Recursion?

- **Iterations**
  - Seen in almost all the programs
  - Used to process a large number of related tasks using “for” loop, “while” loop, “do”..”while” loop

- **Recursions**
  - A close sibling of iterations
  - Programming technique in which a method calls itself
  - Duplicate itself but with different parameters until task is trivial
  - It is well-suited for the “divide and conquer” strategy

- **Divide and Conquer strategy**
  - Break big problem into smaller problems
  - Solve smaller problems
  - Combine/Merge results
  - This strategy can be applied recursively until solution is trivial
Example 1: Triangular numbers/ summation

- Can you guess the next number: 1, 3, 6, 10, 15, 21, ...?
- $1=1$, $1+2=3$, $1+2+3=6$, $1+2+3+4=10$, $1+2+3+4+5=15$, etc.
- $\text{Sum}(7)=1+2+3+4+5+6+7=28$
- Visualization using triangular arrangements:
Example 1: Triangular numbers/ summation

- How to find the n\textsuperscript{th} term?
  - Solution 1: Using iterations

```c
int sum(int n)
{
    int total=0;
    while (n>0)// until n is one
    {
        total=total+n;
        n--;
    }
    return total
}
```
Example 1: Triangular numbers/ summation

Solution 2: Using Recursion

- Programming equivalent to mathematical induction

\[
\begin{align*}
1+2+3+4+5+\ldots+100 &= \sum(100) \\
&= 1+2+3+\ldots+100 \\
&= 1+2+\ldots+99+100 \\
&= \text{sum} (99) + 100 \\
\text{sum}(n) &= \text{sum} (n-1) + n & \text{Recursion step} \\
\text{sum}(0) &= 0 & \text{Base step}
\end{align*}
\]
Example 1: Triangular numbers/ summation

- Solution 2: Using Recursion
  - Sum all the $n$ columns of the triangle
  - Find the first tallest column, which has the value $n$
  - Sum all the other remaining columns $n-1$ columns

```c
int sum(int n)
{
    if (n==1) // base step
        return 1;
    else // recursive step
        {
            int temp=n+sum(n-1);
            return temp
        }
}
```

- It is critical for every recursive method to have a base case
Example 1: What is really happening?

- Modify code using output statement

```java
int total=sum(5)
System.out.println("Result is "+total);
```

```java
int sum(int n)
{
    System.out.println("Enter n="+n);
    if (n==1) // base step
    {
        System.out.println("Return 1 (base)");
        return 1;
    }
    else // recursive step
    {
        int temp=n+sum(n-1);
        System.out.println("Return "+temp);
        return temp
    }
}
```

Enter 5
Enter 4
Enter 3
Enter 2
Enter 1
Return 1 (base)
Return 3
Return 6
Return 10
Return 15
Result is 15
Example 1: What is really happening?

- Computer keeps track of each instance
- Each level of recursion keeps building in the memory stack (LIFO)

Example for $n=4$
Recursion efficiency

- **is recursion efficient?**
  - Calling a method involves some overhead
  - Working with the local memory stack could improve efficiency
  - Danger of memory leak (stack overflow) is an issue (recursion is almost never used in high-performance computing)
  - Most recursive problems can also be solved using iterations

- **So why use recursion?**
  - Can simplify a problem conceptually
  - More practical than iterations in some cases (find permutations, etc.)
  - Simpler than iterations when using multiple calls
  - Elegant programming
Example 2: Factorial

- Similar concept than summation (triangular number) but multiplication is used instead of addition

\[
4! = 4 \times 3 \times 2 \times 1 \\
n! = n \times (n-1) \times \ldots \times 1 \\
4! = 4 \times 3! \\
n! = n \times (n-1)! \\
\]

factorial(n) = n * factorial(n-1) //recursive step
factorial(1) = 1 //base step

```int fact(int n) {
    if (n==1) // base step
        return 1;
    else // recursive step
        {
            int temp=n*fact(n-1);
            return temp
        }
} ```
Example 3: Anagrams

- cat → cat, cta, atc, act, tca, tac (6 permutations)
- For a $n$ letter word, there is $n!$ Possibilities
- Remark: fact(6) means 720 permutations!
- Algorithm steps to anagram n letters:
  - Anagram the rightmost n-1 letters
  - Rotate all the n letters
- See code textbook p262
Example 4: A Recursive Binary Search

- Remember binary search?
  - Divide ordered array in half and select the correct half with new lowerbound, upperbound; Divide in half again, and so on

- Recursion can then replace the loop (still $O(\log(N))$)

```java
private int recFind(long searchKey, int lowerBound, int upperBound)
{
    int curIn;

    curIn = (lowerBound + upperBound) / 2;
    if(a[curIn]==searchKey)
        return curIn; // found it
    else if(lowerBound > upperBound)
        return nElems; // can't find it
    else // divide range
    {
        if(a[curIn] < searchKey) // it's in upper half
            return recFind(searchKey, curIn+1, upperBound);
        else // it's in lower half
            return recFind(searchKey, lowerBound, curIn-1);
    } // end else divide range
} // end recFind()

public int find(long searchKey)
{
    return recFind(searchKey, 0, nElems-1);
}
```
Example 5: Fibonacci numbers

- This is an example of recursion with multiple calls (more difficult to follow)
- Can you guess the next number: 0, 1, 1, 2, 3, 5, 8, 13, … ?

\[
\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \quad \text{Recursive case}
\]

\[
\begin{align*}
\text{Fib}(0) &= 0 \\
\text{Fib}(1) &= 1
\end{align*} \quad \text{Base case}
\]

```c
int fib(int n) {
    if (n<=1) // base step
        return n; // 0 or 1
    else // recursive step
    {
        int temp=fib(n-1)+fib(n-2);
        return temp
    }
}
```
Example 6: Display Linked-List

- Print a linked-list recursively

```
public void displayList()
{
    Link current = first; // start probe
    while (current!=null) {
        System.out.println(current.name);
        current = current.next;
    }
}
```

Non-recursive

```
public void displayList(Link current);
{
    if (current==null) return;
    else {
        System.out.println(current.name);
        displayList(current.next);
    }
}
```

Recursive