CS61B: Lecture 31
Wednesday, April 9, 2014

QUICKSORT

Quicksort is a recursive divide-and-conquer algorithm, like mergesort. Quicksort is in practice the fastest known comparison-based sort for arrays, even though it has a Theta(n^2) worst-case running time. If properly designed, however, it virtually always runs in O(n log n) time. On arrays, this asymptotic bound hides a constant smaller than mergesort's, but mergesort is often slightly faster for sorting linked lists.

Given an unsorted list I of items, quicksort chooses a "pivot" item v from I, then puts each item of I into one of two unsorted lists, depending on whether its key is less or greater than v's key. (Items whose keys are equal to v's key can go into either list; we'll discuss this issue later.)

```
Start with the unsorted list I of n input items.
Choose a pivot item v from I.
Partition I into two unsorted lists I1 and I2.
  - I1 contains all items whose keys are smaller than v's key.
  - I2 contains all items whose keys are larger than v's.
  - Items with the same key as v can go into either list.
  - The pivot v, however, does not go into either list.
Sort I1 recursively, yielding the sorted list S1.
Sort I2 recursively, yielding the sorted list S2.
Concatenate S1, v, and S2 together, yielding a sorted list S.
```

The recursion bottoms out at one-item and zero-item lists. (Zero-item lists can arise when the pivot is the smallest or largest item in its list.) How long does quicksort take? The answer is made apparent by examining several possible recursion trees. In the illustrations below, the pivot v is always chosen to be the first item in the list.

```
                  ---------------------------       ---------------------------
                  |4 | 7 | 1 | 5 | 9 | 3 | 0|       |0 | 1 | 3 | 4 | 5 | 7 | 9|
v = pivot         ---------------------------       ---------------------------
                       /       |       \           / |             \
* = empty list    ----------- --- -----------     / --- -----------------------
                  |1 | 3 | 0| |4| |7 | 5 | 9|    *  |0| |1 | 3 | 4 | 5 | 7 | 9|
                I1----------- --- -----------I2     --- -----------------------
                   /   |   \   v   /   |   \         v / |           \
                  --- --- ---     --- --- ---         / --- -------------------
                  |0| |1| |3|     |5| |7| |9|        *  |1| |3 | 4 | 5 | 7 | 9|
                I1--- --- ---I2 I1--- --- ---I2         --- -------------------
                       v               v                 v / |         \
                                                          / --- ---------------
                   0   1   3   4   5   7   9             *  |3| |4 | 5 | 7 | 9|
                                                            --- ---------------
                                                             v / |       \
                                                              / --- -----------
                                                             *  |4| |5 | 7 | 9|
                                                                --- -----------
                                                                 v / |     \
                                                                  / --- -------
                                                                 *  |5| |7 | 9|
                                                                    --- -------
                                                                     v / |   \
                                                                      / --- ---
                                                                     *  |7| |9|
                                                                        --- ---
                                                                         v
```

In the example at left, we get lucky, and the pivot always turns out to be the item having the median key. Hence, each unsorted list is partitioned into two pieces of equal size, and we have a well-balanced recursion tree. Just like in mergesort, the tree has O(log n) levels. Partitioning a list is a linear-time operation, so the total running time is O(n log n).

The example at right, on the other hand, shows the Theta(n^2) performance we suffer if the pivot always proves to have the smallest or largest key in the list. (You can see it takes Omega(n^2) time because the first n/2 levels each process a list of length n/2 or greater.) The recursion tree is as unbalanced as it can be. This example shows that when the input list I happens to be already sorted, choosing the pivot to be the first item of the list is a disastrous policy.

Choosing a Pivot

We need a better way to choose a pivot. A respected, time-tested method is to randomly select an item from I to serve as pivot. With a random pivot, we can expect "on average" to obtain a 1/4 - 3/4 split; half the time we'll obtain a worse split, half the time better. A little math (see Goodrich and Tamassia Section 11.2.1) shows that the average running time of quicksort with random pivots is in O(n log n). (We'll do the analysis late this semester in a lecture on "Randomized analysis.")

An even better way to choose a pivot (when n is larger than 50 or so) is called the "median-of-three" strategy. Select three random items from I, and then choose the item having the middle key. With a lot of math, this strategy can be shown to have a smaller constant (hidden in the O(n log n) notation) than the one-random-item strategy.

Quicksort on Linked Lists

I deliberately left unresolved the question of what to do with items that have the same key as the pivot. Suppose we put all the items having the same key as v into the list I1. If we try to sort a list in which every single item has the same key, then _every_ item will go into list I1, and quicksort will have quadratic running time! (See illustration at right.)

```
---------------------------
|5 | 5 | 5 | 5 | 5 | 5 | 5|
---------------------------
           /             |
----------------------- ---
|5 | 5 | 5 | 5 | 5 | 5| |5|
----------------------- ---
I1                       v
```

When sorting a linked list, a far better solution is to partition I into three unsorted lists I1, I2, and Iv. Iv contains the pivot v and all the other items with the same key. We sort I1 and I2 recursively, yielding S1 and S2. Iv, of course, does not need to be sorted. Finally, we concatenate S1, Iv, and S2 to yield S.

```
---------------------------
|5 | 7 | 5 | 0 | 6 | 5 | 5|
---------------------------
 /         |           \
--- --------------- -------
|0| |5 | 5 | 5 | 5| |7 | 6|
--- --------------- -------
I1   v     Iv            I2
```

This strategy is quite fast if there are a large number of duplicate keys, because the lists called "Iv" (at each level of the recursion tree) require no further sorting or manipulation.

Unfortunately, with linked lists, selecting a pivot is annoying. With an array, we can read a randomly chosen pivot in constant time; with a linked list we must walk half-way through the list on average, increasing the constant in our running time. However, if we restrict ourselves to pivots near the beginning of the linked list, we risk quadratic running time (for instance, if I is already in sorted order, or nearly so), so we have to pay the price. (If you are clever, you can speed up your implementation by choosing random pivots during the partitioning step for the _next_ round of partitioning.)

```
Quicksort on Arrays
-------------------
```
Quicksort shines for sorting arrays. In-place quicksort is very fast. But a fast in-place quicksort is tricky to code. It's easy to write a buggy or quadratic version by mistake. Goodrich and Tamassia did.

Suppose we have an array a in which we want to sort the items starting at a[low] and ending at a[high]. We choose a pivot v and move it out of the way by swapping it with the last item, a[high].

We employ two array indices, i and j. i is initially "low - 1", and j is initially "high", so that i and j sandwich the items to be sorted (not including the pivot). We will enforce the following invariants.
  - All items at or left of index i have a key <= the pivot's key.
  - All items at or right of index j have a key >= the pivot's key.

To partition the array, we advance the index i until it encounters an item whose key is greater than or equal to the pivot's key; then we decrement the index j until it encounters an item whose key is less than or equal to the pivot's key. Then, we swap the items at i and j. We repeat this sequence until the indices i and j meet in the middle. Then, we move the pivot back into the middle (by swapping it with the item at index i).

An example is given at right. The randomly selected pivot, whose key is 5, is moved to the end of the array by swapping it with the last item. The indices i and j are created. i advances until it reaches an item whose key is >= 5, and j retreats until it reaches an item whose key is <= 5. The two items are swapped, and i advances and j retreats again. After the second advance/retreat, i and j have crossed paths, so we do not swap their items. Instead, we swap the pivot with the item at index i, putting it between the lists I1 and I2 where it belongs.

```
     ---------------------------
     |3 | 8 | 0 | 9 | 5 | 7 | 4|
     ---------------------------
     low              v     high

     ---------------------------
     |3 | 8 | 0 | 9 | 4 | 7 | 5|
     ---------------------------
   ^                          ^
   i                          j

     ---------------------------
     |3 | 8 | 0 | 9 | 4 | 7 | 5|
     ---------------------------
advance:  i           j

     ---------------------------
     |3 | 4 | 0 | 9 | 8 | 7 | 5|
     ---------------------------
swap:     i           j

     ---------------------------
     |3 | 4 | 0 | 9 | 8 | 7 | 5|
     ---------------------------
advance:      j   i

     ----------- --- -----------
     |3 | 4 | 0| |5| |8 | 7 | 9|
     ----------- --- -----------
     I1            i          I2
```

What about items having the same key as the pivot? Handling these is particularly tricky. We'd like to put them on a separate list (as we did for linked lists), but doing that in place is too complicated. As I noted previously, if we put all these items into the list I1, we'll have quadratic running time when all the keys in the array are equal, so we don't want to do that either.

The solution is to make sure each index, i and j, stops whenever it reaches a key equal to the pivot. Every key equal to the pivot (except perhaps one, if we end with i = j) takes part in one swap. Swapping an item equal to the pivot may seem unnecessary, but it has an excellent side effect: if all the items in the array have the same key, half these items will go into I1, and half into I2, giving us a well-balanced recursion tree. (To see why, try running the pseudocode below on paper with an array of equal keys.) WARNING: The code on page 530 of Goodrich and Tamassia gets this WRONG. Their implementation has quadratic running time when all the keys are equal.

```
public static void quicksort(Comparable[] a, int low, int high) {
  // If there's fewer than two items, do nothing.
  if (low < high) {
    int pivotIndex = random number from low to high;
    Comparable pivot = a[pivotIndex];
    a[pivotIndex] = a[high];                       // Swap pivot with last item
    a[high] = pivot;

    int i = low - 1;
    int j = high;
    do {
      do { i++; } while (a[i].compareTo(pivot) < 0);
      do { j--; } while ((a[j].compareTo(pivot) > 0) && (j > low));
      if (i < j) {
        swap a[i] and a[j];
      }
    } while (i < j);

    a[high] = a[i];
    a[i] = pivot;                  // Put pivot in the middle where it belongs
    quicksort(a, low, i - 1);                     // Recursively sort left list
    quicksort(a, i + 1, high);                   // Recursively sort right list
  }
}
```

Can the "do { i++ }" loop walk off the end of the array and generate an out-of-bounds exception? No, because a[high] contains the pivot, so i will stop advancing when i == high (if not sooner). There is no such assurance for j, though, so the "do { j-- }" loop explicitly tests whether "j > low" before retreating.

```
Postscript
----------
```
The journal "Computing in Science & Engineering" did a poll of experts to make a list of the ten most important and influential algorithms of the twentieth century, and it published a separate article on each of the ten algorithms. Quicksort is one of the ten, and it is surely the simplest algorithm on the list. Quicksort's inventor, Sir C. A. R. "Tony" Hoare, received the ACM Turing Award in 1980 for his work on programming languages, and was conferred the title of Knight Bachelor in March 2000 by Queen Elizabeth II for his contributions to "Computing Science."