Algorithm Analysis

Task-1

Algorithm of Array-based accumulation approach

```
void print_array_results (Index *index, int n_results, int n_documents)

        Check if n_results are not zero                                      // O(1)

        Initialise a float array of size n_documents                         // O(1)
        Run a loop from 0 to n_documents-1                                   // O(n)
                Set all the array index to zero                              // O(1)

        Run a loop from 0 to num_of_terms-1                                  // O(n)
                Traverse through each list and look for a document           // O(n)
                        Add Score of the document to the array               // O(1)

        Initialise a min-heap with size n_results                            // O(1)

        Run a loop from 0 to n_documents-1                                   // O(n)
                If array index is less than n_results
                        Insert score and id into min-heap                    // O(logn)
                Else
                        If score in array is greater than score in heap
                                Remove min id with min score from heap       // O(logn)
                                Insert array score into heap                 // O(logn)
        Print the heap                                                       // O(n)
        Free heap                                                            // O(1)

Total time complexity, T(n) = O(1) + O(n) + O(n^2) + O(1) + O(n) + O(n) + O(nlogn) + O(nlogn)
+ O(nlogn) + O(1) + O(1)
```

Hence Time complexity = O(n^2) for worst case scenario.

Task-2

Algorithm of Priority queue-based multi-way merge approach

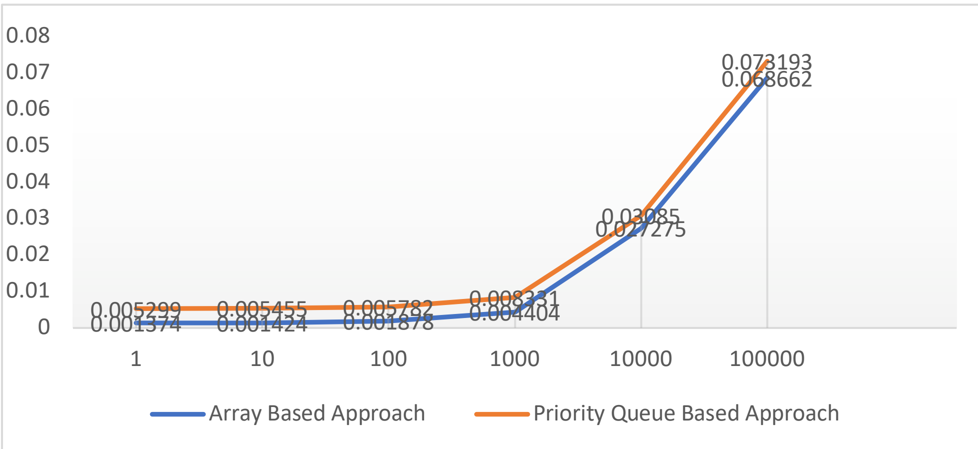
```
void print_merge_results(Index *index, int n_results)

        Check if n_results are not zero                                        // O(1)
        Initialise min-heaplist of size num_terms                              // O(1)
        Initialise a node variable                                             // O(1)
        Initialise a new list                                                  // O(1)
        Initialise n_size variable to 1                                        // O(1)
        Initialise array to store the pointer pointing to first node of doclists.   // O(1)

        Run a loop from 0 to num_terms-1                                       // O(n)
                Assign all the first nodes of doclist to array address         // O(1)
                Insert into heaplist document id, score, first node, doclist index  // O(logn)

        Run for loop for the heap size greater than zero                       // O(n)
                Peek the node of min id from heap and assign it to node        // O(1)
                Peak the doclist index from which the id belongs to            // O(1)
                Remove score of min id                                         // O(logn)
                Assign pointer of document id to Document type                 // O(1)

                Update the n_size of total no of document id                   // O(1)

                Add min id and score to new list l                             // O(1)

                Update the array of pointer with next node of list             // O(1)

                If the next node of doclist is not NULL
                        Assign its data to Document type                       // O(1)
                        Insert into heaplist document id, score, next node
                        doclist index                                          // O(logn)
```

```
Initialise float array of size n_size                                   // O(1)
Run loop from 0 to n_size-1 and set each index to 0                     // O(n)

Traverse through new list l and look at each document                   // O(n)
        Add Score of the document to the float array                    // O(1)

Initialise a min-heap with size n_results                               // O(1)

Run a loop from 0 to n_size-1                                           // O(n)
        If array index is less than n_results
                Insert score and id into min-heap                       // O(logn)
        Else
                If score in array is greater than score in heap
                        Remove min id with min score from heap          // O(logn)
                        Insert array score into heap                    // O(logn)
Print the heap                                                          // O(n)
Free heap, heaplist, list                                   // O(n) + O(1) + O(1)
```

Hence Time complexity = O(nlogn) for worst case scenario.

Based on the asymptotic time complexity that we had calculated, the array-based approach comes out to be O(n^2) and the priority-based approach comes out to be O(nlogn). This means that the array-based approach dominates the priority-based approach when n_results -> ∝.

Analysis of computation time of algorithms with increase in n_results



Graph 1: Where X axis is the n_results and Y axis is the time computation. The code was run in macbook pro (16gb ram).

Task 1 is the array-based approach and Task 2 is the priority-based approach. The Graph 1 shows the time computation of each algorithm in different sets of n_results ranging from 1 to 100000. "Hello" and "world" were used as our query terms. Here we can see that for different values of n_results there is an increase time computation for both the algorithms. The Task 2 takes more time than Task 1 however both start to converge when n_results crosses 1000. The time computation for both approaches appears quadratic as seen from the graph.

Conclusion

Based on the asymptotic time complexity analysis, priority queue-based approach should perform better. However, experimental analysis of Task 1 shows less computation time than Task 2 (refer to graph above). Hence, array-based approach is better than priority queue-based approach though the computation time difference is very small. However, as per the graph the computation time will be less for priority-queue based approach when compared to the array-based approach as n_results -> ∝. Which means when large computations are required, priority queue-based approach will be better.