17.26 Sparse Similarity: The similarity of two documents (each with distinct words) is defined to be the size of the intersection divided by the size of the union. For example, if the documents consist of integers, the similarity of `{1, 5, 3}` and `{1, 7, 2, 3}` is `0.4`, because the intersection has size 2 and the union has size 5.

We have a long list of documents (with distinct values and each with an associated ID) where the similarity is believed to be "sparse." That is, any two arbitrarily selected documents are very likely to have similarity 0. Design an algorithm that returns a list of pairs of document IDs and the associated similarity.

Print only the pairs with similarity greater than 0. Empty documents should not be printed at all. For simplicity, you may assume each document is represented as an array of distinct integers.

EXAMPLE

Input:

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10}
```

Output:

```
ID1, ID2 : SIMILARITY
13, 19   : 0.1
13, 16   : 0.25
19, 24   : 0.14285714285714285
```

pg 190

SOLUTION

This sounds like quite a tricky problem, so let's start off with a brute force algorithm. If nothing else, it will help wrap our heads around the problem.

Remember that each document is an array of distinct "words", and each is just an integer.

Brute Force

A brute force algorithm is as simple as just comparing all arrays to all other arrays. At each comparison, we compute the size of the intersection and size of the union of the two arrays.

Note that we only want to print this pair if the similarity is greater than 0. The union of two arrays can never be zero (unless both arrays are empty, in which case we don't want them printed anyway). Therefore, we are really just printing the similarity if the intersection is greater than 0.

How do we compute the size of the intersection and the union?

The intersection means the number of elements in common. Therefore, we can just iterate through the first array (A) and check if each element is in the second array (B). If it is, increment an `intersection` variable.

To compute the union, we need to be sure that we don't double count elements that are in both. One way to do this is to count up all the elements in A that are *not* in B. Then, add in all the elements in B. This will avoid double counting as the duplicate elements are only counted with B.

Alternatively, we can think about it this way. If we *did* double count elements, it would mean that elements in the intersection (in both A and B) were counted twice. Therefore, the easy fix is to just remove these duplicate elements.

```
union(A, B) = A + B - intersection(A, B)
```

This means that all we really need to do is compute the intersection. We can derive the union, and therefore similarity, from that immediately.

This gives us an $O(AB)$ algorithm, just to compare two arrays (or documents).

However, we need to do this for all pairs of D documents. If we assume each document has at most W words then the runtime is $O(D^2 W^2)$.

Slightly Better Brute Force

As a quick win, we can optimize the computation for the similarity of two arrays. Specifically, we need to optimize the intersection computation.

We need to know the number of elements in common between the two arrays. We can throw all of A's elements into a hash table. Then we iterate through B, incrementing `intersection` every time we find an element in A.

This takes $O(A + B)$ time. If each array has size W and we do this for D arrays, then this takes $O(D^2 W)$.

Before implementing this, let's first think about the classes we'll need.

We'll need to return a list of document pairs and their similarities. We'll use a `DocPair` class for this. The exact return type will be a hash table that maps from `DocPair` to a double representing the similarity.

```
1  public class DocPair {
2    public int doc1, doc2;
3
4    public DocPair(int d1, int d2) {
5      doc1 = d1;
6      doc2 = d2;
7    }
8
9    @Override
10   public boolean equals(Object o) {
11     if (o instanceof DocPair) {
12       DocPair p = (DocPair) o;
13       return p.doc1 == doc1 && p.doc2 == doc2;
14     }
15     return false;
16   }
17
18   @Override
19   public int hashCode() { return (doc1 * 31) ^ doc2; }
20 }
```

It will also be useful to have a class that represents the documents.

```
1  public class Document {
2    private ArrayList<Integer> words;
3    private int docId;
4
5    public Document(int id, ArrayList<Integer> w) {
6      docId = id;
7      words = w;
8    }
9
10   public ArrayList<Integer> getWords() { return words; }
11   public int getId() { return docId; }
```

```
12    public int size() { return words == null ? 0 : words.size(); }
13 }
```

Strictly speaking, we don't need any of this. However, readability is important, and it's a lot easier to read `ArrayList<Document>` than `ArrayList<ArrayList<Integer>>`.

Doing this sort of thing not only shows good coding style, it also makes your life in an interview a lot easier. You have to write a lot less. (You probably would not define the entire `Document` class, unless you had extra time or your interviewer asked you to.)

```
1  HashMap<DocPair, Double> computeSimilarities(ArrayList<Document> documents) {
2    HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
3    for (int i = 0; i < documents.size(); i++) {
4      for (int j = i + 1; j < documents.size(); j++) {
5        Document doc1 = documents.get(i);
6        Document doc2 = documents.get(j);
7        double sim = computeSimilarity(doc1, doc2);
8        if (sim > 0) {
9          DocPair pair = new DocPair(doc1.getId(), doc2.getId());
10         similarities.put(pair, sim);
11       }
12     }
13   }
14   return similarities;
15 }
16
17 double computeSimilarity(Document doc1, Document doc2) {
18   int intersection = 0;
19   HashSet<Integer> set1 = new HashSet<Integer>();
20   set1.addAll(doc1.getWords());
21
22   for (int word : doc2.getWords()) {
23     if (set1.contains(word)) {
24       intersection++;
25     }
26   }
27
28   double union = doc1.size() + doc2.size() - intersection;
29   return intersection / union;
30 }
```

Observe what's happening on line 28. Why did we make `union` a `double`, when it's obviously an `integer`?

We did this to avoid an integer division bug. If we didn't do this, the division would "round" down to an integer. This would mean that the similarity would almost always return 0. Oops!

Slightly Better Brute Force (Alternate)

If the documents were sorted, you could compute the intersection between two documents by walking through them in sorted order, much like you would when doing a sorted merge of two arrays.

This would take $O(A + B)$ time. This is the same time as our current algorithm, but less space. Doing this on D documents with W words each would take $O(D^2 W)$ time.

Since we don't know that the arrays are sorted, we could first sort them. This would take $O(D * W \log W)$ time. The full runtime then is $O(D * W \log W + D^2 W)$.

We cannot necessarily assume that the second part "dominates" the first one, because it doesn't necessarily. It depends on the relative size of D and $\log W$. Therefore, we need to keep both terms in our runtime expression.

Optimized (Somewhat)

It is useful to create a larger example to really understand the problem.

```
13: {14, 15, 100, 9, 3}
16: {32, 1, 9, 3, 5}
19: {15, 29, 2, 6, 8, 7}
24: {7, 10, 3}
```

At first, we might try various techniques that allow us to more quickly eliminate potential comparisons. For example, could we compute the min and max values in each array? If we did that, then we'd know that arrays with no overlap in ranges don't need to be compared.

The problem is that this doesn't really fix our runtime issue. Our best runtime thus far is $O(D^2 W)$. With this change, we're still going to be comparing all $O(D^2)$ pairs, but the $O(W)$ part might go to $O(1)$ sometimes. That $O(D^2)$ part is going to be a really big problem when D gets large.

Therefore, let's focus on reducing that $O(D^2)$ factor. That is the "bottleneck" in our solution. Specifically, this means that, given a document docA, we want to find all documents with some similarity—and we want to do this without "talking" to each document.

What would make a document similar to docA? That is, what characteristics define the documents with similarity > 0?

Suppose docA is {14, 15, 100, 9, 3}. For a document to have similarity > 0, it needs to have a 14, a 15, a 100, a 9, or a 3. How can we quickly gather a list of all documents with one of those elements?

The slow (and, really, only way) is to read every single word from every single document to find the documents that contain a 14, a 15, a 100, a 9, or a 3. That will take $O(DW)$ time. Not good.

However, note that we're doing this repeatedly. We can reuse the work from one call to the next.

If we build a hash table that maps from a word to all documents that contain that word, we can very quickly know the documents that overlap with docA.

```
1 -> 16
2 -> 19
3 -> 13, 16, 24
5 -> 16
6 -> 19
7 -> 19, 24
8 -> 19
9 -> 13, 16
...
```

When we want to know all the documents that overlap with docA, we just look up each of docA's items in this hash table. We'll then get a list of all documents with some overlap. Now, all we have to do is compare docA to each of those documents.

If there are P pairs with similarity > 0, and each document has W words, then this will take $O(PW)$ time (plus $O(DW)$ time to create and read this hash table). Since we expect P to be much less than D^2, this is much better than before.

Optimized (Better)

Let's think about our previous algorithm. Is there any way we can make it more optimal?

If we consider the runtime—O(PW + DW)—we probably can't get rid of the O(DW) factor. We have to touch each word at least once, and there are O(DW) words. Therefore, if there's an optimization to be made, it's probably in the O(PW) term.

It would be difficult to eliminate the P part in O(PW) because we have to at least print all P pairs (which takes O(P) time). The best place to focus, then, is on the W part. Is there some way we can do less than O(W) work for each pair of similar documents?

One way to tackle this is to analyze what information the hash table gives us. Consider this list of documents:

```
12: {1, 5, 9}
13: {5, 3, 1, 8}
14: {4, 3, 2}
15: {1, 5, 9, 8}
17: {1, 6}
```

If we look up document 12's elements in a hash table for this document, we'll get:

```
1 -> {12, 13, 15, 17}
5 -> {12, 13, 15}
9 -> {12, 15}
```

This tells us that documents 13, 15, and 17 have some similarity. Under our current algorithm, we would now need to compare document 12 to documents 13, 15, and 17 to see the number of elements document 12 has in common with each (that is, the size of the intersection). The union can be computed from the document sizes and the intersection, as we did before.

Observe, though, that document 13 appeared twice in the hash table, document 15 appeared three times, and document 17 appeared once. We discarded that information. But can we use it instead? What does it indicate that some documents appeared multiple times and others didn't?

Document 13 appeared twice because it has two elements (1 and 5) in common. Document 17 appeared once because it has only one element (1) in common. Document 15 appeared three times because it has three elements (1, 5, and 9) in common. This information can actually directly give us the size of the intersection.

We could go through each document, look up the items in the hash table, and then count how many times each document appears in each item's lists. There's a more direct way to do it.

1. As before, build a hash table for a list of documents.
2. Create a new hash table that maps from a document pair to an integer (which will indicate the size of the intersection).
3. Read the first hash table by iterating through each list of documents.
4. For each list of documents, iterate through the pairs in that list. Increment the intersection count for each pair.

Comparing this runtime to the previous one is a bit tricky. One way we can look at it is to realize that before we were doing O(W) work for each similar pair. That's because once we noticed that two documents were similar, we touched every single word in each document. With this algorithm, we're only touching the words that actually overlap. The worst cases are still the same, but for many inputs this algorithm will be faster.

```
1   HashMap<DocPair, Double>
2   computeSimilarities(HashMap<Integer, Document> documents) {
```

```
  HashMapList<Integer, Integer> wordToDocs = groupWords(documents);
  HashMap<DocPair, Double> similarities = computeIntersections(wordToDocs);
  adjustToSimilarities(documents, similarities);
  return similarities;
}

/* Create hash table from each word to where it appears. */
HashMapList<Integer, Integer> groupWords(HashMap<Integer, Document> documents) {
  HashMapList<Integer, Integer> wordToDocs = new HashMapList<Integer, Integer>();

  for (Document doc : documents.values()) {
    ArrayList<Integer> words = doc.getWords();
    for (int word : words) {
      wordToDocs.put(word, doc.getId());
    }
  }

  return wordToDocs;
}

/* Compute intersections of documents. Iterate through each list of documents and
 * then each pair within that list, incrementing the intersection of each page. */
HashMap<DocPair, Double> computeIntersections(
    HashMapList<Integer, Integer> wordToDocs {
  HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
  Set<Integer> words = wordToDocs.keySet();
  for (int word : words) {
    ArrayList<Integer> docs = wordToDocs.get(word);
    Collections.sort(docs);
    for (int i = 0; i < docs.size(); i++) {
      for (int j = i + 1; j < docs.size(); j++) {
        increment(similarities, docs.get(i), docs.get(j));
      }
    }
  }

  return similarities;
}

/* Increment the intersection size of each document pair. */
void increment(HashMap<DocPair, Double> similarities, int doc1, int doc2) {
  DocPair pair = new DocPair(doc1, doc2);
  if (!similarities.containsKey(pair)) {
    similarities.put(pair, 1.0);
  } else {
    similarities.put(pair, similarities.get(pair) + 1);
  }
}

/* Adjust the intersection value to become the similarity. */
void adjustToSimilarities(HashMap<Integer, Document> documents,
                          HashMap<DocPair, Double> similarities) {
  for (Entry<DocPair, Double> entry : similarities.entrySet()) {
    DocPair pair = entry.getKey();
    Double intersection = entry.getValue();
    Document doc1 = documents.get(pair.doc1);
```

```
59        Document doc2 = documents.get(pair.doc2);
60        double union = (double) doc1.size() + doc2.size() - intersection;
61        entry.setValue(intersection / union);
62     }
63  }
64
65  /* HashMapList<Integer, Integer> is a HashMap that maps from Integer to
66   * ArrayList<Integer>. See appendix for implementation. */
```

For a set of documents with sparse similarity, this will run much faster than the original naive algorithm, which compares all pairs of documents directly.

Optimized (Alternative)

There's an alternative algorithm that some candidates might come up with. It's slightly slower, but still quite good.

Recall our earlier algorithm that computed the similarity between two documents by sorting them. We can extend this approach to multiple documents.

Imagine we took all of the words, tagged them by their original document, and then sorted them. The prior list of documents would look like this:

1_{12}, 1_{13}, 1_{15}, 1_{16}, 2_{14}, 3_{13}, 3_{14}, 4_{14}, 5_{12}, 5_{13}, 5_{15}, 6_{16}, 8_{13}, 8_{15}, 9_{12}, 9_{15}

Now we have essentially the same approach as before. We iterate through this list of elements. For each sequence of identical elements, we increment the intersection counts for the corresponding pair of documents.

We will use an Element class to group together documents and words. When we sort the list, we will sort first on the word but break ties on the document ID.

```
1   class Element implements Comparable<Element> {
2     public int word, document;
3     public Element(int w, int d) {
4       word = w;
5       document = d;
6     }
7
8     /* When we sort the words, this function will be used to compare the words. */
9     public int compareTo(Element e) {
10      if (word == e.word) {
11        return document - e.document;
12      }
13      return word - e.word;
14    }
15  }
16
17  HashMap<DocPair, Double> computeSimilarities(
18        HashMap<Integer, Document> documents) {
19    ArrayList<Element> elements = sortWords(documents);
20    HashMap<DocPair, Double> similarities = computeIntersections(elements);
21    adjustToSimilarities(documents, similarities);
22    return similarities;
23  }
24
25  /* Throw all words into one list, sorting by the word and then the document. */
26  ArrayList<Element> sortWords(HashMap<Integer, Document> docs) {
27    ArrayList<Element> elements = new ArrayList<Element>();
```

```
28    for (Document doc : docs.values()) {
29       ArrayList<Integer> words = doc.getWords();
30       for (int word : words) {
31          elements.add(new Element(word, doc.getId()));
32       }
33    }
34    Collections.sort(elements);
35    return elements;
36 }
37
38 /* Increment the intersection size of each document pair. */
39 void increment(HashMap<DocPair, Double> similarities, int doc1, int doc2) {
40    DocPair pair = new DocPair(doc1, doc2);
41    if (!similarities.containsKey(pair)) {
42       similarities.put(pair, 1.0);
43    } else {
44       similarities.put(pair, similarities.get(pair) + 1);
45    }
46 }
47
48 /* Adjust the intersection value to become the similarity. */
49 HashMap<DocPair, Double> computeIntersections(ArrayList<Element> elements) {
50    HashMap<DocPair, Double> similarities = new HashMap<DocPair, Double>();
51
52    for (int i = 0; i < elements.size(); i++) {
53       Element left = elements.get(i);
54       for (int j = i + 1; j < elements.size(); j++) {
55          Element right = elements.get(j);
56          if (left.word != right.word) {
57             break;
58          }
59          increment(similarities, left.document, right.document);
60       }
61    }
62    return similarities;
63 }
64
65 /* Adjust the intersection value to become the similarity. *
66 void adjustToSimilarities(HashMap<Integer, Document> documents,
67                           HashMap<DocPair, Double> similarities) {
68    for (Entry<DocPair, Double> entry : similarities.entrySet()) {
69       DocPair pair = entry.getKey();
70       Double intersection = entry.getValue();
71       Document doc1 = documents.get(pair.doc1);
72       Document doc2 = documents.get(pair.doc2);
73       double union = (double) doc1.size() + doc2.size() - intersection;
74       entry.setValue(intersection / union);
75    }
76 }
```

The first step of this algorithm is slower than that of the prior algorithm, since it has to sort rather than just add to a list. The second step is essentially equivalent.

Both will run much faster than the original naive algorithm.

Advanced Topics

XI

This section includes topics that are mostly beyond the scope of interviews but can come up on occasion. Interviewers shouldn't be surprised if you don't know these topics well. Feel free to dive into these topics if you want to. If you're pressed for time, they're low priority.

XI

Advanced Topics

When writing the 6th edition, I had a number of debates about what should and shouldn't be included. Red-black trees? Dijkstra's algorithm? Topological sort?

On one hand, I'd had a number of requests to include these topics. Some people insisted that these topics are asked "all the time" (in which case, they have a very different idea of what this phrase means!). There was clearly a desire—at least from some people—to include them. And learning more can't hurt, right?

On the other hand, I know these topics to be rarely asked. It happens, of course. Interviewers are individuals and might have their own ideas of what is "fair game" or "relevant" for an interview. But it's rare. When it does come up, if you don't know the topic, it's unlikely to be a big red flag.

> Admittedly, as an interviewer, I *have* asked candidates questions where the solution was essentially an application of one of these algorithms. On the rare occasions that a candidate already knew the algorithm, they did not benefit from this knowledge (nor were they hurt by it). I want to evaluate your ability to solve a problem you haven't seen before. So, I'll take into account whether you know the underlying algorithm in advance.

I believe in giving people a fair expectation of the interview, not scaring people into excess studying. I also have no interest in making the book more "advanced" so as to help book sales, at the expense of your time and energy. That's not fair or right to do to you.

(Additionally, I didn't want to give interviewers—who I know to be reading this—the impression that they can or should be covering these more advanced topics. Interviewers: If you ask about these topics, you're testing knowledge of algorithms. You're just going to wind up eliminating a lot of perfectly smart people.)

But there are many borderline "important" topics. They're not often asked, but sometimes they are.

Ultimately, I decided to leave the decision in your hands. After all, you know better than I do how thorough you want to be in your preparation. If you want to do an extra thorough job, read this. If you just love learning data structures and algorithms, read this. If you want to see new ways of approaching problems, read this.

But if you're pressed for time, this studying isn't a super high priority.

Useful Math

Here's some math that can be useful in some questions. There are more formal proofs that you can look up online, but we'll focus here on giving you the intuition behind them. You can think of these as informal proofs.