Raport z laboratorium 1

Filip Nikolow

7 marca 2021

1 Cel laboratorium

Celem laboratorium było zapoznanie się z trzema algorytmami wyszukiwania wzorców: algorytmem naiwnym, automatem skończonym oraz algorytmem KMP. W ramach laboratorium dokonałem implementacji powyższych algorytmów i dodałem możliwość pomiaru czasu ich działania.

2 Implementacja

Aby zaimplementować algorytmy naiwny oraz automat skończony posiłkowałem się wiadomościami z wykładu, natomiast informacje na temat algorytmu KMP zaczerpnąłem z *Wprowadzenia do Algorytmów* Cormena. W załączonych plikach każdy algorytm znajduje się w osobnym module. Każdy z nich na wejściu przyjmuje pewien tekst oraz poszukiwany wzorzec i zwraca listę poprawnych przesunięć. Każdy algorytm posiada także drugą wersję o nazwie [nazwa]_bench.py z zaimplementowanym pomiarem czasu - te algorytmy zwracają słownik posiadający listę przesunięć ('matches'), oraz czasy działania ('times') - osobno czas inicjalizacji struktur i czas wyszukiwania. Pomiarów dokonuje za pomocą funkcji time z modułu time poprzez odczytanie czasu na początku i końcu mierzonego fragmentu kodu a następnie wzięcie różnicy. Poniżej załączam kod podstawowych wersji tych algorytmów bez pomiaru czasu, kod z pomiarem czasu oraz wywołania załączam dla porządku na końcu tego sprawozdania:

2.1 Algorytm naiwny

```
1  def naive(text, pattern):
2      m = len(pattern)
3      n = len(text)
4      res = []
5      for s in range(n - m + 1):
6          if text[s:s + m] == pattern:
7              res.append(s)
8      return res
9
10
11 if __name__ == '__main__':
12     print(naive("hhhh", "hh"))
13     print(naive("abababab", 'ab'))
```

2.2 Automat skończony

```python
def automaton(text, pattern):

    def generate_delta_function():

        def sigma_function(P):
            k = len(P)
            for i in range(k):
                if P[i:] == pattern[:k - i]:
                    return k - i
            return 0

        alphabet_dict = {}
        for c in pattern:
            alphabet_dict[c] = 0

        delta_f = [alphabet_dict.copy() for i in range(m + 1)]

        for i in range(m + 1):
            for c in alphabet_dict.keys():
                delta_f[i][c] = sigma_function(pattern[:i] + c)
        return delta_f

    m = len(pattern)

    q = 0   # current state
    delta_f = generate_delta_function()
    res = []
    for i, c in enumerate(text):
        q = delta_f[q].get(c, 0)
        if q == m:
            res.append(i - m + 1)
    return res


if __name__ == '__main__':
    print(automaton("hhhh", "hh"))
    print(automaton("abababab", 'ab'))
```

2.3 Algorytm KMP

```python
def kmp(text, pattern):

    def generate_prefix_function():
        prefix_function = [0] * m
        k = 0
        for q in range(1, m):
            while k > 0 and pattern[k] != pattern[q]:
                k = prefix_function[k - 1]
            if pattern[k] == pattern[q]:
                k += 1
            prefix_function[q] = k
        return prefix_function

    m = len(pattern)
    prefix_function = generate_prefix_function()
    res = []
    q = 0
    for i, c in enumerate(text):
        while q > 0 and pattern[q] != c:
            q = prefix_function[q - 1]
        if pattern[q] == c:
            q += 1
        if q == m:
            res.append(i - m + 1)
            q = prefix_function[q - 1]
    return res


if __name__ == '__main__':
    print(kmp("hhhh", "hh"))
    print(kmp("abababab", 'ab'))
```

3 Realizacja poleceń

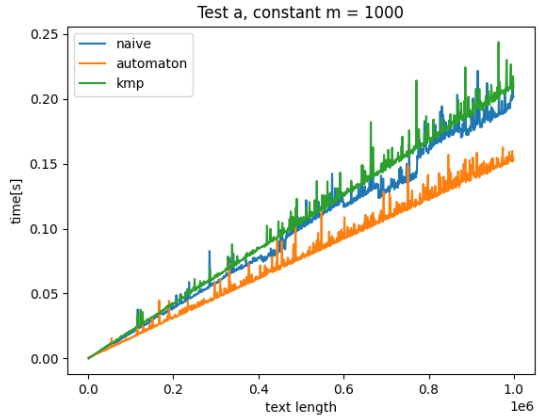
3.1 Testy porównujące szybkość działania algorytmów

Do testów porównawczych wymyśliłem dwa wzorce:

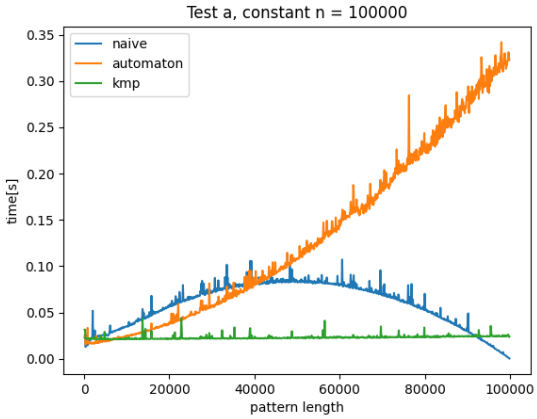
a) wzorzec: "a"$\cdot m$, tekst: "a"$\cdot n$

b) wzorzec: "a"$\cdot(m-1)+$"b", tekst: ("a"$\cdot(m-2)+$"b")$*n/(m-1)$

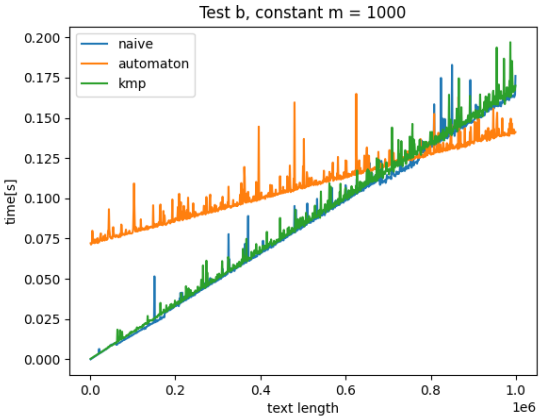
Dla obu wzorców przeprowadziłem po dwa testy, jeden ze stałym m, drugi ze stałym n. Oto wyniki pomiarów:



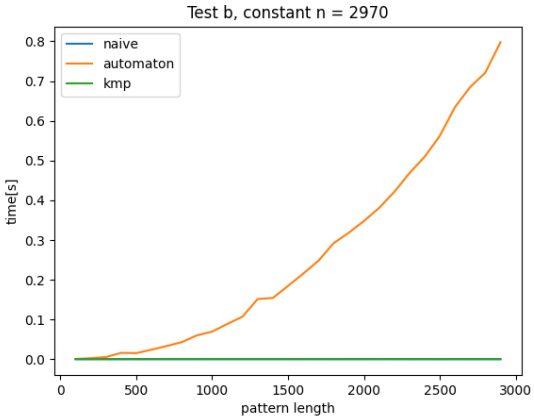
Rys. 1



Rys. 2



Rys. 3



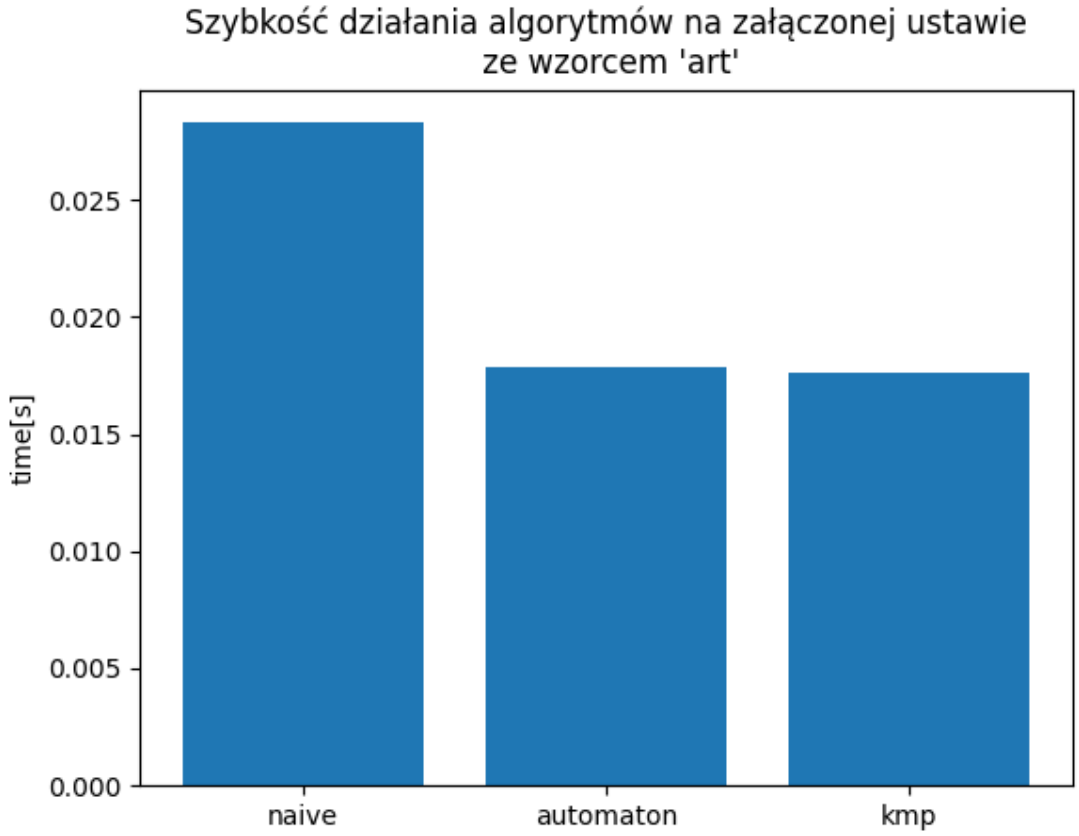
Rys. 4

Jak widać na rysunku 1, czas działania każdego z algorytmów rośnie liniowo ze wzrostem długości tesktu. Dla wybranego m = 1000 algorytm naiwny działa zadziwiająco dobrze - najprawdopodobniej jest to kwestia optymalizacji pythona, a konkretniej optymalizacja porównywania dwóch stringów/tablic: z tego co udało mi się znaleźć w sieci, python takie porównanie wykonuje za pomocą memcmp(), która jest znacznie szybsza niż porównanie w pętli w pythonie (w wersji z pomiarem czasu jest zakomentowany kod, który wykonuje porównanie tablic ale w pętli for, czyli rezygnując z usprawnień pythona - taki kod wykonuje się rzędy wielkości wolniej). Na rysunku 2 możemy zauważyć kwadratowy charakter algorytmu naiwnego - maksimum czasu występuje gdy długość wzorca jest równa mniej więcej połowie długości tekstu. Dodatkowo widać, że złożoność implementacji funkcji generującej tablicę przejść automatu nie ma złożoności liniowej względem długości wzorca, chociaż akurat wzorzec o zbiorze liter alfabetu równym jeden jest przypadkiem optymistycznym zaimplementowanej funkcji - wysoką złożoność znacznie lepiej widać na rysunku 3 i 4.

Na podstawie tych testów można wnioskować, że algorytm KMP jest najbardziej stabilny, tzn. działa zawsze w czasie liniowym, algorytm automatu ma najlepszą stałą (mniejsze nachylenie na rysunkach 1 i 3), natomiast algorytm naiwny wcale nie jest taki zły - szczególnie gdy python optymalizuje w nim porównania stringów, podczas gdy pozostałe algorytmy nie mają takich usprawnień, a przynajmniej nie w takim wydaniu. Należy też podkreślić, że automat miałby znacznie lepsze wyniki w zaproponowanych testach, gdyby nie wolna implementacja generowania tablicy przejść - wtedy sensowne byłoby przetestowanie jeszcze czasów wykonania w zależności od ilości różnych znaków we wzorcu, gdyż to jest pesymistyczny przypadek dla automatu.

3.2 Przeszukiwanie ustawy (zad 2 i 3)

Przeszukałem załączoną w poleceniu ustawę za pomocą każdego z algorytmów. Wszystkie zwróciły dokładnie taką samą listę przesunięć znajdując 273 wystąpienia wzorca 'art'. Poniżej załączam porównanie czasów działania algorytmów:



Wyniki nie są szczególnie zaskakujące - algorytm naiwny jest wolniejszy, gdyż często następowało dopasowanie pierwszej litery wzorca ('a'), natomiast KMP jest tak szybki jak automat ponieważ żaden prefix wzorca nie jest jego sufiksem, a więc algorytm nie musi się 'cofać'.

3.3 Pesymistyczny przypadek algorytmu naiwnego (zad 4)

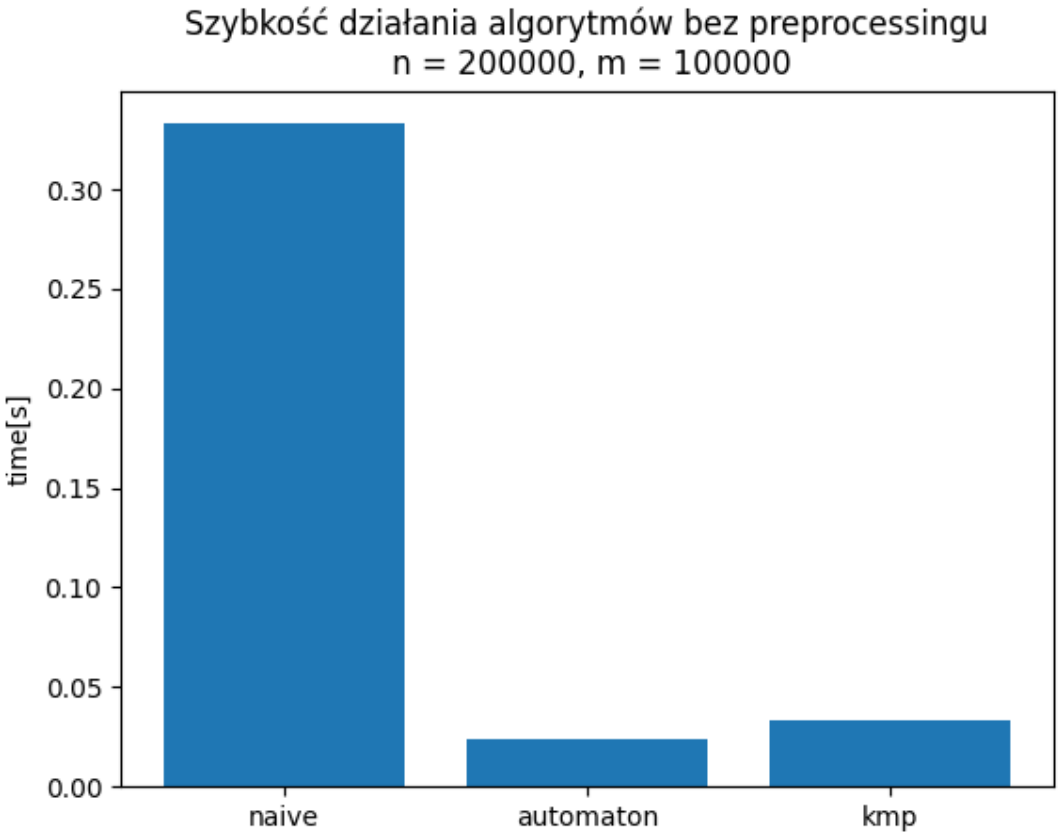
Aby znaleźć tekst i wzorzec dla których algorytm naiwny będzie działać dłużej od automatu i kmp, wystarczy spojrzeć na złozoności obliczeniowe tych algorytmów oraz na rysunek 2:

- Naiwny: $O((n-m+1)m)$
- Automat: $O(n+m\Sigma)$ - (zaimplementowana przeze mnie wersja, zgodnie z ustaleniami z laboratorium, ma złożoność $O(n+m^3\Sigma)$)
- KMP: $O(m+n)$

Przy czym n to długość tekstu, a m - długość wzorca. Pytanie w zadaniu dotyczy jedynie czasu wyszukiwania, z pominięciem czasu preprocessingu, a więc zarówno automat jak i kmp upraszczają swoją złożoność do $O(n)$. Widać więc, że szukamy maksimum wyrażenia $(n-m+1)m$ przy jednocześnie możliwie małym n. Jak widać z rysunku 2 takie maksimum jest dla $n \approx 2 \cdot m$. Trzeba więc jedynie zagwarantować, że rzeczywiście algorytm naiwny będzie wykonywał $\sim (n-m+1)m$ porównań (np. dla wzorca 'b' i tekstu 'a'*n algorytm naiwny

wykona jedynie $(n - m + 1)$ porównań). Takim przypadkiem jest dokładnie "test a" z sekcji 3.1, gdyż każde przesunięcie jest poprawne i algorytm naiwny zawsze musi przyrównywać cały wzorzec. Sprawdźmy więc następujący przykład:

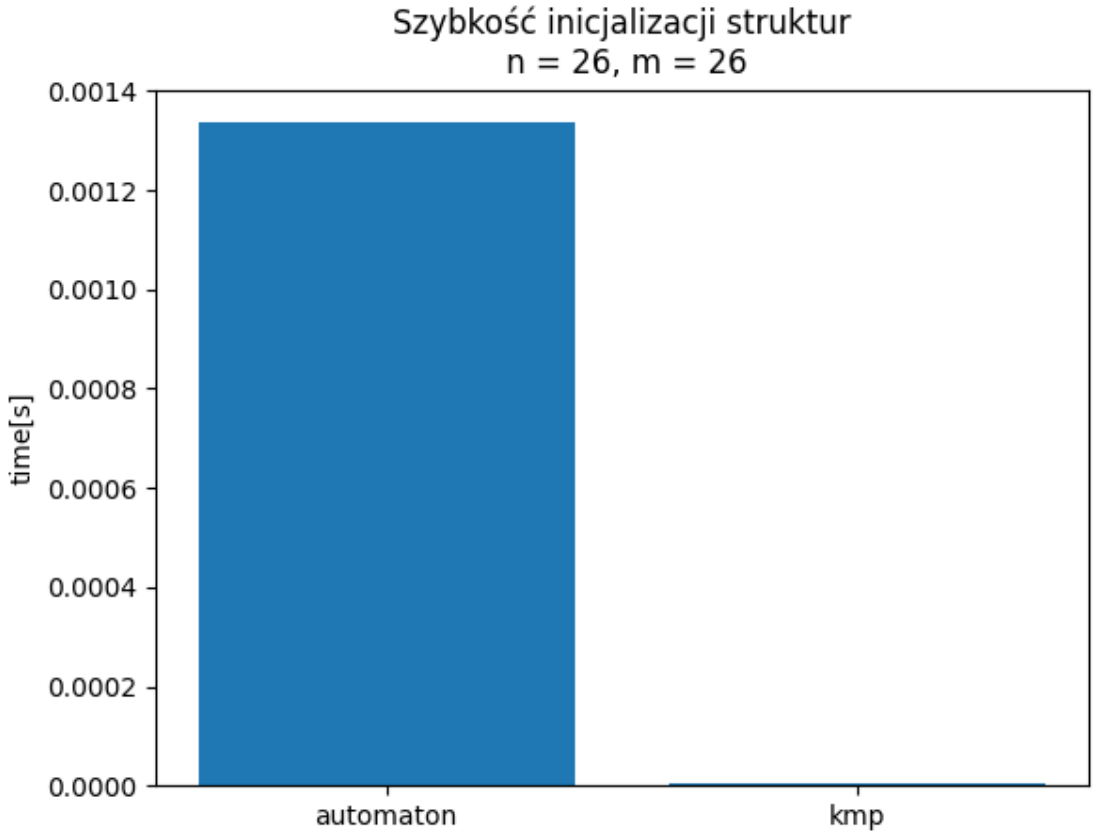
wzorzec: "a"$\cdot 10^5$, tekst: "a"$\cdot 2^{\cdot 5}$



3.4 Pesymistyczny przypadek dla generowania tablicy przejścia automatu skończonego (zad 5)

Jako że złożoność zaimplementowanej funkcji generującej wynosi $O(m^3\Sigma)$, a nie $O(m\Sigma)$, w zasadzie każdy przykład z odpowiednio długim wzorcem i alfabetem przynajmniej dwuliterowym spełni warunek zadania. Jednakże zakładając złożoność generowania funkcji przejścia automatu $O(m\Sigma)$ oraz biorąc złożoność obliczenia funkcji przejścia algorytmu KMP $O(m)$ postępowanie jest jasne: należy wziąć wzorzec w którym każdy znak jest inny. W ten sposób minimalizujemy długość m jednocześnie maksymalizując Σ. Dla porządku poniżej załączam pomiary czasów inicjalizacji struktur dla wzorca złożonego ze wszystkich liter alfabetu angielskiego:

wzorzec: 'qwertyuiopasdfghjklzxcvbnm'



4 Szczegóły techniczne platformy testowej

Wszystkie pomiary czasów zostały przeprowadzone na maszynie wirtualnej VirtualBox z systemem Ubuntu 20. Używana wersja języka Python to 3.8.5. Użyty procesor to AMD Ryzen 3900X.

5 Kod algorytmów z dodanymi pomiarami czasu

5.1 Algorytm naiwny

```
1  from time import time
2
3
4  def naive(text, pattern):
5      t1 = time()
6      m = len(pattern)
7      n = len(text)
8      res = []
9      for s in range(n - m + 1):
10         if text[s:s + m] == pattern:
11             res.append(s)
12         # # Alternative string comparison without using Python optimizations
13         # for i in range(m):
14         #     if text[s + i] != pattern[i]:
```

```
        #         break
        # else:
        #     res.append(s)
    t2 = time()
    return {'matches': res, 'times': {'init_time': 0.0, 'matching_time': t2 - t1}}


if __name__ == '__main__':
    print(naive("hhhh", "hh"))
    print(naive("abababab", 'ab'))
```

5.2 Automat skończony

```
from time import time


def automaton(text, pattern):

    def generate_delta_function():

        def sigma_function(P):
            k = len(P)
            for i in range(k):
                if P[i:] == pattern[:k - i]:
                    return k - i
            return 0

        alphabet_dict = {}
        for c in pattern:
            alphabet_dict[c] = 0

        delta_f = [alphabet_dict.copy() for i in range(m + 1)]
        for i in range(m + 1):
            for c in alphabet_dict.keys():
                delta_f[i][c] = sigma_function(pattern[:i] + c)

        return delta_f

    ti1 = time()
    m = len(pattern)
    delta_f = generate_delta_function()
    ti2 = time()
    tm1 = time()
    q = 0   # current state
    res = []
    for i, c in enumerate(text):
        q = delta_f[q].get(c, 0)
        if q == m:
            res.append(i - m + 1)
    tm2 = time()
    return {'matches': res, 'times': {'init_time': ti2 - ti1, 'matching_time': tm2 - tm1}}
```

```python


if __name__ == '__main__':
    print(automaton("hhhh", "hh"))
    print(automaton("abababab", 'ab'))
```

5.3 Algorytm KMP

```python
from time import time


def kmp(text, pattern):

    def generate_prefix_function():
        prefix_function = [0] * m
        k = 0
        for q in range(1, m):
            while k > 0 and pattern[k] != pattern[q]:
                k = prefix_function[k - 1]
            if pattern[k] == pattern[q]:
                k += 1
            prefix_function[q] = k
        return prefix_function

    ti1 = time()
    m = len(pattern)
    prefix_function = generate_prefix_function()
    ti2 = time()
    tm1 = time()
    res = []
    q = 0
    for i, c in enumerate(text):
        while q > 0 and pattern[q] != c:
            q = prefix_function[q - 1]
        if pattern[q] == c:
            q += 1
        if q == m:
            res.append(i - m + 1)
            q = prefix_function[q - 1]
    tm2 = time()
    return {'matches': res, 'times': {'init_time': ti2 - ti1, 'matching_time': tm2 - tm1}}


if __name__ == '__main__':
    print(kmp("hhhh", "hh"))
    print(kmp("abababab", 'ab'))
```

5.4 Testowanie, generowanie wykresów itp.

```
from naive_bench import naive
from automaton_bench import automaton
from kmp_bench import kmp
import matplotlib.pyplot as plt


def bench(text, pattern, p=True):
    naive_res = naive(text, pattern)
    automaton_res = automaton(text, pattern)
    kmp_res = kmp(text, pattern)
    if naive_res['matches'] != automaton_res['matches'] or automaton_res['matches'] != kmp_res[
            'matches']:
        print('Matches differ!!!')
        exit()
    if p:
        print('Match count:', len(naive_res['matches']))
        print('naive:', naive_res['times'])
        print('automaton:', automaton_res['times'])
        print('kmp:', kmp_res['times'])
    return naive_res, automaton_res, kmp_res


# Zadanie 1
print('=================Zad1==================')
# Test a, constant m
samples = 1000
s = 1000
text = 'a' * s
pattern = 'a' * s
dnaive, dauto, dkmp = [], [], []
size = []
for i in range(1, samples):
    print(i)
    n, a, k = bench(text * i, pattern, False)
    size.append(i * s)
    dnaive.append(sum(n['times'].values()))
    dauto.append(sum(a['times'].values()))
    dkmp.append(sum(k['times'].values()))
plt.plot(size, dnaive, label='naive')
plt.plot(size, dauto, label='automaton')
plt.plot(size, dkmp, label='kmp')
plt.legend(loc="upper left")
plt.xlabel('text length')
plt.ylabel('time[s]')
plt.title('Test a, constant m = ' + str(len(pattern)))
plt.show()

# Test a, constant n
samples = 1000
s = 100
```

```
51  text = 'a' * s * samples
52  pattern = 'a' * s
53  dnaive, dauto, dkmp = [], [], []
54  size = []
55  for i in range(1, samples):
56      print(i)
57      n, a, k = bench(text, pattern * i, False)
58      size.append(i * s)
59      dnaive.append(sum(n['times'].values()))
60      dauto.append(sum(a['times'].values()))
61      dkmp.append(sum(k['times'].values()))
62  plt.plot(size, dnaive, label='naive')
63  plt.plot(size, dauto, label='automaton')
64  plt.plot(size, dkmp, label='kmp')
65  plt.legend(loc="upper left")
66  plt.xlabel('pattern length')
67  plt.ylabel('time[s]')
68  plt.title('Test a, constant n = ' + str(len(text)))
69  plt.show()
70
71  # Test b, constant m
72  samples = 1000
73  s = 1000
74  text = ('a' * (s - 2) + 'b')
75  pattern = 'a' * (s - 1) + 'b'
76  dnaive, dauto, dkmp = [], [], []
77  size = []
78  for i in range(1, samples):
79      print(i)
80      n, a, k = bench(text * i, pattern, False)
81      size.append(i * s)
82      dnaive.append(sum(n['times'].values()))
83      dauto.append(sum(a['times'].values()))
84      dkmp.append(sum(k['times'].values()))
85  plt.plot(size, dnaive, label='naive')
86  plt.plot(size, dauto, label='automaton')
87  plt.plot(size, dkmp, label='kmp')
88  plt.legend(loc="upper left")
89  plt.xlabel('text length')
90  plt.ylabel('time[s]')
91  plt.title('Test b, constant m = ' + str(len(pattern)))
92  plt.show()
93
94  # Test b, constant n
95  samples = 30
96  s = 100
97  text = ('a' * (s - 2) + 'b') * samples
98  pattern = 'a' * (s - 1) + 'b'
99  dnaive, dauto, dkmp = [], [], []
100 size = []
101 for i in range(1, samples):
102     print(i)
103     n, a, k = bench(text, pattern * i, False)
```

```
104     size.append(i * s)
105     dnaive.append(sum(n['times'].values()))
106     dauto.append(sum(a['times'].values()))
107     dkmp.append(sum(k['times'].values()))
108 plt.plot(size, dnaive, label='naive')
109 plt.plot(size, dauto, label='automaton')
110 plt.plot(size, dkmp, label='kmp')
111 plt.legend(loc="upper left")
112 plt.xlabel('pattern length')
113 plt.ylabel('time[s]')
114 plt.title('Test b, constant n = ' + str(len(text)))
115 plt.show()
116
117 # Zadanie 2,3
118 print('=================Zad2,3==================')
119 with open("ustawa.txt") as f:
120     text = f.read()
121     pattern = 'art'
122     n, a, k = bench(text, pattern)
123     print('n,a,k')
124     plt.bar(['naive', 'automaton', 'kmp'],
125             [sum(n['times'].values()),
126              sum(a['times'].values()),
127              sum(k['times'].values())])
128     plt.ylabel('time[s]')
129     plt.title('Szybkość działania algorytmów na załączonej ustawie\n ze wzorcem \'art\'')
130     plt.show()
131
132 # Zadanie 4
133 print('=================Zad4==================')
134 pattern = "a" * 1000000
135 text = pattern * 2
136 n, a, k = bench(text, pattern)
137 plt.bar(
138     ['naive', 'automaton', 'kmp'],
139     [n['times']['matching_time'], a['times']['matching_time'], k['times']['matching_time']])
140 plt.ylabel('time[s]')
141 plt.title('Szybkość działania algorytmów bez preprocessingu\n n = ' + str(len(text)) +
142           ', m = ' + str(len(pattern)))
143 plt.show()
144
145 # Zadanie 5
146 print('=================Zad5==================')
147 pattern = 'qwertyuiopasdfghjklzxcvbnm'
148 text = pattern
149 n, a, k = bench(text, pattern)
150 plt.bar(['automaton', 'kmp'], [a['times']['init_time'], k['times']['init_time']])
151 plt.ylabel('time[s]')
152 plt.title('Szybkość inicjalizacji struktur\n n = ' + str(len(text)) + ', m = ' +
153           str(len(pattern)))
154 plt.show()
```