SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 29(0), 1–4 (0 1999)

A Killer Adversary for Quicksort

M. D. MCILROY
Dartmouth College, Hanover, NH 03755, USA

SUMMARY

Quicksort can be made to go quadratic by constructing input on the fly in response to the sequence of items compared. The technique is illustrated by a specific adversary for the standard C qsort function. The general method works against any implementation of quicksort–even a randomizing one–that satisfies certain very mild and realistic assumptions.

When using quicksort one often feels a nagging tension: suppose it goes quadratic? Tactics to avoid embarrassing results in some low-entropy cases, such as already ordered input, are cited in most algorithms books[1,2] and are widely used. Nevertheless, production implementations have been caught going quadratic in real-life applications.[3] No matter how hard implementers try, they cannot (without great sacrifice of speed) defend against all inputs. This note describes an adversarial method that finds chinks in the defenses of any implementation.

A polymorphic implementation of quicksort, such as the standard C function `qsort`, never looks at the data. It relies instead on an externally supplied comparison function. And that allows us to monitor and influence the program's progress noninvasively. To do so we make a comparison function that observes the pattern of comparisons and constructs adverse data on the fly.

Recall that quicksort sorts a sequence of n data items in three phases:

1. Pick a data item as pivot. We assume that this phase uses $O(1)$ comparisons.
2. Partition the data into three parts that respectively contain all items less than the pivot, the pivot item itself, and all items greater than the pivot. The placement of items equal to the pivot varies among implementations.
3. Recursively sort the low and high parts.

An adversary can make such a quicksort go quadratic by arranging for the pivot to compare low against almost all items not seen during pivot selection, so the partition will be lopsided. Those items may be regarded as a "gas" of values whose relationship to each other is unknown. The exact values don't matter as long as they are not compared against each other. Quadratic behavior is guaranteed since $n - O(1)$ gas values must survive pivot selection among n items. Almost all partition high.

Initially the adversary makes all items gas. When two gas items are compared, one gets "frozen" into a definite "solid" value, greater than any already solid value. Then the operands are compared afresh. When a solid item is compared to a gas item, it compares low. When two solid items are compared, the answer depends on the frozen values.

The essential trick is to make sure that the pivot gets frozen early in the partition phase if it has not already been frozen. No further gas items will become frozen as long as the pivot is involved in every comparison–that is, for the duration of the partitioning phase.

A simple heuristic suffices to guess the pivot and freeze it. A "pivot candidate" is the gas item that most recently survived a comparison. When an item is to be frozen (in a gas-gas comparison) a pivot

CCC 0038–0644/99/000001–04 *Received 1998*
 Revised 1998

candidate is preferred. While there may be no useful pivot candidate at the start of the partition phase, one will emerge as soon as a gas item is examined. If the pivot is already solid, the candidate doesn't matter. Otherwise, the first gas-gas comparison in the partition phase results in the pivot either getting frozen or becoming the pivot candidate. In the worst case the pivot will become frozen at the second gas-gas comparison in the partitioning phase. With at most two items getting frozen during partitioning, we are still assured that $n - O(1)$ items will partition high.

To defend against the possibility of the subject quicksort working on copied data that may not see changes that the adversary makes, we let the subject sort pointers to immovable items instead of the items themselves. The subject and the adversary are protected from each other because the subject works on pointers while the adversary works on items. When the pointers have finally been rearranged into sorted order, the array of items holds the constructed input that drove quicksort quadratic.

The adversarial method works for almost any polymorphic program recognizable as quicksort. The subject quicksort may copy values at will, or work with lists rather than arrays. It may even pick the pivot at random. The quicksort will be vulnerable provided only that it satisfies some mild assumptions that are met by every implementation I have seen:

1. The implementation is single-threaded.
2. Pivot-choosing takes $O(1)$ comparisons; all other comparisons are for partitioning.
3. The comparisons of the partitioning phase are contiguous and involve the pivot value.
4. The only data operations performed are comparison and copying.
5. Comparisons involve only input data values or copies thereof.

An invasive version of the method may be used when the caller does not control the comparison function, by bugging the comparison steps in quicksort itself.

The Appendix shows an adversary for C's `qsort`. The function `antiqsort(n, a)` constructs in array a a bad permutation of $0..n-1$ and returns the number of comparisons `qsort` took to sort it. Gas is coded as the top value, $n-1$, which ultimately persists in the single item that survives the last gas-gas comparison. The pointers described above are realized as indexes into an array of item values. In accordance with the C standard, the arguments of the comparison function are C pointers to these index "pointers".

The adversary is effective, as Table 1 shows. Comparison counts $C(n)$ were measured at single values of n where $C(n) > 10^7$, with quadratic behavior confirmed by the truth of $C(n) > 3.99C(n/2)$. Remember that the table describes the effectiveness of a particular adversary, not the ultimate worst-case behavior of the implementations.

Table 1. Performance of `antiqsort` against various `qsort`s.

Implementation	Pivot choice	Approximate count
Digital Unix 4.0	Arbitrary	$0.25n^2$ (exact)
Irix 6.4	Median of three	$0.25n^2$
Bentley	Median of medians	$0.088n^2$
gcc 2.7 (Windows)	?	$0.097n^2$

Against the `qsort` in Digital Unix `antiqsort` generates inputs exemplified by Figure 1. These inputs force exactly $\lceil n^2/4 \rceil$ comparisons for $n > 3$. This `qsort` chooses a pivot arbitrarily as the middle item in the array. By luck, it causes the adversary to freeze two items per partition–the maximum possible. Thus the size of the high side of the partition decreases by two at each recursion level. The recursion continues right down to $n = 1$. If n is even, the total number of comparisons at all levels is $(n-1) + (n-3) + \ldots + 1$, a sum of odd numbers and hence a perfect square, as observed. An

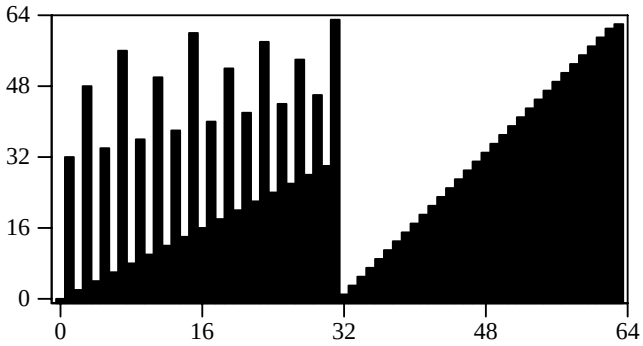

Figure 1. A 64-item adverse input for Digital Unix `qsort`. An end effect sets the order of items 31 and 63. Pivots, with odd values, form the right staircase. The remaining items, with even values, form 5 interleaved staircases with steps at $2^k - 1 \bmod 2^{k+1}$, for $k = 0..4$. Each item in staircase k gets swapped with k different pivots during sorting.

unlucky implementation would cause only one item to be frozen at each level, giving almost twice as many comparisons. Any other arbitrary pivot choice, such as the first item or a random item, would yield similar results.

Somewhat surprisingly, Table 1 shows a median-of-three quicksort[2] doing no better than the (lucky) arbitrary-choice quicksort. To see why, notice that two items get frozen during an optimum computation of the median of three. One partitions low and the other is the median. Hence the size of the high side decreases by two at each level, as with arbitrary choice. The median calculation is wasted effort.

The adversary cannot push the comparison count of Bentley's `qsort`[3] as high, because that program considers–and freezes–more items in choosing a pivot. When all considered items are gas, as is likely, the high side of the partition shrinks by six per recursion level. Thus the comparison count should be nearly a factor of 3 less than that for a median-of-three choice. Table 1 confirms this prediction.

The adversary is highly specific to quicksort. Against an insertion sort it did as badly as possible, forcing only $n - 1$ comparisons to sort n items.

I thank Jon Bentley for critical reading and the referees for prompting the presentation of results.

REFERENCES

1. Jeffrey H. Kingston, *Algorithms and Data Structures: Design, Correctness, Analysis*, Addison-Wesley, 1990.
2. R. Sedgewick, *Algorithms in C++*, Addison-Wesley, 1992.
3. J. L. Bentley and M. D. McIlroy, 'Engineering a sort function', *Software–Practice and Experience*, **23**, 1249–1265 (1993).

Appendix. An adversary for `qsort`.

```
#include <stdlib.h>

int     *val;                           /* item values */
int     ncmp;                           /* number of comparisons */
int     nsolid;                         /* number of solid items */
int     candidate;                      /* pivot candidate */
int     gas;                            /* gas value */

#define freeze(x) val[x] = nsolid++

int cmp(const void *px, const void *py)  /* per C standard */
{
        const int x = *(const int*)px;
        const int y = *(const int*)py;
        ncmp++;
        if(val[x]==gas && val[y]==gas)
                if(x == candidate)
                        freeze(x);
                else
                        freeze(y);
        if(val[x] == gas)
                candidate = x;
        else if(val[y] == gas)
                candidate = y;
        return val[x] - val[y];          /* only the sign matters */
}

int antiqsort(int n, int *a)
{
        int i;
        int *ptr = malloc(n*sizeof(*ptr));
        val = a;
        gas = n - 1;
        nsolid = ncmp = candidate = 0;
        for(i=0; i<n; i++) {
                ptr[i] = i;
                val[i] = gas;
        }
        qsort(ptr, n, sizeof(*ptr), cmp);
        free(ptr);
        return ncmp;
}
```