# Cheat Sheet: Kadane’s Algorithm & Maximum Subarray Sum (Java - DSA)

## 1. Introduction

Kadane’s Algorithm is a **Dynamic Programming** approach used to solve the problem of finding the **maximum sum subarray** from a given integer array. This is a classic problem that tests understanding of greedy choices and optimization in contiguous subarrays.

## 2. Problem Statement

Given an integer array nums, find the **contiguous subarray** (containing at least one number) which has the **largest sum** and return its **sum**.

## 3. Brute Force Approach

### Description:

Try every possible subarray, compute its sum, and return the maximum.

### Code:

public int maxSubarraySumBruteForce(int[] nums) {

int maxSum = Integer.MIN\_VALUE;

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

int sum = 0;

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

sum += nums[j];

maxSum = Math.max(maxSum, sum);

}

}

return maxSum;

}

### Time and Space:

* Time: O(n²)
* Space: O(1)

## 4. Optimal Approach: Kadane’s Algorithm

### Description:

Kadane’s Algorithm scans the array and at each step, decides whether to:

1. Start a new subarray with the current element.
2. Extend the existing subarray by adding the current element.

It maintains:

* currMax: Maximum subarray sum ending at current index.
* maxSoFar: Global maximum across all indices.

### Code:

public int maxSubArrayKadane(int[] nums) {

int maxSoFar = nums[0];

int currMax = nums[0];

for (int i = 1; i < nums.length; i++) {

currMax = Math.max(nums[i], currMax + nums[i]);

maxSoFar = Math.max(maxSoFar, currMax);

}

return maxSoFar;

}

### Time and Space:

* Time: O(n)
* Space: O(1)

## 5. Track Subarray Indices

### Description:

Sometimes, we need to return the **actual subarray** in addition to the sum. To do that, we track the start and end positions.

### Code:

public int[] maxSubarrayWithIndices(int[] nums) {

int maxSoFar = nums[0], currMax = nums[0];

int start = 0, end = 0, tempStart = 0;

for (int i = 1; i < nums.length; i++) {

if (nums[i] > currMax + nums[i]) {

currMax = nums[i];

tempStart = i;

} else {

currMax += nums[i];

}

if (currMax > maxSoFar) {

maxSoFar = currMax;

start = tempStart;

end = i;

}

}

return new int[]{maxSoFar, start, end};

}

## 6. Example

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: [4, -1, 2, 1] is the subarray with the maximum sum.

## 7. Time & Space Complexity Comparison

| **Approach** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| Brute Force | O(n²) | O(1) |
| Kadane’s Algorithm | O(n) | O(1) |

## 8. Use Cases in Real Problems

* Stock price analysis
* Temperature fluctuation patterns
* Maximum profit from subarray intervals
* Used in many subarray-based coding interviews

## 9. Practice Problems

1. [Leetcode 53. Maximum Subarray](https://leetcode.com/problems/maximum-subarray/)
2. Find maximum sum circular subarray
3. Find maximum product subarray

## 10. Tips & Tricks

* Works with both positive and negative integers.
* Can be extended to 2D arrays (Kadane’s in 2D).
* Modify logic to find minimum subarray sum (negation).

**Theoretical Explanation of Array Series Topics**

**1. Maximum Subarray Problem**

**Definition:**

The Maximum Subarray Problem involves finding the contiguous subarray within a one-dimensional array of numbers that has the largest sum. A subarray is defined as a contiguous part of the array.

**Key Points:**

- Contiguous Elements: The subarray must consist of consecutive elements from the original array.

- At Least One Element: The subarray must contain at least one number (even if all numbers are negative).

- Applications: Stock price analysis (best time to buy/sell), signal processing, data mining.

**Example:**

For the array `[-2, 1, -3, 4, -1, 2, 1, -5, 4]`, the maximum subarray is `[4, -1, 2, 1]` with sum `6`.

**2. Brute Force Approach**

**Method:**

Check all possible subarrays and compute their sums to find the maximum.

**Steps:**

1. Iterate over all possible starting indices `i`.

2. For each `i`, iterate over all ending indices `j ≥ i`.

3. Compute the sum of elements from `i` to `j`.

4. Track the maximum sum encountered.

**Time Complexity**:

- O(n²) for nested loops.

- O(1) space (if sums are computed on the fly).

**Limitations:**

- Inefficient for large arrays (e.g., n > 10,000).

**3. Kadane’s Algorithm**

**Optimal Solution:**

- Time Complexity: O(n) (single pass through the array).

- Space Complexity: O(1) (uses constant extra space).

**Intuition:**

Instead of recomputing sums for every subarray, Kadane’s algorithm efficiently tracks:

1. Maximum Subarray Ending at Current Position (`maxEndingHere`):

- Either extend the previous subarray or start a new subarray at the current element.

- `maxEndingHere = max(nums[i], maxEndingHere + nums[i])`.

2. Global Maximum Subarray (`maxSoFar`):

- Updated whenever a new maximum is found.

- `maxSoFar = max(maxSoFar, maxEndingHere)`.

**Why It Works:**

- Negative numbers reset `maxEndingHere` (starting a new subarray is better).

- Positive numbers extend the current subarray.

**Example Walkthrough:**

Array: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

**Step-by-Step:**

i=0: maxEnd=-2, maxFar=-2

i=1: maxEnd=max(1, -2+1)=1, maxFar=max(-2,1)=1

i=2: maxEnd=max(-3, 1-3)=-2, maxFar=max(1,-2)=1

i=3: maxEnd=max(4, -2+4)=4, maxFar=max(1,4)=4

i=4: maxEnd=max(-1, 4-1)=3, maxFar=max(4,3)=4

i=5: maxEnd=max(2, 3+2)=5, maxFar=max(4,5)=5

i=6: maxEnd=max(1, 5+1)=6, maxFar=max(5,6)=6

i=7: maxEnd=max(-5, 6-5)=1, maxFar=max(6,1)=6

i=8: maxEnd=max(4, 1+4)=5, maxFar=max(6,5)=6

Result: 6

**4. Edge Cases & Variations**

**Case 1: All Negative Numbers**

- Input: `[-2, -3, -1, -5]`

- Output: `-1` (subarray `[-1]`).

- Behavior: Kadane’s still works because it compares each element individually.

**Case 2: Circular Subarray (Wrap-Around)**

- Problem: Maximum subarray may wrap around the array ends (e.g., `[5, -3, 5]` → `[5, -3, 5] = 7`).

- Solution:

- Compute regular Kadane’s result.

- Compute total sum – minimum subarray sum (using inverted Kadane’s).

- Maximum of these two is the answer.

**Case 3: Maximum Product Subarray**

- Problem: Find the contiguous subarray with the largest product (e.g., `[2, 3, -2, 4]` → `6`).

- Solution: Track both `minEndingHere` and `maxEndingHere` (since two negatives can yield a positive).

**5. Java Implementation**

**Basic Kadane’s Algorithm:**

public int maxSubArray(int[] nums) {

int maxEndingHere = nums[0];

int maxSoFar = nums[0];

for (int i = 1; i < nums.length; i++) {

maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);

maxSoFar = Math.max(maxSoFar, maxEndingHere);

}

return maxSoFar;

}

**Extended (Tracking Subarray Indices):**

public int[] maxSubArrayWithIndices(int[] nums) {

int maxEndingHere = nums[0], maxSoFar = nums[0];

int start = 0, end = 0, tempStart = 0;

for (int i = 1; i < nums.length; i++) {

if (nums[i] > maxEndingHere + nums[i]) {

maxEndingHere = nums[i];

tempStart = i;

} else {

maxEndingHere += nums[i];

}

if (maxEndingHere > maxSoFar) {

maxSoFar = maxEndingHere;

start = tempStart;

end = i;

}

}

return new int[]{maxSoFar, start, end};

}