« 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:

```
1 ABC
2 BC
3 C
4 B
5 AC
6 C
7 A
8 AB
9 B
10 A
```

Removing the duplicates you have:

```
1 ABC
2 AC
3 A
4 BC
5 B
6 CD
7 C
```

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

```
1
   function sub(word) {
2
       var seq = [word];
3
4
        if (word.length === 1) {
5
            return seq;
6
7
8
        for (var i = 0; i < word.length; i++) {</pre>
9
            seq = seq.concat(
                sub (
10
11
                    word.substring(0, i) + word.substring(i + 1, word.length)
12
13
           ) ;
14
15
16
       return seq;
17
```

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:

```
1
   function sub(sequence) {
2
       var sequences = {};
3
       function doIt(word) {
4
5
            sequences[word] = true;
6
7
            if (word.length === 1) {
8
                return;
9
10
11
           for (var i = 0; i < word.length; i++) {</pre>
12
13
                s = word.substring(0, i) + word.substring(i + 1, word.length);
                // Only calculate subsequences if they haven't already been calculated
14
15
                if (!sequences[s]) {
                    doIt(s);
16
17
18
19
20
21
       doIt(sequence);
22
23
       return sequences;
24
```

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:

```
1
   function palindromicSubsequence(sequence) {
2
       var sequences = {};
3
       var largestPalindrome = '';
4
5
       function isPalindrome(word) {
           var left = 0;
6
7
            var right = word.length - 1;
8
9
            while (left < right) {</pre>
10
                if (word[left] !== word[right]) {
11
                    return false;
12
13
                left++;
14
                right--;
15
16
17
            return true;
18
19
20
        function sub(word) {
21
            // If the largest palindrome is already larger than this word then there
22
            // is no point on continuing on this path
23
            if (largestPalindrome.length >= word.length) {
24
                return;
25
26
27
            if (isPalindrome(word)) {
28
                largestPalindrome = word;
29
30
            sequences[word] = true;
31
32
            if (word.length === 1) {
33
                return;
34
35
36
            var s;
37
            for (var i = 0; i < word.length; i++) {</pre>
                s = word.substring(0, i) + word.substring(i + 1, word.length);
38
39
                // Only calculate subsequences if they haven't already been calculated
40
                if (!sequences[s]) {
                    sub(s);
41
42
43
44
45
46
        sub (sequence);
47
```

```
return largestPalindrome;
49 }
```

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:

```
1
   If the sequence is represented as W[0, n-1] and the largest palindrome is represent
2
3
   L[i, i] is always 1
   Every single character is a palindrome
4
5
   if (W[0] !== W[n-1]) then L[0, n-1] = max(L[0, n-2], L[1, n-1])
6
7
   If the first and last characters are not the same then the
8
   longest palindrome is the longest palindrome of the characters
   0 to n-2 or 1 to n-1
9
10
11 if (W[0] === W[n-1]) and n === 2 then the result is 2
12 If the first and last characters are the same and the
13 word is two letter long then the result is 2
14
15 if (W[0] === W[n-1]) and n > 2 then L[0, n-1] = L[1, n-2] + 2
16 If the first and last characters are the same then
17 the longest palindrome is 2 + L[1, n-2]
```

Using these observations we can write a faster program:

```
1
   function ps(sequence) {
2
       var found = {};
3
4
       function doIt(word) {
5
           function max(l, r) {
                return 1 > r ? 1 : r;
6
7
8
9
            if (word.length === 1) {
                return 1;
10
11
12
            // If this word has already been calculated don't do it again
13
            if (found[word]) {
14
15
                return found[word];
16
17
18
           var res;
19
           if (word[0] !== word[word.length - 1]) {
```

```
20
                res = max(ps(word.substring(1)), ps(word.substring(word.length -1, -1)
21
            } else {
                if (word.length === 2) {
22
                   res = 2;
23
24
                } else {
25
                    res = ps(word.substring(1, word.length -1)) + 2;
26
27
28
29
           found[word] = res;
           return res;
30
31
32
33
       return doIt(sequence);
34
```

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

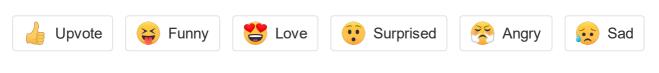
Using service workers for caching

Debugging assembly with GDB

Assembly - Variables, instructions and addressing modes

What do you think?

0 Responses







Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?









Name

Be the first to comment.

⊠ Subscribe





DISQUS





