Understanding the Two Pointers Technique - Part 2

Aug 17, 2024

Prerequisites: Part 1 of this post.

The goal today is to discover the linear time algorithm to 11. Container With Most Water and how it has the same theme as Two Sum. Once again, I encourage you to find a quadratic time algorithm first.

11. Container With Most Water

Summary of problem statement:

We are given a sorted array of integers $a_1, a_2, \ldots, a_n$.

For any pair of indices (i, j), where $i < j$, define the **score** of (i, j) to be

$$\min(a_i, a_j) \cdot (j - i).$$

Notice that the score is the product of the height and width of the rectangle created by i and j: the height is $\min(a_i, a_j)$, and the width is $j - i$[1].

Goal: Determine the maximum score over all pairs (i, j).

First attempt (Time Limit Exceeded):

Just as in the Two Sum solution, trying all possible pairs of indices (i, j) takes $O(n^2)$ time. This time though, we store the `best` result we've seen so far.

```
int maxArea(vector<int>& a) {
        const int n = a.size();
        int best = 0;
        for (int i = 0; i < n; i++) {
                for (int j = i+1; j < n; j++) {
                        best = max(best, min(a[i], a[j]) * (j-i));
                }
        }
        return best;
}
```

Making Observations:

Once again, there are too many pairs to check.[2] We'll have to figure out how to rule out most pairs without checking them.

Consider the array $a = [1, 8, 6, 2, 5, 4, 8, 3, 5]$, which is modified from LeetCode's example. Let's look for patterns:

(i, j)	2	3	4	5	6	7	8	9
1	$1 \cdot 1 = 1$	$1 \cdot 2 = 2$	$1 \cdot 3 = 3$	$1 \cdot 4 = 4$	$1 \cdot 5 = 5$	$1 \cdot 6 = 6$	$1 \cdot 7 = 7$	$1 \cdot 8 = 8$
2		$6 \cdot 1 = 6$	$2 \cdot 2 = 4$	$5 \cdot 3 = 15$	$4 \cdot 4 = 16$	$8 \cdot 5 = 40$	$3 \cdot 6 = 18$	$5 \cdot 7 = 35$
3			$2 \cdot 1 = 2$	$5 \cdot 2 = 10$	$4 \cdot 3 = 12$	$6 \cdot 4 = 24$	$3 \cdot 5 = 15$	$5 \cdot 6 = 30$
4				$2 \cdot 1 = 2$	$2 \cdot 2 = 4$	$2 \cdot 3 = 6$	$2 \cdot 4 = 8$	$2 \cdot 5 = 10$
5					$4 \cdot 1 = 4$	$5 \cdot 2 = 10$	$3 \cdot 3 = 9$	$5 \cdot 4 = 20$
6						$4 \cdot 1 = 4$	$3 \cdot 2 = 6$	$4 \cdot 3 = 12$
7							$3 \cdot 1 = 3$	$5 \cdot 2 = 10$
8								$3 \cdot 1 = 3$

I've written $(\text{min}) \cdot (\text{length}) = \text{score}$ in each cell. The pattern for length is clear, but it seems that multiplying by min causes complications.

Still, there is some regularity:

- Consider the first row ($i = 1$). Since a_1 is the smallest value in the array, $\min(a_1, a_j)$ is constant for this entire row. That means the maximum score for this row must be at $(1, 9)$.
- Consider the second row ($i = 2$). Since a_2 is very large, $\min(a_2, a_j)$ is **not** constant on this row – it varies with a_j. It's hard to know the maximum score for this row without checking a lot of values.[3]
- Instead, consider the ninth column ($j = 9$). $\min(a_i, a_9)$ fluctuates, but we know that a_2 is very large, in the sense that $a_2 \geq a_9$. This shows that $(3, 9)$, $(4, 9)$, $(5, 9)$,$(6, 9)$, $(7, 9)$, and $(8, 9)$ must have smaller score than $(2, 9)$. Can you see why?

> **Claim**
>
> Consider any two indices l and r, where $l < r$.
>
> 1. If $a_l \leq a_r$, then the score of (l, r) is strictly greater than the score of (l, j) for all $l \leq j < r$.
> 2. If $a_l \geq a_r$, then the score of (l, r) is strictly greater than the score of (i, r) for all $l < i \leq r$.

Proof of part 1: Suppose $a_l \leq a_r$, and consider any index j where $l \leq j < r$.

The score of (l, j) is $\min(a_l, a_j) \cdot (j - l)$, and the score of (l, r) is $\min(a_l, a_r) \cdot (r - l)$.

Comparing each term, we get:

- $\min(a_l, a_j)$ is at most a_l, but $\min(a_l, a_r)$ is exactly a_l (because $a_l \leq a_r$).
- $j - l$ is strictly smaller than $r - l$, because $j < r$.
  These two points combine to show that the score of (l, j) is less than the score of (l, r), which proves part 1 of the claim.

A good idea is to draw a diagram (or use the one on LeetCode) to follow along. It's easier to visually see how the lengths and heights of the rectangles formed by (l, j) and (l, r) relate.

Exercise: Prove or convince yourself that part 2 of the claim also works, i.e., nothing changes when the picture is flipped.

Using the claim:

ⓘ Algorithm for finding the maximum score.

Let b be the best score found over all pairs checked so far. Also, let $l = 1$ and $r = n$.

The values l and r represent the range we still have to check. So, we are initially looking for the best score over all pairs (i, j) where i and j are in the range $[1, n]$.

As long as there are still pairs to check, repeat the following steps:

1. If the score of (l, r) is greater than b, then update b.
2. If $a_l \leq a_r$, then update the range to $[l + 1, r]$.
3. If $a_l > a_r$, then update the range to $[l, r - 1]$.

Finally, report b as the maximum score over all pairs.

Steps 2 and 3 use our claim to reduce the number of pairs we had to check.

The idea is that if $a_l \leq a_r$, checking (l, r) makes it useless to check (l, j) for any other value of j. **Thus, we're completely done with** l, so we can increment it.

Similarly, if $a_l > a_r$, checking (l, r) makes it useless to check (i, r) for any other i. That means we're done with r and can decrement it.

There are $O(n)$ pairs to check, since the number of values in the range $[l, r]$ decreases by 1 after each iteration. Therefore, our algorithm runs in $O(n)$ time.

Exercise: If $a_l = a_r$, can we instead update the range from $[l, r]$ to $[l + 1, r - 1]$?

Code (Accepted):

```
int maxArea(vector<int>& a) {
        const int n = a.size();
        int best = 0;
        for (int l = 0, r = n-1; l < r; ) {
                best = max(best, min(a[l], a[r]) * (r-l));
                if (a[l] <= a[r]) l++;
                else r--;
        }
        return best;
}
```

Once again, the words "two pointers" are just an implementation detail. The key theme, which is shared with Two Sum, is that we can quickly reduce the problem of checking pairs in a range $[l, r]$ to some smaller range, either $[l+1, r]$ or $[l, r-1]$. In fact, you could draw a table of `?`s for this problem, and see that the process of checking and skipping pairs is almost identical to that of Two Sum[4].

Aside: Why I don't like NeetCode's explanation

Firstly, to his credit – the statement that

> The minimum height is our limiting factor

is true, and it is one of the key observations used in our claim.

However, I take issue with the other intuition he provides.

> (6:42) How are we going to update our pointer? Well, we're going to look at what's the minimum height. [...] Why would I shift my right pointer, [which] has a height of 7, when I could instead shift my left pointer, which has a height of 1, **and potentially increase it?**

Given pointers l and r, it seems that NeetCode is saying the goal is to try to update the pointers to maximize a_l and a_r. **This is completely wrong**. If this really was the goal, why are we even using the two pointers technique – can't we directly find the two largest values in the array?

Notice that if we increase l by 1, the algorithm will never check another pair (l, i) with the old value of l again, for any value of i. So, the only way we can justify increasing l is if we know that **all other pairs using l are not optimal.**

This is where we used our claim. NeetCode makes no mention of this, but he obtains the correct algorithm for reasons that are unrelated to the intuition he provides. There are also other parts of the video where I disagree with his intuition (e.g., why $l = 1$ and $r = n$ initially, and which pointer to change when $a_l = a_r$), but the above quote is my main issue.

Aside - An example where being greedy fails

Consider 416. Partition Equal Subset Sum. Here's an algorithm that shows choosing the best next step from our current point of view, called a **greedy approach**, rather than taking a global view, can be wrong.

> ⓘ **Bogus Solution.**
>
> Start with two empty sets S_1 and S_2.
> Check each element a_i in the original array, in order, and do the following:
>
> 1. If the sum of elements in S_1 is less than the sum of elements in S_2, add a_i to S_1.
> 2. Otherwise, add a_i to S_2.
>
> If the final sum of elements in S_1 and S_2 are equal, return `true`. Otherwise, return `false`.

The idea is that we are minimizing the difference between the sums of elements in S_1 and S_2 at each step. But, there's no guarantee that putting a_i in the other set wouldn't lead to a valid partition, and LeetCode's first example $[1, 5, 11, 5]$ serves as a concrete example of this (what should you do with a_2?).

In this problem, it's much harder to rule out any particular combination of elements. The simplest solution is pretty much to **try all possible partitions**, with a few optimizations involving what to do when two subsets give you the same sum. The details are more advanced and outside the scope of this post.

---

1. We could have also ignored the restriction $i < j$ and defined the score as $\min(a_i, a_j) \cdot |j - i|$. ↩
2. This time, LeetCode's constraint of $n \leq 10^5$ is more reasonable, since in the max case the number of pairs to check is $\frac{n(n-1)}{2} \approx 5 \cdot 10^{10}$. A rough estimate is that your code can run about 10^6 to 10^9 instructions per second, which is too slow for a 2-second time limit. ↩
3. I changed a_9 from 7 to 5 because $(2, 9)$ happened to still be the maximum value. You might say this is cheating – how did I know to do this, instead of assuming the incorrect claim that the maximum score always occurs at one of the endpoints? The answer is a bit unsatisfying: my intuition told me this claim

was fishy because a_9 could be any value. In general, you should be suspicious of unproven claims or patterns. In this case, you'd likely to find a counterexample after trying a few more arrays. ↩

4. This idea can be generalized to sliding window problems such as 121. Best Time to Buy and Sell Stock. The pairs you check are different, but you still use the same idea of skipping most pairs by proving that they can't be optimal. ↩