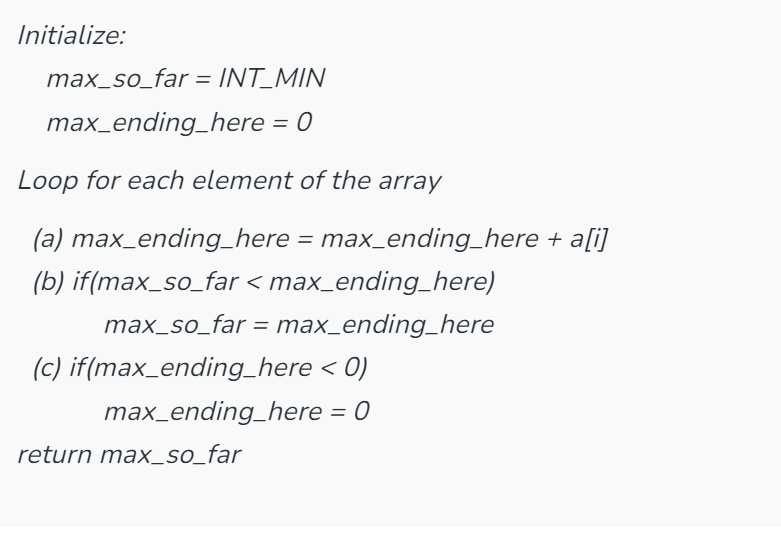
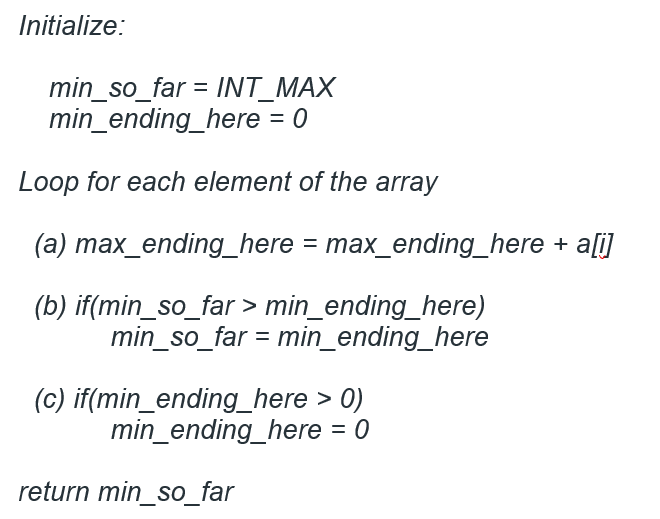
**Kadane Algorithm**

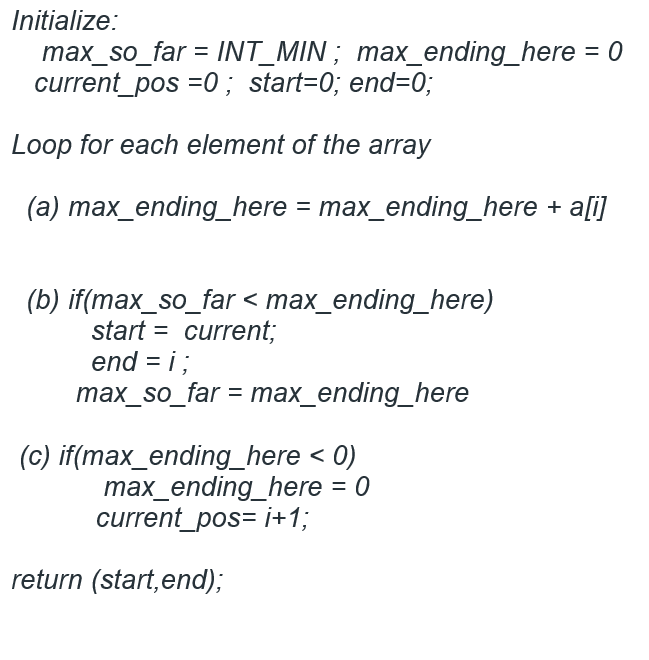
**Algo:** To find maximum contiguous array sum.



**Algo:** To find minimum contiguous array sum.

**

**Algo**: To Find the start and end pos of Maximum contiguous Array.

**

**Dutch National flag algorithm**.

For DNF (Dutch National Flag), we sort an array of 0, 1, and 2's in linear time that does not consume any extra space. We must keep in mind that this algorithm can be implemented only on an array that has three unique elements.(but we must know three unique element)

*The idea is to sort the array of size N using three pointers: low = 0, mid = 0 and high = N – 1 such that the array is divided into three parts:*

* ***arr[0] to arr[low – 1]:****This part will have all the****zeros****.*
* ***arr[low] to arr[mid – 1]:****This part will have all the****ones****.*
* ***arr[mid] to arr[N – 1]:****This part will have all the****twos****.*

 If arr[mid] == 0: Swap the element at mid with the element at low, then move both low and mid forward.

 If arr[mid] == 1: The element is already in the correct middle section, so just move mid forward.

 If arr[mid] == 2: Swap the element at mid with the element at high, then move high backward. Do **not** increment mid, as the new value at mid after the swap still needs to be checked.

The loop terminates when mid exceeds high, meaning all elements have been sorted.

function dutch\_flag\_sort(arr):

low = 0 mid = 0 high = len(arr) - 1

while mid <= high:

if arr[mid] == 0:

# Swap arr[low] and arr[mid], move low and mid pointers

swap(arr[low], arr[mid])

low = low + 1

mid = mid + 1

else if arr[mid] == 1:

# Move mid pointer only (since 1 is in the correct middle position)

mid = mid + 1

else:

# arr[mid] == 2

# Swap arr[mid] and arr[high], move high pointer

swap(arr[mid], arr[high])

high = high – 1

return arr

**Boyer-Moore Majority Voting Algorithm**.

The **Boyer-Moore voting** algorithm is one of the popular optimal algorithms which is used to find the majority element among the given elements that have more than N/ 2 occurrences.

This algorithm works on the fact that if an element occurs more than N/2 times, it means that the remaining elements other than this would be less than N/2. So, let us check the proceeding of the algorithm.

*1.****Initialization:****We start by initializing two variables, “candidate” and “count,” to null and 0, respectively.*

*2.****Majority Element Detection:****We iterate through the array and update the “candidate” and “count” variables as follows:*

* *If “count” is 0, we set the current element as the “candidate.”*
* *If the current element is equal to the “candidate,” we increment “count” by 1.*
* *If the current element is different from the “candidate,” we decrement “count” by 1.*

*3.****Verification:****After the iteration, the “candidate” variable stores a potential majority element. We then perform a second pass to verify if the “candidate” indeed appears more than n/2 times in the array.*

*// Function to find majority element*

**public** **static** int findMajority(int[] nums)

{

int count = 0, candidate = -1;

*// Finding majority candidate*

**for** (int index = 0; index < nums.length; index++) {

**if** (count == 0) {

candidate = nums[index];

count = 1;

}

**else** {

**if** (nums[index] == candidate)

count++;

**else**

count--;

}

}

*// Checking if majority candidate occurs more than*

*// n/2 times*

count = 0;

**for** (int index = 0; index < nums.length; index++) {

**if** (nums[index] == candidate)

count++;

}

**if** (count > (nums.length / 2))

**return** candidate;

**return** -1;

}

**Generalization for Boyer-Moore Majority Voting Algorithm:** *In some scenarios, we may need to find elements that occur more than****n/k****times, where****k****is a positive integer.(Use HashMap to store frequency and count max frequency).*