Complexity Bounds of Comparison-Based Sorting Algorithms

No matter how much we tinker with mergesort, quicksort, heapsort, or any other sort that makes decisions by comparing elements in the array, the algorithm that results must be $\Omega(N \log N)$ for some case.

See Weiss for a sketch of this proof.
Bucket and Radix Sorting

When we started talking about sorting, we described SORT as an ADT operation on the Array ADT. The domain of SORT, until now, has been arrays whose elements support the ADT operations <, > and =.

Now we will focus on a more restricted domain (although still quite useful) and see how we can take advantage of this domain to devise sorting algorithms that may be quite efficient, or even ideal.
A Restricted Domain

• The domain we will use is somewhat difficult to define in precise mathematical terms without going off on a long digression... so we won’t bother.

• Intuitively: we will restrict the domain of the array elements to the set of sets that can be "mapped" by a function $F(x)$ to integers $0..M$ (where $M$ is constant and known in advance), such that for all $x, y, z$ in the set:
  
  • $0 \leq F(x) \leq M$
  
  • $<, =, >$ are defined:
    
    • $x > y$ if and only if $F(x) > F(y)$
    • $x = y$ if and only if $F(x) = F(y)$
    • $x < y$ if and only if $F(x) < F(y)$
    • one of these relations must be true.
    • these relations are transitive
  
  • Intuitively, the elements we are sorting behave like positive integers, and we know how large the largest possible integer will be.
  
  • For the sake of simplicity, we will always use the integers $0..M$ in our examples.
Bucket Sort Algorithm

- To sort an array \( A \) of \( N \) integers in the range \( 0..M \)
  - Get an array of length \( M + 1 \) integers (if each element in \( A \) is known to be unique, a bit array will suffice). Call this array \( B \). Initialize \( B \) to contain zeros.
  - Increment each \( B[x] \) where \( x \) is a value in \( A \):
    
    ```
    for (i = 0; i < N; i++) {
        B[A[i]]++;
    }
    ```
  - Sweep through \( B \), noting which elements were non-zero, and adding the corresponding number to the output.
    
    ```
    k = 0;
    for (i = 0; i <= M; i++) {
        for (j = 0; j < B[i]; j++) {
            A[k++] = i;
        }
    }
    ```
Bucket Sort Analysis

• Initializing the bucket array is $O(M)$.

• $O(N)$ to increment the appropriate buckets (assuming array access is $O(1)$).

• $O(\text{Max } (N,M))$ to sweep through the bucket array in the final step. This is slightly tricky, because each invocation of the inner loop will take an unknown amount of time-- but we know that the total of iterations of the inner loop is exactly $N$.

• In practical terms:

  1. If $N$ is *much* smaller than $M$, this is terrible. $O(M)$

     $N = 100, M = 1,000,000$

  2. If $N$ is within a "small" constant factor of $M$, this will run quickly but possibly consume a lot of memory. $O(N)$

  3. If $N > M$, this will be fast. $O(N)$

So it’s **not always** appropriate to use bucketsort, and it always requires allocating $O(M)$ extra space, but for some applications bucketsort is a big win.

Things to Note

• *We never* compared any pair of elements that we were sorting.

• In fact, there were not any conditional statements in the entire algorithm - just a few loops.
Stable Sorting Algorithms

Before we continue, we need to define a new term:

• A sorting algorithm is **stable** if it does not cause two array elements whose keys are equal to be inverted in the result array. (they may get re-ordered during the sorting process, but they wind up in the same relative order as they began).

**Uses:**

• Telephone Book:
  • Sort by First name, then Last name, and the result is in order by Last name and First name.

• Yearbook:
  • Sort by First name, Last name, then Class. The result is in order by Class, Last name and First name.
Bucket Sort Stability

• Bucket sort, as presented, is not stable and it only works for the very simple case of positive integers.

• We can fix this by making each "bucket" in the B array into a queue and doing ENQUEUE and DEQUEUE instead of just keeping a count.

• For example - a yearbook, already sorted by Name, being bucket sorted by Year:

  1 ➔ Alice ➔ Bobbie ➔ Susan
  2 ➔ Beth ➔ Laura ➔ Roberta
  3 ➔ Carol ➔ Diane ➔ Mary
  4 ➔ Anna ➔ Patty ➔ Terry

• The elements are ENQUEUED in the same relative order that they appeared in the original array, so all the elements in each queue are in the same relative order as they were in the original array. (A queue is a “stable” structure).
Stable Bucket Sort Algorithm

• To sort an Array A of N integer keys in the range 0..M with corresponding values.

1. Get an array of M+1 Queues, initially empty.

2.
   for (i = 0; i < N; i++) {
      Enqueue (A[i]) in the Queue A[i].key
   }

3.
   k = 0;
   for (i = 0; i <= M; i++) {
      while (!IsEmpty (Queue[i])) {
         A[k++] = Dequeue (Queue[i]);
      }
   }
Analysis of Stable Bucket Sort

• \textsc{Enqueue} and \textsc{Dequeue} can be done in $O(1)$ time (and we assume that creating, initializing, and destroying each queue is $O(1)$).

• We know (or have reason to believe) that queue access is somewhat slower than simple array access, but this doesn’t change the big-O.

• Therefore stable bucketsort is $O(\text{Max}(N,M))$, but generally slower than the ordinary bucket sort.

• \textbf{Note} - All of the other sorts we’ve seen can be implemented in a stable form.
  
  • Quicksort is mildly tricky, but not horribly so-- it all depends whether the partitioning algorithm is stable or not.
  
  • Heapsort requires some thought.

• \textbf{Pet peeve} - designers who sacrifice stability (and therefore the ability to perform “nested” sorts) in order to gain a tiny improvement in speed. Fah!
Radix Sort

Radixsort builds on these ideas - it uses a stable bucket sort repeatedly.

In the previous examples, we sorted on first name, then class in order to get a list that was sorted by class, and then first name: we sorted from the "least significant" field and worked our way up to the "most significant", but if we used a stable sort, then the keys remained in order with respect to the less significant fields during the later sorts on the more significant fields.

In radixsort, we take this idea to its logical extreme.

In radix sort, we break each key into several "sub-keys" and then do a stable bucket sort on each sub-key, starting with the least significant and working our way up to the most significant.

The keys must be divisible into sub-keys in such a way that the sub-keys can be ordered by significance.

Examples

- Base 10 integers (Radix 10) - in radix 10, each decimal digit is a sub-key. The least significant sub-key is the 1’s, then 10’s, 100’s, 1,000’s,... digit.
- Binary integers - each 4 digits (Radix 16), or even each bit (Radix 2)
Radix Sort Algorithm

• For sub-key = least significant to most significant, do a stable bucketsort of the array by this sub-key.

• For base 10 digits (radix 10) the sub-keys are
  
  \[
  (x/10^0) \% 10, (x/10^1) \% 10, (x/10^2) \% 10, \ldots (x/10^i) \% 10
  \]

• For base 16 "hexadecimal digits" (radix 16)
  
  \[
  (x >> 0) & 0xf, (x >> 4) & 0xf, \ldots (x >> (4*i)) & 0xf
  \]

Note that a radix that is a power of 2 is very convenient for sorting integers with radixsort; the value of each sub-keys can be extracted by inexpensive shift-and-mask operations.

Division and modulo are generally expensive, but division and modulo by powers of 2 can be performed by shifts and masks.

Note:

*If this all seems like C gobbledegook, don’t panic. You don’t need to know this C trivia in order to do the assignment.*