Bucket Sort

The idea behind bucket sort is that if we know the range of our elements to be sorted, we can set up buckets for each possible element, and just toss elements into their corresponding buckets. We then empty the buckets in order, and the result is a sorted list.

In implementing this algorithm, we can easily use an array to represent our buckets, where the value at each array index will represent the number of elements in the corresponding bucket. If we have integers on the range `[0..max]`, then we set up an array of `(max + 1)` integers and initialize all the values to zero. We then proceed sequentially through the unsorted array, reading the value of each element, going to the corresponding index in the buckets array, and incrementing the value there.

Then our function, in C, is as follows:

```
void bucketsort(int array[], int n, int max) {

   int i, j = 0;

   /* Declare an array of size (max + 1) and initialize all values to zero. */
   int *bucket = calloc(max + 1, sizeof(int));

   /* Place each element from the unsorted array into its corresponding bucket. */
   for (i = 0; i < n; i++)
       bucket[array[i]]++;

   /* Sequentially empty each bucket back into the original array. */
   for (i = 0; i < max; i++)
       while(bucket[i]--)
           array[j++] = i;
}
```

We immediately see two drawbacks to this sorting algorithm. Firstly, we must know the maximum value of any element that can be found in the unsorted array. Without this information, we do know how many buckets to initialize.

Secondly, we must be able to create enough buckets in memory for every element that we might possibly encounter in the unsorted array. It may be impossible to declare an array large enough to fulfill this requirement on some systems. Accordingly, we should augment our function above to determine whether the array was properly allocated at runtime. This can be achieved by inserting the following line after the `calloc()` statement:

```
   if (bucket == NULL) return;
```

Thus, we see a tradeoff between time and space complexity here. The O(n) bucket sort out-performs any comparison based sort (in terms of its Big-Oh notation, at least), but places a much higher demand on system memory than any of the other sorting algorithms we have seen so far.

The reader should be aware that there are other variations of bucket sort that rely on buckets that accommodate small ranges of elements, rather than single values. A discussion of that sort of implementation is available on Wikipedia (http://en.wikipedia.org/wiki/Bucket_sort). The notes presented here are loosely based on information from that page.

Radix Sort

The idea behind radix sort is slightly more complex than that of bucket sort, although not terribly so. The algorithm proceeds as follows:

1. Take the least significant digit of each element in the unsorted array.
2. Perform a *stable sort* (see below) based on that key.
3. Repeat the process sequentially with each more significant digit.

By a *stable sort*, we mean that the original order of the elements is maintained in the event of ties.

For example, supposed we want to sort the list:

849, 770, 67, 347, 201, 618, 66, 495, 13, 45

Sorting by the least significant digit (i.e., the ones digit), we get:

770, 201, 13, 495, 45, 66, 67, 347, 618, 849

Notice that although 495 and 45 (as well as 67 and 347) are equal in terms of their least significant digit, their order in the original list is preserved (i.e., 495 came before 45 in the original list, and so it comes before 45 after sorting by the least significant digit; so too for 67 and 347). This is what we mean when we say that the sorting algorithm must be stable.

We then proceed to sort by the more significant tens digit, resulting in the order:

201, 13, 618, 45, 347, 849, 66, 67, 770, 495

Notice that we have once again employed a stable sort. 347 and 849 are clustered together because they share the same tens digit, but the former precedes the latter because it does so in the previous list.

In our last pass, we sort by the most significant digit, the hundreds digit. In the case of 13, 45, 66, and 67, we consider the hundreds digit to be zero. The result is the following sorted list:

13, 45, 66, 67, 201, 347, 495, 618, 770, 849

So far, we have ignored the precise sorting mechanism employed at step two in each iteration of the algorithm. That's because radix sort gives us the freedom to choose whichever sorting algorithm we please, so long as it is a stable sorting algorithm. The runtime analysis will, of course, depend on the runtime of the sorting algorithm employed at step two of each iteration.

It is quite common, and therefore worth mentioning, to use bucket sort for our stable sorting algorithm in radix sort. Because we are only sorting by one digit at a time, we only have to set up ten buckets, [0..9], to accomplish this goal. However, the implementation of those buckets will necessarily vary from the one given in the previous section of these notes, because each bucket must be able to hold multiple values, which will not be equal to the index for that bucket. One common solution, then, is to implement the buckets as queues.

When using bucket sort in conjunction with radix sort, if we have n elements to sort and the maximum length of any of those elements is k, then we must perform k iterations of the O(n) bucket sort, giving us an overall Big-Oh of O(nk) for this particular implementation of radix sort. For instance, in the example above, we have $k = 3$ since the largest element has three digits, and we end up doing three iterations of the bucket sort. k is not a constant, though. It will vary depending on the input, though, and so it must be included in the Big-Oh analysis of the algorithm. Further discussion on implementation issues, as well as more examples of the radix sort in action, are available on Wikipedia (http://en.wikipedia.org/wiki/Radix_sort). The notes presented here are loosely based on information from that page.