Lista druga z algorytmów tekstowych

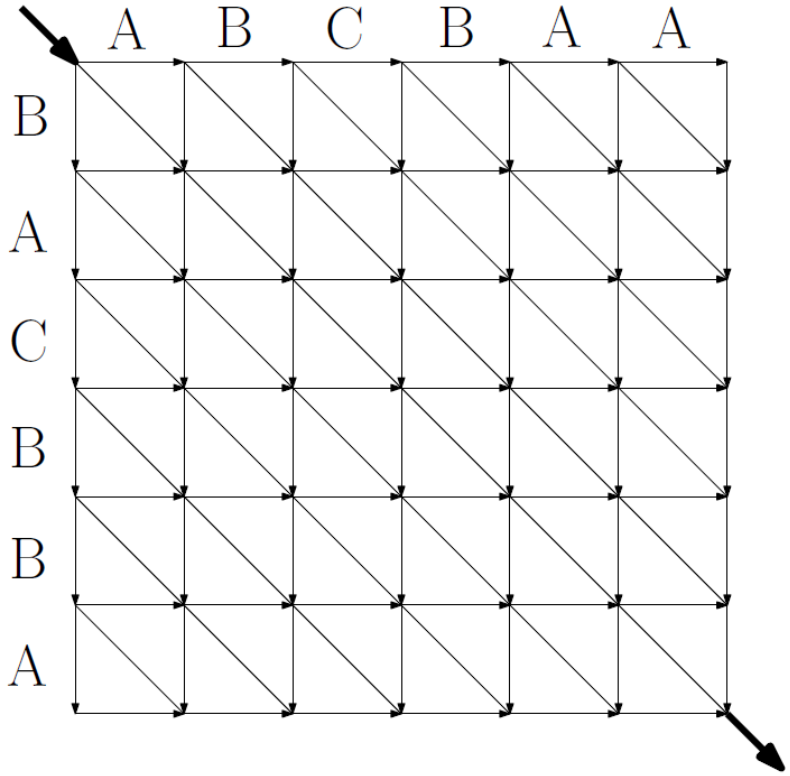
Łukasz Klasinski

21 grudnia 2020

Zadanie 1

Write the pseudocode of the O(n+D2) time algorithm for computing the edit distance assuming that it is equal to D. Try to be as precise as possible (but you can assume that the longest common prefix of any two suffixes can be computed in constant time, don't describe this part).

Przedstawiamy wizualnie problem za pomocą grafu przejść:



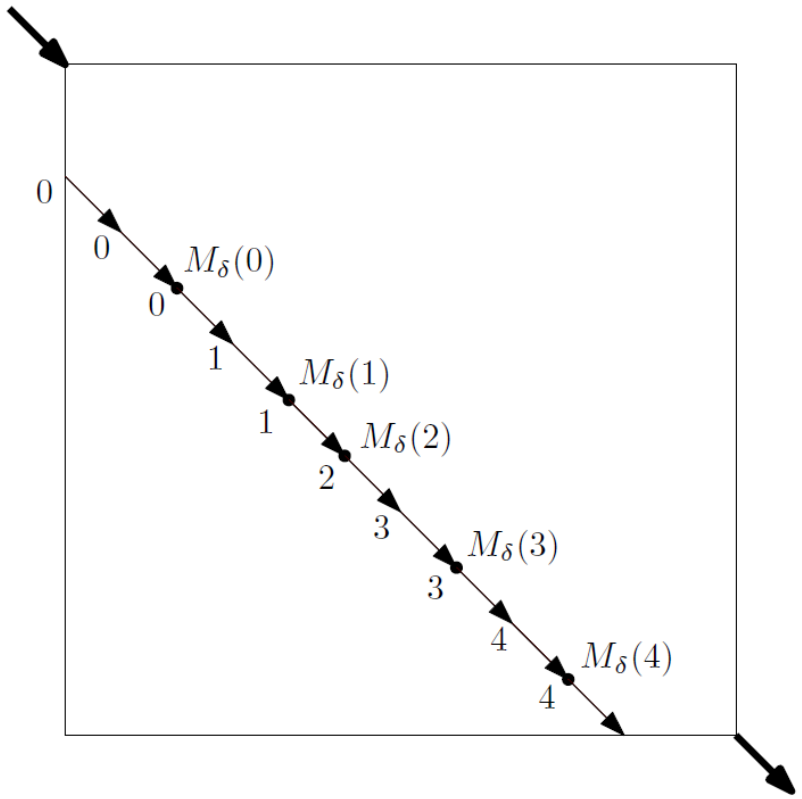
Przekątne symbolizują odpowiednie operacje wykorzystywane przy edit distance (zatem koszt przejścia to 1), natomiast tam, gdzie nie trzeba nic zmieniać (oba ciągi mają wspólny substring), to koszt przejścia wynosi 0. Wtedy aby znaleźć rozwiązanie, należy przejść cały taki graf z punktu $0, 0$ do punktu N, N, tak aby pokonana droga miała jak najmniejszą wagę. Można też zauważyć, że wtedy edit distance odpowiada temu, ile na takiej ścieżce było diagonali o koszcie 0.

Do rozwiązania zadania, używamy sposobu przedstawionego na wykładzie:

Obliczamy kolejno diagonale M_δ dla coraz większych, `zgadniętych` wartości d'. Dzięki temu, że obliczenia z poprzednich iteracji nie ulegają zmianom oraz wykorzystaniu faktu, ze mamy dostepne wyszukiwanie największego wspólnego prefixu dwóch suffixów, to obliczenie takiej jedej diagonali zajmuje $O(D)$, gdzie D to prawdziwa wartość edit distance pomiędzy słowami wejściowymi. Jako że po zgadnięciu prawdziwej wartości D (czyli kiedy $d' = D$) środkowa diagonala dotrze punktu N, N, to całkowity czas trwania algorytmu to koszt wyliczenia diagonali $2 * D + 1 = O(D^2)$ oraz $O(n)$ na preprocessing RMQ queries do odpowiadania w czasie stałym. Łącznie mamy $O(n + D^2)$, czyli tak jak chcemy.

Przypomnienie o konstrukcji M_δ M_δ to ostatnia para (i, j) na diagonali $i - j = \delta$

taka, że $edit\_distance(i, j) = x$ gdzie $x = 0, 1, .., D$



Zauważmy, że taka konstrukcja automatycznie dodaje nam zmniejszenie domeny przeszukiwań, bo szukamy tylko takie wartości, gdzie $d(i, j) \leq D$ dlatego, że M_δ nigdy nie przekroczy wartości D (liczymy ją dla $x = 0..D$). Dodatkowo liczymy tylko te M_δ, któe mają szansę dotrzeć do $(0, 0)$ zakładając daną d', czyli takie δ, że $\delta = i - j$ i $|i - j| \leq D$.

Wartość $M_\delta[x + 1]$ można policzyć korzystając z wartości poprzednich $M\delta[x]$ w następujący sposób:

Sprawdzamy wszystkie trzy możliwości na sąsiadujące diagonale oraz tą którą obliczamy - $i' - j' \in \{\delta - 1, \delta, \delta + 1\}$ takie, że $M_{i'-j'} = x$. Wybieramy takie to M, które jest położone najbliżej do punktu końcowego (N, N) (wtedy możemy przejść z tego punktu do naszego jedną krawędzią o koszcie 1). Następnie przydzielamy $M_\delta[x + 1]$ do tego punktu który znaleźliśmy + tyle ile uda nam się z niego dojść używając common prefix quary.

Poniżej sam psudokod z komentarzami:

```
# wejście, z założeniem, ze ¦w1¦ = ¦w2¦ = N
w1, w2 = input
N = len(w1)

# odpowiadanie na maksymalny wspólny prefix dwóch suffixów w czasie stałym
def max_pref_jump(w1, w2):
    jump = |max_pref(w1, w2)|
    return (jump, jump)

# sprawdzenie jak daleko jest dany punkt w grafie przejścia w stosunku do (N, N)
def cost_fun(p1):
    return manhatan_dist(p1, (N, N))

# Na początku liczymy dla d' = 0, czyli próbujemy skoczyć od punktu (0,0) w przód.
# Jeśli oba słowa są takie same, to algorytm się zakończy, bo nasza diagonala (0,0)
# dotrze do końca grafu.
jump = max_pref(w1, w2)
# M ^= x, reprezentuje dodanie wartości x na koniec listy M
M[0] ^= (0, 0) + max_pref_jump(w1, w2)
if M[0][0] == (N, N) then:
    return 0
# Sprawdzamy wszystkie możliwe wartości d' 0 .. N, bo gdzieś w tym przedziale kryje
# się prawdziwe D.
for D in 1..N do:
    # dodajemy dwie wartości Mδ przy każdej iteracji, będące sąsiadami najbardziej
    # zewnętrznych diagonali i obliczamy ich wartość poczatkową
    M[-D] ^= (0, D) + max_pref_jump(w1, w2[D:])
    M[ D] ^= (D, 0) + max_pref_jump(w1[D:], w2)
    # Iterujemy po kolejnych wartościach x i aktualizujemy wartości diagnoali takich,
    # które nie mają dodanej wartości dla danego x.
    # W szczególności można po prostu dodać d' -1 wartości do nowo dodanych M[-D], M[D]
    # oraz po jednej wartości do M z poprzedniej iteracji, ale tak jest czytelniej
    for x in 1..D do:
        # Aktualizujemy Mδ takie, które nie mają danego x. W szczególności nowo dodane
        # diagnoale będą wymagać dodania d'-1 nowych wartości
        for δ in (δ | len(M[δ]) == x):
            # bierzemy ostatnio dodany punkt do Mδ
            let (i, j) = M[δ][-1]
            # jeśli któraś ze współrzędnych jest na krawiędzi grafu, to stwierdzamy, że
            # wszystkie jego kolejne wartości będą równe (i, j).
            # Dzięki temu nie liczymy niepotrzebnie danej wartości w dół i inne
            # diagonale moga potem sie do nich przesuwać (kiedy będą miały takie same
            # wartości x)
            if i == N or j == N:
                # dodatkowo taki zabieg wyklucza daną diagonalę z obliczeń, bo jej długość
                # jest równa inf != x
```

```
                fill M[δ][-1:inf] with (i, j)
            else:
                # wyznaczamy nową wartość M, za pomocą poprzednich wartości
                # M[δ+1], M[δ-1], M[δ]
                # wyznaczamy takie c, że odległośc miedzy M_c[x-1] a (N,N) jest minimalna
                # oczywiście bierzemy tylko takie δ, dla których są policzone diagonale
                let c = c | min(cost_fun(M[c][x-1])) & c in [δ, δ-1, δ+1]
                # Wyznaczamy nowy punkt, przesuwając go odpowiednio tak, aby był na naszej
                # diagonali (w szczególności z sąsiedniej. Jeśli jest na naszej, to
                # przechodzimy o 1 na ukos)
                (i, j) = M[c][x-1] >> 1
                # robimy offset znalezionego (x,y) o to ile uda nam się skoczyć używając
                # wspólnych prefixów danych suffixów
                M[δ] ^= (i, j) + max_pref_jump(w1[i:], w2[j:])
    # Jeśli nasze M0 dotarło do końca grafu, oznacza to, że istnieje w nim ściezka
    # o wadze d', która wyznacza nasz eiditing distance. Zatem zwracamy d'.
    if M[0][-1] == (N, N):
        return D
```

Zatem dzięki temu, ze na każde M używamy zamortyzowanie $O(D)$, to łączyn czas to $O(n + D^2)$

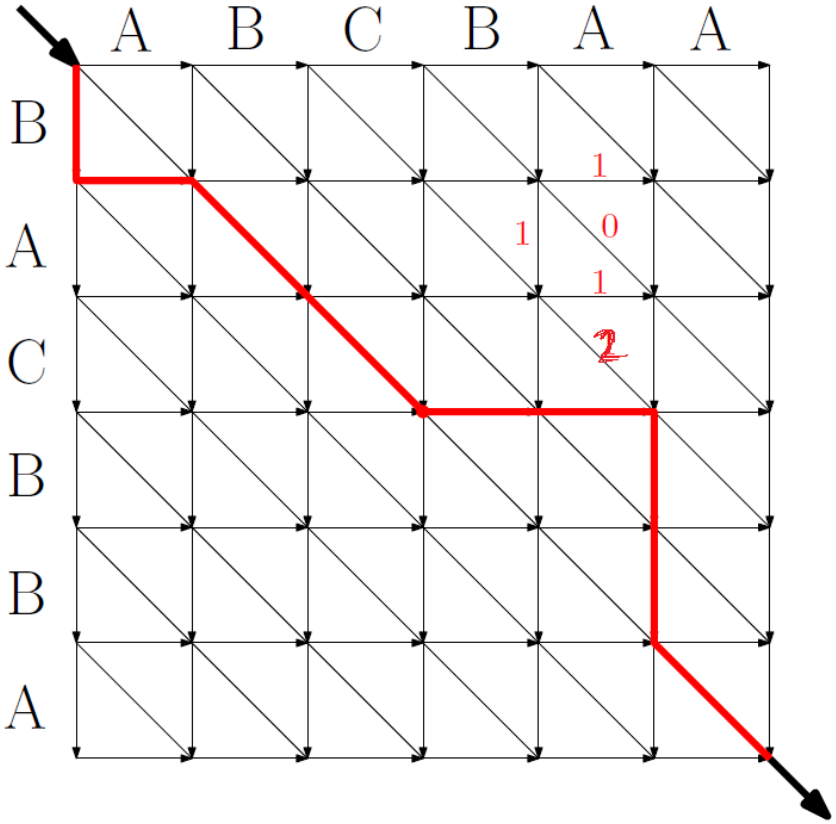
Zadanie 2

Show how to use the Four Russian trick together with bit-packing to solve the LCS problem in $O(n^2\ log\ log/log^2 n)$ time (for alphabet$\{1, 2, ..., \}$).

Pokażę jak rozwiązać ten problem zaczynając od zwykłego dynamika, dodając do niego kolejne ulepszenia.

1. Dynamik podstawowy

Przypomnijmy zatem, jak ma działać nasz dynamik - pracujemy na grafie przejść, zakładamy, że poszczególne operacje mają następujące wagi: * $R_{a,b} = 0$ kiedy $a = b$ * 2 wpp * Insert(I) = delete(D) = 1



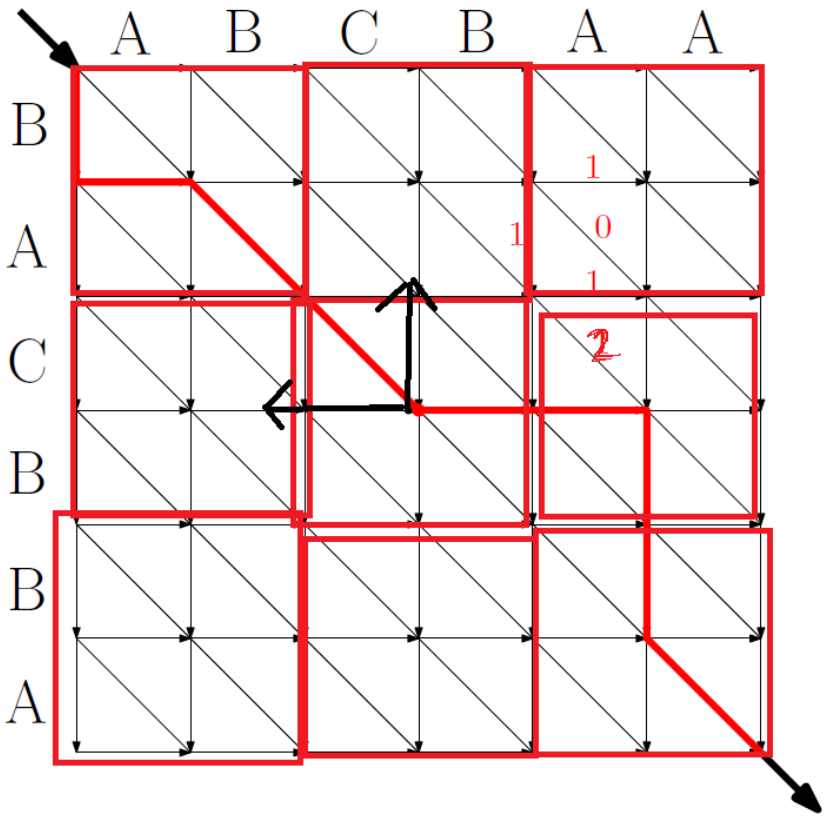
Teraz korzystamy z wzorku rekurencyjnego:

$\delta_{i,j} = min(\delta_{i-1,j-1} + R_{A_i,B_j}, \delta_{i-1,j} + D_{A_i}, \delta_{i,j-1} + I_{B_j})$.

Czas $O(n^2)$

2. Wersja z trikiem Powiedzmy, że na wejściu mamy słowa $w1, w2$ na których sprawdzamy LCS.

Polepszymy czas działania, używając triku 4 Ruskich. Podzielmy nasz graf na mniejsze kwadraciki o krawędziach wielkości x. Można zauważyć, że wtedy dany kwadracik będzie zależny od ich najbliższego kwadracika po lewej oraz z góry (podobnie jak we wzorku na dynamika, tylko teraz patrzymy na kwadraty zamiast pojedyńcze wartości). Poniżej przykład dla $x = 2$:



W szczególności wystarczą nam wyłącznie wektory z którymi styka się nasz kwadrat.

Podsumowując, każdy kwadrat możemy opisać jako:

wektor w, oznaczający różnice δ, dolnej części kwadratu wektor v, oznaczający różnice δ w prawej części kwadratu string s_1 - będący fragmentem stringu ze słowa w_1, któremu odpowiada dany box. string s_2 - będący fragmentem stringu ze słowa w_2, któremu odpowiada dany box.

Przez różnice δ rozumiemy tutaj różnice wynikającą z poniższego wzoru:

$$\delta_{i,j} - \delta_{i-1,j} = min\{R_{a_i,b_j} - (\delta_{i-1,j} - \delta_{i-1,j-1}), D_{A_i}, I_{B_j} + (\delta_{i,j-1} - \delta_{i-1,j-1}) - (\delta_{i-1,j} - \delta_{i-1,j-1})\}$$

$$\delta_{i,j} - \delta_{i,j-1} = min\{R_{a_i,b_j} - (\delta_{i,j-1} - \delta_{i-1,j-1}), D_{A_i}, (\delta_{i-1,j} - \delta_{i-1,j-1}) - (\delta_{i,j-1} - \delta_{i-1,j-1}), I_{B_j}\}$$

Z wykładu wiemy także, że wartości te należą do przedziału $\{-1, 0, 1\}$.

Na razie załóżmy, że mamy skończony alfabet Σ. Chcemy teraz wyliczyć oraz zapisać wszystkie możliwe inputy wraz z tym jakie outputy miałby boxy. Innymi słowy generujemy wszystkie możliwe inputy:

$w = \{w_i, w_{i+1}, ..., w_x\}$ $w_{i..x} in \{-1, 0, 1\}$ $v = \{v_i, v_{i+1}, ..., v_x\}$ $v_{i..x} in \{-1, 0, 1\}$ $s_1 = \Sigma^x$ $s_2 = \Sigma^x$

I teraz generujemy wszywstkie możliwe pary wejściowe wybierając je z (w, v) $oraz$ (s_1, s_2). Dla takiego wygenerowanego inputu, uruchamiamy algorytm dynamiczny, który wylicza nam wartości wektorów na wyjściu, które sobie zapisujemy w tablicy.

Tablicy z wynikami używamy później w wyliczaniu wszystkich boxów w czasie $O(1)$ - wystarczy, że znajdziemy wynik który wcześniej wyliczyliśmy dla danej konfiguracji i zapiszemy do boxa.

Zauważmy, że ilość którą musimy wegenerować wynosi: $3^x * 3^k * |\Sigma|^x * |\Sigma^3|$. Biorąc odpowiedni $x = logn$, całość redukuje nam się do jakiejś stałej. Poza tym potrzebujemy k^2 czasu na obliczenie dynamika w środku. Ostatecznie mamy $O(1) * x^2 = O(x^2)$.

```
for each pair C, D of strings in Σ^m and
    each pair of length m step vectors R and S
do
    begin
        for i = 1 to m do
            begin
                T(i, 0) := R(i);
                U(0, i) := S(i);
            end;
        for i = 1 to m do
        for j = 1 to m do
            begin
                T(i,j) := min{R_{C_i,D_j} − U(i − 1, j), D_{C_i},
                                I_{D_j} + T(i, j − 1) − U(i − 1, j)};
                U(i,j) := min{R_{C_i,D_j} − T(i, j − 1),
                                D_{C_i} + U(i − 1, j) − T(i, j − 1), I_{D_j}}
            end;
        R' := ⟨T(1, m),..., T(m, m)⟩;
        S' := ⟨U(m, 1),..., U(m, m)⟩;
        Store (R', S', R, S, C, D);
    end;
```

algorytm:

Zatem powinniśmy otrzymać złożoność $\frac{x^2}{x^2}\frac{n^2}{x^2}$(bo mamy $\frac{n^2}{x^2}$ elementów). Niezbyt dobrze, bo wychodzi po prostu n^2. Aby to poprawić zastosujemy bipacking.

Dzięki temu, że nasze wartości są w przedziałach $\{-1, 0, 1, 2\}$, to możemy reprezentować taki wektor jako ciąg binarny - np. stosując kodowanie: $\{-1 : b11, 0 : b00, 1 : b01, 2 : b10\}$, jesteśmy w stanie stworzyć funkcje, które przyjmując dane wektory w postaci binarnej wykonuje na nich odejmowanie, dodawania lub wyznacza minimum i zwraca jako odpowiedni wektor w postaci binarnej. Potem można go łatwo odkodować. Pozwala nam to na ominięcie jednej pętli z części odpowiadającej za uruchomienie dynamika na danym boxie i zmniejszenie złożoności.

Ostatecznie, przy $x = logn$, założeniu że liczymy wektory stosując bit-packing, złożność takiego algorytmu wynosi $O(\frac{n^2}{x}) = O(\frac{n^2}{logn})$.

Niestety dalej mamy wymóg, że alfabet jest skończony, więc taki algorytm ma niszowe zastosowanie.

3. Dodajemy skończony alfabet

Problemem jest to, że przy nieograniczonym, bądź dużym alfabecie nasz algorytm nie działa, lub jest zbyt wolny. Pomijamy to stosując poniższy trik:

Kiedy analizujemy dany kwadrat, którego inputem jest fragment stringa wejściowego, zamiast działać na oryginalnym alfabecie, zastosujemy następujące mapowanie:

weźmy substring s_1 należący do danego boxa. Zamieńmy go na listę char'ów i posortujmy leksykograficznie. Od teraz stosujemy następujące mapowanie:

Dla każdej literki a, zamiast a używamy teraz jej pozycji w posortowanej tablicy.

Np. dla ABA -> ['A', 'B'] -> 010

Teraz analizując s_2, używamy tego samego mapowania, co przy s_1, tylko w przypadku kiedy dana literka nie ma mapowania, to zamieniamy ją na specjalny indeks $m = len(s1.sort())$. W ten sposób jesteśmy w stanie stworzyć mapowanie dla danego kwadratu, który zamienia dowolne znaki na liczby $\{0, 1, ..x\}$. W takim razie możemy zmienić nasz alfabet dla wszystkich kwadratów do $\Sigma = \{0, 1, 2, ...x\}$. Stosując takie mapowanie można użyć algorytmu z punktu 2).

Niestety jeśli zastosujemy takie mapowanie do każdego box'a, to popsujemy sobie złożoność ($xlogx$ operacji na każdy box). Dlatego stosujemy tą transformację dla zbioru superboxa x *na* x.

Teraz dzięki zastosowaniu wszystkich poznanych trików, ilosć operacji na każdy superbox zmienia się na $x^2 log\, x^2$, na zwykły box $xlogx$ i mamy n^2/x^2 takich boxów. Ostatecznie nasza złożoność zmienia się na $O(\frac{x^2 log^2\, x}{x^2})$ i po podstawieniu z $x = logn$ dostajemy $O(\frac{n^2 log^2\, logn}{log^2\, n})$

Zadanie 3

We consider two functions defined for every position in a string $T[1..n]$

1. $f(i) = max\{l : T[i..(i+l-1)] = T[j..(j+l-1)]$ for some $j < i\}$
2. $g(i) = max\{l : T[i..(i+l-1)] = T[j..(j+l-1)]$ for some j such that $j + k - 1 < i\}$

Show how to compute both functions for all positions in O(n)total time. You can use thesuffix array together with its LCP array (and constant-time RMQ queries).

- obliczanie $f(i)$ where $i \in \{1..n\}$

Można łatwo zauważyć, że funkcja $f(i)$ przedstawia tak naprawdę wartość $max_j(LCP(i, j))$. Poza tą własnością, skorzystamy z tablicy suffiksowej SA oraz jej odwrotności SA^{-1} dla danego stringa wejściowego w.

Z tego, jak zbudowana jest tablica suffiksowa, widać iż jeśli chcemy znaleźć maksymalne $LCP(i, j)$, to wystarczy odnaleźć i-tą wartość w SA i następnie sprawdzić wartości LCP dla pierwszej wartości k mniejszej od i takiej, że

$$SA^{-1}[k] < SA^{-1}[i]$$

(oznacza to, że szukamy na lewo w tablicy suffiksów) oraz pierwszej wartości l mniejszej od i takiej, że

$$SA^{-1}[l] > SA^{-1}[i]$$

(czyli tutaj patrzymy na prawo w tabicy suffiksów). Wpisy te można znaleźć dzięki temu, że tablice suffiksowe mamy posortowane leksykograficznie.

Problemem zostaje fakt, że przeszukiwanie na prawo i na lewo zajmuje $O(n)$, zatem trzeba użyć trik, w którym obliczymy oddzielnie wartości dla wszystkich i idąc w tablicy suffiksowej na lewo oraz na prawo i na koniec wybrać maksimum z tych wartości. Podczas przechodzenia będziemy trzymać wartości na stosie i dobierać je zachłannie dla kolejno wczytywanych wartości z SA.

Zatem nasze rozwiązanie stworzy 2 tablice M_L oraz M_R a wynikiem będzie tablica

$$M = [i \in \{0..n\}\, max\{M_L[i], M_R[i]\}]$$

```
w = input
# tablica wynikowa wypełniona 0 - f(i) = 0
M = [0 for i in range(|w|)]
q = queue()
# obliczamy tablice suffiksową w O(n)
SA = suff_arr(w)
ML = []
MR = []
for i in SA:
    # ściągamy elementy ze stosu dopóki nie trafimy na jakiś, który jest większy niż
    # ten na który wskazuje i (czyli mamy spełniony warunek, ze wybieramy pierwsze i,
    # które jest mniejsze
    while not q.empty() and head(q) > i:
        MR ^= LCP(q.pop())
    # dodajemy dane i na stos
    q.push(i)

q = queue()
for i in SA.reverse():
    while not q.empty() and head(q) < i:
        ML ^= LCP(q.pop())
    q.push(i)

M = [r if r > l else l for r,l in zip(MR, ML)]
return M
```

Widać zatem, ze algorytm działa w $O(n)$, ponieważ dwukrotnie iteruje po wszystkich wartościach SA, każdą co najwyżej stałą ilość razy.

- obliczanie $g(i)$ gdzie $i \in \{1..n\}$

Jak mysleć o tej funkcji - możemy przedstawić ją jako: $min(LCP(i,j), i-j)$, gdzie $j < i$. Można to zinterpretować w taki sposób, że pierwsza część to po prostu LCP takich substringów, które na siebie nie mogą nachodzić, natomiast $i-j$ odpowiada za te które nachodzą, ale są ucięte w punkcie i (zatem ich wspólny prefix ma wartość $i-j$).Przez to też obliczenie tej funkcji wydaje się być znacznie trudniejsze niż funkcji f, ponieważ nie możemy brać najbliższych wartości z SA, gdyż mogą one zostać ucięte i nie być optimum. Potrzebujemy zatem serii trików i obserwacji, które nas uwolnią od przeszukiwania całego SA w celu znalezienia największego wspólnego prefilksu.

Wpierw podzielmy problem na dwa podproblemy:

- $g_o(i) = max\{l : T[i..(i+l-1)] = T[j..(j+l-1)]$ for some j such that $j+k-1 < i\}$ and $j+l = i$
- $g_m(i) = max\{l : T[i..(i+l-1)] = T[j..(j+l-1)]$ for some j such that $j+k-1 < i\}$

Czyli $g_o(i)$ oznacza takie wspólne maksymalne prefiksy p_1, p_2 między $w[0..i]$ i $w[i..]$, że między p_1 i p_2 nie ma żadnych dodatkowych liter. Na przykład weźmy słowo

$w = missisippy$ $i = 6$ wtedy $g_o(i)$ wynosi 2 - $mis[si|si]ppy$ Widać zatem, że tak naprawdę mamy tutaj powtórzenie jakiegoś substringa z w, którego środkiem jest i.

Z tego co czytałem, można zrobić preprocessing tego w $O(n)$ na RMQ, które konstruujemy używając algorytmu znajdowania maksymalnego powtórzenia podstringów w słowie.

> R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In Proc. Symposium on Foundations of Computer Science (FOCS), pages 596–604, 1999.

Kiedy wyliczymy już g_o oraz g_m, masza wyjściowa tablica to oczywiście $g(i) = max\{g_o(i), g_m(i)\}$

Pozostaje zatem pokazać jak obliczyć g_m.

Podobnie jak w f, policzymy oddzielnie g_mL oraz g_mR.

Przyda się nam też poniższa obserwacja: $g_m(i) > 0 \implies g_m(i+1) \geq g_m(i) - 1$