1. Intro. I'm experimenting with the interesting algorithm proposed by Sourav Chakraborty, N. V. Vinodchandran, and Kuldeep S. Meel [ESA 2022, the 30th annual European Conference on Algorithms, paper 39] for estimating the number of distinct elements in a given stream.

I've modified their algorithm slightly, so that its estimates will be thoroughly unbiased, in the sense that the expected value will be exact. More precisely, the expected value would be exact if I were able to use infinite-precision arithmetic, instead of the 31-bit precision of this particular program.

The command line names five things: *bufsize*, the buffer size; *del*, the sampling interval; *length*, the total number of elements in the stream; *param*, a parameter that helps to define the simulated streaming data; and *seed*, the seed for pseudorandom numbers.

Every *del* steps, I print the current estimate, the actual number of distinct elements seen so far, the ratio of those two numbers, and the current probability cutoff. This continues until *length* elements have been streamed.

The stream is simulated here as a sequence of random nonnegative integers less than *param*. With change files I can supply other test data.

The buffer is implemented as a treap of maximum size *bufsize*. I also maintain a simple hash table — but only to keep track of the actual number of distinct elements. (The hash table isn't part of the algorithm.)

```
#define hashsize (1 ≪ 25)     /* must be a power of 2 */
#define maxsize 1000000      /* the treap has at most this many nodes */
#include <stdio.h>
#include <stdlib.h>
#include "gb_flip.h"
  int bufsize, del, length, param, seed;     /* command-line parameters */
  int ssize;     /* bufsize + 1 */
  ⟨Type definitions 7⟩;
  ⟨Global variables 5⟩;
  ⟨Subroutines 10⟩;
  void main(int argc, char *argv[])
  {
    register int a, h, i, j, k, l, m, q, r, u;
    register unsigned int p;     /* a probability, times 2^31 */
    ⟨Process the command line 2⟩;
    ⟨Do the experiment 3⟩;
  }
```

2. ⟨Process the command line 2⟩ ≡

```
  if (argc ≠ 6 ∨ sscanf(argv[1], "%d", &bufsize) ≠ 1 ∨ sscanf(argv[2], "%d", &del) ≠ 1 ∨ sscanf(argv[3],
        "%d", &length) ≠ 1 ∨ sscanf(argv[4], "%d", &param) ≠ 1 ∨ sscanf(argv[5], "%d", &seed) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣bufsize␣del␣length␣param␣seed\n", argv[0]);
    exit(−1);
  }
  ssize = bufsize + 1;
  if (bufsize ≤ 0 ∨ bufsize ≥ maxsize) {
    fprintf(stderr, "the␣buffer␣size␣must␣be␣positive␣and␣less␣than␣%d!\n", maxsize);
    exit(−2);
  }
  gb_init_rand(seed);     /* warm up the random number generator */
  hashmult = ((int)(.61803398875 * (double) hashsize)) | 1;     /* get ready to hash */
```

This code is used in section 1.

3. The CVM algorithm is wonderfully simple: We maintain a buffer of previously seen elements, and a probability p, such that each previously seen element independently appears in the buffer with probability p. Consequently the expected value of the current size of the buffer, divided by p, is the number of distinct elements seen.

The buffer always contains at most *bufsize* elements, except for brief periods of time. When it overflows, we randomly delete one of its elements and adjust p appropriately.

```
⟨Do the experiment 3⟩ ≡
  ⟨Set $p = 1$ and make the treap empty 9⟩;
  for ($m = 1$; $m \le length$; $m{+}{+}$) {
    ⟨Input $a$, the $m$th element of the stream 4⟩;
    ⟨Put $a$ into the treap with probability $p$ 12⟩;
    ⟨If the treap is overfull, purge its most volatile element and decrease $p$ 18⟩;
    if ($m \equiv length \lor \neg(m \,\%\, del)$) ⟨Print the current statistics 19⟩;
  }
```

This code is used in section 1.

4.
```
⟨Input $a$, the $m$th element of the stream 4⟩ ≡
  $a = gb\_unif\_rand(param)$;
  if ($m < 10$) fprintf(stderr, "a%d␣is␣%d\n", $m, a$);
  ⟨Insert $a$ into the hash table 6⟩;
```

This code is used in section 3.

5.
```
⟨Global variables 5⟩ ≡
  int hash[hashsize];      /* the hash table */
  int hashmult;      /* multiplier for the initial probe */
  int count;      /* this many distinct elements seen so far */
```

See also sections 8 and 20.

This code is used in section 1.

6. Ye olde linear probing. I assume that $a \ne$ #80000000. (If a does happen to have that value, it isn't counted.)

```
#define signbit #80000000      /* xored to a so that the result is nonzero */

⟨Insert $a$ into the hash table 6⟩ ≡
  for ($k = a \oplus signbit, h = (k * hashmult)\,\&\,(hashsize - 1)$; $hash[h]$; $h = (h \mathbin{?} h - 1 : hashsize - 1)$)
    if ($hash[h] \equiv k$) goto found;
  $count{+}{+}, hash[h] = k$;
  if ($count > (7 * hashsize)/8$) {
    fprintf(stderr, "Sorry,␣there␣are␣more␣than␣%d␣elements!\n", $(7 * hashsize)/8$);
    fprintf(stderr, "Recompile␣me␣with␣a␣larger␣hashsize.\n");
    exit(−6);
  }
found:
```

This code is used in section 4.

7. Treaps. I love the "treap" data structure, which was introduced by Jean Vuillemin under the name "Cartesian tree" in *Communications of the ACM* **23** (1980), 229–239, then renamed and extended by Cecilia Aragon and Raimund Seidel in *IEEE Symposium on Foundations of Computer Science* **30** (1989), 540–546. Indeed, it matches the CVM algorithm so nicely, I almost suspect that treaps were invented for precisely this application.

A treap is a binary tree whose nodes have two key fields. The primary key, which in our application is an element of the stream, obeys tree-search order: All descendants of the left child of node q have a primary key that is less than the primary key of q, and all descendants of its right child have a primary key that is greater. The secondary key, which in our application is a pseudorandom integer called the volatility of the element, obeys heap order: The secondary key of p's children is less than p's own secondary key.

A given set of nodes with distinct primary keys and distinct secondary keys can be made into a treap in exactly one way. This unique treap can be obtained, for example, by using ordinary tree insertion with respect to primary keys while inserting nodes in decreasing order of their secondary keys. It follows that the binary tree will almost always be quite well balanced, because the volatilities are uniformly random.

```
⟨ Type definitions 7 ⟩ ≡
  typedef struct node_struct {
    int elt;    /* identity of this element (the primary key) */
    int vol;    /* its volatility (the secondary key) */
    int left, right;    /* left and right children of this node, or −1 */
  } node;
```

This code is used in section 1.

8.
```
⟨ Global variables 5 ⟩ +≡
  node treap[maxsize];
  int avail;    /* top of the stack of unused nodes, or −1 */
  int root;    /* the node at the root of the treap, or −1 */
```

9. We maintain a list of unused nodes, beginning at *avail* and linked through the *left* pointers. Space is provided for up to $ssize = bufsize + 1$ nodes. The treap is "overfull" if and only if $avail = -1$, if and only if $treapcount/treapcountinc = ssize$.

```
⟨ Set p = 1 and make the treap empty 9 ⟩ ≡
  p = signbit;
  root = −1;    /* the treap is null */
  for (k = 0; k < ssize; k++) treap[k].left = k − 1;
  avail = k − 1;
```

This code is used in section 3.

10. Here's a routine that prints the top l levels of a given treap, for use when debugging. The nodes appear in preorder.

```
⟨ Subroutines 10 ⟩ ≡
  void print_treap(int r, int l)
  {
    if (r ≥ 0 ∧ l > 0) {
      fprintf(stderr, "%8x:␣%8x␣%8x␣%8x␣%8x\n", r, treap[r].vol, treap[r].elt, treap[r].left, treap[r].right);
      print_treap(treap[r].left, l − 1);
      print_treap(treap[r].right, l − 1);
    }
  }
```

See also section 11.

This code is used in section 1.

11. The algorithms for treap maintenance play fast and loose with pointers. So I'd better check that my implementation is sound, when I'm debugging this code or making changes.

```
#define sanity_checking 0   /* set this nonzero when you suspect a bug */
⟨Subroutines 10⟩ +≡
  int nodes_used;
  void treapcheck(int r, int prev)
  {
    int p, q;
    if (r ≥ 0) {
      nodes_used++;
      p = treap[r].left, q = treap[r].right;
      if (p ≥ 0) {   /* yes there's a left subtreap */
        if (treap[p].vol > treap[r].vol) fprintf(stderr, "heap␣order␣violated␣at␣node␣%d!\n", p);
        treapcheck(p, prev);
      }
      if (treap[r].elt < prev) fprintf(stderr, "element␣order␣violated␣at␣node␣%d!\n", r);
      if (q ≥ 0) {   /* yes there's a right subtreap */
        if (treap[q].vol > treap[r].vol) fprintf(stderr, "heap␣order␣violated␣at␣node␣%d!\n", q);
        treapcheck(q, treap[r].elt);
      }
    }
  }
  void treapsanity(void)
  {
    register int p;
    for (nodes_used = 0, p = avail; p ≥ 0; p = treap[p].left) nodes_used++;
    treapcheck(root, −1);
    if (nodes_used ≠ ssize) fprintf(stderr, "memory␣leak␣(only␣%d␣nodes␣used!\n", nodes_used);
  }
```

12. There's a nontrivial chance that two elements will have exactly the same volatility, because I'm using only 31-bit random numbers. It's less likely, but still possible, that two such elements will have one of the few volatilities that turn out to be relevant in this particular stream. So I print a warning on *stderr* when equality occurs. The treap will never contain any volatility $\ge p$.

```
⟨Put a into the treap with probability p 12⟩ ≡
  ⟨Remove a from the treap if it is present 13⟩;
  u = gb_next_rand();   /* uniformly random 31-bit number */
  if (u ≤ p) {
    if (u ≡ p) fprintf(stderr, "(discarded␣element␣%d␣of␣vol␣%08x␣at␣time␣%d)\n", a, u, m);
    else ⟨Insert a into the treap, with volatility u 16⟩;
  }
  if (sanity_checking) treapcheck(root, −1);
```

This code is used in section 3.

13. During this process, q is the current node and r tells us who pointed to q.

⟨ Remove a from the treap if it is present 13 ⟩ ≡

```
{
  for ($q = root, r = \sim ssize$; $q \ge 0$; ) {
    if ($treap[q].elt \equiv a$) break;      /* yes it's there */
    if ($treap[q].elt > a$) $r = \sim q, q = treap[q].left$;      /* move to left subtreap */
    else $r = q, q = treap[q].right$;      /* move to right subtreap */
  }
  if ($q \ge 0$) ⟨ Delete node $q$ 14 ⟩;
}
```

This code is used in section 12.

14. The deletion process essentially merges the two subtreaps that are children of the deleted node.

⟨ Delete node q 14 ⟩ ≡

```
{
  $l = treap[q].left, treap[q].left = avail, avail = q$;
  $treapcount = treapcount - treapcountinc$;
  $q = treap[q].right$;
  ⟨ Merge $l$ with $q$, storing the result as specified by $r$ 15 ⟩;
}
```

This code is used in sections 13 and 18.

15. ⟨ Merge l with q, storing the result as specified by r 15 ⟩ ≡

```
do {
  if ($l < 0$) $j = q, i = \sim ssize$;      /* $j$ denotes the result, $i$ denotes the next $r$ */
  else if ($q < 0$) $j = l, i = \sim ssize$;      /* it's easy to merge with an empty subtreap */
  else if ($treap[l].vol > treap[q].vol$) $j = l, i = l, l = treap[l].right$;
      /* left subtreap is the result, it retains its left subtreap */
  else $j = q, i = \sim q, q = treap[q].left$;      /* right subtreap is the result, it retains its right subtreap */
  if ($r \ge 0$) $treap[r].right = j$;
  else if ($r \equiv \sim ssize$) $root = j$;
  else $treap[\sim r].left = j$;
  $r = i$;
} while ($i \ne \sim ssize$);
```

This code is used in section 14.

16. ⟨ Insert a into the treap, with volatility u 16 ⟩ ≡

```
{
  $l = avail, avail = treap[l].left, treapcount = treapcount + treapcountinc$;
  $treap[l].elt = a, treap[l].vol = u$;
  for ($r = \sim ssize, q = root$; $q \ge 0 \land treap[q].vol > u$; ) {
    if ($treap[q].elt > a$) $r = \sim q, q = treap[q].left$;
    else $r = q, q = treap[q].right$;
  }
  if ($r \ge 0$) $treap[r].right = l$;
  else if ($r \equiv \sim ssize$) $root = l$;
  else $treap[\sim r].left = l$;
  ⟨ Create the subtreaps of $a$ by splitting the subtreaps of $q$ 17 ⟩;
}
```

This code is used in section 12.

17. At this point we must do the opposite of merging: The subtreaps whose elements just precede and follow a are gradually determined by splitting appropriate subtreaps of node q.

A formal proof of the slick code here would be a bit tricky, and probably difficult to verify. Maybe I'll have time some day to work out such a proof. Meanwhile I can understand it today by drawing pictures such as Fig. 25 in Section 6.2.3 of *Sorting and Searching*. Fortunately the computer doesn't slow down in this loop, like people do when they get to a conceptually harder part of an algorithm.

```
⟨Create the subtreaps of a by splitting the subtreaps of q 17⟩ ≡
  i = ∼l, j = l;    /* slots to fill in as we split at left and right of a */
  while (q ≥ 0) {
    if (a < treap[q].elt) {
      if (j ≥ 0) treap[j].right = q; else treap[∼j].left = q;
      j = ∼q, q = treap[q].left;
    } else {
      if (i ≥ 0) treap[i].right = q; else treap[∼i].left = q;
      i = q, q = treap[q].right;
    }
  }
  if (i ≥ 0) treap[i].right = −1; else treap[∼i].left = −1;
  if (j ≥ 0) treap[j].right = −1; else treap[∼j].left = −1;
```
This code is used in section 16.

18. See the note above, regarding elements from different times that have the same volatility by pure coincidence. If several elements of the tree have the maximum volatility, we must delete them all. Thus, if there are k such elements, we print $k-1$ warnings.

```
⟨If the treap is overfull, purge its most volatile element and decrease p 18⟩ ≡
  if (avail < 0) {
    p = treap[root].vol;
    r = ∼ssize, q = root;
    ⟨Delete node q 14⟩;
    while (root ≥ 0 ∧ treap[root].vol ≡ p) {
      fprintf(stderr, "(purged␣element␣%d␣of␣vol␣%08x␣at␣time␣%d)\n", treap[root].elt, p, m);
      r = ∼ssize, q = root;
      ⟨Delete node q 14⟩;
    }
    if (sanity_checking) treapsanity();
  }
```
This code is used in section 3.

19.
```
⟨Print the current statistics 19⟩ ≡
  {
    estimate = treapcount/((double) p + 0.5);    /* prevent division by 0 */
    ratio = estimate/(double) count;
    printf("%.4f%12d␣(%.4f),␣vol␣%08x␣after␣%d\n", estimate, count, ratio, p, m);
  }
```
This code is used in section 3.

20.
```
⟨Global variables 5⟩ +≡
  double treapcount;    /* the treap currently has this many nodes, times 2^−31 */
  double treapcountinc = (double) signbit;    /* 2^31 */
  double estimate;
  double ratio;
```

21. Index.

CVM-ESTIMATES