Recursion

- A process in which the result of each repetition is dependent upon the result of the next repetition.

It Works Like Iteration

- Consider the following function, which computes
- \( N! = 1 \times 2 \times \ldots \times N \)

```c
int Factorial(int n) {
    int Product = 1,
    Scan = 2;
    while ( Scan <= n ) {
        Product = Product * Scan;
        Scan = Scan + 1;
    }
    return (Product);
}
```

Recursive Version

```c
int Factorial(int n) {
    if ( n > 1 )
        return( n * Factorial(n-1) );
    else
        return(1);
}
```

Recursion

- Every recursive algorithm can be implemented non-recursively.
  - recursion \(\leftrightarrow\) iteration
- Eventually, the routine must not call itself, allowing the code to "back out".
- Recursive routines that call themselves continuously are termed:
  - infinite recursion \(\leftrightarrow\) infinite loop
- Problem with recursive factorial implementation? Negative numbers!
- Recursion is inefficient at runtime.
Array Summation

```c
int SumArray(const int X[], int Start, int Stop) {
    // error check
    if (Start > Stop || Start < 0 || Stop < 0)
        return 0;
    else if (Start == Stop)  // base case
        return X[Stop];
    else // recursion
        return (X[Start] + SumArray(X, Start + 1, Stop));
}
```

The call:
```c
const int Size = 5;
int X[Size] = {37, 14, 22, 42, 19};
SumArray(X, 0, Size- 1); // note Stop is last valid index
```

```
return(X[0]+SumArray(X,1,4))          // == 37 + 97
return(X[1]+SumArray(X,2,4))       // == 14 + 83
return(X[2]+SumArray(X,3,4) )    // == 22 + 61
return(X[3]+SumArray(X,4,4)) // == 42 + 19
```

Coding Recursion

- Solve the trivial "base" case(s).
- Restate general case in 'simpler' or 'smaller' terms of itself.
  - Divide and Conquer
- List Example
  - Determine the size of a single linked list.
    - Base Case : Empty List, size = 0
    - General Case : 1 + Size(Rest of List)

Linked List Recursion

```c
int LinkList::SizeList ()
{ return(listSize(Head)); }

//private function: listSize
int LinkList::listSize (LinkNode* list)
{ if (list == NULL)
    return( 0 );
    else
        return( 1 + listSize(list->getNext()) );
}
```

```
Trace of SizeList

Trace listSize(list)
  listSize(list=(6, 28, 120, 496))
  = {1 + listSize(list=(28, 120, 496)))
  = {1 + (1 + listSize(list=(120, 496)))})
  = {1 + (1 + (1 + listSize(list=(496))))})
  = {1 + (1 + (1 + (1 + listSize(list=())))))})
  = {1 + (1 + (1 + (1 + 0))))
  = {1 + (1 + (1 + 1))})
  = {1 + (1 + 2))
  = {1 + 3) = 4
```

Types of Recursion

- Tail recursive
  - functions are characterized by the recursive call being the last statement in the function, (can easily be replaced by a loop).
- Head Recursive
  - Recursive call is the first statement (maybe after a base case check)
- Middle Decomposition
void intComma ( long num ) {
    if (num < 0) { // display sign for negatives
        cout << '-';
        num = -num;
    }
    if (num < 1000)
        cout << setw(3) << num;
    else {
        intComma(num / 1000); // Head recursive
        cout << ',';         // display digits
        num = num % 1000;    // separately
        cout << (num / 100); // for zeroes
        num = num % 100;
        cout << (num / 10) << (num % 10);
    }
}

Middle Decomposition
int rMax(const int ray[], int start, int end ) {
    const int Unknown = -1;
    int mid, h1max, h2max;
    if (end < start) return Unknown;
    mid = (start + end) / 2;
    h1max = rMax(ray, start, mid-1); // left half
    if (h1max == Unknown) h1max = start;
    h2max = rMax(ray, mid+1, end); // right half
    if (h2max == Unknown) h2max = end;
    if ( (ray[mid] >= ray[h1max]) &&
        (ray[mid] >= ray[h2max]) )
        return mid;
    else
        return( (ray[h1max]>ray[h2max]) ? h1max : h2max);
}

rMax Trace
- Give trace of the following array:
  ray = 56 23 66 44 78
- Give start, end, and return for each call, including the base cases in the order they RETURNED