Presented by: Debdoot Manna

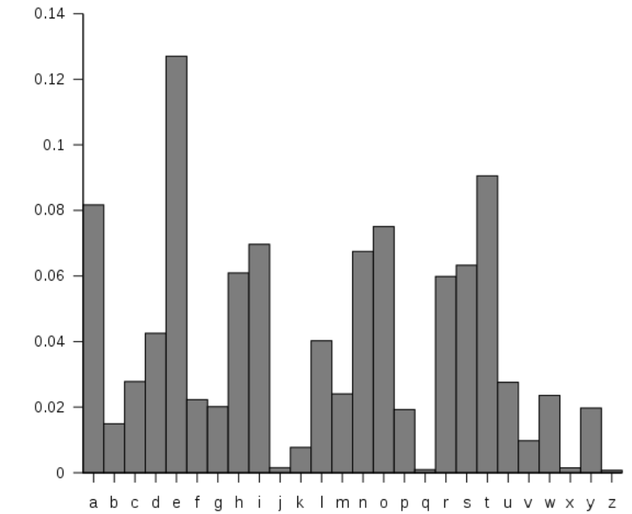
spotify mod apk download without any virus

A Single Traversal Algorithm Approach

Finding the First k Non-repeating Characters in a String in a Single Traversal

- Character frequency analysis is a fundamental operation in text processing
- Finding non-repeating characters has applications in:
  - Data compression
  - Cryptography
  - Pattern recognition
  - Text analysis
- Today's focus: An efficient algorithm to find the first k non-repeating characters in a single traversal

Introduction



Feeling bored already?

Problem Definition

I/O

Input:

- A string s of length n
- An integer k representing the number of non-repeating characters to find

Output:

- The first k characters from the string that appear exactly once

Important...

Constraint:

- Must be solved in a single traversal of the string
- Optimize for both time and space complexity

Naive Approach

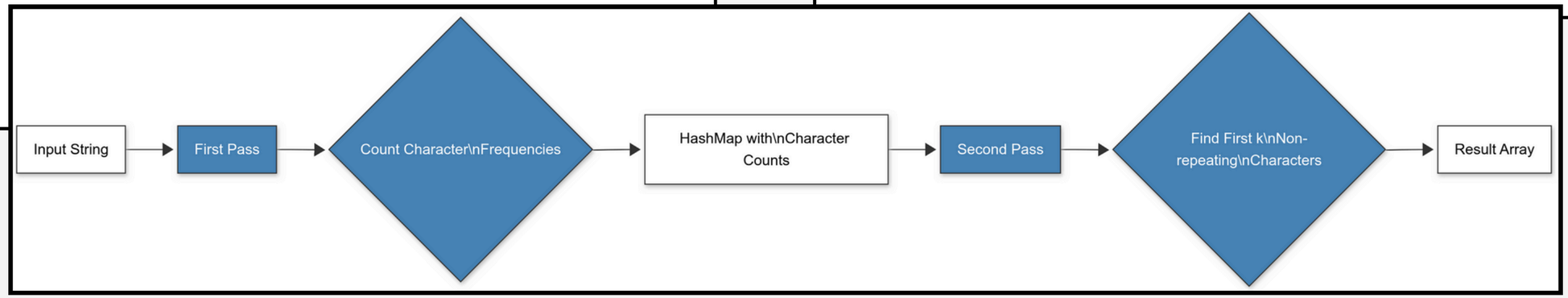
Why Not Multiple Traversals?

A simple approach would require multiple traversals:

1. Count frequency of each character
2. Scan the string again to find non-repeating characters

Issues with this approach:

- Time complexity: O(n + n) = O(n), but requires multiple passes
- Not optimal for streaming data or very large strings
- Doesn't handle dynamic updates efficiently



Single Traversal Algorithm

How to get out from here..

- Our goal: Find the first k non-repeating characters in a single pass
- Key data structures:
  - HashMap: To track character frequencies
  - Queue: To maintain the order of characters as they appear
- Main idea:
  - Process each character once
  - Use the data structures to efficiently identify and retrieve non-repeating characters

I am just doing this because I was told to do it...

Note: If I stutter, its because I am bored

Run

Compile

Character from the string

Key

Frequency count of the character

Value

HashMap (Dictionary)

Queue

Maintains the order of characters as they appear in the string

Allows for efficient retrieval of the first non-repeating characters

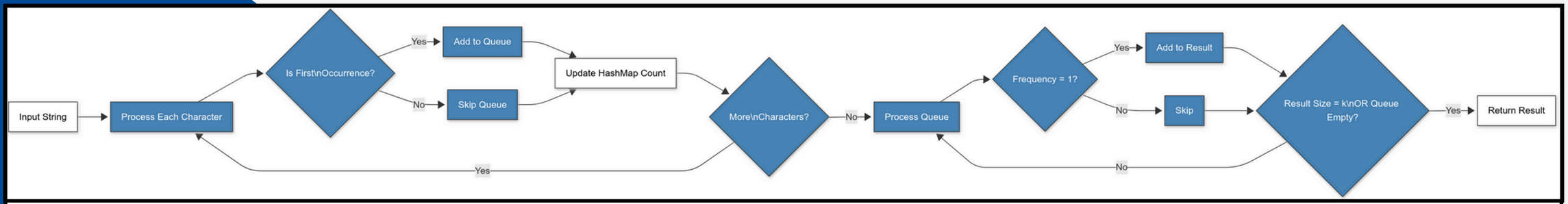
O(1) average time for insertion and lookup

HashMap

O(1) for enqueue and dequeue operations

Queue

Time complexity benefits



Algorithm Steps

Step-by-Step Procedure

1

Initialize an empty HashMap to store character frequencies

2

Initialize an empty Queue to maintain character order

3

Process each character in the string:

- Increment its count in the HashMap
- Add the character to the Queue if it's encountered for the first time

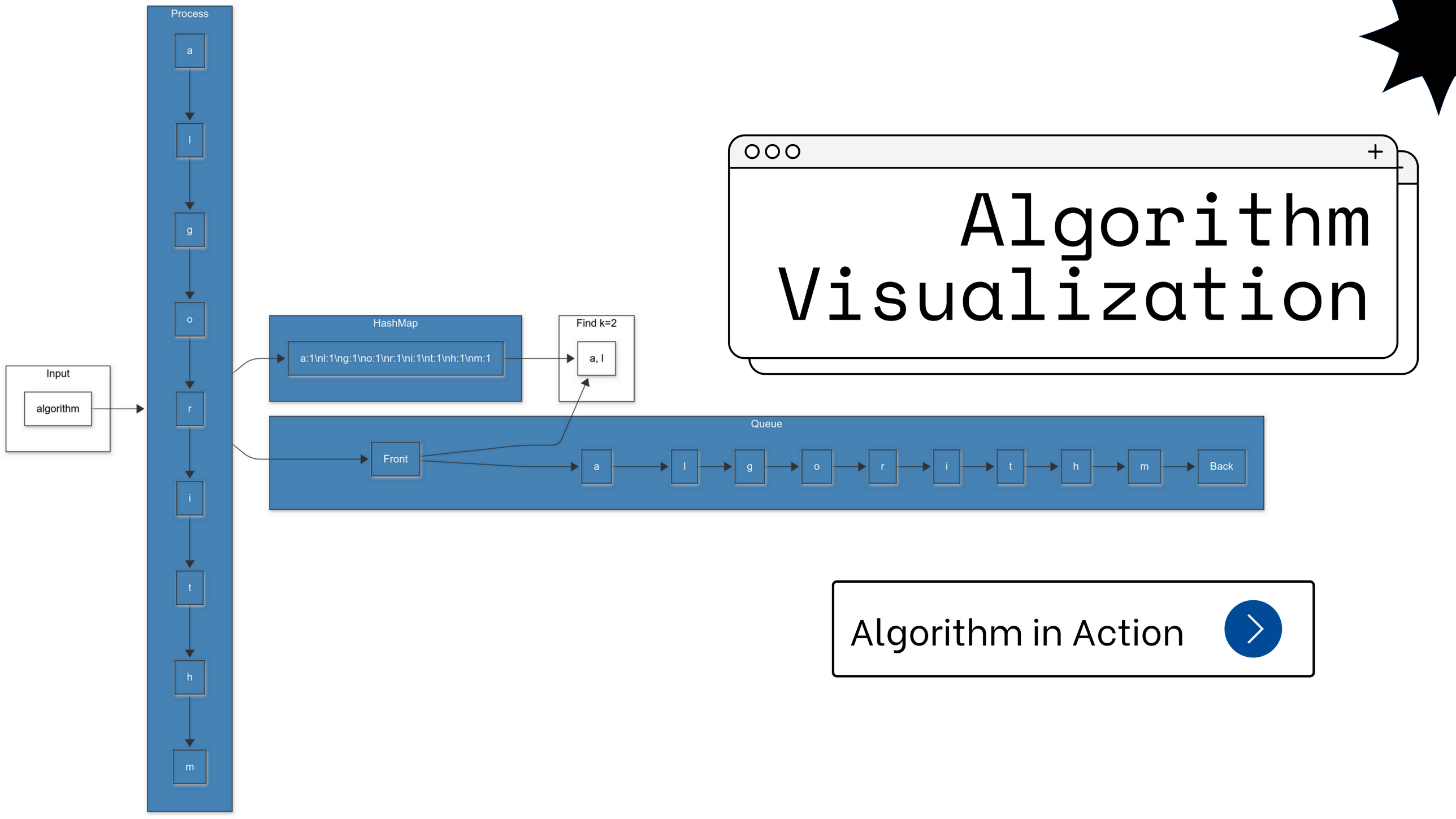
4

After processing all characters:

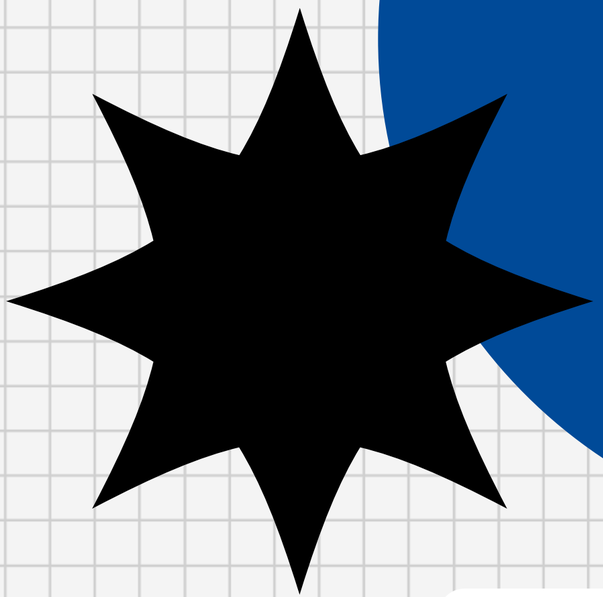
- Dequeue characters and check their frequency in the HashMap
- If frequency is 1, add to result list
- Continue until k non-repeating characters are found or queue is empty

5

Return the first k non-repeating characters


Process
a
l
g
o
r
i
t
h
m
Input
algorithm
HashMap
a:1\nl:1\ng:1\no:1\nr:1\ni:1\nt:1\nh:1\nm:1
Find k=2
a, l
Queue
Front
a
l
g
o
r
i
t
h
m
Back
Algorithm Visualization
Algorithm in Action

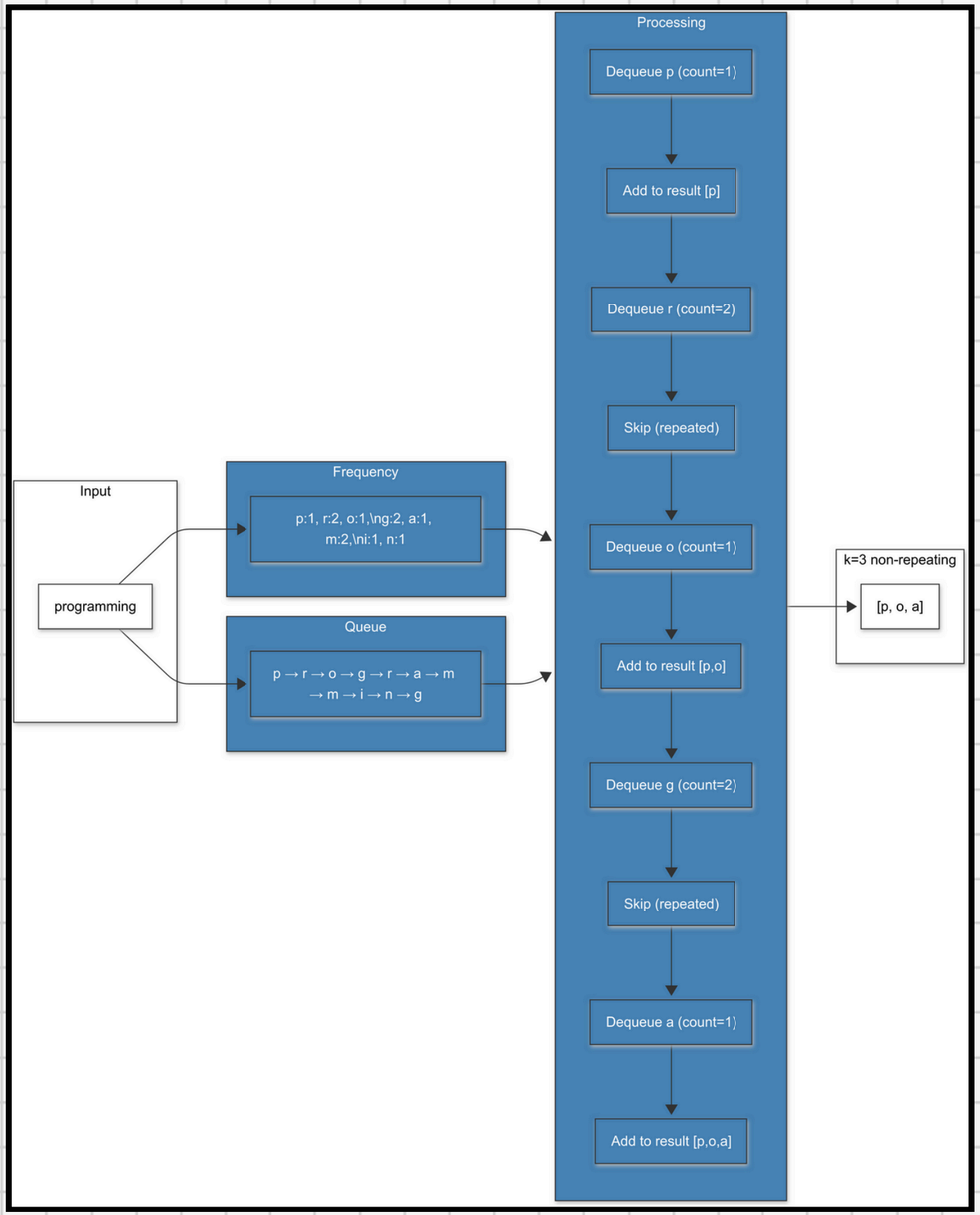
Processing the String



Input: s = “algorithm“, k = 2
Process each character:
’a’: Add to Queue, HashMap = {a:1}
’l’: Add to Queue, HashMap = {a:1, l:1}
’g’: Add to Queue, HashMap = {a:1, l:1, g:1}
’o’: Add to Queue, HashMap = {a:1, l:1, g:1, o:1}
’r’: Add to Queue, HashMap = {a:1, l:1, g:1, o:1, r:1}
’i’: Add to Queue, HashMap = {a:1, l:1, g:1, o:1, r:1, i:1}
’t’: Add to Queue, HashMap = {a:1, l:1, g:1, o:1, r:1, i:1, t:1}
’h’: Add to Queue, HashMap = {a:1, l:1, g:1, o:1, r:1, i:1, t:1, h:1}
’m’: Add to Queue, HashMap = {a:1, l:1, g:1, o:1, r:1, i:1, t:1, h:1, m:1}

Sorry for my poor editing skills

Still in learning progress...



Finding Non-repeating Characters

After processing the string “algorithm“:

Queue contains: [a, l, g, o, r, i, t, h, m]

All characters have frequency 1 in the HashMap

Results:

- First non-repeating character: ’a’
- Second non-repeating character: ’l’

Final output: [’a’, ’l’]

Code Implementation

*this section is for nerds

OnlyNerds

Algorithm Implementation in Python

```
from collections import Counter
from queue import Queue
def first_k_non_repeating(s, k):
    # Initialize counter for character frequencies
    char_count = Counter()

    # Queue to maintain order of characters
    char_queue = Queue()

    # List to store the result
    result = []

  # Process each character in the string
  for char in s:
  # Increment the character count
  char_count[char] += 1

  # Add character to queue if it's the first occurrence
  if char_count[char] == 1:
  char_queue.put(char)
```

```
  # Find the first k non-repeating characters
  while not char_queue.empty() and len(result) < k:
  # Get the next character from the queue
  char = char_queue.get()

  # Add to result if it appears exactly once
  if char_count[char] == 1:
  result.append(char)

  return result

  # Example usage
  s = "programming"
  k = 3
  print(first_k_non_repeating(s, k)) # Output: ['p', 'o', 'a']
```

Kindly don't ask coding questions...

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
#include <queue>
#include <vector>
std::vector<char> firstKNonRepeating(const std::string& s, int k) {
 // Map: char frequencies
 std::unordered_map<char, int> count;
 // Queue: char order
 std::queue<char> q;
 // Result: non-repeating chars
 std::vector<char> res;
 // Process string
 for (char c : s) {
 count[c]++;
 if (count[c] == 1) q.push(c);
 }
```

```cpp
 // Find k non-repeating
   while (!q.empty() && res.size() < k) {
     char c = q.front(); q.pop();
     if (count[c] == 1) res.push_back(c);
   }
   return res;
 }
 int main() {
   std::string s = “programming“;
   int k = 3;
   std::vector<char> res = firstKNonRepeating(s, k);
   for (char c : res) std::cout << c << “ “; // Output: p o a
   return 0;
 }
```

C++ Example

“You know, you could’ve just, like, looked at the string, right? Or is that too ’low-level’?“ (that’s what you’re thinking right?)

Imp...

Time and Space Complexity Analysis

- Time Complexity: O(n)
  - Each character is processed exactly once
  - HashMap operations take O(1) average time
  - Queue operations take O(1) time
  - Overall linear time complexity relative to the string length

- Space Complexity: O(min(n, c))
  - HashMap stores at most min(n, c) entries where c is the character set size
  - Queue stores at most n characters
  - For ASCII strings, space complexity is effectively O(1) since c is constant (256)
  - For Unicode strings, c can be much larger

This is not Interstellar. It’s more difficult than that.

Edge Cases and Handling

Edge Cases

- Empty string: Return an empty result
- k = 0: Return an empty result
- k greater than number of non-repeating characters: Return all available non-repeating characters
- No non-repeating characters: Return an empty result
- Case sensitivity: By default, 'A' and 'a' are treated as different characters
- Special characters and spaces: Processed like any other character

```
def first_k_non_repeating_with_edge_cases(s, k):

  # Handle edge cases
  if not s or k <= 0:
    return []
  char_count = Counter()
  char_queue = Queue()
  result = []

  # Process each character
  for char in s:
    char_count[char] += 1
    if char_count[char] == 1:
      char_queue.put(char)
```

```
# Find non-repeating characters
  while not char_queue.empty() and len(result) < k:
    char = char_queue.get()
    if char_count[char] == 1:
      result.append(char)

  return result

# Edge case examples

print(first_k_non_repeating_with_edge_cases(“”, 3))  # []
print(first_k_non_repeating_with_edge_cases(“aabbcc“, 2))
# []
print(first_k_non_repeating_with_edge_cases(“abcde“, 10))
# [’a’, ’b’, ’c’, ’d’, ’e’]
```

Handling Edge Cases in Code

Optimizations

- Early termination: Stop processing once we know we can’t find k non-repeating characters
- Linked HashMap: Use a custom data structure that combines hashmap and linked list functionality
- Batch processing: For very large strings, process in chunks to improve memory locality
- Character set optimization: Use array instead of hashmap for fixed character sets
- Queue optimization: Only enqueue characters that might be part of the result

Real-world Applications

Practical Applications

- Data compression: Identifying unique patterns in data
- Cryptography: Analyzing character distributions for frequency analysis
- Text processing: Finding unique identifiers or markers in text
- Error detection: Identifying anomalies in streams of data
- Network packet analysis: Finding unique identifiers in packet streams
- Database querying: Optimizing first-k-distinct type queries

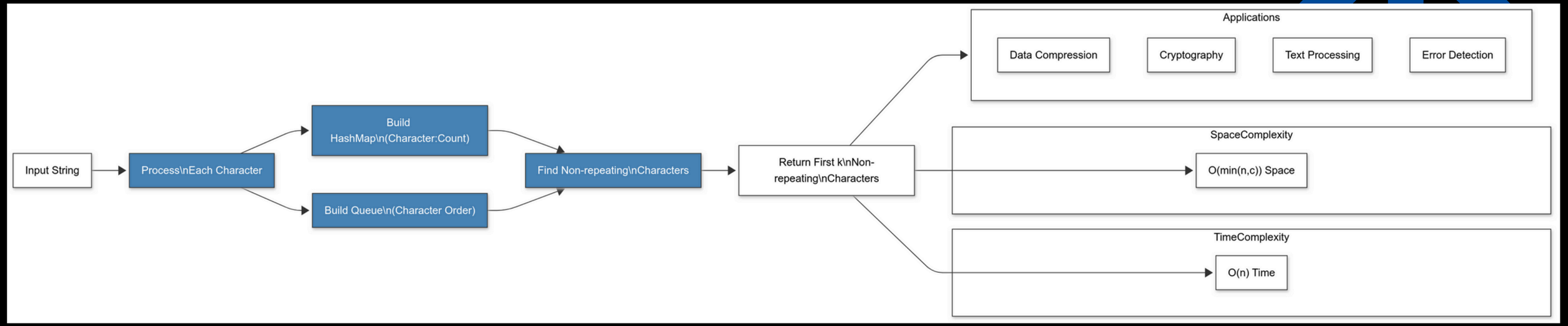
Finally...

CONCLUSION

- Efficient algorithm to find first k non-repeating characters in O(n) time
- Single traversal approach using HashMap and Queue data structures
- Balanced approach for time and space complexity

- Handles various edge cases gracefully
- Practical applications in text processing, data analysis, and more
- Extensible approach that can be optimized for specific use cases

Remember all the things I said today



Thank you!

For tollerating me :)

References

If there is any queries, just create an OpenAI account and sign in to ChatGPT... You'll get more than algorithms, iykyk...

- ChatGPT
- ChatGPT
- ChatGPT
- ChatGPT
- ChatGPT
- Google
- Yep, that's it