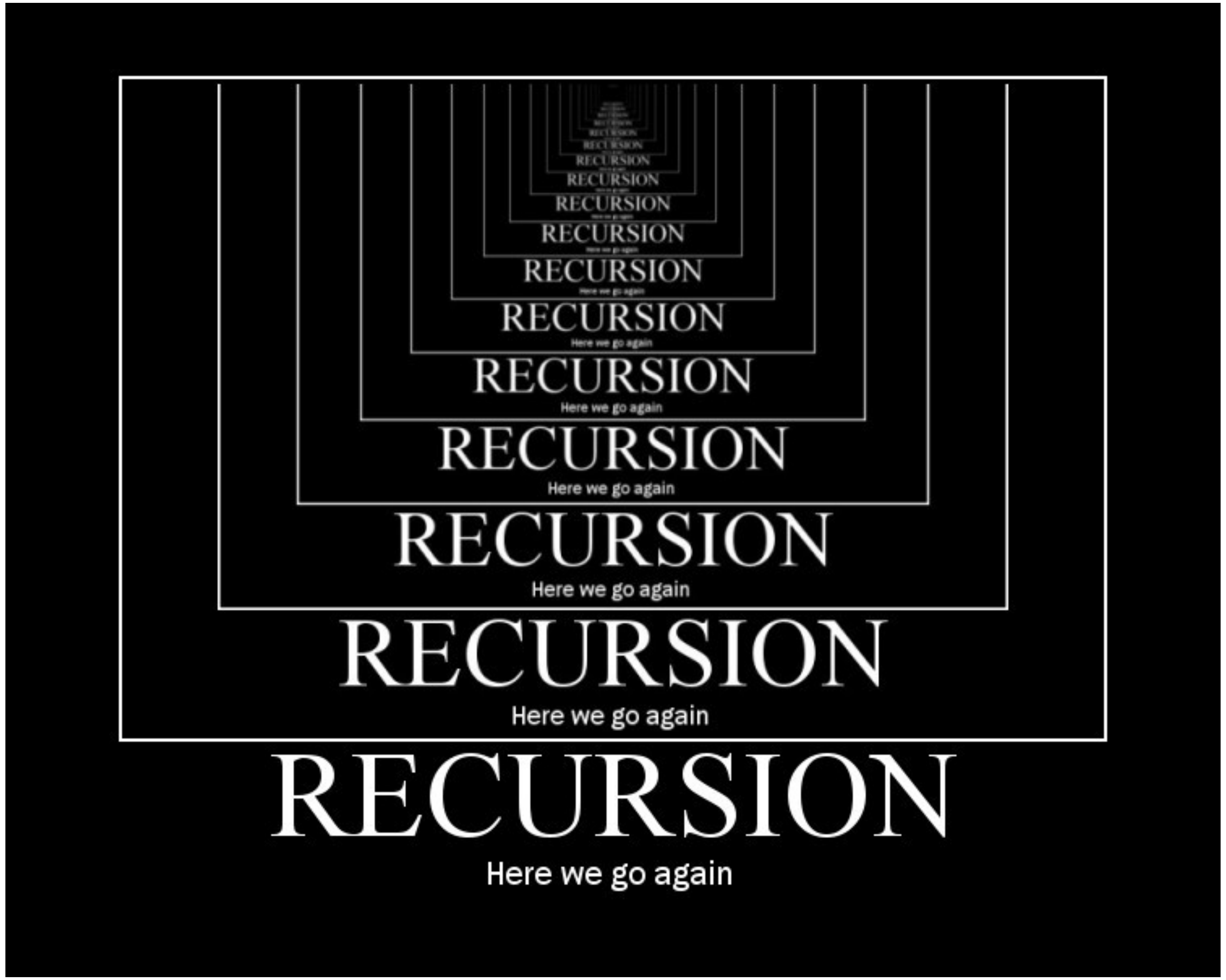
CS-2001

Programing Fundamentals

FALL 2022

Recursion

Mr. Muhammad Yousaf
National University of Computer and Emerging Sciences,
Faisalabad, Pakistan.



What is Recursion?

what is recursion

All Images Videos News Books More Tools

About 146,000,000 results (0.66 seconds)

https://www.geeksforgeeks.org › introduction-to-recursi...

Introduction to Recursion - Data Structure and Algorithm ...

19-Sept-2022 — A **recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems**. Many more ...

1): Terminates when the base case becomes t... 3): Every recursive call needs extra space i...
2): Used with functions 4): Smaller code size

What is Recursion?

Recursion:

A problem solving technique in which problems are solved by reducing them to smaller problems *of the same form*.

Why Recursion?

1. Great style
2. Powerful tool
3. Master of control flow

Pedagogy

Many simple examples

Recursion in Programming

In programming, recursion simply means that a function will call itself:

```
int main() {
     main();
     return 0;
}
```

SEG FAULT!

(this is a terrible example, and will crash!)

main() isn't supposed to call itself, but if we do write this program, what happens?

We'll get back to programming in a minute...

Recursion in Real Life?

Recursion

- How to solve a jigsaw puzzle recursively ("solve the puzzle")
  - Is the puzzle finished? If so, stop.
  - Find a correct puzzle piece and place it.
  - Solve the puzzle

ridiculously hard puzzle

In C++:

```
int numStudentsBehind(Student curr) {
if (noOneBehind(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1
   }
}
```

Recursive call!

Recursion ?

- A recursive function is a function that calls itself either directly or indirectly through another function.
  - A piece the function knows how to do and
  - A piece the function doesn’t knows how to do.

- Because this new problem looks like the original problem the function launches (calls) a fresh (new) copy of itself to go to work on smaller problems. This is referred to as a recursive call and is also called recursion step.

Recursion ?

- Given a positive integer n, n factorial is defined as the product of all integers between n and 1. for example…

5! = 5 * 4 * 3 * 2 * 1 = 120

0! = 1

so

n! = 1 if (n== 0)

n! = n * (n-1) * (n-2) * (n-3) …… * 1 if n> 0

Iteration ?

So we can present an algorithm that accepts integer n and returns the value of n! as follows.

```
int fact = 1, n;

cin>>n;

for (int i = 1; i <= n; i++)

{

   fact = fact * i;

 }
```

Such an algorithm is called iterative algorithm because it calls itself for the explicit (precise) repetition of some process until a certain condition is met.

Iterative Solution ?

In above program **n!** is calculated like as follows.

0! = 1

1! = 1

2! = 2 * 1

3! = 3 * 2 * 1

4! = 4 * 3 * 2 * 1

Recursive Solution ?

Let us see how the recursive definition of the factorial can be used to evaluate 5!.

1. 5! = 5 * 4!
2. 4! = 4 * 3!
3. 3! = 3 * 2!
4. 2! = 2 * 1!
5. 1! = 1 * 0!
6. 0! = 1

In above each case is reduced to a simpler case until we reach the case 0! Which is defined directly as 1. we may therefore backtrack from line # 6 to line # 1 by returning the value computed in one line to evaluate the result of previous line.

Recursive Solution ?

```
int factorial( int numb )
{
   if( numb <= 0 )
        return 1;
   else
        return numb * factorial( numb - 1 );
}//------------------------------------------
void main()
{
   int n;
   cout<<" \n Enter no for finding its Factorial.\n";
   cin>>n;
   cout<<"\n Factorial of "<<n<<" is : "<<factorial( n );
  return 0;
}
```

In C++:

The structure of recursive functions is typically like the following:

```
recursiveFunction() {
    if (test for simple case) {
        Compute the solution without recursion
    } else {
        Break the problem into subproblems of the same form
        Call recursiveFunction() on each subproblem
        Reassamble the results of the subproblems
    }
}
```

In C++:

Every recursive algorithm involves at least **two** cases:

- base case: The simple case; an occurrence that can be answered directly; the case that recursive calls reduce to.

- recursive case: a more complex occurrence of the problem that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem.

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

More Examples!

The power() function:

Write a recursive function that takes in a number (x) and an exponent (n) and returns the result of x^n

Powers

$$x^0 = 1$$

$$x^n = x \cdot x^{n-1}$$

Powers

Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
// second call: power (5, 2)
// third call: power (5, 1)
// fourth call: power (5, 0)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
// second call: power (5, 2)
// third call: power (5, 1)
// fourth call: power (5, 0)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

This call returns 1

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
in
in
}
```

```
// second call: power (5, 2)
i
}
```

```
// third call: power (5, 1)
int power(int x, int exp) {  if (exp
      == 0) {
     } else { return 1;
          return x * power(x, exp - 1);
     }
}
```

equals 1 from call

this entire statement returns 5 * 1

• Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)
i
n
}
```

```
// second call: power (5, 2)  int
power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 5 from call

this entire statement returns 5 * 5

Powers

- Each previous call waits for the next call to finish (just like any function).

```
cout << power(5, 3) << endl;
```

```
// first call: power (5, 3)  int power(int
x, int exp) {
      if (exp == 0) {  return 1;
      } else {
            return x * power(x, exp - 1);
      }
}
```

equals 25 from call

this entire statement returns 5 * 25

the original function call was to this one, so it returns 125, which is 5^3

```
int power(int x, int exp) {  if(exp == 0)
      {
            // base case  return
            1;
      } else {
            if (exp % 2 == 1) {
            // if exp is odd
                  return x * power(x, exp - 1);
            } else {
                  // else, if exp is even  int y =
                  power(x, exp / 2);  return y * y;
            }
      }
}
```

Exponentiation by squaring

Big O???

O(log n) -- yay!

Mystery Recursion: Trace this function

```
int mystery  (int n) {
     if (n <10) {
          return n;
     }
     else {
          int a = n/10;
          int b = n % 10;
          return mystery(a + b);
     }
}
```

What is the result of mystery(648)?

A. 8
B. 9
C. 54
D. 72
E. 648

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
        if (n < 10) {
            return n;
        } else {
            int a = n/10; // a = 64
            int b = n % 10; // b = 8
            return mystery(a + b); // mystery(72);
        }
}
```

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
    int mystery(int n) { // n = 72
            if (n < 10) {
             return n;
        } else {
             int a = n/10; // a = 7
             int b = n % 10; // b = 2
             return mystery(a + b); // mystery(9);
}       }
    }
```

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
int mystery(int n) { // n = 72
int mystery(int n) { // n = 9
        if (n < 10) {
          return n; // return 9;
        } else {
          int a = n/10;
          int b = n % 10;
          return mystery(a + b);
        }
}
}
}
```

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
    int mystery(int n) { // n = 72
            if (n < 10) {
              return n;
        } else {
              int a = n/10; // a = 7
              int b = n % 10; // b = 2
}             return mystery(a + b); // mystery(9);
        }
                                returns 9
    }
```

Mystery Recursion: Trace this function

```
int mystery(int n) { // n = 648
        if (n < 10) {
            return n;
    } else {
        int  a = n/10; // a  = 64
        int  b = n % 10; // b =  8
          return mystery(a + b); //
    }
}
```

returns 9

What is the result of mystery(648)?

A. 8

B. 9

C. 54

D. 72

E. 648

More Examples! isPalindrome(string s)

Write a recursive function isPalindrome accepts a string and returns true if it reads the same forwards as backwards.

isPalindrome("**madam**") → true
isPalindrome("**racecar**") → true
isPalindrome("**step on no pets**") → true
isPalindrome("**Java**") → false
isPalindrome("**byebye**") →false

Three Musts of Recursions

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

isPalindrome(string s)

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
bool isPalindrome(const string& s) {
    if (s.length() < 2) { // base case
        return true;
    } else { // recursive case
        if (s[0] != s[s.length() - 1]) {
        return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
```

Flashback to 106A: Hailstone

```
// Couts the sequence of numbers from n to one
// produced by the Hailstone (aka Collatz) procedure
void hailstone(int n) {
	cout << n << endl;
	if(n == 1) {
		return;
	} else {
		if(n % 2 == 0) {
			// n is even so we repeat with n/2
			hailstone(n / 2);
		} else {
			// n is odd so we repeat with 3 * n + 1
			hailstone(3 * n + 1);
		}
	}
}
```

Flashback to 106A: Hailstone

```
// Couts the sequence of numbers from n to one
// produced by the Hailstone (aka Collatz) procedure
void hailstone(int n) {
    cout << n << endl;
    if(n == 1) {
        return;
```

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

```
            // n is odd so we repeat with 3 * n + 1
            hailstone(3 * n + 1);
        }
    }
}
```

Is this simpler???

Flashback to 106A: Hailstone

`hailstone(int n)`

Hailstone has been checked for values up to 5×10^{18} but

no one has proved that it always reaches 1!

There is a cash prize for proving it!

The prize is $1400.

Flashback to 106A: Hailstone

Print the sequences of numbers that you take to get from N until 1, using the Hailstone (Collatz) production rules:

If n == 1, you are done.
If n is even your next number is n / 2.
If n is odd your next number is 3*n + 1.

n=3; 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, …
n=4; 2, 1, 4, 2, 1, …
n=7; 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1,

Fibonacci Sequence: Recursion

Print the nth terms of a fibonacci sequence using recursion:

- Fibonacci Series:
  - 0,1,1,2,3,5,8,13,21,.....etc
- first two terms:
  - fo = 0, f1 = 1

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

Fibonacci Sequence: Recursion

```
int fibonacci(int n)
{
    if (n <=1)
        return n; // base cases
    else
        return fibonacci(n-1) + fibonacci(n-2); // recursive step
}
int main()
{
    cout << fibonacci(6) <<endl;
    return 0;
}
```

Towers of Hanoi: C++ Code

```
void TOH(int n,char Sour, char Aux,char Des)
{
    if(n==1)
        { cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;      return; }

    TOH(n-1,Sour,Des,Aux);
    cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
    TOH(n-1,Aux,Sour,Des);
}

int main()
{
    int n;
    cout<<"Enter no. of disks:";
    cin>>n;
    TOH(n,'A','B','C');

    return 0;
}
```

Efficiency of Recursion

- In general, a non recursive version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a new block (system stack) is avoided in the non recursive version.

- However, sometimes a recursive solution is the most natural and logical way of solving a problem as we will soon observe while studying different Tree data structures.

Recap

- **Recursion**
  - Break a problem into smaller subproblems of the same form, and call the same function again on that smaller form.
  - Super powerful programming tool
  - Not always the perfect choice, but often a good one
  - Some beautiful problems are solved recursively

- **Three Musts for Recursion:**

1. Your code must have a case for all valid inputs
2. You must have a base case that makes no recursive calls
3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

Practice Tasks

1. Given a string as input, find all the subsequences of word, separated by commas,

 * where a subsequence is a string of letters found in word

 * in the same order that they appear in word.

subsequences("**abc**") might return "**abc,ab,bc,ac,a,b,c,”**