Time Complexity of function that contains another function

Asked 3 years, 6 months ago Modified 3 years, 6 months ago Viewed 564 times

-1

Can someone meticulously explain how do I figure out the time complexity of this code?

```
int f(int n)
{
    int sum = 0;
    while (n>1)
    {
        sum +=g(n)
        n = sqrt(n)
    }
    return sum;
}
```

where g(n) is given by:

```
int g(int n)
{
    int sum  = 0;
    for (int i = 1; i<n; i*=2)
        sum +=i;
    return sum;
}
```

Thanks in advance!

for-loop time while-loop complexity-theory

Share Improve this question Follow

edited Aug 29, 2018 at 11:52
Amadan
176k 19 214 272

asked Aug 29, 2018 at 11:37
Ron73404
21 3

3 Answers

Active Oldest Votes

1

A slightly more concrete way of proving the result:

As a previous answer correctly stated, the complexity of `g(n)` is `O(log n)`. The precise number of times the loop in `g(n)` executes is `floor(log2(n)) + 1`.

Now for `f(n)`. The value of `n` after the `m` -th iteration of the loop, with respect to the *original* value of `n`, is:

$$
\begin{aligned}
n_0 &= n \\
n_1 &= \sqrt{n} &&= n^{\frac{1}{2}} \\
n_2 &= \sqrt{\sqrt{n}} &&= n^{\frac{1}{2^2}} \\
n_3 &= \sqrt{\sqrt{\sqrt{n}}} &&= n^{\frac{1}{2^3}} \\
&\cdots \\
n_m &= n^{\frac{1}{2^m}}
\end{aligned}
$$

From this, using the loop condition `n > 1`, the number of times this loop executes is:

$$m = \lfloor \log_2 \log_2 n \rfloor$$

This allows one to express the complexity function of `f(n)` as a summation:

$$
\begin{aligned}
T(n) &= \sum_{m=0}^{\lfloor \log_2 \log_2 n \rfloor} \left(\left\lfloor \log_2 n^{\frac{1}{2^m}} \right\rfloor + 1 \right) \\
&= \sum_{m=0}^{\lfloor \log_2 \log_2 n \rfloor} \left(\frac{1}{2^m} \log_2 n + O(1) + 1 \right) \qquad (*) \\
&= \log_2 n \times \sum_{m=0}^{\log_2 \log_2 n + O(1)} \frac{1}{2^m} + O\left(\log \log n\right) \\
&= \log_2 n \times \underbrace{\frac{1 - 2^{-\log_2 \log_2 n + O(1)}}{1 - \frac{1}{2}}} + O\left(\log \log n\right) \quad (**)
\end{aligned}
$$

In `(*)` I used the fact that a number rounded down only differs from its original value by *less* than 1 (hence `O(1)`). In `(**)` I used the standard result for geometric series sums.

The underlined term in `(**)` has a negative power of 2. When `n` tends to infinity, this term vanishes, so the underlined term itself converges to 2.

Therefore the final complexity is just `O(log n + log log n) = O(log n)` , since the first term dominates.

Share Improve this answer Follow

answered Aug 29, 2018 at 12:30

meowgoesthedog
13.9k 4 23 36

0

`g` is logarithmic on its arguments (if you pass it `n` , its loop repeats `log[2](n)` times, since it takes that many iterations for the doubling of `i` to reach `n` .

`f` is doubly logarithmic - it halves the *exponent* of `n` in each operation, for `log[2](log[2](n))` repetitions.

We can disregard the fact that `g` is a separate function - effectively, it is a loop nested within another loop. We can find a better limit if we analyse exactly how the number of repetitions of `g` decreases as `f` progresses, but `O(log n * log log n)` is meh good enough. (Complexity theory is like seafood: while "I ate bluefin tuna" might be *the* correct answer, "I ate fish" is not wrong.)

EDIT:

> However the right answer is O(log(n)) (final test answer) and I don't understand why....

As I said:

> We can find a better limit if we analyse exactly how the number of repetitions of `g` decreases as `f` progresses

but honestly, this is easier done from results than code. Say `n` starts off as 65536. This will give us 16 iterations of `g`. Root of it is 256, which will allow `g` to run 8 times. Next up is 16, for 4 iterations of `g`. Then 4 for 2, and 2 for 1. This looks like a geometric progression: `16+8+4+2+1 = 32-1`, where `32` is `2 * log[2](65536)`, which is consistent with O(log n).

Or you could notice that in the first iteration of `f` there will be a lot of iterations of `g`, compared to which all the other invocations of `g` are irrelevant (disappearing logarithmically). Since that first invocation of `g` is O(log(n))`, we can just truncate it there and say that's the complexity.

Share Improve this answer Follow

edited Aug 29, 2018 at 12:17

answered Aug 29, 2018 at 11:59

Amadan
176k 19 214 272

However the right answer is O(log(n)) (final test answer) and I don't understand why.... – Ron73404 Aug 29, 2018 at 12:01

-1

Big O notation to describe the asymptotic behavior of functions. Basically, it tells you how fast a function grows or declines

For example, when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by

```
T(n) = 4 n^2 - 2 n + 2
```

If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms, we could say "T(n)" grows at the order of n^2 " and write:T(n) = O(n^2)

For the formal definition, suppose f(x) and g(x) are two functions defined on some subset of the real numbers. We write

```
f(x) = O(g(x))
```

(or f(x) = O(g(x)) for x -> infinity to be more precise) if and only if there exist constants N and C such that

```
|f(x)| <= C|g(x)| for all x>N
```

Intuitively, this means that f does not grow faster than g

If a is some real number, we write

```
f(x) = O(g(x)) for x->a
```

if and only if there exist constants d > 0 and C such that

```
|f(x)| <= C|g(x)| for all x with |x-a| < d
```

So for your case it would be

```
O(n) as |f(x)| > C|g(x)|
```

Reference from http://web.mit.edu/16.070/www/lecture/big_o.pdf

```
for r from 0 to xlist:  // --> n time
       for c from 0 to ylist: // n time
         sum+= values[r][c]
         n+1

}
```

Big O Notation gives an assumption when value is very big outer loop will run n times and inner loop is running n times

Assume n -> 100 than total n^2 10000 run times

Share Improve this answer Follow

answered Aug 29, 2018 at 12:36

SarthAk

1,428 3 14 24