CS 61B Data Structures and Programming Methodology

July 28, 2008

David Sun

Announcements

- Midterm II on Wed 7/31, 11:00am – 12:30pm
  - In class open book. No computational devices allowed.
  - Covers everything up and including today's lecture.
- Revision class tomorrow.
  - 45 minute mini-mock exam.
- Project 3 will be out later today.
  - Check newsgroup and course website.
  - Option to work in pairs or groups of three.
- It's your responsibility to stay up-to-date with the newsgroup!

Space Complexity

- Not-in-place version:
  - Partitioning uses an additional *Theta(n)* storage space (best case) and*Theta(n^2)* (worst case).
- To 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
}
```

Some Details

- 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.

Comparison of *O(n log n)* Algorithms

	Worst-case	**Space**	**Comments**
Quicksort	Theta(n^2)	Theta(logn) (in-place partitioning)	
Heapsort	Theta(nlogn)	Theta(1)	Heapsort requires efficient random access.
Mergesort	Theta(nlogn)	Theta(n) (on arrays)	Works well with linked lists (eg.. disk storage).

Selection

- The selection problem: we want to find the *kth* smallest key in a list, i.e. what's the *kth* item in the list when it's in sorted order?
- One approach: sort the list, then look up the item at index *k*.
- But what if we don't care if the rest of the list is in sorted order or not, is there a faster way?

Quickselct

- In quicksort observe that:
  - after the partitioning step, we have three lists: L1, Iv, and L2.
  - We know which of the three lists contains index j, because we know the lengths of I1 and I2.
  - Therefore, we only need to search one of the three lists.

Quickselect

```
Start with an unsorted list I of n input items.
Choose a pivot item v from I.
Partition I into three unsorted lists I1, Iv, 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.
Iv contains the pivot v.
if (j < |I1|) {
     Recursively find the item with index j in I1; return
it.
} else if (j < |I1| + |Iv|) {
     Return the pivot v.
} else { // j >= |I1| + |Iv|.
 Recursively find the item with index j - |I1| - |Iv| in I2;
 return it.
}
```

Comparison Sort

- All the sorting algorithms we've seen so far are ***comparison sorts****:*
  - the ordering of the elements can be determined by comparison of their keys.
  - All actions taken by the sorting algorithm are based on the results of a sequence of true/false questions (a two way decision).
- If all you can do is comparison, then it can be proven that you can do no better than $\Omega(n \log n)$ in the worst case to sort n elements
  - In this sense, merge sort and heapsort are asymptotically optimal.

Lower Bound of Comparison Sort

- Given a random scrambling of *n* numbers in an array, with each number from *1...n* occurring once. How many possible orders (or permutations) can the numbers be in?
  - The answer is ***n!***, where $n! = 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \times n$.
  - Observe that if $n > 0$,
    $n! = 1 \times 2 \times \ldots \times (n-1) \times n$ **<=** $n \times n \times \ldots \times n \times n \times n = n^n$ and
    $n! = 1 \times 2 \times \ldots \times (n-1) \times n$ **>=** $n/2 \times (n/2 + 1) \times \ldots \times (n-1) \times n$
    >= $(n/2)^{(n/2)}$
  - So $(n/2)^{(n/2)} <= n! <= n^n$.
  - Let's look at the logarithms of both these numbers:
    $\log((n/2)^{(n/2)}) = (n/2) \log (n/2)$, which is in $\Theta(n \log n)$, and $\log (n^n) = n \log n$.
  - Hence, ***log(n!)* is also in *Theta(n log n)*.**

Lower Bound of Comparison Sort

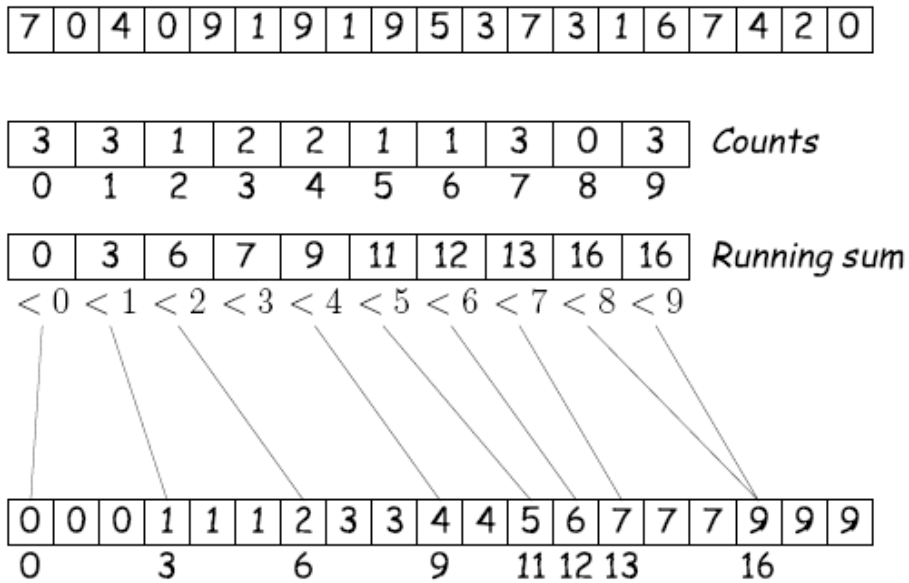
- Given *n!* of the input, a *correct* sorting algorithm must do *n!* different permutations of comparisons and swap operations.
- Therefore, there must be *n!* possible permutations of *true/false* answers for the *n!* permutations of inputs.
- If a sorting algorithm never asks more than *k true/false* questions, it generates less than or equal to 2^k different sequences of true/false answers.
  - If it correctly sorts every permutation of *1...n*, then $n! <= 2^k$, so $log_2 (n!) <= k$, and *k* is *in* $\Omega(n \log n)$.

Linear Sorting Algorithms

- But suppose can do more than comparing keys.
- What if we know *something* about the keys that are being sorted:
  - For example, how can we sort a set of n integer keys whose values range from 0 to kn , for some small constant k?
- One technique: for each element x, determine the number of elements *less* than x. We can place x immediately into its right position.
  - If M_p = #items with value $< p$, then in sorted order, the jth item with value p must be $\#M_p + j$.
- Gives linear-time algorithm.

Counting Sort

- Suppose all items are between 0 and 9:



- "Counts" line gives # occurrences of each key.
- "Running sum" gives cumulative count of keys each value which tells us where to put each key:
  - The first instance of key k goes into slot m, where m is the number of key instances that are < k.

Running Time of Counting Sort

- *Theta(n + k)* where n is the size of the input and k is the length of the counting array.
  - In order for this algorithm to be efficient, *k* must not be much larger than *n*.
- The indices of the counting array must run from the minimum to the maximum value in the input to be able to index directly with the values of the input.
- Otherwise, the values of the input will need to be translated (shifted), so that the minimum value of the input matches the smallest index of the counting array

Bucket Sort

- Again, uses the fact that
  - the keys are distributed with in some small range of values. e.g. from 0 to *q-1*, and
  - the number of items *n* is larger than, or nearly as large as, *q*, i.e. *q* is in *O(n)*.
- Allocate an array of *q* queues, numbered from 0 to *q-1*, called buckets.
- We walk through the list of input items, and enqueue each item in the appropriate queue:
  - an item with key *i* goes into queue *i*.
- When we're done, we concatenate all the queues together in order.

Running Time of Bucket Sort

- Theta(q + n) time - in the best case and in the worst case.
  - It takes Theta(q) time to initialize the buckets in the beginning
  - It takes Theta(q) to concatenate the queues in the buckets together in the end.
  - It takes Theta(n) time to put all the items in their buckets.
- If q is in O(n) - that is, the number of possible keys isn't much larger than the number of items we're sorting - then bucket sort takes Theta(n) time.

Stability of Sorting

- A sort is stable if items with equal keys come out in the same order they went in.
- Bucket sort is a stable sort.
- Previously seen sorting algorithms: insertion sort, selection sort, and mergesort can all be implemented so that they are stable.
- The linked list version of quicksort we have seen can be stable, but the array version is decidedly not.
- Heapsort is never stable.