Recursion

Manolis Koubarakis

Recursion

- Recursion is a **fundamental concept** of Computer Science.
- It usually help us to write simple and elegant solutions to programming problems.
- You will learn to program recursively by working with many examples to develop your skills.

Recursive Programs

- A **recursive program** is one that calls itself in order to obtain a solution to a problem.
- The reason that it calls itself is to compute a solution to a **subproblem** that has the following properties:
  - The subproblem is **smaller** than the problem to be solved.
  - The subproblem can be solved **directly (as a base case)** or **recursively by making a recursive call**.
  - The subproblem's solution can be **combined** with solutions to other subproblems to obtain a solution to the overall problem.

Example

- Let us consider a simple program to add up all the squares of integers from `m` to `n`.

- An **iterative function** to do this is the following:

```
int SumSquares(int m, int n)
{
   int i, sum;

   sum=0;
   for (i=m; i<=n; ++i) sum +=i*i;
   return sum;
}
```

Recursive Sum of Squares

```
int SumSquares(int m, int n)
{
   if (m<n) {
      return m*m + SumSquares(m+1, n);
   } else {
      return m*m;
   }
}
```

Recursive call

Base case

Comments

- In the case that the range `m:n` contains more than one number, the solution to the problem can be found by adding (a) the solution to the smaller subproblem of summing the squares in the range `m+1:n` and (b) the solution to the subproblem of finding the square of `m`. (a) is then solved in the same way (recursion).

- We stop when we reach the **base case** that occurs when the range `m:n` contains just one number, in which case `m==n`.

- This recursive solution can be called **“going-up” recursion** since the successive ranges are `m+1:n`, `m+2:n` etc.

Going-Down Recursion

```
int SumSquares(int m, int n)
{
   if (m<n) {
      return SumSquares(m, n-1) + n*n;
   } else {
      return n*n;
   }
}
```

Recursive call

Base case

Recursion Combining Two Half-Solutions

```
int SumSquares(int m, int n)
{
   int middle;

   if (m==n) {
     return m*m;
   } else {
     middle=(m+n)/2;
     return
        SumSquares(m,middle)+SumSquares(middle+1,n);
   }
}
```

Base case

Recursive call

Recursive call

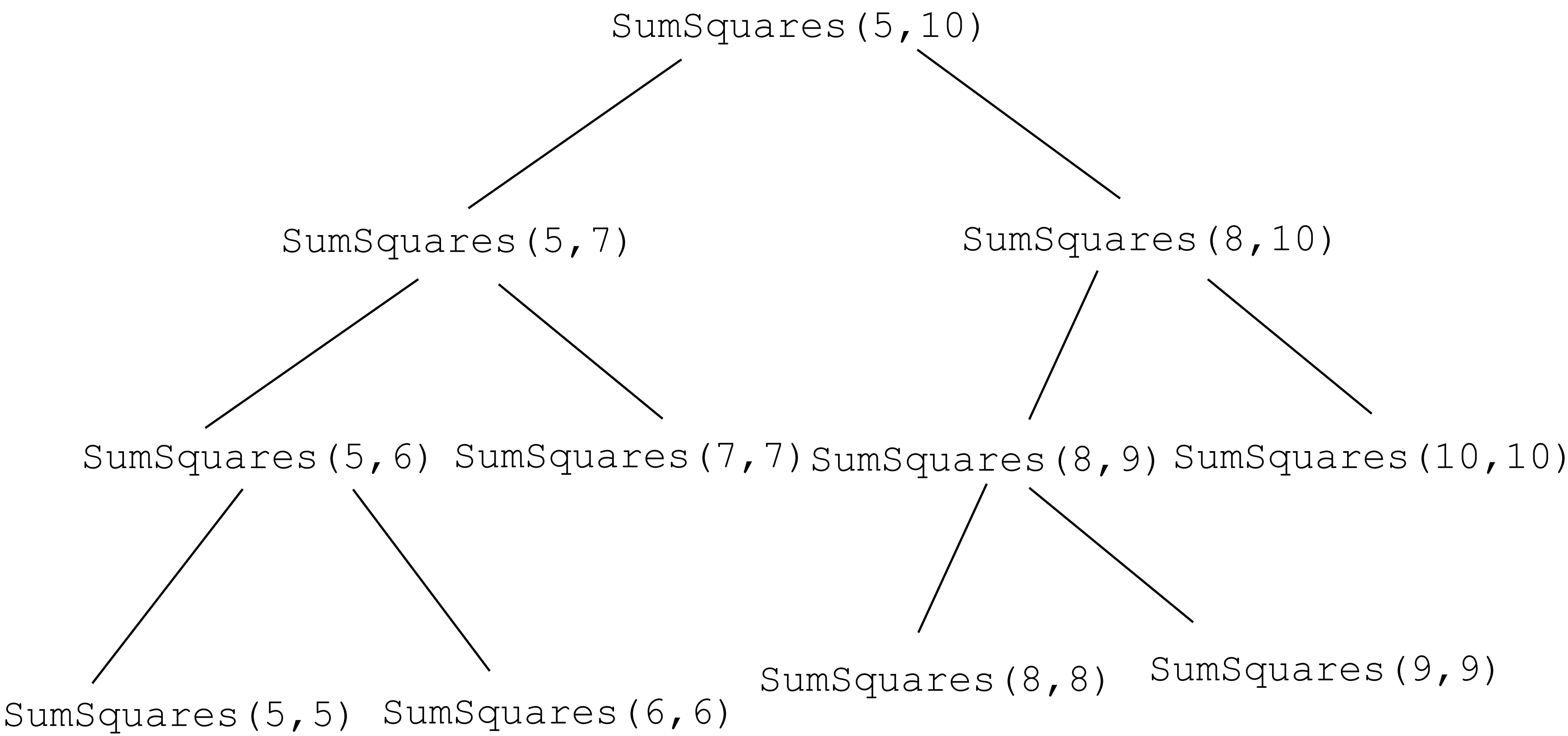
Comments

- The **recursion** here says that the sum of the squares of the integers in the range `m:n` can be obtained by adding the sum of the squares of the left half range, `m:middle`, to the sum of the squares of the right half range, `middle+1:n`.
- We stop when we reach the **base case** that occurs when the range contains just one number, in which case `m==n`.
- The middle is computed by using **integer division** (operator `/`) which keeps the quotient and throws away the remainder.

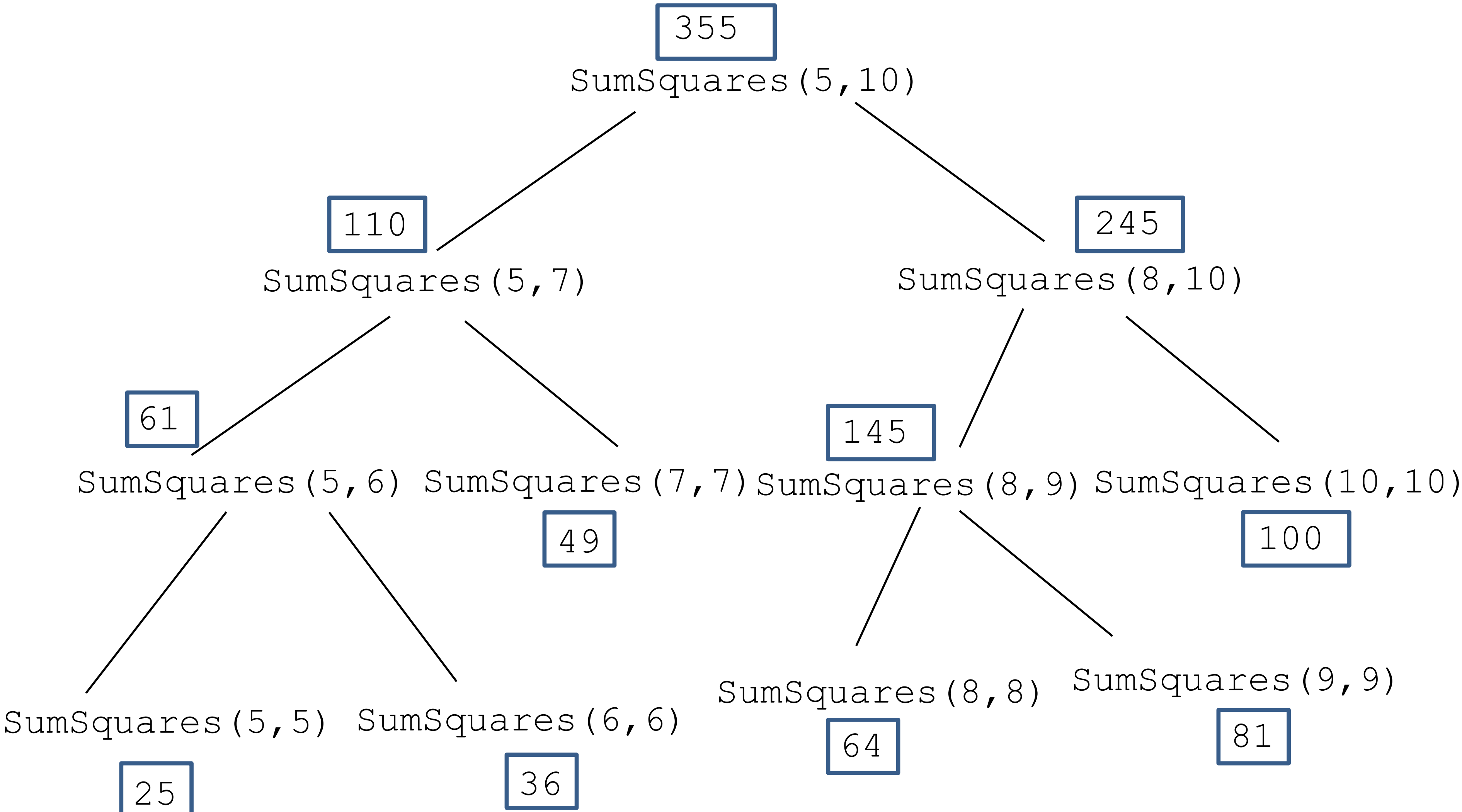
Call Trees and Traces

- We can depict graphically the behaviour of recursive programs by drawing **call trees** or **traces**.

Call Trees


SumSquares(5,10)
SumSquares(5,7)
SumSquares(8,10)
SumSquares(5,6)
SumSquares(7,7)
SumSquares(8,9)
SumSquares(10,10)
SumSquares(5,5)
SumSquares(6,6)
SumSquares(8,8)
SumSquares(9,9)

Annotated Call Trees



Traces

```
SumSquares(5,10)=SumSquares(5,7)+SumSquares(8,10)=
                =SumSquares(5,6)+SumSquares(7,7)
                   +SumSquares(8,9)+SumSquares(10,10)
                =SumSquares(5,5)+SumSquares(6,6)
                       +SumSquares(7,7)
                   +SumSquares(8,8)+SumSquares(9,9)
                       +SumSquares(10,10)
                =((25+36)+49)+((64+81)+100)
                =(61+49)+(145+100)
                =(110+245)
                =355
```

Computing the Factorial

- Let us consider a simple program to compute the factorial `n!` of `n`.

- An **iterative function** to do this is the following:

```
int Factorial(int n)
{
   int i, f;

   f=1;
   for (i=2; i<=n; ++i) f*=i;
   return f;
}
```

Recursive Factorial

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

Base case

Recursive call

Computing the Factorial (cont'd)

- The previous program is a "going-down" recursion.
- Can you write a "going-up" recursion for factorial?
- Can you write a recursion combining two half-solutions?
- The above tasks do not appear to be easy.

Computing the Factorial (cont'd)

- It is easier to first write a function `Product(m,n)` which **multiplies** together the numbers in the range `m:n`.
- Then `Factorial(n)=Product(1,n).`

Multiplying `m:n` Together Using Half-Ranges

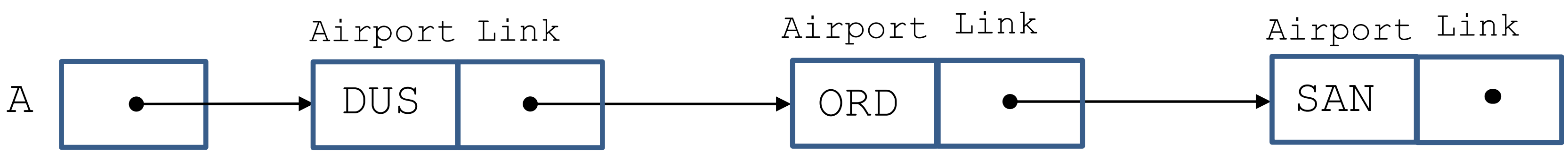
```
int Product(int m, int n)
{
   int middle;

   if (m==n) {
     return m;
   } else {
     middle=(m+n)/2;
     return Product(m,middle)*Product(middle+1,n);
   }
}
```

Base case

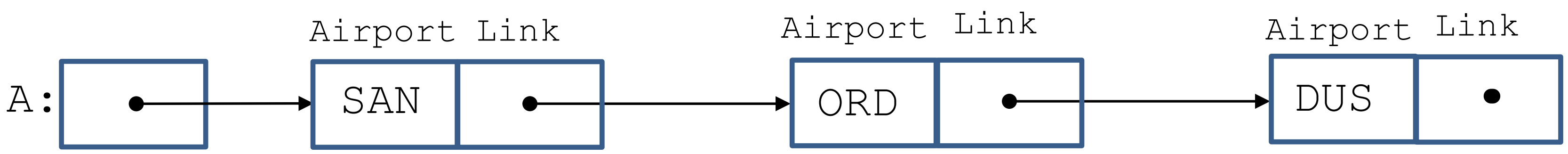
Recursive call

Recursive call

Reversing a Linked List



The Result



Reversing a Linked List

- Let us now writing a function for reversing a linked list `L`.
- The type `NodeType` has been defined in the previous lecture as follows:

```
typedef char AirportCode[4];
typedef struct NodeTag {
                AirportCode Airport;
                struct NodeTag *Link;
                        } NodeType;
```

Reversing a List Iteratively

- An **iterative function for reversing a list** is the following:

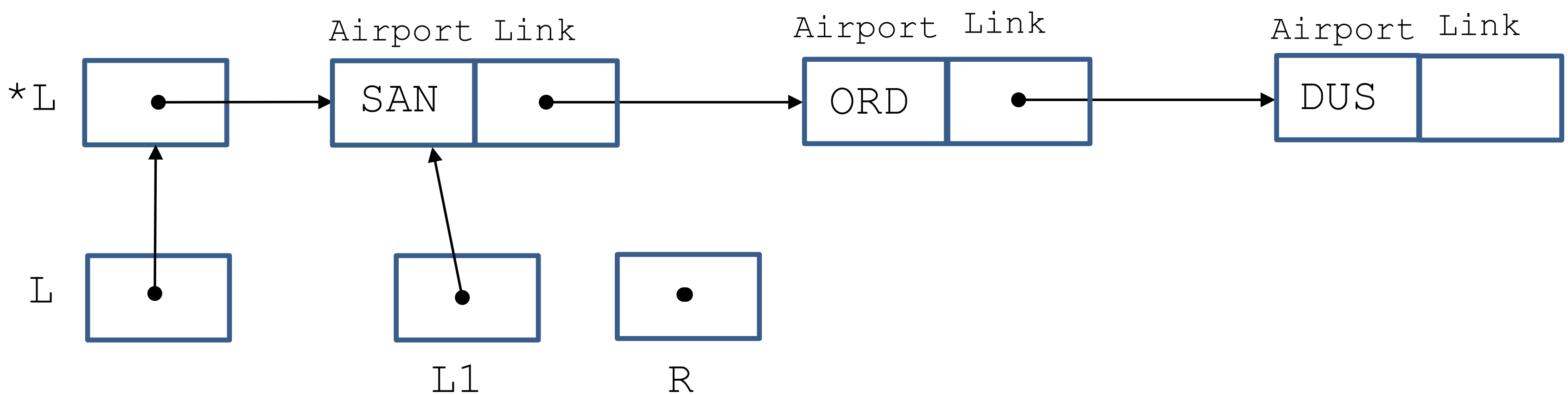
```
void Reverse(NodeType **L)
{
   NodeType *R, *N, *L1;

   L1=*L;
   R=NULL;
   while (L1 != NULL) {
      N=L1;
      L1=L1->Link;
      N->Link=R;
      R=N;
   }
   *L=R;
}
```

Reversing a List Iteratively (cont'd)

- In addition to variable `L`, the function uses the variables `R`, `N`, `L1` which are pointers to structures of type `NodeType`. These variable are used as follows:
  - `L1` is used to traverse the list to be reversed.
  - `N` always points to the previous node of the node `L1` points to, as `L1` traverses the list to be reversed.
  - `R` is initially `NULL` and later points to the last node of the sublist of `L` which has been reversed already.

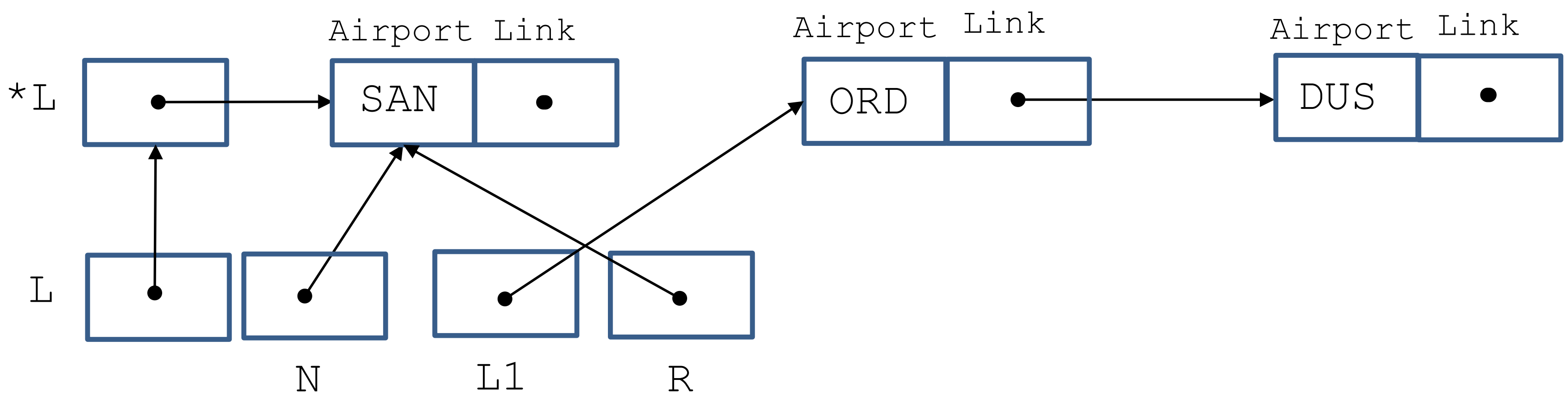
Before the `while` Loop

```
L1=*L;
R=NULL;
```



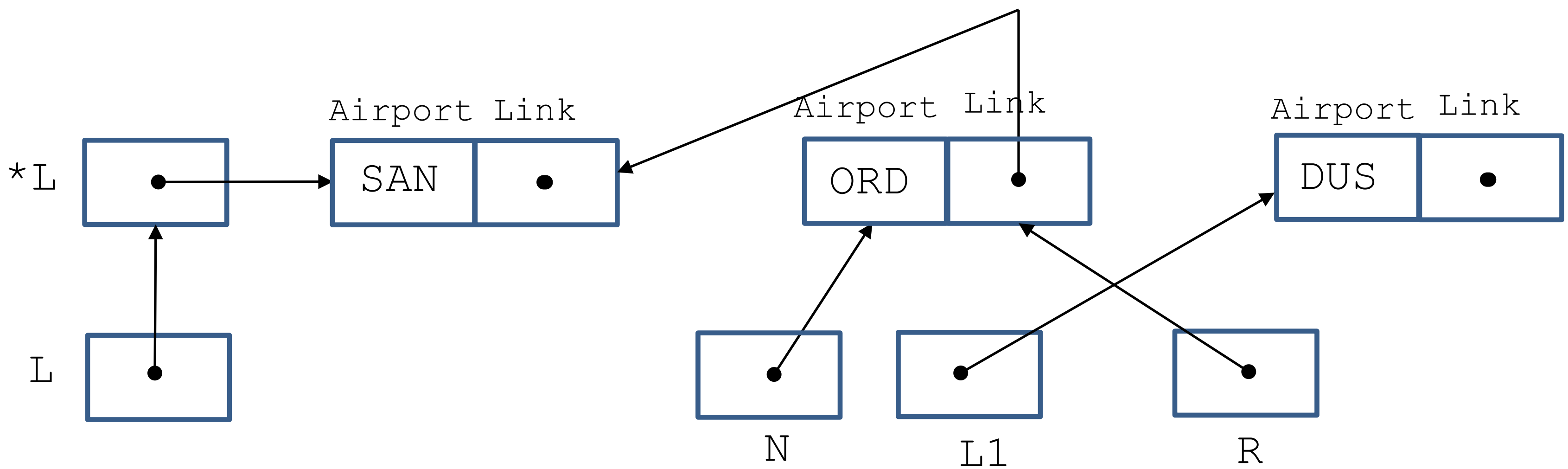
After the First Execution of the `while` Loop

```
while (L1 != NULL) {
      N=L1;
      L1=L1->Link;
      N->Link=R;
      R=N;
   }
```



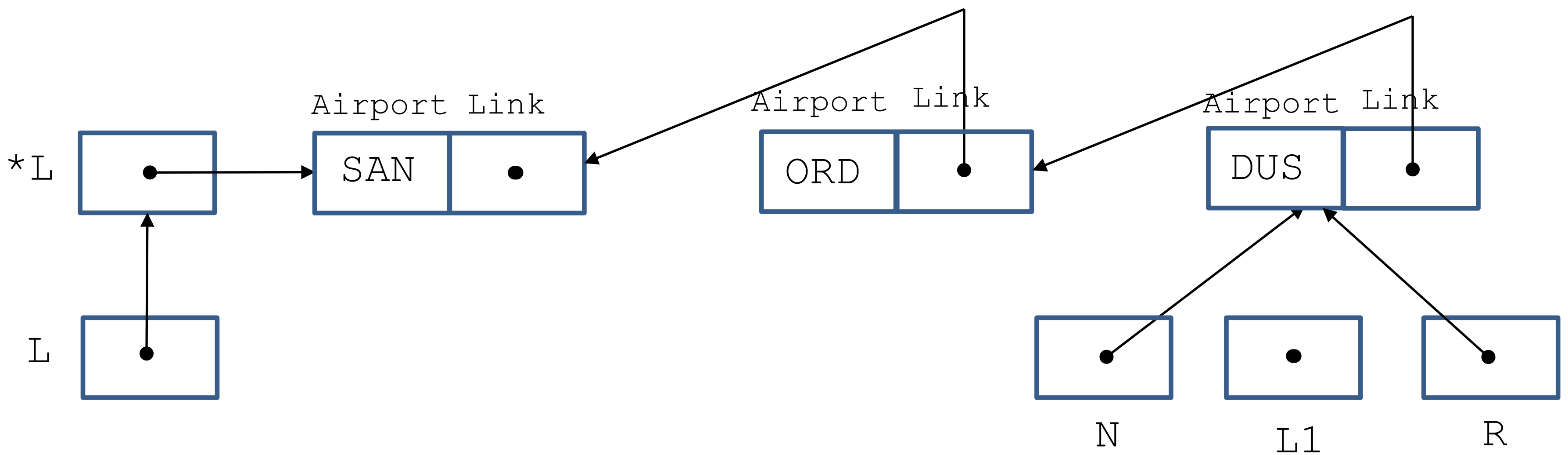
After the Second Execution of the `while` Loop

```
while (L1 != NULL) {
      N=L1;
      L1=L1->Link;
      N->Link=R;
      R=N;
   }
```



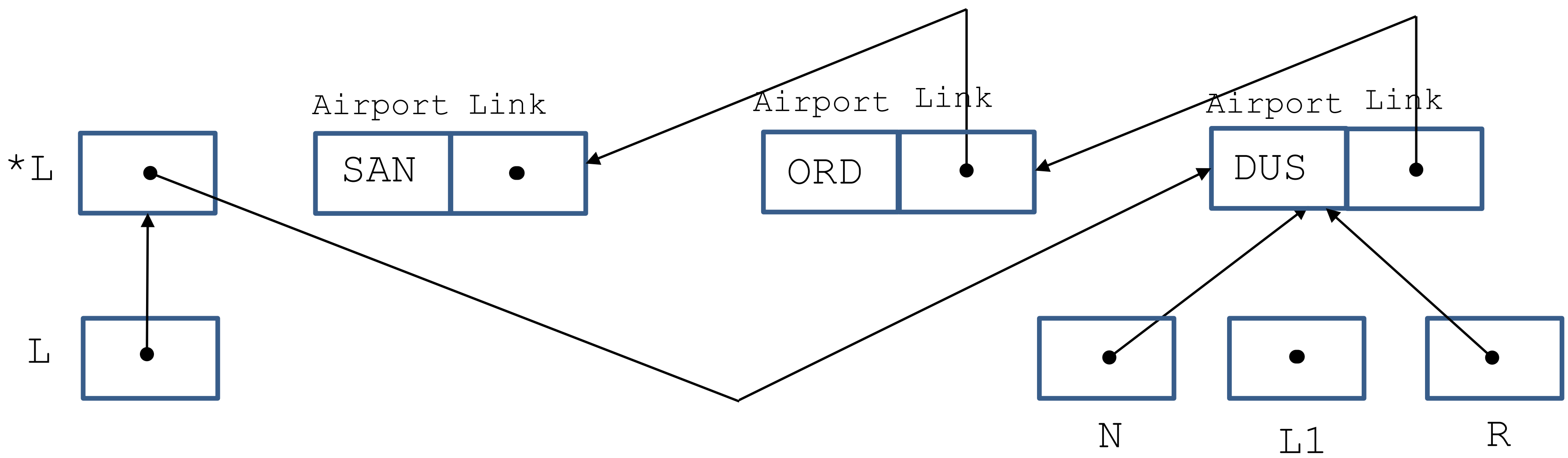
After the Third Execution of the `while` Loop

```
while (L1 != NULL) {
      N=L1;
      L1=L1->Link;
      N->Link=R;
      R=N;
   }
```



After the `while` Loop Terminates

`*L=R;`



Question

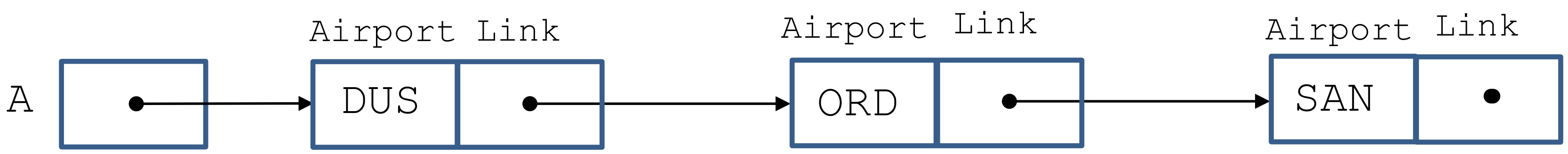
- If in our main program we have a list with a pointer `A` to its first node, how do we call the previous function?

Answer

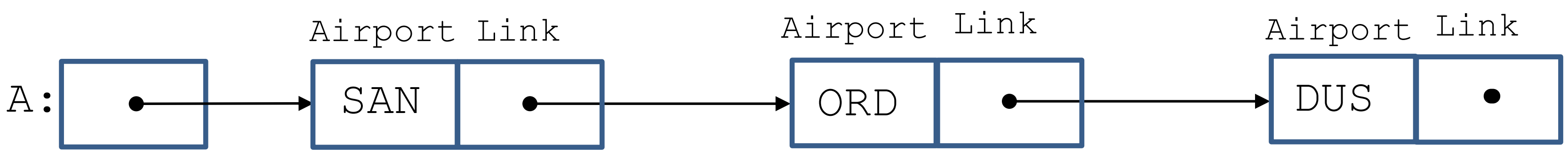
- We should make the following call:
  `Reverse(&A)`

Example

- Let us now call `Reverse(&A)` for the following list.



The Resulting List



Reversing Linked Lists (cont'd)

- A recursive solution to the problem of reversing a list `L` is found by partitioning the list into its **head** `Head(L)` and **tail** `Tail(L)` and then concatenating the reverse of `Tail(L)` with `Head(L)`.

Head and Tail of a List

- Let `L` be a list. `Head(L)` is a list containing the first node of `L`. `Tail(L)` is a list consisting of `L`'s second and succeeding nodes.
- If `L==NULL` then `Head(L)` and `Tail(L)` are not defined.
- If `L` consists of a single node then `Head(L)` is the list that contains that node and `Tail(L)` is `NULL`.

Example

- Let `L=(SAN, ORD, BRU, DUS)`. Then `Head(L)=(SAN)` and `Tail(L)=(ORD, BRU, DUS)`.

Reversing Linked Lists (cont'd)

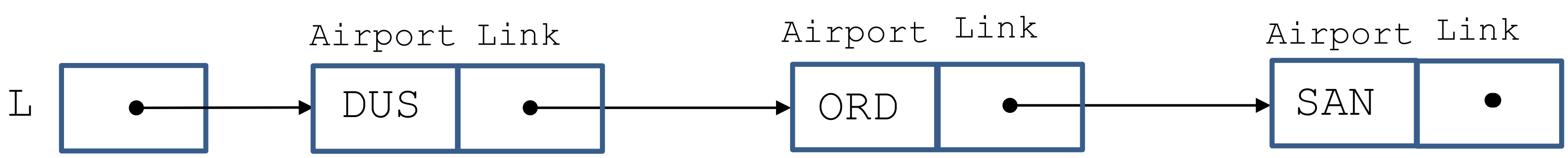
```
NodeType *Reverse(NodeType *L)
{
     NodeType *Head, *Tail;

     if (L==NULL) {
          return NULL;
     } else {
          Partition(L, &Head, &Tail);
          return Concat(Reverse(Tail), Head);
    }
}
```

Reversing Linked Lists: Partitioning the List into Head and Tail

```
void Partition(NodeType *L, NodeType **Head,
NodeType **Tail)
{
     if (L != NULL) {
         *Tail=L->Link;
         *Head=L;
         (*Head)->Link=NULL;
     }
}
```

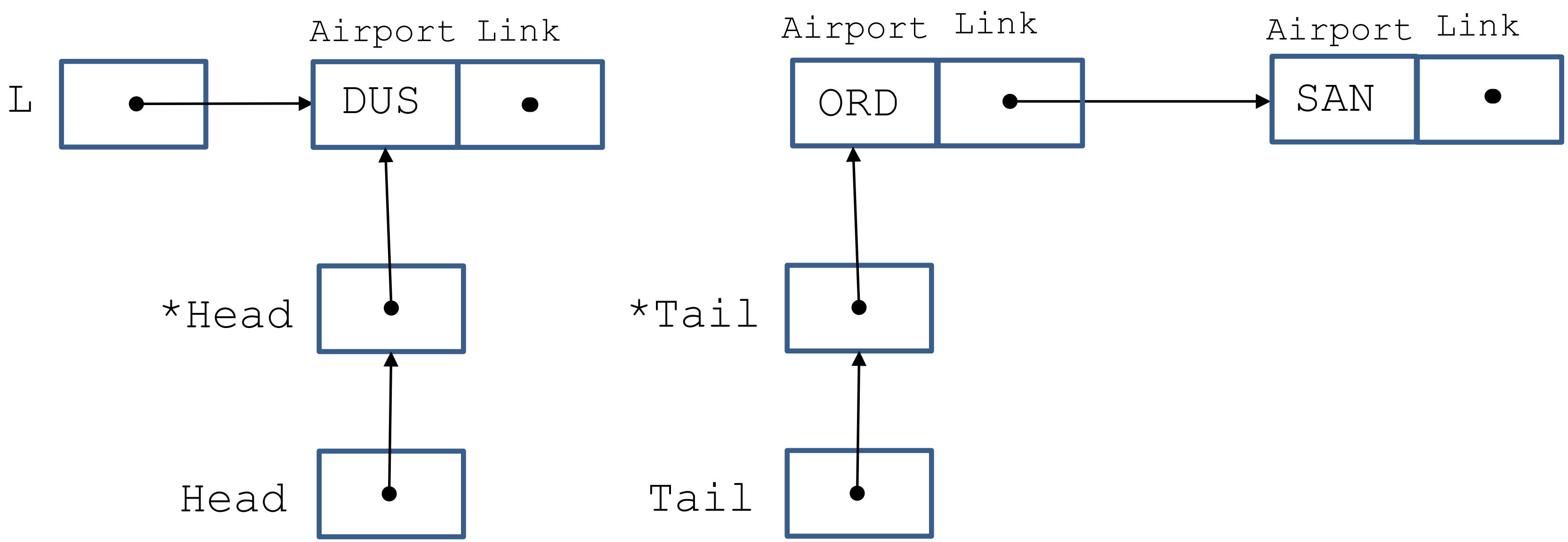
Example

- Let us execute `Partition(L, &Head, &Tail)` for the following list.



Example (cont'd)

```
*Tail=L->Link;
*Head=L;
(*Head)->Link=NULL;
```

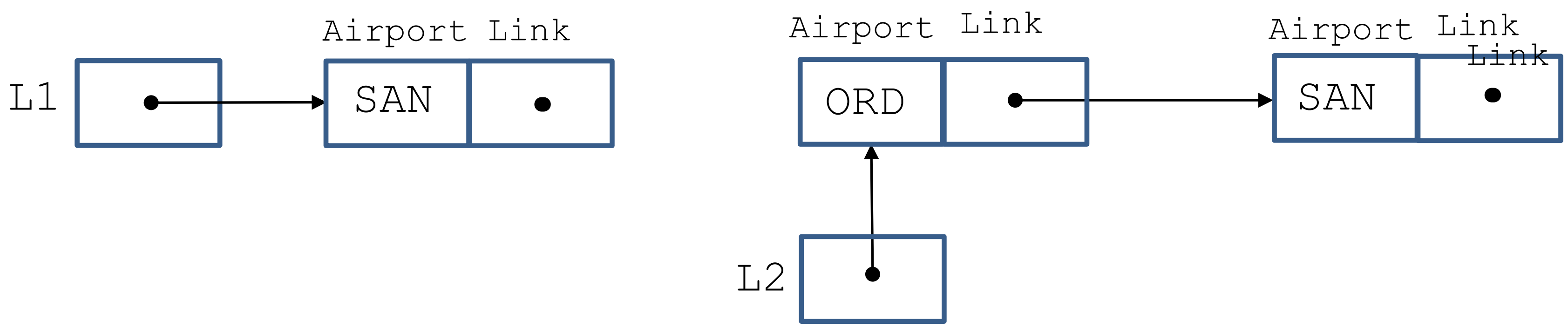


Reversing Linked Lists: Concatenation

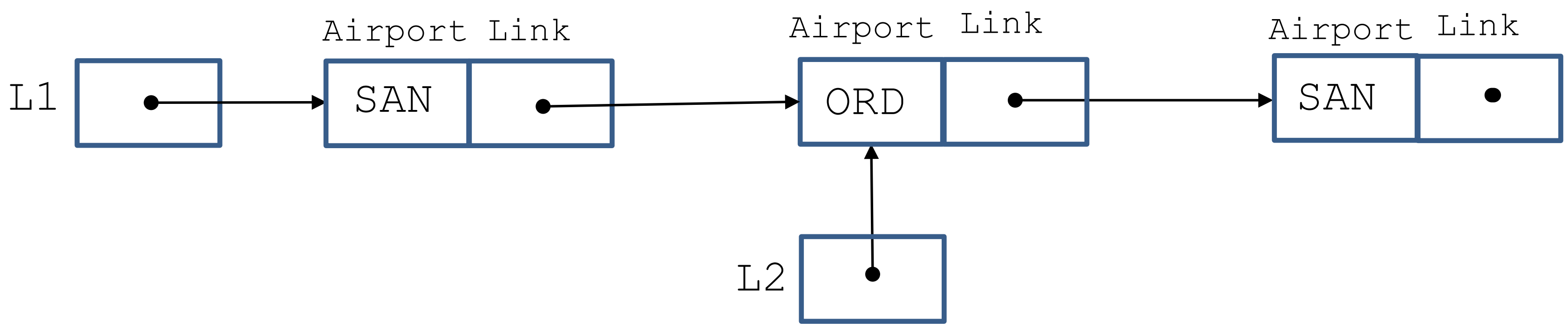
```
NodeType *Concat(NodeType *L1, NodeType *L2)
{
   NodeType *N;

   if (L1 == NULL) {
      return L2;
   } else {
      N=L1;
      while (N->Link != NULL) N=N->Link;
      N->Link=L2;
      return L1;
   }
}
```

Example

- Let us execute `Concat(L1,L2)` for the following two lists



The Resulting List



Infinite Regress

- Let us consider again the recursive factorial function:

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

- What happens if we call `Factorial(0)`?

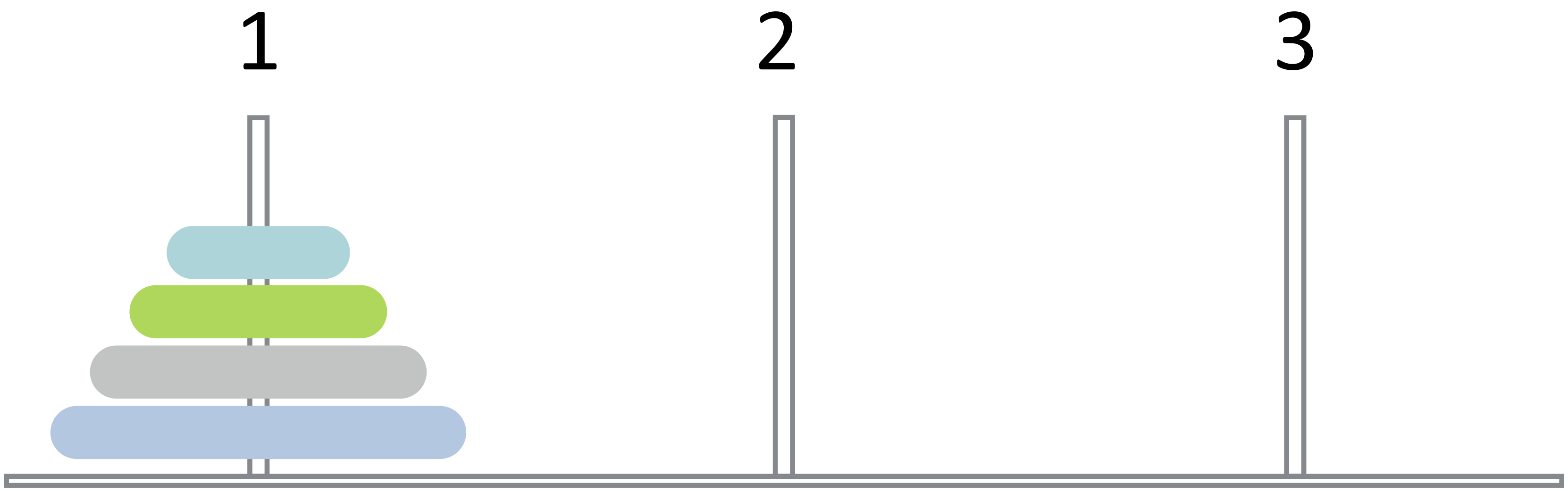
Infinite Regress (cont'd)

```
Factorial(0)= 0 * Factorial(-1)
            = 0 * (-1) * Factorial(-2)
            = 0 * (-1) * (-2) * Factorial(-3)
```

and so on, in an infinite regress.

When we execute this function call, we get "Segmentation fault (core dumped)".

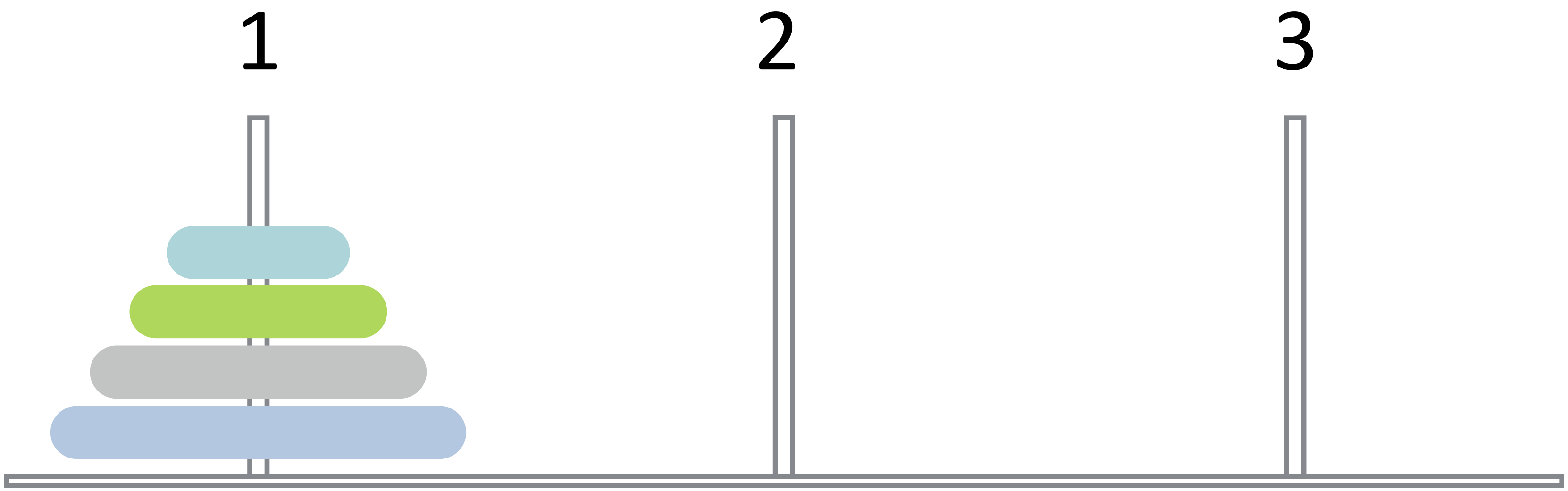
The Towers of Hanoi



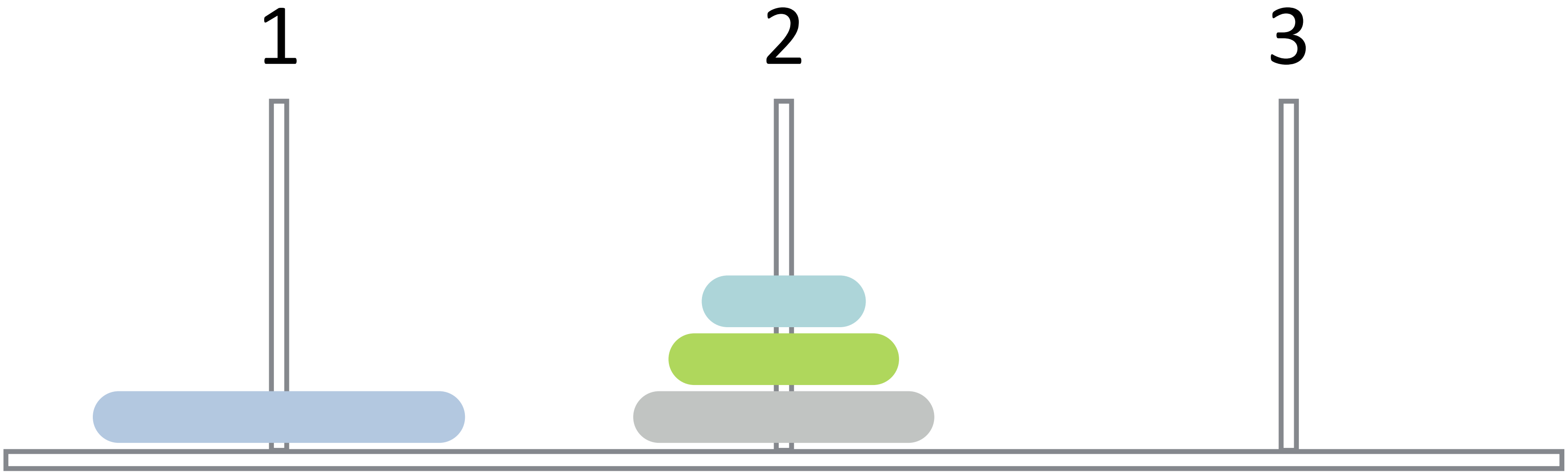
The Towers of Hanoi (cont’d)

- To Move the 4 disks from Peg 1 to Peg 3 using Peg 2 as an intermediate stop:
  - Move the top 3 disks from Peg 1 to Peg 2 using Peg 3 as an intermediate stop.
  - Move the remaining 1 disk from Peg 1 to Peg 3.
  - Move 3 disks from Peg 2 to Peg 3 using Peg 1 as an intermediate stop.

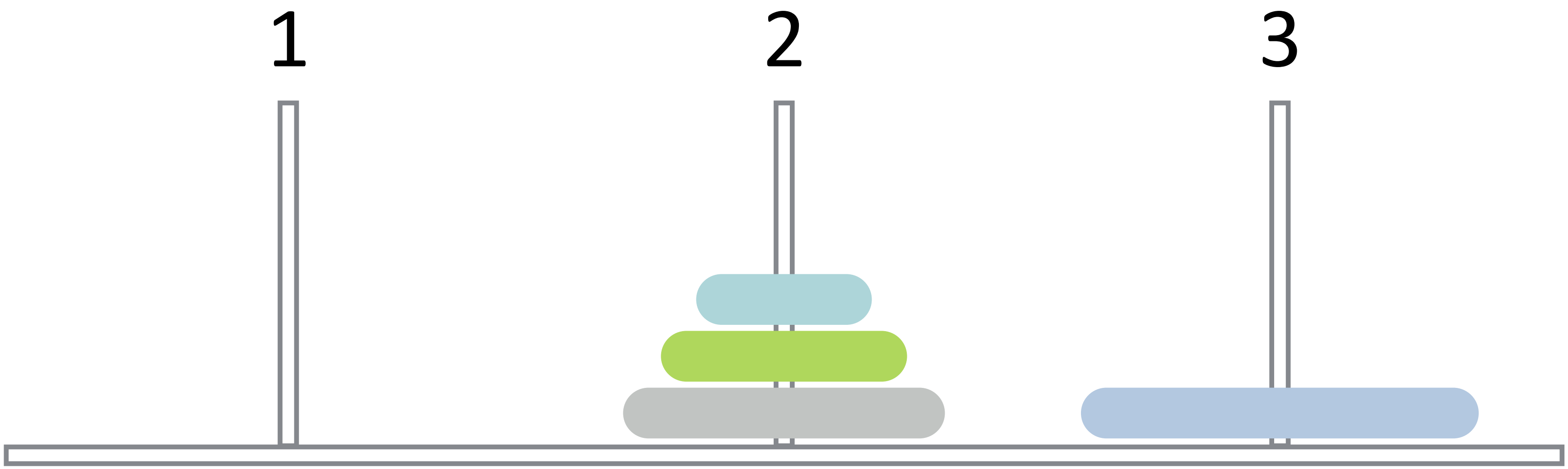
Move 3 Disks from Peg 1 to Peg 2



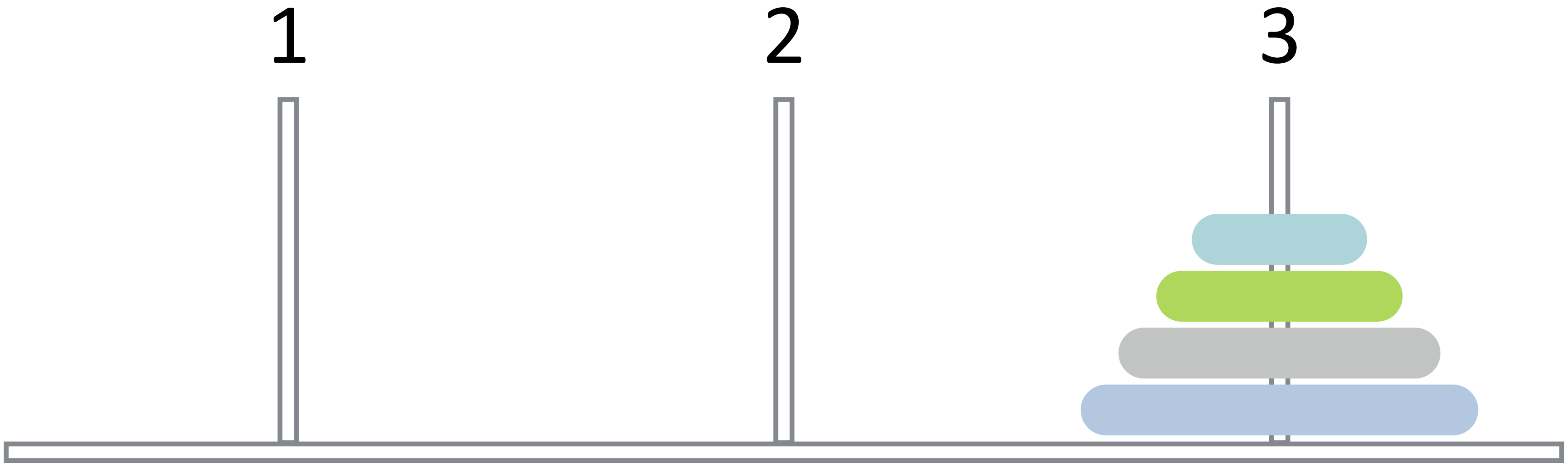
Move 1 Disk from Peg 1 to Peg 3



Move 3 Disks from Peg 2 to Peg 3



Done!



A Recursive Solution

```
void MoveTowers(int n, int start, int finish, int spare)
{
   if (n==1){
      printf("Move a disk from peg %1d to peg %1d\n", start, 
finish);
   } else {
      MoveTowers(n-1, start, spare, finish);
      printf("Move a disk from peg %1d to peg %1d\n", start, 
finish);
      MoveTowers(n-1, spare, finish, start);
   }
}
```

Analysis

- Let us now compute the **number of moves** `L(n)` that we need as a function of the number of disks `n`:

  `L(1)=1`

  `L(n)=L(n-1)+1+L(n-1)=2*L(n-1)+1, n>1`

The above are called **recurrence relations**. They can be solved to give:

$L(n)=2^n-1$

Analysis (cont'd)

- Techniques for solving recurrence relations are taught in the Algorithms and Complexity course.

- The running time of algorithm `MoveTowers` is **exponential** in the size of the input.

Readings

- T. A. Standish. *Data structures, algorithms and software principles in C*.

  Chapter 3.

- (προαιρετικά) R. Sedgewick. *Αλγόριθμοι σε C*. Κεφ. 5.1 και 5.2.