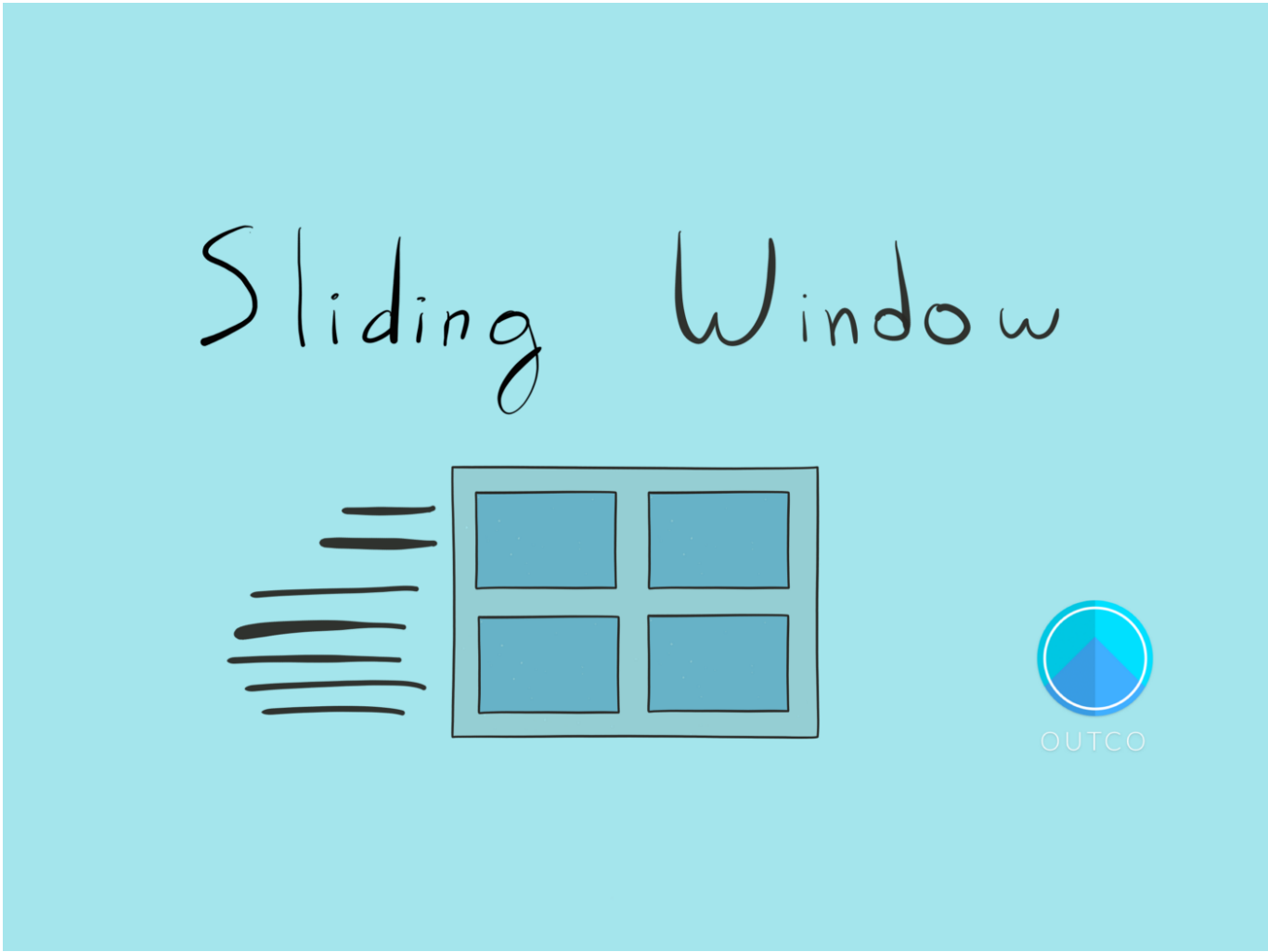
# How to Solve Sliding Window Problems

## An Intro To Dynamic Programming

Sergey Piterman Follow

Sep 29, 2018 · 11 min read




**Sliding Window** problems are a type of problem that frequently gets asked during software engineering interviews and one we teach at Outco. They are a subset of dynamic programming problems, though the approach to solving them is quite different from the one used in solving *tabulation* or

*memoization* problems. So different in fact, that to a lot of engineers it isn't immediately clear that there even is a connection between the two at all.

This blog post aims to clear up a lot of confusion around solving this kind of problem and answer some common questions engineers typically have. Hopefully, it will show that the approach is actually relatively straightforward if you have the right thinking, and once you solve a few of these problems you should be able to solve any variation of them that gets thrown your way.

### How do you identify them?

So the first thing you want to be able to do is to identify a problem that uses a sliding window paradigm. Luckily, there are some common giveaways:

- The problem will involve a data structure that is ordered and iterable like an array or a string
- You are looking for some subrange in that array/string, like a longest, shortest or target value.
- There is an apparent naive or brute force solution that runs in $O(N^2)$, O(2^N) or some other large time complexity.

But the biggest giveaway is that the thing you are looking for is often some kind of **optimal**, like the **longest** sequence or **shortest** sequence of something that satisfies a given condition **exactly**.

And the amazing thing about sliding window problems is that most of the time they can be solved in O(N) time and O(1) space complexity.

For example, in **Bit Flip**, you are looking for the **longest** continuous sequence of 1s that you can form in a given array of 0s and 1s, if you have the ability to flip some number of those 0s to 1s.

Maximize number of 0s by flipping a subarray - GeeksforGeeks

Given a binary array, find the maximum number zeros in an array with one flip of a subarray allowed. A flip operation...

www.geeksforgeeks.org

In **Minimum Window Substring**, you are looking for the **shortest** sequence of characters in a string that contains all of the characters in a given set.

Minimum Window Substring - LeetCode

Minimum Window Substring - LeetCode

Level up your coding skills and quickly land a job. This is the best place to expand your knowledge and get prepared...

leetcode.com

### Why is this dynamic programming?

This search for an **optimum** hints at the relationship between sliding window problems and other dynamic problems. You are using the **optimal substructure** property of the problem to guarantee that an optimal solution to a subproblem can be reused to help you find the optimal solution to a larger problem.

Optimal Substructure Property in Dynamic Programming | DP-2 - GeeksforGeeks

As we discussed in Set 1, following are the two main properties of a problem that suggest that the given problem can be...

www.geeksforgeeks.org

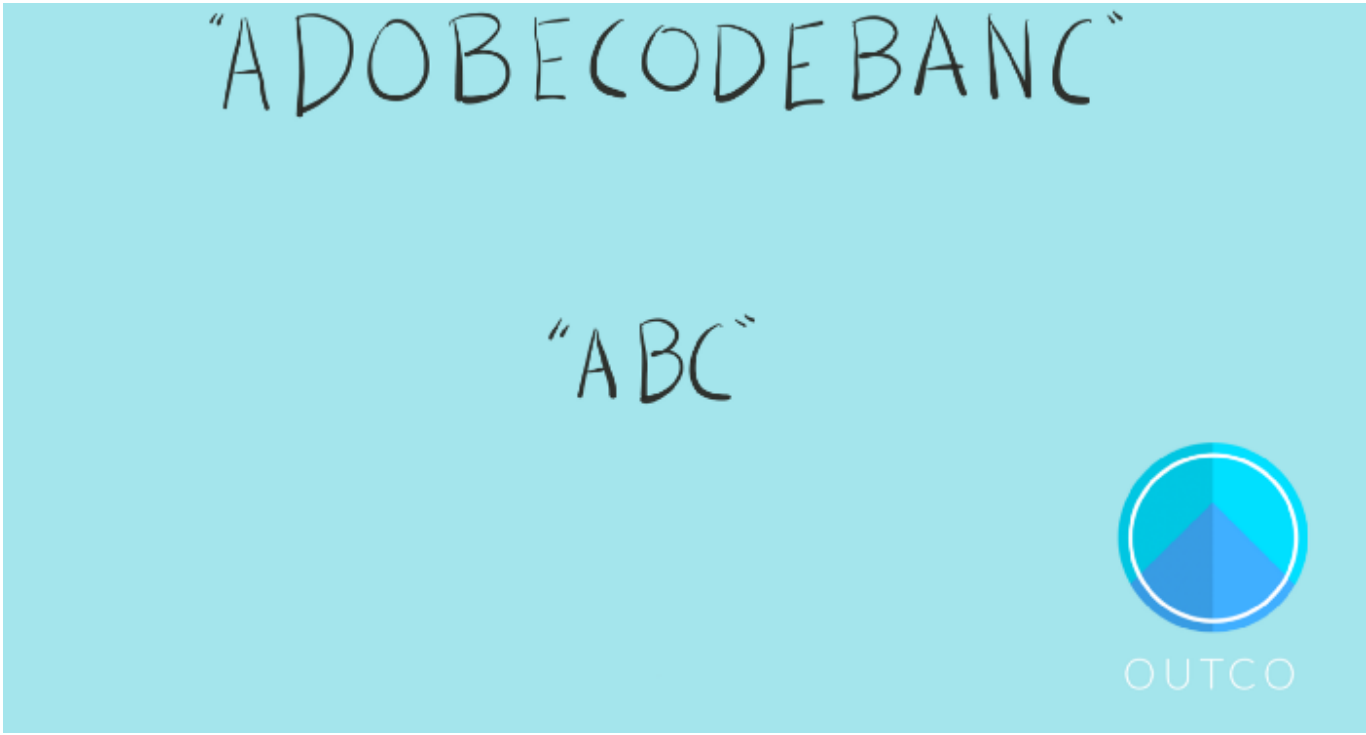
You are also using the fact that there are **overlapping subproblems** in the naive approach, to reduce the amount of work you have to do. Take the **Minimum Window Substring** problem. You are given a string, and a set of

characters you need to look for in that string. There might be multiple overlapping substrings that contain all the characters you are looking for, but you only want the shortest one. These characters can also be in any order.

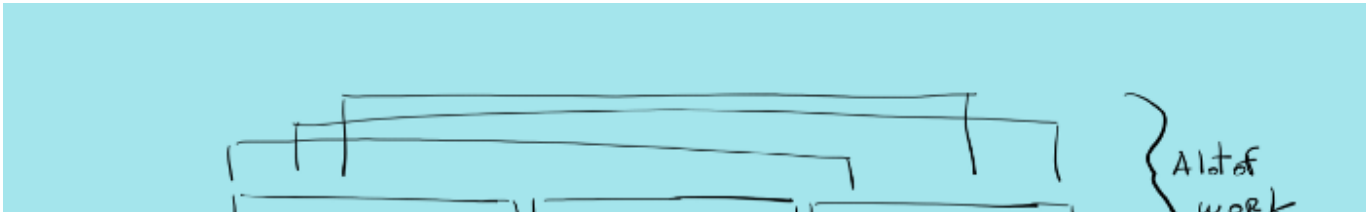


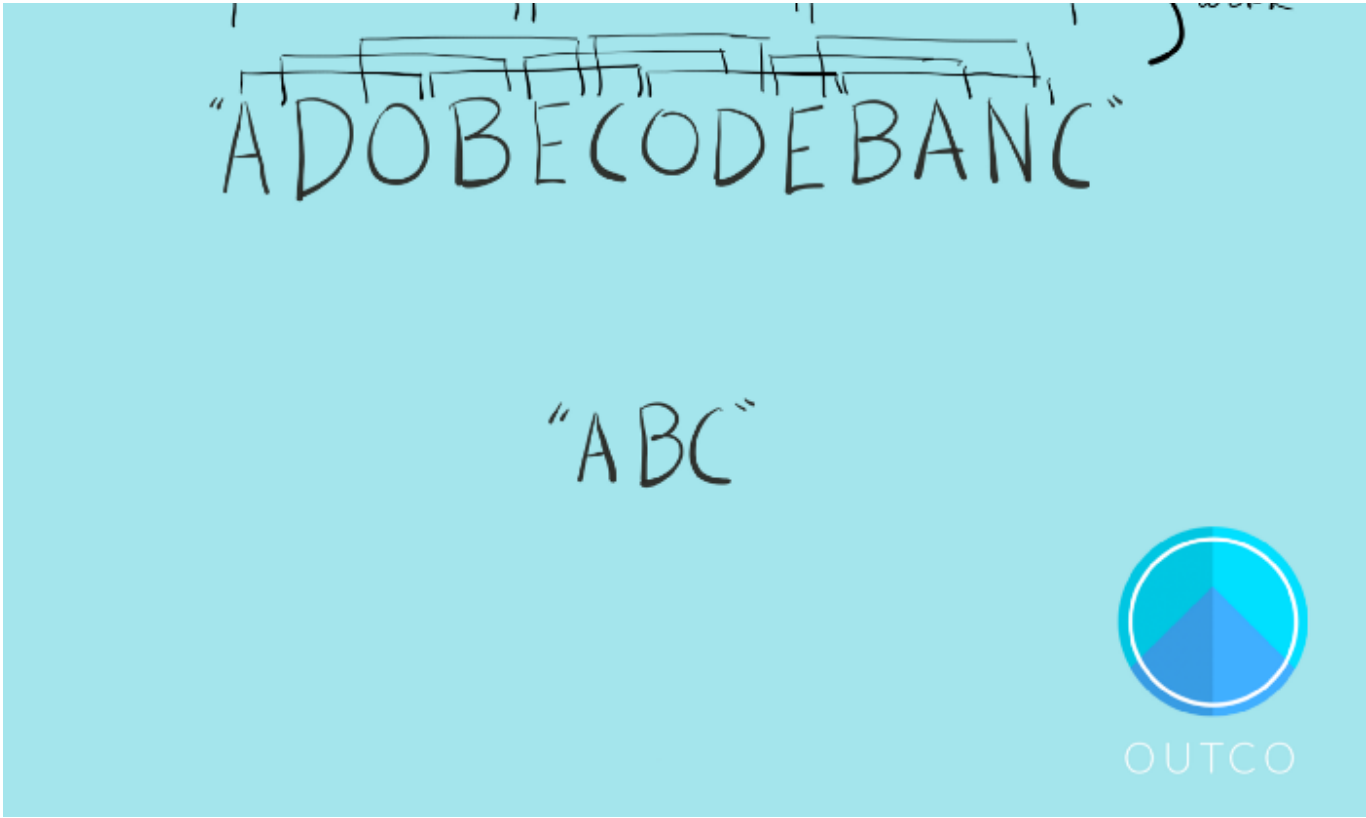
Here's an example:

```
String: "ADOBECODEBANC"
Characters: "ABC"
```




The naive way to approach this would be to first, scan through the string, looking at ALL the substrings of length 3, and check to see if they contain the characters you're looking for. If you can't find any of length 3, then try all substrings of length 4, and 5, and 6, and 7 and so on until you reach the length of the string.






If you reach that point, you know that those characters are not in there.

This is really inefficient and runs in $O(N^2)$ time. And what's happening is that you are missing out a lot of good information on each pass by constraining yourself to look at fixed length windows, and you're re-examining a lot of parts of the string that don't need to be re-examined.

**You're throwing out a lot of good work, and you're redoing a lot of useless work .**

This is where the idea of a **window** comes in.

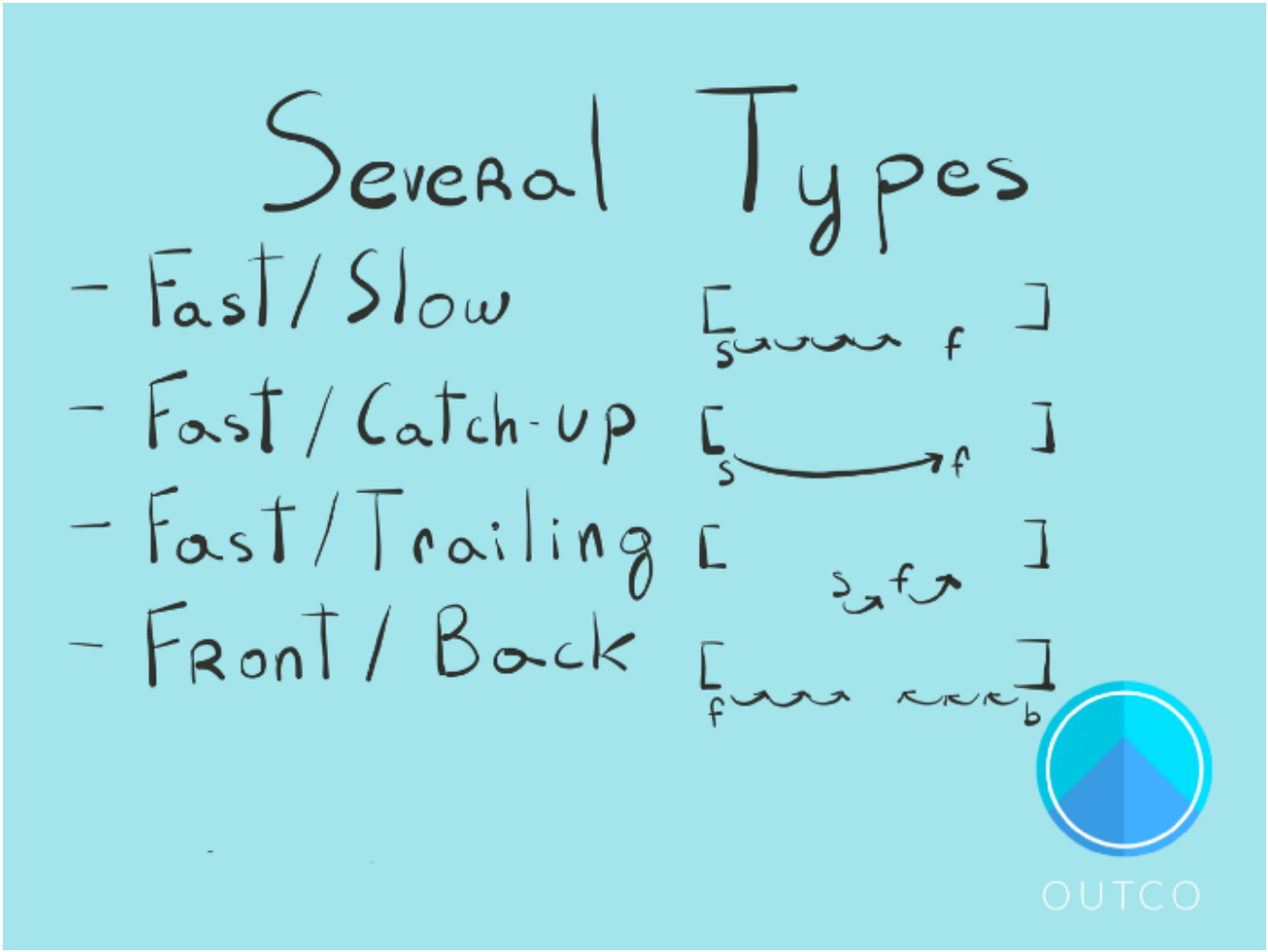
Your window represents the current section of the string/array that you are "looking at" and usually there is some information stored along with it in the form of constants. At the very least it will have 2 pointers, one indicating the index corresponding beginning of the window, and one indicating the end of the window.

You usually want to keep track of the previous best solution you've found if any, and some other current information about the window that takes up O(1) space. I see a lot of engineers get tripped up by O(1) space, but all this means the amount of memory you use doesn't scale with the input size. So things like a **current_sum** variable, or **number** of bit flips (in the bit flip problem) remaining, or even the number of characters that you still need to find (since there is a fixed number of ASCII characters).

But once you have what variables you want to store figured out, all you have to think about then are two things: **When do I grow this window? And when do I shrink it?**

### Different Kinds of Windows

There are several kinds of sliding window problems. The main one we'll talk about today is the first kind, but there are a few others worth mentioning that will make their way into a different post.



**1) Fast/Slow**

These ones have a fast pointer that grows your window under a certain condition. So for **Minimum Window Substring**, you want to grow your window until you have a window that contains **all** the characters you're looking for.

It will also have a slow pointer, that shrinks the window. Once you find a valid window with the fast pointer, you want to start sliding the slow pointer up until you no longer have a valid window.

So in the **Minimum Window Substring** problem, once you have a substring that contains all the characters you're looking for, then you want to start shrinking it by moving the slow pointer up until you no longer have a valid substring (meaning you no longer have all the characters you're looking for)

**2) Fast/Catchup**

This is very similar to the first kind, except, instead of incrementing the slow pointer up, you simply move it up the fast pointer's location and then

keep moving the fast pointer up. It sort of “jumps” to the index of the fast pointer when a certain condition is met.

This is apparent in the **Max Consecutive Sum** problem. Here you’re given a list of integers, positive and negative, and you are looking for a consecutive sequence that sums to the largest amount. Key insight: **The slow pointer “jumps” to the fast pointer’s index when the current sum ends up being negative**. More on how this works later.

For example, in the array: `[1, 2, 3, -7, 7, 2, -12, 6]` the result would be: `9 (7 + 2)`

Again, you’re looking for some kind of **optimum** (ie the maximum sum).

**3) Fast/Lagging**

This one is a little different, but essentially the slow pointer is simply referencing one or two indices behind the fast pointer and it’s keeping track of some choice you’ve made.

In the **House Robber** problem you are trying to see what the maximum amount of gold you can steal from houses that are not next door to each

other. Here the choice is whether or not you should steal from the current house, given that you could instead have stolen from the *previous* house.


The **optimum** you are looking for is the maximum amount of gold you can steal.

### 4) Front/Back

These ones are different because instead of having both pointers traveling from the front, you have one from the front, and the other from the back. An example of this is the **Rainwater Problem** where you are calculating the **maximum** amount of rainwater you can capture in a given terrain. Again, you are looking for a maximum value, though the logic is slightly different, you are still optimizing a brute force $O(N^2)$ solution.

These four patterns should come as no surprise. After all, there are only so many ways you can move two pointers through an array or string in linear time.


Problems that use each type

### Look for “Key Insights”

One final thing to think about with these problems is the key insight that “unlocks” the problem. I talk about it bit more in my other post on how to approach algorithm problems in general. It usually involves deducing some fact based on the constraints of the problem that helps you look at it in a different way.

For example, in the **House Robber** problem, you can’t rob adjacent houses, but all houses have a positive amount of gold (meaning you can’t rob a house and have less gold after robbing it). The key insight here is that the **maximum** distance between any two houses you rob will be two. If you had three houses between robberies, you could just rob the one in the center of the three and you will be guaranteed to increase the amount of gold you steal.

For the **Bit Flip** problem, you don’t need to actually mutate the array in the problem, you just need to **keep track of how many flips you have remaining**. As you grow your window, you subtract from that number until you’ve exhausted all your flips, and then you shrink your window until you encounter a zero and gain a flip back.

### Wrapping up Our Example

So let's wrap up the **Minimum Window Substring** problem.

We established the that we need to grow our window until we have a valid substring that contains all the letters we are looking for, and then shrink that window until we no longer have a valid substring.

The key insight here is that **the smallest window will always be bounded by letters that we are searching for**. If it weren't, we could always shorten our window but lopping off unused characters at the start or end.

Some other insights to consider: there may be repeats of certain characters within our window, and that's okay. But it does hint that we need some kind of way of keeping track of the number of repeats we've seen within our window, and not just whether we've seen a character we're looking for.

This should immediately imply the use of a hash map, where the keys are the characters, and the values are the number of times we've seen a character in our window.

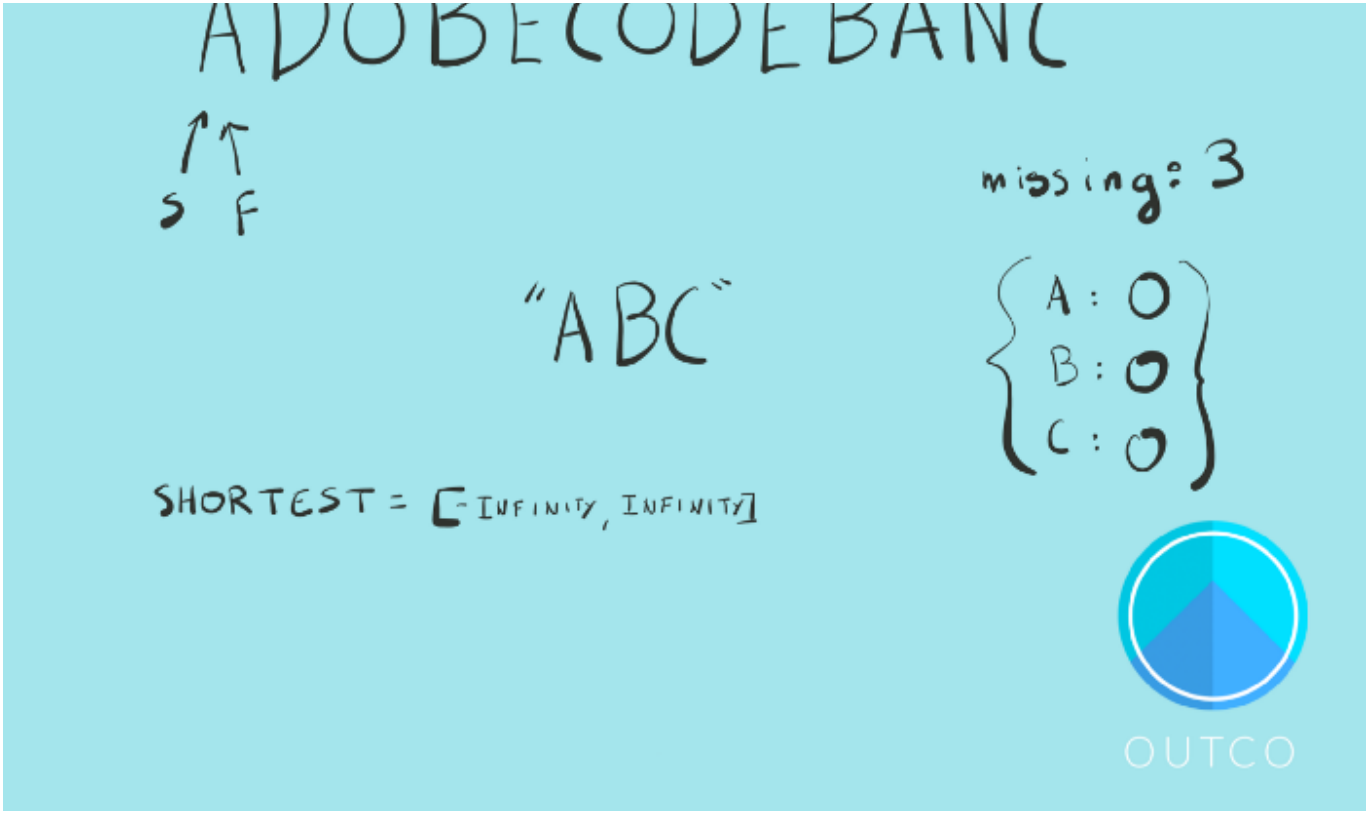
We also need an integer to keep track of how many characters we're missing to complete our set.

This would only decrement when we see a character in our window that belongs to the set, but that hasn't been seen in that particular window.

So let's summarize the algorithm:

What we need:

1) A `result` tuple (or two-element array) that represents the start and end index of the shortest substring that contains all the characters. Initialized to the largest possible range (for example, `[-Infinity, Infinity]`.
2) A `hashMap` to keep track of how letters in the set you've seen in the current window, initialized with all the characters in the set as keys, and all the values as `0`.
3) A `counter` to keep track of any time we see a new letter from the set when we grow the window, or lose a letter from the set when we shrink the window. Initialized to the number of characters we are looking for.
4) A `fast` and `slow` pointer, both initialized to `0`.



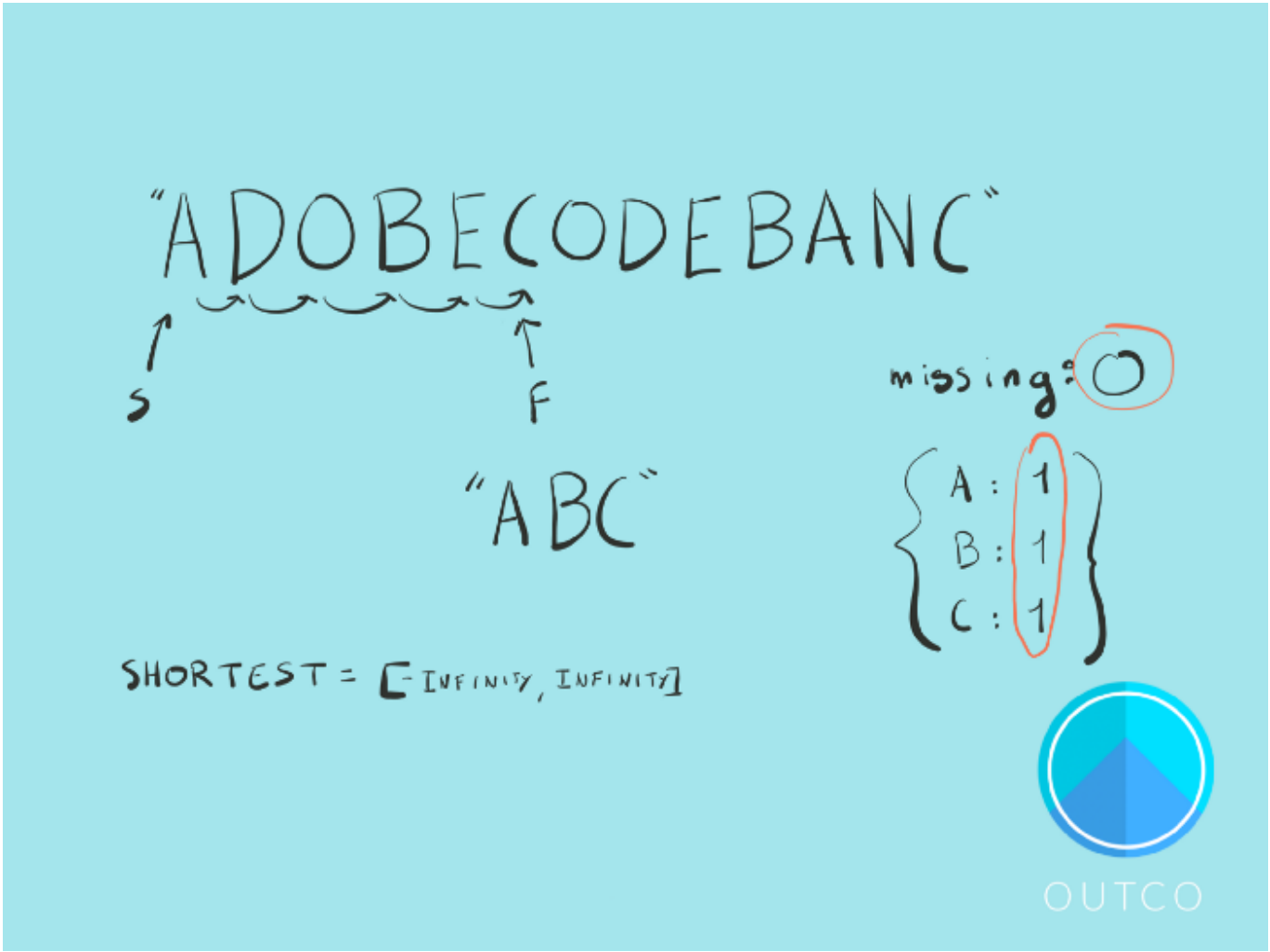

Then all we do is have a `for` loop where the `fast` pointer increments every round.

Within that `for` loop, if we see a character from the `hashMap`, we increment its value in the map.

If its value was a `0` in the hash map before, then we decrement the number of characters missing. But if we have repeats of a character we're searching

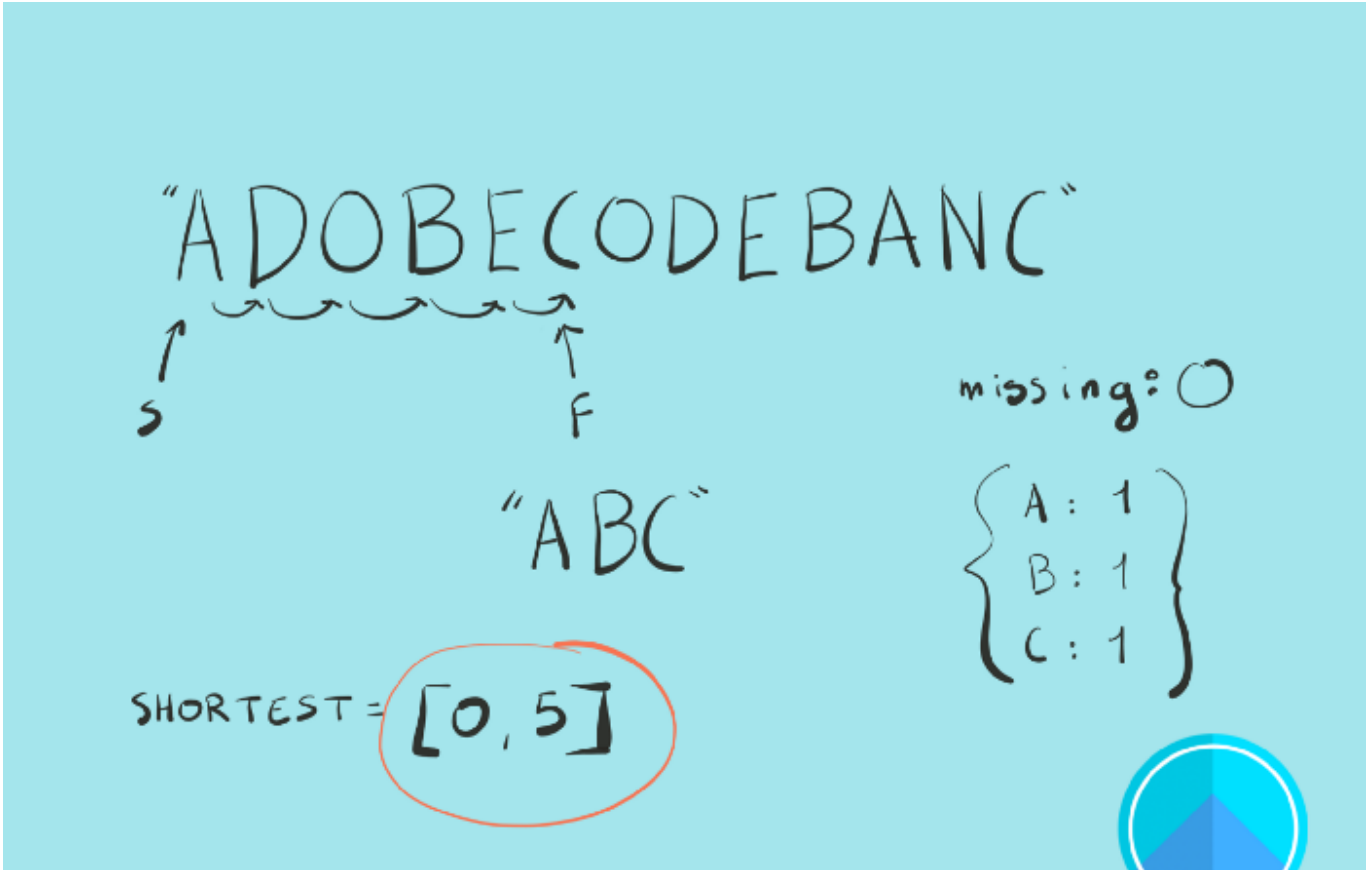
for we don't decrement the counter.

Once you've seen all the characters you're looking for, that `counter` will reach `0` and



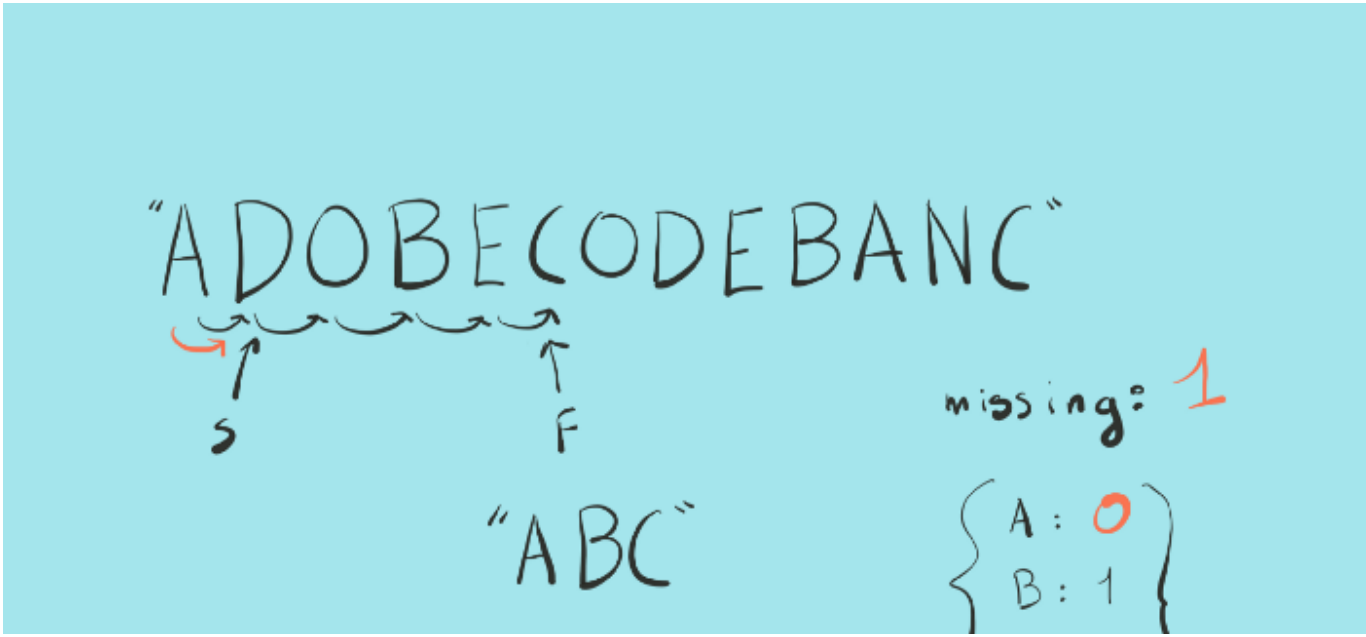
Then we have a `while` loop within the `for` loop that only runs while the number of `counter` is `0`.

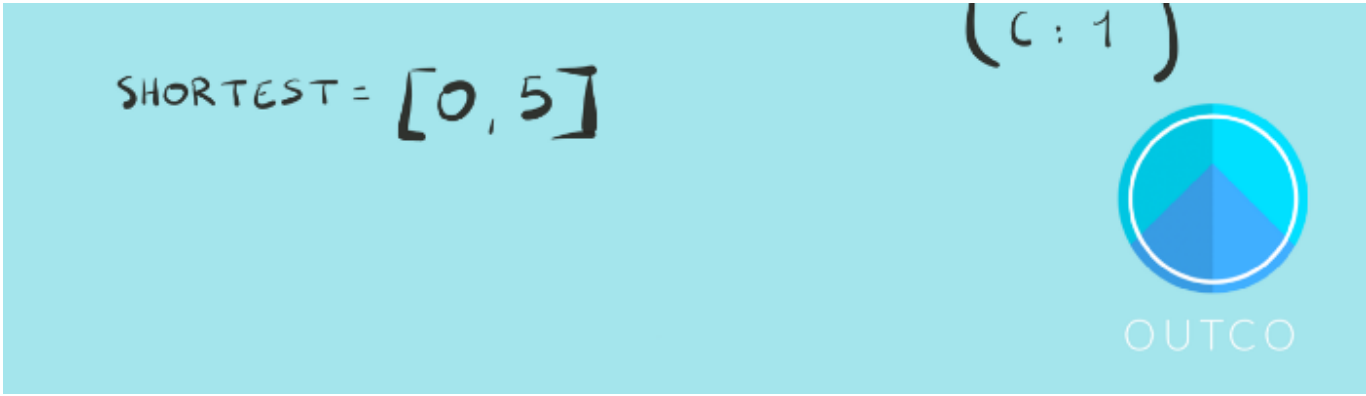
Within that `while` loop, if the difference between our `fast` and `slow` pointer is less than the difference between what is stored in our `result` tuple, then we can replace that tuple with a new smallest window. By default, the first time we find a valid window, it will update.



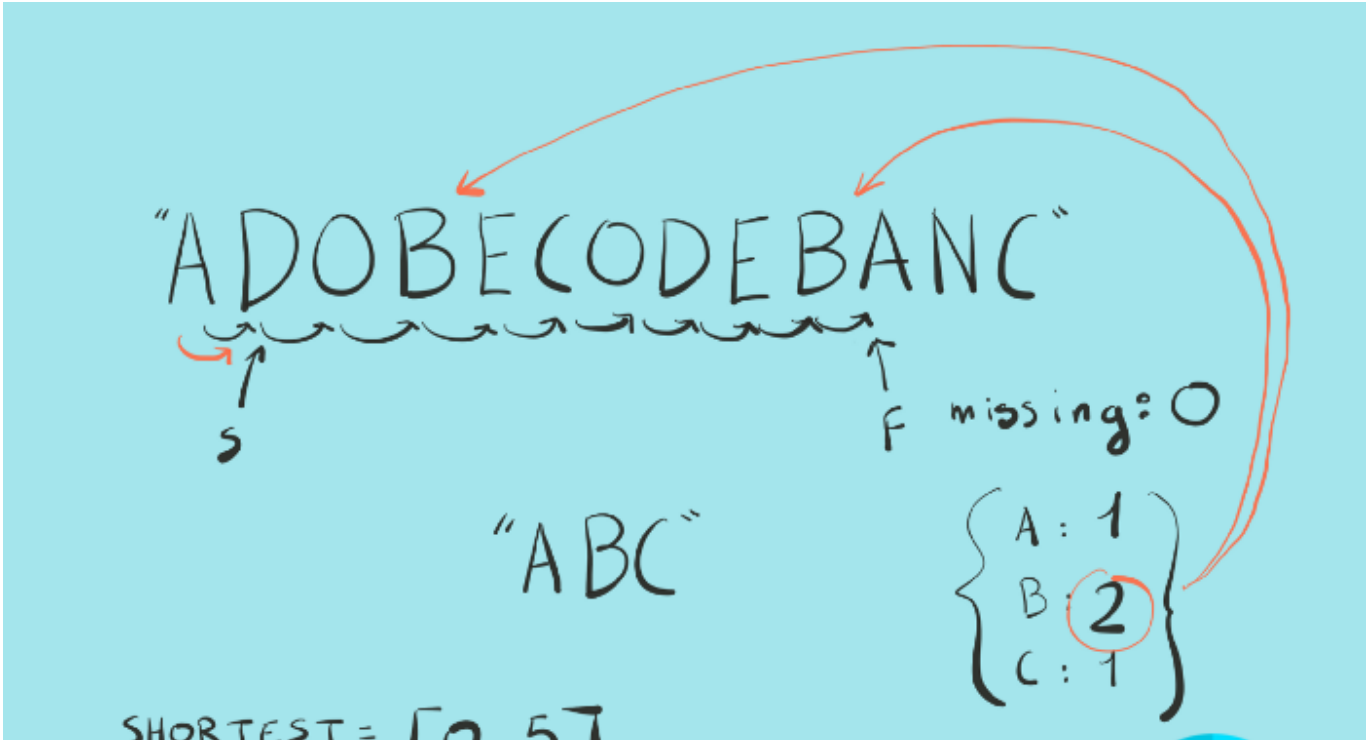

Then all we do is increment our `slow` pointer. If we see a character in our set, then we need to decrement its value in the `hashMap` by `1`, as it is moving out of our window.

If its value in the `hashMap` reaches `0`, then the number of characters we are missing now increments to `1`, and we will break out of the `while` loop next round.

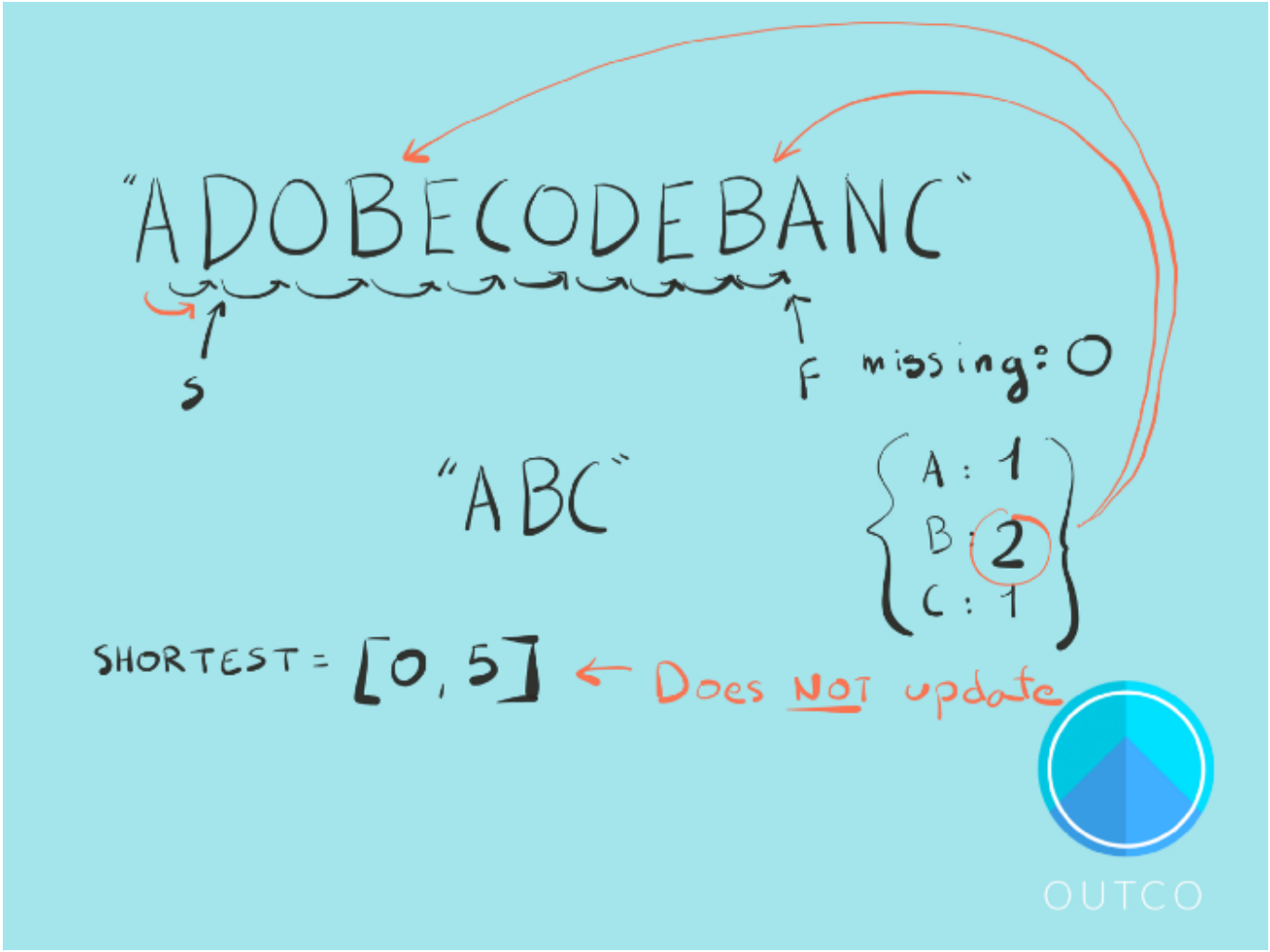


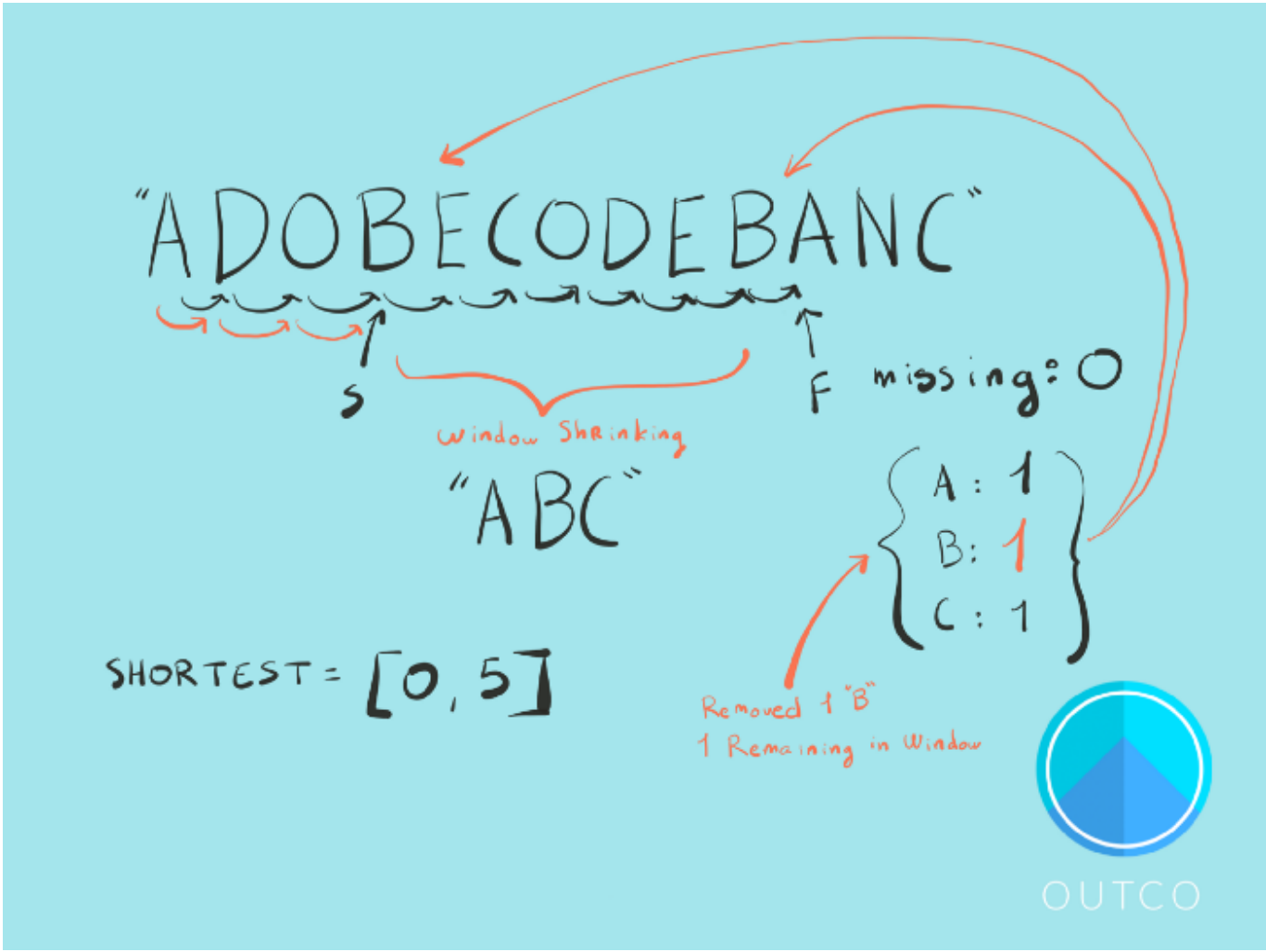



And that's the entire algorithm. Here are the slides to the next steps in our example:

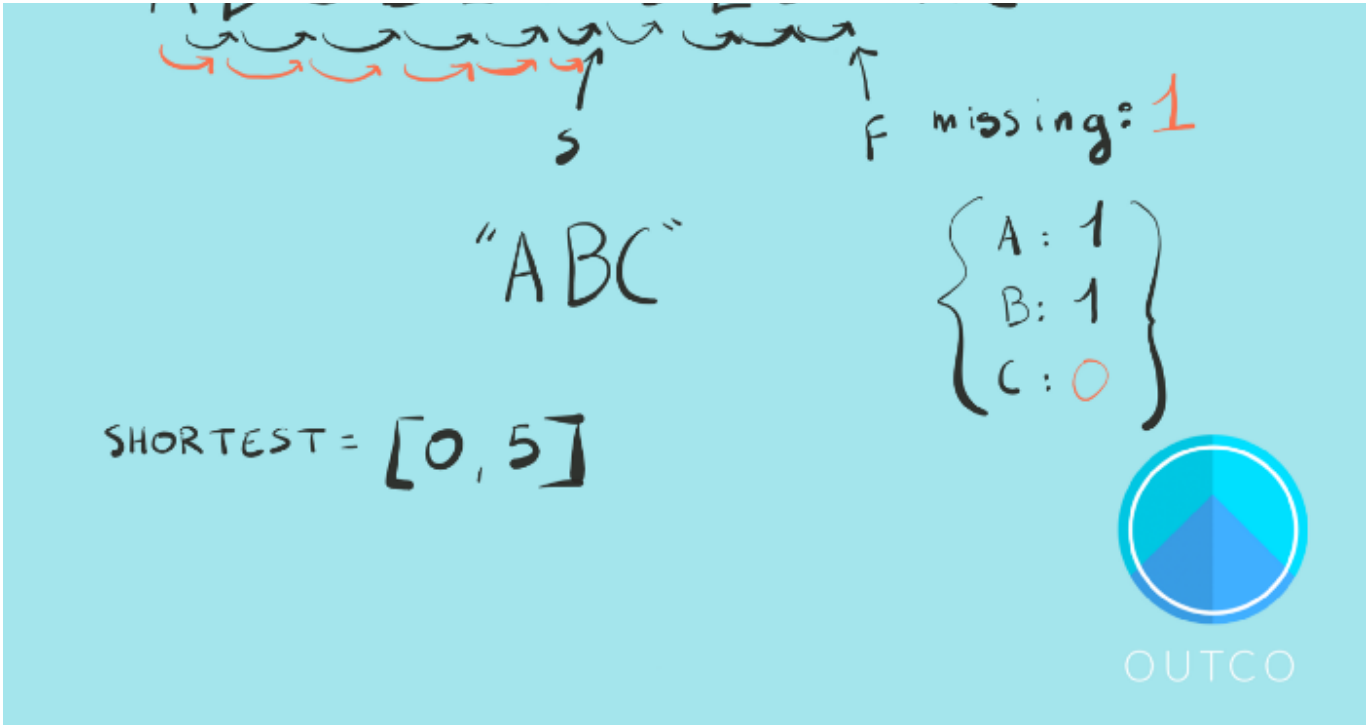


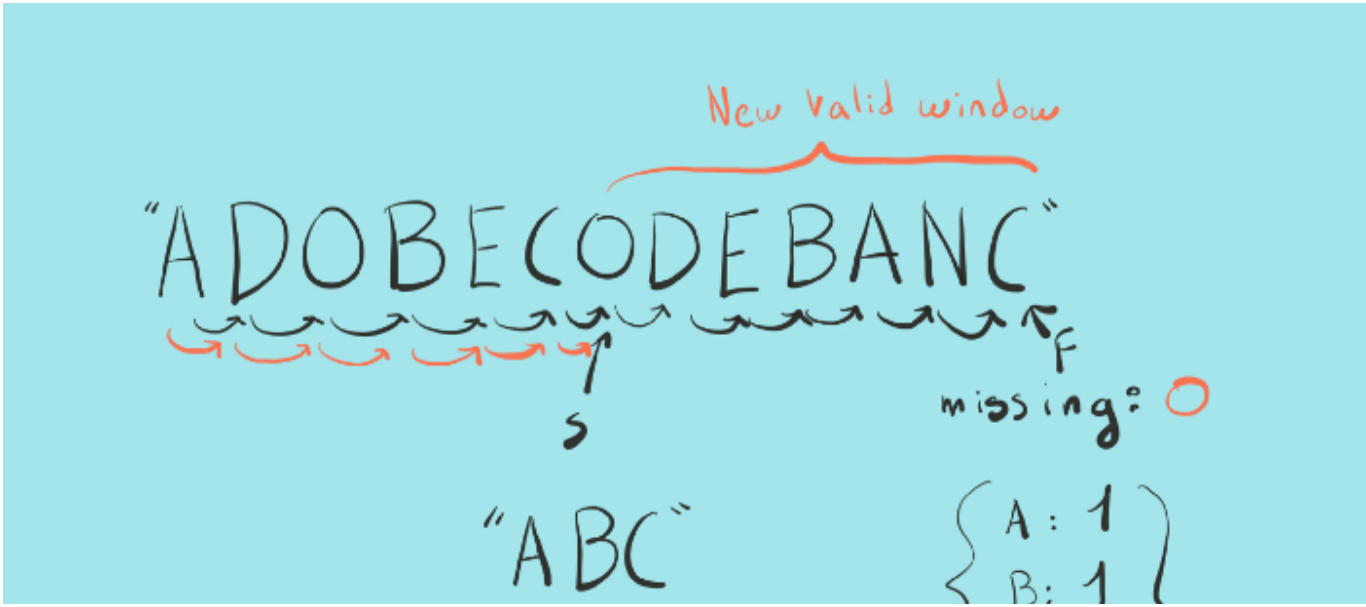

[0, 5]
OUTCO

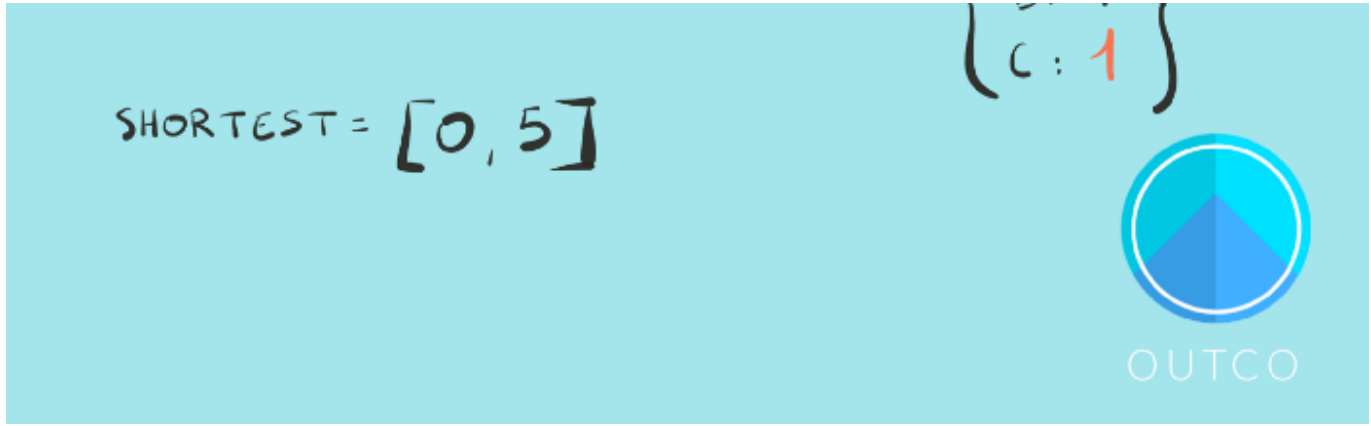

"ADOBECODEBANC"
S
F
missing: O
"ABC"
A : 1
B : 2
C : 1
SHORTEST = [0, 5] ← Does NOT update
OUTCO

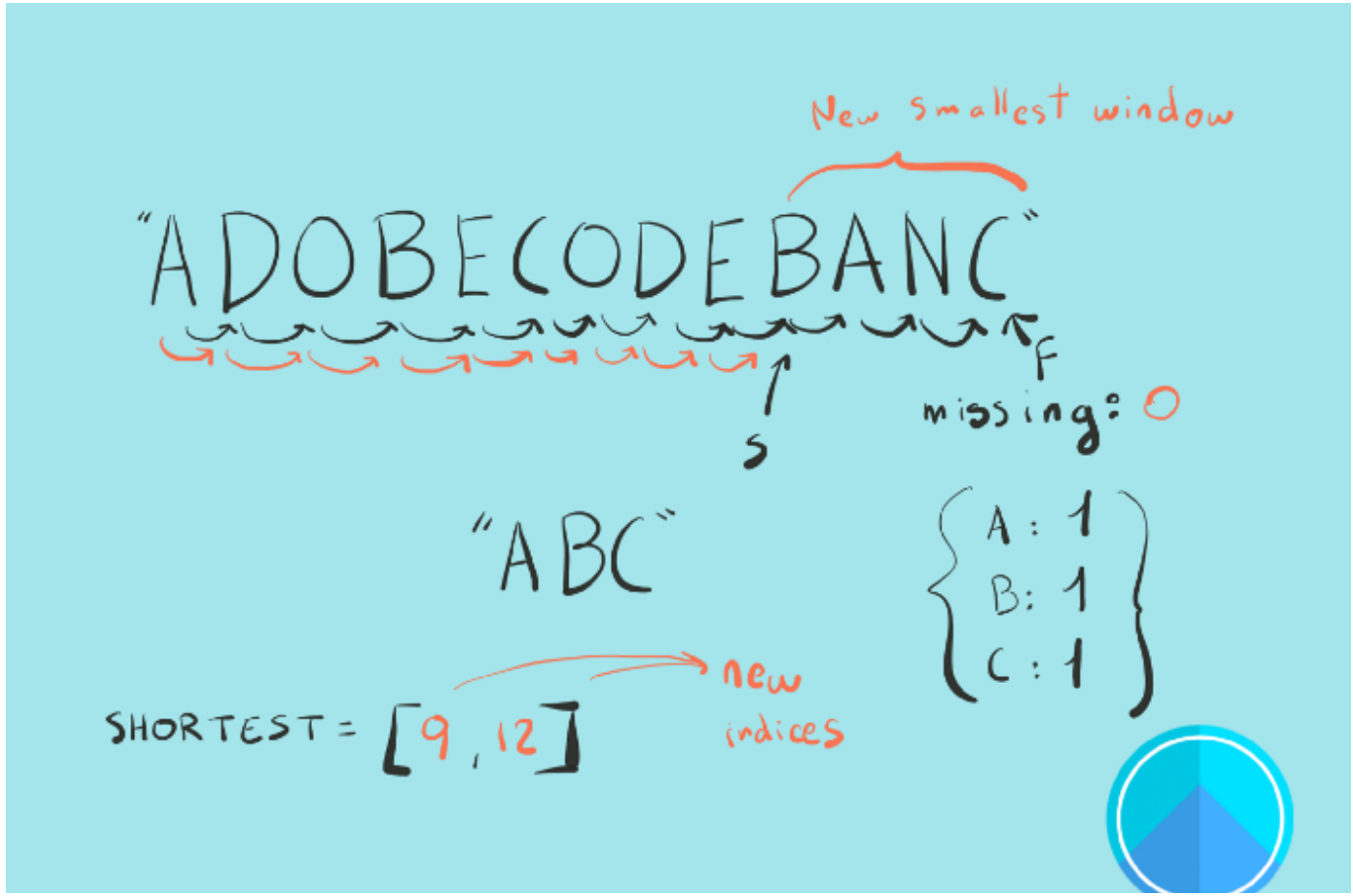

"ADOBECODEBANC"
S
F
missing: 0
window Shrinking
"ABC"
A : 1
B: 1
C : 1
SHORTEST = [0, 5]
Removed 1 "B"
1 Remaining in Window
OUTCO

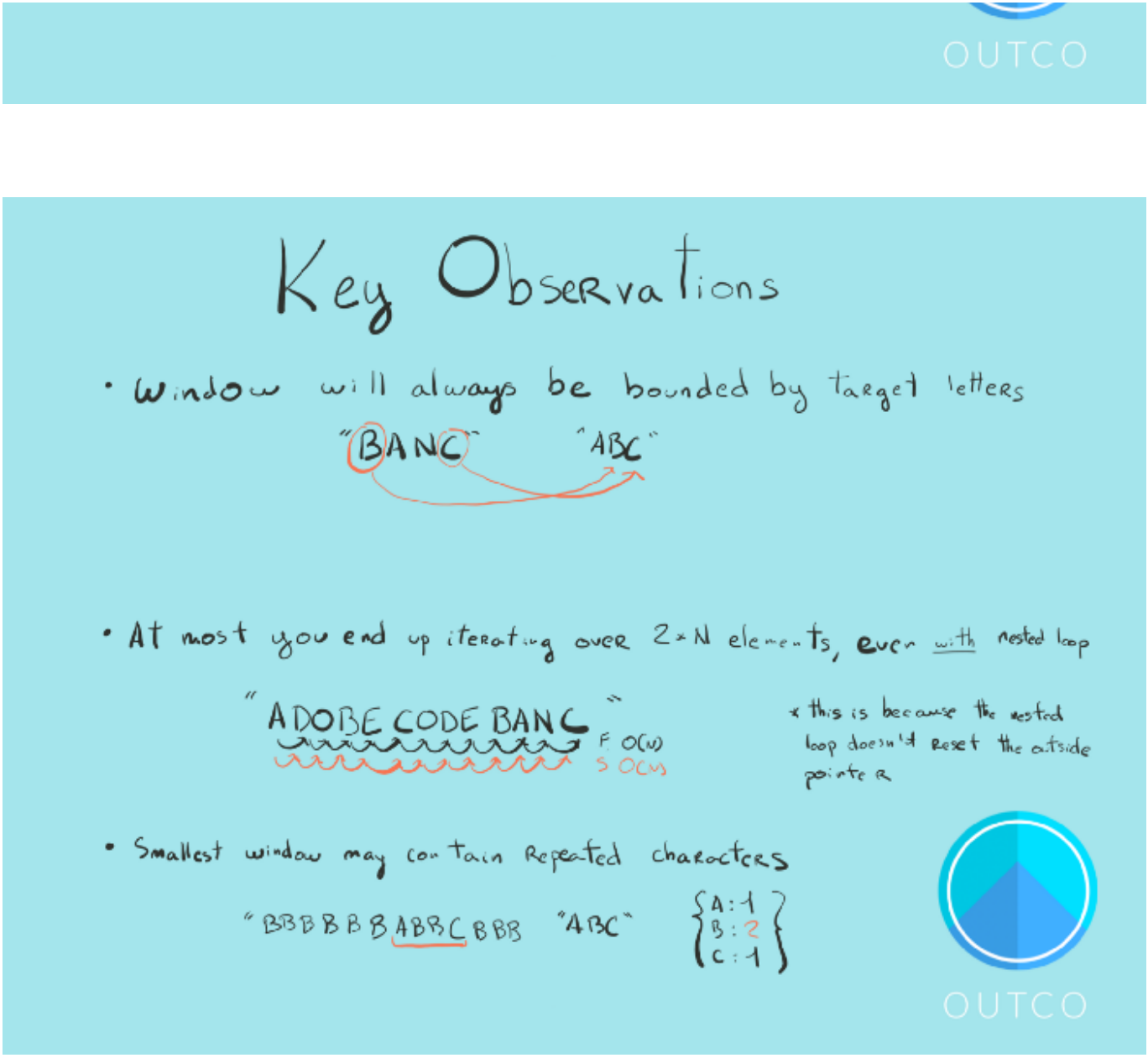

Removed from Hash Map
"ADOBECODEBANC"


S
F
missing: 1
"ABC"
A: 1
B: 1
C: 0
SHORTEST = [0, 5]
OUTCO


New valid window
"ADOBECODEBANC"
S
F
missing: 0
"ABC"
A: 1
B: 1


C : 1
SHORTEST = [0, 5]
OUTCO


New smallest window
"ADOBECODEBANC"
F
S
missing: 0
"ABC"
A : 1
B: 1
C : 1
SHORTEST = [9, 12]
new indices




And below is the code in javascript:

```
function minimumWindowSubstring(str, substr) {
  let lettersSeen = {};
  let lettersNeeded = {};
  let lettersMissing = 0;

  for(let i = 0; i < substr.length; i++) {
    if(substr[i] in lettersNeeded){
      lettersNeeded[substr[i]] += 1;
    } else {
      lettersNeeded[substr[i]] = 1;
      lettersSeen[substr[i]] = 0;
      lettersMissing += 1;
    }
  }

  let fast = 0;
  let slow = 0;

  let result = [-Infinity, Infinity];

  for(fast; fast < str.length; fast++) {

    if(str[fast] in lettersNeeded) {
      lettersSeen[str[fast]] += 1;
      if(lettersSeen[str[fast]] === lettersNeeded[str[fast]]) {
        lettersMissing -= 1;
      }
    }

    while(lettersMissing === 0) {
      if(fast - slow < result[1] - result[0]) {
        result[0] = slow
        result[1] = fast
```

```
        result[1] = fast
      }
      if(str[slow] in lettersNeeded) {
        lettersSeen[str[slow]] -= 1;
        if(lettersSeen[str[slow]] < lettersNeeded[str[slow]]) {
          lettersMissing += 1;
        }
      }

      slow++;
    }
  }

  return result[0] === -Infinity ? "" : str.slice(result[0], result[1] + 1);
}
```

MinimumWindowSubstring.js hosted with ❤ by GitHub view raw

As a challenge, try to think about how you might modify this solution to deal with repeated characters in our set. For example: `"AABCC"` . Think about when we would want to grow or shrink our window, and how we could efficiently check when those conditions are met.

Hopefully, this post helps with understanding sliding window problems. They take some time to wrap your head around, but in the end, it's a valuable pattern to know, and it's not beyond anyone with the right intuition.

A good way to solidify learning it is to explain your solution to another engineer and see if you can help them understand how it works.

If you like what you see and want to keep leveling up on Algorithms and Data Structures, check us out at Outco.io. New cohorts starting every month!

Algorithms Software Development Interview Programming Engineering