Recursion

What is a Recursion?

- A recursive function is a function that directly or indirectly calls itself.
- **Recursion** is:
  - Is a very powerful problem-solving approach, that can generate simple solutions to certain kinds of problems that would be difficult to solve in other ways.
- **Recursion** splits a problem:
  - Into one or more simpler and smaller versions of itself.

The General Approach of Recursion

if problem is "*small enough*" **do**

solve it *directly*

else

- break it into one or more *smaller subproblems*
- solve each subproblem *recursively*
- *combine* results into solution to the whole problem

Recursion Cases

- There are two main parts to recursive functions:
  - **base case:** the case for which the solution can be stated non-recursively. Here, a solid solution is found.
  - **general (recursive) case:** the case for which the solution is expressed in terms of a smaller version of itself.

- A proper recursive function must always have a *base case*. The base case is a way to **return without making a recursive call**.

Recursive Definitions: Factorial

- We indicate the factorial of n by ***n!***
- **It is the multiplication of all numbers** from n down to **1**.
- **Examples:**
  - 5! = 5*4*3*2*1
  - 7! = 7*6*5*4*3*2*1
  - 1!=1
  - 0!=1

Recursive Definitions: Factorial

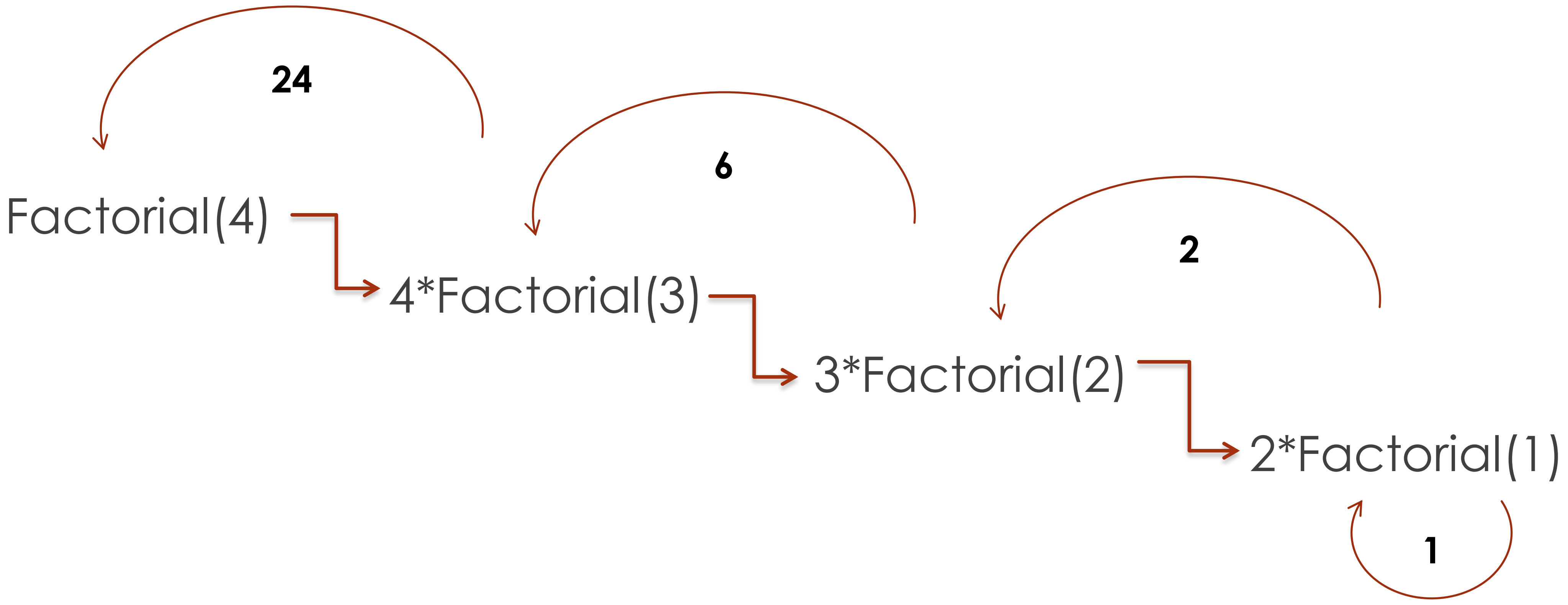
- For positive values of *n we can write **n!** as*
  - n! = n * **(n-1) * (n-2) * (n-3) ….. *3*2*1**
  - We can write **(n-1) * (n-2) * (n-3) ….. *3*2*1 as (n-1)!**
  - So, **n!= n * (n-1)!**
  - For example: 5!= 5* (4)!, (4)!=4 *(3)!, (3)!=3*(2)!, (2)!=2*1 and (1)!=

- $$n! = \begin{cases} 1 & if\ n == 1\ (base\ case) \\ n * (n-1)! & if\ n > 1\ (recursive\ case) \end{cases}$$

Recursive Definitions: Factorial Pseudocode

```
int factorial (int n) {
  if (n == 1)                                //Base case
    return 1
  else
     return n * factorial(n-1)              //Recursive case
}
```

Recursive Definitions: Factorial



Fibonacci Sequence

- The Fibonacci Sequence $f_0, f_1, f_2, f_3, \ldots$ is the series of numbers:

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

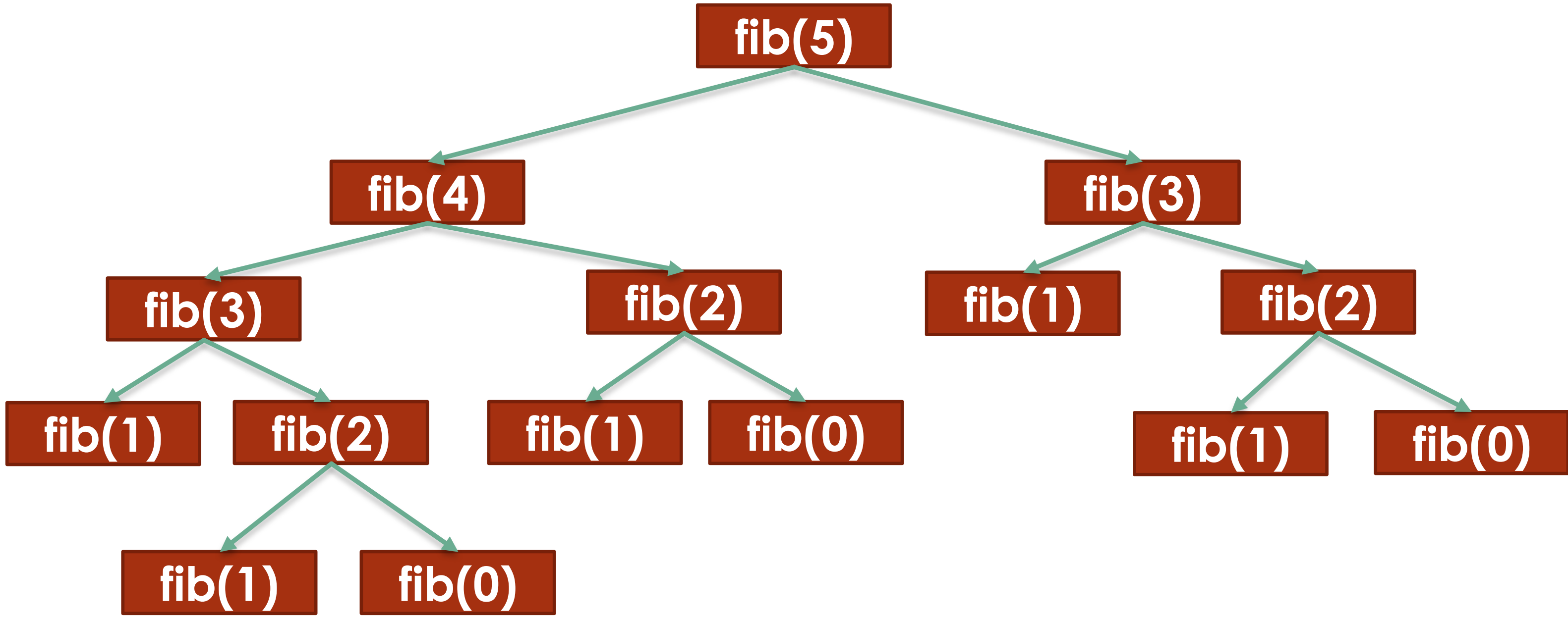
- $f_0 = 0,\ f_1 = 1, f_2 = 1, f_3 = 2, \ldots$
- The next number is found by adding up the two numbers before it.
  - The 2 is found by adding the two numbers before it (1+1)
  - The 3 is found by adding the two numbers before it (1+2),
  - And the 5 is (2+3),
  - and so on!

$$f_i = \begin{cases} 1 & if\ i == 1\ or\ i == 0\ (base\ case) \\ f_{i-1} + f_{i-2} & if\ n > 1\ \ (recursive\ case) \end{cases}$$

Fibonacci Sequence Pseudocode

```
int fib(int n)
{
  if (n <= 1)                  // if(n==1 || n==0)  Base Case
    return n;
  else
    return fib(n-1) + fib(n-2);    // Recursive Case
}
```

This is an efficient implementation for finding nth Fibonacci number.
- this implementation does a lot of repeated work.



Implement an efficient recursive function to find nth Fibonacci number?

Power

- The power of a number can be calculated as x^y where x is the number and y is its power.
- For example:
  - $2^3 = 8$
  - $9^3 = 729$
  - $5^0 = 1$

- What is the base case and recursive case of power? Justify your answer.
- Write a Pseudocode to calculate the power of a given number using recursion.

More examples will be given with tree and sorting algorithms.

Exercises

- What does the following recursive code do?

```
void recursive_function(int n)
{
     if(n == 0)
            return;
    else
      {
          recursive_function(n-1);
          cout<<n<<endl;
      }
}
int main()
{
     recursive_function(10);
     return 0;
}
```

Exercises

- 2. Which of the following problems can't be solved using recursion?
  a) Factorial of a number
  b) Nth Fibonacci number
  c) Length of a string
  d) Problems without base case
- In recursion, the condition for which the function will stop calling itself is
  a) Best case
  b) Worst case
  c) Base case
  d) There is no such condition