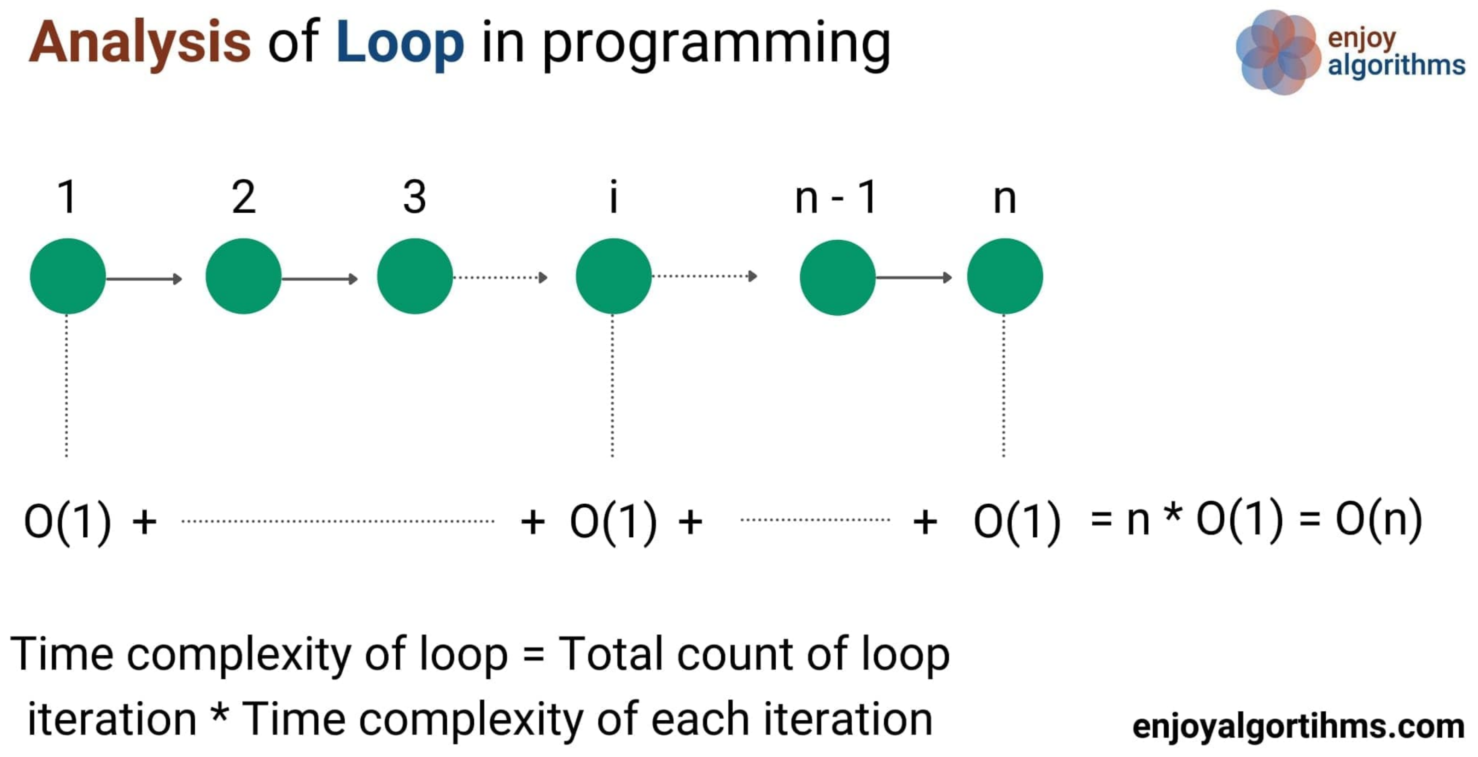
enjoy algorithms

Analysis of loop in Programming


Analysis of Loop in programming
enjoy algorithms
1
2
3
i
n - 1
n
O(1) + + O(1) + + O(1) = n * O(1) = O(n)
Time complexity of loop = Total count of loop iteration * Time complexity of each iteration
enjoyalgortihms.com

Loop is a fundamental problem-solving operation in programming. A lot of coding problem solutions involve various kinds of loop structures. Moreover, some problem-solving approaches are purely based on loop—building partial solutions using a single loop, two-pointers approach, sliding window approach, BFS traversal, bottom-up approach of dynamic programming, problem-solving using stack, etc. So the efficiency of such an algorithm depends on the loop structure and operations inside the loop.

These two loop patterns appear frequently in our solutions:

- **Single loop:** loop running constant time, a loop is running n times, loop growing exponentially, a loop running based on the specific condition, a loop running with data structure, consecutive single loops, etc.
- **Nested loops:** two nested loops, three nested loops, single loop with nested loops

One idea would be simple: to design a better algorithm or optimize the code further, we should learn to analyze the time complexity of the loop in terms of Big-O notation. Learning time complexity analysis of loops is not difficult. After some basic practice of various loop patterns, we can make optimization decisions quickly and save a lot of analysis time.

Steps to analyze the time complexity of the loop

- **Counting the total loop iteration in the worst case:** we can get this insight from considering the worst-case scenario, initial and final value of the loop variable, loop condition, and increment or decrement operation. Most of the time loop will be running for each data element or total input size.
- **Calculating the time complexity of the code in the loop body:** loop executes this code on each iteration to get the final result. This code may contain conditional statements, comparison operation, swapping operation, assignment operation, etc.
- **The time complexity of loop** = (count of loop iterations in the worst case) * (time complexity of the code in the loop body). We represent this in the form of Big-O notation by ignoring lower-order terms and coefficients.

Sometimes, we can also follow another simple approach:

Identify the most critical operation inside the loop, which executes the maximum number of times in the worst case. This critical operation would be the dominating factor in the time complexity function.

Now calculate the total count of this operation for the complete loop in terms of input size. Representing this expression in terms of Big-O notation will give the time complexity of the loop.

Let's analyze the time complexity of the various loop pattern.

Time complexity analysis of a single loop

A loop running constant times: O(1)

```
for (int i = 1; i <= c; i = i + 1)
{
    some O(1) expressions
}
```

Here loop is running constant times and performing O(1) operation at each iteration of the loop. Time complexity = c*O(1) = O(1)*O(1) = O(1)

A loop running n times and incrementing/decrementing by a constant: O(n)

Example 1: loop incrementing by some constant c

```
for (int i = 1; i <= n; i = i + c)
{
    some O(1) expressions
}
```

Example 2: loop decrementing by some constant c

```
for (int i = n; i > 0; i = i - c)
{
    some O(1) expressions
}
```

Here both loops are running some n times and performing O(1) operation at each iteration of the loop. Time complexity = n*O(1) = O(n)*O(1) = O(n)

A loop running constant multiple of n times: O(n)

```
for (int i = 1; i <= 3*n; i = i + 1)
{
    some O(1) expressions
}
```

Here loop is running some cn times and performing O(1) operation at each iteration of the loop. Time complexity = cn*O(1) = O(n)*O(1) = O(n)

Two pointers loop: O(n)

```
l = 0, r = n - 1
while (l <= r)
{
    if (condition)
    {
        some O(1) expressions
        l = l + 1
    }
    else
    {
        some O(1) expressions
        r = r - 1
    }
    some O(1) expressions
}
```

In the above loop, based on some conditions, we are either incrementing l or decrementing r by one and performing an O(1) operation at each step of the iteration. Loop will run n times because l and r are starting from opposite ends and end when l > r. So time complexity = n*O(1) = O(n)

A loop incrementing/decrementing by a constant factor: O(logn)

Example 1: loop incrementing by a factor of 2

```
for (int i = 0; i < n; i = i*2)
{
    some O(1) expressions
}
```

Example 2: loop decrementing by a factor of 2

```
for (int i = n; i > 0; i = i/2)
{
    some O(1) expressions
}
```

Here loop is running in the range of 1 to n, and the loop variable increases or decreases by a factor of 2 at each step. Thus, we need to count the total number of iterations performed by the loop to calculate the time complexity. Let's assume the loop will terminate after k steps where the loop variable increases or decreases by a factor of 2. Then 2^k must be equal to the n i.e. 2^k = n and k = logn = O(logn).

So loop will run O(logn) number of times and doing O(1) operation at each step. Time complexity = k * O(1) = O(logn)* O(1) = O(logn)

Loop incrementing by some constant power: O(log(logn))

```
// Here c is a constant greater than 1
for (int i = 2; i < = n; i = pow(i, c))
{
    some O(1) expressions
}
```

Here, the loop is running in the range of 1 to n, but the loop variable increases by factor i power constant c. So how do we calculate the total number of loop steps? Let's think!

> The first iteration of the loop is starting with i = 2.
>
> At second iteration, value of i = 2^c.
>
> At third iteration, value of i = (2^c)^c = 2^(c^2)
>
> And it will go so on till the end. At any ith iteration the value of i = 2^(c^i)

Loop will end when 2^(c^i) = n

```
2^(c^i) = n
Let's take log of base 2 from both sides.
=> log2(2^(c^i)) = log2(n)
=> c^i = logn

Again take log of base c from both sides.
=> logc(c^i) = logc(logn)
=> i = logc(logn)
=> i = O(log(logn))
```

So loop will run logc(log(n)) number of times, where each iteration is taking O(1) time. So the overall time complexity = O(log(log(n))) * O(1) = O(log(log(n)))

Consecutive single loops: O(m + n)

```
for (int i = o; i < m; i = i + 1)
{
    some O(1) expressions
}
for (int i = 0; i < n; i = i + 1)
{
    some O(1) expressions
}
```

For calculating such consecutive loops, we need to do the sum of the time complexities of each loop. So overall time complexity = Time complexity of loop 1 + Time complexity of loop 2 = O(m) + O(n) = O(m + n)

Time complexity analysis of the nested loops

The time complexity of nested loops is equal to the number of times the innermost statement is executed.

Two nested loops: O(n^2)

```
for (int i = 0; i < n; i = i + 1)
{
    for (int j = 0; j < n; j = j + 1)
    {
          some O(1) expressions
    }
}
```

In the above nested-loop example, the inner loop is running n times for every iteration of the outer loop. So total number of nested loop iteration = total number of iteration of outer loop total number of iteration of inner loop = $n * n = n^2 = O(n^2)$.

At each step of the iteration, the nested loop is doing an O(1) operation. So overall time complexity = $O(n^2) * O(1) = O(n^2)$

```
for (int i = 0; i < n; i = i + 1)
{
    for (int j = i + 1; j < n; j = j + 1)
    {
        some O(1) expressions
    }
}
```

In the above nested loop example, outer loop is running n times and for every iteration of the outer loop, inner loop is running (n - i) times. So total number of nested loop iteration = (n - 1) + (n - 2) + (n - 3)…..+ 2 + 1 = sum of arithmatic series from i = 0 to n - 1 = n(n - 1)/2 = $n^2/2$ - n/2 = $O(n^2)$ At each step of the iteration, the nested loop is doing an O(1) operation. So overall time complexity = $O(n^2)$ * O(1) = $O(n^2)$

Note: it’s an exercise for you to analyze the following loop.

```
for (int i = 0; i < n; i = i + 1)
{
    int j = i
    while (j > 0)
    {
        some O(1) expressions
        j = j - 1
    }
}
```

Combination of single and nested loops

We need to do the sum of the time complexities of each loop. In such a case, the time complexity is dominated by the time complexity of the nested loop.

```
for (int i = 0; i < n; i = i + 1)
{
    some O(1) expressions
}

for (int i = 0; i < n; i = i + 1)
```

```
{
    for (int j = 0; j < m; j = j + 1)
    {
        some O(1) expressions
    }
}

for (int k = 0; k < n; k = k + 1)
{
    some O(1) expressions
}
```

So, time complexity = Time complexity of loop 1 + Time complexity of loop 2 + Time complexity of loop 3 = O(n) + O(mn) + O(n) = O(mn)

Three nested loops: $O(n^3)$

```
for (int i = 0; i < m; i = i + 1)
{
    for (int j = 0; j < n; j  = j + 1)
    {
        for (int k = 0; k < n; k  = k + 1)
        {
            some O(1) expressions
        }
    }
}
```

All three nested loops are running n times and doing O(1) operation at each iteration, so time complexity = n * n * n*O(1) = n^3 * O(1) = O(n^3)*O(1) = O(n^3)

```
for(int i = 1; i < n; i = i + 1)
{
    for(int j = i + 1; j <= n; j  = j + 1)
    {
        for(k = i; k <= j; k  = k + 1)
        {
            some O(1) expressions
        }
    }
}
```

In the above three nested loop situations, the outer loop runs n - 1 time, but two inner loops run n - i and j -i + 1 time. So what would be the total count of the nested loop iterations? Let's think.

```
       n - 1     n        j
T(n) =   ∑       ∑        ∑    O(1)
       i = 1  j = i + 1  k = i
```

Now we solve this tripple summation by expanding the summation one by one.

$$T(n) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=i}^{j} c = c\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(j-i+1)$$
$$T(n) = c\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n} j - \sum_{j=i+1}^{n} i + \sum_{j=i+1}^{n} 1)$$
$$T(n) = c\sum_{i=1}^{n-1}([\frac{(n-i)(i+n+1)}{2}] + i(n-i) + [n-i])$$
$$T(n) = c\sum_{i=1}^{n-1}([\frac{(n-i)[(i+n+1)+2i+2]}{2}]) = c\sum_{i=1}^{n-1}([\frac{n^2+2in+3n-3i^2-3i}{2}])$$
$$T(n) = \frac{c}{2}\sum_{i=1}^{n-1}(n^2) + \sum_{i=1}^{n-1}(2in) - 3(\sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} n)$$
$$T(n) = \frac{c}{2}[(n-1)n^2 + n^2(n-1) - (\frac{n(n-1)(2n-1)}{2}) - \frac{3n(n-1)}{2} + (n-1)3n)$$
$$T(n) = \frac{c[n(n-1)(n+2)]}{2}$$

Higher-order term in T(n) is n^3, then T(n) = O(n^3). We are ignoring lower-order terms and coefficients.

Enjoy learning, Enjoy coding, Enjoy algorithms!

Author
Shubham Gautam

Reviewer
EnjoyAlgorithms Team

coding-interview-concepts time-complexity-analysis

Name

Email address

Message

Share Feedback

More from EnjoyAlgorithms

What is Data Structure? Types, Classification and Applications

The code structure of a well-designed algorithm using data structure is just like a structure of a good house. So a design of

Analysis of Recursion in Data Structure and Algorithms

Learning analysis of recursion is critical to understand the time complexity analysis of recursive algorithms. We will discuss an algorithm must be based on a good understanding of data...

Read More

these concepts related to the recursion analysis: Recurrence...

Read More

How to develop algorithmic thinking in data structure and algorithms?

Algorithmic thinking definition: It is a method for solving algorithms and data structure problems based on a clear definition of the steps logically and repeatedly. Therefore,...

Read More

Sorting Algorithms Comparison

Comparison of sorting algorithms based on different parameters helps us choose an effcient sorting method in various problem-solving scenarios. You will get an answer to...

Read More

Bubble Sort, Selection Sort and Insertion Sort

Sorting algorithms are the most fundamental problems to study in data structure and algorithms. But the critical question

is - why we learn the design, code, and analysis of the sorting...

Read More

Iterative Binary Tree Traversal using Stack: Preorder, Inorder and Postorder

In recursive DFS traversals of a binary tree, we have three basic elements to traverse— root, left subtree, and right subtree. The traversal order depends on the order in which we process the...

Read More

Recursive Tree Traversals of a Binary Tree: Preorder, Inorder and Postorder Traversal

To process data stored in a binary tree, we need to traverse each tree node, and the process to visit all nodes is called binary tree traversal. In this blog, we will be discussing three...

Read More

Recursion Explained: How recursion works in Programming?

Recursion means solving the problem via the solution of the smaller sub-problem. This blog will answer some critical questions like - what is recursion? What are its advantages an...

Read More

Introduction to Heap Data Structure

A heap is a complete binary tree structure where each element satisfies a heap property. We learn two types of the heap in

pr[illegible] 1) max-heap, which satisfies max heap propert...

Read More

Array In Data Structures

An array is a contiguous block of memory of the same type of elements where the size is equal to the number of elements in that array, which must be fixed. It is a structure of data such...

Read More

What is an Algorithm? Properties and Applications of Algorithms in real life

The concept of algorithm is essential for building high-performance software applications and cracking the coding

Step by Step Guide for Cracking the Coding Interview

A comprehensive guide to master data structures and algorithms and crack the coding interview. This could help interview to get a high-paying job in the software industry. So...

Read More

programmers prepare a step-by-step learning plan for the...

Read More

Our Weekly Newsletter

Subscribe to get free weekly content on data structure and algorithms, machine learning, system design, oops design and mathematics.

Email address

Subscribe

Home

Coding Interview

System Design

Machine Learning

OOPS Design

Latest Content

Success Stories

About Us

Contact Us

Follow us on: