Non-Comparison Sorts

Binsort

Binsort is a type of sort that is not based on comparisons between key values but instead simply assigns records to "*bins*" based of the key value of the record alone.

These bins, in Abstract Data Type terms, are Queue data structures, which maintain the order that records are inserted into them.

The final step of the binsort is to concatenate the queues together in order to get a single list of records with all records of the first bin followed by all records of the second bin, etc.

Binsort is a stable sort because values that belong in the same bin are enqueued in the order that they appear in the input.

Binsort does one pass through the input to fill the bins, and one pass through the bins to create the output list, so this is $O(n)$.

Binsort Example

For example, with a shuffled deck of 52 playing cards you can do a pass through the deck separating out each card into one of 13 piles by their face value (Ace, 2, 3,. . . , Jack, Queen, King). Each pile, or *bin*, would end up with 4 cards with the same face value, but the suits (Hearts, Diamonds, Clubs, Spades) within each bin would still be mixed up. We can now put all the piles together to make a single pile of 52 cards, which are sorted by face value but not by suit.

If we now do another binsort on the pile obtained from our first binsort, but this time based on suits rather than face values, you end up with 4 piles of 13 cards, with one pile for each suit. This time, because of the stability of binsort, each pile **WILL** be sorted by face value: since the input was sorted by face value, cards are put into each suit pile in face value order.

Further Binsort Examples

Dates are suitable values to do such "*multi-phase*" binsorts on: sort first by day, then by month, then by year to obtain the list of dates in Year, Month, Day order.

A variant on binsort is *bucketsort*, where instead of "*scattering*" records into bins based just on a value (which could be numeric or categorical), they are scattered into buckets based on a range of numeric values or a set of categories.

Radix Sort

Radix sort is a multi-phase binsort where the key sorted on in each phase is a different, more significant base power of the integer key. For example, in base (or radix) 10, an integer has digits for units, 10s, 100s, 1000s, etc. In a radix sort, a binsort on the units digit is performed first, then on the 10s digit, then on the 100s digit etc. The result final result will be that the keys are sorted first by the most significant digit, then by the next most significant digit, . . . , until finally by the least significant digit. That is, they will be sorted into normal integer order.

Radix Sort Example

Here we sort a set of numbers using a 3-phase binary radix sort, i.e. the base, or radix of the sort is 2, so there are two bins used: bin 0 and bin 1:

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\\hline 101\\111\\001\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\\hline 010\\110\\111\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\001\\010\\011\\\hline 100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits in a key
- Reduce to $O\left(\frac{kn}{m}\right)$ by grouping m bits together and using 2^m bins, e.g. $m = 4$ and use 16 bins.

Pigeonhole Sort

A special case is when the keys to be sorted are the numbers from 0 to $n-1$. This sounds unnecessary, i.e. why not just generate the numbers in order from 0 to $n-1$?, but remember that these keys are typically just fields in records and the requirement is to put the records in key value order, not just the key values.

The idea here is to create an output array of size n, and iterate through the input list directly assigning the input records to their correct location in the output array. Clearly, this is $O(n)$.

```
1 pigeonhole_sort(a, n){
2   create array b of size n
3   for (i = 0 ; i < n ; i++ )
4     b[a[i]] = a[i]
5   copy array b into array a
6 }
```

Pigeonhole Sort in-place

We can avoid allocating the extra array and doing the extra copy as follows:

```
1 pigeonhole_sort_inplace(a, n){
2   for (i = 0 ; i < n ; i++ )
3     while ( a[i] != i )
4       swap a[a[i]] and a[i]
5 }
```

3	0	4	1	2
1	0	4	3	2
0	1	4	3	2
0	1	2	3	4

Every swap results in at least one key in its correct position, and once a key is in its correct position, it is never again swapped, so there are at most $n - 1$ swaps, therefore the sort is $O(n)$