CS 61B Data Structures and Programming Methodology

July 24, 2008

David Sun

Quick Sort

- 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 *Theta*(n^2) worst-case running time.
- If properly designed, however, it virtually always runs in *O(n log n)* time.

The Algorithm

1. Given an unsorted list I of n items
2. **Choose a pivot** item v from I.
3. 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. Equal values can go either way.

- (continued)

4. After the partition v is in it’s final position.
5. Sort I1 recursively, yielding the sorted list S1.
6. Sort I2 recursively, yielding the sorted list S2.
7. Concatenate S1, v, and S2 together, yielding a sorted list S.

Running Time of Quicksort

- The running time depends on whether the partitioning is *balanced* or *unbalanced*, which depends on the choice of the pivot
  - If choice of pivot is good, quicksort runs asymptotically as fast as mergesort: *Theta(n log n)*.
  - If choice of pivot is bad, then it runs asymptotically as slowly as insertion sort: *Theta*(n^2).

Worst-case Partitioning

- The worst-case behavior occurs the partitioning is *maximally unbalanced:* produces one subproblem with *n-1* elements and one with *0* elements.
- Assuming this occurs at each level of the recursive call.
  - The partitioning costs *Theta(n).*
  - Recursive call on an array of size 0 just returns, *T(0) = Theta(1).*
  - We get the recurrence *T(n) = T(n-1) + T(0) + Theta(n).*
  - Summing the cost at each level, we get $T(n) = Theta(n^2)$

Best-case Partitioning

- The most even split produces two subproblems, each of size no more than *n/2*.
- Assuming this occurs in each recursive call:
  - The running time is *T(n) <= 2T(n/2) + Theta(n),* which has the solution *T(n) = O(nlogn).*
  - Equal balancing at the two sides of the partition at every level of the recursive process produces an asymptotically faster algorithm.

Balanced Partitioning

- The *average case* running time of quicksort is much *closer* to the best case than to the worst case.
- We'll need some math to formally analyze this, but let's get a feel for why this is the case.
  - Consider a 9-to-1 proportional split…
  - Now, a 99-to-1 proportional split…

Average Case

- It's unlikely that the splits will always happen the same way in each level of the recursion.
- The average case should really consider the frequency that we expect to encounter the different splits.
  - In the average we expect the partitioning process to produce a mix of "good" and "bad" splits, evenly distributed in the tree.
  - Let's consider a simplified case where the best and worst splits happen in alternating orders…

Randomized Quicksort

- Randomly select an item from *I* to serve as pivot works well:
  - 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.
  - The expected running time of quicksort with random pivots is in *O(n log n).*
  - For large datasets, you can get better randomization by using the "median-of-three" strategy: choose three random items from *I*, and then choose the item having the middle key.

Quicksort on Linked Lists

- What do we do with items with the same key as the pivot?
  - Putting them all in the same list can cause quicksort to run in quadratic time.
- Partition *l* into three unsorted lists *l1*, *l2*, and *lv*.
  - *lv* contains the pivot *v* and all items with the same key.
  - Quicksort *l1* and *l2* recursively, giving *S1* and *S2*.
  - Concatenate the lists in order of *S1*, *lv* and *S2* to give *S*.
  - Works well with a large number of duplicated keys (since *lv* need not to be sorted).

Quicksort on Linked Lists

- But…
  - Selecting a random pivot will require walking through *half* of the list on average.
  - Restricting to pivots near the beginning of the list risks quadratic time.

Quicksort on Arrays

- Partitioning uses an additional *Theta(n)* storage space (best case) and *Theta(n^2).*
- A more complicated version can partition in-place:
  - Given an array *a* and sort all elements between *l(eft)* and *r(ight)*.
  - Choose a pivot *v* and swap with *a[r]*.
  - Initialize *i* to *l - 1*, and *j* to *r* so *i* and *j* sandwich the items to be sorted (not including the pivot).
  - 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.

Quicksort on Arrays

- (continued)
  - Advance *i* to the first *a[i]* greater than or equal to the pivot.
  - Decrement *j* until the first *a[j]* less than or equal to the pivot.
  - *a[i]* and *a[j]* are on the wrong side of the partition, so swap *a[i]* and *a[j]*.
  - Repeat until the indices *i* and *j* meet in the middle.
  - Move the pivot back into the middle – swapping the last item with *a[i]*.

```
public static void quicksort(Comparable[] a, int left, int right) {
   // If there's fewer than two items, do nothing.
   if (left < right) {
        int pivotIndex = random number from left to right;
        //Swap pivot with last item
        Comparable pivot = a[pivotIndex];
        a[pivotIndex] = a[right];
        a[right] = pivot;
        int i = left - 1;
        int j = right;
        do {
               do { i++; } while (a[i].compareTo(pivot) < 0);
               do { j--; } while ((a[j].compareTo(pivot) > 0) && (j > left));
               if (i < j) { swap a[i] and a[j]; }
        } while (i < j);
        a[right] = a[i];
        a[i] = pivot;
        // Put pivot in the middle where it belongs
        quicksort(a, left, i - 1); // Recursively sort left list
        quicksort(a, i + 1, right); // Recursively sort right list }
```

Comment

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

Readings

- Objects, Abstraction, Data Structures and Design
  - Chapter 10 pp545 - 554