Jeuring's algorithm on palindromes

An implement with literate programming and C++ programming language

July 20, 2012

1 Introduction

All the techniques used in this article is well-introduced in Don Knuth's magnificent book [1], which is strongly-suggested in Computer Science. This is a very efficient algorithm, designed by Johan Jeuring, which is used to determine the longest palindrome in $O(n)$ time. You can read his original article [4] and his pretty Haskell code. Now, I want to show how his algorithm works and why it's right.

As usual, a program (especially C/C++ code) is made up of three parts: preprocessors, global variables and routines.

1a ⟨*pldrm.cc* 1a⟩≡

⟨*preprocessors* 1b⟩
⟨*globals* 2c⟩
⟨*main* 2a⟩

It's essential to include the standard libraries, such as cstdio, cstdlib and cstring.

1b ⟨*preprocessors* 1b⟩≡ (1a) 2b ▷

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <algorithm>
using namespace std;
```

2 Main routine

Here's the main routine.

2a ⟨*main* 2a⟩≡ (1a)

```
int main()
{
  ⟨mainvar 3b⟩
  ⟨input 2d⟩
  ⟨main loop 3c⟩
  ⟨output 2e⟩
  return 0;
}
```

Given that $s[1\mathinner{..}n]$ is the string inputted, and n is the length of s, where $s[0] = 1, s[n+1] = 0$.

2b ⟨*preprocessors* 1b⟩+≡ (1a) ◁1b

```
#define MAX_LEN 100000
```

2c ⟨*globals* 2c⟩≡ (1a) 3a▷

```
int n;
char s[MAX_LEN+2];
```

2d ⟨*input* 2d⟩≡ (2a)

```
s[0] = 1;
scanf("%s", &s[1]);
n = strlen(&s[1]);
```

Definition 1. *We say $l + r$ is the center of substring $s[l\mathinner{..}r]$. For example, 4 is the center of $s[2\mathinner{..}2]$ or $s[1\mathinner{..}3]$.*

a_k is the length of the longest palindrome whose center is k, and $a[0\mathinner{..}2n]$ is the array to save $\langle a_k \rangle$.

2e ⟨*output* 2e⟩≡ (2a)

```
for (int k=1; k<=2*n+1; k++) {
  printf("%d\n", a[k]);
}
```

3 Main loop

Definition 2. *We call some string A is a tail palindrome of the other string B if and only if A is a palindromic tail substring of B. For example, $A = aba$ and $B = aaaababa$, where A is palindromic and A is a tail substring of B.*

The main loop calculate array a in the increasing order of the index. We will see that the invariant of main loop is assertion 1.

3a ⟨*globals* 2c⟩+≡ (1a) ◁2c

```
int a[2*MAX_LEN+4];
```

3b ⟨*mainvar* 3b⟩≡ (2a) 4a▷

```
int j, l;
```

3c ⟨*main loop* 3c⟩≡ (2a)

```
j = 1;
l = 1;
a[0] = 1;
a[1] = 0;
for (int k=2; k<=n+1; k++) {
  ⟨process 3d⟩
  advance:
  ;
}
```

Process is made up of an infinite loop, which is used to find the longest tail palindrome of $s[0\,..\,k]$. There are two exits of it. One is in extension subroutine, while the other one is in move loop. The way of exit is *goto advance;*. The loop invariant for process is assertion 3.

3d ⟨*process* 3d⟩≡ (3c)

```
for (;;) {
  ⟨check 3e⟩
  ⟨move loop 4b⟩
}
```

The check subroutine checks whether a tail palindrome of $s[0\,..\,k-1]$ could be extended to that of $s[0\,..\,k]$. If so, exit from the process loop and advance k, otherwise start the move loop.

3e ⟨*check* 3e⟩≡ (3d)

```
if (s[k] == s[k-l-1]) {
  l += 2;
  goto advance;
}
```

Here's the move loop, which looks short and easy. It's used to find a shorter tail palindrome of $s[0\,..\,k-1]$.

4a ⟨*mainvar* 3b⟩+≡ (2a) ◁3b

```
int p;
```

4b ⟨*move loop* 4b⟩≡ (3d)

```
a[++j] = l;
for (p=j-1; --l>=0&&l!=a[p]; p--) {
  a[++j] = min(a[p], l);
}
if (l < 0) {
  l = 1;
  goto advance;
}
```

4 The outline of the proof

Lemma 1. *An arbitrary tail palindrome of* $s[0\,..\,m]$ *is also a tail palindrome of* $s[1\,..\,m]$ *for all* $m > 0$.
Lemma 2. *The substring of* s *whose center is* c *and length is* l *is* $s[(c-l+1)/2\,..\,(c+l-1)/2]$.

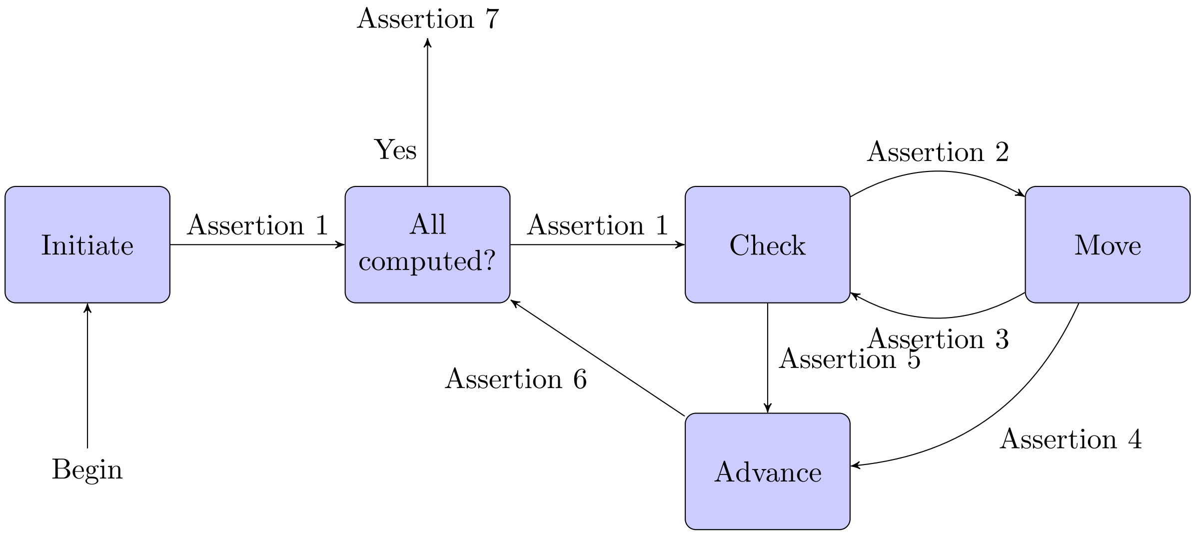

Figure 1: Flowchart of the main loop

Now let's label some arrows with an assertion on the flow chart of the main loop. This is a technique to prove the validity of the algorithm, introduced in Don Knuth's *The Art of Computer Programming* [1], section §1.2.1, mathematical induction. I quote the most important statement on that book here:

> if an assertion attached to any arrow leading into the box is true before the operation in that box is performed, then all of the assertions on relevant arrows leading away from the box are true after operation ...... For we can now use induction on the number of steps of the computation, in the sense of the number of arrows traversed in the flow chart.

Assertions are stated:
Assertion 1. $2 \le k \le n+1$, *and the length and the center of the longest tail palindrome of* $s[0\,..\,k-1]$ *is* l *and* $j+1$, *while* $a[0\,..\,j]$ *is computed. (Notice that once assigned,* $a[t]$ *will not be reassigned, therefore that* $a[t]$ *is computed/calculated/determined means that* $a[t]$ *is assigned and* $a[t] = a_t$*.)*
Assertion 2. $2 \le k \le n+1$, $s[k-l\,..\,k-1]$ *is a palindrome or empty string, whose center is* $j+1$, $l \ge 0$, *the length of the longest tail palindrome of* $s[0\,..\,k]$ *is less than* $l+2$, *and* $a[0\,..\,j]$ *is calculated.*
Assertion 3. $2 \le k \le n+1$, $s[k-l\,..\,k-1]$ *is a palindrome or empty string, whose center is* $j+1$, $l \ge 0$, *the length of the longest tail palindrome of* $s[0\,..\,k]$ *is not more*

than $l+2$, *and* $a[0\,..\,j]$ *is assigned.*
Assertion 4. $2 \le k \le n+1$, *the length and the center of the longest tail palindrome of* $s[0\,..\,k]$ *is* $l = 1$ *and* $j+1$, *where* $a[0\,..\,j]$ *is determined.*
Assertion 5. $2 \le k \le n+1$, *the length and the center of the longest tail palindrome of* $s[0\,..\,k]$ *is* $l \ge 2$ *and* $j+1$, *and* $a[0\,..\,j]$ *is calculated.*
Assertion 6. $3 \le k \le n+2$, *the length and the center of the longest tail palindrome of* $s[0\,..\,k-1]$ *is* l *and* $j+1$, *and* $a[0\,..\,j]$ *is computed.*
Assertion 7. *the length and the center of the longest tail palindrome of* $s[0\,..\,k-1]$ *is* l *and* $j+1$, *and* $[0\,..\,j]$ *is determined, where* $k = n+2, l = 1$ *and* $j = 2n+1$.

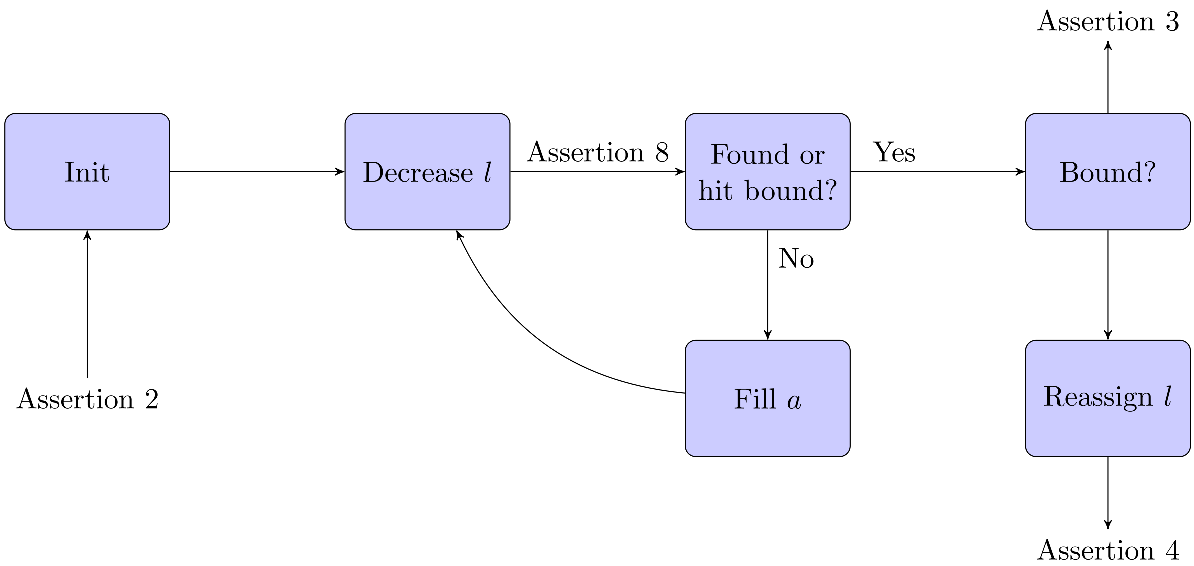

Figure 2: Flowchart of the move loop

Now we observe a non-trivial lemma, which is the key to the move loop.
Lemma 3. *Suppose that* $l = a_c$, j *is a positive integer such that* $j \le l$, *and* $a_{c-j} \ne l - j$, *we have* $a_{c+j} = \min(a_{c-j}, l - j)$.

Just like the flowchart of the main loop, we can make some assertions on the flow chart of the move loop. I will figure out the critical part, so that others could be discovered mechanically.
Assertion 8. $2 \le k \le n+1, l \ge -1$, *the center of* $s[k-l-1\,..\,k]$ *is* $j+1$, $a[0\,..\,j]$ *is well-computed,* $p + j = 2j_0 - 1$, *and the length of the longest tail palindrome of* $s[0\,..\,k]$ *is not more than* $l+2$, *where* j_0 *is the* j *after* init.

5 Analysis of the algorithm

It's so hard for me to analyze the algorithm, so I'll show some partial results, which results in the running time complexity is $\Theta(n)$.

Look at the flow chart of the main loop. Suppose that A, B is the times of going through the arrow from *Move* to *Check* and *Move* to *Advance*, and C is the times of running of *Fill a*. Out of Kirchhoff's first law for the flow chart, we obtain:

Initiate	1
All computed?	$n+1$
Check	$n+A$
Advance	n
Init	$A+B$
Decrease l	$A+B+C$
Found or hit bound?	$A+B+C$
Fill a	C
Bound?	$A+B$
Reassign l	B

For there are only two operations containing $j \leftarrow j+1$: *Init* and *Fill a*, we have $A+B+C = 2n+1-1 = 2n$. Given that b_k is the length of longest tail palindrome of $s[0\mathinner{..}k]$, we have

$$B = \sum_{k=2}^{n+1} [b_k = 1]$$

where $[P]$ is Iverson bracket [8], which is well introduced in *Concrete Mathematics* [2]. I cannot get the more precise result for A and C. Anybody good at mathematics or familiar with analysis of algorithms could help me!

References

[1] Donald E. Knuth: *The Art of Computer Programming* Volume 1, third edition.

[2] Ronald L. Graham, Donald E. Knuth, Oren Patashnik: *Concrete Mathematics* second edition.

[3] *One-Page Guide to Using Noweb with LaTeX*

[4] Johan Jeuring: *Finding palindromes efficiently*

[5] T Oetiker: *The Not So Short Introduction to LaTeX 2ε* (1995)

[6] Kjell Magne Fauske: *Example: Simple flow chart*

[7] The TikZ and PGF manual: *Example: State machine*

[8] *Iverson bracket*