|  |
| --- |
| Cover Sheet |
| Huck Sort |
| A pre-sort sort |
|  |
| **Ross Studtman** |
| **7/7/2013** |

|  |
| --- |
|  |

Huck Sort Version 1.0

Dr. Ross Studtman, A.A.S. C.I.S., D.C.

…a work in progress…

Abstract

Huck sort is a pre-sort, comparison sort that seeks to move a collection to a more sorted state but without, by itself, completing the sort. The pre-sort collection can then be handed off to classic sorting algorithms such as Insertion sort which perform much faster on nearly-sorted collections.

**Introduction – The Problem With Turtles**

Huck sort was originally conceived as a solution to the problem of turtles in Bubble sort.

Bubble sort is a comparison sort that compares adjacent element values. When a larger value is compared against a smaller value the larger value moves forward one index, the smaller value moves backward by one index, i.e., they are swapped. Comparisons continue until the largest value in the list is sorted to the end of the list.

In bubble sort large values can move quickly toward the end of the list, only being halted if a larger value is discovered. In this manner, during a single iteration, a large value can move quickly toward its final sort position. This is why larger values are termed "rabbits." In contrast, during a single iteration a smaller value can only move backward toward its final sort position by one index. This is why smaller values are termed "turtles."

Rabbits reach their sorted positions quickly. Turtles do not. A natural question to ask, then, is “Can turtles be sped up?”

A popular solution to the problem of turtles has been addressed by Cocktail sort. Cocktail sort, which goes by many pseudonyms, basically performs a “reverse run” over the collection and in so doing turns turtles into rabbits and rabbits into turtles.

Huck sort takes a different approach and asks the question, “When a rabbit encounters a turtle why not ‘huck’ the turtle toward the end where it belongs rather than moving it one step backward?”

**Evolution of Huck Sort**

Initially the Huck algorithm was nested inside Bubble sort in an academic attempt to improve Bubble sort’s speed. This was a mild success, leading to 50% improvements in sort speed. Then the thinking evolved to “Why not move the Huck algorithm completely outside of the sort for which it was designed to improve?” And with that the Huck algorithm became the Huck pre-sort and when used as a pre-sort for insertion sort, which favors a nearly-sorted list, the improvement can climb as high as 1,000%.

**Huck Sort**

Huck sort pre-sorts a list, reorganizing the list from its original state into a more ordered state. Then this pre-sorted list is handed off to another sort, like insertion sort, to complete the task.

Huck sort’s general tasks are:

1. Determine what value is used to identify a turtle
2. Determine where a hucked turtle should land (which index)
3. Determine how many iterations to perform.
4. Traverse the list and huck turtles toward the front of the list.
5. If more iterations are to be performed:
   1. Determine new turtle value prior to next iteration.
   2. Begin iteration after previously hucked turtles, i.e., once a turtle has been hucked it should not be hucked again.
      1. Keep running tally of hucked turtles.

The best approach for determining the value of a turtle (with each iteration) and how many iterations to perform has not been well determined. Two approaches to these issues are presented.

**Scenario 1:** ***demonstrated by HuckSort\_V1***

* Number of trips is hard coded.
* Turtle value determined by:
  + Obtain statistical mean of collection (as integer)
  + Divide statistical mean of collection by number of trips
    - Number of trips is decremented each trip, hence turtle value increases with each trip

**Scenario 2:** ***demonstrated by HuckSort\_V2***

* Number of trips is hard coded. Determined by length of array in step four, below.
* Turtle value determined by:
  1. Obtain statistical mean of collection (as double)
  2. Obtain standard deviation of collection (as double)
  3. Divide mean by statistical deviation (coefficient of variation)
  4. Take the following array of double values: 0.25, 0.50, 2.00, 2.25, 2.50 and divide each by Coefficient of Variation.
  5. Turtle value calculated each trip by multiplying collection’s mean by the value found in the array of step 4.
     + mean \* denominators[loopCounter]

**Scenario Three:** **not yet coded**

Scenarios 1 & 2 both calculate array statistics once and then modify those values via some hard-coded mechanism.

Instead of modifying the statistical values with each iteration of Huck sort, consider recalculating statistics for the un-hucked portion of the collection. Hence, each iteration of the outer loop results in a new mean, a new standard deviation, a new coefficient of variance, etc. This, it is supposed, would allow for a more finely tuned turtle selection process; and a process that can dynamically adjust to whatever values are left (un-hucked values).

Additionally, a process like this might use a single hard-coded calculation to conjure the turtle value threshold. For instance: “turtleValue = mean (0.25 / CoV)”, with a calculated mean that adjusts to the remaining unsorted turtles this could be an option for creating different turtle values with each Huck iteration.

**Scenario Four:** **not yet coded**

Instead of using Huck as only a pre-sort, consider with each pass throwing hucked turtles to their own array, sort those mini-arrays with supposedly tight value ranges, then zip back together all the various sorted turtle arrays into a single larger array, a bit like the divide and conquer sorts do; or perhaps look for a Shell sort-like approach.

**Other Scenarios:**

It would be clever to come up with an algorithm that allowed the number of iterations of the outer loop to change based on some kind of statistical analysis of the array. Along this line of thinking, and along the lines presented in Scenario Three (above), the number of outer loop iterations might run until the number of hucked turtles eclipses a certain percentage of the array to sort’s size; the intrigue of this is further enhanced by thinking of recalculating the statistical figures for the un-hucked portion of the array, and keep doing that until (perhaps) the number of hucked turtles equals the length of the array.

**Known Issues with Huck Sort Program**

* (May not be an issue any longer after mild refactoring) Clicking “Perform Sorts” a second time does obtain an array stuffed with new random variables, as the array statistics indicate by slight fluctuations, but the JIT compiler optimizes the code on the second run through and this is why values improve even though array-generating metrics remain the same. See StackOverflow question regarding this issue:
  + <http://stackoverflow.com/questions/17774073/understanding-why-how-javas-native-sorting-of-an-int-array-is-optimized-on-succ>
* **Huck sort has a cool story, is easy to tell, and I suspect there are loads of clever optimizations that can be done.**

APPENDIX

**Bubble Sort**

Bubble sort is a comparison sort that compares adjacent element values. When a larger value is compared against a smaller value the larger value moves forward one index, the smaller value moves backward by one index, i.e., they are swapped. Comparisons continue until the largest value in the list is sorted to the end of the list.

Large values can move quickly toward the end of the list, only being halted if a larger value is discovered. In this manner, during a single iteration, a large value can move quickly toward its final sort position. This is why larger values are sometimes called "rabbits."

In contrast, during a single iteration a smaller value can only move backward toward its final sort position by one index. This is why smaller values are sometimes called "turtles."

Bubble sort is O(N^2) because it uses nested loops.

Each iteration of the outer loop results in the largest element value being sorted to the end of the list.

Each iteration of the inner loop starts at the beginning of the list and has to evaluate all element values except those that have been sorted via previous iterations of the outer loop.

**Insertion Sort**

Insertion Sort: (explanation assumes sorting small to large)

Insertion sort is a comparison sort that assumes the first element of a list is already sorted, which means that at the very beginning of this sort there exist two sub lists inside the list to be sorted:

1) an already-sorted portion

2) an un-sorted portion

The insertion sort begins each iteration by comparing the first element of the un-sorted portion to the last element of the already-sorted portion, i.e., compare the current element with the previous element.

If the un-sorted element is smaller than the previous element, move the previous element forward one index, move the list pointer back one element, and repeat the comparisons until the proper location of the un-sorted element is found.

References

Bubble sort. (n.d.). In *Wikipedia.* Retrieved July 6, 2013 from <http://en.wikipedia.org/wiki/Bubble_sort>

Liang, D.Y. (2013). *Introduction to Java Programming (9th ed.).* Boston: Pearson.