Module 1; (Time and Space complexity)

Video 1-0;(Introduction)

What will we learn in this module :

1. Time and Space Complexity
2. O(1), O(logN), O(sqrt(N))
3. O(N), O(NlogN), O(N\*N)

Video 1-1; What is Time Complexity?  
  
Big O notation -> O(steps);

Steps depends on 2 things: 1. Grammar(syntax and logics of code), 2. Input.  
  
int main(void) Time (Unit)

{  
 int a = 10; 1  
 int b = 20; 1  
 int c = a + b; 1  
 printf(“Sum: %d”, c); 1  
  
 return 0; 1  
}  
  
for now, we can say O(1+1+1+1+1) = O(5);

for (int I = 1; i <= N; i++) { O(N+1)

cout << “HELLO” << endl; O(N)

}

Grammar:

1. Always Calculate for worst case(there are 3 cases, best, worst, average)
2. Ignore the constants.
   1. For example: 2N, here we will ignore 2 and write N
   2. For example: N/3, here we will ignore 1/3 and write N

imagine this,

O(n + n/2 + 100 + 1)  
= O(n + n) here we ignored constant values right? [line 2]  
= O(2n) [line 3]  
= O(n) here we ignored constant value also [line 4]

Think in line 2, we ignored 100, 1 and ½ because they are constant values,  
imagine n = 1000000000000000000000, then the constant values we ignored are nothing compared to n, so they are ignorable.

More examples:  
  
Example 1:

for (int I = 1; i <= 2N; i++) { O(N+1) here we ignored constant value 2

cout << “HELLO” << endl; O(N) here also we ignored constant value 2

}

Example 2:

for (int I = 1; i <= N/2; i++) { O(N+1) here we ignored constant value 1/2

cout << “HELLO” << endl; O(N) here also we ignored constant value

}

Example 3: (Input based)

for (int I = 1; i <= N; i++) { O(N+1)

cout << “HELLO” << endl; O(N)

}

for (int I = 1; i <= M; i++) { O(M+1)

cout << “HELLO” << endl; O(M)

}  
  
here we can write O(N+M); because, we don’t know value of both variables N and M.

Generally we calculate for the maximum possible value, since we don’t know the value of N and M, we don’t know what will user give input, he can give any input for example:

For N he can input: 100000000000000000;  
for M he can input: 1000000;

So here we should write O(N) because N is the biggest possible value between N and M so instead of writing O(N+M) we can write O(N). Because M is nothing compared to N. But still, what the user will input that is unpredictable, so we can write O(N+M);

Since we now know how to calculate basic time complexity, remember this program that we wrote at the before in this part of module.

int main(void)

{  
 int a = 10; 1  
 int b = 20; 1  
 int c = a + b; 1  
 printf(“Sum: %d”, c); 1  
  
 return 0; 1  
}  
  
we calculated O(1+1+1+1+1) = O(5);

Now let’s calculate again,

int main(void)

{  
 int a = 10; 1 = we can write it as O(1)  
 int b = 20; 1 = we can write it as O(1)  
 int c = a + b; 1 = we can write it as O(1)  
 printf(“Sum: %d”, c); 1 = we can write it as O(1)  
  
 return 0; 1 = we can write it as O(1)  
}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Total Time complexity = O(1) [since they are constant values]

Comment:

1. We won’t be taking time complexity from single line statements that has a constant time complexity O(1) [pronounce: order of 1]
2. Most of the time complexity we will be taking are from loops. Because they depend on user input.
3. We will be ignoring constant values.

Video 1-2; How Time is Calculated from Time Complexity?

Each server gives maximum 1s time for each test case.

In 1s, server can perform (almost)operations ≈ (safely) operations.

This means we can perform operations safely. If we go far more than this the server may throw time limit exceeded (timeout).

In 2s, server can perform (safely) 2x operations = 20000000 operations.

In 3s, server can perform (safely) 3xoperations = 30000000 operations.

This operation may not include only loops but also general statements in the code.  
  
Imagine a program that has a O(n) time complexity.  
O(n) here we can write O() which means the program will take 1s to run.  
  
This is how we can calculate the program’s run time using time complexity.

For example:  
 O(2x) means the program will take 2 seconds to run.  
 O(3x) means the program will take 3 seconds to run.

Since [O(n)] n is a input value from the user. So we will/can calculate the time complexity based on user input.

= 1000 takes 0.0001s  
 = 10000 takes 0.001s  
 = 100000 takes 0.01s  
 = 1000000 takes 0.1s  
 = 10000000 takes 1s [main point]  
 = 100000000 takes 10s  
 = 1000000000 takes 100s  
 = 10000000000 takes 1000s  
 = 100000000000 takes 10000s

Why efficiency matters?  
Answer: Imagine a problem, if there is a time limit of 1s per test case.

Program’s time complexity is O(n) where n is user input. If the user input is will it work? The program will run but the online judge will not accept the code.  
Because it takes more time than 1s [O() should take 100000s] which is not allowed. That means we need to write a better code that is efficient enough to solve the problem within 1s. So we have to develop it further and reduce the time complexity.

Video 1-3; O(N) Time Complexity

i+=2, i+=3, i+=10 be like: , , so after ignoring the constants the time complexity is still O(n).

addition, subtraction, everything we do with n is useless cause constants will be ignored forever.

Example:

, , n-10, still time complexity will be O(n) because 1/2, 2, -10 will be ignored.

Video 1-4; O(logN) Time Complexity

i\*=10, i\*77, n/=10, n/=99

When we use multiplication or division as update to iterate over anything that gives us time complexity O(logN). Generally it means tearing apart a problem into half or more to reduce the time complexity and making the problem smaller.

Example 1.1: **( i starting from 2)**

for (int i = 2; i <= n; i\*=2) {

cout << “hello world” << endl;

}  
Here suppose N is 1000(user input), lets calculate how the log came and how many times it will iterate.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of iterations | Equation  (you can ignore this line for for loop, it given just to understand how it works and to make an equation to prove logarithm time complexity of this loop) | Exponent | Value of  i | Condition(true)  i<=n |
| 1 | = 500 | = 1 | i = 2 | 2<=1000 |
| 2 | = 250 | = 4 | i = 4 | 4<=1000 |
| 3 | = 125 | = 8 | i = 8 | 8<=1000 |
| 4 | = 62 | = 16 | i = 16 | 16<=1000 |
| 5 | = 31 | = 32 | i = 32 | 32<=1000 |
| 6 | = 15 | = 64 | i = 64 | 64<=1000 |
| 7 | = 7 | = 128 | i = 128 | 128<=1000 |
| 8 | = 3 | = 256 | i = 256 | 256<=1000 |
| 9 | = 1 | = 512 | i = 512 | 512<=1000 |
| 10 | = 0 | = 512 | i = 1024 | false |

Ideal equation,

= 1

n =

log n = log

log n = k log 2

**k = log n** [ = 1 ]

So here time complexity becomes O(logN), k is the number of iteration it will take.

k =

k = 9 iterations

Hence, k = log n (proved)

Example 1.2: **(What if now I(me) start i from 1? i = 1)**

for (int i = 1; i <= n; i\*=2) {

cout << “hello world” << endl;

}  
Here suppose n is 1000(user input), lets calculate how the log came and how many times it will iterate.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of iterations | Equation  (you can ignore this line for for loop, it given just to understand how it works and to make an equation to prove logarithm time complexity of this loop) | Exponent | Value of  i | Condition(true)  i<=n |
| 1 | = 1000 | = 1 | i = 1 | 1<=1000 |
| 2 | = 500 | = 2 | i = 2 | 2<=1000 |
| 3 | = 250 | = 4 | i = 4 | 4<=1000 |
| 4 | = 125 | = 8 | i = 8 | 8<=1000 |
| 5 | = 62 | = 16 | i = 16 | 16<=1000 |
| 6 | = 31 | = 32 | i = 32 | 32<=1000 |
| 7 | = 15 | = 64 | i = 64 | 64<=1000 |
| 8 | = 7 | = 128 | i = 128 | 128<=1000 |
| 9 | = 3 | = 256 | i = 256 | 256<=1000 |
| 10 | = 1 | = 512 | i = 512 | 512<=1000 |
| 11 | = 0 | = 1024 | i = 1024 | false |

Ideal equation,

= 1

n =

log n = log

log n = k log 2

**k = log n** [ = 1 ]

So here time complexity becomes O(logN), k is the number of iteration it will take.

k =

k = 9 but we calculated 10 iterations right?

Now we have 10 iteration, which is large of k by 1, so we can write,

k = 10  
=► k = 9+1   
=► k = log n + 1;

So the time complexity is,

O(log n + 1)   
or  
 O(log n) + O(1)

But if we remember that we have to ignore all the constants values, right? The last iteration that happened we can’t ignore it but we have taken it’s time complexity as constant, So ultimately the new time complexity is,

O(log n)

Hence, k = log n (proved)

NOTE: We can achieve O(logN) in for, while, do while loop and recursive functions.

Example 2.1: **(starting with n>1)**

while (n > 1) {

sum += n%10;  
 n /= 10;

}

Here suppose user input is n = 123456789

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of iterations | Condition(true)  n>1 | Equation | Exponent | Value of  n |
| 1 | 123456789 > 1 | = 12345678 | = 10 | n = 12345678 |
| 2 | 12345678 > 1 | = 1234567 | = 100 | n = 1234567 |
| 3 | 1234567 > 1 | = 123456 | = 1000 | n = 123456 |
| 4 | 123456 > 1 | = 12345 | = 10000 | n = 12345 |
| 5 | 12345 > 1 | = 1234 | = 100000 | n = 1234 |
| 6 | 1234 > 1 | = 123 | = 1000000 | n = 123 |
| 7 | 123 > 1 | = 12 | = 10000000 | n = 12 |
| 8 | 12 > 1 | = 1 | = 100000000 | n = 1 |
| 9 | false | = 0 | = 1000000000 | n = 0 |

Ideal equation,

= 1

n =

log n = log

log n = k log 10

**k = log n** [ = 1 ]

So here time complexity becomes O(logN), k is the number of iteration it will take.

k =

k = 8 iterations

Hence, k = log n (proved)

যখন চক্রাকারে n = ১২৩৪৫৬৭৮৯ শর্ত সত্য হবার পরে সরাসরি ভিতরে ঢুকে আপডেটেশন বা মান বদলানো শুরু হয়, আমাদের কাজ একটি ভ্যালু থাকা পর্যন্তই সীমাবদ্ধ কারণ ১ টা ভ্যালু থাকা পর্যন্তই কাজ করা যাবে(যদি না কোনো আলাদা শর্ত দেয়া হয় যেমন এই ক্ষেত্রে ন > ১ শর্ত দেয়া হয়েছে, যার মানে এই দাঁড়ায় যে ন এর মান যতক্ষণ ১ এর চেয়ে বড় ততক্ষন চক্রাকারে কাজ হতে থাকবে), কোনো ভ্যালু না থাকলে কাজ করবো কি দিয়ে তাই না? (পূর্ব নির্ধারিত বা ডিফল্ট ন > ০)

Example 2.1: **(starting with n>0)**

while (n) {

sum += n%10;  
 n /= 10;

}

Here suppose user input is n = 123456789

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of iterations | Condition(true)  n>1 | Equation | Exponent | Value of  n |
| 1 | 123456789 > 0 | = 12345678 | = 10 | n = 12345678 |
| 2 | 12345678 > 0 | = 1234567 | = 100 | n = 1234567 |
| 3 | 1234567 > 0 | = 123456 | = 1000 | n = 123456 |
| 4 | 123456 > 0 | = 12345 | = 10000 | n = 12345 |
| 5 | 12345 > 0 | = 1234 | = 100000 | n = 1234 |
| 6 | 1234 > 0 | = 123 | = 1000000 | n = 123 |
| 7 | 123 > 0 | = 12 | = 10000000 | n = 12 |
| 8 | 12 > 0 | = 1 | = 100000000 | n = 1 |
| 9 | 1 > 0 | = 0 | = 1000000000 | n = 0 |
| 10 | false | = 0 | = 10000000000 | n = 0 |

Ideal equation,

= 1

n =

log n = log

log n = k log 10

**k = log n** [ = 1 ]

So here time complexity becomes O(logN), k is the number of iteration it will take.

k =

k = 8 but we calculated 9 iterations right?

Now we have 9 iteration, which is large of k by 1, so we can write,

k = 9  
=► k = 8+1   
=► k = log n + 1;

So the time complexity is,

O(log n + 1)   
or  
 O(log n) + O(1)

But if we remember that we have to ignore all the constants values, right? The last iteration that happened we can’t ignore it but we have taken it’s time complexity as constant, So ultimately the new time complexity is,

O(log n)

Hence, k = log n (proved)

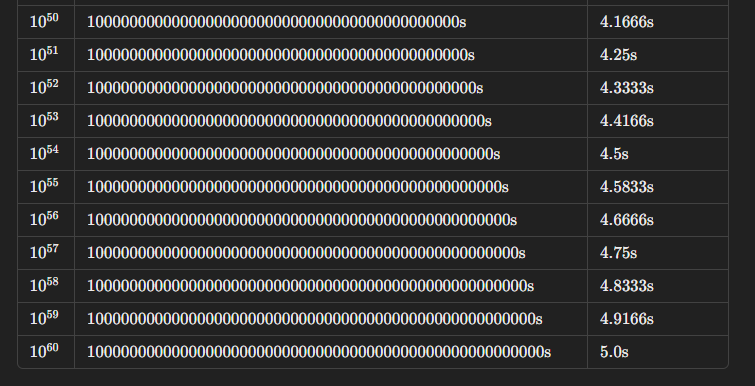
Example’s index summary of O(logN);  
Example 1.1 = for loop: initialized i = 2, O(log n)  
Example 1.2 = for loop: initialized i = 1, O(log n + 1) = O(log n)  
Example 2.1 = while loop: condition = n > 1, O(log n)  
Example 2.2 = while loop: condition = n > 0, O(log n + 1) = O(log n)

Comparing O(N) and O(logN):

If N time complexity takes steps;  
then, 1 time complexity takes steps;  
so, logN time complexity takes steps =► = 23 steps;  
  
again,

If steps takes 1 second;  
then, 1 step takes seconds;  
so, 23 steps takes seconds =► 0.0000023 or 2.3x seconds

|  |  |  |
| --- | --- | --- |
| Value | O(N) run time | O(logN) run time |
| N = | 0.0000001s | 0s |
| N = | 0.000001s | <1s |
| N = | 0.00001s | <1s |
| N = | 0.0001s | <1s |
| N = | 0.001s | <1s |
| N = | 0.01s | <1s |
| N = | 0.1s | <1s |
| N = | 1s | 0.0000023s (2325 nanoseconds) |
| N = | 10s | <1s |
| N = | 100s | <1s |
| N = | 1000s | <1s |
| N = | 10000s | 0.0000036s (3654 nanoseconds)<1s |
| N = | 100000s | 1s |
| N = | 1000000s | 1.0833s |
| N = | 10000000s | 1.1666s |
| N = | 100000000s | 1.25s |
| N = | 1000000000s | 1.3333s |
| N = | 10000000000s | 1.4166s |
| N = | 100000000000s | 1.5s |



Even if we take 60% time because 1 minute = 60 seconds, so (5x.0.6) = 3 seconds  
to iterate over steps.

On the other hand if we take 60% of steps to accurately calculate the steps in given time (x 0.6) = steps in 5 seconds which doesn’t sounds correct or accurate. So we should follow the time reducing technique.

If we think O(n) takes 1 second to perform steps, then O(log n) should do,

= 430107 times of steps ≈ 4xsteps in 1 second.

If we think O(n) takes 1 second to perform steps, then O(log n) should do,

= 50173 times of steps ≈ 5xsteps in 1 second.

(we first calculated steps in 1 seconds for O(logn) using O(n), then we calculated rest of the time of given steps using 1 second that we calculated from O(n)[base])

**This is an estimate and actual performance may vary based on clock speed, hardware and other factors.**

Video 1-4; O() Time Complexity