Given a rope of length n meters, cut the rope in different parts of integer lengths in a way that maximizes product of lengths of all parts. You must make at least one cut. Assume that the length of rope is more than 2 meters.

Examples:

```
Input: n = 2
Output: 1 (Maximum obtainable product is 1*1)

Input: n = 3
Output: 2 (Maximum obtainable product is 1*2)

Input: n = 4
Output: 4 (Maximum obtainable product is 2*2)

Input: n = 5
Output: 6 (Maximum obtainable product is 2*3)

Input: n = 10
Output: 36 (Maximum obtainable product is 3*3*4)
```

1) Optimal Substructure:
This problem is similar to Rod Cutting Problem. We can get the maximum product by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.

Let maxProd(n) be the maximum product for a rope of length n. maxProd(n) can be written as following.

maxProd(n) = max(i*(n-i), maxProdRec(n-i)*i) for all i in {1, 2, 3 .. n}

2) Overlapping Subproblems
Following is simple recursive C++ implementation of the problem. The implementation simply follows the recursive structure mentioned above.

```
// A Naive Recursive method to find maxium product
#include <iostream>
using namespace std;

// Utility function to get the maximum of two and three integers
int max(int a, int b) { return (a > b)? a : b;}
int max(int a, int b, int c) { return max(a, max(b, c));}

// The main function that returns maximum product obtainable
// from a rope of length n
int maxProd(int n)
{
    // Base cases
    if (n == 0 || n == 1) return 0;

    // Make a cut at different places and take the maximum of all
    int max_val = 0;
    for (int i = 1; i < n; i++)
      max_val = max(max_val, i*(n-i), maxProd(n-i)*i);

    // Return the maximum of all values
    return max_val;
}

/* Driver program to test above functions */
int main()
{
```

```
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}
```

Output:

```
Maximum Product is 36
```

Considering the above implementation, following is recursion tree for a Rope of length 5.

```
mP() ---> maxProd()

                              mP(5)
                  /         /      \     \
                 /         /        \      \
            mP(4)        mP(3)     mP(2)   mP(1)
           /  |  \        /  \        |
          /   |   \      /    \       |
     mP(3) mP(2) mP(1) mP(2) mP(1) mP(1)
    / \        |          |
   /   \       |          |
 mP(2) mP(1) mP(1)       mP(1)
```

In the above partial recursion tree, mP(3) is being solved twice. We can see that there are many subproblems which are solved again and again. Since same suproblems are called again, this problem has Overlapping Subprolems property. So the problem has both properties (see this andthis) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array val[] in bottom up manner.

```
// A Dynamic Programming solution for Max Product Problem
int maxProd(int n)
{
   int val[n+1];
   val[0] = val[1] = 0;

   // Build the table val[] in bottom up manner and return
   // the last entry from the table
   for (int i = 1; i <= n; i++)
   {
      int max_val = 0;
      for (int j = 1; j <= i/2; j++)
         max_val = max(max_val, (i-j)*j, j*val[i-j]);
      val[i] = max_val;
   }
   return val[n];
}
```

Time Complexity of the Dynamic Programming solution is O(n^2) and it requires O(n) extra space.

A Tricky Solution:
If we see some examples of this problems, we can easily observe following pattern.
The maximum product can be obtained be repeatedly cutting parts of size 3 while size is greater than 4, keeping the last part as size of 2 or 3 or 4. For example, n = 10, the maximum product is obtained by 3, 3, 4. For n = 11, the maximum product is obtained by 3, 3, 3, 2. Following is C++ implementation of this approach.

```
#include <iostream>
using namespace std;

/* The main function that teturns the max possible product */
int maxProd(int n)
```

```
{
   // n equals to 2 or 3 must be handled explicitly
   if (n == 2 || n == 3) return (n-1);

   // Keep removing parts of size 3 while n is greater than 4
   int res = 1;
   while (n > 4)
   {
       n -= 3;
       res *= 3; // Keep multiplying 3 to res
   }
   return (n * res); // The last part multiplied by previous parts
}

/* Driver program to test above functions */
int main()
{
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}
```

Output:

```
Maximum Product is 36
```