### **Problem 1**

|  |  |  |
| --- | --- | --- |
| **Meeting Date** | **Who was in attendance** | **What was done (1-3 sentences)** |
| MM/DD/YY |  |  |
| 2/15/2024 | Austin  2. David  3. Liam | Discussion about initial overview and focuses of work. Delegated attempts at each algorithm. |
| 2/19/2024 | Austin  2. David  3. Liam | Group programming, covering each other’s work/algorithms and fixing them all (none of them worked). |
|  |  |  |

## **Problem 2 (Heapsort)**

Implement the **book’s** version of Heap-Sort and compare its performance to the performance of Merge-Sort, which you implemented in Homework 1.

To solve this problem, you’ll find it useful to store the binary heap exactly as the book does with the root at index 1 and so on. In other words, just **ignore** the value at index 0 in your list. If you do this, you don’t need to modify the book’s pseudocode very much at all. To put this another way, if you want to store the list 8, 9, 3, 4, 6, 12, 32, 7 you would do it like this:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 8 | 9 | 3 | 4 | 6 | 12 | 32 | 7 |  |
| *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |

You’ll need to implement parent, left, right, max-heapify, build-max-heap, and heapsort.

Once it is working, test its performance on randomly-generated lists of numbers and **create a graph** showing its runtime vs. the runtime of mergesort.

## 

## **Problem 3 (Quicksort w/ Rightmost Item as Pivot)**

Now, implement the book’s version of Quicksort (requiring Quicksort and Partition methods). Compare its performance to randomly-generated arrays with both Mergesort and Heapsort. **Produce illustrative graphs.**

## 

## **Problem 4 (Quicksort w/ Randomly Selected Pivot)**

The problem with Quicksort is that its worst-case behavior can occur if the list is sorted (or almost sorted). Test all 3 algorithms (Heap, Merge, Quick) when sorting **already sorted** arrays. How large an array can you do before Quicksort completely **bombs?** 1100

RecursionError: maximum recursion depth exceeded in comparison

The book’s solution is to randomly select the pivot in the partitioning step. (See Section 7.3). Implement that change to the algorithm and **run experiments (with accompanying graphs)** to show that it actually works).

## 

## **Problem 5 (Quicksort w/ Median-of-3 Selected Pivot)**

Another way of resolving the “Quicksort worst-case problem” is to select the pivot using the Median-of-3 method (described in Problem 7-5). In this situation, if the length of the array is greater than 3, we choose the pivot by examining three values and choosing the median value as the pivot. By tradition, those three values are:

A[p]: The leftmost value

A[r]: The rightmost value and

A[(p+r)//2]: The value at the middle index.

For instance, if our list is [5, 8, 2, 3, 4], we would take the median of 5, 4, and 2. The median of those three values is 4. Then, to use the book’s partitioning algorithm, we would swap that value with whatever is at A[r] and proceed as normal.

Implement this version of Quicksort and compare its performance with the other two versions of Quicksort on randomly-generated lists **and** completely sorted lists.

As an example, for randomly generated-arrays, you might get a graph that looks something like this:

