1. **Definition:**

The expression “Recursion” is derived from Latin: *Re-* = back and *currere* = to run, or to happen again, especially at repeated intervals. Many problems can be solved recursively, *e.g.* games of all types from simple ones like the Towers of Hanoi problem to complex ones like chess. When a data structure is defined recursively, it may be processed by a recursive function.

In programming, a Recursive Algorithm (function) is usually a function that has the ability to call itself. The function is supposed to do a finite number of recursive calls with different input parameters in each call. The desired result is obtained from the current action and the contributions of previous actions (history). The terminal action is predefined (does not need previous history). The termination condition (Base Case) is obviously extremely important. If it is omitted, then the function will continue to call itself indefinitely.

Recursion is used when the current result depends on previous history. Also, some problems are already defined in a recursive way and hence it is easier to code them recursively. Moreover, we use recursive algorithms when processing a large data structure that is composed of similar but smaller structures (e.g. trees).

As an example, let us consider a simple function to compute the sum of the first (n) positive integers, with *n* > 0. We know that this function is given by \( \text{Sum (n)} = 1+2+\ldots+n \). If we choose an iterative implementation, then the function would be:

```c
// An Iterative Algorithm
int Sum (int n)
{
    int i;
    int s = 0;
    for ( i = 1; i <= n; i++) s = s + i;
    return s;
}
```

The above iterative form uses some local variables (*int i, int s*) and a *for* loop. The questions now are:

“*Can the function Sum (n) be defined in terms of itself with a different parameter?*” and

“*Is there a base case for this function?*”

The answer to these questions is “yes”. Since \( \text{Sum (n)} = 1+2+\ldots(n-1) + n \) and since \( \text{Sum (n-1)} = 1+2+\ldots(n-1) \), then our recursive definition of the function would be:
Sum (n) = Sum (n-1) + n for n > 1 \textit{(this is the general case)}
and Sum (n) = 1 for n = 1 \textit{(this is the Base Case)}

Using these definitions, we can implement the function as a \textit{``Recursive Algorithm''} as follows:

```c
// A Recursive Algorithm
int Sum (int n)
{
    if ( n == 1) return 1; // Base Case
    else return n + Sum(n-1) ; // General Case
}
```

But, will this work? For example, if we call this function from the main function with

```
int k = Sum (3);
```

will we get \( k = 1+2+3 = 6 \)?

Let us trace the recursive function for \( n = 3 \). First it will call Sum (3), and since \( n = 3 \) it will try to return \( 3 + \text{Sum} (2) \) to the main. The result is postponed and a second call is made as Sum (2). Since now \( n = 2 \), it will try to return \( 2 + \text{Sum} (1) \) to the previous call. The result is postponed and a third call is made as Sum (1). Since now \( n = 1 \) and this is the base case, Sum (1) = 1 is returned to the previous call so the Sum (2) = 2 + 1 = 3. This is returned again to the previous call so that Sum (3) = 3 + 3 = 6 which is finally returned to the main.

These steps are shown as follows:

![Recursive Sum Algorithm](image)

We can see that the main idea is to solve a bigger problem (Sum (3)) by trying to solve a smaller problem (Sum (2)) and if this is not given, we try to solve it with a still smaller problem (Sum (1)), and so on until we reach a given solution (Base case). When the base case is reached, we trace our way back, combining the solutions we obtained till we reach again the main call with the combined solutions. Hence, there are two passes; the “Quest” for the base case, and the “Combine pass”.

Prof. Amr Goneid, AUC
For our example of Sum (3),
The “Quest” pass is:
main → Sum(3) = 3 + Sum(2) → Sum(2) = 2 + Sum(1) → Sum(1) = 1 (Base case reached).
The “combine pass” is:
1 + 2 → Sum(2) = 3 → 3 + Sum(2) = 3 + 3 = 6 → main

2. Rules for Recursion and Basic Forms

The rules for a recursive function are simple:

- There must be a **base case**.
- There is a **general case** other than the base case.
- There is a **path** from a general case to the base case.
- Successive recursive calls should take us **towards the base case**. The problem should get smaller in that direction.

The following basic structure is commonly used in building recursive functions:

```markdown
if (Base Case)
    {Do Terminal Action}
else {Do General Case Actions};
```

When the base case is to do nothing, we may use a modified structure as follows:

```markdown
if (Not Base Case)
    {Do General case Actions};
```

An example of the first structure, consider the **factorial function** \( n! \) defined as:

\[
    n! = \begin{cases} 
    1 & \text{for } n = 0 \\
    n(n-1)! & \text{for } n > 0 
    \end{cases}
\]

Hence, we may implement it as follows:

```c
int factorial (int n)
{
    if (n == 0) return 1; // Base Case
    else return ( n * factorial (n-1)); // General Case
}
```
An example of the second structure is when we implement a recursive function to print the contents of an array for locations $s$ up to and including $e$. Obviously this can only be done if $s \leq e$. If $s > e$ we do nothing (Base case).

```cpp
void printlist (int A[], int s, int e)  
{  
   if (s <= e) // General Case only
      {  
         cout << A[s];
         printlist (A, s+1, e);
      }
}
```

Notice that if $s > e$, the function will exit without doing anything.

3. **Two Famous Recursive Methods**

**Exclude & Conquer**

A well known recursive method is to exclude one case and conquer (process) the rest of cases recursively.

The previous examples are all Exclude & Conquer algorithms. Another example is a function to compute $x^n$ where $n \geq 0$ is integer. This function is defined as:

$$x^n = \begin{cases} 1 & \text{for } n = 0 \\ x \ast x^{n-1} & \text{for } n > 0 \end{cases}$$
An Exclude & Conquer algorithm would be:

```c
double power (double x, int n)
{
    if (n == 0)  return 1.0; // Base Case
    else return ( x * power(x,n-1)); // General Case
}
```

Another example of this method is to compute the sum of elements of an integer array `A` from location `s` through location `e`. This may be defined as:

\[
sum(A,s,e) = \sum_{i=s}^{e} A_i = \begin{cases} 
A_s & \text{for } s = e \\
A_s + \sum_{i=s+1}^{e} A_i & \text{for } s < e 
\end{cases}
\]

The implementation would be:

```c
int array_sum (int A[ ], int s, int e)
{
    if (s == e) return A[s]; // Base Case
    else return (A[s] + array_sum (A, s+1, e)); // General Case
}
```

Another example is a recursive function to return the number of zero elements in an array from location `s` through location `e`. We may define the function by the following algorithm:

```c
int nzeros (A, s, e)
{
    let k = 1 if A_s is zero and k = 0 otherwise;
    if we have only one element, i.e., s = e, return k
    else we have more than one element
        and in this case we return k + the number of zeros in elements from s+1 through e.
}
```

Following this algorithm, the function would be:

```c
int nzeros (int A[ ], int s, int e)
{
    int k = (A[s] == 0? 1 : 0); // k = 1 if A_s is zero and k = 0 otherwise
    if (s == e) return k; // if we have only one element
       else return (k + nzeros (A, s+1, e)); // if more than one element
}
```
Divide & Conquer

In this method, we keep dividing the problem into parts (usually two almost equal parts) until the problems become small enough to reach base cases.

Let us redesign the algorithm to compute $x^n$ where $n \geq 0$ is integer. This function may also be defined as:

$$x^n = \begin{cases} 
1 & \text{if } n = 0 \\
x & \text{if } n = 1 \\
(x^* x)^{n/2} & \text{if } n \text{ even} \\
x^*(x^* x)^{n/2} & \text{if } n \text{ odd}
\end{cases}$$

A Divide & Conquer algorithm would be:

```cpp
double power (double x, int n) {
    if (n == 0) return 1.0; else if (n == 1) return x;
    else if (n mod 2) return power(x*x , n / 2) * x;
    else return power(x*x , n / 2);
}
```

Another example is to find the maximum element in an array from a location $s$ through location $e$ using Divide & Conquer (D&Q). In this case, if we have one element, then it is the maximum. For more than one element, we divide the range $\{s,e\}$ in the approximate middle (say location $m$) to get two sub-arrays, one from $s$ to $m$ and the second from $m+1$ to $e$. We keep repeating this division recursively until we end up with single elements. Here is the algorithm:
Assume we have a function to compare two values (a, b) and return the maximum of the two:

```c
int max2 (int a, int b)
{
    return ((a > b)? a : b);
}
```

Then the maximum in an array may be computed by the recursive D&Q function:

```c
int maximum (int a[], int s, int e)
{
    if (s == e) return a[s]; // Base Case
    else // General Case
    {
        int m = (s + e)/2; // Divide in the middle
        int maxL = maximum (a, s, m); // Conquer left half
        int maxR = maximum (a, m+1, e); // Conquer right half
        return max2 (maxL, maxR); // Combine
    }
}
```

A famous example of a recursive D&Q search algorithm is Binary Search. Here we are searching for an item X in a presorted array A of items from a location s through a location e. If the item X is found, we return its location (which is somewhere in the domain {s, e}), otherwise we return a -1, indicating that it is not found in that range.

The algorithm finds the approximate middle location (mid) between s and e and checks for a match with X at that location. If there is a match, the item is found, otherwise we check if the item is greater than the middle (in the right half from mid+1 to e) or it is in the left half (from s to mid-1). Therefore, one half is immediately discarded by one question. The process of D&Q is repeated recursively on the promising half until the item is found or s becomes greater than e (item does not exist and we return -1). The implementation of this strategy is as follows:

```c
int Bsearch (int A[], int x, int s, int e)
{ int mid;
    if (s > e) return -1; // Base case: No elements left, search failed
    else // General case
    {
        mid = (s+e) / 2; // Divide in the middle
        if (x == A[mid]) return mid; // Success at mid
        else if (x > A[mid]) // Conquer right
            return Bsearch(A,x,mid+1,e);
        else // Conquer left
            return Bsearch(A,x,s,mid-1);
    }
}
```