Q1

Method 1

Modification is like this:

<term, number of docs containing term;
doc1: (paragraph1, sentence1, position1), (paragraph2, sentence2, position2) …;
doc2: (paragraph1, sentence1, position1), (paragraph2, sentence2, position2) …;
etc.>

e.g.
<be: 993427;
1: (1,2,7), (1,5,18), (2,3,33), (2,4,72), (3,5,86), (5,2,231);
2: (1,2,2), (33,55,149);
..... …>

Method2

Another way to modify the inverted posting list is like this: for each document, maintain three list, namely the term position list, the list of paragraph ID that contains the term, the list of sentence ID that contain the term. This is much less redundant than method1 .

<term, number of docs containing term;
doc1: position1, position2 … ;
paragraph1, paragraph2…;
sentence1, sentence2…;
doc2: position1, position2 … ;
paragraph1, paragraph2…;
sentence1, sentence2…;

etc.>

In the postings, for each term, group the paragraph number, sentence number and word position number together and store it, in ascending order by paragraph number, then by sentence number, then by position number.

(i)/k within k words is already available in original positional inverted index, and this modification will still support /k.
a simple description of algorithm for /k option: take term1/k term2 for example, here /k is a constant number.
steps:
1. retrieve the above modified posting list for term 1 and term 2
2. first merge: only retrieve the document that has both terms. the result has two postings, i.e. posting list 1, posting list 2.
Posting list 1 contains **document Ids and the position number** group for term 1, the document in posting list1 should also contain term2. I call it **position list.**

Similarly, posting list 2 contains document Ids and position number for term 2, the document in posting list2 should also contain term2.
So literally, posting list 1 and posting list 2 has same document IDs, but posting list 1 has the position for term 1, posting list 2 has the position for term 2.
3. second merge: merge the position list.
process doc by doc. For each document, we got two pointers, pointer 1 points to the position list for term 1 for the specific document, pointer 2 points to the position list for term 2 for the specific document. If the value difference for pointer1 and pointer 2 are within k, we add the doc ID to the final answer and process the next document. Otherwise we increment the pointer with a smaller value and do the same comparison until one pointer is nil.

(ii)/s is for terms that appear in the same sentence in a document. So when using the merge algorithms, need to check the terms before and after /s option that these terms within the same document possess the same sentence number.
A simple description of the /s option: suppose we want to get term1/s term2, here are some steps:
1. retrieve the above modified posting list for term 1 and term 2
2. first merge: only retrieve the document that has both terms. the result has two postings, i.e. posting list 1, posting list 2.
Posting list 1 contains **document Ids and the sentence number** group for term 1, the document in posting list1 should also contain term2. I call it **sentence position list.**
Similarly, posting list 2 contains document Ids and sentence number for term 2, the document in posting list2 should also contain term2.
So literally, posting list 1 and posting list 2 has same document IDs, but posting list 1 has the sentence position for term 1, posting list 2 has the sentence position for term 2.
3. second merge: merge the sentence list.
process doc by doc. For each document, we do an intersection for **sentence position list** in term 1 and **sentence position list** in term 2 for doc. If the intersection result is not empty, add the docID to the final result.

(iii) /p is for terms that appear in the same paragraph in a document. So when using the merge algorithms, need to check the terms before and after /p option that these terms within the same document possess the same sentence number.
The algorithm is similar to /s option, only to retrieve the paragraph position list instead. And do the intersection for the paragraph position list, remain only non-empty sets.

Then finally, we got the answer set containing doc IDs for each proximity operator or singular term. We then do an AND operation for each proximity operator result, a.k.a, and intersection. But as a space is treated as disjunction, we then do a union for the spaces.

as for the /k, /s, /p proximity operators, the merging algorithms are pretty much the same. e.g, for /k, retrieve the position list and merge, for /p, retrieve the paragraph list and merge, for /s, retrieve the sentence list and merge.

Q2

(1) Suppose we choose n skip pointers($n \leqslant L$). Then the length of the gap between each pointer is $\frac{L}{n}$.

For step 1, Since every posting list has same probability of being searched, the worse case search time is n (has to skip to the last pointer).

For step 2, the length of the target segment is $\frac{L}{n}$, and for the worst case, the time is $\frac{L}{n}$ - 2 (fist and last pointer already checked)

So total worse case time is $n + \frac{L}{n}$.

Let f(n) = $n + \frac{L}{n}$ - 2. we want to find the minimum value for f(n), so we need to take the derivatives of f(n) and find a n such that f(n)`=0

f(n)`=1-$\frac{L}{n^2}$ . let f(n)` = 0, we have n=$\sqrt{L}$.

So choosing $\sqrt{L}$ skip pointers has the best worst case performance if perform sequential search in both step 1 and step2.

(2) Assume we got n pointers in step 1. Then $n \leqslant log_2 L$, and the gap(i.e. length) of the target segment is G:= $\frac{L}{2^{n-1}}$.

for step one , the time complexity is O(n)

For step two, since every posting has same probability of being searched, the average searching time for step two is $\log_2 G = \log_2 \frac{L}{2^{n-1}} = \log_2 L + \log_2 2^{1-n} = \log_2 L - n + 1$

Total time is f(n) = $n + \log_2 L + 1 - n = \log_2 L + 1$

So the time complexity is constant time, it doesn't really affect the time complexity what number we assign for n.

(3) Assume we got n pointers in step 1. Then $n \leqslant log_2 L$, and the gap(i.e. length) of the target segment is

for step one, the time is $\log_2 n$.

For step two, since every posting has same probability of being searched, the average searching time for step two is L/(2n)

Total time is f(n) = $\log_2 n + L/n$

Now we want to find the minimum value for f(n) so we need to take the derivatives of f(n) and find a n such that f(n)`=0

f(n)`= $\frac{1}{n \ln 2}$ - L/ (n^2) = 0

n=L·ln2

so L·ln2 is a solution.

and f(n) is a decrement function. So we should take n as big as possible. Since $n \leqslant log_2 L$, hence $log_2 L$ is the best number of skip pointers. Which means that sequential search averagely performs worst than binary search, meaning the second step doesn't really need if we want to get the best search performance.

Q3

$$score(d,Q) = \sum_{t \in Q} idf_t \cdot \frac{(k_1+1)tf_{t,d}}{k_1((1-b)+b\frac{L_d}{L_{ave}})+tf_{t,d}} \cdot \frac{(k_3+1)tf_{t,Q}}{k_3+tf_{t,Q}}$$

score(d,Q) = $\sum_{t\in Q} idf_t \cdot \frac{3tf_{t,d}}{2+tf_{t,d}} \cdot \frac{3*1/3}{2+1/3} = \sum_{t\in Q} idf_t \cdot \frac{3tf_{t,d}}{2+tf_{t,d}}$

score(d,A)$|_{tf=1}$ = $idf_A \cdot \frac{3tf_{A,d}}{2+tf_{A,d}}$ = 6 * $\frac{3}{3}$ = 6 ;

score(d,A)$|_{tf=10}$ = $idf_A \cdot \frac{3tf_{A,d}}{2+tf_{A,d}}$ = 6 * $\frac{30}{12}$ = 15 ;

score(d,A)$|_{tf=100}$ = $idf_A \cdot \frac{3tf_{A,d}}{2+tf_{A,d}}$ = 6 * $\frac{300}{102}$ = 17.64 ;

score(d,A)$|_{tf=1000}$ = $idf_A \cdot \frac{3tf_{A,d}}{2+tf_{A,d}}$ = 6 * $\frac{3000}{1002}$ =17.96≈18 ;

so we can see that, with the tf→ ∞, maxscore for keyword A is very close to a constant number 18, so it doesn't really matter with the posting list.

score(d,B)$|_{tf\text{-}1000}$= $idf_B \cdot \frac{3tf_{B,d}}{2+tf_{B,d}}$ = 2 * $\frac{3000}{1002}$ ≈6 ;

so we can see that, with the tf→ ∞, maxscore for keyword B is very close to a constant number 6, so it doesn't really matter with the posting list.

score(d,C)$|_{tf\text{-}1000}$= $idf_C \cdot \frac{3tf_{C,d}}{2+tf_{C,d}}$ = 1 * $\frac{3000}{1002}$ ≈3 ;

so we can see that, with the tf→ ∞, maxscore for keyword B is very close to a constant number 3, so it doesn't really matter with the posting list.

Above all, the given fomular limits the impacts of tf(when tf→ ∞)

score(d,A) ⩽18, score(d,B) ⩽6 ; score(d,C) ⩽3 ;

so the maxscores for the term A, B, C are 18, 6, and 3.

（2） in(1) we have:

score(d,Q) = $\sum_{t\in Q} idf_t \cdot \frac{3tf_{t,d}}{2+tf_{t,d}} \cdot \frac{3*1/3}{2+1/3} = \sum_{t\in Q} idf_t \cdot \frac{3tf_{t,d}}{2+tf_{t,d}}$

let f(t) = $\frac{3t}{2+t}$,

we have

score(d,Q)$|_{Q=\{A,B,C\}}$= 6f(tf1)+2f(tf2)+f(tf3)

we have score(d)

for document D1, we have

score(D1) = 6f(1)+2f(1)+f(1)=9;

for document D2, the score is

score(D2) = 6f(8)+2f(0)+f(2)= 14.4+1.5= 15.9

At this point, score(D1) and score(D2) are the current top-2 results. τ`= 9. Since 3+6⩽9, Only A need to be considered.

No need to calculate the score of D4.

Then we score D5.

Score(D5)=6f(3) + 2f(4) + f(2) = 10.8 + 4 + 1.5 = 16.3 and . τ`= 15.9

The next Document to process is D8

Score(D8)= 6f(10) + 2f(0) + f(1) = 16

Now the posting list of A is ended, so the final top-2 documents are D5 and D8.

In conclusion, the above algorithm calculates the score of 4 documents and visited 10 postings.

term	idf	postings
A	6	$(D_1:1),(D_2:8),(D_5:3),(D_8:10)$
B	2	$(D_1:1),(D_5:4),(D_6:1),(D_7:4)$
C	1	$(D_1:1),(D_2:2),(D_4:1),(D_5:2),(D_6:3),(D_8:1),(D_9:1),(D_{10}:3),(D_{11}:7)$

TABLE 1. Posting Lists

Q4

k	1	2	3	4	5	6	7	8	9	10
Precision(%)	100	100	66.67	50	40	33.33	28.57	25	33.33	39
recall(%)	12.5	25	25	25	25	25	25	25	37.5	37.5
k	11	12	13	14	15	16	17	18	19	20
precision(%)	36.36	33.33	30.77	28.57	33.33	31.25	29.41	27.78	26.32	30
recall(%)	50	50	50	50	62.5	62.5	62.5	62.5	62.5	75

(1) What is the precision of the system on the top-20?

6/20

(2) What is the F1 on the top-20?

F1 = (2*6/8*6/20)/(6/20+6/8) = 0.4286

(3) What is/are the uninterpolated precision(s) of the system at 25% recall?

100%, 66.67%, 50.00%, 40.00%, 33.33%, 28.57%, 25.00%

(4) What is the interpolated precision at 33% recall?

It's the maximum precision achieved for k⩾9. Which is 4/11 = 0.363636

(5) Assume that these 20 documents are the complete result set of the system. What is the MAP for the query?

(1/1 + 2/2 + 3/9 + 4/11 + 5/15 + 6/20) /8 = 0.4163

Assume, now, instead, that the system returned the entire 10; 000 documents in a ranked list, and these are the 1st 20 results returned.

(6) What is the largest possible MAP that this system could have?

(1/1 + 2/2+3/9+4/11+5/15+6/20+7/21+8/22)/8 = 0.5034

(7) What is the smallest possible MAP that this system could have?

(1/1 + 2/2+3/9+4/11+5/15+6/20+7/9999+8/10000)/8 = 0.4165

(8) In a set of experiments, only the top-20 results are evaluated by hand. The result in (5) is used to approximate the range (6) to (7). For this example, how large (in absolute terms) can the error for the MAP be by calculating (5) instead of (6) and (7) for this query?

0.5034-0.4163 = 0.0871