# Simple Recursion

A procedure or function which calls itself is a recursive routine.

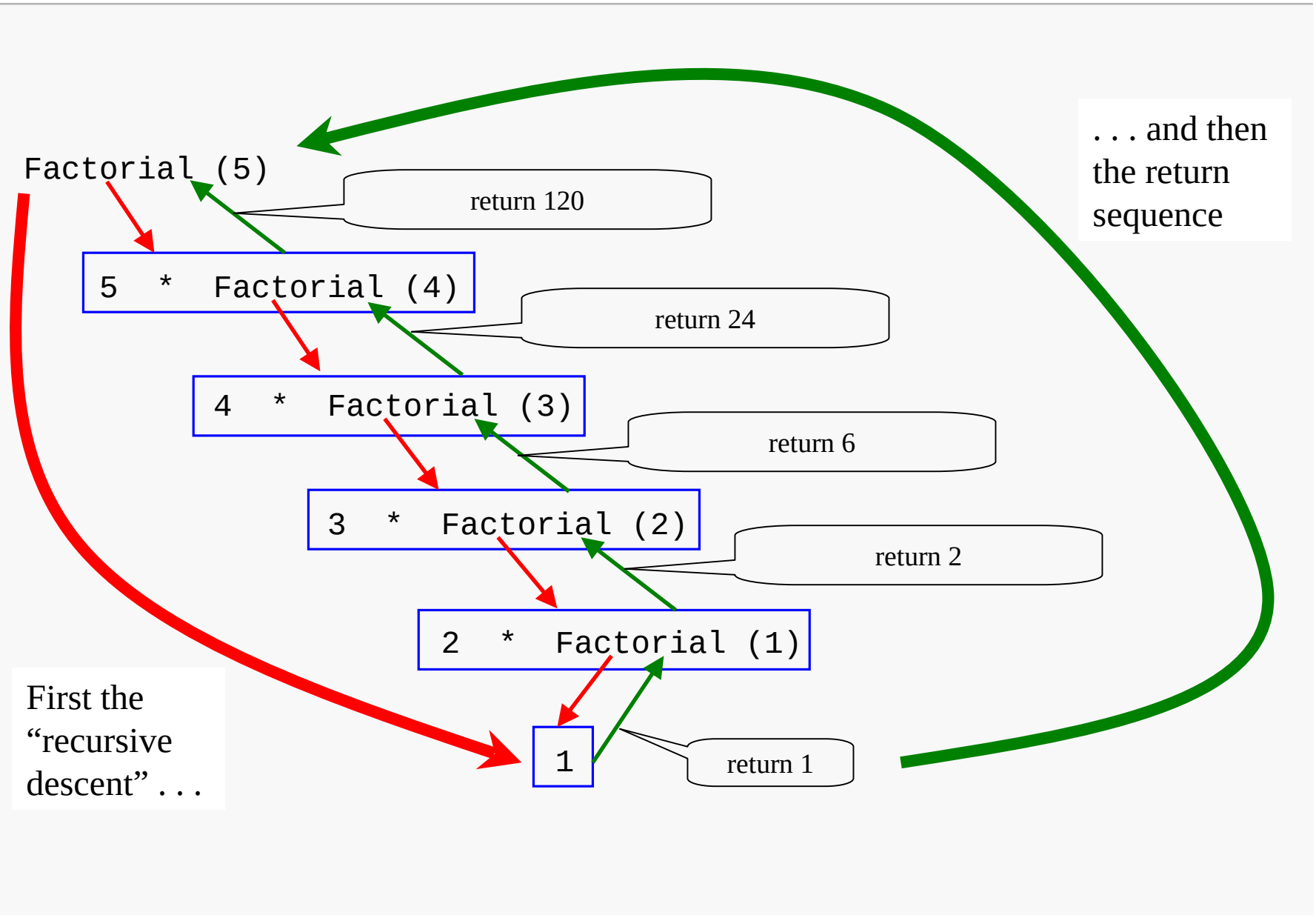
Consider the following function, which computes N! = 1 * 2 * ... * N

```
int Factorial(int n) {
  int Product = 1,
      Scan    = 2;

  while ( Scan <=  n ) {
    Product  =  Product  *  Scan ;
    Scan = Scan + 1 ;
  }
  return (Product) ;
}
```

Now consider a recursive version of `Factorial`:

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

# Recursive Execution Trace



# Simple Recursion

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

# Recursive Array Summation

Here is a recursive function that takes an array of integers and computes the sum of the elements:

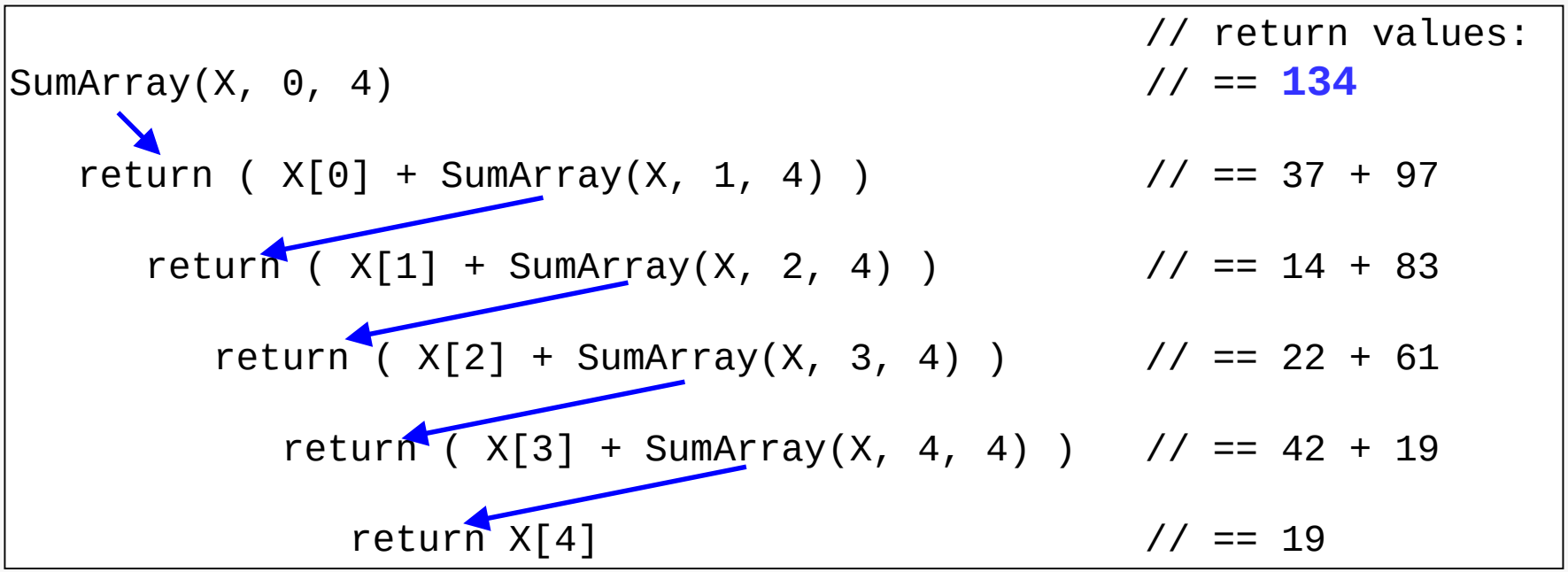
```
// X[]     array of integers to be summed
// Start   start summing at this index . . .
// Stop    . . . and stop summing at this index
//
int SumArray(const int X[], int Start, int Stop) {

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

# Recursive Array Summation Trace

The call:

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

would result in the recursive trace:



# Recursive Selection Sort

```
// X[]     array of integers to be sorted
// Start   start selection pass here . . .
// Stop    . . . and stop the search pass here
//
void SelSort(int X[], int Start, int Stop) {

   int Begin = Start,
       Check ,
       SmallSoFar = Begin;
   int tmpInt;

   if (Start < Stop) {
      for (Check = Begin + 1; Check <= Stop; Check++) {
         if (X[Check] < X[SmallSoFar])
            SmallSoFar = Check;
      }
      tmpInt = X[Begin];
      X[Begin] = X[SmallSoFar];
      X[SmallSoFar] = tmpInt;

      SelSort(X, Start + 1, Stop);
   }
}
```

# Recursive Selection Sort Trace

The call:

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

would result in the recursive trace: