# Question

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 <= 104

# Solution

#### **Approach 1: Brute-force Enumeration [Time Limit Exceeded]**

**Intuition**

The problem asks us to find the **least** numbers of square numbers that can sum up to a given number. We could rephrase the problem as follows:

Given a list of square numbers and a positive integer number n, one is asked to find a combination of square numbers that sum up to n, and the combination should contain the **least** numbers among all possible solutions.  
Note: one could reuse the square numbers in the combination.

From the above narrative of the problem, it seems to be a combination problem, to which an intuitive solution would be the brute-force enumeration where we enumerate all possible combinations and find the minimal one of them.

We could formulate the problem in the following formula:



From the above formula, one can translate it into a **recursive** solution literally. Here is one example.

|  |
| --- |
| class Solution(object):  def numSquares(self, n):  square\_nums = [i\*\*2 for i in range(1, int(math.sqrt(n))+1)]  def minNumSquares(k):  """ recursive solution """  # bottom cases: find a square number  if k in square\_nums:  return 1  min\_num = float('inf')  # Find the minimal value among all possible solutions  for square in square\_nums:  if k < square:  break  new\_num = minNumSquares(k-square) + 1  min\_num = min(min\_num, new\_num)  return min\_num  return minNumSquares(n) |

The above solution could work for small numbers. However, as one would find out, we would quickly run into the Time Limit Exceeded exception even for medium-size numbers (e.g. 55).

Or simply we might encounter the Stack Overflow due the to the excessive recursion.

#### **Approach 2: Dynamic Programming**

**Intuition**

The reason why it failed with the brute-force approach is simply because we re-calculate the sub-solutions over and over again. However, the formula that we derived before is still valuable. All we need is a better way to implement the formula.

One might notice that, the problem is similar to the [Fibonacci number problem](https://leetcode.com/problems/fibonacci-number/), judging from the formula. And like Fibonacci number, we have several more efficient ways to calculate the solution, other than the simple recursion.

One of the ideas to solve the stack overflow issue in recursion is to apply the **Dynamic Programming** (DP) technique, which is built upon the idea of reusing the results of intermediate sub-solutions to calculate the final solution.

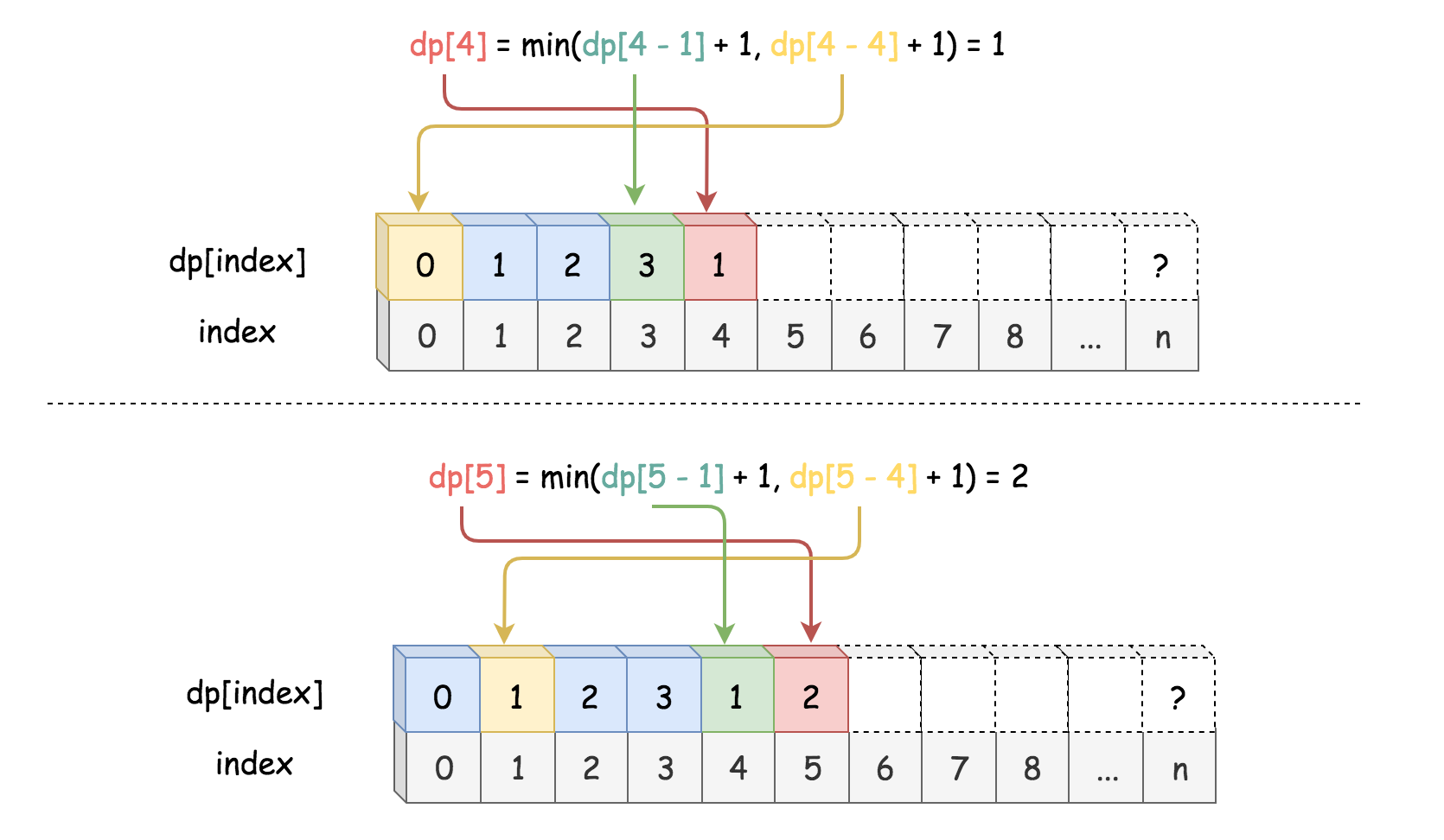
To calculate the value of numSquares(*n*), first we need to calculate all the values before n*n*, i.e. numSquares(*n*−*k*) ∀*k*∈{square numbers}. If we have already kept the solution for the number n-k*n*−*k* in somewhere, we then would not need to resort to the recursive calculation which prevents the stack overflow.

**Algorithm**

Based on the above intuition, we could implement the DP solution in the following steps.

* As for almost all DP solutions, we first create an array dp of one or multiple dimensions to hold the values of intermediate sub-solutions, as well as the final solution which is usually the last element in the array. Note that, we create a fictional element dp[0]=0 to simplify the logic, which helps in the case that the remainder (n-k) happens to be a square number.
* As an additional preparation step, we pre-calculate a list of square numbers (i.e. square\_nums) that is less than the given number n.
* As the main step, we then loop from the number 1 to n, to calculate the solution for each number i (i.e. numSquares(i)). At each iteration, we keep the result of numSquares(i) in dp[i], while resuing the previous results stored in the array.
* At the end of the loop, we then return the last element in the array as the result of the solution.

In the graph below, we illustrate how to calculate the results of numSquares(4) and numSquares(5) which correspond to the values in dp[4] and dp[5].

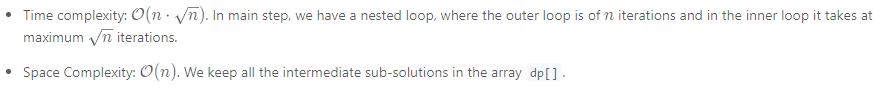


Here are some sample implementations. In particular, the Python solution took ~3500 ms, which was faster than ~50% submissions at the time.

**Note:** the following python solution works for Python2 only. For some unknown reason, it takes significantly longer time for Python3 to run the same code.

|  |
| --- |
| class Solution {  public int numSquares(int n) {  int dp[] = new int[n + 1];  Arrays.fill(dp, Integer.MAX\_VALUE);  // bottom case  dp[0] = 0;  // pre-calculate the square numbers.  int max\_square\_index = (int) Math.sqrt(n) + 1;  int square\_nums[] = new int[max\_square\_index];  for (int i = 1; i < max\_square\_index; ++i) {  square\_nums[i] = i \* i;  }  for (int i = 1; i <= n; ++i) {  for (int s = 1; s < max\_square\_index; ++s) {  if (i < square\_nums[s])  break;  dp[i] = Math.min(dp[i], dp[i - square\_nums[s]] + 1);  }  }  return dp[n];  }  } |

**Complexity**



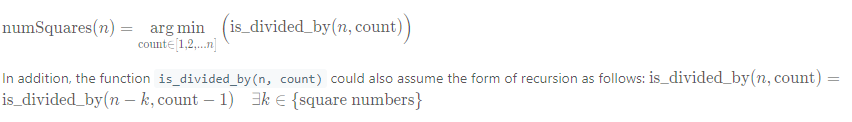
#### **Approach 3: Greedy Enumeration**

**Intuition**

Recursion isn't bad though. Above all, it provides us a concise and intuitive way to understand the problem. We could still resolve the problem with recursion. To improve the above brute-force enumeration solution, we could add a touch of Greedy into the recursion process. We could reformulate the enumeration solution as follows:

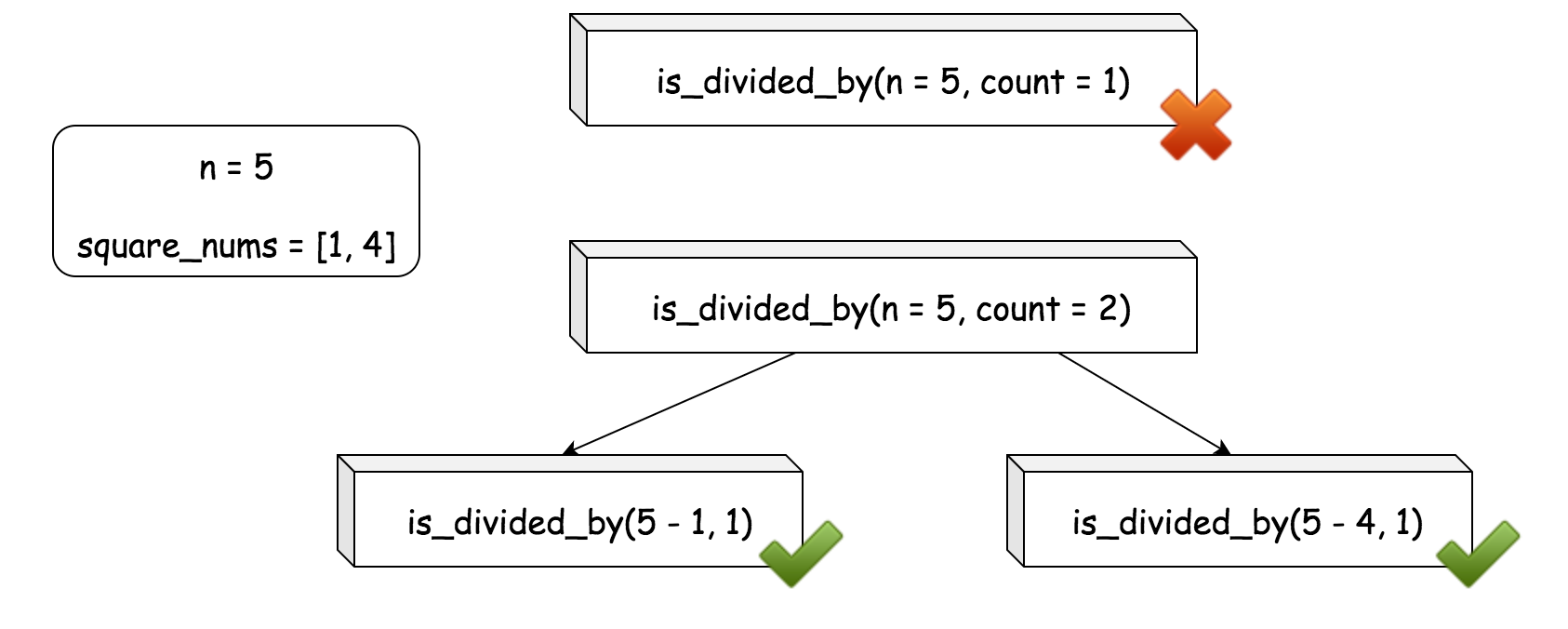
Starting from the combination of one single number to multiple numbers, once we find a combination that can sum up to the given number n, then we can say that we must have found the smallest combination, since we enumerate the combinations greedily from small to large.

To better explain the above intuition, let us first define a function called is\_divided\_by(n, count) which returns a boolean value to indicate whether the number n can be divided by a combination with count number of square numbers, rather than returning the exact size of combination as the previous function numSquares(n).



Different from the recursive function of numSquare(n), the recursion process of is\_divided\_by(n, count) would boil down to its bottom case (i.e. count==1) much more rapid.

Here is one example on how the function is\_divided\_by(n, count) breaks down for the input n=5 and count=2.



With this trick of reformulation, we could dramatically reduce the risk of stack overflow.

**Algorithm**

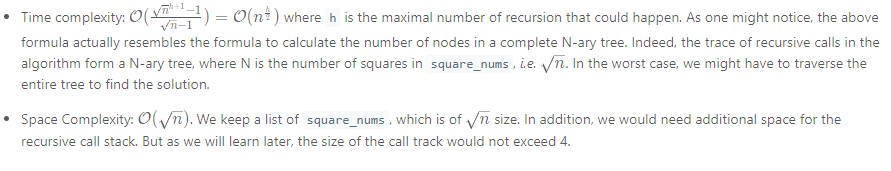
* First of all, we prepare a list of square numbers (named square\_nums) that are less than the given number n.
* In the main loop, iterating the size of combination (named count) from one to n, we check if the number n can be divided by the sum of the combination, i.e. is\_divided\_by(n, count).
* The function is\_divided\_by(n, count) can be implemented in the form of recursion as we defined in the intuition section.
* In the bottom case, we have count==1, we just need to check if the number n is a square number itself. We could use the inclusion test with the list square\_nums that we prepared before, i.e. *n* ∈ square\_nums. And if we use the set data structure for square\_nums, we could obtain a faster running time than the approach of n == int(sqrt(n)) ^ 2.

Concerning the correctness of the algorithm, often the case we could prove the Greedy algorithm by **contradiction**. This is no exception. Suppose we find a count=m that can divide the number n, and suppose in the later iterations there exists another number count=p that can also divide the number and the combination is smaller than the found one i.e. p < m. Given the order of the iteration, the count=p would have been discovered before count=m which is contradict to the fact that p comes later than m. Therefore, we can say that the algorithm works as expected, which always finds the minimal size of combination.

Here are some sample implementation. In particular, the Python solution took ~70ms, which was faster than ~ 90% submissions at the time.

|  |
| --- |
| class Solution {  Set<Integer> square\_nums = new HashSet<Integer>();  protected boolean is\_divided\_by(int n, int count) {  if (count == 1) {  return square\_nums.contains(n);  }  for (Integer square : square\_nums) {  if (is\_divided\_by(n - square, count - 1)) {  return true;  }  }  return false;  }  public int numSquares(int n) {  this.square\_nums.clear();  for (int i = 1; i \* i <= n; ++i) {  this.square\_nums.add(i \* i);  }  int count = 1;  for (; count <= n; ++count) {  if (is\_divided\_by(n, count))  return count;  }  return count;  }  } |

**Complexity**



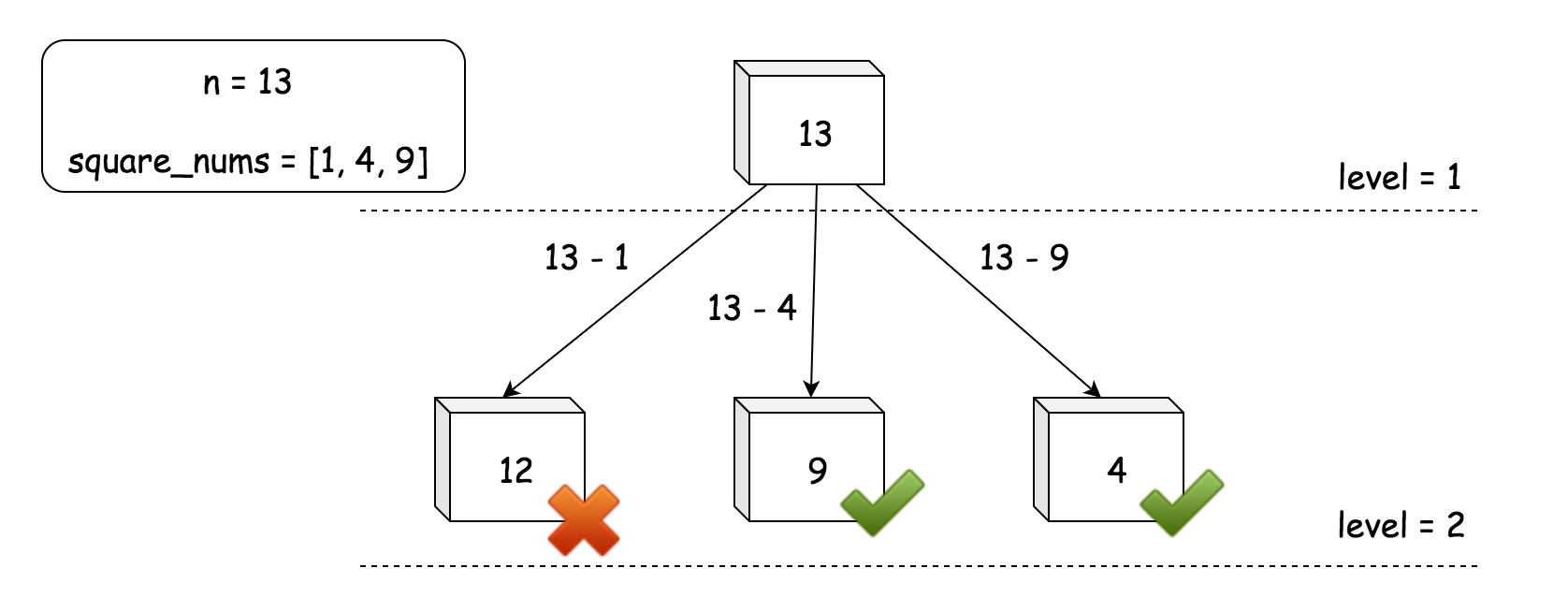
#### **Approach 4: Greedy + BFS (Breadth-First Search)**

**Intuition**

As we mentioned in the complexity analysis in the above Greedy approach, the trace of the call stack forms a N-ary tree where each node represents a call to the is\_divided\_by(n, count) function. Based on the above intuition, again we could reformulate the original problem as follows:

Given a N-ary tree, where each node represents a **remainder** of the number n subtracting a combination of square numbers, our task is to find a node in the tree, which should meet two conditions: 1). the value of the node (i.e. the remainder) should be a square number as well. 2). the distance between the node and the root should be minimal among all nodes that meet the condition (1).

Here is an example how the tree would look like.



In the previous Approach #3, due to the Greedy strategy that we perform the calls, we were actually constructing the N-ary tree level-by-level from top to down. And the we were traversing it in a **BFS** (Breadth-First Search) manner. At each level of the N-ary tree, we were enumerating the combinations that are of the same size.

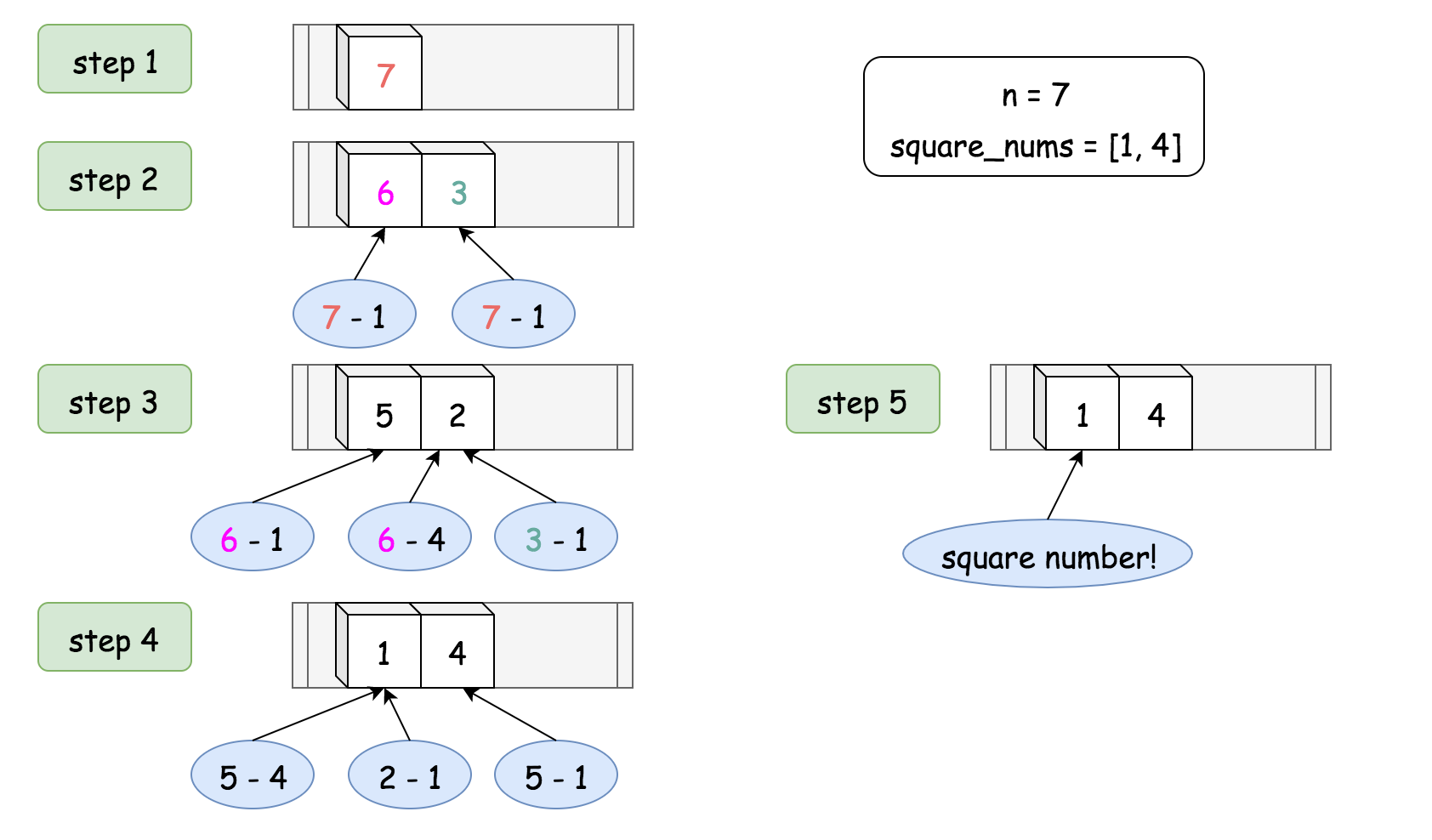
The order of traversing is of BFS, rather than DFS (Depth-First Search), is due to the fact that before exhausting all the possibilities of decomposing a number n with a fixed amount of squares, we would not explore any potential combination that needs more elements.

**Algorithm**

* Again, first of all, we prepare a list of square numbers (named square\_nums) that are less than the given number n.
* We then create a queue variable which would keep all the remainders to enumerate at each level.
* In the main loop, we iterate over the queue variable. At each iteration, we check if the remainder is one of the square numbers. If the remainder is not a square number, we subtract it with one of the square numbers to obtain a new remainder and then add the new remainder to the next\_queue for the iteration of the next level. We break out of the loop once we encounter a remainder that is of a square number, which also means that we find the solution.

**Note**: in a typical BFS algorithm, the queue variable usually would be of array or list type. However, here we use the set type, in order to eliminate the redundancy of remainders within the same level. As it turns out, this tiny trick could even provide a 5 times speedup on running time.

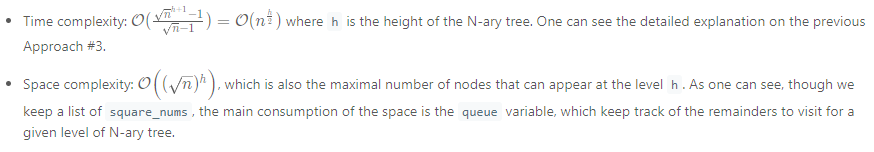
In the following graph, we illustrate the layout of the queue, on the example of numSquares(7).



Here are some sample implementations. In particular, the Python implementation inspired from the post of [ChrisZhang12240](https://leetcode.com/problems/perfect-squares/discuss/71475/Short-Python-solution-using-BFS) took ~200 ms which was faster than ~72% of submission at that time.

|  |
| --- |
| class Solution {  public int numSquares(int n) {  ArrayList<Integer> square\_nums = new ArrayList<Integer>();  for (int i = 1; i \* i <= n; ++i) {  square\_nums.add(i \* i);  }  Set<Integer> queue = new HashSet<Integer>();  queue.add(n);  int level = 0;  while (queue.size() > 0) {  level += 1;  Set<Integer> next\_queue = new HashSet<Integer>();  for (Integer remainder : queue) {  for (Integer square : square\_nums) {  if (remainder.equals(square)) {  return level;  } else if (remainder < square) {  break;  } else {  next\_queue.add(remainder - square);  }  }  }  queue = next\_queue;  }  return level;  }  } |

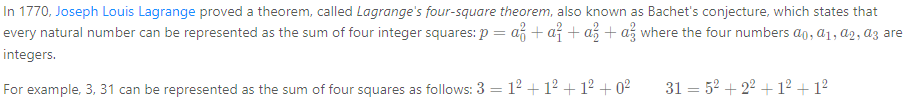
**Complexity**



#### **Approach 5: Mathematics**

**Intuition**

The problem can be solved with the mathematical theorems that have been proposed and proved over time. We will break down the problem into several cases in this section.



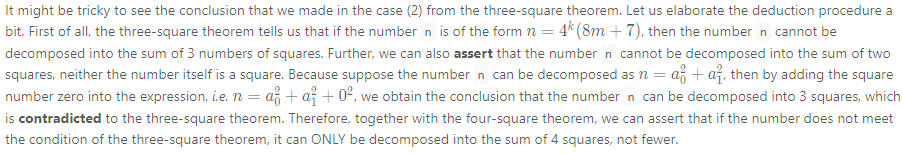
Case 1). The Lagrange's four-square theorem sets the upper bound for the results of the problem, i.e. if the number n cannot be decomposed into a fewer number of squares, at least it can be decomposed into the sum of **4** square numbers, i.e. numSquares(*n*)≤4.

As one might notice in the above example, the number zero is also considered as a square number, so we can consider that the number 3 can either be decomposed into 3 or 4 square numbers.

However, Lagrange's four-square theorem does not tell us directly the least numbers of square to decompose a natural number.



Case 2). Unlike the four-square theorem, Adrien-Marie Legendre's three-square theorem gives us a **necessary** and **sufficient** condition to check if the number can **ONLY** be decomposed into 4 squares, not fewer.



If the number meets the condition of the three-square theorem, we know that if can be decomposed into 3 squares. But what we don't know is that whether the number can be decomposed into fewer squares, i.e. one or two squares. So before we attribute the number to the bottom case (three-square theorem), here are the two cases remained to check, namely:

Case 3.1). if the number is a square number itself, which is easy to check e.g. n == int(sqrt(n)) ^ 2.

Case 3.2). if the number can be decomposed into the sum of two squares. Unfortunately, there is no mathematical weapon that can help us to check this case in one shot. We need to resort to the **enumeration** approach.

**Algorithm**

One can literally follow the above cases to implement the solution.

* First, we check if the number n is of the form n = 4^{k}(8m+7), if so we return 4 directly.
* Otherwise, we further check if the number is of a square number itself or the number can be decomposed the sum of two squares.
* In the bottom case, the number can be decomposed into the sum of 3 squares, though we can also consider it decomposable by 4 squares by adding zero according to the four-square theorem. But we are asked to find the least number of squares.

We give some sample implementations here. The solution is inspired from the posts of [TCarmic](https://leetcode.com/problems/perfect-squares/discuss/376795/100-O(log-n)-Python3-Solution-Lagrange's-four-square-theorem) and [StefanPochmann](https://tinyurl.com/y4falx4f) in the [Discussion](https://leetcode.com/problems/perfect-squares/discuss/) forum.

|  |
| --- |
| class Solution {  protected boolean isSquare(int n) {  int sq = (int) Math.sqrt(n);  return n == sq \* sq;  }  public int numSquares(int n) {  // four-square and three-square theorems.  while (n % 4 == 0)  n /= 4;  if (n % 8 == 7)  return 4;  if (this.isSquare(n))  return 1;  // enumeration to check if the number can be decomposed into sum of two squares.  for (int i = 1; i \* i <= n; ++i) {  if (this.isSquare(n - i \* i))  return 2;  }  // bottom case of three-square theorem.  return 3;  }  } |

**Complexity**

