CMSC 451 Project 2 – Heap Sort
Esther Ho

Heap Sort Analysis

Introduction

A. About Heap Sort

- A comparison-based sorting algorithm based on a binary heap data structure (A complete binary tree where a parent can only have 2 children nodes. All levels are filled, except the last level where nodes are placed as left as possible).
- Uses the heap property to sort elements. I chose to use a max heap, where the maximum element is always the root, and parent nodes must be greater than children nodes. A min heap can be used instead, where the minimum element is the root, and parent nodes must be lesser than children nodes.
- This implementation uses an array to sort elements (zero-indexed). Let a parent node be at index i, the left child node will be at index 2i +1, and the right child node will be at 2i +2.
- This version of heap sort is an in-place sort, but not a stable sort.

B. High-level pseudocode for Heap Sort Implementations (Iterative and Recursive)

General Algorithm:

1. Build max heap
2. Swap largest element (at root) to the end of the heap. Decrement heap size, then percolate item down to restore heap order property (heapify).
3. Repeat step 2 until heap size is 0. The result is the sorted array.

```
// ITERATIVE
public iterativeHeapSort (list)

     // build heap
     for i ←0 to heapSize

          n = heapSize // end of current heap

          // start at bottom, and go up til we reach root
          while (n > 0)

               p = (n-1)/2 // index of n's parent node
```

```
            if (list[n] > list[p])
                 swap (list[n], list[p])
                 n = p; // check parent now

            // max heap property remains intact
            else
                 break
       END while
  END for

  // heapify, and "remove max" from heap
  while (heapSize > 0)
       swap (list[0], list[--heapSize])
       n = 0 // index of element being swapped down

       // go down each level of the heap and keep heap
       property
       while true
            left = 2n + 1

            // LAST level of Heap

            // element has no left child
            if (left >= heapSize)
                 break // bottom, heap is heapified

            right = left + 1

            // element has left child, but no right
            if (right >= heapSize)

                 if (list[left] > list[n])
                      swap(list[left], list[n])
                 break // heapified

            // MORE levels of heap to go through

            // left > n
            if (list[left] > list[n])

                 // left > right && left > n
                 if (list[left] > list[right])
                      swap(list[left], list[n])
                      n = left
                      continue // look at left child
```

```
                    // right > left > n
                    else
                         swap (list[right], list[n])
                         n = right // look at right child
                         continue

               // n > left
               else

                    // right > n > left
                    if (list[right] > list[n])
                         swap (list[right], list[n]
                         n = right // look at right child
                         continue

                    // n > left && n > right
                    else
                         break // heapified

          END while inner

     END while outer

     return list
// END iterative heap sort

//RECURSIVE
public recursiveHeapSort(list, size)

     buildMaxHeap(list, size)

     for i = size-1 to 0{
          swap (list[0], list[i])

          maxHeapify(list, i, 0)
     }

     return list
//END recursive heap sort

private buildMaxHeap(list, size)
     for i = n/2 -1 to 0
          maxHeapify(list, size, i)

// END build max heap

private maxHeapify(list, size, root)
```

```
      // indices
      largest = root
      left = 2i + 1
      right = 2i + 2

      if (left < size) AND (list[left] > list[largest])
           largest = left

      if (right < n) AND (list[right] > list[largest])
           largest = right

      if (largest != i)
           swap(list[i], list[largest])
           maxHeapify(list, n, largest)

// END max heapify
```

C. Big-θ Analysis of the two Heap Sort versions

- Best case: θ (nlgn)
- Average case: θ (nlgn)
- Worst case: θ (nlgn)

Heap sort is interesting, because the best, worst, and average cases are all θ (nlgn).

Building the heap is θ (n), as we must take each element to place it in the heap.

One call of maxHeapify() operation takes θ(lg n) as our heap is a binary heap where the height of the original heap is lgn. This maxHeapify() operation is completed n times, for the original heap and each subsequent sub-heap (with n-1 elements).
→ maxHeapify: θ(nlgn)

All together, the performance is θ(n + nlgn) = θ(nlgn)

We must maxHeapify for every sub-heap, since we swap the first and last element, which throws off the order. That means the number of comparisons made is about the same, regardless of the original order. There may be a difference in constant from the number of swaps that happen.

- The iterative and recursive implementations share this run time complexity of θ(nlgn). Although the implementations are different in design, the overall result is the same: you must continue to compare elements down the heap until the

heap order is restored.

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

The JVM uses the JIT compiler before running a program, where bytecode is compiled into native code, and leads to an overall improvement in performance at run time. But the JIT must perform optimizations and garbage collection, which could be a massive number when sample sizes or program sizes are huge.

So this approach is from baeldung.com, where a Dummy class is instantiated 100,000 times in a static method. This is a manual warm-up to be performed before creating a BenchmarkSorts object to start the benchmarking process.

One test run had a warm up time of 12150153 ns, and another run had a warm up time of 9692071 ns.

E. Discussion of Critical Operation chosen to count & why it was selected

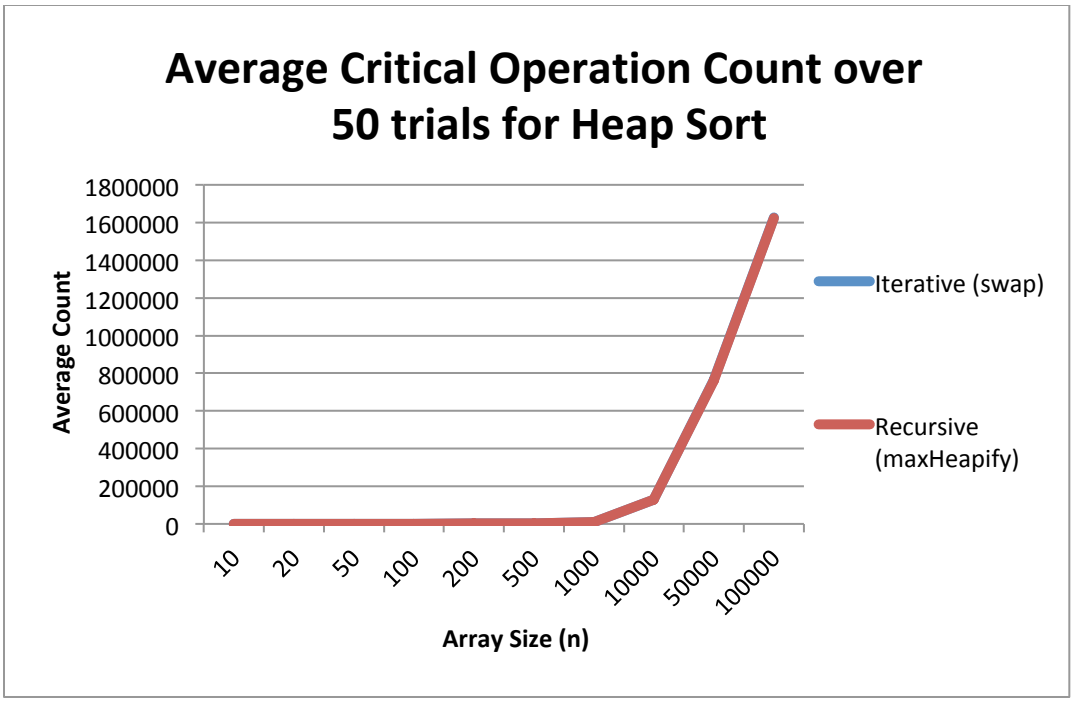
- For the iterative sort, I chose the number of calls to swap() as the critical operation. This will have a positive correlation with the sample size, but will also reflect the execution time growth relationship as sample size increases. Swapping elements is the operation that eventually sorts the whole array.
- For the recursive sort, I chose the number of calls to maxHeapify() as the critical operation. This is a recursive function, and reflects how many times the algorithm checks that the heap order property is maintained. The number of calls to maxHeapify does not reflect the number of swaps, as swaps only happen when the heap order is broken.

Analysis

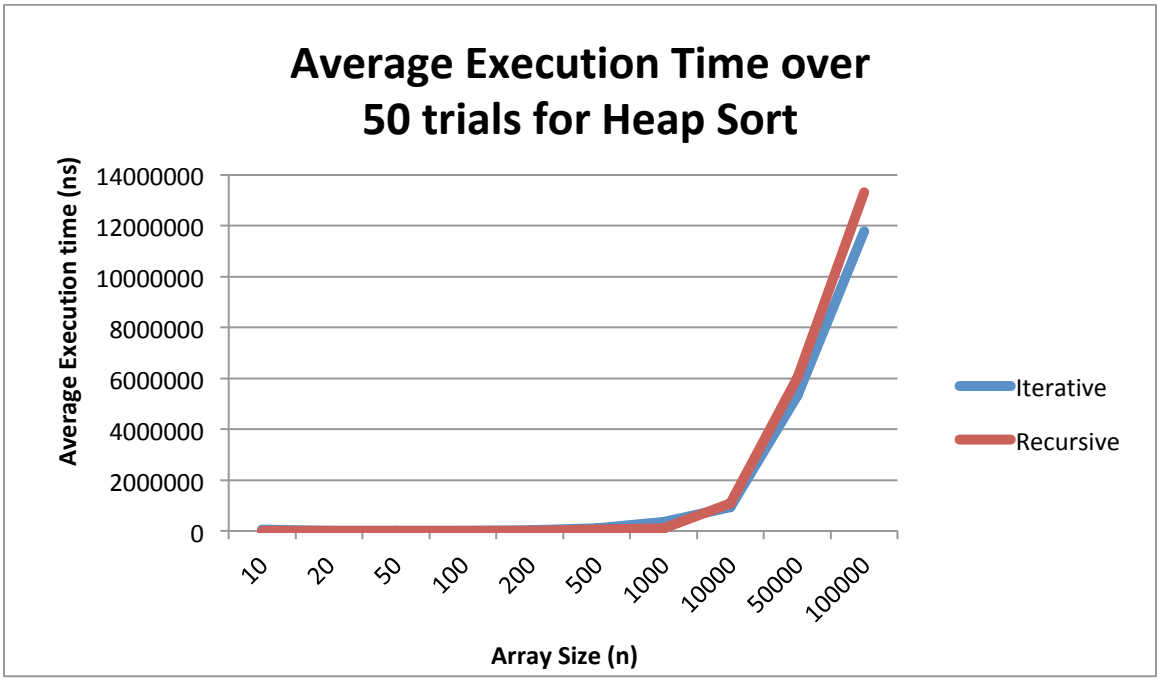
A. Graphs of Heap Sort

a. Critical Operations

CMSC 451 Project 2 – Heap Sort
Esther Ho



b. Execution Times



B. Comparison of performance – Iterative vs. Recursive

Based on the graphs above, iterative and recursive implementation of heap sort are similar in performance. In fact, you cannot see the iterative line in the first graph for the average critical operation count. If n were to increase even more, it

would be interesting to see if the recursive heap sort execution time would grow at an increasingly different rate. But at least for an array size of 100,000, the two heap sorts perform about the same.

C. Comparison of critical operation results & actual execution time measurements

Again, the critical operation count and execution times are pretty similar between iterative and recursive heap sort. One interesting thing to note is the execution time. The average execution time is greater for both iterative and recursive heap sort when n = 10 versus n = 20. Also, there are spikes in execution time for a medium size array (n = 50 or n = 100).

Sample outputs

Trial 1:

Benchmark Results for Heap Sort:

Size N	Iterative				Recursive			
	Average Critical Operation Count	Coefficient of Variance Count	Average Execution Time (ns)	Coefficient of Variance Time (ns)	Average Critical Operation Count	Coefficient of Variance Count	Average Execution Time (ns)	Coefficient of Variance Time (ns)
10	29.54	8.9619	85590.02	653.6637	33.06	5.6235	7345.28	85.1845
20	77.72	5.7486	8517.74	66.9597	82.04	3.2061	4264.04	99.2159
50	260.16	3.3513	20683.62	108.8415	266.92	1.9977	22205.64	382.7413
100	624.58	2.0611	13729.14	30.1115	632.58	1.0113	28393.74	366.9062
200	1453.00	1.2593	25473.26	21.8274	1460.72	0.6523	16489.88	37.8733
500	4287.40	0.7122	91363.90	22.8927	4294.92	0.3074	64479.02	133.4716
1000	9605.12	0.5676	275648.70	84.6441	9583.34	0.2283	100442.40	24.4640
10000	129537.46	0.1040	1088182.44	56.5618	129211.20	0.0498	1092291.26	17.7938
50000	764348.98	0.0466	5308711.52	5.9954	762480.78	0.0222	6059232.96	7.0311
100000	1628678.38	0.0300	11617161.34	7.9018	1624897.44	0.0135	13462060.24	13.9539

Trial 2:

Benchmark Results for Heap Sort:

Size N	Iterative				Recursive			
	Average Critical Operation Count	Coefficient of Variance Count	Average Execution Time (ns)	Coefficient of Variance Time (ns)	Average Critical Operation Count	Coefficient of Variance Count	Average Execution Time (ns)	Coefficient of Variance Time (ns)
10	29.34	9.6789	67885.58	638.2677	32.58	5.1393	5931.26	54.8706
20	77.52	6.5281	10398.90	123.1906	82.88	4.0425	5768.66	161.3464
50	262.54	3.3555	14777.10	92.4518	268.72	1.8607	10635.92	41.1402
100	623.28	2.1144	14248.00	49.3980	631.34	1.2070	13191.02	26.1814
200	1452.10	1.4718	31404.58	51.8328	1458.62	0.7552	18844.00	53.0588
500	4289.76	0.7133	141112.38	78.5724	4294.46	0.3525	49633.48	32.3228
1000	9584.82	0.5254	294594.04	109.2417	9580.44	0.2221	129514.82	66.3757
10000	129568.38	0.1074	946912.46	18.4155	129196.10	0.0562	1108823.20	23.1538
50000	764334.98	0.0435	5552117.26	25.7399	762501.34	0.0220	6109714.10	7.3805
100000	1628786.64	0.0328	11201255.48	3.5758	1624943.68	0.0180	13068087.90	10.1755

CMSC 451 Project 2 – Heap Sort
Esther Ho

Trial 3:

Benchmark Results for Heap Sort:

Size N	Iterative				Recursive			
	Average Critical Operation Count	Coefficient of Variance Count	Average Execution Time (ns)	Coefficient of Variance Time (ns)	Average Critical Operation Count	Coefficient of Variance Count	Average Execution Time (ns)	Coefficient of Variance Time (ns)
10	29.36	10.4150	71315.72	653.7300	32.48	6.5526	3960.64	50.6078
20	77.70	5.4253	20239.24	421.7843	82.66	3.2786	5854.68	273.3694
50	262.36	3.7336	23294.78	286.9474	268.48	1.6777	12000.80	94.3255
100	624.66	2.4938	10837.04	22.5242	631.92	1.1675	10445.84	15.7496
200	1453.86	1.5213	27112.74	85.8974	1461.54	0.7363	15676.74	43.6586
500	4292.56	0.9935	77625.20	40.0204	4294.78	0.4185	39372.78	16.2619
1000	9589.64	0.5438	269220.72	70.6852	9584.70	0.2716	103358.26	24.2866
10000	129566.90	0.1447	947552.52	14.5007	129215.84	0.0608	1080990.34	15.8237
50000	764469.96	0.0461	5504086.22	8.1377	762526.88	0.0186	6505090.82	27.0220
100000	1628859.48	0.0359	11354979.70	5.0328	1624945.90	0.0171	13047807.00	5.7205

D. Significance of coefficient of variance results

The coefficient of variance is a measure of relative variability. The coefficient of variance for the critical operation count tells us that as the size n increases, the variability between the 50 trials decreases to almost 0. In contrast, the coefficient of variance for the execution time is much bigger. In particular, this measure for recursive heap sort reached over 100 for some trials (Trial 3, n = 20, Trial 1, n = 50 & 100). This implies that the 50 randomly generated arrays for each array size varied greatly in the time it took to sort recursively. My guess as to why is because of how the arrays were generated. Perhaps the time performance can be different based on if the array is mostly sorted, mostly opposite order, or scrambled.

E. Results compared to Big-θ Analysis

Compared to the Big- Θ analysis of Θ(nlgn), the iterative and recursive critical operation counts were very similar. The calculated nlgn provided an upper bound for the actual counts.

CMSC 451 Project 2 – Heap Sort
Esther Ho

Critical Operation			
Size (n)	**Iterative**	**Recursive**	**Θ(nlgn)**
10	29.16	32.64	33.22
20	77.24	82.12	86.44
50	259.46	267.08	282.19
100	623.46	631.66	664.39
200	1455.52	1462.76	1528.77
500	4287.9	4292.36	4482.89
1000	9595.94	9583.14	9965.78
10000	129559.34	129208.28	132877.12
50000	764416.1	762471.16	780482.02
100000	1628892.88	1624994.58	1660964.05

Conclusion

Heap sort is not the worst sort out there when comparing time complexity, as best, worst, and average case all are Θ(nlgn). As previously mentioned, my implementations are in-place sorting which improves the space complexity. But it is not a stable sort, meaning that if I were to sort data pairs of (x, y), pairs with the same x but different y's will be together but may not be ordered by y value. Due to the nature of heap sort, to repetitively compare an element to its child nodes (breaking up the heap into a mini heap), implementing the sort iteratively ends up being quite similar to the recursive sort. When I first saw the results, I was not sure of them because I am used to the iterative and recursive versions of functions having a different time complexity.

My implementations ended up being very similar in time performance and critical operation count. My results matched with the big-theta analysis. Heap sort is a better sort when it comes to larger data sets, compared to a sort like insertion sort or bubble sort with an average case $\Theta(n^2)$.

CMSC 451 Project 2 – Heap Sort
Esther Ho

References:

"Algorithm Implementation/Sorting/Heapsort." [For Iterative Heap Sort]. *Development Cooperation Handbook/Guidelines/How to Manage Programmes for a Learning Organization That Is Projectized and Employee Empowering - Wikibooks, Open Books for an Open World*, 11 Apr. 2018, en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Heapsort#Java_2.

Baeldung. "How to Warm Up the JVM." *Baeldung*, 15 Apr. 2018, http://www.baeldung.com/java-jvm-warmup.

"HeapSort." [For Recursive Heap Sort]. *GeeksforGeeks*, 7 Sept. 2018, www.geeksforgeeks.org/heap-sort/.