Lista trzecia z algorytmów tekstowych

Łukasz Klasinski

25 stycznia 2021

Zadanie 2

Design a linear-time algorithm for constructing the suffix tree given the suffix array and the LCP array. The children of each node should be stored in the sorted order (but you don't have to build any clever structure for extracting the edge starting from a given character).

Na wejściu dostajemy SA oraz LCP array. Konstrukcję drzewa rozpoczynamy od stworzenia roota, który reprezentuje drzewo suffiksowe pustego słowa. Algorytm będzie przechodzić przez kolejne suffiksy w SA i dodawać je do naszego drzewa. Schemat postępowania wygląda nastepująco:

1. Weź kolejny, i-ty suffix z SA.
2. Niech n_l to ostatni node z drzewa, na którym skończyliśmy po dodaniu poprzedniego suffiksa, a $\#L_{n_l}$ to długość labeli od node'a n do korzenia.
3. Wracamy w górę drzewa od n_l do roota, dopóki nie będzie spełniony warunek $\#L_{n'} \leq LA[i]$, gdzie n' to node'y na tej drodze. W szczególności możemy iść w górę do samego root'a (gdzie warunek zawsze się spełni, bo $\#L_{root} = 0$, a w LA mamy wartości $0..N$).
4. Kiedy dojdziemy do node'a n', który spełnia warunek z poprzedniego punktu, to mamy przypadki:
   - $\#L_{n'} = LA[i]$ - oznacza to, że suffiks, który teraz rozważamy ma wspólny prefix z poprzednim suffiksem wynoszący $\#L_{n'}$, oraz jakiś ciąg znaków, który się różni (od $\#L_{n'}$-tej litery). Zatem, dodajemy do drzewa suffiksowego nowy node m, który podpinamy do node'a n'. Nowo utworzoną krawędź labelujemy napisem, który reprezentuje SA w drzewie suffiksowym - $SA[i][\#L_{n'}..]$, natomiast w samym nodzie m zapiszemy indeks $\#SA[i]$.
   - $\#L_{n'} < LA[i]$ - oznacza to, że label najbardziej prawego syna danego node'a, ma wspólny suffix z $SA[i]$ wynoszący $\#L_{n'}$ oraz jakieś dodatkowe znaki. Oznaczmy tego syna przez k. Usuwamy krawędź w między nim a jego rodzicem n' i dodajemy nowy node k' w jego miejsce. Będzie on połączony z rodzicem krawędzią $SA[i][0..\#L_{n'}]$. Następnie podpinamy do node'a k' stary node k, używając fragmentu starej krawędzi w - $w[\#L_{n'}..]$. Pozostaje dodanie node'a, który reprezentuje wcześniej wspomniane "dodatkowe znaki". Dodajemy zatem nowe dziecko l do node'a k'. Łączymy je krawędzią $SA[i][\#L_{n'}..]$, a w samym nodzie l, zapisujemy indeks $SA[i]$

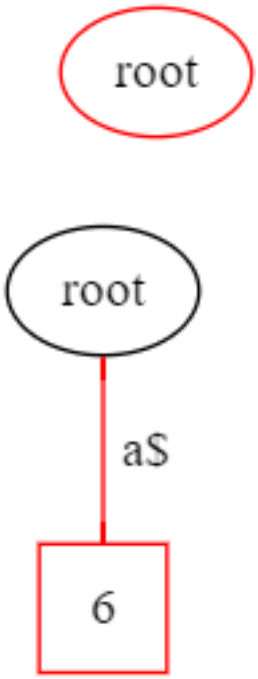
Dodatkowo mamy special case kiedy $\#L_{n'} = LA[i]$, ale node n', w którym się znajdujemy jest liściem. Wtedy stosujemy strategię $\#L_{n'} < LA[i]$.

Zauważmy, że w tym algorytmie zawsze dodajemy nowy node'a jako jakieś prawe dziecko (nowy node jest umieszany z prawej strony drzewa), a dzięki kolejności suffiksowej kolejne dzieci wszystkich node'ów będą posortowane leksykograficznie. Poza tym można zauważyć, że algorytm działa w czasie $O(n)$, ponieważ kiedy w kroku 3 idziemy w górę drzewa to podczas dodawania kolejnych suffiksów nigdy się już nie wrócimy do node'ów z których już kiedyś wychodziliśmy (ponieważ bądą to jakieś słowa z SA, które są wcześniej na liście i nie mają już wspólnego prefiksu), a zatem każdy node zostanie odwiedzony co najwyżej 2 razy. Otrzymujemy zatem złożoność $O(n)$.

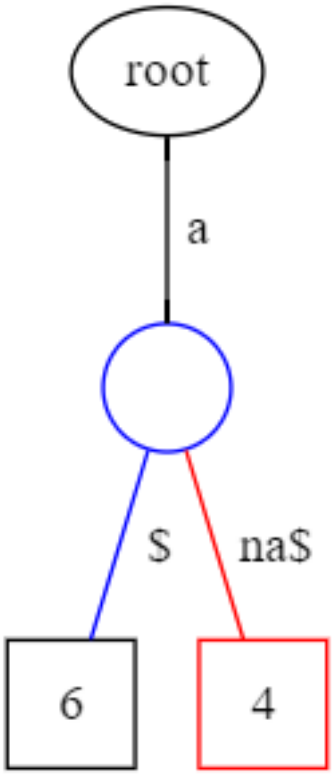
Poniżej przykładowa konstrukcja dla słowa banana\$

$SA \quad = [7, 6, 4, 2, 1, 5, 3]$ $LCP = [-1, 0, 1, 3, 0, 0, 2]$

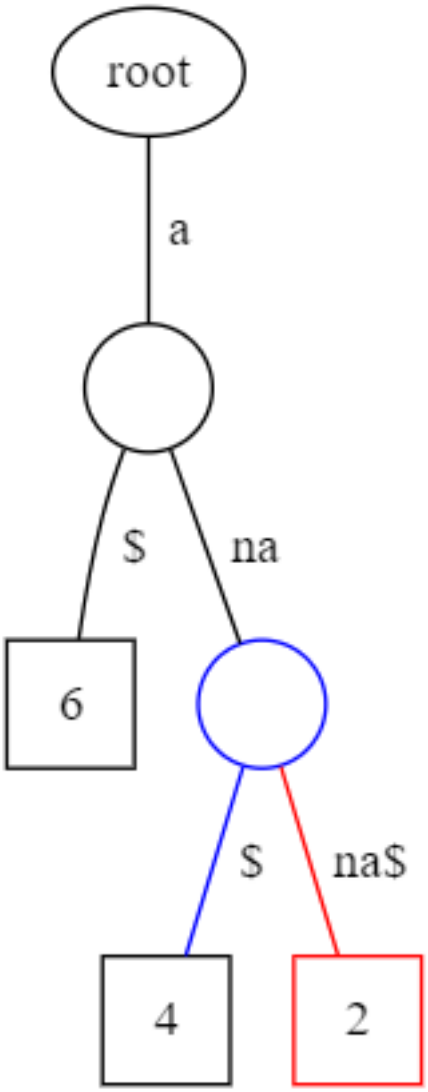
Na czerwono oznaczony jest ostatnio odwiedzony/dodany node, natomiast na niebiesko świeżo dodane krawędzie i node'y



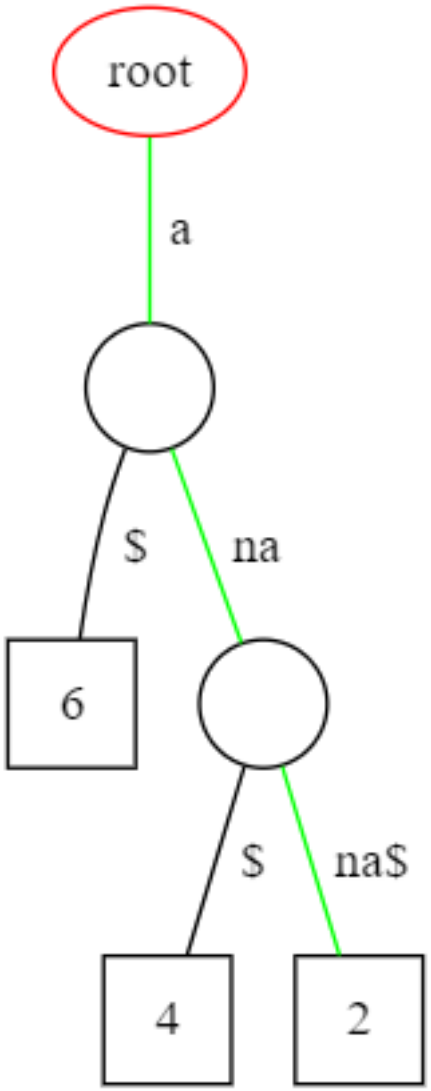
$\#L_{n'} = 0 \;\; LCP = 0 \;\; i = 0$



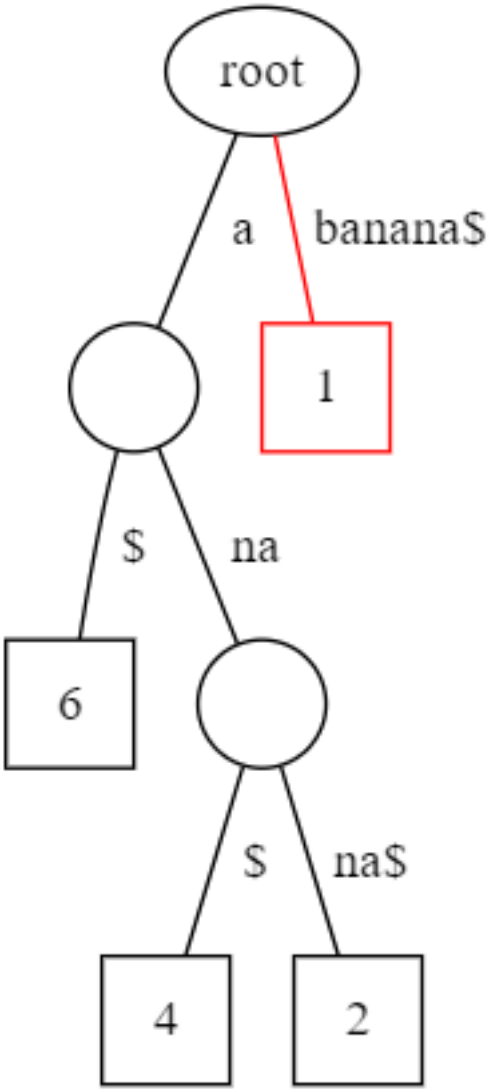
$\#L_{n'} = 1 \;\; LCP = 1 \;\; i = 1$



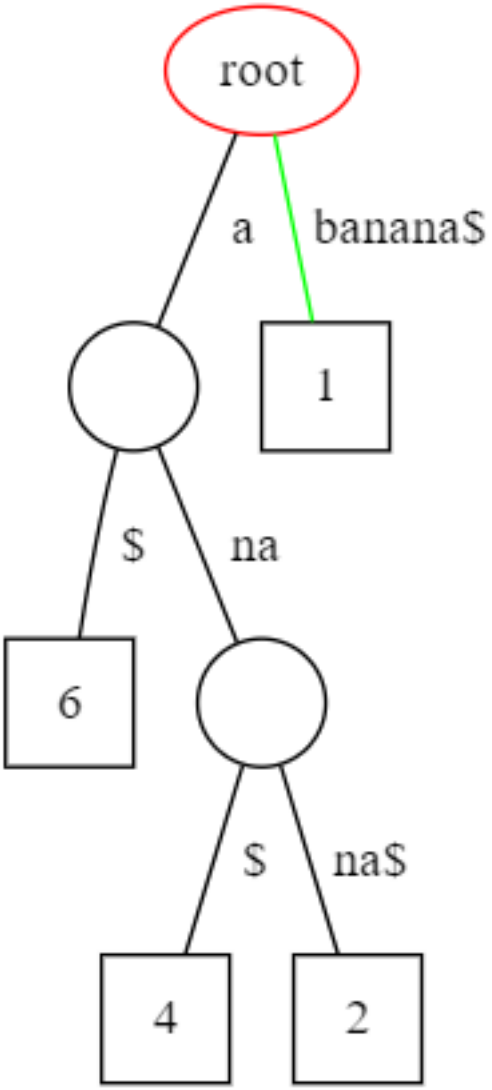
$\#L_{n'} = 3 \;\; LCP = 3 \;\; i = 2$



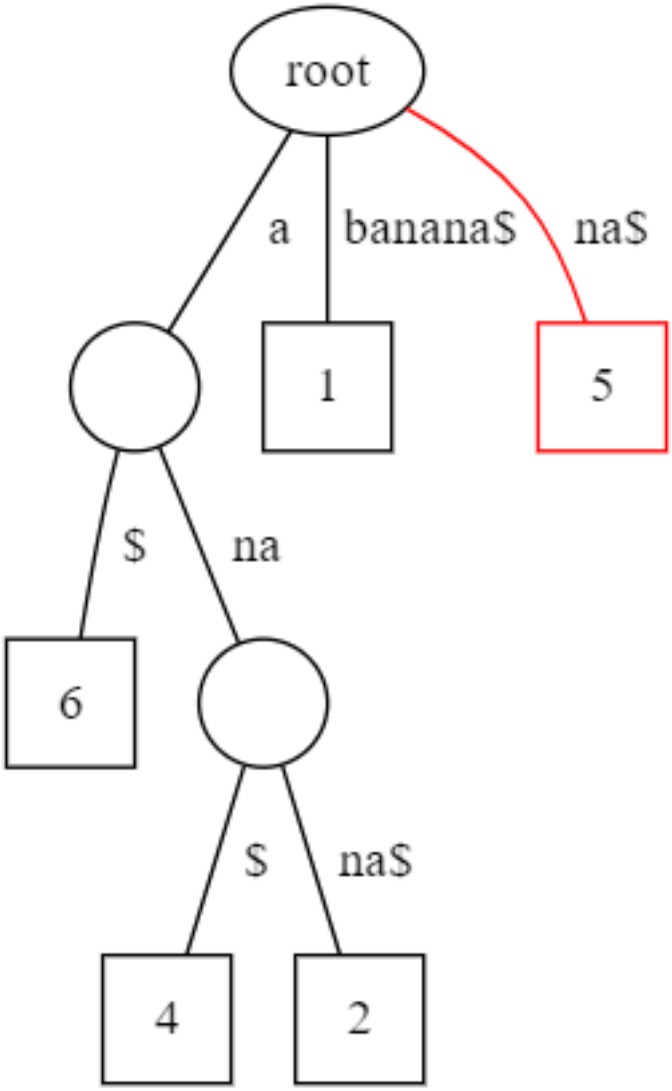
W tej sytuacji trzeba wracać się aż dojedzimy do korzenia (żeby $\#L_{n'}$ było równe 0).



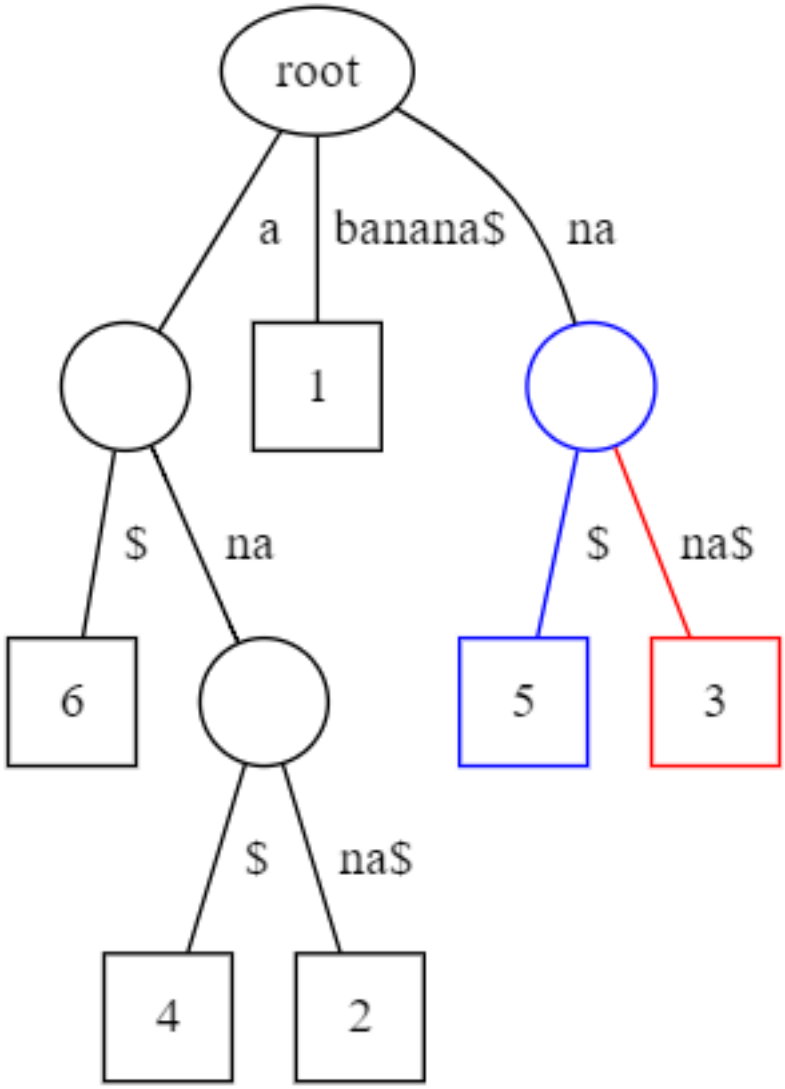
$\#L_{n'} = 6 \;\; LCP = 0 \;\; i = 3$



$\#L_{n'} = 0 \;\; LCP = 6 \;\; i = 4$



$\#L_{n'} = 0 \;\; LCP = 6 \;\; i = 4$



$\#L_{n'} = 2 \;\; LCP = 2 \;\; i = 5$

Zadanie 4

Consider a LZ parse consisting of zphrases and describing a textT[1..N]. Show how to constructa structure of size $O(zlogN)$ capable of extracting anyT[i..j] in $O((j-i+1)+logN)$ time. For extra credit: an efficient $\Theta(n)$ time construction algorithm.

Na wejściu dostajemy Z krotek, które kodują nam jakiś tekst T.

Jak będzie wyglądać nasza struktura - najpierw ponumerujmy krotki od 0 do Z. Teraz dla każdej krotki możemy wyliczyć w czasie $O(n)$ przedział, który będzie oznaczać fragment T, który dana krotka skompresowała. Oznaczmy tą wartość przed $Z[\_].start$. Można zauważyć, że wystarczy nam tylko indeks początkowy, jako że w krotce mamy zawartą informację o długości fragmentu który ona kompresuje. Wtedy dana krotka i, reprezentuje fragment $T[Z[i].start \,..\, Z[i].length]$.

Jak mamy przedziały, to możemy znaleźć w czasie $O(logN)$ (binary search) krotkę (lub krotki), która odpowiada za dany fragment tekstu (czyli nasze zapytania z zadania). Pozostaje nam dodanie jakiejś struktury, która pozwoli nam je odtworzyć ten fragment T, który ona reprezentuje. Aby go odzyskać, należy sprawdzić rekurencyjnie kolejne krotki, które zostały użyte do zakodowania naszej krotki - nazwijmy ją x. Wiemy, że nasza krotka x użyła jakiegoś fragmentu ze słownika, w zaczynającego się od indeksu $x.i$ oraz mającym długość $x.len$. W takim razie możemy łatwo wyznaczyć krotki, które odpowiadają za ten fragment - wystarczy wyszukać które krotki w Z odpowiadają za fragment tekstu $T[x.i \,...\, x.len]$. Możemy to oczywiście zrobić w czasie $O(logN)$ (znowu binary search). Teraz

mając tą listę krotek (oznaczmy przez x.desc), możemy dla nich sprawdzić, które ich fragmenty potrzebujemy. Oczywiście wystarczy to zobaczyć dla $x.desc[0]$ oraz $x.desc[-1]$, ponieważ krotki na tej liście które są pomiędzy nimi będą w całości wykorzystane. Dodatkowo oznaczamy, za który fragment krotki x odpowiadają. Czynność powtarzamy dla każdej krotki. Ostatecznie zużyjemy $O(Z \cdot X)$ pamięci, gdze X, to miejsce, jakie zajmują nasze listy potomków, które tworzą dane node'y. Wydaje mi się, że X powinien być ograniczony przez $O(logN)$ - możemy zauważyć, że w najgorszym przypadku, każdy node będzie miał listę wielkości $Z-1, Z-2, Z-3 \dots 1$, czyli Z^2 ale wtedy $Z = logN$ (bo dostajemy optymalną kompresję lz77 - każdy node jest dwukrotnością dict'a), zatem dostajemy wielkość pamięciową $logn \cdots logN$, czyli tak jak chcemy. Zatem zamortyzowanie powinniśmy mieć maksymalnie $O(Z \cdot logN)$ zajętej pamięci na te listy.

Jak chcemy wyciągać z tej struktury wartości - na wejściu dostajemy i, j

1. Szukamy w $O(logZ)$ listę krotek, które tworzą poszukiwany tekst $T[i \dots j]$. Zauważmy, że wystarczy nam rozpatrywać je osobno, ponieważ możemy podzielić tekst odpowiednio tak, aby każdy przedział wpadał do jedej krotki i po uruchomieniu na nich algorytmu, skonkatenować wynik. Rozpatrzymy zatem przypadek, kiedy $T[i \dots j]$ wpada do jedej krotki.
2. Kiedy binsearch znajdzie nam krotkę k, to możemy rozpatrzyć 3 podproblemy:

- Tekst, który poszukiwany jest tworzony w całości przez krotkę - możemy wtedy do wyniku dodać $k.char$ a następnie rekursywnie wykonać się na kolejnych krotkach z listy $k.desc$ od końca i dodawać ich output do wyniku. W rekursywnych wykonaniach musimy przekazywać ile jeszcze liter zostało nam do odzyskania (żebyśmy wiedzieli ile odzyskać w elemencie $k.desc[0]$) aby nie wygenerować zbyt długiego tekstu. Zauważmy, że przejdziemy przez maksymalnie $O(j-i)$ krotek, bo w każdej z nich dodamy jakąś literkę (chyba że mamy inny case, ale o tym niżej), więc potrzebujemy $O(j-i)+O(logZ)$. Jako że $O(logZ) \leq O(logN)$, to dostajemy złożoność $O(j-i)+X$, gdzie X będzie dodatkowym kosztem, który może wyniknąć z innego podprzypadku i tam zostanie wyjaśniony.
- Tekst, który szukamy, jest częścią jakiejś krotki k, ale jest domnięty z prawej strony (np. $k = (0, 5, c)$, odpowiada za fragment tekstu [10..15] i poszukiwany tekst to T[13..15]). Wtedy mamy podobny przypadek co wyżej, tylko kończymy przechodzenie listy potomków kiedy zbierzemy odpowiednią liczbę literek.
- Tekst, który szukamy, jest częścią jakiejś krotki k, ale jest domnięty z lewej strony (np. $k = (0, 5, c)$, odpowiada za fragment tekstu [10..15] i poszukiwany tekst to T[10..13]). Teraz mamy problem, bo nie weźmiemy literki $k.char$ - nie należy ona do poszukiwanego przez nas fragmentu. Musimy zatem iść rekursywnie do $k.desc$, ale tym razem od początku i dodawać literki. Problemem jest to, że już nie mamy fajnego założenia, że kazde wejście do krotki dodaje nam jedną literkę, tylko w najgorszym przypadku możemy przechodzić przez takie krotki, które sa domknięte z lewej strony i nic nie dodawać. Stwierdzam, że nas to nie boli, bo każdy taki skok powinien zmniejszyć nam tekst, który opisuje fragment który nas interesuje o połowę. Zatem możemy wykonać maksymalnie $logN$ skoków (i tu pojawia się tajemniczy X z 1 podprzypadku). Zatem ostatecznie mamy $O(j-i)+O(logN)+O(logN) = O(j-i)+O(logN)$.
- Tekst, który szukamy, jest w środku krotki - ten podprzypadek w zasadzie jest krótki - można zauważyc, że jeśli z tego podprzypadku, dojdziemy kiedyś do którego, który jest powyżej, to niż nigdy na niego nie natrafimy (bo zawsze będziemy mogli brać kolejne literki idąc z lewej bądź prawej strony $k.desc$). Ale może być taka sytuacja, że będziemy chcieli wielokrotnie szukać jakiegoś fragmentu, który jest w środku jakiejś krotki. Po pierwsze musimy wykonać binsearch na $k.desc$, aby można było znaleźć listę krotek, które tworzą nasz fragment. Zajmie to nam $O(logZ)$, bo maksymalnie w $k.desc$ możemy mieć $Z-1$ elementów. Następnie kontynuujemy na nich algorytm tak jak wcześniej. Pozostaje kwestia tego, jak wpłynie to na złożoność. Każdy taki binsearch coś nasz kosztuje, ale można zauważyć, że każdy kolejny tekst node'a (na którym użyliśmy binsearcha) należącego do $k.desc$ powinien być dwukrotnie mniejszy (szczególnie gdy $Z = O(logN)$, więc możemy zamknąć tą złożoność pod $O(logN)$.

Zatem ostatecznie otrzymujemy $O((j-i+1)+logN)$ na przeszukiwanie.