UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Bruno Monteiro

String Matching with a Dynamic Pattern

Belo Horizonte
2024

Bruno Monteiro

String Matching with a Dynamic Pattern

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Vinicius Fernandes dos Santos

Belo Horizonte
2024

2024, Bruno Maletta Monteiro.
Todos os direitos reservados

Monteiro, Bruno Maletta.

M775s String matching with a dynamic pattern [recurso eletrônico] / Bruno Maletta Monteiro. – 2024.

1 recurso online (71 f. il., color.) : pdf.

Orientador: Vinícius Fernandes dos Santos

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 68-70.

1. Computação – Teses. 2. Algoritmos de computador – Teses. 3. Complexidade computacional – Teses. 4. Processamento de vetor (Computação) – teses. 5. Reconhecimento de padrões – Teses. I. Santos, Vinicius Fernandes dos. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*52(043)

Ficha catalográfica elaborada pela bibliotecária Irenquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais – ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

String Matching with a Dynamic Pattern

BRUNO MALETTA MONTEIRO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Vinícius Fernandes dos Santos
PROF. VINÍCIUS FERNANDES DOS SANTOS - Orientador
Departamento de Ciência da Computação - UFMG

Victor Campos
PROF. VICTOR CAMPOS
Departamento de Computação - UFC

Felipe Alves da Louza
PROF. FELIPE ALVES DA LOUZA
Faculdade de Engenharia Elétrica - UFU

Belo Horizonte, 4 de dezembro de 2024.

Dedico este trabalho a você, que o está lendo.

Acknowledgments

Agradeço a meu orientador Vinicius dos Santos por acreditar em mim durante todo o processo e por me orientar em um tema fora da sua área de pesquisa.

Agradeço também aos meus pais, por todo o investimento que culminou neste momento. Nada disso teria sido possível sem vocês.

Por fim, agradeço aos amigos e colegas que me incentivaram a continuar e completar este trabalho.

"*O verdadeiro mestrado são os amigos que fizemos no caminho.*"
(Autor desconhecido)

Resumo

Neste trabalho, nós estudamos variações do problema Casamento de Padrões: dadas duas strings, um padrão P e um texto T, queremos computar quantas vezes o padrão ocorre no texto. Nossa contribuição é focada no caso de um padrão dinâmico, ou seja, queremos suportar adição e remoção de caracteres no padrão, e após cada operação computar quantas vezes ele ocorre no texto. Nós mostramos um algoritmo simples usando Suffix Arrays que usa tempo $\mathcal{O}(\log |T|)$, depois de tempo $\mathcal{O}(|T|)$ de pré-processamento. Nós mostramos como estender nossa solução para suportar remoção, transposição (mover a substring para outra posição) e cópia (copiar a substring e colar em uma posição específica) de substrings, na mesma complexidade de tempo. Nossa solução ainda pode ser estendida para suportar um texto online (adicionar caracteres em uma ponta do texto), mantendo as mesmas complexidades amortizadas de tempo. Também fazemos uma análise do tempo de execução do algoritmo proposto contra um algoritmo ingênuo, para demonstrar sua viabilidade. Também é discutida uma generalização do suffix array para várias strings.

Palavras-chave: Strings, Algoritmos, Vetor de Sufixos, Casamento de Padrões.

Abstract

In this work, we study variations of the String Matching Problem: given two strings, a pattern P and a text T, we want to find how many times the pattern occurs in the text. We focus our contributions on the case of a dynamic pattern, that is, we want to support character additions and deletions to the pattern, and after each operation compute how many times it occurs in the text. We show a simple algorithm using Suffix Arrays that achieves $\mathcal{O}(\log |T|)$ update time, after $\mathcal{O}(|T|)$ preprocess time. We show how to extend our solution to support substring deletion, transposition (moving a substring to another position of the pattern), and copy (copying a substring and pasting it in a specific position), in the same time complexities. Our solution can also be extended to support an online text (adding characters to one end of the text), maintaining the same amortized bounds. We also do a running time analysis of the proposed algorithm versus a naive solution, to illustrate its feasibility. A generalization of the suffix array for several strings is also discussed.

Keywords: Strings, Algorithms, Suffix Array, String Matching.

List of Figures

1.1 Example of the String Matching Problem. 16

2.1 Example of π_S for $S =$ `abcabcd`. 21
2.2 Highlight of $\pi_S(4)$ for $S =$ `abcabcd`. 21
2.3 Matching insight from $T =$ `abababababcd` and $P =$ `abababcd`. 22
2.4 Example of using prefix function to directly compute indices of the matches. . 24
2.5 Some strings with highlighting according to their periods. 25
2.6 The prefix function seen as an automaton for the pattern $P =$ `abababaca`. . . . 26
2.7 Aho-Corasick automaton representing strings `a`, `ab`, `bca`, and `caa`. 27

3.1 $\mathcal{SA}$ and $\mathcal{LCP}$ of $S =$ `mississippi`. 34
3.2 Character grouping trick example of the DC3 algorithm. 35
3.3 $\mathcal{SA}$ and $\mathcal{LCP}$ of $S =$ `baabac` (left). Binary search tree with a pair $(\overline{\mathcal{SA}}, \mathcal{LCP})$ (right). 41
3.4 Example of how the $\mathcal{SA}$, $\overline{\mathcal{SA}}$, and $\mathcal{LCP}$ change after a character addition. . . 42
3.5 A possible split representation of the sting `bacabababaca`. 44
3.6 Example of Figure 3.5 after removing 4 characters from the right. 44

4.1 Example trie and the strings it represents. 47
4.2 Example of trie that induces a string of quadratic length if the strings corresponding to paths from leaves to the root are concatenated. 48
4.3 Example trie with nodes colored by their depths mod 3. 49
4.4 Compressed trie of Figure 4.3 for the generalized DC3 algorithm. 50

5.1 For $S =$ `abacabababaaca`, in red, $\mathcal{SR}($`aba`$, S)$, and, in blue, $\mathcal{SR}($`ba`$, S)$. 54
5.2 The set of $\mathcal{SR}(S, T)$ for all S and for $T =$ `mississippi` organized as a tree, in red. 60

6.1 Graph representing the average running time of 5 rounds to execute n pattern updates with a text of size n, for both algorithms. 66

List of Tables

1.1 Deterministic worst-case query time for the Indexing Problem. 17

1.2 Overview of the time bounds we achieve for the *modified pattern reporting problem*. 18

1.3 Overview of the time bounds we achieve for the *modified pattern reporting problem* with an online text. 19

5.1 Time bounds for Problem 5.1. 61

5.2 Time bounds for Problem 5.2. 63

6.1 Average time of 5 rounds to execute n pattern updates with a text of size n, for both algorithms. 66

List of Algorithms

2.1 Computation of π_P. 23
2.2 KMP Algorithm. 24
2.3 Aho-Corasick Construction. 28
2.4 Aho-Corasick Search. 28

3.1 DC3 Algorithm. 36
3.2 Kasai's Algorithm. 37
3.3 Slow Algorithm for Suffix Range. 39
3.4 Fast Algorithm for Suffix Range. 40

5.1 Suffix Range Concatenation. 55

List of Problems

2.1 String Matching Problem. 21
2.2 String Matching with Several Patterns. 26
2.3 String Matching with a Dynamic Pattern Set. 29
2.4 String Matching with Mismatches. 30
2.5 Wildcard Matching. 31
2.6 Indexing Problem. 32

3.1 Online Text Indexing. 41
3.2 Deque Text Indexing. 43
3.3 Longest Repeated Substring. 45
3.4 Number of Distinct Substrings 45
3.5 Longest Common Substring. 46

5.1 Dynamic Pattern and Static Text Matching. 52
5.2 Dynamic Pattern and Online Text Matching. 62

List of Symbols

Σ 20
The alphabet set.

$S[i, j)$ 20
The substring of string S from index i to $j - 1$.

$\circ$ 20
String concatenation.

$\prec, \preceq, \succ, \succeq$ 20
Lexicographical comparison between strings.

$\mathcal{SA}(S)$ 20
Suffix array of S.

$\mathcal{ISA}(S)$ 20
Inverse suffix array of S.

$\mathcal{LCP}(S)$ 20
Lcp array of S.

$\mathbb{N}$ 21
The set of natural numbers (non-negative integers).

$\mathcal{SR}(P, T)$ 37
Suffix range of P with respect to T (see Definition 3.3).

$|P|_T$ 55
Minimum size of an occurrence partition of P with respect to T (see Definition 5.1).

Contents

1 Introduction **16**
1.1 Related work 17
1.2 Goal of the thesis 18
1.3 Overview of results 18
1.4 Organization of the thesis 19

2 Background **20**
2.1 Preliminaries 20
2.2 String matching problem 21
2.2.1 The prefix function 21
2.2.2 Solving Pattern Matching using the prefix function 23
2.2.3 Other applications of the prefix function 24
2.3 String matching with several patterns 25
2.3.1 Static pattern set 25
2.3.2 Dynamic pattern set 29
2.4 String matching with mismatches 30
2.5 Wildcard matching 31
2.6 Indexing problem 32

3 Suffix Array **33**
3.1 Computation of the suffix array and longest common prefix array 33
3.1.1 DC3 algorithm 34
3.1.2 Kasai's algorithm 36
3.2 Suffix range 37
3.2.1 Computation of the suffix range 38
3.3 Online text indexing 40
3.3.1 Online suffix array 41
3.3.2 Pattern queries in the online suffix array 43
3.4 Deque text indexing 43
3.5 Other suffix array applications 45
3.5.1 Longest repeated substring 45
3.5.2 Number of distinct substrings. 45
3.5.3 Longest common substring 46

4 **Generalized Suffix Array** 47
4.1 Adapting the DC3 algorithm . . . 48
4.1.1 Time Complexity . . . 49
4.2 Computing the LCP array . . . 50

5 **Pattern Matching with a Dynamic Pattern** 52
5.1 Suffix range concatenation . . . 53
5.2 Dealing with patterns that do not occur . . . 55
5.3 Improving pattern search . . . 57
5.4 Substring editions . . . 58
5.5 Other pattern search insights . . . 58
5.5.1 Pattern compression . . . 59
5.5.2 Small alphabets . . . 60
5.5.3 Searching for repeated patterns . . . 61
5.6 Time complexities discussion . . . 61
5.7 Online text . . . 61

6 **Experimental Results** 64
6.1 Implementation decisions . . . 64
6.2 Worst case scenario . . . 65
6.3 Results and discussion . . . 65

7 **Conclusion** 67
7.1 Future work . . . 68

References 69

Chapter 1

Introduction

The String Matching Problem is a fundamental problem in the study of strings [10]. This problem consists of, given two strings usually called the *text* and the *pattern*, computing the indices where the pattern occurs in the text (see Figure 1.1).

$$\begin{array}{l} \quad\;\;\downarrow\qquad\quad\;\;\downarrow\;\;\downarrow \\ T = \texttt{aabaacaadaabaaba} \\ P = \texttt{aaba} \end{array}$$

Figure 1.1: Example of the String Matching Problem with text T and pattern P. Occurrences happen on indices 0, 9, 12, highlighted by arrows.

It is trivial to solve this problem in quadratic time, by trying to match the pattern in every position of the text. It turns out that this problem admits a linear time solution, the first of which was published by Knuth et al. [22] and became known as the KMP Algorithm. If we have several patterns known in advance, it is also possible to compute the number of occurrences of all patterns in the text in linear time [1], if the alphabet size is constant.

Another interesting variation of the String Matching Problem is the Indexing Problem, which consists of a fixed text and several queries for counting the number of occurrences of a pattern. In this case, we do not want to spend time linear on the size of the text for each query. However, suffix data structures such as suffix trees and suffix arrays can solve this problem *online* in logarithmic time on the size of the text, per query [32, 25].

Recently more attention has been given to the case of *dynamic* strings. It is possible to efficiently apply changes to the text while maintaining fast query time [17, 15]. Most recently, Amir and Kondratovsky [2] studied another natural variation of the dynamic case: when the pattern is modified. They showed that we can support drastic changes to some pattern efficiently and report how many times this pattern occurs in the text, by using suffix trees.

We continue the study of this last scenario, in which we want to apply changes to the pattern and update the number of times it occurs in the text. By using suffix arrays instead of suffix trees, we achieve similar bounds as Amir and Kondratovsky [2], but with simpler algorithms and requiring less preprocessing space and time.

1.1 Related work

After Knuth, Morris, and Pratt settled the String Matching Problem with a linear time solution in 1970 [27, 22], work was done on variations of the problem. Fischer and Paterson [16] introduced the problem of *Pattern Matching with Wildcards*, in which we can have "don't care" symbols in the text or the pattern, that match with any other symbol. Another studied variation is the *Approximate Pattern Matching*, which consists of finding occurrences of the pattern subject to at most k mismatches [18, 23].

Another problem of interest is the Indexing Problem. This problem can be solved *offline* (if all patterns are known in advance) in linear time [1], for a constant size alphabet. In the *online* scenario, we want to preprocess only the text and answer queries efficiently. Classical solutions for this problem use suffix trees or suffix arrays. Both data structures can be computed directly in linear time and space [14, 19], assuming the alphabet symbols are integers bounded by $|T|^c$, for text T and $c \in \mathcal{O}(1)$. Of course, if the alphabet is unbounded, in the comparison model there is a lower bound of $\Omega(|T| \log |T|)$ (from sorting) for building these data structures [10]. When it comes to query time, the data structures achieve different bounds (see Table 1.1).

Data Structure	Query Time	Source
Suffix Tree	$\mathcal{O}(\|P\| \log \|\Sigma\|)$ or $\mathcal{O}(\|P\| + \log \|T\|)$	[32] [8]
Suffix Array	$\mathcal{O}(\|P\| + \log \|T\|)$	[25]
Suffix Tray	$\mathcal{O}(\|P\| + \log \|\Sigma\|)$	[9]

Table 1.1: Deterministic worst-case query time for the Indexing Problem (for pattern P and text T) for different data structures that can be built in linear time and space.

It is also possible to maintain the $\mathcal{O}(|P| + \log |T|)$ query time while supporting updates of adding a symbol to the end of the text (also called *online text*) in $\mathcal{O}(\log |T|)$ worst-case time [3]. Studies have been made in the case of a *dynamic* text, in which we can add or remove symbols in arbitrary positions of the text while supporting efficient queries [17, 15].

The case of a *dynamic pattern* has been given less attention. Amir and Kondratovsky [2] were the first to give a sub-linear solution for updating the pattern and querying for the number of occurrences of the pattern in the text. They show how to maintain a pattern subject to symbol change (insertion and deletion) in $\mathcal{O}(\log |T|)$ time, and substring copy-pasting and deletion in $\mathcal{O}(\log |T| + \ell)$ time, with ℓ being the size of the modified substring. This is done after $\mathcal{O}(|T|\sqrt{\log |T|})$ preprocessing time and memory.

1.2 Goal of the thesis

The goal of the thesis is to apply the suffix array data structure to the *modified pattern reporting problem* defined by Amir and Kondratovsky [2], that is about maintaining matching information under a dynamic pattern, and to study what bounds we can achieve for this problem.

We also describe several variations of the classic String Matching Problem, and how to apply interesting algorithmic techniques to solve them. In addition, we describe a generalization of the suffix array for a trie, and show how to compute it.

1.3 Overview of results

Using several insights regarding the suffix array, we achieve better time bounds for the *modified pattern reporting problem*, with a simpler algorithm (see Table 1.2). We show how we can maintain a dynamic pattern, subject to character addition or deletion and substring deletion, transposition (moving a substring to another position of the pattern), and copy (copying a substring and pasting it in a specific position). After every update to the pattern, we can output the number of times the pattern occurs in the text.

Operation	[2]	This work
Preprocess Time and Space	$\mathcal{O}(\lvert T\rvert\sqrt{\log\lvert T\rvert})$	$\mathcal{O}(\lvert T\rvert)$
Pattern Symbol Edition	$\mathcal{O}(\log\lvert T\rvert)$	$\mathcal{O}(\log\lvert T\rvert)$
Pattern Substring Edition	$\mathcal{O}(\log\lvert T\rvert+\ell)$	$\mathcal{O}(\log\lvert T\rvert)$

Table 1.2: Overview of the time bounds we achieve for the *modified pattern reporting problem*. Here, edition is an addition or deletion, and ℓ is the size of the modified substring.

In addition, we apply our algorithm for an online text, that is, supporting addition of a symbol to one end of the text. We achieve amortized logarithmic bounds (see Table 1.3). The paper by Amir and Kondratovsky [2] seems to claim worst-case logarithmic bounds, but we could not understand if this is really the case, and we could not get in touch with the authors, despite our best efforts.

Operation	[2]	This work
Preprocess Time and Space	$\mathcal{O}(\lvert T\rvert\sqrt{\log\lvert T\rvert})$	$\mathcal{O}(\lvert T\rvert)$
Pattern Symbol Edition	$\mathcal{O}(\log\lvert T\rvert)$	$\mathcal{O}(\log\lvert T\rvert)^\star$
Pattern Substring Edition	$\mathcal{O}(\log\lvert T\rvert+\ell)$	$\mathcal{O}(\log\lvert T\rvert)^\star$
Text Symbol Extension	$\mathcal{O}(\log\lvert T\rvert)$	$\mathcal{O}(\log\lvert T\rvert)^\star$

Table 1.3: Overview of the time bounds we achieve for the *modified pattern reporting problem* with an online text. Here, edition is an addition or deletion, and ℓ is the size of the modified substring.

1.4 Organization of the thesis

This thesis is organized in the following manner: in Chapter 2, we show several classic problems related to String Matching and well-known solutions for them. The solutions highlight several interesting algorithmic techniques and properties of strings.

In Chapter 3, we describe the Suffix Array and show how it is computed, why it is useful for the problems we are interested in, and some other interesting problems it can solve. This chapter consists of well-known problems and algorithms.

The next three chapters consists of our contributions. In Chapter 4, we discuss a generalization of the Suffix Array for several strings. We show how to adapt the DC3 Algorithm to sort these strings and to produce the LCP array.

Chapter 5 consists of the main contributions of this work. We formally define the problem we are tackling, and describe and prove our solution. We also discuss improvements in special cases. At the end of the chapter, we see how to extend the solution to support extensions to one end of the text.

In Chapter 6, we implement the algorithm and run a quick benchmark to compare its running time with a naive solution. We cannot compare with the algorithm from Amir and Kondratovsky [2], since they did not provide an implementation.

Finally, Chapter 7 contains final considerations about the thesis.

Chapter 2

Background

In this chapter, we will describe well-known solutions for variations of String Matching.

2.1 Preliminaries

Let Σ be a set of symbols which we call the *alphabet*. A *string* S over Σ is a finite sequence of $|S|$ symbols from Σ, indexed from 0 to $|S|-1$. For $0 \leq i < |S|$, $S[i]$ is the i-th symbol from S. We denote as $S[i,j)$ the string $S[i]S[i+1]\ldots S[j-1]$, called a *substring* of S. Note that $S[0,|S|) = S$ and $|S[i,j)| = j - i$. $S[i,j], S(i,j]$, and $S(i,j)$ are defined similarly. We call $S[0,j)$ a *prefix* of S and $S[i,|S|)$ a suffix of S, also denoted by S_i. We say that there is an *occurrence* of a string P in a string T at index i if $T[i, i+|P|) = P$. The reverse of a string S, that is, the string formed by characters of S in reverse order, is denoted by $\overline{S}$.

The *longest common prefix* (*lcp*) between two strings A and B is the size of the largest string that is a prefix of both A and B. Particularly, if $A[0] \neq B[0]$, then $lcp(A,B) = 0$. Also, if a string S is lexicographically smaller than T this is denoted by $S \prec T$, and similarly for $\preceq$, $\succ$, and $\succeq$. We also denote the concatenation between strings S and T as $S \circ T$ or simply ST. The concatenation of a string with itself can be denoted by exponentiation notation, that is, $S^2 = S \circ S$.

A *suffix array* of a string S, denoted as $\mathcal{SA}(S)$, is an array of length $|S|$ such that $\mathcal{SA}(S)[i]$ stores the starting index of the i-th smallest suffix of S, in lexicographical order (note that there are no ties, so the suffix array is unique). The *inverse suffix array* of a string S, denoted as $\mathcal{ISA}(S)$, is an array of length $|S|$ such that $\mathcal{SA}(S)[\mathcal{ISA}(S)[i]] = i$. We also define the *longest common prefix array*, or simply *lcp array* of S, denoted as $\mathcal{LCP}(S)$, as an array of size $|S|-1$ such that $\mathcal{LCP}(S)[i] = lcp(S_{\mathcal{SA}[i]}, S_{\mathcal{SA}[i+1]})$, that is, $\mathcal{LCP}(S)[i]$ is the *lcp* between the i-th and $(i+1)$-th smallest suffixes of S.

Throughout the entire work, whenever there are pattern and text strings, we assume that the pattern is never larger than the text.

2.2 String matching problem

Problem 2.1. String Matching Problem.

Input: Two strings, a text T and a pattern P.

Output: Indices of all occurrences of the pattern in the text.

See Figure 1.1 for an example of the String Matching Problem. It turns out that this problem can be solved in linear time, $\mathcal{O}(|T| + |P|)$, with the algorithm first discovered by Knuth et al. [22], properly named the KMP Algorithm. Before describing the algorithm, we need to introduce the concept of the *prefix function* of a string.

2.2.1 The prefix function

We first need to compute what is known as the *prefix function* $\pi_S : [0, |S|) \to \mathbb{N}$ of a string S. It is defined as follows.

$$\pi_S(i) = \begin{cases} 0, & i = 0 \\ \max_{0 \le k \le i}\{k : S[0,k) = S(i-k,i]\}, & i > 0. \end{cases}$$

What this means is: the value of π for some index only considers the prefix of the string up to that point. The value of π is the largest *proper* prefix that is equal to a suffix of the given prefix. Proper means that it can not be equal to the whole prefix, otherwise we would have $\pi_S(i) = i\,\forall i$. See Figure 2.1 and Figure 2.2.

i	0	1	2	3	4	5	6
$\pi_S(i)$	0	0	0	1	2	3	0
S	a	b	c	a	b	c	d

Figure 2.1: Example of π_S for $S =$ `abcabcd`.

i	0	1	2	3	4	5	6
$\pi_S(i)$	0	0	0	1	**2**	3	0
S	a	b	c	a	b	c	d

Figure 2.2: Highlight of $\pi_S(4)$ for $S =$ `abcabcd`. Note that $\pi_S(4)$ only considers the prefix $S[0,4]$, and is equal to 2 because there is a proper prefix of $S[0,4]$ that is also a suffix of $S[0,4]$ of size 2, and there is none of size greater than 2.

To see why this is useful, let us consider the naive algorithm of trying to match the pattern on every position of the text. Consider the example in Figure 2.3.

T = `abababababcd`	T = `abababababcd`
P = `abababcd`	P = `abababcd`
	`abababcd`

Figure 2.3: Matching insight from $T =$ `abababababcd` and $P =$ `abababcd`.

When trying to match at index 0, we can match only 6 characters (left). The right position to match would be starting at index 2 (right). The quadratic nature of the naive algorithm is such that we have to look at all the red characters again. However, since we have already matched those, we want to find some way of reusing the information that they have already matched with the text.

Note, however, that this is *exactly* the prefix function: the number of characters we can skip (because they already matched) is exactly $\pi_P(5) = 4$. We formalize how this works later, but first we need to actually compute π_P.

From the definition, we have that $\pi_P(0) = 0$. Now we need a way to compute $\pi_P(i)$ using the previous values of π_P. For this we need Lemma 2.1.

Lemma 2.1. *For $0 < i < |P|$, we have that $\pi_P(i) \leq \pi_P(i-1) + 1$.*

Proof. If $\pi_P(i) \leq 1$, the lemma is trivially true. Otherwise, by definition of π_P, for $P[0, i]$, there exists a prefix that is also a suffix of size $\pi_P(i)$. But note that $P[0, \pi_P(i) - 2]$ is a prefix of $P[0, i-1]$ that is also a suffix, and it has size $\pi_P(i) - 1$, so $\pi_P(i-1) \geq \pi_P(i) - 1 \implies \pi_P(i) \leq \pi_P(i-1) + 1$. □

Using the proof of Lemma 2.1, we see that any proper prefix of $P[0, i]$ that is also a suffix of size k implies that there exists a proper suffix of $P[0, i-1]$ that is also a suffix of size $k-1$. Our strategy to find $\pi_P(i)$ will be, therefore, to find the largest proper prefix of $P[0, i-1]$ that is also a suffix that can be extended using $P[i]$.

So we start with $j = \pi_P(i-1)$, the largest prefix of $P[0, i-1]$ that is also a suffix. If $P[j] = P[i]$, we are done, and we set $\pi_P(i) \leftarrow j + 1$. Otherwise, we want to try the *second largest* prefix of $S[0, i-1]$ that is also a suffix. Lemma 2.2 shows that we just need to assign $j \leftarrow \pi_P(j-1)$ and repeat.

Lemma 2.2. *The sequence p such that $p[0] = \pi_P(|P| - 1)$ and $p[i] = \pi_P(p[i-1] - 1)$ for $i > 0$, stopping when $p[i] = 0$, gives the sizes of all the proper prefixes of P that are suffixes of P, in decreasing order.*

Proof. Consider the k-th largest prefix of P that is also a suffix, of size $p[k]$. The $(k+1)$-th largest is the maximum x such that $P[0, x) = P[|P| - x, |P|)$ and $x < p[k]$. But note that $P[0, x)$ is always a prefix of $P[0, p[k])$ and $P[|P| - x, |P|)$ is always a suffix of

$P[|P| - p[k], |P|)$, following simply by the fact that $x < p[k]$. But, from the definition of $p[k]$, we have that $P[|P| - p[k], |P|) = P[0, p[k])$. Therefore, what we are looking for is exactly the largest proper prefix of $P[0, p[k])$ that is also a suffix, and that, by definition, is $\pi_P[p[k] - 1]$. □

In this way we can iterate through all the prefixes that are suffixes in decreasing order, until we find one we can extend. This is implemented in Algorithm 2.1.

Algorithm 2.1. Computation of π_P.

Input: string P.
Output: π_P.
Time Complexity: $\mathcal{O}(|P|)$.

```
π_P(0) ← 0
j ← 0                                      ▷ j is the last computed value of π_P

for i = 1, ..., |P| − 1 do
    while j > 0 and P[j] ≠ P[i] do         ▷ If j = 0, we reached the end of the sequence
        j ← π_P(j − 1)
    if P[j] = P[i] then
        j ← j + 1                          ▷ We have successfully extended a previous prefix / suffix
    π_P(i) ← j
return π_P
```

The time complexity of Algorithm 2.1 might be unclear at first glance, but we can see that every time the inner loop executes, j decreases by at least one, since $\pi_P(j-1) \leq j - 1$. Moreover, the value of j increases by at most one for every iteration of the outer loop.

So, the *total* number of times the inner loop can execute is bounded by the total number of times the outer loop executes, so the algorithm runs in $\mathcal{O}(|P|)$ time.

2.2.2 Solving Pattern Matching using the prefix function

Now we formalize the intuition exemplified in Figure 2.3. What we will do is the naive algorithm with some sort of optimization using the prefix function.

So, for every position i of the text we will maintain the largest j such that $T(i - j, i] = P[0, j)$, that is, the largest suffix of the characters of the text we have seen so far that is also a prefix of the pattern.

Using the value of j on iteration $i-1$, we need to update it for iteration i (adding $T[i]$). This is very similar to the computation of π_P itself. If $P[j] = T[i]$, we just increase j by one, and this maintains correctness. Otherwise, we want to update j to be the *second largest* suffix of $T[0, i-1]$ that is also a prefix of P. From the same reasoning as we did before, we see that we need to set j to $\pi_P(j-1)$. This is implemented in Algorithm 2.2.

Algorithm 2.2. KMP Algorithm.

Input: text T, pattern P.
Output: list of indices of all occurrences of P in T.
Time Complexity: $\mathcal{O}(|T|)$.

$j \leftarrow 0$
matches $\leftarrow$ empty list

for $i = 0, \ldots, |T|-1$ **do**
  while $j > 0$ and $P[j] \neq T[i]$ **do** ▷ If $j = 0$, we reached the end
    $j \leftarrow \pi_P(j-1)$
  if $P[j] = T[i]$ **then**
    $j \leftarrow j+1$ ▷ We have successfully extended a previous match
  if $j = |P|$ **then** ▷ We found a match
    Append $i - j + 1$ to *matches*
return *matches*

The time complexity analysis is also similar to what we did before, and we get that it runs in $\mathcal{O}(|T|)$ time.

Therefore the whole KMP Algorithm, with the computation of π_P, runs in linear time, $\mathcal{O}(|P| + |T|)$.

2.2.3 Other applications of the prefix function

Another way to use the prefix function to solve the String Matching Problem is to compute $\pi_{P \circ \$ \circ T}$, given that \$ is some character that does not occur in Σ. Now we can easily recover the indices of the occurrences, see Figure 2.4.

`abc$cbabca`

Figure 2.4: Example of using prefix function to directly compute indices of the matches with $P =$ `abc` and $T =$ `cbabca`. We can see that we just need to find the indices such that $\pi_{P \circ \$ \circ T}$ equals $|P|$.

Another interesting application of the prefix function is to find *periods* of a string, see Definition 2.1 and Figure 2.5.

Definition 2.1. Period of a String.

Let S be a string. We say that $k \in \mathbb{N}$ is a *period* of S if S is a prefix of $S[0,k)^t$, for some $t > 0$.

```
ababab   abacaba   abaaba
ababab   abacaba   abaaba
ababab   abacaba   abaaba
```

Figure 2.5: Some strings with highlighting according to their periods.

There is a very close relationship between periods of a string and prefixes that are suffixes, as we formalize in Lemma 2.3.

Lemma 2.3. *k is a period of S if, and only if, S has a prefix that is a suffix of length $|S| - k$.*

Proof. Let k be a period of S. It follows from the definition that the string repeats itself every k characters, so skipping the first k characters gives us the beginning of the string: $S[k, |S|)$ is a prefix of S; so we have a prefix that is a suffix of length $|S| - k$. On the other hand, assume that S has a prefix that is a suffix of length k. This means that $S[0] = S[|S| - k], S[1] = S[|S| - k + 1]$, and so on. In fact, it means that $S[i] = S[j]$ if $i \equiv j \pmod{|S| - k}$, and this implies directly that $|S| - k$ is a period of S. □

Therefore, we can find all periods of a string in sorted order in linear time, by iterating through all prefixes that are suffixes.

2.3 String matching with several patterns

2.3.1 Static pattern set

Problem 2.2. String Matching with Several Patterns.

Input: A string T that represents the text, and several strings that represent patterns.

Output: Total number of occurrences of the patterns in the text. That is, the sum, for every pattern, of how many times it occurs in the text.

We will describe a solution proposed by Aho and Corasick [1], that works in linear time assuming $|\Sigma| \in \mathcal{O}(1)$.

It turns out that we can generalize the KMP Algorithm to solve Problem 2.2. For this, we can look into the KMP Algorithm through the perspective of an *automaton* (see Figure 2.6).

i	0	1	2	3	4	5	6
$\pi_P(i)$	0	0	1	2	3	0	1
P	`a`	`b`	`a`	`b`	`a`	`c`	`a`

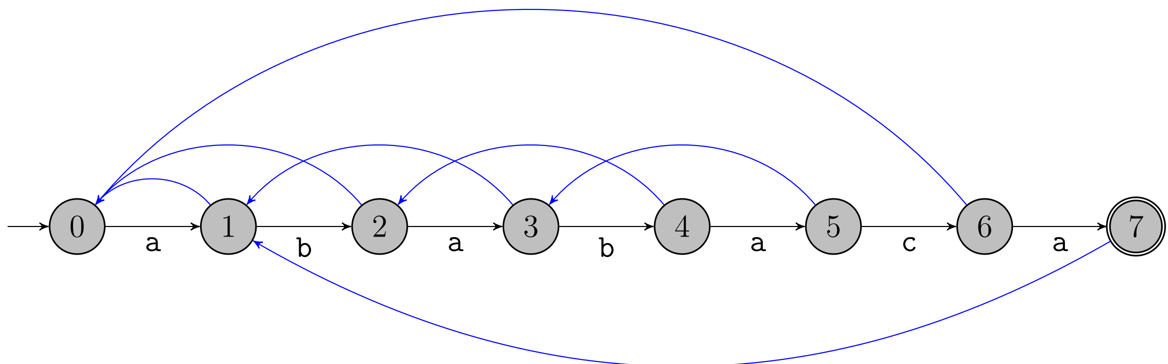


Figure 2.6: The prefix function seen as an automaton for the pattern $P =$ `ababaca`.

We can now interpret computing the matching positions of some pattern in some text as following the transitions of the characters of the text, and if we get to the terminal state, we have found a match. If we can not follow a transition, we follow the blue transition, usually called *failure link*. Note that the failure link of a node that represents some string S goes to the right-most node that represents some proper suffix of S.

This interpretation generalizes well for several patterns. Now, instead of a path, we have a tree (commonly called *trie*) representing all of the patterns. The failure link of some node that represents a string S goes to the deepest node that represents some proper suffix of S. See Figure 2.7.

With the automaton, we need to follow the transitions of the text, using the failure links when the transition does not exist. But after processing each character of the text, it is not enough to check if the current node is terminal (for example, the text `ca` in the example of Figure 2.7). What we actually need is, for every node, to compute the number

of terminal nodes in the path following the failure links from the node to the root. We call this the *term* value of the node.

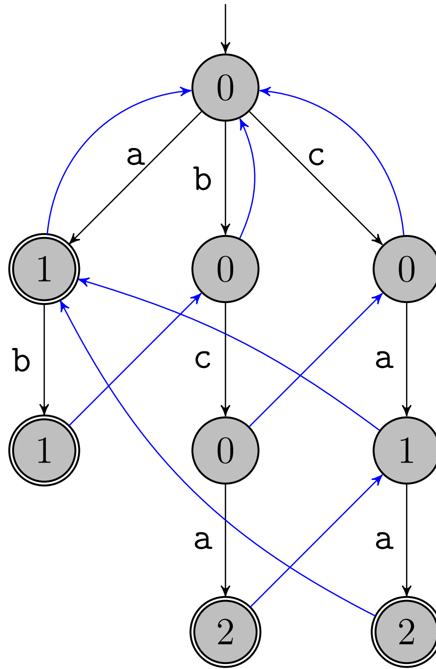


Figure 2.7: Aho-Corasick automaton representing strings `a`, `ab`, `bca`, and `caa`. Failure links are represented in blue. In each node, its *term* value is represented.

Now, after processing each character of the text, we can just add the current *term* value to our answer (see Algorithm 2.4). Now we are left with the task of computing the failure links (*link*) and *term* values.

However, this is easy to do by generalizing the prefix function construction in Algorithm 2.1. We process the nodes in increasing order of depth. When we process node v with parent u and edge (u, v) associated with character c, we check if $link(u)$ has a transition with c. If it does, the node that it transitions to is what we should assign to $link(v)$. This follows from the same reasoning from Lemma 2.1 in this generalized scenario.

If our previous check fails, we follow the failure links and try again, just like in the prefix function. This is detailed in Algorithm 2.3. It is also easy to compute *term* values, using the *term* value of the failure link of the node.

The time complexity of the construction might be unclear at first glance. However, the number of total iterations of the inner loop can be bounded by the sum of the number of iterations it runs for every path from the root to some terminal node. This is at most the depth of the terminal node, from the same argument as for the prefix function.

Therefore, adding it all up, the total number of iterations of the inner loop is bounded by the sum of the sizes of all the patterns. Therefore the construction takes

Algorithm 2.3. Aho-Corasick Construction.

Input: trie $\mathcal{T}$ representing the patterns.
Output: *link* and *term* values for each node.
Time Complexity: $\mathcal{O}(m)$, if m is the sum of the sizes of the patterns.

```
link(0) ← 0                    ▷ Assuming 0 is the root and other nodes are positive integers
term(0) ← 0

for each node v in T in increasing order of depth do
    u ← parent of v
    c ← label of (u, v)
    ℓ ← link(u)
    while ℓ > 0 and ℓ does not have transition c do
        ℓ ← link(ℓ)
    if ℓ has transition c then
        ℓ ← node that ℓ transitions to using c
    link(v) ← ℓ
    term(v) ← term(link(v))
    if v is a terminal node then
        term(v) ← term(v) + 1
return (link, term)
```

Algorithm 2.4. Aho-Corasick Search.

Input: text T.
Output: number of occurrences of all patterns in T.
Time Complexity: $\mathcal{O}(|T|)$.

```
cur ← 0
matches ← 0

for each character c of the text do
    while cur > 0 and cur does not have transition c do
        cur ← link(cur)
    if cur has transition c then
        cur ← node that cur transitions to using c
    matches ← matches + term(cur)
return matches
```

linear time in the sum of the sizes of the patterns, assuming $|\Sigma| \in \mathcal{O}(1)$. Otherwise, we need some data structure to represent the transitions of the trie. We can use a balanced binary search tree, achieving then $\mathcal{O}(m \log |\Sigma|)$ time for the construction, if m is the total size of all patterns.

The time for executing the search also follows the same analysis as in the KMP Algorithm, and it takes $\mathcal{O}(|T|)$ time if $|\Sigma| \in \mathcal{O}(1)$. Otherwise, with balanced binary search trees we can implement it in $\mathcal{O}(|T| \log |\Sigma|)$.

It is not hard to also compute, for every pattern, how many times it occurs in the text. Note that the failure links define a tree, by assigning a parent node for every node. When we process the text, we increase a counter for every node we visit. At the end, we can run a depth-first search on the tree defined by the failure links, and aggregate the counters of the subtree of every vertex. For every terminal node, the aggregated value will be the number of times this pattern occurs in the text.

2.3.2 Dynamic pattern set

Problem 2.3. String Matching with a Dynamic Pattern Set.

Input: Several updates of addition or removal of a pattern to the pattern set, several *online* queries, each represented by a text.

Output: For every query, the total number of occurrences of the patterns (currently in the pattern set) in the text.

Our previous solution using Aho-Corasick's algorithm already works to answer the queries online. However, our Aho-Corasick data structure (AC) does not support adding a new pattern, as many modifications to the data structure might be required to update it.

Let us first only deal with additions of patterns. We employ the following trick: maintain several AC data structures - for every p, we will have at most one AC with 2^p of the strings, so we have at most $\log q$ non-empty ACs, for q patterns added in total.

Every time we add a pattern, let p be the smallest number such that there is no AC of size 2^p. We take the patterns of all smaller ACs and the added pattern and create a new AC of size $1 + \sum_{i=0}^{p-1} 2^i = 2^p$. The smaller ACs are destroyed.

To answer a query, we can just add up the answers of all ACs, taking $\mathcal{O}(|T| \log q) \subseteq \mathcal{O}(|T| \log m)$, if T query text and m is the total size of the patterns.

To bound the total time of all AC constructions, we can note that every string can be promoted to a larger AC at most $\log q$ times, and its contribution to the time complexity is its size times the number of times it is promoted, assuming linear time construction of each AC. Therefore, the total cost of all AC constructions can be bounded by $\mathcal{O}(m \log q) \subseteq \mathcal{O}(m \log m)$.

To support removal of patterns, we can just maintain two sets of the whole procedure: one for the added strings, and one for the removed strings, and subtract their answers.

2.4 String matching with mismatches

Problem 2.4. String Matching with Mismatches.

Input: A text T and a pattern P.

Output: For every position of the text, how many symbols from the pattern match if we try to match at that position.

To solve this problem, we exploit a classic relationship between string matching and polynomial multiplication.

Note that, if we start matching at position k, we want characters of indices i in the text to match those of j in the pattern, such that $i - j = k$. If we reverse the pattern, we want to match characters of indices $|P| - 1 - j$, therefore maintaining $(i) + (|P| - 1 - j) = k + |P| - 1$, similar to polynomial multiplication (finding the sum over pairs associated with numbers with a fixed sum).

Let us focus on the case when $\Sigma = \{0, 1\}$. For the text T and the pattern P, let us define the following polynomials:

$$q_T(x) = \sum_{i=0}^{|T|-1} T[i]\, x^i,$$

$$q_P(x) = \sum_{j=0}^{|P|-1} P[j]\, x^{|P|-1-j}.$$

Notice that q_P is reversed in relation to P. The coefficient of $x^{k+|P|-1}$ in the product $q_T(x) \cdot q_P(x)$ is equal to the pair of indices i from the T and j from P such that $i + (|P| - 1 - j) = k + |P| - 1$ and $T[i]P[j] = 1 \implies T[i] = P[j] = 1$. That is, this is the number of characters equal to 1 that match, if we match the pattern at position k of the text.

We can apply this procedure for every character in Σ, and count, for every position of the text, how many times that fixed character matches. Summing for all symbols from Σ, we can compute how many characters match for every position.

To implement $|\Sigma|$ polynomial multiplications, we can use the Fast Fourier Transform (FFT) algorithm to compute each polynomial multiplication in $\mathcal{O}(n \log n)$ time, if n is the size of the largest polynomial [10]. Therefore, the whole algorithm can be implemented in $\mathcal{O}(|\Sigma||T| \log |T|)$ time, if T is the text.

We can remove the dependency on $|\Sigma|$ in the time complexity in the following manner. Let $r = |P| + |T|$, the total number of characters, and let t be an integer we

will fix later. For every character from Σ, if it appears less than t times in $P \circ T$, iterate through all pairs of positions it occurs in the pattern and in the text in quadratic time and increase the number of matched characters of the match in the corresponding position. For the other characters (there can be at most $\frac{r}{t}$ of them), run the FFT algorithm. This all runs in

$$\mathcal{O}\left(rt + \frac{r}{t} \cdot r \log r\right).$$

The first term is bounded by rt because the worst-case is when all symbols occur almost t times, and in that case, we have at most $\frac{r}{t}$ symbols; so this is overall $\frac{r}{t} \cdot t^2 = rt$. If we choose $t = \sqrt{r \log r}$, we get an overall complexity of $\mathcal{O}\left(r^{\frac{3}{2}} \sqrt{\log r}\right) = \mathcal{O}\left(|T|^{\frac{3}{2}} \sqrt{\log |T|}\right)$.

2.5 Wildcard matching

Problem 2.5. Wildcard Matching.

Input: A text T and a pattern P. The text and pattern might contain a special symbol, that matches with any other symbol.

Output: How many times that pattern occurs in the text.

We will describe the solution by Clifford and Clifford [7]. Another way to solve String Matching Problem would be to compute, for every index k,

$$\sum_{i=0}^{|P|-1} (T[k+i] - P[i])^2 .$$

If this value is zero, then all the terms are zero, so we have a match a position k. Now we can expand this expression:

$$\sum_{i=0}^{|P|-1} (T[k+i] - P[i])^2 = \sum_{i=0}^{|P|-1} \left(T[k+i]^2 - 2T[k+i]P[i] + P[i]^2\right).$$

The first and last terms can be easily computed for all k using standard prefix sum techniques. The middle term can be computed for all k using polynomials and FFT in $\mathcal{O}(|T| \log |T|)$, if T is the text, similar to the approach we used in the previous problem.

Now we introduce wildcards. If we define the value of the characters such that the wildcard is zero and the other characters are positive, we can see that, for matching at position k,

$$\sum_{i=0}^{|P|-1} (T[k+i] - P[i])^2 \, T[k+i]P[i]$$

gives us what we want: for this sum to be zero, every term has to be zero, so every pair of characters either match or at least one of them is a wildcard, which is exactly what we wanted. Expanding the product, we get:

$$\sum_{i=0}^{|P|-1} \left(T[k+i]^3 P[i] - 2T[k+i]^2 P[i]^2 + T[k+i]P[i]^3\right).$$

Each of these 3 terms can be computed using FFT if we square and cube the coefficients of the polynomials accordingly before applying the same trick as before. We sum this up and if we get zero, this means we have a match. The algorithm runs in $\mathcal{O}(|T| \log |T|)$.

2.6 Indexing problem

Problem 2.6. Indexing Problem.

Input: A string T that represents the text, several *online* queries, each containing a pattern.

Output: For every query, how many times that pattern occurs in the text.

In this context, *online* means that we do not know all the queries in advance, that is, we should answer each query at a time. Therefore, we can not use the same solution as before, because it requires us to have all patterns from the beginning.

Note that this shifts the perspective we had in Problem 2.1 and Problem 2.2. On those problems, we did some computation on the patterns, and then used it for the given text to compute the matches.

For Problem 2.6, we want to process the text, and then for every pattern use our computation to find the matches. This can be solved with suffix data structures [14, 19]. In Chapter 3, we will see how to use *suffix arrays* to do so.

Chapter 3

Suffix Array

This chapter consists of well known definitions and algorithms from literature. The suffix array was first developed by Manber and Myers [25] as a more space-efficient alternative to Suffix Trees, a data structure developed for solving Problem 2.6 [32, 25].

Definition 3.1. Suffix Array of a String.

The *suffix array* of a string S, denoted as $\mathcal{SA}(S)$, is an array of length $|S|$ such that $\mathcal{SA}(S)[i]$ stores the starting index of the i-th smallest suffix of S, in lexicographical order.

Note that there are no ties, so the suffix array is unique. It is very common to use, along with the suffix array, the *lcp* array.

Definition 3.2. Longest Common Prefix Array of a String.

The *longest common prefix array* (or simply *lcp* array) of a string S, denoted as $\mathcal{LCP}(S)$, is an array of size $|S| - 1$ such that $\mathcal{LCP}(S)[i] = lcp(S_{\mathcal{SA}[i]}, S_{\mathcal{SA}[i+1]})$, that is, $\mathcal{LCP}(S)[i]$ is the *lcp* between the i-th and $(i + 1)$-th smallest suffixes of S.

See Figure 3.1 for an example. Both the suffix array and the *lcp* array can be computed in linear time [21, 19, 20].

3.1 Computation of the suffix array and longest common prefix array

We will first describe the linear time suffix array construction algorithm called DC3 by Kärkkäinen and Sanders [19]. After that we will see how to use the suffix array to compute the longest common prefix array in linear time, using the algorithm by Kasai et al. [20], known by Kasai's algorithm.

i	$\mathcal{SA}[i]$	$\mathcal{LCP}[i]$	$S_{\mathcal{SA}[i]}$
0	10	1	i
1	7	1	ippi
2	4	4	issippi
3	1	0	ississippi
4	0	0	mississippi
5	9	1	pi
6	8	0	ppi
7	6	2	sippi
8	3	1	sissippi
9	5	3	ssippi
10	2		ssissippi

Figure 3.1: $\mathcal{SA}$ and $\mathcal{LCP}$ of $S =$ `mississippi`.

Since the suffix array order is also a sorted order of the string, we have a time complexity lower bound of the complexity needed to sort the characters of the string. Therefore, we can assume that the symbols have been sorted and $\Sigma = \{1, 2, \ldots, n\}$, if n is the size of the string.

3.1.1 DC3 algorithm

The DC3 algorithm is a divide-and-conquer recursive algorithm. The first observation is that we can reduce the size of the string by compressing groups of characters. We can do this in linear time using Radix Sort, and Counting Sort for each character of the group [10].

For example, consider the string `512213122134`. We can group the characters in pairs: [`51`][`22`][`13`][`12`][`21`][`34`]. The Radix Sort consists of sorting these pairs using two Counting Sorts: first by the first character, and then by the second. After replacing the character pairs by their relative orders, we get `642135`, and the size of the string is halved (see Figure 3.2).

After computing the suffix array of `642135` recursively, this would give us enough information to compare the suffixes of the original string starting at even indices (first, third, etc.). Also, we can compare two odd index suffixes of the original string, by first comparing the first character, and then comparing the rest of the suffixes – two even index suffixes – using the order of the suffixes of the recursion string.

The problem is that we can not compare one even index suffix with an odd index suffix, so our divide-and-conquer scheme is incomplete. The trick to solving this issue is to arrange the characters in groups of *three*.

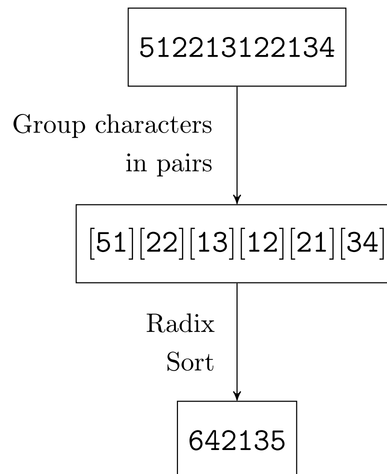


Figure 3.2: Character grouping trick example of the DC3 algorithm.

We will group the characters starting from the indices congruent to 1 and 2 mod 3, and concatenate them. We can consider a special character 0 at the end of the string, smaller than all characters from Σ. For the string `512213122134`, we get `[122][131][221][340] [221][312][213][400]`.

The reason for concatenating them is that this gives us a way to compare not only suffixes of the same type, but also one type of suffix with the other. Note that the comparison between suffixes of different types will never tie and will be equivalent to comparing the corresponding suffixes in the original string.

After running Radix Sort (using Counting Sort three times), we get a string that is $\frac{2}{3}$ the size of the original string. Let us call a comparison between a suffix starting at an index that is congruent to a mod 3 with another congruent to b an ab comparison. With the suffix array of the relabeled string computed recursively we can perform 11, 22, and 12 comparisons.

The other comparisons are also easy: to perform, for example, a 01 comparison, we compare the first characters from the suffixes. If they differ, we know how they compare. Otherwise, we are left with a 12 comparison, which we can do. In fact, all comparisons: 00, 01, and 02 can be performed like this (for the 02 comparison, we need to compare the first 2 characters and we are left with an 21 comparison).

We can also use the same trick to sort the remaining suffixes, the ones with indices congruent to 0 mod 3 (0-suffixes), by using Radix Sort (counting sort by the first character, and then by the order of 1-suffixes from the suffix array of the recursion), in linear time. We are then left with merging the orders of 0-suffixes with the orders of 1 and 2-suffixes we got from the suffix array of the recursion. This is easy to do in linear time, using the comparison trick and standard comparison-based merging. The algorithm is outlined in Algorithm 3.1.

All steps except the recursion can be done in linear time, namely Radix Sort and comparison-based merging. The recursion is called with a string of size $\lceil \frac{2}{3} n \rceil$, if n is the

Algorithm 3.1. DC3 Algorithm.

Input: string S.
Output: suffix array of S.
Time Complexity: $\mathcal{O}(|S|)$.

$S' \leftarrow$ sorted order of groups of 3 characters starting from indices congruent to 1 and 2 mod 3, concatenated ▷ $|S'| \leq \lceil \frac{2}{3}n \rceil$
$sa' \leftarrow \mathcal{SA}(S')$ ▷ Computed recursively
$R0 \leftarrow$ sorted order of suffixes congruent to 0 mod 3, using Radix Sort
$sa \leftarrow$ suffix array from S, merging $R0$ and sa'
return sa

size of the string. Solving the recurrence $T(n) = T(\frac{2}{3}n) + f(n)$, with $f(n) \in \mathcal{O}(n)$, we get the linear bound: $T(n) \in \mathcal{O}(n)$.

3.1.2 Kasai's algorithm

We can use the suffix array to compute the longest common prefix array in linear time. The algorithm by Kasai et al. [20] is very simple. Let S be the string, and let $\mathcal{ISA}$ be the inverse suffix array – for every index i, $\mathcal{ISA}[i]$ tells us in what position of the suffix array the suffix starting at i is at. We will compute the lcp array in their original order of indices: we first compute $lcp(S_0, S_{\mathcal{SA}[\mathcal{ISA}[0]+1]})$, then $lcp(S_1, S_{\mathcal{SA}[\mathcal{ISA}[1]+1]})$, etc.

For some index i, let $j = \mathcal{SA}[\mathcal{ISA}[i] + 1]$, that is, j is the index of the suffix after suffix i in the $\mathcal{SA}$ order. Assume we compute $\ell = \mathcal{LCP}[\mathcal{ISA}[i]] = lcp(S_i, S_j)$, we want to compute $\ell' = \mathcal{LCP}[\mathcal{ISA}[i+1]] = lcp(S_{i+1}, S_{\mathcal{SA}[\mathcal{ISA}[i+1]+1]})$ next. It turns out that $\ell' \geq \ell - 1$. If $\ell = 0$, this is trivially true.

Otherwise $S[i] = S[j]$ and $lcp(S_{i+1}, S_{j+1}) = lcp(S_i, S_j) - 1 = \ell - 1$. Also, since $S_i \prec S_j$, then $S_{i+1} \prec S_{j+1}$, because these are the same strings with the first character removed. Indeed, note that $S_{i+1} \prec S_{\mathcal{SA}[\mathcal{ISA}[i+1]+1]} \preceq S_{j+1}$, following from the suffix array order. From Lemma 3.1, we can see that $\ell' = lcp(S_{i+1}, S_{\mathcal{SA}[\mathcal{ISA}[i+1]+1]}) \geq lcp(S_{i+1}, S_{j+1}) = \ell - 1$.

Lemma 3.1. *Let S be a string and $\mathcal{SA}$ its suffix array. For $0 \leq i < j < |S|$, if there is a common prefix between $S_{\mathcal{SA}[i]}$ and $S_{\mathcal{SA}[j]}$ of length ℓ, then $lcp(S_{\mathcal{SA}[i]}, S_{\mathcal{SA}[k]}) \geq \ell$, for all $i < k < j$.*

Proof. From definition of suffix array, we know that $S_{\mathcal{SA}[i]} \prec S_{\mathcal{SA}[k]} \prec S_{\mathcal{SA}[j]}$. By contradiction, assume that the statement is false. This means that, for some $i < k < j$, there is an integer $x < \ell$ such that $lcp(S_{\mathcal{SA}[i]}, S_{\mathcal{SA}[k]}) = x$ and $S_{\mathcal{SA}[i]}[x] < S_{\mathcal{SA}[k]}[x]$. But

since $lcp(S_{\mathcal{SA}[i]}, S_{\mathcal{SA}[j]}) \geq \ell$, we have that $S_{\mathcal{SA}[i]}[x] = S_{\mathcal{SA}[j]}[x]$, implying $S_{\mathcal{SA}[j]} \prec S_{\mathcal{SA}[k]}$, a contradiction. □

What this means is that we can utilize the computation of the last *lcp* value to compute the next value, by first subtracting one. We then try to increase the value naively in linear time, until the characters differ. This runs in linear time, since the value decreases at most 1 between iterations, and it can increase in at most $|S|$ plus the number of times it decreases, so the whole algorithm runs in linear time. See Algorithm 3.2.

Algorithm 3.2. Kasai's Algorithm.

Input: string S, suffix array $\mathcal{SA}(S)$.
Output: $\mathcal{LCP}(S)$.
Time Complexity: $\mathcal{O}(|S|)$.

```
ISA ← array of size |S|
for i = 0,...,|S| − 1 do
    ISA[SA[i]] ← i
LCP ← array of size |S| − 1
ℓ ← 0
for i = 0,...,|S| − 1 do
    if ISA[i] = |S| − 1 then
        ℓ ← 0                    ▷ LCP value is not defined for the last suffix
    else
        j ← SA[ISA[i] + 1]
        while i + ℓ < |S| and j + ℓ < |S| and S[i + ℓ] = S[j + ℓ] do
            ℓ ← ℓ + 1
        LCP[ISA[i]] ← ℓ
    if ℓ > 0 then
        ℓ ← ℓ − 1
return LCP
```

3.2 Suffix range

The reason why suffix arrays are useful for the Indexing Problem is because the occurrence of any pattern defines a *range* in the suffix array, which we call *suffix range* (see Definition 3.3).

Definition 3.3. Suffix Range.

Let P, T be strings, and let S be the set of indices where P occurs in T. The *suffix range* of P with respect to T, denoted by $\mathcal{SR}(P,T)$ is the set $\{i : \mathcal{SA}(T)[i] \in S\}$. From Lemma 3.2, this set is a range of indices, so we can represent it by a range $[\ell, r)$, meaning that the suffix range is $\ell, \ell+1, \ldots, r-1$. If P does not occur in T, we represent the suffix range by the empty range $[0, 0)$.

Lemma 3.2. *If P occurs in T, then $\mathcal{SR}(P,T)$ is a range of indices.*

Proof. Let $\mathcal{SA}$ be the suffix array of T and let S be the set of indices where P occurs in T. By contradiction, assume that there are indices $i < k < j$ such that $\mathcal{SA}[i], \mathcal{SA}[j] \in S$ and $\mathcal{SA}[k] \notin S$. We know that $lcp(T_{\mathcal{SA}[i]}, T_{\mathcal{SA}[j]}) \geq |P|$, so from Lemma 3.1 we have that $lcp(T_{\mathcal{SA}[i]}, T_{\mathcal{SA}[k]}) \geq |P|$, so $\mathcal{SA}[k]$ is an occurrence of P, a contradiction. □

Lemma 3.3. *The lcp between any two suffixes of S, S_i and S_j, is equal to the minimum over the range in the lcp array (assuming $\mathcal{ISA}[i] < \mathcal{ISA}[j]$):*

$$lcp(S_i, S_j) = \min_{\mathcal{ISA}[i] \leq k < \mathcal{ISA}[j]} \mathcal{LCP}[k],$$

This is also true for any set of strings sorted lexicographically.

Proof. Let $\ell = lcp(S_i, S_j)$. From Lemma 3.1, we know that the first ℓ characters of S_i are equal to those of $S_{\mathcal{SA}[\mathcal{ISA}[i]+1]}$ (the next suffix in the suffix array order), and are also equal to those of $S_{\mathcal{SA}[\mathcal{ISA}[i]+2]}$, and so on until S_j. Therefore, the $\mathcal{LCP}$ values of these positions are at least ℓ. And they can not be all greater than ℓ, otherwise we would have $lcp(S_i, S_j) > \ell$. So the minimum of the $\mathcal{LCP}$ values of the range is exactly ℓ. All these arguments are also valid for any set of strings sorted lexicographically. □

From Lemma 3.3 we can solve lcp queries of arbitrary suffixes in constant time, since range minimum queries can be solved in constant time with linear time construction [4].

So the answer for Problem 2.6 is simply the size of the suffix range for the given pattern.

3.2.1 Computation of the suffix range

Now we are left with the task of computing $\mathcal{SR}(P,T)$. It turns out that this can be computed in $\mathcal{O}(|P| + \log |T|)$ [25]. For this, we compute the first and last indices of

the range independently.

Let us first see how to compute the first index of the range in $\mathcal{O}(|P| \log |T|)$. This can be easily done with a binary search, since what we want is the first suffix that is not smaller than P, and the suffixes are sorted. Note that the $|P|$ factor in the complexity comes from string comparison. This is implemented in Algorithm 3.3. We are using the type of binary search that maintains that the answer is in the open range (L, R) [28].

Algorithm 3.3. Slow Algorithm for Suffix Range.

Input: pattern P, suffix array of the text $\mathcal{SA}(T)$.
Output: first index of $\mathcal{SR}(P,T)$, or $|T|$ if P does not occur in T.
Time Complexity: $\mathcal{O}(|P| \log |T|)$.

if $lcp(P, T_{\mathcal{SA}[0]}) = |P|$ **then return** 0

$(L, R) \leftarrow (0, |T| - 1)$

while $R - L > 1$ **do**
  $M \leftarrow \left\lfloor \frac{L+R}{2} \right\rfloor$
  if $P \preceq T_{\mathcal{SA}[M]}$ **then**
    $R \leftarrow M$
  else
    $L \leftarrow M$

if $lcp(P, T_{\mathcal{SA}[R]}) < |P|$ **then return** $|T|$ ▷ P does not occur in T
return R

For the faster $\mathcal{O}(|P| + \log |T|)$ implementation, we need to use the lcp information. At every iteration of the binary search, we maintain two values ℓ and r, the lcp between P and the suffixes $T_{\mathcal{SA}[L]}$ and $T_{\mathcal{SA}[R]}$, respectively.

Now, we need to compute $lcp(P, T_{\mathcal{SA}[M]})$, and then the next character will tell us what side of the binary search we need to go to. For that, let us assume that $\ell \geq r$ (the other case will be symmetric). We now compare ℓ with $lcp(T_{\mathcal{SA}[L]}, T_{\mathcal{SA}[M]})$ (which we can compute in constant time from Lemma 3.3). We have two cases:

1. $lcp(T_{\mathcal{SA}[L]}, T_{\mathcal{SA}[M]}) < \ell$: in this case, $lcp(P, T_{\mathcal{SA}[M]}) = lcp(T_{\mathcal{SA}[L]}, T_{\mathcal{SA}[M]})$, because P matches ℓ characters from $T_{\mathcal{SA}[L]}$, and $T_{\mathcal{SA}[L]}$ matches less than ℓ characters from $T_{\mathcal{SA}[M]}$.

2. $lcp(T_{\mathcal{SA}[L]}, T_{\mathcal{SA}[M]}) \geq \ell$: now we can find the smallest $i > \ell$ such that $P[i] \neq T_{\mathcal{SA}[M]}[i]$, and assign $lcp(P, T_{\mathcal{SA}[M]}) = i$.

It turns out that if we do Case 2 naively and update our (ℓ, r) accordingly, the total cost of these comparisons throughout the binary search is $\mathcal{O}(|P|)$, because each time we increase $\max(\ell, r)$, and these values can only increase up to $|P|$. Also, on case

Case 1 we always go left on the binary search (because $r \leq lcp(T_{\mathcal{SA}[L]}, T_{\mathcal{SA}[M]}) < \ell$), thus maintaining or increasing the value of r. In fact, both ℓ and r never decrease.

Therefore the total time cost of the binary search is $\mathcal{O}(|P| + \log |T|)$ (see Algorithm 3.4).

Algorithm 3.4. Fast Algorithm for Suffix Range.

Input: pattern P, suffix array of the text $\mathcal{SA}(T)$.
Output: first index of $\mathcal{SR}(P, T)$, or $|T|$ if P does not occur in T.
Time Complexity: $\mathcal{O}(|P| + \log |T|)$.

```
ℓ ← lcp(P, T_SA[0])
r ← lcp(P, T_SA[|T|−1])
if ℓ = |P| then return 0

(L, R) ← (0, |T| − 1)

while R − L > 1 do
    M ← ⌊(L+R)/2⌋
    if ℓ ≥ r then
        if lcp(T_SA[L], T_SA[M]) < ℓ then
            m ← lcp(T_SA[L], T_SA[M])            ▷ Computed in constant time
        else
            m ← ℓ + lcp(P_ℓ, T_SA[M]+ℓ)          ▷ Computed naively
    else
        if lcp(T_SA[R], T_SA[M]) < r then
            m ← lcp(T_SA[R], T_SA[M])            ▷ Computed in constant time
        else
            m ← r + lcp(P_r, T_SA[M]+r)          ▷ Computed naively

    if m = |T| or (SA[M] + m < |T| and P[m] < T_SA[M][m]) then
        (R, r) ← (M, m)
    else
        (L, ℓ) ← (M, m)

plcp ← max(ℓ, r)                 ▷ Largest prefix of P that occurs in T
if plcp < |P| then return |T|
return R
```

3.3 Online text indexing

Problem 3.1. Online Text Indexing.

Input: A string T that represents the text, several *online* queries of a pattern, several updates of adding or removing a character to the beginning of the text.

Output: For every query, how many times does that pattern occur in the current text.

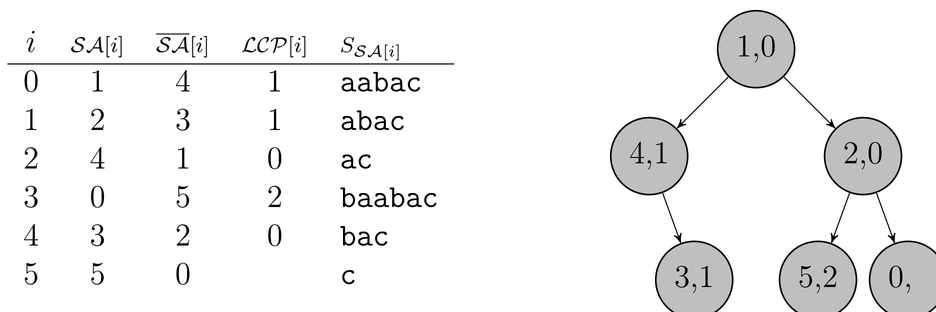
We now discuss the case when the text is given *online*, that is, character by character (Problem 3.1). What we wish to do is maintain the suffix array updated under these constraints.

3.3.1 Online suffix array

We will describe a well-known $\mathcal{O}(\log |T|)$ time solution for adding or removing a character at one end of the text and updating the suffix array [26].

Adding a character at the end of the text can cause several changes in the suffix array. But what we can do is add the character at the *beginning* of the string. This only creates one extra suffix, and we are left to figure out the position it should occupy in the new suffix array. Note that, if we can update the string in the beginning, we can still solve several problems that require updates at the end, by keeping all strings reversed.

We maintain the suffix array in a binary search tree, such that the in-order traversal of the nodes gives us the suffix array order. It will be useful to represent the indices reversed, so when we add a character to the beginning, the reversed indices are maintained. We call the array with the reversed indices of the suffix array $\overline{\mathcal{SA}}$ (see Figure 3.3).



i	$\mathcal{SA}[i]$	$\overline{\mathcal{SA}}[i]$	$\mathcal{LCP}[i]$	$S_{\mathcal{SA}[i]}$
0	1	4	1	aabac
1	2	3	1	abac
2	4	1	0	ac
3	0	5	2	baabac
4	3	2	0	bac
5	5	0		c

Figure 3.3: $\mathcal{SA}$ and $\mathcal{LCP}$ of $S =$ baabac (left). Binary search tree with a pair $(\overline{\mathcal{SA}}, \mathcal{LCP})$ (right). Note that the in-order traversal of the binary search tree gives back the $\overline{\mathcal{SA}}$.

It is easy to see that after adding a character to the beginning of the string, at most one value of the $\mathcal{LCP}$ we already had will change. So we only need to find this new $\mathcal{LCP}$ value, the $\mathcal{LCP}$ value of the new suffix, and the position at which we should insert the new suffix (see Figure 3.4).

i	$\mathcal{SA}[i]$	$\overline{\mathcal{SA}}[i]$	$\mathcal{LCP}[i]$	$S_{\mathcal{SA}[i]}$
0	1	4	1	aabac
1	2	3	1	abac
2	4	1	0	ac
3	0	5	2	baabac
4	3	2	0	bac
5	5	0		c

adding b $\Longrightarrow$

i	$\mathcal{SA}[i]$	$\overline{\mathcal{SA}}[i]$	$\mathcal{LCP}[i]$	$S_{\mathcal{SA}[i]}$
0	2	4	1	aabac
1	3	3	1	abac
2	5	1	0	ac
3	1	5	2	baabac
4	4	2	1	bac
5	0	6	0	bbaabac
6	6	0		c

Figure 3.4: Example of how the $\mathcal{SA}$, $\overline{\mathcal{SA}}$, and $\mathcal{LCP}$ change after a character addition (changes are highlighted in red). Note that, if we store $\overline{\mathcal{SA}}$ and $\mathcal{LCP}$, we need to change at most one $\mathcal{LCP}$ value and add the new $(\overline{\mathcal{SA}}, \mathcal{LCP})$ pair.

It should be noted that the binary search tree representation still allows us to do arbitrary *lcp* queries (between any two suffixes), by maintaining the minimum over the *lcp* values of the subtree for every node. Now we are left with a classic binary search tree operation – to find the aggregate (in this case, minimum) over a range of nodes. This is well-known and simple to solve in $\mathcal{O}(\log n)$ time, if we have n nodes in the tree [10].

To find where to insert the new suffix, we need a way to compare the suffixes we already have in constant time. One way to do this is to associate an extra value *tag* for every node, such that node x comes before node y in the binary search tree (equivalently, suffix x is lexicographically smaller than suffix y) if, and only if, $tag(x) < tag(y)$.

This can be done if we choose a suitable binary search tree. This will make additions and deletions $\mathcal{O}(\log n)$ amortized time, and $\Theta(n)$ worst-case [12, 5, 26].

If we wish a worst-case $\mathcal{O}(\log n)$ time per operation data structure, we can maintain a separate (and much more complicated) *order maintenance data structure*, that can be implemented in worst-case constant time for update and query [11].

Assuming we have the *tag* values, we are left to insert the new suffix to the binary search tree. For this, it is enough to know how to compare the new suffix with any other suffix that we already have inserted.

Note, however, that this is easy: we compare the first letter. If it is different, we know which one is smaller. Otherwise, we are left to compare suffixes starting at the next index, that is, two suffixes we already have in our data structure, so we can use the *tag* values.

Therefore, by comparing the new suffix with any other suffix in constant time, we can insert the new suffix in the binary search tree in $\mathcal{O}(\log |T|)$ time.

The *lcp* values we need to update can be computed in a similar way: we compare the first characters. If they differ, the *lcp* is equal to zero. Otherwise, we are left with an *lcp* query between two suffixes already in our data structure, and we can solve this in $\mathcal{O}(\log |T|)$ time as discussed.

To remove a character, we just go down in the binary search tree to find the corresponding suffix and delete it. When removing the suffix, the *lcp* value of the node that comes right before the deleted node might change. However, it is easy to update, since its new value is the minimum between the old value and the *lcp* value of the deleted node, from Lemma 3.3.

3.3.2 Pattern queries in the online suffix array

It turns out that we can adapt Algorithm 3.4 to work in the same $\mathcal{O}(|P| + \log |T|)$ time complexity using the binary search tree representation. Instead of doing a binary search, we go down on the binary search tree, and each time decide if we should go to the left or right child.

The only non-trivial aspect is the *lcp* query that we did in constant time in Algorithm 3.4, and at first this would take logarithmic time using the binary search tree. But note that the queries we need to do are queries of entire subtrees, so by storing the minimum over *lcp* values in each subtree, we can also do it in constant time.

Therefore, in the online text scenario, we can also find the suffix range in $\mathcal{O}(|P| + \log |T|)$ time.

3.4 Deque text indexing

Problem 3.2. Deque Text Indexing.

Input: A string T that represents the text, several *online* queries of a pattern, several updates of adding or removing a character to the beginning or the end of the text.

Output: For every query, how many times does that pattern occur in the current text.

It turns that we can extend what we did for Problem 3.1 to solve Problem 3.2.

We know how to support updates to one end of the text (we discussed a solution that updates the text on the left, but we can simulate updates on the right by working with the reverse of the strings, and this maintains the matching information), and we wish to support updates to both.

We represent the text split between two parts: a left part, to which we will update on the left, and a right part (that we will store reversed in another binary search tree), to which we will update on the right (see Figure 3.5).

$\leftarrow$ `bacaba|baca` $\rightarrow$

Figure 3.5: A possible split representation of the sting `bacababaca`.

Now, to answer a query for some pattern P, we just need to add how many times it occurs in the left and right parts (we will have to reverse the pattern to query for the right part). But note that we are not counting matches that begin in the left part and end in the right part.

But we can count them separately: just take the last $|P| - 1$ characters of the left part, concatenate them with the first $|P| - 1$ characters of the right part, and run the KMP Algorithm. This is done $\mathcal{O}(|P|)$ time.

The only problem we are left to solve is when one of the parts gets empty and we need to remove a character from that side, as in Figure 3.6. We can not trivially remove another character from the right.

$\leftarrow$ `bacaba|` $\rightarrow$ $\quad\Longrightarrow\quad$ $\leftarrow$ `bac|aba` $\rightarrow$

Figure 3.6: Example of Figure 3.5 after removing 4 characters from the right.

But we can use a trick: if one side gets empty, re-construct the entire structure in linear time, now splitting it in the middle, as in Figure 3.6.

This turns out to not add to the time complexity, in the amortized sense. We can bound the time complexity by summing, for every character, the number of times it gets rebuilt. Consider some time we rebuild. For every character that gets rebuilt, either it is the first time it gets rebuilt, or it has been rebuilt before. In this last case, the last time it was rebuilt, it had a "matching" character on the other side of the division that now got deleted, so we can charge the cost of rebuilding this character to the deletion of the matching character.

Therefore, summing the number of times each character gets rebuilt, we get a linear bound (considering the first time each character gets rebuilt) plus another linear bound, because each time a character is rebuilt again, a unique matching character gets deleted. So the whole cost of all rebuilds is linear.

Our solution then runs in $\mathcal{O}(|P| + \log |T|)$ worst-case time per pattern query, and $\mathcal{O}(\log |T|)$ amortized time per update.

3.5 Other suffix array applications

We can use the $\mathcal{SA}$ and $\mathcal{LCP}$ arrays to solve several interesting problems.

3.5.1 Longest repeated substring

Problem 3.3. Longest Repeated Substring.

Input: String S.

Output: Size of the largest string that occurs in S at least twice.

Assume the answer occurs at indices i and j in S. We know that $\ell = lcp(S_i, S_j)$ is the answer. Since any lcp query between two suffixes is a range minimum query in the $\mathcal{LCP}$ array (Lemma 3.3), we know there is a value in the $\mathcal{LCP}$ array that is at least ℓ.

On the other hand, any value in the $\mathcal{LCP}$ array implies two equal substrings with that size in S. Therefore, the answer is the maximum value in the $\mathcal{LCP}$ array.

3.5.2 Number of distinct substrings.

Problem 3.4. Number of Distinct Substrings

Input: String S.

Output: How many **distinct** substrings the string S has.

To solve this problem, let us remember that any substring is a prefix of some suffix of S. We will process the suffixes in the order of the suffix array. For some index i, we will count how many prefixes of $S_{\mathcal{SA}[i]}$ we have not yet counted, that is, how many prefixes of $S_{\mathcal{SA}[i]}$ are not prefixes of $S_{\mathcal{SA}[j]}$, for $j < i$.

But note that this is exactly all prefixes of $S_{\mathcal{SA}[i]}$ except for the *lcp* between this suffix and the previous suffix in the suffix array; prefixes smaller than that, we have considered before, and the others are new. So the answer equals

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

3.5.3 Longest common substring

Problem 3.5. Longest Common Substring.

Input: Strings S, T.

Output: Size of the largest string that occurs in both S and T.

We can compute the suffix array of $S \circ \$ \circ T$, such that \$ is a character that does not appear on the strings. Now we look at the $\mathcal{LCP}$ array. The answer is the maximum *lcp* between a suffix from the first $|S|$ characters of the concatenation with a suffix from the last $|T|$ characters of the concatenation.

It again follows from Lemma 3.3 that it is enough to consider adjacent suffixes in the suffix array order. Therefore, we can just iterate through adjacent suffixes in the $\mathcal{SA}$, and, if they originate from different strings, consider their *lcp* for the answer. The maximum of such values is the size of the longest common substring.

Chapter 4

Generalized Suffix Array

In this chapter, we introduce a generalization of suffix arrays. As we sill see, the ideas from previous algorithms generalize well, but we include the details here as we could not find such generalization in the literature. Imagine we want to sort the suffix of several strings. Of course, we can concatenate all strings and compute the suffix array of the resulting string, as discussed by Louza et al. [24].

We can generalize the suffixes of a string in the following way. Consider a *trie*, that is, a rooted tree such that each edge is assigned a symbol from Σ, and edges originating from the same vertex have different labels. We can define, for every vertex that is not the root, the string that is formed by going up the edges to the root (see Figure 4.1). Note that this is the reverse of the strings usually defined for the vertices of a trie. These are the strings we want to sort.

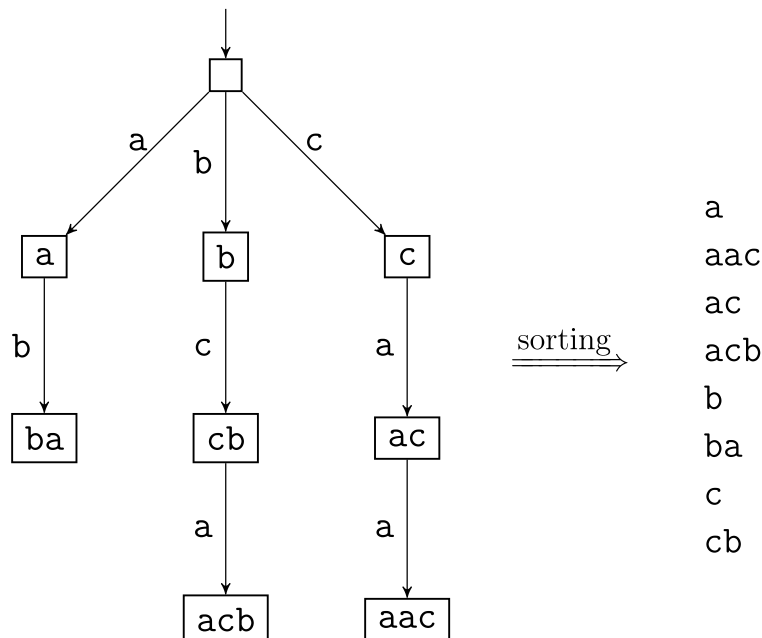


Figure 4.1: Example trie and the strings it represents. For each node, we assign the string we get following parent edges to the root.

Recall that we denote by $\overline{S}$ the reverse of the string S. Note that, if we have

the trie only containing $\overline{S}$, this is equivalent to sorting the suffixes of S, so this indeed generalizes the suffix array. Also note that we can create a string by concatenating the strings formed by the path from leaves to the root, and, by computing the suffix array of that string, sort the strings defined by the vertices. But this concatenated string can have quadratic size on the size of the trie, as shown in Figure 4.2.

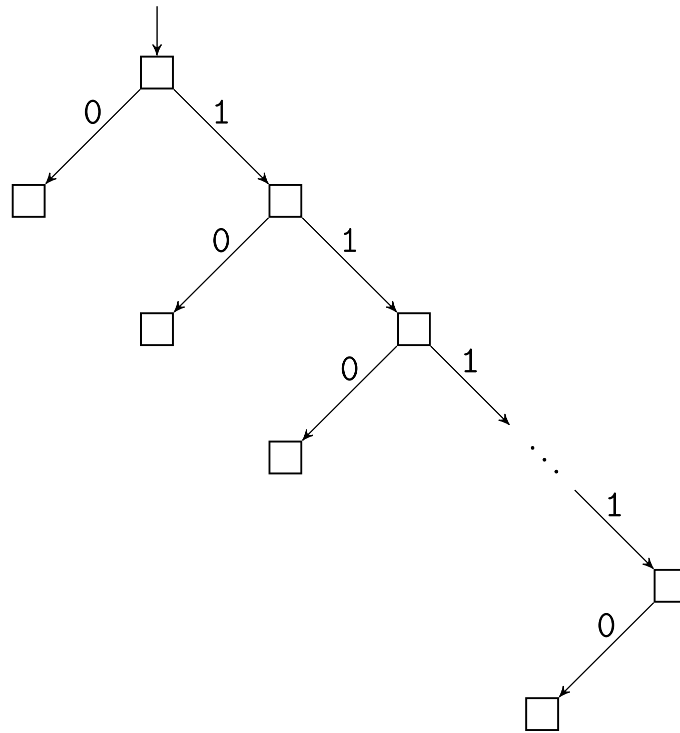


Figure 4.2: Example of trie that induces a string of quadratic length if the strings corresponding to paths from leaves to the root are concatenated. For a trie with n nodes, the size of the concatenated string is $1 + 2 + \cdots + \frac{n}{2} = \frac{n^2+2n}{4} \in \Theta(n^2)$.

4.1 Adapting the DC3 algorithm

It turns out we can adapt the DC3 algorithm described in Subsection 3.1.1 to work in this scenario. Instead of dividing by the indices mod 3, we partition the trie nodes by depth. For every equivalence class of nodes by their depth mod 3, we can define a new trie containing those nodes and define the parent of a node as the node you get to if you follow 3 parent edges (see Figure 4.3 and Figure 4.4). We assign for each edge the concatenation of the symbols on the 3 edges we previously had, and then use Radix

Sort to relabel them to a single symbol maintaining their lexicographical order, as in the original DC3 algorithm.

We can then solve the problem recursively for two of the tries (we create a new root and connect both tries to it, as in Figure 4.4). The order of the strings defined by the remaining vertices can be computed in the same way as in DC3, by using Radix Sort comparing the first character, and then using the order we got from the recursion. Finally, we apply standard comparison-based merging to get the final order of all strings.

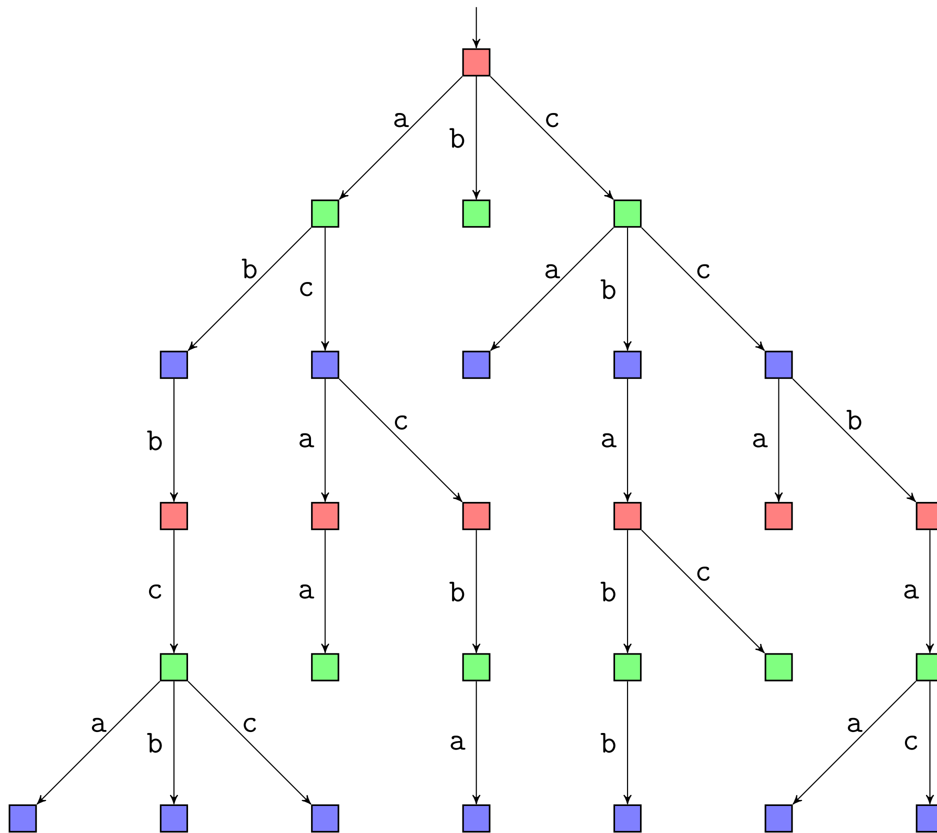


Figure 4.3: Example trie with nodes colored by their depths mod 3.

4.1.1 Time Complexity

Note that we can choose which of the 3 equivalence classes of nodes by their depths mod 3 to not include in the recursion. Say their sizes are n_1, n_2, n_3, and n is the total number of nodes in the trie. Since $n_1 + n_2 + n_3 = n$, we have that $\max(n_1, n_2, n_3) \geq \lceil \frac{n}{3} \rceil$, otherwise $n_1 + n_2 + n_3 < n$. Therefore, we can choose the largest partition size to leave out of the recursion, and the size of the remaining trie will be at most $\lfloor \frac{2}{3} n \rfloor$.

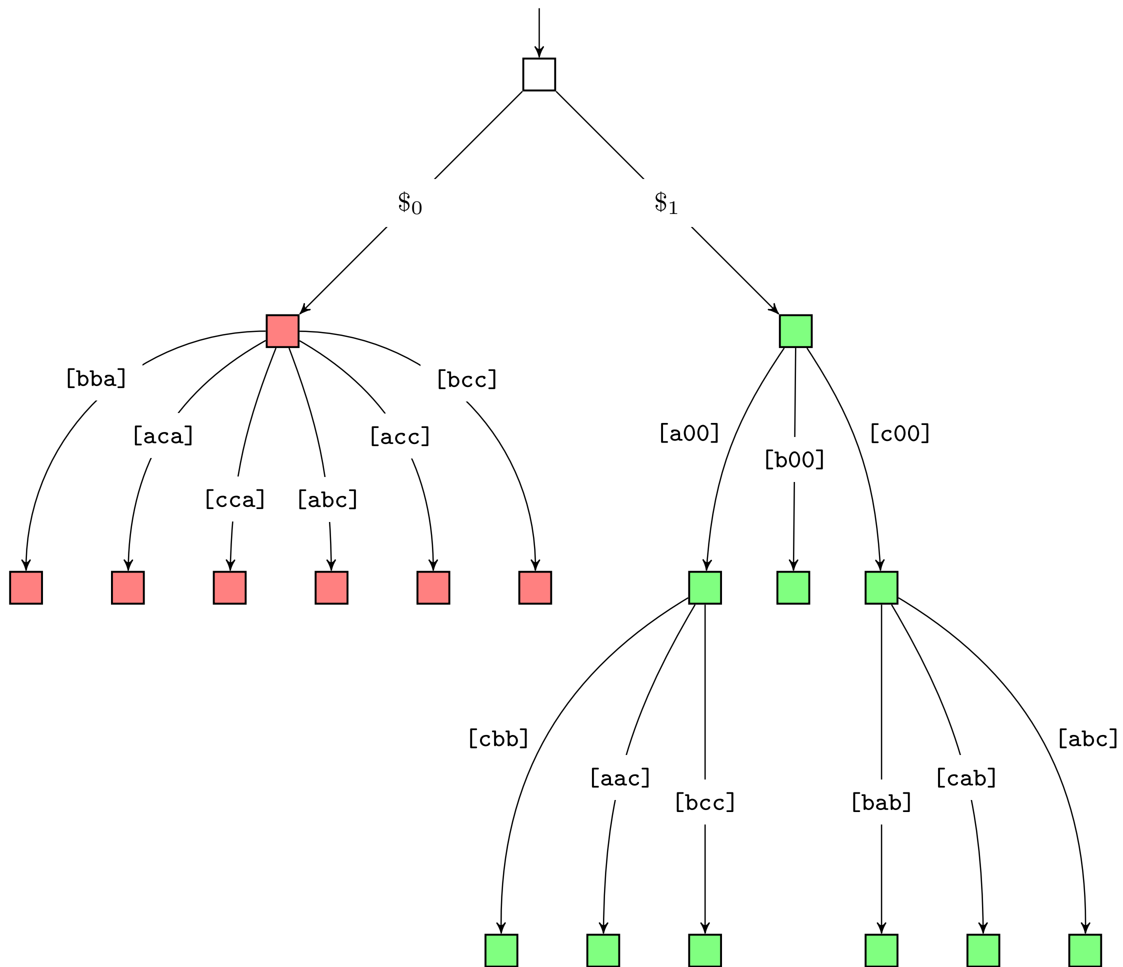


Figure 4.4: Compressed trie of Figure 4.3 for the generalized DC3 algorithm. For the green trie on the right, we add a special character 0 (that compares smaller than all characters we had) to the triplets of characters, since there are no more parent edges to follow in the original trie. Also, note that we need to assign more special characters $\$_0 \neq \$_1$ to connect the tries to a new root. These characters need to compare smaller than all characters from both tries.

Using this trick, the algorithm takes time $T(n) = T\left(\frac{2}{3}n\right) + f(n)$, with $f(n) \in \mathcal{O}(n)$, which solves to $T(n) \in \mathcal{O}(n)$.

4.2 Computing the LCP array

The algorithm by Kasai et al. [20] does not generalize well to this scenario. However, we can apply the same idea as described by Kärkkäinen and Sanders [19] to compute the $\mathcal{LCP}$ array during recursion of the DC3 algorithm.

We will make use of arbitrary *lcp* queries, that can be done in constant time by Lemma 3.3 and by using linear time construction and constant time query range minimum query data structures [4]. Let us assume that the recursive call computes not only the $\mathcal{SA}$, but also the $\mathcal{LCP}$ array of the recursion trie. We then construct the range minimum query data structure over this $\mathcal{LCP}$ array in linear time.

After that, in the comparison merging step of the algorithm, we will compute the final $\mathcal{LCP}$ array. When we put string i after string j in the final ordering, we need to compute their *lcp*. If they both came from the recursion trie, we already know their *lcp*. If not, we can apply a similar trick to the one we use for comparing: we compare the first character. If they are different, the *lcp* is 0. Otherwise, the *lcp* is 1 plus the *lcp* of the strings after removing their first character, that is, following a parent edge. By doing this at most twice, we will be left with two strings from the recursion trie.

Chapter 5

Pattern Matching with a Dynamic Pattern

This chapter consists of the main contributions of this work: we propose a solution for the problem of having a static text, and maintaining a pattern under several types of updates, while keeping the matching information.

Problem 5.1. Dynamic Pattern and Static Text Matching.

Input: A string T that represents the text, several updates to the pattern string P that can be one of the following.

1. Pattern search: set the current pattern to be some pattern given in the input;
2. Pattern symbol edition: addition or deletion of some character of the current pattern;
3. Pattern substring edition: substring deletion, transposition (moving the substring to another position), or copy on the current pattern.

Output: After every operation, the number of times the current pattern occurs in the text.

This problem was solved by Amir and Kondratovsky [2] using suffix trees, however our solution with suffix arrays is considerably simpler, reduces the preprocess and space, and improves upon the time required for the substring operations.

Note that in this work we do not assume that the text T is necessarily much larger than the pattern P ($|T| \gg |P|$). Even though we assume $|T| > |P|$, factors of, for example, $\log |T|$ and $|P|$ are both taken into account in the time complexity analysis. That is, we are interested in the cases when the pattern can be large in relation to the text. If it is always small, we can solve Problem 5.1 by using text indexing (Subsection 3.2.1).

5.1 Suffix range concatenation

Our main insight is the fact that if we have $|A|, |B|, \mathcal{SR}(A,T)$, and $\mathcal{SR}(B,T)$, we can compute $\mathcal{SR}(A \circ B, T)$ efficiently, as we will see in Theorem 5.1 and Algorithm 5.1.

Lemma 5.1. *If $A \circ B$ occurs in T, then $\mathcal{SR}(A \circ B, T)$ is a sub-range of $\mathcal{SR}(A,T)$.*

Proof. All of the suffixes from $\mathcal{SR}(A \circ B, T)$ have A as a prefix, therefore they are also in $\mathcal{SR}(A,T)$. □

Theorem 5.1. *Given $|A|, |B|, \mathcal{SR}(A,T)$, and $\mathcal{SR}(B,T)$, we can compute $\mathcal{SR}(A \circ B, T)$ in $\mathcal{O}(\log |T|)$ time.*

Proof. From Lemma 5.1, we want to find a sub-range of $\mathcal{SR}(A,T)$, so it is enough to find its first and last position. Let $[\ell, r)$ be the range $\mathcal{SR}(A,T)$. We want to find the first $i \in [\ell, r)$ such that $T_{\mathcal{SA}[i]+|A|}$ has B as a prefix. But note that, since $T_{\mathcal{SA}[\ell]}, T_{\mathcal{SA}[\ell+1]}, \ldots, T_{\mathcal{SA}[r-1]}$ all have an *lcp* of at least $|A|$, then $T_{\mathcal{SA}[\ell]+|A|}, T_{\mathcal{SA}[\ell+1]+|A|}, \ldots, T_{\mathcal{SA}[r-1]+|A|}$ appear in this order in the suffix array, because their lexicographical comparison is not decided by their first $|A|$ characters, so skipping them maintain their order. Therefore, we can do a binary search on the range $[\ell, r)$, and when checking some suffix, we look at the position of the suffix that skips $|A|$ characters from that suffix, and check if it is in $\mathcal{SR}(B,T)$, using the $\mathcal{ISA}$. We can therefore find the first and last position in $[\ell, r)$ that correspond to $\mathcal{SR}(A \circ B, T)$, with two binary searches. □

A visual representation of Lemma 5.1 is shown in Figure 5.1. The algorithm is implemented in Algorithm 5.1, using the type of binary search that maintains that the answer is in the half-open range $[\ell, r)$ [28].

Let us see how to delete some characters from the beginning or the end of the pattern, and update the suffix range accordingly. For this, we need the following lemma.

Lemma 5.2. *Given a single index $i \in \mathcal{SR}(P,T)$, we can find $\mathcal{SR}(P,T)$ in $\mathcal{O}(\log |T|)$ time.*

Proof. We just need to find $\ell = \min_{j \le i}\{j : lcp(T_{\mathcal{SA}[j]}, T_{\mathcal{SA}[i]}) \ge |P|\}$ and $r = \min_{i<j}\{j : lcp(T_{\mathcal{SA}[j]}, T_{\mathcal{SA}[i]}) < |P|\}$, and this gives us $\mathcal{SR}(P,T) = [\ell, r)$. Both these indices can be found with a binary search using *lcp* queries, since from Lemma 3.3 the *lcp* value is monotonic when we increase the range: if $k < j \le i$, $lcp(T_{\mathcal{SA}[k]}, T_{\mathcal{SA}[i]}) \le lcp(T_{\mathcal{SA}[j]}, T_{\mathcal{SA}[i]})$, and symmetrically for the other direction. □

Lemma 5.3. *If P occurs in T, given $\mathcal{SR}(P,T)$ we can compute the suffix range of P with $k < |P|$ characters removed from the beginning or the end in $\mathcal{O}(\log |T|)$ time.*

i	$\mathcal{SA}[i]$	$\mathcal{LCP}[i]$	$S_{\mathcal{SA}[i]}$
0	13	1	a
1	10	1	aaca
2	8	3	abaaca
3	6	5	abababaaca
4	4	3	ababaaca
5	0	1	abacababababaaca
6	11	3	aca
7	2	0	acabababaaca
8	9	2	baaca
9	7	4	babaaca
10	5	2	bababaaca
11	1	0	bacabababaaca
12	12	2	ca
13	3		cababababaaca

Figure 5.1: For S = abacabababaaca, in red, $\mathcal{SR}(\texttt{aba}, S)$, and, in blue, $\mathcal{SR}(\texttt{ba}, S)$. The arrows illustrate the fact that the suffixes from $\mathcal{SR}(\texttt{aba}, S)$ skipping $|\texttt{aba}| = 3$ characters occur in increasing order, so we can use binary search to find the ones that end up in $\mathcal{SR}(\texttt{ba}, S)$, as in Theorem 5.1.

Proof. Using Lemma 5.2 it is easy to remove some amount of characters from the end of the pattern: from Lemma 5.1, we know that the suffix range contains the suffix range we had. So we can take any index of the suffix range we had and extend it to find the new suffix range.

To remove some amount of characters from the beginning, we apply a similar strategy. Let P' be the pattern P with the first k characters removed. If $i \in \mathcal{SR}(P, T)$, then $\mathcal{ISA}(\mathcal{SA}[i] + k) \in \mathcal{SR}(P', T)$. Therefore, we can again use Lemma 5.2 and compute the new suffix range. □

Now we are ready to tackle arbitrary character edition in the pattern.

Theorem 5.2. *Let P be a pattern that occurs in the text T. Let P' be P with the i-th character edited (added or deleted). Given $\mathcal{SR}(P, T)$, we can compute $\mathcal{SR}(P', T)$ in $\mathcal{O}(\log |T|)$ time.*

Proof. Assume we want to perform a character addition. From Lemma 5.3, we can compute $\mathcal{SR}(P[0, i), T)$ and $\mathcal{SR}(P[i, |P|), T)$ in $\mathcal{O}(\log |T|)$ time. That is, we split the pattern at the index we want to change. To add some character $c \in \Sigma$, we can then compute $\mathcal{SR}(c, T)$ in $\mathcal{O}(\log |T|)$ time using Algorithm 3.3 (binary search), and then concatenate the suffix ranges as outlined in Theorem 5.1. In the case of a deletion, we just need to concatenate $\mathcal{SR}(P[0, i), T)$ and $\mathcal{SR}(P[i + 1, |P|), T)$. □

Algorithm 5.1. Suffix Range Concatenation.

Input: $|A|, |B|, \mathcal{SR}(A,T), \mathcal{SR}(B,T)$.
Output: $\mathcal{SR}(A \circ B, T)$.
Time Complexity: $\mathcal{O}(\log |T|)$.

$(lb, rb) \leftarrow \mathcal{SR}(B,T)$

$(\ell, r) \leftarrow \mathcal{SR}(A,T)$ ▷ Compute first index of the answer
while $\ell < r$ **do**
  $m \leftarrow \left\lfloor \frac{\ell+r}{2} \right\rfloor$
  if $\mathcal{SA}[m] + |A| = |T|$ or $\mathcal{ISA}[\mathcal{SA}[m] + |A|] < lb$ **then**
    $\ell \leftarrow m + 1$
  else
    $r \leftarrow m$
$first \leftarrow \ell$

$(\ell, r) \leftarrow \mathcal{SR}(A,T)$ ▷ Compute last index of the answer
while $\ell < r$ **do**
  $m \leftarrow \left\lfloor \frac{\ell+r}{2} \right\rfloor$
  if $\mathcal{SA}[m] + |A| = |T|$ or $\mathcal{ISA}[\mathcal{SA}[m] + |A|] < rb$ **then**
    $\ell \leftarrow m + 1$
  else
    $r \leftarrow m$
$last \leftarrow \ell$

if $l = r$ **then**
  return $[0, 0)$ ▷ Empty range: no occurrences
else
  return $[first, last)$

5.2 Dealing with patterns that do not occur

Now we look at the case when the pattern might not occur in the text. What we can do is represent the pattern as a *concatenation* of strings that occur in the text, which we call an *occurrence partition*, see Definition 5.1.

Definition 5.1. Occurrence Partition.

Let P be a pattern and T be a text. An *occurrence partition* of P with respect to T is a sequence of strings $(P_1, P_2, \dots, P_k)$ that partition P, that is, $P = P_1 \circ P_2 \circ \dots \circ P_k$, and satisfies two properties:

1. Occurrence: P_i either occurs as a substring of T or $|P_i| = 1$ and P_i represents a character that does not occur in T.
2. Maximality: $P_i \circ P_{i+1}$ does not occur in T.

We also denote the minimum size of such partition as $|P|_T$.

Note that not all occurrences partitions are minimum, for example, for $T =$ `aabcaba` and $P =$ `abaaba`, (`ab`, `aab`, `a`) is an occurrence partition, but there is another of size two: (`aba`, `aba`).

Lemma 5.4. *Assuming all characters of P occur in T, a pattern P occurs in a text T if, and only if, the occurrence partition of P with respect to T has size one.*

Proof. Follows directly from the definition. □

The idea is then to represent an occurrence partition of the pattern, and, for every string in the partition, maintain its suffix range. When asked to return the number of occurrences of the pattern, we return zero if the size of the partition is greater than one, or the length of the suffix range otherwise, as per Lemma 5.4 (taking care of characters that do not occur in the text).

We can represent the occurrence partition in a balanced binary search tree, since we want to apply modifications, and this will allow us to do that in $\mathcal{O}(\log |P|)$ time.

We are left with the task of maintaining the occurrence partition properties when modifying some character. Turns out that this is easy: given some index to edit, we can find the string from the partition that should be edited (by traversing the binary search tree). After that, we can apply Theorem 5.2, but not merge the suffix ranges when they would result in an empty range (occurrence property). This might break the maximality property, so we might need to apply some concatenations. It turns out that at most 4 concatenations are needed to ensure the maximality property, as per Lemma 5.5.

Lemma 5.5. *Let P be a pattern and $(P_1, P_2, \dots, P_k)$ some occurrence partition of P with respect to the text T. After a character edition in P, we can find an updated occurrence partition using at most 4 concatenations.*

Proof. Assume our edit was in the i-th string of the partition, so we had the partition $(\dots, P_{i-2}, P_{i-1}, P_i, P_{i+1}, P_{i+2}, \dots)$. Following our strategy, we split P_i might be split into

3 parts (in the case of an addition, the middle part represents the new character), say Q_1, Q_2, Q_3. It might be the case that $P_{i-1} \circ Q_1 \circ Q_2 \circ Q_3 \circ P_{i+1}$ occurs in the text (4 concatenations are necessary in this case). But this is the worst-case, since, from the maximality property of the initial partition, we know that $P_{i-2} \circ P_{i-1}$ does not occur in the text, so it can not be the case that $P_{i-2} \circ P_{i-1} \circ S$ occurs, for any string S. The same applies with P_{i+1} and P_{i+2}. □

Let us go through an example. Assume that we have the text $T =$ `cababaa` and the pattern $P =$ `abcaabb`. An occurrence partition of P with respect to T is (`ab`, `c`, `aa`, `b`, `b`). Now assume we want to insert a character `b` in P at index 4, that is, between the two `a`'s. This will turn P into `abcababb`. To do this, we first need to split the string `aa`, so our representation becomes (`ab`, `c`, `a`, `a`, `b`, `b`); then we find the suffix range of the new character `b`, and insert it to our representation: (`ab`, `c`, `a`, `b`, `a`, `b`, `b`). Now we are left with concatenating adjacent strings in the representation (starting from the `b` we inserted, in both directions), to fix the maximality property. After doing that, the final representation will become (`ab`, `cabab`, `b`), so we performed 4 concatenations. We can see that, since `ab` ∘ `c` (concatenation of first and second strings in the initial representation) does not occur in T, then `ab` ∘ `c` ∘ `abab` also does not occur in T.

Theorem 5.3. *Let P be a pattern and let T be the text. Let P' be P with the i-th character edited (added or deleted). If we have the occurrence partition of P, we can find an occurrence partition of P' in $\mathcal{O}(\log |T|)$ time.*

Proof. From Lemma 5.5, we only need to use Theorem 5.1 $\mathcal{O}(1)$ times. Since we can do all necessary operations in the binary search tree used to represent the occurrence partition in $\mathcal{O}(\log |P|) \subseteq \mathcal{O}(\log |T|)$ time, we can edit the pattern in $\mathcal{O}(\log |T|)$ time. □

5.3 Improving pattern search

We now discuss the pattern search operation, that is, set the current pattern as some pattern P given in the input. We assume the current pattern before the operation is empty. Of course, this can be viewed as repeated pattern edition, and therefore can be done in $\mathcal{O}(|P| \log |T|)$ time. But we can also compute it in $\mathcal{O}(|P| + |P|_T \log |T|)$ time.

Lemma 5.6. *Greedily taking each time the largest prefix of the pattern that occurs in the text produces an occurrence partition of size $|P|_T$.*

Proof. Any minimum occurrence partition can be transformed to the partition that the greedy algorithm produces, by repeatedly shifting the positions of the divisions between the strings to the right (from left to right). □

Lemma 5.7. *Algorithm 3.4 runs in $\mathcal{O}(\log|T|+\ell)$ time, if ℓ is the size of the largest prefix of P that occurs in T.*

Proof. The only operation inside the binary search that has nontrivial cost is the *lcp* computation between the pattern and some suffix of the text. But note that the total time that takes is bounded by ℓ, since every time it runs another iteration it is increasing the size of a prefix of P that occurs in T. □

From Lemma 5.7, we can repeatedly run Algorithm 3.4, and each time from its output we can figure out the size of the string to use for our partition. This is running the greedy algorithm, and from Lemma 5.6 we know that this produces a partition of size $|P|_T$, and adding the total cost we get $\mathcal{O}(|P|)$ plus $\mathcal{O}(\log|T|)$ times the size of the partition. Therefore, we can find the occurrence partition in $\mathcal{O}(|P| + |P|_T \log|T|)$ time.

5.4 Substring editions

Substring modifications can be achieved by operations in the binary search tree that represents the occurrence partition. Deletion and transposition of a substring of the pattern can be done by split and join operations in the binary search tree. By similar arguments as before, we see that only a constant number of merges of suffix ranges are needed after each operation.

Substring copying can be achieved using persistent binary search trees in a similar way, since the persistence of nodes and subtrees allows us to "copy" a subtree [29, 13].

5.5 Other pattern search insights

Here we describe other ways we can improve the pattern search operation.

5.5.1 Pattern compression

Using substring copying, we can also apply an interesting optimization for the pattern search. For this, let us assume the $\Sigma = \{\texttt{0}, \texttt{1}\}$. We could reduce $|P|$ binary searches needed to compute the occurrence partition of P by first computing the occurrence partitions of `00`, `01`, `10`, and `11`, and then copying and concatenating these occurrences partitions to generate that of P, two symbols at a time. Therefore, we would only need $4 + \frac{|P|}{2}$ pattern searches and concatenations of suffix ranges to perform the pattern search.

To generalize this strategy for any Σ, let us assume we first do a pattern search on all strings in Σ^k, that is, all $|\Sigma|^k$ strings of length k using symbols from Σ. This has cost $\mathcal{O}(|\Sigma|^k \cdot k \log |T|)$, since each one takes $\mathcal{O}(k \log |T|)$ time. After this, we copy and concatenate their occurrence partitions to get that of P, k characters at a time. This has cost $\mathcal{O}(\frac{|P|}{k} \log |T|)$. Choosing $k = \frac{1}{2} \log_{|\Sigma|} |P|$, we get the time complexity:

$$\begin{aligned}
&\mathcal{O}\left(|\Sigma|^k \cdot k \log |T| + \frac{|P|}{k} \log |T|\right) \\
=&\mathcal{O}\left(|\Sigma|^{\frac{1}{2}\log_{|\Sigma|}|P|} \cdot \frac{1}{2} \log_{|\Sigma|} |P| \cdot \log |T| + \frac{|P|}{\frac{1}{2}\log_{|\Sigma|} |P|} \log |T|\right) \\
=&\mathcal{O}\left(\sqrt{|P|} \cdot \log_{|\Sigma|} |P| \cdot \log |T| + \frac{|P|}{\log_{|\Sigma|} |P|} \log |T|\right) \\
\subseteq&\mathcal{O}\left(|P| + \frac{|P|}{\log_{|\Sigma|} |P|} \log |T|\right).
\end{aligned}$$

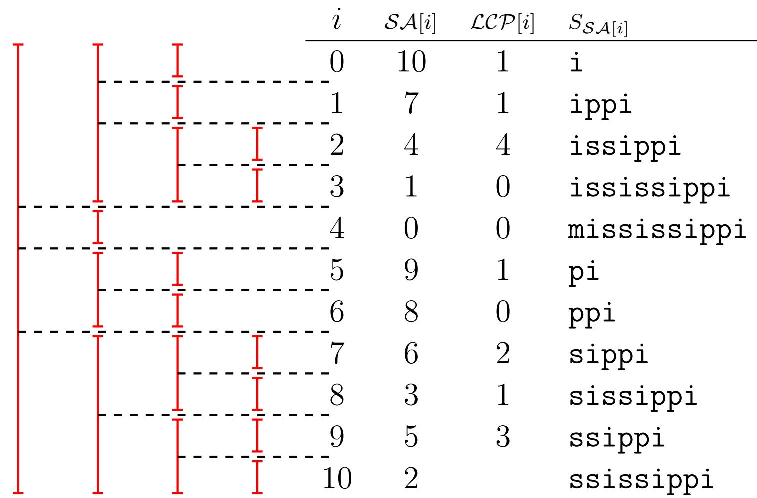
That is, assuming $|\Sigma| \in \mathcal{O}(1)$, we get a speedup of a factor of $\mathcal{O}(\log |P|)$, in relation to the naive approach of performing $|P|$ character editions, taking $\mathcal{O}(|P| \log |T|)$ time.

We can achieve $\mathcal{O}\left(|P| + \min\left(|P|_T, \frac{|P|}{\log_{|\Sigma|}|P|}\right) \log |T|\right)$ time complexity by running the $\mathcal{O}\left(|P| + |P|_T \log |T|\right)$ algorithm until we run more than $\Theta\left(\frac{|P|}{\log_{|\Sigma|}|P|}\right)$ binary searches, and then switch to the $\mathcal{O}\left(|P| + \frac{|P|}{\log_{|\Sigma|}|P|} \log |T|\right)$ algorithm.

Note that this algorithm performs well when the alphabet set Σ is small. In the next section, we will discuss a faster solution for small alphabets.

5.5.2 Small alphabets

It turns out we can, given $\mathcal{SR}(P,T)$, compute $\mathcal{SR}(P \circ \mathtt{c}, T)$ for some character $\mathtt{c} \in \Sigma$ in $\mathcal{O}(\log|\Sigma|)$ time after $\mathcal{O}(|T|)$ preprocessing. First note that $|\{\mathcal{SR}(S,T) \mid S \in \Sigma^*\}| < 2|T|$, that is, we have a linear amount of possible suffix ranges. This is easy to see if we think of the suffix ranges as a tree with $|T|$ leaves, and we assign $\mathcal{SR}(A,T)$ parent of $\mathcal{SR}(B,T)$ if $\mathcal{SR}(A \circ \mathtt{c}) = \mathcal{SR}(B,T)$, for $\mathtt{c} \in \Sigma$. That is, the set of children of a suffix range is the set of ranges it gets split into after adding a character (see Figure 5.2).



i	$\mathcal{SA}[i]$	$\mathcal{LCP}[i]$	$S_{\mathcal{SA}[i]}$
0	10	1	`i`
1	7	1	`ippi`
2	4	4	`issippi`
3	1	0	`ississippi`
4	0	0	`mississippi`
5	9	1	`pi`
6	8	0	`ppi`
7	6	2	`sippi`
8	3	1	`sissippi`
9	5	3	`ssippi`
10	2		`ssissippi`

Figure 5.2: The set of $\mathcal{SR}(S,T)$ for all S and for $T = \mathtt{mississippi}$ organized as a tree, in red.

After computing $\mathcal{SA}(T)$, we can do a depth-first search in this tree by using *lcp* queries: from Lemma 3.3, the indices of the minimum over a suffix range are precisely the positions we need to split on to get the children ranges.

Now we simply store these children ranges for every range, sorted by the character that when added to the string shrinks the range. This can be done for all suffix ranges in $\mathcal{O}(|T|)$ time.

Using this, for every character extension we binary search for the next character in the suffix range in $\mathcal{O}(\log|\Sigma|)$ time. Therefore, the pattern search can be implemented in $\mathcal{O}(|P|\log|\Sigma|)$ time.

5.5.3 Searching for repeated patterns

We can also apply an optimization for repeated patterns: patterns encoded as S^k, the string S concatenated with itself k times. This can be done with binary exponentiation [10], and by using substring copying as outlined before.

Therefore, we can implement it in $\mathcal{O}((|S| + \log k) \log |T|)$ time, that is, in time proportional to $\log |T|$ times the input size.

5.6 Time complexities discussion

The time complexities of our solution for Problem 5.1 are described in Table 5.1, along with the complexities achieved by Amir and Kondratovsky [2] using suffix trees. Our solution for the static text case is considerably simpler, and we improve on the preprocessing time and space and on the substring editions. We also discuss an improvement upon pattern search for small Σ.

Operation	[2]	This work
Preprocess Time and Space	$\mathcal{O}(\|T\|\sqrt{\log \|T\|})$	$\mathcal{O}(\|T\|)$
Pattern Search	$\mathcal{O}(\|P\| \log \|T\|)$	$\mathcal{O}(\|P\| + \|P\|_T \log \|T\|)$ or $\mathcal{O}(\|P\| \log \|\Sigma\|)$
Pattern Symbol Edition	$\mathcal{O}(\log \|T\|)$	$\mathcal{O}(\log \|T\|)$
Pattern Substring Edition	$\mathcal{O}(\log \|T\| + \ell)$	$\mathcal{O}(\log \|T\|)$

Table 5.1: Time bounds for Problem 5.1. ℓ is the size of the modified substring.

5.7 Online text

We can extend our solutions for Problem 5.1 to maintain an *online* text, that is, also support text extensions (on one end). We will assume we want to extend the text on the left.

Problem 5.2. Dynamic Pattern and Online Text Matching.

Input: A string T that represents the text, several updates to the pattern string P that can be one of the following.

1. Pattern search: computing the representation of a pattern;
2. Pattern symbol edition: addition or deletion of some character of the pattern;
3. Pattern substring edition: substring deletion, transposition (moving the substring to another position), or copy.

Also, we support adding a new symbol to the left of the text.

Output: After every update to the pattern, the number of times it occurs in the text.

We can solve Problem 5.2 by using the online suffix array described in Subsection 3.3.1. The problem is that, after a symbol extension in the text, the occurrence partition might lose the maximality property.

But note that this is not a problem: we just need to try to concatenate the first and second elements of the occurrence partition after every operation.

This obviously maintains correctness, because we just need to know if the pattern will occur or not, and this will be the case if, and only if, we are able to join all elements of the occurrence partition into one (Lemma 5.4).

Also note that this does not add to the time complexities of the operations, in an amortized sense. The idea is that the number of joins we make is bounded by the number of splits, and we do a constant number of splits per operation. This is formalized in Theorem 5.4.

Theorem 5.4. *For a text T, Problem 5.2 can be solved with $\mathcal{O}(|T|)$ time construction and $\mathcal{O}(\log |T|)$ amortized time per operation.*

Proof. We will use a potential function to define our amortized complexities [10]. Let D_i be the state of our data structure after operation i, and let $E(D_i)$ be the number of elements of the occurrence partition. Define our potential function $\Phi(D_i) = E(D_i) \cdot \log |T|$. Every operation increases the number of elements by a constant, so this adds an amortized cost of $\mathcal{O}(\log |T|)$ to each operation. If we assume that we then perform k concatenations, we have $\Phi(D_i) - \Phi(D_{i-1}) = c_1 \log |T| - k \cdot \log |T|$, for some constant c_1.

Finally, summing the real cost with the change of our amortized function, we get the amortized cost, for some constant c_2:

$$
\begin{aligned}
& c_2 \log |T| + k \log |T| + (\Phi(D_i) - \Phi(D_{i-1})) \\
= & c_2 \log |T| + k \log |T| + c_1 \log |T| - k \cdot \log |T| \\
= & c_2 \log |T| + c_1 \log |T| \\
\in & \mathcal{O}(\log |T|)
\end{aligned}
$$

□

The time complexities for Problem 5.2 are summarized in Table 5.2.

Operation	[2]	This work
Preprocess Time and Space	$\mathcal{O}(\|T\|\sqrt{\log \|T\|})$	$\mathcal{O}(\|T\|)$
Pattern Symbol Edition	$\mathcal{O}(\log \|T\|)$	$\mathcal{O}(\log \|T\|)^\star$
Pattern Substring Edition	$\mathcal{O}(\log \|T\| + \ell)$	$\mathcal{O}(\log \|T\|)^\star$
Text Symbol Extension	$\mathcal{O}(\log \|T\|)$	$\mathcal{O}(\log \|T\|)^\star$

Table 5.2: Time bounds for Problem 5.2. The $\star$ symbol represents amortized bounds, and ℓ is the size of the modified substring.

Chapter 6

Experimental Results

Although the goal of this work is to find good asymptotic bounds for pattern matching problems, in order to illustrate the feasibility of the proposed solution, it was implemented in C++ in a public repository[1], and was tested with random test data against a naive solution.

In this section we reference two algorithms for maintaining a dynamic pattern with matching information:

- Fast Algorithm: the algorithm described in Chapter 5, that maintains a dynamic pattern and supports updates in $\Theta(\log n)$ time for a text of size n;
- Naive Algorithm: the indexing algorithm described in Subsection 3.2.1, that supports updates in linear time on the size of the current pattern and queries in $\Theta(\ell + \log n)$, if ℓ is the largest prefix of the current pattern that occurs in the text.

We could not compare with the solution from Amir and Kondratovsky [2], since they did not provide an implementation for their algorithm.

6.1 Implementation decisions

We implemented the algorithm to maintain matching information over a pattern subject to character editions (insertions and deletions). For the binary search tree, we chose to implement a Treap [30], a randomized tree that achieves $\Theta(\log n)$ time for SPLIT and JOIN operations for n nodes, with high probability.

[1] https://github.com/brunomaletta/DynamicPatternMatching

6.2 Worst case scenario

The naive solution takes linear time on the pattern for each update, and for each query it takes $\Theta(\ell + \log n)$ time, if n is the size of the text and ℓ is the largest prefix of the pattern the occurs in the text. So, for random data, it performs much quicker than the worst case of linear time on the pattern per query.

Therefore, to force the worst case, we generate instances with all equal characters, so the patterns always occurs in the text. In this way, each query on the naive algorithm takes linear time on the pattern size. This does not significantly impact the running time of the fast algorithm.

Moreover, we also generate cases in which both the text and the pattern have similar sizes, that is, the pattern is very large.

We understand that these circumstances are not reasonable to the real world, but the point of this analysis is to illustrate that the constant of the algorithm is not too large as to have no real world applications.

6.3 Results and discussion

The running time of the fast algorithm was greater than what was expected, but this might be explained by a somewhat large constant factor. We only do simple binary searches, but for adding a character we do up to 12 binary searches: each time we do 2, one for the beginning and one for the end of the suffix range, and we do 1 split, 1 character search and up to 4 concatenations (see Lemma 5.5), totaling $2 \cdot (1 + 1 + 4) = 12$.

Even so, with n operations over a text of size n, with n in the order of one million, we see considerable gains in relation to the naive solution (see Figure 6.1 and Table 6.1).

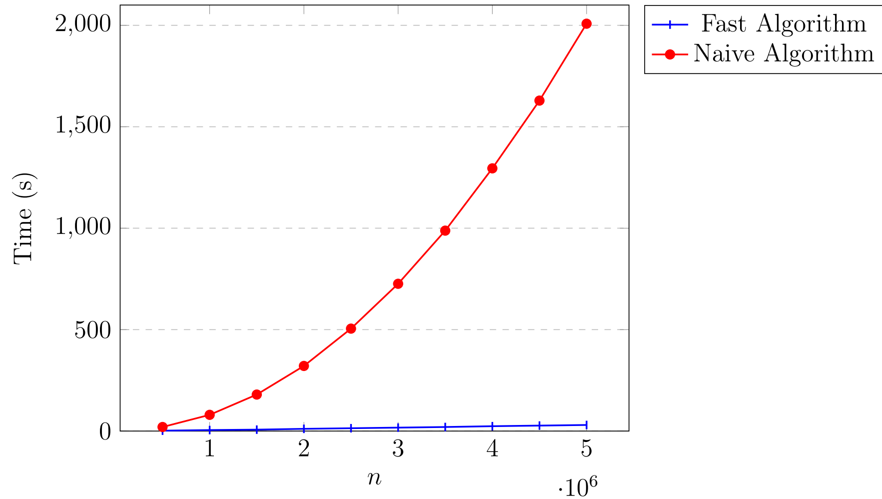


Figure 6.1: Graph representing the average running time of 5 rounds to execute n pattern updates with a text of size n, for both algorithms.

n	Fast Algorithm time (s)	Naive Algorithm time (s)
$0.5 \cdot 10^6$	2	20
$1.0 \cdot 10^6$	5	80
$1.5 \cdot 10^6$	7	180
$2.0 \cdot 10^6$	11	321
$2.5 \cdot 10^6$	14	505
$3.0 \cdot 10^6$	17	726
$3.5 \cdot 10^6$	20	988
$4.0 \cdot 10^6$	24	1295
$4.5 \cdot 10^6$	27	1629
$5.0 \cdot 10^6$	30	2008

Table 6.1: Average time of 5 rounds to execute n pattern updates with a text of size n, for both algorithms.

Chapter 7

Conclusion

In this thesis we discussed the String Matching Problem and several variations. For each variation, well-known solutions were shown, along with their time complexity analysis.

We described the Suffix Array and how to compute it in linear time, along with the Longest Common Prefix Array. We went through several properties of the Suffix Array, and how to use it to solve the Indexing Problem. Some variations were also discussed.

A generalization of the Suffix Array was then proposed: sorting strings defined by a trie as the reverse of the path from the root to each node. We adapted algorithms from the literature to solve this generalization efficiently.

Equipped with this understanding of the Suffix Array, we tackled the *modified pattern reporting problem* defined by Amir and Kondratovsky [2]. We improved upon their algorithm, with a simpler and faster algorithm – our preprocess time is linear, and we can support substring editions in sub-linear time (see Table 5.1).

In addition, we apply our algorithm for an online text, achieving amortized logarithmic bounds (see Table 5.2). The paper by Amir and Kondratovsky [2] seems to claim worst-case logarithmic bounds, but we could not understand if this is really the case, and we could not get in touch with the authors, despite our best efforts.

Besides the operations we discussed, it is easy to see that we can also insert a new substring in an arbitrary position of the pattern by doing a pattern search on the new substring, and then insert it in $\mathcal{O}(\log |T|)$ time, if T is the text.

Another operation we can easily support is substring reversing: we always maintain the representation of the reversed pattern, and when asked to reverse a substring, swap it with its reversed in the reversed representation. This takes $\mathcal{O}(\log |T|)$ time.

In practice, the algorithm performed orders of magnitude faster than the naive algorithm (in the worst case) even for strings with size in the order of one million (see Figure 6.1), suggesting its practicability.

It is interesting to see a number of scenarios in which we can perform a faster pattern search (computing the representation of a pattern, that can later be updated in logarithmic time) of a pattern P on a text T, compared to the standard $\mathcal{O}(|P| \log |T|)$ worst-case time algorithm:

- If large sections of the pattern occur in the text: the search is computed in $\mathcal{O}(|P| +$

$|P|_T \log |T|)$ time, with $|P|_T$ being the size of the smallest *occurrence partition* of P with respect to T (see Definition 5.1). In particular, if P occurs in T, the search is computed in $\mathcal{O}(|P| + \log |T|)$ time.

- If the alphabet set Σ is small: the search is computed in $\mathcal{O}(|P| \log |\Sigma|)$ time. This requires some more preprocessing, but the whole preprocessing still takes $\mathcal{O}(|T|)$ time. In particular, if $|\Sigma| \in \mathcal{O}(1)$, the search is computed in $\mathcal{O}(|P|)$ time.
- If the pattern has several adjacent equal characters: instead of the cost of $\mathcal{O}(\log |T|)$ per character of the pattern, we can search for chunks of k equal characters in $\mathcal{O}(k + \log k \log |T|)$ time. This means that, if P can be broken up into t chunks of equal characters, we can compute the search in $\mathcal{O}(|P| + t \log |P| \log |T|)$ time. In particular, we can search for $P = c^k$ in $\mathcal{O}(\log k \log |T|)$ time, for $c \in \Sigma$.

7.1 Future work

Future work on this problem might include trying to support more drastic updates to the text, such as removing characters from one end, or even supporting updates to both ends of the text.

One important case is that of $\Sigma \in \mathcal{O}(1)$, which is usually the case in practical applications. We did not delve into this case much, but several optimizations seem possible, that might speed up some operations we defined.

It is interesting to think of applications of the Generalized Suffix Array as we defined it. One of such might be to compute substring information about a changing string that is updated in one of its ends (adding and removing characters). We can solve this problem offline by defining a trie using the updates, and computing its Generalized Suffix Array.

We also have hope to come up with a faster solution for the *modified pattern reporting problem*: given $\mathcal{SR}(A, T)$ and $\mathcal{SR}(B, T)$, it is possible to compute $\mathcal{SR}(A \circ B, T)$ in $\mathcal{O}(\log \log |T|)$ using the framework developed by Bille et al. [6]. However, we would need a way to expand the suffix range (as in Lemma 5.2) faster than $\mathcal{O}(\log |T|)$ time. A van Emde Boas tree [31] might be useful, instead of a binary search tree, to store the occurrence partition, in the hope of implementing the operations in $\mathcal{O}(\log \log |P|)$ instead of $\mathcal{O}(\log |P|)$.

Other improvement would be to get worst case logarithmic time bounds for the *modified pattern reporting problem* with an online text (Problem 5.2), as opposed to our amortized logarithmic time bounds.

References

[1] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Amihood Amir and Eitan Kondratovsky. Searching for a modified pattern in a changing text. In *International Symposium on String Processing and Information Retrieval*, pages 241–253. Springer, 2018.

[3] Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *International Symposium on String Processing and Information Retrieval*, pages 67–78. Springer, 2005.

[4] Michael A Bender and Martin Farach-Colton. The lca problem revisited. In *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000 Proceedings 4*, pages 88–94. Springer, 2000.

[5] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Algorithms—ESA 2002: 10th Annual European Symposium Rome, Italy, September 17–21, 2002 Proceedings*, pages 152–164. Springer, 2002.

[6] Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80 (11):3207–3224, 2018.

[7] Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007.

[8] Richard Cole and Moshe Lewenstein. Multidimensional matching and fast search in suffix trees. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 851–852, 2003.

[9] Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: structures for faster text indexing. In *International Colloquium on Automata, Languages, and Programming*, pages 358–369. Springer, 2006.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2022.

[11] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 365–372, 1987.

[12] Paul F Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127, 1982.

[13] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[14] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[15] Paolo Ferragina and Roberto Grossi. Optimal on-line search and sublinear time update in string matching. *SIAM Journal on Computing*, 27(3):713–736, 1998.

[16] Michael J Fischer and Michael S Paterson. String matching and other products. In *Complexity of Computation, RM Karp (editor), SIAM-AMS Proceedings*, volume 7, pages 113–125, 1974.

[17] Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '94, page 697–704, USA, 1994. Society for Industrial and Applied Mathematics. ISBN 0898713293.

[18] AG Ivanov. Distinguishing an approximate word's inclusion on turing machine in real time. *Izv. Acad. Nauk USSR Ser. Mat*, 48:520–568, 1984.

[19] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003.

[20] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM*, volume 2089, pages 181–192. Springer, 2001.

[21] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2-4):126–142, 2005.

[22] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[23] Gad M Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[24] Felipe A Louza, Simon Gog, and Guilherme P Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678:22–39, 2017.

[25] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[26] Bruno Monteiro. Dynamic suffix arrays. https://codeforces.com/blog/entry/93042, 2021. Access date: 2024-08-18.

[27] James Morris Jr and Vaughan Pratt. *A linear pattern-matching algorithm*. University of California, Berkeley, 1970.

[28] nor. Binary search and other “halving”' methods. https://nor-blog.pages.dev/posts/2021-11-07-binary-search/, 2021. Access date: 2024-08-19.

[29] Neil Sarnak and Robert E Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[30] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16 (4):464–497, 1996.

[31] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical systems theory*, 10(1):99–127, 1976.

[32] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.