CPS 188

Computer Programming Fundamentals

Prof. Alex Ufkes

Topic 9.1: Recursion

Toronto Metropolitan University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

Previously

Functions
&
Looping

Previously: Functions

Breaking a large problem into smaller sub-problems

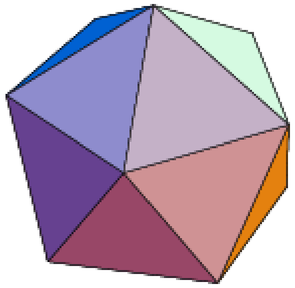
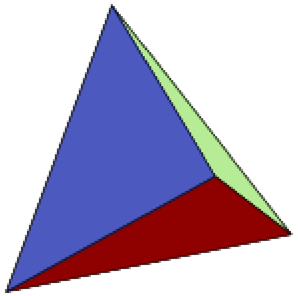
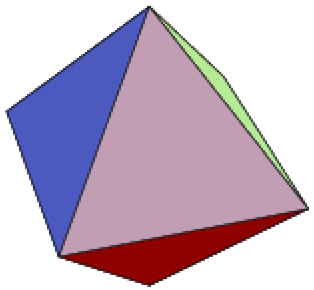
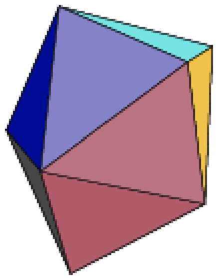
- We’ve written many functions to solve many problems.
- Many of these have themselves called other functions!
- It is of course perfectly legal for one function to call another.
- Feel free to write helper functions for the graded problems.
- **However!** Even if we will never reuse the code, writing helper functions can aid us in simplifying the problem.
- Two smaller problems are often easier to solve than one larger problem

Functions Calling Functions

Consider the following:

<u>Write a program to do the following:</u> Compute the surface area of a deltahedron, given the number of faces and the edge length.

A deltahedron is a solid whose faces are all equilateral triangles

To `main()`

From `main()`

```
double deltahedron_area(double len, int nFaces)
{
      double surfaceArea;
      surfaceArea = nFaces * equTriArea(len);
      return surfaceArea;
}

double equTriArea(double sideLength)
{
      double a;
      a = (sqrt(3)/4)*sideLength*sideLength;
      return a;
}
```

```c
#include <stdio.h>
#include <math.h>

double deltahedron_area(double len, int nFaces);
double equTriArea(double sideLength);

int main (void)
{
	int n;
	double length, sa;

	printf("Enter number of faces, length of sides");
	scanf("%d %lf", &n, &length);

	sa = deltahedron_area(length, n);

	printf("Surface area is %.2lf\n", sa);

	return 0;
}
```

Previously

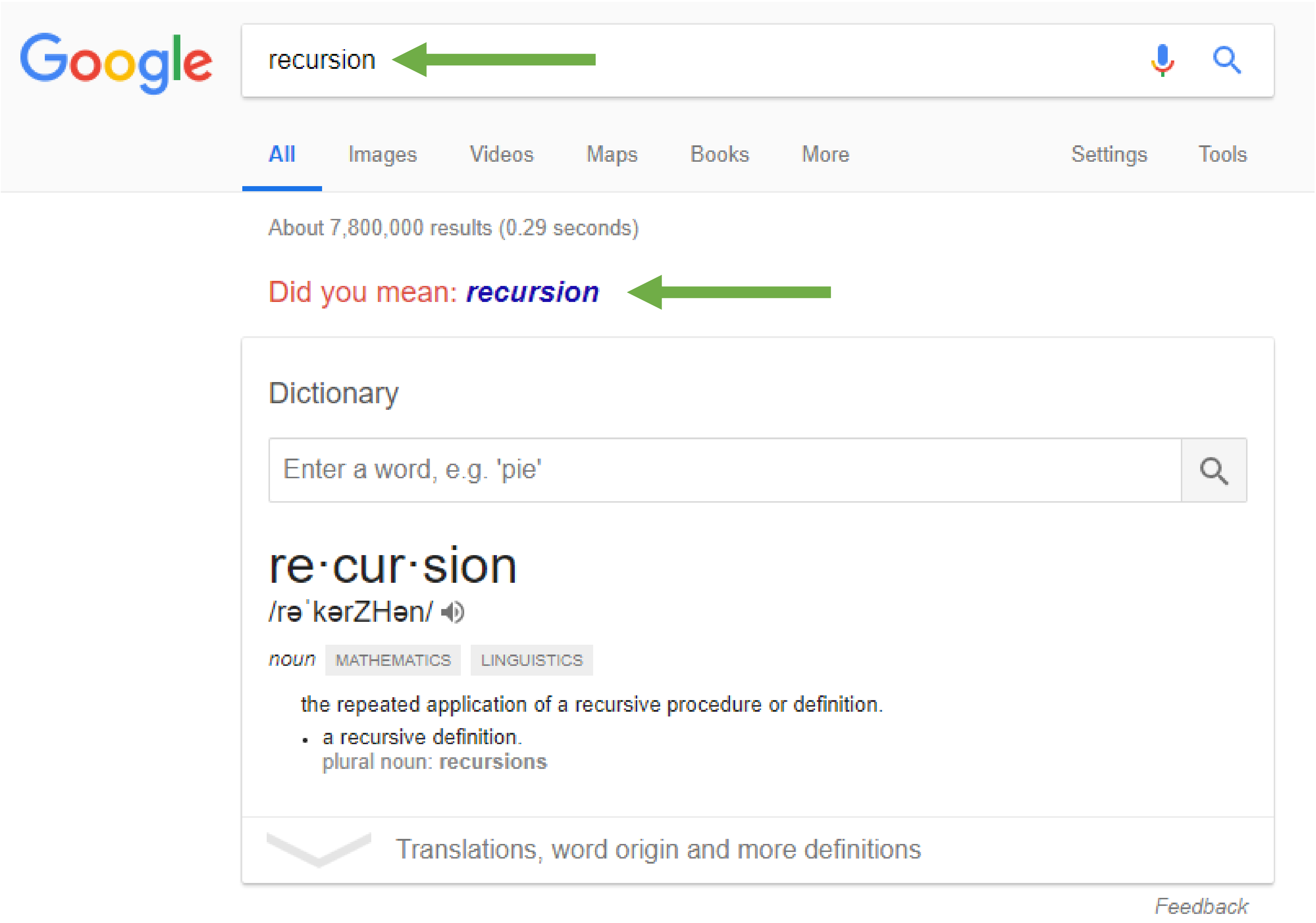
```
int num, i, fact = 1;
printf("Enter a number: ");
scanf("%d", &num);

for (i = 1; i <= num; i++)
{
    fact = fact * i;
}
printf("Factorial of %d is %d \n", num, fact);
```


BORING

Today

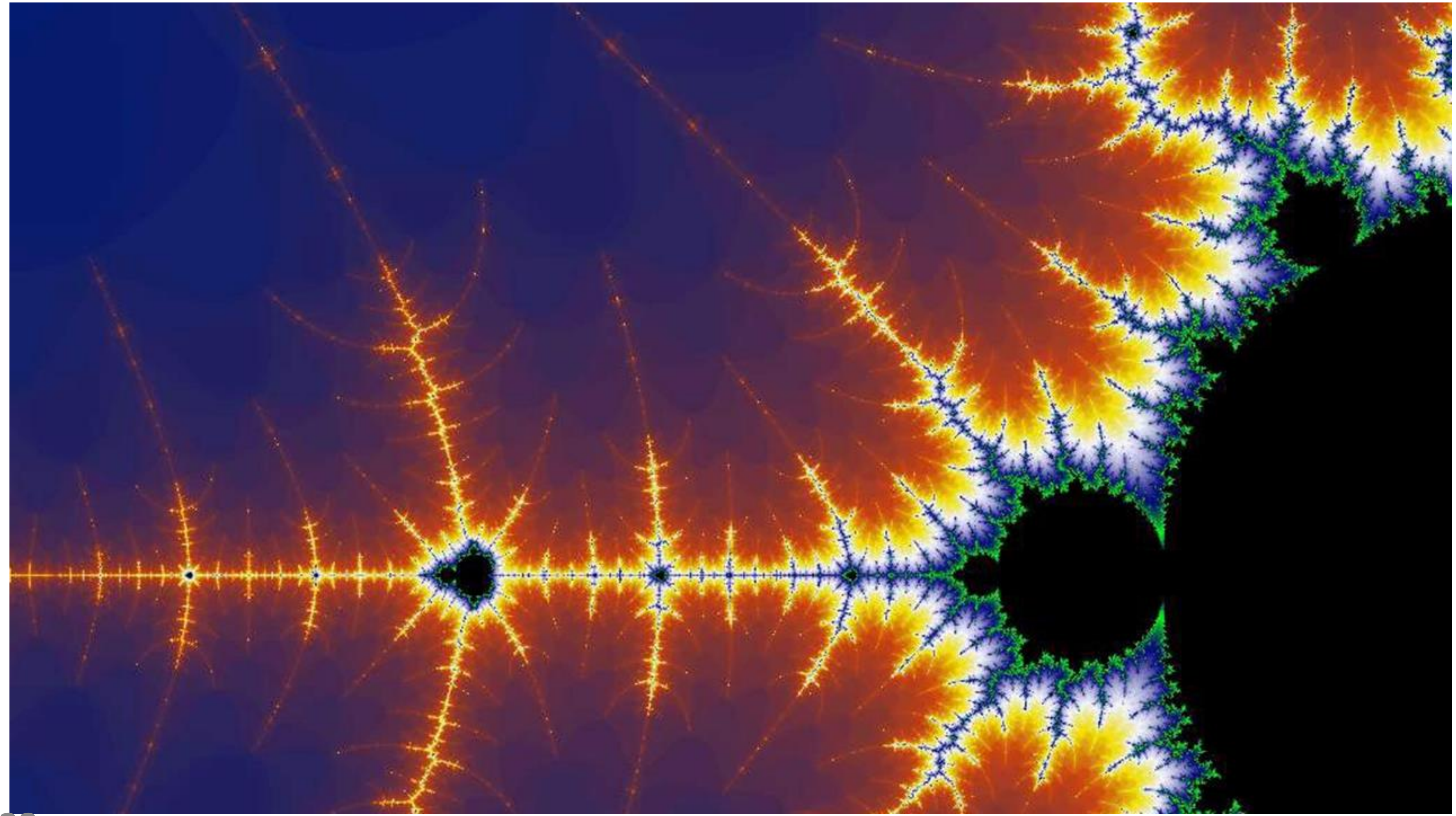
Recursion

Recursion



recursion
All
Images
Videos
Maps
Books
More
Settings
Tools
About 7,800,000 results (0.29 seconds)
Did you mean: recursion
Dictionary
Enter a word, e.g. 'pie'
re·cur·sion
/rəˈkərZHən/
noun
MATHEMATICS
LINGUISTICS
the repeated application of a recursive procedure or definition.
a recursive definition.
plural noun: recursions
Translations, word origin and more definitions
Feedback

Self-Similar Problems



Self-Similar Problems

- A thing is said to be self-similar if it contains a strictly smaller version of itself inside it as a proper part.
- Each Fibonacci number is the sum of the previous two
- Every factorial is created from a smaller factorial (5! = 5*4!).
- Recursion, a function calling itself with ever-smaller argument values, is often a natural way to solve self-similar problems.
- ***<u>Assuming</u>*** that we can spot the underlying self-similarity.
- This is often the most difficult part!

Recursion

Defining a function (or method) in terms of itself

In other words, a function that calls itself

Recall our factorial solution…

Factorial:

The product of all positive integers less than or equal to a given non-negative number.

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

We've already seen the iterative solution…

```c
int factorial (int n)
{
    int i, fact = 1;
    for (i = n; i > 1; i--)
        fact = fact * i;
    return fact;
}
```

Iteration VS Recursion

Every **iterative** solution has a semantically equivalent **recursive** solution.

Every **recursive** solution has a semantically equivalent **iterative** solution.

Some problems are easier to solve iteratively, others are
(much, *much*) easier to solve recursively.

(What does "easier" mean here? Generally, simpler for the programmer)

Factorial can be solved easily either way, so let's try!

Factorial: Iterative definition

`n! = n*(n-1)*(n-2)*…*3*2*1`

Factorial: Recursive definition

`n! = n*(n-1)!`

We've defined `n!` in terms of another (*smaller*) factorial

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case**
2. Contain a **recursive case**
3. Make **progress** towards the base case

Factorial: Base Case

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

5! = 5*4!

5! = 5*4*3!

5! = 5*4*3*2!

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

5! = 5*4*3*2*1*0!
```

- Factorial is not defined for negative integers.
- Our definition ends here!
- 0! Is called our ***base case***.
- **`0! = 1`**
- The value of the base case must ***NOT*** contain a factorial!

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case** $\boxed{0! = 1}$
2. Contain a **recursive case**
3. Make **progress** towards the base case

Factorial: Recursive Case

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

Can we generalize for any factorial?

```
n! = n * (n-1)!
```

- We call this the ***recursive case***.
- Factorial is defined in terms of another factorial.

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case**
2. Contain a **recursive case**
3. Make **progress** towards the base case

```
0! = 1
```

```
n! = n*(n-1)!
```

Factorial: Progress Towards Base Case?

$$n! = n*(n-1)!$$

$$n! = n*\overbrace{(n-1)*(n-2)!}$$

$$n! = n*(n-1)*\overbrace{(n-2)*\underline{(n-3)!}}$$

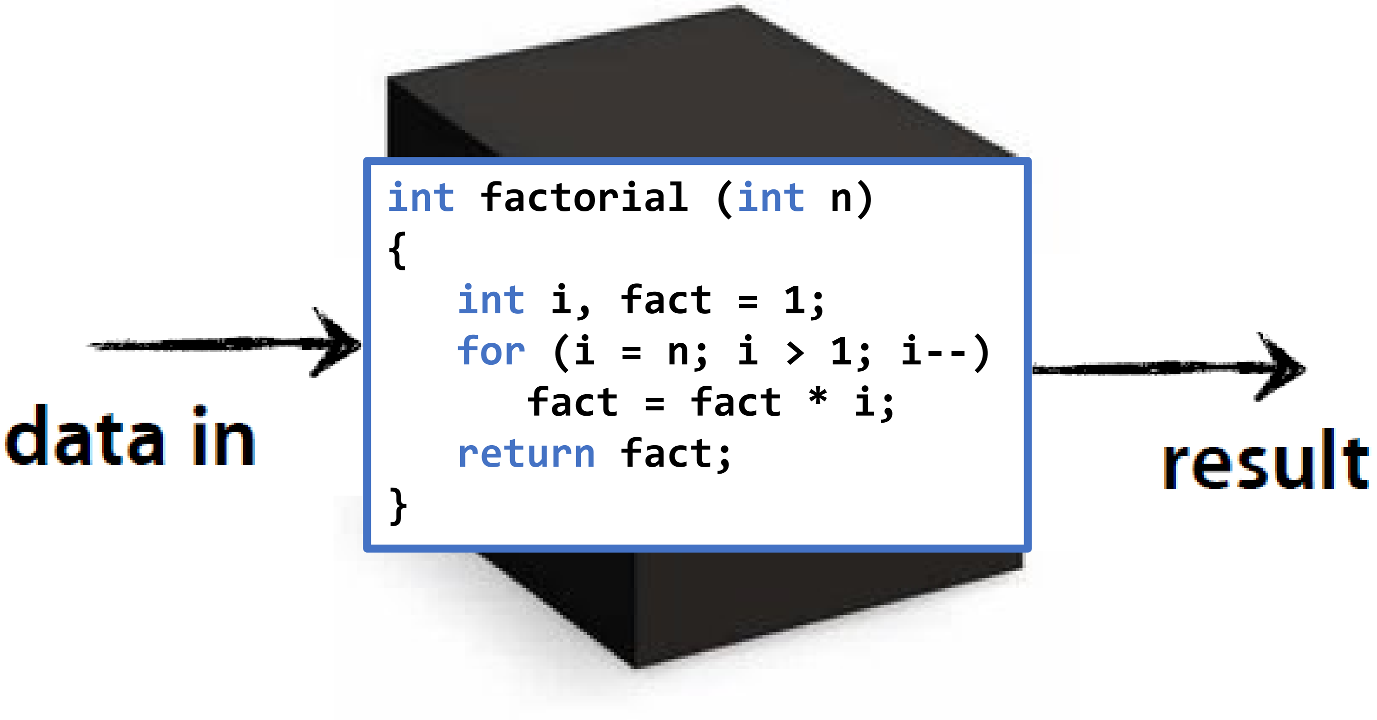
- If we continue unwrapping this, will we eventually hit the base case?
- In other words, will the above term eventually be **0!** for any **n**?

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case** `0! = 1`
2. Contain a **recursive case** `n! = n*(n-1)!`
3. Make **progress** towards the base case `Yes.`

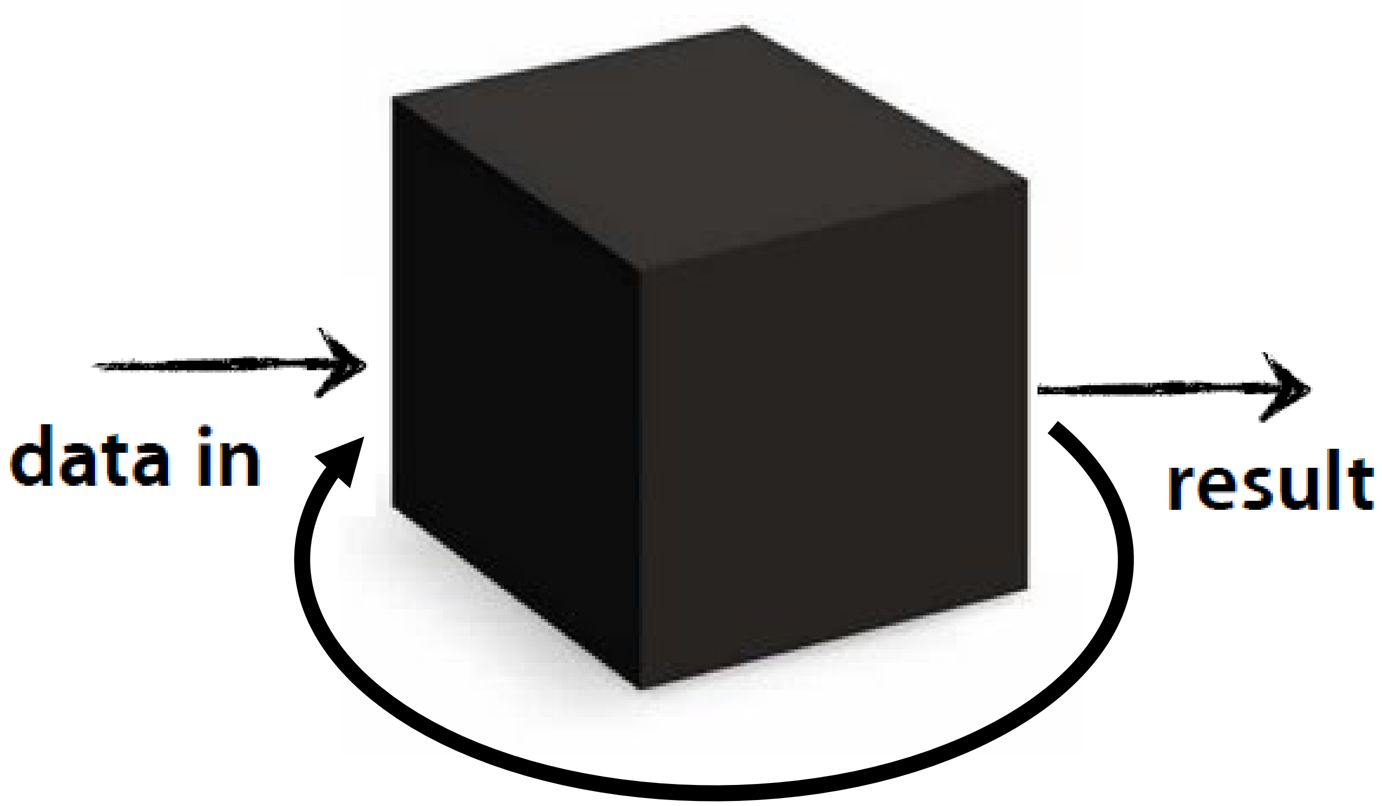
Recall, of Functions




```
int factorial (int n)
{
   int i, fact = 1;
   for (i = n; i > 1; i--)
      fact = fact * i;
   return fact;
}
```

- Simple enough to conceptualize iteratively.
- What about recursively? The situation looks a bit different…

Recursive Functions



Recursive call…?

All well and good… How do we code this?

Iterative Solution:

```
int factorial (int n)
{
    int i, fact = 1;
    for (i = n; i > 1; i--)
        fact = fact * i;
    return fact;
}
```

Recursive Solution:

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}
```

recursion.c

```c
#include <stdio.h>

int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n*factorial(n-1);
}

int main (void)
{
    printf("%d\n", factorial(0));
    printf("%d\n", factorial(1));
    printf("%d\n", factorial(2));
    printf("%d\n", factorial(3));
    printf("%d\n", factorial(4));
    printf("%d\n", factorial(5));
    printf("%d\n", factorial(6));

    return (0);
}
```

Base Case

Recursive Case

C:\WINDOWS\SYSTEM32\cmd.exe

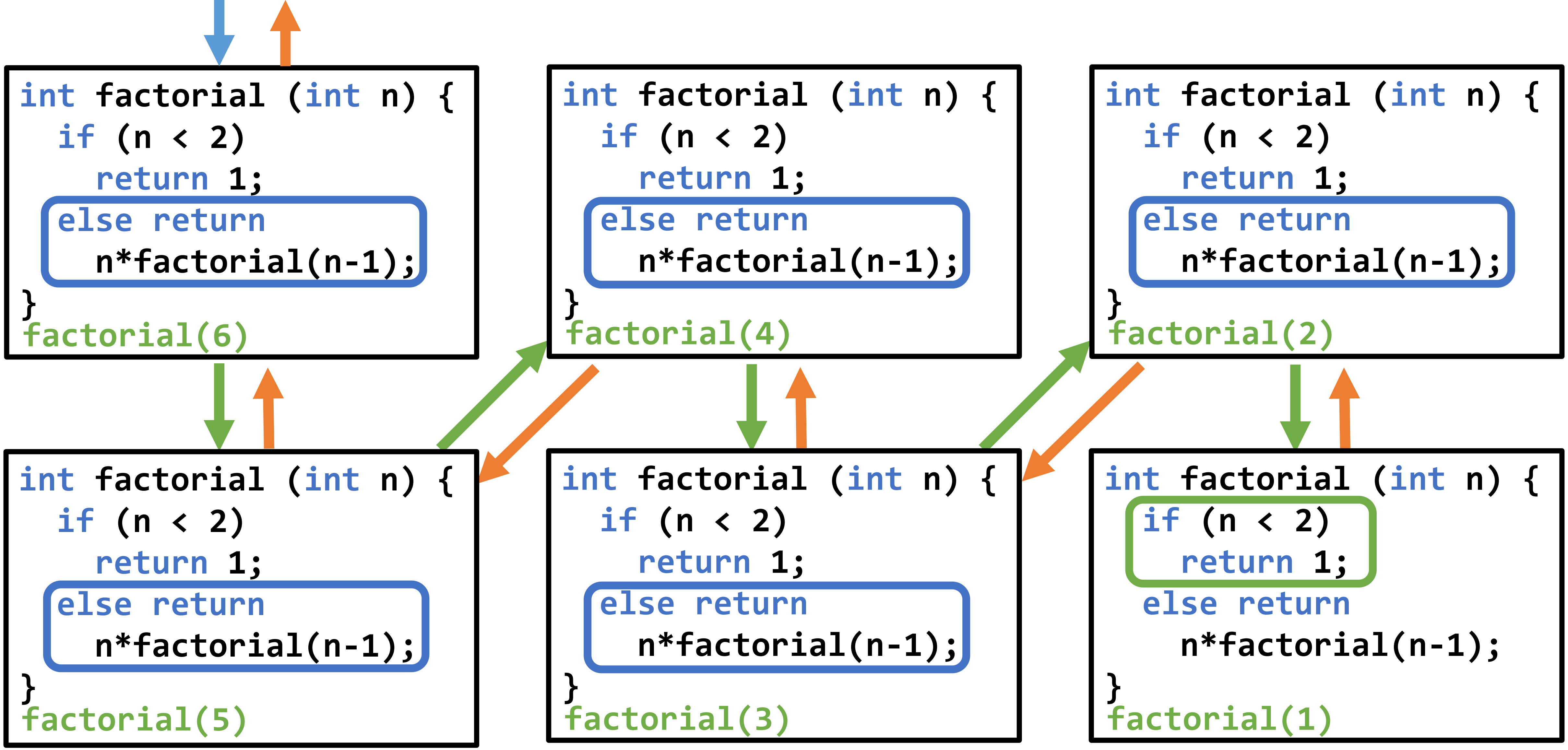
```
1
1
2
6
24
120
720
```

First call is to `factorial(6)`

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return n * factorial(n-1);
}
```

False!

- `factorial(6)` calls `factorial(5)`
- We don't actually return yet!
- We *can't* return until `factorial(5)` is computed



- We have six *unique* function instances! Each has its own stack frame.
- They are all **different** *instances* of the **same** *function*.

Call factorial(6) from outside

Return to outside

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return
        n*factorial(n-1);
}
```

factorial(6) — return 6*factorial(5)

factorial(5) — return 5*factorial(4)

factorial(4) — return 4*factorial(3)

factorial(3) — return 3*factorial(2)

factorial(2) — return 2*factorial(1)

factorial(1) — return 1

Base case! No recursive call!

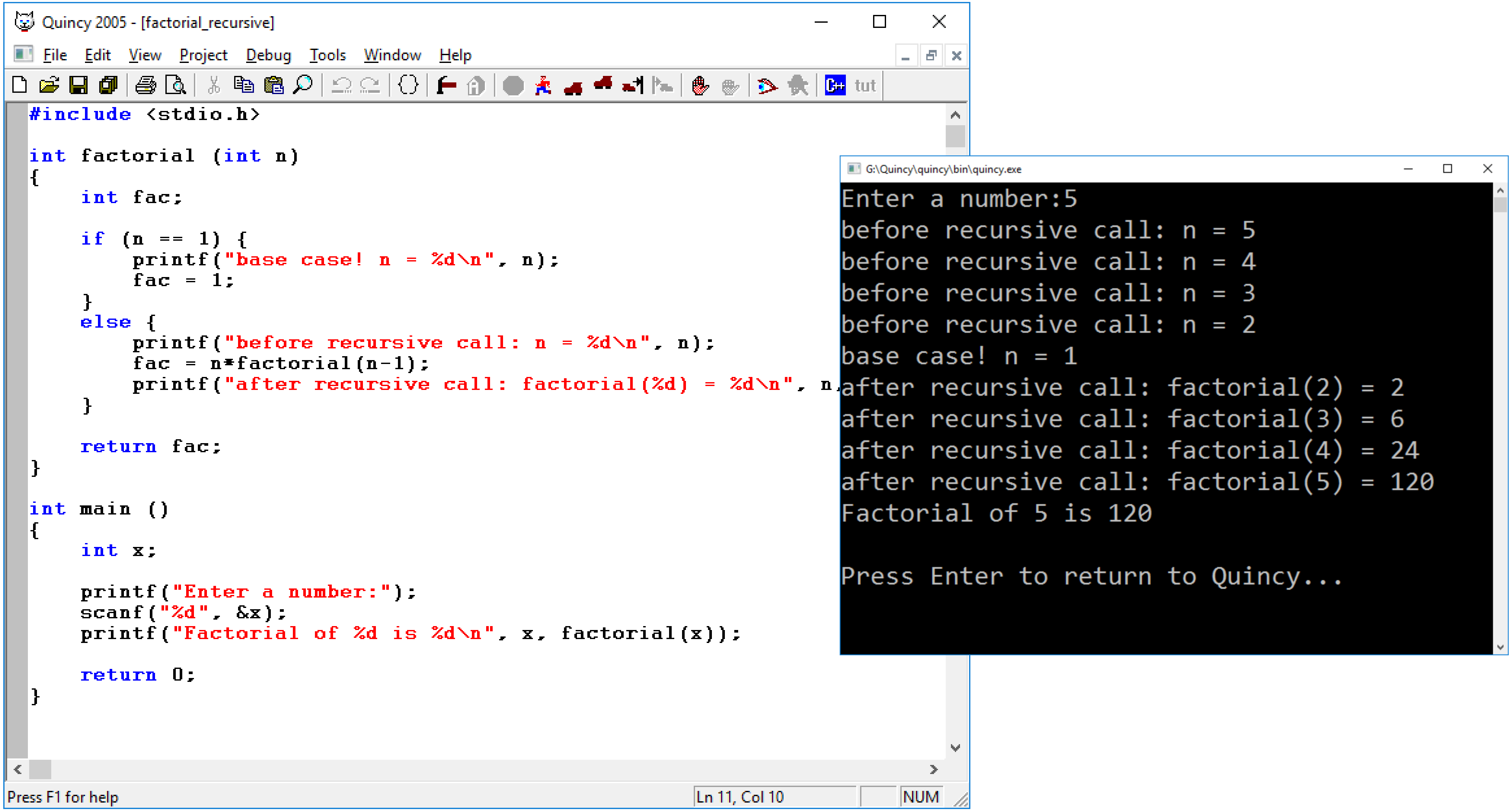
Function Call Trace

factorial(6) → factorial(6)

```
int factorial (int n)
{
    if (n < 2)
        return 1;
    else
        return
        n*factorial(n-1);
}
```

```
factorial(6)
  factorial(5)
    factorial(4)
      factorial(3)
        factorial(2)
          factorial(1)      <- Base case!
          return 1
        return 2*factorial(1) = 2
      return 3*factorial(2) = 6
    return 4*factorial(3) = 24
  return 5*factorial(4) = 120
return 6*factorial(5) = 720
```

return to outside ← return 6*factorial(5) = 720


Quincy 2005 - [factorial_recursive]
File Edit View Project Debug Tools Window Help
tut
#include <stdio.h>
int factorial (int n)
{
int fac;
if (n == 1) {
printf("base case! n = %d\n", n);
fac = 1;
}
else {
printf("before recursive call: n = %d\n", n);
fac = n*factorial(n-1);
printf("after recursive call: factorial(%d) = %d\n", n,
}
return fac;
}
int main ()
{
int x;
printf("Enter a number:");
scanf("%d", &x);
printf("Factorial of %d is %d\n", x, factorial(x));
return 0;
}
Press F1 for help
Ln 11, Col 10
NUM
G:\Quincy\quincy\bin\quincy.exe
Enter a number:5
before recursive call: n = 5
before recursive call: n = 4
before recursive call: n = 3
before recursive call: n = 2
base case! n = 1
after recursive call: factorial(2) = 2
after recursive call: factorial(3) = 6
after recursive call: factorial(4) = 24
after recursive call: factorial(5) = 120
Factorial of 5 is 120
Press Enter to return to Quincy...


ANOTHER!

Another!

Write two functions, one **iterative** and one **recursive**, to implement multiplication using addition. Assume operands are >= 0.

```
3*5 = 3 + 3 + 3 + 3 + 3
```

Iterative Multiplication

```
int mult (int a, int b)
{
    int product = 0, i;

    for (i = 0; i < b; i++)
    {
        product += a;
    }

    return product;
}
```

Easy.

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case**
2. Contain a **recursive case**
3. Make **progress** towards the base case

Recursive Multiplication

```
3*5 = 3 + 3 + 3 + 3 + 3
```

3*4

```
3*5 = 3 + 3*4
```

```
a*b = a + a*(b-1)
```

recursive definition

multiplication defined in terms of another multiplication.

Unwrap the recursion until we hit the end

```
a*b = a + a*(b-1)
a*5 = a + a*4
a*4 = a + a*3
a*3 = a + a*2
a*2 = a + a*1
a*1 = a
```

recursive cases

Base case

Recursion Properties

Recursive solutions **must** satisfy these three rules:

1. Contain a **base case**
2. Contain a **recursive case**
3. Make **progress** towards the base case

`a*1 = a`

`a*b = a + a*(b-1)`

Start at b>=1, go to b-1, eventually we will hit b=1

Recursive Multiplication

```
int mult (int a, int b)
{
    if (b == 1)
        return(a);
    else
        return(a + mult(a, b - 1));
}
```

base case

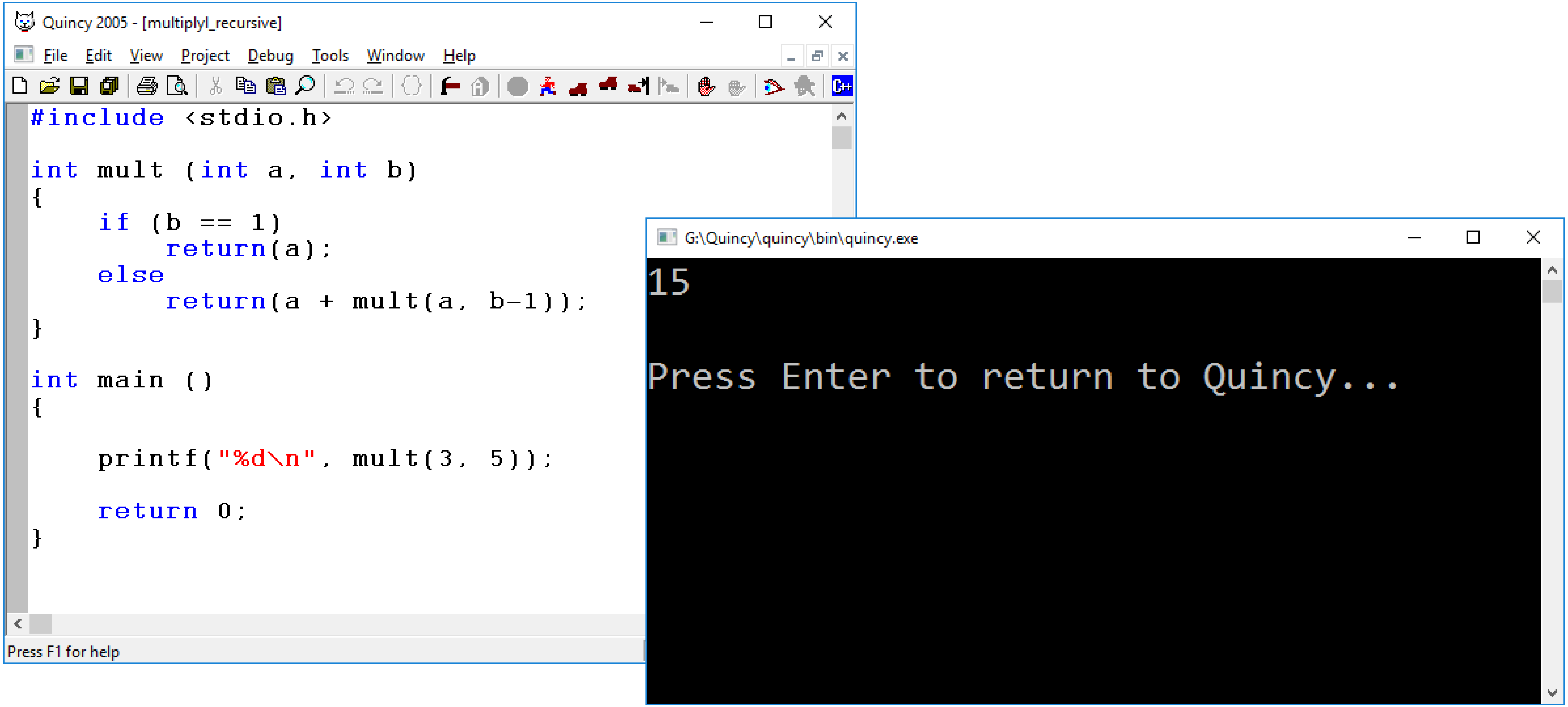
recursive case


```
int mult (int a, int b)
{
  if (b == 1)
    return(a);
  else
    return(a + mult(a, b-1));
}
```

mult(3,5) → mult(3,5)

```
mult(3,5)
  mult(3,4)
    mult(3,3)
      mult(3,2)
        mult(3,1)   ← Base case!

        return 3
      return 3 + mult(3,1) = 6
    return 3 + mult(3,2) = 9
  return 3 + mult(3,3) = 12
return 3 + mult(3,4) = 15
```

Return to outside ←


Quincy 2005 - [multiplyl_recursive]
File Edit View Project Debug Tools Window Help
#include <stdio.h>
int mult (int a, int b)
{
if (b == 1)
return(a);
else
return(a + mult(a, b-1));
}
int main ()
{
printf("%d\n", mult(3, 5));
return 0;
}
Press F1 for help
G:\Quincy\quincy\bin\quincy.exe
15
Press Enter to return to Quincy...


ANOTHER!

Another!

Write two functions, one **iterative** and one **recursive**, to find the sum of digits in an integer.

543

5 + 4 + 3 = 12

Iterative Digit Sum

```
int sum_digits(int n)
{
	int sum = 0;
	while (n != 0) {
		sum += n % 10;
		n /= 10;
	}
	return sum ;
}
```

Retrieve last digit

Remove last digit (integer division!)

recursion.c

```c

int sum_digits(int n)
{
    int sum = 0;
    while (n != 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum ;
}

int main (void)
{
    printf("%d\n", sum_digits(123));
    printf("%d\n", sum_digits(234));
    printf("%d\n", sum_digits(345));
    printf("%d\n", sum_digits(999));
    printf("%d\n", sum_digits(101));

    return (0);
}

```

C:\WINDOWS\SYSTEM32\cmd.exe

```
6
9
12
27
2



------------------
(program exited with code: 0)
```

Recursive Digit Sum

```
int sum_digits(int n)
{
    if (n == 0)
        return 0;
    return n%10 + sum_digits(n/10);
}
```

Base Case

Recursive Case: Retrieve last digit, make recursive call with last digit stripped off.

recursion.c

```c

int sum_digits(int n)
{
    if (n == 0)
        return 0;
    return n%10 + sum_digits(n/10);
}

int main (void)
{
    printf("%d\n", sum_digits(123));
    printf("%d\n", sum_digits(234));
    printf("%d\n", sum_digits(345));
    printf("%d\n", sum_digits(999));
    printf("%d\n", sum_digits(101));

    return (0);
}

```

C:\WINDOWS\SYSTEM32\cmd.exe

```
6
9
12
27
2


-------------------
(program exited with code: 0)

Press any key to continue . . .
```

Recursion or Iteration?

- Entirely problem dependent.
- Some tasks are far easier to solve recursively.
- Others are far easier to solve iteratively.
- One (or few) base cases? Obvious recursive case? Try recursion.
- Otherwise try iteration.
- Recursive functions can be very elegant, but don’t force it

Questions?