# Lab Assignment #4

Heap Sort

Full Name: Shreyas Srinivasa
BlazerId: SSRINIVA

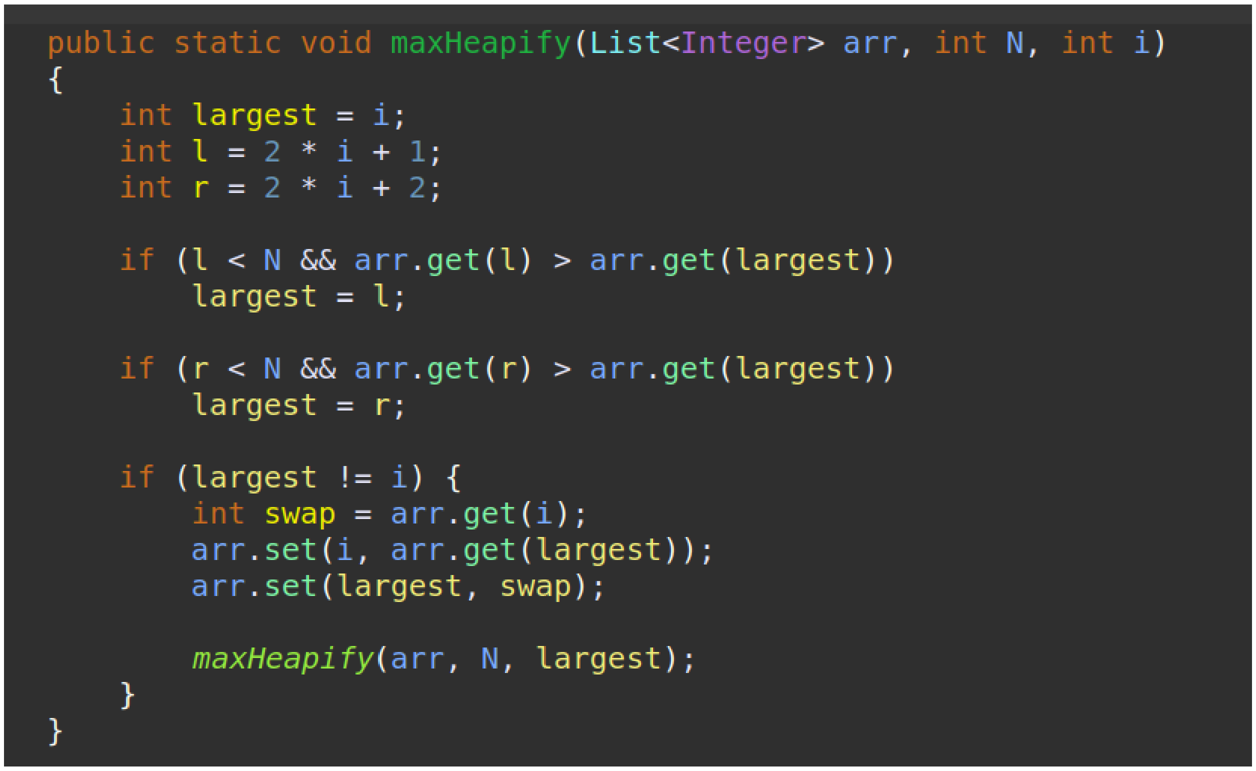
CS 303 Algorithms and Data Structures

Spring 2023

## 1. Problem Specification

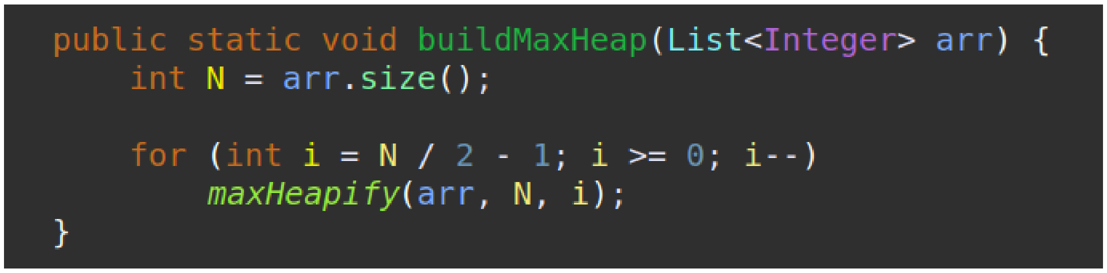
The purpose of this project was to implement an algorithm for heap sort and measure its execution time with a number of datasets of progressively increasing size. This execution time was used to study the connection between real time performance and theoretical complexity analyses. The data collected for heap sort was further compared to the data collected for insertion sort and merge sort and all were studied together to determine which algorithm performs best.

## 2. Program Design



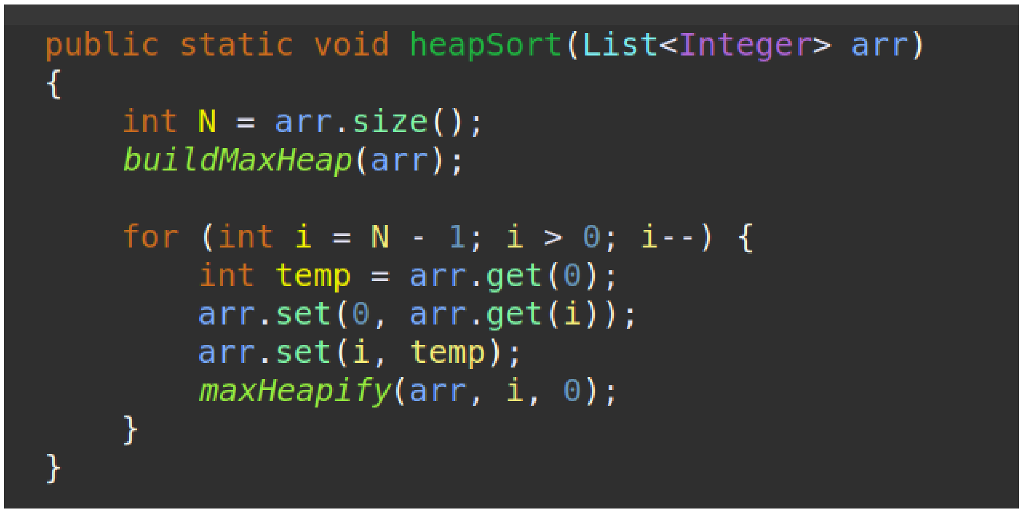
```java
public static void maxHeapify(List<Integer> arr, int N, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < N && arr.get(l) > arr.get(largest))
        largest = l;

    if (r < N && arr.get(r) > arr.get(largest))
        largest = r;

    if (largest != i) {
        int swap = arr.get(i);
        arr.set(i, arr.get(largest));
        arr.set(largest, swap);

        maxHeapify(arr, N, largest);
    }
}
```

Function to perform heapify operation was implemented in Java code



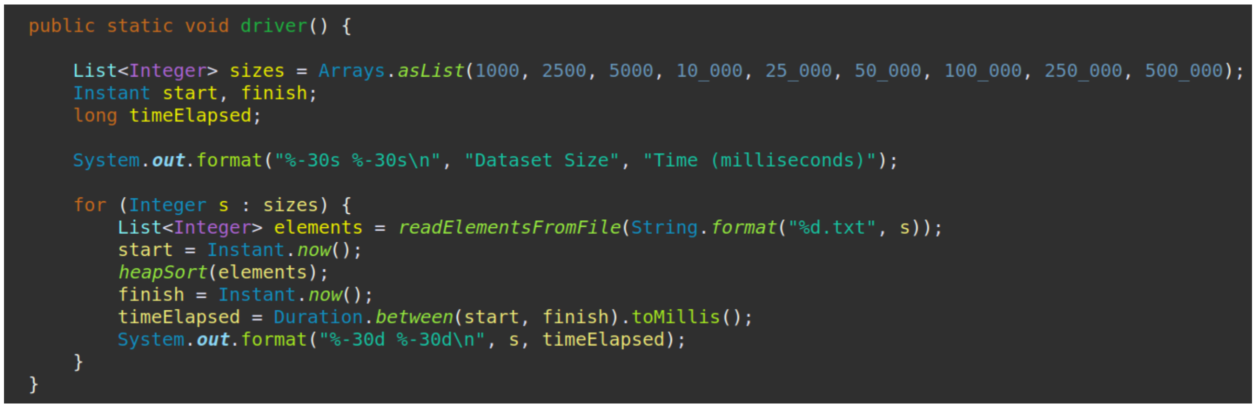
```java
public static void buildMaxHeap(List<Integer> arr) {
    int N = arr.size();

    for (int i = N / 2 - 1; i >= 0; i--)
        maxHeapify(arr, N, i);
}
```

This function will build us a max heap data structure



```java
public static void heapSort(List<Integer> arr)
{
    int N = arr.size();
    buildMaxHeap(arr);

    for (int i = N - 1; i > 0; i--) {
        int temp = arr.get(0);
        arr.set(0, arr.get(i));
        arr.set(i, temp);
        maxHeapify(arr, i, 0);
    }
}
```

This function will swap the last element from the heap with the root and perform heapify operation on the new root essentially performing heap sort on the entire array by iterating through all nodes

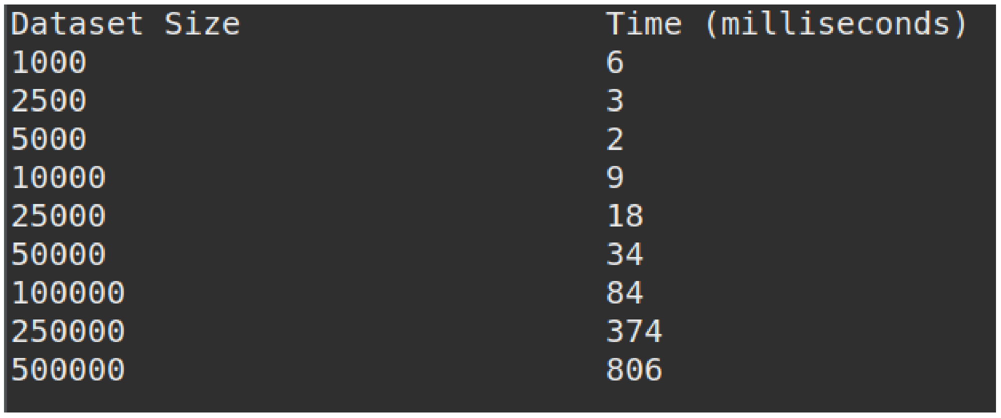


```java
public static void driver() {

    List<Integer> sizes = Arrays.asList(1000, 2500, 5000, 10_000, 25_000, 50_000, 100_000, 250_000, 500_000);
    Instant start, finish;
    long timeElapsed;

    System.out.format("%-30s %-30s\n", "Dataset Size", "Time (milliseconds)");

    for (Integer s : sizes) {
        List<Integer> elements = readElementsFromFile(String.format("%d.txt", s));
        start = Instant.now();
        heapSort(elements);
        finish = Instant.now();
        timeElapsed = Duration.between(start, finish).toMillis();
        System.out.format("%-30d %-30d\n", s, timeElapsed);
    }
}
```

The driver function for measuring the execution times of merge sort along with increasing input size was also implemented.

The driver function which provides our second set of results with the modified merge sort algorithm

## 3. **Output**



```
Dataset Size                  Time (milliseconds)
1000                          6
2500                          3
5000                          2
10000                         9
25000                         18
50000                         34
100000                        84
250000                        374
500000                        806
```

**Output 1.** Heap Sort Results

| Dataset Size | Time (milliseconds) |
|---|---|
| 1000 | 13 |
| 2500 | 3 |
| 5000 | 8 |
| 10000 | 5 |
| 25000 | 8 |
| 50000 | 18 |
| 100000 | 44 |
| 250000 | 174 |
| 500000 | 379 |
| 1000000 | 838 |

**Table 1.** Merge Sort Results

| Dataset Size | Time (milliseconds) |
|---|---|
| 1000 | 20 |
| 2500 | 17 |
| 5000 | 58 |
| 10000 | 185 |
| 25000 | 1435 |
| 50000 | 6813 |
| 100000 | 25776 |
| 250000 | 343166 |
| 500000 | 2426168 |
| 1000000 | 10414209 |

**Table 2.** Insertion Sort Results

| Dataset Size | Time (milliseconds) |
|---|---|
| 1000 | 6 |
| 2500 | 3 |
| 5000 | 2 |
| 10000 | 9 |
| 25000 | 18 |

| | |
|---|---|
| 50000 | 34 |
| 100000 | 84 |
| 250000 | 374 |
| 500000 | 806 |

**Table 3.** Heap Sort Results

## 4. Analysis and Conclusions

**InterviewCake Conclusion:-**

For the heapify step, we're examining every item in the tree and moving it downwards until it's larger than its children. Since our tree height is *O(lg(n))*, we could do up to *O(lg(n))* moves. Across all n nodes, that's an overall time complexity of *O(n*lg(n))*.

After transforming the tree into a heap, we remove all n elements from it—one item at a time. Removing from a heap takes *O(lg(n))* time, since we have to move a new value to the root of the heap and bubble down. Doing n remove operations will be *O(n*lg(n))* time.

Is this analysis too pessimistic? Each time we remove an item from the heap it gets smaller, so won't later removes be cheaper than earlier ones?

Putting these steps together, we're at **O(n*lg(n)) time in the worst case (and on average)**.

But what happens if all the items in the input are the same?

Every time we remove an element from the tree root, the item replacing it won't have to bubble down at all. In that case, each remove takes *O(1)* time, and doing n remove operations takes *O(n)*.

So the **best case time complexity is O(n)**. This is the runtime when everything in the input is identical.

Since we cleverly reused available space at the end of the input array to store the item we removed, we only need O(1) space overall for heapsort.

**GeeksForGeeks Advantages and Disadvantages**

**Advantages of Heap Sort:**

- *Efficiency* – The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.

- *Memory Usage* – Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- *Simplicity* – It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

**Disadvantages of Heap Sort:**

- *Costly*: Heap sort is costly.
- *Unstable*: Heat sort is unstable. It might rearrange the relative order.
- *Efficient*: Heap Sort are not very efficient when working with highly complex data.

**As we already know from previous labs:-**
The Time Complexity of an algorithm/code is **not** equal to the actual time required to execute a particular code, but the number of times a statement executes.

The **actual time required to execute code is machine-dependent**.

Instead of measuring actual time required in executing each statement in the code,Time Complexity considers how many times each statement executes.

**Performance compared to Merge Sort and Insertion Sort**

- On the theoretical front, heap sort time complexity matches merge sort and is clearly better than insertion sort. So we expect heap sort to definitely perform better than insertion sort. On seeing the machine specific execution times, we clearly observe that heap sort is performs better than insertion sort for all datasets and this ratifies the theoretical analysis performed in previous labs. In the comparison with merge sort, we see that heap sort initially performs better than merge sort with small datasets but quickly deteriorates in performance with larger datasets, with merge sort emerging to be the optimal choice to sort these datasets on this machine. This also indicates that merge sort is more practical and versatile than the other algorithms that have been studied till now.

## 5. References

- https://www.geeksforgeeks.org/heap-sort/
- https://www.interviewcake.com/concept/java/heapsort
- https://www.happycoders.eu/algorithms/heapsort/
- https://www.codesdope.com/course/algorithms-heapsort/