# Complete KMP Algorithm Implementation Guide

**Comprehensive Analysis with Examples and Tracing**

---

## Table of Contents

---

## 1. Introduction to KMP Algorithm {#introduction}

The Knuth-Morris-Pratt (KMP) algorithm is a string-searching algorithm that searches for occurrences of a "pattern" within a main "text" string. It was conceived by Donald Knuth, Vaughan Pratt, and James Morris in 1977.

**Key Advantages:**

- **Time Complexity**: O(n + m) where n = text length, m = pattern length
- **No Backtracking**: Never moves backward in the text
- **Optimal**: Examines each character in text at most twice
- **Complete**: Finds all occurrences including overlapping ones

**Core Concept:**

The algorithm preprocesses the pattern to create a "failure function" (LPS array) that encodes information about how the pattern matches against shifts of itself. This information is used to skip characters when a mismatch occurs.

---

**2. Complete Code Overview {#code-overview}**

```
def compute_lps(pattern):
    """
    Bne7seb el Longest Prefix Suffix (LPS) array lel pattern.
    LPS[i] = a6wal prefix mn pattern[0..i] elli heya suffix bardu.
    Da bysa3edna ne skip el comparisons elli msh darooriya lamma yeb2a fe mismatch.
    """
    lps = [0] * len(pattern)
    length = 0  # tool el prefix suffix el sabek
    i = 1

    while i < len(pattern):
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                # Hena msh bnzawed el i, bas bnraga3 el length
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps


def kmp_search(text, pattern):
    """
    Knuth-Morris-Pratt (KMP) algorithm el asli lel string matching.
    A7san mn el naive search - el time complexity beta3o O(n+m).
    """
    if not pattern:
        return []

    lps = compute_lps(pattern)
    result = []
    i = 0  # el index beta3 el text
    j = 0  # el index beta3 el pattern

    while i < len(text):
        if pattern[j] == text[i]:
            i += 1
            j += 1
            if j == len(pattern):
                # La2ina match kamla!
                result.append(i - j)
                j = lps[j - 1]  # Ngahaz lel match elli ba3daha
        else:
            if j != 0:
                j = lps[j - 1]  # Nesta3mel el LPS 3ashan neskip 7agat
        else:
            i += 1  # Mafish match, fa nro7 lel character elli ba3daha
    return result

# Ed5al el text wel pattern mn el user
text = input("Ed5al el text elli 3ayez tedawwar fih: ")
pattern = input("Ed5al el pattern elli 3ayez tedawwar 3aleih: ")

# Ne3red el natiga
result = kmp_search(text, pattern)
if result:
    print("El pattern mawgood 3and el indices:", result)

    # Nezhar makanet el pattern fel text
    print("\nShakl el pattern fel text:")
    print(text)

    # Ne3mel indicator ta7t kol match
    indicators = [" " for _ in range(len(text))]
    for pos in result:
        for i in range(pos, min(pos + len(pattern), len(text))):
            indicators[i] = "^"
    print("".join(indicators))

    # Ne3red kol match 3ala 7eda
    print("\nKol el matches:")
```

```
  for pos in result:
    match_end = min(pos + len(pattern), len(text))
    print(f"- Fel position {pos}: '{text[pos:match_end]}'")
else:
  print("El pattern mesh mawgood fel text")
```

## 3. Function 1: compute_lps() - Detailed Analysis {#compute-lps}

**Purpose and Definition:**

The LPS (Longest Prefix Suffix) array stores the length of the longest proper prefix of pattern[0..i] which is also a suffix of pattern[0..i] for each position i.

**Variable Definitions:**

- lps[]: The LPS array we're computing
- length: Length of the current longest prefix-suffix
- i: Current position in the pattern (starts from 1)

**Detailed Step-by-Step Analysis:**

**Initialization:**

```
lps = [0] * len(pattern)  # Initialize all values to 0
length = 0          # No prefix-suffix match initially
i = 1               # Start from second character
```

**Why start i from 1?**

- lps[0] is always 0 because a single character cannot have both proper prefix and suffix
- We compare each character with characters from the beginning of the pattern

**Main Loop Logic:**

**Case 1: Characters Match**

```
if pattern[i] == pattern[length]:
    length += 1
    lps[i] = length
    i += 1
```

When characters match:

1. Increment the length of current prefix-suffix
2. Store this length in lps[i]
3. Move to next character

**Case 2: Characters Don't Match**

```
else:
    if length != 0:
        length = lps[length - 1]  # Key insight: use previous LPS value
    else:
        lps[i] = 0
        i += 1
```

When characters don't match:

- If we had some prefix-suffix match (length > 0), use the LPS value to find the next best match
- If no match at all (length = 0), set lps[i] = 0 and move forward

**Complete Example: Pattern "ABABCABAB"**

Let's trace through the computation step by step:

```
Pattern: A B A B C A B A B
Index:   0 1 2 3 4 5 6 7 8
LPS:    [?, ?, ?, ?, ?, ?, ?, ?, ?]
```

Pattern: A B A B C A B A B

Index:   0 1 2 3 4 5 6 7 8

LPS:    [?, ?, ?, ?, ?, ?, ?, ?, ?]

**Step-by-Step Execution:**

| **Step** | **i** | **length** | **pattern[i]** | **pattern[length]** | **Action** | **lps[i]** | **LPS Array** |
|---|---|---|---|---|---|---|---|
| Init | 1 | 0 | B | A | - | - | [0,?,?,?,?,?,?,?,?] |
| 1 | 1 | 0 | B | A | Mismatch, length=0 | 0 | [0,0,?,?,?,?,?,?,?] |
| 2 | 2 | 0 | A | A | Match! | 1 | [0,0,1,?,?,?,?,?,?] |
| 3 | 3 | 1 | B | B | Match! | 2 | [0,0,1,2,?,?,?,?,?] |
| 4 | 4 | 2 | C | A | Mismatch, length=lps[1]=0 | 0 | [0,0,1,2,0,?,?,?,?] |
| 5 | 5 | 0 | A | A | Match! | 1 | [0,0,1,2,0,1,?,?,?] |
| 6 | 6 | 1 | B | B | Match! | 2 | [0,0,1,2,0,1,2,?,?] |
| 7 | 7 | 2 | A | A | Match! | 3 | [0,0,1,2,0,1,2,3,?] |
| 8 | 8 | 3 | B | B | Match! | 4 | [0,0,1,2,0,1,2,3,4] |

**Final LPS Array: [0, 0, 1, 2, 0, 1, 2, 3, 4]**

**Interpretation of LPS Values:**

- lps[0] = 0: Single character, no prefix-suffix
- lps[1] = 0: "AB" has no proper prefix that's also suffix
- lps[2] = 1: "ABA" - prefix "A" matches suffix "A"
- lps[3] = 2: "ABAB" - prefix "AB" matches suffix "AB"
- lps[4] = 0: "ABABC" - no prefix matches suffix
- lps[5] = 1: "ABABCA" - prefix "A" matches suffix "A"

- lps[6] = 2: "ABABCAB" - prefix "AB" matches suffix "AB"
- lps[7] = 3: "ABABCABA" - prefix "ABA" matches suffix "ABA"
- lps[8] = 4: "ABABCABAB" - prefix "ABAB" matches suffix "ABAB"

---

**4. Function 2: kmp_search() - Detailed Analysis {#kmp-search}**

**Purpose:**

Find all occurrences of pattern in text using the KMP algorithm with the precomputed LPS array.

**Variable Definitions:**

- text: The string we're searching in
- pattern: The string we're looking for
- lps: Precomputed LPS array
- result: List to store starting indices of matches
- i: Pointer for text (never goes backward)
- j: Pointer for pattern (can reset using LPS)

**Detailed Algorithm Flow:**

**Step 1: Handle Edge Cases**

```
if not pattern:
  return []
```

If pattern is empty, return empty result immediately.

**Step 2: Preprocessing**

```
lps = compute_lps(pattern)

result = []

i = 0  # text pointer

j = 0  # pattern pointer
```

**Step 3: Main Search Loop**

```
while i < len(text):
```

**Step 3: Main Search Loop**

```
while i < len(text):
```

-
-
-
-

Continue until we've examined all characters in text.

**Step 4: Character Comparison**

**Case 1: Characters Match**

```
if pattern[j] == text[i]:
  i += 1
  j += 1
  if j == len(pattern):
    # Complete match found!
    result.append(i - j)
    j = lps[j - 1]
```

-
-
-
-

When characters match:

1. Advance both pointers
2. Check if we've matched the entire pattern
3. If complete match: record starting position (i - j)

4. Reset j using LPS to continue searching for overlapping matches

**Case 2: Characters Don't Match**

```
else:
  if j != 0:
    j = lps[j - 1]  # Use LPS to skip redundant comparisons
  else:
    i += 1  # No partial match, move to next character in text
```

When characters don't match:

- If we had partial match (j > 0): use LPS to find next possible match position
- If no match at all (j = 0): simply advance text pointer

**Why This Works:**

The key insight is that when we have a mismatch after matching j characters, we don't need to start over completely. The LPS array tells us how many characters we can "keep" from our partial match.

---

**5. User Interface and Visualization {#user-interface}**

**Input Section:**

```
text = input("Ed5al el text elli 3ayez tedawwar fih: ")
pattern = input("Ed5al el pattern elli 3ayez tedawwar 3aleih: ")
```

- Prompts user for text and pattern in Arabic
- Stores input for processing

**Results Display:**

```
result = kmp_search(text, pattern)
if result:
  # Show detailed results
```

```
else:
   print("El pattern mesh mawgood fel text")
```

-
-
-
-

**Detailed Output Components:**

**1. Basic Results:**

```
print("El pattern mawgood 3and el indices:", result)
```

-
-
-
-

Shows list of all starting positions where pattern was found.

**2. Text Display:**

```
print("\nShakl el pattern fel text:")
print(text)
```

-
-
-
-

Shows the original text for reference.

**3. Visual Indicator Creation:**

```
indicators = [" " for _ in range(len(text))]
for pos in result:
   for i in range(pos, min(pos + len(pattern), len(text))):
        indicators[i] = "^"
print("".join(indicators))
```

-
-
-
-

**Detailed Breakdown:**

- Create array of spaces matching text length
- For each match position, mark corresponding characters with "^"
- Use min() to prevent index overflow
- Join array into string for display

**4. Individual Match Details:**

```
print("\nKol el matches:")
for pos in result:
    match_end = min(pos + len(pattern), len(text))
    print(f"- Fel position {pos}: '{text[pos:match_end]}'")
```

-
-
-
-

Lists each match with its position and actual matched substring.

---

**6. Complete Examples with Step-by-Step Tracing {#examples}**

**Example 1: Simple Pattern Matching**

**Input:**

- Text: "ABABCABAB"

- Pattern: "ABAB"

**Step 1: Compute LPS for "ABAB"**

Pattern: A B A B

Index:  0 1 2 3

LPS:  [0, 0, 1, 2]

- 
- 
- 
-

**Step 2: KMP Search Trace**

| Step | i | j | text[i] | pattern[j] | Action | Result | Explanation |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | A | A | Match | i=1, j=1 | Characters match |
| 2 | 1 | 1 | B | B | Match | i=2, j=2 | Characters match |
| 3 | 2 | 2 | A | A | Match | i=3, j=3 | Characters match |
| 4 | 3 | 3 | B | B | Match | i=4, j=4 | Characters match |
| 5 | 4 | 4 | - | - | Complete match! | Record pos 0, j=2 | Found match at position 0 |
| 6 | 4 | 2 | C | A | Mismatch | j=0 | Use lps[1]=0 |
| 7 | 4 | 0 | C | A | Mismatch | i=5 | Advance text pointer |
| 8 | 5 | 0 | A | A | Match | i=6, j=1 | Characters match |
| 9 | 6 | 1 | B | B | Match | i=7, j=2 | Characters match |
| 10 | 7 | 2 | A | A | Match | i=8, j=3 | Characters match |
| 11 | 8 | 3 | B | B | Match | i=9, j=4 | Characters match |

| Step | i | j | text[i] | pattern[j] | Action | Result | Explanation |
|---|---|---|---|---|---|---|---|
| 12 | 9 | 4 | - | - | Complete match! | Record pos 5 | Found match at position 5 |

**Final Result:** [0, 5]

**Step 3: Visualization Output**

El pattern mawgood 3and el indices: [0, 5]

Shakl el pattern fel text:

ABABCABAB

^^^^  ^^^^

Kol el matches:

- Fel position 0: 'ABAB'

- Fel position 5: 'ABAB'

- 
- 
- 
-

**Example 2: Overlapping Pattern**

**Input:**

- Text: "AAAAAAA"
- Pattern: "AAA"

**LPS for "AAA": [0, 1, 2]**

**Search Trace:**

| Step | i | j | text[i] | pattern[j] | Action | Result |
|---|---|---|---|---|---|---|
| 1-3 | 0-2 | 0-2 | A,A,A | A,A,A | All match | i=3, j=3 |
| 4 | 3 | 3 | - | - | Complete match! | Record pos 0, j=2 |

| Step | i | j | text[i] | pattern[j] | Action | Result |
|---|---|---|---|---|---|---|
| 5 | 3 | 2 | A | A | Match | i=4, j=3 |
| 6 | 4 | 3 | - | - | Complete match! | Record pos 1, j=2 |
| 7 | 4 | 2 | A | A | Match | i=5, j=3 |
| 8 | 5 | 3 | - | - | Complete match! | Record pos 2, j=2 |
| ... | ... | ... | ... | ... | Continue... | ... |

**Result:** [0, 1, 2, 3, 4] - All possible overlapping positions

**Example 3: No Match Found**

**Input:**

- Text: "ABCDEF"
- Pattern: "XYZ"

**Search Result:**

No characters match, algorithm advances through text without finding pattern.

**Output:**

El pattern mesh mawgood fel text

-
-
-
-

---

**7. Time and Space Complexity Analysis {#complexity}**

**Time Complexity: O(n + m)**

**LPS Computation: O(m)**

- Each character in pattern is examined at most twice

- Inner while loop (when length != 0) can execute at most m times total
- Amortized analysis shows linear time

**Search Phase: O(n)**

- Text pointer i never goes backward
- Pattern pointer j resets using LPS, but total character comparisons ≤ 2n
- Each character in text examined at most twice

**Total: O(m) + O(n) = O(n + m)**

**Space Complexity: O(m)**

**LPS Array: O(m)**

- Stores one integer per pattern character

**Other Variables: O(1)**

- Fixed number of pointers and counters

**Result Storage: O(k)**

- Where k is number of matches found
- In worst case, k could be O(n), but this doesn't change the algorithm's space complexity

**Comparison with Naive Algorithm:**

| Aspect | Naive Algorithm | KMP Algorithm |
|---|---|---|
| Time Complexity | O(n × m) | O(n + m) |
| Space Complexity | O(1) | O(m) |
| Best Case | O(n) | O(n + m) |
| Worst Case | O(n × m) | O(n + m) |
| Backtracking | Yes (in text) | No |
| Preprocessing | None | O(m) |

**8. Comparison with Other Algorithms {#comparison}**

**1. Naive String Matching**

```
def naive_search(text, pattern):
    result = []
    for i in range(len(text) - len(pattern) + 1):
        j = 0
        while j < len(pattern) and text[i + j] == pattern[j]:
            j += 1
        if j == len(pattern):
            result.append(i)
    return result
```

- 
- 
- 
-

**Pros:** Simple, no extra space **Cons:** O(n×m) time complexity, lots of redundant comparisons

**2. Boyer-Moore Algorithm**

**Approach:** Scan pattern from right to left, skip characters based on mismatch **Time:** O(n×m) worst case, O(n/m) average case **Space:** O(σ) where σ is alphabet size **Best for:** Large alphabets, long patterns

**3. Rabin-Karp Algorithm**

**Approach:** Use rolling hash to compare pattern with text substrings **Time:** O(n×m) worst case, O(n+m) average case **Space:** O(1) **Best for:** Multiple pattern search

**Why Choose KMP?**

1. **Guaranteed Performance:** Always O(n+m), no worst-case degradation
2. **Educational Value:** Teaches important algorithmic concepts

3. **Reliability:** No hash collisions or alphabet size dependencies
4. **Completeness:** Finds all occurrences including overlapping ones

---

**9. Real-World Applications {#applications}**

**1. Text Editors**

- Find/Replace functionality
- Syntax highlighting
- Auto-completion

**2. Bioinformatics**

- DNA sequence analysis
- Protein pattern matching
- Genome assembly

**3. Network Security**

- Intrusion detection systems
- Virus scanning
- Pattern matching in network traffic

**4. Compilers**

- Lexical analysis
- Token recognition
- Comment detection

**5. Search Engines**

- Full-text search
- Index construction
- Query processing

**6. Data Mining**

- Log file analysis

- Pattern discovery
- Anomaly detection

---

**10. Discussion Points and Q&A Preparation {#discussion}**

**Expected Questions and Answers:**

**Q1: "Why not use built-in string functions?"**

**Answer:** "While built-in functions are optimized, implementing KMP teaches us:

- Algorithm design principles
- Time complexity analysis
- The importance of preprocessing
- How to avoid redundant work
- Understanding of string algorithms used in real systems"

**Q2: "When would KMP be slower than naive search?"**

**Answer:** "KMP might have higher constant factors for very small patterns (length 1-2) due to LPS preprocessing overhead. However, asymptotically, KMP is never slower and becomes significantly faster as pattern size increases."

**Q3: "How does KMP handle edge cases?"**

**Answer:** "The implementation handles several edge cases:

- Empty pattern: Returns empty result immediately
- Pattern longer than text: No matches found naturally
- Single character pattern: Works correctly with LPS[0] = 0
- Overlapping matches: Uses LPS to continue searching after each match"

**Q4: "What's the most complex part of the algorithm?"**

**Answer:** "The LPS computation is the most subtle part. The key insight is that when we have a mismatch after partial match, we use previously computed LPS values to determine how much of the partial match we can 'keep' rather than starting over."

**Q5: "Can you explain the mathematics behind LPS?"**

**Answer:** "LPS leverages the mathematical property that if a string has a border (a prefix that's also a suffix), then finding this border helps us skip redundant comparisons. The LPS array essentially precomputes all possible 'fallback' positions for every position in the pattern."

**Key Points to Emphasize:**

1. **Efficiency Gain:** Transform O(n×m) to O(n+m)
2. **No Backtracking:** Text pointer never goes backward
3. **Smart Preprocessing:** LPS array captures pattern's structure
4. **Practical Relevance:** Used in real-world applications
5. **Algorithmic Beauty:** Demonstrates how mathematical insight improves algorithms

**Demonstration Tips:**

1. **Start with Simple Example:** Use short pattern like "ABAB" in "ABABCABAB"
2. **Show LPS Computation:** Step through LPS array construction
3. **Trace Search Process:** Show how mismatches are handled
4. **Compare with Naive:** Highlight the efficiency difference
5. **Discuss Applications:** Connect to real-world usage

**Potential Weaknesses to Address:**

1. **Space Overhead:** Acknowledge O(m) space requirement
2. **Implementation Complexity:** More complex than naive approach
3. **Constant Factors:** May be slower for very small patterns
4. **Not Always Best:** Other algorithms might be better for specific cases

**Conclusion Statement:**

"The KMP algorithm represents a perfect example of how theoretical computer science concepts translate into practical improvements. By investing time in preprocessing to understand the pattern's structure, we achieve significant performance gains that make the difference between feasible and infeasible for large-scale text processing applications."

---

**Final Note:** This implementation demonstrates not just the algorithm itself, but also good software engineering practices including clear documentation, user-friendly interfaces, and comprehensive output formatting. The bilingual comments show consideration for diverse development teams, and the visualization components make the algorithm's behavior transparent to users.