Section 1: Document Loaders (Text, PDF, CSV, DOCX, Notion, URLs)

1. Load a `.txt` file using `TextLoader` from LangChain. Display the first 500 characters of the content and show the metadata structure.

2. Use `CSVLoader` from LangChain to read a `.csv` file. Assume the column containing main content is called `"description"`. Extract each row as a document and print the number of documents.

3. Using `UnstructuredWordDocumentLoader`, load a `.docx` document and inject custom metadata like file source, date, and page number.

4. Load and parse a Notion export file (`notion_export.json`) using LangChain's Notion loader. Extract all pages with tag `"final"` and print their titles.

5. Using LangChain or LlamaIndex, load content from a public website URL using `WebBaseLoader`. Preprocess the text by removing scripts and ads. Save the cleaned content to a local text file.

Section 2: Chunking Strategies (Recursive, Fixed, Semantic)

6. Load a long document using `TextLoader` and chunk it using `RecursiveCharacterTextSplitter` with chunk size 500 and overlap 100. Print the first 5 chunks and highlight the overlapping regions.

7. Implement chunking using `TokenTextSplitter` with a chunk size of 256 tokens and overlap of 50. Use any tokenizer like tiktoken or HuggingFace tokenizer. Print the total number of chunks generated.

8. Create semantic chunks using sentence-based segmentation, and then group every 3 consecutive sentences into one semantic chunk. Count how many chunks are created and analyze the average length.

9. Write a function to dynamically switch between recursive, token-based, and sentence-based chunking depending on the document type (e.g., `.txt` uses recursive, `.csv` uses token, `.docx` uses sentence).

10. Compare and report on the number of chunks generated, average token size, and processing time for each chunking strategy: fixed-size, recursive, and semantic. Use the same input document for all strategies.

Section 3: Metadata Injection and Filtering

11. Inject metadata like `"chunk_id"`, `"document_type"`, and `"source"` into every chunk generated from a `.pdf` file. Display the metadata of the first three chunks.

12. Create a filtering function that removes chunks shorter than 30 words or longer than 400 tokens. Apply it to any document split into chunks and show how many chunks are left after filtering.

Section 4: Text Splitters (RecursiveCharacterTextSplitter, TokenTextSplitter, SentenceSplitter)

13. Implement three different text splitting strategies using LangChain: `RecursiveCharacterTextSplitter`, `TokenTextSplitter`, and `NLTKTextSplitter`. For each, report:

- Number of chunks
- Average chunk length in tokens
- Total time taken

14. Build a custom splitter that uses `nltk.sent_tokenize` to split text into sentences, then groups 4 sentences per chunk with a one-sentence overlap. Print 3 example chunks and show overlaps.

Section 5: Ideal Chunk Sizes (Tokens vs Words)

15. Write a script to chunk a document using token sizes: 128, 256, 512, 1024 (using `TokenTextSplitter`). Analyze and report:

- Total number of chunks per size
- Average semantic coherence (manually review a few)

16. Create a visualization (bar chart) comparing word-based vs token-based chunking across different chunk sizes. Show chunk count and average size in both formats.

Section 6: Embeddings and Cosine Similarity (Using EURI and Others)

17. Use the `euriai` API to generate embeddings for 5 different chunks from a document. Print each vector and store them in a list.

18. Implement a manual cosine similarity function and compare its output with `sklearn.metrics.pairwise.cosine_similarity`. Verify both methods produce the same result for EURI embeddings.

19. Store all EURI-generated embeddings in a FAISS vector store. Implement a search query using LangChain that returns the top-3 most similar chunks and their similarity scores.

20. Compare EURI embeddings with OpenAI and HuggingFace embeddings on the same document. For each method:

- Generate embeddings
- Compute cosine similarity between 3 pairs of chunks
- Plot similarity score comparison as a heatmap