Struktur Data Eertree dan Variasinya dalam Menyelesaikan Permasalahan Substring Palindrom

Hocky Yudhiono – 1906285604
hocky.yudhiono@ui.ac.id
Fakultas Ilmu Komputer, Universitas Indonesia

1 Pendahuluan

Karya ini merupakan hasil studi pustaka dari ***Eertree: An efficient data structure for processing palindromes in strings*** oleh Rubinchik, M. dan Shur, A. M. pada tahun 2018. Kemiripan observasi dan alur pendekatan permasalahan yang serupa merupakan hasil pemikiran dari penulis *paper* tersebut. Penjelasan, bukti, dan elaborasi ditulis ulang dalam bentuk pemahaman saya.

Palindrom merupakan salah satu bentuk dari sebuah *string*. Sebuah *string* S dengan panjang n disebut palindrom ketika *string* tersebut sama dengan kebalikan dari *string* itu sendiri. Dengan kata lain, berlaku $S = c_1c_2 \dots c_n = c_n \dots c_2c_1$. *Palindromic Tree* atau biasa dikenal dengan Eertree merupakan salah satu struktur data yang ditemukan dan dirilis jurnalnya oleh Mikhail Rubinchik pada tahun 2017. Eertree dapat melakukan berbagai macam operasi terhadap palindrom pada sebuah *string* S dalam kompleksitas waktu $O(n \log |\sigma|)$, untuk $|\sigma|$ merupakan kardinalitas dari himpunan σ yang merupakan karakter berbeda yang menyusun S. Selanjutnya, kita definisikan pula $S[L, R]$ sebagai *substring* inklusif dari indeks L hingga R.

2 Masalah

Ada banyak jenis operasi yang dapat dilakukan oleh Eertree. Berikut ialah beberapa permasalahan yang dapat dikerjakan dengan Eertree.

2.1 Substring Palindrom Berbeda

2.1.1 Deskripsi dan Batasan

Diberikan sebuah *string* S sepanjang n, tentukan ada berapa banyak *substring* palindrom tak kosong berbeda yang terdapat untuk setiap *prefix* S, yaitu *substring* $S[1, 1], S[1, 2], S[1, 3], \dots, S[1, n]$. Tanpa kehilangan sifat umum, definisikan batasan-batasan untuk masalah ini ialah sebagai berikut.

- $1 \leq n \leq 1\,000\,000$
- *String* S hanya terdiri dari huruf latin kecil ($\texttt{a} - \texttt{z}$). Dengan kata lain, $\sigma = [\texttt{a}, \texttt{z}]$ dan $|\sigma| = 26$.

2.1.2 Contoh Masukan dan Keluaran

Algoritma menerima masukan sebuah *string* S sepanjang n dan akan mengembalikan n buah bilangan bulat yang menyatakan banyaknya *substring* palindrom berbeda untuk setiap *prefix*

$S[1, i]$ $(1 \leq i \leq n)$. Misalkan masukan merupakan sebuah *string* $S =$ `eertree`. Observasi untuk setiap *prefix* $S[1, i]$ ialah sebagai berikut.

- Untuk $i = 1$, $S[1, 1] =$ `e`. Hanya terdapat sebuah *substring* palindrom tak kosong berbeda, yaitu {`e`}.
- Untuk $i = 2$, $S[1, 2] =$ `ee`. Terdapat dua, yaitu {`e`, `ee`}.
- Untuk $i = 3$, $S[1, 3] =$ `eer`. Terdapat tiga, yaitu {`e`, `ee`, `r`}.
- Untuk $i = 4$, $S[1, 4] =$ `eert`. Terdapat empat, yaitu {`e`, `ee`, `r`, `t`}.
- Untuk $i = 5$, $S[1, 5] =$ `eertr`. Terdapat lima, yaitu {`e`, `ee`, `r`, `t`, `rtr`}.
- Untuk $i = 6$, $S[1, 6] =$ `eertre`. Terdapat enam, yaitu {`e`, `ee`, `r`, `t`, `ertre`}.
- Untuk $i = 7$, $S[1, 7] =$ `eertree`. Terdapat tujuh, yaitu {`e`, `ee`, `r`, `t`, `ertre`, `eertree`}.

Maka algoritma akan mengembalikan $\{1, 2, 3, 4, 5, 6, 7\}$. Selain itu, struktur data Eertree ini sendiri masih ada dan bisa digunakan untuk berbagai operasi-operasi lain yang akan dijelaskan pada bagian selanjutnya.

3 Studi Pustaka

3.1 Struktur Data Eertree

Permasalahan yang dijelaskan di atas sebelumnya diketahui dapat diselesaikan dalam waktu linear menggunakan algoritma Manacher dan/atau struktur data Suffix Tree secara *online* [Rubinchik and Shur, 2018]. Istilah *online* di sini ialah apabila sewaktu-waktu *string* S ditambahkan karakternya, maka jawaban masih dapat diperoleh dan tidak perlu menghitungnya dari awal. Selain menggunakan algoritma yang di atas, permasalahan ini dapat dikerjakan menggunakan Eertree yang lebih sederhana dan implementasi-nya yang lebih singkat.

Eertree mendukung operasi `add(c)`, yaitu menambahkan sebuah karakter ke belakang *string* S, dan dapat memperbaharui struktur datanya dalam $O(n)$. Namun perlu ditekankan bahwa kompleksitas ini merupakan kasus terburuk dalam memasukkan sebuah karakter. Selanjutnya, dapat dibuktikan bahwa secara *amortized*, kompleksitas total yang dibutuhkan dalam melakukan n buah operasi `add(c)` ialah $O(n \log |\sigma|)$. Perlu diketahui bahwa komponen log dalam kompleksitas didapatkan saat menyimpan sisi-sisi atau *edges* dari Eertree. Dalam praktiknya, biasanya σ merupakan himpunan yang kecil sehingga bisa disimpan menggunakan *array* sederhana yang membuat kompleksitas dari pembangunan Eertree ini menjadi linear.

Untuk pembahasan lebih lanjut, *substring* kosong atau dinotasikan dengan ε, tidak termasuk ke dalam *substring* palindrom yang dimaksud di dalam pembahasan *paper* ini.

Lema 3.1. *Untuk setiap operasi* `add(c)`*, paling banyak hanya akan ada satu substring palindrom berbeda yang akan bertambah.*

Bukti. Untuk *string* S sepanjang n, paling banyak terdapat n *substring* palindrom berbeda, dan hanya akan ada paling banyak satu *substring* palindrom berbeda yang akan bertambah bila dilakukan operasi `add(c)`, yaitu melakukan *append* karakter c terhadap *string* S saat ini. Dengan induksi, akan dibuktikan bahwa ini benar.

Pada tahap basis, definisikan pada awalnya *string* $S = \varepsilon$, yang merupakan *string* kosong. *String* ini memiliki 0 *substring* palindrom berbeda. Pada tahap induksi, asumsikan benar untuk setiap *string* sepanjang k untuk $k \geq 0$. Akan dibuktikan pula untuk *string* sepanjang $k + 1$. Berikut analisis tiga kasus yang berkaitan dengan operasi `add(c)`, dengan penomoran karakter dalam *string* dimulai dari satu.

- Apabila karakter c yang baru ditambahkan $\notin S[1, n]$, maka hanya akan satu *substring* palindrom yang ditambahkan, yaitu $S[n + 1, n + 1]$ atau *substring* yang hanya berisi karakter c saja. Dalam kasus ini, banyaknya *substring* palindrom berbeda akan bertambah satu.

- Anggap $n > 0$, maka sebuah *substring* palindrom berbeda bisa jadi akan bertambah satu — karena bisa saja tidak bertambah — pada karakter $c \in S[1, n]$ jika dan hanya jika terdapat *substring* palindrom $S[i, n]$, dengan $1 < i$, dan berlaku $S[i-1] = c$ atau karakter ke-$(i - 1)$ adalah c. Serta, tidak ada *substring* palindrom $S[j, n]$, di mana $S[j - 1] = c$ dengan $1 < j < i$. Dengan kata lain, $S[i, n]$ adalah *suffix* palindrom terpanjang dari *string* saat ini sedemikian hingga $S[i - 1] = c$. Perhatikan juga bila substring palindrom ini memang sudah pernah ada sebelumnya pada $S[1, n]$, maka substring palindromm tidak bertambah sama sekali.

  Penjelasan Berdasarkan definisi palindrom, perhatikan bahwa *string* $u = S[i - 1, n]c$ merupakan sebuah palindrom. Apabila terdapat j, $(1 < j < i)$ sedemikian sehingga *string* $v = S[j - 1, n]c$ juga merupakan palindrom, maka jelas bahwa u merupakan *substring* dari v. Tentunya *substring* palindrom u sudah pernah dihitung sebelumnya. Karena v bukan *substring* dari setiap kemungkinan *substring* yang ada pada Sc, maka terdapat penambahan sebuah *substring* palindrom berbeda, yaitu *string* v.

- Kasus terakhir, apabila tidak ada i yang memenuhi kasus kedua. Bisa saja *suffix*-nya merupakan sebuah *string* kosong, yang berarti secara intuitif hanya akan ada paling banyak satu *substring* palindrom yang bertambah, yaitu $S[n + 1, n + 1]$ atau *string* yang hanya mengandung satu karakter c. Namun dalam kasus ini, bila $c \in S[1, n]$, maka jumlah *substring* palindrom berbeda akan tetap.

Berdasarkan semua kasus tersebut, maka benar bahwa paling banyak terdapat n *substring* palindrom berbeda dalam *string* sepanjang n. □

Eertree merupakan sebuah struktur data berbentuk *tree* dengan masing-masing *node* merepresentasikan *substring* palindrom berbeda yang ada pada dalam sebuah *string*. Eertree dalam bentuk paling dasarnya merupakan sebuah graf berarah dan *tree*.

Demi kemudahan, akan dibuat penomoran untuk *node* ke-i sebagai representasi *node* v. Setiap *node* v akan disimpan beberapa informasi, antara lain sebagai berikut.

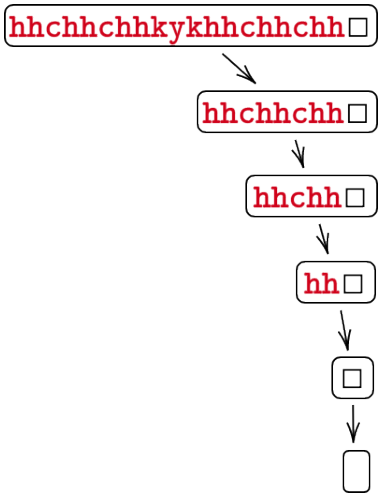
- $len[v]$ — panjang dari *subpalindrom* yang direpresentasikan.
- $link[v]$ — *node suffix link* selanjutnya.
- $edge[v][c]$ — sisi atau *edge* yang menghubungkan *node* v dengan anak-anaknya yang *berkorespondensi* dengan karakter c.

Sebuah *node* v yang merepresentasikan sebuah *string* T akan memiliki anak pada karakter c jika dan hanya jika terdapat *substring* palindrom cTc yang dihubungkan oleh $edge[v][c] = u$, dengan u ialah indeks *node* yang merepresentasikan *string* cTc. Untuk definisi dari *suffix link* yang lebih jelas, akan dibahas pada bagian selanjutnya dalam bab ini.

Eertree memiliki 2 *root*, dengan kata lain Eertree bisa dikatakan merupakan sebuah *forest* dengan dua buah *tree*. Dua *root* ini merupakan *node* khusus yang merepresentasikan palindrom kosong. *Root* pertama merepresentasikan sebuah palindrom ganjil yang kosong dan memiliki indeks -1. Berlaku $len[-1] = -1$ serta $link[-1] = -1$. Root kedua merepresentasikan sebuah palindrom genap yang kosong dan memiliki indeks 0. Berlaku $len[0] = 0$ serta $link[0] = -1$.

Berdasarkan bukti yang telah dipaparkan sebelumnya, ketahui bahwa terdapat paling banyak $n + 2$ *node* (n buah *substring* palindrom berbeda ditambah 2 *node* khusus) dalam Eertree yang merepresentasikan sebuah *string* S dengan panjang n.

Definisi 3.2. *Suffix link sebuah node u, yaitu $link[u] = v$. v merupakan proper suffix palindrom terpanjang dari u.*



Gambar 1: Ilustrasi *Suffix Link* untuk *String* `hhchhchhkykhhchhchh`

Lema 3.3. *Pada node ke-v, setiap $edge[v][c] = u$ akan mengarah ke node u dengan $len[u] = len[v] + 2$. Secara trivial, node selain root akan memiliki tepat satu edge yang masuk ke node tersebut.*

Proposisi 3.4. *Untuk string S sepanjang n, Eertree dari string ini dapat dibuat secara online dalam kompleksitas waktu terburuk $O(n \log |\sigma|)$.*

Bukti. Perhatikan bahwa akan dilakukan iterasi untuk setiap karakter untuk S dari 1 sampai n. Dalam iterasi ke-i, akan disimpan posisi penunjuk *node* (*pointer*) saat ini. Penunjuk *node* akan menunjuk ke *node* yang merepresentasikan *substring* palindrom terpanjang yang berakhir pada indeks i.

Berdasarkan Lema 3.3, setiap *node* v bisa kita anggap memiliki level atau tingkatan berdasarkan panjangnya, yaitu $len[v]$. Definisikan penunjuk saat ini merupakan p untuk indeks i. Diketahui bahwa *node* yang ditunjuk oleh p memiliki panjang $len[p]$ dan merepresentasikan *substring* palindrom $S[i - len[p] + 1, i]$.

Saat melakukan penambahan karakter baru $c = S[i+1]$, maka akan dibandingkan apakah $S[i - len[p]] = c$. Bila tidak sama, maka kita dapat membandingkannya dengan *suffix link* dari p, atau dibuat menjadi $p' := link[p]$. Proses ini tentunya menurunkan panjang dari p untuk iterasi selanjutnya. Bila saat melakukan *traversal* dan membandingkan $S[i-len[p]] = c$ bernilai benar. Akan ada dua kasus yang harus diobservasi.

- Bila $edge[p][c]$ sudah ada, maka atur $p = edge[p][c]$. Tentunya, *node* ini sudah ada sebelumnya, dan jumlah *substring* palindrom berbeda tidak akan bertambah.
- Bila $edge[p][c]$ belum mengarah ke *node* mana pun, maka ditemukan sebuah *substring* palindrom baru yang belum pernah ada di dalam Eertree. Akan dibuat *node* baru dengan indeks terkecil yang belum ada di dalam *tree*, yang merepresentasikan *substring* $cS[i - len[p]+1, i]c = S[i-len[p], i+1]$. Definisikan *node* ini sebagai u, arahkan $edge[p][c] := u$, dan $len[u] = len[p]+2$. Sekarang, akan dicari *suffix link* dari *node* u. *Suffix link* dari *node* ini merupakan *suffix* palindrom kedua terpanjang yang berakhir pada indeks $i+1$. Dalam mencarinya, dapat dilakukan *traversal* terus menerus melanjutkan iterasi $p' = link[p]$, hingga ditemukan p' sedemikian sehingga $S[i - len[p']] = c$.

Pencarian *suffix link* tentu saja akan berhenti, dengan kasus basis saat $p = -1$, tentunya *node* tersebut akan berakhir dengan sebuah *substring* satu karakter c dalam kasus terburuk pencariannya. Kompleksitas waktu dari algoritma penambahan karakter ini dapat dianalisis secara *amortized*.

Meskipun *traversal suffix link* terlihat dapat terjadi dalam $\Theta(n)$ dalam kasus terburuknya, secara keseluruhan dalam pembuatan *tree* ini tidak akan mencapai $\Theta(n^2)$. Secara intuitif, misalkan terdapat dua *pointer* atau penunjuk, l dan r. l menunjuk kepada batas kiri dari *substring* palindrom terpanjang pada iterasi ke-r. Dalam setiap iterasinya, r akan bertambah 1 karena karakter baru akan ditambahkan. Saat melakukan *traversal suffix link* untuk mencari nilai p selanjutnya, penunjuk l akan bertambah 1 atau bergerak ke kanan, hingga berlaku $S[l-1] = S[r+1]$ dan l akan bergerak ke kiri sebanyak satu kali setiap iterasinya.

Perhatikan bahwa pergerakan ke kanan akan terjadi paling banyak n kali secara keseluruhan, dan tepat n kali ke kiri (tidak menutup kemungkinan terdapat sebuah iterasi yang membuat pergerakan ke kanan dilakukan sebanyak r kali secara langsung). Secara total, pergerakan tersebut tidak akan bergerak lebih dari $2n$ kali.

Selain mencari nilai p selanjutnya, mencari *suffix link* dari *node* yang baru juga sama. Bisa direduksi menjadi kasus mencari *suffix* palindrom kedua terpanjang yang ada. Simpan pula sebuah nilai m yang menunjuk pada *suffix* palindrom kedua terpanjang pada iterasi ke-r. Dengan argumen yang sama, pergerakan m secara total tidak akan bergerak lebih dari $2n$ kali.

Faktor kompleksitas waktu $\log |\sigma|$ muncul saat mencari nilai dari $edge[v][c]$, untuk sembarang *node* v dan karakter c. Dengan struktur data *hash*, atau *array* sederhana, nilai indeks ini dapat dicari dalam kompleksitas waktu $O(1)$. Secara analisis, bisa digunakan struktur data *Balanced Binary Search Tree* sederhana untuk himpunan σ yang lebih banyak. Sehingga, kompleksitas waktu dalam membuat Eertree untuk *string* S dengan panjang n ialah $O(n \log |\sigma|)$, dengan kompleksitas memori $O(n)$. □

4 Implementasi dan Eksperimen

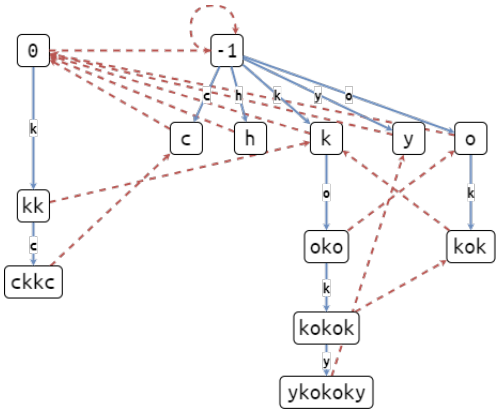
Berikut ialah pseudocode dari bagian inisialisasi dari Eertree.

Algoritma 1: Melakukan inisialisasi pada Eertree

```
1 len[0] := −1                          // Indexing 0-based, 2 node khusus
2 link[0] := 0
3 len[1] := 0
4 link[1] := 0
5 lastPointer := 0                      // lastPointer awalnya pada node -1
6 n := 0                                // Panjang string yang sudah diproses
7 size := 2                             // Ukuran Eertree
```

Pseudocode dari bagian penambahan karakter c pada Eertree saya cantumkan pula. Kompleksitas waktu yang diperlukan untuk bagian ini ialah $O(n)$. Secara total, dalam menambahkan n karakter, dibutuhkan waktu $\Theta(n)$ saja.

Berikut ialah ilustrasi berjalannya algoritma saat memproses *string* `hckkcykokoky`.



Gambar 2: Ilustrasi Eertree untuk *String* `hckkcykokoky`

Pada mulanya hanya akan ada *node* yang bertanda 0 dan −1 sebagai *root*. Garis putus-putus berwarna merah ditandai sebagai *suffix link* dan garis biasa berwarna biru menandakan adanya *edge* dari *node* tersebut ke anak-anaknya, dengan huruf pada *edge* merupakan karakter yang *berkorespondensi* atas hubungan tersebut. *Node* penunjuk saat ini akan menunjuk pada *node* yang di-*highlight* biru.

Gambar 3: Kondisi Eertree Setelah Inisialisasi

Pada awalnya *node* akan penunjuk berada pada *node* −1. Saat ditambahkan huruf `h`, akan dicek apakah *suffix* dengan panjang $-1 + \mathbf{2} = 1$ karakter, termasuk karakter yang baru ditambahkan ini merupakan palindrom. Secara trivial, jelas bahwa *suffix* ini merupakan karakter. Untuk *node* dengan panjang 1, *suffix link* secara khusus akan dihubungkan pada *node* 0. Begitu pula selanjutnya bila ditambahkan *node-node* lain. Perhatikan bahwa pada saat *traversal suffix*

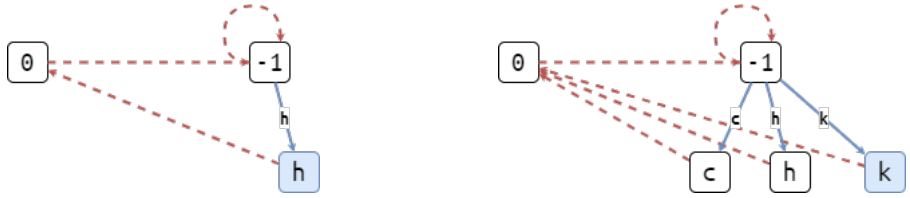
Algoritma 2: Algoritma untuk melakukan `add`(c) pada Eertree

Masukan:

c — karakter yang ingin ditambahkan
n — panjang *string* yang sudah dimasukkan ke Eertree
$lastPointer$ — penunjuk *node* setelah memasukkan karakter terakhir, yaitu $S[n]$
S — *string* yang diproses
$edge[v][c]$ — *adjacency node* untuk *node* v terhadap setiap karakter $c \in \sigma$
$link[v]$ — *suffix link* dari node v
$len[v]$ — panjang dari *node* v
$size$ — ukuran Eertree

while $n - len[lastPointer] - 1 < 0$ **atau** $S[n - len[lastPointer] - 1] \neq c$ **do**
    `// Mencari longest suffix palindrom terbaru`
    $lastPointer = link[lastPointer]$
end
if *edge[lastPointer][c]* belum ada **then**
    `// Membuat node baru`
    $newNode := size$
    $size := size + 1$
    $edge[lastPointer][c] := newNode$
    $len[newNode] = len[lastPointer] + 2$
    if *len[newNode] = 1* **then**
        `// Kasus saat panjang = 1`
        $link[lastPointer] := 1$
    else
        `// Mencari second best suffix palindrom`
        $secondBest := link[lastPointer]$
        while $S[n - len[secondBest] - 1] \neq c$ **do**
            $secondBest := link[secondBest]$
        end
        $link[newNode] := edge[secondBest][c]$
    end
end
$n := n + 1$
$lastPointer := edge[lastPointer][c]$

link pada *node* 0, akan dicek apakah *suffix* palindrom dengan panjang 2, (misal `hh`, `cc`, atau `kk`) merupakan palindrom. Tentunya dalam kasus ini pencocokan *string* tidak berhasil, sehingga akan kembali ke *suffix link*-nya, yaitu -1.

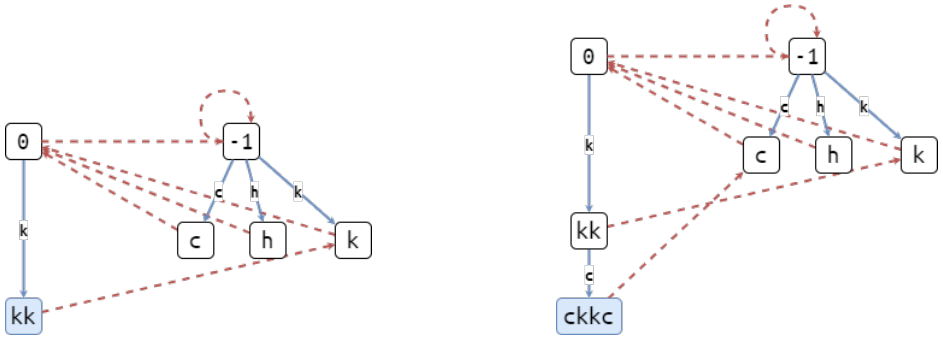


(a) $S[1, i] =$ `h` (b) $S[1, i] =$ `hck`

Gambar 4: Kondisi Eertree

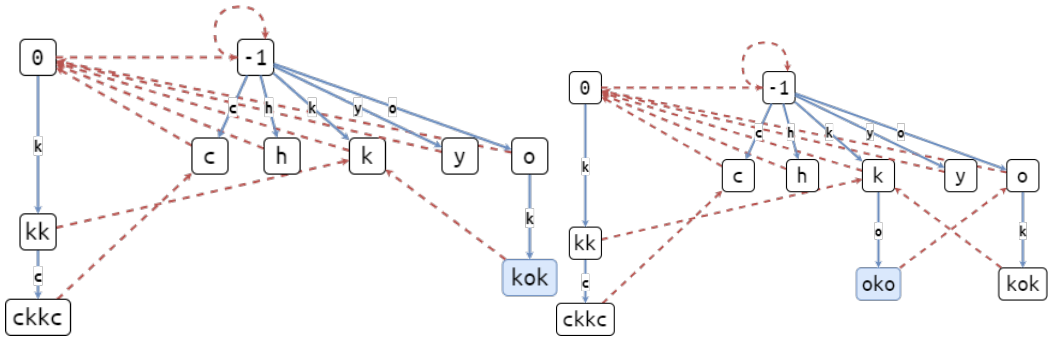
Perhatikan bahwa saat penambahan karakter ke-4, terdapat *suffix* palindrom `kk` dengan panjang 2, selanjutnya akan dibuat *node* baru yang merepresentasikan *substring* `kk`. Akan dicari *suffix link* untuk *node* baru ini. Dari `k` akan menuju ke *node* 0 *traversal suffix link*-nya, ditemukan *suffix* palindrom dengan panjang 1, yaitu `k`. Sehingga *node* baru `kk` yang dibuat akan dihubungkan ke `k`.

Pada saat penambahan karakter ke-5 kasusnya mirip. Awalnya *traversal suffix link* akan dimulai dari `k`, akan dicek apakah terdapat *node* `ckc`, yang bisa menjadi potensial arah *suffix link* dari *node* baru `ckkc`, ternyata tidak ada. Kemudian akan lanjut ke *node* 0, ditemukan *node* `c` sebagai *suffix link* dari `ckkc`.



(a) $S[1, i] =$ `hckk` (b) $S[1, i] =$ `hckkc`

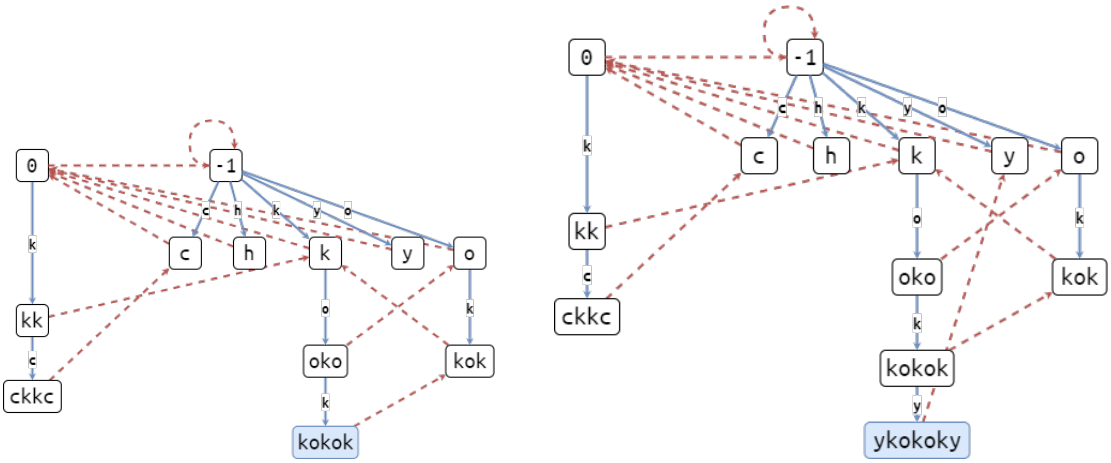
Gambar 5: Kondisi Eertree



(a) $S[1, i] =$ `hckkcykok` (b) $S[1, i] =$ `hckkcykoko`

Gambar 6: Kondisi Eertree

Proses yang sama dapat dilakukan saat penambahan *string-string* selanjutnya, serupa dengan menangani *string* dengan panjang genap. Selanjutnya perhatikan juga bahwa saat penambahan karakter `k` pada Gambar 7.a. *Suffix link* pada *node* baru `kokok` akan dihubungkan menuju *node* `kok`, karena *suffix link* kedua *terpanjang*-nya merupakan `kok`.



(a) $S[1, i] =$ `hckkcykokok` (b) $S[1, i] =$ `hckkcykokoky`

Gambar 7: Kondisi Eertree

Selanjutnya, akan dicoba implementasi Eertree tersebut pada C++. Menggunakan kode yang sama, dilakukan percobaan 5 kali untuk masing-masing *string* S yang terdiri dari huruf latin kecil acak (`a - z`) sepanjang n dan dihitung rata-rata dari penggunaan waktunya hanya dalam pembuatan Eertree untuk *string* tersebut. Program juga dijalankan pada komputer penulis, sehingga perbedaan waktu pada mesin yang berbeda tidak dapat dihindari. Eksperimen ini dilakukan hanya untuk mengetahui laju peningkatan waktu komputasi.

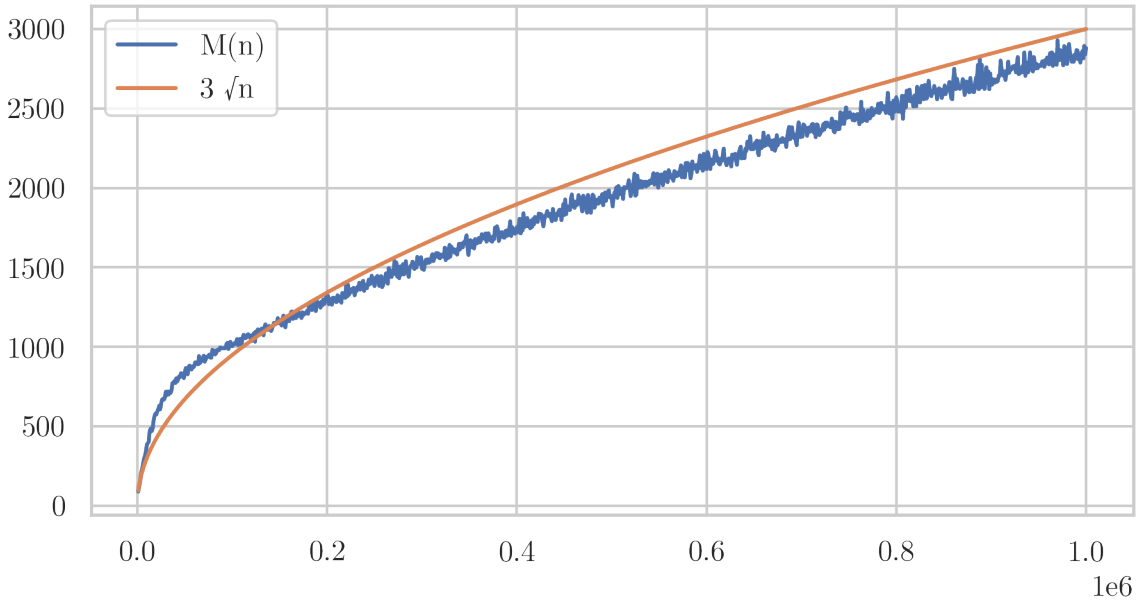
n	10^3	10^4	10^5	10^6	10^7	10^8
t (aproksimasi dalam ms)	0.002	0.006	1.4126	10.9952	92.2064	1043.3976

Tabel 1: Waktu Pembuatan Eertree untuk *String* S Sepanjang n Tertentu

Untuk $n = 10^8$, waktu yang diperlukan ialah sekitar 1 detik. Bila diperhatikan, disini perkembangan waktunya linear. Hal ini disebabkan karena $|\sigma|$ yang berjumlah tetap, yaitu 26. Alokasi memori disediakan oleh sistem operasi menggunakan array. Sehingga bila dihitung menggunakan ram model, kompleksitas waktu yang diperlukan ialah $\Theta(n)$. Dari eksperimen dan pembuktian, dapat terlihat bahwa kecepatan fungsi waktu tumbuh memang bersifat linear pula. Namun berbeda dengan kompleksitas memorinya. Dilakukan eksperimen pula untuk banyaknya verteks rata-rata dengan batasan yang sama. Dilakukan 5 kali percobaan, dan dihitung rata-ratanya.

n	10^3	10^4	10^5	10^6	10^7	10^8
Percobaan 1	77	381	996	2850	12540	29929
Percobaan 2	96	345	1042	2913	12591	30001
Percobaan 3	72	370	1019	2813	12577	30023
Percobaan 4	90	383	1003	2867	12595	29923
Percobaan 5	71	390	992	2839	12670	30062
Rata-rata	81.2	373.8	1010.4	2856.4	12594.6	29987.6

Tabel 2: Banyaknya Verteks Eertree untuk *String* S Sepanjang n Tertentu



Gambar 8: Nilai *String* S Acak Sepanjang n terhadap Banyaknya Verteks pada Eertree

Berdasarkan eksperimen yang saya lakukan dan pembuktian yang sudah pernah dilakukan juga, diketahui bahwa kompleksitas memori rata-rata yang dibutuhkan Eertree untuk *string* S acak sepanjang n ialah $\Theta(\sqrt{n|\sigma|})$ [Rubinchik and Shur, 2016].

5 Aplikasi dan Variasi Eertree

5.1 Mencari Kemunculan Jumlah Setiap Substring Palindrom

Perhatikan bahwa saat menambahkan satu karakter c, kemunculan *substring-substring* palindrom (tidak harus berbeda) akan bertambah. *Substring-substring* ini merupakan semua *substring* dari root 0 atau -1 hingga ke *node* saat ini. Bila kita perhatikan kembali Gambar 7.a. Saat menambahkan karakter k, maka akan bertambah kemunculan dari *kokok*, *kok*, dan k. Untuk menghitung ada berapa banyak kemunculan masing-masing palindrom ini, bisa ditambahkan sebuah nilai $occ[lastPointer]$ untuk setiap *node* yang dikunjungi oleh $lastPointer$ sembari memasukkan setiap karakter. Setelah pembuatan Eertree selesai, iterasi dapat dilakukan untuk indeks *node* yang terbesar hingga yang terkecil, dengan melakukan penambahan seperti *suffix sum*, dengan kata lain $occ[link[v]] := occ[link[v]] + occ[v]$. Setelah komputasi ini, maka akan didapatkan kemunculan *substring* tersebut untuk setiap *node* v.

Algoritma 3: Menghitung kemunculan jumlah setiap *substring*

```
for i := 1 to n do
    add(S[i])
    occ[lastPointer] := occ[lastPointer] + 1
end
for i := size to 1 do
    occ[link[i]] := occ[link[i]] + occ[i]
end
```

5.2 Mencari Banyaknya Substring Palindrom yang Diakhiri pada Indeks Tertentu

Lema 5.1. *Suffix link membentuk tree dengan root pada* -1 *dan* 0.

Bukti. Perhatikan bahwa setiap *node* memiliki tepat satu *suffix link* dengan asumsi bahwa *suffix link* yang menghubungkan -1 dengan dirinya sendiri tidak dihitung, maka tepat terdapat $n+1$ *edge* dan $n+2$ *node* yang terbentuk dari *tree suffix link* ini untuk sebuah *string* S sepanjang n. Karena *suffix link* sebuah *node* v, yaitu $link[v] = u$ hanya menghubungkan *proper suffix* palindrom dan berlaku $len[u] < len[v]$. Maka dipastikan tidak ada *cycle* yang terbentuk pada graf ini. Sehingga graf ini merupakan *tree*. □

Perhatikan bahwa banyaknya *string* yang diakhiri pada suatu indeks R ialah panjang atau kedalaman *node* saat ini bila dilihat dari *tree* yang terbentuk dari *suffix link*-nya. Sehingga, dapat dilakukan *Breadth First Search* atau *Depth First Search* dari *node* -1 hingga ke *node* n. Selanjutnya, setelah melakukan $add(c)$ untuk setiap indeks i, perlu disimpan $idx[i] := lastPointer$ nilai penunjuk *node* setelah memasukkan karakter ke-i, yaitu $S[i]$.

Kemudian simpan pula nilai $occ[v]$ untuk setiap *node* pada Eertree yang awalnya nilai satu. Setelah dilakukan *Depth First Search* yang serupa dengan *prefix sum* melalui *tree suffix link*-nya, atau $occ[v] := occ[v] + occ[link[v]]$. Banyaknya *substring* palindrom yang diakhiri pada indeks R ialah $occ[idx[R]]$. Untuk mendapatkan banyaknya *substring* palindrom yang dimulai pada suatu indeks L, dapat dibuat sebuah Eertree untuk kebalikan atau *reverse* dari *string* S. Selanjutnya lakukan komputasi yang serupa. Banyaknya *substring* palindrom yang dimulai pada suatu indeks L ialah $occ[idx[n - L + 1]]$.

Algoritma 4: Menghitung banyak *substring* yang diakhiri pada indeks R

```
for i := 1 to n do
    add(S[i])
    idx[i] := lastPointer
end
for i := 2 to size do
    occ[i] := 1
    occ[i] := occ[i] + occ[link[i]]
end
return occ[idx[R]]
```

5.3 Lebih Dari Satu String

Eertree tidak hanya bisa mewakili satu *string* saja. Ada variasi lain dari Eertree yang bisa dimanfaatkan. Eertree yang dibentuk dari beberapa *string* dapat dibuat dengan membuat Eertree untuk *string* pertama, mengembalikan penunjuk *node* saat ini ke posisi *node* awal, yaitu -1. Kemudian lakukan operasi pembuatan Eertree kembali untuk *string* kedua, dengan tidak mengabaikan *node-node*, *edges*, dan *suffix link* yang sudah ada pada *Eertree* saat ini. Begitu seterusnya hingga setiap *string* selesai diproses. Pada umumnya, untuk setiap *node*, akan disimpan pula suatu penanda apakah *node* ini ada pada *string* ke-i. Sehingga, setiap *node* akan menyimpan nilai tambahan $exist[v][i]$ yang menandakan bahwa *node* ke-v terkandung pada saat membuat *string* ke-i.

5.4 Operasi Penghapusan

Selain operasi `add(c)`, terdapat variasi Eertree yang mendukung operasi `pop()`. Salah satu yang menjadi hambatan dalam mendukung operasi ini ialah operasi `add(c)` yang memiliki kompleksitas waktu $O(n)$. Sehingga operasi `pop()` dan `add(c)` terus menerus pada kasus terburuk pencarian *suffix link* terpanjang-nya, bisa menyebabkan kompleksitas $O(nq)$. Untuk q banyaknya operasi `pop` yang dilakukan. Salah satu kasusnya ialah:

$$\underbrace{\texttt{add(h)}, \texttt{add(h)}, \ldots, \texttt{add(h)}, \texttt{add(h)}}_{\text{Sebanyak } n \text{ kali}} \underbrace{\texttt{add(o)}, \texttt{pop()}, \ldots, \texttt{add(o)}, \texttt{pop()}}_{\text{Sebanyak } q \text{ kali}}$$

Setiap kali operasi `add(o)` dilakukan, pencarian *suffix link* dari `o` akan terus menerus dilakukan dengan *suffix* palindrom sebagai berikut.

$$\underbrace{hhhh\ldots hhhh}_{\text{Sepanjang n}} \rightarrow \underbrace{hhhh\ldots hhh}_{\text{Sepanjang n - 1}} \rightarrow \cdots \rightarrow hhh \rightarrow hh \rightarrow h \rightarrow \text{Node } 0 \rightarrow \text{Node } -1$$

Terdapat beberapa alternatif dalam mempercepat proses `add(c)` ini, oleh penemu struktur data ini, dinamakan *quick link* dan *direct link*.

5.4.1 Quick Link

Quick link pada dasarnya mempercepat proses saat ada beberapa karakter yang sama dalam sebuah rantai *suffix link*. Saat *traversal suffix link*, akan dilakukan perbandingan-perbandingan karakter sebelum *substring* dengan karakter baru yang akan ditambahkan untuk setiap *suffix* palindrom.

Pada *suffix* palindrom seperti `hhh...hhh` atau `haha...haha`, huruf `h` akan dibandingkan berulang-ulang kali dengan kompleksitas panjang $\Theta(n)$ dengan n panjang *substring* tersebut. Secara intuitif, perhatikan bahwa untuk dua huruf berbeda dalam sebuah rantai *suffix link* bisa terbuat, maka panjang palindrom harus setidaknya dua kalinya. Misalnya kita akan mencoba secara *greedy* untuk membuat sebuah rantai *suffix* palindrom terpendek sehingga saat tidak ada dua huruf berurutan yang sama akan dibandingkan dalam *traversal*-nya. Tanpa mengurangi sifat umum, anggap dua huruf di sini `a` dan `h`.

$$\texttt{h} \rightarrow \texttt{ah} \rightarrow \texttt{hhah} \rightarrow \texttt{ahahhah} \rightarrow \texttt{hhahhahahhah} \rightarrow \ldots$$

Pembuktian lebih formal dan lengkapnya dapat dilihat pada jurnal [Kosolobov et al., 2015]. Karena adanya sifat ini, maka *quick link* dapat mempercepat *traversal suffix link* dan membuat kompleksitasnya hanya menjadi $O(\log n)$ untuk setiap operasi `add(c)`. Untuk tambahan memori sendiri hanya perlu $\Theta(1)$ untuk setiap *node*-nya. *Implementasi*-nya sederhana, cukup menambahkan sebuah variabel baru $quickLink[v] = u$ yang menandakan u adalah *node* pertama dalam rantai *suffix link* yang karakter ke $S[i - len[u]]$ nya tidak sama dengan $S[i - len[v]]$.

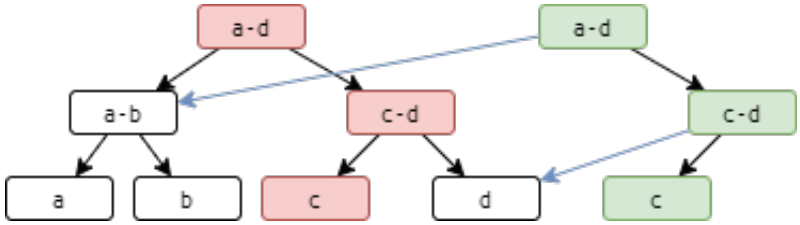
5.4.2 Direct Link dengan Naive Array

Berbeda dengan *quick link*, sesuai namanya, akan disimpan sebuah *link* yang menghubungkan ke *suffix* palindrom dengan karakter yang langsung sama dengan karakter yang dibandingkan. Definisikan sebuah variabel baru, misalnya $u = direct[v][c]$ yang merupakan *suffix link direct* ke *node* yang memiliki karakter $S[i - len[u]] = c$. Perhatikan bahwa kini kompleksitas waktunya menjadi $\Theta(\log |\sigma|)$ dengan asumsi *direct link* disimpan menggunakan suatu *Balanced Binary Search Tree*, dan kompleksitas memorinya akan bertambah sebanyak $\Theta(|\sigma|)$ untuk setiap *node*-nya.

Namun perhatikan bahwa biasanya banyak karakter dalam aplikasi sehari-hari tidak banyak, hanya 26 pada umumnya, sehingga kompleksitas waktu dan memori tambahan untuk setiap *node*-nya bisa dianggap konstan.

5.4.3 Direct Link dengan Persistent Edges

Untuk mempercepat implementasi *direct link* yang sebelumnya, kita dapat menggunakan suatu struktur data persisten. Dalam kasus ini, yang paling mudah diimplementasi ialah Persistent Segment Tree. Untuk intuisinya, anggap terdapat sebuah *segment tree* untuk setiap *node*-nya. Pada dasarnya sebuah *segment tree* pada leaf ke-c akan menyimpan *direct link* ke-c untuk suatu *node* v. Kemudian, bila ditambahkan modifikasi persisten, kurang lebih *segment tree*-nya akan berbentuk sebagai berikut.



Gambar 9: Ilustrasi Persistent Segment Tree

Perhatikan bahwa *tree* pada sebelah kiri istilahnya menunjukkan versi lama, dan pada saat melakukan *insertion* pada *segment tree*, akan dilakukan tambal sulam dan dibuat *node-node* baru sebanyak $\Theta(\log n)$ untuk n ukuran *segment tree*, atau setara dengan kedalamannya. Perhatikan bahwa sekarang hanya perlu disimpan *root* dari setiap versi, dan bisa diakses nilai-nilai dari setiap versinya. Sama halnya dalam kasus *direct link* ini, setiap versi diibaratkan seba-

gai suatu *node* v baru. Setiap kali ada *node* v baru yang karakternya berbeda dengan rantai sebelumnya, maka bisa ditambahkan suatu tambal sulam baru.

Kompleksitas memori *direct link* yang sebelumnya harus menyimpan nilai-nilai lainnya secara naif kini bisa turun. kompleksitas waktunya menjadi $O(\log |\sigma|)$ untuk operasi `add(c)` dan kompleksitas memorinya menjadi $O(\log |\sigma|)$ pula. Untuk lebih ketatnya, perhatikan bahwa bila karakter dalam rantai sama, maka tidak perlu dilakukan *insertion* pada *persistent segment tree*, dan berdasarkan bukti yang sudah dipaparkan sebelumnya, kompleksitas rantai *quick link* paling panjang hanya $O(\log n)$, maka kompleksitas memorinya ialah $O(\min(\log |\sigma|, \log(\log n)))$.

Pustaka

[Borozdin et al., 2017] Borozdin, K., Kosolobov, D., Rubinchik, M., and Shur, A. M. (2017). Palindromic Length in Linear Time. In Kärkkäinen, J., Radoszewski, J., and Rytter, W., editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:12, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Driscoll et al., 1989] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. (1989). Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124.

[Fici et al., 2014] Fici, G., Gagie, T., Kärkkäinen, J., and Kempa, D. (2014). A subquadratic algorithm for minimum palindromic factorization. *Journal of Discrete Algorithms*, 28:41–48.

[Kosolobov et al., 2013] Kosolobov, D., Rubinchik, M., and Shur, A. M. (2013). Finding distinct subpalindromes online.

[Kosolobov et al., 2015] Kosolobov, D., Rubinchik, M., and Shur, A. M. (2015). Pal^k is linear recognizable online.

[Mieno et al., 2022] Mieno, T., Watanabe, K., Nakashima, Y., Inenaga, S., Bannai, H., and Takeda, M. (2022). Palindromic trees for a sliding window and its applications. *Information Processing Letters*, 173:106174.

[Rubinchik and Shur, 2016] Rubinchik, M. and Shur, A. M. (2016). The number of distinct subpalindromes in random words. *Fundamenta Informaticae*, 145(3):371–384.

[Rubinchik and Shur, 2018] Rubinchik, M. and Shur, A. M. (2018). Eertree: An efficient data structure for processing palindromes in strings. *European Journal of Combinatorics*, 68:249–265. Combinatorial Algorithms, Dedicated to the Memory of Mirka Miller.

[Rubinchik and Shur, 2020] Rubinchik, M. and Shur, A. M. (2020). Palindromic k-factorization in pure linear time.

Lampiran

Kode Sumber 1: Eertree

```cpp
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;
#define sz(x) (int)(x).size()

struct EerTree {
  struct Node {
    int next[26];
    int suffixLink;
    int length;
    Node() {
      memset(next, 0, sizeof(next));
      suffixLink = length = 0;
    }
  };
  string s;
  int n, lastPointer, lastIndex;
  vector <Node> tree;
  EerTree(const string &_s) {
    s = _s; tree.resize(3); init();
    for (int i = 0; i < sz(s); i++) {
      addChar(s[i]);
    }
  }
  EerTree(int _n) {
    s = ""; tree.resize(3); init();
  }
  void append(char ch) {
    s += ch; assert(sz(tree) > sz(s));
    addChar(s.back());
  }
  void init() {
    tree[0].suffixLink = 0;
    tree[0].length = -1;
    tree[1].suffixLink = 0;
    tree[1].length = 0;
    lastPointer = n = 0;
    lastIndex = 1;
  }
  void addChar(char ch) {
    int let = ch - 'a';
    while (n - tree[lastPointer].length - 1 < 0 || s[n - tree[lastPointer].length - 1] != ch)
      lastPointer = tree[lastPointer].suffixLink;
    if (!tree[lastPointer].next[let]) {
      tree[lastPointer].next[let] = ++lastIndex;
      if(lastIndex >= sz(tree)) tree.push_back(Node());
      tree[lastIndex].length = tree[lastPointer].length + 2;
      if (tree[lastIndex].length == 1) tree[lastIndex].suffixLink = 1;
```

```
      else {
        int ancestor = tree[lastPointer].suffixLink;
        while (s[n - tree[ancestor].length - 1] != ch) ancestor = tree[ancestor].suffixLink;
        tree[lastIndex].suffixLink = tree[ancestor].next[let];
      }
    }
    lastPointer = tree[lastPointer].next[let];
    n++;
  }
};

int main() {
  cin.tie(0)->sync_with_stdio(0);
  cin.exceptions(cin.failbit);
  string s; cin >> s;
  auto start = high_resolution_clock::now();
  EerTree solve(s);
  auto stop = high_resolution_clock::now();
  auto duration = duration_cast<microseconds>(stop - start);
  // cout << duration.count() << endl;
  cout << solve.lastIndex - 1 << endl;
}
```

Kode Sumber 2: Kode Plot Kompleksitas Memori

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(7, 3.5))
sns.set_style(style='whitegrid')
sns.set_style({'font.family': 'CMU Serif'})
sns.lineplot(x = X, y = Y)
sns.lineplot(x = X, y = 3 * np.sqrt(X))
plt.legend(labels=["M(n)", "3\u221An"])
plt.savefig('memory.png', dpi = 300)
plt.show()
```

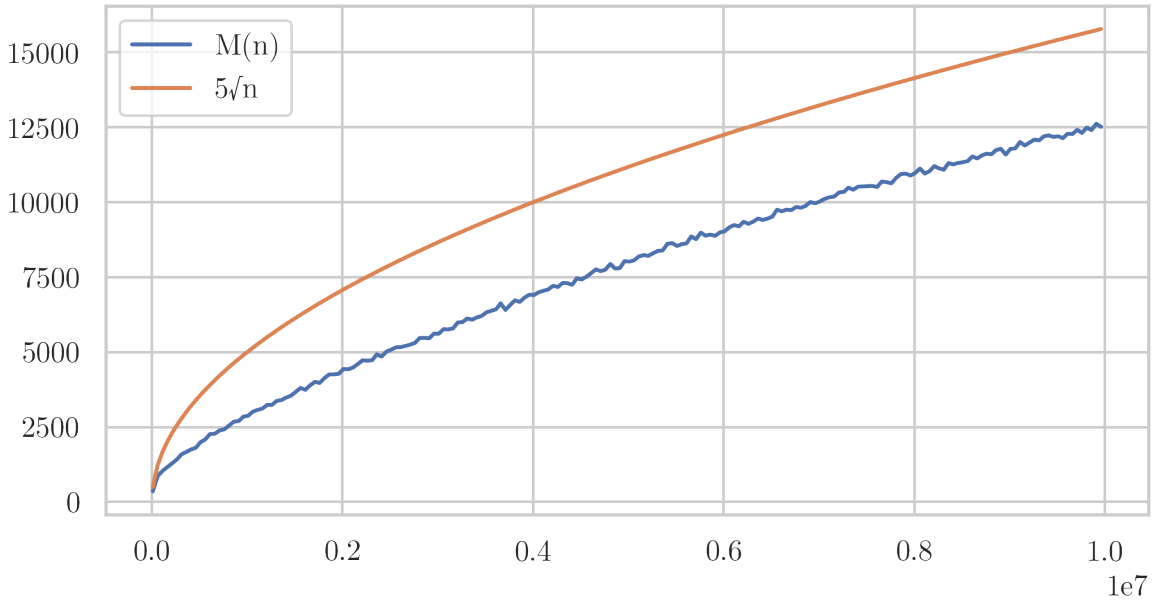
Kode Sumber 3: Kode Generator *String* Acak

```
#include <bits/stdc++.h>
using namespace std;
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

int getRange(int a, int b){
   int ran = b-a+1;
   return (rng()%ran)+a;
}

```

```cpp
int main(){
  int n; cin >> n;
  for(int i = 1;i <= n;i++){
    cout << (char)(getRange(0, 25) + 'a');
  }
  cout << endl;
}
```



Gambar 10: Nilai *String* S Acak Sepanjang n terhadap Banyaknya Verteks pada Eertree