Recursion

- **A Recursive function** is a function that calls itself.

- Recursive functions can be useful in solving problems that can be broken down into smaller or simpler subproblems of the same type.

- A **base case** should eventually be reached, at which time the recursion will stop.

- A recursive function should include a test for the base case(s).

- With each recursive call, the parameter controlling the recursion should move closer to the base case.

- Eventually, the parameter reaches the base case and the chain of recursive calls terminates.

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables being created.

- As each copy finishes executing, it returns to the copy of the function that called it.

- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function.
**Demo Recursive Program**

// This program demonstrates a simple recursive function.
#include <iostream>
using namespace std;

// Function prototype
void countDown(int);

int main()
{
    countDown(10);
    return 0;
}

void countDown(int count)
{
    if (count == 0) //base case
        cout << "Blastoff!" << endl;
    else
    {
        cout << count << "..." << endl;
        countDown(count - 1); //recursive call
    }
    return;
}
Types of Recursion

- **Direct Recursion:**
  - a function calls itself

- **Indirect Recursion:**
  - function A calls function B which calls function A ...
  - function A calls function B which calls function C ...
    - which call function A ...
The Recursive Factorial Function

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

- \( n-1! = (n-1) \times (n-2)! \)

...  
- \( 0! = 1 \)  //base case

Recursive Factorial Demo Program:
// This program demonstrates a recursive function  
// to calculate the factorial of a number.
#include <iostream>
using namespace std;

// Function prototype  
int factorial(int);

int main()
{
    int number;

    cout << "Enter an integer value and I will display its factorial: ";
    cin >> number;
    cout << "The factorial of " << number << " is ";
    cout << factorial(number) << endl;
    return 0;
}

// Definition of factorial. A recursive function to  
// calculate the factorial of the parameter, num.
int factorial(int num)
{
    if (num == 0) //base case
        return 1;
    else
        return num * factorial(num - 1); //recursive call
}
The Recursive gcd (greatest common divisor) Function

- **gcd(x, y)** (where x and y are two positive integers):
  
  ```
  if (x % y) == 0) //base case
     gcd(x,y) = y;
  else
     gcd(x,y) = gcd(y, x%y); //recursive call
  ```

// This program demonstrates a recursive function to calculate the greatest common divisor (gcd) of two numbers.
#include <iostream>
using namespace std;

// Function prototype
int gcd(int, int);

int main()
{
    int num1, num2;

    cout << "Enter two integers: ";
    cin >> num1 >> num2;
    cout << "The greatest common divisor of " << num1;
    cout << " and " << num2 << " is ";
    cout << gcd(num1, num2) << endl;
    return 0;
}

// Definition of gcd. This function uses recursion to calculate the greatest common divisor of two integers, passed into the parameters x and y.
int gcd(int x, int y)
{
    if (x % y == 0) //base case
        return y;
    else
        return gcd(y, x % y); //recursive call
}
Solving Recursively Defined Problems

- Some problems naturally lend themselves to recursive solutions.

- **Fibonacci Numbers:**
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
  
  - \(\text{fib}(0) = 0\)
  - \(\text{fib}(1) = 1\)
  - \(\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)\) for \(n \geq 2\)

// This program demonstrates a recursive function
// that calculates Fibonacci numbers.
#include <iostream>
using namespace std;

// Function prototype
int fib(int);

int main()
{
    cout << "The first 10 Fibonacci numbers are:\n";
    for (int x = 0; x < 10; x++)
        cout << fib(x) << " ";
    cout << endl;
    return 0;
}

// ********************************************
// Function fib. Accepts an int argument in n. This function returns
// the nth Fibonacci number.
// ********************************************
int fib(int n)
{
    if (n <= 0) // base case
        return 0;
    else if (n == 1) // base case
        return 1;
    else
        return fib(n - 1) + fib(n - 2); // recursive call
}
A Recursive Binary Search Function

// This program demonstrates a recursive function that performs a binary search on an integer array.
#include <iostream>
using namespace std;

// Function prototype
int binarySearch(int [], int, int, int);
const int SIZE = 20;

int main()
{
    int tests[SIZE] = { 101, 142, 147, 189, 199, 207, 222,
                       234, 289, 296, 310, 319, 388, 394,
                       417, 429, 447, 521, 536, 600};

    int result; // Result of the search
    int empID; // What to search for

    cout << "Enter the Employee ID you wish to search for: ";
    cin >> empID;
    result = binarySearch(tests, 0, SIZE - 1, empID);
    if (result == -1)
        cout << "That number does not exist in the array.\n";
    else
    {
        cout << "That ID is found at element " << result;
        cout << " in the array\n";
    }
    return 0;
}
The binarySearch function performs a recursive binary search on a range of elements of an integer array. The parameter first holds the subscript of the range's starting element, and last holds the subscript of the range's last element. The parameter value holds the search value. If the search value is found, its array subscript is returned. Otherwise, -1 is returned indicating the value was not in the array.

```c
int binarySearch(int array[], int first, int last, int value) {
    int middle; // midpoint of search
    if (first > last) // base case
        return -1;
    middle = (first + last)/2;
    if (array[middle] == value) // base case
        return middle;
    else if (array[middle] < value) // recursive call
        return binarySearch(array, middle + 1, last, value);
    else // (array[middle] > value) // recursive call
        return binarySearch(array, first, middle-1, value);
}
```
The QuickSort Algorithm

- QuickSort is a **very efficient** recursive sorting algorithm.
- QuickSort begins by determining a **pivot value**.
- Once the pivot value is determined, two sublists (sublist1 and sublist2) are created and values are shifted so that elements in sublist1 are $<$ the pivot value and elements in sublist2 are $\geq$ the pivot value.
- The algorithm then recursively sorts sublist1 and sublist2.
- The base case is when a sublist has 1 element.
// This program demonstrates the Quicksort algorithm.
#include <iostream>
#include <algorithm>   //needed for swap function
using namespace std;

// Function prototypes
void quickSort(int [], int, int);
int partition(int [], int, int);

int main()
{
    // Array to be sorted.
    const int SIZE = 10;
    int array[SIZE] = {17, 53, 9, 2, 30, 1, 82, 64, 26, 5};

    // Echo the array to be sorted.
    for (int k = 0; k < SIZE; k++)
        cout << array[k] << " ";
    cout << endl;

    // Sort the array using Quicksort.
    quickSort(array, 0, SIZE-1);

    // Print the sorted array.
    for (int k = 0; k < SIZE; k++)
        cout << array[k] << " ";
    cout << endl;

    return 0;
}
void quickSort(int arr[], int start, int end) {
    if (start < end) // test for base case
    {
        // Partition the array and get the pivot point.
        int p = partition(arr, start, end);

        // Sort the portion before the pivot point.
        quickSort(arr, start, p - 1);

        // Sort the portion after the pivot point.
        quickSort(arr, p + 1, end);
    }
    return; // base case
// partition() rearranges the entries in the array arr from
// start to end so all values greater than or equal to the
// pivot are on the right of the pivot and all values less
// than are on the left of the pivot.
//************** ************** ************** ***

int partition(int arr[], int start, int end)
{
    // The pivot element is taken to be the element at
    // the start of the subrange to be partitioned.
    int pivotValue = arr[start];
    int pivotPosition = start;

    // Rearrange the rest of the array elements to
    // partition the subrange from start to end.
    for (int pos = start + 1; pos <= end; pos++)
    {
        if (arr[pos] < pivotValue)
        {
            // arr[pos] is the "current" item.
            // Swap the current item with the item to the
            // right of the pivot element.
            swap(arr[pivotPosition + 1], arr[pos]);
            // Swap the current item with the pivot element.
            swap(arr[pivotPosition], arr[pivotPosition + 1]);
            // Adjust the pivot position so it stays with the
            // pivot element.
            pivotPosition ++;
        }
    }

    return pivotPosition;
}

• Note:
  The swap() function used in partition() is part of the Standard
  Template Library (STL). You need to #include <algorithm>
in order to use it.
The Towers of Hanoi

- **Setup:**
  - 3 pegs, one has n disks on it, the other two pegs empty.

- The disks are arranged in increasing diameter, top to bottom

- **Objective:**
  - move the disks from peg 1 to peg 3 using peg 2 as a temp

- **Rules:**
  - only one disk moves at a time
  - all remain on pegs except the one being moved
  - a larger disk cannot be placed on top of a smaller disk

- **Recursive Solution:**
  if (n > 0)
  {
    move n-1 disks from peg 1 to peg 2 using peg 3 as a temp
    move a disk from peg 1 to peg 3
    move n-1 disks from peg 2 to peg 3 using peg 1 as a temp
  }
else //base case
  do nothing
// This program displays a solution to the towers of Hanoi game.

#include <iostream>
using namespace std;

// Function prototype
void moveDisks(int, string, string, string);

int main()
{
    // Play the game with 3 disks.
    moveDisks(3, "peg 1", "peg 3", "peg 2");
    cout << "All the disks have been moved!";
    return 0;
}

//*********************************************
// The moveDisks function displays disk moves used
// to solve the Towers of Hanoi game.
// The parameters are:
//    n     : The number of disks to move.
//    source : The peg to move from.
//    dest   : The peg to move to.
//    temp   : The temporary peg.
//*********************************************
void moveDisks(int n, string source, string dest, string temp)
{
    if (n > 0) //test for base case n == 0
    {
        // Move n - 1 disks from source to temp
        // using dest as the temporary peg.
        moveDisks(n - 1, source, temp, dest);

        // Move a disk from source to dest.
        cout << "Move a disk from " << source << " to " << dest << endl;

        // Move n - 1 disks from temp to dest
        // using source as the temporary peg.
        moveDisks(n - 1, temp, dest, source);
    }
    return; //base case
}
Exhaustive and Enumeration Algorithms

- **Enumeration Algorithm:**
  An algorithm that generates all possible combinations of items of a certain type.

- **Exhaustive Algorithm:**
  An algorithm that searches through such a set of all possible combinations to find the optimal one.
Recursion vs. Iteration

- **Recursion:**
  - Natural formulation of solution for certain problems.
  - Results in shorter, simpler functions.
  - At times, it may not execute efficiently.

- **Iteration:**
  - Executes more efficiently than recursion.
  - May not be as natural as recursion for some problems.

- **Note:**
  - In general, recursion should be used whenever a problem has a natural recursive solution that does not unnecessarily recompute solutions to subproblems and the equivalent solution based on iteration is not obvious or is difficult.