<https://leetcode.com/problems/factor-combinations/>

Numbers can be regarded as product of its factors. For example,

8 = 2 x 2 x 2;

= 2 x 4.

Write a function that takes an integer n and return all possible combinations of its factors.

You may assume that n is always positive.

Factors should be greater than 1 and less than n.

**Example1**

Input: 12

Output:

[

[2, 6],

[2, 2, 3],

[3, 4]

]

Explanation:

2\*6 = 12

2\*2\*3 = 12

3\*4 = 12

**Example2**

Input: 32

Output:

[

[2, 16],

[2, 2, 8],

[2, 2, 2, 4],

[2, 2, 2, 2, 2],

[2, 4, 4],

[4, 8]

]

Explanation:

2\*16=32

2\*2\*8=32

2\*2\*2\*4=32

2\*2\*2\*2\*2=32

2\*4\*4=32

4\*8=32

**Attempt 1: 2022-10-31**

**TLE Solution, caused by not tight enough on factors range limitation on next recursion**

public class Solution {

/\*\*

\* @param n: a integer

\* @return: return a 2D array

\* we will sort your return value in output

\*/

public List<List<Integer>> getFactors(int n) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

helper(n, result, new ArrayList<Integer>(), 1);

return result;

}

private void helper(int n, List<List<Integer>> result, List<Integer> tmp, int product) {

if(product == n) {

result.add(new ArrayList<Integer>(tmp));

return;

}

if(product > n) {

return;

}

// Start from 2 and no more than half of n is the range

for(int i = 2; i <= n / 2; i++) {

// Condition 1: if mod not equal to 0 skip this factor

// Condition 2: if coming factor strictly smaller than existing last element on 'tmp'

// skip this factor otherwise it will create duplicate combinations

if(n % i != 0 || tmp.size() > 0 && i < tmp.get(tmp.size() - 1)) {

continue;

}

tmp.add(i);

// 'product' is primitive type variable, not like 'tmp' object, NO backtrack needed in

// recursion, when next recursion level finished and return to current recursion level,

// 'product' value will auto restore

//product \*= i;

helper(n, result, tmp, product \* i);

//product /= i;

tmp.remove(tmp.size() - 1);

}

}

}

**Solution 1:  Recursive traversal with gradually tighter limitation on factors range on next recursion (30min)**

public class Solution {

/\*\*

\* @param n: a integer

\* @return: return a 2D array

\* we will sort your return value in output

\*/

public List<List<Integer>> getFactors(int n) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

helper(n, result, new ArrayList<Integer>(), n, 2);

return result;

}

private void helper(int n, List<List<Integer>> result, List<Integer> tmp, int product, int start) {

if(product == 1) {

// Filter out 'product' itself

if(tmp.size() > 1) {

result.add(new ArrayList<Integer>(tmp));

}

return;

}

if(product < 1) {

return;

}

for(int i = start; i <= product; i++) {

if(product % i != 0) {

continue;

}

tmp.add(i);

// Next recursion factors range limit will update from current [start, product] to [i, product / i]

// e.g n = 12, 1st recursion scan between [2, 12], 2nd recursion scan between [2, 6], 3rd recursion

// scan between [2, 3], 4th recursion is [3, 3], the range limit gradually become tighter

helper(n, result, tmp, product / i, i);

tmp.remove(tmp.size() - 1);

}

}

}

Time Complexity: O(logn \* logn) ~ O(n \* logn) -> logn is depth of recursion tree and (logn ~ n) is for loop in each recursion level

Space Complexity: O(logn)

**Time Complexity is a problem:**

<https://stackoverflow.com/questions/58607349/what-is-the-time-complexity-of-this-function-that-generates-all-unique-factor-c>

Q: The way this algorithm works is I iterate over all the factors and multiply cur by the factor I am iterating over and as long as cur \* factors[i] <= n I can add that factor to my path and keep recursing.

I can't figure out the time complexity in terms of n though. I can say that in the worst case the recursion tree is gonna have depth log n (that would be 2 x 2 x 2 ... x 2 if n is a power of 2) but I am stuck on making sense of the branching factor for this tree.

Any help to calculate the time complexity of this algorithm is welcome, but I would be very grateful for an intuitive way to look at it (something I can replicate in an interview)... more formal methods are also welcome.

EDIT 1:

So I can say this recurrence has log(n) branches (number of factors) and log(n) depth in the worst case resulting in a runtime of O(log(n)^log(n)) is this reasoning good ?

EDIT 2:

However another way of looking at it is we have O(log(n)) factors and we are just doing a subset of all the possible combinations, which is a 2^n exercise thus resulting in 2^log(n) = n different solutions. And for each of the n solutions we have log(n) (tree depth) multiplications resulting in a O(nlog(n)) runtime ... so my question --> Which analysis is correct ?

A1: One observation:

The number of factors of n in the worst case happens when n is the product of a consecutive number of the smallest primes. I.e. 2\*3\*5\*7\*11 etc.

I was curious about how fast this number grows, as a function of n (again, in the worst case). Using Python, and looking at the first 100 or so primes, it seems the 10-based logarithm of n grows a little bit faster than the number of factors in n. For small values, the numbers are almost the same, but the difference keeps getting bigger and after 70 or so factors (that is - the product of the 70 first primes), the logarithm is more than twice the number of factors.

Another way of putting it is that [number of factors of n] grows slower than log n.

Comments:

1. This is a well-studied problem and, indeed, the average number of divisors of n is log n.
2. Yes indeed James, I did find that information --> So I can say this recurrence has log(n) branches and log(n) depth in the worst case resulting in a runtime of O(log(n)^log(n)) do you agree ?

A2: The general case of time complexity is T(n) = sum\_{i=1}{number of factors of n}(T(n/f\_i)) + c (f\_is are factors of n). If n = 2^k, the time complexity is T(n) = log(n) T(n/2) + c. Hence:

T(n) = log(n) T(n/2) + c =

log(n) (log(n/2) T(n/2^2) + c) + c =

log(n) log(n/2) T(n/2^2) + c (1 + log(n)) =

k \* (k-1) \* T(n/2^2) +‌c (1 + k) =

k \* (k-1) \* (log(n/2^2) T(n/2^3) + c) + c (1 + k) =

k \* (k-1) \* (k-2) T(n/2^3) +‌c (1 + k + k \* (k-1)) = Omega(log(n)!)

We used the Omega because it is just for the case of 2^k. To know more about the complexity, you need to scrutinize more into the general term.