WIKIPEDIA

Library sort

Library sort, or **gapped insertion sort** is a sorting algorithm that uses an insertion sort, but with gaps in the array to accelerate subsequent insertions. The name comes from an analogy:

> Suppose a librarian were to store his books alphabetically on a long shelf, starting with the As at the left end, and continuing to the right along the shelf with no spaces between the books until the end of the Zs. 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 Bs all the way down to the Zs 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.

Library sort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$
Best-case performance	$O(n)$
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$

The algorithm was proposed by Michael A. Bender, Martín Farach-Colton, and Miguel Mosteiro in 2004[1] and was published in 2006.[2]

Like the insertion sort it is based on, library sort is a stable comparison sort and can be run as an online algorithm; however, it was shown to have a high probability of running in O(n log n) time (comparable to quicksort), rather than an insertion sort's O(n^2). The mechanism used for this improvement is very similar to that of a skip list. There is no full implementation given in the paper, nor the exact algorithms of important parts, such as insertion and rebalancing. Further information would be needed to discuss how the efficiency of library sort compares to that of other sorting methods in reality.

Compared to basic insertion sort, the drawback of library sort is that it requires extra space for the gaps. The amount and distribution of that space would be implementation dependent. In the paper the size of the needed array is *(1 + ε)n*,[2] but with no further recommendations on how to choose ε.

One weakness of insertion sort is that it may require a high number of swap operations and be costly if memory write is expensive. Library sort may improve that somewhat in the insertion step, as fewer elements need to move to make room, but is also adding an extra cost in the rebalancing step. In addition, locality of reference will be poor compared to mergesort as each insertion from a random data set may access memory that is no longer in cache, especially with large data sets.

Contents

Implementation

Algorithm

Let us say we have an array of n elements. We choose the gap we intend to give. Then we would have a final array of size (1 + ε)n. The algorithm works in log n rounds. In each round we insert as many elements as there are in the final array already, before re-balancing the array. For finding the position of inserting, we apply Binary Search in the final array and then swap the following elements till we hit an empty space. Once the round is over, we re-balance the final array by inserting spaces between each element.

Following are three important steps of the algorithm:

1. **Binary Search**: Finding the position of insertion by applying binary search within the already inserted elements. This can be done by linearly moving towards left or right side of the array if you hit an empty space in the middle element.
2. **Insertion**: Inserting the element in the position found and swapping the following elements by 1 position till an empty space is hit.
3. **Re-Balancing**: Inserting spaces between each pair of elements in the array. This takes linear time, and because there are log n rounds in the algorithm, total re-balancing takes O(n log n) time only.

Pseudocode

```
procedure rebalance(A, begin, end) is
    r ← end
    w ← end * 2
    while r ≥ begin do
        A[w+1] ← gap
        A[w] ← A[r]
        r ← r - 1
        w ← w - 2
```

```
procedure sort(A) is
    n ← length(A)
    S ← new array of n gaps
    for i ← 1 to floor(log2(n) + 1) do
        for j ← 2^i to 2^(i+1) do
            ins ← binarysearch(A[j], S, 2^(i-1))
            insert A[j] at S[ins]
```

Here, `binarysearch(el, A, k)` performs binary search in the first k elements of A, skipping over gaps, to find a place where to locate element el. Insertion should favor gaps over filled-in elements.

References

1. Bender, Michael A.; Farach-Colton, Martín; Mosteiro, Miguel A. (1 July 2004). "Insertion Sort is O(n log n)". arXiv:cs/0407003 (https://arxiv.org/abs/cs/0407003).
2. Bender, Michael A.; Farach-Colton, Martín; Mosteiro, Miguel A. (June 2006). "Insertion Sort is O(n log n)" (http://csis.pace.edu/~mmosteiro/pub/paperToCS06.pdf) (PDF). *Theory of Computing Systems*. **39** (3): 391–397. arXiv:cs/0407003 (https://arxiv.org/abs/cs/0407003). doi:10.1007/s00224-005-1237-z (https://doi.org/10.1007%2Fs00224-005-1237-z).

External links

- Gapped Insertion Sort (http://www.cs.sunysb.edu/~bender/newpub/BenderFaMo06-librarysort.pdf)