Shafro Batyrov

CMSC 451

10-17-2018

# Introduction

## Brief introduction of Heap Sort

* Heap sort is a comparison based sorting technique based on Binary Heap data structure. A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. It is similar to selection sort where we first find the maximum element and place it at the end. We repeat the same process for the remaining elements.
* Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.
* Heap Sort Algorithm for sorting in increasing order:  
  **1.** Build a max heap from the input data.  
  2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.  
  3. Repeat above steps while size of heap is greater than 1.

## High-level pseudocode for Heap Sort Implementations

IterativeSort (A)

Int N = list.length-1

Build Heap Iterative List

For i ←N, i>0, i-1

Exchange 0 to I in the list

Decrement N

Add N to max Heap Iterative List

End For

Return A

End IterativeSort)

Recursive Sort(A, N)

If n >0

Exchange 0 to N in the list

MaxHeapRecursive list swap 0 for decremented N

RecursiveSort list adds N

End If

End RecursiveSort()

buildHeapRecursiveSort(A)

For i←(list length-1)/2, i>=0, i-1

MaxHeapRecursive list swap I for list.length-1

End For

maxHeapRecursive(A,I,N)

Increment count

Left node = 2\*i

Right node = (2\*i)+1

Max=i

If Left <=N and left child array > i array

Max=Left

If Right <=N and right child array > max array

Max=Right

If Max does not equal i

Exchange I for max in the array

Exchange max for N in maxHeapRecursive array

End If

End maxHeapRecursive()

BuildHeapIterative(A)

For i= (list.length-1)/2; i >= 0; i-1

-1 from maxHeapIterative List

End buildHeapIterative()

maxHeapIterative(A,N)

For i=0, i<N, i+1

Increment count

Left=2\*i

Right=(2\*i)+1

Max=i

If Left <=N and left child array > i array

Max=Left

If Right <=N and right child array > max array

Max=Right

If Max does not equal i

Exchange I for max in the array

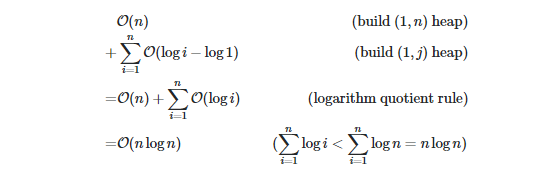
Exchange max for i in maxHeapIterative array

End For

End maxHeapIterative

## Big-Θ analysis of the two versions of Heap Sort

* The worst case and best case complexity for heap sort are both Θ(nlogn). Therefore, heap sort needs Θ(nlogn) comparisons for any input array. Complexity of heap sort:



* The space complexity of heap sort is O(1). Only O(1) additional space is required because the heap is built inside the array to be sorted.

## Explanation of Approach to avoiding the problems associated with JVM warm-up

Within the JVM, the compiler used is known as the Just In Time (JIT) compiler. The JIT compiles bytecode into native code during runtime after a program has started. Running a benchmark at the start of an application will cause poor results due to numerous JIT class optimizations and JVM garbage collection.

In my code, I followed a guide by baeldung.com to run a basic JVM warmup consisting of thousands of dummy class instantiation and iterations. To quote: “Once class-loading is complete, all important classes (used at the time of process start) are pushed into the JVM cache (native code) – which makes them accessible faster during runtime. Other classes are loaded on a per-request basis” (Baeldung, 2017).

## Discussion of the critical operation chosen and why selected

The critical operation recorded in the recursive and iterative implementations was the total number of swaps. I suppose the other operation that could be counted is the total number of swaps to insert each element in its correct position. However, I find the number of counts to be a clearer indication of performance between the two implementations since swapping elements will always be performed the same in both cases.

# Analysis

## Graph of critical operations for both algorithms and one for the execution times

## Comparison of the performance of the two versions of the algorithm

It is clear from these graphs that the recursive implementation of heap sort is more efficient, regardless of the data size. It requires a fraction of the amount of critical operations (comparisons) and performs them with an average execution tens of times faster than the iterative sort.

Below is the output from the algorithm:

Test Case 1:

Case Data Size: 50

Iterative Algorithm Results:

Average Critical Operation Count: 2401.00

Standard Deviation of Count: 2351.00

Coefficient of Variation of Count: 97.92

Average Execution Time: 0.480 ns

Standard Deviation of Time: 49.520 ns

Coefficient of Variation of Time: 10309.25

Recursive Algorithm Results:

Average Critical Operation Count: 314.76

Standard Deviation of Count: 264.76

Coefficient of Variation of Count: 84.11

Average Execution Time: 0.124 ns

Standard Deviation of Time: 49.876 ns

Coefficient of Variation of Time: 40160.44

Test Case 2:

Case Data Size: 100

Iterative Algorithm Results:

Average Critical Operation Count: 9801.00

Standard Deviation of Count: 9751.00

Coefficient of Variation of Count: 99.49

Average Execution Time: 1.528 ns

Standard Deviation of Time: 48.472 ns

Coefficient of Variation of Time: 3172.08

Recursive Algorithm Results:

Average Critical Operation Count: 729.14

Standard Deviation of Count: 679.14

Coefficient of Variation of Count: 93.14

Average Execution Time: 0.115 ns

Standard Deviation of Time: 49.885 ns

Coefficient of Variation of Time: 43430.88

Test Case 3:

Case Data Size: 250

Iterative Algorithm Results:

Average Critical Operation Count: 62001.00

Standard Deviation of Count: 61951.00

Coefficient of Variation of Count: 99.92

Average Execution Time: 2.070 ns

Standard Deviation of Time: 47.930 ns

Coefficient of Variation of Time: 2316.00

Recursive Algorithm Results:

Average Critical Operation Count: 2144.18

Standard Deviation of Count: 2094.18

Coefficient of Variation of Count: 97.67

Average Execution Time: 0.239 ns

Standard Deviation of Time: 49.761 ns

Coefficient of Variation of Time: 20815.95

Test Case 4:

Case Data Size: 500

Iterative Algorithm Results:

Average Critical Operation Count: 249001.00

Standard Deviation of Count: 248951.00

Coefficient of Variation of Count: 99.98

Average Execution Time: 7.819 ns

Standard Deviation of Time: 42.181 ns

Coefficient of Variation of Time: 539.49

Recursive Algorithm Results:

Average Critical Operation Count: 4790.88

Standard Deviation of Count: 4740.88

Coefficient of Variation of Count: 98.96

Average Execution Time: 0.564 ns

Standard Deviation of Time: 49.436 ns

Coefficient of Variation of Time: 8767.60

Test Case 5:

Case Data Size: 750

Iterative Algorithm Results:

Average Critical Operation Count: 561001.00

Standard Deviation of Count: 560951.00

Coefficient of Variation of Count: 99.99

Average Execution Time: 17.164 ns

Standard Deviation of Time: 32.836 ns

Coefficient of Variation of Time: 191.30

Recursive Algorithm Results:

Average Critical Operation Count: 7646.96

Standard Deviation of Count: 7596.96

Coefficient of Variation of Count: 99.35

Average Execution Time: 0.801 ns

Standard Deviation of Time: 49.199 ns

Coefficient of Variation of Time: 6143.36

Test Case 6:

Case Data Size: 1000

Iterative Algorithm Results:

Average Critical Operation Count: 998001.00

Standard Deviation of Count: 997951.00

Coefficient of Variation of Count: 99.99

Average Execution Time: 28.525 ns

Standard Deviation of Time: 21.475 ns

Coefficient of Variation of Time: 75.29

Recursive Algorithm Results:

Average Critical Operation Count: 10578.18

Standard Deviation of Count: 10528.18

Coefficient of Variation of Count: 99.53

Average Execution Time: 1.187 ns

Standard Deviation of Time: 48.813 ns

Coefficient of Variation of Time: 4110.81

Test Case 7:

Case Data Size: 2000

Iterative Algorithm Results:

Average Critical Operation Count: 3996001.00

Standard Deviation of Count: 3995951.00

Coefficient of Variation of Count: 100.00

Average Execution Time: 111.686 ns

Standard Deviation of Time: 61.686 ns

Coefficient of Variation of Time: 55.23

Recursive Algorithm Results:

Average Critical Operation Count: 23148.84

Standard Deviation of Count: 23098.84

Coefficient of Variation of Count: 99.78

Average Execution Time: 2.283 ns

Standard Deviation of Time: 47.717 ns

Coefficient of Variation of Time: 2090.35

Test Case 8:

Case Data Size: 3000

Iterative Algorithm Results:

Average Critical Operation Count: 8994001.00

Standard Deviation of Count: 8993951.00

Coefficient of Variation of Count: 100.00

Average Execution Time: 233.002 ns

Standard Deviation of Time: 183.002 ns

Coefficient of Variation of Time: 78.54

Recursive Algorithm Results:

Average Critical Operation Count: 36583.70

Standard Deviation of Count: 36533.70

Coefficient of Variation of Count: 99.86

Average Execution Time: 3.373 ns

Standard Deviation of Time: 46.627 ns

Coefficient of Variation of Time: 1382.22

Test Case 9:

Case Data Size: 4000

Iterative Algorithm Results:

Average Critical Operation Count: 15992001.00

Standard Deviation of Count: 15991951.00

Coefficient of Variation of Count: 100.00

Average Execution Time: 411.632 ns

Standard Deviation of Time: 361.632 ns

Coefficient of Variation of Time: 87.85

Recursive Algorithm Results:

Average Critical Operation Count: 50319.74

Standard Deviation of Count: 50269.74

Coefficient of Variation of Count: 99.90

Average Execution Time: 4.596 ns

Standard Deviation of Time: 45.404 ns

Coefficient of Variation of Time: 987.79

Test Case 10:

Case Data Size: 5000

Iterative Algorithm Results:

Average Critical Operation Count: 24990001.00

Standard Deviation of Count: 24989951.00

Coefficient of Variation of Count: 100.00

Average Execution Time: 816.839 ns

Standard Deviation of Time: 766.839 ns

Coefficient of Variation of Time: 93.88

Recursive Algorithm Results:

Average Critical Operation Count: 64592.42

Standard Deviation of Count: 64542.42

Coefficient of Variation of Count: 99.92

Average Execution Time: 6.493 ns

Standard Deviation of Time: 43.507 ns

Coefficient of Variation of Time: 670.01

## Comparison of the critical operation results and the actual execution time measurements

Looking at the first set of graphs, the separation of implementation efficiency is apparent from the very beginning, with the iterative sort running over 10,000 comparisons at a data size of just 250. In comparison, the recursive heapsort only gets an average critical operation count over 10,000 at 1,000 data size. At just 250 data size, the iterative heapsort skyrockets from 9801 comparisons at 100 data to 62001 comparisons! At these same intervals the recursive heapsort jumped from 264.76 to 679.14, only a little more than double its previous count.

Looking at the execution times for both algorithms also brings us to the same conclusion. From the get-go, the iterative sort trails behind exponentially as the recursive sort barely exceeds 1 full nanosecond by data size 1000. By contrast, iterative exceeded 1.5 ns by data size 100. No matter which way you look at it, recursive heapsort is more cost effective and time efficient than iterative heapsort.

## Discussion of the significance of the coefficient of variance results and how it reflects the data sensitivity of Selected algorithm

Coefficient of variation is the ratio of standard deviation and mean. The main purpose of coefficient of variation is to study quality assurance by measuring the dispersion of the population data of a probability or frequency distribution, or by determining the content or quality of the sample data of substances.

I believe that the coefficient of variance would be useful in comparing how efficient the different implementations of heapsort were. Alongside efficiency, the CV would be useful in comparing the ratio of execution time and critical operation count within each implementation.

Interestingly enough, the iterative implementation had a much smaller coefficient of variance than the recursive implementation. I was under the impression that it should be the other way since the iterative algorithm seemed to fluctuate much more with its execution times and critical count. But I believe I am using the correct formula of (SD/mean) \* 100.

## Comparison of Results vs. Big-Θ analysis

For my actual results, I could see that Big Θ analysis and actual results were different. Below is a table of the best/worst cases vs the actual cases:

|  |  |  |  |
| --- | --- | --- | --- |
| Critical Operations | | | |
| Data Set | Iterative Actual | Recursive Actual | Best/Worst Case O(nlogn) |
| 50 | 2401 | 314.76 | 84.94850022 |
| 100 | 9801 | 729.14 | 200 |
| 250 | 62001 | 2144.18 | 599.4850022 |
| 500 | 249001 | 4790.88 | 1349.485002 |
| 750 | 561001 | 7646.96 | 2156.295948 |
| 1000 | 998001 | 10578.18 | 3000 |
| 2000 | 3996001 | 23148.84 | 6602.059991 |
| 3000 | 8994001 | 36583.7 | 10431.36376 |
| 4000 | 15992001 | 50319.74 | 14408.23997 |
| 5000 | 24990001 | 64592.42 | 18494.85002 |

You can see from this table that the calculated Big O is way off from what was actually generated from the formulas. I am wondering if this is an error of mine or just a deviation from the norm.

# Conclusion

Based on the results of both implementations of heap sort, it is clear that the recursive implementation is more efficient in critical count and time than the iterative implementation. Creating the heap is O(nlogn) Popping items is O(1), and fixing the heap after the pop is logn. There are n pops, so there is another O(nlogn) factor, which is O(nlogn) overall. This behavior makes heapsort ideal for real-time applications

Honestly, I feel as if I did not correctly implement the iterative version of insertion sort. I knew that the recursive implementation was going to perform better than the iterative implementation, but not by this big of a margin. I was under the impression that both implementations would perform around the same. The critical operation count of the recursive implementation appears to be correct, but I am not convinced that the iterative implementation’s critical operation counts are.

In conclusion, if the heap sort algorithm must be used on a large data set, the recursive implementation should be used. However, it is important that the system has enough memory to support the space complexity of the recursion.

# References

Baeldung. (2017, August 12). How to Warm Up the JVM. Retrieved February 28, 2018, from <http://www.baeldung.com/java-jvm-warmup>

Pandey. (2008, July & aug.). Study and Comparison of Various Sorting Algorithms. Retrieved March 01, 2018, from <http://dspace.thapar.edu:8080/jspui/bitstream/10266/583/3/T583.pdf>