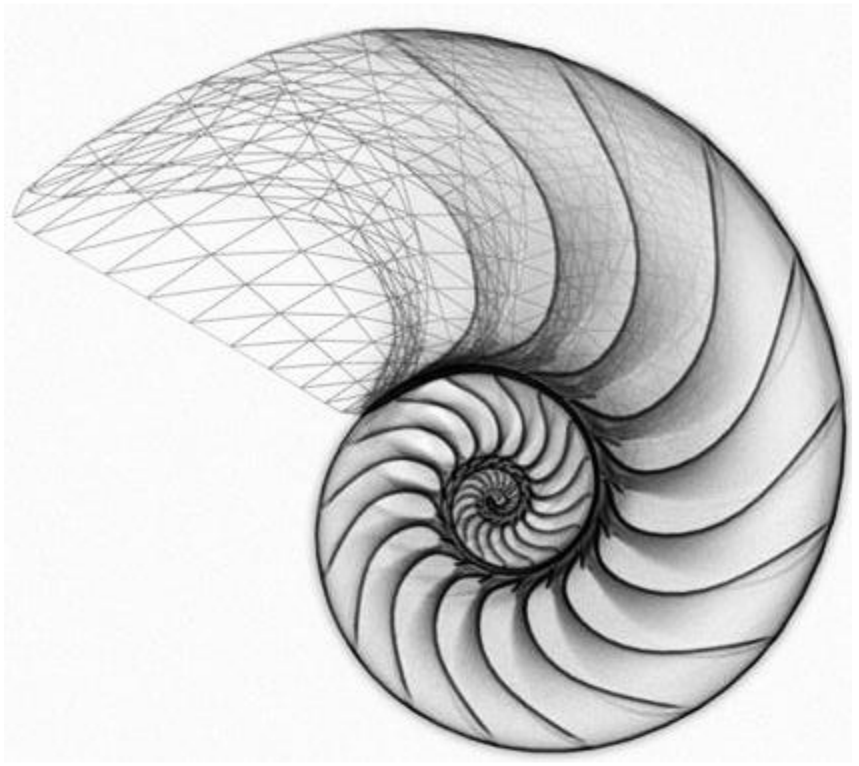


Lecture 11

Recursion

CSE115: Computing Concepts

A Look Back at Functions

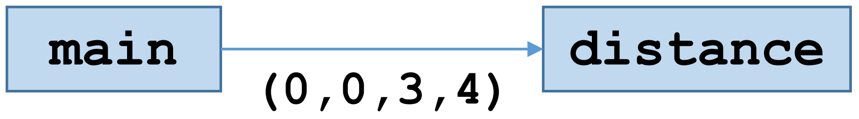
```
#include <stdio.h>
#include <math.h>
double distance(double x1, double y1, double x2, double y2)
{
        double dist;
        dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
        return dist;
}
int main()
{
        double milage = distance(0, 0, 3, 4);
        printf("Milage: %lf\n",milage);
        return 0;
}
```

A Look Back at Functions

```c
#include <stdio.h>
#include <math.h>
double distance(double x1, double y1, double x2, double y2)
{
        double dist;
        dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
        return dist;
}
int main()
{
        double milage = distance(0, 0, 3, 4);
        printf("Milage: %lf\n",milage);
        return 0;
}
```

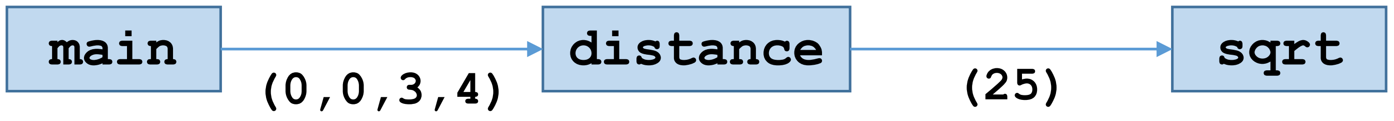
A Look Back at Functions

```
#include <stdio.h>
#include <math.h>
double distance(double x1, double y1, double x2, double y2)
{
        double dist;
        dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
        return dist;
}
int main()
{
        double milage = distance(0, 0, 3, 4);
        printf("Milage: %lf\n",milage);
        return 0;
}
```



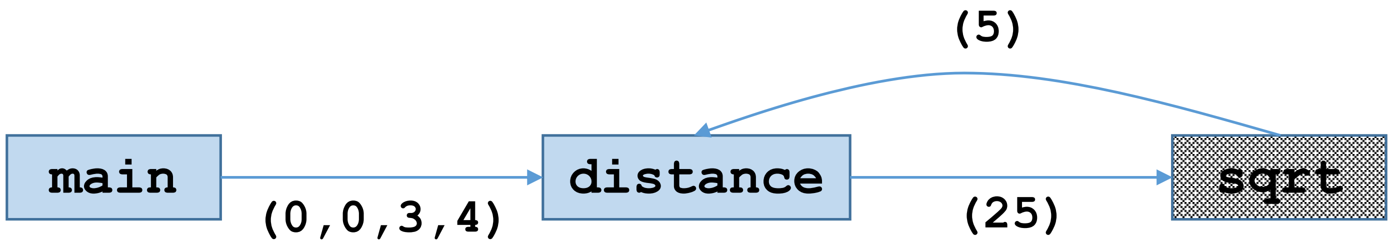
A Look Back at Functions

```
#include <stdio.h>
#include <math.h>
double distance(double x1, double y1, double x2, double y2)
{
        double dist;
        dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
        return dist;
}
int main()
{
        double milage = distance(0, 0, 3, 4);
        printf("Milage: %lf\n",milage);
        return 0;
}
```



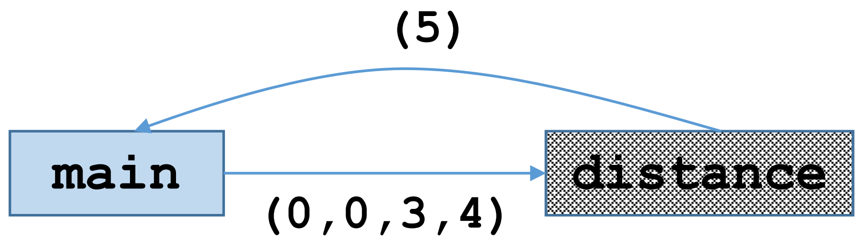
A Look Back at Functions

```
#include <stdio.h>
#include <math.h>
double distance(double x1, double y1, double x2, double y2)
{
        double dist;
        dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
        return dist;
}
int main()
{
        double milage = distance(0, 0, 3, 4);
        printf("Milage: %lf\n",milage);
        return 0;
}
```



A Look Back at Functions

```
#include <stdio.h>
#include <math.h>
double distance(double x1, double y1, double x2, double y2)
{
        double dist;
        dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
        return dist;
}
int main()
{
        double milage = distance(0, 0, 3, 4);
        printf("Milage: %lf\n",milage);
        return 0;
}
```



Recursion

- ***Recursion*** is a technique that solves a problem by solving a smaller problem of the same type.
- Recursion can be implemented using a ***recursive function*** (a function invoking itself, either directly or indirectly).
- Recursion can be used as an alternative to iteration.
- It is an important and powerful tool in problem solving and programming.
- It is a programming technique that naturally implements the divide-and-conquer problem solving methodology.

Factorial – Iterative Definition

n! = n * (n-1) * (n-2) * … * 2 * 1 for any integer n>0

0! = 1

Iterative Definition in C:

```
fval = 1;
for (i = n; i >= 1; i--)
   fval = fval * i;
```

Factorial – Recursive Definition

- To define ***n!*** recursively, ***n!*** must be defined in terms of the factorial of a smaller number.
- Observation (problem size is reduced):

  n! = n * (n-1)!
- Base case: 0! = 1
- We can reach the base case, by subtracting 1 from n if n is a positive integer.

Recursive Definition:

n! = 1 if *n = 0*

n! = n(n-1)!* if *n > 0*

The Nature of Recursion

1. One or more simple cases of the problem (called the *stopping cases or base case*) have a simple non-recursive solution.
2. The other cases (general cases) of the problem can be reduced (*using recursion*) to problems that are closer to stopping cases.
3. Eventually the problem can be reduced to base cases only, which are relatively easy to solve.

<u>In general:</u>

```
if (base case)
    solve it
else
    reduce the problem using recursion // general case
```

Factorial of 4 (Recursive)

Calculate 4!
4! = 4 * (4-1)!
4! = 4 * 3!
4! = 4 * __

Factorial of 4 (Recursive)

Calculate 4!
*4! = 4 * (4-1)!*
*4! = 4 * 3!*
*4! = 4 * __*

Calculate 3!
*3! = 3 * (3-1)!*
*3! = 3 * 2!*
*3! = 3 * __*

Factorial of 4 (Recursive)

Calculate 4!
4! = 4 * (4-1)!
4! = 4 * 3!
4! = 4 * __

Calculate 3!
3! = 3 * (3-1)!
3! = 3 * 2!
3! = 3 * __

Calculate 2!
2! = 2 * (2-1)!
2! = 2 * 1!
2! = 2 * __

Factorial of 4 (Recursive)

Calculate 4!
4! = 4 * (4-1)!
4! = 4 * 3!
4! = 4 * __

Calculate 3!
3! = 3 * (3-1)!
3! = 3 * 2!
3! = 3 * __

Calculate 2!
2! = 2 * (2-1)!
2! = 2 * 1!
2! = 2 * __

Calculate 1!
1! = 1 * (1-1)!
1! = 1 * 0!
1! = 1 * __

Factorial of 4 (Recursive)

Calculate 4!
4! = 4 * (4-1)!
4! = 4 * 3!
4! = 4 * __

Calculate 3!
3! = 3 * (3-1)!
3! = 3 * 2!
3! = 3 * __

Calculate 2!
2! = 2 * (2-1)!
2! = 2 * 1!
2! = 2 * __

Calculate 1!
1! = 1 * (1-1)!
1! = 1 * 0!
1! = 1 * __

Calculate 0!
0! = 1

Factorial of 4 (Recursive)

Calculate 4!
4! = 4 * (4-1)!
4! = 4 * 3!
4! = 4 * __

Calculate 3!
3! = 3 * (3-1)!
3! = 3 * 2!
3! = 3 * __

Calculate 2!
2! = 2 * (2-1)!
2! = 2 * 1!
2! = 2 * __

Calculate 1!
1! = 1 * (1-1)!
1! = 1 * 0!
1! = 1 * 1
1! = 1

Factorial of 4 (Recursive)

Calculate 4!
4! = 4 * (4-1)!
4! = 4 * 3!
4! = 4 * __

Calculate 3!
3! = 3 * (3-1)!
3! = 3 * 2!
3! = 3 * __

Calculate 2!
2! = 2 * (2-1)!
2! = 2 * 1!
2! = 2 * 1
2! = 2

Factorial of 4 (Recursive)

Calculate 4!
*4! = 4 * (4-1)!*
*4! = 4 * 3!*
*4! = 4 * __*

Calculate 3!
*3! = 3 * (3-1)!*
*3! = 3 * 2!*
*3! = 3 * 2*
3! = 6

Factorial of 4 (Recursive)

Calculate 4!

4! = 4 * (4-1)!

4! = 4 * 3!

4! = 4 * 6

4! = 24

Recursive Factorial Function

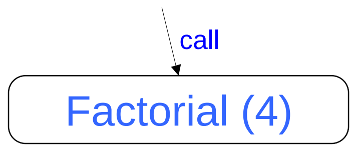
```
// Computes the factorial of a nonnegative integer.
// Precondition: n must be greater than or equal to 0.
// Postcondition: Returns the factorial of n; n is unchanged.
int Factorial(int n)
{
 if (n ==0)
      return (1);
 else
       return (n * Factorial(n-1));
}
```

Tracing Factorial(4)

```
Factorial(4)
```

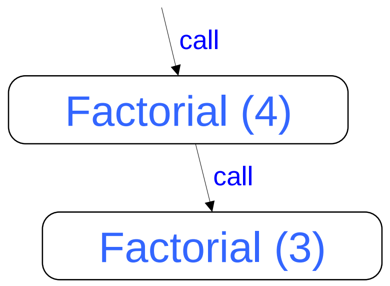
Tracing Factorial(4)

```
Factorial(4)
```



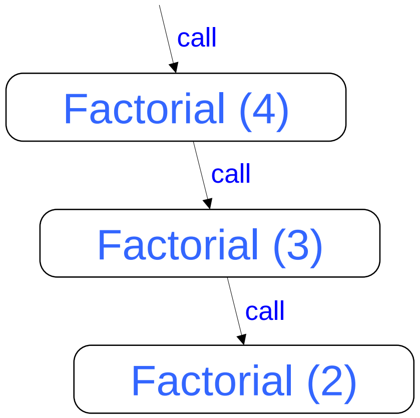
Tracing Factorial(4)

```
Factorial(4)      = 4 * Factorial(3)
```



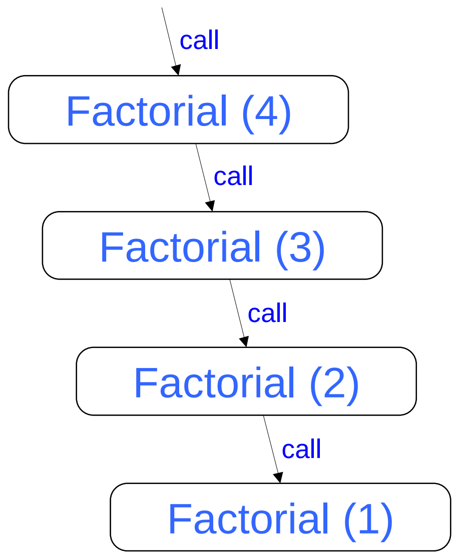
Tracing Factorial(4)

```
Factorial(4)       = 4 * Factorial(3)
             = 4 * (3 * Factorial(2))
```



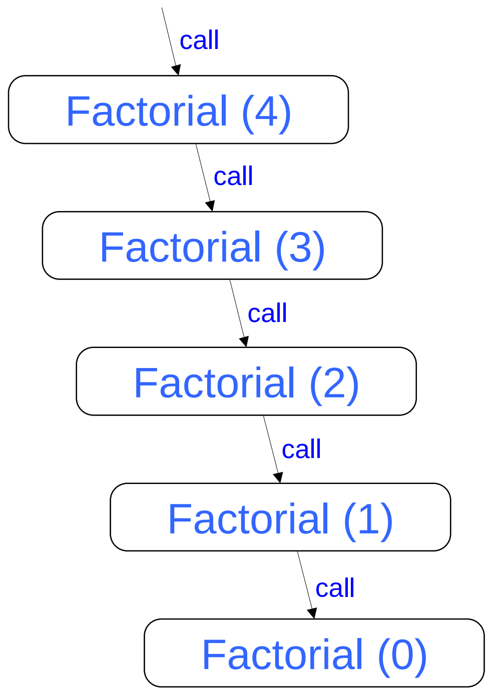
Tracing Factorial(4)

```
Factorial(4)       = 4 * Factorial(3)
             = 4 * (3 * Factorial(2))
             = 4 * (3 * (2 * Factorial(1)))
```



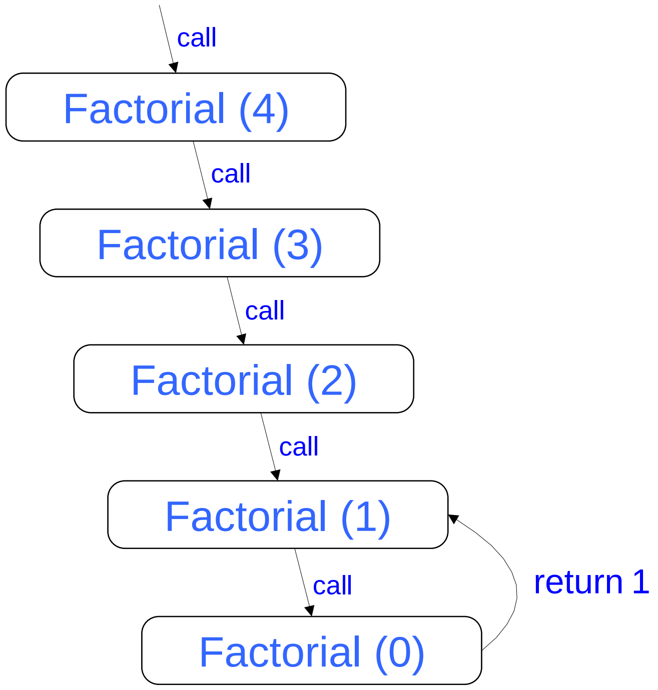
Tracing Factorial(4)

```
Factorial(4)      = 4 * Factorial(3)
            = 4 * (3 * Factorial(2))
            = 4 * (3 * (2 * Factorial(1)))
            = 4 * (3 * (2 * (1 * Factorial(0))))
```



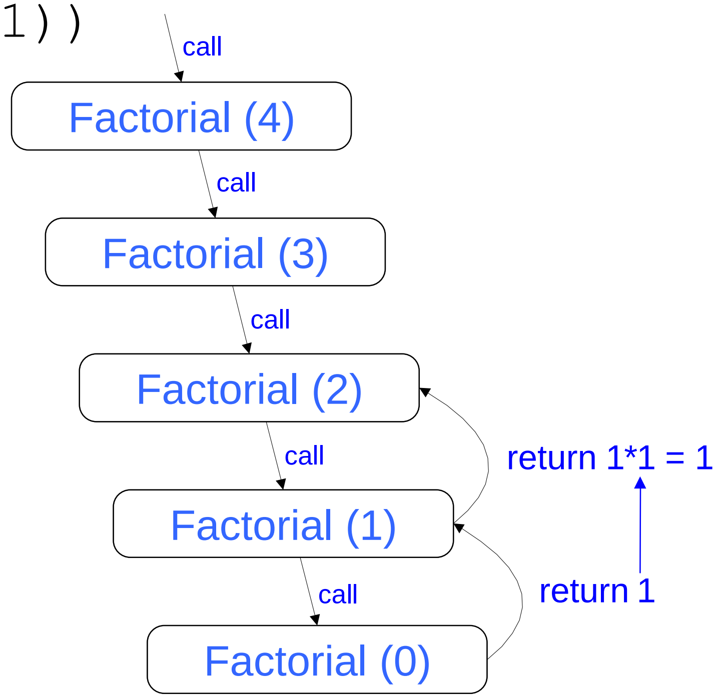
Tracing Factorial(4)

```
Factorial(4)       = 4 * Factorial(3)
             = 4 * (3 * Factorial(2))
             = 4 * (3 * (2 * Factorial(1)))
             = 4 * (3 * (2 * (1 * Factorial(0))))
             = 4 * (3 * (2 * (1 * 1)))
```



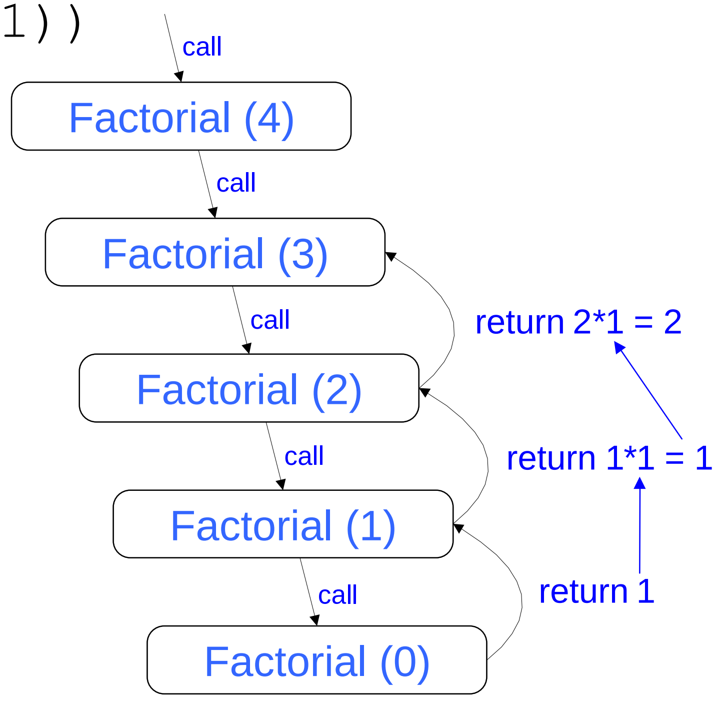
Tracing Factorial(4)

```
Factorial(4)      = 4 * Factorial(3)
            = 4 * (3 * Factorial(2))
            = 4 * (3 * (2 * Factorial(1)))
            = 4 * (3 * (2 * (1 * Factorial(0))))
            = 4 * (3 * (2 * (1 * 1)))
            = 4 * (3 * (2 * 1))
```



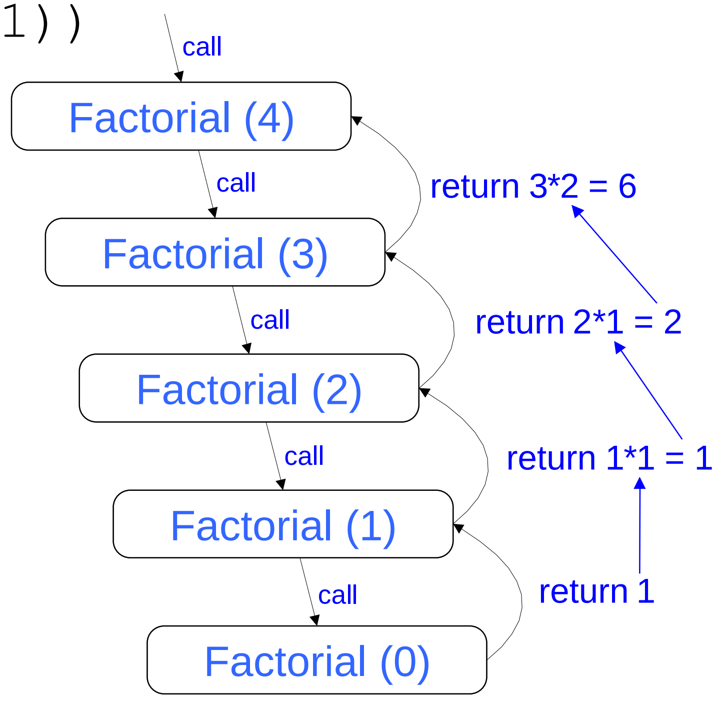
Tracing Factorial(4)

```
Factorial(4)      = 4 * Factorial(3)
             = 4 * (3 * Factorial(2))
             = 4 * (3 * (2 * Factorial(1)))
             = 4 * (3 * (2 * (1 * Factorial(0))))
             = 4 * (3 * (2 * (1 * 1)))
             = 4 * (3 * (2 * 1))
             = 4 * (3 * 2)
```



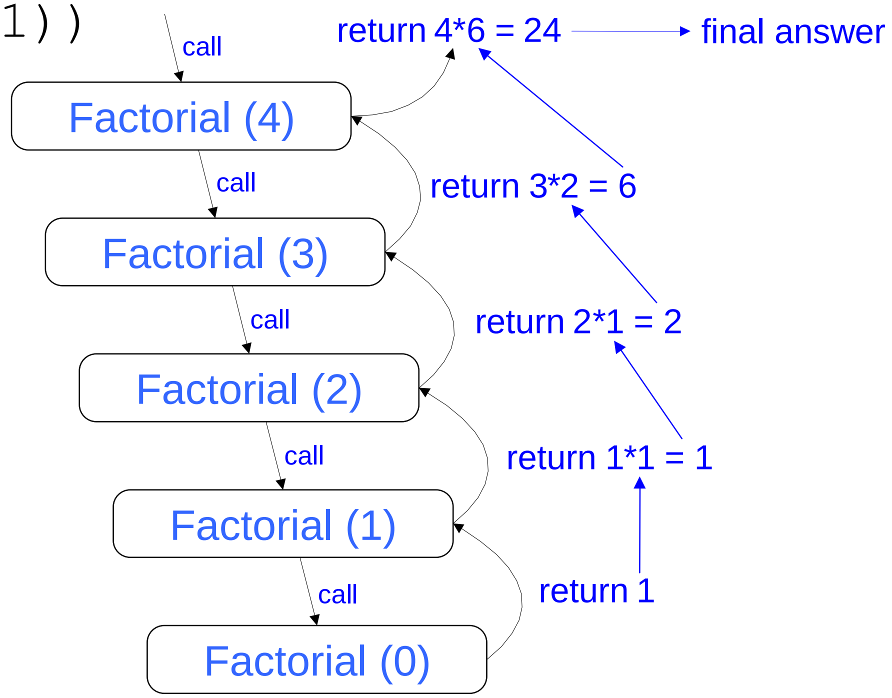
Tracing Factorial(4)

```
Factorial(4)      = 4 * Factorial(3)
            = 4 * (3 * Factorial(2))
            = 4 * (3 * (2 * Factorial(1)))
            = 4 * (3 * (2 * (1 * Factorial(0))))
            = 4 * (3 * (2 * (1 * 1)))
            = 4 * (3 * (2 * 1))
            = 4 * (3 * 2)
            = 4 * 6
```



Tracing Factorial(4)

```
Factorial(4)       = 4 * Factorial(3)
             = 4 * (3 * Factorial(2))
             = 4 * (3 * (2 * Factorial(1)))
             = 4 * (3 * (2 * (1 * Factorial(0))))
             = 4 * (3 * (2 * (1 * 1)))
             = 4 * (3 * (2 * 1))
             = 4 * (3 * 2)
             = 4 * 6
             = 24
```



A Couple of Things You Should Know

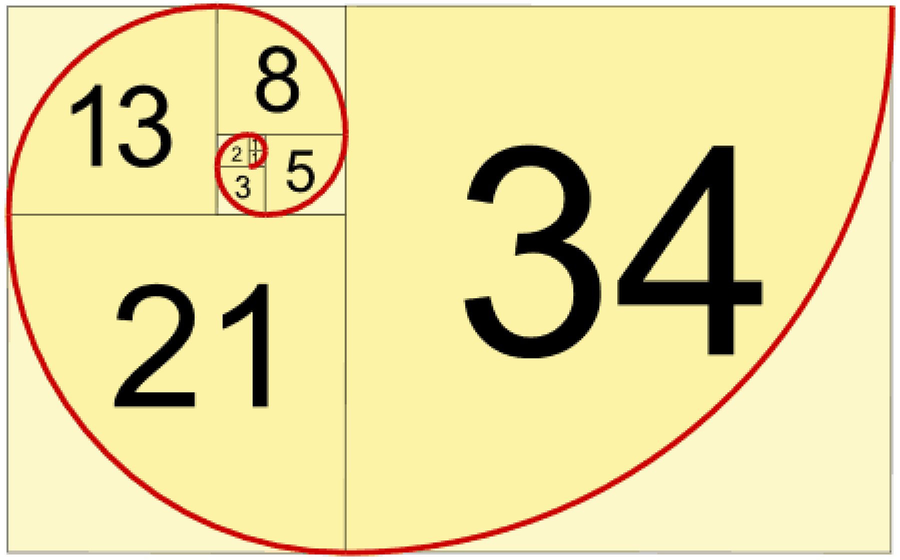
- A **stack** is used to keep track of function calls.
- Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement (this gives the computer the return point after execution of the function)

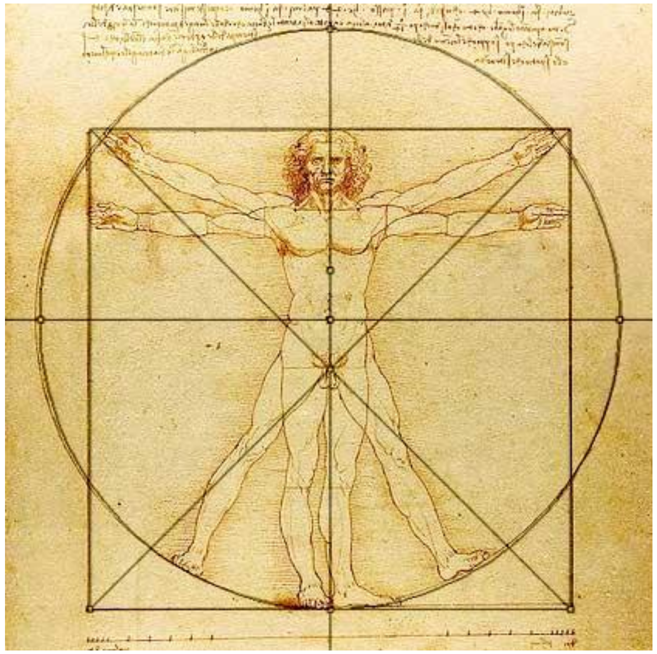
Pitfalls of Recursion

- If the recursion never reaches the base case, the recursive calls will continue until the computer runs out of memory and the program crashes. Experienced programmers try to examine the remains of a crash. The message “stack overflow error” or “heap storage exhaustion” indicates a possible runaway recursion.
- When programming recursively, you need to make sure that the algorithm is moving toward the base case. Each successive call of the algorithm must be solving a simpler version of the problem.
- Any recursive algorithm can be implemented iteratively, but sometimes only with great difficulty. However, a recursive solution will always run more slowly than an iterative one because of the overhead of opening and closing the recursive calls.

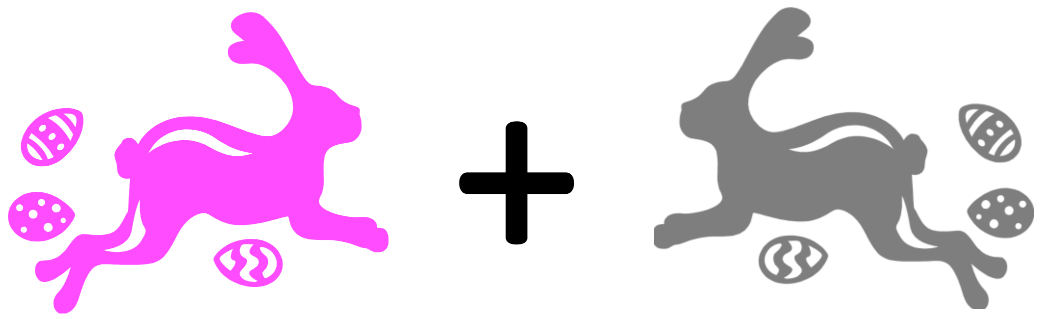
Fibonacci's Problem

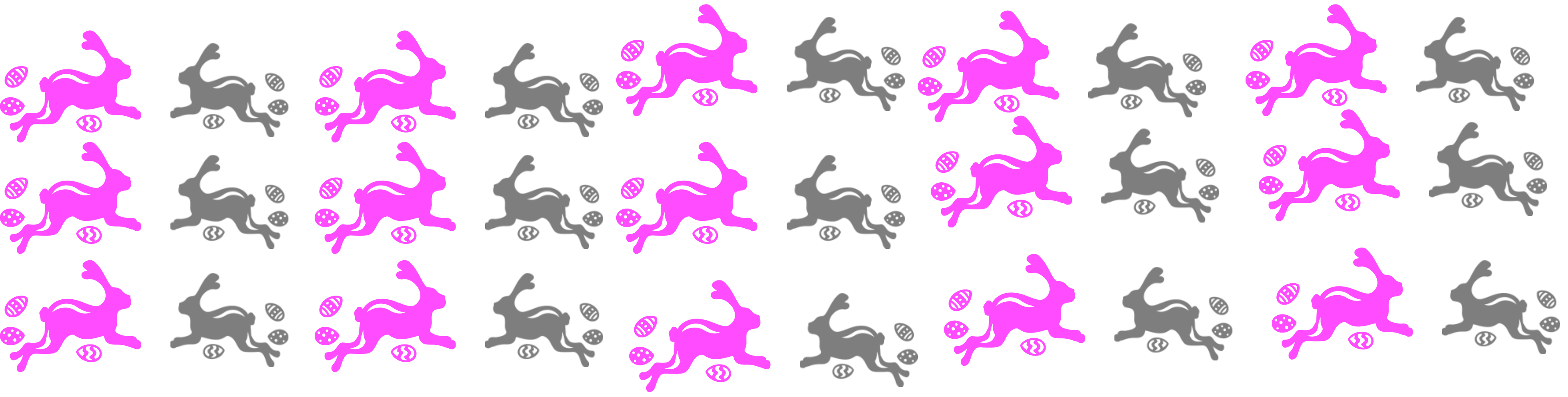
"Fibonacci"
(Leonardo de Pisa)
1170-1240





Fibonacci's Problem





Rabbit Rules

1. All pairs of rabbits consist of a male and female
2. One pair of newborn rabbits is placed in hutch on January 1
3. When this pair is 2 months old they produce a pair of baby rabbits
4. Every month afterwards they produce another pair
5. All rabbits produce pairs in the same manner
6. Rabbits don’t die

Fibonacci’s Problem

How many pairs of rabbits will there be 12 months later?

Jan 1

0

Feb 1

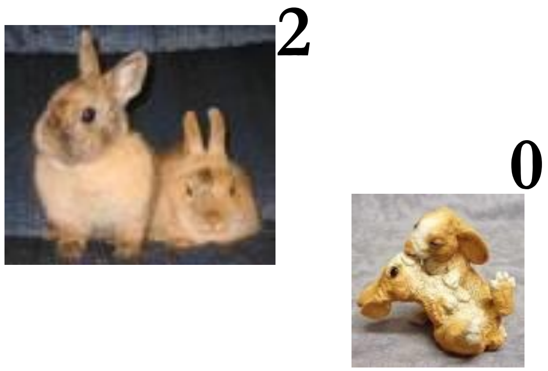
1

Mar 1

2

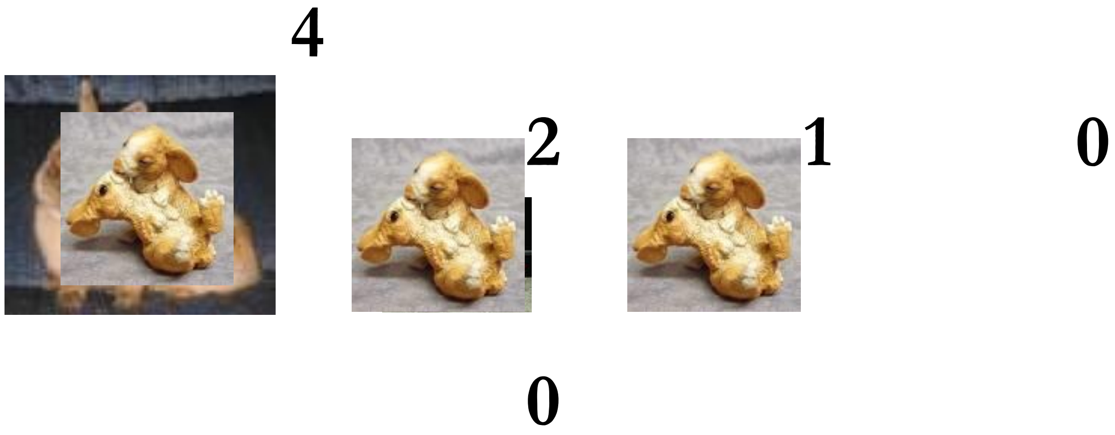
0

Mar 1



Apr 1

May 1



May 1

4

2

1

0

0

June 1

5

3

2

1

1

In General,

$$\boxed{\text{Pairs this month}} = \boxed{\text{Pairs last month}} + \boxed{\text{Pairs of newborns}}$$

$$\boxed{\text{Pairs this month}} = \boxed{\text{Pairs last month}} + \boxed{\begin{array}{l}\text{Pairs 2 months}\\ \text{ago}\end{array}}$$

Fibonacci Numbers

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

where each number is the sum of the preceding two.

- Recursive definition:
  - **`F(0) = 0;`**
  - **`F(1) = 1;`**
  - **`F(number) = F(number-1)+ F(number-2);`**

Fibonacci Numbers

```
int Fibonacci(int n)
{
        if(n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
                return Fibonacci(n-1)+Fibonacci(n-2);
}
```

Fibonacci Numbers

```
int Fibonacci(int n)
{
        if(n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
                return Fibonacci(n-1)+Fibonacci(n-2);
}
```

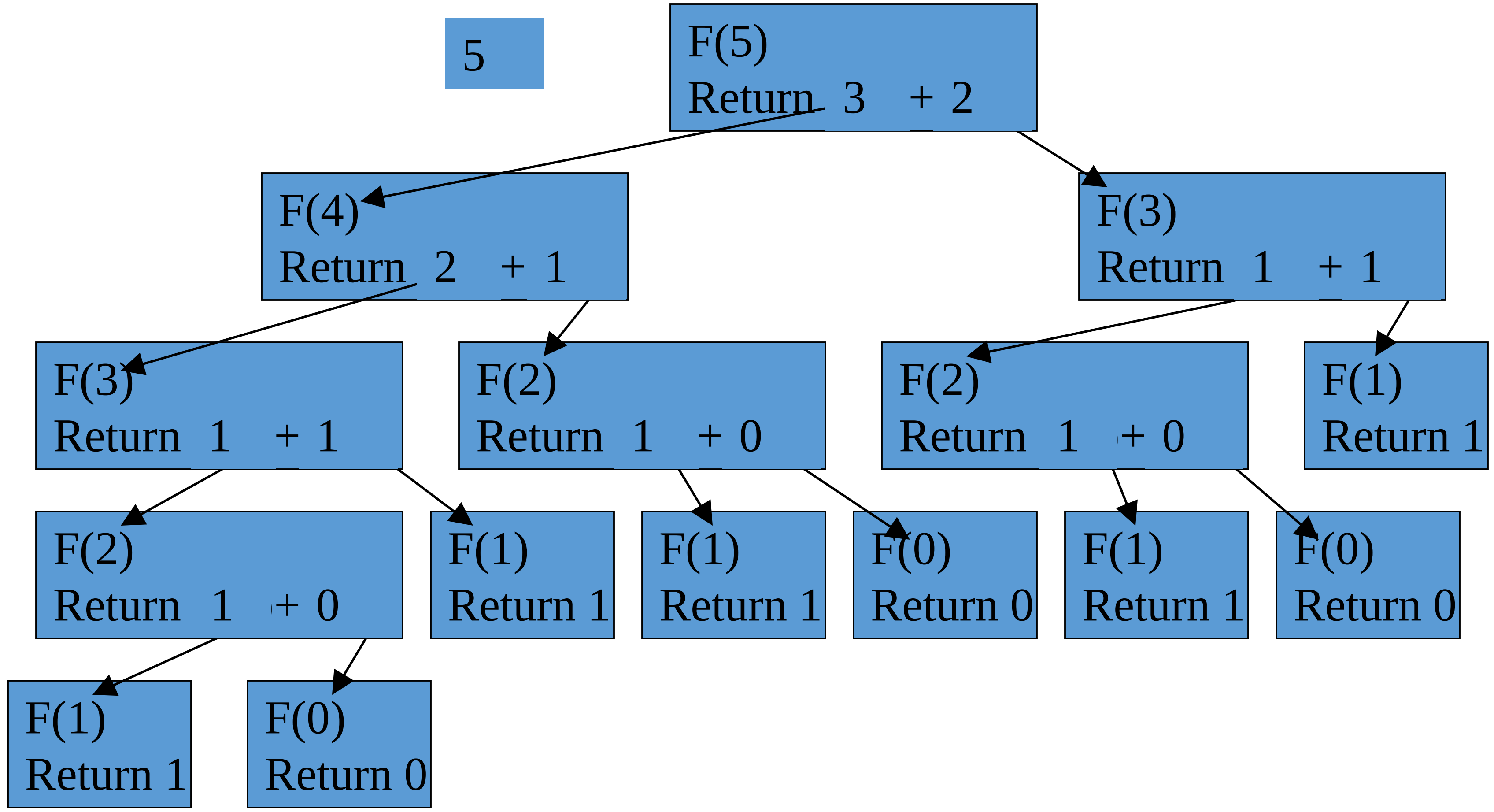
Base case

Fibonacci Numbers

```
int Fibonacci(int n)
{
	if(n == 0)
		return 0;
	else if (n == 1)
		return 1;
	else
		return Fibonacci(n-1)+Fibonacci(n-2);
}
```

General case

Tracing Fibonacci(5)



Another Example: *n* choose *r* (combinations)

- Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{r} = \frac{n!}{r! \times (n-r)!}, 1 < r < n \, (closed\ \ form)$$

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}, 1 < r < n \, (recursive)$$

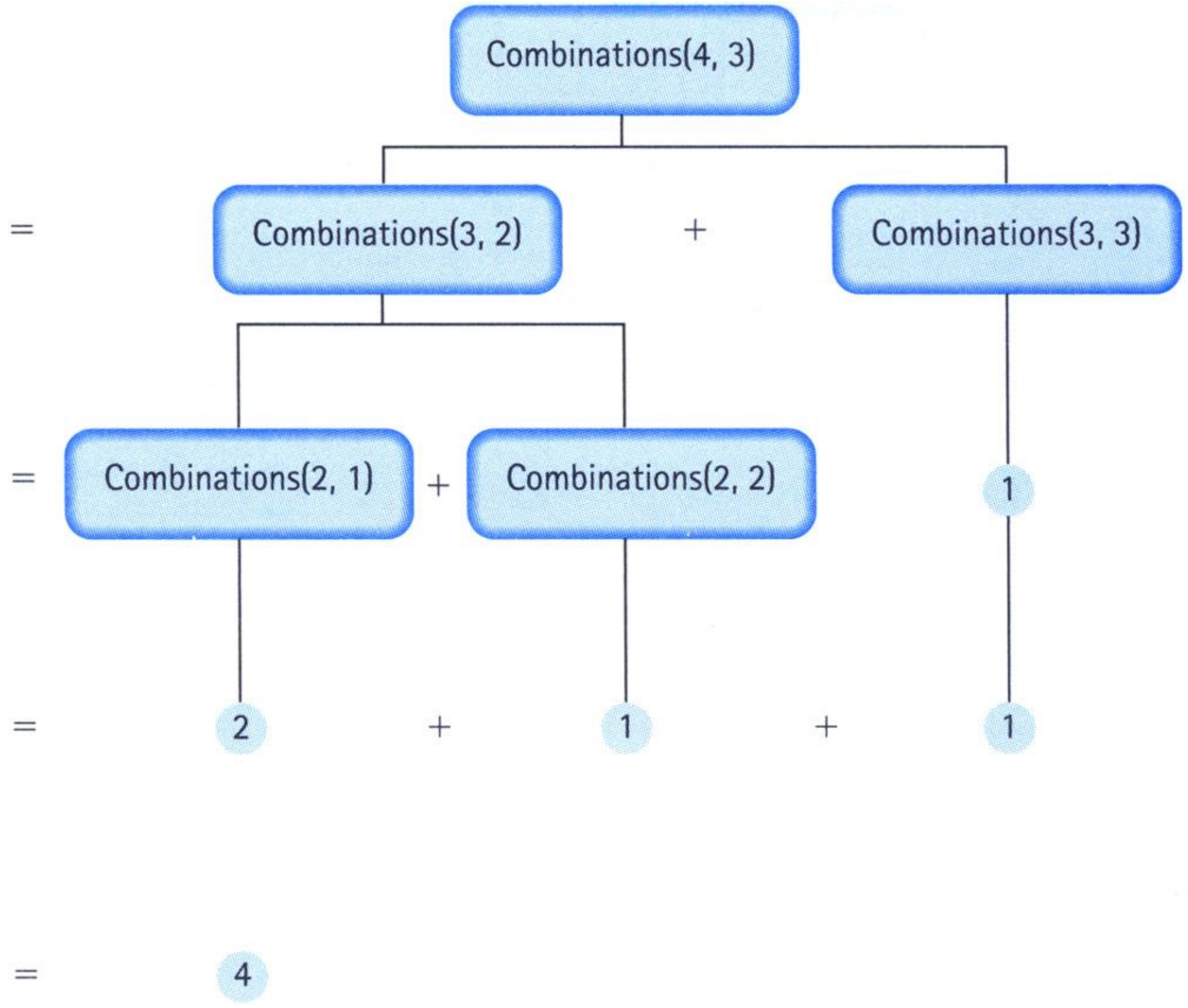
with base cases:

$$\binom{n}{1} = n \ \ and \ \binom{n}{n} = 1$$

n choose *r* (Combinations)

```
int Combinations(int n, int r)
{
  if(r == 1)  // base case 1
       return n;
  else if (n == r)  // base case 2
       return 1;
  else //general case
       return(Combinations(n-1, r-1) + Combinations(n-1, r));
}
```

Tracing Combinations(4,3)


Combinations(4, 3)
= Combinations(3, 2) + Combinations(3, 3)
= Combinations(2, 1) + Combinations(2, 2) 1
= 2 + 1 + 1
= 4

Home-works

Implement the following function prototypes. You must implement them recursively.

- `int sum(int n);`

Description: Calculates the the sum $1 + 2 + \cdots + n$

- `void listNumbersAsc(int start, int end);`

Description: Outputs the numbers from start to end in ascending order.

- `void listNumbersDesc(int start, int end);`

Description: Outputs the numbers from start to end in descending order.

- `void printBinary(int x);`

Description: Prints the binary equivalent of x.

- `int mul(int a, int b);`

Description: Returns the product of a and b. The only arithmetic operation that you are allowed to use in this problem is addition (+).

- `int power(int a, int b);`

Description: Returns a^b. The only arithmetic operation that you are allowed to use in this problem is multiplication (*).

- `double harmonicSum(int n);`

Description: Returns the sum $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$.

- `int sumOfDigits(int x);`

Description: Returns the sum of digits of the positive integer x. For example, when called with the parameter 12345, this function returns 15.