31st August 2020

Studio 3
Orders of Growth, Higher-Order Functions and Scope of Names

CS1101S AY20/21 SEM 1

Studio 03A

Chen Xihao
Year 2 Computer Science

chenxihao@u.nus.edu
@BooleanValue

Studio X
Agenda

- Admin and announcements
- Recap:
  - Order of growth
  - Higher order functions
  - Scope of names
- Studio sheets

Admin

Admin Stuff

Reading Assessment 1

- RA1 is <u>this friday during lecture slot!!!</u>
- Past year papers on LumiNUS
- All MCQs
- Don’t get rekt

Admin Stuff
Asking Questions

- Ask in the studio telegram group first, or message your friends
- Refer to piazza, many questions have been answered already
- Message me as <u>LAST</u> resort
  - Please don’t message me past 2300hrs
- Asking the right questions is an important skill!

Admin Stuff

Mission - Curve Introduction

- Use of `connect_rigidly`, `transform`:
    - Challenge yourself to not use these functions!
    - Don’t need to tell me, just make sure you understand

Recap: Orders of Growth

Recap

Orders of Growth - Terminology

- Time complexity, space complexity (resources needed)
- Described with: Big Omega, Big Theta, Big Oh
- Constant, Logarithmic, Linear, “Linearithmic”, Quadratic, Exponential
- Efficiency

i don’t think this is an actual word

Future mods:

- CS2040S, CS3230, CS3233 (not for the faint hearted)

Recap

Orders of Growth - Definition

- “Big Oh” O( ): Upper bound
  - Most common, used in analysing worse case scenarios
- “Big Theta” Θ( ): Tight bound
  - Most useful, difficult to compute (or near impossible)
- “Big Omega” Ω( ): Lower bound
  - Not useful in most scenarios.

Recap

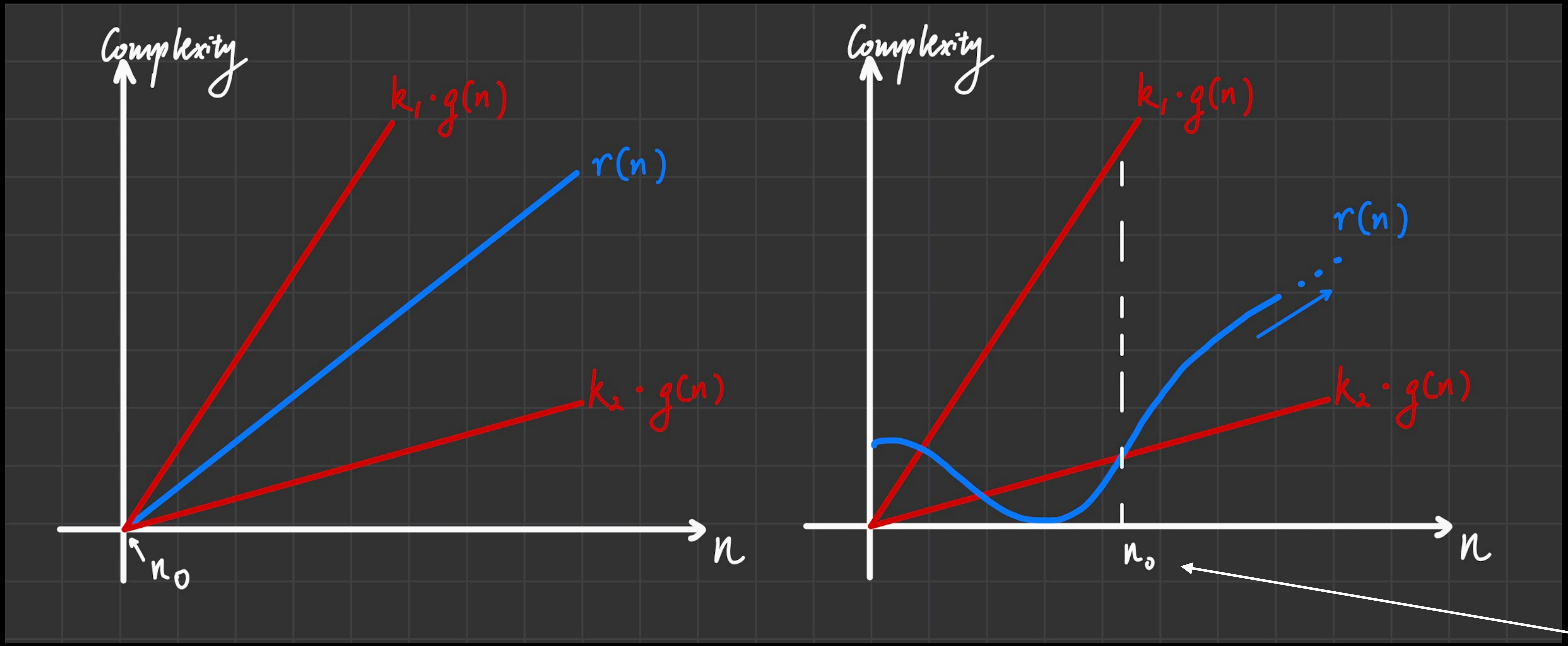
Orders of Growth - Definition

- Let n denote the size of the problem, and let $r(n)$ denote the resource needed solving the problem of size n.
- **Big Theta:**
  - the function r has order of growth $\Theta(g(n))$, if there are positive constants k1 and k2,
  - such that $k1 \cdot g(n) \leq r(n) \leq k2 \cdot g(n)$ for any sufficiently large value of n.
- **Big Oh:**
  - the function r has order of growth $O(g (n))$ if there is a positive constant k,
  - such that $r(n) \leq k \cdot g(n)$ for any sufficiently large value of n.
- **Big Omega:**
  - the function r has order of growth $\Omega(g (n))$ if there is a positive constant k,
  - such that $k \cdot g(n) \leq r(n)$ for any sufficiently large value of n.

huh?????

Recap

Orders of Growth - Big Theta (Tight Bound)

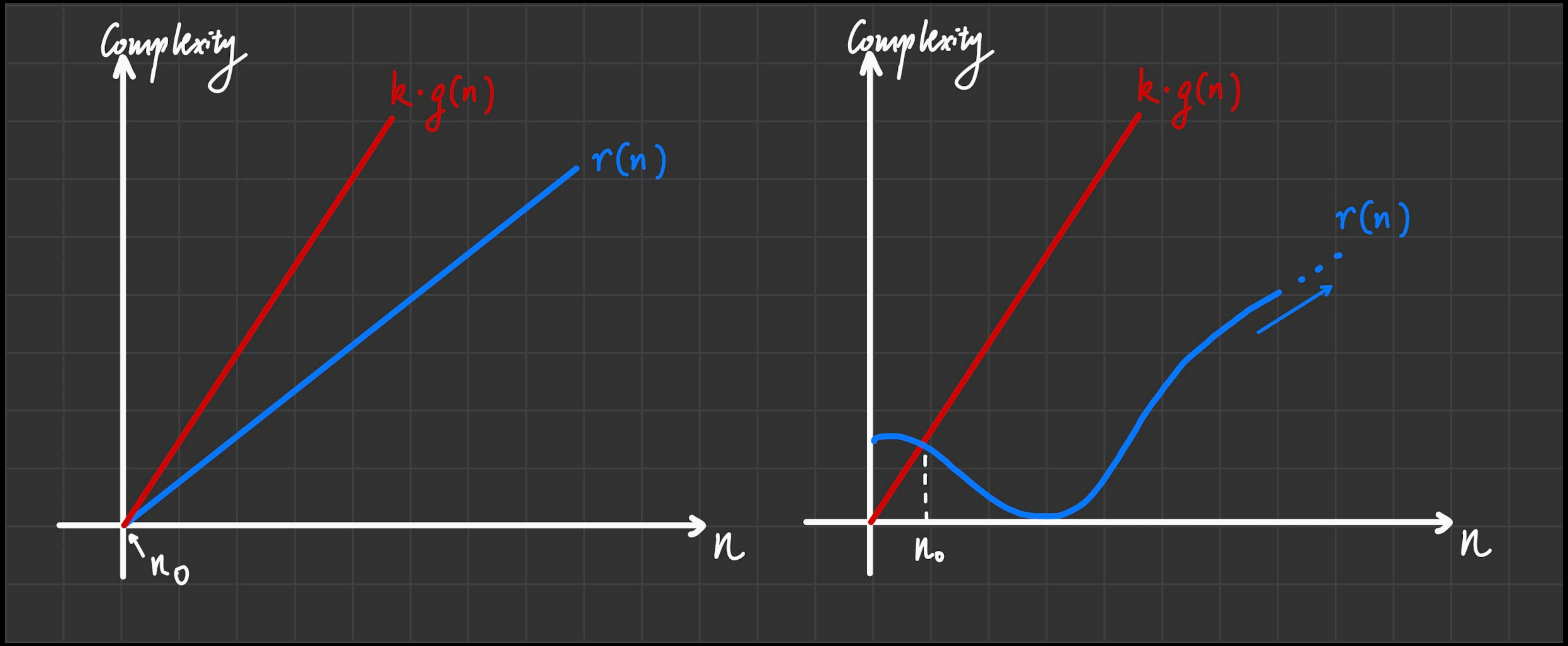
- Assume *g(n)* = *n*
- 

Hence, the functions have orders of growth *Θ(g(n))* for n ≥ n0
More specifically, they have orders of growth *Θ(n)*

recall:
"for any sufficiently large values of n", this only applies for n >= n0

Recap

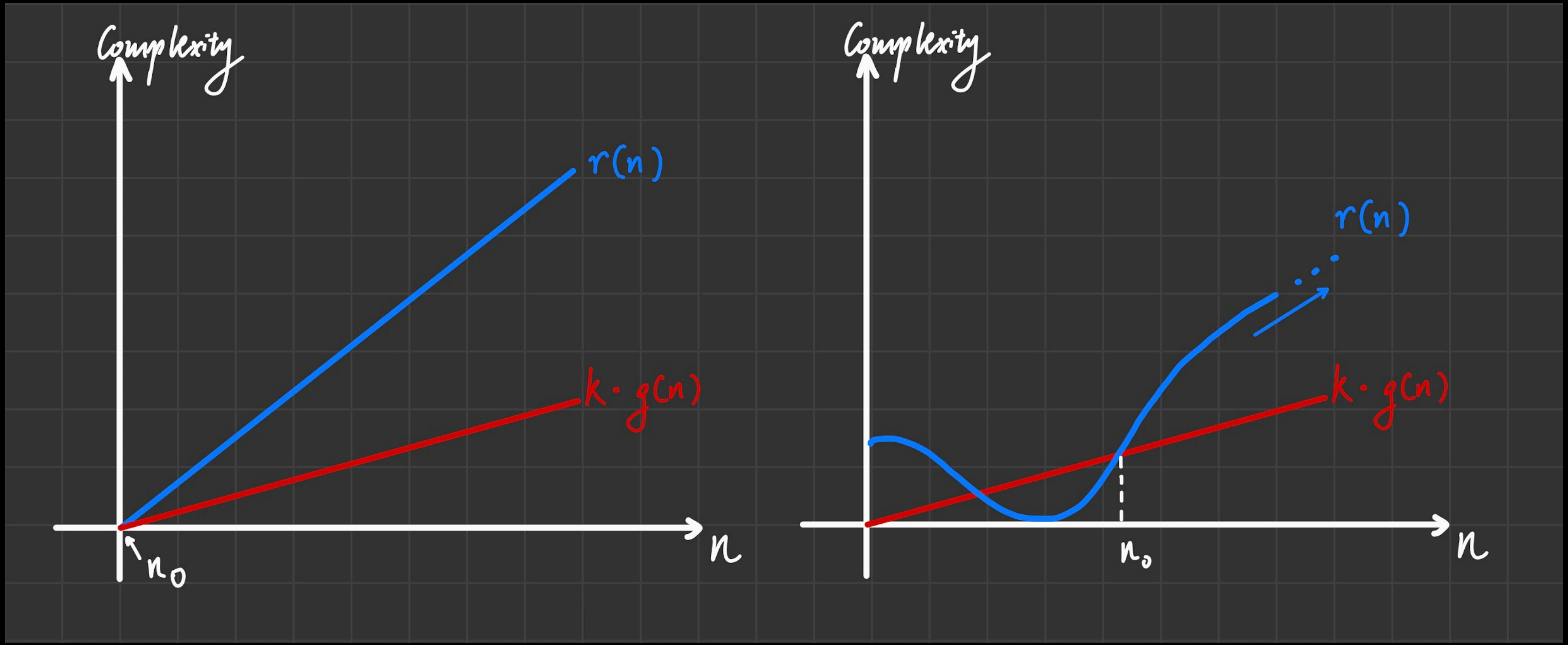
Orders of Growth - Big Oh (Upper Bound)

- Assume $g(n) = n$
- 

Hence, the functions have orders of growth *O(g(n))* for n ≥ n0
More specifically, they have orders of growth *O(n)*

Recap

Orders of Growth - Big Omega (Lower Bound)

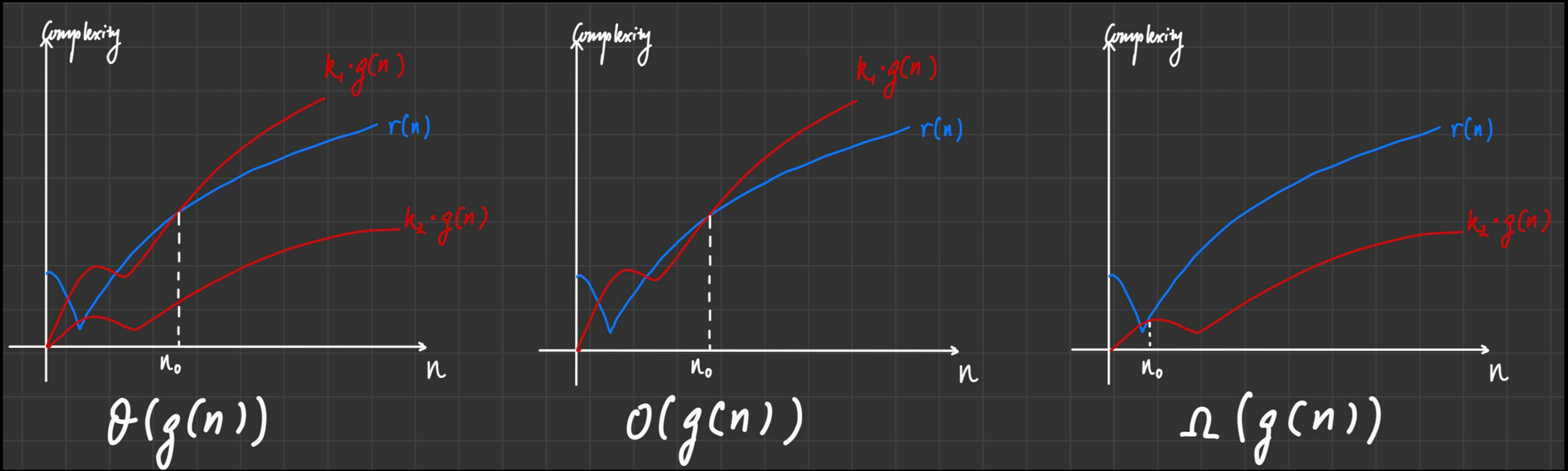
- Assume $g(n) = n$
- 

Hence, the functions have orders of growth $\Omega(\, g(n) \,)$ for n ≥ n0
More specifically, they have orders of growth $\Omega(\, n \,)$

Recap

Orders of Growth - in General

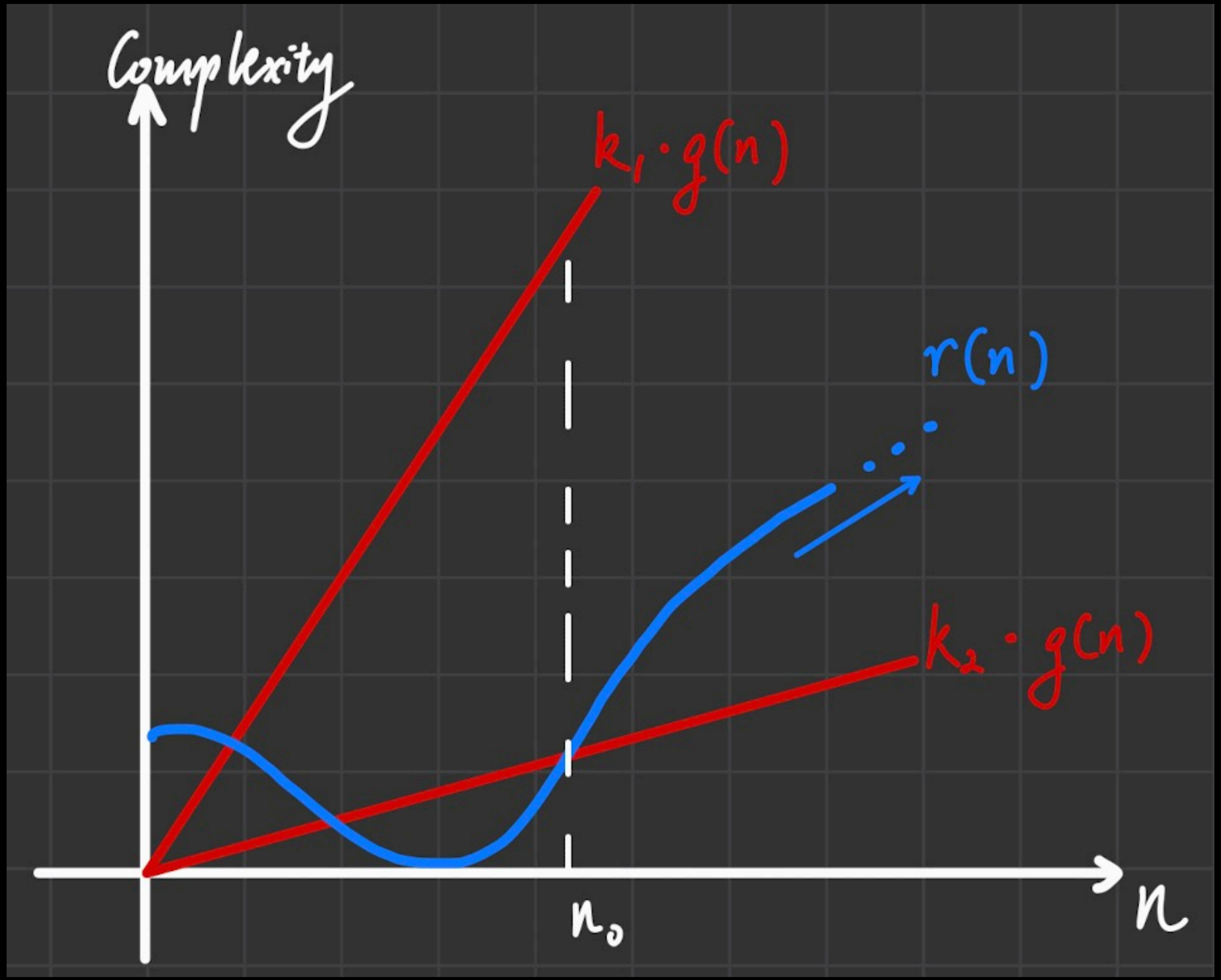
- *g(n)* and *r(n)* can be all sorts of funny functions!
- O, Ω, Θ stand as long as there's some function *g(n)* that binds *r(n)* for *n* ≥ *n0*



Recap

Orders of Growth - Why Do We Use Them?

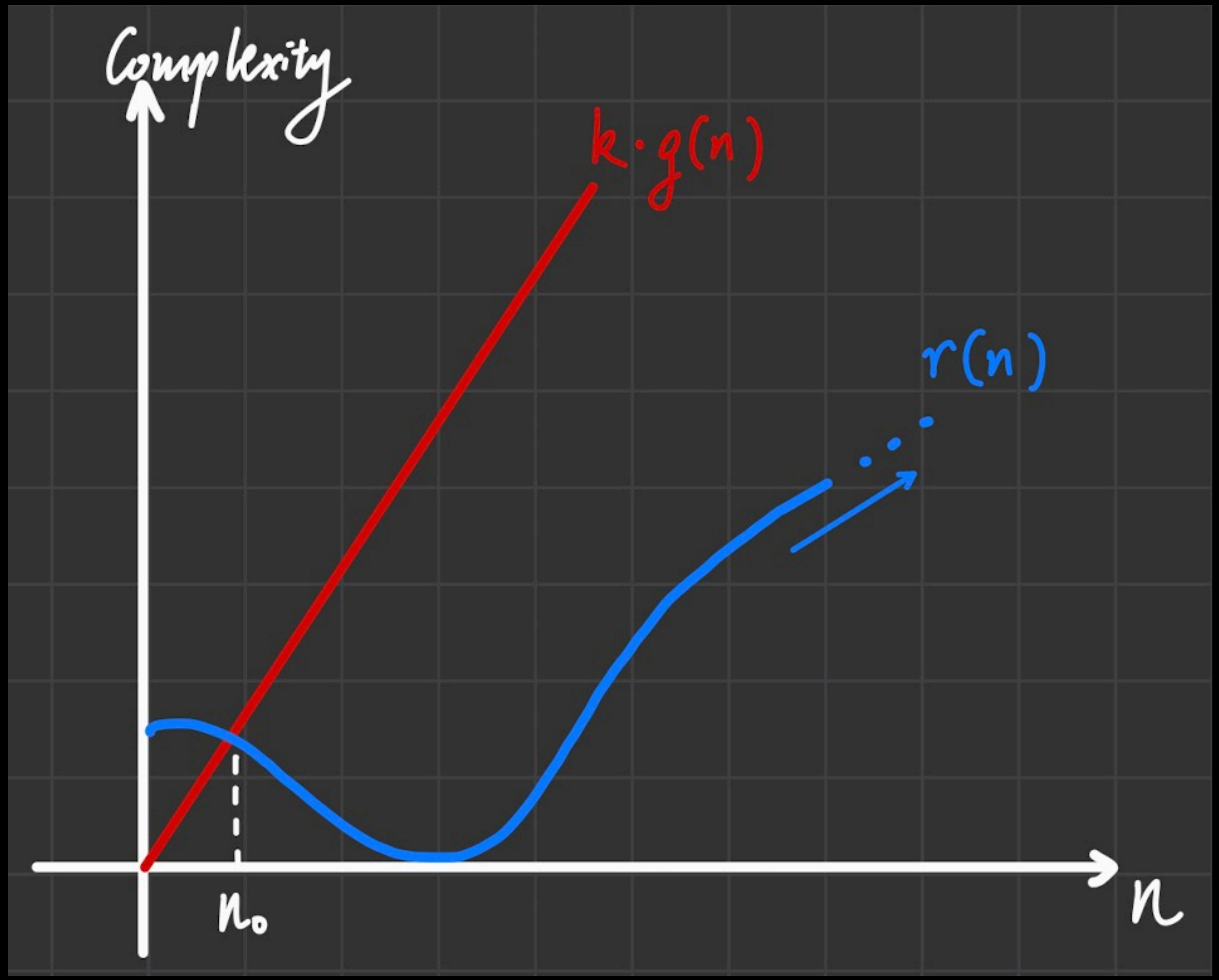
- Why <u>Big Theta</u>?
  - Tightest bound we can find for the algorithm's complexity
  - Represents the complexity most accurately



Recap

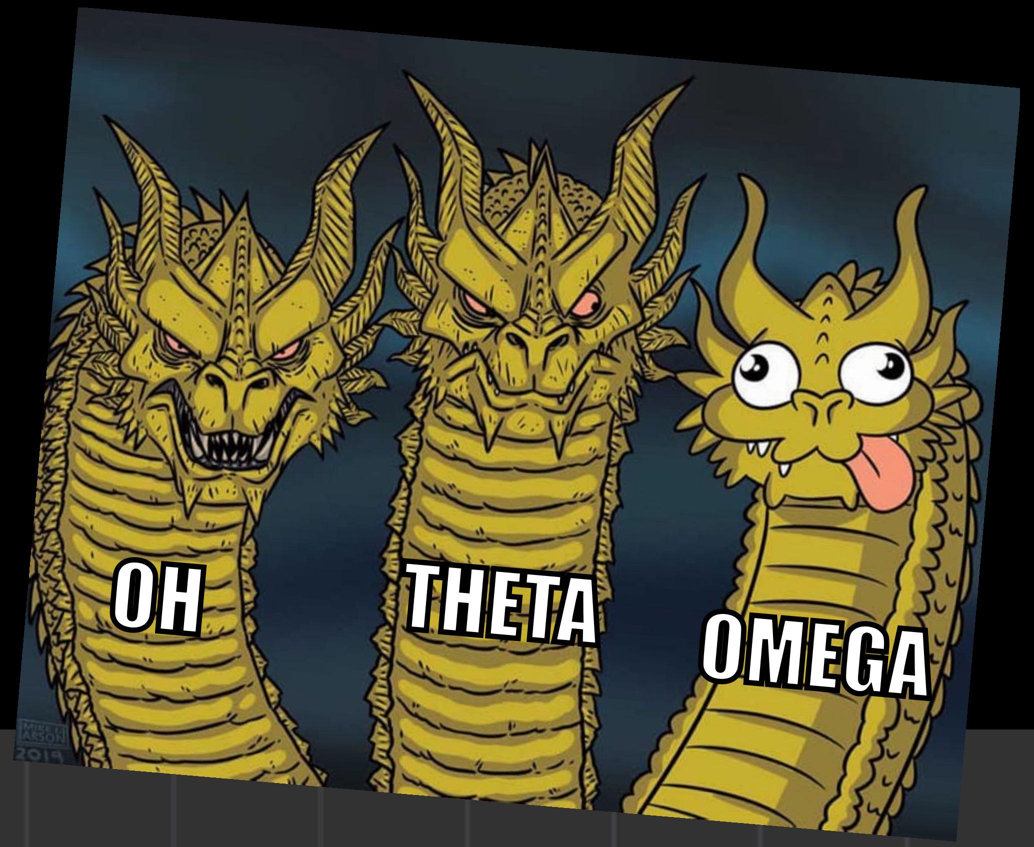
Orders of Growth - Why Do We Use Them?

- Why Big Oh?
  - What does the red line represent?
  - The *worst case* scenario of the algorithm!
  - i.e. it says: “when n is large enough, r(n) will be bad, but it can’t do worse than me!”

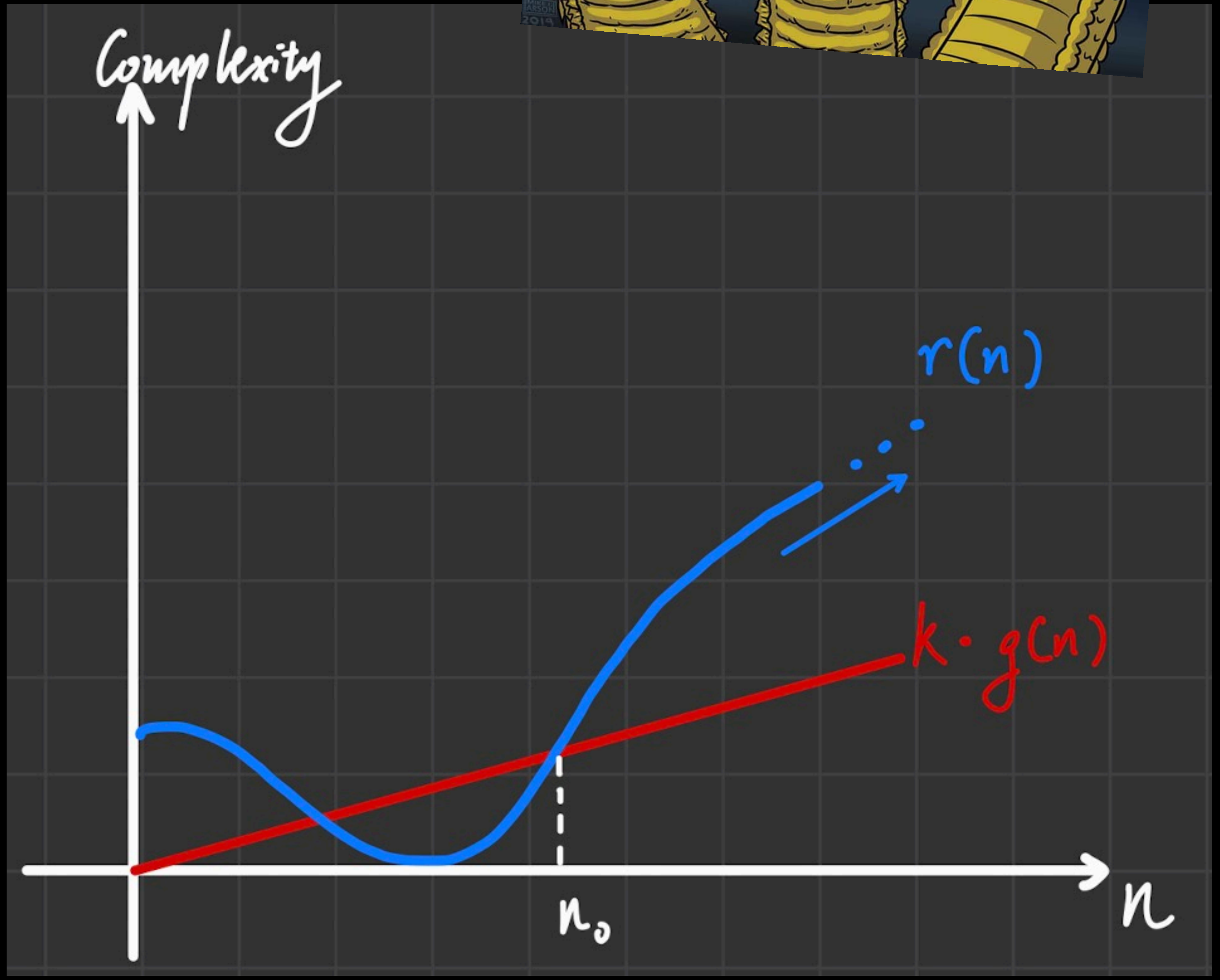


Recap

Orders of Growth - Why Do We Use Them?



- Why Big Theta?
  - Red line: the *best case* scenario of the algorithm
  - We usually don't concern ourselves with that
    - or at least not in CS1101S



Recap

Orders of Growth - Constants

- We do ***<u>not</u>*** consider constants in complexity
  - O(1) ≡ O(2), Θ(n) ≡ Θ(99999 * n), etc.
  - Changing base:

$$\log_a N = \boxed{\frac{1}{\log_b a}} \log_b N$$

just a constant

$$\log_a N = C \log_b N$$

$$O(\log_a N) = O(C \log_b N)$$

$$\boxed{O(\log_a N) = O(\log_b N)}$$

in general, O(log N)

Recap

Orders of Growth - Quiz

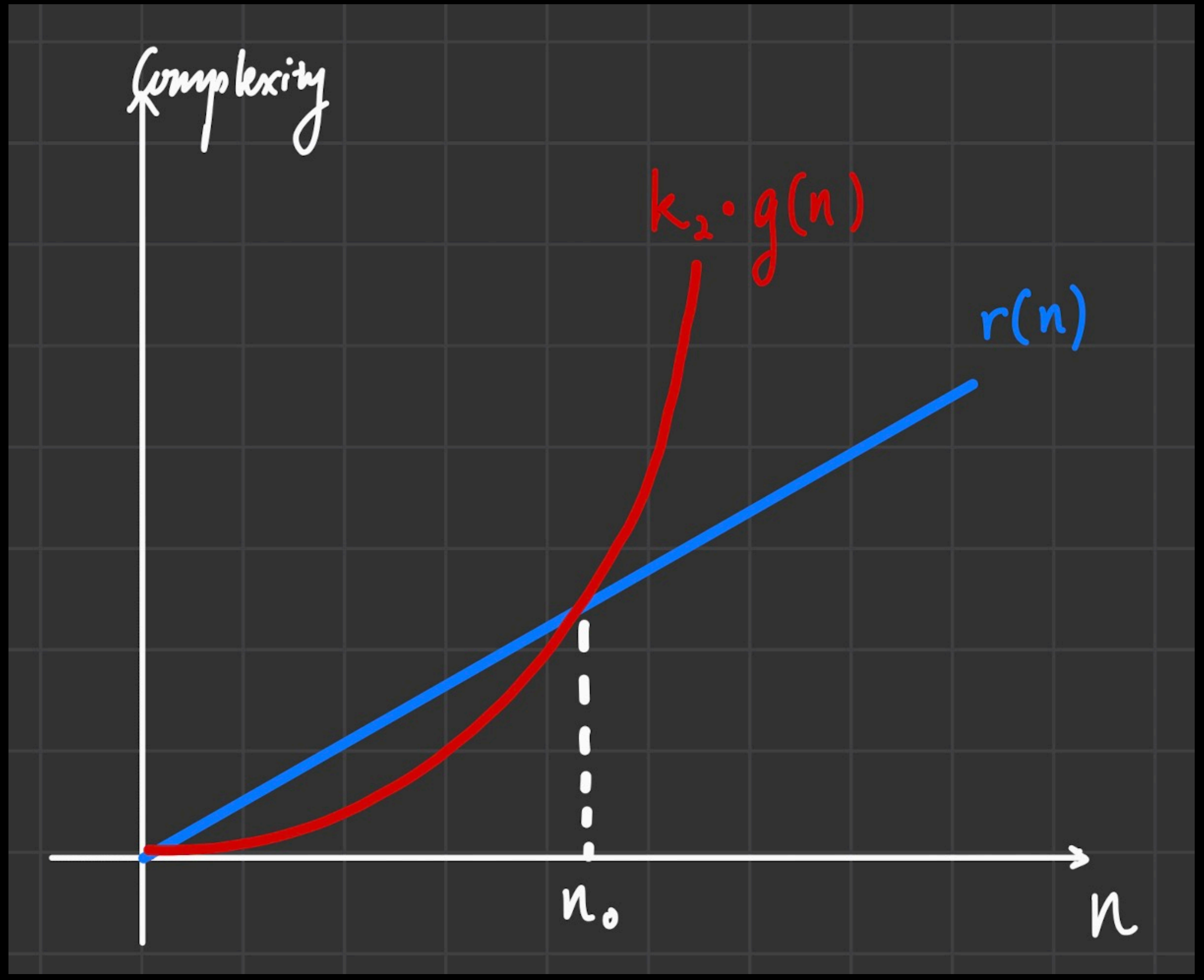
- Claim: “if my algorithm runs in $\Theta(1)$ time then it runs in $O(n)$ time”
  - true or false?
  - Answer: true!

some tips

Recap

Orders of Growth - Equivalence Relationships

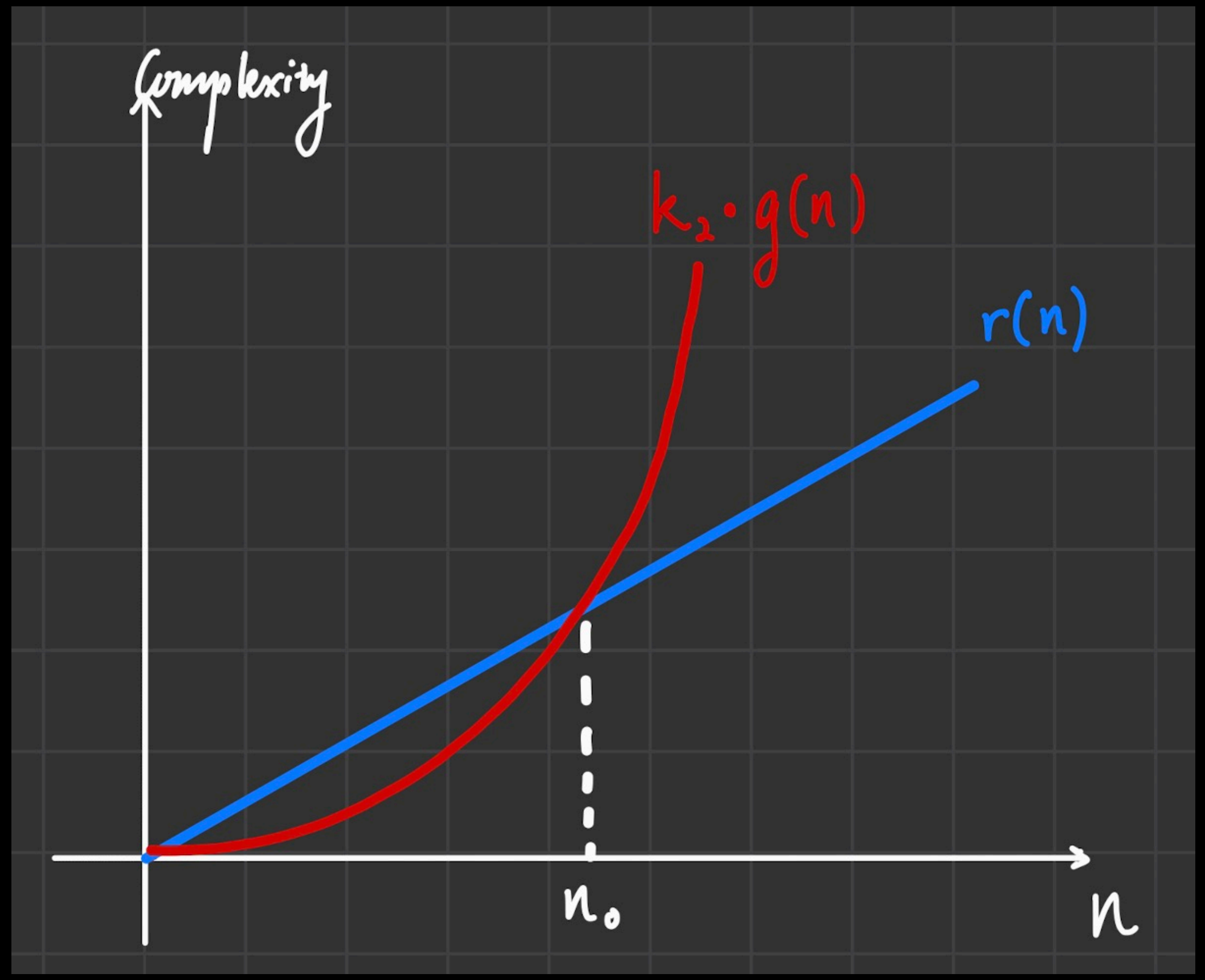
- Assume a function with complexity *Θ(n)* and a resource function $r(n) = n$
  - Thus, $k1 \cdot g(n) \leq r(n) \leq k2 \cdot g(n)$ for some *k1, k2, g and n0*
- Can we find a k2 and g such that r(n) $\lesseqgtr$ $k2 \cdot g(n)$ *for* $n \geq n0$?



Recap

Orders of Growth - Equivalence Relationships

- So what can we deduce from this?
  - No upper limit to Big Oh, O(infinity) perhaps
  - Lower limit to Big Omega: $\Omega(1)$



Recap

Orders of Growth - Equivalence Relationships

- If a algorithm runs in $\Theta(n)$, then the algorithm is also:
  - $\Omega(1)$, $\Omega(n)$ and $O(n)$, $O(n^2)$
- Every algorithm’s complexity is technically $\Omega(1)$ and O(infinity)
  - trivial (if you don’t understand, go back to the definitions!)
  - but this doesn’t help us with analysis :(

Recap

Orders of Growth - Ranked

- From least resource complex to most:
  - $O(1) < O(\log n) < O(n \cdot \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$
- Aim to find a more efficient algorithm!
  - Recall from S3: `expt` vs `fast_expt`

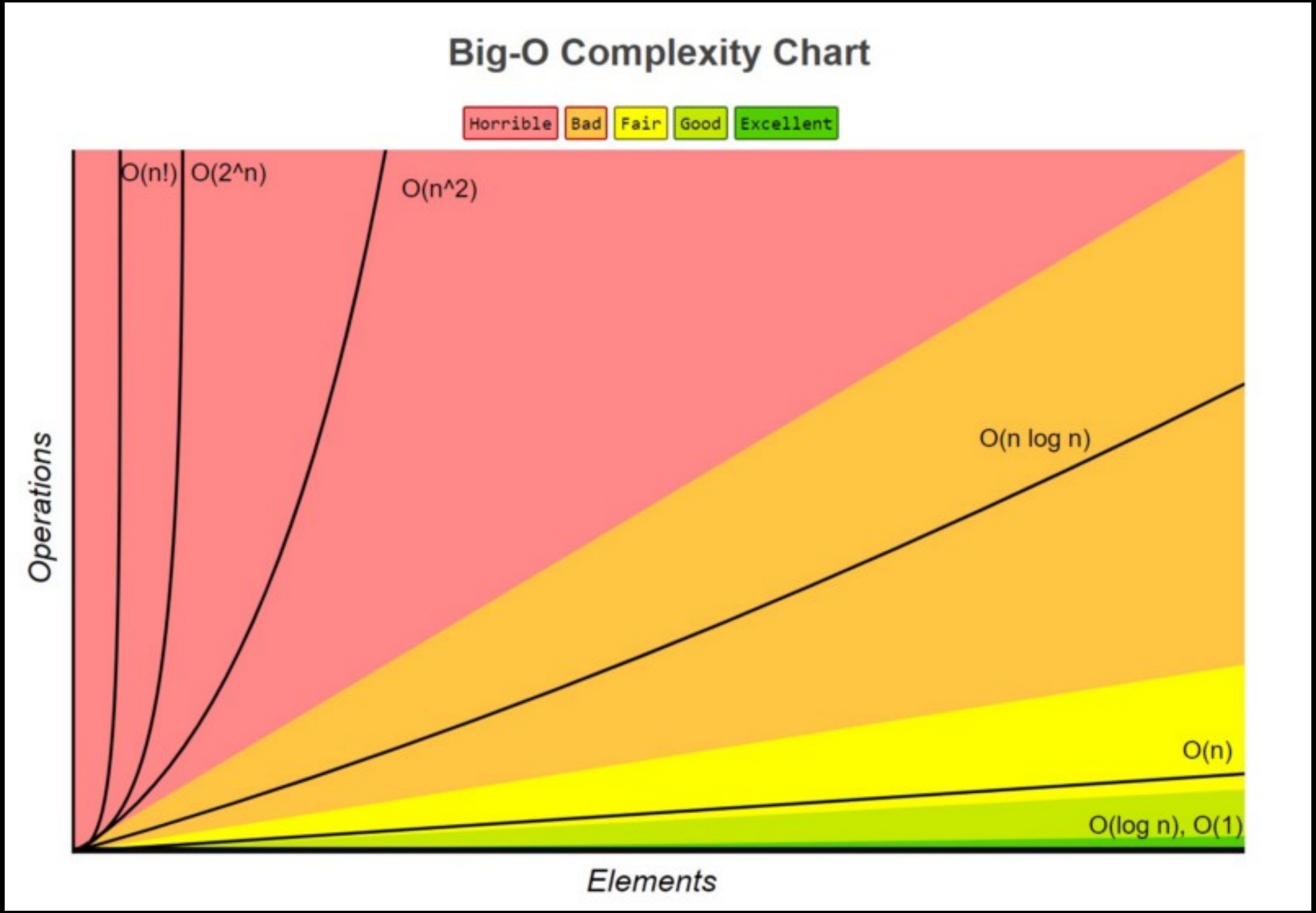
O(n)

O(log n)

for the interested:
trying to brute force the Travelling Salesman Problem, or finding all possible permutations

Recap

Orders of Growth - Ranked



Recap

Orders of Growth - Summation

- When adding: drop the less significant terms
  - $O(n^2 + 2n + 1) \equiv O(n^2)$
  - $O(\log(n) + n) \equiv O(n)$

Recap

Orders of Growth - Multiplication

- When multiplying: take the product
  - $O(n^2 * n) \equiv O(n^3)$
  - $O(\log(n) * n) \equiv O(n \cdot \log n)$

Recap

Orders of Growth - Coefficients

- Drop coefficients
    - $O(n^2 \cdot 0.5) \equiv O(n^2)$
    - $O(3) \equiv O(3 * 1) \equiv O(1)$

Recap

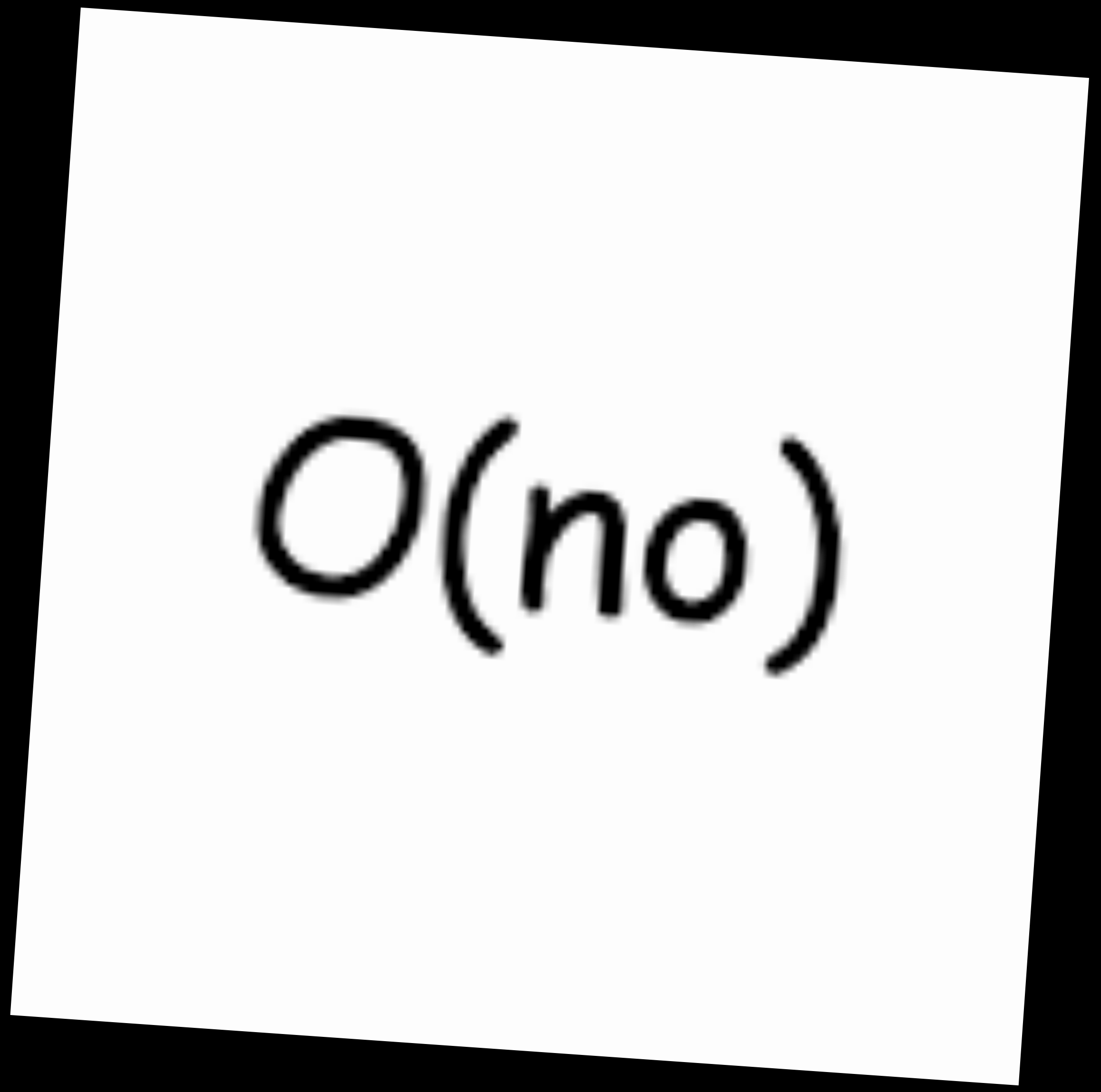
Orders of Growth - Summary

- Complexities are used to indicate the resources used for an algorithm
- Most commonly used: Θ and O
- Addition and multiplication rules
- Coefficients are discarded

Recap

Orders of Growth - Summart

- You WILL be asked to do complexity analysis during exams.
  - Analyse a given programme
  - Write a programme in XXX time complexity
  - Write a programme, THEN analyse it!



Any questions?

Recap: Higher Order Functions

Recap

Higher Order Functions - Function Definitions

- In S2, we've learnt how to define functions:

```
function add_one(x) {
    return x + 1;
}
```

- Now we can define "anonymous" functions (arrow functions)
    - Makes your programmes more concise

Recap

Higher Order Functions - Function Definitions

- Defining anonymous functions:

```
x => x + 1;   // no name
```

```
(x, y) => x + y;   // no name
```

parameters expression

- To give names:

```
const add_one = x => x + 1;
```

equivalent to:

```
function add_one(x) {
    return x + 1;
}
```

- To call this function:

```
add_one(2);   // same as normal functions
```

Recap

Higher Order Functions - Function Signatures

- Every function has a “signature”
- It takes in something(s) and return something(s)
- Functions are sensitive to their signatures
    - Can’t run if signature is incorrect!

Recap

Higher Order Functions - Function Signatures

- Example:
  - `const sum = (x, y) => x + y;`
  - `// Signature: (number, number) -> number`
  - i.e. function `sum` takes in 2 numbers, does some stuff, and returns 1 number
- Recall: abstraction!

Recap

Higher Order Functions - Function Signatures

- Spot the mistake:
    - `show(stackn(heart, sail));`
- `Error:`
    - `Signature: stackn: (number, rune, rune) -> rune`
    - But we only gave two runes!

Recap

Higher Order Functions - Function Signatures

- Another thing to take note:
  - Type is important too!

```
function add_one(x) {
    return x + 1;
}
add_one(rcross);
```

- How can we possibly add a number to a rune???

Recap

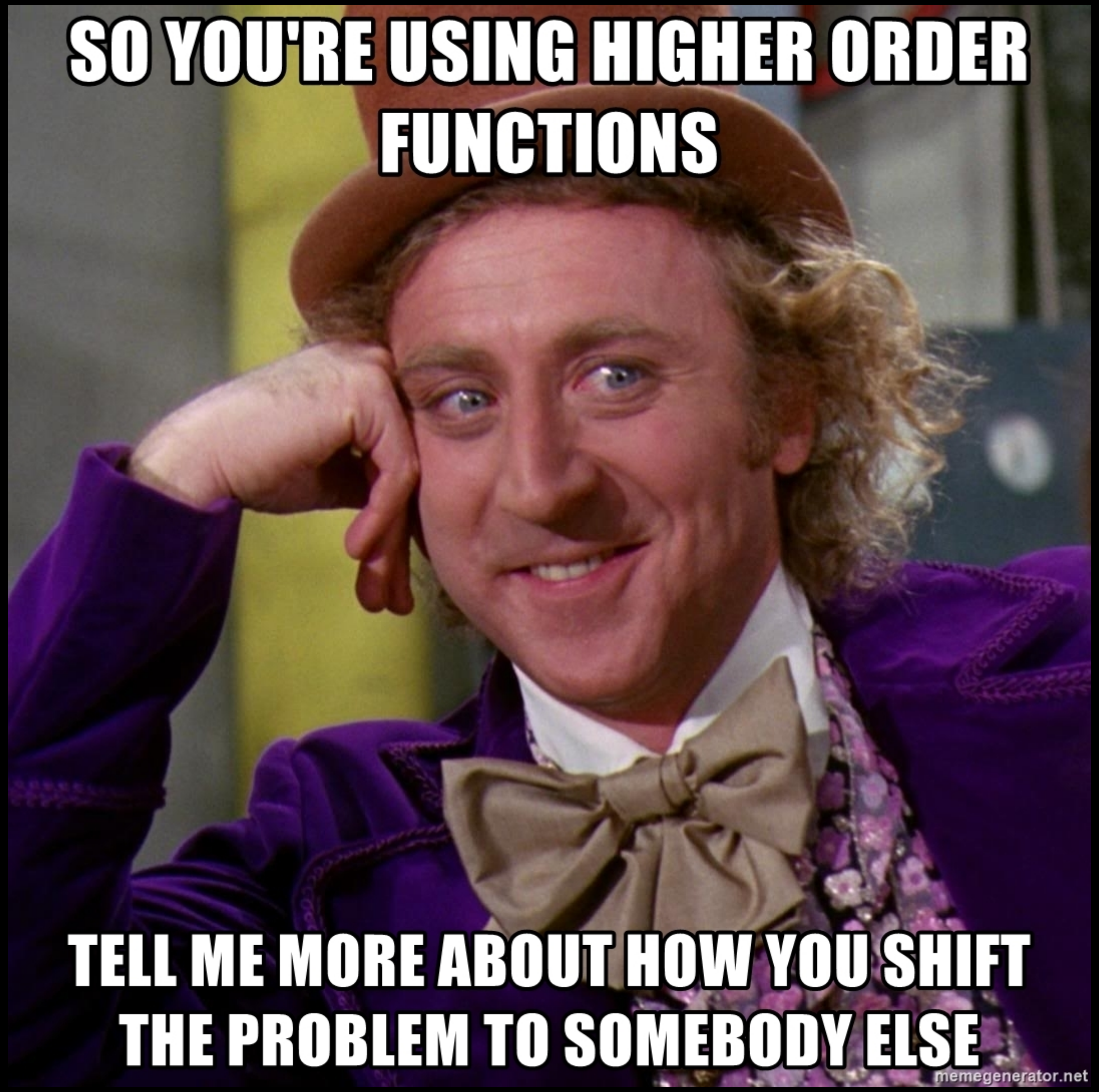
Higher Order Functions - Quiz

- Identify the signatures:
  - `const f1 = () => 1;`
  - `const f2 = some_val => some_val ? false : true;`
  - `const f3 = func => x => func(x);`
  - `const f4 = (x, y, z) => 0;`
- Discuss in your pairs!

Recap

Higher Order Functions - Functions as Arguments

- Functions can <u>take in</u> functions
- Functions can <u>return</u> functions
- i.e. make some other dude do your dirty work



Recap

Higher Order Functions - Quiz

- Functions as arguments follow parameter name binding rules
  - Creates a key-value pair, where the value is a function, instead of numbers, booleans or runes

```
function foo(f, x) {

  return f(x);

}

foo(x => x + 1, 10);  // binds `f` in `foo` to (x => x + 1)
```

Recap

Higher Order Functions - Interpreting HoF

- Strategy to interpret higher order functions:
  - Find the left most arrow
  - Consider the two sides:
  - Whatever’s on the left of the arrow are the parameters
  - Whatever’s on the right of the arrow is returned

Recap

Higher Order Functions - Interpreting HoF

- `const f3 = f => x => f(x + 1);`
  - Parameter: `f` (some function)
  - Returns: `x => f(x + 1)` (another function)
- Signature: function -> function

Recap

Higher Order Functions - Summary

- Functions can be anonymous
- Functions can be declared as `constants`
- Functions can take in functions and return functions

Recap

Higher Order Functions - Another Quiz

```
const foo = a => a;

const bar = abc => def => abc * def;
```

- Evaluate these expressions:
  - `foo(117);`
  - `bar(5)(10);`
  - `bar(foo(1))(foo(7));`
  - `bar(bar(3)(4))(bar(5)(foo(6)));`

Recap: Scope of Names

Recap

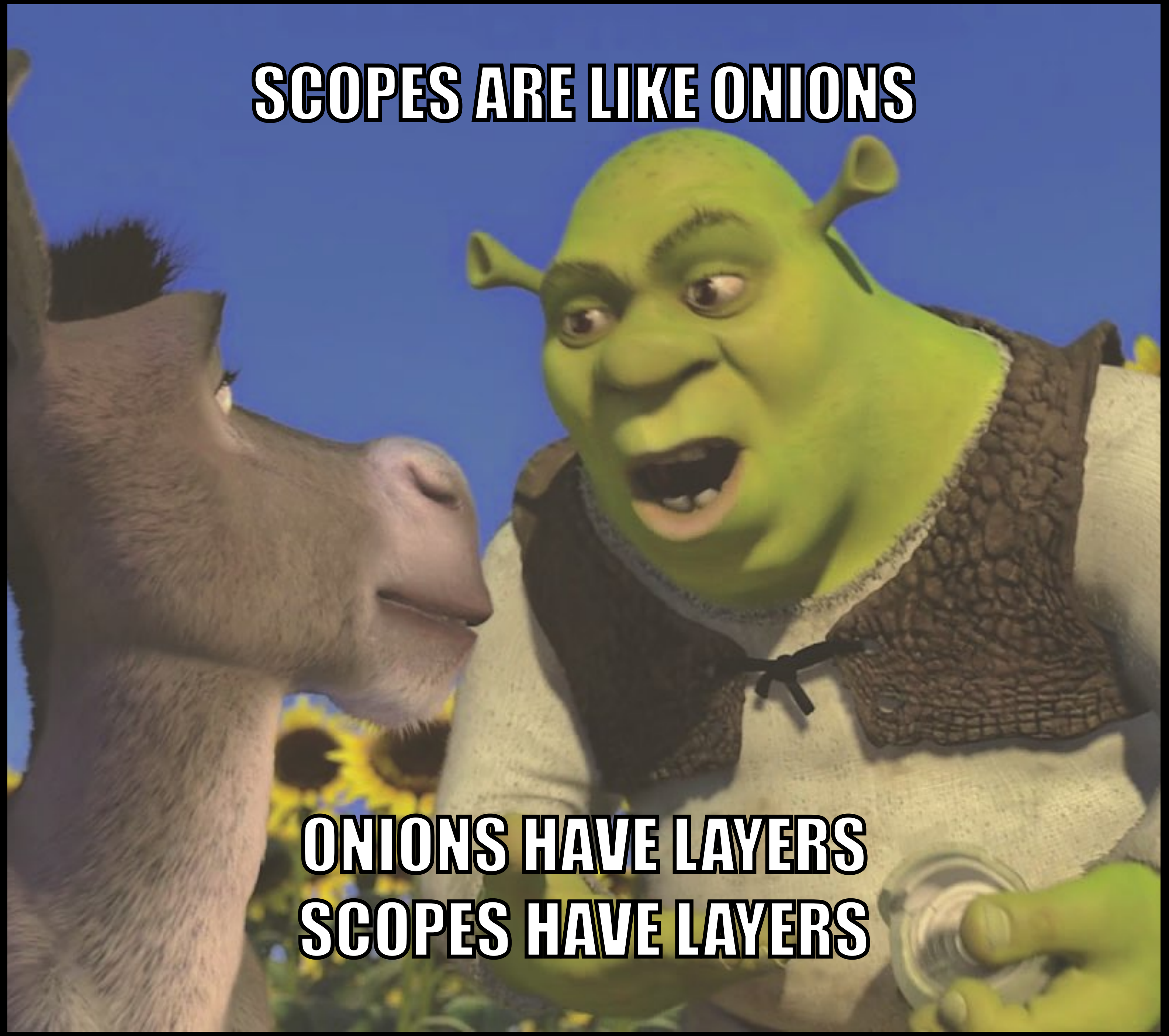
Scope of Names - Lexical Scoping

- Scoping rules:
  - A name occurrence refers to the <u>closest surrounding declaration</u>

Recap

Scope of Names - Overview

- Scopes are like onions: there are layers
    - When we refer to a name, the interpreter looks for it in the current scope
    - If it doesn't exist, it looks outwards in the layers until it finds it, or the global scope is reached
    - If it's still not found in the global layer, then the interpreter raises an error for "name XXX not declared"



Recap

Scope of Names - Overview

- What entails a scope?
  - Global context is a scope (the big universe)
    - Contains predefined functions and values (display, math_PI, etc)
  - Braces provide block scope `{ … }`
  - Functions create scope

```
function add_one(x) {

    return x + 1;

}
```

one scope
(together with params)

```
(x, y) => x + y;
```

one scope

Recap

Scope of Names - Shadowing

- Consider this programme:

```
const x = 7;

function f(y) {

    const x = 0;

    return x * y;

}

f(10);  // what’s the result?
```

Recap

Scope of Names - Shadowing

- Consider this NEW programme: (without using SourceAcademy)

```
const x = 7;

function f(y) {

    // const x = 0;

    return x * y;

}

f(10);  // what’s the new result?
```

Recap

Scope of Names - Shadowing

- What are the differences?
- Why did this happen?
- Discuss among your pairs

Recap

Scope of Names - Lexical Scoping

- Scoping rules:
  - A name occurrence refers to the <u>closest surrounding declaration</u>
  - The interpreter can only look upwards and outwards (to the left)

End of Recap

Challenge

Challenge

Start Your Brains

- Consider this programme:
- Attempt without running the programme in playground

```
const cs1101s = path => (quest, mission) => contest => contest(quest, mission);

const cs = ceg => ((infosec, bza, infosys) => bza);



cs(cs1101s(false))(NaN, 117, “hello”);

// NaN is just a pre-declared value in the global scope
```