Algorytmy Tekstowe

Tomasz Waleń

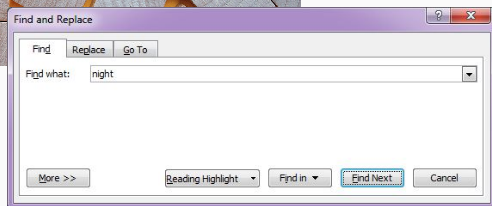
MIM UW

Algorytmy i Struktury Danych, 2018–12–03

slajdy: `http://bit.ly/aisd-20181203`

Licencja: Creative Commons Attribution-NonCommercial

Problem wyszukiwania wzorca



Problem wyszukiwania wzorca

Definicja

Dla tekstu T i wzorca P ($|T| = n$, $|P| = m$)
wyznacz wszystkie wystąpienia wzorca w tekście:

$Occ(P, T) = \{i :$ wzorzec P występuje w tekście T na pozycji $i\}$

Problem wyszukiwania wzorca

Definicja

Dla tekstu T i wzorca P ($|T| = n$, $|P| = m$) wyznacz wszystkie wystąpienia wzorca w tekście:

$$Occ(P, T) = \{i : T[i, \ldots, i + m - 1] = P\}$$

Przykład

P: abaaba

```
    0 1 2 3 4 5 6 7 8 9 10  12  14  16  18  20  22
T:  a b a b a a b a a b a b a a b a a a b a a b a a
        a b a a b a     a b a a b a  a b a a b a
              a b a a b a
```

Occ: 2, 5, 10, 17

Algorytm naiwny

Algorytm 1: NaiveSearch

Dane: T – tekst, P – wzorzec

Wynik: pozycje w tekście w T na których występuje wzorzec P

1 $Occ := \emptyset$

2 **foreach** $i \in \{0, \ldots, |T| - |P|\}$ **do**

3   **if** $T[i, \ldots, i + |P| - 1]$=$P$ **then**

        ▷ `zgłoś wystąpienie` P `na pozycji` $T[i]$

4         $Occ := Occ + \{i\}$

5 **return** Occ

Algorytm naiwny - analiza

Lemat

W pesymistycznym przypadku algorytm naiwny wykonuje $O(nm)$ operacji.

Dowód.

Dla $T = \mathtt{a}^{n+1+m}$ i $P = \mathtt{a}^m\mathtt{b}$, algorytm naiwny stara się dopasować wzorzec na n pozycjach, niestety każde dopasowanie wymaga $m + 1$ porównań.
Stąd algorytm wykonuje $O(nm)$ operacji. □

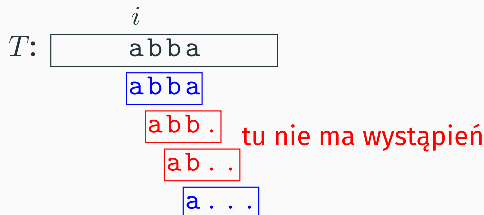
Gdzie tkwi nieefektywność algorytmu naiwnego?

Za każdym razem przesuwamy obliczenia tylko o jedną pozycję w prawo i każde kolejne dopasowanie ignoruje informację otrzymaną we wcześniejszych krokach.

Gdzie tkwi nieefektywność algorytmu naiwnego?

Za każdym razem przesuwamy obliczenia tylko o jedną pozycję w prawo i każde kolejne dopasowanie ignoruje informację otrzymaną we wcześniejszych krokach.

Przykład

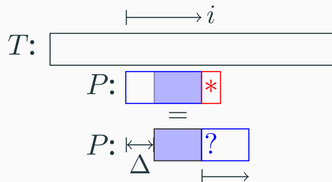


Jak mógłby wyglądać lepszy algorytm?

Założenia szybszego algorytmu

1. maksymalne wykorzystanie zdobytej wiedzy – nigdy nie cofamy się w tekście,
2. sprytniejsze kolejne dopasowania (Δ) – nie mogą być za małe (żeby algorytm nie był zbyt wolny), ani zbyt duże (żeby nie zgubić żadnego wystąpienia).

Przykład



Proste definicje

Dla słowa $w = a_0, \ldots, a_{n-1}$:

- (właściwym) prefiksem słowa w nazywamy słowa postaci $\mathrm{pref}(w, \ell) = a_0, \ldots, a_{\ell-1}$, gdzie $0 \leq \ell < n$,
- (właściwym) sufiksem słowa w nazywamy słowa postaci $\mathrm{suf}(w, \ell) = a_{n-\ell}, \ldots, a_{n-1}$, gdzie $0 \leq \ell < n$,
- puste słowo ϵ jest zarówno prefiksem jak i sufiksem dowolnego niepustego słowa.

Przykład

$w =$ `alamakota`

- prefiksy: ϵ, `a`, `al`, `ala`, $\ldots$
- sufiksy: ϵ, `a`, `ta`, `ota`, `kota`, $\ldots$

A może przedefiniujemy problem?

Definicja

Dla tekstu T i wzorca P ($|T| = n$, $|P| = m$)
należy wyznaczyć tablicę L długości n:

$$L[i] = \max\{j : \mathrm{pref}(P, j) \text{ jest sufiksem } T[0, \ldots, i]\}$$

A może przedefiniujemy problem?

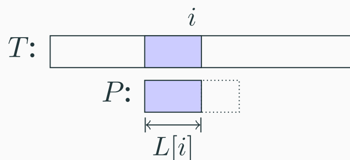
Definicja

Dla tekstu T i wzorca P ($|T| = n$, $|P| = m$)
należy wyznaczyć tablicę L długości n:

$$L[i] = \max\{j : \mathrm{pref}(P, j) \text{ jest sufiksem } T[0, \ldots, i]\}$$

A prościej?

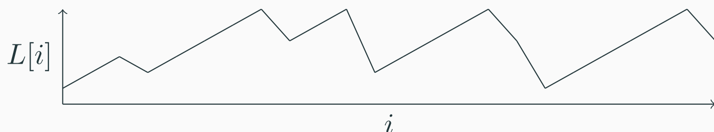
Dla każdej pozycji $T[i]$ w tekście chcemy się wiedzieć jaki jest **najdłuższy** (początkowy) fragment wzorca który kończy się na $T[i]$.



Rozszerzony problem – przykład

P: a b a a b a

	0	1	2	3	4	5	6	7	8	9	10		12		14		16		18		20		22	
T:	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	a	a	b	a	a	b	a	a
$L[i]$:	1	2	3	2	3	4	5	**6**	4	5	**6**	2	3	4	5	**6**	4	1	2	3	4	5	**6**	4
														a	b	a	a	b	a					

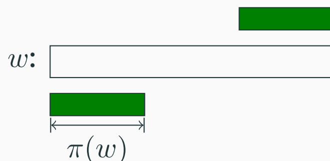


Funkcja prefikso-sufiksowa

Definicja

$$\pi(w) = \max\{\ell : \mathrm{pref}(w, \ell) = \mathrm{suf}(w, \ell)\}$$

Długość najdłuższego prefiksu w, który jest jednocześnie sufiksem.
Dodatkowo definiujemy $\pi(\epsilon) = \epsilon$.

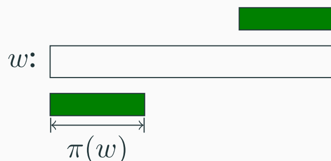


Funkcja prefikso-sufiksowa

Definicja

$$\pi(w) = \max\{\ell : \mathrm{pref}(w, \ell) = \mathrm{suf}(w, \ell)\}$$

Długość najdłuższego prefiksu w, który jest jednocześnie sufiksem.
Dodatkowo definiujemy $\pi(\epsilon) = \epsilon$.



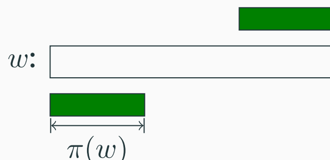
Ale właściwie dlaczego?

Funkcja prefikso-sufiksowa

Definicja

$$\pi(w) = \max\{\ell : \mathrm{pref}(w, \ell) = \mathrm{suf}(w, \ell)\}$$

Długość najdłuższego prefiksu w, który jest jednocześnie sufiksem.
Dodatkowo definiujemy $\pi(\epsilon) = \epsilon$.



Ale właściwie dlaczego?
– ta funkcja mówi o ile można przesunąć obliczenia!

Prefikso-sufiks – przykłady

`ab` (sufiks)	`ababa` (sufiks)	(pusty sufiks)
w: `abaaaab`	w: `abababa`	w: `ababcde`
`ab` (prefiks)	`ababa` (prefiks)	(pusty prefiks)
$\pi(w) = 2$	$\pi(w) = 5$	$\pi(w) = 0$

Definicja tablicy prefikso-sufiksów

A co by było, gdyby policzyć wartości funkcji π dla wszystkich prefiksów słowa?

Definicja

Dla słowa w definiujemy tablicę prefikso-sufiksów:

$$\pi_w[\ell] = \pi(\mathrm{pref}(w, \ell))$$

Dla $0 \leq \ell \leq |w|$.

Tablica prefikso-sufiksów – przykład

Dla $w = \texttt{abaaba}$:

ℓ	$\pi[\ell]$	
0	0	-
1	0	$\pi(\texttt{a}) = 0$ (ϵ)
2	0	$\pi(\texttt{ab}) = 0$ (ϵ)
3	1	$\pi(\texttt{aba}) = 1$ (a)
4	1	$\pi(\texttt{abaa}) = 1$ (a)
5	2	$\pi(\texttt{abaab}) = 2$ (ab)
6	3	$\pi(\texttt{abaaba}) = 3$ (aba)

Tablica prefikso-sufiksów – przykład

Dla $w = \texttt{abaaba}$:

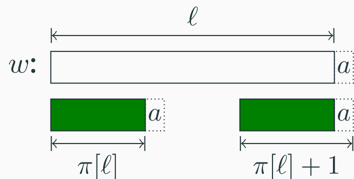
ℓ	$\pi[\ell]$	
0	0	-
1	0	$\pi(\texttt{a}) = 0$ (ϵ)
2	0	$\pi(\texttt{ab}) = 0$ (ϵ)
3	1	$\pi(\texttt{aba}) = 1$ (a)
4	1	$\pi(\texttt{abaa}) = 1$ (a)
5	2	$\pi(\texttt{abaab}) = 2$ (ab)
6	3	$\pi(\texttt{abaaba}) = 3$ (aba)

Ale jak efektywnie liczyć tablicę π?

Tablica prefikso sufiksów - przypadki

Jeśli $\ell > 1$ i $w[\pi_w[\ell]] = w[\ell]$, to

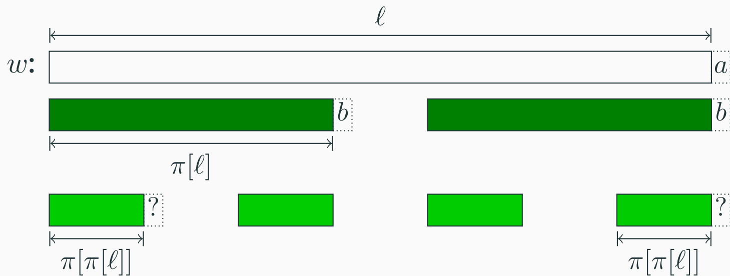
$$\pi_w[\ell + 1] = \pi_w[\ell] + 1$$



Tablica prefikso sufiksów - przypadki

Jeśli $\ell > 1$ i $w[\pi_w[\ell]] \neq w[\ell]$, to

$$\pi_w[\ell + 1] \leq \pi_w[\pi_w[\ell]]$$



Algorytm wyznaczania tablicy prefikso-sufiksów

Algorytm 2: Compute-π

Dane: P, $|P| = m$

Wynik: tablica $\pi_P[\ell]$

```
1 j = 0
2 π[0] := 0
3 foreach ℓ ∈ {1, ..., m} do
4     while j > 0 and P[ℓ − 1] ≠ P[j] do
5         j := π[j]                         ▷ zmniejsz wartość j
6     if j < ℓ − 1 and P[ℓ − 1] = P[j] then
7         j := j + 1
8     π[ℓ] := j
9 return π
```

Złożoność czasowa Compute-π

Lemat
Złożoność czasowa algorytmu Compute-π wynosi $O(m)$.

Dowód.
Zauważmy, że licznik j może zostać zwiększony co najwyżej m razy (tylko raz dla każdej wartości ℓ). Każda iteracja pętli **while** zmniejsza licznik (co najmniej o 1). Stąd sumaryczna liczba iteracji pętli **while** nie może przekroczyć m. □

Algorytm KMP

Algorytm 3: KMP

Dane: tekst T, wzorzec P, $|T| = n$, $|P| = m$

Wynik: tablica wystąpień Occ

```
1  π = Compute−π(P)
2  ℓ = 0
3  Occ := ∅
4  foreach i ∈ {0, ..., n − 1} do
5      while ℓ > 0 and T[i] ≠ P[ℓ] do
6          ℓ := π[ℓ]
7      if T[i] = P[ℓ] then
8          ℓ = ℓ + 1;
9      if ℓ = m then
           ▷ znaleziono wystąpienie kończące się na T[i]
10         Occ = Occ + {i − m + 1}
11         ℓ = π[ℓ]
12 return Occ
```

Złożoność czasowa algorytmu KMP

Lemat
Złożoność czasowa algorytmu KMP wynosi $O(n+m)$.

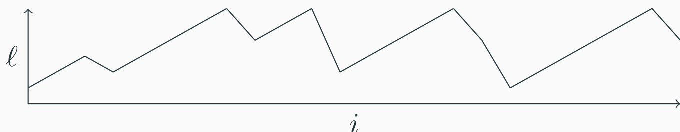
Dowód.
Analogicznie jak w poprzednim dowodzie zauważamy, że licznik ℓ może zostać zwiększony co najwyżej n razy. Każda iteracja pętli **while** zmniejsza licznik (co najmniej o 1). Stąd sumaryczna liczba iteracji pętli **while** nie może przekroczyć n. □

Algorytm KMP – przykład

P: a b a a b a

	0	1	2	3	4	5	6	7	8	9	10		12		14		16		18		20		22
T:	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	a	a	b	a	a	b	a	a
ℓ:	1	2	3	2	3	4	5	**6**	4	5	**6**	2	3	4	5	**6**	**4**	1	2	3	4	5	**6**	4

a b a a b a



Problem wyszukiwania wzorca – ciekawostki

- algorytm KMP jest bardzo tolerancyjny jeśli chodzi o alfabet (może być dowolny, nawet nie musi być sortowalny) i definicję równości słów – można go zaadoptować nawet do bardzo egzotycznych problemów,
- w praktycznych zastosowaniach zamiast KMP używa się szybszych algorytmów, np. `grep` używa wariantu algorytmu Boyera-Moora,
- problem wyszukiwania wzorca można rozwiązać w czasie $O(n + m)$ i dodatkowej pamięci $O(1)$,
- dzięki rozszerzeniom SIMD w nowoczesnych procesorach, wiele standardowych algorytmów można zaimplementować znacznie efektywniej (zwłaszcza dla krótkich wzorców i małych alfabetów)

Indeksowanie tekstów

Indeksowanie tekstów

Definicja

Dla tekstu T należy przygotować strukturę danych, która umożliwia zapytania:

$OccQuery(P)$: lista wystąpień wzorca P w tekście T.

Indeksowanie tekstów – rozwiązanie

Definicja

Dla tekstu T, *tablica sufiksowa* (Suffix Array / SA) zawiera *uporządkowanie leksykograficzne wszystkich sufiksów*.

$$SA[i] = \{j : \text{t. że } T[j, \ldots] \text{ jest } i\text{-tym co do rangi sufiksem}\}$$

Zauważmy, że dzięki uporządkowaniu, pozycje w tekście zaczynające się od podobnych słów będą w tablicy sufiksowej blisko siebie.

Tablice sufiksowe pełnią podobną rolę, jak drzewa sufiksowe. Często stosuje się je zamiennie.

Przykład

	0	1	2	3	4	5	6	7	8	9	10
T:	m	i	s	s	i	s	s	i	p	p	i

p	$T[p,\ldots]$
0	mississippi
1	ississippi
2	ssissippi
3	sissippi
4	issippi
5	ssippi
6	sippi
7	ippi
8	ppi
9	pi
10	i

i	$SA[i]$	$T[SA[i],\ldots]$
0	10	i
1	7	ippi
2	4	issippi
3	1	ississippi
4	0	mississippi
5	9	pi
6	8	ppi
7	6	sippi
8	3	sissippi
9	5	ssippi
10	2	ssissippi

Zastosowania tablicy sufiksowej

- wyszukanie wzorca $Occ(P)$ – w czasie $O(|P| + |Occ| + \log|T|)$,
- zliczanie liczby różnych podsłów – w czasie $O(|T|)$,
- najdłuższe wspólne podsłowo – w czasie $O(|T_1| + |T_2|)$,
- LZ-faktoryzacja (używana do kompresji ZIP),
- rozwiązanie jednego z zadań laboratoryjnych 🙂
- i wiele wiele innych.

Zastosowania tablicy sufiksowej – ale jak?

Niestety potrzebujemy dodatkowych struktur danych:

- tablicy LCP,
- tablicy SA^{-1},
- często również struktury RMQ.

lcp - najdłuższy wspólny prefiks

Definicja

$\mathrm{lcp}(X, Y) =$ długość najdłuższego wspólnego prefiksu X i Y

X: x

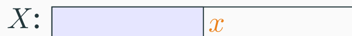
Y: y

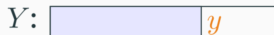
$\mathrm{lcp}(X, Y)$

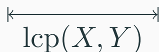
lcp - najdłuższy wspólny prefiks

Definicja

$\mathrm{lcp}(X, Y) =$ długość najdłuższego wspólnego prefiksu X i Y







Przykład

$\mathrm{lcp}(\texttt{alamakota}, \texttt{alamapsa}) = 5$

Tablica LCP

Definicja

Tablica LCP zawiera informacje o wartościach lcp dla pomiędzy sąsiednimi sufiksami w tablicy sufiksowej.

$$\mathrm{LCP}[i] = \mathrm{lcp}(T[\mathrm{SA}[i-1], \ldots], T[\mathrm{SA}[i], \ldots])$$

Dla $i = 0$, $\mathrm{LCP}[i]$ nie jest zdefiniowane.

Tablica LCP – przykład

i	$SA[i]$	$T[SA[i], \ldots]$	$LCP[i]$
0	10	i	
1	7	ippi	1
2	4	issippi	1
3	1	ississippi	**4**
4	0	mississippi	0
5	9	pi	0
6	8	ppi	1
7	6	sippi	0
8	3	sissippi	2
9	5	ssippi	1
10	2	ssissippi	3

SA^{-1} – funkcja odwrotna do SA

$$\mathrm{SA}^{-1}[p] = \mathrm{RANK}[p] = i \text{ wtw } \mathrm{SA}[i] = p$$

0 1 2 3 4 5 6 7 8 9 10
T: m i s s i s s i p p i

$\mathbf{SA^{-1}[2] = 10}$

i	$SA[i]$	$T[SA[i], \ldots]$
0	10	i
1	7	ippi
2	4	issippi
3	1	ississippi
4	0	mississippi
5	9	pi
6	8	ppi
7	6	sippi
8	3	sissippi
9	5	ssippi
10	2	ssissippi

Ale jak efektywnie wyznaczyć SA?

Zacznijmy od budżetu i rachunków:

- naszym celem jest złożoność $O(n)$,
- idealnie byłoby gdyby algorytm nie był zbyt zależny od alfabetu (ale zakładamy, że alfabet jest sortowalny w $O(n)$),
- jeśli koncentrujemy się na algorytmach rekurencyjnych to następująca funkcja wygląda interesująco:

$$T(n) = T\left(\frac{2n}{3}\right) + O(n)$$

- oczywiście $T(n) = O(n)$.

Algorytm KS - Trochę historii

Juha Kärkkäinen, Peter Sanders,
"Simple linear work suffix array construction"
ICALP 2003
~530 cytowań + ~350 (wersja czasopismowa: J. of ACM 2006)

Ciekawostka:
praca zawiera implementację algorytmu (~50 wierszy w C)

Algorytm KS - Szkic

Algorytm 4: ComputeSA

Dane: słowo T, $|T| = n$

Wynik: tablica sufiksowa SA słowa T

1 podziel sufiksy słowa T na trzy kategorie S_0, S_1, S_2
  ▷ $|S_i| \approx n/3$

2 wygeneruj słowo X reprezentujące sufiksy z S_1 i S_2
  ▷ $|X| \approx 2n/3$

3 $\mathrm{SA}_X = \mathrm{ComputeSA}(X)$

4 $\mathrm{SA}_{1,2} =$ wyznacz z SA_X kolejność leks. sufiksów z S_1 i S_2

5 $\mathrm{SA}_0 =$ wyznacz z SA_X i T kolejność leks. sufiksów z S_0

6 $\mathrm{SA} =$ scal $\mathrm{SA}_{1,2}$ i SA_0 ▷ `ta część jest dosyć sprytna`

7 **return** SA

Algorytm KS - Analiza złożoności czasowe

Lemat
Algorytm KS ma złożoność czasową $O(n)$

Dowód.
Zakładając, że uda nam się wykonać wszystkie kroki związane z obliczaniem słowa X, tablic $\mathrm{SA}_{1,2}$ i SA_0 oraz ich scalaniem, w czasie $O(n)$. Otrzymujemy następujące równanie rekurencyjne:

$$T(n) = O(n) + T(2n/3)$$

co rzeczywiście, daje nam rozwiązanie $T(n) = O(n)$. □

Podział sufiksów na S_0, S_1 i S_2

Drobna uwaga techniczna

W zależności od wartości $|T| \mod 3$, uzupełniamy T o 1, 2 lub 3 znaki $\$$ (zakładamy, że znak $\$$ jest mniejszy od dowolnego innego).

Podział sufiksów na S_0, S_1 i S_2

Drobna uwaga techniczna

W zależności od wartości $|T| \mod 3$, uzupełniamy T o 1, 2 lub 3 znaki $\$$ (zakładamy, że znak $\$$ jest mniejszy od dowolnego innego).

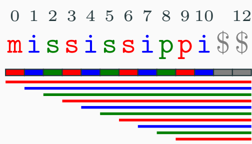
Zazwyczaj najprostsze pomysły są najlepsze, więc po prostu definiujemy:

$$S_j = \{i : i < |T| \text{ oraz } i \mod 3 = j\}$$

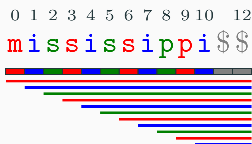
$$S_0 = \{0, 3, 6, 9\}$$

$$S_1 = \{1, 4, 7, 10\}$$

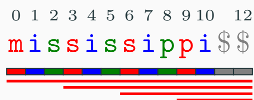
$$S_2 = \{2, 5, 8\}$$



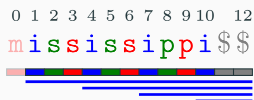
Podział sufiksów na S_0, S_1 i S_2



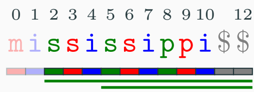
$S_0 = \{0, 3, 6, 9\}$



$S_1 = \{1, 4, 7, 10\}$



$S_2 = \{2, 5, 8\}$



Algorytm KS - słowo X

Algorytm 5: ComputeSA

Dane: słowo T, $|T| = n$

Wynik: tablica sufiksowa SA słowa T

1 podziel sufiksy słowa T na trzy kategorie S_0, S_1, S_2
  ▷ $|S_i| \approx n/3$

2 wygeneruj słowo X reprezentujące sufiksy z S_1 i S_2
  ▷ $|X| \approx 2n/3$

3 $\mathrm{SA}_X = \mathrm{ComputeSA}(X)$

4 $\mathrm{SA}_{1,2} =$ wyznacz z SA_X kolejność leks. sufiksów z S_1 i S_2

5 $\mathrm{SA}_0 =$ wyznacz z SA_X i T kolejność leks. sufiksów z S_0

6 $\mathrm{SA} =$ scal $\mathrm{SA}_{1,2}$ i SA_0

7 **return** SA

Słowo X reprezentujące sufiksy S_1 i S_2

$S_1 = \{1, 4, 7, 10\}$

0 1 2 3 4 5 6 7 8 9 10 12

m i s s i s s i p p i $ $

$S_2 = \{2, 5, 8\}$

0 1 2 3 4 5 6 7 8 9 10 12

m i s s i s s i p p i $ $

0 1 2 3 4 5 6 7 8 9 10 12 14 16 18 20

X': | i s s | i s s | i p p | i $ $ | s s i | s s i | p p i |

Słowo X reprezentujące sufiksy S_1 i S_2

$S_1 = \{1, 4, 7, 10\}$

0 1 2 3 4 5 6 7 8 9 10 12
m i s s i s s i p p i $ $

$S_2 = \{2, 5, 8\}$

0 1 2 3 4 5 6 7 8 9 10 12
m i s s i s s i p p i $ $

0 1 2 3 4 5 6 7 8 9 10 12 14 16 18 20
X': iss iss ipp i$$ ssi ssi ppi

Niestety słowo X' jest zbyt długie (ma $\approx 2n$ znaków zamiast oczekiwanych $2n/3$).

Słowo X reprezentujące sufiksy S_1 i S_2

A gdyby skompresować X'?

Słowo X reprezentujące sufiksy S_1 i S_2

A gdyby skompresować X'?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
X':	i	s	s	i	s	s	i	p	p	i	\$	\$	s	s	i	s	s	i	p	p	i

i\$\$ →0 ipp →1 iss →2 ppi →3 ssi →4

	0	1	2	3	4	5	6
X:	2	2	1	0	4	4	3

Sukces!

Algorytm KS - $\mathrm{SA}_{1,2}$

Algorytm 6: ComputeSA

Dane: słowo T, $|T| = n$

Wynik: tablica sufiksowa SA słowa T

1. podziel sufiksy słowa T na trzy kategorie S_0, S_1, S_2
   $\triangleright$ $|S_i| \approx n/3$
2. wygeneruj słowo X reprezentujące sufiksy z S_1 i S_2
   $\triangleright$ $|X| \approx 2n/3$
3. $\mathrm{SA}_X = \mathrm{ComputeSA}(X)$
4. $\mathrm{SA}_{1,2} =$ wyznacz z SA_X kolejność leks. sufiksów z S_1 i S_2
5. $\mathrm{SA}_0 =$ wyznacz z SA_X i T kolejność leks. sufiksów z S_0
6. $\mathrm{SA} =$ scal $\mathrm{SA}_{1,2}$ i SA_0
7. **return** SA

Tablica $\mathrm{SA}_{1,2}$

Warto zauważyć, że cała procedura przejścia z T do X jest "odwracalna" i zachowuje porządek sufiksów.
Dla dowolnego sufiksu z S_1 i S_2 możemy wskazać odpowiadający mu sufiks w X.

	0	1	2	3	4	5	6
X:	2	2	1	0	4	4	3

	0	1	2	3	4	5	6	7	8	9	10		12
T:	m	i	s	s	i	s	s	i	p	p	i	\$	\$

i	$\mathrm{SA}_X[i]$	suf. w X	suf. w S	$\mathrm{SA}_{1,2}[i]$
0	3	0443	i\$\$	10
1	2	10443	ippi\$\$	7
2	1	210443	issippi\$\$	4
3	0	2210443	ississippi\$\$	1
4	6	3	ppi\$\$	8
5	5	43	ssippi\$\$	5
6	4	443	ssissippi\$\$	2

Algorytm KS - SA_0

Algorytm 7: ComputeSA

Dane: słowo T, $|T| = n$

Wynik: tablica sufiksowa SA słowa T

1. podziel sufiksy słowa T na trzy kategorie S_0, S_1, S_2
   ▷ $|S_i| \approx n/3$
2. wygeneruj słowo X reprezentujące sufiksy z S_1 i S_2
   ▷ $|X| \approx 2n/3$
3. $\mathrm{SA}_X = \mathrm{ComputeSA}(X)$
4. $\mathrm{SA}_{1,2} =$ wyznacz z SA_X kolejność leks. sufiksów z S_1 i S_2
5. $\mathrm{SA}_0 =$ wyznacz z SA_X i T kolejność leks. sufiksów z S_0
6. $\mathrm{SA} =$ scal $\mathrm{SA}_{1,2}$ i SA_0
7. **return** SA

Tablica SA_0

Niestety nie jest możliwe kolejne wywołanie rekurencyjne, żeby obliczyć SA_0. Stąd musimy wykorzystać informację z $\mathrm{SA}_{1,2}$ w celu obliczenia SA_0.

Każdy sufiks i z SA_0 można opisać jako:

$$(T[i],\ \ T[i+1,\ldots])$$

Tablica SA_0

Niestety nie jest możliwe kolejne wywołanie rekurencyjne, żeby obliczyć SA_0. Stąd musimy wykorzystać informację z $\mathrm{SA}_{1,2}$ w celu obliczenia SA_0.
Każdy sufiks i z SA_0 można opisać jako:

$$(T[i], \;\; T[i+1,\ldots])$$

Ponieważ $i+1 \in S_1$, znamy rangę tego sufiksu:

$$(T[i], \;\; \mathrm{SA}_{1,2}^{-1}[i+1])$$

Tak wygenerowane pary zajmują mało miejsca i co najważniejsze możemy je posortować w czasie liniowym. Uporządkowane pary dadzą nam kolejność SA_0.

Algorytm KS - scalanie SA_0 i $\mathrm{SA}_{1,2}$

Algorytm 8: ComputeSA

Dane: słowo T, $|T| = n$

Wynik: tablica sufiksowa SA słowa T

1. podziel sufiksy słowa T na trzy kategorie S_0, S_1, S_2
   ▷ $|S_i| \approx n/3$
2. wygeneruj słowo X reprezentujące sufiksy z S_1 i S_2
   ▷ $|X| \approx 2n/3$
3. $\mathrm{SA}_X = \mathrm{ComputeSA}(X)$
4. $\mathrm{SA}_{1,2} =$ wyznacz z SA_X kolejność leks. sufiksów z S_1 i S_2
5. $\mathrm{SA}_0 =$ wyznacz z SA_X i T kolejność leks. sufiksów z S_0
6. $\mathrm{SA} =$ scal $\mathrm{SA}_{1,2}$ i SA_0
7. **return** SA

Scalanie SA_0 i $\mathrm{SA}_{1,2}$

Został nam już tylko jeden krok — scalanie SA_0 i $\mathrm{SA}_{1,2}$.

i	$\mathrm{SA}_0[i]$	suf. w T
0	0	mississippi$$
1	9	pi$$
2	6	sippi$$
3	3	sissippi$$

i	$\mathrm{SA}_{1,2}[i]$	suf. w T
0	10	i$$
1	7	ippi$$
2	4	issippi$$
3	1	ississippi$$
4	8	ppi$$
5	5	ssippi$$
6	2	ssissippi$$

Scalanie SA_0 i $\mathrm{SA}_{1,2}$

Żeby scalić te dwa uporządkowane (leksykograficznie) podzbiory sufiksów musimy pokazać, że możemy w czasie $O(1)$ porównywać sufiksy $i \in \mathrm{SA}_0$ i $j \in \mathrm{SA}_{1,2}$.

Scalanie SA_0 i $\mathrm{SA}_{1,2}$

Żeby scalić te dwa uporządkowane (leksykograficznie) podzbiory sufiksów musimy pokazać, że możemy w czasie $O(1)$ porównywać sufiksy $i \in \mathrm{SA}_0$ i $j \in \mathrm{SA}_{1,2}$.

Problem polega na tym, że te sufiksy nie są z tych samych światów:

$$\mathrm{cmp}(\text{jabłko}, \text{ziemniak})$$

Scalanie SA_0 i $\mathrm{SA}_{1,2}$

$$\mathrm{cmp}(i \in S_0,\ j \in S_1 \cup S_2)$$

Mamy dwa przypadki:

- $j \in S_1$: porównujemy
  $(T[i],\ \ \mathrm{SA}_{1,2}^{-1}[i+1])$ i $(T[j],\ \ \mathrm{SA}_{1,2}^{-1}[j+1])$
  ($i+1 \in S_1$, $j+1 \in S_2$)

Scalanie SA_0 i $\mathrm{SA}_{1,2}$

$$\mathrm{cmp}(i \in S_0,\ j \in S_1 \cup S_2)$$

Mamy dwa przypadki:

- $j \in S_1$: porównujemy
  $(T[i],\ \mathrm{SA}^{-1}_{1,2}[i+1])$ i $(T[j],\ \mathrm{SA}^{-1}_{1,2}[j+1])$
  $(i+1 \in S_1,\ j+1 \in S_2)$
- $j \in S_2$: porównujemy
  $(T[i],\ T[i+1],\ \mathrm{SA}^{-1}_{1,2}[i+2])$ i $(T[j],\ T[j+1],\ \mathrm{SA}^{-1}_{1,2}[j+2])$
  $(i+2 \in S_2,\ j+2 \in S_1)$

Porównanie – przykład

$$\text{cmp}(\texttt{sippi\$\$} \in S_0, \texttt{ssippi\$\$} \in S_2)$$

redukujemy do:

$$\text{cmp}\left((\texttt{s},\ \texttt{i},\ \text{SA}_{1,2}^{-1}[\texttt{ppi\$\$}]),\quad (\texttt{s},\ \texttt{s},\ \text{SA}_{1,2}^{-1}[\texttt{ippi\$\$}])\right)$$

i	$\text{SA}_{1,2}[i]$	suf. w T
0	10	i$$
1	7	ippi$$
2	4	issippi$$
3	1	ississippi$$
4	8	ppi$$
5	5	ssippi$$
6	2	ssissippi$$

Dziękuję za uwagę!

Byłbym bardzo wdzięczny za wypełnienie ankiety i uwagi: `http://bit.ly/aisd-1203-ankieta`

Przydatne artykuły:

- `https://www.youtube.com/watch?v=NinWEPPrkDQ` — wykład E. Demaine dotyczący algorytmów tekstowych
- `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/suffixarrays.pdf` — bardzo dobre slajdy o tablicach sufiksowych
- `https://www.dmi.unict.it/~faro/papers/conference/faro34.pdf` — Fast Packed String Matching for Short Patterns
- `https://en.wikipedia.org/wiki/Suffix_array`