Wyszukiwanie wzoru (ustalonego podsłowa) w tekście

Projekt na lekcje informatyki

M. K. Feiler J. Kamiński T. Rahn P. Wawreniuk

IV Liceum Ogólnokształcące w Toruniu im. Tadeusza Kościuszki, kl. 2e

listopad—grudzień 2015

Spis treści

Zasadność zadania

Początkowo możnaby się zastanawiać nad zasadnością zadania — ot, wystarczy wykorzystać standardowe biblioteki języka i tyle. Możnaby uznać, że chodzi o pochwalenie się znajomością tychże bibliotek i zaprezentowanie fragmentu dokumentacji.
Jednak, jak wiele zadań na lekcjach informatyki, i to wymaga od nas napisania algorytmu mimo istnienia gotowców, nawet w bibliotece standardowej, a bez wykorzystania tychże.
Zadania takie uniezależniają nas od podstawowych typów języka — dzięki temu będziemy je mogli zoptymalizować, pisząc je dla własnych, konkretnych typów (chociażby liczb w dziwnym systemie — nie będziemy ich przecież zamieniać na znak, bo tylko do niego są gotoce, czyż nie?). Będziemy je też w stanie zaimplementować, pisząc kiedyś np. w Assemblerze, Basicu czy inym języku niskiego poziomu, lub chociażby w języku, dla którego nie będzie gotowców bądź będą gorsze niż w przypadku C, Go, Javy.

Przykładowe gotowce dla języka C/C++

`std::string::find(str, pos)` — zwraca pozycję znalezionego podsłowa lub `string::npos`.

Przykładowe gotowce dla języka C/C++

`std::string::find(str, pos)` — zwraca pozycję znalezionego podsłowa lub `string::npos`.
Możnaby też brutalnie zastosować escape'owane regexy, szczęśliwie jednak zostało to zakazane.

Spis treści

Algorytm naiwny

Algorytm naiwny sprawdza po kolei każdy kolejny znak tekstu w poszukiwaniu pierwszego znaku wzorca; jeżeli go znajdzie, sprawdza następujących po nim w tekście tyle znaków, ile ma długość wzorca, i porównuje z kolejnymi znakami wzorca; jeżeli któryś z nich nie będzie równy odpowiedniemu znakowi wzorca, kończy pętlę i kontynuuje w miejscu po owym znalezionym pierwszym znaku wzorca; kończy, gdy suma pozycji na której jest i długości wzorca przekracza długość tekstu.

Algorytm naiwny

Algorytm naiwny sprawdza po kolei każdy kolejny znak tekstu w poszukiwaniu pierwszego znaku wzorca; jeżeli go znajdzie, sprawdza następujących po nim w tekście tyle znaków, ile ma długość wzorca, i porównuje z kolejnymi znakami wzorca; jeżeli któryś z nich nie będzie równy odpowiedniemu znakowi wzorca, kończy pętlę i kontynuuje w miejscu po owym znalezionym pierwszym znaku wzorca; kończy, gdy suma pozycji na której jest i długości wzorca przekracza długość tekstu.
Pesymistyczna złożoność obliczeniowa tego algorytmu jest równa iloczynowi długości tekstu i wzoru, gdyż tekst może zawierać wiele powtórzeń całkiem długich prefiksów wzorca.

Spis treści

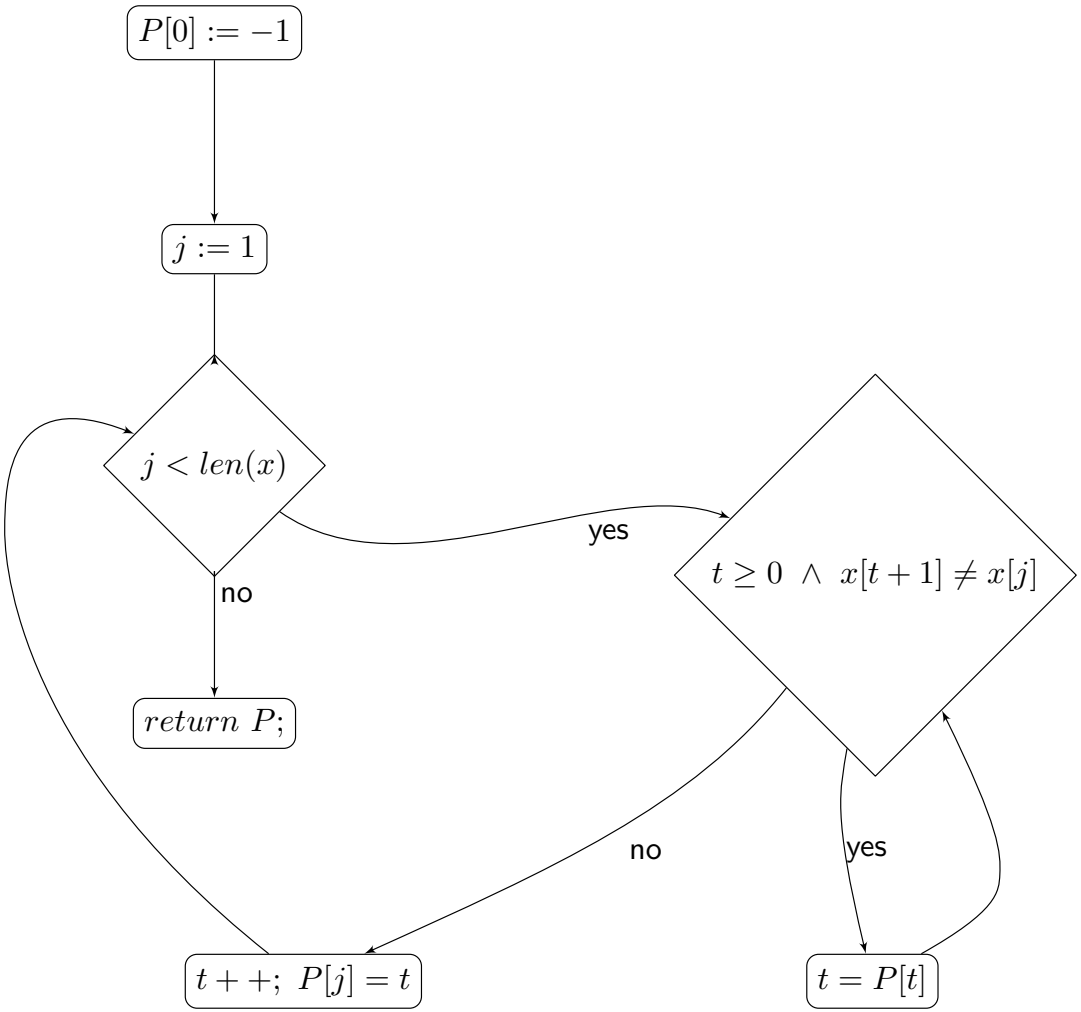
Definicja pojęć

Prefiks i sufiks — podsłowo odpowiednio zaczynające się w początku słowa lub kończące się w końcu słowa.
Okres słowa x — każda niezerowa liczba naturalna p taka, że $x[i] = x[i+p]$, gdzie $i + p < |x|$
Prefikso-sufiks — najdłuższy, nie będący całym słowem, prefiks słowa będący też jego sufiksem. Jeśli $p_{min}(x) = |x|$, to jest to słowo puste o długości 0. Długość prefikso-sufiksu x wynosi zawsze $|x| - p_{min}(x)$.

Tablica prefikso-sufiksów

Oznaczmy przez $P[k]$ rozmiar prefikso-sufiksu z $x[1..k]$ (od 1, nie ~~$x[0..k]$~~).
Z tego mamy $p_{min}(x) = |x| - P[|x|]$.
$P[0]$ jest wartością sztuczną — przyjmujemy $P[0] = -1$.
Jeśli $x[j] = x[P[j-1]+1]$, to $P[j] = P[j-1]+1$.
W algorytmie obliczania $P[j]$ korzystamy z wartości $P[k]$, dla $k < j$

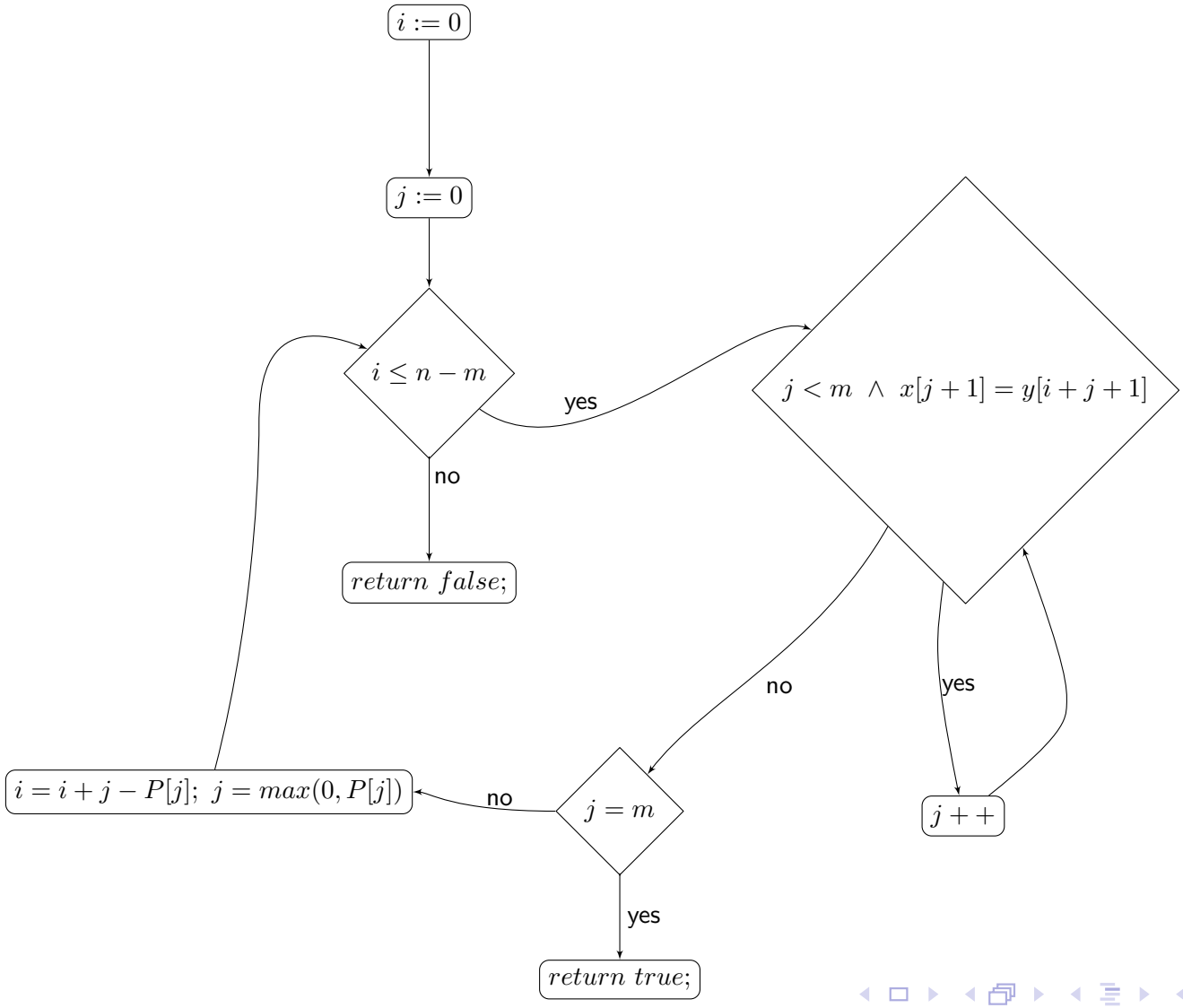
Schemat blokowy



Algorytm Morrisa-Pratta

Algorytm ten, wynaleziony w roku 1974 przez Vaughana Pratta i niezależnie przez Jamesa H. Morrisa w 1977 roku (acz wszyscy opublikowali go wspólnie) wykorzystuje przygotowywaną na początku programu tablicę prefikso-suffiksów wzorca. Jego ogromną zaletą jest fakt, iż (w przeciwieństwie do algorytmu Boyera-Moore'a, który omówimy za chwilę) nie potrzebuje on buforowania tekstu przeszukiwanego — umożliwia to czytanie znak po znaku np. z dysku, przy dużych wzorcach nie zajmując cennych zasobów pamięci komputera.

Schemat blokowy



Tablica „silnych" prefikso-suffiksów, czyli algorytm Knutha-Morrisa-Pratta

Tablicę „silnych" prefikso-sufiksów wzorca będziemy oznaczać przez P'.

Algorytm liczenia „silnych" prefikso-sufiksów bazuje na następującej możliwości zdefiniowania P' z użyciem P:

$$P'[j] = \begin{cases} P[j] & \text{gdy } x[P[j]+1] \neq x[j+1] \\ P'[P[j]] & \text{gdy } x[P[j]+1] = x[j+1] \end{cases}$$

Wykorzystanie tablicy „silnych" prefikso-sufiksów wzorca w miejsce „zwykłej" to jedyne, co odróżnia algorytm KMP od algorytmu MP. Algorytm ten działa szybciej, jednak więcej czasu zajmuje mu preprocessing tablicy.

Spis treści

Algorytm Boyera-Moore'a

Algorytm Boyera-Moore'a to obecnie najszybszy znany algorytm przeszukiwania tekstu. Stosowany w niemal wszystkich programach do odczytu i edycji tekstu. Jego jedyną wadą jest konieczność korzystania z bufora o długości większej bądź równej długości wzorca — algorytm KMP można zaimplementować metodą bezbuforową. Jednakże uznaliśmy, że opóźnienie na wejściu jest wystarczająco małe, by zastosować taki bufor. I tak zresztą stdin jest buforowane.

Opiera się on przede wszystkim na tablicy ostatnich wystąpień znaków we wzorcu. Jest to właściwie mapa, o domyślnych wartościach takich, że jeśli pominąć fakt, że przeliczamy ją na początku programu, można ją rozumieć jako funkcję zdefiniowaną na całym alfabecie.

Uproszczony algorytm Boyera-Moore'a

1. Dla $i := 0, 1, ..., |y| - 1 : \ Last[y[i]] = i + 1$ $(Last - map[char]uint)$
2. $i := 0$ (deklarujemy zmienną całkowitą wartości przesunięcia)
3. Dopóki $i \leq |x| - |y|$: (gdzie $|x|$ jest długością tekstu, a $|y|$ wzorca)
   - 3.1. $j := |y| - 1$
   - 3.2. Dopóki $j \geq 0 \ \wedge \ y[j] = x[i + j] : \ j - -$
   - 3.3. Jeżeli $j < 0$, zwróć $true$.
   - 3.4. $i+ = j - Last[x[i + j]] + 1$
4. Zwróć $false$.

Pełny algorytm Boyera-Moore'a

Uproszczony algorytm BM przedstawiony na poprzednim slajdzie bierze pod uwagę jedynie niezgodność znaku i wykorzystuje tablicę ostatnich wystąpień do znalezienia przesunięcia okna. Postępowanie to nazywamy heurystyką niepasującego znaku. Nie korzystamy w nim z informacji, iż pewien sufiks wzorca pasował do przeszukiwanego tekstu. Mogło się więc zdarzyć, że ten sufiks zawierał się we wzorcu, bądź też w sufiksie tym zawierał się pewien prefikso-sufiks wzorca. W takim wypadku możemy od razu przesunąć okno tak, by się pokryły. Ta strategia nosi nazwę heurystyki pasującego sufiksu.

Tablica wartości przesunięć dla poszczególnych sufiksów wzorca

Do wyznaczenia przesunięcia z użyciem heurystyki pasującego sufiksu będzie nam potrzebna tablica wartości przesunięć dla poszczególnych sufiksów wzorca, zwana dalej tablicą $BMNext$. Tablica ta jest niejako odwróceniem tablicy „silnych" prefikso-sufiksów wykorzystywanej w algorytmie Knutha-Morrisa-Pratta. Wyznaczymy ją w dwóch etapach.

Etap 1 — tablica prefikso-sufiksów wzorca: wyjaśnienie

Ten etap jest bardzo podobny do przetwarzania wzorca w algorytmie KMP. Pasujący sufiks jest prefikso-sufiksem pewnego sufiksu wzorca.
Musimy ustalić prefikso-sufiksy sufiksów wzorca, jednak w porównaniu do algorytmu KMP będzie obowiązywało odwrotne odwzorowanie pomiędzy prefikso-sufiksem, a najkrótszym sufiksem zawierającym ten prefikso-sufiks. Jednocześnie dany prefikso-sufiks nie może być rozszerzalny w lewo przez znak, który spowodował niezgodność z przeszukiwanym tekstem. W przeciwnym razie po przesunięciu okna otrzymalibyśmy natychmiastową niezgodność, ponieważ w tym samym miejscu znów znalazłby się ów niezgodny znak.
W etapie 1 obliczana jest pomocnicza tablica Π. $\Pi[i]$ to początkowa pozycja maksymalnego prefikso-sufiksu sufiksu wzorca rozpoczynającego się na pozycji i-tej.

Etap 1 — tablica prefikso-sufiksów wzorca: lista kroków

1. $i := |y|$
2. $j := |y| + 1$
3. $\Pi[i] = j$
4. Dopóki $i > 0$:
   - 4.1 Dopóki $(j \leq |y|) \ \wedge \ (y[i-1] \neq y[j-1])$:
     - 4.1.1 Jeśli $BMNext[j] = 0$, to $BMNext[j] = j - i$
     - 4.1.2 $j := \Pi[j]$
   - 4.2 $j - -$
   - 4.3 $i - -$
   - 4.4 $\Pi[i] = j$

Etap 2 — maksymalny prefikso-sufiks wzorca zawarty w pasującym sufiksie

W etapie 1 otrzymaliśmy wypełnioną tablicę Π zawierającą położenia prefikso-sufiksów kolejnych sufiksów oraz częściowo wypełnioną tablicę $BMNext$, zawierającą przesunięcia okna dla kolejnych sufiksów. W etapie 2 sprawdzamy, czy istnieje największy prefikso-sufiks wzorca zawarty w całości w pasującym sufiksie. Wzorzec można wtedy przesunąć tak daleko, jak pozwoli na to pasujący prefikso-sufiks. Wyznaczymy więc teraz dla każdego sufiksu najdłuższy prefikso-sufiks wzorca, który jest zawarty w tym sufiksie. Pozycja startowa najszerszego prefikso-sufiksu wzorca jest zawarta w $\Pi[0]$. Wartość ta zostaje umieszczona w kolejnych, wolnych elementach tablicy $BMNext$. Jednakże gdy sufiks wzorca stanie się krótszy od prefikso-sufiksu wzorca (czyli gdy go już nie może zawierać), to kolejnym prefikso-sufiksem stanie się jego prefikso-sufiks wewnętrzny. W efekcie otrzymamy pełną tablicę $BMNext$.

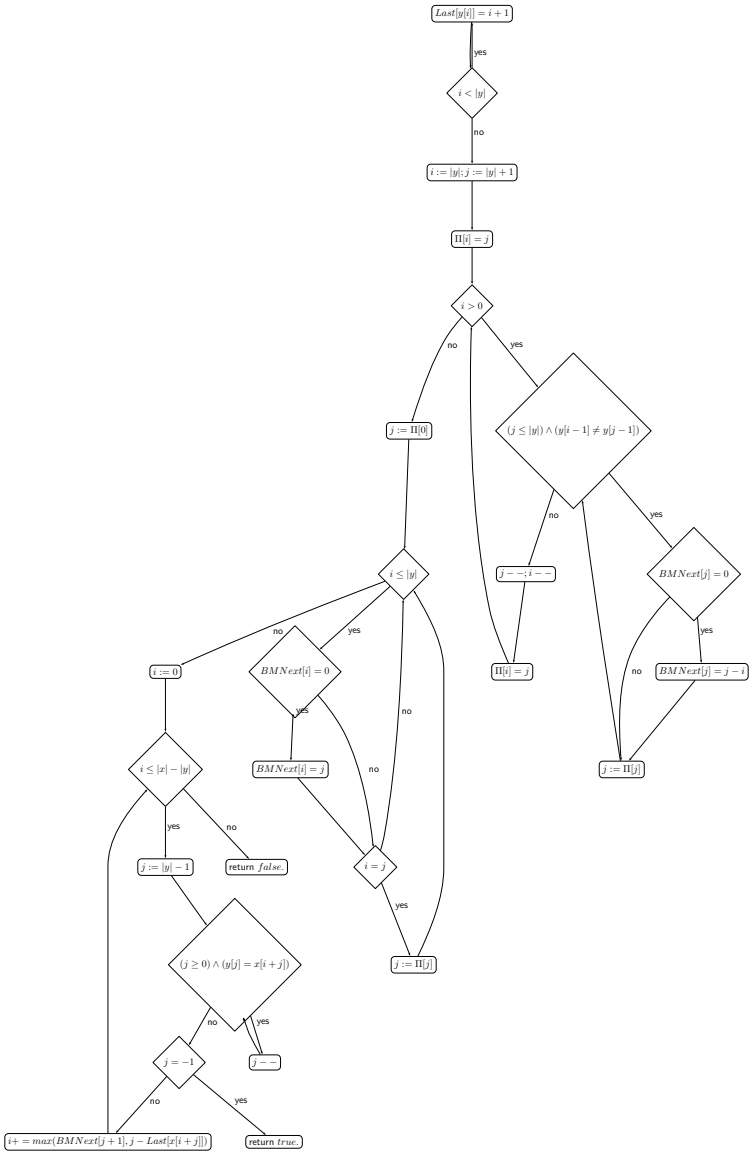
Etap 2 — maksymalny prefikso-sufiks wzorca zawarty w pasującym sufiksie: lista kroków

1. $j := \Pi[0]$
2. Dla $i := 0, 1, ..., |y|$:
   - 2.1 Jeśli $BMNext[i] = 0$, to $BMNext[i] = j$
   - 2.2 Jeśli $i = j$, to $j := \Pi[j]$

Algorytm pełny BM: lista kroków

1. Wyznacz tablicę ostatnich wystąpień dla wzorca
2. Wyznacz tablicę $BMNext$ dla wzorca
3. $i := 0$
4. Dopóki $i \leq |x| - |y|$:
   - 4.1 $j := |y| - 1$
   - 4.2 Dopóki $(j \geq 0) \ \wedge \ (y[j] = x[i + j])$: $j - -$
   - 4.3 Jeśli $j = -1$: zwróć $true$.
   - 4.4 $i+ = max(BMNext[j + 1], j - Last[x[i + j]])$
5. Zwróć $false$.

http://archiet.platinum.edu.pl/zadmatch-fullbm.pdf



Algorytm hashujący Karpa-Rabina

Do dużych tekstów i przy wielu wzorcach (np. wyszukiwania plagiatów) stosuje się też algorytm hashujący Karpa-Rabina. Z braku czasu zdecydowaliśmy się jednak go nie omawiać.

Źródła

http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych/Algorytmy_tekstowe_I
https://pl.wikipedia.org/wiki/Pods\OT4\lowo
http://eduinf.waw.pl/inf/alg/001_search/0043.php
https://pl.wikipedia.org/wiki/Algorytm_Knutha-Morrisa-Pratta
https://pl.wikipedia.org/wiki/Algorytm_Boyera_i_Moore%27a
https://pl.wikipedia.org/wiki/Algorytm_Karpa-Rabina
https://pl.wikipedia.org/wiki/Kategoria:Algorytmy_tekstowe
https://pl.wikipedia.org/wiki/Drzewo_sufiksowe
https://en.wikipedia.org/wiki/Substring
https://en.wikipedia.org/wiki/String_searching_algorithm
https://en.wikipedia.org/wiki/Deterministic_finite_automaton
http://math.uni.lodz.pl/~kubinski/wspw/morrisa-pratta.pdf
http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf
http://www-igm.univ-mlv.fr/~lecroq/string/node8.html
http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm
http://smurf.mimuw.edu.pl/node/394
http://whocouldthat.be/visualizing-string-matching

Dziękuję za uwagę :)
Prezentacja znajduje się pod adresem
`http://archiet.platinum.edu.pl/zadmatch.pdf`

O kod źródłowy proszę pisać na archiet@platinum.edu.pl