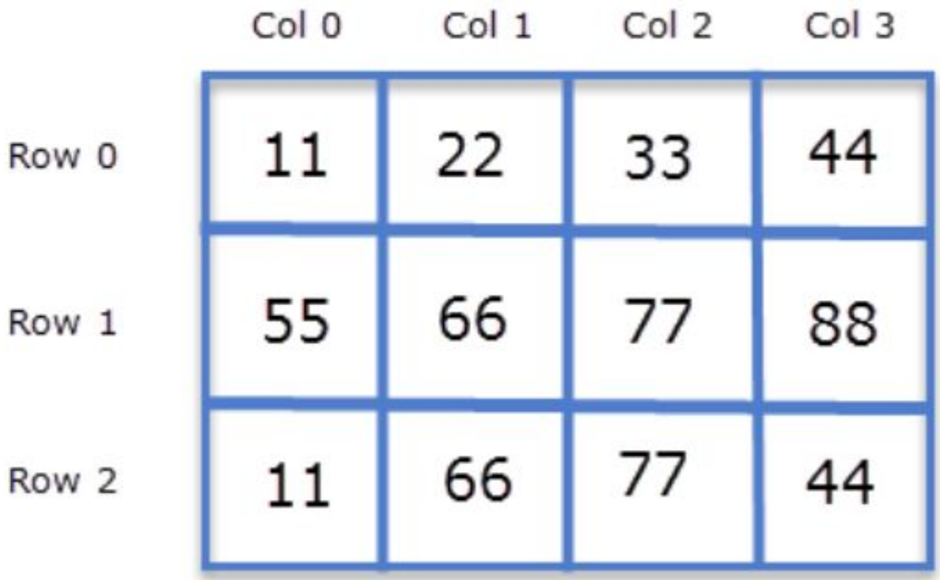
Lecture 8

DYNAMIC SAFE ARRAYS LAST THOUGHTS & INTRODUCTION TO RECURSION

September 21, 2021
Tuesday

2D Vs 1D ARRAY

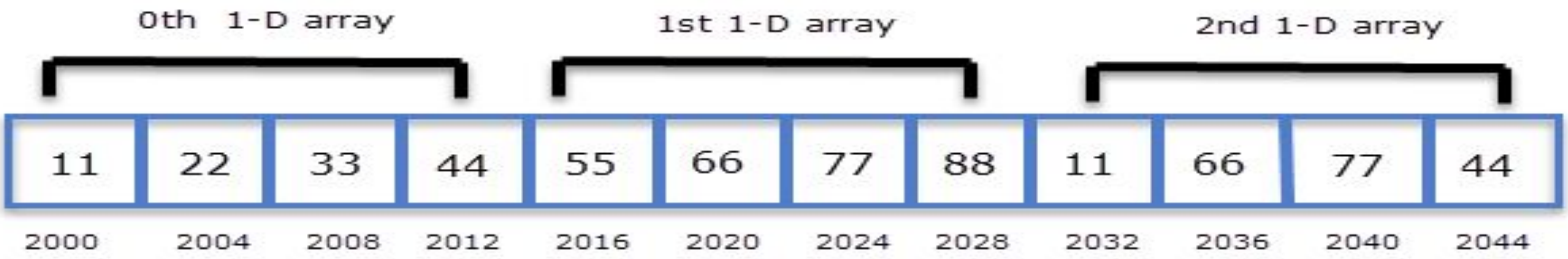
- We generally visualize a 2D array as a Matrix or Table.
- We refer to 2D arrays as composition of rows and columns.
- However, **Memory is Linear** and there is no such thing as row and column



	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44

2D Vs 1D ARRAY

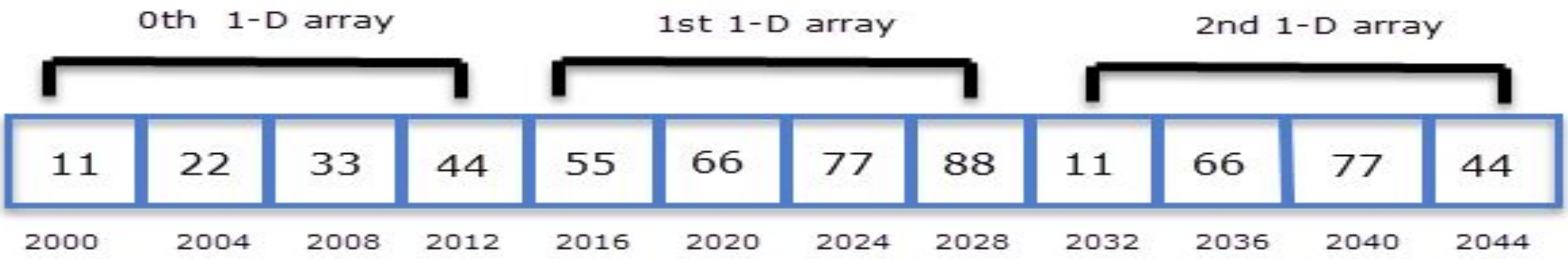
- Statically defined 2D arrays are stored in **row-major order** as 1D array in memory.
- The order dictates that all elements of row 0 should be stored first, followed by all the elements of row 1 and so on.



11	22	33	44	55	66	77	88	11	66	77	44
2000	2004	2008	2012	2016	2020	2024	2028	2032	2036	2040	2044

2D Vs 1D ARRAY

- We can still utilize the 2D indexing conventions to read/write on this 1D array.
- The order dictates that all elements of row 0 should be stored first, followed by all the elements of row 1 and so on.



11	22	33	44	55	66	77	88	11	66	77	44
2000	2004	2008	2012	2016	2020	2024	2028	2032	2036	2040	2044

FLATTENED 2D ARRAY | CONSTRUCTOR

```
DynamicArrayFlat (int r, int c ) : n_rows ( r ), n_cols ( c ), size (r * c), p2fa (NULL) {

    p2fa = new (nothrow) int [n_rows * n_cols) { };

    if (!p2fa) {
        // Memory allocation and initialization successful.
    } else {
        // Memory allocation failed, throw exception
        exit(EXIT_FAILUE);
    }
}

~DynamicArrayFlat ( ) {

    delete [ ] p2fa;
}
```

FLATTENED 2D ARRAY | ACCESS OPERATOR

```
int& operator ( ) (int row_index, int col_index) {
    if ( row_index < row_lower_bound || row_index >= row_upper_bound ||
         col_index < col_lower_bound || col_index >= col_upper_bound ) {
            // throw Index Out Of Bound Exception
    }

    return p2fa [ row_index * n_cols + col_index];
}
```

FLATTENED 2D ARRAY | GET

ONLY FUNCTION DEFINITION HAS CHANGED BODY IS SAME AS ACCESS OPERATOR

```
int get ( ) (int row_index, int col_index) {
    if ( row_index < row_lower_bound || row_index >= row_upper_bound ||
        col_index < col_lower_bound || col_index >= col_upper_bound ) {
          // throw Index Out Of Bound Exception
    }

    return p2fa [ row_index * n_cols + col_index];
}
```

FLATTENED 2D ARRAY | SET

```
void set ( ) (int row_index, int col_index, int value) {
        if ( row_index < row_lower_bound || row_index >= row_upper_bound ||
             col_index < col_lower_bound || col_index >= col_upper_bound ) {
               // throw Index Out Of Bound Exception
        }

        p2fa [ row_index * n_cols + col_index] = value;
}
```

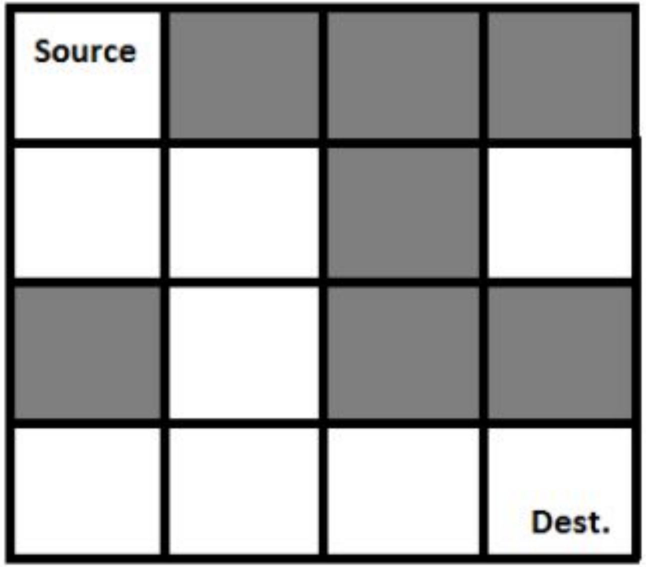
FLATTENED 2D ARRAY | PUSHBACK ROW

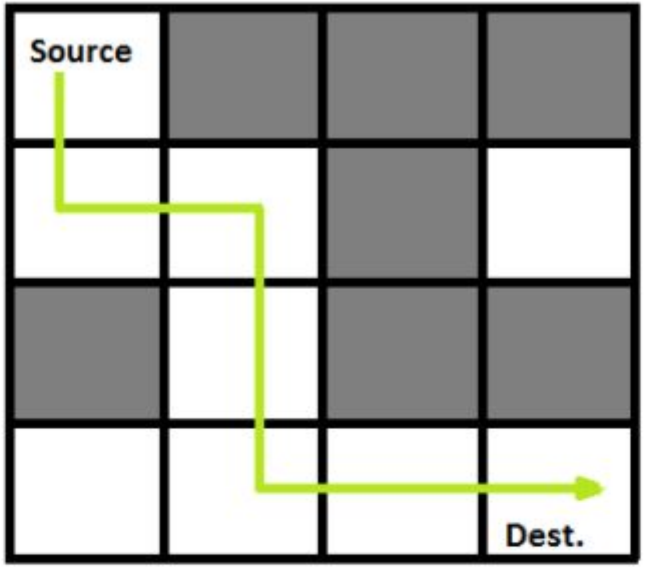
```
void pushback_row ( int row [ ], int row_cols ) {
    if (row_cols = n_cols ){
        // row size mismatch throw an exception
    }
    int* newArray = new(nothrow) int [ size+ n_cols + growth_factor ] { };
    for (int i = 0; i < size; i++)
        newArray [ i ] = p2fa [ i ];
    for (int j = size; j < size + n_cols; j++)
        newArray [ j ] = row [ j - size];
    delete [ ] p2fa;
    p2fa = newArray;
    n_rows++;
    size = n_rows * n_cols;
}
```

FLATTENED 2D ARRAY | PUSHBACK COLUMN

```
void pushback_row ( int col [ ], int col_rows ) {
    int* newArray = new(nothrow) int [ size+ n_rows ] { };
    for (int i = 0; i < size + n_rows; i++) {
        if ( ( ( i + 1) % (n_cols + 1) ) == 0 ) {
            newArray [ i ] = col [ i ];
        } else {
            newArray [ i ] = p2fa [ i ];
        }
    }
    delete [ ] p2fa;
    p2fa = newArray;
    n_rows++;
    size = n_rows * n_cols;
}
```

MAZE WITH A UNIQUE SOLUTION





MAZE WITH A UNIQUE SOLUTION

0	1	1	1
0	0	1	0
1	0	1	1
0	0	0	0

MAZE WITH A UNIQUE SOLUTION

1. Given
   a. Start is (0, 0) Index.
   b. Destination is (n_rows, n_cols)

2. Algorithm
   a. Check if start is blocked (return false);   // hint: if (maze [row] [col] == 1)
   b. Check if destination is reached (return true);
      i. // hint if (row == n_rows && col == n_cols)
   c. Check if right path is open (move right);
      i. //hint if(maze [row] [col+1] == 0)
      ii. Check if down path is open (move down);
         1. return false;

SOME INTERESTING PROBLEMS TO THINK ABOUT

- Finding common elements in 2 Arrays.
- Finding repeating elements in 2 or 3 Dynamic Safe Arrays.
- Finding Lower and Upper Bound of a Given Value.
- Merging 2 Dynamic Safe Arrays with only unique items (INTERSECTION).
- Merging 2 Dynamic Safe Arrays with all items (UNION).
- Find more problems on internet, books exercises etc and practice to strengthen your concepts.

RECURSION

WHAT IS RECURSION

A function call in which function being called is the same as the one making the call.

RECURSION

- In C++ any function can invoke another function.
    - A function can even invoke itself.
    - When a function invokes itself, it makes a recursive call.
- Recursive by definition means
    - Having the characteristics of coming up again or repeating.
- When a function directly calls itself, we refer to it as **Direct Recursion.**
- When a chain of two or more function returns to the same function that originated the chain, we refer to it as **Indirect Recursion.**

RECURSION

- Recursion offers simple and elegant solutions to the problems which require iteration.
- However, these solutions can be less efficient than iterative solutions to the same problem.
- Some programming languages does not offer recursion
  - FORTRAN, BASIC and COBOL
- C++ is more generous and offers both
  - Iterative
  - Recursive

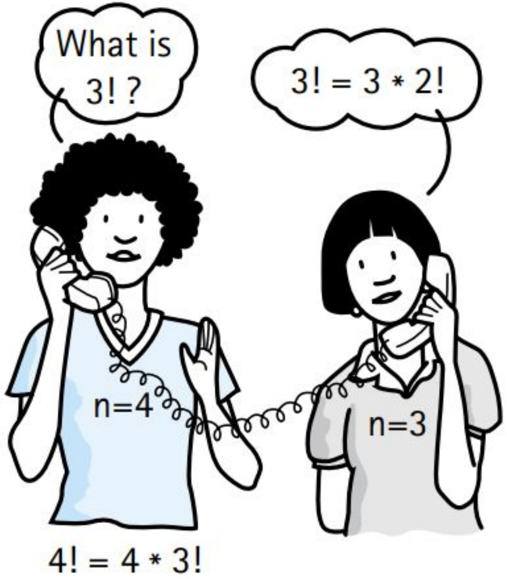
CLASSIC EXAMPLE OF RECURSION

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1) * (n-2) * \cdots * 1, & \text{if } n > 0 \end{cases}$$

- Recursive Definition
    - A definition in which something is defined in terms of a smaller version of itself.
    - 4! = 4 * 3 * 2 * 1 = 24

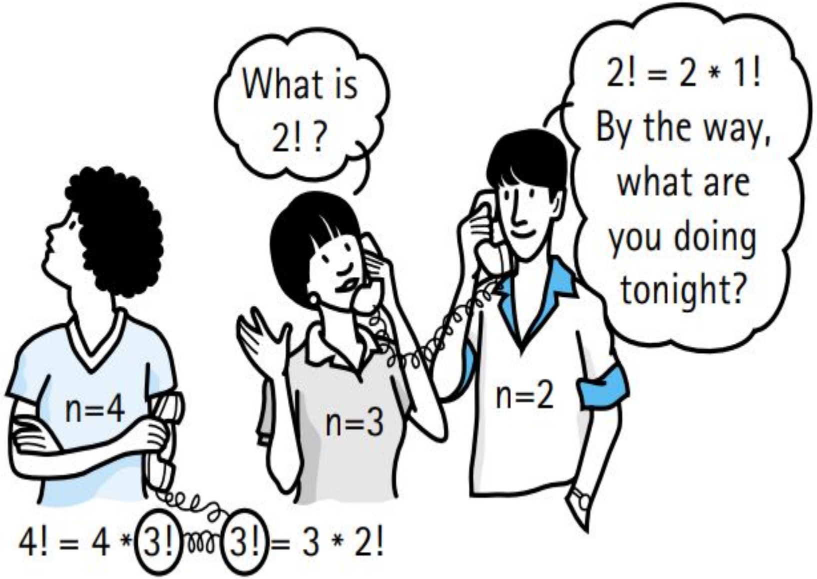
RECURSION | THE FACTORIAL

- What is 4!
  - 4! = 4 * (4 - 1)! = 4 * 3!



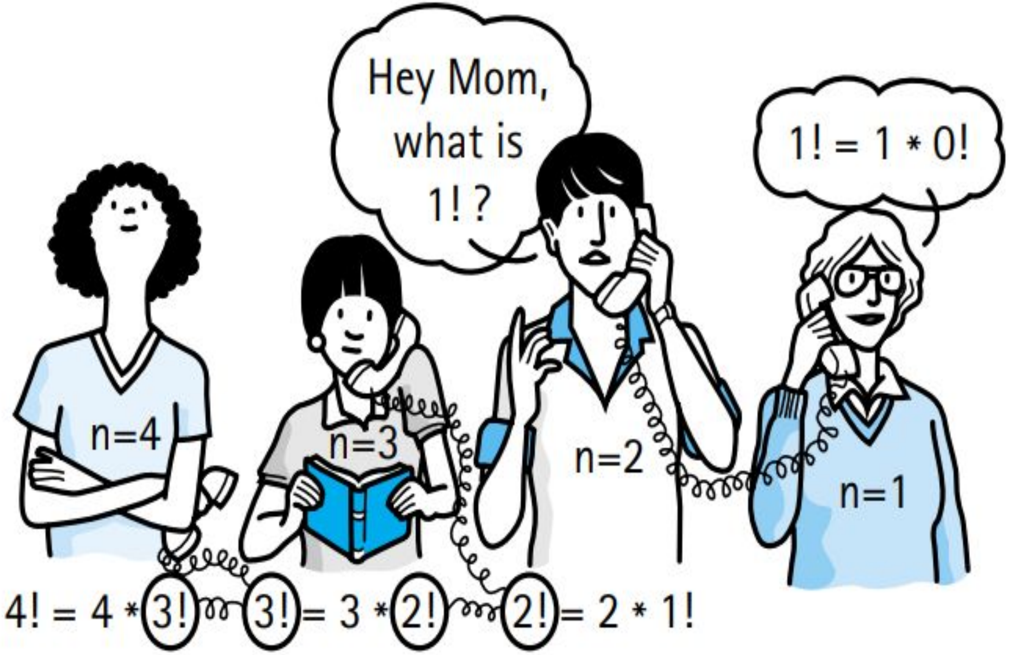
RECURSION | THE FACTORIAL

- What is 4!
  - 3! = 3 * (3 - 1)! = 3 * 2!



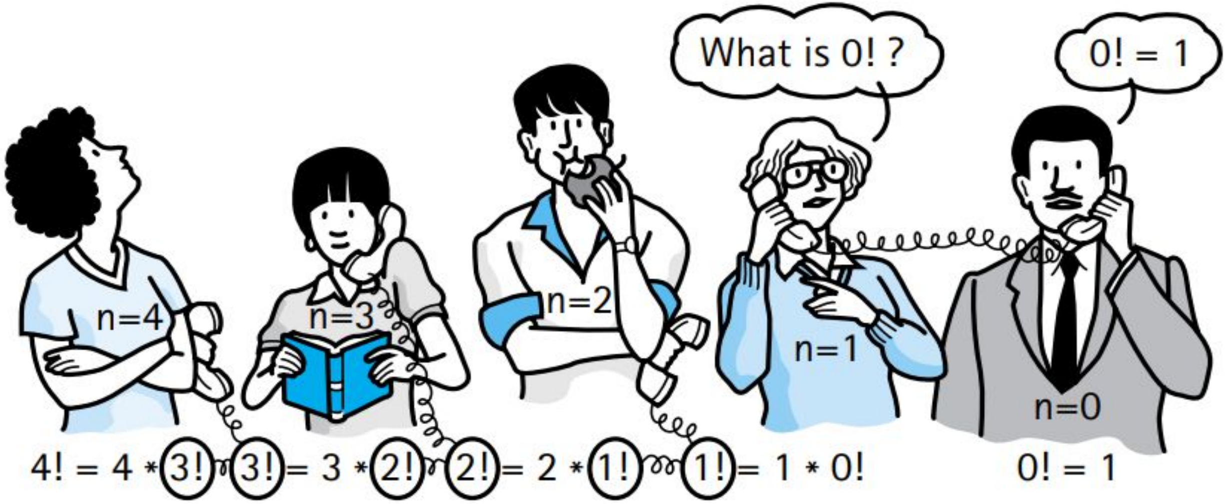
RECURSION | THE FACTORIAL

- What is 4!
    - 2! = 2 * (2 - 1)! = 2 * 1!

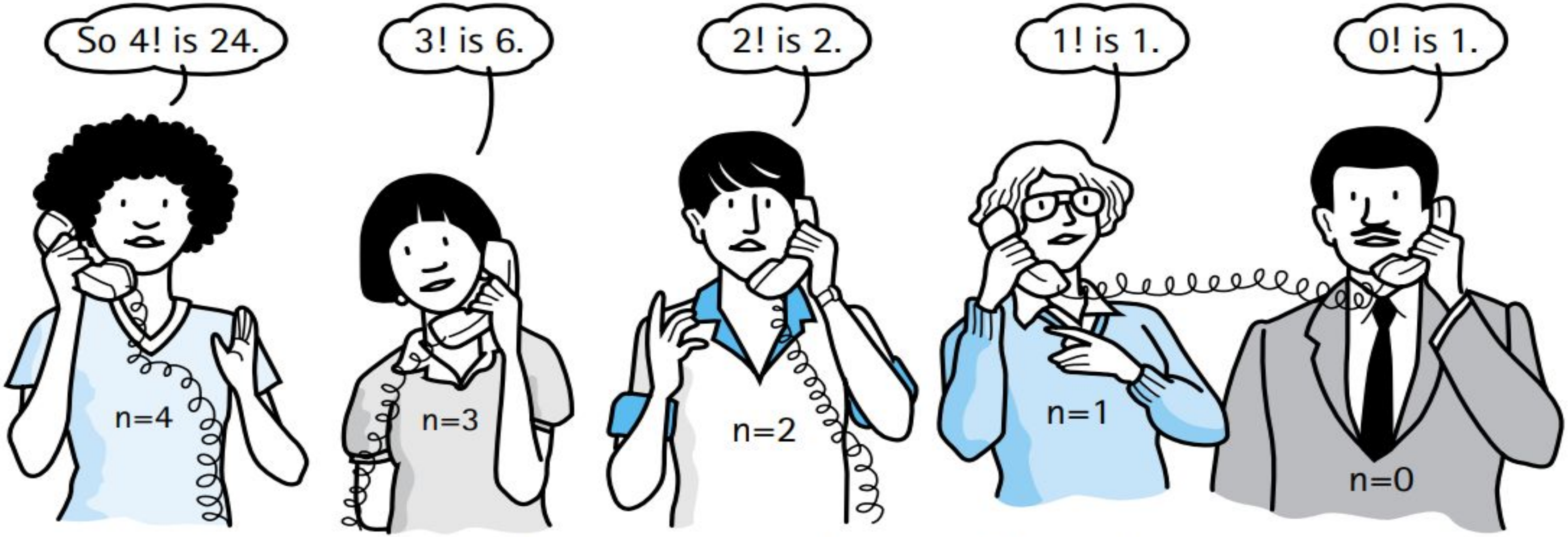


RECURSION | THE FACTORIAL

- What is 4!
  - 1! = 1 * (1 - 1)! = 1 * 0!



RECURSION | THE FACTORIAL



4! = 4 * 3! =
4 * 6 = 24

3! = 3 * 2! =
3 * 2 = 6

2! = 2 * 1! =
2 * 1 = 2

1! = 1 * 0! =
1 * 1 = 1

0! = 1

RECURSION | COMPONENTS

- Base Case
  - The case for which the solution can be stated non recursively.
  - Terminating condition.
- Recursive Case
  - The case for which the solution is expressed in terms of smaller version of itself.
- Recursive Algorithm
  - A solution that is expressed in terms of
    - smaller instances of itself
    - has a base case

RECURSION | FACTORIAL

```
// Precondition: number is non-negative.
// Post: function value = factorial of number.


int Factorial (int number) {
        if (number == 0)
                return 1;
        else
                return number * Factorial (number - 1);


}
```

RECURSION | VERIFYING

- Base Case Question
  - Will the recursion end?
  - Will the control reach base case?
- Smaller Case Question
  - Does each recursive call leads to the smaller case of the original problem?
  - Will this lead to base case?
- General Case Question
  - Assuming first two conditions are met.
  - Does the function as a whole works correctly?

RECURSION | WRITING RECURSION

- Get an exact definition of the problem to be solved.
    - This is the first step in solving any programming problem.
- Determine the size of the problem to be solved on this call to the function.
- Identify and solve the base case.
    - This ensures the yes answer to the base case question.
- Identify and solve the general case.
    - This ensures the yes answer to the smaller instance and general case question.

RECURSION | VALUE IN THE LIST

- Get an exact definition of the problem to be solved.
    - This is the first step in solving any programming problem.
- Determine the size of the problem to be solved on this call to the function.
- Identify and solve the base case.
    - This ensures the yes answer to the base case question.
- Identify and solve the general case.
    - This ensures the yes answer to the smaller instance and general case question.

RECURSION | VALUE IN THE LIST

- Get an exact definition of the problem to be solved.
  - This is the first step in solving any programming problem.
- Determine the size of the problem to be solved on this call to the function.
- Identify and solve the base case.
  - This ensures the yes answer to the base case question.
- Identify and solve the general case.
  - This ensures the yes answer to the smaller instance and general case question.

RECURSION | COMBINATIONS

```
int Combinations ( int group, int members) {
        if (members == 1)
                return group;
        else if (members == group)
                return 1;
        else
                (Combinations (group - 1, members - 1) + Combinations (group - 1, members) );

}
```