CS69201: Computing Lab-1
Test-3
August 6, 2024
Time - 2:10PM - 4.40 PM

========== Instructions==========

1. In the case of user input assume only valid values will be passed as input.
2. You can use C or C++ as the programming language. **However, you are not allowed to use any STL libraries in C++**
3. Regarding Submission: For each question create a separate C file. -> <rollno>_Q1.c, <rollno>_Q2.c, <rollno>_Q3.c,<rollno>_Q4.c. Create a zip file of all these C files in the name <rollno>_T3.zip and submit it to Moodle. For example, if your roll number is 24CS60R15, then your file names will be 24CS60R15_Q1.c, 24CS60R15_Q2.c, 24CS60R15_Q3.c, 24CS60R15_Q4.c and your zip file name will be 24CS60R15_T3.zip.
4. Use the most optimal approach to solve the questions.
5. Inputs should be taken from the user through the terminal and outputs should be displayed on the terminal.

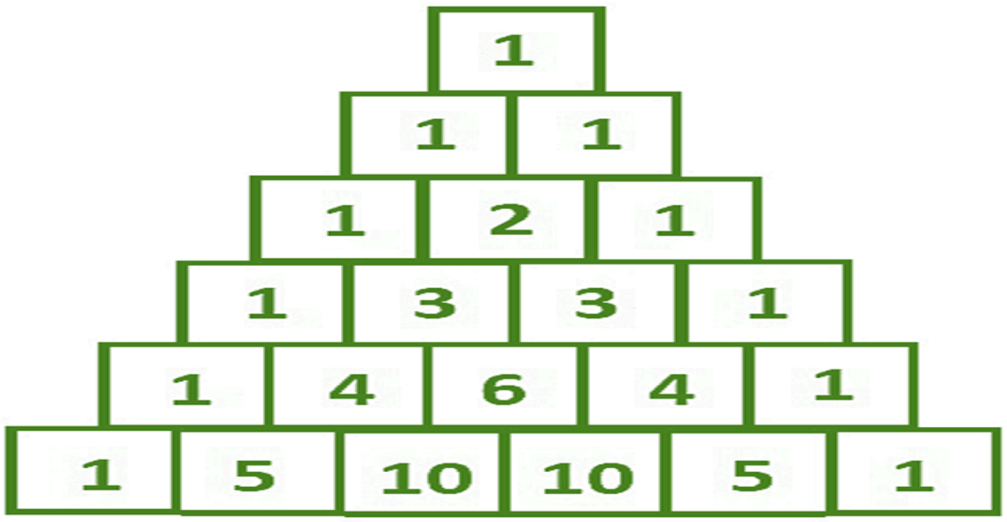
==========

Question 1

Given an integer `numRows`, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each row begins and ends with 1 and each number is the sum of the two numbers directly above it. Refer the below figure: [20]

[Note : you have to solve the question using dynamic programming]



Example :

Input : 5
Output :
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Question 2:

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. We cannot schedule a job after it passes its deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. Maximize the total profit if only one job can be scheduled at a time. Return the maximum profit sequence of jobs. [20]

Sample Input:

Job id: arr[] = {'a','b','c','d}
Profit: arr[] = {20,10,40,30}
Deadline: arr[] = {4,1,1,1}

Sample Output:
c a

Question 3

You need to construct an array with *n* elements such that each element is between 1 and *k*, inclusive. The first and last element of the array to be 1 and *x,* where *x* is a number between 1 and k, inclusive. Find the number of ways to construct the array such that consecutive positions contain different values. Since the answer may be large, only find it modulo 10^9+7. [30]

(Important Note: Solve the question using all three approaches as mentioned in the solution. Write three functions in the same C file for this question.)

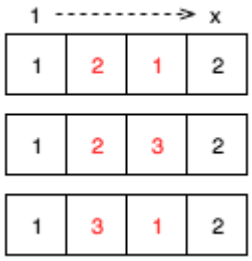
Sample Input

n = 4, k = 3, x = 2

Sample output

3

Explanation: The figure shows all the possible arrays



Solution/Hint :

Approach 1: Recursion

First we can find the number of arrays with length 1 to n and containing elements from 1 to k.
Let's assume that we know the count of arrays containing n-1 elements and end with the numbers 1 to k. Now, if we want to find the count of arrays containing n elements and ends with numbers from 1 to k, we can use the below recursion formula.

$$count(n, j) = \sum_{i=1}^{k} count(n-1, i) - count(n-1, j)$$

That is from the count of n-1, subtract the arrays that ends with j.

So for the given question, we need to find the arrays that ends with x:
count (4,2) = count(3,1)+count(3,2)+count(3,3)-count(3,2)
count(3,1) = count(2,1)+count(2,2)+count(2,3)-count(2,1)
count(3,2) = count(2,1)+count(2,2)+count(2,3)-count(2,2)
.
.
.
The base conditions are:
count(1,1) = 1, ie the count of arrays containing one element and ending with 1 is one.
count(1,2) = 0, since we want the starting element to be 1, the count of arrays that starts with 1 and ends with 2 containing one element is zero
count(1,3) = 0

Approach 2: Using 2d DP

If we analyse the recursion formula we can find the overlapping subproblem. So we can use a 2d array to store the values. The size of the 2d array will be, no of rows = n, no of columns = k and return dp[n][x].

Approach 3: Optimal approach using two 1d arrays

If we analyse the problem even more, we can notice that to find the count(n,j) we only care about the count(n-1,ends with j) and count(n-1,does not end with j). We can store this information using 2 simple 1d arrays. In the first array we will store for each value of n, the number of arrays that ends with x and in the second array we will store for each value of n, the number of arrays that does not end with x.
Let's say array A contains the count of arrays ending with x and array B contains the count of arrays that does not end with x. Then the recursion equation will look like the following.
A[i] means the count of arrays that contains i number of elements and ends with x.
B[i] means the count of arrays that contains i number of elements and does not end with x.
A[i] = B[i-1]
B[i] = B[i-1]*(k-2) + A[i-1]*(k-1)
Where i can vary from 1 to n, and the answer will be A[n].

Base condition is:
if(x=1)
    A[0] = 1
    B[0] = 0
if(x!=1)
    ? (find out by yourself).

Question 4

Hari is given n apples, indexed from 0 to n - 1. Each apple is painted with a number on it represented by an array nums. Hari is hungry and needs to eat all the apples.

If he eats the ith apple, he will get nums[i - 1] * nums[i] * nums[i + 1] points. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is an extra apple with a 1 painted on it.

Return the maximum points Hari can get by eating the apples wisely. [30]

Input:
n = 4
nums = [3, 1, 5, 8]
Output: 167
Explanation:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
points = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

Input:

2
1 5
Output: 10
Explanation:
nums = [1,5] --> [5] --> []
points = 1*1*5 + 1*5*1 = 10

Solution/Hint :

Bruteforce:

Generate all permutations for the ordering of apples and check which permutation yields the best number of points. But it will take O(n!) time. You don't need to code this.

DP Memoization approach:

Define Subproblems

For a DP solution to exist, we need to define the subproblems. Let's define the problem first as:

solve(nums, i, j)

by which I mean that hari needs to eat apples starting from index i to index j. At the beginning, they'll be 0, nums.size() -1 respectively. Let's suppose we eat the kth apple in the first chance. We will get nums[k-1] * nums[k] * nums[k+1] points. Now let's define the subproblems as:

solve(nums, i, k - 1) , solve(nums, k + 1, j)

As the apple k is already ate, we solve the subproblems from i to k -1 and k + 1 to j. But wait, what's going wrong here? The subproblem solve(nums, i, k - 1) and solve(nums, k + 1, j) are not independent since after eating kth apple, apple k - 1 and k + 1 have become adjacent and they will need each other in order to calculate the points.

So, as we saw that if we choose the kth apple to be the first one to be ate, we can't make the subproblems independent. Let's try the other way round. **We choose the kth apple as the last one to be eaten.** Now the subproblems will become independent since (k - 1)th apple and (k + 1)th apple won't need each other in order to calculate the answer. **(Try it out using pen and paper).**

Now for each k starting from i to j, we choose the kth apple to be the last one to be eaten and calculate the points by solving the subproblems recursively. Whichever choice of k gives us the best answer, we store it and return.
Important point to be noted here is that the apples in the range (i, k - 1) and (k + 1, j) will be

eaten BEFORE kth apple. **So, when Hari eats the kth apple, the profit will be nums[i - 1] * nums[k] * nums[j + 1] PROVIDED that index i - 1 and j + 1 are valid.**

In every recursive call, pick a k such that eating the kth apple at last will maximize the profit.

For example:

```
for(int k = i; k <= j; k++){

                // Eat the kth apple after eating (i, k - 1) and (k + 1, j) apples
            int temp = nums[k];

            if(j + 1 < n)  // As apple  j + 1 will become adjacent to k after eating  k + 1 to j apples
                temp *= nums[j + 1];

            if(i - 1 >= 0) // As apple  i- 1 will become adjacent to k after eating  i  to k -1 apples
                temp *= nums[i - 1];

                  // Recursively solve the left and right subproblems and add their contribution
            temp += (solve(nums, i, k - 1) + solve(nums, k + 1, j));

                  // If this choice of k yields a better answer
            ans = max(ans, temp);
        }
```

Use recursion and memoize the answer.

Expected Time Complexity: $O(n^3)$
Expected Space Complexity: $O(n^2)$

----------XXX----------