Chapter 2
Basic Concepts

In the last decade more than before researchers have shown that combinatorial pattern matching and data compression are strongly related. On one hand, pattern matching data structures and techniques are used to develop time efficient implementations of almost any data compression algorithm. On the other hand, many classical pattern matching data structures suffer of space inefficiencies which could be mitigated by reducing the redundancy present in the indexed text via data compression.

The aim of the present chapter is that of introducing the most important models of computation and useful notation as well as some well-known results on combinatorial pattern matching and data compression that will be often referred in subsequent chapters.

The content of this chapter can be safely skipped by readers who are familiar with the fields of textual data compression and combinatorial pattern matching.

2.1 Models of Computation

In order to reason about algorithms and data structures, we need models of computation that grasp the essence of technological aspects of real computers so that algorithms that are proved efficient in the model are also efficient in practice. The next subsections present two of the most important models: RAM Model and External-Memory Model.

2.1.1 RAM Model

The RAM model tries to model a realistic computer. RAM stands for Random Access Machine, which differentiates the model from classic but unrealistic computation models such as a tape-based Turing Machine. The machine is formed of a CPU,
which executes primitive operations, and a memory, which stores the program and the data. The memory is divided in cells, called words, each having size \( w \) bits. It is usually assumed that \( \log n \leq w = \Theta(\log n) \), where \( n \) is the size of the problem.\(^1\) This assumption (usually referred as trans-dichotomous assumption (Fredman and Willard 1994)) is actually very realistic: a word must be large enough to store pointers and indices into the data, since otherwise we cannot even address the input. We can operate on words using at least a basic instruction set consisting of: direct and indirect addressing, and a number of computational instructions, including addition, subtraction, multiplication, division, bitwise boolean operations and left and right shifts. Each of these operations requires a constant amount of time and can only manipulate \( O(1) \) words at a time.

### 2.1.2 External-Memory Model

The RAM model does not reflect the memory hierarchy of real computers that have more than one layer of memory, with different access characteristics. The external-memory model (or I/O model) was introduced by Aggarwal and Vitter in (1988) to capture memory hierarchy with two layers. The model abstracts a computer which consists of two memory levels: a fast and small (internal) memory of size \( M \), and a slow but potentially unbounded disk. Actually, this model can be used to abstract any two different layers in the memory hierarchy (e.g., disk and network). Both the memory and disk are divided into blocks of size \( B \). The CPU can only operate directly on the data stored in memory, which consists of \( M/B \) blocks. Algorithms can make memory transfer operations, that is they can either read one block from disk to memory, or write one block from memory to disk. The cost of an algorithm is the number of memory transfers required to complete the task. In this model, thus, the cost of performing any other operation on data in memory is considered of secondary importance. Clearly any algorithm that has running time \( T(N) \) in the RAM model can be trivially converted into an external-memory algorithm that requires no more than \( T(N) \) memory transfers. However, faster algorithms can be often designed by carefully organizing and orchestrating accesses to data on the disk.

### 2.2 Notation

It is convenient to fix a common notation regarding strings. Let \( T[1, n] \) be a string drawn from an totally ordered alphabet \( \Sigma = [\sigma] \).\(^2\) We will refer to the \( i \)th symbol of \( T \) as \( T[i] \). Given any two positions \( i, j \in [n] \) such that \( i \leq j \), we will use \( T[i : j] \)

---

\(^1\) In the following we will assume that all logarithms are taken to the base 2, whenever not explicitly indicated, and we assume \( 0 \log 0 = 0 \).

\(^2\) The notation \([\sigma]\) denotes the set \( \{1, 2, \ldots, \sigma\} \). Unless otherwise stated, we will assume that \( \sigma = O(\text{poly}(n)) \).
to denote the substring $T[i]T[i + 1]...T[j]$. We will use $T_i = T[i : n]$ to denote the $i$-th suffix of $T$.

We will often compare strings resorting to their lexicographic order. The lexicographic ordering, denoted by $\leq$, is a total ordering induced by an ordering of symbols in the alphabet $\Sigma$ and it is defined as follows. Given two strings $S$ and $S'$ drawn from alphabet $\Sigma$, we say that $S$ is lexicographically smaller than $S'$ ($S < S'$) if and only if $S[1] < S'[1]$ or both $S[1] = S'[1]$ and $S_1 < S'_1$. The empty string is considered lexicographically smaller than any non-empty string.

### 2.3 Classical Full-Text Indexes

The aim of this section is that of introducing the Text Searching Problem and classical full-text indexes that will be intensively referred in the next chapters. The Text Searching Problem is stated as follows.

**Problem 2.1** Given a text $T[1, n]$ and a pattern $P[1, p]$, we wish to answer the following queries:

1. **COUNT($P$)** that returns the number of occurrences (occ) of $P$ in $T$;
2. **LOCATE($P$)** that reports the occ positions in $T$ where $P$ occurs.

In literature are known several algorithms to solve this problem in linear time via a sequential scan of $T$ (Gusfield 1997). Despite the increase in processing speeds, sequential text searching long ago ceased to be a viable alternative for many applications, and indexed text searching has became mandatory. A (full-)text index is a data structure built over a text $T$ which significantly speeds up subsequential searches for arbitrary pattern strings. This speed up came at the cost of additional space consumption (namely, the space required to store the index). Many different indexing data structures have been proposed in the literature, most notably suffix trees and suffix arrays (e.g., see (Aluru and Ko 2008; Gusfield 1997) and references therein).

#### 2.3.1 Suffix Tree

Let $S$ be a sorted set of strings drawn from an alphabet $\Sigma$ such that no string is a proper prefix of any other string. The (compact) trie of $S$ is the rooted ordered tree defined as follows:

- each edge is labeled with a non-empty string;
- the labels of any two edges leaving the same node begin with different symbols;
- all internal nodes, except possibly the root, are branching;
- the tree has $|S|$ leaves;
the \(i\)-th leaf is associated with the \(i\)-th lexicographic smaller string \(S\) of \(S\) and the concatenation of the labels on the path from the root to this leaf is exactly equal to \(S\).

The suffix tree of a text \(T[1, n]\) is the compacted trie, denoted as \(ST(T)\) or simply \(ST\), built on all the \(n\) suffixes of \(T\). We can ensure that no suffix is a proper prefix of another suffix by simply assuming that a special symbol, say $, terminates the text \(T\). The symbol $ does not appear anywhere else in \(T\) and is lexicographically smaller than any other symbol in \(\Sigma\). This constraint immediately implies that each suffix of \(T\) has its own unique leaf in the suffix tree, since any two suffixes of \(T\) will eventually follow separate branches in the tree.

For a given edge, the edge label is simply the substring in \(T\) corresponding to the edge. For edge between nodes \(u\) and \(v\) in \(ST\), the edge label (denoted \(\text{label}(u, v)\)) is always a non-empty substring of \(T\). For a given node \(u\) in the suffix tree, its path label (denoted \(\text{pathlabel}(u)\)) is defined as the concatenation of edge labels on the path from the root to \(u\). The string depth of node \(u\) is simply \(|\text{pathlabel}(u)|\). For any two suffixes \(T_i\) and \(T_j\), if \(w\) is their longest common prefix, then there exists a node \(u\) in \(ST\) whose path label is equal to \(w\). This node \(u\) is the lowest common ancestor of the two leaves labeled \(i\) and \(j\). In the following, we will use \(\text{Lcp}(T_i, T_j)\) to denote the length of \(w\). Figure 2.1 shows the suffix tree of the example text \(T = \text{abracadabra}\$.

In order to allow a linear space representation of the tree, each edge label is represented by a pair of integers denoting, respectively, the starting and ending positions in the \(T\) of the substring describing the edge label. If the edge label corresponds to a repeat substring, the indices corresponding to any these occurrences could be used. In this way, the suffix tree can be stored in \(\Theta(n \log n)\) bits of space which may be, however, much more than the \(n \log \sigma\) bits needed to represent text itself. We notice that this space bound is not optimal for any \(\sigma\), since there are just \(\sigma^n\) different possible strings while \(n \log n\) bits are suffice to represent \(n^n\) different elements.

An optimal algorithm for building the suffix tree of a text \(T\) is the elegant solution by Farach-Colton (1997) which requires \(O(n)\) time.\(^3\)

In order to solve the Text Searching Problem with the suffix tree we observe that if a pattern \(P\) occurs in \(T\) starting from position \(i\), then \(P\) is a prefix of \(T_i\). This implies that the searching algorithm should identify the highest node \(u\) in \(ST\) such that \(P\) is prefix of \(\text{pathlabel}(u)\). From this observation we can derive the following algorithm for \(\text{Count}(P)\): Start from the root of \(ST\) and follow the path matching symbols of \(P\), until a mismatch occurs or \(P\) is completely matched. In the former case \(P\) does not occur in \(T\). In the latter case, each leaf in the subtree below the matching position gives an occurrence of \(P\). This algorithm counts the \(\text{occ}\) occurrences of any pattern \(P[1, p]\) in time \(O(p \log \sigma)\). These positions can be located by traversing the subtree in time proportional to its size. It is easy to see that this size is \(O(\text{occ})\). The complexity of counting can be reduced to \(O(p)\) by placing a (minimal) perfect hashing function (Hagerup and Tholey 2001) in each node to speed up percolation. This will increase the space just by a constant factor.

\(^3\) Recall the assumption \(\sigma = O(\text{poly}(n))\).
Fig. 2.1 The suffix tree of the example text $T = \text{abracadabra}\$

For further details, the reader can consult the books by Gusfield (1997) and Crochemore and Rytter (2003) that provide a comprehensive treatment of suffix trees, their construction, and their applications.

### 2.3.2 Suffix Array

The *suffix array* (Manber and Myers 1993) is a compact version of the suffix tree, obtained by storing in an array $SA[1, n]$ the starting positions of the suffixes of $T$ listed in lexicographic order. As the suffix tree, this array requires $\Theta(n \log n)$ bits in the worst case. The main practical advantage is given by the fact that the constant hidden in the big-Oh notation is smaller (namely, it is less than 4). $SA$ can be obtained by traversing the leaves of the suffix tree, or it can be built directly in optimal linear time via ad-hoc sorting methods (Gusfield 1997; Puglisi et al. 2007). Since any substring
of $T$ is the prefix of a text suffix, the solution to the Text Searching Problem consists in finding the interval of positions in $SA$ corresponding all text suffixes that start with $P$. Once this interval $SA[sp, ep]$ has been identified, $\text{COUNT}(P)$ is solved by returning the value $\text{occ} = ep - sp + 1$, and $\text{LOCATE}(P)$ is solved by retrieving the entries $SA[sp], SA[sp + 1], \ldots SA[ep]$. The interval $SA[sp, ep]$ can be binary searched in $O(p \log n)$ time, since each binary search step requires to compare up to $p$ symbols of a text suffix and the pattern. This time can be reduced to $O(p + \log n)$ by using an auxiliary array called $LCP$ that doubles the space requirement of the suffix array. The array $LCP[1, n]$ essentially captures information on the heights of the internal nodes in the suffix tree of $T$. It is defined such that its entry $LCP[i]$ is equal to the length of the longest common prefix of the $(i - 1)$-st and $i$-th lexicographically smallest suffixes in $T$ (namely, $LCP[i] = \text{Lcp}(TSA[i-1], TSA[i])$). The $LCP$ array can be computed in linear time starting from the suffix array and, in conjunction with an auxiliary data structure to solve $\text{Range Minimum Queries}$ (RMQ), it suffices to compute the length of longest common prefix between any pair of suffixes of $T$.

We conclude this section by introducing the inverse suffix array. Even if it is not directly related to the Text Searching Problem, it will be often referred in the next chapters. The inverse suffix array, denoted as $SA^{-1}$, is an array of $n$ elements defined as

$$SA^{-1}[i] = j \iff SA[j] = i.$$ 

In other words, since $SA$ is just a permutation of $[n]$, $SA^{-1}$ is defined to be its inverse. Therefore, $SA^{-1}[i] = j$ tells us that suffix $T_i$ is the $j$-th suffix in lexicographic order. We finally notice that $SA$ and $SA^{-1}$ can be easily computed one from the other in linear time. Figure 2.2 shows $SA$, $SA^{-1}$ and $LCP$ arrays for the example text $T = \text{abracadabra}$.

### 2.4 Compression

In this section we present some concepts related to compression that will be very useful in the next pages. In particular, in Sect. 2.4.1 we introduce the notion of empirical entropy which is a well-known measure of the compressibility of a text. This measure will be used in almost all the chapters either to optimize the performance of different compressors (Chap. 3) or to establish upper bounds to performance of compression schemes or compressed indexes (Chaps. 5–7). In Sect. 2.4.2 we introduce the Burrows-Wheeler Transform which is an amazing algorithm for data compression. Even if we will introduce some other compression algorithms in the next chapters, we decided to introduce this tool earlier since it plays a central role in all the thesis.

Fig. 2.2 The table shows $SA$, $SA^{-1}$ and $LCP$ arrays for the example text $T = \text{abracadabra}$.

<table>
<thead>
<tr>
<th>$SA$</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>1</th>
<th>4</th>
<th>6</th>
<th>9</th>
<th>2</th>
<th>5</th>
<th>7</th>
<th>10</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SA^{-1}$</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$LCP$</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
2.4 Compression

2.4.1 Empirical Entropy

The **empirical entropy** resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source), but now it is defined for any finite individual string and can be used to measure the performance of compression algorithms without any assumption on the input distribution [Manzini (2001)].

The 0-th order empirical entropy is currently a well-established measure of compressibility for a single string (Manzini 2001), and it is defined as follows. For each \( c \in \Sigma \), we let \( n_c \) be the number of occurrences of \( c \) in \( T \). The zero-th order empirical entropy of \( T \) is defined as

\[
H_0(T) = \sum_{c \in \Sigma} n_c \log \frac{n}{n_c}
\]

(2.1)

Note that \( |T| H_0(T) \) provides an information-theoretic lower bound to the output size of any compressor that encodes each symbol of \( T \) with a fixed code (Witten et al. 1999). The so-called zero-th order statistical compressors (such as Huffman or Arithmetic (Witten et al. 1999)) achieve an output size which is very close to this bound. However, they require to know information about frequencies of input symbols (called the **model** of the source). Those frequencies can be either known in advance (**static** model) or computed by scanning the input text (**semistatic** model). In both cases the model must be stored in the compressed file to be used by the decompressor. The compressed size achieved by zero-th order compressors over \( T \) is bounded by \(|C_0(T)| \leq \lambda n H_0(T) + f_0(n, \sigma)\) bits, where \( \lambda \) is a positive constant and \( f_0(n, \sigma) \) is a function including the extra costs of encoding the source model and/or other inefficiencies of \( C \). As an example, for Huffman \( f_0(n, \sigma) = \sigma \log \sigma + O(\sigma) + n \) bits and \( \lambda = 1 \), and for Arithmetic \( f_0(n, \sigma) = O(\sigma \log n) \) bits and \( \lambda = 1 \).

In order to evict the cost of the model, we can resort to zero-th order **adaptive** compressors that do not require to know the symbols’ frequencies in advance, since they are computed incrementally during the compression. The zero-th order **adaptive empirical** entropy of \( T \) (Howard and Vitter 1992)) is then defined as

\[
H_0^a(T) = \sum_{c \in \Sigma} \log \frac{n!}{n_c!}
\]

(2.2)

The compress size achieved by zero-th order adaptive compressors over \( T \) is bounded by \(|C_0^a(T)| \leq n H_0^a(T) + f_0(n, \sigma)\) bits where \( f_0(n, \sigma) \) is a function including inefficiencies of \( C \).

Let us now come to more powerful compressors. For any string \( u \) of length \( k \), we denote by \( u_T \) the string of single symbols following the occurrences of \( u \) in \( T \), taken from left to right. For example, if \( T = abracadabra\$ \) and \( u = ab \), we have \( u_T = x r x \) since the two occurrences of \( ab \) in \( T \) are both followed by symbol \( x \). The \( k \)-th order **empirical** entropy of \( T \) is defined as
\[ H_k(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T| H_0(u_T). \] (2.3)

Analogously, the \( k \)-th order adaptive empirical entropy of \( T \) is defined as

\[ H^a_k(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T| H^a_0(u_T). \] (2.4)

We have \( H_k(T) \geq H_{k+1}(T) \) for any \( k \geq 0 \). As usual in data compression (Manzini 2001), the value \( nH_k(T) \) is an information-theoretic lower bound to the output size of any compressor that encodes each symbol of \( T \) with a fixed code that depends on the symbol itself and on the \( k \) immediately preceding symbols. Recently (see e.g. (Kosaraju and Manzini 1999; Manzini 2001; Ferragina et al. 2005a, 2009a; Mäkinen and Navarro 2007; Ferragina et al. 2005b) and references therein) authors have provided upper bounds in terms of \( H_k(|T|) \) for sophisticated data compression algorithms, such as dictionary based (Kosaraju and Manzini 1999), Bwt-based (Manzini 2001; Ferragina et al. 2005a; Kaplan et al. 2007), and PPM-like. These bounds have the form \(|C(T)| \leq \lambda |T| H^*_k(T) + f_k(|T|, \sigma)\), where \( \lambda \) is a positive constant and \( f_k(|T|, \sigma) \) is a function including the extra-cost of encoding the source model and/or other inefficiencies of \( C \). The smaller are \( \lambda \) and \( f_k() \), the better is the compressor \( C \).

2.4.2 Burrows-Wheeler Transform

In Burrows and Wheeler (1994) introduced a new compression algorithm based on a reversible transformation, now called the Burrows-Wheeler Transform (BWT from now on). The BWT transforms the input string \( T \) into a new string that is easier to compress. The BWT of \( T \), hereafter denoted by \( \text{Bwt}(T) \) or simply \( \text{Bwt} \), is built with three basic steps (see Fig. 2.3):

1. append at the end of \( T \) a special symbol $ smaller than any other symbol of \( \Sigma \);
2. form a conceptual matrix \( \mathcal{M}_T \) whose rows are the cyclic rotations of string \( T\$ \) in lexicographic order;
3. construct string \( L \) by taking the last column of the sorted matrix \( \mathcal{M}_T \). We set \( \text{Bwt}(T) = L \).

Every column of \( \mathcal{M}_T \), hence also the transformed string \( L \), is a permutation of \( T\$ \). In particular the first column of \( \mathcal{M}_T \), call it \( F \), is obtained by lexicographically
Fig. 2.3 Example of Burrows-Wheeler transform for the string \( T = \text{abracadabra}\). The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the column \( L = \text{ard}$rcaaaabb\)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>abracadabra$</td>
<td>$</td>
<td>abracadabr a</td>
<td></td>
</tr>
<tr>
<td>bracadabra$a</td>
<td>a</td>
<td>$abracadab r</td>
<td></td>
</tr>
<tr>
<td>racadabra$ab</td>
<td>a</td>
<td>bra$abraca d</td>
<td></td>
</tr>
<tr>
<td>acadabra$abr</td>
<td>a</td>
<td>bracadabra $</td>
<td></td>
</tr>
<tr>
<td>cadabra$abra</td>
<td>a</td>
<td>cadabra$ab r</td>
<td></td>
</tr>
<tr>
<td>dabra$abra$c</td>
<td>a</td>
<td>dabra$abra c</td>
<td></td>
</tr>
<tr>
<td>dabra$abra$ca</td>
<td>b</td>
<td>ra$abra cad a</td>
<td></td>
</tr>
<tr>
<td>abra$abra$cad</td>
<td>b</td>
<td>racabra$ a</td>
<td></td>
</tr>
<tr>
<td>bra$abra$ca</td>
<td>c</td>
<td>adabra$abr</td>
<td></td>
</tr>
<tr>
<td>ra$abra$cadab</td>
<td>d</td>
<td>abra$abra c</td>
<td></td>
</tr>
<tr>
<td>a$abra$cadabr</td>
<td>r</td>
<td>a$abra c ada b</td>
<td></td>
</tr>
<tr>
<td>$abra$cadabr</td>
<td>r</td>
<td>acadabra$ a b</td>
<td></td>
</tr>
</tbody>
</table>

sorting the symbols of \( T$ (or, equally, the symbols of \( L \)). Note that the sorting of the rows of \( \mathcal{M}_T \) is essentially equal to the sorting of the suffixes of \( T \), because of the presence of the special symbol $. This shows that: (1) symbols following the same substring (context) in \( T \) are grouped together in \( L \), and thus give raise to clusters of nearly identical symbols; (2) there is an obvious relation between \( \mathcal{M}_T \) and \( SA \). Property 1 is the key for devising modern data compressors (see e.g. (Manzini 2001)), Property 2 is crucial for designing compressed indexes (see e.g. (Navarro and Mäkinen 2007)) and, additionally, suggests a way to compute the Bwt through the construction of the suffix array of \( T \): \( L[0] = T[n] \) and, for any \( 1 \leq i \leq n \), set \( L[i] = T[SA[i] - 1] \).

Burrows and Wheeler (1994) devised two properties for the invertibility of the Bwt:

(a) Since the rows in \( \mathcal{M}_T \) are cyclically rotated, \( L[i] \) precedes \( F[i] \) in the original string \( T \).

(b) For any \( c \in \Sigma \), the \( \ell \)-th occurrence of \( c \) in \( F \) and the \( \ell \)-th occurrence of \( c \) in \( L \) correspond to the same character of the string \( T \).

As a result, the original text \( T \) can be obtained backwards from \( L \) by resorting to function \( LF \) (also called Last-to-First column mapping or LF-mapping) that maps row indexes to row indexes, and is defined as:

\[ LF(i) = C[L[i]] + \text{RANK}_{L[i]}(L, i), \]

where \( C[L[i]] \) counts the number of occurrences in \( T \) of symbols smaller than \( L[i] \) and \( \text{RANK}_{L[i]}(L, i) \) is a function that returns the number of times symbol \( L[i] \) occurs
in the prefix $L[1 : i]$. We talk about LF-mapping because the symbol $c = L[i]$ is located in the first column of $\mathcal{M}_T$ at position $LF(i)$. The LF-mapping allows one to navigate $T$ backwards: if $T[k] = L[i]$, then $T[k - 1] = L[LF(i)]$ because row $LF(i)$ of $\mathcal{M}_T$ starts with $T[k]$ and thus ends with $T[k - 1]$. In this way, we can reconstruct $T$ backwards by starting at the first row, equal to $\$T$, and repeatedly applying $LF$ for $n$ steps. As an example, see Fig. 2.3 in which the 3rd $a$ in $L$ lies onto the row which starts with $\text{bracadabra}$ and, correctly, the 3rd $a$ in $F$ lies onto the row which starts with $\text{abracadabra}$. That symbol $a$ is $T[1]$. 
Compressed Data Structures for Strings
On Searching and Extracting Strings from Compressed Textual Data
Venturini, R.
2014, XIV, 118 p. 18 illus., Hardcover
ISBN: 978-94-6239-032-4
A product of Atlantis Press