컴퓨터공학과 20211507 김동건

Two Pointers를 사용하여 더 효율적으로 풀 수 있는 문제

정수 n개로 이루어진 **정렬된** 배열 A가 주어진다. 이때, 합이 가장 0에 가까운 두 정수를 찾는 문제가 있다. 가장 나이브한 방식은 O(n^2)에 모든 쌍을 확인하는 것이다. 우리는 이 문제를 Two Pointers를 이용하여 풀 수 있다. 배열이 정렬되어 있음을 이용하자. 풀이에 앞서 배열 A의 인덱스는 1-based이다. 즉, 1 ~ n의 값을 가진다. 우선, 왼쪽 포인터를 1에 두고, 오른쪽 포인터를 n에 두자. 왼쪽 포인터는 오른쪽으로만 이동하고, 오른쪽 포인터는 왼쪽으로만 이동하게 알고리즘을 짤 것이다. 그 후, 두 포인터가 서로 엇갈리기 전까지 포인터를 움직여가며 문제를 해결하면 된다. 만약 현재 a[left] + a[right]의 절댓값이 지금까지 구한 답보다 작다면 답을 갱신한다. 이제 포인터를 움직여줄 차례이다. 만약 두 값의 합이 0보다 작다면 left 포인터를 1 증가시켜준다. right 포인터가 더 클 때는 이미 다 고려되었다. 반대의 경우는 right 포인터를 1 감소시켜준다. 이를 반복하면 된다. 각 포인터는 최대 N번 움직이므로, 기존 방식보다 빠른 O(N)에 문제를 해결하였다. 수도코드로 나타내면 아래와 같다.  
  
left = 1, right =n, ans = INF

while (left < right) {

if( abs(a[left] + a[right] < ans ) 답 갱신

if(a[left] + a[right] < 0) left++;

else right--;

}  
  
이때, a[left] + a[right]인 경우 left를 증가시켜도 되는 이유는, right가 고정되어 있을 때, left가 더 작아지면 0에서 더 멀어지기 때문이다. right는 오른쪽에서 왔기 때문에 right가 더 큰 경우는 이미 고려된 상태이다. else의 경우도 비슷하게 보일 수 있다.

Sliding window를 사용하여 더 효율적으로 풀 수 있는 문제

정수 n개로 이루어진 **정렬된** 배열 a와 k(1<=k<=n-2)가 주어진다. 이때, k개의 수를 제거해서 남은 수열을 b라하자. b에서 모든 쌍의 차이 중 가장 큰 값을 M, 가장 작은 값을 m이라하자. M + m의 최솟값을 구해야 하는 문제가 있다고 하자.

우선 나이브하게 k개를 지우는 모든 경우를 보게 되면, O(2^k \* n^2)의 시간복잡도가 든다. k개를 지우는 모든 경우와, 모든 쌍을 확인하는 시간복잡도가 곱해진 것이다.  
 좀 더 좋은 풀이를 생각해보자. 답에 대해서 m을 이루는 두 수는 무조건 인접해야 한다. 이는 자명하다. 또, M을 최소로 만들려면 k개를 지우고 남은 수들이 모두 인접해야 한다. 그렇지 않다면 M은 더 커지기 때문이다. 즉, 우리는 모든 길이가 n-k인 연속한 구간 중 답이 있다는 사실을 알게 되었다. 여기까지만 보아도 슬라이딩 윈도우로 풀이의 시간복잡도를 다항시간으로 줄일 수 있게 되어 충분히 이득이지만, 끝까지 문제를 해결해보자. O(N)풀이를 소개한다.

구간이 정해지면 M은 쉽게 구할 수 있다. 양 끝값이기 때문이다. 그러나, 모든 인접한 곳의 차 중 가장 작은 것은 어렵다. 덱을 하나 관리하자. 덱에는 {인접한 원소 차이, 인덱스}를 pair로 관리한다. 이제, 왼쪽부터 길이가 n-k인 모든 구간을 살펴보자. 덱은 인접한 원소 차이의 오름차순으로 관리할 것이다. 그렇다면, 현재 위치에서 추가되는 인접한 원소 차이를 덱에 반영하기 전, 덱의 프론트에서 구간 길이가 n-k를 벗어나는 것을 모두 pop해준다. 그 후, 뒤쪽을 보면서 이번 차이보다 큰 값을 모두 pop해준다. 그 후, 덱에 뒤쪽에 이번 원소 차이를 넣자. 이 작업 후, 덱의 프론트에 있는 값이 m의 값이다.

각 원소마다 덱에 들어가는 것 한 번, 뽑히는 것 한 번이기 때문에 시간복잡도가 O(N)이다. 수도 코드로 살펴보자.

p = n - k, answer = inf

deque dq;

for i in (1,n] :

while( !dq.empty && dq.front().인덱스 <= i - p + 1) dq.pop\_front();  
 while( !dq.empty && dq.back().값 >= a[i] - a[i-1]) dq.pop\_back();  
 dq.push\_back({a[i]-a[i-1], i});  
 If(i < p) continue;

m = dq.front().값

M = a[i] - a[i - p + 1]

if m + M is less than answer then ans = m + M

return ans

위 의사코드를 보면 각 반복문은 길이가 p인 구간을 보고 있다는 것을 알 수 있다. 물론 덱에는 개수가 p보다 적을 수 있지만, 고려하는 구간이 p인 사실은 변하지 않는다. 즉 우리는 구간 길이 p를 한 칸씩 미루어 가며 답을 구한다고 생각할 수 있고, 이는 슬라이딩 윈도우 기법이다.