Introduction to Competitive Programming

Why Competitive Programming

- Publicly demonstrate your skills.
- Prepares you well for Technical Interviews.
- Makes you a desirable Candidate for Major Companies.
- Makes you more faster and focused.
- Helps you solve Complicated Problems.
- Guaranteed Brain Exercise.
- Teaches you how to work in Teams.
- It’s Fun :)

Problem Structure

- Paragraphs - Problem Description
- Input Format
- Output Format
- Constraints
- Sample Input and Output
- Explanation (Optional)

ATM | Problem Code: HS08TEST

All submissions for this problem are available.

Pooja would like to withdraw `X` $US from an ATM. The cash machine will only accept the transaction if `X` is a multiple of 5, and Pooja's account balance has enough cash to perform the withdrawal transaction (including bank charges). For each successful withdrawal the bank charges 0.50 $US. Calculate Pooja's account balance after an attempted transaction.

Input

Positive integer 0 < `X` <= 2000 - the amount of cash which Pooja wishes to withdraw.

Nonnegative number 0<= `Y` <= 2000 with two digits of precision - Pooja's initial account balance.

Output

Output the account balance after the attempted transaction, given as a number with two digits of precision. If there is not enough money in the account to complete the transaction, output the current bank balance.

Example - Successful Transaction

Input:

```
30 120.00
```

Output:

```
89.50
```

All Submissions

Successful Submissions +

How to Approach a Problem

- Read the Problem Description and try to understand the question.
- Analyse the Constraints and try to think of an algorithm.
- Read the input and output format.
- Code your Solution accordingly.
- Run on Sample Input and Output.
- Submit your Solution

Analysing the Constraints

- Time Limit - { 1s, 2s, 3s }
- Memory Limit - { 256 mb }
- Analysing the values of input constraints given.

Types Of Verdict

- Accepted (AC)
- Wrong Answer (WA)
- Compilation Error
- Time Limit Exceeded (TLE)
- Memory Limit Exceeded (MLE)
- Runtime Error

Program Metrics

Time Complexity

Space Complexity

Time Complexity

- We take into account that the Function is dependent on the Input size.
- We Tend to find the Worst case running time of a Program.
- Time Complexity basically tells the time elapsed as a function of the input size put in when a particular algorithm is run on a system.

Order-Analysis
(Big O-Notation)

- Consider Only the Highest Power in Case Of Polynomials.
- Remove All other Terms.
- Do Not worry about the coefficient as well!

1. Sequence of statements

statement 1;
statement 2;
...
statement k;

The total time is found by adding the times for all statements: total time = time(statement 1) + time(statement 2) + ... + time(statement k)

2. if-then-else statements

```
if (condition) {
    sequence of statements 1
 }
  else {
  sequence of statements 2
  }
```

Here, either sequence 1 will execute, or sequence 2 will execute.
Therefore, the worst-case time is the slowest of the two possibilities:
max(time(sequence 1), time(sequence 2)).

3. for loops

```
for (i = 0; i < N; i++) {
      sequence of statements
     }
```

The loop executes N times, so the sequence of statements also executes N times.

4. Nested loops

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
     }
          }
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M).

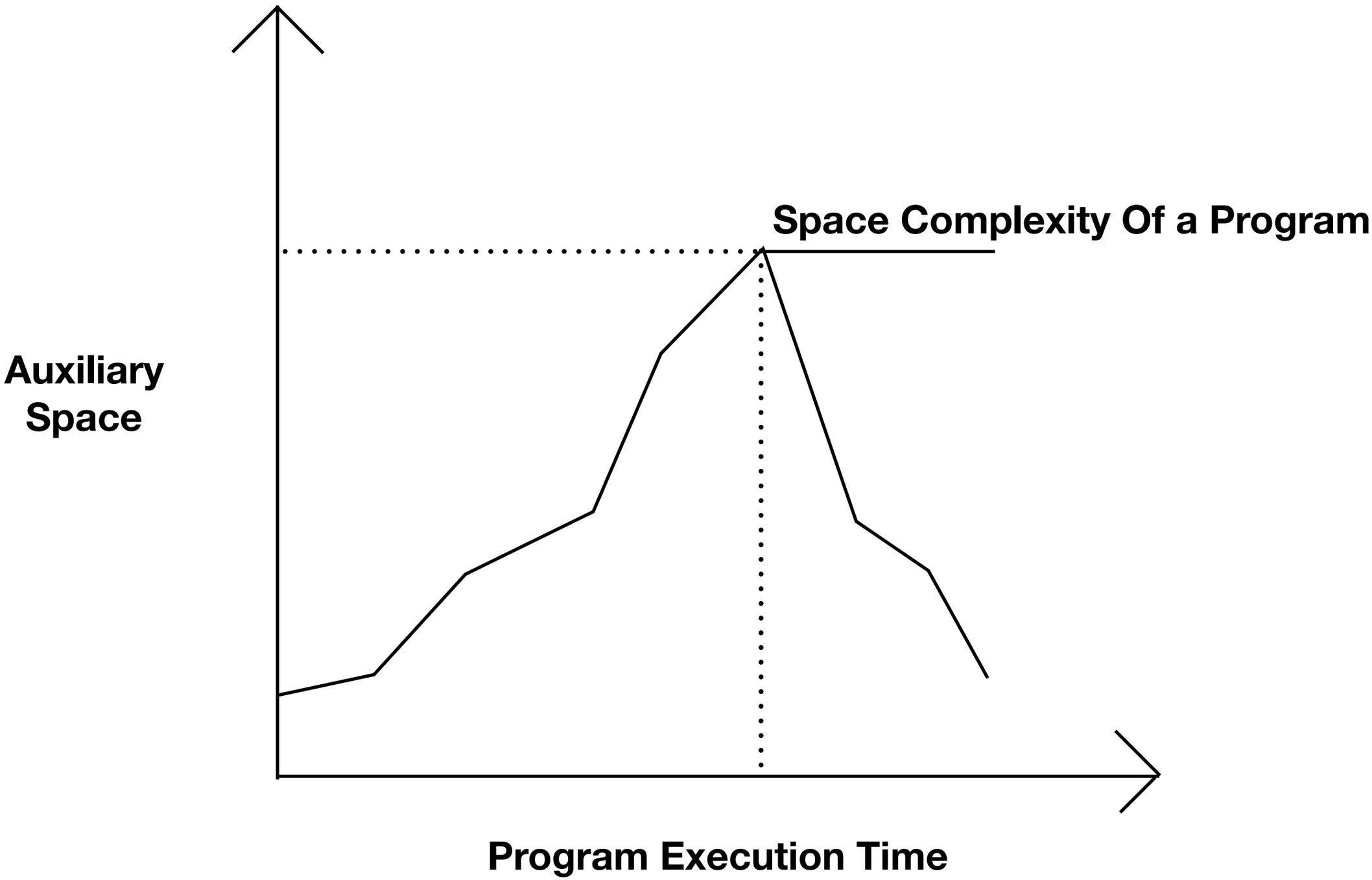
A Clean Software Practice - Black Box VS White Box Technique

Value of Constraints VS Algorithmic Time Complexity

Max Value Of constraints (N)	Max Time Complexity
• 10 ^18 • 10 ^8 • 10 ^7 • 10 ^5 • 10 ^4 • 10 ^2 • 20 • 11	• O(log(N)) • O(N) • O(N log(N)) • O(N log(N) * log(N)) • O(N * N) • O(N * N * N * N) • O(2 ^ (N)) • O(N !)

Space Complexity

- Here we talk about Auxiliary Space.
- Space Complexity stands for maximum Extra Space taken while running the algorithm.
- It is proportional to the Input Size.


Space Complexity Of a Program
Auxiliary Space
Program Execution Time

Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. By same nature it actually means that the approach that we use to solve the original problem can be used to solve smaller problems as well. So in other words in recursion a function calls itself to solve smaller problems.

Recursion AND PMI

Steps :

We can define the steps of a recursive solution as follows:

1. Base Case:
A recursive function must have a terminating condition at which the function will stop calling itself. Such a condition is known as a base case.

2. Recursive Call:
The recursive function will recursively invoke itself on the smaller version of problem. We need to be careful while writing this step as it is important to correctly figure out what your smaller problem is on whose solution the original problem's solution depends.

3. Small Calculation:
Generally we perform a some calculation step in each recursive call. We can perform this calculation step before or after the recursive call depending upon the nature of the problem.

Questions :

Finding Fibonacci Numbers

Number Of Digits

Given the code to find out and return the number of digits present in a number recursively. But it contains few bugs, that you need to rectify such that all the test cases should pass.

Input Format :
Integer n

Output Format :
Count of digits

Constraints :
1 <= n <= 10^6

Sample Input :
156
Sample Output :
3

Check Number

Given an array of length N and an integer x, you need to find if x is present in the array or not. Return true or false.
Do this recursively.

Input Format :
Line 1 : An Integer N i.e. size of array
Line 2 : N integers which are elements of the array, separated by spaces
Line 3 : Integer x

Output Format :
true or false

Constraints :
1 <= N <= 10^3

Sample Input :
3
9 8 10
8
Sample Output :
true

Power Of A Number

Write a program to find x to the power n (i.e. x^n). Take x and n from the user. You need to return the answer.

Do this recursively.

Input format :
Two integers x and n (separated by space)

Output Format :
x^n (i.e. x raise to the power n)

Constraints :
1 <= x <= 30
0 <= n <= 30

Sample Input 1 :
3 4
Sample Output 1 :
81