<https://leetcode.com/problems/maximum-subarray/>

Given an integer array nums, find the subarray with the largest sum, and return *its sum*.

**Example 1:**

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

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

**Example 2:**

Input: nums = [1]

Output: 1

Explanation: The subarray [1] has the largest sum 1.

**Example 3:**

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

Output: 23

Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

**Constraints:**

1 <= nums.length <= 10^5

-10^4 <= nums[i] <= 10^4

 **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.

**Attempt 1: 2023-03-09**

**Solution 1: Brute Force using three points control 'start' , 'end' and loop respectively as O(N^3) (10 min, TLE)**

class Solution {

    public int maxSubArray(int[] nums) {

        int n = nums.length;

        int maxSum = Integer.MIN\_VALUE;

        // 'i' control 'start', 'j' control 'end',

        // 'k' control loop

        for(int i = 0; i < n; i++) {

            for(int j = i; j < n; j++) {

                int curSum = 0;

                for(int k = i; k <= j; k++) {

                    curSum += nums[k];

                }

                maxSum = Math.max(maxSum, curSum);

            }

        }

        return maxSum;

    }

}

Time complexity: O(n^3)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/405559/easy-understand-java-solutions-with-explanations-b-f-divide-and-conquer-dp/>

<https://leetcode.com/problems/maximum-subarray/solutions/1411166/c-brute-force-to-optimal-3-solutions-easy-to-understand/>

Run three loops, get all possible subarrays in two loops and their sum in another loop, then return the maximum of them.For each element, we construct all possible subarrays starting from this element. Totally there are at most

N^2 subarrays. Also, calculating the sum of each subarray takes O(N).

public int maxSubArray(int[] nums) {

  int n = nums.length;

  int maxSum = Integer.MIN\_VALUE;

  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 += nums[k];

      }

      maxSum = Math.max(maxSum, sum);

    }

  }

  return maxSum;

}

**Time:** O(N^3)

**Space:** O(1)

**Solution 2:  Brute Force with optimization using two points control 'start' , ('end' & loop) respectively as O(N^2) (30 min, TLE)**

class Solution {

    public int maxSubArray(int[] nums) {

        int n = nums.length;

        int maxSum = Integer.MIN\_VALUE;

        // 'i' control 'start', 'j' control 'end' and loop

        for(int i = 0; i < n; i++) {

            // The optimization: no need additional variable to tag 'end'

            // index, now 'i' tag the different 'start', 'j' play two roles

            // (1) loop through (2) in each loop tag the different 'end'

            int curSum = 0;

            for(int j = i; j < n; j++) {

                curSum += nums[j];

                maxSum = Math.max(maxSum, curSum);

            }

        }

        return maxSum;

    }

}

Time complexity: O(n^2)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/1411166/c-brute-force-to-optimal-3-solutions-easy-to-understand/>

We can start with brute-force by trying out every possible sub-array in nums and choosing the one with maximum sum.

If we optimize the above approach, we can get sum of all possible subarrays in two loops only, then return the maximum so far. The optimization: no need additional variable to tag 'end' index, now 'i' tag the different 'start', 'j' play two roles (1) loop through (2) in each loop tag the different 'end'

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int max\_sum = INT\_MIN;

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

            int curr\_sum = 0;

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

                curr\_sum += nums[j];

                //cout<<curr\_sum<<endl;

                max\_sum = max(max\_sum, curr\_sum);

            }

        }

    return max\_sum;

    }

};

**Time:** O(N^2)

**Space:** O(1)

**Solution 3: Recursion (360 min, too hard to organize the logic on hard point 'considerPreValue', TLE)**

**Style 1: boolean 'considerPreValue' flag, return int**

class Solution {

    public int maxSubArray(int[] nums) {

        return helper(nums, 0, false);

    }

    private int helper(int[] nums, int index, boolean considerPreValue) {

        // considerPreValue = true means: Took at least one element and

// reached the end, now cannot take anything so return 0, else

// considerPreValue = false means: Haven't taken anything till now,

        // not a valid case, return -1e9 as minimum value under

        // 1 <= nums.length <= 10^5 and -10^4 <= nums[i] <= 10^4 constraints

        if(index >= nums.length) {

            return considerPreValue ? 0 : (int)-1e9;

        }

        // If taken a value previously, either return with that value

        // (by not continue with current value and return 0 to maintain

        // previous value), or take current value again and recurse

        if(considerPreValue) {

            return Math.max(0, nums[index] + helper(nums, index + 1, true));

        }

        // If not taken any value before, take current value and recurse

        // or not take current value and recurse

        return Math.max(nums[index] + helper(nums, index + 1, true), helper(nums, index + 1, false));

    }

}

Time complexity: O(n^2)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/1595195/c-python-7-simple-solutions-w-explanation-brute-force-dp-kadane-divide-conquer/>

❌ ***Solution - II (Recursive) [TLE]***

Another way to consider every subarray and return the one that has maximum sum is using recursive approach. Here we can state the approach as -

At each index i, we can either pick that element or not pick it.

If we pick current element, then all future element must also be picked since our array needs to be contiguous.

If we had picked any elements till now, we can either end further recursion at any time by returning sum formed till now or we can choose current element and recurse further. This denotes two choices of either choosing the subarray formed from 1st picked element till now or expanding the subarray by choosing current element respectively.

In the code below, we will use mustPick to denote whether we must compulsorily pick current element. When mustPick is true, we must either return 0 or pick current element and recurse further. If mustPick is false, we have both choices of not picking current element and moving on to next element, or picking the current one.

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        return solve(nums, 0, false);

    }

    int solve(vector<int>& A, int i, bool mustPick) {

// our subarray must contain atleast 1 element. If mustPick is false at end means no element is picked and this is not valid case

        if(i >= size(A)) return mustPick ? 0 : -1e5;

        if(mustPick)

            return max(0, A[i] + solve(A, i+1, true)); // either stop here or choose current element and recurse

        return max(solve(A, i+1, false), A[i] + solve(A, i+1, true)); // try both choosing current element or not choosing

    }

};

***Time Complexity :*** O(N^2), we are basically considering every subarray sum and choosing maximum of it.

***Space Complexity :*** O(N), for recursive space

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/2213371/c-recursive-mmemoized-and-divide-and-conquor-methods-no-kadanes-algo/>

class Solution {

public:

    int f(vector<int>& nums, int idx, bool takeit)  {

// took a value and reached the end, now cannot take anything so return 0, else it means you havent taken anything till now, not a valid case

        if(idx >= nums.size()) {

            return takeit ? 0 : -1e9;

        }

        // if taken a value previously, either return with that value, or take curr value again and recurse

        if(takeit) {

            return max(0, nums[idx] + f(nums, idx + 1, true));

        }

        // if not taken any value before, take curr value and recurse or not take curr value and recurse

        return max(nums[idx] + f(nums, idx + 1, true), f(nums, idx + 1, false));

    }

    int maxSubArray(vector<int>& nums) {

        return f(nums, 0, false);

    }

};

**Style 2 to 4 getting from chatGPT**

**Style 2: Change the boolean flag 'considerPreValue' to int 'currentSum', void return**

class Solution {

    private int maxSum = Integer.MIN\_VALUE;

    public int maxSubArray(int[] nums) {

        helper(nums, 0, 0);  // Start DFS from index 0 with an initial sum of 0

        return maxSum;

    }

    private void helper(int[] nums, int index, int currentSum) {

        // Base case: If we reach the end of the array, stop the recursion

        if (index >= nums.length) {

            return;

        }

        // Include the current element in the current subarray or start a new subarray

        currentSum = Math.max(nums[index], currentSum + nums[index]);

        // Update the global maximum sum

        maxSum = Math.max(maxSum, currentSum);

        // Recurse for the next element

        helper(nums, index + 1, currentSum);

    }

}

**Style 3: Change the boolean flag 'considerPreValue' to int 'currentSum', return int similar to Style 1**

class Solution {

    public int maxSubArray(int[] nums) {

        return helper(nums, 0, Integer.MIN\_VALUE, 0);

    }

    private int helper(int[] nums, int index, int globalMax, int currentSum) {

        // Base case: when we reach the end of the array

        if (index >= nums.length) {

            return globalMax;

        }

        // Update the current sum by adding the current number

        currentSum += nums[index];

        // Update the global maximum to track the highest sum found so far

        globalMax = Math.max(globalMax, currentSum);

        // If the current sum is less than 0, reset it to 0

        // (effectively "breaking" the subarray at this point)

        if (currentSum < 0) {

           currentSum = 0;

        }

        // Recurse to the next index

        return helper(nums, index + 1, globalMax, currentSum);

    }

}

**Q: It looks like the condition to reset currentSum to 0 when it is negative comes before updating globalMax ?**

A: The reset to 0 is intended to discard negative sums so that they don’t impact future subarrays. However, the contribution of the current sum must be considered **before** this reset to avoid losing valid results.

This ensures that the algorithm correctly handles edge cases like a single negative number, where the maximum subarray is the number itself.

**Style 4: Evolution from Style 3, but merge explicit reset of 'currentSum' to 0 logic before setup 'globalMax'**

class Solution {

    public int maxSubArray(int[] nums) {

        return helper(nums, 0, Integer.MIN\_VALUE, 0);

    }

    private int helper(int[] nums, int index, int globalMax, int currentSum) {

        // Base case: when we reach the end of the array

        if (index == nums.length) {

            return globalMax;

        }

        // Update the current sum by adding the current number

        // and if the current sum is less than 0, reset it to 0

        // (effectively "breaking" the subarray at this point)

        currentSum = Math.max(nums[index], currentSum + nums[index]);

        // Update the global maximum to track the highest sum found so far

        globalMax = Math.max(globalMax, currentSum);

        // Recurse to the next index

        return helper(nums, index + 1, globalMax, currentSum);

    }

}

**Solution 4: Memoization + Recursion (30 min, based on Solution 3)**

class Solution {

    public int maxSubArray(int[] nums) {

        Integer [][] memo = new Integer[nums.length][2];

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

            Arrays.fill(memo[k], -1);

        }

        return helper(nums, 0, 0, memo);

    }

    // To use 'Integer[][] memo' has to change 'considerPreValue' type from boolean to int

    private int helper(int[] nums, int index, int considerPreValue, Integer[][] memo) {

        // Took a value and reached the end, now cannot take anything so

        // return 0, else it means you haven't taken anything till now,

        // not a valid case, return -1e9 as minimum value under

        // 1 <= nums.length <= 10^5 and -10^4 <= nums[i] <= 10^4 constraints

        if(index >= nums.length) {

            return considerPreValue == 1 ? 0 : (int)-1e9;

        }

        if(memo[index][considerPreValue] != -1) {

            return memo[index][considerPreValue];

        }

        // If taken a value previously, either return with that value

        // (by not continue with current value and return 0 to maintain

        // previous value), or take current value again and recurse

        if(considerPreValue == 1) {

            return memo[index][considerPreValue] = Math.max(0, nums[index] + helper(nums, index + 1, 1, memo));

        }

        // If not taken any value before, take current value and recurse

        // or not take current value and recurse

        return memo[index][considerPreValue] = Math.max(nums[index] + helper(nums, index + 1, 1, memo), helper(nums, index + 1, 0, memo));

    }

}

Time complexity: O(n)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/1595195/c-python-7-simple-solutions-w-explanation-brute-force-dp-kadane-divide-conquer/>

✔️ ***Solution - III (Dynamic Programming - Memoization)***

We can observe a lot of repeated calculations if we draw out the recursive tree for above solution -

                                                f(0, False)              🔽 => repeated calculations

                                            /            \

                                  f(1, False)              f(1, True)

                                  /          \      🔽          \      🔽

                            f(2, False)      f(2, True)          f(2, True)

                          /            \  🔽      \  🔽          \  🔽

                      f(3, False)  f(3,True)    f(3, True)          f(3, True)

                    /        \            \          \                  \

                    ...        ...          ...        ...                ...

These redundant calculations can be eliminated if we store the results for a given state and reuse them later whenever required rather than recalculating them over and over again.

Thus, we can use memoization technique here to make our solution more efficient. Here, we use a dp array where dp[mustPick][i] denotes the maximum sum subarray starting from i and mustPick denotes whether the current element must be picked compulsorily or not.

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        vector<vector<int>> dp(2, vector<int>(size(nums), -1));

        return solve(nums, 0, false, dp);

    }

    int solve(vector<int>& A, int i, bool mustPick, vector<vector<int>>& dp) {

        if(i >= size(A)) return mustPick ? 0 : -1e5;

        if(dp[mustPick][i] != -1) return dp[mustPick][i];

        if(mustPick)

            return dp[mustPick][i] = max(0, A[i] + solve(A, i+1, true, dp));

        return dp[mustPick][i] = max(solve(A, i+1, false, dp), A[i] + solve(A, i+1, true, dp));

    }

};

***Time Complexity :*** O(N), we are calculating each state of the dp just once and memoizing the result. Thus, we are calculating results for

2\*N states and returning them directly in future recursive calls.

***Space Complexity :*** O(N), for recursive space

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/2213371/c-recursive-mmemoized-and-divide-and-conquor-methods-no-kadanes-algo/>

class Solution {

public:

    int f(vector<vector<int>>&dp,vector<int>& nums,int idx,bool takeit)

    {

        if(idx>=nums.size())

        {

            return takeit?0:-1e9;

        }

        if(dp[idx][takeit]!=-1)

        {

            return dp[idx][takeit];

        }

        //if taken a value previosuly, either return with that value, or take curr value again and recurse

        if(takeit)

        {

            return dp[idx][takeit]=max(0,nums[idx]+f(dp,nums,idx+1,true));

        }

        //if not taken any value before, take currvalue and recurse or not take curr value and recurse

        return dp[idx][takeit]=max(nums[idx]+f(dp,nums,idx+1,true) , f(dp,nums,idx+1,false));

    }

    int maxSubArray(vector<int>& nums) {

        vector<vector<int>>dp(nums.size(),vector<int>(2,-1));

        return f(dp,nums,0,false);

    }

};

**Solution 5: Divide and Conquer (360 min)**

**Style 1: Typical  1 (base case)-> 2 (递归成为更小的问题) -> 3 (进行当前层的处理计算)**

class Solution {

    public int maxSubArray(int[] nums) {

        return helper(nums, 0, nums.length - 1);

    }

    private int helper(int[] nums, int start, int end) {

        if(start > end) {

            return Integer.MIN\_VALUE;

        }

        // Divide

        int mid = start + (end - start) / 2;

        // leftMax = max subarray sum in [L, mid-1] and starting from mid-1

        int leftMax = helper(nums, start, mid - 1);

        // rightMax = max subarray sum in [mid+1, R] and starting from mid+1

        int rightMax = helper(nums, mid + 1, end);

        // Processing

        int curSum = 0;

        int leftSum = 0;

        int rightSum = 0;

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

            curSum += nums[i];

            leftSum = Math.max(leftSum, curSum);

        }

        curSum = 0;

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

            curSum += nums[i];

            rightSum = Math.max(rightSum, curSum);

        }

        // Conquer

        // return max of 3 cases

        return Math.max(leftSum + rightSum + nums[mid], Math.max(leftMax, rightMax));

    }

}

Time complexity: O(nlogn)

Space complexity: O(logn)

**Refer to (For Style 1)**

<https://leetcode.com/problems/maximum-subarray/solutions/20452/c-dp-and-divide-and-conquer/>The Divide-and-Conquer algorithm breaks

nums into two halves and find the maximum subarray sum in them recursively. Well, the most tricky part is to handle the case that the maximum subarray spans the two halves. For this case, we use a linear algorithm: starting from the middle element and move to both ends (left and right ends), record the maximum sum we have seen. In this case, the maximum sum is finally equal to the middle element plus the maximum sum of moving leftwards and the maximum sum of moving rightwards.

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        return maxSubArray(nums, 0, nums.size() - 1);

    }

private:

    int maxSubArray(vector<int>& nums, int l, int r) {

        if (l > r) {

            return INT\_MIN;

        }

        int m = l + (r - l) / 2, ml = 0, mr = 0;

        int lmax = maxSubArray(nums, l, m - 1);

        int rmax = maxSubArray(nums, m + 1, r);

        for (int i = m - 1, sum = 0; i >= l; i--) {

            sum += nums[i];

            ml = max(sum, ml);

        }

        for (int i = m + 1, sum = 0; i <= r; i++) {

            sum += nums[i];

            mr = max(sum, mr);

        }

        return max(max(lmax, rmax), ml + mr + nums[m]);

    }

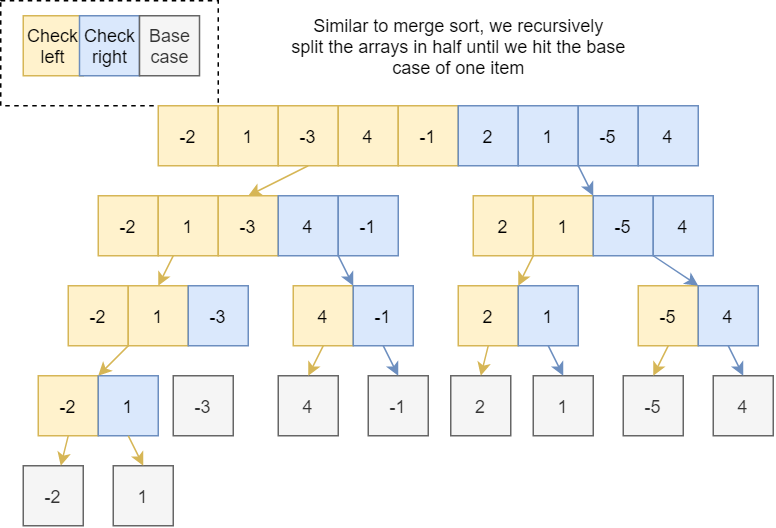
};

**Refer to (For Style 1)**

<https://leetcode.com/problems/maximum-subarray/solutions/1136682/javascript-divide-and-conquer-with-bonus-pictures/>

I understood the O(n) Kadane's algorithm fine, but had trouble wrapping my head around the divide and conquer approach. I drew out what was happening which helped me understand it and wanted to share.

The first part, the divide, looks a lot like diagrams of merge sort. We keep splitting the array into left and right until we get to our base case, a single item.

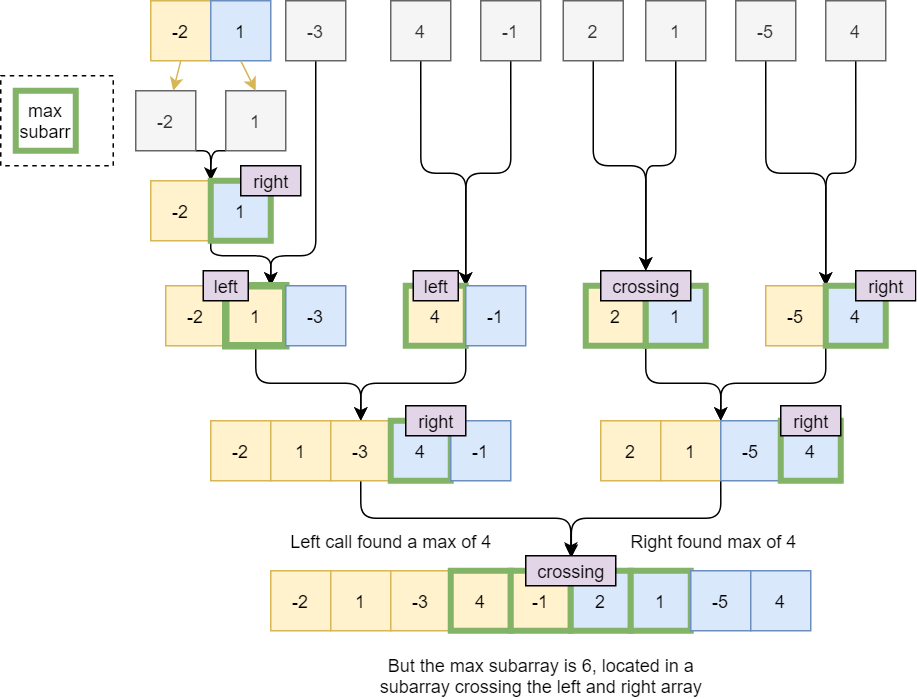


Now for conquering on the way up. At each division, we check is the max array in the left side or the right side? The special case is that the max array exists across the left and right side, so for each division we check from the middle out to find the crossing max sum.

An example of each case.

[1,-1] -> left side has max[-5, 2] -> right side has max[1,2] -> the max exists "crossing" both sides

And from the problems example, green show where the max array was found and the small text box shows when it was right, left or crossing.



And now pictures to code...

/\*\*

\* @param {number[]} nums

\* @return {number}

\*/

var maxSubArray = function(nums) {

    return findMax(nums, 0, nums.length - 1)

};

var findMax = function (nums, l, r) {

    if (l == r) {

        //Base case, return num here

        return nums[l]

    }

    let mid = Math.floor((l + r)/2)

    let leftSum = findMax(nums, l, mid) //Recursively check left side for max sum

    let rightSum = findMax(nums, mid+1, r) //recursively check right side for max sum

    const crossingSum = findCrossingSum(nums, l, mid, r) //Find max sum that includes left and right side

    return Math.max (crossingSum, leftSum, rightSum) //return whichever is largest

}

var findCrossingSum = function(nums, l, mid, r) {

    //A crossing sum exists on the left side and right side

    // so if we count mid as on the left side, the crossing sum MUST

    // include nums[mid] and nums[mid+1]. From these starting points we search for the max sum

    let sum=0

    let maxLSum = -Infinity

    for(let i = mid; i >= l; i--) {

        sum += nums[i]

        maxLSum = Math.max(maxLSum, sum)

    }

    sum =0

    let maxRSum = -Infinity

    for (let i = mid + 1; i <= r; i++) {

        sum += nums[i]

        maxRSum = Math.max(maxRSum, sum)

    }

    return maxRSum + maxLSum

}

**Style 2: Not typical 1 (base case)-> 2 (递归成为更小的问题) -> 3 (进行当前层的处理计算) but 1 -> 3 -> 2 more like recursion**

class Solution {

    public int maxSubArray(int[] nums) {

        return helper(nums, 0, nums.length - 1);

    }

    private int helper(int[] nums, int start, int end) {

        if(start > end) {

            return Integer.MIN\_VALUE;

        }

        int mid = start + (end - start) / 2;

        int leftSum = 0;

        int rightSum = 0;

        int curSum = 0;

        // leftSum = max subarray sum in [L, mid-1] and starting from mid-1

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

            curSum += nums[i];

            leftSum = Math.max(leftSum, curSum);

        }

        curSum = 0;

        // rightSum = max subarray sum in [mid+1, R] and starting from mid+1

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

            curSum += nums[i];

            rightSum = Math.max(rightSum, curSum);

        }

        // return max of 3 cases

        return Math.max(leftSum + rightSum + nums[mid], Math.max(helper(nums, start, mid - 1), helper(nums, mid + 1, end)));

    }

}

Time complexity: O(nlogn)

Space complexity: O(logn)

**Refer to (For Style 2)**

<https://leetcode.com/problems/maximum-subarray/solutions/1595195/c-python-7-simple-solutions-w-explanation-brute-force-dp-kadane-divide-conquer/>

✔️ ***Solution - VI (Divide & Conquer)***

We can solve this using divide and conquer strategy. This is the follow-up question asked in this question. This involves modelling the problem by observing that the maximum subarray sum is such that it lies somewhere -

entirely in the left-half of array [L, mid-1], OR

entirely in the right-half of array [mid+1, R], OR

in array consisting of mid element along with some part of left-half and some part of right-half such that these form contiguous subarray - [L', R'] = [L', mid-1] + [mid] + [mid+1,R'], where L' >= L and R' <= R

With the above observation, we can recursively divide the array into sub-problems on the left and right halves and then combine these results on the way back up to find the maximum subarray sum.

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        return maxSubArray(nums, 0, size(nums)-1);

    }

    int maxSubArray(vector<int>& A, int L, int R){

        if(L > R) return INT\_MIN;

        int mid = (L + R) / 2, leftSum = 0, rightSum = 0;

        // leftSum = max subarray sum in [L, mid-1] and starting from mid-1

        for(int i = mid-1, curSum = 0; i >= L; i--)

            curSum += A[i],

            leftSum=max(leftSum, curSum);

        // rightSum = max subarray sum in [mid+1, R] and starting from mid+1

        for(int i = mid+1, curSum = 0; i <= R; i++)

            curSum += A[i],

            rightSum = max(rightSum, curSum);

// return max of 3 cases

        return max({ maxSubArray(A, L, mid-1), maxSubArray(A, mid+1, R), leftSum + A[mid] + rightSum });

    }

};

***Time Complexity :*** O(NlogN), the recurrence relation can be written as T(N) = 2T(N/2) + O(N) since we are recurring for left and right half (2T(N/2)) and also calculating maximal subarray including mid element which takes O(N) to calculate. Solving this recurrence using master theorem, we can get the time complexity as O(NlogN)

***Space Complexity :*** O(logN), required by recursive stack

**Solution 6: Divide and Conquer Optimization with pre sum (60 min, hard logic, not able to self-reproduce)**

class Solution {

    int[] pre;

    int[] suf;

    public int maxSubArray(int[] nums) {

        pre = nums;

        suf = nums;

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

            pre[i] += Math.max(0, pre[i - 1]);

        }

        for(int i = nums.length - 2; i >= 0; i--) {

            suf[i] -= Math.max(0, suf[i + 1]);

        }

        return helper(nums, 0, nums.length - 1);

    }

    private int helper(int[] nums, int start, int end) {

        if(start == end) {

            return nums[start];

        }

        // Divide

        int mid = start + (end - start) / 2;

        // leftMax = max subarray sum in [L, mid-1] and starting from mid-1

        int leftMax = helper(nums, start, mid);

        // rightMax = max subarray sum in [mid+1, R] and starting from mid+1

        int rightMax = helper(nums, mid + 1, end);

        // Conquer

        // return max of 3 cases

        return Math.max(pre[mid] + suf[mid + 1], Math.max(leftMax, rightMax));

    }

}

Time complexity: O(n)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/1595195/c-python-7-simple-solutions-w-explanation-brute-force-dp-kadane-divide-conquer/>

✔️ ***Solution - VII (Optimized Divide & Conquer)***

We can further optimize the previous solution. The O(N) term in the recurrence relation of previous solution was due to computation of max sum subarray involving nums[mid] in each recursion.

But we can reduce that term to O(1) if we precompute it. This can be done by precomputing two arrays pre and suf where pre[i] will denote maximum sum subarray ending at i and suf[i] denotes the maximum subarray starting at i. pre is similar to dp array that we computed in dynamic programming solutions and suf can be calculated in similar way, just by starting iteration from the end.

class Solution {

public:

    vector<int> pre, suf;

    int maxSubArray(vector<int>& nums) {

        pre = suf = nums;

        for(int i = 1; i < size(nums); i++)  pre[i] += max(0, pre[i-1]);

        for(int i = size(nums)-2; ~i; i--)  suf[i] += max(0, suf[i+1]);

        return maxSubArray(nums, 0, size(nums)-1);

    }

    int maxSubArray(vector<int>& A, int L, int R){

        if(L == R) return A[L];

        int mid = (L + R) / 2;

        return max({ maxSubArray(A, L, mid), maxSubArray(A, mid+1, R), pre[mid] + suf[mid+1] });

    }

};

***Time Complexity :*** O(N), the recurrence relation can be written as T(N) = 2T(N/2) + O(1) since we are recurring for left and right half (2T(N/2)) and calculating maximal subarray including mid element in O(1). Solving this recurrence using master theorem, we can get the time complexity as O(N)

***Space Complexity :*** O(N), required by suf and pre.

💡 **Note**:

The above divide and conquer solution works in O(N) but is **once you have calculated pre and suf, does it even make sense to go into divide and conquer approach?** I don't think divide and conquer approach after calculating pre & sufis useful, unless you Really want to solve it using **❝Divide and Conquer❞ only**. You can instead do the following (which is similar to dp)-

class Solution {

public:

int maxSubArray(vector<int>& nums) {

vector<int> pre = nums;

for(int i = 1; i < size(nums); i++) pre[i] += max(0, pre[i-1]);

return \*max\_element(begin(pre), end(pre));

}

};

**Solution 5: Dynamic Programming (60 min)**

**Style 1: 2D DP**

class Solution {

    public int maxSubArray(int[] nums) {

        int[][] dp = new int[2][nums.length];

        // dp[1][i] denotes maximum subarray sum ending at i (including nums[i])

        // dp[0][i] denotes maximum subarray sum upto i (may or may not include nums[i])

        dp[1][0] = nums[0];

        dp[0][0] = dp[1][0];

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

            // At each index, we update dp[1][i] as max between either only

            // choosing current element - nums[i] or extending from previous

            // subarray and choosing current element as well - dp[1][i-1] + nums[i]

            dp[1][i] = Math.max(nums[i], dp[1][i - 1] + nums[i]);

            // Similarly, dp[0][i] can be updated as max between maximum sum

            // subarray found till last index - dp[0][i-1] or max subarray

            // sum found ending at current index dp[1][i]

            dp[0][i] = Math.max(dp[0][i - 1], dp[1][i]);

        }

        return dp[0][nums.length - 1];

    }

}

Time complexity: O(n)

Space complexity: O(n)

**Style 2: 1D DP**

class Solution {

    // dp[i] denotes maximum subarray sum ending at i (including nums[i])

    // maxSubArray(int nums[], int i), which means the maxSubArray for nums[0:i] which must has A[i] as the end element.

    // maxSubArray(nums, i) = maxSubArray(nums, i - 1) > 0 ? maxSubArray(nums, i - 1) : 0 + nums[i];

    public int maxSubArray(int[] nums) {

        int[] dp = new int[nums.length];

        dp[0] = nums[0];

        int max = dp[0];

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

            dp[i] = (dp[i - 1] > 0 ? dp[i - 1] : 0) + nums[i];

            max = Math.max(max, dp[i]);

        }

        return max;

    }

}

Time complexity: O(n)

Space complexity: O(n)

**Style 3: 1D DP with O(1) space**

class Solution {

    public int maxSubArray(int[] nums) {

        // Replace dp[i] with 'sum'

        int sum = nums[0];

        int max = nums[0];

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

            //if(sum < 0) {

            //    sum = nums[i];

            //} else {

            //    sum += nums[i];

            //}

            sum = sum < 0 ? nums[i] : sum + nums[i];

            max = Math.max(max, sum);

        }

        return max;

    }

}

Time complexity: O(n)

Space complexity: O(1)

**Refer to (for 2D-DP)**

<https://leetcode.com/problems/maximum-subarray/solutions/1595195/c-python-7-simple-solutions-w-explanation-brute-force-dp-kadane-divide-conquer/>

✔️ ***Solution - IV (Dynamic Programming - Tabulation)***

We can employ similar logic in iterative version as well. Here, we again use dp array and use bottom-up approach. Here dp[1][i] denotes maximum subarray sum ending at i (including nums[i]) and dp[0][i] denotes maximum subarray sum up to i (may or may not include nums[i]).

At each index, we update dp[1][i] as max between either only choosing current element - nums[i] or extending from previous subarray and choosing current element as well - dp[1][i-1] + nums[i]

Similarly, dp[0][i] can be updated as max between maximum sum subarray found till last index - dp[0][i-1] or max subarray sum found ending at current index dp[1][i].

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        vector<vector<int>> dp(2, vector<int>(size(nums)));

        dp[0][0] = dp[1][0] = nums[0];

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

            dp[1][i] = max(nums[i], nums[i] + dp[1][i-1]);

            dp[0][i] = max(dp[0][i-1], dp[1][i]);

        }

        return dp[0].back();

    }

};

We can actually do away with just 1 row as well. We denoted dp[1][i] as the maximum subarray sum ending at i. We can just store that row and calculate the overall maximum subarray sum at the end by choosing the maximum of all max subarray sum ending at i.

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        vector<int> dp(nums);

        for(int i = 1; i < size(nums); i++)

            dp[i] = max(nums[i], nums[i] + dp[i-1]);

        return \*max\_element(begin(dp), end(dp));

    }

};

***Time Complexity :*** O(N), we are just iterating over the nums array once to compute the dp array and once more over the dp at the end to find maximum subarray sum. Thus overall time complexity is O(N) + O(N) = O(N)

***Space Complexity :*** O(N), for maintaining dp.

**Refer to (for 1D-DP)**

<https://leetcode.com/problems/maximum-subarray/solutions/20193/dp-solution-some-thoughts/>

Analysis of this problem:

Apparently, this is a optimization problem, which can be usually solved by DP. **So when it comes to DP, the first thing for us to figure out is the format of the sub problem(or the state of each sub problem). The format of the sub problem can be helpful when we are trying to come up with the recursive relation.**

**At first, I think the sub problem should look like: maxSubArray(int A[], int i, int j), which means the maxSubArray for A[i: j]. In this way, our goal is to figure out what maxSubArray(A, 0, A.length - 1) is. However, if we define the format of the sub problem in this way, it's hard to find the connection from the sub problem to the original problem(at least for me). In other words, I can't find a way to divide the original problem into the sub problems and use the solutions of the sub problems to somehow create the solution of the original one.**

**So I change the format of the sub problem into something like: maxSubArray(int A[], int i), which means the maxSubArray for A[0:i ] which must has A[i] as the end element.** Note that now the sub problem's format is less flexible and less powerful than the previous one because **there's a limitation that A[i] should be contained in that sequence and we have to keep track of each solution of the sub problem to update the global optimal value.** **However, now the connect between the sub problem & the original one becomes clearer:**

**maxSubArray(A, i) = maxSubArray(A, i - 1) > 0 ? maxSubArray(A, i - 1) : 0 + A[i];**

And here's the code

public int maxSubArray(int[] A) {

        int n = A.length;

        int[] dp = new int[n];//dp[i] means the maximum subarray ending with A[i];

        dp[0] = A[0];

        int max = dp[0];

        for(int i = 1; i < n; i++){

            dp[i] = A[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);

            max = Math.max(max, dp[i]);

        }

        return max;

}

**Refer to (for 1D-DP O(1) space)**

<https://leetcode.com/problems/maximum-subarray/solutions/20193/dp-solution-some-thoughts/comments/20474>

My DP reasoning is as follows:

To calculate sum(0,i), you have 2 choices: either adding sum(0,i-1) to a[i], or not. If sum(0,i-1) is negative, adding it to a[i] will only make a smaller sum, so we add only if it's non-negative.

sum(0,i) = a[i] + (sum(0,i-1) < 0 ? 0 : sum(0,i-1))

We can then use O(1) space to keep track of the max sum(0, i) so far.

public int maxSubArray(int[] nums) {

if (nums == null || nums.length == 0) { return 0; }

int max = nums[0], sum = nums[0];

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

if (sum < 0) { sum = nums[i]; }

else {sum += nums[i]; }

max = Math.max(max, sum);

}

return max;

}

**Solution 6: Kadane's Algorithm (10 min)**

**Same as 1D DP O(1) space solution**

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/1595195/c-python-7-simple-solutions-w-explanation-brute-force-dp-kadane-divide-conquer/>

✔️ ***Solution - V (Kadane's Algorithm)***

We can observe that in the previous approach, dp[i] only depended on dp[i-1]. So do we really need to maintain the whole dp array of N elements? One might see the last line of previous solution and say that we needed all elements of dp at the end to find the maximum sum subarray. But we can simply optimize that by storing the max at each iteration instead of separately calculating it at the end.

Thus, we only need to maintain curMax which is the maximum subarray sum ending at i and maxTillNow which is the maximum sum we have seen till now. And this way of solving this problem is what we popularly know as **Kadane's Algorithm**

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int curMax = 0, maxTillNow = INT\_MIN;

        for(auto c : nums)

            curMax = max(c, curMax + c),

            maxTillNow = max(maxTillNow, curMax);

        return maxTillNow;

    }

};

***Time Complexity :*** O(N), for iterating over nums once

***Space Complexity :*** O(1), only constant extra space is being used.

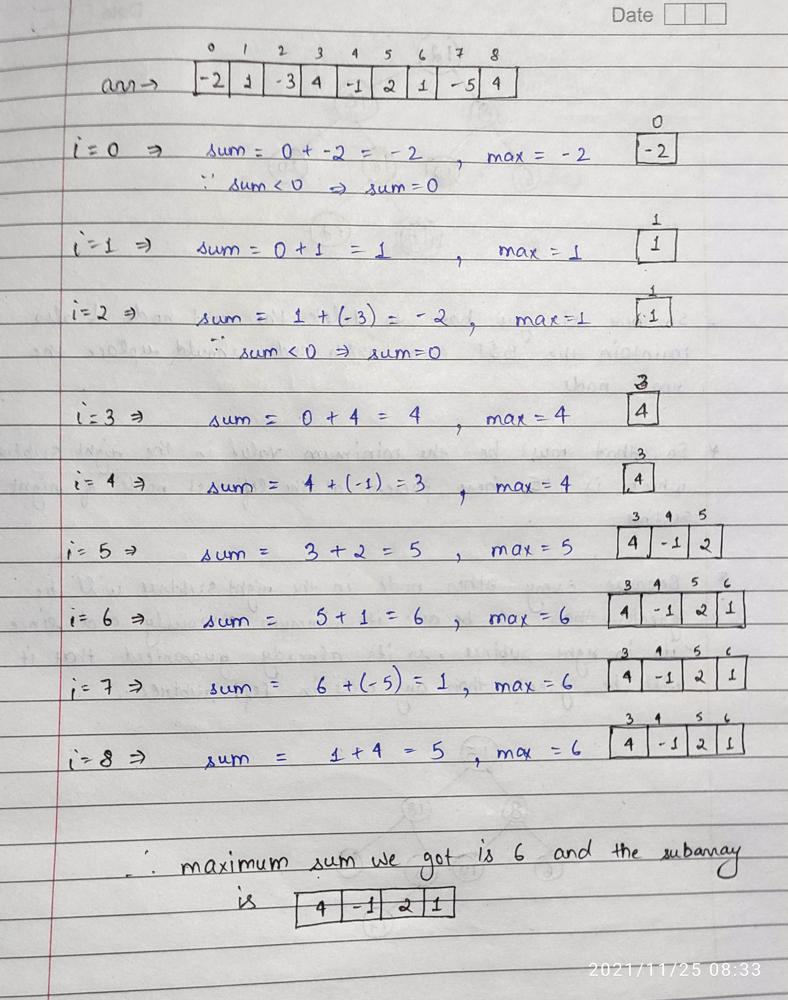
*PS:* I have tried running Solution - IV and this solution multiple times and this solution, despite being O(1) shows higher memory usage(28.8 MB on average vs 28 MB used by solution - IV) by LC consistently. If anyone can figure out the reason behind this, I will be happy to hear it out🙂

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/1595097/java-kadane-s-algorithm-explanation-using-image/>

**Intution:** Start traversing your array keep each element in the sum and every time keep the max of currSum and prevSum.

**But the catch here is that if at any point sum becomes negative then no point keeping it because 0 is obviously greater than negative, so just make your sum 0.**



class Solution {

    public int maxSubArray(int[] nums) {

        int n = nums.length;

        int max = Integer.MIN\_VALUE, sum = 0;

        for(int i=0;i<n;i++){

            sum += nums[i];

            max = Math.max(sum,max);

            if(sum<0) sum = 0;

        }

        return max;

    }

}

**Why Leetcode 53 cannot use Sliding Window ?**

**Refer to chatGPT**

**1. Sliding Window Requires a Fixed or Adjustable Condition**

The sliding window technique is effective when there's a clear condition to expand or shrink the window (e.g., "sum ≤ target", "window size = k").

In Leetcode 53, the problem is to find the maximum sum of a **contiguous subarray**. However, there isn't a fixed condition to decide when to shrink the window because **a negative sum may still lead to a better overall result later**. This dynamic nature prevents using a simple rule to shrink or expand the window.

**2. Negative Numbers Can Reset the Optimal Subarray**

If part of the subarray results in a negative sum, including it in the current window might decrease the total sum. For example:

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

If the sliding window tried to extend through all positive numbers, it wouldn't know when to "reset" after encountering negatives like -3 or -5.

This behavior makes **Kadane's Algorithm** more suitable, as it tracks the local maximum (subarray ending at the current position) and resets when the local sum becomes negative.

**3. Sliding Window Doesn't Track History**

Sliding window focuses on the **current window** and adjusts it to meet specific conditions. It doesn't inherently remember past windows or results.

In Leetcode 53, you need to **remember the maximum sum encountered so far**, which isn't naturally handled by a sliding window. Kadane's Algorithm achieves this by maintaining:

currentSum: The maximum sum of the subarray ending at the current position.

maxSum: The global maximum sum found so far.

**Comparison of Approaches**

|  |  |  |
| --- | --- | --- |
| Feature | Sliding Window | Kadane's Algorithm |
| Works with dynamic conditions? | Yes (e.g., sum <= target, window size) | No (tracks cumulative subarray sums) |
| Handles negatives dynamically? | No | Yes |
| Tracks historical results? | No | Yes |
| Time Complexity | O(n) | O(n) |
| Space Complexity | O(1) | O(1) |

**Correct Solution Using Kadane's Algorithm**

Here’s the correct implementation for Leetcode 53:

public int maxSubArray(int[] nums) {

int maxSum = nums[0];

int currentSum = nums[0];

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

// Update currentSum to include nums[i] or start a new subarray

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

// Update maxSum to track the highest sum so far

maxSum = Math.max(maxSum, currentSum);

}

return maxSum;

}

**Key Differences from Sliding Window**

Dynamic Inclusion: Kadane’s Algorithm decides whether to include the current element in the ongoing subarray or start fresh based on currentSum + nums[i].

No Fixed Window: The size of the subarray is not predefined or adjustable based on a condition; it dynamically evolves.

Global Tracking: The algorithm keeps track of the global maximum sum (maxSum) throughout the process.

**Follow up:**

**Now here in this question you can see that you can also be asked some more things like :**

Length of the max subarray

Elements of the max subarray

Start and End index of max subarray

**This is very important concept from interview point so try to get the ans of above mentioned point and have confidence on this algorithm.**

class Solution {

    public int maxSubArray(int[] nums) {

        // Replace dp[i] with 'sum'

        int sum = nums[0];

        int max = nums[0];

        // The final start and end position of the maximum sum subarray

        int start = 0;

        int end = 0;

        // The temporary start position

        int s = 0;

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

            //sum = sum < 0 ? nums[i] : sum + nums[i];

            if(sum < 0) {

                // Have to reset 'start' with temporary 's'

                s = i;

                sum = nums[i];

            } else {

                // Find a larger subarray

                start = s;

                end = i;

                sum += nums[i];

            }

            max = Math.max(max, sum);

        }

        System.out.println("Maximum subarray get at start index = " + start + " end index = " + end);

        return max;

    }

}

=====================================================================================================

No need the temporary start position 's'

class Solution {

    public int maxSubArray(int[] nums) {

        // Replace dp[i] with 'sum'

        int sum = nums[0];

        int max = nums[0];

        // The final start and end position of the maximum sum subarray

        int start = 0;

        int end = 0;

        // The temporary start position

        //int s = 0;

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

            //sum = sum < 0 ? nums[i] : sum + nums[i];

            if(sum < 0) {

                // Have to reset 'start' with temporary 's'

                //s = i;

                start = i;

                sum = nums[i];

            } else {

                // Find a larger subarray

                //start = s;

                end = i;

                sum += nums[i];

            }

            max = Math.max(max, sum);

        }

        System.out.println("Maximum subarray get at start index = " + start + " end index = " + end);

        return max;

    }

}

**Refer to**

<https://leetcode.com/problems/maximum-subarray/solutions/1595097/java-kadane-s-algorithm-explanation-using-image/comments/1195076>

//And also print that subarray

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int n = nums.size();

        int maxi = nums[0];

        int start = 0, end = 0; // the final start and end position of the maximum sum subarray

        int sum = 0;

        int s = 0, // the temporary start position

        for(int i=0;i<n;i++){

            sum += nums[i];

            if(sum > maxi){

                maxi = sum;

                start = s;

                end = i;

            }

            if(sum < 0){

                sum = 0;

                s = i+1;

            }

        }

        return maxi;

    }

};

**Refer to**

[L821.Shortest Distance to a Character (Ref.L845)](note://WEB790653af463124a332a0f62fb0d60ff3)

[L918.Maximum Sum Circular Subarray (Ref.L1658)](note://WEB1495ae8db0086f5ab30de99c6528274c)