EECS 114: Assignment 3

Nov 19, 2015

Due Sat 28 Nov 2015 at 11:59pm

1 Divide-and-Conquer Sorting

In the assignment you will implement two fundamental *Divide-and-Conquer* sorting algorithms, *Quicksort* and *Merge-sort*, and measure their performance on data of various size and consistency. In addition to your implementations of each, you will provide short analysis of the performance in terms of space and time complexity.

2 Quicksort

In a file named `Quicksort.java` implement the Quicksort algorithm. For additional background refer to the lecture slides or chapter 7 in the CLRS[1] textbook. The required pivot finding techniques are listed below. For debugging purposes, you may want to initially set the pivot to be index of the first element in the unsorted input sequence to the algorithm.

Once you have determined that your implementation of the algorithm sorts correctly, extend the functionality to allow for the following pivots: random, median of three, deterministic quick select. For the median of three, pick three elements from the unsorted sequence at random, then take the median of the three to be the pivot. Two selection-based pivots are presented in the next section. For testing we will call your method in the following manner:

```
int[] a_sorted = Quicksort.quicksort(a, pivot);
```

The argument `a` is an already defined array of type int and the String `pivot` indicates which pivot is used. The `pivot` can be one of the following String values: `first`, `random`, `median3`, `dqselect`.

3 Selection

For any vector `V` of comparable values, let `select(V,k)` denote the k^{th} smallest value in `V`, with `select(V,1)` being the smallest. Obviously, `select(V,k)` is the same as `V[k-1]` if and only if the values in `V` are in sorted order, which normally requires $O(n log(n))$ comparisons to achieve, where n denotes `V.size()`. We will present two versions of the QuickSelect algorithm. The randomized version has linear average-case complexity but requires n^2 comparisons in the worst case. The deterministic version has linear worst-case complexity but much higher overhead.

Median. The *median* of an odd number of values is the one in the middle. But, if there are an even number of values, there are two in the middle. Statisticians take the average of those two values, but computer scientists prefer to select one. When `V.size()` is even:

- `select( V, 1+V.size()/2 )` selects the larger of the two.[2]
- `select( V, (1+V.size())/2 )` selects the smaller of the two.

But when `V.size()` is odd both expressions select the middle value. Both are equally valid, but the latter the more common definition.

[1] Introduction to Algorithms, 3rd Edition, by Cormen, Leisersen, Rivest, and Stein

[2] Note that, if `V` is sorted, `V[i]` is equal to `select(V, i+1)`, in general, and in particular, `V[ V.size()/2 ]` is equal to `select( V, 1+V.size()/2 )`.

3.1 Random Quick Select

To find `select(V,k)` via the randomized Quick Select algorithm:

- Assert that `V` is not empty and that `k` is between `1` and `V.size()`, inclusively.
- Pick a so-called "pivot" value, `p`, at random from `V`.[3]
- Construct three new vectors `L`, `E`, and `G` containing, respectively, all entries of `V` that are **L**ess than `p`, all entries that are **E**qual to `p`, and all entries that are **G**reater than `p`.
- If `k <= L.size()`, then `select(V,k)` must be in `L`, so return `select(L,k)`, which, of course, involves a recursive call to `select`.
- Otherwise, if `k <= L.size()+E.size()`, then `select(V,k)` is surely in `E`, so simply return `p`.
- Otherwise, `select(V,k)` is surely in `G`, so decrement `k` by `L.size()+E.size()` and return the `k-th` smallest member of `G`, which again can be found via a recursive call to `select`, i.e., return `select(G,k-(L.size()+E.size()))`.

3.2 Deterministic Quick Select (Blum, Floyd, Rivest, Pratt, Tarjan)

Unfortunately, if on each recursion we happen to randomly select the smallest member of the vector of values, then all values other than that pivot will be in `G`, so the recursions will go n deep requiring $O(n^2)$ comparisons.

To force, rather than simply expect, the number of comparisons to be linear in the size of the input, we proceed as above, but, to select our pivot, we partition `V` into quintuples, find the median of each of those quintuples, and select the median of those medians. Insertion sort can be used to sort the quintuples. See CLRS *9.3 Selection in worst-case linear time*, p. 220.

Clearly, at least half of the medians will now be in `G` along with their respective above-median quintuple-mates. So at least 30% of the members of `V` are in the union of `G` and `E`, i.e., at most 70% of the members of `V` are in `L`. Similarly, at most 70% of the members of `V` are in `G`.

To aid conceptualization, imagine an array whose columns are those quintuples each order in decreasing order, top to bottom. And imagine that those quintuples are arranged in order of increasing median, left to right.

```
                    SORTED QUINTUPLES ORDERED BY MEDIANS

ABOVE     ->      *      *     *  g  g     g     g
ABOVE     ->      *      *     *  g  g     g     g
MEDIANS   ->      l ...  l ... l  p  g ... g ... g
BELOW     ->      l      l     l  l  *     *     *
BELOW     ->      l      l     l  l  *     *     *
```

4 Mergesort

In a file named `Mergesort.java` implement the Mergesort algorithm as presented below. See also, CLRS textbook *2.3.1 The divide-and-conquer approach*. For testing we will call your method in the following manner with the argument `a` being an already defined array of type int, `int[] a_sorted = Mergesort.mergesort(a);`

[3] The closer the pivot is to the median, the sooner the algorithm is likely to terminate.

```
MERGE-SORT(A, p, r)
  if p < r                                  // check for base case
      q = ⌊(p + r)/2⌋                       // divide
      MERGE-SORT(A, p, q)                   // conquer
      MERGE-SORT(A, q + 1, r)               // conquer
      MERGE(A, p, q, r)                     // combine
```

Figure 1: Merge-sort pseudocode from CLRS text, p. 34.

```
MERGE(A, p, q, r)
  n1 = q − p + 1
  n2 = r − q
  let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i − 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

Figure 2: Merge pseudocode from CLRS text, p. 31.

5 Analysis

For each of the sorting algorithms answer the following questions. Justify your answers with no more than a sentence or two and refer to specific input sequences. Be sure your include analysis for all pivot selection techniques. Put your analysis in a text file named `analysis.txt`.

1. What is the *worst-case* time complexity of the sorting algorithm? Explain.
2. What is the *average-case* time complexity of the sorting algorithm? Explain.
3. What is the *best-case* time complexity of the sorting algorithm? Explain.
4. Is the sorting algorithm *stable*? If not, why?
5. In terms of memory, what is the *space-complexity* of the algorithm? Explain.

6 Submission

Turn-in via Piazza an `assn3.tgz` tar archive that includes your `Mergesort.java, Quicksort.java, Main.java,` and `analysis.txt`. Don't forget to include the class header on each file.