How Many Ways Can YOU Sort a Heap? A Survey of Modifications to Heapsort

Gradeigh D. Clark, Rutgers University
gradeigh.clark@rutgers.edu

Heapsort is a unique, unstable in-place sorting algorithm that guarantees $O(N\ log\ N)$ complexity with $O(1)$ additional space. This paper offers a survey of improvements and modifications to Heapsort that are meant to address some of its issues (cache-inefficiency, Quicksort out-performance, et cetera) as a cursory review of what computer scientists in the field have done with this an algorithm. A focus is put on analyzing the complexity effects of the modifications to Heapsort's structure with pseudocode and graphical analysis for the different types. The algorithms that are analyzed are: Heapsort, d-ary Heapsort, QuickHeapsort, and Smoothsort.

Contents

This work is for Rutgers Course 16:332:573 - **Data Structures and Algorithms**, with Dr. Shantenu Jha. The template used herein is a modification of the **ACM Small Standard Format**, available in full at http://www.acm.org/publications/latex_style/v2-acmsmall.zip.

May 2014.

1. INTRODUCTION

Sorting can be considered to be one of the most (if not THE most) fundamental and important tasks in computer science. The number of applications where sorting is either required by default or appears as a subroutine in an algorithm (since some problems have better performance after sorted order) is legion. As such, sorting is required knowledge for any computer scientist and engineer. There are quite a few sorting algorithms now in existence, and the most well known ones all perform the sorting operation in different ways.

Heapsort [Williams 1964] is a unique sorting algorithm in that it relies on the use of a heap data structure as a priority queue and sorts via the removal of minimum/maximum values to achieve sorted order. It was first introduced in 1964 by J.W.J. Williams. The algorithm is interesting considering that it is an example of a sorting method that can guarantee the $O(N\,log\,N)$ comparison-based sorting bound [1] via the clever use of a data structure that re-balances itself as elements are removed. This is true for all cases of Heapsort.

Heapsort's is often compared against Quicksort [Hoare 1961] and Mergesort [Sedgewick and Wayne 2011], both of which can achieve $O(N\,log\,N)$ sorting. Most often, the algorithm of choice is Quicksort since it can achieve better performance on average [Sedgewick and Wayne 2011]. However, Heapsort enjoys some advantages over these two:

— Heapsort is always guaranteed $O(N\,log\,N)$, no matter the input. Quicksort, even with three-way partitioning, can potentially perform in $O(N^2)$ time.
— Heapsort works better for arrays that are more partially ordered when compared to Quicksort and Mergesort. Indeed, one variant to Heapsort, Smoothsort [Dijkstra 1982], exploits this for further gain.
— Heapsort is performed in-place having a worst case $O(1)$ space complexity. This is superior to the worst case modified Quicksort space complexity ($O(log\,N)$) [Sedgewick 1978] and the $O(N)$ Mergesort space complexity. So, for large data sets, Heapsort is a better choice if memory becomes an issue in sorting.

For all of that, however, Heapsort does have disadvantages. As stated earlier, it is slower on average when sorting versus Quicksort and it is not a stable algorithm. Additionally, it is considered cache-inefficient [LaMarca and Ladner 1999] in trials against Mergesort and Quicksort.

The following sections will outline the terminology for understanding how Heapsort works. It is useful to review Heapsort and its subroutines since they will be repeatedly referenced when looking at the modified algorithms.

2. TERMINOLOGY

2.1. Complete Binary Tree

A complete binary tree is defined to be a type of binary tree where every level must be fully filled, exempting the last one. The last level may be either fully filled or partially filled, and if it is partially filled the partial filling must be from left to right. This can be seen in Diagram 1.

[1] This follows from the fact, given N objects, there are $N!$ ways to arrange those objects and it would take $log(N!)$ time to binary search them. Combine this with Stirling's approximation, $log(N!) = N\,log\,N - N + O(log\,N)$, it follows that the bound is given by $O(N\,log\,N)$.

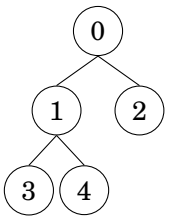

Diagram 1.

Note that in Diagram 1, the nodes values are equivalent to their index location in the array representation in a tree. For a given level i, the left child is located at $2i$ and the right child is at $2i + 1$. However, an incomplete binary tree would be seen in Diagram 2, where the tree is not filled as far left as it could be. Rather, each of the root's child nodes have a single leaf – this violates the definition of a complete binary tree.

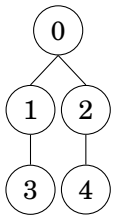

Diagram 2.

2.2. Heap

A heap is a type of complete binary tree that obeys what is caled the heap condition. The heap condition states that that the parent node is always larger than the child nodes[2]. For a heap to exist, the children themselves must also be heaps. This is illustrated in Diagram 3. The root is the largest value in the tree; its children are the largest values in the left and right subtrees. This heap property continues down to the leaf nodes, where both a leaf and a null node also satisfy the heap condition.

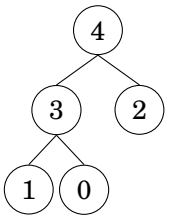

Diagram 3.

There are then three conditions that follow to check for a heap:

— The tree is a complete binary tree, with the last level filled as far left as possible.
— The root is the largest value in the heap.

[2]Note that this is referred to as a **maxheap**; an alternative heap condition would have the parent node smaller than its children – this is referred to as a **minheap**.

— The root's children are subheaps.

It is important to note that what has been displayed so far are binary heaps – every parent node has two children. Indeed, there are heap implementations that have more than two children attached to a rooted parent.

2.3. Leonardo Heap

A Leonardo Heap refers to a forest of heaps whose internal heaps all satisfy the heap condition and whose number of nodes follow the Leonardo numbers. The Leonardo numbers are a recursive sequence defined similarly to the Fibbonacci numbers:

$$L(0) = L(1) = 1 \tag{1}$$

$$L(n+2) = L(n) + L(n+1) + 1 \tag{2}$$

Following the recurrence relation, the first few sets of values are (1,1,3,5,9,15,25...). The data structure is therefore is a collection of disjoint binary heaps whose size M follow the Leonardo numbers. What that means is that: the first heap has one node ($L(0)$), the second heap has one node ($L(1)$), and since $L(2) = L(0) + L(1) = 3$, the third heap must have 3 nodes. And the heap sizing continues on like that; they do not connect to each other in any fashion. When inserting values into a Leonardo heap, no heap size that is not a Leonardo heap is tolerated; heaps can (and do) merge to satisfy these properties. The roots of the heaps need to be in ascending order from left to right [Schwartz 2011]; this means that the largest root is in the smallest heap and the smallest root is in the largest heap.

2.4. Two-Layer Heap

A two-layer heap is a deviational change to the heap data structure that is used in some variants of Heapsort and is analogous (in description, not in execution) to that of a red-black tree [Sedgewick and Wayne 2011]. The definition follows from Ultimate Heapsort [Katajainen 1998].

A two-layer heap is a heap of size N that is partitioned into two sets of elements: $green$ and red. All of the elements in the $green$ set are smaller than the elements in the red set and all of the elements obey the the heap property (children are smaller than parent). Additionally, the children of a red node must themselves be red. The labeling of the nodes comes from the idea that the $green$ nodes are selectable for extraction, whereas the red nodes must remain in the heap.

3. HEAPSORT

Given an array of values of length N, Heapsort will work by performing the following actions:

(1) Heapify the input data. This means taking the input sequences one by one, adding them to a growing heap, and restructuring the heap so that it maintains the heap condition. The end result will be a binary heap with the largest value at the root.
(2) Remove the maximum value (the root) by swapping it with the right-most leaf node.
(3) Re-heapify the array by sinking the new root with successive swaps until the maximum value in the heap is at the root. We are only look at a subarray of length $(N-1)$ inside the original array. By the ordering of the heap, the maximum value is now at the end of the heap and values $0, 1, ...(N-2)$ are the unsorted values.

(4) Continue remove the maximum and sink operations until there is only one value left in the heap. By definition, this last value must be the minimum and the array is now in sorted order.

Note that the removal of the maximum by swapping with the rightmost leaf node only violates the heap condition momentarily. After re-heapifying the subarray, we will again be left with a heap where remove the maximum can be performed until it is sorted.

Essentially, Heapsort performs sorting by continual heapify and remove-the-maximum operations. Since the swap operations on the various heap elements only uses at most one extra variable to store one of the values being swapped it becomes clear that this is why Heapsort has $O(1)$ space complexity – there is a dependence on the input size for the number of swaps, not for the space required for the swap operation.

3.1. Pseudocode

Code Segment 1: Heapsort

Input: An array $input$ of length N.
Output: The array in sorted order.

```
// Order the input data into a heap
heapify(input)

// Values from 0 to heapEnd are unsorted, which initially is all values
heapEnd = (N − 1)
while heapEnd > 0 do
    // Swap the root and the right-most leaf
    swap(heapEnd,0)
    // The heap size is reduced by one with the removal of the maximum
    heapEnd = (heapEnd − 1)
    // The heap needs to be restructured by sinking the child down to heapEnd and
       replacing it with the largest value in the heap,
    sink(input,0,heapEnd)
end
```

Code Segment 2: Heapify

Input: An array $input$ of length N.
Output: The array $input$ restructured as a heap.

```
// Find the parent of the last parent node in the array. Knowing that the last
element, (N − 1) is the rightmost leaf, the parent is found by working backwards
from 2i + 1
parent = ⌊(N − 2)/2⌋
while parent > −1 do
    // Sink all nodes to their rightful place to begin restructuring the heap.
    sink(input,parent,(N − 1))
    // Since the first assignment of start placed it above the leaf nodes, going
       backwards until start = 0 will sink all parent nodes until the heap condition
       is satisfied.
    start = (start − 1)
end
```

Code Segment 3: Sink

Input: An array $input$ of length N, a starting index $start$, and an ending index end.
Output: The array $input$ restructured with a node at $start$ sunk down as low as end.

```
parent = start
// While the current parent has at least one child
while parent * 2 + 1 <= end do
    // Assign the corresponding child
    child = 2 * parent + 1
    // Value to swap is the parent, store it temporarily
    temp = parent

    if data[temp] < data[child] then
        // If the swap value is less than the child value, then what needs to be sunk
           is the child; store it temporarily
        temp = child
    end
    if child + 1 <= end and data[swap] < data[child + 1] then
        // If the next child isn't past the ending point and if that child is larger
           than swap, then what needs to be sunk is the next child
        temp = child + 1
    end
    if swap != parent then
        // If after all that, the parent and the temp are not equal then swap them
        swap(parent,temp)
        parent = temp
    else
        return
    end
end
```

3.2. Analysis

To begin, let's examine the heapify operation. We need to perform N insertions into the binary heap. The sinking operation takes $O(log\, N)$ time since it is proportional to the height of the heap. This means that we are doing $O(N\, log\, N)$ to heapify the input data (N operations, each time performing the $log\, N$ sink).

Then, we perform removal of the maximum and restructuring of the heap. We perform this loop $(N-1)$ times, and internally that contains the sink method which can only be as bad as the height of the heap, so $log\, N$. Thus, the removal and restructure loop takes $O(N\, log\, N)$ as well.

The total complexity would be the sum of the heapify and the removal loop, which is still $O(N\, log\, N)$. This is the worst-case possible sorting complexity. Indeed, it has been stated and shown with much more rigor that the worst case is $2N\, log\, N$ [Sedgewick and Wayne 2011; Schaffer and Sedgewick 1993], so the top-level analysis here holds. Indeed, it could alternatively be shown that the recurrence relation for this algorithm correlates to an O($N\, log\, N$) case of the master theorem [3].

We can observe from the heap operations why the sort is unstable. The removal order and heap construction is based purely on size order. For two equally sized keys,

[3]The master theorem provides solution cases for recurrence relations of the form $F(n) = aF(n/b) + f(n)$, which many divide-and-conquer algorithms have. Indeed, Heapsort's sorting method is not intrinsically divide-and-conquer but the $heapify$ method is.

they will be inserted into the heap without regard to their original ordering. They are further removed from the heap without consideration to which was ordered first.

4. D-ARY HEAPSORT

One of the issues holding back Heapsort from beating Quicksort in typical runtime scenarios has to do with the performance increase Quicksort obtains from caching. Cache performances depends on two concepts:

(1) Temporal locality. This is the principle that if data has been accessed recently, it will be accessed again soon so it is in the computer's best interest to keep it in cache to avoid having to access higher memory and incur the associated miss penalties (which leads to a much longer wait time).
(2) Spatial locality. Data values that are close to each other in memory have some probability of being accessed soon. This, of course, makes sense; an array stores all of its value alongside itself in memory. If there is a loop iterating over the array, then it makes sense for the computer to store the entire array in the cache once it recognizes that values of the array are being accessed. It naturally follows that when one value is accessed, more accesses are to come.

Quicksort, as an algorithm, is what is considered to be cache-friendly [LaMarca and Ladner 1999]. Because of its nature as a divide-and-conquer algorithm, it enjoys both good temporal and spatial locality. It continually works on smaller and smaller subarrays, resulting in lower cache misses since those portions of the array are certain to be in the cache (the smaller it is, the greater benefit to spatial locality).

The reason why Heapsort doesn't obtain any performance increases from improved caching is because of how it is structured. The typical Heapsort implementation uses a binary heap, which means a larger height. The cache misses from accessing the heap for comparisons turns out to be proportional to the height of the heap [LaMarca and Ladner 1999]. An easy and obvious solution would be to reduce the height of the heap by increasing the number of children, thus leading to d-ary Heapsort.

4.1. Pseudocode

The implementation is effectively the same. The only operations that would change across Heapsort, Heapify, and Sink would be (assuming the levels are indexed by i):

$$Parent = \lceil (i-1)/d \rceil \tag{3}$$

$$Child = d(i-1) + 2 \tag{4}$$

$$Height = log_d\, N \tag{5}$$

Other than this, the code is too similar to bear repeating.

4.2. Analysis

The analysis of the algorithm would be the same as with Heapsort. The performance due to caching is not possible to quantify with mathematical rigor. However, it can be said that this new heap construction would lead to in an increase in the number of comparison operations. This should be obvious; since there are additional values to check at a given height before making the correct assumption for which child branch to pursue during the heapify/sink operations. So, theoretically, the d-ary Heapsort should have worse running time performance than the typical implementation of Heapsort if the analysis is based on the cost of comparisons and swaps [Islam and Kaykobad 2006]. But, however, this turns out not to be the case [LaMarca and Ladner 1999].

5. QUICKHEAPSORT

QuickHeapsort [Cantone and Cincotti 2002] is an algorithm that combines both Quicksort and Heapsort to make use of both of their respective advantages. QuickHeapsort can be shown to have an excellent average run time complexity, beating both normal Quicksort and Heapsort. As we observed in the typical Heapsort and can be observed from Code Segment 3, the sink operation requires two comparisons at each level (check the left or right child of the binary heap), and other variations (like a ternary heap) would require more than that in the worst case. Quicksort defeats Heapsort in the average case typically because of these extra comparisons (and caching, as addressed in d-ary Heapsort).

The motivation behind examining QuickHeapsort stems from the fact that it adopts wholesale many algorithm changes to Heapsort that were introduced by other Heapsort variants (Bottom-Up Heapsort [Wegener 1993], External Heapsort [Wegner and Teuhola 1989], Ultimate Heapsort [Katajainen 1998]). Many of the algorithms it draws inspiration exist only as theoretical curosities [Diekert and Weiß 2013], whereas QuickHeapsort is an algorithm that is practically useful and can achieve equal or greater performance to Quicksort.This makes it an optimal choice for a survey.

This algorithm follows these steps:

(1) Upon receiving an array of size N, randomly choose a pivot element p.
(2) Partition the array into three subarrays according to p just as in Quicksort. This means that the left subarray consists of values less than p, the middle subarray contains all values equal to p, and the right subarray contains all values greater than p.
(3) Now, construct a two-layer heap out of the partitioned array from Step 2.
    (a) Construct the subset of $green$ nodes out of the the smallest sized subarray from Step 2. Whichever one contains less elements, the left or right, must be the green nodes.[4]
    (b) If the left half of the subarray is selected to be $green$, then the two layer heap must be a maxheap. Else, it is a minheap.
(4) Now, the left half of the array exists as the two-layer heap and the right half of the array will be used as the storage component. Assuming that we have a maxheap, at the first iteration the root is removed and swapped with the last value (this would be index $(N-1)$) of the right subarray.
(5) The new root of the heap would now be a blank or null value. The largest child in the heap rises to replace this hole, and it gets swapped down until it reaches a leaf. At this point, the null value is updated with the swapped value from the right subarray that the extracted root replaced. The value is labeled red.
(6) The pointer in the right subarray decrements to point from $(N-1)$ to $(N-2)$, indicating the next value to be swapped out with a $green$ value from the heap.
(7) This continues until the array reaches sorted order. After all of the $green$ elements are moved out, the pivot element is placed at the final position and any leftover elements (there will be leftovers from the larger subarray) are sorted recursively.

An improvement of QuickHeapsort is given by [Diekert and Weiß 2013], where during the heap construction phase they ensure that the left child is always less than the right child in the heap. Every time the null value needs to be sunk down, they save a compare of finding the largest child (Step 5) over all the recursive calls by investing time earlier to structure the heap in such a way that additional compares are not needed.

[4] Of course, the smallest subarray could be the middle subarray of values equal to the pivot. However, these values are already in their final sorted position! So it does not behoove us to select them for sorting.

5.1. Pseudocode

Code Segment 4: QuickHeapsort

Input: An array $input$ of length N with starting index i.
Output: The array $input$ in sorted order.

```
if N > 1 then
    // ChoosePivot randomly selects a value from input to be the pivot element. It
       returns the value of this pivot element as p.
    p = ChoosePivot(input)
    // Partition takes an array input and performs standard Quicksort partitioning
       using a pivot element, p. It return the index k in the array, which is the
       final index of the pivot p.
    k = Partition(input,p)
    // If the pivot element index is less than half of the array, the smallest
       subarray is on the left so we must form a maxheap there
    if k <= N/2 then
        // Form the maxheap up to (k − 1)
        TwoLayerMaxHeap(input,(k − 1))
        // Swap out the root
        swap(input,k,(N − k + 1))
        // Recursively sort
        QuickHeapsort(input,0,(N − k))
    else
        // Since the larger subarray is on the left, form a minheap from the right
        TwoLayerMinHeap(input,(N − k))
        swap(input,k,(N − k + 1))
        // Recursively sort
        QuickHeapsort(input,(N − k + 2),N)
    end
end
```

Code Segment 5: TwoLayerMaxHeap

Input: An array $input$ of length N with ending index end.
Output: The array $input$, with components up to end restructured as a maxheap.

```
// Call a modified heapify function that works up until index end
heapify(input,end)
// Iterate and perform the sinking command until discovery of a leaf node via
   SpecialLeaf
for i = 1 to end do
    temp = input[N-i]
    input[N-i]=input[0]
    j = SpecialLeaf(input,end)
    input[j] = temp
end
```

Code Segment 6: SpecialLeaf

Input: An array $input$ of length N with ending index end.
Output: The index i of the location of the leaf.

```
// Perform comparisons on the childs of a parent node, looking for the proper child
   location
i = 1
while 2i <= end do
    if 2i + 1 <= m and input[2i + 1] ¿ input[2i] then
        input[i] = input[2i + 1]
        i = 2i + 1
    else
        input[i] = input[2i]
        i = 2i
    end
end
return i
```

There isn't any need to repeat the structure for the case of a minheap since it is effectively the same as the above cases.

5.2. Analysis

It should be obvious that the performance is still $O(N\,log\,N)$ in time complexity (it is a mix of two algorithms, both of which obey the sorting bound) and $O(1)$ extra space. This is a battle that must be won through the leading coefficients normally surpressed by Big-O notation. QuickHeapsort, with the modifications from [Diekert and Weiß 2013], makes improvements over standard Heapsort. Though not explicitly stated in algorithmic terms (rather it is done mathematically) by [Cantone and Cincotti 2002; Diekert and Weiß 2013], we can deduce the reasons why. Heapsort's ineffeciencies in comparison are derived from both the size of the heap and the face that items need to be pulled from the bottom, swapped with the top, and then sank down via compares and swaps. It reduces the number of comparisons required to get an array in sorted order in a few ways:

— The partitioning of the array that is borrowed from Quicksort. This only requires $(N - 1)$ comparisons at the most to structure the array, compared to the typical beginning construction of the heap in Heapsort.
— Prior to heap construction, we'll have a middle subarray of size at least equal to one already in sorted order.
— The heap construction operates recursively on the smallest subarray of the original array, which means already that we are dealing with a heap less than the size of the input.
— The modification to the heapify from [Diekert and Weiß 2013], where the left and right child are ordered by size, can reduce the comparisons needed when operating recursively. Indeed, no comparisons are needed now; the right child is placed directly into the parent.

The algorithm really is quite fascinating in its blend of properties with Heapsort and Quicksort. Formally, the cost of QuickHeapsort in the worst case is shown to be [Cantone and Cincotti 2002; Diekert and Weiß 2013]:

$$f(N) = N\,log\,N - 0.03N + O(N) \tag{6}$$

6. SMOOTHSORT

Smoothsort [Dijkstra 1982] is another algorithm that inherits from Heapsort, with the additional caveat that it is adaptive – if the input is in some kind of sorted order, then it is possible to achieve $O(N)$ time complexity. The power of Smoothsort comes from the interesting type of heap Dijkstra constructed – heaps whose connections follow the Leonardo numbers.

The algorithm conditions are as follows:

(1) Heapify the array according to the Leonardo distribution. This means partitioning the array into a collection of heaps via insertion operations.
(2) There are a few rules to obey for insertion into the string of heaps:
   (a) Check to see if the resulting heap (after insertion) obeys a Leonardo number, $L(x)$, and that the string of heaps are sorted in descending order of size (the smallest heap is on the right side of the array, largest heap on the left side).
   (b) The roots of each Leonardo heap must be in ascending order from left to right. This was mentioned earlier; the smallest heap must have the largest root and vice versa.
   (c) Each Leonardo heap after insertion must obey the max-heap property.
(3) During insertion, check to see if the two smallest Leonardo are of size $L(x)$ and $L(x+1)$. After insertion of an additional value, the two smallest heaps should be merged together into a single heap of size $L(x+2)$.
(4) To preserve the heap root ordering after insertion of a new value, perform insertion sort [Sedgewick and Wayne 2011] on the roots of the heaps.
   (a) If the heap root to the left of the rightmost heap is larger than the right root and the right root is larger than the children of the left heap, perform the swap.
   (b) Call heapify on the rightmost heap if the heap property needs to be satisfied. Note that the left heap that was swapped does not need heapification; it satisfies the heap condition.

6.1. Pseudocode

Pseudocode for this is quite taxing given the complication of cases. An array of prestored Leonardo numbers is required as well as many pointers to keep track of the different heaps, and even [Dijkstra 1982] gets long in the tooth in the explanation of the pseudocode. It is better to keep the pseudocode at a high level since it mostly repeats concepts outlined in the basic Heapsort, with dashes taken from insertion sort.

Code Segment 7: Smoothsort

Input: An array $input$ of length N.
Output: The array $input$ in sorted order.

```
// Construct a string of Leonardo heaps from input
LeonardoHeapify(input)
// Identify the last index in the array as the back
back = (N − 1)
heapSize = N
while heapSize > 0 do
    // Find the index of the maximum value in the heap
    maximum = HeapMaximum(input)
    // Swap the maximum value with the back of the heap
    swap(maximum,back)
    back = back − 1
    heapSize = heapSize − 1
end
```

6.2. Analysis

Since this algorithm is quite unique with its choice of heap, it behooves us to examine it at its myriad stages. Firstly, creation of the initial Leonardo trees is constant (two immediate inserts to form $L(1)$, $L(2)$). Insertion of any value after that would become proportional to the height of the newly constructed heap ($L(3)$), so that would be $O(log\, N)$ in the worst case (as seen from other standard insertions in heaps or trees in general). Element removal is the same here.

What is interesting here is the insertion of maximum value pairs into the data structure. Maximums are insertion sorted by the method outlined earlier in constant time; the heapify operation will always take the same amount of time but the time it takes to swap this element to its final place can be done in $O(1)$; it does not depend on input size. This is central to how the algorithm can adapt to $O(N)$ [Dijkstra 1982; Schwartz 2011].

Over the long haul, this will guarantee $O(N\, log\, N)$ time complexity just like Heapsort does. The way Smoothsort obtains $O(1)$ extra space complexity is by using a bit vector to identify the size of each individual heap (otherwise, we would need O(N) extra space to store the size of each heap in the forest as an integer). A bit vector works here because the Leonardo numbers are unique and identifiable with a given bit pattern.

The use of Leonardo heaps has to do with a central result iterated by [Dijkstra 1982]. Put succintly, any positive integer n can be written as the sum of $O(log\, n)$ distinct Leonardo numbers. If the input is already structured in such a way that it is already sorted, the removal operation takes constant time. As such, the sorting performs in $O(N)$ time on an array of N sorted integers.

It is quite an interesting sorting method based on a strange selection of heaps to obtain linear time sorting with a priority queue on near-sorted inputs. The work is quite involved to implement it and more to understand it, but it contains inside of it many lessons to teach about how the right selection of a data structure paired with a corresponding algorithm can work together to increase operation performance.

7. GRAPHICAL ANALYSIS

Finally, as a point of order, it is useful to measure the running time of the algorithms for large inputs and see how they perform. Pictured below is the result.

It is evident that QuickHeapsort is leading the pack, and quite well ahead of other implementations of Heapsort. This was expected due to the reasons outlined in the QuickHeapsort section (effectively, reduction of the number of compares, two-layer heap operating on smaller subarrays, et cetera). It is interesting that it wins by such a large stretch, but this happens when Heapsort competes against standard Quicksort in the first place[5]. The simple implementation of Heapsort is the least effective, with all other implementations outperforming it. What is notable is that the best alternative implementation of the d-ary Heapsort is the 4-way Heapsort. This is very system dependent, but it turns out in general that having a larger number of children for the heap does not automatically translate to better performance [LaMarca and Ladner 1999]; indeed, 4-ary Heapsort is often the recommended choice.

Smoothsort rounds out by being the third place winner. This is interesting; it is entirely possible that partial orderings during the sort improves performance as the sorting continues on. Since Smoothsort is a sorting algorithm that adapts well to order, let's examine performance for purely sorted arrays.

Well, it appears that the implementation coded up for QuickHeapsort suffers from the quadratic runtime that can happen for equal keys and sorted arrays in Quicksort.

[5]Quicksort and QuickHeapsort seem to have similar performance, but proper selection of a pivot can apparently tip the favor towards QuickHeapsort [Diekert and Weiß 2013].

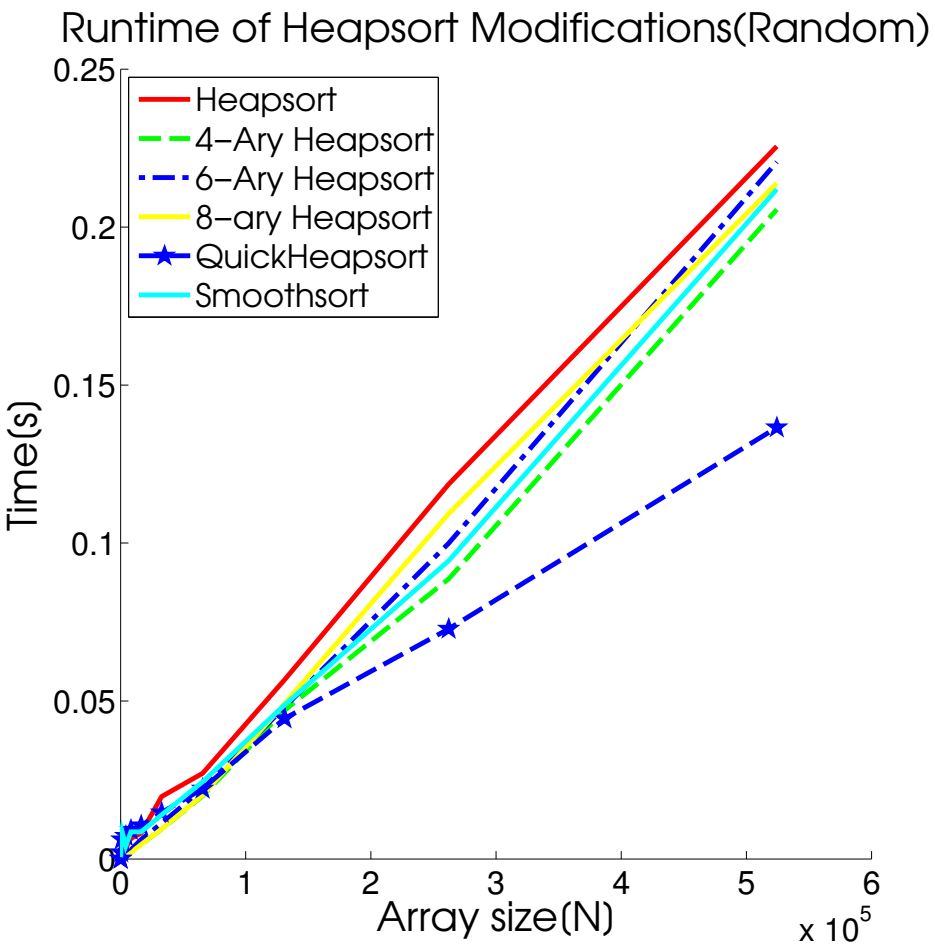

Fig. 1. Algorithm running times for inputs of $N = 512$ up to $N = 52488$.

QuickHeapsort, like Quicksort, can be modified to avoid the $O(N^2)$ running time so it is not an issue; this is a weakness in this implementation of the algorithm by the author of this paper, and not necessarily the algorithm. But it does point to an old lesson that the input highly affects how the algorithm will perform.

Case in point: Smoothsort. Smoothsort outperformed all the other arrays in this instance, as [Dijkstra 1982] postulated it would. The algorithm is very difficult to program, but the payoffs for sorted (or nearly sorted) can't be ignored. It does not appear to be exactly linear, but has linearithmic trends. There is some logarithmic operations on the heaps when restructuring (since we insert in order into heaps), but it's likely that dies out as the array size gets larger and converges to $O(N)$. All of the other algorithms (exempting QuickHeapsort) obey the same linearithmic properties as in the first figure. An interesting note is that 8-ary Heapsort outperformed the d-ary Heapsort algorithms, but they are so close that it is likely if the input got larger 4-ary Heapsort would again win out.

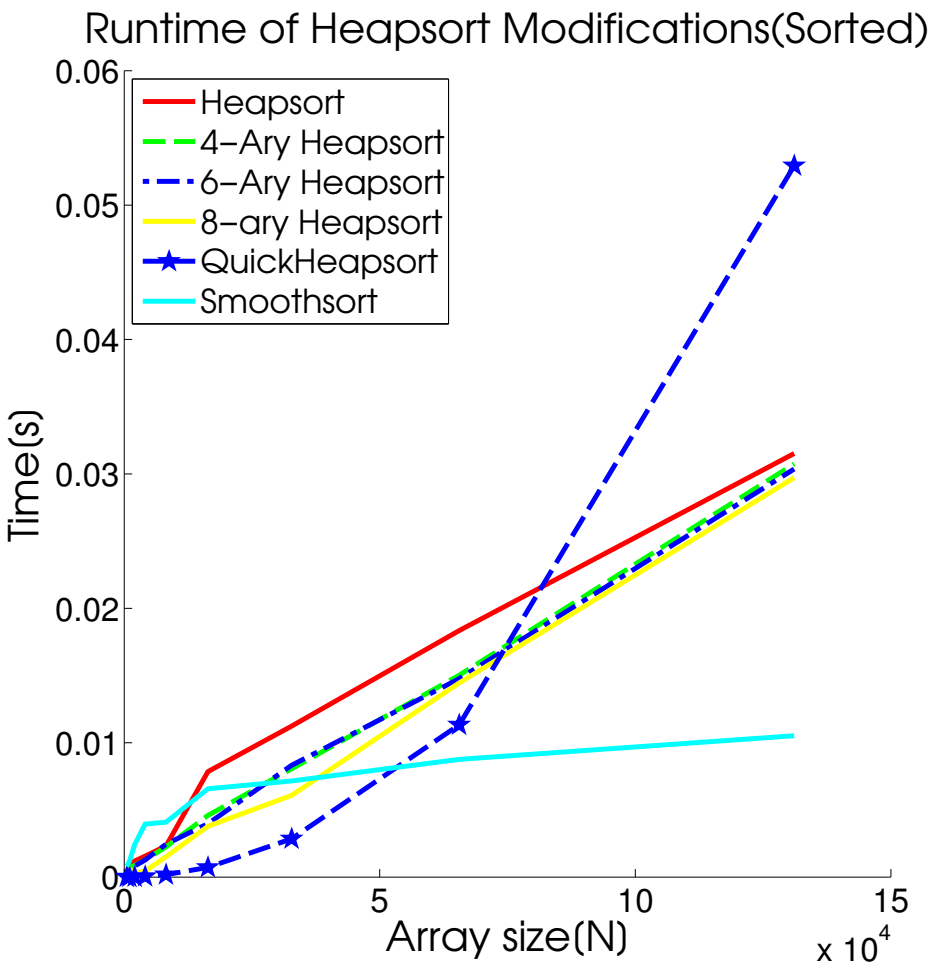

Fig. 2. Algorithm running times for inputs of $N = 512$ up to $N = 131072$.

8. CONCLUSIONS

In this paper, we looked at several modifications to Heapsort that can help it overcome challenges from other sorting algorithms. Of course, there is no real king of the hill; sorting is a task that is situationally-dependent. There is no general purpose sorting algorithm that is optimal for all input types (that would be the "sorting holy grail" [Sedgewick and Wayne 2011]), but there are interesting algorithmic changes that can be made to any sorting algorithm to increase performance or address weaknesses (as seen herein).

Of curious note here is the lack of mention of how to fix Heapsort's stability problem. Indeed, the very nature of how the heap becomes structured removes any aspect of stability. Certainly, one could keep track of the number of equal elements and impose external order on them, but that increases the space complexity to $O(N)$, which is decidedly less than Heapsort's initial $O(1)$. Modifying Heapsort to be stable is not difficult in and of itself, but preserving $O(N \log N)$ time complexity and $O(1)$ space complexity while being in-place and stable seems roughly impossible.

It lends itself to a thought experiment. Indeed, any method will require at least $\Omega(N)$ in order to track the equal pairs. One way to at least identify equal pairs could be by doing a pass over the array initially and counting the number of equal pairs (without regard to the magnitude of those pairs) and encode the value in a constant whose binary representation contains '0' where there is no pair and '1' where there is. Where to go from there, however, is unclear. Since the heap is typically structured inside of the array, it would need to be partitioned such that the equal pairs are kept on one side and the heap on the other (and still preserve the relative order). Likely, what needs to happen is that equal pairs have to be inserted into a pre-sorted array in groups. It is a difficult problem to overcome and hasn't seen any solution at the time of this writing.

Finally, it can be seen that the basic Heapsort algorithm is not an optimal implementation. The modifications shown herein can be used to improve its performance, especially QuickHeapSort. So, depending on the input, there are modifications to Heapsort that can be used to help make it optimal.

ACKNOWLEDGMENT

Many acknowledgements go to the authors of the various algorithms presented herein. Without their hard work, this survey would not be possible. Liberal use of their knowledge and effort has been used, with a dash of rewriting or further explanation to try to open the algorithms to a broader audience.

REFERENCES

Domenico Cantone and Gianluca Cincotti. 2002. QuickHeapsort, an efficient mix of classical sorting algorithms. *Theor. Comput. Sci.* 285, 1 (2002), 25–42.

Volker Diekert and Armin Weiß. 2013. QuickHeapsort: Modifications and Improved Analysis. In *CSR*. 24–35.

Edsger W. Dijkstra. 1982. Smoothsort, an Alternative for Sorting In Situ. *Sci. Comput. Program.* 1, 3 (1982), 223–233.

C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (1961), 321.

Tarique Mesbaul Islam and M. Kaykobad. 2006. Worst-case analysis of generalized heapsort algorithm revisited. *Int. J. Comput. Math.* 83, 1 (2006), 59–67.

Jyrki Katajainen. 1998. The Ultimate Heapsort. In *CATS*. 87–96.

Anthony LaMarca and Richard E. Ladner. 1999. The Influence of Caches on the Performance of Sorting. *J. Algorithms* 31, 1 (1999), 66–104.

Russel Schaffer and Robert Sedgewick. 1993. The Analysis of Heapsort. *J. Algorithms* 15, 1 (1993), 76–100.

Keith Schwartz. 2011. Smoothsort Demystified. (2011).

Robert Sedgewick. 1978. Implementing Quicksort Programs. *Commun. ACM* 21, 10 (1978), 847–857.

Robert Sedgewick and Kevin Wayne. 2011. *Algorithms, 4th Edition*. Addison-Wesley. I–XII, 1–955 pages.

Ingo Wegener. 1993. BOTTOM-UP-HEAPSORT, a New Variant of HEAPSORT, Beating, on an Average, QUICKSORT (if n is not Very Small). *Theor. Comput. Sci.* 118, 1 (1993), 81–98. http://dblp.uni-trier.de/db/journals/tcs/tcs118.html#Wegener93

Lutz Michael Wegner and Jukka Teuhola. 1989. The External Heapsort. *IEEE Trans. Software Eng.* 15, 7 (1989), 917–925.

J. W. J. Williams. 1964. Algorithm 232: Heapsort. *Commun. ACM* 7, 6 (1964), 347348.