KMP Search Algorithm

Here, I am trying to note down some important observations I made while reading the cp-algorithms.com post on building prefix function. It is the most important step in the KMP algorithm.

Link to CP algos blog on KMP

Link to problem on binarysearch

Naive Solution

```
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;
}
```

Time complexity: $O(n^3)$

First optimization:

As stated in the blog:

"The first important observation is, that the values of the prefix function can only increase by at most one."

- How does the code look like after using the first optimization?

```cpp
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n,0);

    for (int i = 1; i < n; i++){
        if( s[i] == s[pi[i-1]]){
            pi[i] = pi[i-1] + 1;
        }
        else{
            for(int size = pi[i-1]; size >= 1; size--){
                if(s.substr(i - size + 1, size) == s.substr(0,size)){
                    pi[i] = size;
                    break;
                }
            }
        }
    }
    return pi;
}
```

- This optimization restricts the range of the length of the prefix strings we need to compare at index i. Also, the string comparisons are made less frequently.
- In the naive code, we were comparing strings of all sizes in the range $[1, i]$. But using the optimization, now we need to compare strings of sizes in the range $[1, \pi[i-1]]$ only.
- Also, we note that the longest proper prefix ending at index i can be atmost $\pi[i-1]+1$. This happens when $s[\pi[i-1]] = s[i]$. So if this condition is true, we can simply set $\pi[i] = \pi[i-1]+1$. This is done in constant time !
- If $s[\pi[i-1]] \neq s[i]$ then we do comparisons of strings of sizes in the range $[1, \pi[i-1]]$ until we get a match.

The time complexity of the above approach is:
$O(a_1^2 + a_2^2 + a_2^2 \ldots + a_m^2)$ where $a_1 + a_2 + \ldots + a_m = n$.

- m is the number of times we enter the else part.
- a_1 is the number of iterations after which we run the else part for the first time.
- a_2 is the number of iterations after which we run the else part for the second time, after having run it for the first time i.e after a_1.
- And so on ...

Since, $a_1^2 + a_2^2 + a_2^2 \ldots + a_m^2 \leq n^2$ when $a_1 + a_2 + \ldots + a_m = n$, the time complexity of this approach is $O(n^2)$.

- By using $\pi[i-1]$, we can compute $\pi[i]$ in $O(1)$ time if $s[\pi[i-1]] = s[i]$. Otherwise, we will need $O(\pi[i-1]^2)$ time.'

Second optimization:

- In the previous approach, for computing $\pi[i]$, we were making use of the previous value $\pi[i-1]$.
- But it turns out that we can do much better if we make use of all the previous values from $1 \ldots i-1$.
- Read this section in the cp-algorithms post carefully.
- We can summarize it as follows:
  - If $s[\pi[i-1]] = s[i]$, then we simply set $\pi[i] = \pi[i-1] + 1$ just as we did previously.
  - If $s[\pi[i-1]] \neq s[i]$, we try to find the longest length j such that $j < \pi[i-1]$ and the prefix property still holds, i.e $s[0 \ldots j-1] = s[i-j \ldots i-1]$.
  - It turns out that this value of j is simply $\pi[\pi[i-1]-1]$!
  - So now we compare $s[i]$ and $s[j]$ where $j = \pi[\pi[i-1]-1]$.
  - We keep on doing this until we get a match and until $j > 0$.
  - Going through the code will make it more clear.

Final Code:

```
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;
}
```