Terminology:	Description:
k = n_results	/*number of results to displace*/
m = n_documents	/*number of documents we have*/
n = n_query	/*number of query words*/
p= aver length of doculists	/*average length of document list*/
log(x)	/*log with base = 2 of value = x*/

Complexity of siftup(), siftdown(), heap_insert(), heap_overwrite_head() and heap_remove_min()

The number of steps required for the function above is related to the while loop in siftup() and siftdown(). And the maximum number that the **While** loop has iterated equals the height of the heap. *// heap.height is 1 based*

h.height = while(heap.max > 0) {count(heap.max / 2)} = times of heap.max can be divided by 2 before getting a 0

i.e. heap.max = $2^{heap.heaight} - 1$ *// 1 for root is odd*

➔ heap.height = log(heap.max+1) ≈ log(heap.max)

∴ complexity of these function ∈ O(heap.height)= O(log(heap.max))

Complexity of swap(), cmp(), min_child(), new_heap(), heap_peek_min(), heap_peek_key(), free_heap(), insert_at_end() and get_heap_size(); print_heap()

As these only have **If** statement and several constant variable assignment, All of these operations ∈Θ(1)

In print_heap(), just print heap's size times. it ∈O(h.cur_size)

query.c task1

For function **initialize_float_array**(), it just sets an array of float to 0.0. So its time complexity ∈Θ(m) as the size of the array is O(m)

For function **traverse_document_list**(), it sums score for each document in doculist[] by a **While** loop traverse doculists in a **For** loop. So its time complexity ∈ Θ(np)

For function **insert_into_heap**(), it uses **heap_insert**() when heap is not full **heap_overwrite_head**() when heap is full and a bigger record is found. So its time complexity ∈O(log(heap.max)).

For function **topk**(), it traverse from document id = 0 to m and try to insert them into the heap of result. So it does **insert_into_heap**() in a **For** loop m times in worst case. So its time complexity ∈O(mlog(heap.max))

For function **rec_print_heap**(), it does heap_peek_min() at most k times then print heap linearly. So its time complexity ∈O(klog(k) + k) ∈O(klog(k))

print_array_results(Index *index, k, m)	
SET score_arr[m]	∈Θ(1)
topk_h←min-heap with h.max= k	∈Θ(1)
score_arr[…] ← 0.0	∈Θ(m)
sum score foreach document	∈Θ(np)
select topk scored document	∈O(mlog(k))
print result	∈O(k log(k))
free_heap(priority_queue)	∈Θ(1)
	∈O(np + mlog(k) + k log(k))

query.c task2

For function **initialize_id_heap**(), it inserts n non-0.0 {score:id} from doculists->head. So its time complexity ∈O(n log(n))

For function **multi_way_merge_topk**(), 2 **While** loops stop when the we have processed all record in doculists. And we access id_heap every loop, push into final result heap with max m times in outer loop. So its time complexity ∈O(np * log(n) + m * log(k))

print_merge_results(Index *index, k)	
priority_queue ← heap with heap.max=k	∈Θ(1)
id_heap ← heap with heap.max=n	∈Θ(1)
tmp ← index->doclists	∈Θ(1)
insert n tmp's head data into id_heap	∈O(nlog(n))
sum score for each document and **select topk** scored document	∈O(nplog(n)+ mlog(k))
print topk selected document	∈O(k log(k))
free(id_heap, priority_queue**)**	∈Θ(1)
	∈O(mlog(k) + nplog(n) + k log(k))

Time Complexity of task1 and task2 analysis

worst case analysis

cost of **print_array_results()**

= O(np + mlog(k) + k log(k))

cost of **print_merge_results**()

= O(mlog(k) + nplog(n) + k log(k))

average case analysis

cost of **print_array_results()** foot note is not power!!!

= O(np + mlog(k)[1] + k log(k))

= O(m + np + klog(k)log(m)), if k <<[2] m = O(m+np)

cost of **print_merge_results**()

= O(mlog(k)[3] + nplog(n) + k log(k))

= O(m + klog(k)log(m) + nplog(n)) , if k << m = O(m+nplog(n))

Space Complexity of task1 and task2 analysis

space of **print_array_results()**	space of **print_merge_results**()
= O(array) + O(prioroty queue)	= O(id_heap) + O(priority queue)
= O(m+k)	= O(n+k)

Task 1&2 in realistic

---

1 "Then the expected number of update steps in total is under k.log n, and each update is O(log k) – and in practice will be 1 or 2 steps. Thus the expected total number of operations is n + k.log k.log n. The worst case is n.log k, for example when the items are in ascending sort order." from w3lec2 slide6 O(mlog(k)) = O(m + klog(k)log(m))

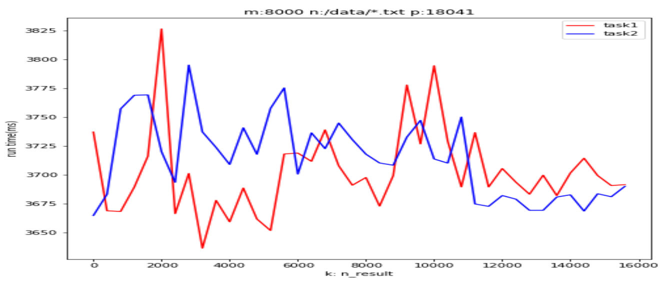
2 << : much smaller

3 simillar as 1

Typically, p will not bigger than m. So p $\leqslant$ m. And the top scored result we want is always smaller than m. So, k << m. And n is smaller than m in common sense. (e.g. search engine like Google has limited query) Both tasks are dominant by O(m) in worst case if n << m. In the use of space, as n < m, Task 2 requires smaller space demand than task1.***Therefore, Task2 has a better performance than task1 in realistic problem.*** For these two algorithms, they are more sensitive to the change in m and p than n and k. Because m is much larger than the other two in real problem.

Analysis of changes in k: n_results

In this case, m, p and n are set to be constant, k is variable. Both tasks are dominant by O(klogk), which means they will have similar time consumption when k is quite large about k=10000. ***So task2 and task1 have similar time consumption when k grows.***

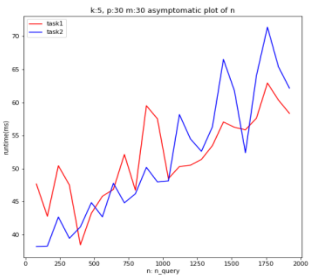


figure(1)[4]

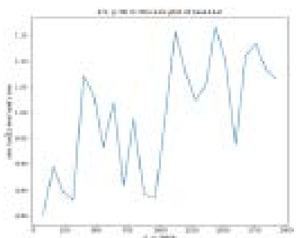
figure(1) is plot for k in range(16000) shows task1 and task2 has different runtime initially. But eventually, they have a trend to have similar runtime as shown in graph. So the analysis above about k is supported by this graph.

Analysis of changes in n: n_queries

In this case, m, p and k are set to be constant, n is variable. On average, task1's complexity is dominant by $O(np)\in O(n)$ as $n\to\infty$. And task2's complexity is dominant by $O(nplog(n))\in O(nlog(n))$ as $n\to\infty$. ***So task2 consumes more time than task1 in the long run as n grows to ∞.***



figure(2)

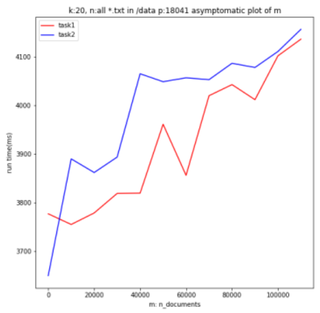


figure(3)

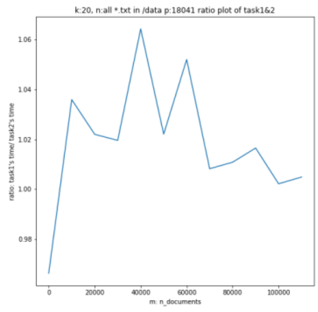
figure(2) shows task2 will requires more time than task1 as n increases in range(2000) excluding print_heap(). figure(3) shows in practical, task1 runs 5%-10% faster than task2 as n >> m around n=1000 as it is a plot about task2 runtime/ task1 runtime. And the rate is increasing which illustrated in the analysis above.

Analysis of changes in m:max number of documents

In this case, n, p and k are set to be constant, m is variable. On average, both task1 and task2 is dominant by O(m) as $m\to\infty$. ***As a conclusion, task1 and task2 will have a similar time consumption , when m grows quite large.***



figure(4)

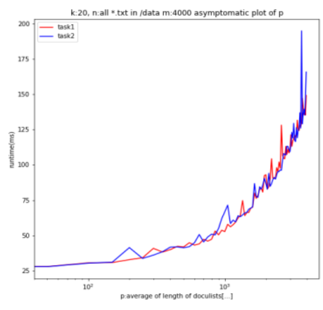


figure(5)

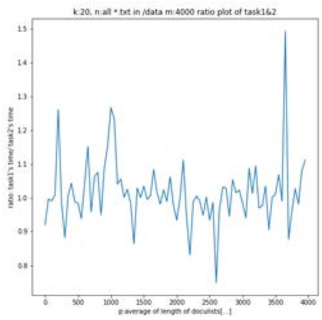
figure(4) is plot for m in range(1000000) including runtime of print_heap(). Which shows when m is dominant, two algorithms have similar time consuming about m=80000. And figure(5) shows the ratio between two algorithm's running time approaches to each other as m $\to\infty$. And two algorithms' runtime has a linear growth determined by O(m).

Analysis of changes in p: average length of doculist[]

In this case, n, m and k are set to be constant, p is variable. On average, both task1 and task2 has $p\leqslant m$. When $p\to m$, task1 $\in O(m+np)=O(p)$ and task2 $\in O(m+nolog(n))=O(p)$. ***As a conclusion, task1 and task2 will have a similar time consumption , as $p\to m$.***



figure(6)



figure(7)

figure(6) plots p in range(4000) excluding print_heap() and shows task1 and task2's runtime approaches to each other as $p\to m$ about p=2500. figure(7) shows the ratio becomes stable when we exclude outlier as p grows. These 2 figs show the analysis of O(p).

Conclusion

As discussed in average analysis, although task2 requires more time consumption than task1 as n grows. But it is just a smaller factor of O(log(n)) in time consumption And task1 and task2 have a huge difference in memory space demand. On average, when n is not so close to m, task2 will require a slightly longer time to get the same result as task1 but require a smaller memory demand.

Task2 is preferred than task1 when n<<m for people who have normal search eager to get their results as soon as possible. Task1 is more preferred than task2 when people who have a huge huge amount of data than our daily query requirement want a shorter query time.

4 every point is iterated 3~4 times and yaxis of plot is runtime or ratio, xaxis is parameter.