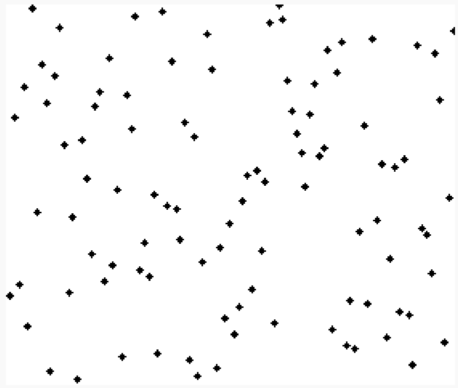
Odd–even sort

From Wikipedia, the free encyclopedia
(Redirected from Odd-even sort)

In computing, an **odd–even sort** or **odd–even transposition sort** (also known as **brick sort**[1][*self-published source*]) is a relatively simple sorting algorithm, developed originally for use on parallel processors with local interconnections. It is a comparison sort related to bubble sort, with which it shares many characteristics. It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

Odd–even sort



Example of odd-even transposition sort sorting a list of random numbers.

Class	Sorting algorithm
Data structure	Array
Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Worst case space complexity	$O(1)$

Contents [hide]

Sorting on processor arrays [edit]

On parallel processors, with one value per processor and only local left–right neighbor connections, the processors all concurrently do a compare–exchange operation with their neighbors, alternating between odd–even and even–odd pairings. This algorithm was originally presented, and shown to be efficient on such processors, by Habermann in 1972.[2]

The algorithm extends efficiently to the case of multiple items per processor. In the Baudet–Stevenson odd–even merge-splitting algorithm, each processor sorts its own sublist at each step, using any efficient sort algorithm, and then performs a merge splitting, or transposition–merge, operation with its neighbor, with neighbor pairing alternating between odd–even and even–odd on each step.[3]

Batcher's odd–even mergesort [edit]

A related but more efficient sort algorithm is the Batcher odd–even mergesort, using compare–exchange operations and perfect-shuffle operations.[4] Batcher's method is efficient on parallel processors with long-range connections.[5]

Algorithm [edit]

The single-processor algorithm, like bubblesort, is simple but not very efficient. Here a zero-based index is assumed:

```
function oddEvenSort(list) {
  function swap( list, i, j ){
    var temp = list[i];
    list[i] = list[j];
    list[j] = temp;
  }

  var sorted = false;
  while(!sorted)
  {
    sorted = true;
```

```
    for(var i = 1; i < list.length-1; i += 2)
    {
      if(list[i] > list[i+1])
      {
        swap(list, i, i+1);
        sorted = false;
      }
    }

    for(var i = 0; i < list.length-1; i += 2)
    {
      if(list[i] > list[i+1])
      {
        swap(list, i, i+1);
        sorted = false;
      }
    }
  }
}
```

This is an example of the algorithm in c++

```
 template <class T>
void OddEvenSort (T a[], int n)
{
    for (int i = 0 ; i < n ; i++)
    {
         if (i & 1) // 'i' is odd
         {
             for (int j = 2 ; j < n ; j += 2)
             {
                  if (a[j] < a[j-1])
                      swap (a[j-1], a[j]) ;
             }
          }
          else
          {
              for (int j = 1 ; j < n ; j += 2)
              {
                   if (a[j] < a[j-1])
                       swap (a[j-1], a[j]) ;
              }
          }
    }
}
```

This is an example of the algorithm in php

```
function oddEvenSorting(&$a) {
 $n = count($a);
 $sorted = false;
 while (!$sorted) {
  $sorted = true;
  for ($i = 1; $i < ($n - 1); $i += 2) {
   if ($a[$i] > $a[$i + 1]) {
    list($a[$i], $a[$i + 1]) = array($a[$i + 1], $a[$i]);
    if ($sorted) $sorted = false;
   }
  }

  for ($i = 0; $i < ($n - 1); $i += 2) {
   if ($a[$i] > $a[$i + 1]) {
    list($a[$i], $a[$i + 1]) = array($a[$i + 1], $a[$i]);
    if ($sorted) $sorted = false;
   }
  }
 }
}
```

This is an example of the algorithm in python.

```
def oddevenSort(x):
 sorted = False
 while sorted == False:
  sorted = True

  for i in range(0, len(x)-1, 2):
   if x[i] > x[i+1]:
    x[i], x[i+1] = x[i+1], x[i]
    sorted = False
  for i in range(1, len(x)-1, 2):
   if x[i] > x[i+1]:
    x[i], x[i+1] = x[i+1], x[i]
    sorted = False
 return x
```

This is an example of the algorithm in MATLAB/OCTAVE.

```
function x = oddevenSort(x)
sorted = false;
n = length(x);
while ~sorted
    sorted = true;
    for ii=1:2:n-1
        if x(ii) > x(ii+1)

            [x(ii), x(ii+1)] = deal(x(ii+1), x(ii));
            sorted = false;
        end
    end
    for ii=2:2:n-1
        if x(ii) > x(ii+1)
            [x(ii), x(ii+1)] = deal(x(ii+1), x(ii));
            sorted = false;
        end
    end
end
```

Proof of Correctness [edit]

Claim: Let $a_1, ..., a_n$ be a sequence of data ordered by <. The odd-even sort algorithm correctly sorts this data in n passes. (A pass here is defined to be a full sequence of odd-even, or even-odd comparisons. The passes occur in order pass 1: odd-even, pass 2: even-odd, etc.)

Proof:

This proof is based loosely on one by Thomas Worsch.[6]

Since the sorting algorithm only involves comparison-swap operations and is oblivious (the order of comparison-swap operations does not depend on the data), by Knuth's 0-1 sorting principle,[7][8] it suffices to check correctness when each a_i is either 0 or 1. Assume that there are e 1's.

Observe that the rightmost 1 can be either in an even or odd position, so it might not be moved by the first odd-even pass. But after the first odd-even pass, the rightmost 1 will be in an even position. It follows that it will be moved to the right by all remaining passes. Since the rightmost one starts in position greater than or equal to e, it must be moved at most $n - e$ steps. It follows that it takes at most $n - e + 1$ passes to move the rightmost 1 to its correct position.

Now, consider the second rightmost 1. After two passes, the 1 to its right will have moved right by at least one step. It follows that, for all remaining passes, we can view the second rightmost 1 as the rightmost 1. The second rightmost 1 starts in position at least $e - 1$ at must be moved to position at most $n - 1$, so it must be moved at most $(n - 1) - (e - 1) = n - e$ steps. After at most 2 passes, the rightmost 1 will have already moved, so the entry to the right of the second rightmost 1 will be 0. Hence, for all passes after the first two, the second rightmost 1 will move to the right. It thus takes at most $n - e + 2$ passes to move the second rightmost 1 to its correct position.

Continuing in this manner, by induction it can be shown that the i-th rightmost 1 is moved to its correct position in at most $n - e + i + 1$ passes. It follows that the e-th rightmost 1 is moved to its correct position in at most $n - e + (e - 1) + 1 = n$ passes (consider: counting starts at value "0"). The list is thus correctly sorted in n passes. QED.

We remark that each pass takes O(n) steps, so this algorithm is O(n^2) complexity.

References [edit]

1. ^ Phillips, Malcolm. "Array Sorting". Retrieved 3 August 2011.
2. ^ N. Haberman (1972) "Parallel Neighbor Sort (or the Glory of the Induction Principle)," CMU Computer Science Report (available as Technical report AD-759 248, National Technical Information Service, US Department of Commerce, 5285 Port Royal Rd Sprigfield VA 22151).
3. ^ S. Lakshmivarahan, S. K. Dhall, and L. L. Miller (1984), Franz L. Alt and Marshall C. Yovits, eds., "Parallel Sorting Algorithms", *Advances in computers* (Academic Press) **23**: 295–351, ISBN 978-0-12-012123-6
4. ^ Robert Sedgewick (2003). *Algorithms in Java, Parts 1-4* (3rd ed.). Addison-Wesley Professional. pp. 454–464. ISBN 978-0-201-36120-9.
5. ^ Allen Kent and James G. Williams (1993). *Encyclopedia of Computer Science and Technology: Supplement 14*. CRC Press. pp. 33–38. ISBN 978-0-8247-2282-1.
6. ^ http://liinwww.ira.uka.de/~thw/vl-hiroshima/slides-4.pdf
7. ^ http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/nulleinsen.htm
8. ^ http://www.net.t-labs.tu-berlin.de/~stefan/netalg13-9-sort.pdf

v · t · e	Sorting algorithms [hide]
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting
Exchange sorts	Bubble sort · Cocktail sort · **Odd–even sort** · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block sort · Timsort · Introsort · Spreadsort · JSort
Other	Topological sorting · Pancake sorting · Spaghetti sort

This computer science article is a stub. You can help Wikipedia by expanding it.

Categories: Sorting algorithms | Comparison sorts | Stable sorts | Computer science stubs