TovaSort: A Quick Analysis
Elena Izotova
25 June 2018

For the purposes of this analysis the modified QuickSort will be referred to as TovaSort and both TovaSort and Vanilla QuickSort will be analyzed in terms of arrays of ints as they appear in C.

A Quick Look at QuickSort

QuickSort is a recursive sorting method that sorts arrays using the following steps:

1. Selects 3 indexes: a pivot (typically the first index), a left index, and a right index
2. Left sweeps through the array until a value is larger than the value at the pivot index
3. Right sweeps through the array until a value is smaller than the value at the pivot index
4. Swaps the values at the left and right indexes
5. Repeat step 2-4 until the left and right indexes meet
6. Swap the values at the pivot index and either the right or left index
7. QuickSort the subarrays at the left and right of the pivot index

TovaSort

TovaSort (TS) is an efficient sorting method, serving as a systematic approach to placing the elements of an array in order. It improves on the vanilla QuickSort (QS) method in two ways: by removing the dependence on a pivot index, and by left and right sweeping through the array in such a way that the array is not required to be split into subarrays of one.

The method works as follows:

1. Find a reference value as one of the members of the array, typically the middle value.
2. Left sweep through the array until a value greater than the reference value is found.
3. Right sweep through the array until a value less than the reference value is found.
4. Swap the two elements of the array in place.
5. Continue this process until the left sweeping index crosses over the right sweeping index.
6. If the right sweeping index is greater than the given starting index, use the TovaSort method from the starting index to the right index.
7. If the left sweeping index is less than the given end index, use the TovaSort method from the left index to the end index.

The method can be used for sorting collections, arrays, matrices, or any groups of objects or elements with at least a single comparable attribute.

QuickSort vs. TovaSort

Multiple variables were tested for each of the sorts. These test cases and variables will be referred to under the following definitions:

Random:

```
for (i = 0; i < SIZE; i++)
{
    numbers[i] = rand();
}
```

In an array (numbers) of size SIZE, the array is filled with numbers using the rand() function. In C, the rand() function returns a pseudo-random number between 0 and RAND_MAX, which is typically defined to be within 15 bits. This is effective for filling arrays of smaller sizes, but when SIZE is defined to be 10 million, this method produces duplicate values.

Modified Random:

```
for (i = 0; i < SIZE; i++)
{
    numbers[i] = (rand() << 15) + rand();
}
```

As opposed to filling an array using the Random method, this method makes duplicate values much less likely by adding a rand() produced number to a number guaranteed to be greater than RAND_MAX.

a_count:
The number of times a sort performs an assignment
c_count:
The number of times a sort compares two values
r_count:
The number of times a sort makes a recursive call

Arrays filled with Random elements:

10 million elements:

```
Number of elements: 10000000
Vanilla Quick Sort:
****
Done
Time Spent = 5.972000 seconds
****
Done
Time Spent = 5.987000 seconds
****
Done
Time Spent = 5.947000 seconds
****
Done
Time Spent = 5.965000 seconds
```

```
Number of elements: 10000000
Tova Sort:
****
Done
Time Spent = 1.504000 seconds
****
Done
Time Spent = 1.510000 seconds
****
Done
Time Spent = 1.502000 seconds
****
Done
Time Spent = 1.508000 seconds
```

1 million elements:

```
Number of elements: 1000000
Vanilla Quick Sort:
****
Done
Time Spent = 0.239000 seconds
r_count = 1934464
a_count = 52978868
c_count = 82936938
****
Done
Time Spent = 0.235000 seconds
r_count = 1934464
a_count = 51804376
c_count = 81867827
****
Done
Time Spent = 0.232000 seconds
r_count = 1934464
a_count = 51331056
c_count = 81058166
****
Done
Time Spent = 0.230000 seconds
r_count = 1934464
a_count = 51607144
c_count = 80390199
****
```

```
Number of elements: 1000000
Tova Sort:
****
Done
Time Spent = 0.177000 seconds
r_count = 746507
a_count = 43836810
c_count = 39844590
****
Done
Time Spent = 0.156000 seconds
r_count = 745990
a_count = 42939346
c_count = 38903031
****
Done
Time Spent = 0.155000 seconds
r_count = 746968
a_count = 42706971
c_count = 38642792
****
Done
Time Spent = 0.154000 seconds
r_count = 746583
a_count = 43296004
c_count = 39283059
****
```

Arrays filled with Modified Random elements:

10 million elements:

```
Number of elements: 10000000
Vanilla Quick Sort:
****
Done
Time Spent = 2.232000 seconds
****
Done
Time Spent = 2.218000 seconds
****
Done
Time Spent = 2.224000 seconds
****
Done
Time Spent = 2.234000 seconds
```

```
Number of elements: 10000000
Tova Sort:
****
Done
Time Spent = 2.050000 seconds
****
Done
Time Spent = 2.029000 seconds
****
Done
Time Spent = 2.036000 seconds
****
Done
Time Spent = 2.023000 seconds
```

1 million elements:

```
Number of elements: 1000000
Vanilla Quick Sort:
****
Done
Time Spent = 0.224000 seconds
r_count = 1333640
a_count = 44194180
c_count = 61046043
****
Done
Time Spent = 0.207000 seconds
r_count = 1332544
a_count = 41342157
c_count = 58759429
****
Done
Time Spent = 0.211000 seconds
r_count = 1333402
a_count = 40854712
c_count = 58347956
****
Done
Time Spent = 0.208000 seconds
r_count = 1333382
a_count = 41278378
c_count = 58701891
****
```

```
Number of elements: 1000000
Tova Sort:
****
Done
Time Spent = 0.194000 seconds
r_count = 890154
a_count = 42502663
c_count = 40404210
****
Done
Time Spent = 0.181000 seconds
r_count = 890447
a_count = 42526884
c_count = 40444383
****
Done
Time Spent = 0.184000 seconds
r_count = 890477
a_count = 42561598
c_count = 40482816
****
Done
Time Spent = 0.181000 seconds
r_count = 890588
a_count = 44287061
c_count = 42279545
****
```

These tests demonstrate a few areas in which TS is more efficient than QS:

- TS performs consistently quicker than QS regardless of size and method used to fill the array.
- QS struggles in handling duplicate values, whereas TS handles arrays with duplicates faster than arrays with less or no duplicate values.
- TS makes as few as approximately 40% of the recursive calls that QS makes due to not depending on subarrays of size one.
- TS makes less comparisons in both Random arrays and Modified Random arrays, and performs less assignments in Random arrays.

Comparison Costs:

Although QS and TS have roughly the same best and worst case comparison costs, $nlog(n)$ and n^2 respectively, the worst case for TS is more difficult to construct than the worst case for QS.