Reduction

- *Reduction* is the technique of solving a problem with the help of **another problem**.
- *Reduction* reduces the main problem into one or more problems.
- For example: $perm(n,r) = \frac{fact(n)}{fact(n-r)}$

Recursion

- *Recursion* is like reduction when the reduction reduces the problem in terms of **itself**!
- *Recursion* is a problem-solving technique in which problems are solved by reducing them to **smaller problems of the same form**.
- In programming, recursion simply means that **a function will call itself**.
- Recursion simplifies the problem by making the computer do more work, so that you can do less work.
- The structure of recursive functions is typically like the following:

```
recursiveFunction(){
    if(test for simple case){
        Compute the solution without recursion
    }
    else{
        Break the problem into subproblems of the same form
        Call recursiveFunction() on each subproblem
        Reassemble the results of the subproblems
    }
}
```

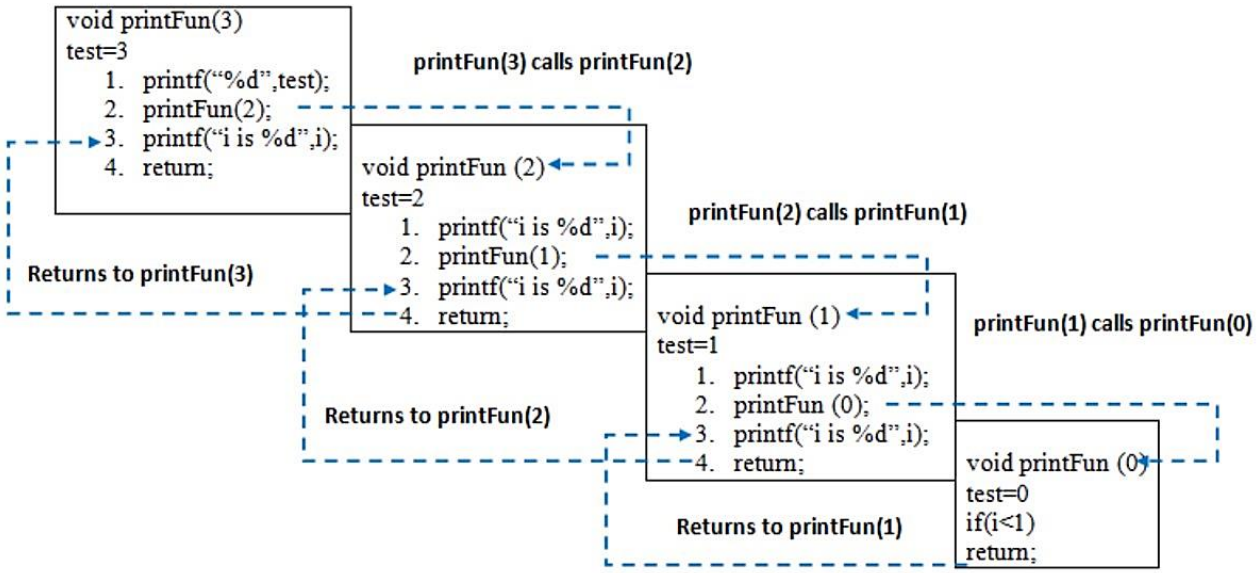
- Every recursive algorithm involves at least two cases:
  - **base case:**
    - The simple case; an occurrence that can be answered directly.
    - The case that recursive calls reduce to.
  - **recursive case:**
    - A more complex occurrence of the problem that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem.

- **3 musts** of Recursion:
  1. Your code must have a case for all valid inputs.
  2. You must have a base case that makes no recursive calls.
  3. When you make a recursive call, it should be to a simpler instance and make forward progress towards the base case.

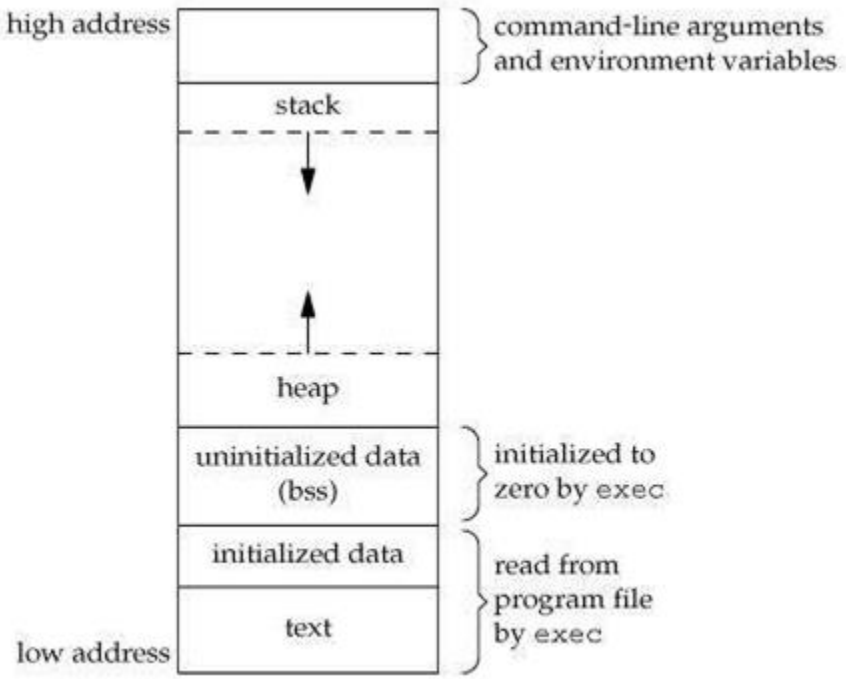
- To successfully apply recursion to a problem, you must be **able to break the problem down into subparts**, at least one of which is similar in form to the original problem. There may be many ways to break a problem down into subproblems such that recursion is useful. It is up to the programmer to determine which decomposition is best.

- **Thinking about recursion:**
  - Think of recursion as working via the power of *wishful thinking*. Each recursive call is the wish you are making and your wish will be granted.
  - Think of recursion as the CEO of a corporation tells the vice-president to perform some task, the CEO doesn't worry about how the task is accomplished; he just relies on the vice-president to get it done.
  - Also, you can think about recursion is to pretend that a recursive call is actually a call to a different function, written by somebody else, that performs the same task that your function performs.

- While the recursive call is executing, the top-level call sits there waiting for the recursive call to terminate. This means that execution doesn't halt when a recursive call finds itself at the base case; once the recursive call returns, the top-level call then continues to execute.



- **Memory allocation:** When any function is called from the **main()**, the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.



- **Types of recursion:**
  - **Direct Recursion –** A function is directly recursive if it contains an explicit call to itself.

```
int foo(int x) {
   if (x <= 0) return x;
   return foo(x - 1);
}
```

  - **Indirect Recursion –** A function (foo) is indirectly recursive if it contains a call to another function (bar) which ultimately calls itself (foo).

```
int foo(int x) {

   if (x <= 0) return x;

   return bar(x);
}

int bar(int y) {
   return foo(y - 1);
}
```

  - **Tail Recursion –** A recursive function is said to be tail recursive if there are no pending operations to be performed on return from a recursive call.

```
///non-tail-recursive
int fact (int n) { /* n >= 0 */

   if (n == 0)  return 1;
   return n * fact(n - 1);
}
```

```
///tail-recursive
int fact_aux(int n, int result) {
   if (n == 1) return result;
   return fact_aux(n - 1, n * result)
}


int fact(n) {
    return fact_aux(n, 1);
}
```

  - **Linear Recursion –** A recursive function is said to be linearly recursive when no pending operation involves another recursive call to the function.

```
int fact(int n) { /* n >= 0 */

   if (n == 0)  return 1;
   return n * fact(n - 1);
}
```

  - **Tree Recursion –** A recursive function is said to be tree recursive (or non-linearly recursive) when the pending operation does involve another recursive call to the function.

```
int fib(int n) { /* n >= 0 */

   if (n == 0) return 0;
   if (n == 1) return 1;

   return fib(n - 1) + fib(n - 2);
}
```

Recursion Practice List

1.1 Find out the maximum value from an array of integers.
1.2 Find out the minimum value from an array of integers.

2.1 Calculate the sum of all the integers from an array.
2.2 Calculate the product of all the integers from an array.
2.3 Calculate the power of an integer (a^b).
2.4 Count the frequency of an integer in an array.

3.1 Print an array in forward order.
3.2 Print an array in reverse order.
3.3 Reverse print a given string.

4.1 Show the digits of an integer in forward order.
4.2 Show the digits of an integer in reverse order.
4.3 Calculate the sum of the digits of an integer.

5.1 Convert a Decimal number into Binary number.
5.2 Convert a Decimal number into Octal number.

6.1 Check for palindrome from a given string.
6.2 Check whether an integer is prime or not.

7.1 Calculate the GCD of two numbers.
7.2 Calculate the LCM of two numbers.

8.1 Calculate the factorial of n.

Recursive formula: $$fact(n) = \begin{cases} 1;\ n = 0 \\ n * fact(n-1);\ n > 0 \end{cases}$$

8.2 Print Hailstone sequence from n to 1.

Recursive formula: $$\text{hailstone(n)} = \begin{cases} 1; n = 1 \\ hailstone(\frac{n}{2}); n\ is\ even \\ hailstone(3n+1); n\ is\ odd \end{cases}$$

9.1 Find out the n^{th} Fibonacci number.
First few Fibonacci numbers for n = 0, 1, 2, 3, 4, … are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … etc.

Recursive formula: $$fib(n) = \begin{cases} 0;\ n = 0 \\ 1;\ n = 1 \\ fib(n-1) + fib(n-2); n > 1 \end{cases}$$

9.2 Find out the n^{th} Catalan number.
First few Catalan numbers for n = 0, 1, 2, 3, 4, … are 1, 1, 2, 5, 14, 42, … etc.

Recursive formula: $$C(n) = \begin{cases} 1; n = 0 \\ \sum_{i=0}^{n-1} C(i) * C(n-1-i) \end{cases}$$

10.1 Calculate the Binomial Coefficient i.e. **nCr** meaning number of ways (disregarding order) that r objects can be chosen from among n objects.

Recursive formula: $C(n,r) = \begin{cases} 1; r = 0 \\ 1; n = r \\ C(n-1,r) + C(n-1,r-1) \end{cases}$

10.2 Calculate the Permutation Coefficient i.e. **nPr** meaning the number of ways to obtain an ordered subset having k elements from a set of n elements.

Recursive formula: $P(n,r) = \begin{cases} 1;\ r = 0 \\ 0; n < r \\ P(n-1,r) + r * P(n-1,r-1) \end{cases}$

10.3 Count the total number of moves necessary in **Tower of Hanoi** problem for n disks.

Recursive formula:

$$T(n, src, dest, tmp) = \begin{cases} 1; n = 1 \\ T(n-1, src, tmp, dest) + T(1, src, dest, tmp) + T(n-1, tmp, dest, src) \end{cases}$$

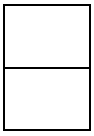
11.1 Preorder, Postorder and Inorder binary tree traversal.

11.2 Binary Search Tree search operation.

11.3 Binary Search operation from a sorted array.

12. **Tiling Problem —** Given a (2 x n) board and tiles of size (2 x 1), count the number of ways to tile the given board using the tiles. A tile can either be placed horizontally or vertically.

Recursive formula: $T(n) = \begin{cases} 1; n = 1 \\ 2; n = 2 \\ T(n-1) + T(n-2) \end{cases}$



13. **Coin Change Problem —** Given a value N, if we want to make change for N cents, and we have infinite supply of each of C = {C_1, C_2, C_3, …, C_N} valued coins, how many ways can we make the change? The order of coins doesn't matter.

Recursive formula:

$$C(coins[], sz, amount) = \begin{cases} 1; amount = 0 \\ 0; amount < 0 \\ 0; amount > 0 \ and \ sz = 0 \\ C(coins[], sz-1, amount) + C(coins[], sz, amount - coin\_value) \end{cases}$$

14. **Friends Pairing Problem –** Given n friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.

Recursive formula: $FP(n) = \begin{cases} 1; n = 1 \\ 2; n = 2 \\ FP(n-1) + (n-1) * FP(n-2) \end{cases}$

15. **Subset Sum Problem –** Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Recursive formula: $S(set[], sz, sum) = \begin{cases} true; sum = 0 \\ false; sum < 0 \\ false; sum > 0 \ and \ sz = 0 \\ S(set[], sz-1, sum) \ || \ S(set[], sz-1, sum - consideredValue) \end{cases}$