Bubble Sort

```
bubbleSort( int a[], int n)
Begin
    for i = 1 to n-1  // n-1 repeats
        sorted := true
        for j = 0 to n-1-i
            if a[j] > a[j+1]
                temp := a[j]
                a[j] := a[j+1]
                a[j+1] := temp
                sorted := false
        end for
        if sorted
            break from i loop
    end for
End
```

Bubble sort uses a loop (inside j loop) to travel thru' an array comparing adjacent values as it moves along. If an array element a[j] is greater than the element immediately to its right a[j+1], it swaps them. The first time around, this process will move or bubble the largest value to the end of the array. So for instance

5	3	1	9	8	2	4	7

will end up as

3	1	5	8	2	4	7	**9**

This process is repeated, on the second iteration, the second largest value will be moved to the second last array position and so on.

In all, the bubble process (inside j loop) is repeated n-1 times for an array of size n.

Bubble Sort Example

		0	1	2	3	4	5	6	7
i = 1	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 2	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 3	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 4	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 5	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 6	0	1	2	3	**4**				
	1	1	2	3					
i = 7	0	1	2	**3**					
		1	**2**						

Note for array of size 8, outside i loop repeats 7 times.

Complexity

Clearly for an array of size n, the outside loop repeats n-1 times.

To begin with the inside loop does n-1 comparisons, next time n-2 and so on. Finally on the last iteration of the outside loop, the inside loop does 1 comparison. So on average the inside loop does $((n-1) + 1) / 2 \approx n/2$ comparisons.

Therefore, the overall number of computation steps is $n * n/2 = n^2/2$

Complexity of bubble sort $= O(n^2)$

Tortoises and Hares

Another example

		0	1	2	3	4	5	6	7	8
i = 1	j	9	8	5	2	7	3	4	1	0
	0	8	9	5	2	7	3	4	1	0
	1	8	5	9	2	7	3	4	1	0
	2	8	5	2	9	7	3	4	1	0
	3	8	5	2	7	9	3	4	1	0
	4	8	5	2	7	3	9	4	1	0
	5	8	5	2	7	3	4	9	1	0
	6	8	5	2	7	3	4	1	9	0
	7	8	5	2	7	3	4	1	**0**	**9**
i = 2										
	0	5	8	2	7	3	4	1	**0**	**9**
	1	5	2	8	7	3	4	1	**0**	**9**
	2	5	2	7	8	3	4	1	**0**	**9**
	3	5	2	7	3	8	4	1	**0**	**9**
	4	5	2	7	3	4	8	1	**0**	**9**
	5	5	2	7	3	4	1	8	**0**	**9**
	6	5	2	7	3	4	1	**0**	**8**	**9**

A tortoise is a large value on the lhs of the array, like 9 above. In one bubble pass it is bubbled to the rhs of the array. In contract, a small value referred to as a hare like 0 above on the rhs, moves only 1 step to the left on a full bubble pass. So this asymmetry is what is referred to as *Tortoises and Hares*.

Comb Sort

The basic idea is to eliminate tortoises, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. *Hares*, 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. The inner loop of 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 a "shrink factor" k: [n/k, n/k^2, n/k^3, ..., 1].

The gap starts out as the length of the list n being sorted divided by the shrink factor k (generally 1.3; see below) and one pass of the aforementioned modified bubble sort is applied with that 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.

The shrink factor has a great effect on the efficiency of comb sort. k = 1.3 has been suggested as an ideal shrink factor by the authors of the original article after empirical testing on over 200,000

random lists. A value too small slows the algorithm down by making unnecessarily many comparisons, whereas a value too large fails to effectively deal with tortoises.

combSort(input[])

```
    gap := input.size // Initialize gap size
    shrink := 1.3 // Set the gap shrink factor
    sorted := false

    loop while sorted = false
        // Update the gap value for a next comb
        gap := floor(gap / shrink)
        if gap > 1
            sorted := false // We are never sorted as long as gap > 1
        else
            gap := 1
            sorted := true // If there are no swaps this pass, we are done
        end if

        // A single "comb" over the input list
        i := 0
        loop while i + gap < input.size
            if input[i] > input[i+gap]
                swap(input[i], input[i+gap])
                sorted := false
                // If this assignment never happens within the loop,
                // then there have been no swaps and the list is sorted.
             end if

             i := i + 1
        end loop
    end loop
end
```

Exercise

Show the workings of comb sort on the second example above.