1
Introduction

1.1 Why this book? Our aim and focus

String matching can be understood as the problem of finding a pattern with some property within a given sequence of symbols. The simplest case is that of finding a given string inside the sequence.

This is one of the oldest and most pervasive problems in computer science. Applications requiring some form of string matching can be found virtually everywhere. However, recent years have witnessed a dramatic increase in interest in string matching problems, especially within the rapidly growing communities of information retrieval and computational biology.

Not only are these communities facing a drastic increase in the text sizes they have to manage, but they are demanding more and more sophisticated searches. The patterns of interest are not just simple strings but also include wild cards, gaps, and regular expressions. The definition of a match may also permit slight differences between the pattern and its occurrence in the text. This is called "approximate matching" and is especially interesting in text retrieval and computational biology.

The problems arising in this field can be addressed from different viewpoints. In particular, string matching is well known for being amenable to approaches that range from the extremely theoretical to the extremely practical. The theoretical solutions have given rise to important algorithmic achievements, but they are rarely useful in practice: A well-known fact in the community is that simpler ideas work better in practice. Two typical examples are the famous Knuth-Morris-Pratt algorithm, which in practice is twice as slow as the brute force approach, and the well-known Boyer-Moore family, whose most successful members in practice are highly simplified variants of the original proposal.

https://doi.org/10.1017/CBO9781316135228.001 Published online by Cambridge University Press

It is hard, however, to find the simpler ideas in the literature. In most current books on text algorithms, the string matching part covers only the classic theoretical algorithms. There are three reasons for that.

First, the practical algorithms are quite recent, the oldest one being just a decade old. Some recent developments are too new to appear in the established literature or in books. These algorithms are usually based on new techniques such as bit-parallelism, which has appeared with the recent generation of computers.

The second reason is that in this area the theoretical achievements are dissociated from the practical advantages. The algorithmic community is interested in theoretically appealing algorithms, that is, those achieving the best complexities and involving complicated algorithmic concepts. The development community focuses solely on algorithms known to be fast in practice. Neither community pays much attention to what the other does. Only in the last few years have new algorithms emerged that combine aspects of both theory and practice (such as BNDM), and the result has been a new trend of fast and robust string matching algorithms. These new algorithms have also not yet found a place in the established literature.

Finally, the search for extended patterns, of much interest nowadays, is largely unrepresented in the established literature. There are no books dealing with such new search problems as multiple or approximate pattern matching.

These reasons make it extremely difficult to find the correct algorithm if one is not in the field: The right algorithms exist, but only an expert can find and recognize them. Consider the case of software practitioners, computational biologists, researchers, or students who are not directly involved in the field and are faced with a text searching problem. They are forced to dig into dozens of articles, most of them of theoretical value but extremely complicated to implement. Finally, they get lost in an ocean of choices, without the background necessary to decide which is better. The typical outcomes of this situation are (*a*) they decide to implement the simplest approach, which, when available, yields extremely poor performance and affects the overall quality of development; and (*b*) they make a (normally unfortunate) choice and invest a lot of work in implementing it, only to obtain a result that in practice is as bad as a naive approach or even worse.

The aim of our book is to present, for a large class of patterns (strings, sets of strings, extended strings, and regular expressions) the existing exact and approximate search approaches, and to present in depth the most practical algorithms. By "practical" we mean that they are efficient in practice and that a normal programmer can implement them in a few hours. Fortunately,

these criteria normally coincide in string matching. We focus on *on-line* searching, which means that we do not build data structures on the text. Indexed searching, although based on on-line searching, would deserve a complete volume by itself.

This book is intended to be of use to a large audience. Computer scientists will find everything needed to understand and implement the fastest search algorithms. If they want to go further in studying text algorithms, we give precise links and research references (books, proceedings, articles) to many related problems. Computational biologists will be able to enter in depth in the pattern matching field and find directly the most simple and efficient algorithms for their sequence searches.

We have implemented and experimented with all the algorithms presented in this book. Moreover, some are ours. We give experimental maps whenever possible to help the reader see at a glance the most appropriate algorithms for a particular application.

This book is not a complete survey on text algorithms. This field is too large for a single book. We prefer to focus on a precise topic and present it in detail. We give a list of related recent books in Section 7.2.

1.2 Overview

Chapter 2: String matching

A *string* is a sequence of characters over a finite alphabet Σ. For instance, `ATCTAGAGA` is a string over $\Sigma = \{A, C, G, T\}$. The string matching problem is to find all the occurrences of a string p, called the pattern, in a large string T on the same alphabet, called the text. Given strings x, y, and z, we say that x is a prefix of xy, a suffix of yx, and a factor of yxz.

We present string matching algorithms according to three general approaches, depending on the way the pattern is searched for in the text.

The first approach consists in reading all the characters in the text one after the other and at each step updating some variables so as to identify a possible occurrence. The **Knuth-Morris-Pratt** algorithm is of this kind, as is the faster **Shift-Or**, which is extensible to more complicated patterns.

The second approach consists in searching for the string p in a window that slides along the text T. For every position of this window, we search backwards for a suffix of the window that matches a suffix of p. The **Boyer-Moore** algorithm uses this approach, but it is generally slower than one of its simplifications, **Horspool**. And when it is not, it is slower than other algorithms of other approaches.

https://doi.org/10.1017/CBO9781316135228.001 Published online by Cambridge University Press

The third approach is more recent and leads to the most efficient algorithms in practice for long enough p. As with the second approach, the search is done backward in a window, but this time we search for the longest suffix of the window that is also a factor of p. The first algorithm using this approach was **BDM**, which, when p is short enough, leads to the simpler and more efficient **BNDM**. For longer patterns, a new algorithm, **BOM**, is the fastest.

We give an experimental map to easily choose the fastest algorithm according to the length of the pattern and the size of the alphabet.

The three approaches represent a general framework in which the most efficient algorithms fit. There exist other algorithms, for instance, those based on hashing, but they are not efficient enough. We give references to these algorithms in the last section.

Chapter 3: Multiple string matching

A set of strings $P = \{p^1, p^2, \ldots, p^r\}$ can be searched for in the same manner as a single string, reading the text once. Many search algorithms for searching a single string have been extended to search a set, with more or less success. This chapter is a survey of the most efficient algorithms. Surprisingly, many of them have just been published as technical reports and it is quite difficult for a nonexpert to know of their existence.

All three approaches to search for a single string lead to extensions to a set of strings. The first one leads to the well-known **Aho-Corasick** algorithm and, when the sum of the pattern lengths, $|P|$, is very small, to the **Multiple Shift-And** algorithm.

The second one leads to the famous **Commentz-Walter** algorithm, which is not very efficient in practice. The extension of the **Horspool** algorithm, **Set Horspool**, is efficient for very small sets on large alphabets. A last algorithm, **Wu-Manber**, mixes the suffix search approach with a hashing paradigm and is usually fast in practice.

The third approach permits an extension of **BOM**, the **SBOM** algorithm, which becomes very efficient when the minimum pattern length grows. Similarly to **Shift-Or**, **BNDM** leads to **Multiple BNDM** when $|P|$ is very small.

We give experimental maps that permit choosing which algorithm to use depending on the total pattern size $|P|$, the minimum length, and the size of the alphabet.

Chapter 4: Extended string matching

In many applications, the search pattern is not just a simple sequence of characters. In this chapter we consider several extensions that appear normally in applications and show how to deal with them. All these extensions can be converted into regular expressions (Chapter 5), but simpler and faster particular algorithms exist for the ones we consider here.

The simplest extension is to permit the pattern to be a sequence of classes (sets) of characters instead of just characters. Any text character in the class will match that pattern position. It is also possible for the classes to appear in the text, not only in the pattern.

A second extension is bounded length gaps: Some pattern positions are designated to match any text sequence whose length is between specified minimum and maximum values. This is of interest in computational biology applications, for example, to search for PROSITE patterns.

A third extension is optional and repeatable characters. An optional pattern character may or may not appear in its text occurrence, while a repeatable character may appear one or more times.

Problems arising from these three extensions and combinations thereof can be solved by adapting **Shift-Or** or **BNDM**. Both algorithms involve bit-parallelism to simulate a nondeterministic automaton that finds all the pattern occurrences (see Section 1.3). In this case we have more complex automata, and the core of the problem is finding a way to simulate them. Extending **Shift-Or** leads to an algorithm unable to skip text characters but whose efficiency is unaffected by the complexity of the pattern. Extending **BNDM**, on the other hand, is normally faster, but the efficiency is affected by the minimum length of an occurrence, the alphabet size, and the sizes of the classes and gaps. No classical algorithm can be extended so easily and obtain the same efficiency.

Finally, we show that a small set of short strings can be searched for using a similar approach, and give references to other theoretical algorithms that search specific kinds of extended strings.

Chapter 5: Regular expression matching

Regular expressions give an extremely powerful way to express a set of search patterns, containing all the previous types of problems we have considered so far. A regular expression specifies simple strings and concatenations, unions, and repetitions of other subexpressions. The algorithms addressing them are more complex and should be used only when the problem cannot be expressed as a simpler one.

Searching for a regular expression is a multistage process. First, we need to parse it to obtain a more workable tree representation. We show at the end of Chapter 5 how to do this. We then use the tree representation throughout the chapter.

The second stage is to build a nondeterministic finite automaton (NFA) from the pattern. The NFA is a state machine which has some states active that change as we read text characters, recognizing occurrences when states designated as "final" are reached. There are two interesting ways to obtain an NFA from a regular expression. Thompson's algorithm obtains an NFA whose number of transitions is proportional to the length of the regular expression and which satisfies some regularity properties that are of interest. Glushkov's algorithm produces an NFA that has the minimum number of states and other interesting regularities.

The NFA can be used directly for searching (we call this algorithm **NFA-Thompson**), but this is slow because many states can be active at any time. It can also be converted into a deterministic finite automaton (DFA), which has only one active state at a time. The DFA is appealing for text searching and is used in one of the most classical algorithms for regular expression searching. We call this algorithm **DFAClassical**. Its main problem is that the size of the DFA can be exponential on that of the NFA, which makes the approach workable only for small patterns. On longer patterns, a hybrid approach that we dub **DFAModules** builds an NFA of small DFAs and retains a reasonable efficiency.

Another trend is to simulate the NFAs using bit-parallelism instead of converting them to DFAs. We present two relatively new approaches, **BP-Thompson** and **BPGlushkov**, which are based on simulating the respective NFAs using their specific properties. We show that **BPGlushkov** should always be preferred over **BPThompson**.

A third approach, also novel, permits skipping text characters. The algorithm **MultiStringRE** computes the minimum length ℓmin of an occurrence of the regular expression and computes all the prefixes (of that length) of all the occurrences. It then conducts a multistring search (Chapter 2) for all those strings. When one such prefix is found, it tries to complete the occurrence. An extension of it, **MultiFactRE**, selects a set of strings of length ℓmin such that some of these strings must appear inside any occurrence (the set of prefixes is just one option). Finally, **RegularBNDM** extends **BNDM** by simulating Glushkov's NFA.

Choosing the best algorithm is a complex choice that depends on the structure of the regular expression. We give simple criteria based on properties of the pattern to decide which algorithm to use.

Chapter 6: Approximate matching

Approximate matching is the problem of finding the occurrences of a pattern in a text where the pattern and the occurrence may have a limited number of differences. This is becoming more and more important in problems such as recovering from typing or spelling errors in information retrieval, from sequence alterations or measurement errors in computational biology, or from transmission errors in signal processing, to name a few.

Approximate matching is modeled using a distance function that tells how similar two strings are. We are given the pattern and a threshold k, which is the maximum allowed distance between the pattern and its occurrences. In this chapter we concentrate on the Levenshtein (or edit) distance, which is the minimum number of character insertions, deletions, and substitutions needed to make both strings equal. Many applications use variants of this distance.

We divide the existing algorithms into four types. The first is based on dynamic programming. This is the oldest approach and still the most flexible one to deal with distances other than edit distance. However, algorithms of this kind are not among the most efficient.

The second type of algorithm converts the problem into the output of an NFA search, which is built as a function of the pattern and k, and then makes the automaton deterministic. The resulting algorithms behave reasonably well with short patterns, but not as fast as newer techniques.

Bit-parallelism is the third approach, and it yields many of the most successful results. The algorithms **BPR** and **BPD** simulate the NFA, while **BPM** simulates the dynamic programming algorithms. **BPM** and **BPD** are the most efficient of the class, but **BPR** is more flexible and can be adapted to more complex patterns.

Finally, the fourth approach is filtration. A fast algorithm is used to discard large text areas that cannot contain a match, and another (nonfiltration) algorithm is used to check the remaining text areas. These algorithms are among the fastest, but their efficiency degrades quickly as k becomes large compared to the pattern length m.

Among the many filtration algorithms, we present the two most efficient ones. **PEX** splits the pattern in $k + 1$ pieces and resorts to multistring searching of them, as at least one must appear unaltered in any occurrence. **ABNDM** is an extension of **BNDM** that simulates the NFA of approximate searching.

We present an experimental map comparing these algorithms. In general, filtration approaches work better for low k/m values. **ABNDM** is best for

small alphabet sizes (such as DNA) while **PEX** is best for larger alphabets (such as proteins or natural language). For larger k/m values, and also to verify the text areas that the filters cannot discard, the best algorithms are the bit-parallel ones.

There are some developments for approximate searching of other types of patterns. For multiple pattern matching with errors, the main algorithms are **MultiHash**, which works only for $k = 1$ but is efficient even when the number of patterns is large; **MultiPEX**, which takes $k+1$ strings from each pattern and is the most efficient choice for low k/m values; and **MultiBP**, which superimposes the NFAs of all the patterns and uses the result as a filter, this being the best choice for intermediate k/m values.

For matching extended strings and regular expressions with errors, there are a few approaches: one based on dynamic programming for regular expressions, one based on an NFA of DFAs permitting errors, and a bit-parallel one based on **BPR**. This last one is the most attractive because of the combination of simplicity and efficiency it offers.

1.3 Basic concepts

1.3.1 Bit-parallelism and bit operations

The *bit-parallelism* technique takes advantage of the intrinsic parallelism of the bit operations inside a computer word. That is, we can pack many values in a single word and update them all in a single operation. By taking advantage of bit-parallelism, the number of operations that an algorithm performs can be cut down by a factor of up to w, where w is the number of bits in the computer word. Since in current architectures w is 32 or 64, the speedup is very significant in practice.

Let us introduce some notation to describe bit-parallel algorithms. We use exponentiation to denote bit repetition, for example, $0^3 1 = 0001$. A sequence of bits $b_\ell \dots b_1$ is called a *bit mask* of length ℓ, which is stored somewhere inside the computer word of length w. We use C–like syntax for operations on the bits of computer words, that is, "|" is the bitwise OR, "&" is the bitwise AND, "∧" is the bitwise XOR, "∼" complements all the bits, and "<<" (">>") moves the bits to the left (right) and enters zeros from the right (left), so that, for example, $b_\ell b_{\ell-1} \dots b_2 b_1 << 3 = b_{\ell-3} \dots b_2 b_1 000$.

We can also perform arithmetic operations on the bits, such as addition and subtraction. These operate on the bits as if they formed a number. For instance, $00010110 + 00010010 = 00101000$ and $10010000 - 1 = 10001111$.

We may have to use many computer words to store a given set of values, and in this case the operations described have to be applied over this entire

https://doi.org/10.1017/CBO9781316135228.001 Published online by Cambridge University Press

representation. This is quite trivial for most operations, but the arithmetic ones need some care because we have to consider the propagation effects. For example, imagine that we have to simulate $Z \leftarrow X+Y$ or $Z \leftarrow X-Y$, where $X = X_t \dots X_1$ and $Y = Y_t \dots Y_1$ are each represented using t computer words. Figure 1.1 shows the algorithm for both operations.

```
Add(X = X_t ... X_1, Y = Y_t ... Y_1)
1.    carry ← 0
2.    For i ∈ 1...t Do
3.        Z_i ← X_i + Y_i + carry
4.        If Z_i < X_i OR (Y_i = 1^w AND carry = 1) Then carry ← 1
5.        Else carry ← 0
6.    End of for
7.    Return Z = Z_t ... Z_1

Subtract(X = X_t ... X_1, Y = Y_t ... Y_1)
8.    carry ← 0
9.    For i ∈ 1...t Do
10.       Z_i ← X_i − Y_i − carry
11.       If Z_i > X_i OR (Y_i = 1^w AND carry = 1) Then carry ← 1
12.       Else carry ← 0
13.   End of for
14.   Return Z = Z_t ... Z_1
```

Fig. 1.1. Algorithms for adding and subtracting unsigned numbers stored in multiple machine words. The first word, Z_1, is the least significant. We ignore the final overflow in the operation, but the overflow information is contained in the variable *carry*.

1.3.2 Labeled rooted tree, trie

Most of the data structures presented in this book are based on classical strings and rooted trees. A *rooted tree* is a set of nodes linked together with unidirectional links. The source node of each link is called the *parent* and the target is called a *child*. One special node has no parent; this node is denoted *root*. The rest of the nodes of the tree have exactly one parent each. Nodes with no children are called *leaves*.

For our purpose, it is convenient to attach a *label* to each link, which is normally a character of the alphabet Σ. An example of such a tree is shown in Figure 1.2.

When the labeled rooted tree is associated to a set of strings, it is called a *trie*. A complete presentation of the trie structure is given in Chapter 3.

https://doi.org/10.1017/CBO9781316135228.001 Published online by Cambridge University Press

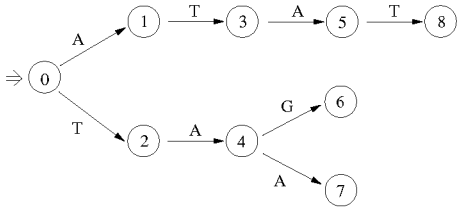


Fig. 1.2. A labeled tree. State 0 is the root. Each node, except the root, has a unique parent.

An algorithm that uses a labeled rooted tree performs computations over the nodes in a specific order. In *prefix order*, the algorithm performs the computation first over a node and then over its children (if any). In *postfix order*, the computation over the node is done after those over the children. For instance, for the tree in Figure 1.2, the nodes we compute over in prefix order are 0, 1, 3, 5, 8, 2, 4, 6, 7, and in postfix order they are 8, 5, 3, 1, 7, 6, 4, 2, 0. The specific order in which sibling nodes are processed is not relevant.

Another frequently used order is the *transversal order*. The *level* of a node is its distance, in terms of the number of intermediate nodes, to the root. In transversal order the nodes are processed in increasing level order. Inside a level, the order has generally no importance, but sometimes we impose one to simplify the algorithms. Two transversal orders are shown in Figure 1.3. The dashed arrows represent the way the nodes are processed.

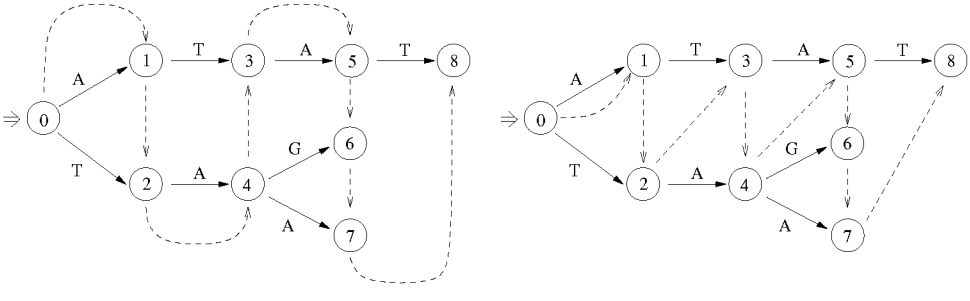


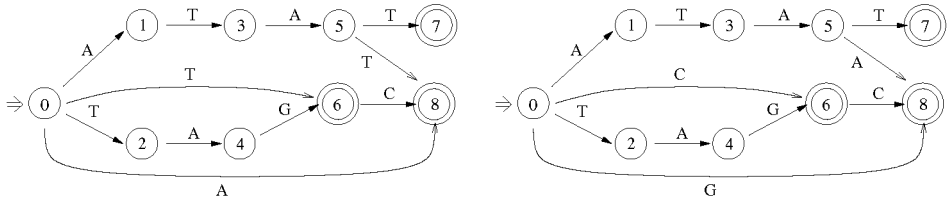
Fig. 1.3. Tree traversals. State 0 is the root. The traversals are shown in dashed arrows.

https://doi.org/10.1017/CBO9781316135228.001 Published online by Cambridge University Press

1.3.3 Automata

The term *automaton* has many meanings in computer science. For our purposes, a finite automaton, which we call simply an automaton, is a finite set of *states* Q, among which one is *initial* (state $I \in Q$) and some are *final* or *terminal* (state set $F \subseteq Q$). Transitions between states are labeled by elements of $\Sigma \cup \{\varepsilon\}$. These are formally defined by a transition function $\mathcal{D}$, which associates to each state $q \in Q$ a set $\{q_1, q_2, \ldots, q_k\}$ of states of Q for each $\alpha \in \Sigma \cup \{\varepsilon\}$. An automaton is then totally defined by $A = (Q, \Sigma, I, F, \mathcal{D})$.

In practice, we distinguish two general types of automata, depending on the form of the transition function. If the function $\mathcal{D}$ is such that there exists a state q associated by a given character α to more than one state, say $\mathcal{D}(q, \alpha) = \{q_1, q_2, \ldots, q_k\}$, $k > 1$, or there is some transition labeled by ε, then the automaton is called a *nondeterministic finite automaton* (NFA), and the transition function $\mathcal{D}$ is denoted by the set of triples $\Delta = \{(q, \alpha, q'),\ q \in Q, \alpha \in \Sigma \cup \{\varepsilon\}, q' \in \mathcal{D}(q, \alpha)\}$. Otherwise, the automaton is called a *deterministic finite automaton* (DFA), and $\mathcal{D}$ is denoted by a partial function $\delta : Q \times \Sigma \to Q$, such that if $\mathcal{D}(q, \alpha) = \{q'\}$, then $\delta(q, \alpha) = q'$. We give examples of both types of automata in Figure 1.4.



(a) Nondeterministic automaton (b) Deterministic automaton

Fig. 1.4. Two automata. In both, the state 0 is initial and the double-circled states are terminal. The left automaton is nondeterministic since from the state 0 by T we reach 2 and 6. The right one is deterministic because for a fixed transition character all the states lead to at most one state.

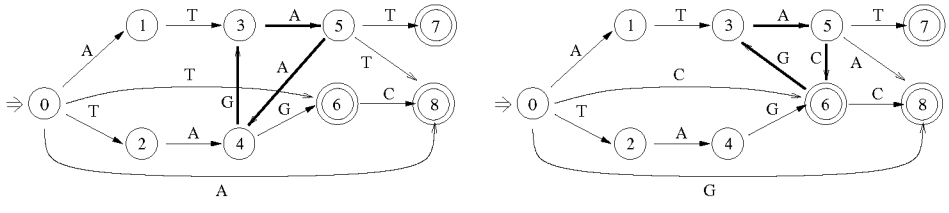
A string is *recognized* by the automaton $A = (Q, \Sigma, I, F, \Delta)$ or $A = (Q, \Sigma, I, F, \delta)$ if it labels a path from an initial to a final state. The *language recognized* by an automaton is the set of strings it recognizes. For instance, the language recognized by the automaton in Figure 1.4 (a) is the set of strings: `A` in state 8, `ATAT` in states 7 and 8, `T` in state 6, `TC` in state 8, `TAG` in state 6, and finally `TAGC` in state 8.

https://doi.org/10.1017/CBO9781316135228.001 Published online by Cambridge University Press

In NFAs, we accept that some transitions are labeled with the empty string ε, and they are called ε-transitions (or empty transitions). This means that we do not have to read a character to go through the transition. If we are at the source state of the ε-transition, we can simply jump to its target state. This can also be seen as reading an empty string. These transitions are generally used to simplify the construction of the NFA, but there always exists an equivalent automaton, recognizing the same language without ε-transitions.

Both in NFAs and DFAs, if a string x labels a path from I to a state s, we say that s is *active* after reading x. DFAs have at most one active state at a time, while NFAs may have many.

The two automata shown in Figure 1.4 have a simple form, in the sense that the transitions do not form cycles. Such automata are called *acyclic*, whether they are deterministic or not. However, we can easily conceive of *cyclic* automata. These automata are useful for regular expression matching. The two automata of Figure 1.5 are cyclic. The language recognized by a cyclic automaton can be infinite. For instance, the automaton of Figure 1.5 (a) recognizes `TAG`, but also `TA·GAA·G`, `TA·GAA·GAA·G`, `TA·GAA·GAA·GAA·G`, and so on.



(a) Nondeterministic cyclic automaton

(b) Deterministic cyclic automaton

Fig. 1.5. Two cyclic automata. In both, the state 0 is initial and the double-circled states are terminal. The cycles are marked in bold.

1.3.4 Complexity notations

We will generally describe the efficiency of our pattern matching algorithms in terms of the number of character comparisons and other basic operations, depending on the size of the pattern, m, and of the text, n. We do not usually give a precise function of n and m, only its growth rate or *complexity order*. The O notation is used to express this idea.

https://doi.org/10.1017/CBO9781316135228.001 Published online by Cambridge University Press

Definition *A function $g(n)$ is said to be $O(f(n))$ if there exist two constants C and n_0 such that $g(n) \leq C \times f(n)$ for all $n > n_0$.*

For instance, $2n^2 + 3n + \log\log n$ is $O(n^2)$, $n\log n + 120n + \log n$ is $O(n\log n)$, and $n^4 + 3n^3 + 15n^2 + n$ is $O(n^4)$. A deeper presentation of complexity notations and their meanings can be found in [Sed88, CLR90].

This notation permits us to compare algorithms of different complexities. For example, if algorithm **A** takes time $O(n)$ and algorithm **B** takes time $O(n\log(m)/m)$, then we know that for large enough m algorithm **B** will be faster than algorithm **A**. We do not know how large is "large enough." Moreover, when both algorithms have the same complexity we do not know which is better.

Sometimes a finer analysis can be done, comparing the exact number of text inspections, table accesses, register accesses, and so on, that are performed by each algorithm. These values are not only more complex to obtain, but they also do not guarantee that we can predict which algorithm is better on a given computer: Not only may the accesses have different costs depending on the architecture, but also caching and pipelining effects complicate any prediction.

The O notation is independent of the architecture, but its predictive power is limited. In many cases we must resort to empirical measures to determine which algorithm is better depending on the instance.

Two complexities are usually studied in the analysis of an algorithm. Its *worst-case* complexity corresponds to the maximum cost over every possible input. Its *average-case* or *expected-case* complexity refers to averaging the cost over all the inputs. This involves assuming a probabilistic distribution of the data. In this book we assume that the pattern and text characters are independent and uniformly distributed over a finite alphabet.