insertion sort in Python

```
def insertion_sort(list):
    n = len(list)
    for i in range(1,n):
        val = list[i]
        j = i-1
        while j >= 0 and val < list[j]:
            list[j+1] = list[j]
            j = j-1
        list[j+1] = val
    return
```

Properties of Insertion Sort

- Easy to code
- In-place
- What's it like if the list is sorted?
  - Or almost sorted?
- Fine for small inputs
  - Why?

Insertion Sort: Analysis

- Worst-Case: $W(n) = \sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$
- Average Behavior
  - Average number of comparisons in inner-loop?

$$\frac{i}{(i+1)} \sum_{j=1}^{i} j + \frac{i}{(i+1)} = \frac{i}{2} + 1 - \frac{1}{(i+1)}$$

    - So for the i[th] element, we do roughly i/2 comparisons
  - To calculate A(n), we note i goes from 2 to n-1

$$A(n) = \sum_{i=2}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{(i+1)} \right) \approx \frac{n^2}{4}$$

- Best-case behavior? One comparison each time

$$B(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Insertion Sort: Best of a breed?

- We know that other I.S. is one of many quadratic sort algorithms, and that log-linear sorts (i.e. $\Theta(n \lg n)$) do exist
- But, can we learn something about I.S. that tells us what it is about I.S. that "keeps it" in the slower class?
  - Yes, by a lower-bounds argument on a restricted set of sort algorithms
  - BTW, this is another example to show you how to make arguments about lower-bounds

Removing Inversions

- Define an *inversion* in a sequence:
  A pair of elements that are out of order
  - E.g. { 2, 4, 1, 5, 3 } not sorted and has 4 inversions:
    pairs (2,1) (4,1) (4,3) (5,3)
  - To sort, we must fix each of these
  - What's the maximum possible number of inversions?
    n(n-1)/2 all possible pairs
    This really can occur, e.g. { 5, 4, 3, 2, 1 }
- Insertion sort only swaps adjacent elements
  - This can only remove at most one inversion!
  - Insertion sort only removes at most one inversion for each key comparison

Lower-bounds and Insertion Sort

- Theorem
  - Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must do at least n(n-1)/2 comparisons in the worst case and at least n(n-1)/4 comparisons on the average (for n elements)
  - Proof of average case? See text…
- Conclusion: Insertion Sort is optimal for algorithms that works "locally" by interchanging only adjacent elements.
  - These include BubbleSort, SelectionSort
- And, for any algorithm to be $o(n^2)$ it must swap elements that are not adjacent!

Quicksort: Introduction

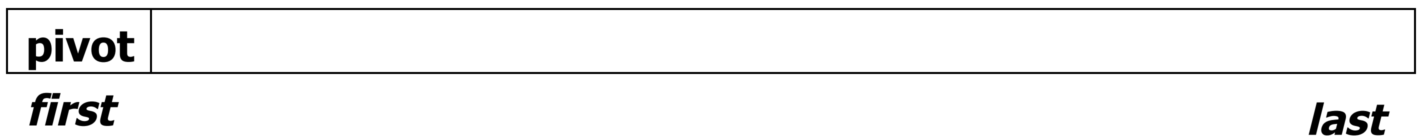
- Developed by C.A.R. (Tony) Hoare (a Turing Award winner)
  http://www.wikipedia.org/wiki/C._A._R._Hoare
  - Published in 1962
- Classic divide and conquer, but…
  - Mergesort does no comparisons to divide, but a lot to combine results (i.e. the merge) at each step
  - Quicksort does a lot of work to divide, but has nothing to do after the recursive calls. No work to combine. (If we're using arrays. Linked lists? Re-examine later.)
- Dividing done with algorithm often called *partition*
  - Sometimes called *split*. Several variations.

Quicksort's Strategy

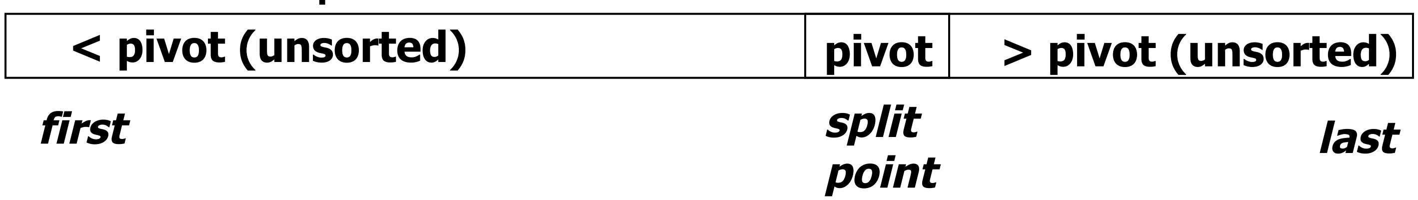
- Called on subsection of array from first to last
  - Like mergesort
- First, choose some element in the array to be the *pivot* element
  - Any element! Doesn't matter for correctness.
  - Often the first. Or, we often move some element into the first position (to get better efficiency)
- Second, call partition, which does two things:
  - Puts the pivot in its proper place, i.e. where it will be in the correctly sorted sequence
  - All elements below the pivot are less-than the pivot, and all elements above the pivot are greater-than
- Third, use quicksort recursively on both sub-lists

Quicksort's Strategy (a picture)

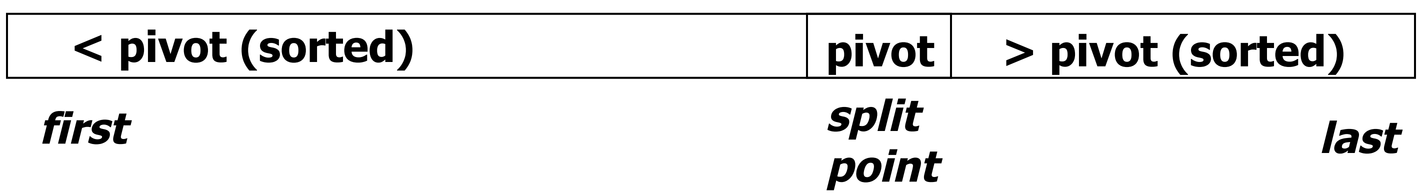
- Use first element as pivot (or pick one and move it there)



- After call to partition…



- Now sort two parts recursively and we're done!



- Note that splitPoint may be anywhere in *first..last*
- Note our assumption that all keys are distinct

Quicksort Code

Input Parameters: *list, first, last*
Output Parameters: *list*

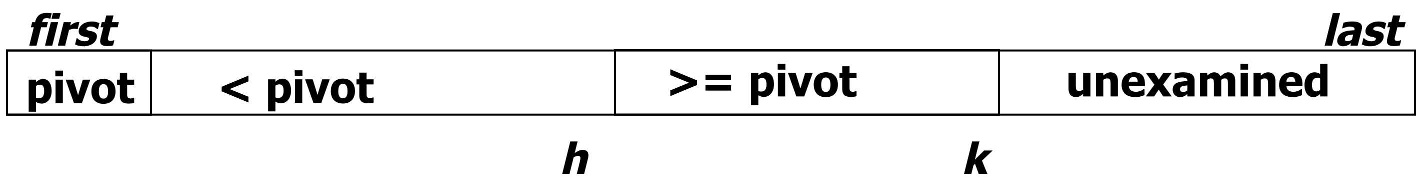
```
def quicksort(list, first, last):
    if first < last:
        p = partition(list, first, last)
        quicksort(list, first, p-1)
        quicksort(list, p+1, last)
    return
```

Partition Does the Dirty Work

- Partition rearranges elements
  - How? How many comparisons? How many swaps?
- How? Two algorithms
  - In the chapter, Lomuto's algorithm
  - In the exercises, an interesting alternative: Hoare's algorithm. (Page 269. Look at on your own.)
  - Important: Both are in-place!

Strategy for Lomuto's Partition

- Invariant:
  - *h* indexes the right-most element <*pivot*
  - *K* indexes the right-most element >= *pivot*



- Strategy:
  - Increment *k* and look at next item *a[k]*
  - If that item >= pivot, all is well!
  - If that item < pivot, increment *h* and then swap items at positions *h* and *k*
  - When done, swap pivot with item at position h

Lomuto's Partition: Code

Input Parameters: *list, first, last*
Output Parameters: *list.* Return value: the split point

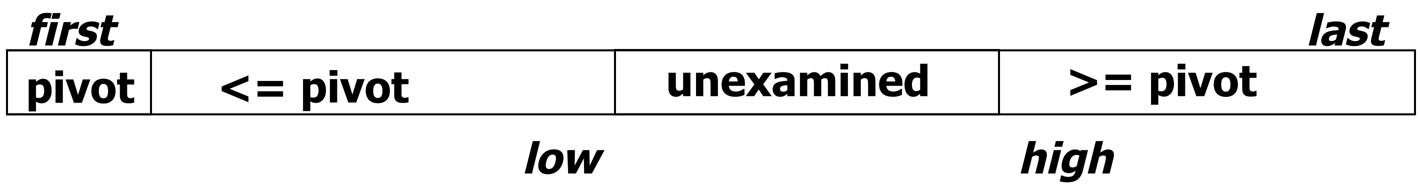
```
def partition(list, first, last):
    val = list[first]
    h = first
    for k in range(first+1,last+1):
        if list[k] < val:
            h = h+1
            (list[h],list[k]) = (list[k],list[h]) # swap!
    (list[first],list[h]) = (list[h],list[first]) # swap!
    return h
```

Strategy for Hoare’s Partition

- Hoare’s strategy: “Burn the candle from both ends!”
  - Move items bigger than pivot to the end
  - Move items smaller than pivot to beginning
  - Items still to be examined are in the middle
  - Keep two indexes pointing into the array to separate the 3 sections
  - These indexes move towards each other. Done when they meet.

Strategy for Hoare's Partition

- Invariant:
  - *low* indexes the right-most element <= *pivot*
  - *high* indexes the left-most element >= *pivot*



- Strategy:
  - Move *low* up until we find element >*pivot*
  - Move *high* down until we find element < *pivot*
  - Swap them to restore invariant

Code for Hoare's Partition

```
int Partition( Key E[], Key pivot, int first, int last ) {
   int low = first;
   int high = last+1;
   int tmp;
   while (true) {
         while ( pivot < E[--high] ) ; // see Note A next slide
         while ( E[++low] < pivot )
                  if ( low == last ) break; // see Note B next slide
         if ( low >= high ) break;
         tmp = E[low]; E[low] = E[high]; E[high] = tmp; // swap
   }
   tmp = E[high]; E[high] = E[first]; E[first] = tmp; // swap
   return high;
}
```

Python Code for Hoare's Partition

```
def partition2(list, first, last):
    i = first      # used like low in previous code
    j = last+1     # used like high in previous code
    pval = list[first]
    while True:
        while True:
            i = i+1
            if i > last or list[i] >= pval: break
        while True:
            j = j-1
            if list[j] <= pval: break
        if i < j:   (list[i],list[j]) = (list[j],list[i]) # swap
        else:       break
    (list[first],list[j]) = (list[j],list[first]) # swap
    return j
```

Notes on Partition Code

- Essentially same strategy as in textbook (p. 269)
  - Note low-section contains elements <= pivot and high-section contains elements >= pivot
- Note A: two inner while-loops move indexes, skipping over elements already in correct section
  - Stops when hits elements that need swapping

Notes on Partition Code (cont'd)

- Convince yourself this version works!
  - It's easy to goof up this code. Have we?
  - What does it do in extreme cases? E.g. pivot is max or min, or all values equal
  - Self-test exercise: use an example in text, and do one call to Partition on entire array.
    - Draw array and show values for high, low at start of outer while-loop
- Duplicate values
  - Some variants of partition handle this better
  - Common situation (E.g. sort all students by major)
  - See other texts (e.g. Sedgewick's algorithms text) for more analysis

Efficiency of Quicksort

- Partition divides into two sub-lists, perhaps unequal size
  - Depends on value of pivot element
- Recurrence for Quicksort

  T(n) = partition-cost +

  T(size of 1st section) + T(size of 2nd section)
- If divides equally, $T(n) = 2\,T(n/2) + n-1$
  - Just like mergesort
  - Solve by substitution or master theorem

    $T(n) \in \Theta(n \lg n)$
- This is the best-case. But…

Worst Case of Quicksort

- What if divides in most unequal fashion possible?
  - One subsection has size 0, other has size n-1
  - T(n) = T(0) + T(n-1) + n-1
  - What if this happens every time we call partition recursively?

$$W(n) = \sum_{k=2}^{n} (k-1) \in \Theta(n^2)$$

  - Uh oh. Same as insertion sort.
    - "Sorry Prof. Hoare – we have to take back that Turing Award now!"
    - Not so fast…

Quicksort's Average Case

- Good if it divides equally, bad if most unequal.
  - Remember: when subproblems size 0 and n-1
  - Can worst-case happen?
    Sure! Many cases. One is when elements already sorted. First element is min, pivot around that.
- What's the average?
  - Much closer to the best case
  - To prove this, fun with recurrences (pages 250-252)
  - Result: If all permutations are equal, then
    $A(n) \cong 1.386\ n \lg n$ (for large n)
- So very fast on average.
- And, we can take simple steps to avoid the worst case!

Avoiding Quicksort's Worst Case

- Make sure we don't pivot around max or min
  - Find a better choice and swap it with first element
  - Then partition as before
- Recall we get best case if divides equally
  - Could find median. But this costs $\Theta(n)$. Instead…
  - Choose a random element between first and last and swap it with the first element
  - Or, estimate the median by using the "median-of-three" method
    - Pick 3 elements (say, first, middle and last)
    - Choose median of these and swap with first. (Cost?)
    - If sorted, then this chooses real median. Best case!

Tuning Quicksort's Performance

- In practice quicksort runs fast
  - A(n) is log-linear, and the "constants" are smaller than mergesort and heapsort
  - Often used in software libraries
  - So worth tuning it to squeeze the most out of it

1. Always do something to avoid worst-case
2. Sort small sub-lists with (say) insertion sort
   - For small inputs, insertion sort is fine
     - No recursion, function calls
   - Variation: don't sort small sections at all. After quicksort is done, sort entire array with insertion sort
     - It's efficient on almost-sorted arrays!

Quicksort's Space Complexity

- Looks like it's in-place, but recursion stack
  - Depends on your definition: some people define *in-place* to **not** include stack space used by recursion
    - E.g. the algorithms textbook by Cormen et. al.
    - Our book differs (p. 224)
  - How much goes on the stack?
    - If most uneven splits, then $\Theta(n)$.
    - If splits evenly every time, then $\Theta(\lg n)$.
- Ways to reduce stack-space used due to recursion
  - Various books cover the details (not ours, though)
  - First, remove 2nd recursive call (tail-recursion)
  - Second, always do recursive call on smaller section

Summary: Quicksort

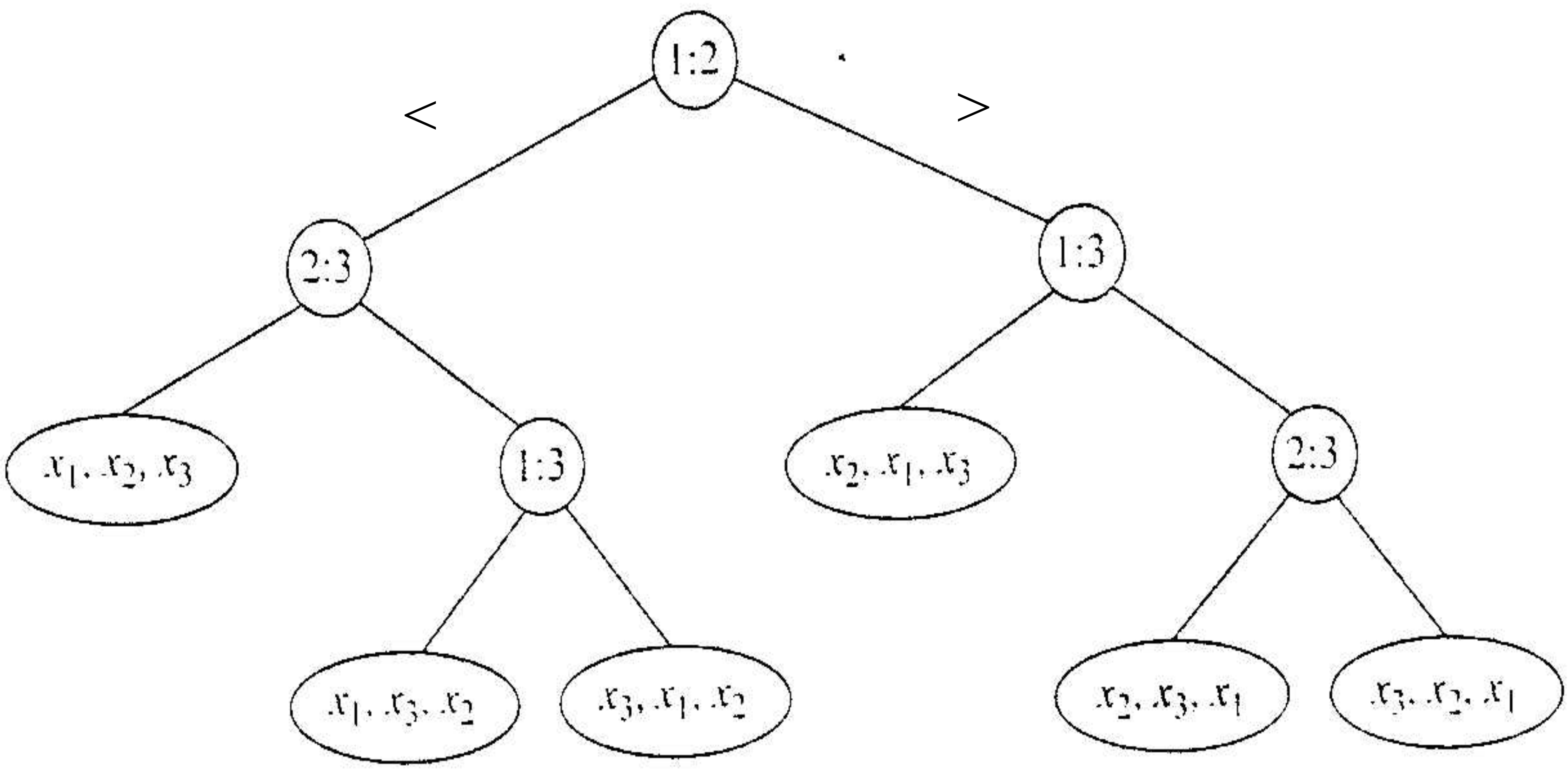
- In worst-case, efficiency is $\Theta(n^2)$
  - But easy to avoid the worst-case
- On average, efficiency is $\Theta(n \lg n)$
- Better space-complexity than mergesort.
- In practice, runs fast and widely used
  - Many ways to tune its performance
  - Can be combined effectively
- Various strategies for Partition
  - Some work better if duplicate keys
- See Sedgewick's algorithms text for more details
  - He's the expert! PhD on this under Donald Knuth

Lower Bounds for Sorting by Comparison of Keys

- What's the best possible sorting algorithm?
  - Lower Bound for Worst Case and for Average Behavior
- We'll use another kind of decision tree for analyzing the class of all sorting algorithms that compare keys
  - Each internal node represents one comparison for keys x_i and x_j; labeled i :j
  - Leaf nodes are different: they represent a particular result. I.e. a permutation of the original sequence
  - The action of Sort on a particular input corresponds to following one path in its decision tree from the root to a leaf.
  - (We assume the keys in the array to be sorted are distinct.)
- What can we say about such trees?
  - Since a correct sort must handle all permutations of n items, there must be at least n! leaves

Decision tree for sorting algorithms

- Remember, the action of sort on a particular input corresponds to following one path in its decision tree from the root to a leaf.
- Some sort, for n = 3



Lower Bound for Worst Case

- Reminder: a tree's height is number of "levels" minus 1
  - Height of this decision tree is the W(n) number of comparisons
- From Theorem 2.6.8 (p. 91):
  - Let L be the number of leaves in a binary tree and let h be its height.
  - Then $L \le 2^h$. (Number of leaves is no more than 2^h.)
  - Therefore $h \ge \lceil \lg L \rceil$ (Height is not less than…)
  - For a correct sorting algorithm, L >= n!
  - Therefore $h \ge \lceil \lg L \rceil \ge \lceil \lg n! \rceil$
- Thus, for any algorithm that sorts by comparison of keys W(n) is at least $\lceil \lg n! \rceil$

Formula for the Lower Bound

- Can we lose that factorial? Sure.
  - Stirling's formula: $(n/e)^n$ sqrt(2πn)
    - Take the log of this approximation of n! and you'll see that it's Θ(n lg n)
  - Better to re-write, use integrals, and…
    - See me or a textbook for details (but not ours)
- Result:

$$W(n) \geq \lceil \lg n! \rceil \geq \lceil n \lg n - 1.443n \rceil$$

  which is of course Θ(n lg n)
- Mergesort is very close to optimal
  - But not for all values of n

Lower Bound for Average Behavior

- How would you find the L.B. for A(n)?
  - Consider all paths through the decision tree. (Messy, huh?)
  - We won't prove this, but it's in many textbooks (not ours)
- This Theorem has been shown:
  - The average number of comparisons done by an algorithm to sort n items by comparison of keys is at least lg n!
  - or approximately n lg n – 1.443 n
- The only difference from the worst-case lower bound is that there is no rounding up to an integer
  - the average needs not be an integer,
  - but the worst case must be.
- Note: LB for average-case is close to LB for worst-case
  - This tells us that Mergesort can't be much better on average than it is in the worst-case

Summing Up So Far

- Our lower-bound proof shows any algorithm must be Ω(n lg n) in the worst-case
  <u>if</u> it works by comparing keys
  - More precisely, $W(n) \geq \lceil \lg n! \rceil \geq \lceil n \lg n - 1.443n \rceil$
  - Algorithms that can sort any type do key-comparisons
- Mergesort and Quicksort are in this order-class
  - Mergesort is very close to the L.B. (but not in-place)
  - But quicksort will run faster generally
    - Why? Constants and lower-order terms are smaller.
      In other words, the overhead per comparison is less.
  - But Quicksort really could be $\Theta(n^2)$ at its worst
    - It does use stack space for recursion
- One more sort: Heapsort! In-place <u>and</u> Θ(n lg n)