COMP6714 ASS01

Zanning Wang z5224151

Q1.

```
answer ← ∅ ;
while p1 ≠ nil ∧ p2 ≠ nil do
    if docID(p1) = docID(p2) then
        l ← [ ];
        pp1 ← positions(p1); pp$ ← positions(p$);
        start = 0; end = pp$;
        range = [start, end];
        while pp1 ≠ nil do
            pp2 = skipTo(p2, docID(p2), pos(pp1));
            if pos(pp1) < pos(pp$)
                while pos(pp1) < pos(pp2) ∧ pos(pp1) ∈ range ∧ pos(pp2) ∈ range then
                    add(l,pos(pp2));
                    pp2 ← next(pp2);
                pp1 ← next(pp1);
                while l ≠ [ ] ∧ pos(pp1) > l[1] do
                    delete(l[1]);
                for each ps ∈ l do
                    answer ← answer ∪ [docID(p1),pos(pp1), ps];
            else
                l ← [ ];
                start = end;
                end = next(pp$);
                pp$ ← next(pp$);
    else
        p$ ← next(p$)
        if docID(p1) < docID(p2) then
            p1 ← next(p1);
        else
            p2 ← next(p2);
return answer;
```

In the box one, set the list empty as an initial, set a list called range to store the sentence index, which means the A and B should both in this range according to the requirement.

In the box two, when we check every pp1 in doc(p1), we should initial the pp2 with the help of skipTo function. The initial pp2 will always after the pp1, after that we should check the same range, if the pairs satisfy the requirement, it will be added to the list L. and in the second while loop it will check the B always after A. and output the final result.

If the box three it will check if the pp1 move exceed pp$, the pp$ will reset to the next pp$ and reset

the range to check the next pp1 and pp2.

Q2.

Part1.

According to the logarithmic merge, when the memory is full, it will create a sub-index and save into disk, when there is a collision at g0 and g1, we will merge sub-indices into new sub-index of g2, for example if there is already g0 and g1 in the disk, at this time, there come a g0, we will merge two g0 and get g1, and plus the older g1 to get a g2, store this g2 in the disk. Obviously, if there are t sub-indexes will be created when apply no merge, In logarithmic merge strategy, the largest sub-index part is t/2 (gn), the smallest sub-index part is 1(g0),therefore the total number of sub-index is same as counting the number of 1, 2, 4, 8 ...t/8, t/4, t/2 which is lower bound($\log_2 t$), as we know lower bound($\log_2 t$) = upper bound(log2t) + 1, so we will have at most $\lceil \log 2t \rceil$ sub-indexes.

Part2.

In the logarithmic merge, the largest part is t/2, which merge $\log_2 t$ times, the smallest part which size is 1, only merge 1 time, therefore the total cost will no more than each sub-index merge $\log_2 t$ times, which is $t \cdot M \cdot \log_2 t$. the total I/O cost of the logarithmic merge is $O(t \cdot M \cdot \log 2t)$.

Q3.

Part1:

0 1000 10111 11000101 11000000 11000011 11011011011

The result of decoding is below: 1, 2, 7, 13, 8, 11, 91 in order.

According to the delta encoding, we can get k_{dd} by counting the number of 1 until the first "0", to take 110 00 101 as an example, in this sequence, k_{dd} is 2, after checking k_{dd}, we should check k_{dr}, which is 00. we translate binary into decimalism, then we get the K_{dr} is 0. After that we should know how many digits that we should check to get the K_r. By calculating $2^{kdd} + k_{dr} - 1$, which is 3 in this sequence. Therefore, we should check 101 which is Kr, after that we get Kr is 5. The final result k is $2^3 + 5 = 13$ for this sequence.

The detailed of each sequence as below:

Sequence	Result	K_d	K_{dd}	K_{dr}	K_r
0	1	0	0	0	0
1000	2	1	1	0	0
10111	7	2	1	1	3
11000101	13	3	2	0	5
11000000	8	3	2	0	0
11000011	11	3	2	0	3
11011011011	91	6	2	3	27

In conclusion, the compressed list is: [1, 2, 7, 13, 8, 11, 91]

Part2:

From the last part, we get the list which store the gaps, as we know the document IDs is a increasing order of sequence, therefore we can get the final documents IDs by adding the gap from 0, which is [1, 3, 10, 23, 31, 42, 133]

Therefore, the Final document IDs is [1, 3, 10, 23, 31, 42, 133].

Q4.

Part1:

The single lines which lead to a bug is at line 21 show belows:

```
1. Function next(θ)
2.    repeat
3.       /* Sort the terms in non decreasing order of
          DID */
4.       sort(terms, posting)
5.       /* Find pivot term - the first one with accumulated
          UB ≥ θ */
6.       pTerm ← findPivotTerm(terms, θ)
7.       if (pTerm = null) return (NoMoreDocs)
8.       pivot ← posting[pTerm].DID
9.       if (pivot = lastID) return (NoMoreDocs)
10.      if (pivot ≤ curDoc)
11.         /* pivot has already been considered, advance
             one of the preceding terms */
12.         aterm ← pickTerm(terms[0..pTerm])
13.         posting[aterm] ← aterm.iterator.next(curDoc+1)
14.      else /* pivot > curDoc */
15.         if (posting[0].DID = pivot)
16.            /* Success, all terms preceding pTerm belong
                to the pivot */
17.            curDoc ← pivot
18.            return (curDoc, posting)
19.         else
20.            /* not enough mass yet on pivot, advance
                one of the preceding terms */
21.            aterm ← pickTerm(terms[0..pTerm])
22.            posting[aterm] ← aterm.iterator.next(pivot)
23.      end repeat
```

The reason is that：

In this loop, we will finally output the result, which the DID of each term are same as the current pivot. One probably reason which may cause the infinite loop is that: every time we change the posting of the aterm to the next iterator, the posting DID of current aterm will keep same as before. After reading the essay given carefully, I find in the given function called: pickTerm(), which have a preference to choose the term with higher IDF among each term. Therefore, if there is a situation that: when all other terms except one have already belonged to the pivot, we only need to move the one that's left to the next DID to get the final output, however the value of current IDF in this term is incredible small compared to the others. In this case, the pickTerm() function will may never picking this term and pick other items with higher IDF instead, which means never move forward to the left one term which indirectly cause an infinite loop in this program.

policy may improve performance, it cannot affect precision. In our current implementation pickTerm() selects the term with the maximal idf, assuming the rarest term will produce the largest skip. Identifying an optimal selection strategy is outside the scope of this paper. A recent on-going research on this issue is described in [3].

Part2.

Suppose there is a situation that:

	A	B	C
UB	6	5	7
list	<1,3> <2,5> <4,4>	<1,4> <2,2> <5,2>	<1,4> <2,4> <4,1> <5,3>

In the first few turn, we set the pivot as 1, and can get result 11 （3+4+4） as expected, but when the program try to get the second result, in this few repeat, we suppose at the beginning term C was selected to set the povit as 2 and move to next iteration, which is <2,4> due to line 10 and 11, and we also suppose in this program, the threshold was set as 12, in order to exceed 12, C was set as pTerm because of the new no deceasing order(ABC), we should use pickTerm() function to choose one of term from A and B to move on in the next few repeat, but due to the mechanism of pickTerm(), at current situation, A was far more likely be selected than B because of IDF, which will lead a bug here, the pointer which point to <2,5> will move to <2,5> again. Hence, the program will run infinity.