Cambridge University Press
978-1-107-16127-6 — Basic Concepts in Data Structures
Shmuel Tomi Klein
Excerpt
More Information

1

Why Data Structures? A Motivating Example

To begin the study of data structures, I demonstrate the usefulness of even quite simple structures by working through a detailed motivating example. We shall afterward come back to the basics and build up our body of knowledge incrementally.

The algorithm presented in this introduction is due to R. S. Boyer and J S. Moore and solves the *string matching problem* in a surprisingly efficient way. The techniques, although sophisticated, do not require any advanced mathematical tools for their understanding. It is precisely because of this simplicity that the algorithm is a good example of the usefulness of *data structures*, even the simplest ones. In fact, all that is needed to make the algorithm work are two small arrays storing integers.

There are two sorts of algorithms that, when first encountered, inspire both perplexity and admiration. The first is an algorithm so complicated that one can hardly imagine how its inventors came up with the idea, triggering a reaction of the kind, "How could they think of that?" The other possibility is just the opposite – some flash of ingeniousity that gives an utterly simple solution, leaving us with the question, "How didn't I think of that?" The Boyer–Moore algorithm is of this second kind.

We encounter on a daily basis instances of the string matching problem, defined generically as follows: given a text $T = T[1]T[2]\cdots T[n]$ of length n characters and a string $S = S[1]S[2]\cdots S[m]$ of length m, find the (first, or all) location(s) of S in T, if one appears there at all. In the example of Figure 1.1, the string $S =$ `TRYME` is indeed found in T, starting at position 22.

To solve the problem, we imagine that the string is aligned underneath the text, starting with both text and string left justified. One can then compare corresponding characters, until a mismatch is found, which enables us to move the string forward to a new potential matching position. We call this the *naive* approach. It should be emphasized that our discourse of moving the pattern

T : `FRIENDS, ROMANS, COUNTRYMEN, LEND ME` ⋯

S : `TRYME`

Figure 1.1. Schematic of the string matching problem.

along an imaginary sliding path is just for facilitating understanding. Actually, text and string remain, of course, in the same location in memory during the entire search process, and their moving is simulated by the changing values of pointers. A *pointer* is a special kind of a variable, defined in many programming languages, holding the address of some data item within computer memory. A pointer may often be simulated by a simple integer variable representing an index in an array.

The number of necessary character comparisons is obviously dependent on the location of S in T, if it appears there at all, so to enable a unified discussion, let us assume that we scan the entire text, searching for all occurrences of S. In the worst case (that is, the worst possible choice of both text T and string S), the naive approach requires approximately $n \times m$ comparisons, as can be seen by considering a text of the form $T = \mathtt{AAA} \cdots \mathtt{AB}$ and a string of similar form $S = \mathtt{A} \cdots \mathtt{AB}$, where the length of the string of As is $2n$ in the text T and n in the string S; only after $(n+1)^2$ comparisons will we find out that S occurs once in T, as a suffix.

The truth is that, actually, this simple algorithm is not so bad under more realistic assumptions, and the worst-case behavior of the previous paragraph occurs just for a rather artificial input of the kind shown. On the average, the number of comparisons in the naive approach will be approximately

$$c \cdot n, \tag{1.1}$$

where c is some constant larger than 1 but generally quite close to 1. It is larger than 1, as every character of the text is compared at least once with some character of the string, and some characters are compared more than once.

In 1977, D. Knuth, J. H. Morris, and V. Pratt published an algorithm that inspects every character of the text and the string exactly once, yielding a complexity of $n + m$ rather than $n \times m$ comparisons. The *complexity* of an algorithm is the time or space it requires, as a function of the size of its input. In particular, the Knuth–Morris–Pratt algorithm also yields $c = 1$ in eq. (1.1). We shall not give here the details of this algorithm, simply because in the same year, Boyer and Moore found an even better algorithm, for which $c < 1$! At first sight, this might look impossible, as $c < 1$ means that the number of characters involved in comparisons is less than the length of the text, or in other words, the algorithm does not inspect all the characters. How could this be possible? We shall see that it all derives from clever use of simple data structures.

1.1 Boyer and Moore's Algorithm

A nice feature of the Boyer–Moore algorithm is that its main idea can be expressed in just four words:

Start from the end.

By repeatedly applying these words as a mantra, we will see how they may help to improve the search.

Let us first try to interpret them correctly. It should be clear that the intention is not just to reverse the process and start by aligning the string S at the end of the text T, and then scanning both from right to left. That would be symmetric to the more natural forward scan from left to right, and the expected search time would be the same. We must therefore conclude that it is only for the string that the scanning will *start at the end* and proceed right to left, whereas the text is scanned in the usual way, from left to right, although with the required minor adaptations.

Figure 1.2 depicts the initial positions of the pointers i and j, showing the current indices in text and string, respectively, for an imaginary text T and the name of my university $S =$ `BAR-ILAN` as a running example for the string. The pointer j is set to the end of the string, that is, $j = m$, 8 in our example, but the initial position of the pointer i is quite unusual – it is neither at the leftmost nor at the rightmost character but rather at that indexed m, corresponding to the last character of the string S.

So what do we gain by this curious setting? The first comparison, in the example of Figure 1.2, would be of character `N` in S against a `W` in T, yielding a mismatch. This disqualifies the current position of S, so the string has to be

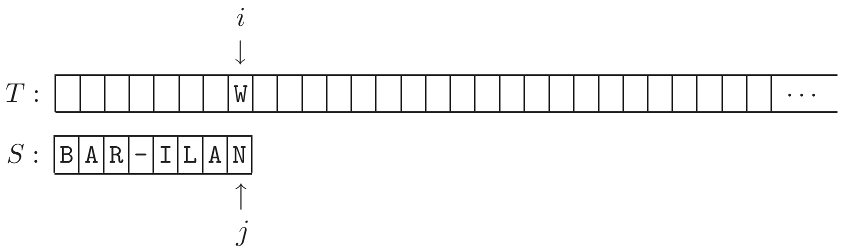

Figure 1.2. Initialization of the Boyer–Moore algorithm.

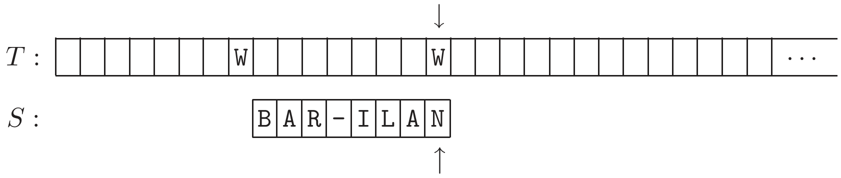

Figure 1.3. After first shift in the Boyer–Moore algorithm.

moved. Does it make sense to move it by 1, 2, . . . , 7 positions to the right? That would still leave the `W` in position 8 of T over one of the characters of S and necessarily lead to some mismatch, because `W` does not appear at all in S. We may therefore move S at once beyond the `W`, that is, to be aligned with position 9. Yet the next comparison, according to our mantra, will again be at the end of S, corresponding now to position $i = 16$ in T, as in Figure 1.3. Note that we have not looked at all at any of the first seven characters in T; nevertheless, we can be sure that no match of S in T has been missed.

The careful reader might feel cheated at this point. The previous paragraph showed an example in which the string could be moved at a step of size m, but this depended critically on the fact that `W` did not appear in S. The natural question is, then, “How do we know that?” An evident approach would be to check it, but this requires m comparisons, exactly counterbalancing the m comparisons we claimed to have saved! To answer these concerns, suppose that in the position of the second comparison, indexed 16, there is again a `W`, as in Figure 1.3. Obviously, there is no need to check again if there is a `W` in S, if we can remember what has already been checked.

1.2 The Bad-Character Heuristic

This leads to the idea of maintaining a Boolean table Δ_0, defining, for each given string S, a function from Σ, the set of all the characters (called also the *alphabet*), to $\{\mathsf{T}, \mathsf{F}\}$: $\Delta_0[x] = \mathsf{T}$, if and only if the character x appears in the string S. The main step of the scanning algorithm, which increases the pointer i into the text, is then

$$\text{if } \Delta_0[T[i]] = \mathsf{F} \qquad i \longleftarrow i + m.$$

The time needed for the construction of Δ_0 is just $m + |\Sigma|$, which is independent of the size n of the text.

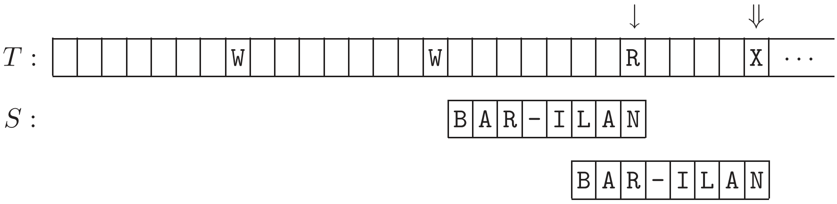

Figure 1.4. Mismatching character appears in S.

And if the mismatching character does appear in S, as in Figure 1.4? Again, one may argue that nothing can be gained from shifting the string one to four positions, so we should, for the given example, shift it by five to align the two Rs, as seen in the lower part of Figure 1.4. The next comparison, however, will be at the end, as usual, indicated by the double arrow. The last ideas may be unified if one redefines the auxiliary table to hold integers rather than Boolean values and to store directly the size of the possible jump of the pointer. More formally, define a table Δ_1, for a given string S, as a function from Σ to the integers, $\Delta_1[x] = r$, if the string can safely be moved by r positions forward in the case of a mismatch at its last character. This reduces the main step of the scanning algorithm to

$$i \longleftarrow i + \Delta_1[T[i]]. \tag{1.2}$$

For our example string $S = \texttt{BAR-ILAN}$, the Δ_1 table is given in Figure 1.5. It can be built by initializing each entry with m, 8 in our example, and then processing the string left to right, setting

$$\textsf{for } j \leftarrow 1 \textsf{ to } m \qquad \Delta_1[S[j]] \leftarrow m - j.$$

This leaves the index for the rightmost appearance, should a character appear more than once in S, like A in our example. The complexity is, as for Δ, $m + |\Sigma|$.

-	A	B	C	D	E	F	G	H	I	J	K	L	M
4	1	7	8	8	8	8	8	8	3	8	8	2	8
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
0	8	8	8	5	8	8	8	8	8	8	8	8	

Figure 1.5. Table Δ_1 for example string $S = \texttt{BAR-ILAN}$ and $\Sigma = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{-}\}$. The upper lines are the characters, and the lower lines are the corresponding Δ_1 values. The entries for characters not appearing in S are in smaller font.

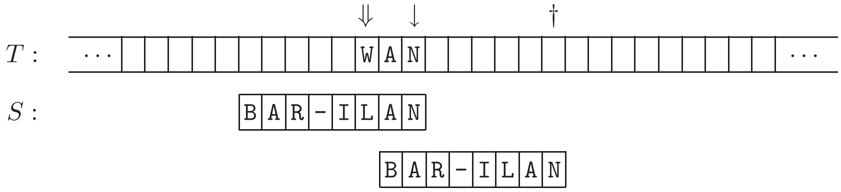

Figure 1.6. Mismatch after a few matches.

So far, only the case of a mismatch at the last character of S has been dealt with. Consider now the possibility of a match, as in Figure 1.6, where the single arrow shows the position of the first comparison for the current location of S, as usual, at the end. In that case, we decrement both i and j and repeat the process. One possibility for exiting this loop is when j reaches zero, that is, the entire string is matching and a success is declared:

$$\text{if } j = 0 \qquad \text{return } \ i + 1.$$

Another possibility is that, after k steps backward, we again encounter a mismatch, as indicated by the double arrow in Figure 1.6, where the mismatch occurs for $k = 2$. The string can then only be shifted beyond the current position, by six positions in our example, and more generally, by $\Delta_1[T[i]] - k$, as in the lower part of Figure 1.6. But we are interested in moving the current position of the pointer i, not in shifting the string, and one has to remember that i has been moved backward by k positions since we started comparing from the end of the string. As the following comparison should again be according to $j = m$, we have to compensate for the decrement by adding k back. The correct updated value of i is therefore

$$i + (\Delta_1[T[i]] - k) + k = i + \Delta_1[T[i]],$$

just as before, so that the assignment in line (1.2) is valid not only for the case of a mismatch at the first trial (at the end of the string) but also for every value of $k > 0$. In our example of Figure 1.6, the current position of i points to W, which does not appear in S, hence i is incremented by $\Delta_1[\texttt{W}] = 8$, bringing us to the position indicated by the dagger sign.

There is possibly a slight complication in the case when the mismatching character of T appears in S to the right of the current position, as would be the case if, in Figure 1.6, there would be an A or N instead of W at the position indicated by the double arrow (there are two As in our example, but recall that the value in Δ_1 refers to the rightmost occurrence of a character in S). This is the case in which $\Delta_1[T[i]] < k$, so to get an alignment, we would actually shift

Cambridge University Press
978-1-107-16127-6 — Basic Concepts in Data Structures
Shmuel Tomi Klein
Excerpt
More Information

the string *backward*, which is useless, because we took care to move the string only over positions for which one could be certain that no match is missed. Incrementing i by k would bring us back to the beginning position of the current iteration; therefore the minimal increment should be at least $k + 1$. The corrected update is therefore

$$i \longleftarrow i + \max(k + 1,\ \Delta_1[T[i]]). \tag{1.3}$$

1.3 The Good-Suffix Heuristic

Actually, this idea of moving the pointer i into the text forward according only to the mismatching character $T[i]$ is already efficient enough to be known as one of the variants of the Boyer–Moore algorithm. But one can do better. Consider the case in which the first mismatch occurs after k steps backward, for $k > 0$, as in Figures 1.6 and 1.7. Instead of concentrating on what went wrong, let us rather insist on the fact that if the first mismatch is at the $k + 1$st trial, this means that there was a success in the k first comparisons. But this implies that when the mismatch occurs, we know what characters appear in the text at positions $i + 1, \ldots, i + k$: these must be the characters of the suffix of length k of S. We can therefore check where there is a reoccurrence of this suffix in the string S, if at all. In Figure 1.7, the suffix `AN` does not appear again in S, so we can move the pattern beyond the position where the present iteration started, as shown in the lower part of Figure 1.7. The next comparison is at the position indicated by the dagger sign, so that i has been incremented from its current position, indicated by the double arrow, by 10. Had we used $\Delta_1[\texttt{I}]$, we could have added only 3 to i.

As previously, we shall not search for another copy of the current suffix during the scanning of the text. There are only m possible suffixes, and one can prepare a table of the possible increments of index i for each of them, independently of the text, in a preprocessing stage. The table Δ_2 will assign a value to each of the possible positions $j \in \{1, \ldots, m\}$ in the string S: $\Delta_2[j]$ will be defined as the number of positions one can move the pointer i in the case where

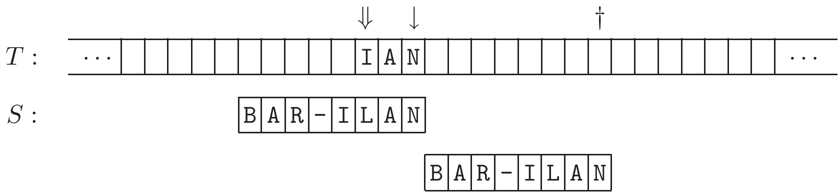

Figure 1.7. The good-suffix heuristic.

j	1	2	3	4	5	6	7	8
$S[j]$	`B`	`A`	`R`	`-`	`I`	`L`	`A`	`N`
shift	8	8	8	8	8	8	8	1
k	7	6	5	4	3	2	1	0
$\Delta_2[j]$	15	14	13	12	11	10	9	1

Figure 1.8. Table Δ_2 for example string $S =$ `BAR-ILAN`.

the first mismatch is at position j, still keeping in mind that we started the comparisons, as usual, from $j = m$.

The increment of i consists of two parts, the first being the number of steps we moved the pointer backward for the current position of the string, the second relating to repositioning the string itself. As we moved already $k = m - j$ steps to the left, i can be increased to point to the position corresponding to the end of the string again, by adding k to i; then we should shift the string S, so as to align the matching suffix with its earlier occurrence in S. $\Delta_2[j]$ will be the sum of k with this shift size.

So which heuristic is better, Δ_1 of the *bad character* or Δ_2 of the *good suffix*? It depends, but because both are correct, we can just choose the maximal increment at each step. The main command would thus become

$$i \longleftarrow i + \max(k + 1, \Delta_1[T[i]], \Delta_2[j]), \tag{1.4}$$

but $\Delta_2[j]$ is k plus some shift, which has to be at least 1. Therefore, the command in line (1.4) is equivalent to

$$i \longleftarrow i + \max(\Delta_1[T[i]], \Delta_2[j]). \tag{1.5}$$

Figure 1.8 depicts the Δ_2 table for our example string. For example, the values in columns 7, 6, and 5 correspond to the first mismatch having occurred with the characters, `A`, `L`, and `I`, which means that there has been a match for `N`, `AN`, and `LAN`, respectively. But none of these suffixes appears again in S, so in all these cases, S may be shifted by the full length of the string, which is 8. Adding the corresponding values of k, 1, 2, and 3, finally gives Δ_2 values of 9, 10, and 11, respectively. Column 8 is a special case, corresponding to a matching suffix that is empty and thus reoccurs everywhere. We can therefore only shift by 1, but in fact it does not matter, as in this case, the Δ_1 value in command (1.5) will be dominant.

The simple form of this table, with increasing values from right to left, is misleading. Let us see what happens if the string changes slightly to $S =$ `BAN-ILAN`. The value in column 6 corresponds to a mismatch with `L` after having

j	1	2	3	4	5	6	7	8
$S[j]$	B	A	N	-	I	L	A	N
shift	8	8	8	8	8	5	5	1
k	7	6	5	4	3	2	1	0
$\Delta_2[j]$	15	14	13	12	11	7	6	1

j	1	2	3	4	5	6	7	8
$S[j]$	B	A	N	-	I	L	A	N
shift	8	8	8	8	8	5	**8**	1
k	7	6	5	4	3	2	1	0
$\Delta_2[j]$	15	14	13	12	11	7	**9**	1

Figure 1.9. Table Δ_2 for example string $S =$ BAN-ILAN.

matched AN. This suffix appears again, starting in position 2 of S, so to align the two occurrences, the string must be moved by five positions. For $j = 7$, the corresponding suffix is of length 1, just N, which seems also to trigger a shift of five positions, like for column 6. For columns $j < 6$, we are looking for LAN or longer suffixes of S, none of which reoccurs in S, thus the string can be shifted by its full length, 8. This yields the table in the upper part of Figure 1.9.

The value in column 7 should, however, be reconsidered. Applying $\Delta_2[7]$ as increment corresponds to a scenario in which there has been a match with N, and a mismatch with the next, preceding, character. We thus know that there is an N in the text, which is preceded by some character that is not A. Therefore, when we look for another occurrence of N, the one found in position 3 does not qualify, because it is also preceded by A; if this lead to a mismatch at the current position, it will again yield a mismatch after the shift. Our strategy can therefore be refined: for a given suffix S' of the string S, we seek its previous occurrence in S, if there is one, but with the additional constraint that this previous occurrence should be preceded by a *different* character than the occurrence at the end of S. For $S' =$ N in our example, there is no such re-occurrence, hence the correct shift of the string is by the full length 8, and not just by 5, which yields the table in the lower part of Figure 1.9. The other entries remain correct. For example, for $j = 6$, we search for another appearance of the suffix AN that is not preceded by L, and indeed, the previous AN is preceded by B, so one may shift the string only by 5.

We are not yet done and there is need for a final slight amendment in certain cases. Consider another small change of the given string to $S =$ LAN-ILAN.

j	1	2	3	4	5	6	7	8
$S[j]$	L	A	N	-	I	L	A	N
shift	8	8	8	8	5	8	8	1
k	7	6	5	4	3	2	1	0
$\Delta_2[j]$	15	14	13	12	8	10	9	1

j	1	2	3	4	5	6	7	8
$S[j]$	L	A	N	-	I	L	A	N
shift	**5**	**5**	**5**	**5**	5	8	8	1
k	7	6	5	4	3	2	1	0
$\Delta_2[j]$	**12**	**11**	**10**	**9**	8	10	9	1

Figure 1.10. Table Δ_2 for example string $S =$ `LAN-ILAN`.

Treating this example in the way we have done earlier would produce the table in the upper part of Figure 1.10. Though the suffixes `N` and `AN` appear earlier, they are preceded by the same characters `A` and `L`, respectively, in both occurrences, and are therefore regarded as if they would not re-appear, yielding a shift of 8. The suffix `LAN`, on the other hand, appears again as prefix of *S*, but is not preceded there by `I`, so we can shift only by 5.

Refer now to Figure 1.11 and suppose the first mismatch is for $j = 4$, comparing the character `W` in the text with the dash character `-` in *S*, after having matched already the suffix `ILAN`. Since this suffix does not re-occur, the upper Δ_2 table of Figure 1.10 suggests to shift by 8, moving the pointer *i* by 12, from the position indicated by the single arrow to that indicated by the double arrow. But this could have resulted in a missed occurrence, as indicated by the brace in the figure.

How could this happen? The answer is that our current string *S* has a special property, namely, that it contains a suffix, `LAN`, that is also a prefix. This allows different occurrences of *S*, or its suffixes, to overlap in the text. One

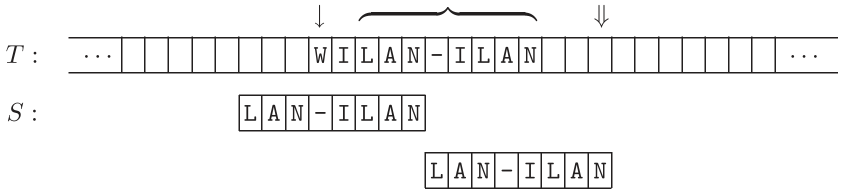

Figure 1.11. Example of a missed occurrence by the upper Δ_2 table of Figure 1.10.