

lab__answer

April 3, 2026

1 Lesson 05 — Context Windows & Long Documents (ANSWERS)

Complete solutions with explanatory comments for all three exercises.

```
[ ]: import pathlib
import anthropic
import tiktoken
from dotenv import load_dotenv

load_dotenv()
client = anthropic.Anthropic()

text = pathlib.Path('demo/sample_document.txt').read_text(encoding='utf-8')
print(f'Document loaded: {len(text)} characters, {len(text.split())} words')
```

1.1 Exercise 1 — Chunking (Solution)

```
[ ]: def chunk_text(text: str, chunk_size: int = 500, overlap: int = 50) -> list[str]:
    """Split text into overlapping chunks of approximately chunk_size words.

    Args:
        text: The input text to split.
        chunk_size: Target number of words per chunk.
        overlap: Number of words shared between consecutive chunks.

    Returns:
        List of text chunks as strings.
    """
    words = text.split() # split on whitespace
    chunks = []
    stride = chunk_size - overlap # how far we advance between chunks

    # guard against degenerate stride
    if stride <= 0:
        stride = 1
```

```

start = 0
while start < len(words):
    end = start + chunk_size
    chunk_words = words[start:end]
    chunks.append(' '.join(chunk_words))
    start += stride
    # if this chunk was the last (didn't fill chunk_size), stop
    if end >= len(words):
        break

return chunks

chunks = chunk_text(text)
print(f"Created {len(chunks)} chunks (chunk_size=500, overlap=50)")
print(f"Words in first chunk: {len(chunks[0].split())}")
print(f"Words in last chunk: {len(chunks[-1].split())}")
print()
print("First chunk preview:")
print(chunks[0][:300], '...')
print()
# Verify overlap: last overlap words of chunk 0 should equal first overlap
# words of chunk 1
overlap = 50
tail_0 = ' '.join(chunks[0].split()[-overlap:])
head_1 = ' '.join(chunks[1].split()[:overlap])
print(f"Overlap verification (should match):")
print(f"End of chunk 0: ...{tail_0[-80:]}")
print(f"Start of chunk 1: {head_1[:80]}...")

```

1.2 Exercise 2 — Map-Reduce Summarisation (Solution)

```

[ ]: def summarize_chunk(chunk: str) -> str:
    """Summarise a single chunk in 2-3 sentences using Claude Haiku."""
    response = client.messages.create(
        model="claude-haiku-4-5",
        max_tokens=150,
        messages=[
            {
                "role": "user",
                "content": (
                    "Summarise the following passage in 2-3 sentences. "
                    "Be concise and factual.\n\n"
                    f"{chunk}"
                )
            }
        ]
    )

```

```

    ]
)
return response.content[0].text.strip()

def map_reduce_summarize(chunks: list[str]) -> str:
    """Summarise all chunks (map), then combine chunk summaries into a final
    ↪summary (reduce)."""
    print(f"Mapping {len(chunks)} chunks...")

    # MAP step: summarize each chunk independently
    chunk_summaries = []
    for i, chunk in enumerate(chunks):
        summary = summarize_chunk(chunk)
        chunk_summaries.append(summary)
        print(f"  Chunk {i+1}/{len(chunks)}: {summary[:80]}...")

    # REDUCE step: combine all chunk summaries into one final summary
    combined = "\n\n".join(
        f"Section {i+1}: {s}" for i, s in enumerate(chunk_summaries)
    )
    print("\nReducing to final summary...")

    response = client.messages.create(
        model="claude-haiku-4-5",
        max_tokens=400,
        messages=[
            {
                "role": "user",
                "content": (
                    "Below are section-by-section summaries of a long document.
                    ↪
                    "Write a single coherent summary of the whole document in
                    ↪4-6 sentences. "
                    "Do not mention the section structure - write as a unified
                    ↪paragraph.\n\n"
                    f"{combined}"
                )
            }
        ]
    )
    return response.content[0].text.strip()

final_summary = map_reduce_summarize(chunks)
print("\n=== FINAL SUMMARY ===")
print(final_summary)

```

1.3 Exercise 3 — Cost Comparison (Solution)

```
[ ]: enc = tiktoken.get_encoding("cl100k_base") # proxy tokenizer (Claude uses own
      ↪tokenizer)

# Pricing constants for claude-haiku-4-5 (check anthropic.com/pricing for
      ↪updates)
INPUT_COST_PER_M = 0.25 # $ per million input tokens
OUTPUT_COST_PER_M = 1.25 # $ per million output tokens

def count_tokens(text: str) -> int:
    """Approximate token count using tiktoken cl100k_base."""
    return len(enc.encode(text))

def estimate_cost(input_tokens: int, output_tokens: int) -> float:
    """Return estimated cost in USD."""
    return (
        input_tokens / 1_000_000 * INPUT_COST_PER_M +
        output_tokens / 1_000_000 * OUTPUT_COST_PER_M
    )

# SINGLE-PASS
single_prompt_overhead = "Summarise the following document in 5-6 sentences:
      ↪\n\n"
single_input_tokens = count_tokens(single_prompt_overhead + text)
single_output_tokens = 200 # estimated output length
single_cost = estimate_cost(single_input_tokens, single_output_tokens)

# MAP-REDUCE
# Each chunk call: ~chunk prompt overhead + chunk text + ~150 output tokens
chunk_prompt = "Summarise the following passage in 2-3 sentences. Be concise,
      ↪and factual.\n\n"
chunk_overhead_tokens = count_tokens(chunk_prompt)

map_input_tokens = sum(count_tokens(c) + chunk_overhead_tokens for c in chunks)
map_output_tokens = 150 * len(chunks)

# Combine call: section summaries are each ~150 tokens, plus prompt overhead
combine_prompt = (
    "Below are section-by-section summaries of a long document. "
    "Write a single coherent summary of the whole document in 4-6 sentences. "
    "Do not mention the section structure - write as a unified paragraph.\n\n"
)
```

```

combine_input_tokens = count_tokens(combine_prompt) + (150 * len(chunks)) #
    ↪approx
combine_output_tokens = 400

mapreduce_input_tokens = map_input_tokens + combine_input_tokens
mapreduce_output_tokens = map_output_tokens + combine_output_tokens
mapreduce_cost = estimate_cost(mapreduce_input_tokens, mapreduce_output_tokens)

# PRINT TABLE
print(f"{'Approach':<20} {'Input tokens':>14} {'Output tokens':>15} {'Est. cost',
    ↪('$')':>14}")
print("-" * 65)
print(f"{'Single-pass':<20} {single_input_tokens:>14,} {single_output_tokens:
    ↪>15,} {single_cost:>14.6f}")
print(f"{'Map-reduce':<20} {mapreduce_input_tokens:>14,}
    ↪{mapreduce_output_tokens:>15,} {mapreduce_cost:>14.6f}")
print()

if mapreduce_cost < single_cost:
    print(f"Map-reduce is CHEAPER by ${single_cost - mapreduce_cost:.6f}")
else:
    print(f"Single-pass is CHEAPER by ${mapreduce_cost - single_cost:.6f}")

print()
print("Note: for a ~4000-word document, single-pass is often cheaper because
    ↪the")
print("map-reduce approach has per-chunk prompt overhead that adds up.")
print("Map-reduce becomes cost-competitive once the document exceeds the
    ↪context window")
print("or when you need to process the same corpus repeatedly (chunk summaries
    ↪can be cached).")

```