CSE2011 Problem Solving, 2016 Spring

2013312343 이상헌

**Homework 4-1**

**1. 문제 이해**

(1) 문제

**Problem 4-1: zero-sum**

- For given one-dimensional array, develop an efficient way to find all consecutive sections, each of whose sum is zero.

- Example: 1 -2 3 -4 5 -3 2 1

- Then, the output is (1,6), (4,7), (6,8).

(2) 문제 정의

- 주어진 1차원 배열에서, 합이 0이 되는 여러 개의 연속된 숫자 조합을 가장 효율적으로 출력하는 방법을 설계하라.

**2. 문제 해결**

*(1) 연속된 두 숫자의 합을 미리 저장.*

- 연속된 숫자의 합이 0인 경우를 최소 access를 통해 출력해야 한다. 먼저, ‘효율적’인 solution을 배열 원소에 access하는 수가 가장 적은 solution으로 정의하였다. 즉, 문제에서의 기준은 원소 access time이다. 이를 최소화하기 위해서, 입력된 수에서의 이웃한 두 수를 따로 저장하는 방법을 생각해보았다.

- 아이디어

입력된 수열에서 이웃한 두 수의 합을 따로 저장한다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 수열 | 1 | | -2 | | 3 | | -4 | | 5 | | -3 | | 2 | | 1 | |
| 합 |  | -1 | | 1 | | -1 | | 1 | | 2 | | -1 | | 3 | |

Table 1. 입력된 수열에 대해 이웃한 두 수의 합.

시작점을 기준을 잡고, 결과 수열 (합이 0이 되는 이웃한 수들의 수열)의 원소 개수에 따라 크게 두 가지로 나눈다. 만일 결과 수열의 원소 개수가 홀수인 경우, 시작점에서 합을 시작한다. 시작점의 다음에 나타나는 두 수의 합부터 시작하여, 1칸 건너의 저장된 합을 더해가면서, 합이 0이 되는 지점을 찾고 저장한다. 예를 들어, 예시 수열에서 시작점이 1이고 홀수 개의 원소를 갖는 결과 수열을 찾고 싶을 때, Table 2에 표시된 수들을 더해간다. 즉, 1에서 시작하여 1+1, 1+1+1, 1+1+1+(-1)을 검사하여 0이 되는 경우를 찾는다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 수열 | **1** | | -2 | | 3 | | -4 | | 5 | | -3 | | 2 | | 1 | |
| 합 |  | -1 | | 1 | | -1 | | 1 | | 2 | | -1 | | 3 | |

Table 2. 시작점을 기준으로, 결과 수열의 원소 개수가 홀수 개인 경우.

만일 결과 수열의 원소 개수가 짝수인 경우, 시작점에 해당하는 합부터 시작하여, 1칸 건너의 저장된 합을 더해가면서, 합이 0이 되는 지점을 찾고 저장한다. 예를 들어, 예시 수열에서 시작점이 1이고 짝수 개의 원소를 갖는 결과 수열을 찾고 싶을 때, Table 3에 표시된 수들을 더해간다. 즉, -1에서 시작하여 -1+(-1), -1+(-1)+2, -1+(-1)+2+3을 검사하여 0이 되는 경우를 찾는다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 수열 | **1** | | -2 | | 3 | | -4 | | 5 | | -3 | | 2 | | 1 | |
| 합 |  | -1 | | 1 | | -1 | | 1 | | 2 | | -1 | | 3 | |

Table 3. 시작점을 기준으로, 결과 수열의 원소 개수가 짝수 개인 경우.

시작점은 수열의 가장 왼쪽 원소부터 오른쪽으로 하나씩 옮겨가며 바꾼다. 어떠한 시작점(1) 에서 검사를 완료하고, 한 칸 옮긴 새로운 시작점(-2) 에서는 앞의 수를 고려할 필요가 없다. 즉, 예시의 수열에서 -4를 시작점으로 검사하는 경우, 그의 왼쪽에 존재하는 1, -2, 3에 대한 검사는 할 필요가 없다. 이는 1, -2, 3이 시작점인 경우에 이미 검사를 완료했기 때문이다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 수열 | **1** | | -2 | | 3 | | **-4** | | 5 | | -3 | | 2 | | 1 | |
| 합 |  | -1 | | 1 | | -1 | | 1 | | 2 | | -1 | | 3 | |

Table 4. 시작점의 왼쪽에 있는 원소는 다시 검사할 필요가 없다.

- Access 수

입력된 수열의 개수가 N개인 경우, 이웃한 두 수의 합을 저장하는 수열을 만드는 데에 필요한 access 수는 (N-2)\*2 + 2\*1 = 2N-2 이다. 이는 양 끝의 원소는 1번의 access, 이외의 원소들은 2번의 access가 이루어지기 때문이다.

시작점을 기준으로 결과 수열을 도출하는 데에 필요한 access 수는 1번이다. 즉, 홀수 개의 원소를 갖는 결과 수열 검사에서 1번, 짝수 개의 원소를 갖는 결과 수열 검사에서 0번의 access만 이루어진다. 이는 이웃한 수의 합을 미리 따로 저장하였기 때문에, 검사에서 필요한 access는 기준점을 access하는 것만으로 사용되기 때문이다. 또한 결과 수열의 원소 개수가 짝수인 경우에는 기준점을 access할 필요가 없다.

시작점이 입력된 수열의 끝에서 두 번째 원소 혹은 가장 끝의 원소인 경우, 홀수 개의 원소를 갖는 결과 수열은 존재하지 않는다. 따라서 검사에 필요한 access수는 0번이다.

결과적으로 결과를 도출하는 데에 필요한 총 access 수는 (2N-2) + 1\*(N-2) + 0\*2 = 3N-4 번이다.

- 이점

미리 이웃하는 두 수의 합을 저장함으로써, 수열 원소의 access 수를 최소화하였다. 또한 모든 원소를 기준점으로 잡고 검사함으로써, 가능한 모든 결과를 정확하게 도출할 수 있다. N 개의 원소를 갖는 입력 수열에서 결과를 도출하는 데에 필요한 access 수는 3N-4 번 이다.

- 한계

이웃한 두 수의 합을 미리 저장하는 방법은, 입력된 수열의 원소 개수에 따라 효율적이거나, 효율적이지 않을 수 있다. 이웃한 두 수의 합을 미리 저장하는 것은, 시작 전에 모든 원소를 1번 이상 access하는 것이기 때문이다. 만일 수열의 원소 개수가 비교적 적은 경우, 이웃한 두 수의 합이 필요하지 않은 다른 방법을 통해 더 적은 access를 이룰 수 있다. 즉, 본 아이디어는 입력된 수열의 원소 개수가 비교적 큰 경우에 효율적으로 적용될 수 있다.

*(2) 원소에 직접 access, 누적 합을 저장.*

- 왼쪽부터 오른쪽으로 원소들을 하나씩 더해 가며 그 합계를 저장한다. 수식의 규칙에 따라 연속한 합이 0인 수열을 찾아낸다.

- 아이디어

초기 값이 0인 어떠한 integer 변수에 입력된 행렬(a[N])을 왼쪽부터 오른쪽으로 하나씩 더하고, 그 값을 다른 1차원 행렬(Sum[N])에 저장한다. 이후 각 단계에서의 Sum 값이 저장된 행렬에서 같은 값을 갖는 두 원소를 찾는다. 만일 두 원소의 위치가 각각 n과 m (n<m)이라 하면, 합이 0이 되는 이웃한 수열은 n+1부터 m까지의 초기 수열에 해당한다. 이는 a[n+1] 부터 a[m]까지의 합이 0이 되기 때문이다. 수식으로 나타내면 다음과 같다.

Sum[n] = a[0] + a[1] + … + a[n]

만일 Sum[m] = Sum[n] (m>n인 두 정수) 이라면

a[0] + a[1] + … + a[n] + a[n+1] + … + a[m] = a[0] + a[1] + … + a[n] 을 만족한다.

양변에 같은 수식이 있으므로 이항하여 제거하면

a[n+1] + a[n+2] + … + a[m] = 0 을 만족한다.

Table 5는 이에 대한 예시이다. N은 번째 수를 나타내고, a[N]은 입력된 행렬, Sum[N]은 a[0]부터 a[N]까지 합을 저장한 행렬이다. Sum 행렬에 저장된 수들 중에 같은 두 수를 찾으면, Sum[3]과 Sum[7], Sum[5]와 Sum[8]이 있다. 여기서, Sum 행렬을 만들기 위해 생성한 integer 변수의 초기값이 0이기 때문에 초기와 Sum[6] 또한 포함한다. 이에 따라 합이 0이 되는 이웃한 수들은 a[4]부터 a[7], a[6]부터 a[8], a[1]부터 a[6]이 된다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| a[N] | 1 | -2 | 3 | -4 | 5 | -3 | 2 | 1 |
| Sum[N] | 1 | -1 | 2 | -2 | 3 | 0 | 2 | 3 |

Table 5. 수열의 각 원소를 더한 Sum 행렬.

- Access 수

위와 같은 과정에서 access는 Sum 행렬을 만드는 과정에서만 발생한다. Sum 행렬을 만들 때에 각 원소마다 1번의 access를 하고, 결국 총 access 수는 입력된 행렬의 원소 수와 같다. 입력된 수열의 원소의 개수가 N개라면, 총 access 수는 N 번 이다.

- 이점

총 access 수는 N 번, 즉 time complexity는 O(N)으로, 비교적 효율적인 결과를 나타낸다. 또한 한 번의 합 과정을 통해 정확한 결과를 도출할 수 있기 때문에, 실제 프로그램 구현 시에 복잡하지 않고, 적은 양의 코드로 구현 가능하다. 자료 구조 또한 입력 행렬 외에 같은 크기의 1차원 배열 하나(Sum) 만을 필요로 하기 때문에 간단한 구조로 나타낼 수 있다.

**3. 고찰**

본 문제는 입력된 배열에서 이웃한 수들의 합이 0인 수열을 가장 효율적으로 찾아내는 알고리즘을 만드는 것이다. 먼저, 문제에서 말한 ‘효율적’인 solution을 ‘입력 수열의 원소에 access하는 수’라는 임의적인 기준을 토대로 분석하였다. 문제를 해결하는 과정에서 크게 두 가지의 알고리즘을 생각해보았다.

두 알고리즘의 효율성은 총 access 수로 비교할 수 있다. 입력 수열의 원소의 개수가 N개라고 가정했을 때, 첫 번째 알고리즘의 총 access 수는 3N-4이고, 두 번째 알고리즘의 총 access 수는 N이다. 이 둘의 효율성의 경계는 3N-4 = N인 경우로, N = 2인 경우이다. 즉, 입력 배열의 원소의 개수가 2보다 작은 경우에는 첫 번째 알고리즘이 더 효율적이고, 2보다 큰 경우에는 두 번째 알고리즘이 더 효율적이라고 할 수 있다. 2인 경우에는 두 알고리즘에서 같은 효율성을 나타낸다. 보통 어느 정도 큰 N에 대해서는 두 번째 알고리즘이 더욱 효율적이라고 판단되었다.

두 알고리즘의 정확성은 예시 및 설명을 통해 증명되었다. 결론적으로 가장 효율적인 알고리즘은 두 번째 알고리즘이다. 이보다 더욱 효율적인 알고리즘이 존재할 지도 모르지만, 총 access수, 구현 시에 코드의 양 및 복잡성, 필요한 자료 구조 등의 방면으로 보았을 때, 매우 효율적이고 정확한 알고리즘이라고 생각한다.

**4. 참고문헌 및 사용**

- 이진규 교수님, [문제해결기법] CSE2011\_2016spring\_Lecture\_Note06, 2016