<https://leetcode.com/problems/perfect-squares/description/>

Given an integer n, return *the least number of perfect square numbers that sum to* n.

A **perfect square** is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.

**Example 1:**

Input: n = 12

Output: 3

Explanation: 12 = 4 + 4 + 4.

**Example 2:**

Input: n = 13

Output: 2

Explanation: 13 = 4 + 9.

**Constraints:**

1 <= n <= 10^4

**Attempt 1: 2023-10-10**

**Solution 1: Native DFS (10 min, TLE 57/588)**

class Solution {

public int numSquares(int n) {

return helper(n);

}

private int helper(int cur) {

// Base case 1:

// When our call hits base case and n becomes 0 we know

// there is a possible way to split our target.

// Since in ques we have to find the min possible way.

// There is no case where mini becomes less than 0.

// So we return 0 that indicates we found a Possible

// Set Of Candidates

if(cur == 0) {

return 0;

}

// Base case 2:

// Same logic as above if our n becomes less than 0 we

// know there is no possible way to split our target,

// we return max integer

if(cur < 0) {

return Integer.MAX\_VALUE;

}

// WHY mini count = n ?

// Suppose we are at a point in our recursion call

// where n = 12 ! what could be the max count to split

// 12 so that each element is a perfect square ?

// YES u are right: 12 = 1 + 1 + 1 + 1 + 1 + 1 + 1 +

// 1 + 1 + 1 + 1 + 1 (12 times 1)

// we cant get any maximum value than 12 for 12 that's

// why mini = n which is 12 in this case.

int count = cur;

for(int i = 1; i \* i <= cur; i++) {

count = Math.min(count, 1 + helper(cur - i \* i));

}

return count;

}

}

Time Complexity: O(sqrt(N)^N)

Space Complexity: O(N)

**Solution 2: DFS + Memoization (10 min)**

class Solution {

public int numSquares(int n) {

Map<Integer, Integer> memo = new HashMap<>();

return helper(n, memo);

}

private int helper(int cur, Map<Integer, Integer> memo) {

// Base case 1:

// When our call hits base case and n becomes 0 we know

// there is a possible way to split our target.

// Since in ques we have to find the min possible way.

// There is no case where mini becomes less than 0.

// So we return 0 that indicates we found a Possible

// Set Of Candidates

if(cur == 0) {

return 0;

}

// Base case 2:

// Same logic as above if our n becomes less than 0 we

// know there is no possible way to split our target,

// we return max integer

if(cur < 0) {

return Integer.MAX\_VALUE;

}

if(memo.containsKey(cur)) {

return memo.get(cur);

}

// WHY mini count = n ?

// Suppose we are at a point in our recursion call

// where n = 12 ! what could be the max count to split

// 12 so that each element is a perfect square ?

// YES u are right: 12 = 1 + 1 + 1 + 1 + 1 + 1 + 1 +

// 1 + 1 + 1 + 1 + 1 (12 times 1)

// we cant get any maximum value than 12 for 12 that's

// why mini = n which is 12 in this case.

int count = cur;

for(int i = 1; i \* i <= cur; i++) {

count = Math.min(count, 1 + helper(cur - i \* i, memo));

}

memo.put(cur, count);

return count;

}

}

Time Complexity: O(sqrt(N)^N)

Space Complexity: O(N)

First we have to pay attention on while loop. It takes O( sqrt(n) ). Next we have to pay attention on the recursive call min( mini, solve( n- ( i \* i ))). In case when our i == 1 we are able to create the longest way. Because of that we can come to a conclusion that the maximum number of levels in a decision tree will be n. Taking all of this into consideration and knowing the width of a tree level and the height of the whole tree we can say that the overall TC of brute recusion is O( sqrt(n) ^ n - 1 ) .

**Refer to**

<https://leetcode.com/problems/perfect-squares/solutions/1513258/very-easy-to-understand-with-picture-python-recursion-memoization/>

Hey Everyone I will try to explain the solution through some pictures. **How each piece of code is working**!!!Was going through DISCUSS Section but couldn't wrap my head around why certain lines were written, so after figuring out I tried to share it out here to save someone's else time. This question has been tagged under **AMAZON** in some places...LET'S BEGIN.

**RECURSIVE CODE** [ TLE ]

def solve(n):

if n==0: # part 1

return 0

if n<0: # part 2

return float("inf")

mini = n # part 3

i = 1

while i\*i<=n: # part 4

mini = min(mini, solve(n-(i\*i)))

i+=1

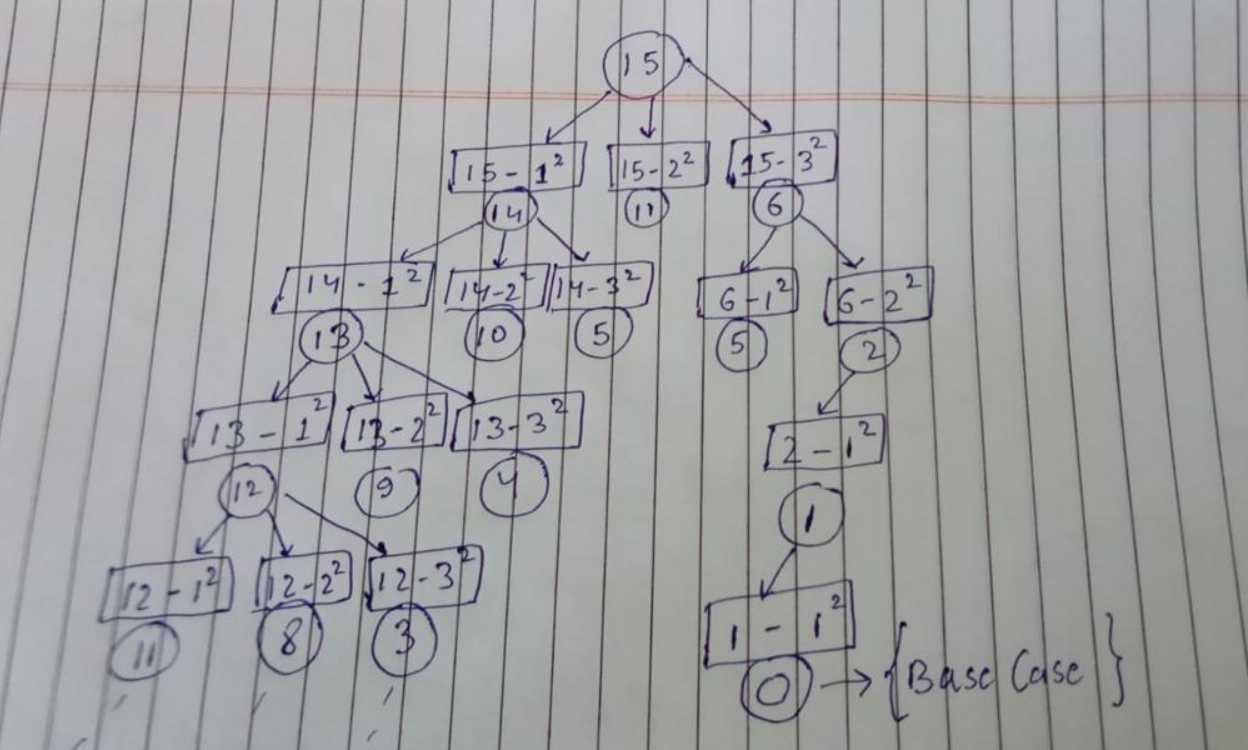
return mini+1 # part 5

solve(n)

**( Time Complexity ) - First we have to pay attention on while loop. It takes O( sqrt(n) ). Next we have to pay attention on the recursive call min( mini, solve( n- ( i \* i ))). In case when our i == 1 we are able to create the longest way. Because of that we can come to a conclusion that the maximum number of levels in a decision tree will be n. Taking all of this into consideration and knowing the width of a tree level and the height of the whole tree we can say that the overall TC of brute recusion is O( sqrt(n) ^ n - 1 ) .**

**Credit** - [Maxim](https://leetcode.com/_Maximus_/)

Let's First Have A Look Over How Our DECISION TREE Looks Like !!



--> **PART 1 [ BASE CASE ]**

When our call hits base case and n becomes 0 we know there is a possible way to split our target. Since in ques we have to find the **min** possible way. There is no case where mini becomes less than 0 . So we return 0 that indicates we found a **Possible Set Of Candidates.**

--> **PART 2**

Same logic as above if our n becomes less than 0 we know there is no possible way to split our target.

--> **PART 3 [ WHY mini = n ?? THIS PART WAS HARD FOR ME TO UNDERSTAND :( ]**

Suppose we are at a point in our recursion call where n = 12 !!what could be the max count to split 12 so that each element is a perfect square??

YES u are right :) **12 = 1 + 1+ 1+ 1+1+ 1+ 1+1 +1+1+1 +1 ( 12 times 1 )**we cant get any maximum value than 12 for 12 thats why mini = n which is 12 in this case.

--> **PART 4 [ WHILE LOOP PART ]**

What is ***least possible perfect Square candidate*** for our ans??YES It is 1 so we start our loop from i=1

Now upto what value of i our loop should execute???YES till n >= i^2 ( in short upto that value when n - i^2 is +ve )

WHY??because after that we will call at value < 0 which anyhow our function will **return infinity** since n<0 **remember base case PART 2. but will lead to infinite looping**

--> **PART 5 [ WHY mini+1?? ]**

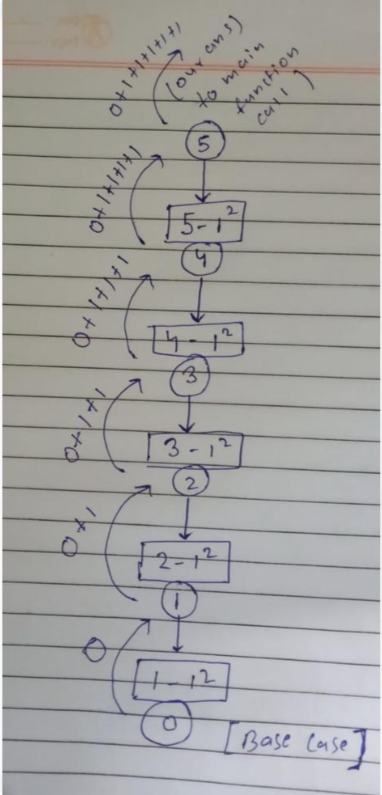
We are counting the steps isn't it??

When we hit the base case n==0 we return **0**.

Also one path for 5 = 1+1+1+1+1 .**Last 1 will get 0** (BASE CASE) and **it will be returning 0+1**

so step count = 0 + 1 + 1 + 1 + 1 + 1 = 5

see the picture for better understanding



**MEMOIZATION [ ACCEPTED ]**

def solve(n):

if n==0:

return 0

if n<0:

return float("inf")

if memo[n]!=-1:

return memo[n]

mini = n

i = 1

while i\*i<=n:

mini = min(mini, solve(n-(i\*i)))

i+=1

memo[n] = mini+1

return memo[n]

memo = [-1]\*(n+1)

solve(n)

**Solution 3: DP (10 min)**

class Solution {

public int numSquares(int n) {

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

// 完美对应DFS中初始化所有数n均只有n个1构成

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

dp[i] = i;

}

// "顶"在n，"底"在0，直接从底到顶

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

for(int j = 1; j \* j <= i; j++) {

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

}

}

return dp[n];

}

}

Time Complexity: N \* sqrt(N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.wang/leetcode-279-Perfect-Squares.html>

# **解法二 动态规划**

**理解了解法一的话，很容易改写成动态规划。递归相当于先压栈压栈然后出栈出栈，动态规划可以省去压栈的过程。**

**动态规划的转移方程就对应递归的过程，动态规划的初始条件就对应递归的出口。**

public int numSquares(int n) {

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

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

//依次求出 1, 2... 直到 n 的解

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

//依次减去一个平方数

for (int j = 1; j \* j <= i; j++) {

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

}

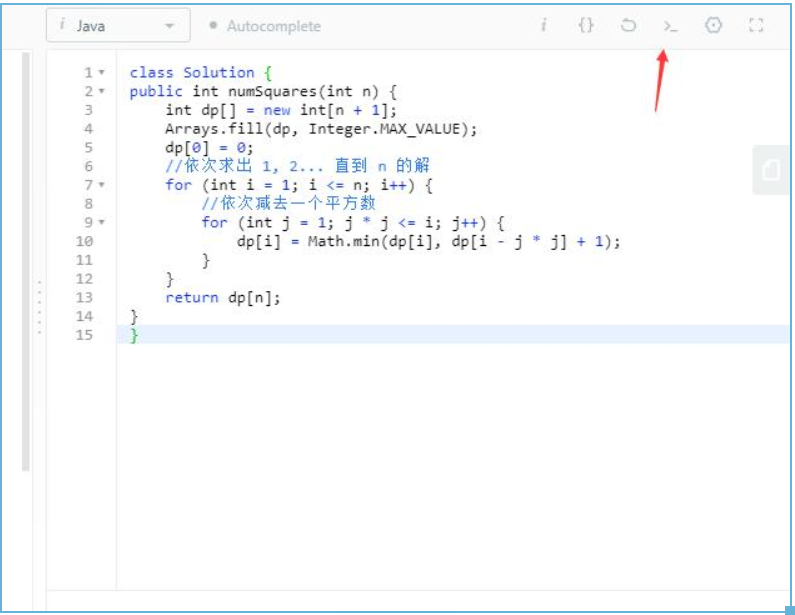
}

return dp[n];

}

这里还提出来一种 Static Dynamic Programming，主要考虑到测试数据有多组，看一下 leetcode 全部代码的逻辑。

点击下图箭头的位置。



然后会看到下边的代码。

class Solution {

public int numSquares(int n) {

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

Arrays.fill(dp,Integer.MAX\_VALUE);

dp[0] = 0;

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

for (int j = 1; j \* j <= i; j++) {

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

}

}

return dp[n];

}

}

public class MainClass {

public static void main(String[] args) throws IOException {

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

String line;

while ((line = in.readLine()) != null) {

int n = Integer.parseInt(line);

int ret = new Solution().numSquares(n);

String out = String.valueOf(ret);

System.out.print(out);

}

}

}

可以看到上边的逻辑，每次求 n 的时候都是创建新的对象然后调用方法。

这样会带来一个问题，假如第一次我们求了 90 的平方数和的最小个数，期间 dp 会求出 1 到 89 的所有的平方数和的最小个数。

第二次如果我们求 50 的平方数和的最小个数，其实第一次我们已经求过了，但实际上我们依旧会求一遍 1 到 50 的所有平方数和的最小个数。

我们可以通过声明 dp 是 static 变量，这样每次调用就不会重复计算了。所有对象将共享 dp

static ArrayList<Integer> dp = new ArrayList<>();

public int numSquares(int n) {

//第一次进入将 0 加入

if(dp.size() == 0){

dp.add(0);

}

//之前是否计算过 n

if(dp.size() <= n){

//接着之前最后一个值开始计算

for (int i = dp.size(); i <= n; i++) {

int min = Integer.MAX\_VALUE;

for (int j = 1; j \* j <= i; j++) {

min = Math.min(min, dp.get(i - j \* j) + 1);

}

dp.add(min);

}

}

return dp.get(n);

}

**Solution 4: BFS (10 min)**

class Solution {

public int numSquares(int n) {

// Use a set to record checked index to avoid repeated work.

// This is the key to reduce the running time to O(N^2).

Set<Integer> visited = new HashSet<>();

Queue<Integer> q = new LinkedList<>();

q.offer(n);

visited.add(n);

int level = 1;

while(!q.isEmpty()) {

int size = q.size();

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

int cur = q.poll();

for(int j = 1; j \* j <= cur; j++) {

int next = cur - j \* j;

// Filter out already visited index

if(!visited.contains(next)) {

if(next == 0) {

return level;

}

q.offer(next);

visited.add(next);

}

}

}

level++;

}

return n;

}

}

Time Complexity: O(sqrt(n) \* n)

Space Complexity: O(k^(l-1)) (maximum number of nodes at a certain level in a queue), where k = sqrt(n) and l = level

**Refer to**

<https://leetcode.wang/leetcode-279-Perfect-Squares.html>

# **解法三 BFS**

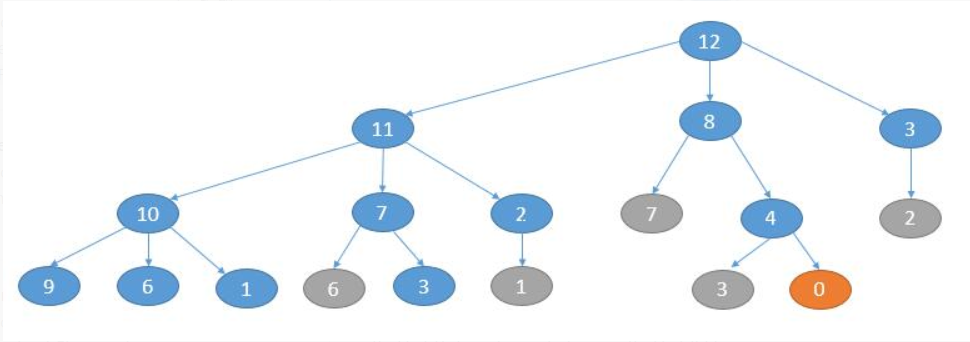
参考 [这里](https://leetcode.com/problems/perfect-squares/discuss/71488/Summary-of-4-different-solutions-(BFS-DP-static-DP-and-mathematics))。

相对于解法一的 DFS ，当然也可以使用 BFS 。

DFS 是一直做减法，然后一直减一直减，直到减到 0 算作找到一个解。属于一个解一个解的寻找。

BFS 的话，我们可以一层一层的算。第一层依次减去一个平方数得到第二层，第二层依次减去一个平方数得到第三层。直到某一层出现了 0，此时的层数就是我们要找到平方数和的最小个数。

举个例子，n = 12，每层的话每个节点依次减 1, 4, 9...。如下图，灰色表示当前层重复的节点，不需要处理。



如上图，当出现 0 的时候遍历就可以停止，此时是第 3 层（从 0 计数），所以最终答案就是 3。

**实现的话当然离不开队列，此外我们需要一个 set 来记录重复的解。**

public int numSquares(int n) {

Queue<Integer> queue = new LinkedList<>();

HashSet<Integer> visited = new HashSet<>();

int level = 0;

queue.add(n);

while (!queue.isEmpty()) {

int size = queue.size();

level++; // 开始生成下一层

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

int cur = queue.poll();

//依次减 1, 4, 9... 生成下一层的节点

for (int j = 1; j \* j <= cur; j++) {

int next = cur - j \* j;

if (next == 0) {

return level;

}

if (!visited.contains(next)) {

queue.offer(next);

visited.add(next);

}

}

}

}

return -1;

}

**Refer to**

<https://leetcode.com/problems/perfect-squares/solutions/1704124/bfs-using-java-easy-visual-explanation/>

Key Idea with an example (below screenshot):

Top-Down and Left-Right (BFS on k-ary tree) approach.

Each input number N is considered a node, the number of children for each node = [number of perfect squares upto N]

The children are the subtracted values from the parent. Each child's value = (parentValue - perfectSquare)

At each level, the queue holds the current level nodes. (Only unique values are pushed into the queue)

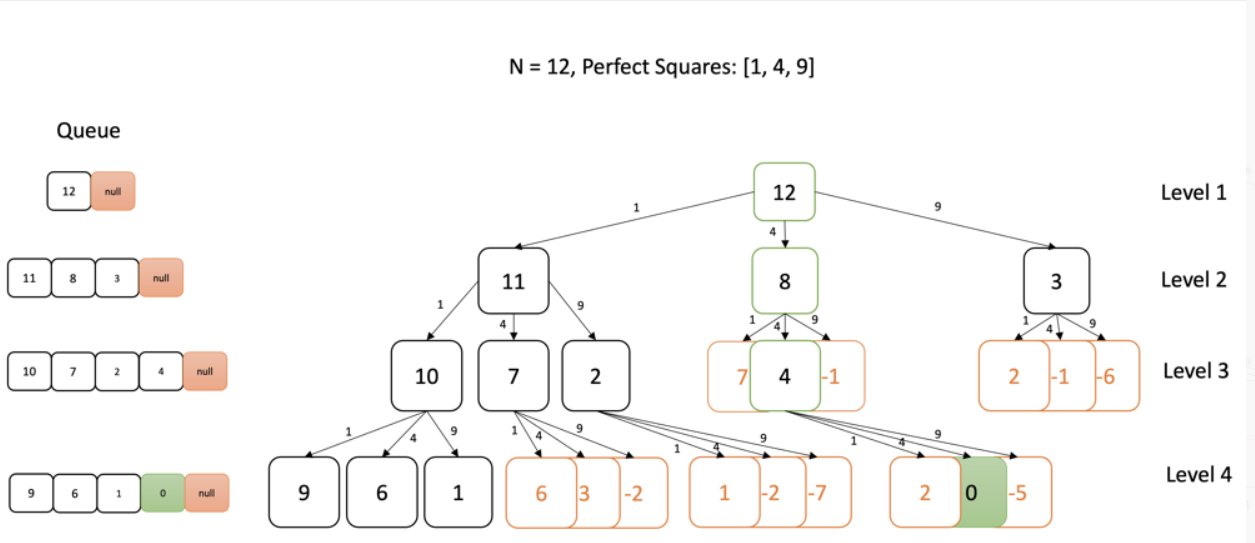
From implementation perspective, you can append null at the end of every level.

During this course, at any point of time you find a node with value 0 - you have found the perfect sum, return the parent's level (or current level based on how you implement - in my case, I update the level after parsing all the nodes of that *current* level).

The levels are ultimately how many numbers are involved in the perfect square sum.

Else at some point of time, the queue will be left with null after parsing through all possible unique values (<= N)

The below screenshot is a visual representation:



**Note:** The orange nodes indicate duplicate or negative values which are invalid. The green is the actual path that led to the required sum.

**Time Complexity:** O(sqrt(n) \* n)

**Space Complexity:** O(k^(l-1)) (maximum number of nodes at a certain level in a queue), where k = sqrt(n) and l = level

class Solution {

public int numSquares(int n) {

int level = 1;

//Perfect squares uptil n - O(sqrt(n))

List<Integer> perfectSquares = new LinkedList<>();

for(int num = 1; num \* num <= n; ++num)

perfectSquares.add(num \* num);

Set<Integer> visitedNumbers = new HashSet<>();

Queue<Integer> queue = new LinkedList<>();

queue.offer(n);

queue.offer(null);

// O(sqrt(n) \* n)

while(queue.size() > 1) {

Integer element = queue.poll();

if(element != null) {

for(Integer perfectSquare: perfectSquares) {

int remain = element - perfectSquare;

if(remain < 0)

break;

else if(remain == 0)

return level;

else if(visitedNumbers.add(remain))

queue.offer(remain);

}

} else { // parsed all the nodes in the current level. Update the variable to current level

level++;

queue.offer(null);

}

}

return 0;

}

}