Algorytmy tekstowe

Krzysztof Turowski

1 Notacja

1.1 Podstawowe pojęcia

Definicja 1.1. ***Alfabet*** $\mathcal{A}$ – (skończony) zbiór symboli.

W większości algorytmów rozmiar alfabetu jest stały i pomijany w analizie złożoności.

Definicja 1.2. ***Słowo*** $w \in \mathcal{A}^*$ to ciąg zbudowany nad alfabetem $\mathcal{A}$.

Słowo puste jest oznaczane symbolem ε. Zbiór słów niepustych to $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$.

Zbiór słów długości n oznaczamy $\mathcal{A}_n$ i definiujemy jako $\mathcal{A}_0 = \{\varepsilon\}$, $A_{n+1} = \bigcup_{x\in\mathcal{A}_n, y\in\mathcal{A}} xy$.

Numerację symboli w słowie zaczynamy od 1, więc $w[i]$ (albo w_i) jest i-tym symbolem w słowie w.

Definicja 1.3. ***Długość słowa*** to funkcja $|\cdot| : \mathcal{A}^* \to \mathbb{N}$ taka, że $|w| = n$ wtedy i tylko wtedy, gdy $w \in \mathcal{A}^n$.

Definicja 1.4. Słowo u jest ***podsłowem*** (ang. *factor*) słowa w, gdy istnieją słowa $v_1 v_2 \in \mathcal{A}^*$ takie, że $w = v_1 u v_2$. Podsłowo jest ***właściwe***, gdy $v_1 v_2 \neq \varepsilon$.

Definicja 1.5. Słowo u jest ***prefiksem*** (***sufiksem***) słowa w, gdy istnieje słowo $v \in \mathcal{A}^*$ takie, że $w = uv$ ($w = vu$). Prefiks (sufiks) jest ***właściwy***, gdy $v \neq \varepsilon$.

Definicja 1.6. Słowo u jest ***podciągiem*** słowa w, gdy istnieją liczby $1 \leq i_1 < i_2 < \ldots < i_k \leq |w|$ takie, że $u = w[i_1]w[i_2]\ldots w[i_k]$.

Definicja 1.7. Słowo $\bar{w}$ jest ***odwrotnością*** słowa w, gdy dla $n = |w|$ mamy $\bar{w} = w[n]w[n-1]\ldots w[1]$.

Definicja 1.8. Słowo w jest ***palindromem***, jeśli istnieją słowa $u, v \in \mathcal{A}^*$ takie, że $|u| \leq 1$ i $w = \bar{v}uv$.

1.2 Porządki i odległości

Złożoność obliczeniowa algorytmów tekstowych obliczana jest przy przyjęciu dwóch (binarnych) operacji atomowych na parach liter z alfabetu: $=$ oraz $\leq$.

Definicja 1.9. Relacja $\preceq$ jest relacją ***porządku prefiksowego*** tj. $u \preceq v$ wtedy i tylko wtedy, gdy u jest prefiksem u.

Definicja 1.10. Relacja $\preceq_R$ jest relacją ***porządku wojskowego*** (*radix*) tj. $u \preceq_R v$ wtedy i tylko wtedy, gdy:

- albo $|u| < |v|$,
- albo $|u| = |v|$ oraz istnieje $1 \leq k \leq |u|$ takie, że $u[k] < v[k]$ i dla wszystkich $1 \leq j < k$ zachodzi $u[j] = v[j]$.

Definicja 1.11. Relacja $<$ jest relacją ***porządku leksykograficznego*** tj. $u < v$ wtedy i tylko wtedy, gdy:

- albo u jest prefiksem v,
- albo istnieje $1 \leq k \leq |u|$ takie, że $u[k] < v[k]$ i dla wszystkich $1 \leq j < k$ zachodzi $u[j] = v[j]$.

Wniosek 1.12. Porządek leksykograficzny i wojskowy są porządkami liniowymi, rozszerzającymi porządek prefiksowy. Dla słów równego długości u, v zachodzi $u < v$ wtedy i tylko wtedy, gdy $u \preceq_R v$.

1.3 Okresy i słowa pierwotne

Definicja 1.13. Liczba $p \in \mathbb{N}_+$ jest ***okresem*** słowa w, jeśli dla każdego $1 \leq i \leq |w| - p$ zachodzi $w[i] = w[i + p]$.
Najmniejszy okres słowa w oznaczamy przez $p(w)$. Z definicji $|w|$ jest okresem słowa, więc $p(w)$ jest dobrze zdefiniowane.

Definicja 1.14. Słowo u jest ***prefikso-sufiksem*** (ang. *border*) słowa w, jeśli istnieją słowa v_1, v_2 takie, że $w = uv_1 = v_2u$.
Z definicji ε jest prefikso-sufiksem każdego słowa w.

Problem 1.1. Pokaż, że następujące warunki są równoważne:

(i) w ma okres p,
(ii) w jest podsłowem pewnego v^k dla $|v| = p$ i $k \geq 1$,
(iii) $w = (uv)^k v$ dla $|uv| = p$, $v \neq \varepsilon$ i $k \geq 1$,
(iv) w ma prefikso-sufiks długości $|w| - p$.

Definicja 1.15. Słowo w jest ***pierwotne***, jeśli nie istnieje słowo v oraz liczba całkowita $k \geq 2$ takie, że $w = v^k$.

Problem 1.2. Słowo w jest słowem pierwotnym wtedy i tylko wtedy, gdy $p(w) = |w|$ lub $p(w)$ nie dzieli $|w|$.

Problem 1.3. Pokaż, że następujące twierdzenia są równoważne:

(i) $p(w^2) = |w|$,
(ii) w jest słowem pierwotnym,
(iii) w^2 zawiera dokładnie dwa wystąpienia w.

Twierdzenie 1.16 (Słaby lemat o okresowości)**.** *Jeśli słowo w ma okresy p i q takie, że $|w| \geq p + q$, to w ma również okres $NWD(p, q)$.*

Dowód. Bez straty ogólności załóżmy, że $p \geq q$. Najpierw wykażemy, że jeżeli słowo w ma okresy p i q oraz $|w| \geq p + q$, to w ma również okres $p - q$.

Dla $1 \leq i \leq q$ mamy $a[i] = a[i + p] = a[i + p - q]$, ponieważ $1 \leq i \leq i + p - q \geq i + p \leq |w|$. Podobnie dla $q + 1 \leq i \leq |w| - (p - q)$ mamy $a[i] = a[i - q] = a[i + p - q]$, ponieważ $1 \leq i - q \leq i + p - q \geq |w|$.

Z algorytmu Euklidesa wynika wprost, że iterując to rozumowanie możemy pokazać, że w ma również okres $NWD(p, q)$. □

Twierdzenie 1.17 (Silny lemat o okresowości)**.** *Jeśli słowo w ma okresy p i q takie, że $|w| \geq p + q - NWD(p, q)$, to w ma również okres $NWD(p, q)$.*

Dowód. Po pierwsze, jeśli słowo w ma okresy o $< q < p \leq |w|$, to prefiks i sufiks w o długości $|w| - q$ mają okresy $p - q$. Dla prefiksu wystarczy zauważyć, że dla dowolnego $1 \leq i \leq |w| - p$ zachodzi $w[i] = w[i + p] = w[i + p - q]$ – a to właśnie jest definicja okresu dla słowa $w[1..(|w| - q)]$.

Po drugie, jeśli w ma okres q i istnieje podsłowo v słowa w z $|v| \geq q$ takim, że r jest okresem v i r dzieli q, to w ma okres r.

Niech $r = NWD(p, q)$. Dowód przebiega przez indukcję ze względu na $s = \frac{p+q}{r}$. Dla $p = q = r$ – więc twierdzenie jest oczywiście spełnione np. dla $s = 2$.

Dla $s > 2$ i $q < p$ weźmy słowo w mające okresy p i q takie, że $|w| \geq p + q - r$. Niech $u = w[1..q]$ oraz $w = uv$. Wówczas z pierwszego faktu wiemy, że v ma okres $p - q$. Jednocześnie v jest podsłowem w oraz $|v| = |w| - q \geq p - r \geq q$, więc v ma również okres q.

Dalej wiemy, że $r = NWD(p - q, q)$, $s > \frac{(p-q)+q}{r}$ oraz

$$|v| = |w| - q \geq (p + q - r) - q = (p - q) + q - NWD(p - q, q).$$

Z założenia indukcyjnego v ma zatem okres r. Ostatecznie z drugiego faktu wiemy, że w ma też okres r. □

Problem 1.4. Korzystając ze słów Fibonacciego pokaż, że warunku $|w| \geq p+q-NWD(p, q)$ w silnym lemacie o okresowości nie da się poprawić.

Definicja 1.18. Liczba $ord(w)$ jest ***rzędem*** słowa w, jeśli $ord(w) = |w|/p(w)$.

Problem 1.5. Pokaż, że słowo Fibonacciego jest słowem pierwotnym.

Dowód. Załóżmy, że istnieją $u, k : u^k = Fib_i, k > 1$. Wprowadźmy oznaczenie $u = st$ gdzie $s = u[1 \ldots |u| - 2]$ oraz $t = u[|u| - 1, |u|]$.

Na podstawie poprzedniego zadania wiemy, że słowa Fibonacciego bez ostatnich dwu znaków są palindromem. Dlatego $(st)^{k-1}s$ powinno też być palindromem. Z tego wynika, że $s = \overline{s}$ oraz $t = \overline{t}$. To

może zachodzić tylko dla dwu przypadków: $t = 00$ i $t = 11$. Na ćwiczeniach jednak zostało pokazane, że dla słów Fibonacciego jedyne dwie opcje są $t = 01$ i $t = 10$. Otrzymujemy sprzeczność, która pokazuje, że żadne słowo Fibonacciego nie można zapisać jako u^k, $k > 1$, więc każde słowo Fibonacciego jest słowem pierwotnym. □

1.4 Słowa sprzężone

Definicja 1.19. Słowa v, w są ***sprzężone*** wtedy i tylko wtedy, gdy istnieją słowa u_1, u_2 takie, że $v = u_1u_2$ i $w = u_2u_1$.

Wniosek 1.20. Relacja sprzężenia (cyklicznego obrotu) jest relacją równoważności, więc definiuje klasy równoważności w $\mathcal{A}^*$.

Problem 1.6. Pokaż, ile elementów ma klasa równoważności dla słowa v.

Problem 1.7. Pokaż dowód małego twierdzenia Fermata na bazie wiedzy, że słowo pierwotne v należy do klasy równoważności o mocy $|v|$.

Dowód. Przypomnijmy na wstępie dowodu wypowiedź małego twierdzenia Fermata: jeżeli p jest liczbą pierwszą, a n dowolną liczbą naturalną, to $p|(n^p - n)$.

Zdefiniujmy taką relację na słowach, że x jest w relacji z y, jeżeli tylko x jest cyklicznym przesunięciem y. Oczywiście jest to relacja równoważności. Rozważmy słowa unarne długości p, czyli słowa postaci a^p, gdzie a jest pewną literą. Niech K będzie zbiorem wszystkich nieunarnych słów długości p nad alfabetem $\{1, \cdots, n\}$. Wszystkie te słowa są pierwotne, ponieważ ich długość jest liczbą pierwszą oraz nie są unarne. Na podstawie założeń dostajemy, że każda klasa równoważności naszej relacji ma dokładnie p elementów. Dodatkowo zauważmy, że zbiór K ma dokładnie $n^p - n$ elementów. Ponieważ K może być podzielony na rozłączne podzbiory mające po p elementów, to $p|(n^p - n)$, co kończy dowód. □

Definicja 1.21. Słowo w jest ***słowem Lyndona*** wtedy i tylko wtedy, gdy jest minimalnym (leksykograficznie, wojskowo) słowem w ramach klasy sprzężenia.

1.5 Morfizmy i klasy słów

Definicja 1.22. Funkcja $f : \mathcal{A}^* \to \mathcal{B}^*$ jest ***morfizmem*** (***podstawieniem***) jeśli $f(xy) = f(x)f(y)$ dla wszystkich $x, y \in \mathcal{A}^*$.
Morfizm jest dosłowny (*literal*), gdy dla każdego $x \in \mathcal{A}$ zachodzi $|f(x)| = 1$.
Morfizm jest nieusuwający (*non-erasing*), gdy dla każdego $x \in \mathcal{A}$ zachodzi $f(x) \neq \varepsilon$.

1.5.1 Słowa Fibonacciego

Definicja 1.23 (Lothaire 2002, s. 10-11)**.** Słowo f_n jest ***słowem Fibonacciego***, gdy $f_0 = 0$, $f_1 = 01$ oraz $f_{k+2} = f_{k+1}f_k$ dla $k = 0, 1, \ldots$.
Równoważnie, $f_k = \phi^n(0)$ dla morfizmu $\phi(0) = 01$, $\phi(1) = 0$.

Problem 1.8. Dla $k \geq 1$ niech $u = f_k f_{k+1}$ i $v = f_{k+1} f_k$. Pokaż, że u powstaje z v przez zamianę dwóch ostatnich liter.

Problem 1.9. Jeśli dla $u \in \mathcal{A}^+$ słowo u^2 jest podsłowem pewnego f_k, to $|u|$ jest pewną liczbą Fibonacciego i u jest sprzężone z pewnym f_l.

1.5.2 Słowa Thuego-Morse'a

Definicja 1.24 (Lothaire 2002, s. 11)**.** Słowo u_n jest ***słowem Thuego-Morse'a***, gdy $u_0 = 0$, $v_0 = 1$ oraz $u_{k+1} = u_k v_k$, $v_{k+1} = v_k u_k$ dla $k = 0, 1, \ldots$.
Równoważnie, $u_k = \mu^n(0)$ dla morfizmu $\mu(0) = 01$, $\mu(1) = 10$.

Problem 1.10. Udowodnić że słowa Thuego-Morse'a u_n są słowami pierwotnymi.

Problem 1.11. Udowodnić że słowa Thuego-Morse'a u_{2n} są palindromami.

Dowód. Zgodnie z definicją rekurencyjną mamy $u_0 = 0$, $v_0 = 1$ oraz $u_{k+1} = u_k v_k$, $v_{k+1} = v_k u_k$. Rozumować będziemy indukcyjnie. Dla $n = 0$, słowo $u_0 = 0$, $v_0 = 1$ są oczywiście palindromami. Załóżmy zatem, że dla wszystkich $k < n$ prawdą jest, że u_{2k} oraz v_{2k} są palindromami. Będziemy chcieli wykazać, że jest to prawdą także dla u_{2n} i v_{2n}. Z definicji rekurencyjnej dostajemy, że $u_{2n} = u_{2n-1} v_{2n-1} = u_{2n-2} v_{2n-2} v_{2n-2} u_{2n-2}$. Ponieważ u_{2n_2} oraz v_{2n-2} są palindromami na mocy założenia indukcyjnego, to u_{2n} też jest palindromem. Dokładnie taki sam argument pokazuje, że v_{2n} również jest palindromem. Zatem zakończyliśmy dowód kroku indukcyjnego, a więc też cały dowód. □

Problem 1.12. Sprawdzić czy słowa Thuego-Morse'a zawierają podsłowa u^3 lub $(uv)^2 u$ dla pewnych $u, v \in \mathcal{A}^+$.

1.6 Kody

Definicja 1.25. $X \subset \mathcal{A}^+$ jest ***kodem***, gdy dla dowolnych $x_1, \ldots, x_n, y_1, \ldots, y_m \in X$ jeśli dla $x_1 \ldots x_n = y_1 \ldots y_m$, to $n = m$ oraz $x_i = y_i$ dla $i = 1, \ldots, n$.

Definicja 1.26. $X \subset \mathcal{A}^+$ jest ***kodem prefiksowym*** gdy X jest kodem i dla żadnych $x, y \in X$ słowo x nie jest prefiksem y.

Problem 1.13. Zbiór $X \subset \mathcal{A}^+$ jest kodem dla $\mathcal{B}^*$ wtedy i tylko wtedy, gdy dowolny morfizm $\phi : \mathcal{B}^* \to \mathcal{A}^*$ indukuje iniekcję z $\mathcal{B}$ na X.

2 Dokładne dopasowanie wzorca

Problem 1: Dokładne dopasowanie wzorca
Wejście: Słowa $t, w \in \mathcal{A}^+$ ($\|t\| = n$, $\|w\| = m$)
Wyjście: TRUE jeśli w jest podsłowem t, FALSE w przeciwnym przypadku

Problem 2: Wyszukiwanie wszystkich wystąpień wzorca w tekście

Wejście: Słowa $t, w \in \mathcal{A}^+$ ($|t| = n, |w| = m$)
Wyjście: Zbiór liczb $S = \{1 \leq i \leq n : t[i..(i+m-1)] = w\}$.

2.1 Algorytm naiwny

Algorytm 1: Naiwne szukanie wzorca w w tekście t

```
def naive_string_matching(t, w, n, m):
  for i in range(n - m + 1):
    j = 0
    while j < m and t[i + j + 1] == w[j + 1]:
      j = j + 1
    if j == m:
      return True
  return False
```

Problem 2.1. Dla ustalonego n i m znajdź przykład słowa w i tekstu t z $|t| = n$, $|w| = m$, dla którego Algorytm 1 wykonuje najwięcej porównań.

Dowód. Najprostszym przykładem jest poszukiwanie wzorca $w = a^{m-1}b$ w tekście $t = a^n$. Dla takiego zestawu algorytm naiwny nie znajdzie dopasowania, a wcześniej będzie dopasowywał prawie całe słowo w, rozpoczynając od każdej pozycji słowa t. Prowadzi to do wykonania dokładnie $m(n-m+1)$ porównań. □

Można również zaimplementować algorytm naiwny, który dopasowywałby wzorzec do tekstu od końca.

Algorytm 2: Naiwne szukanie wzorca w w tekście t

```
def brute_force(t, w, n, m):
  i = 1
  while i <= n - m + 1:
    j = m
    while j > 0 and t[i + j - 1] == w[j]:
      j = j - 1
    if j == 0:
      yield i
    i = i + 1
```

2.2 Algorytm Morrisa-Pratta

Algorytm naiwny dokonuje w razie wykrycia niedopasowania przesunięcia wzorca tekstu o 1. Załóżmy, że zachodzi $t[i+k+1] = w[k+1]$ dla pewnego k oraz $t[i+j] = w[j]$ dla $1 \leq j \leq k$ tj. mamy zgodność wzorca z tekstem na k literach.

Jasne jest jednak, że jeśli v nie jest prefikso-sufiksem $w[1..k]$, to istnieje pozycja $0 \leq l \leq k-1$ taka, że $v[|v|-l] \neq w[k-l] = t[i+k-l]$ – a zatem nie ma sensu dopasowywać w. Z kolei jeśli v jest prefikso-sufiksem $w[1..k]$, to z góry wiemy, że jest zgodne z $t[(i+k-|v|)..(i+k)]$.

Wobec tego najmniejsze sensowne przesunięcie to przejście do sprawdzania dla największego prefikso-sufisku $w[1..k]$ – czyli przesunięcie o $p(w[1..k])$.

Dla słowa w można zdefiniować tablice najdłuższych prefikso-sufiksów dla każdego prefiksu:

$$B[i] = \begin{cases} -1 & \text{dla } i = 0, \\ \max\{k \geq 0 : w[1 \ldots k] \text{ jest właściwym sufiksem } w[1 \ldots i]\} & \text{dla } 1 \leq i \leq |w|. \end{cases}$$

Lemat 2.1. *Jeśli u jest prefikso-sufiksem v i v jest prefikso-sufiksem w, to u jest prefikso-sufiksem w.*

Wniosek 2.2. Ciąg $(B[|w|], B[B[|w|]], \ldots, 0)$ zawiera długości wszystkich prefikso-sufiksów słowa w w kolejności malejącej.

Wniosek 2.3. Ciąg $(|w|, \ldots, |w| - B[B[|w|]], |w| - B[|w|])$ zawiera długości wszystkich okresów słowa w w kolejności malejącej.

Algorytm 3: Tablica prefikso-sufiksów

```
def prefix_suffix(w, m):
  B, t = [-1] + [0] * m, -1
  for i in range(1, m + 1):
    while t >= 0 and w[t + 1] != w[i]:
      t = B[t] # prefikso-sufiks to relacja przechodnia
    t = t + 1
    B[i] = t
  return B
```

Problem 2.2. Pokaż, jaki dokładny pesymistyczny czas działania ma Algorytm 3.

Algorytm 4: Algorytm Morrisa-Pratta

```
def morris_pratt_string_matching(t, w, m, n):
  B = prefix_suffix(w)
  i = 0
  while i <= n - m + 1:
```

```
    j = 0
    while j < m and t[i + j + 1] == w[j + 1]:
      j = j + 1
    if j == m:
      return True
    i, j = i + j - B[j], max(0, B[j])
  return False
```

Problem 2.3. Pokaż, ile razy w najgorszym razie pojedynczy symbol tekstu t może zostać porównany z wzorcem w Algorytmie 4.

Problem 2.4. Pokaż, ile dokładnie porównań wykonuje w najgorszym przypadku zmodyfikowany Algorytm 4, zwracający wszystkie wystąpienia (tj. pozycje indeksów początkowych) w w t.

2.3 Algorytm Knutha-Morrisa-Pratta

Przesunięcie o $p(w[1..k]) = k - B[k]$ nie musi być najlepsze możliwe. Jeśli niedopasowanie zaszło na znakach $t[i+k+1] \neq w[j+1]$ oraz wiadomo z analizy wzorca, że $w[k+1] = w[k-p(w)+1]$, to wiadomo, że nie znajdziemy wzorca przez przesunięcie tylko o $p(w)$. Za to możemy szukać prefikso-sufiksu w' dla słowa $w[1..j]$ takiego, że następuje po nim znak inny niż $w[j+1]$ – czyli tzw. silnego prefikso-sufiksu.

Dla słowa w można zdefiniować tablice silnych prefikso-sufiksów następująco:

$$sB[i] = \begin{cases} \max\{0 \leq k < i : w[1\ldots k] \text{ jest sufiksem } w[1\ldots i] \\ \qquad\qquad \text{i } w[k+1] = w[i+1]\} & \text{dla } 1 \leq i \leq |w|-1, \\ B[i] & \text{dla } i = |w|, \\ -1 & \text{w pozostałych przypadkach.} \end{cases}$$

Algorytm 5: Tablica silnych prefikso-sufiksów

```
def strong_prefix_suffix(w, m):
  sB, t = [-1] + [0] * m, -1
  for i in range(1, m + 1): # niezmiennik: t = B[i - 1]
    while t >= 0 and w[t + 1] != w[i]:
      t = sB[t]
    t = t + 1
    if i == m or w[t + 1] != w[i + 1]:
      sB[i] = t
    else:
      sB[i] = sB[t]
  return sB
```

Problem 2.5. Pokaż, jaki dokładny pesymistyczny czas działania ma Algorytm 5.

Algorytm 6: Algorytm Knutha-Morrisa-Pratta

```
def knuth_morris_pratt_string_matching(t, w, m, n):
  sB = strong_prefix_suffix(w)
  i = 0
  while i <= n - m + 1:
    j = 0
    while j < m and t[i + j + 1] == w[j + 1]:
      j = j + 1
    if j == m:
      return True
    i, j = i + j - sB[j], max(0, sB[j])
  return False
```

Problem 2.6. Pokaż, ile razy w najgorszym razie pojedynczy symbol tekstu t może zostać porównany z wzorcem w Algorytmie 6.

Problem 2.7. Pokaż, ile dokładnie porównań wykonuje w najgorszym przypadku zmodyfikowany Algorytm 6, zwracający wszystkie wystąpienia (tj. pozycje indeksów początkowych) w w t.

2.4 Algorytm Boyera-Moore'a

Algorytm Boyera-Moore'a polega na ulepszeniu naiwnego dopasowywania wzorca od prawej do lewej. Schemat jest taki sam jak dla algorytmów Morrisa-Pratta i Knutha-Morrisa-Pratta: wykonujemy dopasowanie dla danej pozycji okna, w razie znalezienia niedopasowania przesuwamy okno o pewną wartość.

Sprawdzanie dopasowania wzorca od prawej do lewej umożliwia, żeby algorytm wykonał $O\left(\frac{m}{n}\right)$ porównań np. dla $t = 0^n$, $w = 0^{m-1}1$. Wystarczy np. wiedzieć, że wzorzec nie jest okresowy tj. $p(w) = |w|$ oraz że zachodzi $t[i] \neq w[m]$, aby następne sprawdzanie rozpocząć od pozycji $i + m$. Zauważmy, że w algorytmie KMP zawsze musimy porównać każdą literę tekstu ze wzorcem.

W praktyce to powoduje, że algorytm Boyera-Moore'a jest zauważalnie szybszy niż KMP lub MP.

Sednem algorytmu Boyera-Moore'a są 2 niezmienniki:

- $cond_1(j, s)$ – dla każdego $j < k \leq m$ zachodzi $s \geq k$ lub $w[k - s] = w[k]$,
- $cond_2(j, s)$ – dla $s < j$ zachodzi $w[j - s] \neq w[j]$.

Wówczas możemy zdefiniować tablicę BM w następujący sposób:

$$BM[j] = \begin{cases} \min\{s > 0 : cond_1(j, s) \wedge cond_2(j, s)\} & \text{dla } 1 \leq j < m, \\ p(w) & \text{dla } j \in \{0, m\}. \end{cases}$$

Tablica ta, podobnie jak tablica silnych prefikso-sufiksów w algorytmie KMP, określa o ile możemy przesunąć wzorzec względem tekstu w razie niedopasowania – tylko od końca.

Aby efektywnie obliczyć tablicę BM będzie potrzebna tablica prefikso-prefiksów:

$$PREF[j] = \begin{cases} \max\{s \geq 0 : w[j : j+s-1] \text{ jest prefiksem } w\} & \text{dla } 1 \leq j \leq m, \\ -1 & \text{dla } j = 0. \end{cases}$$

Algorytm 7: Tablica prefiksowo-prefiksowa

```
def prefix_prefix(w, m):
  def naive_scan(i, j):
    r = 0
    while i + r <= m and j + r <= m and w[i + r] == w[j + r]:
      r += 1
    return r
  PREF, s = [-1] * 2 + [0] * (m - 1), 1
  for i in range(2, m + 1):
    # niezmiennik: s jest takie, ze s + PREF[s] - 1 jest maksymalne i PREF[s] > 0
    k = i - s + 1
    s_max = s + PREF[s] - 1
    if s_max < i:
      PREF[i] = naive_scan(i, 1)
      if PREF[i] > 0:
        s = i
    elif PREF[k] + k - 1 < PREF[s]:
      PREF[i] = PREF[k]
    else:
      PREF[i] = (s_max - i + 1) + naive_scan(s_max + 1, s_max - i + 2)
      s = i
  PREF[1] = m
  return PREF
```

Twierdzenie 2.4. *Algorytm 7 wyznacza poprawną tablicę prefikso-prefiksów dla słowa w długości m.*

Dowód. Niech tablica $PREF$ będzie poprawnie wypełniona dla wszystkich $2 \leq j < i$ oraz niech $1 \leq s < i$ będzie wartością, dla której $s + PREF[s]$ jest maksymalne. Niech ponadto $s_{max} = s + PREF[s] - 1$, $k = i - s + 1$ oraz $r = PREF[k]$

Wówczas mamy 3 sytuacje:

1. $s + PREF[s] < i$ – wówczas obliczamy wartość $PREF[i]$ naiwnie porównując w i $w[i : m]$,

2. $s+PREF[s] \geq i$ oraz $PREF[k]+k-1 < PREF[s]$ dla $k = i-s+1$ – wówczas z definicji s wiemy, że $w[s : s_{max}] = w[1 : (s_{max} - s + 1)]$ oraz $i \in [s, s_{max}]$ (z pierwszego) oraz $w[1 : r] = w[k : (r + k - 1)]$, $w[r + 1] \neq w[r + k]$ (z drugiego), wobec czego również $w[1 : r] = w[i : (r + i - 1)]$ i $w[r + 1] \neq w[r + i]$, a więc $PREF[i] = r$,
3. $s + PREF[s] \geq i$ oraz $PREF[k] + k - 1 \geq PREF[s]$ dla $k = i - s + 1$ – wówczas podobnie jak powyżej mamy $w[1 : (s_{max} - i + 1)] = w[i : s_{max}]$, więc sprawdzamy wydłużanie zgodności, naiwnie porównując $w[s_{max} - i : m]$ oraz $w[s_{max} + 1 : m]$.

□

Twierdzenie 2.5. *Algorytm 7 dla słowa długości m działa w czasie $O(m)$.*

Dowód. Wystarczy zauważyć, że w funkcji `naive_scan` kolejne wartości $j + r$ w ramach wszystkich wywołań tworzą ciąg rosnący. □

Algorytm 8: Tablica maksymalnych sufiksów

```
def maximum_suffixes(w, m):
  w_rev = w[0] + w[-1:0:-1]
  PREF = prefix.prefix_prefix(w_rev, m)
  S = [PREF[0]] + PREF[-1:0:-1]
  return S
```

Algorytm 9: Tablica przesunięć według dobrych sufiksów

```
def boyer_moore_shift(w, m):
  B = prefix.prefix_suffix(w, m)
  BM = [m - B[m]] + [m] * m
  S, j = maximum_suffixes(w, m), 0
  for k in range(m - 1, -1, -1):
    if k == S[k]:
      while j < m - k:
        BM[j] = m - k
        j += 1
  for k in range(1, m):
    BM[m - S[k]] = m - k
  return BM
```

Problem 2.8. Pokaż, że Algorytm 9 poprawnie oblicza wartości tablicy BM w czasie $O(m)$.

Problem 2.9. Pokaż, jak policzyć tablicę $wBM = \min\{s > 0 : cond_1(j, s)\}$ w czasie $O(m)$.

Algorytm 10: Algorytm Boyera-Moore'a

```
def boyer_moore(t, w, n, m):
  BM = suffix.boyer_moore_shift(w, m)
  i = 1
  while i <= n - m + 1:
    j = m
    while j > 0 and t[i + j - 1] == w[j]:
      j = j - 1
    if j == 0:
      yield i
    i = i + BM[j]
```

Problem 2.10. Pokaż, że dla wszystkich n, m istnieje tekst t i wzorzec w ($|t| = n$, $|w| = m$) takie, że wyszukanie wzorca w w tekście t z użyciem algorytmu Boyera-Moore'a wymaga czasu $O(n + m)$ a dla algorytmu wykorzystującego tablicę $wBM = \min\{s > 0 : cond_1(j, s)\}$ zamiast BM wymaga czasu $\Omega(nm)$.

2.4.1 Znajdowanie wszystkich wystąpień wzorca

Tak jak algorytm KMP, tak algorytm BM można łatwo zmodyfikować, by zwracał wszystkie wystąpienia wzorca w tekście. Wystarczy przyjąć, że $BM[0] = p(w)$.

Problem 2.11. Pokaż, że jeśli tekst t zawiera r wystąpień wzorca w, to czas działania algorytmu Boyera-Moore'a wynosi $\Omega(rm)$.

Wniosek 2.6. Jeśli wzorzec występuje w tekście $\Theta(n)$ razy, to czas działania algorytmu Boyera-Moore'a wynosi $\Omega(nm)$.

Rozwiązaniem jest wykorzystanie zmiennej $memory$, zawierającej numer pierwszego indeksu, którego nie musimy sprawdzać. W większości obrotów pętli $memory = 0$, jedynym wyjątkiem jest sytuacja, gdy zaszło dopasowanie – wtedy wiemy, że $m - p(w)$ pierwszych symboli już jest dopasowanych i wystarczy porównać $w[(m - p(w) + 1)..m]$ z tekstem.

Algorytm 11: Algorytm Boyera-Moore'a-Galila

```
def boyer_moore_galil(t, w, n, m):
  BM = suffix.boyer_moore_shift(w, m)
  i, memory = 1, 0
  while i <= n - m + 1:
    j = m
    while j > memory and t[i + j - 1] == w[j]:
      j = j - 1
    if j == memory:
      yield i
```

```
    i, memory = i + BM[0], m - BM[0]
  else:
    i, memory = i + BM[j], 0
```

2.4.2 Dowód liniowej złożoności algorytmu

Dowód Cole'a

Twierdzenie 2.7. *Algorytm 10 wymaga $O(n' + m)$ czasu do stwierdzenia pierwszego wystąpienia wzorca w z tekście t, gdy to wystąpienie zostało znalezione na pozycji n'.*

Wniosek 2.8. Algorytm 10 wymaga $O(n + m)$ czasu do stwierdzenia, czy słowo t zawiera podsłowo w.

Twierdzenie 2.9. *Algorytm 11 działa w czasie $O(n + m)$.*

Dowód. Z poprzedniego twierdzenia łatwo wykazać, że znalezienie wszystkich wystąpień wymaga czasu $O(n + rm)$ dla r wystąpień wzorca w tekście dla Algorytmu 10, a więc również dla Algorytmu 11.

Jeśli $p(w) \geq \frac{m}{2}$, to z faktu $r \leq \frac{n}{p(w)}$ wynika, że całkowita złożoność wynosi $O(n)$.

Jeśli $p(w) < \frac{m}{2}$, to możemy pogrupować wystąpienia wzorca jeśli dwa kolejne w ciągu są odległe o $p(w)$. W każdym takim łańcuchu w Algorytmie 11 przejrzymy każdy symbol w łańcuchu tylko raz. Jeśli natomiast wykryjemy niedopasowanie, to wiemy, że następne przesuniecie zgodnie z dobrym sufiksem wynosi $|w| - p(w) \geq \frac{m}{2}$. Wobec każdy symbol poza łańcuchami odczytamy co najwyżej 2 razy, a zatem złożoność również będzie liniowa. □

2.4.3 Heurystyka *occurrence shift*

W oryginalnym artykule Boyera i Moore'a zaproponowano również dodatkową heurystykę *occurrence shift*, opartą na znajomości wystąpień liter z alfabetu we wzorcu.

Zdefiniujmy tablicę $LAST$ w następujący sposób:

$$L[x] = \begin{cases} \max\{1 \leq i \leq m : w[i] = x \wedge \forall_{i<j\leq m} w[j] \neq m\} & \text{gdy } x \text{ występuje w } w, \\ 0 & \text{w przeciwnym przypadku.} \end{cases}$$

Jeśli wiemy, że niedopasowanie nastąpiło na $w[j] \neq t[i+j]$, to wiemy również, że nie ma sensu sprawdzać przesunięć krótszych niż $j - LAST[t[i + j]]$. Mówiąc prościej, jeśli $t[i + j] = x$ oraz x występuje w w najpóźniej na pozycji k (tj. $k = LAST[t[i + j]]$), to mamy 2 sytuacje:

- albo $k > j$, wtedy wiedza z tablicy $LAST$ nam nic nie pomaga,
- albo $k < j$, wtedy wiemy, że pierwszą okazją na znalezienie dopasowania jest zestawienie ze sobą $w[k]$ i $t[i + j]$ – czyli właśnie przesunięcie o $j - k$.

Dla znanego, skończonego alfabetu widać wprost, że wyznaczenie tablicy $LAST$ wymaga czasu i pomięci $O(m + |\mathcal{A}|)$.

Ponieważ heurystyka nie wyklucza się z obserwacjami dokonanymi na bazie tablicy BM, jej zastosowanie polega na zastąpieniu przesunięć o $BM[j]$ przesunięciami o $\max\{BM[j], j - LAST[t[i+j]]\}$.

W praktyce heurystyka ma sens wtedy, gdy alfabety są większe, ale dla alfabetu binarnego jest mało skuteczna. Co ciekawe, brakuje teoretycznych analiz skuteczności użycia tej heurystyki np. względem podstawowego algorytmu Boyera-Moore'a.

Problem 2.12. Niech w Algorytmie 10 w linii ??? będzie miał zamiast $i = i + BM[j]$ polecenie $i = i + \max\{1, j - LAST[t[i+j]]\}$. Pokaż dla dowolnych n i m przykłady słów t i w ($|t| = n$, $|w| = m$) takich, że zmodyfikowany algorytm wymaga $\Omega(nm)$ porównań.

Ten sam pomysł pojawia się u Horspoola, który postanowił korzystać tylko z przesuwania o *occurrence shift* zgodnie z ostatnim znakiem aktualnie przetwarzanego tekstu.

Algorytm 12: Algorytm Horspoola

```
def horspool(t, w, n, m):
  LAST = suffix.last_occurrence(w[:-1])
  i = 1
  while i <= n - m + 1:
    c = t[i + m - 1]
    if w[m] == c:
      j = 1
      while j < m and t[i + j - 1] == w[j]:
        j = j + 1
      if j == m:
        yield i
    bad_character = LAST.get(c, 0)
    i = i + (m - bad_character)
```

Problem 2.13. Pokaż dla dowolnych n i m przykłady słów t i w ($|t| = n$, $|w| = m$) takich, że algorytm Horspoola wymaga $\Omega(nm)$ porównań.

Raita zaproponował dodatkowo, żeby zamiast przechodzić do porównania tekstu z wzorcem za każdym razem wykonać najpierw porównanie symbolu ostatniego, pierwszego i środkowego. Uzasadniał to tym, że wyszukiwanie słów w językach naturalnych często natrafia na problem częstych sufiksów (np. w angielskim *-ing*, *-ion*, *-ed*) Raita, 1992, aczkolwiek dalsze eksperymenty na tekstach losowych również przyniosły podobne rezultaty, co sugeruje że korzyści mogą być spowodowane czymś innym Smith, 1994.

2.4.4 Quick search

Można zaproponować również inną heurystykę opartą o *occurrence shift*: jeśli zaszło niedopasowanie na fragmencie tekstu $t[(i-j)..(i-1)]$, to wiemy, że następne porównanie tekstu z wzorcem będzie zawierało

$t[i]$. Wobec tego można od razu przesunąć tekst tak, aby uzgodnić ostatnie wystąpienie litery $t[i]$ we wzorcu z tekstem.

Algorytm 13: Algorytm *quick search*

```
def quick_search(t, w, n, m):
  LAST = suffix.last_occurrence(w)
  i = 1
  while i <= n - m + 1:
    j = 1
    while j <= m and t[i + j - 1] == w[j]:
      j = j + 1
    if j == m + 1:
      yield i
    bad_character = LAST.get(t[i + m], 0) if i + m <= n else 0
    i = i + (m + 1 - bad_character)
```

Smith zauważył, że można połączyć podejście Horspoola i Quick Search tj. brać maksimum z obu zalecanych przesunięć, aby otrzymać praktyczne przyspieszenie algorytmu Smith, 1991.

2.5 Algorytm *fast-on-average*

Algorytm 14: Algorytm *fast-on-average*

```
def fast_on_average(t, w, n, m):
  S = build_suffix_tree(w)
  i, r = m, 2 * math.ceil(math.log(m, 2))
  while i <= n:
    if S.contains(t[(i - r):(i + 1)]):
      subt = t[0] + t[(i - m + 1):min(i - r + m - 1, n + 1)]
      subn = min(i - r + m - 1, n + 1) - (i - m + 1)
      for out in knuth_morris_pratt(subt, w, subn, m)
        yield out
    i = i + m - r
```

Twierdzenie 2.10 (Crochemore i Rytter 2002, Theorem 2.5)**.** *Algorytm 14 działa w czasie oczekiwanym* $O\left(n\frac{\log m}{m}\right)$ *i pesymistycznym* $O(n)$*. Przetwarzanie wstępne wzorca wymaga czasu* $O(m)$*.*

Dowód. Stworzenie struktury do sprawdzania, czy słowo $t[(i-r)..i]$ jest podsłowem w (np. drzewa sufiksowego lub DAWG) wymaga czasu $O(m)$.

Jeśli $t[(i-r)..i]$ nie jest podsłowem w, to w nie może być podsłowem t rozpoczynającym się na żadnej z pozycji od $i-m+1$ do $i-r$.

Słowo $t[(i-r)..i]$ ma $2^{r+1} \geq m^2$ możliwych wartości. Z drugiej strony, wiemy że w zawiera mniej niż m podsłów długości $r+1$. Jeśli t jest losowy, to prawdopodobieństwo, że $t[(i-r)..i]$ jest podsłowem w jest mniejsze niż $\frac{1}{m}$. Wobec tego po sprawdzeniu czy słowo $t[(i-r)..i]$ jest podsłowem w (w czasie $O(r)$) jedynie z prawdopodobieństwem mniejszym niż $\frac{1}{m}$ będziemy musieli wykonać algorytm KMP na tekście $t[(i-m+1)..(i-r+m-1)]$ oraz wzorcu w (w czasie $O(m)$). Łączny oczekiwany czas działania tej procedury jest równy zatem $O(r)$, natomiast pesymistyczny $O(m)$.

Każda pojedyncza iteracja rozstrzyga dopasowanie dla $m-r$ możliwych początków indeksów. Wystarczy zatem uruchamiać algorytm dla $i = 1, m-r+1, 2(m-r)+1, \ldots$ – a zatem $O\left(\frac{n}{m}\right)$ razy – i rozwiązać problem niezależnie dla każdego podprzypadku. □

2.6 Algorytm *two-way*

Definicja 2.11. Dla dowolnych słów u, v niepuste słowo w jest *powtórzeniem* (repetition) (u, v), jeśli:

- w jest sufiksem u lub u jest sufiksem w,
- w jest prefiksem v lub v jest prefiksem w.

Zwróćmy uwagę, że dla dowolnych słów u, v zawsze istnieje co najmniej jedno powtórzenie (u, v): $w = vu$.

Definicja 2.12. Dla dowolnych słów u, v lokalny okres jest zdefiniowany jako minimalna długość słowa będącego powtórzeniem (u, v).

$$lp(u, v) = \min\{|w| : w \text{ jest powtórzeniem } (u, v)\}$$

Problem 2.14. Pokaż, że $1 \leq lp(u, v) \leq p(uv)$ dla dowolnych słów u, v.

Problem 2.15. Pokaż jak w czasie $O(n)$ dla dowolnego słowa w obliczyć tablicę $lp(u, v)$ dla wszystkich u, v takich, że $w = uv$.

Definicja 2.13. Faktoryzacją krytyczną słowa w nazywamy parę słów (u, v) takich, że $uv = w$ i $lp(u, v) = p(w)$.

Problem 2.16. Pokaż, że dla dowolnego niepustego słowa w istnieje faktoryzacja krytyczna (u, v) spełniająca $|u| < p(w)$.

Potrzebne będzie znalezienie faktoryzacji krytycznej. Okazuje się, że można to zrobić na bazie algorytmu znajdującego maksymalny sufiks słowa.

Algorytm 15: Wyznaczanie faktoryzacji krytycznej dla tekstu t

```
def critical_factorization(text, n):
  index_lt, p_lt = maximum_suffix(text, n)
  index_gt, p_gt = maximum_suffix(
      text, n, less = operator.__gt__)
  return (index_lt, p_lt) if index_lt >= index_gt else (index_gt, p_gt)
```

Twierdzenie 2.14 (Apostolico i Galil 1997, Theorem 1.18, s. 40)**.** *Algorytm 15 zwraca faktoryzację krytyczną* (u, v) *dla słowa* w*. Dodatkowo,* $|u| < p(w)$*.*

Dowód. Niech $\leq$ będzie porządkiem leksykograficznym zgodnym z kolejnością liter w alfabecie, natomiast $\leq^*$ porządkiem leksykograficznym zgodnym z odwrotną kolejnością liter w alfabecie.

Dla dowolnego niepustego słowa w niech $w = uv$ z v jako maksymalnym sufiksem według $\leq$ oraz $w = u'v'$ z v' jako maksymalnym sufiksem według $\leq^*$.

Jeśli $w = a^n$ dla pewnego $a \in \mathcal{A}$, to każde (u, v) takie, że $w = uv$ jest faktoryzacją krytyczna. Jeśli nie, to $v \neq v'$. Bez straty ogólności załóżmy, że $|v| < |v'|$.

Niech x będzie najkrótszym powtórzeniem dla (u, v). Wystarczy tylko dowieść, że $|x|$ jest okresem w, bo z lematu wyżej wiadomo, że $|x| = lp(u, v) \leq p(w)$.

Możemy wyróżnić cztery przypadki:

1. x jest sufiksem u, v jest prefiksem x – ale wtedy $v < x < xv$ i xv jest sufiksem w, więc v nie byłoby maksymalnym sufiksem w,
2. x jest sufiksem u, x jest właściwym prefiksem v z $v = xz$ dla pewnego $z \in \mathcal{A}^+$ – wtedy z jest również sufiksem w, wobec tego $z < v$ i $v = xz < xv$, więc v nie byłoby maksymalnym sufiksem w,
3. u jest właściwym sufiksem x, v jest prefiksem x – wtedy $|x|$ jest okresem w,
4. u jest właściwym sufiksem x, x jest właściwym prefiksem v z $v = xz$ dla pewnego $z \in \mathcal{A}^+$ – wtedy $v' = yv$ dla pewnego $y \in \mathcal{A}^+$. Skoro v' jest sufiksem $w = uv$, to y jest sufiksem u, a zatem jest też sufiksem x. Wobec tego yz jest sufiksem v oraz całego w, a więc $yz \leq^* v' = yv$, czyli $z \leq^* v$. Zarazem $z \leq v$, bo z definicji z też jest sufiksem w – a zatem z jest prefiksem v. Wobec tego z jest prefikso-sufiksem v a $|x|$ jest okresem v. $|x|$ jest ostatecznie okresem całego w, bo w jest podsłowem x^2z.

Ponieważ pierwsze dwa przypadki kończą się sprzecznością, to wiemy, że u jest właściwym sufiksem x – a więc $|u| < |x| = p(w)$. □

Powyższe obserwacje stały się podstawą pierwszego optymalnego (tj. działającego w czasie liniowym i ze stałą dodatkową pamięcią) algorytmu wyszukiwania wzorca z tekście, tzw. *two-way algorithm*.

Załóżmy, że mamy (u, v), faktoryzację krytyczną wzorca w. Wówczas możemy najpierw sprawdzać dopasowanie odpowiedniej części tekstu do v od lewej do prawej (jak w algorytmie Morrisa-Pratta), a następnie sprawdzać dopasowanie wcześniejszej części do u od prawej do lewej (jak w algorytmie Boyera-Moore'a). Jak zwykle, w razie niedopasowania wykonujemy odpowiednie przesunięcia.

Algorytm 16: Algorytm Two-Way

```
def two_way(text, word, n, m):
  index, p, use_memory = *critical_factorization(word, m), True
  if index - 1 > m / 2 or not word[index:index + p].endswith(word[1:index]):
    p, use_memory = max(len(word[1:index]), len(word[index:])) + 1, False
  i, memory = 1, 0
```

```
while i <= n - m + 1:
  j = max(index - 1, memory)
  while j < m and text[i + j] == word[j + 1]:
    j = j + 1
  if j < m:
    i, memory = i + j + 2 - index, 0
    continue
  j = max(index - 1, memory)
  while j > memory and text[i + j - 1] == word[j]:
    j = j - 1
  if j == memory:
    yield i
  i, memory = i + p, m - p if use_memory else 0
```

Jak widać, dla jednego z przypadków zostało wykorzystane zapamiętywanie dopasowania, analogicznie jak w algorytmie Boyera-Moore'a-Galila. Okazuje się, że dla drugiego nie jest ono potrzebne, ponieważ okres wzorca jest dostatecznie długi.

Lemat 2.15 (Apostolico i Galil 1997, Lemma 1.19, s. 43)**.** *Niech* (u, v) *jest faktoryzacją krytyczną* w *taką, że* $|u| < p(w)$*. Jeśli* u *nie jest sufiksem* $v[1 \ldots p(v)]$*, to* u *występuje dokładnie raz w* w*.*

Dowód. Załóżmy, że u nie jest sufiksem $v[1 \ldots p(v)]$, ale występuje w w drugi raz. Z okresowości v wiemy, że v musi zaczynać się pewnym sufiksem u – ale wtedy $lp(u, v) \leq |u|$. To przeczy jednak założeniu, że (u, v) jest faktoryzacją krytyczną, a więc $lp(u, v) = p(w) > |u|$. □

Wniosek 2.16. Niech (u, v) jest faktoryzacją krytyczną w taką, że $|u| < p(w)$. Jeśli u nie jest sufiksem $v[1 \ldots p(v)]$, to

$$p(w) \geq \max\{|u|, |v|\} + 1 > \frac{|w|}{2}.$$

Dowód. Wprost z założenia wynika, że $p(w) \geq |u| + 1$. Z poprzedniego lematu wiemy, że u występuje dokładnie raz w w, a zatem największy możliwy prefikso-sufiks w jest nie dłuższy niż $|u| - 1$. Wobec tego $p(w) \geq |w| - (|u| - 1) = |v| + 1$.

Druga nierówność jest oczywista na mocy faktu $w = uv$. □

Twierdzenie 2.17 (Apostolico i Galil 1997, Theorem 1.18, s. 43)**.** *Algorytm 16 zwraca poprawnie wszystkie wystąpienia podsłów w tekście.*

Dowód. Przy przeglądaniu w prawo w razie niedopasowania po prostu przesuwamy wzorzec o tyle znaków, ile przejrzeliśmy.

Przy przeglądaniu w lewo mamy dwa przypadki, w zależności od tego, czy dla wzorca w i jego faktoryzacji krytycznej (u, v) rzeczywiście u jest sufiksem $v[1 \ldots p(v)]$. Jeśli tak, to postępujemy dokładnie jak w algorytmie 11. W tym przypadku zachodzi $p(w) = p(v)$.

Jeśli nie, to z 2.16 wiemy, że $p(w) \geq \max\{|u|, |v|\}+1$ – a zatem możemy ustawić minimalne przesunięcie na tę drugą wartość. Gwarantuje to zarazem, że nie musimy się martwić o przypadek nakładających się wystąpień wzorca na siebie, więc nie potrzebujemy zapamiętywania prefiksu zgodnie z regułą Galila. □

Twierdzenie 2.18. *Algorytm 16 działa w czasie $O(n+m)$ i wymaga $O(1)$ dodatkowej pamięci.*

Dowód. W dowodzie zakładamy, że wyznaczanie maksymalnego sufiksu jest wykonywane w czasie $O(n+m)$ i w $O(1)$ dodatkowej pamięci np. wykorzystując 25. Wykorzystanie pamięci $O(1)$ przez cały algorytm jest oczywiste.

Jeśli patrzymy na pozycje tekstu t porównywane tylko przy przeglądaniu w prawo w ciągu całego algorytmu, to tworzą one ciąg rosnący. Wobec tego takich porównań mamy co najwyżej n.

Jeśli patrzymy na pozycje tekstu t porównywane tylko przy przeglądaniu w prawo w ciągu całego algorytmu, to żadna nie zostanie porównana dwa razy, ponieważ między kolejnymi iteracjami głównej pętli przesuwamy okno o $p > |u|$ (na mocy dodatkowego warunku dla faktoryzacji krytycznej), a w każdym oknie sprawdzamy co najwyżej $|u|$ znaków w ten sposób. Takich porównań również mamy co najwyżej n. □

2.7 Algorytm Karpa-Rabina

Alternatywne podejście można oprzeć o obliczanie funkcji skrótu h dla poszczególnych podsłów długości m. Jeśli funkcja jest zwykłą liniową funkcją modulo, to łatwo obliczyć $h(t[i+1..m+i+1])$ na podstawie $h(t[i..m+i])$.

Algorytm 17: Algorytm Karpa-Rabina

```
def karp_rabin(text, word, n, m):
  MOD = 257
  A = random.randint(2, MOD - 1)
  Am = pow(A, m, MOD)
  def generate(t):
    return sum(ord(c) * pow(A, i, MOD) for i, c in enumerate(t[::-1])) % MOD
  text += '$'
  hash_text, hash_word = generate(text[1:m + 1]), generate(word[1:])
  for i in range(1, n - m + 2):
    if hash_text == hash_word and text[i:i + m] == word[1:]:
      yield i
    hash_text = (A * hash_text + ord(text[i + m]) - Am * ord(text[i])) % MOD
```

Warto zwrócić uwagę, że algorytm można zaimplementować w dwóch wersjach:

- Monte Carlo – zwracamy dopasowanie od razu, gdy wykryjemy równość haszy, więc czas działania to $O(n)$, ale możliwe fałszywe dopasowania,
- Las Vegas – zwracamy dopasowanie jedynie, gdy wykryjemy równość haszy i sprawdzimy z całym wzorcem, stąd brak błędów, ale pesymistyczny czas działania to $O(nm)$.

3 Indeksowanie tekstu

Problem 3: Indeksowanie tekstu

Wejście: Słowo $t \in \mathcal{A}^+$ ($|t| = n$)

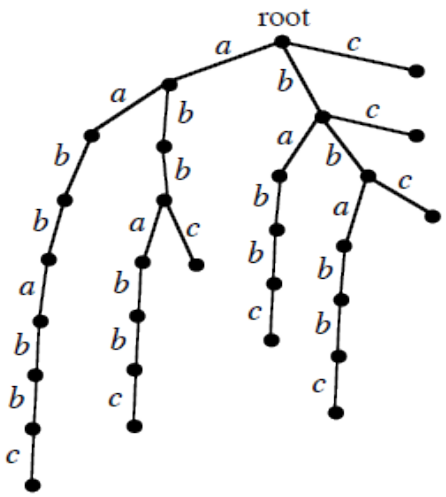
Wyjście: Struktura danych, na której można wykonywać zapytania *czy w jest podsłowem t* dla dowolnego $w \in \mathcal{A}^*$ ($|w| = m$).

Typowo zakładamy, że $m \ll n$ oraz wymagamy, aby zapytanie było wykonywalne w czasie $O(m)$, niezależnym od n. Czasem dopuszczamy, aby czas wykonania zapytań był logarytmiczny względem n.

Dodatkowo wstępne przetwarzanie tekstu powinno było wykonalne w czasie liniowym lub przynajmniej liniowo-polilogarytmicznym względem n (oraz rozmiaru alfabetu $\mathcal{A}$).

3.1 Drzewo sufiksowe

Drzewo sufiksowe (*suffix trie*) to jedna ze struktur danych, która umożliwia łatwe sprawdzanie podsłów. W najprostszej, nieskompresowanej wersji jest to zwykłe drzewo słownikowe $Trie(t)$ zawierające wszystkie sufiksy słowa t.



Sprawdzenie, czy słowo w jest podsłowem t polega po prostu na sprawdzeniu, czy w drzewie sufiksowym $Trie(t)$ istnieje ścieżka zgodna z kolejnymi literami słowa w.

Pewnym ułatwieniem rozważań na drzewach sufiksowych jest zastosowanie specjalnego znaku $\$ \notin \mathcal{A}$ na końcu t. W ten sposób zapewniamy, że każdy sufiks t kończy się w liściu tj. żaden sufiks nie jest prefiksem innego sufiksu.

Problem 3.1. Dla dowolnego n pokaż tekst t długości n taki, że rozmiar drzewa sufiksowego jest możliwie największy.

Ponieważ podstawowe drzewo sufiksowe może mieć rozmiar większy niż liniowy, to wygodniejsze jest przejście na skompresowane drzewo sufiksowe (*suffix tree*) $ST(t)$ tj. drzewo, w którym ściągnięto wszystkie wierzchołki stopnia 2 różne od korzenia. Każda krawędź w drzewie zamiast odpowiadać pojedynczej literze, odpowiada teraz pewnemu słowu z $\mathcal{A}^+$. Dowolne dwie krawędzie wychodzące z tego samego wierzchołka odpowiadają słowom, których największych prefiks jest pusty (tj. różnią się pierwszym znakiem). Oczywiście sprawdzenie, czy w jest podsłowem t przebiega dokładnie tak samo jak poprzednio.

Twierdzenie 3.1. *Skompresowane drzewo sufiksowe dla dowolnego słowa t o długości n ma rozmiar $O(n)$.*

Dowód. Wiemy, że drzewo $ST(t)$ ma co najwyżej n liści (jeśli kończy się symbolem specjalnym, to ma dokładnie n). Z definicji każdy wierzchołek wewnętrzny drzewa skompresowanego ma co najmniej dwa poddrzewa, wobec tego liczba wierzchołków wewnętrznych musi być mniejsza od liczby liści. □

Z powyższych twierdzeń jasne jest, że konstrukcja skompresowanego drzewa sufiksowego przez najpierw zbudowanie, a następnie kompresję drzewa nieskompresowanego jest nieoptymalna.

Optymalne podejścia przedstawione poniżej polegają na iteracyjnym rozbudowywaniu drzewa sufiksowego przez wstawianie kolejnych sufiksów we właściwej kolejności, od najdłuższego lub od najkrótszego. Okazuje się, że samo to jednak nie wystarczy:

Problem 3.2. Dla dowolnego n pokaż tekst t ($|t| = n$) taki, że wstawianie sufiksów do drzewa sufiksowego w dowolnej kolejności zaczynając zawsze od korzenia wymaga $\Theta(n^2)$ operacji.

3.1.1 Algorytm McCreighta

W omawianiu algorytmów będziemy korzystać oznaczenia $ST(t, i)$ jako (tymczasowego) drzewa słownikowego, zawierającego i najdłuższych sufiksów t. Oznaczmy przez $leaf(i)$ liść wstawiony przy przejściu z $ST(t, i-1)$ do $ST(t, i)$, odpowiadający sufiksowi $t[i..n]$. Dodatkowo, $head(i) = parent(leaf(i))$.

Ponadto, zdefiniujmy dla każdego wierzchołka v funkcję $word(v)$ przypisującą mu słowo złożone z etykiet ścieżki od korzenia do v. Jest to szczególnie użyteczne do definicji linków sufiksowych, wykorzystywanego w algorytmie: jeśli $word(v) = aw$ ($a \in \mathcal{A}$, $w \in \mathcal{A}^*$), to $S[v] = u$, gdzie u jest wierzchołkiem, dla którego $word(u) = w$.

Algorytm 18: Algorytm McCreighta budowania drzewa sufiksowego

```
def break_node(parent, child, index):
  u = trie.TrieNode(child.label[:index])
  child.label = child.label[index:]
  u.add_child(child)
  parent.add_child(u)
  return u

def fast_find(v, w):
  index = 0
```

```
  while index < len(w) and w[index] in v.children:
    child = v.children[w[index]]
    if index + len(child.label) > len(w):
      return break_node(v, child, len(w) - index)
    v, index = child, index + len(child.label)
  return v

def slow_find(v, w):
  index = 0
  while index < len(w) and w[index] in v.children:
    child = v.children[w[index]]
    for i, c in enumerate(child.label):
      if w[index + i] != c:
        return break_node(v, child, i), len(w) - (index + i)
    v, index = child, index + len(child.label)
  return v, len(w) - index

def mccreight(text, n):
  text = text + '$'
  root, leaf = trie.TrieNode(""), trie.TrieNode(text[1:])
  root.add_child(leaf)
  S, head = { }, root
  for i in range(2, n + 1):
    # niezmiennik: S[v] jest zdefiniowane dla wszystkich v != head(i - 1)
    if head == root:
      # wyjątek 1: drzewo z jednym liściem
      beta, gamma, v = "", head.children[leaf.label[0]].label[1:], root
    else:
      if head.parent == root:
        # wyjątek 2: head.parent jest rootem
        beta = head.parent.children[head.label[0]].label[1:]
      else:
        beta = head.parent.children[head.label[0]].label
      gamma = head.children[leaf.label[0]].label
      v = fast_find(S[head.parent], beta)
    S[head] = v
    head, remaining = slow_find(v, gamma)
    leaf = trie.TrieNode(text[-remaining:])
    head.add_child(leaf)
  return root
```

W dowodzie optymalności algorytmu McCreighta wykorzystywane będą następujące trzy własności:

Problem 3.3. Pokaż, że $head(i)$ jest potomkiem $S[head(i-1)]$ w $ST(t,i)$.

Problem 3.4. Pokaż, że $S[v]$ jest potomkiem $S[parent(v)]$ dla dowolnego v w każdym $ST(t,i)$.

Problem 3.5. Pokaż, że $depth(v) \leq depth(S[v]) + 1$ dla dowolnego $v \neq head(i)$ w każdym $ST(t,i)$.

Twierdzenie 3.2. *Algorytm McCreighta zwraca poprawne drzewo sufiksowe.*

Dowód. Na początek załóżmy, że dla dowolnego $i = 2, 3, \ldots, n$ mamy zbudowane $ST(t, i-1)$ oraz wszystkie wierzchołki wewnętrzne $ST(t,i-1)$ inne niż $head(i-1)$ mają dobrze zdefiniowane $S[v]$ (samo $S[head(i-1)]$ też może być zdefiniowane wcześniej, ale nie musi). Ten niezmiennik jest utrzymany przez cały algorytm.

Oczywiście $ST(t,1)$ złożone z korzenia r, liścia l_1 i krawędzi między nimi z etykietą t spełnia te warunki.

Korzystając obu własności z problemów powyżej wiemy, że $S[parent(head(i-1)]$ jest dobrze zdefiniowany oraz istnieje ścieżka z $S[parent(head(i-1)]$ do $S[head(i-1)]$ w $ST(t,i-1)$ – jedynie z tym zastrzeżeniem, że może kończyć się „w środku" krawędzi tj. z $head(i-1)$ jako wierzchołkiem skompresowanym.

Wystarczy zatem dodać wierzchołek $leaf(i)$, być może rozbijając krawędź i tworząc nowy wierzchołek $head(i-1)$, oraz nadać wartość $S[head(i-1)]$ tak, żeby $S[head(i-1)]$ było przodkiem $head(i)$, aby stworzyć drzewo, które jest $ST(t,i)$ oraz ma dobrze zdefiniowane $S[v]$ dla wszystkich wierzchołków wewnętrznych v poza $head(i)$. □

Twierdzenie 3.3. *Algorytm McCreighta ma złożoność $O(n \log |\mathcal{A}|)$.*

Dowód. Złożoność czasowa algorytmu wynika wprost z trzech faktów.

Po pierwsze, poza funkcjami `fast_find` i `slow_find` algorytm wykonuje stałą liczbę operacji w każdej iteracji – a zatem $O(n)$ operacji łącznie. Dla alfabetu $\mathcal{A}$ znalezienie w wierzchołku odpowiedniej krawędzi zaczynającej się od danego symbolu wynosi $O(\log |\mathcal{A}|)$. Wszystkie pozostałe operacje wykonywane są w czasie stałym.

Po drugie, niech funkcja `fast_find` schodzi od wierzchołka $S[parent(head(i-1))]$ w dół do wierzchołka $head(i)$ po dokładnie d_i krawędziach. Wówczas wiemy, że

$$depth(head(i-1)) \leq depth(S[head(i-1)]) + 1 = depth(head(i)) - d_i + 1.$$

Wobec tego całkowita liczba przejść po krawędziach w funkcji `fast_find` wynosi

$$\sum_{i=2}^{n+1} d_i \leq \sum_{i=2}^{n+1} (depth(head(i)) - depth(head(i-1)) + 1) \leq depth(head(n)) + n \leq 2n$$

Po trzecie, funkcja `slow_find` w i-tej iteracji algorytmu wykonuje s_i porównań, gdzie zachodzi $s_i \leq |word(head(i))| - |word(head(i-1))|$. Analogicznie jak powyżej, łączna liczba operacji jest ograniczona z góry przez $O(n)$. □

Warto zauważyć, że jeśli $S[head(i-1)]$ jednak jest dobrze określone w i-tym kroku algorytmu, to możemy od razu przejść do tego wierzchołka, zamiast wykonywać przejście przez rodzica $head(i-1)$ (tzw. *up-link-down*). Taka optymalizacja nie zmienia jednak asymptotycznego czasu wykonania algorytmu.

Problem 3.6. Pokaż, że jeśli w Algorytmie 18 zamiast `fast_find` użyjemy funkcji `slow_find`, to dla dowolnego n istnieje tekst t długości n taki, że konstrukcja $ST(t)$ wymaga czasu $\Omega(n^2)$.

3.1.2 Algorytm Ukkonena

Algorytm Ukkonena polega na tym, że w tym przypadku budowane jest w zasadzie poprawne drzewo sufiksowe dla kolejnych prefiksów słowa t. W trakcie wykonania algorytmu wyznaczamy dokładnie taką samą tablicę linków sufiksowych jak w algorytmie McCreighta.

Algorytm 19: Algorytm Ukkonena budowania drzewa sufiksowego

```
def ukkonen(text, n):
  text = text + '$'
  root, leaf = trie.TrieNode(''), trie.TrieNode(text[1:])
  root.add_child(leaf)
  S, head, shift = {root : root}, root, 0
  for i in range(2, n + 2):
    # niezmiennik: S[v] jest zdefiniowane dla wszystkich v != head(i - 1)
    child = head.children.get(text[i - shift])
    if (child is None or shift >= len(child.label)
        or text[i] != child.label[shift]):
      previous_head = None
      while shift > 0 or text[i] not in head.children:
        v = (break_node(head, head.children[text[i - shift]], shift)
             if shift > 0 else head)
        v.add_child(trie.TrieNode(text[i:]))
        if head == root:
          shift -= 1
        if previous_head is not None:
          S[previous_head] = v
        previous_head, head = v, S[head]
        if shift > 0:
          head, shift = fast_find(head, text[i - shift:i], split = False)
      if previous_head is not None:
        S[previous_head] = head
      child = head.children.get(text[i - shift]) if shift >= 0 else None
    if child is not None and len(child.label) == shift + 1:
      head, shift = head.children[text[i - shift]], 0
    else:
```

```
        shift += 1
    return root, S
```

Problem 3.7. Pokaż, że wartości $S[v]$ wyznaczane w algorytmach McCrieghta i Ukkonena są identyczne.

Twierdzenie 3.4. *Algorytm Ukkonena zwraca poprawne drzewo sufiksowe.*

Dowód. Przede wszystkim zauważmy, że każdy nowododany liść pozostaje zawsze liściem do samego końca wykonania algorytmu – w ten sposób $leaf(i)$ nie tylko reprezentuje sufiks $t[i]$ w $ST(t,i)$, ale również sufiks $t[i..j]$ w $ST(t,j)$ dla dowolnego $j \geq i$.

Należy przy tym pamiętać, że w rzeczywistości, inaczej niż w powyższym algorytmie, pamiętamy nie całą etykietę na krawędzi, tylko parę liczb oznaczających początek i koniec etykiety w tekście. Wystarczy zatem ustawić indeksy końcowe w liściach na $+\infty$, unikając w ten sposób problemu z dodawaniem kolejnych znaków na końcach liści.

Aby z $ST(t,i-1)$ otrzymać $ST(t,i)$ wystarczy wstawić do drzewa wszystkie sufiksy słowa $t[1..i]$. Załóżmy jednak, że pewnym momencie natrafimy na (być może skompresowany) wierzchołek v, dla którego już istnieje (być może skompresowany) wierzchołek v' spełniający $word(v) = word(v')\,t[i]$. Wówczas wiemy, że w drzewie nieskompresowanym dla wszystkich wierzchołków u odpowiadających sufiksom słowa $word(v)$ istnieją wierzchołki u' takie, że $word(u) = word(u')\,t[i]$. Wobec tego wszystkie mniejsze sufiksy słowa $t[1..i]$ już muszą istnieć.

Idea algorytmu jest zatem taka: w i-tej iteracji wychodzimy od $head(i-1)$ (zob. dowód algorytmu McCreighta) i chodząc po linkach sufiksowych rozbudowujemy drzewo o nowe liście zgodnie z $t[i]$ aż do momentu, gdy natrafimy na istniejący (może skompresowany) wierzchołek. Tak otrzymane drzewo (gdybyśmy zastąpili wszystkie indeksy końcowe $+\infty$ przez i) jest drzewem sufiksowym $ST(t,i)$. □

Twierdzenie 3.5. *Algorytm Ukkonena ma złożoność* $O(n \log |\mathcal{A}|)$.

Dowód. Wszystkie operacje poza `fast_find` wymagają czasu $O(\log |\mathcal{A}|)$ na każdą iterację głównej pętli.

Pętla `while` za każdym razem dodaje jeden liść, więc podczas całego wykonania algorytmu funkcja `fast_find` zostanie wykonana $O(n)$ razy. Ponieważ wiemy, że $S[v]$ przyjmuje dokładnie takie same wartości, jak w algorytmie McCreighta, to również tu całkowita liczba odwiedzionych wierzchołków w funkcji `fast_find` wyniesie $O(n)$. □

3.1.3 Dalsze zagadnienia

Giegerich i Kurtz (1997) pokazali, że chociaż teoretyczne idee u podstaw wszystkich trzech algorytmów się różnią, to istnieje głębokie podobieństwo strukturalne. W szczególności można pokazać, że algorytmy McCreighta i Ukkonena wykonują dokładnie te same ciągi abstrakcyjnych operacji, a zatem możliwe jest tłumaczenie krok po kroku wykonania jednego algorytmu na drugi. Również algorytm Weinera przypomina strukturalnie algorytm Ukkonena z tą różnicą, że jest budowany „od końca” i przez to korzysta jedynie z „przybliżonych” linków.

Okazuje się, że możliwa jest konstrukcja drzewa sufiksowego w czasie $O(n)$, niezależnym od $\mathcal{A}$ (Farach, 1997). Wymaga to co prawda utożsamienia alfabetu z podzbiorem liczb naturalnych zamiast z dowolnym zbiorem z liniowym porządkiem, ale w przeciwieństwie do sortowania nie stanowi żadnego problemu.

Drzewa sufiksowe umożliwiają nie tylko szybkie obliczanie zapytań o przynależność, ale mogą również służyć do rozwiązywania kilku innych problemów.

Problem 3.8. Pokaż, jak w czasie $O(m)$ odpowiadać na zapytania, ile razy słowo w ($|w| = m$) występuje w tekście t, gdy mamy skonstruowane $ST(t)$.

Dowód. Mając zbudowane drzewo sufiksowe, idziemy od korzenia w dół czytając kolejne symbole w, a czasami posuwamy się po wewnętrznej etykiecie pewnej krawędzi. Wówczas zbiór wystąpień odpowiada zbiorowi liści w poddrzewie węzła, do którego udało nam się w ten sposób dojść. Liczbę takich liści możemy ustalić bardzo łatwo: budując drzewo sufiksowe możemy od razu obliczyć w każdym węźle trzymać ile liści jest w jego poddrzewie. Jeżeli po drodze nie dało się dalej schodzić po drzewie, oznacza to, że w nie jest podsłowem słowa t, więc odpowiedzią jest zero. □

Problem 3.9. Pokaż, jak w czasie $O(n)$ policzyć, ile różnych podsłów zawiera słowo t ($|t| = n$).

Dowód. Gdybyśmy zbudowali takie pełne drzewo sufiksowe bez kompresji, to wówczas jest to po prostu drzewo trie wszystkich sufiksów danego słowa. Wówczas liczba różnych podsłów słowa n jest po prostu równa liczbie wierzchołków w stworzonym drzewie. W przypadku, gdy budujemy drzewo sufiksowe z kompresją, aby nie mieć kwadratowego rozmiaru całej struktury, należy również policzyć wierzchołki w zbudowanym drzewie, ale dodatkowo licząc wierzchołki liczymy je z odpowiednimi wagami, które wynikają z tego, że wierzchołki w skompresowanym drzewie sufiksowym są tak naprawdę ścieżkami wierzchołków i je też należy uwzględnić, co możemy zrobić bardzo łatwo, poprzez popatrzenie jakie etykiety są trzymane w wierzchołkach i dodając odpowiednie krotności. Możemy to wszystko zrobić jednym przejściem po drzewie, skąd wynika liniowa złożoność względem długości słowa. □

Problem 3.10. Pokaż, jak w czasie $O(nk)$ wyznaczyć najdłuższe wspólne podsłowo dla zbioru słów $\{t_i : 1 \leq i \leq k\}$ długości co najwyżej n.

Dowód. Rozwiązanie jest oczywiście bardzo proste. Wystarczy zbudować drzewo sufiksowe, ale nie dla każdego słowa osobno, ale jedno wspólne. Robimy to w ten sposób, że najpierw budujemy drzewo sufiksowe dla pierwszego słowa. Następnie wstawiamy do niego sufiksy drugiego słowa, trzeciego itd aż do k-tego, z tą różnicą, że wstawiając słowa, w wierzchołkach pamiętamy sobie informacje jaki sufiks tędy przechodzi w dół, tzn. każdy wierzchołek ma informacje na temat tego, sufiksy którego słowa przez niego przechodziły. Wówczas odpowiedzią, czyli najdłuższym wspólnym podsłowem, będzie najgłębiej położony węzeł w naszym drzewie zbudowanym dla n słów, przez który przeszedł jakiś sufiks każdego słowa. Oczywiście można tego LCSa bardzo łatwo odtworzyć jak już mamy taki najgłębszy wierzchołek o tej własności. Wystarczy przejść się od korzenia do niego i zobaczyć jakie etykiety mijaliśmy po drodze. □

3.1.4 Algorytm Weinera

Algorytm Weinera, historycznie najwcześniejszy algorytm optymalny, polega na budowaniu drzewa sufiksowego od prawej do lewej. Zauważmy, że dzięki przeglądaniu tekstu w tej kolejności gwarantujemy, że w i-tym kroku budowana struktura danych jest pełnoprawnym drzewem sufiksowym dla słowa $t[(n - i - 1)..n]$.

Tak jak w algorytmie McCreighta, najważniejszą częścią algorytmu jest efektywne wyznaczanie linków między wierzchołkami drzewa – z tą różnicą, że w tym przypadku są to linki odwrotne do linków sufiksowych. Definiujemy je następująco: jeśli $word(v) = w$ ($w \in \mathcal{A}^*$) oraz $t[i] = a$, to $link_a[v] = u$, gdzie u jest pewnym wierzchołkiem, dla którego $word(u) = a\, word(v)$.

Niniejsza implementacja jest oparta na uproszczonej wersji algorytmu, zamieszczonej w Breslauer i Italiano (2013).

Algorytm 20: Algorytm Weinera budowania drzewa sufiksowego

```
def weiner(text, n):
  text = text + '$'
  root = trie.TrieNode("")
  link, head = { (root, ""): root }, root
  for i in range(n + 1, 0, -1):
    # niezmiennik: link[v][c] = u dla wewnętrznych u i v takich, że word(u) = c word(v)
    v, depth = head, n + 2
    while v != root and link.get((v, text[i])) is None:
      v, depth = v.parent, depth - len(v.label)
    u = link.get((v, text[i]))
    if u is None or text[depth] in u.children:
      if u is None:
        u, remaining = slow_find(root, text[depth - 1:])
      else:
        u, remaining = slow_find(u, text[depth:])
      v, _ = fast_find(v, text[depth:-remaining], False)
      depth = len(text) - remaining
      if u != root:
        link[(v, text[i])] = u
    leaf = trie.TrieNode(text[depth:])
    u.add_child(leaf)
    head = leaf
  return root, link
```

Problem 3.11. Pokaż, że Algorytm 20 wylicza w każdej iteracji poprawne przejścia $link[v, c] = u$.

Problem 3.12. Pokaż, że Algorytm 20 zwraca poprawne drzewo sufiksowe.

Problem 3.13. Pokaż, że Algorytm 20 działa w czasie $O(n \log |\mathcal{A}|)$.

3.2 Tablica sufiksowa

Tablica sufiksowa jest dużo prostszą strukturą, wprowadzoną jako alternatywa dla drzew sufiksowych równolegle w (Gonnet, Baeza-Yates i Snider, 1992) i (Manber i Myers, 1993). Wartości tablicy dla słowa t długości n definiujemy następująco:

$$SA[i] = |\{j : t[j..n] \leq t[i..n]\}|,$$

czyli na i-tej pozycji mamy początkowy indeks i-tego najmniejszego sufiksu w porządku leksykograficznym.

Naiwny algorytm liczenia tablicy SA wymaga czasu $O(n^2 \log n)$, gdy wykorzystujemy sortowanie wymagające $O(n \log n)$ porównań, ponieważ każde porównanie wymaga w najgorszym przypadku czasu $O(n)$.

Można również obliczyć tablicę SA na podstawie zadanego drzewa sufiksowego w czasie $O(n)$: wystarczy przejrzeć drzewo wgłąb lub rekurencyjnie zgodnie z kolejnością liter w alfabecie. Zwróćmy uwagę, że Algorytm 21 wyznacza długości słów na podstawie znajomości tekstu i aktualnej głębokości przeszukiwania.

Algorytm 21: Algorytm budowania tablicy sufiksowej na podstawie drzewa sufiksowego

```
def suffix_array_from_suffix_tree(text, n):
  def traverse(v, depth):
    depth -= len(v.label)
    if len(v.children) == 0:
      return [depth]
    return [d for _, child in sorted(v.children.items())
            for d in traverse(child, depth)]
  ST, _ = suffix_tree.mccreight(text, n)
  return traverse(ST, n + 2)
```

3.2.1 Algorytm Karpa-Millera-Rosenberga

Algorytm *prefix-doubling*, zaproponowany w (Karp, Miller i Rosenberg, 1972), opiera się na pomyśle prostej rekurencji. Załóżmy, że dla podsłów $t[i : i + k]$[1] dla pewnego $k \geq 1$ oraz $1 \leq i \leq n$ mamy dostęp do funkcji $rank(i, k)$, zwracającej numer $t[i : i+k]$ w porządku leksykograficznym. Wówczas wystarczy umieć wyznaczać na jej podstawie efektywnie wartości $rank(i, 2k)$ dla wszystkich $1 \leq i \leq n$.

Algorytm 22: Algorytm Karpa-Millera-Rosenberga budowania tablicy sufiksowej

```
def reverse(SA):
  reverse = [0] * len(SA)
```

[1] Ściślej chodzi nam o $t[i : \min\{i + k, n\}]$ – albo można równoważnie założyć, że na końcu dodajemy nieskończony ciąg symboli \$.

```
  for i in range(len(SA)):
    reverse[SA[i] - 1] = i + 1
  return reverse

def prefix_doubling(text, n):
  '''Computes suffix array using Karp-Miller-Rosenberg algorithm'''
  text += '$'
  mapping = {v: i + 1 for i, v in enumerate(sorted(set(text[1:])))}
  R, k = [mapping[v] for v in text[1:]], 1
  while k < 2 * n:
    pairs = [(R[i], R[i + k] if i + k < len(R) else 0) for i in range(len(R))]
    mapping = {v: i + 1 for i, v in enumerate(sorted(set(pairs)))}
    R, k = [mapping[pair] for pair in pairs], 2 * k
  return reverse(R)
```

Lemat 3.6 (Crochemore, Hancart i Lecroq 2007, Lemma 4.8, s. 169-170)**.** *Dla dowolnych* $1 \leq i \leq n$ *oraz* $k \geq 1$ *wartość* $rank(i, 2k)$ *jest równa pozycji pary* $(rank(i, k), rank(i + k, k))$ *w porządku leksykograficznym na wszystkich takich parach.*

Dowód. Z założenia, $rank(i, k) < rank(j, k)$ wtedy i tylko wtedy, gdy $t[i : i + k] < t[j : j + k]$ oraz $rank(i, k) = rank(j, k)$ wtedy i tylko wtedy, gdy $t[i : i + k] = t[j : j + k]$ dla dowolnych i, j, k.

Jeśli $t[i : i + k] < t[j : j + k]$, to oczywiście zachodzi $t[i : i + 2k] < t[j : j + 2k]$ dla dowolnych i, j, k. Podobnie jeśli $t[i : i + k] = t[j : j + k]$ oraz $t[i + k + 1 : i + 2k] = t[j + k + 1 : j + 2k]$, to również $t[i : i + 2k] < t[j : j + 2k]$ dla dowolnych i, j, k.

Składając te wszystkie obserwacje razem, otrzymujemy wprost, że $t[i : i + 2k] < t[j : j + 2k]$ wtedy i tylko wtedy, gdy $rank(i, 2k) < rank(j, 2k)$ dla dowolnych i, j, k.

Aby zakończyć dowód wystarczy zaobserwować, że dla dwóch sąsiednich elementów w porządku $rank(i, 2k)$ nie może istnieć żaden element rozdzielający je w porządku $(rank(i, k), rank(i+k, k))$. □

Wniosek 3.7. Algorytm 22 zwraca poprawną tablicę sufiksową.

Dowód. Po $\log n$ iteracjach głównej pętli otrzymujemy z $rank(i, 1)$ tablicę wartości $rank(i, n)$ – a to z definicji jest dokładnie tablica sufiksowa. □

Twierdzenie 3.8. *Algorytm 22 wykonuje się w czasie* $O(n \log n)$.

Dowód. Całkowita złożoność algorytmu 22 zależy od zastosowanego sortowania. Jeśli jest to sortowanie pozycyjne (*radix sort*), to pojedyncza iteracja wymaga czasu $O(n)$. Iteracji mamy dokładnie $\log n$. □

3.2.2 Algorytm Kärkkäinena-Sandersa

Algorytm *skew*, wprowadzony w (Kärkkäinen i Sanders, 2003), umożliwia zejście do liniowego czasu obliczania tablicy sufiksowej. Wymaga on założenia o tym, że alfabet można utożsamić ze zbiorem liczb naturalnych. W przeciwieństwie jednak np. do problemu sortowania w algorytmach tekstowych jest to założenie bardzo naturalne i często spotykane w praktyce (podobnie jak założenie, że $|\mathcal{A}| = O(1)$).

Sednem algorytmu Kärkkäinena-Sandersa jest zamiana słowa t na słowo t' takie, że $|t'| = \frac{2}{3}|t|$. Zauważmy, że bez straty ogólności możemy założyć, że słowo t jest długości podzielnej przez 3 – wystarczy dołożyć na koniec odpowiednią liczbę znaków $\# \notin \mathcal{A}$ takich, że $\# > a$ dla każdego $a \in \mathcal{A}$.

Dokładniej, niech $triple(i)$ oznacza pozycję $t[i : i+3]$ wśród wszystkich unikalnych trzyliterowych podsłów t posortowanych leksykograficznie. Tworzymy nowe słowo t' w następujący sposób:

$$t' = triple(1) \cdot triple(4) \ldots \cdot triple(2) \cdot triple(5) \ldots$$

Zwrócony wynik dla tego słowa umożliwia odrębne posortowanie sufiksów t osobno dla zbiorów

$$P_{12} = \{1 \leq i \leq n : i \mod 3 \in \{1, 2\}\}$$
$$P_0 = \{1 \leq i \leq n : i \mod 3 = 0\}.$$

Co więcej, na podstawie danego t' możliwe jest scalenie również częściowych tablic sufiksowych dla P_{12} i P_0.

Algorytm 23: Algorytm Kärkkäinena-Sandersa budowania tablicy sufiksowej

```
def skew(text, n):
  '''Computes suffix array using Kärkkäinen-Sanders algorithm'''
  def convert(data):
    # zamiana tablicy liczb na string UTF-32 w tym samym porzadku znakow
    return '#' + ''.join(chr(ord('0') + v) for v in data)
  def compare(i, j):
    if i % 3 == 1:
      return (text[i], S.get(i + 1, 0)) >= (text[j], S.get(j + 1, 0))
    return (text[i:i + 2], S.get(i + 2, 0)) >= (text[j:j + 2], S.get(j + 2, 0))

  if n <= 4:
    return naive(text, n)
  text += '$'
  P12 = [i for i in range(1, n + 2) if i % 3 == 1] \
      + [i for i in range(1, n + 2) if i % 3 == 2]
  triples = _rank([text[i:i + 3] for i in P12])
  recursion = skew(convert(triples), (2 * n + 1) // 3 + 1)[1:]
  L12 = [P12[v - 1] for v in recursion]

  mapping = {v: i + 1 for i, v in enumerate(L12)}
```

```
S = {v: mapping[v] for v in P12}

P0 = [i for i in range(1, n + 2) if i % 3 == 0]
tuples = [(text[i], S.get(i + 1, 0)) for i in P0]
L0 = [P0[i - 1] for i in _reverse(_rank(tuples))]
return _merge(L12, L0, compare = compare)
```

Problem 3.14. Pokaż że 23 na podstawie t' wyznacza poprawnie tablicę sufiksów dla P_{12}.

Problem 3.15. Pokaż że 23 na podstawie t' wyznacza poprawnie tablicę sufiksów dla P_0.

Twierdzenie 3.9. *Algorytm 23 na podstawie t' poprawnie scala tablice sufiksów dla P_{12} i P_0.*

Dowód. Na mocy założenia mamy dostępne posortowane listy sufiksów osobno dla indeksów należących do P_{12} i P_0.

Scalenie odbywa się zgodnie z funkcją *merge* identyczną jak w algorytmie mergesort. Mamy 2 przypadki:

1. jeśli mamy $i \in P_{12}$ takie, że $i \mod 3 = 2$ oraz pewne $j \in P_0$, to wiemy, że $i+1, j+1 \in P_{12}$ – wystarczy zatem porównać ze sobą pary $(t[i], S[i+1])$ i $(t[j], S[j+1])$,
2. jeśli mamy $i \in P_{12}$ takie, że $i \mod 3 = 1$ oraz pewne $j \in P_0$, to wiemy, że $i+2, j+2 \in P_{12}$ – wówczas postępujemy podobnie, tylko porównujemy trójki $(t[i], t[i+1], S[i+2])$ i $(t[j], t[j+1], S[j+2])$.

Zauważmy, że $S[i+1], S[i+2]$ lub $t[i+1]$ mogą nie istnieć (podobnie dla j). Wówczas wystarczy zwracać wartości, które będą zawsze mniejsze od wartości odpowiednio z S lub $\mathcal{A}$. □

Twierdzenie 3.10. *Czas działania algorytmu 23 wynosi $O(n)$.*

3.2.3 Wykonywanie zapytań w czasie $O(m \log n)$

Sprawdzanie, czy słowo w o długości m występuje w tekście t dla danej tablicy sufiksowej $SA(t)$ jest wykonalne w czasie $O(m \log n)$: wystarczy zwykłe binarne wyszukiwanie słowa w w $SA(t)$.

Co więcej, tablica sufiksowa umożliwia nie tylko zliczenie wszystkich wystąpień słowa w w tekście w czasie $O(m \log n)$, ale również odnalezienie ich w czasie $O(m \log n + k)$. Wystarczy tylko wyszukać pozycje odpowiadające słowom w oraz $w\#$, gdzie $\# \notin \mathcal{A}$ oraz $a < \#$ dla każdego $a \in \mathcal{A}$. Pozycje te wyznaczają w tablicy $SA(t)$ sekwencję początków sufiksów, dla których w musi być prefiksem – chociaż niekoniecznie w kolejności rosnącej.

3.2.4 Wykonywanie zapytań w czasie $O(m + \log n)$

Algorytm Manbera-Myersa

3.2.5 Dalsze zagadnienia

Li, Li i Huo (2018) pokazali algorytm obliczania tablicy sufiksowej w czasie $O(n)$ wymagający zaledwie $O(1)$ dodatkowej pamięci.

3.3 Tablica największych wspólnych prefiksów

Konstrukcja tablicy LCP.

Twierdzenie 3.11. *Algorytm Kasai zwraca poprawną tablicę sufiksową.*

Dowód. Weźmy dwa dowolne sufiksy słowa t sąsiednie w porządku leksykograficznym tj. zaczynające się na pozycjach $SA[i]$ oraz $SA[i+1]$ dla pewnego $1 \leq i \leq n-1$. Niech długość najdłuższego wspólnego prefiksu $LCP(t[SA[i] : n], t[SA[i+1] : n]) = k$ dla pewnego $k \geq 0$.

Weźmy teraz j takie, że $SA[j] = SA[i]+1$, tj. $t[SA[j] : n] = t[SA[i]+1 : n]$. Wówczas wiemy, że

$$LCP(t[SA[j] : n], t[SA[j+1] : n]) \geq \max\{LCP(t[SA[i] : n], t[SA[i+1] : n]) - 1, 0\},$$

ponieważ $t[SA[j] : n] < t[SA[j+1] : n] \leq t[SA[i+1]+1 : n]$ oraz

$$\begin{aligned} LCP(t[SA[j] : n], t[SA[i+1]+1 : n]) &= LCP(t[SA[i]+1 : n], t[SA[i+1]+1 : n]) \\ &= \max\{LCP(t[SA[i] : n], t[SA[i+1] : n]) - 1, 0\}. \end{aligned}$$

Wykorzystujemy tu prosty fakt, że jeśli mamy dowolne słowa $w_1 \leq w_2 \leq w_3$, to $LCP(w_1, w_2) \geq LCP(w_1, w_3)$.

Jeśli nie istnieje szukane j, to $SA[i] = n$, a zatem rzecz jasna $t[SA[i]+1 : n] = \varepsilon$, a zatem jego najdłuższy wspólny prefiks z dowolnym innym podsłowem wynosi $k = 0$. □

Twierdzenie 3.12. *Liczba różnych podsłów dla słowa t ($|t| = n$) wynosi*

$$\binom{n}{2} - \sum_{i=1}^{n-1} LCP[i]$$

Dowód. Wystarczy policzyć podsłowa zaczynające się na pozycjach $SA[i+1]$ dla $0 \leq i \leq n-1$ tj. prefiksy $t[SA[i+1] : n]$ niebędące prefiksami słów $t[SA[j] : n]$ dla $1 \leq j \leq i$.

Ponieważ sufiksy są posortowane, to wiemy, że każdy prefiks $t[SA[i+1] : n]$ długości $0 \leq k < LCP[i]$ już wystąpił wcześniej – w szczególności jako prefiks $t[SA[i] : n]$.

Dalej, wiemy że $t[SA[i] : n] < t[SA[i+1] : SA[i+1]+LCP[i]+1]$, ponieważ słowa są posortowane oraz $t[SA[i+1] + LCP[i] + 1]$ jest z definicji LCP pierwszym znakiem różnym w słowach $t[SA[i] : n]$ i $t[SA[i+1] : n]$. Oczywiście z przechodniości relacji $<$ wynika, że również

$$t[SA[j] : n] \leq t[SA[i] : n] < t[SA[i+1] : SA[i+1] + LCP[i] + k]$$

dla wszystkich $k \geq 1$ oraz $1 \leq j \leq i$, a zatem każdy prefiks $t[SA[i+1] : n]$ długości większej niż $LCP[i]$ na pewno nie jest prefiksem $t[SA[j] : n]$ dla $1 \leq j \leq i$.

$t[SA[i+1] : n]$ ma łącznie $n - SA[i+1]$ prefiksów, więc takich prefiksów jest oczywiście $n - SA[i+1] - LCP[i]$. Dla $i = 0$ oczywiście będzie to tylko $n - SA[1]$. Ostatecznie, całkowita liczba różnych słów wynosi

$$\sum_{i=0}^{n-1} (n - SA[i+1]) - \sum_{i=1}^{n-1} LCP[i-1] = \binom{n}{2} - \sum_{i=1}^{n-1} LCP[i],$$

ponieważ $\sum_{i=0}^{n-1} (n - SA[i+1])$ to zarazem liczba wszystkich (niekoniecznie różnych) podsłów w tekście. □

3.4 Suffix trays

Uzupełnić – ale z niskim priorytetem

4 Największy sufiks

Problem 4: Największy leksykograficznie sufiks

Wejście: Słowo $w \in \mathcal{A}^+$ ($|w| = m$)
Wyjście: Liczba $1 \leq i \leq m$ taka, że $w[i..m]$ jest największym leksykograficznie sufiksem w.

4.1 Algorytm na bazie tablicy prefikso-sufiksów

Lemat 4.1. *Jeśli dla dowolnego słowa w słowo u jest maksymalnym sufiksem $w[1..(j-1)]$ a słowo v jest takim sufiksem słowa $w[1..(j-1)]$, że $uw[j] < vw[j]$, to v jest prefikso-sufiksem u.*

Dowód. Skoro u jest maksymalnym sufiksem $w[1..(j-1)]$, to $u > v$. Jeśli v nie było prefiksem u, to istniałaby pozycja $1 \leq i \leq |v|$ taka, że $v[i] < u[i]$. Ale wówczas $uy \geq u > vy$ dla dowolnego $y \in \mathcal{A}^*$ – sprzeczność z tym, że $uw[j] < vw[j]$. □

Lemat 4.2. *Jeśli dla dowolnego słowa w słowo u jest maksymalnym sufiksem $w[1..(j-1)]$ oraz dla pewnego prefikso-sufiksu v słowa u zachodzi $uw[j] < vw[j]$, to dla wszystkich słów v' takich, że v' jest prefikso-sufiksem u i v jest prefikso-sufiksem v' zachodzi $uw[j] < v'w[j] < vw[j]$.*

Dowód. Dla dowolnego v' będącego prefikso-sufiksem u istnieje słowo x takie, że $u = v'x$. Ponieważ u jest maksymalnym sufiksem $w[1..(j-1)]$, to $u = v'x > vx$, ponieważ vx też jest sufiksem $w[1..(j-1)]$. Wówczas zachodzi $vx < u < uw[j] < vw[j]$, a zatem $x < w[j]$. Wobec tego $uw[j] = v'xw[j] < v'w[j]$.

Wiemy, że $vw[j] > uw[j]$. Skoro $|u| > |v|$, to $vw[j] > u[1..(|v|+1)]y$ dla dowolnego $y \in \mathcal{A}^*$. Łatwo sprawdzić, że każde v' takie, że v' jest prefikso-sufiksem u i v jest prefikso-sufiksem v jest postaci $u[1..(|v|+1)]y$. □

Wniosek 4.3. Jeśli dla dowolnego słowa w słowo u jest maksymalnym sufiksem $w[1..(j-1)]$ a słowo $vw[j]$ jest maksymalnym sufiksem słowa $w[1..j]$, to wystarczy schodzić po prefikso-sufiksach u dopóki to możliwe.

Algorytm 24: Wyszukiwanie największego leksykograficznie sufiksu

```
def from_prefix_suffix(text, n, less = operator.__lt__):
  def equal(a, b):
    return not less(a, b) and not less(b, a)
  B, t, out = [-1] + [0] * n, -1, 1
  for i in range(1, n + 1):
    # niezmiennik: out = maximum_suffix(text[0..i - 1]))
    # niezmiennik: B[0..i - out] = prefix_suffix(text[out..i - 1])
    # niezmiennik: t = B[i - out]
    while t >= 0 and less(text[out + t], text[i]):
      out, t = i - t, B[t]
    while t >= 0 and not equal(text[out + t], text[i]):
      t = B[t]
    t = t + 1
    B[i - out + 1] = t
  return out, (n + 1 - out) - B[n + 1 - out]
```

Problem 4.1. Pokaż, że $B[j]$ to taka liczba, że $w[1..B[j]]$ jest największym prefikso-sufiksem maksymalnego sufiksu $w[1..j]$.

4.2 Algorytm o stałej pamięci

Algorytm 25: Wyszukiwanie największego leksykograficznie sufiksu w pamięci $O(1)$

```
def constant_space(text, n, less = operator.__lt__):
  def equal(a, b):
    return not less(a, b) and not less(b, a)
  out, p, i = 1, 1, 2
  while i <= n:
    r = (i - out) % p
    if equal(text[i], text[out + r]):
      i = i + 1
    elif less(text[i], text[out + r]):
      i, p = i + 1, i + 1 - out
```

```
    else:
      out, i, p = i - r, i - r + 1, 1
  return out, p
```

Problem 4.2. Pokaż, że algorytm 25 wyznacza poprawnie największy leksykograficznie sufiks.

5 Odległość edycyjna

Problem 5: Odległość edycyjna

Wejście: Słowa $t_1, t_2 \in \mathcal{A}^+$ ($|t_i| = n_i$ dla $i = 1, 2$)
Wyjście: Minimalna liczba operacji typu *insert*, *delete* i *substitute* przeprowadzająca słowo t_1 w t_2.

Zwróćmy uwagę, że analogiczny problem, w którym dozwolona jest tylko operacja *substitute* to obliczanie odległości Hamminga. Można ją wprost wyznaczyć przeglądając t_1 i t_2 od lewej do prawej, zliczając pozycje, dla których $t_1[i] \neq t_2[i]$.

Podstawowy algorytm obliczania odległości edycyjnej sprowadza się do sekwencyjnego obliczania wartości funkcji rekurencyjnej

$$d_e(i,j) = \begin{cases} 0 & \text{gdy } i = 0 \text{ lub } j = 0, \\ d_e(i-1, j-1) & \text{gdy } t_1[i] = t_2[j], \\ \min\{d_e(i-1,j-1), d_e(i-1,j), d_e(i-1,j-1)\} + 1 & \text{gdy } t_1[i] \neq t_2[j]. \end{cases}$$

Poprawność algorytmu wynika z indukcji ze względu na porządek par (i, j). Czas działania algorytmu to $O(n_1 n_2)$. Jak widać z powyższego wzoru, do obliczeń wystarczy zapamiętanie poprzedniego wiersza lub kolumny, więc wymagana pamięć wynosi $O(\min\{n_1, n_2\})$.

Algorytm 26: Obliczanie odległości edycyjnej w czasie $O(n_1 n_2)$ i pamięci $O(\min\{n_1, n_2\})$

```
def edit_distance(text_1, text_2, n_1, n_2):
  if n_1 < n_2:
    return edit_distance(text_2, text_1, n_2, n_1)
  previous_row, current_row = None, range(n_2 + 1)
  for i, ci in enumerate(text_1[1:]):
    previous_row, current_row = current_row, [i + 1] + [None] * n_2
    for j, cj in enumerate(text_2[1:]):
      insertion = previous_row[j + 1] + 1
      deletion = current_row[j] + 1
      substitution = previous_row[j] + (ci != cj)
      current_row[j + 1] = min(insertion, deletion, substitution)
  return current_row[-1]
```

Technika może być łatwo uogólniona do dowolnych macierzy ocen, przypisujących różne wagi za odległości dla operacji *insert*, *delete* i *substitute*. Przykładem takiego wykorzystania są rodziny macierzy PAM i BLOSUM dla algorytmów dopasowań ciągów w zastosowaniach bioinformatycznych.

Można również łatwo zamienić problem na obliczanie funkcji podobieństwa jako maksymalnej (zamiast minimalnej) wartości dla danej macierzy operacji, tym razem interpretowanej jako macierz nagród (Sellers, 1974).

5.1 Technika czterech Rosjan

Dla obliczeń macierzowych istnieje ogólna technika przyspieszania obliczeń, nazywana *Four Russians speedup*[2].

Główna idea tej metody polega na podziale wejścia na małe bloki o niewielkim zakresie wartości. Załóżmy, że pewien problem jest w ogólności rozwiązywany przez funkcję $y_n = f(y_{n-1}, x_n)$, a jedna iteracja wymaga czasu $F(n)$ – więc całość wykonuje się w czasie $\sum_{i=1}^{n-1} F(i)$.

Załóżmy teraz, że każda pozycja wejściowa pochodzi z małego zbioru X, a każda pozycja wyjściowa również z małego zbioru Y. Wówczas dla dowolnego k zawsze możemy obliczyć z wyprzedzeniem tablicę wszystkich $|X|^k|Y|$ wartości funkcji $f^{(k)}$, czyli k-krotnego złożenia f. Wówczas można policzyć f_n wykorzystując $\frac{n}{k}$ wyszukiwania w tablicy. Całkowity czas działania (zakładając stały dostęp do tablicy wartości) wynosi zatem

$$|X|^k|Y| \sum_{i=1}^{k-1} F(i) + \frac{n}{k}.$$

Przykładowo dla $F(n) = n^2$, $|X| = O(1)$ i $|Y| = O(n)$ wystarczy dobrać $k = \log_{|X|} n$, aby zmniejszyć czas wykonania z $O(n^3)$ do $O(n^2 \log^3 n)$.

Metoda ta używana jest m.in. do mnożenia i odwracania macierzy binarnych, a także do efektywnego liczenia domknięcia przechodniego grafu. Można ją również wykorzystać do optymalizacji algorytmu obliczającego odległość edycyjną (Masek i Paterson, 1980). Technika ta również uogólnia się na różnorodne macierze nagród i kar.

Algorytm i dowód

6 Najdłuższy wspólny podciąg

Problem 6: Najdłuższy wspólny podciąg

Wejście: Zbiór słów $T = \{t_1, t_2, ..., t_k\} \subseteq \mathcal{A}^+$ takich, że $|t_i| \leq n$ dla $1 \leq i \leq k$.
Wyjście: Słowo t będące najdłuższym podciągiem wszystkich słów z T.

[2] Najbardziej znanym z autorów był Jefim Dinic, znany również z algorytmu dla znajdowania maksymalnego przepływu w grafach.

Problem najdłuższego wspólnego podciągu jest powiązany z problemem odległości edycyjnej. Dla $|T| = 2$ niech $d(t_1, t_2)$ oznacza odległość dla dopuszczalnych operacji *insert* i *delete*. Wówczas zachodzi

$$|t_1| + |t_2| = 2|t| + d(t_1, t_2),$$

wystarczy bowiem zauważyć, że t_1 w t przeprowadzają wszystkie operacje *delete* a t w t_2 – *insert*.

6.1 Algorytm Needlemana-Wunscha

Podstawowy algorytm obliczania najdłuższego wspólnego podciągu dla dwóch słów jest prostym rozszerzeniem algorytmu 26. Jedyna różnica polega na konieczności trzymania całej tablicy odległości. Odtwarzanie wyniku następuje na podstawie przejścia po ścieżce od ostatniej wartości tj. $d(t_1, t_2)$ do początku, odpowiadającemu $d(\epsilon, \epsilon)$. Wybieramy rzecz jasna tylko te przejścia, które spełniają relację min w równaniu rekurencyjnym.

Oczywiście można odtwarzać kolejne przejścia i uzgodnione litery na podstawie t_1, t_2 porównując odpowiednie wartości macierzy d – lub też można zapisywać odpowiednie informacje równolegle w trakcie wypełniania tablicy d.

Algorytm 27: Obliczanie najdłuższego wspólnego podsłowa w czasie i pamięci $O(n_1 n_2)$

```
def needleman_wunsch(text_1, text_2, n_1, n_2):
  Data = namedtuple('Data', ['distance', 'previous', 'letter'])
  d = { (0, 0): Data(0, None, '') }
  for i, ci in enumerate(text_1[1:]):
    d[(i + 1, 0)] = Data(i + 1, (i, 0), ci)
  for i, ci in enumerate(text_2[1:]):
    d[(0, i + 1)] = Data(i + 1, (0, i), ci)
  for i, ci in enumerate(text_1[1:]):
    for j, cj in enumerate(text_2[1:]):
      insertion = Data(d[(i, j + 1)].distance + 1, (i, j + 1), '')
      deletion = Data(d[(i + 1, j)].distance + 1, (i + 1, j), '')
      if ci == cj:
        agreement = Data(d[(i, j)].distance, (i, j), ci)
        if agreement.distance <= min(insertion.distance, deletion.distance):
          d[(i + 1, j + 1)] = agreement
          continue
      if insertion.distance <= deletion.distance:
        d[(i + 1, j + 1)] = insertion
      else:
        d[(i + 1, j + 1)] = deletion
  text, p = '', (n_1, n_2)
  while p != (0, 0):
    p_next = d[p].previous
```

```
    if p[0] - p_next[0] == 1 and p[1] - p_next[1] == 1:
      text = d[p].letter + text
    p = p_next
  return text, d[(n_1, n_2)].distance
```

6.2 Algorytm Hirschberga

Ponieważ dla dostatecznie dużych n_1 i n_2 w praktyce pamięć jest bardziej dotkliwą barierą niż czas, można zastanowić się nad usprawnieniem algorytmu, aby wykorzystywał tylko $O(\min\{n_1, n_2\})$ pamięci.

Okazuje się, że wystarczy zastosować podejście dziel-i-rządź. Z jednej strony jasne jest, że odpowiadająca najdłuższemu podciągowi optymalna ścieżka od punktu (0, 0) do (n_1, n_2) w macierzy d musi przejść przez pewien punkt (i, j) dla dowolnego $0 \leq i \leq n_1$. Możemy zatem wybrać i w połowie aktualnie rozpatrywanego przedziału.

Znalezienie j nie jest trudne. Okazuje się, że wystarczy do tego obserwacja z poniższego lematu:

Problem 6.1. Punkt (i, j) leży na (pewnej) optymalnej ścieżce w macierzy d wtedy i tylko wtedy, gdy

$$j = \arg\max\{0 \leq k \leq n_2 : d(t_1[1..i], t_2[1..k]) + d(t_1[i..n_2], t_2[k..n_2])\}.$$

Algorytm 28: Obliczanie najdłuższego wspólnego podsłowa w czasie $O(n_1 n_2)$ i pamięci $O(\min\{n_1, n_2\})$

```
def hirschberg(text_1, text_2, n_1, n_2):
  if n_1 < n_2:
    return hirschberg(text_2, text_1, n_2, n_1)
  if n_2 == 0:
    return '', n_1
  if n_2 == 1:
    return needleman_wunsch(text_1, text_2, n_1, n_2)
  split_1 = n_1 // 2
  distance_previous = distance.indel_distance_row(
      text_1[:split_1 + 1], text_2, split_1, n_2)
  distance_next = distance.indel_distance_row(
      text_1[0] + text_1[n_1:split_1:-1], text_2[0] + text_2[n_2:0:-1],
      n_1 - split_1, n_2)[::-1]
  distance_sum = [d_1 + d_2 for d_1, d_2 in zip(
      distance_previous, distance_next)]
  split_2 = distance_sum.index(min(distance_sum))
  out_previous = hirschberg(
      text_1[:split_1 + 1], text_2[:split_2 + 1], split_1, split_2)
  out_next = hirschberg(
      text_1[0] + text_1[split_1 + 1::], text_2[0] + text_2[split_2 + 1:],
```

```
      n_1 - split_1, n_2 - split_2)
  return out_previous[0] + out_next[0], out_previous[1] + out_next[1]
```

6.3 Inne wyniki

W ramach wyników negatywnych dla problemu odległości edycyjnej wiadomo, że nie istnieje żaden algorytm dla obliczania odległości edycyjnej o złożoności $\Theta(n^{2-\varepsilon})$ dla dowolnego $\varepsilon > 0$, o ile Strong Exponential Time Hypothesis jest prawdziwa (Backurs i Indyk, 2015). Twierdzenie to zachodzi nawet dla przypadku alfabetu binarnego (Bringmann i Künnemann, 2015).

7 Dopasowanie wzorca z wieloznacznikami

Wprowadźmy teraz symbol specjalny ? jako wieloznacznik lokalny (*don't care symbol*), pasujący do każdego pojedynczego symbolu z $\mathcal{A}$.

Definicja 7.1. Relacja $\simeq$ jest relacją ***dopasowania z wieloznacznikiem lokalnym*** tj. $u \simeq v$ wtedy i tylko wtedy, gdy $|u| = |v|$ oraz dla wszystkich $1 \leq i \leq |u|$ zachodzi $u[i] = v[i]$ lub $u[i] = ?$, lub też $v[i] = ?$.

Problem 7: Wyszukiwanie wszystkich wystąpień wzorca z wieloznacznikami lokalnymi w tekście

---

Wejście: Słowa $t, w \in (\mathcal{A} \cup \{?\})^+$ $(|t| = n, |w| = m)$
Wyjście: Zbiór liczb $S = \{1 \leq i \leq n : t[i..(i+m-1)] \simeq w\}$.

Istnieje bardzo pomysłowy algorytm obliczania wszystkich wystąpień wzorca z wieloznacznikami lokalnymi w tekście, oparty o szybką transformatę Fouriera. Jak wiadomo, FFT umożliwia obliczanie dla dwóch wektorów X i Y o długości odpowiednio n i m (zakładając, że $n \geq m$) w czasie $O(n \log n)$ operacji splotu tj. takiego Z, że dla wszystkich $0 \leq i \leq n - m$ zachodzi

$$Z[i] = \sum_{j=0}^{m-1} X[i+j]Y[m-1-j].$$

Zwróćmy uwagę, że – wyjątkowo – mamy w tym przypadku indeksowanie od 0.

Algorytm 29: Obliczanie wszystkie wystąpień wzorca z wieloznacznikami lokalnymi w tekście

```
def basic_fft(text, word, n, m):
  if n < m:
    return
  A = set(list(text[1:] + word[1:]))
  A.discard('?')
```

```
  mismatches = [0] * (n - m + 1)
  for a in A:
    text_binary = [int(c == a) for c in text[1:]]
    for b in A:
      if a == b:
        continue
      word_binary = [int(c == b) for c in reversed(word[1:])]
      mismatches_ab = scipy.signal.convolve(
        text_binary, word_binary, mode = 'valid', method = 'fft')
      mismatches = [x + y for x, y in zip(mismatches, mismatches_ab)]
  yield from (i + 1 for i, v in enumerate(mismatches) if v == 0)
```

Twierdzenie 7.2. *Algorytm 29 wyznacza poprawnie wszystkie wystąpień wzorca z wieloznacznikami lokalnymi w tekście.*

Dowód. Kluczowa obserwacja jest następująca: wybierzmy dwie różne litery $a, b \in \mathcal{A}$ i wyznaczmy odpowiednio dla wszystkich $0 \leq i \leq n-1$ oraz $0 \leq j \leq m-1$

$$X[i] = 1 \text{ gdy } t[i+1] = a, \text{w przeciwnym przypadku } X[i] = 0,$$
$$Y[j] = 1 \text{ gdy } w[m-j] = b, \text{w przeciwnym przypadku } Y[j] = 0.$$

Wówczas

$$Z[i] = \sum_{j=0}^{m-1} X[i+j]Y[m-1-j] = |\{0 \leq j \leq m-1 : t[i+j+1] = a \wedge w[j+1] = b\}|,$$

a zatem $Z[i]$ jest po prostu zliczeniem wszystkich niedopasowań między tekstem $t[i+1..i+m]$ a wzorcem w, takich że w tekście występuje a, a we wzorcu b. Powtarzając to dla wszystkich par liter w alfabecie możemy zliczyć wszystkie niedopasowania między $t[i+1..i+m]$ a w – a zatem dopasowania będą na na tych pozycjach, na których niedopasowań brak. □

Jak łatwo zauważyć, algorytm 29 działa w czasie $O(n \log n |\mathcal{A}|^2)$.

8 Przybliżone dopasowanie wzorca

9 Kompresja

10 Najkrótsze wspólne nadsłowo

Problem 8: Najkrótsze wspólne nadsłowo

Wejście: Zbiór słów $T = \{t_1, t_2, \ldots, t_k\} \subseteq \mathcal{A}^+$ takich, że $|t_i| \leq n$ dla $1 \leq i \leq k$.
Wyjście: Słowo t będące najkrótszym nadsłowem wszystkich słów z T.

Bez straty ogólności zakładamy, że dla żadnej pary słów t_i i t_j nie zachodzi sytuacja, że t_i jest podsłowem t_j. Gdyby taka sytuacja wystąpiła, to moglibyśmy usunąć t_i z T bez zmiany optymalnego rozwiązania.

W ogólności problem jest trudny, zarówno dla krótkich słów, jak i dla małych alfabetów.

Twierdzenie 10.1 (Gallant, Maier i Storer 1980)**.** *Niech* $T = \{t_1, t_2, ..., t_k\}$ *i* $|t_i| = 3$ *dla* $1 \leq i \leq k$. *Problem znalezienia najkrótszego nadsłowa dla T jest NP-trudny.*

Twierdzenie 10.2 (Gallant, Maier i Storer 1980)**.** *Niech* $T = \{t_1, t_2, ..., t_k\}$ *oraz* $|\mathcal{A}| = 2$. *Problem znalezienia najkrótszego nadsłowa dla T jest NP-trudny.*

Pokazano, że problem znalezienia najkrótszego nadsłowa należy do klasy złożoności MAX-SNP, a zatem istnieje stała $c > 1$, dla której nie istnieje algorytm c-przybliżony dla tego problemu. Vassilevska (2005) dowiodła, że $c \geq \frac{1217}{1216}$, Karpinski i Schmied (2013) poprawili to do $c \geq \frac{333}{332}$.

Problem 10.1. Niech $T = \{t_1, t_2, ..., t_k\}$ przy $|t_i| \leq 2$ dla $1 \leq i \leq k$. Pokaż, jak możliwe jest wyznaczenie najkrótszego nadsłowa t dla T w czasie wielomianowym.

10.1 Algorytm zachłanny

Kod i dowód złożoności

Niech $T = \{c(ab)^k, (ba)^k, (ab)^k c\}$, wówczas algorytm zachłanny najpierw złoży ze sobą dwa skrajne ciągi, a zatem $t_{APX} = (ba)^k c(ab)^k c$, $|t_{APX}| = 4k + 2$, podczas gdy $t_{OPT} = c(ab)^{k+1}c$, $|t_{OPT}| = 2k + 4$.

Została postawiona hipoteza, że algorytm zachłanny jest 2-przybliżony, ale dotychczas udało się tylko pokazać współczynnik aproksymacji równy 4 (Blum i in., 1994), a następnie poprawić współczynnik go do 3.5 (Kaplan i in., 2005).

Zamiast porównywać długości słowa optymalnego i zwróconego przez algorytm możemy też zmienić kryterium optymalizacji. Naturalnym alternatywnym kryterium wydaje się wielkość kompresji:

Problem 9: Wspólne nadsłowo o największej kompresji

Wejście: Zbiór słów $T = \{t_1, t_2, \ldots, t_k\} \subseteq \mathcal{A}^+$ takich, że $|t_i| \leq n$ dla $1 \leq i \leq k$.
Wyjście: Słowo t będące nadsłowem wszystkich słów z T i maksymalizujące $d = \sum_{i=1}^{k} |t_i| - |t|$.

Oczywiście zbiór rozwiązań optymalnych dla obu problemów jest identyczny.

Okazuje się, że przy tych założeniach można pokazać, że algorytm zachłanny jest 2-przybliżony tj. osiąga kompresję taką, że $2d_{APX} \geq d_{OPT}$ (Tarhio i Ukkonen, 1988).

10.2 Inne wyniki

Oprócz algorytmu zachłannego opracowywano inne algorytmy o współczynnikach aproksymacji kolejno $\frac{8}{3}$ (Armen i Stein, 1996), $\frac{5}{2}$ (Kaplan i in., 2005; Sweedyk, 2000), $\frac{57}{23}$ (Mucha, 2013) i – obecnie najlepszego –

$\frac{71}{30}$ (Paluch, 2014). Większość z tych rezultatów polega na odpowiednio sprytnej redukcji do odpowiednio efektywnego algorytmu dla problemu maksymalnej ścieżki komiwojażera w grafie asymetrycznym.

Dodatkowo, jeśli przyjmiemy, że $|t_i| = r \geq 3$, to istnieje algorytm $\frac{r^2+r-4}{4r-6}$-przybliżony (Golovnev, Kulikov i Mihajlin, 2013). W Braquelaire i in. (2018) ten wynik został nieznacznie poprawiony – pozostaje on nadal jednak lepszy niż algorytm ogólny tylko dla $2 \leq r \leq 7$.

Bibliografia

Apostolico, Alberto i Zvi Galil (1997). *Pattern Matching Algorithms*. Oxford University Press.

Armen, Chris i Clifford Stein (1996). „A 2 2/3-approximation algorithm for the shortest superstring problem". W: *Annual Symposium on Combinatorial Pattern Matching*. Springer, s. 87–101.

Backurs, Arturs i Piotr Indyk (2015). „Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)". W: *Proceedings of the 47th Annual ACM Symposium on Theory of Computing*, s. 51–58.

Berstel, Jean i Juhani Karhumäki (2003). „Combinatorics on Words – A Tutorial". W: *Bulletin of the EATCS* 79, s. 178–228.

Blum, Avrim, Tao Jiang, Ming Li, John Tromp i Mihalis Yannakakis (1994). „Linear approximation of shortest superstrings". W: *Journal of the ACM* 41.4, s. 630–647.

Braquelaire, Tristan, Marie Gasparoux, Mathieu Raffinot i Raluca Uricaru (2018). „On improving the approximation ratio of the r-shortest common superstring problem". W: *arXiv preprint arXiv:1805.00060*.

Breslauer, Dany (1996). „Saving comparisons in the Crochemore-Perrin string-matching algorithm". W: *Theoretical Computer Science* 158.1-2, s. 177–192.

Breslauer, Dany i Giuseppe Italiano (2013). „Near real-time suffix tree construction via the fringe marked ancestor problem". W: *Journal of Discrete Algorithms* 18, s. 32–48.

Bringmann, Karl i Marvin Künnemann (2015). „Quadratic conditional lower bounds for string problems and dynamic time warping". W: *Proceedings of 56th Annual Symposium on Foundations of Computer Science*, s. 79–97.

Chang, William i Eugene Lawler (1990). „Approximate string matching in sublinear expected time". W: *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, s. 116–124.

Clifford, Peter i Raphaël Clifford (2007). „Self-normalised distance with don't cares". W: *Annual Symposium on Combinatorial Pattern Matching*, s. 63–70.

Cole, Richard i Ramesh Hariharan (2002). „Approximate string matching: A simpler faster algorithm". W: *SIAM Journal on Computing* 31.6, s. 1761–1782.

Commentz-Walter, Beate (1979). „A string matching algorithm fast on the average". W: *International Colloquium on Automata, Languages, and Programming*, s. 118–132.

Crochemore, Maxime, Christophe Hancart i Thierry Lecroq (2007). *Algorithms on Strings*. Cambridge University Press.

Crochemore, Maxime i Dominique Perrin (1991). „Two-way string-matching". W: *Journal of the ACM* 38.3, s. 650–674.

Crochemore, Maxime i Wojciech Rytter (1994). *Text Algorithms*. Oxford University Press.

— (2002). *Jewels of Stringology*. World Scientific.

Farach, Martin (1997). „Optimal suffix tree construction with large alphabets". W: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, s. 137–143.
Gallant, John, David Maier i James Storer (1980). „On finding minimal length superstrings". W: *Journal of Computer and System Sciences* 20.1, s. 50–58.
Giegerich, Robert i Stefan Kurtz (1997). „From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction". W: *Algorithmica* 19.3, s. 331–353.
Golovnev, Alexander, Alexander Kulikov i Ivan Mihajlin (2013). „Approximating shortest superstring problem using de bruijn graphs". W: *Annual Symposium on Combinatorial Pattern Matching*, s. 120–129.
Gonnet, Gaston, Ricardo Baeza-Yates i Tim Snider (1992). „New Indices for Text: Pat Trees and Pat Arrays." W: *Information Retrieval: Data Structures & Algorithms* 66, s. 82.
Gusfield, Dan (1997). *Algorithms on Strings, Trees and Sequences*. Cambridge University Press.
Hirschberg, Daniel (1975). „A linear space algorithm for computing maximal common subsequences". W: *Communications of the ACM* 18.6, s. 341–343.
Jacquet, Philippe i Wojciech Szpankowski (2015). *Analytic Pattern Matching: from DNA to Twitter*. Cambridge University Press.
Kaplan, Haim, Moshe Lewenstein, Nira Shafrir i Maxim Sviridenko (2005). „Approximation algorithms for asymmetric TSP by decomposing directed regular multigraphs". W: *Journal of the ACM* 52.4, s. 602–626.
Kaplan, Haim i Nira Shafrir (2005). „The greedy algorithm for shortest superstrings". W: *Information Processing Letters* 93.1, s. 13–17.
Kärkkäinen, Juha i Peter Sanders (2003). „Simple linear work suffix array construction". W: *International Colloquium on Automata, Languages, and Programming*, s. 943–955.
Karp, Richard, Raymond Miller i Arnold Rosenberg (1972). „Rapid identification of repeated patterns in strings, trees and arrays". W: *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, s. 125–136.
Karpinski, Marek i Richard Schmied (2013). „Improved inapproximability results for the shortest superstring and related problems". W: *Proceedings of the Nineteenth Computing: The Australasian Theory Symposium-Volume 141*, s. 27–36.
Landau, Gad i Uzi Vishkin (1989). „Fast parallel and serial approximate string matching". W: *Journal of Algorithms* 10.2, s. 157–169.
Li, Zhize, Jian Li i Hongwei Huo (2018). „Optimal in-place suffix sorting". W: *International Symposium on String Processing and Information Retrieval*, s. 268–284.
Lothaire, Monsieur (2002). *Algebraic Combinatorics on Words*. Cambridge University Press.
— (2005). *Applied Combinatorics on Words*. Cambridge University Press.
Manber, Udi i Gene Myers (1993). „Suffix arrays: a new method for on-line string searches". W: *SIAM Journal on Computing* 22.5, s. 935–948.
Masek, William i Michael Paterson (1980). „A faster algorithm computing string edit distances". W: *Journal of Computer and System Sciences* 20.1, s. 18–31.
Mucha, Marcin (2013). „Lyndon words and short superstrings". W: *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, s. 958–972.

Navarro, Gonzalo i Mathieu Raffinot (2002). *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press.
Paluch, Katarzyna (2014). „Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring". W: *arXiv preprint arXiv:1401.3670*.
Raita, Timo (1992). „Tuning the Boyer-Moore-Horspool string searching algorithm". W: *Software: Practice and Experience* 22.10, s. 879–884.
Sellers, Peter (1974). „On the theory and computation of evolutionary distances". W: *SIAM Journal on Applied Mathematics* 26.4, s. 787–793.
Smith, Peter (1991). „Experiments with a very fast substring search algorithm". W: *Software: Practice and Experience* 21.10, s. 1065–1074.
— (1994). „On tuning the Boyer-Moore-Horspool string searching algorithm". W: *Software: Practice and Experience* 24.4, s. 435–436.
Stephen, Graham (1994). *String Searching Algorithms*. World Scientific.
Sweedyk, Elizabeth (2000). „2 1/2-Approximation Algorithm for Shortest Superstring". W: *SIAM Journal on Computing* 29.3, s. 954–986.
Szpankowski, Wojciech (2011). *Average Case Analysis of Algorithms on Sequences*. John Wiley & Sons.
Tarhio, Jorma i Esko Ukkonen (1988). „A greedy approximation algorithm for constructing shortest common superstrings". W: *Theoretical computer science* 57.1, s. 131–145.
Valiente, Gabriel (2009). *Combinatorial pattern matching algorithms in computational biology using Perl and R*. Chapman i Hall/CRC.
Vassilevska, Virginia (2005). „Explicit inapproximability bounds for the shortest superstring problem". W: *International Symposium on Mathematical Foundations of Computer Science*, s. 793–800.
Vazirani, Vijay (2005). *Algorytmy aproksymacyjne*. Wydawnictwa Naukowo-Techniczne.
Wyner, Aaron D i Jacob Ziv (1994). „The sliding-window Lempel-Ziv algorithm is asymptotically optimal". W: *Proceedings of the IEEE* 82.6, s. 872–877.
Ziv, Jacob i Abraham Lempel (1977). „A universal algorithm for sequential data compression". W: *IEEE Transactions on information theory* 23.3, s. 337–343.
— (1978). „Compression of individual sequences via variable-rate coding". W: *IEEE Transactions on Information Theory* 24.5, s. 530–536.