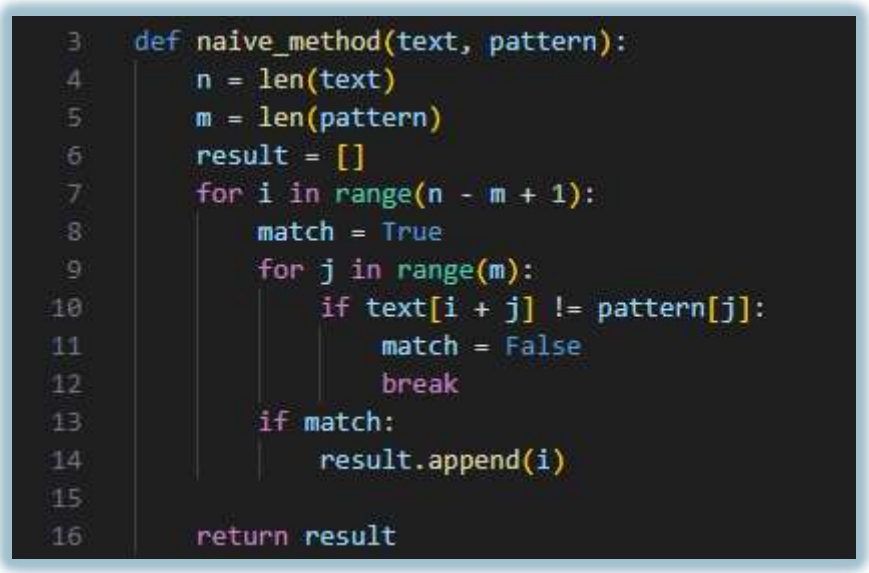
Omówienie programów

https://github.com/jstefaniak99/Lab1_Wzorce

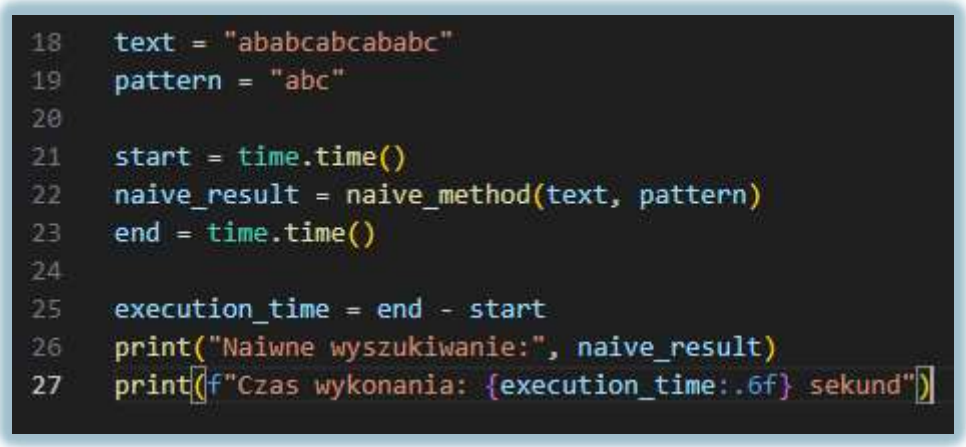
Lab1 – naiwna metoda wyszukiwania

Pierwszy program implementuje naiwną metodę wyszukiwania wzorca w tekście. Metoda przeszukuje tekst w poszukiwaniu wszystkich ciągów we wzorcu.



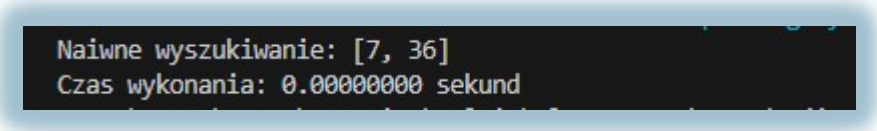
```
3   def naive_method(text, pattern):
4       n = len(text)
5       m = len(pattern)
6       result = []
7       for i in range(n - m + 1):
8           match = True
9           for j in range(m):
10              if text[i + j] != pattern[j]:
11                  match = False
12                  break
13          if match:
14              result.append(i)
15
16      return result
```

Algorytm iteruje po każdej pozycji w tekście, gdzie wzorzec może się pojawić. Dla każdej pozycji sprawdza, czy wzorzec pasuje do fragmentu tekstu rozpoczynającego się w danej pozycji. Jeżeli wzorzec pasuje, wynik dodawany jest do „result”



```
18  text = "ababcabcababc"
19  pattern = "abc"
20
21  start = time.time()
22  naive_result = naive_method(text, pattern)
23  end = time.time()
24
25  execution_time = end - start
26  print("Naiwne wyszukiwanie:", naive_result)
27  print(f"Czas wykonania: {execution_time:.6f} sekund")
```

Na samym końcu wypisuje wynik użytkownikowi, przy jednoczesnym sprawdzaniu, jak długo się wykonywał.

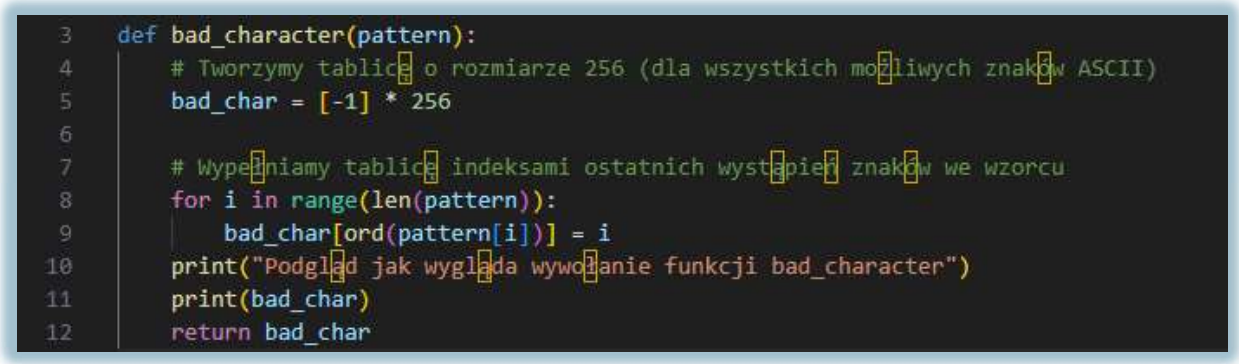


Jeżeli chodzi o zalety to algorytm jest łatwy do zrozumienia jak i zaimplementowania, może być używany do wyszukiwania wzorca w jakimś ciągu znaków, bez konieczności dodatkowych założeń

Natomiast jeżeli chodzi o wadę to złożoność czasowa O może być nieakceptowalna dla dużych tekstów i wzorców. Algorytm jest szczególnie nieefektywny, gdy długość wzorca m jest zbliżona do długości tekstu n.

Lab1_2 – Algorytm Boyera-Moore

Program implementuje algorytm Boyer-Moore do wyszukiwania wzorca w tekście. Algorytm ten jest bardziej wydajny niż naiwny algorytm wyszukiwania, zwłaszcza w przypadku długich tekstów i wzorców, dzięki wykorzystaniu dwóch heurystyk: złego znaku i dobrego sufiksu.



```
3   def bad_character(pattern):
4       # Tworzymy tablicę o rozmiarze 256 (dla wszystkich możliwych znaków ASCII)
5       bad_char = [-1] * 256
6
7       # Wypełniamy tablicę indeksami ostatnich wystąpień znaków we wzorcu
8       for i in range(len(pattern)):
9           bad_char[ord(pattern[i])] = i
10      print("Podgląd jak wygląda wywołanie funkcji bad_character")
11      print(bad_char)
12      return bad_char
```

Tworzy tablicę „bad_char” z wartościami -1, oznaczającymi brak wystąpień znaku i wypełnia tablicę „bad_char” indeksami ostatnich wystąpień znaków we wzorcu.

```python
def boyer_moore_search(text, pattern):
    n = len(text)
    m = len(pattern)
    bad_char = bad_character(pattern)
    result = []

    s = 0
    while s <= n - m:
        j = m - 1
        while j >= 0 and pattern[j] == text[s + j]: # porównanie wzorca z tekstem od prawej do lewej strony
            j -= 1
        # Jeśli wzorzec pasuje (indeks j staje się -1)
        if j < 0:
            result.append(s)
            s += (m - bad_char[ord(text[s + m])] if s + m < n else 1)
        else:
            # Przesuwamy wzorzec zgodnie z heurystyką złego znaku
            s += max(1, j - bad_char[ord(text[s + j])])
    return result
```

Wywołuje funkcję „bad_character" dla wzorca, aby uzyskać tablicę „bad_char". Iteruje przez tekst, przesuwając wzorzec zgodnie z heurystyką złego znaku. Gdy wzorzec pasuje do fragmentu tekstu, zapisuje pozycję do „result".

Przykład wyszukiwania wzorca:

```
Podgląd jak wygląda wywołanie funkcji bad_character
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 0, 1, 2, -1, -1, -1, -1, -1, -1
, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
Boyer-Moore wyszukiwanie: [7, 36]
Czas wykonania: 0.0009965897 sekund
```

Do zalet możemy zaliczyć to, że jest bardzo szybkim algorytmem wyszukiwania wzorca, dzięki efektywnemu pomijaniu części tekstu. Przekłada się to na większą wydajność w większości przypadków.

Wadą jest na pewno, fakt iż jest bardziej skomplikowanym algorytmem, ponadto w niektórych specyficznych przypadkach złożoność czasowa algorytmu może wzrosnąć.

Podsumowanie:

Pomimo większej złożoności implementacji, algorytm Algorytm Boyera-Moore w praktyce często przewyższa proste algorytmy wyszukiwania wzorów np. wyżej zaprezentowany algorytm naiwny. Dzięki temu, algorytm Boyera-Moore, lepiej odnajduje się w pracy, gdzie musimy wyszukiwać wzorce na dużych zbiorach danych