Algorytmy tekstowe

Krzysztof Chmielewski

Struktury Sufiksowe i Dopasowywanie Wzorców

Struktura projektu

Pliki załączone do tego sprawozdania składają się z:

- Testów (**tests**)
- Plików tekstowych używanych do analizy (**text_samples**)
- Plików ze zmodyfikowanymi algorytmami z poprzednich laboratoriów (**utils**)
- Plików z algorytmami Ukkonena, tablicami sufiksów oraz problemami sufiksowymi
- **Notebooków Jupyter** zawierających analizę poszczególnych zagadnień tego laboratorium z omówieniem i komentarzami (w języku angielskim dla wygody)
- Plików PDF, które są exportem Notebooków Jupytera (**pdfs**)

Algorytm Ukkonena jest opisany w tym pliku w sekcji o tej samej nazwie.

Analiza porównawcza algorytmów znajdowania najdłuższego wspólnego podciągu wraz z analizą algorytmów dla problemu najdłuższego wspólnego podciągu dla wielu ciągów znaków oraz najdłuższego palindromicznego podciągu znajdują się w Notebooku **longest_common_substring_compare.ipynb** (lub **PDF**).

Analiza porównawcza struktur sufiksowych znajduje się w Notebooku **suffix_structure_compare.ipynb** (lub **PDF**).

Analiza porównawcza algorytmów wyszukiwania wzorca w tekście znajduje się w Notebooku **pattern_matching_analysis.ipynb** (lub **PDF**).

Algorytm Ukkonena

```
class End:
    def __init__(self, value):
        self.value = value

class Node:
    def __init__(self, start, end):
        self.children = {}
        self.suffix_link = None
        self.start = start
        self.end = end
        self.id = -1
    def __repr__(self):
        return f"{self.id}"

class SuffixTree:
    def __init__(self, text: str):
        """
        Construct a suffix tree for the given text using Ukkonen's algorithm.

        Args:
            text: The input text for which to build the suffix tree
        """
        self.text = text + "$"
        self.root = Node(None, None)
        self.active_node = self.root
        self.active_edge = 0
        self.active_length = 0
        self.remainder = 0
        self.build_tree()
        self.count_compares = False

    def build_tree(self):
        """
        Build the suffix tree using Ukkonen's algorithm.
        """
        ID = 0
        END = End(0)

        def split_node(node : Node, last_split : Node = None) -> Node:
            nonlocal ID, END
            parent_node = Node(node.start, node.start + self.active_length)
            # parent_node.id = ID
            # ID += 1

            self.active_node.children[self.text[parent_node.start]] = parent_node

            # Rule 2
            if last_split is not None:
                last_split.suffix_link = parent_node

            node.start = parent_node.end
            new_leaf = Node(END.value - 1, END)
            new_leaf.id = ID
```

```
            ID += 1

            parent_node.children[self.text[node.start]] = node
            parent_node.children[self.text[new_leaf.start]] = new_leaf

            last_split = parent_node
            return last_split

        def add_child(node : Node, char, start) -> None:
            nonlocal ID, END
            child = Node(start, END)
            child.id = ID
            ID += 1
            node.children[char] = child

        def get_edge_text_len(node : Node):
            return node.end.value - node.start if isinstance(node.end, End) else node.end -
node.start

        n = len(self.text)
        for i in range(n):
            char = self.text[i]
            self.remainder += 1
            END.value = i + 1
            last_split = None

            while self.remainder > 0:

                if self.active_length == 0:
                    self.active_edge = i
                edge_char = self.text[self.active_edge]

                if edge_char not in self.active_node.children.keys():
                    add_child(self.active_node, char, i)

                    if last_split is not None:
                        last_split.suffix_link = self.active_node
                        last_split = None
                    if self.active_node is not self.root:
                        self.active_node = self.active_node.suffix_link or self.root

                else:
                    node = self.active_node.children[edge_char]
                    edge_len = get_edge_text_len(node)

                    if self.active_length >= edge_len:
                        self.active_node = node
                        self.active_length -= edge_len
                        self.active_edge += edge_len
                        continue

                    if self.text[node.start + self.active_length] == char:
                        self.active_length += 1
                        if last_split:
                            last_split.suffix_link = self.active_node
                        break
```

```python
                last_split = split_node(node, last_split)


                if self.active_node is self.root:
                    self.active_length -= 1
                    self.active_edge += 1
                else:
                    self.active_node = self.active_node.suffix_link or self.root

            self.remainder -= 1

    def find_pattern(self, pattern: str):
        """
        Find all occurrences of the pattern in the text.

        Args:
            pattern: The pattern to search for

        Returns:
            A list of positions where the pattern occurs in the text
        """
        # Implement pattern search using the suffix tree
        node = self.root
        i = 0
        n = len(pattern)
        compares = 0
        while i < n:
            if pattern[i] not in node.children.keys():
                return ([], compares) if self.count_compares else []

            compares += 1
            child = node.children[pattern[i]]
            edge_end = child.end.value if isinstance(child.end, End) else child.end
            edge_text = self.text[child.start : edge_end]

            j = 0
            m = len(edge_text)
            while j < m and i < n:
                if pattern[i] != edge_text[j]:
                    return ([], compares) if self.count_compares else []
                compares += 1
                j += 1
                i += 1
            node = child

        results = []

        def collect_DFS(node : Node):
            if not node.children:
                results.append(node.id)
                return

            for child in node.children.values():
                collect_DFS(child)
```

```
collect_DFS(node)
if self.count_compares: return results, compares
return results
```

Implementacja algorytmu Ukkonena służącego do budowy drzew sufiksowych zawierająca łącza sufiksowe i osobną strukturę do wskazywania końca buduje drzewo sufiksów w czasie O(n) dla tekstu o długości n, ponieważ wykonuje operację zmiany zmiennych i dodania zaległych sufiksów w czasie O(1), a dodaje sufiksy do drzewa O(n) razy.

Algorytmy wspólnych podciągów

Analiza porównawcza z wykresami znajduje się w Notebooku **longest_common_substring_compare.ipynb** (lub **PDF**).

```
from ukkonen import SuffixTree, Node
from suffix_array import SuffixArray
import random, string

def longest_common_substring_dp(str1: str, str2: str) -> str:
    """
    Find the longest common substring of two strings using dynamic programming.

    Args:
        str1: First string
        str2: Second string

    Returns:
        The longest common substring
    """
    n = len(str1)
    m = len(str2)

    max_len, end_pos = 0, 0

    DP = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[j - 1] == str2[i - 1]:
                DP[i][j] = DP[i - 1][j - 1] + 1

                if DP[i][j] > max_len:
                    max_len = DP[i][j]
                    end_pos = j
            else:
                DP[i][j] = 0

    return str1[end_pos - max_len: end_pos]

def longest_common_substring_sa(str1: str, str2: str) -> str:
    """
    Find the longest common substring of two strings using a suffix array.

    Args:
        str1: First string
        str2: Second string

    Returns:
        The longest common substring
    """
    # Concatenate the strings with a unique separator
    combined = str1 + "#" + str2 + "$"
    seperator_index = len(str1)
```

```python
    sa = SuffixArray(combined)

    n = len(sa.suffixes)
    lcp = [0] * (n-1)
    rank = [0] * n
    for i in range(n):
        rank[sa.suffixes[i]] = i

    k = 0
    for i in range(n):
        if rank[i] == n - 1:
            k = 0
            continue

        j = sa.suffixes[rank[i] + 1]
        while i + k < n and j + k < n and sa.text[i + k] == sa.text[j + k]:
            k += 1

        lcp[rank[i]] = k
        if k > 0:
            k -= 1

    max_len = 0
    position = 0
    for i in range(1, n):
        s1 = sa.suffixes[i]
        s2 = sa.suffixes[i - 1]

        if (s1 < seperator_index) != (s2 < seperator_index):
            if lcp[i - 1] > max_len:
                max_len = lcp[i - 1]
                position = s1

    return combined[position : position + max_len]

def longest_common_substring_st(str1: str, str2: str) -> str:
    """
    Find the longest common substring of two strings using a suffix tree.

    Args:
        str1: First string
        str2: Second string

    Returns:
        The longest common substring
    """
    # Concatenate the strings with a unique separator
    combined = str1 + "#" + str2 + "$"
    seperator_index = len(str1)
    longest_substring = ""

    # Build a suffix tree for the combined string
    st = SuffixTree(combined)

    # Traverse the tree to find the longest path that occurs in both strings
    def DFS(node : Node, path : list):
```

```python
        nonlocal longest_substring
        bits = set()

        if not node.children:
            if node.id < seperator_index:
                bits.add(0)
            elif node.id > seperator_index:
                bits.add(1)
            return bits

        for child in node.children.values():
            edge_end = child.end.value if hasattr(child.end, 'value') else child.end
            edge_text = st.text[child.start : edge_end]

            bits.update(DFS(child, path + [edge_text]))

        if 0 in bits and 1 in bits:
            substring = "".join(path)
            if len(substring) > len(longest_substring):
                longest_substring = substring

        return bits

    DFS(st.root, [])
    return longest_substring

def longest_common_substring_multiple(strings: list[str]) -> str:
    """
    Find the longest common substring among multiple strings using suffix structures.

    Args:
        strings: List of strings to compare

    Returns:
        The longest common substring that appears in all strings
    """
    # Implement an algorithm to find the longest common substring in multiple strings
    # You may use either suffix trees or suffix arrays

    n = len(strings)
        # Concatenate the strings with a unique separator
    if n == 0: return ""
    if n == 1: return strings[0]

    combined = ""
    seperator_indexes = []
    available_separators = list(set(string.punctuation) - set(["".join(s) for s in strings]))
    seperators = random.sample(available_separators, n)

    j = 0
    for i, str in enumerate(strings):
        combined += str + seperators[i]
        seperator_indexes.append((j, j + len(str)))
        j += len(str) + 1
    longest_substring = ""
```

```python
    # Build a suffix tree for the combined string
    st = SuffixTree(combined)

    # Traverse the tree to find the longest path that occurs in both strings
    def DFS(node : Node, path : list):
        nonlocal longest_substring
        bits = set()

        if not node.children:
            for k, (start, end) in enumerate(seperator_indexes):
                if start <= node.id < end:
                    bits.add(k)
                    break
            return bits

        for child in node.children.values():
            edge_end = child.end.value if hasattr(child.end, 'value') else child.end
            edge_text = st.text[child.start : edge_end]

            bits.update(DFS(child, path + [edge_text]))

        if len(bits) == n:
            substring = "".join(path)
            if len(substring) > len(longest_substring):
                longest_substring = substring

        return bits

    DFS(st.root, [])
    return longest_substring

def longest_palindromic_substring(text: str) -> str:
    """
    Find the longest palindromic substring in a given text using suffix structures.

    Args:
        text: Input text

    Returns:
        The longest palindromic substring
    """
    # Create a new string concatenating the original text and its reverse
    # Use suffix structures to find the longest common substring between them
    # Handle the case where palindrome centers between characters
    revtext = text[::-1]
    combined = text + "#" + revtext + "$"
    st = SuffixTree(combined)
    n = len(text)
    lps = ""

    def DFS(node : Node, path : list):
        nonlocal lps
        bits = set()
        # positions = []
        positions = set()
```

```python
        if not node.children:
            if node.id < n:
                bits.add(0)
                positions.add(node.id)
            elif node.id > n:
                bits.add(1)
                positions.add(node.id - (n+1))
            return bits, positions

        for child in node.children.values():
            edge_end = child.end.value if hasattr(child.end, 'value') else child.end
            edge_text = st.text[child.start : edge_end]

            child_bits, child_positions = DFS(child, path + [edge_text])
            bits.update(child_bits)
            positions.update(child_positions)

        if 0 in bits and 1 in bits:
            substring = "".join(path)
            l = len(substring)

            for pos in positions:
                revindex = n - (pos + l)
                if revindex in positions and l > len(lps):
                    lps = substring

        return bits, positions

    DFS(st.root, [])
    return lps
```

Porównanie struktur sufiksowych

Analiza porównawcza z wykresami znajduje się w Notebooku **suffix_structure_compare.ipynb** (lub **PDF**).

```
from ukkonen import SuffixTree, Node
from suffix_array import SuffixArray, Suffix
import time, psutil, os, sys

def get_memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss / 1024  # in KB

def get_suffix_tree_size(st: SuffixTree):
    size_of_node = sys.getsizeof(Node(None, None))

    def dfs(node: Node):
        total = size_of_node

        for child in node.children.values():
            total += dfs(child)
        return total

    return dfs(st.root)

def get_suffix_array_size(sa: SuffixArray):
    size_of_suffix = sys.getsizeof(Suffix("", None))
    return len(sa.suffixes) * size_of_suffix

def compare_suffix_structures(text: str) -> dict:
    """
    Compare suffix array and suffix tree data structures.

    Args:
        text: The input text for which to build the structures

    Returns:
        A dictionary containing:
        - Construction time for both structures
        - Memory usage for both structures
        - Size (number of nodes/elements) of both structures
    """
    prior_mem = get_memory_usage()

    # Measure time and memory usage
    start_time = time.time()
    sa = SuffixArray(text)
    end_time = time.time()
    sarray_construction = (end_time - start_time) * 1000
    mem_after_sa = get_memory_usage()
    sarray_mem_usage = mem_after_sa - prior_mem

    start_time = time.time()
    st = SuffixTree(text)
    end_time = time.time()
```

```
    stree_construction = (end_time - start_time) * 1000
    mem_after_st = get_memory_usage()
    stree_mem_usage = mem_after_st - mem_after_sa

    # Measure size of structures in bytes
    sarray_size = get_suffix_array_size(sa)
    stree_size = get_suffix_tree_size(st)

    return {
        "suffix_array": {
            "construction_time_ms": sarray_construction,
            "memory_usage_kb": sarray_mem_usage,
            "size": sarray_size
        },
        "suffix_tree": {
            "construction_time_ms": stree_construction,
            "memory_usage_kb": stree_mem_usage,
            "size": stree_size
        }
    }
```

Porównanie algorytmów wyszukiwania wzorca

A porównawcza z wykresami znajduje się w Notebooku **pattern_matching_analysis.ipynb** (lub **PDF**).

```python
def compare_pattern_matching_algorithms(text: str, pattern: str) -> dict:
    """
    Compare the performance of different pattern matching algorithms.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A dictionary containing the results of each algorithm:
        - Execution time in milliseconds
        - Memory usage in kilobytes
        - Number of character comparisons made
        - Positions where the pattern was found
    """
    prior_mem = get_memory_usage()

    # Implement algorithm comparisons
    # For each algorithm:
    #   1. Measure execution time
    #   2. Measure memory usage
    #   3. Count character comparisons
    #   4. Find pattern positions

    # Measuring time and memory usage

    # Naive algorithm time exec and mem usage
    start_time = time.time()
    naive_result, naive_compares = naive_pattern_match(text, pattern)
    end_time = time.time()
    mem_after_naive = get_memory_usage()
    naive_time_exec = (end_time - start_time) * 1000
    naive_mem_usage = mem_after_naive - prior_mem

    # Suffix array time exec and mem usage
    sa = SuffixArray(text)
    start_time = time.time()
    sa.count_compares = True
    sa_result, sa_compares = sa.find_pattern(pattern)
    end_time = time.time()
    mem_after_sa = get_memory_usage()
    sarray_time_exec = (end_time - start_time) * 1000
    sarray_mem_usage = mem_after_sa - naive_mem_usage

    # Suffix tree time exec and mem usage
    st = SuffixTree(text)
    start_time = time.time()
```

```
    st.count_compares = True
    st_result, st_compares = st.find_pattern(pattern)
    end_time = time.time()
    mem_after_st = get_memory_usage()
    stree_time_exec = (end_time - start_time) * 1000
    stree_mem_usage = mem_after_st - mem_after_sa

    # KMP time exec and mem usage
    start_time = time.time()
    kmp_result, kmp_compares = kmp_pattern_match(text, pattern)
    end_time = time.time()
    mem_after_kmp = get_memory_usage()
    kmp_time_exec = (end_time - start_time) * 1000
    kmp_mem_usage = mem_after_kmp - mem_after_st

    # Boyer-Moore time exec and mem usage
    start_time = time.time()
    bm_result, bm_compares = boyer_moore_pattern_match(text, pattern)
    end_time = time.time()
    mem_after_bm = get_memory_usage()
    bm_time_exec = (end_time - start_time) * 1000
    bm_mem_usage = mem_after_bm - mem_after_kmp

    # Rabin-Karp time exec and mem usage
    start_time = time.time()
    rk_result, rk_compares = rabin_karp_pattern_match(text, pattern)
    end_time = time.time()
    mem_after_rk = get_memory_usage()
    rk_time_exec = (end_time - start_time) * 1000
    rk_mem_usage = mem_after_rk - mem_after_bm

    # Aho-Corasick time exec and mem usage
    ac = AhoCorasick([pattern])
    start_time = time.time()
    ac_result, ac_compares = ac.search(text)
    end_time = time.time()
    mem_after_ac = get_memory_usage()
    ac_time_exec = (end_time - start_time) * 1000
    ac_mem_usage = mem_after_ac - mem_after_rk

    sa_result.sort()
    st_result.sort()
    kmp_result.sort()
    bm_result.sort()
    rk_result.sort()
    ac_result = [index for index, _ in ac_result]
    ac_result.sort()

    return {
        "Naive": {
            "execution_time_ms": naive_time_exec,
            "memory_usage_kb": naive_mem_usage,
            "compares": naive_compares,
            "results": naive_result
        },
        "Suffix array": {
```

```
            "execution_time_ms": sarray_time_exec,
            "memory_usage_kb": sarray_mem_usage,
            "compares": sa_compares,
            "results": sa_result
        },
        "Suffix tree": {
            "execution_time_ms": stree_time_exec,
            "memory_usage_kb": stree_mem_usage,
            "compares": st_compares,
            "results": st_result
        },
        "Knuth-Morris-Pratt": {
            "execution_time_ms": kmp_time_exec,
            "memory_usage_kb": kmp_mem_usage,
            "compares": kmp_compares,
            "results": kmp_result
        },
        "Boyer-Moore": {
            "execution_time_ms": bm_time_exec,
            "memory_usage_kb": bm_mem_usage,
            "compares": bm_compares,
            "results": bm_result
        },
        "Rabin-Karp": {
            "execution_time_ms": rk_time_exec,
            "memory_usage_kb": rk_mem_usage,
            "compares": rk_compares,
            "results": rk_result
        },
        "Aho-Corasick": {
            "execution_time_ms": ac_time_exec,
            "memory_usage_kb": ac_mem_usage,
            "compares": ac_compares,
            "results": ac_result
        }
    }
```

Algorytmy z poprzednich laboratoriów

Algorytm naiwny

```
def naive_pattern_match(text: str, pattern: str) -> list[int]:
    """
    Implementation of the naive pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the text
    """
    if pattern == "": return []

    def compare(str1, str2):
        if len(str1) != len(str2): return False, 0
        compares = 0
        for i in range(len(str1)):
            compares += 1
            if str1[i] != str2[i]:
                return False, compares
        return True, compares

    n = len(text)
    m = len(pattern)
    result = []
    compares = 0
    for i in range(n):
        comp_result, add_to_compares = compare(text[i:i+m], pattern)
        if comp_result:
            result.append(i)
        compares += add_to_compares

    return result, compares
```

Aho-Corasick

```python
from collections import deque
from typing import List, Tuple, Optional

class AhoCorasickNode:
    def __init__(self, char):
        self.char = char
        self.children = dict()
        self.fail_link = None
        self.outputs = []

class AhoCorasick:
    def __init__(self, patterns: List[str]):
        self.root = AhoCorasickNode(None)
        self.patterns = list(filter(lambda x: len(x) != 0, patterns))

    def _build_trie(self):
        """Builds the trie structure for the given patterns."""
        for pattern in self.patterns:

            node = self.root
            for c in pattern:

                if c not in node.children:
                    node.children[c] = AhoCorasickNode(c)

                node = node.children[c]

            node.outputs.append(pattern)

    def _build_failure_links(self):
        """Builds failure links and propagates outputs through them."""
        Q = deque([])

        for node in self.root.children.values():
            node.fail_link = self.root
            Q.append(node)

        while Q:
            node = Q.popleft()

            for c, child in node.children.items():
                fnode = node.fail_link

                while fnode is not None and c not in fnode.children:
                    fnode = fnode.fail_link

                child.fail_link = fnode.children[c] if fnode else self.root
                child.outputs += child.fail_link.outputs if child.fail_link else []
                Q.append(child)
```

```python
def search(self, text: str) -> Tuple[List[Tuple[int, str]], int]:
        """
        Searches for all occurrences of patterns in the given text.

        Returns:
            List of tuples (start_index, pattern).
        """
        if len(text) == 0: return [], 0
        results = []
        self._build_trie()
        self._build_failure_links()
        node = self.root
        compares = 0

        for i, c in enumerate(text):
            compares += 1
            while node is not None and c not in node.children:
                compares += 1
                node = node.fail_link

            if node is None:
                node = self.root
                continue
            node = node.children[c]

            for pattern in node.outputs:
                results.append((i - len(pattern) + 1, pattern))

        return results, compares
```

Boyer-Moore

```python
def compute_bad_character_table(pattern: str) -> dict:
    """
    Compute the bad character table for the Boyer-Moore algorithm.

    Args:
        pattern: The pattern string

    Returns:
        A dictionary with keys as characters and values as the rightmost position
        of the character in the pattern (0-indexed)
    """
    table = {}
    for i, c in enumerate(pattern):
        table[c] = i
    return table

def compute_good_suffix_table(pattern: str) -> list[int]:
    """
    Compute the good suffix table for the Boyer-Moore algorithm.

    Args:
        pattern: The pattern string
```

```
    Returns:
        A list where shift[i] stores the shift required when a mismatch
        happens at position i of the pattern
    """
    m = len(pattern)
    shift = [0] * (m + 1)
    border_pos = [0] * (m + 1)
    compares = 0

    i = m
    j = m + 1
    border_pos[i] = j

    while i > 0:
        compares += 1
        while j <= m and pattern[i - 1] != pattern[j - 1]:
            compares += 1
            if shift[j] == 0:
                shift[j] = j - i
            j = border_pos[j]
        i -= 1
        j -= 1
        border_pos[i] = j

    j = border_pos[0]
    for i in range(m + 1):
        if shift[i] == 0:
            shift[i] = j
        if i == j:
            j = border_pos[j]

    return shift, compares


def boyer_moore_pattern_match(text: str, pattern: str) -> tuple[list[int], int]:
    """
    Implementation of the Boyer-Moore pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the text
    """
    if not text or not pattern: return [], 0

    BCT, (GST, compares) = compute_bad_character_table(pattern),
compute_good_suffix_table(pattern)
    result = []
    i = 0
    n, m = len(text), len(pattern)
    while i + m <= n:
        j = m - 1
        while j >= 0 and pattern[j] == text[i+j]:
```

```
            compares += 1
            j -= 1
        compares += 1

        if j < 0:
            result.append(i)
            i += GST[0]
        else:
            BC_shift = j - BCT.get(text[i+j], -1)
            GS_shift = GST[j + 1]
            i += max(BC_shift, GS_shift, 1)

    return result, compares
```

Knuth-Morris-Pratt

```python
def compute_lps_array(pattern: str) -> list[int]:
    """
    Compute the Longest Proper Prefix which is also Suffix array for KMP algorithm.

    Args:
        pattern: The pattern string

    Returns:
        The LPS array
    """

    n = len(pattern)
    result = [0] * n
    l = 0
    for i in range(1,n):
        while l > 0 and pattern[i] != pattern[l]:
            l = result[l - 1]

        if pattern[i] == pattern[l]:
            l += 1

        result[i] = l

    return result


def kmp_pattern_match(text: str, pattern: str) -> tuple[list[int], int]:
    """
    Implementation of the Knuth-Morris-Pratt pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the text
    """
    if pattern == "" or text == "": return [], 0
    LPS = compute_lps_array(pattern)
    n = len(text)
    m = len(pattern)
    result = []

    ti = 0
    pi = 0
    compares = 0
    while ti < n:
        if pattern[pi] == text[ti]:
            ti += 1
            pi += 1
            compares += 1

        if pi == m:
            result.append(ti - pi)
```

```
            pi = LPS[pi - 1]

        elif ti < n and pattern[pi] != text[ti]:
            compares += 1
            if pi != 0:
                pi = LPS[pi - 1]
            else:
                ti += 1

    return result, compa
```

Rabin-Karp

```
def rabin_karp_pattern_match(text: str, pattern: str, prime: int = 101) ->
tuple[list[int], int]:
    """
    Implementation of the Rabin-Karp pattern matching algorithm.

    Args:
        text: The text to search in
        pattern: The pattern to search for
        prime: A prime number used for the hash function

    Returns:
        A list of starting positions (0-indexed) where the pattern was found in the text
    """

    def compare(str1, str2):
        if len(str1) != len(str2): return False, 0
        compares = 0
        for i in range(len(str1)):
            compares += 1
            if str1[i] != str2[i]:
                return False, compares
        return True, compares

    def hash(old_hash, oldc):
        return (old_hash + oldc) % prime

    def unhash(old_hash, oldc):
        return (old_hash - oldc + prime) % prime

    def hash_string(string):
        hash_res = 0
        for char in string:
            hash_res = hash(hash_res, ord(char))
        return hash_res

    if text == "" or pattern == "": return [], 0

    result = []
    n = len(text)
    m = len(pattern)
    hp = hash_string(pattern)
    htw = hash_string(text[:m])
    compares = 0

    i = 0
    while True:
        if htw == hp and text[i:i+m] == pattern:
            comp_result, add_to_compares = compare(text[i:i+m], pattern)
            compares += add_to_compares
            if comp_result:
                result.append(i)

        if n <= i + m:
            break
```

```
        htw = hash(htw, ord(text[i+m]))
        htw = unhash(htw, ord(text[i]))
        i += 1
    return result, compares
```

Testy

Test algorytmu Ukkonena tworzenia drzew sufiksowych (tests/test_ukkonen.py)

```
import pytest

from ukkonen import SuffixTree
import sys
sys.setrecursionlimit(10000)

class TestUkkonen:

    def test_ukkonen_by_number_of_leaves(self):
        short_texts = [
            "abc",
            "aaa",
            "abab",
            "racecar",
            "banana",
            "abcabc",
            "aaaaa",
            "xyzzy",
            "aabbaa",
            "skibidi",
            "addadda"
        ]

        medium_texts = [
            "abracadabrabracabracad",
            "mississippiisreallycool",
            "abcabcabcabcabcabcabcabcabc",
            "zzzyyyxxxyyyzzzzyx",
            "abababababababababab",
            "abcdabcdabcdabcdabcd",
            "xyzxyzxyzxyzxyzxyzxyzxyz",
            "thequickbrownfoxjumpsoverthelazydog",
            "loremipsumdolorsitametconsectetur",
            "aaaaabbbbbcccccdddddeeeeefffff"
        ]

        long_texts = [
            "a" * 100,
            "ab" * 150,
            "abcde" * 60,
            "banana" * 50,
            "ababababababababababababababababababababababababa",
            "loremipsum" * 25,
            "zzzyyyxxxwwwvvvuuutttsssrrrqqqpppooonnnmmmlllkkkjjjiii",
            "skibiditoilet" * 20,
            "helloworld" * 30,
            "abcdefghijklmnopqrstuvwxyz" * 4,
        ]
```

```
        very_long_text = [
            "a" * 512,
            "abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc",
            "abababab" * 70,
            "xyz" * 170 + "end",
            "".join(["abcde"[i % 5] for i in range(600)]),  # cyclic
            "mississippimississippimis",
            "mississippimississippimississippimississippimississippimississippi",
            "Nory was a Catholic because her mother was a Catholic, and Nory's mother was a Catholic because her father was a Catholic, and her father was a Catholic because his mother was a Catholic, or had been."
        ]

        def check_leaves(tree : SuffixTree) -> bool:
            def count_leaves(node):
                if not node.children:
                    return 1
                return sum(count_leaves(child) for child in node.children.values())
            return len(tree.text) == count_leaves(tree.root)

        list = short_texts + medium_texts + long_texts + very_long_text
        for text in list:
            suffix_tree = SuffixTree(text)
            assert check_leaves(suffix_tree)




    def test_ukkonen_finding_pattern(self):

        def test_batch(text : str, patterns : dict) -> bool:
            suffix_tree = SuffixTree(text)
            for pattern, expected in patterns.items():
                result = suffix_tree.find_pattern(pattern)
                result.sort()
                assert result == expected, f"Failed for pattern '{pattern}': got {result}, expected {expected}"
            print(f"PASSED pattern matching test")
            return True

        assert test_batch("abracadabra", {
            "abra": [0, 7],
            "cad": [4],
            "a": [0, 3, 5, 7, 10],
            "ra": [2, 9],
            "xyz": [],
        })

        assert test_batch("banana", {
            "na": [2, 4],
            "ana": [1, 3],
            "nana": [2],
            "banana": [0],
            "bananas": [],
```

```
        })

        assert test_batch("LLLorem im daolor sit amet, consectetur adipLorema elit, sed do eiusmod
Loremc incididunt ut labore ", {"Lorem" : [2, 44, 72]})
        assert test_batch("abracad" * 50, {"abra" : list(range(0,350,7))})
        assert test_batch("Nory was a Catholic because her mother was a Catholic, and Nory's" \
        " mother was a Catholic because her father was a Catholic, and her father was a Catholic" \
        " because his mother was a Catholic, or had been.", {"Catholic" : [11, 45, 79, 113, 144, 178]})
```

Test znajdowania wzorca za pomocą tablicy sufiksów (tests/test_suffix_array.py)

```
import pytest
from suffix_array import SuffixArray

class TestSuffixArray:
    def test_array_finding_pattern(self):

        def test_batch(text : str, patterns : dict) -> bool:
            suffix_array = SuffixArray(text)
            for pattern, expected in patterns.items():
                result = suffix_array.find_pattern(pattern)
                result.sort()
                assert result == expected, f"Failed for pattern '{pattern}': got {result}, expected
{expected}"
            print(f"PASSED pattern matching test")
            return True

        assert test_batch("abracadabra", {
            "abra": [0, 7],
            "cad": [4],
            "a": [0, 3, 5, 7, 10],
            "ra": [2, 9],
            "xyz": [],
        })

        assert test_batch("banana", {
            "na": [2, 4],
            "ana": [1, 3],
            "nana": [2],
            "banana": [0],
            "bananas": [],
        })

        assert test_batch("abracad" * 50, {"abra" : list(range(0,350,7))})
        assert test_batch("Nory was a Catholic because her mother was a Catholic, and Nory's" \
        " mother was a Catholic because her father was a Catholic, and her father was a Catholic" \
        " because his mother was a Catholic, or had been.", {"Catholic" : [11, 45, 79, 113, 144,
178]})
```

Test dla problemów sufiksowych (tests/test_substring_problems.py)

```
import pytest

from substring_problems import longest_common_substring_sa, longest_common_substring_st,
longest_common_substring_dp, longest_palindromic_substring, longest_common_substring_multiple

class TestSubstringProblems:
    def test_lcs_dp(self):
        test_cases = [
            ("abcdef", "abc", "abc"),
            ("xyz", "abc", ""),
            ("banana", "banana", "banana"),
            ("hello", "lo", "lo"),
            ("startmatch", "start", "start"),
            ("abracadabra", "racad", "racad"),
            ("bajojajo bajojajo", "ja ci dam pajacu bajojajo", " bajojajo"),
            ("ababababab", "babababa", "babababa"),
            ("Zażółć gęślą jaźń", "gęślą", "gęślą"),
            ("abcdef", "", ""),
            ("", "", ""),
            ("x", "x", "x"),
            ("abcxyz123", "xyx789abc", "abc"),
            ("aaaaabbbbcccc", "xxxbbbbyyyy", "bbbb"),
            ("testlongest", "longest", "longest"),
            ("prefixmatch", "matchpostfix", "match"),
            ("hello, world!", "world!", "world!"),
            ("hello😀world", "😀wo", "😀wo"),
        ]

        for s1, s2, expected in test_cases:
            result = longest_common_substring_dp(s1,s2)
            assert result == expected
    def test_lcs_sa(self):
        test_cases = [
            ("abcdef", "abc", "abc"),
            ("xyz", "abc", ""),
            ("banana", "banana", "banana"),
            ("hello", "lo", "lo"),
            ("startmatch", "start", "start"),
            ("abracadabra", "racad", "racad"),
            ("bajojajo bajojajo", "ja ci dam pajacu bajojajo", " bajojajo"),
            ("ababababab", "babababa", "babababa"),
            ("Zażółć gęślą jaźń", "gęślą", "gęślą"),
            ("abcdef", "", ""),
            ("", "", ""),
            ("x", "x", "x"),
            ("abcxyz123", "xyx789abc", "abc"),
            ("aaaaabbbbcccc", "xxxbbbbyyyy", "bbbb"),
            ("testlongest", "longest", "longest"),
            ("prefixmatch", "matchpostfix", "match"),
            ("hello, world!", "world!", "world!"),
            ("hello😀world", "😀wo", "😀wo"),
        ]
```

```
    for s1, s2, expected in test_cases:
        result = longest_common_substring_sa(s1,s2)
        assert result == expected

def test_lcs_st(self):
    test_cases = [
        ("abcdef", "abc", "abc"),
        ("xyz", "abc", ""),
        ("banana", "banana", "banana"),
        ("hello", "lo", "lo"),
        ("startmatch", "start", "start"),
        ("abracadabra", "racad", "racad"),
        ("bajojajo bajojajo", "ja ci dam pajacu bajojajo", " bajojajo"),
        ("ababababababab", "bababababa", "bababababa"),
        ("Zażółć gęślą jaźń", "gęślą", "gęślą"),
        ("abcdef", "", ""),
        ("", "", ""),
        ("x", "x", "x"),
        ("abcxyz123", "xyx789abc", "abc"),
        ("aaaaabbbbcccc", "xxxbbbbyyyy", "bbbb"),
        ("testlongest", "longest", "longest"),
        ("prefixmatch", "matchpostfix", "match"),
        ("hello, world!", "world!", "world!"),
        ("hello😀world", "😀wo", "😀wo"),
    ]

    for s1, s2, expected in test_cases:
        result = longest_common_substring_st(s1,s2)
        assert result == expected

def test_multiple_lcs(self):
    test_cases = {
        ("abc", "abc"): "abc",
        ("abc", "def"): "",
        ("abcde", "cdefg", "defgh"): "de",
        ("ababab", "babab", "abab"): "abab",
        ("aaaa", "aa", "aaa"): "aa",
        ("racecar", "myrace", "racing"): "rac",
        ("abcXXXXcba", "abc123cba", "zzzcbayy"): "cba",
        ("a#b@c", "#b@c$d", "@c$def"): "@c",
        ("123!abc", "!abc456", "xyz!abc"): "!abc",
        ("",): "",
        ("abc",): "abc",
        (): "",
        ("banana", "ananas", "bandana"): "ana",
        ("thequickbrownfox", "quickfox", "lazyquickfox"): "quick",
        ("123456", "456789", "0456123"): "456",
        ("👀hello👋", "👋hello", "sayhello👋"): "hello",
    }

    for strings, expected in test_cases.items():
        result = longest_common_substring_multiple(strings)
        assert result == expected
```

```python
def test_palindrome(self):
      test_cases = {
          "cbbd": "bb",
          "a": "a",
          "racecar": "racecar",
          "abacdfgdcaba": "aba",
          "": "",
          "abcddcbazzz": "abcddcba",
          "banana": "anana",
          "level": "level",
          "aaabaaaa": "aaabaaa",
          "xyzzyx": "xyzzyx",
          "bajojab cd ad d" : "bajojab",
      }

      for text, expected in test_cases.items():
          result = longest_palindromic_substring(text)
          assert result == expected
```

Wyniki testów

```
(text) krzysztof@krzysztof-ThinkPad-E15-Gen-4:~/PROJECTS/Text-Algorithms/lab5$ pytest tests/
============================== test session starts ==============================
platform linux -- Python 3.12.3, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/krzysztof/PROJECTS/Text-Algorithms/lab5
collected 8 items

tests/test_substring_problems.py .....                                   [ 62%]
tests/test_suffix_array.py .                                             [ 75%]
tests/test_ukkonen.py ..                                                 [100%]

=============================== 8 passed in 0.07s ===============================
```