Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Busca de padrões em textos

algoritmos e estruturas de dados

Marco Alves de Alcantara

Monografia Final

mac 499 — Trabalho de
Formatura Supervisionado

Supervisora: Profª. Drª. Cristina Gomes Fernandes

São Paulo
2022

Agradecimentos

Gostaria de agradecer primeiramente aos meus avós, Raimundo e Lourdes, bem como à minha mãe Fabiana, por todo o apoio que eles me deram ao longo dessa jornada.

Agradeço também à professora Cris por toda a ajuda durante o desenvolvimento desse TCC, principalmente pelo auxílio na escrita dessa monografia, com todas as sugestões e revisões feitas ao longo da escrita desse texto. Também agradeço pelo oferecimento de MAC0385 – Estruturas de Dados Avançadas em 2021, o qual serviu de base para tudo que foi feito nessa monografia.

Também gostaria de mencionar aqui os demais docentes do IME-USP por contribuírem na minha formação.

Resumo

Marco Alves de Alcantara. **Busca de padrões em textos: *algoritmos e estruturas de dados***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Essa monografia aborda o problema de encontrar todas as ocorrências de uma palavra em um texto. Definiremos diversas estruturas de dados que podem ser usadas em uma resolução eficiente do problema e descreveremos os algoritmos necessários para construí-las e utilizá-las. As estruturas utilizadas permitem realizar várias buscas sucessivas de palavras diferentes sobre o texto original de maneira computacionalmente eficiente. Uma possível utilidade disso é contar o número de ocorrências de certas palavras em um texto longo, ou ainda o número de ocorrências de certas sequências de DNA no genoma humano. Estamos supondo que o texto é imutável, portanto essas estruturas de dados não são adequadas para aplicações como editores de texto, onde o texto pode ser alterado em tempo de execução.

Palavras-chave: Buscas em texto. Vetor de sufixos. Árvore de sufixos.

Abstract

Marco Alves de Alcantara. **String searching: *algorithms and data structures***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

This report addresses the string search problem. We will define several data structures which can be used to efficiently solve this problem and describe the algorithms needed to construct and use them. The data structures being used also allow us to efficiently search several different strings successively in the original text. A possible use for this is counting the number of occurrences of specific words in a long text, or even counting the number of occurrences of certain DNA sequences in the human genome. We are working under the assumption that the text is immutable, therefore these data structures are not suitable for applications such as text editors, where the text may be modified at runtime.

Keywords: String searching. Suffix array. Suffix tree.

Sumário

Capítulo 1

Introdução

Há vários algoritmos diferentes para buscar uma palavra P em um texto T. O mais simples desses algoritmos consiste em percorrer T caractere por caractere, e, em cada iteração, verificar se P é um prefixo de T a partir daquele caractere. Apesar de esse algoritmo ser simples, não precisar de pré-processamento e nem de estruturas de dados auxiliares, ele tem tempo de execução proporcional a $|P|.|T|$, no pior caso, o que efetivamente significa que mesmo se as palavras forem curtas, ele ainda é proporcional a $|T|$.

Há também outros algoritmos, como o Algoritmo de Boyer-Moore [Sedgewick e Wayne, 2011a] e o KMP [Cormen *et al.*, 2009a], que são mais eficientes pois fazem um pré-processamento de P, o que faz com que sejam necessárias menos comparações diretas de caracteres de P e de T, obtendo uma complexidade de tempo menor (proporcional a $\frac{|T|}{|P|}$ e $|T|$ respectivamente, para o melhor caso possível). Entretanto, nesse TCC iremos considerar aplicações onde o texto é longo e **fixo**, o que significa que esses algoritmos não são ideais, já que o texto é longo. Para essas aplicações, o ideal seria um algoritmo de busca em tempo proporcional a $|P|$.

Em vez de fazer um pré-processamento da palavra, os algoritmos abordados nesse TCC pré-processam o texto. Uma vantagem é que isso possibilita buscar várias palavras diferentes sucessivamente, e, além disso, cada busca é feita em tempo proporcional a $|P|$ (em vez de $|T|$), sem que seja necessário fazer um pré-processamento entre cada uma dessas buscas. Uma desvantagem disso é que armazenar os dados obtidos ao pré-processar o texto consome espaço extra proporcional a $|T|$.

1.1 Preliminares e notação

Seja Σ uma tupla ordenada de símbolos distintos, denominada *alfabeto*. Dizemos que um símbolo x é lexicograficamente menor que y se x ocorre antes de y em Σ. Iremos denotar por \$ um símbolo tal que $\$ \notin \Sigma$, e $\$ < x$ para todo $x \in \Sigma$. Isso é, \$ não ocorre em Σ, mas é considerado lexicograficamente menor que qualquer símbolo do alfabeto. O *rank* de um caractere x em Σ, que será denotado por $\text{rank}_\Sigma(x)$, é a posição de x em Σ. Consideramos que $\text{rank}_\Sigma(\$) = 0$.

Definimos uma *palavra* (ou texto) do alfabeto Σ como sendo uma sequência $a_0 a_1 a_2 \cdots a_m$ de símbolos desse alfabeto Σ. No que segue, deixamos implícito o alfabeto Σ.

Para uma palavra $P = a_0 \cdots a_m$, denotamos por $P[i..j]$ a sequência $a_i \cdots a_j$ contida em P. No caso particular em que $i = j$, essa sequência é denotada simplesmente por $P[i]$. No caso em que $i = 0$, denotaremos a sequência por $P[..j]$, e essa sequência é denominada um *prefixo* de P. Para o caso $j = m$, denotaremos a sequência por $P[i..]$, e essa sequência é denominada um *sufixo* de P. Nos casos particulares em que $i > m$ ou $j < 0$, a sequência resultante é vazia.

Além disso, $|P|$ denota o comprimento da palavra, isso é, o comprimento da sequência de caracteres. Nesse exemplo, vale que $|P| = m + 1$.

Caso P seja a sequência vazia, a palavra será denotada por ϵ. Vale que $|\epsilon| = 0$.

Nessa monografia iremos falar também sobre textos, cuja definição é idêntica à definição de palavra, exceto que, ao longo dessa monografia, iremos considerar que textos geralmente são mais longos que palavras.

Dada uma palavra P e um texto T, dizemos que P *ocorre em* T caso exista um índice i tal que $T[i + j] = P[j]$ para todo j tal que $0 \leq j < |P|$.

Dadas duas palavras não-vazias A e B, dizemos que $A < B$ caso $A[0] < B[0]$, ou $A[0] = B[0]$ e $A[1..] < B[1..]$. Vale também que $\epsilon < P$, para qualquer palavra não-vazia P. Analogamente podemos definir os casos onde $A > B$ e $A = B$.

Para duas palavras $A = a_0 \cdots a_n$ e $B = b_0 \cdots b_m$, a *concatenação* de A e B será a palavra $C = a_0 \cdots a_n b_0 \cdots b_m$, e será denotada por $A \cdot B$. Se $A = \epsilon$, então $A \cdot B = B$, e, analogamente, se $B = \epsilon$, então $A \cdot B = A$; ou seja, ϵ é um elemento neutro para a concatenação.

1.2 Algoritmo trivial de busca de padrão

O algoritmo trivial funciona por meio de dois laços aninhados (linhas 3 e 5 do Algoritmo 1), sendo que o primeiro, indexado por i, é responsável por testar todos os possíveis índices de T a partir dos quais P poderia ocorrer, enquanto o laço interno é responsável por comparar todos os caracteres de P com os caracteres correspondentes em $T[i..]$. Caso P ocorra em T (linha 7), o algoritmo adiciona o índice i da ocorrência ao conjunto devolvido pelo algoritmo (linha 8).

Algoritmo 1: Algoritmo trivial para busca de palavra em texto

Dados: palavra P, texto T
Resultado: conjunto dos índices de ocorrências de P em T

```
1  x ← ∅
2  i ← 0
3  enquanto i ≤ |T| − |P| faça
4  |  j ← 0
5  |  enquanto j < |P| e P[j] = T[i + j] faça
6  |  |  j ← j + 1
7  |  se j = |P| então
8  |  |  x ← x ∪ {i}
9  |  i ← i + 1
10 retorna x
```

Apesar de o Algoritmo 1 ser simples, seu consumo de tempo no pior caso é proporcional a $|T||P|$, e até mesmo no melhor caso, ainda é proporcional a $|T|$, pois é necessário iterar i aproximadamente $|T|$ vezes. Isso torna esse algoritmo ineficiente caso o texto seja muito longo. Nesse TCC, iremos explorar algoritmos de busca que consomem tempo proporcional a $|P|$ para realizar uma busca; esses algoritmos são mais eficientes para textos longos.

No Capítulo 2, iremos apresentar a definição do vetor de sufixos e mostrar como buscar P em T em tempo $\mathcal{O}(|P| + \log(|T|))$ utilizando o vetor de sufixos de T. No Capítulo 3, apresentamos o Algoritmo DC3, que constrói o vetor de sufixos de um texto T em tempo $\mathcal{O}(|T|)$. Já no Capítulo 4 iremos apresentar a definição da árvore de sufixos de um texto, explicar como buscar P em T em tempo $\mathcal{O}(|P|)$ e mostrar como obter a árvore de sufixos de T a partir do seu vetor de sufixos em tempo linear. No Capítulo 5, apresentaremos algumas breves conclusões.

Capítulo 2

Vetor de Sufixos

Seja T' o texto do alfabeto Σ no qual queremos realizar as buscas. Seja também T o texto T' acrescido do caractere de controle \$ no final. Como \$ é um caractere que não ocorre em Σ, ele não ocorre em nenhuma palavra P que possa ser buscada, e sua única ocorrência em T é no final do texto. A partir de agora, sempre que nos referirmos a um texto T, deixaremos implícito que o texto termina com o caractere especial \$.

Os vetores de sufixos, bem como o vetor LCP (que será explicado a seguir) e o algoritmo de busca descrito neste capítulo foram todos propostos como uma alternativa às árvores de sufixos por Udi Manber e Gene Myers em 1990 [Manber e Myers, 1990]. Os vetores de sufixos também foram independentemente descobertos por Gaston Gonnet, Ricardo A. Baeza-Yates e Tim Snider em 1987 sob o nome "vetores PAT" [Gonnet *et al.*, 1992].

Definimos o vetor de sufixos de T, denotado por VS(T), ou simplesmente VS, como sendo o vetor dos índices dos sufixos de T ordenados lexicograficamente. Ou seja, para todo par de índices i e j de VS, se $i < j$, então T[VS[i]..] é lexicograficamente menor que T[VS[j]..] [Wikipedia, 2023].

Exemplo

Para T = "abracadabra\$", temos o seguinte vetor de sufixos:

$T[i..]$	i	VS[i]	T[VS[i]..]
abracadabra$	0	11	$
bracadabra$	1	10	a$
racadabra$	2	7	abra$
acadabra$	3	0	abracadabra$
cadabra$	4	3	acadabra$
adabra$	5	5	adabra$
dabra$	6	8	bra$
abra$	7	1	bracadabra$
bra$	8	4	cadabra$
ra$	9	6	dabra$
a$	10	9	ra$
$	11	2	racadabra$

Tabela 2.1: *Vetor VS de sufixos para "abracadabra$"*

2.1 Vetor LCP

Dado um texto T de comprimento n e seu vetor de sufixos VS, definimos o seu vetor LCP (de *Longest Common Prefix*) como sendo um vetor de comprimento $n - 1$, onde LCP[i] é o comprimento do maior prefixo comum entre T[VS[$i - 1$]..] e T[VS[i]..], para $0 < i \leq n - 1$. O vetor é indexado a partir de 1, isto é, LCP[0] não é definido.

Exemplo

Ainda usando T = "abracadabra$" como exemplo, e aplicando essa definição para o vetor LCP, obtém-se o seguinte resultado:

i	VS[i]	LCP[i]	T[VS[i]..]
0	11		$
1	10	0	a$
2	7	1	**a**bra$
3	0	4	**abra**cadabra$
4	3	1	**a**cadabra$
5	5	1	**a**dabra$
6	8	0	bra$
7	1	3	**bra**cadabra$
8	4	0	cadabra$
9	6	0	dabra$
10	9	0	ra$
11	2	2	**ra**cadabra$

Tabela 2.2: *Vetores de sufixos e LCP para "abracadabra$"*

Dados um vetor LCP e dois índices l e r tais que $0 \leq l < r < |T|$, denotaremos por LCP(l, r) o menor dos valores contidos entre os índices $l + 1$ e r do vetor LCP; ou seja, LCP(l, r) = min{LCP[i]: $l + 1 \leq i \leq r$}. Vale que LCP($r - 1, r$) = LCP[r], e, de um modo mais geral, se LCP(l, r) = k, isso significa que, para todo i tal que $l \leq i \leq r$, o prefixo comum máximo de todos os sufixos de T iniciados nas posições VS[i] tem comprimento igual a k. Essa notação generaliza a ideia do vetor LCP, pois em vez de se restringir a dois sufixos consecutivos, ela se aplica a um intervalo de sufixos em VS.

No exemplo 2.2.1, vale que LCP(6, 7) = 3, pois os sufixos "bra$" e "bracadabra$" têm um prefixo comum máximo de comprimento 3 ("bra"). Temos também que LCP(1, 5) = 1, já que os sufixos entre VS[1] e VS[5] têm um prefixo comum máximo de um único caractere.

2.2 Busca por uma palavra

Para buscar uma palavra P no texto T, iremos fazer uma busca binária no vetor de sufixos pelo índice do sufixo predecessor de P, denotado por L, com o auxílio do vetor LCP. Considerando a ordem lexicográfica, o sufixo predecessor de P é definido como o maior sufixo de T que é menor que P. Ou seja, o algoritmo devolve um índice L tal que $T[\text{VS}[L]..] < P < T[\text{VS}[L+1]..]$. No caso extremo em que $P = \epsilon$, devolvemos $L = -1$, pois $P < T[\text{VS}[0]..] =$ "$". No caso em que P é maior que o maior sufixo de T, devolvemos $|T| - 1$, que é o índice do último sufixo de T.

O algoritmo de busca binária (Algoritmo 3) usa o algoritmo Prefixo Comum (Algoritmo 2), que recebe a palavra P, o texto T e dois índices i e v tais que $i < |P|$ e $v+i < |T|$, e devolve o comprimento do prefixo comum máximo entre $P[i..]$ e $T[v+i..]$.

Para determinar o comprimento do maior prefixo comum entre $P[i..]$ e $T[v+i..]$, basta comparar os caracteres apropriados de P e T diretamente. Como todos os sufixos de T terminam com "$", e esse caractere não ocorre em P, os dois sufixos sempre serão diferentes.

Algoritmo 2: Prefixo Comum

Dados: palavra P, texto T, índice i, índice v em que $i < |P|$ e $v + i < |T|$
Resultado: comprimento do prefixo comum máximo entre $P[i..]$ e $T[v+i..]$

```
1 c ← 0
2 enquanto i + c < |P| e P[i + c] = T[v + i + c] faça
3 |   c ← c + 1      // Não precisamos verificar se v + i + c < |T|, já que
4 retorna c          // a comparação com $ no final do texto vai falhar
```

O Algoritmo 2 (Prefixo Comum) consome tempo linear no número de comparações de caracteres entre o texto T e a palavra P, ou seja, o consumo de tempo é proporcional ao comprimento do prefixo comum entre $P[i..]$ e $T[v+i..]$.

Apresentaremos na página seguinte o Algoritmo 3, que realiza a busca binária por uma palavra com o auxílio do Algoritmo 2, e, em seguida, iremos explicar mais detalhes sobre o Algoritmo 3, incluindo seus invariantes, uma análise geral de seu funcionamento, e a análise de sua complexidade.

Algoritmo 3: Busca Binária com LCPs

Dados: palavra P, texto T, vetores LCP e VS
Resultado: índice do predecessor lexicográfico de P em VS

```
1  se P = ϵ então
2  |  retorna −1
3  R ← |T| − 1
4  r ← PrefixoComum(P, T, 0, VS[R])
5  se r < |P| e P[r] > T[VS[R] + r] então
6  |  retorna R                    // P é maior que o maior sufixo de T
7  L, l ← 0
8  enquanto L < R − 1 faça
9  |  M ← ⌊(R+L)/2⌋
10 |  se l = r então
11 |  |  c ← PrefixoComum(P, T, l, VS[M])
12 |  |  se l + c < |P| e P[l + c] > T[VS[M] + l + c] então
13 |  |  |  L ← M                          // P[l..] > T[VS[M] + l..]
14 |  |  |  l ← l + c
15 |  |  senão
16 |  |  |  R ← M
17 |  |  |  r ← r + c
18 |  senão se l > r então
19 |  |  se l < LCP(L, M) então
20 |  |  |  L ← M
21 |  |  senão se l > LCP(L, M) então
22 |  |  |  R ← M
23 |  |  |  r ← LCP(L, M)
24 |  |  senão
25 |  |  |  c ← PrefixoComum(P, T, l, VS[M])
26 |  |  |  se l + c < |P| e P[l + c] > T[VS[M] + l + c] então
27 |  |  |  |  L ← M
28 |  |  |  |  l ← l + c
29 |  |  |  senão
30 |  |  |  |  R ← M
31 |  |  |  |  r ← l + c
32 |  senão
33 |  |  se r < LCP(M, R) então
34 |  |  |  R ← M
35 |  |  senão se r > LCP(M, R) então
36 |  |  |  L ← M
37 |  |  |  l ← LCP(M, R)
38 |  |  senão
39 |  |  |  c ← PrefixoComum(P, T, r, VS[M])
40 |  |  |  se r + c < |P| e P[r + c] > T[VS[M] + r + c] então
41 |  |  |  |  L ← M
42 |  |  |  |  l ← r + c
43 |  |  |  senão
44 |  |  |  |  R ← M
45 |  |  |  |  r ← r + c
46 retorna L
```

No início do laço principal do Algoritmo 3 na linha 8, vamos argumentar que valem os invariantes que $T[\text{VS}[L]..] < P < T[\text{VS}[R]..]$, que o prefixo comum mais longo entre P e $T[\text{VS}[L]..]$ tem comprimento l, e o prefixo comum mais longo entre P e $T[\text{VS}[R]..]$ tem comprimento r. Nesse laço, imediatamente após a execução da linha 9, também vale que $T[\text{VS}[L]..] < T[\text{VS}[M]..] < T[\text{VS}[R]..]$, já que $L < M < R$ e o vetor VS é ordenado lexicograficamente.

Como $T[\text{VS}[0]..] =$ "\$", e esse caractere não ocorre em nenhuma palavra, se $P \neq \epsilon$, ao entrar no laço da linha 8, então $L = l = 0$ e os invariantes sobre L e l valem. Para garantir os invariantes para R e r, é necessário comparar P com o último sufixo de T, já que é possível que P seja lexicograficamente maior que todos os sufixos de T. Isso é feito nas linhas 3 e 4 e, caso P seja maior que o último sufixo, o algoritmo já retorna $|T| - 1$ como sendo o predecessor de P. Caso contrário, r já guarda o número de caracteres iguais obtido na comparação de P com o sufixo de índice $|T| - 1$ para garantir os invariantes ao entrar no laço da linha 8.

Esses invariantes continuam valendo no início de todas as iterações desse laço. Suponha que eles valem no início de uma iteração qualquer da execução do Algoritmo 3.

Se $l = r$, entramos no caso da linha 10 do algoritmo, e c será igual ao número de comparações bem-sucedidas de caracteres entre $P[l..]$ e $T[\text{VS}[M] + l..]$. Vale que $l + c \leq |P|$, pois o algoritmo PrefixoComum para de comparar os caracteres ao terminar a palavra P. Se $l + c = |P|$, então P é um prefixo de $T[\text{VS}[M] + l..]$, e, como esse sufixo de T é mais longo que P, já que ele termina com \$, que não ocorre em P, então vale que P é estritamente menor que esse sufixo do texto, e o algoritmo segue para a linha 16. Caso $l + c < |P|$ e $P[l + c] < T[\text{VS}[M] + l + c]$, então a primeira comparação mal-sucedida de caracteres entre P e T indica que $P[l..] < T[\text{VS}[M] + l..]$, e, nesse caso o algoritmo também segue para a linha 16. Caso contrário, vale que a palavra P é maior que esse sufixo de T, e o algoritmo seguiria para a linha 13. Como $l = r$, então $P[0..l - 1]$ é um prefixo comum tanto de P quanto de todos os sufixos de índices entre L e R em VS. Se o algoritmo seguir para a linha 13, vale que $P[l..] > T[\text{VS}[M] + l..]$ e, ao concatenar o prefixo $P[0..l - 1]$ ao início de ambos os textos, concluímos que $P > T[\text{VS}[M]..]$, a busca binária avança para o lado direito, e, ao fazermos $L \leftarrow M$, e $l \leftarrow l + c$, incrementando l para contar os c caracteres que foram comparados nessa iteração, preservamos os invariantes do algoritmo para a próxima iteração. Analogamente conclui-se que se seguirmos para a linha 16 do algoritmo, como $l = r$, conclui-se que nesse caso o algoritmo também preserva os invariantes do laço principal da linha 8. Portanto, o caso analisado na linha 10, de fato preserva os invariantes.

Se $l > r$, entramos na linha 18, e iremos comparar l com $\text{LCP}(L, M)$. Como l é o comprimento do prefixo comum máximo entre P e $T[\text{VS}[L]..]$ e $T[\text{VS}[L]..] < P$, isso implica que $T[\text{VS}[L] + l] < P[l]$. Lembrando da definição da notação estendida para o vetor LCP, se $\text{LCP}(L, M) = k$, então todos os sufixos de índices no intervalo $[L, M]$ em VS tem um prefixo comum máximo de comprimento k. Se $l < \text{LCP}(L, M)$ (linha 19), então a busca binária deve avançar para a direita, pois o comprimento do prefixo comum máximo entre $T[\text{VS}[L]..]$ e $T[\text{VS}[M]..]$ é maior que l, ou seja, $T[\text{VS}[M] + l] = T[\text{VS}[L] + l] < P[l]$. Nesse caso, o algoritmo prossegue para a linha 20, e a busca avança para a direita sem fazer comparações de caracteres entre P e T, mantendo os invariantes na próxima ite-

ração. Se $l > \text{LCP}(L, M)$ (linha 21), então a palavra P tem um prefixo comum mais longo com $T[\text{VS}[L]..]$ do que com $T[\text{VS}[M]..]$, pois como $\text{LCP}(L, M) < l$, o primeiro caractere diferente entre os sufixos L e M ocorre antes do primeiro caractere diferente entre P e o sufixo L. Como os sufixos estão ordenados, vale que $P < T[\text{VS}[M]..]$, e a busca binária deve avançar para a esquerda (linha 22 do Algoritmo 3). Além disso, o comprimento do prefixo comum máximo entre P e $T[\text{VS}[M]..]$ terá comprimento $\text{LCP}(L, M)$ já que $l > \text{LCP}(L, M)$. Portanto, nesse caso, são mantidos os invariantes na próxima iteração do algoritmo. Vale também observar que $\text{LCP}(L, R) = r$ pois $l > r$ e portanto $\text{LCP}(L, M) \geq r$. Esse fato significa que, nesse caso, o índice r não diminui ao iniciarmos a próxima iteração. Essa observação é relevante para a análise da complexidade do algoritmo. Se $l = \text{LCP}(L, M)$, o algoritmo prossegue para a linha 25, e realiza comparações diretas entre caracteres de P e T para manter os invariantes, de maneira análoga ao que foi feito na linha 10 no caso em que $l = r$. Com isso, o algoritmo mantém os invariantes na próxima iteração para todas as possíveis comparações entre l e $\text{LCP}(L, M)$ a partir do caso em que $l > r$, tratado na linha 18.

Por fim, o caso $l < r$ é tratado na linha 32, de maneira análoga ao caso anterior. Com isso, o algoritmo mantém os invariantes em todos os possíveis casos no laço principal da linha 8, fazendo com que os invariantes valham no início de todas as iterações. Após sair do laço, o algoritmo simplesmente devolve L, pois, como nesse momento, $L = R - 1$, temos que $T[\text{VS}[L]..] < P < T[\text{VS}[L + 1]..]$, e portanto L é o índice do sufixo predecessor de P em T.

A operação de obter o LCP de um intervalo, denotada no Algoritmo 3 por $\text{LCP}(i, j)$, não é feita diretamente; é possível pré-calcular todos os intervalos necessários, fazendo com que cada chamada a $\text{LCP}(i, j)$ na execução do Algoritmo 3 seja realizada em tempo constante. O pré-processamento é feito apenas uma vez ao processar o texto, o que significa que o resultado não depende de qual palavra está sendo buscada. Ele será explicado com mais detalhes ainda nesse capítulo.

O Algoritmo 3 consome tempo $\mathcal{O}(|P| + \log(|T|))$. As três primeiras linhas são executadas em tempo constante, e a linha 4 em tempo proporcional ao comprimento do prefixo comum entre P e o último sufixo de T. Como o comprimento desse sufixo é limitado superiormente pelo comprimento da palavra P, e as linhas de 5 a 7 do algoritmo são executadas em tempo constante, o Algoritmo gasta, no máximo, tempo linear no comprimento de P até entrar no laço da linha 8.

Quanto ao laço da linha 8, em todos os casos possíveis no fluxo de controle do algoritmo, um dos delimitadores do intervalo da busca binária (L ou R) é movido para o meio do intervalo, ou seja, todas as iterações diminuem o intervalo pela metade, logo, como $|\text{VS}| = |T - 1|$, o laço da linha 8 é iterado $\mathcal{O}(\log(|T|))$ vezes, como seria esperado de uma busca binária. Mas algumas iterações gastam mais tempo do que outras pois nelas é necessário comparar caracteres de P e T diretamente.

Para analisar o consumo de tempo do Algoritmo 3, é possível observar que, a cada iteração do laço da linha 8 em que há comparações diretas entre caracteres de P e T, é chamado o algoritmo PrefixoComum, cujo consumo de tempo é proporcional ao comprimento do prefixo comum máximo entre um sufixo de P e de T. De qualquer forma, a cada chamada a esse algoritmo, o número de comparações bem-sucedidas entre caracteres é devolvido e armazenado em c nas linhas 11, 25 ou 39, e há uma única comparação mal-sucedida entre

caracteres a cada chamada a esse algoritmo. O valor armazenado em c é então somado a l ou r dependendo do fluxo de controle do laço, nas linhas 14, 17, 28, 31, 42 ou 45, mas, de qualquer forma, vale que $l + r \leq 2|P|$ durante toda a execução do laço, já que esses índices são o comprimento do prefixo comum máximo entre P e um sufixo de T. Além dessas somas, esses índices só são alterados pelo laço nas linhas 23 ou 37, mas como foi observado anteriormente na análise dos invariantes do algoritmo, essas escritas jamais decrementam esses índices. Portanto, l e r são índices cuja soma é, no máximo, proporcional a $|P|$, e que jamais são decrementados pelo algoritmo. Juntando isso ao fato de que todas as outras instruções nos casos em que são comparados caracteres entre P e T são executadas em tempo constante, é possível concluir que, todas as iterações em que há comparações de caracteres que são executadas pelo Algoritmo 3 são feitas em tempo linear em $|P|$. Além disso, como todas as demais iterações do laço são executadas em tempo constante já que os valores necessários de $\text{LCP}(i, j)$ são pré-processados, e essas consultas são feitas em tempo constante, então cada iteração em que não há comparação direta de caracteres é realizada em tempo constante.

Como há um número proporcional a $\log(|T|)$ de iterações realizadas em tempo constante cada, e um certo número de iterações cujo consumo de tempo somado é $\mathcal{O}(|P|)$, então o consumo de tempo do Algoritmo 3 é $\mathcal{O}(|P| + \log(|T|))$. Como apesar de T geralmente ser muito mais longo que P, vale que $\log(|T|)$ é muito menor que $|T|$, e pode até mesmo ser comparável a $|P|$, então apesar de o Algoritmo 3 não ser exatamente linear apenas no comprimento de P, ainda é possível que na prática ele consuma tempo aproximadamente proporcional a $|P|$, e, de qualquer forma, ainda é muito mais eficiente que algoritmos com consumo de tempo proporcional a $|T|$.

2.3 Obtenção das ocorrências a partir dos resultados das buscas

Após executar o Algoritmo 3 para um padrão P, teremos um índice p do sufixo predecessor de P. Esse índice tem a propriedade de que, se P ocorre em T, então P será um prefixo do sufixo de índice $p + 1$, pois se P ocorre em T, então existe um sufixo cujo prefixo contém P. Logo, se estivermos interessados apenas em decidir se P ocorre em T ou não, bastaria olhar se P é um prefixo de $T[\text{VS}[p + 1]..]$ ou não.

Já se quisermos saber onde estão as ocorrências, ou até mesmo apenas obter o número de ocorrências, é necessário fazer outra busca binária. O índice p do predecessor funciona como um limitante inferior para o intervalo que contém as ocorrências de P em VS. Para obter o intervalo completo, é também necessário um limitante superior q. Para isso, podemos construir uma palavra Q a partir de P, cujo predecessor sirva de limitante superior a esse intervalo.

Seja z o último caractere do alfabeto do qual as palavras são obtidas. Se todos os caracteres de P são z, não seria necessário fazer outra busca binária, pois nesse caso, $q = |T| - 1$, já que todos os sufixos a partir de $p + 1$ teriam P como prefixo. Caso contrário (P contém pelo menos um caractere que não é z), Q pode ser obtida a partir de P incrementando o último caractere diferente de z, isso é, substituindo-o por seu sucessor no alfabeto, e truncando a palavra para terminar nesse caractere incrementado.

Por exemplo, se P_1 = “abra”, então construiríamos Q_1 = “abrb”, já para P_2 = “abczz”, a palavra construída para a segunda busca seria Q_2 = “abd”.

Se estivéssemos buscando por P = “abra”, os predecessores obtidos seriam os da tabela abaixo.

i	VS[i]	T[VS[i]..]		
0	11	$		
1	10	**a$**	⟵	p = pred(P = “**ab**ra”)
2	**7**	abra$		
3	**0**	**abra**cadabra$	⟵	q = pred(Q = “**abrb**”)
4	3	acadabra$		
5	5	adabra$		
6	8	bra$		
7	1	bracadabra$		
8	4	cadabra$		
9	6	dabra$		
10	9	ra$		
11	2	racadabra$		

Tabela 2.3: *Predecessores obtidos ao buscar P = “abra” em T*

Se P ocorre em T, então P será um prefixo do sufixo de índice q em VS, e o intervalo de suas ocorrências é dado por $[p + 1, q]$. Portanto, o número de ocorrências de P em T é $q - p$, e, caso seja necessário saber onde P ocorre em T, os índices nesse mesmo intervalo do VS são as posições em T a partir das quais P ocorre.

Isso é, se $x_1, \ldots, x_{q-p}$ são os índices entre $p + 1$ e q em VS, então P é um prefixo de cada sufixo $T[x_i..]$. Como a construção de Q pode ser feita em tempo $\mathcal{O}(|P|)$, e estamos apenas fazendo uma busca a mais (pelo predecessor de Q), então contar o número de ocorrências também leva tempo $\mathcal{O}(|P| + \log(|T|))$.

Já obter as ocorrências levaria um certo tempo a mais, pois seria necessário percorrer VS entre as posições $p + 1$ e q, o que pode tornar a busca significativamente mais lenta caso hajam muitas ocorrências. A complexidade nesse caso seria $\mathcal{O}(|P| + \log(|T|) + x)$, onde x é o número de ocorrências.

2.4 Construção do vetor LCP

O algoritmo a seguir foi proposto por Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa e Kunsoo Park [Kasai *et al.*, 2001], e permite construir LCP(T) a partir do texto T e de seu vetor de sufixos.

Algoritmo 4: Construção linear do vetor LCP

```
   Dados: texto T, vetor de sufixos VS
   Resultado: vetor LCP de T
 1 i ← 0
 2 enquanto i < |T| faça
 3 |   rank[VS[i]] ← i
 4 |   i ← i + 1
 5 h, i ← 0
 6 enquanto i < |T| faça
 7 |   se rank[i] > 0 então
 8 |   |   k ← VS[rank[i] − 1]
 9 |   |   enquanto T[i + h] = T[k + h] faça
10 |   |   |   h ← h + 1
11 |   |   LCP[rank[i]] ← h
12 |   |   se h > 0 então
13 |   |   |   h ← h − 1
14 |   i ← i + 1
15 retorna LCP
```

Nas primeiras 4 linhas do Algoritmo 4, é construído um vetor rank, que é o vetor inverso do vetor de sufixos. Isso é, se VS[i] = x, então rank[x] = i. Em seguida, o laço da linha 6 é onde efetivamente ocorre a construção do vetor LCP. Na linha 7, se rank[i] = 0, então i = VS[0], e, nesse caso não faria sentido calcular o LCP para esse índice pois LCP[0] não é definido.

Durante a execução do laço da linha 6 do Algoritmo 4, vale o invariante de que, se rank[i] > 0, ao entrar na linha 8, LCP[rank[i]] ≥ h. Essa condição vale para a primeira iteração, pois h é inicializado com 0 na linha 5, e os valores do vetor LCP são todos maiores ou iguais a 0.

Suponha agora que o invariante valha para alguma iteração do laço; iremos mostrar que ele continua valendo na iteração seguinte. Se $h \leq$ LCP[rank[i]], isso quer dizer que T[VS[rank[i] − 1]..] e T[VS[rank[i]]..] tem um prefixo comum de pelo menos h caracteres, o que significa que o laço interno das linhas 9 e 10 calcula corretamente o valor de LCP[rank[i]] já que os h caracteres que não forem comparados ainda pertencem ao prefixo comum desses sufixos de T, e h é incrementado até que seja encontrado um caractere diferente neles. Se $h \leq 1$, após tentar executar as linhas 12 e 13 do algoritmo, teremos que $h = 0$, e o invariante sobre h continua valendo na próxima iteração. Caso $h > 1$, ao decrementar h na linha 13, o invariante continua válido na iteração seguinte, pois o próximo índice a ser iterado corresponde ao sufixo obtido ao remover o primeiro caractere de T[i..].

Nos casos onde rank[i] > 0, cada iteração no laço da linha 6 calcula o valor de uma posição do vetor LCP. Na linha 8, k é o índice do sufixo anterior a rank[i] em VS; vale que rank[k] = rank[i] − 1. Intuitivamente, na i-ésima iteração do laço, iremos calcular o LCP entre um sufixo x tal que VS[x] = i, e o sufixo x − 1, imediatamente anterior a x no vetor de sufixos. Note que VS[x − 1] = k, isso é, a i-ésima iteração irá calcular LCP[x], onde x = rank[i].

Essa contagem é realizada no laço interno da linha 9, onde comparamos $T[i + h]$ e $T[k + h]$. Como i e k são dois sufixos consecutivos do vetor de sufixos, basta contar o número de caracteres iguais entre eles diretamente para obter LCP[rank[i]]. A variável h indica o número de comparações bem-sucedidas entre caracteres desses sufixos, e é usado para inicializar os índices de LCP na linha 11.

Para entender as linhas 12 e 13 do Algoritmo 4, é necessário lembrar que, se LCP[x] = h > 0, então T[VS[x − 1]..] e T[VS[x]..] tem um prefixo comum máximo de comprimento h, e observar que, se removêssemos o primeiro caractere em comum entre esses prefixos, os dois sufixos obtidos teriam um LCP maior ou igual a h − 1. Esses dois sufixos também aparecem no vetor de sufixos e esse algoritmo itera o laço 6 convenientemente de modo que a próxima iteração irá comparar um desses dois sufixos após remover seu primeiro caractere, com o sufixo imediatamente anterior a ele no vetor de sufixos (pois rank é o inverso de VS). Isso significa que o LCP entre esses dois sufixos sendo comparados será no mínimo h − 1, já que o sufixo atual estará sendo comparado ou com o outro sufixo obtido após remover o primeiro caractere, ou com um sufixo entre esses dois no VS. Portanto, as linhas 12 e 13 e a variável h funcionam como um atalho, lembrando dos resultados de comparações feitas anteriormente a fim de evitar repetir as mesmas comparações, e simplesmente decrementando h se h > 0, e esse atalho é possível pela ordem em que o laço itera os sufixos.

Exemplo

Iremos usar novamente T = “abracadabra$” para exemplificar a execução do Algoritmo 4. Após a execução das primeiros 4 linhas, obtém-se o seguinte vetor rank:

i	VS[i]	rank[i]	T[i..]
0	11	3	abracadabra$
1	10	7	bracadabra$
2	7	11	racadabra$
3	0	4	acadabra$
4	3	8	cadabra$
5	5	5	adabra$
6	8	9	dabra$
7	1	2	abra$
8	4	6	bra$
9	6	10	ra$
10	9	1	a$
11	2	0	$

Tabela 2.4: *Vetor rank obtido durante a construção do vetor LCP*

Em seguida, i e h são inicializados com 0 na linha 5, e os estados da execução do algoritmo a cada iteração i estão listados na tabela abaixo, onde h denota o valor de h ao longo da iteração i e LCP[rank[i]] denota o valor que foi armazenado no vetor LCP imediatamente após as comparações diretas entre caracteres. Os prefixos de T comparados a cada iteração são os indexados por i e k, sendo que h, obtido a partir das comparações de caracteres feitas em iterações anteriores do algoritmo, é um limitante inferior para o comprimento do prefixo comum mais longo entre $T[i..]$ e $T[k..]$. O fato de haver esse limitante é que torna o algoritmo linear em vez de quadrático, pois é possível pular essas comparações a cada iteração.

i	rank[i]	k = VS[rank[i] − 1]	h	LCP[rank[i]]	$T[i..]$
0	3	7	∅ ~~1~~ ~~2~~ ~~3~~ 4	4	**abra**cadabra$
1	7	8	3	3	**bra**cadabra$
2	11	9	2	2	**ra**cadabra$
3	4	0	1	1	**a**cadabra$
4	8	1	0	0	cadabra$
5	5	3	∅ 1	1	**a**dabra$
6	9	4	0	0	dabra$
7	2	10	∅ 1	1	**a**bra$
8	6	5	0	0	bra$
9	10	6	0	0	ra$
10	1	11	0	0	a$
11	0	-	0	-	$

Tabela 2.5: *Estados das variáveis principais do Algoritmo 4 para T = "abracadabra$"*

Por exemplo, na iteração $i = 2$, temos que $k = 9$, e, portanto, iremos calcular o LCP entre $T[2..]$ = "**ra**cadabra$" e $T[9..]$ = "**ra**$". Como h inicia a iteração valendo 2, já sabemos que ambos sufixos tem um prefixo comum de comprimento 2 ("ra"), e após tentar executar o laço interno das linhas 9 e 10, como $T[2 + 2] = c \neq T[9 + 2] = \$$, descobrimos que esse prefixo comum tem comprimento máximo pois a primeira comparação do laço falhou. Isso quer dizer que LCP[11] = 2, e o algoritmo decrementa h para 1, preparando-se para processar o prefixo "**a**cadabra$".

Para entender porque o Algoritmo 4 consome tempo linear em $|T|$, basta observar que seu consumo de tempo será proporcional ao número de vezes que as linhas 9 e 10 serão executadas, pois elas correspondem ao laço interno do algoritmo. A variável h é limitada superiormente por $|T|$ já que os caracteres sendo comparados não irão ultrapassar o final do texto T, e cada comparação bem-sucedida incrementa h em apenas uma unidade. Além disso, a linha 13 é executada no máximo uma vez por iteração do laço 6, e no máximo $|T|$ vezes no total, o que significa que a variável h não pode ser incrementada mais que $2|T|$ vezes pela linha 10. Portanto, o Algoritmo 4 consome tempo proporcional a $|T|$.

2.5 Extensão do vetor LCP

Para que seja possível realizar a busca eficientemente, é necessário pré-processar (*estender*) o vetor LCP para que cada consulta por LCP(L, M) e LCP(M, R) durante a busca pelo predecessor seja feita em tempo $\mathcal{O}(1)$. Para isso iremos criar dois vetores auxiliares novos, LLCP e RLCP (também chamados de *vetores laterais*). Apesar de existirem $\mathcal{O}(|T|^2)$ intervalos distintos no vetor LCP, apenas $\mathcal{O}(|T|)$ desses intervalos podem aparecer durante uma busca binária. O pré-processamento consiste em percorrer recursivamente o vetor LCP de modo similar a uma busca binária, mas a cada passo, em vez de chamar a recursão para apenas uma metade do intervalo atual, iremos chamá-la para ambas as metades, concluindo a extensão do LCP em tempo $\mathcal{O}(|T|)$.

A observação mais importante para entender o papel dos vetores laterais é que, para cada índice M, existe um único intervalo $[L; R]$ que pode ser obtido em um passo de uma busca binária, tal que M seja o meio desse intervalo. Isso porque cada passo diminui o intervalo pela metade, e uma das extremidades da busca se torna o índice M da iteração anterior, o que impossibilita que M se repita em dois passos distintos.

LLCP[M] corresponde a LCP(L, M), que por sua vez é o mínimo de LCP[i] para $L \leq i \leq M$, e RLCP[M] corresponde a LCP(M, R), onde L e R são as extremidades do intervalo da busca binária que obtém M como o índice do meio da busca. Cada vetor lateral é indexado de 1 a $|T| - 2$. O índice 0 não é definido, tal como ocorre com o vetor LCP, mas ao contrário do LCP, o índice $|T| - 1$ também não é definido, isso porque não há como o índice M ser uma das extremidades do vetor, já que ele deve estar estritamente contido entre L e M.

Exemplo

Ainda considerando T = “abracadabra$”:

i	LCP[i]	$[L, R]$	LLCP[i]	RLCP[i]	T[VS[i]..]
0					$
1	0	[0, 2]	0	1	a$
2	1	[0, 5]	0	1	**a**bra$
3	4	[2, 5]	4	1	**abra**cadabra$
4	1	[3, 5]	1	1	**a**cadabra$
5	1	[0, 11]	0	0	**a**dabra$
6	0	[5, 8]	0	0	bra$
7	3	[6, 8]	3	0	**bra**cadabra$
8	0	[5, 11]	0	0	cadabra$
9	0	[8, 11]	0	0	dabra$
10	0	[9, 11]	0	2	ra$
11	2				**ra**cadabra$

Tabela 2.6: *Intervalos da busca que obtém i como índice do meio, e vetores laterais do LCP*

Por exemplo, para $i = 7$, o intervalo da busca binária para o qual $M = 7$ é o correspondente a $L = 6$, $R = 8$, e, nesse caso, LLCP[7] = LCP(6, 7) = 3 (“bra”). Já para $i = 10$, o intervalo é delimitado por $L = 9$, $R = 11$, e RLCP[10] = LCP(10, 11) = 2 (“ra”).

O Algoritmo 5, descrito abaixo, descreve a parte recursiva da extensão.

Algoritmo 5: Estende LCP

Dados: vetor LCP, índices i e j, lado da busca (lado)

```
se i = j − 1 então
    x ← LCP[j]
senão
    M ← ⌊(i+j)/2⌋
    EstendeLCP(LCP, i, M, esquerda)
    EstendeLCP(LCP, M, j, direita)
    x ← min(LLCP[M], RLCP[M])
se lado = esquerda então
    LLCP[j] ← x
se lado = direita então
    RLCP[i] ← x
```

Cada chamada ao Algoritmo 5 que não chega à base da recursão gera mais duas chamadas recursivas, mas por se tratar de uma busca binária, cada chamada reduz o intervalo à metade de seu comprimento atual. Como a parte não-recursiva do Algoritmo 5 é executada em tempo $\mathcal{O}(1)$, se $T(n)$ é o número de unidades de tempo gastas pelo algoritmo para um intervalo de comprimento $n = j - i + 1$, então o consumo de tempo do algoritmo satisfaz a recorrência $T(n) = 2 \times T(\frac{n}{2}) + \mathcal{O}(1)$, com $T(2) = 1$. Essa recorrência pode ser resolvida pelo Teorema Mestre [Cormen *et al.*, 2009b], o que nos permite concluir que $T(n) = \mathcal{O}(n)$, ou seja, a primeira chamada será concluída em tempo $\mathcal{O}(|T|)$.

A chamada principal do Algoritmo 5 para a extensão do vetor LCP é feita por meio do Algoritmo 6, apresentado a seguir.

Algoritmo 6: Início da extensão do LCP

Dados: vetor LCP, inteiro $|T|$

Resultado: vetores laterais LLCP e RLCP

```
LLCP ← ∅
RLCP ← ∅
se |T| > 2 então
    EstendeLCP(LCP, 0, |T| − 1, null)
retorna LLCP, RLCP
```

Caso $|T| \leq 2$, os vetores LLCP e RLCP seriam vazios, mas na prática não faria diferença já que, nesse caso, a busca pelo predecessor não chegaria a usar esses vetores pois ela terminaria imediatamente, sem de fato iterar a busca binária. Além disso, a primeira chamada recursiva é feita com o lado sendo "nulo", já que não há um lado associado a ela, e nós não precisamos das posições 0 ou $|T| - 1$ dos vetores laterais. Também é fácil ver que o Algoritmo 6 tem a mesma complexidade que a primeira chamada recursiva ao Algoritmo 5, e, portanto, também é linear em $|T|$.

Capítulo 3

Construção Linear do Vetor de Sufixos

Neste capítulo, iremos descrever e analisar um algoritmo linear para a construção do vetor de sufixos VS(T) a partir de um texto T, bem como analisar sua complexidade para provar que ele é, de fato, linear em $|T|$. Trata-se do *DC3/Skew Algorithm* (*DC* de "*difference cover*", referente ao fato de o algoritmo dividir o texto em três partes, e "cobrir" uma delas indiretamente, algo que será explicado em detalhes nesse capítulo). O Algoritmo DC3 foi proposto por Juha Kärkkäinen e Peter Sanders em 2006 [Kärkkäinen e Sanders, 2006], e a maneira que ele será explicado nesse capítulo é baseada nas notas de um Gist do Github escrito por Mark Ormesher [Ormesher, 2017].

3.1 Preliminares

Uma observação importante é que para esse algoritmo, em vez de construir o vetor de sufixos para um texto T composto por caracteres em um alfabeto Σ, usaremos uma sequência de números naturais, que de agora em diante será denotada por S. Essa sequência é obtida diretamente a partir de T e, apesar de ser composta de números naturais em vez de caracteres, também será considerada como um texto, e será construída de tal modo que VS(T) = VS(S), da maneira descrita abaixo. O motivo de construirmos VS(T) a partir de S em vez de T diretamente será explicado mais à frente.

Algoritmo 7: Conversão para texto numérico

Dados: texto T
Resultado: texto numérico S equivalente a T

```
1 i ← 0
2 enquanto i < |T| faça
3 |   S[i] = rank_{Σ(T)}(T[i])
4 |   i ← i + 1
5 retorna S
```

No Algoritmo 7, a notação $\text{rank}_{\Sigma(T)}(T[i])$ denota o rank do caractere em $T[i]$ no

alfabeto $\Sigma(T)$, formado pelos caracteres que ocorrem em T. Em particular, $\text{rank}_\Sigma(\$) = 0$. Como a conversão de T para S trata-se apenas da criação de rótulos (ou ranks) para cada caractere, a ordem relativa de cada sufixo é preservada, e ambos os textos têm o mesmo vetor de sufixos.

Exemplo

Supondo que T = "abracadabra\$", tem-se $\Sigma(T) = \{\$, a, b, c, d, r\}$. A tabela a seguir exemplifica a obtenção do texto numérico correspondente:

i	$T[i]$	$S[i]$
0	a	1
1	b	2
2	r	5
3	a	1
4	c	3
5	a	1
6	d	4
7	a	1
8	b	2
9	r	5
10	a	1
11	\$	0

Tabela 3.1: *Associação entre cada índice de T e S*

Portanto, obtém-se $S = (1, 2, 5, 1, 3, 1, 4, 1, 2, 5, 1, 0)$ para T = "abracadabra\$". Como todos os elementos de S nesse exemplo são inteiros de um único dígito, de agora em diante iremos omitir as vírgulas ao escrever S, ou seja, escreveremos $S = 125131412510$.

Durante o algoritmo, serão feitas algumas comparações entre triplas de caracteres de S e, em algumas dessas comparações, poderá ser necessário comparar caracteres cujos índices estariam fora dos limites de S. Nesses casos, para fins de comparações de caracteres, iremos considerar que qualquer caractere com índice fora de S será 0. Um modo de garantir isso ao implementar esse algoritmo é criar uma função auxiliar para acessar os índices de S em vez de acessá-los diretamente, como descrito na função abaixo. Um outro modo de fazer isso é acrescentar um par de zeros ao final de S no início de cada nível da recursão para que todas as triplas necessárias sejam bem-formadas.

Algoritmo 8: RadixKey

Dados: texto S, índice i
Resultado: chave de ordenação associada a $S[i]$

```
1 se i < 0 ou i ≥ |S| então
2 |   retorna 0
3 retorna S[i]
```

O algoritmo é recursivo. No caso base da recursão em que $|S| < 3$, o vetor de sufixos de S é calculado diretamente em tempo $\mathcal{O}(1)$ por meio de comparações diretas dos caracteres de S. Se $|S| \geq 3$, o processo é dividido em três etapas. Antes de explicar essas etapas em mais detalhes, são necessárias mais algumas definições.

De agora em diante, iremos denotar a tripla $(S[i], S[i+1], S[i+2])$ para cada índice i de S por $\tau(i)$. Sejam S_0, S_1 e S_2 três vetores tais que cada vetor S_i é composto pelos inteiros k tais que $0 \leq k < |S|$ e $k \equiv i \bmod 3$. Cada um desses vetores S_i irá representar as triplas de caracteres iniciadas em posições congruentes a $i \bmod 3$ em S. Como $|S| \geq 3$, pois não estamos mais no caso base da recursão, nenhum desses vetores é vazio. Seja também S_{12} a concatenação de S_1 e S_2.

Exemplo

Para T = "abracadabra$", a sequência de índices obtida para S_{12} (no primeiro nível da recursão) está representada na tabela abaixo, bem como qual sufixo e tripla cada um desses índices representa. Os inteiros de S que fazem parte de cada tripla $\tau(S_{12}[i])$ estão sublinhados na coluna da esquerda.

$S[S_{12}[i]..]$	i	$S_{12}[i]$	$\tau_T(S_{12}[i])$
25131412510	0	1	251
31412510	1	4	314
12510	2	7	125
10	3	10	100
5131412510	4	2	513
1412510	5	5	141
2510	6	8	251
0	7	11	000

Tabela 3.2: *S_{12} no primeiro nível da recursão*

Na primeira etapa do algoritmo iremos construir VS_{12} recursivamente e seu inverso, SV_{12}. Na segunda etapa iremos ordenar os sufixos de S indexados por S_0 para obter um vetor VS_0. Por fim, na terceira etapa, iremos juntar esses vetores em tempo linear em $|S|$ de uma maneira parecida com a função Merge do algoritmo MergeSort [Sedgewick e Wayne, 2011b] para obter $\mathrm{VS}(S) = \mathrm{VS}(T)$. A seguir está uma descrição *top-down* do algoritmo.

Algoritmo 9: DC3SkewAlg

Dados: texto numérico S
Resultado: vetor de sufixos VS(S)

```
1 se |S| < 3 então
2 |   VS(S) ← construa VS(S) diretamente
3 |   retorna VS(S)
4 rank12 ← OrdenaTriplas12(S)                    // Início da Etapa 1
5 VS12 ← DC3SkewAlg(rank12)
6 SV12 ← InverteVetor(VS12)                        // Fim da Etapa 1
7 VS0 ← ConstróiVS0(S, VS12)                              // Etapa 2
8 VS(S) ← Merge(S, VS0, VS12, SV12)                       // Etapa 3
9 retorna VS(S)
```

3.2 Construção de SV_{12} e VS_{12}

Essa etapa consiste em ordenar as triplas $\tau(i)$ para cada índice i em S_{12}, obtendo o vetor rank_{12}, obter $VS_{12} = VS(\text{rank}_{12})$ recursivamente, e, então, construir seu inverso, SV_{12}. Mas primeiro, será necessário construir o vetor S_{12} explicitamente; o motivo disso será explicado em detalhes no final do capítulo.

3.2.1 Construção de S_{12}

O vetor S_{12} será obtido diretamente a partir de S em tempo linear em $|S|$. Nesse vetor, os índices das primeiras $|S_1|$ posições serão índices 1 mod 3, e os demais serão índices 2 mod 3. Essa distinção entre os índices de S_1 e de S_2 em S_{12} baseada na relação com $|S_1|$ será útil na etapa 3.

Algoritmo 10: ConstróiS12

Dados: texto numérico S
Resultado: subtexto S_{12} de índices 1 e 2 mod 3 de S

```
1 i ← 0
2 para m ← 1 até 2 faça
3 |   j ← m
4 |   enquanto j < |T| faça
5 |   |   S12[i] = j
6 |   |   j ← j + 3
7 |   |   i ← i + 1
8 retorna S12
```

O algoritmo simplesmente copia os índices j congruentes a m mod 3 na linha 5 para S_{12}. É importante lembrar que, como o algoritmo é recursivo, cada etapa deverá ser realizada em tempo linear em $|S|$ em vez de $|T|$ para que a complexidade total do algoritmo seja linear em $|T|$, pois a cada nível da recursão, o texto S irá encolher com relação ao nível anterior. A análise da complexidade da recursão será feita após a explicação do algoritmo.

3.2.2 Ordenação

Agora que temos S_{12}, o próximo passo é ordenar as triplas de caracteres. Isso pode ser feito em tempo linear em $|S|$ por meio do *RadixSort* [Cormen *et al.*, 2009c]. Trata-se de um algoritmo de *ordenação estável*. Um algoritmo de ordenação é *estável* se preserva a ordem relativa de duas chaves iguais (nesse caso, os inteiros em S_{12}) após o término de sua execução. O *RadixSort* é um algoritmo onde são feitas um número constante de passagens, cada uma usando como chave uma parte dos dados. Nesse caso, serão feitas três passagens, e, em cada uma delas, iremos ordenar estavelmente segundo um caractere de cada tripla. Nesse exemplo, iremos ordenar as triplas, dos caracteres menos significativos para os mais significativos.

Iremos explicar mais alguns detalhes sobre a implementação do RadixSort para esse algoritmo posteriormente, pois nesse caso não estaremos simplesmente "ordenando um vetor" (ou, pelo menos não todos os seus elementos diretamente), pois nenhuma passagem do RadixSort irá ordenar todos os elementos de S, já que queremos ordenar apenas as triplas iniciadas em índices de S_{12}.

Exemplo

A seguir iremos exemplificar como o RadixSort ordena o vetor S_{12} baseado em $\tau(S_{12}[i])$. A cada passagem, ordena-se um inteiro da tripla. A tabela abaixo mostra o vetor S_{12} e as triplas de inteiros em S correspondentes a cada posição do vetor S_{12} original, e após cada passagem do RadixSort. Os caracteres destacados em vermelho são aqueles que foram ordenados pela passagem anterior, e as bordas horizontais delimitam cada conjunto de posições com um determinado caractere.

i	$S_{12}[i]$	$\tau(S_{12}[i])$		$S_{12}[i]$	$\tau(S_{12}[i])$		$S_{12}[i]$	$\tau(S_{12}[i])$		$S_{12}[i]$	$\tau(S_{12}[i])$
0	1	251		10	10**0**		10	1**0**0		11	**0**00
1	4	314		11	00**0**		11	0**0**0		10	**1**00
2	7	125	1	1	25**1**	2	2	5**1**3	3	7	**1**25
3	10	100	⟶	5	14**1**	⟶	4	3**1**4	⟶	5	**1**41
4	2	513		8	25**1**		7	1**2**5		1	**2**51
5	5	141		2	51**3**		5	1**4**1		8	**2**51
6	8	251		4	31**4**		1	2**5**1		4	**3**14
7	11	000		7	12**5**		8	2**5**1		2	**5**13

Tabela 3.3: *Ordenação das triplas de S_{12} pelo RadixSort*

De agora em diante, iremos chamar de S'_{12} o vetor S_{12} após essa ordenação.

Ranks das triplas (obtenção de rank_{12})

Agora iremos analisar as triplas ordenadas de inteiros, e considerar que, em vez de triplas de inteiros, elas podem ser, na verdade, elementos de um alfabeto maior que o do passo atual da recursão. Isso é, um alfabeto Σ' onde cada elemento corresponde a uma das triplas de caracteres de Σ, em vez de apenas um inteiro. A partir dessa consideração, iremos construir o vetor rank_{12}, cujos elementos são os ranks de cada tripla de S'_{12} em Σ'. Isso pode ser feito em tempo linear em $|S|$ com o algoritmo abaixo, que será explicado com mais detalhes.

Algoritmo 11: EvalRank

Dados: texto numérico S, vetor S'_{12} com as triplas ordenadas de S_{12}
Resultado: vetor rank_{12}, inteiro c

```
1  c ← 0
2  para i ← 0 até |S'_12| − 1 faça
3      se i > 0 e τ(S'_12[i]) ≠ τ(S'_12[i − 1]) então
4          c ← c + 1
5      se S'_12[i] mod 3 = 1 então
6          rank_12[⌊S'_12[i]/3⌋] ← c
7      senão
8          rank_12[⌊S'_12[i]/3⌋ + |S_1|] ← c
9  c ← c + 1
10 retorna rank_12, c
```

No algoritmo EvalRank, a variável c conta o número de triplas distintas de S iniciadas com índices em S_{12}. Isso porque, como S'_{12} é uma permutação de S_{12}, e as triplas foram ordenadas pelo RadixSort, todas as triplas iguais estarão em posições consecutivas em S'_{12}, ou seja, basta comparar cada tripla $\tau(S'_{12}[i])$ com a anterior para decidir se ela é repetida ou não (linha 3 do algoritmo). Como pulamos essa comparação para $i = 0$, devemos incrementar c após o laço para contar a primeira tripla (nesse caso, 000). Após o término do laço, a variável c será utilizada apenas para uma otimização que pode ser feita para a chamada recursiva ao DC3; essa otimização será explicada em detalhes no final desse capítulo. O valor c devolvido pelo algoritmo será também $|\Sigma'|$. A comparação de triplas na linha 3 pode ser feita comparando diretamente as posições de S indexadas a partir de $S'_{12}[i]$ e $S'_{12}[i] - 1$.

Já o vetor rank_{12} será, na verdade, uma re-rotulação de S_{12}, onde para cada índice i, o conteúdo da posição $S_{12}[i]$ será substituído por $\text{rank}_{\Sigma'}(\tau(S_{12}[i]))$, ou seja, a ordem de cada tripla no alfabeto novo Σ'.

Exemplo

Para ficar mais fácil de entender o que é o vetor rank_{12}, a tabela abaixo mostra o vetor obtido após o término do algoritmo EvalRank. Nela, S_{12} representa o vetor S_{12} que foi criado no início da etapa 1. O vetor não-ordenado S_{12} não é mais necessário nessa chamada recursiva do algoritmo, mas ele foi colocado na tabela para ajudar a entender o que é o vetor rank_{12}. Além disso, a coluna com o rank de cada tripla em Σ' também pode ser pensada como o valor de c após iterar aquela tripla em EvalRank.

i	$S'_{12}[i]$	$\tau(S'_{12}[i])$	Rank em Σ'	$S_{12}[i]$	$\tau(S_{12}[i])$	$\text{rank}_{12}[i]$
0	11	000	0	1	251	4
1	10	100	1	4	314	5
2	7	125	2	7	125	2
3	5	141	3	10	100	1
4	1	251	4	2	513	6
5	8	251	4	5	141	3
6	4	314	5	8	251	4
7	2	513	6	11	000	0

Tabela 3.4: *Vetor* rank_{12} *obtido por EvalRank*

As três colunas mais à esquerda da tabela contém os mesmos dados obtidos pela ordenação com o RadixSort, e o rank de cada tripla em Σ' é o valor de c após iterar aquela tripla no laço principal na linha 2 do algoritmo EvalRank.

O bloco formado pela segunda até a quarta coluna da tabela nos dá informações sobre o alfabeto das triplas. Iremos considerar que Σ' contém somente as triplas de inteiros que realmente foram ordenadas, desconsiderando as demais triplas que poderíam ser formadas com todos os elementos de Σ, mas que não ocorrem em S. Com isso, obtém-se o seguinte alfabeto:

$$\Sigma' = \{000, 100, 125, 141, 251, 314, 513\}.$$

No próximo bloco da tabela (as três últimas colunas), a borda horizontal serve para explicitar a divisa entre os dados relativos a S_1 e S_2 nos vetores. No início da etapa 1, S_{12} havia sido construído como a concatenação de S_1 e S_2, o que significa que os índices de 0 a $|S_1| - 1$ de S_{12} eram de sufixos de S_1, e os índices de $|S_1|$ até $|S_{12}| - 1$ eram de sufixos de S_2. Cada posição $\text{rank}_{12}[i]$ é, na verdade, o rank da tripla $\tau(S_{12}[i])$ no alfabeto novo Σ' (compare as triplas de caracteres na tabela com seus respectivos ranks nas colunas seguintes). Ou seja, as primeiras $|S_1|$ posições de rank_{12} se referem a S_1, ou seja, os sufixos de S iniciados em posições congruentes a 1 mod 3. Isso porque o algoritmo EvalRank, apesar de sempre inicializar uma posição do vetor rank_{12} com c, decide qual posição inicializar baseado no resto da divisão por 3 de $S'_{12}[i]$, de modo similar à construção de S_{12} a partir de S. O processo para obter o vetor rank_{12} vai na verdade possibilitar que comparemos as triplas de caracteres de S_{12}, pois agora em vez de termos o índice onde cada tripla é iniciada, temos a ordem em que cada uma delas se encontra em Σ'.

3.2.3 Recursão (obtenção de VS_{12})

Agora que temos rank_{12}, podemos considerar que ele é um texto numérico derivado a partir de S, onde cada caractere é, na verdade, uma tripla de caracteres de S. Se construirmos o vetor de sufixos de rank_{12}, seremos capazes de comparar todos os sufixos de S iniciados em índices congruente a 1 e 2 mod 3. Iremos construir $VS_{12} = VS(\text{rank}_{12})$ recursivamente.

i	$S_{12}[i]$	$\tau(S_{12}[i])$	$\text{rank}_{12}[i]$
0	**1**	**251**	**4**
1	4	314	5
2	7	125	2
3	10	100	1
4	2	513	6
5	5	141	3
6	**8**	**251**	**4**
7	11	000	0

Ou seja, é necessário construir o vetor de sufixos do texto numérico $(4, 5, 2, 1, 6, 3, 4, 0)$ recursivamente (vetor rank_{12}); isso porque precisaremos dos ranks dos *sufixos* iniciados em cada posição de S indexada por S_{12} em vez de apenas das triplas que ocorrem no início de cada um desses sufixos, isso para descobrir qual dos sufixos iniciados com "251" ($S[1..]$ e $S[8..]$) tem um rank menor. A tabela abaixo mostra rank_{12} e o vetor de sufixos VS_{12} obtido recursivamente, bem como cada sufixo correspondente no texto numérico S.

i	$S_{12}[i]$	$S[S_{12}[i]..]$	$\text{rank}_{12}[i]$	$\text{VS}_{12}[i]$	$\text{rank}_{12}[\text{VS}_{12}[i]..]$
0	1	**251**31412510	4	7	0
1	4	**314**12510	5	3	16340
2	7	**125**10	2	2	216340
3	10	**10**	1	5	340
4	2	**513**1412510	6	6	**40**
5	5	**141**2510	3	0	**45**216340
6	8	**251**0	4	1	5216340
7	11	**0**	0	4	6340

Na coluna da direita, nas linhas correspondentes aos sufixos 6 e 0 do vetor rank_{12}, foram destacados os quatros (correspondentes ao rank das triplas "251" em Σ'), bem como o próximo elemento de cada sufixo, que é necessário para conseguir diferenciá-los e decidir qual deles é menor em VS_{12}.

Com isso, obtemos $\text{VS}_{12} = \text{VS}(\text{rank}_{12})$, e agora, iremos invertê-lo para criar o vetor SV_{12}, que faz o mesmo papel que o vetor rank_{12}, exceto pelo fato de que, em vez dos ranks das triplas em Σ', o vetor SV_{12} nos dá o rank de todo o sufixo iniciado em cada uma dessas triplas.

i	$S_{12}[i]$	$\text{SV}_{12}[i]$	$S[S_{12}[i]..]$
0	1	5	25131412510
1	4	6	31412510
2	7	2	12510
3	10	1	10
4	2	7	5131412510
5	5	3	1412510
6	8	4	2510
7	11	0	0

Isso conclui a etapa 1 do algoritmo.

3.3 Construção de VS_0

Agora que temos SV_{12}, construir VS_0 é simples. A ideia será construir pares ordenados, onde o primeiro elemento do par é um caractere de S indexado por S_0, e o segundo elemento é o rank do sufixo iniciado imediatamente após esse caractere.

O motivo de isso ser simples é também o motivo de esse algoritmo ser conhecido como *DC3* (*Difference Cover*). Todos os elementos de S_0 são os índices congruentes a 0 módulo 3 de S (i.e. {0, 3, 6, ..., k}, onde k é o maior inteiro divisível por 3 que é estritamente menor que $|S|$). Como S_1 contém os índices congruentes a 1, e temos SV_{12}, sabemos os ranks de todos esses sufixos de S_1, que se iniciam imediatamente após os caracteres indexados por S_0.

i	$S_0[i]$	$S_1[i]$	$S[S_0[i]]$	$S[S_{12}[i]..]$	$(S[S_0[i]], \text{SV12}[i])$
0	0	1	1	25131412510	(1, 5)
1	3	4	1	31412510	(1, 6)
2	6	7	4	12510	(4, 2)
3	9	10	5	10	(5, 1)

Cada sufixo de S_0 pode ser considerado como sendo um par, composto por um caractere indexado por S_0 (quarta coluna da tabela), e o sufixo após esse caractere (quinta coluna da tabela). A representação de cada par está na última coluna da tabela. Apesar de os ranks usados para construir os pares na última coluna serem relativos a S_{12}, a ordem relativa se considerássemos apenas os ranks dos sufixos de S_1 continuaria sendo a mesma, pois os sufixos de S_2 não são usados nessa etapa do algoritmo.

Para de fato implementar essa etapa do algoritmo, não iremos construir esses pares explicitamente. Iremos, em vez disso, criar um vetor de tamanho $|S_0|$, e inicializá-lo com os índices de S_0 ordenados com relação ao sufixo após esses índices.

Isso porque, na nossa implementação do RadixSort precisamos, além do vetor S, cujos caracteres serão lidos, também de um outro vetor que armazena quais índices de S devem ser ordenados pelo algoritmo. Por esse motivo, para fins de implementação dessa etapa do algoritmo, iremos na verdade construir VS_0 a partir de VS_{12}.

Apresentaremos, em seguida, um algoritmo para isso. É importante observar que, dependendo do texto S que estamos ordenando, nem todos os índices de S_0 terão, de fato, um sufixo imediatamente após esses índices. Se $|S| \bmod 3 = 1$, o texto S termina com um caractere de S_0 (que, será necessariamente um 0), e não há um sufixo após esse caractere. Esse caso pode ser tratado verificando o tamanho de S, considerando que esse sufixo vazio teria um rank menor que todos os outros (como, por exemplo, -1), e inicializando a primeira posição de S_0 com $|S| - 1$ antes de ordenar com o RadixSort nesse caso.

Algoritmo 12: ConstróiVS0

Dados: texto S, vetor de sufixos VS_{12} de rank_{12}
Resultado: vetor ordenado VS_0 dos sufixos de S_0

```
1  j ← 0
2  se |S| mod 3 = 1 então
3  |   S_0[0] ← |S| − 1
4  |   j ← 1
5  para i ← 0 até |VS_12| − 1 faça
6  |   se VS_12[i] < |S_1| então
7  |   |   S_0[j] ← 3 × VS_12[i]
8  |   |   j ← j + 1
9  VS_0 ← RadixPass(S, S_0)
10 retorna VS_0
```

Como dito anteriormente, S_0 deve ser o vetor com todos os índices divisíveis por 3 do vetor S. Se esses índices já estiverem ordenados (com relação ao rank do sufixo após o caractere indexado por eles), bastaria ordenar esse vetor S_0 apenas com relação ao caractere $S[S_0[i]]$, para cada índice i de S_0 para terminar de ordenar os pares descritos anteriormente (pois a ordenação do RadixSort é estável).

Na linha 2, o algoritmo verifica se $|S| \bmod 3 = 1$. Esse é o caso em que S termina com um caractere de S_0 e não há um sufixo após o último caractere para ser ordenado. Portanto, iremos simplesmente inicializar a primeira posição de S_0 com $|S| - 1$. Vale que esse índice é divisível por 3, e portanto deve pertencer a S_0, e como o sufixo após esse caractere é vazio ($S[|S|..] = \epsilon$), o índice $|S| - 1$ é o primeiro em S_0.

Caso contrário, para todo índice de S_0, existe um índice de S_1 após ele cujo sufixo já foi ordenado no vetor SV_{12}. Ao iterar o vetor VS_{12} na linha 5, que é o inverso de SV_{12}, estamos percorrendo os sufixos ordenados em ordem crescente, e, na linha 6, a condição $\text{VS}_{12}[i] < |S_1|$ verifica se o sufixo $\text{VS}_{12}[i]$ pertence a S_1. Isso porque, quando construímos o vetor SV_{12}, os ranks dos sufixos de S_1 foram todos colocados nas $|S_1|$ primeiras posições.

Portanto, ao chegar na linha 9, os índices de S_0 já estão ordenados com relação ao sufixo após cada caractere, e basta fazer uma passagem do RadixSort para terminar de ordenar os sufixos iniciados nos índices de S_0, obtendo VS_0.

A tabela a seguir exemplifica os índices que serão obtidos para o vetor S_0 que será ordenado pelo RadixSort no nosso exemplo, bem como alguns dados sobre os sufixos que foram ordenados para facilitar a compreensão do algoritmo.

i	$VS_{12}[i]$	$S_{12}[VS_{12}[i]]$	$S[S_{12}[VS_{12}[i]]..]$	$3 \times VS_{12}[i]$	$S[3 \times VS_{12}[i]..]$
0	7	11	0		
1	**3**	**10**	**10**	9	5·10
2	**2**	**7**	**12510**	6	4·12510
3	5	5	1412510		
4	6	8	2510		
5	**0**	**1**	**25131412510**	0	1·25131412510
6	**1**	**4**	**31412510**	3	1·31412510
7	4	2	5131412510		

Na segunda coluna, temos simplesmente o vetor VS_{12}. Na próxima coluna, estamos explicitando a qual índice em S se refere a cada um dos sufixos ordenados do vetor original S_{12} criado no início da etapa 1, e, em seguida, mostramos quais são esses sufixos, que já foram ordenados pelo vetor VS_{12}.

As linhas que estão destacadas em vermelho indicam quais iterações do laço da linha 5 do algoritmo irão passar pela condição da linha 6 (temos que $|S_1| = 4$), e fica claro que a linha 6 do algoritmo realmente está verificando se um sufixo é iniciado ou não por um índice de S_1, pois todos os índices destacados da terceira coluna têm resto 1 quando divididos por 3.

Também vale que $3 \times VS_{12}[i] = S_{12}[VS_{12}[i]] - 1$, isso é, ao calcular $3 \times VS_{12}[i]$ na linha 7 do algoritmo, obtemos os índices de S_0 que correspondem precisamente à posição do caractere anterior aos sufixos na quarta coluna da tabela. A última coluna exemplifica que os sufixos indexados por S_0 realmente representam o par descrito entre um caractere de S_0 e um sufixo de S_1, sendo que nesses pares, os sufixos de S_1 já estão ordenados, pois foram obtidos diretamente a partir do vetor VS_{12}, faltando apenas ordenar estavelmente o primeiro caractere desses pares.

Após a ordenação estável de S_0, obtém-se o vetor VS_0. Com isso, todos os sufixos de S foram ordenados em dois vetores distintos em tempo linear em $|S|$, sendo que um vetor (VS_0) contém todos os sufixos iniciados a partir de índices múltiplos de 3, enquanto o vetor VS_{12} contém todos os demais sufixos. Para obter $VS(S)$, basta agora juntar ambos os vetores com a função Merge do MergeSort [Sedgewick e Wayne, 2011b].

i	$VS_0[i]$	$T[VS_0[i]..]$
0	0	125131412510
1	3	131412510
2	6	412510
3	9	510

Tabela 3.5: *Vetor* VS_0 *obtido no final da etapa 2 e sufixos correspondentes*

3.4 Merge de VS_0 e VS_{12}

Agora que temos VS_0 e VS_{12}, já ordenamos os sufixos de S em dois vetores distintos. Agora precisamos juntar ambos os vetores, mas primeiro, é necessário ajustar os índices de VS_{12}. Isso porque, enquanto VS_0 contém índices válidos de S_0 (isso é, os índices divisíveis por 3), o vetor VS_{12} foi construído como sendo o vetor de sufixos de $rank_{12}$ – o que significa que ele contém inteiros de 0 a $|S_{12}| - 1$, em vez de índices válidos de S_{12} (ou seja, os índices não-divisíveis por 3).

Para converter os índices, é necessário lembrar que, por se tratar do vetor de sufixos de $rank_{12}$, para cada posição i, se $VS_{12}[i] < |S_1|$, o sufixo $VS_{12}[i]$ pertence a S_1, e, caso contrário, pertence a S_2. Como S_{12} foi originalmente construído dessa forma, e os vetores S_1 e S_2 são crescentes, podemos converter um índice i de VS_{12} para um índice de S_{12} com o algoritmo a seguir.

Algoritmo 13: Conversão para S_{12}

Dados: inteiros i e $|S_1|$
Resultado: índice referente a i em S_{12}

```
1 se i < |S1| então
2 |   retorna 3 × i + 1
3 senão
4 |   retorna 3 × (i − |S1|) + 2
```

De agora em diante, representaremos a aplicação do Algoritmo 13 para um índice i simplesmente por $c_{12}(i)$. A tabela a seguir demonstra as conversões dos índices de VS_{12}.

i	$VS_{12}[i]$	$c_{12}(VS_{12}[i])$	$S[c_{12}(VS_{12}[i])..]$
0	7	11	0
1	3	10	10
2	2	7	12510
3	5	5	1412510
4	6	8	2510
5	0	1	25131412510
6	1	4	31412510
7	4	2	5131412510

A tabela acima demonstra que é possível recuperar os índices originais de S_{12} a partir de VS_{12}, e que os sufixos com esses índices realmente estão ordenados. A tabela a seguir mostra os vetores VS_0 e VS_{12} (com seus índices convertidos para S_{12}), e os respectivos sufixos ordenados.

i	$VS_0[i]$	$S[VS_0[i]..]$
0	0	125131412510
1	3	131412510
2	6	412510
3	9	510
i	$c_{12}(VS_{12}[i])$	$S[c_{12}(VS_{12}[i])..]$
0	11	0
1	10	10
2	7	12510
3	5	1412510
4	8	2510
5	1	25131412510
6	4	31412510
7	2	5131412510

Para garantir que o Merge seja feito em tempo linear em $|S|$, é necessário que cada comparação seja feita em tempo $\mathcal{O}(1)$. Para garantir isso, iremos fazer algo parecido com o processo de construção de VS_0 a partir de VS_{12}. Iremos formar pares (ou triplas) compostas por caracteres de S e ranks de S_1 ou S_2. Cada par ou tripla pode, então, ser comparado em tempo constante.

- Cada sufixo de S_0 pode formar um par composto por um caractere de índice em S_0 e o rank do sufixo de S_1 seguinte a esse caractere.
- Cada sufixo de S_0 pode, além disso, formar uma tripla composta por dois caracteres consecutivos, e o rank do sufixo de S_2 seguinte a eles.
- Cada sufixo de S_1 pode formar um par entre um caractere de S_1 e o rank do sufixo de S_2 seguinte a ele.
- Por fim, cada sufixo de S_2 pode formar uma tripla, composta por dois caracteres consecutivos a partir de um índice de S_2, e o rank do sufixo de S_1 após esse par de caracteres.

Os caracteres de S serão lidos com a função RadixKey (caso não tenha sido acrescentado um par de zeros ao final de S no início da recursão) para que os caracteres após o fim do texto sejam considerados como 0, e iremos considerar que caso um sufixo seja vazio, seu rank será -1. Para saber se um sufixo é vazio ou não, basta verificar se seu índice de início é maior ou igual a $|S|$.

Ou seja, podemos comparar um sufixo de S_0 e um de S_1 comparando um par, enquanto sufixos de S_0 e S_2 podem ser comparados por meio da comparação entre uma tripla.

Na tabela a seguir, mostramos quais pares e triplas são obtidos a partir da descrição acima. No último elemento de cada um deles, a função R é apenas uma notação para representar o rank de um sufixo (obtido a partir do vetor $rank_{12}$).

i	$\mathrm{VS}_0[i]$	Pares de S_0	Triplas de S_0
0	0	(1, R(25131412510))	(1, 2, R(5131412510))
1	3	(1, R(31412510))	(1, 3, R(1412510))
2	6	(4, R(12510))	(4, 1, R(2510))
3	9	(5, R(10))	(5, 1, R(0))
i	$c_{12}(\mathrm{VS}_{12}[i])$	Pares de S_1	Triplas de S_2
0	11		(0, 0, R(ϵ))
1	10	(1, R(0))	
2	7	(1, R(2510))	
3	5		(1, 4, R(12510))
4	8		(2, 5, R(10))
5	1	(2, R(5131412510))	
6	4	(3, R(1412510))	
7	2		(5, 1, R(31412510))

Tabela 3.6: *Pares e triplas*

Os caracteres de cada par ou tripla podem ser obtidos diretamente a partir de S, bastando ler um inteiro (ou dois inteiros consecutivos no caso das triplas) a partir do índice da segunda coluna da tabela. Em todos os pares e triplas, os sufixos necessários para construir o último elemento são aqueles que eram indexados pelo vetor S_{12} no início do algoritmo (exceto pelo sufixo vazio ϵ na tripla iniciada pelo sufixo de índice 11), o que significa que, como eles já foram ordenados nos vetores SV_{12} e VS_{12}, é possível compará-los em tempo $\mathcal{O}(1)$. Agora iremos explicar como obter os pares e triplas a partir dos vetores obtidos até agora e dos índices de cada sufixo (segunda coluna da tabela), independentemente da função R.

A tabela a seguir recapitula qual foi o vetor SV_{12} obtido no final da etapa 1 do algoritmo.

i	$S_{12}[i]$	$\mathrm{SV}_{12}[i]$	$S[S_{12}[i]..]$
0	1	5	25131412510
1	4	6	31412510
2	7	2	12510
3	10	1	10
4	2	7	5131412510
5	5	3	1412510
6	8	4	2510
7	11	0	0

Tabela 3.7: *Ranks dos sufixos de* S_{12}

Algoritmo 14: Par_0

Dados: inteiro i_0, vetores S, VS_0, SV_{12}
Resultado: par de S_0 correspondente ao índice i_0

```
1 c ← VS_0[i_0]
2 se c + 1 < |S| então
3 |   r ← SV_12[c/3]
4 senão
5 |   r ← −1
6 retorna (S[c], r)
```

No algoritmo acima, a variável c da linha 1 será o índice do i_0-ésimo caractere de VS_0 (ou seja, o índice da segunda coluna da tabela para $i = i_0$). Se $c+1 < |S|$, então c não corresponde ao último caractere de S, e, portanto, existe um sufixo não-vazio em S após $S[c]$. Para completar o par, precisamos calcular $R(S[c + 1..])$, isso é, o rank do sufixo de S a partir de $c + 1$. Como $c \in S_0$, então $c + 1 \in S_1$, e como os elementos de S_0 e S_1 crescem de 3 em 3 unidades a cada índice, e são iniciados por 0 e 1 respectivamente, então $c + 1 = S_1[\frac{c}{3}]$, e, portanto, $c + 1 = S_{12}[\frac{c}{3}]$, já que $S_{12} = S_1 \cdot S_2$. Para o exemplo sendo apresentado, basta comparar os sufixos de S apresentados na Tabela 3.7, com os pares de S_0 formados na Tabela 3.6, e constatar que estamos, de fato, construindo os pares com os ranks dos sufixos adequados. Por fim, se $c + 1 \geq |S|$, então o sufixo após c é vazio, e, nesse caso, consideramos que $R(\epsilon) = -1$. Em ambos os casos, o algoritmo simplesmente retorna o par com o caractere e o rank do sufixo.

Algoritmo 15: Tripla_0

Dados: inteiros i_0, $|S_1|$, vetores S, VS_0, SV_{12}
Resultado: tripla de S_0 correspondente ao índice i_0

```
1 c ← VS_0[i_0]
2 se c + 2 < |S| então
3 |   r ← SV_12[c/3 + |S_1|]
4 senão
5 |   r ← −1
6 retorna (S[c], S[c + 1], r)
```

A construção das triplas de S_0 é quase idêntica à dos pares; as únicas diferenças são que é necessário verificar se $c + 2 < |S|$, em vez de $c + 1 < |S|$ para determinar se o sufixo restante é vazio ou não, devemos devolver dois caracteres consecutivos ($S[c]$ e $S[c + 1]$), e o rank do sufixo no final da tripla pertence a S_2. Pelo fato de pertencer a S_2, devemos somar $|S_1|$ ao índice que deve ser acessado do vetor SV_{12} para pular os ranks dos sufixos de S_1. Para o exemplo, é possível comparar os sufixos de S_2 (o último bloco da Tabela 3.7) com os sufixos das triplas de S_0 na Tabela 3.6, e verificar que eles são iguais.

Algoritmo 16: Par_1

Dados: inteiros $j = \text{VS}_{12}[i_{12}]$, $|S_1|$, vetores S, SV_{12}
Resultado: par de S_1 correspondente ao índice j

```
1 c ← 3 × j + 1                    // c ← c12(j) (conversão para índice de S12)
2 se c + 1 < |S| então
3 |   r ← SV12[j + |S1|]
4 senão
5 |   r ← −1
6 retorna (S[c], r)
```

No Algoritmo 16, o inteiro j é um elemento de VS_{12} que corresponde a um sufixo de S_1 (ou seja, a pré-condição para o algoritmo é $j < |S_1|$). Na linha 1, a conversão de j para c serve para que possamos indexar S diretamente; isso é, j indexa um sufixo de S_1 em SV_{12}, enquanto c indexa o caractere $3 \times j + 1$ de S. O caractere do par será $S[c]$, e o sufixo seguinte deverá ser um sufixo de S_2. Para obter o rank desse sufixo, basta lembrar que como $S_{12} = S_1 \cdot S_2$, e $j < |S_1|$, então $\text{SV}_{12}(j + |S_1|)$ será o rank do sufixo seguinte a $S[c]$ em S_2.

As Tabelas 3.8 e 3.9 a seguir exemplificam a função Par_1 para $j = 1$. Nesse exemplo, vale que $c = c_{12}(j) = 4$, e que $|S_1| = 4$, portanto temos que $r = \text{SV}_{12}[5]$. Por fim, conclui-se que $\text{R}(1412510) = r = 3$, e o par que deve ser formado é $(S[c], r) = (S[4], 3) = (3, 3)$.

i	$S_{12}[i]$	$\text{SV}_{12}[i]$	$S[S_{12}[i]..]$
0	1	5	25131412510
$j = 1$	$c = 4$	6	31412510
2	7	2	12510
3	10	1	10
4	2	7	5131412510
$j + \|S_1\| = 5$	5	**3**	**1412510**
6	8	4	2510
7	11	0	0

Tabela 3.8: *Exemplo para $j = 1$ e $c = 4$*

i	$c_{12}(\text{VS}_{12}[i])$	Pares de S_1	Triplas de S_2
0	11		(0, 0, R(ϵ))
1	10	(1, R(0))	
2	7	(1, R(2510))	
3	5		(1, 4, R(12510))
4	8		(2, 5, R(10))
5	1	(2, R(5131412510))	
6	$c = 4$	(3, R(**141251$**))	
7	2		(5, 1, R(31412510))

Tabela 3.9: *Exemplo para $j = 1$ e $c = 4$*

Algoritmo 17: Tripla_2

Dados: inteiros $j = \text{VS}_{12}[i_{12}]$, $|S_1|$, vetores S, SV_{12}
Resultado: tripla de S_2 correspondente ao índice j

```
1 c ← 3 × (j − |S_1|) + 2          // c ← c_12(j) (conversão para índice de S_12)
2 se c + 2 < |S| então
3 |   r ← SV_12[j − |S_1| + 1]
4 senão
5 |   r ← −1
6 retorna (S[c], S[c + 1], r)
```

Já o Algoritmo Tripla_2 é similar ao Par_1, exceto que a pré-condição é que $j \geq |S_1|$ e é devolvida uma tripla (ou seja, na linha 2, é necessário verificar se $c + 2 < |S|$ em vez de $c + 1 < |S|$, já que serão lidos dois caracteres de S). Em ambos os algoritmos (Tripla_2 e Par_1), a conversão de j para c na linha 1 é, na verdade, $c \longleftarrow c_{12}(j)$ com o Algoritmo de Conversão para S_{12}, descrito no início dessa seção, mas devido às pré-condições desses algoritmos, é possível pular a verificação se j pertence a S_1 ou S_2. Na linha 3 do Algoritmo Tripla_2, estamos fazendo quase o inverso do que foi feito na linha 3 de Par_1, em que o índice j é deslocado de $|S_1|$ posições em SV_{12}, para alternar entre S_1 e S_2; a diferença é que aqui, apenas deslocar j de $|S_1|$ posições retornaria o rank do sufixo a partir de $c - 1$. Portanto, ao acrescentar 1 no índice de acesso ao rank, estamos obtendo o rank de $S[c - 1 + 3..] = S[c + 2..]$, que é o sufixo necessário para formar a tripla corretamente a partir do índice c.

Com isso, podemos agora, de fato, construir os pares e triplas necessários. A tabela a seguir mostra os pares e triplas obtidos após a execução desses algoritmos.

i	$\text{VS}_0[i]$	Pares de S_0	Triplas de S_0	Pares de S_0	Triplas de S_0
0	0	(1, R(25131412510))	(1, 2, R(5131412510))	(1, 5)	(1, 2, 7)
1	3	(1, R(31412510))	(1, 3, R(1412510))	(1, 6)	(1, 3, 3)
2	6	(4, R(12510))	(4, 1, R(2510))	(4, 2)	(4, 1, 4)
3	9	(5, R(10))	(5, 1, R(0))	(5, 1)	(5, 1, 0)
i	$c_{12}(\text{VS}_{12}[i])$	Pares de S_1	Triplas de S_2	Pares de S_1	Triplas de S_2
0	11		(0, 0, R(ϵ))		(0, 0, -1)
1	10	(1, R(0))		(1, 0)	
2	7	(1, R(2510))		(1, 4)	
3	5		(1, 4, R(12510))		(1, 4, 2)
4	8		(2, 5, R(10))		(2, 5, 1)
5	1	(2, R(5131412510))		(2, 7)	
6	4	(3, R(1412510))		(3, 3)	
7	2		(5, 1, R(31412510))		(5, 1, 6)

Tabela 3.10: *Pares e triplas construídos para fazer o Merge*

Cada par e cada tripla é construído em tempo $\mathcal{O}(1)$, e, além disso, dois pares ou duas triplas podem ser comparados em tempo $\mathcal{O}(1)$, pois apenas é necessário compará-los elemento-por-elemento até decidir qual dos dois é menor. Como todos os sufixos usados para construir os pares e triplas vieram de S_{12}, eles já haviam sido completamente ordenados na etapa 1, o que significa que todos eles têm ranks distintos. Como todos os pares têm o rank de um sufixo distinto como último elemento, isso significa que não é possível haver dois pares iguais, tampouco duas triplas iguais pelo mesmo motivo.

Algoritmo 18: Merge

Dados: vetores S, VS_0, VS_{12}, SV_{12}
Resultado: $\mathrm{VS}(S)$

```
1  i_0, i_12, k ← 0
2  enquanto k < |S| faça
3      j ← VS_12[i_12]
4      se j < |S_1| então
5          p_1 ← Par_1(j, |S_1|, S, SV_12)
6          p_0 ← Par_0(i_0, S, VS_0, SV_12)
7          m_0 ← (p_0 <? p_1)
8      senão
9          t_2 ← Tripla_2(j, |S_1|, S, SV_12)
10         t_0 ← Tripla_0(i_0, |S_1|, S, VS_0, SV_12)
11         m_0 ← (t_0 <? t_2)
12     se m_0 = verdadeiro então
13         VS(S)[k] ← VS_0[i_0]
14         k ← k + 1
15         i_0 ← i_0 + 1
16         se i_0 = |S_0| então
17             enquanto i_12 < |S_12| faça
18                 VS(S)[k] ← c_12(VS_12[i_12])
19                 k ← k + 1
20                 i_12 ← i_12 + 1
21     senão
22         VS(S)[k] ← c_12(VS_12[i_12])
23         k ← k + 1
24         i_12 ← i_12 + 1
25         se i_12 = |S_12| então
26             enquanto i_0 < |S_0| faça
27                 VS(S)[k] ← VS_0[i_0]
28                 k ← k + 1
29                 i_0 ← i_0 + 1
30 retorna VS(S)
```

Durante o Algoritmo Merge, i_0 e i_{12} são os próximos índices de VS_0 e VS_{12} que devem ser comparados, e $k = i_0 + i_{12}$ é a próxima posição de VS(S) que será inicializada. Na linha 3, é necessário primeiro ler um elemento de VS_{12} para decidir como prosseguir, pois se $j < |S_1|$, então devemos comparar um sufixo de S_1 com um de S_0, ou seja, é necessário comparar dois pares. Caso contrário, iremos comparar um sufixo de S_2 com um de S_0 por meio da comparação de duas triplas. Em ambos os casos, construímos os pares ou triplas que devem ser comparados, e, nas linhas 7 e 11, o operador $x <_? y$ devolve verdadeiro caso $x < y$, ou falso caso $x > y$ (lembrando que não é possível ocorrer igualdade pelo motivo comentado antes da apresentação do Algoritmo Merge). As linhas 7 e 11 fazem a comparação direta entre os pares ou triplas, elemento por elemento, até que seja possível saber qual dos dois é menor. Isso é feito em tempo $\mathcal{O}(1)$. Já a variável m_0 serve para lembrar se o menor sufixo é o de S_0 ou de S_{12} na hora de copiar um elemento para VS(S). Isso é, m_0 é verdadeiro se, e somente se, $S[VS_0[i_0]..] < S[c_{12}(VS_{12}[i_{12}])..]$. Após a inicialização de m_0, podemos inicializar um elemento de VS(S). Se m_0 for verdadeiro, iremos copiar o próximo índice de S_0 (ou seja, $VS_0[i_0]$) para VS(S)[k], incrementar os índices k e i_0, e, em seguida, caso $i_0 = |S_0|$, significa que já copiamos todos os sufixos de S_0, e basta agora terminar de copiar VS_{12} para VS(S), o que faz com que o algoritmo saia do laço principal da linha 2, e retorne VS(S). O caso em que m_0 é falso é análogo a m_0 ser verdadeiro, apenas invertendo qual vetor é copiado para VS(S).

Com isso, obtém-se VS(S), e, caso estejamos no primeiro nível da recursão, valerá que VS(S) = VS(T), pois S será apenas uma re-rotulação de T, mantendo a ordem relativa de cada caractere. O Algoritmo Merge é executado em tempo $\mathcal{O}(|S|)$, pois vale que $i_0 + i_{12} = k < |S|$, e, como cada iteração de um laço, seja o laço externo da linha 2, ou um dos laços internos das linhas 17 e 26, sempre incrementa i_0 ou i_{12} e k, e nenhum desses índices diminui ao longo da execução do algoritmo, então o algoritmo é executado em tempo $\mathcal{O}(|S|)$, já que todas as demais operações de cada iteração do laço da linha 2 são feitas em tempo $\mathcal{O}(1)$.

A etapa 2, bem como toda a parte não-recursiva da etapa 1 do Algoritmo DC3, também foi executada em tempo $\mathcal{O}(|S|)$. A cada chamada recursiva na etapa 1, o comprimento do texto numérico S é reduzido a dois terços de seu comprimento na chamada anterior até alcançarmos o caso base da recursão. Sejam n o comprimento do texto original T, e $T(n)$ o consumo de tempo da execução do algoritmo DC3 para um texto numérico de comprimento n.

O consumo de tempo da primeira chamada recursiva será da forma $T(n) = T(\frac{2n}{3}) + \mathcal{O}(n)$, com $T(1) = T(2) = 1$. Essa recorrência pode ser resolvida pelo Teorema Mestre [Cormen *et al.*, 2009b], o que nos permite concluir que o consumo de tempo do Algoritmo DC3 é $\mathcal{O}(n)$. Ou seja, podemos construir VS(T) em tempo linear em $|T|$.

3.5 Alguns detalhes sobre a implementação

Essa seção irá explicar alguns detalhes sobre o algoritmo que não foram mencionados anteriormente por serem mais relevantes com relação à implementação do Algoritmo DC3 do que com relação à explicação do algoritmo.

3.5.1 Sobre a recursão

Durante a explicação do Algoritmo DC3, foi dito que é necessário usar a recursão para obter o vetor VS_{12}, porém em alguns casos é possível construir esse vetor diretamente, sem que seja necessária a recursão. No final da etapa 1, foi apresentado o Algoritmo EvalRank, que construía o vetor $rank_{12}$ a partir das triplas ordenadas de S_{12}. Esse algoritmo também devolve um inteiro c, que é o número de triplas distintas (e, portanto, $c = |\Sigma'|$). Porém, caso $c = |rank_{12}|$, isso significa que todas as triplas de caracteres são distintas, e, portanto, só as triplas já são o suficiente para diferenciar todos os sufixos de S_{12}, não sendo necessária a chamada recursiva.

Portanto, em vez de chamar diretamente a recursão para obter VS_{12}, e inverter o vetor para construir SV_{12}, podemos usar o Algoritmo EstendeRank12, descrito abaixo, para chamar a recursão só quando for realmente necessário.

Algoritmo 19: EstendeRank12

Dados: vetor $rank_{12}$, inteiro c
Resultado: vetores SV_{12} e $VS_{12} = VS(rank_{12})$

```
1 se c < |rank12| então
2 |   VS12 ← DC3SkewAlg(rank12)
3 |   para i ← 0 até |rank12| − 1 faça
4 |   |   SV[VS12[i]] ← i
5 senão
6 |   para i ← 0 até |rank12| − 1 faça
7 |   |   VS12[rank12[i]] ← i
8 retorna SV12, VS12
```

Se $c < |rank_{12}|$, então havia triplas de caracteres repetidas em S_{12}, o que significa que há repetições no vetor $rank_{12}$. Nesse caso, é necessário obter o vetor de sufixos recursivamente, e invertê-lo para obter SV_{12}.

Caso contrário ($c = |rank_{12}|$), então não há repetições em $rank_{12}$, e apenas os primeiros três caracteres já são suficientes para ordenar todos os sufixos, portanto, basta inverter $rank_{12}$ para obter VS_{12}.

Uma otimização que pode ser feita é usar o mesmo vetor para armazenar $rank_{12}$ e SV_{12} ao implementar o DC3, já que em nenhum momento é necessário ter ambos os vetores simultaneamente. O Algoritmo EstendeRank12 supõe que essa otimização será feita, já que caso $c = |rank_{12}|$, ele não inicializa SV_{12} (pois nesse caso, ambos os vetores $rank_{12}$ e SV_{12} seriam iguais). Caso essa otimização não seja feita, também seria necessário copiar $rank_{12}$ para SV_{12} no laço da linha 7 do Algoritmo 19.

3.5.2 Sobre o texto numérico S

Ao converter o texto original T para o texto numérico S antes do primeiro nível da recursão, é necessário determinar qual o rank de cada caractere em T no alfabeto Σ considerando somente os caracteres que ocorrem em T. Isso pode ser feito com o algoritmo descrito abaixo.

Algoritmo 20: CriaTextoNumérico

Dados: texto T
Resultado: texto numérico S, inteiro r com o tamanho do alfabeto restrito a T

```
1  para i ← 0 até |Σ| − 1 faça
2  |   r_T[i] ← −1
3  para i ← 0 até |T| − 1 faça
4  |   r_T[T[i]] ← 0 // Inicializa com 0 os caracteres que ocorrem em T
5  r ← 0
6  para i ← 0 até |Σ| − 1 faça
7  |   se r_T[i] ≠ −1 então
8  |   |   r_t[i] ← r
9  |   |   r ← r + 1
10 para i ← 0 até |T| − 1 faça
11 |   S[i] ← r_T[T[i]]
12 retorna S, r
```

No laço da linha 1, os elementos do vetor r_T são inicializados com -1, e, na linha 3, os elementos correspondentes aos caracteres que ocorrem em T são inicializados com 0. Já no laço da linha 6, vale o invariante que r é o rank atual do caractere i no alfabeto restrito a T. Nesse laço, iremos simplesmente atribuir o rank atual r a cada um dos caracteres em r_T que ocorrem em T e incrementar r para a iteração seguinte. Por fim, o laço da linha 10 inicializa S com os ranks de cada caractere de T nesse alfabeto. O motivo de o algoritmo também devolver r será explicado em breve, mas vale que o valor r devolvido pelo algoritmo será o tamanho do alfabeto restrito a T.

O motivo de usarmos um texto numérico em vez de um texto com caracteres é que, a cada chamada recursiva, o tamanho do alfabeto de S pode ser maior que no da chamada anterior, isso porque o alfabeto passa a ser de triplas de inteiros do alfabeto anterior. Se usássemos caracteres (de, por exemplo, 1 byte cada) para representar os elementos de S ao implementar o algoritmo, se houvesse mais do que $2^8 = 256$ triplas distintas de caracteres em S, ocorreria um *overflow* na chamada recursiva seguinte, e não obteríamos o vetor de sufixos correto no final do algoritmo.

Caso seja optado por acrescentar dois zeros ao final de S em vez de acessar S com a função RadixKey, isso deve ser feito ao alocar o vetor rank_{12} em cada chamada recursiva. Isso é, devemos alocar espaço para os dois últimos zeros ao criar o vetor rank_{12} (ou o primeiro vetor S antes da primeira chamada recursiva), e inicializar as duas últimas posições com zeros *antes* da próxima chamada recursiva. Isso porque em cada chamada recursiva, esse vetor irá fazer o papel de S na próxima chamada (pois estaremos calculando $\text{VS}(\text{rank}_{12})$). Nesse caso, também é necessário passar o comprimento de S **sem** os dois zeros explicitamente

como parâmetro para o DC3, e todas as referências a $|S|$ ao longo do algoritmo devem usar esse parâmetro em vez do verdadeiro comprimento de S.

Se tivéssemos, por exemplo 5 triplas com ranks $3, 1, 0, 3, 2$ respectivamente, o vetor rank_{12} alocado antes da recursão deveria ser $\text{rank}_{12} = (3, 1, 0, 3, 2, 0, 0)$. Então nesse caso, seria passado $S = \text{rank}_{12}$ e $|S| = 5$ para a próxima chamada recursiva do Algoritmo DC3; isso porque o algoritmo não deve considerar que o par de zeros realmente faz parte de S, ele serve apenas para que possamos ler até dois índices após o final de S como 0.

Além disso, dado o parâmetro $|S|$, podemos calcular $|S_0|$, $|S_1|$, $|S_2|$ e $|S_{12}|$ diretamente a partir de $|S|$:

$$|S_0| = \left\lfloor \frac{|S| + 2}{3} \right\rfloor, |S_1| = \left\lfloor \frac{|S| + 1}{3} \right\rfloor, |S_2| = \left\lfloor \frac{|S|}{3} \right\rfloor, |S_{12}| = |S_1| + |S_2| = \left\lfloor \frac{2|S|}{3} \right\rfloor.$$

3.5.3 Sobre o RadixSort

Ao implementar o RadixSort para esse algoritmo, precisaremos armazenar de alguma forma as triplas de caracteres e ordená-las mas sem alterar o texto S. Para isso, cada passagem do nosso RadixSort receberá três vetores de inteiros como parâmetro, sendo um deles o texto S; um deles, que será chamado de f, contém os índices de início de cada tripla que deve ser ordenada (isso é, cada tripla a ser ordenada tem início na posição $f[i]$ em S). O outro vetor, que será chamado de t irá armazenar os resultados da ordenação. Também receberemos $|\Sigma|$ e um inteiro $\delta \in \{0, 1, 2\}$ como parâmetro, que indica qual dos três caracteres da tripla será ordenado.

Ou seja, cada passagem do RadixSort irá ordenar o vetor f, armazenando o resultado no vetor t, e o critério a ser usado pela ordenação será o valor do inteiro em $S[f[i] + \delta]$ para cada índice i de f, e será usado um vetor temporário de tamanho $|\Sigma|$ durante esse processo. Pelo fato de ser necessário conhecer $|\Sigma|$ previamente para que possamos alocar um vetor com tamanho adequado para cada passagem do RadixSort, cada chamada ao DC3 deve também receber como parâmetro o inteiro $|\Sigma|$. Para a primeira chamada, esse será o inteiro r devolvido pelo Algoritmo CriaTextoNumérico (Algoritmo 20), descrito na seção anterior, e já para as chamadas recursivas, será o inteiro c, que foi devolvido pelo Algoritmo EvalRank (Algoritmo 11), descrito no final da etapa 1 do DC3.

Algoritmo 21: RadixPass

```
   Dados: vetores S, f, t, inteiros |Σ|, δ
 1 f_p ← vetor de zeros com comprimento |Σ|
 2 para i ← 0 até |f| − 1 faça
 3 |   j ← S[f[i] + δ]
 4 |   f_p[j] ← f_p[j] + 1          // Conta a frequência de cada caractere j
 5 s ← 0
 6 para i ← 0 até |Σ| − 1 faça
 7 |   c ← f_p[i]
 8 |   f_p[i] ← s  // Calcula a frequência acumulada de cada caractere i
 9 |   s = s + c
10 para i ← 0 até |f| − 1 faça
11 |   j ← S[f[i] + δ]
12 |   t[f_p[j]] ← f[i]                                  // Ordena f para t
13 |   f_p[j] ← f_p[j] + 1
14 apaga f_p
```

Cada chamada à função RadixPass consome tempo $\mathcal{O}(|f| + |\Sigma|)$. Como durante o Algoritmo DC3, o parâmetro correspondente a $|\Sigma|$ é sempre menor que $|S|$ para cada chamada recursiva pois é considerado somente o alfabeto das triplas que realmente ocorrem em S. Durante a etapa 1 do algoritmo, $|f|$ será $|S_{12}|$, e durante a etapa 2 do algoritmo, $|f|$ será $|S_0|$, então todas as ordenações de cada chamada recursiva do DC3 terão consumo de tempo proporcional a $|S|$. Cabe observar que, caso não tenha sido acrescentado um par de zeros ao final de S, os acessos a S nas linhas 3 e 11 do RadixPass deverão usar a função RadixKey em vez de acessar S diretamente.

O laço da linha 2 simplesmente conta o número de ocorrências (a frequência) de cada caractere j que deve ser ordenado, e, em seguida, o laço da linha 6 sobrescreve o vetor f_p para armazenar a frequência acumulada de cada um deles.

Após a atribuição da linha 8, se $f_p[i] = s$, então todos os caracteres de 0 a $i-1$ de Σ terão uma frequência acumulada de s. Isso é importante pois significa que após ordenarmos o vetor, a posição $t[s]$ será a primeira ocorrência desse caractere i, já que todos os caracteres antes dele têm uma frequência acumulada de s. Essa observação serve para entender o último laço, na linha 10. Nele, estamos usando o vetor f_p como um vetor de índices onde devemos escrever cada caractere que será lido, e, em seguida, ao incrementar $f_p[j]$, estamos atualizando esse índice para refletir a escrita que acabou de ser feita em t, evitando escrever duas vezes na mesma posição de t.

Exemplo

Voltando para o exemplo com T = “abracadabra\$” durante a ordenação das triplas na etapa 1, a tabela a seguir mostra quais triplas são ordenadas naquele exemplo. Os caracteres destacados em vermelho são aqueles que devem ser ordenados na primeira passagem do RadixSort (onde vale que $\delta = 2$).

i	$S_{12}[i]$	$\tau(S_{12}[i])$
0	1	25**1**
1	4	31**4**
2	7	12**5**
3	10	10**0**
4	2	51**3**
5	5	14**1**
6	8	25**1**
7	11	00**0**

Tabela 3.11: *Para $f = S_{12}$ e $\delta = 2$, o algoritmo irá ordenar o último caractere de cada tripla de S_{12}*

Na tabela a seguir, $\overline{f_p}[i]$ representa o estado do vetor f_p após o laço da linha 2, enquanto $f_p[i]$ representa seu estado após o laço da linha 6, com as frequências acumuladas de cada caractere. Vale que $f_p[i] = \Sigma_{k=0}^{i-1}\overline{f_p}[k]$, ou ainda $f_p[i+1] = \overline{f_p}[i] + f_p[i]$.

i	$\overline{f_p}[i]$	$f_p[i]$
0 (“$”)	2	0
1 (“a”)	3	2
2 (“b”)	0	5
3 (“c”)	1	5
4 (“d”)	1	6
5 (“r”)	1	7

Tabela 3.12: *Frequências de cada caractere de S*

Com isso, é fácil ordenar os caracteres, já que $f_p[i]$ contém o índice no qual o próximo caractere i deve ficar em t. Por exemplo, para $i = 1$, temos que o primeiro caractere “a” deve ficar na posição $f_p[1] = 2$, pois há dois caracteres menores (ambos “$”) que são menores que “a”. Já o primeiro (e único) caractere “c” deve ficar em $t[5]$, pois $f_p[3] = 5$; isso porque há cinco caracteres menores que “c” sendo ordenados. O fato de incrementarmos $f_p[i]$ após escrever o caractere i em t mantém esse invariante sobre a próxima posição onde cada caractere deve ser escrito, e também faz com que a ordenação seja estável, pois os índices de ocorrência dos primeiros caracteres em f são inicializados primeiro em t, mantendo a ordem relativa inicial de caracteres iguais.

Usando RadixPass no DC3

Agora que sabemos como o RadixPass pode ser implementado, o pseudocódigo a seguir pode ser usado para implementar o RadixSort para ordenar as triplas na etapa 1 do DC3. O vetor S_{12} criado no início da etapa 1 só é usado diretamente para ordenar as triplas de caracteres com o RadixSort. Ele não é usado diretamente pelo DC3 após isso, e, portanto, pode ser sobrescrito pelo RadixSort durante a ordenação. Também vale observar que o vetor S'_{12} devolvido pela ordenação pode, na verdade, ser o mesmo vetor que VS_{12}, isso é, é possível reutilizar S'_{12} para armazenar VS_{12} posteriormente, economizando a memória que seria necessária para armazená-los separadamente.

Algoritmo 22: RadixSort

Dados: vetores S e S_{12}, inteiro $|\Sigma|$
Resultado: vetor S'_{12} com os índices das triplas ordenadas

```
1 S'_12 ← vetor de comprimento |S_12|
2 RadixPass(S, S_12, S'_12, |Σ|, 2)      // Ordena o terceiro caractere da tripla
3 RadixPass(S, S'_12, S_12, |Σ|, 1)       // Ordena o segundo caractere da tripla
4 RadixPass(S, S_12, S'_12, |Σ|, 0)      // Ordena o primeiro caractere da tripla
5 retorna S'_12
```

Por fim, o algoritmo a seguir é uma adaptação da construção de VS_0 apresentada na etapa 2 do Algoritmo DC3, levando em consideração a explicação mais detalhada do RadixPass que foi apresentada agora. A única diferença é que o vetor VS_0 é alocado antes da passagem do Radix e é passado como parâmetro para ela. O parâmetro $\delta = 0$ refere-se ao fato de que S_0 deve ser ordenado com base no índices em S_0 sem somar δ a nenhum desses índices para ler os caracteres de S. Isso porque só precisamos ler o primeiro caractere dos pares, e esse caso corresponde a $\delta = 0$.

Algoritmo 23: ConstróiVS0Implementação

Dados: texto S, vetor de sufixos VS_{12} de rank_{12}, inteiro $|\Sigma|$
Resultado: vetor ordenado VS_0 dos sufixos de S_0

```
1  j ← 0
2  se |S| mod 3 = 1 então
3  |   S_0[0] ← |S| − 1
4  |   j ← 1
5  para i ← 0 até |VS_12| − 1 faça
6  |   se VS_12[i] < |S_1| então
7  |   |   S_0[j] ← 3 × VS_12[i]
8  |   |   j ← j + 1
9  VS_0 ← vetor de comprimento |S_0|
10 RadixPass(S, S_0, VS_0, |Σ|, 0)
11 retorna VS_0
```

Com isso terminamos de cobrir o DC3/Skew Algorithm para a construção de $\mathrm{VS}(T)$ em tempo linear em $|T|$. Isso quer dizer que agora podemos concluir todo o pré-processamento em tempo $\mathcal{O}(|T|)$, pois é possível construir $\mathrm{VS}(T)$ a partir de T, $\mathrm{LCP}(T)$ a partir de $\mathrm{VS}(T)$, e obter os LCPs estendidos a partir de $\mathrm{LCP}(T)$, sendo que todos esses passos do pré-processamento são feitos em tempo linear em $|T|$.

O capítulo seguinte irá cobrir as árvores de sufixos, que podem ser usadas como alternativa ao vetor de sufixos para realizar as buscas.

Capítulo 4

Árvore de Sufixos

Uma outra estrutura de dados muito conhecida que pode ser usada para dar suporte a buscas de padrões num texto fixo é a chamada árvore de sufixos. Elas foram propostas por Peter Weiner [Weiner, 1973]. A árvore de sufixos do texto T é denotada por AS(T).

Nesse capítulo será explicado como obter a árvore de sufixos de um texto T a partir de VS(T) e LCP(T) em tempo linear em $|T|$, e também como obter esses vetores a partir da AS, também em tempo linear. É possível obter AS(T) em tempo linear em $|T|$ diretamente sem precisar desses vetores, por meio do Algoritmo de Ukkonen [Ukkonen, 1995], mas não iremos abordá-lo.

4.1 Definição

Uma árvore de sufixos é uma árvore enraizada. Cada aresta corresponde a um intervalo do texto, e cada nó tem um vetor de filhos de tamanho $|\Sigma|$, onde Σ é o alfabeto do texto. Exceto pela raiz, cada nó é indexado pelo primeiro caractere do intervalo representado pela aresta entre ele e seu nó pai, portanto, para cada letra de Σ, deve haver no máximo um filho cujo rótulo da aresta com o pai se inicia com essa letra. Cada folha corresponde ao índice de um sufixo de T (e todo sufixo de T é representado por exatamente uma folha), sendo que esse sufixo é a concatenação dos intervalos das arestas no caminho da raiz até essa folha.

Em vez do vetor de tamanho $|\Sigma|$ para cada nó, para indicar os filhos do nó, pode-se usar uma árvore binária de busca com a primeira letra do rótulo da aresta para o filho como chave. O vetor consome espaço $\Theta(|\Sigma|)$ por nó, mas dá acesso a cada filho em tempo $\mathcal{O}(1)$. Já a árvore consome espaço proporcional ao número k de filhos do nó e leva $\mathcal{O}(\log(k))$ para acessar um filho, caso a árvore seja balanceada.

Sejam r a raiz da AS, e $C(n)$ a concatenação dos intervalos das arestas no caminho de r até n para cada nó n de AS(T) (em particular, $C(r) = \epsilon$). Para cada nó n de AS(T) que não é uma folha, todos (e somente) os sufixos de T que têm $C(n)$ como prefixo são representados por folhas que estão na sub-árvore enraizada em n (ou seja, folhas que estão "abaixo" de n em AS(T)).

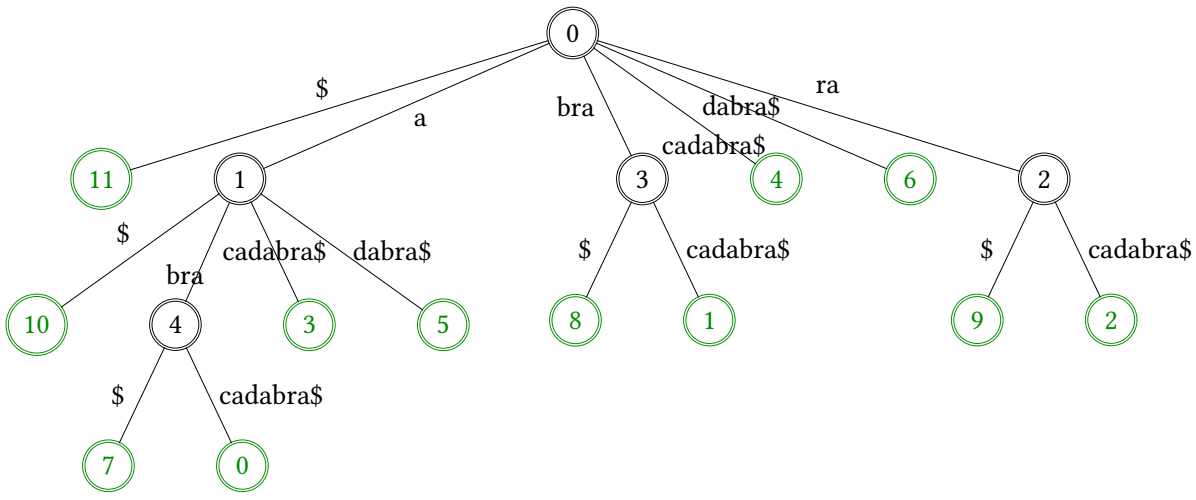
As definições até agora já são suficientes para caracterizar uma estrutura de dados que permita fazer as buscas no texto, entretanto, para que essa busca seja de fato eficiente, é necessário também que a árvore seja comprimida. Isso é, ela deve ter o menor número possível de nós. Efetivamente, isso significa que, se $T \neq$ "$", então não existe nenhum nó em AS(T) com exatamente um nó filho, já que nesse caso seria possível eliminar esse nó, e criar uma aresta direta entre seu pai e seu filho com o intervalo correspondente à concatenação dos intervalos das arestas entre eles. Caso $T =$ "$", então a AS teria apenas a raiz e uma folha representando o sufixo "$". Como a árvore tem $|T|$ folhas e cada nó interno tem pelo menos dois filhos, então a árvore terá menos que $2|T|$ nós ao todo.

Cada nó n da árvore tem um apontador para o seu nó pai e dois inteiros início e fim, de modo que $T[n.\text{início}..n.\text{fim}]$ é o rótulo da aresta entre n e n.pai. Isso permite armazenar o rótulo de cada aresta em espaço $\mathcal{O}(1)$ em seu nó filho, pois a representação do rótulo independe do comprimento do intervalo. Esses campos não serão usados para a raiz da árvore pois não há um nó acima dela. Se n é um nó interno, na nossa implementação, ele também terá um vetor de tamanho $|\Sigma|$ para armazenar seus filhos, e dois contadores, sendo um para o número de nós filhos de n, e outro para o número de folhas na sub-árvore enraizada em n. Cada folha f da árvore tem também um campo suf, que é o índice do sufixo associado a f.

Caso seja usado um vetor de tamanho $|\Sigma|$ para armazenar os filhos de cada nó (como é feito aqui), o espaço consumido por um nó é $\Theta(|\Sigma|)$, o que significa que o espaço consumido pela árvore inteira é $\Theta(|\Sigma| \times |T|)$. Caso seja usada uma árvore binária para armazenar os filhos de um nó, cada nó irá ocupar espaço apenas para armazenar os filhos que não são nulos. Como na árvore há menos que $2|T|$ nós ao todo, então o espaço ocupado pela árvore inteira seria $\Theta(|T|)$, já que o número de filhos não nulos na árvore corresponde ao número de arestas nela. Nesse caso, cada nó ocuparia espaço amortizado $\mathcal{O}(1)$.

Exemplo

Novamente usando $T =$ "abracadabra$", AS($T$) tem a seguinte estrutura:



O número em cada folha f é o campo f.suf, que é o índice de início do sufixo $C(f)$. Já o número em cada nó interno n da árvore é $|C(n)|$, ou seja o número de caracteres soletrados

até n a partir da raiz. Também é possível ver que, se a árvore for percorrida visitando os filhos de cada nó em ordem alfabética no rótulo entre eles, a ordem dos sufixos das folhas é a mesma que no vetor de sufixos.

Como a árvore tem menos de $2|T|$ nós, é possível obter VS(T) em tempo linear em $|T|$ a partir da árvore percorrendo-a nesta ordem, e escrevendo os índices de cada folha. O LCP também pode ser obtido em tempo linear se cada nó interno n armazenar $|C(n)|$ (como ocorre na figura). Para cada sufixo de T e seu predecessor, cuja folha é a anterior na ordem, se a é o menor ancestral comum entre as folhas que representam esses sufixos, o LCP entre eles será $|C(a)|$.

4.2 Busca por uma palavra

A busca com árvore de sufixos é bem mais simples do que com o VS. Em vez de buscar um predecessor, dessa vez iremos tentar buscar um certo nó da árvore (que iremos chamar de n) que pode ou não existir. Mais precisamente, n deve ser o nó menos fundo a partir da raiz, tal que $C(n)$ contém P. O algoritmo para buscar por n está descrito a seguir. Nos comentários sobre o algoritmo, denotamos a concatenação de palavras pelo símbolo "·", ou seja, $x \cdot y$ significa a concatenação da palavra x com a palavra y.

Algoritmo 24: Busca pelo nó n

Dados: árvore de sufixos AS(T), palavra P
Resultado: nó n menos fundo tal que $C(n)$ contém P, null se P não ocorre em T

```
1  n ←AS.raiz
2  enquanto verdadeiro faça
3      c ← |C(n)|
4      se c = |P| então
5          retorna n
6      m ← n.filhos[P[c]]
7      se m = null então
8          retorna null
9      a ← m.fim − m.início + 1
10     i ← 0
11     enquanto i < a faça
12         se T[m.início + i] ≠ P[c + i] então
13             retorna null
14         i ← i + 1
15         se c + i = |P| então
16             retorna m
17     n ← m
```

No Algoritmo 24, vale o invariante de que $C(n)$ é um prefixo de P, e que esse prefixo tem comprimento c. Na linha 4, se $c = |P|$, isso quer dizer que $C(n) = P$, e, pelo fato de o algoritmo não ter terminado numa iteração anterior, n é o nó menos fundo cujo caminho a partir da raiz contém P. Em seguida, na linha 7, se m = null, quer dizer que P não ocorre em T, pois n não tem um filho iniciado com o caractere $P[c]$. Em seguida, a partir da linha 9,

o algoritmo tenta decidir se $C(m)$ também é um prefixo de P. Na linha 12, comparamos cada caractere da aresta entre m e n com $P[c..]$, e se encontrarmos um caractere diferente, decidimos que P não ocorre em T e devolvemos null. Já a linha 15 serve para verificar se P é um prefixo de $C(n) \cdot T[m.\text{início}..m.\text{fim}] = C(m)$. Se isso acontece, então m é o nó menos fundo que estamos procurando. Caso contrário, conclui-se que $C(m)$ é um prefixo de P, e o algoritmo volta para o começo do laço com m fazendo o papel de n na iteração seguinte.

Como o algoritmo só compara P a partir do c-ésimo caractere em cada iteração, ele faz no máximo a comparações diretas de caracteres no laço interno na linha 12 para cada iteração, e como $C(n) \cdot T[m.\text{início}..m.\text{fim}] = C(m)$, o algoritmo nunca compara o mesmo caractere de P duas vezes. Isso quer dizer que o laço interno é executado no máximo $|P|$ vezes, e o algoritmo é executado em tempo linear em $|P|$.

Após obter o nó n com o Algoritmo 24, podemos obter informações sobre as ocorrências de P em T. Se n = null, então P não ocorre nenhuma vez em T, e, caso contrário, o número de ocorrências é igual ao número de folhas abaixo de n, que pode ser determinado em tempo $\mathcal{O}(1)$ após obter n. Além disso, os índices das ocorrências serão exatamente os índices de sufixos armazenados nessas folhas abaixo de n. Esses índices podem ser obtidos em tempo $\mathcal{O}(x)$ após obter n, onde x é o número de ocorrências. Para obtê-los, basta percorrer a sub-árvore enraizada em n procurando pelas folhas. Essa busca leva tempo $\mathcal{O}(x)$ pois como a sub-árvore é compacta e o número de folhas é exatamente x, então, o tamanho da sub-árvore é menor que $2x$.

Com isso, calcular o número x de ocorrências de P leva tempo $\mathcal{O}(|P|)$, e encontrar os índices dessas ocorrências leva $\mathcal{O}(|P| + x)$, enquanto essas operações usando o vetor de sufixos e LCP, como descrito no Capítulo 2, levam tempo $\mathcal{O}(|P|+\log(|T|))$ e $\mathcal{O}(|P|+\log(|T|)+x)$ respectivamente.

4.3 Construção a partir do VS e LCP

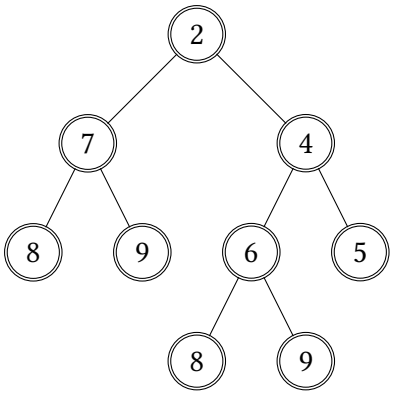
A construção de AS(T) é dividida em três etapas. Na primeira, iremos construir a árvore cartesiana do vetor LCP para obter o "esqueleto" da árvore de sufixos; essa etapa não constrói a árvore inteira, apenas os seus nós internos. A segunda etapa consiste em percorrer a árvore construída na primeira etapa, e povoá-la com as folhas. A terceira consiste em inicializar corretamente todos os campos dos nós da árvore de sufixos. A construção que será apresentada aqui é baseada nas notas de aula do curso *Advanced Data Structures*, lecionado por Erik Demaine [Demaine, 2021].

4.3.1 Árvore cartesiana

Uma *árvore cartesiana* de uma sequência é um min-heap tal que, se for percorrido em in-ordem, a sequência dos nós percorridos resulta na sequência original.

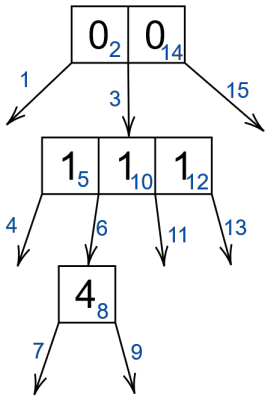
Exemplos

Para, por exemplo, a sequência $(8, 7, 9, 2, 8, 6, 9, 4, 5)$, obtém-se a seguinte árvore cartesiana. Percorrendo-a em in-ordem, obtemos a sequência original, e a árvore também é um min-heap.



No exemplo, não há nós com chave igual à de um dos seus filhos, mas se houvesse, seria necessário juntá-los em um único nó. Ao juntar $n \geq 2$ nós, criamos um "nó maior", o qual além de ter um filho esquerdo e um filho direito como ocorre em uma árvore binária, terá também um filho (potencialmente nulo) entre cada par consecutivo de nós que foram aglomerados para criar o nó maior, totalizando mais $n - 1$ filhos intermediários.

A figura abaixo ilustra a árvore cartesiana da sequência $(0, 1, 4, 1, 1, 0)$, explicitando também todos os filhos nulos de cada nó da árvore (representados pelas setas soltas). Para essa sequência, foi necessário juntar ambos os zeros no nó da raiz, e os três uns em um único nó. É fácil ver que a árvore resultante é um min-heap. Para percorrê-la em "in-ordem", é necessário percorrer os filhos dos nós aglomerados da esquerda para a direita (como ocorreria em uma árvore binária), mas entre as visitas a cada filho, contamos a chave do nó atual mais uma vez (o que também é uma generalização do que acontece em árvores binárias). Os números em azul indicam os passos do percurso em "in-ordem" dessa árvore, devolvendo os elementos da sequência original nos passos 2, 5, 8, 10, 12 e 14.



A propriedade de min-heap da árvore cartesiana do vetor LCP reflete o fato que os nós internos contam quantos caracteres foram soletrados até chegar neles (ou seja, $|C(n)|$), e que essa quantidade de caracteres aumenta quanto mais fundo na árvore a busca chegar; em particular, para um nó interno n, os sufixos de todas as folhas abaixo de n na árvore têm um LCP estendido de $|C(n)|$. A propriedade de a árvore gerar o LCP ao ser percorrida em "in-ordem" se deve ao fato que o LCP nos dá informações sobre o prefixo comum entre dois sufixos consecutivos em VS(T), e que as folhas em "in-ordem" contém os índices dos sufixos na mesma ordem que VS(T).

Construção

Para a etapa 1, iremos denotar a chave heap de um nó n por ch(n) (podemos usar o campo suf, que normalmente seria usado apenas pelas folhas da árvore, para armazenar essa chave). A chave heap será também o número de caracteres soletrados da raiz até o nó n (ch(n) = $|C(n)|$). Usaremos uma pilha, que será denotada por p, e o nó do topo da pilha por p.topo. Na etapa 1, p começa vazia, e, para cada índice i do LCP, o algoritmo atualiza o esqueleto da árvore com LCP[i]. O primeiro índice é tratado separadamente antes de começarmos a, de fato, iterar os índices, pois é o único índice para o qual a pilha está vazia antes da inserção. Para esse índice, é criado um nó de chave 0, pois LCP[1] é o LCP entre VS[0] e VS[1], mas VS[0] será o índice do sufixo "$" de T, já que esse caractere vem antes de todos os outros do alfabeto, e não ocorre em nenhuma outra posição de T, o que significa que o vetor LCP sempre se inicia com 0 (lembrando que ele é indexado a partir de 1). Isso impede que a pilha se esvazie durante o laço principal da etapa 1.

Nem todas as iterações irão necessariamente criar um novo nó na árvore; quando um novo nó não é criado, será necessário expandir um nó n (ou seja, aglomerá-lo com LCP[i] pois LCP[i] = n.ch). Ao expandir um nó, iremos também inicializar seu próximo filho (potencialmente nulo) conforme o Algoritmo abaixo.

Algoritmo 25: Expande Nó

Dados: nó pai n e filho m, que pode ser null

1 n.filhos[n.numeroDeFilhos] ⟵ m
2 n.numeroDeFilhos ⟵ n.numeroDeFilhos + 1

O Algoritmo 25 é executado em tempo $\mathcal{O}(1)$ e será usado para estender os nós ao construir a árvore cartesiana no Algoritmo 26, a seguir. Filhos null no Algoritmo 26 serão substituídos por folhas na etapa 2.

A seguir, apresentaremos o Algoritmo 26, que constrói a árvore cartesiana a partir do vetor LCP, bem como uma análise sobre sua complexidade e a explicação sobre o funcionamento do algoritmo.

Algoritmo 26: Árvore cartesiana a partir do vetor LCP

Dados: vetor LCP(T), inteiro $|T|$
Resultado: árvore cartesiana A(T) de LCP(T)

```
1  se |T| = 1 então
2  |  retorna null
3  n ← novo nó com chave 0
4  ExpandeNó(n, null)
5  p.empilha(n)
6  para i ← 2 até |T| − 1 faça
7  |  t ← null
8  |  enquanto ch(p.topo) > LCP[i] faça
9  |  |  ExpandeNó(p.topo, t)
10 |  |  t ← p.topo
11 |  |  p.desempilha
12 |  se LCP[i] > ch(p.topo) então
13 |  |  n ← novo nó com chave LCP[i]
14 |  |  ExpandeNó(n, t)
15 |  |  p.empilha(n)
16 |  senão
17 |  |  ExpandeNó(p.topo, t)
18 t ← null
19 enquanto |p| > 1 faça
20 |  ExpandeNó(p.topo, t)
21 |  t ← p.topo
22 |  p.desempilha
23 ExpandeNó(p.topo, t)
24 retorna p.topo
```

O Algoritmo 26 consome tempo $\mathcal{O}(|T|)$, pois, no laço principal da linha 6, é empilhado no máximo um nó a cada iteração e a pilha nunca fica vazia já que o primeiro nó empilhado tem chave 0 e a condição da linha 8 irá falhar se ele for o único nó em p. Isso quer dizer que o laço interno da linha 8 é executado no máximo $|T|$ vezes ao todo, pois cada iteração desse laço desempilha um nó e o número de vezes que isso ocorre é limitado pelo número de nós que foram empilhados anteriormente. Como todas as outras operações do laço externo da linha 6 consomem tempo $\mathcal{O}(1)$, e o laço secundário da linha 19 consome tempo linear em $|p|$, então todo o Algoritmo 26 é linear em $|T|$.

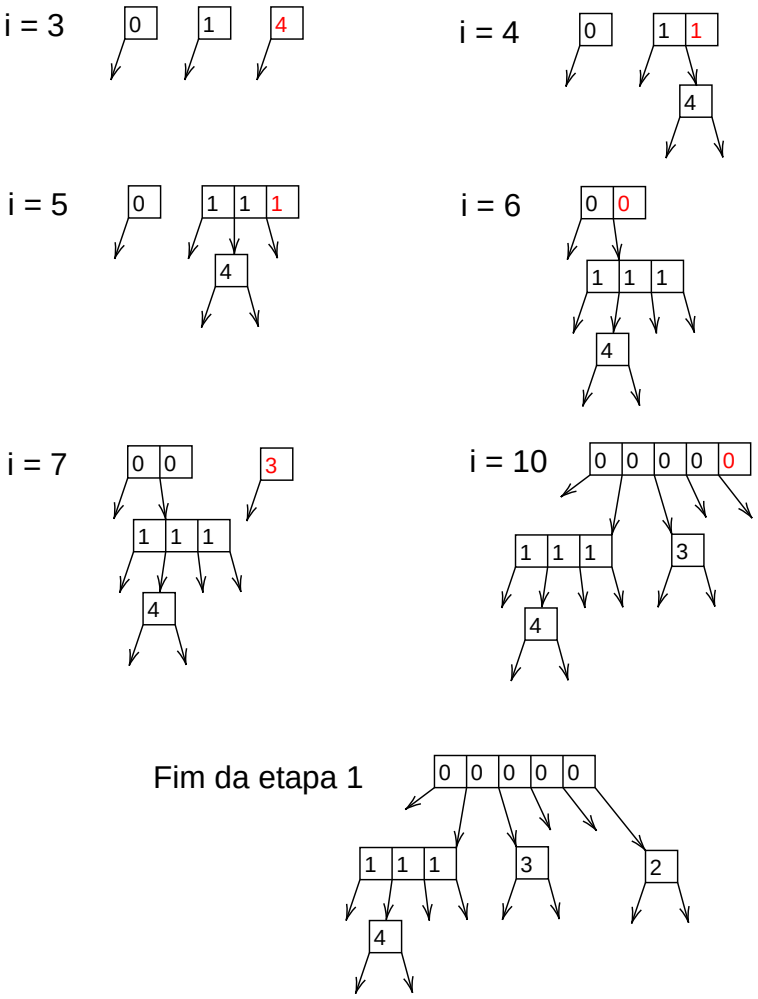
Para o laço principal da linha 6, vale o invariante de que, no início de cada iteração, para cada nó n que está na pilha, todas as subárvores de seus filhos indexados de 0 até n.numeroDeFilhos − 1 já foram completamente construídas, mas que n ainda não foi completamente construído, o que quer dizer que falta determinar pelo menos um filho de n. Esse invariante é válido no início do laço, pois nesse momento a pilha contém um único nó com chave 0, com um único filho null, e como a árvore é comprimida, então n ainda não foi finalizado já que cada nó interno deve ter pelo menos dois filhos. Além disso, também vale que todos os nós presentes na pilha estão em ordem estritamente crescente de chave até o topo da pilha e que a concatenação das sequências em "in-ordem" de cada

nó da pilha, do fundo até o topo, será LCP[1..i − 1]. Isso vale para a primeira iteração já que o único nó presente na pilha tem chave 0 = LCP[1..1].

No início de cada iteração, t é inicializado com null, e o laço interno da linha 8 serve para manter o invariante de que as chaves da pilha estão em ordem crescente (que, por sua vez irá manter a propriedade de que a árvore cartesiana será um min-heap). Para o laço interno, vale o invariante de que, no início de cada iteração, o nó t não está na pilha, e é uma sub-árvore completa. Cada iteração do laço interno expande o topo da pilha com t, que é uma sub-árvore completa, e desempilha o topo, que passa a fazer o papel de t, o que significa que o nó que foi desempilhado agora está completo. O laço da linha 8 termina quando ch(p.topo) ≤ LCP[i], e, mesmo ao terminar, continuam valendo as condições dos invariantes do laço sobre t. No caso em que LCP[i] > ch(p.topo), iremos empilhar um novo nó com chave LCP[i], que tem t como seu primeiro filho. Se t = null, vale que a concatenação das sequências dos nós da pilha será LCP[1..i], pois simplesmente inserimos LCP[i] no topo da pilha. Se $t \neq$ null, essa condição também continua válida, pois os nós que foram desempilhados estão todos na sub-árvore de t, na mesma ordem em que tinham sido empilhados, e t será o primeiro filho do nó com chave LCP[i], o que significa que a sua sequência "in-ordem" virá antes de LCP[i]. Tudo isso irá manter os invariantes do laço principal sobre os nós da pilha. Caso valha que ch(p.topo) = LCP[i], se t = null, o invariante sobre as sequências continua válido, pois seria apenas acrescido de ch(p.topo) = LCP[i] ao final da sequência resultante (pois essa iteração simplesmente expandiu o nó do topo da pilha). Caso $t \neq$ null, a ordem também continuaria a mesma, já que t estava acima do atual topo da pilha no início da iteração, e, ao estender o atual topo com t, estamos mantendo a sequência anterior, e acrescentando LCP[i] ao final da sequência resultante ao expandir o atual topo da pilha.

Ao sair do laço externo da linha 6, iremos desempilhar todos os nós até que sobre apenas o primeiro nó da pilha, e, ao inserir t como último filho de cada nó que estava no topo, estamos completando cada sub-árvore antes de desempilhar, preservando a ordem da sequência dos nós, pois cada nó é inserido como próximo (e último) filho do nó que está abaixo na pilha. A propriedade do min-heap também continua válida pois a pilha estava em ordem crescente, e cada nó é inserido abaixo do anterior. Por fim, a linha 23 serve para, de fato, completar a árvore do primeiro nó que foi empilhado. Uma observação a ser feita é que o número de nós null na árvore devolvida pelo Algoritmo 26 é exatamente igual a $|T|$. Isso se deve ao fato de que, o laço principal da linha 6 sempre estende exatamente um nó com null a cada iteração e é iterado $|T| - 2$ vezes, e, além disso, é feita uma extensão com null antes do laço, e exatamente uma depois do laço. No caso extremo em que $|T| = 1$, o algoritmo simplesmente devolve um único null na linha 2, pois, caso isso aconteça, vale que T = \$ e o vetor LCP é vazio.

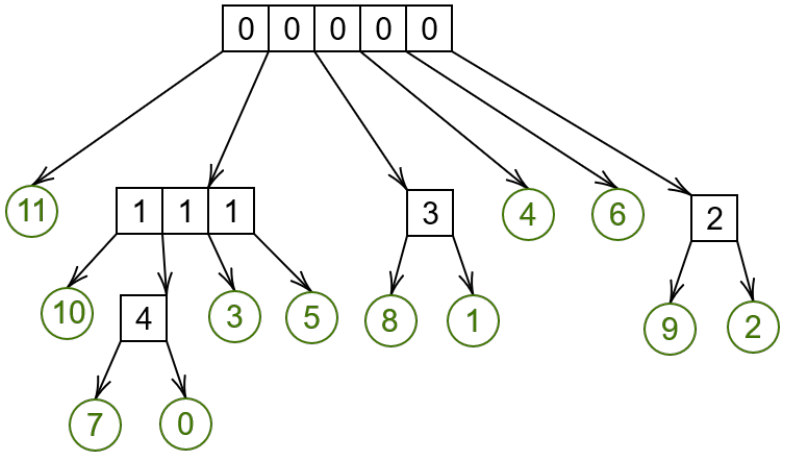
Com isso, o Algoritmo 26 constrói a árvore cartesiana de LCP(T) em tempo $\mathcal{O}(|T|)$. A seguir, estão alguns passos da execução do Algoritmo 26 para T = "abracadabra\$" e LCP(T) = (0, 1, 4, 1, 1, 0, 3, 0, 0, 0, 2). Para cada valor de i demonstrado, a pilha p contém os nós que são raízes (ou seja, os nós exatamente à direita do texto indicando o valor atual de i), com o topo da pilha sendo o nó mais à direita. Em vermelho está também destacado o valor de LCP[i] que foi inserido na iteração atual. As setas indicam quais filhos de cada nó já foram inicializados com a função ExpandeNo() − Algoritmo 25.



4.3.2 Construção das folhas

Em seguida, para a etapa 2, onde são construídas e acrescentadas as folhas, basta percorrer a árvore em "in-ordem", e, ao encontrar um filho null, substituí-lo por uma folha nova. Como há exatamente $|T|$ nós null na árvore, haverá exatamente uma folha para cada sufixo de T. Ao criar as folhas, a i-ésima folha criada corresponderá ao sufixo VS$[i]$ de T. O fato da árvore ser cartesiana significa que o menor ancestral comum entre quaisquer duas folhas corresponde ao LCP estendido entre elas, pois percorrer a árvore em "in-ordem" gera a sequência do LCP, e ao mesmo tempo, as folhas também foram inseridas na ordem do vetor de sufixos. Como a árvore é um min-heap, então o menor ancestral comum entre duas folhas corresponde ao menor entre os LCPs entre todos os sufixos do intervalo entre elas. A etapa 2 também consome tempo linear em $|T|$, já que criar uma folha e acessar o VS a partir do número atual de folhas criadas consome tempo $\mathcal{O}(1)$, e percorrer a árvore inteira consome tempo $\mathcal{O}(|T|)$, pois há menos que $|T|$ nós internos na árvore pelo fato de ela ser comprimida. No caso extremo em que $|T| = 1$ e a etapa 1 devolveu apenas null, a árvore será uma única folha com índice 0. Nessa etapa, já é possível inicializar também o contador do número de folhas abaixo de cada nó; basta que cada nó incremente seu

contador de folhas com o número de folhas criadas ao percorrer cada um de seus filhos "in-ordem". A figura abaixo mostra a árvore cartesiana após a inserção das folhas.



4.3.3 Inicialização dos nós

Por fim, na etapa 3, iremos preencher os demais campos dos nós da árvore. É necessário ajustar as posições de seus nós filhos porque, durante a construção da árvore cartesiana, os filhos eram indexados de 0 a n.numeroDeFilhos − 1, em um intervalo contínuo, mas na nossa implementação da árvore de sufixos, os filhos devem ser indexados por caracteres de T. Também é necessário inicializar os campos início e fim de cada nó para que os rótulos das arestas estejam corretos.

A etapa 3 é feita percorrendo a árvore inteira novamente, e pode inclusive ser realizada junto da etapa 2. Para cada folha f, o final do rótulo da aresta será $|T| - 1$ porque ao chegar em uma folha, o rótulo chegou ao final do texto, completando aquele sufixo. Vale que $C(f) = T[f.\text{suf}..]$, e, como a chave heap de cada nó interno n durante a etapa 1 é na verdade o número de caracteres soletrados até ele a partir da raiz (ou seja, $|C(n)|$), o início do rótulo da aresta da folha é f.suf + ch(f.pai), pois essa aresta deve continuar o sufixo iniciado em f.suf a partir de onde a aresta anterior parou.

Já para cada nó interno n, os índices do intervalo podem ser calculados a partir dos nós filhos (portanto, é necessário inicializar os campos dos filhos recursivamente primeiro), e da chave heap do pai de n. O comprimento do intervalo será ch(n) − ch(n.pai), pois essa diferença corresponde ao número de caracteres no rótulo dessa aresta. O índice do final do intervalo de n pode ser obtido a partir do índice de início de qualquer um de seus filhos. Assim, n.fim pode ser inicializado com, por exemplo, n.filhos[1].início − 1. Em vez de 1, poderia ser usado o índice de qualquer filho de n, mas como todos os nós internos da árvore têm pelo menos dois filhos (pois a árvore é compacta), é preferível usar o índice 0 ou 1, pois eles independem de n e são válidos para todos os nós internos da árvore. O início do intervalo pode ser calculado diretamente a partir do final do intervalo e de seu comprimento.

Cabe ressaltar que, como não há uma aresta acima da raiz, os campos início e fim não estão definidos e não serão utilizados para a busca, logo não precisam ser inicializados. O fato de haver índices não utilizados é consequência de haver mais nós do que arestas.

Por fim, é necessário inicializar corretamente o vetor de filhos de cada nó interno n para ser indexado pelos caracteres de T. Como os filhos foram inseridos na ordem do vetor LCP, que por sua vez é baseada na ordem do VS, então a ordem relativa dos filhos de n já está correta. É necessário mudar apenas os índices deles em n.filhos. Iremos re-ordená-los do maior índice para o menor, pois caso contrário haveria o risco de sobrescrever o ponteiro de um filho que ainda não foi iterado caso ele esteja no índice do caractere pelo qual o filho que está sendo iterado deve ser indexado. Por fim, é conveniente que a correção dos índices de n.filhos seja a última inicialização de n feita na etapa 3, pois os outros campos de n são mais fáceis de ser inicializados se os filhos estiverem em um intervalo contínuo.

O Algoritmo 27 a seguir consome tempo linear no número de filhos do nó n, e como o número de nós da árvore é proporcional a $|T|$, então a execução da etapa 3 para a árvore inteira é feita em tempo linear em $|T|$. A condição n.numeroDeFilhos = 0 serve para decidir se n é uma folha, enquanto n.pai ≠ null serve para decidir se n não é a raiz da árvore.

Algoritmo 27: Etapa3

```
   Dados: nó n, texto T
 1 se n.numeroDeFilhos = 0 então
 2 |   se n.pai ≠ null então
 3 |   |   n.fim ← |T| − 1                    // n é uma folha e não é a raiz
 4 |   |   n.início ← n.suf + ch(n.pai)
 5 senão
 6 |   para i ← 0 até n.numeroDeFilhos − 1 faça
 7 |   |   Etapa3(n.filhos[i], T)
 8 |   se n.pai ≠ null então
 9 |   |   a ← ch(n) − ch(n.pai)
10 |   |   n.fim ← n.filhos[1].início − 1
11 |   |   n.início ← n.fim − a + 1
12 |   i ← n.numeroDeFilhos − 1
13 |   enquanto i ≥ 0 faça
14 |   |   c ← T[n.filhos[i].início]       // Primeiro caractere dessa aresta
15 |   |   n.filhos[c] ← n.filhos[i]
16 |   |   n.filhos[i] ← null
17 |   |   i ← i − 1
```

Com isso, supondo que os vetores de filhos estão inicialmente com null em todas as suas posições, obtém-se AS(T) a partir de VS(T) e LCP(T) em tempo $\mathcal{O}(|T|)$. Como LCP(T) pode ser obtido a partir de VS(T) em tempo linear em $|T|$ por meio do Algoritmo de Kasai et al. [Kasai *et al.*, 2001], descrito no Capítulo 2, e o vetor de sufixos de T pode ser construído em tempo linear em $|T|$ por meio do Algoritmo DC3, descrito no Capítulo 3 [Kärkkäinen e Sanders, 2006; Ormesher, 2017], então é possível obter AS(T) a partir de T em tempo linear em $|T|$. Em vez de obter AS(T) por meio desses vetores, como foi descrito nesse capítulo, também é possível obtê-la diretamente a partir de T por meio do Algoritmo de Ukkonen [Ukkonen, 1995], que constrói AS(T) iterativamente, acrescentando um sufixo novo de T em tempo amortizado $\mathcal{O}(1)$ a cada iteração, finalizando a construção de AS(T) em tempo $\mathcal{O}(|T|)$.

Capítulo 5

Conclusão

Esse trabalho tinha por objetivo principal abordar os tópicos sobre *strings* vistos em MAC0385 – Estruturas de Dados Avançadas. Enquanto havia referências para alguns tópicos vistos nessa disciplina, a literatura sobre a parte de *strings*, em particular sobre o Algoritmo DC3, parecia mais escassa.

Ao longo dessa monografia foram explicados detalhadamente sobre os vetores de sufixos e LCP, e a árvore de sufixos, bem como algoritmos lineares para a construção dessas estruturas. Para o algoritmo de construção linear do vetor de sufixos, o DC3, tanto a parte da implementação quanto da escrita nessa dissertação acabou sendo bastante difícil e demandou bastante tempo; para esse algoritmo, acabamos usando a explicação de Mark Ormesher como base [Ormesher, 2017], mas acrescentamos também alguns detalhes mais minuciosos e sutis sobre a implementação.

Pretendíamos abordar também outros assuntos como o Algoritmo de Ukkonen [Ukkonen, 1995] para a construção linear direta da árvore de sufixos, e outras possíveis aplicações de árvores de sufixos além da busca por uma palavra, tais como encontrar a maior *substring* entre duas palavras ou buscar pelos palíndromos maximais de uma palavra, porém isso não foi possível devido ao tempo ter sido insuficiente para abordar todos esses assuntos.

A implementação dos algoritmos e estruturas de dados abordados nessa monografia foi feita na linguagem C++, e o repositório com a implementação pode ser acessado em https://github.com/MarcoAlves1504/mac0499tcc. Além das implementações, o repositório conta também com um *Makefile* para auxiliar na compilação do código fonte.

Referências

[Cormen *et al.* 2009a] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. 3ª ed. 2009, pgs. 1002–1011 (citado na pg. 1).

[Cormen *et al.* 2009b] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. 3ª ed. 2009, pg. 94 (citado nas pgs. 16, 35).

[Cormen *et al.* 2009c] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. 3ª ed. 2009, pgs. 197–199 (citado na pg. 21).

[Demaine 2021] Erik Demaine. *Advanced Data Structures (Spring '21) Lecture Notes*. Lecture 15 §2.1 e Lecture 16 §4.2. 2021 (citado na pg. 45).

[Gonnet *et al.* 1992] Gaston H. Gonnet, Ricardo A. Baeza-Yates e Tim Snider. "New indices for text: PAT trees and PAT arrays". Em: *Information Retrieval: Data Structures and Algorithms* (1992), pgs. 66–82 (citado na pg. 4).

[Kärkkäinen e Sanders 2006] Juha Kärkkäinen e Peter Sanders. "Simple Linear Work Suffix Array Construction". Em: *Journal of the ACM* 53.6 (2006), pgs. 918–936 (citado nas pgs. 17, 52).

[Kasai *et al.* 2001] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa e Kunsoo Park. "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications". Em: *Combinatorial Pattern Matching, 12th Annual Symposium* (2001), pgs. 181–192 (citado nas pgs. 12, 52).

[Manber e Myers 1990] Udi Manber e Gene Myers. "Suffix arrays: a new method for on-line string searches". Em: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms* (1990), pgs. 319–327 (citado na pg. 4).

[Ormesher 2017] Mark Ormesher. *The Skew (Linear Time) Algorithm for Suffix Array Construction*. 2017. URL: https://gist.github.com/markormesher/59b990fba09972b4737e7ed66912e044 (citado nas pgs. 17, 52, 53).

[Sedgewick e Wayne 2011a] Robert Sedgewick e Kevin Wayne. *Algorithms*. 4ª ed. 2011, pgs. 770–773 (citado na pg. 1).

[Sedgewick e Wayne 2011b] Robert Sedgewick e Kevin Wayne. *Algorithms*. 4ª ed. 2011, pgs. 270–283 (citado nas pgs. 19, 27).

[Ukkonen 1995] Esko Ukkonen. "On-line construction of suffix trees". Em: *Algorithmica* 14.3 (1995), pgs. 249–260 (citado nas pgs. 42, 52, 53).

[Weiner 1973] Peter Weiner. "Linear Pattern Matching Algorithms". Em: *14th Annual Symposium on Switching and Automata Theory* (1973), pgs. 1–11 (citado na pg. 42).

[Wikipedia 2023] Wikipedia. *Suffix array — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Suffix_array&oldid=1131398496 (citado na pg. 4).