1 Алгоритъм на Lempel-Ziv'78

Представих алгоритъма Lempel-Ziv качествено грешно. Коректното му описание е следното.

1. Нека $T = t_1 t_2 \dots t_n$ е текст, който трябва да бъде компресиран.
2. Компресията представлява последователност от двойки (p_i, a_i), където p_i е число, а a_i – буква. Смисълът на p_i е, че това е индекс на инфикс в T_i, който е бил кодиран по-рано. Така с всяка двойка (p_i, a_i) свързваме конкретен инфикс $\kappa(i) = T[j..l+1]$ като $a_i = t_{l+1}$. Уславяме се $\kappa(0) = \varepsilon$.

   Алгоритъмът работи итеративно така:

   (а) Нека $T[1..m]$ вече е представен като $(p_1, a_1) \dots (p_k, a_k)$.

   (б) Нека ℓ максимално, така че $T[m+1..m+\ell] = \kappa(p_i, a_i)$ за някое $i \leq k$.

   (в) Полагаме $(p_{k+1}, a_{k+1}) = (i, t_{m+\ell+1})$

   (г) Съответно $\kappa(k+1) = \kappa(i)a_{k+1} (= T[m+1..m+\ell+1]$.

   (д) Така $(p_1, a_1) \dots (p_k, a_k), (p_{k+1}, a_{k+1})$ е компресираният вариант на $T[1..m+\ell+1]$.

Имплементация на компресирането: Така за имплентацията на този алгоритъм не са необходими сложни структури от данни, както се бях заблудил, и може да се използва trie за стринговете $\kappa(i)$. Също така, не е трудно да се види, че за всяко i всички непразни префикси на $\kappa(i)$ се срещат като $\kappa(j)$ с $j < i$.

По този начин, може да си мислим, че trie-ът $\mathcal{T}$ има за състояния $\{\kappa(i)\}_{i=0}^{k}$. Тогава когато на поредната стъпка трябва да намерим ℓ и представянето на $T[m+1..m+\ell+1]$, просто траверсираме $\mathcal{T}$ с думата $T[m+1..n]$ докато можем. Върхът, до който сме стигнали е точно $\kappa(i)$, а дължината на обработения префикс от $T[m+1..n]$ е ℓ. Така сме получаваме $(p_{k+1}, a_{k+1}) = (i, t_{m+\ell+1})$.

Остава да обновим trie-а $\mathcal{T}$. За целта, добавяме нов преход от $\kappa(i)$ с $a_{k+1} = t_{m+\ell+1}$ към новото състояние $\kappa(k+1)$.

Имплементация на декомпресирането: Декомпресирането на редицата $(p_1, a_1), (p_2, a_2), \dots, (p_N, a_N)$ може да се получи лесно от следната рекурсивно дефиниция на κ^{-1}. (тук $\kappa^{-1}(k)$ е думата, която съответства на (p_k, a_k))

$$\kappa^{-1}(k) = \begin{cases} a_k \text{ ако } p_k = 0 \\ \kappa^{-1}(p_k)a_k \text{ ако } p_k > 0. \end{cases}$$

Важното свойство на тази редица е че всички думи $\kappa(i)$ са различни, поради което N не може да бъде твърде голямо. Оценката обаче не е $N \leq \frac{n}{\log_2 n}$, ами:

Лема 1.1. *Ако текст $T = t_1 t_2 \dots t_n \in \{0,1\}^n$ е представен като конкатенация на N различни думи $T = w_1 w_2 \dots w_N$, то:*

$$N \leq \frac{n \log\log n}{(1-\varepsilon_n)\log n}$$

като $\lim_{n\to\infty} \varepsilon_n = 0$.

Доказателство: Ясно е, че различните думи от $\{0,1\}^i$ с $i \leq n$ са 2^i. Поради това сумата от дължините на всички различни думи с дължина не по-голяма от k е:

$$\begin{aligned}\Delta_k = \sum_{i=0}^{k} i2^i &= k\sum_{i=0}^{k} 2^i - \sum_{j=0}^{k-1}\sum_{i=0}^{j} 2^i \\ &= k(2^{k+1}-1) - \sum_{j=0}^{k-1}(2^{j+1}-1) \\ &= k(2^{k+1}-1) - 2^k + k = (k-2)2^{k+1}.\end{aligned}$$

Да допуснем сега, че $N = 2^{k+1}-1$ за някое k. Това е точно броят на думите с дължина не по-голяма от k. Тогава сумарната дължина на различните думи $w_1, w_2, \ldots, w_N$ ще е поне колкото сумарната дължина на думите с дължина не по-голяма от k. От друга страна, $n = \sum_{i=1}^{N} |w_i|$. Следователно:

$$n \geq (k-2)2^{k+1} = (\log_2 N - 3)(N+1).$$

Сега, ако $N > \frac{n \log\log n}{(1-\varepsilon_n)\log n}$, то $\log_2 N - 3 > \log n - \log\log n - 3 = \log n(1-\varepsilon_n)$, където $\varepsilon_n = \frac{\log\log n + 3}{\log n}$. Следователно бихме получили, че:

$$(\log_2 N - 3)(N+1) > n,$$

което не е вярно.

Общият случай, когато $2^{k+1}-1 \leq N < 2^{k+2}-1$ не изисква нови идеи, но е технически по-пиква. Може да се види в [CT06].

При горната модификация, в [CT06], глава 13.5, може да се види оптималността на компресията[1] на Lempel-Ziv, която при определени вероятностни предположения за генерирането на символите в постоянно нарастващ текст T, постига ниво:

$$\leq H_k(T) = \lim_{n\to\infty} \frac{1}{n} \sum_{s=1}^{n} -\log_2 p(T[s+k]|T[s..s+k-1])$$

бита средно на символ за всяко фиксирано k, където $p(\sigma|w)$ е вероятността символът σ да следва срещане на думата w.

2 Построяване на суфиксен масив. Ниски – високи суфикси

Задачата за построяването на суфиксен масив за текст $T = t_1 t_2 \ldots t_n$ се състои в това да се намери пермутация ϕ на $\{1, 2, \ldots, n\}$, за която:

$$S_{\phi(1)}(T) \prec_{lex} S_{\phi(2)}(T) \prec_{lex} \dot{\prec}_{lex} S_{\phi(n)}(T),$$

където $S_i(T) = t_i t_{i+1} \ldots t_n$ е суфиксът в T, започващ на позиция i. Когато текстът T се подразбира ще пишем просто S_i вместо $S_i(T)$. Уславяме се $S_{|T|+1}(T) = \varepsilon$.

Дефиниция 2.1. *Казваме, че суфикс S_i (или просто позицията i) е нисък (съответно висок), ако $S_i \prec_{lex} S_{i+1}$ (съответно $S_i \succ_{lex} S_{i+1}$).*

[1] Ако се абстрахираме от символите във всяка двойка, тоест се концентрираме върху битовете, необходими за представянето на числовата информация p_i.

Нека $T = t_1 \dots t_n$. От дефиницията веднага получаваме, че:

1. S_n е висок, защото $S_{n+1} = \varepsilon$.
2. за $i < n$, S_i е нисък точно когато $t_i < t_{i+1}$ или $t_i = t_{i+1}$ и S_{i+1} е нисък (което според дефиницията е същото като $S_{i+1} \prec_{lex} S_{i+2}$).

Горното наблюдение позволява за време $O(n)$ да намерим типовете (висок/нисък) на всички суфикси на T. Искаме, рекурсивно да сортираме суфиксите от единия тип (този който не се среща по-често) и след това да попълним суфиксния масив за T.

2.1 Рекурсивно свеждане на сортирането на един от двата типа високи/ниски суфикси с двойно по-къс текст

Нека:

$$H(T) = \{i \leq n \mid S_i \text{ е висок } \} \text{ и } L(T) = \{i \mid S_i \text{ е нисък } \}.$$

Ясно, че или $|H(T)| \leq n/2$ или $|L(T)| \leq n/2$. Идеята е да кодираме по-малкото от двете множества, да кажем $H(T)$, с текст с дължина $|H(T)|$, така че да запазим лексикографската наредба. За разлика, обаче, от алгоритъма с триплетите, който разгледахме, сегментите между началните позиции на два високи/ниски суфикса не е фиксирано и това води до проблеми. Най-големият и може би основен, е проблемът с префиксите. Следното твърдение дава представа как да преодолеем тази трудност:

Лема 2.1. *Ако $T[i..i+d] = T[j..j+d]$ и S_{i+d} е висок, а S_{j+d} е нисък, то $S_i \prec_{lex} S_j$.*

Доказателство: Ако $i + d = n$, то $S_i = T[i..i+d]$, което според даденото е префикс на S_j и твърдението следва. Оттук нататък предполагаме, че $i + d < n$. Тъй като S_{j+d} е нисък, а S_n – висок, то $j + d < n$. Следователно, t_{i+d+1} и t_{j+d+1} са добре дефинирани и от наблюдението за ниските и високите суфикси от по-горе, получаваме, че:

1. $t_{i+d} > t_{i+d+1}$ или $t_{i+d} = t_{i+d+1}$ и S_{i+d+1} е висок.
2. $t_{j+d} < t_{j+d+1}$ или $t_{j+d} = t_{j+d+1}$ и S_{j+d+1} е нисък.

Сега, ако поне в една от горните две точки имаме строго неравенство, то от $t_{i+d} = t_{j+d}$ получаваме, че $t_{i+d+1} < t_{j+d+1}$ и следователно S_i и S_j се различават за първи път на позиция $d + 1$, където S_i е по-малък. Оттук и желаното $S_i \prec_{lex} S_j$.

Алтернативата е едновременно: (i) $t_{i+d} = t_{i+d+1}$ и S_{i+d+1} е висок; (ii) $t_{j+d} = t_{j+d+1}$ и S_{j+d+1} е нисък. Но тогава $t_{i+d+1} = t_{j+d+1}$, което влече $T[i..i+d+1] = T[j..j+d+1]$ заедно с S_{i+d+1} е висок, а S_{j+d+1} е нисък.

Тези наблюдения позволяват да завършим доказателството с индукция по $n - (i + d)$. Базата я обсъдихме в началото. При индуктивната стъпка, настъпват двете точко от по-горе, като при строго неравенство в една от тях, заключението следва. В противен случай настъпва алтернативата, при която свеждаме твърдението към $n-(i+d+1)$, което е вярно по индукционната хипотеза.

Да разгледаме първо случая, когато $|H(T)| = k \leq n/2$ и нека елементите на $H(T)$ са:

$$h_1 < h_2 < \dots < h_k = n \text{ и нека } h_{k+1} = n.$$

Сортираме лексикографски $T[h_i..h_{i+1}]$, тоест пресмятаме функция $\tau : \{1, 2, \dots, k\} \to \{1, 2, \dots, k\}$ е функция, за която:

$$\tau(i) < \tau(i+1) \iff T[h_i..h_{i+1}] \prec_{lex} T[h_j..h_{j+1}].$$

По-долу ще обясним как това може да се направи за време $O(n)$. Засега да си мислим, че това е направено. Тогава построяваме думата:

$$\mathcal{T} = \tau(1)\tau(2)\dots\tau(k).$$

Ключовото наблюдение е, че:

Лема 2.2. *$S_{h_i}(\mathcal{T}) \prec_{lex} S_{h_j}(\mathcal{T})$ точно когато $S_i(\mathcal{T}) \prec_{lex} S_j(\mathcal{T})$.*

Доказателство: Индукция по сумата от $|S_i(\mathcal{T})| + |S_j(\mathcal{T})|$. Ако $\tau(i) = \tau(j)$, то свеждаме към индуктивното предположение, така че да предположим, че $\tau(i) \neq \tau(j)$ и без ограничение на общността нека $\tau(i) < \tau(j)$. От дефиницията на τ имаме, че:

$$T[h_i..h_{i+1}] \prec_{lex} T[h_j..h_{j+1}].$$

Сега, ако $T[h_i..h_{i+1}]$ и $T[h_j..h_{j+1}]$ се различават на някоя позиция и първата такава е d, т.е. $T[h_i..h_i + d - 1] = T[h_j..h_j + d - 1]$ и $t_{h_i+d} \neq t_{h_j+d}$, то очевидно $t_{h_i+d} < t_{h_j+d}$ и тогава също е ясно, че и $S_{h_i} \prec_{lex} S_{h_j}$.

Нетривиалният случай е когато, $T[h_i..h_{i+1}]$ е същински префикс на $T[h_j..h_{j+1}]$. Нека $d = h_{i+1} - h_i \geq 1$. Тогава $T[h_i..h_i + d] = T[h_j..h_j + d]$. Освен това, $S_{h_i+d} = S_{h_{i+1}}$ е висок, а тъй като $h_j + d < h_{j+1}$, то S_{h_j+d} е нисък. Тогава от предишната лема $S_{h_i} \prec_{lex} S_{h_j}$. Това завършва доказателството.

Случаят, когато $|L(T)| \leq |H(T)|$ е подобен, но изисква малко внимание при дефиницията на τ. Разликата е в това, че ако в случая на ниски, префиксите всъщност са *по-големи*. Това е единствената разлика. Ето и подробностите:

Лема 2.3. *Нека $\ell_1 < \ell_2 < \dots < \ell_k$ са ниските позиции $L(T)$, подредени по големина и нека $\ell_{k+1} = n$. Нека $\tau' : \{1, 2, \dots, k\} \to \{1, 2, \dots, k\}$ е функция, за която:*

$$\tau'(i) \leq \tau'(j) \iff \begin{array}{c} T[\ell_j..\ell_{j+1}] \prec_{pref} T[\ell_i..\ell_{i+1}] \text{ или} \\ T[\ell_j..\ell_{j+1}] \not\prec_{pref} T[\ell_i..\ell_{i+1}] \text{ и } T[\ell_i..\ell_{i+1}] \prec_{lex} T[\ell_j..\ell_{j+1}]. \end{array}$$

Тогава:

1. *$\tau'(i) \leq \tau'(j)$ точно когато $S_{\ell_i} \prec_{lex} S_{\ell_j}$.*
2. *ако $\mathcal{T}' = \tau'(1)\dots\tau'(k)$, то $S_i(\mathcal{T}) \prec_{lex} S_j(\mathcal{T})$ точно когато $S_{\ell_i}(T) \prec_{lex} S_{\ell_j}(T)$.*

Накрая ми се иска да коментирам ефективното намиране на функциите τ, съответно τ'. И в двата случая задачата се свежда до построяването на trie за съответните думи и обхождането на този trie в дълбочина като синовете на всеки връх се посещават в нарастващ лексикографски ред. Разликата между τ и τ' е, че при τ', първо разглеждаме, тоест номерираме, наследниците, а след това самия връх, ако той също е финален. При τ нещата са непосредствени върховете получават своите номера при първото им посещаване.

По-особено обаче е самото построяване на trie-а, защото азбуката не е константна. Поради рекурсивното извикване, тя може да стане $O(n)$. Поради това добавянето на думите една по една няма да е ефективно, защото проверката дали имаме преход с дадена буква от дадено състояние няма да може да осъществим за константно време. Поддържането на лексикографски ред също е трудно за време $O(1)$.

Възможно решение е следното. То протича на два паса. На първия пас построяваме trie-а в *широчина*, без да се грижим за наредбата на синовете. На втория пас използваме counting-sort за преходите като ги сортираме по символ. По този начин вторият пас ще сортира преходите на trie-а правилно за време $O(n)$.

Остана да обясним първия пас. Инвариантът, който се поддържа е следният. Към всеки връх на trie-а поддържаме множеството от стринговете (само техните индекси, разбира се), които имат за префикс съответния връх на trie-а. Това ни позволява да добавим различните преходи от този връх наведнъж и не се налага да проверяваме дали даден преход съществува или не. Нека v е този връх на trie-а и на него му съответстват $\{i_1, i_2, \ldots, i_k\}$, тоест $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ започват с префикс v и други такива (от тези, които ни интересуват) няма. Тъй като целта е само да поставим преходи с различните букви $t_{i_j+|v|}$, не се налага да ги сортираме, а само да разберем кои са. Поддържаме глобален масив Ind с размер n, който е индиксиран по азбуката. Също така поддържаме масив от n списъка L. В началото на обработката на върха v от trie-а всички елементи $Ind[\sigma] = -1$ и всички списъци $L[i] = \emptyset$. Имаме и брояч *count*, за броя на непразните списъци, в началото $count = 0$. Разглеждаме последователно множеството $i_1, \ldots, i_k$:

1. Когато разглеждаме i_j, намираме $\sigma = t_{i_j+|v|}$, ако действително ни интересува този символ[2].
2. Ако $Ind[\sigma] = -1$, тогава:
   - $Ind[\sigma] \leftarrow count$
   - $count \leftarrow count + 1$.
3. Нека $p = Ind[\sigma]$, добавяме i_j към $L[p]$ – списъка, който съответства на буквата σ.

По същество разбихме преходите на класове на еквивалентност. Сега минаваме по списъците от $L[0]$ до $L[count - 1]$. За всеки от тях създаваме нов преход и ново състояние. Надписваме прехода с правилната буква $\sigma = t_{i_j+|v|}$, където i_j е първият елемент на съответния списък. След това прехвърляме елементите от разглеждания списък към новосъздадения връх на trie-а. Накрая поправяме стойността $Ind[\sigma] = -1$. Накрая почистваме списъците, които сме използвали, като връщаме $count = 0$. По този начин, може да започнем работа по следващия връх от trie-а, например някой от синовете на v, ако има такива.

Очевидно, времето, необходимо в горния метод е пропорционално на сумата от дължините на стринговете, които обработваме, защото всеки такъв се разглежда за време $O(1)$ за всеки свой префикс. Тъй като ние разглеждаме не повече от $n/2 = |T|/2$ думи, които имат не повече от $n/2$ общи букви, то общата им дължина е най-много $3n/2$. Тоест времето за горната сортировка и намирането на подходящата функция τ (съответно τ') е $O(n)$.

2.2 Попълване на суфиксния масив

По-наблюдателните сигурно са забелязали, че и тук има проблем.

Случаите $H(T)$ и $L(T)$ са дуални, с един малък детайл при ниските. Основната идея обаче е една и съща. Тя стъпва на следното директно следствие от лемата 2.1 от предишния параграф:

Следствие 2.1. *Ако $t_i = t_j$ и S_i е висок, а S_j – нисък, то $S_i \prec_{lex} S_j$.*

Доказателството се получава като положим $d = 0$ в цитираната лема.

Да допуснем, че буквите, които се срещат в текста T са числата $\{0, 1, \ldots, m\}$. Разбиваме суфиксния масив, ϕ, който искаме да построим, на блокове, по един за всяка буква. От горното наблюдение знаем, че във всеки блок ниските следват високите. Това означава, че ако сме сортирали високите, то може да попълним началните отрези на всеки от блоковете, а ако сме сортирали ниските, то може да попълним крайните отрези на всеки от блоковете.

[2] Нас не ни интересува целия суфикс S_{i_j}, а само негов префикс, виж по-горе

Ето как стоят нещата по-формално. Нека:

$$occ(\sigma) = |\{i \mid t_i = \sigma\}| \text{ и } C(\sigma) = 1 + \sum_{\sigma' < \sigma} occ(\sigma').$$

Тогава блокът за буквата σ е $\phi[C(\sigma)..C(\sigma+1)-1]$, където подразбираме, че $C(m+1) = n+1$.

Да разгледаме първо случая, когато $H(T)$ са сортирани. Тоест, нека $h_1 < h_2 < \cdots < h_k$ са елементите на $H(T)$ и:

$$S_{h_{\phi'(i)}} \prec_{lex} S_{h_{\phi'(j)}} \iff i < j,$$

където ϕ' е известна. От горното наблюдение, знаем, че всички високи предшестват всички ниски. Поради това:

1. полагаме $count[\sigma] = C(\sigma)$ за $\sigma \leq m$.
2. за $j = 1$ до k:
   (а) $i \leftarrow h_{\phi'(j)}$
   (б) $\sigma \leftarrow t_i$
   (в) $\phi[count[\sigma]] \leftarrow i$
   (г) $count[\sigma] \leftarrow count[\sigma] + 1$.

Сега разглеждаме масива ϕ отзад напред и попълваме всеки от блоковете с липсващите ниски също отзад напред. По-точно:

1. полагаме $count[\sigma] = C(\sigma + 1) - 1$ за $\sigma \leq m$, това са последните позиции в съответните блокове.
2. за $j = n$ в намаляващ ред до 1:
   (а) $i \leftarrow \phi[j]$
   (б) ако $i > 1$, $\sigma = t_{i-1}$
   (в) ако $i > 1$ и S_{i-1} е нисък:
      i. $\phi[count[\sigma]] \leftarrow i - 1$
      ii. $count[\sigma] \leftarrow count[\sigma] - 1$

Това е всичко, като процедура, в този случай. Очевидно е линеен този алгоритъм. Въпросът е защо е коректен. Най-подозителен изглежда последният блок. Хем започваме от края му, хем ниските в него са след високите. Е, работата е, че няма ниски, които да започват с най-голямата буква $\sigma = m$:

Лема 2.4. *Ако $t_i = m$ е лексикографски най-голямата буква, то S_i е висок.*

Доказателство: Разсъждаваме с индукция по $n - i$. При $i = n$, S_n е висок по дефиниция. Нека $i < n$. Тъй като $t_i = m$, то $t_i \geq t_{i+1}$. Ако $t_i > t_{i+1}$, то по дефиниция $S_i \succ_{lex} S_{i+1}$, откъдето S_i е висок. В противен случай $t_i = t_{i+1}$. Може да приложим индуктивното предположение за S_{i+1} и получаваме, че S_{i+1} е висок. Но тогава и S_i е висок.

Горната лема показва, че последният блок $\phi[C(m)..n]$ е попълнен, при това коректно. Сега с индукция по j ще покажем, че $\phi[j..n]$ е попълнен коректно:

Лема 2.5. *За всяко j, когато разглеждаме $\phi[j]$ в процедурата по-горе, $\phi[j]$ съдържа елемент i при това S_i е точно j-тият суфикс по азбучен/лексикографски ред.*

Доказателство: Разсъждаваме (с пълна математическа) индукция по $n - j$. Случаят, когато $j \in [C(m), n]$ е уреден от предишната лема. Нека $j < [C(m)..n]$. Нека i е индексът на j-тият по големина суфикс по азбучен ред и нека $\sigma = t_i$. Ако S_i е висок, то той е попълнен правилно точно на позиция j. Тогава $\phi[j] = i$ още след първата част на процедурата. Остава случаят, когато S_i е нисък. Тъй като S_i е нисък, а S_n – висок, то $i < n$. Следователно t_{i+1} и S_{i+1} са добре дефинирани. При това, тъй като S_i е нисък, то или $S_i \prec_{lex} S_{i+1}$. Следователно S_{i+1} има ред $j' > j$ в лексикографската наредба на суфиксите. От индуктивното предположение $\phi[j'] = i + 1$ на стъпка j'. Тогава в този момент, алгоритъмът ще попълни i в последната свободна клетка $c \in [C(\sigma), C(\sigma + 1) - 1]$ в блока за $\sigma = t_i$. Тоест $\phi[c] = i$. Сега, ако $c > j$, то очевидно $\phi[c]$ няма как да съдържа c-тия по големина суфикс и това е противоречие с индуктивното предположение. Следователно $c \leq j$. Да допуснем, че $c < j$. Тогава, клетката $\phi[j]$ е заета от друг суфикс p. Тъй като p е попълнен по-рано, да кажем при разглеждането на $p + 1 = \phi[j'']$, то $j' < j''$ и от индуктивното предположение $S_{i+1} \prec_{lex} S_{p+1}$. Тъй като $t_p = t_i = \sigma$, то $S_i \prec_{lex} S_p$. Но щом S_i е j-ти по ред, последното означава, че S_p има ред, да кажем r, строго по-голям от j ($r > j$). От индуктивното предположение за r, $\phi[r] = p$. Но тогава, p не може да е попълнен във $\phi[j]$, защото всеки индекс се попълва най-много веднъж в тази фаза. Това доказва, че $c = j$ и $\phi[j] = i$, както и искахме да докажем.

Накрая, да очертаем разликите, когато $L(T)$ са сортирани. Тоест, нека $\ell_1 < \ell_2 < \cdots < \ell_k$ са елементите на $L(T)$ и:

$$S_{\ell_{\phi'(i)}} \prec_{lex} S_{\ell_{\phi'(j)}} \iff i < j,$$

където ϕ' е известна. Отново намираме блоковете на всяка от буквите и този път попълваме ниските в края на съответния им блок. Накрая попълваме и n в първата клетка на блока за t_n, тоест $\phi[C[t_n]] = n$. Това е коректно, защото $S_n = t_n$ е най-малкият суфикс, който започва с буквата t_n – той е префикс на всеки друг такъв. Причината за тази асиметрия е, че S_n е (винаги) висок, а позицията n не е следвана от никоя друга. Тоест, позициите на ниските суфикси винаги се следват от позиция, докато за n е изключение при високите. С тази модификация, може да довършим алгоритъма дуално, тоест започваме от ляво надясно и попълваме високите в нарастващ ред. Ето и детайлите:

1. полагаме $count[\sigma] = C(\sigma + 1) - 1$ за $\sigma \leq m$.
2. за $j = k$ в намаляващ ред до 1:
   (а) $i \leftarrow \ell_{\phi'(j)}$
   (б) $\sigma \leftarrow t_i$
   (в) $\phi[count[\sigma]] \leftarrow i$
   (г) $count[\sigma] \leftarrow count[\sigma] - 1$.
3. $\phi[C(t_n)] \leftarrow n$.

Сега разглеждаме масива ϕ отпред назад и попълваме всеки от блоковете с липсващите ниски също отпред назад. По-точно:

1. полагаме $count[\sigma] = C(\sigma)$ за $\sigma \neq t_n$ и $count[t_n] = C(t_n) + 1$, първите незаети позиции в съответните блокове.
2. за $j = 1$ в нарастващ ред до n:
   (а) $i \leftarrow \phi[j]$

(б) ако $i > 1$, $\sigma = t_{i-1}$

(в) ако $i > 1$ и S_{i-1} е висок:

i. $\phi[count[\sigma]] \leftarrow i - 1$

ii. $count[\sigma] \leftarrow count[\sigma] + 1$

Сега твърдението, че блокът за 0-та буква изобщо казано не е вярно! То остава в сила, ако $t_n \neq 0$:

Лема 2.6. *Ако $t_i = 0 < t_n$ е лексикографски най-малката буква, то S_i е нисък.*

Доказателство: Отново индукция по $n - i$. Да разгледаме i, така че $t_i = 0$. Тъй като $0 < t_n$, то $i < n$. Сега, ако $t_{i+1} \neq 0$, то $0 < t_{i+1}$ и S_i е нисък. В противен случай $t_{i+1} = 0$ и от индуктивното предположение S_{i+1} е нисък. Следователно и S_i е нисък.

Лема 2.7. *За всяко j, когато разглеждаме $\phi[j]$ в процедурата по-горе, $\phi[j]$ съдържа елемент i при това S_i е точно j-тият суфикс по азбучен/лексикографски ред.*

Доказателство: Индукция по j. При $j = 1$ имаме, че или $0 < t_n$, и тогава целият първи блок е попълнен от горната лема и при това коректно. Или $t_n = 0$ и тогава отново първият елемент на блока е попълнен коректно от модификацията, която направихме. Индуктивната стъпка върви дуално на случая $H(T)$ с единствената особеност, когато $j = C[t_n] = n$. Тогава трябва да се позовем отново на модификацията, защото в този случай n няма следващ елемент.

Литература

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* John Wiley & Sons, 2006.