Summer Bootcamp 2021

Introduction to Computer Science

Lecture 3 (Part II)

Interative and Recursive Algorithms

Artem Burmyakov

Slides prepared with the help of Mikhail Martovitskiy

August 04, 2021

INNOPOLIS UNIVERSITY

Factorial Function – a widely used function in the theory of algorithms

$$n! = 1\times 2\times 3\times \cdots \times (n-1)\times n$$

Factorial Function: a possible implementation

$$n! = 1\times 2\times 3\times \cdots \times (n-1)\times n$$

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Factorial Function: a possible implementation

$$n! = 1\times 2\times 3\times \cdots \times (n-1)\times n$$

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Implementation is based on *for*-loop iterations

Factorial Function: another possible implementation

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

This implementation instead explores the recursive property:

$$n! = n \times (n-1)!$$

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Factorial Function: another possible implementation

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

This implementation instead explores the recursive property:

$$n! = n \times (n-1)!$$

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Base case of the recursion

Factorial Function: possible Implementations

$$n! = 1\times 2\times 3\times \cdots \times (n-1)\times n$$

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Factorial Function: Iterative and Recursive Implementations

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Factorial Function: Iterative and Recursive Implementations

Iterative	**Recursive**
Function repeats until some condition fails (employs loop constructions)	Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Base case of the recursion

Factorial Function: Comparison of Iterative and Recursive Implementations

Iterative Function repeats until some condition fails (employs loop constructions)	**Recursive** Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Which approach computes faster?

Factorial Function: Comparison of Iterative and Recursive Implementations

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Recursive
Function calls itself, until some condition is met,
called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Which approach computes faster?

We need to perform runtime analysis!

Factorial Function: Comparison of Iterative and Recursive Implementations

Iterative	**Recursive**
Function repeats until some condition fails (employs loop constructions)	Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Which approach computes faster?

We need to perform runtime analysis!

Key idea: each operation takes time (or has its cost)

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Operation
Integer variable declaration

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Operation	$cost$
Integer variable declaration	c_1

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Operation	$cost$	*Invocation times*
Integer variable declaration	c_1	2

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Operation	$cost$	$Invocation\ times$
Integer variable declaration	c_1	2
"if" condition in "for" loop	c_2	n

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Operation	$cost$	$Invocation\ times$
Integer variable declaration	c_1	2
"if" condition in "for" loop	c_2	n
"for" loop decrement	c_3	$n-1$

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Operation	$cost$	$Invocation\ times$
Integer variable declaration	c_1	2
“if” condition in “for” loop	c_2	n
“for” loop decrement	c_3	$n-1$
Multiplication	c_3	$n-1$

Runtime Estimation for Iterative Implementation

Iterative
Function repeats until some condition fails
(employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Operation	$cost$	*Invocation times*
Integer variable declaration	c_1	2
"if" condition in "for" loop	c_2	n
"for" loop decrement	c_3	$n-1$
Multiplication	c_3	$n-1$
Function calls number	1	

A total estimated execution time:

The number of times this function is invoked

$$t^{\text{iter}} = 1 \times (2 * c_1 + n * c_2 + (2 * n - 2) * c_3)$$
$$= 2 * c_1 + n * c_2 + (2 * n - 2) * c_3$$

Parameters of given equations are not expensive

Recursive and Iterative implementations: Example of a Factorial Computation

Recursive
Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Recursive and Iterative implementations: Example of a Factorial Computation

Operation	*cost*	*Invocation times*
Integer variable declaration	c_1	2
“if” condition in “for” loop	c_2	n
“for” loop decrement	c_3	$n-1$
Multiplication	c_3	$n-1$
Function call overhead	c_4	1

Recursive

Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Recursive and Iterative implementations: Example of a Factorial Computation

Operation	$cost$	$Invocation\ times$
Integer variable declaration	c_1	2
"if" condition in "for" loop	c_2	n
"for" loop decrement	c_3	$n-1$
Multiplication	c_3	$n-1$
Function call overhead	c_4	1

Recursive
Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Total time of iterative version will be:

$$t^{\text{recurs}} = n\times(c_2 + c_3 + c_4) = n * c_2 + n * c_3 + n * c_4$$

Recursive and Iterative implementations: Example of a Factorial Computation

Operation	*cost*	*Invocation times*
Integer variable declaration	c_1	2
"if" condition in "for" loop	c_2	n
"for" loop decrement	c_3	$n-1$
Multiplication	c_3	$n-1$
Function call overhead	c_4	1
Function calls number (due to recursion)	n	

Recursive
Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Total time of iterative version will be:

$$t^{\text{recurs}} = n \times (c_2 + c_3 + c_4) = n * c_2 + n * c_3 + n * c_4$$

Recursive and Iterative implementations: Example of a Factorial Computation

Operation	*cost*	*Invocation times*
Integer variable declaration	c_1	2
“if” condition in “for” loop	c_2	n
“for” loop decrement	c_3	$n-1$
Multiplication	c_3	$n-1$
Function call overhead	c_4	1
Function calls number (due to recursion)	n	

Recursive
Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Total time of iterative version will be:

$$t^{\text{recurs}} = n\times(c_2 + c_3 + c_4) = n * c_2 + n * c_3 + n * c_4$$

Cost of function call is greater than cost of other operation due to implementation details (the usage of execution stack)

$$c_4 \gg c_1, c_2, c_3$$

Recursive and Iterative implementations: Example of a Factorial Computation

Iterative Function repeats until some condition fails (employs loop constructions)	**Recursive** Function calls itself, until some condition is met, called as a base case

Iterative

Function repeats until some condition fails (employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

$$t^{\text{iter}} = 2 * c_1 + n * c_2 + (2 * n - 2) * c_3$$

Recursive

Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

$$t^{\text{recurs}} = n * c_2 + n * c_3 + n * c_4$$

$c_4 > c_1, c_2, c_3$ in average case

$$t^{\text{recurs}} > t^{\text{iter}}$$

Recursive and Iterative implementations: Example of a Factorial Computation

Iterative
Function repeats until some condition fails (employs loop constructions)

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Works faster (for larger n)

Recursive
Function calls itself, until some condition is met, called as a base case

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Slower, due to function call overheads

(overhead depend on a programming language and many other aspects)

Recursive and Iterative implementations: Example of a Factorial Computation

Iterative Function repeats until some condition fails (employs loop constructions)	**Recursive** Function calls itself, until some condition is met, called as a base case
see code 1 below	see code 2 below
Works faster (for larger n)	Slower, due to function call overheads (overhead depend on a programming language and many other aspects)
Consumes less memory	Consumes more memory

Iterative:

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Recursive:

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Recursive and Iterative implementations: Example of a Factorial Computation

Iterative Function repeats until some condition fails (employs loop constructions)	Recursive Function calls itself, until some condition is met, called as a base case
`unsigned long factorial_itr(unsigned n) { unsigned long res = 1; for (unsigned itr = n; itr > 1; itr--) res = res*itr; return res; }`	`unsigned long factorial_rec(unsigned n) { if (n > 1) return n*factorial_rec(n-1); else // n = 1 return n; }`
Works faster (for larger n)	Slower, due to function call overheads (overhead depend on a programming language and many other aspects)
Consumes less memory	Consumes more memory

Iterative code:

```
unsigned long factorial_itr(unsigned n) {

    unsigned long res = 1;

    for (unsigned itr = n; itr > 1; itr--)
        res = res*itr;

    return res;
}
```

Recursive code:

```
unsigned long factorial_rec(unsigned n) {

    if (n > 1)
        return n*factorial_rec(n-1);
    else // n = 1
        return n;
}
```

Any recursive implementation can be transformed into an iterative implementation (possibly, with a use of an additional data structure, such as stack)

Types of Recursive Functions

```
unsigned long factorial_ntrec(unsigned n) {
    if (n == 1) return n;
    return n*factorial_ntrec(n-1);
}
```

Recursive call is not the last function operation

Types of Recursive Functions

Non-Tail Recursion

Recursive call is not the last function operation

```
unsigned long factorial_ntrec(unsigned n) {
    if (n == 1) return n;
    return n*factorial_ntrec(n-1);
}
```

Types of Recursive Functions

Non-Tail Recursion

Recursive call is not the last function operation

```
unsigned long factorial_ntrec(unsigned n) {
    if (n == 1) return n;
    return n*factorial_ntrec(n-1);
}
```

Tail Recursion

Recursive call is the last function operation

```
unsigned long factorial_trec(unsigned n,
                             unsigned long a) {
    if (n == 1) return a;
    return factorial_trec(n-1, a*n);
}

unsigned long factorial(unsigned n) {
    return factorial_trec(n, 1);
}
```

Faster, due to an optimized usage of a function call stack

Comparison of Iterative and Recursive Implementation

Properties	Function implementations	
	Iterative	Recursive
Run-time complexity in an average case	Proportional to $n * \log(n)$	
Speed	Faster	Slower
Memory	Lower	Higher
Complexity	High	Low

Recursion depth problem: The number of recursive calls can be limited by the Operating System