Solution Review: Big O of Nested Loop with Multiplication

This review provides a detailed analysis of the different ways to solve the Big O of Nested Loop with Multiplication Quiz!

We'll cover the following

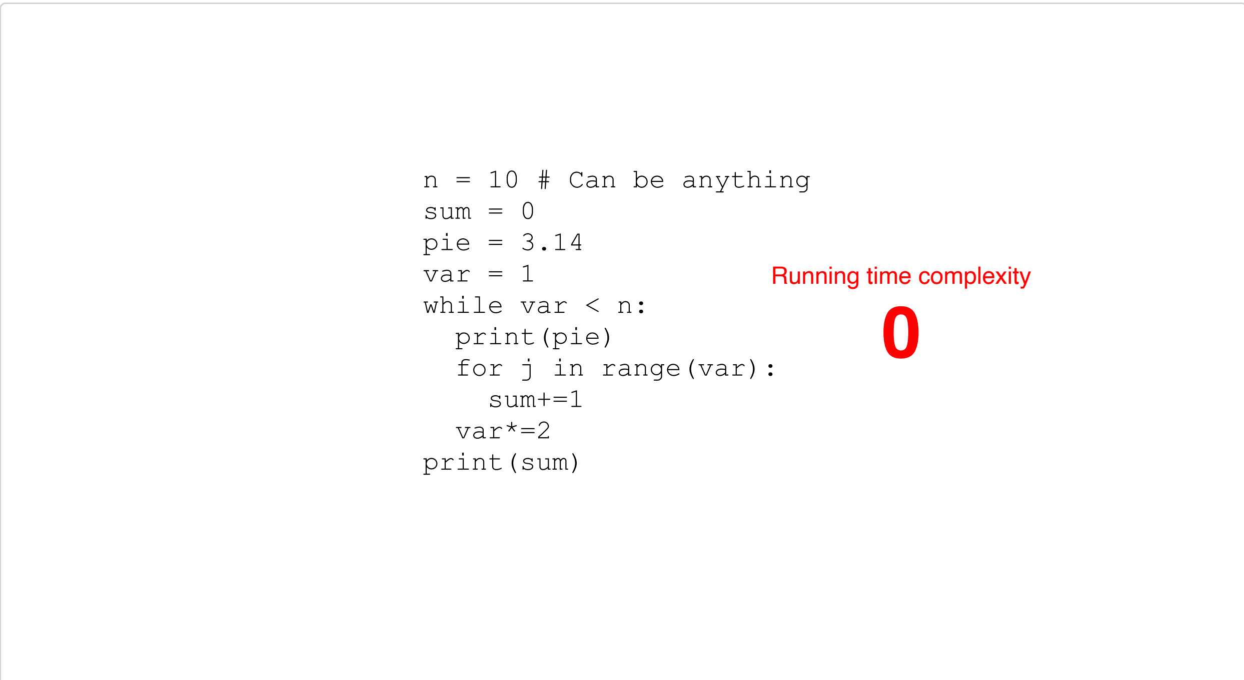
Solution

```python
n = 10  # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
    print(pie)
    for j in range(var):
        sum += 1
    var *= 2
print(sum)

```

Explanation

The answer is $O(n)$. Have a look at the slides below for an in-depth explanation of the answer.



Lets dry run this code to calculate its running time complexity


```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

1

initializing n


```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

2

initializing sum

```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

3

initializing sum


```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

4

initializing sum

```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

4

The number of times that the while loop runs depends on two variables: var and n. Let's track how they change in the following few slides


```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

4

In the entire body of the while loop, n does not change, so lets look at var now

```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

4

var gets doubled on every iteration. So how many iterations will we have in total? Let's count them

`var = 1 = 2^0`	first
`var = 2 = 2^1`	second
`var = 4 = 2^2`	third
`var = 8 = 2^3`	fourth
`var = 16 = 2^4 = 2^(ceil(log_2(n)))`	last but condition that var < n is not met

the values of `var` as the loop progresses

Hence, the outer loop runs 4 times which is equal to ceil(log_2(n))

The loop runs ceil(log_2(n))

So the total running time of the outer loop is comparisons in while
+
printing pie
+
doubling var

Lets figure it out

So the total running time of the outer loop is
comparisons in while
+
printing pie
+
doubling var

The while comparisons occur once more than the times the loop runs (for obvious reasons)

So the total running time of the outer loop is
ceil(log_2(n))+1
+
printing pie
+
doubling var

Plugging in the number of times that the loop runs plus 1

So the total running time of the outer loop is
ceil(log_2(n))+1
+
printing pie
+
doubling var

Now lets figure out how long printing pie takes

So the total running time of the outer loop is
ceil(log_2(n))+1
+
ceil(log_2(n))
+
doubling var

pie is printed at every iteration of the loop so lets plug that value in

So the total running time of the outer loop is
ceil(log_2(n))+1
+
ceil(log_2(n))+1
+
doubling var

Not to see how much doubling var costs

So the total running time of the outer loop is
ceil(log_2(n))+1
+
ceil(log_2(n))+1
+
doubling the value of var
+
setting var equal to new value

We can break this one down into unit statements like so

So the total running time of the outer loop is
ceil(log_2(n))+1
+
ceil(log_2(n))
+
ceil(log_2(n))
+
ceil(log_2(n))

Each costs us this much based on the iterations of the while loop

So the total running time of the outer loop is
4ceil(log_2(n))+1

The total running time complexity of the outer loop

Great! Let's move on to the inner loop now.
To understand the running time of the inner loop,
we'll work with an example where n = 16

Now let's move on to the inner loop

`n = 16`

`var = 1`

var = 1 when n = 16 in the first iteration of the outer while loop.

For loop iterations

1 + ...

So the statement inside the while loop runs once

n = 16

var = 1 x 2

Then, var becomes 4 as it is multiplied by 2 on line 9

```
n = 16

var = 2
```

var is now 2


```
For loop iterations

1 + 2 + ...
```

The statement inside the for loop runs twice on the second iteration of the outer while loop

```
n = 16

var = 2 x 2
```

Then, var becomes 4 as it is multiplied by 2 on line 9


```
n = 16

var = 4
```

var is now 4

For loop iterations

1 + 2 + 4 + ...

The statement inside the for loop runs four times on the third iteration of the outer while loop

n = 16

var = 4 x 2

Then, var becomes 8 as it is multiplied by 2 on line 9

```
n = 16

var = 8
```

var is now 8


```
For loop iterations

1 + 2 + 4 + 8 + ...
```

var is still less than n so the outer while loop keeps going. The statement inside the for loop runs eight times on the fouth iteration of the outer while loop.

```
n = 16
```

```
var = 8 x 2
```

Then, var becomes 16 as it is multiplied by 2 on line 9


```
n = 16
```

```
var = 16
```

var is now 16. The outer while loop stops at this point because 16 is not less than 16.

Total for loop iterations

1 + 2 + 4 + 8
= 2^0 + 2^1 + 2^2 + 2^3
= 2^0 + 2^1 + 2^2 + .. + 2^k
= 2^(k+1)-1

So to figure out how many times this for loop runs, we need to calculate the value of k.

2^k < n

We know that the last value of var has to be less than n, so 2^k has to be less than n too

```
log_2(2^k) < log_2(n)
```

We can now apply the log_2 function to both sides of the equation


```
k < log_2(n)
k is in O(log_2(n))
```

That leaves us with k < log_2(n). So its safe to say that k is in order of log_2(n) while it may never be equal to log_2(n).

Total for loop iterations

1 + 2 + 4 + 8

= 2^0 + 2^1 + 2^2 + 2^3

= 2^0 + 2^1 + 2^2 + .. + 2^k

= 2^(k+1)-1

Let's plug the value of k back into our original equation

Total for loop iterations

= 2^(k+1)-1

Let's plug the value of k back into our original equation

Total for loop iterations

= 2^(k+1)-1

k is in O(log_2(n))

Let's plug the value of k back into our original equation

Total for loop iterations

= 2^(k+1)-1

k is in O(log_2(n))

= 2^(log_2(n)+1)-1

Let's plug the value of k back into our original equation

Total for loop iterations

= 2^(k+1)-1

k is in O(log_2(n))

= 2^(log_2(n)+1)-1

= 2^(log_2(n))2^1-1

Let's simplify the equation a bit

Total for loop iterations

= 2^(k+1)-1

k is in O(log_2(n))

= 2^(log_2(n)+1)-1

= 2^(log_2(n))2^1-1

= 2n - 1

Let's simplify the equation a bit

Total for loop iterations

< 2n - 1

The statements inside the for loop run in order of 2n-1


```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

outer + inner

Lets get the time complexity of the entire code by summing the time complexity of the outer and inner loops

```
n = 10 # Can be anything
sum = 0
pie = 3.14
var = 1
while var < n:
  print(pie)
  for j in range(var):
    sum+=1
  var*=2
print(sum)
```

Running time complexity

outer + inner

Why do we sum the complexities and not multiply them you ask? Well thats because we've already considered the time complexity of the outer loop when we calculated the time complexity for the inner loop with the geometric series!

Running time complexity of inner for loop is rtc of for loop statement + rtc of statements in the loop

Lets first calculate the running time of the inner loop. For this, we'll have to calculate the time complexity of the statements that make up the for loop.

Running time complexity
of inner for loop is
rtc of for loop statement
(for j in range var)
+
rtc of statements in the loop

Lets start with the for loop statement

Running time complexity
of inner for loop is
rtc of list generation from range
+
rtc of setting j equal to
values from the list
+
rtc of statements in the loop

The for loop statement can be broken down further into two parts

Running time complexity
of inner for loop is
$< 2n-1$
+
$< 2n-1$
+
rtc of statements in the loop

Each costs us less than 2n-1

Running time complexity
of inner for loop is
$< 4n-2$
+
rtc of statements in the loop

Together, they cost us less than 4n-2

Running time complexity of inner for loop is
< 4n-2
+
rtc of statements in the loop (sum+=1)

Lets move on to the statements inside the loop itself

Running time complexity of inner for loop is
< 4n-2
+
adding one to the value of sum
+
setting sum equal to new value

They can be further broken down like so

Running time complexity of inner for loop is

< 4n-2

+

< 2n-1

+

< 2n-1

Each of these is cost less than 2n-1 units

Running time complexity of inner for loop is

< 4n-2

+

< 4n-2

Together they cost us less than 4n-2

Running time complexity of inner for loop is

$< 8n-4$

Finally, all the statements in the inner for loop cost less than 8n-4 time

Running time complexity

outer + inner

Lets plug both values back into this formula

Running time complexity

4(ceil(log_2(n)))+1

+

8n-4

This is what we get. Let's see what the answer is in big O next.

Running time complexity

ceil(log_2(n))

+

n

Dropping the constants

Running time complexity

n

Dropping the lower order terms

60 of 61

Running time complexity

O(n)

Voila! The answer is O(n). Such a harmless looking time complexity took so much effort! Do read the logic below though, its a lot simpler and easier to understand. These slides are just meant for thoroughness,

61 of 61

Time Complexity

The outer loop here runs $log(n)$ times and the inner loop runs $2n - 1$ times where the value of `var` is 1 in the first iteration, then 2, then 4, then 8 and so on until 2^k such that $2^k <$ `n`. So, in total the inner loop runs $1 + 2 + 4 + 8 + \cdots + 2^k$ times. We'll use geometric series (https://en.wikipedia.org/wiki/1_%2B_2_%2B_4_%2B_8_%2B_%E2%8B%AF) to figure out this value. To make this calculation simpler, let's assume that $2^k =$ `n`

$$2^0 + 2^1 + 2^2 ... + 2^k = 2^{k+1} - 1$$

$= 2^k 2^1 - 1$

Substituting n for 2^k we get,

$= 2n - 1$

So it appears that the inner loop runs a total of `2n−1` times, but remember that we assumed that $n = 2^k$ when $n > 2^k$ so, in actuality, the inner loop runs less than $2n - 1$ times but we can consider this to be the upper bound.

Hence, the Big O Time Complexity is $O(n)$

← Back

Challenge 3: Big O of Nested Loop wit...

Next →

Challenge 4: Nested Loop with Multipl...

Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-big-o-of-nested-loop-with-multiplication__introduction-to-complexity-measures__data-structures-for-coding-interviews-in-python)