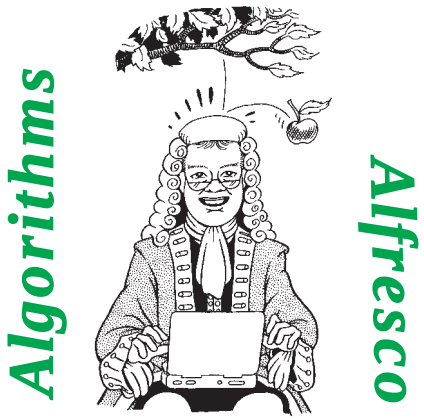
Hunting High And Low

Searching for patterns in ASCII text



by Julian Bucknall

One of the subjects our Editor, Chris Frizelle, and I have been batting around for a while as a topic for *Algorithms Alfresco* is processing text. When I say 'batting around', Chris would propose it as a topic and I'd laugh, and say something along the lines of 'Give me a 500-page book and a 5-figure advance and I'll think about it' and toss it back, and he'd appeal to my vanity, 'Only you could do the subject justice for our Delphi readership' and I recognize it as a not-so-subtle ploy to preen my pride, but I shall give in anyway...

Nevertheless, processing text is a huge subject, so, in the absence of a 5-figure advance (not even in yen, note!) I'll take the easy way out and use several articles to explore the topic. Also, I think it'll turn out to be a series of occasional articles, rather than a block of *N* successive ones, so remember to keep your subscription up to date.

Train Of Thought

What do I mean by processing text? First, let's define text. At a very basic level, text is a bunch of characters. For the present, I'll just talk about single byte character set text (SBCS), rather than text that uses double byte character sets (DBCS) or UNICODE. The text will appear to be a series of words in lines (or paragraphs), delimited by carriage return and linefeed character pairs (or some other sequence, for example single linefeed characters). We'll be looking at text that is available either in a memory block or in a file.

Having defined it, we can think about what we shall want to do with text. I suggest the following (non-exhaustive) list: search for strings or patterns in the text, replace strings in the text with others, edit text (or rather investigate data structures for doing so), look at the buffering of large text files and check out methods for indexing text in files or database memo BLOBs.

Having laid some bare foundations, let's move onto our first topic, the subject of this article: searching for strings in text. The string we search for is known as the 'pattern', so this article will describe a couple of algorithms for searching for patterns in text. We'll limit ourselves to patterns of simple characters instead of metacharacters, as in regular expressions, so we'll be searching for phrases like 'The tongues of mocking wenches are as keen As is the razor's edge invisible' in Shakespeare's *Love's Labour's Lost*.

I think that everyone is familiar with the `Pos` function in the `System` unit. The syntax for `Pos` is:

```
function Pos(SubStr : string;
  S : string) : integer;
```

where `SubStr` is the pattern we are looking for in the text `S` (to put it in terms of this article). The result is 0 if the pattern is not found in the text, or is the position of the first character of the first occurrence of the pattern if it is. `Pos` is written in assembly for speed (and, in fact, there are two versions in Delphi 2 and later: one for short strings and one for long strings), and implements a simple search as described by the pseudo-code in Listing 1.

The `Pos` routine performs the search in two loops. Firstly, it searches for the initial character of the pattern in the text (loop 1), and secondly, for every match it finds, it attempts to compare the pattern's characters to the text at the match position (loop 2). Both the scan for the initial character and the comparison at a match are done by fast assembler `REPx SCASB` instructions. The only other optimization is to recognize that you don't have to scan the whole of the text for the initial character of the pattern. For example, if the text was 50 characters long and the pattern was 49 characters, you'd only have to scan for the initial character of the pattern in the first two positions of the text, since you couldn't get a 49 character string from anywhere else in the text.

`Pos` is extremely good at what it does. For a test, I used a complete version of *Love's Labour's Lost*, read it into a single memory block (125.6Kb) and searched for the words 'keel' (only appears at the very end of the play), 'keen' (appears right at the start), and 'keek' (does not appear at all). I timed the searches over enough iterations to give a reasonably accurate time. The times for 5000 searches were 27.9, 0.1 and 27.4 seconds respectively.

➤ *Listing 1: pseudo-code for the Delphi Pos function.*

```
CharsToCheck := length(S) - length(SubStr) + 1
for I := 1 to CharsToCheck do {loop 1}
  if S[I] = SubStr(1) then
    {loop 2}
    Compare rest of chars in Subtr to S[I+1] onwards
    if equal then
      signal success and exit immediately
endforloop
signal failure
```

At first glance this sounds pretty good, but is there a catch? An obvious one to point out is that my test was very favorable to `Pos`: the initial letter of my three words was k, which I'm sure you'll agree is not that common a letter in English text. What happens is that the search for the initial letter k would not be interrupted all that often. In fact, the most common character in *Love's Labour's Lost* is the space, so I repeated the test prefixing the three words with a space. The results were nearly twice as long: 53.1, 0.3, 53.2 seconds respectively: interrupting the scan for the initial letter costs time.

One big drawback with `Pos` is that it's only good for a case-sensitive search: the pattern has to exist *exactly* in the text, we can't search for 'keel' and expect to get 'Keel', for example. Obviously, we could uppercase both the pattern and the text and then do the search, but sometimes the text can be pretty long. Another problem is that it only finds the *first* occurrence of the pattern in the text: if you want to find the second or subsequent one, you're out of luck, time to recode it or to fiddle around with the text string.

Stay On These Roads

So what, if anything, can we do to improve the time that `Pos` takes? I'm afraid, not a lot. It's written in assembly language, following a simple algorithm, so I'd have to say we need to look further afield. The first stop on our journey is the Boyer-Moore algorithm.

One of the problems with `Pos` is what happens if we almost match the pattern, but don't quite manage. Suppose we are looking for the word 'twenty-two' in the phrase 'twenty and two is twenty-two'. Following the pseudo-code for the `Pos` routine, we first scan for the initial t of the pattern. We hit it straight away in the text. We try and match the rest of the pattern. We match six characters in the pattern (t, w, e, n, t, y) before failing on the space (space does not equal hyphen). We then go back and scan for the initial t again, but we start this time from the w, the second letter of the text. We *could* have started from the letter after the space instead, since there are no spaces in the pattern. If only we'd prepared some 'information' about the pattern we could have been more intelligent about where to start the scan again for the first letter.

This analysis of the pattern is the basis for the Boyer-Moore algorithm. We generate a list of 'skip' values for each letter of our character set and then use these to skip over large tracts of the text. The other oddity is that we scan for the *last* character of the pattern, not the first. Otherwise, the Boyer-Moore method still follows the dual loop method: scan for a character for the first loop, and match the pattern for the second loop. The match loop proceeds in reverse, of course.

It turns out that there are two types of skip we could do. The first is the easiest to understand. Suppose we were looking for 'rat' in 'cats chase rats'. Assuming that we

are using the Boyer-Moore method (I know, I know, I haven't actually told you how it works yet, but bear with me for a little while longer), we are scanning for the final t of the pattern. Suppose we've scanned to the s in the word cats. It doesn't match (duh!), *however* we can say something very definite: the character s does not appear in the pattern. Therefore, we can skip over 3 characters of the text (the length of the pattern) before resuming our scan for the last character. Why? There is no s in the pattern, anywhere, so, if the pattern is to appear in the text, the first place it can appear is after the s, hence we skip 3 characters. Wham! We're really cooking with gas: instead of skipping one character at a time in the scan phase as with `Pos`, we jump over chunks of the text in leaps and bounds. If our pattern string was fairly long, we'd be moving at warp speed through the text.

Before we get too enthusiastic, the preceding argument only applies to characters that don't appear in the pattern. What if, in our scan for the last character of the pattern, we hit an r in the text, say. The letter r *does* appear in our pattern. Can we skip more than one character, this time? The answer is yes. What we need to do is to line up the rightmost r of our pattern (ie, the r in rat) with the r we hit in the text, and then continue the scan for the last character. In effect, we skip two characters instead of the one.

I think you can see what the initial analysis of the pattern entails. It involves the calculation of the skip values for every single character. The majority of the skip values will be equal to the length of the pattern (basically for those characters that do not appear in the pattern). For characters that can be found in the pattern, the skip value will be equal to its distance from the last character (the smallest distance if it appears more than once in the pattern).

So what happens if we *do* match the last character in the pattern in the scan phase? We'll slip into the match loop where we try and match the entire pattern. If we do manage this, then all well and good, we've completed our search. If we don't, there will be some character in the text where we mismatch.

One skip we can do is to pretend that this mismatch occurred at the scan for last character level and advance the same number of characters. In our example, suppose we'd hit on the word cat in the text. We can't match it with our pattern, since the c in the text does not equal the r in the pattern. So we advance the pattern *past* the c (since there is no c in the pattern) and then continue searching for the last character. This does *not* mean taking our previous scanned position and adding three, the length of the pattern. Suppose we'd scanned for the terminating t and matched the t in the word cat at a text position of X. We then try and match our pattern: it's a no go with the c at position X-2. We then skip 3 characters from where we failed to match, which is X+1, and continue the scan for the last character at that point.

➤ *Listing 2: Generate skip values for Boyer-Moore search.*

```
type
  PBoyerMooreSkips = ^TBoyerMooreSkips;
  TBoyerMooreSkips = array [char] of byte;
procedure GenerateSkips(aPattern : string; var S : TBoyerMooreSkips);
var
  i      : integer;
  LenPat : integer;
  Skip   : integer;
begin
  {assumption: 0 < length(aPattern) <= 255}
  LenPat := length(aPattern);
  FillChar(S, sizeof(S), byte(LenPat));
  Skip := pred(LenPat);
  for i := 1 to pred(LenPat) do begin
    S[aPattern[i]] := Skip;
    dec(Skip);
  end;
end;
```

And You Tell Me

Let's be more concrete and see how the entire process works when we search for the pattern 'rat' in the text 'cats chase rats'. The skip values for r, a, t, are 2, 1, 3 respectively. All other characters have skip values of 3. We start as follows:

```
Text:     cats chase rats
Position    ^
Pattern:  rat
```

We have a match on the t, and on the a, but we mismatch on the r in the pattern with the c in the text. We skip by 3 characters (the skip value for c) from the point of the mismatch.

```
Text:     cats chase rats
Position     ^
Pattern:   rat
```

No luck: the t does not match the s, and as s is not in the pattern we advance by 3.

```
Text:     cats chase rats
Position        ^
Pattern:      rat
```

No match again, and as h is not in the pattern we advance by 3.

```
Text:     cats chase rats
Position           ^
Pattern:         rat
```

No match again, and as e is not in the pattern we advance by 3.

```
Text:     cats chase rats
Position             ^
Pattern:           rat
```

Again no match, however a *is* in the pattern and has a skip value of 1.

```
Text:     cats chase rats
Position               ^
Pattern:             rat
```

Bingo, we now match the pattern all the way.

Touchy!

However, before we congratulate ourselves, there is a big problem with this approach. Suppose we

```
function BMPos(aPattern : string; aText : string) : integer;
var
  TextInx, TextInx2 : integer;
  PatInx            : integer;
  PatLen, TextLen   : integer;
  SkipValue, SkipValue2 : integer;
  Skips    : TBoyerMooreSkips;
  LastChar : char;
  Matched  : boolean;
begin
  {quick easy checks}
  PatLen := length(aPattern);
  TextLen := length(aText);
  if (aPattern = '') or (PatLen > TextLen) then begin
    Result := 0;
    Exit;
  end;
  if (PatLen > 255) then
    raise Exception.Create('Pattern is too long');
  {generate the skip values}
  GenerateSkips(aPattern, Skips);
  {start looking for the last character of the pattern}
  LastChar := aPattern[PatLen];
  TextInx := PatLen;
  while TextInx <= TextLen do begin
    {calc skip value, based on current text character}
    SkipValue := Skips[aText[TextInx]];
    {if we don't have a match on the last character, skip}
    if (LastChar <> aText[TextInx]) then
      inc(TextInx, SkipValue)
    {if we do have a match on the last char,
     try matching the rest}
    else begin
      Matched := true;
      Result := TextInx;
      for PatInx := pred(PatLen) downto 1 do begin
        dec(Result);
        if (aPattern[PatInx] <> aText[Result]) then begin
          NewTextInx := Result + Skips[aText[Result]];
          inc(TextInx, SkipValue);
          if (TextInx < NewTextInx) then
            TextInx := NewTextInx;
          Matched := false;
          Break;
        end;
      end;
      if Matched then
        Exit;
    end;
  end;
  Result := 0;
end;
```

➤ *Listing 3: The Boyer-Moore pattern matching routine, BMPos.*

are searching for 'ballon' in a large amount of text which contained the sentence 'The French word for balloon is ballon.' We'd be scanning for the last n in our pattern, and let's say we match it with the final n in balloon (at position X, say).

```
Text:     ..word for balloon is..
Position                  ^
Pattern:              ballon
```

The o in ballon matches the final o in balloon as well. However, there's a mismatch on the first o at X-2: it doesn't match with the final l in ballon. If we apply the trick we just used we find we have a problem: the letter o says to skip one character. If we do this, we find that we'll start the scan for the last character at X-1 as the very next step.

```
Text:     ..word for balloon is..
Position                  ^
Pattern:              ballon
```

It won't match, so we'll skip one character to position X (the o character in the text we tried to match says, skip one), and we're back where we started from. In other words, an infinite loop.

I don't know about you, but infinite loops are a no-no for me. So, we need to revise the algorithm a little bit for this case.

The easiest thing to do is calculate the two new positions, one from the last mismatch position, as above, and the other as if we *hadn't* matched on the last character. We then use the maximum of these two values to continue our search for the last character. In the ballon/balloon case that means the maximum of X-1 (which we've already calculated) or X+6 (obtained by assuming that the n characters didn't match: the skip value for n is 6). So we'd jump six positions, and resume the scan for the last character.

Pattern	Pos time	BMPos time
'keel'	27.9	25.0
'keep'	0.1	0.2
'keek'	27.4	24.0
' keel'	53.1	21.4
' keep'	0.3	0.1
' keek'	53.2	20.3

➤ *Table 1: Comparison between Pos and BMPos for various patterns.*

The Living Daylights

After all this theorizing, time for some concrete code. First we need to calculate the skip values for each character in the pattern. We'll use an array of byte, one element for each possible ASCII character. Notice that this limits us to maximum skip values of 255, so the pattern must be 255 characters or less in size (we could use an array of integer instead, but would require 4 times as much space in 32-bit land). We initialize all elements to the length of the pattern (ie, we assume that all characters have the maximum skip value), and then we calculate the skip values for each of the characters in the pattern, by calculating the distance, in characters, to the final character of the pattern. The code in Listing 2 will do perfectly well.

Notice that the code makes the assumption that the length of the pattern string must be between 1 and 255 inclusive. We could put in some kind of check here, but we'll do so elsewhere.

Next is the actual Boyer-Moore search. To reiterate: we need two loops in this code, one to scan for the final character, the other the scan for an actual match with the rest of the pattern once we have a match on the last character. The algorithm goes as follows. Scan through the text for the last character in the pattern. Every time we have a mismatch, calculate the skip value and skip that number of characters. If we match the last character at some point, compare the rest of the pattern. If we match, hooray, it's all over; if we don't match, calculate the maximum skip we can make and proceed scanning for the last character. The full routine is in Listing 3.

We'll do the same thing as we've just done: search for the same six patterns we used in the `Pos` case

within the complete text for *Love's Labour's Lost*. Before I present the answer, I'd just like to say that at this point I didn't have much hope that the Pascal version of Boyer-Moore would be faster. After all, I was comparing a hand-optimized assembly routine with compiled Pascal, and I was looking forward no end (yeah, right) to converting the Boyer-Moore routine to assembly to prove my point. I needn't have worried: Even my *compiled Pascal* Boyer-Moore routine was faster than `Pos` in my test, except for one small case. Table 1 shows the results of `Pos` versus `BMPos` for the six patterns.

The one case where `Pos` was faster than `BMPos` could be due to the fact that the margin for error in such a small measurement is so large, but I think, more likely, it's due to the fact that `BMPos` requires a table of skip lists to be generated and this takes time. Note also that the latter three patterns are longer and that they take a smaller length of time with the Boyer-Moore routine. In general, this is the case: the Boyer-Moore algorithm works better with longer patterns because of the larger skip values. Just to prove this point, I searched for 'tongues of mocking wenches' 5000 times with both `Pos` and `BMPos`, and `Pos` took over 4 times longer to find it (the line is about three quarters of the way through the play, and is spoken by Boyet, a part I played about 4 years ago).

Hurry Home

As I mentioned earlier, one of the drawbacks with the stock `Pos` routine is that it doesn't work with case-insensitive searches. Of course we could write one that did, but writing it in Pascal would be too slow, and we'd be left with trying to shoehorn the case insensitivity testing into the assembly code. For the Boyer-Moore routine I've presented, a case-insensitive version could easily be written. Firstly, the pattern string could be converted to an uppercase version in a local variable before starting the scan and matching processes. Next, every character the routine looked at in the text would have to be converted to uppercase. You could do this from first principles by calling a Windows API routine, or you could create a table of uppercased characters before starting, and use that throughout: a case of space versus speed again. Because you will be skipping over blocks of characters in the text, you will be converting less characters in the text to uppercase than when using `Pos` (which after all requires pretty much all characters in the text to be so converted). I've included a case-insensitive version of the Boyer-Moore routine in the code with this month's diskette.

As regards the other drawback I mentioned with `Pos`, I'm sure you'll agree that changing `BMPos` to accept a starting position from which to begin the search is simplicity itself. Pass the start position in a new parameter (indeed, with Delphi 4 you can declare a default parameter as well, and set it to 1), and at the point where the original code sets `TextInx` the first time, set it equal to the maximum of `PatLen` and the passed start position. Using this enhanced Boyer-Moore routine you can find the first occurrence of the pattern with the first call (say it returns X) and then search for the second by calling the routine again, starting from `X+PatternLength`.

So, in conclusion, we can see that Boyer-Moore is a very good alternative to the `Pos` routine in the `System` unit. It works consistently better if there's a lot of text to search and the pattern is several characters long, so that the generation of the skip values can be offset by the extra speed of the search. It's *infinitely* better if we want to do things with our search that the `Pos` function was never designed to do (of course!).

Julian Bucknall is a programmer by trade and ability, actor by inclination. He finds that searching for *ah ha!* moments doesn't always work; as Archimedes found, they come by themselves in the bath. The code that accompanies this article is freeware and can be used as-is in your own applications.

Errata

Well, it didn't take long. Reader Charlie McKeegan used the code from my ternary tree article in the June issue in an application of his and found a bug. The bug was discovered in the `Insert` method of the ternary tree class, the one where I tried to remove the recursion from the standard algorithm. Charlie found that if you perform the following set of operations:

```
Insert:
'abcd'
Insert:
'abde'
Delete:
'abcd'
Insert:
'abcd'
```

and then look for 'abcd', you wouldn't find it in the tree. I do apologize for this mistake, and present the new code on this month's diskette.

As a matter of fact, this bug is an exemplary lesson in not removing recursion for the sake of it, as I pointed out in my July article on the subject. If only I'd written the two articles the other way round!

As always, if you find bugs in my articles or code, you can contact me at julianb@turbopower.com