Count sort

This strange little sorting algorithm runs in linear time. It has its limitation: the values in the array `c` to be sorted are in the range 0..`b`-1, for some given `b`. It is called *Count sort* because it counts (and then uses) the number of times each value occurs in `c`. After describing the basic algorithm, we revise it into a subroutine used in another important linear-time sorting algorithm, *radix sort*.

How counting sort works

The method creates an array `res` that contains the elements of input array `c`, but sorted. Here is an outline:

0. Suppose `c` contains the values shown to the right, so `b` = 5: `c`: (3, 4, 0, 0, 3, 1)

1. Make up an array `d`[0..4], where `d[i]` is the number of times `i` appears in `b`: `d`: (2, 1, 0, 2, 1)

2. Change array `d` so that `d`[i] is the index where the first `i` in `c` belongs.
For example, the first 0 in `c` obviously belongs in `res`[0].
Since there are two 0's in `c`, the first 1 in `c` belongs in `res`[2].
Since there are two 0's and one 1 in `c`, the first 2 belongs in `res`[3], and so on. `d`: (0, 2, 3, 3, 5)

3. Place each value `c[h]` in its position in `res`, starting with `c`[0]. If `c[h]` = `t`, then by point 3 above, `d[t]` is the index in res where `c[h]` is to be placed, so store `c[h]` in `res[d[t]]` and increase `d[t]` to indicate where the next value `t` in `c[h..]` is to be placed. We step throgh this, given arrays `b`, `d` above to the right, with values placed in `res` in red.

`h` = 0, `c`[0] = 3. Store `c`[0] in `res[d`[3]], i.e. `res`[3]. Increment `d`[3]. New `res`: (0, 0, 0, **3**, 0, 0). `d`: (0, 2, 3, 4, 5)
`h` = 1, `c`[1] = 4. Store `c`[1] in `res[d`[4]], i.e. `res`[5]. Increment `d`[4]. New `res`: (0, 0, 0, 3, 0, **4**). `d`: (0, 2, 3, 4, 6)
`h` = 2, `c`[2] = 0. Store `c`[2] in `res[d`[0]], i.e. `res`[0]. Increment `d`[0]. New `res`: (0, 0, 0, 3, 0, 4). `d`: (1, 2, 3, 4, 6)
`h` = 3, `c`[3] = 0. Store `c`[3] in `res[d`[0]], i.e. `res`[1]. Increment `d`[1]. New `res`: (0, **0**, 0, 3, 0, 4). `d`: (2, 2, 3, 4, 6)
`h` = 4, `c`[4] = 3. Store `c`[4] in `res[d`[3]], i.e. `res`[4]. Increment `d`[3]. New `res`: (0, 0, 0, 3, **3**, 4). `d`: (2, 2, 3, 5, 6)
`h` = 5, `c`[5] = 1. Store `c`[5] in `res[d`[1]], i.e. `res`[2]. Increment `d`[1]. New `res`: (0, 0, **1**, 3, 3, 4). `d`: (2, 3, 3, 5, 6)

The algorithm, in Java

```
 int[] d= new int[b];

 // Store in each d[t] the number of times t occurs in c.
for (int t: c) d[t]= d[t] + 1;

// Change each d[i] to the sum of d[0..i-1].
// Using the notation dInt[i] for the initial value of d[i], we have:
//    invariant: 0 <= h < b and
//          total = sum of dInit[0..h-1] and
//          For all i, 0 <= i < h, d[i] = sum of dInit[0..i-1] and
//          For all i, h <= i < c.length, d[i] = dInit[i].
int total= 0;
for (int h= 0; h < d.length; h= h+1) {
   int t= d[h];  d[h]= total;  total= total + t;
}

// Store keys in sorted, stable order in res
int[] res= new int[c.length];
// invariant: keys c[0..h-1] have been moved to their correct position.
//          For each t, 0 <= t < b, the next value in c[h..] that equals t
//                   (if there is one) belongs in res[d[t]].
for (int h= 0; h < c.length; h= h+1) {
   int t= c[h];
   res[d[t]]= c[h];
   d[t]= d[t] + 1;
}
```

Count sort

A more useful count sort

The algorithm presented on the previous page uses the values of the array themselves in sorting. But we can rewrite Counting sort to be more useful. For example, we can use it to:

- Sort int array `c` according to the least significant decimal digit of each `c[i]` (`b` = 10).
- Sort int array `c` according to the most significant decimal digit of each `c[i]` (`b` = 10).
- For `c` an array of dates, sort `b` according to the month (`b` = 12)
- For `c` an array of dates, sort `b` according to the day of the month (`b` = 32)

This requires giving Count sort a function `f`, where `f(c[i])` is an integer in the appropriate range 0..`b`-1. For example to sort int array `c` on its least significant decimal digit, use `f(t)` = `t` % 10.

We wrote two similar versions of this algorithm in Java. The first uses an int array. The second used an array of generic type `T`. We present the first one here. Throughout, *key* refers to a value in array `c`.

```
/** Return c sorted according to function f. The sort is stable.
  * Precondition: f(key) returns an integer in the range 0..b-1, i.e.:
  * The keys c[i] satisfy 0 <= f(key) < b.
  * The output array contains items with f(key) = 0, then items with f(key) = 1, and so on, i.e.
  * Postcondition:
  *    -----------------------------------------------------------------------------------
  * c | keys with f(key) = 0 | keys with f(key) = 1 | ... | keys with f(key) = k-1 |
  *    -----------------------------------------------------------------------------------
  * Using n for c.length, we give the space and time complexity:
  * Space used: O(n + b). Worst-case and expected time: O(n + b). */
 public static int[] countSort(int[] c, ToIntFunction<Integer> f, int b) {
    // Store in each d[i] the number of times i occurs in c.
    int[] d= new int[b];
    for (int t: c) d[f.applyAsInt(t)] += 1;

    // Change each d[i] to the sum of d[0..i-1].
    // Using the notation dInt[i] for the initial value of d[i], we have:
    // invariant: 0 <= h < b and
    //        total= sum of dInit[0..h-1] and
    //        For all i, 0 <= i < h, d[i] = sum of dInit[0..i-1] and
    //        For all i, h <= i < d.length, d[i] = dInit[i].
    int total= 0;
    for (int h= 0; h < d.length; h= h+1) {
       int t= d[h];   d[h]= total;   total= total + t;
    }

    // Store keys (i.e. values in array c) in sorted, stable order in res
    int[] res= new int[c.length];
    // invariant: keys c[0..h-1] have been placed in their correct position in res.
    //        For each t, 0 <= t < b, the next key in c[h..] with f(key) = t
    //              (if there is one) belongs in res[d[t]].
    for (int h= 0; h < c.length; h= h+1) {
       int t= f.applyAsInt(c[h]);
       res[d[t]]= c[h];
       d[t]= d[t] + 1;
    }

    return res;
 }
```