CMPSCI 187: Programming With Data Structures

Lecture #34: Efficient Sorting Algorithms
David Mix Barrington
3 December 2012

Efficient Sorting Algorithms

- Sorting With O(n log n) Comparisons
- Merge Sort
- The Merge Operation
- Analyzing Merge Sort
- Quick Sort
- Analyzing Quick Sort
- Heap Sort
- Heapifying a List

Sorting With O(n log n) Comparisons

- In last week's discussion, we presented an argument that any comparison-based sorting algorithm must use at least log(n!) comparisons. This is because any such algorithm can be represented as a **decision tree**, and such a tree must have at least n! leaves in order to be able to give any of the n! possible correct answers.

- The log of n!, as a real number, is equal to log(n) + log(n-1) + log(n-2) + ... + log(2) + log(1). These are n terms, each *at most* log n, so the sum is at most n log n. But the first half of those terms are also at least log(n/2) = log(n) - 1, so the sum is at least (n/2)(log(n) - 1) which is about (1/2)(n log n). Thus log(n!) = O(n log n).

- Can we achieve a sort with only O(n log n) comparisons? We've actually seen how to do this -- enqueue each of the n elements into a heap-based priority queue, then dequeue them one by one. Each enqueue and dequeue operation takes only O(log n) comparisons. In this lecture we'll see three other ways to sort in O(n log n) comparisons, which are a bit better than this.

Merge Sort

- The first two of our three O(n log n) sorts use a **divide and conquer** strategy. We divide our list into two pieces, recursively sort each piece, then combine the two pieces. In **Merge Sort**, we divide the list arbitrarily and work to combine the two sorted lists. In **Quick Sort**, we work to divide the list into a large and a small sublist, which we can just concatenate to combine them.

- DJW's method takes a range of the array, which may be entirely unsorted, and leaves it sorted. It does nothing if the range has size 1, of course. The method it calls to do all the work takes two ranges, each assumed to be sorted, and merges them into a single sorted range. (I wouldn't have made `middle + 1` a parameter, as we always have the two ranges adjacent.)

```
static void mergeSort (int first, int last) {
   if (first < last) {
      int middle = (first + last)/2;
      mergeSort (first, middle);
      mergeSort (middle + 1, last);
      merge (first, middle, middle + 1, last);}}
```

The Merge Operation

- DJW keep the growing merged list in a copy of the values array, choosing simplicity of code over miserly use of memory. Note that they also treat the parameters `lFirst` and `rFirst` as variables within the method -- this causes no odd side effects because *primitive* parameters are **passed by value**.

```
static void merge (int lFirst, int lLast, int rFirst, int rLast) {
   int [ ] tempArray = new int[SIZE];
   int index = lFirst, saveFirst = lFirst;
   while ((lFirst <= lLast) && (rFirst <= rLast)) {
      if (values[lFirst] < values[rFirst]) {
         tempArray[index] = values[lFirst]; lFirst++;}
      else tempArray[index] = values[rFirst]; rFirst++;}
      index++;}
   while (lFirst <= lLast) {
      tempArray[index] = values[lFirst]; lFirst++; index++;}
   while (rFirst <= rLast) {
      tempArray[index] = values[rFirst]; rFirst++; index++;}
   for (index = saveFirst; index <= rLast; index++)
      values[index] = tempArray[index];}
```

Analyzing Merge Sort

- Suppose for convenience that our array size N is a power of 2, so that whenever a range is split, it is split exactly in half. How many comparisons does Merge Sort use to sort this array?

- To merge an array of size X with an array of size Y, it should be clear that the merge method uses O(X+Y) comparisons. The exact number is very close to X + Y, as for the most part each comparison leads to an item being copied into the temporary array.

- To count the total comparisons, we can divide the entire set of merge operations into phases, where the operations in each phase take a total of O(n) time. The first phase is the merging of each single element into a pair. The second is the merging of pairs into sets of size 4, the third merges those into size 8, and so on until phase number log n creates the final sorted list.

- There are thus log n phases of O(n) comparisons each for O(n log n) in all.

Quick Sort

- Quick Sort also divides the list in two, sorts each piece recursively, and combines the pieces. But here we make sure that *every* item in the first piece is less than or equal to *every* item in the second piece. We do this by taking a **pivot element** and comparing each other element to the pivot. Items smaller than the pivot go in the first piece, and items larger than it go in the second. We could put elements equal to the pivot in either -- we'll put them in the first.

- We could use a temp array as in Merge Sort, but Quick Sort is able to sort pretty much in place. We'll create a method `split` that will choose a pivot, do the comparisons, and leave the pivot in its correct place with everything before it less than or equal to it and everything after it greater than it.

```
static void quickSort (int first, int last) {
   if (first < last) {
      int splitPoint;
      splitPoint = split(first, last);
      quickSort(first, splitPoint - 1);
      quicksort(splitPoint + 1, last);}}
```

The Splitting Method

- The way to think of this is two “fingers” first and last, which move until two items can be moved to the correct side by a swap. When the fingers meet, we have identified the place into which we can swap the pivot element.

```
static int split (int first, int last) {
   int splitVal = values[first]; saveF = first;
   boolean onCorrectSide;
   first++;
   do {onCorrectSide = true;
      while (onCorrectSide)
         if (values[first] > splitVal) onCorrectSide = false;
         else {first++; onCorrectSide = (first <= last);}
      onCorrectSide = (first <= last);
      while (onCorrectSide)
         if (values[last] <= splitVal) onCorrectSide = false;
         else {last--; onCorrectSide = (first <= last);}
      if (first < last) {swap(first, last); first++; last--;}}
   while (first <= last);
   swap (saveF, last);
   return last;}
```

Analyzing Quick Sort

- The number of comparisons used by Quick Sort is more complicated to analyze, and you won't see the real answer until CMPSCI 311.
- First note that the worst-case behavior is terrible -- O(n^2) comparisons. This is because if the pivot is the first or last element, the split step does nothing.
- If the pivot were always the **median** element, on the other hand, the divide-and-conquer analysis would be the same as for Merge Sort -- O(n log n).
- The actual behavior is somewhere in between, but in CMPSCI 311 you'll show that the average-case number is O(n log n). And in fact the constant in the big-O running time is smaller than that for Merge Sort or Heap Sort. Thus the sort methods in commercial code (such as Java's `Arrays.sort`) usually use Quick Sort. The algorithm rewards effort on optimizing tricks -- see "Engineering a Sort Function" by Bentley and McIlroy.

Heap Sort

- Our last O(n log n) sort starts with the idea of our first one -- use a heap to get O(log n) enqueueing and dequeuing for a priority queue, then just put all the elements in and take them out. But Heap Sort improves this with a few tricks.
- Remember that our Heap method reheapDown took a value as parameter and placed that value in the heap in place of whatever was at the root, preserving the heap property. Here we use a variant that places the value in the first suitable place within the range given by the other two parameters.
- We form the list into a heap, then swap each element from the root to its place.

```
static void heapSort( ) {
   int index;
   for (index = SIZE/2 - 1; index >= 0; index--)
      reheapDown(values[index], index, SIZE - 1);
   for (index = SIZE - 1; index >= 1; index--) {
      swap(0, index);
      reheapDown(values[0], 0, index - 1);}}
```

Heapifying an Unsorted List

- The first loop of the Heap Sort code forms the original list into a heap in a **bottom-up** fashion. We consider each node x of the heap that is not a leaf. Assuming that the subtrees under each of x's children are heaps, but that the value at x itself may violate the heap property, we want to make the subtree under x into a heap. Once we have done this for the root, we have a heap.

- The revised reheapDown method acts just like the old one -- it begins with a hole at the given location, then moves the hole down by promoting elements until it finds a location where the given value may be inserted without violating the heap property.

- We take at most O(log n) time to reheap at each of N/2 nodes, which would be O(n log n). But note that for most nodes we only take a few steps -- the subheap has only one level for N/4 nodes, only two for N/8, only three for N/16, and so on. A careful analysis shows that we need only O(n) to heapify the list.