WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
فارسی
Српски / srpski
Edit links

Flashsort

From Wikipedia, the free encyclopedia

Flashsort is a distribution sorting algorithm showing linear computational complexity $O(n)$ for uniformly distributed data sets and relatively little additional memory requirement. The original work was published in 1998 by Karl-Dietrich Neubert.[1]

Contents

Concept [edit]

The basic idea behind flashsort is that in a data set with a known 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_i is the quantile, computed as:

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

where A is 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]

Memory efficient implementation [edit]

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 A in 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 A and its class is calculated. The pointers in vector L are used to insert the cycle-leader into the correct class, and the class's pointer in L is 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 A the algorithm is processing. Let this index be i. Elements A_0 through A_{i-1} have already been classified and inserted into the correct class. Suppose that i is greater than the current pointer to A_i's class. Now suppose that the A_i is 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_i itself, 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][2]

Performance [edit]

The only extra memory requirements are the auxiliary vector L for 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 m of 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][3]

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 $m = 0.42n$ to be optimal.

Memory-wise, flashsort avoids the overhead needed to store classes in the very similar bucketsort. For $m = 0.1n$ with uniform random data, flashsort is faster than heapsort for all n and faster than quicksort for $n > 80$. It becomes about as twice as fast as quicksort at $n = 10000$.[1]

Due to the *in situ* permutation that flashsort performs in its classification process, flashsort is not stable. If stability is required, it is possible to use a second, temporary, array so elements can be classified sequentially. However, in this case, the algorithm will require $O(n)$ space.

References [edit]

1. ^ a b c d Neubert, Karl-Dietrich (February 1998). "The Flashsort Algorithm". *Dr. Dobb's Journal*: 123. Retrieved 2007-11-06.
2. ^ a b Karl-Dietrich Neubert (1998). "The FlashSort Algorithm". Retrieved 2007-11-06.
3. ^ Li Xiao, Xiaodong Zhang, Stefan A. Kubricht. "Cache-Effective Quicksort". *Improving Memory Performance of Sorting Algorithms*. Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795. Archived from the original on 2007-11-02. Retrieved 2007-11-06.

External links [edit]

- Sorting in Linear time
- Implementations of Randomized Sorting on Large Parallel Machines (1992)
- Implementation of Parallel Algorithms (1992)
- Visualization of Flashsort

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

Categories: Sorting algorithms

This page was last modified on 25 October 2014, at 23:59.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

Privacy policy About Wikipedia Disclaimers Contact Wikipedia Developers Mobile view