1910. Remove All Occurrences of a Substring

Given two strings `s` and `part`, perform the following operation on `s` until **all** occurrences of the substring `part` are removed:

- Find the **leftmost** occurrence of the substring `part` and **remove** it from `s`.

Return `s` *after removing all occurrences of* `part`.

A **substring** is a contiguous sequence of characters in a string.

Example 1:

```
Input: s = "daabcbaabcbc", part = "abc"
Output: "dab"
Explanation: The following operations are done:
- s = "daabcbaabcbc", remove "abc" starting at index 2, so s = "dabaabcbc".
- s = "dabaabcbc", remove "abc" starting at index 4, so s = "dababc".
- s = "dababc", remove "abc" starting at index 3, so s = "dab".
Now s has no occurrences of "abc".
```

Example 2:

```
Input: s = "axxxxyyyyb", part = "xy"
Output: "ab"
Explanation: The following operations are done:
- s = "axxxxyyyyb", remove "xy" starting at index 4 so s = "axxxyyyb".
- s = "axxxyyyb", remove "xy" starting at index 3 so s = "axxyyb".
- s = "axxyyb", remove "xy" starting at index 2 so s = "axyb".
- s = "axyb", remove "xy" starting at index 1 so s = "ab".
Now s has no occurrences of "xy".
```

Constraints:

- `1 <= s.length <= 1000`
- `1 <= part.length <= 1000`
- `s` and `part` consists of lowercase English letters.

1910. Remove All Occurrences of a Substring

```
/*
    In place: Straight forward simulation built-in functions
  Time complexity:   O(n^2)
  Space complexity:  O(1)
  n=|s|,m=|part|
*/
class Solution {
  public:
    std::string removeOccurrences(std::string& s, std::string& part){
      // While part exists in s
      std::size_t pos=s.find(part);
      while(pos!=std::string::npos){
        // Erase it
        s.erase(pos,part.size());
        pos=s.find(part);
      }
      return s;
    }
};
```

Complexity Analysis

Let n be the length of the string s and m be the length of the substring `part`.

- Time complexity: $O(n^2)$

  The algorithm uses a `while` loop to repeatedly remove the leftmost occurrence of `part` from `s`. Each iteration of the loop involves finding the index of `part`, which takes $O(n \cdot m)$ time, and then creating a new string by concatenating the segments before and after `part`, which takes $O(n)$ time. In the worst case, there are $O(\frac{n}{m})$ such iterations (e.g., when `part` is non-overlapping and removed sequentially). The total time across all iterations is $O((n \cdot m) \cdot (\frac{n}{m})) = O(n^2)$.

- Space complexity: $O(n)$

  Although the algorithm does not explicitly use additional data structures, each iteration creates a new string by concatenating the segments before and after part. This results in the creation of intermediate strings, each of size up to $O(n)$. The space required to store these intermediate strings dominates the space complexity, leading to $O(n)$ space usage.

1910. Remove All Occurrences of a Substring

```
/*
   Stack-Like
   Time complexity:   O(n*m)
   Space complexity:   O(1), if output not counted
                       O(n), if output counted
   n=|s|,m=|part|
*/

class Solution {
   public:
      std::string removeOccurrences(std::string s, std::string part) {
         int n=s.size();
         int m=part.size();

         std::string ans;
         // For each character in s
         for(int i=0;i<n;++i){
            ans.push_back(s[i]); // Add it to the answer
            // If answer's length is greater than part's length
            if(ans.size()>=m){
               // If the m last characters of ans are equal to part
               if(ans.substr(ans.size()-m,m)==part){
                  // Erase them from the answer
                  for(int j=0;j<m;++j) ans.pop_back();
               }
            }
         }

         return ans;
      }
};
```

1910. Remove All Occurrences of a Substring

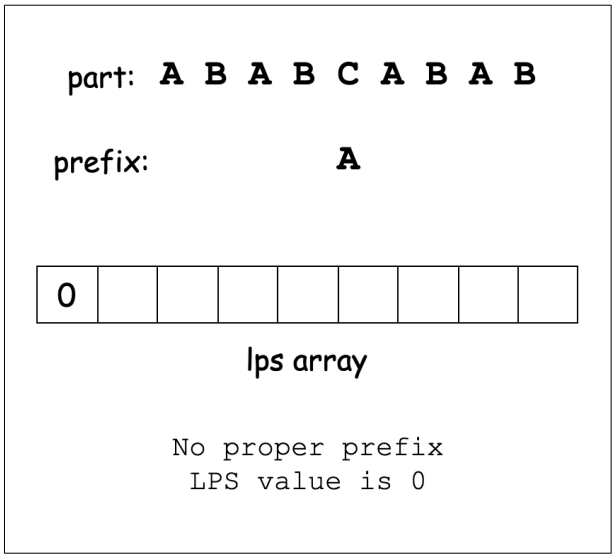
Approach 3: Knuth-Morris-Pratt (KMP) Algorithm

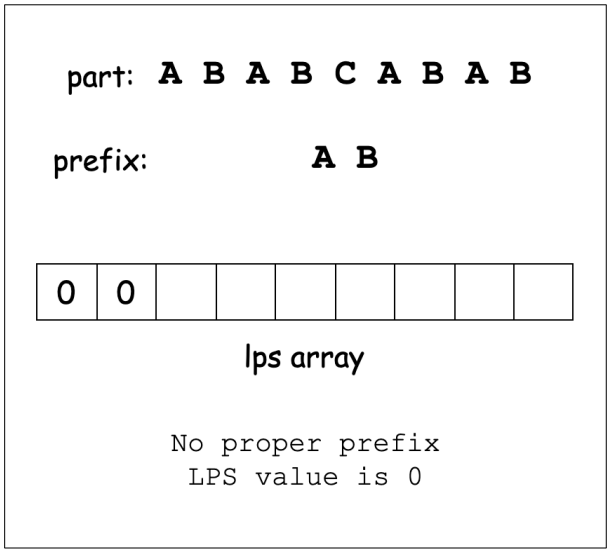
So far, we have relied on a naive approach for pattern matching, where we slide the pattern (`part`) over the string (`s`) one character at a time and check for a match. For example, if `s = "ABABDABACDABABCABAB"` and `part = "ABABCABAB"`, the naive approach compares `part` with every substring of `s` of the same length, often rechecking characters unnecessarily. Consider the scenario where the first four characters, `"ABAB"`, match, but a mismatch occurs with the fifth character. In the naive approach, the pattern is shifted by just one character, and the comparison restarts from the beginning of `part`, rechecking `"BAB"` again. This results in redundant comparisons and inefficiency.

The Knuth-Morris-Pratt (KMP) algorithm optimizes this by using a longest prefix-suffix (LPS) array for the pattern. The LPS array helps determine how much of the pattern has been matched so far, allowing the algorithm to skip redundant comparisons. When a mismatch happens, instead of starting over from the beginning, we use the LPS array to shift the pattern by an appropriate amount.

For example, if we've matched `"ABABC"` but encounter a mismatch at the 6th character, the LPS value for `"ABABC"` is 1. We then shift the pattern by 4 characters (5 – 1) and continue matching. This avoids rechecking parts of the pattern we've already matched.

For example, consider the pattern `part = "ABABCABAB"`. Let's see how we build up the LPS array in the slideshow below:





part: **A B A B C A B A B**

prefix: **A B A**

0	0	1						

lps array

proper prefix **A** is also suffix
LPS value is 0

part: **A B A B C A B A B**

prefix: **A B A B**

0	0	1	2					

lps array

proper prefix **AB** is also suffix
LPS value is 2

part: **A B A B C A B A B**

prefix: **A B A B C**

0	0	1	2	0				

lps array

no proper prefix that is also suffix
LPS value is 0

part: **A B A B C A B A B**

prefix: **A B A B C A**

0	0	1	2	0	1			

lps array

proper prefix **A** is also suffix
LPS value is 1

part: **A B A B C A B A B**

prefix: **A B A B C A B**

0	0	1	2	0	1	2		

lps array

proper prefix **AB** is also suffix
LPS value is 2

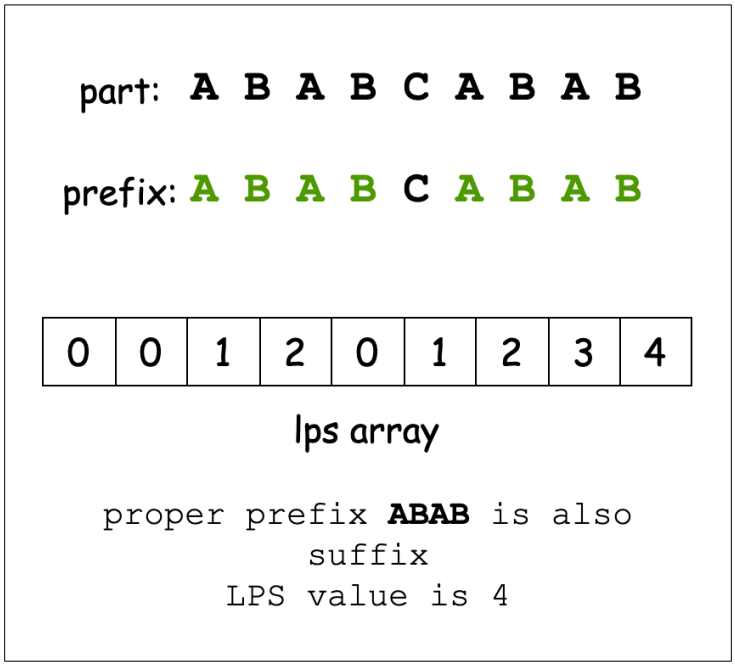
part: **A B A B C A B A B**

prefix: **A B A B C A B A**

0	0	1	2	0	1	2	3	

lps array

proper prefix **ABA** is also suffix
LPS value is 3



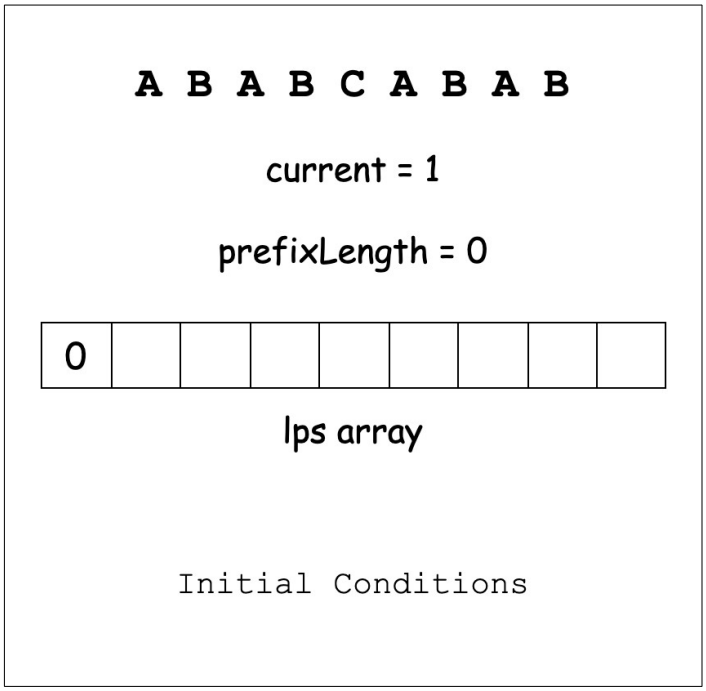
The LPS array allows the KMP algorithm to skip unnecessary comparisons when a mismatch occurs. When a mismatch happens, instead of starting over from the beginning of the pattern, the algorithm uses the LPS array to determine how much of the pattern has already been matched. It then shifts the pattern by an appropriate amount and continues matching.

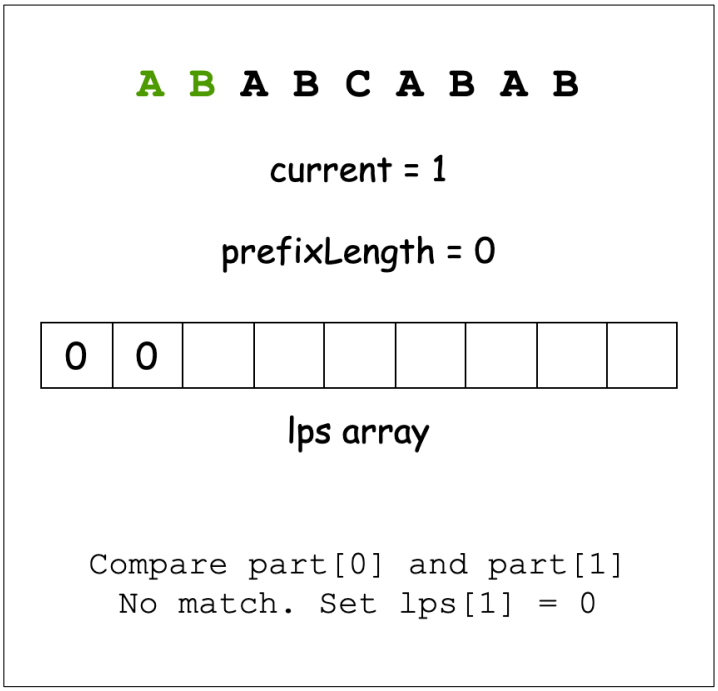
For example, let's say we're matching `part = "ABABCABAB"` against `s = "ABABDABACDABABCABAB"`. Suppose we've matched the first 4 characters (`"ABAB"`) but encounter a mismatch at the 5th character. The LPS value for the prefix `"ABAB"` is `2`, so we know that the first 2 characters of the pattern are already matched. Instead of starting over, we shift the pattern by 2 characters (length of the matched prefix minus the LPS value: `4 - 2 = 2`) and continue matching. This skipping of unnecessary comparisons makes the KMP algorithm much more efficient.

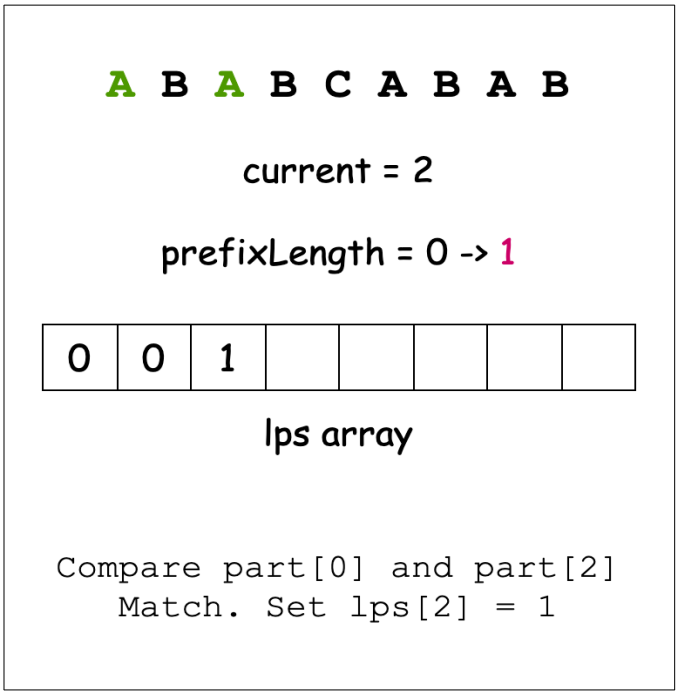
The LPS array is built using a linear iterative approach. We initialize two pointers: `current` (to traverse `part`) and `prefixLength` (to track the length of the matching prefix-suffix). We then iterate through the pattern:

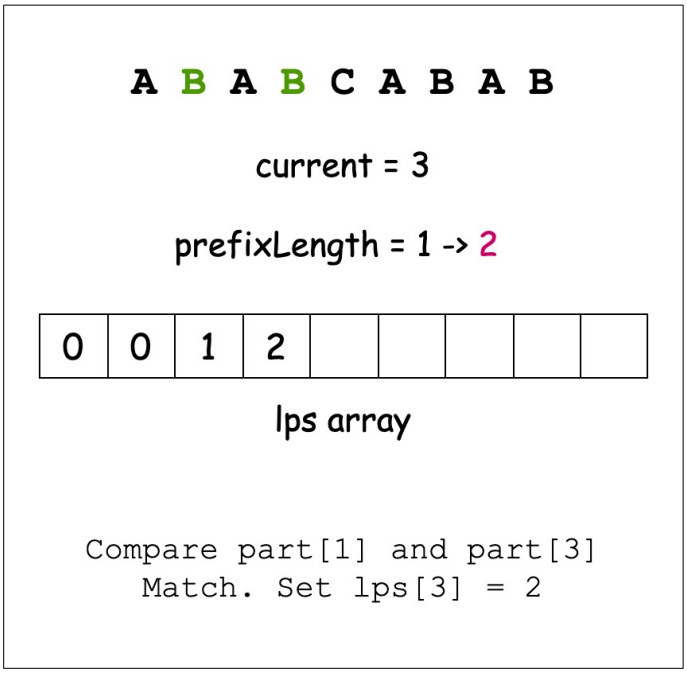
- If the characters at current and `prefixLength` match, we increment both pointers and set `lps[current] = prefixLength`.
- If they don't match and `prefixLength` is not zero, we backtrack `prefixLength` to `lps[prefixLength - 1]`.
- If they don't match and `prefixLength` is zero, we set `lps[current] = 0` and increment `current`.

Here's a slideshow to visualize this process better:









A B A B C A B A B

current = 4

prefixLength = 2 -> 0

0	0	1	2					

lps array

Compare part[2] and part[4]
No match. Backtrack

A B A B C A B A B

current = 4

prefixLength = 0

0	0	1	2	0				

lps array

Compare part[0] and part[4]
No match. Set lps[4] = 0

A B A B C A B A B

current = 5

prefixLength = 0 -> 1

0	0	1	2	0	1			

lps array

Compare part[0] and part[5]
Match. Set lps[5] = 1

A B A B C A B A B

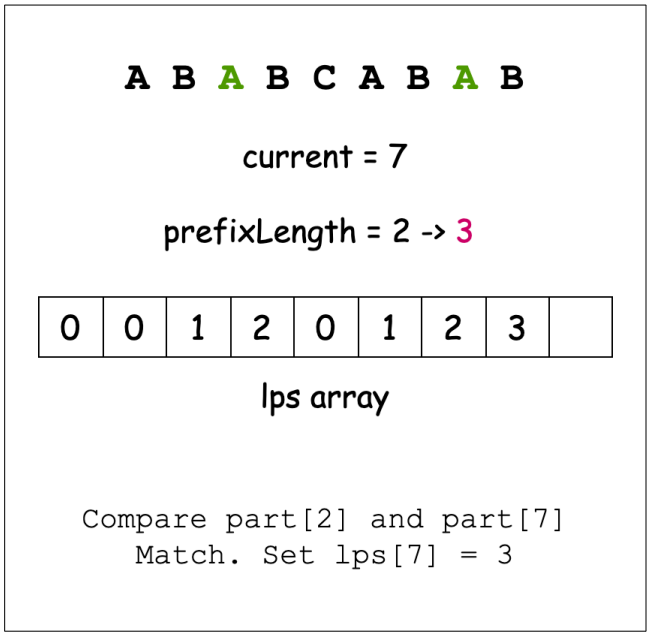
current = 6

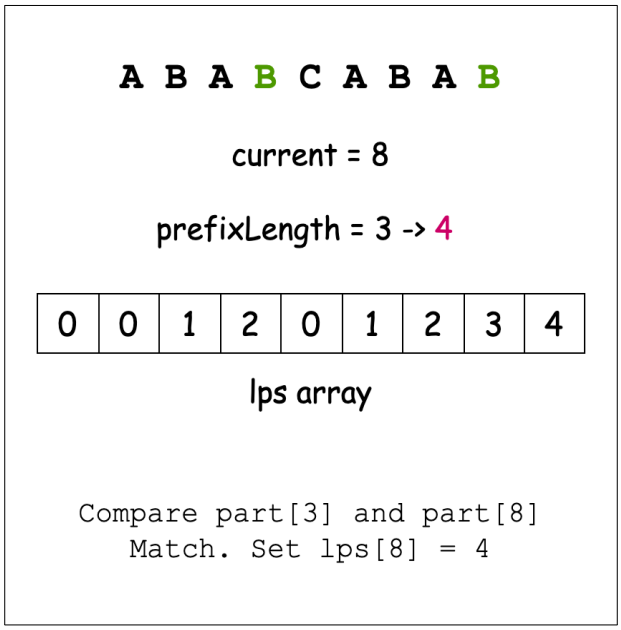
prefixLength = 1 -> 2

0	0	1	2	0	1	2		

lps array

Compare part[1] and part[6]
Match. Set lps[6] = 2





Finally, we process each character of `s` while using the LPS array to track how much of `part` has been matched. We iterate over `s` and when a complete match is found, we remove the matched substring from the stack. If a mismatch occurs, we use the LPS array to backtrack and continue matching.

After processing all characters of `s`, the stack contains the characters of `s` with all occurrences of part removed. We convert the stack into a string by popping characters and reversing the `result` (since stacks are last-in-first-out). We return this `result` as our answer.

1910. Remove All Occurrences of a Substring

```
/*
   KMP
   Time complexity: O(m+(n/m).m)=O(m+n)
   Space complexity: O(2m+n)
*/
class Solution {
   public:
      // KMP: return the longest prefix suffix array
      std::vector<int> kmp(std::string& needle){
         int m=needle.size();
         std::vector<int> lps(m,0);
         for(int current=1;current<m;++current ){
            int prefix_len=lps[current-1];
            while(prefix_len>0 && needle[prefix_len]!=needle[current]){
               prefix_len=lps[prefix_len-1];
            }
            lps[current]=prefix_len+(needle[current]==needle[prefix_len]);
         }
         return lps;
      }
```

```
std::string removeOccurrences(std::string s, std::string part) {
        int n=s.size();
        int m=part.size();

        std::vector<int> lps=kmp(part);

        std::string ans; // To store the answer

        // To store the matching prefix length during traversal
        std::vector<int> prefix_match_len;

        // For each character in s
        for(int i=0;i<n;++i){
            // Add it to answer
            ans.push_back(s[i]);

            // Get the length of the last matching prefix
            int j=prefix_match_len.empty()?0:prefix_match_len.back();

            // While the length of last matching prefix is not zero,
            // and we have a mismatch between the current character in s
            // and the j-th character in part, the back the previous match,
            // which is lps[j-1].
            while(j>0 && s[i]!=part[j]) j=lps[j-1];

            // If the current character in s and the j-th character in part are equal
            // pass to next character in part.
            if(s[i]==part[j]) j++;

            // Add that length to the array
            prefix_match_len.push_back(j);

            // If we have a complete match
            if(j==m){
                // Delete de last m characters from the answer
                for(int j=0;j<m;++j){
                    ans.pop_back();
                    prefix_match_len.pop_back();
                }
            }
        }
        return ans;
    }};
```