**(LSD) Radix Sort for Integers**

<http://www.geekviewpoint.com/java/sorting/bucketsort>

<http://www.geekviewpoint.com/java/sorting/countingsort>

<http://www.geekviewpoint.com/java/sorting/radixsort>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Author: Isai Damier

 \* Title: Radix Sort

 \* Project: geekviewpoint

 \* Package: algorithms

 \*

 \* Statement:

 \*   Given a disordered list of integers, rearrange them in natural order.

 \*

 \* Sample Input: {18,5,100,3,1,19,6,0,7,4,2}

 \*

 \* Sample Output: {0,1,2,3,4,5,6,7,18,19,100}

 \*

 \* Time Complexity of Solution:

 \*   Best Case O(k\*n); Average Case O(k\*n); Worst Case O(k\*n),

 \*   where k is the length of the longest number and n is the

 \*   size of the input array.

 \*

 \*   Note: if k is greater than log(n) then an n\*log(n) algorithm would be a

 \*         better fit. In reality we can always change the radix to make k

 \*         less than log(n).

 \*

 \* Approach:

 \*   radix sort, like counting sort and bucket sort, is an integer based

 \*   algorithm (i.e. the values of the input array are assumed to be

 \*   integers). Hence radix sort is among the fastest sorting algorithms

 \*   around, in theory. The particular distinction for radix sort is that it

 \*   creates a bucket for each cipher (i.e. digit); as such, similar to

 \*   bucket sort, each bucket in radix sort must be a growable list that may

 \*   admit different keys.

 \*

 \*   For decimal values, the number of buckets is 10, as the decimal system

 \*   has 10 numerals/cyphers (i.e. 0,1,2,3,4,5,6,7,8,9). Then the keys are

 \*   continuously sorted by significant digits.

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

 public void radixsort(int[] input) {

  final int RADIX = 10;

  // declare and initialize bucket[]

  List<Integer>[] bucket = new ArrayList[RADIX];

  for (int i = 0; i < bucket.length; i++) {

    bucket[i] = new ArrayList<Integer>();

  }

  // sort

  boolean maxLength = false;

  int tmp = -1, placement = 1;

  while (!maxLength) {

    maxLength = true;

    // split input between lists

    for (Integer i : input) {

      tmp = i / placement;

      bucket[tmp % RADIX].add(i);

      if (maxLength && tmp > 0) {

        maxLength = false;

      }

    }

    // empty lists into input array

    int a = 0;

    for (int b = 0; b < RADIX; b++) {

      for (Integer i : bucket[b]) {

        input[a++] = i;

      }

      bucket[b].clear();

    }

    // move to next digit

    placement \*= RADIX;

  }

}

**(LSD) Radix Sort for String**

<http://users.cis.fiu.edu/~weiss/dsaajava3/code/RadixSort.java>

import java.util.ArrayList;

import java.util.List;

import java.util.Arrays;

import java.util.Random;

public class RadixSort

{

/\*

\* Radix sort an array of Strings

\* Assume all are all ASCII

\* Assume all have same length

\*/

public static void radixSortA( String [ ] arr, int stringLen )

{

final int BUCKETS = 256;

ArrayList<String> [ ] buckets = new ArrayList<>[ BUCKETS ];

for( int i = 0; i < BUCKETS; i++ )

buckets[ i ] = new ArrayList<>( );

for( int pos = stringLen - 1; pos >= 0; pos-- )

{

for( String s : arr )

buckets[ s.charAt( pos ) ].add( s );

int idx = 0;

for( ArrayList<String> thisBucket : buckets )

{

for( String s : thisBucket )

arr[ idx++ ] = s;

thisBucket.clear( );

}

}

}

/\*

\* Counting radix sort an array of Strings

\* Assume all are all ASCII

\* Assume all have same length

\*/

public static void countingRadixSort( String [ ] arr, int stringLen )

{

final int BUCKETS = 256;

int N = arr.length;

String [ ] buffer = new String[ N ];

String [ ] in = arr;

String [ ] out = buffer;

for( int pos = stringLen - 1; pos >= 0; pos-- )

{

int[ ] count = new int [ BUCKETS + 1 ];

for( int i = 0; i < N; i++ )

count[ in[ i ].charAt( pos ) + 1 ]++;

for( int b = 1; b <= BUCKETS; b++ )

count[ b ] += count[ b - 1 ];

for( int i = 0; i < N; i++ )

out[ count[ in[ i ].charAt( pos ) ]++ ] = in[ i ];

// swap in and out roles

String [ ] tmp = in;

in = out;

out = tmp;

}

// if odd number of passes, in is buffer, out is arr; so copy back

if( stringLen % 2 == 1 )

for( int i = 0; i < arr.length; i++ )

out[ i ] = in[ i ];

}

/\*

\* Radix sort an array of Strings

\* Assume all are all ASCII

\* Assume all have length bounded by maxLen

\*/

public static void radixSort( String [ ] arr, int maxLen )

{

final int BUCKETS = 256;

ArrayList<String> [ ] wordsByLength = new ArrayList<>[ maxLen + 1 ];

ArrayList<String> [ ] buckets = new ArrayList<>[ BUCKETS ];

for( int i = 0; i < wordsByLength.length; i++ )

wordsByLength[ i ] = new ArrayList<>( );

for( int i = 0; i < BUCKETS; i++ )

buckets[ i ] = new ArrayList<>( );

for( String s : arr )

wordsByLength[ s.length( ) ].add( s );

int idx = 0;

for( ArrayList<String> wordList : wordsByLength )

for( String s : wordList )

arr[ idx++ ] = s;

int startingIndex = arr.length;

for( int pos = maxLen - 1; pos >= 0; pos-- )

{

startingIndex -= wordsByLength[ pos + 1 ].size( );

for( int i = startingIndex; i < arr.length; i++ )

buckets[ arr[ i ].charAt( pos ) ].add( arr[ i ] );

idx = startingIndex;

for( ArrayList<String> thisBucket : buckets )

{

for( String s : thisBucket )

arr[ idx++ ] = s;

thisBucket.clear( );

}

}

}

public static void main( String [ ] args )

{

List<String> lst = new ArrayList<>( );

Random r = new Random( );

final int LEN = 5;

for( int i = 0; i < 100000; i++ )

{

String str = "";

int len = LEN; // 3 + r.nextInt( 7 ); // between 3 and 9 characters

for( int j = 0; j < len; j++ )

str += (char) ( 'a' + r.nextInt( 26 ) );

lst.add( str );

}

String [ ] arr1 = new String[ lst.size( ) ];

String [ ] arr2 = new String[ lst.size( ) ];

lst.toArray( arr1 );

lst.toArray( arr2 );

long start, end;

start = System.currentTimeMillis( );

Arrays.sort( arr1 );

end = System.currentTimeMillis( );

System.out.println( "Elapsed: " + ( end - start ) );

start = System.currentTimeMillis( );

countingRadixSort( arr2, LEN );

end = System.currentTimeMillis( );

System.out.println( "Elapsed: " + ( end - start ) );

for( int i = 0; i < arr1.length; i++ )

if( !arr1[ i ].equals( arr2[ i ] ) )

System.out.println( "OOPS!!" );

}

}

Bucket Sort

<http://www.geekviewpoint.com/java/sorting/bucketsort>

<http://en.wikipedia.org/wiki/Bucket_sort>

**Bucket sort**, or **bin sort**, is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) that works by partitioning an [array](http://en.wikipedia.org/wiki/Array_data_structure) into a number of [buckets](http://en.wikipedia.org/wiki/Bucket_(computing)). Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a [distribution sort](http://en.wikipedia.org/wiki/Distribution_sort), and is a cousin of [radix sort](http://en.wikipedia.org/wiki/Radix_sort) in the most to least significant digit flavour.

Pseudocode

**function** bucketSort(array, n) **is**

buckets ← new array of n empty lists

**for** i = 0 **to** (length(array)-1) **do**

insert *array[i]* into buckets[msbits(array[i], k)]

**for** i = 0 **to** n - 1 **do**

nextSort(buckets[i])

**return** the concatenation of buckets[0], ...., buckets[n-1]

The function *nextSort* is a sorting function; using *bucketSort* itself as *nextSort* produces a relative of [radix sort](http://en.wikipedia.org/wiki/Radix_sort); in particular, the case *n = 2* corresponds to [quicksort](http://en.wikipedia.org/wiki/Quicksort) (although potentially with poor pivot choices).

Optimizations

A common optimization is to put the elements back in the original array *first*, then run [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) over the complete array; because [**insertion sort's**](http://en.wikipedia.org/wiki/Insertion_sort) runtime is based on how far each element is from its final position, the number of comparisons remains relatively small, and the memory hierarchy is better exploited by storing the list contiguously in memory.[[1]](http://en.wikipedia.org/wiki/Bucket_sort#cite_note-1)

## Comparison with other sorting algorithms

Bucket sort can be seen as a generalization of [counting sort](http://en.wikipedia.org/wiki/Counting_sort); in fact, if each bucket has size 1 then bucket sort degenerates to counting sort. The variable bucket size of bucket sort allows it to use O(*n*) memory instead of O(*M*) memory, where *M* is the number of distinct values; in exchange, it gives up counting sort's O(*n* + *M*) worst-case behavior.

Bucket sort with two buckets is effectively a version of [quicksort](http://en.wikipedia.org/wiki/Quicksort) where the pivot value is always selected to be the middle value of the value range. While this choice is effective for uniformly distributed inputs, other means of choosing the pivot in quicksort such as randomly selected pivots make it more resistant to clustering in the input distribution.

The *n*-way [mergesort](http://en.wikipedia.org/wiki/Mergesort" \o "Mergesort) algorithm also begins by distributing the list into *n* sublists and sorting each one; however, the sublists created by mergesort have overlapping value ranges and so cannot be recombined by simple concatenation as in bucket sort. Instead, they must be interleaved by a merge algorithm. However, this added expense is counterbalanced by the simpler scatter phase and the ability to ensure that each sublist is the same size, providing a good worst-case time bound.

Top-down [radix sort](http://en.wikipedia.org/wiki/Radix_sort) can be seen as a special case of bucket sort where both the range of values and the number of buckets is constrained to be a power of two. Consequently, each bucket's size is also a power of two, and the procedure can be applied recursively. This approach can accelerate the scatter phase, since we only need to examine a prefix of the bit representation of each element to determine its bucket.

Radix Sort

<http://en.wikipedia.org/wiki/Radix_sort>

## Least significant digit radix sorts

A [least significant digit](http://en.wikipedia.org/wiki/Least_significant_digit) (LSD) radix sort is a fast [stable](http://en.wikipedia.org/wiki/Stable_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) which can be used to sort keys in integer representation order. Keys may be a [string](http://en.wikipedia.org/wiki/String_(computer_science)) of characters, or numerical digits in a given 'radix'. The processing of the keys begins at the [least significant digit](http://en.wikipedia.org/wiki/Least_significant_digit) (i.e., the rightmost digit), and proceeds to the [most significant digit](http://en.wikipedia.org/wiki/Most_significant_digit) (i.e., the leftmost digit). The sequence in which digits are processed by a [LSD](http://en.wikipedia.org/wiki/Least_significant_digit) radix sort is the opposite of the sequence in which digits are processed by a [most significant digit](http://en.wikipedia.org/wiki/Most_significant_digit) (MSD) radix sort.

An [LSD](http://en.wikipedia.org/wiki/Least_significant_digit) radix sort operates in [O](http://en.wikipedia.org/wiki/Big_O_notation)(*nk*) time, where *n* is the number of keys, and *k* is the average key length. This kind of performance for variable-length keys can be achieved by grouping all of the keys that have the same length together and separately performing an LSD radix sort on each group of keys for each length, from shortest to longest, in order to avoid processing the whole list of keys on every sorting pass.

A radix sorting algorithm was originally used to sort [punched cards](http://en.wikipedia.org/wiki/Punched_card) in several passes. A computer algorithm was invented for radix sort in 1954 at [MIT](http://en.wikipedia.org/wiki/Massachusetts_Institute_of_Technology) by [Harold H. Seward](http://en.wikipedia.org/wiki/Harold_H._Seward). In many large applications needing speed, the computer radix sort is an improvement on (slower) comparison sorts.

LSD radix sorts have resurfaced as an alternative to high performance [comparison-based sorting algorithms](http://en.wikipedia.org/wiki/Comparison_sort) (like [heapsort](http://en.wikipedia.org/wiki/Heapsort" \o "Heapsort) and [mergesort](http://en.wikipedia.org/wiki/Mergesort" \o "Mergesort)) that require Ω(*n* · log *n*) comparisons, where *n* is the number of items to be sorted. [Comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort) can do no better than Ω(*n* · log *n*) execution time but offer the flexibility of being able to sort with respect to more complicated orderings than a lexicographic one; however, this ability is of little importance in many practical applications.

### Definition

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit. Then, the process repeats with the next neighboring digit until there are no more digits to process. In other words:

1. Take the least significant digit (or group of bits, both being examples of [radices](http://en.wikipedia.org/wiki/Radix)) of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys. (This is what makes the LSD radix sort a [stable sort](http://en.wikipedia.org/wiki/Stable_sort)).
3. Repeat the grouping process with each more significant digit.

The sort in step 2 is usually done using [**bucket sort**](http://en.wikipedia.org/wiki/Bucket_sort)**or**[**counting sort**](http://en.wikipedia.org/wiki/Counting_sort), which are efficient in this case since there are usually only a small number of digits.

### An example

Some LSD radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The number of times that each digit occurs is stored in an [array](http://en.wikipedia.org/wiki/Array_data_type). Consider the previous list of keys viewed in a different way:

170, 045, 075,090, 002, 024, 802, 066

The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:

2 (bucket size for digits of 0: 170, 090)

2 (bucket size for digits of 2: 002, 802)

1 (bucket size for digits of 4: 024)

2 (bucket size for digits of 5: 045, 075)

1 (bucket size for digits of 6: 066)

A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:

2 (bucket size for digits of 0: 002, 802)

1 (bucket size for digits of 2: 024)

1 (bucket size for digits of 4: 045)

1 (bucket size for digits of 6: 066)

2 (bucket size for digits of 7: 170, 075)

1 (bucket size for digits of 9: 090)

A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:

6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)

1 (bucket size for digits of 1: 170)

1 (bucket size for digits of 8: 802)

At least one LSD radix sort implementation now counts the number of times that each digit occurs in each column for all columns in a single counting pass. (See the [external links](http://en.wikipedia.org/wiki/Radix_sort#External_links)section.) Other LSD radix sort implementations allocate space for buckets dynamically as the space is needed.

### Iterative version using queues

A simple version of an LSD radix sort can be achieved using [queues](http://en.wikipedia.org/wiki/Queue_(data_structure)) as buckets. The following process is repeated for a number of times equal to the length of the longest key:

1. The integers are enqueued into an array of ten separate queues based on their digits from right to left. Computers often represent integers internally as fixed-length binary digits. Here, we will do something analogous with fixed-length decimal digits. So, using the numbers from the previous example, the queues for the 1st pass would be:

0: 170, 090

1: none

2: 002, 802

3: none

4: 024

5: 045, 075

6: 066

7–9: none

1. The queues are dequeued back into an array of integers, in increasing order. Using the same numbers, the array will look like this after the first pass:

170, 090, 002, 802, 024, 045, 075, 066

1. For the second pass:

Queues:

0: 002, 802

1: none

2: 024

3: none

4: 045

5: none

6: 066

7: 170, 075

8: none

9: 090

Array:

002, 802, 024, 045, 066, 170, 075, 090  
(note that at this point only 802 and 170 are out of order)

1. For the third pass:

Queues:

0: 002, 024, 045, 066, 075, 090

1: 170

2–7: none

8: 802

9: none

Array:

002, 024, 045, 066, 075, 090, 170, 802 (sorted)

While this may not be the most efficient radix sort algorithm, it is relatively simple, and still quite efficient.

## Most significant digit radix sorts

A [most significant digit](http://en.wikipedia.org/wiki/Most_significant_digit) (MSD) radix sort can be used to sort keys in [lexicographic order](http://en.wikipedia.org/wiki/Lexicographic_order). Unlike a least significant digit (LSD) radix sort, a most significant digit radix sort does not **necessarily** preserve the original order of duplicate keys. An MSD radix sort stops rearranging the position of a key when the processing reaches a unique prefix of the key. Some MSD radix sorts use one level of buckets in which to group the keys. See the [counting sort](http://en.wikipedia.org/wiki/Counting_sort) and [pigeonhole sort](http://en.wikipedia.org/wiki/Pigeonhole_sort) articles. Other MSD radix sorts use multiple levels of buckets, which form a [trie](http://en.wikipedia.org/wiki/Trie" \o "Trie) or a path in a trie. A [postman's sort / postal sort](http://en.wikipedia.org/wiki/Bucket_sort#Postman.27s_sort) is a kind of MSD radix sort.

### Recursion

A [recursively](http://en.wikipedia.org/wiki/Recursion) subdividing MSD radix sort algorithm works as follows:

1. Take the most significant digit of each key.
2. Sort the list of elements based on that digit, grouping elements with the same digit into one [bucket](http://en.wikipedia.org/wiki/Bucket_(computing)).
3. Recursively sort each bucket, starting with the next digit to the right.
4. [Concatenate](http://en.wikipedia.org/wiki/Concatenate) the buckets together in order.

#### [[edit](http://en.wikipedia.org/w/index.php?title=Radix_sort&action=edit&section=11)]Implementation

A two-pass method can be used to first find out how big each bucket needs to be and then place each key (or pointer to the key) into the appropriate bucket. A single-pass system can also be used, where each bucket is dynamically allocated and resized as needed, but this runs the risk of serious memory fragmentation, discontiguous allocations of memory, which may degrade performance. This memory fragmentation could be avoided if a fixed allocation of buckets is used for all possible values of a digit, but, for an 8-bit digit, this would require 256 (28) buckets, even if not all of the buckets were used. So, this approach might use up all available memory quickly and go into paging space, where data is stored and accessed on a hard drive or some other secondary memory device instead of main memory, which would radically degrade performance. A fixed allocation approach would only make sense if each digit was very small, such as a single bit.

### [[edit](http://en.wikipedia.org/w/index.php?title=Radix_sort&action=edit&section=12)]Recursive forward radix sort example

Sort the list:  
170, 045, 075, 090, 002, 024, 802, 066

1. Sorting by most significant digit (100s place) gives:  
   Zero hundreds bucket: 045, 075, 090, 002, 024, 066  
   One hundreds bucket: 170  
   Eight hundreds bucket: 802
2. Sorting by next digit (10s place) is only needed for those numbers in the zero hundreds bucket (no other buckets contain more than one item):  
   Zero tens bucket: 002  
   Twenties bucket: 024  
   Forties bucket: 045  
   Sixties bucket: 066  
   Seventies bucket: 075  
   Nineties bucket: 090
3. Sorting by least significant digit (1s place) is not needed, as there is no tens bucket with more than one number. Therefore, the now sorted zero hundreds bucket is concatenated, joined in sequence, with the one hundreds bucket and eight hundreds bucket to give:  
   002, 024, 045, 066, 075, 090, 170, 802

This example used [base](http://en.wikipedia.org/wiki/Base_(exponentiation)) ten digits for the sake of readability, but of course binary digits or perhaps [bytes](http://en.wikipedia.org/wiki/Byte) might make more sense for a binary computer to process.

### [[edit](http://en.wikipedia.org/w/index.php?title=Radix_sort&action=edit&section=13)]In-place MSD radix sort implementations

Binary MSD radix sort, also called binary quicksort, can be implemented in-place by splitting the input array into two bins - the 0's bin and the 1's bin. The 0's bin is grown from the beginning of the array, whereas the 1's bin is grown from the end of the array. The 0's bin boundary is placed before the first array element. The 1's bin boundary is placed after the last array element. The most significant bit of the first array element is examined. If this bit is a 1, then the first element is swapped with the element in front of the 1's bin boundary (the last element of the array), and the 1's bin is grown by one element by decrementing the 1's boundary array index. If this bit is a 0, then the first element remains at its current location, and the 0's bin is grown by one element. The next array element examined is the one in front of the 0's bin boundary (i.e. the first element that is not in the 0's bin or the 1's bin). This process continues until the 0's bin and the 1's bin reach each other. The 0's bin and the 1's bin are then sorted recursively based on the next bit of each array element. Recursive processing continues until the least significant bit has been used for sorting.[[2]](http://en.wikipedia.org/wiki/Radix_sort#cite_note-2) [[3]](http://en.wikipedia.org/wiki/Radix_sort#cite_note-3) Handling signed integers requires treating the most significant bit with the opposite sense, followed by unsigned treatment of the rest of the bits.

In-place MSD binary-radix sort can be extended to larger radix and retain in-place capability. [Counting sort](http://en.wikipedia.org/wiki/Counting_sort) is used to determine the size of each bin and their starting index. Swapping is used to place the current element into its bin, followed by expanding the bin boundary. As the array elements are scanned the bins are skipped over and only elements between bins are processed, until the entire array has been processed and all elements end up in their respective bins. The number of bins is the same as the radix used - e.g. 16 bins for 16-Radix. Each pass is based on a single digit (e.g. 4-bits per digit in the case of 16-Radix), starting from the [most significant digit](http://en.wikipedia.org/wiki/Most_significant_digit). Each bin is then processed recursively using the next digit, until all digits have been used for sorting.[[4]](http://en.wikipedia.org/wiki/Radix_sort#cite_note-4)[[5]](http://en.wikipedia.org/wiki/Radix_sort#cite_note-5)

In-place binary-radix sort and n-bit-radix sort discussed in paragraphs above are both not [stable algorithms](http://en.wikipedia.org/wiki/Sorting_algorithm).

### [[edit](http://en.wikipedia.org/w/index.php?title=Radix_sort&action=edit&section=14)]Stable MSD radix sort implementations

MSD Radix Sort can be implemented as a stable algorithm, but requires the use of a memory buffer of the same size as the input array. This extra memory allows the input buffer to be scanned from the first array element to last, and move the array elements to the destination bins in the same order. Thus, equal elements will be placed in the memory buffer in the same order they were in the input array. The MSD-based algorithm uses the extra memory buffer as the output on the first level of recursion, but swaps the input and output on the next level of recursion, to avoid the overhead of copying the output result back to the input buffer. Each of the bins are recursively processed, as is done for the in-place MSD Radix Sort. After the sort by the last digit has been completed, the output buffer is checked to see if it is the original input array, and if it's not, then a single copy is performed. If the digit size is chosen such that the key size divided by the digit size is an even number, the copy at the end is avoided.[[6]](http://en.wikipedia.org/wiki/Radix_sort#cite_note-6)

<http://www.cs.washington.edu/education/courses/cse326/06au/lectures/lect20.pdf>

Radix Sort

• In practice

– RadixSort only good for large number of elements with relatively small values

– Hard on the cache compared to MergeSort/QuickSort

<http://www.cs.manchester.ac.uk/ugt/COMP26912/lecture/lecture-sorting.pdf>

**Bucket Sort**

Buckets need not be of size 1. If larger buckets are used (as above),

then an extra sorting procedure can be used to sort the contents of

each bucket.

Bucket sort can be implemented to be stable.

**Radix Sort**

(See pp. 242–243 of Goodrich & Tamassia)

Principle: To sort keys consisting of a sequence of symbols (e.g.

words, n-digit numbers), we can apply a bucket sort to each symbol in

turn, i.e. do multiple passes of the bucket sort.

Time Complexity: Radix sort is an O(n) sorting algorithm, if the

number of symbols in each sorting key is considered to be a

constant. Why?

Advantages: This extends the range of applications for which a

bucket sort is suitable. It would be impractical to sort (dictionary)

words by a bucket sort because it would be difﬁcult to index words

into buckets. Using radix sort, we just need 26 buckets (1 per letter).

To sort a sequence of integers of arbitrary length, ﬁrst we left-ﬁll each

integer with zeros so that all of them have the same length. E.g.

1, 100, 33 becomes 001, 100, 033

Then we use bucket sort to sort by the least signiﬁcant digit.

100, 001, 033

Next, we sort by the next more signiﬁcant digit.

100, 001, 033

And ﬁnally by the most signiﬁcant digit.

001, 033, 100.

The bucket sort must be stable. Why?

* **Radix-Sort is a stable sort**.

The running time of Radix-Sort is d times the running time of the algorithm for digit-wise sorting. **Can use counting sort to do this**

* **Bucket-Sort**:
  1. sort each bucket using a simple algorithm, e.g., Insertion-Sort,

Radix Sort

* This sort is unusual because it does not directly compare any of the elements
* We instead create a set of buckets and repeatedly separate the elements into the buckets
* On each pass, we look at a different part of the elements
* The buckets are actually queues so the elements are added at the end of the bucket
* On the second pass, we separate the elements based on the “tens” digit, and on the third pass we separate them based on the “hundreds” digit
* Each pass must make sure to process the elements in order and to put the buckets back together in the correct order
* Though this is a very time efficient algorithm it is not space efficient
* If an array is used for the buckets and we have ***B* buckets**, we would need ***N*\**B* extra memory locations** because it’s possible for all of the elements to wind up in one bucket
* If linked lists are used for the buckets you have the overhead of pointers

The Algorithm to sort a set of numeric keys

**shift = 1**

for pass = 1 to keySize do

for entry = 1 to N do

bucketNumber = (**list[entry] / shift**) **mod 10**

**Append**( bucket[bucketNumber], list[entry] )

end for

list = **CombineBuckets()**

**shift = shift \* 10**

end for