CS 61B: Lecture 30
Wednesday, April 9, 2014

SORTING
=======
The need to sort numbers, strings, and other records arises frequently. The entries in any modern phone book were sorted by a computer. Databases have features that sort the records returned by a query, ordered according to any field the user desires. Google sorts your query results by their "relevance". We've seen that Kruskal's algorithm uses sorting. So do hundreds of other algorithms.

Sorting is perhaps the simplest fundamental problem that offers a huge variety of algorithms, each with its own inherent advantages and disadvantages. We'll study and compare eight sorting algorithms.

Insertion Sort

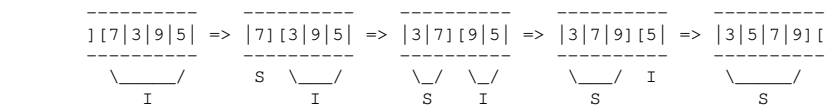
Insertion sort is very simple and runs in O(n^2) time. We employ a list S, and maintain the invariant that S is sorted.

```
  Start with an empty list S and the unsorted list I of n input items.
  for (each item x in I) {
    insert x into the list S, positioned so that S remains in sorted order.
  }
```

S may be an array or a linked list. If S is a linked list, then it takes Theta(n) worst-case time to find the right position to insert each item. If S is an array, we can find the right position in O(log n) time by binary search, but it takes Theta(n) worst-case time to shift the larger items over to make room for the new item. In either case, insertion sort runs in Theta(n^2) worst-case time--but for a different reason in each case.

If S is an array, one of the nice things about insertion sort is that it's an in-place sort. An _in-place_sort_ is a sorting algorithm that keeps the sorted items in the same array that initially held the input items. Besides the input array, it uses only O(1) or perhaps O(log n) additional memory.

To do an in-place insertion sort, we partition the array into two pieces: the left portion (initially empty) holds S, and the right portion holds I. With each iteration, the dividing line between S and I moves one step to the right.



If the input list I is "almost" sorted, insertion sort can be as fast as Theta(n)--if the algorithm starts its search from the _end_ of S. In this case, the running time is proportional to n plus the number of _inversions_. An inversion is a pair of keys j < k such that j appears after k in I. I could have anywhere from zero to n (n - 1) / 2 inversions.

If S is a balanced search tree (like a 2-3-4 tree or splay tree), then the running time is in O(n log n); but that's not what computer scientists mean when they discuss "insertion sort." This is our first O(n log n) sorting algorithm, but we'll pass it by for others that use less memory and have smaller constants hidden in the asymptotic running time bounds.

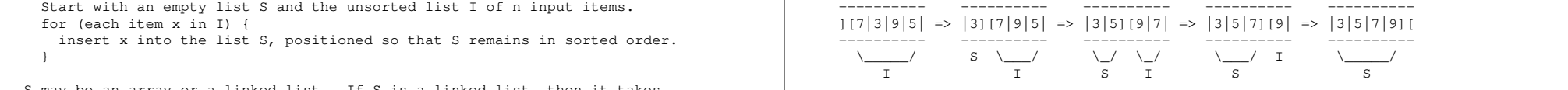
Selection Sort

Selection sort is equally simple, and also runs in quadratic time. Again we employ a list S, and maintain the invariant that S is sorted. Now, however, we walk through I and pick out the smallest item, which we append to the end of S.

```
  Start with an empty list S and the unsorted list I of n input items.
  for (i = 0; i < n; i++) {
    Let x be the item in I having smallest key.
    Remove x from I.
    Append x to the end of S.
  }
```

Whether S is an array or linked list, finding the smallest item takes Theta(n) time, so selection sort takes Theta(n^2) time, even in the best case! Hence, it's even worse than insertion sort.

If S is an array, we can do an in-place selection sort. After finding the item in I having smallest key, swap it with the first item in I, as shown here.



If I is a data structure faster than an array, we call it...

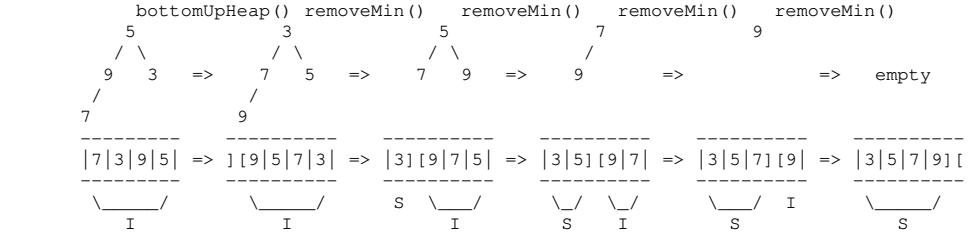
Heapsort

Heapsort is a selection sort in which I is a heap.

```
  Start with an empty list S and an unsorted list I of n input items.
  toss all the items in I onto a heap h (ignoring the heap-order property).
  h.bottomUpHeap();                          // Enforces the heap-order property
  for (i = 0; i < n; i++) {
    x = h.removeMin();
    Append x to the end of S.
  }
```

bottomUpHeap() runs in linear time, and each removeMin() takes O(log n) time.
Hence, heapsort is an O(n log n)-time sorting algorithm.

There are several ways to do heapsort in place; I'll describe just one.
Maintain the heap _backward_ at the _end_ of the array. This makes the
indexing a little more complicated, but not substantially so. As items are
removed from the heap, the heap shrinks toward the end of the array, making
room to add items to the end of S.



Heapsort is excellent for sorting arrays, but it is an awkward choice for
linked lists. The easiest way to heapsort a linked list is to create a new
array of n references to the listnodes. Sort the array of references (using
the keys in the listnodes for comparisons). When the array is sorted, link all
the listnodes together into a sorted list.

The array of references uses extra memory. There is another O(n log n)
algorithm that can sort linked lists using very little additional memory.

Mergesort

Mergesort is based on the observation that it's possible to merge two sorted
lists into one sorted list in linear time. In fact, we can do it with queues:

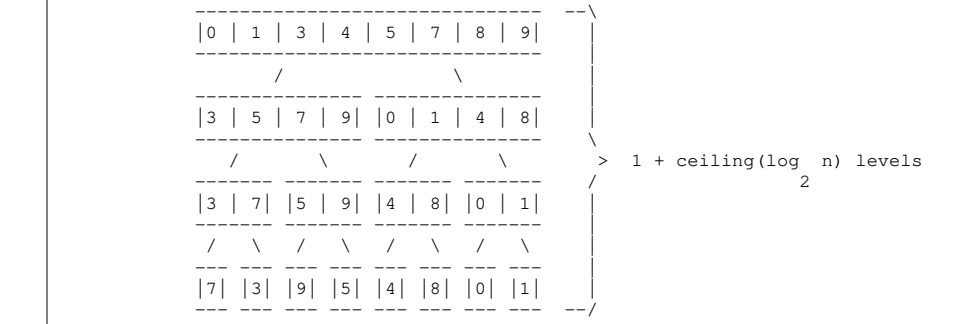
```
  Let Q1 and Q2 be two sorted queues. Let Q be an empty queue.
  while (neither Q1 nor Q2 is empty) {
    item1 = Q1.front();
    item2 = Q2.front();
    move the smaller of item1 and item2 from its present queue to end of Q.
  }
  concatenate the remaining non-empty queue (Q1 or Q2) to the end of Q.
```

The merge routine is a kind of selection sort. At each iteration, it chooses
the item having smallest key from the two input lists, and appends it to the
output list. Since the two input lists are sorted, there are only two items to
test, so each iteration takes constant time. Hence, merging takes O(n) time.

Mergesort is a recursive divide-and-conquer algorithm, in which the merge
routine is what allows us to reunite what we divided:

```
  Start with the unsorted list I of n input items.
  Break I into two halves I1 and I2, having ceiling(n/2) and floor(n/2) items.
  Sort I1 recursively, yielding the sorted list S1.
  Sort I2 recursively, yielding the sorted list S2.
  Merge S1 and S2 into a sorted list S.
```

The recursion bottoms out at one-item lists. How long does mergesort take?
The answer is made apparent by examining its recursion tree.



(Note that this tree is not a data structure. It's the structure of a sequence
of recursive calls, like a game tree.)

Each level of the tree involves O(n) operations, and there are O(log n) levels.
Hence, mergesort runs in O(n log n) time.

What makes mergesort a memory-efficient algorithm for sorting linked lists
makes it a memory-inefficient algorithm for sorting arrays. Unlike the other
sorting algorithms we've considered, mergesort is not an in-place algorithm.
There is no reasonably efficient way to merge two arrays in place. Instead,
use an extra array of O(n) size to temporarily hold the result of a merge.