8 The maximum-sum subarray

Given an array of integers, find the contiguous subarray having the largest sum. Return its sum.

Example: for `[-1, 1, 2, 3, -2]`, the sum is 6 for the subarray `[1, 2, 3]`.

Clarification questions

Q: What result should be returned for an empty array?
A: The result is 0.

Q: What result should be returned when all elements are negative?
A: The result is the largest negative element, corresponding to a subarray of length 1 containing that element.

Q: What is the maximum length of the input?
A: 1,000,000 elements.

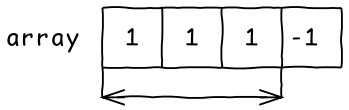
We asked the clarification questions to learn how to handle the edge cases, as well as find out what are the time complexity constraints for the solution. To handle 1 million elements, we must look for an algorithm with time complexity below $O(n^2)$, such as $O(n)$ or $O(n \log n)$.

Solution 1: dynamic programming, $O(n)$ time, $O(n)$ space

The problem involves a search combined with an optimization (maximizing the sub of the subarray), which points to the possibility of a dynamic programming solution. In order to propose such a solution, we need to formulate the problem in terms of smaller subproblems; or the other way around: starting from a smaller subproblem, find a way to extend it to arrive to the original problem.

To find out if this is possible, we first need to understand the properties of the maximum sum subarray. In particular, we are looking for ways of either growing it element by element; or of building it from smaller chunks. Let's analyze some simple examples to find out how this could work.

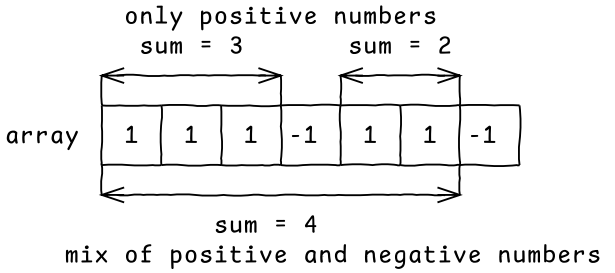
Firstly, it is clear that when a positive element is part of the maximum-sum subarray, all the positive elements around it should also be part of the maximum-sum subarray. Consider the following example:



All of the first three elements form the maximum-sum subarray, with sum 3. It does not make sense to take fewer (such as the first two), since the sum would be suboptimal.

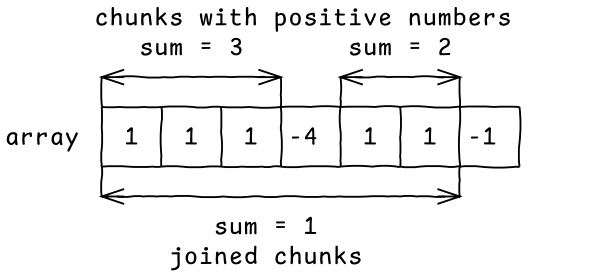
Similarly, it does not make sense to expand to neighboring negative numbers: taking also the fourth element would decrease the sum from 3 to only 2, which is suboptimal.

Does it ever make sense to have negative numbers as part of the solution? Yes, it does, as we can see from the following example:



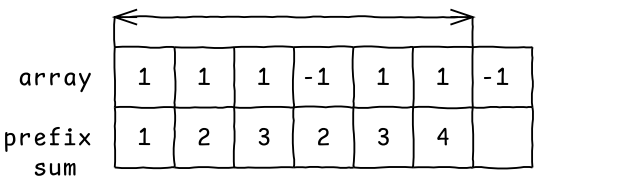
If we use only positive numbers, the best subarray we can find has sum 3. However, if we allow using the -1 element from the middle, we can pay a small penalty to join the two chunks, to obtain a better sum of 4.

Does it always make sense to pay the penalty to join chunks containing positive numbers? Not when the penalty exceeds the benefit of joining the chunks, as we can see from the following example:



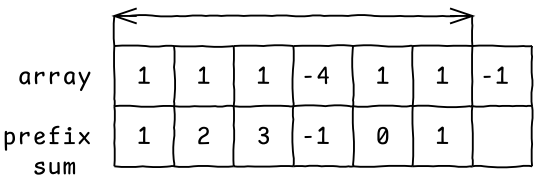
Here the negative number is smaller (-4 instead of -1 from before), so joining the chunks does not pay off.

Let's see if we can take advantage of this property when growing the solution element by element. We consider again the example where it makes sense to join the two chunks, and show how the sum of the subarray evolves as we add new elements:



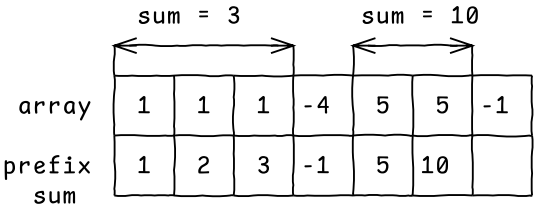
The prefix sum grows as we add elements from the first chunk, then it decreases slightly when we add the -1 in the middle, after which it increases again as we add elements from the second chunk. So far everything works as expected.

Let's see what is different when the negative element in the middle is heavier:

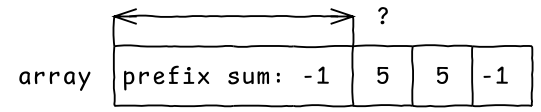


Here we see that the prefix sum goes below zero when adding the -4, after which it grows again but to less than it was before. In other words, when we reach the chunk on the right of -4, we have a prefix with negative sum. It does not make sense to keep it: dropping it and startig a new chunk instead is preferrable.

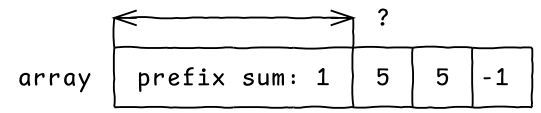
Let's change the example so that the chunk on the right becomes the final solution, to demonstrate the reasoning behind starting a new chunk when the prefix sum is negative:



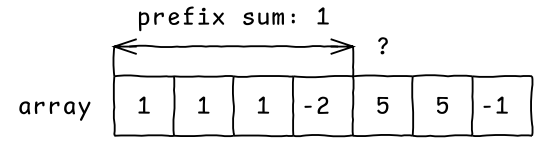
Indeed we see that the solution is formed by the chunk on the right with sum 10. Had we included the chunk on the left as well, the sum would have been only 9. We can say that the prefix sum abstracts away the details of the current growing chunk, keeping only what matters: the sum of the numbers in the chunk:



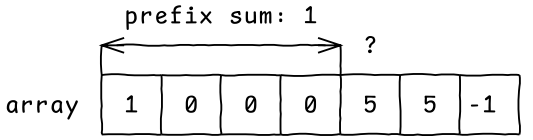
Here is another example where the sum of the prefix is positive:



It does not matter what actual elements were part of the prefix. It could have been the following:



Or it could have been the following:

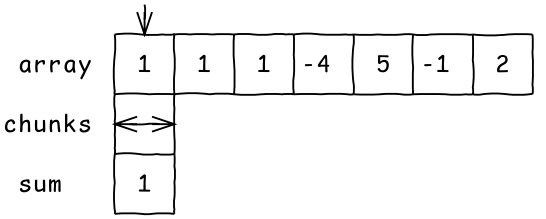


In either case, the outcome is the same: we append 5 to a prefix having sum 1, to obtain a subarray with sum 6.

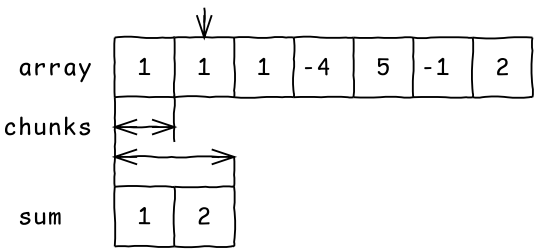
To summarize what we have learned so far:

- To find the maximum-sum subarray, we can examine smaller chunks of the array;
- A bigger chunk can be formed by appending an item to the chunk preceding it; this should be done only when the sum of the preceding chunk is positive;
- When the sum of the preceding chunk is negative, we should start a new chunk containing a single element;
- The maximum-sum subarray is the chunk with the largest sum.

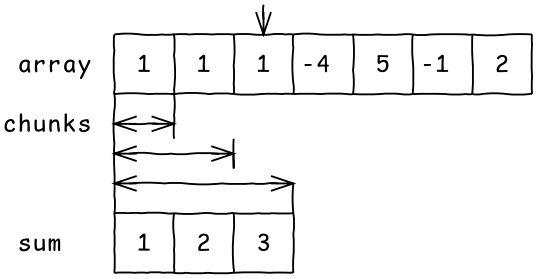
Let's see a full search running step by step, applying these rules:



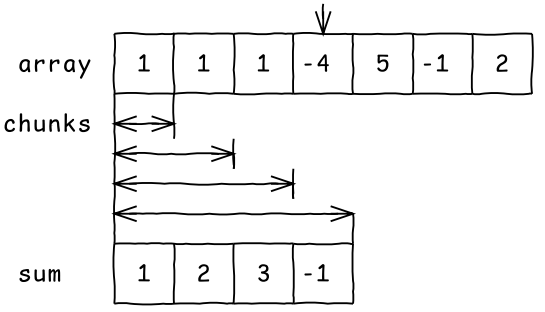
We start with a chunk containing the first element, having sum 1.



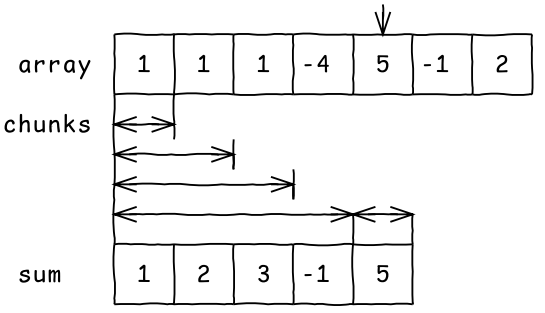
Then we examine the second element. Since the previous chunk has positive sum, we grow it to form a new chunk.



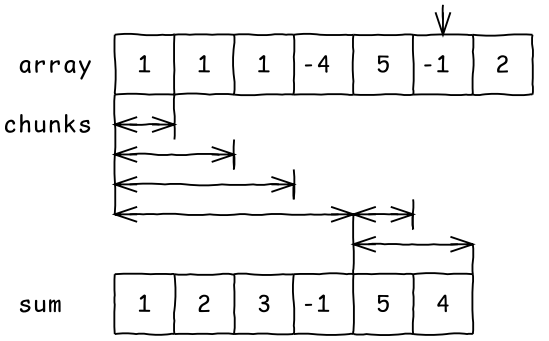
The same for the third element: we grow the chunk again.



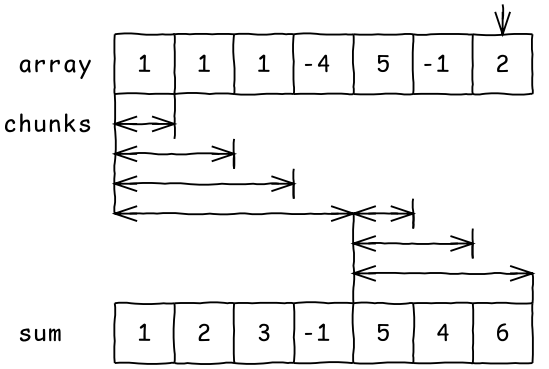
We add element -4 to form a chunk with sum -1. We record it even if we will discard it at the next step.



We examine element 5, and we discard the preceding chunk since it has negative sum. We start a new chunk containing only 5.



We reach element -1, and we create new chunk with sum 4.



We reach element 2, and we grow the preceding chunk to form a new chunk with sum 6.

We finished examining all the elements of the array. We now simply choose the chunk with the largest sum, which is 6.

This algorithm can be implemented as:

```
def find_max_sum_subarray(array):
    # chunks[i] = sum of the optimal chunk ending at array[i].
    chunks = []
    for item in array:
        if not chunks:
            # Start a new chunk containing the first element.
            chunks.append(item)
        else:
            if chunks[-1] < 0:
                # Start a new chunk.
                chunk = item
            else:
                # Grow the previous chunk.
                chunk = chunks[-1] + item
            # Record the chunk.
```

```
            chunks.append(chunk)
    # Choose the best chunk.
    return max(chunks) if chunks else 0
```

The running time is linear; the space is linear as well, due to having to store the `chunks` array.

Solution 2: dynamic programming, $O(n)$ time, $O(1)$ space

We can improve the previous algorithm by reducing its memory requirements. Notice that at each step, only the last element of the array `chunks` is examined, representing the previous chunk. The only reason to keep the whole array is to compute `max(chunks)` at the end of the search. We can avoid this by computing a running maximum instead, by keeping track of the best chunk ever seen. Then the `chunks` array can be replaced with a variable storing the sum of the last chunk, `previous_chunk`.

This is called Kadane's algorithm, and can be implemented as follows.

```
def find_max_sum_subarray(array):
    # Handle the edge case here to simplify the code handling normal cases.
    if not array:
        return 0
    previous_chunk = -float('inf')
    best_chunk = -float('inf')
    for item in array:
        if previous_chunk < 0:
            # Start a new chunk.
            chunk = item
        else:
            # Grow the previous chunk.
            chunk = previous_chunk + item
        # Update the running maximum.
        best_chunk = max(best_chunk, chunk)
        # Record the chunk.
        previous_chunk = chunk
    return best_chunk
```

Unit tests

For testing, we would first like to cover simple cases to make sure the algorithm works at all. We choose short arrays, so that troubleshooting is easy in case of a bug.

We must also cover the edge cases. We think about not just possible weak points of our implementation, but also cases that could be tricky to handle in general by other implementations, so that we catch regressions in case the code is changed in the future. Some ideas that come to mind are:

- the empty array;

- arrays that contain only negative numbers;
- arrays that contain only positive numbers;
- arrays that contain only zeroes;
- arrays that contain multiple subarrays with positive sum.

Finally, we should have performance tests with large inputs, to benchmark the implementation as well as to catch performance regressions.

The unit tests can be written as:

```
class TestMaxSumSubarray(unittest.TestCase):
    def test_1_simple(self):
        self.assertEqual(find_max_sum_subarray([-1, 1, 2, 3, -2]), 6)

    def test_2_negative_elements(self):
        self.assertEqual(find_max_sum_subarray([-1, 1, 2, 3, -2, 3]), 7)

    def test_3_local_optimum(self):
        self.assertEqual(find_max_sum_subarray([-1, 1, 2, -9, 4, -1]), 4)
        self.assertEqual(find_max_sum_subarray([-1, 4, -9, 1, 2, -1]), 4)

    def test_4_all_positive(self):
        self.assertEqual(find_max_sum_subarray([1] * 10), 10)

    def test_5_all_negative(self):
        self.assertEqual(find_max_sum_subarray([-1] * 10), -1)

    def test_6_empty_input(self):
        self.assertEqual(find_max_sum_subarray([]), 0)

    def test_7_perf(self):
        self.assertEqual(find_max_sum_subarray([1] * 1000000), 1000000)
```

How much time to spend on unit tests

During an interview you are usually not expected to write a full implementation of the unit tests, unless you are applying for a testing position. However, it is common to be asked for a verbal description of the tests, along with a justification of your reasoning. If on top of that you are able to show that you can write correctly a sample test, or at least mention typical tools you would use such as a unit testing framework or the assertion predicates, this may put you slightly ahead of other candidates.