**Time Complexity:**

1. **Reading PDF**:
   * Reading each page: O(n) where n is the number of pages.
   * Extracting text: Assuming it's proportional to the text length, let’s keep this as O(T).
2. **Tokenizing Text**:
   * Splitting the text: O(T), where T is the length of the text.
3. **Matching Values**:
   * Creating a map of tokens: O(m) where m is the number of words in the text.
   * Matching tokens with reference list: O(r) where r is the length of the reference list. Comparing each token: O(m∗r).

Overall, combining these, the total time complexity can be approximated as: \[O(n) + O(T) + O(m) + O(m \* r)\]

**Space Complexity:**

1. **PDF Content Storage**:
   * Text storage: O(T).
2. **Tokenization**:
   * Storing tokens: O(m).
3. **Matched Words**:
   * Storage for matched words map: O(r).

Combining these, the total space complexity is approximately: \[O(T) + O(m) + O(r)\]

**Optimization Suggestions:**

1. **Efficient Data Structures**: Use hash sets for matching values to bring down the comparison to O(1) per word.
2. **Lazy Processing**: Process one page at a time if the PDF is very large instead of loading everything into memory.

If you can handle this with your current resources, great! Otherwise, these tips should help streamline things. Let me know if you need more insights or improvements!

Switching to storing the PDF as a ByteStream could definitely help with memory usage, especially for large PDFs. Here’s how:

**Memory Usage:**

* **String Type**: Storing text as a string can consume a significant amount of memory, especially with large documents.
* **ByteStream**: Storing data as a ByteStream is more efficient as it processes raw binary data, reducing the overhead.

**Implementation:**

To optimize memory usage by handling PDFs as ByteStreams, you can use PDFBox's InputStream:

import org.apache.pdfbox.pdmodel.PDDocument;

import org.apache.pdfbox.text.PDFTextStripper;

import java.io.\*;

import java.util.\*;

public class PDFTextMatcher {

public static String readPDF(String filePath) throws IOException {

try (InputStream inputStream = new FileInputStream(new File(filePath))) {

PDDocument document = PDDocument.load(inputStream);

PDFTextStripper pdfStripper = new PDFTextStripper();

String text = pdfStripper.getText(document);

document.close();

return text;

}

}

// ... (other methods remain unchanged)

public static void main(String[] args) {

try {

// Static values

String filePath = "BrianMason\_Using\_Rest\_API-2.pdf";

List<String> referenceList = Arrays.asList("SOAP", "Swagger", "Java");

Map<String, Integer> matchedWords = new HashMap<>();

String text = readPDF(filePath);

Map<String, Integer> tokenCountMap = tokenizeText(text);

int matches = matchValues(tokenCountMap, referenceList, matchedWords);

double matchPercentage = calculateMatchPercentage(matches, referenceList.size());

outputResult(matchedWords, matchPercentage);

} catch (IOException e) {

e.printStackTrace();

}

}

}  
**Benefits:**

* **Lower Memory Footprint**: By processing the file in chunks rather than loading the entire text into memory.
* **Efficiency**: Handling raw bytes can be faster, especially for I/O operations.

This approach helps in managing large PDFs more effectively. Give it a try and see the improvements! Anything else on your mind?

You can  check if a page is over and get the count of pages in a PDF using PDFBox, regardless of whether you're using a String or ByteStream approach.

Here's how you can do it:

**Using String:**

import org.apache.pdfbox.pdmodel.PDDocument;

public class PDFPageCounter {

public static void main(String[] args) {

try {

String filePath = "BrianMason\_Using\_Rest\_API-2.pdf";

PDDocument document = PDDocument.load(new File(filePath));

// Get the number of pages

int pageCount = document.getNumberOfPages();

System.out.println("Total pages: " + pageCount);

document.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Using ByteStream:**

When using a ByteStream, the process is similar, just with an InputStream:

import org.apache.pdfbox.pdmodel.PDDocument;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

public class PDFPageCounter {

public static void main(String[] args) {

try (InputStream inputStream = new FileInputStream("BrianMason\_Using\_Rest\_API-2.pdf")) {

PDDocument document = PDDocument.load(inputStream);

// Get the number of pages

int pageCount = document.getNumberOfPages();

System.out.println("Total pages: " + pageCount);

document.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

A simple way:-

Removing the method to show the individual count of occurrences will slightly reduce both time and space complexity, but let's quantify the impact:

**Time Complexity**

1. **Reading PDF**: O(n)+O(T)
2. **Tokenizing Text**: O(T)
3. **Matching Values**: Without the count, it's O(r) comparisons for m tokens.

Without tracking occurrences: \[O(n) + O(T) + O(T) + O(m \* r)\]

With tracking occurrences: \[O(n) + O(T) + O(T) + O(m \* r)\]

**Space Complexity**

1. **PDF Content Storage**: O(T)
2. **Tokenization**: O(m)
3. **Matched Words**: Without count, it's O(r). With count, it's O(r).

Without tracking occurrences: \[O(T) + O(m) + O(r)\]

With tracking occurrences: \[O(T) + O(m) + O(r + 1)\]

Overall, the main complexity contributor remains the same—processing the text and matching values. Tracking the individual counts adds a bit to space complexity (storing counts) and to time complexity (counting occurrences), but it’s not significantly changing the overall order of magnitude.

If your focus is strictly on reducing complexity, the optimizations we discussed (efficient data structures, lazy processing) would have a more noticeable impact.

import org.apache.pdfbox.pdmodel.PDDocument;

import org.apache.pdfbox.text.PDFTextStripper;

import java.io.File;

import java.io.IOException;

import java.util.\*;

public class PDFTextMatcher {

public static String readPDF(String filePath) throws IOException {

PDDocument document = PDDocument.load(new File(filePath));

PDFTextStripper pdfStripper = new PDFTextStripper();

String text = pdfStripper.getText(document);

document.close();

return text;

}

public static Set<String> tokenizeText(String text) {

String[] tokens = text.split("\\W+");

return new HashSet<>(Arrays.asList(tokens));

}

public static int matchValues(Set<String> tokens, List<String> referenceList, List<String> matchedWords) {

int matches = 0;

for (String word : referenceList) {

if (tokens.contains(word)) {

matches++;

matchedWords.add(word);

}

}

return matches;

}

public static double calculateMatchPercentage(int matches, int referenceListLength) {

return ((double) matches / referenceListLength) \* 100;

}

public static void outputResult(double matchPercentage, List<String> matchedWords) {

System.out.printf("Match Percentage: %.2f%%%n", matchPercentage);

System.out.println("Matched Words: " + matchedWords);

}

public static void main(String[] args) {

try {

// Static values

String filePath = "BrianMason\_Using\_Rest\_API-2.pdf";

List<String> referenceList = Arrays.asList("SOAP", "Swagger", "Java");

List<String> matchedWords = new ArrayList<>();

String text = readPDF(filePath);

Set<String> tokens = tokenizeText(text);

int matches = matchValues(tokens, referenceList, matchedWords);

double matchPercentage = calculateMatchPercentage(matches, referenceList.size());

outputResult(matchPercentage, matchedWords);

} catch (IOException e) {

e.printStackTrace();

}

}

}

Using **efficient data structures** combined with **lazy processing with ByteStream** will likely be the more efficient approach. Here’s why:

**Efficient Data Structures**

Using data structures like HashSet for token storage and comparison ensures that your look-up operations are O(1).

**Lazy Processing**

Handling data in chunks (lazy processing) reduces memory footprint since you won’t load the entire file into memory at once.

**ByteStream**

Processing the file as a ByteStream avoids the overhead of converting the entire PDF content to a single string, reducing both memory usage and potentially processing time.

Combining these means you’ll be optimizing both the memory usage and the processing time effectively. Implementing this method should give you a more scalable and performant solution.

 Let's compare the two approaches using Big O notation to clearly see the differences in their time and space complexity.

**Approach 1: Efficient Data Structures + Lazy Processing with String**

**Time Complexity:**

1. **Reading PDF**: O(n)+O(T)
2. **Tokenizing Text**: O(T)
3. **Matching Values**: O(m∗r)

Total Time Complexity: \[O(n) + O(T) + O(T) + O(m \* r) \approx O(n + T + m \cdot r)\]

**Space Complexity:**

1. **PDF Content Storage**: O(T)
2. **Tokenization**: O(m)
3. **Matched Words Storage**: O(r)

Total Space Complexity: \[O(T + m + r)\]

**Approach 2: Efficient Data Structures + Lazy Processing with ByteStream**

**Time Complexity:**

1. **Reading PDF**: O(n)
2. **Tokenizing Text**: O(T)
3. **Matching Values**: O(m∗r)

Total Time Complexity: \[O(n) + O(T) + O(m \cdot r) \approx O(n + T + m \cdot r)\]

**Space Complexity:**

1. **ByteStream Handling**: O(n) (Reading the file in chunks reduces the in-memory text storage)
2. **Tokenization**: O(m)
3. **Matched Words Storage**: O(r)

Total Space Complexity: \[O(n + m + r)\]

**Summary:**

* **Time Complexity** for both approaches is quite similar: O(n+T+m⋅r).
* **Space Complexity** is where the major difference lies:
  + With ByteStream, you significantly reduce the memory footprint from storing the entire text (O(T)) to just handling the bytes (O(n)).

**Summary:**

* **Time Complexity** for both approaches is quite similar: O(n+T+m⋅r).
* **Space Complexity** is where the major difference lies:
  + With ByteStream, you significantly reduce the memory footprint from storing the entire text (O(T)) to just handling the bytes (O(n)).

By using ByteStream for lazy processing, you handle less data in memory at a time, which can significantly reduce the space complexity, especially for large PDF files. This approach helps manage large datasets more efficiently, ensuring you don’t hit memory limitations.

**Apache PDFBox** is an open-source Java library for working with PDF documents1. It allows you to create, manipulate, and extract content from PDF files1. Here are some key features:

* **Create PDFs**: You can create new PDF documents from scratch, including embedding fonts and images1.
* **Extract Text**: Extract text from PDF files, including Unicode text1.
* **Split & Merge**: Split a single PDF into multiple files or merge multiple PDF files1.
* **Fill Forms**: Extract data from PDF forms or fill out PDF forms1.
* **Preflight**: Validate PDF files against the PDF/A-1b standard1.
* **Print**: Print PDF files using the standard Java printing API1.
* **Save as Image**: Save PDFs as image files, such as PNG or JPEG1.
* **Sign PDFs**: Digitally sign PDF files1.

Apache PDFBox is widely used for its flexibility and ease of integration into Java applications1. It's a great tool for handling PDF documents programmatically.

[Apache PDFBox | A Java PDF Library](https://pdfbox.apache.org/?form=MG0AV3)