**Maximum Subarray**

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

A **subarray** is a **contiguous** part of an array.

**Example 1:**

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

**Output:** 6

**Explanation:** [4,-1,2,1] has the largest sum = 6.

**Example 2:**

**Input:** nums = [1]

**Output:** 1

**Example 3:**

**Input:** nums = [5,4,-1,7,8]

**Output:** 23

**Constraints:**

* 1 <= nums.length <= 105
* -104 <= nums[i] <= 104

**Follow up:** If you have figured out the O(n) solution, try coding another solution using the **divide and conquer** approach, which is more subtle.

Solution

Approach 1: Optimized Brute Force

**Intuition**

This algorithm doesn't reliably run under the time limit here on LeetCode. We'll still look briefly at it though, as in an interview scenario it would be a great start if you're struggling to come up with a better approach.

Calculate the sum of all subarrays, and keep track of the best one. To actually generate all subarrays would take O(N^3)*O*(*N*3) time, but with a little optimization, we can achieve brute force in O(N^2)*O*(*N*2) time. The trick is to recognize that all of the subarrays starting at a particular value will share a common prefix.

**Algorithm**

1. Initialize a variable maxSubarray = -infinity to keep track of the best subarray. We need to use negative infinity, not 0, because it is possible that there are only negative numbers in the array.
2. Use a for loop that considers each index of the array as a starting point.
3. For each starting point, create a variable currentSubarray = 0. Then, loop through the array from the starting index, adding each element to currentSubarray. Every time we add an element it represents a possible subarray - so continuously update maxSubarray to contain the maximum out of the currentSubarray and itself.
4. Return maxSubarray.

**Implementation**

class Solution {

public int maxSubArray(int[] nums) {

int maxSubarray = Integer.MIN\_VALUE;

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

int currentSubarray = 0;

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

currentSubarray += nums[j];

maxSubarray = Math.max(maxSubarray, currentSubarray);

}

}

return maxSubarray;

}

}

**Complexity Analysis**

* Time complexity: O(N^2)*O*(*N*2), where N*N* is the length of nums.

We use 2 nested for loops, with each loop iterating through nums.

* Space complexity: O(1)*O*(1)

No matter how big the input is, we are only ever using 2 variables: ans and currentSubarray.

Approach 2: Dynamic Programming, Kadane's Algorithm

**Intuition**

Whenever you see a question that asks for the maximum or minimum of something, consider Dynamic Programming as a possibility. The difficult part of this problem is figuring out when a negative number is "worth" keeping in a subarray. This question in particular is a popular problem that can be solved using an algorithm called [Kadane's Algorithm](https://en.wikipedia.org/wiki/Maximum_subarray_problem" \l "Kadane's_algorithm). If you're good at problem solving though, it's quite likely you'll be able to come up with the algorithm on your own. This algorithm also has a very greedy-like intuition behind it.

Let's focus on one important part: where the optimal subarray **begins**. We'll use the following example.

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

We can see that the optimal subarray couldn't possibly involve the first 3 values - the overall sum of those numbers would always *subtract* from the total. Therefore, the subarray either starts at the first 4, or somewhere further to the right.

What if we had this example though?

nums = [-2,1000000000,-3,4,-1,2,1,-5,4]

We need a general way to figure out when a part of the array is **worth** keeping.

As expected, any subarray whose sum is *positive* is worth keeping. Let's start with an empty array, and iterate through the input, adding numbers to our array as we go along. Whenever the sum of the array is negative, we know the entire array is not worth keeping, so we'll reset it back to an empty array.

However, we don't actually need to build the subarray, we can just keep an integer variable current\_subarray and add the values of each element there. When it becomes negative, we reset it to 0 (an empty array).

**Algorithm**

1. Initialize 2 integer variables. Set both of them equal to the first value in the array.
   * currentSubarray will keep the running count of the current subarray we are focusing on.
   * maxSubarray will be our final return value. Continuously update it whenever we find a bigger subarray.
2. Iterate through the array, starting with the 2nd element (as we used the first element to initialize our variables). For each number, add it to the currentSubarray we are building. If currentSubarray becomes negative, we know it isn't worth keeping, so throw it away. Remember to update maxSubarray every time we find a new maximum.
3. Return maxSubarray.

**Implementation**

A clever way to update currentSubarray is using currentSubarray = max(num, currentSubarray + num). If currentSubarray is negative, then num > currentSubarray + num.

class Solution {

public int maxSubArray(int[] nums) {

// Initialize our variables using the first element.

int currentSubarray = nums[0];

int maxSubarray = nums[0];

// Start with the 2nd element since we already used the first one.

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

int num = nums[i];

// If current\_subarray is negative, throw it away. Otherwise, keep adding to it.

currentSubarray = Math.max(num, currentSubarray + num);

maxSubarray = Math.max(maxSubarray, currentSubarray);

}

return maxSubarray;

}

}

**Complexity Analysis**

* Time complexity: O(N)*O*(*N*), where N*N* is the length of nums.

We iterate through every element of nums exactly once.

* Space complexity: O(1)*O*(1)

No matter how long the input is, we are only ever using 2 variables: currentSubarray and maxSubarray.

Approach 3: Divide and Conquer (Advanced)

**Intuition**

This approach is slower than the second approach and uses more space, but it's still a nice and different way to approach the problem. In an interview, sometimes you may be asked for alternative ways to solve a problem - and divide and conquer is an extremely common type of algorithm. This solution will make use of recursion - if you aren't familiar with recursion, check out the [recursion explore card](https://leetcode.com/explore/featured/card/recursion-i/).

Divide and conquer algorithms involve splitting up the input into smaller chunks until they're small enough to be easily solved, and then combining the solutions to get the final overall solution. If you're unfamiliar with them, check out [this explore card](https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/2897/).

If we were to split our input in half, then logically the optimal subarray either:

* Uses elements only from the left side
* Uses elements only from the right side
* Uses a combination of elements from both the left and right side

Thus, the answer is simply the largest of:

* The maximum subarray contained only in the left side
* The maximum subarray contained only in the right side
* The maximum subarray that can use elements from both sides

Finding the maximum subarray from the left and right half is straightforward - just recurse using the respective half of the array. So, the hard part is figuring out how to find the best subarray that uses elements from both sides. Lets use a smaller example, nums = [5, -2, 1, -3, 4, -2, 1]. Since we want to use elements from both sides, we also must use the middle element, -3. Now, we can also take from the left and the right, but every element must be connected to the middle (since we're still forming a subarray).

The fact that every element must be connected to the middle actually makes our lives a lot easier - every subarray we consider **must** contain the element immediately beside the center, which means there are only as many subarrays as there are elements. In our example, the right side is [4, -2, 1]. There are only 3 subarrays to consider - [4], [4, -2], and [4, -2, 1]. To find the best chain of elements we can take from a half, iterate from the middle to the end (start of the array for the left half) and continuously update some counter curr.

1 / 11

**Algorithm**

Now that we know how to find the best subarray containing elements from both sides of any given array, the algorithm is as follows:

1. Define a helper function that we will use for recursion.
   * This function will take an input left and right, which defines the bounds of the array. The return value of this function will be the best possible subarray for the array that fits between left and right.
   * If left > right, we have an empty array. Return negative infinity.
   * Find the midpoint of our array. This is (left + right) / 2, rounded down. Using this midpoint, find the best possible subarray that uses elements from both sides of the array with the algorithm detailed in the animation above.
   * The best subarray using elements from both sides is only 1 of 3 possibilities. We still need to find the best subarray using only the left or right halves. So, call this function again, once with the left half, and once with the right half.
   * Return the largest of the 3 values - the best left half sum, the best right half sum, and the best combined sum.
2. Call our helper function with the entire input array (left = 0, right = length - 1). This is our final answer, so return it.

**Implementation**

class Solution {

private int[] numsArray;

public int maxSubArray(int[] nums) {

numsArray = nums;

// Our helper function is designed to solve this problem for

// any array - so just call it using the entire input!

return findBestSubarray(0, numsArray.length - 1);

}

private int findBestSubarray(int left, int right) {

// Base case - empty array.

if (left > right) {

return Integer.MIN\_VALUE;

}

int mid = Math.floorDiv(left + right, 2);

int curr = 0;

int bestLeftSum = 0;

int bestRightSum = 0;

// Iterate from the middle to the beginning.

for (int i = mid - 1; i >= left; i--) {

curr += numsArray[i];

bestLeftSum = Math.max(bestLeftSum, curr);

}

// Reset curr and iterate from the middle to the end.

curr = 0;

for (int i = mid + 1; i <= right; i++) {

curr += numsArray[i];

bestRightSum = Math.max(bestRightSum, curr);

}

// The bestCombinedSum uses the middle element and the best

// possible sum from each half.

int bestCombinedSum = numsArray[mid] + bestLeftSum + bestRightSum;

// Find the best subarray possible from both halves.

int leftHalf = findBestSubarray(left, mid - 1);

int rightHalf = findBestSubarray(mid + 1, right);

// The largest of the 3 is the answer for any given input array.

return Math.max(bestCombinedSum, Math.max(leftHalf, rightHalf));

}

}

**Complexity Analysis**

* Time complexity: O(N \cdot \log N)*O*(*N*⋅log*N*), where N*N* is the length of nums.

On our first call to findBestSubarray, we use for loops to visit every element of nums. Then, we split the array in half and call findBestSubarray with each half. Both those calls will then iterate through every element in that half, which combined is every element of nums again. Then, both those halves will be split in half, and 4 more calls to findBestSubarray will happen, each with a quarter of nums. As you can see, every time the array is split, we still need to handle every element of the original input nums. We have to do this \log Nlog*N* times since that's how many times an array can be split in half.

* Space complexity: O(\log N)*O*(log*N*), where N*N* is the length of nums.

The extra space we use relative to input size is solely occupied by the recursion stack. Each time the array gets split in half, another call of findBestSubarray will be added to the recursion stack, until calls start to get resolved by the base case - remember, the base case happens at an empty array, which occurs after \log Nlog*N* calls.