# Computing Laboratory

## Problem Solving Skills

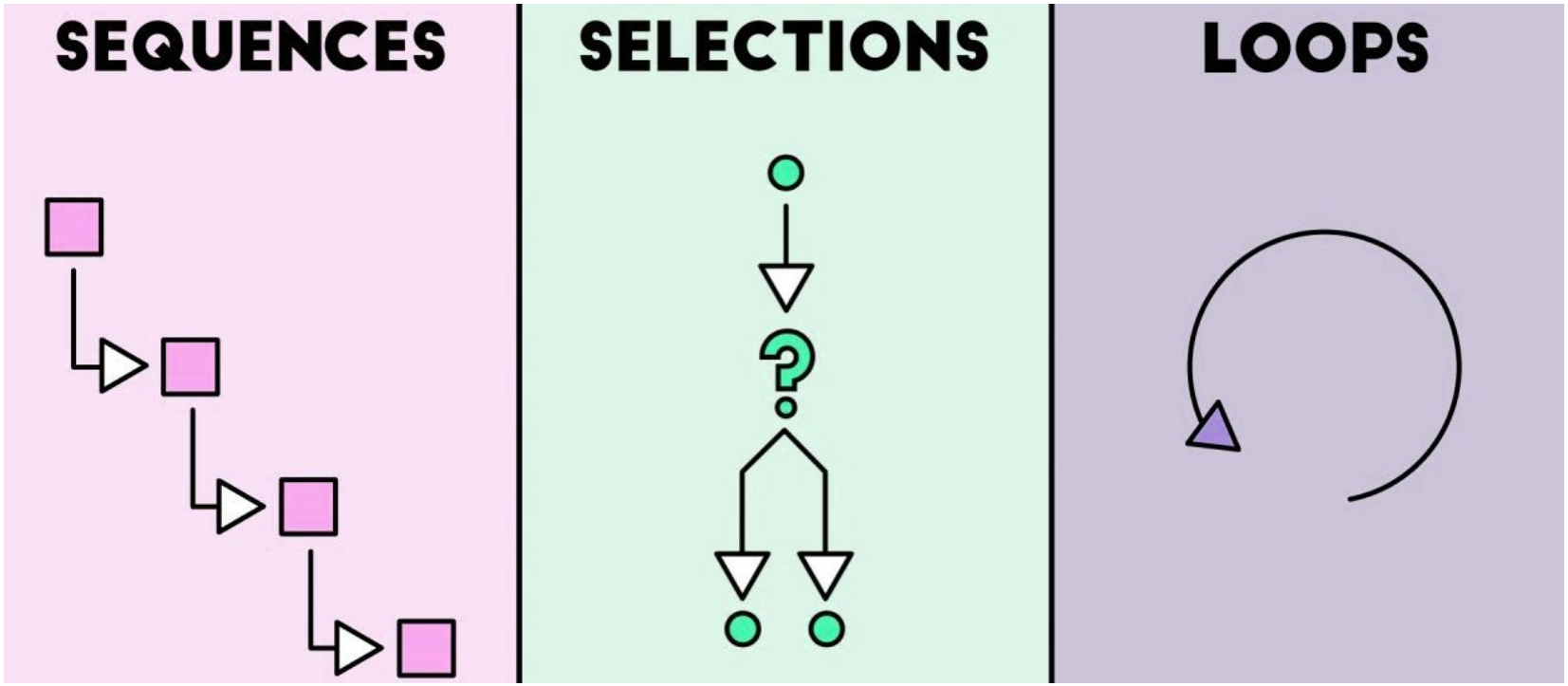
Malay Bhattacharyya

Assistant Professor

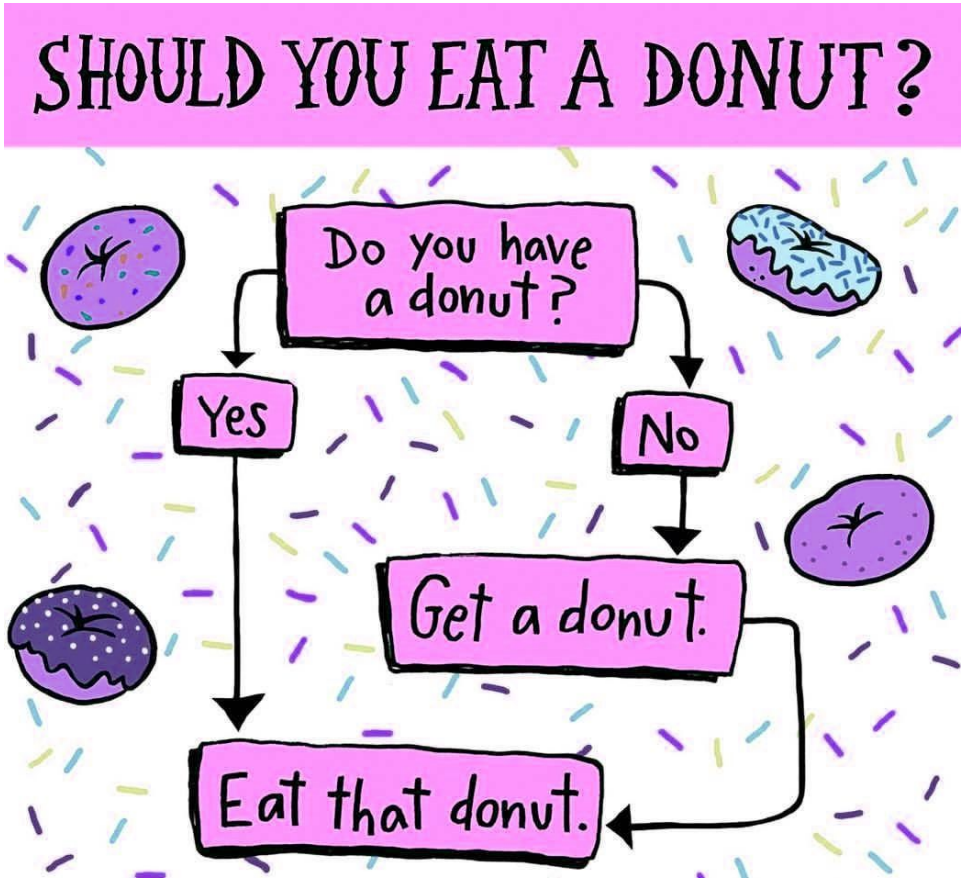
Machine Intelligence Unit
Indian Statistical Institute, Kolkata

November, 2021

# Types of control flows



# Decision making is everywhere



# Finding the maximum

Given three distinct integers as input, find out the maximum of them and show the same as output.

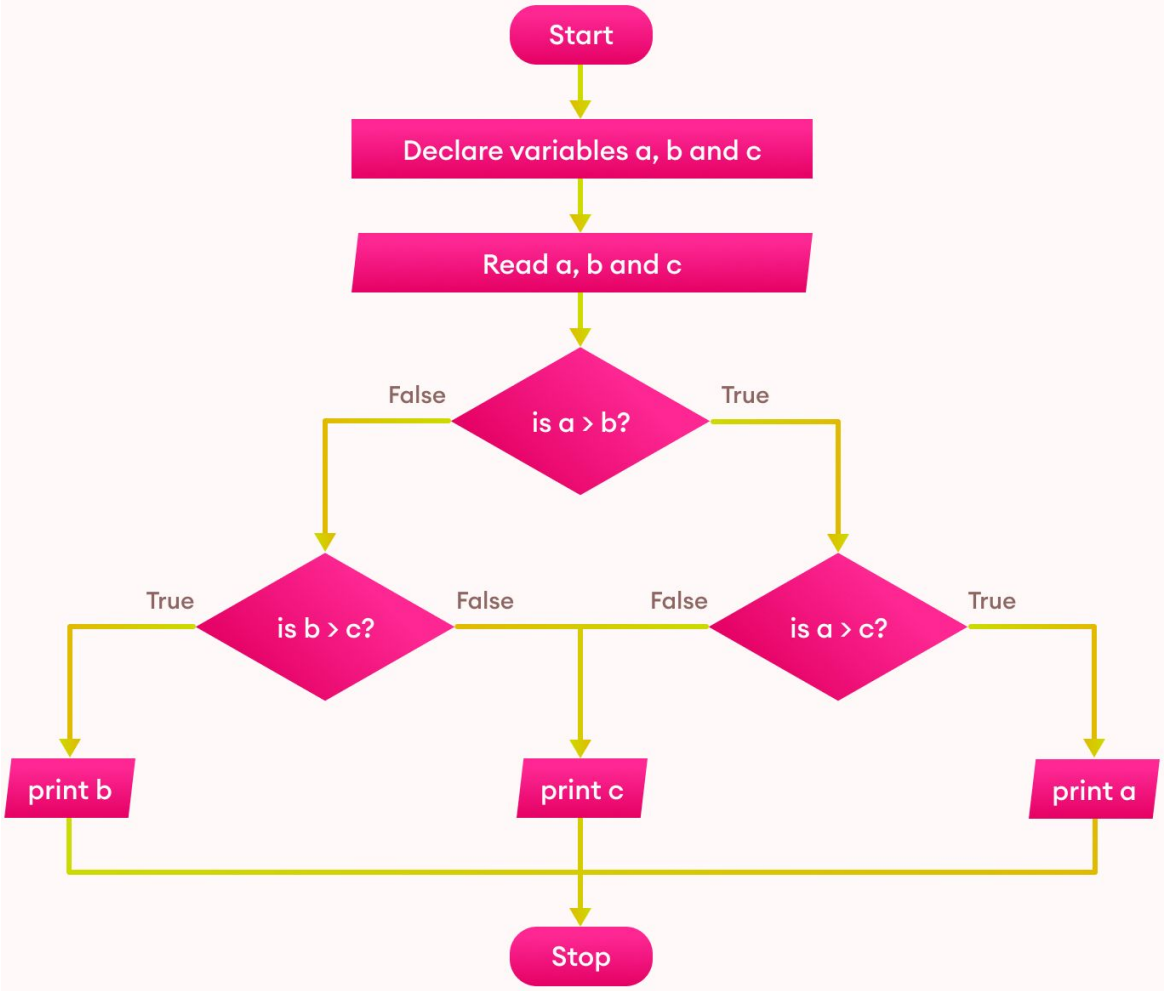
# Choosing the maximum – A naive approach

```
Inputs: a, b, c  // All are distinct values
if a > b and a > c then do // 2 comparisons
        Output a
end if
if b > a and b > c then do // 2 comparisons
        Output b
end if
if c > a and c > b then do // 2 comparisons
        Output c
end if
```

# Finding the maximum – A better approach

```
Inputs: a, b, c // All are distinct values
if a > b then do // 1 comparison
    if a > c then do // 1 comparison
        Output a
    end if
    else do
        Output c
    end else
end if
else do
    if b > c then do // 1 comparison
        Output b
    end if
    else do
        Output c
    end else
end if
```

# Finding the maximum – Using flowcharts



# Problem I

Print the following half pyramid pattern in C.

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

# 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`.

```
int i, j, row;
for(i = 0; i < row; i++){
    for(j = 0; j <= i; j++)
        printf("* ");
    printf("\n");
}
```

# Problem II

Print the following Pascal's triangle pattern in C.

```
          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`.

```
int row, coef = 1, space, i, j;
for(i = 0; i < row; i++){
    for(space = 1; space <= row - i; space++)
        printf("  ");
    for(j = 0; j <= i; j++){
        if(j == 0 || i == 0)
            coef = 1;
        else
            coef = coef * (i-j+1)/j;
        printf("%4d", coef);
    }
    printf("\n");
}
```

# Problem I

Write a program in C to reverse a string.

# 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 string.
- The required controls are iterative in nature.
- Do we need a pair of loops?
- Do we need to loop through the entire string?

# Understanding the problem

- It is sufficient to loop though the string 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 string `str`.

```
char t, *str; // Dynamically allocate memory
int i = 0, L = strlen(str);
for(i = 0; i < L/2; i++){
    t = str[i];
    str[i] = str[L-i-1];
    str[L-i-1] = t;
}
```

**Note:** Include the header file `string.h`

# Looking into the strrev() funcion

```
#include <string.h>
#include <sys/types.h>
void strrev(unsigned char *str)
{
    int i;
    int j;
    unsigned char a;
    unsigned len = strlen((const char *)str);
    for (i = 0, j = len - 1; i < j; i++, j--)
    {
        a = str[i];
        str[i] = str[j];
        str[j] = a;
}
}
```

# Problem II

Write a program in C 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 as an index of an array. Note that, `Frequency['A']` denotes the element `Frequency[65]`.

# The approach

1. Initialize an array with zero values (dynamically with a `calloc()` function) 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`

```
char *str, *frequency; // Dynamically allocate memory
int i, rc = -1;
for(i = 0; *(str + i); i++)
    frequency[*(str + i)]++;
while(--i){
    if(frequency[*(str + i)] > 1){
            rc = i;
            break;
    }
}
if(-1 == rc)
    printf("No repeating character");
else
    printf("Last repeating character: %c", *(str + rc));
```

# Problem I

Write a C 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 `rand()` and `srand()` functions.

# Some random thoughts!!!

- How do we generate random values? We have `rand()` and `srand()` 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

```
int i;
srand(time(0));
for(i = 0; i<1000; i++){
    if(rand()%10 < 7)
        printf(" %.2f ", (float)(rand()%50)/100);
    else
        printf(" %.2f ", (float)(51+rand()%50)/100);
}
```

**Note:** Include the header file `stdlib.h`

# 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

```
int m, n, i, j;
float Angle, AngleMin, AngleHour;
scanf("%d %d", &m, &n);
Angle = (float)m/n;
for(i=0;i<60;i++){
    AngleMin = (float)2*i/60;
    for(j=0;j<720;j++){
        AngleHour = (float)2*j/720;
        if(fabs(AngleMin-AngleHour)==Angle ||
                2-fabs(AngleMin-AngleHour)==Angle){
            printf("VALID");
            exit(1);
        }
    }
}
printf("INVALID");
```

**Note:** Include the header file `stdlib.h`

# Writing the code – Efficient version

```
int m, n, i, j;
scanf("%d %d", &m, &n);
for(i=0;i<60;i++){
    for(j=0;j<720;j++){
        if(n*abs(12*i-j)==360*m || n*(720-abs(12*i-j))==360*m){
            printf("VALID");
            exit(1);
        }
    }
}
printf("INVALID");
```

**Note:** Include the header file `stdlib.h`