Array Challenges Part 3

Maximum Sum Subarray Array

1. **Brute Force:**
   Idea: For each subarray arr[i..j], calculate its sum.
   Time Complexity: $O(N^3)$
   Space Complexity: O(1)

```
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        int sum = 0;
        for (int k = i; k <= j; k++) {
            sum += arr[k];
        }
        maxSum = max(maxSum, sum);
    }
}
```

2. **Prefix Sum Technique:**

   Idea: For each subarray arr[i..j], calculate its sum. Using prefix sum can reduce time to calculate the sum of arr[i..j] to O(1)

   Time Complexity: $O(N^2)$

   Space Complexity: O(N)

```
int cumsum[n + 1];
cumsum[0] = 0;
for (int i = 1; i <= n; i++) {
    cumsum[i] = cumsum[i - 1] + arr[i - 1];
}
int maxSum = INT_MIN;
for (int i = 1; i <= n; i++) {
    int sum = 0;
    maxSum = max(maxSum, cumsum[i]);
    for (int j = 1; j <= i; j++) {
        sum = cumsum[i] - cumsum[j - 1];
        maxSum = max(maxSum, sum);
    }
}
cout << maxSum << endl;
```

3. **Kadane's Algorithm:**

   Idea: Start taking the sum of the array, as soon as it gets negative, discard the current subarray, and start a new sum.

   Time Complexity: O(N)

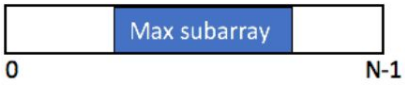
   Space Complexity: O(1)

```
int currentSum = 0;
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
    currentSum += arr[i];
    if (currentSum < 0) {
        currentSum = 0;
    }
    maxSum = max(maxSum, currentSum);
}
```

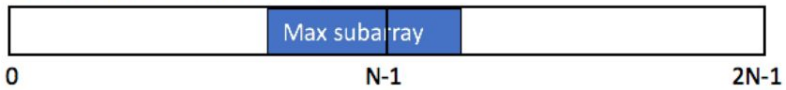
Maximum Sum Circular Subarray:

Idea: There will 2 cases,

Case 1: max subarray is not circular.



Case 2: max subarray is circular.



equals

To get the Min subarray we multiply the array by -1 and get the maximum sum subarray.

Time Complexity: O(N)

```
int wrapsum;
int nonwrapsum;
nonwrapsum = kadane(arr, n);
int totalsum = 0;
for (int i = 0; i < n; i++) {
    totalsum += arr[i];
    arr[i] = -arr[i];
}
wrapsum = totalsum + kadane(arr, n);
```

Pair Target Sum Problem

Find whether there exist 2 numbers that sum to K.

Important: The Array should be sorted for two pointer approach.

Idea: keep a low and high pointer, decide which pointer to move on the basis of arr[low] + arr[high].

Time Complexity: O(N)

Space Complexity: O(1)

```cpp
bool pairsum(int arr[], int n, int k) {
    int low = 0;
    int high = n - 1;
    while (low < high) {
        if (arr[low] + arr[high] == k) {
            cout << low << " " << high << endl;
            return true;
        }
        else if (arr[low] + arr[high] > k) {
            high--;
        }
        else {
            low++;
        }
    }
    return false;
}
```