Test Yourself

Analysis of Complexity

Dov Kruger

Department of Electrical and Computer Engineering
Stevens Institute of Technology

September 6, 2022

Complexity

Complexity is a measure of how much time or space is used by a program as a function of the size of the problem

Big-O $O()$ worst case, the most time (or space) the algorithm will take

Omega $\Omega()$ best case (the shortest possible time to complete)

For the special case where $O() = \Omega()$ theta $\theta()$

Later, we will also talk about **amortized complexity**, the average case

Worst case and Best case: Finding a number in a List

$1, 3, 4, 19, 20, 2, 16, 5, \ldots, n = 10^6$

how long to determine whether the number 9 is in the list? $O()$

or for this list?

$9, 3, 4, 19, 20, 2, 16, 5, \ldots, n = 10^6$ $\Omega()$

Worst case and Best case: Sum a List

$1 + 4 + 6 + 19 + 20 + ... \; + n = 10^6$

Q: How long to compute the sum of the list? $O()$

Q: Is there any way to end early? $\Omega()$

Worst case and Best case: Sum a List

$1 + 4 + 6 + 19 + 20 + ... + n = 10^6$

Q: How long to compute the sum of the list? $O()$

Q: Is there any way to end early? $\Omega()$

NO! The only way to compute the sum is to add all the numbers

Because $O(n) = \Omega(n)$ summing a list is $\theta(n)$

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Common Misconception/Mistake for $\Omega()$

Remember that complexity is the asymptotic behavior **as the problem grows**

- You just saw the problem of summing a list on the last page
- Many people understand it, but then sometimes, will do the following

sum(list) $O(n)$ $\Omega(1)$ when $n = 1$

No! When $n = 1$, the problem is small, of course complexity is 1

- Question: When n is BIG, is there any way of ending early?
- Answer: No. To compute the sum, you must sum ALL THE NUMBERS

Therefore $\Omega(n)$ not $\Omega(1)$

moral: Saying $\Omega(1)$ when $n = 1$ is meaningless

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY

$O()$ Formal Derivation

Big-O is formally defined as
$f(n) = O(g(n))$ means there exists a constant c such that
$f(n) \leq cg(n)$ for some n

Examples:

$f(n) = 2n \quad O(?)$ pick $c = 3, 2n \leq cn$, therefore $O(n)$
$f(n) = .00001n^2$ pick $c = 1, .00001n^2 \leq cn^2$ therefore $O(n^2)$

In other words, the constant in front of the term does not matter

$O()$ Slightly More Complicated Example

$g(n) = .01n^3 + 5n^2 + 10^9$

$O(g(n)) =?$

All we have to do is pick c bigger than $.01$
As n grows, n^3 will become the biggest term very quickly

suppose $c = 1,\ cn^3 > g(n)$ for some n?

Of course: n=1 no, since .01(1)+5(1)+109 the constant is big
At $n = 1000,\ .01 * (1000)^3$ is already $10,000,000$.
At $n = 10000,\ 1n^3$ is already bigger than $g(n)$

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

$O()$ Informal Derivation

Less formally: as n grows, the highest power of n is dominant, constants are irrelevant

Examples:

$O(n+5)$ as n grows (think $n = 10^6$) the 5 is insignificant $O(n)$
$O(3n) = \quad O(n)$
$O(.001n) = \quad O(n)$
$O(n^2 + n)$ as n grows, the n^2 term grows much faster.
n^2 (the leading term) dominates therefore $O(n^2)$
$O(5000n + .00001n^3 + \sqrt{n}) = O(n^3)$

All Logs have a Constant Factor

For the special case of $O(\log n)$ it does not matter whether it is

- $\log_2 n$
- $\log_3 n$
- Any other base

The reason is that all logs are related by a constant

$$\log_2 n = c\ \log_3 n, \qquad c = \log_2 3$$

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY

$O()$ for Various Functions

$$
\begin{array}{ll}
g(n) = n^3 + 1000000n^2 & O() \\
g(n) = n + n & O() \\
g(n) = logn + n & O() \\
g(n) = (n+n)^2 & O() \\
g(n) = 4n^3 + 1000000n^2 & O() \\
g(n) = 4n^4 + 1000000n^3 & O() \\
g(n) = 4n^{4/3} + 1000000n & O() \\
g(n) = 4\sqrt{n} + log\ n & O()
\end{array}
$$

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY

$O()$ for Various Functions

$g(n) = n^3 + 1000000n^2$	$O(n^3)$
$g(n) = n + n$	$O(n)$
$g(n) = logn + n$	$O(n)$
$g(n) = (n + n)^2$	$(2n)^2 = 4n^2 = O(n^2)$
$g(n) = 4n^4 + 1000000n3$	$O(n^4)$
$g(n) = 4n^{4/3} + 1000000n$	$O(n^{4/3})$
$g(n) = 4\sqrt{n} + log\ n$	$O(\sqrt{n})$

Test Yourself

How Functions Grow Asymptotically (Polynomials)

As n grows, n^2 is obviously a lot worse. And n^3 is even worse
Try to find algorithms that are as low as possible on the complexity scale

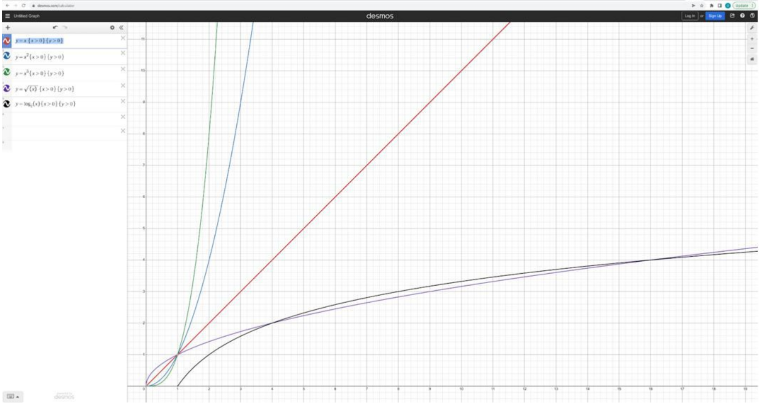
n	n^2	n^3
1	1	1
10	100	1000
100	10^4	10^6
1000	10^6	10^9
10^6	10^{12}	10^{18}

How Functions Grow (Log and Square Root)

n	$log_2 n$	$\sqrt{n}$
1	0	1
10	3.3	3.3
100	6.7	10
1000	9.9	33
10^6	20	1000
10^9	30	$30,000$
10^{12}	40	10^6

Graph Comparing O() Functions

https://www.desmos.com/calculator/nk3rfmfvle



Key Skill: Reading Code and Analyzing Complexity

It is vital to be able to read code or pseudocode

- Analyze complexity of current code
- Determine if possible to improve?

We will analyze the following programming features

- Loops
- Loops Containing if Statements
- Sequential loops
- Nested loops
- Recursive Functions

Complexity of Loops

Pseudocode or real programming language? In this course we don't care! You should be able to state your algorithm or analyze either

for $i \leftarrow 1$ to n $\quad //O() \quad \Omega()$

...

end

```
for (int i = 0; i < n+1; i++) {//O()
}

for (int i = 1; i <= n+500; i++) {//O()
}

for (int i = 17; i < 1000*n; i += 3) {//O()
}
```

Complexity of Loops - Answers

Pseudocode or real programming language? In this course we donât care! You should be able to state your algorithm or analyze either

for $i \leftarrow 1$ to n $\quad //O(n) \quad \Omega(n)$

...

end

```
for (int i = 0; i < n+1; i++) {//O(n)
}

for (int i = 1; i <= n+500; i++) {//O(n)
}

for (int i = 17; i < 1000*n; i += 3) {//O(n)
}
```

Nonlinear Loops

```
for (int i = 1; i <= n; i *= 2) { //O()

}

for (int i = 1; i <= n; i *= 3) { //O()

}

for (int i = 1; i <= n; i = i * 2 + 3) { //O()

}
```

Nonlinear Loops – answers

```
for (int i = 1; i <= n; i *= 2) {//O(log n)
}
for (int i = 1; i <= n; i *= 3) {// O(log n)
}
for (int i = 1; i <= n; i = i * 2 + 3){//O(log n)
}
```

Logarithmic Complexity

Which is faster?

$n = 10^6 \quad 1, 2, 4, 8, 16, 32, 64 \ldots$

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

$n = 10^6 \quad 1, 3, 9, 27, 81, \ \ldots$

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

Did you know? All logs differ by only a constant factor

$\log_2(n) = c \ \log_3 n$

c=

Complexity of Loops Containing if statements

array $\leftarrow [9, 1, 2, 3, 6, ...]$ list has length n
complexity to find a number in the list?

```
linear_search(array, target)
  for i = 0 to length(array) − 1 //O(?)
    if array[i] == target      // O(?)
      return i
    end
  end
  return −1
end
```

Complexity of Loops Containing if statements: Answers

array $\leftarrow [9, 1, 2, 3, 6, ...]$ list has length n
complexity to find a number in the list?

```
linear_search(array, target)
  for i = 0 to length(array) − 1 //O(n)
    if array[i] == target        //
```

Test Yourself

Try to analyze the code snippets and determine the complexity

Complexity of Sequential and Nested Loops

```
// O(?)
for (int i = 0; i < n; i++) {          //O()
}
for (int i = 0; i < n; i++) {          //O()
}


//O()
for (int i = 0; i < n; i++) {          //O()
   for (int j = 1; j <=n; j++) {       //O()
   }
}
```

Complexity of Sequential and Nested Loops, answers

```
// n+n = 2n = O(n)
for (int i = 0; i < n; i++) {          //O(n)
}
for (int i = 0; i < n; i++) {          //O(n)
}


//O(n^2)
for (int i = 0; i < n; i++) {          //O(n)
  for (int j = 1; j <=n; j++) {        //O(n)
  }
}
```

Complexity of Nested Loops, part 2

In this case notice the inner loop depends on the outer one

```
for (int i = 0; i <= n; i++) { // O(?) outer loop
    for (int j = 1; j <= i; j++) { // average = n/2
    }
}
```

First time inner loop executes 1, second time 2, then $3, 4, \ ...n$
Total computation $= 1 + 2 + 3 + \ ... \ + n = ?$

Complexity of Nested Loops, part 2, answer

In this case notice the inner loop depends on the outer one

```
for (int i = 0; i <= n; i++) { // O(n) outer loop
    for (int j = 1; j <= i; j++) { // average = n/2
    }
}
```

First time inner loop executes 1, second time 2, then $3, 4,\ \ldots n$
Total computation $= 1 + 2 + 3 +\ \ldots\ + n = n(n+1)/2 = O(n^2)$

Nested Loops, part 3

What is the total complexity of this nested loop?

```
//O(?)
for (i = 1; i <= n; i++) { // O()
  for (j = 1; j < i; j *= 2) { // O()
  }
}
```

Nested Loops, part 3 (answer)

What is the total complexity of this nested loop?

```
//O(n log n)
for (i = 1; i <= n; i++) { // O(n)
   for (j = 1; j < i; j *= 2) { // O(log n)
   }
}
```

Nested Loops, part 4

This time, the outer loop is logarithmic, and the inner one goes up to the outer variable

Surprise! **It is not the same result**

```
//O(?)
for (i = 1; i <= n; i *= 2) { //O()
  for (j = 1; j <= i; j++) { //O()
  }
}
```

Nested Loops, part 4 (answer)

This time, the outer loop is logarithmic, and the inner one goes up to the outer variable

Surprise! **It is not the same result**

```
//2n = O(n)
for (i = 1; i <= n; i *= 2) { //O(log n)
  for (j = 1; j <= i; j ++) { //O() 1 + 2 + 4 + 8 + ... n
  }
}
```

Deriving Sum of Powers of 2

$1 + 2 + 4 + 8 + \ \dots \ + n/2 + n = ?$

Informally:

$1 + 2 + 4 + 8 + 16 + 32 + 64 = 127(2n - 1)$
therefore $O(2n) = O(n)$

Formally:

[equation for google doc here. !@$!@ google does not do equations in slides]

Memorize two Equations

Summary: you don't need to memorize too many equations in this course, but you do need these two

$$\Sigma_{i=1}^{n} i = n(n+1)/2 = O(n^2)$$

$$\Sigma_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{n/2} + \frac{1}{n} = 2$$

Test Yourself

Analyze these real code snippets and try to determine complexity

Recursion

A recursive function is one that

- Calls itself
- Has a termination or base-case condition

Example:

```
factorial(n)
  if n <= 0
    return 1 // termination or base case
  end
  return n * factorial(n-1)     // recursive rule
end
```

Recursion Uses a Stack

Factorial example is tail recursion and is equivalent to a loop

Implemented using a stack

factorial(5)=5*factorial(4)	$5 * 24 = 120$
factorial(4)=4*factorial(3)	$4 * 6 = 24$
factorial(3)=3*factorial(2)	$3 * 2 = 6$
factorial(2)=2*factorial(1)	$2 * 1 = 2$
factorial(1)=1*factorial(0)	$1 * 1 = 1$
factorial(0)=1(base case)	

Tail Recursion Complexity

The factorial example is called tail recursion because the last act of the function is to call itself

Complexity: $O(n) + O(n) = O(2n) = O(n)$

Tail recursion can be automatically turned into a loop

For example, C++ will turn the previous code into the equivalent of:

```
int prod = 1;
while (n > 0)
  prod *= n--;
```

The loop has the same complexity, but a much lower constant

Explosive Recursion: Multiple Calls

Tail recursion like factorial is efficient

Unfortunately recursive functions that call themselves multiple times are exponential

Example: Fibonacci

The fibonacci series starts with 1,1.

Each new number is the sum of the previous two.

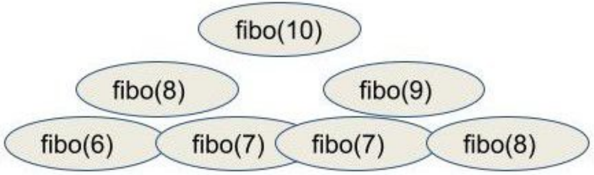
$1, 1, 2, 3, 5, 8, 13, 21, 34, 55,\ \ldots$

Fibonacci, Iteratively

The most direct way to compute the Fibonacci numbers is with a loop:

```
uint64_t fibo(uint64_t n) {
  uint64_t a = 1, b = 1, c;
  for (uint64_t i = 0; i < n; i++) { //O(n)
    c = a + b;
    a = b;
    b = c;
  }
  return c;
}
```

Fibonacci, Recursive



A recursive definition of fibonacci is simple, but inefficient:

```
uint64_t fibo(uint32_t n) {
  if (n <= 2)
    return 1;
  return fibo(n-1) + fibo(n-2); //O(2n)
}
```

Recursive Exponential Explosions

Like a chain reaction, a recursive function that calls itself multiple times is exponential

In this case, fibo called itself two times, so the cost is 2n

This means that fibo(10) is not 10 times worse than fibo(1), it is $2^{10} = 1024$ times worse

fibo(20) $=$ 1024 times the work of fibo(10)
fibo(30) $=$ 1024 times the work of fibo(20)...

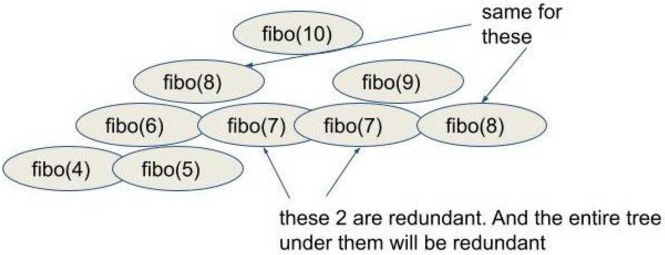
STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Dynamic Programming: Turn $O(2^n) \rightarrow O(n)$

It is possible to solve fibonacci in O(n)

Dynamic programming is a general technique to make exponential recursion more efficient by remembering previous answers.

Also called **memoization**



Dynamic Programming: Never Recompute an Answer

```
uint64_t fibo(uint32_t n) {
  static uint64_t memo[400] = {1, 1}; // all zero to start
  if (memo[n] != 0)
    return memo[n];
  // store each new answer before returning
  return memo[n] = fibo(n-2) + fibo(n-1); // O(n)
}
```

Test Yourself