Assignment 10

Benjamin Jakubowski

August 2, 2016

1 Naive-String-Matcher with unique character pattern

Suppose that all characters in the pattern P are different. We show how to accelerate `Naive-String-Matcher` to run in time $O(n)$ on an n-character text T.

```
1: function Naive-String-Matcher(P, T)
2:     k = 1
3:     for i = 1 to n do
4:         if P[k] == T[i] then
5:             k++                    ▷ Characters match, so check next char. in pattern
6:         else
7:             if P[1] == T[i] then                        ▷ See if matches start of pattern
8:                 k = 2                                   ▷ Next char. to check is P[2]
9:             else
10:                k = 1              ▷ Chars. didn't match, so check pattern from start
```

This algorithm maintains two indices into T and P, and exploits the uniqueness of characters in P, which allows us to run a single scan of T. It runs in $O(n)$ time, since

Line number	**Runtime**
2: $k = 1$	$O(1)$
3-6: `for` loop	$O(n)$, since we are making at most two comparisons and changing two indices in each step of the `for` loop

2 Pattern matching with a gap character $\diamondsuit$

Suppose we allow the pattern P to contain occurrences of a gap character $\diamondsuit$ that can match an arbitrary string of characters (even one of zero length). For example, the pattern $ab\diamondsuit ba\diamondsuit c$ occurs in the text $cabccbacbacab$ as $c\underline{ab}cc\underline{ba}cba\underline{c}ab$ and as $c\underline{ab}ccbac\underline{ba}\underline{c}ab$.

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. We give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T, and analyze the running time of our algorithm.

First, we outline our algorithm:

- First, we split the pattern $P = P_1 \diamondsuit P_2 \diamondsuit \cdots \diamondsuit P_k$ into an ordered list of k subpatterns that don't include $\diamondsuit$.
- Then, for each subpattern starting with P_1, we use a variant of `Naive_string_matcher` to find a match in T.
- Then we search for the next subpattern, beginning with the first unmatched character in T.

Next we present the algorithm in pseudocode:

```
function MAIN_GAP_MATCHER(P, T)
    P_list = split(P, ◇)                ▷ Split P list of substrings using ◇ as separator
    string_match_with_gaps(P_list, T, 0, 0)
function STRING_MATCH_WITH_GAPS(P_list, T, i, j)
    """
    i is index to first unseen character in T
    j is index of subpattern P_j in P_list, i.e. P_list[j].
    Note we use the notation m_j to represent length(P_j).
    """
    for s = 0 to n − m_j − (i − 1) do:
        if P_j matches T[s + i . . s + i + m_j − 1] then
            if j < k then
                return string_match_with_gaps(P_list, T, i + s + m_j, j + 1)
            else
                return "Pattern matched"
    return "Pattern not matched"         ▷ Only reached if control falls out of loop
```

Now we analyze the runtime of this algorithm.

- Splitting the pattern string P on $\diamondsuit$ is $O(n)$.
- The function `string_match_with_gaps` is called at most k times (the number of subpatterns in P). Each call runs `Naive_string_matcher` on a substring of T. Hence, the call `string_match_with_gaps`$(P_{list}, T, 0, 0)$ is

  $$O(k)O((n - \max_{j \in 1, \cdots, k} m_j + 1) \max_{j \in 1, \cdots, k} m_j)$$

  Thus, letting $m = \max_{j \in 1, \cdots, k} m_j$ we have $O(km(n - m - 1))$, which is polynomial.

3 Running Rabin-Karp

Working modulo $q = 11$, the Rabin-Karp matcher encounters 3 spurious hits with the text $T = 3141592653589793$ when looking for the pattern $P = 26$.

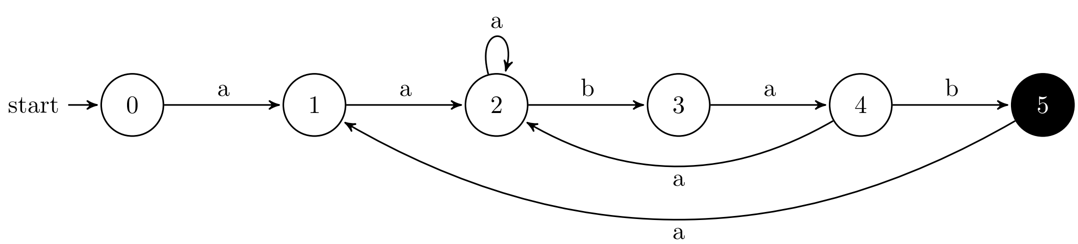
```
Running rabin-karp-matcher with T = 3141592653589793, P = 26, and q = 11
Spurious hit at shift 3 with pattern 15
Spurious hit at shift 4 with pattern 59
Spurious hit at shift 5 with pattern 92
Pattern occurs at shift 6 with pattern 26
```

4 Constructing a string-matching automaton

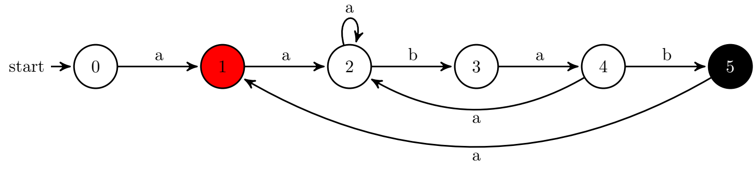
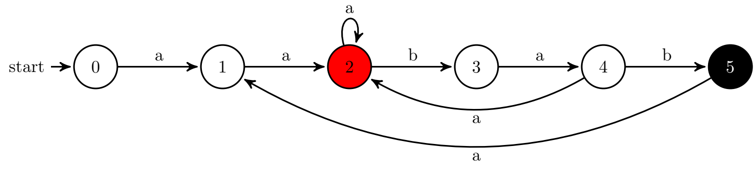
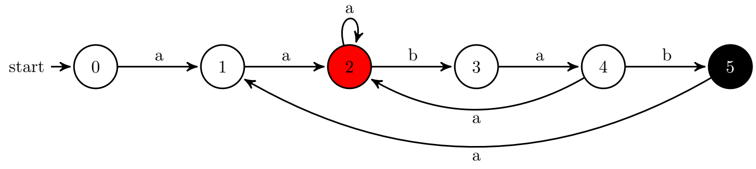
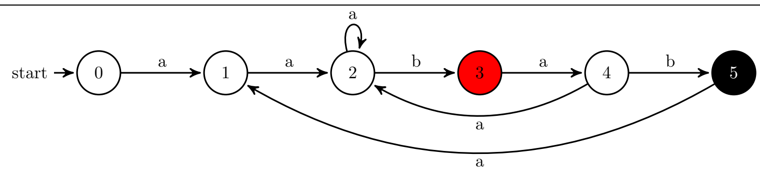
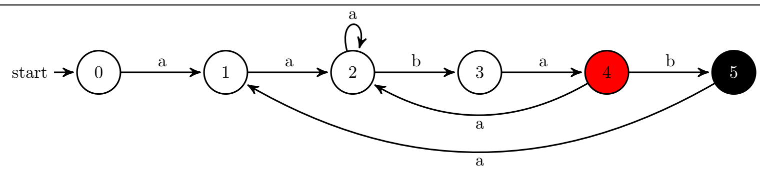
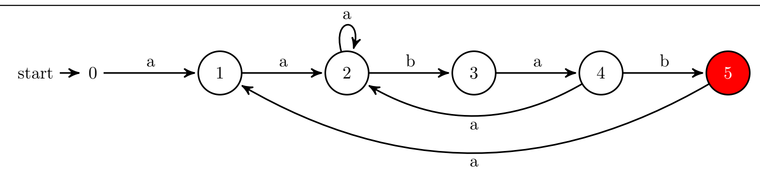
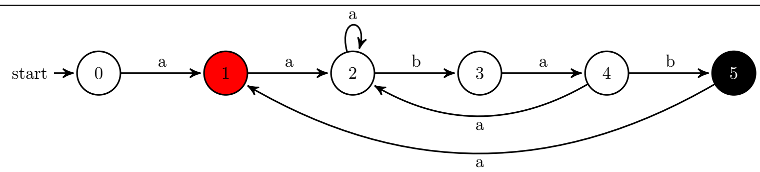
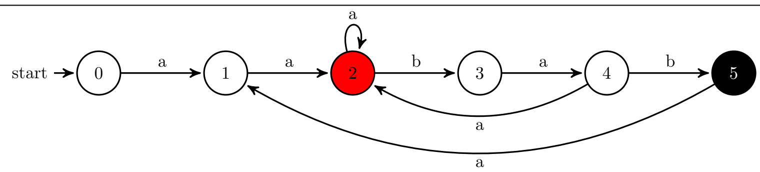
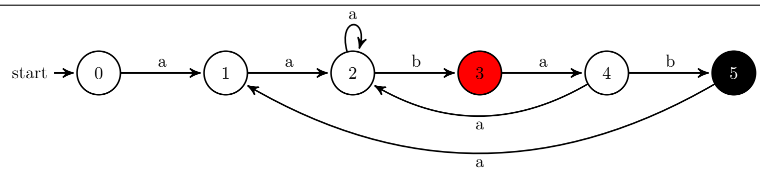
We construct the string-matching automaton for the pattern $P = aabab$ and illustrate its operation on the text string $T = aaababaabaababaab$. First, here is the transition table:

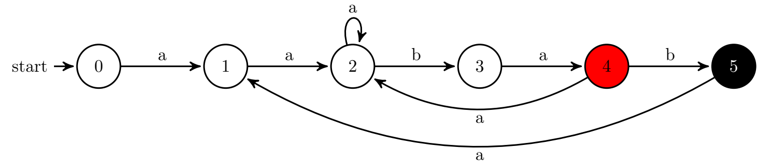
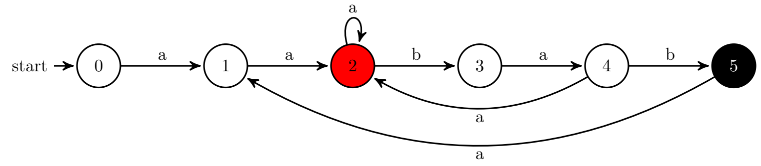
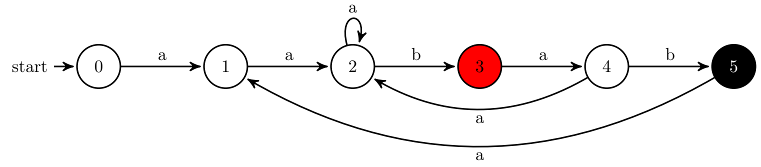
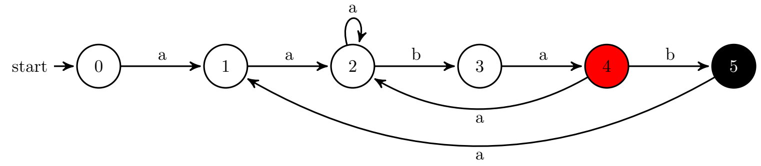
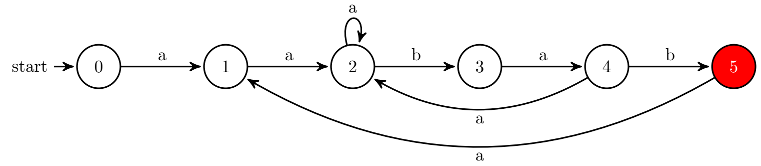
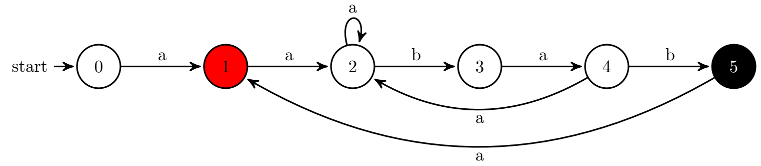
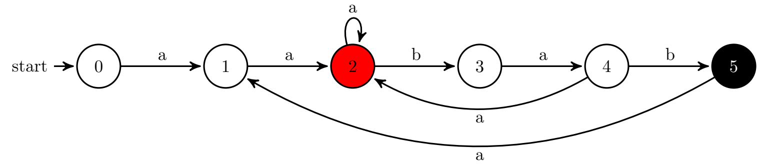
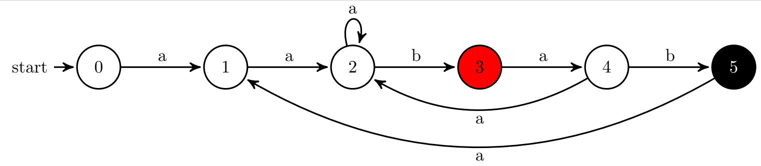
State	**Input = a**	**Input = b**
0	1	0
1	2	0
2	2	3
3	4	0
4	2	5
5	1	0

Here is the automaton shown as a graph- note (per convention) missing edges from state q on input *char* reflect the transition $\delta(q, char) = 0$:



Next, we illustrate its operation on the text string T.

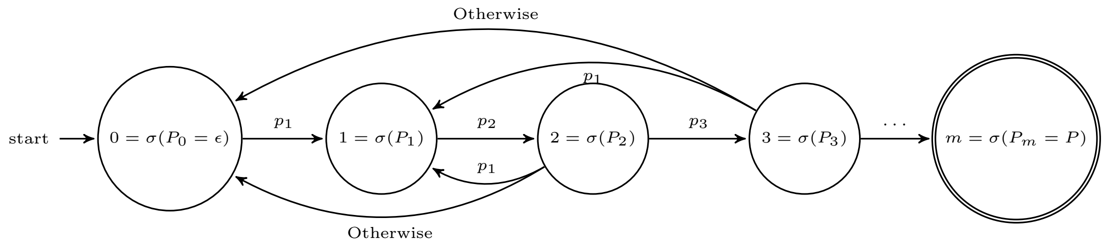
Chars. consumed	State (current state in red)
a	
aa	
aaa	
aaab	
aaaba	
aaabab	
aaababa	
aaababaa	
aaababaab	

$aaababaaba$	
$aaababaabaa$	
$aaababaabaab$	
$aaababaabaaba$	
$aaababaabaabab$	
$aaababaabaababa$	
$aaababaabaababaa$	
$aaababaabaababaab$	

5 Automaton structure for non-overlappable patterns

We call a pattern P *nonoverlappable* if $P_k \sqsupset P_q$ implies $k = 0$ or $k = q$. We describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

First, here's a graph (partially) showing the state-transition diagram of the string-matching automaton for a nonoverlappable. A full justification of this diagram is given below:



To see this diagram is in fact correct, first note if $P_k \sqsupset P_q$ implies $k = 0$ or $k = q$, then the first character in the pattern must be unique (i.e. $p_1 \neq p_j$ for $j = 2, \cdots, m$, where $m = |P|$). If not, then there is some $P_q, q \neq 1$, such that $P_1 \sqsupset P_q$, which is a contradiction.

But then, given a state P_k (with $\sigma(P_k) = k$),

$$\delta(P_k, a) = \begin{cases} \sigma(P_{k+1}) & \text{if } a = p_{k+1} \\ \sigma(P_1) & \text{if } a = p_1 \\ \sigma(P_0) & \text{otherwise} \end{cases}$$

We justify each of these cases in turn:

- $\delta(P_k, a) = \sigma(P_{k+1})$ if $a = p_{k+1}$: Note by CLRS definition 32.4, $delta(P_k, p_{k+1}) = \sigma(P_k p_{k+1}) = \sigma(P_{k+1})$.
- $\delta(P_k, a) = \sigma(P_1)$ if $a = p_1$: We show this by contradiction. If, for some $k > 1$, $\delta(P_k, p_1) \neq \sigma(P_1)$, then $\delta(P_k, p_1) = P_j = p_1 \cdots p_{j-1} p_1$ (noting that obviously $j \geq 1$, and by assumption $j \neq 1$). But then p_1 is not unique in P, yielding a contradiction. Hence $\delta(P_k, p_1) = \sigma(P_1)$.
- $\delta(P_k, a) = \sigma(P_0)$ otherwise: Again we show this by contradiction. If, for some k and some $a \neq p_1$ or p_{k+1}, $\delta(P_k, a) = \sigma(P_k a) = \sigma(P_j)$ for some $j > 1$, then we must have $P_{j-1} \sqsupset P_k$, yielding our contradition. Hence $\delta(P_k, a) = \sigma(P_0)$ for all $a \neq p_{k+1}, p_1$.

6 Prefix function

Below is the prefix function π for the pattern *ababbabbabbababbabb*.

Prefix	q **(i.e. input to prefix function** π**)**	$\pi[q]$
a	1	0
ab	2	0
aba	3	1
$abab$	4	2
$ababb$	5	0
$ababba$	6	1
$ababbab$	7	2
$ababbabb$	8	0
$ababbabba$	9	1
$ababbabbab$	10	2
$ababbabbabb$	11	0
$ababbabbabba$	12	1
$ababbabbabbab$	13	2
$ababbabbabbaba$	14	3
$ababbabbabbabab$	15	4
$ababbabbabbababb$	16	5
$ababbabbabbababba$	17	6
$ababbabbabbababbab$	18	7
$ababbabbabbababbabb$	19	8

7 Linear-time algorithm to determine if T is a cyclic rotation of T'

We give a linear-time algorithm to determine whether a text T is a cyclic rotation of another string T'. For example, arc and car are cyclic rotations of each other.

function CYCLIC_ROTATION(T, T')
  $T'' = T + T$ ▷ + is the string concatenation operator
  `KMP-Matcher`(T'', T')

Note this is a linear-time algorithm, since

- String concatenation is assumed to be a linear time operation $O(n)$
- Inside `KMP-Matcher`, `compute-prefix-function`(T) runs in $O(n)$ time
- The rest of `KMP-Matcher` runs in $O(2n) = O(n)$ time

Moreover, this is algorithm is clearly correct, since if T and T' are cyclic rotations of each other, then T' is a substring of $T" = T + T$, so the algorithm finds the cyclic rotation required to match the strings.

8 Longest palindromic substring

The longest palindromic substring is a maximum-length contiguous substring of a given string that is a palindrome. For example, the longest palindromic substring of *ultramarine* is *ramar*.

We give an efficient algorithm to determine the longest palindromic substring of a given string, then we explain the algorithm and illustrate its operation on the string *evenness*. First, we present the algorithm (Manacher's Algorithm). The basic idea of this algorithm is to iterate from left to right in (a preprocessed version) of the input string. As we iterate, we store information about previously seen palindromes. In particular, we keep track of the known palindrome that extends furthest to the right. By storing this information, we can exploit the inherent symmetry of palindromes to reduce the total number of comparisons necessary to identify the longest palindromic substring, and achieve a runtime of $O(n)$. The algorithm is:

```
function LONGEST_PALINDROME(S)
    S2 = preprocess(S)
    P = array of zeros with length(S2)
    C = 1                                  ▷ Position of current palindrome center
    R = 1                      ▷ Position of right boundary of current palindrome
    for i = 2 to length(S2) - 1 do
        m = 2C - 1      ▷ Position of mirror element across current palindrome center
        if i < R then                                ▷ Case 1- see explanation
            if R - i > P[m] then                    ▷ Case 1a - see explanation
                P[i] = P[m]
            else                                    ▷ Case 1b- see explanation
                P[i] = R - i
                while S2[i - (1 + P[i])] == S2[i + (1 + P[i])] do
                    P[i] ++
                if i + P[i] > R then
                    C = i
                    R = i + P[i]
        else                                         ▷ Case 2- see explanation
            while S2[i - (1 + P[i])] == S2[i + (1 + P[i])] do
                P[i] ++
            if i + P[i] > R then
                C = i
                R = i + P[i]
    return palindrome_from_span(S2, P)
```

Before giving pseudocode for the helper functions `palindrome_from_span` and `preprocess`, we explain the logic behind this algorithm:

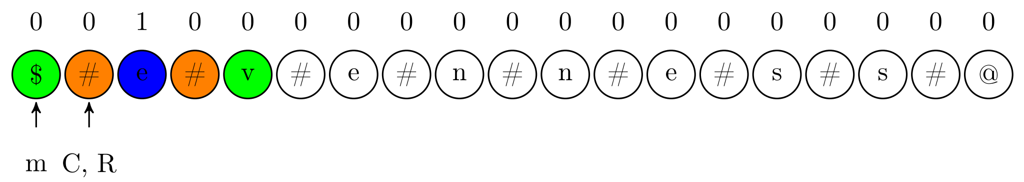
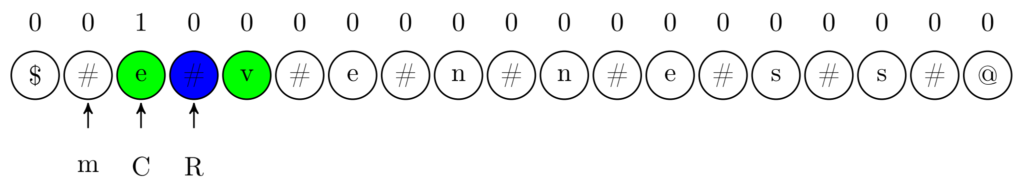
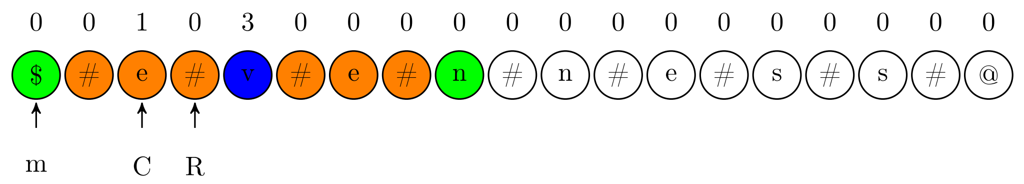
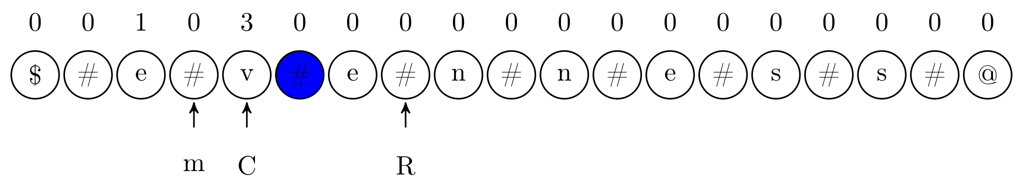
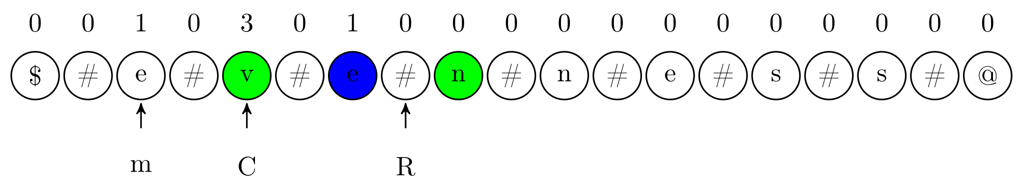
- **Case 1a**: In this case, by symmetry, P[i] = P[m]. This is because the palindromes centered on `m` (and by symmetry on `i`) are both fully contained within the palindrome centered on `c`, and thus are identical.

- **Case 1b**: In this case, the palindrome centered on `m` goes at least to the left edge of the current palindrome (centered on `c`). Hence, by symmetry, the palindrome centered on `i` extends as least as far as `R`, but it may extend further. Thus, we need to make additional comparisons (past `R`), and potentially update `C` and `R` (if we find a palindrome that extends past `R`).

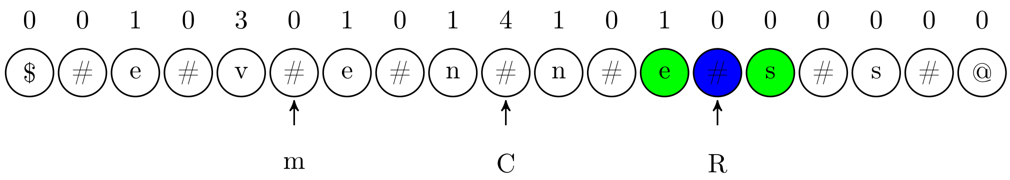
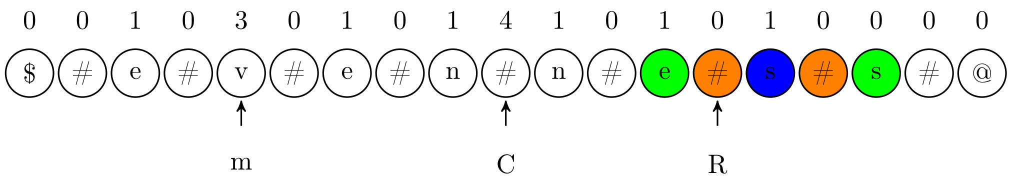
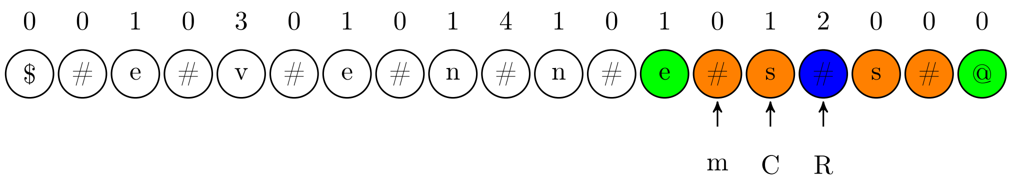
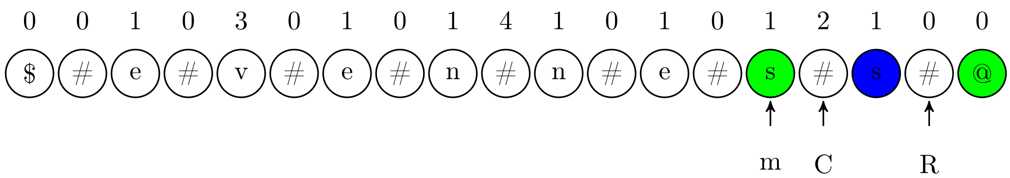
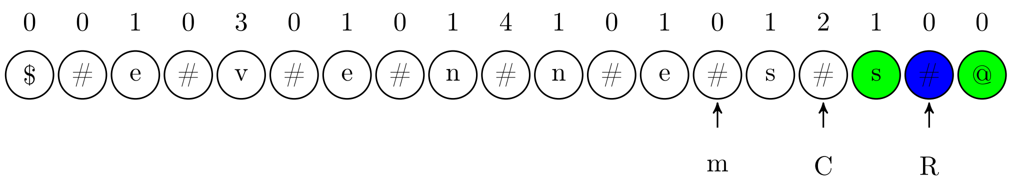
- **Case 2**: In this case, `i` is outside the current palindrome (past `R`) and thus we don't know anything about `P[i]`. Thus we immediately begin making comparisons and potentially update `C` and `R`.

Now that we've presented and explained the main algorithm, we present psuedocode for the two helper functions, then illustrate the operation of the algorithm on the string *evenness*.

```
function PREPROCESS(S)
    S = insert '#' between every character in S
    S2 = '$#' + S + '@'
    return S2
function PALINDROME_FROM_SPAN(S2, P)
    max_span = max(P)
    max_center = index of max_span in P
    start = max_center - max_span + 1          ▷ +1 since palindrome starts with '#'
    stop = max_center + max_span
    result = new String
    next_char = start
    while next_char < stop do
        result = result + S2[next_char]        ▷ Add next_char to result
        next_char = next_char + 2
    return result
```

<table>
<tr><td><code>i</code></td><td>Diagram showing algorithm operation.
The pointers for <code>C,R, m</code>, and the current position <code>i</code> shown in blue indicate the state when the <code>for</code> loop is entered.
The <code>while</code> loop comparisons that match are shown in orange, and those that don't match are shown in green.
The final incremented palindrome length <code>P[i]</code> (i.e. the state when control leaves the <code>while</code> loop) is drawn above.</td></tr>
<tr><td>2</td><td>
</td></tr>
<tr><td>3</td><td>
</td></tr>
<tr><td>4</td><td>
</td></tr>
<tr><td>5</td><td>
</td></tr>
<tr><td>6</td><td>
</td></tr>
</table>

7	0	0	1	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	$	#	e	#	v	#	e	#	n	#	n	#	e	#	s	#	s	#	@
		m			C			R											
8	0	0	1	0	3	0	1	0	1	0	0	0	0	0	0	0	0	0	0
	$	#	e	#	v	#	e	#	n	#	n	#	e	#	s	#	s	#	@
	m				C			R											
9	0	0	1	0	3	0	1	0	1	4	0	0	0	0	0	0	0	0	0
	$	#	e	#	v	#	e	#	n	#	n	#	e	#	s	#	s	#	@
								m	C	R									
10	0	0	1	0	3	0	1	0	1	4	1	0	0	0	0	0	0	0	0
	$	#	e	#	v	#	e	#	n	#	n	#	e	#	s	#	s	#	@
									m	C				R					
11	0	0	1	0	3	0	1	0	1	4	1	0	0	0	0	0	0	0	0
	$	#	e	#	v	#	e	#	n	#	n	#	e	#	s	#	s	#	@
								m		C				R					
12	0	0	1	0	3	0	1	0	1	4	1	0	1	0	0	0	0	0	0
	$	#	e	#	v	#	e	#	n	#	n	#	e	#	s	#	s	#	@
							m			C				R					

13	0 0 1 0 3 0 1 0 1 4 1 0 1 0 0 0 0 0 0 $ # e # v # e # n # n # e # s # s # @ m C R
14	0 0 1 0 3 0 1 0 1 4 1 0 1 0 1 0 0 0 0 $ # e # v # e # n # n # e # s # s # @ m C R
15	0 0 1 0 3 0 1 0 1 4 1 0 1 0 1 2 0 0 0 $ # e # v # e # n # n # e # s # s # @ m C R
16	0 0 1 0 3 0 1 0 1 4 1 0 1 0 1 2 1 0 0 $ # e # v # e # n # n # e # s # s # @ m C R
17	0 0 1 0 3 0 1 0 1 4 1 0 1 0 1 2 1 0 0 $ # e # v # e # n # n # e # s # s # @ m C R