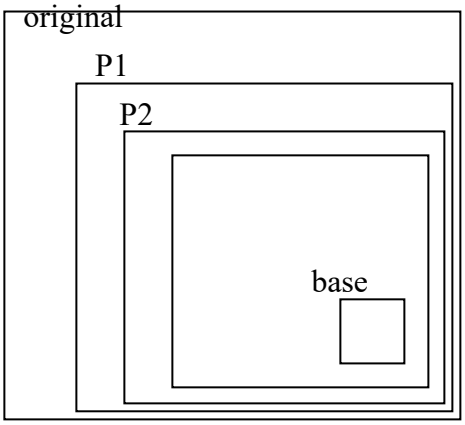
CSCI 2170
Recursion

Recursion breaks a problem into several smaller problems, all of which are of exactly the same type/form as the original problem: $P_{original}$> P1 > P2 > P3 …>$P_{base,}$ Solving a smaller problem helps solving the bigger problem, which eventually solves the original problem.

Poriginal can be solved if P1 is solved,
P1 is solved if P2 is solved,
P2 is solved if P3 is solved,
…
Pbase is solved



Base case: a special case where the solution to the problem is obvious or trivial

Four criteria that each recursive function should satisfy:

- The function makes recursive call to the function itself
- The size of the problem is reduced at each recursive call
- The reduction of problem size ensures that the base case will eventually be met
- When the base case is met, the recursive call ends (backtracking starts)

An example: Guessing a number between 0-100
PseudoCode for One good strategy is :

```
GuessInMiddle(beginning, end)
{
        guess = (beginning + end)/2;   // mid point

        if (guess == MagicNumber)
                Guessed correctly, stop

        Else
        {
                if (guess is higher than MagicNumber)
                        GuessInMiddle(beginning, guess-1);
                Else
                        GuessInMiddle(guess+1, end);
        }
}
```

<how does this example satisfy the four criteria listed above?>

Example 1. Factorial number

1! = 1; factorial (1) = 1 ➔ base case
2!=2*1 = 2 factorial (2) = 2*factorial(1) = 2
3! = 3*2*1 = 3*2!=6 factorial (3) = 3*factorial(2) = 6
4!=4*3*2*1 = 4*3!=24 factorial (4) = 4*factorial(3) = 24
…
n! = n*(n-1)*…*2*1 = n*(n-1)!
➔ recurrence relation factorial(n) = n*factorial(n-1)

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

Does this function satisfy the 4 criteria of the recursive function?
Trace program execution for Factorial (5)

Example 2: Counting using Fibonacci sequence

< counting the number of pairs of rabbits: assume rabbits never die, it takes 2 months for a pair of rabbits to mature and give birth to baby rabbits, 1 pair of rabbits give birth to 1 pair of rabbits (always born in male-female pair)>

month :	1	2	3	4	5	6	7	8	9	10	11	12 …
Pairs(month):	1	1	2	3	5	8	13	21	34	55	89	144 …

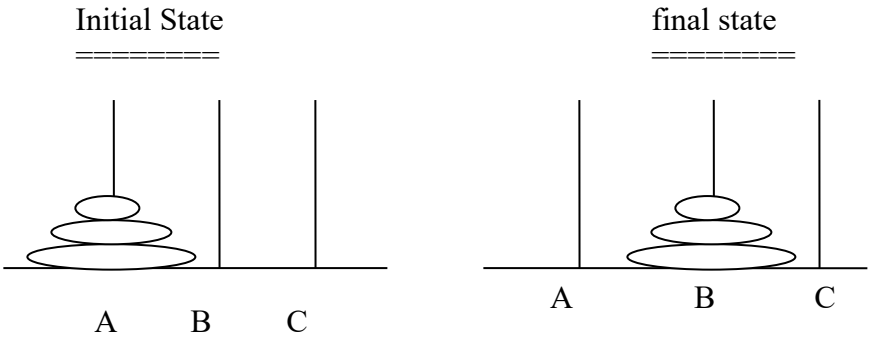
Recurrence relation: Pairs(month) = Pairs(month-1) + Pairs(month-2)
Base case: Pairs(0) = 0, Pairs(1) = 1, Pairs(2)=1
Change Pairs to Fibonacci sequence

```
int    Fibonacci(int   n)
{
        if (n<=2)
                return 1;
        else
                return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Trace the execution of function call "Fibonacci(5)"

Example 3: The SolveTowers of Hanoi problem

N disks, each with hole in middle
3 poles : A – source; B – destination; C – spare;
Goal: move N disks from pole A to pole C, where at anytime a disk can only be put on a disk that is larger than itself



This problem can be elegantly solved using recursion

The movement of the Towers can be captured as
SolveTowers(count, source, destination, spare)

- Step 1: start with the initial state, move the top N-1 disks from A to C using B as spare pole ➔ SolveTowers(N-1, A, C, B)
- Step2: move the largest disk from A to B ➔ SolveTowers(1, A, B, C)
- Step 3: move N-1 disks from C to B using A as spare ➔ SolveTowers(N-1, C, B, A)

C++ code for the solution

```
void  SolveTowers(int count, char source, char destination, char spare)
{
        if (count ==1)
                cout << "moving disk from " << source << " to "
                     << destination << endl;
        else
        {
                SolveTowers(count-1, source, spare, destination);
                SolveTowers(1, source, destination, spare);
                SolveTowers(count-1, spare, destination, source);
        }
}
```

When N=3 , i.e., solve SolveTowers(3, A, B, C)
Trace of program execution

SolveTowers(3, A, B, C)
SolveTowers(2, A, C, B) → SolveTowers(1, A, B, C) A→B
SolveTowers(1, A, C, B) A→C
SolveTowers(1, B, C, A) B→ C
SolveTowers(1, A, B, C) → A→ B
SolveTowers(2, C, B, A) → SolveTowers(1, C, A, B) C→ A
→ SolveTowers(1, C, B, A) C→ B
→ SolveTowers(1, A, B, C) A→ B

Will these steps solve the Tower of Hanoi problem with N=3?

binary search in recursion

```
void BinarySearch (int array[], int first, int last, int key, bool & found, int & location)
{
	int mid;
	if (first > last)
		found = false;
	else
	{
		mid = (first+last) / 2;
		if (key = array[mid])
		{
			found = true;
			location = mid;
		}
		else if (key < array[mid])
			BinarySearch(array, first, mid-1, key, found, location);
		else if (key > array[mid])
			BinarySearch(array, mid+1, last, key, found, location);
	}
}
```

Find the k^{th} smallest value in an array (not sorted)

kSmall(k, anArray, first, last)

```
=  kSmall(k, anArray, first, pivotIndex-1)                          if k < pivotIndex-first+1
   p                                                                if k = pivotIndex-first +1
   kSmall(k-(pivotIndex-first+1), anArray, pivotIndex+1, last)  if k>pivotIndex-first+1
```

```
kSmall(int k, int anArray[], int first, int last)
{
        choose pivot item p from anArray[first … last]
        //partition the items of anArray[first … last] about p
        Partition(anArray, first, last, p);

        if (k < pivotItem – first + 1)
                return   kSmall(k, anArray, first, pivotIndex-1);
        else if (k == pivotIndex – first +1)
                return p;
        else
                return  kSmall((k – (pivotIndex – first+1)), anArray, pivotIndex+1, last);
}

void Partition(dataType A[], int F, int L, int& PivotIndex)
{
  dataType Pivot = A[F];     // copy pivot

  // initially, everything but pivot is in unknown
  int LastS1 = F;           // index of last item in S1
  int FirstUnknown = F + 1;  // index of first item in unknown

  // move one item at a time until unknown region is empty
  for (; FirstUnknown <= L; ++FirstUnknown)
  {  // move item from unknown to proper region
    if (A[FirstUnknown] < Pivot)
    {  // item from unknown belongs in S1
      ++LastS1;
      Swap(A[FirstUnknown], A[LastS1]);
    }  // end if

    // else item from unknown belongs in S2
  }  // end for

  // place pivot in proper position and mark its location
  Swap(A[F], A[LastS1]);
  PivotIndex = LastS1;
}  // end Partition
```