Two pointers

IICPC - Ashish Ahuja

Intro

Roughly speaking, we have two pointers which are used to iterate through an array(s) with each pointer moving in a fixed direction.

As each pointer moves in one direction, the algorithm is efficient. What exactly we do with the pointers depends on the exact problem.

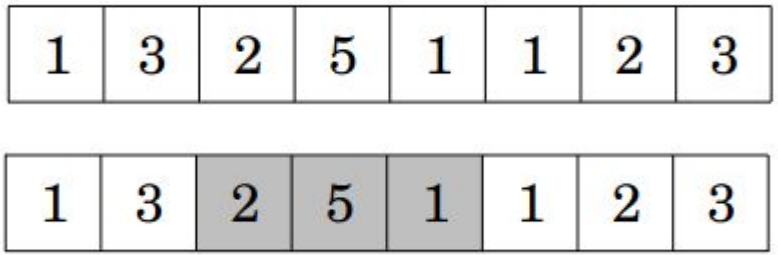
https://usaco.guide/animations/two_pointers.mp4

Subarray Sum

Given an array of n integers and a target sum S, find any subarray whose sum is S (or report that there isn’t one).

Can be solved in O(n) using two pointers.

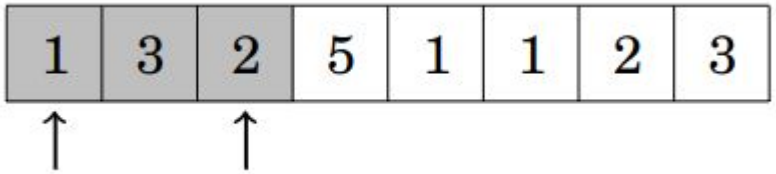
For eg, say n = 8, S = 8.



Subarray Sum

Idea: we maintain two pointers, where the left pointer points to the start of the subarray under consideration, and the right pointer points to the end of the subarray under consideration.

We keep moving the right pointer to the right till the sum of the subarray b/w the two pointers is at most S.



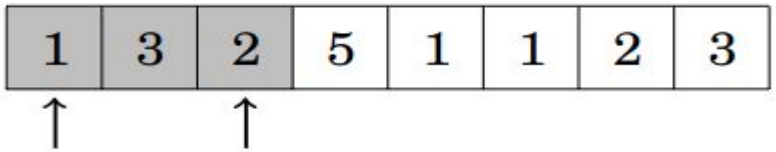
Subarray Sum

Both the pointers point to the first element initially.

We keep moving the right pointer to the right till the sum of the subarray under consideration is less than equal to S.

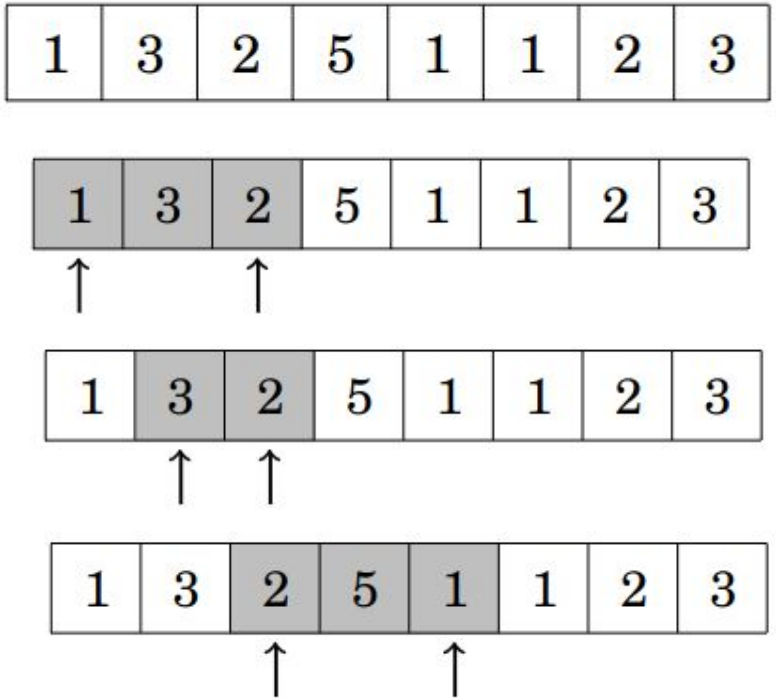
If we find a subarray with sum exactly equal to S, we are done.

Else, we move the left pointer to the right by one step and repeat.



Subarray Sum

S = 8



Subarray Sum

Time Complexity?

Why two pointers?

In many problems where two pointers is applicable, it helps us reduce the time complexity by a factor of log(n).

In certain cases it also helps in reducing the space complexity, as we will see soon - Floyd’s Tortoise and Hare algorithm finds cycles in functional graphs with a space complexity of O(1) compared to the usual space complexity of O(n)!

Easy to implement, easy to estimate the time complexity (once you’ve done a few problems).

So what is two pointers?

Sort of like a subset of greedy, similar to sliding window problems at times.

Basically a technique where we use two pointers to traverse the given data till we reach a suitable stopping point, such as the end of data, a collision, etc.

Variations of two pointers exist, such as the fast-slow pointers used to detect cycles in a functional graph.

What will be covered in the remaining lecture

1. A few more basic problems.
2. A few problems following a general pattern.
3. A different way to apply two pointers to find cycles in functional graphs (Floyd’s Tortoise and Hare Algorithm).

Sum of Two Values

https://www.cses.fi/problemset/task/1640 Done

Sum of Two Values

Multiple methods (first obvious method is to just use a set), but we'll do it using two pointers.

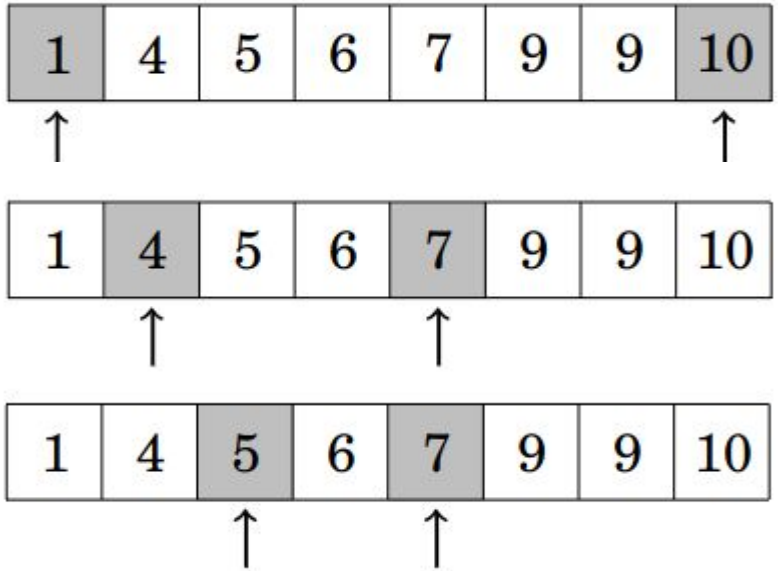
First, we sort.

Left pointer at first element, right pointer at last element.

Right pointer moves to the left until the sum of the values at the two pointers is at most x.

Stop when the sum is exactly x or at a collision.

Sum of Two Values



Sum of Two Values

Time complexity?

Each pointer moves at most N times, so O(N)?

Books

https://codeforces.com/contest/279/problem/B

Done

Books

https://usaco.guide/animations/two_pointers.mp4

The code has two nested loops but the complexity is still O(N) because each pointer moves at most N times.

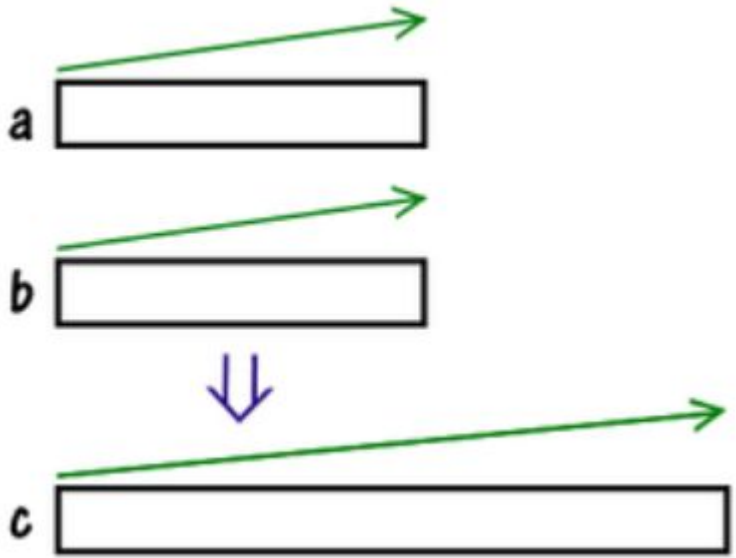
Merging Two Arrays

So far the problems were on a single array, this is one where each pointer points to elements in different arrays.

Problem: Given two arrays sorted in ascending order, merge them into a single array also in ascending order.

Obvious solution: Extend one array with the other and sort it. But is a linear time solution possible?

Merging Two Arrays



Merging Two Arrays

Idea using two pointers: Maintain two pointers, say X and Y for the two arrays A and B. Both point to the first elements of their respective arrays.

Push the smaller of the two values being pointed at to the new array. If say the value pointed to by X was smaller, we push that value and move X to the right.

Ends when both X and Y hit the ends of their respective arrays.

Merging Two Arrays

Time complexity?

$$a = [\cancel{1}, \overset{\overset{i}{\downarrow}}{3}, 5, 8, 10]$$

$$b = [\cancel{2}, \underset{\underset{j}{\uparrow}}{6}, 7, 9, 13]$$

$$c = [1, 2]$$

$$a = [\cancel{1}, \cancel{3}, \cancel{5}, \cancel{8}, \cancel{10}]^{\overset{i}{\downarrow}}$$

$$b = [\cancel{2}, \cancel{6}, \cancel{7}, \cancel{9}, \underset{\underset{j}{\uparrow}}{13}]$$

$$c = [1, 2, 3, 5, 6, 7, 8, 9, 10]$$

Segment with Good Sum

Given an array of non-negative integers, find the length of the longest subarray [L;R] such that its sum does not exceed S.

Segment with Good Sum

Problem: Given an array of non-negative integers, find the length of the longest subarray [L;R] such that its sum does not exceed S.

Idea: We maintain two pointers L and R, and the sum of the segment from L to R.

We keep moving the right pointer to the right till the sum is less than equal to S.

Once the sum exceeds S, we move the left pointer to the right till the sum of the segment again drops below S.

Can now easily maintain the length of the longest valid subarray.

Subarray with Good Sum

Time complexity?

```
x = 0, L = 0
for R = 0..n-1
    x += a[R]
    while x > s:
        x -= a[L]
        L++
    res = max(res, R - L + 1)
```

Variation: Subarray with Good Sum

Instead of a valid subarray being one with sum less than equal to S, we want the subarray to have a sum greater than equal to S.

Another variation would be to also count the number of these longest valid subarrays.

A general pattern

This last problem (and its variations) all follow a fairly standard pattern which can be generalised:

Problem: Find the longest/shortest segment or count the number of segments satisfying a certain property (which we will call a “good” segment).

When can we use two pointers for such problems?

A general pattern

Problem: Find the longest/shortest segment or count the number of segments satisfying a certain property (which we will call a “good” segment).

Firstly, one of the following two must hold:

1. If the segment [L;R] is good, any segment nested in it is also good.
2. If the segment [L;R] is good, any segment containing it is also good.

Secondly, we must be able to re-check whether the current segment is good or bad while expanding or compressing the segment by one element.

A general pattern

Problem: Find the longest/shortest segment or count the number of segments satisfying a certain property (which we will call a “good” segment).

```
L = 0
for R = 0..n-1
    add(a[R])
    while not good():
        remove(a[L])
        L++
```

Segment with a small set

Problem: Given an array a, find a segment [L;R] of maximum length which has at most k distinct elements.

Can this be solved using two pointers?

Segment with a small set

Problem: Given an array a, find a segment [L;R] of maximum length which has at most k distinct elements.

Let’s check the first condition.

If a segment [L;R] does not have more than k distinct elements, then any nested segment in [L;R] also won’t have more than k distinct elements.

Segment with a small set

What about the second condition?

Need to be able to maintain a structure which counts the number of distinct elements in the current segment.

Can easily be done using a map or multiset, and if the elements are small enough we can also use an array to do it in linear time.

Segment with a small set

Say the elements are all fairly small (say <= N) so that we can use an array for simplicity. Let’s look at the exact implementation.

If the elements are big (say <= 1e9), my preferred solution would be to just use a map/unordered_map.

Segment with a small set

```
cnt[x] - how many positions exist L <= i <= R, such that a[i] = x

num - how many distinct numbers on the segment L..R exists

good():
    return num <= k

add():
    cnt[x]++
    if cnt[x] == 1:
        num++
```

Segment with a small set

```
add():
    cnt[x]++
    if cnt[x] == 1:
        num++

remove():
    cnt[x]--
    if cnt[x] == 0:
        num--
```

Floyd’s Tortoise and Hare Algorithm

An application of two pointers which is fairly different from the examples we have seen so far. Also called many other names such as Floyd’s Cycle Finding Algorithm.

It is used to find cycles in a functional graph using two pointers, one of which is “fast” while the other being “slow” (the tortoise and the hare).

https://visualgo.net/en/cyclefinding

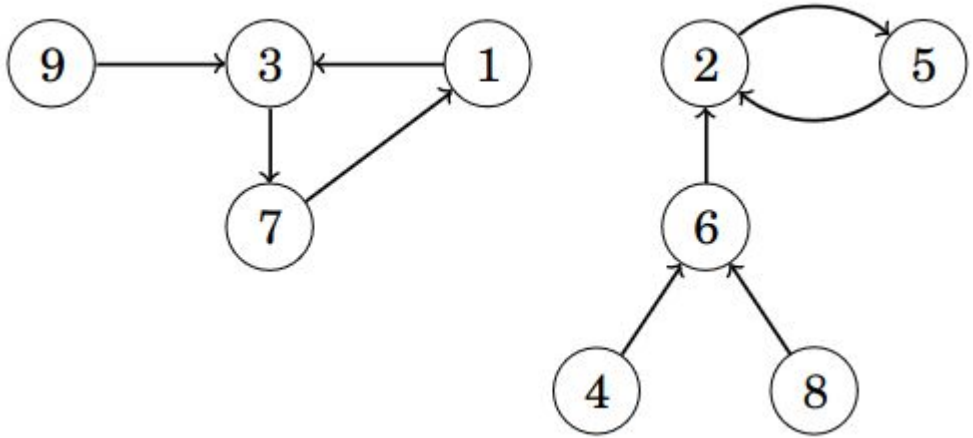
What is a functional graph?

Also called successor graph, it is basically a directed graph where every vertex has exactly one outgoing edge.

Can also think of it like a singly linked list (which may have cycles), or like any function which maps a finite set to itself.

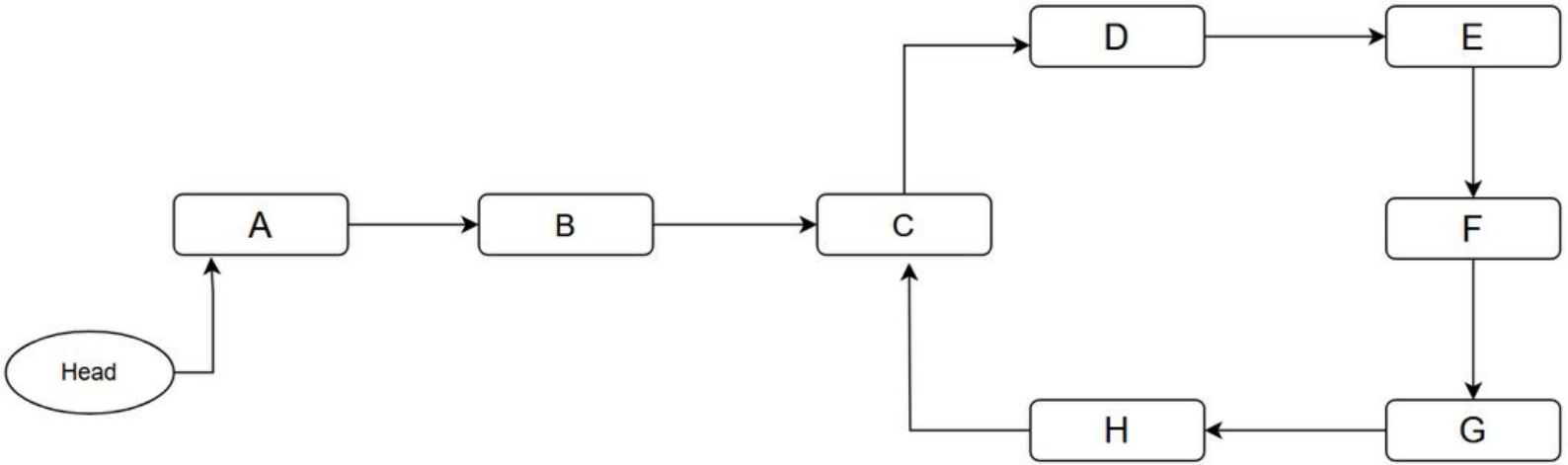
What is a functional graph?

x	1	2	3	4	5	6	7	8	9
succ(x)	3	5	7	6	2	2	1	6	3

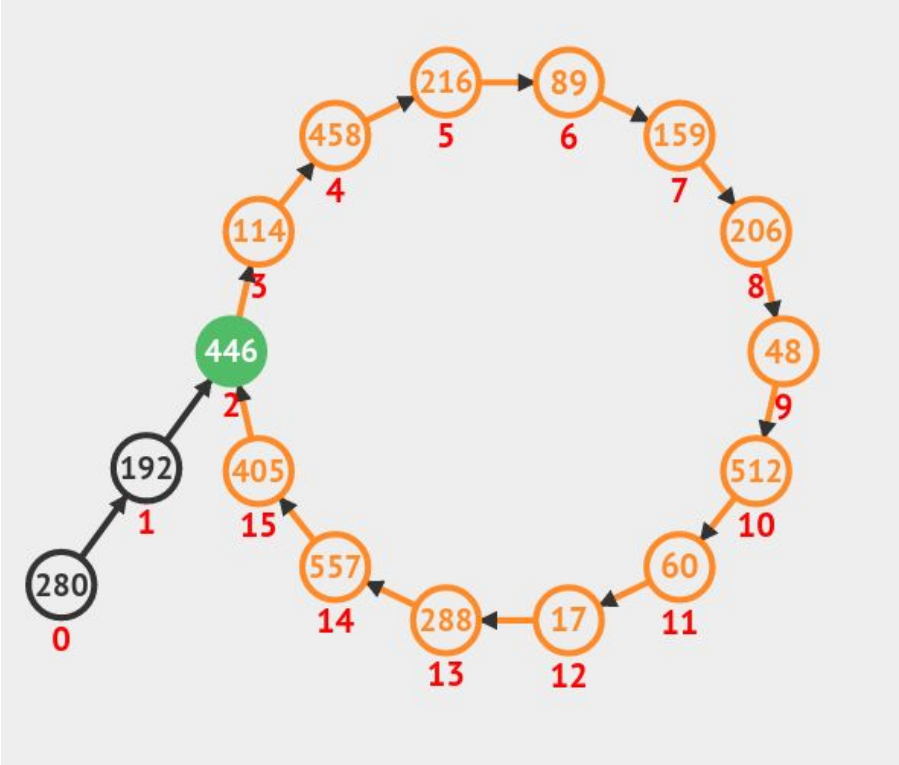


What is a functional graph?

Like a linked list.



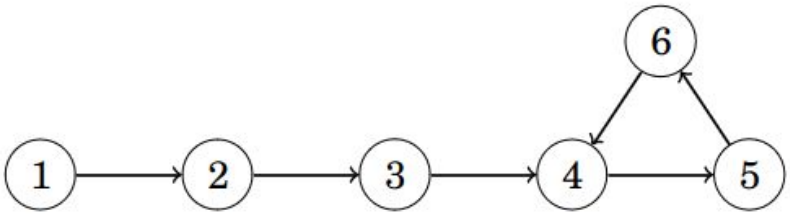
What is a functional graph?



The target of the algorithm

1. To check whether a cycle exists.
2. If a cycle exists, to find the following:
   a. The first node in the cycle.
   b. The length of the cycle.

We want to calculate the above goals for a functional graph which contains only a path which (possibly) ends in a cycle.



Isn’t this straightforward to solve?

A fairly simple solution to this problem is just to walk the graph starting from the head while maintaining an array of which nodes have been visited. If while walking the graph we arrive at a node which has already been visited, we know that that is the first node in the cycle and we can also easily get the length of the cycle.

While this solution has a time complexity of O(N), it also has a space complexity of O(N) (because of the visited array).

Using Floyd’s Algorithm we can reduce the space complexity to O(1) (disregarding the space used to store the graph itself).

The algorithm

We maintain two pointers, a slow pointer (the tortoise) and a fast pointer (the hare). Both the pointers start at the head.

On each step we move the slow pointer ahead by one edge and the fast pointer ahead by two edges.

Checking if a cycle exists

If at any point both of them point to the same node (i.e they collide) a cycle exists.

Or else, there is no cycle.

We stop when the two pointers collide or when one of the pointer reaches a node which has no outgoing edge.

Finding the starting point

We move the slow (a) and fast (b) pointers till they meet each other. We reset say pointer “a” to the head, and move both pointers one step at a time till they meet again. This is the starting point.

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

Finding the length

Fairly self-explanatory.

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

Why does this work?

https://visualgo.net/en/cyclefinding

The methods to check whether a cycle exists and finding the length if it does is fairly intuitive.

For the former once both the nodes are in the cycle the fast pointer “catches up” to the slow pointer. For the latter the length is just iterating over the cycle from the starting point.

Why the method to find the starting point works has a more complicated proof which we will omit here (is also a problem in Knuth).

Acknowledgements

Resources used:

1. cp-algorithms.com
2. USACO Guide
3. Codeforces EDU
4. CP Handbook
5. And others.

Thanks.