Topics: Hashing, Sorting

1 Hashing

While binary search trees provide logarithmic search times on average, we would like to do even better. For integer keyed items, we can trivially get constant search time by using an array, with one position for each possible key. However, with 2^{32} possible keys, there's no way we could create an array large enough.

Instead, we could get almost constant search time if we create an array twice as large as our data set, and store a value k at position $kmodm$, where m is the size of the array. To search for a value k, we again compute k mod m and then check the corresponding array position. We would expect an even distribution of values, so we expect one or two values per array position. To retain this expectation, we may need to resize the array as we insert values. We want to keep the *load factor*, $\frac{k}{m}$, at about 0.5. Resizing potentially requires us to move all values, but the total amortized cost of insertion is still constant. Searching is also constant in this scheme.

There is the problem of multiple values mapping to the same array position, or *bucket*. This situation is called a *collision*. There are multiple ways of resolving collisions. One way is to place a value that maps to an occupied bucket to the next empty bucket. Another is to have linked lists at each bucket, and store values in the lists. This is the solution we'll use.

This is fine for integer values, but what about arbitrary objects? We first compute an integer corresponding to the object, called a *hash code*, and use that to map the objects to positions. There are two conditions on a hash code. Equal objects (as defined by the `equals()` method) must have equal hash codes, and the hash code for an object cannot change while it is in a *hash table*. Then, assuming the hash code can be computed quickly and distributes objects evenly in a table, this gives us constant-time searching for arbitrary objects.

How should we compute a hash code on an object? Many objects can be represented as k-tuples of integers. For example, an object corresponding to a point in two-dimensional space can be represented by a double of integers, converting the floating point x and y values bitwise to integers. A good hash code for a k-tuple $(x_0, x_1, ..., x_{k-1})$ is

$$x_0 \cdot a^{n-1} + x_1 \cdot a^{n-2} + ... + x_{n-2} \cdot a + x_{n-1}$$

where a is a constant. Choosing a prime number minimizes the chance of a collision.

Our hash code formula works even if k is unbounded, such as in strings. Java's `String` class uses the following formula:

```
public int hashCode() {
  int code = 0;
  for (int i = 0; i < length(); i++) {
    code = 31 * code + charAt(i);
  }
}
```

This is our formula, with $a = 31$ and $x_i =$`charAt(i)`.

2 Sorting

The topic of sorting really requires no introduction. We start with an unsorted sequence, and want a sorted sequence as the result. We begin by discussing simple but slow sorts, and move on to quicker but more complicated algorithms. We will neglect the possibility of duplicate elements for most of the sorts, and we will use integers as the keys in all of them.

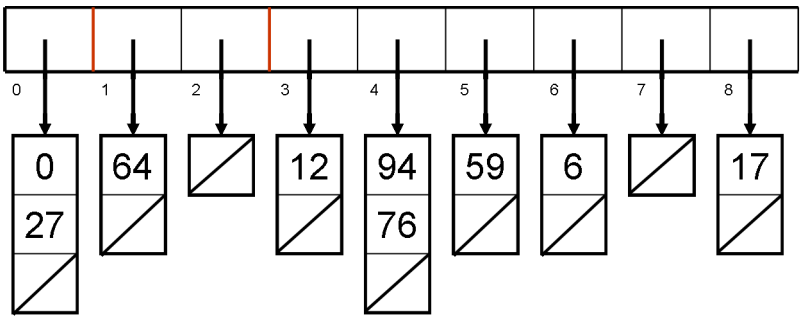

Figure 1: A fast integer lookup table, with a load factor of 1.

2.1 Selection Sort

An obvious sorting algorithm is to repeatedly find and remove the smallest item in a set and store that item in a result set. This is a type of *selection sort*. The actual algorithm we will use is *in-place*, so the result set is the start set. We have to keep track of what part of the set is sorted and what part isn't. Instead of removing the smallest element, we swap it with the first element in the unsorted part, and then extend the sorted part to include the smallest element. A Java implementation is as follows:

```
void selectionSort(int[] x) {
  int tmp, minpos;
  for (int i = 0; i < x.length; i++) { // i is the start of the unsorted part
    // find the smallest remaining element
    minpos = i;
    for (int j = i+1; j < x.length; j++) {
      if (x[j] < x[minpos]) {
        minpos = j;
      }
    }
    // swap the smallest element with the first unsorted element
    tmp = x[i];
    x[i] = x[minpos];
    x[minpos] = tmp;
  }
}
```

Selection sort is a particulary slow algorithm. Finding the smallest remaining unsorted element takes $\frac{n}{2}$ time on average, and must be done n times, for an $O(n^2)$ running time. The running time is always the same regardless of the distribution of values in the input.

2.2 Insertion Sort

Another obvious way to sort a set is to start with an empty result set and repeatedly remove the first element in the original set and insert it into the proper location in the result set. This can be easily accomplished using linked lists. A pseudocode algorithm:

```
LinkedList insertionSort(LinkedList x) {
  Create a new LinkedList y;
  while x is not empty do:
    temp = x.remove(0);
```

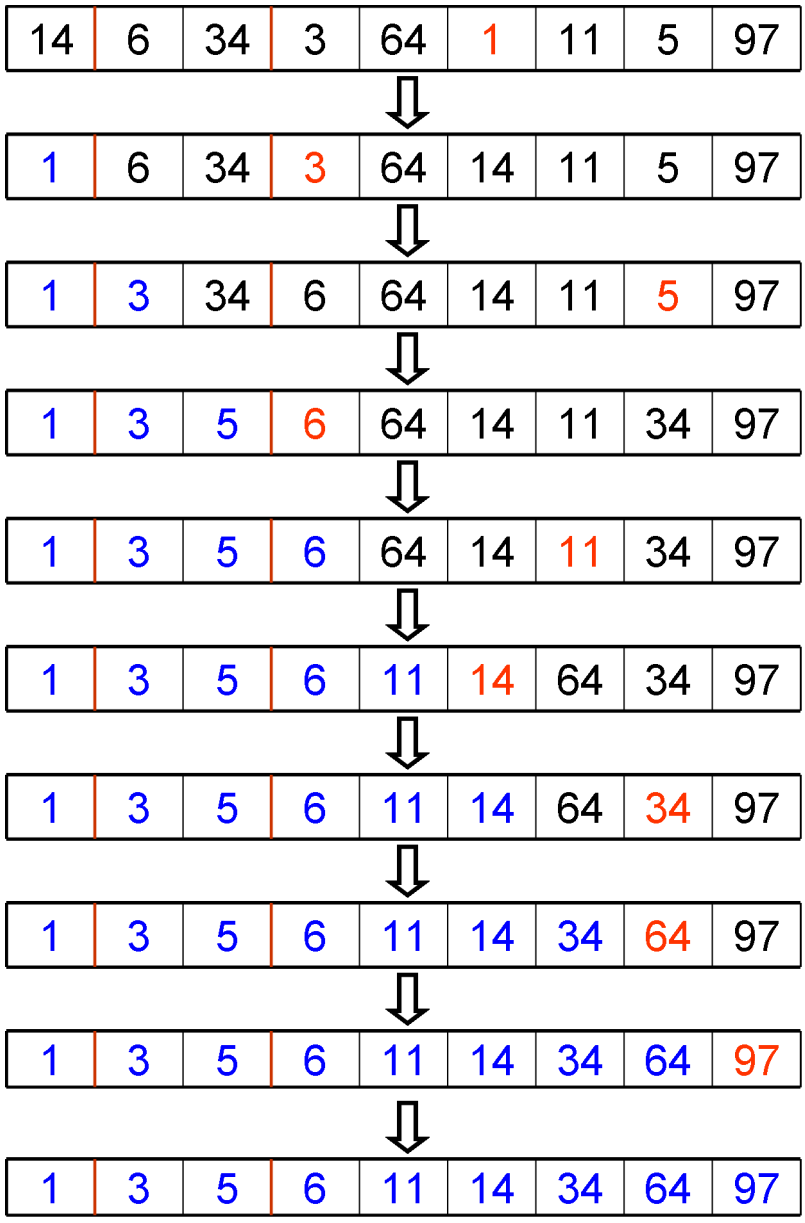

Figure 2: Selection sort applied to a sequence of integers.

```
    Find the first element in y that is greater than temp, call it z;
    if z exists then:
      Insert x before z;
    else:
      Insert x at the end of y;
    fi;
  od;
  return y;
}
```

On average, finding the greater element in the result list takes $\frac{n}{2}$ time, and must be done n times, for a total of $O(n^2)$ running time. However, in some cases such as a reverse sorted list, finding the element is constant, so the running time reduces to linear. Modifications to the algorithm can be made such that sorted rather than reverse sorted lists can be linearly resorted.

2.3 Merge Sort

A common method of reducing the time required to solve a problem is to use the *divide and conquer* approach. Rather than directly tackling the problem, divide the problem into smaller pieces and recursively solve the smaller pieces (of course, dividing them into smaller pieces as well, and those pieces into yet smaller pieces, and so on).

Merge sort is an example of divide and conquer applied to sorting. A set is divided into two as equal as possible subsets, and the subsets are recursively merge sorted. Then the two now sorted subsets are *merged*, or combined such that the ordering is preserved. This merging can be done by walking simultaneously through both subsets, choosing the smaller of the two current elements to place in the result set, and advancing the position in the subset that contained that element. The algorithms for merge and merge sort are as follows:

```
int[] merge(int[] y, int[] z) {
  Create a new array u;
  int i = j = 0;
  while i < y.length and j < z.length do:
    if y[i] < z[j] then:
      u[i+j] = y[i];
      i = i + 1;
    else:
      u[i+j];
      j = j + 1;
    fi;
  if i < y.length then:
    Append remaining elements of y to u;
  else:
    Append remaining elements of z to u;
  fi;
  return u;
}

int[] mergeSort(int[] x) {
  if x.length == 1 then:
    return x;
  else:
    Split the array into two as equal as possible pieces y and z.
    y = mergeSort(y);
```

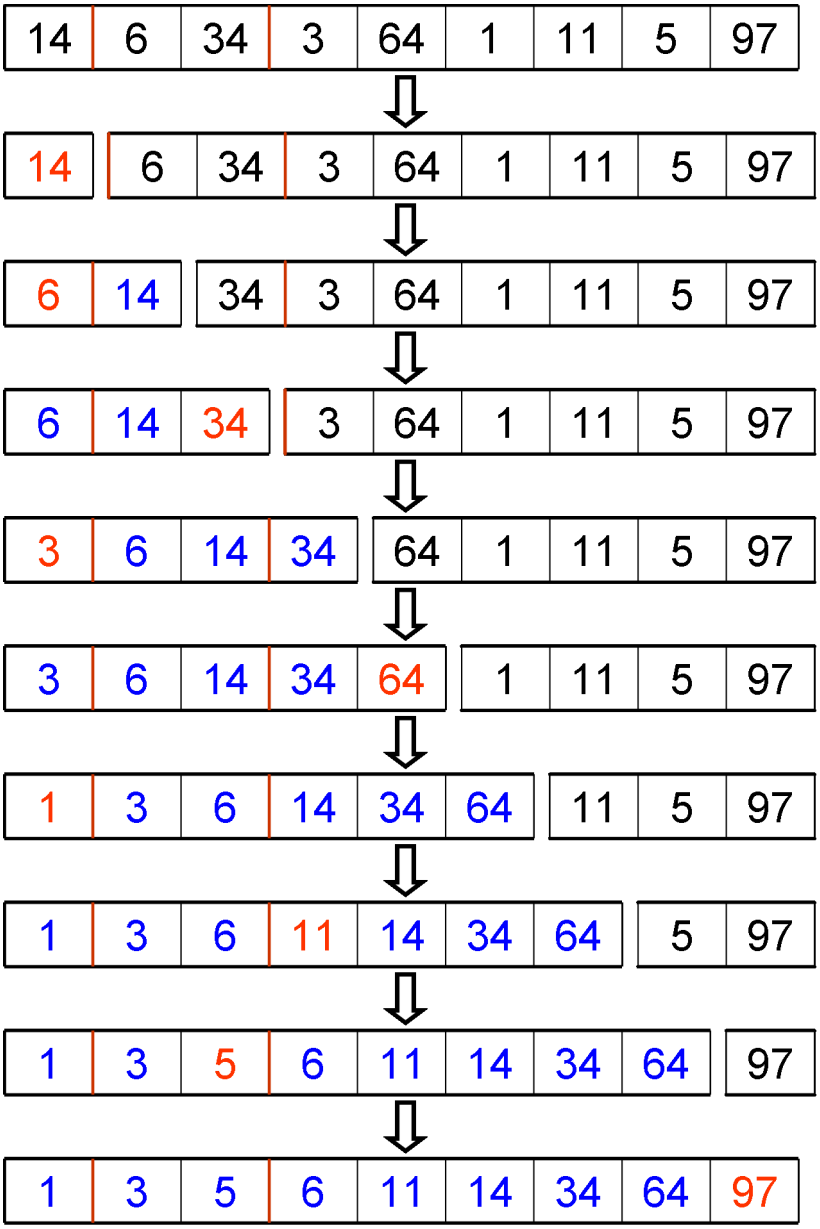

Figure 3: Insertion sort applied to a sequence of integers.

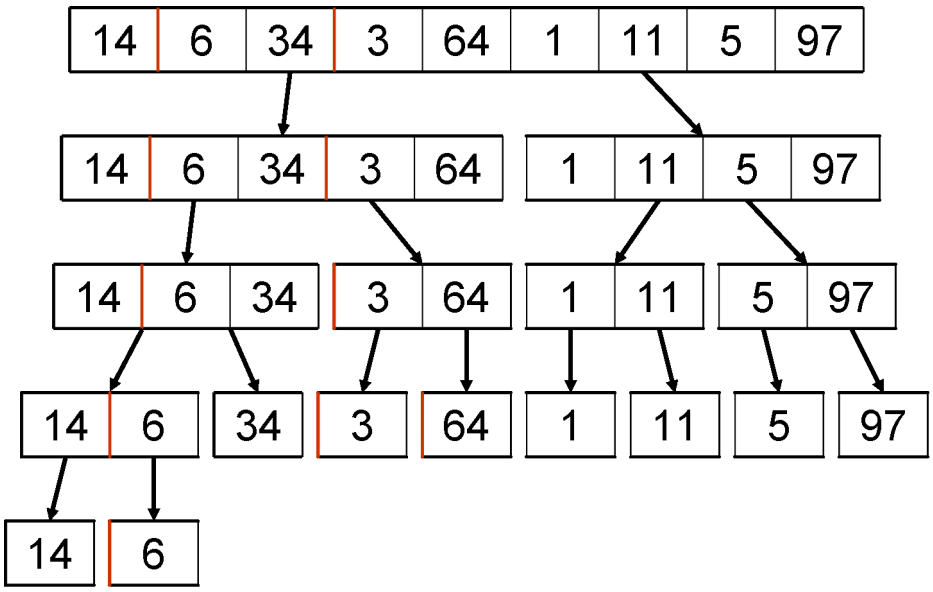

Figure 4: The divide step of a merge sort.

```
    z = mergeSort(z);
    return merge(y, z);
  fi;
}
```

The stack of recursive calls in merge sort are arranged as a complete binary tree, with about $2n$ recursive calls. Thus the divide portion of the algorithm runs in linear time. Now before each recursive call returns, it merges the results of the next two recursive calls, which is linear in the number of elements to be merged. But the total number of elements to be merged in each level of the tree is n, so each level takes n time to merge. The tree has about $\lg n$ levels, so the merge step takes $n \lg n$ total time. Thus merge sort runs in $O(n \lg n)$.

2.4 Quick Sort

Quick sort is another divide and conquer algorithm, but rather than doing all the sorting work in the conquer step, quick sort does it when dividing. Then the conquer part consists only of concatenating already sorted subsets.

In the quick sort algorithm, a set is divided into subsets according to a *pivot*, chosen as either the last element or a random element in the set. All elements less than the pivot are placed in one subset, and all elements greater than the pivot are placed in another. The two subsets are recursively quick sorted, and the results concatenated but with the pivot in between. Quick sort can be done in place, but the following algorithm is not in place:

```
LinkedList quickSort(LinkedList x) {
  if x.length() <= 1 then:
    return x;
  else:
    Choose a "pivot" element in x;
    Create new LinkedLists y and z;
    Place all elements less than the pivot in y;
```

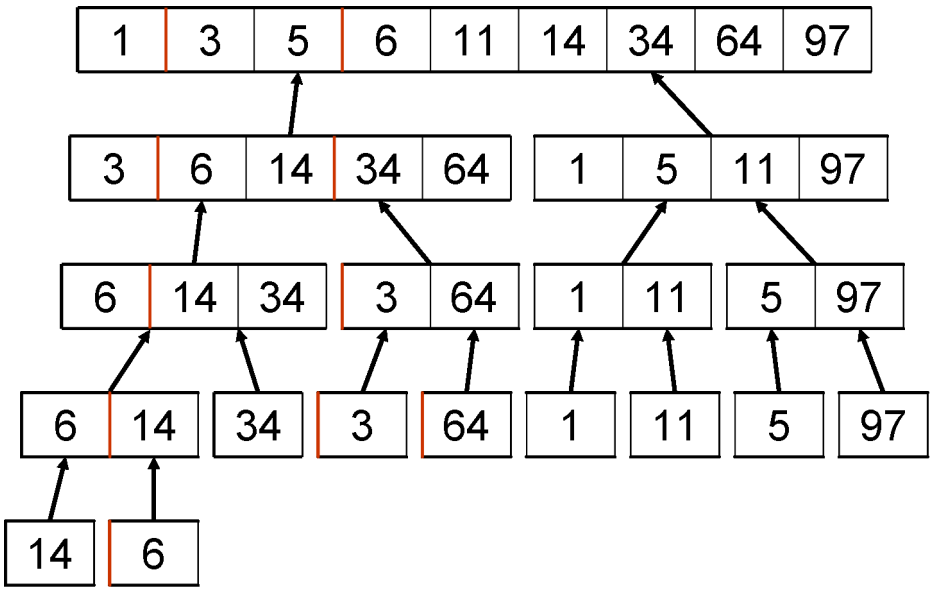

Figure 5: The merge step of a merge sort.

```
    Place all elements greater than the pivot in z;
    y = quickSort(y);
    z = quickSort(z);
    return y + pivot + z;
  fi;
}
```

In each set division, we expect to divide the set in half, so we expect the stack of recursive calls to again look like a complete tree. Dividing each level takes linear time, so we expect a running time in $O(n \lg n)$. However, in the worst case, the set is already sorted, so the stack of recursive calls will look more like a linked list than a tree if we choose the last element as the pivot. So in the worst case, quick sort runs in $O(n^2)$. If we choose random pivots, then we expect even a sorted set to take $O(n \lg n)$ time. But for any arbitrary set, there is some sequence of pivot choices that will result in quadratic running time, and our random number generator may return just this sequence. So the worst case running time is still in $O(n^2)$.

2.5 Bucket Sort

Thus far, we have only looked at comparison-based sorts, algorithms that compare elements to each other in the set to be sorted. Such sorts must take at least $O(n \lg n$ time in the worst case (the proof is in the text). In order to do better, we must come up with some way of ordering the elements without comparing them to each other.

Looking at our integer lookup table from our discussion on hash tables, we might get some inspiration. What if we created a table with one bucket for each element in our set, placed each element in its corresponding set, and iterated through the table to get the result? This is harder than it seems. Actually figuring out which buckets are necessary and then arranging them in the proper order cannot be done in less than $O(n \lg n$ time. On the other hand, we can find the range of the elements (the difference of the largest and smallest element) in linear time and just create one bucket for each possible value in the range. Then we can place each element in its corresponding bucket and iterate through the table to retrieve the ordered set. This algorithm is called *bucket sort*.

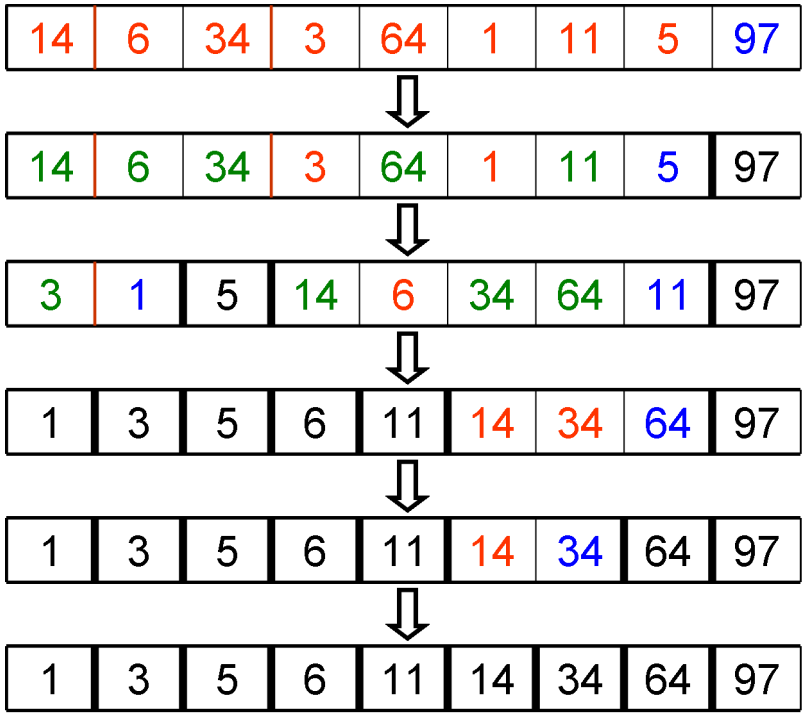

Figure 6: An in place quick sort. Set boundaries are denoted by thick lines.

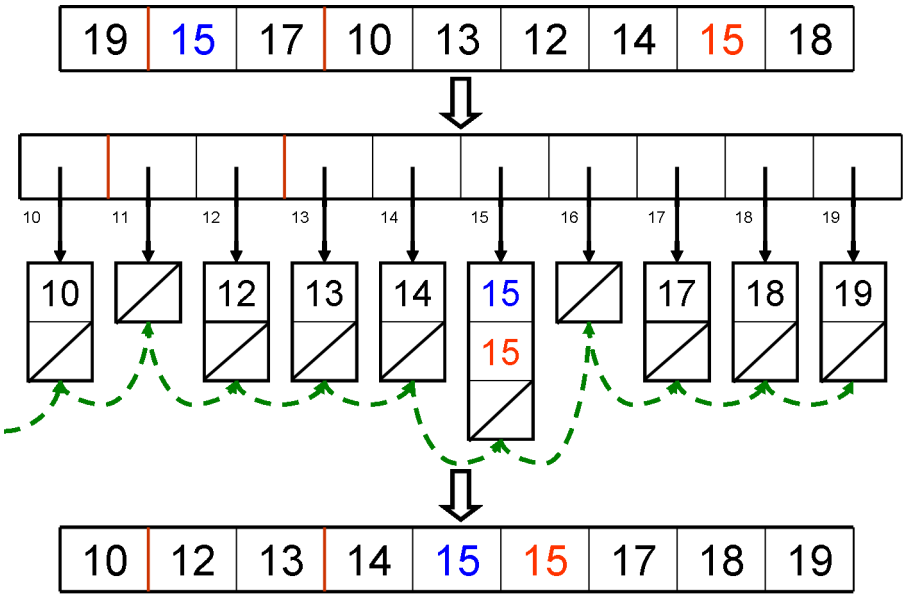

Figure 7: A bucket sort of integer values. The sort is stable.

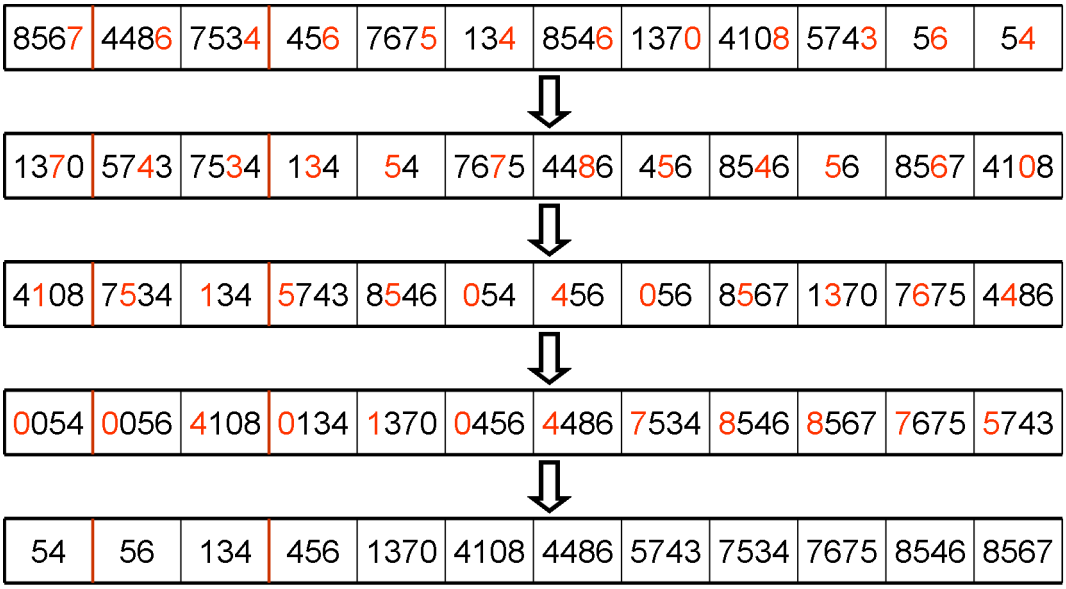

Figure 8: An LSD radix sort.

It takes linear time to compute the range of the elements and linear time to place the elements in their corresponding buckets. But iterating through the table takes time linear in the number of buckets, or linear in the range of the elements. So the running time of bucket sort is in $O(\max(n, r))$ where n is the number of elements and r is their range. If the range is on the order of n, then bucket sort is linear. Buf if the range is large, then the sort may be slower than a quadratic algorithm.

2.6 Radix Sort

Rather than directly applying bucket sort to a set of elements, we can do better by bucket sorting the set digit by digit. This is called *radix sort*.

There are two general types of radix sort. One type sorts starting with the least significant digit (LSD), and continuing on until the most significant digit (MSD). This is generally called an *LSD radix sort*. The other type starts with the most significant digit and ends with the least significant digit, called an *MSD radix sort*. The MSD radix sort is a bit trickier to implement, so we will only talk about LSD radix sort here.

In order for LSD radix sort to work, it requires bucket sort to be *stable*. A stable sort is one in which equal elements preserve their relative ordering across the sort. Bucket sort, if implemented correctly, is stable. Then all there is to LSD radix sort is to apply bucket sort to each digit, starting at the least significant digit.

The question of how fast radix sort is somewhat complicated. Each application of bucket sort is linear in the number of elements, but we must apply bucket sort k times where k is the number of digits the largest element has. So we might think that the running time is in $O(k \cdot n)$.

However, for all the comparison-based sorts, we have ignored the cost of comparing two elements. For integer values, the cost is constant since the actual comparison is a primitive CPU instruction. However, for non-integer items such as strings, the comparison is not constant. In fact, it takes k time where k is the number of digits in the item. We expect k to be logarithmic in the size of the input for large input sizes, so we assume $k = \lg n$. Then the running times for selection and insertion sorts would be in $O(n^2 \lg n)$ and for merge and quick sorts in $O(n \lg^2 n)$. Radix sort would run in a faster $O(n \lg n)$.

If we choose to ignore the comparison cost, then we can say that radix sort runs in linear time, since only in the case of integers is the comparison cost less than the number of digits. However, the number of digits

in an integer is at most 32 (since we would use the base-two representation in practice), and we can treat this as a constant factor. So radix sort is always linear.