# Problem 1: Cut the Rope

*/\*\*  
 \* Cut The Rope  
 \* Given a rope with length n, find the maximum value maxProduct, that can be achieved for product len[0] \* len[1] \* ... \* len[m - 1],  
 \* where len[] is the array of lengths obtained by cutting the given rope into m parts.  
 \*  
 \* Note that  
 \* there should be atleast one cut, i.e. m >= 2.  
 \* All m parts obtained after cut should have non-zero integer valued lengths.  
 \*  
 \* For input n = 5, output will be:  
 \* 6  
 \*  
 \* Constraints:  
 \* 2 <= n <= 111  
 \* We have to cut at least once. (2 <= m).  
 \* Length of the rope, as well as the length of each part are in positive integer value. (i.e. can't do partial cuts.)  
 \*  
 \* Sample Input:  
 \* 4  
 \*  
 \* Sample Output:  
 \* 4  
 \*  
 \* Explanation:  
 \* For n = 4, there are two cuts possible: 1 + 3 and 2 + 2.  
 \* We'll pick 2 + 2, because their product 2 \* 2 = 4 is greater than product of the first one 1 \* 3 = 3.  
 \* (So our m = 2, n[0] = 2 and n[1] = 2 and product is n[0] \* n[1] = 4.)  
 \*  
 \* resources/CutRopeMaxProductRecursion.jpg  
 \* resources/CutRopeMaxProductDp.jpg  
 \* resources/CutRopeMaxProductRecursion1.jpg  
 \* resources/CutRopeMaxProductTrickyApproach.jpg  
 \*  
 \* Recursion Solution  
 \*  
 \* Time Complexity: O(n^n) O(n!)  
 \* Space Complexity: O(n)  
 \*  
 \* Dp Solution:  
 \*  
 \* Time Complexity: O(n^2)  
 \* Space Complexity: O(n)  
 \*  
 \* Tricky Approach from observation  
 \*  
 \* For i >= 5, there's always one cut of length = 3  
 \*  
 \* Time Complexity: O(n)  
 \* Space Complexity: O(n)  
 \*  
 \* Another tricky Approach  
 \*  
 \* 1) If the rope is multiple of 3, cut into 3 equal pieces.  
 \* 2) If rope is multiple of 3 + 1, cut one piece of len 4 and remaining of len 3  
 \* 3) If rope is multiple of 3 + 2, cut one piece of length 2 and remaining of len 3  
 \*  
 \* Power function:  
 \* Time complexity: O(logb) where b = n/3  
 \* Space complexity: O(logb) where b = n/3  
 \*/*

If getting wrong answer then first check if you are using appropriate data type in intermediate calculations. (For the given constraints integer will overflow.)

We have provided three solutions:

1) other\_solution\_1.cpp: dp quadratic solution.

2) other\_solution\_2.cpp: dp linear solution. Solution from observing fixed pattern.

3) optimal\_solution.cpp: solution from observing fixed pattern. (Even though if you are able to directly come to this solution, we expect you to write dp solution once.)

Have a look at the solutions. All of them contain detailed comments.

other\_solution\_1.cpp:

Time Complexity:

O(n^2).

We are finding maximum product for all the rope lengths, from 1 to n.

And to find maximum product for each rope length we are iterating over all previous rope lengths.

So that is O(1 + 2 + 3 + ... + (n - 1)) = O(n^2).

Auxiliary Space Used:

(n).

Because we are using array of length n + 1.

Space Complexity:

O(n).

Because auxiliary space used is O(n).

other\_solution\_2.cpp:

Time Complexity:

O(n).

We are finding maximum product for all the rope lengths, from 1 to n, in constant time.

So that is O(n).

Auxiliary Space Used:

O(n).

Because we are using array of length n + 1.

Space Complexity:

O(n).

Because auxiliary space used is O(n).

optimal\_solution.cpp:

Time Complexity:

O(log(n)). (or more specifically O(log(n / 3)).)

Because we are using power function.

Auxiliary Space Used:

O(log(n)).

YES IT IS NOT O(1).

Power function is recursive hence due to recursive function call stack it will be O(log(n)).

Note that here we can use iterative power function to reduce the auxiliary space used to O(1).

But for readability purpose we have used recursive power function.

Space Complexity:

O(log(n)).

Because auxiliary space used is O(log(n)).

Other Note:

We can use direct multiplication instead of power function but its time complexity will be O(n / 3) instead of O(log(n / 3)).

// Function to find maximum product by cutting rope of length n into pieces.

long long int max\_product\_from\_cut\_pieces(int n){

// max\_product[i] = maximum possible product for i, with at least one cut.

vector<long long int> max\_product(n + 1);

// Assign maximum possible product manually for 2 <= i <= min(n, 4).

// Maximum possible product for 2 = 1 \* 1.

// Given that n >= 2, so we do not need to check if (n >= 2){

max\_product[2] = 1LL;

if (n >= 3){

// Maximum possible product for 3 = 2 \* 1.

max\_product[3] = 2LL;

}

if (n >= 4){

// Maximum possible product for 4 = 2 \* 2.

max\_product[4] = 4LL;

}

// Calculate maximum possible product using DP for 5 <= i <= n.

for (int i = 5; i <= n; i++){

/\*

Use the observation that for i >= 5, there will be at least one cut

of length 3.

1) i - 3LL => when exactly two cuts => one cut of length 3 and another

cut of length i - 3.

2) max\_product[i - 3] => when more than two cuts => one cut of length 3

and other cuts from max\_product[i - 3].

\*/

max\_product[i] = max(i - 3LL, max\_product[i - 3]) \* 3LL;

}

return max\_product[n];

}

-------------------- STOP ---------------------------

// -------------------- START ---------------------------

// Function to find a^b in O(log(b)).

// Use long long int otherwise it will overflow, for the given constraints.

long long int power(int a, int b)

{

// a^0 = 1.

if (b == 0)

{

return 1LL;

}

/\*

Suppose we want to find a^13.

a^13 = a^6 \* a^6 \* a^1.

Now instead of finding a^6 two times we can calculate it once (to speedup) and then use it.

Then multiply with remaining a.

\*/

long long int ret = power(a, b / 2);

ret = ret \* ret;

if (b % 2)

{

ret = ret \* (long long int)a;

}

return ret;

}

// Function to find maximum product by cutting rope of length n into pieces.

// Also need to use long long int otherwise it will overflow for the given constraints.

long long int max\_product\_from\_cut\_pieces(int n)

{

// Base cases.

if (n <= 3)

{

return (long long int)n - 1;

}

// Try some examples and will notice that there is fixed pattern.

// Cut the rope such that all pieces have length 3.

if (n % 3 == 0)

{

return power(3, n / 3);

}

// Cut the rope such that one piece has length 4 and rest pieces have length 3.

if (n % 3 == 1)

{

return power(3, (n - 4) / 3) \* 4LL;

}

// Cut the rope such that one piece has length 2 and rest pieces have length 3.

return power(3, (n - 2) / 3) \* 2LL;

}

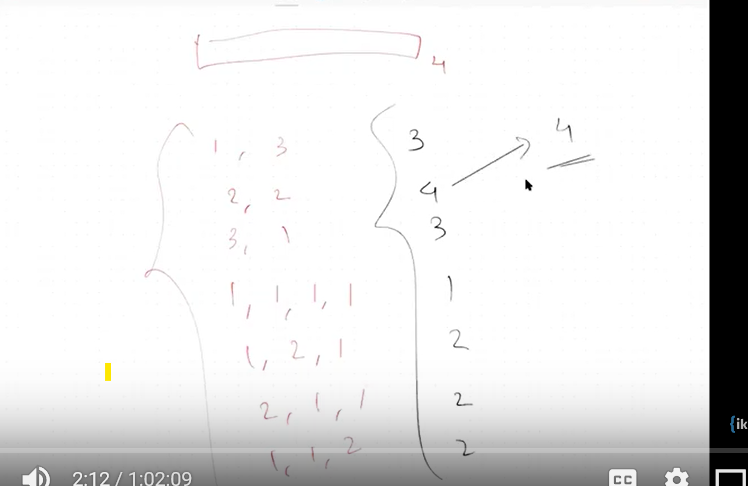
// ----------------

# Problem 2: Number of Paths in Matrix

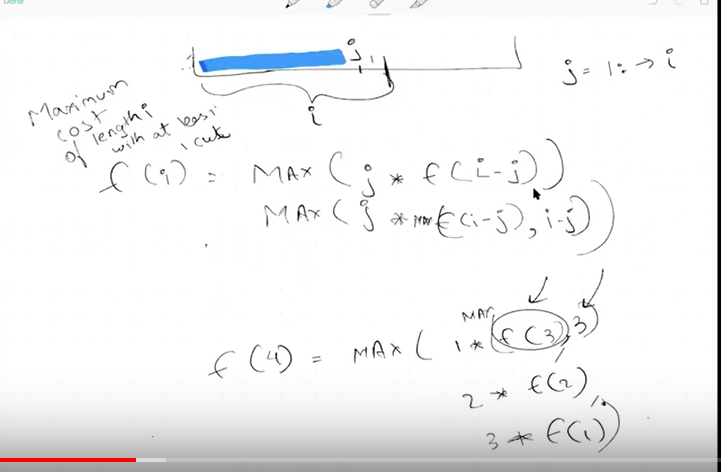
*/\*\*  
 \* Number Of Paths In A Matrix  
 \*  
 \* Consider a maze mapped to a matrix with an upper left corner at  
 \* coordinates (row, column) = (0, 0). You can only move either towards right  
 \* or down from a cell. You must determine the number of distinct paths  
 \* through the maze. You will always start at a position (0, 0), the top left,  
 \* and end up at (n-1, m-1), the bottom right.  
 \*  
 \* As an example, consider the following diagram where '1' indicates an open  
 \* cell and '0' indicates blocked. You can only travel through open cells,  
 \* so no path can go through the cell at (0, 2). There are two distinct paths  
 \* to the goal.  
 \*  
 \* There are two possible paths from the cell (0, 0) to cell (1, 3) in this matrix.  
 \* Complete the function numberOfPaths. The function must return the number of paths through the matrix, modulo (10^9 + 7).  
 \*  
 \* For example:  
 \*  
 \* 3  
 \* 3  
 \* 1 1 0  
 \* 1 1 1  
 \* 0 1 1  
 \*  
 \* Output Format:  
 \* 4  
 \*  
 \* Constraints:  
 \* 1 <= n\*m <= 2\*10^6  
 \* Each cell, matrix[i][j], contains 1, indicating it is accessible or 0,  
 \* indicating it is not accessible, where 0<=i<n and 0<=j<m.  
 \*  
 \* Input 1:  
 \* 3  
 \* 4  
 \* 1 1 1 1  
 \* 1 1 1 1  
 \* 1 1 1 1  
 \*  
 \* Sample Output 1:  
 \* 10  
 \*  
 \* Explanation 1:  
 \* There are 10 possible paths from cell (0, 0) to cell (2, 3).﻿﻿﻿  
 \*  
 \* Sample Input 2:  
 \* 2  
 \* 2  
 \* 1 1  
 \* 0 1  
 \*  
 \* Sample Output 2:  
 \* 1  
 \*  
 \* Explanation 2:  
 \* There is 1 possible path from the cell (0, 0) to cell (1, 1).  
 \*  
 \* resources/NumberOfPathsInMatrixDp.jpg  
 \* resources/NumberOfPathsInMatrixRecursion.jpg  
 \*  
 \* Recursion:  
 \*  
 \* 1) Start from (0,0), go right and down. Add them.  
 \* 2) Base cases  
 \* if (row > n - 1 || col > m - 1) return 0  
 \* if (matrix[row][col] == 0) return 0  
 \* if (row == n - 1 && col == m - 1) return 1  
 \*  
 \* Time Complexity: O(2 ^ (m + n))  
 \* Space Complexity: O(m + n)  
 \*  
 \* Dynamic Programming  
 \*  
 \* Time Complexity: O(mn) where m = number of rows and n = number of columns  
 \* Space Complexity: O(mn) where m = number of rows and n = number of columns  
 \*/*

# Video Link: <https://youtu.be/nuKiJzqnHjs>

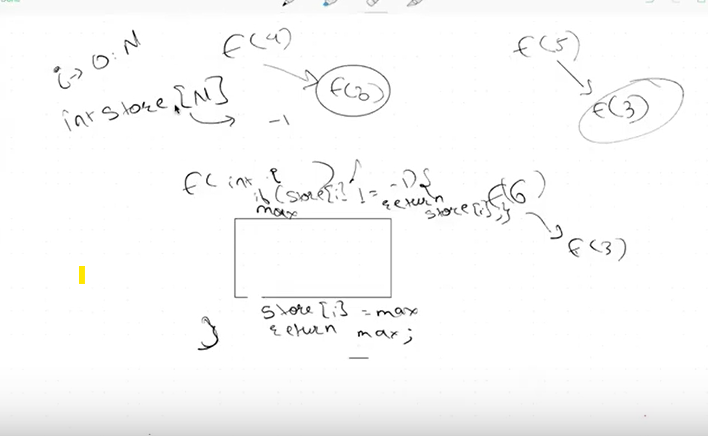
# Problem 1: Cut the rope, return the max product.



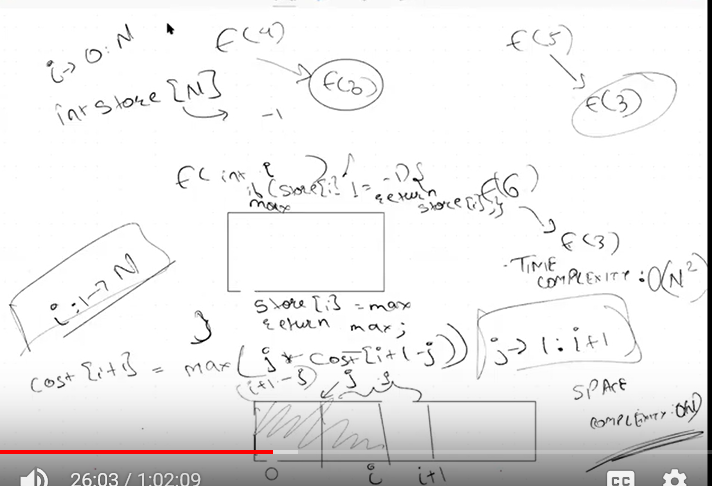
# Recursive relationship



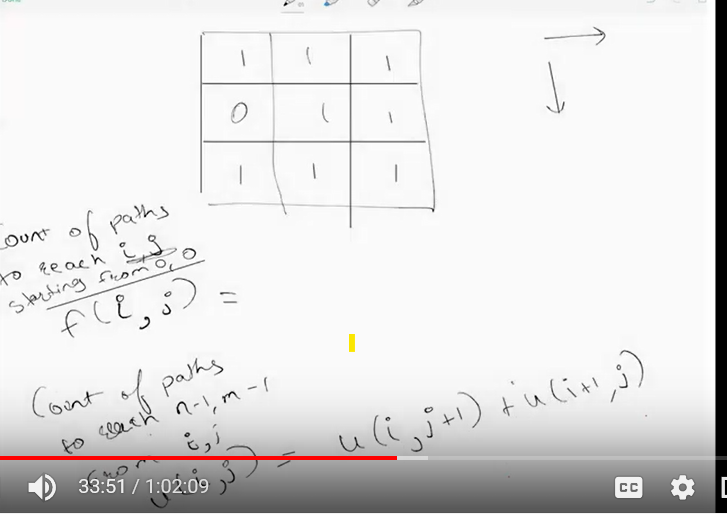
# Memorization

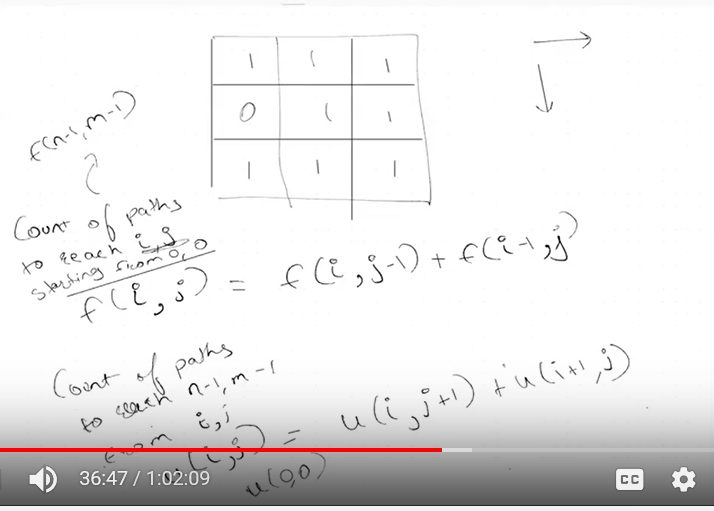


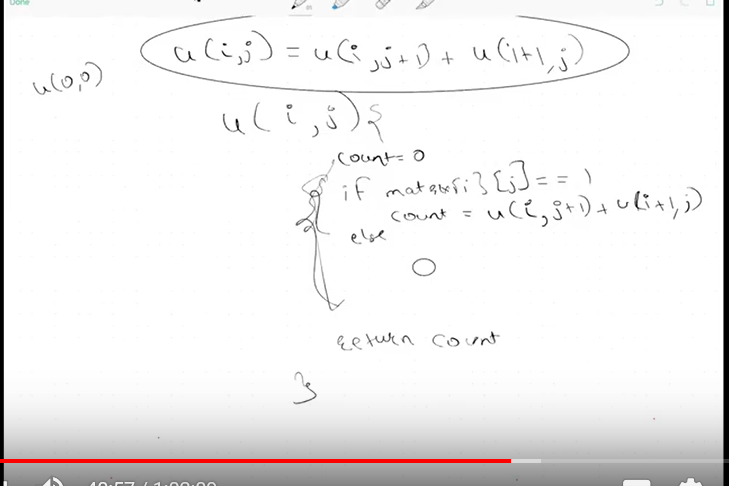
# DP Solution



# Problem 2 Count number of paths in matrix







Memorization

