ICS 161: Design and Analysis of Algorithms Lecture notes for January 23, 1996

Bucket Sorting

We've seen various algorithms for sorting in O(n log n) time and a lower bound showing that O(n log n) is optimal. But the lower bound only applies to comparison sorting. What if we're allowed to do other operations than comparisons? The results will have to depend on what specific data type we want to sort; typical types might be integer, floating point, or character string.

Let's start with a really simple example: We want to sort n integers that are all in the range 1..n, no two the same. How quickly can we determine the sorted order?

> Answer: O(1), without even computing anything you know it has to be 1,2,3,...n-1,n.

As a less stupid example, suppose all numbers are still in the range 1..n but some might be duplicates. We can use an array to count how many copies of each number there are:

```
sort(int n, X[n])
{
int i,j, Y[n+1]
for (i = 0; i < n; i++) Y[i] = 0;
for (i = 0; i < n; i++) Y[X[i]]++;
for (i = 0, j = 0; i < n; i++)
    while(Y[i]-- > 0) X[j++] = i
}
```

The three loops here take O(n) time (we go back to time instead of counting comparisons since this algorithm doesnt use any comparisons).

One not-completely-obvious point: one of the loops is nested. Normally when we see a collection of nested loops, the time bound is the product of the number of iterations of each loop. Why is this nested loop O(n) rather than O(n^2)?

> Answer: It's possible for the inner loop to execute as many as n times, but not on all n iterations of the outer loop. The easiest way to see this is to match up the times when we increment the array entries with the times when we decrement them. Since there are only n increments, there are also only n decrements.

This algorithm is already close to useful. But it is less likely that we want to sort numbers exactly, and more likely that we want to sort records by some number derived from them. (As an example, maybe we have the data from the UCI phone book, and we want to sort all the entries by phone number. It's not so useful to sort just the phone numbers by themselves; we want to still know which name goes with which number.)

So suppose you have a list of n records each with a key that's a number from 1 to k (we generalize the problem a little so k is not necessarily equal to n).

We can solve this by making an array of linked lists. We move each input record into the list in the appropriate position of the array then concatenate all the lists together in order.

```
bucket sort(L)
{
list Y[k+1]
for (i = 0; i <= k; i++) Y[i] = empty
while L nonempty
{
    let X = first record in L
    move X to Y[key(X)]
}
for (i = 0; i <= k; i++)
concatenate Y[i] onto end of L
}
```

There are two loops taking O(k) time, and one taking O(n), so the total time is O(n+k). This is good when k is smaller than n. E.g. suppose you want to sort 10000 people by birthday; n=10000, k=366, so time = O(n).

Stability

We say that a sorting algorithm is *stable* if, when two records have the same key, they stay in their original order. This property will be important for extending bucket sort to an algorithm that works well when k is large. But first, which of the algorithms we've seen is stable?

- Bucket sort? Yes. We add items to the lists Y[i] in order, and concatenating them preserves that order.
- Heap sort? No. The act of placing objects into a heap (and heapifying them) destroys any initial ordering they might have.
- Merge sort? Maybe. It depends on how we divide lists into two, and on how we merge them. For instance if we divide by choosing every other element to go into each list, it is unlikely to be stable. If we divide by splitting a list at its midpoint, and break ties when merging in favor of the first list, then the algorithm can be stable.
- Quick sort? Again, maybe. It depends on how you do the partition step.

Any comparison sorting algorithm can be made stable by modifying the comparisons to break ties according to the original positions of the objects, but only some algorithms are automatically stable.

Radix sort

What to do when k is large? Think about the decimal representation of a number

```
x = a + 10 b + 100 c + 1000 d + ...
```

where a,b,c etc all in range 0..9. These digits are easily small enough to do bucket sort.

```
radix sort(L):
{
bucket sort by a
bucket sort by b
bucket sort by c
...
}
```

or more simply

```
radix sort(L):
{
while (some key is nonzero)
{
    bucket sort(keys mod 10)
    keys = keys / 10
}
}
```

The only possibly strange part: Why do we do the sort least important digit first? For that matter, why do we do more than one bucket sort, since the last one is the one that puts everything into place?

> Answer: If we're trying to sort things by hand we tend to do something different: first do a bucket sort, then recursively sort the values sharing a common first digit. This works, but is less efficient since it splits the problem up into many subproblems. By contrast, radix sorting never splits up the list; it just applies bucket sorting several times to the same list.
>
> In radix sorting, the last pass of bucket sorting is the one with the most effect on the overall order. So we want it to be the one using the most important digits. The previous bucket sorting passes are used only to take care of the case in which two items have the same key (mod 10) on the last pass.

Correctness:

> We prove that the algorithm is correct by induction. The induction hypothesis is that after i steps, the numbers are sorted by key modulo 10^i. Certainly after no steps, all numbers are the same modulo 1, and are therefore sorted by that value, so the base case is true. Inductively, step i+1 sorts by key / 10^i. If two numbers have the same value of key/10^i, the stability property of bucket sorting leaves them sorted by lower order digits; and if they don't have the same value, the bucket sort on step i+1 puts them in the right order, so in either case the induction hypothesis holds. For i sufficiently large, taking the keys mod 10^i doesn't change them, at which point the list is sorted.

Analysis:

> The algorithm takes O(n) time per bucket sort.
> There are log_10 k = O(log n) bucket sorts.
> So the total time is O(n log k).

Is this ever the best algorithm to use?

> Answer: No. If k is smaller than n, this takes O(n log k) while bucket sort takes only O(n). And if k is larger than n, the O(n log k) taken by this method is worse than the O(n log n) taken by comparison sorting.

How can we make it better?

> Answer: don't use decimal notation. Multiplication and division is expensive, so it's better to use a base that's a power of 2 (this saves a constant factor but is an easy optimization). More importantly, 10 is too small; we can increase the base up to as large as O(n) without increasing the bucket sort times significantly. If we use base n notation (or base some power

of 2 close to n) we get time bounds of the form

```
O(n (1 + (log k)/(log n)))
```

Example: sorting a million 32-bit numbers. If we use 2^16 (roughly 64000) as our base (i.e. if we use this number in place of 10 in the radix sorting pseudocode above) then the two O(k) loops are a minuscule fraction of the total time per bucket sort, and we only need two bucket sorting passes to solve the problem.

With some more complicated methods one can make integer sorting algorithms having bounds O(n loglog k) or O(n log n / log log n) but these are of less practical interest, and are only good when k is enormously greater than n.

Sorting floating point numbers

Floating point represents numbers roughly in the form x = a * 2^b. If all numbers had same value of b, we could just bucket sort by a. Different b's complicate the picture but basically b is more important than a so sort by a first (radix sort) then one more bucket sort by b.

Sorting character strings

Essentially, alphabetical order no different than a base-26 representation of numbers and sorting by ascii character value is no different than a base-256 representation. As usual, the first character is the most important and therefore the one you want to save for last -- we can just work backwards through the strings doing bucket sorts by the character values at each position. But there are some details involved in applying this idea to get a quick algorithm.

What if strings have different lengths? There doesn't always exist a character in position i. But we can test i against the string's length, and use character value 0 if the string is too short since e.g. "a" should come before "aardvark". One way to think about this is that we just "pad" the strings out with null characters to make them all the same length.

However, this padding idea can significantly increase the size of the strings (or in other words it makes a sorting algorithm based on it too slow). If you have n strings, most very short, but one of length n, the time will be O(n^2) even though the input length is O(n), so this idea of using radix sort for strings would end up no better than a bad comparison sort algorithm.

The problem is that when we sort the strings by their i'th character values, we're wasting a lot of work sorting the strings shorter than that. One possible solution would be to, in each bucket sort, only sort the strings longer than that length. How do we find this list of long strings? We can do it by another call to bucket sorting! Just sort the strings by their lengths.

There still remains a more subtle problem with this approach. Recall that bucket sort is efficient only when n<k. In early passes of the radix sort algorithm, we'll only be sorting really long strings, so there may be very few of them, and we won't have n<k, so those bucket sorts may not be efficient

Solution: We'll use the same array of k buckets for each radix sort pass, so putting things into buckets (distribute) is still fast. Concatenating nonempty buckets (coalesce) is also still fast. The only slowdown comes from finding the nonempty buckets. So we make a list of which buckets will be nonempty using bucket sorting in yet a third way! We sort pairs (i,string[i]) by two bucket sorts. The buckets of the

second bucket sort give, for each i, a sorted list of the characters in position i of the strings, which tells us which buckets will be nonempty in the radix sort part of the algorithm.

This is getting pretty complicated and confusing, but it's not too bad once we express it in pseudocode:

```
string sort(L)
{
make list of pairs (i,str[i])
bucket sort pairs by str[i]
bucket sort pairs by i (giving lists chars[i])
bucket sort strings by length
i = max length
L = empty
while (i > 0)
{
    concatenate strings of length = i before start of L
    distribute into buckets by chars in position i
    coalesce by concatenating buckets in chars[i]
    i--
}
concatenate list of empty strings at start of L
return L
}
```

The time for the first three bucket sorts is O(N+k). Each remaining pass takes time O(n_i) where n_i is the number of strings having a character in position i, so the total time for the while loop is O(N) again. Overall, the algorithm's total time is O(N + k), where N is the total length of all strings and k is number of character values.
