Michał Szczurek - lab 1.

Zadanie 0. - Implementacja Algorytmów

0.1. Algorytm nawiwny

In [1]:
```
def naive(text, pattern, printing=True):
    count = 0
    for s in range(len(text)-len(pattern)+1):
        if (pattern == text[s : s + len(pattern)]):
            count += 1
            if printing:
                print(f"Przesunięcie {s} jest poprawne")
    return count
```

In [2]:
```
naive("alazzzala","ala")
```

```
Przesunięcie 0 jest poprawne
Przesunięcie 6 jest poprawne
```

Out[2]:
```
2
```

0.2 Autmoat skończony

In [3]:
```
def fa_string_matching(text, delta, printing=True):
    q = 0
    count = 0
    for s in range(0, len(text)):
        if text[s] not in delta[0]:
            q = 0
            continue
        q = delta[q][text[s]]
        if(q == len(delta) - 1):
            if printing:
                print(f"Przesunięcie {s+1-q} jest poprawne")
            count += 1
    return count
```

In [4]:
```
def transition_table(pattern):
    characters = set()
    for letter in pattern:
        characters.add(letter)

    result = []
    for q in range(0, len(pattern) + 1):
        result.append({})
        for a in characters:
            # nie można przejść więcej niż 1 stan na raz, a ostatni to już akceptujący
            k = min(len(pattern), q + 1)
            while True:
                # tmp to pierwsze q-1 znaków pattern (bo jetseśmy w stanie q, więc musieliśmy przejść przez q-1 jako
w)
                # i badany znak.
                # Trzeba znaleźć znajdłuższy sufix tmp, taki, żeby był prefixem pattern
                tmp = pattern[:q] + a
                if k== 0 or (tmp[-k:] == pattern[:k]):
                    break
                k = k - 1
            result[q][a] = k
    return result
```

In [5]:
```
fa_string_matching("alazzzala",transition_table("ala"))
```

```
Przesunięcie 0 jest poprawne
Przesunięcie 6 jest poprawne
```

Out[5]:
```
2
```

0.3 Algorytm KMP

In [6]:
```
def kmp_string_matching(text, pattern, pi, printing=True):
    # pi jest parametrem, by łatwiej oddzielić czas preprocessingu od czasu algorytmu
    q = 0
    count = 0
    for i in range(0, len(text)):
        while(q > 0 and pattern[q] != text[i]):
            q = pi[q-1]
        if(pattern[q] == text[i]):
            q = q + 1
        if(q == len(pattern)):
            count += 1
            if printing:
                print(f"Przesunięcie {i + 1 - q} jest poprawne")
            q = pi[q-1]
    return count
```

In [7]:
```
def prefix_function(pattern):
    pi = [0]
    k = 0
    # pi[i] - długość najdłuższego poprawnego prefixu pattern[0..i], który jednocześnie jest sufixem pattern[0..i]
    for q in range(1, len(pattern)):
        while(k > 0 and pattern[k] != pattern[q]):
            k = pi[k-1]
        if(pattern[k] == pattern[q]):
            k = k + 1
        pi.append(k)
    return pi
```

In [8]:
```
kmp_string_matching("alazzzala","ala", prefix_function("ala"))
```

```
Przesunięcie 0 jest poprawne
Przesunięcie 6 jest poprawne
```

Out[8]:
```
2
```

Zadanie 1. Funkcje mierzące czas

In [10]:
```
from timeit import default_timer as timer
```

Ze względu na konieczność rozdzielenia czasów preprocessingu i właściwych algorytmów, algorytmy miały różną liczbę parametrów co znacząco utrudniłoby wykonanie uniwersalnej funkcji, bez modyfikowania algorytmów (np dodając zbędne parametry). Wobec tego dla każdego algorytmu napisana została osobna funkcja.

In [11]:
```
def get_time_naive(text, pattern):
    start = timer()
    naive(text, pattern, False)
    end = timer()
    print(f"Czas działania algorytmu: {end - start}")
```

In [12]:
```
def get_time_fa(text, pattern):
    start = timer()
    delta = transition_table(pattern)
    processed = timer()
    fa_string_matching(text, delta, False)
    end = timer()
    print(f"Czas preprocessingu: {processed - start}")
    print(f"Czas dopasowania: {end - processed}")
    print(f"Sumaryczny czas działania: {end - start}")
```

In [13]:
```
def get_time_kmp(text, pattern):
    start = timer()
    pi = prefix_function(pattern)
    processed = timer()
    kmp_string_matching(text, pattern, pi, False)
    end = timer()
    print(f"Czas preprocessingu: {processed - start}")
    print(f"Czas dopasowania: {end - processed}")
    print(f"Sumaryczny czas działania: {end - start}")
```

Poniżej znajduje się kod funkcji ogólnej prównującej czasy wszystkich algorytmów.

In [14]:
```
def compare_times(text, pattern):
    print("==== ALGORYTM NAIWNY ====")
    get_time_naive(text, pattern)

    print("\n==== AUTOMAT SKOŃCZONY ====")
    get_time_fa(text, pattern)

    print("\n==== ALGORYTM KMP ====")
    get_time_kmp(text, pattern)
```

Zadanie 2. Wyszukanie wzorca "art" w ustawie

W poniższym przykładzie ograniczyłem się do wypisania liczby wystąpień dla czytelności. Aby to zmienić należy zmienić argument False na True.

In [9]:
```
file = open("1997_714.txt",encoding="utf8")
text = file.read().replace("\n", " ")
pattern = "art"

print(naive(text, pattern, False))
print(fa_string_matching(text, transition_table(pattern), False))
print(kmp_string_matching(text, pattern, prefix_function(pattern), False))
```

```
273
273
273
```

Zadanie 3 Porównanie czasów

In [18]:
```
file = open("1997_714.txt",encoding="utf8")
text = file.read().replace("\n", " ")
pattern = "art"
compare_times(text, pattern)
```

```
==== ALGORYTM NAIWNY ====
Czas działania algorytmu: 0.06925649999993766

==== AUTOMAT SKOŃCZONY ====
Czas preprocessingu: 2.4700000033955665e-05
Czas dopasowania: 0.03436059999989993
Sumaryczny czas działania: 0.03438529999993989

==== ALGORYTM KMP ====
Czas preprocessingu: 3.5000000480067683e-06
Czas dopasowania: 0.04379850000093554
Sumaryczny czas działania: 0.04380200000014156
```

Zgodnie z oczekiwaniami najgorzej wypadł algorytm naiwny. Czas preprocessingu automatu skończonego był, zgodnie z tym na co wskazuje włożoność, dłuższy od czasu preprocessingu dla KMP. Należy jednak zwrócić uwagę na fakt, że pomimo iż zarówno AS jak i KMP mają taką samą teoretyczną złożoność dopasowania, to KMP wypadł wyraźnie wolniej, co zaskutkowało najszybszym wykonaniem AS.

Zadanie 4.

In [19]:
```
text = "x"*200000
pattern = "x"*100000
compare_times(text, pattern)
```

```
==== ALGORYTM NAIWNY ====
Czas działania algorytmu: 0.7388263000000279

==== AUTOMAT SKOŃCZONY ====
Czas preprocessingu: 0.6291839999998956
Czas dopasowania: 0.06827180000004773
Sumaryczny czas działania: 0.6974557999999433

==== ALGORYTM KMP ====
Czas preprocessingu: 0.02485490000003665
Czas dopasowania: 0.07111000000008971
Sumaryczny czas działania: 0.09596490000012636
```

Długość wzorca m została dobrana tak, by wartość (n-m)(m) była jak największa (pominięto 1 we wzorze na złożoność czasu dopasowania ze względu na wielokość n). Następnie zwiększono n, tak by różnica była zauważalna. Wzorzec i tekst składa się tylko, ze znaku "x", ponieważ jest to pesymistyczny przypadek dla algorytmu naiwnego i umożliwia efektywne wykorzystanie właściwości algorytmu KMP - wiele znaków można "przeskoczyć".

Zadanie 5.

In [27]:
```
text = "x"*1000
pattern = "qwertyuiopasdfghjklzxcvbnm"
compare_times(text, pattern)
```

```
==== ALGORYTM NAIWNY ====
Czas działania algorytmu: 0.0001889000001192675

==== AUTOMAT SKOŃCZONY ====
Czas preprocessingu: 0.00400500000023384 9
Czas dopasowania: 0.0002288999999913036 46
Sumaryczny czas działania: 0.0042339000001447 86

==== ALGORYTM KMP ====
Czas preprocessingu: 6.6000002334476 45e-06
Czas dopasowania: 0.0001816999997572566
Sumaryczny czas działania: 0.0001882999999070424
```

Aby zwiększyć czas utworzenia tablicy przejścia AS względem funkcji przejści KMP, należy odpowiednio zwiększyć liczbę symboli alfabetu. Należy zwrócić uwagę, że czas preprocessingu KMP zapisany jest w notacji wykładniczej.