Two Pointers

Lecture 3: Summer of Competitive Programming

Rishabh Dhiman

Algorithms and Coding Club
Indian Institute of Technology Delhi

3 July 2021

Intervals with Small Sum

Problem

Given an array A of N positive integers, Find the number of subarrays with sum less than or equal to s.

For example, for $s = 7$ and the following array,

1	2	4	2	5	9	5	8

For this array, the intervals $[0, 1)$, $[0, 2)$, $[0, 3)$, $[1, 2)$, $[1, 3)$, $[2, 2)$, $[2, 4)$, $[3, 4)$, $[3, 5)$, $[4, 5)$, $[6, 7)$ work. So the answer is 11.

Intervals with Small Sum

Observation

We'll call a subarray with sum less than s *smol*.

Intervals with Small Sum

Observation

We'll call a subarray with sum less than s *smol*.
We make the following observations:

- If $[L, R)$ is *smol* then any interval $[L', R') \subset [L, R)$ is also *smol*.
- If $[L, R)$ is *not smol* then any interval $[L', R') \supset [L, R)$ is also *not smol*.

This lets us solve the problem in $O(N \log N)$ using binary search.

Algorithm

1. Iterate through all L.
2. Binary search to find the largest R such that $[L, R)$ is *smol*.
3. Add $R - L$ to the answer.

We only used the fact that if $[L, R)$ is *smol* then $[L, R')$ is *smol* for $R' \leq R$, and if $[L, R)$ is *not smol* then $[L, R')$ is *not smol* for $R' \geq R$.

Intervals with Small Sum

Two Pointers

In the previous section, while we made use of the fact that if $[L, R)$ is *smol* then $[L, R')$ is *smol* for $R' \leq R$ (and similar condition for *not smol*).
We did not make use of the fact that if $[L, R)$ is *smol* then $[L', R)$ is also *smol* for $L' \geq L$ (and similar condition for *not smol*),

In the previous section, while we made use of the fact that if $[L, R)$ is *smol* then $[L, R')$ is *smol* for $R' \leq R$ (and similar condition for *not smol*).
We did not make use of the fact that if $[L, R)$ is *smol* then $[L', R)$ is also *smol* for $L' \geq L$ (and similar condition for *not smol*),

Algorithm Idea

As in the previous algorithm, we iterate through all L. Along with this we also keep track of R.
When we move from $L \to L+1$, $R' \geq R$, since $[L+1, R)$ is *smol*.

Intervals with Small Sum

Two Pointers

$L = 0 \quad R = 3$

1	2	4	2	5	9	5	8

Intervals with Small Sum

Two Pointers

$L = 0$	$R = 3$	1	2	4	2	5	9	5	8
$L = 1$	$R = 3$	1	2	4	2	5	9	5	8

Intervals with Small Sum

Two Pointers

$L = 0$	$R = 3$	1	2	4	2	5	9	5	8
$L = 1$	$R = 3$	1	2	4	2	5	9	5	8
$L = 2$	$R = 4$	1	2	4	2	5	9	5	8

Intervals with Small Sum

Two Pointers

$L = 0$	$R = 3$	1	2	4	2	5	9	5	8
$L = 1$	$R = 3$	1	2	4	2	5	9	5	8
$L = 2$	$R = 4$	1	2	4	2	5	9	5	8
$L = 3$	$R = 5$	1	2	4	2	5	9	5	8

Intervals with Small Sum

Two Pointers

$L = 0$	$R = 3$	1	2	4	2	5	9	5	8
$L = 1$	$R = 3$	1	2	4	2	5	9	5	8
$L = 2$	$R = 4$	1	2	4	2	5	9	5	8
$L = 3$	$R = 5$	1	2	4	2	5	9	5	8
$L = 4$	$R = 5$	1	2	4	2	5	9	5	8
$L = 5$	$R = 5$	1	2	4	2	5	9	5	8
$L = 6$	$R = 7$	1	2	4	2	5	9	5	8
$L = 6$	$R = 7$	1	2	4	2	5	9	5	8

Intervals with Small Sum

Two Pointers

Algorithm

```
long long ans = 0, sum = 0;
for (int l = 0, r = 0; l < n; ++l) {
    while (r < n && sum + a[r] <= s) {
        sum += a[r];
        ++r;
    }
    ans += r - l;
    sum -= a[l];
}
```

In each iteration, `l` moves forward by one. So the time due to movement of `l` is $\sum_{l \in [0,N)} 1 = N$.
In each iteration, `r` may move arbitrarily times. However,

- `r` only increases,
- `r` starts at 0 and ends at a value $< N$.

So the time due to movement of `r` is $\leq N$.
Thus, the total time complexity of the algorithm is $O(N)$.

Skeleton of Two Pointers Method

1. Consists of keeping track of two pointers across multiple iterations.
2. Move the pointers monotonically while maintaining some invariants.
3. Stop the current iteration once a condition has been achieved.

The number of pointer moves in each iteration may be unbounded but the total number of moves is bounded.

Common Variants of Two Pointers Method

None of these names are standard.

1. Maximal good intervals
2. Disjoint interval splitting
3. Forward-backward Iteration
4. Sliding window

Maximal good intervals

An example of this is the problem we worked with earlier. For a fixed endpoint we have to find the maximal interval which satisfy some property. The code usually resembles,

```
for (int l = 0, r = 0; l < n; ++l) {
    while (good()) {
        add(a[r]);
        ++r;
    }
    // Process [l, r)
    remove(a[l]);
}
```

Common Variants of Two Pointers Method

Maximal good intervals

```cpp
for (int l = 0, r = 0; r < n; ++r) {
    add(a[r]);
    while (!good()) {
        remove(a[l]);
        ++l;
    }
    // Process [l, r]
}
```

Common Variants of Two Pointers Method
Disjoint Interval Splitting

We wish to split the array into disjoint intervals such that each interval satisfies some property.
An example problem would be the following:

Problem

Given an array A of N integers, split it into intervals such that each interval is nondecreasing, In other words, find it into $0 = i_1 < i_2 < i_3 < \cdots < i_k = n$ such that $A_j \leq A_{j+1}$ for each $i_r \leq j < i_{r+1}$ and $A_{i_r} > A_{i_r+1}$.

For example, given the array $\{1, 2, 3, 2, 2, 1\}$. We'll split it into $\{1, 2, 3\}$, $\{2, 2\}$ and $\{1\}$.

Common Variants of Two Pointers Method

Disjoint Interval Splitting

```
for (int l = 0, r; l < n; l = r) {
    r = l + 1;
    while (r < n && a[r - 1] <= a[r]) ++r;
    // Process [l, r)
}
```

The generic code looks like,

```
for (int l = 0, r; l < n; l = r) {
    reset(); add(a[l]);
    while (good()) add(a[r]), ++r;
    // Process [l, r)
}
```

Common Variants of Two Pointers Method

Forward-backward Iteration

An example of this would be the 2SUM problem.

Problem

Given a sorted array A of N integers, find any two integers which sum to x.

```
for (int l = 0, r = n - 1; l < r; ) {
    if (a[l] + a[r] < x) ++l;
    else if (a[l] + a[r] > x) --r;
    else {
        // found
        break;
    }
}
```

Common Variants of Two Pointers Method

Forward-backward Iteration

```
for (int l = 0, r = n - 1; l < r; ) {
    if (condition_1(l, r)) ++l;
    else if (condition_2(l, r)) --r;
    else // found
}
```