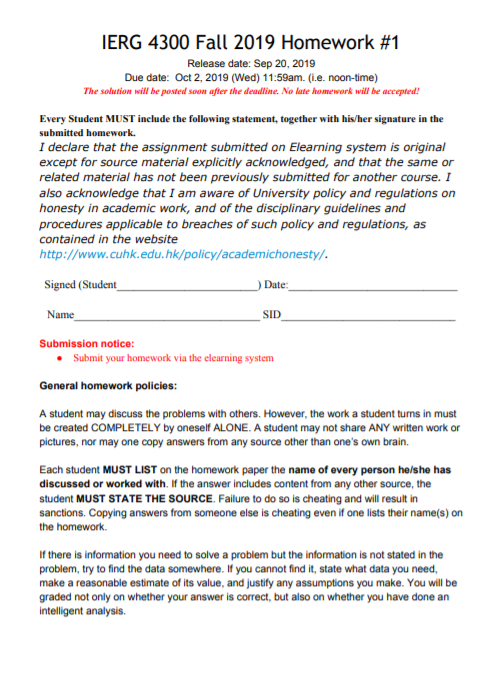
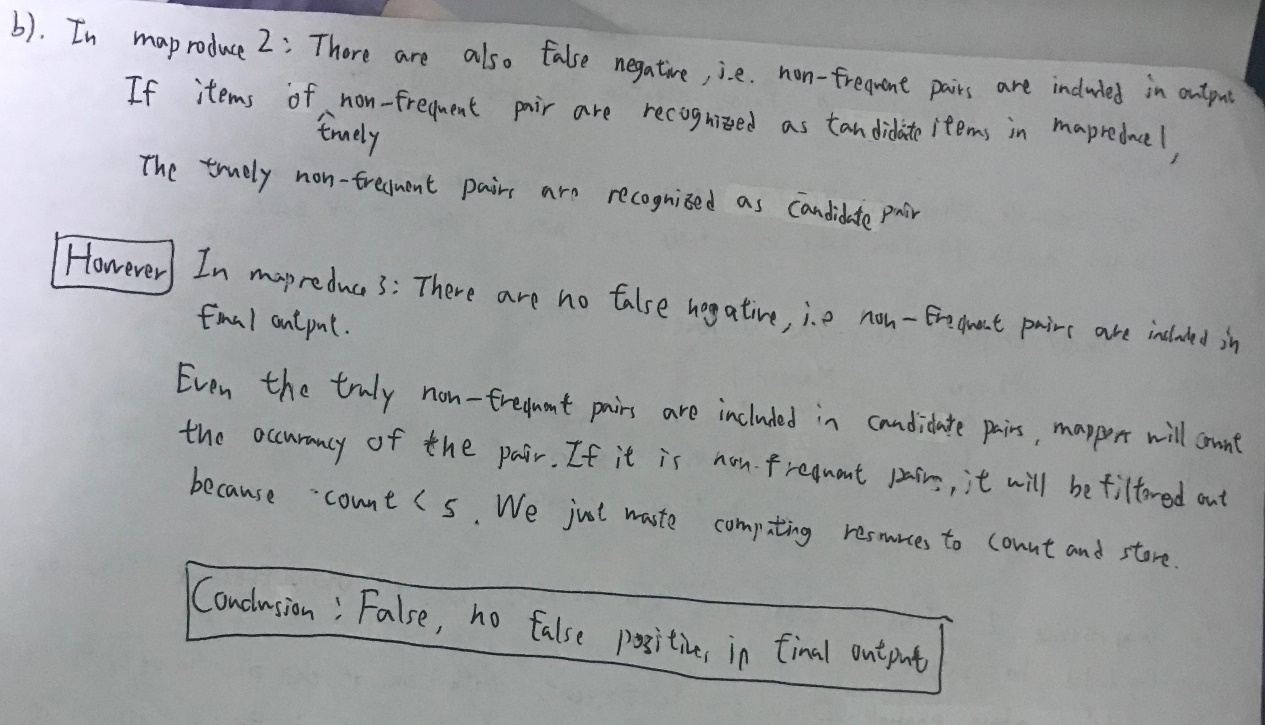
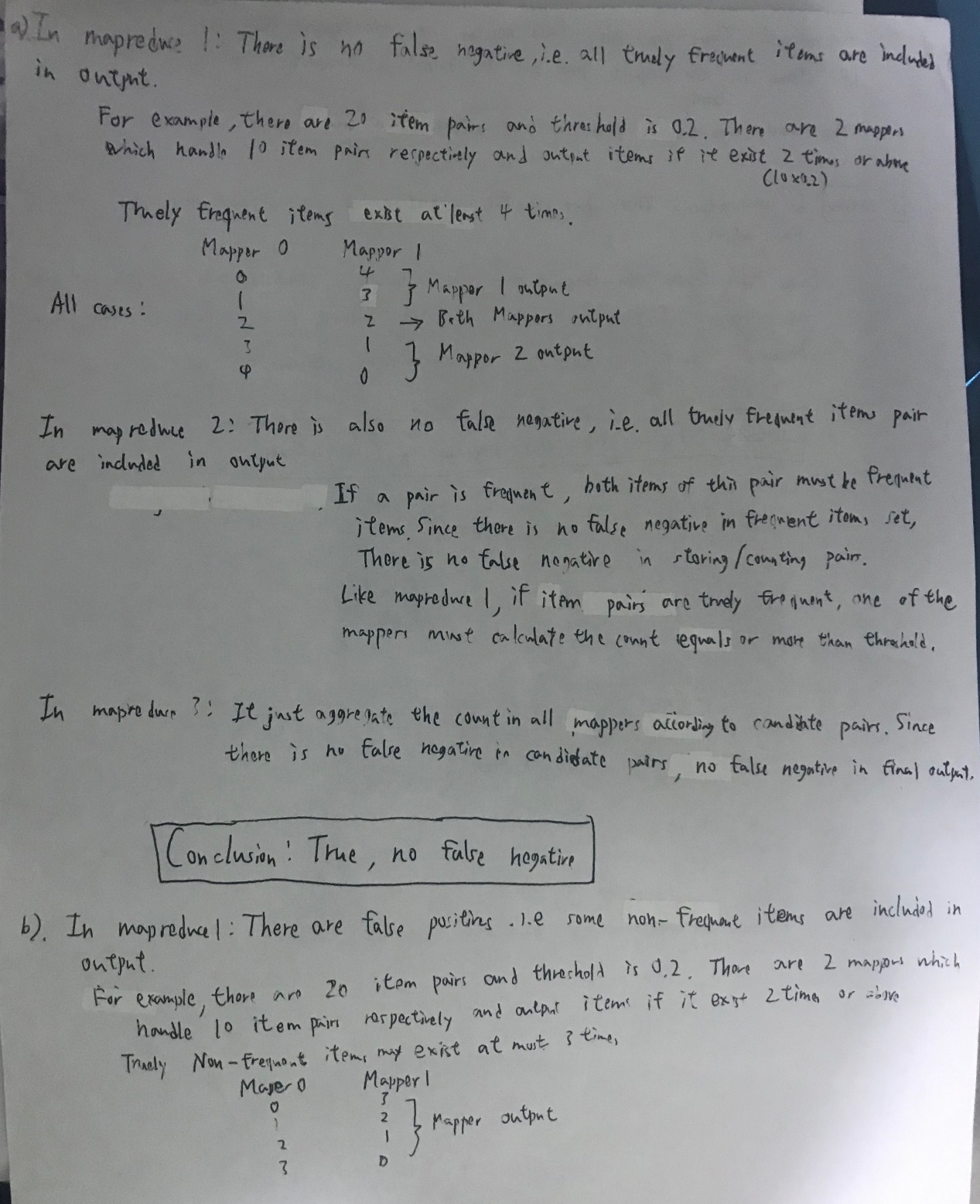
Chim Ka Long

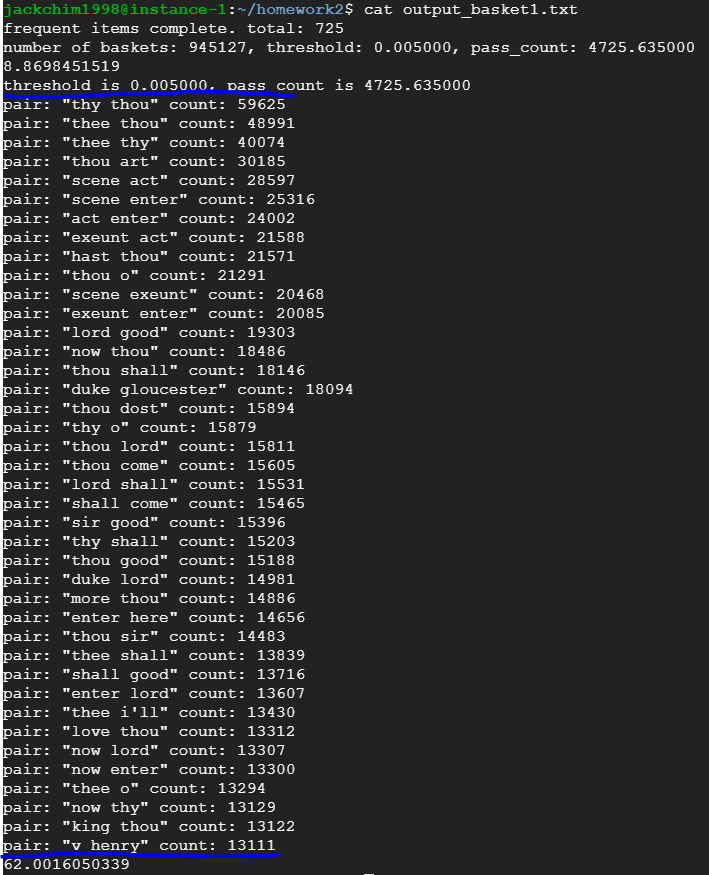
1155094482



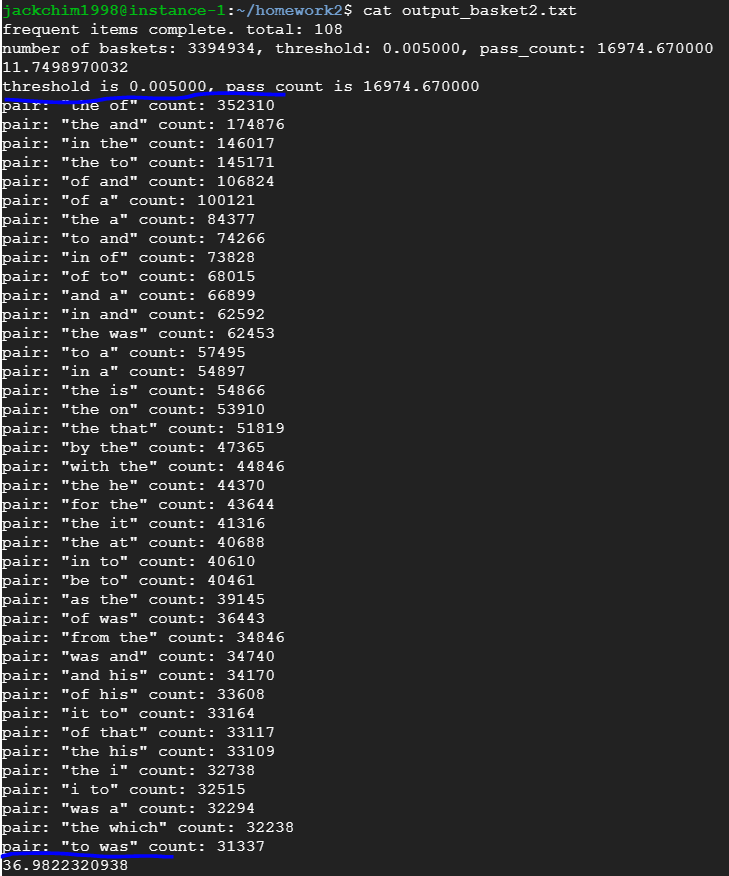
Q0:



Q1a) basket1 result:



Basket2 result:



Remark: the number in last line is the elapsed time

Algorithm:

The words in blanket is the variable name in code or explanation.

In pass 1, split each line of input into list of word and process (filter duplicate words) them. Create dictionary (dic) with “word” as key and “count of word” as value. Then loop over whole input file and count the number of baskets (no\_of\_baskets).

After the pass 1, we can calculate the minimum number of count (pass\_count) to pass the threshold (threshod \* no\_of\_baskets). Loop over the dictionary(dic) and put the frequent words into frequent dictionary(dic\_freq) which is consist of “word” as key and “code” as value. Then create a 2D array (pair\_matrix) which has length and width same as number of frequent words to store number of pair of frequent words. We use 2D array instead of triangular matrix because it is easy to implement, and the memory consumption is enough for 2 sample input.

In pass 2, loop over the input and count the pair of items in same basket into 2d array if the both items of pair are frequent items.

After pass 2, we create a list(code\_list) with code as index and word as element to let us find word quickly by code. Also, we create array(pair\_arr\_freq) to store the pair and count. Then loop over 2D array(pair\_matrix) and sum up count (index i,j) and count (index j,i) into pair class (x,y,count)(x and y are code not word) which will be stored in pair\_arr\_freq. Finally, we sort the pair\_arr\_freq according to count in descending order and cut the end those elements do not pass the threshold. Print out the pair in words according to code\_list and count.

Q1b) The result is same as 1a.

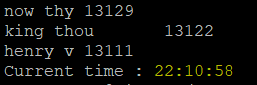
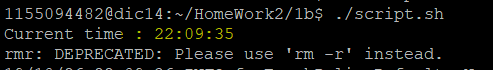
Algorithm:

The mapreduce job0 find candidate pair which is still possible to be frequent pairs. Mapper0 use the A-priori algorithm which in same as 1a. The key idea is a frequent pair must pass the number (local number of baskets in each mapper \* threshold) in one of mappers. Reducer0 just output the pair to file local\_can\_pair\_basket\*.txt.

The mapreduce job1 count all pairs if they are candidate pair and filter out those pairs which do not satisfy with threshold. That is why we use local\_can\_pair\_basket\*.txt as supplementary file to mapper1. In reducer1, we need to know the total number of baskets to calculate pass line (threshold \* number of baskets), so that we need count.txt as supplementary file to reducer1.

Comparison between 1a and 1b:

With basket1 as input, 1a use 62s and 1b use 93s.



With basket2 as input, 1a use 37s and 1b use 66s.

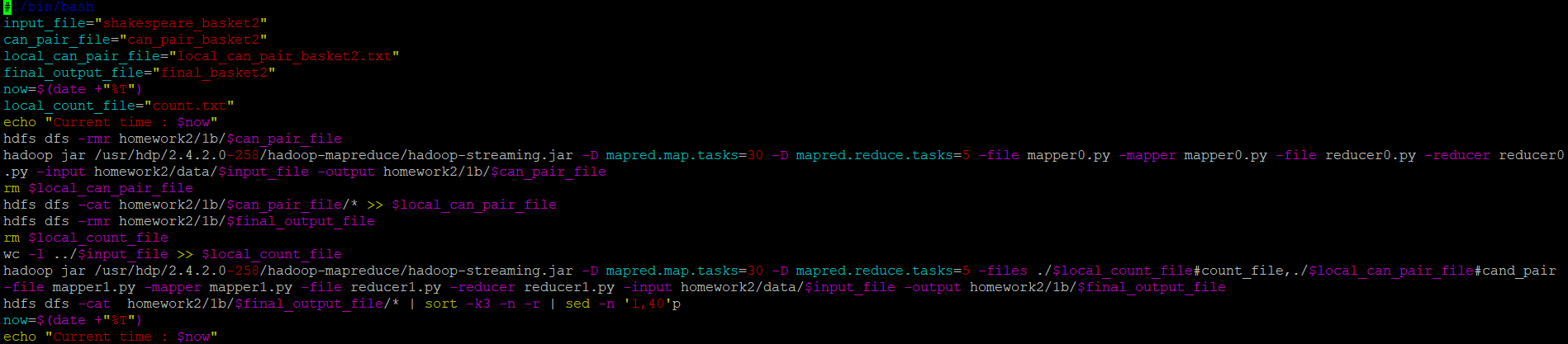
Why using mapreduce consume more time than running in single machine?

First, we need to see the script first. In script, we have command to delete directory of previous output file in HDFS and command to get the result of mapreduce job0 down. Those action will cause overhead.

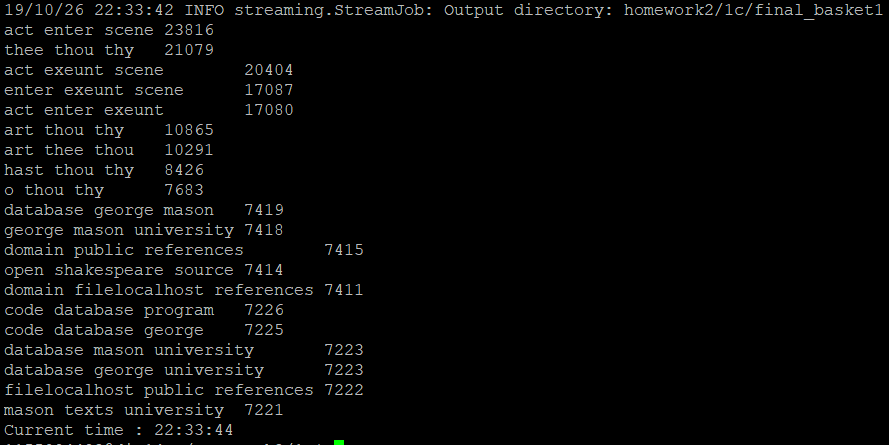
Second, the mapreduce specific 30 mappers and 5 reducers. These number affect the performance. Also, others may also run job on these machines, so that we may need to wait for them finish. However, that is not my situation.

Third, the input file is transferred between machines. Network performance is affected by network overhead. Even the file is very small, we still need to pay those time of overhead.

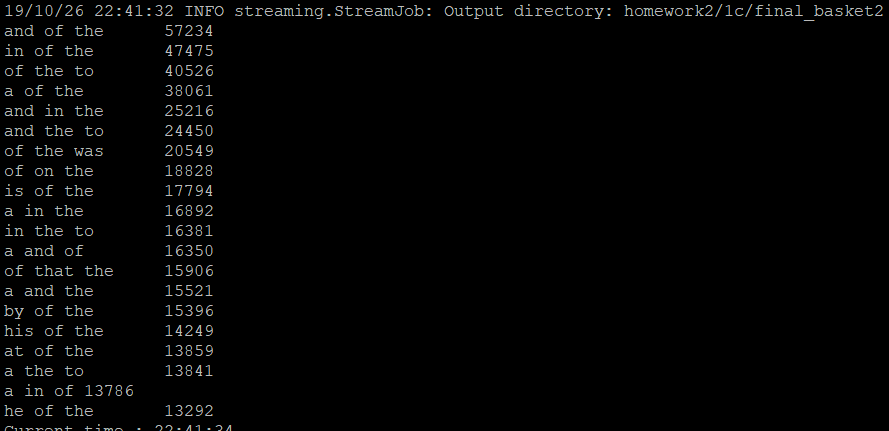
As a result, If the input file is much larger e.g. 100GB, the overhead may become smaller in percentage. If we use more mappers and reducers, the time of calculating can also be smaller.



Q1c) result of basket1:



Result of basket2:



Algorithm:

It is almost same as part 1b, except we find candidate triplet in mapreduce0 and count candidate triplet in mapreduce1. The most different part is how we implement A-priori algorithm to find candidate triplet in part marpper0.

In mapper0, we still need to find candidate pair as part 1a and store them into dictionary(pair\_freq\_dic) which has “pair” (word1 + “ ”+ word2, the word1 is smaller than word2 ) as key and “True” (it can be anything, we use true in our case) as value.

We loop over the baskets to find triplet where items are all frequent. Then we put them into temporary array and sort them. If the any pair of these 3 items are frequent pair in pair\_freq\_dic, we store this triplet into dictinary(cand\_tri\_dic) which has “triplet” as key and “count” as value. Finally, we output those triplet has count over threshold.