RECURSIVE SOLUTION

If we have to split an integer n in two parts, the two parts should be equal in order to get the maximum multiplication possible. If n is and odd number, then n is going to be split between (n/2 and ((n/2)+ 1).

The following algorithm is going to produce the highest multiplication by recursively splitting the ribbon in half for each call, and stopping the process when the ribbon piece has length 3 or lower. That's because you could split any nmber grater than 3 to gain a greater product.

For example: 7 * 3 < 2 * 2 * 3 * 3

When the algorithm has done cutting that piece, insert the length of that piece in a list.

In order to get the maximum product, we should replace any triplet of 2 in the list with a couple of 3, as 2+2+2 = 3+3 but 2*2*2 < 3*3. In general, the correct solution comes with a list of ribbon pieces of length 3, plus eventually one additional ribbon of length 2 or 4, depending on which integer N we are going to use as an input.

Finally we are going to multiplicate the length of every produced piece, which are the elements of the produced list, to get the highest multiplication value.

Recursive pseudocode (Algorithm **1**):

Input: integer n

```
Global out_list
Global has_cut
has_cut = false

cut_ribbon(n):

  If (n < 4 and has_cut == true):
      out_list.append(n)
      return
  Else:
       has_cut=true
       if(n is even):
           cut_ribbon(n/2)
           cut_ribbon(n/2)
       if(n is odd):
           cut_ribbon(n//2)
           cut_ribbon((n//2) +1)

cut_ribbon(n)
Replace_triplets(out_list)
Multiply_all_elements(out_list)
```

For the input n=15 the ribbon is going to be split as follows:

15 -> (8,7) -> (4,4,3,4) -> (2,2,2,2,3,2,2)

(2,2,2,2,3,2,2) -> (3,3,3,3,3)

Output: 3*3*3*3*3 = 243

COMPLEXITY ANALYSIS

We are now going to analyze the complexity of this solution on the computational cost of **cut_ribbons**(), because it's the most expensive function used in the algorithm.

This function calls itself 2 times for each call until the ribbon piece is down to length 3 or less. This means that we are halving the input for every call. So for n = 10 the algorithm will call **cut_ribbons**() as follows:

```
Cut_ribbons(15)
    Cut_ribbons(7)
        cut_ribbons(3)
        cut_ribbons(4)
            cut_ribbons(2)
            cut_ribbons(2)
    Cut_ribbons(8)
        cut_ribbons(4)
            cut_ribbons(2)
            cut_ribbons(2)
        cut_ribbons(4)
            cut_ribbons(2)
            cut_ribbons(2)
```

So for n=15 the function will be executed 13 times.

For n=50 the function will be executed 42 times

For n=4000 the function will be executed 3910 times

As we can see, the number of calls is always really close to N. We can have a practical demonstration of this by implementing a counter that increments for each call of **cut_ribbons**().

Therefore we can say that this algorithm is going to execute an **O**(1) cost function nearly about N times. That's because for each step we double the number of function calls but we also halve the input size: cut_ribbons(N) produces two cut_ribbon(N/2), which then produce four cut_ribbon(N/4) and so on. We can then say that the asymptotical complexity of the algorithm is **O**(N).

We are leaving out **Replace_triplets**() and **Multiply_all_elements**() from this analysis because their computational cost linearly depends on the length of the produced list, which is always going to be shorter than the input **N** (something between N/2 and N/3), so they are going to be cheaper (**O**(length(out_list))).

ITERATIVE SOLUTIONS

The iterative resolution to this task is much simpler. We just need to cut the ribbons by pieces of length 3 one at a time, as we update the product for every cut, until N is lower than 5 which is going to be our last piece.

We will have to explicitly handle ribbons of initial length lower than 4.

Pseudocode for iterative solution (Algorithm **2**):

```
iterative_cut_ribbon(n):
if (n == 2 or n == 3):
       return (n - 1);
out = 1;
       while (n > 4):
           n -= 3;
           out = out*3
return (n * out)
```

This solution uses one loop which is going to iterate N/3 times, so the asymptotic complexity of the algorithm is going to be **O**(N/3), which belongs to **O**(N) class time complexity.

This algorithm is slightly better than the recursive one because it is going to compute roughly N/3 operations, while with the recursive solution we are going to compute a number of operations that can eventually sum up to "almost N". Although they belong to the same time complexity class **O**(N)

An even simpler solution would be a single iteration of a function which returns the following number:

$$3^{\frac{n}{3}} \times r\left(\frac{n}{3}\right)$$

Where n/3 is an integer division and r(n/3) is the rest of that division.

If we really had to use a **sub-optimal** solution just for the sake of dynamic programming, we could use this iterative algoritm instead (Algorithm **3**):

```
Dynamic_cut_ribbon(n)
    list = [0 for i in range n+1]
    list[1] = 0
    list[2] = 1
    for i in range(n+1)
        for j in range(i)
           list[i] = max(list[i], max(i - j, list[i - j]) * j)

    return list[n]
```

This algorithm creates an in-place solution by building the list choosing the highest value between the element at current index *i* and the highest value between *i-j* and *j* times the element at index *i-j*. This process is ultimately going to produce the highest product at his last iteration.

Because there are 2 nested loops whose number of iterations linearly depends on n, we can say that the asymptotical time complexity is **O**(N^2).

RUNNING TIME ANALYSIS

We are going to plot running times for all 3 algorithms.

Remember that:

1. Recursive O(N)
2. Iterative O (N/3) ~ O(N)
3. Dynamic Programming $O(N^2)$

For inputs within range [2,25], we produce the following running times:

Recursive: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Iterative: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Dynamic Programming: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0005011558532714844, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0005002021789550781, 0.0]

As we can see, algorithms run too fast on our machine to register decent running times for such small input.

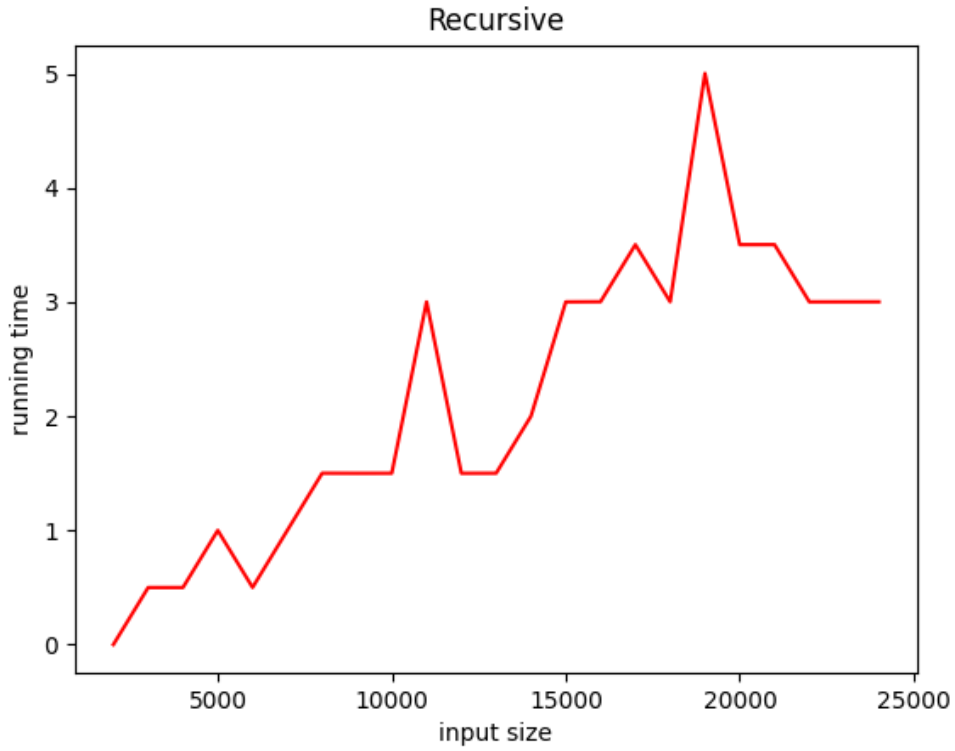
For this reason we are going to plot inputs within range [2000, 25000], with an interval of 1000. We are now producing following running times for this range:

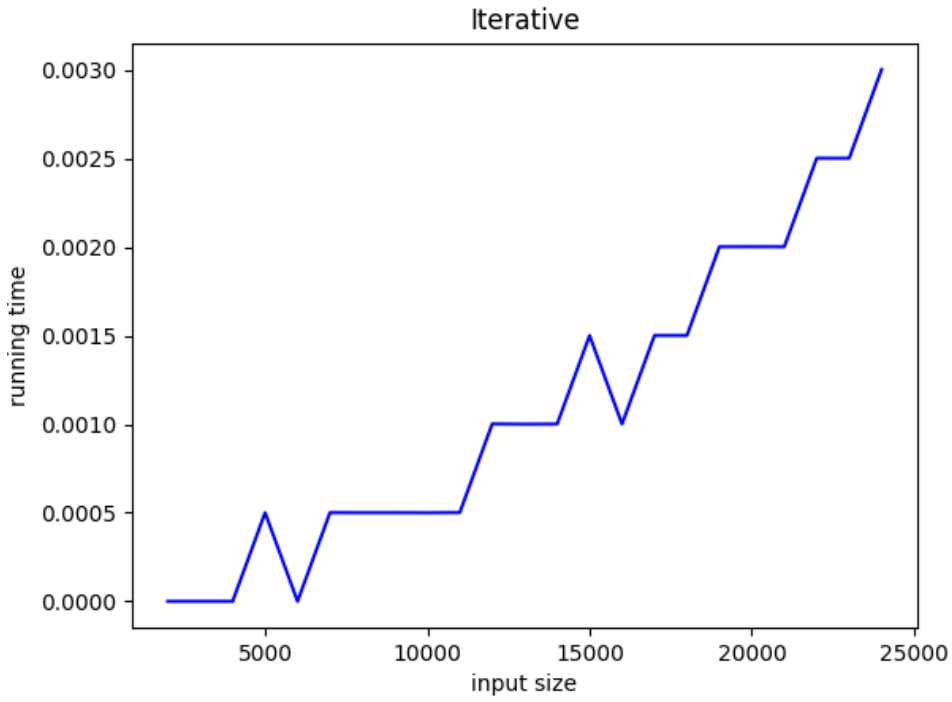
Recursive: [1.001119613647461, 1.5015602111816406, 13.010978698730469, 21.518468856811523, 20.517587661743164, 69.56005096435547, 158.13636779785156, 226.19390487670898, 280.2412509918213, 299.757719039917, 163.64073753356934, 280.2400588989258, 607.52272605896, 1074.922800064087, 1664.930820465088, 2077.2852897644043, 2426.084041595459, 2701.3213634490967, 2722.839117050171, 2847.4466800689697, 2320.9939002990723, 1577.354907989502, 751.1463165283203]

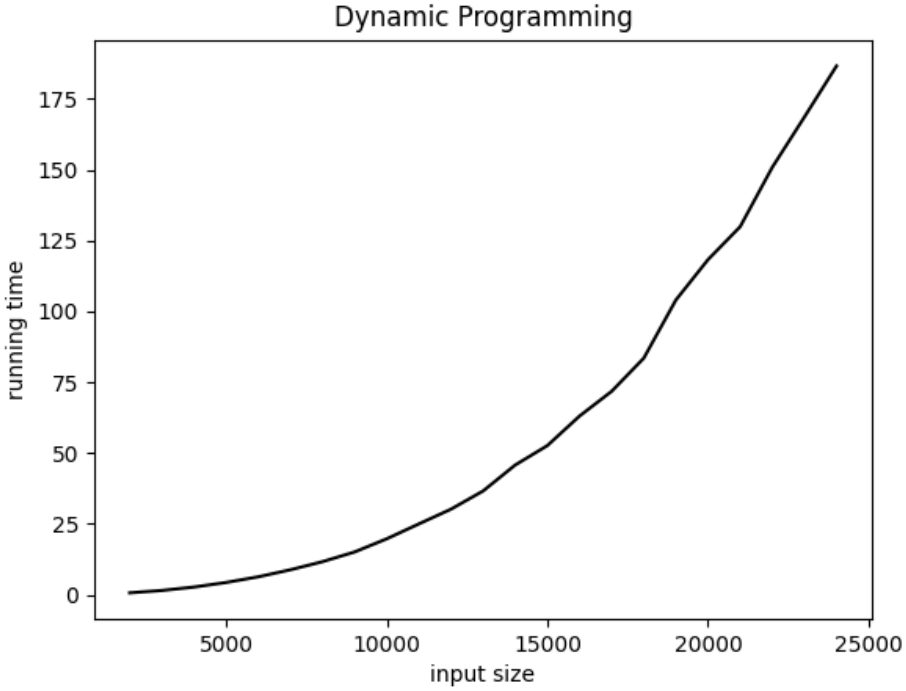
Iterative: [0.0005002021789550781, 0.0, 0.0005004405975341797, 0.0, 0.0005004405975341797, 0.0004999637603759766, 0.0005006790161132812, 0.0005004405975341797, 0.0010006427764892578, 0.0010008811950683594, 0.0004999637603759766, 0.0010013580322265625, 0.0010006427764892578, 0.0015015602111816406, 0.0015008449554443336, 0.0015008449554443336, 0.0020012855529785156, 0.0015010833740234375, 0.002001523971557617, 0.0020024776458740234, 0.0025022029876708984, 0.0025014877319335938, 0.002501249313354492]

Dynamic Programming: [0.6470556259155273, 1.5007901191711426, 2.9130022525787354, 4.485854387283325, 6.65071439743042, 8.911656379699707, 11.896220684051514, 15.508825302124023, 20.722304105758667, 24.990471363067627, 30.729902982711792, 37.36410140991211, 44.97063851356506, 53.652597188949585, 60.95937538146973, 70.41449975967407, 81.33337998390198, 92.48896503448486, 105.39655470848083, 117.96134948730469, 131.81175112724304, 146.9797821044922, 162.58318853378296]

From this dataset we get the following plots. Running time is expressed in seconds.


Recursive
running time
input size
0
1
2
3
4
5
5000
10000
15000
20000
25000


Iterative
running time
input size
0.0000
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030
5000
10000
15000
20000
25000



Lowest running times are registered on iterative algorithm, while the slowest is the one that uses Dynamic Programming.

We can see that Recursive and Iterative growths are pretty much linear, while Dynamic Programming is quadratic. This respects each algorithm's time class complexity ($O(N),O(N),O(N^2)$).

We can say that the worthiest algorithm is the Iterative one.