Hincu Alice Ramona

Problem 1

Doc1 breakthrough drug for schizophrenia
Doc2 new approach for treatment of schizophrenia
Doc3 new hopes for schizophrenia patients
Doc4 new schizophrenia drug

1.1 Draw the term-document incidence matrix for this document collection.

Term – A “normalized” (case, morphology, spelling etc) and unique word. It is included in the index.

First of all we need to extract the terms: breakthrough, drug, for, schizophrenia, new, approach, treatment, of, hopes, patients.

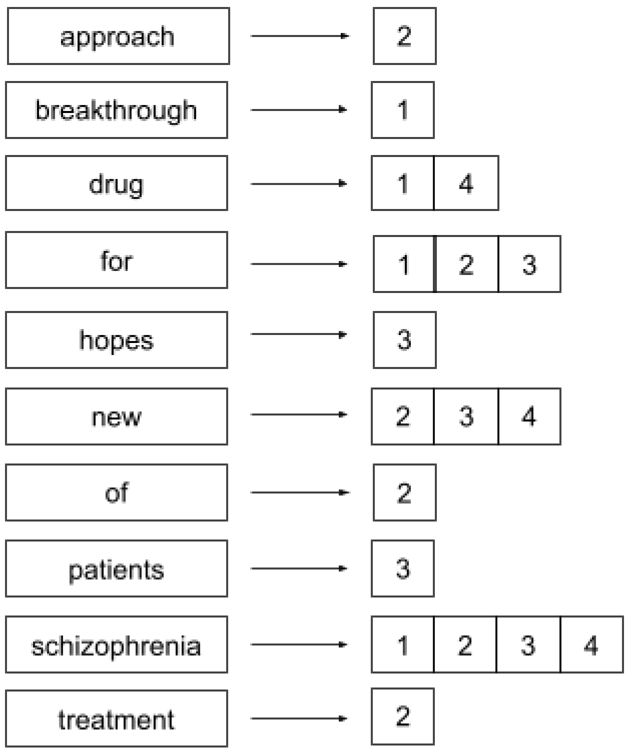
In a term-document incidence matrix, rows represent terms and columns represent documents. Entry is 1 if term occurs.Entry is 0 if term doesn’t occur.

Then, we create the term-document incidence matrix

	Doc 1	Doc 2	Doc 3	Doc 4
approach	0	1	0	0
breakthrough	1	0	0	0
drug	1	0	0	1
for	1	1	1	0
hopes	0	0	1	0
new	0	1	1	1
of	0	1	0	0
patients	0	0	1	0
schizophrenia	1	1	1	1
treatment	0	1	0	0

1.2 Draw the inverted index representation for this collection, as in Figure 1.3 in IIR.

For each term t, we store a list of all documents that contain t.



1.3. What are the returned results for these queries:
(a) schizophrenia AND drug
(b) for AND NOT(drug OR approach)

(a) schizophrenia AND drug

To find all matching documents using inverted index:

1 Locate *schizophrenia* in the dictionary
2 Retrieve its postings list from the postings file
3 Locate *drug* in the dictionary
4 Retrieve its postings list from the postings file
5 Intersect the two postings lists
6 Return intersection to user

schizophrenia -> 1->2->3->4
drug -> 1->4
=> schizophrenia AND drug -> 1->4

The result is: document 1 and document 4

(b) for AND NOT(drug OR approach)

To find all matching documents using inverted index:

1 Locate *drug* in the dictionary
2 Retrieve its postings list from the postings file
3 Locate *approach* in the dictionary
4 Retrieve its postings list from the postings file
5 Make the union of these postings lists
6 Apply the NOT operation to the union
7 Locate *for* in the dictionary
8 Retrieve its postings list from the postings file
9 Make the intersection of these postings list
10 Return intersection to user

drug -> 1->4
approach -> 2
=> drug OR approach -> 1->2->4
=> NOT (drug OR approach) -> 3
for -> 1->2->3
=> for AND NOT(drug OR approach) -> 3

The result is: document 3

Problem 2

2.1. Write out a postings merge algorithm, in the style of Figure 1.6 in IIR, for an x OR y query.

```
OR(p1, p2)
answer <- ( )
while p1 ≠ NIL and p2 ≠ NIL do
    if docID(p1) = docID(p2) then
        ADD(answer, docID(p1))
        p1 <- next(p1)
        p2 <- next(p2)
    else if docID(p1) < docID(p2) then
        ADD(answer, docID(p1))
        p1 <- next(p1)
    else
        ADD(answer, docID(p2))
        p2 <- next(p2)

while p1 ≠ NIL do
    ADD(answer, docID(p1))
    p1 <- next(p1)
while p2 ≠ NIL do
    ADD(answer, docID(p2))
    p2 <- next(p2)

return answer
```

Explanation: This is the pseudocode for merging two lists. It has two pointers, p1 and p2, that are used to track positions within each list. When the elements at both pointers are equal, the element is added to the result list, and both pointers advance. If elements differ, the smaller one is added to the result, and its pointer advances. This process repeats until one list is exhausted, after which the remaining elements from the other list are appended to the result.

2.2. Write out a postings merge algorithm, in the style of Figure 1.6 in IIR, for an x AND NOT y query.

```
AND_NOT(p1, p2)
answer <- ( )
while p1 ≠ NIL and p2 ≠ NIL do
    if docID(p1) = docID(p2) then
        p1 <- next(p1)
        p2 <- next(p2)
    else if docID(p1) < docID(p2) then
        ADD(answer, docID(p1))
        p1 <- next(p1)
    else
        p2 <- next(p2)
while p1 ≠ NIL do
    ADD(answer, docID(p1))
    p1 <- next(p1)
return answer
```

Explanation: It has two pointers, p1 and p2, that are used to track positions within each list. When the elements at both pointers are equal, the element is skipped, and both pointers advance. If elements differ, if the smaller one is from the first list, it is added to the result, and its pointer advances. If the smaller one is from the excluded list, it is skipped, since we are interested in elements that are not in that list. This process repeats until one list is exhausted, after which the remaining elements from the first list are appended to the result.

Problem 3

Recommend a query processing order for:
(tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes) given the following postings list sizes:

Term	**Postings size**
eyes	**213312**
kaleidoscope	**46653**
marmalade	**107913**
skies	**271658**
tangerine	**87009**
trees	**316812**

Solution: Get frequencies for all terms, estimate the size of each or by the sum of its frequencies (conservative) and process in increasing order of or sizes.

First, let's add up the postings sizes for the "OR" pairs, where

N(x) = postings size for term x

<u>(tangerine OR trees)</u>: N(tangerine) + N(trees) = 87009 + 316812 = 403,821

<u>(marmalade OR skies)</u>: N(marmalade) + N(skies) = 107913 + 271658 = 379,571

<u>(kaleidoscope OR eyes)</u>: N(kaleidoscope) + N(eyes) = 46653 + 213312 = 259,965

Using the conservative estimate of the length of unioned postings lists, the recommended order is:

1. kaleidoscope OR eyes (1)
2. marmalade OR skies (2)
3. tangerine OR trees (3)

The merging process should start from right to left:

1. intermediateResult ← (1) AND (2)
2. finalResult ← intermediateResult AND (3)

<u>Problem 4</u>

How should the Boolean query x OR NOT y be handled? Why is the naive evaluation of this query normally very expensive? Write out a postings merge algorithm that evaluates this query efficiently. Hints:
• You may want to use one of de Morgan's laws (https://mathworld.wolfram.com/deMorgansLaws.html for the latter part: NOT (x OR y) = (NOT x) AND (NOT y).
• You have access to a pointer to the beginning of the complete list of documents.
• You may reuse the algorithm from problem 2. If you do so, you do not need to rewrite the whole algorithm from problem 2 here. Assume you have a procedure that implements it, and just call the procedure in your pseudocode.

The naive evaluation of this query is normally costly because it involves negation which requires knowledge of the complete set of documents in the collection. In practical cases, the number of documents that do not contain a given term is huge, meaning that it could be close to the size of the entire collection of documents. This can be very <u>time-consuming</u> and <u>memory-intensive</u>. Inverted indexes are not designed to find documents that do not contain a specific term.
A naive evaluation of the query would be calculating NOT y first as a new postings list (inversed indexed list), then merging it with x. This is a UNEFFICIENT algorithm:

```
OR_NOT(p_x, p_y, p_all)
  not_y ← AND_NOT(p_all, p_y) //from problem 2
  result ← OR(p_x, not_y) //from problem 2
  return result
```

<u>Solution with the efficient approach:</u>

```
OR_NOT(pX, pY, pDocs)
  answer ← ()
  while pDocs ≠ NIL do
     if pX ≠ NIL and docID(pX) = docID(pDocs) then
        ADD(answer, docID(pX))
        pX ← next(pX)
     else if pY = NIL or docID(pY) ≠ docID(pDocs) then
        ADD(answer, docID(pDocs))
     if pY ≠ NIL and docID(pY) = docID(pDocs) then
        pY ← next(pY)
     pDocs ← next(pDocs)
  return answer
```

Explanation: This is the pseudocode for merging two lists. It has three pointers: pX and pY, that are used to track positions within each list, and pDocs, which is used to track the position of the list of all documents. If the current document is present in pX’s list, it is added to the final result and px is incremented. If the current document is not in pY’s list or pY’s list is finished, it is added to the final result. Independently of the previous conditions, if the current document is present in pY’s list, the pointer pY is advanced, without the document being added since it doesn’t satisfy the NOT y condition. After each document is processed, the pointer pDocs is incremented.

Using this approach, we do not have to calculate the negation anymore, and we calculate the OR NOT in one go.

<u>Problem 5</u>

Assume a biword index. Give an example of a document (could be a made up paragraph) which will be returned for a query of “Technical University of Barcelona” but is actually a false positive which should not be returned.

In a biword index, each pair of consecutive terms forms a biword or a phrase. So the query "Technical University of Barcelona" would be indexed as "Technical University", "University of", and "of Barcelona". A document that contains these biwords could be considered a match for the query. However, if these biwords are part of different contexts within the document, it could be a false positive.

The students from the Arizona **Technical University** and from the **University of** Magic gathered at the bus stop to wait for the bus that will take them to the magic book **of Barcelona.**

Problem 6

Write down the entries in the permuterm index dictionary that are generated by the term “pandemic”.

1. pandemic$
2. andemic$p
3. ndemic$pa
4. demic$pan
5. emic$pand
6. mic$pande
7. ic$pandem
8. c$pandemi
9. $pandemic

Problem 7

Compute the edit distance between “london” and “donut”. What are the N (rows) and M (columns) dimensions of the edit distance matrix? Write down the N × M array of distances between all prefixes as computed by the edit distance algorithm in Figure 3.5 in IIR. For each cell in the matrix, use the four-number representation to keep track of your intermediate results.

Given two character strings s1 and s2, the edit distance between them is the minimum number of edit operations required to transform s1 in s2.

edit distance = 4 (the proof is the edit distance matrix).

- **d**ondon: replace l with d at position 1
- d**o**ndon: don’t change o at position 2
- do**n**don: don’t change n at position 3
- don**u**on: replace d with u at position 4
- donu**t**n: replace o with t at position 5
- donu**t**: delete n at position 6

The matrix is 7 x 6 where N = 7 = nr of rows, M = 6 = nr of columns

		d	o	n	u	t
	0	**1 1**	**2 2**	**3 3**	**4 4**	**5 5**
l	**1** **1**	**1** 2 2 **1**	**2** 3 **2 2**	**3** 4 **3 3**	**4** 5 **4 4**	**5** 6 **5 5**
o	**2** **2**	**2 2** 3 **2**	**1** 3 3 **1**	3 4 **2 2**	4 5 **3 3**	5 6 **4 4**
n	**3** **3**	**3 3** 4 **3**	3 **2** 4 **2**	**1** 3 3 **1**	3 4 **2 2**	4 5 **3 3**
d	**4** **4**	**3** 4 5 **3**	4 **3** 4 **3**	3 **2** 4 **2**	**2** 3 3 **2**	**3** 4 **3 3**
o	**5** **5**	5 **4** 6 **4**	**3** 4 5 **3**	4 **3** 4 **3**	**3 3** 4 **3**	**3** 4 4 **3**
n	**6** **6**	6 **5** 7 **5**	5 **4** 6 **4**	**3** 4 5 **3**	**4 4** **4 4**	**4 4** 5 **4**

Problem 8

Consider the standard edit distance algorithm below:

```
LevenshteinDistance(s1, s2)
1 for i ← 0 to |s1|
2 do m[i, 0] = i
3 for j ← 0 to |s2|
4 do m[0, j] = j
5 for i ← 1 to |s1|
6 do for j ← 1 to |s2|
7         do m[i, j] = min{
8                 m[i − 1, j − 1] + if (s1[i] == s2[j]) then 0 else 1 fi,
9                 m[i − 1, j] + 1,
10                m[i, j − 1] + 1
11        }
12 return m[|s1|, |s2|]
```

Apple Inc. hired you to change this algorithm so it works on the new iPhone.

That is, the cost of replacing one character with another is no longer 1, but it is a function of the position of the two characters on the iPhone keyboard: if the two characters are adjacent, then the cost is 0.5; otherwise the cost is 1.

Suppose you are given a procedure: adjacent(c1,c2), which returns True if the characters are adjacent on the keyboard, and False otherwise.

Write below ONLY the changed lines of pseudocode to support this functionality.

To adapt the standard Levenshtein Distance algorithm to handle the new cost conditions based on the adjacency of characters on an iPhone keyboard, it should be modified the part of the algorithm that calculates the cost of substitution. In the algorithm, this is what it is done:

m[i, j] = min(replace/copy, delete, insert)

So line 8 is the cost of replace/copy, line 9 of delete and line 10 of insert.

Line 8 takes care of replacing the character. Line 8 should be changed to:

```
m[i - 1, j - 1] + if (s1[i] == s2[j]) then 0 else if adjacent(s1[i], s2[j]) then 0.5 else 1 fi,
```