Recursive functions and recursion

Programming Fundamentals

Recursive functions and recursion

- A recursive function is a function that calls itself. And, this technique is known as recursion.

```
void recurse()
{
	recurse();
}
int main()
{
	recurse();
}
```

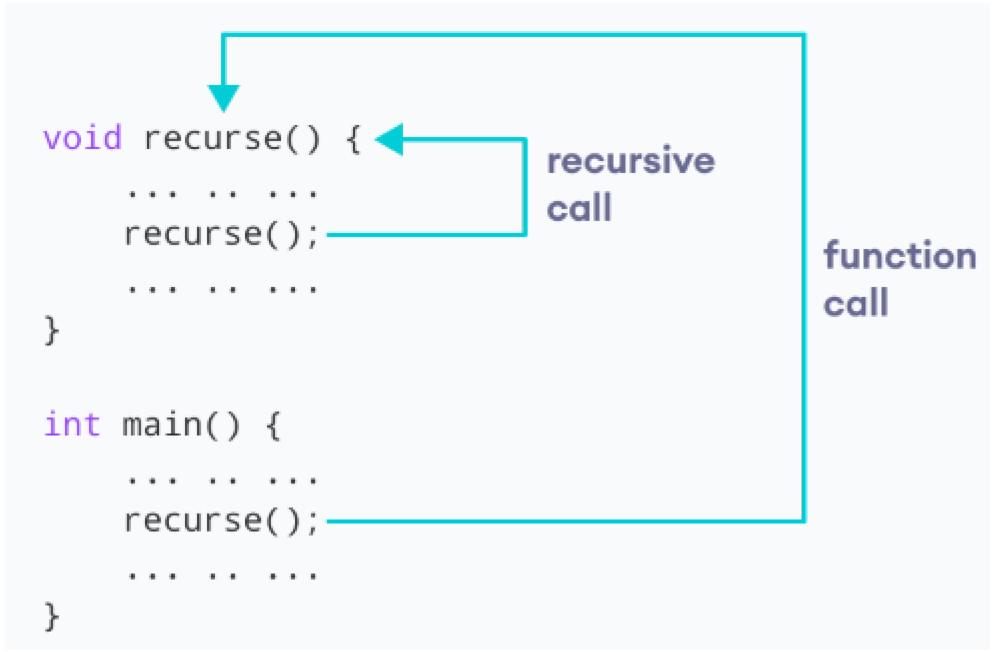
Motivation:

- Many times, a problem broken down into smaller parts is more efficient. Dividing a problem into smaller parts aids in conquering it. Hence, recursion is a divide-and-conquer approach to solving problems.
  - Sub-problems are easier to solve than the original problem
  - Solutions to sub-problems are combined to solve the original problem

WE WILL LOOK INTO THIS LATER ON..

The recursion continues until some condition is met.
To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

Infinite recursion…never stops…



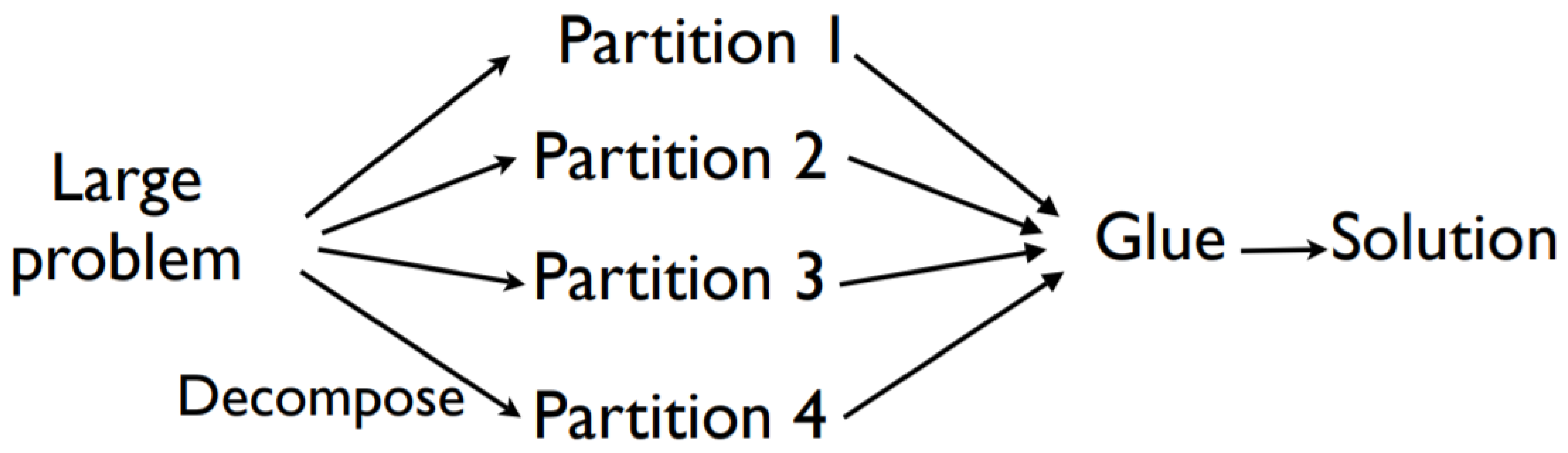
```
void recurse() {
    ... .. ...
    recurse();
    ... .. ...
}

int main() {
    ... .. ...
    recurse();
    ... .. ...
}
```

A recursive definition has two parts
1.Base **- an initial, simple definition which the solution can be stated nonrecursively**
2.Recursion **- the part that refers to itself in terms of a smaller version of itself (inductive part)**
Recursive algorithm: A solution that is expressed in terms of **(a) a base case** and **(b) smaller instances of itself**

Basic Idea of Divide and Conquer:

- If the problem is easy, solve it directly
- If the problem cannot be solved as is, decompose it into smaller parts,. Solve the smaller parts

- Often, the parts must be "glued"into a solution

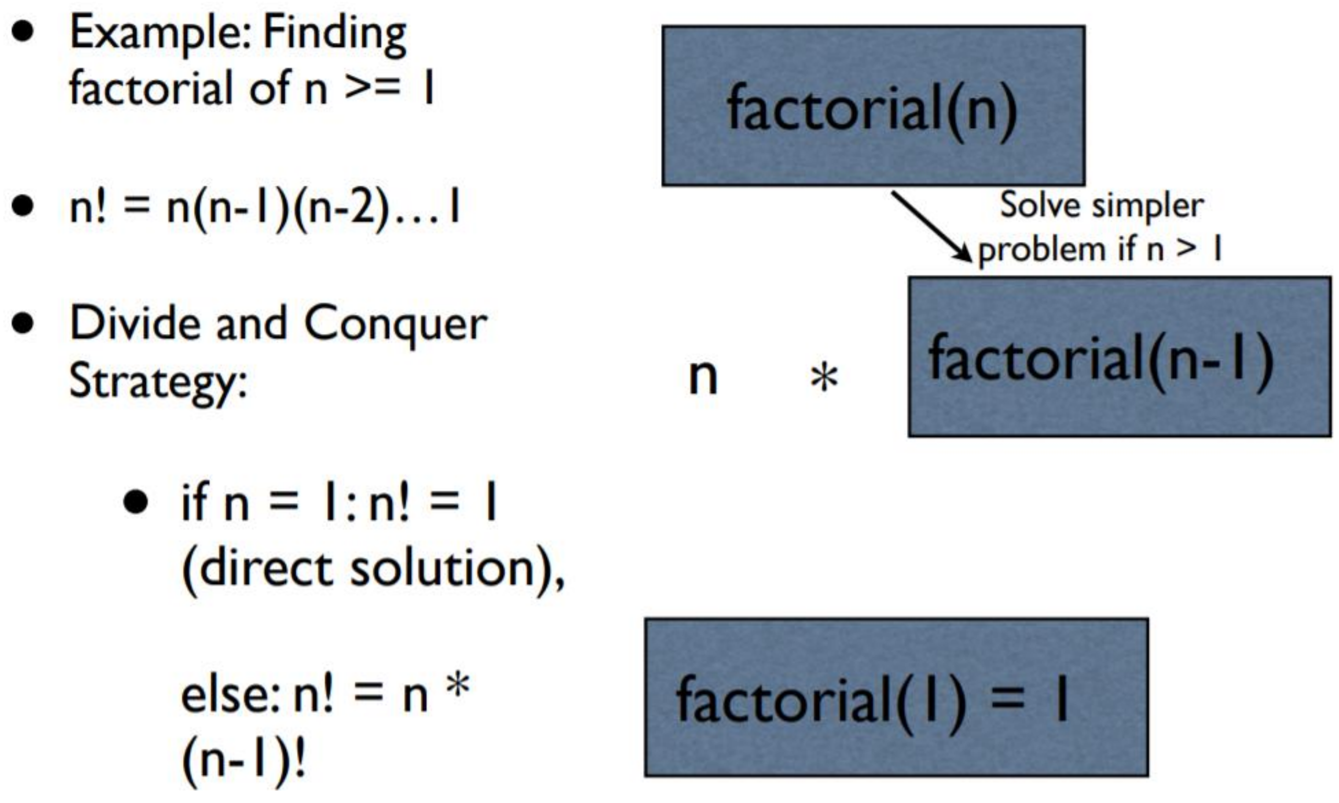


Example: finding factorial

- Finding factorial of 5:
  - ***5x4x3x2x1*** *One way of to multiply 5 numbers*
  - Sub-problems:
  - ***5x4x3x2x1***
  - ***5x4x3x2***
  - ***5x4x3x2***
  - ***5x4x6***
  - ***5x4x6***
  - ***5x24***
  - ***5x24***
  - ***120***

Each sub problem refers to multiplying 2 numbers

Gradually, we reach a final solution



- Example: Finding factorial of n >= 1
- n! = n(n-1)(n-2)…1
- Divide and Conquer Strategy:
  - if n = 1: n! = 1 (direct solution),

    else: n! = n * (n-1)!

factorial(n)
Solve simpler problem if n > 1
n * factorial(n-1)
factorial(1) = 1

n! = n * (n-1)! (divide-and-conquer)

Recursion in functions

- When a function makes use of itself, as in a divide-and-conquer strategy, it is called recursion
- Recursion requires:
  - Base case or direct solution step. (e.g., factorial(1))
- Recursive step(s):
  - A function calling itself on a smaller problem. E.g., n*factorial(n-1)
- Eventually, all recursive steps must reduce to the base case

Base case

- One critical requirement of recursive functions is termination point or base case. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

Now lets explore how to find factorial of a number using iterative approach (using simple method of loops) and recursion that we just covered..

Iterative approach

```
#include <iostream>
using namespace std;
int main()
{
        int n = 6, fact = 1, i;
        for(i=1; i<=n; i++)
             fact = fact * i;
        cout<<"Factorial of "<< n <<" is "<<fact;
        return 0;
}
```

Iterative vs recursive

- Usage of either of these techniques is a trade-off between time complexity and size of code. If time complexity is the point of focus, and number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go.
- **Recursion**: Recursion involves calling the same function again, and hence, has a very small length of code. However, as we saw in the analysis, the time complexity of recursion can get to be exponential when there are a considerable number of recursive calls. Hence, usage of recursion is advantageous in shorter code, but higher time complexity.
- **Iteration**: Iteration is repetition of a block of code. This involves a larger size of code, but the time complexity is generally lesser than it is for recursion.

PROPERTY	RECURSION	ITERATION
Definition	Function calls itself.	A set of instructions repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code Size	Smaller code size	Larger Code Size.
Time Complexity	Very high(generally exponential) time complexity.	Relatively lower time complexity(generally polynomial-logarithmic).

Recursive approach

```
#include <iostream>
using namespace std;
int fact(int n)
{
  if ((n==0)||(n==1))
            return 1;
  else
            return n*fact(n-1);
}
int main()
{
        int n = 4;
        cout<<"Factorial of "<<n<<" is "<<fact(n);
        return 0;
}
```

Base case (for termination)

Recursive calls..sub problems

We can see what is happening if we insert a debugger statement into the code and use devtools to step though it and watch the call stack.

- The execution stack places fact() with 5 as the argument passed. The base case is false, so enter the recursive condition.
- The execution stack places fact() a second time with n-1 = 4 as argument. Base case is false, enter recursive condition.
- The execution stack places fact() a third time with n-1 (4–1) = 3 as argument. Base case is false, enter recursive condition.
- The execution stack places fact() a fourth time with n-1(3–1) = 2 as argument. Base case is false, enter recursive condition.
- The execution stack places fact() a fifth time with n-1 (2–1) = 1 as argument. Now the base case is true, so return 1.
- At this point, we have decreased the argument by one on each function call until we reach a condition to return 1.

From here the last execution context completes, n== 1, so that function returns 1.

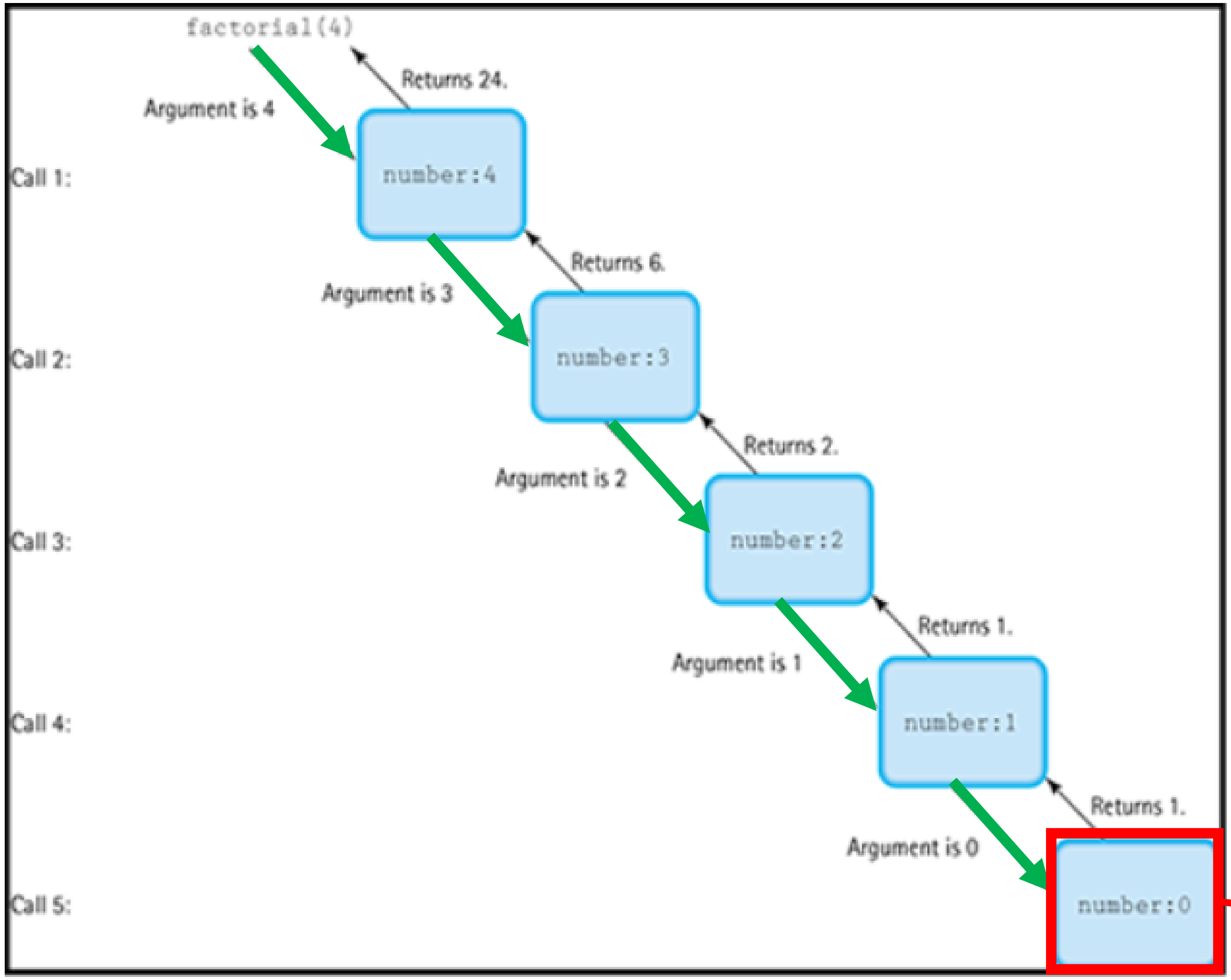
- Next n == 2, so the return value is 2. (1×2).
- Next n == 3, so the return value is 6, (2×3).

So far we have 1×2×3.

- Next, n== 4, (4×6). 24 is the return value to the next context.
- Finally, n== 5, (5×24) and we have 120 as the final value.

- Each recursive call to a function is saved as a state..
- In that state, all the variables with their relevant values are stored and tracked..
- When base case executes, the recursion stops, and backtracking to previous states happens
- Lets explore an example of finding factorial of 4
  - 4x3x2x1=24

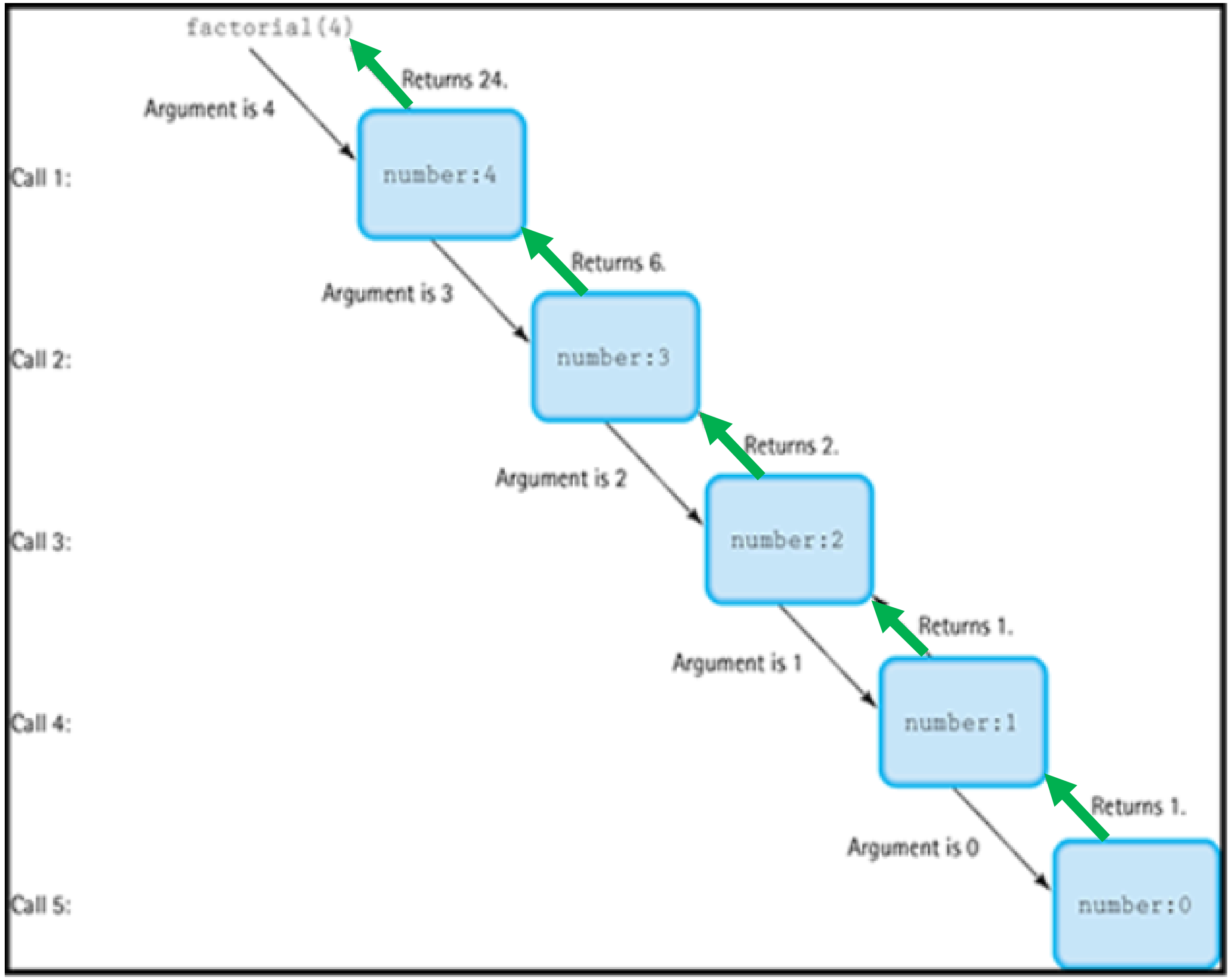
- First, recursive calls to a function are made, without returning..
- And this continues, till base case is reached…



See green arrows

Base case (for termination)

- Next, once base case is met, recursion stops, and backtracking happens..
- That is, starting from the last state, the control moves back to each older state
- Second last state, then third last state..so on till first state is reached.



See green arrows