

Behind the Scenes of Ctrl + F

Ajitesh Dasaratha

October 2025

Every one of you has used a ctrl + f search before - What's going on behind the scenes though?

The problem of ctrl + f can be rephrased more formally as: Given a string/document *text*, and a string *pattern*. how do we find the occurrences of *pattern* in *text*?

Let's also define *textlen* as the number of characters in the document, and *patternlen* as the length of the pattern

Approach 1: I just learned nested loops

The most naive approach would be: Loop through every single character in the text, and then ask; do the next *patternlen* characters exactly match the characters of *pattern*? If yes, you've found a match, and if not, then that's not a match. As an example, here is what this algorithm looks like, while looking for the string *abac* in *ababcaababac*

a b a b c a a b a b a c
a b a c

Line up *pattern* with first index of *text*

a b a b c a a b a b a c	a b a b c a a b a b a c
a b a c	a b a c
a b a b c a a b a b a c	a b a b c a a b a b a c
a b a c	a b a c

Check character by character if everything matches

a b a b c a a b a b a c
a b a c

Upon mismatch, move everything one spot to the right and repeat

Note: I am aware that I have only given the intuition for how this is done. If you want implementation details, then [this webpage does a great job of that](#). TLDR: set two pointers, i to mark the index of *text* you are looking at, j to mark the index of *pattern* you are looking at. At any step in the algorithm, the "line-up" is what you get when you align the i^{th} character of *text* with the j^{th} character of *pattern*. You move *pattern* ahead/behind by increasing/decreasing i/j

In the worst case, the runtime of this is $O(\text{textlen} \times \text{patternlen})$, since you have to loop through around textlen alignments, and at each alignment, in the worst case, you need to compare patternlen characters before you find a mismatch. In practice though, the actual runtime is usually closer to $O(\text{textlen})$, since most alignments will mismatch within the first few characters.

Approach 2: I did the LeetCode question

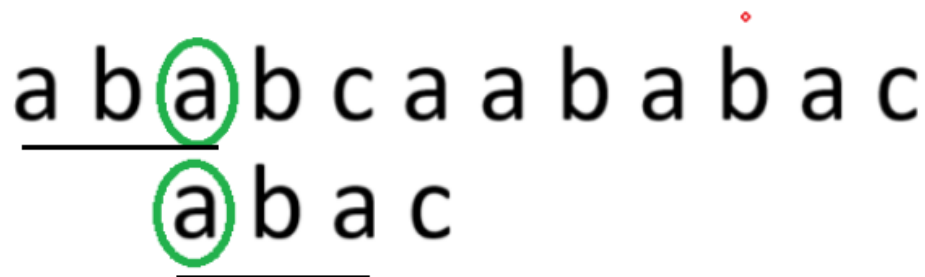
Let's continue with the example from before: looking for the string *abac* in *ababcaababac*. They both start off doing the same thing: compare character by character until a mismatch is found. i.e. this happens the same as it did for before:

a b a b c a a b a b a c	a b a b c a a b a b a c
a b a c	a b a c
a b a b c a a b a b a c	a b a b c a a b a b a c
a b a c	a b a c

Check character by character if everything matches

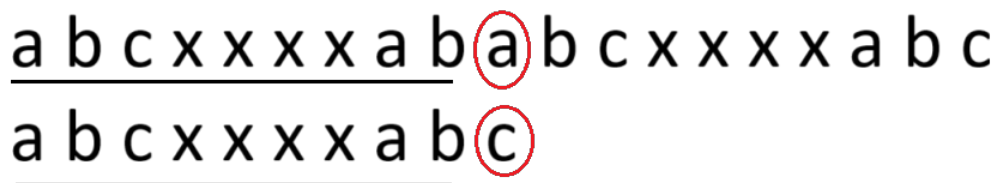
At this point, brute force would just move the alignment ahead by one and restart the process. What's the problem with this? Well, we know that we were able to match the *aba* part of *pattern* to a substring of *text*. If we blindly move ahead by one character, we are not using this information to our advantage at all.

Here's how we use the information to our advantage: of the characters that have matched, what is the largest string that is both a proper prefix (can't be the whole string) and a proper suffix (again, can't be the whole string) of it? For example *a* is both a prefix and suffix of *aba*. I can move *pattern* to the right such that the prefix *a* of *pattern* lines up with the suffix *a* from the characters in *text* that matched up. The previous sentence is probably easier explained by these diagrams:

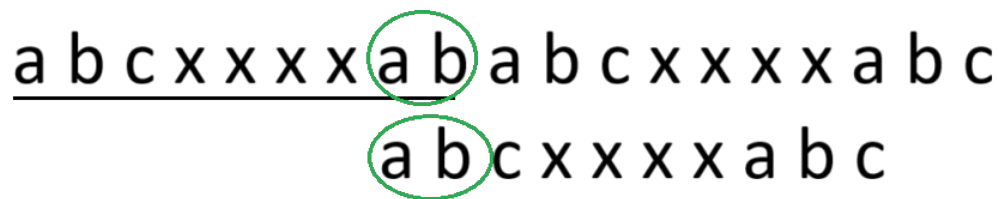


The diagram shows two strings. The top string is "a b a b c a a b a b a c" with a red diamond above the 10th character 'a'. The bottom string is "a b a c". A green circle is drawn around the 3rd character 'a' in the top string, and another green circle is drawn around the 1st character 'a' in the bottom string. Both strings have a horizontal line under the first two characters "a b".

Another example to show how this shifting works: say we were looking for *abcxxxxabc* in *abcxxxxababcxxxxabc*



The diagram shows two strings. The top string is "a b c x x x x a b a b c x x x x a b c". The bottom string is "a b c x x x x a b c". A red circle is drawn around the 8th character 'a' in the top string, and another red circle is drawn around the 8th character 'c' in the bottom string. Both strings have a horizontal line under the first seven characters "a b c x x x x a b".



The diagram shows two strings. The top string is "a b c x x x x a b a b c x x x x a b c". The bottom string is "a b c x x x x a b c". A green circle is drawn around the 8th characters "ab" in the top string, and another green circle is drawn around the 8th characters "ab" in the bottom string. Both strings have a horizontal line under the first seven characters "a b c x x x x a b".

The great thing about this is you now skip a bunch of alignments, as you say in the last example. This limits the worst case to $O(\text{textlen})$. This is because, at every step we are doing one of two things: either we are advancing the character of *text* we are looking at (and we never go backwards in this algorithm), or we are moving the whole pattern forward by 1 or more steps (which can happen a maximum of textlen times). However, in practice this isn't too useful, since the worst case rarely happens anyways, and the average case is still $O(\text{textlen})$, which is the same as the naive approach. It does however help when you search through some special kinds of strings, like DNA sequences which resemble the kind of strings I've used as examples so far.

Again, this is just intuition. If you want implementation details, then this is a famous algorithm called Knuth-Morris-Pratt (KMP) and [this webpage does a great job of showing its implementation](#). TLDR: you make a table as a pre-processing step which tells you: If I've matched the first c characters of *pattern* before finding a mismatch, how much should I shift *pattern* to the right?. You also use pointers i and j to mean the same things as they did in brute force approach.

Approach 3: I now live in the real world

Like I said, KMP only improves the worst case, which rarely happens in practice. Let's look at our third approach, Boyer-Moore-Horspool, that actually helps a lot in practice:

Here is an example (and you'll be relieved that there are actual words in this, rather than some curated string like *abacaadacb*). Say I am looking through one of Dr. Nickvash Kani's fan pages, and want to search for the string "Dr. Connie". Let's say the text starts off with

the sentence "The best professor at UIUC is, without a doubt, Dr. Nickvash Kani, affectionately aliased Dr. Connie by some of his most competent staff members." Here is what the initial line-up looks like (note: I have used the _ to mean whitespace):

```
The_best_p_r_o_f_e_s_s_o_r_a_t_U_I_U_C
Dr._C_o_n_n_i_e
```

I notice that the *e* from *pattern* is a mismatch with *p* in *text*. What's more, I notice that the character *p* doesn't even occur in *pattern*! This means any alignment I try that aligns any character of "Dr. Connie" with the *p* isn't a good alignment, and I should skip it. The next alignment I should try out is:

```
The_best_p_r_o_f_e_s_s_o_r_a_t_U_I_U_C
      Dr._C_o_n_n_i_e
```

That's a huge jump, and funnily enough, if you look at the end, we have a mismatch again ($a \neq e$). The character *a* doesn't show up in "Dr. Connie" either, which means we get *another* huge jump right after that!

I didn't even curate an example to specifically highlight where Boyer-Moore-Horspool is especially good. I just thought it was funny, but it ended up illustrating the point really well.

What about in the case where there is a mismatch, but the character in *text* is also in *pattern*? Well, in that case, you just line up pattern

to match up the previous occurrence in pattern. For example, say you are looking for the word "check" in the sentence is "Let him do his thing - you don't need to check on him every two seconds", here is the initial alignment:

Let_ **h** im _ do _ his _ thing _ ...
c h e c **k**

In this case, the mismatch happens since $h \neq k$. However, the h we found in *text* does exist in pattern though: so any valid line-up would have the h from *text* line up with the one from *pattern*. We can't move the string way past like we did in the previous case, but we can still shift *pattern* so that the h 's are in line - still a good amount of skipping

Let_ **h** im _ do _ his _ thing _ ...
c **h** e c k

One final case before this algorithm is complete, what if you match multiple characters from the back, but then encounter a mismatch in the middle of the word? Say you're looking for the word "nation" in the sentence "Notion is a wonderful productivity tool." Then our misalignment is:

Notion_is_a_wonderful_...
nation

We do *exactly* what we would have done, if the rightmost character had been a mismatch (using the above 2 rules). In this case, the rightmost character matches the *n*'s of "nation" and "notion". We will slide the word "nation", so that the first *n* in "nation" is lined up with the *n* from "notion". So now we have:

Notion_n_is_a_wonderful_...
 nation

In the worst case, the number of comparisons for this algorithm is still $O(\text{textlen} \times \text{patternlen})$. An example of where this might happen is if you try finding the string *baaa* in *aaaaaaaaaaaaabaaa*. Except, that almost never happens in most of the documents that you ctrl + f. In fact, in most cases (and as you saw in the examples above), you skip almost *patternlen* characters. So, you only need to check around one in every *patternlen* characters. This gives you a best-case runtime of $O(\frac{\text{textlen}}{\text{patternlen}})$, which is a *massive* improvement over any of the other algorithms. And this Big-O runtime actually means something, since as we've seen, the average runtime is quite close to the best case.

If you want implementation details, then [this webpage does a great job of that](#). TLDR: You make a table that tells you: if I see a mismatch,

and the character from *text* that was matched to the rightmost character of *pattern* is some character *char*, how much do I get to shift by? And then shift by that much every time there's a mismatch. Like the above algorithms you have pointers *i* and *j* to move through *text* and *pattern*.

A quick bonus lesson from this - theoretical Big-O isn't everything. Most of the times you've used ctrl + f, something similar to Boyer-Moore-Horspool is what retrieved the string for you, even though it has a worse theoretical runtime than KMP in the worst case. What matters in real life is how fast an algorithm works for the cases you usually see, which sometimes leads you to pick a "worse" algorithm than what makes sense theoretically.