Faster Sorting Methods

Corky Cartwright
Stephen Wong
Department of Computer Science
Rice University

Large-scale Sorting

Given an array R of records (tuples with named fields), construct an array R' containing the same set of records as R but in ascending (non-descending) order according to a specified key field of the record. The type of the key must be totally ordered; in the simple standard case the key is an int or a long.

- Two common scenarios and one pathology:
  - R comfortably fits in main memory (internal sort)
  - R is too big to fit in main memory (external sort)
  - R barely fits in main memory (Radix sort [discussed later] with links; QuickSort uses a little extra memory)

External Sorting

- Not as important as in the past because main memories are so large.
- Modern approach: use multiple machines. If keys are randomly distributed, you can pre-partition the data into disjoint ordered chunks that will each fit in memory (with very high probability). If a chunk doesn't fit, divide it in 2). Sort each chunk internally and concatenate the sorted chunks into a single file. In practice an array of "chunk" files is probably a better representation anyway than a single file. If pre-partioning is not possible, follow the traditional approach adapted to multiple machines.
- Traditional approach: single processor and tape drives. Partition file into chunks that fit in memory. (No key distribution is assumed.) Sort each of them. Then sort the data in these ordered chunks into a single file by repeated merging. Merging can be done cleverly using a tournament tree to reduce I/O. See Knuth, Volume 3 (Sorting and Searching)
- Time is O(N log N) except for case where pre-partitioning into disjoint, ordered subsets is possible.

Large Scale Internal Sorting

- Consider the problem of sorting on the order of billion records based on an int key field. How fast can we do it? O(n^2)? O(n log n)? O(n)?
- First essential kernel: counting sort for small keys
  - Central idea (same as in hashing): index a table by the key.
  - We can count how many records have a given key value in linear time.
  - We accumulate these counts to form an offset array so that offset[i] is the index of the first record with key I when placed in sorted order.
  - Given the offset array, we can copy our original array of records to a new sorted array in linear time
- Second essential kernel: radix sorting

Stable sorting

- A sorting algorithm is stable iff it preserves the ordering of records with equal keys.

Radix sorting

- Canonical algorithm for sorting punched cards using a card sorter (obsolete).
- Stably sort a deck of cards (list of records) on each radix digit from LSD (least significant digit) to MSD (most significant digit) using a counting sort to perform each pass.

Radix sorting using an int/long key

- Divide int/long key into two/four digits of size $2^{16.}$
- Perform radix sort using these digits.
- Unbelievably fast for a large data set. Runs in linear time (worst case!)
- With randomized key distribution, we can do even better by using an MSD radix sort (which is messier). See the Sedgwick reference. But in the int case, we only need to do a counting sort on the lead 16 bit digit and clean up with “straight” insertion sorting.
- Note: straight insertion is generally the best way to sort a short array of ints; it repeatedly inserts the ith element a[I] in proper position (relative to a[0], … a[i-1]) by shifting elements among a[0], … a[i-1] greater than a[i] up one position.

Radix sorting cont.

- Perhaps a good match for C rather than Java.
  - Coding is very easy; array is only data structure.
  - Storage management is very easy.
- Java Arrays.sort(…) is much, much slower. What good is it? For smaller arrays, the overhead of traversing 2^{16} buckets overwhelms the cost of the sort.
- The power of radix sorting is under-emphasized in Comp Sci curricula.
- See *Algorithms in Java* by Robert Sedgwick
-

Minor Sorting Kernel: Straight insertion

- Same idea as functional list insertion (our first soring algorithm that we learned in Scheme) applied to arrays
- Given an array `int keys[]`, we can sort it using a for loop

```
for (int i = 1; i < keys.length; i++) {
  // insert key[i] in proper position in keys[0:i-1]
  int j = i - 1;
  int current = keys[i];
  // invariant: keys[0:j] || keys[j+1:i] is sorted,
  //   current < keys[j+1:i], 0 < i < keys.length, -1 <= j < i
  while (j >= 0 && keys[j] > current) {
    keys[j+1] = keys[j];
    j--;
  }
  keys[j+1] = current;
}
```

Exam preparation

- Read the notes on OO Design up through end of Ch. 2.
- Emphasis on how to write clean OO code using design patterns. The functional subset is important. Given a simple Scheme program manipulating inductively defined data, you should be able to convert it to a corresponding Java program (same recursion pattern) defined on a corresponding composite class hierarchy. Then perform tail recursion optimization. Then convert it to a loop. More precisely
  - Convert the data definition to OO form (composite with optional singleton).
  - Convert the Scheme function to a method defined over the composite using the interpreter pattern.
  - Convert method to tail recursive form (if possible) by introducing a help method.

Exam preparation cont.

- Convert tail recursive method with help function to a loop (with no help function). Loop iteration corresponds to a call on help function.
- Convert interpreter definition of method to visitor form.
- Understanding generics helps.

For Next Class

- See you next year in Comp 411 (nee Comp 311)?