

Recursion

Fundamentals, strategy and examples in C++

Professors de Pro1

Department of Computer Science

Contents

- 1 What is Recursion?
- 2 Advantages and Disadvantages
- 3 Guide to Recursive Programming
- 4 Another Example: Reversed Output
- 5 Proposed Exercises

What is Recursion?

Definition

An abstract tool for concisely expressing a sequence of elements (instructions, computations, or tasks).

Frequently used in **mathematical definitions** and in **algorithms** that are described in terms of themselves.

Core idea

A recursive function calls itself with “smaller” parameters until it reaches a **base case** that is solved directly.

Motivating example: power of 2

Recursive definition

A natural integer n is a power of 2 when:

- 1 $n = 1$, **or**
- 2 n is even **and** $n/2$ is a power of 2.

Motivating example: power of 2

Recursive definition

A natural integer n is a power of 2 when:

- 1 $n = 1$, **or**
- 2 n is even **and** $n/2$ is a power of 2.

Is 12 a power of 2?

$$12 \xrightarrow{\div 2} 6 \xrightarrow{\div 2} 3 \text{ (odd)} \Rightarrow \mathbf{NO}$$

Is 8 a power of 2?

$$8 \xrightarrow{\div 2} 4 \xrightarrow{\div 2} 2 \xrightarrow{\div 2} 1 \Rightarrow \mathbf{YES}$$

Recursive code: is_pow_2

```
1 //pre:  n >= 1
2 //post: returns true if n is a power of 2, false otherwise
3 bool is_pow_2(int n) {
4     if (n == 1) return true;
5     else if (n%2 == 1) return false;
6     else
7         return is_pow_2(n/2);  // n is even
8 }
```

Recursive code: is_pow_2

```
1 //pre:  n >= 1
2 //post: returns true if n is a power of 2, false otherwise
3 bool is_pow_2(int n) {
4     if (n == 1) return true;
5     else if (n%2 == 1) return false;
6     else
7         return is_pow_2(n/2);  // n is even
8 }
```

Preferred iterative version

```
1 bool is_pow_2(int n) {
2     while (n%2 == 0) n = n/2;
3     return n == 1;
4 }
```

Advantages and disadvantages of recursion

Advantages

- Solutions are often **simpler and shorter** than their iterative equivalents.
- Directly mirrors inductive mathematical definitions.
- Facilitates **correctness proofs**.

Disadvantages

- Tend to be **less efficient** (call stack overhead).
- Can cause stack overflow if the recursion depth is very large.

Practical conclusion

When a simple and clear iterative implementation exists, **prefer the iterative version**. Use recursion when it significantly simplifies the solution.

Guide to recursive programming: overview

We propose a strategy in several **stages**. The rules are general and admit exceptions.

Stages

Stage 0 Skeleton: conditional structure

Stage 1 Identify the **direct cases**

Stage 2 Complete the code for the direct cases

Stage 3 Complete the **recursive cases**

3.a Propose the recursive call

3.b **Magic point**: assume it works

3.c Additional code after the call

We illustrate the stages with the joint example.

Guided example: joint(m, n)

Specification

```
//pre:  m >= 0, n >= 0
//post:  returns the number whose leading digits are those of m
//       and whose trailing digits are those of n
int joint(int m, int n)
```

Examples

joint(351, 23) → 35123

joint(34, 987) → 34987

Stage 0: skeleton

The skeleton of every recursive function is a conditional

```
1 int joint(int m, int n) {  
2     if      (...) ...    // direct case 1  
3     else if (...) ...    // direct case 2  
4     // ...  
5     else      ...    // recursive case  
6 }
```

- **Direct cases:** branches *without* a recursive call.
- **Recursive cases:** branches *with* at least one recursive call.

Stage 1: identify direct cases

Key question

For which parameter values is it **easy** to compute the function?

For joint: the simplest case is when n has **a single digit** ($n < 10$).

$$\text{joint}(1234, 8) = 1234 \times 10 + 8 = 12348$$

Stage 1: identify direct cases

Key question

For which parameter values is it **easy** to compute the function?

For joint: the simplest case is when n has **a single digit** ($n < 10$).

$$\text{joint}(1234, 8) = 1234 \times 10 + 8 = 12348$$

Proposed skeleton:

```
1 if (n < 10) ... // direct case
2 else ... // recursive case
```

Stage 2: complete the direct cases

The direct cases must be easy to code (by definition):

```
1 if (n < 10) return 10*m + n;  
2 else      ...
```

Verification

$\text{joint}(89, 7) = 10 \times 89 + 7 = 897 \checkmark$

Stage 3.a: propose the recursive call

Rule

The parameters of the recursive call must **approach the direct case**.

Correct candidate:

`joint(m, n/10)`

Reduces the number of digits of `n`. ✓

Incorrect candidate:

`joint(m, 10*n)`

Moves away from the direct case. ✗

Stage 3.b: the magic point

Magic point

We assume the recursive call works correctly. We document its result in a comment. (Equivalent to the “induction hypothesis” in mathematics.)

```
1 if (n < 10) return 10*m + n;  
2 else {  
3     int z = joint(m, n/10);  
4     /**Magic point**: leading digits of z = digits of m  
5     /**trailing digits of z = digits of n/10  
6     ...  
7 }
```

Stage 3.b: the magic point

Magic point

We assume the recursive call works correctly. We document its result in a comment. (Equivalent to the “induction hypothesis” in mathematics.)

```
1 if (n < 10) return 10*m + n;  
2 else {  
3     int z = joint(m, n/10);  
4     /**Magic point**: leading digits of z = digits of m  
5     /**trailing digits of z = digits of n/10  
6     ...  
7 }
```

Mental trace

joint(23, 148): we call joint(23, 14)

Magic point: $z = 2314$

Target result: $23148 = z \times 10 + (148 \bmod 10)$

Stage 3.c: additional code

What is missing?

We need code that, starting from the magic point result, produces the final result.

$$z = \underbrace{2314}_{\text{"joint(m, n/10)"}} , \quad n \bmod 10 = 8, \quad \text{result} = 23148 = z \times 10 + (n \bmod 10)$$

```
1 return 10*z + n%10;
```

Final code: joint

```
1 //pre:  m >= 0, n >= 0
2 //post: returns the number whose leading digits are those of m
3 //      and whose trailing digits are those of n
4 int joint(int m, int n) {
5     if (n < 10)
6         return 10*m + n;
7     else {
8         int z = joint(m, n/10);
9         // leading digits of z are those of m,
10        // trailing digits of z are those of n/10
11        return 10*z + n%10;
12    }
13 }
```

Verification

`joint(351, 23):` $z = \text{joint}(351, 2) = 3512$, $\text{result} = 3512 \times 10 + 3 = 35123$ ✓

Example: write a list of integers in reverse

Specification

```
//pre:  list of integers in the input channel  
//post:  list written in reverse order to the output  
void write_backwards()
```

Example

Input: 14 3 238 0 12 9

Output: 9 12 0 238 3 14

Note

The reasoning is more subtle: the implicit parameter is the “list of remaining numbers in the input”.

Stages 1 and 2: direct case of write_backwards

Direct case

Empty input list: nothing to do.

```
1 int n;  
2 if (not (cin >> n)); // direct case: empty list  
3 else ...             // recursive case
```

Stages 3.a and 3.b: recursive call and magic point

3.a We already read the first element (stored in `n`). The remaining list is shorter, moving us toward the direct case.

```
1 write_backwards(); // list = input minus the first element already read
```

Stages 3.a and 3.b: recursive call and magic point

3.a We already read the first element (stored in `n`). The remaining list is shorter, moving us toward the direct case.

```
1 write_backwards(); // list = input minus the first element already read
```

3.b Magic point: we assume it works.

```
1 write_backwards();  
2 /**Magic point**: writes the following numbers in reverse
```

Mental trace

Original input: 1 2 3 4 5 6 7

We read `n = 1`; remaining input: 2 3 4 5 6 7

Magic point: `write_backwards()` prints 7 6 5 4 3 2

We only need to print 1 at the end.

Final code: write_backwards

3.c After the call, we just print the first element:

```
1 //pre:  list of integers in the input channel
2 //post: list written in reverse order to the output
3 void write_backwards() {
4     int n;
5     if (cin >> n) {
6         write_backwards();
7         // writes in reverse the numbers following the first one
8         cout << n << endl;
9     }
10 }
```

Equivalence with the version with an empty branch

The version with `if (not (cin >> n)); else {...}` is semantically identical; we drop the empty branch for clarity.

Proposed exercises

Write **recursive** functions for:

- 1 `remove_zeros(z)`: removes the zeros from the number `z` in base 10.
`remove_zeros(50103) -> 513`
- 2 `number_fusion(a, b)`: interleaves the digits of `a` and `b`.
`number_fusion(472, 15) -> 407125`
- 3 `is_two_three_balanced(z)`: checks whether the number of exact divisions by 2 equals the number of exact divisions by 3.
`is_two_three_balanced(36) -> true`
- 4 `write_num_back(z)`: prints `z` with its digits reversed.
`write_num_back(102) -> "201"`
- 5 `is_subchain(a, b)`: checks whether all digits of `a` appear in `b` in the same order.
`is_subchain(122, 213202) -> true`

Summary of the recursive strategy

The 4 stages on one slide

Stage 0 Write the skeleton: `if/else` structure.

Stage 1 Ask: when is it **trivial** to compute the function? Those are the direct cases.

Stage 2 Code the direct cases (they must be simple).

Stage 3 For each recursive case:

- (a) Propose a call that **approaches the direct case**.
- (b) **Magic point:** assume it works and document it.
- (c) Complete the code to obtain the final result.

If you cannot complete 3.c, go back to 3.a and choose another call!

Questions?

Recursion: Fundamentals, strategy and examples in C++