Introduction to Programming and Data Structures

Problem Solving Skills

Malay Bhattacharyya

Associate Professor

MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

August, 2023

Problem I

Print the following half pyramid pattern in Python.

```
*
* *
* * *
* * * *
* * * * *
```

Some random thoughts!!!

- Given the number of rows, do we really need a pair of loops?

Understanding the problem

We have to print '*' i number of times in the ith row.

Brainstorming on the problem

- Given the number of rows, we need to have controls over those.

Brainstorming on the problem

- Given the number of rows, we need to have controls over those.
- The controls that we require over each row are iterative in nature.

Brainstorming on the problem

- Given the number of rows, we need to have controls over those.
- The controls that we require over each row are iterative in nature.
- For each row, we need further controls over the columns.

The approach

1. for each row i
2. for each column j
3. Print '*' i number of times padded with spaces.
4. Go to a new line before switching to a new row.

Writing the code

Input: The number of rows `row`.

```
for i in range(row):
    for j in range(i+1):
        print('* ', end = '')
    print('')
```

Problem II

Print the following Pascal's triangle pattern in Python.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

Some random thoughts!!!

- Given the number of rows, do we really need a pair of loops?

Understanding the problem

Look at the problem closely. Don't we need to print these values in some decorated way?

```
1
11
121
1331
14641
```

Brainstorming on the problem – Idea I

Look at the patterns in the output.

```
Take: 1
i=0:  1
Take: 1   1
i=1:  1   1
Take: 1 1+1 1
i=2:  1   2   1
Take: 1 1+2 2+1 1
i=3:  1   3    3   1
Take: 1 1+3 3+3 3+1 1
i=4:  1   4    6    4   1
```

Brainstorming on the problem – Idea I

Look at the patterns in the output.

```
Take: 1
i=0:  1
Take: 1  1
i=1:  1  1
Take: 1 1+1 1
i=2:  1  2  1
Take: 1 1+2 2+1 1
i=3:  1  3   3  1
Take: 1 1+3 3+3 3+1 1
i=4:  1  4   6   4  1
```

Note: The element in row = i, column = j is obtained by adding the elements in row = i - 1, column = j - 1 and row = i - 1, column = j, except for the first and last column.

Brainstorming on the problem – Idea II

Look at the patterns in the output.

```
Take: {0 C 0}
i=0:      1
Take: {1 C 0} {1 C 1}
i=1:      1         1
Take: {2 C 0} {2 C 1} {2 C 2}
i=2:      1         2         1
Take: {3 C 0} {3 C 1} {3 C 2} {3 C 3}
i=3:      1         3         3         1
Take: {4 C 0} {4 C 1} {4 C 2} {4 C 3} {4 C 4}
i=4:      1         4         6         4         1
```

Brainstorming on the problem – Idea II

Look at the patterns in the output.

```
Take: {0 C 0}
i=0:      1
Take: {1 C 0} {1 C 1}
i=1:      1         1
Take: {2 C 0} {2 C 1} {2 C 2}
i=2:      1         2         1
Take: {3 C 0} {3 C 1} {3 C 2} {3 C 3}
i=3:      1         3         3         1
Take: {4 C 0} {4 C 1} {4 C 2} {4 C 3} {4 C 4}
i=4:      1         4         6         4         1
```

Note: The element in row = i, column = j is obtained by computing {i C j} denoting ${}^{i}C_{j} = \binom{i}{i}$.

Brainstorming on the problem – Idea III

Look at the patterns in the output.

```
Take: 1
i=0:  1
Take: 1 *(1-1+1)/1
i=1:  1     1
Take: 1 *(2-1+1)/1 *(2-2+1)/2
i=2:  1     2          1
Take: 1 *(3-1+1)/1 *(3-2+1)/2 *(3-3+1)/3
i=3:  1     3          3          1
Take: 1 *(4-1+1)/1 *(4-2+1)/2 *(4-3+1)/3 *(4-4+1)/4
i=4:  1     4          6          4          1
```

Brainstorming on the problem – Idea III

Look at the patterns in the output.

```
Take: 1
i=0:  1
Take: 1 *(1-1+1)/1
i=1:  1     1
Take: 1 *(2-1+1)/1 *(2-2+1)/2
i=2:  1     2          1
Take: 1 *(3-1+1)/1 *(3-2+1)/2 *(3-3+1)/3
i=3:  1     3          3          1
Take: 1 *(4-1+1)/1 *(4-2+1)/2 *(4-3+1)/3 *(4-4+1)/4
i=4:  1     4          6          4          1
```

Note: The element in row = i, column = j is obtained by multiplying the element in row = i, column = j - 1 with (i-j+1)/j, except for the first row and first column.

Brainstorming on the problem – Idea II vs Idea III

Note that ${}^{i}C_0 = 1$.

Further observe that ${}^{i}C_j \ / \ {}^{i}C_{j-1}$

$$= \frac{i!}{(i-j)!j!} * \frac{(i-j+1)!(j-1)!}{i!} = \frac{(i-j+1)}{j}.$$

Hence, we finally have ${}^{i}C_j = {}^{i}C_{j-1} * \frac{(i-j+1)}{j}$.

The approach

We will choose the one efficient in both time and space requirements considering the final implementation. Idea III appears to be the best.

Writing the code

Input: The number of rows `row`.

```
for i in range(row):
    for space in range(row-i+1):
        print(' ', end = '')
    for j in range(i+1):
        if j == 0 or i == 0:
            coef = 1
        else:
            coef = (coef * (i-j+1))//j
        print(' ', end = '')
        print(coef, end = '')
    print('')
```

Problem I

Write a program in Python to reverse a list of characters.

Some random thoughts!!!

- We need to swap values between those appearing one from the beginning to the end and end to the beginning.
- We need controls over the same list.
- The required controls are iterative in nature.
- Do we need a pair of loops?
- Do we need to loop through the entire list?

Understanding the problem

- It is sufficient to loop though the list from two different corners until the middle position.
- We need to think about swapping values.

Brainstorming on the problem - Idea I

Swapping the values between two variables:

```
// Let a = 10, b = 20
t = a              // t = 10, a = 10, b = 20
a = b              // t = 10, a = 20, b = 20
b = t              // t = 10, a = 20, b = 10
// Now a = 20, b = 10
```

Brainstorming on the problem - Idea II

Swapping the values between two variables:

```
// Let a = 10, b = 20
a = a * b              // a = 200, b = 20
b = a // b              // a = 200, b = 10
a = a // b              // a = 20, b = 10
// Now a = 20, b = 10
```

Brainstorming on the problem - Idea III

Swapping the values between two variables:

```
// Let a = 10, b = 20
a = a + b            // a = 30, b = 20
b = a - b            // a = 30, b = 10
a = a - b            // a = 20, b = 10
// Now a = 20, b = 10
```

Brainstorming on the problem - Idea IV

Swapping the values between two variables:

```
// Let a = 10 (i.e., 00001010), b = 20 (i.e., 00010100)
a = a ^ b              // a = 00011110, b = 00010100
b = a ^ b              // a = 00011110, b = 00001010
a = a ^ b              // a = 00010100, b = 00001010
// Now a = 20 (i.e., 00010100), b = 10 (i.e., 00001010)
```

Recall that, `0 ^ 0` and `1 ^ 1` both returns 0, whereas `0 ^ 1` and `1 ^ 0` both returns 1.

The approach

1. Loop from the beginning and the end simultaneously until the middle position.
2. Swap values using a temporary variable.

Writing the code

Input: The list `ls`.

```
i = 0
L = len(ls)
for i in range(L//2):
    t = ls[i]
    ls[i] = ls[L-i-1]
    ls[L-i-1] = t
```

Looking into the reverse_slice() function

```
static void reverse_slice(PyObject **lo, PyObject **hi){
    assert(lo && hi);
    --hi;
    while (lo < hi) {
        PyObject *t = *lo;
        *lo = *hi;
        *hi = t;
        ++lo;
        --hi;
    }
}
```

Source: https://github.com/python/cpython/blob/main/Objects/listobject.c

Problem II

Write a program in Python to find out the last repeating character in a string.

Some random thoughts!!!

- Do we need a pair of nested loops?

Understanding the problem

- For each character we have to figure out whether it is repeating or not.
- We have to browse the string from the end to the beginning.

Brainstorming on the problem

Considering time efficiency:

- Looping through the entire string for each character increases the time complexity.

Brainstorming on the problem

Considering time efficiency:

- Looping through the entire string for each character increases the time complexity.

Solution:

- Having two independent loops is better than a pair of nested loops.
- We can use some auxiliary spaces for repetition check.
- What if we compute the frequencies beforehand in an array?
- A character can be used to compute an index of an array. Note that, `Frequency[ASCII('a') - 97]` denotes the element `Frequency[0]`.

The approach

1. Initialize an array with zero values for storing frequency of characters.
2. Find out the frequency of occurrence of each character in the string.
3. Find out the last character having a frequency value more than 1.

Writing the code

Input: The string `str`

```
rc = -1
frequency = [0]*26
for i in range(len(str)):
    frequency[ord(str[i]) - 97] += 1
i -= 1
while i:
    if frequency[ord(str[i]) - 97] > 1:
        rc = i
        break
    i -= 1
if rc == -1:
    print('No repeating character')
else:
    print('Last repeating character:', str[rc])
```

Problem I

Write a Python program to simulate an environment that generates 1000 random values with the following requirements:

- Values within [0, 0.5) are generated with a probability 0.7.
- Values within (0.5, 1] are generated with a probability 0.3.
- 0.5 is never generated.

Some random thoughts!!!

- How do we generate random values?

Some random thoughts!!!

- How do we generate random values? We have `randint()` and `random()` functions.

Some random thoughts!!!

- How do we generate random values? We have `randint()` and `random()` functions.
- Do we have two different randomizations here?

Understanding the problem

- Generate random values with conditional control flows over the values generated.
- Let us simplify the problem to generate random values up to two decimal places only.
- The probability values 0.7 and 0.3 signify 7 out of 10 cases and 3 out of cases, respectively.

Brainstorming on the problem

- We can control the probability of occurrence with randomization.

Brainstorming on the problem

- We can control the probability of occurrence with randomization. Generating a random value is nothing but a probabilistic event.

Brainstorming on the problem

- We can control the probability of occurrence with randomization. Generating a random value is nothing but a probabilistic event.
- We can generate random values with randomization.

The approach

1. Generate a random value between [0, 9].
2. If the random value is less than 7 execute the steps 3-4 and exit, else execute the steps 5-6 and exit.
3. Generate a random value within [0, 49].
4. Map it to [0, 0.5)
5. Generate a random value within [51, 100].
6. Map it to (0.5, 1]

Writing the code

```
import random
for i in range(1000):
    if random.randint(0, 9) < 7:
        print(random.randint(0, 49)/100)
    else:
        print((51+random.randint(0, 49))/100)
```

Problem I

Recall that any arbitrary pair of hands (denoting `hour`, `minute`, and `second`) of a clock forms two possible angles within themselves. Write a program that takes an angle (any one of the possible two) between the other pair of hands (`minute` and `hour`) and returns whether there is any valid time satisfying the given angle or not, assuming that the `second` hand of a clock is residing at 12. Consider that an angle between a pair of hands of a clock will always remain within $[0, 2\pi]$. The input is a pair of integers (say m and n) representing the fractional angle between the `minute` and `hour` hands (in radian), i.e. the angle is $\frac{m\pi}{n}$ radian.

Brainstorming on the problem

- The angles that can be formed by the minute hand are within $[0, 2\pi]$. It can have 60 possible values.

Brainstorming on the problem

- The angles that can be formed by the minute hand are within $[0, 2\pi]$. It can have 60 possible values.
- The angles that can be formed by the hour hand are within $[0, 2\pi]$. It can have 720 possible values.

Brainstorming on the problem

- The angles that can be formed by the minute hand are within $[0, 2\pi]$. It can have 60 possible values.
- The angles that can be formed by the hour hand are within $[0, 2\pi]$. It can have 720 possible values.

The approach

1. Generate all possible angles created by the minute hand.
2. Generate all possible angles created by the hour hand.
3. Take every combination of difference of angles between the minute and hour hands.
4. Verify whether it is the same as m/n or not.

Writing the code – Inefficient version

Input: The integers `m` and `n`

```
import math
Angle = m/n
flag = 0
for i in range(60):
    AngleMin = 2*i/60
    for j in range(720):
        AngleHour = 2*j/720
        if math.fabs(AngleMin-AngleHour)==Angle or
        2-math.fabs(AngleMin-AngleHour)==Angle:
            flag = 1
            break
if flag == 1:
    print('VALID')
else:
    print('INVALID')
```

Writing the code – Efficient version

Input: The integers `m` and `n`

```
Angle = m/n
flag = 0
for i in range(60):
    for j in range(720):
        if(n*abs(12*i-j)==360*m or n*(720-abs(12*i-j))==360*m):
            flag = 1
            break
if flag == 1:
    print('VALID')
else:
    print('INVALID')
```

Problem II

Write a program in Python for computing the factorial of a number.

Brainstorming on the problem

- Iterative versus Recursive version.

Brainstorming on the problem

- Iterative versus Recursive version.
- Is there a better approach?

Brainstorming on the problem

- Iterative versus Recursive version.
- Is there a better approach?

The approach

$$n! = 2^{n-s(n)} \prod_{k \geq 1} \left(\prod_{n2^{-k} < j \leq n2^{-k+1}} j^{[j \text{ is odd}]} \right)^k,$$

where
$s(n) =$ Count of set bits in n,
$[j \text{ is odd}] = j$ if j is odd; 1 otherwise.

Source: `http://www.luschny.de/math/factorial/binarysplitfact.html`

Implementation

$s(n)$ is computed as follows:

```
def s(n):
    count = 0
    while n != 0:
        count += 1
        n &= n - 1
    return count
```

Problems

1. Given a list of non-negative digits (0-9), not necessarily distinct, write a program to find out the largest multiple of 3 that can be formed from these digits. E.g., if the sample input is 19234493219, the output will be 9994433211.
2. Write a program to print the following matrix pattern (increasing prime numbers from the centre toward the boundaries) using generic controls over print. You are not allowed to use any auxiliary space to keep the matrix. Let the line number be user input. E.g., for an input of 5 the program will return the following output.

```
3 3 3 3 3
3 2 2 2 3
3 2 1 2 3
3 2 2 2 3
3 3 3 3 3
```

Problems

3. Write a program that takes a 9×9 matrix as input and checks whether it is a valid Sudoku matrix or not. Recall that a Sudoku matrix is filled in with numbers from 1-9 with no repeated numbers in each line, horizontally or vertically. E.g., the following is a valid Sudoku matrix.

```
1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3
7 8 9 1 2 3 4 5 6
2 3 4 5 6 7 8 9 1
5 6 7 8 9 1 2 3 4
8 9 1 2 3 4 5 6 7
3 4 5 6 7 8 9 1 2
6 7 8 9 1 2 3 4 5
9 1 2 3 4 5 6 7 8
```