Comb sort

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

Comb sort is a relatively simplistic sorting algorithm originally designed by Wlodzimierz Dobosiewicz in 1980. Later it was rediscovered and popularized by Stephen Lacey and Richard Box with a Byte Magazine article published in April 1991. Comb sort improves on bubble sort, and rivals algorithms like Quicksort. 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 one. (Shell sort is also based on this idea, but it is a modification of insertion sort rather than bubble sort.)

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) for 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 authors suggested 1.3 after trying some random lists and finding it to be generally the most effective. A value too small slows the algorithm down because more comparisons must be made, whereas a value too large may not kill enough turtles to be practical.

Text describes an improvement to comb sort using the base value $1/\left(1-\frac{1}{e^{\varphi}}\right) \approx 1.247330950103979$as the shrink factor. 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 or merge sort), combsort is more effective in its earlier passes than it is during the final passes, when it resembles a 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 or cocktail sort, or, even better, an insertion sort, will increase the sort's overall efficiency.

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.

Implementations

Pseudocode

```
function combsort(array input)
    gap := input.size //initialize gap size

    loop until gap = 1 and swaps = 0
        //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
        swaps := 0 //see bubblesort for an explanation


        //a single "comb" over the input list
        loop until i + gap >= input.size //see shellsort for similar idea
            if input[i] > input[i+gap]
                swap(input[i], input[i+gap])
                swaps := 1 // Flag a swap has occurred, so the
                           // list is not guaranteed sorted
            end if
            i := i + 1
        end loop


    end loop
end function
```

C++

This is an STL-style implementation. It will sort any range between first and last. This works with any forward iterators, but is more effective with random access iterators or pointers.

```
template<class ForwardIterator>
void combsort ( ForwardIterator first, ForwardIterator last )
{
    static const double shrink_factor = 1.247330950103979;
    typedef typename std::iterator_traits<ForwardIterator>::difference_type
difference_type;
    difference_type gap = std::distance(first, last);
    bool swaps = true;

    while ( (gap > 1) || (swaps == true) ){
        if (gap > 1)
            gap = static_cast<difference_type>(gap/shrink_factor);

        swaps = false;
        ForwardIterator itLeft(first);
        ForwardIterator itRight(first); std::advance(itRight, gap);

        for ( ; itRight!=last; ++itLeft, ++itRight ){
            if ( (*itRight) < (*itLeft) ){
                std::iter_swap(itLeft, itRight);
                swaps = true;
            }
        }
    }
}
```

Java

```
public static <E extends Comparable<? super E>> void sort(E[] input) {
    int gap = input.length;
    boolean swapped = true;
    while (gap > 1 || swapped) {
        if (gap > 1)
            gap = (int) (gap / 1.247330950103979);

        int i = 0;
        swapped = false;
        while (i + gap < input.length) {
            if (input[i].compareTo(input[i + gap]) > 0) {
                E t = input[i];
                input[i] = input[i + gap];
                input[i + gap] = t;
                swapped = true;
            }
```

```
            i++;
        }
    }
}
```

C

```c
void combsort(int *arr, int size) {
    float shrink_factor = 1.247330950103979;
    int gap = size, swapped = 1, swap, i;

    while ((gap > 1) || swapped) {
        if (gap > 1)
           gap = gap / shrink_factor;

        swapped = 0;
        i = 0;

        while ((gap + i) < size) {
            if (arr[i] - arr[i + gap] > 0) {
                swap = arr[i];
                arr[i] = arr[i + gap];
                arr[i + gap] = swap;
                swapped = 1;
            }
            ++i;
        }
    }
}
```

See also

- Bubble sort, a generally slower algorithm, is the basis of comb sort.
- Cocktail sort, or bidirectional bubble sort, is a variation of bubble sort that also addresses the problem of turtles, albeit less effectively.

External links

- .NET Implementation of comb sort and several other algorithms [1]

References

[1] http://www.sharpdeveloper.net/content/archive/2007/08/14/dot-net-data-structures-and-algorithms.aspx

Article Sources and Contributors

Comb sort *Source*: http://en.wikipedia.org/w/index.php?oldid=398479313 *Contributors*: Adriantam, Bazzargh, Bgoldenberg, Booyabazooka, CJLL Wright, Cbmeeks, Cgibbard, Chriswbell, Comocomocomocomo, Davidkazuhiro, Dcoetzee, Decrypt3, Deewiant, Destynova, DevastatorIIC, Doradus, Emurphy42, Fesharakif, Foobar, Fresheneesz, Gamma, Ggurbet, Glrx, Graue, Happypal, INS Pirat, IanOsgood, Jafet, Jogers, Josh Kehn, Kehrbykid, Kolobok, LittleOldMe, Mav, Nlfiedler, Oberono, Ohanian, Oskar Sigvardsson, Oğuz Ergin, Paddy3118, PerVognsen, Pulveriser, Quuxplusone, Rainer3000, Rhanekom, Rursus, Silly rabbit, Sligocki, Smjg, TakuyaMurata, Timwi, VivekKhanna CSE, Yugsdrawkcabeht, ZeroOne, Олександр Кравчук, 71 anonymous edits

License