**1. American flag sort**

From Wikipedia, the free encyclopedia

Jump to: [navigation](http://en.wikipedia.org/wiki/American_flag_sort#mw-head), [search](http://en.wikipedia.org/wiki/American_flag_sort#p-search)

An [**american flag**](http://en.wikipedia.org/wiki/American_flag) **sort** is an efficient, [in-place](http://en.wikipedia.org/wiki/In-place) variant of [radix sort](http://en.wikipedia.org/wiki/Radix_sort) that distributes items into hundreds of [buckets](http://en.wikipedia.org/wiki/Bucket_%28computing%29). Non-comparative sorting algorithms such as radix sort and american flag sort are typically used to sort large objects such as strings, for which comparison is not a unit-time operation.[[1]](http://en.wikipedia.org/wiki/American_flag_sort#cite_note-mcilroy-0)

American flag sort iterates through the bits of the objects, considering several bits of each object at a time. For each set of bits, american flag sort makes two passes through the array of objects: first to count the number of objects that will fall in each bin, and second to place each object in its bucket. This works especially well when sorting a byte at a time, using 256 buckets. With some optimizations, it is twice as fast as [quicksort](http://en.wikipedia.org/wiki/Quicksort) for large sets of [strings](http://en.wikipedia.org/wiki/String_%28computer_science%29).[[1]](http://en.wikipedia.org/wiki/American_flag_sort#cite_note-mcilroy-0)

The name comes by [analogy](http://en.wikipedia.org/wiki/Analogy) with the [Dutch national flag problem](http://en.wikipedia.org/wiki/Dutch_national_flag_problem) in the last step: efficiently [partition](http://en.wikipedia.org/wiki/Partition_of_a_set) the array into many "stripes".

|  |
| --- |
|  |

**Algorithm**

Sorting algorithms in general sort a list of objects according to some ordering scheme. In contrast to [comparison-based sorting algorithms](http://en.wikipedia.org/wiki/Comparison_sort), such as [quicksort](http://en.wikipedia.org/wiki/Quicksort), american flag sort can only sort integers (or objects that can be interpreted as integers). In-place sorting algorithms, including american flag sort, run without allocating a significant amount of memory beyond that used by the original array. This is a significant advantage, both in memory savings and in time saved copying the array.

American flag sort works by successively dividing a list of objects into buckets based on the first digit of their base-N representation (the base used is referred to as the *radix*). When N is 2, each object can be swapped into the correct bucket by using the [Dutch national flag algorithm](http://en.wikipedia.org/wiki/Dutch_national_flag_problem). When N is larger, however, objects cannot be immediately swapped into place, because it is unknown where each bucket should begin and end. American flag sort gets around this problem by making two passes through the array. The first pass counts the number of objects that belong in each of the N buckets. The beginning and end of each bucket in the original array is then computed as the sum of sizes of preceding buckets. The second pass swaps each object into place.

American flag sort is most efficient with a radix that is a power of 2, because bit-shifting operations can be used instead of expensive logarithms to compute the value of each digit. When sorting strings, it is typical to use a radix of 256, which corresponds by sorting character-by-character.[[1]](http://en.wikipedia.org/wiki/American_flag_sort#cite_note-mcilroy-0)

**Pseudocode**

american\_flag\_sort(Array, Radix)

for each digit D:

# first pass: compute counts

Counts <- zeros(Radix)

for object X in Array:

Counts[digit D of object X in base Radix] += 1

# compute bucket offsets

Offsets <- [ sum(Counts[0..i]) for i in 1..Radix]

# swap objects into place

for object X in Array:

swap X to the bucket starting at Offsets[digit D of X in base Radix]

for each Bucket:

american\_flag\_sort(Bucket, Radix)

**Sample implementation in Python**

This example written in the Python programming language will perform american flag sort for any radix of 2 or greater. Simplicity of exposition is chosen over clever programming, and so the log function is used instead of bit shifting techniques.

def get\_radix\_val(x, digit, radix):

return int(floor(x / radix\*\*digit)) % radix

def compute\_offsets(a\_list, start, end, digit, radix):

counts = [0 for \_ in range(radix)]

for i in range(start, end):

val = get\_radix\_val(a\_list[i], digit, radix)

counts[val] += 1

offsets = [0 for \_ in range(radix)]

sum = 0

for i in range(radix):

offsets[i] = sum

sum += counts[i]

return offsets

def swap(a\_list, offsets, start, end, digit, radix):

i = start

next\_free = copy(offsets)

cur\_block = 0

while cur\_block < radix-1:

if i >= offsets[cur\_block+1]:

cur\_block += 1

continue

radix\_val = get\_radix\_val(a\_list[i], digit, radix)

if radix\_val == cur\_block:

i += 1

continue

swap\_to = next\_free[radix\_val]

a\_list[i], a\_list[swap\_to] = a\_list[swap\_to], a\_list[i]

next\_free[radix\_val] += 1

def american\_flag\_sort\_helper(a\_list, start, end, digit, radix):

offsets = compute\_offsets(a\_list, start, end, digit, radix)

swap(a\_list, offsets, start, end, digit, radix)

if digit == 0:

return

for i in range(len(offsets)-1):

american\_flag\_sort\_helper(a\_list, offsets[i], offsets[i+1], digit-1, radix)

def american\_flag\_sort(a\_list, radix):

for x in a\_list:

assert(type(x) == int)

max\_val = max(a\_list)

max\_digit = int(floor(log(max\_val, radix)))

american\_flag\_sort\_helper(a\_list, 0, len(a\_list), max\_digit, radix)

# 2. Bead sort

From Wikipedia, the free encyclopedia

Jump to: [navigation](http://en.wikipedia.org/wiki/Bead_sort#mw-head), [search](http://en.wikipedia.org/wiki/Bead_sort#p-search)

**Bead sort** is a [natural](http://en.wikipedia.org/w/index.php?title=Natural_algorithm&action=edit&redlink=1) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm), developed by [Joshua J. Arulanandham](http://en.wikipedia.org/w/index.php?title=Joshua_J._Arulanandham&action=edit&redlink=1), [Cristian S. Calude](http://en.wikipedia.org/wiki/Cristian_S._Calude) and [Michael J. Dinneen](http://en.wikipedia.org/wiki/Michael_J._Dinneen) in 2002, and published in The Bulletin of the [European Association for Theoretical Computer Science](http://en.wikipedia.org/wiki/European_Association_for_Theoretical_Computer_Science). Both [digital](http://en.wikipedia.org/wiki/Digital) and [analog](http://en.wikipedia.org/wiki/Analog_computer) hardware [implementations](http://en.wikipedia.org/wiki/Implementation) of bead sort can achieve a sorting time of [*O*](http://en.wikipedia.org/wiki/Big_O_Notation)(*n*); however, the implementation of this algorithm tends to be significantly slower in [software](http://en.wikipedia.org/wiki/Software) and can only be used to sort lists of [positive integers](http://en.wikipedia.org/wiki/Positive_integer). Also, it would seem that even in the best case, the algorithm requires [*O*](http://en.wikipedia.org/wiki/Big_O_Notation)(*n2*) space.

|  |
| --- |
|  |

## Algorithm overview

The bead sort operation can be compared to the manner in which beads slide on parallel poles, such as on an [abacus](http://en.wikipedia.org/wiki/Abacus). However, each pole may have a distinct number of beads. Initially, it may be helpful to imagine the beads suspended on vertical poles. In Step 1, such an arrangement is displayed using *n=5* rows of beads on *m=4* vertical poles. The numbers to the right of each row indicate the number that the row in question represents; rows 1 and 2 are representing the positive integer 3 (because they each contain three beads) while the top row represents the positive integer 2 (as it only contains two beads).[[1]](http://en.wikipedia.org/wiki/Bead_sort#cite_note-rowconventions-0)

If we then allow the beads to fall, the rows now represent the same integers in sorted order. Row 1 contains the largest number in the set, while row *n* contains the smallest. If the above-mentioned convention of rows containing a series of beads on poles 1..*k* and leaving poles *k*+1..*m* empty has been followed, it will continue to be the case here.

The action of allowing the beads to "fall" in our physical example has allowed the larger values from the higher rows to propagate to the lower rows. If the value represented by row *a* is smaller than the value contained in row *a+1*, some of the beads from row *a+1* will fall into row *a*; this is certain to happen, as row *a* does not contain beads in those positions to stop the beads from row *a+1* from falling.

The mechanism underlying bead sort is similar to that behind [counting sort](http://en.wikipedia.org/wiki/Counting_sort); the number of beads on each pole corresponds to the number of elements with value equal or greater than the index of that pole.

**Complexity**

Bead sort can be implemented with three general levels of complexity, among others:

* [*O*](http://en.wikipedia.org/wiki/Big_O_Notation)(1): The beads are all moved simultaneously in the same time unit, as would be the case with the simple physical example above. This is an abstract complexity, and cannot be implemented in practice.
* [*O*](http://en.wikipedia.org/wiki/Big_O_Notation)(): In a realistic physical model that uses gravity, the time it takes to let the beads fall is proportional to the square root of the maximum height, which is proportional to n.
* [*O*](http://en.wikipedia.org/wiki/Big_O_Notation)(n): The beads are moved one row at a time. This is the case used in the analog and [digital hardware](http://en.wikipedia.org/wiki/Digital_hardware) solutions.
* [*O*](http://en.wikipedia.org/wiki/Big_O_Notation)(S), where S is the sum of the integers in the input set: Each bead is moved individually. This is the case when bead sort is implemented without a mechanism to assist in finding empty spaces below the beads, such as in software implementations.

Like the [Pigeonhole sort](http://en.wikipedia.org/wiki/Pigeonhole_sort), bead sort is unusual in that it can perform faster than [*O*](http://en.wikipedia.org/wiki/Big_O_Notation)(*n*[log](http://en.wikipedia.org/wiki/Logarithm)*n*), the fastest performance possible for a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). This is possible because the key for a bead sort is always a positive integer and bead sort exploits its structure.

**References and notes**

1. [**^**](http://en.wikipedia.org/wiki/Bead_sort#cite_ref-rowconventions_0-0) By convention, a row representing the positive integer *k* should have beads on poles 1..*k* and poles *k*+1..*m* should be empty. This is not a strict requirement, but will most likely simplify implementation.

# 3. Bogosort

Jump to: [navigation](http://en.wikipedia.org/wiki/Bogosort#mw-head), [search](http://en.wikipedia.org/wiki/Bogosort#p-search)

|  |  |
| --- | --- |
| Bogosort | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | Unbounded[[1]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Fun07-1) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | *Ω*(n)[[1]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Fun07-1) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | *O*(n × n!)[[1]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Fun07-1) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |

In [computer science](http://en.wikipedia.org/wiki/Computer_science), **bogosort**[[1]](http://en.wikipedia.org/wiki/Bogosort" \l "cite_note-Fun07-1)[[2]](http://en.wikipedia.org/wiki/Bogosort#cite_note-KSFS-2) (also **stupid sort**[[3]](http://en.wikipedia.org/wiki/Bogosort#cite_note-3) or **slowsort**[[4]](http://en.wikipedia.org/wiki/Bogosort" \l "cite_note-Naish86-4)[[5]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Naish95-5)) is a particularly ineffective [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) based on the [generate and test](http://en.wikipedia.org/wiki/Trial_and_error) paradigm. It is not useful for sorting, but may be used for educational purposes, to contrast it with other more realistic algorithms; it has also been used as an example in [logic programming](http://en.wikipedia.org/wiki/Logic_programming).[[2]](http://en.wikipedia.org/wiki/Bogosort#cite_note-KSFS-2)[[4]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Naish86-4)[[5]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Naish95-5) If bogosort were used to sort a [deck of cards](http://en.wikipedia.org/wiki/Deck_of_cards), it would consist of checking if the deck were in order, and if it were not, throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted. Its name comes from the word *bogus*.

|  |
| --- |
|  |

**Description of the algorithm**

Following is a description of the algorithm in [pseudocode](http://en.wikipedia.org/wiki/Pseudocode).

**Pseudocode**

**while not** isInOrder(deck):

shuffle(deck)

**Running time and termination**

This [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) is probabilistic in nature. If all elements to be sorted are distinct, the expected number of comparisons in the average case is [asymptotically equivalent to](http://en.wikipedia.org/wiki/Asymptotic_analysis) , and the expected number of swaps in the average case equals .[[1]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Fun07-1) The expected number of swaps grows faster than the expected number of comparisons, because if the elements are not in order, this will usually be discovered after only a few comparisons no matter how many elements there are, but the work of shuffling the collection is proportional to its size. In the worst case, the number of comparisons and swaps are both unbounded, for the same reason that a tossed coin might turn up heads any number of times in a row.

The best case occurs if the list as given is already sorted; in this case the expected number of comparisons is , and no swaps at all are carried out.[[1]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Fun07-1)

For any collection of fixed size, the expected running time of the algorithm is finite for much the same reason that the [infinite monkey theorem](http://en.wikipedia.org/wiki/Infinite_monkey_theorem) holds: there is some probability of getting the right permutation, so given an unbounded number of tries it will [almost surely](http://en.wikipedia.org/wiki/Almost_surely) eventually be chosen. However, if a [pseudorandom number generator](http://en.wikipedia.org/wiki/Pseudorandom_number_generator) is used in place of a random source, it may never terminate, since these exhibit long-term cyclic behavior.[*[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)*]

**Related algorithms**

Goro sort

is a sorting algorithm introduced in the 2011 [Google Code Jam](http://en.wikipedia.org/wiki/Google_Code_Jam).[[6]](http://en.wikipedia.org/wiki/Bogosort#cite_note-6) As long as the list is not in order, a subset of all elements is randomly permuted. If this subset is optimally chosen each time this is performed, the [expected value](http://en.wikipedia.org/wiki/Expected_value) of the total number of times this operation needs to be done is equal to the number of misplaced elements.

Bozo sort

is another sorting algorithm based on random numbers. If the list is not in order, it picks two items at random and swaps them, then checks to see if the list is sorted. The running time analysis of Bozo Sort is more difficult, but some estimates are found in H. Gruber's analysis of perversely awful randomized sorting algorithms.[[1]](http://en.wikipedia.org/wiki/Bogosort#cite_note-Fun07-1) O(n!) is found to be the expected average case.

Quantum bogosort

An [in-joke](http://en.wikipedia.org/wiki/In-joke) among some computer scientists is that [quantum computing](http://en.wikipedia.org/wiki/Quantum_computing) could be used to effectively implement a bogosort with a time complexity of O(n). It uses true quantum randomness to randomly permute the list. The list is then inspected, and if it is not in order, the universe is destroyed. By the [many-worlds interpretation](http://en.wikipedia.org/wiki/Many-worlds_interpretation) of quantum physics, the quantum randomization spawns (where N is the number of random bits) universes and one of these will be such that this single shuffle had produced the list in sorted order.

**See also**

* [Las Vegas algorithm](http://en.wikipedia.org/wiki/Las_Vegas_algorithm)
* [Stooge sort](http://en.wikipedia.org/wiki/Stooge_sort)

**References**

1. ^ [***a***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Fun07_1-0) [***b***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Fun07_1-1) [***c***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Fun07_1-2) [***d***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Fun07_1-3) [***e***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Fun07_1-4) [***f***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Fun07_1-5) [***g***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Fun07_1-6) Gruber, H.; Holzer, M.; Ruepp, O., ["Sorting the slow way: an analysis of perversely awful randomized sorting algorithms"](http://www.hermann-gruber.com/data/fun07-final.pdf), *4th International Conference on Fun with Algorithms, Castiglioncello, Italy, 2007*, Lecture Notes in Computer Science, **4475**, Springer-Verlag, pp. 183–197, [doi](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1007/978-3-540-72914-3\_17](http://dx.doi.org/10.1007%2F978-3-540-72914-3_17).
2. ^ [***a***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-KSFS_2-0) [***b***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-KSFS_2-1) Kiselyov, Oleg; Shan, Chung-chieh; Friedman, Daniel P.; Sabry, Amr (2005), ["Backtracking, interleaving, and terminating monad transformers: (functional pearl)"](http://www.dicta.org.uk/programming/LogicT.pdf), *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*, SIGPLAN Notices, pp. 192–203, [doi](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1145/1086365.1086390](http://dx.doi.org/10.1145%2F1086365.1086390)
3. [**^**](http://en.wikipedia.org/wiki/Bogosort#cite_ref-3) E. S. Raymond. "bogo-sort". *The New Hacker’s Dictionary*. MIT Press, 1996.
4. ^ [***a***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Naish86_4-0) [***b***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Naish86_4-1) Naish, Lee (1986), "Negation and quantifiers in NU-Prolog", *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, **225**, Springer-Verlag, pp. 624–634, [doi](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1007/3-540-16492-8\_111](http://dx.doi.org/10.1007%2F3-540-16492-8_111).
5. ^ [***a***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Naish95_5-0) [***b***](http://en.wikipedia.org/wiki/Bogosort#cite_ref-Naish95_5-1) Naish, Lee (June 1995), *Pruning in logic programming*, Tech. Report 95/16, Melbourne, Australia: Department of Computer Science, University of Melbourne, [CiteSeerX](http://en.wikipedia.org/wiki/CiteSeer#CiteSeerX): [10.1.1.54.2347](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.2347).
6. [**^**](http://en.wikipedia.org/wiki/Bogosort#cite_ref-6) [Google Code Jam 2011, Qualification Rounds, Problem D](http://code.google.com/codejam/contest/dashboard?c=975485#s=p3)

**4.** CartesianTreeSort

 \* File: CartesianTreeSort.hh  
 \* Author: Keith Schwarz (htiek@cs.stanford.edu)  
 \*  
 \* An implementation of a sort-style STL algorithm that uses a Cartesian tree  
 \* sort.  Cartesian tree sort is an adaptive, out-of-place sorting algorithm  
 \* with O(n) best-case behavior, O(n lg n) worst-case behavior, and O(n)  
 \* memory usage.  
 \*  
 \* A Cartesian tree is a tree created from a set of data that obeys the  
 \* following structural invariants:  
 \*  
 \* 1. The tree obeys in the min (or max) heap property - each node is less (or  
 \*    greater) than its children.  
 \* 2. An inorder traversal of the nodes yields the values in the same order in  
 \*    which they appear in the initial sequence.  
 \*  
 \* It's easy to see that this tree is unique by a quick induction on the size  
 \* of the input sequence.  As a base case, if the input sequence is empty,  
 \* then the empty tree is the unique Cartesian tree over that sequence.  For  
 \* the inductive case, assume that for all trees containing n' < n elements,  
 \* there is a unique Cartesian tree for each sequence of n' nodes.  Now take  
 \* any sequence of n elements.  Because a Cartesian tree is a min-heap, the  
 \* smallest element of the sequence must be the root of the Cartesian tree.  
 \* Because an inorder traversal of the elements must yield the input sequence,  
 \* we know that all nodes to the left of the min element must be in its left  
 \* subtree and similarly for the nodes to the right.  Since the left and right  
 \* subtree are both Cartesian trees with at most n - 1 elements in them (since  
 \* the min element is at the root), by the induction hypothesis there is a  
 \* unique Cartesian tree that could be the left or right subtree.  Since all  
 \* our decisions were forced, we end up with a unique tree, completing the  
 \* induction.  
 \*  
 \* An interesting note is that Cartesian trees are not necessarily height-  
 \* balanced.  In particular, any sequence in sorted or reverse-sorted order  
 \* will have a Cartesian tree that degrades to a linked list.  For example:  
 \*  
 \*                                    1 2 3 4 5  
 \*  
 \* has the following Cartesian tree:  
 \*  
 \*                                1  
 \*                                 \  
 \*                                  2  
 \*                                   \  
 \*                                    3  
 \*                                     \  
 \*                                      4  
 \*                                       \  
 \*                                        5  
 \*  
 \* In general, these Cartesian trees have height O(n).  
 \*  
 \* Interestingly, it's possible to build a Cartesian tree from a sequence of  
 \* data in linear time.  The algorithm as follows.  Beginning with the empty  
 \* tree, scan across the sequence from the left to the right adding new nodes  
 \* as follows:  
 \*  
 \* 1. Position the node as the right child of the rightmost node.  
 \* 2. Scan upward from the node's parent up to the root of the tree until a  
 \*    node is found whose value is less than the current value.  
 \* 3. If such a node is found, set its right child to be the new node, and  
 \*    set the new node's left child to be the previous right child.  
 \* 4. If no such node is found, set the new child to be the root, and set the  
 \*    new node's left child to be the previous tree.  
 \*  
 \* At first, this algorithm might not seem to run in linear time.  After all,  
 \* if the tree can become so imbalanced that it has height O(n) and we're  
 \* doing O(n) insertions, then it seems like the runtime should be O(n^2).  
 \* This bound is correct, but it isn't tight.  In particular, we can show that  
 \* the amortized cost of any insert is O(1), giving the net operations a total  
 \* runtime of O(n).  Define the potential of the tree to be the number of  
 \* nodes in its right spine.  The actual cost of any insertion is O(k), where  
 \* k is the number of nodes we considered on the way up from the rightmost  
 \* node to the node's new parent.  However, after we find where the node  
 \* belonds, we change the tree by moving k - 1 nodes into the left subtree of  
 \* the newly-inserted node.  This means that the tree's potential decreases by  
 \* k - 1.  We then increase the number of nodes in the right spine by one by  
 \* adding the new node there.  This means that the change in potential is  
 \* 1 - (k - 1) = 2 - k, giving an amortized cost of k + 2 - k = 2 = O(1) as  
 \* requested.  
 \*  
 \* Once we have built a Cartesian tree from a range of elements, we can sort  
 \* that range efficiently using a modified version of heapsort.  Construct a  
 \* binary heap that holds nodes in Cartesian trees and initialize it to the  
 \* root of the Cartesian tree.  Then, until the heap is empty, continuously  
 \* dequeue an element from the heap, add its root element to the next spot in  
 \* the sorted sequence, and then add the node's children to the heap.  This  
 \* visits each node in the sequence once, never visits a node until all of  
 \* its parents in the heap are visited, and visits the exposed roots in sorted  
 \* order.  This guarantees that the nodes come back sorted.  To see why,  
 \* suppose for the sake of contradiction two nodes come back out of order.  
 \* Call these nodes A and B with A > B.  Since the Cartesian tree is a min-  
 \* heap, B must not be an ancestor of A.  Since A came back first, B must not  
 \* have been an exposed root, or it would have come out of the heap before A.  
 \* But since B is not exposed, one of its ancestors must still be in the heap,  
 \* and since its ancestor has a value no greater than B's it would have come  
 \* out of the heap before A, a contradiction.  
 \*  
 \* Let's now consider the runtime of this phase.  We know that we will be  
 \* making O(n) insertions and dequeues from the binary heap, so each operation  
 \* takes at most O(lg n) time.  This gives us a worst-case runtime of  
 \* O(n lg n), matching the runtime guarantee of heapsort.  However, this bound  
 \* may not be tight.  In particular, suppose that our Cartesian tree is the  
 \* degenerate linked list described above.  Then initially the queue will have  
 \* exactly one element in it, and every time we dequeue the node and add its  
 \* children we'll only add a singleton node to the queue.  This means that the  
 \* queue will always have exactly one element in it, and so all the heap  
 \* operations will take O(1) time for a net runtime of O(n).  In fact, in any  
 \* Cartesian tree where each node has one child, we'll get this runtime.  
 \*  
 \* The interesting part about this algorithm is that it's possible to  
 \* explicitly quantify how much faster than O(n lg n) the algorithm will run  
 \* by using a measure called oscillation.  For any element x in the input  
 \* sequence, define cross(x) to be the number of adjacent pairs of entries  
 \* (y, z) in the input sequence such that either y <= x <= z or y >= x >= z.  
 \* Then we define the oscillation of the input sequence (denoted k) as the  
 \* average of cross(x) over all entries in the sequence.  It can be shown that  
 \* the overall runtime of the algorithm is O(n lg k), where k is this measure  
 \* of oscillation.  An interesting detail is that if the input is broken down  
 \* into S different sorted subsequences, k = O(S).  Consequently, if the  
 \* number of sorted subsequences in the input sequence is small (say, O(1)),  
 \* the algorithm will run in o(n lg n) time.  This result is due to the paper  
 \* "Heapsort, Adapted for Presorted Files," which first introduced Cartesian  
 \* tree sort.  Because the first step of the algorithm (building up the  
 \* Cartesian tree) runs in O(n), the overall runtime of the algorithm is  
 \* O(n lg k), which is at best O(n) and at worst O(n lg n).  
 \*/  
  
#ifndef CartesianTreeSort\_Included  
#define CartesianTreeSort\_Included  
  
/\*\*  
 \* void CartesianTreeSort(ForwardIterator begin, ForwardIterator end);  
 \* Usage: CartesianTreeSort(v.begin(), v.end());  
 \* ---------------------------------------------------------------------------  
 \* Sorts the range [begin, end) into ascending order according to the default  
 \* ordering using the Cartesian tree sort algorithm.  
 \*/  
template <typename ForwardIterator>  
void CartesianTreeSort(ForwardIterator begin, ForwardIterator end);  
  
/\*\*  
 \* void CartesianTreeSort(ForwardIterator begin, ForwardIterator end,  
 \*                        Comparator comp);  
 \* Usage: CartesianTreeSort(v.begin(), v.end(), std::greater<int>());  
 \* ---------------------------------------------------------------------------  
 \* Sorts the range [begin, end) into ascending order according to specified  
 \* comparator using the Cartesian tree sort algorithm.  
 \*/  
template <typename ForwardIterator, typename Comparator>  
void CartesianTreeSort(ForwardIterator begin, ForwardIterator end,  
                       Comparator comp);  
  
/\* \* \* \* \* Implementation Below This Point \* \* \* \* \*/  
#include <iterator>   // For iterator\_traits  
#include <functional> // For less  
#include <memory>     // For auto\_ptr  
#include <stack>  
#include <queue>  
#include <vector>  
  
namespace cartesiantreesort\_detail {  
  /\* A utility struct representing a node in a Cartesian tree. \*/  
  template <typename T> struct Node {  
    const T value;      // The node's value  
    Node\* left, \*right; // Pointers to the proper subtrees  
  
    /\* Constructor: Node(const T& value);  
     \* Usage: Node\* node = new Node(value);  
     \* -----------------------------------------------------------------------  
     \* Constructs a new Node having the specified value and no children.  
     \*/  
    explicit Node(const T& value) : value(value) {  
      /\* Initially this node is isolated. \*/  
      left = right = NULL;  
    }  
  
    /\* Destructor: ~Node();  
     \* Usage: (implicit)  
     \* -----------------------------------------------------------------------  
     \* Deallocates the tree rooted at this Node.  
     \*/  
    ~Node() {  
      delete left;  
      delete right;  
    }  
  };  
  
  /\* Node<T>\* MakeCartesianTree(InputIterator begin, InputIterator end,  
   \*                            Comparator comp);  
   \* Usage: Node<T>\* tree = MakeCartesianTree(begin, end, comp);  
   \* -------------------------------------------------------------------------  
   \* Constructs and returns a Cartesian tree containing the specified values  
   \* and sorted as a min-heap with respect to the given comparator.  The  
   \* return type of this function is a bit messy because it has to introspect  
   \* on the iterator type to figure out what's being stored.  
   \*/  
  template <typename InputIterator, typename Comparator>  
  Node<typename std::iterator\_traits<InputIterator>::value\_type>\*  
  MakeCartesianTree(InputIterator begin, InputIterator end,  
                    Comparator comp) {  
    /\* For sanity's sake, typedef the type being iterated over. \*/  
    typedef typename std::iterator\_traits<InputIterator>::value\_type T;  
  
    /\* Keep track of the root of the tree, which is initially NULL because the  
     \* tree is empty.  
     \*/  
    Node<T>\* root = NULL;  
  
    /\* In addition to this, we'll maintain a stack of the nodes on the right  
     \* spine of the tree, in the order in which you would encounter them if  
     \* you marched upward from the rightmost node to the root.  
     \*/  
    std::stack< Node<T>\* > rightSpine;  
  
    /\* To avoid edge cases later on, we'll add NULL to the right spine.  This  
     \* does make some sense mathematically, since if we walk from the  
     \* rightmost node to the root and upward we'd walk off the tree at some  
     \* point.  
     \*/  
    rightSpine.push(NULL);  
  
    /\* Scan across the elements, adding them one at a time. \*/  
    for (; begin != end; ++begin) {  
      /\* Construct the new node to insert. \*/  
      Node<T>\* node = new Node<T>(\*begin);  
  
      /\* Starting at the rightmost node, walk upward along the right spine  
       \* until we find a node that can serve as the parent.  Because the spine  
       \* is never empty (NULL will always be there), we don't need to worry  
       \* about an empty stack.  
       \*/  
      Node<T>\* curr;  
      for (curr = rightSpine.top(); curr != NULL; rightSpine.pop(), curr = rightSpine.top())  
        if (comp(curr->value, node->value))  
          break;  
  
      /\* At this point, there are two cases to consider.  First, this new node  
       \* might be the new minimum.  In that case, we make it the global tree  
       \* root, and to preserve the inorder walk requirement make the old tree  
       \* its right child.  
       \*/  
      if (curr == NULL) {  
        node->left = root;  
        root = node;  
      }  
      /\* Otherwise, we need to pull the current node's right subtree so that  
       \* it's the left subtree of the current node, and then set the new node  
       \* as the right child of the current node.  
       \*/  
      else {  
        node->left = curr->right;  
        curr->right = node;  
      }  
  
      /\* This new node is now on the right spine, so we'll add it to the stack  
       \* of nodes stored there.  
       \*/  
      rightSpine.push(node);  
    }  
  
    /\* Hand back the resulting tree. \*/  
    return root;  
  }  
  
  /\* A utility comparator class that compares Node<T>\*s by the reverse of  
   \* their comparison by some comparator.  The rationale is that we will use  
   \* this comparator in a priority\_queue of Node<T>\*s, and will need some way  
   \* to ensure that the nodes are compared so that the smallest elements come  
   \* back first.  
   \*/  
  template <typename T, typename Comparator>  
  class NodeComparator {  
  public:  
    /\* Constructor: NodeComparator(Comparator comp);  
     \* Usage: NodeComparator comp(rawComp);  
     \* -----------------------------------------------------------------------  
     \* Constructs a new NodeComparator that uses the specified comparator on  
     \* the values in Node<T>\*s.  
     \*/  
    explicit NodeComparator(Comparator comp) : comp(comp) {  
      // Handled in initializer list  
    }  
  
    /\* Comparator: bool operator() (const Node<T>\* lhs, const Node<T>\* rhs) const;  
     \* Usage: comp(one, two);  
     \* -----------------------------------------------------------------------  
     \* Returns whether the first node compares at least as large as the second  
     \* node using the stored comparator.  
     \*/  
    bool operator() (const Node<T>\* lhs, const Node<T>\* rhs) const {  
      /\* Check if lhs >= rhs by seeing if lhs < rhs returns false. \*/  
      return !comp(lhs->value, rhs->value);  
    }  
  
  private:  
    Comparator comp; // The actual comparator to use  
  };  
}  
  
/\* Actual implementation of Cartesian tree sort, using a parameterized  
 \* comparator.  
 \*/  
template <typename ForwardIterator, typename Comparator>  
void CartesianTreeSort(ForwardIterator begin, ForwardIterator end,  
                       Comparator comp) {  
  /\* As an edge case, check if the input is empty.  This avoids a problem  
   \* later on in this function where we might try enqueueing a NULL tree node  
   \* into the queue.  
   \*/  
  if (begin == end) return;  
  
  /\* Grant access to our helper types and classes. \*/  
  using namespace cartesiantreesort\_detail;  
  
  /\* Again, for sanity's sake, typedef the type being iterated over. \*/  
  typedef typename std::iterator\_traits<ForwardIterator>::value\_type T;  
    
  /\* A type representing a priority queue that compares the value fields of  
   \* Cartesian tree nodes.  
   \*/  
  typedef std::priority\_queue<Node<T>\*, std::vector<Node<T>\*>,  
                              NodeComparator<T, Comparator> > PQueue;  
  
  /\* Construct a priority queue, wrapping up the comparator provided by the  
   \* client.  Due to the Most Vexing Parse, we have to parenthesize the  
   \* argument so this isn't accidentally interpreted as a function declaration.  
   \*/  
  PQueue pq((NodeComparator<T, Comparator>(comp)));  
  
  /\* Obtain a Cartesian tree over the input.  We'll store the result in a   
   \* const auto\_ptr to ensure that  
   \*  
   \* 1. The memory is reclaimed when the function exits and the auto\_ptr  
   \*    leaves scope.  
   \* 2. The memory isn't accidentally transferred elsewhere, because the  
   \*    auto\_ptr is const.  
   \*/  
  const std::auto\_ptr< Node<T> > tree(MakeCartesianTree(begin, end, comp));  
  
  /\* Initialize the priority queue to hold the Cartesian tree of the input. \*/  
  pq.push(tree.get());  
  
  /\* Now, scan across the sequence, placing the smallest known value at the  
   \* next open position and updating the queue accordingly.  
   \*/  
  for (ForwardIterator itr = begin; itr != end; ++itr) {  
    /\* Grab the next node from the queue. \*/  
    Node<T>\* curr = pq.top(); pq.pop();  
  
    /\* Store its value back into the sequence. \*/  
    \*itr = curr->value;  
  
    /\* Add any non-NULL subtrees of the current tree back into the queue. \*/  
    if (curr->left) pq.push(curr->left);  
    if (curr->right) pq.push(curr->right);  
  }  
}  
  
/\* Non-comparator version implemented in terms of the comparator version. \*/  
template <typename ForwardIterator>  
void CartesianTreeSort(ForwardIterator begin, ForwardIterator end) {  
  CartesianTreeSort(begin, end,  
                    std::less<typename std::iterator\_traits<ForwardIterator>::value\_type>());  
}

# 4(ii) Cartesian tree

A sequence of numbers and the Cartesian tree derived from them.

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **Cartesian tree** is a [binary tree](http://en.wikipedia.org/wiki/Binary_tree) derived from a sequence of numbers; it can be uniquely defined from the properties that it is [heap](http://en.wikipedia.org/wiki/Heap_%28data_structure%29)-ordered and that a [symmetric (in-order) traversal](http://en.wikipedia.org/wiki/Tree_traversal) of the tree returns the original sequence. Introduced by [Vuillemin (1980](http://en.wikipedia.org/wiki/Cartesian_tree#CITEREFVuillemin1980)) in the context of geometric [range searching](http://en.wikipedia.org/wiki/Range_searching) [data structures](http://en.wikipedia.org/wiki/Data_structure), Cartesian trees have also been used in the definition of the [treap](http://en.wikipedia.org/wiki/Treap) and [randomized binary search tree](http://en.wikipedia.org/wiki/Randomized_binary_search_tree) data structures for [binary search](http://en.wikipedia.org/wiki/Binary_search) problems. The Cartesian tree for a sequence may be constructed in [linear time](http://en.wikipedia.org/wiki/Linear_time) using a [stack](http://en.wikipedia.org/wiki/Stack_%28data_structure%29)-based algorithm for finding [all nearest smaller values](http://en.wikipedia.org/wiki/All_nearest_smaller_values) in a sequence.

|  |
| --- |
|  |

**Definition**

The Cartesian tree for a sequence of distinct numbers can be uniquely defined by the following properties:

1. The Cartesian tree for a sequence has one node for each number in the sequence. Each node is associated with a single sequence value.
2. A [symmetric (in-order) traversal](http://en.wikipedia.org/wiki/Tree_traversal) of the tree results in the original sequence. That is, the left subtree consists of the values earlier than the root in the sequence order, while the right subtree consists of the values later than the root, and a similar ordering constraint holds at each lower node of the tree.
3. The tree has the [heap property](http://en.wikipedia.org/wiki/Binary_heap): the parent of any non-root node has a smaller value than the node itself.[[1]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-1)

Based on the heap property, the root of the tree must be the smallest number in the sequence. From this, the tree itself may also be defined recursively: the root is the minimum value of the sequence, and the left and right subtrees are the Cartesian trees for the subsequences to the left and right of the root value. Therefore, the three properties above uniquely define the Cartesian tree.

If a sequence of numbers contains repetitions, the Cartesian tree may be defined by determining a consistent tie-breaking rule (for instance, determining that the first of two equal elements is treated as the smaller of the two) before applying the above rules.

An example of a Cartesian tree is shown in the figure above.

**Range searching and lowest common ancestors**

Two-dimensional range-searching using a Cartesian tree: the bottom point (red in the figure) within a three-sided region with two vertical sides and one horizontal side (if the region is nonempty) may be found as the nearest common ancestor of the leftmost and rightmost points (the blue points in the figure) within the slab defined by the vertical region boundaries. The remaining points in the three-sided region may be found by splitting it by a vertical line through the bottom point and recursing.

Cartesian trees may be used as part of an efficient [data structure](http://en.wikipedia.org/wiki/Data_structure) for [range minimum queries](http://en.wikipedia.org/wiki/Range_Minimum_Query), a [range searching](http://en.wikipedia.org/wiki/Range_searching) problem involving queries that ask for the minimum value in a contiguous subsequence of the original sequence.[[2]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-2) In a Cartesian tree, this minimum value may be found at the [lowest common ancestor](http://en.wikipedia.org/wiki/Lowest_common_ancestor) of the leftmost and rightmost values in the subsequence. For instance, in the subsequence (12,10,20,15) of the sequence shown in the first illustration, the minimum value of the subsequence (10) forms the lowest common ancestor of the leftmost and rightmost values (12 and 15). Because lowest common ancestors may be found in constant time per query, using a data structure that takes linear space to store and that may be constructed in linear time,[[3]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-3) the same bounds hold for the range minimization problem.

[Bender & Farach-Colton (2000](http://en.wikipedia.org/wiki/Cartesian_tree#CITEREFBenderFarach-Colton2000)) reversed this relationship between the two data structure problems by showing that lowest common ancestors in an input tree could be solved efficiently applying a non-tree-based technique for range minimization. Their data structure uses an [Euler tour](http://en.wikipedia.org/wiki/Euler_tour) technique to transform the input tree into a sequence and then finds range minima in the resulting sequence. The sequence resulting from this transformation has a special form (adjacent numbers, representing heights of adjacent nodes in the tree, differ by ±1) which they take advantage of in their data structure; to solve the range minimization problem for sequences that do not have this special form, they use Cartesian trees to transform the range minimization problem into a lowest common ancestor problem, and then apply the Euler tour technique to transform the problem again into one of range minimization for sequences with this special form.

The same range minimization problem may also be given an alternative interpretation in terms of two dimensional range searching. A collection of finitely many points in the [Cartesian plane](http://en.wikipedia.org/wiki/Cartesian_plane) may be used to form a Cartesian tree, by sorting the points by their *x*-coordinates and using the *y*-coordinates in this order as the sequence of values from which this tree is formed. If *S* is the subset of the input points within some vertical slab defined by the inequalities *L* ≤ *x* ≤ *R*, *p* is the leftmost point in *S* (the one with minimum *x*-coordinate), and *q* is the rightmost point in *S* (the one with maximum *x*-coordinate) then the lowest common ancestor of *p* and *q* in the Cartesian tree is the bottommost point in the slab. A three-sided range query, in which the task is to list all points within a region bounded by the three inequalities *L* ≤ *x* ≤ *R* and *y* ≤ *T*, may be answered by finding this bottommost point *b*, comparing its *y*-coordinate to *T*, and (if the point lies within the three-sided region) continuing recursively in the two slabs bounded between *p* and *b* and between *b* and *q*. In this way, after the leftmost and rightmost points in the slab are identified, all points within the three-sided region may be listed in constant time per point.[[4]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-gbt-4)

The same construction, of lowest common ancestors in a Cartesian tree, makes it possible to construct a data structure with linear space that allows the distances between pairs of points in any [ultrametric space](http://en.wikipedia.org/wiki/Ultrametric_space) to be queried in constant time per query. The distance within an ultrametric is the same as the [minimax path](http://en.wikipedia.org/wiki/Widest_path_problem) weight in the [minimum spanning tree](http://en.wikipedia.org/wiki/Minimum_spanning_tree) of the metric.[[5]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-5) From the minimum spanning tree, one can construct a Cartesian tree, the root node of which represents the heaviest edge of the minimum spanning tree. Removing this edge partitions the minimum spanning tree into two subtrees, and Cartesian trees recursively constructed for these two subtrees form the children of the root node of the Cartesian tree. The leaves of the Cartesian tree represent points of the metric space, and the lowest common ancestor of two leaves in the Cartesian tree is the heaviest edge between those two points in the minimum spanning tree, which has weight equal to the distance between the two points. Once the minimum spanning tree has been found and its edge weights sorted, the Cartesian tree may be constructed in linear time.[[6]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-6)

**Treaps**

*Main article:* [*Treap*](http://en.wikipedia.org/wiki/Treap)

Because a Cartesian tree is a binary tree, it is natural to use it as a [binary search tree](http://en.wikipedia.org/wiki/Binary_search_tree) for an ordered sequence of values. However, defining a Cartesian tree based on the same values that form the search keys of a binary search tree does not work well: the Cartesian tree of a sorted sequence is just a [path](http://en.wikipedia.org/wiki/Path_graph), rooted at its leftmost endpoint, and binary searching in this tree degenerates to [sequential search](http://en.wikipedia.org/wiki/Sequential_search) in the path. However, it is possible to generate more-balanced search trees by generating *priority* values for each search key that are different than the key itself, sorting the inputs by their key values, and using the corresponding sequence of priorities to generate a Cartesian tree. This construction may equivalently be viewed in the geometric framework described above, in which the *x*-coordinates of a set of points are the search keys and the *y*-coordinates are the priorities.

This idea was applied by [Seidel & Aragon (1996](http://en.wikipedia.org/wiki/Cartesian_tree#CITEREFSeidelAragon1996)), who suggested the use of random numbers as priorities. The data structure resulting from this random choice is called a [treap](http://en.wikipedia.org/wiki/Treap), due to its combination of binary search tree and binary heap features. An insertion into a treap may be performed by inserting the new key as a leaf of an existing tree, choosing a priority for it, and then performing [tree rotation](http://en.wikipedia.org/wiki/Tree_rotation) operations along a path from the node to the root of the tree to repair any violations of the heap property caused by this insertion; a deletion may similarly be performed by a constant amount of change to the tree followed by a sequence of rotations along a single path in the tree.

If the priorities of each key are chosen randomly and independently once whenever the key is inserted into the tree, the resulting Cartesian tree will have the same properties as a [random binary search tree](http://en.wikipedia.org/wiki/Random_binary_search_tree), a tree computed by inserting the keys in a randomly chosen [permutation](http://en.wikipedia.org/wiki/Permutation) starting from an empty tree, with each insertion leaving the previous tree structure unchanged and inserting the new node as a leaf of the tree. Random binary search trees had been studied for much longer, and are known to behave well as search trees (they have [logarithmic](http://en.wikipedia.org/wiki/Logarithm) depth with high probability); the same good behavior carries over to treaps. It is also possible, as suggested by Aragon and Seidel, to reprioritize frequently-accessed nodes, causing them to move towards the root of the treap and speeding up future accesses for the same keys.

**Efficient construction**

A Cartesian tree may be constructed in [linear time](http://en.wikipedia.org/wiki/Linear_time) from its input sequence. One method is to simply process the sequence values in left-to-right order, maintaining the Cartesian tree of the nodes processed so far, in a structure that allows both upwards and downwards traversal of the tree. To process each new value *x*, start at the node representing the value prior to *x* in the sequence and follow the path from this node to the root of the tree until finding a value *y* smaller than *x*. This node *y* is the parent of *x*, and the previous right child of *y* becomes the new left child of *x*. The total time for this procedure is linear, because the time spent searching for the parent *y* of each new node *x* can be [charged](http://en.wikipedia.org/wiki/Potential_method) against the number of nodes that are removed from the rightmost path in the tree.[[4]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-gbt-4)

An alternative linear-time construction algorithm is based on the [all nearest smaller values](http://en.wikipedia.org/wiki/All_nearest_smaller_values) problem. In the input sequence, one may define the *left neighbor* of a value *x* to be the value that occurs prior to *x*, is smaller than *x*, and is closer in position to *x* than any other smaller value. The *right neighbor* is defined symmetrically. The sequence of left neighbors may be found by an algorithm that maintains a [stack](http://en.wikipedia.org/wiki/Stack_%28data_structure%29) containing a subsequence of the input. For each new sequence value *x*, the stack is popped until it is empty or its top element is smaller than *x*, and then *x* is pushed onto the stack. The left neighbor of *x* is the top element at the time *x* is pushed. The right neighbors may be found by applying the same stack algorithm to the reverse of the sequence. The parent of *x* in the Cartesian tree is either the left neighbor of *x* or the right neighbor of *x*, whichever exists and has a larger value. The left and right neighbors may also be constructed efficiently by [parallel algorithms](http://en.wikipedia.org/wiki/Parallel_algorithm), so this formulation may be used to develop efficient parallel algorithms for Cartesian tree construction.[[7]](http://en.wikipedia.org/wiki/Cartesian_tree#cite_note-7)

**Application in sorting**

Pairs of consecutive sequence values (shown as the thick red edges) that bracket a sequence value *x* (the darker blue point). The cost of including *x* in the sorted order produced by the Levcopoulos–Petersson algorithm is proportional to the [logarithm](http://en.wikipedia.org/wiki/Logarithm) of this number of bracketing pairs.

[Levcopoulos & Petersson (1989](http://en.wikipedia.org/wiki/Cartesian_tree#CITEREFLevcopoulosPetersson1989)) describe a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) based on Cartesian trees. They describe the algorithm as based on a tree with the maximum at the root, but it may be modified straightforwardly to support a Cartesian tree with the convention that the minimum value is at the root. For consistency, it is this modified version of the algorithm that is described below.

The Levcopoulos–Petersson algorithm can be viewed as a version of [selection sort](http://en.wikipedia.org/wiki/Selection_sort) or [heap sort](http://en.wikipedia.org/wiki/Heap_sort) that maintains a [priority queue](http://en.wikipedia.org/wiki/Priority_queue) of candidate minima, and that at each step finds and removes the minimum value in this queue, moving this value to the end of an output sequence. In their algorithm, the priority queue consists only of elements whose parent in the Cartesian tree has already been found and removed. Thus, the algorithm consists of the following steps:

1. Construct a Cartesian tree for the input sequence
2. Initialize a priority queue, initially containing only the tree root
3. While the priority queue is non-empty:
   * Find and remove the minimum value *x* in the priority queue
   * Add *x* to the output sequence
   * Add the Cartesian tree children of *x* to the priority queue

As Levcopoulos and Petersson show, for input sequences that are already nearly sorted, the size of the priority queue will remain small, allowing this method to take advantage of the nearly-sorted input and run more quickly. Specifically, the worst-case running time of this algorithm is O(*n* log *k*), where *k* is the average, over all values *x* in the sequence, of the number of consecutive pairs of sequence values that bracket *x*. They also prove a lower bound stating that, for any *n* and *k* = ω(1), any comparison-based sorting algorithm must use Ω(*n* log *k*) comparisons for some inputs.

# 5. Cocktail sort

|  |  |
| --- | --- |
| Cocktail sort | |
|  | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |

**Cocktail sort**, also known as **bidirectional bubble sort**, **cocktail shaker sort**, **shaker sort** (which can also refer to a variant of [selection sort](http://en.wikipedia.org/wiki/Selection_sort)), **ripple sort**, **shuffle sort**,[[1]](http://en.wikipedia.org/wiki/Cocktail_sort#cite_note-Duhl1986-0) **shuttle sort** or **happy hour sort**, is a variation of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) that is both a [stable](http://en.wikipedia.org/wiki/Stable_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) and a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). The algorithm differs from a [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) in that it sorts in both directions on each pass through the list. This sorting algorithm is only marginally more difficult to implement than a bubble sort, and solves the problem of [turtles](http://en.wikipedia.org/wiki/Bubble_sort#Rabbits_and_turtles) in bubble sorts.

|  |
| --- |
|  |

**Pseudocode**

The simplest form of cocktail sort goes through the whole list each time:

|  |
| --- |
|  |

**procedure** cocktailSort( A **:** list of sortable items ) **defined as:**

**do**

swapped := false

**for each** i **in** 0 **to** length( A ) - 2 **do:**

**if** A[ i ] > A[ i + 1 ] **then** // test whether the two elements are in the wrong order

swap( A[ i ], A[ i + 1 ] ) // let the two elements change places

swapped := true

**end if**

**end for**

**if** swapped = false **then**

// we can exit the outer loop here if no swaps occurred.

**break do-while loop**

**end if**

swapped := false

**for each** i **in** length( A ) - 2 **to** 0 **do:**

**if** A[ i ] > A[ i + 1 ] **then**

swap( A[ i ], A[ i + 1 ] )

swapped := true

**end if**

**end for**

**while** swapped // if no elements have been swapped, then the list is sorted

**end procedure**

The first rightward pass will shift the largest element to its correct place at the end, and the following leftward pass will shift the smallest element to its correct place at the beginning. The second complete pass will shift the second largest and second smallest elements to their correct places, and so on. After *i* passes, the first *i* and the last *i* elements in the list are in their correct positions, and do not need to be checked. By shortening the part of the list that is sorted each time, the number of operations can be halved (see [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort#Alternative_implementations)).

**procedure** cocktailSort( A **:** list of sortable items ) **defined as:**

// `begin` and `end` marks the first and last index to check

begin := -1

end := length( A ) - 2

**do**

swapped := false

// increases `begin` because the elements before `begin` are in correct order

begin := begin + 1

**for each** i **in** begin **to** end **do:**

**if** A[ i ] > A[ i + 1 ] **then**

swap( A[ i ], A[ i + 1 ] )

swapped := true

**end if**

**end for**

**if** swapped = false **then**

**break do-while loop**

**end if**

swapped := false

// decreases `end` because the elements after `end` are in correct order

end := end - 1

**for each** i **in** end **to** begin **do:**

**if** A[ i ] > A[ i + 1 ] **then**

swap( A[ i ], A[ i + 1 ] )

swapped := true

**end if**

**end for**

**while** swapped

**end procedure**

**Differences from bubble sort**

Cocktail sort is a slight variation of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort). It differs in that instead of repeatedly passing through the list from bottom to top, it passes alternately from bottom to top and then from top to bottom. It can achieve slightly better performance than a standard bubble sort. The reason for this is that [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) only passes through the list in one direction and therefore can only move items backward one step each iteration.

An example of a list that proves this point is the list (2,3,4,5,1), which would only need to go through one pass of cocktail sort to become sorted, but if using an ascending [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) would take four passes. However one cocktail sort pass should be counted as two bubble sort passes. Typically cocktail sort is less than two times faster than bubble sort.

Another optimization can be that the algorithm remembers where the last actual swap has been done. In the next iteration, there will be no swaps beyond this limit and the algorithm has shorter passes. As the Cocktail sort goes bidirectionally, the range of possible swaps, which is the range to be tested, will reduce per pass, thus reducing the overall running time.

**Complexity**

The complexity of cocktail sort in [big O notation](http://en.wikipedia.org/wiki/Big_O_notation) is for both the worst case and the average case, but it becomes closer to if the list is mostly ordered before applying the sorting algorithm, for example, if every element is at a position that differs at most k (k ≥ 1) from the position it is going to end up in, the complexity of cocktail sort becomes .

Cocktail sort is also briefly discussed in the book [*The Art of Computer Programming*](http://en.wikipedia.org/wiki/The_Art_of_Computer_Programming), along with similar refinements of bubble sort. In conclusion, Knuth states about bubble sort and its improvements (Knuth 1998, p. 110):

But none of these refinements leads to an algorithm better than straight insertion [that is, [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort)]; and we already know that straight insertion isn't suitable for large *N*. [...] In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

—D. E. Knuth[[2]](http://en.wikipedia.org/wiki/Cocktail_sort" \l "cite_note-1)

# 6. Comb sort

|  |  |
| --- | --- |
| Comb sort | |
|  | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | [[1]](http://en.wikipedia.org/wiki/Comb_sort#cite_note-1) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) |  |

**Comb sort** is a relatively simplistic [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) originally designed by [Włodzimierz Dobosiewicz](http://en.wikipedia.org/w/index.php?title=W%C5%82odzimierz_Dobosiewicz&action=edit&redlink=1) in 1980. Later it was rediscovered and popularized by [Stephen Lacey](http://en.wikipedia.org/w/index.php?title=Stephen_Lacey_%28computer_scientist%29&action=edit&redlink=1) and [Richard Box](http://en.wikipedia.org/w/index.php?title=Richard_Box&action=edit&redlink=1) with a [Byte Magazine](http://en.wikipedia.org/wiki/Byte_Magazine) [article published in April 1991](http://cs.clackamas.cc.or.us/molatore/cs260Spr03/combsort.htm). Comb sort improves on [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), and rivals algorithms like [Quicksort](http://en.wikipedia.org/wiki/Quicksort) ([visual comparison](http://scripts.franciscocharrua.com/javascript/sort-algorithms/compare.php?comb&quick)). The basic idea is to eliminate *turtles*, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. *Rabbits*, large values around the beginning of the list, do not pose a problem in bubble sort.

In bubble sort, when any two elements are compared, they always have a *gap* (distance from each other) of 1. The basic idea of comb sort is that the gap can be much more than 1 ([Shell sort](http://en.wikipedia.org/wiki/Shell_sort) is also based on this idea, but it is a modification of [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) rather than bubble sort).

In other words, the inner loop of [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), which does the actual *swap*, is modified such that gap between swapped elements goes down (for each iteration of outer loop) in steps of shrink factor. i.e. [ input size / shrink factor, input size / shrink factor^2, input size / shrink factor^3, .... , 1 ]. Unlike in [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), where the gap is constant i.e. 1.

The gap starts out as the length of the list being sorted divided by the *shrink factor* (generally 1.3; see below), and the list is sorted with that value (rounded down to an integer if needed) as the gap. Then the gap is divided by the shrink factor again, the list is sorted with this new gap, and the process repeats until the gap is 1. At this point, comb sort continues using a gap of 1 until the list is fully sorted. The final stage of the sort is thus equivalent to a bubble sort, but by this time most turtles have been dealt with, so a bubble sort will be efficient.

|  |
| --- |
|  |

**Shrink factor**

The shrink factor has a great effect on the efficiency of comb sort. In the original article, the author suggested . A value too small slows the algorithm down because more comparisons must be made, whereas a value too large means that no comparisons will be made. Text[*[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)*] describes an improvement to comb sort using the base value as the shrink factor (where is the [golden ratio](http://en.wikipedia.org/wiki/Golden_ratio)). It also contains a pseudocode implementation with a pre-defined gap table.

**Variations**

**Combsort11**

With a shrink factor around 1.3, there are only three possible ways for the list of gaps to end: (9, 6, 4, 3, 2, 1), (10, 7, 5, 3, 2, 1), or (11, 8, 6, 4, 3, 2, 1). Experiment shows that significant speed improvements can be made if the gap is set to 11 whenever it would otherwise become 9 or 10. This variation is called Combsort11.

If either of the sequences beginning with 9 or 10 were used, the final pass with a gap of 1 is less likely to completely sort the data, necessitating another pass with a gap of 1. The data is sorted when no swaps were done during a pass with *gap* = 1.

It is also possible to use a predefined table, to choose which gaps to use every pass.

**Combsort with different end**

Like many other sort efficient algorithms (like [quick sort](http://en.wikipedia.org/wiki/Quick_sort) or [merge sort](http://en.wikipedia.org/wiki/Merge_sort)), combsort is more effective in its earlier passes than it is during the final passes, when it resembles a [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort). Combsort can be made more effective if the sorting method is changed once the gaps reach numbers small enough. For example, once the gap reaches a size of about 10 or smaller, stopping the combsort and doing a simple [gnome sort](http://en.wikipedia.org/wiki/Gnome_sort) or [cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort), or, even better, an [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), will increase the sort's overall efficiency.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

Another advantage of this method is that there is no need to keep track of swaps during the sort passes to know if the sort should stop or not.

**Examples**

**Pseudocode**

**function** combsort(**array** input)

gap := input.size //initialize gap size

**loop until** gap = 1 **and** swapped = false

//update the gap value for a next comb. Below is an example

gap := int(gap / 1.247330950103979)

**if** gap < 1

//minimum gap is 1

gap := 1

**end if**

i := 0

swapped := false //see [bubblesort](http://en.wikipedia.org/wiki/Bubblesort) for an explanation

//a single "comb" over the input list

**loop until** i + gap >= input.size //see [shellsort](http://en.wikipedia.org/wiki/Shellsort) for similar idea

**if** input[i] > input[i+gap]

[swap](http://en.wikipedia.org/wiki/Swap_%28computer_science%29)(input[i], input[i+gap])

swapped := true // Flag a swap has occurred, so the

// list is not guaranteed sorted

**end if**

i := i + 1

**end loop**

**end loop**

**end function**

**C**

void comb\_sort(int \*input, size\_t size) {

int swap;

size\_t i, gap = size;

bool swapped = false;

while ((gap > 1) || swapped) {

if (gap > 1) {

gap = (size\_t)((double)gap / 1.247330950103979);

}

swapped = false;

for (i = 0; gap + i < size; ++i) {

if (input[i] - input[i + gap] > 0) {

swap = input[i];

input[i] = input[i + gap];

input[i + gap] = swap;

swapped = true;

}

}

}

}

# 7. Cycle sort

|  |  |
| --- | --- |
| Cycle sort | |
| Example of cycle sort sorting a list of random numbers. | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | Θ(*n*2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | Θ(*n*2) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | Θ(*n*2) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | Θ(*n*) total, Θ(*1*) auxiliary |

**Cycle sort** is an in-place, unstable [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm), a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) that is theoretically optimal in terms of the total number of writes to the original [array](http://en.wikipedia.org/wiki/Array_data_structure), unlike any other in-place sorting algorithm. It is based on the idea that the [permutation](http://en.wikipedia.org/wiki/Permutation) to be sorted can be factored into [cycles](http://en.wikipedia.org/wiki/Cycle_%28mathematics%29), which can individually be rotated to give a sorted result.

Unlike nearly every other sort, items are *never* written elsewhere in the array simply to push them out of the way of the action. Each value is either written zero times, if it's already in its correct position, or written one time to its correct position. This matches the minimal number of overwrites required for a completed in-place sort.

Minimizing the number of writes is useful when making writes to some huge data set is very expensive, such as with [EEPROMs](http://en.wikipedia.org/wiki/EEPROM) or [Flash memory](http://en.wikipedia.org/wiki/Flash_memory) where each write reduces the lifespan of the memory.

**Algorithm**

The following [algorithm](http://en.wikipedia.org/wiki/Algorithm) finds cycles and rotates them, giving a sorted result. Note that **range(*a*, *b*)** goes from ***a*** to ***b* ‑ 1**.

# Sort an array in place and return the number of writes.

def cycleSort(array):

writes = 0

# Loop through the array to find cycles to rotate.

for cycleStart in range(0, len(array) - 1):

item = array[cycleStart]

# Find where to put the item.

pos = cycleStart

for i in range(cycleStart + 1, len(array)):

if array[i] < item:

pos += 1

# If the item is already there, this is not a cycle.

if pos == cycleStart:

continue

# Otherwise, put the item there or right after any duplicates.

while item == array[pos]:

pos += 1

array[pos], item = item, array[pos]

writes += 1

# Rotate the rest of the cycle.

while pos != cycleStart:

# Find where to put the item.

pos = cycleStart

for i in range(cycleStart + 1, len(array)):

if array[i] < item:

pos += 1

# Put the item there or right after any duplicates.

while item == array[pos]:

pos += 1

array[pos], item = item, array[pos]

writes += 1

return writes

**Specific-situation optimizations**

When the array contains only duplicates of a relatively small number of items, a [constant-time](http://en.wikipedia.org/wiki/Time_complexity#Constant_time) [perfect hash function](http://en.wikipedia.org/wiki/Perfect_hash_function) can greatly speed up finding where to put an item[1](http://en.wikipedia.org/wiki/Cycle_sort#endnote_origpaper), turning the sort from Θ(*n*2) time to Θ(*n* + *k*) time, where *k* is the total number of hashes. The array ends up sorted in the order of the hashes, so choosing a hash function that gives you the right ordering is important.

Before the sort, create a [histogram](http://en.wikipedia.org/wiki/Histogram), sorted by hash, counting the number of occurrences of each hash in the array. Then create a table with the cumulative sum of each entry in the histogram. The cumulative sum table will then contain the position in the array of each element. The proper place of elements can then be found by a constant-time hashing and cumulative sum table lookup rather than a [linear search](http://en.wikipedia.org/wiki/Linear_search).

# 8. Flashsort

**Flashsort** is a [distribution sorting](http://en.wikipedia.org/wiki/Sorting_algorithm#Distribution_sort) algorithm showing [linear computational complexity [O(n)](http://en.wikipedia.org/wiki/O_notation)](http://en.wikipedia.org/wiki/O_notation)for uniformly distributed data sets and relatively little additional memory requirement. The original work was published in 1998 by Karl-Dietrich Neubert.[[1]](http://en.wikipedia.org/wiki/Flashsort#cite_note-neubert_journal-0)

|  |
| --- |
|  |

## Concept

The basic idea behind flashsort is that in a data set with a known [distribution](http://en.wikipedia.org/wiki/Probability_distribution), it is easy to immediately estimate where an element should be placed after sorting when the range of the set is known. For example, if given a uniform data set where the minimum is 1 and the maximum is 100 and 50 is an element of the set, it’s reasonable to guess that 50 would be near the middle of the set after it is sorted. This approximate location is called a class. If numbered 1 to m, the class of an item A_iis computed as:

K(A_i) = 1+\textrm{INT}\left((m-1)\frac{A_i-A_\textrm{min}}{A_\textrm{max}-A_\textrm{min}}\right)

where Ais the input set. The range covered by every class is equal, except the last class which includes only the maximum(s). The classification ensures that every element in a class is greater than any element in a lower class. This partially orders the data and reduces the number of inversions. Insertion sort is then applied to the classified set. As long as the data is uniformly distributed, class sizes will be consistent and insertion sort will be computationally efficient.[[1]](http://en.wikipedia.org/wiki/Flashsort#cite_note-neubert_journal-0)

## Memory efficient implementation

To execute flashsort with its low memory benefits, the algorithm does not use additional data structures to store the classes. Instead it stores the upper bounds of each class on the input array Ain an auxiliary vector L. These upper bounds are obtained by counting the number of elements in each class, and the upper bound of a class is the number of elements in that class and every class before it. These bounds serve as pointers into the classes.

Classification is implemented through a series of cycles, where a cycle-leader is taken from the input array Aand its class is calculated. The pointers in vector Lare used to insert the cycle-leader into the correct class, and the class’s pointer in Lis decremented after each insertion. Inserting the cycle-leader will evict another element from array A, which will be classified and inserted into another location and so on. The cycle terminates when an element is inserted into the cycle-leader’s starting location.

An element is a valid cycle-leader if it has not yet been classified. As the algorithm iterates on array A, previously classified elements are skipped and unclassified elements are used to initiate new cycles. It is possible to discern whether an element has been classified or not without using additional tags: An element has been classified if and only if its index is greater than the class’s pointer value in L. To prove this, consider the current index of array Athe algorithm is processing. Let this index be i. Elements A_0through A_\textrm{i-1}have already been classified and inserted into the correct class. Suppose that iis greater than the current pointer to A_i’s class. Now suppose that the A_iis unclassified and could be legally inserted into the index indicated by its class pointer, which would replace a classified element in another class. This is impossible since the initial pointers of each class are their upper bounds, which ensures that the exact needed amount of space is allocated for each class on the array A. Therefore, every element in A_i’s class, including A_iitself, has already been classified. Also, if an element has already been classified, the class’s pointer would have been decremented below the element’s new index.[[1]](http://en.wikipedia.org/wiki/Flashsort#cite_note-neubert_journal-0)[[2]](http://en.wikipedia.org/wiki/Flashsort#cite_note-neubert_code-1)

## Performance

The only extra memory requirements are the auxiliary vector Lfor storing class bounds and the constant number of other variables used.

In the ideal case of a balanced data set, each class will be approximately the same size, and sorting an individual class by itself has complexity O(1). If the number mof classes is proportional to the input set size n, the running time of the final insertion sort is m \cdot O(1) = O(m) = O(n). In the worst-case scenarios where almost all the elements are in a few or one class, the complexity of the algorithm as a whole is limited by the performance of the final-step sorting method. For insertion sort, this is O(n^2). Variations of the algorithm improve worst-case performance by using better-performing sorts such as quicksort or recursive flashsort on classes that exceed a certain size limit.[[2]](http://en.wikipedia.org/wiki/Flashsort#cite_note-neubert_code-1)[[3]](http://en.wikipedia.org/wiki/Flashsort#cite_note-2)

Choosing a value for m, the number of classes, trades off time spent classifying elements (high m) and time spent in the final insertion sort step (low m). Based on his research, Neubert found m=0.42nto be optimal.

Memory-wise, flashsort avoids the overhead needed to store classes in the very similar bucketsort. For m=0.1nwith uniform random data, flashsort is faster than heapsort for all nand faster than quicksort for n>80. It becomes about as twice as fast as quicksort at n=10000.[[1]](http://en.wikipedia.org/wiki/Flashsort#cite_note-neubert_journal-0)

Due to the classification process, flashsort is not [stable](http://en.wikipedia.org/wiki/Stable_sort#Stability).

# 9. Gnome sort

|  |  |
| --- | --- |
| Gnome sort | |
| Visualisation of gnome sort. | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(1)auxiliary |

**Gnome sort (Stupid sort)**, originally proposed by [Hamid Sarbazi-Azad](http://en.wikipedia.org/w/index.php?title=Hamid_Sarbazi-Azad&action=edit&redlink=1) in 2000 and called [Stupid sort](http://sina.sharif.edu/%7Eazad/stupid-sort.PDF) (not to be confused with [Bogosort](http://en.wikipedia.org/wiki/Bogosort)), and then later on described by [Dick Grune](http://en.wikipedia.org/wiki/Dick_Grune) and named "Gnome sort",[[1]](http://en.wikipedia.org/wiki/Gnome_sort#cite_note-1) is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) which is similar to [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), except that moving an element to its proper place is accomplished by a series of swaps, as in [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort). It is conceptually simple, requiring no nested loops. The running time is [O](http://en.wikipedia.org/wiki/Big_O_notation)(n^2), but tends towards O(*n*) if the list is initially almost sorted.[[2]](http://en.wikipedia.org/wiki/Gnome_sort#cite_note-2) In practice the algorithm can run as fast as [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort)[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]. The average runtime is O(n^2).

The algorithm always finds the first place where two adjacent elements are in the wrong order, and swaps them. It takes advantage of the fact that performing a swap can introduce a new out-of-order adjacent pair only right before or after the two swapped elements. It does not assume that elements forward of the current position are sorted, so it only needs to check the position directly before the swapped elements.

|  |
| --- |
|  |

## Description

Here is [pseudocode](http://en.wikipedia.org/wiki/Pseudocode) for the gnome sort using a [zero-based array](http://en.wikipedia.org/wiki/Array_data_type#Index_origin):

procedure gnomeSort(a[])

pos := 1

while pos < length(a)

if (a[pos] >= a[pos-1])

pos := pos + 1

else

swap a[pos] and a[pos-1]

if (pos > 1)

pos := pos - 1

else

pos := pos + 1

end if

end if

end while

end procedure

### Example

Given an unsorted array, a = [5, 3, 2, 4], the gnome sort would take the following steps during the while loop. The "current position" is highlighted in **bold**:

|  |  |
| --- | --- |
| **Current array** | **Action to take** |
| [5, **3**, 2, 4] | a[pos] < a[pos-1], swap: |
| [3, **5**, 2, 4] | a[pos] >= a[pos-1], increment pos: |
| [3, 5, **2**, 4] | a[pos] < a[pos-1], swap and pos > 1, decrement pos: |
| [3, **2**, 5, 4] | a[pos] < a[pos-1], swap and pos <= 1, increment pos: |
| [2, 3, **5**, 4] | a[pos] >= a[pos-1], increment pos: |
| [2, 3, 5, **4**] | a[pos] < a[pos-1], swap and pos > 1, decrement pos: |
| [2, 3, **4**, 5] | a[pos] >= a[pos-1], increment pos: |
| [2, 3, 4, **5**] | a[pos] >= a[pos-1], increment pos: |
| [2, 3, 4, 5] | pos == length(a), finished. |

## Optimization

The gnome sort may be optimized by introducing a variable to store the position before traversing back toward the beginning of the list. This would allow the "gnome" to [teleport](http://en.wikipedia.org/wiki/Teleportation) back to his previous position after moving a flower pot. With this optimization, the gnome sort would become a variant of the [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort).

Here is [pseudocode](http://en.wikipedia.org/wiki/Pseudocode) for an optimized gnome sort using a [zero-based array](http://en.wikipedia.org/wiki/Array_data_type#Index_origin):

procedure optimizedGnomeSort(a[])

pos := 1

last := 0

while pos < length(a)

if (a[pos] >= a[pos-1])

if (last != 0)

pos := last

last := 0

end if

pos := pos + 1

else

swap a[pos] and a[pos-1]

if (pos > 1)

if (last == 0)

last := pos

end if

pos := pos - 1

else

pos := pos + 1

end if

end if

end while

end procedure

## C++ Implementation

This implementation uses STL vector

#include <algorithm>

#include <vector>

template <typename T>

void gnomeSort(std::vector<T> &v)

{

typename std::vector<T>::size\_type i=1;

while (i < v.size())

{

if (v[i-1] <= v[i])

++i;

else

{

std::swap(v[i-1], v[i]);

if (i > 1)

--i;

}

}}

# 10. Introsort

|  |  |
| --- | --- |
| Introsort | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |

**Introsort** or **introspective sort** is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) designed by [David Musser](http://en.wikipedia.org/wiki/David_Musser) in 1997. It begins with [quicksort](http://en.wikipedia.org/wiki/Quicksort) and switches to [Heapsort](http://en.wikipedia.org/wiki/Heapsort) when the recursion depth exceeds a level based on (the [logarithm](http://en.wikipedia.org/wiki/Logarithm) of) the number of elements being sorted. It is the best of both worlds, with a worst-case [O](http://en.wikipedia.org/wiki/Big-O_notation)(*n* log *n*) runtime and practical performance comparable to quicksort on typical data sets. Since both algorithms it uses are [comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort), it too is a comparison sort.

In quicksort, one of the critical operations is choosing the pivot: the element around which the list is partitioned. The simplest pivot selection algorithm is to take the first or the last element of the list as the pivot, causing poor behavior for the case of sorted or nearly sorted input. [Niklaus Wirth](http://en.wikipedia.org/wiki/Niklaus_Wirth)'s variant uses the middle element to prevent these occurrences, degenerating to O(*n*²) for contrived sequences. The median-of-3 pivot selection algorithm takes the median of the first, middle, and last elements of the list; however, even though this performs well on many real-world inputs, it is still possible to contrive a *median-of-3 killer* list that will cause dramatic slowdown of a quicksort based on this pivot selection technique. Such inputs could potentially be exploited by an aggressor, for example by sending such a list to an Internet server for sorting as a [denial of service](http://en.wikipedia.org/wiki/Denial_of_service) attack.

Musser reported that on a median-of-3 killer sequence of 100,000 elements, introsort's running time was 1/200 that of median-of-3 quicksort. Musser also considered the effect on [caches](http://en.wikipedia.org/wiki/CPU_cache) of [Sedgewick](http://en.wikipedia.org/wiki/Robert_Sedgewick_%28computer_scientist%29)'s delayed small sorting, where small ranges are sorted at the end in a single pass of [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). He reported that it could double the number of cache misses, but that its performance with [double-ended queues](http://en.wikipedia.org/wiki/Double-ended_queue) was significantly better and should be retained for template libraries, in part because the gain in other cases from doing the sorts immediately was not great.

Similarly, Musser also introduced a worst-case linear [selection algorithm](http://en.wikipedia.org/wiki/Selection_algorithm) with time comparable to that of *Hoare's algorithm*, a simple adaptation of quicksort that is the most efficient selection algorithm used in practice. This is called introspection selection or [Introselect](http://en.wikipedia.org/wiki/Selection_algorithm#Introselect).

The June 2000 [SGI](http://en.wikipedia.org/wiki/Silicon_Graphics) C++ [Standard Template Library](http://en.wikipedia.org/wiki/Standard_Template_Library) [stl\_algo.h](http://www.sgi.com/tech/stl/stl_algo.h) implementation of [unstable sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Stability) uses the Musser introsort approach with the recursion depth to switch to heapsort passed as a parameter, median-of-3 pivot selection and the Sedgewick final insertion sort pass. The element threshold for switching to the simple insertion sort was 16.

# 11. Library sort

From Wikipedia, the free encyclopedia

Jump to: [navigation](http://en.wikipedia.org/wiki/Library_sort#mw-head), [search](http://en.wikipedia.org/wiki/Library_sort#p-search)

|  |  |
| --- | --- |
| Library sort | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |

**Library sort**, or **gapped insertion sort** is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) that uses an [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), but with gaps in the array to accelerate subsequent insertions. The name comes from an analogy:[[1]](http://en.wikipedia.org/wiki/Library_sort" \l "cite_note-0)

Suppose a librarian were to store his books alphabetically on a long shelf, starting with the A's at the left end, and continuing to the right along the shelf with no spaces between the books until the end of the Z's. If the librarian acquired a new book that belongs to the B section, once he finds the correct space in the B section, he will have to move every book over, from the middle of the B's all the way down to the Z's in order to make room for the new book. This is an insertion sort. However, if he were to leave a space after every letter, as long as there was still space after B, he would only have to move a few books to make room for the new one. This is the basic principle of the Library Sort.

The algorithm was proposed by [Michael A. Bender](http://en.wikipedia.org/w/index.php?title=Michael_A._Bender&action=edit&redlink=1), [Martín Farach-Colton](http://en.wikipedia.org/w/index.php?title=Mart%C3%ADn_Farach-Colton&action=edit&redlink=1), and [Miguel Mosteiro](http://en.wikipedia.org/w/index.php?title=Miguel_Mosteiro&action=edit&redlink=1) in 2006.[[2]](http://en.wikipedia.org/wiki/Library_sort#cite_note-1)

Like the insertion sort it is based on, library sort is a [stable](http://en.wikipedia.org/wiki/Stable_sort) [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) and can be run as an [online algorithm](http://en.wikipedia.org/wiki/Online_algorithm); however, it was shown to have a high probability of running in O(n log n) time (comparable to [quicksort](http://en.wikipedia.org/wiki/Quicksort)), rather than an insertion sort's O(n2). Its implementation is very similar to a [skip list](http://en.wikipedia.org/wiki/Skip_list). The drawback to using the library sort is that it requires extra space[*[specify](http://en.wikipedia.org/wiki/Wikipedia:Citing_sources" \o "Wikipedia:Citing sources)*] for the gaps.

# 12. Odd–even sort

|  |  |
| --- | --- |
| Odd–even sort | |
| Example of odd-even transposition sort sorting a list of random numbers. | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |

In computing, an **odd–even sort** or **odd–even transposition sort** (also known as **brick sort**[[1]](http://en.wikipedia.org/wiki/Odd%E2%80%93even_sort" \l "cite_note-1)) is a relatively simple [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm), developed originally for use on parallel processors with local interconnections. It is a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) related to [bubble sort](http://en.wikipedia.org/wiki/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.

|  |
| --- |
|  |

## Sorting on processor arrays

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]](http://en.wikipedia.org/wiki/Odd%E2%80%93even_sort#cite_note-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]](http://en.wikipedia.org/wiki/Odd%E2%80%93even_sort#cite_note-3)

## Batcher's odd–even mergesort

A related but more efficient sort algorithm is the [Batcher odd–even mergesort](http://en.wikipedia.org/wiki/Batcher_odd%E2%80%93even_mergesort), using compare–exchange operations and perfect-shuffle operations.[[4]](http://en.wikipedia.org/wiki/Odd%E2%80%93even_sort#cite_note-4) Batcher's method is efficient on parallel processors with long-range connections.[[5]](http://en.wikipedia.org/wiki/Odd%E2%80%93even_sort#cite_note-5)

## Algorithm

The single-processor algorithm, like [bubblesort](http://en.wikipedia.org/wiki/Bubblesort), is simple but not very efficient. Here a [zero-based](http://en.wikipedia.org/wiki/Zero-based) index is assumed:

/\* Assumes a is an array of values to be sorted. \*/

var sorted = false;

while(!sorted)

{

sorted=true;

for(var i = 1; i < list.length-1; i += 2)

{

if(a[i] > a[i+1])

{

swap(a, i, i+1);

sorted = false;

}

}

for(var i = 0; i < list.length-1; i += 2)

{

if(a[i] > a[i+1])

{

swap(a, i, i+1);

sorted = false;

}

}

}

# 13. Patience sorting

|  |  |
| --- | --- |
| Patience sorting | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n \log n) |

**Patience sorting** is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm), based on a [solitaire](http://en.wikipedia.org/wiki/Solitaire) [card game](http://en.wikipedia.org/wiki/Card_game), that has the property of being able to efficiently compute the length of a [longest increasing subsequence](http://en.wikipedia.org/wiki/Longest_increasing_subsequence) in a given [array](http://en.wikipedia.org/wiki/Array_data_structure).

|  |
| --- |
|  |

## The card game

The game begins with a [shuffled](http://en.wikipedia.org/wiki/Shuffle) deck of cards, labeled 1, 2, \ldots, n.

The cards are dealt one by one into a sequence of piles on the table, according to the following rules.

1. Initially, there are no piles. The first card dealt forms a new pile consisting of the single card.
2. Each new card may be placed either on an existing pile whose top card has a value higher than the new card's value, thus increasing the number of cards in that pile, or to the right of all of the existing piles, thus forming a new pile.
3. When there are no more cards remaining to deal, the game ends.

The object of the game is to finish with as few piles as possible. D. Aldous and P. Diaconis[[1]](http://en.wikipedia.org/wiki/Patience_sorting" \l "cite_note-Aldous-0) suggest defining 9 or fewer piles as a winning outcome for n = 52, which has approximately 5% chance to happen.

## Algorithm for sorting

Given an n-element array with an [ordering](http://en.wikipedia.org/wiki/Total_order) [relation](http://en.wikipedia.org/wiki/Binary_relation) as an input for the sorting, consider it as a collection of cards, with the (unknown in the beginning) statistical ordering of each element serving as its index. Note that the game never uses the actual value of the card, except for comparison between two cards, and the relative ordering of any two array elements is known.

Now simulate the patience sorting game, played with the [*greedy strategy*](http://en.wikipedia.org/wiki/Greedy_algorithm), i.e., placing each new card on the leftmost pile that is legally possible to use. At each stage of the game, under this strategy, the labels on the top cards of the piles are increasing from left to right. To recover the sorted sequence, repeatedly remove the minimum visible card.

### Complexity

If values of cards are in the range 1, \ldots, n, there is an efficient implementation with O(n \cdot \log \log n)[worst-case](http://en.wikipedia.org/wiki/Worst-case) running time for putting the cards into piles, relying on a [van Emde Boas tree](http://en.wikipedia.org/wiki/Van_Emde_Boas_tree). A description is given in the work by S. Bespamyatnikh and M. Segal.[[2]](http://en.wikipedia.org/wiki/Patience_sorting#cite_note-Bespamyatnikh-1)

When no assumption is made about values, the greedy strategy can be implemented in O(n \log n)comparisons in worst case. In fact, one can implement it with an array of [stacks](http://en.wikipedia.org/wiki/Stack_data_structure) ordered by values of top cards and, for inserting a new card, use a [binary search](http://en.wikipedia.org/wiki/Binary_search_algorithm), which is O(\log p)comparisons in worst case, where pis the number of piles. To complete the sorting in an efficient way (aka O(n \log n)worst case), each step will retrieve the card with the least value from the top of leftmost pile, and then some work has to be done. Finding the next card by searching it among all tops of piles, as in the wikibooks implementation suggested below, gives a O(n \sqrt n)worst case. However, we can use an efficient priority queue(for example, a binary heap) to maintain the piles so that we can extract the maximum data in O(log n) time.

## Algorithm for finding a longest increasing subsequence

First, execute the sorting algorithm as described above. The number of piles is the length of a longest subsequence. Whenever a card is placed on top of a pile, put a back-[pointer](http://en.wikipedia.org/wiki/Pointer_%28computer_programming%29) to the top card in the previous pile (that, by assumption, has a lower value than the new card has). In the end, follow the back-pointers from the top card in the last pile to recover a decreasing subsequence of the longest length; its reverse is an answer to the longest increasing subsequence algorithm.

S. Bespamyatnikh and M. Segal[[2]](http://en.wikipedia.org/wiki/Patience_sorting" \l "cite_note-Bespamyatnikh-1) give a description of an efficient implementation of the algorithm, incurring no additional [asymptotic](http://en.wikipedia.org/wiki/Asymptotic) cost over the sorting one (as the back-pointers storage, creation and traversal require linear time and space). They further show how to report *all* the longest increasing subsequences from the same resulting [data structures](http://en.wikipedia.org/wiki/Data_structure).

## C++ Implementation

This is an implementation using Patience Sorting to sort an array, performing O(n log n) time complexity.

#include <vector>

#include <algorithm>

#include <stack>

#include <iterator>

template<typename PileType>

bool pile\_less(const PileType& x, const PileType& y)

{

return x.top() < y.top();

}

// reverse less predicate to turn max-heap into min-heap

template<typename PileType>

bool pile\_more(const PileType& x, const PileType& y)

{

return pile\_less(y, x);

}

template<typename Iterator>

void patience\_sort(Iterator begin, Iterator end)

{

typedef typename std::iterator\_traits<Iterator>::value\_type DataType;

typedef std::stack<DataType> PileType;

std::vector<PileType> piles;

for (Iterator it = begin; it != end; it++)

{

PileType new\_pile;

new\_pile.push(\*it);

typename std::vector<PileType>::iterator insert\_it =

std::lower\_bound(piles.begin(), piles.end(), new\_pile,

pile\_less<PileType>);

if (insert\_it == piles.end())

piles.push\_back(new\_pile);

else

insert\_it->push(\*it);

}

// sorted array already satisfies heap property for min-heap

for (Iterator it = begin; it != end; it++)

{

std::pop\_heap(piles.begin(), piles.end(), pile\_more<PileType>);

\*it = piles.back().top();

piles.back().pop();

if (piles.back().empty())

piles.pop\_back();

else

std::push\_heap(piles.begin(), piles.end(), pile\_more<PileType>);

}

}

## Java Implementation

import java.util.\*;

public class PatienceSort

{

public static <E extends Comparable<? super E>> void sort (E[] n)

{

List<Pile<E>> piles = new ArrayList<Pile<E>>();

// sort into piles

for (E x : n)

{

Pile<E> newPile = new Pile<E>();

newPile.push(x);

int i = Collections.binarySearch(piles, newPile);

if (i < 0) i = ~i;

if (i != piles.size())

piles.get(i).push(x);

else

piles.add(newPile);

}

System.out.println("longest increasing subsequence has length = " + piles.size());

// priority queue allows us to retrieve least pile efficiently

PriorityQueue<Pile<E>> heap = new PriorityQueue<Pile<E>>(piles);

for (int c = 0; c < n.length; c++)

{

Pile<E> smallPile = heap.poll();

n[c] = smallPile.pop();

if (!smallPile.isEmpty())

heap.offer(smallPile);

}

assert(heap.isEmpty());

}

private static class Pile<E extends Comparable<? super E>> extends Stack<E> implements Comparable<Pile<E>>

{

public int compareTo(Pile<E> y) { return peek().compareTo(y.peek()); }

}

}

## History

According to D. Aldous and P. Diaconis,[[1]](http://en.wikipedia.org/wiki/Patience_sorting#cite_note-Aldous-0) patience sorting was first recognized as an algorithm to compute the longest increasing subsequence length by Hammersley,[[3]](http://en.wikipedia.org/wiki/Patience_sorting#cite_note-2) and by A.S.C. Ross and independently [Robert W. Floyd](http://en.wikipedia.org/wiki/Robert_W._Floyd) as a sorting algorithm. Initial analysis was done by Mallows.[[4]](http://en.wikipedia.org/wiki/Patience_sorting#cite_note-3)

## Use

The [Bazaar](http://en.wikipedia.org/wiki/Bazaar_%28software%29) version control system uses the patience sorting algorithm for merge resolution.

# 14. Polyphase merge sort

From Wikipedia, the free encyclopedia

Jump to: [navigation](http://en.wikipedia.org/wiki/Polyphase_merge_sort#mw-head), [search](http://en.wikipedia.org/wiki/Polyphase_merge_sort#p-search)

A **polyphase merge sort** is an algorithm which decreases the number of *runs* at every iteration of the main loop by merging runs into larger runs. It is used for [external sorting](http://en.wikipedia.org/wiki/External_sorting).

|  |
| --- |
|  |

## Ordinary merge sort

Typically, a [merge sort](http://en.wikipedia.org/wiki/Merge_sort) splits items into sorted runs and then recursively merges each run into larger runs. When there's only one run left, that is the sorted result.

An ordinary merge sort could use four working files organized as a pair of input files and a pair of output files. At each iteration, two input files are read. The odd numbered runs of the two input files are merged to the first output file, and the even numbered runs are merged to the second output file. When the input is exhausted, the new output files are used as the input for the next iteration. The number of runs decreases by a factor of 2 at each iteration. At each iteration, the same level/phase of merge occurs -- a file is either completely read or completely written during an iteration.

If the four files were on four separate [tape drives](http://en.wikipedia.org/wiki/Tape_drive), watching an ordinary merge sort would show some interesting details. On the first iteration, only one input drive is used -- the other input file is empty. On subsequent iterations, each input drive runs at half speed[[1]](http://en.wikipedia.org/wiki/Polyphase_merge_sort" \l "cite_note-0), while one output drive runs at full speed and the second output drive stands idle waiting for the next run. The situation is even worse when six tape drives are used -- at least two will stand idle. Someone watching the tapes spin would wonder if the idle drives could be more useful.

The polyphase merge found a way to use the idle drives. It can sort using just three sequential files rather than the four required by merge sort.

## Polyphase merge

The polyphase merge changes the game. There might be Nfiles, but the polyphase merge will read from N-1files and write only one output file at a time. The writing to that output file continues until an input file is exhausted, and then that input file becomes the new output file. The number of runs in each file is related to [Fibonacci numbers](http://en.wikipedia.org/wiki/Fibonacci_number) and [Fibonacci numbers of higher order](http://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers)[[2]](http://en.wikipedia.org/wiki/Polyphase_merge_sort#cite_note-Knuth1973-1)[[3]](http://en.wikipedia.org/wiki/Polyphase_merge_sort#cite_note-2).

## Perfect 3 file polyphase merge sort

It is easiest to look at the polyphase merge starting from its ending conditions and working backwards. At the start of each iteration, there will be two input files and one output file. At the end of the iteration, one input file will have been completely consumed and will become the output file for the next iteration. The current output file will become an input file for the next iteration. The remaining files (just one in the 3 file case) have only been partially consumed and their remaining runs will be input for the next iteration.

File 1 just emptied and became the new output file. One run is left on each input tape, and merging those runs together will make the sorted file.

File 1 (out): <1 run> \* (the sorted file)

File 2 (in ): ... | <1 run> \* --> ... <1 run> | \* (consumed)

File 3 (in ): | <1 run> \* <1 run> | \* (consumed)

... possible runs that have already been read

| marks the read pointer of the file

\* marks end of file

Stepping back to the previous iteration, we were reading from 1 and 2. One run is merged from 1 and 2 before file 1 goes empty. Notice that file 2 is not completely consumed -- it has one run left to match the final merge (above).

File 1 (in ): ... | <1 run> \* ... <1 run> | \*

File 2 (in ): | <2 run> \* --> <1 run> | <1 run> \*

File 3 (out): <1 run> \*

Stepping back another iteration, 2 runs are merged from 1 and 3 before file 3 goes empty.

File 1 (in ): | <3 run> ... <2 run> | <1 run> \*

File 2 (out): --> <2 run> \*

File 3 (in ): ... | <2 run> \* <2 run> | \*

Stepping back another iteration, 3 runs are merged from 2 and 3 before file 2 goes empty.

File 1 (out): <3 run> \*

File 2 (in ): ... | <3 run> \* --> ... <3 run> | \*

File 3 (in ): | <5 run> \* <3 run> | <2 run> \*

Stepping back another iteration, 5 runs are merged from 1 and 2 before file 1 goes empty.

File 1 (in ): ... | <5 run> \* ... <5 run> | \*

File 2 (in ): | <8 run> \* --> <5 run> | <3 run> \*

File 3 (out): <5 run> \*

Looking at the number of runs merged working backwards: 1, 1, 2, 3, 5, ... reveals a Fibonacci sequence.

For everything to work out right, the initial file to be sorted must be distributed to the proper input files and each input file must have the correct number of runs on it. In the example, that would mean an input file with 13 runs would write 5 runs to file 1 and 8 runs to file 2.

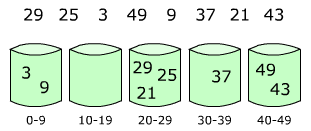
In practice, the input file won't happen to have a Fibonacci number of runs it (and the number of runs won't be known until after the file has been read). The fix is to pad the input files with dummy runs to make the required Fibonacci sequence.

For comparison, the ordinary merge sort will combine 16 runs in 4 passes using 4 files. The polyphase merge will combine 13 runs in 5 passes using only 3 files. Alternatively, a polyphase merge will combine 17 runs in 4 passes using 4 files. (Sequence: 1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355, 653, 1201, ...)

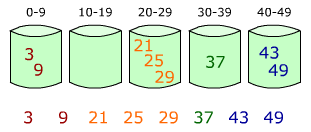
An iteration (or pass) in ordinary merge sort involves reading and writing the entire file. An iteration in a polyphase sort does not read or write the entire file[[4]](http://en.wikipedia.org/wiki/Polyphase_merge_sort" \l "cite_note-3), so a typical polyphase iteration will take less time than a merge sort iteration.

# 15. Proxmap sort

|  |  |
| --- | --- |
| [Insertion sorting into buckets during proxmap.](http://en.wikipedia.org/wiki/File:Insertion_Sorting_during_proxmap.PNG)  Example of insertion sort sorting a list of random numbers. | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n^2) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |

[](http://en.wikipedia.org/wiki/File:Bucket_sort_1.png)

Elements are distributed among bins

[](http://en.wikipedia.org/wiki/File:Bucket_sort_2.png)

Unlike bucket sorting which sorts after all the buckets are filled, the elements are [insertion sorted](http://en.wikipedia.org/wiki/Insertion_sort) as they are inserted

**ProxmapSort**, or **Proxmap sort**, is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) that works by partitioning an [array](http://en.wikipedia.org/wiki/Array_data_structure) of data items, or keys, into a number of "subarrays" (termed [buckets](http://en.wikipedia.org/wiki/Bucket_%28computing%29), in similar sorts). The name is short for computing a "proximity map," which indicates for each key K the beginning of a subarray where K will reside in the final sorted order. Keys are placed into each subarray using [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). If keys are "well distributed" among the subarrays, sorting occurs in linear time, much faster than [comparison-based](http://en.wikipedia.org/wiki/Comparison_sort) sorting, which can do no better than O(nlogn). The [computational complexity](http://en.wikipedia.org/wiki/Computational_complexity) estimates involve the number of subarrays and the proximity mapping function, the "map key," used. It is a form of [bucket](http://en.wikipedia.org/wiki/Bucket_sort) and [radix sort](http://en.wikipedia.org/wiki/Radix_sort). The algorithm scales up well as the number of data become large.

Once a ProxmapSort is complete, **ProxmapSearch** can be used to find keys in the sorted array in O(1)time if the keys were well distributed during the sort.

|  |
| --- |
|  |

## History

* Invented in the late 1980s by [Thomas A. Standish](http://www.ics.uci.edu/faculty/profiles/view_faculty.php?ucinetid=standish), Prof. Emeritus, Department of Informatics, [Donald Bren School of Information and Computer Sciences](http://en.wikipedia.org/wiki/Donald_Bren_School_of_Information_and_Computer_Sciences), University of California, Irvine.

## Overview

### Basic strategy

In general: Given an array **A** with *n* keys:

* map a key to a subarray of the destination array **A2**, by applying the map key function to each array item
* determine how many keys will map to the same subarray, using an array of **"hit counts," H**
* determine where each subarray will begin in the destination array so that each bucket is exactly the right size to hold all the keys that will map to it, using an array of **"proxmaps," P**
* for each key, compute the subarray it will map to, using an array of **"locations," L**
* for each key, look up its location, place it into that cell of **A2**; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key. Since the subarray is big enough to hold all the keys mapped to it, such movement will never cause the keys to overflow into the following subarray.

Simplied version: Given an array **A** with *n* keys

1. **Initialize**: Create and initialize 2 arrays of *n* size: **hitCount**, **proxMap**, and 2 arrays of **A**.length: **location**, and **A2**.
2. **Partition**: Using a carefully chosen **mapKey** function, divide the **A2** into subarrays using the keys in **A**
3. **Disperse**: Read over **A**, dropping each key into its bucket in **A2**; insertion sorting as needed.
4. **Collect**: Visit the subarrays in order and put all the elements back into the original array, or simply use **A2**.

Note: "keys" may also contain other data, for instance an array of Student objects that contain the key plus a student ID and name. This makes ProxMapSort suitable for organizing groups of objects, not just keys themselves.

### Example

Consider a full array: **A**[*0* to *n-1*] with *n* keys. Let *i* be an index of A. Sort **A'**s keys into array **A2** of equal size.

The map key function is defined as mapKey(key) = floor(K).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Array table | | | | | | | | | | | | | |
| **A1** | 6.7 | 5.9 | 8.4 | 1.2 | 7.3 | 3.7 | 11.5 | 1.1 | 4.8 | 0.4 | 10.5 | 6.1 | 1.8 |
| **H** | 1 | 3 | 0 | 1 | 1 | 1 | 2 | 1 | 1 | 0 | 1 | 1 |  |
| **P** | 0 | 1 | -9 | 4 | 5 | 6 | 7 | 9 | 10 | -9 | 11 | 12 |  |
| **L** | 7 | 6 | 10 | 1 | 9 | 4 | 12 | 1 | 5 | 0 | 11 | 7 | 1 |
| **A2** | 0.4 | 1.1 | 1.2 | 1.8 | 3.7 | 4.8 | 5.9 | 6.1 | 6.7 | 7.3 | 8.4 | 10.5 | 11.5 |

### Pseudocode

// compute hit counts

for i = 0 to 11 // where 11 is n

{

H[i] = 0;

}

for i = 0 to 12 // where 12 is A.length

{

pos = MapKey(A[i]);

H[pos] = H[pos] + 1;

}

runningTotal = 0; // compute prox map – location of start of each subarray

for i = 0 to 11

if H[i] = 0

P[i] = -9;

else

P[i] = runningTotal;

runningTotal = runningTotal + H[i];

for i = 0 to 12 // compute location – subarray – in A2 into which each item in A is to be placed

L[i] = P[MapKey(A[i])];

for I = 0 to 12; // sort items

A2[I] = <empty>;

for i = 0 to 12 // insert each item into subarray beginning at start, preserving order

{

start = L[i]; // subarray for this item begins at this location

insertion made = false;

for j = start to (<the end of A2 is found, and insertion not made>)

{

if A2[j] == <empty> // if subarray empty, just put item in first position of subarray

A2[j] = A[i];

insertion made = true;

else if A[i] < A2[j] // key belongs at A2[j]

int end = j + 1; // find end of used part of subarray – where first <empty> is

while (A2[end] != <empty>)

end++;

for k = end -1 to j // move larger keys to the right 1 cell

A2[k+1] = A2[k];

A2[j] = A[i];

insertion made = true; // add in new key

}

}

Here **A** is the array to be sorted and the mapKey functions determines the number of subarrays to use. For example, floor(K) will simply assign as many subarrays as there are integers from the data in **A**. Dividing the key by a constant reduces the number of subarrays; different functions can be used to translate the range of elements in **A** to subarrays, such as converting the letters A–Z to 0–25 or returning the first character (0–255) for sorting strings. Subarrays are sorted as the data comes in, not after all data has been placed into the subarray, as is typical in [bucket sorting](http://en.wikipedia.org/wiki/Bucket_sorting).

## Proxmap Searching

ProxmapSearch uses the **proxMap** array generated by a previously done ProxmapSort to find keys in the sorted array **A2** in constant time.

### Basic strategy

* Sort the keys using ProxmapSort, keeping the**MapKey** function, and the **P** and **A2** arrays
* To search for a key, go to P[MapKey(k)], the start of the subarray that contains the key, if that key is in the data set
* Sequentially search the subarray; if the key is found, return it (and associated information); if find a value greater than the key, the key is not in the data set
* Computing P[MapKey(k)] takes O(1)time. If a map key that gives a good distribution of keys was used during the sort, each subarray is bounded above by a constant *c*, so at most *c* comparisons are needed to find the key or know it is not present; therefore ProxmapSearch is O(1). If the worst map key was used, all keys are in the same subarray, so ProxmapSearch, in this worst case, will require O(n)comparisons.

### Pseudocode

**function** mapKey(key)

**return** floor(key)

proxMap ← previously generated proxmap array of size n

A2 ← previously sorted array of size n

**function** proxmap-search(key)

**for** i = proxMap[mapKey(key)] **to** length(array)-1

**if** (sortedArray[i].key == key)

**return** sortedArray[i]

## Analysis

### Performance

Computing H, P, and L all take O(n)time. Each is computed with one pass through an array, with constant time spent at each array location.

* Worst case: MapKey places all items into one subarray, resulting in a standard insertion sort, and time of O(n^2).
* Best case: MapKey delivers the same small number of items to each subarray in an order where the best case of insertion sort occurs. Each insertion sort is O(c), *c* the size of the subarrays; there are *p* subarrays thus **p \* c = n**, so the insertion phase take O(n); thus, ProxmapSort is O(n).
* Average case: Each subarray is at most size *c*, a constant; insertion sort for each subarray is then O(c^2) at worst – a constant. (The actual time can be much better, since c items are not sorted until the last item is placed in the bucket). Total time is the number of buckets, **(n/c)**, times O(c^2)= O(n).

Having a good MapKey function is imperative for avoiding the worst case. We must know something about the distribution of the data to come up with a good key.

### Optimizations

1. Save time: Save the MapKey(i) values so they don't have to be recomputed (as they are in the code above)
2. Save space: The proxMaps can be stored in the hitCount array,as the hit counts are not needed once the proxmap is computed; the data can be sorted back into A, instead of using A2, if one takes care to note which A values are have been sorted so far, and which not.

### Comparison with other sorting algorithms

Since ProxmapSort is not a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort), the Ω(*n* log *n*) lower bound is inapplicable. Its speed can be attributed to it not being comparison-based and using arrays instead of dynamically allocated objects and pointers that must be followed, such as is done with when using a [binary search tree](http://en.wikipedia.org/wiki/Binary_search_tree).

ProxmapSort allows for the use of ProxmapSearch. Despite the O(n) build time, ProxMapSearch makes up for it with its O(1)average access time, making it very appealing for large databases. If the data doesn't need to be updated often, the access time may make this function more favorable than other [non-comparison sorting](http://en.wikipedia.org/w/index.php?title=Non-comparison_sorting&action=edit&redlink=1) based sorts.

### Generic bucket sort related to ProxmapSort

Like ProxmapSort, bucket sort generally operates on a list of *n* numeric inputs between zero and some maximum key or value *M* and divides the value range into *n* buckets each of size *M*/*n*. If each bucket is sorted using [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), ProxmapSort and bucket sort can be shown to run in predicted linear time.[[1]](http://en.wikipedia.org/wiki/Proxmap_sort#cite_note-0) However, the performance of this sort degrades with clustering (or too few buckets with too many keys); if many values occur close together, they will all fall into a single bucket and performance will be severely diminished. This behavior also holds for ProxmapSort: if the buckets are too large, its performance will degrade severely.

# 16. Smoothsort

|  |  |
| --- | --- |
| Smoothsort | |
| [A run of the smoothsort algorithm sorting an array that is mainly in order but with a few out-of-sequence elements.](http://en.wikipedia.org/wiki/File:Smoothsort.gif) | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n)total, O(1)auxiliary |

**Smoothsort**[[1]](http://en.wikipedia.org/wiki/Smoothsort" \l "cite_note-1) (method) is a [comparison-based](http://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm). It is a variation of [heapsort](http://en.wikipedia.org/wiki/Heapsort) developed by [Edsger Dijkstra](http://en.wikipedia.org/wiki/Edsger_Dijkstra) in 1981. Like heapsort, smoothsort's upper bound is [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*). The advantage of smoothsort is that it comes closer to O(*n*) time if the [input is already sorted to some degree](http://en.wikipedia.org/wiki/Adaptive_sort), whereas heapsort averages O(*n* log *n*) regardless of the initial sorted state.

|  |
| --- |
|  |

## Overview

Like [heapsort](http://en.wikipedia.org/wiki/Heapsort), smoothsort builds up an implicit heap data structure in the array to be sorted, then sorts the array by continuously extracting the maximum element from that heap. Unlike heapsort, smoothsort does not use a [binary heap](http://en.wikipedia.org/wiki/Binary_heap), but rather a custom heap based on the [Leonardo numbers](http://en.wikipedia.org/wiki/Leonardo_numbers) L(n). The heap structure consists of a string of heaps, the sizes of which are all Leonardo numbers, and whose roots are stored in ascending order. The advantage of this custom heap over binary heaps is that if the sequence is already sorted, it takes only O(n)time to construct and deconstruct the heap, hence the better runtime.

Breaking the input up into a sequence of heaps is simple – the leftmost nodes of the array are made into the largest heap possible, and the remainder is likewise divided up. It can be proven [[2]](http://en.wikipedia.org/wiki/Smoothsort#cite_note-2) that:

* Any array of any length can so be divided up into sections of size L(x).
* No two heaps will have the same size. The string will therefore be a string of heaps strictly descending in size.
* No two heaps will have sizes that are consecutive Leonardo numbers, except for possibly the final two.

Each heap, having a size of L(x), is structured from left to right as a sub-heap of size L(x-1), a sub-heap of size L(x-2), and a root node, with the exception of heaps with a size of L(1) and L(0), which are singleton nodes. Each heap maintains the heap property that a root node is always at least as large as the root nodes of its child heaps (and therefore at least as large as all nodes in its child heaps), and the string of heaps as a whole maintains the string property that the root node of each heap is at least as large as the root node of the heap to the left.

The consequence of this is that the rightmost node in the string will always be the largest of the nodes, and, importantly, an array that is already sorted needs no rearrangement to be made into a valid series of heaps. This is the source of the adaptive qualities of the algorithm.

The algorithm is simple. We start by dividing up our unsorted array into a single heap of one element, followed by an unsorted portion. A one-element array is trivially a valid sequence of heaps. This sequence is then grown by adding one element at a time to the right, performing swaps to keep the sequence property and the heap property, until it fills the entire original array.

From this point on, the rightmost element of the sequence of heaps will be the largest element in any of the heaps, and will therefore be in its correct, final position. We then reduce the series of heaps back down to a single heap of one element by removing the rightmost node (which stays in place) and performing re-arrangements to restore the heap condition. When we are back down to a single heap of one element, the array is sorted.

## Operations

Ignoring (for the moment) Dijkstra's optimisations, two operations are necessary – increase the string by adding one element to the right, and decrease the string by removing the right most element (the root of the last heap), preserving the heap and string conditions.

### Grow the string by adding an element to the right

* If the last two heaps are of size L(x+1) and L(x) (i.e.: consecutive leonardo numbers), the new element becomes the root node of a bigger heap of size L(x+2). This heap will not necessarily have the heap property.
* If the last two heaps of the string are not consecutive Leonardo numbers, then the rightmost element becomes a new heap of size 1. This 1 is taken to be L(1), unless the rightmost heap already has size L(1), in which case the new one-element heap is taken to be of size L(0).

After this, the heap and string properties must be restored, which is usually done via a variant of [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). This is done as follows:

1. The rightmost heap (the one that has just been created) becomes the "current" heap
2. While there is a heap to the left of the current heap and its root is larger than the current root **and** both of its child heap roots
   * Then swap the new root with the root on the heap to the left (this will not disturb the heap property of the current heap). That heap then becomes the current heap.
3. Perform a "filter" operation on the current heap to establish the heap property:
   * While the current heap has a size greater than 1 and either child heap of the current heap has a root node greater than the root of the current heap
     + Swap the greater child root with the current root. That child heap becomes the current heap.

The filter operation is greatly simplified by the use of Leonardo numbers, as a heap will always either be a single node, or will have two children. One does not need to manage the condition of one of the child heaps not being present.

#### Optimisation

* If the new heap is going to become part of a larger heap by the time we are done, then don't bother establishing the string property: it only needs to be done when a heap has reached its final size.
  + To do this, look at how many elements are left after the new heap of size L(x). If there are more than L(x-1)+1, then this new heap is going to be merged.
* Do not maintain the heap property of the rightmost heap. If that heap becomes one of the final heaps of the string, then maintaining the string property will restore the heap property. Of course, whenever a new heap is created, then the rightmost heap is no longer the rightmost and the heap property needs to be restored.

### Shrink the string by removing the rightmost element

If the rightmost heap has a size of 1 (i.e., L(1) or L(0)), then nothing needs to be done. Simply remove that rightmost heap.

If the rightmost heap does not have a size of 1, then remove the root, exposing the two sub-heaps as members of the string. Restore the string property first on the left one and then on the right one.

#### Optimisation

* When restoring the string property, we do not need to compare the root of the heap to the left with the two child nodes of the heaps that have just been exposed, because we know that these newly exposed heaps have the heap property. Just compare it to the root.

## Memory usage

The smoothsort algorithm needs to be able to hold in memory the sizes of all of the heaps in the string. Since all these values are distinct, this is usually done using a [bit vector](http://en.wikipedia.org/wiki/Bit_vector). Moreover, since there are at most O(log n) numbers in the sequence, these bits can be encoded in O(1) machine words, assuming a [transdichotomous machine model](http://en.wikipedia.org/wiki/Transdichotomous_model).

## Java implementation

This code uses **lo** and **hi** as the bounds of the array *inclusive*. Note that this is not the usual convention. Further note, that this implementation is a little flawed: The bitmap p only holds 32 bits and therefore limits the maximum number of heaps in the entire string to 32 (not counting the right-most heap of size LP[0]), which will eventually overflow if you try to sort an array of more than LP[32]+1 = 7049156 elements. This can be solved by using a long-bitmap wide enough to hold a single bit for every Leonardo-number that could be used as an array index in Java, which is exactly one more than the index of the largest such number (i.e. the largest one that fits into a 32-bit signed integer, as commented on LP) and is therefore equal to the length of LP which is 43.

// by keeping these constants, we can avoid the tiresome business

// of keeping track of Dijkstra's b and c. Instead of keeping

// b and c, I will keep an index into this array.

static final int LP[] = { 1, 1, 3, 5, 9, 15, 25, 41, 67, 109,

177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, 13529, 21891,

35421, 57313, 92735, 150049, 242785, 392835, 635621, 1028457,

1664079, 2692537, 4356617, 7049155, 11405773, 18454929, 29860703,

48315633, 78176337, 126491971, 204668309, 331160281, 535828591,

866988873 // the next number is > 31 bits.

};

public static <C extends Comparable<? super C>> void sort(C[] m,

int lo, int hi) {

int head = lo; // the offset of the first element of the prefix into m

// These variables need a little explaining. If our string of heaps

// is of length 38, then the heaps will be of size 25+9+3+1, which are

// Leonardo numbers 6, 4, 2, 1.

// Turning this into a binary number, we get b01010110 = 0x56. We represent

// this number as a pair of numbers by right-shifting all the zeros and

// storing the mantissa and exponent as "p" and "pshift".

// This is handy, because the exponent is the index into L[] giving the

// size of the rightmost heap, and because we can instantly find out if

// the rightmost two heaps are consecutive Leonardo numbers by checking

// (p&3)==3

int p = 1; // the bitmap of the current standard concatenation >> pshift

int pshift = 1;

while (head < hi) {

if ((p & 3) == 3) {

// Add 1 by merging the first two blocks into a larger one.

// The next Leonardo number is one bigger.

sift(m, pshift, head);

p >>>= 2;

pshift += 2;

} else {

// adding a new block of length 1

if (LP[pshift - 1] >= hi - head) {

// this block is its final size.

trinkle(m, p, pshift, head, false);

} else {

// this block will get merged. Just make it trusty.

sift(m, pshift, head);

}

if (pshift == 1) {

// LP[1] is being used, so we add use LP[0]

p <<= 1;

pshift--;

} else {

// shift out to position 1, add LP[1]

p <<= (pshift - 1);

pshift = 1;

}

}

p |= 1;

head++;

}

trinkle(m, p, pshift, head, false);

while (pshift != 1 || p != 1) {

if (pshift <= 1) {

// block of length 1. No fiddling needed

int trail = Integer.numberOfTrailingZeros(p & ~1);

p >>>= trail;

pshift += trail;

} else {

p <<= 2;

p ^= 7;

pshift -= 2;

// This block gets broken into three bits. The rightmost

// bit is a block of length 1. The left hand part is split into

// two, a block of length LP[pshift+1] and one of LP[pshift].

// Both these two are appropriately heapified, but the root

// nodes are not necessarily in order. We therefore semitrinkle

// both of them

trinkle(m, p >>> 1, pshift + 1, head - LP[pshift] - 1, true);

trinkle(m, p, pshift, head - 1, true);

}

head--;

}

}

private static <C extends Comparable<? super C>> void sift(C[] m, int pshift,

int head) {

// we do not use Floyd's improvements to the heapsort sift, because we

// are not doing what heapsort does - always moving nodes from near

// the bottom of the tree to the root.

C val = m[head];

while (pshift > 1) {

int rt = head - 1;

int lf = head - 1 - LP[pshift - 2];

if (val.compareTo(m[lf]) >= 0 && val.compareTo(m[rt]) >= 0)

break;

if (m[lf].compareTo(m[rt]) >= 0) {

m[head] = m[lf];

head = lf;

pshift -= 1;

} else {

m[head] = m[rt];

head = rt;

pshift -= 2;

}

}

m[head] = val;

}

private static <C extends Comparable<? super C>> void trinkle(C[] m, int p,

int pshift, int head, boolean isTrusty) {

C val = m[head];

while (p != 1) {

int stepson = head - LP[pshift];

if (m[stepson].compareTo(val) <= 0)

break; // current node is greater than head. Sift.

// no need to check this if we know the current node is trusty,

// because we just checked the head (which is val, in the first

// iteration)

if (!isTrusty && pshift > 1) {

int rt = head - 1;

int lf = head - 1 - LP[pshift - 2];

if (m[rt].compareTo(m[stepson]) >= 0

|| m[lf].compareTo(m[stepson]) >= 0)

break;

}

m[head] = m[stepson];

head = stepson;

int trail = Integer.numberOfTrailingZeros(p & ~1);

p >>>= trail;

pshift += trail;

isTrusty = false;

}

if (!isTrusty) {

m[head] = val;

sift(m, pshift, head);

}

}

# 17. Smoothsort

|  |  |
| --- | --- |
| Smoothsort | |
| [A run of the smoothsort algorithm sorting an array that is mainly in order but with a few out-of-sequence elements.](http://en.wikipedia.org/wiki/File:Smoothsort.gif) | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n)total, O(1)auxiliary |

**Smoothsort**[[1]](http://en.wikipedia.org/wiki/Smoothsort" \l "cite_note-1) (method) is a [comparison-based](http://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm). It is a variation of [heapsort](http://en.wikipedia.org/wiki/Heapsort) developed by [Edsger Dijkstra](http://en.wikipedia.org/wiki/Edsger_Dijkstra) in 1981. Like heapsort, smoothsort's upper bound is [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*). The advantage of smoothsort is that it comes closer to O(*n*) time if the [input is already sorted to some degree](http://en.wikipedia.org/wiki/Adaptive_sort), whereas heapsort averages O(*n* log *n*) regardless of the initial sorted state.

|  |
| --- |
|  |

## Overview

Like [heapsort](http://en.wikipedia.org/wiki/Heapsort), smoothsort builds up an implicit heap data structure in the array to be sorted, then sorts the array by continuously extracting the maximum element from that heap. Unlike heapsort, smoothsort does not use a [binary heap](http://en.wikipedia.org/wiki/Binary_heap), but rather a custom heap based on the [Leonardo numbers](http://en.wikipedia.org/wiki/Leonardo_numbers) L(n). The heap structure consists of a string of heaps, the sizes of which are all Leonardo numbers, and whose roots are stored in ascending order. The advantage of this custom heap over binary heaps is that if the sequence is already sorted, it takes only O(n)time to construct and deconstruct the heap, hence the better runtime.

Breaking the input up into a sequence of heaps is simple – the leftmost nodes of the array are made into the largest heap possible, and the remainder is likewise divided up. It can be proven [[2]](http://en.wikipedia.org/wiki/Smoothsort#cite_note-2) that:

* Any array of any length can so be divided up into sections of size L(x).
* No two heaps will have the same size. The string will therefore be a string of heaps strictly descending in size.
* No two heaps will have sizes that are consecutive Leonardo numbers, except for possibly the final two.

Each heap, having a size of L(x), is structured from left to right as a sub-heap of size L(x-1), a sub-heap of size L(x-2), and a root node, with the exception of heaps with a size of L(1) and L(0), which are singleton nodes. Each heap maintains the heap property that a root node is always at least as large as the root nodes of its child heaps (and therefore at least as large as all nodes in its child heaps), and the string of heaps as a whole maintains the string property that the root node of each heap is at least as large as the root node of the heap to the left.

The consequence of this is that the rightmost node in the string will always be the largest of the nodes, and, importantly, an array that is already sorted needs no rearrangement to be made into a valid series of heaps. This is the source of the adaptive qualities of the algorithm.

The algorithm is simple. We start by dividing up our unsorted array into a single heap of one element, followed by an unsorted portion. A one-element array is trivially a valid sequence of heaps. This sequence is then grown by adding one element at a time to the right, performing swaps to keep the sequence property and the heap property, until it fills the entire original array.

From this point on, the rightmost element of the sequence of heaps will be the largest element in any of the heaps, and will therefore be in its correct, final position. We then reduce the series of heaps back down to a single heap of one element by removing the rightmost node (which stays in place) and performing re-arrangements to restore the heap condition. When we are back down to a single heap of one element, the array is sorted.

## Operations

Ignoring (for the moment) Dijkstra's optimisations, two operations are necessary – increase the string by adding one element to the right, and decrease the string by removing the right most element (the root of the last heap), preserving the heap and string conditions.

### Grow the string by adding an element to the right

* If the last two heaps are of size L(x+1) and L(x) (i.e.: consecutive leonardo numbers), the new element becomes the root node of a bigger heap of size L(x+2). This heap will not necessarily have the heap property.
* If the last two heaps of the string are not consecutive Leonardo numbers, then the rightmost element becomes a new heap of size 1. This 1 is taken to be L(1), unless the rightmost heap already has size L(1), in which case the new one-element heap is taken to be of size L(0).

After this, the heap and string properties must be restored, which is usually done via a variant of [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort). This is done as follows:

1. The rightmost heap (the one that has just been created) becomes the "current" heap
2. While there is a heap to the left of the current heap and its root is larger than the current root **and** both of its child heap roots
   * Then swap the new root with the root on the heap to the left (this will not disturb the heap property of the current heap). That heap then becomes the current heap.
3. Perform a "filter" operation on the current heap to establish the heap property:
   * While the current heap has a size greater than 1 and either child heap of the current heap has a root node greater than the root of the current heap
     + Swap the greater child root with the current root. That child heap becomes the current heap.

The filter operation is greatly simplified by the use of Leonardo numbers, as a heap will always either be a single node, or will have two children. One does not need to manage the condition of one of the child heaps not being present.

#### Optimisation

* If the new heap is going to become part of a larger heap by the time we are done, then don't bother establishing the string property: it only needs to be done when a heap has reached its final size.
  + To do this, look at how many elements are left after the new heap of size L(x). If there are more than L(x-1)+1, then this new heap is going to be merged.
* Do not maintain the heap property of the rightmost heap. If that heap becomes one of the final heaps of the string, then maintaining the string property will restore the heap property. Of course, whenever a new heap is created, then the rightmost heap is no longer the rightmost and the heap property needs to be restored.

### Shrink the string by removing the rightmost element

If the rightmost heap has a size of 1 (i.e., L(1) or L(0)), then nothing needs to be done. Simply remove that rightmost heap.

If the rightmost heap does not have a size of 1, then remove the root, exposing the two sub-heaps as members of the string. Restore the string property first on the left one and then on the right one.

#### Optimisation

* When restoring the string property, we do not need to compare the root of the heap to the left with the two child nodes of the heaps that have just been exposed, because we know that these newly exposed heaps have the heap property. Just compare it to the root.

## Memory usage

The smoothsort algorithm needs to be able to hold in memory the sizes of all of the heaps in the string. Since all these values are distinct, this is usually done using a [bit vector](http://en.wikipedia.org/wiki/Bit_vector). Moreover, since there are at most O(log n) numbers in the sequence, these bits can be encoded in O(1) machine words, assuming a [transdichotomous machine model](http://en.wikipedia.org/wiki/Transdichotomous_model).

## Java implementation

This code uses **lo** and **hi** as the bounds of the array *inclusive*. Note that this is not the usual convention. Further note, that this implementation is a little flawed: The bitmap p only holds 32 bits and therefore limits the maximum number of heaps in the entire string to 32 (not counting the right-most heap of size LP[0]), which will eventually overflow if you try to sort an array of more than LP[32]+1 = 7049156 elements. This can be solved by using a long-bitmap wide enough to hold a single bit for every Leonardo-number that could be used as an array index in Java, which is exactly one more than the index of the largest such number (i.e. the largest one that fits into a 32-bit signed integer, as commented on LP) and is therefore equal to the length of LP which is 43.

// by keeping these constants, we can avoid the tiresome business

// of keeping track of Dijkstra's b and c. Instead of keeping

// b and c, I will keep an index into this array.

static final int LP[] = { 1, 1, 3, 5, 9, 15, 25, 41, 67, 109,

177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, 13529, 21891,

35421, 57313, 92735, 150049, 242785, 392835, 635621, 1028457,

1664079, 2692537, 4356617, 7049155, 11405773, 18454929, 29860703,

48315633, 78176337, 126491971, 204668309, 331160281, 535828591,

866988873 // the next number is > 31 bits.

};

public static <C extends Comparable<? super C>> void sort(C[] m,

int lo, int hi) {

int head = lo; // the offset of the first element of the prefix into m

// These variables need a little explaining. If our string of heaps

// is of length 38, then the heaps will be of size 25+9+3+1, which are

// Leonardo numbers 6, 4, 2, 1.

// Turning this into a binary number, we get b01010110 = 0x56. We represent

// this number as a pair of numbers by right-shifting all the zeros and

// storing the mantissa and exponent as "p" and "pshift".

// This is handy, because the exponent is the index into L[] giving the

// size of the rightmost heap, and because we can instantly find out if

// the rightmost two heaps are consecutive Leonardo numbers by checking

// (p&3)==3

int p = 1; // the bitmap of the current standard concatenation >> pshift

int pshift = 1;

while (head < hi) {

if ((p & 3) == 3) {

// Add 1 by merging the first two blocks into a larger one.

// The next Leonardo number is one bigger.

sift(m, pshift, head);

p >>>= 2;

pshift += 2;

} else {

// adding a new block of length 1

if (LP[pshift - 1] >= hi - head) {

// this block is its final size.

trinkle(m, p, pshift, head, false);

} else {

// this block will get merged. Just make it trusty.

sift(m, pshift, head);

}

if (pshift == 1) {

// LP[1] is being used, so we add use LP[0]

p <<= 1;

pshift--;

} else {

// shift out to position 1, add LP[1]

p <<= (pshift - 1);

pshift = 1;

}

}

p |= 1;

head++;

}

trinkle(m, p, pshift, head, false);

while (pshift != 1 || p != 1) {

if (pshift <= 1) {

// block of length 1. No fiddling needed

int trail = Integer.numberOfTrailingZeros(p & ~1);

p >>>= trail;

pshift += trail;

} else {

p <<= 2;

p ^= 7;

pshift -= 2;

// This block gets broken into three bits. The rightmost

// bit is a block of length 1. The left hand part is split into

// two, a block of length LP[pshift+1] and one of LP[pshift].

// Both these two are appropriately heapified, but the root

// nodes are not necessarily in order. We therefore semitrinkle

// both of them

trinkle(m, p >>> 1, pshift + 1, head - LP[pshift] - 1, true);

trinkle(m, p, pshift, head - 1, true);

}

head--;

}

}

private static <C extends Comparable<? super C>> void sift(C[] m, int pshift,

int head) {

// we do not use Floyd's improvements to the heapsort sift, because we

// are not doing what heapsort does - always moving nodes from near

// the bottom of the tree to the root.

C val = m[head];

while (pshift > 1) {

int rt = head - 1;

int lf = head - 1 - LP[pshift - 2];

if (val.compareTo(m[lf]) >= 0 && val.compareTo(m[rt]) >= 0)

break;

if (m[lf].compareTo(m[rt]) >= 0) {

m[head] = m[lf];

head = lf;

pshift -= 1;

} else {

m[head] = m[rt];

head = rt;

pshift -= 2;

}

}

m[head] = val;

}

private static <C extends Comparable<? super C>> void trinkle(C[] m, int p,

int pshift, int head, boolean isTrusty) {

C val = m[head];

while (p != 1) {

int stepson = head - LP[pshift];

if (m[stepson].compareTo(val) <= 0)

break; // current node is greater than head. Sift.

// no need to check this if we know the current node is trusty,

// because we just checked the head (which is val, in the first

// iteration)

if (!isTrusty && pshift > 1) {

int rt = head - 1;

int lf = head - 1 - LP[pshift - 2];

if (m[rt].compareTo(m[stepson]) >= 0

|| m[lf].compareTo(m[stepson]) >= 0)

break;

}

m[head] = m[stepson];

head = stepson;

int trail = Integer.numberOfTrailingZeros(p & ~1);

p >>>= trail;

pshift += trail;

isTrusty = false;

}

if (!isTrusty) {

m[head] = val;

sift(m, pshift, head);

}

}

# 18. Stooge sort

|  |  |
| --- | --- |
| Stooge sort | |
| Visualization of Stooge sort. | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n*log 3 /log 1.5) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n*) |

**Stooge sort** is a [recursive](http://en.wikipedia.org/wiki/Recursion) [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) with a time complexity of O(*n*log 3 / log 1.5 ) = O(*n*2.7095...). The running time of the algorithm is thus extremely slow compared to efficient sorting algorithms, such as [Merge sort](http://en.wikipedia.org/wiki/Merge_sort), and is even slower than [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), a canonical example of a fairly inefficient and simple sort.

The algorithm is defined as follows:

* If the value at the end is smaller than the value at the start, swap them.
* If there are three or more elements in the current list subset, then:
  + Stooge sort the initial 2/3 of the list
  + Stooge sort the final 2/3 of the list
  + Stooge sort the initial 2/3 of the list again
* else: exit the procedure

The algorithm gets its name from [slapstick](http://en.wikipedia.org/wiki/Slapstick) routines of the [Three Stooges](http://en.wikipedia.org/wiki/Three_Stooges), in which each stooge hits the other two.[*[citation needed](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)*]

## Implementation

algorithm stoogesort(array L, i = 0, j = length(L)-1)

if L[j] < L[i] then

L[i] ↔ L[j]

if (j - i + 1) >= 3 then

t = (j - i + 1) / 3

stoogesort(L, i , j-t)

stoogesort(L, i+t, j )

stoogesort(L, i , j-t)

return L

# 19. Tim sort

|  |  |
| --- | --- |
| Timsort | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n)[[1]](http://en.wikipedia.org/wiki/Timsort#cite_note-1) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n\log n) |
| [**Worst case space complexity**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(n) |

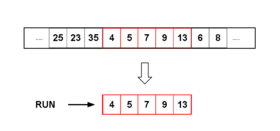
**Timsort** is a hybrid [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm), derived from [merge sort](http://en.wikipedia.org/wiki/Merge_sort) and [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), designed to perform well on many kinds of real-world data. It was invented by Tim Peters in 2002 for use in the [Python programming language](http://en.wikipedia.org/wiki/Python_%28programming_language%29). The [algorithm](http://en.wikipedia.org/wiki/Algorithm) finds subsets of the data that are already ordered, and uses the subsets to sort the data more efficiently. This is done by merging an identified subset, called a run, with existing runs until certain criteria are fulfilled. Timsort has been Python's standard sorting algorithm since version 2.3. It is now also used to sort arrays in [Java SE 7](http://en.wikipedia.org/wiki/Java_7),[[2]](http://en.wikipedia.org/wiki/Timsort" \l "cite_note-2) and on the [Android platform](http://en.wikipedia.org/wiki/Android_%28operating_system%29).[[3]](http://en.wikipedia.org/wiki/Timsort#cite_note-3)

|  |
| --- |
|  |

## Operation

Timsort was designed to take advantage of partial orderings that already exist in most real-world data. Timsort operates by finding runs, subsets of at least two elements, in the data. Runs can be either non-descending (each element is equal to or greater than its predecessor) or strictly descending (each element is lower than its predecessor). If it is descending, it must be strictly descending, since descending runs are later reversed by a simple swap of elements from both ends converging in the middle. This method is [stable](http://en.wikipedia.org/wiki/Sorting_algorithm#Stability) if the elements are present in a strictly descending order. After obtaining such a run in the given array, timsort processes it, and then continues its search for the next run.

### Minrun

[](http://en.wikipedia.org/wiki/File:Selection_of_minrun_by_timsort.png)

[http://bits.wikimedia.org/static-1.21wmf3/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Selection_of_minrun_by_timsort.png)

Timsort algorithm searches for such ordered sequences, minruns, to perform its sort

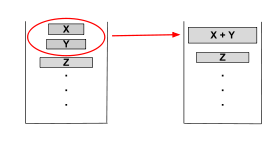
A natural run is a sub-array that is already ordered. Natural runs in real-world data may be of varied lengths. Timsort chooses a sorting technique depending on the length of the run. For example, if the run length is smaller than a certain value, insertion sort is used. Thus timsort is classified as an adaptive sort.[[4]](http://en.wikipedia.org/wiki/Timsort#cite_note-python_timsort-4)

The size of the run is checked against the minimum run size. The minimum run size (minrun) depends on the size of the [array](http://en.wikipedia.org/wiki/Array_data_type). For an array of fewer than 64 elements, the minrun is the size of the array, making Timsort essentially an insertion sort. For larger arrays, a number, referred to as minrun, is chosen from the range 32 to 65, such that the size of the array, divided by the minimum run size, is equal to, or slightly smaller than, a power of two. The final algorithm for this simply takes the six most significant bits of the size of the array, adds one if any of the remaining bits are set, and uses that result as the minrun. This algorithm works for all cases, including the one in which the size of the array is smaller than 64.[[4]](http://en.wikipedia.org/wiki/Timsort#cite_note-python_timsort-4)

### Insertion Sort

When an array is random, a natural run will most likely contain less than minrun elements. In this case, an appropriate number of succeeding elements are selected, and insertion sort is used to increase the size of the run to minrun size. Thus, most runs in a random array are, or become, minrun in length. This results in balanced merges, which provides an efficient way to proceed. It also results in a reasonable number of function calls in the implementation of the sort.[[5]](http://en.wikipedia.org/wiki/Timsort#cite_note-drmaciver-5)

### Merge Memory

[](http://en.wikipedia.org/wiki/File:Representation_of_stack_for_merge_memory_in_Timsort.svg)

[http://bits.wikimedia.org/static-1.21wmf3/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Representation_of_stack_for_merge_memory_in_Timsort.svg)

The minruns are inserted in a [stack](http://en.wikipedia.org/wiki/Stack_%28data_structure%29). If X < Y + Z then X and Y are merged and then inserted into a stack. In this way, merging is continued until all arrays satisfy a) X > Y + Z and b) Y > Z

Once run lengths are optimized, the runs are merged. The principle of Timsort implies that it will be merged by a specific technique that will ensure the highest efficiency. When a run is found, the algorithm pushes its base address and length on a stack. A function is then called which determines whether the run should be merged with previous runs. Timsort does not merge non-consecutive runs because doing this would cause the element common to all three runs to become out of order with respect to the middle run.

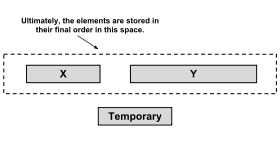
Thus, merging is always done on two consecutive runs. For this, the three top-most runs in the stack which are unsorted are considered. If, say, X, Y, Z represent the lengths of the three uppermost runs in the stack, the algorithm merges the runs so that ultimately the following two rules are satisfied:

1. X > Y + Z
2. Y > Z[[4]](http://en.wikipedia.org/wiki/Timsort#cite_note-python_timsort-4)

For example, if the first of the two rules is not satisfied by the current run status, that is, if X < Y + Z, then, Y is merged with the smaller of X and Z. The merging continues until both the rules are satisfied. Then the algorithm goes on to determine the next run.[[5]](http://en.wikipedia.org/wiki/Timsort#cite_note-drmaciver-5)

The rules above aim at maintaining run lengths as close to each other as possible to ensure balanced merges, which are more efficient. At the same time only a small number of runs may be remembered, as the stack is of a specific size. The algorithm also tries to exploit the fresh occurrence of the runs to be merged, in [cache memory](http://en.wikipedia.org/wiki/CPU_cache). Thus a compromise is attained between delaying merging, and exploiting fresh occurrence in cache memory.

### Merging Procedure

[](http://en.wikipedia.org/wiki/File:Merging_procedure_for_timsort.svg)

[http://bits.wikimedia.org/static-1.21wmf3/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Merging_procedure_for_timsort.svg)

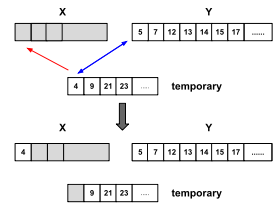
Algorithm creates a temporary memory equal to size of smaller array. Then, it shifts elements in (say if X is smaller) X to the temporary memory and then sorts and fills elements in final order into combined space of X and Y

Merging two adjacent runs is done with the help of temporary memory. The temporary memory is of the size of the minimum of the two runs. The algorithm copies the smaller of the two runs into this temporary memory and then uses the original memory (of the smaller run) and the memory of the other run to store the final run after sorting.

A simple merge algorithm is then run left to right or right to left depending on which run is smaller, on the temporary memory and original memory of the larger run, the final sorted run being stored in the original memory of the two initial runs. In order to make this more efficient, Timsort searches for appropriate positions for the starting element of one array in the other using an adaptation of [binary search](http://en.wikipedia.org/wiki/Binary_search).

Say, for example, two runs A and B are to be merged, with A as the smaller run. In this case a binary search is conducted in order to find the position in A where the first element of B will fit. Note that A and B are already sorted individually. Therefore, when such an appropriate position is found, the algorithm can ignore elements before that position in A while inserting (after comparing) elements of B. Similarly, the algorithm also looks for the position which the last element of A needs to take in B. The elements in B after this position can also be ignored for the merging. This preliminary searching may not prove efficient in the case of random data, however it is found to be highly efficient in other situations and is hence included.

### Galloping Mode

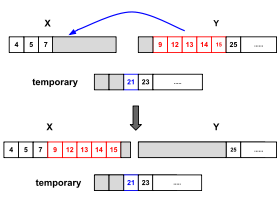
[](http://en.wikipedia.org/wiki/File:One-one_merging_timsort.svg)

[http://bits.wikimedia.org/static-1.21wmf3/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:One-one_merging_timsort.svg)

Elements (pointed to by blue arrow) are compared and the smaller element is moved to its final position (pointed to by red arrow).

Generally the merge occurs in what is called the ‘one pair at a time’ mode, where respective elements of both runs are compared. In the case where function merge\_lo is invoked, that is, when the algorithm merges left-to-right, the smaller of the two is brought to a merge area. A count of the number of times the final element appears in a given run is recorded. When this value reaches a certain threshold, MIN\_GALLOP, the merge switches to what is called the ‘galloping mode’. In this mode we use the previously mentioned adaptation of binary search to identify where the first element of the smaller array must be placed in the larger array and vice-versa. Thus the entire set of elements, in one array, occurring before this location can be moved to the merge area as a group and the other way round. This is possible as we have both the runs to be merged, ordered individually. The galloping mode is entered only when it is the most optimum method for the merge; this is decided by the value of min-gallop. Min-gallop is a variable initialized to MIN\_GALLOP. However the functions merge-lo and merge-hi increment the value of the variable, if galloping is not efficient, and decrement it if it is. If too many consecutive elements come from different runs, galloping mode is exited.[[4]](http://en.wikipedia.org/wiki/Timsort#cite_note-python_timsort-4)

When in galloping mode, the algorithm searches for the first element of one array in the other. This is done by comparing that first element (initial element) with the zeroth element of the other array, then the first, the third and so on, that is (2k - 1)th element, so as to get a range of elements between which the initial element will lie. This provides a shorter range to conduct binary search on, thus increasing efficiency. Galloping proves to be more efficient except in cases with especially long runs, but random data usually has shorter runs. Also, in cases where galloping is found to be less efficient as compared to [binary search](http://en.wikipedia.org/wiki/Binary_search_algorithm), galloping mode is exited from.

[](http://en.wikipedia.org/wiki/File:Copy_galloping_mode_timsort%282%29.svg)

[http://bits.wikimedia.org/static-1.21wmf3/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Copy_galloping_mode_timsort%282%29.svg)

All red elements are smaller than blue (here, 21). Thus they can be moved in a chunk to the final array.

However it is found that galloping is not always efficient. One reason is due to excessive function calls. Function calls are expensive and thus when they are large in number, they hamper program efficiency. Further there are cases where galloping mode requires a larger number of comparisons than a simple [linear search](http://en.wikipedia.org/wiki/Linear_search) (one at a time search). While for the first few cases both modes may require the same number of comparisons, over time galloping mode requires 33% more comparisons than linear search to arrive at the same results. Moreover all comparisons in galloping mode are done by [function calls](http://en.wikipedia.org/wiki/Function_call).

Also, it is seen that galloping is beneficial only when the initial element is not one of the first seven elements of the other run. This also results in MIN\_GALLOP being set to 7. To avoid the drawbacks of galloping mode, the merging functions adjust the value of min-gallop. If the element is from the array currently under consideration (that is, the array which has been returning the elements consecutively for a while), the value of min-gallop is reduced by one. Otherwise, the value is incremented by one, thus discouraging entry back to galloping mode. When this is done, in the case of random data, the value of min-gallop becomes so large, that the entry back to galloping mode never takes place.

In the case where merge-hi is used (that is, merging is done right-to-left), galloping needs to start from the right end of the data, that is the last element. Galloping from the beginning also gives the required results, but makes more comparisons than required. Thus, the algorithm for galloping includes the use of a variable which gives the index at which galloping should begin. Thus the algorithm can enter galloping mode at any index and continue thereon as mentioned above, as in, it will check at the next index which is offset by 1, 3, 7,...., (2k - 1).. and so on from the current index. In the case of merge-hi, the offsets to the index will be -1, -3, -7,....[[4]](http://en.wikipedia.org/wiki/Timsort" \l "cite_note-python_timsort-4)

## Performance

According to [information theory](http://en.wikipedia.org/wiki/Information_theory), no [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) can perform better than \Theta(n \log n)comparisons in the average case. On real-world data, Timsort often requires far fewer than \Theta(n \log n)comparisons, because it takes advantage of the fact that sublists of the data may already be in order.[[6]](http://en.wikipedia.org/wiki/Timsort#cite_note-6) In case of random data, there are no partially ordered subarrays to take advantage of. In this case, timsort approaches the theoretical limit of \log(n!), which is in \Theta(n \log n).[[4]](http://en.wikipedia.org/wiki/Timsort#cite_note-python_timsort-4)

The following table compares the time complexity of timsort with other comparison sorts.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Timsort** | [**Merge sort**](http://en.wikipedia.org/wiki/Merge_sort) | [**Quicksort**](http://en.wikipedia.org/wiki/Quicksort) | [**Insertion sort**](http://en.wikipedia.org/wiki/Insertion_sort) | [**Selection sort**](http://en.wikipedia.org/wiki/Selection_sort) | [**Smoothsort**](http://en.wikipedia.org/wiki/Smoothsort) |
| **Best Case** | \Theta(n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n) | \Theta(n^2) | \Theta(n) |
| **Average Case** | \Theta(n \log n) | \Theta(n \log n) | \Theta(n \log n) | \Theta(n^2) | \Theta(n^2) | \Theta(n \log n) |
| **Worst Case** | \Theta(n \log n) | \Theta(n \log n) | \Theta(n^2) | \Theta(n^2) | \Theta(n^2) | \Theta(n \log n) |

The following table provides a comparison of the space complexities of the various sorting techniques. Note that for merge sort, the *worst case* space complexity is usually O(n).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Timsort** | [**Merge sort**](http://en.wikipedia.org/wiki/Merge_sort) | [**Quicksort**](http://en.wikipedia.org/wiki/Quicksort) | [**Insertion sort**](http://en.wikipedia.org/wiki/Insertion_sort) | [**Selection sort**](http://en.wikipedia.org/wiki/Selection_sort) | [**Smoothsort**](http://en.wikipedia.org/wiki/Smoothsort) |
| **Space Complexity** | O(n) | O(n) | O(\log n) | O(1) | O(1) | O(1) |

Note, however, that the space complexity of both Timsort and merge sort can be reduced to \log nat the cost of speed (see in-place merge sort).

# 20. Tournament sort

|  |  |
| --- | --- |
| Tournament sort | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |

|  |  |
| --- | --- |
|  |  |

**Tournament sort** is a [sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm). It improves upon the naive [selection sort](http://en.wikipedia.org/wiki/Selection_sort) by using a [priority queue](http://en.wikipedia.org/wiki/Priority_queue) to find the next element in the sort. In the naive selection sort, it takes O(*n*) operations to select the next element of *n* elements; in a tournament sort, it takes O(log *n*) operations (after building the initial tournament in O(*n*)). Tournament sort is a variation of [heapsort](http://en.wikipedia.org/wiki/Heapsort).

## Common application

Tournament replacement selection sorts are used to gather the initial runs for external sorting algorithms. Conceptually, an external file is read and its elements are pushed into the priority queue until the queue is full. Then the minimum element is pulled from the queue and written as part of the first run. The next input element is read and pushed into the queue, and the min is selected again and added to the run. There's a small trick that if the new element being pushed into the queue is less than the last element added to the run, then the element's sort value is increased so it will be part of the next run. On average, a run will be 100% longer than the capacity of the priority queue[[1]](http://en.wikipedia.org/wiki/Tournament_sort" \l "cite_note-0).

Tournament sorts may also be used in N-way merges.

## The tournament

The name comes from its similarity to a [single-elimination tournament](http://en.wikipedia.org/wiki/Single-elimination_tournament) where there are many players (or teams) that play in two-sided matches. Each match compares the players, and the winning player is promoted to play at match at the next level up. The hierarchy continues until the final match determines the ultimate winner. The tournament determines the best player, but the player who was beaten in the final match may not be the second best—he may be inferior to other players the winner bested.

# 21. Tree sort

|  |  |
| --- | --- |
| Tree sort | |
| **Class** | [Sorting algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm) |
| **Data structure** | [Array](http://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n*2) (unbalanced)  O(*n* log *n*) (balanced) |
| [**Best case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n*) |
| [**Average case performance**](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) | O(*n* log *n*) |

A **tree sort** is a [sort algorithm](http://en.wikipedia.org/wiki/Sort_algorithm) that builds a [binary search tree](http://en.wikipedia.org/wiki/Binary_search_tree) from the keys to be sorted, and then traverses the tree ([in-order](http://en.wikipedia.org/wiki/Tree_traversal)) so that the keys come out in sorted order. Its typical use is when sorting the elements of a stream from a file. Several other sorts would have to load the elements to a temporary data structure, whereas in a tree sort the act of loading the input into a data structure is sorting it.

## Efficiency

Adding one item to a binary search tree is on average an O(log(n)) process, so adding n items is an O(n log(n)) process, making tree sort a so-called 'fast sort'. But adding an item to an unbalanced binary tree needs [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n*) time in the worst-case, when the tree resembles a [linked list](http://en.wikipedia.org/wiki/Linked_list) ([degenerate tree](http://en.wikipedia.org/wiki/Binary_Tree#Types_of_binary_trees)), causing a worst case of O(n2) for this sorting algorithm. The worst case scenario then is triggered by handing a Tree Sort algorithm an already sorted set. This would make the time needed to insert all elements into the binary tree O(n2). The dominant process in the Tree Sort algorithm is the "insertion" into the binary tree, assuming that the time needed for retrieval is O(n).

The worst-case behaviour can be improved upon by using a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree). Using such a tree, the algorithm has an O(n log(n)) worst-case performance, thus being degree-optimal.

## Example

In a simple [functional programming](http://en.wikipedia.org/wiki/Functional_programming) form, the algorithm (in [Haskell](http://en.wikipedia.org/wiki/Haskell_%28programming_language%29)) would look something like this:

data Tree a = Leaf | Node (Tree a) a (Tree a)

insert :: Ord a => a -> Tree a -> Tree a

insert x Leaf = Node Leaf x Leaf

insert x (Node t y t') | x <= y = Node (insert x t) y t'

insert x (Node t y t') | x > y = Node t y (insert x t')

flatten :: Tree a -> [a]

flatten Leaf = []

flatten (Node t x t') = flatten t ++ [x] ++ flatten t'

treesort :: Ord a => [a] -> [a]

treesort = flatten . foldr insert Leaf

Mind that in the above example, both the insertion algorithm and the retrieval algorithm have O(n2) worst case scenarios.