Rohit Singhal

Follow 401 Followers About

Kadane's Algorithm — (Dynamic Programming) — How and Why does it Work?

Rohit Singhal Dec 31, 2018 · 6 min read

If you are here, then chances are that you were trying to solve the "Maximum Subarray Problem" and came across Kadane's Algorithm but couldn't figure out how something like that is working. Or maybe you were tired of using Kadane's Algorithm as a "black-box". Or maybe you wanted to understand the dynamic programming aspect of it. Or maybe you just want to learn about a new concept which can make you better at programming. Whatever the reason, you've come to the right place.

To better understand Kadane's Algorithm, first, we would go through a short introduction of Dynamic Programming. Then, we would look at a quite popular programming problem, the Maximum Subarray Problem. We would see how this problem can be solved using a brute force approach and then we would try to improve our approach and come up with a better algorithm, aka, Kadane's Algorithm.

So, let's get into it.

Dynamic Programming

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure (array, map, etc.). So the next time the same sub-problem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time.

> *Those who cannot remember the past are condemned to repeat it. — Dynamic Programming*

Here's a brilliant explanation on the concept of Dynamic Programming on Quora — Jonathan Paulson's answer to How should I explain dynamic programming to a 4-year-old?

Though there's more to dynamic programming, we would move forward to understand the Maximum Subarray Problem.

Maximum Subarray Problem

The **maximum subarray problem** is the task of finding the largest possible sum of a contiguous subarray, within a given one-dimensional array A[1…n] of numbers.

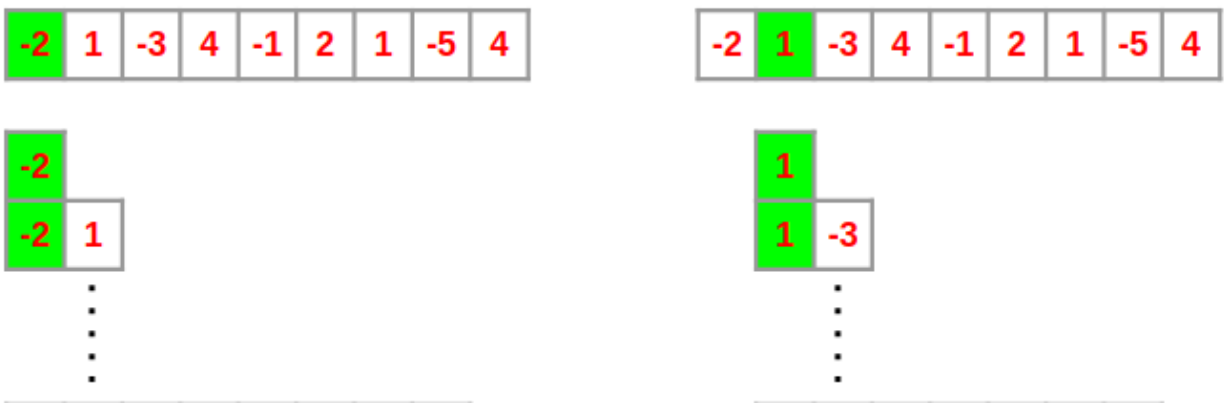
Maximum Sum Subarray (In Yellow)

For example, for the array given above, the contiguous subarray with the largest sum is [4, -1, 2, 1], with sum 6. We would use this array as our example for the rest of this article. Also, we would assume this array to be zero-indexed, *i.e.* -2 would be called as the '0th' element of the array and so on. Also, ***A[i]*** would represent the value at index ***i***.

Now, we would have a look at a very obvious solution to the given problem.

Brute Force Approach

One very obvious but not so good solution is to calculate the sum of every possible subarray and the maximum of those would be the solution. We can start from index ***0*** and calculate the sum of every possible subarray starting with the element ***A[0],*** as shown in the figure below. Then, we would calculate the sum of every possible subarray starting with ***A[1]***, ***A[2]*** and so on up to ***A[n-1]***, where ***n*** denotes the size of the array (n = 9 in our case). Note that every single element is a subarray itself.



Brute Force Approach: Iteration 0 (left) and Iteration 1 (right)

We will call the maximum sum of subarrays starting with element ***A[i]*** the ***local_maximum*** at index ***i***. Thus after going through all the indices, we would be left with *local_maximum* for all the indices. Finally, we can find the maximum of these *local_maximum*s and we would get the final solution, *i.e.* the maximum sum possible. We would call this the *global_maximum*.

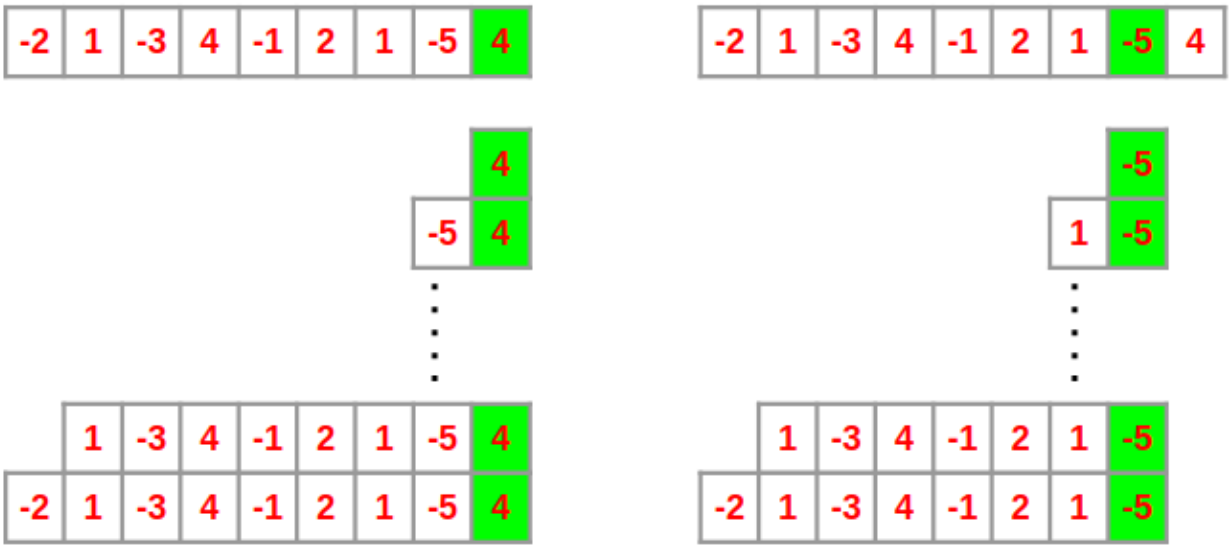
But you might notice that this is not a very good method because as the size of array increases, the number of possible subarrays increases rapidly, thus increasing computational complexity. Or to be more precise, if the size of the array is ***n***, then the time complexity of this solution is $O(n^2)$ which is not very good.

How can we improve this? Is there any way to use the concept of dynamic programming? Let's find out.

Kadane's Algorithm

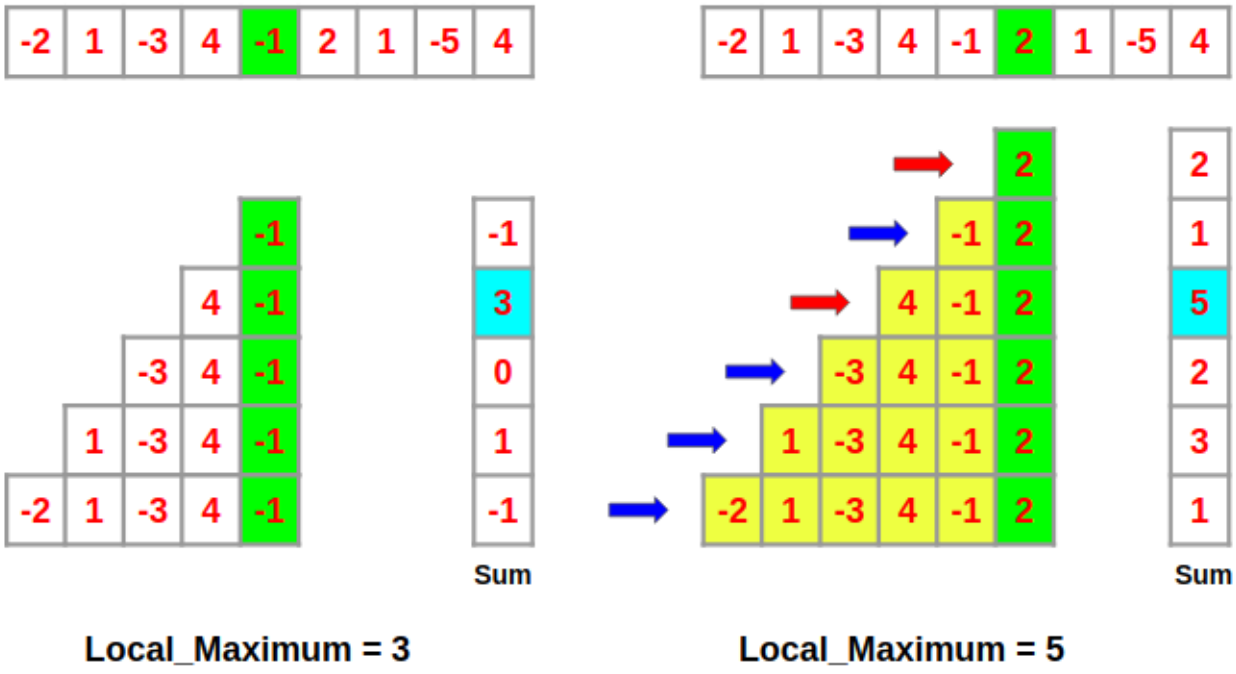
In this section, we would use the brute force approach discussed above again, but this time we would start backward. How would that help? Let's see.

We would start from the last element and calculate the sum of every possible subarray ending with the element ***A[n-1]***, as shown in the figure below. Then, we would calculate the sum of every possible subarray ending with ***A[n-2]***, ***A[n-3]*** and so on up to ***A[0]***.



Backward Brute Force Approach: Iteration 0 (left) and Iteration 1 (right)

Now let's focus on the subarrays ending with the element ***A[4]*** (=-1) and ***A[5]*** (=2) shown in the figure below.



From the figure above, we see that the ***local_maximum[4]*** is equal to 3 which is the sum of the subarray [4, -1]. Now have a look at the subarrays ending with ***A[5]***. You'll notice that these subarrays can be divided into two parts, the subarrays ending with ***A[4]*** (highlighted with yellow) and the single element subarray ***A[5]*** (in green).

Let's say somehow I know the ***local_maximum[4]***. Then we see that to calculate the ***local_maximum[5]***, we don't need to compute the sum of all subarrays ending with ***A[5]*** since we already know the result from arrays ending with ***A[4]***. Note that if array [4, -1] had the maximum sum, then we only need to check the arrays highlighted with the red arrows to calculate ***local_maximum[5]***. And this leads us to the principle on which Kadane's Algorithm works.

> *local_maximum at index i is the maximum of A[i] and the sum of A[i] and local_maximum at index i-1.*

$$\text{local\_maximum}[i] = \max(A[i], A[i] + \text{local\_maximum}[i-1])$$

This way, at every index ***i***, the problem boils down to finding the maximum of just two numbers, ***A[i]*** and ***(A[i] + local_maximum[i-1])***. Thus the maximum subarray problem can be solved by solving these sub-problems of finding ***local_maximums*** recursively. Also, note that ***local_maximum[0]*** would be ***A[0]*** itself.

Using the above method, we need to iterate through the array just once, which is a lot better than our previous brute force approach. Or to be more precise, the time complexity of Kadane's Algorithm is ***O(n)***.

Finally, let's see how this all would work in code.

Code Walkthrough

Below is a very much self-explanatory implementation (in C++) of a function which takes an array as an argument and returns the sum of the maximum subarray.

```
int maxSumSubArray(vector<int> &A)
{
    int n = A.size(); // Size of the array
    int local_max = 0;
    int global_max = INT_MIN; // -Infinity

    for (int i = 0; i < n; i++)
    {
        local_max = max(A[i], A[i] + local_max);
        if (local_max > global_max)
        {
           global_max = local_max;
        }
    }

    return global_max;
}
```

Note that instead of using an array to store ***local_maximums***, we are simply storing the latest ***local_maximum*** in an *int* type variable '*local_max*' because that's what we need to calculate next ***local_maximum***. Also, as we are using a variable '*global_max*' to keep track of the maximum value of ***local_maximum***, which in the end comes out to be the required output.

Conclusion

Because of the way this algorithm uses optimal substructures (the maximum subarray ending at each position is calculated in a simple way from a related but smaller and overlapping subproblem: the maximum subarray ending at the previous position) this algorithm can be viewed as a simple example of dynamic programming. Kadane's

algorithm is able to find the maximum sum of a contiguous subarray in an array with a runtime of $O(n)$.

Programming Kadane Algorithms Dynamic Programming