Algorithms: Time and Space complexity (Part 2)

Topic to be covered

Time Complexity

Space Complexity

Calculating Time and Space Complexity

MCQs

Practice Problems

#180DaysofPurpose

Relevel
by Unacademy

Time Complexity (30 mins)

Applying the asymptotic analysis to measure the time requirement of an algorithm as a function of input size is known as time complexity. For time complexity analysis we assume each individual instruction takes a constant amount of time let's say c.

We need to follow the general rule of time complexity which is running time is equal to the summation of all the running time of disconnected fragments.

Let's go through some of the examples to get a grip on how to calculate time complexity

For example, consider the following function where we need to search an element in a list:

```
function searchElement(list, N, element){
  for(let i = 0; i < N; i++){
    if(list[i] === element){
      return true;
    }
  }
  return false;
}
```

Here we are running the for loop N time which is the size of the list and in each iteration, we are performing one conditional check. If the first item of the list is equal to the search element the loop will execute one time, whereas if the list does not contain the search element then the loop will execute N times. So for this function based on the nature of input worst-case complexity is O(N) and the best case complexity is Ω(1)

Time Complexity (continued)

```javascript
function foo(n) { 
  let ans = 0;
  for (let i = 1; i < n; i++) {
    for (let j = 1; j < Math.log(i); j++) {
      ans += 1;
    }
  }
  console.log(ans);
}
```

T(N) = Number of times inner loop executes in total
= N * (log(N) * c1) + c2 //c1, c2 are the constant
= c1*NlogN + c2
O(T(N)) = O(c1NlogN + c2) = O(NlogN)

Time Complexity (continued)

```
function foo(n, m) { ■
  let a = 0;
  for (let i = 1; i < n; i++) {
    a += i;
  }
  let b = 0;
  for (let i = 1; i < m; i++) {
    b += i;
  }
}
```

Here we have two independent for loops one executing n times and the other executing m times. Hence O(T(n,m)) = O(n + m). Note that we don't know the relation between n, m, hence we cannot write O(T(n)) = O(n) or O(m). Because let's say m = 2^n. Then O(T(n,m)) = O(n) will be wrong answer.

Time Complexity (continued)

```javascript
function foo(n) { 🟩
  let ans = 0;
  for (let i = 1; i < n; i++) {
    for (let j = n; j > i; j--) {
      ans += 1;
    }
  }
  console.log(ans);
}
```

T(n) = Number of times inner loop executes in total
= n-1 + n-2 + ... + 1

Therefore O(T(n)) = O(n^2)

Time Complexity (continued)

```javascript
function foo(n) { ■
  for (let i = 2; i <= Math.sqrt(n); ++i) {
    if (n % i === 0) {
      return false;
    }
  }
  return true;
}
```

Here we can see that there is one loop that runs for √n-2 times. Hence **O(T(n)) = O(√n)**. Note that this program is checking whether the input number is prime or not.

Time Complexity (continued)

Let's analyze the time complexity of function foo:
First, let's analyze function boo:
We have one for loop running from m to 1,
with loop iteration variable reducing by half in every iteration.
Loop will execute until it hits the following condition, where k is the number of iterations:
=> m/2^k = 1
=> m = 2^k
=> log2(m) = k
Therefore the number of times the loop executes is log2(m).
In function foo we are basically calculating 2^(N+1) which is input to bar.
Therefore m = 2^(N+1) = O(2^n).
Therefore total complexity of bar in terms of n is O(log2(2^n)) = O(n).
Let T(n) denote the time complexity of the function. Then we can write following recurrence equation:
T(n) = T(n - 1) + O(n) + 1
       = T(n-2) + 2*O(n) + 2

       ...
       = T(n-k) + k*O(n) + k

```
function foo(n) {
  let a = 0;
  if (n == 0) {
    a = 2;
  } else {
    a = 2 * foo(n - 1);
  }
  bar(a);
  return a;
}
function bar(m) {
  for (let i = m; i > 1; i /= 2);
}
```

Relevel
by Unacademy

Time Complexity (continued)

We know that T(0) = 0, therefore n - k = 0.
T(n) = T(0) + n * O(n) + n
        = O(n^2 + n)
        = O(n)

Time Complexity (continued)

```javascript
function foo(n) {
  let k = 1;
  if (n && !(n & (n - 1))) {
    return n;
  }
  while (k < n) {
    k <<= 1;
  }
  return k;
}
```

This program checks if the input is of power of 2 or not. If it is not then calculate the next nearest power of 2. If you analyze the while loop for input n = 15 which has binary representation of 1111, we will shift n 3 times. Therefore the loop will execute 3 times = no of bits - 1. Since we are dealing with binary representation of numbers we know that a number n number of bits required to represent is O(logn). Therefore the time complexity of this function is O(logn).

Time Complexity (continued)

```
function foo(n) {
  let p = 0;
  let q = 0;
  for (let i = 0; i < n; i++) {
    p = 0;
    for (let j = n; j > 1; j = j / 2) {
      p++;
      for (let k = 1; k < p; k *= 2) {
        q++;
      }
    }
  }
}
```

Here we have three nested loops. You can observe that the innermost loop will have time complexity of O(logp). The second inner loop has a time complexity of O(logn). We can say p = logn. Therefore the total number of times the innermost loop will get executed is (log log n). Then we have the outermost loop that runs for n times. Therefore the overall time complexity is O(nlog log n).

Space Complexity (20 mins)

Applying the asymptotic analysis to measure the space requirement of an algorithm as a function of input size is known as space complexity. We measure the variables and data structures created by the program as a function of the input. Strictly speaking, space complexity also includes the size of the input to the function. But usually, we only care about what is known as auxiliary space complexity which does not include the size of the input.

For a recursive function, we look for two factors while calculating space complexity:
- Amount of new memory that gets consumed per function call
- The maximum function call stack depth

Time Complexity (30 mins)

For example, consider the following recursive function to print first N natural numbers:

```
function printNaturalNumbers(N) {
  if (N === 0) {
    return;
  }
  printNaturalNumbers(N - 1);
  console.log(N);
}
```

Here we are making N + 1 function calls that are present in the call stack space of the process, and in each call, we are using only constant space lets say c. Therefore space complexity of this program is **c*(N+1) = c*N + c = O(N)**, ignoring constant terms.

Note that in the example above in each function call we are only consuming constant space but since we are making N function call hence space complexity ends up O(N). In the case where we are creating additional data structures like arrays, matrices, hash tables, etc in our function, we need to account for those in our space complexity computation.

Time Complexity (30 mins)

Let's consider an iterative function:

```
function getNaturalNumbersList(N) {
  const list = [];
  for (let i = 1; i <= N; i++) {
    list.push(i);
  }
  return list;
}
```

Here we are running a for loop and in every iteration, we are creating a new variable and storing it in the list. Therefore the size of the list is equal to the number of times for loop executes which is equal to **N**.
Hence the space complexity of this program = **O(N + c) = O(N)**, where c is the constant space consumed loop iteration variable.

In the case where we are creating additional data structures like arrays, matrices, hash tables, etc in our function, we need to account for those in our space complexity computation.

#180DaysofPurpose

Relevel
by Unacademy

Calculating Time and Space Complexity (40 mins)

The time and space complexity of most of the programs depends on factors like
- How many times does a loop get executed in the case of an iterative function?
- How much extra space is being consumed due to data structures created inside the function?
- What is the maximum depth of the recursive function call stack?

But for algorithms that fall in the category of Divide and Conquer we need to arrive at a recurrence relation of complexity as a function of input and then apply Master's Theorem to perform proper time and space complexity analysis.
Master's Theorem will be covered in the next lecture but till then we can look at a few examples on how to arrive at time and space complexity.

Calculating Time and Space Complexity (continued)

Let's consider a **recursive function** to return sum of all the elements in the list

```javascript
function sumList(list, N, index) {
  if (index === N) {
    return 0;
  }
  const sum = sumList(list, N, index + 1);
  return sum + list[index];
}
```

Calculating Time and Space Complexity (continued)

Let **T(N) represent time complexity of the function**, where N is the size of the list.
Then we can write the following recurrence relation:
=> **T(N) = T(N-1) + c** //c is a constant representing time taken by operations like conditional checks and addtion.
=> T(N-1) = T(N-2) + c
=> T(N-2) = T(N-3) + c

...
=> T(0) = c
By performing back substitution we can write:
=> T(N) = T(N-2) + 2*c
=> T(N) = T(N-3) + 3*c

...
=> T(N) = T(0) + N*c = c*N
Therefore we can say **O(T(N)) = O(c*N) = O(N)**

Let **S(N) represent the space complexity of the function**, where N is size of the list.
Then we can write the following recurrence relation:
=> **S(N) = S(N-1) + c** //where c is the constant space used for variable like sum
This is same as in the case of time complexity and therefore we can conclude **O(S(N)) = O(N)**.

Therefore for the recursive function both time and space complexity it O(N).

Calculating Time and Space Complexity (continued)

Now let's consider **iterative** version of the same function:

```javascript
function sumList(list, N) {
  let sum = 0;
  for (let i = 0; i < N; i++) {
    sum += list[i];
  }
  return sum;
}
```

Calculating Time and Space Complexity (continued)

Lets calculate time complexity. Here we can see that we have a **for loop** that runs N times and performs **constant time operation in each run**. Therefore time complexity of the function
T(N) = time taken by the for loop + constant
T(N) = c1*N + c2 //c1 and c2 are constants
Therefore **O(T(N)) = O(c1*N + c2) = O(N)**

Lets calculate space complexity. We can observe that only extra space needed by the function is for **variable sum, and iteration variable i**, which is **constant space with respect to input**. Therefore space complexity
=> S(N) = c
=> **O(S(N)) = O(c) = O(1)**

If we now compare the above two function which returns the sum of all element of a list we can say that the iterative version is better than recursive because its space complexity is less even though the time complexity is same for both.

For any practical purpose while comparing between two algorithm performing same task we prefer the one with lower time and space complexity.

Calculating Time and Space Complexity (continued)

Let's consider another example:

```javascript
function foo(n) { 🟩
  let count = 0;
  for (let i = n; i > 0; i /= 2) {
    for (let j = 0; j < i; j++) {
      count += 1;
    }
  }
  return count;
}
```

Calculating Time and Space Complexity (continued)

Lets analyse the time complexity of this function.
For i = n, innermost loop runs n times
For i = n/2 innermost loop runs n/2 times

Therefore total number of times the inner loop executes is
= n + n/2 + n/4 + ... + 1
= n(1 + ½ + ¼ + ... + 1/n)
= O(n) // summation of the above series tends to 2 which is a constant
Therefore T(n) = O(n)

Lets analyse the space complexity of this function.
For the code it is evident that we are not creating any new data structure. Only few local variables are created which consumes constant space. Therefore S(n) = O(1)

Calculating Time and Space Complexity (continued)

Let's consider one of the most classical programs to compute Fibonacci Number

```
function fibonacci(n) {
  if (n < 2) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Let's see the behavior of the program on some of the inputs

fibonacci(0) = 0;
fibonacci(1) = 1;
fibonacci(2) = fibonacci(1) + fibonacci(0) = 1;
fibonacci(3) = fibonacci(2) + fibonacci(1) = 2;
fibonacci(4) = fibonacci(3) + fibonacci(2) = 3;
fibonacci(5) = fibonacci(4) + fibonacci(3) = 5;
fibonacci(6) = fibonacci(5) + fibonacci(4) = 8;

Calculating Time and Space Complexity (continued)

This can be visualized as a recursive tree given below:
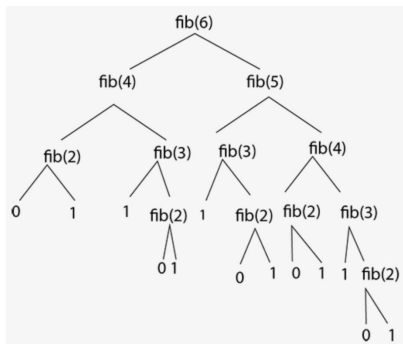
Let's analyze the **space complexity** of the program.
For an input N, from the above diagram, you can observe that the depth of the recursive tree is N. This means in the worst case there will be N number of active function call stacks present in the process. Also in each function call, we are only consuming constant space. Therefore the space complexity in the worst case will be O(N).

Relevel
by Unacademy

Calculating Time and Space Complexity (continued)

Let's analyze the time complexity of the program.

Let T(n) denote the time complexity of the function. Then we can write the recurrence relation as :
T(n) = T(n-1) + T(n-2) + 1 for n > 1, and
T(n) = 0 for n <= 1

Let's solve the recurrence equation via back substitution method:
T(n) = T(n-1) + T(n-2) + 1
    = T(n-1) + T(n-1) + 1 //here we have assumed T(n-1) = T(n-2) which is not a wrong assumption in our case since we anyways want to find the upper bound of the function. Plus this will help us in simplifying the recurrence equation.
T(n) = 2*T(n-1) + 1   ->(1)
T(n-1) = 2*T(n-2) + 1   ->(2)

Back substituting eq (2) in eq (1) we get

T(n)   = 2 * [2 * T(n-2) + 1] + 1
       = 4*T(n-2) + 3
       = 8*T(n-3) + 7
       = 16*T(n-4) + 15
       ...

Therefore we can write

T(n) = 2^kT(n-k) + (2^k - 1)

We know that at T(0), n - k = 0, therefore

T(n) = 2^n*T(0) + 2^n - 1
       = 2^n + 2^n -1
       = O(2^n)

Therefore we can conclude that the time complexity of the above Fibonacci program is exponential.

MCQ

1. Find time complexity of following function:

```javascript
function foo(n) {
  let ans = 0;
  while (n > 0) {
    ans += n;
    n /= 2;
  }
  console.log(ans);
}
```

A. O(log2n)

B: O(n)

C: O(1)

D: O(nlog2n)

Answer: A

2. Find time complexity of following function:

```javascript
function foo(n) {
  let ans = 0;
  while (n > 0) {
    ans += n % 10;
    n /= 10;
  }
  console.log(ans);
}
```

A. O(log10n)
B: O(n)
C: O(1)
D: O(nlog10n)
Answer: A

3. Find time complexity of following function:

```javascript
function foo(n) {
  let ans = 0;
  for (let i = 1; i < n; i++) {
    for (let j = i; j <= n; j += i) {
      ans += 1;
    }
  }
  console.log(ans);
}
```

A. O(n^2)

B: O(n^3)

C: O(n^2logn)

D: O(nlogn)

Answer: D

4. What is the time complexity of fun()?

```
function fun (n)
{
    let temp = 0;
    for (let i = n; i>0; i /=2)
        for (let j = 0; j>i; j++)
            temp += 1;
        console.log(temp);
}
console.log(n);
```

A. O(n^2)

B. O(nLogn)

C. O(n)

D. O(nLognLogn)

Answer: C

O(n) For an input integer n, the innermost statement of fun() is executed following times. n + n/2 + n/4 + ... 1 So time complexity T(n) can be written as T(n) = O(n + n/2 + n/4 + ...1) = O(n) The value of count is also n + n/2 + n/4 + ... +1

Relevel
by Unacademy

5. What is the time complexity of fun() ?

```
function fun(n)
{
     let temp= 0;
     for (let i=0; i<n; i++)
         for(let j=i; j>0; j- -)
                temp= temp + 1;
     console.log(temp);
}
```

A. Theta(n)

B. Theta(n^2)

C. Theta(n*Logn)

D. Theta(nLognLogn)

Answer: B (Theta(n^2))

The time complexity can be calculated by counting the number of times the expression "Count = count + 1;" is executed. The expression is executed 0+1+2+3+4+...+(n-1) times. Time complexity = Theta(0+1+2+3+...+n-1) = Theta(n*(n-1)/2)=Theta(n^2)

Practice Problems

1. Write a program to compute the Nth prime number using the concept of Sieve Of Eratosthenes and derive its worst case time and space complexity.
2. Write a program to multiply two 2D matrices and derive its worst case time and space complexity.

Teaser for Upcoming Class

- Complexity Notations: Omega, Theta
- Creating and solving recurrence relations
- Master's Theorem

Thank You!