CS 430 - HW 3

Project Management Document

ISSUED BY

Team Glazed Donuts

Members

Gladys Toledo-Rodriguez, #59 A20419684	Grace Arnold, #6 A20415197

Tasks:

Gladys will work on the Quick Sort algorithm and Grace will code the Heap Sort algorithm. Both will work on the GUI. You can look at work on [Github](https://github.com/gtoledorodriguez/Heap_and_Quick_Sort_GUI).
(https://github.com/gtoledorodriguez/Heap_and_Quick_Sort_GUI)

March 9:

Gladys:

- Started working on the GUI
    - Recreated previous GUI used
        - Modified to not include previous algorithms
- Started programming quick sort algorithm

Grace:

- Started programming heap sort algorithm
    - Translated professor’s pseudocode to Java code

March 10:

Gladys

- Finished quick sort
    - Tested quick sort
        - Did not sort properly, Code rewritten
    - Tested with new code
        - Sorts on a pivot from the start of the array

Grace

- Fixed heap sort class

March 11:

Gladys

- Reimplemented and tested new quicksort
    - Realized one values not being sorted properly

Grace

- Updated the heap sort function to print out each step
- Updated the heap sort function to print each step as a heap

March 12:

Gladys

- Added toString to print out array as it sorts

- Fixed algorithm because it was going to slow

Grace

- Added the heapsort functionality to the GUI

Analysis:

Heap Sort:

The general case for Heap Sort has a complexity of $O(n\log_2(n))$.
Here's the breakdown for this analysis:

The max-heapify function (pseudocode is given below) runs in $O(\log_2(n))$ time.

```
MAX-HEAPIFY(A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
3 if l <= A.heap-size and A[l] > A[i]
4       largest = l
5 else largest = i
6 if r  A.heap-size and A[r] > A[largest]
7       largest = r
8 if largest != i
9       exchange A[i] with A[largest ]
10      MAX-HEAPIFY(A, largest)
```

The time to fix A[Left(i)], A[Right(i)], and A[i] and the relationships among them is $\Theta(1)$. Additionally, the time to run max-heapify on a subtree rooted at one of the children of the node i is added. The height of these subtrees is at most 2n/3, when the bottom level of the tree is exactly half full.
The running time of max-heapify can be written as the recurrence:
$T(n) <= T(2n/3) + \Theta(1)$
By case 2 of the master theorem where a = 1 and b = 3/2, so $T(n) = \Theta(n^{\log 3/2(1)}\log_2(n)) = \Theta(\log_2)$, the solution to this recurrence is
$T(n) = O(\log_2(n))$

The build-max-heap function (pseudocode is below) runs in linear time, or O(n).

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A:length
2 for i = ⌊A.length/2⌋ downto 1
3         MAX-HEAPIFY(A,i)
```

Each call to max-heapify takes $O(\log_2(n))$ time, and this happens O(n) times. That makes the running time/upper bound O(nlog(n)), which is not asymptotically tight. To give it a tighter bound, observe that an n-element heap has height $\lfloor \log_2(n) \rfloor$.

$$\sum_{h=0}^{\lfloor log_2(n) \rfloor} \lceil \frac{n}{2^k} \rceil O(h) = O(n \sum_{h=0}^{\lfloor log_2(n) \rfloor} \frac{h}{2^h})$$

Substitute x = ½

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-2/2)^2}$$

So the bound of the running time of build-heap-max is

$$O(\sum_{h=0}^{\lfloor log_2(n) \rfloor} \frac{h}{2^h}) = O(\sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

The heapsort function (pseudocode is below) itself runs in $O(n\log_2(n))$.

```
HEAPSORT.A
1 BUILD-MAX-HEAP(A)
2 for i = A.length downto 2
3         exchange A[1] with A[i]
4         A.heap-size = A.heap-size - 1
5         MAX-HEAPIFY(A, 1)
```

This function calls build-max-heap once, which takes O(n) time, and calls $O(\log_2(n))$ n - 1 times. This makes heapsort take $O(n\log_2(n))$ time.

Quick Sort:

The general case for Quick Sort has a complexity of $O(n\log_2(n))$.

Here’s the breakdown for this analysis:

```
QUICKSORT(A, p, r)
1 if p<r
2       q = PARTITION (A, p, r)
3       QUICKSORT (A, p, q - 1)
4       QUICKSORT (A, q + 1, r)
```

```
Partition(A, p, r)
1 x = A[r]
1 i = p-1
3 for j = p to r - 1
4       if A[j] <= x
5             i = i +1
6             exchange A[i] with A[j]
7 exchange A[i+1] with A[r]
8 return i+1
```

The best and average case for a quickSort algorithm is O(n log n). For the recurrence the best running time is

$$T(n) = 2T(n/2) + O(n)$$

By the case 2 of the master theorem, this recurrence has the solution

$$T(n) = O(n \log n)$$

Likewise, the recurrence for the average running time is for an example of t to 1 proportional split

$$T(n) = T(9n/10) + T(n/10) + cn$$

Which reduces to

$$T(n) = O(n \log n)$$

In the worst case run time, the partitioning is unbalanced and as such the reccurence looks like

$$T(n) = T(n-1) + T(0) + O(n)$$

$$= T(n-1) + O(n)$$

$$T(n) = O(n\text{^}2)$$

Heap Sort vs Quick Sort:

Using the GUI, we discovered that the runtime for HeapSort was 208640899 nanoseconds (ns). For QuickSort the runtime is 41094601 ns. When comparing these two we see that the HeapSort has a longer runtime than QuickSort.

Users Manual:

The program was created using the Integrated Development Environment (IDE) Eclipse. This can run without Eclipse but works best with Eclipse.

How to use the heap sort

To run the heap sort algorithm, run the GUIRunner, and when the window pops up, click the heap button. The unsorted random number array displayed at the top will be sorted with heap sort. The white box in the middle of the window will be populated with the steps the algorithm performs in order. At the very bottom of this white box, the final sorted array will be displayed.
You can now click the reset button to create a new unsorted random number array, which you can sort again, or you can click the quick sort button to see how the quick sort would work on the same array.

How to use the quick sort

To run the quick sort algorithm, you'll want to click the run button in Eclipse. This will pop up a window. The user will already be given an unsorted array. From there they click on the button labeled quick, to run the quick sort algorithm. Afterward, you can choose to either try out the heap sort, or you can use the reset button to get a new unsorted array.

Additional notes:

If you want to sort an array of a size different than 100, edit the Values interface. Change the SIZE variable to be the size of the array you would like to sort.