« Back home

Maximum length palindromic subsequence

Posted by adrian.ancona on April 8, 2015

Given a sequence of characters, find the longest palindromic sub-sequence. A sub-sequence is any sequence that can be formed by removing 0 or more characters from the given sequence. For example, the possible sub-sequences for ABC are:

```
ABC
BC
C
B
AC
C
A
AB
B
A
```

Removing the duplicates you have:

```
ABC
AC
A
BC
B
CD
C
```

We can calculate all the sub-sequences using this code:

```javascript
function sub(word) {
    var seq = [word];

    if (word.length === 1) {
        return seq;
    }

    for (var i = 0; i < word.length; i++) {
        seq = seq.concat(
            sub(
                word.substring(0, i) + word.substring(i + 1, word.length)
            )
        );
    }

    return seq;
}
```

The problem with this code is that it has an exponential time complexity and it calculates sub-sequences more than once. We can make this a little more efficient avoiding duplicates:

```javascript
function sub(sequence) {
    var sequences = {};

    function doIt(word) {
        sequences[word] = true;

        if (word.length === 1) {
            return;
        }

        var s;
        for (var i = 0; i < word.length; i++) {
            s = word.substring(0, i) + word.substring(i + 1, word.length);
            // Only calculate subsequences if they haven't already been calculated
            if (!sequences[s]) {
                doIt(s);
            }
        }
    }

    doIt(sequence);

    return sequences;
}
```

Because an object(hasmap) is used to keep a record of the sub-sequences that have already been calculated we avoid duplicate work. We can now incorporate a function to check if the sub-sequence is a palindrome:

```
function palindromicSubsequence(sequence) {
    var sequences = {};
    var largestPalindrome = '';

    function isPalindrome(word) {
        var left = 0;
        var right = word.length - 1;

        while (left < right) {
            if (word[left] !== word[right]) {
                return false;
            }
            left++;
            right--;
        }

        return true;
    }

    function sub(word) {
        // If the largest palindrome is already larger than this word then there
        // is no point on continuing on this path
        if (largestPalindrome.length >= word.length) {
            return;
        }

        if (isPalindrome(word)) {
            largestPalindrome = word;
        }
        sequences[word] = true;

        if (word.length === 1) {
            return;
        }

        var s;
        for (var i = 0; i < word.length; i++) {
            s = word.substring(0, i) + word.substring(i + 1, word.length);
            // Only calculate subsequences if they haven't already been calculated
            if (!sequences[s]) {
                sub(s);
            }
        }
    }

    sub(sequence);

    return largestPalindrome;
}
```

I'm not really sure about the complexity of this algorithm but it should be a lot faster than the previous version.

There is another option that takes O(n^2) that makes use of these observations:

```
If the sequence is represented as W[0, n-1] and the largest palindrome is represent

L[i, i] is always 1
Every single character is a palindrome

if (W[0] !== W[n-1]) then L[0, n-1] = max(L[0, n-2], L[1, n-1])
If the first and last characters are not the same then the
longest palindrome is the longest palindrome of the characters
0 to n-2 or 1 to n-1

if (W[0] === W[n-1]) and n === 2 then the result is 2
If the first and last characters are the same and the
word is two letter long then the result is 2

if (W[0] === W[n-1]) and n > 2 then L[0, n-1] = L[1, n-2] + 2
If the first and last characters are the same then
the longest palindrome is 2 + L[1, n-2]
```

Using these observations we can write a faster program:

```
function ps(sequence) {
    var found = {};

    function doIt(word) {
        function max(l, r) {
            return l > r ? l : r;
        }

        if (word.length === 1) {
            return 1;
        }

        // If this word has already been calculated don't do it again
        if (found[word]) {
            return found[word];
        }

        var res;
        if (word[0] !== word[word.length - 1]) {
            res =  max(ps(word.substring(1)), ps(word.substring(word.length -1, -1)
        } else {
            if (word.length === 2) {
                res = 2;
            } else {
                res = ps(word.substring(1, word.length -1)) + 2;
            }
        }

        found[word] = res;
        return res;
    }

    return doIt(sequence);
}
```

According to a test in JSPerf, this last option is about 20 times faster.

[computer_science algorithms javascript programming]

Share this:

Push notifications on web applications

Null terminated and length prefixed strings

Using service workers for caching

Debugging assembly with GDB

Assembly - Variables, instructions and addressing modes

What do you think?

0 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments ncona.com Disqus' Privacy Policy 1 Login

♡ Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your site Do Not Sell My Data

DISQUS