Integer Break

Given a positive integer n, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given n = 2, return 1 (2 = 1 + 1); given n = 10, return 36 (10 = 3 + 3 + 4).

Note: you may assume that n is not less than 2.

1. There is a simple O(n) solution to this problem.
2. You may check the breaking results of n ranging from 7 to 10 to discover the regularities.

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

Solution 1

I saw many solutions were referring to factors of 2 and 3. But why these two magic numbers? Why other factors do not work? Let's study the math behind it.

For convenience, say **n** is sufficiently large and can be broken into any smaller real positive numbers. We now try to calculate which real number generates the largest product. Assume we break **n** into **(n / x)** **x**'s, then the product will be $x^{n/x}$, and we want to maximize it.

Taking its derivative gives us $n * x^{n/x-2} * (1 - \ln(x))$. The derivative is positive when $0 < x < e$, and equal to **0** when $x = e$, then becomes negative when $x > e$, which indicates that the product increases as **x** increases, then reaches its maximum when $x = e$, then starts dropping.

This reveals the fact that if **n** is sufficiently large and we are allowed to break **n** into real numbers, the best idea is to break it into nearly all **e**'s. On the other hand, if **n** is sufficiently large and we can only break **n** into integers, we should choose integers that are closer to **e**. The only potential candidates are **2** and **3** since $2 < e < 3$, but we will generally prefer **3** to **2**. Why?

Of course, one can prove it based on the formula above, but there is a more natural way shown as follows.

$6 = 2 + 2 + 2 = 3 + 3$. But $2 * 2 * 2 < 3 * 3$. Therefore, if there are three **2**'s in the decomposition, we can replace them by two **3**'s to gain a larger product.

All the analysis above assumes **n** is significantly large. When **n** is small (say $n <= 10$), it may contain flaws. For instance, when $n = 4$, we have $2 * 2 > 3 * 1$. To fix it, we keep breaking **n** into **3**'s until **n** gets smaller than **10**, then solve the problem by brute-force.

written by lixx2100 original link here

Solution 2

Given a number n lets say we have a possible product P = p1 * p2 * *...pk. Then we notice what would happen if we could break pi up into two more terms lets say one of the terms is 2 we would get the terms pi-2 and 2 so if 2*(pi-2) > pi we would get a bigger product and this happens if pi > 4. since there is one other possible number less then 4 that is not 2 aka 3. Likewise for 3 if we instead breakup the one of the terms into pi-3 and 3 we would get a bigger product if 3*(pi-3) > pi which happens if pi > 4.5.

Hence we see that all of the terms in the product must be 2's and 3's. So we now just need to write n = a*3 + b*2 such that P = (3^a) * (2^b) is maximized. Hence we should favor more 3's then 2's in the product then 2's if possible.

So if n = a*3 then the answer will just be 3^a.

if n = a*3 + 2 then the answer will be 2*(3^a).

and if n = a*3 + 2*2 then the answer will be 2 * 2 * 3^a

The above three cover all cases that n can be written as and the Math.pow() function takes O(log n) time to preform hence that is the running time.

```java
public class Solution {
    public int integerBreak(int n) {
        if(n == 2)
            return 1;
        else if(n == 3)
            return 2;
        else if(n%3 == 0)
            return (int)Math.pow(3, n/3);
        else if(n%3 == 1)
            return 2 * 2 * (int) Math.pow(3, (n - 4) / 3);
        else
            return 2 * (int) Math.pow(3, n/3);
    }

}
```

written by kevin36 original link here

Solution 3

```
If we want to break a number, breaking it into 3s turns out to be the most efficie
nt.
2^3 < 3^2
4^3 < 3^4
5^3 < 3^5
6^3 < 3^6
...

Therefore, intuitively, we want as many 3 as possible
if a number % 3 == 0, we just break it into 3s -> the product is Math.pow(3, n/3)

As for numbers % 3 == 1, we don't want the 'times * 1' in the end;
    borrowing a 3 is a natural thought.
    if we borrow a 3, 3 can be divided into
         case 1: 1 + 2 -> with the extra 1, we have 2*2 = 4
         case 2: (0) + 3 -> with the extra 1, we have 4
         turns out these two cases have the same results
    so, for numbers % 3 == 1 -> the result would be Math.pow(3, n/3-1)*4

Then we have the numbers % 3 == 2 left
     again, we try to borrow a 3,
         case 1: 1+2 -> with the extra 2, we have 1*5 or 3*2 => 3*2 is better
         case 2: 0+3 -> with the extra 2, we have 2*3 or 5 => 2*3 is better
     and we actually just end up with not borrowing at all!
     so we can just *2 if we have an extra 2 -> the result would be Math.pow(3, n/
3)*2

Then, we have a couple corner cases two deal with since so far we only looked at
numbers  that are larger than 3 -> luckily, we only have 2 and 3 left,
which are pretty easy to figure out

Thus my final solution is

public class Solution {
    public int integerBreak(int n) {
        if(n <= 3) return n-1; //assuming n >= 2
        return n%3 == 0 ? (int)Math.pow(3, n/3) : n%3 == 1 ? (int)Math.pow(3, n/3
-1)*4 : (int)Math.pow(3, n/3)*2;
    }
}
```

written by aiscong original link here