Byte Pair Encoding (BPE) is a **subword tokenization algorithm** used in natural language processing, particularly with large language models. It works by iteratively merging the most frequent pair of bytes (or characters) in a text into a single token, effectively breaking down words into smaller, meaningful units. This allows the model to handle rare or unknown words by representing them as combinations of these learned subword units

BPE starts from character-level representations and progressively merges frequently co-occurring patterns to build a vocabulary of **meaningful** subword tokens. This constructed vocabulary and the tokenization rules are then used as input for large language models, enabling efficient training and inference.

Core Idea:
It works by iteratively merging the most frequent pair of adjacent characters (or subwords) in a text into a single, new token. This effectively breaks down words into smaller, more meaningful units.

Why do we need?

Addressing Rare Word / Out-Of-Vocabulary (OOV)

Traditional word-level tokenization struggles with words not present in its training vocabulary (e.g., new words, typos, compound nouns). These are often mapped to an 'unknown' token (`[UNK]`), leading to information loss.
* **BPE** tokenizes words into subwords. For instance, "unfriendable" might be broken down into `['un', 'friend', 'able']`. Since `un`, `friend`, and `able` are common subwords, **the model can still understand and process the word effectively,** even if "unfriendable" itself wasn't in the original vocabulary.

Managing Vocabulary Size:

There's an ever-growing number of words in any language. Including every single word in a vocabulary would result in an extremely large vocabulary, making model training and inference computationally expensive and inefficient.
* BPE helps manage vocabulary size by creating a reasonably sized set of subword tokens that can represent most words, balancing coverage and efficiency.

Preserving Meaning and Compression

Words like "running," "runner," and "runs" share a common root: "run." BPE can identify and merge such common morphological units (e.g., "run") into a single subword token. This helps the model learn and exploit morphological similarities between words, leading to better generalization and a more compact representation.

Process

1. Initial Vocabulary:
The process starts with a vocabulary consisting of individual characters present in the training data.

`['l', 'o', 'w', 'e', 'r', 'n', 's', 't']`

extract all unique individual characters present in the given text (training data) to form the initial vocabulary (token set). While BPE can start at the byte level ("Byte Pair Encoding"), it's often explained starting at the character level. We will proceed at the character level here.

```
def initialize_vocabulary(text: str):
    """
    Extracts all unique individual characters from the given text to form the initial vocabulary.

    Args:
        text (str): The text data to be used for training.

    Returns:
        tuple: A tuple containing:
            - set: A set of unique characters present in the text (initial vocabulary).
            - list: A list of lists, where each inner list represents a word tokenized into characters.
    """
    # Split the text into words based on spaces
    words = text.split()

    # Collect all unique characters to create the initial vocabulary
    initial_vocab = set()
    for char in text:
        initial_vocab.add(char)

    # Tokenize each word into individual characters (needed for subsequent merging steps)
    tokenized_words = [[char for char in word] for word in words]

    print(f"Original Text: '{text}'")
    print(f"Initial Vocabulary (Unique Characters): {sorted(list(initial_vocab))}")
    print(f"Initial Tokenized Words: {tokenized_words}")

    return initial_vocab, tokenized_words

# Example Usage:
sample_text = "low low low lower newest"
initial_vocab, tokenized_words_list = initialize_vocabulary(sample_text)

print("\n--- Additional Explanation ---")
print(f" `initial_vocab` represents all basic tokens available at the start of BPE.")
print(f" `tokenized_words_list` stores each word broken down into individual characters,")
print("which will be used in subsequent steps to find and merge the most frequent pairs.")
```

2. Frequency Counting:
The algorithm then counts the frequency of adjacent character pairs in the training data.

```
tokenized_words_list = [['l', 'o', 'w'], ['l', 'o', 'w'], ['l', 'o', 'w'],
                        ['l', 'o', 'w', 'e', 'r'], ['n', 'e', 'w', 'e', 's', 't']]
```

```
from collections import defaultdict

def count_pair_frequencies(tokenized_words_list: list[list[str]]):
    """
    Counts the frequency of adjacent token pairs within the tokenized words list.

    Args:
      tokenized_words_list (list[list[str]]): A list of lists, where each inner list
                                              represents a word tokenized into characters
                                              or subwords.

    Returns:
      dict: A dictionary where keys are (token1, token2) tuples and values are their frequencies.
    """
    pair_frequencies = defaultdict(int)

    for word_tokens in tokenized_words_list:
        # Iterate through adjacent pairs in each word
        # For a word like ['l', 'o', 'w'], pairs are ('l', 'o') and ('o', 'w')
        for i in range(len(word_tokens) - 1):
            pair = (word_tokens[i], word_tokens[i+1])
            pair_frequencies[pair] += 1

    print("\n--- Step 2: Frequency Counting ---")
    print(f"Input tokenized words: {tokenized_words_list}")
    print("Calculated Pair Frequencies:")
    # Sort for consistent output, though not strictly necessary for the algorithm
    sorted_frequencies = sorted(pair_frequencies.items(), key=lambda item: item[1], reverse=True)
    for pair, freq in sorted_frequencies:
        print(f"  {pair}: {freq}")

    return pair_frequencies

# Continue from the previous step's output:
# initial_vocab, tokenized_words_list = initialize_vocabulary(sample_text) # This line would be run first
# Example Usage:
# Assuming tokenized_words_list is from the previous step:
# sample_text = "low low low lower newest"
# initial_vocab, tokenized_words_list = initialize_vocabulary(sample_text) # Run this if not already run

pair_counts = count_pair_frequencies(tokenized_words_list)

print("\n--- Additional Explanation ---")
print("This function identifies the most common adjacent sequences of tokens.")
print("The pair with the highest frequency will be chosen for merging in the next step.")
```

3. Merging:
The most frequent pair is merged into a new token, and the vocabulary is updated with this new token.

identify the most frequent adjacent token pair from the `count_pair_frequencies` function in the previous step. This most frequent pair is then merged into a single new token. The vocabulary is subsequently updated to include this new token. This merging process is the core of BPE, as it systematically generates new subword tokens.

```
from collections import defaultdict

# Re-including previous functions for context and executability
def initialize_vocabulary(text: str):
    """
    Extracts all unique individual characters from the given text to form the initial vocabulary.
    """
    words = text.split()
    initial_vocab = set()
    for char in text:
        initial_vocab.add(char)
    tokenized_words = [[char for char in word] for word in words]
    print(f"Original Text: '{text}'")
    print(f"Initial Vocabulary (Unique Characters): {sorted(list(initial_vocab))}")
    print(f"Initial Tokenized Words: {tokenized_words}")
    return initial_vocab, tokenized_words

def count_pair_frequencies(tokenized_words_list: list[list[str]]):
    """
    Counts the frequency of adjacent token pairs within the tokenized words list.
    """
    pair_frequencies = defaultdict(int)
    for word_tokens in tokenized_words_list:
        for i in range(len(word_tokens) - 1):
            pair = (word_tokens[i], word_tokens[i+1])
            pair_frequencies[pair] += 1

    print("\n--- Step 2: Frequency Counting (called internally by Step 3) ---")
    print("Calculated Pair Frequencies:")
    # Sort for consistent output, though not strictly necessary for the algorithm
    sorted_frequencies = sorted(pair_frequencies.items(), key=lambda item: item[1], reverse=True)
    for pair, freq in sorted_frequencies:
        print(f"  {pair}: {freq}")
    return pair_frequencies

# --- New Code for Step 3: Merging ---

def merge_pair(tokenized_words_list: list[list[str]], pair_to_merge: tuple[str, str], new_token: str):
    """
    Merges the specified pair into a new token within the tokenized words list.

    Args:
      tokenized_words_list (list[list[str]]): The current list of words tokenized into subwords.
      pair_to_merge (tuple[str, str]): The (token1, token2) pair to be merged.
      new_token (str): The string representation of the new merged token (e.g., "lo" from ("l", "o")).

    Returns:
      list[list[str]]: The updated tokenized words list after merging.
    """
    merged_tokenized_words = []
    first, second = pair_to_merge

    for word_tokens in tokenized_words_list:
        new_word_tokens = []
        i = 0
        while i < len(word_tokens):
            # Check if the current and next token form the pair to merge
            if i + 1 < len(word_tokens) and word_tokens[i] == first and word_tokens[i+1] == second:
                new_word_tokens.append(new_token)
                i += 2  # Skip both merged tokens
            else:
                new_word_tokens.append(word_tokens[i])
                i += 1
        merged_tokenized_words.append(new_word_tokens)

    return merged_tokenized_words

def perform_bpe_merge_step(current_tokenized_words: list[list[str]], current_vocab: set[str]):
    """
    Performs one step of BPE: finds the most frequent pair, merges it, and updates the vocabulary.

    Args:
      current_tokenized_words (list[list[str]]): The current list of words tokenized into subwords.
      current_vocab (set[str]): The current set of available tokens (vocabulary).

    Returns:
      tuple: A tuple containing:
        - list[list[str]]: The updated tokenized words list after merging.
        - set[str]: The updated vocabulary.
        - tuple: The pair that was merged in this step.
        - str: The new token created by the merge.
    """
    # 1. Count frequencies of all adjacent pairs
    pair_frequencies = count_pair_frequencies(current_tokenized_words)

    if not pair_frequencies:
        print("No pairs found for merging. BPE process might be complete or input is empty.")
        return current_tokenized_words, current_vocab, None, None

    # 2. Find the most frequent pair
    # The `max` function with `key=pair_frequencies.get` efficiently finds the key (pair)
    # with the highest value (frequency) in the dictionary.
    most_frequent_pair = max(pair_frequencies, key=pair_frequencies.get)

    # Construct the new token from the merged pair by joining its elements
    new_token_str = "".join(most_frequent_pair)

    print("\n--- Step 3: Merging ---")
    print(f"Most frequent pair found: {most_frequent_pair} with frequency {pair_frequencies[most_frequent_pair]}")
    print(f"New token created: '{new_token_str}'")

    # 3. Perform the merge operation across all tokenized words
    updated_tokenized_words = merge_pair(current_tokenized_words, most_frequent_pair, new_token_str)

    # 4. Update the vocabulary by adding the new token
    updated_vocab = current_vocab.copy() # Create a copy to avoid modifying the original set directly
    updated_vocab.add(new_token_str)

    print(f"Updated Tokenized Words after merge: {updated_tokenized_words}")
    print(f"Updated Vocabulary after merge: {sorted(list(updated_vocab))}")

    return updated_tokenized_words, updated_vocab, most_frequent_pair, new_token_str

# --- Example Usage (connecting all steps) ---

# Step 1: Initialize the vocabulary and tokenize words into characters
sample_text = "low low low lower newest"
initial_vocab, tokenized_words_list = initialize_vocabulary(sample_text)

# Store current state for subsequent iterations
current_tokenized_words = tokenized_words_list
current_vocab = initial_vocab

# Step 3: Perform a single BPE merge step
# This function internally calls count_pair_frequencies (from Step 2) and merge_pair
updated_words_after_merge, updated_vocab_after_merge, merged_pair, new_token = \
    perform_bpe_merge_step(current_tokenized_words, current_vocab)

print("\n--- Additional Explanation ---")
print("This step demonstrates one iteration of the BPE algorithm.")
print("In a complete BPE implementation, this `perform_bpe_merge_step` function")
print("would be repeated for a fixed number of iterations or until a desired vocabulary size is reached.")
```

4. Iteration:
Steps 2 and 3 are repeated until a desired vocabulary size or number of iterations is reached.

5. Tokenization:
When processing new text, the same merging rules learned during training are applied to break down words into the learned subword units.

Tokenization follows the training process closely, in the sense that new inputs are tokenized by applying the following steps:
1. Normalization
2. Pre-tokenization
3. Splitting the words into individual characters
4. Applying the merge rules learned in order on those splits