Time Complexity Analysis of Loop in Programming

Loop is a fundamental problem-solving operation in programming, and solutions to many coding problems involve various kinds of loop structures. Even some problem-solving approaches are 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 algorithm depends on the loop structure and operations inside the loop.

These two loop patterns frequently appear in our solutions:

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

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 basic practice with various loop patterns, we can make optimization decisions quickly and save time.

Steps to analyze time complexity of loop

Counting total loop iteration in the worst case: we can get this insight by 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 run for each data element or total input size.

Calculating time complexity of code in the loop body: loop executes this code on each iteration to get the final result. This code may contain conditional statements, comparison operations, swapping operations, assignment operations, etc.

So the time complexity of loop = (Count of loop iterations in the worst case) * (Time complexity of code in the loop body). We represent this in 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 worst case. This critical operation would be dominating factor in the time complexity function.
- Now, calculate the total count of this operation for complete loop in terms of input size. Representing this expression in terms of Big-O notation will give the time complexity of loop.

Let’s analyze the time complexity of various loop patterns.

Time complexity analysis of single loop

for 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. Time complexity = c * O(1) = O(1) * O(1) = O(1)

for loop running n times and incrementing/decrementing by 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 n/c times and performing O(1) operation at each iteration. Time complexity = n/c * O(1) = O(n) * O(1) = O(n)

for loop running constant multiple of n: O(n)

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

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

Two pointers while 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 1 and performing O(1) operation at each iteration. Loop will run n times because l and r are starting from opposite ends and stop when l > r. So time complexity = n * O(1) = O(n)

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

Example 1: Loop incrementing by factor of 2

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

Example 2: Loop decrementing by 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 loop variable increases or decreases by a factor of 2 at each iteration. So, we need to count the total number of iterations performed by loop to calculate the time complexity.

Let’s assume the loop will terminate after k steps where loop variable increases or decreases by factor of 2. So, 2^k must be equal to the n => 2^k = n => k = logn. So loop will run **logn** number of times and doing O(1) operation at each step. Time complexity = k * O(1) = logn* O(1) = **O(logn)**

for 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^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^(c²)
- 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)
```

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

Consecutive single for 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 do the sum of 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 nested for loops

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

Two nested for loops: O(n²)

```
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, inner loop is running n times for every iteration of the outer loop. So total number of nested loop iteration = Total iteration of outer loop * total iteration of inner loop = n * n = n^2

At each step of iteration, nested loop is doing an O(1) operation. So overall time complexity of nested for loops = 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 iteration, nested loop is doing O(1) operation. So overall time complexity of nested for loops = 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 for loops

In such a case, we need to do the sum of time complexities of each loop, but time complexity will be dominated by the time complexity of 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
}
```

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 for 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, outer loop runs n - 1 time and two inner loops run n - i and j - i + 1 time. So what would be the total count of 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.

Note: There is one error in the third line of the above image. Instead of + i(n - i), it would be - i (n - i).