MAS2602 Sorting Project

Joe Marks
B8063804

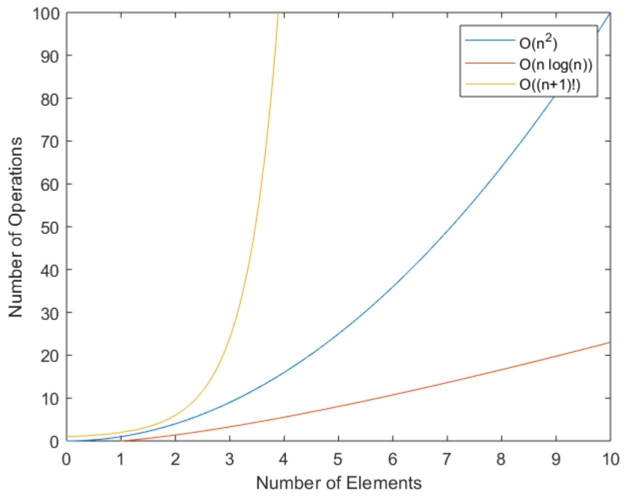
Tuesday 7th January

0 Background

Heapsort is a sorting algorithm first used in 1964 by John William Joseph Williams. The number of operations for heapsort to run is described normally with 'Big-O Notation', $O(f(n))$, which relates the size of the input to the time until completion.

Although Big-O Notation provides an asymptotic upper bound, it is not always reflective of an algorithm's performance - for example, although quick sort is placed in the same run-time category as several different algorithms, in the real world it is often faster.

Sorter	**Average Time**
Heap Sort	$O(n\log(n))$
Merge Sort	$O(n\log(n))$
Quick Sort	$O(n\log(n))$
Bubble Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Permutation Sort	$O((n+1)!)$



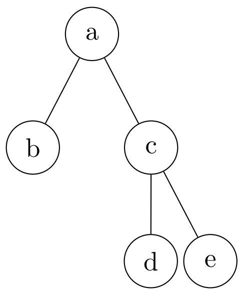
1 The Heapsort Algorithm

1.1 Binary Trees

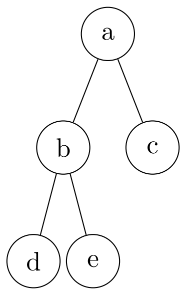
Trees are a form of structure with a hierarchy, where every entry is called a node. In a binary tree, nodes have the following properties:

- Nodes with others below them are called parents, and the ones below them called their children.
- The topmost node is called the root node.
- Each node may have a maximum of two children.
- Nodes with no children are called leaves.

These are examples of binary trees:



(a) Binary Tree

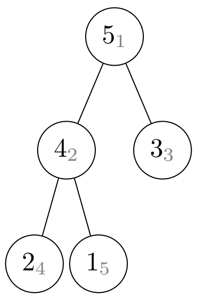


(b) Complete Binary Tree

For the purposes of the heapsort algorithm, we are interested in a variant of the binary tree called a complete binary tree, where every level other than the last must be filled (i.e. no gaps) and all the leaves are on the leftmost of the tree.

1.2 Heaps

A binary heap is a structure based on a complete binary tree but with an ordering such that each parent node is either greater or smaller than its children. In the case that the parent's node is greater, the structure is called a max heap, for example:



In the above diagram, the grey subscript values represent the index of the node, i, and the other numbers represent the value of the node.

One useful observation we can make from the underlying complete binary tree structure is that this binary heap can also be represented as a row vector $\mathbf{x} = (5, 4, 3, 2, 1)$ when we define the nodes as the i-th entries of $\mathbf{x}$. We can also calculate the following given some starting node i:

I. A left child is given by x_{2i},

II. a right child is given by x_{2i+1},

III. a parent is given by $x_{\lfloor i/2 \rfloor}$

Where x_i represents the i-th entry of $\mathbf{x}$.

To apply this to our example, if we consider the node $i = 2$, we can see that $x_2 = 4$ in agreement with our diagram. If we need to calculate the parent of this node, we calculate, using III: $x_{\lfloor i/2 \rfloor} = x_{\lfloor 2/2 \rfloor} = x_1 = 5$. We can similarly find the children using I and II.

This is useful in the sifting process for making direct comparisons between the sizes of parents and children nodes.

1.3 The Process

The process used can be outlined as followed:

1. Set input vector as new binary heap.

   See example in (1.2) for equivalence of row vector and binary heap.

   In this implementation this step is implicit as a part of the 'sifting' process, however we can see our observations being used.

   In line 5, observation III is used by setting $i =$ (last value of $\mathbf{x}$) in order to find the last parent node. Additionally, observations I & II are used in lines 28 & 32 respectively to distinguish between left and right children.

2. Create a max heap.

   This is accomplished (see line 9) using the same technique in the next step.

3. Swap the first node with the last node, remove the last node from the heap.

   This is the most complicated part of the process, which involves 'sifting' using `sift` (line 23-44).

   The algorithm begins with the last parent, and compares whether its children are larger (see line 36 if statement). If either is, the parent and children node are swapped (line 38/39), and the same operation is carried out for the parent above.

   After a few iterations, the largest value is moved to the position of the root node. Once there, it is swapped with the last node of the heap, and then the following search has its consideration size reduced by one (line 18), meaning the largest value has been fixed at the end of the heap.

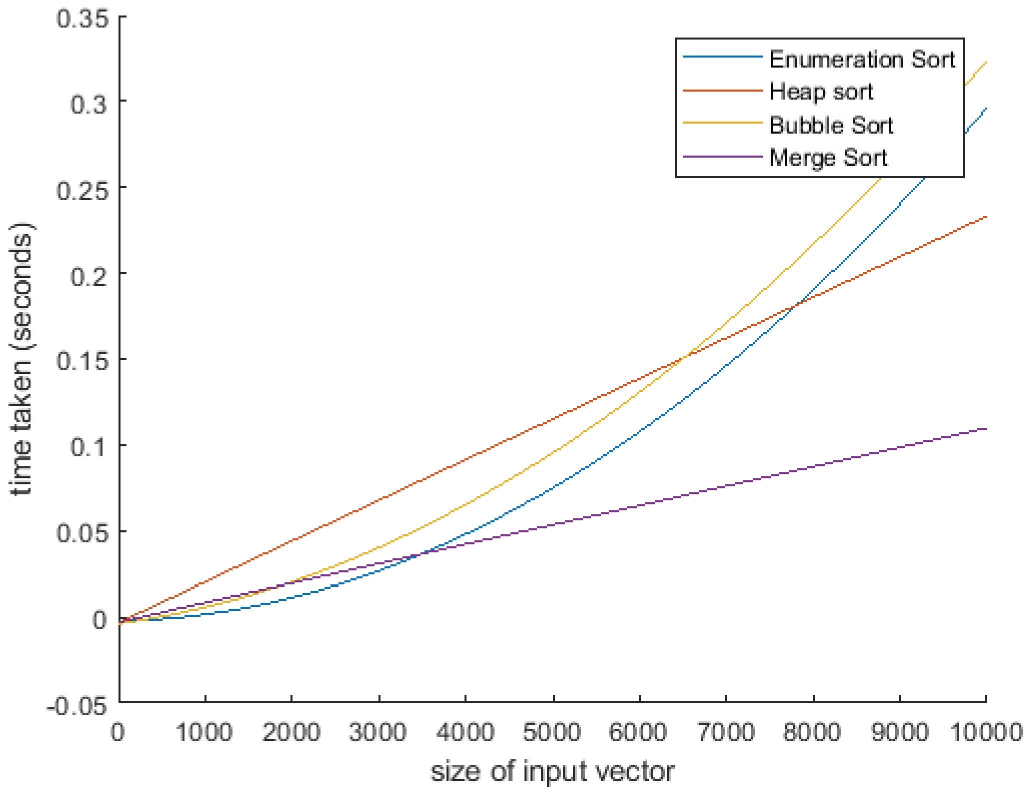
4. Find new root node, return to step 2.

   As this process repeats, the sorted vector is built from right-to-left, largest-to-smallest, and after each sifting cycle the number of fixed entries increments.

   Eventually the heapsize reaches a size of only two nodes, in which case the last comparison is made (while loop at line 14 stops afterwards) and the resulting vector is now sorted in ascending order.

2 Analysis

A comparison of different sorting algorithms is plotted below, where for each algorithm 100 random input vectors were tested ranging in size up to 10,000 elements and the time taken recorded. The data was then fit to approximately compare between sorting algorithms.



We can see the quadratic behaviour of both bubble sort and enumeration sort, where for large input vectors heap sort and merge sort are both faster.

We should also note that although merge sort and heap sort are in the same Big-O category, merge sort appears consistently faster than heap sort.

Sources

[1] Cormen, T., Leiserson C., Rivest R. Stein C., (2001) *Introduction to Algorithms*. 2nd edn. MIT Press.

[2] Jain, S. *Analysis of Different Sorting Techniques* and *Binary Tree Data Structure*. Available at: www.geeksforgeeks.org

[3] Allain, A. *Sorting Algorithm Comparison*. Available at: www.cprogramming.com

[4] *Heapsort - Pseudocode*, www.codecodex.com

[5] *Heapsort - Pseudocode*, www.wikipedia.org

[6] *Heap Sort Algorithm*, www.programiz.com

Code Appendix: heapsort.m

```
function x = heapsort(x)
    heapSize = length(x);

    % Set the parent of the last node as first for sifting
    startNode = floor(heapSize/2);

    while startNode >= 1
        % Build max heap, reduce parent node index until last parent
        x = sift(x, startNode, heapSize);
        startNode = startNode - 1;
    end

    % While loop contininues until last two nodes ordered
    while heapSize >= 2
        % Swap first and last entries of heap
        x([heapSize 1]) = x([1 heapSize]);
        % Reduce heap size
        heapSize = heapSize - 1;
        x = sift(x,1,heapSize);
    end
end

function x = sift(x,currentParent,lastNode)
    % Test if child node needs to be moved up
    % While loop continues only if left child node is in heap
    while (currentParent * 2) <= lastNode
        % Assume left child is larger than right
        childNode = currentParent * 2;

        % Check if left child is larger than right
        if (childNode + 1 <= lastNode) && (x(childNode) < x(childNode+1))
            childNode = childNode + 1;
        end

        % Find whether largest child is larger than parent
        if x(currentParent) < x(childNode)
            % Replace parent with child
            x([currentParent childNode]) = x([childNode currentParent]);
            currentParent = childNode;
        else
            return % Stop and return to heapsort function if parent > child
        end
    end
end
```