MONKEYSORT

K. B. Gallagher[*]
Department of Computer Science
University of Durham
South Road
Durham, DH1 3LE
United Kingdom
k.b.gallagher@durham.ac.uk www.durham.ac.uk/k.b.gallagher

Abstract

Monkeysort *is a pedagogical program that turns the usual concern of efficiency upside down by attempting to be as "dumb" as possible, yet still correct. In this program, whose inner workings are accessible to all students,* monkeysort *exhibits significant ideas that are central to computer science: partial correctness, generate-and-test solutions to NP-hard problems, Stirling's approximation of n!, and subtle applications of the use of permutations. It also demonstrates central software engineering ideas: integer overflow; the use of coverage tools; CPU monitoring tools; and timing analysis approaches more sophisticated than mere statement counting. An appropriately tailored classroom discussion of* monkeysort *can be used as fodder for graduate student homeworks, or to illustrate to the non-scientist exactly what it is that Computer Scientists do.*

1 INTRODUCTION

Teaching sophisticated computing ideas to lower division Computer Science Majors is a daunting task. While it seems that most students require concrete examples as an introduction to a concept, finding examples that are accessible to their current abilities is a challenge. Indeed, it is these ideas that attracted us to the discipline of computing, so finding such examples is paramount to instruct, and captivate, novices.

Partial correctness is one of these sophisticated ideas: *If* a program stops then it has solved the problem correctly, but *it may never stop.* The second captivating idea is the "generate-and-test" solution being the best known solution for NP-complete problems. To this end we present

[*]On professional leave from Loyola College in Maryland.

an accessible problem that illustrates partial correctness by using a generate-and-test solution. Actually, the solution is worse than generate-and-test for it merely guesses and tests; thus it is likely that a particular incorrect solution may be guessed and rejected more than once.

This paper presents what amounts to a lesson plan: a problem; an initial discussion of what the solution might look like; a solution; and a discussion of that particular implementation. The discussion can be tailored to the audience by omitting some of the more arcane points. I have used this discussion in courses that range from required general education courses for liberal arts students to graduate courses on software specification. Its themes are eminently suited to algorithm analysis courses at any level.

In the following discussion, please carefully note the use Computer Science jargon that should be adapted to the level of audience. For instance, below, I use “satisfies the postcondition” which makes sense to those of us in the discipline, but would only serve to befuddle the uninitiate. Here is a list of the some of the jargon used in this paper: *rearrangement, criteria, functional, implementation, specification, indices, satisfies the postcondition, algebraically, addresses, permutation, randomly, algorithm, entry condition, while-statement, partial correctness, assert(ion), conjunct(ion),* and *guard.* This list also serves as a reminder of how much jargon we casually use, and of the care we must take when talking to novices.

2 THE PROBLEM AND INITIAL DISCUSSION

The problem is the “usual suspect:” sorting an array of integers.

One way to discuss and define sorting an array of values is to view the result as a rearrangement of the input values such that the sorting criteria is satisfied. Rearrangement gives a nice functional view of the sorting process, while not focusing on a particular algorithm; i.e., rearrange the values so that they are in order without regard to a particular technique. So an early observation is that sorting is merely a (particular) rearrangement of the inputs that need not be tied to a particular implementation.

Then we jot down a specification: $\mathrm{a[i]} \leq \mathrm{a[i{+}1]}$, with suitable bounds on the indices. Noting that if only $\mathrm{a[i]} \leq \mathrm{a[i{+}1]}$ is required, then assigning 0 to every element satisfies the postcondition and leads to the simple observation that the input matters, so we had better say something about the inputs when we write the specification.

Algebraically, we can say that $\mathrm{a}[\pi(\mathrm{i})] \leq \mathrm{a}[\pi(\mathrm{i}+1)]$ for a permutation π of the array indices $1 \ldots \mathrm{n}$. In this formulation, the elements of the array are not swapped; the indices are reordered. To write the classical sort (i.e.; rearrange the elements, not the addresses), we would have to write something like $\mathrm{b} = \pi(\mathrm{a})$ and $\mathrm{b(i)} \leq \mathrm{b(i+1)}$.

With this in mind, a naive solution is to merely generate all possible permutations (ways of reordering) of the input and check to see if the permutation is sorted. Any permutation can viewed as a possible ordering of the indices that might be a solution. It is generating the permutations that is the genesis of this particular solution.

3 A SOLUTION

One way to generate permutations is to do it randomly. As a simple illustration, we envision sorting a deck of cards. The following is an unusual sorting technique, for it also requires a washtub and a broom and is as follows:

```
while (! deck-is-sorted)
   throw cards in tub
   stir with broom
   pick up deck
end
```

The students sense that this "algorithm" may take a long time to complete, and that has the *possibility* of going forever. I give this algorithm the name *monkeysort*. The name comes from the observation that if one places infinitely many monkeys in front of infinitely many typewriters, arbitrarily striking the keys, they will ultimately produce the works of Shakespeare, the Bible, the Oxford English dictionary, etc. as one of the random sequences. The trick of monkeysort, or the typing monkeys, is, of course, to be patient.

Next, we attempt another solution that at first blush seems better:

```
while (! deck-is-sorted)
   pick a card, any card
   pick a card, any card
   swap the two selected cards
end
```

While it is conceptually simpler to pick only two cards to swap, we will end up observing that there is no essential analytical difference between the 2 approaches. And having seen that, it doesn't really matter which one we implemented.

For both of the algorithms, noting that the negation of an entry condition of a while statement may be asserted as true when the loop ends, we observe that *if* this algorithm halts, it is correct. This example gives a gentle introduction to partial correctness and a nice example of reasoning about programs. Thus we can add `assert(deck-is-sorted)` at the end of each algorithm as a simple illustration of program reading.

I sometimes digress here to talk about this particular reading / comprehension technique. We note that if a `while` statement has multiple conjuncts for the entry condition, then the first thing that the software must do on exit is determine which of the clauses caused the exit.

```
while ( A & B & C & D)
  ...
  end
assert( !( A & B & C & D) )

if (!A) ....
elseif (!B) ...
elseif (!C) ...
else ...
```

While this may seem obvious to those of us who have been around software a long time, I have seen many good students nod with insight at this observation; this in turn leads to a discussion and clear thinking about what kind of conditions should be used to guard loops, and what kind of processing needs to be done after a loop completes.

4 THE PROGRAM

The *monkeysort* source is shown below. It implements the second version above: two elements are randomly selected and swapped. The program takes one command line argument, the number of random values to generate and sort. The output is the number of swaps that were made before the program halted. The program serves as an introductory example of how the vector class from the standard template library can be used. In addition, it shows how to seed the random number generator with the system time. Note the implementation is dumber than the pick-a-card version, for the same card can be selected twice and thus “swapped” with itself. (The drivers are the pair who typed the source into the classroom display device, while the backseat drivers critiqued.)

```
// Drivers:  Humaa & Patrick
// Backseat Drivers: John,
// Andrew(2), Rachel

#include<iostream>
#include<vector>
#include<time.h>

bool CheckSort(vector<int> & a)
{
  vector<int>::iterator j;
  for (j=a.begin(); j != a.end()-1; j++)
  {
    if (*j > *(j+1))
       return false;
  }
  return true;
}

void Transpose(vector<int> & a)
{
   unsigned long int  i, j, temp;
   i = (int) random() % a.size();
   j = (int) random() % a.size();
   temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}

main ( int argc, char* argv[])
{
  vector < int > a (size);
  srandom(time((time_t*)0));
  int i;
  int Count=0;

  if (argc != 2)
    {
      cout << "usage is: " << argv[0]
      << " number-of-elements-to-sort \n" ;
      exit (-1);
    }

  int size = strtol (argv[1],NULL,10);

  for(i=0;i<size; i++)  { a[i] =(int) random (); }

  while(!CheckSort(a))
   {
     Transpose(a);
      Count++;
   }

  cout << Count << endl;
}
```

5 EXECUTION

Now the fun begins. We run a *monkeysort* few times on very small values: 2, 3, and 4. For parameter 2 the results are uninteresting, except to note that the program does indeed halt. It also appears that about half the time that the random values are already in order. This data supports the observation if one generates 2 random numbers, about half the time they will be in order! We then write a shell script to run the program 100 times. Usually, the largest number of guesses that are needed is 5, which occurs about 2-3% of the time.

We now try the program with argument 3. The number of swaps increases to a maximum of about 35, but the program again halts. We also get 0 swaps, meaning that the numbers are generated in order. With argument 4, the maximum number of swaps is around 300. The program continues to halt. The processor speed is such that the return from the invocation is immediate.

The class now believes that the program will halt. We run it with inputs 5, 6 and 7. For input 7, we usually get a maximum number of swaps of about 60,000. We also observe that sometimes for larger values that the number of swaps may indeed be less than the number of swaps for a smaller array. Table 1 shows some data collected for this paper.

For input 8 the outputs range from 9 to about 440,000. Occasionally there is a short pause before the program halts, but halt it does. For input 9 the outputs range from about 30 to 4.3 *million*. During the runs when the larger values result, the pause is long enough for someone to wonder out loud whether or not it will ever halt. But it does. For input 10, the results range from about 700 to about 39,000,000. The pauses are getting longer…For 11, the pauses are running to minutes, usually about 1-2, and as high as 6, but again, always halting. For 12, it seems that we have run into a non-halting version; it just turns out, though, that 500 million swaps take a long time.

6 DISCUSSION POINTS

This section notes some of the issues that can be discussed while *monkeysort* is grinding away.

6.1 GUESSING, PERMUTATIONS AND FUNCTIONS

One of the first questions to pose is "*About* how many guesses would one expect?" Since there are $n!$ permutations, in a strict enumeration of the solution, one might expect about $n!/2$ guesses. Table 1 shows that the average number of guesses is more than $n!$, but this is to be expected, as *monkeysort* permits multiple guesses of the same permutation. Table 2 suggests that the limit of the average number of guesses *might* be approaching $n!$, and that the maximum *seems* to be about $10 * n!$. Graduate students in algorithm analysis can be given these for homework exercises.

At this point one can also discuss a permutation as a product of transpositions; that is, every cycle $(i_1 i_2 \dots i_n)$ can be written as $(i_1 i_n)(i_1 i_{n-1}) \dots (i_1 i_2)$. Since every permutation can be written as a product of disjoint cycles, and thus a product of transpositions, *monkeysort* as written is not "dumber" than another version that would generate random n-ary permutations. It is "dumber" because it can generate a sequence of identity transpositions; and it can repeatedly generate a previously guessed solution. The follow up question is it appropriate then, to count each transposition as a permutation? Shouldn't we group them into collections that are *input_size*d? That is, should we divide the number of transpositions by the input size to get the true number of attempts for *monkeysort*?

As usual, we give the scholar's answer of "that depends…" The key to counting *monkeysort*'s iterations is to note that each new transposition is applied to the result of the previous transposition, and not to the original ordering. Applying single transpositions to the original input and then returning to the original state would never sort the array. Applying a single transposition to the result of the previous application is central to the "correctness" of *monkeysort*. Moreover, since every permutation can be viewed as having the same length (by including the identity mappings in each case) we can view *monkeysort* as building one permutation up (using the previous result) and applying that mapping to the original. This is a nice application of the concept of composition of functions.

input	*min*	*avg*	*max*
4	0	33	298
5	0	164	1440
6	0	945	9144
7	0	6186	64145
8	9	47937	443701
9	28	422700	4396274
10	690	4144904	38585752
11	175	45168348	389202850
12	237627	539952999	5040709046
13	9749933	6291915902	40877452599

Table 1: The minimum, average and maximum number of guesses on 10,000 trials for inputs of 4-11; 1000 trials for 12; and 525 trials for input 13.

Another interesting point is that the expected minimum of 0 is not obtained in every instance. To put it another way, one would expect that in rows 8 and higher of Table 1 that there would be at least one instance in which the values were generated in order. Ten thousand random sequences were produced and *not one* was already sorted! The reasonableness or unreasonableness of this observation is another algorithm analysis homework.

input	*avg / n!*	*max / n!*
4	1.333	12.42
5	1.358	12.00
6	1.311	12.70
7	1.288	12.73
8	1.188	11.00
9	1.165	12.12
10	1.142	10.63
11	1.131	9.75
12	1.127	10.52
13	1.001	6.56

Table 2: average / n! and max / n! for inputs 4-13.

One way to entice students to the Computer Science is to discuss the NP-complete problems: those whose *best* and *known* solutions are the equivalent of *monkeysort*. This particular solution is NP. There are "better" ones, of course. So I then ask: Have I put myself in the Computer Science history books by showing that $P = NP$? The ensuing discussion gives a glimpse at what Computer Scientists do; i.e., we don't write programs all day.

Another way to look at this is to consider how difficult sorting would be if *monkeysort* were the *only* known way to sort. What would that mean to us as scientists? This question can be viewed in a larger context also: how long would it take to produce a telephone book if the names could only be arranged via *monkeysort*? How often do we depend upon the sortedness of data in our everyday lives? Then the difficulty of the garbage truck and other NP-hard problems becomes manifestly evident.

A final point to make regards guessing versus enumeration. We could systematically list all of the arrangements (or perhaps assign it as homework?) and test our enumerations for order via `CheckSort`. How much time does this really save? Depending on the level of the audience, one can wade into a discussion of the how and why of constants not mattering in *big-O* notation. This leads naturally to a discussion of Stirling's approximation.

6.2 STIRLING'S APPROXIMATION

Stirling's Approximation, also called Stirling's Formula, shown below, which is an approximation to $n!$, is a remarkable mathematical achievement. While the derivation of the approximation is not in the scope of an elementary programming course, it does present a glimpse of how mathematics is used in our work. And advanced Computer Science students can indeed understand the derivation.

$$n! \approx \sqrt{2\pi e}\left(\frac{n}{e}\right)^n$$

Stirling's Approximation gives us an estimate of how many permutations of n there are. We simply note

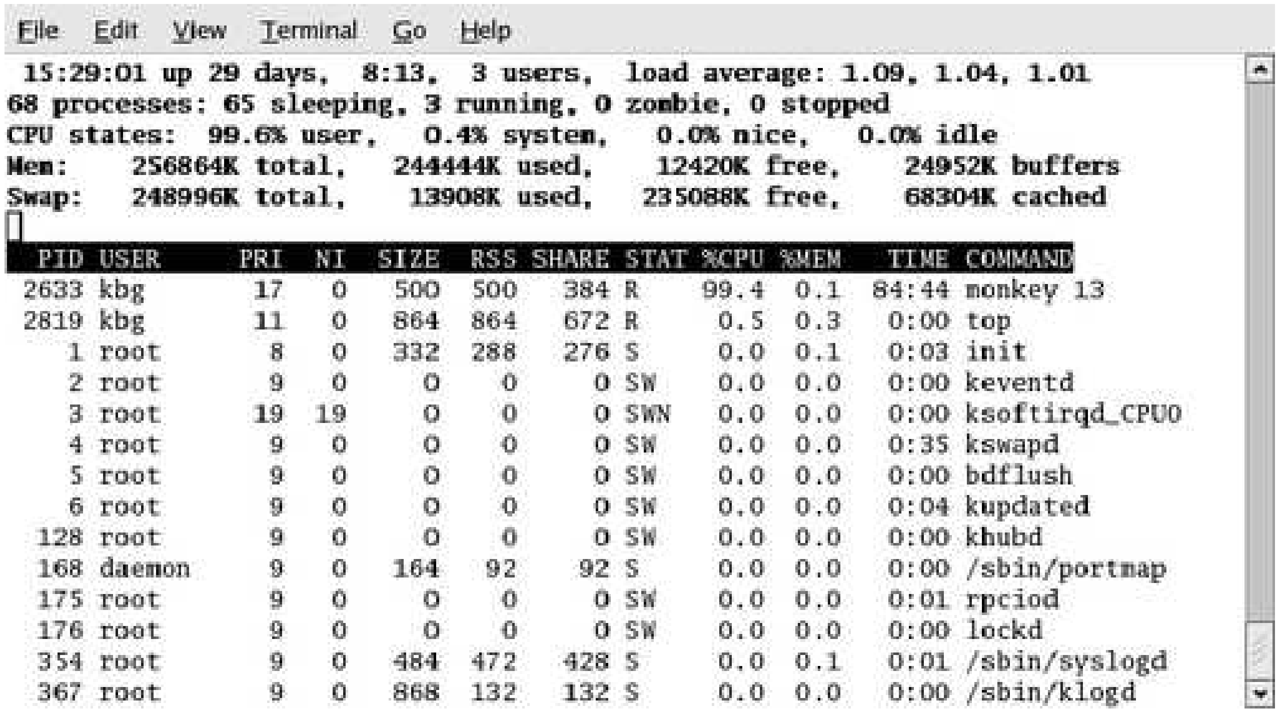
$$n! \approx \sqrt{2\pi e}\left(\frac{n}{e}\right)^n \text{ is } O(n^n)$$

as an instance of *big-O* notation. Thus Stirling's approximation suggests that there will be about n^n swaps, on average, before we get the sorted one. There is a problem: the data that we are collecting shows that the average seems to be heading toward $n!$, not n^n. Again, depending on the level of the audience, I can discuss relative and absolute error estimates of the approximation, and further emphasize that the numbers that we are using are truly small.

6.3 TOP

I open the GNU/Linux utility *top* in another window. From the *man* page we see that "*top* provides an ongoing look at processor activity in real time. It displays a listing of the most CPU-intensive tasks on the system..." In the new window, we observe upward of 99% of the CPU being used to run *monkeysort*, as displayed in Figure 1.

If time permits, or inclination suggests, one can digress into a discussion of the processes with single digit process id's: `init`, `keventd`, `ksoftirqd_CPU0`, `kwsapd`, `bdflush`, and `kupdated`. Some of these commands have man pages, others do not. Finding what they all do can be left to enterprising students.



```
 15:29:01 up 29 days,  8:13,  3 users,  load average: 1.09, 1.04, 1.01
68 processes: 65 sleeping, 3 running, 0 zombie, 0 stopped
CPU states:  99.6% user,   0.4% system,   0.0% nice,   0.0% idle
Mem:    256864K total,   244444K used,    12420K free,    24952K buffers
Swap:   248996K total,    13908K used,   235088K free,    68304K cached

  PID USER       PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 2633 kbg         17   0   500  500   384 R    99.4  0.1  84:44 monkey 13
 2819 kbg         11   0   864  864   672 R     0.5  0.3   0:00 top
    1 root         8   0   332  288   276 S     0.0  0.1   0:03 init
    2 root         9   0     0    0     0 SW    0.0  0.0   0:00 keventd
    3 root        19  19     0    0     0 SWN   0.0  0.0   0:00 ksoftirqd_CPU0
    4 root         9   0     0    0     0 SW    0.0  0.0   0:35 kswapd
    5 root         9   0     0    0     0 SW    0.0  0.0   0:00 bdflush
    6 root         9   0     0    0     0 SW    0.0  0.0   0:04 kupdated
  128 root         9   0     0    0     0 SW    0.0  0.0   0:00 khubd
  168 daemon       9   0   164   92    92 S     0.0  0.0   0:00 /sbin/portmap
  175 root         9   0     0    0     0 SW    0.0  0.0   0:01 rpciod
  176 root         9   0     0    0     0 SW    0.0  0.0   0:00 lockd
  354 root         9   0   484  472   428 S     0.0  0.1   0:01 /sbin/syslogd
  367 root         9   0   868  132   132 S     0.0  0.0   0:00 /sbin/klogd
```

Figure 1: Output of `top` utility. Observe the Heisenberg effect of `top` on the CPU utilization.

6.4 COUNTING, OVERFLOW AND TIMING

On the day that this program is introduced, there is no time to show integer overflow; it does occur, however. On the machine that I use in class it takes about an hour. On days following the introduction of *monkeysort*, I will start it up with a parameter of 13 or higher just as class starts. On days that we are lucky, we get a negative integer for the output just as class ends! Someone notices or I point out that the `Count` variable is merely an `int`. This observation leads to a nice discussion (in a subsequent class) about integer overflow and other integral data types: `long`, `unsigned`, `unsigned long` and `unsigned long long`. I use this as an opportunity to inspect the contents of `/usr/include/limit.h`.

In the meantime I have run *monkeysort* 100 times with input 13 as a background job, and collected execution time data in a file using the Unix `time` command. What we end up seeing is that about 25% of the time the output of *monkeysort* overflows as indicated by a negative number. A more careful inspection of the collected timing and count data shows that an invocation of *monkeysort* that took almost 5 hours to complete has a smaller output than one that took 2 hours to complete! The phenomena of wrap-around becomes obvious, and we see that our “computer-generated” data is worthless. I suggest changing `Count` to `unsigned`; the students note that this buys only another hour. Similar discussion ensues regarding other data types. A student suggests using `float`s. Another points out that the overflow problem doesn’t go away. We look at `MAXFLT`; then write a program that prints out `MAXFLT - 1`. The values are the same. I point out that an enterprising student might use the *BigInt* package, or something like it, that permits unbounded integers (at a cost, of course).

It turns out that counting using the built-in arithmetic is no way to measure *monkeysort*. Use of the `time` command improves our analysis. How do we use it? Off we go to the `man` pages. Is wall clock time good enough? I run two *monkeysort*s on the same machine at the same time. Clearly wall clock time is insufficient. Thankfully, the `time` command also permits us to measure computation time in addition to wall clock time.

To correctly use the output of `time` as a measure, we must first calibrate the time used to the number of `Transpose(a)` calls. Merely running a counter over an empty loop is not enough; we have to assure that an "appropriate" number of calls to `CheckSort(a)` and `Transpose(a)` are considered. Since we use the standard template library, this timing data is especially crucial. Thus, we "trick" `CheckSort(a)` into always running halfway down the array, the average number of comparisons it will make before it returns. We still need to attend to the time of the calls to `Transpose(a)`. Rather than depend on the return value of `CheckSort(a)`, we predetermine how many times we will need to run a loop that calls both `Transpose(a)` and `CheckSort(a)`. That is, we write a program that has calls to `Transpose(a)` and `CheckSort(a)` in the body of a `for` statement, with input as the number of iterations. This program is the one we time to get the calibration. We can then time the execution of this program a number of times (with the same input) to get a more confident estimate of the number of swaps in relation to the time taken. Then, and only then, can we use the output of the `time` command as an *approximation* to the number of swaps that were required to order the array. Note how much work is required because computers can't count over a certain value!

6.5 GCOV

The GNU/Linux test coverage program, `gcov`, also gives some insight into the inner workings of *monkeysort*. `Gcov` is a simple statement counting analyzer. With appropriate compilation switches, the code gets instrumented; after the program is run, it is then post-processed by `gcov`, which produces, in its simplest version, a listing of the program with each statement preceded by the number of times it was executed. Below is a sample of `gcov` output when *monkeysort* is run with input 10. This version, which has `quicksort` and `bubblesort` routines, invokes the *monkeysort* approach when a third parameter is sent. (The particular value of the parameter is immaterial.) The listing gives nice empirical data for the "efficacy" of *monkeysort*. When the third parameter is not sent, only `quicksort` and `bubblesort` are invoked. With a larger input parameter, students can *see* the difference between $O(n^2)$ and $O(n * ln(n))$.

```
// CS302.01  multiple sorts for gcov:
#include<iostream>
#include<vector>
#include<time.h>

          bool CheckSort(vector<int> & a)
 661756    { vector<int>::iterator j;
 661756      for (j=a.begin(); j != a.end()-1; j++)
              {
1158115        if (*j > *(j+1))
```

```
 661755          return false;
 496360      }
      1     return true;
           }

           void Transpose(vector<int> & a)
661755     {  unsigned long int  i, j, temp;
661755        i = (int) random() % a.size();
661755        j = (int) random() % a.size();
661755        temp = a[i];
661755        a[i] = a[j];
661755        a[j] = temp;
            }

           void qsort(vector<int> & v, int left, int right)
    17     { int i, last, t;
    17       if (left >= right) return ;
     8       t = v[right];
     8       v[right] = v[left];
     8       v[left] = t;
     8       last = left;
     8       for( i  = left +1; i <=right; i++)
    28         if (v[i] <  v[left] )
    14           {
    14            t = v[i];
    14            v[i] = v[++last];;
    14            v[last] = t;
                 }
    28        t = v[left];
    28        v[left] = v[last];
    28        v[last] = t;

     8        qsort(v, left, last - 1);
     8        qsort(v, last+1, right);
             }

           void bsort(vector<int> & v, int size)
     1      {
     1       for (int ii = 0; ii < size ; ii++)
               {
    10           for( int jj = 0 ; jj < size ;  jj ++)
                  {
   100              if ( v[jj] > v[ii])
```

```
    21               {
    21                int t = v[jj] ;
    21                v[jj] = v[ii] ;
    21                v[ii] = v[jj] ;
                     }
   100            }
    10         }

           }
          main ( int argc, char* argv[])
     1    {
     1      int size = atoi (argv[1]);
     1      vector < int > orig (size);
     1      vector < int > bubble (size);
     1      vector < int > quick (size);
     1      vector < int > monkey (size);

     1      srandom(time((time_t*)0));
            int i;

     1      for(i=0;i<size; i++){
    10        orig[i] = (int) random ();
    10      }

     1      bubble = quick = monkey = orig;

     1      bsort( bubble, size);
     1      qsort( quick, 0, size);

     1      if (argc == 3)
     1        {
     1          int Count=0;
     1          while(!CheckSort(monkey))
                  {
661755              Transpose(monkey);
661755               Count++;
                  }
     1         cout << Count << endl;
              }
            }
```

The demonstration of gcov leads to other relevant observations. The first is that programs should be *empirically* analyzed for “hot spots.” Then the 80-20 epigram can be invoked: 20% of the program does 80% of the work. It is in this 20% that efficiency machinations should be done, if at all. Another application is in a later software engineering course, in which I can insist on at least statement coverage for the project code, as evidenced by `gcov`. And lastly, if not finally, I can digress into the interesting intellectual challenges in the field of software testing.

7 CONCLUSION

Monkeysort turns out to be an interesting piece of software; indeed, a colleague, during a coffee break conversation at a conference, convinced me it was worth reporting to the community. I had always considered it a throw-away idea. My students have always liked it though; perhaps I missed the subtle hint.

We are often caught up in trying to make our software more "efficient," a vestige of days when both CPU cycles and memory were expensive. Since they are cheap now (think of the "wasted" CPU cycles in any organization in one day) we need not consider efficiency until it is an engineering imperative.

Monkeysort turns the efficiency question on its head by trying to be as "dumb" possible, yet still correct. In this explicit attempt to be dumb, significant ideas emerge.

8 ACKNOWLEDGMENTS

The presentation of this paper was improved immeasurably by the comments of John Monroe and the referees.

References