Bucket sort

CSE 30331/34331: Data Structures

October 24, 2015

Previously we saw how any priority queue data structure induces a sorting algorithm: unsorted lists give selection sort, sorted lists give insertion sort, binary heaps give heap sort.

What about hash tables? Well, hash tables aren't priority queues. They don't even require their element type to be linearly ordered (that is, to define `operator<`). So the question doesn't seem to make sense. But, if we tweak them a little bit, they do lead to a sorting algorithm – a fast one.

1 Pigeonhole sort and counting sort

Let's back up and think about bit vectors again, which we introduced as a very simple kind of hash table for integers in a known range $[0, m)$. In this case, the element type *is* linearly ordered and we *do* know how to find the minimum element.

If we need to sort n unique integers in the range $[0, m)$, all we have to do is put them all into a bit vector, then iterate over the bit vector in order, outputting those i for which bit i is set. This algorithm runs in $O(n + m) = O(m)$ time. In pseudocode:

Precondition: a is an array of n unique integers, $0 \leq a[i] < m$ for $0 \leq i < n$
Postcondition: a is sorted in increasing order
$b \leftarrow$ an empty bit vector of size m
for $i \leftarrow 0, \ldots, n-1$ **do**
    assert $b[a[i]] = 0$
    $b[a[i]] \leftarrow 1$
end for
$i \leftarrow 0$
for $x \leftarrow 0, \ldots m-1$ **do**
    if $b[x] = 1$ **then**
        $a[i] = x$
        $i \leftarrow i + 1$
    end if
end for
assert $i = n$

To drop the uniqueness assumption, just add counts to the bit vector. This runs in $O(n+m)$ time. (This is the optimal solution to a question you had on the homework about sorting a large pile of coins by value.)

Precondition: a is an array of n integers, $0 \le a[i] < m$ for $0 \le i < n$
Postcondition: a is sorted in nondecreasing order
$b \leftarrow$ an array of m zeros
for $i \leftarrow 0, \ldots, n-1$ **do**
    $b[a[i]] \leftarrow b[a[i]] + 1$
end for
$i \leftarrow 0$
for $x \leftarrow 0, \ldots m-1$ **do**
    for $j \leftarrow 0, \ldots, b[x]$ **do**
        $a[i] = x$
        $i \leftarrow i + 1$
    end for
end for
assert $i = n$

If m is small – for example, in the case of US coins, $m = 6$, then this sorting algorithm is $O(n)$. Previously, the fastest sorting algorithm we've seen is $O(n \log n)$. Indeed, on the midterm exam, quite a few students asserted that there is no sorting algorithm that is faster than $O(n \log n)$. But here is such an algorithm. What's going on? All the sorts that we saw previously are called *comparison-based* sorts because the only assumption they make about the things to be sorted is that they can be compared using the less-than operator. For such sorts, the provable lower bound is $\Omega(n \log n)$. But pigeonhole and counting sort aren't comparison-based sorts. They look directly at the values being sorted, and they make additional assumptions about their range. So they're able to break the $\Omega(n \log n)$ lower bound.

2 Bucket sort

The algorithms above worked because the buckets of b were in order. But in a hash table, that is no longer true, because the hash function doesn't respect ordering. For example, if $h(x) = x \,\%\, 1000$, then 2000 goes into bucket 0, but 1001 goes into bucket 1. So if we output the contents of the buckets from 0 to 999, then 2000 will be output before 1001, which is wrong.

In bucket sort, we use a data structure that looks just like a hash table, but we put another requirement on the hash function: if $h(x_1) < h(x_2)$, then $x_1 < x_2$. For example, if we know that the keys are in the range $[0, 100000)$, then we can use $h(x) = \lfloor x/100 \rfloor$ to map the keys into 1000 buckets. Then the algorithm looks like this:

Precondition: a is an array of n objects
Postcondition: a is sorted in nondecreasing order
$b \leftarrow$ an array of m buckets

```
for i ← 0, ..., n − 1 do
    x ← a[i]
    insert x into b[h(x)]
end for
i ← 0
for k ← 0, ... m − 1 do
    sort b[k]
    for x in b[k] do
        a[i] = x
        i ← i + 1
    end for
end for
assert i = n
```

The inner sort can be any sort appropriate to the data structure used for the buckets.

What is the running time of bucket sort? If the key distribution and the hash function are such that the keys are evenly distributed among the buckets, and if we choose m to be $O(n)$ (for example, just set $m = n$), then the algorithm runs in time $O(n)$.

3 Radix sort

Above, we mentioned that the inner sort could be any sorting algorithm. It could even be another bucket sort, as long as the hash functions are defined correctly. If it's bucket sorts all the way down, this is known as MSD radix sort.

Suppose that we are sorting an array of strings. The strings contain only capital letters, but we pad the strings with trailing zeros so that they are all the same length.

Intuitively, we can think of the algorithm as follows. Sort all the strings by their first letter by putting them into 27 buckets. Then, take the bucket of strings starting with A, remove the first letter, and sort those. Similarly for the bucket of strings starting with B, and so on.

This is just bucket sort where the hash function selects the first character of the string. Now, if the inner sort is again a bucket sort, then its hash function has to select the second character of the string. And so on, so that the hash function at depth d selects the d'th character of the string.

In pseudocode, the algorithm is:

```
Precondition: a is an array of n alphabetic strings, zero-padded to length ℓ
Postcondition: a is sorted in nondecreasing order
procedure RadixSort(a, ℓ, d)
    if d = ℓ then
        return
    end if
    b ← an array of m = 27 buckets
    for i ← 0, ..., n − 1 do
```

```
        x ← a[i]
        insert x into b[x[d]]
    end for
    i ← 0
    for k ← 0, ... m − 1 do
        RadixSort(b[k], ℓ, d + 1)
        for x in b[k] do
            a[i] = x
            i ← i + 1
        end for
    end for
    assert i = n
end procedure
RadixSort(a, ℓ, 0)
```