String Algorithms in C

Efficient Text Representation and Search

Thomas Mailund

Apress®

String Algorithms in C

Efficient Text Representation and Search

Thomas Mailund

Apress®

String Algorithms in C: Efficient Text Representation and Search

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-5919-1 ISBN-13 (electronic): 978-1-4842-5920-7
https://doi.org/10.1007/978-1-4842-5920-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Jonathan J Castellon on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484259191. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. He has a background in math and computer science. For the past decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species. He has published *R Data Science Quick Reference*, *The Joys of Hashing*, *Domain-Specific Languages in R, Beginning Data Science in R, Functional Programming in R*, and *Metaprogramming in R*, all from Apress, as well as other books.

About the Technical Reviewer

Jason Whitehorn is an experienced entrepreneur and software developer and has helped many companies automate and enhance their business solutions through data synchronization, SaaS architecture, and machine learning. Jason obtained his Bachelor of Science in Computer Science from Arkansas State University, but he traces his passion for development back many years before then, having first taught himself to program BASIC on his family's computer while still in middle school.

When he's not mentoring and helping his team at work, writing, or pursuing one of his many side projects, Jason enjoys spending time with his wife and four children and living in the Tulsa, Oklahoma region. More information about Jason can be found on his website: https://jason.whitehorn.us.

CHAPTER 1

Introduction

Algorithms operating on strings are fundamental to many computer programs, and in particular searching for one string in another is the core of many algorithms. An example is searching for a word in a text document, where we want to know everywhere it occurs. This search can be exact, meaning that we are looking for the positions where the word occurs verbatim, or approximative, where we allow for some spelling mistakes.

This book will teach you fundamental algorithms and data structures for exact and approximative search. The goal of the book is not to cover the theory behind the material in great detail. However, we will see theoretical considerations where relevant. The purpose of the book is to give you examples of how the algorithms can be implemented. For every algorithm and data structure in the book, I will present working C code and nowhere will I use pseudocode. When I argue for the correctness and running time of algorithms, I do so intentionally informal. I aim at giving you an idea about why the algorithms solve a specific problem in a given time, but I will not mathematically prove so.

You can copy all the algorithms and data structures in this book from the pages, but they are also available in a library on GitHub: `https://github.com/mailund/stralg`. You can download and link against the library or copy snippets of code into your own projects. On GitHub you can also find all the programs I have used for time measurement experiments so you can compare the algorithm's performance on your own machine and in your own runtime environment.

Notation and conventions

Unless otherwise stated, we use x, y, and p to refer to strings and i, j, k, l, and h to denote indices. We use ϵ to denote the empty string. We use a, b, and c for single characters. As in C, we do not distinguish between strings and pointers to a sequence of characters. Since the book is about algorithms in C, the notation we use matches that which is used for strings, pointers, and arrays in C. Arrays and strings are indexed from zero,

T. Mailund, *String Algorithms in C*, https://doi.org/10.1007/978-1-4842-5920-7_1

that is, $A[0]$ is the first value in array A (and $x[0]$ is the first character in string x). The ith character in a string is at index $i - 1$.

When we refer to a substring, we define it using two indices, i and j, $i \leq j$, and we write $x[i, j]$ for the substring. The first index is included and the second is not, that is, $x[i, j] = x[i]x[i + 1] \cdots x[j - 1]$. If a string has length n, then the substring $x[0, n]$ is the full string. If we have a character a and a string x, then ax denotes the string that has a as its first character and is then followed by the string x. We use a^k to denote a sequence of as of length k. The string $a^3 x$ has a as its first three characters and is then followed by x. A substring that starts at index 0, $x[0, i]$, is a *prefix* of the string, and it is a *proper prefix* if it is neither the empty string $x[0, 0] = \epsilon$ nor the full string $x[0, n]$. A substring that ends in n, $x[i, n]$, is a *suffix*, and it is a proper suffix if it is neither the empty string nor the full string. We will sometimes use $x[i,]$ for this suffix.

We use \$ to denote a *sentinel* in a string, that is, it is a character that is not found in the rest of the string. It is typically placed at the end of the string. The zero-terminated C strings have the zero byte as their termination sentinel, and unless otherwise stated, \$ refers to that. All C strings x have a zero sentinel at index n if the string has length n, $x = x[0]x[1] \cdots x[n - 1]0$. For some algorithms, the sentinel is essential; in others, it is not. We will leave it out of the notation when a sentinel isn't needed for an algorithm, but naturally include the sentinel when it is necessary.

Graphical notation

Most data structures and algorithmic ideas are simpler to grasp if we use drawings to capture the structure of strings rather than textual notation. Because of this, I have chosen to provide more figures in this book than you will typically see in a book on algorithms. I hope you will appreciate it. If there is anything you find unclear about an algorithm, I suggest you try to draw key strings yourself and work out the properties you have problems with.

In figures, we represent strings as rectangles. We show indices into a string as arrows pointing to the index in the string; see Figure 1-1. In this notation, we do not distinguish between pointers and indices. If a variable is an index j and it points into x, then what it points to is $x[j]$, naturally. If the variable is a pointer, y, then what it points to is $*y$. Whether we are working with pointers or indices should be clear from the context. It will undoubtedly be clear from the C implementations. We represent substrings by boxes of a different color inside the original string-rectangle. If we specify the indices defining the substring, we include their start and stop index (where the stop index points one after the end of the substring).

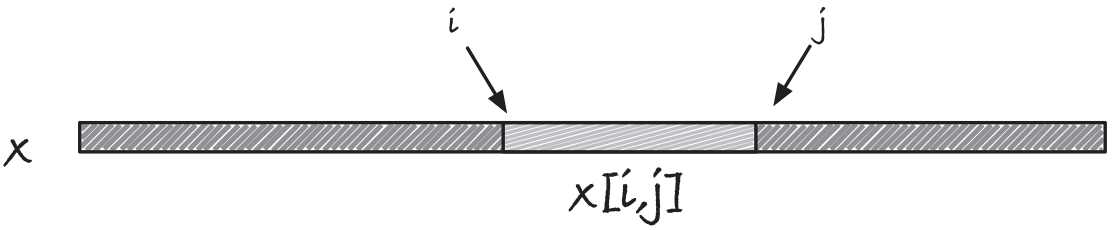

Figure 1-1. *Graphical string notation*

When we compare two strings, we imagine that we align the boxes representing them, so the parts we are comparing are on top of each other. For example, if we compare the character at index *j* in a string *x* with the character at index *i* in another string *p*, then we draw a box representing *x* over a box representing *p*, and we draw pointers for the two indices; see Figure 1-2. Since we are comparing the characters in the two indices, the two pointers are pointing at each other. Conceptually, we imagine that *p* is aligned under *x* starting at position $j - i$.

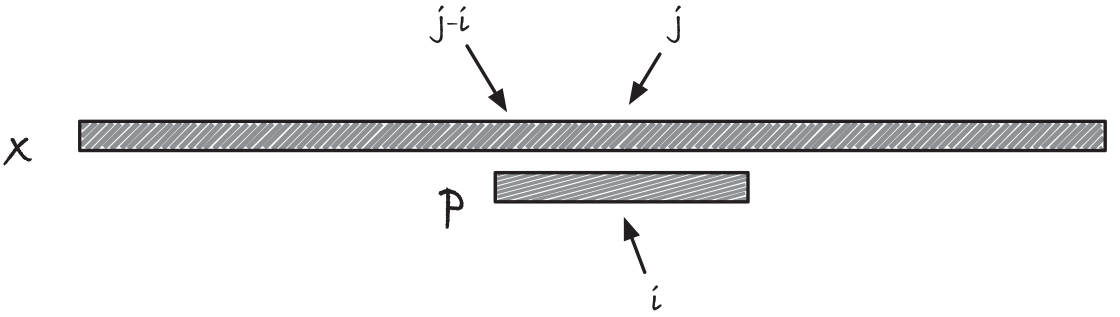

Figure 1-2. *Graphical notation for comparing indices in two different strings*

Code conventions

There is a trade-off between long variables and type names and then the line within a book. In many cases, I have had to use an indentation that you might not be used to. In function prototypes and function definitions, I will generally write with one variable per line, indented under the function return type and name, for example:

```
void compute_z_array(
    const unsigned char *x,
    uint32_t n,
    uint32_t *Z
);
```

```
void compute_reverse_z_array(
    const unsigned char *x,
    uint32_t m,
    uint32_t *Z
);
```

If a return type name is long, I will put it on a separate line:

```
static inline uint32_t
edge_length(struct suffix_tree_node *n) {
    return range_length(n->range);
}
struct suffix_tree *
mccreight_suffix_tree(
    const unsigned char *string
);
struct suffix_tree *
lcp_suffix_tree(
    const unsigned char *string,
    uint32_t *sa,
    uint32_t *lcp
);
struct suffix_tree_node *
st_search(
    struct suffix_tree *st,
    const char *pattern
);
```

I make an exception for functions that take no arguments, that is, have `void` as their argument type.

There are many places where an algorithm needs to use characters to look up in arrays. If you use the conventional C string type, `char *`, then the character can be either signed or unsigned, depending on your compiler, and you have to cast the type to avoid warnings. A couple of places we also have to make assumptions about the alphabet size. Because of this, I use arrays of `uint8_t` with a zero termination sentinel as strings. On practically all platforms, `char` is 8 bits so this type is, for all intents and purposes, C strings. We are just guaranteed that we can use it unsigned and that the alphabet size

is 256. Occasionally it is necessary to cast a `uint8_t *` string to a C string. A direct cast, `(char *)x`, will most likely work unless you are on an exotic platform. If it doesn't, you have to build a `char` buffer and copy characters byte by byte. It has to be a *very* exotic platform if you cannot store 8 bits in a `char`! Because I assume that you can always cast to `char *`, I will use the C library string functions (with a cast) when this is appropriate. It is a small matter to write your own if it is necessary.

I will use `uint32_t` for indices, assuming that strings are short enough that we can index them with 32 bits. You can change it as needed, but I find it a good trade-off between likely lengths of strings and the space I need for data structures. I work in bioinformatics, so hundreds of millions of characters are usually the longest I encounter.

Reporting a sequence of results

In search algorithms, we report each occurrence of a pattern. This sounds straightforward, but there is a design choice in how we report the occurrences. Consider the following algorithm. It is the Boyer-Moore-Horspool (BMH) algorithm that you will see in the next chapter. It takes a string, *x*, and a pattern, *p*, and searches for all occurrences of *p* in *x*. First, it does some preprocessing, and then it searches. This is a general pattern for the algorithms in the next chapter. In the search, when it has found an occurrence of *p*, it reports the position by calling the `REPORT(j)` function.

```
void bmh_search(
    const uint8_t *x,
    const uint8_t *p
) {
    uint32_t n = strlen((char *)x);
    uint32_t m = strlen((char *)p);

    // Preprocessing
    int jump_table[256];

    for (int k = 0; k < 256; k++) {
        jump_table[k] = m;
    }
    for (int k = 0; k < m - 1; k++) {
        jump_table[p[k]] = m - k - 1;
    }
```

```
    // Searching
    for (uint32_t j = 0;
         j < n - m + 1;
         j += jump_table[x[j + m - 1]]) {

        int i = m - 1;
        while (i > 0 && p[i] == x[j + i])
            --i;
        if (i == 0 && p[0] == x[j]) {
            REPORT(j);
        }
    }
}
```

If a global report function is all you need in your program, then this is an excellent solution. Often, however, we need different reporting functions for separate calls to the search function. Or we need the report function to collect data for further processing (and preferably not use global variables). We need some handle to choose different report functions and to provide them with data.

One approach is using callbacks: Provide a report function and data argument to the search function and call the report function with the data when we find an occurrence. In the following implementation, I am assuming we have defined the function type for reporting, `report_function`, and the type for data we can add to it, `report_function_data`, somewhere outside of the search function.

```
void bmh_search_callback(
    const uint8_t *x,
    const uint8_t *p,
    report_function report,
    report_function_data data
) {
    uint32_t n = strlen((char *)x);
    uint32_t = strlen((char *)p);

    // Preprocessing
    uint32_t jump_table[256];
```

```
    for (int k = 0; k < 256; k++) {
        jump_table[k] = m;
    }
    for (int k = 0; k < m - 1; k++) {
        jump_table[p[k]] = m - k - 1;
    }

    // Searching
    for (uint32_t j = 0;
         j < n - m + 1;
         j += jump_table[x[j + m - 1]]) {

        int i = m - 1;
        while (i > 0 && p[i] == x[j + i])
            --i;
        if (i == 0 && p[0] == x[j]) {
            report(j, data);
        }
    }
}
```

Callback functions have their uses, especially to handle events in interactive programs, but also some substantial drawbacks. To use them, you have to split the control flow of your program into different functions which hurts readability. Especially if you need to handle nested loops, for example, iterate over all nodes in a tree and for each node iterate over the leaves in another tree where for each node-leaf pair you find occurrences… (the example here is made up, but there are plenty of real algorithms with nested loops, and we will see some later in the book).

We can get the control flow back to the calling function using the iterator design pattern. We define an iterator structure that holds information about the loop state, and we provide functions for setting it up, progressing to the next point in the loop, and reporting a match and then a function for freeing resources once the iterator is done.

The general pattern for using an iterator looks like this:

```
struct iterator iter;
struct match match;
iter_init(&iter, data);
```

```
    while (next_func(&iter, &match)) {
        // Process occurrence
    }
    iter_dealloc(&iter);
```

The iterator structure contains the loop information. That means it must save the preprocessing data from when we create it and information about how to resume the loop after each time it is suspended. To report occurrences, it takes a "match" structure through which it can inform the caller about where matches occur. The iterator is initialized with data that determines what it should loop over. The loop is handled using a "next" function that returns true if there is another match (and if it does it will have filled out `match`). If there are no more matches, and the loop terminates, then it returns false. The iterator might contain allocated resources, so there should always be a function for freeing those.

In an iterator for the BMH, we would keep the string, pattern, and table we build in the preprocessing.

```
struct bmh_match_iter {
    const uint8_t *x; uint32_t n;
    const uint8_t *p; uint32_t m;
    int jump_table[256];
    uint32_t j;
};
struct match {
    uint32_t pos;
};
```

We put the preprocessing in the iterator initialization function

```
void init_bmh_match_iter(
    struct bmh_match_iter *iter,
    const uint8_t *x, uint32_t n,
    const uint8_t *p, uint32_t m
) {
    // Preprocessing
    iter->j = 0;
    iter->x = x; iter->n = n;
    iter->p = p; iter->m = m;
```

```
    for (int k = 0; k < 256; k++) {
        iter->jump_table[k] = m;
    }
    for (int k = 0; k < m - 1; k++) {
        iter->jump_table[p[k]] = m - k - 1;
    }
}
```

and in the next function we do the search

```
bool next_bmh_match(
    struct bmh_match_iter *iter,
    struct match *match
) {
    const uint8_t *x = iter->x;
    const uint8_t *p = iter->p;
    uint32_t n = iter->n;
    uint32_t m = iter->m;
    int *jump_table = iter->jump_table;

    // Searching
    for (uint32_t j = iter->j;
         j < n - m + 1;
         j += jump_table[x[j + m - 1]]) {

        int i = m - 1;
        while (i > 0 && p[i] == x[j + i]) {
            i--;
        }
        if (i == 0 && p[0] == x[j]) {
            match->pos = j;
            iter->j = j +
                   jump_table[x[j + m - 1]];
            return true;
        }
    }
    return false;
}
```

We set up the loop with information from the iterator and search from there. If we find an occurrence, we store the new loop information in the iterator and the match information in the `match` structure and return true. If we reach the end of the loop, we report false.

We have not allocated any resources when we initialized the iterator, so we do not need to free anything.

```
void dealloc_bmh_match_iter(
    struct bmh_match_iter *iter
) {
    // Nothing to do here
}
```

Since the deallocation function doesn't do anything, we could leave it out. Still, consistency in the use of iterators helps avoid problems. Plus, should we at some point add resources to the iterator, then it is easier to update one function than change all the places in the code that should now call a deallocation function.

Iterators complicate the implementation of algorithms, especially if they are recursive and the iterator needs to keep track of a stack. Still, they greatly simplify the user interface to your algorithms, which makes it worthwhile to spend a little extra time implementing them. In this book, I will use iterators throughout.

CHAPTER 2

Classical algorithms for exact search

We kick the book off by looking at classical algorithms for exact search, that is, finding positions in a string where a pattern string matches precisely. This problem is so fundamental that it received much attention in the very early days of computing, and by now, there are tens if not hundreds of approaches. In this chapter, we see a few classics.

Recall that we use iterators whenever we have an algorithm that loops over results that should be reported. All iterators must be initialized, and the resources they hold must be deallocated when we no longer need the iterator. When we loop, we have a function that returns true when there is something to report and false when the loop is done. The values the iterator reports are put in a structure that we pass along to the function that iterates to the next value to report. For the algorithms in this chapter, we initialize the iterators with the string in which we search, the pattern we search for, and the lengths of the two strings. Iterating over all occurrences of the pattern follows this structure:

```
struct iterator iter;
struct match match;
iter_init(iter, x, strlen(x), p, strlen(p));
while (next_func(&iter, &match)) {
   // Process occurrence
}
iter_dealloc(&iter);
```

When we report an occurrence, we get the position of the match, so the structure the iterator use for reporting is this:

```
struct match {
    uint32_t pos;
};
```


T. Mailund, *String Algorithms in C*, https://doi.org/10.1007/978-1-4842-5920-7_2

Naïve algorithm

The simplest way imaginable for exact search is to iteratively move through the string x, with an index j that conceptually runs the pattern p along x, and at each index start matching the pattern against the string using another index, i (see Figure 2-1). The algorithm has two loops, one that iterates j through x and one that iterates i through p, matching $x[i+j]$ against $p[i]$ along the way. We run the inner loop until we see a mismatch or until we reach the end of the pattern. In the former case, we move p one step forward and try matching again. In the second case, we report an occurrence at position j and then increment the index so we can start matching at the next position. We stop the outer loop when index j is greater than $n - m$. If it is, there isn't room for a match that doesn't run past the end of x.

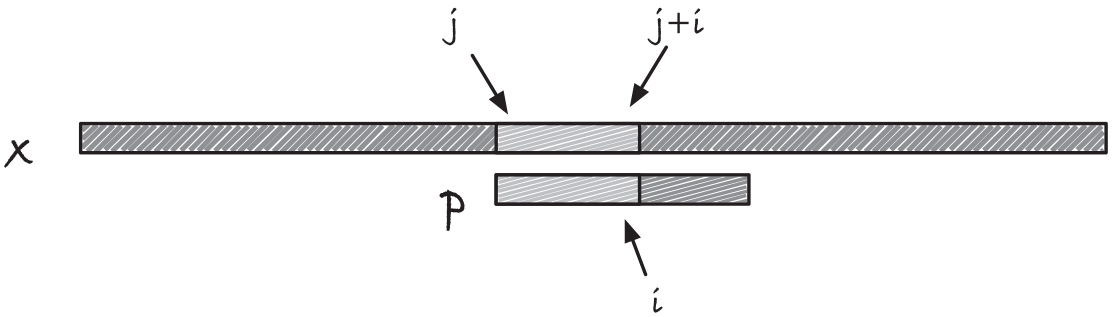

Figure 2-1. *Exact search with the naïve approach*

We terminate the comparison of $x[i+j]$ and $p[i]$ when we see a mismatch, so in the best case, where the first character in p never matches a character in x, the algorithm runs in time $O(n)$ where n is the length of x. In the worst case, we match all the way to the end of p at each position, and in that case, the running time is $O(nm)$ where m is the length of p.

To implement the algorithm using an iterator, the iterator needs to remember the string to search in and the pattern to search for—so we do not need to pass these along each time we increment the iterator with potentials for errors if we use the wrong strings—and we keep track of how far into the string we have searched.

```
struct naive_match_iter {
    const uint8_t *x; uint32_t n;
    const uint8_t *p; uint32_t m;
    uint32_t current_index;
};
```

When we initialize the iterator, we remember the two strings and set the current index to zero—before we start iterating we are at the beginning of the string.

```
void init_naive_match_iter(
    struct naive_match_iter *iter,
    const uint8_t *x, uint32_t n,
    const uint8_t *p, uint32_t m
) {
    iter->x = x; iter->n = n;
    iter->p = p; iter->m = m;
    iter->current_index = 0;
    iter->current_index = 0;
}
```

When we increment the iterator, we follow the algorithm as described earlier except that we start the outer loop at the index saved in the iterator. We search from this index in an outer loop, and at each new index, we try to match the pattern with an inner loop. We break the inner loop if we see a mismatching character, and if the inner loop reaches the end, we have a match and report it. Before we return, we set the iterator index and store the matching position in the `match` structure.

```
bool next_naive_match(
    struct naive_match_iter *iter,
    struct match *match
) {
    uint32_t n = iter->n, m = iter->m;
    const uint8_t *x = iter->x;
    const uint8_t *p = iter->p;

    if (m > n) return false;
    if (m == 0) return false;

    for (uint32_t j = iter->current_index; j <= n - m; j++) {
        uint32_t i = 0;
        while (i < m && x[j+i] == p[i]) {
            i++;
        }
```

```
        if (i == m) {
            iter->current_index = j + 1;
            match->pos = j;
            return true;
        }
    }

    return false;
}
```

The code

```
    if (m > n)  return false;
    if (m == 0) return false;
```

makes sure that it is possible to match the pattern at all and that the pattern isn't empty. This is something we could also test when we initialize the iterator. However, we do not have a way of reporting that we do not have a possible match there, so we put the test in the "next" function.

We do not allocate any resources when we initialize the iterator, so we do not need to do anything when deallocating it either. We still need the deallocator function, however, so we always use the same design pattern when we use iterators. To make sure that if we, at some point in the future, need to free something that we put in an iterator, then all users of the iterator (should) have added code for this.

```
void dealloc_naive_match_iter(
    struct naive_match_iter *iter
) {
    // Nothing to do here...
}
```

Border array and border search

It is possible to get $O(n + m)$ running times for both best and worst case, and several algorithms exist for this. We will see several in the following sections. The first one is based on the so-called *borders* of strings.

Borders and border arrays

A border of a string is any substring that is both a prefix and a suffix of the said string; see Figure 2-2. For example, the string x = *ababa* has borders *aba*, *a*, and the empty string. There is always at least one border per string—the empty string. It is possible to list all borders by brute force. For each index i in x, test if the substrings $x[0, i]$ matches the string $x[n - i, n]$. This approach makes time $O(n)$ per comparison, and we need it for all possible borders which means that we end up with a running time of $O(n^2)$. It is possible to compute the longest border in linear time, as we shall see. The way we compute it shows that sometimes more is less; we will compute more than the length of the longest suffix. What we will compute is the *border array*. This is an array that for each index i holds the length of the longest border of string $x[0, i]$. Consider x = *ababa*. For index 0 we have string *a* which has border *a*, so the first element in the border array is 1. The string *ab* only has the trivial, nonempty border, so the border array value is zero. The next string is *aba* with border *a*, so we get 1 again. Now *abab* has borders *ab*, so the border array holds 2. The full string x = *ababa* with border *aba* so its border array looks like ba = [1, 0, 1, 2, 3].

Figure 2-2. *A string with three borders*

We can make the following observation about borders and border arrays: The longest border of $x[0, i]$ is either the empty string or an extension of a border of $x[0, i - 1]$. If the letter at $x[i]$ is *a*, the border of $x[0, i]$ is some string *y* followed by *a*. The *y* string must be both at the beginning and end of $x[0, i - 1]$ (see Figure 2-3), so it is a border of $x[0, i - 1]$. The longest border for $x[0, i]$ is the longest border of $x[0, i - 1]$ that is followed by *a* (which may be the empty border if the string x begins with *a*) or the empty border if there is no border we can extend with *a*.

Another observation is that if you have two borders to a string, then the shorter of the two is a border of the longer; see Figure 2-4.

The two observations combined gives us an approach to computing the border array. The first string has the empty border as its only border, and after that, we can use the border array up to $i - 1$ to compute the length of the longest border of $x[0, i]$. We start by testing if we can extend the longest border with $x[i]$, and if so, $\text{ba}[i] = \text{ba}[i - 1] + 1$. Otherwise, we look at the second-longest border, which must be the longest border of $x[0, \text{ba}[i - 1]]$. If the character after this border is $x[i]$, then $\text{ba}[i] = \text{ba}[\text{ba}[i - 1]] + 1$. We continue this way until we have found a border we can extend (see Figure 2-5). If we reach the empty border, we have a special case—either we can extend the empty border because $x[0] = x[i]$, in which case $\text{ba}[i] = 1$, or we cannot extend the border because $x[0] \neq x[i]$, in which case $\text{ba}[i] = 0$.

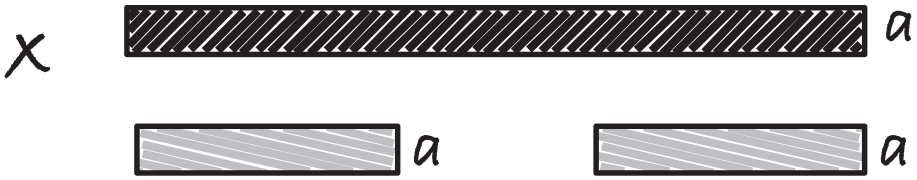

Figure 2-3. *Extending a border*

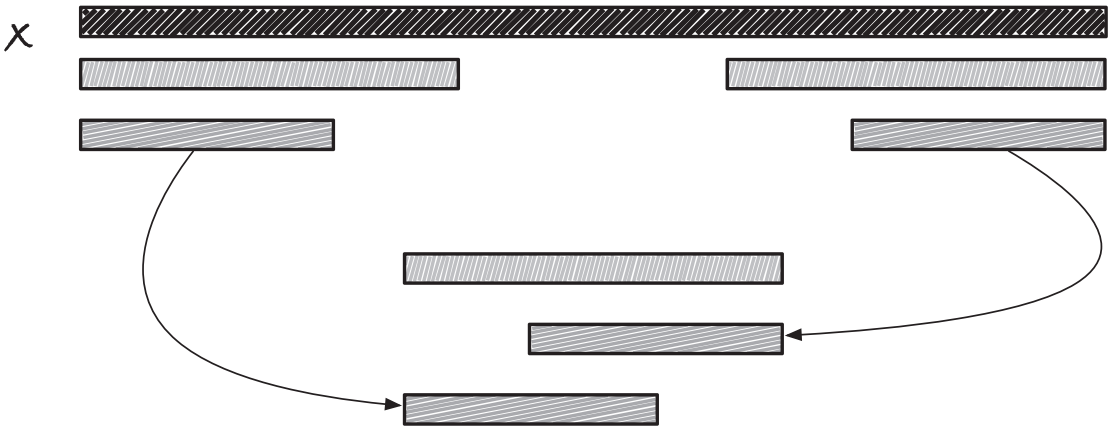

Figure 2-4. *A shorter border is always a border of a longer border*

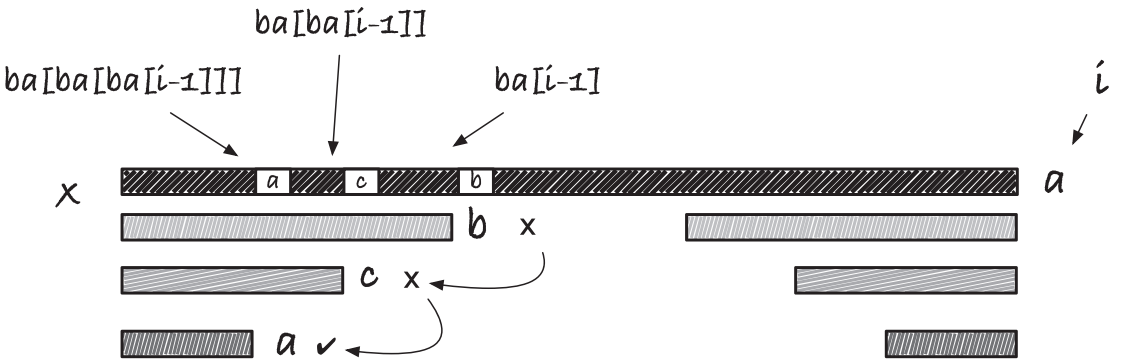

Figure 2-5. *Searching for the longest border we can extend with letter a*

An implementation of the border array construction algorithm can look like this:

```
ba[0] = 0;
for (uint32_t i = 1; i < m; ++i) {
    uint32_t b = ba[i - 1];
    while (b > 0 && x[i] != x[b])
        b = ba[b - 1];
    ba[i] = (x[i] == x[b]) ? b + 1 : 0;
}
```

The running time is m for a string x of length m. It is straightforward to see that the outer loop only runs m iterations but perhaps less easy to see that the inner loop is bounded by m iterations in total. But observe that in the outer loop, we at most increment b by one per iteration. We can assign $b + 1$ to ba[i] in the last statement in the inner loop and then get that value in the first line of the next iteration, but at no other point do we increment a value. In the inner loop, we always decrease b—when we get the border of $b - 1$, we always get a smaller value than b. We don't allow b to go below zero in the inner loop, so the total number of iterations of that loop is bounded by how much the outer loop increase b. That is at most one per iteration, so we can decrement b by at most m, and therefore the total number of iterations of the inner loop is bounded by $O(m)$.

Exact search using borders

The reason we built the border array was to do an exact search, so how does the array help us? Imagine we build a string consisting of the pattern we search for, *p*, followed by the string we search in, *x*, separated by a sentinel, $ character not found elsewhere in the two strings, $y = p\$x$. The sentinel ensures that all borders are less than the length of *p*, *m*, and anywhere we have a border of length *m*, we must have an occurrence of *p* (see Figure 2-6). In the figure, the indices are into the $p\$x$ string and not into *x*, but you can remap this by subtracting $m + 1$. The start index of the match is $i - m + 1$ rather than the more natural $i - m$ because index *i* is at the end of the match and not one past it.

We can construct the string $p\$x$ in linear time and compute the border array—and report occurrences in the process—in linear time, $O(m + n)$. You don't need to create the concatenated string, though. You can build the border array for *p* and use that when computing the border array for *x*. You pretend that *p* is prepended to *x*. When you do this, the sentinel between *p* and *x* is the null-termination sentinel in the C-string *p*.

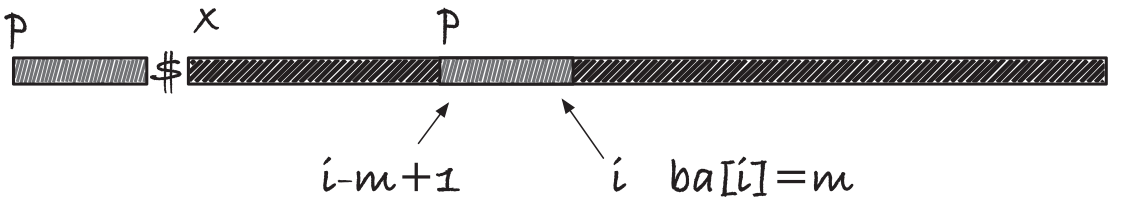

Figure 2-6. *Searching using a border array*

An iterator that searches a string with this algorithm must contain the border array of *p*, the index into *x* we have reached, and the *b* variable from the border array construction algorithm.

```
struct border_match_iter {
    const uint8_t *x; uint32_t n;
    const uint8_t *p; uint32_t m;
    uint32_t *border_array;
    uint32_t i; uint32_t b;
};
```

When we initialize the iterator, we set its index to zero. That, after all, is where we start searching in *x*. We also set the iterator's *b* variable to zero. We imagine that we start the search after a sentinel, so the longest border at the start index for *x* has length zero. We then allocate and compute the border array.

```
void init_border_match_iter(
    struct border_match_iter *iter,
    const uint8_t *x, uint32_t n,
    const uint8_t *p, uint32_t m
) {
    iter->x = x; iter->n = n;
    iter->p = p; iter->m = m;
    iter->i = iter->b = 0;

    uint32_t *ba = malloc(m * sizeof(uint32_t));
    compute_border_array(p, m, ba);
    iter->border_array = ba;
}
```

Since we allocated the border array when we initialized the iterator, we need to free it again when we deallocate it.

```
void dealloc_border_match_iter(
    struct border_match_iter *iter
) {
    free(iter->border_array);
}
```

A third of my implementation for incrementing the following iterator is setting up aliases for the variables in the iterator, so I don't have to write `iter->b` and `iter->m` for variables `b` and `m`, respectively. Other than that, there are the tests for whether it is possible at all to have a match, that we also saw in the previous section, and then there is the border array construction algorithm again, except that we never update an array but instead report when we get a border of length *m*.

```
bool next_border_match(
    struct border_match_iter *iter,
    struct match *match
) {
    const uint8_t *x = iter->x;
    const uint8_t *p = iter->p;
    uint32_t *ba = iter->border_array;
    uint32_t b = iter->b;
    uint32_t m = iter->m;
    uint32_t n = iter->n;

    if (m > n) return false;
    if (m == 0) return false;

    for (uint32_t i = iter->i; i < iter->n; ++i) {
        while (b > 0 && x[i] != p[b])
            b = ba[b - 1];
        b = (x[i] == p[b]) ? b + 1 : 0;
        if (b == m) {
            iter->i = i + 1;
            iter->b = b;
            match->pos = i - m + 1;
            return true;
        }
    }

    return false;
}
```

When we report an occurrence, we naturally set the position we matched in the report structure, and we remember the border and index positions.

Knuth-Morris-Pratt

The Knuth-Morris-Pratt (KMP) algorithm also uses borders to achieve a best- and worst-case running time of $O(n + m)$, but it uses the borders in a slightly different way. Before we get to the algorithm, however, I want to convince you that we can, conceptually, move the pattern p through x in two different ways; see Figure 2-7. We can let j be an index into x and imagine p starting there. When we test if p matches there, we use a pointer into p, i, and test $x[j + i]$ against $p[i]$ for increasing i. To move p to another position in x, we change j, for example, to slide p one position to the right we increment j by one. Alternatively, we can imagine p aligned at position $j - i$ for some index j in x and an index i into p. If we change i, we move $j - i$ so we move p. If, for example, we want to move p one step to the right, we can decrement i by one. To understand how the KMP algorithm works, it is useful to think about moving p in the second way. We will increment the j and i indices when matching characters, but when we get a mismatch, we move p by decrementing i.

The idea in KMP is to move p along x as we would in the naïve algorithm, but move a little faster when we have a mismatch. We use index j to point into x and i to point into p. We match $x[j]$ against $p[i]$ as we scan along x and the pattern is aligned against x at index $j - i$. We can move p's position by modifying either i or j. Consider a place in the algorithm where we have matched $p[0, i]$ against $x[j - i, j]$ and see a mismatch. In the naïve algorithm, we would move p one step to the right and start matching p again at that position. We would set i to zero to start matching from the beginning of p, and we would decrement j to $j - i + 1$ to match at the new position at which we would match p. With KMP we will skip positions where we know that p cannot match, and we use borders to do this.

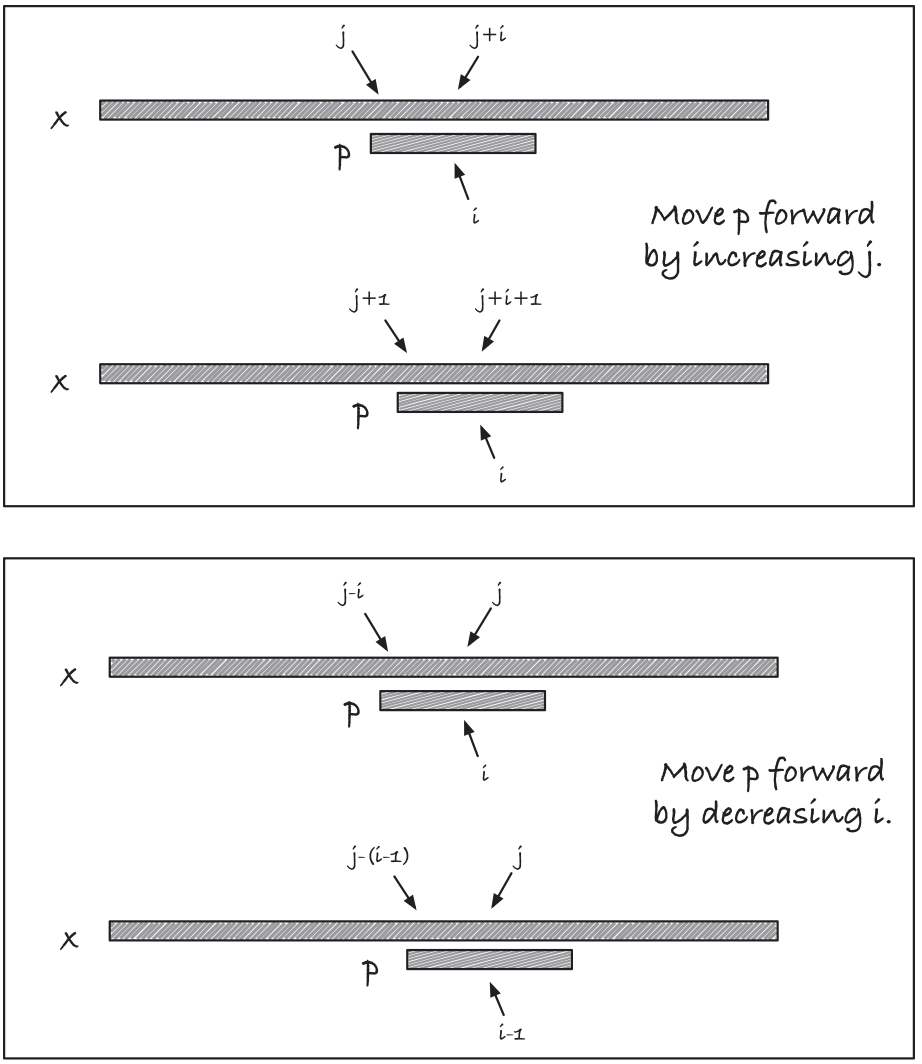

Figure 2-7. *Two ways to conceptually look at matching*

If we have matched p up to index i and then had a mismatch, we know that the only next position at which we could possibly have a match is one where we match a border of $p[0, i-1]$ against a suffix of the string we already matched $x[j-i, j-1]$; see Figure 2-8. It has to be a border of $p[0, i-1]$ and not $p[0, i]$, although that might look like a better choice from the figure. However, we know that $p[0, i]$ doesn't match at the last index, so we need a border of the pattern up to index $i-1$. When we move p, we must be careful not to slide it past possible matches, but if we pick the longest border of $p[0, i-1]$, then this cannot happen. Aligning the longest border moves the pattern the shortest distance