AISD lista 9

Dominik Szczepaniak

July 30, 2024

1 Zadanie 1

Idea: Chcemy tylko powiedzieć czy dany pattern występuje w tekście. Podzielmy więc nasz pattern tak, że mamy $a_1, a_2, a_3, ..., a_n$ gdzie a_k to część patternu która jest oddzielona gap characterami od sąsiednich części. Czyli np. $ab \circ cde \circ gj$ byłoby $a_1 = ab, a_2 = cde, a_3 = gj$. Teraz wystarczy, że najpierw znajdziemy pierwsze wystąpienie a_1, później zaczynając w miejscu gdzie skończyło się a_1 zaczniemy szukać a_2 itd. Wszystkie litery pomiędzy a_i i a_{i+1} będą należec do gap characteru.

Algorytm:

```
a - tablica z patternami podzielonymi juz na podpatterny (ktore sa oddz
m - dlugosc patternu
x - dlugosc tekstu
t - tekst

def pi(s):
    n = s.length
    pi = [0] * n
    j = 0
    for i in range(1, n+1):
        j = pi[i-1]
        while j > 0 && s[i] != s[j]:
            j = pi[j-1]
        if s[i] == s[j]:
            j++
        pi[i] = j
```

```
    return pi

def main():
    start = m+1
    jest = [false] * a.lenght
    j = 0
    for s in a:
        newt = s + "#" + t
        tab = pi(newt)
        for i in range(start, x+m):
            if(tab[i] == s.lenght):
                start = i+1
                jest[j] = true
                j++
                break
    for i in jest:
        if i == False:
            return False
    return True
```

Dowód:

Dowód w zasadzie wynika z konstrukcji, szukamy po kolei podpatternów i sprawdzamy czy istnieją, a znaki które są pomiędzy wpisujemy jako gap charactery. Jeżeli dla pierwszego wystąpienia a_i nie znaleźliśmy odpowiedzi, to tym bardziej nie znajdziemy dla jakiegoś następnego wystąpienia a_i, bo mamy te same znaki dalej, ale odcieliśmy jakąś część tekstu od tego do poprzedniego wystąpienia.

Złożoność:

Dla każdego podpatternu wykonujemy KMP, podpatternów może być m, złożoność KMP to n, więc mamy złożoność O(m*n).

Pamięciowa:

O(n) - usuwamy tablice z funkcją KMP za każdym razem.

2 Zadanie 2

Idea: Chcemy w czasie liniowym odpowiedzieć czy napis T jest równy napisowi T' z przesunięciem cyklicznym. W takim razie jeśli zrobimy T = T+T oraz T' będzie występować w nowym T, to będzie to prawda.

Algorytm:

```
def pi(s):
    j = 0
    pi = [0] * (n+1)
    n = s.length
    for i in range(1, n+1):
        j = pi[i-1]
        while(j > 0 && s[i] != s[j]):
            j = pi[j-1]
        if(s[i] == s[j]):
            j++
        pi[i] = j

def main(s1, s2):
    s1 = s1+s1
    t = s2 + "#" + s1
    tab = pi(t)
    for i in range(s2.length+2, t.length):
        if tab[i] == s2.length:
            return True
    return False
```

Dowód:

Załóżmy, że T' występuje w T+T, ale T' nie jest przesunięciem cyklicznym T. W takim razie jeśli T' występuje w T+T, to musi zaczynać się w jakimś znaku i oraz kończyć w znaku j. W takim razie wróćmy do początkowego słowa T i zacznijmy sprawdzać od znaku i. Gdy dojdziemy do końca do przejdziemy do początku napisu i pójdziemy aż do i-1 i otrzymamy dokładnie napis T', ponieważ tylko w taki sposób mógło powstać T+T. Mamy więc sprzeczność, że T' nie jest przesunięciem cyklicznym słowa T.

Złożoność:

Funkcja pi ma złożoność O(n), więc na wykonana na podwojonym napisie, o długości 2*n, dalej będzie to O(n).

Pamięciowa: O(n)

3 Zadanie 3

Idea: automat albo idzie do następnej litery jeśli pasuje albo idzie do najdłuższego prefikso-sufiksu i kolejnej litery (czyli wyniku automatu dla najdluzszego prefikso-sufiksu). Najdłuższy prefikso-sufiks znajdziemy za pomocą tablicy z KMP.

Algorytm:

```
prefikso-sufiks(s):
    n = len(s)
    pi = [0]*n
    for i in range(1, n):
        j = pi[i-1]
        while j>0 && s[i] != s[j]:
            j = pi[j-1]
        if s[i] == s[j]:
            j++
        pi[i] = j
    return pi

def automata(s):
    n = len(s)
    pi = prefikso-sufiks(s)
    aut = [[0]*alfabet for _ in range(n+1)]
    for i in range(n):
        for j in range(alfabet):
            if j == s[i]:
                aut[i][j] = i+1
            else:
                aut[i][j] = aut[pi[i-1]][j]
```

Analiza złożoności:

Tablica prefikso-sufiksowa buduje się w O(m), a automat w O(m*alfabet). Zlozonosc pamieciowa to O(m*alfabet + m)

Dowód:

Dowód wynika z konstrukcji algorytmu.

4 Zadanie 4

https://cs.stackexchange.com/questions/10990/colour-a-binary-tree-to-be-a-red-black-tree