(/cs/)

(/cs/) Start Here (https://www.baeldung.com/cs/start-here) About ▾

(https://www.baeldung.com/cs/)

Recursion and Looping

Last updated: August 1, 2022

Written by: Nikhil Bhargav (https://www.baeldung.com/cs/author/nikhilbhargav)

Core Concepts (https://www.baeldung.com/cs/category/core-concepts)

Recursion (https://www.baeldung.com/cs/tag/recursion)

1. Introduction

In this tutorial, we'll learn about recursion and looping. Recursion (/cs/convert-recursion-to-iteration) and looping (/cs/looping-spiral) are both programming constructs that repeatedly execute a set of instructions. But they differ in the way they carry out this repetition.

In simple terms, we can define **looping or iteration as the process where the same set of instructions is repeated multiple times in a single call.** In contrast, we can enumerate **recursion as the process where the output of one iteration from a function call becomes the input of the next in a separate function call**.

2. Looping

We can define looping or iteration as invoking the same set of instructions more than once until the condition in the iteration statement becomes false. **Looping is always applied at the instruction (/cs/instructions-programs) or statement level**.

2.1. Looping Structure

Let's see the structure of a generic iteration or looping block. Each looping has three blocks:

- *the initialization block* that sets the control variable
- *the comparison block* that evaluates the control variable against a predefined condition
- *the update block* that updates the control variable

2.2. Looping Flow

Now, let's read about the flow for the looping construct:

1. Firstly, we initialize the control variable. We do it only once.
2. Secondly, we evaluate the control variable against the looping condition in most iterative constructs.
    2.1 If the condition holds, we execute the set of statements inside the looping block.
    2.2 If it is false, we exit the loop.
3. Finally, we update the control variable and repeat step 2.

2.3. Looping Internals

We mostly implement looping in our source code with constructs such as a *for* loop (/cs/loop-invariant), *while* loop, and *do-while* loop. Looping doesn't involve any functional call and is executed in the context of the current

running method or function.

2.5. Looping Example

Let's understand Looping by finding the factorial (/java-calculate-factorial) of a number. The factorial $n!$ of a number n is the product of all its factors. For example, the factorial of 5 is 120, which is the product of all its factors, i.e., 5, 4, 3, 2, 1.

With this background, let's represent the factorial function *fact* as follows:

$$fact(n) = \begin{cases} 1 & \text{if } n == 1 \\ n * (n-1)! & \text{if } n > 1 \end{cases} \quad (1)$$

The following python code uses a *for* loop in function *fact* to calculate the factorial of 5:

```
def fact(num):
    factorial=1
    for i in range(1, num+1):
        factorial=factorial*i
    return factorial

if __name__=='__main__':
    num=5
    print(f"Factorial of {num} is {fact(num)}")
```

Let's analyze this code script. It calls the function *fact()* from the function *main()* with the number *num=5* whose factorial we want. The function *fact()* runs a loop 5 times and finally returns the factorial of 5 to the function *main()*. Further, when we run this code for 100000 iterations, then we find its execution time as 3.86 microseconds. Please note that this time would vary from machine to machine.

3. Recursion

In this section, we'll go through the concept of recursion.

3.1. Functional Call Stack

Most programming languages invoke functions or methods using a special stack called the functional call stack (/cs/call-stack). Before we move further, let's introduce the data structure stack. Stack is a data structure that is used to store data in such a way that data is entered and taken out from a single end. Thus, it is called a *LIFO (Last In First Out)* since the latest data item is extracted first.

When we run any program, the system internally creates the functional call stack. It stores the local variables and parameters during function execution. As and when we invoke a function, say *childFunc()* from the currently running function, say *parentFunc(),* then the context from the last statement in *parentFunc()* (just before calling *childFunc()*) would be stored in the call stack. Then the arguments to the function *childFunc()* would be stored on the call stack, and the control moves to *childFunc()*. Once the *childfunc()* gets over, its context is popped off, and control goes to the last stored context of *parentFunc()*.

This stacking (/cs/stack-constant-time), unstacking operation is very costly in both time and memory. In case of recursion, every call to itself is pushed to the call stack till we reach the base condition. So, we find the recursive implementation slower and heavier as compared to a similar implementation using looping. On the other hand, an iterative function doesn't have the overhead of repeated function calls. This means that it has no stacking and unstacking operation overhead. Thus, its execution is faster than the recursive function.

But, an iterative function has a larger code footprint, and we find it difficult to read and understand. Moreover, the absence of the control condition or incorrectly written condition in an iteration statement may result in an infinite loop.

3.2. Recursion Internals

We define the recursive or the circular approach as an implementation that calls itself at each step until it hits the base or the termination condition. **The repeated calling is called the inductive step, whereas the terminating condition is called the base step**.

The recursion is always applied at the function level. At each inductive step, the recursive function not only gets a new stack frame but also creates new copies of all local variables and parameters. These are stored on the new stack frame at the top of the function call stack. **Thus, the recursive function reduces the size of the code and makes it easy to read, understand and maintain.**

We terminate the sequence of inductive steps by giving a well-defined base step. We should make sure that the base step doesn't call itself. Its absence in the definition of a recursive function will result in infinite recursion.

3.3. Recursion Example

Let's understand recursion by solving the same problem from a previous section. The following python code uses recursive function *fact()* to calculate the factorial of 5:

```
def fact(num):
    if num==1:
        return num
    else:
        return num * fact(num-1)

if __name__=='__main__':
    num=5
    print(f"Factorial of {num} is {fact(num)}")
```

Let's analyze this Python (/cs/languages-learn-data-structures) code script. It calls the function *fact()* from the function *main()* with the argument as the number *num=5* whose factorial we want. The function *fact()* defines two things. One, it defines a base case where it returns 1 if *num* is equal to 1. Second, it defines the inductive step where it calls itself with the argument as *num=num-1*. This implementation involves stacking and unstacking operation 5 times (since originally, *num* was equal to 5).

Furthermore, when we run this code for 100,000 iterations, we find its execution time as 3.94 microseconds. We

see the recursive approach a tad slower than the iterative approach in this case. Please note that this time would vary from machine to machine.

4. Comparison Between Recursion and Looping

In this section, we'll compare both these approaches against different parameters.

4.1. Internal Semantics

Although we now know that both recursion and looping are used to repeat a set of instructions (/cs/instructions-programs), they both achieve this differently. Recursion works at the method or the function level, whereas looping is applied at the instruction level.

We use iteration by repeatedly executing a set of instructions until the terminating condition is hit. **If the terminating condition doesn't get hit, then it'll lead to an infinitive loop. An infinite loop will waste precious CPU cycles.**

On the other hand, we write a recursive function by defining a base step and an inductive step. The base step evaluates a terminating condition and returns without calling itself, whereas the inductive step calls itself with different argument values. **If we never hit the base step's condition, then it will make the recursive function infinite. Infinite recursion can easily crash the system by consuming all the available memory.**

4.2. Speed

We can say that if we keep all other factors the same, **recursion is slower than looping**. This is so because recursion adds function calling overhead to its execution. And that makes it slower than iteration.

4.3. Memory

Recursion uses a function call stack to store the set of new local variables and parameters on each function call. On the other side of the spectrum, we have looping that doesn't use any stack. Hence, recursion incurs more memory (/cs/memory-allocation) cost since they add the overhead of stacking/unstacking operation. Moreover, we find that recursion often leads to stack overflow problems for larger input.

We prefer Looping over recursion for the most potent reason that iterative programming doesn't use a stack to store the context.

4.4. Code Footprint

Iterative code generally will have more lines of code than the corresponding recursive implementation. Moreover, looping is hard to read and maintain in the long run compared to recursion. We also have to write more comments for an iterative code (/cs/code-coverage) to explain the terminating condition and the purpose of the loop.

5. How to Decide Between Looping and Recursion

Let's now focus on how we can make an informed and correct choice between looping and recursion for different problems. We broadly use the following parameters to make this choice:

5.1. Design Complexity

We should consider it as the most important criteria for making this choice. If the original problem is complex and repetitive, then we use the divide and conquer (/cs/divide-and-conquer-strategy) approach and then solve it. So, in this case, we prefer to use recursion than looping. For example, let's consider the problem of traversing a binary

tree in order.

Here, we first traverse the left sub-tree. After that, we traverse the root of the tree, and then we traverse the right sub-tree. This is a repetitive problem on a hierarchical data structure (/cs/advanced-data-structures), so recursion is more suited than iteration.

5.2. Choice of Coding Language

We should take the coding language as another important parameter. Generally speaking, **we can say that Java (/java-single-file-source-code), C (/cs/languages-learn-data-structures), and Python (/cs/max-int-java-c-python), have recursion more costly than looping**. This is because these languages take recursion as a new functional call and implement stacking/unstacking operations.

Many C compilers (including GCC) have a flag-based implementation that treats some designated types of recursion (tail recursion) as inexpensive jumps instead of function calls. This way, they save on memory and time.

As a generic rule of thumb, we can say that recursion will be faster for a functional language, whereas iteration will be faster for an imperative language.

5.3. Space and Time Constraints

We have space and time constraints on a certain type of embedded and small form factor system. On such systems, we write optimized code that has near real-time response and minimal memory usage. So, for such systems, we prefer to use an iterative approach over a recursive approach.

5.4. Code Readability and Maintainability

We can say that **the iterative code is less readable and harder to understand and follow than the corresponding recursive code**. Moreover, in general, we can say that it is more difficult to maintain the iterative code than the recursive code since the code footprint of a recursive function is smaller than the iterative function.

In addition, every recursive implementation has a base case and an inductive case which makes it highly intuitive to grasp.

6. Conclusion

In this article, we have studied recursion and looping in computer programming. We started by defining each of these terms and then proceeded to explain them using some coding examples. After that, we thoroughly compared the two in terms of semantics, speed, memory, and code footprint. Further, we suggested some ideal situations to use each of them.

We can conclude that both recursion and looping perform a repeated set of operations but in a different manner. Both of them offer distinct advantages and disadvantages. We find a recursive function easy to write and understand but performs poorly compared to iteration. On the other hand, we find iteration hard to write and understand but scores high on performance than recursion for most computing problems.

We should choose either recursion or looping based on the underlying problem statement, available computing resources, and the binding output criteria.

Comments are closed on this article!

CATEGORIES

ALGORITHMS (/CS/CATEGORY/ALGORITHMS)
ARTIFICIAL INTELLIGENCE (/CS/CATEGORY/AI)
CORE CONCEPTS (/CS/CATEGORY/CORE-CONCEPTS)
DATA STRUCTURES (/CS/CATEGORY/DATA-STRUCTURES)
GRAPH THEORY (/CS/CATEGORY/GRAPH-THEORY)
LATEX (/CS/CATEGORY/LATEX)

NETWORKING (/CS/CATEGORY/NETWORKING)
SECURITY (/CS/CATEGORY/SECURITY)

SERIES

ABOUT

ABOUT BAELDUNG (HTTPS://WWW.BAELDUNG.COM/ABOUT)
THE FULL ARCHIVE (/CS/FULL_ARCHIVE)
EDITORS (HTTPS://WWW.BAELDUNG.COM/EDITORS)

TERMS OF SERVICE (HTTPS://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTPS://WWW.BAELDUNG.COM/PRIVACY-POLICY)
COMPANY INFO (HTTPS://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)