Sorting Coursework

Fundamentals of Computer Science

Oliver Barnwell
25011080

February 23, 2018

Contents

1 Abstract **2**

2 Algorithm Description **2**

3 Pseudocode **3**

4 Implementation **4**

4.1 Heapsort Entry Method 4

4.2 Iterative Sift (First Sift Variant) 5

4.3 Recursive Sift (Second sift variant) 6

4.4 Heapify 7

4.5 Custom Comparator 7

4.6 Helper Methods 8

5 Testing **8**

5.1 Can sort the empty list of length 0? 8

5.2 Can sort a list of length 1? 8

5.3 Can sort a list of length 2? 9

5.4 Can sort a list of variable length? 9

5.5 Can sort a list with at least two equal elements? 9

5.6 Can sort minus zero? 9

5.7 Can sort positive and negative infinity? 10

5.8 Can handle at least one NaN? 10
5.9 Can handle at least two NaNs with different mantissas? 10
5.10 Can handle a list of pseudorandom data? 10

6 Time Order Prediction **11**

7 Time Measurements **12**
7.1 Graphed Data 18
7.2 Discussion of data 19
7.3 Discussion of potential performance improvements 20

1 Abstract

The purpose of this coursework is to perform an in-depth investigation and analysis of a specific sorting algorithm: **Heap Sort**. The design of the algorithm is than described in plain English. An implementation of the algorithm is provided along with the relevant pseudocode. Supplementing this is the code that describes the necessary test runs along with the results. Additionally, the time order of the algorithm is predicted and the subsequent data returned from running the algorithm is graphed and tabulated. Finally the performance of the algorithm is discussed in addition to how the implementation and algorithm may be improved.

2 Algorithm Description

Heap Sort can be described as "a comparison-based sorting algorithm [1]" meaning that, when sorting, the algorithm relies upon the **direct comparisons of list elements** to make decisions. It makes use of certain properties that are an inherent part of the binary heap data structure. Specifically the fact that, when a complete binary heap is formed, the value of each node is less than or equal to its parent.

A binary heap is constructed from the list of elements to be sorted. The root parent of the heap is continually swapped with the last element in the heap, with the heap being "repaired" in order to place the freshly swapped last element in the correct position. The root element, after being swapped, is considered 'sorted' and is no longer considered as part of the sorting process. Following each iteration the considered size of the heap is decremented by one.

The process of swapping the root element with the last element in the heap and repairing the heap is then repeated until the considered heap size is equal

to one element. At this point the list is known to be sorted.

3 Pseudocode

The following pseudocode entries are sourced from Wikipedia [1].

```
procedure heapsort(a, count) is
    input: an unordered array a of length count

    (Build the heap in array a so that largest value is at the root)
    heapify(a, count)

    (The following loop maintains the invariants that a[0:end] is a
    heap and every element
     beyond end is greater than everything before it (so a[end:count]
    is in sorted order))
    end ← count - 1
    while end > 0 do
        (a[0] is the root and largest value. The swap moves it in front
     of the sorted elements.)
        swap(a[end], a[0])
        (the heap size is reduced by one)
        end ← end - 1
        (the swap ruined the heap property, so restore it)
         siftDown(a, start, end)
```

Listing 1: Heapsort Entry Method. [1]

```
procedure heapify(a, count) is
    (start is assigned the index in 'a' of the last parent node)
    (the last element in a 0-based array is at index count-1; find the
    parent of that element)
    start ← iParent(count-1)

    while start  0 do
        (sift down the node at index 'start' to the proper place such
    that all nodes below
         the start index are in heap order)
        siftDown(a, start, count - 1)
        (go to the next parent node)
        start ← start - 1
    (after sifting down the root all nodes/elements are in heap order)
```

Listing 2: Put elements of 'a' in heap order; in-place. [1]

```
procedure siftDown(a, start, end) is
    root ← start

    while iLeftChild(root)  end do    (While the root has at least one
    child)
        child ← iLeftChild(root)   (Left child of root)
        swap ← root                (Keeps track of child to swap with)

        if a[swap] < a[child]
            swap ← child
```

```
        (If there is a right child and that child is greater)
        if child+1  end and a[swap] < a[child+1]
            swap ← child + 1
        if swap = root
            (The root holds the largest element. Since we assume the
    heaps rooted at the
             children are valid, this means that we are done.)
            return
        else
            swap(a[root], a[swap])
            root ← swap            (repeat to continue sifting down the
     child now)
```

Listing 3: Repair the heap whose root element is at index 'start'; assuming the heaps rooted at its children are valid [1]

4 Implementation

This implementation of heap sort was written using the Golang language. The key parts of the implementation that are essential for it to function are included as code below. Wherever relevant I have made efforts to comment the source code for additional clarity.

A quick note on the terminology used in comments that explain functionality: The term "***slice***" is used in place of "array". In Golang a slice represents a "dynamically sized, flexible view into the elements of an array [2]". As slices are passed as references to the underlying array in the code this terminology is used for the sake of consistency with the language when commenting.

4.1 Heapsort Entry Method

```
/**
 * Completes a heapsort on the provided slice.
 *
 * a []float64
 *   The slice to be sorted.
 */
func heapsort(a []float64) {
  count := len(a)

  // Construct the initial heap.
  heapify(a, count)

  // Initialise the end as the last element in the heap (smallest).
  end := count - 1

  // Until the end escapes the bounds of the slice iterate.
  for(end > 0) {
    // Swap the last element with the first.
    swap(a, end, 0)

```

```
    // We now consider the new last element to be fixed, so we
    decrement end so it is not considered.
    end--

    // "Sift" the new first element into the correct position in the
    heap.
    siftDownIterative(a, 0, end)
  }
}
```

Listing 4: Main control method for the Heap Sort.

4.2 Iterative Sift (First Sift Variant)

```
/**
 * Repairs the provided heap where the root element is at index start.
 *   This method is applied iteratively.
 *
 * a []float64
 *   The heap to be repaired (heapified).
 * start, end int
 *   The start and end indexes.
 *
 */
func siftDownIterative(a []float64, start, end int) {
  // Set the root index to the specified start.
  root := start

  // While the index of the root's left child is less than the
    specified end index iterate. Instead this could be implemented as a
     recursive method.
  for(iLeftChild(root) <= end) {
    // Calculate the children's indexes and initialise a variable to
    keep track of the largest child.
    left := iLeftChild(root)
    right := left + 1
    largestChild := root

    // Compare root with its left child.
    // If left child is greater choose it.
    if sortCompareLessThan(a[largestChild], a[left]) {
      largestChild = left
    }

    // Compare current largest with right child.
    // If right child is larger choose it.
    if right <= end && sortCompareLessThan(a[largestChild], a[right]) {
      largestChild = right
    }

    // Check if the root is the largest of it and its children.
    // If so we have achieved our goal of ordering the tree.
    if largestChild == root {
      return
    }

    // Otherwise, swap the largest child with the original root.
```

```
    swap(a, root, largestChild)

    /// Assign a new root for the next iteration.
    root = largestChild
  }
}
```

Listing 5: The procedure which restores the properties of the heap following the removal of the root node. This implementation follows the pseudocode listed earlier which operates iteratively.

4.3 Recursive Sift (Second sift variant)

```
/**
 * Repairs the provided heap where the root element is at index start.
 *   This method is applied recursively.
 *
 * a []float64
 *   The heap to be repaired (heapified).
 * start, end int
 *   The start and end indexes.
 *
 */
func siftDownRecursive(a []float64, start, end int) {
    // Calculate the children's indexes and initialise a variable to
    keep track of the largest child.
    left := iLeftChild(start)
    right := left + 1
    largestChild := start

    // Compare root with its left child.
    // If left child is greater choose it.
    if left <= end && sortCompareLessThan(a[largestChild], a[left]) {
      largestChild = left
    }

    // Compare current largest with right child.
    // If right child is larger choose it.
    if right <= end && sortCompareLessThan(a[largestChild], a[right]) {
      largestChild = right
    }

    // Check if the root is the largest of it and its children.
    // If so we have achieved our goal of ordering the tree.
    if largestChild != start {
      // Otherwise, swap the largest child with the original root.
      swap(a, start, largestChild)
      siftDownRecursive(a, largestChild, end)
    }
}
```

Listing 6: The procedure which restores the properties of the heap following the removal of the root node. This implementation is functionally the same as its iterative counterpart.

4.4 Heapify

```
/**
 * Create the initial heap structure.
 *
 * a []float64
 *   Slice to be heapified.
 * count int
 *   Max size of the heap
 */
func heapify(a []float64, count int) {
  start := iParent(count - 1)
  for(start >= 0) {
    siftDownIterative(a, start, count - 1)
    start--
  }
}
```

Listing 7: Method used to convert the given list of elements into a complete binary heap.

4.5 Custom Comparator

```
/**
 * Custom comparison function which returns the desired results when
    comparing against NaN and negative zero.
 *
 */
func sortCompareLessThan(a, b float64) bool {
  var result bool = false
  aNaN := math.IsNaN(a)
  aSign := math.Signbit(a)
  bSign := math.Signbit(b)

  // Base less than check.
  if a < b {
    result = true
  }

  // Check for NaN.
  if aNaN {
    result = true
  }

  // Check for negative zero.
  if (a == 0 && aSign) && (b == 0 && !bSign) {
    result = true
  }

  return result
}
```

Listing 8: Custom comparison function used in order to sort negative zero entries and NaN entries correctly

4.6 Helper Methods

```
/**
 * Returns the array index of the specified heap parent.
 */
func iParent(i int) int {
  return int(math.Floor(float64((i - 1) / 2)))
}

/**
 * Returns the array index of the specified left heap child.
 */
func iLeftChild(i int) int {
  return 2 * i + 1
}

// Swaps two elements in the provided slice.
func swap(a []float64, i, j int) {
  a[i], a[j] = a[j], a[i]
}
```

Listing 9: Helper functions used by the implementation. iParent and iLeftChild are both methods related to heap management and swap simply swaps elements in the provided slice.

5 Testing

A method was created called "testRunner" in order to reduce code repetition, its job is to print the name of the test; execute the heapsort method and print the sorted output.

It is defined as follows:

```
func testRunner(arr []float64, message string) {
  fmt.Printf("%s\n", message)
  heapsort(arr)
  fmt.Printf("%v\n", arr);
}
```

Listing 10: testRunner definition

5.1 Can sort the empty list of length 0?

```
  testArray := []float64{}
  testRunner(testArray, "Sorting an empty list.")
```

Test Output:

```
Sorting an empty list.
[]
```

5.2 Can sort a list of length 1?

```
testArray = []float64{1.0}
testRunner(testArray, "Sorting a list with one element.")
```

Test Output:

```
Sorting a list with one element.
[1]
```

5.3 Can sort a list of length 2?

```
testArray = []float64{2.0, 1.0}
testRunner(testArray, "Sorting a list with two elements.")
```

Test Output:

```
Sorting a list with two elements.
[1 2]
```

5.4 Can sort a list of variable length?

```
testArray = []float64{2.0, 4.0, 1.0, 5.0}
testRunner(testArray, "Sorting a list with a general number of
    elements")
```

Test Output:

```
Sorting a list with a general number of elements
[1 2 4 5]
```

5.5 Can sort a list with at least two equal elements?

```
testArray = []float64{6.0, 1.0, 2.0, 1.0, 3.0, 100.0, 1.0}
testRunner(testArray, "Sorting a list where some elements are the
    same")
```

Test Output:

```
Sorting a list where some elements are the same
[1 1 1 2 3 6 100]
```

5.6 Can sort minus zero?

```
zero := float64(0)
neg_zero := -zero
testArray = []float64{-100.0, neg_zero, -50.0, -1.0, 3.0, 0.0, 2.0,
    neg_zero, 1.0, neg_zero, 5.0, 0.0, 0.0, 1.0}
testRunner(testArray, "Sorting a list where some of the elements are
    minus zero")
```

Test Output:

```
Sorting a list where some of the elements are minus zero
[-100 -50 -1 -0 -0 -0 0 0 0 1 1 2 3 5]
```

5.7 Can sort positive and negative infinity?

```
testArray = []float64{1.0, 100.0, math.Inf(1.0), 2.0, math.Inf(-1.0)}
testRunner(testArray, "Sorting a list where some of the elements are
    infinity and minus infinity")
```

Test Output:

```
Sorting a list where some of the elements are infinity and minus
    infinity
[-Inf 1 2 100 +Inf]
```

5.8 Can handle at least one NaN?

```
  testArray = []float64{math.NaN(), 6.0, 1.0, math.NaN()}
  testRunner(testArray, "Sorting a list where some elements are NaN")
```

Test Output:

```
Sorting a list where some elements are NaN
[NaN NaN 1 6]
```

5.9 Can handle at least two NaNs with different mantissas?

```
var example_nan float64 = math.Sqrt(-1)
var example_nan_two float64 = math.Sin(math.Inf(1.0))
testArray = []float64{-1.0, 6.0, 2.0, math.NaN(), 2.0, 1000.0,
    example_nan, example_nan_two}
testRunner(testArray, "Sorting a list where some elements are NaN with
    different mantissas.")
```

Test Output:

```
Sorting a list where some elements are NaN with different mantissas.
[NaN NaN NaN -1 2 2 6 1000]
```

5.10 Can handle a list of pseudorandom data?

```
  testArray = make([]float64, 100)
  for i := 0; i < 100; i++ {
    testArray[i] = rand.Float64() * 100
  }
  testRunner(testArray, "Sorting a list of 100 pseudorandom numbers")
```

Test Output:

```
Sorting a list of 100 pseudorandom numbers
[1.7875329270832938 3.743559301738659 4.017586617227677
    4.255112727065778 4.446517853465413 4.936423000749997
    6.458537413289717 7.584119594457869 7.935323568733206
    7.940532916721166 9.298925122599364 9.554338988456838
    9.743346776198326 11.267742487293187 11.541355956923764
    13.024722640673023 14.734749617745937 16.18378748116687
```

```
16.338610634495225 16.51109391752422 16.55671266229901
16.628125301691185 17.04874238394088 17.285085359363286
17.883564149947865 19.602424553964543 19.86991367908718
20.080611912602507 21.046273030949948 21.1197590505135
21.67016965394035 22.057900471381764 22.60829731529768
24.505327051589234 24.580741782392494 24.86248599239121
25.349681439044492 25.85300833937772 26.25285600925247
27.395079786558817 28.554483193467284 28.574424476815114
29.03477233007291 29.728363460089962 31.231634532055967
32.065657848404356 32.947691554606926 33.54220123116506
34.12447111923807 36.11870264889829 37.633441367560565
37.650824884336274 40.05781874566627 40.177868196700395
42.54073755530162 44.305141417822206 44.729156852268275
46.025105348458254 46.235451686381765 47.665701520209964
49.80139292282418 50.2798120026658 51.89604887842768
52.00540772247994 52.058247871909614 52.24384878044316
55.479357773635776 57.76574819375164 58.769131842412214
59.637440933228994 60.075571174487585 60.16529502413248
60.355092688221326 61.47145446055787 61.757571850682325
61.84822706690224 62.82124946008558 62.99292367316859
66.41707595964473 66.9746144564068 68.80214028294158
71.09202402609719 75.15333898976276 78.9364267191067
80.67305181721875 80.74083706993265 81.59306266226429
81.68331910754362 82.3146694361774 84.568434368249
85.95650511136442 86.02600302498495 86.56350810015844
87.63513041431477 90.16088627181635 90.23617098955718
92.18860614727073 92.80533600055378 92.85501568685352
99.53003436157013]
```

6 Time Order Prediction

The siftDown method as implemented for this sort is better known as the "Max-Heapify" algorithm, used in order to correct a single violation of the heap property in the root of a subtree. It recursively compares the root which needs fixing with the left and right children below it and swaps with the largest child until it is larger than the nodes directly below it or there are no nodes left to traverse.

The worst possible case for the MaxHeapify algorithm would be for a swap to happen for every level of the heap from the root to the maximum possible level of the binary tree. This is defined as being a heap where *"the bottom level of the tree is exactly half full. [3]"*

The total number of items in the left subtree of the heap (which is a nearly complete binary tree) will therefore be double the number of items in the right subtree of the heap.

So we can define the number of elements in the right subtree as: k

The number of elements in the left subtree in the worst case will then be: $2k+1$, with the additional one element being added because it is an incomplete tree.

We can then let the total number of nodes equal $(2k+1)+(k)+(1) = 3k+2$, with another element being added to represent the root node of the heap.

Therefore the ratio of the largest subtree to the total number of elements can be considered to be: $(2k+1)/(3k+2)$ which has a limit of $2/3$.

This means that the time order of this algorithm has a recurrence relation:

$$T(n) = T(2n/3) + f(1)$$

Because the MaxHeapify algorithm follows a divide and conquer pattern; recursion analysis can be performed using the master theorem using the recurrence relation found previously as described in this quote:

"$a = 1, b = 3/2$, one obtains $log_b a = log_{3/2} 1 = 0$. Since $f(n) = (1) = (n^0)$, Case 2 applies. Thus, $T(n) = O(lgn)$. [4]"

Therefore the average time complexity of the MaxHeapify algorithm is:

O(log n)

Because the Heap Sort algorithm calls the maxHeapify method $n-1$ times (where n is the length of the list) the time order of the algorithm as a whole can be classed as:

O(n log n)

7 Time Measurements

Time measurements of the performance of the algorithm were carried out by running the implementation of Heap Sort with an array of pseudorandom floats. The list of considered floats increased in size by 500,000 per iteration. The time taken for the algorithm to execute was recorded for each iteration until a maximum limit of 100,000,000 elements was reached. For each iteration the sort was run five times and the results from each averaged in order to reduce the likelihood of any outliers in the resulting data set.

The benchmark code itself will have likely had minimal impact upon the asymptotic runtime of the algorithm. Mainly down to the fact that the benchmark code should have a very small effect relatively on the time taken to execute the program. This is also because the time for each execution of the sort was only recorded from when it began until when it finished; not taking into the time spent by the benchmark code whatsoever. Following the completion of the benchmark the program outputs the data as a two column CSV formatted data structure to STDOUT.

The benchmark was executed on an Intel i5-6200U CPU @ 2.30GHz with 8GB of RAM running Arch Linux under Linux kernel version 14.5.3.

The timing of the sorting algorithm was completed using the Golang time API. The calculation of the time difference can be seen on lines 13 - 15 of the

code listing below. The current clock time is recorded before the algorithm is executed and following its synchronous execution the difference is calculated using the “time.Since” method.

```
var factor float64 = 500000

siftCounters := make([]int, 201)
dataRecords := [][]string{{"element_count", "time"}}
for i := range siftCounters {
  var multiple float64 = float64(i) * factor
  statArray := make([]float64, int(multiple))
  for j := range statArray {
    statArray[j] = rand.Float64() * 100
  }
  var avgTime int64 = 0
  for k := 0; k < 5; k++ {
    start := time.Now()
      heapsort(statArray)
    since := time.Since(start).Nanoseconds()
    avgTime += since
  }
  avgTime /= 5
  dataRecords = append(dataRecords, []string{fmt.Sprintf("%d", int(
  multiple)), fmt.Sprintf("%d", avgTime)})
}

w := csv.NewWriter(os.Stdout)

for _, record := range dataRecords {
  if err := w.Write(record); err != nil {
    log.Fatalln("error writing record to csv:", err)
  }
}

w.Flush()

if err := w.Error(); err != nil {
  log.Fatal(err)
}
```

Table 1: Table of sorted element count vs recorded time in ms

Element Count	Sort Exectuion Time (ms)
0	0.0039
500000	692.0933
1000000	1271.609
1500000	2133.7457
2000000	2990.2826
2500000	4218.8092
3000000	4560.594
3500000	5327.5892
4000000	6209.5722
4500000	7057.7342
5000000	7937.187

Table 1 continued from previous page

5500000	8856.0616
6000000	9747.164
6500000	10671.8038
7000000	11535.9135
7500000	12448.0076
8000000	13360.6156
8500000	14368.1772
9000000	15215.399
9500000	16196.3633
10000000	17160.5968
10500000	18202.0828
11000000	19076.4491
11500000	20103.6415
12000000	21011.5467
12500000	22053.4898
13000000	23035.501
13500000	24116.1865
14000000	25019.4148
14500000	25982.59
15000000	27023.0443
15500000	28114.8373
16000000	29038.5442
16500000	30167.1442
17000000	31181.3646
17500000	32138.2924
18000000	33168.5516
18500000	34236.994
19000000	35209.8806
19500000	36329.2038
20000000	37447.664
20500000	38404.2392
21000000	39491.7524
21500000	40543.1014
22000000	41598.121
22500000	42540.6916
23000000	43680.0193
23500000	44718.7485
24000000	46001.2375
24500000	46929.2321
25000000	47930.6017
25500000	48821.5602
26000000	49278.9833
26500000	50295.4031
27000000	51439.0267
27500000	52584.1094

Table 1 continued from previous page

28000000	53468.1625
28500000	54422.4217
29000000	55506.1068
29500000	56590.1578
30000000	57932.6584
30500000	58699.5802
31000000	59794.7465
31500000	61100.5993
32000000	62232.3724
32500000	63066.101
33000000	64112.8183
33500000	65190.3467
34000000	66236.892
34500000	67464.0675
35000000	68566.0802
35500000	69381.1305
36000000	70499.1021
36500000	71523.5474
37000000	72779.9815
37500000	73847.5192
38000000	75524.11
38500000	76639.6998
39000000	77613.054
39500000	78907.9454
40000000	79708.4133
40500000	80809.1842
41000000	81946.1445
41500000	82774.1732
42000000	83923.4042
42500000	85129.1533
43000000	86265.5956
43500000	87832.1168
44000000	89326.3352
44500000	90130.6476
45000000	91200.9856
45500000	92351.103
46000000	93246.7366
46500000	94148.2823
47000000	95596.0068
47500000	96475.9357
48000000	97779.5155
48500000	99314.0977
49000000	100162.9629
49500000	101413.0147
50000000	102517.7509

Table 1 continued from previous page

50500000	103381.6555
51000000	104330.4055
51500000	105596.4873
52000000	106656.3687
52500000	108193.3463
53000000	109811.0143
53500000	110616.7096
54000000	111718.9269
54500000	113090.2793
55000000	113885.4346
55500000	114974.7252
56000000	115799.6393
56500000	117515.8967
57000000	118956.818
57500000	119901.7771
58000000	120832.9902
58500000	122165.3894
59000000	122728.1267
59500000	123775.5596
60000000	125085.7848
60500000	126734.153
61000000	127674.7336
61500000	128624.9218
62000000	129217.7624
62500000	130774.531
63000000	132049.0785
63500000	133074.1754
64000000	134290.8092
64500000	135498.6988
65000000	136371.1202
65500000	137556.4879
66000000	139093.7689
66500000	140151.3119
67000000	142236.5237
67500000	143381.0368
68000000	144005.3402
68500000	145252.36
69000000	146361.5602
69500000	147312.6666
70000000	149106.4438
70500000	150739.2312
71000000	151143.7326
71500000	152208.7733
72000000	153073.5592
72500000	154068.8833

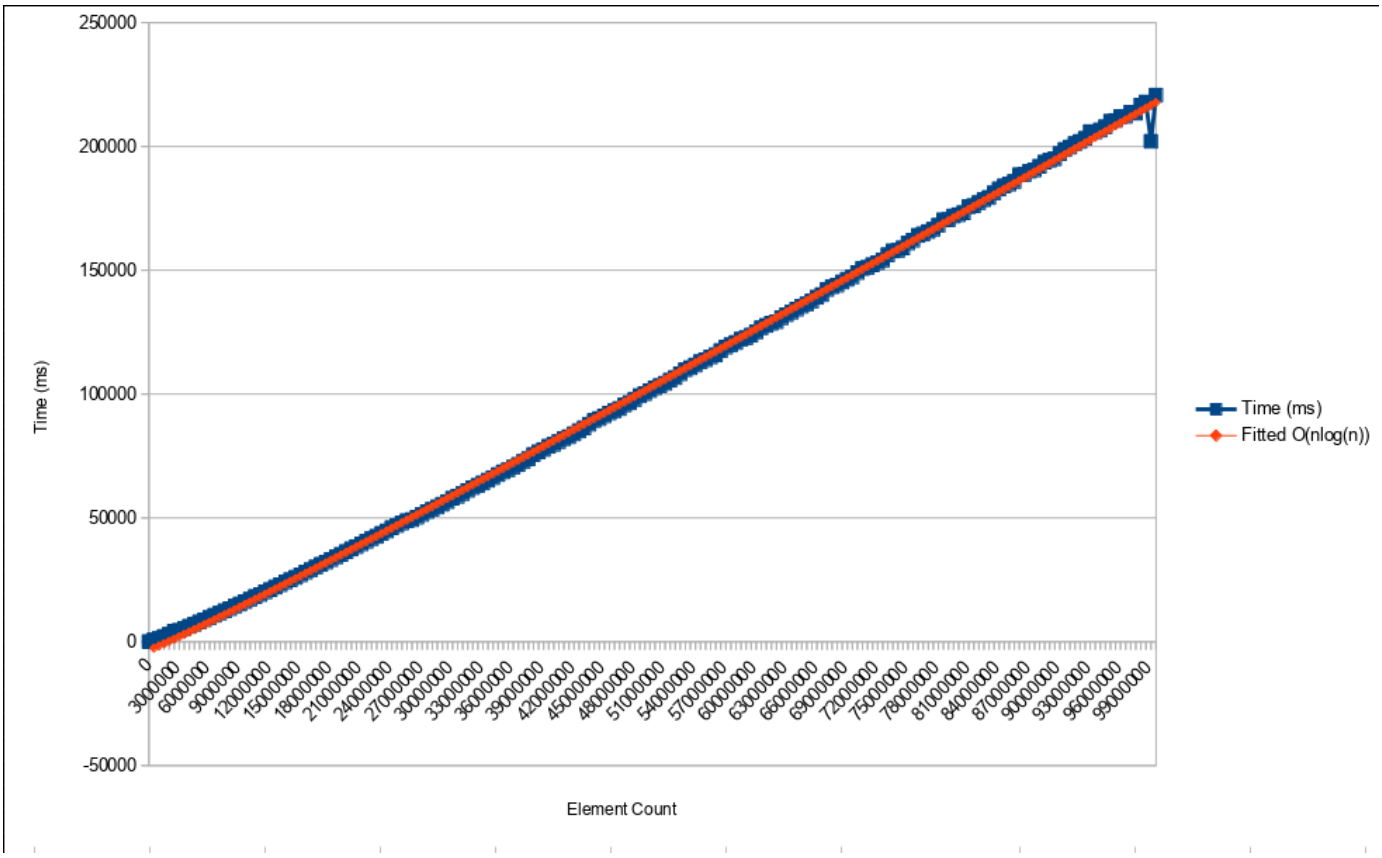
Table 1 continued from previous page

73000000	156215.0235
73500000	157804.8742
74000000	157974.9899
74500000	159010.674
75000000	161006.408
75500000	162114.7625
76000000	164041.0121
76500000	164619.24
77000000	165626.6304
77500000	166556.5088
78000000	168143.3362
78500000	170318.5359
79000000	170388.2674
79500000	171886.4311
80000000	172425.7322
80500000	173228.8225
81000000	175660.7555
81500000	176061.7178
82000000	177282.6203
82500000	178517.623
83000000	179360.0201
83500000	181234.5752
84000000	182828.8056
84500000	184030.8635
85000000	184787.8122
85500000	185842.7016
86000000	188523.1096
86500000	188630.414
87000000	190020.9836
87500000	190505.2278
88000000	191944.949
88500000	193645.5247
89000000	194442.4665
89500000	195009.3745
90000000	197068.3639
90500000	198695.0439
91000000	199738.3158
91500000	201145.4023
92000000	202009.4
92500000	203249.7879
93000000	205981.3999
93500000	205854.6877
94000000	206621.6782
94500000	207920.2284
95000000	210145.7485

Table 1 continued from previous page

95500000	210474.1532
96000000	212032.5303
96500000	212173.811
97000000	213795.0159
97500000	213416.5434
98000000	216674.4669
98500000	217932.5418
99000000	202065.8414
99500000	220716.2415
100000000	222129.946

7.1 Graphed Data



The spreadsheet software *LibreOffice* was used to render the graph and to calculate the linear regression mentioned below.

In order to provide a good benchmark for the recorded time information a fitted $O(nlog(n))$ graph is also plotted. This was calculated by finding the

linear regression between the predicted $O(log(n))$ time values and the actual time values recorded in the table above.

Once the linear regression was calculated, the slope (m) and intercept (c) were used to "fit" the standard O(nlogn) graph to the data like so:

$$O(n) = m * n * log(n) + c$$

The linear regression gradient and intercept values obtained were:

$$m = 8.36E - 05$$
$$c = -3398.13598405745$$

7.2 Discussion of data

Once plotted, the data appears to follow the predicted O(nlogn) time order very well up to around the first 80 million elements. Past this point the two curves begin to gently deviate, with the fitted line gradually starting to outperform the actual recorded times of the algorithm. There could be several explanations for this.

Perhaps the most obvious explanation is that the rudimentary linear regression-based curve fitting carried out may simply not be taking into account some factors which is causing the fitted graph to not follow the correct path.

It could be that some of the operations with lesser time complexity which are ignored when calculating the asymptotic complexity of the algorithm are gradually becoming more and more prevalent when larger numbers of elements are involved.

The CPU may have also had thermal throttling applied as the workload increased which would have slowed the overall performance of the algorithm.

There is also an fairly significant outlier at around the 95 million item mark where the algorithm performance a lot better than expected. In this scenario the random data set generated may have been particularly favourable to the algorithm; producing, by chance, a well balanced binary tree heap which would take less time for the maxHeapify function to sift through.

Overall however the predicted time order of the algorithm is followed very well, with one being able to confidently state that the actual performance of the algorithm follows that of the predicted performance to an acceptable degree after seeing the evidence presented.

7.3 Discussion of potential performance improvements

In terms of implementation improvements, both the recursive and iterative Max Heapify implementations could be compared in order to see if one has a particular time advantage over the other. Additionally, the custom comparator method *sortCompareLessThan* which is used in order to correctly compare negative zero and NaN values carries out some equality checks which could likely be avoided altogether if a different method were to be used, perhaps with some application specific modifications to the IEEE floating point specification.

A possible performance involving modifications to the algorithm is suggested in *Heapsort using Multiple Heaps* [5] which investigates using multiple heaps in the Heap Sort process as opposed to a single heap. Through doing this the whole tree will not have to be repaired using the MaxHeapify function after every iteration of the sort as in a standard Heap Sort. The basic concept of the process is described below as quoted from the original paper.

> The basic idea is to use the fact that after the removal of the root, instead of recreating the heap, the remainder of the structure can be viewed as two different heaps in which their roots are the nodes which were children of the node that has just been removed.
>
> ...
>
> In this way, when we have one heap which contains all the elements, the algorithm runs as it was originally described in [Williams, 1964]. On the other hand, when the two heaps are used, at the cost of one comparison (the two elements in the root position of the two heaps) we gain two comparisons.
>
> ...
>
> The new depth is log (n + N) – log N. An additional cost is that of finding the minimum of the N roots. Due to the breaking of the heap into N heaps, we have to examine log N less levels than before
>
> [5]

The performance of the benchmarking process could be improved by some degree by parallelising the iterative process used to execute each of the sorting benchmarks. Several sorts could be started on individual threads inside a closure which would then report the results of the sort upon finishing. Golang has good support for this kind of concurrency, using the concept of typed channels. A test implementation of a concurrent version of the same single threaded benchmark shown earlier can be seen below:

```
1  ch := make(chan []string, recordAmount)
2  wg := sync.WaitGroup{}
3
4  for i := 0; i < recordAmount; i++ {
5    wg.Add(1)
6    var multiple int = factor * i
7    go func(multiple int, ch chan []string, wg *sync.WaitGroup) {
8      statArray := make([]float64, multiple)
```

```
9       for j := range statArray {
10        statArray[j] = rand.Float64() * 100
11      }
12      start := time.Now()
13      heapsort(statArray)
14      since := time.Since(start).Nanoseconds()
15      returnVal := []string{fmt.Sprintf("%d", multiple), fmt.Sprintf("%
      d", since)}
16      ch <- returnVal
17      wg.Done()
18    }(multiple, ch, &wg)
19  }
20  wg.Wait()
21  close(ch)
```

Listing 11: Concurrent sorting benchmark runner

References

[1] "Heap sort." Available at https://en.wikipedia.org/wiki/Heapsort.

[2] "Go tour." Available at https://tour.golang.org/moretypes/7.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction To Algorithms Third Edition*. 2009.

[4] "Heap sort - data structures and algorithms." Available at http://www.cems.uvm.edu/~rsnapp/teaching/cs124/notes/cs124notes_031914.pdf.

[5] D. Levendeas and C. Zaroliagis, "Heapsort using multiple heaps," tech. rep., Department of Computer Engineering and Informatics and Computer Technology Institute, N. Kazantzaki Str, Patras University Campus. Available at http://students.ceid.upatras.gr/~lebenteas/Heapsort-using-Multiple-Heaps-final.pdf.