ESCUELA UNIVERSITARIA DE INFORMÁTICA

EXACT STRING PATTERN RECOGNITION

Francisco Gómez Martín

COURSE NOTES

October 2009 - January 2010

Contents

Acknowledgements

I want, above all, to thank my wife, Lola. This project has been carried out against the elements, including strong winds of skepticism, and she has strongly supported me on this from the very first day. With her tireless, thorough and strict reviews of the drafts she taught me that enlightening, sound thoughts are always held in tight, succinct sentences. Easy to understand, hard to master.

I thank my son Paquito because, believe it or not, he gave me emotional support.

I want to express my gratitude to my colleague Blanca Ruiz. We enjoyed sharing office during the writing of these notes. She was always willing to proof-read several drafts of these notes. I found her comments relevant, pertinent and sharp.

I thank my students for spotting mistakes in the early versions of these notes.

I also wish to thank the Universidad Politécnica de Madrid for giving this project financial support.

Last in sequence, but not last in importance, there is Vanessa, who made a wonderful review of my English.

Preface

These notes have been written for the course on exact string pattern recognition that I teach at *Escuela Universitaria de Informática* (Universidad Politécnica de Madrid). The course is intended for sophomores with a small background in Data Structures and Algorithms. In fact, they are taking more advanced courses on those subjects while taking this course. Therefore, the material is tailored to their level.

Certainly, the material could have been covered more in depth, but my intention was, above all, to motivate them. With little, self-contained and rigourously presented material, I tried to get the students to gain hands-on experience, both in theory –design and formal proof of algorithms, complexity– as well as in practice –actual programming, design of test data, running experiments–. The first chapter is a review of algorithms and complexity. It follows a chapter introducing the basic problems in string pattern recognition; a brief historical account of the invention of the main algorithms is also given. The next few chapters constitute the core of the notes. The main fundamental string matching algorithms are covered. After the core chapters, there is a chapter on English phonetics. My student's level of English was unexpectedly low and I had to give a few lessons on phonetics to help them understand me properly. Therefore, I decided to write this part up in order to help them.

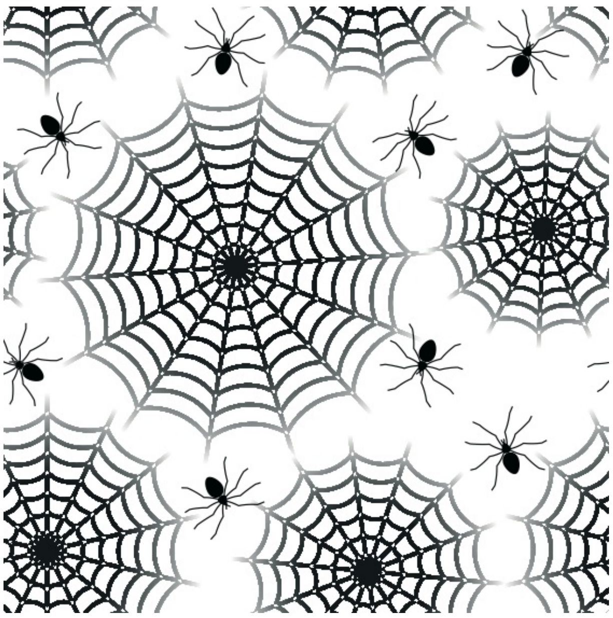
I always treat my students as rational beings, whatever the cost of such attitude is. Some of them are scared away by my stance. Treating somebody as a rational being is treating him as a free being. By giving students freedom, which they deserve as rational beings, I gave them the possibility and the responsibility of choosing their own way. This freedom, which many are not used to whatsoever, can only be won through the effort of thinking in the acknowledgement of your own ignorance. Teaching is not about guiding students with a firm arm through a trimmed trail, familiar to the teacher but full of weeds of preconceived ideas; no, certainly not, it is about pointing out those trails they may take with the sole companionship of their effort, creativity and freedom. That is what teaching is about. I am deeply convinced that a good teacher must be, among other things, an agitator. Being paternalistic is not a good policy for it relents learning. I flame the eternal dialectic between teacher and students through irony and provocation. Some time ago I started to keep a course blog where I reflected what happened in the classroom (in a somewhat literary way). Sometimes exchanges of replies and counter-replies between students and teacher are profoundly revealing and I thought it would be a good idea to write them down. The last chapter is the blog of this year's course. Student's names have been anagrammatized to preserve their privacy.

The writing conventions in these notes are the following. String of characters are written with `typewriter font`. Except for very technical ones, definitions will be not formally stated. Definitions will be introduced as needed and will be noted by using **boldface**. Theorems and lemmas will be stated formally. Pseudocode will be written by using CAPITAL LETTERS.

Paco Gómez Martín.
February 2010.

Chapter 1

Algorithms and Complexity



1.1 Algorithms

Algorithms are at the core of Computer Science and, in particular, of string pattern recognition. However, the word "algorithm" does not possess a generally accepted definition. For the purposes of these notes, we will adopt an informal definition, which will be flexible enough to describe the solutions to problems in string pattern recognition but not so formal as to be lengthy and unfriendly. An **algorithm** is a procedure that takes an **input** and, after performing a series of well-defined computational operations, produces an **output**. Knuth [Knu73] has identified five properties –widely accepted– that an algorithm should have:

1. **Input:** The initial values or input instance of the problem.
2. **Definiteness:** Every algorithm must be precisely defined so that there is no ambiguity.
3. **Finiteness:** An algorithm must terminate after a finite number of steps.

4. **Output:** An algorithm must produce a set of final values out of the input.

5. **Efectiveness:** All the operations must be sufficiently basic.

Berlinski in his year-2000 book [Ber00] provides a snappy, humoresque definition of an algorithm:

> "**In the logician's voice:**
> "an algorithm is
> a finite procedure,
> written in a fixed symbolic vocabulary,
> governed by precise instructions,
> moving in discrete steps, 1, 2, 3, . . .,
> whose execution requires no insight, cleverness,
> intuition, intelligence, or perspicuity,
> and that sooner or later comes to an end."

There are many ways of expressing algorithms, ranging from descriptions in natural language to programming language. In these notes, algorithms will be described in **pseudocode** so that we can focus on the ideas to solve a given problem rather than on implementation details. The pseudocode we will use here is very much like C, Pascal, Maple or MatLab. The actual implementation of algorithms will be taken care of in the programming classes of this course.

Two important points are associated with algorithms: **correction** and **complexity**. Given a computational problem, an algorithm is said to be **correct** if, for every input instance, it produces a solution to the problem (that is, a correct output). In such case we say that the algorithm **correctly solves** the problem. Correction of algorithms is typically proved through mathematical techniques such as induction and the like. **Complexity** refers to the time and the space an algorithm needs to be run.

One of the most common problems that arises in computer science is the **sorting problem**. This problem can be stated as follows:

THE SORTING PROBLEM

Input A sequence of real numbers $\{x_1, x_2, \ldots, x_n\}$.

Output A reordering $\{x_{i_1}, x_{i_2}, \ldots, x_{i_n}\}$ of the input sequence holding the property $x_{i_1} \leq x_{i_2} \leq \ldots \leq x_{i_n}$.

If we are given the sequence $\{17, 8, 23, 12, 5, 19, 17, 11\}$, a correct sorting algorithm should return $\{5, 8, 11, 12, 17, 19, 23, \}$ as output. The input sequence is called an **instance** of the sorting problem. An algorithm is proven to be correct when it returns the correct output for *every* instance of the problem.

Let us consider perhaps the simplest sorting algorithm that can be possibly thought of: the **insertion sort**. The general idea of the algorithm is to insert a number into the

```
   INSERTION-SORT(M)
1  for $j \leftarrow 2$ to $n$
2    do $key \leftarrow M[j]$
3      $i \leftarrow j - 1$
4      while $i > 0$ and $M[i] > key$
5        do $M[i+1] \leftarrow M[i]$
6           $i \leftarrow i - 1$
7      $M[i+1] \leftarrow key$
```

Table 1.1: The *INSERTION-SORT* algorithm.

correct position in a previously sorted sequence of numbers. The behavior of this algorithm resembles the way a card player sorts his hand. Receiving the last card, he puts it in place by comparing it with each to the cards already in the hand.

Our pseudocode for insertion sort is presented below. It takes as a parameter an array $M[1..n]$ –the input– containing n numbers to be sorted. The output will be also stored within the array M. The numbers will be output in nondecreasing order.

At the step j of the algorithm, subarray $M[1..j-1]$ contains the currently sorted numbers, whereas the remaining non-sorted numbers are found in the subarray $M[j..n]$. Then, number $M[j]$ is compared with the numbers in $M[1..j-1]$. In order to keep the value of $M[j]$, the current number to be compared, is necessary to make the assignment $key \leftarrow M[j]$. The comparisons are carried out in the loop given by lines 4-7 from number $M[j-1]$ down to $M[1]$. If a number $M[i]$ is greater than *key*, then that number is moved up a position in the array M. When the algorithm finds the first position such that $M[i]$ is less or equal than *key*, then *key* is inserted there (line 7).

1.2 Correctness of Algorithms

We now turn to the tricky question of correctness of algorithms. How to prove that an algorithm is correct? In other words, how can we make certain that for *every* input instance the algorithm will return a correct output? The right way to prove algorithm is through a mathematical proof, often an induction proof, but always a constructive proof (as o-pposed to proofs by contradiction). For example, how can we guarantee that the algorithm *INSERTION-SORT* above will actually sort any sequence of real numbers? Here there is a theorem that provides such a proof.

Theorem 1.2.1 *INSERTION-SORT correctly sorts a sequence of real numbers.*

Proof: At the j-th step the algorithm keeps a property (the invariant) that subarray $M[1..j-1]$ is already sorted. We will prove by induction that such property is always true along the execution of the algorithm.

Base case: Let n be 2. Array M consists of two elements, $M[1]$ and $M[2]$. Let us first assume that $M[1] \leq M[2]$. The loop starting at line 1 is only executed once. The variable *key* contains the number $M[2]$. Because $M[1] \leq M[2]$, the loop $4-6$ is not executed and the algorithm terminates after the assignment $M[2] \leftarrow key$, which is correct. Let us now assume that $M[1] > M[2]$. In this case the loop $4-7$ is executed once. In line 5 the assignment $M[2] \leftarrow M[1]$ takes place. Again, line 7 sets $M[1]$ to *key*, which is the value of $M[2]$ before entering in the loop $4-6$. Therefore, the numbers of array M have been swapped, and the algorithm correctly sorts any two-element array.

Inductive case: Consider now an array of length $n > 2$. Assume that at the j-th step the subarray $M[1..j-1]$ is correctly sorted. In line 2 the value $M[j]$ is stored in variable *key*. Let k be the smallest number such that every element in $M[k..j-1]$ is greater than $M[j]$. The loop in lines $4-6$ locate that k and then copy subarray $M[k..j-1]$ onto subarray $M[k+1..j]$. Finally, line 7 assigns *key* to $M[k-1]$. Notice that k is equal to i after the while loop is performed. Therefore, after lines $2-7$ are executed at step j, subarray $M[1..j]$ is sorted. After the execution of the for loop, the whole array M is sorted. ▮

Notice that proving that an algorithm is correct may require a considerable dose of rigor. The award, of course, is being sure our algorithms will work. In this course we will make special emphasis upon correctness of algorithms.

1.3 Analysis of Algorithms

In practice the problem of choosing an algorithm to solve a given problem appears very often. There are many criteria to make such a choice, depending on the specific situation. As it is quite natural to seek fast algorithms, the **analysis of running time**, also called **analysis of time complexity**, is the most used and studied, both in theory and practice. **Space complexity** analysis or the analysis of the amount of memory required by an algorithm is also common.

Analyzing an algorithm is to assign costs to its basic operations and compute how much its execution costs. As a matter of fact, different inputs will produce different costs. A **model of computation** is a specification of the primitive operations and their respective costs. The primitive operations are those operations of fixed cost. The model of computation that we will use in this course is the **real RAM**. For this model the following operations have constant cost:

1. The arithmetic operations: $+, -, x, /$.
2. Comparisons between two real numbers: $<, >, \leq, \geq, =, \neq$.
3. Indirect addressing of memory.
4. Assignment of numbers to variables.

This is a high-level model of computation. It focuses more on high-level operations than on specific programming details. The reason to choose this model is that in this course we will study the general ideas and techniques behind the design of algorithms.

We illustrate the analysis of algorithms by analyzing the algorithm *INSERTION-SORT*. Below we revisit the pseudocode of the algorithm to which we have added the cost and the number of times each sentence is executed.

```
   INSERTION-SORT(M)                           Cost    Time
1  for $j \leftarrow 2$ to $n$                 $c_1$   $n-1$
2    do $key \leftarrow M[j]$                  $c_2$   $n-1$
3      $i \leftarrow j-1$                      $c_3$   $n-1$
4      while $i > 0$ and $M[i] > key$          $c_4$   $\sum_{j=2}^{n} p_j$
5          do $M[i+1] \leftarrow M[i]$         $c_5$   $\sum_{j=2}^{n}(p_j-1)$
6             $i \leftarrow i-1$               $c_6$   $\sum_{j=2}^{n}(p_j-1)$
7      $M[i+1] \leftarrow key$                 $c_7$   $n-1$.
```

Table 1.2: Computing the time complexity of *INSERTION-SORT*.

The total running time is computed by summing all the running times for each statement executed. The numbers c_i in the pseudocode are the costs of each statement. A statement executed n times will contribute $c_i n$ to the total running time. If we let $T(n)$ be the total running time of *INSERTION-SORT*, then we obtain:

$$\begin{aligned} T(n) &= c_1(n-1)+c_2(n-1)+c_3(n-1)+ \\ &\quad c_4\sum_{j=2}^{n} p_j + c_5\sum_{j=2}^{n}(p_j-1)+c_6\sum_{j=2}^{n}(p_j-1)+c_7(n-1). \end{aligned} \tag{1.1}$$

In the previous equation the c_i's are constant by virtue of our choice of the model of computation, the real RAM. However, the p_j's are unknown numbers that vary with the particular input. If the algorithm receives the input in nondecreasing order, the loop in lines 4-6 will be executed just once, and hence $p_j = 1$, for $j = 2, \ldots, n-1$. In this case, the time complexity $T(n)$ becomes a linear function of n:

$$\begin{aligned} T(n) &= c_1(n-1)+c_2(n-1)+c_3(n-1)+c_4(n-1)+c_7(n-1) \\ &= n(c_1+c_2+c_3+c_4+c_7)-(c_1+c_2+c_3+c_4+c_7). \end{aligned} \tag{1.2}$$

If the input data are given in reverse sorted order, then the loop in lines $4-6$ is performed j times, that is, now $p_j = j$, for $j = 2, \ldots, n-1$. By making use of the formula to sum the first n natural numbers

$$\sum_{i=1}^{n} j = \frac{n(n+1)}{2},$$

we can work out the equations for $T(n)$:

$$\begin{aligned}
T(n) &= c_1(n-1)+c_2(n-1)+c_3(n-1)+c_4\left(\frac{n(n+1)}{2}-1\right)+ \\
&\quad c_5\frac{n(n-1)}{2}+c_6\frac{n(n-1)}{2}+c_7(n-1) \\
&= \frac{n^2}{2}(c_4+c_5+c_6)+n\left(c_1+c_2+c_3+\frac{c_4}{2}-\frac{c_5}{2}-\frac{c_6}{2}+c_7\right) \\
&\quad -(c_1+c_2+c_3+c_4+c_7). \qquad (1.3)
\end{aligned}$$

As we can see from equations 1.2 and 1.3, the time complexity of *INSERTION-SORT* may range from a linear function to a quadratic function. The first situation, when the input is already sorted, is called the **best case**, while the second situation, when the input is given in reverse sorted order, is the **worst case**. This example illustrates how variable the behavior of an algorithm may be. The two main analysis of time complexity are the **worst-case analysis**, which will be the most used in this course, and the **average-case analysis**. The worst-case analysis estimates the longest running for *any* input of size n. In general, the worst-case analysis provides an upper bound for the running time of the algorithm. This upper bound may be conservative if the worst case occurs rarely, or it may be quite tight if the worst case occurs fairly often. The **average-case analysis** implies a probabilistic analysis of the average running time of the algorithm. In general, the average-case analysis is harder than the worst-case, as it involves more advanced mathematical techniques.

1.4 Asymptotic Notation

Once the time complexity of an algorithm is computed, we obtain a positive function $T(n)$ of n. Comparing the time complexity of two algorithms, for example, to choose the fastest, is equivalent to comparing their associated functions. In this section we will introduce (rather than review) the asymptotic notations O, Ω and Θ. The asymptotic notation provides a tool to effectively compare the time complexity of algorithms. In the following, $f(n)$ and $g(n)$ will be two positive functions defined over the natural numbers.

Definition 1.4.1 *$O(f(n))$ is the set of functions defined by:*

$$O(f(n)) = \{g(n) \mid \exists c > 0, n_0 > 0 \mid 0 \leq g(n) \leq cf(n), \forall n \geq n_0\}$$

Figure 1.1(a) illustrates the definition above. A function $g(n)$ belongs to $O(f(n))$ when there exists two constants $c, n_0 > 0$ such that, for $n > n_0$, the inequality $g(n) \leq cf(n)$ holds. The constants n_0 and c depend on the particular functions $f(n)$ and $g(n)$. Let us work out an example in order to gain some insight into the O-notation.

Example. Let $f(n) = n^2$ and $g(n) = 3n + 5$ be two positive functions defined over $\mathbb{N}$. Let us prove that $g(n) \in O(f(n))$. For now, we will apply the definition of $O(f(n))$ to the letter to prove it; later on we will give a result that reduces that proof to a simple limit calculation. According to the definition, we must find two constants c, n_0 such that $3n + 5 \leq cn^2$.

$$3n + 5 \leq cn^2 \Longrightarrow \frac{3}{n} + \frac{5}{n^2} \leq c.$$

If c is set to 1, then that inequality is true for $n > 6$, since in this case $\frac{3}{n}, \frac{5}{n^2}$ are both less than $\frac{1}{2}$. Therefore, it suffices to take $n_0 = 6$.

If $g(n)$ belongs to the class $O(f(n))$, then $f(n)$ is called an **asymptotic upper bound** for $g(n)$. In the analysis of algorithms, asymptotic upper bounds are given to estimate the worst-case running time, and algorithms are classified according to their upper bounds. For example, **linear algorithms** are those whose worst-case complexity is a function in $O(n)$. An algorithm whose time complexity is $O(n \log n)$ is preferable to a quadratic algorithm (with time complexity in $O(n^2)$).

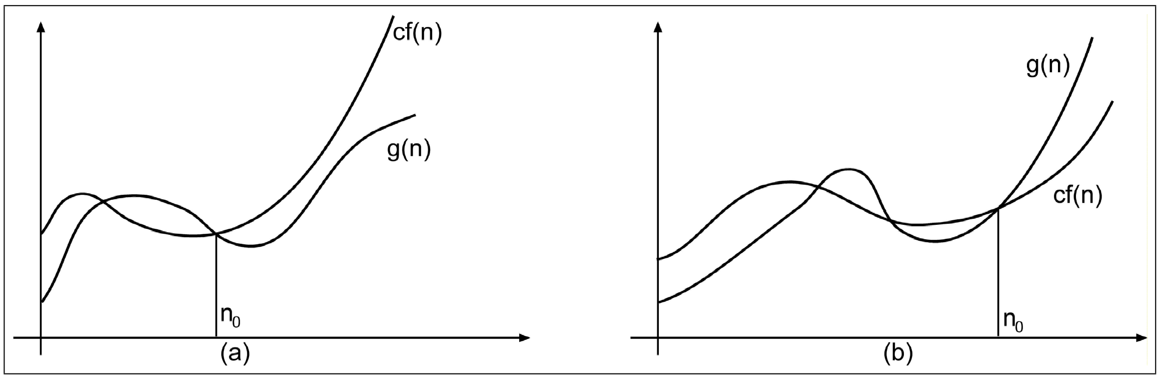


Figure 1.1: The class of functions O and Ω.

The following class of functions provides another useful concept, the **asymptotic lower bound**, which, when applied to the analysis of algorithms, refers to the minimum amount of time that it takes the algorithm to solve the problem. Figure 1.1(b) shows the geometrical interpretation of the definition.

Definition 1.4.2 $\Omega(f(n))$ *is the set of functions defined by:*

$$\Omega(f(n)) = \{g(n) \mid \exists c > 0, n_0 > 0 \mid 0 \leq cf(n) \leq g(n), \forall n \geq n_0\}$$

Example. It is easy to prove that $n \in \Omega(n^k)$, for $0 < k \leq 1$. Furthermore, n^l belongs to $\Omega(n^k)$, for $k \leq l$. In general, exponential functions are in $\Omega(n^k)$, for $k > 0$.

Finally, we introduce the **asymptotically tight bound**, which is a class of functions defined as the intersection of $O(f(n))$ and $\Omega(f(n))$. Figure 1.2 gives an intuitive picture of the definition.

Definition 1.4.3 $\Theta(f(n))$ *is the set of functions defined by:*

$$\Theta(f(n)) = \{g(n) \mid \exists c_1, c_2 > 0, n_0 > 0 \mid 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n), \forall n \geq n_0\}$$

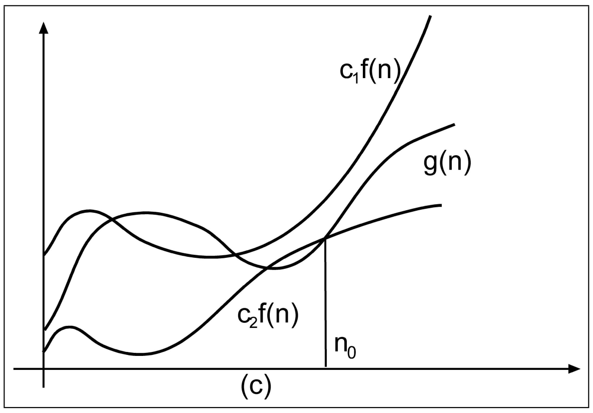


Figure 1.2: The class of functions Θ.

Example. Let $f(n) = n^2$ and $g(n) = \frac{n^2}{2} - 3n + 5$ be two polynomials. We shall prove that $g(n) \in \Theta(f(n))$, again by applying the definition.

$$c_1 n^2 \leq \frac{n^2}{2} - 3n + 5 \leq c_2 n^2 \Longrightarrow c_1 \leq \frac{1}{2} - \frac{3}{n} + \frac{5}{n^2} \leq c_2.$$

If n is set to a value greater than 6, then $\frac{3}{n} > \frac{1}{2}$. This implies that c_1 can safely be set to $\frac{5}{36}$. As for c_2, if $n > 6$, the expression $\frac{1}{2} - \frac{3}{n} + \frac{5}{n^2}$ is always less than 1. Hence, $c_1 = \frac{5}{36}, c_2 = 1$ and $n_0 = 6$.

Coming back to the algorithm *INSERTION-SORT*, if $T(n)$ represents its worst-case complexity, we may state that $T(n) \in O(n^2)$ and $T(n) \in \Omega(n)$. However, we can state that neither $T(n) \in \Theta(n)$ nor $T(n) \in \Theta(n^2)$.

Using the definition of asymptotic notation to classify functions is rather cumbersome, as we have seen from the examples. The following theorem states the necessary and sufficient conditions to classify functions into the $O-, \Omega-$ and Θ-classes and reduces such a classification to a limit calculation.

Theorem 1.4.4 *Assume that the limit* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L \in \mathbb{R}$ *does exist. Then, the following properties are true.*

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = L < \infty \iff g(n) \in O(f(n)) \tag{1.4}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L < \infty \iff g(n) \in \Omega(f(n)) \tag{1.5}$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in L \in \mathbb{R} - \{0\} \iff g(n) \in \Theta(f(n)) \tag{1.6}$$

Proof: See the exercises.

Example. Let us prove that $\log n \in O(n^k)$, for $k > 1$. It is enough to carry out the following calculation (L'Hôpital's rule is used):

$$\lim_{n\to\infty} \frac{\log n}{n^k} = \lim_{n\to\infty} \frac{1/n}{n^{k-1}} = \lim_{n\to\infty} \frac{1}{n^k} = 0.$$

Therefore, by Theorem 1.4.4, function n^k is an asymptotic upper bound for function $\log n$.

Example. Given two polynomials $P_k(n), Q_k(n)$ of the same degree k, it holds that $P_k(n) \in \Theta(Q_k(n))$. The proof consists of just computing the limit $\lim_{n\to\infty} \frac{P_k(n)}{Q_k(n)}$, which is a finite number equal to the ratio of the leading coefficients of both polynomials.

Exercises

$\boxed{1}$ **Complexity of two searching algorithms. Complexity and experiments.** Consider the algorithms linear search and binary search in an array. Compare their running times by using asymptotic notation. Also, perform experiments to compare the performance of both algorithms. Plot your results.

$\boxed{2}$ **Choice of constants.** Make a reasonable assignment of constants c_i in the algorithm INSERTION-SORT and give a closer form of its best-case and worst-case running times.

$\boxed{3}$ **The mergesort algorithm.** Consider the algorithm *MERGE-SORT* to sort n real numbers. Prove its correctness and analyze its time complexity. If you do not know this algorithm, look it up on the web or in the bibliography.

$\boxed{4}$ Sort in increasing order of complexity the following set of functions: $\{\log n, n, 1, n \log n, 2^n, n^n, n^2, n!, n^3\}$. Use asymptotic notation.

$\boxed{5}$ **Running times matter**. For each function $T(n)$ and time t in the table below, determine the largest size n of a problem that can be solved in time t. We will assume that $T(n)$ is measured in microseconds. For this problem you should use a mathematical package, such as Derive or Maple, to solve the equations.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\log n$							
$\sqrt{n}$							
n							
$n \log n$							
n^2							
n^3							
2^n							
$n!$							

$\boxed{6}$ Prove that $f(n) = \sum_{i=1}^{n} i$ is in $\Theta(n^2)$ without using the explicit formula given in the text.

$\boxed{7}$ Let $T_1(n) = 100n^2$ and $T_2(n) = 2^n$ be two time complexities of two algorithms. What is the smallest value of n such that the first algorithm runs faster than the second algorithm?

$\boxed{8}$ Prove that $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$ by using the definition of Θ-notation.

$\boxed{9}$ **A mathematical problem.** Prove Theorem 1.4.4

1.5 Chapter Notes

As said at the outset, there is no generally accepted definition of algorithm. The definition presented in these notes is the most used for teaching algorithms and follows Knuth's ideas [Knu73]. More sophisticated definitions are related to computability theory and logic [HR87, GBJ02] or even philosophy [Den95], but they are stated in rather abstract frameworks. For a quick survey on the definition of algorithm, see the article *Algorithm characterizations* on Wikipedia [Wik09b] and the references therein (also [Wik09a]).

There can never be too much emphasis on proving the correctness of algorithms. An algorithm can be viewed as a mathematical function (among other interpretations) whose behavior must be guaranteed through a mathematical proof. By exhibiting a proof of correctness both the behavior and result of the algorithm can be relied upon. The habit of proving algorithms should be instilled into students so that it becomes a reflex for them. A good intuitive idea may be a first promising step to algorithmically solve a problem; actually proving the algorithm must be the following, urgent, compulsory step.

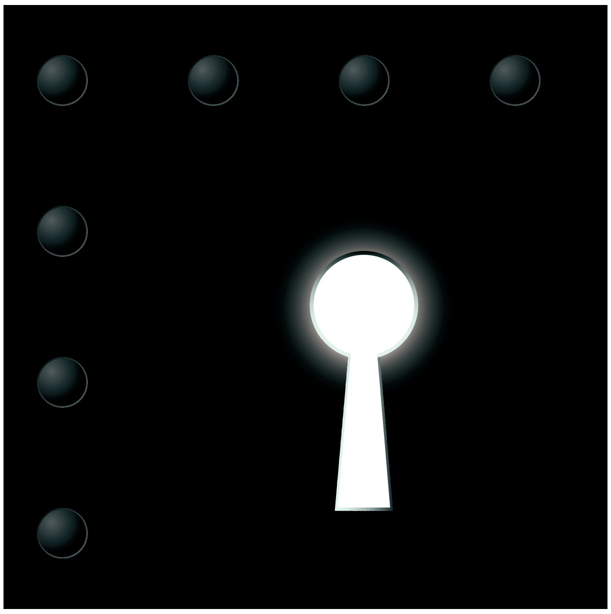
Note that in order to prove an algorithm we have to "describe" it in terms of mathematical relations and properties. That includes a correct specification of input and output (sometimes called precondition and postcondition of algorithm) as well as the invariant. Very frequently, the proof of correctness is made by induction on the invariant of the algorithm. See [CLRS01, Kin97] for more information on mathematical techniques for proving algorithms.

Some students often confuse design of algorithms with its actual implementation in a programming language. How to translate an algorithm to a given programming language is a separate issue. Designing algorithms is an abstract, problem-related task as opposed to programming, which is concrete and language-oriented.

The analysis of algorithms in terms of complexity allows us to classify them. Algorithms whose complexity is $O(n)$ are called **linear algorithms**. When the complexity is $O(n^2)$, they are called **quadratic algorithms**; in general, algorithms of $O(n^k)$ are called **polynomial algorithms**. Algorithms with time complexity of $O(a^n)$, with $a > 1$, are known as **exponential algorithms**.

Chapter 2

Introduction to Exact String Pattern Recognition



2.1 Introduction

The original problem in string matching pattern recognition is so easy to state that even a child could understand it. Two strings of characters are given; one is called the **pattern**, and the other the **text**. The problem is then to find all the occurrences of the pattern in the text. Usually, the text is longer than the pattern. We will call m the length of the pattern and n that of the text. From the beginning of the era of digital computing this problem was identified as a fundamental one in computer science. However, the real applications of string matching were scant, basically text-finding in word processors. For some time the true interest of this problem lied in solving it in linear time, that is, in $O(n + m)$ time. Morris and Pratt [MP70] were the first to succeed in such an endeavor.

As computational methods have been pervading almost every field of knowledge, applications of string matching techniques became ubiquitous and gained in importance. In the late 80's the need of intensive computing to perform analysis of biological data brought forth the hot field of Bioinformatics. In this field a number of string-matching techniques are used for sequence analysis, gene finding, evolutionary biology studies, analysis of protein expression, just to name a few; see [BO05], [CH06] and [Pev00] for an overview. Other fields such as Music Technology, Computational Linguistics, Artificial Intelligence, Artificial Vision, among others, have been using string matching algorithm as part of their arsenal of theoretical and practical tools. New problems in string matching appeared as a result of such continuous, exhaustive use, which in turn were promptly solved by the computer scientists. This situation has culminated in productive, ceaseless feedback between string matching theorists and its practitioners.

2.2 History

The obvious way to solve the string matching problem is to slide the pattern along the text and compare it to the corresponding portion of the text. This algorithm is called the **brute-force algorithm**, which takes $O(nm)$ time, where m is the length of the pattern and n is that of the text. Faster algorithms are obtained by speeding up one of the following steps of the brute-force algorithm:

1. The comparison step, such as in the Karp-Rabin algorithm.
2. The sliding step by pre-processing the pattern, such as in the Knuth-Morris-Pratt algorithm and its variants or in the linear-time versions of the Boyer-Moore algorithm.
3. The sliding step by pre-processing the text, such as Ukkonen's algorithm to construct suffix trees or in Weiner's algorithm.

As early as 1956, just a few years after the invention of ENIAC, Kleene[Kle56] proved the equivalence between finite automaton and regular expressions, which led to solving the string matching problem. For a long time, many string pattern recognition problems were formulated in terms of finite automata. This approach reduced the string matching problem to a language recognition problem. Solutions provided by applying these techniques were often conceptually complicated and had high running times in practice. Even Ukkonen's 1995-algorithm for constructing suffix trees was described in terms of deterministic finite automata. However, as Gusfield [Gus97] showed, many of those algorithms may be reformulated without finite automata terminology. This approach will be followed in these notes.

In 1970 Morris and Pratt [MP70] came up with the first linear-time algorithm to solve the string matching problem. This algorithm is able to skip comparisons by studying the internal structure of the pattern. Seven years later, in 1977, Knut, Morris and Pratt [KMP77] enhanced that algorithm. Although they achieved the same time complexity, their algorithm works much better in practice. A few years later, in 1991, Colussi [Col91] put forward a refinement of the Knuth-Morris-Pratt algorithm that performs fewer comparisons in the worst case. Also, in 1977 Boyer and Moore [BM77] proposed a quadratic algorithm, which, in spite of that bound, is very much used in practice because of its good performance. Several variants/enhancements followed the Boyer-Moore algorithm: the Galil algorithm [Zvi79], an enhancement for periodic patterns that runs in linear time; the Apostolico-Giancarlo algorithm [AG86] (published in 1986), which achieves the same bound but it allows a clean worst-case analysis of its running time; for a thorough review of algorithms see [CL04] and the references therein.

In 1987 Karp and Rabin published an algorithm that ameliorates the comparison step by computing fingerprints of the pattern and the text (see next chapter). Its worst-case running time is quadratic, but, when set up properly, its average case is linear.

In 1993 Simon [Sim93] presented an economical implementation of finite automata running in linear time; Simon observed that only a few edges in the automaton are relevant. A year later Crochemore and Rytter [CR94] came up with a linear-time algorithm with a better performance than that of the Simon algorithm.

As for the algorithms processing the text instead of the pattern, the first algorithm was designed by Weiner [Wei73] in 1973. Three years later McCreight [McC76] gave a different algorithm, which substantially reduced the amount of space used. Much later, in 1995, Ukkonen [Ukk95] presented a neat algorithm to build suffix trees. Suffix trees are very much used in Bioinformatics, where huge biological databases are pre-processed to allow fast queries of many distinct patterns.

Gusfield [Gus97] put forward a linear-time algorithm, the Z algorithm, that was conceptually concise and elegant and made no use of finite automata or similar complicated techniques. To date, it is the easiest algorithm we know of. Furthermore, other more complicated algorithms can be readily re-interpreted in terms of the Z algorithm.

Figure 2.1 shows a timeline with some of the main algorithms to solve the string matching problem.

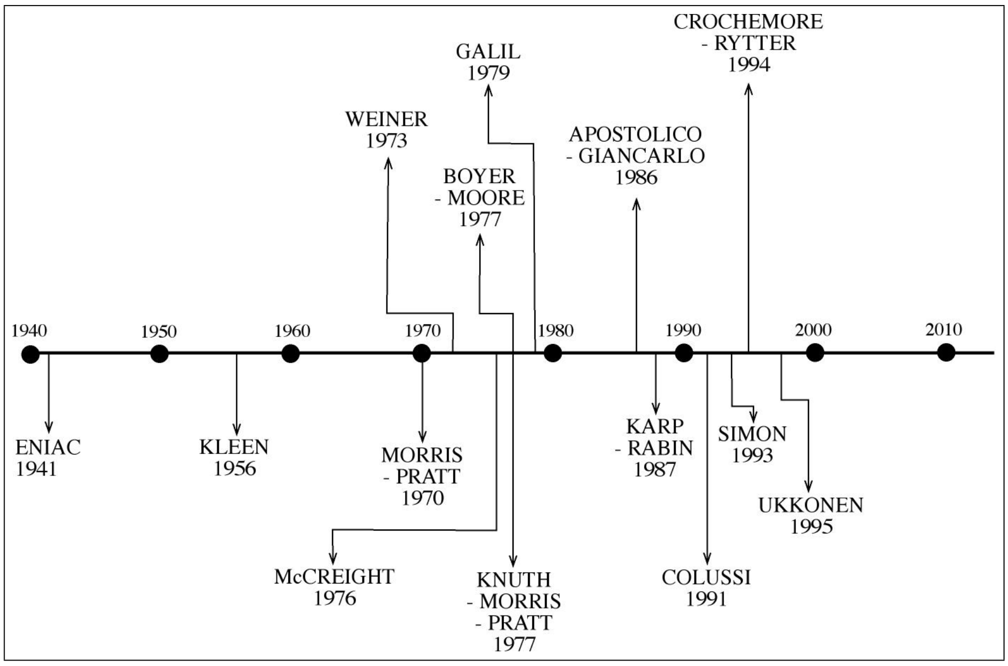
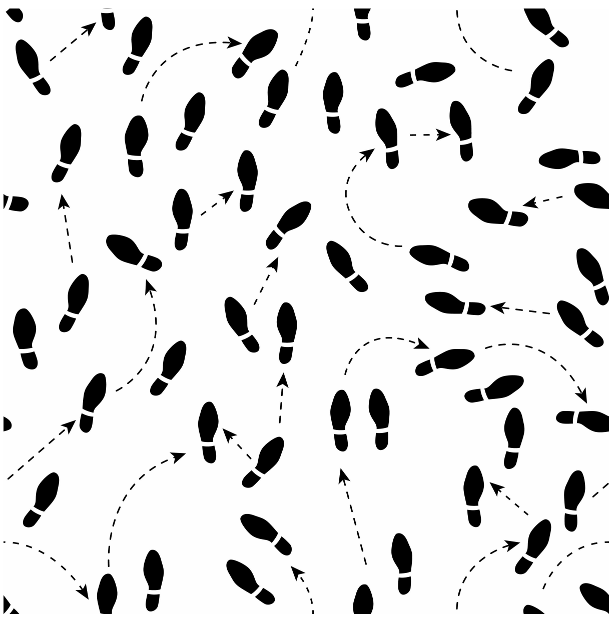


Figure 2.1: Timeline of the history of some string matching algorithms.

Chapter 3

String Matching: First Algorithms



3.1 String-Matching Problems

There are many fundamental problems in String Pattern Recognition, which have been arisen either out of theoretical considerations or practical problems, the latter chiefly originated in Bioinformatics. We start by introducing some definitions prior to presenting those problems.

Typically, a string pattern recognition problem is defined by specifying an object, the **pattern**, to be searched in a larger object, the **text**, under certain conditions. In principle, both the pattern and the text will be arrays of symbols. We shall denote the pattern by $P = P[1..m]$ and the text by $T = T[1..n]$, where m and n are their lengths, respectively. Traditionally, in pattern recognition symbols are just called characters. Characters in P and T are drawn from a common finite alphabet Σ. Typical alphabets are $\{0, 1\}$, $\{a, b, \ldots, z\}$, the ASCII code (in computer science applications), $\{C, G, T, A\}$ (in Bioinformatics applications), sets of integers or real numbers (in musical applications and others) and, in general, finite sets of symbols. The number of characters in Σ will be denoted by d.

The first fundamental problem in string pattern recognition, **the string-matching existence problem (SME problem)**, can be stated as follows.

> **The string-matching existence problem**: Given a pattern P and a text T, determine whether there is an occurrence of P in T.

The immediate generalization of the existence problem is **the string-matching computation problem (SMC)**, namely, reporting how many times a pattern occur in a text.

> **The string-matching computation problem**: Given a pattern P and a text T, determine all the occurrences of P in T.

Unless stated otherwise, by the string matching problem will mean the computation problem.

Example. Assume we are in the context of Bioinformatics. We want to know whether pattern $P = \{CGGTAGC\}$ occurs in text T given below. Here the alphabet is composed of the four main bases found in DNA and RNA, $\{C, G, T, A\}$.

$$\begin{aligned} T \;=\; & \{AGTCCTGAA\underline{G}GTTAA\mathbf{CGGT\underline{A}GC}AGTCCTG\underline{A}AGGTT \\ & AA\mathbf{CG\underline{G}TAGC}AAATT\underline{T}GGGCCCCGT\underline{A}\} \end{aligned}$$

Underlined characters in T are marked at every 10 positions for ease of reading. Without the help of boldface and just by eye-inspection, it would be tedious to check whether P is in T or not. Since pattern P occurs at position 16, the answer to the string-matching existence problem (SME problem from now on) is yes. If we carry out an exhaustive verification, then we will conclude that pattern P also appears at position 38 and it does not occur at any other position. Therefore, the answer to the string-matching computation problem (SMC problem from now on) is positions 16 and 38.

Example. Realistic numbers for the size of P and T in some Bioinformatics applications may be as large as $m = 1000$ and $n = 10,000,000$. The *Diccionario de la RAE*, the official Spanish dictionary, has $87,000$ entries, which, in turn, are based on a database of 270 millions entries. Search for the word "Obama" in Google produces $213,000,000$ pages in 0.24 seconds approximately. Those are just a few examples of situations where string pattern recognition, together with other algorithmic techniques, is used in real practice.

Example. Let us now consider a more complicated example taken from music technology. A fugue is a contrapuntal composition where voices imitate each other. A fugue opens with one main theme, the subject, which then sounds successively in each voice in imitation. Figure 3.1 below shows a few measures from Bach's Fugue number 2 in C minor from The Well-Tempered Clavier. A typical string pattern recognition task would be to recognize entries and imitations of the main theme. The main theme is essentially composed by a descending minor second, followed by an ascending minor second, followed by a descending interval greater than a fourth. Only these four notes are enough for the listener to recognize the main theme. In Figure 3.1 the leftmost square (voice 2) is the main theme (C-B natural-C-G); subsequent occurrences of the theme and its imitations are shown within the squares. Given a music file, for example in MIDI format, an automatic search of those sequences of notes constitutes a string pattern recognition problem.

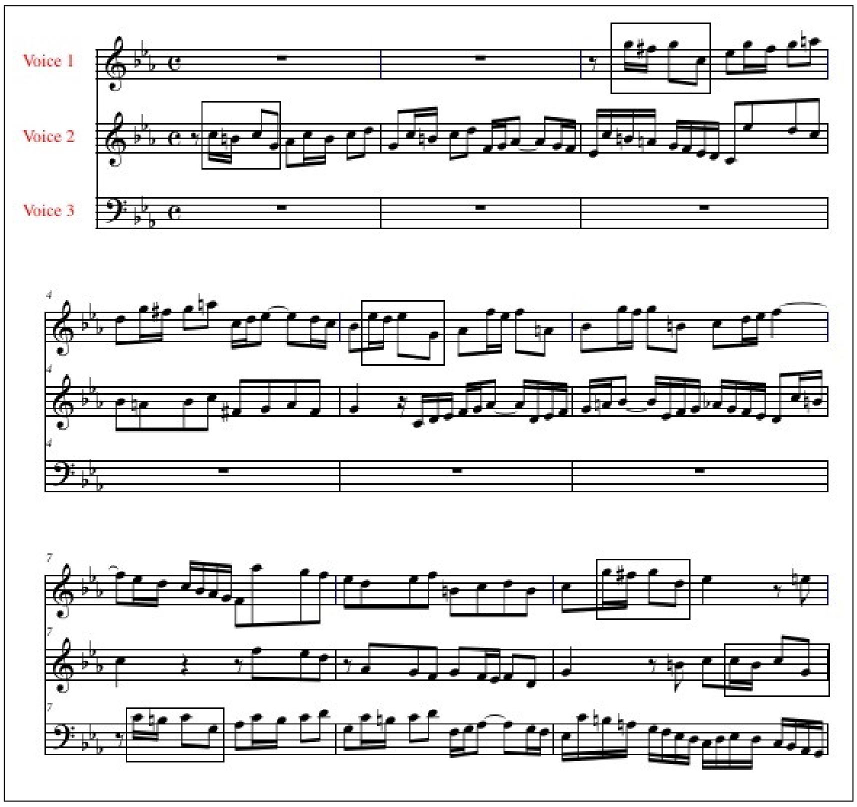


Figure 3.1: String-matching in music.

3.2 Definitions

Arrays of characters are commonly thought of as **strings of characters**. A string of characters is an ordered list of characters. We will use both terms interchangeably. A **shift** s, where $0 \leq s \leq n-m$, consists of verifying whether pattern $P[1..m]$ occurs in $T[s+1..s+m]$.

Here the shift refers to when the pattern is shifted over the text rather than the position the match occurs at (see Figure 3.2).

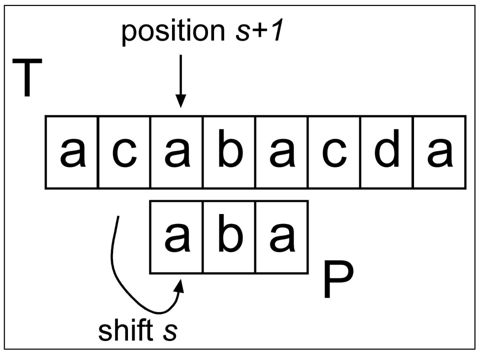


Figure 3.2: Shifts of strings.

Valid shifts are those shifts s such that $P[1..m] = T[s+1..s+m]$; otherwise, they are called **invalid shifts**. Finding a valid shift s of P in T is equivalent to stating that P occurs beginning at position $s+1$ in text T.

Example. Given text $T = \{\texttt{abccabdadcca}\}$ and pattern $P = \{\texttt{cca}\}$, $s = 2$ and $s = 9$ are valid shifts for P. The actual matching positions are 3 and 10.

String matching problems can be restated in terms of shifts. For instance, the SMC problem can formulated as that of finding all the valid shifts of P in T. The SME problem can reworded as that of finding a valid shift of P in T.

Let us fix an alphabet Σ. We denote by Σ^* the set of all finite-length strings formed by drawing characters from alphabet Σ. The empty string is denoted by ε. Given a string $x \in \Sigma^*$ the **length** of x is its number of characters; it will be denoted by $|x|$. The **concatenation** of two strings x and y is written as xy. This operation just consists of aligning the characters of y after those of x in its given order. The length of xy is $|x| + |y|$. Given a string w, the n-times concatenation of w will be written as $w^n = w \overset{n}{\ldots} w$.

Example. Let $S_1 = \{\texttt{ababab}\}$ and $S_2 = \{\texttt{cdcdc}\}$ be two strings. Then, $|S_1| = 6$ and $|S_2| = 5$. Concatenation of S_1, S_2 is $S_1S_2 = \{\texttt{abababcdcdc}\}$. Its length is $|S_1S_2| = 11$.

3.3 Prefixes and Suffixes

We say that a string y is a **prefix** of x, denoted by $y \sqsubset x$, if $x = yw$, for some string $w \in \Sigma^*$. Then, $|y| \leq |x|$. Similarly, we say that a string y is a **suffix** of a string x, to be written as $y \sqsupset x$, if $x = wy$, for some $w \in \Sigma^*$. Also, $|y| \leq |x|$. By convention the empty string is a suffix and a prefix of any string.

Example. Consider strings $S_1 = \{\texttt{abca}\}, S_2 = \{\texttt{cbac}\}$ and $S_3 = \{\texttt{abcaabccbacbac}\}$. String S_1 is a prefix of S_3 since there exists a string $w_1 = \{\texttt{abccbacbac}\}$ such that $S_3 = S_1 w_1$. Analogously, S_2 is a suffix of S_3 because $S_3 = w_2 S_2$, where $w_2 = \{\texttt{abcaabccba}\}$.

Theorem 3.3.1 *$\sqsubset$ and $\sqsupset$ are order relations, that is, they are reflexive, antisymmetry and transitivity relations.*

Proof: The proof is left for the exercises. ∎

The following Lemma will be useful later.

Lemma 3.3.2 (The overlapping-suffix lemma.) *Suppose that x, y, z are strings such that $x \sqsupset z$ and $y \sqsupset z$. Then:*

- *If $|x| \leq |y|$, then $x \sqsupset y$.*
- *If $|y| \leq |x|$, then $y \sqsupset x$.*
- *If $|y| = |x|$, then $y = x$.*

Proof: It is enough to make the proof for the first case; the rest is quite similar. Assume that $x \sqsupset z$. Then, there exists a string w_1 such that $z = w_1 x$. Analogously, since $y \sqsupset z$, we can find another string w_2 such that $z = w_2 y$. Therefore, $z = w_1 x = w_2 y$. Given that $|x| \leq |y|$, w_2 must be a substring of w_1. Consider the substring w_3 formed by the characters in w_1 that are not in w_2 listed in its own order. It follows that $y = w_3 x$ and, hence, $x \sqsupset y$. ∎

We shall denote the k-character prefix $P[1..k]$ of the pattern $P = P[1..m]$ by P_k. By using this notation, we can state the string-matching computation problem as that of finding all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$.

Example. Given text $T = \{\texttt{abccabdadcca}\}$ and pattern $P = \{\texttt{cca}\}$ as above, we find that P is a suffix of $T_{2+3} = T_5 = \{\texttt{abcca}\}$ and $T_{9+3} = T_{12} = \{\texttt{abccabdadcca}\}$.

Exercises

$\boxed{1}$ Prove Theorem 3.3.1.

$\boxed{2}$ Can two or more occurrences of a pattern P in a text T overlap? If so, put examples of it.

$\boxed{3}$ For given pattern P and text T below, find all the valid shifts. Describe the valid shifts in terms of suffixes.

- $P = \{\texttt{CGTAT}\}$ and $T = \{\texttt{CGCGTACCGTAGCGTAACGTATGCTATCGCG}\}$.
- $P' = \{\texttt{TTAGC}\}$ and $T' = \{\texttt{AGTGTACTGTAGGCCGTATACTATTAGTGCG}\}$.

3.4 The Naive String-Matching Algorithm

The naive or brute-force algorithm finds occurrences of P in T by checking whether P occurs all of the $n-m+1$ consecutive substrings of length m of T. The following algorithm is a straightforward translation of that idea for the computation problem.

```
   NAIVE-STRING-MATCHER(P, T)
1  $n \leftarrow length[T]$
2  $m \leftarrow length[P]$
3  for $s \leftarrow 0$ to $n-m$
4    $i \leftarrow 1$
5    while ($P[i] = T[i+s]$ and $i \leq m$) # Check whether $T[s+1..s+m] = P[1..m]$.
6      do $i \leftarrow i+1$
7    if $i = m+1$ then print “$P$ occurs at position $s$”
```

Table 3.1: The *NAIVE-STRING-MATCHER(P, T)* algorithm for the computation problem.

Note that we have explicitly expanded the comparison $T[s+1..s+m] = P[1..m]$ as a while loop in lines 5–6. We did so this time to ease the computation of time complexity. In the future, we will treat the comparison $T[s+1..s+m] = P[1..m]$ as a primitive. The complexity of performing the comparison $T[s+1..s+m] = P[1..m]$ is $\Theta(r+1)$, where r is the length of the longest string z such that z is a common prefix of $T[s+1..s+m]$ and $P[1..m]$. Obviously, $r = O(m)$.

For the computation problem the time complexity is $O((n-m+1)m)$, which results from reckoning $n-m+1$ comparisons $T[s+1..s+m] = P[1..m]$, for $s = 0, \dots, n-m$, at a cost of $O(m)$ each. The worst case takes places when there are many valid shifts. For example, consider the text string $T = \mathtt{a}^n = \{\mathtt{a} \overset{n}{\dots} \mathtt{a}\}$ and the pattern $P = \mathtt{a}^m = \{\mathtt{a} \overset{m}{\dots} \mathtt{a}\}$. In this case, for each of the $n-m+1$ comparisons the while loop must be executed m times to validate the shift. Thus, the complexity reaches $O((n-m+1)m)$. In fact, that complexity may go up to $O(n^2)$. If $m = \lfloor \frac{n}{2} \rfloor$, then the time complexity will grow up to $O(n^2)$. Note that if $m = O(1)$, then the naive algorithm runs in linear time.

Solving the existence problem by using a naive algorithm is left for the exercises as well as its complexity analysis.

The naive string-matching algorithm is inefficient because it systematically ignores all the information gained as the pattern is slid over the text. In fact, many efficient string-matching algorithms take advantage of that information in a number of ways to lower the complexity, as will be seen in the next few chapters. However, there is just one situation where the naive string-matching algorithm is efficient. Such situation arises when the pattern and the text are randomly chosen. The following theorem proves this point.

Theorem 3.4.1 *Let Σ be a d-symbol alphabet, where $2 \leq d < \infty$. Assume that pattern P and text T are randomly chosen from Σ. Then, the expected number of character-to-character comparisons performed by the naive string matcher is*

$$(n-m+1)\frac{1-\left(\frac{1}{d}\right)^m}{1-\frac{1}{d}} \leq 2(n-m+1).$$

Proof: Let X be the number of character-to-character comparisons performed during one execution of the while loop of *NAIVE-STRING-MATCHER* (lines 5–6). X is a random variable taking values from $\{1, 2, \ldots, m\}$. There are several cases for the probabilities associated with X.

- Variable $X = 1$ when just one comparison is made, which only happens if the first comparison is a mismatch. That happens with probability $1 - \frac{1}{d}$.
- If $2 < i < m$, $X = i$ when the first $i-1$ comparisons are matches and the last one is a mismatch. That happens with probability $(\frac{1}{d})^{i-1} \cdot (1-\frac{1}{d})$. Bear on mind that results of consecutive comparisons are independent.
- Finally, m comparisons are made when either $m-1$ matches are found and the last comparison is a mismatch or all the comparisons are matches (pattern P has been found).

$$P(X=m) = \left(\frac{1}{d}\right)^{m-1} \cdot \left(1-\frac{1}{d}\right) + \left(\frac{1}{d}\right)^{m} = \left(\frac{1}{d}\right)^{m-1} - \left(\frac{1}{d}\right)^{m} + \left(\frac{1}{d}\right)^{m} = \left(\frac{1}{d}\right)^{m-1}.$$

Let us check that the probabilities obtained actually are a mass function (that is, their sum equals 1).

$$\begin{aligned}\sum_{i=1}^{m-1}\left[\left(\frac{1}{d}\right)^{i-1} \cdot \left(1-\frac{1}{d}\right)\right] + \left(\frac{1}{d}\right)^{m-1} &= \frac{1-(\frac{1}{d})^{m-1}}{1-\frac{1}{d}}\left(1-\frac{1}{d}\right) + \left(\frac{1}{d}\right)^{m-1} \\ &= 1 - \left(\frac{1}{d}\right)^{m-1} + \left(\frac{1}{d}\right)^{m-1} = 1.\end{aligned}$$

Here we have made use of the formula to sum k consecutive terms of a geometric progression.

We now can safely compute the expected value of random variable X.

$$\begin{aligned}E(X) &= \sum_{i=1}^{m} iP(X=i) = 1 \cdot \left(1-\frac{1}{d}\right) + 2 \cdot \left(1-\frac{1}{d}\right) \cdot \frac{1}{d} \\ &\quad +3 \cdot \left(1-\frac{1}{d}\right)\left(\frac{1}{d}\right)^2 + \ldots + (m-1) \cdot \left(1-\frac{1}{d}\right)\left(\frac{1}{d}\right)^{m-1} + m\left(\frac{1}{d}\right)^{m-1} \\ &= 1 - \frac{1}{d} + \frac{2}{d} - \frac{2}{d^2} + \frac{3}{d^2} - \frac{3}{d^3} + \ldots + \left(\frac{1}{d}\right)^{m-1} \\ &= 1 + \frac{1}{d} + \frac{1}{d^2} + \ldots + \left(\frac{1}{d}\right)^{m-1} = \sum_{i=0}^{m-1}\left(\frac{1}{d}\right)^{i} = \frac{1-(\frac{1}{d})^m}{1-\frac{1}{d}}.\end{aligned}$$

Since each execution of the for loop is also independent of the others, the overall expected value is:

$$(n-m+1)\,\frac{1-(\frac{1}{d})^m}{1-\frac{1}{d}}.$$

The term $\frac{1-(\frac{1}{d})^m}{1-\frac{1}{d}}$ depends on the size d of the alphabet Σ. Its numerator is always less than 1, for any $d > 2$; moreover, its denominator attains its minimum value when $d = 2$. Therefore, the previous expression can be bounded as follows:

$$(n-m+1)\,\frac{1-(\frac{1}{d})^m}{1-\frac{1}{d}} < (n-m+1)\,\frac{1}{1-\frac{1}{2}} = 2(n-m+1).$$

▮

Exercises

$\boxed{1}$ Show the trace of the naive string matcher for the pattern $P = \{\texttt{aaab}\}$ and text $T = \{\texttt{aaaab}\}$ (computation problem).

$\boxed{2}$ Consider the existence string-matching problem. Design an algorithm to solve that problem. Characterize the best- and worst-case and give bounds for its time complexity.

$\boxed{3}$ Plot the function

$$f(d,m) = \frac{1-(\frac{1}{d})^m}{1-\frac{1}{d}}$$

for several values of d and m. Confirm experimentally that $f(d,m)$ has an upper bound of 2.

$\boxed{4}$ Compute the probability the for loop in *NAIVE-EXISTENCE-CHECKER* is executed k times.

$\boxed{5}$ Suppose that all characters in pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in linear time on an n-character text T.

3.5 The Rabin-Karp Algorithm

In spite of the fact that other algorithms have a better worst-case complexity, the Karp-Rabin algorithm is still interesting in its own right because it performs well in practice in some cases, up to certain extent, and also because it easily generalizes to other problems, such as the two-dimensional pattern matching and the many-patterns problem.

The naive algorithm is composed of two steps, the shifting step, performed in the for loop, and the comparison $T[s+1..s+m] = P[1..m]$, performed in the while loop. Whereas most algorithms rely on improving the number of shifts by examining the structure of P or T, the Karp-Rabin algorithm improves the character-to-character comparison step. The worst-case complexity, as we shall see below, is still $O((n-m+1)m)$, and even $O(n^2)$ for certain choices of P and T. However, the algorithm is used in practice because of its good average-case behaviour, which lowers time complexity to $O(n+m)$. This algorithm may be considered as a numerical algorithm rather than a comparison-based algorithm.

3.5.1 Strings as Numbers

Let us assume, only for expository purposes, that the alphabet Σ is $\{0, 1, \ldots, 9\}$. Generalization to other alphabets is straightforward (see the exercises). One of the key ideas of the algorithm is to treat strings as decimal numbers, in general, as numbers represented in radix d. For example, string $S = \{\mathtt{147}\}$ will be interpreted as the number $147 = 1\cdot 10^2 + 4\cdot 10^1 + 7\cdot 10^0$. Let us denote by p the numerical value of P and by t_s that of $T[s+1..s+m]$. A direct computation of p involves $\Theta(m^2)$ arithmetic operations. Indeed, by examining the expansion of p in powers of 10,

$$p = \sum_{i=1}^{m} P[i]\cdot 10^{m-i} = P[m] + P[m-1]\cdot 10^1 + \ldots + P[1]\cdot 10^{m-1},$$

we note that the i-th term $P[i]\cdot 10^{m-i}$ takes $\Theta(m-i)$ time, which gives an overall time of $\sum_{i=1}^{m}\Theta(i) = \Theta(m^2)$. A faster way to obtain p is to apply Horner's rule, a recursive rule to evaluate polynomials. Thus, p can be written as

$$\begin{aligned}
p &= P[m] + P[m-1]\cdot 10^1 + P[m-2]\cdot 10^2 + \ldots + P[1]\cdot 10^{m-1}\\
&= P[m] + 10\cdot(P[m-1] + P[m-2]\cdot 10^1 + \ldots + P[1]\cdot 10^{m-2})\\
&= P[m] + 10\cdot(P[m-1] + 10(P[m-2] + \ldots + P[1]\cdot 10^{m-3}))\\
&= \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&= P[m] + 10\cdot(P[m-1] + 10\cdot(P[m-2] + \ldots + 10P[2] + P[1]))\ldots).
\end{aligned}$$

Analogously, the numerical value

$$t_s = \sum_{i=1}^{m} T[s+i]\cdot 10^{m-i} = T[s+m] + T[s+m-1]\cdot 10^1 + \ldots + T[s+1]\cdot 10^{m-1}$$

is recursively computed with Horner's rule.

The following theorem guarantees that numerical equality between t_s and p is equivalent to finding an occurrence of P in T.

Theorem 3.5.1 *If $t_s = p$, then pattern P occurs in text T at position s.*

Proof: The result follows from the well-known fact that representation of integers as sum of positive powers is unique. Therefore, if $t_s = p$, then $P[i] = T[s+i]$, for $i = 1, \dots, m$. ▮

Next task is to show a recursive relation to efficiently compute the numerical value t_s from t_{s-1}. By comparing expansions of t_s and t_{s-1}, we will find a recursive relation between them. Indeed, there are certain terms common to both expansions:

$$\begin{array}{rcllll} t_{s-1} & = & & T[s-1+m] + T[s-1+m-1]\cdot 10^1 + \dots + & T[s+1]\cdot 10^{m-2} + \\ & & T[s]\cdot 10^{m-1} & & & \\ t_s & = & T[s+m] + & T[s+m-1]\cdot 10^1 + \dots + T[s+2]\cdot 10^{m-2} + & T[s+1]\cdot 10^{m-1}. \end{array}$$

If we multiply t_{s-1} by 10 and subtract both expressions, we will achieve the desired relation.

$$\begin{array}{rcl} t_s - 10t_{s-1} & = & T[s+m] - T[s]\cdot 10^m \Longrightarrow \\ t_s & = & 10(t_{s-1} - 10^{m-1}T[s]) + T[s+m]. \end{array}$$

The last equality provides a fast way to compute t_s from t_{s-1}. Computing t_0 is $\Theta(m)$ and thanks to the equation above, the remaining $m-1$ values are obtained at constant cost each. In total, computing $\{t_0, t_1, \dots, t_{n-m}\}$ takes $\Theta(m) + \Theta(n-m) = \Theta(n)$ time.

Example. Take string $T = \{\texttt{123456}\}$ and assume that $m = 3$. Numerical value t_0 is 123. We compute the remaining values t_s, $s = 1, 2, 3$, with the recursive formula.

$$\begin{array}{rcl} t_1 & = & 10(t_0 - 10^{3-1}T[1]) + T[1+3] = 10(123 - 10^2\cdot 1) + 4 = 234, \\ t_2 & = & 10(t_1 - 10^{3-1}T[2]) + T[2+3] = 10(234 - 10^2\cdot 2) + 5 = 345, \\ t_3 & = & 10(t_2 - 10^{3-1}T[3]) + T[3+3] = 10(345 - 10^2\cdot 3) + 5 = 456. \end{array}$$

Unfortunately, there is a fly in the ointment. For typical values of m, numbers t_s and p may be very large. In fact, they may be so large as to no longer assume basic operations can be performed in constant time or their storage takes constant space. Let us make a little digression on models of computation here. When introduced the real RAM in Chapter 1, we did not address the issue of number representation, which, however, is necessary to deal with now. As a matter of fact, infinite precision cannot be assumed and real RAM sets a maximum number w of bits to be used to store integers. This value w is, of course, independent of n and m. The maximum representable integer is $2^w - 1 = \sum_{i=0}^{w-1} 2^i$. This situation seems a tough nut to crack, but there is magic bullet to help us out: modular arithmetic. We will perform all the computations modulo some suitable integer $q < 2^w$. This way we keep our calculations within bounds.

3.5.2 Fingerprints

Let us fix an integer $q > 0$. We call the **fingerprint** of an integer x, to be denoted by $F(x)$, x reduced modulo q. For instance, if $q = 256$, the fingerprint of $x = 806$ is 38 because $806 = 3 \cdot 256 + 38$. Fingerprints are also called **signatures**. Although taking moduli guarantees the final result is within precision, overflow may happen during the computation of intermediate steps. The next example illustrates this situation.

Example. Assume that $w = 16$ and $q = 32771$. The largest representable integer is $2^{16} - 1 = 65535$. Take $P = \{\texttt{789986}\}$ as the pattern. Let us compute p, its numerical value, which is clearly greater than 65535. Below is the trace of Horner's rule applied to pattern P.

$$\begin{array}{ll} \text{Horner's rule:} & 6 + 10 \cdot (8 + 10 \cdot (9 + 10 \cdot (9 + 10 \cdot (8 + 7 \cdot 10)))) \\ & 7 \cdot 10 + 8 = 78 < 65535 \\ & 78 \cdot 10 + 9 = 789 < 65535 \\ & 789 \cdot 10 + 9 = 7899 < 65535 \\ & 7899 \cdot 10 + 8 = 78998 > 65535 \end{array}$$

At this point overflow happens. Because of it, the number 78998, whose binary representation is 10011010010010110, is truncated to 13462, 0011010010010110 in binary (just keep the first 16 bits). Finally, we perform the last step:

$$13462 \cdot 10 + 6 = 134626 > 65535.$$

Again, because of overflow, 134626 is converted into 3554. When computing the fingerprint of P, we obtain $F(789986) = 3554 \bmod 32771 = 3554$. However, the true value of $F(789986)$ is $3482 = 789986 \bmod 32771$ (because $789986 = 32771 \cdot 24 + 3482$).

The following theorem ensures that all the results obtained when computing the numerical values of P and T through Horner's rule are kept within precision.

Theorem 3.5.2 *Let x a positive integer whose decimal representation is $\sum_{i=0}^{k} a_i 10^i$. If fingerprint $F(x)$ is computed with the following rule:*

$$\begin{cases} x_0 & = & a_0 \\ x_i & = & (10x_{i-1} \bmod q + a_i) \bmod q, \;\; i = 1, \ldots, k, \end{cases}$$

then $x_k = F(x)$ and no number ever exceeds $10q$ during the computation.

Proof: That $x_k = F(x)$ is immediate from the correctness of Horner's rule. The largest number it may appear during the computation of $F(x)$ corresponds to $x_{i-1} = q - 1$ and $a_i = 9$. In such case the intermediate result will be $10(q-1) + 9 = 10q - 1 < 10q$, as desired. ∎

Again, unfortunately, there is another fly in the ointment. When taking moduli the uniqueness guaranteed by Theorem 3.5.1 no longer holds. The fact that two numbers are equal modulo q does not imply they are actually equal. Take, for instance, x and $x+q$. When $t_s = p \bmod q$ but $t_s \neq p$, we say there is a **false match** between P and T with shift s.

False positives forces the Karp-Rabin algorithm to explicitly perform the comparison $T[s+1..n] = P[1..m]$ when $t_s = p \bmod q$. Now we have gathered all the ingredients to put forward the pseudocode for the algorithm.

```
     RABIN-KARP-MATCHER(P, T, d, q)
1    $n \leftarrow length[T]$
2    $m \leftarrow length[P]$
3    $a \leftarrow d^{m-1} \bmod q$
4    $p \leftarrow 0$
5    $t \leftarrow 0$
6    for $i \leftarrow 1$ to $m$
7      $p \leftarrow (dp + P[i]) \bmod q$
8      $t \leftarrow (dt_0 + T[i]) \bmod q$
9    for $s \leftarrow 0$ to $n - m$
10     if $t = p \bmod q$ then
11       if $T[s+1..s+m] = P[1..m]$ then print "$P$ occurs at position $s+1$"
12     if $s < n - m$ then $t \leftarrow (d \cdot (t - T[s+1] \cdot a \bmod q) + T[s+m+1]) \bmod q$
```

Table 3.2: The *RABIN-KARP-MATCHER(P, T)* algorithm for the computation problem.

The complexity of the algorithm depends on the number of false positives. Let us call that number v. Lines 1–8 of the algorithm take $\Theta(m)$ time. The for loop is executed $n-m+1$ times, in v of which the explicit comparison $T[s+1..n] = P[1..m]$ is made to resolve the numerical equality $t_s = p$. Therefore, the overall complexity is

$$\Theta(m) + \Theta(n - m + 1 - v) + \Theta(vm) = \Theta(n - v + vm) = \Theta(n + vm).$$

The number of false positives v may be as high as $\Theta(n-m)$. Consider again the text string $T = \mathtt{a}^n$ and the pattern $P = \mathtt{a}^m$. In this case, $v = n - m + 1$ and the Karp-Rabin algorithm becomes a $\Theta(nm)$-time algorithm. However, the reason the Karp-Rabin algorithm is used in practice is due to its good average-case performance. In the next section we will analyse its average-case complexity.

3.5.3 Average-Case Analysis of the Karp-Rabin Algorithm

We will exploit several properties of modular arithmetic to reduce the size of v to even a constant. The key idea arises from selecting q as a prime within a certain range. We will state (without proof) some results about prime numbers needed for our analysis.

Let $\pi(x)$ be the numbers of primes that are less or equal than x, where x is a positive integer. By convention number 1 will not be considered as a prime number. Here there are a few values of $\pi(x)$: $\pi(10) = 4, \pi(100) = 25, \pi(10^6) = 78,498, \pi(10^9) = 50,487,534$. Function $\pi(x)$ is called the **prime distribution function**.The next theorem establishes the order of growth of $\pi(x)$.

Theorem 3.5.3 (Hardy and Littlewood [HL16] and Rosser and Schoenfield [RS62]) *Function $\pi(x) \in \Theta\left(\frac{x}{\ln(x)}\right)$; moreover, the following tight inequalities hold:*

$$\frac{x}{\ln(x)} \leq \pi(x) \leq 1.26\frac{x}{\ln(x)}.$$

In Figure 3.3 function $\pi(x)$ is drawn in black; the bounding functions $\frac{x}{\ln(x)}$ and $1.26\frac{x}{\ln(x)}$ are shown in red.

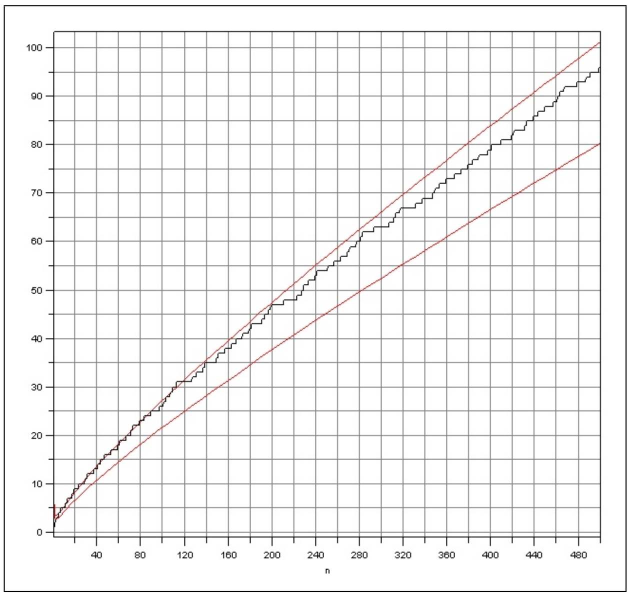


Figure 3.3: Order of growth of function $\pi(x)$.

Theorem 3.5.4 (Rosser and Schoenfield [RS62]) *Let $x \geq 29$ be an integer. Then, the product of all the primes that are less or equal to x is greater than 2^x.*

Example. Let us choose $x = 31$. Primes less or equal than 31 are $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31\}$. The product $2 \cdot 3 \cdot \ldots \cdot 31 = 200,560,490,130$, whereas $2^{31} = 2,147,483,648$. If $x = 17$, we have that the product $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 = 510510$ is less than 2^{31}.

Theorem 3.5.5 *If $x \geq 29$ be an integer. If y is an integer less or equal than 2^x, then the number of prime divisors of y is less than $\pi(x)$.*

Proof: For the sake of a contradiction, assume y has more divisors than $\pi(x)$, say, r. Call those divisors $q_1, q_2, \ldots, q_r$. Since some of those primes may be repeated in the decomposition of y, we can write

$$q_1 \cdot q_2 \cdot \ldots \cdot q_r \leq y < 2^x.$$

However, by assumption y has more divisors than $\pi(x)$, which in turn implies that $q_1 \cdot q_2 \cdot \ldots \cdot q_r$ is greater than the product of the $\pi(x)$ prime divisors of x. By Theorem 3.5.4, $q_1 \cdot q_2 \cdot \ldots \cdot q_r > 2^x$. This causes a contradiction ($2^x < 2^x$). ▮

Example. Again, consider $x = 31$. Let $y = 10^6$, which is less than $2^{31} = 2,147,483,648$. According Theorem 3.5.5, the number of prime divisors of y should be less or equal than $\pi(x)$. Indeed, $\pi(10^6) = 78,499$ and $\pi(2^{31})$ is greater than $6,927,366$ (see Theorem 3.5.6 below).

Theorem 3.5.6 (after Karp and Rabin [KR87]) *Let P and T be strings such that $nm \geq 29$. Let q be a prime randomly chosen from the interval $[1, b]$, where b is a positive integer. Then, the probability of a false match between P and T is not greater than*

$$\frac{\pi(nm \log_2(10))}{\pi(b)}.$$

Proof: Let R the set of positions where numerical values t_s and p are different. In other words, if r is a position in R, then $t_r \neq p$. A prime q will produce a false match if it is a prime factor of some of the numbers $|t_r - p|$, for some $r \in R$. Indeed, when q evenly divides $|t_r - p|$, it follows that $t_r = p \bmod q$.

Now, let us consider the product

$$\prod_{r \in R} |t_r - p|.$$

The set of prime factors of numbers $|t_r - p|, r \in R$, is equal to the set of prime factors of $\prod_{r \in R} |t_r - p|$. Since $|t_r - p| < 10^m$, the whole product is bounded by 10^{nm}. In order to be able to use the previous theorems, we express that bound in base 2, that is, $10^{nm} = 2^{nm \log_2(10)}$. By applying Theorem 3.5.5, we deduce that $\prod_{r \in R} |t_r - p|$ has at most $\pi(nm \log_2(10))$ distinct prime divisors. Prime q was randomly chosen from interval $[1, b]$, which contains $\pi(b)$ total primes. Furthermore, q is a a false match chosen out of at most $\pi(nm \log_2(10))$ primes. Therefore, the probability of a false match between P and T is bounded by

$$\frac{\pi(nm \log_2(10))}{\pi(b)}.$$

▮

Example. Let $T = \{\texttt{23456}\}$ and $P = \{\texttt{17}\}$ be a text and a pattern, respectively. All the positions where numerical values t_r and $p = 17$ differ are $R = \{1, 2, 3, 4\}$. The actual numerical differences are:

$$\begin{cases} r = 1 & : \quad 23 - 17 = 6 \\ r = 2 & : \quad 34 - 17 = 17 \\ r = 3 & : \quad 45 - 17 = 28 \\ r = 4 & : \quad 56 - 17 = 39 \end{cases}$$

The set of prime factors of those numerical differences is $\{2, 3, 7, 13, 37\}$. When one of these primes is chosen as prime q false matches appear in the Karp-Rabin algorithm. Let us consider product $\prod_{r\in R} |t_r - p| = 6 \cdot 17 \cdot 28 \cdot 39 = 111,384$. The set of prime factors of $111,384$ is again $\{2, 3, 7, 13, 37\}$. Since $nm \log_2(10) = 5 \cdot 2 \log_2(10) \approx 33.21928 > 29$, Theorem 3.5.5 ensures that product $\prod_{r\in R} |t_r - p|$ has no more than $\pi(nm \log_2(10)) = 11$ prime divisors. Let us set $b = 10^4$. The number of possible primes q can be chosen from is $\pi(b) = 1,229$. The real probability of a false match is

$$\frac{|\{2, 3, 7, 13, 17\}|}{1,229} = \frac{5}{1,229} \approx 0.004068,$$

whereas the upper bound provided by the Karp-Rabin theorem is

$$\frac{\pi(nm \log_2(10))}{1,229} = \frac{11}{1,229} \approx 0.0089503.$$

Example. Set $n = 1000$, $m = 50$ and $b = 10^6$. In this case the probability of a false match is

$$\frac{\pi(1000 \cdot 50 \log_2(10))}{\pi(10^6)} = \frac{15181}{78499} = 0.19339.$$

Notice that for this particular choice of b the probability of a false match is quite high, around a 20%. In real practice this percentage is unacceptable.

In the last part of this section we will tackle with the problem of choosing the interval $[1, b]$ so that probability $\frac{\pi(nm \log_2(10))}{\pi(b)}$ is made small. A good choice of b is $b = n^2 m \log_2(10)$. The bounding below, carried out with the help of Theorem 3.5.3, justifies that choice.

$$\begin{aligned}
\frac{\pi(nm \log_2(10))}{\pi(n^2 m \log_2(10))} &\leq \frac{1.26 \frac{nm \log_2(10)}{\ln(nm \log_2(10))}}{\frac{n^2 m \log_2(10)}{\ln(n^2 m \log_2(10))}} = 1.26 \cdot \frac{\ln(n^2 m \log_2(10))}{n \ln(nm \log_2(10))} \\
&= \frac{1.26}{n} \cdot \frac{\ln(n) + \ln(nm \log_2(10))}{\ln(nm \log_2(10))} \\
&= \frac{1.26}{n} \cdot \left(1 + \frac{\ln(n)}{\ln(n) + \ln(m) + \ln(\log_2(10))}\right) \\
&\leq \frac{1.26}{n} \cdot 2 = \frac{2.52}{n}.
\end{aligned}$$

Example. Set the maximum number of bits w to 32. Suppose that $n = 5000$ and $m = 50$. We choose b to be $n^2 \cdot m \cdot log_2(10) = 4,152,410,119 < 2^{32} - 1 = 4,294,967,295$. Thus, we are within the bounds of our computer. Applying the previous reasoning, we find that the probability of a false match is *bounded* by $\frac{2.52}{5,000} = 0.000504$.

Appropriately enough, this is good point to consider **randomized algorithms**. Randomized algorithms are algorithms that make use of some degree of randomness. The goal of employing such randomness is to achieve a good performance in the average-case. A good example of how randomization works is given by the quicksort, a very commonly-used sorting algorithm. By making certain decisions at random the quicksort achieves an $O(n \log n)$ average-case bound, whereas the worst-case bound is $O(n^2)$. Randomization minimizes the chances the worst case appears. See the references in the chapter notes.

Turning to the Karp-Rabin algorithm, it suffices to pick prime q at random *each* time the algorithm is run. This way the algorithm is randomized and in the average we guarantee the linear time bound $\Theta(n + mv)$.

Exercises

$\boxed{1}$ **Horner's rule.** Prove formally that Horner's rule correctly evaluates a polynomial $P_k(x) = a_0 + a_1 x + \ldots + a_k x^k$.

$\boxed{2}$ Compute the following fingerprints, where $q = 257$: $F(138), F(560), F(10,349)$, $F(10^6)$ and $F((257)^2 + 1)$.

$\boxed{3}$ Working modulo $q = 11$, count the number of false matches in the Rabin-Karp algorithm for the text $T = 3141537026045859293048 1$ when looking for the pattern $P = 26$?

$\boxed{4}$ Give lower and upper bounds for the number of primes less than $x = 10^6$.

$\boxed{5}$ What would it be the minimum size of text T so that a probability of a false match being less than 10^{-4} can be ensured?

$\boxed{6}$ **Circular patterns - I.** Given two strings A, B of n characters each, design an algorithm that determines whether one is a circular permutation of the other. For example, string `abcdefg` is a circular permutation of `cdefgab`. Prove that your algorithm is correct and analyse its complexity.

$\boxed{7}$ **Circular pattern - II.** Design an algorithm that determines whether string A is a substring of a circular string B. In a circular string the last character is considered to precede the first character. Prove that your algorithm is correct and analyse its complexity.

$\boxed{8}$ How would you extend the Rabin-Karp algorithm to handle the problem of searching a text string for an occurrence of any one of a given set of k patterns?

$\boxed{9}$ Supply all the details to change the description of the Karp-Rabin algorithm when using an arbitrary alphabet. In particular, show how the new alphabet would affect to the probability of a false match and its bound.

$\boxed{10}$ How would the naive string matching algorithm be randomized? Would that randomization improve its time complexity?

3.6 More String-Matching Problems

In this last section we present a few more problems on string matching, all of them being relevant both in theory and practice. We will return to those problems in the next few chapters.

3.6.1 The Two-Dimensional String-Matching Problem

The first problem is the generalization of the computation problem to two dimensions. This problem often arises in image processing and bioinformatics. It is stated as follows.

The 2-dimensions string matching problem (2DSM problem): Given two matrices $T[1..n_1, 1..n_2]$ and $P[1.m_1, 1..m_2]$, find all the occurrences of P in T.

Finding an occurrence of P in T means there exists a shift (s_1, s_2) such that

$$T[s_1 + 1..s_1 + m_1, s_2 + 1..s_2 + m_2] = P[1..m_1, 1..m_2].$$

Figure 3.4 illustrates this problem with an example taken from image processing. Here we have an image given as an two-dimensional array of pixels. We want to find those pixels having maximum brightness, that is, whose RGB values are $(255, 255, 255)$. The three brightest areas appears circled in Figure 3.4. This problem and others similar to this arise in contexts such as filtering adult content on the web, facial recognition or medical imaging.

Unlike the usual mathematical notation, we will refer to the elements in matrices by two indices, the first one pointing to columns and second one to rows.

Example. Given matrices T and P

$$T = \begin{pmatrix} \mathtt{a} & \mathtt{c} & \mathtt{b} & \mathtt{c} & \mathtt{b} & \mathtt{a} \\ \mathtt{b} & \mathtt{b} & \mathtt{c} & \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{d} & \mathtt{d} & \mathtt{a} & \mathtt{a} \\ \mathtt{d} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{c} \\ \mathtt{a} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{d} & \mathtt{c} \\ \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{a} & \mathtt{c} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{c} \end{pmatrix}, \quad P = \begin{pmatrix} \mathtt{b} & \mathtt{a} \\ \mathtt{a} & \mathtt{b} \end{pmatrix},$$

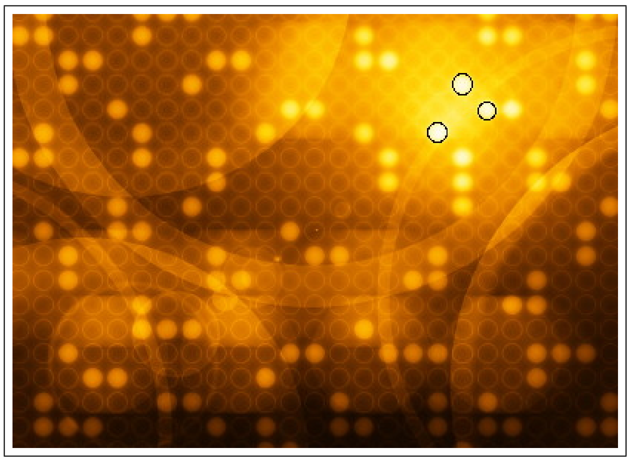


Figure 3.4: Two-dimensional string matching problem.

valid shifts of P in T are $(4, 0), (2, 3)$ and $(1, 5)$.

One of the main assets of the Karp-Rabin algorithm is its immediate generalization to $2D$ problems. Below there is the pseudocode for a $2D$ version. In this pseudocode we indexed characters starting at 1 and rows starting at 0 (columns point to characters and it is more natural to start at 1, but rows denote shifts and it seems more adequate to start at 0).

```
    RABIN-KARP-2D-MATCHER(P, T, d, q)
    # Initialization step .
1   $n_1 \leftarrow$ columns$[T]$
2   $n_2 \leftarrow$ rows$[T]$
3   $m_1 \leftarrow$ columns$[P]$
4   $m_2 \leftarrow$ rows$[P]$
    # Initializing numerical values for the rows of $P$ and $T$.
5   $a \leftarrow d^{m_1-1} \bmod q$
6   for $i_2 \leftarrow 0$ to $n_2 - 1$
7     $p_{i_2} \leftarrow 0$
8     $t_{i_2,0} \leftarrow 0$
9   for $i_2 \leftarrow 0$ to $n_2 - 1$
10    for $i_1 \leftarrow 1$ to $m_1$
11      if $i_2 < m_2$ then $p_{i_2} \leftarrow (dp_{i_2} \bmod q + P[i_1, i_2 + 1]) \bmod q$
12      $t_{i_1,0} \leftarrow (dt_{i_1,0} \bmod q + T[i_1, i_2 + 1]) \bmod q$
13  $t \leftarrow t_{1,0}$
    #Main body of algorithm.
14  for $s_2 \leftarrow 0$ to $n_2 - m_2$
15    for $s_1 \leftarrow 0$ to $n_1 - m_1$
16      if $t \bmod q = p_{s_2} \bmod q$ then
17        if $T[s_1 + 1..s_1 + m_1, s_2 + 1..s_2 + m_2] = P[1..m_1, 1..m_2]$
18        then print "$P$ occurs with shift $(s_1, s_2)$"
    #Managing the change of row in text $T$.
19      if $s_1 < n_1 - m_1$ then $t \leftarrow (d \cdot (t - T[s_1 + 1, s_2] \cdot a \bmod q)$
                                   $+T[s_1 + m_1 + 1, s_2]) \bmod q$
20      else if $s_2 < n_2 - m_2$ then $t \leftarrow t_{s_2,0}$
```

Table 3.3: The two-dimensional *RABIN-KARP-2D-MATCHER* algorithm.

There are several remarks to be made about the $2D$ version.

- The initialization step is more complex. It requires to compute all the initial $p_{i_2}, t_{i_2,0}$, for $i_2 = 0, \ldots, n_2 - 1$. This step takes $\Theta(m_1 + m_2 + m_1 n_2) = \Theta(m_1 n_2)$ time.

- The main body of the algorithm consists of two nested loops. Equality of the corresponding fingerprints is verified. If equality happens, the explicit character-to-character comparisons across the matrix is performed. Let v be the number of false matches between P and T. The time complexity is then

$$\Theta((n_1 - m_1)(n_2 - m_2) - v + v m_1 m_2).$$

  The worst case appears when $v = \Theta((n_1 - m_1)(n_2 - m_2))$, which leads to $\Theta((n_1 - m_1)(n_2 - m_2) \cdot m_1 m_2)$ time complexity. However, when the parameters of the algorithm are well chosen, v may be as low as $O(1)$. In that case, the overall time complexity is $\Theta((n_1 - m_1)(n_2 - m_2) + m_1 m_2) = \Theta(n_1 n_2 + m_1 m_2)$.

- The change of row has to be managed. That is done in lines 20–21.

3.6.2 The String-Matching Problem with Wild Charactes

We now introduce one of the most fundamental problems in string pattern recognition, namely, the string-matching problem with **wild characters** (sometimes called **gap characters**). A wild character, to be denoted by the Greek letter ϕ, is a character that matches any sequence of characters in the text. For example, the pattern $P = \{\texttt{abc}\phi\texttt{abc}\phi\}$ contains two wild characters and is found in texts $T_1 = \{\texttt{abcdcadbaccabc}\}$, $T_2 = \{\texttt{abcabc}\}$ or $T_3 = \{\texttt{abcdcadbaccabcddccabc}\}$ (P appears twice in text T_3).

This problem of string matching with wild cards is formally stated as follows.

The string-matching problem with wild characters (SMWC problem): Given a pattern with wild characters, find all the occurrences of P in T.

Matching patterns with wild cards is an ubiquitous problem in string pattern recognition. It appears in many practical contexts, but it is of paramount importance in Bioinformatics. Wild cards occur in **DNA transcription factors**. Transcription factors are proteins that bind to precise regions of DNA and regulate, either by suppression or activation, the transcription of DNA into RNA. Given two transcription factors, determining whether they belong to the same family (because they have similar binding sites) can be converted into a string pattern recognition problem. Figure 3.5 shows a piece of real DNA with some binding sites (labelled as Crea, X1nR and PacC).

ATCCGGCAAAATGAGGTTGAACCTGCATTTCTTCACGGCAG
ATCAAT**GCGGGG**AAGCAGTCGAGGCCTTACGCAGATCCCTC
Crea

CTATTCGAGGC**CCCCAG**CCGAGACACCACCGTAACTCTATC
Crea

ATGTCTCACCCTCAATTTGCTTATTTTG**CCTGGC**GGAATCG
PacC

TCCCGAAACGCCGATCGATTCAGCCTGATTGAATCAATACG
ACTTCTTCAC**GGCTAAA**TGAAGCAAGTTGCCT**GCCAAG**TCT
X1nr **PacC**

Figure 3.5: Binding sites in a piece of DNA (after [CSN+04]).

This piece of DNA can be represented as a string containing wild cards, namely, $\{\phi\texttt{GCGGGG}\phi\texttt{CCCCAG}\phi\texttt{CCTGGC}\phi\texttt{GGCTAAA}\phi\texttt{GCCAAG}\}$. See [Gus97] for more details on the relationship between computer science and biology; see [Lat97] for an overview on transcription factors.

Patterns with wild cars also appears in problems in music technology. Sometimes melodies are played with ornamentation (trills, mordents, cadences, appoggiaturas and others). A typical problem in melodic recognition is to automatically detect ornamentation and obtain the core melody behind. Figure 3.6 shows two melodies; the one in the first staff contains the first few measures of a very popular children's song *Twinkle, Twinkle Little Star* (in French known as *Ah, vous dirais-je, maman?*), while the second staff contains an ornamented version of that melody. Here the problem is to recognize the original melody from the ornamented version. The original version corresponds to string {`CCGGAAGFFEEDEC`}; the ornamented version includes wild cards. Again, this problem can be formulated in terms of a problem of string matching.

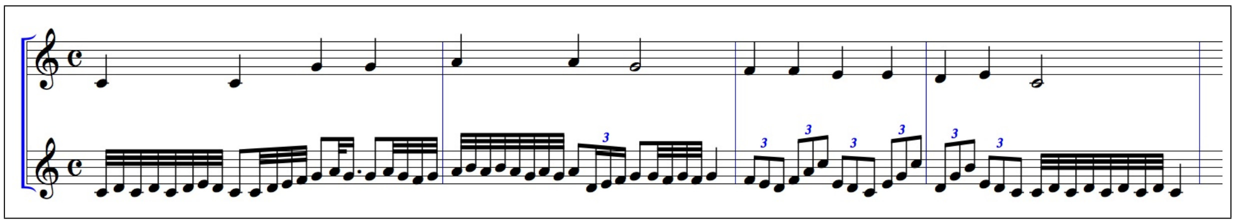


Figure 3.6: String matching with wild cards in music.

Exercises

$\boxed{1}$ Solve the existence problem of string matching with wild cards, that is, determine whether pattern P with wild cards occurs in text T. Analyse the running time.

3.7 Chapter Notes

In this chapter the naive and the Karp-Rabin algorithm were studied. We carried out a probabilistic analysis of the naive algorithm in the case characters in P and T were randomly drawn from the alphabet. As for the Karp-Rabin algorithm, worst-case and average-case complexities were examined. For the latter case a thorough study was carried out, which included the characterization and bounding of the probability of finding a false match. The randomized version of the algorithm was also described. The table below summarizes all the results.

Algorithm	Best-case	Worst-case	Average-case
Naive algorithm	$\Theta(n-m)$	$\Theta((n-m+1)m$	$\Theta(2(n-m+1))$
Karp-Rabin algorithm	$\Theta(n-m)$	$\Theta((n-m+1)m$	$\Theta(n+vm)$
Karp-Rabin randomized	$\Theta(n-m)$	$\Theta((n-m+1)m$	$\Theta(n+vm)$
Karp-Rabin $2D$	$\Theta(n_1n_2+m_1m_2)$	$\Theta((n_1-m_1)(n_2-m_2)m_1m_2)$	$\Theta(n_1n_2+m_1m_2)$

Table 3.4: Comparisons of the algorithms seen in Chapter 3.

The Karp-Rabin algorithm can also be viewed as a hash algorithm. A **hash function** is a function that takes a large set of data and maps it onto a smaller set. Obviously, the hash function will assign the same value to two or more different data (**collisions**) and it is desirable that such collisions be minimized. Frequently, the probability of collision is minimized through randomization. Fingerprints in the Karp-Rabin algorithm are just modular hash functions. See [CLRS01, MR95, Cro02] for an in-depth hash approach to the Karp-Rabin algorithm. To learn more about randomized algorithms we refer the interested reader to [MR95, MU05]. See [CL04, Cro02, GBY91] for references addressing implementation issues.

Chapter 4

String Matching: the Linear Algorithm

In this chapter we will introduce a linear-time algorithm to solve the string matching computation problem. Unlike the usual way to proceed, where algorithms are presented as they were published, here we will put forward a more recent linear-time algorithm. Although, this algorithm was not the first linear algorithm, it is conceptually relatively simple and fundamental and will help the reader understanding forthcoming algorithms later on. This algorithm is called the Z **algorithm** and is attributed to Gusfield [Gus96, Gus97].

Many string matching algorithms try to avoid computing some of the $n - m + 1$ shifts of the naive algorithm. To do so, they spend a small amount of time (as compared to the overall complexity) learning about the complexity of the pattern or the text. That learning step is called **pre-processing**. Algorithms such as the Knut-Morris-Pratt algorithm [KMP77], the Boyer-Moore algorithm [BM77] or the Apostolico-Giancarlo algorithm [AG86] are examples where the pattern undergoes pre-processing before actually being searched in the text. On the other hand, methods such as suffix trees are based on text pre-processing. Some of those algorithms are **alphabet-dependent**, that is, their time complexity depends on the size of Σ.

4.1 Definitions

Since pre-processing is used in more contexts other than string pattern recognition, we will denote by S an arbitrary string instead of using the usual letter P. Let S be a string and $i > 1$ one of its positions. Starting at i, we consider all the substrings $S[i..j]$, where $i \leq j \leq |S|$, so that $S[i..j]$ matches a prefix of S itself. Among all the possible values j may take, select the greatest one, $j_{\max}$ and denote the number $j_{max} - i + 1$ by $Z_i(S)$. If $S[i]$ is different from $S[1]$, then such a j does not exist and $Z_i(S)$ is set to 0. In other words, $Z_i(S)$ is defined as the maximum length such that $S[i..i + Z_i(S) - 1] \sqsubset S[1..Z_i(S)]$. When S is clear from the context, we will simplify the notation and just write $Z(S)$.

As an example, consider string $S = \{\texttt{aabadaabcaaba}\}$. The table below shows the $Z_i(S)$ values for S.

i	$S[i..i+Z_i(S)-1]$	$S[1..Z_i(S)]$	Does $Z_i(S)$ exist?	Possible values	$Z_i(S)$
2	a	a	Yes	1	1
3	b	a	No	0	0
4	a	a	Yes	1	1
5	d	a	No	0	0
6	aab	aab	Yes	$\{1,2,3\}$	3
7	a	a	Yes	1	1
8	b	a	No	0	0
9	c	a	No	0	0
10	aaba	aaba	Yes	$\{1,2,3,4\}$	4
11	a	a	Yes	1	1
12	b	a	No	0	0
13	a	a	Yes	1	1

Table 4.1: Computation of $Z_i(S)$ values.

For positive values of Z_i, define the Z**-box** at position i as the interval starting at i and ending at $i + Z_i - 1$. Geometrically, this is equivalent to drawing a box whose base is the interval $[i, i + Z_i - 1]$. Figure 4.1 displays the Z-boxes corresponding to string $S = \{\texttt{aabadaabcaabc}\}$. When $Z_i = 1$, the box is reduced to a line segment.

For a fixed $i > 1$, consider all the Z-boxes starting at j, where $2 \leq j \leq i$. Among all those Z-boxes, select the rightmost endpoint and call it r_i. For the box realizing value r_i, call l_i its starting point. Figure 4.2 shows all the Z-boxes associated to string S. Arrows point to the Z-boxes of each element. The values of r_i and l_i are displayed in the table below the figure.

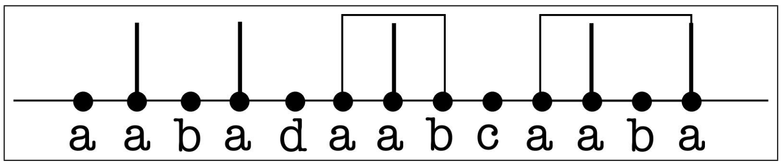
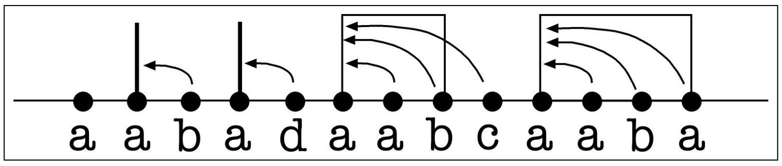


Figure 4.1: Z-boxes.



i	1	2	3	4	5	6	7	8	9	10	11	12	13
l_i	–	2	2	4	4	6	6	6	6	10	10	10	10
r_i	–	2	2	4	4	8	8	8	8	13	13	13	13

Figure 4.2: Values r_i and l_i for string S.

Exercises

$\boxed{1}$ Given string $S = \{\texttt{aaabcaabdaabcaaab}\}$, compute its values $Z_i(S)$, draw its Z-boxes and obtain values r_i and l_i, as done in the text.

4.2 The Pre-Processing Step

We carry on by describing the Z algorithm in detail. When computing values Z_k, the algorithm only needs values r_{k-1} and l_{k-1}. Because of this, we will remove subscripts from variables r_k and l_k, thus simplifying the notation. The algorithm needs to know no more than two values Z_i, but those may be arbitrary, and subscripts cannot be removed.

The algorithm starts by explicitly computing Z_2. In order to do so, substrings $S[1..|S|]$ and $S[2..|S|]$ are compared until a mismatch occurs. Value Z_2 verifies that $S[1..Z_2] = S[2..Z_2]$ and $S[1..Z_2+1] \neq S[2..Z_2+1]$. Once Z_2 is obtained the algorithm proceeds iteratively. Let us compute the k-th value Z_k. Assuming that previous values are already computed, the algorithm performs the following case analysis.

Case 1: $k > r$. In this case the algorithm cannot use previous values of Z_i to construct the Z-box for k. It simply compares substrings starting at k with prefixes of S until a mismatch is found. Thus, Z_k is set to the length of the matching substring. Furthermore, $l = k$ and $r = k + Z_k - 1$. See Figure 4.3.

Case 2: $k \leq r$. This time some previous information may be employed to compute Z_k. Since $k \leq r$, k must be contained in a Z-box. By the definition of l and r, $S[k]$

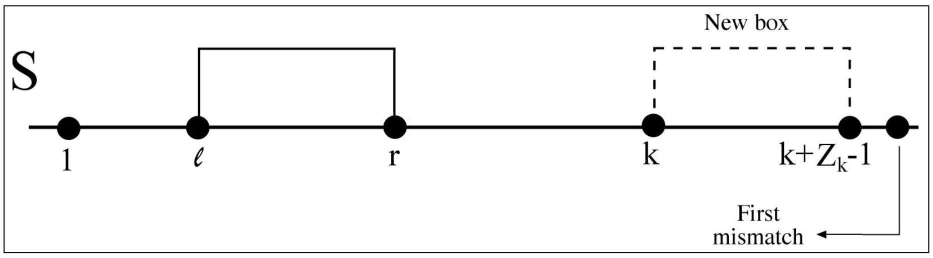


Figure 4.3: Case 1 of the Z algorithm.

belongs to substring $S[l..r]$. Denote by A that substring. Substring A matches a prefix of S. Therefore, character $S[k]$ also appears in substring $S[1..r-l+1]$ at position $k^{'} = k-l+1$. Note that substring $S[k..r]$ is contained in substring $S[1..r-l+1]$ (also note that $Z_l = r-l+1$). We denote substring $S[k..r]$ by B. The copy of substring B in the prefix $S[1..Z_l] = A$ is substring $S[k^{'}..Z_l]$. Figure 4.4 illustrates all these definitions.

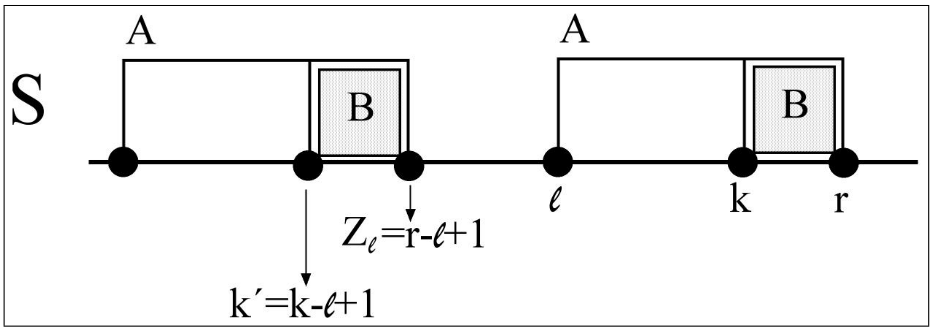


Figure 4.4: Case 2 of the Z algorithm.

When $k^{'}$ was computed, a Z-box of length $Z_{k'}$ was obtained; we will call it C; see Figure 4.5. Such Z-box is also a prefix of S. Therefore, if we produce a substring from character k matching a prefix of S, it will have at least length the minimum of $Z_{k'}$ and $|B|$. Recall that $|B| = r-k+1$.

According to the value of $\min(Z_{k'}, |B|)$ two more cases arise:

Case 2a: $Z_{k'}$ is less than $|B|$. In this case the Z-box at k is the same as the one at $k^{'}$. Hence, Z_k is set to $Z_k^{'}$. Values r and l remain unchanged. See Figure 4.5

Case 2b: $Z_{k'}$ is greater or equal than $|B|$. In this case it substring $S[k..r]$ is a prefix of S and value Z_k is at least $|B| = k-r+1$. However, substring B could be extended to the right while remaining a prefix of S. That would make Z_k be greater than $Z_{k'}$. The algorithm, starting at position $r+1$, searches for the first mismatch. Let q be the position of that mismatch. Then, Z_k is set to $q-1-(k-1) = q-k$, r is set to $q-1$ and l is set to k. See Figure 4.6

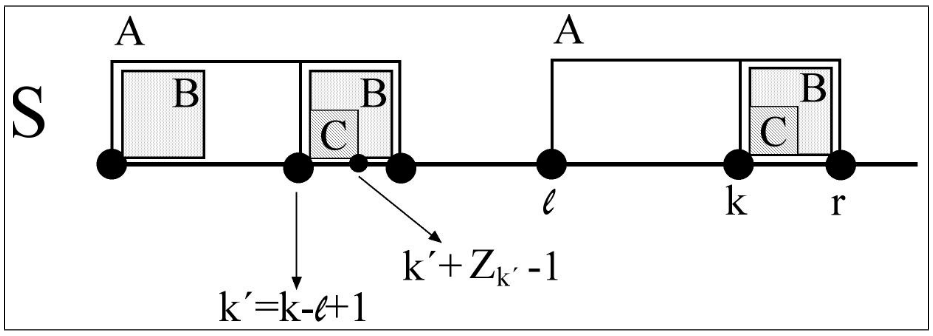


Figure 4.5: Case 2a of the Z algorithm.

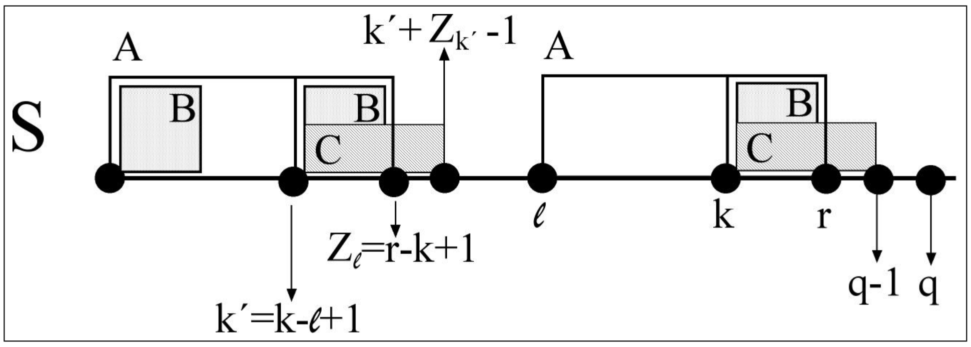


Figure 4.6: Case 2b of the Z algorithm.

Example. Let us trace the algorithm with string $S = \{$`aabcdaabcxyaabcdaabcdx`$\}$. The table below shows values taken by variables Z_i, l_i and r_i as well as which cases the algorithm runs through. Substrings A, B and C are also displayed for Case 2a and Case 2b.

Character	a	a	b	c	d	a	a	b	c	x	y
i	1	2	3	4	5	6	7	8	9	10	11
Z_i	–	1	0	0	0	4	1	0	0	0	0
l_i	–	2	2	2	2	6	6	6	6	6	6
r_i	–	2	2	2	2	9	9	9	9	9	9
Case	–	1	1	1	1	1	2a	2a	2a	1	1
Boxes A, B, C	–	–	–	–	–	–	$A = \{$aabc$\}$ $B = \{$abc$\}$ $C = \{$a$\}$	Same A, $B, C = \{\varepsilon\}$	Same A, B, C	–	–

Character	a	a	b	c	d
i	12	13	14	15	16
Z_i	9	1	0	0	0
l_i	12	12	12	12	
r_i	20	20	20	20	
Case	1	2a	2a	2a	2a
Boxes A, B, C	–	$A = \{$aabcdaabc$\}$ $B = \{$abcdaabc$\}$ $C = \{$a$\}$	Same A, $B, C = \{\varepsilon\}$	Same A, B, C	Same A, B, C

Character	a	a	b	c	d	x
i	17	18	19	20	21	22
Z_i	5	1	0	0	0	0
l_i	17	17	17	17	17	17
r_i	21	21	21	21	21	21
Case	2b	2a	2a	2a	2a	1
Boxes A, B, C	$A = \{$aabcdaabc$\}$ $B = \{$aabc$\}$ $C = \{$a$\}$, $q = 22$	$A = \{$aabcd$\}$, $B, C = \{$a$\}$	Same A, $B, C = \{\varepsilon\}$	Same A, B, C	–	–

Table 4.2: Tracing the Z algorithm.

Pseudocode for this algorithm is left to the reader as an exercise. Translating the main ideas to pseudocode should be straightforward. In the next two theorems we will prove correctness and complexity analysis of the Z algorithm.

Theorem 4.2.1 *The pre-processing algorithm correctly computes all the Z_i, r_i and l_i values.*

Proof: In Case 1 value Z_k is always correctly computed as it is obtained by explicit comparisons. None of the Z-boxes starting at a position j, $2 \leq j < k$, ends at or after position k. Therefore, the new value $r = k + Z_k - 1$ is the correct one for r. Setting $l = k$ is also correct.

As for Case 2a, we know that there is a substring $S[k..k + Z_{k'} - 1]$ matching a prefix of S. Such substring has length $Z_{k'} < |B|$. If that substring could be extended to the right, because of the presence of more matching characters, then that would contradict the property of $Z_{k'}$ being maximal. Indeed, if character $Z_{k'}+1$ matches character $k+Z_{k'}$, it follows that $Z_{k'}$ is not the maximal value such that $S[k'..Z_{k'}] \sqsubset S[1..Z_{k'}]$.

It remains to prove Case 2b. B is a prefix of S and the algorithm just extends B to the right by making explicit comparisons. Hence, Z_k is correctly computed. Also, the new r, which is set to $k+Z_k-1$ ($=q-1$), is greater than any of the other Z-boxes starting below position k. Again, r is correctly updated. In both Cases 2a and 2b l is also correctly updated. ▮

Theorem 4.2.2 *The pre-processing algorithm runs in linear time on the size of S.*

Proof: The Z algorithm performs iterations and character-to-character comparisons. No more than $|S|$ comparisons are made (that corresponds to variable k). The number of comparisons is split between matches and mismatches. The number of mismatches is bounded by $|S|$, the worst case being when all comparisons are mismatches. Then number of matches is kept in variable r_k. Note that r_k is non-decreasing in all the three cases considered by the Z algorithm. If Case 1, r_k is always made bigger than r_{k-1}. If Case 2a, r_k remains unchanged. If Case 2b, r_k is changed to $q-1$, which by construction is greater than r_{k-1}. Therefore, since $r_{|S|} \leq |S|$, the number of matches is bounded by $|S|$.

Thus, we conclude the whole algorithm runs in $O(|S|)$ time. ▮

Exercises

$\boxed{1}$ Write a pseudocode for the Z algorithm.

$\boxed{2}$ Consider the Z algorithm and box A appearing in Case 2a and 2b. Can that box and its prefix overlap? Justify your answer.

$\boxed{3}$ Consider the Z algorithm. If $Z_2 = a > 0$, is it true that all the values $Z_3, \ldots, Z_{a+2}$ can be obtained without further comparisons? Justify your answer.

$\boxed{4}$ Analyse the complexity of the Z algorithm including the constant hidden behind the asymptotic notation. It should be similar to the analysis of *INSERTION-SORT* presented in Chapter 2. Use the pseudocode you were asked to write in the previous section.

4.3 The Linear-Time Algorithm

The Z algorithm can be turned into a string matching algorithm that runs in linear time $O(n+m)$ and linear space $O(n+m)$. The idea is witty and simple. Gusfield, who came up with the idea, claims that "it is the simplest algorithm we know of" ([Gus96], page 10).

Let \$ be a character found neither in P nor in T. Build the string $S = P\$T$. String S has length $n+m+1$, where $m \leq n$. Run the Z algorithm on S. The resulting values Z_i hold the property that $Z_i \leq m$. This is due to the presence of character \$ in S. More important, any value $Z_i = m$ identifies an occurrence of P in T at position $i - (m+1)$. By definition of Z_i,

$$S[i..i + Z_i - 1] = S[i..i + m - 1] = T[i - m - 1..i - 1] = S[1..m] = P[1..m].$$

The converse is also true. When P is found in T at position i, the Z algorithm will output $Z_{m+1+i} = m$. We finish by stating the following theorem.

Theorem 4.3.1 *The string matching computation problem can be solved in linear time and space by using the Z algorithm.*

We will refer to the Z algorithm both when it is used to compute the longest substring that is a prefix of S and when it is used to solve a string matching problem. The context will make clear the meaning in each case.

Example. Let $P = \{$`aab`$\}$ and $T = \{$`abcaaabxy`$\}$. Their respective lengths are $m = 3$ and $n = 9$. Form S by setting $S = P\$T = \{$`aab$abcaabxy`$\}$. Table 4.3 shows the Z_i values for S.

i	1	2	3	4	5	6	7	8	9	10	11	12
$Z_i(S)$	–	1	0	0	1	0	0	3	1	0	0	0

Table 4.3: Solving the SMC problem by using the Z algorithm.

So, at position $8 - (3 + 1) = 4$ we find pattern P in T.

Exercises

$\boxed{1}$ Let $P = \{$`abca`$\}$ and $T = \{$`abcbabcaay`$\}$. Trace the Z algorithm to find all the occurrences of P in T.

$\boxed{2}$ Can the Z algorithm be randomized so that its average-case is sublinear? How is the behaviour of the Z algorithm like when characters of P and T are randomly drawn from the alphabet?

$\boxed{3}$ **Match-Count**. Consider two n-length strings S_1, S_2 and a parameter k, where $1 \leq k \leq n$. There are $n - k + 1$ substrings of length k in each string. Take a substring from each string, align them and count the number of matches. The total number of matches is called the **match-count** of strings S_1 and S_2. A naive algorithm to compute the match-count would take $O(kn^2)$ time. Improve that complexity and produce an $O(n^2)$-time algorithm.

$\boxed{4}$ Can the Z algorithm be easily generalized to solve the 2D string matching problem? Justify your answer.

4.4 More String Matching Problems

An **on-line algorithm** is one that does not have the whole input available from the beginning, but receives it piece by piece. In particular, the algorithm outputs the solution at step i before reading the $(i+1)$-th input symbol. For instance, selection sort requires that the entire list be given before sorting, while insertion sort does not. Selection sort is an **off-line algorithm**. A **real-time algorithm** is an on-line algorithm with the extra condition that a constant amount of work is done between consecutive readings of the input. Intrinsically, off-line and on-line algorithms are quite different. An on-line algorithm, since it cannot examine the whole input, may make decisions that later will turn out not to be optimal. Although on-line and real-time algorithms does not appear often in practice, they still have theoretical interest. See [AHU83, Zvi76] for more information.

The real-time string matching problem is defined by assuming that the text is not entirely available and the pattern is. The text is given to the algorithm character by character in its order. The algorithm has to solve the problem with the information received so far by doing a constant amount of work. Formally, the problem is stated as follows.

The real-time string matching problem (RTSM problem): Find all occurrences of pattern P in text T, where T is input one character at a time.

Notice that trying to solve this problem by just checking whether the last m characters received actually form pattern P is equivalent to the off-line naive algorithm. Its time complexity would be $O(nm)$.

Exercises

$\boxed{1}$ Give an $O(n+m)$ algorithm to solve the real-time string matching problem. Prove the correctness of your algorithm and analyse its time complexity.

4.5 Chapter Notes

Since the Z algorithm solves the SMC problem in linear time, it is simple and efficient, it would seem that it is not worth examining other more complex algorithms. That is not the case. In the next few chapters we will examine several algorithms that are relevant because historical reasons, good performance in practice, good average-case complexity and ease of generalization to other string matching problems.

Chapter 5

Suffix Trees and its Construction

5.1 Introduction to Suffix Trees

Sometimes fundamental techniques do not make it into the mainstream of computer science education in spite of its importance, as one would expect. Suffix trees are the perfect case in point. As Apostolico[Apo85] expressed it, suffix trees possess "myriad of virtues." Nevertheless, even elementary texts on algorithms, such as [AHU83], [CLRS01], [Aho90] or [Sed88], present brute-force algorithms to build suffix trees, notable exceptions are [Gus97] and [Ste94]. That is a strange fact as much more complicated algorithms than linear constructions of suffix trees are found in many computer science curricula. The two original papers that put forward the first linear-time algorithms have a reputation of being obscure and that they might have contributed to this situation. In the following, we will try to give a clear and thorough explanation of their construction.

Initially, building suffix trees appeared in order to solve the so-called **substring problem**. This problem can be stated as follows.

The substring problem: Pre-process text T so that the computation string matching problem is solved in time proportional to m, the length of pattern P.

This problem has many more applications than the reader can imagine. Perhaps, the most immediate one is performing intensive queries on a big database, which is represented by T. Once the suffix tree for T is built each query is proportional to $O(m)$, not like the algorithms seen so far, which would take $O(n+m)$ time. However, the most important feature of suffix trees is the way it exposes the internal structure of a string and how it eases the access to it.

Among the applications of suffix trees we can mention solving the exact string matching problem (both SME and SMC problems), the substring problem, the longest common substring of two strings problem and the DNA contamination problem. All these problems will be dealt with in the next few sections. For many other applications of suffix trees, the reader is referred to [Gus97] and [Apo85].

In 1973 Weiner [Wei73] presented the first linear-time algorithm for constructing suffix trees. This 11-page paper had an obscure terminology that made it hard to read and be understood. Three years later McCreight [McC76] gave a different algorithm, much better in terms of space efficiency, which was also linear. McCreight's paper was much more streamlined than Weiner's. For twenty years no new algorithm was come up with until 1995. In that year Ukkonen [Ukk95] put forward a clean, easy-to-understand algorithm to build suffix trees. Following Gusfield's presentation [Gus97], we will start by showing a naive algorithm and we will refine it, from top to bottom, until obtaining the desired linear-time algorithm.

5.2 Definitions and the Naive Algorithm

Again, as it happened with the Z algorithm, not only suffix trees are used in string matching, but also in other contexts. Thus, we will describe suffix trees on a generic string S of length s (not to confuse with the shift s used in previous chapters).

A **suffix tree** of a string S is a tree with the following properties:

SF1: It has exactly s leaves numbered from 1 to s.

SF2: Except for the root, every internal node has at least two children.

SF3: Each edge is labelled with a non-empty substring of S.

SF4: No two edges starting out of a node can have string-labels beginning with the same character.

SF5: The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix $S[i..m]$, for $i = 1, \ldots, s$.

Example. Consider string $S = \{\texttt{abcabx}\}$. Figure 5.1 shows the suffix tree corresponding to S.

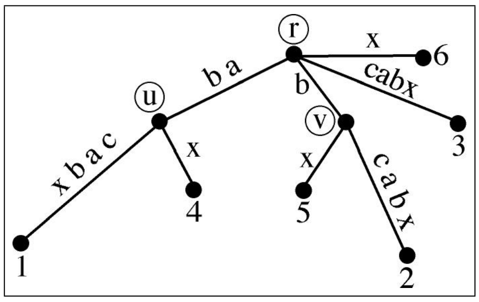


Figure 5.1: The suffix tree of a string.

We can easily check that properties SF1–SF4 are satisfied. As for the fifth property, note that for leaf 4, says, the path from r to 4 is $\{\texttt{abx}\}$, which gives the suffix $S[4..6]$. From now on, the path between two nodes u, v will be denoted by $u \longrightarrow v$. Path $r \longrightarrow 5$ is $S[5..6] = \{\texttt{bx}\}$ and path $r \longrightarrow 2$ spells out $S[2..6] = \{\texttt{bcabx}\}$. (Note the reader that some edge-labels are read from left to right and others right to left; do not be confused by this fact.)

However, there is a catch hidden in the above definition. The definition may suggest that any string must have an associated suffix tree, but that is not true in all instances. The problem arises when one suffix of S matches a prefix of another suffix. In that case it is not possible to build a suffix tree satisfying the definition given above. To illustrate this situation consider string $S' = S - \{\texttt{x}\} = \{\texttt{abcab}\}$. Now, suffix $S'[5..6] = \{\texttt{ab}\}$ matches a prefix of suffix $S'[1..6] = \{\texttt{abcab}\}$. If suffix $S'[5..6] = \{\texttt{ab}\}$ ends at a leaf, necessarily labels $\{\texttt{ab}\}$ and $\{\texttt{abcab}\}$ will start from the root, which would violate property SF5; see Figure 5.2 (a). On the other hand, if suffix $S'[1..6] = \{\texttt{abcab}\}$ is found as the concatenation of $S'[1..2] = \{\texttt{ab}\}$ and $S'[3..5] = \{\texttt{cab}\}$, then there will not be a path from the root to a leaf spelling out $S'[1..2] = \{\texttt{ab}\}$; see Figure 5.2 (b).

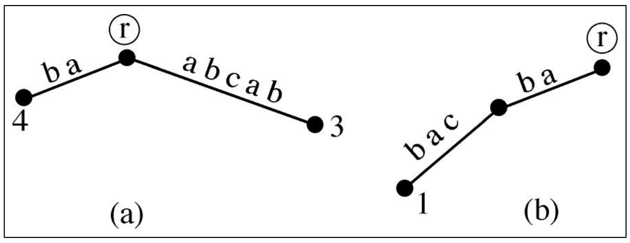


Figure 5.2: The bad cases of the definition of suffix tree.

In order to avoid this problem, from now on we will add a special character to the end of string S. This character, customarily denoted by \$, does not appear in string S. Character

\$ is called the **termination character**. This way, no suffix of S matches a prefix of another suffix of S. In the following, the suffix tree of $\mathcal{T}(S)$ will be defined as the suffix tree of $\mathcal{T}(S\$)$.

We call the **label of a path** starting at the root and ending at a node the concatenation of the strings in its order found along that path. The **path-label of a node** is the label of the path from the root to the node. Given a node v, the **string-depth** of v is the number of characters in the edge ending at v.

Example. In the suffix tree of $S = \{\texttt{abcabx}\}$ depicted in Figure 5.1 the label of path $r \longrightarrow 2$ is $\{\texttt{bcabx}\}$. The path-label of node u is $\{ab\}$. The string-depth of u is 2, the number of characters of string $\{\texttt{ab}\}$.

Next, we describe a brute-force algorithm to build a suffix tree of a string. We call $\mathcal{T}(S)$ the suffix tree of string S. The algorithm works in an incremental way, by processing the m suffixes $S[i..m]$ of S, from $i = 1$ to m, one by one. Let $\mathcal{T}_i$ be the tree obtained at step i. We will show how to construct $\mathcal{T}_{i+1}$ from $\mathcal{T}_i$. Tree $\mathcal{T}_m$ will be the final tree $\mathcal{T}(S)$.

At the first step, $S[1..m]\$$ is inserted in an empty tree and suffix tree $\mathcal{T}_1$ is composed of a unique node. At step $i+1$, suffix $S[i+1..m]\$$ is inserted in $\mathcal{T}_i$ as follows. Traverse the tree starting at the root r and find the longest prefix that matches a path of $\mathcal{T}_i$. If such prefix is not found, this is because none of the previous suffixes $S[j..m]\$$, for $j = 1$ to $j = i$, starts by character $S[i+1]$. In this case, a new leaf numbered $i+1$ with edge-label $S[i+1..m]\$$ is created and joined to the root.

Thus, assume there is a path such that $S[i+1..m]\$$ is a prefix of maximal length of that path. Because of the presence of the termination character \$, the prefix cannot be a substring of a suffix entered into the tree early on. Therefore, there is a character at position k such that $S[i+1..k]\$$ is the prefix; let $S[k..m]\$$ be the non-matching substring. Let (u, v) be the edge of the path where character $S[k]$ is found. The algorithm creates a new node w and a leaf numbered $i+1$. Node w splits edge (u, v) into edges (u, w) and (w, v), and edge joining w and leaf $i+1$ receives edge-label $S[i+1..k]\$$. The algorithm finishes when the termination character is inserted.

For illustration purposes, let us follow the construction of the suffix tree for $S\$ = \{\texttt{abcab\$}\}$. The first tree $\mathcal{T}_1$ is just $S[1..5]\$ = \{\texttt{abcab\$}\}$, as shown in Figure 5.3 (a). Next, $S[2..5]\$ = \{\texttt{bcab\$}\}$ is inserted into $\mathcal{T}_1$. Since $b \neq a$, there is no common prefix of $S[1..5]\$$ and $S[2..5]\$$. Thus, a new leaf numbered 2 is created. The same situation happens with the third suffix $S[3..6]\$ = \{\texttt{cab\$}\}$; see Figure 5.3 (b) and (c).

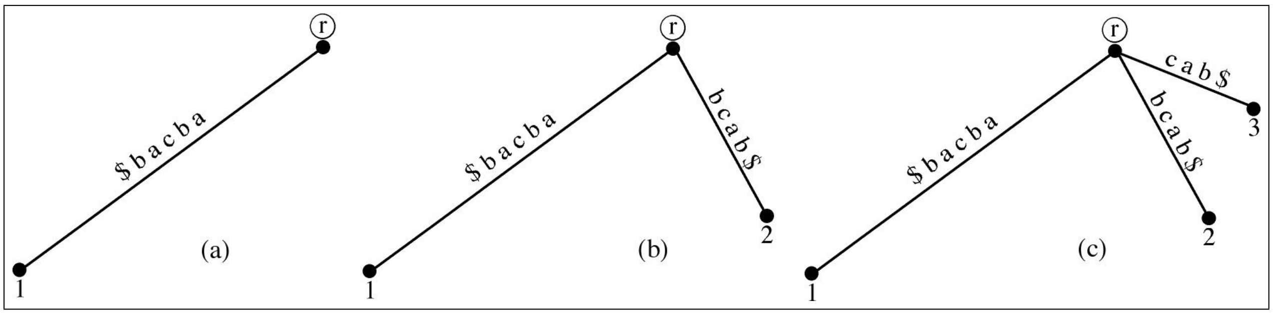


Figure 5.3: Processing the first three suffixes.

The fourth suffix $S[4..5]\$ = \{\texttt{ab\$}\}$ shares a common prefix with $S[1..5]\$ = \{\texttt{abcab\$}\}$. The longest common prefix is $\{\texttt{ab}\}$. Therefore, a new node u is inserted, which splits $\{\texttt{abcab\$}\}$ into substrings $\{\texttt{ab}\}$ and $\{\texttt{cab\$}\}$, as shown in Figure 5.4 (a). Moreover, leaf 4 is created. Next, suffix $S[5..5]\$ = \{\texttt{b\$}\}$ is processed. Suffix $S[5..5]$ is found as a prefix of $S[2..5]\$ = \{\texttt{bcab\$}\}$. Consequently, edge $(r, 2)$ is split into edges (r, v) and $(v, 2)$ and edge $(v, 5)$ is also added to the suffix tree; see Figure 5.4 (b). Finally, the termination character $\{\$\}$ is processed, which merely produces an isolated edge, depicted in Figure 5.4 (c).

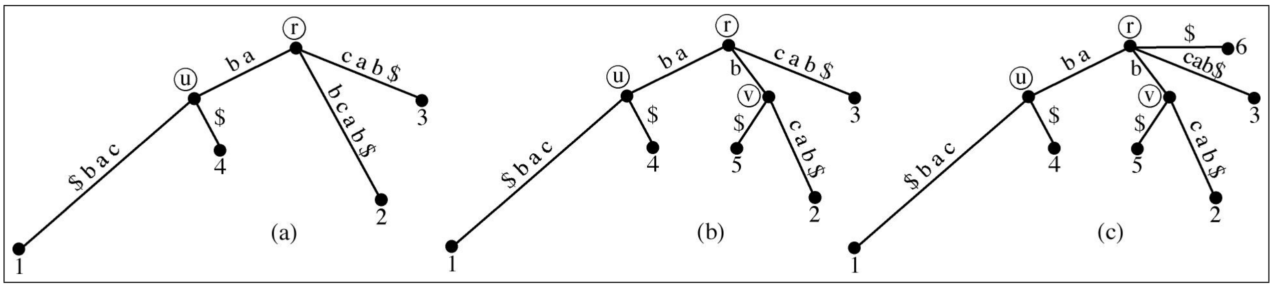


Figure 5.4: Processing the last three suffixes.

Theorem 5.2.1 *The naive algorithm described above correctly constructs the suffix tree of a string.*

Proof: The proof of this theorem is left as an exercise to the reader. ∎

Theorem 5.2.2 *The naive algorithm runs in $\Theta(s^2)$, where s is the length of string S.*

Proof: At step i the algorithm finds the longest common prefix between $S[i..s]$ and the previous i suffixes $S[1..s]\$, \ldots, S[i-1..s]\$$. Let $S[i..k]\$$ be such prefix. Then $k+1$ comparisons have been performed to identify such prefix. After that, the remaining of the suffix, $S[k+1..s]\$$ has to be read and assigned to the new edge. Therefore, the total amount of work at step i is $\Theta(i)$. Hence, the overall complexity is $\sum_{i=1}^{s} \Theta(i) = \Theta(s^2)$. ∎

Exercises

$\boxed{1}$ Suppose you want to use the Karp-Rabin algorithm to solve the substring problem. What is the best you can do?

$\boxed{2}$ Build and draw the suffix trees for string $S_1 = \{\texttt{abcdefg}\}, S_2 = \{\texttt{aaaaaaax}\}, S_3 = \{\texttt{abcabcx}\}$. Assume that the last character is the termination character.

$\boxed{3}$ Any suffix tree will always have at least an edge joining the root and a leaf, namely, the one corresponding to the termination character (it will be edge $(r, s+1)$). If we do not count that edge, is there a string whose suffix tree is a complete binary tree?

$\boxed{4}$ Prove that the naive algorithm to build suffix trees given in the text is correct.

$\boxed{5}$ Find out what the best case for the naive algorithm for building suffix trees is.

$\boxed{6}$ With regard to string S, what is the degree of the root of $\mathcal{T}(S)$? What is the number of internal nodes of $\mathcal{T}(S)$?

$\boxed{7}$ Given a string S, its **reverse string** is denoted by S^r. The reverse string is defined as the string S output in reverse order. Is there any relation between $\mathcal{T}(S)$ and $\mathcal{T}(S^r)$?

5.3 Ukkonen's Linear-Time Algorithm

Ukkonen's original algorithm is described in terms of finite automata. Although that choice may result in a more compact, concentrated exposition of the algorithm, we prefer to give an explanation without that terminology, as Gusfield [Gus97] does. Ukkonen's idea for computing suffix trees in linear time is a combination of several nice insights into the structure of suffix trees and several clever implementation details. The way we will explain the algorithm is to start off by presenting a brute-force algorithm, and after that introduce Ukknonen's speed-ups so that the linear-time algorithm is achieved. Ukkonen's algorithm computes suffix trees from another data structures, the so-called implicit suffix trees. Computing implicit suffix trees in a straightforward way does not give linear complexity. Special pointers called suffix links are then implemented to speed up tree traversal. In addition, edge-label compression is also applied so that memory space is reduced. Finally, a couple of properties, the halt condition and the fast leaf update rule, will take part to skip redundant insertions in the tree and speed up others.

5.3.1 Implicit Suffix Trees

An **implicit suffix tree** for a string S is a tree obtained from $\mathcal{T}(S\$)$ by performing the following operations:

1. Remove all the terminal symbols \$.

2. From the resulting tree, remove all edges without label.

3. Finally, from the resulting tree, remove all nodes that do not have at least two children.

The implicit tree associated with S will be denoted by $\mathcal{I}(S)$. The implicit tree of $S[1..i]$ is defined as the implicit tree of $S[1..i]\$$; it will be denoted by $\mathcal{I}_i(S)$. When S is fixed, we will just write $\mathcal{I}$ or $\mathcal{I}_i$. Figure 5.5 illustrates the implicit suffix tree construction for string $S\$ = \{\texttt{abcab\$}\}$.

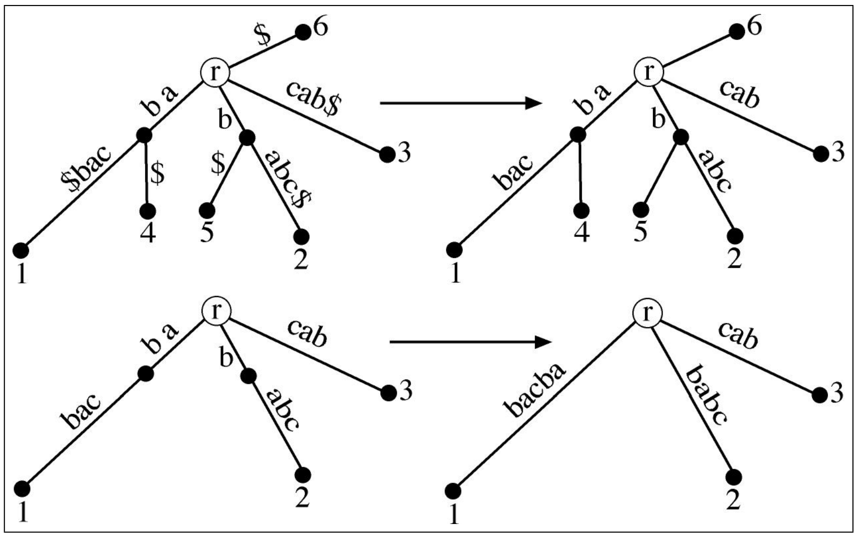


Figure 5.5: Implicit suffix trees.

Implicit trees are introduced because (hopefully) they will require less space, one of the reasons explaining the high complexity of the naive construction of suffix trees. The following result characterizes when an implicit suffix tree possesses fewer leaves than the original suffix tree.

Theorem 5.3.1 *Let S be a string of length s. Its implicit suffix tree will have less than s leaves if and only if at least one of its suffixes is a prefix of another suffix of S.*

Proof: The proof has two parts.

$\Longrightarrow$) Assume that $\mathcal{I}(S)$ has fewer than s leaves. That can only happen because there was an edge (u, i) ending at leaf i with label \$. By property SF2, node u has at least two children and, therefore, path $r \longrightarrow u$ spells out a common prefix for two suffixes of S.

$\Longleftarrow$) Assume that there exists at least one suffix of S (but not of $S\$$) that is a prefix of another suffix of S. Among all nested suffixes, select the one with minimum length. Let $S[i..s]$ and $S[j..s]$ be those suffixes such that $1 \leq i < j \leq s$ and $S[j..s] \sqsubset S[i..s]$. When suffix $S[j..s]$ is inserted in the tree, a new leaf with label \$ is created. This edge will be later removed when the suffix tree is pruned. ▮

In particular, if S ends by a character not found anywhere else in S, this result guarantees that $\mathcal{T}(S\$)$ and $\mathcal{I}(S\$)$ have $s+1$ and s leaves, respectively. Figure 5.7 shows the implicit suffix tree for string $S = \{\texttt{abcabd}\}$, which has 6 leaves, just one less than $\mathcal{T}(S\$)$.

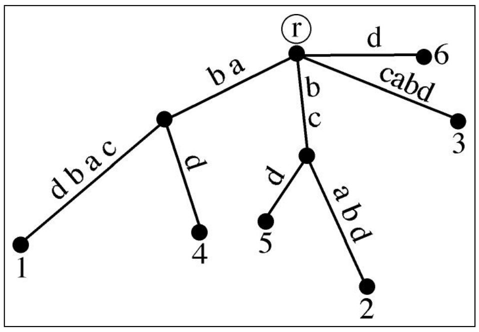


Figure 5.6: An implicit tree with one leaves less than $\mathcal{T}(S\$)$.

Exercises

$\boxed{1}$ Obtain the suffix trees and their associated implicit suffix trees for strings $S_1 = \{\texttt{abcdefg}\}, S_2 = \{\texttt{aaaaaaa}\}$ and $S_3 = \{\texttt{abcabc}\}$.

$\boxed{2}$ With regard to string S, what is the degree of the root of $\mathcal{I}(S)$? What is the number of internal nodes of $\mathcal{I}(S)$?

$\boxed{3}$ Given the following implicit suffix tree, find the suffix tree it comes from.

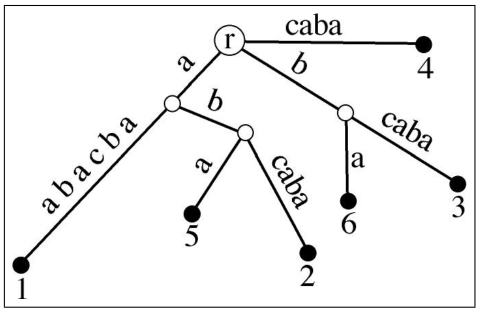


Figure 5.7: Computing suffix trees from their implicit counterparts.

5.3.2 Suffix Extension Rules

Ukkonen's algorithm builds the implicit suffix tree $\mathcal{I}_{i+1}(S)$ from $\mathcal{I}_i(S)$ in an incremental way. In his terminology that step from $\mathcal{I}_i(S)$ to $\mathcal{I}_{i+1}(S)$ is called a **transition**, but it is also known as a **phase** ([Gus97]). Unlike the naive algorithm, Ukkonen's algorithm processes string S by introducing the consecutive prefixes of S $S[1..i]$, for $i = 1$ to $i = s$, one by one. As we will see later, there is a strong reason to proceed so. Furthermore, each phase is decomposed into suffix extensions. If we are in phase i, Ukkonen's algorithm will take prefix $S[1..i]$ and insert all its suffixes $S[j..i]$, for $j = 1$ to $j = i$, in its order in the tree. Each of these insertions is called a **suffix extension**, which is produced by the following procedure. Assume we are in suffix extension j of phase $i + 1$. The procedure has the following steps:

1. Find the unique path that spells out suffix $S[j..i]$.
2. Find the end of such path.
3. Extend suffix $S[j..i]$ by adding character $S[i + 1]$.

The extension actually needs a more precise description as three cases may arise. The following rules, the **extension rules**, specify the three cases for suffix extension; see Figure 5.9.

Rule 1: Path containing $S[j..i]$ ends at a leaf. Character $S[i + 1]$ is simply added to $S[j..i]$ to produce label $S[j..i]S[i + 1] = S[j..i + 1]$.

Rule 2: Path containing $S[j..i]$ does not end at a leaf. Next character to label $S[j..i]$ is not character $S[i + 1]$. Split the edge and create a new node u and a new leaf numbered j; assign character $S[i + 1]$ to the label of $(u, i + 1)$. The new leaf represents the suffix starting at position j.

Rule 3: Path containing $S[j..i]$ does not end at a leaf. Next character to label $S[j..i]$ is character $S[i + 1]$. Then, $S[i + 1]$ is already in the tree. No action is taken.

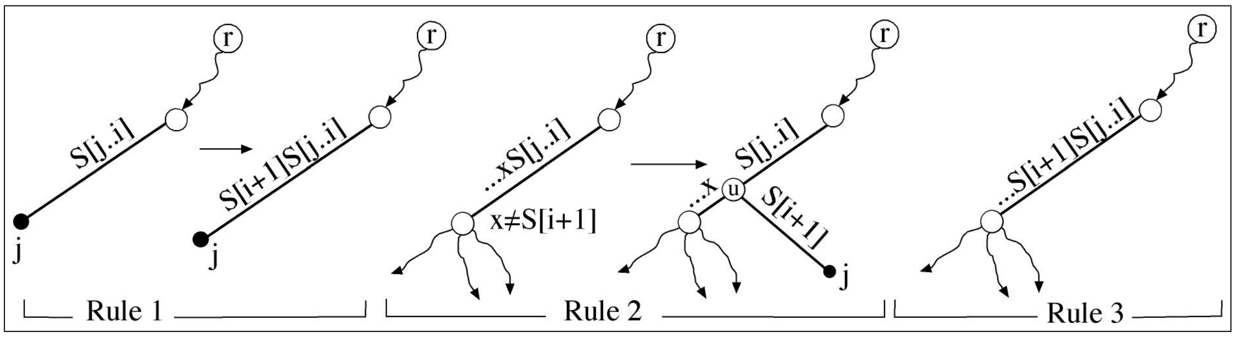


Figure 5.8: Suffix extension rules.

Example. The implicit suffix tree of string $S = \{\texttt{abba}\}$ is displayed in Figure 5.9.

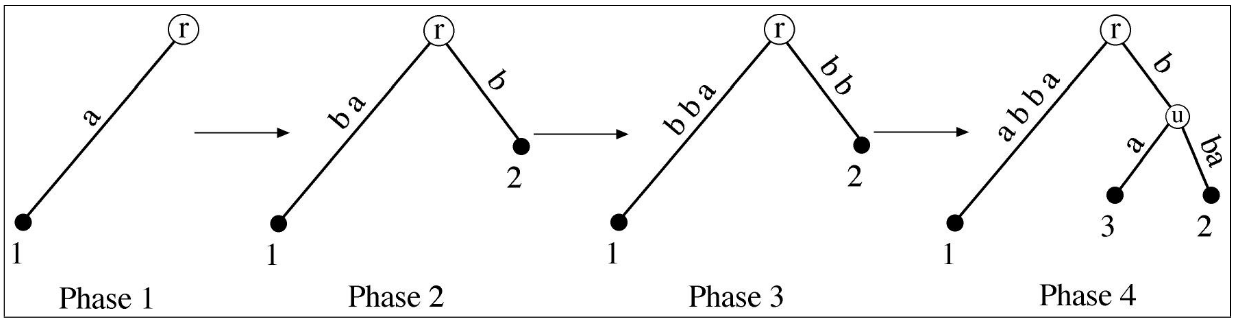


Figure 5.9: Suffix extension rules.

Suffix rule extensions have been traced for this string and the results are in Table 5.1.

Char.	Phase	Extension	Rule	Node	Label
{a}	1	1	2	$(r, 1)$	{a}
{ab}	2	1	1	$(r, 1)$	{ab}
{b}	2	2	2	$(r, 2)$	{b}
{abb}	3	1	1	$(r, 1)$	{abb}
{bb}	3	2	1	$(r, 2)$	{bb}
{b}	3	3	3	$(r, 2)$	{b}
{abba}	4	1	1	$(r, 1)$	{abba}
{bba}	4	2	1	$(r, 2)$	{bba}
{ba}	4	3	2	$(u, 3)$	{a}
{a}	4	4	3	$(u, 1)$	{a}

Table 5.1: Applying suffix extension rules to build implicit suffix trees.

So far, the description of this algorithm may seem ineffective in terms of complexity as compared to the naive algorithm presented earlier. Processing all phases and extensions already takes $\sum_{i=1}^{s} \Theta(i) = \Theta(s^2)$ time. Moreover, each suffix extension, if performed in a direct way, may take time proportional to the size of the current tree. In total, the complexity may reach $O(s^3)$ and certainly be $\Omega(s^2)$. What is then the point to use this algorithm? We beg the reader to lay to our account and be patient. After all speed-ups are introduced we will see how the complexity is shrunk down to the desirable linear complexity. Waiting will be worth it.

Exercises

$\boxed{1}$ Build the implicit suffix tree for string $S = \{\texttt{aaba}\}$ and trace its construction as done in the text.

$\boxed{2}$ Give a string S of arbitrary length such that its implicit suffix tree only has two leaves.

5.3.3 Suffix Links

Suffix links are simply pointers between internal nodes of an (implicit) suffix trees. They will allow us to lower the time complexity of phase i to $\Theta(i)$. This, in turn, will lower the time complexity of the whole algorithm to $\Theta(s^2)$. Additional speed-ups will eventually lead to the desired linear-time algorithm. Suffix links save time on traversing the tree from the root to the suffixes of S.

Let A be an arbitrary substring of S, including the possibility $A = \varepsilon$. Let z be a single character of S. Suppose there are two internal nodes v, w, the former having path-label $\{zA\}$ and the latter having path-label $\{A\}$. A pointer from v to w is called a **suffix link**. The case $A = \varepsilon$ will also be taken into account and its suffix link then points to the root. The root itself is not considered as an internal node and, therefore, no suffix link comes out from it. Typically, zA will be a suffix $S[j..i]$ and A the next suffix $S[j+1..i]$.

Example. Let $S = \{\texttt{aabbabaa}\}$ be a string. Figure 5.10 displays $\mathcal{I}(S)$ along with its suffix links (dashed lines). Tree $\mathcal{I}(S)$ has four suffix links coming from its four internal nodes v_1, v_2, v_3 and v_4. In general, a suffix link between two nodes v, w will be denoted by $v \longrightarrow w$. Do not confuse this notation with paths from the root to nodes, denoted similarly; context will make the difference. Path-label of v_4 is $\{\texttt{ba}\}$ and path-label of v_1 is $\{\texttt{a}\}$; therefore, there is a suffix link $v_4 \longrightarrow v_1$. Analogously, v_1 and v_3 have suffix links to the root because their respective path-labels are one-single character. Finally, there is another suffix link from v_2 to v_3.

The following results will prove that in any implicit suffix tree *every* internal node has one suffix link. This fact is not obvious from the definition itself. The first theorem studies the mechanism for creation of new internal nodes.

Theorem 5.3.2 *Assume the algorithm is executing extension j of phase $i+1$. If a new internal node v with path-label $S[j..i+1]$ is created, then one of the two following situations is true:*

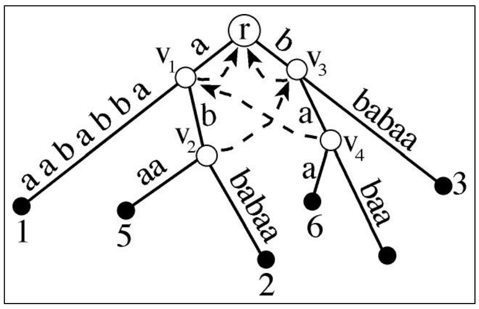


Figure 5.10: Suffix links in the implicit suffix tree of string {aabbabaa}.

1. *Path-label $S[j+1..i+1]$ already exists in the current tree.*
2. *An internal node at the end of string $S[j..i+1]$ will be created in extension $j+1$ of phase $i+1$.*

Proof: New internal nodes are only created when extension Rule 2 is applied. Since the algorithm is in extension j of phase $i+1$, it has to find the end of path $S[j..i]$ and insert v after such path. Let us assume that the end of path $S[j..i]$ is on edge (u_1, u_2). Let us call B the continuation of the path containing $S[j..i]$ on edge (u_1, u_2). Such continuation must exist; otherwise, Rule 2 could not be applied; see Figure 5.11.

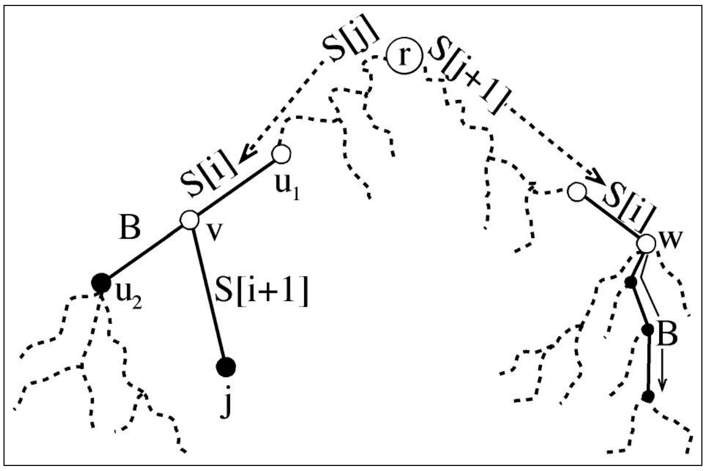


Figure 5.11: Creating a new internal node at extension j of phase i.

By Rule 2 edge (u_1, u_2) is split, internal node v and leaf j are inserted and label $S[i+1]$ is assigned to node (v, j).

Let us now consider extension $j+1$ of phase $i+1$, in which suffix $S[j+1..i+1]$ of prefix $S[1..i+1]$ is added to the current tree. Since $S[j+1..i]$ was already added in extension $j+1$ of phase i, path $S[j+1..i]$ is already found in the tree. Moreover, B continues path $S[j+1..i]$. If B is the only string continuing the path, then a new internal node w will be created and path $S[j+1..i]$ then ends at an internal node. If there is another continuation, it must be through another branch and, therefore, there must be an internal node already. In this case, path $S[j+1..i]$ also ends at an internal node. ▮

Theorem 5.3.3 *Assume the algorithm is executing extension j of phase $i+1$. Then, any newly created node will have a suffix link by the end of extension $j+1$.*

Proof: The proof will be by induction on the number of phases.

Base case: $\mathcal{I}_1$ is just a one-edge tree and has no suffix links.

Inductive case: Assume that at the end of phase i every internal node has a suffix link. If an internal node is created by extension j of phase $i+1$, by Theorem 5.3.2, it will have a suffix link by the end of extension $j+1$. Finally, note that no new internal node is created in extension $i+1$. This extension just consists of inserting character $S[i+1]$ in the tree. This insertion is always managed by Rule 1 or Rule 3. ▮

By combining Theorem 5.3.2 and 5.3.3 we have proved the existence of suffix links for each internal node. The following theorem states this fact formally.

Theorem 5.3.4 *Let $\mathcal{I}_i$ an implicit suffix tree. Given an internal node v with path-label $S[j..i+1]$, there always exists one node w of $\mathcal{I}_i$ with path-label $S[j+1..i+1]$.*

Given that any internal node v has a suffix link, we will call $s(v)$ the node to which the suffix link of v points to.

Exercises

$\boxed{1}$ Identify the suffix links in the implicit suffix tree of string $S = \{\texttt{aabcaba}\}$.

5.3.4 Extension Algorithm

Now, we will show how to use suffix links to reduce the time complexity of performing extensions in Ukkonen's algorithm. Assume the algorithm is just starting phase $i+1$. We describe the whole process, which will be called the **extension algorithm**.

- **Extension 1, phase $i+1$.** The first suffix to be inserted in the tree is $S[1..i+1]$. First, suffix $S[1..i]$ must be located and, subsequently, by the suffix extension rules, $S[i+1]$ is inserted. Since $S[1..i]$ is the longest string represented so far in the tree, it must end at a leaf of $\mathcal{I}_i$. By keeping a pointer f to the leaf containing the current full string, this extension can be handled in constant time. Note that Rule 1 always manages the suffix extension from $S[1..i]$ to $S[1..i+1]$.

- **Extension 2, phase $i+1$.** Here Theorem 5.3.4 comes into play. Consider the edge ending at leaf 1. Call v the internal node (or possibly the root) forming that edge and call α the edge-label of $(v, 1)$. In order to find $S[2..i]$, we walk up from leaf 1 to node v; from there we follow the suffix link of v, which points to node $s(v)$. Once in $s(v)$ we walk down the path labelled α until reaching the end of path $S[2..i]$. Then, we apply the pertinent suffix extension rule and insert $S[i + 1]$ accordingly. See Figure 5.12 for an illustration of this process.

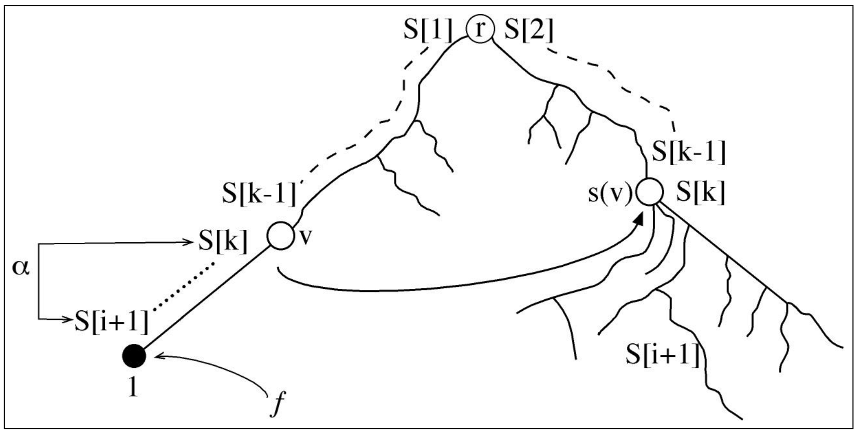


Figure 5.12: Extension algorithm.

Let k be the position such that edge-label α is $S[k..i + 1]$. After traversing suffix link $(v, s(v))$, we are at the beginning of another copy of α in *another* subtree of $\mathcal{I}_i$. Because of previous insertions of suffixes, that subtree may not be simply an edge, but we know that there is one path $S[k..i]$ from $s(v)$ at the end of which character $S[i + 1]$ will be inserted.

- **Extensions** $j > 2$**, phase** $i + 1$. The procedure is the same as in extension 2. The only difference is that in extensions $j > 2$, the end of $S[j..i]$ may be at a node having already a suffix link (that is, it is an internal node with degree greater than 2). In this case, the algorithm should follow that suffix link.

Example. Let us consider the implicit suffix tree of string $S = \{\texttt{aabbaacb}\}$. In extension 5, phase 7 suffix $S[5..7] = \{\texttt{aac}\}$ of prefix $S[1..7] = \{\texttt{aabbaac}\}$ is inserted in the tree. In order to do so, suffix $S[5..6] = \{\texttt{aa}\}$ is searched for from the root and after that suffix character $S[7] = \{\texttt{c}\}$ is then finally inserted. The tree at this point, including its suffix links, is displayed in Figure 5.13.

Character $S[7] = \{\texttt{c}\}$ is inserted by virtue of Rule 2, which gives place to node 3. By using the extension algorithm described above we do not need to start the subsequent searches from the root for the remaining extensions. Indeed, for extension 6, where $S[6..6] = \{\texttt{a}\}$ has to be located, we just walk up one node and find that path. Because character $S[7] = \{\texttt{c}\}$ is not found on the only outgoing edge from node 1, Rule 2 must be applied and a new leaf is added. For extension 7, we just follow the suffix link at node 1 and arrive at the root. Character $S[7] = \{\texttt{c}\}$ is not found among the outgoing edges and thus has to be inserted.

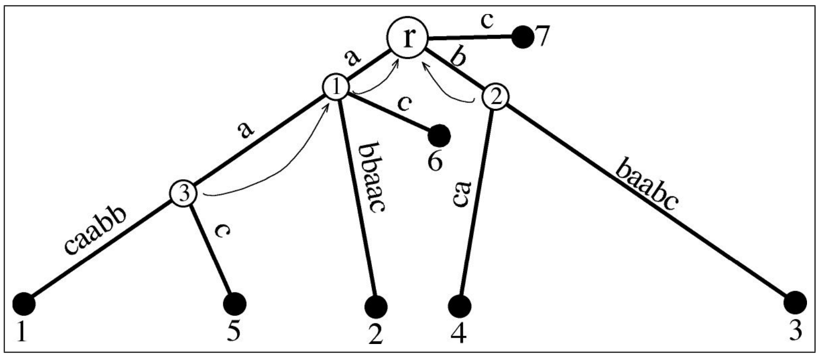


Figure 5.13: Extension 5, phase 7 of Ukkonen's algorithm.

To finish this section, we will introduce one more speed-up for computing extensions, the so-called **skip/count speed-up**. In terms of time bounds, we have not achieved a linear-time bound yet. Even if a proper implementation of suffix links were used, the worst-case time complexity would still be cubic. In each extension a linear number of comparisons in the length of the string is required. The next speed-up will show how to reduce the complexity of walking on the tree. The resulting complexity will be proportional to the number of nodes rather than to the number of characters on the tree edges. The number of nodes is linear (Euler's formula), whereas the number of characters may be quadratic.

Recall that, in general, α will be the edge-label of node (v, j) in extension j of phase $i+1$; see Figure 5.14. Let c be the length of string α. As we know, there is a path starting at the subtree rooted at $s(v)$ spelling out path α itself. Let us call $c_1, \ldots, c_k$ the lengths of the edges of that path. First, we find the correct outgoing edge from $s(v)$. Let c_1 be its length. We compare c and c_1; if $c > c_1$, then we have to carry on. At this point we set a variable h to $c_1 + 1$. This variable will point to the following character to be searched at each node. At the following node, we check again the condition $c > c_1 + c_2$. If satisfied, the end of path α has not been found. We set h to $c_1 + c_2 + 1$ and search for character h of α among all the current outgoing edges. At a generic node we check the condition $c > \sum_{i=1}^{l} c_i$, where l is the number of nodes already visited. If true, we set h to $\sum_{i=1}^{l} c_i + 1$ and jump to the next node. Eventually, c will be less or equal than $\sum_{i=1}^{k} c_i$. When that happens, we can safely state that the end of path α is character $\sum_{i=1}^{l} c_i - c$ of the current edge. Figure 5.14 illustrates this description of the algorithm.

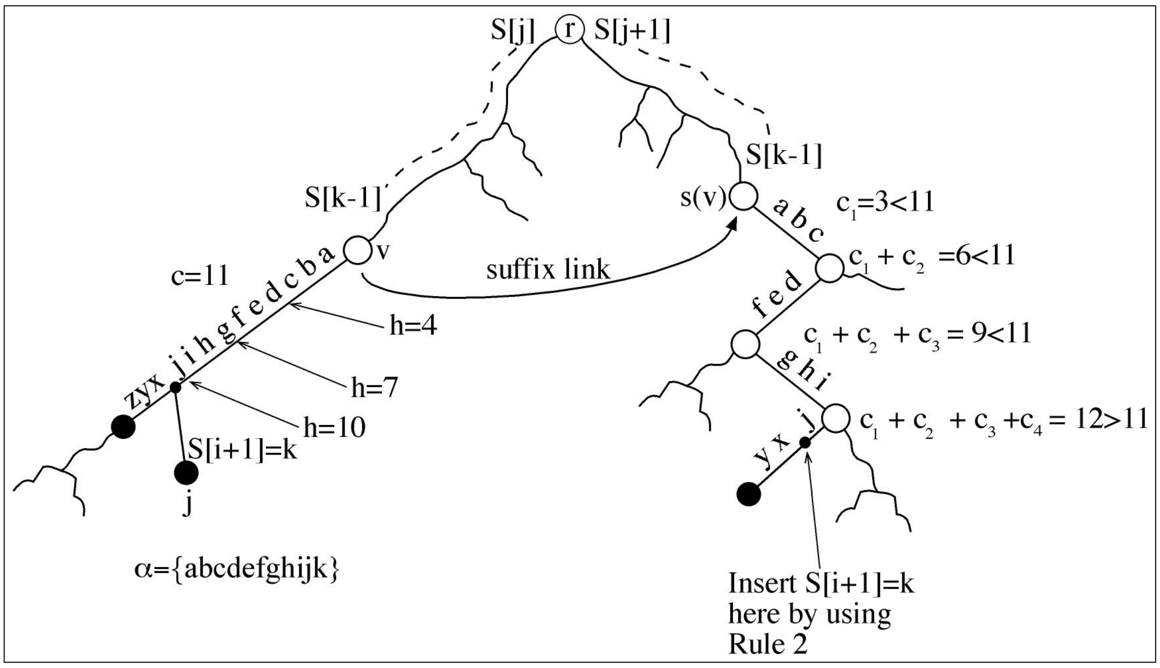


Figure 5.14: The skip/count speed-up.

For this speed-up to work properly two operations have to be implemented so that they take constant time: (1) retrieval of the number of characters on each edge; (2) extraction of any character from S at any given position.

We call the **node-depth** of a node v the number of nodes on the path from the root to v. This definition will be used in the proof of the following theorem.

Theorem 5.3.5 *Let $(v, s(v))$ be a suffix link traversed in some extension j, phase $i + 1$. Then node-depth of v is greater than the node-depth of $s(v)$ at most by one.*

Proof: Consider the paths from the root $S[j..i + 1]$ and $S[j + 1..i + 1]$ and an internal node v in path $S[j..i + 1]$. The key of this proof is to establish that every ancestor of v is linked through suffix links to a unique ancestor of $s(v)$ that is an internal node. By Theorem 5.3.4, this is true for all internal nodes except for the first character of path $S[j..i + 1]$. This first character is linked to the root. By the definition of suffix link, no two internal nodes in path $S[j + 1..i + 1]$ can receive the same suffix link. From this correspondence between internal nodes on both paths, it follows that the difference between the node-depth of v and the node-depth of $s(v)$ cannot be greater than one. ▮

Now we will introduce the definition of **current node-depth**. This definition will be used to prove that the skip/count speed-up actually computes any extension in $O(s)$ time. The current node-depth is the depth of the node currently being processed by the algorithm.

Theorem 5.3.6 *If the skip/count speed-up is implemented, any phase of Ukkonen's algorithm takes $O(s)$ time.*

Proof: In a particular extension the algorithm does the following: (1) starts at an edge; (2) it walks up to a node; (3) it follows its suffix link to another node; (4) and from there it walks down until reaching the point where inserting a character by applying the proper suffix extension rules. Steps (1), (2) and (3) take constant time as seen in the discussion above. As for step (4), we note that the number of nodes in the down-walk is bounded by s and at each node, because of the skip/count speed-up, we do a constant amount of work.

It remains to prove that over the whole phase the complexity is still linear in s. We will prove that claim by analysing how the current node-depth varies as the algorithm is executed. Clearly, the complexity is proportional to number of times the current node-depth is changed. When an up-walk is performed the current node-depth just decreases by one; after following the suffix link the current node-depth is decreased at most by one. In the down-walk step the current node-depth is increased by some amount. Over the whole phases that amount is not greater than the height of the implicit suffix tree, which is $O(s)$. Therefore, the total amount of work of computing the phases in Ukkonen's algorithm is $O(s)$. ▮

As a consequence of the previous theorems we can state the following result, which is actually the first decrease of the cubic complexity.

Theorem 5.3.7 *Ukkonen's algorithm can be implemented with suffix links to run in $O(s^2)$ time.*

Exercises

$\boxed{1}$ Compute the implicit suffix tree for string $S = \{\texttt{ababababaa}\}$ and check the order in which the internal nodes are created. Can it, then, be guaranteed that internal nodes are created in increasing order of depth in each phase?

$\boxed{2}$ Show a string S whose suffix tree construction takes quadratic time, if the skip/count speed-up is not used.

Finally, for arbitrary-length strings, choose an even number s and just set S as

$$\{(\texttt{ab})^{\frac{(s-2)}{2}}\texttt{a}(\texttt{ab})^{\frac{(s-2)}{2}}\texttt{b}\}.$$

5.3.5 Edge-Label Compression

There is an intrinsic difficulty that does not allow us to lower the complexity of computing suffix trees to linear and that is the amount of space required to store edge-labels in the tree. In the worst case, an $\Omega(s^2)$ amount of memory may be required to store the information during the computation of intermediate implicit suffix trees. For example, string $S = \{\texttt{abcdefghijklmnopqrstuvwxyz}\}$ has a tree with a root with 26 outgoing edges. The total sum of the edge-labels is proportional to s^2. Even for finite-sized alphabet strings suffix trees using quadratic memory are relatively easy to find; see the exercises.

An alternate labelling scheme, called **edge-label compression**, is needed to actually shrink down the complexity of the algorithm. The algorithm has a copy of string S at any moment during its execution. We will replace the label (substring) of each edge by a pair of indices pointing to the initial and final positions in string S. By doing so, each edge only keeps a constant amount of memory. Since the total number of edges is $\Theta(s)$, the storage of the edge-labels has been reduced to $O(s)$.

Figure 5.15 shows an implicit suffix tree of string $S\{$`aabbaabb`$\}$ with the ordinary labels and the new labels.

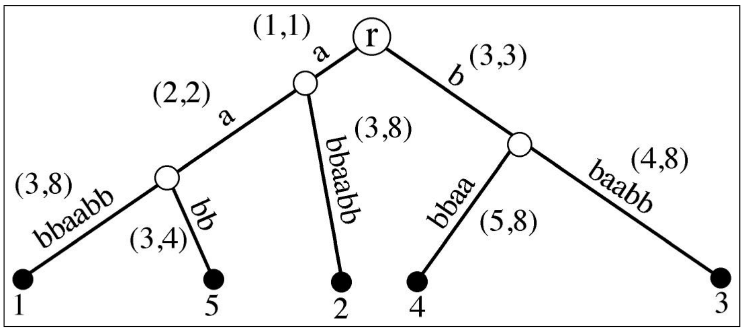


Figure 5.15: Edge-label compression.

It remains to address the question of how the suffix extension rules are applied with those indices.

1. Rule 1. If an edge has label (j, i), adding character $S[i+1]$ is just updated to $(j, i+1)$.
2. Rule 2. Let (u_1, u_2) be the edge where the new internal node will be inserted. Let (j, i) be its label and k be the position after which the new node v is created. Rule 2 then creates three new edges: (u_1, v) with label (j, k), (v, u_2) with label $(k+1, i)$, and node (v, j) with label $(i+1, i+1)$.
3. Rule 3. No change in the edge-label is made.

Therefore, the suffix rule extensions works with the new label scheme.

Exercises

$\boxed{1}$ Give an infinite family of strings such that the total length of the edge-labels is $\Omega(s^2)$, where s is the length of the string. Suppose that the alphabet is finite.

5.3.6 Phase Algorithm

Finally, we will introduce two more speed-ups, namely, the **halt condition** and the **fast leaf update rule**. These two speed-ups are heuristics that deal with the interaction of successive phases. It could seem that in an given phase its extensions are built by applying the suffix extensions rules in an unpredictable manner. However, this is not the case. Application of suffix extension rules actually happen in order, Rule 1 being the first to be applied, then Rule 2 and finally Rule 3. This particular fact, if implemented correctly, will allow the algorithm to run in linear time. This part of the algorithm is called the **phase algorithm**.

The halt condition[1] reads as follows: If Rule 3 is applied in extension j of phase $i+1$, then it will also be applied in the next extensions until the end of the current phase. This observation is easy to prove. Suffix $S[j..i+1]$ is inserted by locating the end of path $S[j..i]$ and then inserting character $S[i+1]$ according to the suffix extension rules. If $S[j..i+1]$ is already in the tree, that means that there are at least two identical substrings in S occurring at different positions. If substring $S[j..i+1]$ is already in the tree, so will strings $S[j+1..i+1], \ldots, S[i+1..i+1]$.

Example. Let $S = \{\texttt{aabacaabac}\}$ be a string. Assume that the algorithm is in phase 9 and extension 6. The prefix $S[1..9] = \{\texttt{aabacaaba}\}$ is being processed; suffixes $S[1..9] = \{\texttt{aabacaaba}\}$ to $S[5..9] = \{\texttt{aaba}\}$ have already inserted in the tree. Now, the algorithm is going to insert suffix $\{\texttt{aaba}\}$. This suffix already appeared at position 1; hence, Rule 3 correctly detects the situation when inserting character $S[9] = \{\texttt{a}\}$ (in boldface) after path $S[6..8] = \{\texttt{aba}\}$; see Figure 5.16. By the halt condition, all the remaining suffixes are already in the tree and the current phase is terminated.

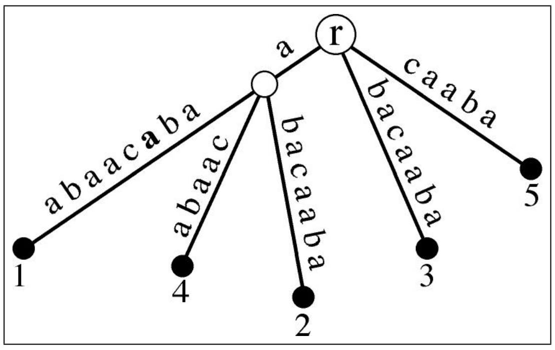


Figure 5.16: The halt condition.

Therefore, once Rule 3 is applied, in order to save time, the algorithm should jump to the next phase. In this case, those phases are said to be built **implicitly**. We call **explicit extensions** are those actually built by the algorithm (like the ones managed by Rule 2).

The second speed-up, the fast leaf update rule[2], takes advantage of the fact that, once leaf j is created, it will remain a leaf until the end of the execution of the algorithm. Note that the suffix extension rules do not schedule any mechanism to modify leaf labels. Hence, once a leaf is numbered j in phase $i+1$, Rule 1 will always apply to extension j of the subsequent phases (from phase $i+1$ to phase s).

[1]In [Gus97] this condition is called with a more striking name, a show stopper.

[2]Gusfield [Gus97] this rule receives the rather long name of "once in a leaf, always in a leaf."

A clever application of this speed-up leads to an essential reduction of the running time. Let us trace Ukkonen's algorithm and find out how to obtain such reduction. The algorithm starts with phase 1. This phase just consists of inserting a character $S[1]$ in an empty tree. We just apply Rule 2 and create the root and a leaf numbered 1 with edge-label $\{S[1]\}$. In the next few phases, more suffixes are added to the tree. Note that the first few insertions are always performed by applying Rule 1. Moreover, after a few insertions through Rule 1, Rule 2 is necessarily applied and, finally, either the phase is done or Rule 3 is applied. Why is Rule 1 first applied when a new phase starts? Assume we are about to start phase $i+1$; further assume that in phase i the implicit suffix tree $\mathcal{I}(S)$ has already k leaves. Consider all the paths from the root to each of those k leaves. These suffixes can only be updated by Rule 1.

Suppose that after extension l of phase $i+1$, we have to apply Rule 2. By the extension algorithm, Rule 2 is applied until either the root is reached or Rule 3 is applied. Rule 1 cannot be applied in between. Otherwise, character $S[l+1]$ would be a leaf edge and this would have been updated in an extension prior to extension l. Therefore, when running the algorithm, there are l implicit suffix extensions through Rule 1, followed by some explicit extensions through Rule 2, say until extension r, and finally Rule 3 is applied from extension $r+1$ to extension $i+1$. Since the number of leaves between two consecutive phases can only be constant or increase, value l is non-decreasing. As stated before, l is equal to the number of current leaves in the tree. Value r is equal to l plus the number of current internal nodes.

In order to take advantage of this situation the algorithm must keep a variable r that stores the last time Rule 2 was applied in phase i. Phase $i+1$ can start in extension r since the $r-1$ previous extensions are all implicit.

Example. Consider again string $S = \{\texttt{aabacaabacx}\}$. In Table 5.2 the first few values r for the phase algorithm are represented.

Phase	Suffix	Value r	Rule 3
Phase 1	{a}	$r = 1$	No
Phase 2	{aa}	$r = 1$	Yes
Phase 3	{aab}	$r = 3$	No
Phase 4	{aaba}	$r = 3$	Yes
Phase 5	{aabac}	$r = 5$	No
Phase 6	{aabaca}	$r = 5$	Yes

Table 5.2: Tracing the values of variable r in the phase algorithm.

Note that from this point on the number of implicit extensions will be at least 5. Figure 5.17 shows the implicit suffix tree after phase 5 is completed.

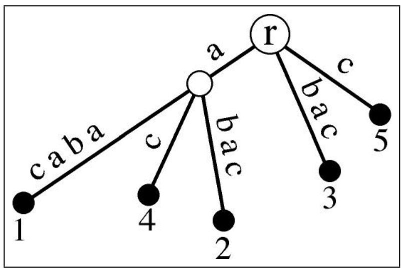


Figure 5.17: The fast leaf update rule.

It only remains to detail how the implicit extensions of Rule 1 will be carried out. If the algorithm actually access to each leaf edge to do so, it will take linear time per phase. Instead of it, we will replace each leaf edge $(p, i+1)$ by (p, e), where e is a variable containing the current phase. By updating that variable at the beginning of each phase all implicit extensions are performed in constant time.

Example. In the previous example, phase 6 starts by implicitly building extensions 1 to 5. If the variable e is kept throughout the algorithm, those extensions are built by setting $e = 6$. That would imply adding character $S = \{\texttt{c}\}$ at the end of the all leaf edge-labels. This updating takes constant time.

Exercises

$\boxed{1}$ Show a string S whose suffix tree construction takes quadratic time, if the fast leaf update rule speed-up is not used.

$\boxed{2}$ Assume the algorithm is starting to process phase $i + 1$.

(a) Describe a string such that no extension will apply Rule 2 after the extensions managed by Rule 1.

(b) Describe a string such that all extensions will apply Rule 2 after the extensions managed by Rule 1.

5.3.7 Generating $\mathcal{T}(S)$ from $\mathcal{I}(S)$

To fully finish the description of Ukkonen's algorithms we need to detail how to obtain the *original* suffix tree $\mathcal{T}(S)$ from the *implicit* suffix tree. Add the termination character \$ to S and run Ukkonen's algorithm for the last time. Since the termination character is not in S, no suffix is now a prefix of any other suffix. Therefore, each suffix will end in a leaf. In other words, the degree-1 nodes that were deleted when creating the implicit suffix trees are put back in the tree. It is also necessary to replace variable e by its true value s.

5.4 Chapter Notes

We have seen how to construct suffix trees by using Ukkonen's algorithm. Suffix trees belong to a general class of data structures specialized in storing a set of strings so that string operations are performed quickly. Other data structures related to suffix trees are tries, keywords trees, radix trees or Patricias [Sed90], [GBY91], [Wik09e]. Morrison [Mor68] was the first to introduce Patricia tries as an index for searching in marked-up text.

The goodness of suffix trees resides in the way it exposes the internal structure of a string. Such representation of the string structure allows solving many problems in string pattern recognition. In the next chapter, we will see how to easily solve the exact string matching problem (SMC), the substring problem for a database of patterns, the longest common substring of two strings, computation of palindromes and the DNA contamination problem. There are many other problems that can be solved via suffix trees; see [Gus97] and [Apo85] and the references therein for more information.

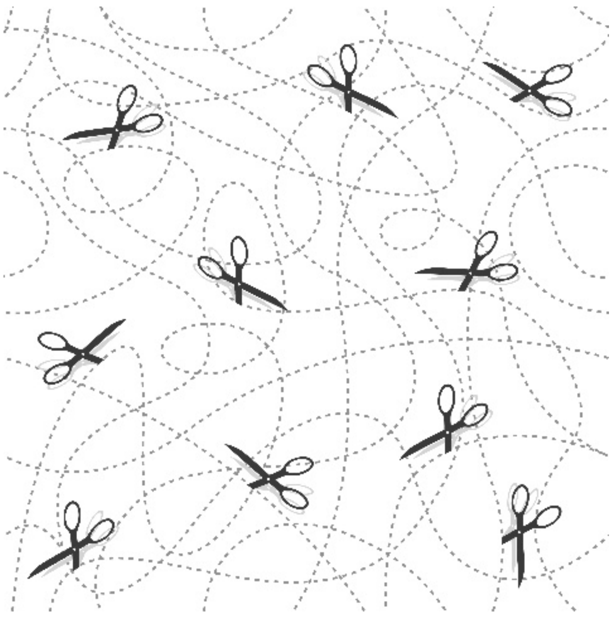
Compared to other algorithms, suffix trees are suitable when the text is fixed and known, and the pattern varies (query mode for the pattern). This is a very common and important situation appearing mainly in Bioinformatics. Think of the text as a large biological database in which many different DNA sequences are searched in. To mention another field, in Music Technology, the text is usually a big database of musical excerpts grouped by composer or gender. A piece of music is searched in the text under several similarity criteria with the aim of attributing authorship or classifying the piece.

When the text is not known or both the pattern and the text are presented at the same time algorithms like the Karp-Morris-Pratt or Boyer-Moore algorithms are better than suffix trees.

As discussed in the outset, suffix trees are not widely taught, in spite of its relevance. There is a big difference between Ukkonen's original paper [Ukk95], who described his algorithm in very abstract terms, and Gusfield's book, who wrote an accessible account of Ukkonen's algorithm. Nevertheless, Gusfield's account has not made Ukkonen's algorithm to enter in the main algorithms and data structures texts. The following list of references tries to facilitate the access to resources where nice descriptions of suffix trees and its computation can be found. Allison [All09] built a web page where he explains how to construct suffix using Ukkonen's terminology, but in a more approachable way. His web page has an application to demonstrate how suffix trees are built. The web page [Wik09f] contains a very complete list of the applications of suffix trees. Lewis [Lew09] described several applications of suffix trees to Computational Biology; his paper is quite self-contained and points to many other interesting sites. Giegerich and Kurtz reviewed in [GK97] the linear time suffix tree constructions by Weiner, McCreight, and Ukkonen. This is a very recommendable paper to gain insight in the suffix tree construction from different approaches. Tata et al. [THP05] address the problem of constructing suffix trees when the text is a very large database. Here, due to the scope of these notes, we have not tackled with the many subtleties involved in the actual programming of suffix trees. The probabilistic analysis of the construction of suffix trees is dealt with in the papers [Szp93b] and [Szp93a] by Wojciech Szpankowski. Grossi and Italiano [GI93] wrote a survey on suffix trees and its applications.

Chapter 6

Suffix Trees and its Applications



As mentioned earlier, Apostolico[Apo85] pointed out the "myriad of virtues" possessed by suffix trees. There are too many to go into detail on each. We will cover the most fundamental and useful ones. We refer the reader to [Gus97] and the references therein for an exhaustive list of the applications of suffix trees.

6.1 Exact String Matching

The most immediate application is to solve the substring problem stated at the beginning of the previous chapter. Recall that the substring problem consists of pre-processing the text such that a query of a given pattern is performed in time proportional only to the length of P. The following demonstration shows how to use suffix trees to solve this problem.

Assume we have at hand the suffix tree corresponding to text T. By performing breadth-first search (or other tree-search techniques), we find a path in the suffix tree that matches pattern P. If no path is found, we may safely state that P is not in T. Otherwise, assume there is such a path. Note that, by the properties of suffix trees, that path is unique. Consider the edge at which that path ends, and for that edge consider its farthest node from the root (recall that a tree is a directed graph). That node is the root of a subtree with, say, k leaves. Given that every path from the root to the leaves of that subtree spells out a suffix of T having P as a substring, the number of occurrences of P in T is exactly k. Since the leaves of the suffix tree are numbered with the starting positions of the suffixes, the exact positions where P occurs can also be recovered from the suffix tree.

Example. Consider pattern $P = \{\texttt{aba}\}$ and text $P = \{\texttt{cabacabadadabac}\}$. Suffix tree $\mathcal{T}(T)$ is shown in Figure 6.1 (the terminal symbol \$ has been added).

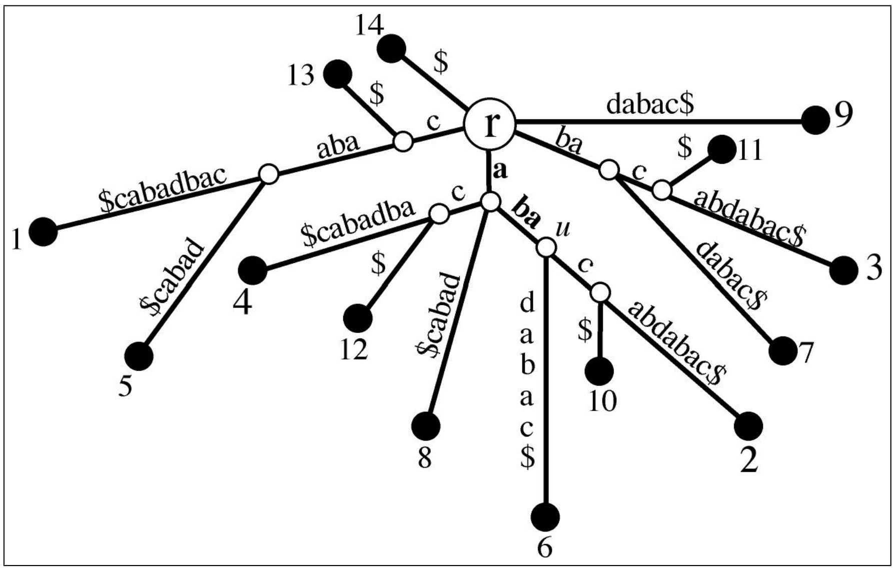


Figure 6.1: Solving the computation string matching problem with suffix trees.

The path that matches pattern $P = \{\texttt{aba}\}$ is shown in boldface in the tree; it ends at node u. The subtree rooted at node u has three leaves numbered $2, 6$ and 10. These three leaves correspond to the three occurrences of string $\{\texttt{aba}\}$ in T. Moreover, the labels of those leaves provide the positions of the pattern P in T.

Finally, we observe that the substring problem can be solved in time $O(m)$ with preprocessing time $O(n)$. In the query mode and for large databases, this represents a great saving of time.

6.2 Longest Common Substring of Two Strings

One of the first problems to arise in string pattern recognition, both because of its theoretical and practical interest, was the longest common substring problem of two strings. The problem is stated as follows.

The longest common substring of two strings (LCS problem): Given two strings S_1, S_2, find the longest common substring occurring both in S_1 and S_2.

The longest common substring of two strings S_1, S_2 will be denoted by $LCS(S_1, S_2)$. Let $S_1 = \{\texttt{contumaciously}\}$, $S_1 = \{\texttt{contumeliously}\}$ be two strings. Their $LCS(S_1, S_2)$ is the set formed by strings $\{\texttt{contum}\}$ and $\{\texttt{iously}\}$, both with 6 characters each. When there is no common substring, $LCS(S_1, S_2)$ is $\{\emptyset\}$.

A naive algorithm could easily take quadratic time or higher. Again, this problem can be solved in linear time by means of suffix trees. Let us first introduce the concept of **generalized suffix tree** of a set of strings. Let $\{S_1, \ldots, S_k\}$ a set of k strings and let $\{\$_1, \ldots, \$_{k-1}\}$ a set of terminal symbols such that $\$_i \notin S_j$, for any i, j. Consider the string $S = S_1\$_1S_2\$_2 \ldots S_{k-1}\$_{k-1}S_k$. The suffix tree of $\mathcal{T}(S)$ is called the generalized suffix tree of the set $S_1, \ldots, S_k$.

Example. Consider three strings $S_1 = \{\texttt{abac}\}, S_2 = \{\texttt{bada}\}$ and $S_3 = \{\texttt{acd}\}$. The set of terminal symbols will be $\{\$, \#\}$; symbol % is the terminal symbol needed for constructing the suffix tree. The resulting suffix tree $\mathcal{T}(S_1\$_1S_2\#S_3) = \mathcal{T}(\texttt{abac\$bada\#acd})$ is shown in Figure 6.2.

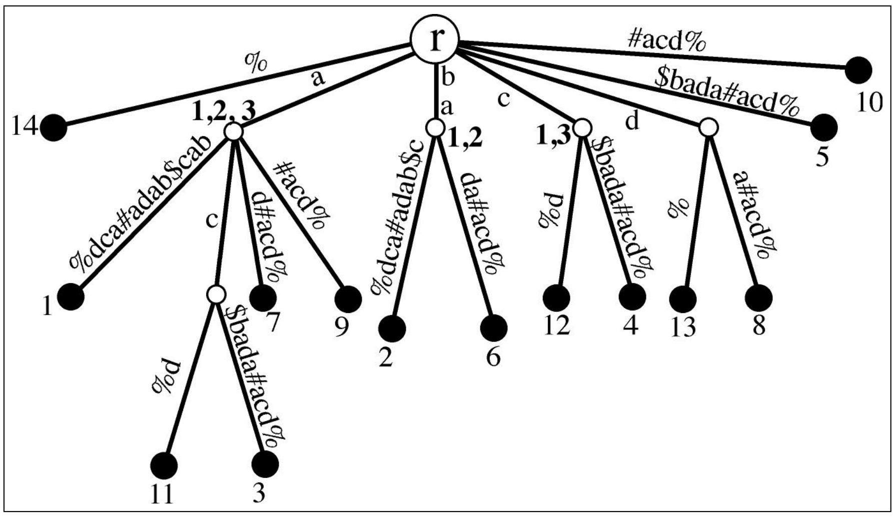


Figure 6.2: Generalized suffix tree of a set of strings.

In order to solve the LCS problem, we just build the generalized tree of strings S_1 and S_2, $\mathcal{T}(S_1\$S_2)$. When constructing it, mark each internal node with a label 1 or 2, depending on what string the current suffix is from. For example, in Figure 6.2 there is an internal nodes marked with three labels, $1, 2, 3$, showing a common substring to S_1, S_2 and S_3, namely, string $\{\texttt{a}\}$. The leaves of the subtree rooted at that internal node posses numbers from the three strings S_1, S_2 and S_3. Other internal nodes are just marked with two numbers such as the corresponding to string $\{\texttt{ba}\}$ or string $\{\texttt{c}\}$.

Therefore, a path-label composed of internal nodes marked with both numbers will spell out a common substring to S_1 and S_2. Finding $LCS(S_1, S_2)$ is achieved by just outputting the deepest string in the tree whose edges are both marked by 1 and 2. Marking the internal nodes and finding the deepest string can all be performed in time proportional to $O(|S_1| + |S_2|)$.

6.3 The Substring Problem for a Database of Patterns

This problem is a generalization of the substring problem. The text is now formed by a set of strings, normally a large one, called the **database of patterns**. The text is known and fixed. In the context of Bioinformatics, the database of patterns is a big biological database in which later on many DNA sequences are searched for. Several strings are presenting (query mode) and the algorithm has to determine what strings in the database contain the query string as a substring. Of course, pre-processing of the database is allowed and once this is done, the queries has to be answered in time proportional to their size. This problem is formally stated as follows.

The substring problem for a database of patterns (SPDP): Given a set of texts $T_1, \ldots, T_k$, the database of patterns, pre-process them so that the string matching computation problem is solved in time proportional to the length of pattern P.

Again, the solution comes to hand through generalized suffix trees. All the patterns of the text, $T_1, \ldots, T_k$, are concatenated with terminal symbols in between. A big text $T = T_1\$_1T_2 \ldots T_{k-1}\$_{k-1}T_k$ is built, where $\$_i \notin T_j$, for any i, j. Constructing $\mathcal{T}(T)$ takes linear time on the size of T. By following a similar argument to that of the substring problem, we can identify the occurrences of P in $\{T_1, \ldots, T_k\}$. Indeed, label the internal nodes with the corresponding text-labels and after that find those nodes marked by all the texts. They will return the solution to this problem; if no such node is found, then pattern P is not in all of the patterns of the database. The time complexity is again $O(|P|)$, once $\mathcal{T}(T)$ is built.

6.4 DNA Contamination

This problem has its origins in Bioinformatics as the title suggests. When DNA is sequenced, or more in general when DNA is processed, it is very undesirable that DNA from other sources become inserted in the sample. This situation can lead to invalid conclusions about the DNA sequences being studied. There have been very embarrassing cases. Some time ago, a few scientists announced they have sequenced dinosaur's DNA from a bone, but other scientists harbour suspicions because of DNA contamination, as was the case finally. Normally, DNA sequences of many contaminants are known.

As a computation problem, the DNA problem is stated as follows.

The DNA contamination problem (DNAC problem): Given a string S, the newly sequenced string, and a set of source of possible contaminants, $C_1, \ldots, C_k$, find all substrings of S that occur in the set of contaminants having a length greater than certain number l.

Once more suffix trees will allow us to solve the problem in linear time. First, build a string C by concatenating all the possible contaminants. As before, insert terminal symbols between consecutive contaminants; the terminal symbols do not occur in any of the strings $C_1, \ldots, C_k$. Next, compute a generalized suffix tree for S and C. Then mark internal nodes with labels showing where the suffixes come from (either S or C). Finally, by traversing the tree, report all the marked nodes with both labels that are at a depth greater than l. If no node of that kind is found, we may state that sample S is not contaminated. The complexity of the whole process is proportional to the size of C and S.

Exercises

$\boxed{1}$ Given a string S, design an algorithm to find the substring with the longest prefix that is also a suffix of that substring. Prove the correctness of your algorithm and compute its complexity.

$\boxed{2}$ Given a string S, design an algorithm to find the longest repeated substring in S. Prove the correctness of your algorithm and compute its complexity.

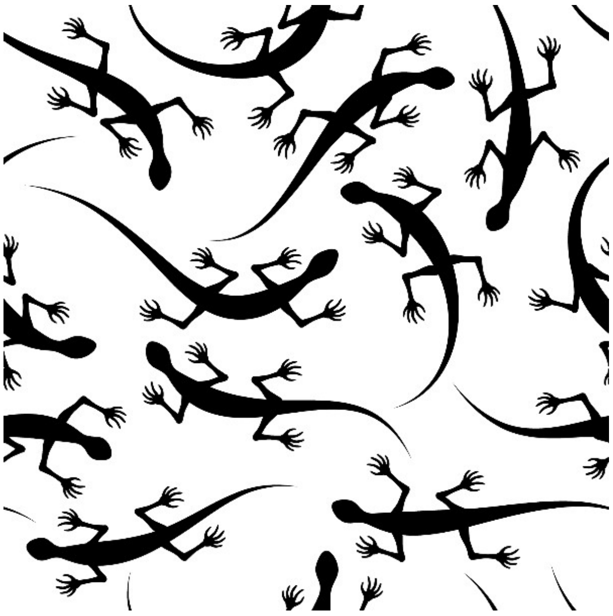
$\boxed{3}$ **Palindromes**. A palindrome is a string such that when read forwards and backwards the same string is obtained. For example, string {`abcdcba`} is a palindrome. Design an algorithm to find the longest palindrome in a string. Prove the correctness of your algorithm and compute its complexity.

6.5 Chapter Notes

We have shown some of the basic applications of suffix trees. However, there are many more than we list here due to lack of space and also the scope of these notes. Other applications of suffix trees include: the common substrings of more than two strings, a problem that appears very often in Bioinformatics; computation of the longest substring common to at least a given number of strings; computation of the matching statistics; computation of all-pairs suffix- prefix matching; computation of repetitive structures in DNA. The list could go on. The reader is referred to [Gus97] and the references therein for more and deeper information on those problems.

Chapter 7

English Phonetics



7.1 Vowels

A vowel is a sound produced by making the vocal chords vibrate and letting the air pass through the oral cavity without obstruction. Vowels are always part of syllables. The position of the tongue determines the production of the different vowels. **The position of the tongue** is governed by two factors: first, vocal height, its position with respect to the roof of the mouth; second, **the vocal backness** its position with respect of the mouth as measured from front to back . The height of the tongue may range from open, as in [a], to close, as in [i]. In general, there are three distinctive positions, namely, open, mid and close positions, and also other in-between, more subtle positions, near-open, open-mid, close-mid and near-close positions. Notice that vocal height is somewhat associated to jaw opening.

The tongue is a very flexible and quick muscle that may be positioned at several places in the mouth. Its position within the mouth defines the vocal backness. The tongue may be at the back, almost touching the **soft palate** or **velum**, as in the sounds [ʌ] and [u]; it may be at a central position, as in [ə]; or it may be at the front, almost reaching the teeth, as in [i].

The chart below represents all the English vowels as a function of vocal height and vocal backness. A third feature that affects the quality of a vowel is whether the lips are rounded or not. This is called the **roundness** of the vowel. For example, [u] is pronounced by rounding the lips, but [a] is not (it is an unrounded vowel). In places where vowels are paired, the right represents a rounded vowel (in which the lips are rounded) while the left is its unrounded counterpart.

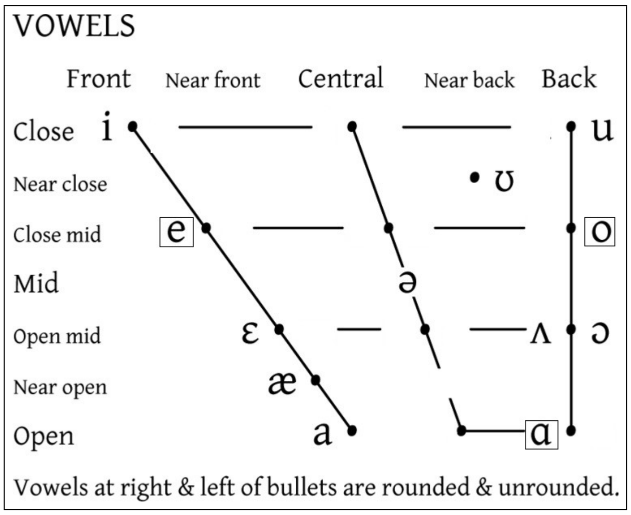


Figure 7.1: English vowels sound.

The vowels within the squares in Figure 7.1 are dialectal variants. For example, the sound [o] is more closed than [ɔ]. The sound [o] is the Spanish o, too.

Besides dialectal variants, there are roughly 12 vowel sounds in English. We strongly recommend the reader to check out the web page *Phonetics: The sounds of American English* [oS01], built by the Department of Spanish of the University of Iowa, where clear and graphical description and Flash animation of the production of English vowels is given. Next, we will describe the vowels sounds and give several examples.

The English vowels:

1. Open front unrounded vowel [a], normally pronounced as a long vowel. For example: heart[ha:t] or palm[pa:m]. In this vowel the tongue is at the bottom of the mouth, as far as possible from the roof of the mouth. Also, the tongue is positioned as far forward as possible in the mouth, just below the teeth. The jaw is open; in fact, it is more open than the Spanish [a].

2. Near-open front unrounded vowel [æ], a short vowel as in man[mæn], hat[hæt], very frequently pronounced in monosyllabic words with a (as in bat, cat, fat, gap, hat, rat,

sat, etc.). This vowel is similar to the previous one, except that the tongue is slightly raised.

3. Open-mid front unrounded vowel [e], a short vowel more closed than the Spanish [e]. The tongue is still more raised. For example: bed[bed], red[red].
4. Short close front unrounded vowel [i], a short sound as in it[it], hit[hit]. Now, the tongue is raised as much as possible.
5. Long close front unrounded vowel [i:], a long i, as in eat[i:t], heat[hi:t].
6. Mid central vowel [ə], a short sound as in rare[reə], so[səʊ]. The tongue is placed halfway between the front and back positions.
7. Open-mid central unrounded vowel [ɜ:], a long sound as in fur[fɜ:], bird[bɜ:d]. It is pronounced as in the previous vowel, but slightly more open.
8. Open-mid back unrounded vowel [ʌ], a short sound as in hut[hʌt], but[bʌt]. The tongue is positioned as far back as possible in the mouth, almost touching the soft palate. This sound does not exist in Spanish.
9. Short open-mid back rounded vowel [ɔ], a short sound as in hot[hɔt], dog[dɔg]. This is a rounded vowel. The only difference between this vowel and [ʌ] is roundedness.
10. Long open-mid back rounded vowel [ɔ:], a long sound as in thought[θɔ:t], caught[k ɔ:t]. This is long version of [ɔ].
11. Near-close near-back vowel [ʊ], a short sound as in put[pʊt], book[bʊk].
12. Close back rounded vowel [u:], a short sound as in moon[mu:n], use[ju:z]. Note that when the long version of [ʊ] becomes a close vowel.

Remarks:

- The length of a vowel is a quality that distinguishes between words, as the following examples show:
  - It[it]$\Longrightarrow$ eat[i:t].
  - Shit[ʃit]$\Longrightarrow$ sheet[ʃi:t].
  - Hill[hil]$\Longrightarrow$ heel[hi:l].
- Vocal backness and vocal height also discriminate between sounds.
  - Heart[ha:t], hut[hʌt] and hat[hæt] are three different words.
  - Vocal backness only also discriminates sounds: were[wɜ:(r)]$\Longrightarrow$war[wɔ:(r)].

- Vocal height by itself distinguishes sounds, too: man[mæn]⟹men[men].

- Roundness also discriminate between sounds.
  - Gut[gʌt]⟹ got[gɔt].
  - Hut[hʌt]⟹ hot[hɔt].

7.2 Consonants

A consonant is a sound produced by a partial or complete obstruction of the air stream. That obstruction may be accomplished by several means. Here there is a classification of consonants.

- **By manner of articulation**, that is, the method to articulate the consonant: nasal (the air gets out through the mouth and the nose), stop (complete obstruction of air; also called plosive consonants), or approximant (vowel-like consonants).
- **By place of articulation**, that is, where in the vocal tract the obstruction of the consonant takes place, and which speech organs are involved. Places include, among others, bilabial (both lips), alveolar (tongue against the gum ridge), and velar (tongue against soft palate).
- **By voicing**, that is, by taking into account whether the vocal cords vibrate during the articulation. When the vocal cords vibrate fully, the consonant is called voiced; when they do not vibrate at all, it is voiceless.

We will not go into further details about consonant production, but we refer the reader to the comprehensible and demystifying book of Roach [Roa09]. The following table shows the consonant phonemes found in English. When consonants are shown in pairs, voiceless consonants appear on the left and voiced consonants on the right.

Consonants	**Bilabial**	**Labio-Dental**	**Dental**	**Alveolar**
Nasal	m			n
Plosive	p, b		t ,d	
Affricate				
Fricative		f, v	θ, ð	s, z
Approximant				r
Lateral				l

Consonants	**Post-Alveolar**	**Palatal**	**Velar**	**Glottal**
Nasal			ɳ	
Plosive			k, g	
Affricate	ʧ, ʤ			
Fricative	ʃ, ʒ			h
Approximant		j	w	

Table 7.1: English consonants

The distinction between voiceless and voiced consonants is crucial in English. For example, such distinction regulates the pronunciation of the endings in the Past Simple; see the section below. The following table shows the English consonants paired by voice (voiceless are on the left and voiced on the right).

Voiceless	**Voiced**
[p] pet	[b] bet
[t] tin	[d] din
[k] cut	[g] gut
[ʧ] cheap	[ʤ] jeep
[ʃ] she	[ʒ] measure
[θ] thin	[ð] then
[f] fat	[v] vat
[s] sap	[z] zap

Table 7.2: Examples of English consonants

Sounds [m], [n], [ɳ], [l], [j], [w] and [r] are all also voiced. Sound [h] is voiceless.

Remarks:

- The voicing of a consonant is more important in English than in other languages.
  - Often, it determines the grammatical function: advice[əd'vais] is a noun and its corresponding verb is advise[əd'vaiz].

  – It may distinguish between words: rice[rais]⟹rise[raiz].

- The pronunciation of Past Simple is based on the voicing of the final sounds of verbs, as we will see below.
- The pronunciation of plurals, possessives and third person of Present Simple are also formed according to voicing.

7.3 Semi-consonants

Semi-consonants are not considered as vowels because the passage through which the air passes is very narrow. The semi-consonants are [j], as in yes[jes], and [w], as in wood[wu:d]. They are always followed by a vowel.

- The semi-consonant [j]. It is pronounced as an [i], but by bringing the tongue close to the palate very much. This sound is, therefore, approximant and palatal. It a-ppears in words like union[ju:niən], you[ju:], student[stju:dənt] (British English, but student[stu:dənt] in American English). In Spanish there exists this sound, as in *ayer*[a'jer] or *hierba*['jerba].
- The semi-consonant [w]. This semi-consonant is a voiced labiovelar approximant sound. That means it is articulated with the back part of the tongue raised toward the soft palate and the lips rounded. It can be found in words like were[wɜ:(r)], water[wɔtɜ:(r)].

7.4 Diphthongs

Diphthongs are sounds produced by pronouncing one vowel and then gradually changing to another, normally through a glide between them. The English diphthongs are the following.

- Diphthong [ai], as in rise[raiz], height[hait].
- Diphthong [ei], as in frame[freim], crane[crein].
- Diphthong [əʊ], as in low[ləʊ], so[səʊ]
- Diphthong [aʊ], as in now[naʊ], cow[kaʊ].
- Diphthong [eə], as in rare[reə], fare[feə].
- Diphthong [iə], as in rear[riə], gear[giə].
- Diphthong [ɔi], as in boy[bɔi], coy [kɔi].
- Diphthong [uə], as in sure[ʃuə], lure[luə].

7.5 Pronunciation of plurals

The pronunciation of plurals depends on the final sound of the noun, in particular, whether such sound is a sibilant consonant. Sibilant consonants produce the sound by forming a very narrow passage and letting the air pass through it. The sibilant consonants are showed in the following table.

Sibilant Consonants	Alveolar	Post-Alveolar
Affricate		ʧ, ʤ
Fricative	s, z	ʃ, ʒ

Table 7.3: Sibilant consonants

Note that the sounds [f, v] are not sibilant because the air is stopped when passing through the teeth. Also, the affricate post-alveolar consonants [ʧ, ʤ] are sibilants since after pronouncing the [t, d], the narrow passage is formed.

Rules for the pronunciation of the plural of nouns:

1. If the word ends by a vowel sound, then the plural is pronounced by adding the sound [z]. For example:
   - Day[day]⟹days[daiz].
   - Boy[bɔi]⟹boys[bɔiz].
2. If the word ends by a non-sibilant sound, then the plural is pronounced by adding the sound [s] when it is a voiceless consonant and the sound [z] when it is a voiced consonant. For example:
   - Pet[pet]⟹pets[pets] (voiceless case).
   - Dog[dɔg]⟹dogs[dɔgz] (voiced case).
3. If the word ends by a sibilant sound, then the plural is pronounced by creating a new syllable [-iz]. For example:
   - Beach[bi:ʧ]⟹beaches['bi:ʧiz].
   - Bridge[briʤ]⟹bridges['briʤiz].
   - Bush[buʃ]⟹bushes['buʃiz].
   - Garage[gæ'ra:ʒ]⟹garages[gæ'ra:ʒiz].
   - Bus[bʌs]⟹buses['bʌsiz].
   - Rose[rəʊz]⟹rose['rəʊziz].

Note from the example that the stress of word in the plural does not change.

There are a few exceptions to the previous rule: house[jaʊs]⟹houses['jaʊziz], mouth[maʊθ]⟹mouth[maʊðz].

Moreover, the same pronunciation rules are applied to possessives (Saxon genitive):

- Mark[ma:k]⟹Mark's[ma:ks].
- Joe[ʤəʊ]⟹Joe's[ʤəʊz].
- George[ʤɔ:ʤ]⟹George's[ʤɔ:ʤiz].

7.6 Pronunciation of the Third Person of Singular of Present Simple

When writing the third person of singular of Present Simple takes either an -s or an -es. The pronunciation of the new added consonant follows the same rules as in the plural.

- I want[ai wɔ:nt]⟹He wants[hi wɔ:nts].
- I read[ai ri:d]⟹He reads[hi ri:dz].
- I bet[ai bet]⟹He bets[hi bets].
- I teach[ai ti:ʧ]⟹He teaches[hi 'ti:ʧiz].

7.7 Pronunciation of Past Simple

The pronunciation of Past Simple is based on the voicing (voiceless versus voiced) of the final sound of the verb. We recall the reader the voiceless consonants: [p, t, k, f, ʧ, θ, s, ʃ, h]; and the voiced consonants: [m, n, ɳ, b, d, g, v, ð, z, ʤ, ʒ, r, l, j, w].

Rules for the pronunciation of Past Simple (regular verbs):

1. If the word ends by a voiced sound different from [d], then the Past Simple is pronounced by adding the sound [d]. For example:
   - I explain[ai eksplein]⟹I explained[ai ekspleind].
   - I file[ai fail]⟹I filed[ai faild].
2. If the word ends by a voiceless sound different from [t], then the Past Simple is pronounced by adding the sound [t]. For example:

- I kiss[ai kis]⟹I kissed[ai kist].
- I risk[ai risk]⟹I risked[ai riskt].

3. If the word ends by [d] or [t], then Past Simple is pronounced by adding [id]. For example:

   - I need[ai ni:d]⟹I needed[ai 'ni:did].
   - I want[ai wɔ:nt]⟹I wanted[ai 'wɔ:ntid].

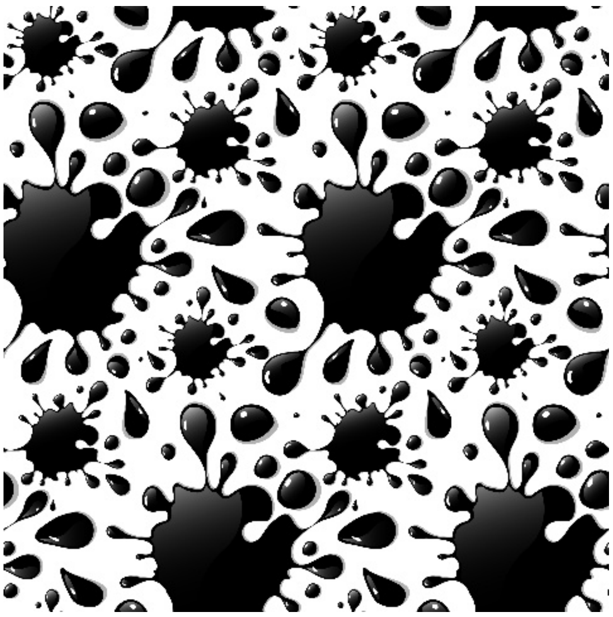
7.8 Chapter Notes

Although largely disregarded in language teaching, phonetics is the only way to learn how to pronounce properly, except if the language is learnt at an early age. Spanish has only five vowels, whereas English, as seen above, has twelve vowels. How will a native Spanish speaker map his vowels onto the English ones? Many methods are purely based on repetition. However, that does not work in general and results in a deficient pronunciation, sometimes even on the verge of intelligibility, and often kept for many years. By learning phonetics, first of all, learners will be able to recognize what sound they are hearing and, therefore, they gain confidence in their understanding of English. Also, they will acquire accuracy in their pronunciation as the rules for producing the sound will be now clear.

Fortunately, there are many resources both online and in form of books to learn English phonetics. We already recommended the excellent web page of the University of Iowa [oS01]. The encyclopedia Wikipedia has two well-written and enlightening articles on phonetics, one about the IPA [Wik09d] (the International Phonetics Alphabet), and the other about English phonology [Wik09c]. As an online dictionary including phonetics transcriptions we recommend Oxford Advanced Learner's Dictionary [Pre09]. As a comprehensive treatise on phonetics we propose the book of Roach [Roa09]. Lastly, for a work integrating phonetics and grammar in a straightforward way we advise the reader to look at the book of Swan [Swa05].

Chapter 8

The Blog



8.1 September 2009

September-28th-09. Welcome to the String Pattern Recognition blog. The beginning of this year's course has been the weirdest ever. It seems that the fact that my course will be taught in English has discouraged students from registering. The very first day of the course I checked on the computer how many students had registered: none. Really sad, very disheartening. So much work preparing the material for nothing.

Later on, while having a cup of coffee at the cafeteria, a couple of students came to see me. They would take my course, but they were not able to register. There was some nonsensical problem with the registration, but meanwhile they were allowed to attend class. I suppose it is the chaos associated with change of the school's curriculum.

Classes will start next Wednesday. I am looking forward to it.

September-30th-09. English Phonetics. Since classes will be taught in English I decided to give a first lecture on English phonetics. I will write the phonetic transcription of the most difficult words on the blackboard. Therefore, I have to ensure they know the IPA alphabet and also they are able to identify the various sounds of English.

After asking, the students told me that they already knew some phonetics. However, pretty soon I realized that they did not know as much as I needed for this course. For most of them, a wealth of the material was unknown.

I (briefly) reviewed the following topics: vowels, consonants, semi-consonants, diphthongs, pronunciation of plurals, pronunciation of the third Person of singular of Present Simple, pronunciation of Past Simple. See the course web page for more information.

8.2 October 2009

October-2nd-09. Review of Algorithms and Complexity - I. Normal, only for this time. Three students, only three, showed up. Later a fourth one also joined the course. This cheered me up. I might end up by having a nice crowd in the classroom.

Students had some difficulties following my lecture in English, mainly, I believe, because of lack of vocabulary. As I suspected so, I created the English corner, a part of the blackboard where I write down (phonetics included) those words I detect they have difficulties with. Attached you will find a list with some of them. I will give a similar list for each lecture in this blog.

Students are very shy when it comes to speaking up. Furthermore, they have trouble making clear, precise sentences that can express their thoughts. I will insist on them to speak up, even if they make mistakes. Let yourself go, guys!

A summary of the last lecture is the following: definition of algorithm, Knuth's definition of algorithm, the distinction between algorithm and program, correction and complexity of algorithms, the sorting problem, insertion sort, prove of correctness for insertion sort, analysis of algorithms, worst-case complexity, best-case complexity, space complexity, average-case complexity, models of computation, real RAM, computing the complexity of insertion sort. And that's all, folks!

<u>Difficult words:</u> Tough/tʌf/, rough/rʌf/, definite/'definət/, definiteness/'definətness/, infinite/'infinət/, finite/'fainait/, finiteness/'fainaitness/, effectiveness/e'fektivness/, pseudocode/ˌsu:do'kəʊd/, insertion/in'sɜ:ʃn/, cost/kɔst/, assignment/ə'sainmənt/, nondecreasing/nɔndekri:siŋ/, work out/wɜ:kaʊt/, mean/mi:n/, fine/fain/, coarse/kɔ:rs/, algorithm/'ælgəriðəm/.

October-5th-09. Review of Algorithms and Complexity - II. Correctnes of algorithms and mergesort. I reviewed the mergesort algorithm. I also gave a proof of correctness. Surprising enough, the lengthy part of the proof is to check that the merge step is correct. That part has to be proven in a straightforward way, without using induction. The main body of the algorithm does have to be proven, since it is a recursive one.

I haven't got my student's names so I can't start the fun part of this blog (the irony). I need their names to anagramize them. Next day I will get them.

Chaos in the registration process is still on. When will it finish? I realize it is a miracle that I have students for these course.

Difficult words: Swine flu/swain flu:/, recursive, return/rɜ:turn/, integer/'intidʒɜ:(r)/, therefore/ðeəfɔ:(r)/, hence/hens/, as a consequence/əzə'kɔnsikwəns/, result/ri'zʌult/, forecast/fɔ:ka:st/, foretell/fɔ:tel/, catch/kætʃ/, namely/neimli/, grasp/græsp/, lack/læk/, deadline/dedlain/, just as well/dʒʌstəzwel/.

October-9th-09. Review of Algorithms and Complexity - III. Two new students arrived, **Sigmund H. Noplace** (from now on, just **Sig**) and **Roger Jean Juro**, a burly guy born to a French mother (hence, his middle name). In total, I have five students. Only five? Five scatterbrains? I don't think so. Let me introduce the rest of the gang: **Leonine Madcalf**, **Auralee Redfez** and **Marina Dazehair**. They all seem determined to finish their double degree. Then, five plucky fellows? Five fellows showing courage and spirit even under the most trying circumstances, for instance, the registering process for this course.

I explained their first graded project. It is about implementing six basic sorting algorithms, compute their complexities and conduct experiments to check the running times in practice. They neither snorted nor puffed, but they seemed a little bit overwhelmed. They haven't sat down and hammered away at it yet.

Difficult words: Hand over/hændəuvɜ:/, odd/ɔd/, subtle/sʌtl/, bring about/briŋə'baʊt/, bumpy/bʌmpi/, raw data/rɔ:deitə/, blind/blaind/, bear on mind/beə:rɔnmaind/, insight/insait/, catch/kætʃ/, rationale/ˌræʃə'na:l/, upper bounds/ʌpɜ:baʊndz/, lower bounds/lʌwɜ:baʊndz/, figure/figɜ:(r)/, last/last/, messy/mesi/, keep somebody posted/'ki:psʌm'bʌdi'pəʊstid/.

October-16th-09. First Pattern Recognition Algorithms. Inversion of terms. In my students went. Satisfied they looked. Are they enjoying? Flu caught Roger Jean and absent he was. In a huge traffic jam Sig Noplace was caught. Possible, it wasn't his arrival. Out he was, waiting, sent into the depths of despair. Late he arrived.

Today is presentation day. To the blackboard students will go. Solving-problems section is coming up. Leonine starts off with a problem on complexity. Good, in general, small mistakes he made. Next, there Auralee goes. Very detailed work. Outstanding, in fact. The important points, she has grasped. Finally, out Marina went. A little misunderstanding happened. Nothing to worry about. She had worked on the problem and that is enough for me.

Difficult words: Claim/klaim/, instill/in'stil/, state/steit/, exist/ig'zist/, on the one hand and on the other hand, hold/həʊld/, workout/wɔ:kaʊ/, suffice/sə'fais/, assume/ə'sju:m/, whereas/weɜ:æz/, staff/stæf/, carry on/kæriɔn/, fertile/fer'tail/, whether/we'ðɜ:(r)/, wonder/wʌndɜ:(r)/, corpus/kɔ:pəes/, fuge/fju:g/, genre/'ʒa:rə/, theme/θi:m/, template/'templeit/, nod/nɔd/, find out/faindaʊt/, straightforward/streit'fɔ:wa:d/, mismatch/mismætʃ/, looking forward to.

October-19th-09. First Pattern Recognition Algorithms. Voluptuous (I). I come in the classroom. Today I feel happy, although I don't know why exactly. I smile at myself. "Today's lecture is about prefixes and suffixes," I remind myself. All of a sudden, I get unexpectedly proud, I feel myself standing. I repeat those words -prefix and suffix- in a faint murmur. My coming aloft, far from fading away, stands still and stiff. I'll sham normality.

Strangely enough, there is a whitish mist floating around in the classroom. I look at my students through the mist, which seems to melt, to dissolve, and to become translucent when the brilliant sun comes through the large glass window and magically colors and outlines everything. I distinguish their expressions among the mist and they look happy today. Moreover, I'd say they are grinning at me with the tenderest understanding expression. However, they can't know. Or can they?

I start my lecture. Symbols for prefixes and suffixes seem to me cases for jewelry, bird's nests for warrior's refuge.

I ask one of my female students to solve a problem on the blackboard. When coming closer, she slips and falls down, and in so doing displays some of her charms; but she jumped up very quickly.

"Did you see my agility, Sir?," says she.

"Yes, Miss, I did," I exclaim, "but I never heard it called by that name before!"

(To be continued...)

Difficult words: Prefix/'prifiks/, suffix/'sʌfiks/, order/ɔ:rdɜ:(r)/, relation/ri'leiʃən/, antisymmetric/'æntisi'metrik/, reckon/'rekən/, naive/naiv/, loop/lu:p/.

October-23th-09. First Pattern Recognition Algorithms. Voluptuous (II). Next class. I can't think of the word "prefix" or "suffix" without feeling a sweet stretching of my morning pride. I also remember my student's agility and feel embarrassed. Sometimes, life puts one in awkward situations. Anyhow, in the cold light of the day, it was just a curious anecdote.

I am ready to give another fun, beautiful lecture on string pattern recognition. Today I will cover the naive algorithm and its complexity. I start speaking calmly but fluidly. Somehow, I don't keep eye contact with my students. I am concentrated on the formulas written on the blackboard. I think I should look at them. I do so.

Radiant countenances, splendorous gazes, dazzlingly beautiful figures, youth oozing out of them at every pore, tireless determination to drink up life, all that was possessed by my students. All that is waiting in front of me. They also look at me, as if waiting some prompting signal. Skeptically but at the same time amazed, I look at them once more.

I see a woman with big brown eyes, so big that they bewilder me as nothing could hide them (not even padding). Rounded face, jet black straight hair, very stylish horn-rimmed glasses, generously curvaceous, disarmingly sonsy, her smile, made of perfectly aligned teeth, is shinning white in the middle of her face, ranging from innocency to mischievousness. Her presence fills up the whole classroom with a vivid sense of sensuality.

I shake my head in wonder and disbelief.

I turn my look from that student and my eyes trip over another female student. A brunette, a slender figure with shapely legs, gracious waistline, the delicate arms of a belly dancer and olive-coloured complexion, just returns my look with a knowing smile. At that moment I notice her neck an then hopelessly sink in the vision of her cleavage. This is the tale of two cities split by a gorge, each of which is on the top of the fullest, most tantalizing mountains. Raspberries grow tumidly.

(To be continued...)

<u>Difficult words:</u> Loop/lu:p/, in fact/in'fækt/ (different from fuck/fʌk/), differ from/'difɜ:(r)frəm/, disjoint/dis'dʒɔint/, cancel out/'kænslaʊt/, expectation/ˌekspek'teiʃn/, geometric/ˌdʒi:ə'metrik/, bound/baʊnd/, trouble/trʌbl/, drop/drɔp/, start from scratch/'sta:tfrəm'skrætʃ/, expository/ˌekspəsi'tɔri/, purposes/pɜ:pəsiz/, speed up/spi:dʌp/, expansion/ik'spanʃən/, there is a fly in the ointment, datum/'deitəm/, data/deitə/, horny/hɔ:ni/, put forward/pʊtfɔ:wəd/, desire/di'zaiə(r)/, no longer, so far so good, truncate/trʌnkeit/, cut off/kʌtɔf/, modulo/'mɔduləʊ/, moduli/'mɔdulai/, score/skɔ:(r)/, crucial/kru:ʃl/, rely on/rilaiɔn/.

October-26th-09. First Pattern Recognition Algorithms. Voluptuous (III). I firmly decide to avert my eyes from my female students as they all seem to be teasing me. Somehow I find that game very dangerous. Even worse, it might not be their game but that of my fevered imagination. If so, why is it betraying me? No, no, that cannot be. They are playing with me, they are mocking at me. It must be that.

I look at the boys and notice one of them. I've seen him since classes started, but suddenly, and for no apparent reason, he looks extremely handsome and irresistible to me. He is neither tall nor short but of the ideal height. He's got rosy cheeks, his face is alert and lively, with a sharp chin and shrewd little eyes. His body is harmoniously proportioned, the daintiness of his hands standing out conspicuously. He looks magnificently and enormously sensual. I feel vexed, losing my composure. I clear my throat a few times as to make a pause, get my strength back and carry on with my explanation. For a few minutes everything seems to be back to normal. My attention is riveted by the material I have to explain. I cover in detail the probabilistic analysis of the naive string matching algorithm. They don't seem to have problems with the random variables involved. They seem to appreciate the use of probabilistic tools in computer science.

Out of the corner of my eye I notice something. Another male student is undeniably staring at me, grinning from ear to ear. What's going on in this classroom? This student is burly -I discreetly examine him while talking- smooth-cheeked and with absolutely no body hair, which gives his skin an amazing shining quality. The smoothness of his skin must be to die for. I can imagine such smoothness under my lips. Again, I strongly shake my head in wonder and disbelief.

I interrupt my explanation. I remain quiet for almost a minute, thinking about the meaning of all this. I realize they are waiting for some prompting signal, but I don't know which one.

(To be continued...)

Difficult words: Lead/li:d/, handicap/'hændicæp/, nuts/nʌts/, browse/braʊz/.

October-30th-09. First Pattern Recognition Algorithms. Voluptuous (IV). I am still wandering up and down the classroom while they are staring at me. What are they expecting from me? My head is boiling and my brain is so full of sensations, feelings and ideas that it feels like it is going to burst. All of the sudden, a quote from Shakespeare crosses my mind. I catch it at full strength in the midst of the pandemonium that my mind has turned into now. It reads

The blood of youth burns not with such excess as gravity's revolt to wantonness.

I savour each word. Like Proust's madeleine this quote triggers fond, vivid, long-forgotten memories, memories of fundamental truths. I will not be gravity revolting to wantonness. I will not smash the blood of youth on behalf of gravity. The grand battle of life is to teach blood limits, to tame it and make it understand man does not live by bread alone. But if blood has never been sated with bread, how can it know of any temperance?

I face them all and with a big smile lift my arms. My eyes are shinning bright. I pronounce each word with exceeding joy.

The flesh is proud.

Let us start this Saturnalia.

(To be continued...)

Difficult words: Transcription/træn'skripʃn/, factor/'fæktə(r)/, bind/baind/, core/kɔ:(r)/, portion/pɔ:rʃn/, utilize/'ju:təlaiz/, collision/kə'liʒn/.

8.3 November 2009

November-2nd-09. The Linear-Time Algorithm. Voluptuous (V). Big brown eyes are caressed in slow motion by the most exquisite cold hands, which are shortly warmed. Also, pretty soon rosebuds turn into raspberries under the sultriest tongues, the keenest fingertips. Gentlemen come home, sweet home, and enjoy the silky, adorable, kissable peaches' commodity. They ring belles' bells with their clappers and the deepest moans are heard in its splendour. No boy in the boat is left without the deserved attention, paid very slowly, with meticulous calm. Shudders ripple across everybody. They couple with each other in an unutterable unity. Two bellies are nailed together and, magnificent, a beast with two backs rumbles, howls, roars and finally laughs heartily.

Watching such frolicsome performance I think art is exceeding what Nature gave to man. There is no point in opposing the blood. Gravity is wrong.

Slender figures glide, slither and wriggle like salamanders among the unruly members of the crowd, pacifying them, quenching the fever consuming them. Belles pray, kneel and worship at the altar. Little and big brothers merrily accept the divine offering. They bite their lips, close their eyes, also moan and some have the sweet agony, that little sweet death. Others, after the prayer, put the devil in hell so that he receives the punishment due.

As for me? I am the most grateful audience of this performance. I will not participate, as youth must learn by itself, but I watch everything with the eagerest eyes.

And then off I go. Yes, indeed, I beat off, jack off, jerk off, jill off, pull off, rub off, whack off, wrench off and work off. I do it with dainty manners, placidly, always with an approving smile. And I come off. My spendings gently meander down my belly. Its warmness makes me happy. I abandon myself to pleasure.

(The End)

Difficult words: Sake /seik/, for old time sake, disposal /di'spəuzəl/, collision /kə'liʒən/, bind /baind/, bare /beə(r)/, naked /'neikəd/, nude /nju:d/, staff/staves /stæf/, stuff /stʌf/, at the bottom line, shrink down /ʃriŋk daun/, awkward /ɔ:kwərd/, nipple /'nipəl/, wit /wit/, outwit /'aut'wit/, indepth /'in'depθ/, try your hardest, get rid of something, character /'kærəktə(r), gather /'gæðə(r)/.

November-6th-09. The Linear-Time Algorithm. Demostration. Student's session today. They have to go to the blackboard and give a lecture. I asked them to solve certain problems. Time to show what they did.

Most of the problems only required thought, pure and plain thought. However, they insisted in exposing their ideas by displaying a piece of code. That was the wrong way to explain an idea to solve a problem. I barely could understand their points, so obscure their presentations were. In some cases they miss the most relevant points such as correctness or complexity.

That's why I am organizing a DEMONSTRATION.

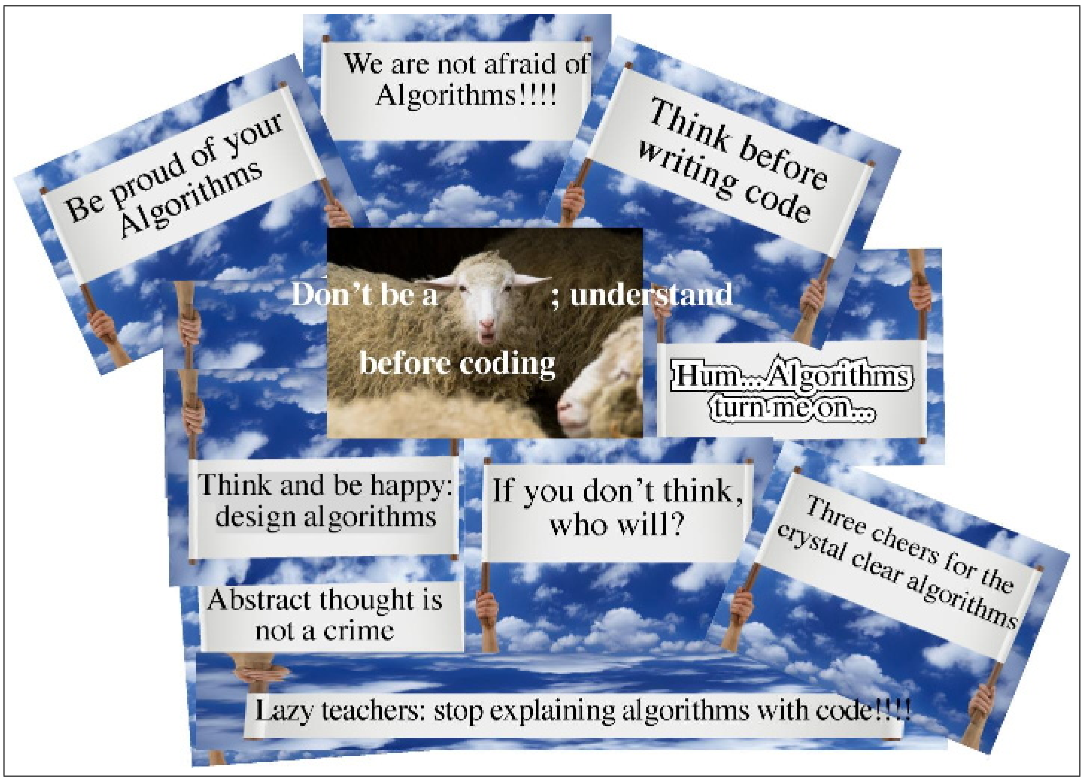

We are not afraid of Algorithms!!!!
Be proud of your Algorithms
Think before writing code
Don't be a ; understand before coding
Hum...Algorithms turn me on...
Think and be happy: design algorithms
If you don't think, who will?
Three cheers for the crystal clear algorithms
Abstract thought is not a crime
Lazy teachers: stop explaining algorithms with code!!!!

Figure 8.1: Demonstration!

Difficult words: drag /dræg/, attain /ə'tein/, you're used to sth ≠ I used to do sth, loader, copycat /'kɒpikæt/, by contradiction /ˌkɒntrə'dikʃən/, from now on, from now onward /'ɒnwəd/, cloud /klaud/, purist /'pjurəst/, pending /'pendiŋ/, yet /jet/, overlap /ˌəuvə'læp/, my piece of advice /əd'vais/

November-13th-09. The Linear-Time Algorithm. Calm and normality. Today's class consisted of solving all the problems associated with Gusfield's linear-time algorithm. Sigmund H. Noplace was missing today. Leonine was pretty sharp and answered most of the questions correctly and with good ideas. Roger participated quite a lot, too. His English is a little shaky and I had some difficulties to understand. Auralee and Marina were somewhat shy. I don't know whether they don't master the material yet or their English is rusty and don't manage to get through it.

There was a final question, how to generalize the Z algorithm to the 2D string matching problem, for which Leonine gave a couple of ideas that I proved incorrect. At the last moment he came up with another idea. I wasn't sure whether it would work. I asked him to think about it carefully and present it next class, either it is incorrect or it is not. A fresh idea..., I like it. I am looking forward to hearing it.

I hope that, in the meantime, Auralee and Marina decide to participate more. I have the feeling they are more afraid of participating than they should.

As you can see from the list of words below, last lecture I was verbose.

Difficult words: Sth will do the job /dʒɔ:b/, lecture /'lektʃə(r)/, teacher /'ti:tʃə(r)/, professor /prə'fesə(r)/, palindrome, pun /pʌn/, now, we're talking /'tɔ:kiŋ/, speed up /spi:d/, deduce /di'du:s/, trap /træp/, tight /tait/, compact /kəm'pækt/, strip /strip/, later on /'leitə(r)/, early on /'ɜ:li/, coordinate /ˌkəu'ɔ:dineit/, come up /kʌm/, counterpart, zip /zip/, tease /ti:z/, draw back /drɔ:/, verbose /vɜ:'bəus/

November-16th-09. The Linear-Time Algorithm. Ideas. Hey, I've got an idea! I'll solve the exercises from Chapter 4. There are some to be solved! Hey, yes, that's what I'll do. They definitively deserve it. I am going to solve Problem 4 of Section 4.3. It is about how to generalize the Z algorithm to two dimensions. In general, it is not easily generalizable. Leonine lifts his arm and, and confidently enough, claims he's got an idea to do it. Please, yes, have ideas! It is all I ask you for. Be university students! Have ideas. Be proud of them! If it is not now, then when?

He presented his idea. It wasn't clear enough; he gave a hand-waving argument, but I thought it could be true once passed through the sieve of rigourness. Roger got excited about the problem and went to the blackboard to discuss it with us. We got more and more excited. We gave arguments for and against Leonine's idea. Auralee and Marine were staring at us not daring to participate. I think they didn't catch a part of the discussion. If so, they should have stopped us and asked for explanations. Do not let an idea pass by without understanding it. Understanding is a pleasure, different from that pleasure, but a pleasure at the end.

I promised to give Leonine more time next lecture to put forward his idea to the utmost detail . I beg you: **Have ideas!**

Difficult words: Difficult words: I don't buy it, align /ə'lain/, stepladder /step'lædə(r)/, at least /li:st/, at worst /wɜ:rst/, hint /hint/, clue /klu:/, close /kləuz/, closed /kləuzd/, diagonals /dai'gənəlz/, trail /'traiəl/

November-20th-09. The Linear-Time Algorithm. Can the linear-time algorithm be generalized to 2D. Discussion. I finished my last blog by asking students to have ideas. In this lecture an idea did come up. In a casual way I solved Problem 4. The problem posed the question of how to generalize the Z algorithm to solve the 2D string matching problem. I took for granted that everybody would agree that the generalization in linear-time is not feasible. I made a quick, comment about the impossibility of generalizing the concept of Z-box to two dimensions. All of a sudden, Leonine sprang from his seat, leaped on his chair and asked to be heard. To my astonishment, he seemed to have something important to present. I gave him the floor to explain himself. His voice faltered as he began his speech. It was difficult for him since he had to speak English. I helped him with some words that didn't come to his mind. I wasn't understanding his explanation in full detail. Some of its pieces didn't fit together. I tried to help him by making some questions. Little by little, the explanation of his idea got clearer. Roger felt he had understood Leonine's idea and with great zest threw himself into the discussion, correcting Leonine, re-wording his explanations and also adding his a bit of his own. The discussion became passionate and fast (in spite of their English).

From the corner of my eye I watched Marina and Auralee. Both were quiet as quiet as a grave, specially the latter. They felt intimidated by the speed at which ideas were presented. Their attitude made me a little sad, a little angry and a little worried. I could see the fear of thinking in Auralee's face; that fear is often unjustified and linked to a lack of self-steem. I will talk to her after my lecture. Please, do sink into the discussion! If you don't understand, ask for explanations. Nobody should feel ashamed because of asking. If you don't ask now, when? The more you fight to understand, the more you creative you are. Don't try to survive my course without thinking; it won't work.

We ran out of time , Leonine was already defending his idea with aplomb. Next lecture we will continue.

Difficult words: Think over /thiŋk əuvə(r)/, mull over /mul əuvə(r)/, make yourself at home, clever /'klevər/, pull my leg /pul mai leg/, giving me hard times /giviŋmi: ha:d taimz/, strand /strnd/, sloppy /'sləupi/, feasible /'fi:zəbl/, layer /leiə(r)/, spell out /'spel aut/.

November-23rd-09. The Linear-Time Algorithm. Leonine's idea. His idea consists of generalizing the Z algorithm to two dimensions by distributing the separation character \$ in a special way. Let P be a two $m_1 \times m_2$-matrix and let T be a $n_1 \times n_2$-matrix, the former being the pattern and the latter being the text. Call $P_1, ...P_{m_1}$ the rows of P. Leonine builds a string S by concatenating each row P_i with $m_2 - n_1$ separation characters and after that also by adding the text by rows. Over string S he runs the pre-processing stage of the Z algorithm. The Z-values of S are thus obtained. A Z-value equal to m_2 at a position greater than $m_1 \cdot n_2$ identifies a copy of P_1 in the text. This is true because P_1 is a prefix of string S. Leonine was right when making this claim. However, this reasoning does not work for the occurrences of P_2, which is not longer than a prefix of S. Therefore, the Z-values from this point on are of no use. He tried, with the help of Roger Jean, to fix this gap in the reasoning, but either his conclusions were still false or led to quadratic, brute-force-like algorithms.

I must thank Leonine for his attempt. I (we, I hope) enjoyed the discussion a great deal. It showed the rest of the class that thinking makes a difference. This time Marina and Auralee participated a little more, but not to my entire satisfaction. I will keep pushing them. Nobody leaves my course without facing themselves in that regard.

Difficult words: Trial and error, rehearsal /ri'hɜ:səl/, fulfill /ful'fil/, busty /bʌsti/, boobs /bu:bz/, tits /tits/, bosom /'buzəm/.

November-27th-09. Suffix Trees - Introduction. English Phonetics.

/Tu'day ai 'sta:tid 'sʌfiks tri:z. It wɔz sʌtʃa smu:ð 'lektʃə. 'Kɔ:nsəpts fləud ænd ðei græspt ðem 'i:zili. Ai təuld ðem ðæt iks'pleiniŋ ðis 'ælgəriðəm wil teik mi: θri: wi:ks. ðei si:md nɔt tu: bi'li:v mi:. Həuld tait./

November-30th-09. Implicit Suffix Trees - The Naive Algorithm. Instructions.

Instructions:

1. You must understand at all times.
2. Understand thoroughly; if you haven't understood something, ask.
3. Asking questions is for free in this class. Do ask.
4. Asking questions does not make you stupid, it proves your interest. Do ask.
5. Asking questions aloud is a way to organize and confirm your ideas. Do ask.
6. Have a genuine interest in this course. You are here to learn.
7. Learning is a fundamental process in human beings. Learning keeps you alive at all ages.
8. Not having a genuine interest is like deceiving yourself.
9. This is not an usual course. I require interest, effort and creativity from you.
10. You are letting yourself down by not giving your best.
11. It is a goal of this course to raise your self-steem by doing good, deep and thorough work. Don't save an effort to achieve this goal.

12. Be prepared to use knowledge obtained from other courses. This is not a watertight compartment. You must put together pieces from different subjects.

13. Be rigourous. You are in the university, the highest centre of education. Don't do a botch job. They are always noticeable and obtain very low grades.

14. Write correctly. Write directly and effectively. Show with your greatest pride your clarity of thought.

15. Be creative.

16. Don't think about the grades. Watch what you learn, what you give of yourself; good grades will be given to you as a natural consequence of the right attitude.

17. Play according to these instructions.

Difficult words: The state of the art /steit/, belong to /bi'lɒŋ/, look /lu:k/ plus at, for, up, down, back, tongue twister /tʌŋ'twistə(r)/, literally /'litərəli/, all /ɔ:l/, call /kɔ:l/.

8.4 December 2009

December-4th-09. Suffix Trees - Suffix Extension Rules and Suffix Links. Haiku. The difficult theorem.

If you pay no attention to stars,
If you don't struggle hard,
Who will?

An aura of strength encircled him,
I asked... He was..
Learning!!

Earning is learning,
Learning is yearning,
...Turning me on!!

Difficult theorem today,
Open up your mind,
Let it blow!!

Go over it one more time,
Try, you will make it,
Persevere.

Be proud of yourself. How? Learn
Learning is like opening
A treasure.
(The treasure of yourself.)

Watch your laziness, the most
Dangerous,hateful
Companion.

Now or never? Soon or late?
Promptly or idly?
You decide.

Difficult words: Hand-waving /hndweiviη/, So called /səukɔ:ld/, sponge /spʌndʒ/, roughly /'rʌfli/, shortcut /shɜ:rtkʌt/.

December-7th-09. Suffix Trees - Creation of Suffix Links - The Difficult Theorem. E-VOID. I got a bit of mortification. I think pupils didn't grasp an important offspring that it was shown last class. I got a thing about making a proof too much compact and difficult to follow. I do it again to my pacification of mind and also th_irs. This offspring is crucial to grasp many parts of this algorithm. This clip it was caught profoundly. Possibly, or such it is my ridiculous simplicity, pupils did will to hit books. Possibly, pupils did study that important offspring from Friday to Monday. I know pupils will throw an inquiry (or many) about this offspring. I augur that it will occur. Just wait.

Second hour (NORMAL): This second class is about how to give speeches. It is always one of my favoured lectures. I love speaking in public. I have a passion for convincing people, communicating ideas and emotions. I enjoy very much moving among ideas and playing with language, dike and springboard of our thought at the same time. First, I spoke to them about the act of communication in general. The key, at least to me, is to speak always, always, always, and with no exception, about what you get excited about. Enthusiasm is of paramount importance when it comes to speaking in public. Without (sincere) enthusiasm there will never be effective communication. Then, I give some guidelines: say something, speak to somebody (not to yourself), repeat and do not repeat, make an outline, always organize, honesty is the best policy, look at your audience, introduce a dash of humour here and there, articulate your words, make a wise use of silence, summarize your points from time to time, and stop.

Difficult words: Thrill /θril/, due /dju:/, overcome /ˌəuvə'kʌm/, decrease /di'kri:s/, scheme /ski:m/, juggle /'dʒʌgəl/, show stopper /ʃəu'stɒpə/, wand /wɒnd/.

December-11th-09. Suffix Trees - Extension Algorithm/Edge-Label Compression - speeches. Today everybody will give their speeches. Oh, yeah! Great acts of communication will be laid in front of us! Marina Dazehair was the first one. She gave an interesting speech about her passion: engraving. She went a little too much into the technical details and less into the roots of her passion. Nevertheless, the way she spoke about engraving greatly showed that passion. Next, Roger Jean Juro took the floor. He spoke about mental maps and creativity. His speech turned out to be very interesting. He had a clear idea of what creativity means and, unfortunately, how much formal education, as is organized presently, has wasted his time... Yes, he was painfully blunt about that particular point. His speech, in spite of his problems with English, had a tight structure and was presented with a sober excitement.

Then, it was Auralee Redfez's turn. This woman said something that perturbed me deeply. When I told them to pick a topic they would have a passion for, she told us she didn't have any, for life had been hard enough on her as for having a clear passion. After some thought, she said that kids could be her passion, especially her nephew. Her speech was about her nephew, but I certainly couldn't feel the passion. Her mind went blank several times because of her English. The content was disorganized, presented in a stiff, ungraceful way. I know she is clever and very hard-working, but to me her apparent lack of passion is a mystery (maybe, it is just latent, waiting to be awakened).

Leonine was next. He gave a funny speech about the virtues of virtual worlds. He told us how wonderful virtual worlds are, how many things you can build with a computer, how much fun you can have with it. However, he wasn't convincing at all. In fact, he didn't transmit the feeling that his speech was prepared. He spoke as if he were in the pub with his buddies. It was entertaining, but it wasn't a true speech.

The conclusion is that giving a speech is not easy. First of all, one must be aware of the significance of such an act of communication. Secondly, it has to be prepared and it has to be rehearsed. There is some technique to be learnt. Bear this in mind.

December-14th-09. Suffix Trees - Phase Algorithm. Defying!

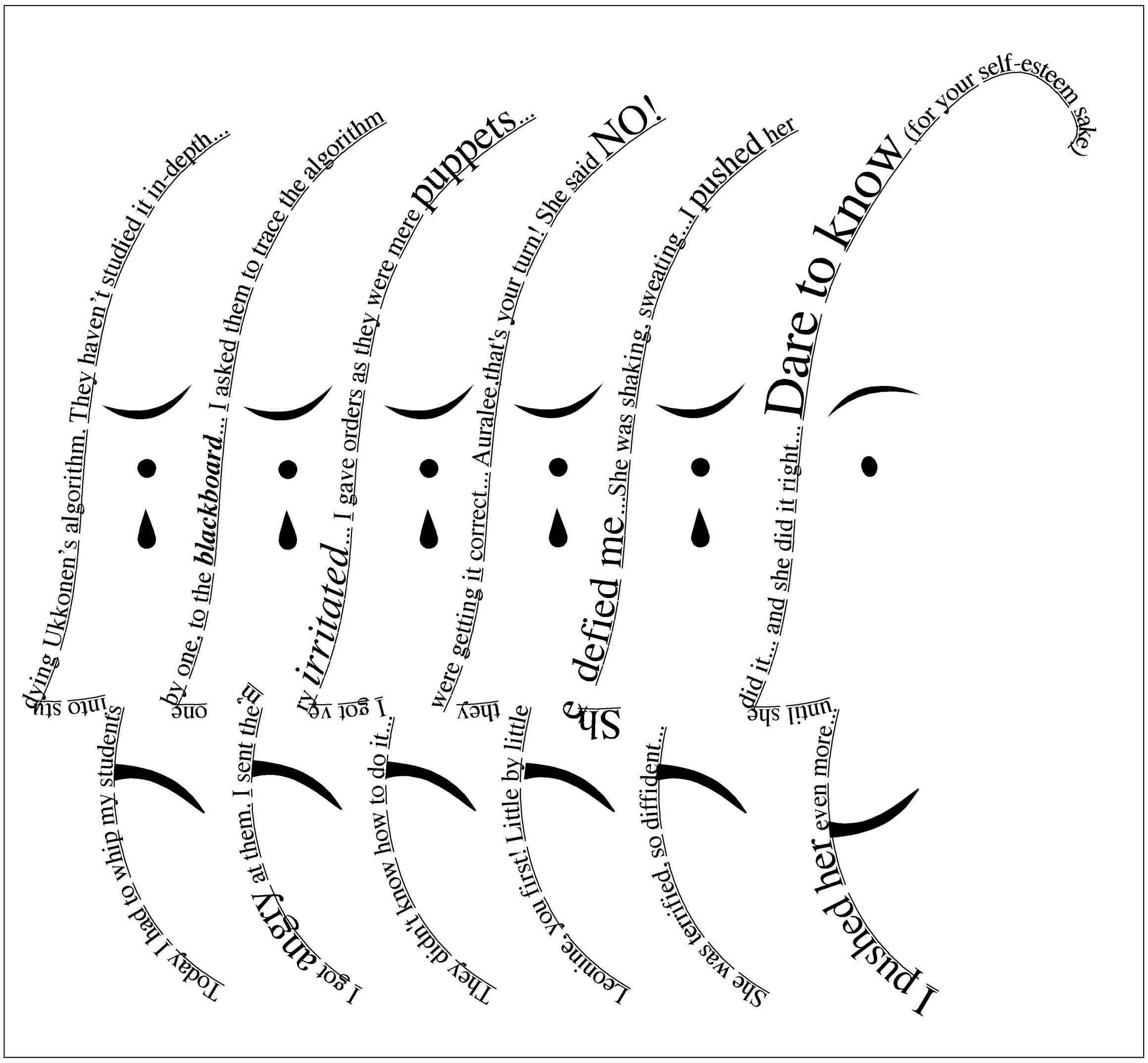

Today I had to whip my students into studying Ukkonen's algorithm. They haven't studied it in-depth...
I got angry at them. I sent them one by one, to the blackboard... I asked them to trace the algorithm
They didn't know how to do it... I got very irritated... I gave orders as they were mere puppets...
Leonine, you first! Little by little they were getting it correct... Auralee, that's your turn! She said NO!
She was terrified, so diffident... She defied me... She was shaking, sweating... I pushed her
I pushed her even more... until she did it... and she did it right... Dare to know (for your self-esteem sake)

Figure 8.2: Defying!

December-18th-09. Questions - Student's Presentations. Compassion. First casualty: Sigmund H. Noplace, pitiful creature, wasn't able to complete the last final project, the big project, the one trying out your real skills as programmers. I know this is hard. Unfortunately, you've never been taught what an algorithm is actually like; you, oh, poor students!, have never seen the proof of correctness of an algorithm; you, poor wretches, have seen very few times what a complexity analysis is; even worse, you, misfortunate victims of the damn flood of final projects raining these days, have not been accustomed to rigor and planning. I, poor me!, was really sad he, poor him!, couldn't make it. Again, I am quite sure Sigmund is clever to do this project, although under the adequate circumstances, which, somehow, they, educational system's scapegoats, don't know how to create. Because of the past projects and his class participation, he, as mortal as himself at least, has got a pass mark.

The rest of the class was in tears. Compassionately, with extreme sweetness, they made many questions to me. They were all the right questions. I, distrustful and unworthy mortal, was surprised by their general knowledge of Ukkonen's algorithm. However, I, contemptible being, had a foreboding about them completing the final project by next Monday. Humbly, I was seriously doubting it. It wouldn't surprise me. Normally, I give them projects that are a little bit above their level so that I can reward effort properly. On the other hand, I have to make an enormous effort to motivate them.

December-21st-09. Suffix Trees - More Student's Presentations. Today is presentation day. Three days ago they had barely started to programm suffix trees. Have they been able to achieve it in such a short period of time? The most optimistic part of myself insists in believing so. The pessimistic part of myself, or should I say the well-informed, realistic part of myself, murmurs "they haven't." I walk to the classroom. None of my students look at me in the face and they are all crestfallen. I understand. They are telling me they were unable to complete the project. They looked tired, exhausted. I give them Christmas holidays to complete the project. I spend the rest of the class answering their questions about implementation, listening to their worries and chatting friendly. Nevertheless, they have already accomplished a whole lot.

Learning is a perpetual cycle of falling down and standing up (hopefully, stronger). If you can find a path with no obstacles, it probably does not lead anywhere. I believe there are more urgent and important occupations than the ridiculous amount of time we waste complaining.

I am looking forward to seeing what will happen January the 22nd, the day when they will finally do their presentations. Will they be able to stand up?

8.5 January 2010

January-22nd-10. Final Presentations.

January-18th-10. Pattern Recognition Course- Final Presentations. Today my students will make their final presentations. I can't wait to see them. They haven't asked me anything about the way to make the presentations or programming Ukkonen's algorithms. That surprised me quite a bit. It could be very bad news or really great news. I'll find out very soon.

Marina is the first speaker. Her English is awkward, unexpected, somewhat twisted, but I can see she has put some effort into it. She introduces the main definitions, but her presentation is following my notes to the letter. I can't see what she has assimilated, digested from the theory. She struggles against her English mistakes, which are more numerous as her presentation proceeds. She should have prepared her speech more. Surely, it is not well rehearsed (I warned them about this).

Then Auralee comes. Her English is awful. I can hardly understand anything. Her grammar and pronunciation are confusing me and the presentation is quite sloppy. She is very nervous. Her mind often goes blank during her speech. I can notice she is sweating. She finishes and walks away, head bowed.

Leonine is next. His accent is strong, but his grammar is acceptable and I can follow his ideas. He presents the core of Ukkonen's algorithm. I can tell he knows what he is talking about. He doesn't get lost in details and is able to answer my questions with aplomb. There were a couple of nasty surprises. The first was that they didn't implement one of the speed-ups as they should. It doesn't increase the time complexity drastically, but it should have been implemented correctly; it is a question of principles. The second has to do with testing software. Their presentation ended without a single mention to how the program was tested. In view of my insisting questions, Leonine said that they had used 3 files to test their program (sic). I was speechless. After some thought, I asked how big those files were. My surprise was bigger. Leonine said that each file had 11 characters. There is no way on earth Ukkonen's algorithm can be tested with three 11-character files. Essentially, the message was that the program hadn't been tested. So sad, so disappointing. So much time and energy talking about how important it is to test your program. Obviously, they lost precious points.

That's all. I have had very good moments with them. It has been really hard. If you are not determined, you better not to teach a course like this. To me, in the long run it has paid off.

See you next time.

Bibliography

[AG86] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil String Searching Strategies Revisited. *SIAM Journal of Computing*, 15:98–105, 1986.

[Aho90] A. Aho. Algorithms for Finding Patterns in Strings. In J. In van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 257–300. Springer-Verlag, 1990.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data structures and Algorithms*. Addison-Wesley, 1983.

[All09] L. Allison. Suffix Trees. http://www.allisons.org/ll/AlgDS/Tree/Suffix/, 2009.

[Apo85] A. Apostolico. The Myriad Virtues of Subword Trees. *Combinatorial Algorithms on Words*, Springer-Verlag:85–96, 1985.

[Ber00] D. Berlinski. *The Advent of the Algorithm: The 300-year Journey from an Idea to the Computer*. Harcourt, Inc., San Diego, 1. edition, 2000. 1. editon 2000.

[BM77] R.S. Boyer and Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20:247–248, 1977.

[BO05] A.D. Baxevanis and B.F.F. (eds.) Ouellette. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. Wiley, 3. edition, 2005.

[CH06] N. Cristianini and M. Hahn. *Introduction to Computational Genomics*. Cambridge University Press, 2006.

[CL04] C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, Cambridge, London, 2. edition, 2001. 1. editon 1993.

[Col91] L. Colussi. Correctness and Efficiency of the Pattern Matching Algorithms. *Information and Computing*, 95(2):225–251, 1991.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[Cro02] M. Crochemore. *Jewels of Stringology.* World Scientific Press, 2002.

[CSN$^+$04] R. Chavez, K. Schachter, C. Navarro, Peirano A., Bulli P., and J. Eyzaguirre. The Acetyl Xylan Esterase II Gene from Penicillium Purpurogenum Is Differentially Expressed in Several Carbon Sources, and Tightly Regulated by pH. *Biological Research*, 37:10–14, 2004.

[Den95] Daniel Dennett. *Darwin's Dangerous Idea.* Touchstone/Simon & Schuster, New York, 1995.

[GBJ02] John P. Burgess George Boolos and Richard Jeffrey. *Computability and Logic.* Cambridge University Press, Cambridge, 2002.

[GBY91] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1991.

[GI93] R. Grossi and G. F. Italiano. Suffix Trees and their Application in String Algorithms. In *First South American Workshop on String Processing (WSP'93)*, pages 151–182, Belo Horizonte, Brazil, 1993.

[GK97] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19(3):331–353, 1997.

[Gus96] Dan Gusfield. Simple Uniform Pre-Processing for Linear-Time String Matching. Technical Report CSE-96-5, UC Davis, Dept. Computer Science, April 1996.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, Cambridge, 1. edition, 1997. 1. editon 1997.

[HL16] G.H. Hardy and J.E. Littlewood. Contributions to the Theory of the Riemann Zeta-Function and the Theory of the Distribution of Primes. *Acta Mathematica*, 41:119–196, 1916.

[HR87] Jr Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* Cambridge University Press, 1987.

[Kin97] Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis.* Addison Wesley Publishing Company, 1997.

[Kle56] S. C. Kleene. *Representation of Events in Nerve Nets and Finite Automata (in Automata Studies).* Princenton University Press, 1956.

[KMP77] D.E. Knuth, J.H. Morris, and V.B. Pratt. Fast Pattern Matching in Strings. *SIAM J. Computing*, 6:323–350, 1977.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973.

[KR87] R. Karp and M. Rabin. Efficient Randomized Pattern Matching Algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[Lat97] David S. Latchman. Efficient Randomized Pattern Matching Algorithms. *The International Journal of Biochemistry and Cell Biology*, 29:1305–1312, 1997.

[Lew09] Cristopher Lewis. Suffix Trees in Computational Biology. http://homepage.usask.ca/∼ctl271/857/suffix_tree.shtml, 2009.

[McC76] Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[Mor68] D. R. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MP70] J.H. Morris and V.R. Pratt. A Linear Pattern-Matching Algorithm. Technical Report 40, University of California, Berkeley, 1970.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[oS01] The Department of Spanish. Phonetics: The Sounds of American English. http://www.uiowa.edu/ acadtech/phonetics/english/frameset.html, 2001.

[Pev00] Pavel A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach.* The MIT Press, 2000.

[Pre09] Oxford University Press. Oxford Advanced Learner's Dictionary. http://www.oup.com/oald-bin/web_getald7index1a.pl, 2009.

[Roa09] Peter Roach. *English Phonetics and Phonology Paperback with Audio CDs (2): A Practical Course.* Cambridge University Press, 2009.

[RS62] Barkley J. Rosser and Lowell Schoenfeld. Approximate Formulas for Some Functions of Prime Numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1988.

[Sed90] R. Sedgewick. *Algorithms in C.* Addison-Wesley, Reading, MA, 1990.

[Sim93] I. Simon. String Matching Algorithms and Automata. In *Proceedings of the 1st American Workshop on String Processing*, pages 151–157, Universidad Federal de Minas Gerais, Brazil, 1993.

[Ste94] G.A. Stephen. *String Searching Algorithms.* World Scientific, Singapore, 1994.

[Swa05] Michael Swan. *Basic English Usage*. Oxford University Press, Oxford, 2. edition, 2005. 1. editon 1984.

[Szp93a] W. Szpankowski. A Generalized Suffix Tree and its (Un)expected Asymptotic Behaviors. *SIAM Journal Computing*, 22:1176–1198, 1993.

[Szp93b] W. Szpankowski. Asymptotic Properties of Data Compression and Suffix Trees. *IEEE Trans. on Information Theory*, 39:1647–1659, 1993.

[THP05] Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical Suffix Tree Construction. *The VLDB Journal*, pages 281–299, 2005.

[Ukk95] E. Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] P. Weiner. Linear Pattern Matching Algorithm. *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[Wik09a] Wikipedia. Algorithm. http://en.wikipedia.org/wiki/Algorithm, 2009.

[Wik09b] Wikipedia. Algorithm Characterizations. http://en.wikipedia.org/wiki/ Algorithm_characterizations, 2009.

[Wik09c] Wikipedia. English Phonology. http://en.wikipedia.org/wiki/English_phonology, 2009.

[Wik09d] Wikipedia. International Phonetics Alphabet. http://en.wikipedia.org/wiki/Ipa, 2009.

[Wik09e] Wikipedia. Radix trees. http://en.wikipedia.org/wiki/Radix_tree, 2009.

[Wik09f] Wikipedia. Suffix trees. http://en.wikipedia.org/wiki/Suffix_tree, 2009.

[Zvi76] Galil Zvi. Real-time Algorithms for String-Matching and Palindrome Recognition. *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 161–173, 1976.

[Zvi79] Galil Zvi. On Improving the Worst Case Running Time of the Boyer-Moore String Searching Algorithm. *Communications of the ACM*, 22(9):505–508, 1979.

List of Figures

List of Tables

Index