Perfect Squares

Given a positive integer n, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to n.

For example, given n = `12`, return `3` because `12 = 4 + 4 + 4`; given n = `13`, return `2` because `13 = 4 + 9`.

Credits:
Special thanks to @jianchao.li.fighter for adding this problem and creating all test cases.

Solution 1

Came up with the 2 solutions of breadth-first search and dynamic programming. Also "copied" StefanPochmann's static dynamic programming solution (https://leetcode.com/discuss/56993/static-dp-c-12-ms-python-172-ms-ruby-384-ms) and davidtan1890's mathematical solution (https://leetcode.com/discuss/57066/4ms-c-code-solve-it-mathematically) here with minor style changes and some comments. Thank Stefan and David for posting their nice solutions!

1.Dynamic Programming: 440ms

```cpp
class Solution
{
public:
    int numSquares(int n)
    {
        if (n <= 0)
        {
            return 0;
        }

        // cntPerfectSquares[i] = the least number of perfect square numbers
        // which sum to i. Note that cntPerfectSquares[0] is 0.
        vector<int> cntPerfectSquares(n + 1, INT_MAX);
        cntPerfectSquares[0] = 0;
        for (int i = 1; i <= n; i++)
        {
            // For each i, it must be the sum of some number (i - j*j) and
            // a perfect square number (j*j).
            for (int j = 1; j*j <= i; j++)
            {
                cntPerfectSquares[i] =
                    min(cntPerfectSquares[i], cntPerfectSquares[i - j*j] + 1);
            }
        }

        return cntPerfectSquares.back();
    }
};
```

2.Static Dynamic Programming: 12ms

```
class Solution
{
public:
    int numSquares(int n)
    {
        if (n <= 0)
        {
            return 0;
        }

        // cntPerfectSquares[i] = the least number of perfect square numbers
        // which sum to i. Since cntPerfectSquares is a static vector, if
        // cntPerfectSquares.size() > n, we have already calculated the result
        // during previous function calls and we can just return the result now.
        static vector<int> cntPerfectSquares({0});

        // While cntPerfectSquares.size() <= n, we need to incrementally
        // calculate the next result until we get the result for n.
        while (cntPerfectSquares.size() <= n)
        {
            int m = cntPerfectSquares.size();
            int cntSquares = INT_MAX;
            for (int i = 1; i*i <= m; i++)
            {
                cntSquares = min(cntSquares, cntPerfectSquares[m - i*i] + 1);
            }

            cntPerfectSquares.push_back(cntSquares);
        }

        return cntPerfectSquares[n];
    }
};
```

3.Mathematical Solution: 4ms

```
class Solution
{
private:
    int is_square(int n)
    {
        int sqrt_n = (int)(sqrt(n));
        return (sqrt_n*sqrt_n == n);
    }

public:
    // Based on Lagrange's Four Square theorem, there
    // are only 4 possible results: 1, 2, 3, 4.
    int numSquares(int n)
    {
        // If n is a perfect square, return 1.
        if(is_square(n))
        {
            return 1;
        }

        // The result is 4 if and only if n can be written in the
        // form of 4^k*(8*m + 7). Please refer to
        // Legendre's three-square theorem.
        while ((n & 3) == 0) // n%4 == 0
        {
            n >>= 2;
        }
        if ((n & 7) == 7) // n%8 == 7
        {
            return 4;
        }

        // Check whether 2 is the result.
        int sqrt_n = (int)(sqrt(n));
        for(int i = 1; i <= sqrt_n; i++)
        {
            if (is_square(n - i*i))
            {
                return 2;
            }
        }

        return 3;
    }
};
```

4.**Breadth-First Search:** 80ms

```
class Solution
{
public:
    int numSquares(int n)
    {
        if (n <= 0)
        {
            return 0;
        }

        // perfectSquares contain all perfect square numbers which
        // are smaller than or equal to n.
        vector<int> perfectSquares;
        // cntPerfectSquares[i - 1] = the least number of perfect
        // square numbers which sum to i.
        vector<int> cntPerfectSquares(n);

        // Get all the perfect square numbers which are smaller than
        // or equal to n.
        for (int i = 1; i*i <= n; i++)
        {
            perfectSquares.push_back(i*i);
            cntPerfectSquares[i*i - 1] = 1;
        }

        // If n is a perfect square number, return 1 immediately.
        if (perfectSquares.back() == n)
        {
            return 1;
        }

        // Consider a graph which consists of number 0, 1,...,n as
        // its nodes. Node j is connected to node i via an edge if
        // and only if either j = i + (a perfect square number) or
        // i = j + (a perfect square number). Starting from node 0,
        // do the breadth-first search. If we reach node n at step
        // m, then the least number of perfect square numbers which
        // sum to n is m. Here since we have already obtained the
        // perfect square numbers, we have actually finished the
        // search at step 1.
        queue<int> searchQ;
        for (auto& i : perfectSquares)
        {
            searchQ.push(i);
        }

        int currCntPerfectSquares = 1;
        while (!searchQ.empty())
        {
            currCntPerfectSquares++;

            int searchQSize = searchQ.size();
            for (int i = 0; i < searchQSize; i++)
            {
                int tmp = searchQ.front();
                // Check the neighbors of node tmp which are the sum
                // of tmp and a perfect square number.
                for (auto& j : perfectSquares)
                {
                    if (tmp + j == n)
                    {
                        // We have reached node n.
                        return currCntPerfectSquares;
                    }
                    else if ((tmp + j < n) && (cntPerfectSquares[tmp + j - 1] ==
0))
                    {
                        // If cntPerfectSquares[tmp + j - 1] > 0, this is not
                        // the first time that we visit this node and we should
                        // skip the node (tmp + j).
                        cntPerfectSquares[tmp + j - 1] = currCntPerfectSquares;
                        searchQ.push(tmp + j);
                    }
                    else if (tmp + j > n)
                    {
                        // We don't need to consider the nodes which are greater
]
                        // than n.
                        break;
                    }
                }

                searchQ.pop();
            }
        }

        return 0;
    }
};
```

written by zhukov original link here

Solution 2

dp[n] indicates that the perfect squares count of the given n, and we have:

```
dp[0] = 0
dp[1] = dp[0]+1 = 1
dp[2] = dp[1]+1 = 2
dp[3] = dp[2]+1 = 3
dp[4] = Min{ dp[4-1*1]+1, dp[4-2*2]+1 }
      = Min{ dp[3]+1, dp[0]+1 }
      = 1
dp[5] = Min{ dp[5-1*1]+1, dp[5-2*2]+1 }
      = Min{ dp[4]+1, dp[1]+1 }
      = 2
                         .
                         .
                         .
dp[13] = Min{ dp[13-1*1]+1, dp[13-2*2]+1, dp[13-3*3]+1 }
       = Min{ dp[12]+1, dp[9]+1, dp[4]+1 }
       = 2
                         .
                         .
                         .
dp[n] = Min{ dp[n - i*i] + 1 },  n - i*i >=0 && i >= 1
```

and the sample code is like below:

```java
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) {
        int min = Integer.MAX_VALUE;
        int j = 1;
        while(i - j*j >= 0) {
            min = Math.min(min, dp[i - j*j] + 1);
            ++j;
        }
        dp[i] = min;
    }
    return dp[n];
}
```

Hope it can help to understand the DP solution.

written by Karci original link here

Solution 3

These solutions use some number theory (see explanation further down).

Ruby solution

```ruby
require 'prime'

def num_squares(n)
  n /= 4 while n % 4 == 0
  return 4 if n % 8 == 7
  return 3 if n.prime_division.any? { |p, e| p % 4 == 3 && e.odd? }
  (n**0.5).to_i**2 == n ? 1 : 2
end
```

Or:

```ruby
require 'prime'

def num_squares(n)
  n /= 4 while n % 4 == 0
  return 4 if n % 8 == 7
  pd = n.prime_division
  return 3 if pd.any? { |p, e| p % 4 == 3 && e.odd? }
  pd.any? { |_, e| e.odd? } ? 2 : 1
end
```

C++ solution

```cpp
int numSquares(int n) {
    while (n % 4 == 0)
        n /= 4;
    if (n % 8 == 7)
        return 4;
    bool min2 = false;
    for (int i=2; i<=n; ++i) {
        if (i > n/i)
            i = n;
        int e = 0;
        while (n % i == 0)
            n /= i, ++e;
        if (e % 2 && i % 4 == 3)
            return 3;
        min2 |= e % 2;
    }
    return 1 + min2;
}
```

C solution

Inspired by kevin36's solution. We don't really need to compute the prime factorization. Knowing that four squares always suffice and using the three-squares test is enough. Single-square and sum-of-two-squares cases can be done simpler.

```c
int numSquares(int n) {
    while (n % 4 == 0)
        n /= 4;
    if (n % 8 == 7)
        return 4;
    for (int a=0; a*a<=n; ++a) {
        int b = sqrt(n - a*a);
        if (a*a + b*b == n)
            return 1 + !!a;
    }
    return 3;
}
```

Explanation

I happen to have given a little talk about just this topic a while back in a number theory seminar. This problem is completely solved, in the sense of being reduced to simple checks of a number's prime factorization. A natural number is...

- ... a **square** if and only if each prime factor occurs to an even power in the number's prime factorization.
- ... a **sum of two squares** if and only if each prime factor that's 3 modulo 4 occurs to an even power in the number's prime factorization.
- ... a **sum of three squares** if and only if it's not of the form $4^a(8b+7)$ with integers a and b.
- ... a **sum of four squares**. Period. No condition. You never need more than four.

Of course single squares can also be identified by comparing a given number with the square of the rounded root of the number.

The problem statement says *"1, 4, 9, 16, ..."*, for some reason apparently excluding 0, but it really is a perfect square and the above theorems do consider it one. With that, you can for example always extend a sum of two squares a^2+b^2 to the sum of three squares $a^2+b^2+0^2$. Put differently, if n isn't a sum of three squares, then it also isn't a sum of two squares. So you can read the above statements as *"... a sum of m* ***(or fewer)*** *squares"*. Thanks to ruben3 for asking about this in the comments.

In my above solutions, I first divide the given number by 4 as often as possible and then do the three-squares check. Dividing by 4 doesn't affect the other checks, and the n % 8 == 7 is cheaper than the prime factorization, so this saves time in cases where we do need four squares.

Armed with just the knowledge that you never need more than four squares, it's also easy to write O(n) solutions, e.g.:

```
int numSquares(int n) {
    int ub = sqrt(n);
    for (int a=0; a<=ub; ++a) {
        for (int b=a; b<=ub; ++b) {
            int c = sqrt(n - a*a - b*b);
            if (a*a + b*b + c*c == n)
                return !!a + !!b + !!c;
        }
    }
    return 4;
}
```

written by StefanPochmann original link here