**Kadane’s Algorithm**

[Kadanes Algorithm Source](https://medium.com/@rsinghal757/kadanes-algorithm-dynamic-programming-how-and-why-does-it-work-3fd8849ed73d)

**Dynamic Programming**

* Dynamic Programming is a method for solving a complex problem by
  + Breaking the problem 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.

|

\*writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper\*  
"What's that equal to?"  
\*counting\* "Eight!"  
\*writes down another "1+" on the left\*  
"What about that?"  
\*quickly\* "Nine!"  
"How'd you know it was nine so fast?"  
"You just added one more"  
"So you didn't need to recount because you remembered there were eight!

*Dynamic Programming* is just a fancy way to say 'remembering stuff to save time later'"

|

**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** of numbers.
* Note that the description of a problem may provide useful hints.
* Kanade’s Algorithm, in particular can be used on **arrays** that are:
  + One-dimensional

and

* + Unsorted
* … to find **contiguous subarrays** with some kind of **condition**

**Example**

Find the contiguous subarray with the largest sum in the integer array [4, -1, 2, 1]

Diagram

Description automatically generated

* In this case, looking at the photo, we know that this contiguous subarray gives us the maximum possible sum, which is 6.
* Let’s explore some of the possible ways to solve this algorithm.

**Brute Force Approach**

* A brute force approach would be to calculate the sum of every possible subarray and then keep an auxiliary integer to keep track of the maximum sum of those subarrays.
* Note that every single element is a subarray itself.
* We can start from index ***0*** and calculate the sum of every possible subarray starting with the element ***A[0]***.
* Then, we would calculate the sum of every possible subarray starting with ***A[1]***, then starting with ***A[2]*** and so on up to ***A[n-1]***,where***n***denotes the size of the array (n = 9 in our case).

Yellow = current index

Green = current subarray

i = 0, j = 0-9

**currentSum** = 0 **maxSum** = Integer.MIN\_VALUE

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |

**currentSum** = -2 **maxSum** = -2

|  |
| --- |
| -2 |

**currentSum** = -1 **maxSum** = -1

|  |  |
| --- | --- |
| -2 | 1 |

**currentSum** = -4 **maxSum** = -1

|  |  |  |
| --- | --- | --- |
| -2 | 1 | -3 |

**currentSum** = 0 **maxSum** = 0

|  |  |  |  |
| --- | --- | --- | --- |
| -2 | 1 | -3 | 4 |

**currentSum** = -1 **maxSum** = 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 |

**currentSum** = 1 **maxSum** = 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 |

**currentSum** = 2 **maxSum** = 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 | 1 |

**currentSum** = -3 **maxSum** = 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 |

**currentSum** = 1 **maxSum** = 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |

i = 1, j = 1-9

**currentSum** = 1 **maxSum** = 2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |

**currentSum** = 1 **maxSum** = 2

|  |  |
| --- | --- |
|  | 1 |

**currentSum** = -2 **maxSum** = 2

|  |  |  |
| --- | --- | --- |
|  | 1 | -3 |

**currentSum** = 2 **maxSum** = 2

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 | -3 | 4 |

**currentSum** = 1 **maxSum** = 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | -3 | 4 | -1 |

**currentSum** = 3 **maxSum** = 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | -3 | 4 | -1 | 2 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | -3 | 4 | -1 | 2 | 1 |

**currentSum** = 4 **maxSum** = 4

**currentSum** = -1 **maxSum** = 4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | -3 | 4 | -1 | 2 | 1 | -5 |

**currentSum** = 3 **maxSum** = 4

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |

… and so on for every i in the array.

* As you may have noticed, the runtime for this algorithm compares each element against every other element starting at i:

int maxSum = Integer.MIN\_VALUE;

for(int i = 0; i < nums.length; i++)

{

int currentSum = 0;

for(int j = i; j < nums.length; j++)

{

currentSum += nums[i];

maxSum = Math.max(maxSum, currentSum);

}

}

* This calculates the value of every possible subarray in the array.
* The runtime complexity of this solution is ***O(n²)***.

**We can clearly do better, but how?**

* If you look at the example of the calculations for the first two indices, you will see a lot of **repeated work**.
* We can save on runtime by caching information we already know for future calculations.
* For example, how did you calculate the currentSum when going from index to index?
  + You took the previous currentSum, then added the next index’s value onto it.
  + This is the perfect example of caching.

**How Can We Improve This?**

* Let’s try the brute force approach again, but this time we start **backwards**.
* We start from the last element in the array ***A[n-1]*** and calculate the sum of every possible subarray going backwards.
* Then, we would calculate the sum of every possible subarray ending with ***A[n-2]*** then ***A[n-3]*** and so on up to ***A[0]***.

Yellow = current index

Green = current subarray

i = 0

**currentSum** = 0 **maxSum** = Integer.MIN\_VALUE

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |

**currentSum** = 4 **maxSum** = 4

|  |
| --- |
| 4 |

**currentSum** = -1 **maxSum** = -1

|  |  |
| --- | --- |
| -5 | 4 |

**currentSum** = 0 **maxSum** = 0

|  |  |  |
| --- | --- | --- |
| 1 | -5 | 4 |

**currentSum** = 2 **maxSum** = 2

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 1 | -5 | 4 |

**currentSum** = 1 **maxSum** = 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -1 | 2 | 1 | -5 | 4 |

**currentSum** = 5 **maxSum** = 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | -1 | 2 | 1 | -5 | 4 |

**currentSum** = 2 **maxSum** = 5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| -3 | 4 | -1 | 2 | 1 | -5 | 4 |

**currentSum** = 3 **maxSum** = 5

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |

**currentSum** = 1 **maxSum** = 5

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |

* We will call the maximum sum of subarrays starting with element ***A[i]*** the ***local\_maximum*** at index ***i***.
* We iterate over every possible subarray in the array, comparing two subarrays at each iteration in the loop.
* During each iteration, we find a ***local\_maximum*** for the current two subarrays we are comparing and compare that to a ***global\_maximum***.
* That way if there were earlier subarray with a greater maximum value, we would keep a record of it in **global\_maximum** while we explore the rest of the array using **local\_maximum**.
* As an example, 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 below, 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.

Chart

Description automatically generated with medium confidence

*local\_maximum at index i is the maximum of A[i] and the sum of A[i] and local\_maximum at index i-1.*

**Kadane’s Algorithm**

* 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 a *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*.