1.2 tokenization

> This guide provides a comprehensive overview of tokenization in Natural Language Processing, designed for data science and machine learning interview preparation. It covers the evolution from classic rule-based methods to the advanced subword algorithms that power modern Large Language Models (LLMs). Key topics include the detailed mechanics, mathematical foundations, and comparative analysis of Byte-Pair Encoding (BPE), WordPiece, and the Unigram Language Model. The guide also explores cutting-edge techniques, the critical role of tokenization in the LLM pipeline, its impact on model performance, efficiency, and fairness, and methods for its evaluation. The document concludes with a curated set of theoretical and practical interview questions, complete with detailed answers and Python code implementations, to solidify understanding and prepare for rigorous technical assessments.

Knowledge Section

1. The Fundamentals of Tokenization

Tokenization is the foundational process in Natural Language Processing (NLP) of breaking down a raw text sequence into smaller, manageable units called "tokens". These tokens can be words, characters, or, most commonly in modern systems, subwords. It is an indispensable preprocessing step for nearly all downstream tasks, including part-of-speech tagging, named entity recognition, sentiment analysis, machine translation, and the operation of Large Language Models (LLMs).

Without effective tokenization, an AI system cannot discern word boundaries or sentence structure, rendering text processing impossible. For instance, the string "theemailisunread" would be meaningless without being tokenized into "the", "email", "is", "unread".

Challenges in Tokenization:

- **Ambiguity:** Handling contractions ("don't" -> "do", "n't"), compound words ("state-of-the-art"), and languages without clear word boundaries (like Chinese or Japanese).
- **Out-of-Vocabulary (OOV) Words:** Dealing with rare words, new terms, or

misspellings that are not in the model's vocabulary.

- **Context Dependency:** The meaning and appropriate tokenization of a word can change with context (e.g., "bank" as a financial institution vs. a river bank).
- **Bias and Fairness:** Tokenizers trained on large, uncurated corpora can inherit and perpetuate biases, leading to unfair or nonsensical tokenization of names, cultural terms, or text from underrepresented groups.
- **Efficiency:** The choice of tokenization directly impacts the length of the token sequence, which in turn affects the computational cost and memory requirements of models like Transformers.

The quality of tokenization directly influences the performance of the entire NLP pipeline. An inappropriate strategy can lead to information loss or distortion, fundamentally limiting a model's ability to understand and generate human language.

2. Rule-Based and Classical Tokenization

These early methods rely on predefined linguistic rules to segment text.

2.1. Simple Rule-Based Methods

- **Whitespace Tokenization:** The simplest method, which splits text based on whitespace characters (spaces, tabs, newlines). It's fast but naive, failing on punctuation and contractions.
- **Punctuation-Based Tokenization:** Treats punctuation marks as separate tokens. This is often combined with whitespace tokenization.

```
import re
import matplotlib.pyplot as plt
import numpy as np

# --- Whitespace Tokenization ---
text_ws = "The quick brown fox jumps over the lazy dog."
tokens_ws = text_ws.split()
print(f"Whitespace Tokenization: {tokens_ws}")

# --- Punctuation Tokenization ---
text_punc = "Hello Geeks! How can I help you?"
# Regex: find sequences of word characters OR any single character that is not a word cha
tokens_punc = re.findall(r'\w+|[^\w\s]', text_punc)
print(f"Punctuation Tokenization: {tokens_punc}")
```

Output:

```
Whitespace Tokenization: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
Punctuation Tokenization: ['Hello', 'Geeks', '!', 'How', 'can', 'I', 'help', 'you', '?']
```

- **Regular Expression (Regex) Tokenizers:** Offer greater flexibility by using complex patterns to define token boundaries. This is useful for extracting specific entities like emails, URLs, or phone numbers.

```
# --- Regex for Specific Patterns ---
text_email = "Contact us at info@example.com or support@example.org for help."
email_pattern = r'[\w\.-]+@[\w\.-]+\.\w+'
emails = re.findall(email_pattern, text_email)
print(f"Extracted Emails: {emails}")
```

Output:

```
Extracted Emails: ['info@example.com', 'support@example.org']
```

Limitations: Rule-based methods are brittle. They require extensive, language-specific rules, are difficult to maintain as language evolves, and handle OOV words poorly.

2.2. Penn Treebank (PTB) Tokenization Standard

The Penn Treebank (PTB) standard is a widely referenced set of rules for English tokenization, often used as a benchmark. It provides a more sophisticated rule-based approach:

- **Contractions:** Splits common contractions (e.g., "don't" becomes `do n't`, "they'll" becomes `they 'll`).
- **Punctuation:** Separates punctuation from words.
- **Hyphens:** Splits hyphenated words.
- **Quotes and Brackets:** Normalizes different types of quotation marks and brackets (e.g., `(` becomes `-LRB-`, `)` becomes `-RRB-`).

The PTB standard highlights that defining a "token" is non-trivial and filled with edge cases, revealing the inherent complexity of natural language.

3. Subword Tokenization: The Modern Paradigm

As deep learning models grew, the limitations of word-level tokenization became critical, leading to the rise of subword algorithms.

3.1. The Need for Subword Units

Word-Level Tokenization Limitations:

1. **Vocabulary Explosion:** Corpora with rich vocabularies lead to massive vocabulary sizes, increasing model parameters and memory usage.
2. **OOV Problem:** Unseen words are mapped to a single `<UNK>` token, losing all semantic information.
3. **No Morphological Awareness:** Fails to recognize that words like "read", "reads", and "reading" are morphologically related.

Character-Level Tokenization Limitations:

1. **Extremely Long Sequences:** Even short sentences produce very long token sequences, increasing computational load and making it hard for models to learn long-range dependencies.
2. **Sparse Semantics:** Individual characters carry little semantic meaning, making it difficult for the model to learn meaningful representations.

Subword Tokenization as the Solution:
Subword tokenization strikes a balance. It keeps common words as single tokens but breaks down rare words into smaller, meaningful subword units.

- **Manages Vocabulary Size:** Represents a vast vocabulary with a smaller, fixed set of subwords.
- **Handles OOV Words:** Represents unknown words as a sequence of known subwords (e.g., "tokenization" -> "token", "##ization"), preserving semantic clues.
- **Captures Morphology:** Related words share common subword stems (e.g., "read", "##ing"), making their relationship explicit to the model.

Subword tokenization frames the problem as a multi-objective optimization: balancing vocabulary size, sequence length, and representational power.

3.2. Byte-Pair Encoding (BPE)

BPE, originally a data compression algorithm, is the foundation for tokenizers used in models like **GPT, Llama, and RoBERTa**. Its core idea is to iteratively merge the most frequent pair of adjacent symbols in the training corpus.

Vocabulary Learning Algorithm:

1. **Initialization:** The initial vocabulary consists of all individual characters (or bytes, for true byte-level BPE) present in the training corpus. Text is typically pre-tokenized (e.g., by spaces), and a special end-of-word symbol (like `</w>`) is appended to each word.
2. **Count Pairs:** Count the frequency of all adjacent symbol pairs in the corpus.
3. **Identify Best Pair:** Find the pair with the highest frequency.
4. **Merge and Update:** Merge this pair into a new, single symbol. Add this new symbol to the vocabulary. Replace all occurrences of the pair in the corpus with the new symbol.
5. **Iterate:** Repeat steps 2-4 for a predefined number of merges, which determines the final vocabulary size.

Mathematical Formulation (Conceptual):
Let V be the vocabulary, initialized as V_0 with all base characters. Let C be the corpus. For each iteration $j = 1, \ldots, k$:

1. Find the most frequent pair (a^*, b^*):

$$(a_j^*, b_j^*) = \arg\max_{(a,b)} \text{freq}(a, b \mid C_{j-1}, V_{j-1})$$

2. Create a new merged symbol $c_j = \text{merge}(a_j^*, b_j^*)$.
3. Update the vocabulary: $V_j = V_{j-1} \cup \{c_j\}$.
4. Update the corpus by replacing all occurrences of (a_j^*, b_j^*) with c_j to get C_j.

Tokenization (Encoding) Process:
For a new text, BPE applies the learned merge rules in the order they were learned (or by priority). It greedily merges pairs in the text until no more merges are possible, effectively segmenting the text into the longest possible known subwords from the vocabulary.

Core Limitation: BPE is a **greedy algorithm**. It doesn't guarantee a globally optimal set of merges. Its performance is also highly dependent on the initial **pre-tokenization** step (e.g., splitting by space), which can prevent the formation of subwords that cross these predefined boundaries (e.g., merging "of" and "the" in "of the").

3.3. WordPiece

WordPiece, used by models like **BERT and DistilBERT**, is similar to BPE but uses a different merge criterion. Instead of merging the most frequent pair, it merges the pair that maximizes the likelihood of the training data.

Vocabulary Learning Algorithm:

1. **Initialization:** Start with a vocabulary of all base characters.
2. **Calculate Merge Score:** For every possible adjacent pair of symbols (t_1, t_2) in the corpus, calculate a score. The score is typically defined as:

$$\text{score}(t_1, t_2) = \frac{\text{freq}(t_1 t_2)}{\text{freq}(t_1) \times \text{freq}(t_2)}$$

   where $t_1 t_2$ is the merged token. This score, related to Pointwise Mutual Information (PMI), prioritizes pairs that co-occur more frequently than would be expected if they were independent.
3. **Select and Merge:** Choose the pair with the highest score, merge it, and add the

new subword to the vocabulary.
4. **Iterate:** Repeat until the desired vocabulary size is reached.

Tokenization (Encoding) Process:
WordPiece uses a **greedy longest-match** strategy. For a given word, it starts from the beginning and finds the longest possible prefix that exists in the learned vocabulary. Non-word-initial subwords are typically marked with a prefix like `##` (e.g., "storybook" -> `story`, `##book`). This process is often implemented efficiently using a Trie data structure.

WordPiece's likelihood-based criterion tends to produce subwords that are more linguistically plausible than BPE's purely frequency-based approach.

3.4. Unigram Language Model

The Unigram tokenizer, used in models like **T5, ALBERT, and XLNet** and implemented in the **SentencePiece** library, takes a fundamentally different, top-down approach.

Vocabulary Learning Algorithm (EM Algorithm):

1. **Initialization:** Start with a very large vocabulary of candidate subwords. This can be generated from all substrings in the corpus or by running BPE with a large number of merges. An initial probability $P(\text{subword})$ is estimated for each candidate, often from its frequency.
2. **Iterative Optimization (Expectation-Maximization):**
    - **E-Step (Expectation):** For each word in the training corpus, use the current subword probabilities and the **Viterbi algorithm** to find the most likely segmentation of that word. The probability of a segmentation is the product of its constituent subword probabilities.
    - **M-Step (Maximization):** Based on the segmentations found in the E-step, re-estimate the probabilities $P(\text{subword})$ for all subwords in the vocabulary.
3. **Pruning:** Calculate a "loss" for each subword, representing how much the total corpus likelihood would decrease if that subword were removed from the vocabulary. Prune a certain percentage (e.g., 10-20%) of the subwords with the lowest loss, ensuring that base characters are never removed.
4. **Iterate:** Repeat the EM optimization and pruning steps until the vocabulary reaches the target size.

Mathematical Formulation:
The Unigram model assumes subwords are independent. The probability of a token sequence $X = (x_1, \ldots, x_m)$ is:

$$P(X) = \prod_{i=1}^{m} P(x_i)$$

The Viterbi algorithm finds the segmentation X^* for a word W that maximizes this probability:

$$X^* = \arg \max_{X \in \text{Segmentations}(W)} \prod_{x_i \in X} P(x_i)$$

This is solved efficiently with dynamic programming. Let $\delta(i)$ be the maximum log-probability of any segmentation for the prefix $W[1..i]$:

$$\delta(i) = \max_{0 \le j < i, \text{ s.t. } W[j+1..i] \in V} (\delta(j) + \log P(W[j+1..i]))$$

Key Feature: Subword Regularization:
Because the Unigram model is probabilistic, it can generate multiple valid segmentations for a single word, each with an associated probability. During model training, instead of always using the most likely (Viterbi) segmentation, we can sample from the possible segmentations. This acts as a form of data augmentation, making the downstream model more robust to tokenization variations. This is a unique advantage over deterministic methods like BPE and WordPiece.

3.5. Comparative Summary

Feature	**Byte-Pair Encoding (BPE)**	**WordPiece**	**Unigram Language Model**
Core Idea	Data Compression (Greedy Merging)	Likelihood Maximization (Greedy Merging)	Probabilistic Model (Pruning)

Feature	Byte-Pair Encoding (BPE)	WordPiece	Unigram Language Model
Build Process	Bottom-up: Merge most frequent pairs	Bottom-up: Merge pairs that maximize likelihood	Top-down: Start large, prune least useful subwords
Segmentation	Apply learned merges greedily	Greedy longest-match	Viterbi algorithm to find most likely segmentation
Stochasticity	Deterministic	Deterministic	Probabilistic (enables subword regularization)
Key Advantage	Simple, fast, effective	Tends to create more linguistically sound subwords	Principled, probabilistic framework; robust
Used By	GPT series, Llama, RoBERTa	BERT, DistilBERT, Electra	T5, ALBERT, XLNet, SentencePiece

4. Advanced and Frontier Tokenization Methods

Research continues to push the boundaries of tokenization.

- **SentencePiece:** Not an algorithm itself, but a library by Google that implements both BPE and Unigram. Its key innovation is treating the input text as a raw stream of Unicode characters, eliminating the need for language-specific pre-tokenization. It can even learn to encode whitespace as part of a subword, making it highly versatile.
- **Stochastic Tokenization (BPE-dropout, StochasTok):** These methods introduce randomness into the deterministic tokenization process of BPE or WordPiece. For example, BPE-dropout randomly "forgets" some merge rules during encoding. This creates multiple possible tokenizations for the same input, acting as a regularizer to improve model robustness.
- **Contextual & Neural Tokenizers:** A new frontier involves using neural networks (e.g., small Transformers) to perform tokenization. These models can make

context-aware decisions, potentially leading to more semantically meaningful segmentations, though they are more complex and less transparent.

5. Tokenization in Large Language Models (LLMs)

(Information current as of early 2024)

The choice of tokenizer is a critical, and often permanent, design decision for an LLM. The model's embedding layer is tied directly to the vocabulary and token IDs produced by its specific tokenizer.

LLM Family	Example Models	Tokenization Algorithm	Vocabulary Size (approx.)	Key Characteristics
GPT	GPT-2, GPT-3, GPT-4	BPE	50k (GPT-2/3), 100k (GPT-4)	Uses a complex regex for pre-tokenization. Known for being inefficient with numbers and some non-English languages.
Llama	Llama 2, Llama 3	BPE	32k (Llama 2), 128k (Llama 3)	Llama 3 significantly increased vocab size to improve multilingual efficiency and code representation. Uses SentencePiece's BPE implementation.
BERT	BERT-base, mBERT	WordPiece	30k (BERT-base)	Uses basic whitespace/ punctuation pre-tokenization and `##` for continued subwords.
T5 / Flan	T5-base, Flan-T5	Unigram (via SentencePiece)	32k	Treats input as a raw stream, avoiding hard pre-tokenization rules, leading to better multilingual handling.
ByT5	ByT5	Byte-level	256	No subword vocabulary;

LLM Family	Example Models	Tokenization Algorithm	Vocabulary Size (approx.)	Key Characteristics
				operates directly on UTF-8 bytes. Avoids all OOV issues but results in very long sequences.

The "Tokenizer Trot" and Cost: The inefficiency of some tokenizers on certain inputs (like numbers, code, or specific languages) is a well-known issue. A string that is short for a human can expand into a very long sequence of tokens. Since many LLM APIs charge per token, an inefficient tokenizer can significantly increase operational costs. For example, GPT-family tokenizers often break numbers into individual digit tokens (`2024` -> `'20', '24'` or even `'2', '0', '2', '4'`). Llama 3's larger vocabulary aims to mitigate this by having more tokens for common number and code patterns.

Bias and Fairness: A tokenizer trained predominantly on English text will be less efficient for other languages. The same semantic content in German or Hindi might require 2-3x more tokens than in English. This "token inequality" disadvantages non-English users in terms of cost and potentially model performance, as the effective context window is shorter.

Interview Questions

Theoretical Questions

Question 1: Compare and contrast BPE, WordPiece, and the Unigram model. What are the core trade-offs when choosing one over the other?

Answer:

This question tests the fundamental understanding of the three main subword tokenization paradigms. The answer should cover their core mechanism, build process, and resulting properties.

- **Core Mechanism & Build Process:**
    - **BPE (Byte-Pair Encoding)** is a **bottom-up, frequency-driven** algorithm. It starts with individual characters and iteratively merges the most frequent adjacent pair of symbols. Its goal is essentially data compression.
    - **WordPiece** is also a **bottom-up** algorithm, but it is **likelihood-driven**. Instead of merging the most frequent pair, it merges the pair that, when added to the vocabulary, maximizes the likelihood of the training data. The scoring function prioritizes pairs whose co-occurrence is statistically surprising.
    - **Unigram** is a **top-down, probabilistic** algorithm. It starts with a large candidate vocabulary and iteratively removes subwords that contribute the least to the overall likelihood of the corpus, as calculated by a unigram language model. This process is optimized using the Expectation-Maximization (EM) algorithm.
- **Segmentation (Encoding):**
    - **BPE** applies the learned merge rules greedily to new text.
    - **WordPiece** uses a greedy "longest-match" approach, finding the longest subword prefix in its vocabulary.
    - **Unigram** uses the Viterbi algorithm to find the single most probable segmentation of a word, given the learned probabilities of its subwords.
- **Key Trade-offs and Properties:**
    i. **Optimality and Principle:**
        - **BPE** is the most heuristic and least "principled." Its greedy nature offers no guarantee of optimality.
        - **WordPiece** is more principled by optimizing for data likelihood, which often leads to more linguistically plausible subwords.
        - **Unigram** is the most principled, operating within a formal probabilistic framework (a unigram language model) and using a well-defined optimization procedure (EM) to derive its vocabulary.
    ii. **Flexibility and Robustness:**
        - **BPE & WordPiece** are deterministic: the same input

always yields the same output. This can make the model brittle.
    - **Unigram** is inherently probabilistic. It can produce multiple segmentations for a word, each with a probability. This enables **subword regularization**, where the model is trained on different sampled segmentations of the same text, improving its robustness. This is a major advantage.

iii. **Complexity:**
    - **BPE** is the simplest to implement and understand.
    - **WordPiece** is slightly more complex due to its likelihood calculation.
    - **Unigram** is the most complex, requiring implementation of the EM algorithm and Viterbi for segmentation.

Conclusion for an Interview: "In summary, the choice involves a trade-off between simplicity and theoretical rigor. BPE is simple and effective. WordPiece improves upon it by using a better criterion for merging. Unigram is the most sophisticated, offering a full probabilistic model that not only tokenizes but also allows for techniques like subword regularization to improve model robustness, at the cost of higher implementation complexity."

Question 2: Explain the mathematical intuition behind WordPiece's merge criterion. How is it fundamentally different from BPE's?

Answer:

This question probes the mathematical details distinguishing WordPiece from BPE.

- **BPE's Criterion: Frequency**
  BPE's merge criterion is purely based on frequency count. At each step, it finds the pair (a, b) that maximizes $\text{freq}(a, b)$. This is simple but naive. It might merge two very common symbols (like `e` and `s`) just because they appear often, even if their combination (`es`) doesn't form a particularly strong semantic unit compared to other less frequent but more cohesive pairs.
- **WordPiece's Criterion: Likelihood / Pointwise Mutual Information (PMI)**

WordPiece's criterion is to merge the pair (t_1, t_2) that maximizes a score function:

$$\text{score}(t_1, t_2) = \frac{\text{freq}(t_1 t_2)}{\text{freq}(t_1) \times \text{freq}(t_2)}$$

Mathematical Intuition:

i. This formula is directly related to **Pointwise Mutual Information (PMI)**, which measures how much more likely two events are to co-occur than if they were independent. In probability terms, if we treat frequencies as proportional to probabilities, the score is proportional to $\frac{P(t_1,t_2)}{P(t_1)P(t_2)}$.

ii. The numerator, $\text{freq}(t_1 t_2)$, rewards pairs that appear together often (like BPE).

iii. The denominator, $\text{freq}(t_1) \times \text{freq}(t_2)$, acts as a **normalizer**. It penalizes pairs made of very common individual symbols.

Example: Consider the pairs `('r', 'e')` and `('q', 'u')`.

- `'r'` and `'e'` are both extremely common characters in English. Their individual frequencies, $\text{freq}(r)$ and $\text{freq}(e)$, will be very high, making the denominator large. While $\text{freq}(re)$ might be high, the score might not be.
- `'q'` is an uncommon character. `'u'` is more common. However, `'q'` is almost *always* followed by `'u'`. Therefore, $\text{freq}(qu)$ will be nearly equal to $\text{freq}(q)$. The score $\frac{\text{freq}(qu)}{\text{freq}(q) \times \text{freq}(u)}$ will be very high because the `qu` combination is statistically significant and "surprising" compared to the low independent frequency of `q`.

Fundamental Difference:

BPE asks: "Which pair appears together most often?"

WordPiece asks: "Which pair's co-occurrence is most unlikely to be a result of random chance?"

This difference leads WordPiece to favor creating subwords that represent stronger linguistic or semantic units (like `qu`, `##ing`, `##tion`) over just concatenating frequent characters.

Question 3: What is the "Tokenizer Trot" or "token inequality," and how do recent LLMs like Llama 3 attempt to address it?

Answer:

This question assesses awareness of practical, real-world issues in LLM deployment and recent industry trends.

- **Definition of Tokenizer Trot / Token Inequality:**
  "Tokenizer Trot" is a colloquial term describing the phenomenon where a tokenizer is significantly less efficient for certain types of text, requiring a disproportionately large number of tokens to represent it. This leads to "token inequality," where different languages or domains are treated unfairly by the model's tokenization process.
- **Causes and Consequences:**
  i. **Training Data Bias:** Most foundational tokenizers are trained on corpora dominated by English text. This means the vocabulary is optimized for English grammar and vocabulary.
  ii. **Language Structure:** Languages with rich morphology (e.g., German, Finnish) or different scripts (e.g., Arabic, Devanagari) are often broken down into many small, sometimes meaningless, subwords or even individual characters by an English-centric tokenizer.
  iii. **Specialized Domains:** Numbers, code, and scientific notation are often not well-represented in the vocabulary and get split into individual digits or symbols (e.g., `2024` -> `2`, `0`, `2`, `4`).

  Consequences:
  - **Increased Cost:** API calls are typically priced per token. Inefficient tokenization directly translates to higher costs for non-English or code-heavy applications.
  - **Reduced Performance:** A longer token sequence consumes more of the model's limited context window. This means for the same amount of information, a user of an "inefficiently tokenized" language has less effective context space, potentially harming model performance on complex tasks.
  - **Fairness Issues:** It creates a systemic disadvantage for users and developers working with non-English languages.

- **How Llama 3 Addresses This:**
  (Based on information from Meta as of early 2024)
  Llama 3's developers made a conscious effort to mitigate this issue. Their primary strategy was to **dramatically increase the vocabulary size** and improve the training data mix.
    i. **Larger Vocabulary:** Llama 3 uses a tokenizer with a vocabulary of **128,000 tokens**, a significant increase from Llama 2's 32,000.
    ii. **Improved Training Data:** The new tokenizer was trained on a much larger and more diverse dataset, with a greater proportion of non-English and code data.
    iii. **Benefits of a Larger Vocabulary:** A larger vocabulary allows the tokenizer to have dedicated tokens for:
      - Common words and phrases in many more languages.
      - Common patterns in programming languages and numbers.
      - This results in better compression (fewer tokens per sentence) for these previously disadvantaged domains.

  Conclusion: "By increasing its vocabulary size from 32k to 128k and training it on a more diverse corpus, Llama 3's tokenizer is significantly more efficient for multilingual and code-based tasks. This directly reduces token inequality, leading to lower costs and better performance for a wider range of applications."

Practical & Coding Questions

Question 4: Implement the core vocabulary learning logic for Byte-Pair Encoding (BPE) from scratch in Python. Your function should take a corpus (as a dictionary of word counts) and the number of desired merges, and return the learned merge rules and the final vocabulary.

Answer:

This question tests the ability to translate an algorithm into clean, correct code. The implementation should handle the core loop of finding the best pair and merging it.

```
from collections import defaultdict
import re

def get_word_char_freqs(corpus_counts):
    """Converts a corpus of word counts into BPE's initial representation."""
    # e.g., {"low": 5} -> {"l o w </w>": 5}
    char_freqs = defaultdict(int)
    for word, freq in corpus_counts.items():
        spaced_word = ' '.join(list(word)) + ' </w>'
        char_freqs[spaced_word] = freq
    return char_freqs

def get_pair_stats(vocab_freqs):
    """Counts frequencies of all adjacent symbol pairs."""
    pairs = defaultdict(int)
    for word_str, freq in vocab_freqs.items():
        symbols = word_str.split()
        for i in range(len(symbols) - 1):
            pairs[(symbols[i], symbols[i+1])] += freq
    return pairs

def merge_pair(pair_to_merge, vocab_freqs_in):
    """Merges a specific pair in the vocabulary representation."""
    vocab_freqs_out = defaultdict(int)
    # Using a negative lookbehind in regex to avoid issues with overlapping matches
    # This ensures we merge 'a b' but not 'c a b' if we are merging 'a b'.
    # A simpler string replace is often shown but can be buggy. Regex is more robust.
    bigram = re.escape(' '.join(pair_to_merge))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')

    for word_str, freq in vocab_freqs_in.items():
        # Replace 'a b' with 'ab'
        new_word_str = p.sub(''.join(pair_to_merge), word_str)
        vocab_freqs_out[new_word_str] = freq

    return vocab_freqs_out

def learn_bpe_vocabulary(corpus_counts, num_merges):
    """
```

```
    Learns a BPE vocabulary from a corpus.

    Args:
        corpus_counts (dict): A dictionary of words to their frequencies.
        num_merges (int): The number of merge operations to perform.

    Returns:
        tuple: A tuple containing:
            - learned_merges (list): A list of the merged pairs in order.
            - vocab (set): The final set of subword tokens.
    """
    # 1. Initialize vocabulary with all individual characters
    initial_vocab = set()
    for word in corpus_counts.keys():
        initial_vocab.update(list(word))
    initial_vocab.add('</w>') # Add the end-of-word token

    # 2. Convert corpus to initial BPE format
    current_vocab_freqs = get_word_char_freqs(corpus_counts)

    learned_merges = []
    vocab = set(initial_vocab)

    print("Initial vocab (chars):", sorted(list(vocab)))

    for i in range(num_merges):
        # 3. Count pairs
        pair_stats = get_pair_stats(current_vocab_freqs)

        if not pair_stats:
            print("No more pairs to merge.")
            break

        # 4. Find the most frequent pair
        best_pair = max(pair_stats, key=pair_stats.get)

        # Optional: Stop if frequency is too low
        if pair_stats[best_pair] < 2:
            print(f"Highest pair frequency is < 2. Stopping early.")
```

```
            break

        print(f"\nMerge {i+1}: Merging pair {best_pair} with frequency {pair_stats[best_pa

        # 5. Merge the pair and update vocabularies
        learned_merges.append(best_pair)
        vocab.add(''.join(best_pair))
        current_vocab_freqs = merge_pair(best_pair, current_vocab_freqs)

        print("Updated corpus fragment:", list(current_vocab_freqs.keys())[0])
        print("Current vocab size:", len(vocab))

    return learned_merges, vocab

# --- Example Usage ---
corpus = {
    "low": 5, "lower": 2, "newest": 6, "widest": 3
}
merges, final_vocab = learn_bpe_vocabulary(corpus, num_merges=10)

print("\n--- Final Results ---")
print("Learned Merges (in order):", merges)
print("Final Vocabulary (sorted):", sorted(list(final_vocab)))
```

Question 5: Using the `tiktoken` library, demonstrate how the GPT-4 tokenizer (`cl100k_base`) handles different types of text: a common sentence, a sentence with numbers and code, and a non-English sentence. Analyze the token counts and explain the observed behavior.

Answer:

This practical question tests familiarity with standard industry tools (`tiktoken`) and the ability to analyze and explain tokenizer behavior, connecting back to theoretical concepts.

```python
import tiktoken
import matplotlib.pyplot as plt

# Load the cl100k_base encoding, used by gpt-4, gpt-3.5-turbo, etc.
enc = tiktoken.get_encoding("cl100k_base")

# --- Define test cases ---
text_common = "Tokenization is a fundamental step in natural language processing."
text_code_nums = "The year is 2024, and the hex code is #FFFFFF."
text_multilingual_de = "Die künstliche Intelligenz hat bemerkenswerte Fortschritte gemacht
text_multilingual_jp = "人工知能は注目すべき進歩を遂げました。" # Japanese

# --- Tokenize and Analyze ---
def analyze_tokenization(text, description):
    tokens = enc.encode(text)
    token_strings = [enc.decode([token]) for token in tokens]

    print(f"--- Analyzing: {description} ---")
    print(f"Original Text: '{text}'")
    print(f"Token Count: {len(tokens)}")
    print(f"Token IDs: {tokens}")
    print(f"Token Strings: {token_strings}\n")
    return len(tokens)

# Run analysis
counts = []
labels = [
    "Common English",
    "Code & Numbers",
    "German Sentence",
    "Japanese Sentence"
]

counts.append(analyze_tokenization(text_common, labels[0]))
counts.append(analyze_tokenization(text_code_nums, labels[1]))
counts.append(analyze_tokenization(text_multilingual_de, labels[2]))
counts.append(analyze_tokenization(text_multilingual_jp, labels[3]))

# --- Visualization ---
```

```
plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(10, 6))
bars = ax.bar(labels, counts, color=['skyblue', 'salmon', 'lightgreen', 'gold'])

ax.set_ylabel('Number of Tokens')
ax.set_title('Token Count for Different Text Types using GPT-4 Tokenizer (cl100k_base)')
ax.set_xticklabels(labels, rotation=15, ha="right")

# Add token counts on top of bars
for bar in bars:
    yval = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2.0, yval, int(yval), va='bottom', ha='center')

plt.tight_layout()
plt.show()
```

Analysis of Expected Output:

1. **Common English:**
    - **Observation:** This sentence will be tokenized very efficiently. Most common English words ("tokenization", "fundamental", "natural", "language", "processing") will be single tokens. The token count will be low, close to the number of words.
    - **Explanation:** The `cl100k_base` tokenizer has a large vocabulary trained on a massive English-heavy corpus. Common English words are well-represented and have their own token IDs.
2. **Code & Numbers:**
    - **Observation:** The token count will be higher than the word count. The number `2024` will likely be split into two tokens, like `'20'` and `'24'`. The hex code `#FFFFFF` will be split into multiple tokens, possibly `['#', 'F', 'FF', 'FF']` or similar.
    - **Explanation:** While `cl100k_base` is better than its predecessors, it still doesn't have a unique token for every possible number or code construct. It breaks them down into common numerical or character sub-units present in its vocabulary. This is an example of the "Tokenizer Trot" where specialized data is less efficiently represented.

3. **German Sentence:**
    - **Observation:** The token count will be significantly higher than the number of words. Long German compound words like `bemerkenswerte` (remarkable) and `Fortschritte` (progress) will be broken into multiple subwords (e.g., `['be', 'mer', 'kens', 'werte']`).
    - **Explanation:** Although the `cl100k_base` vocabulary has some non-English tokens, its coverage is not as comprehensive as for English. It must resort to breaking down many German words into smaller, more primitive subwords, illustrating token inequality.
4. **Japanese Sentence:**
    - **Observation:** This will likely result in the highest token-to-character ratio. Since Japanese does not use spaces, the tokenizer must segment words like `人工知能` (artificial intelligence). It will likely break this down into multiple tokens, some representing whole characters or common multi-character concepts.
    - **Explanation:** This demonstrates the challenge for BPE-based tokenizers with logographic languages. The model has learned common character combinations from its training data, but many words will be decomposed, leading to long token sequences compared to the original character count.