Readings about Strings and Tries

Diego Linares - kiwiAipom

I. Manacher's Algorithm

Statement: Find all pairs (i, j) which make a palyndrome substring.

A. Detailed Statement

Worst case we would have $O(n^2)$ palyndromes.

Compact way of keeping palyndromes: $i = 0 \dots n-1$, $d_1[i], d_2[i]$ represent the number of palyndromes with odd and even length with their center in i.

Ex. *abababc*, has $d_1 = 3$ (odd length), and *cbaabd* has $d_2[3] = 2$ (even length).

Sub-palindrome with l size with center in i, we also have with $l-2, l-4, \dots$.

Both palyndrom arrays can be calculated in linear time.

Solution: Can be done with string hashing and suffix trees, but this has a smaller constant and memory complexity.

B. Trivial Algorithm

Tries to increase the answer by one until it's possible for each center i. It is $O(n^2)$ in time. Implementation being:

```
vector<int> d1(n),  d2(n);
for (int i = 0; i < n; i++) {
    d1[i] = 1; // Pair with itself
    while (0 <= i - d1[i] && i + d1[i] < n &&
        s[i - d1[i]] == s[i + d1[i]]) {
        d1[i]++;
    }

    d2[i] = 0;
    while (0 <= i - d2[i] - 1 && i + d2[i] < n
         && s[i - d2[i] - 1] == s[i + d2[i]])
        {
        d2[i]++;
    }
}
```

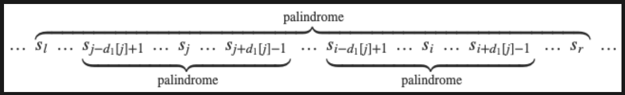
C. Manacher's Algorithm

Allows to find all the sub-palyndromes with odd length (even length is just a small mod).

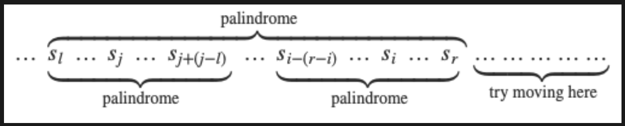
Borders: (l, r) of the palyndrome with maximal r. Initially we assume $l = 0, r = -1$

We want to calculate $d_1[i]$ with all the previous i already calculated.

- If it is ourside of the current rightmost sub-palyndrome $i > r$, we launch the trivial algorithm, allowing us to calculate $d_i[i]$ and updating (l, r).
- $i \leq r$. We can flip i into $j = l + (r - i)$ and since they are symmetrical, we can **almost always** do $d_1[i] = d_1[j]$. Like so:


palindrome
$\dots s_l \dots s_{j-d_1[j]+1} \dots s_j \dots s_{j+d_1[j]-1} \dots s_{i-d_1[j]+1} \dots s_i \dots s_{i+d_1[j]-1} \dots s_r \dots$
palindrome
palindrome

- **Trick case:** When the inner palyndrome reaches the border of the outer one. $j - d_1[j] + 1 \leq l$. Since the symmetry of the outer palyndrome is not guaranteed, we assign $d_1[i] = r - i + 1$, and then run the trivial algorithm to increase the value if necessary, updating in the end. Like so:


palindrome
$\dots s_l \dots s_j \dots s_{j+(j-l)} \dots s_{i-(r-i)} \dots s_i \dots s_r \dots\dots\dots\dots\dots$
palindrome
palindrome
try moving here

Note we only use the part of the palyndrome with guaranteed symmetry before moving to the *try moving here part*.

A similar algorithm is then used for the even part.

D. Complexity

Not as intuitive, but if we look at *Z-function building algorithm* we can check it's similar and linear.

We can also notice that r can only increase by one ine very iteration of the trivial algorithm, and that it can't decrease. $O(n)$ iterations. The rest is linear.

E. Implementation

Here is the implementation, which is fairly similar for both d_1, d_2.

```
vector<int> d1(n), d_2(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 1 : min(d1[l + r - i], r
         - i + 1);
    while (0 <= i - k && i + k < n && s[i - k]
         == s[i + k]) {
        k++;
    }
    d1[i] = k--;
    if (i + k > r) {
        l = i - k;
        r = i + k;
    }
}
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 0 : min(d2[l + r - i +
        1], r - i + 1);
    while (0 <= i - k - 1 && i + k < n && s[i
        - k - 1] == s[i + k]) {
        k++;
    }
    d2[i] = k--;
    if (i + k > r) {
        l = i - k - 1;
        r = i + k ;
    }
}
```

II. Z-function

String s of length n. The function returns an array of length n where $Z[i]$ is equal to **greatest number of characters starting from position i, that coincide with the first characters of s**. Longest common prefix between s, and suffix of s starting at i.

We assume 0 based indexing and $z[0] = 0$. It is $O(n)$ time.

Ex. $aaaaa = [0, 4, 3, 2, 1], aaabaab = [0, 2, 1, 0, 2, 1, 0], abacaba = [0, 0, 1, 0, 3, 0, 1]$.

A. Trivial Algorithm

This one is $O(n^2)$:

```
vector<int> z_function_trivial(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1; i < n; ++i)
        while (i + z[i] < n && s[z[i]] == s[i
            + z[i]])
            ++z[i];
    return z;
}
```

Basically calculating $z[i]$ for each i independently.

B. Efficient

We make use of the previously computed values.

Segment Match: a substring which coincides with a prefix of s. So $z[i]$ is the length of the segment match that starts at i at ends at $i + z[i] - 1$.

(l, r) will be the indices of our rightmost segment match. r can be seen as the boundary to which our string has been scanned by the algorithm.

If we are in index i, there are two options to calculate:

- $i > r$, so we haven't processed the current position yet. So we run the trivial algorithm. If $z[i] > 0$ we have to update $r = i + z[i] - 1$.
- Otherwise, we are inside the current segment match. We can use the previou values to give us a head start, through the value $z[i - l]$, since $sl..r$ and $s[0..l - r]$ match.
  However, the value might be too large, and when applied to i we could get an overflow. For example $aaaabaa$, in $i = 6$ we are not able to initialize it with $z[1] = 3$ since we would overflow the array. So we instead do:
  $$z_0[i] = min(r - i + 1, z[i - l])$$
  Then we increment $z[i]$ with the trivial algorithm, to see if it continues to match or not.

The only difference between the cases is the initial value of $z[i]$, then both branches use the trivial algorithm.

C. Implementation

```
vector<int> z_function(string s) {
  int n = (int) s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r)
      z[i] = min (r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[
        i]])
      ++z[i];
    if (i + z[i] - 1 > r)
      l = i, r = i + z[i] - 1;
  }
  return z;
}
```

Returns an array of size n.

The initial array is initially filled with 0 and the rightmost match is $[0, 0]$. We then first check what option of the algorithm we are gonna use, and in the end if necessary, we update the rightmost segment $(i + z[i] - 1 > r)$.

D. Asymptotic Behavior

It's $O(n)$, and we only need to focus on the `while` loop, showing that each iteration will increase the right border r. So we consider both branches of the algorithm.

- $i > r$: If $s[0] \neq s[i]$ it won't make any iterations, otherwise it will necessarily move to the right and update r.
- $i \leq r$. We initialize $z[i]$ to z_0, and we need to compare this to $r - i + 1$ so we have a trichotomy:
  - $z_0 < r - i + 1$: No iteration of the loop will take place. If it did, the initial z_0 approximation was inaccurate. But that is not true because we know $s[l..r]$ and $s[0...r - l]$ match.
  - $z_0 = r - i + 1$: In this case, it will make a few iterations, but each will lead to an increase of r since we will start comparing from $s[r + 1]$ increasing the (l, r) interval.
  - $z_0 > r - i + 1$: By definition, can't happen.

Each iteration of the inner loop makes r increase, and at most $n - 1$ times, since it won't be able to overflow the array.

The rest of the algorithm clearly runs in $O(n)$ so it is proven.

E. Applications

Very similar to the prefix function ones.

1) Search the Substring: Find all the ocurrences of a pattern p inside a text t.

We create a new string $p + \alpha + t$, α being a separator character we are sure won't appear.

Compute the function for this string, then for every i in the range $[0; length(t) - 1]$ consider the corresponding value.

$$k = z[i + length(p) + 1]$$

If k is equal to $length(p)$ we know there is one ocurrence in the i position of the text.

Running time and memory consumption of $O(length(t) + length(p))$

2) Number of distinct substrings: **Iterative approach:** Knowing the current number of different substrigns, recalculate after adding the end of s one character.

k being the current number, we add c to s. There can be new substrings which end in c (all the ones that end with this symbol and we haven't encountered yet).

Take a string $t = s + c$ and invert it. So, now we have to count how many prefixes in t are not found anywhere else in t.

Compute the Z-function of t and find its max value, z_{max}. The number of new substrings that appear when symbol c is appended to s is equal to:

$$length(t) - z_{max}$$

Running time being $O(n^2)$ We can recalculate in $O(n)$ time the amount of substrings when adding at the beginning and removing.

3) String Compression: In s find a string t of shortest length, such that s can be represented as concatenations of t.

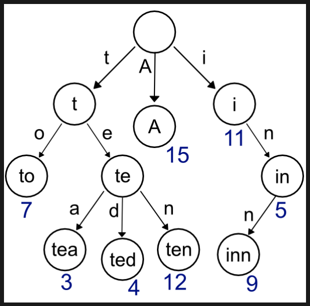
Z-function solution: compute de function of s loop through all i which divide n, stop at the first i such that $i + z[i] = n$. The string then can be compressed to length i.

Can also be done with prefix function, check that for the proof.

III. Tutorial on Trie

Useful in bit manipulation problems.

A trie stores information about keys/numbers/strings in a tree, each node storing a character or bit.



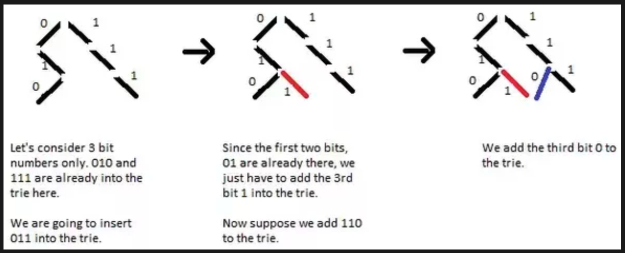
A. Problem 1

Given an array of integers, find two elements with maximum XOR.

Let's have an structure that satisfies queries:

- Insert a number
- Given a number, find the one that gives maximum XOR.

Inserting elements into the trie is fairly simple.


Let's consider 3 bit numbers only. 010 and 111 are already into the trie here.

We are going to insert 011 into the trie.

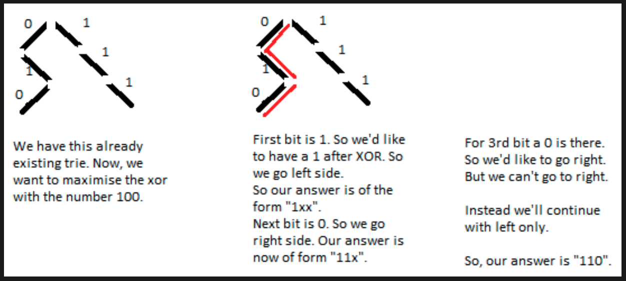
Since the first two bits, 01 are already there, we just have to add the 3rd bit 1 into the trie.

Now suppose we add 110 to the trie.

We add the third bit 0 to the trie.

This means insertion time is $O(n) = \lg max$.

For the second query we have $Y = b_1, b_2, ...$ each of them being a bit. For the XOR maximum,we try to make the most significant bit 1, after taking XOR. If $b_1 = 0$ we'll need a 1 and viceversa. We go to the required bit side, unless it is not possible. And we will do this for all $i \ldots n$ values. Like so:


We have this already existing trie. Now, we want to maximise the xor with the number 100.

First bit is 1. So we'd like to have a 1 after XOR. So we go left side.
So our answer is of the form "1xx".
Next bit is 0. So we go right side. Our answer is now of form "11x".

For 3rd bit a 0 is there. So we'd like to go right. But we can't go to right.

Instead we'll continue with left only.

So, our answer is "110".

Note that the answer is 110 because that is the value of the XOR, while the path is the number we use.

The complexity is $\lg max$

B. Problem 2

Find the maximum XOR subarray.

$F(L, R)$ is the XOR of the subarray. And we have the following property:

$$F(L, R) = F(1, R) \oplus F(1, L - 1)$$

Let's say our subarray with maximum XOR ends at i. We need to maximize $F(L, i) = F(1, i) \oplus F(1, L - 1)$. If we inserted $F(1, L - 1)$ in our trie for all $L \leq i$, it is an iteration of problem 1:

```
ans=0
pre=0
Trie.insert(0)
for i=1 to N:
  pre = pre XOR a[i]
  Trie.insert(pre)
  ans=max(ans, Trie.query(pre))
print ans
```

C. Problem 3

Print the number of subarray whose XOR is less than k.

For each index $i = 1 \ldots n$ we can count how many subarrays which end at i satisfy the condition:

```
ans=0
p=0
for i=1 to N:
  q=p XOR A[i]
  ans += Trie.query(q,k)
  Trie.insert(q)
  p=q
```

`query(q,k)` returns how many integers I in the structure which $q \oplus I < k$.

Compare corresponding bits of q, k, starting from most significant.

Ex. Suppose you are comparing a, b, then we do:

- In the root node we store how many numbers we could reach.

- If $a = 1, b = 1$, for example, the XOR with right side will give us 0. Less than k, so all the nodes on that side are marked as valid.
- Plus we go down further from left side because we still can encounter lesser XORs even if current bit is same as the current bit of k.

We need to keep the number of leaf nodes reachable from current node, to reduce the complexity (traversing the tree over and over).

The implementation of the insertion with pointers would be:

```
insert(root, num, level):
  if level==-1: return root
  x=level'th bit of num
  if x==1:
    if root->right is NULL: create root->right
    else: insert(root->right,num,level-1)
  else:
    if root->left is NULL: create root->left
    else: insert(root->left,num,level-1)
```

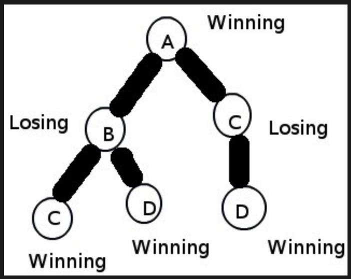
For queries we recursively traverse the tree.

D. Problem 4 - A subproblem

n non-empty strings. Players move in turns, with an empty word, adding a single letter in the end of the word. The resulting one must be a prefix of one of the strings of the initial group.

Build a trie of all the strings since you have all the prefixes there.

How to know if a player has a winning strategy? A node v has immediate children u. If the first player has a losing strategy for u, it has a winning one for v. Example for $[''abc'', ''abd'', ''acd'']$.



IV. Prefix and Knuth-Morris-Pratt

A. Definition

With string s of length n. The **prefix function** is an array π of the same size, where $\pi[i]$ is the length of the longest prefix (that is not the whole string) of $s[0 \dots i]$ that is also a suffix. By definition $\pi[0] = 0$.

$$\pi[i] = max_{k=0..i}\{k : s[0..k-1] = s[i-(k-1)...i]\}$$

Ex. $abcabcd = [0, 0, 0, 1, 2, 3, 0]$.

B. Trivial Algorithm

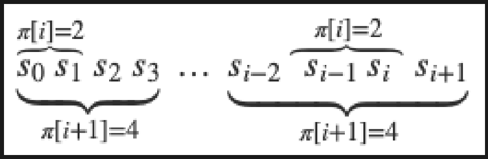
Implementation with $O(n^3)$ complexity:

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 0; i < n; i++)
    for (int k = 0; k <= i; k++)
      if (s.substr(0, k) == s.substr(i-k+1, k)
        )
        pi[i] = k;
  return pi;
}
```

C. Efficient Algorithm

1) First Optimization: The values of the prefix function can increase by 1 at most.

Proof: By contradiction, we could take the suffix ending in $i + 1$, with the length $\pi[i + 1]$ and remove its last character. Which would result in something better than $\pi[i]$. Like so:

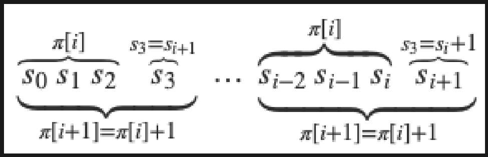


We can see that supposedly $\pi[i] = 2, \pi[i + 1] = 4$. So $s_0s_1s_2s_3 = s_{i-2}s_{i-1}s_is_{i+1}$, so $s_0s_1s_2 = s_{i-2}s_{i-1}s_i$, so $\pi[i] = 3$ really.

So the value can only vary by one, so our complexity reduces to $O(n^2)$, since it can grow/decreease at most n, performin $O(n)$ string comparisons.

2) Second Optimization: Use information computed in previous steps.

Supposing we want to compute $i + 1$. If $s[i + 1] = s[\pi[i]]$ then $\pi[i + 1] = \pi[i] + 1$, since the suffix at i of length $\pi[i]$ is equal to prefix of the same length. Like so:



If $s[i + 1] \neq s[\pi[i]]$, we need to move to a shorter string, or the longest j such that $s[0..j - 1] = s[i - j + 1...i]$.

Then we compare $s[i+1], s[j]$ if they are equal, then $\pi[i+1] = j+1$. Otherwise we need to find the largest value smaller than j that holds this property, which can go all the way to $j = 0$, if this happens:

- $s[i + 1] = s[0]$, then $\pi[i + 1] = 1$
- $\pi[i + 1] = 0$ otherwise.

The only remaining apr is to find the lengths for j. It has to be the value of $\pi[j - 1]$ which we have calculated earlier.

3) Final Algorithm: Doesn't perform string comparisons, only $O(n)$ operations.

- We compute the prefix values $\pi[i]$ in a loop from $i = 1..n - 1$ ($\pi[0] = 0$)
- To alculate it, we set the variable j denoting the length of the best suffix for $i - 1$. Initially $j = \pi[i - 1]$
- $s[j] = s[i]$? to test if the suffix of length $j + 1$ is also a prefix, if it is $\pi[i] = j + 1$, otherwise $j = \pi[j - 1]$ and repeat.

- If we reach 0 and still don't have a match, just go with 0.

```
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i-1];
    while (j > 0 && s[i] != s[j])
      j = pi[j-1];
    if (s[i] == s[j])
      j++;
    pi[i] = j;
  }
  return pi;
}
```

It processes the data as it arrives (so you can read the string one by one). If we know beforehand the max value M the prefix function can take on the string, we can only store $M+1$ characters of the string and the values of the function.

D. Applications

1) Search for a substring in a string: Display all the positions of all occurrences of the string s (size n) in the text t (size m).

Generate $s + \# + t$ (# being a separator). We get the prefix function, and we care about the values after the first $n+1$ entries. If $\pi[i]$ shows the longes length of a substring ending in i that coincides with the prefix, in our case it means the largest block that coincides with s and ends at i. If $\pi[i] = n$ appears, it means that the string s appears completely in this position (**it ends on it**).

In simpler terms, if $\pi[i] = n \rightarrow i - 2n$ is the position in the string t in which s appears.

A further optimization is not storing the whole string and function, but only $s+\#$ and its prefix function. And then read a character at a time for t. $O(n+m)$ time.

2) Number of ocurrences of each prefix: Count the appearances of $s[0..i]$ in the same string. Another variation is finding them in another string t.

First problem: Consider $\pi[i]$, remember this means that the prefix of that length appears and ends on that i. We can have shorter prefixes that end at this position. Which is kind of a question we already answered: Given a prefix of size j, what is the next smallest suffix that also ends at i. We can obtaine this like: $\pi[\pi[i]-1], \pi[\pi[\pi[i]-1]-1], \ldots$. Implemented like:

```
vector<int> ans(n + 1);
for (int i = 0; i < n; i++)
  ans[pi[i]]++;
for (int i = n-1; i > 0; i--)
  ans[pi[i-1]] += ans[i];
for (int i = 0; i <= n; i++)
  ans[i]++;
```

First we count how may times it occurs in the array π, and then compute. If the length prefix i appears exactly $ans[i]$ times, this number must be added to the number of occurences of its longest suffix that is also a prefix (what...).

3) Number of different substrings in a string: Iterative solution: knowing the current number of different substrings, how to recompute by adding a character to the end.

k be the current value, and we add c. We now make $t = s+c$ and reverse it. Peform the prefix function and obtain π_{max}, which is the longest string which appears in s. The number of new substrings will be $|s| + 1 - \pi_{max}$.

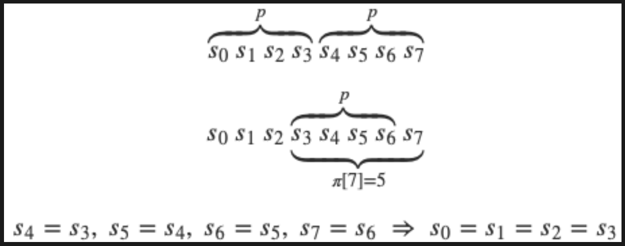
Since this is done for each character, it's $O(n^2)$. Can also be done with adding at the beginning, or deleting in both ends.

4) Compressing a string: Smallest string t that can represent s through concatenation of its copies. We only need the length, since t will be a prefix.

Compute the prefix function. $k = n - \pi[n-1]$, if k divides n, it is the answer, otherwise the answer is n.

Is it the optimum? Well if it wasn't, the prefix function at the end would be bigger than $n-k$.

And in the cases it is n? Well suppose there's an answer p which divides n. The value of the prefix function would be greater than $n-p$. The contradiction illustration is like so:



$\underbrace{s_0\ s_1\ s_2\ s_3}_{p}\ \underbrace{s_4\ s_5\ s_6\ s_7}_{p}$
$s_0\ s_1\ s_2\ \overbrace{s_3\ s_4\ s_5\ s_6}^{p}\ s_7$
$\pi[7]=5$
$s_4 = s_3,\ s_5 = s_4,\ s_6 = s_5,\ s_7 = s_6 \Rightarrow s_0 = s_1 = s_2 = s_3$

5) Building an Automaton: Let's have $s+\#+t$, and due to the separtor, the prefix function $\leq |s|$. We only need to store $s+\#$ and compute the values of the prefix functions for it, and the part of t can be done on the go. We only need the next character $c \in t$ and the value of the prefix function of the previous position.

Automaton that has the value of the prefix function and transitions through the next character. The $O(n * 26)$ for the letters of the alphabet is the follwing:

```
void compute_automaton(string s, vector<vector
    <int>>& aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = prefix_function(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < 26; c++) {
      if (i > 0 && 'a' + c != s[i])
        aut[i][c] = aut[pi[i-1]][c];
      else
        aut[i][c] = i + ('a' + c == s[i]);
    }
  }
}
```

Accelerates calculating the prefix function for $s+\#+t$, by building this automaton we don't need to store $s+\#$, since we have its transitions.

A less obvious application is when the string is very big and constructed using some rules (like a recursive combination).

Like Gray strings:

$$\begin{aligned} g_1 &= a \\ g_2 &= aba \\ g_3 &= abacaba \\ g_4 &= abacabadabacaba \end{aligned} \tag{1}$$

The string grows quickly, but we only needs the value of the prefix function at the start.

We also compute $G[i][j]$, which is the value of processing the string g_i starting in j, and $K[i][j]$, the number of occurences of s in g_i, during the processing of g_i starting with j. $K[k][0]$ being our answer.

We start with $G[0][j] = j$ and $K[0][j] = 0$. The rest can be calculated with the automaton. Remember $g_i = g_{i-1} + alphabet[i] = g_{i-1}$:

$$mid = aut[G[i-1][j]][i]$$

$$G[i][j] = G[i-1][mid]$$

Then finally, to obtain $K[i][j]$:

$$= K[i-1][j] + (mid == |s|) + K[i-1][mid]$$

So, to solve these kind of problems we construct the automaton of the prefix function and then calculate the transitions in for each pattern by using the previous results. **Further Investigate on that.**

References

[1] *Manacher's Algorithm - Finding all sub-palindromes in $O(N)$*, jakobkogler for E-maxx. From: https://cp-algorithms.com/string/manacher.html

[2] *Z-function and its calculation*, E-maxx. From: https://cp-algorithms.com/string/z-function.html

[3] *Tutorial on Trie and example problems*, Lalit Kundo, From: https://www.quora.com/q/threadsiiithyderabad/Tutorial-on-Trie-and-example-problems

[4] *Prefix function. Knuth-Morris-Pratt Algorithm*, tcNickolas for E-maxx. From: https://cp-algorithms.com/string/prefix-function.html