Two-pointer technique is commonly used to solve array problems very efficiently. Whenever an array question deals with finding two numbers in an array that satisfy a certain condition, either directly or indirectly, two-pointer should be the first strategy that comes to mind. This method rules out large numbers of redundant potential solutions. Although it is tricky to construct, by following certain principles, two-pointer is a simple and powerful technique to implement.

Using two-pointer is best illustrated with an example. Consider the classic two-sum problem, which asks you to find which of two elements in an array adds to another number. Although there may be many solutions, it only asks for one. In this case, let the target by 13. One solution is (9, 4).

The brute force solution would be to run through every possible pair of elements, a solution that runs in $O(n^2)$ time. However, we can use two-pointer to narrow the runtime to $O(n)$, a drastic speedup. Note that technically, the runtime has been limited to $O(n \log n)$, which is the runtime of the fastest sorting algorithms (you will understand why soon). Not all two-pointer problems require sorting, though.

First, we sort the list, for reasons you will see soon:

Next, we place one pointer at the front and another at the end of the array. Because the list is sorted, this corresponds to the smallest and largest numbers, respectively.

For each step, we calculate the sum of the two numbers being pointed to.

If the sum is less than the target, we want to increase the estimate by moving the left pointer one to the right. This brings the estimate closer to the target value. If, however, we were to move the right pointer left, our estimate would have been brought down, leading to a redundant test case.

Current estimate: 1+9=12
Target: 13

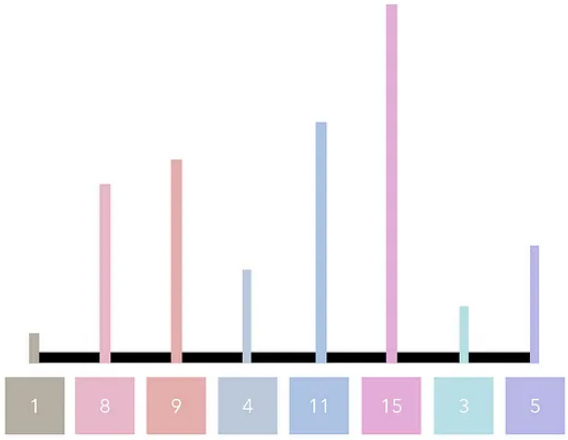
On the other hand, if our sum is larger than the estimate, we want to decrease the estimate by moving the right pointer one to the left.

If a solution is found, we can break and return the answer. On the other hand, if there is no solution, the left and right pointers will eventually point to the same number (somewhere in the middle), and all the possibilities have been explored. The reason why we can be confident in having explored all the possibilities in linear runtime lies in the fact that the list was sorted.

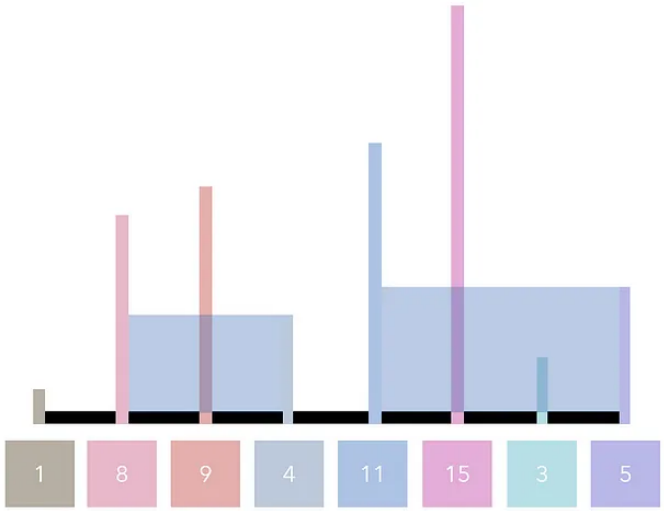
After all, if the target was 14 and we have already tested (4, 8) as false, it wouldn't make any sense to test another pair of numbers with a lower value. Hence, the two-pointer technique uses a comparison of an objective function that measures a candidate's value to rule out vast swathes of potential solutions.

However, there are many variants of the two-pointer technique. For instance, not all scenarios can be solved by setting the two pointers at opposite ends — some problems may require pointers to begin at the same spot. Others may require pointers to move at different speeds. Two-pointer is a framework to address search problems in arrays, but many of the details relies on the programmer's critical thinking and understanding of the problem.

Let us look at a similar but more complex problem. Consider the following arrangement of vertical sticks, whose lengths are written below them:

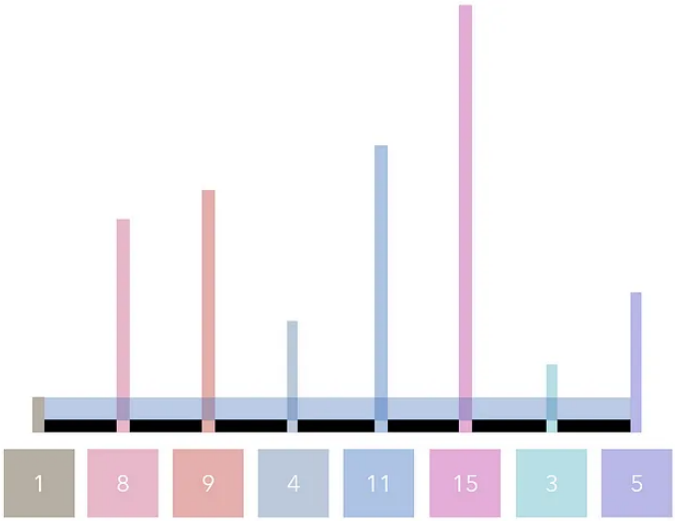


Assuming that the distance between each stick is one unit, what is the maximum amount of water one could hold between two sticks (disregarding any sticks in between)? Note that the amount of water that can be held is determined by the distance between the sticks and the height of the shorter.



This is a two-pointer problem, but we cannot sort the array because the order is important to the problem. If we sort the array by stick length, we are altering the distances between individual sticks. However, we can change our two-pointer approach a little to address this problem.

Let us begin with our two pointers on the outward elements. This maximizes the distance between the two sticks, and is a good starting point for our area. Since this problem has no target, we will keep a current maximum and repeatedly update it if we come across a larger area.

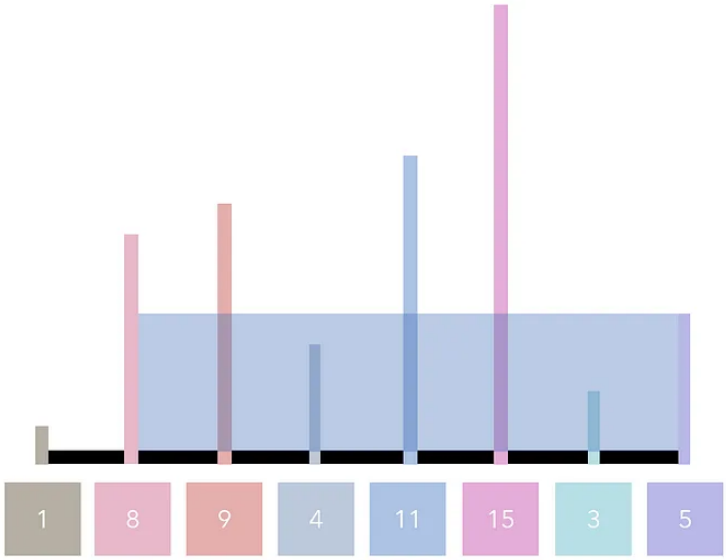


Current maximum: 7

Now that our two pointers have been initialized, we need to find how to move them. In the two-sum problem, we moved our pointers such that the estimate would move closer to the actual target value. In this scenario, we want to increase the area. What might be a good decision-making process to do this?

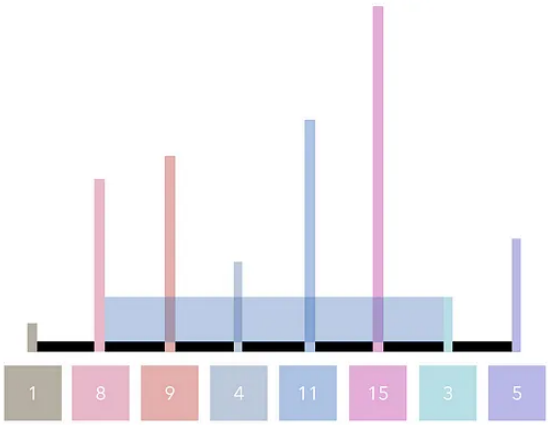
There are two potential pointers moves: move the left pointer right one, or move the right pointer left one. If we were to move the right pointer left, the area would decrease no matter how large the right pointer's value would be,

since the height is restricted by the smaller stick. Hence, our rationale for moving will be to move whichever of the pointers points to the smaller bar.



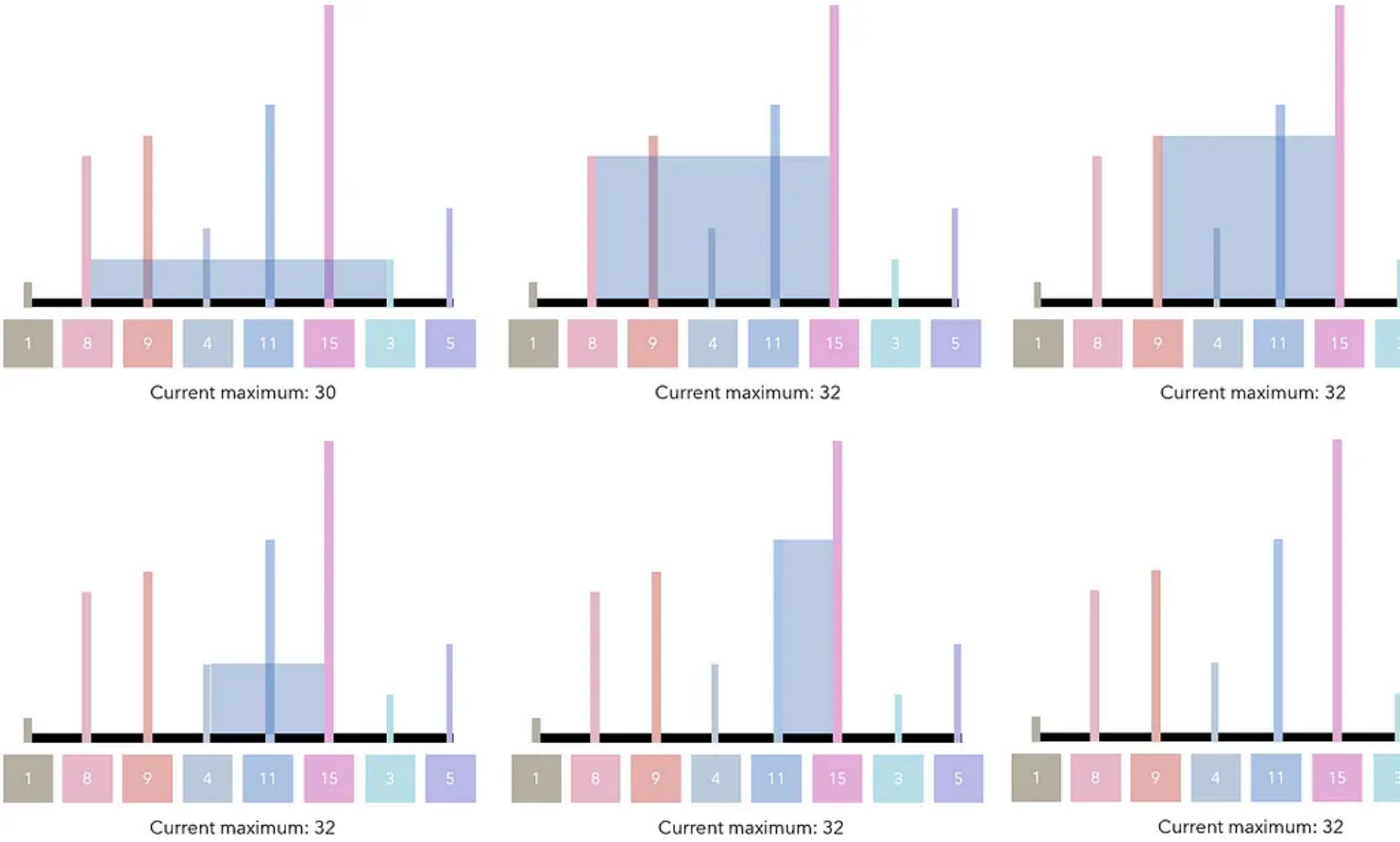
Current maximum: 30

Since the maximum area increased to 30 (a distance of six multiplied by a smaller height of 5), we set the current maximum to that value. Since the right pointer points to the smaller bar, we move it one to the left.



Current maximum: 30

If we continue to let this algorithm play out, where we move according to the rationale that the smaller bar gets moved, we find that the largest area is 32 units squared. The algorithm terminates when the left and right pointers meet, since all possibilities have hence been explored.



Already, our algorithm is much faster than what it would have been without two-pointer. However, we can make a few adjustments to make it even faster. For instance, if our two pointers are at 9 and 15, we would move 9 right one to the next stick, with value 4. This obviously has a less area, and it will be moved to the next value anyway, since it is still less than 15.

A drastic speedup to our algorithm would be to move a pointer to the next value with a greater value (in this case, jumping from 9 directly to 11 instead of stopping redundantly at 11). Like many algorithms, analysis of redundant solution considerations can lead to significant efficiency boosts. Two-pointer's value comes not only from itself but the framework it rests on, which allows the programmer to easily implement speedups.

But two-pointer is not for every problem, and trying to use it as a magical wand can lead to neglecting certain cases. In order to use two-pointer technique, one should identify or verify the following items:

- Does the problem request a search of two or more items? If this is not directly what the question is asking for, can it be reduced to such a task?
- Does there exist an initialization of two pointers such that the rest of the algorithm can function properly for *all* cases?
- Does there exist some sort of function to evaluate the goodness of the current locations of the pointers?
- Based on the goodness, does there exist a universal rationale to move the pointers to maximize the goodness?
- Could the moving of the pointers eliminate valid solutions?

- Should the array be sorted before pointers are initialized?
- Should pointers be moving at different speeds?
- Should pointers be updated one at a time or simultaneously?
- How can additional aspects of efficient computing, like creating lists to store previous pointer elements, assist pointers in further ruling out redundant solutions or to solve the problem more efficiently?
- How should relationships between three or more pointers be managed (if applicable to the problem)?
- If managing three or more pointers is not possible or very complex, can it be reduced to multiple two-pointer problems?

With these guiding principles in mind, using and designing two-pointer solutions to array searches is simple and can be extraordinarily fast.

Some additional problems to try two-pointer on your own:

- Identify palindromes within strings. For example, '`abcdedc`' has the palindrome '`cdedc`'.
- Find three numbers in an array that sum to zero. For instance, the solution for `[-3, 2, 1, 7, 9]` would be `-3`, `2`, and `1`.
- Remove duplicates from an array. Convert `[1, 1, 1, 4, 4, 5, 5, 5]` into `[1, 4, 5]`.
- Merge two sorted lists. Convert `[1, 3, 4, 6]` and `[2, 4, 7, 7]` into `[1, 2, 3, 4, 6, 7, 7]`.