Recursion

1. Introduction to Recursion (20)

Recursion deals with how functions are called. A recursive function is one that calls itself. Here's an example of a recursive function.

```
#include <iostream>
using namespace std;

void message() {
    cout << "This is a recursive function" << endl;
    message();
}

int main() {
    message();
}
```

Do you see a problem with the above function? There's no way to stop the recursive calls. This function is like an infinite loop. What do you think will happen when you run this program?

Like a loop, a recursive function must have some method to control the number of times it repeats.

```
#include <iostream>
using namespace std;

void message(int times) {
    cout << "Called with times=" << times << endl;

    if (times > 0) {
        cout << "Recursive function call with times=" << times << endl;
        message(times - 1);
    }

    cout << "Returning with times=" << times << endl;
}

int main() {
    message(5);
}
```

Sample run:

```
Called with times=5
Recursive function call with times=5
Called with times=4
Recursive function call with times=4
Called with times=3
Recursive function call with times=3
Called with times=2
Recursive function call with times=2
Called with times=1
```



```
Recursive function call with times=1
Called with times=0
Returning with times=0
Returning with times=1
Returning with times=2
Returning with times=3
Returning with times=4
Returning with times=5
```

2. Solving Problems with Recursion

A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem. The examples in section 1 show how to write a recursive function but it doesn't show us why we would want to use a recursive function. Recursion can be a powerful tool for solving repetitive problems and is an important topic in upper-level computer science courses.

Example: Find the factorial of n.

if $n == 0$ then $n! = 1$

if $n > 0$ then $n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n$

rewrite as

if $n == 0$ then factorial(n) = 1

if $n > 0$ then factorial(n) $= 1 \times 2 \times 3 \times 4 \times \ldots \times n$

$$= n \times (n\text{-}1) \times (n\text{-}2) \times \ldots \times 3 \times 2 \times 1$$

```
// Iterative solution for finding the factorial of n
#include <iostream>
using namespace std;

int factorial(int n) {

    int product = 1;
    for (int i=1; i <= n; i++) {
        product = product * i;
    }
    return product;
}

int main() {
    cout << factorial(5);
}
```

To solve the problem using recursion, we first identify the two parts:

The base case is the part that can be solved without recursion. This is when n is equal to 0.

The recursive part is when n is greater than 0, and we can express it as

$$\text{if } n > 0 \text{ then factorial}(n) = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots \times 3 \times 2 \times 1$$

$$= n \times \text{factorial}(n - 1)$$

```
// Recursive solution for finding the factorial of n
#include <iostream>
using namespace std;

int factorial(int n) {
    cout << "Called with n=" << n << endl;
    if (n == 0) {
        return 1;                              // base case
    } else {
        cout << "Recursive function call with n=" << n << endl;
        int product = n * factorial(n - 1);    // recursive case
        cout << "Returning with n=" << n << endl;
        return product;
    }
}

int main() {
    cout << "5! = " << factorial(5) << endl;
}
```

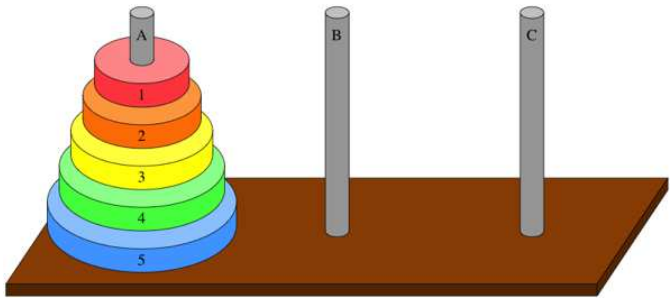
Sample run:

```
Called with n=5
Recursive function call with n=5
Called with n=4
Recursive function call with n=4
Called with n=3
Recursive function call with n=3
Called with n=2
Recursive function call with n=2
Called with n=1
Recursive function call with n=1
Called with n=0
Returning with n=1
Returning with n=2
Returning with n=3
Returning with n=4
Returning with n=5
5! = 120
```

3. Solving the Tower of Hanoi problem

All problems can be solved without using recursion. However, there are some problems where using recursion to solve them makes the code so much shorter but very difficult to understand how the code works.

The Tower of Hanoi is one such problem.



The problem is to move all the disks from peg A to peg C using an intermediate peg B. The rule is that you can only move one disk at a time and you cannot put a larger disk on top of a smaller disk.

Legend has it that in an Indian temple there was a stack of 64 golden disks. The monks have been moving these 64 golden disks according to the rule. Based on an ancient prophecy, when the last disk is move, the world will end. (The world hasn't ended yet and they are still moving the disks! See Problem 2.)

```
// Recursive solution for solving the Tower of Hanoi
#include <iostream>
using namespace std;

void move(int ndisks, int frompeg, int topeg, int temppeg) {
   if (ndisks > 1) {
      move(ndisks-1, frompeg, temppeg, topeg);
      cout << "move disk " << ndisks << " from peg " << frompeg << " to peg " << topeg
           << endl;
      move(ndisks-1, temppeg, topeg, frompeg);
   } else {
      cout << "move disk " << ndisks << " from peg " << frompeg << " to peg " << topeg
           << endl << endl;
   }
}

int main(){
   int ndisks;
   int frompeg = 1;
   int topeg = 3;
   int temppeg = 2;
   cout << "Enter the number of disks you want to move (use 25)? ";
   cin >> ndisks;
   move(ndisks, frompeg, topeg, temppeg);
   cout << "Done";
```



```
   system("pause");
}
```

Sample run:

```
Enter the number of disks you want to move (use 25)? 3
move disk 1 from peg 1 to peg 3

move disk 2 from peg 1 to peg 2
move disk 1 from peg 3 to peg 2

move disk 3 from peg 1 to peg 3
move disk 1 from peg 2 to peg 1

move disk 2 from peg 2 to peg 3
move disk 1 from peg 1 to peg 3
```

4. Problems (Problems with an asterisk are more difficult)

1. Try out the Tower of Hanoi program with 5 disks.

2. Try out the Tower of Hanoi program with 64 disks. How long do you think it'll take the computer to solve the problem?

