고급소프트웨어실습 8주차 과제

> **학번: 20171665 / 이름: 이선호**

> Sliding Window 실습 3의 효율적인 풀이를 위해서는 어떤 자료구조를 사용해야 되는지를 이유와 함께 제시하시오

Sliding window를 사용하는 실습 3의 효율적인 풀이법의 핵심은 바로 window를 하나씩 이동하면서 window 범위에서 나가는 원소 값 하나와 새로 들어오는 원소 값 하나만을 고려한 채 나머지 window에 포함된 원소 각각의 값을 일일이 loop를 돌면서 확인하지 않고 그들의 합을 고려하여 평균을 계산해야 하며, 이를 구현하기 위해 적절한 자료구조를 사용해야 한다.

Stack은 LIFO(Last-In First-Out) 방식으로 구현된 자료구조이며, 가장 최근에 탐색한 원소를 빠르게 포함하거나 제외하는 데 유용하다. 그러나 본 문제처럼 sliding window를 첫 번째 원소부터 마지막 원소를 훑는 방향으로 차례대로 이동해야 하는 경우에는 부적절할 수 있다. Queue는 FIFO(First-In First-Out) 방식으로 구현된 자료구조이며, 가장 오래 전에 window에 포함된 왼쪽 끝의 원소와 최근에 window에 포함된 오른쪽 끝의 원소를 평균을 구하는 합에서 제외하거나 포함시키는 데 유리하다. 그러나 window 내의 양 끝의 원소를 제외한 나머지 원소의 합을 일일이 보지 않으려면 이 범위의 값들의 합을 저장하여 접근할 수 있는 자료구조가 추가로 요구된다.

가장 좋은 자료구조는 배열(array)이라고 본다. 각 원소의 값을 배열의 원소로 저장하고 첫 window의 합만 미리 구한 채 window를 왼쪽에서 오른쪽으로 이동해 가면서 왼쪽 끝의 배열 원소 값은 window 합에서 빼고 오른쪽 끝의 배열 원소 값은 더하는 방식으로 이동하면, 양끝을 제외한 나머지 원소의 값을 일일이 알지 않아도 시간복잡도를 $O(\text{전체 수 list 길이})$에 효율적으로 해결할 수 있다.

> 실습에서 다루지 않은 문제 중에 Two pointers와 Sliding window를 사용했을 때 더 효율적으로 풀 수 있는 문제를 각각 하나씩 들고, 어떤 면에서 효율적인지 알고리즘을 설명하시오.

Two Pointers 사용 시 더 효율적인 문제

[문제 링크] https://www.acmicpc.net/problem/16472

Two pointers로 풀 수 있는 이 문제는 제 5회 한양대학교 프로그래밍 경시대회 Advanced Division의 E번으로 출제되었던 문제이며, 개인적으로 블로그에 풀이법을 정리하여 올린 적이 있다. (블로그 링크)

번역기가 인식할 수 있는 알파벳의 종류의 최대 개수가 N으로 주어지는데, 이는 주어진 문자열에서 최대 N개의 종류의 알파벳을 지닌 연속된 문자열을 인식한다는 의미이다. 이때, 번역기가 인식할 수 있는 문자열의 최대 길이를 구하는 것이 문제이다.

이 문제는 크게 두 가지의 방법으로 풀 수 있는데, 이분 탐색 사용 방법과 two pointers 사용 방법으로 나눌 수 있다. 먼저 이분 탐색으로 푸는 방법을 소개하고, two pointers를 대신 사용했을 때 이분 탐색보다 왜 효율적인지를 분석하고자 한다.

이분 탐색 사용 방법

번역기가 인식할 수 있는 문자열의 길이를 이분 탐색의 대상인 `mid` 로 설정하면, 번역기는 최대 `mid` 길이만큼 연속된 문자열을 인식할 수 있다고 볼 수 있다.그래서 인식 가능한 문자열의 길이인 `mid` 를 `left` 와 `right` 을 가지고 탐색할 수 있다. 주어진 문자열에서 `mid` 길이만큼 연속되는 부분 문자열을 탐색하면서 각 연속된 부분 문자열을 구성하는 알파벳 종류의 개수인 `cnt` 와 번역기가 인식 가능한 최대 알파벳 개수인 N을 비교하는데, `cnt` 가 N보다 작거나 같으면 해당 부분 연속 문자열을 인식할 수 있고, `cnt` 가 N보다 크면 해당 부분 연속 문자열을 인식할 수 없다. 만약 인식할 수 있는 부분 연속 문자열이 전체 문자열에서 하나라도 존재하면, `mid` 보다 더 긴 길이의 문자열도 인식할 수 있는지를 확인해 봐야 하므로 `left` 를 `mid + 1` 으로 업데이트하고, 그렇지 않으면 `mid` 보다 더 긴 길이의 부분 연속 문자열은 당연히 인식할 수 없으므로 `right` 을 `mid - 1` 로 업데이트한다. `left` 와 `right` 가 같아지는 종료 조건까지 이분 탐색을 진행하여 최종적인 답인 `mid` 값을 구할 수 있다.

주의할 점은 매번 전체 문자열에서 `mid` 길이만큼의 부분 연속 문자열을 일일이 확인하면서 해당 부분 연속 문자열에 있는 알파벳의 출현 여부를 판단하는 것은 비효율적이므로 이분 탐색 전에 누적 합을 사용하여 첫 번째 문자에서 i 번째 문자까지 각 알파벳이 몇 번 나타났는지 그 횟수를 배열로 저장해놓는 것이 바람직하다.

이분 탐색을 사용하면 위에서 언급한 바처럼 누적 합을 미리 구해야 하고, 이분 탐색 시 매번 `mid` 를 업데이트할 때마다 전체 문자열에서 `mid` 길이만큼의 모든 부분 연속 문자열을 한 번씩 확인하므로 시간복잡도가 최대 $O(n \log n)$이다.

Two Pointers 사용 방법

전체 문자열을 왼쪽에서부터 오른쪽으로 보면서 부분 연속 문자열을 확인하는데, 각 부분 연속 문자열에 몇 개의 서로 다른 종류의 알파벳이 사용되었는지를 세면서 조건에 따라 부분 연속 문자열의 시작 위치를 가리키는 `left` 포인터 또는 끝나는 위치를 가리키는 `right` 포인터를 이동한다. 구체적으로 조건을 서술하면 다음과 같다.

1. 현재 부분 연속 문자열을 구성하는 알파벳 종류 수가 N보다 클 경우

   그 이후에 바로 이어지는 부분을 합쳐서 부분 연속 문자열을 구성해도 인식이 불가능하므로 `left` 포인터를 1만큼 오른쪽으로 이동시켜서 현재 부분 연속 문자열의 부분 연속 문자열이 인식 가능한지 확인한다.

2. 현재 부분 연속 문자열을 구성하는 알파벳 종류 수가 N보다 작거나 같을 경우

   그 이후에 바로 이어지는 부분을 합쳐서 부분 연속 문자열을 구성했을 때 인식이 가능한지 더 확인해 봐야 하므로 `right` 포인터를 1만큼 오른쪽으로 이동시킨다.

이를 구현하기 위해 다음과 같은 자료구조를 정의하는 것이 필요하다.

> **`cnt`** : `left` 포인터부터 `right` 포인터까지의 부분 연속 문자열에서 사용된 알파벳 종류 개수

> **`letter[c]`** : `left` 포인터부터 `right` 포인터까지의 부분 연속 문자열에서 알파벳 c가 사용된 횟수

Two Pointers 사용이 효율적인 이유

Two pointers를 사용하면 이분 탐색의 시간복잡도인 $O(n \log n)$를 $O(N)$로 줄일 수 있다. 또한 누적 합을 미리 계산할 필요 없이 `left` 포인터 또는 `right` 포인터를 증가시킬 때 각 포인터가 가리키는 문자에 따라서 부분 연속 문자열의 알파벳 출현 횟수인 `letter`를 업데이트 할 수 있다는 장점이 있다.

Sliding Window 사용 시 더 효율적인 문제

[문제 링크] https://www.acmicpc.net/problem/5444

Sliding window로 풀 수 있는 이 문제는 2010년 ICPC Nortwestern European 지역문제로 출제되었던 '시리얼 넘버'라는 문제이며, 위의 문제처럼 개인적으로 블로그에 풀이법을 정리하여 올린 적이 있다. (블로그 링크)

서로 다른 시리얼 넘버가 적힌 N개의 기타 중에서 기타리스트 강토의 기타를 찾아야 하는데, 강토가 가지고 있는 기타는 시리얼 넘버를 모두 합하면 M의 배수가 된다. 이때, 가능한 강토의 기태 개수 중 최댓값을 구하는 것이 문제이다.

이 문제는 크게 보면 동적 프로그래밍(Dynamic Programming)으로 풀 수 있는 문제이지만, 공간복잡도 문제를 해결하기 위해 sliding window 기법을 사용할 필요가 있다.

Dynamic Programming 풀이 방법

먼저 동적 프로그래밍 풀이를 위해 다음과 같은 배열을 정의해야 한다. 이때 주목할 점은 바로 시리얼 넘버의 합이 M의 배수가 된다는 점이며, M의 배수인지에 관한 판별은 결국 시리얼 넘버의 합을 M으로 나누었을 때 나머지가 0으로 나누어 떨어지는지를 확인하면 된다. 이점을 주목하여 메모이제이션(memoization) 할 배열을 정의한다.

> **`dp[i][j]`** : 첫 번째부터 i번째 기타까지 본 상태에서 시리얼 넘버 합을 M으로 나누었을 때의 값이 j일 때 가능한 강토의 기타 개수의 최댓값

정의한 배열을 가지고 공간복잡도를 고려하지 않은 채 일반적인 동적 프로그래밍으로 풀이하면 다음과 같다.

1. 먼저 배열 `dp` 값을 모두 -1로 초기화한다.
2. i 번째 기타를 강토의 기타에 포함시키지 않는 경우 (`dp[i][j] < dp[i -1][j]`)
   배열 `dp` 에서 `i-1` 번째 원소까지 고려한 값으로 업데이트 한다. (`dp[i][j] = dp[i - 1][j]`)
3. i 번째 기타를 강토의 기타에 포함시키는 경우 (`dp[i][(j + s[i]) % m] < dp[i -1][j] + 1`)
   배열 `dp` 에서 `i-1` 번째 원소까지 고려한 값에 1을 더한 값으로 업데이트 하되 시리얼 넘버 합을 M으로 나누었을 때의 값이 `(j + s[i]) % m` 인 원소에 업데이트 한다. (`dp[i][(j + s[i]) % m] = dp[i - 1][j] + 1`)

Sliding Window 풀이 추가

이처럼 문제를 해결할 경우 문제의 조건에 따라 N의 최댓값이 500이고 M의 최댓값이 $100,000$이므로 배열 `dp` 를 위해 $500 \times 100,000$개 원소만큼 메모리를 할당해야 하는데, 이는 공간복잡도 면에서 효율적이지 않다.

사실 잘 생각해보면 모든 N개의 기타를 탐색하는 과정에서 i번째 기타를 살펴볼 때 $i-1$번째까지 살펴봤을 때의 정보만 필요하지, 그 이전의 정보는 사용하지 않으므로 N개의 모든 배열 `dp` 원소 값이 필요한 게 아니다. 이는 Markov assumption으로 이해해 볼 수 있으며, 위의 동적 프로그래밍 풀이 방법의 2, 3번에서의 관계식을 살펴봐도 알 수 있다.

배열 `dp` 를 `dp[500][100,000]` 가 아니라 `dp[2][100,000]` 만큼만 선언해서 구현할 수 있으며, 결국 크기가 2인 sliding window를 가지고 이동해가면서 배열 `dp` 를 업데이트 하는 것으로 볼 수 있다.

> **`dp[1][j]`** : i번째 기타 살펴볼 때 업데이트 할 강토가 지닐 수 있는 기타 개수의 최댓값
>
> **`dp[0][j]`** : $i-1$번째 기타까지 봤을 때의 강토가 지닐 수 있는 기타 개수의 최댓값

`dp[1][j]` 를 구했으면, 다음 $i+1$번째 업데이트를 수행하기 직전에 값을 `dp[0][j]` 에 업데이트하면 된다.

Sliding Window 사용이 효율적인 이유

동적 프로그래밍에 sliding window까지 사용하면 기존 동적 프로그래밍 풀이에서의 공간복잡도인 $O(NM)$에서 $O(M)$인 선형적인 복잡도로 줄일 수 있다는 장점이 있다. 문제에 메모리 제한이 존재한다면 sliding window를 사용하여 메모리 사용량을 줄여서 공간복잡도를 줄일 필요성이 있다.