Lesson A9

Recursion

Objectives

- Recursion
- Pitfalls of Recursion
- Recursion Practice

Vocabulary

- BASE CASE
- RECURSION
- STACK
- STACK OVERFLOW ERROR

Recursion

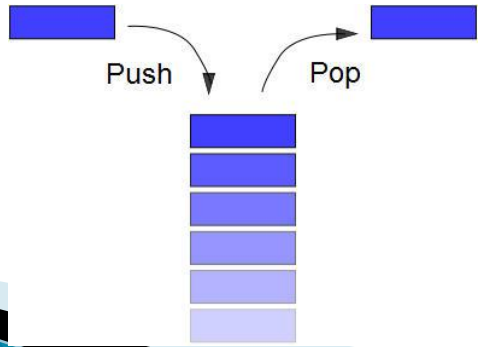
- Process of a method calling itself to solve another smaller version of the same problem
- With each recursive call, the problem becomes simpler and moves towards a *base case*

Base case

- When the solution to the problem can be calculated without another recursive call
- Each recursive method must have at least one base case

Stack

- Recursion involves the internal use of a *stack*
- A stack is a data abstraction that works like this:
  - New data is “pushed” or added to the top of the stack
  - When information is retrieved from the stack it is “popped” or removed from the top of the stack
  - This type of input/output is call Last in first out (LIFO)
  - Example of a stack is a stack of trays in a cafeteria
- The recursive calls of a method will be stored on a stack and manipulated in a similar manner



Factorial

- Factorials are computed using recursion

  $1! = 1$

  $2! = 2 * 1$ or $2 * 1!$

  $3! = 3 * 2 * 1$ or $3 * 2!$

  $4! = 4 * 3 * 2 * 1$ or $4 * 3!$

- Factorial is computed using recursion
- Rewrite the Factorial function fact(n)

$$fact(n) = \begin{cases} 1 & n = 1 \\ n \times fact(n-1) & n > 1 \end{cases}$$

Base case

Recursive Method – factorial

- A recursive method to solve the factorial problem is given below. Notice the recursive call in the last line of the method.
- The method calls another implementation of itself to solve a smaller version of the problem.

```
int fact(int n){
// returns the value of n!
// precondition: n >= 1
  if (n == 1){
    return 1;
  }else{
    return n * fact(n - 1);
  }
}
```

Recursion in Action – fact(4)

```
int fact(int n){
// returns the value of n!
// precondition: n >= 1
  if (n == 1){
    return 1;
  }else{
    return n * fact(n - 1);
  }
}
```

fact(n)	Suspended Actions	Popping back up the stack
fact(4)	4 * fact(3)	4 * 6 = 24
fact(3)	3 * fact(2)	3 * 2 = 6
fact(2)	2 * fact(1)	2 * 1 = 2
fact(1)		1

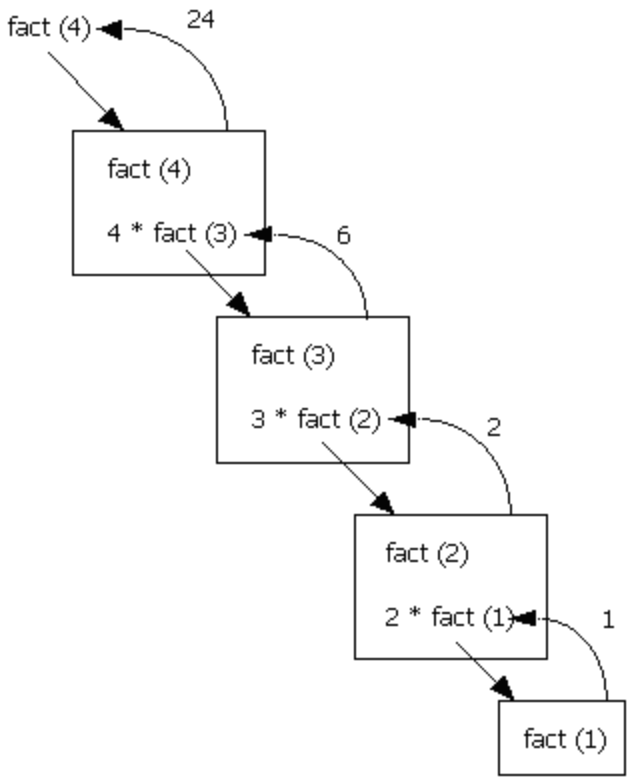
Tracing the call fact(4)

- When a recursive call is made, the current computation is temporarily suspended and placed on the stack with all its current information available for later use
- A completely new copy of the method is used to evaluate the recursive call. When that is completed, the value returned by the recursive call is used to complete the suspended computation. The suspended computation is removed from the stack and its work now proceeds.
- When the base case is encountered, the recursion will now unwind and result in a final answer. The expressions below should be read from right to left.

  fact(4) = 4 * fact(3) = 3 * fact(2) = 2 * fact(1) = 1

  24 ← 4 * 6 ← 3 * 2 ← 2 * 1

Recursive Boxes



Pitfalls of Recursion

- If recursion never reaches the base case, the recursive calls will continue until the computer runs out of memory and the program crashes
- The message “stack overflow error” or “heap storage exhaustion” indicates a possible runaway recursion
- When designing the algorithm, make sure that the algorithm is moving toward a base case

Recursive Power function

- Write a recursive power method that raises a base to some exponent, n. Use integers to keep things simple.

```
public double power(int base, int n)
{
        if (n == 1)
                return base;
        return base * power(base, n-1);
}
```

Recursion in Action – power(2,5)

```
public double power(int base, int n)
{
        if (n == 1)
                return base;
        return base * power(base, n-1);
}
```

power(base,n)	Suspended Actions	Popping back up the stack
power(2,5)	2 * power(2,4)	2 * 16 = 32
power(2,4)	2 * power(2, 3)	2 * 8 = 16
power(2,3)	2 * power(2,2)	2 * 4 = 8
power(2,2)	2 * power(2,1)	2 * 2 = 4
power(2,1)		2