Complexity analysis

For this analysis, **n** = number of documents indexed (ie maximum length of one docList), **m** = number of terms in the search query, **k**= number of top results return, **log()** = log base 2. Due to the 2 page limit, only worst case analysis was done. General complexities for buildHeap , heapSort, and siftDown (heap operations) are assumed.

Solution for task 1 consists of the following steps (not all O(1) steps outlined):

Step	Description	Complexity
1.	Initialise float array 'Scores' of length n to 0.0	One operation for each document indexed. -best/worst/avg: Θ(n)
2.	Add scores from each document in each docList to Scores[docID]	One operation per document in each docList. Worst case when all m doclists contain scores for all n docs; eg might occur when search terms are generic ("the a"). -Worst case: O(n*m)
3.	Build a document min heap and use to pick top 'k' scores from 'Scores'	Worst case occurs when the first k items are put into the heap O(1) each and buildHeap used O(k), with every item afterwards (= n-k items) in the array needing to be added to the heap (i.e. increasing score order): one comparison/substitution of root of heap O(1) + one siftdown heap operation O(log(k)) for each item = O((n-k)*log(k)). -Worst case: O(k+k+(n-k)*log(k)) = O(n*log(k)) since n>=k.
4.	Heapsort to print topk values in order.	Pop off each item at root and reorganise the heap. General complexity of heapsort. -best/worst/avg: Θ(k*log(k))
TOT		Worst case: O(n*log(k) + m*n + k*log(k)) = O(m*n+n*log(k)) since n>=k

Solution for task 2 consists of the following steps (not all O(1) steps outlined):

Step	Description	Complexity
1.	Build a docList heap (Node heap)	Inserts m Nodes into the heap array O(1) and then uses 'build heap' function to establish heap condition O(m). -Worst case: O(m)
2.	Build a top k heap	Get the first k document scores from the node Heap, and then use build heap function to establish heap property. Worst case is we need to do m*k heap operations on nodeHeap of O(log(m)) each, and insertion without sifting to topk of O(1) for each doc, then buildHeap O(k). -Worst case: O(m*k+k) = O(m*k) since m >= 1.
3.	Accumulate score for lowest document ID from node heap and compare to minimum in topk heap	In worst case where each docList has a score for each doc ID and the document scores are in ascending order, requires m*(n-k) heap operations on the nodeHeap of O(log(m)) to accumulate score and 1 heap operation on topk of O(log(k)) per document ID to siftdown. -Worst case: O((n-k)*(m*log(m)+log(k)) = O(n*(m*log(m)+log(k)) since n>=k
4.	Heapsort to print topk values in order.	Pop off each item at root and reorganise the heap. General complexity of heapsort. -best/worst/avg: Θ(k*log(k))
TOT		-Worst case: O(m*k+n*(m*log(m)+log(k))) = O(n*(m*log(m)+log(k))) since n>=k.

Investigation of input parameters' effects on run time

The time taken for the algorithms for produce a min-heap of the top k results was measured (i.e. time for printing of this heap in order via heapsort was not included since same functionality for both approaches) for various configurations of the inputs. All data points represent the average of 3 measurements, and error bars are the standard deviation of the measurements.

Firstly, the number of input documents n was increased using the -d flag. This simulates having a large number of total documents, but with only a small percentage of them having a score > 0; as might be the case when investigating increasingly uncommon query terms. As shown in figure 1, the runtime for task 1 increased linearly with n. This may be attributed to the increase in the size of the 'Scores' array: since the length of 'Scores' is always n, initialising the array to 0.0 and comparing the array scores against the topk heap are both Ω(n) operations. Task 2 is not affected by the number of documents which are unscored, and so the time to complete is not affected by increasing n without increasing the number of scoring lines in the doclists, as is also demonstrated in figure 1.

Next the number of query terms 'm' was modified (figure 2). Tests were completed with queries made of either a common term ("the") or an uncommon term ("cryptocurrency"), e.g. the query for m=2 for uncommon terms would be "cryptocurrency cryptocurrency". The x axis is made logarithmic for readability. Number of results was set at 10 and the -d flag was unused.

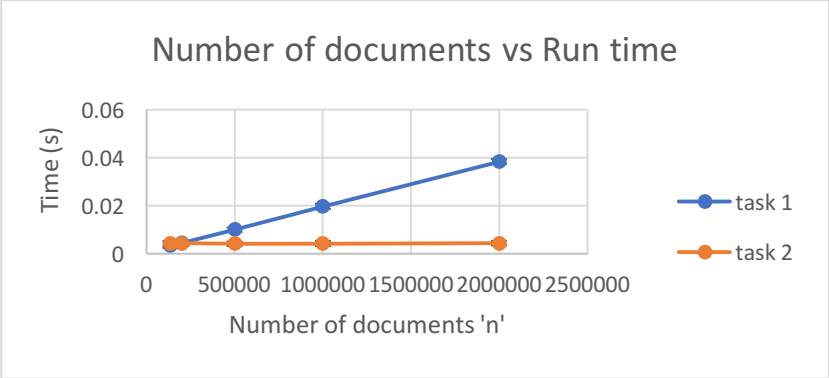

Figure 1: The effect of number of total documents on run time.

It can be seen that task 2 performs extremely efficiently when there are fewer doclists (smaller m) with fewer scores in them (uncommon words), as task 2 only does 'top k heap' comparisons for documents with scores > 0 , vs task 1 which has to compare each value in the 'scores' array of length n regardless. Task 1, however, performs more efficiently with a larger number of common queries, since in this case task 2 now has to compare the top k heap to approximately n documents as most documents have a score, but also has to perform heap operations on the doclistHeap which scale with 'm', which task 1 does not.

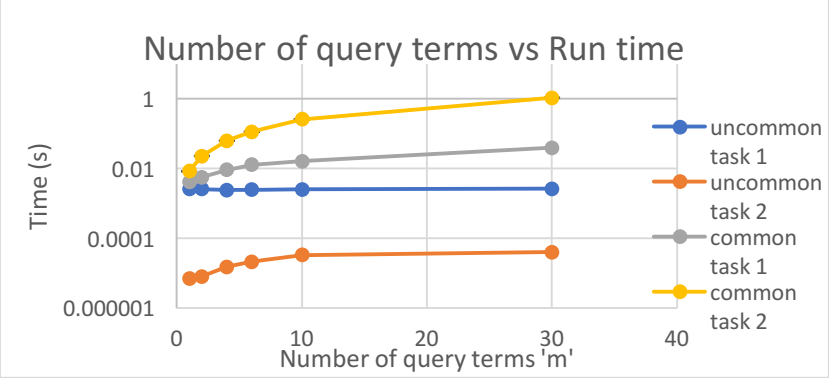

Figure 2: The effect of number and type of query terms on run time. Y axis logarithmic for readability

Finally, the effect of the number of search results requested was investigated (figure 3). A common word query "the" and an uncommon word query "cryptocurrency" were used as the input, and the -d flag was not set.

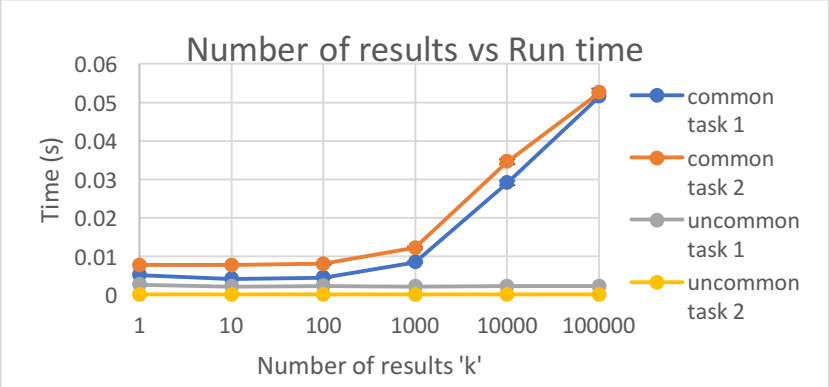

Figure 3: The effect of number of results on run time. X axis logarithmic for readability.

This data shows that tasks 1 and 2 performed similarly for both uncommon and common search terms, which is expected since increasing the size of k simply increases the potential number of swaps a topk heap operation requires. Again, task 2 performs slightly worse than task 1 for common search terms, but slightly better than task 1 for less common terms, for the aforementioned reasons. The common tasks' runtimes increase with k as heap sifting operations are more expensive when a new value that should be in the topk is found, since sifting scales with O(log(k)). The uncommon terms do not increase significantly in runtime since there are fewer than k documents with scores > 0, so topk never fills completely and thus no sifts are needed (aside from one 1 buildheap).

Conclusion

Task 1's approach was found to be faster when most documents have a score attached to them and/or there are many search query terms, as there is no necessity for a secondary heap of docLists. As such, when a search contains many terms that are found in most documents (eg "the is a …"), task 1 is the better choice. On the other hand, Task 2 is faster when there are many documents with a score of 0, as it does not have any $\Omega(n)$ complexity steps. Thus, task 2 is preferable for a fewer number of query terms where the terms are uncommon. Additionally, task 2 might be preferred because it does not require an array of length n to be stored in memory, which might not be possible on systems with limited memory or when dealing with large numbers of documents.