Finding palindromic substrings

Given a string of characters, your job is to find the longest palindrome in it. How? **Julian Bucknall** indulges us with some recreational programming theory

What's covered

▶ PALINDROMIC SUBSTRINGS

Although the problem is part of recreational programming and can be simply stated as: 'write a program to find the longest palindrome within a given string', its solution can bring up some thorny, yet interesting computer science. Let's take a look at how to solve this particular problem, and at the data structures and algorithms that come into play.

A quick definition of what we're trying to find before we start: a palindrome is a sequence of characters that read the same forwards as backwards. In common English, in forming palindromes we tend to ignore spaces, punctuation and letter casing. For example, 'Madam, I'm Adam' is a well-known palindrome, as is 'A man, a plan, a canal: Panama!' For this article, though, we shall be stricter: all characters are significant.

Take a look at this sentence: 'Steve jumped into a race car and drove off, tooting the horn.' Your starter for 10 points is to discover the longest sequence of characters found in that sentence that's also a palindrome.

When I first thought about this problem, I started by considering each character and looking at its immediate left and right neighbours. If they were the same, I then expanded the range under consideration to their surrounding neighbours, and so on. For example, using this algorithm you could find 'eve' pretty quickly, but you would miss 'toot'.

Here's our first realisation: a palindrome can be centred on a single character (the 'v' in my first example), which is an odd-length palindrome or on the gap found between two characters (the gap between the two 'o's in 'toot' in the second example), which is an even-length palindrome.

Centre of attention

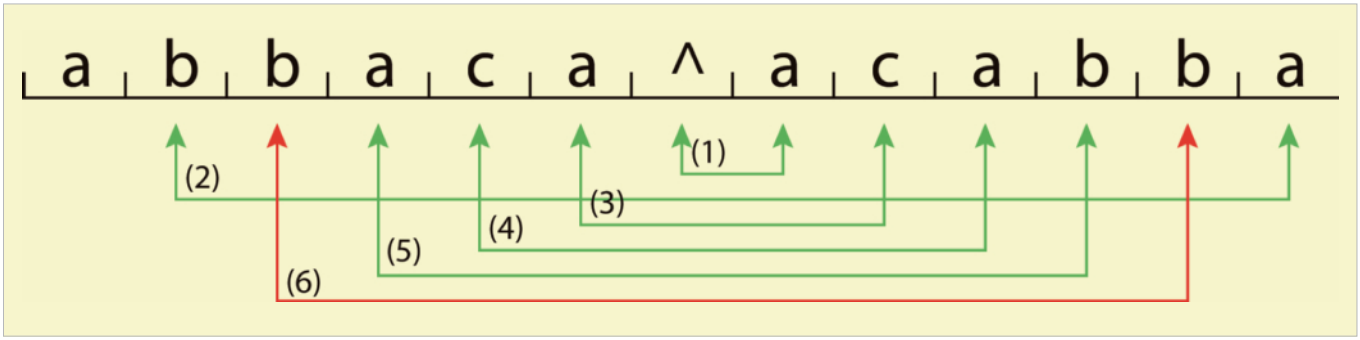
Thinking of the problem analytically, the first (and easiest) solution is to step through the string, looking at each character in the string and in between each two characters. Let's call each point of consideration a centre – after all, we're trying to find a palindrome centered on that spot. For each centre, we perform the 'find a palindrome' algorithm by looking at neighbours, and neighbours of neighbours, and so on. If the current two neighbours match, we

Spotlight on... Finite automata

A finite automaton is essentially a state machine, a model consisting of a finite set of states with well-defined transitions between those states. They can model a whole slew of different mathematical and programming problems, especially in the area of parsing. An example is a state machine that can recognise decimal numbers of the form '1.23' and signal errors for malformed strings like '1..2'. (There are three states, and we can call them 'before the decimal point', 'the decimal point', and 'after the decimal point'. The transitions between 1 and 2 is reading a decimal point, and between 2 and 3, a digit. There are also transitions between 1 and itself (a digit) and between 3 and itself (again a digit).)

One of the most striking results of finite automata theory is that given an alphabet of symbols, it is impossible to create a state machine that will recognize every palindrome written with symbols from that alphabet. Even in the case of very simple alphabets like {a, b}, it's impossible to write a state machine that would recognize aa, aba, abba, baaabaaab, and so on as palindromes. If you are familiar with regular expressions from your programming environment, this remarkable result also implies that you can't write a regular expression that will match any palindrome. (In essence, every regular expression can be converted to a state machine, and no state machine exists to recognize every palindrome). Compare this result with my example of a state machine that can recognize any properly formed decimal number, no matter how many digits there may be before or after the decimal point.

Of course, it's really easy to write a routine that can recognise whether a string is a palindrome or not and, as we've seen, it's relatively straightforward to write one that can find the longest palindromic substring.



▲ **Figure 1: The steps taken to find an even-length palindrome for abbaca**

continue; if they don't, we stop. So for example, if the current centre was between the two 'p's in unflappable, we'd see the gap, 'pp', 'appa' and then we would note that the next two neighbours: 'l' and 'b', are not the same and stop. If we find a palindrome of length two or more, and it's longer than the longest we've found so far, we make a note of it and move to the next step.

If the string is of length n, there are n-2 character centres (there's no need to examine the first and last characters since they don't have neighbours on one side and so can't be possible palindrome centres), and there are n-1 inter-character gaps. This makes 2n-3 possible centres. For each centre, we would have to check up to m characters either side, where m <= n/2. All in all then, this algorithm is O(n^2); that is, it will execute in time that is proportional to the square of the number of characters. In other words, finding the longest substring palindrome in a string of 1,000 characters would take roughly 100 times as long as finding that for a string of 100 characters.

Can we do better and get a faster implementation?

Suffix array

The next step to a faster algorithm is to use a suffix array. A suffix array is a list of indexes; character positions in a given string. Each index denotes the starting character of a suffix of the string. The suffixes are alphabetically sorted and the order of the indexes in the array corresponds to this sorted list of suffixes.

Let's see how this works with regard to the word apparatus. The suffixes (and their corresponding indexes) are: apparatus (0), pparatus (1), paratus (2), aratus (3), ratus (4), atus (5), tus (6), us (7), s (8). Now sort the suffixes: apparatus (0), aratus (3), atus (5), paratus (2), pparatus (1), ratus (4), s (8), tus (6), us (7). Now we can drop the (explicit) suffix strings and create the suffix array to contain just the indexes: [0, 3, 5, 2, 1, 4, 8, 6, 7]. The nice thing about a suffix array is that we don't need to store the suffixes themselves, we can just read them off using the index value and the original string.

From the suffix array, we can easily calculate another important array, known as the longest common prefix or lcp array. This array contains the length of the common prefix between a suffix and its predecessor in the suffix array. Returning to our example, the first entry in the lcp array is 0 (there is no predecessor to apparatus). The next entry is 1 (the common prefix between apparatus and aratus is a), next is 1 (the common prefix of aratus and atus is a), then 0, 1, with the remaining entries 0. The full lcp array is then [0, 1, 1, 0, 1, 0, 0, 0, 0].

Using the lcp array, we can easily work out the longest common prefix between suffixes that are not adjacent in the suffix array: we take the minimum lcp length between the index of the first suffix and the index of the second. For example, the lcp length between apparatus (element 0) and atus (element 2) is the minimum of that between elements 0 and 1 (which is 1), and between elements 1 and 2 (which is also 1); hence, 1.

How does this help us with our palindrome problem? What we do is to concatenate our original string with some special character and the reverse of our original string to form another, longer string.

SUFFIXES	SUFFIX ARRAY	LCP ARRAY
^acabba	6	0
a	12	0
a^acabba	5	1
abba	9	1
abbaca^acabba	0	4
aca^acabba	3	1
acabba	7	3
ba	11	0
baca^acabba	2	2
bba	0	1
bbaca^acabba	1	3
ca^acabba	4	0
cabba	8	2

▲ **Table 1: The suffix and lcp arrays for abbaaca^acaabba**

So, for example, if our original string were abbaca we would create abbaca^acabba, as the new string.

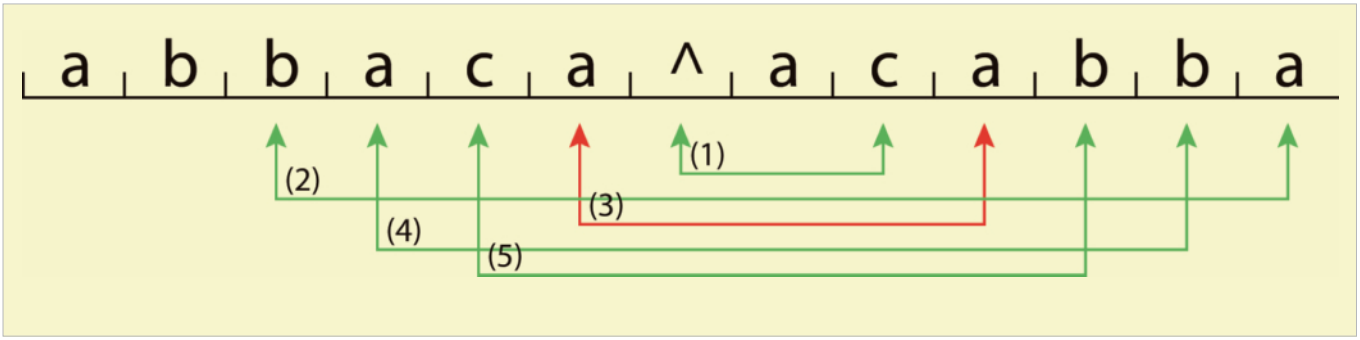
The next step is to create the suffix array and the lcp array for this new string. I've shown this in table 1 (above). I'm sure that, in looking at this table, you're beginning to get an idea for what is about to happen: the suffix that starts the longest palindrome for the original string is going to be pretty close (if not adjacent) to the suffix for the same palindrome in the reversed string, and the lcp value is going to dictate the length of the palindrome.

Even-length

So let's see how this method pans out. There are two cases: discovering the longest even-length palindrome and finding the longest odd-length palindrome.

For an even-length palindrome, the centre will be a gap between two characters. The idea here is to match suffixes from the front of the longer string to the equivalent positions at the end, but off by one. Using the lcp array, we can calculate half the length of the palindrome, and therefore the actual palindrome itself is easy to generate.

The insight is that we consider the indexes as pairs. The pairs are (1, 12), (2, 11), (3, 10) and so on. The second element of the pair (which is in the reversed part of the string) is the character before the character denoted by the first element (which is in the first part of the string). We can also make an optimization that says, if the characters at the



▲ **Figure 2: The steps taken to find an odd-length palindrome for abbaca**

Suffix trees

Suffix arrays are derived from suffix trees, but use a great deal less memory. A suffix tree is a special kind of tree that is a data structure for storing the suffixes of a given string. The structure provides fast algorithms to solve many string-related problems, such as finding the longest repeated substring, approximate string matching, string comparisons and so on. At least in theory, a suffix tree can be built over a given string in time proportional to the length of the string. The suffix tree, when read using an in-order traversal, will then provide the suffixes in sorted order.

The main problem with suffix trees is that, for any respectable problem space (say the area of the human genome) they are memory hungry enough to require that they are stored on disk. Fortunately, there are many refined algorithms for building suffix trees to go on a disk for such long strings, such as TRELLIS and DiGeST.

indexes in each pair are not the same, ignore that pair – it won't form an even-length palindrome (we're essentially ignoring palindromes of length 0).

So we go through the suffix array element by element, considering pairs. In our example, the first element of the suffix array is 6, paired with 7. These characters (^ and a) are not the same, so we ignore them. (Follow along with the numbered steps found in figure 1.) The next element is 12, paired with 1. Again the characters (a and b) are not the same, so we ignore them. Next is 5, paired with 8; again different characters. We continue like this until we reach 11, paired with 2. The characters at those indexes are the same (b), and we can calculate the lcp length between them: 2. This means that there is a palindrome of length four at this centre. We can back up two positions (the lcp length) from index 2, and count out four characters to get the palindrome abba.

We've now considered all possible pairs of indexes and only found one even-length palindrome: abba. (It helps if we make a note somewhere of which indexes we've considered as we iterate through the suffix array. A Boolean array would suffice; mark elements true if we've visited them).

Odd-length

Now for the odd-length palindrome. Here the centre of the palindrome will be a character. We do the same kind of trick with index pairs but we match the suffix of the character after the centre in the first half of the longer string with the suffix of the character before the centre in the reversed part of the string. The pairs are then (2, 12), (3, 11), (4, 10), (5, 9), (6, 8). Again we can do the trick of checking the characters at the indexes to be equal before doing anything more (we're essentially not interested in palindromes of length one). Again we need to iterate through the suffix array, marking those indexes we've visited.

You can follow along with the numbered steps in Figure 2. You will find that (6, 8) doesn't work since the characters are different; neither does (12, 2). The next pair to be considered are (5, 9) and the characters match. The lcp array tells us that the longest prefix is one character long, which we can calculate as a. We can now calculate the palindrome as aca. (3, 11) and (10, 4) don't produce palindromes. Hence, overall the longest odd-length palindrome is aca.

Therefore, taking both searches into account, the longest palindrome in abbaca is abba.

Let's consider the overall complexity of this algorithm. First of all the calculation of the suffix array is O(n. logn) – a standard result for an algorithm dominated by a sorting process. The calculation of the lcp array is linear. Because of the optimisations I have made, checking the characters at the indexes to be equal before doing anything else, the search through the suffix array to find the other half of a pair is guaranteed to be local, and hence on average linear in time. Therefore, overall, this algorithm, despite its seeming complexity, is O(n.logn), which is is very much smaller than O(n^2).

Manacher's Algorithm

There is, however, yet another algorithm, Manacher's Algorithm, that proves to be linear. I'll sketch out the details here. Let's take abbaca again as our example string in which we want to discover the longest palindrome.

To make it easier, let's insert a special character in between each of the letters to make another string (and then we don't have to worry about the gaps): *a*b*b*a*c*a*. We'll assume that the palindromes we look for cannot contain the special character (otherwise, we'll be saying that *a* is a palindrome).

Let's create (from scratch) an array for each character position in this longer string that defines the longest palindrome at that centre. This gets me [0, 1, 0, 1, 4, 1, 0, 1, 0, 3, 0, 1, 0] – the 4 is for the gap between the two 'b's, the 3 is positioned at the 'c'. Notice something interesting about this array? The numbers form palindromes as well.

Using this insight, we can fill in this array ahead of the current centre as we read through the input string. Instead of reading through the string character by character, we shall read through it centre by centre, recording the current centre's palindrome length, and estimating the palindrome lengths that are in front of us.

So when we get to the gap between the 'b's, we're at this situation: *a*b*|b*a*c*a* (where the vertical mark indicates where we've reached), and the lengths array is [0, 1, 0, 1, 4, ?] since we've worked out that abba is a palindrome centred there. We could fill in the lengths array to the right with guestimates from what's happened on the left: [0, 1, 0, 1, 4, *1*, *0*, *1*, *0*, ?] (guestimates are in italics). The second 'b' can't be a centre (it's in the middle of the palindrome), but that second 'a' certainly could be, so we jump there and make it the next centre we consider for a palindrome.

This is the essence of Manacher's Algorithm. There are some special cases where a palindrome substring is the prefix to a longer palindrome, but in essence, using this technique, we find we can jump ahead in the string and not have to consider every character as a centre. For this reason Manacher's Algorithm turns out to be linear in time.

So there you have it, by considering a recreational puzzle, we've delved into some interesting algorithms on the way to creating a fast solution to finding palindromic strings. And the solution to the very first puzzle? I wonder if you got 'a racecar a'. ■