**Coin Change**

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

**Example 1:**

**Input:** coins = [1,2,5], amount = 11

**Output:** 3

**Explanation:** 11 = 5 + 5 + 1

**Example 2:**

**Input:** coins = [2], amount = 3

**Output:** -1

**Example 3:**

**Input:** coins = [1], amount = 0

**Output:** 0

**Constraints:**

* 1 <= coins.length <= 12
* 1 <= coins[i] <= 231 - 1
* 0 <= amount <= 104

/\*\*

\* @param {number[]} coins

\* @param {number} amount

\* @return {number}

\*/

var coinChange = function(coins, amount) {

};

Solution Article

Approach #1 (Brute force) [Time Limit Exceeded]

**Intuition**

The problem could be modeled as the following optimization problem : \min\_{x} \sum\_{i=0}^{n - 1} x\_i \\ \text{subject to} \sum\_{i=0}^{n - 1} x\_i\*c\_i = Smin*x*​∑*i*=0*n*−1​*xi*​subject to∑*i*=0*n*−1​*xi*​∗*ci*​=*S*

, where S*S* is the amount, c\_i*ci*​ is the coin denominations, x\_i*xi*​ is the number of coins with denominations c\_i*ci*​ used in change of amount S*S*. We could easily see that x\_i = [{0, \frac{S}{c\_i}}]*xi*​=[0,*ci*​*S*​].

A trivial solution is to enumerate all subsets of coin frequencies [x\_0\dots x\_{n - 1}][*x*0​…*xn*−1​] that satisfy the constraints above, compute their sums and return the minimum among them.

**Algorithm**

To apply this idea, the algorithm uses backtracking technique to generate all combinations of coin frequencies [x\_0\dots x\_{n-1}][*x*0​…*xn*−1​] in the range ([{0, \frac{S}{c\_i}}])([0,*ci*​*S*​]) which satisfy the constraints above. It makes a sum of the combinations and returns their minimum or -1−1 in case there is no acceptable combination.

public class Solution {

public int coinChange(int[] coins, int amount) {

return coinChange(0, coins, amount);

}

private int coinChange(int idxCoin, int[] coins, int amount) {

if (amount == 0)

return 0;

if (idxCoin < coins.length && amount > 0) {

int maxVal = amount/coins[idxCoin];

int minCost = Integer.MAX\_VALUE;

for (int x = 0; x <= maxVal; x++) {

if (amount >= x \* coins[idxCoin]) {

int res = coinChange(idxCoin + 1, coins, amount - x \* coins[idxCoin]);

if (res != -1)

minCost = Math.min(minCost, res + x);

}

}

return (minCost == Integer.MAX\_VALUE)? -1: minCost;

}

return -1;

}

}

// Time Limit Exceeded

**Complexity Analysis**

* Time complexity : O(S^n)*O*(*Sn*). In the worst case, complexity is exponential in the number of the coins n*n*. The reason is that every coin denomination c\_i*ci*​ could have at most \frac{S}{c\_i}*ci*​*S*​ values. Therefore the number of possible combinations is :

\frac{S}{c\_1}\*\frac{S}{c\_2}\*\frac{S}{c\_3}\ldots\frac{S}{c\_n} = \frac{S^{n}}{{c\_1}\*{c\_2}\*{c\_3}\ldots{c\_n}}*c*1​*S*​∗*c*2​*S*​∗*c*3​*S*​…*cn*​*S*​=*c*1​∗*c*2​∗*c*3​…*cn*​*Sn*​

* Space complexity : O(n)*O*(*n*). In the worst case the maximum depth of recursion is n*n*. Therefore we need O( n)*O*(*n*) space used by the system recursive stack.

#### Approach #2 (Dynamic programming - Top down) [Accepted]

**Intuition**

Could we improve the exponential solution above? Definitely! The problem could be solved with polynomial time using Dynamic programming technique. First, let's define:

F(S)*F*(*S*) - minimum number of coins needed to make change for amount S*S* using coin denominations [{c\_0\ldots c\_{n-1}}][*c*0​…*cn*−1​]

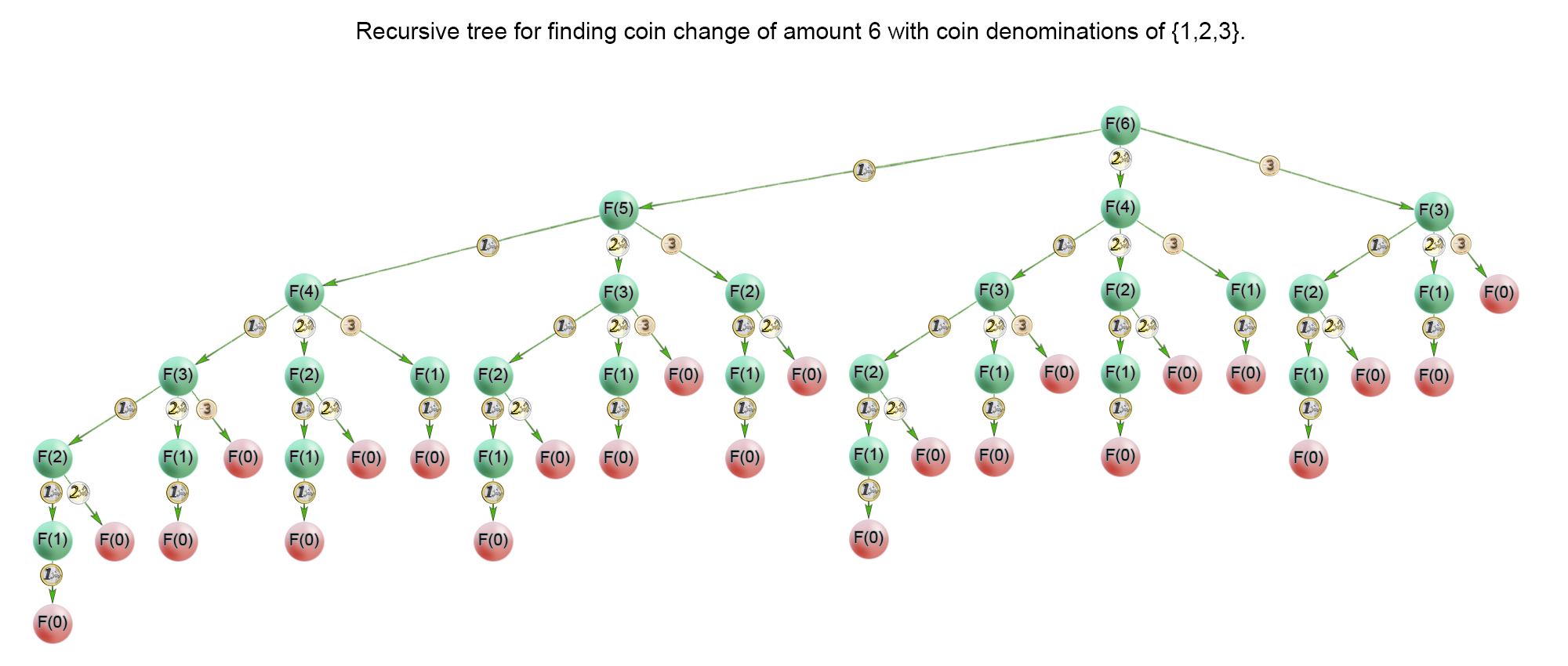
We note that this problem has an optimal substructure property, which is the key piece in solving any Dynamic Programming problems. In other words, the optimal solution can be constructed from optimal solutions of its subproblems. How to split the problem into subproblems? Let's assume that we know F(S)*F*(*S*) where some change val\_1, val\_2, \ldots*val*1​,*val*2​,… for S*S* which is optimal and the last coin's denomination is C*C*. Then the following equation should be true because of optimal substructure of the problem:

F(S) = F(S - C) + 1*F*(*S*)=*F*(*S*−*C*)+1

But we don't know which is the denomination of the last coin C*C*. We compute F(S - c\_i)*F*(*S*−*ci*​) for each possible denomination c\_0, c\_1, c\_2 \ldots c\_{n -1}*c*0​,*c*1​,*c*2​…*cn*−1​ and choose the minimum among them. The following recurrence relation holds:

F(S) = \min\_{i=0 ... n-1} { F(S - c\_i) } + 1 \\ \text{subject to} \ S-c\_i \geq 0 \\*F*(*S*)=min*i*=0...*n*−1​*F*(*S*−*ci*​)+1subject to *S*−*ci*​≥0

F(S) = 0 , \text{when} S = 0 \\ F(S) = -1 , \text{when} n = 0*F*(*S*)=0,when*S*=0*F*(*S*)=−1,when*n*=0



In the recursion tree above, we could see that a lot of subproblems were calculated multiple times. For example the problem F(1)*F*(1) was calculated 1313 times. Therefore we should cache the solutions to the subproblems in a table and access them in constant time when necessary

**Algorithm**

The idea of the algorithm is to build the solution of the problem from top to bottom. It applies the idea described above. It use backtracking and cut the partial solutions in the recursive tree, which doesn't lead to a viable solution. Тhis happens when we try to make a change of a coin with a value greater than the amount *SS*. To improve time complexity we should store the solutions of the already calculated subproblems in a table.

public class Solution {

public int coinChange(int[] coins, int amount) {

if (amount < 1) return 0;

return coinChange(coins, amount, new int[amount]);

}

private int coinChange(int[] coins, int rem, int[] count) {

if (rem < 0) return -1;

if (rem == 0) return 0;

if (count[rem - 1] != 0) return count[rem - 1];

int min = Integer.MAX\_VALUE;

for (int coin : coins) {

int res = coinChange(coins, rem - coin, count);

if (res >= 0 && res < min)

min = 1 + res;

}

count[rem - 1] = (min == Integer.MAX\_VALUE) ? -1 : min;

return count[rem - 1];

}

}

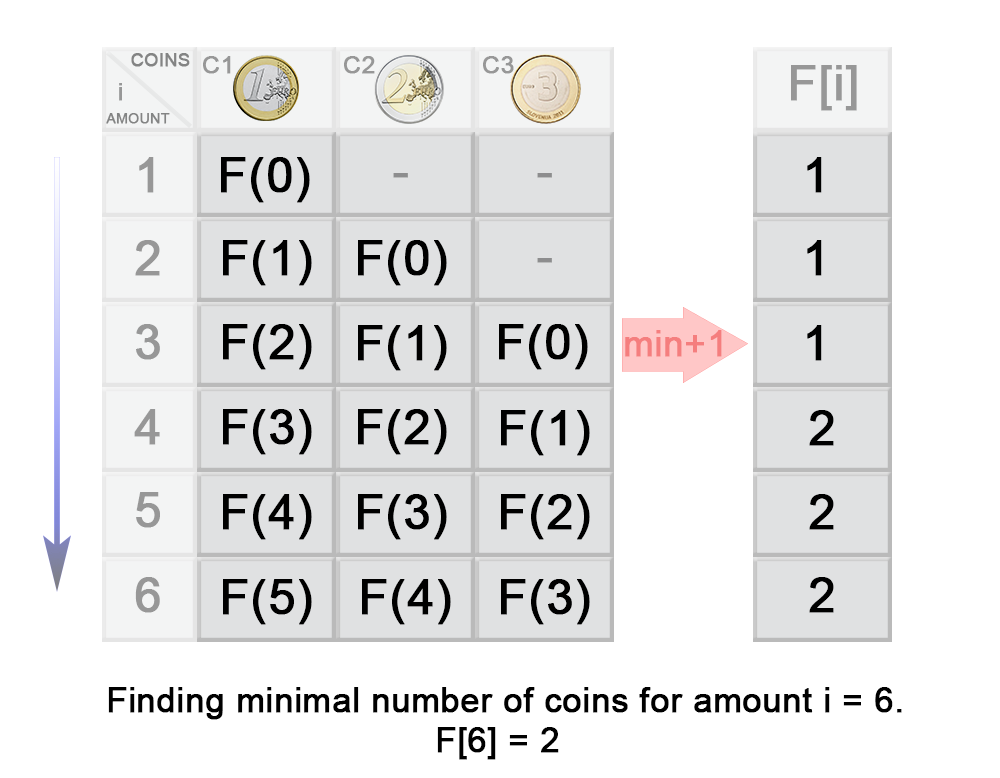
**Complexity Analysis**

* Time complexity : O(S\*n)*O*(*S*∗*n*). where S is the amount, n is denomination count. In the worst case the recursive tree of the algorithm has height of S*S* and the algorithm solves only S*S* subproblems because it caches precalculated solutions in a table. Each subproblem is computed with n*n* iterations, one by coin denomination. Therefore there is O(S\*n)*O*(*S*∗*n*) time complexity.
* Space complexity : O(S)*O*(*S*), where S*S* is the amount to change We use extra space for the memoization table.

Approach #3 (Dynamic programming - Bottom up) [Accepted]

**Algorithm**

For the iterative solution, we think in bottom-up manner. Before calculating *F(i)F(i)*, we have to compute all minimum counts for amounts up to i*i*. On each iteration i*i* of the algorithm *F(i)F(i)* is computed as \min\_{j=0 \ldots n-1}{F(i -c\_j)} + 1min*j*=0…*n*−1​*F*(*i*−*cj*​)+1



In the example above you can see that:

\begin{aligned} F(3) &= \min\{{F(3- c\_1), F(3-c\_2), F(3-c\_3)}\} + 1 \\ &= \min\{{F(3- 1), F(3-2), F(3-3)}\} + 1 \\ &= \min\{{F(2), F(1), F(0)}\} + 1 \\ &= \min\{{1, 1, 0}\} + 1 \\ &= 1 \end{aligned}*F*(3)​=min{*F*(3−*c*1​),*F*(3−*c*2​),*F*(3−*c*3​)}+1=min{*F*(3−1),*F*(3−2),*F*(3−3)}+1=min{*F*(2),*F*(1),*F*(0)}+1=min{1,1,0}+1=1​

public class Solution {

public int coinChange(int[] coins, int amount) {

int max = amount + 1;

int[] dp = new int[amount + 1];

Arrays.fill(dp, max);

dp[0] = 0;

for (int i = 1; i <= amount; i++) {

for (int j = 0; j < coins.length; j++) {

if (coins[j] <= i) {

dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);

}

}

}

return dp[amount] > amount ? -1 : dp[amount];

}

}

**Complexity Analysis**

* Time complexity : O(S\*n)*O*(*S*∗*n*). On each step the algorithm finds the next *F(i)F(i)* in n*n* iterations, where 1\leq i \leq S1≤*i*≤*S*. Therefore in total the iterations are S\*n*S*∗*n*.
* Space complexity : O(S)*O*(*S*). We use extra space for the memoization table.

Solution

Overview

We have an array of n*n* non-negative integers which we must split into m*m* subarrays. The goal is to split it in such a way that the largest sum of a subarray among these m*m* subarrays is minimized.

While dividing the array, we can observe that for each integer, there are two options: either add it to the current subarray or start a new subarray with it (as long as the number of subarrays does not exceed m*m*). The maximum number of possible combinations is {n - 1} \choose {m - 1}(*m*−1*n*−1​) (because we must split the array at m - 1*m*−1 positions to obtain m*m* subarrays, and there are n - 1*n*−1 positions where the array can be split). The brute force approach is to enumerate every possible combination and select the combination with the smallest maximum sum subarray. However, given the problem constraints, the worst-case scenario will have {999} \choose {49}(49999​) combinations, which is extraordinarily large. So let's try to find a better-optimized approach.

There are two characteristics of this problem that we should take note of at this time. First, as we iterate over each element, we must decide whether to add the element to the current subarray or to start a new subarray. This decision will depend on the number of subarrays we have already made. In other words, each decision we make is affected by the previous decisions we have made. Second, the problem is asking to **minimize** the largest sum of subarrays. These are two common characteristics of dynamic programming problems, and as such we will first approach this problem using dynamic programming.

**Note:** If you arrived at this conclusion before reading this article, then you have done well! The clues in the problem description hint that we should consider using dynamic programming to solve this problem. However, what makes this problem especially tricky is that the optimal solution does not use dynamic programming! This speaks to the importance of taking a moment to consider other possible approaches, even after arriving at the first possible solution. Take a moment to try to come up with another viable approach now, and we will discuss the optimal approach last.

Approach 1: Top-Down Dynamic Programming

**Intuition**

Let's start with the first subarray, which can have a range like [0, i]. We need to decide the value of index i. Once we have decided the value of i, we can find the sum for the first subarray sum[0, i]. Then the problem simplifies to finding the maximum value out of sum[0, i] and the largest sum subarray for the array in the range [i + 1, n - 1] with m - 1*m*−1 subarrays. This way we are able to divide the problem into smaller subproblems.

How do we find the optimal value of i? Let's try with every possible value of i. We can try every possible first subarray and then recursively solve the problem for the remaining array. Let's define our recurrence relation F[currIndex, subarrayCount] to be the minimum largest subarray sum for the array [currIndex, n - 1] with subarrayCount subarrays, we can write it in the following way:

F[currIndex, subarrayCount] = min( max(sum[currIndex, i], F[i + 1, subarrayCount - 1]) ), For all i in range [currIndex, n - subarrayCount]

Note that the function F shown here, represents getMinimumLargestSplitSum in the code provided below.

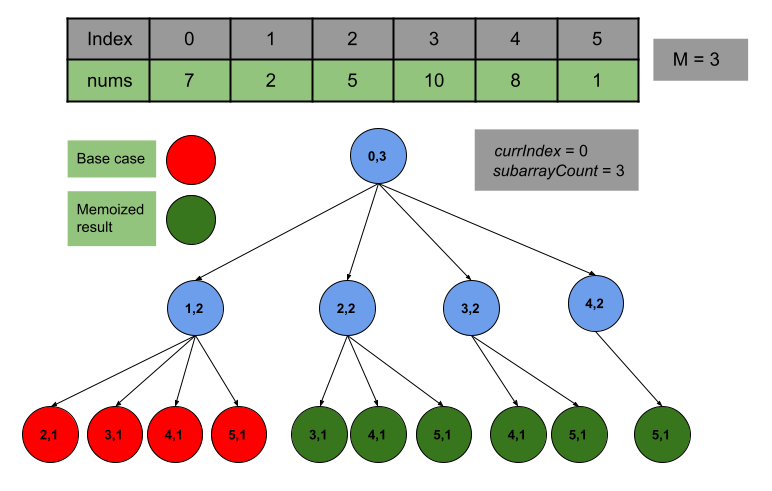
Let's break down the expression max(sum[currIndex, i], F[i + 1, subarrayCount - 1]) so that we understand it better.

* F[i + 1, subarrayCount - 1] represents the smallest possible largest sum subarray in the range [i + 1, n - 1] with subarrayCount - 1 subarrays.
* sum[currIndex, i] represents the sum of the current subarray spanning the range of [currIndex, i].
* We take the max of these two values, the expression as a whole represents the largest sum subarray in the range [currIndex, n - 1] with subarrayCount subarrays, when we make the first split at index i.
* To find the optimal place to split the array, we calculate this for all i in the range [currIndex, n - subarrayCount] and take the minimum value as the result.

Also, it's worth noting a small optimization that we just did here by deciding the range for i as [currIndex, n - subarrayCount] instead of [currIndex, n - 1] this is because we need to divide the array into exactly m subarrays. Hence, if i goes beyond n - subarrayCount we won't be able to make m subarrays even if we only place a single element in each of the remaining subarrays.

If we can calculate the result for the subproblem without using the above recurrence relation (a base case) then we can simply return the result instead of making further recursive calls. In this problem, when there is only one subarray remaining, all of the numbers remaining must go in that subarray, so when subarrayCount is 1, we can simply return the sum of the numbers between currIndex and the end of the array.

This recursive approach will have repeated subproblems; this can be observed in the figure below. Notice the green nodes are repeat subproblems signifying that we have already solved these subproblems before.



To avoid spending time recalculating the results for previously seen subproblems, the first time we calculate the minimum largest sum for a certain range and number of subarrays, we will store the value in an array. Then, the next time we need to calculate the result for this same range and subarray count we can look up the result in constant time. This technique is known as memoization and it helps us avoid recalculating repeated subproblems.

**Algorithm**

1. Fill the array prefixSum. The i^{th}*ith* index of prefixSum will have the sum of integers in nums in the range [0, i - 1] with prefix[0] = 0. (We need prefixSum because each time we reach a base case, we must return the sum of the remaining elements, and a prefix sum array allows us to do this in constant time.)
2. Start with index currIndex as 0 and the number of subarrays subarrayCount as m, this represents the subarray with range [0, n - 1] and m subarrays.
3. Select which elements will go in the current subarray by traversing over the indices starting from currIndex to N - subarrayCount. At each index:
   * Use prefixSum to find the sum of the elements in the current subarray (firstSplitSum).
   * Recursively call getMinimumLargestSplitSum to find the minimum largest subarray that can be obtained from the remaining elements.
   * The maximum of these two values (largestSplitSum) will be the largest subarray sum **if** the first subarray is [currIndex, i].
   * Repeat this process for all i up to n - subarrayCount, then store the minimum possible largestSplitSum in minimumLargestSplitSum.
4. Return minimumLargestSplitSum; to avoid repeat calculations, also store it in the memoization table memo corresponding to currIndex and subarrayCount.
5. Base case: If subarrayCount is 1, then we know that all of the remaining numbers **must** go in the current subarray. So instead of making recursive calls according to step 3, when subarrayCount is 1, we can simply return the sum of numbers between currIndex and the end of the array.

**Note:** In the for loop for traversing i, once firstSplitSum is larger than minimumLargestSplitSum it is impossible for us to find a better minimumLargestSplitSum because firstSplitSum will only continue to increase (because all numbers are non-negative) and so does the largestSplitSum. So we can prune our search by breaking when firstSplitSum becomes greater than or equal to the minimumLargestSplitSum.

**Implementation**

class Solution {

// Defined it as per the maximum size of array and split count

// But can be defined with the input size as well

Integer[][] memo = new Integer[1001][51];

private int getMinimumLargestSplitSum(int[] prefixSum, int currIndex, int subarrayCount) {

int n = prefixSum.length - 1;

// We have already calculated the answer so no need to go into recursion

if (memo[currIndex][subarrayCount] != null) {

return memo[currIndex][subarrayCount];

}

// Base Case: If there is only one subarray left, then all of the remaining numbers

// must go in the current subarray. So return the sum of the remaining numbers.

if (subarrayCount == 1) {

return memo[currIndex][subarrayCount] = prefixSum[n] - prefixSum[currIndex];

}

// Otherwise, use the recurrence relation to determine the minimum largest subarray

// sum between currIndex and the end of the array with subarrayCount subarrays remaining.

int minimumLargestSplitSum = Integer.MAX\_VALUE;

for (int i = currIndex; i <= n - subarrayCount; i++) {

// Store the sum of the first subarray.

int firstSplitSum = prefixSum[i + 1] - prefixSum[currIndex];

// Find the maximum subarray sum for the current first split.

int largestSplitSum = Math.max(firstSplitSum,

getMinimumLargestSplitSum(prefixSum, i + 1, subarrayCount - 1));

// Find the minimum among all possible combinations.

minimumLargestSplitSum = Math.min(minimumLargestSplitSum, largestSplitSum);

if (firstSplitSum >= minimumLargestSplitSum) {

break;

}

}

return memo[currIndex][subarrayCount] = minimumLargestSplitSum;

}

public int splitArray(int[] nums, int m) {

// Store the prefix sum of nums array.

int n = nums.length;

int[] prefixSum = new int[n + 1];

for (int i = 0; i < n; i++) {

prefixSum[i + 1] = prefixSum[i] + nums[i];

}

return getMinimumLargestSplitSum(prefixSum, 0, m);

}

}

**Complexity Analysis**

Here N*N* is the length of the array and M*M* is the number of subarrays allowed.

* Time complexity: O(N^2 \cdot M)*O*(*N*2⋅*M*)

Each state is defined by the values currIndex and subarrayCount. As such, there are N \cdot M*N*⋅*M* possible states, and we must visit almost every state in order to solve the original problem. Each state (subproblem) requires O(N)*O*(*N*) time because we have a for loop from currIndex to N - subarrayCount. Thus the total time complexity is equal to O(N^2 \cdot M)*O*(*N*2⋅*M*).

* Space complexity: O(N \cdot M)*O*(*N*⋅*M*)

The memoization results are stored in the table memo with size N \cdot M*N*⋅*M*. Also, stack space in the recursion is equal to the maximum number of active functions. The maximum number of active functions can be equal to M*M* as we only make a recursive call when splitting the array and we do not make a recursive call when there is only 11 subarray remaining. Hence, the space complexity is equal to O(N \cdot M)*O*(*N*⋅*M*).

Approach 2: Bottom-Up Dynamic Programming

**Intuition**

In the previous approach, the recursive calls incurred stack space. This can be avoided by applying the same approach iteratively which is generally faster than the top-down approach. We will follow a similar approach as the previous one just in a reverse manner.

In this approach, we will use the same recurrence relation and the base case as we used in the previous approach. So the code will remain largely the same as in the previous approach. The difference lies in the way of filling the memoization table memo. In the top-down implementation, we visited each subproblem by making recursive calls until we reached a base case. Whereas in the bottom-up implementation, we will use two for loops to iterate over every subproblem (currIndex, subarrayCount), starting from the base case (subarrayCount = 1). We will iterate subarrayCount from 1 to m, and for each value, we will find the largest minimum subarray sum for every range [currIndex, n - 1], where currIndex varies from 0 to n - 1.

**Algorithm**

1. Fill the array prefixSum. The i^{th}*ith* index of prefixSum will have the sum of integers in nums in the range [0, i - 1] with prefix[0] = 0.
2. Initialize an array memo where memo[currIndex][subarrayCount] will store the result for the subproblem (currIndex, subarrayCount).
3. We need to find the value of memo[0][m] which represents the minimum largest subarray sum starting at index 0 with m subarrays. But we only know what the result will be for the base cases. To fill the memo array, we will iterate subarrayCount over the range [1, m] (starting at 1 because that is our base case) and iterate currIndex over the range [0, n - 1].
4. For each value of subarrayCount and currIndex, we will update memo[subarrayCount][currIndex]:
   * As the sum of the elements between currIndex and the end of the array if we are at a base case (subarrayCount equals 1).
   * Otherwise, we will use the recurrence relation and the results from previously solved subproblems to calculate memo[subarrayCount][currIndex].
5. Return the value stored at memo[0][m].

**Implementation**

class Solution {

// Defined it as per the maximum size of array and split count

// But can be defined with the input size as well

int[][] memo = new int[1001][51];

public int splitArray(int[] nums, int m) {

int n = nums.length;

// Store the prefix sum of nums array

int[] prefixSum = new int[n + 1];

for (int i = 0; i < n; i++) {

prefixSum[i + 1] = prefixSum[i] + nums[i];

}

for (int subarrayCount = 1; subarrayCount <= m; subarrayCount++) {

for (int currIndex = 0; currIndex < n; currIndex++) {

// Base Case: If there is only one subarray left, then all of the remaining numbers

// must go in the current subarray. So return the sum of the remaining numbers.

if (subarrayCount == 1) {

memo[currIndex][subarrayCount] = prefixSum[n] - prefixSum[currIndex];

continue;

}

// Otherwise, use the recurrence relation to determine the minimum largest subarray

// sum between currIndex and the end of the array with subarrayCount subarray remaining.

int minimumLargestSplitSum = Integer.MAX\_VALUE;

for (int i = currIndex; i <= n - subarrayCount; i++) {

// Store the sum of the first subarray.

int firstSplitSum = prefixSum[i + 1] - prefixSum[currIndex];

// Find the maximum subarray sum for the current first split.

int largestSplitSum = Math.max(firstSplitSum, memo[i + 1][subarrayCount - 1]);

// Find the minimum among all possible combinations.

minimumLargestSplitSum = Math.min(minimumLargestSplitSum, largestSplitSum);

if (firstSplitSum >= minimumLargestSplitSum) {

break;

}

}

memo[currIndex][subarrayCount] = minimumLargestSplitSum;

}

}

return memo[0][m];

}

}

**Complexity Analysis**

Here N*N* is the length of the array and M*M* is the number of subarrays allowed.

* Time complexity: O(N^2 \cdot M)*O*(*N*2⋅*M*)

Each state is defined by the values currIndex and subarrayCount. As such, there are N \cdot M*N*⋅*M* possible states, and we must visit almost every state in order to solve the original problem. Each state (subproblem) requires O(N)*O*(*N*) time because we have a for loop from currIndex to N - subarrayCount; because we have stored the result in the table memo, any repeated calls will only require constant time. Thus the total time complexity is equal to O(N^2 \cdot M)*O*(*N*2⋅*M*).

* Space complexity: O(N \cdot M)*O*(*N*⋅*M*)

The results are stored in the table memo with size N \cdot M*N*⋅*M*. Hence, the space complexity is equal to O(N \cdot M)*O*(*N*⋅*M*).

Approach 3: Binary Search

**Intuition**

This approach is quite different from the previous approaches. The characteristics of this problem are a good fit for the dynamic programming solution hence, it's easy to overlook the idea of using binary search. This problem satisfies the property that we can guess the answer (the minimum largest sum subarray value) and check if that value was too high or too low, thus narrowing our search space. It requires a different perspective to think of this approach, but after realizing that this problem is a candidate for a binary search solution, it becomes easier to implement than the previous approaches.

The goal of this problem is to find the minimum largest subarray sum with m*m* subarrays. Instead of finding the answer directly, what if we try to guess the answer (say X*X*), and check whether this particular value could be the largest subarray sum with m*m* subarrays. If this is possible, we can check all values for X \geq 0*X*≥0, and the first value that satisfies the condition will be the answer. Thus, by repeatedly solving the following problem, we can find the minimum largest subarray sum needed to split nums into m*m* subarrays:

Given an array of n*n* integers and a value X*X*, determine the minimum number of subarrays the array needs to be divided into such that no subarray sum is greater than X*X*.

If the minimum number of subarrays required is less than or equal to m*m* then the value X*X* could be the largest subarray sum.

The solution to this newly defined problem is as follows

* First, make sure X*X* is greater than or equal to the maximum element in the array. Otherwise, no solution would be possible because we cannot divide an element.
* Start from 0th index and keep adding elements to sum only if adding the element does not make sum greater than X*X*.
* If adding the current element would make the sum greater than X*X* then we have to split the subarray here. So we will increment the number of splits required (splitsRequired) and set sum to the value of current element (signifying that the current subarray only contains the current element).
* Once we traversed the whole array, return splitsRequired + 1. This is the minimum number of subarrays required.

Now the problem with the current solution is that the value of X*X* can be as large as the sum of integers in the given array. Hence, checking for all values of X*X* is not feasible.

The key observation here is that if we are able to split the array into m*m* or fewer subarrays for a value X*X* then we will also be able to do it for any value greater than X*X*. This is because the number of subarrays would be even less in case of any value greater than X*X*. Similarly, if it's not possible for a value X*X* it will not be possible to have m*m* or fewer splits for any value less than X*X*.

The following slideshow demonstrates this algorithm:

1 / 8

Therefore, instead of searching linearly for X*X*, we can do a binary search! If we can split the array into m*m* or fewer subarrays all with a sum that is less than or equal to X*X* then we will try a smaller value for X*X* otherwise we will try a larger value for X*X*. Each time we will select X*X* so that we reduce the size of the search space by half.

**Algorithm**

1. Store the sum of elements in nums in the variable sum and the maximum element of the array in maxElement.
2. Initialize the boundary for binary search. The minimum value for the largest subarray sum is maxElement and the maximum value is equal to sum. Hence assign left and right to maxElement and sum respectively.
3. Then while the left is not greater than right:

a. Find the mid-value in range [left, right], this is equal to the maximum subarray sum allowed. Store it in maxSumAllowed.

b. Find the minimum number of subarrays required to have the largest subarray sum equal to maxSumAllowed.

* + If the number of subarrays is less than or equal to m*m* then assign maxSumAllowed as the answer minimumLargestSplitSum and decrease the value of our right boundary.
  + If the number of subarrays is more than m*m* this can't be the answer hence, increase the value of our left boundary.

1. Return minimumLargestSplitSum.

**Implementation**

class Solution {

private int minimumSubarraysRequired(int[] nums, int maxSumAllowed) {

int currentSum = 0;

int splitsRequired = 0;

for (int element : nums) {

// Add element only if the sum doesn't exceed maxSumAllowed

if (currentSum + element <= maxSumAllowed) {

currentSum += element;

} else {

// If the element addition makes sum more than maxSumAllowed

// Increment the splits required and reset sum

currentSum = element;

splitsRequired++;

}

}

// Return the number of subarrays, which is the number of splits + 1

return splitsRequired + 1;

}

public int splitArray(int[] nums, int m) {

// Find the sum of all elements and the maximum element

int sum = 0;

int maxElement = Integer.MIN\_VALUE;

for (int element : nums) {

sum += element;

maxElement = Math.max(maxElement, element);

}

// Define the left and right boundary of binary search

int left = maxElement;

int right = sum;

int minimumLargestSplitSum = 0;

while (left <= right) {

// Find the mid value

int maxSumAllowed = left + (right - left) / 2;

// Find the minimum splits. If splitsRequired is less than

// or equal to m move towards left i.e., smaller values

if (minimumSubarraysRequired(nums, maxSumAllowed) <= m) {

right = maxSumAllowed - 1;

minimumLargestSplitSum = maxSumAllowed;

} else {

// Move towards right if splitsRequired is more than m

left = maxSumAllowed + 1;

}

}

return minimumLargestSplitSum;

}

}

**Complexity Analysis**

Here N*N* is the length of the array and S*S* is the sum of integers in the array.

* Time complexity: O(N \cdot \log(S))*O*(*N*⋅log(*S*))

The total number of iterations in the binary search is \log(S)log(*S*), and for each such iteration, we call minimumSubarraysRequired which takes O(N)*O*(*N*) time. Hence, the time complexity is equal to O(N \cdot \log(S))*O*(*N*⋅log(*S*)).

* Space complexity: O(1)*O*(1)

We do not use any data structures that require more than constant extra space.