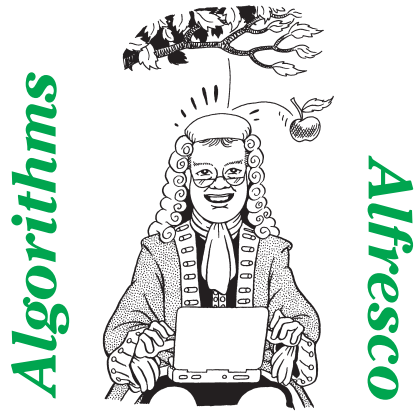
Distribution And Radix Sorts

This month we look at sorts that don't use comparisons



by Julian Bucknall

Way back in the mists of time (September 1998, to be precise) the *Algorithms Alfresco* column was on sorting, the process by which you rearrange a set of items according to some ordering rule. At the time, all I wanted to talk about was sorting keys by using comparisons of the keys. We discussed bubble sort and insertion sort, selection sort and shellsort, and quicksort, as well as a couple of others. Later on, in November 1998, we talked about heapsort, another important sorting algorithm, and finally we saw mergesort in November 2000.

It turns out that all of these sorting algorithms rely on *key comparisons* to do their work. We compare two keys (however the keys may be defined, or whatever type they might be) and, based on the comparison, we may swap them over. For bubble sort, the two keys are always neighbours, but in the more advanced sorting algorithms like quicksort, the keys may be very distant from each other (and indeed it is because we can swap keys and records over long distances that quicksort is faster than bubble sort). It can be shown mathematically that any sorting algorithm that relies on key comparisons is at best an O($n$log(n)) algorithm.

Since we'll be talking a little about performance issues in this article, let's briefly recap the big-Oh notation. All it means is that, for sufficiently large n, the performance of the algorithm in question is proportional to the expression inside the big-Oh parentheses. So an algorithm that's described as O(n) has performance that is proportional to the number of items under consideration. If we had twice as many items, it would take twice as long. An algorithm that has O($n$log(n)) performance runs proportional to n times the logarithm of n. If we measure a test with 1,000 items, say, to take 1 second, then we calculate the proportionality constant to be about 0.00015 (in other words, 0.00015 * 1000 * log(1000) = 1, using natural logs). Hence, it will take about 2.3 seconds to apply the algorithm to 2,000 items.

Does that mean that *any* sorting algorithm is limited mathematically to O($n$log(n)) performance? The answer, surprisingly, is no. Since I didn't want to particularly muddy the waters in the previous *Algorithms Alfresco* articles, I ignored this important point at the time, meaning to come back to it when I wanted to. Well, it's now time, so this month we'll look at ways to improve sorting performance for certain specialized keys.

And that, in fact, is an important point to understand before we start: we can only improve sorting performance by understanding and exploiting certain peculiarities of certain key types. If we want to generalize sorting to any key types, with unknown comparison methods, the best we can do is an optimized quicksort.

Before jumping in at the deep end, we need to set some ground rules and definitions. We shall be sorting a set of items of some type, most often an array of items. The items will all reside in memory at the same time (in other words, we won't be considering any form of external or disk-based sorting). The items we'll be looking at have a key. It is through the key that we distinguish the items from each other when we sort them. We'll assume that the key is some field of the item (in other words, the item is supposed to be a record or an object), and we'll state what type the key is when we need to. (Notice that this is different from last time, where we never stated anything in particular about the keys and items we were sorting, instead relying on a comparison routine to tell us if the key for an item was less than, greater than, or equal to another.)

Distribution Sort

The first sorting routine we'll discuss is called *distribution sort* or *counting sort*. For this we assume that the keys are integers in some small range (say, byte values from 0 to 255, or character values).

If the keys all have the same value, then there's nothing to do since the items are already sorted by definition.

Suppose now, though, that the keys have two distinct values, we'll call them 0 and 1. We can sort the items like this. Make a pass through the set of items, counting the number of zeros and the number of ones. If the number of items with key 0 were n, say, then, when sorted, they would occupy the first n slots in the sorted array. The remainder of the items, those with key 1, would appear in the second part of the array. This gives us a small hint: we shall need to copy the items from one array to another. We use two counters: the first will be for items with key 0 and is initialized to that count, the second for the items with key 1, initialized to that count plus the count of items with key 0. (In other words, the first count is the number of items whose key is less

than or equal to 0, the second count is the number of items whose key is less than or equal to 1.) We shall use these counters as indexes to tell us where to put a corresponding item when we copy it over to the auxiliary array. Now we make a second pass through the unsorted array, going from the end to the beginning. For each item with key 0, we decrement the first counter and copy the item over to the auxiliary array, using the counter's value as an index. For every item we get with key 1, we decrement the second counter, and we copy the item over to the auxiliary array using the counter's value as an index.

It is unlikely we would ever want to sort items with only two possible keys, and so it seems pointless to discuss it. However, this algorithm will extend very easily to items with byte-sized keys, or keys that are characters, or keys that are all within some 'small' numeric range.

So let us suppose that we want to sort an array of items using a byte-sized key. There are only 256 possible values for this key. The first step is to count the number of occurrences for each possible value. Simple enough, declare a 256 element array of integers, our `Counter` array, set them all to zero, and then make a pass through the array of items. For each key value encountered, increment the counter for that key. At the end of this pass, `Counter[0]` will contain the number of items with key 0, `Counter[1]`, the number of items with key 1, and so on.

We now need to set the initial index values for each key value. Make a pass through the `Counter` array, from the start to the end, setting each value equal to itself plus the value of the previous counter. After this pass, `Counter[i]` will hold the number of items whose key is less than or equal to *i*.

Now we have the cumulative distribution values, we can copy the items in the original array to the auxiliary array using the `Counter` values as indexes, in the same manner as we did in the simple two-valued key example. Stated explicitly: make a pass through the original array from end to start; for each item, get its key, decrement that particular counter, and use the new value as an index where to copy the item. We should, as a final step, move all the items back from the auxiliary array into the original array in their sorted order.

Notice that we never make a comparison of the byte keys: we just use their values as index values into one array or another according to the dictates of the algorithm. Distribution sort is a O(*n*) algorithm: every item is moved twice (once to the auxiliary array, and once back again), and every key is referenced twice (once for the counting of the distributions, and once as an index for an item move).

Listing 1 shows an implementation of a distribution sort for items that have a byte field as a key. It starts off by creating an array of unsorted items (each item having a string data field and a byte key), and then launches into distribution sort. Finally the code verifies that the sorting operation was successful.

If the keys are of a larger type, say dates using the `TDateTime` type, but restricted to a small range, say a 10-year period, we can still use distribution sort relatively easily by subtracting the start point of the range from each key before using it. The range of keys is also important: after all we shall be declaring and using an array of integers of the same size as the range. If the range were larger than the total count of the items (say the range is from zero to 1,000,000, but we only have 1,000 items to sort, the time taken in housekeeping (zeroing the counters, calculating the cumulative values) will start to have a greater and greater part to play in the overall performance of the algorithm.

➤ *Listing 1: Distribution sort on byte keys.*

```
type
  PaaItemByteKey = ^TaaItemByteKey;
  TaaItemByteKey = record
    ibkData : string;  // the data
    ibkKey  : byte;    // the key
  end;
  PaaItemByteKeyList = ^TaaItemByteKeyList;
  TaaItemByteKeyList =
    array [0..pred(ItemListCount)] of TaaItemByteKey;
procedure aaDistributionSort;
var
  i : integer;
  ItemList : PaaItemByteKeyList;
  AuxList  : PaaItemByteKeyList;
  Counter : array [0..255] of integer;
  PrevData : string;
  PrevKey  : byte;
begin
  writeln('Distribution sort on a byte key');
  {create an array of items with random keys to be sorted}
  writeln('..building array to be sorted');
  ItemList := AllocMem(sizeof(TaaItemByteKeyList));
  for i := 0 to pred(ItemListCount) do begin
    ItemList^[i].ibkData := Format('Item %5d', [i]);
    ItemList^[i].ibkKey := random(256);
  end;
  writeln('..done, now starting sort...');
  {clear the counter array}
  FillChar(Counter, sizeof(Counter), 0);
  {calculate the distribution of each key}
  for i := 0 to pred(ItemListCount) do
    inc(Counter[ItemList^[i].ibkKey]);
  {calculate the cumulative distribution}
  for i := 1 to 255 do
    inc(Counter[i], Counter[i-1]);
  {create the auxiliary list}
  New(AuxList);
  {copy over the items to the auxiliary list in sorted
    order}
  for i := pred(ItemListCount) downto 0 do begin
    dec(Counter[ItemList^[i].ibkKey]);
    AuxList^[Counter[ItemList^[i].ibkKey]] := ItemList^[i];
  end;
  writeln('..checking sort order...');
  PrevData := '';
  PrevKey := 0;
  for i := 0 to pred(ItemListCount) do begin
    if (AuxList^[i].ibkKey < PrevKey) then begin
      writeln('Error: key out of sequence'); readln;
    end
    else if (AuxList^[i].ibkKey = PrevKey) then begin
      if (AuxList^[i].ibkData <= PrevData) then begin
        writeln('Error: sort not stable'); readln;
      end;
      PrevData := AuxList^[i].ibkData;
    end
    else begin
      PrevKey := AuxList^[i].ibkKey;
      PrevData := AuxList^[i].ibkData;
    end;
  end;
  writeln('..done');
  Finalize(ItemList^);
  FreeMem(ItemList);
  Finalize(AuxList^);
  FreeMem(AuxList);
end;
```

Notice also that the way we move the items means that distribution sort is *stable*. Recall that the stability of a sort defines what happens to items with the same key during the sorting process. If items *a*, *b*, *c* have the same key and are in that order in the unsorted list (*a* appears before *b*, and *b* appears before *c*, possibly with items in between) then they will be in that same order in the sorted list, provided that the sort is stable. If the sort were not stable, they could be reorganized in any order whatsoever. The stability of a sort is not something we need to worry about generally; sometimes, however, it's very important. The code that initializes the array to sort in Listing 1 sets a data field to be a string value that's sequential and increasing. The verification routine also checks that the items with equal keys keep their data fields in sorted increasing order, in other words verifying that the sort is stable.

And String Keys?

Having seen distribution sort we can immediately see that it's not very applicable to keys that are strings. Or is it?

Let's do a thought experiment. Suppose we had a shuffled deck of cards. How would we efficiently sort them? We could try and hold all 52 of them in our hand and then apply a standard insertion sort using our other hand, but it's going to be very unwieldy. Instead, let's use distribution sort on them. The first key value we'll use is the suit of each card. Ready? Deal out the cards into four piles, one each for clubs, diamonds, hearts and spades. That's how simple distribution sort can be!

Now each pile can be distribution sorted separately (using distribution sort for this is a little bizarre, but bear with me: this is a thought experiment, remember). What have we done in this little experiment? It's as if we had a two-character key associated with each card. The first character is the suit (C, D, H, S); the second is the pip value of the card (A, 2, ..., 9, T, J, Q, K). We use distribution sort on the first character of each key to sort the cards into suits. We then use distribution sort on each of the subsets we created (the suits) using the second character of the key. This is almost like quicksort: to sort the whole list, you partition the list and then apply the same quicksort algorithm to each of the parts.

Let's generalize a little. Suppose the keys are strings of lowercase letters (that is, there are 26 possible values for each character). We apply distribution sort to the first character of the keys. What this will do, in effect, is to partition the items into *bins*, each bin containing items whose keys start with a particular letter. We then apply the distribution sort to each individual bin, using the second letter of the item's key as the distribution sort key. This will partition each bin into at most 26 other sub-bins. We then recursively apply distribution sort to each of these sub-bins, using the third letter of each key. We don't recurse, of course: should there be one item or no items in each sub-sub-sub-bin.

MSD Radix Sort

This is known as MSD radix sorting, MSD standing for *most significant digit*. What we are doing, in essence, is treating the string as a large number, with each character being a digit in that number. The number of possible values for each digit is the radix, or base. So, in our example, sorting keys containing lowercase letters, the radix is 26. We sort from the most significant digit down to the least significant (from start to end of the string).

Listing 2 shows the MSD radix sort using all 256 values for each character, rather than just 26. Notice that each invocation of the MSD routine will allocate a local variable to hold the counter array, and the routine is recursive. As we want to cater for the possibility that all character values would be used, the counter array size is 1Kb in size (that is, 256 * `sizeof(integer)`), and the effect of the recursion on the stack could become a problem, especially if the strings were very long. In fact, so that we can identify the bins when making the recursive call, we need a copy of the `Counter` array (here called the `Bins` array) and so each recursive call takes at least 2Kb of stack. That means that, for a set of 40-character keys, we'll be using at least 80Kb of stack. Not for the faint-hearted.

Having seen MSD radix sorting, I'm sure you can see that it could be very inefficient. We're continually copying items to the auxiliary array and back again. Even worse, as we descend into the recursion, the number of items in a sub-sub-bin is mostly zero, so we're doing less and less work as we go through the sub-sub-bins, although there are more and more of them.

Think about this for a moment: the first sorting pass is based on the initial letter, and we'll find that only a few bins are empty, providing that the keys are mostly random in nature. The second pass on each of the 26 bins from the first pass will produce more bins that are empty, and the third pass even more. If the keys are not random in nature, but are English words or zip codes or phone numbers, the number of empty bins will be very many, and they'll overshadow the bins that have at least one item in them.

Also, keys in real life tend not to be nice and random, but repetitive and similar. Think of sorting the names in a telephone directory: there's an awful lot of Smiths in there, and that would mean that all the Smith records would need at least five recursions before we found a difference and were able to start partitioning out the records.

The lots of empty bins problem is hard to solve in the general case. We have to try and use information about our keys in order to guide the separate recursions. For example, what is the minimum character, the maximum? We can calculate this as we're reading through the keys calculating the counts, and this knowledge will save us some time by avoiding lots of bins with zero counts.

If you have reviewed the *Algorithms Alfresco* article from

September 1998 (or you have read chapter 5 of my book *Tomes of Delphi: Algorithms and Data Structures [Plug, what plug? Ed]*) you will know that I discussed optimizing quicksort by using insertion sort once the partitions got small enough (about 15 items in a partition). Insertion sort is a good sorting candidate on a set of items that is in roughly sorted order because its performance characteristic in that situation is O(n). We can use the same trick here. MSD radix sort, after the first pass, has sorted the set of items as far as the first character. If we do a further MSD radix pass on the second character, the items will be very close to their final position. (Unless of course all the items have keys that are the same as far as the first two characters go. This isn't too far from reality: the example of which I'm thinking here is that of sorting the names of the source files for one of TurboPower's products. We name source files so that the first two letters of the name reference the product, for example 'FF' for FlashFiler, 'LB' for LockBox, and so on. After two recursions of MSB radix sort on this little lot, we'll have done absolutely nothing.) We could then apply an insertion sort to finish off the sorting, it being a linear algorithm in that case.

LSD Radix Sort

Up until now, we've been discussing performing a radix sort on string keys as a set of distribution sorts going from first to last character. It's not well known (but, paradoxically, in a time when computers used card decks, it *was* well known since it was the method of choice for sorting punched cards) that you can sort string keys going from the last character to the first.

This is hard to visualize, to say the least: sorting by starting at the end of the string keys and moving to the start with each pass. Surely the subsequent passes would mess up the previous ones? Ordinarily, yes, but not with a stable sort, and therein lies the trick.

Let's imagine we wanted to sort the five three-letter words: *cat*, *pet*, *mat*, *pen*, *one*. We perform a standard distribution sort on the last character. This gives us: *one*, *pen*, *cat*, *pet*, *mat*. (Notice that, since distribution sort is stable, the words *cat*, *pet*, and *mat* did not change relative position even through they have the same key character in this pass.) We now perform a distribution sort on the middle character (the second from the end). The letters concerned are: *n*, *e*, *a*, *e*, and *a*, and, seeing them like that it's easy to do the sort. We get: *cat*, *mat*, *pen*, *pet*, *one*. Finally we do a distribution sort on the first character to get: *cat*, *mat*, *one*, *pen*, *pet*.

With this easy example you can see that the stability of the

➤ *Listing 2: MSD radix sort on string keys.*

```
type
  PaaItemStrKey = ^TaaItemStrKey;
  TaaItemStrKey = record
    ibkData : string;     // the data
    ibkKey  : string[9];  // the key
  end;
  PaaItemStrKeyList = ^TaaItemStrKeyList;
  TaaItemStrKeyList =
    array [0..pred(ItemListCount)] of TaaItemStrKey;
function GetRandomString : string;
var
  i : integer;
begin
  SetLength(Result, random(5) + 5);
  for i := 1 to length(Result) do
    Result[i] := char(random(26) + ord('a'));
end;
procedure MSD(aFromList, aToList : PaaItemStrKeyList;
  aFirst, aLast : integer; aCharInx : integer);
var
  i : integer;
  Inx : integer;
  Counter : array [0..255] of integer;
  Bins    : array [-1..255] of integer;
begin
  {exit if we reached the maximum character position}
  if (aCharInx > 9) then
    Exit;
  {if only one item, just exit: there's nothing to do}
  if (aLast = aFirst) then
    Exit;
  {clear the counter array}
  FillChar(Counter, sizeof(Counter), 0);
  {calculate the distribution of each key}
  for i := aFirst to aLast do
    if (length(aFromList^[i].ibkKey) < aCharInx) then
      inc(Counter[0])
    else
      inc(Counter[byte(aFromList^[i].ibkKey[aCharInx])]);
  {calculate the cumulative distribution}
  Bins[-1] := 0;
  Bins[0] := Counter[0];
  for i := 1 to 255 do begin
    inc(Counter[i], Counter[i-1]);
    Bins[i] := Counter[i];
  end;
  {copy over the items to the "to" list in sorted order}
  for i := aLast downto aFirst do begin
    if (length(aFromList^[i].ibkKey) < aCharInx) then begin
      dec(Counter[0]);
      aToList^[aFirst + Counter[0]] := aFromList^[i];
    end
    else begin
      Inx := byte(aFromList^[i].ibkKey[aCharInx]);
      dec(Counter[Inx]);
      aToList^[aFirst + Counter[Inx]] := aFromList^[i];
    end;
  end;
  {move the sorted data back}
  Move(aToList^[aFirst], aFromList^[aFirst],
       succ(aLast - aFirst) * sizeof(TaaItemStrKey));
  {recursively sort each of the bins}
  for i := 0 to 255 do begin
    if (Bins[i] > Bins[i-1]) then
      MSD(aFromList, aToList, aFirst + Bins[i-1],
        aFirst + pred(Bins[i]), succ(aCharInx));
  end;
end;
procedure aaMSDRadixSortStr;
var
  i : integer;
  ItemList : PaaItemStrKeyList;
  AuxList  : PaaItemStrKeyList;
  PrevKey  : string;
begin
  writeln('MSD radix sort on a string key');
  {create an array of items with random keys to be sorted}
  writeln('..building array to be sorted');
  ItemList := Allocmem(sizeof(TaaItemStrKeyList));
  for i := 0 to pred(ItemListCount) do begin
    ItemList^[i].ibkData := Format('Item %5d', [i]);
    ItemList^[i].ibkKey := GetRandomString;
  end;
  writeln('..done, now starting sort...');
  {allocate the auxiliary array}
  New(AuxList);
  {sort the items}
  MSD(ItemList, AuxList, 0, pred(ItemListCount), 1);
  writeln('..checking sort order...');
  PrevKey := '';
  for i := 0 to pred(ItemListCount) do begin
    if (ItemList^[i].ibkKey < PrevKey) then begin
      writeln('Error: key out of sequence');
      readln;
    end
    else begin
      PrevKey := ItemList^[i].ibkKey;
    end;
  end;
  writeln('..done');
  Finalize(ItemList^);
  FreeMem(ItemList);
  Finalize(AuxList^);
  FreeMem(AuxList);
end;
```

```
procedure aaLSDRadixSortStr;
var
  i : integer;
  Inx      : integer;
  CharInx  : integer;
  ItemList : PaaItemStrKeyList;
  AuxList  : PaaItemStrKeyList;
  FromList : PaaItemStrKeyList;
  ToList   : PaaItemStrKeyList;
  Temp     : PaaItemStrKeyList;
  Counter  : array [0..255] of integer;
  PrevKey  : string;
begin
  writeln('LSD radix sort on a string key');
  {create an array of items with random keys to be sorted}
  ...
  writeln('..done, now starting sort...');
  {allocate the auxiliary array}
  New(AuxList);
  {prepare for the loop}
  FromList := ItemList;
  ToList := AuxList;
  {for each character in the key strings, from end to
    start...}
  for CharInx := 9 downto 1 do begin
    {clear the counter array}
    FillChar(Counter, sizeof(Counter), 0);
    {calculate the distribution of each key}
    for i := 0 to pred(ItemListCount) do
      if (length(FromList^[i].ibkKey) < CharInx) then
        inc(Counter[0])
      else
        inc(Counter[byte(FromList^[i].ibkKey[CharInx])]);
    {calculate the cumulative distribution}
    for i := 1 to 255 do
      inc(Counter[i], Counter[i-1]);
    {copy over the items to the "to" list in sorted order}
    for i := pred(ItemListCount) downto 0 do begin
      if (length(FromList^[i].ibkKey) < CharInx) then begin
        dec(Counter[0]);
        ToList^[Counter[0]] := FromList^[i];
      end
      else begin
        Inx := byte(FromList^[i].ibkKey[CharInx]);
        dec(Counter[Inx]);
        ToList^[Counter[Inx]] := FromList^[i];
      end;
    end;
    {switch over the to and from.lists}
    Temp := FromList;
    FromList := ToList;
    ToList := Temp;
  end;
  if (FromList <> ItemList) then
    Move(FromList^, ItemList^, sizeof(FromList^));
  ...
end;
```

➤ *Listing 3: LSD radix sort on string keys.*

distribution sort ensures that *pen* and *pet*, once sorted according to their last letter, stay sorted that way through the other passes where they have the same key characters, no matter how many other words end up between them, until they end up next to each other in the final pass. This sort is known as *LSD radix sort* (where LSD stands for *least significant digit*).

There is a small fly in the ointment, however. When we want to sort string keys, we generally do not have strings that have the same length (which this algorithm seems to require). For null-terminated strings it's a pain in the neck, agreed. However, for Pascal strings (be they long or short), we have the length value to help us. If we're sorting on a character index beyond the length of the string, we assume that the key's relevant character is the null character. You can see that knowing the length of the string helps immeasurably here. For null-terminated strings of unequal length, it's more long-winded: we shall have to calculate the length of each string key beforehand.

Listing 3 shows the LSD radix sort sorting string keys. It uses an auxiliary array (remember that distribution sort requires this extra array since it must copy items over as part of the algorithm), but it uses it and the original array in a to-and-fro pattern to minimize the number of times we copy the items (recall that standard distribution sort requires each item to be copied twice: once from the original array to the auxiliary array and once back again). There may have to be a final copy at the end of the LSD radix sort should the length of the longest key be odd.

This sort is pretty fast, to say the least. I did a comparison between it and the fastest quicksort I had (the one I'd developed for my book) to sort 100,000 9-character strings, and the radix sort came out tops. But only just. I'll talk about this is a moment.

And Integer Keys?

So is LSD radix sort just for strings? It certainly seems that way from my description and introduction, but in reality it isn't. Let's suppose you wanted to sort a set of items whose key was a 32-bit unsigned value (a `dword` or `longword`). Could we use LSD radix sort for this? Well, one way would be to convert all the `dword` values to strings and then sort the strings, but that seems to be a sledgehammer approach. A better way is to use our understanding of how the `dword` values are held in memory. A `dword` value is 4 bytes long, and can be viewed as a 4-digit number with each digit being a value base 256. The digits are held in reverse order in memory. For example, the `dword` value `$12345678` is held as four digits, `$12`, `$34`, `$56`, `$78`, in reverse order (that is, the least significant digit `$78` is at the lowest address, then `$56`, then `$34`, and finally `$12` is at the highest address). Ordinarily we needn't worry about this, except when using a hex viewer, but we can make use of this for the LSD radix sort. View the `dword` value as being a string of four characters in reverse order, and do the normal LSD radix sort, except that we go from first digit to last digit, instead of the other way round. Listing 4 shows this specialized radix sort.

What about `longint` values, where the value could be negative? This is harder. To see why, take an example, the values `-1` and `0`. `-1` is held as `$FFFFFFFF`, whereas 0 is `$00000000`. If we do a standard LSD radix sort in the manner described above for `longwords`, the `-1` value will be sorted after the `0` value. What can we do? Well, we must alter the key value so that the problem with the negative values is removed or hidden, and then proceed as before. This trick used to be somewhat well-known prior to Delphi 5 when we didn't have a proper `longword` value and had to use a `longint` instead.

To make the trick clear, suppose instead we had `smallint` values from `-128` to `127` but we can only sort them as unsigned bytes. In hex, these values look like `$80` to `$FF` for `-128` to `-1`, `$00` for `0`, and then `$01` to `$7F` for `1` to `127`. Looking at the hex values you can see that although they seem to be pretty

much in order, they're actually separated in two parts, with both parts being ordered, but with the second half not being in order with the first half. If you ponder for a while you can see that for the negative numbers we need to clear the sign bit (the most significant bit), and for all the others we need to set it. We would then have the representation `$00` for `-128`, `$01` for `-127`, to `$7F` for `-1`. `0` would be `$80`, and `1` to `127` would be `$81` to `$FF`. So the sign bit needs to be toggled, but the other bits need to be left alone. The `XOR` operation is the toggling operation (since `1 XOR 1` is `0`, and `0 XOR 1` is `1`, we need to `XOR` the sign bit with `1`). It can also be used as a 'leave it alone' operation (since `0 XOR 0` is `0` and `1 XOR 0` is `1`, we need to `XOR` all the remaining bits with `0`). Hence the simple expression (`SmallintValue XOR $80`) will give us a byte value that can be sorted in the correct order, such that the original `smallint` values are correctly sorted.

Back to the `longint` value case: I'm sure you can now see that `XOR`ing a `longint` value with `$80000000` will give us a `dword` value that can be sorted using LSD radix sort. In fact, thinking about it, it is only to the most significant digit we need apply this trick, all the other lesser significant digits remain unchanged. Listing 5 shows the small change you need to make in order to sort `longint` values using LSD radix sort.

Performance

Now we've looked at the various sorts I wanted to discuss in this article, let's talk a little about their performance aspects. Throughout I mentioned the 'official' big-Oh expressions for the sorts based on distribution sort: basically they're all O(*n*). However, that is not the complete picture. Take LSD radix sort ordering string keys as an example. Let's say, for a large set of items, it takes one second to perform one of the distribution sort cycles on one of the character positions. Then, if the keys are all ten characters long, it will take (roughly) 10 seconds to fully sort the keys. (I say 'roughly' because there is some overhead involved with the cycling through the characters in the keys.) If the keys are all basically random in nature then quicksort will tend to be just as fast, if not faster. Why? Because the string key comparisons in quicksort will not use all the characters in the key. Mostly only one, two or three characters of two string keys will need to be compared in order to ascertain which is the larger. Quicksort can take advantage of this quite easily, whereas poor old LSD radix sort *has* to use all the characters in the keys.

In this kind of situation, we can make a similar optimization with LSD radix sort as we did with MSD radix sort. We can sort based on the first two or four (or some low even number) characters of each key. In other words, we start at the fourth, and cycle back to the first. What does this gain us? Well, after this process the keys are pretty well ordered, with only a few out of sequence. Again, we switch to insertion sort to finish off since it's a linear algorithm in these circumstances. (By the way, the reason we choose an even number is to make sure that all the items are in the original array at the end of the partial LSD radix sort, and insertion sort is an in-place sort.)

I hasten to add, though, that this analysis is based on the keys being random in nature. Since Quicksort exploits the randomness by having fast comparisons, all we're doing is trying to exploit the same effect: that nearly all keys are distinguishable by the first few characters. If our keys followed some other pattern, say surnames starting with 'Smi', we'd have to alter our optimization accordingly.

I think that's one of the big lessons to be learned here. Although distribution sort is very fast and the performance is linear with

➤ *Listing 4: LSD radix sort on longword keys.*

```
type
  DWordAsBytes = array [0..3] of byte;
type
  PaaItemU32Key = ^TaaItemU32Key;
  TaaItemU32Key = record
    ibkData : string;    // the data
    ibkKey  : longword;  // the key
  end;
  PaaItemU32KeyList = ^TaaItemU32KeyList;
  TaaItemU32KeyList =
    array [0..pred(ItemListCount)] of TaaItemU32Key;
function GetRandomU32 : longword;
var
  i : integer;
begin
  for i := 0 to 3 do
    DWordAsBytes(Result)[i] := random(256);
end;
procedure aaLSDRadixSortU32;
var
  i : integer;
  Inx      : integer;
  CharInx  : integer;
  ItemList : PaaItemU32KeyList;
  AuxList  : PaaItemU32KeyList;
  FromList : PaaItemU32KeyList;
  ToList   : PaaItemU32KeyList;
  Temp     : PaaItemU32KeyList;
  Counter  : array [0..255] of integer;
  PrevKey  : longword;
  StartTime : dword;
  EndTime   : dword;
begin
  writeln('LSD radix sort on an unsigned 32-bit key');
  {create an array of items with random keys to be sorted}
  ...
  writeln('..done, now starting sort...');
  {allocate the auxiliary array}
  New(AuxList);
  {prepare for the loop}
  FromList := ItemList;
  ToList := AuxList;
  {for each digit in the key longwords, from start to
   end...}
  for CharInx := 0 to 3 do begin
    {clear the counter array}
    FillChar(Counter, sizeof(Counter), 0);
    {calculate the distribution of each key}
    for i := 0 to pred(ItemListCount) do
      inc(Counter[
        DWordAsBytes(FromList^[i].ibkKey)[CharInx]]);
    {calculate the cumulative distribution}
    for i := 1 to 255 do
      inc(Counter[i], Counter[i-1]);
    {copy over the items to the "to" list in sorted order}
    for i := pred(ItemListCount) downto 0 do begin
      Inx := DWordAsBytes(FromList^[i].ibkKey)[CharInx];
      dec(Counter[Inx]);
      ToList^[Counter[Inx]] := FromList^[i];
    end;
    {switch over the to and from lists}
    Temp := FromList;
    FromList := ToList;
    ToList := Temp;
  end;
  ...
end;
```

respect to the number of items, the optimizations we can investigate and implement will ultimately depend on the keys we're trying to sort. We have to tailor and tune the radix sorts to suit our data; we can't get optimal speed through a generic process.

Another important point I've glossed over, but is worth bearing in mind, is this one: the distribution sort and the radix sorts all require an auxiliary array. Items are copied over from one array to the other and back again during the progress of the sort. Certainly we can be very precise about the number of times an item gets copied (twice for distribution sort, n or $n+1$ times for a radix sort on a string of n characters) but the arrays are in different parts of memory. For large arrays, we are more liable to be plagued with page swaps and page faults; thrashing, in other words. This will be more noticeable with the sorts we're discussing here rather than the in-place sorts we looked at before: there are two arrays instead of one.

➤ *Listing 5: LSD radix sort on longint keys.*

```
...
{calculate the distribution of each key}
if (CharInx = 3) then
  for i := 0 to pred(ItemListCount) do begin
    Inx := (DWordAsBytes(FromList^[i].ibkKey)[CharInx]) xor $80;
    inc(Counter[Inx]);
  end
else
  for i := 0 to pred(ItemListCount) do begin
    Inx := DWordAsBytes(FromList^[i].ibkKey)[CharInx];
    inc(Counter[Inx]);
  end;
...
{copy over the items to the "to" list in sorted order}
if (CharInx = 3) then
  for i := pred(ItemListCount) downto 0 do begin
    Inx := (DWordAsBytes(FromList^[i].ibkKey)[CharInx]) xor $80;
    dec(Counter[Inx]);
    ToList^[Counter[Inx]] := FromList^[i];
  end
else
  for i := pred(ItemListCount) downto 0 do begin
    Inx := DWordAsBytes(FromList^[i].ibkKey)[CharInx];
    dec(Counter[Inx]);
    ToList^[Counter[Inx]] := FromList^[i];
  end;
...
```

Summary

The distribution and radix sorts discussed here are very important when you have a highly specific sorting application that fits their talents. The next time you have a large amount of sorting to do, experiment with one of the radix sorts and try optimizing with insertion sort, instead of just plumping for standard quicksort.

Julian Bucknall works for TurboPower Software. You can reach him at julianb@turbopower.com