CS 61B: Lecture 35
Monday, April 21, 2014

Today's reading: Goodrich & Tamassia, Sections 11.3.2.

Counting Sort

If the items we sort are naked keys, with no associated values, bucket sort can be simplified to become _counting_sort_. In counting sort, we use no queues at all; we need merely keep a count of how many copies of each key we have encountered. Suppose we sort 6 7 3 0 3 1 5 0 3 7:

```
               0       1       2       3       4       5       6       7
           -----------------------------------------------------------------
    counts |   2   |   1   |   0   |   3   |   0   |   1   |   1   |   2   |
           -----------------------------------------------------------------
```

When we are finished counting, it is straightforward to reconstruct the sorted keys from the counts: 0 0 1 3 3 3 5 6 7 7.

Counting Sort with Complete Items

Now let's go back to the case where we have complete items (key plus associated value). We can use a more elaborate version of counting sort. The trick is to use the counts to find the right index to move each item to.

Let x be an input array of objects with keys (and perhaps other information).

```
        0      1      2      3      4      5      6      7      8      9
    ----------------------------------------------------------------------
  x |   .  |   .  |   .  |   .  |   .  |   .  |   .  |   .  |   .  |   .  |
    ----|------|------|------|------|------|------|------|------|------|---
        v      v      v      v      v      v      v      v      v      v
      -----  -----  -----  -----  -----  -----  -----  -----  -----  -----
      | 6 |  | 7 |  | 3 |  | 0 |  | 3 |  | 1 |  | 5 |  | 0 |  | 3 |  | 7 |
      -----  -----  -----  -----  -----  -----  -----  -----  -----  -----
```

Begin by counting the keys in x.

```
  for (i = 0; i < x.length; i++) {
    counts[x[i].key]++;
  }
```

Next, do a _scan_ of the "counts" array so that counts[i] contains the number of keys _less_than_ i.

```
               0       1       2       3       4       5       6       7
           -----------------------------------------------------------------
    counts |   0   |   2   |   3   |   3   |   6   |   6   |   7   |   8   |
           -----------------------------------------------------------------
```

```
  total = 0;
  for (j = 0; j < counts.length; j++) {
    c = counts[j];
    counts[j] = total;
    total = total + c;
  }
```

Let y be the output array, where we will put the sorted objects. counts[i] tells us the first index of y where we should put items with key i. Walk through the array x and copy each item to its final position in y. When you copy an item with key k, you must increment counts[k] to make sure that the next item with key k goes into the next slot.

```
  for (i = 0; i < x.length; i++) {
    y[counts[x[i].key]] = x[i];
    counts[x[i].key]++;
  }
```

```
    ---------------------      ---------------------------------
  y |.|.|.|.|.|.|.|.|.|.|    counts | 0 | 2 | 3 | 3 | 6 | 6 | 8 | 8 |
    ---------------|-----      ---------------------------------
                   v
                   6

    ---------------------      ---------------------------------
  y |.|.|.|.|.|.|.|.|.|.|    counts | 0 | 2 | 3 | 3 | 6 | 6 | 8 | 9 |
    ---------------|-|---      ---------------------------------
                   v v
                   6 7

    ---------------------      ---------------------------------
  y |.|.|.|.|.|.|.|.|.|.|    counts | 0 | 2 | 3 | 4 | 6 | 6 | 8 | 9 |
    -------|-------|-|---      ---------------------------------
           v       v v
           3       6 7

    ---------------------      ---------------------------------
  y |.|.|.|.|.|.|.|.|.|.|    counts | 1 | 2 | 3 | 4 | 6 | 6 | 8 | 9 |
    -|-----|-------|-|---      ---------------------------------
     v     v       v v
     0     3       6 7

    ---------------------      ---------------------------------
  y |.|.|.|.|.|.|.|.|.|.|    counts | 1 | 2 | 3 | 5 | 6 | 6 | 8 | 9 |
    -|-----|-|-----|-|---      ---------------------------------
     v     v v     v v
     0     3 3     6 7

    ---------------------      ---------------------------------
  y |.|.|.|.|.|.|.|.|.|.|    counts | 1 | 3 | 3 | 5 | 6 | 6 | 8 | 9 |
    -|---|-|-|-----|-|---      ---------------------------------
     v   v v v     v v
     0   1 3 3     6 7

...

    ---------------------      ----------------------------------
  y |.|.|.|.|.|.|.|.|.|.|    counts | 2 | 3 | 3 | 6 | 6 | 7 | 8 | 10 |
    -|-|-|-|-|-|-|-|-|-|-      ----------------------------------
     v v v v v v v v v v
     0 0 1 3 3 3 5 6 7 7
```

Bucket sort and counting sort both take O(q + n) time. If q is in O(n), then they take O(n) time. If you're sorting an array, counting sort is slightly faster and takes less memory than bucket sort, though it's a little harder to understand. If you're sorting a linked list, bucket sort is more natural, because you've already got listnodes ready to put into the buckets.

However, if q is not in O(n)--there are many more _possible_values_ for keys than keys--we need a more aggressive method to get linear-time performance.

Radix Sort

Suppose we want to sort 1,000 items in the range from 0 to 99,999,999. If we use bucket sort, we'll spend so much time initializing and concatenating empty queues we'll wish we'd used selection sort instead.

Instead of providing 100 million buckets, let's provide q = 10 buckets and sort on the first digit only. (A number less than 10 million is said to have a first digit of zero.) We use bucket sort or counting sort, treating each item as if its key is the first digit of its true key.

```
      0      1      2      3      4      5      6      7      8      9
 -----------------------------------------------------------------------
 |   .  |   .  |   *  |   .  |   *  |   .  |   .  |   .  |   *  |   .  |
 ----|------|-------------|-------------|------|------|-------------|---
     v      v             v             v      v      v             v
  ------ ------        ------        ------ ------ ------        ------
  | 342| |1390|        |3950|        |5384| |6395| |7394|        |9362|
  |9583| |5849|        |8883|        |2356| |1200| |2039|        |9193|
  ---|-- ------        ---|--        ------ ------ ---|--        ---|--
     v                    v                           v             v
  ------               ------                      ------        ------
  |  59|               |3693|                      |7104|        |9993|
  |2178|               |7834|                      |2114|        |3949|
  ------               ------                      ------        ------
```

Once we've dealt all 1,000 items into ten queues, we could sort each queue recursively on the second digit; then sort the resulting queues on the third digit, and so on. Unfortunately, this tends to break the set of input items into smaller and smaller subsets, each of which will be sorted relatively inefficiently.

Instead, we use a clever but counterintuitive idea: we'll keep all the numbers together in one big pile throughout the sort; but we'll sort on the _last_ digit first, then the next-to-last, and so on up to the most significant digit.

The reason this idea works is because bucket sort and counting sort are stable. Hence, once we've sorted on the last digit, the numbers 55,555,552 and 55,555,558 will remain ever after in sorted order, because their other digits will be sorted stably. Consider an example with three-digit numbers:

```
Sort on 1s:    771 721 822 955 405   5 925 825 777  28 829
Sort on 10s:   405   5 721 822 925 825  28 829 955 771 777
Sort on 100s:    5  28 405 721 771 777 822 825 829 925 955
```

After we sort on the middle digit, observe that the numbers are sorted by their last two digits. After we sort on the most significant digit, the numbers are completely sorted.

Returning to our eight-digit example, we can do better than sorting on one decimal digit at a time. With 1,000 keys, sorting would likely be faster if we sort on two digits at a time (using a base, or _radix_, of q = 100) or even three (using a radix of q = 1,000). Furthermore, there's no need to use decimal digits at all; on computers, it's more natural to choose a power-of-two radix like q = 256. Base-256 digits are easier to extract from a key, because we can quickly pull out the eight bits that we need by using bit operators (which you'll study in detail in CS 61C).

Note that q is both the number of buckets we're using to sort, and the radix of the digit we use as a sort key during one pass of bucket or counting sort. "Radix" is a synonym for the base of a number, hence the name "radix sort."

How many passes must we perform? Each pass inspects log2 q bits of each key. If all the keys can be represented in b bits, the number of passes is ceiling(b / log2 q). So the running time of radix sort is in

$$O\left((n + q) \left\lceil \frac{b}{\log_2 q} \right\rceil \right).$$

How should we choose the number of queues q? Let's choose q to be in O(n), so each pass of bucket sort or counting sort takes O(n) time. However, we want q to be large enough to keep the number of passes small. Therefore, let's choose q to be approximately n. With this choice, the number of passes is in O(1 + b / log2 n), and radix sort takes

$$O\left(n + \frac{b}{\log n} n\right) \text{ time.}$$

For many kinds of keys we might sort (like ints), b is technically a constant, and radix sort takes linear time. Even if the key length b tends to grow logarithmically with n (a reasonable model in many applications), radix sort runs in time linear in the total number of bits in all the keys together.

A practical, efficient choice is to make q equal to n rounded down to the next power of two. If we want to keep memory use low, however, we can make q equal to the square root of n, rounded to the nearest power of two. With this choice, the number of buckets is far smaller, but we only double the number of passes.

Postscript: Radix Sort Rocks (not examinable)

Linear-time sorts tend to get less attention than comparison-based sorts in most computer science classes and textbooks. Perhaps this is because the theory behind linear-time sorts isn't as interesting as for other algorithms. Nevertheless, the library sort routines for machines like Crays use radix sort, because it kicks major ass in the speed department.

Radix sort can be used not only with integers, but with almost any data that can be compared bitwise, like strings. The IEEE standard for floating-point numbers is designed to work with radix sort combined with a simple prepass and postpass (to flip the bits, except the sign bit, of each negative number).

Strings of different lengths can be sorted in time proportional to the total length of the strings. A first stage sorts the strings by their lengths. A second stage sorts the strings character by character (or several characters at a time), starting with the last character of the longest string and working backward to the first character of every string. We don't sort every string during every pass of the second stage; instead, a string is included in a pass only if it has a character in the appropriate place.

For instance, suppose we're sorting the strings CC, BA, CCAAA, BAACA, and BAABA. After we sort them by length, the next three passes sort only the last three strings by their last three characters, yielding CCAAA BAABA BAACA. The fifth pass is on the second character of each string, so we prepend the two-character strings to our list, yielding CC BA CCAAA BAABA BAACA. After sorting on the second and first characters, we end with

```
  BA BAABA BAACA CC CCAAA.
```

Observe that BA precedes BAABA and CC precedes CCAAA because of the stability of the sort. That's why we put the two-character strings before the five-character strings when we began the fifth pass.