CS341: ALGORITHMS (F23)

Lecture 1

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca

TABLE OF CONTENTS

- Course mechanics
- Models of computation
- Worked example: Bentley’s problem
  - Multiple solutions,
    demonstrating **different algorithm design techniques**
  - **Analyzed** in different models of computation

COURSE MECHANICS

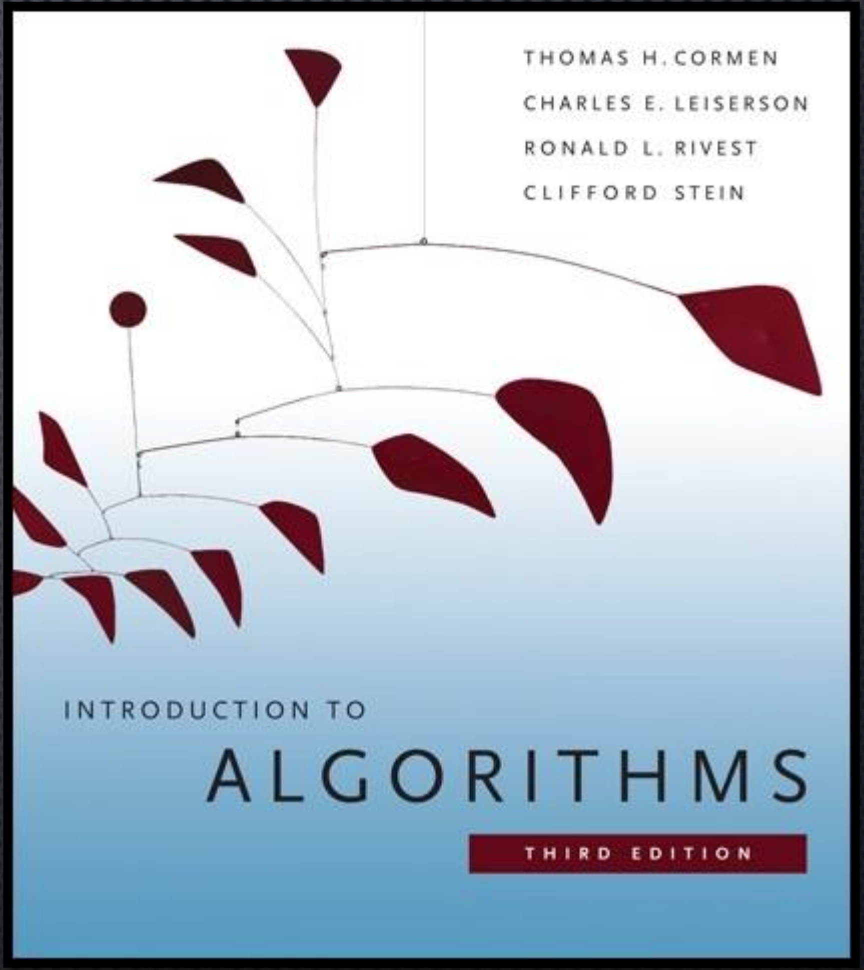
COURSE MECHANICS

- **In person**
  - Lectures
  - “Lab” section is for tutorials
- **Course website:** https://student.cs.uwaterloo.ca/~cs341/
  - Syllabus, calendar, policies, slides, assignments…
  - Read this and **mark** important dates.
- **Keep up with the lectures:** Material **builds** over time…
- **Piazza:** For questions and announcements.

ASSESSMENTS

- **All sections** have **same** assignments, midterm and final
- Sections are roughly synchronized to ensure necessary content is taught
- Tentative plan is 5 assignments, midterm, final
- See website for grading scheme, etc.

TEXTBOOK



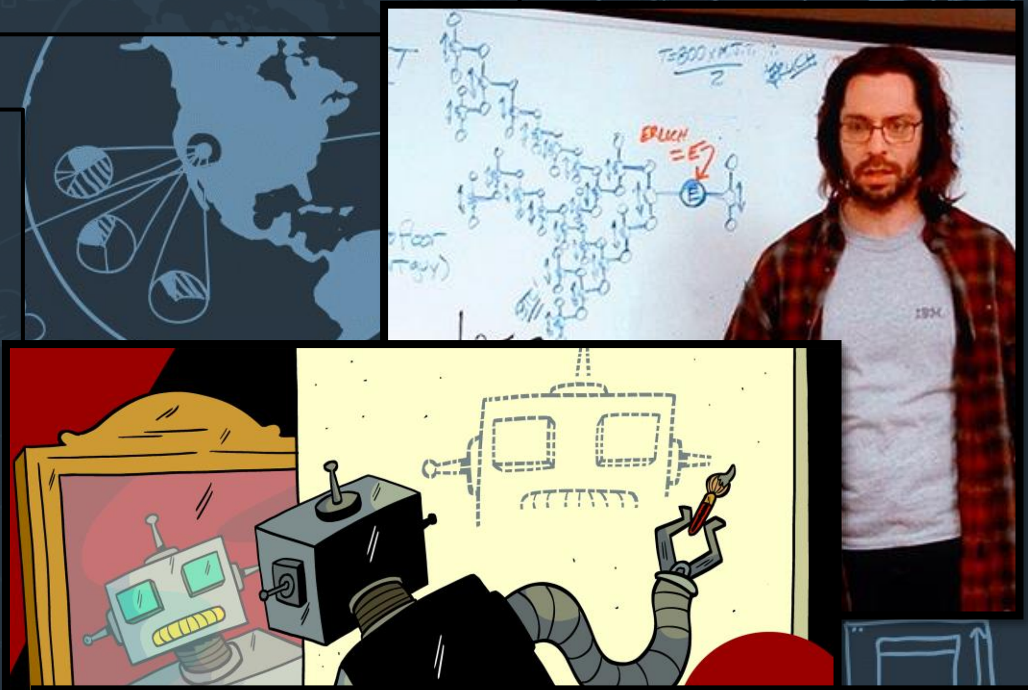
- Available **for free** via library website!
- You are expected to know
  - entire textbook sections, as listed on course website
  - **all the material presented in lectures** (unless we explicitly say you aren't responsible for it)
- Some other textbooks cover some material better… see www

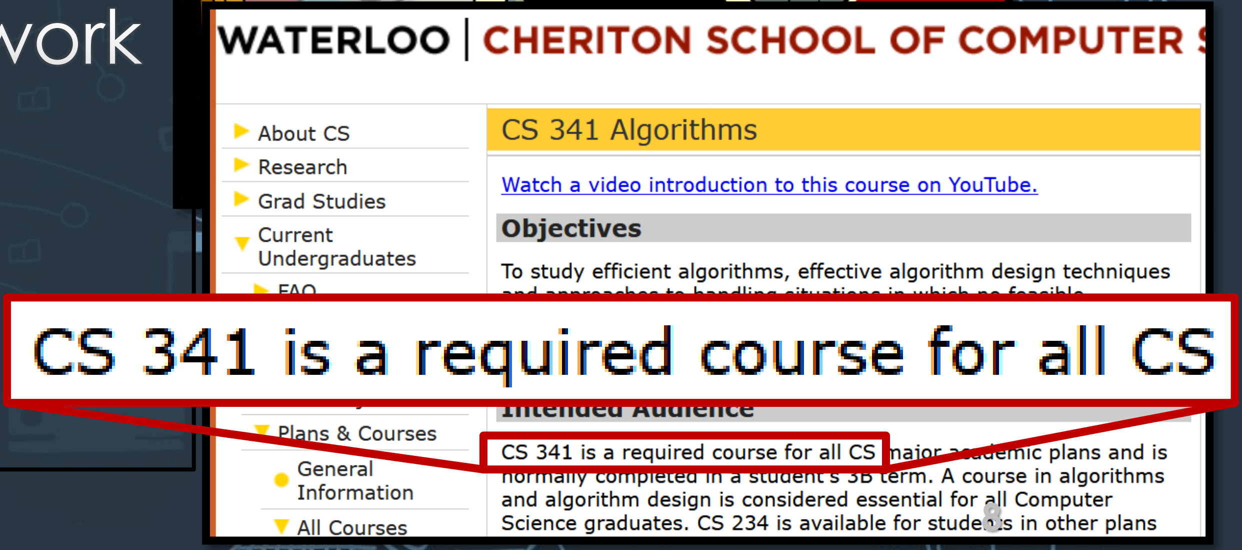
ACADEMIC OFFENSES

- Beware plagiarism
  - **High level discussion** about solutions with individual students is **OK**
  - Don’t take written notes away from such discussions
  - Class-wide discussion of solutions is **not** OK (until deadline+2 days)

WHY IS CS341 IMPORTANT FOR YOU?

- Algorithms is the heart of CS
  - It appears often in later courses
  - It dominates **technical interviews**
    - Master this material…
      make your interviews easy!
- Designing algorithms is **creative** work
  - Useful for some of the more interesting jobs out there
- And, you want to graduate…





WATERLOO | CHERITON SCHOOL OF COMPUTER S

- About CS
- Research
- Grad Studies
- Current Undergraduates
  - FAQ
  - Plans & Courses
    - General Information
    - All Courses

CS 341 Algorithms

Watch a video introduction to this course on YouTube.

Objectives

To study efficient algorithms, effective algorithm design techniques and approaches to handling situations in which no feasible

CS 341 is a required course for all CS

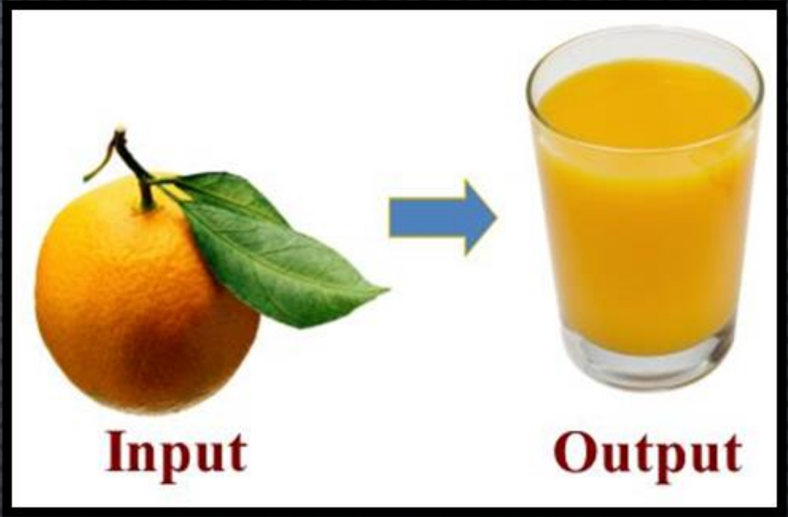
Intended Audience

CS 341 is a required course for all CS major academic plans and is normally completed in a student's 3B term. A course in algorithms and algorithm design is considered essential for all Computer Science graduates. CS 234 is available for students in other plans

MODELS OF COMPUTATION

WHAT IS A **COMPUTATIONAL PROBLEM**?

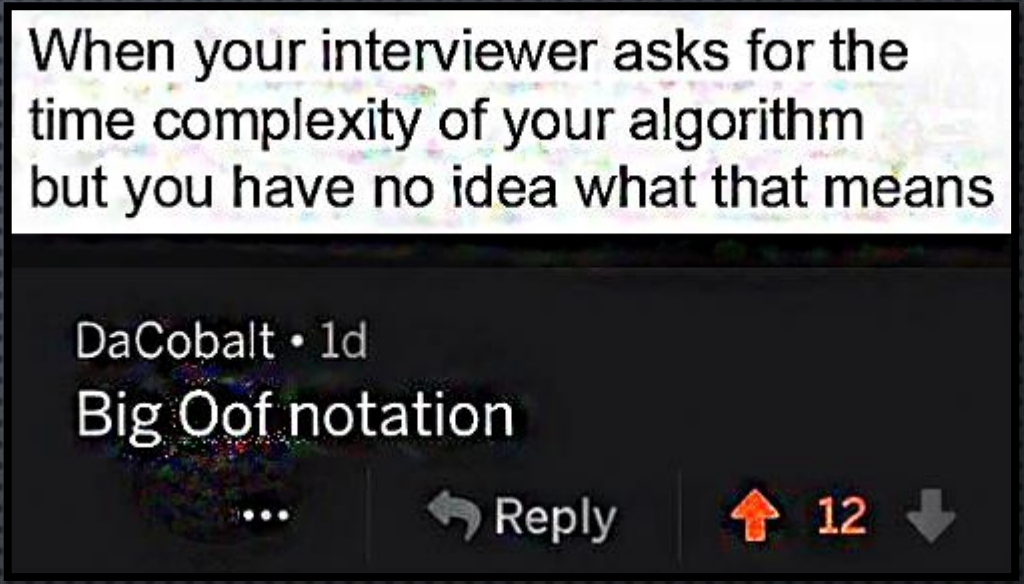
- Informally: A description of input, and the **desired output**



WHAT IS AN **ALGORITHM**?

- Informally: A well-defined **procedure** (sequence of steps) to solve a **computational problem**

ANALYSIS OF ALGORITHMS



When your interviewer asks for the time complexity of your algorithm but you have no idea what that means

DaCobalt • 1d

Big Oof notation

- Every program uses **resources**
  - CPU instructions / cycles → **time**
  - Memory (RAM) → **space**
  - Others: I/O, network bandwidth/messages, locks… (not covered in this course)
- **Analysis** is the study of **how many** resources an algorithm uses
  - Usually using big-O notation (to ignore constant factors)

Running Time of a Program: $T_M(I)$ denotes the running time of a program M on a problem instance I.

Worst-case Running Time as a Function of Input Size: $T_M(n)$ denotes the *maximum* running time of program M on instances of size n:

$$T_M(n) = \max\{T_M(I) : \text{Size}(I) = n\}.$$

Average-case Running Time as a Function of Input Size: $T_M^{avg}(n)$ denotes the *average* running time of program M over all instances of size n:

$$T_M^{avg}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I:\text{Size}(I)=n\}} T_M(I).$$

But how do we know how much **time** M will take on input I?

Depends on the **model of computation**

MODELS OF COMPUTATION

- Make analysis possible
- Ones covered in this course
  - **Unit cost** model
  - **Word RAM** model
  - **Bit complexity** model

UNIT COST MODEL

- Each variable (or array entry) is a **word**
- Words can contain unlimited bits
- **Basic operations on words take O(1) time**
  - Read/write a word in O(1)
  - Add two words in O(1)
  - Multiply two words in O(1)
- **Space complexity** is the **number of words** used (excluding the input)

BUT SOMETIMES WE CARE ABOUT WORD SIZE

- Suppose we want to **limit** the size of words
- Must consider how many **bits** are needed to represent a number $\boldsymbol{n}$

Need $\boldsymbol{\lfloor \log_2 n \rfloor + 1}$ **bits** to store $\boldsymbol{n}$

i.e., $\boldsymbol{\Theta(\log n)}$ **bits**

n in decimal	n in binary	$\boldsymbol{\lfloor \log_2 n \rfloor + 1}$
1	1	$\lfloor 0 \rfloor + 1 = 1$
2	10	$\lfloor 1 \rfloor + 1 = 2$
3	11	$\lfloor 1.58 \rfloor + 1 = 2$
4	100	$\lfloor 2 \rfloor + 1 = 3$
5	101	$\lfloor 2.32 \rfloor + 1 = 3$
6	110	$\lfloor 2.58 \rfloor + 1 = 3$
7	111	$\lfloor 2.81 \rfloor + 1 = 3$
8	1000	$\lfloor 3 \rfloor + 1 = 4$
9	1001	$\lfloor 3.17 \rfloor + 1 = 4$
10	1010	$\lfloor 3.32 \rfloor + 1 = 4$
11	1011	$\lfloor 3.46 \rfloor + 1 = 4$
12	1100	$\lfloor 3.58 \rfloor + 1 = 4$

WORD RAM MODEL

- Key difference: we care about the **size of words**
- Words can contain **O(lg n)** bits,
  where **n** is the **number of words in the input**
  - Word size depends on input size!
  - Intuition: if the input is an **array of n words**,
    a **word** is large enough to store an **array index**
- **Basic operations on words still take O(1) time**
  - (but the values they can contain are limited)

BIT COMPLEXITY MODEL

- Each variable (or array entry) is a **bit string**
- Size of a variable **x** is the number of bits it needs
  - It takes **O(log v) bits** to represent a **value v**
  - So if **v** is stored in **x**, the size of **x** must be $\Omega(\lg v)$ bits
- **Basic operations** are performed on **individual bits**
  - Read/write a bit in O(1)
  - Add/multiply two bits in O(1)
- **Space complexity** is the total **number of bits** used (excluding the input)

BENTLEY'S PROBLEM

A worked example to demonstrate algorithm design & analysis

Bentley's Problem (introductory example)

Given an array of n integers, A[1], … , A[n], find the maximum sum of consecutive entries of A (return 0 if all entries of A are negative).

Example 1	**1**	**7**	**4**	**0**	**2**	**1**	**3**	**1**
Array index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Solution: 19
(take **all** of A[1..8])

Example 2	**-1**	**-7**	**-4**	**-1**	**-2**	**-1**	**-3**	**-1**
Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Solution: 0
(take **no** elements of A)

Example 3	**1**	**-7**	**4**	**0**	**2**	**-1**	**3**	**-1**
Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

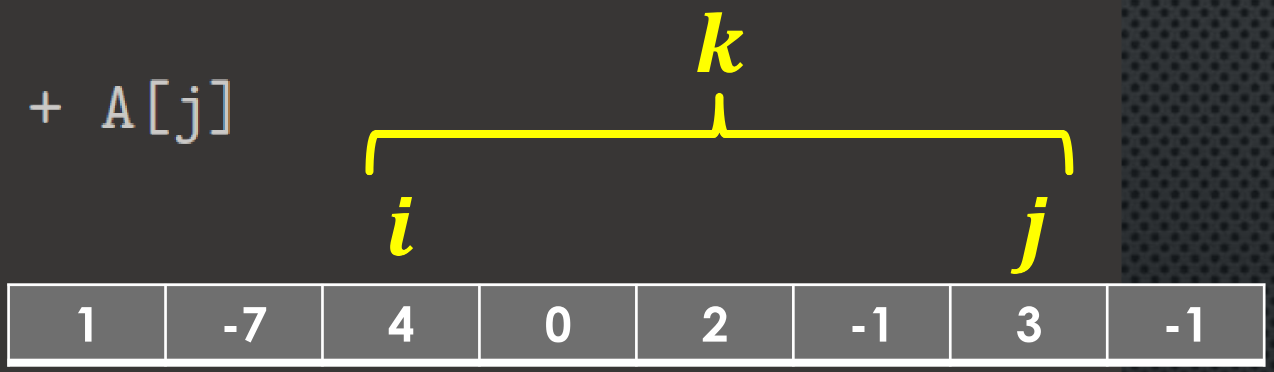
Solution: 8
(take A[3..7])

Bentley's Problem: Solution 1

Design: brute force

```
max := 0;
for i := 1 to n do
   for j := i to n do
      // compute  A[i] + ... + A[j]
      sum := 0;
      for k := i to j do
         sum := sum + A[k];
      // compare to maximum sum observed so far
      if sum > max then max := sum;
```

Try all combinations of i, j
And for each combination,
sum over $k = i\,..\,j$



1	-7	4	0	2	-1	3	-1

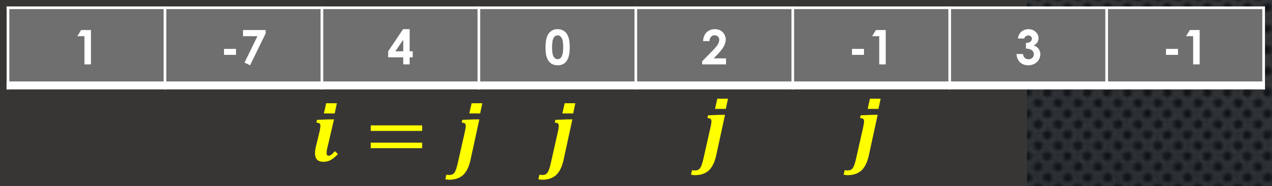
Time: in unit cost model?

Bentley's Problem: Solution 2

Design: slightly better brute force

<u>Avoid</u> repeatedly summing over $k = i\,..\,j$

```
max := 0;
for i := 1 to n do
  // for each j, compute A[i] + ... + A[j]
  sum := 0;
  for j := i to n do
    // update sum by adding the next entry A[j]
    sum := sum + A[j];
    // compare to maximum sum observed so far
    if sum > max then max := sum;
```



1	-7	4	0	2	-1	3	-1
		$i = j$	j	j	j		

Time: in unit cost model?

Bentley's Problem: Solution 3

Divide-and-Conquer can also be used here:

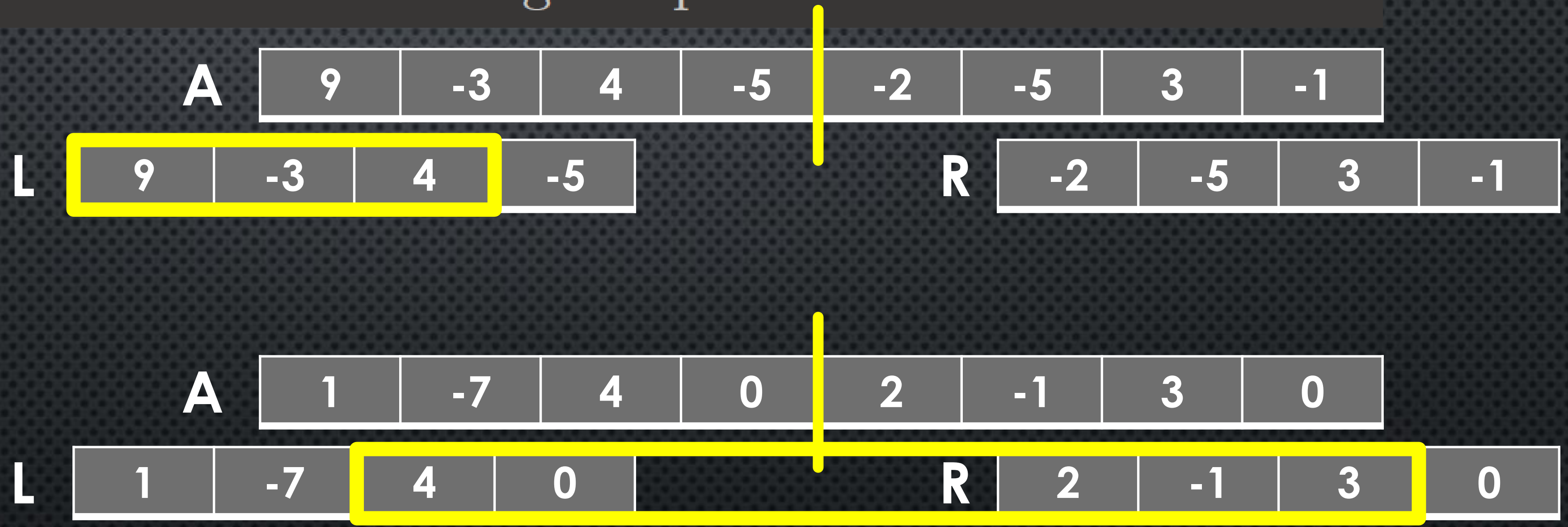
Divide an array into two equally-sized parts.

Our solution must either be entirely in the left part, or entirely in the right part, or it must be crossing the partition line.

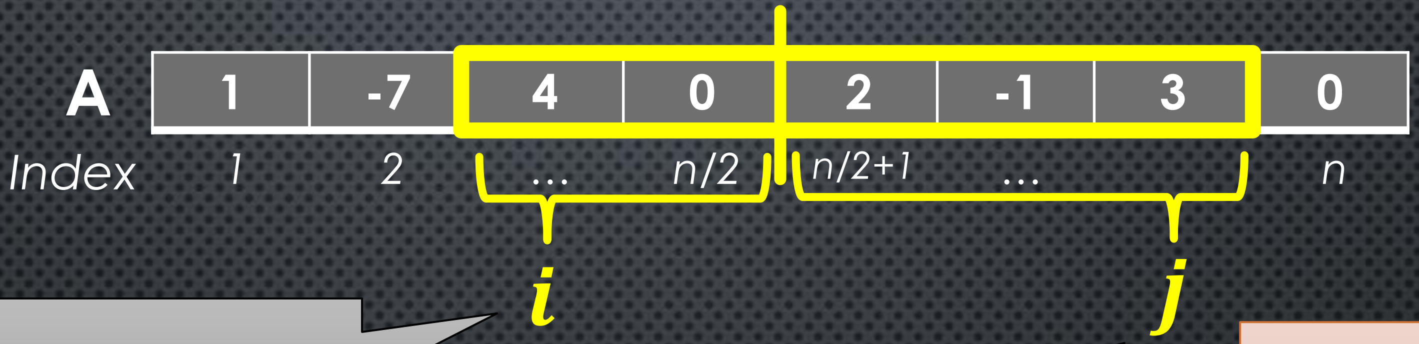
Case 1: optimal sol'n is **entirely in L**

Case 2: optimal sol'n is **entirely in R**

Case 3: optimal sol'n **crosses the partition**



Find: maximum subarray **going over the middle partition**



A	1	-7	4	0	2	-1	3	0
Index	*1*	*2*	…	*n/2*	*n/2+1*	…		*n*

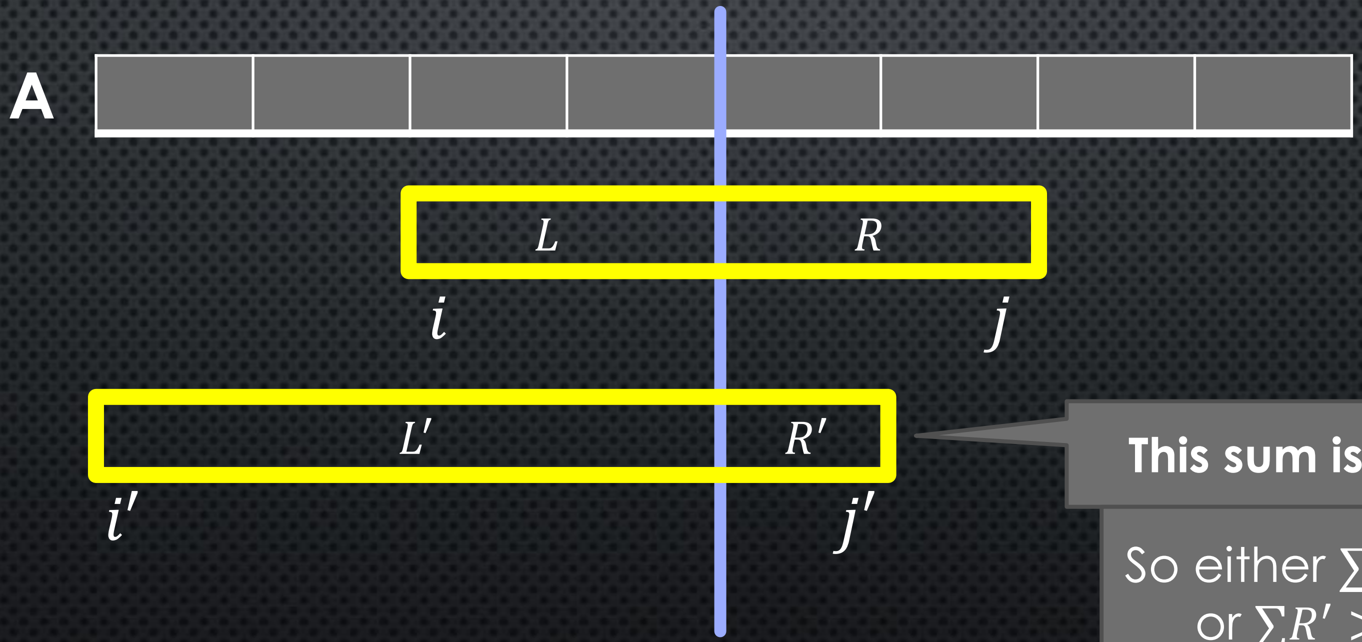
Find $\boldsymbol{i}$ that maximizes the sum over $\boldsymbol{i \ldots n/2}$

Find $\boldsymbol{j}$ that maximizes the sum over $\left(\boldsymbol{\frac{n}{2}} + \boldsymbol{1}\right) \boldsymbol{\ldots j}$

We can prove $\boldsymbol{A[i \ldots j]}$ **<u>is</u>** the maximum subarray going over the middle partition!

WHY $A[i \dots j]$ IS MAXIMAL

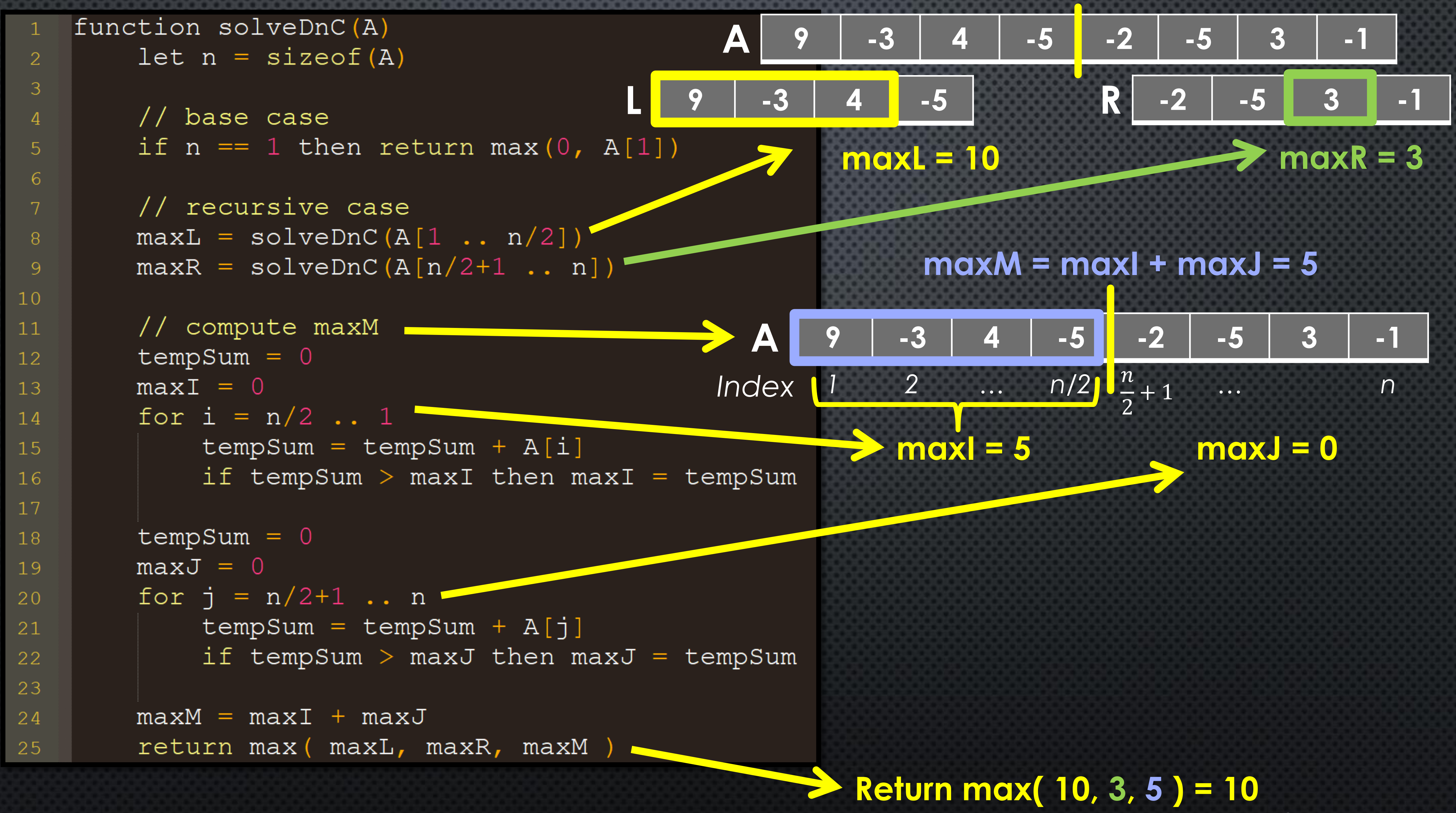
- Suppose not for contradiction
- Then some $A[i' \dots j']$ that crosses the **partition** has a **larger** sum



This sum is bigger

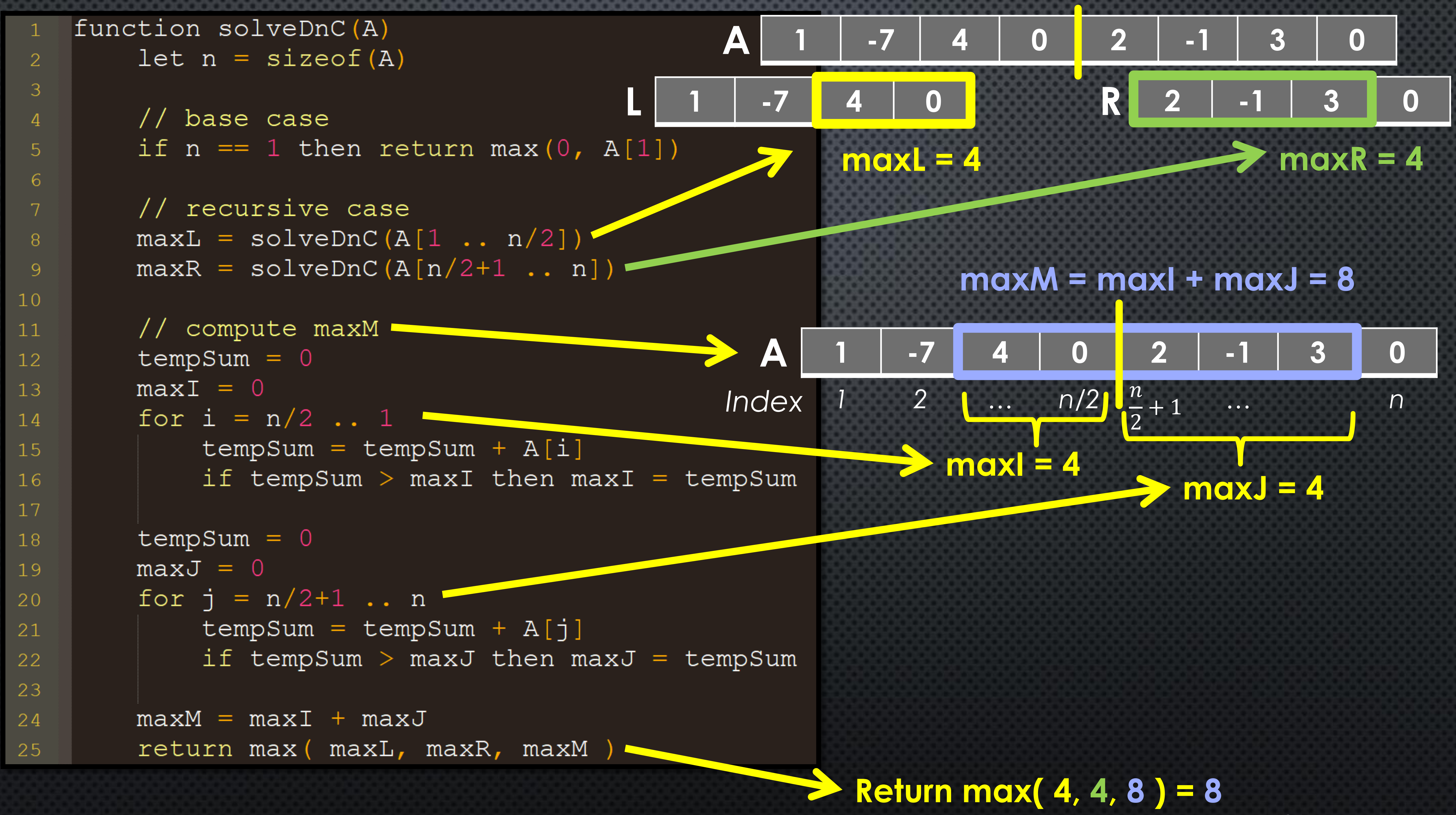
So either $\sum L' > \sum L$ or $\sum R' > \sum R$

But both are impossible!



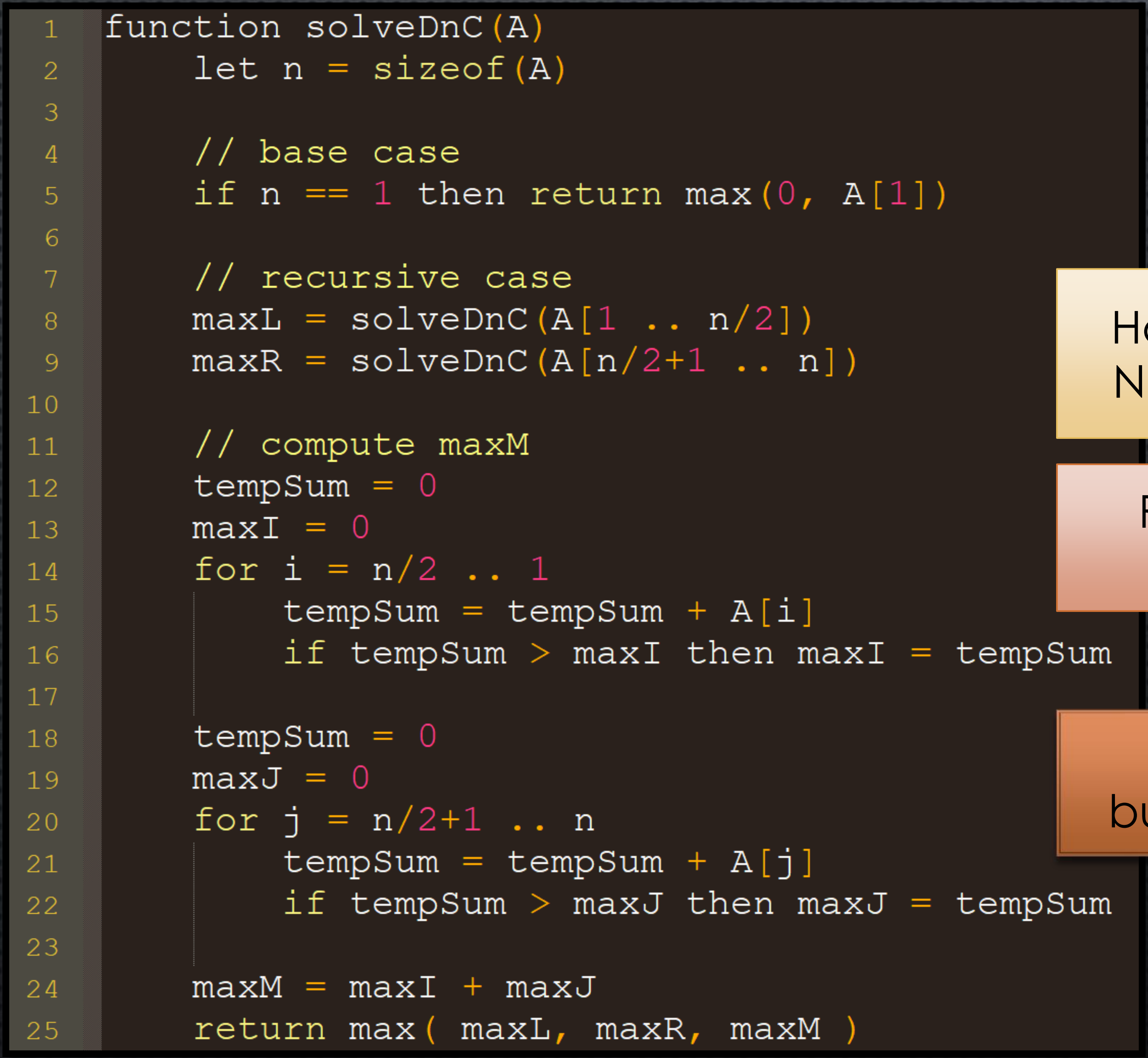
```
function solveDnC(A)
    let n = sizeof(A)

    // base case
    if n == 1 then return max(0, A[1])

    // recursive case
    maxL = solveDnC(A[1 .. n/2])
    maxR = solveDnC(A[n/2+1 .. n])

    // compute maxM
    tempSum = 0
    maxI = 0
    for i = n/2 .. 1
        tempSum = tempSum + A[i]
        if tempSum > maxI then maxI = tempSum

    tempSum = 0
    maxJ = 0
    for j = n/2+1 .. n
        tempSum = tempSum + A[j]
        if tempSum > maxJ then maxJ = tempSum

    maxM = maxI + maxJ
    return max( maxL, maxR, maxM )
```

Return max(10, 3, 5) = 10



```
function solveDnC(A)
    let n = sizeof(A)

    // base case
    if n == 1 then return max(0, A[1])

    // recursive case
    maxL = solveDnC(A[1 .. n/2])
    maxR = solveDnC(A[n/2+1 .. n])

    // compute maxM
    tempSum = 0
    maxI = 0
    for i = n/2 .. 1
        tempSum = tempSum + A[i]
        if tempSum > maxI then maxI = tempSum

    tempSum = 0
    maxJ = 0
    for j = n/2+1 .. n
        tempSum = tempSum + A[j]
        if tempSum > maxJ then maxJ = tempSum

    maxM = maxI + maxJ
    return max( maxL, maxR, maxM )
```

Return max(4, 4, 8) = 8



```
function solveDnC(A)
    let n = sizeof(A)

    // base case
    if n == 1 then return max(0, A[1])

    // recursive case
    maxL = solveDnC(A[1 .. n/2])
    maxR = solveDnC(A[n/2+1 .. n])

    // compute maxM
    tempSum = 0
    maxI = 0
    for i = n/2 .. 1
        tempSum = tempSum + A[i]
        if tempSum > maxI then maxI = tempSum

    tempSum = 0
    maxJ = 0
    for j = n/2+1 .. n
        tempSum = tempSum + A[j]
        if tempSum > maxJ then maxJ = tempSum

    maxM = maxI + maxJ
    return max( maxL, maxR, maxM )
```

Time: $\Theta(n \log n)$

(in unit cost model)

How do we analyze this running time?
Need new mathematical techniques!

Recurrence relations, recursion tree methods, master theorem…

This result is really quite good…
but can we do **asymptotically** better?

ANALYSIS IN THE **BIT COMPLEXITY** MODEL

- Revisiting Solution 1

```
max := 0;
for i := 1 to n do
  for j := i to n do
    // compute  A[i] + ... + A[j]
    sum := 0;
    for k := i to j do
      sum := sum + A[k];
    // compare to maximum sum observed so far
    if sum > max then max := sum;
```

Can only add a **pair of bits** in O(1) time. How many bits are added here?

$\mathrm{size}(\mathrm{A}[k]) \in O(\log \mathrm{A}[k])$ **bits.**

$\mathrm{size}(\mathrm{sum}) \in ???$

$\mathrm{sum} = \mathrm{A}[i] + \cdots + \mathrm{A}[k-1]$

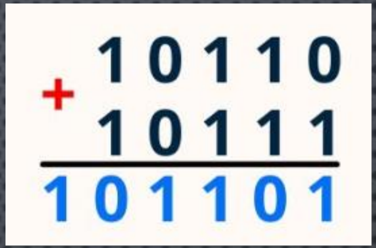
so $\mathrm{size}(\mathrm{sum}) \in O\big(\log(\mathrm{A}[i] + \cdots + \mathrm{A}[k-1])\big)$ **bits**

How to simplify?

COMPLEXITY OF ADDITION

Adding two numbers x+y takes O(max{size(x), size(y)}) bit operations

This can be rewritten O(size(x)+size(y)) = O(lg x + lg y)



$$\begin{array}{r} 10110 \\ +\ 10111 \\ \hline 101101 \end{array}$$

Fun fact: the size of x+y can be 1 bit larger than either x or y (multiplication can double #bits)

Let $\mathbf{M} = \boldsymbol{max}\{\boldsymbol{A}[\mathbf{1}], \dots, \boldsymbol{A}[\boldsymbol{n}]\}$

$\mathbf{size(sum)} \in \mathbf{O}\big(\mathbf{log}\ (\mathbf{A}[\boldsymbol{i}] + \cdots + \mathbf{A}[\boldsymbol{k} - \mathbf{1}])\big)$
$\in \mathbf{O}\big(\mathbf{log}(\mathbf{M} + \cdots + \mathbf{M})\big)$ **bits**

$\in \boldsymbol{O}(\mathbf{log}\ ((\mathbf{k} - \boldsymbol{i})\mathbf{M})$ **bits**

Optional: simplify to $\boldsymbol{O}(\mathbf{log\ kM})$

ADDING SUM AND A[K]

```
sum := sum + A[k];
```

- **Recall** $\mathrm{size}(sum) \in O(\log kM)$, $\mathrm{size}(\mathrm{A}[k]) \in O(\log A[k])$ bits
- Adding them takes $O(\log(kM) + \log A[k])$ bit operations
- And since $\log A[k] \le \log M$ we get:
$$O(\log(kM) + \log M)$$
- And the first term asymptotically dominates:
$$\boldsymbol{O(\log kM)}$$

ZOOMING OUT TO THE K LOOP

```
for k := i to j do
  sum := sum + A[k];
```

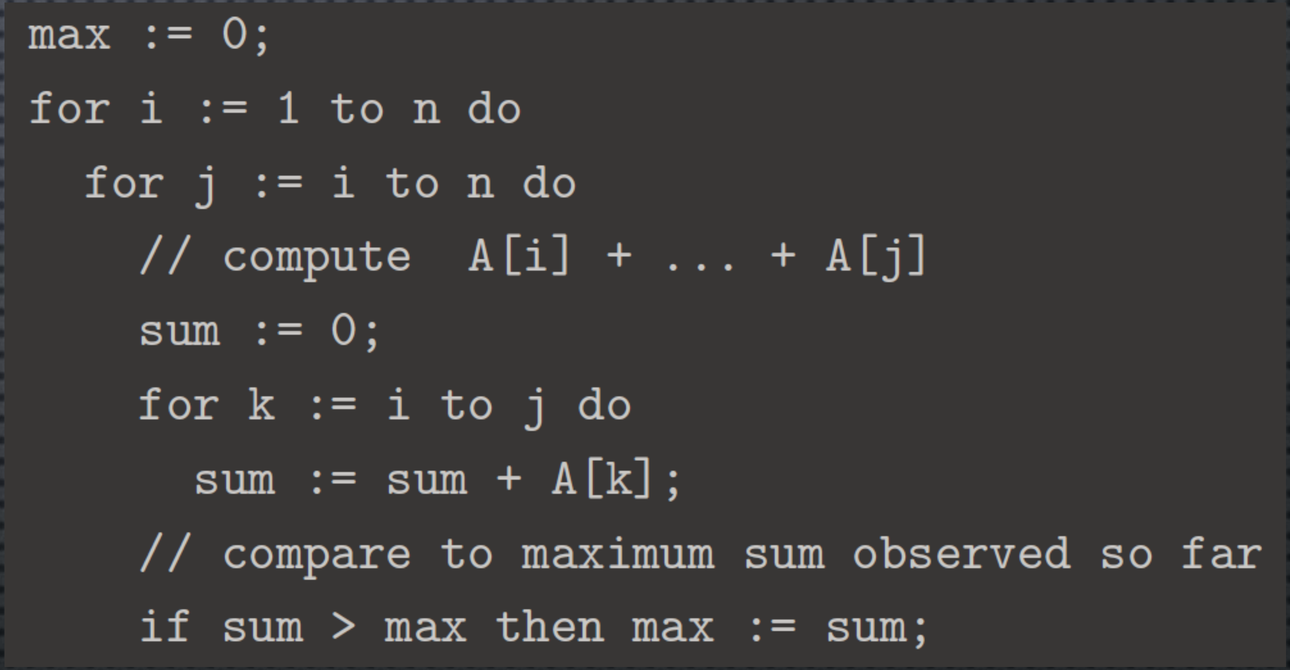
- The addition happens for all values of k
- Total time for the loop is at most $\sum_{k=i}^{j} O(\log kM)$
- Complicated to sum for $k = i \dots j$
  so get an upper bound with $k = 1 \dots n$
- $\sum_{k=1}^{n} O(\log kM) = O(\log M + \log 2M + \log 3M + \cdots + \log nM)$
- $\subseteq O(\underbrace{\log nM + \log nM + \log nM + \cdots + \log nM}_{n})$
- $= O(n \log nM)$

Careful to check this does not affect the Θ complexity (much). (Check by finding similar Ω result.)

And similarly for this…

ACCOUNTING FOR THE OUTER LOOPS

- k loop is repeated at most n^2 times
- Each time taking at most $O(n \log nM)$ time
- So total runtime is $O(n^3 \log nM)$ time



```
max := 0;
for i := 1 to n do
  for j := i to n do
    // compute  A[i] + ... + A[j]
    sum := 0;
    for k := i to j do
      sum := sum + A[k];
    // compare to maximum sum observed so far
    if sum > max then max := sum;
```

Compare to unit cost model:
$O(n^3)$ time

Difference is due to
(1) growth in variable sizes and
(2) cost of bitwise addition

log-factor difference is common…

HOW ABOUT WORD RAM?

- If each variable fits in a single word,
  the analysis (and result) is as in the unit cost model
- Since there are n input words,
  each $A[k]$ will fit in one word only if $\text{size}(A[k]) \in O(\log n)$
    - i.e., if $O(\log A[k]) = O(\log n)$
- If a variable is too big to fit in a word,
  it is stored in multiple words,
  and analysis looks more like bit complexity model

BENTLEY'S SOLUTIONS: RUNTIME IN PRACTICE

- Consider solutions implemented in C
  - Some values **measured** on a Threadripper 3970x
  - Red values **extrapolated** from measurements
  - 0 represents time under 0.01s

n	Sol.4 O(n)	Sol.3 O(n lg n)	Sol.2 O(n^2)	Sol.1 O(n^3)
100	0	0	0	0
1,000	0	0	0	0.12
10,000	0	0	0.036	2 minutes
100,000	0	0.002	3.582	33 hours
1M	0.001	0.017	6 minutes	4 years
10M	0.012	0.195	12 hours	3700 years
100M	0.112	2.168	50 days	3.7M years
1 billion	1.124	24.57	1.5 years	> age of life
10 billion	19.15	5 minutes	150 years	> age of universe

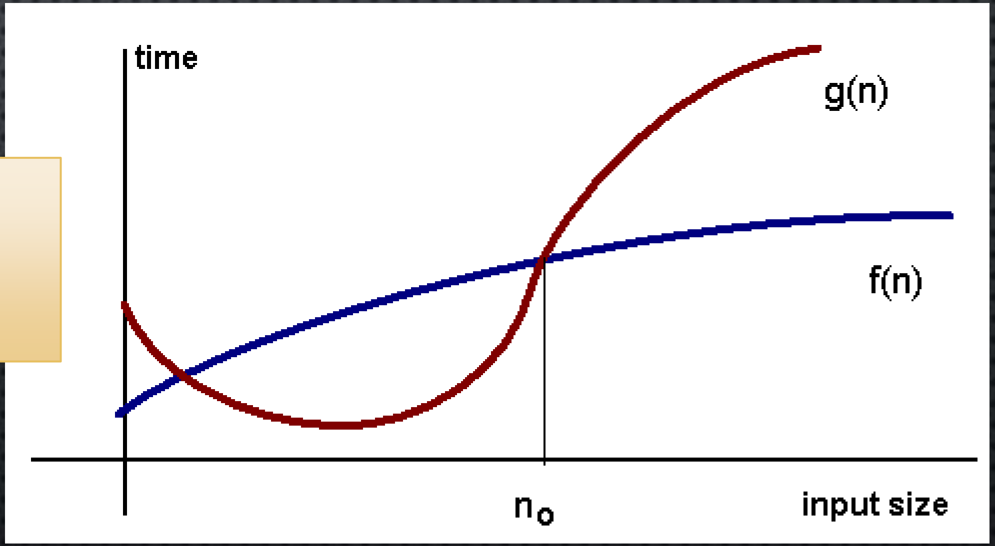
HOMEWORK: BIG-O REVIEW & EXERCISES

O-notation:

$f(n) \in O(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Here the complexity of f is **not higher** than the complexity of g.

$$f(n) \in O(g(n))$$

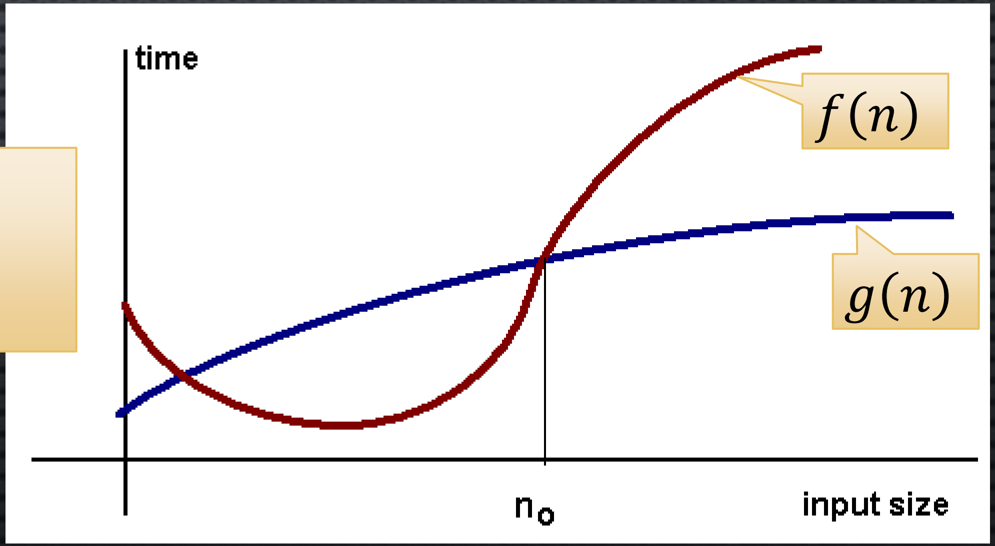


Ω-notation:

$f(n) \in \Omega(g(n))$ if **there exist** constants $c > 0$ and $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

Here the complexity of f is **not lower** than the complexity of g.

$f(n) \in \Omega(g(n))$



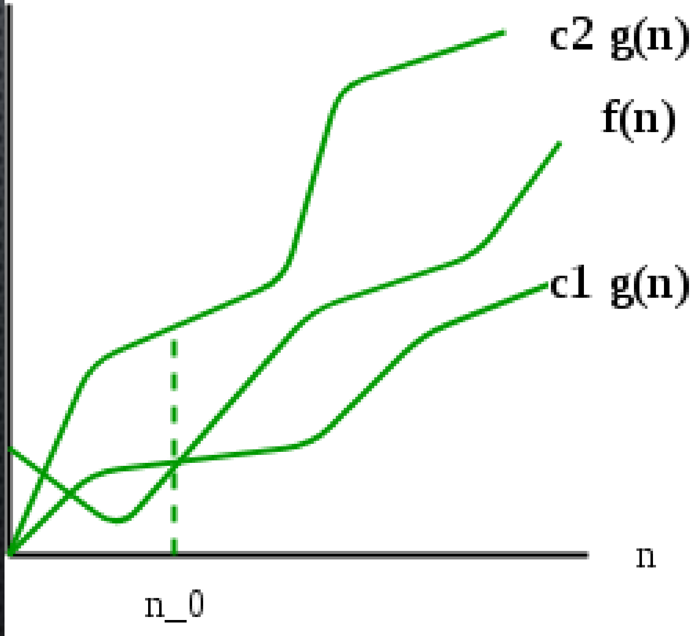
Θ-notation:

$f(n) \in \Theta(g(n))$ if **there exist** constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

Here f and g have the **same complexity**.

$f(n) \in \Theta(g(n))$

$g(n) \in \Theta(f(n))$



$f(n) \in O(g(n))$

$f(n) \in \Omega(g(n))$

$O + \Omega = \Theta$

o-notation:

$f(n) \in o(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Here f has **lower complexity** than g.

$f(n) \in o(g(n))$
implies
$f(n) \in O(g(n))$

But NOT
vice versa

ω-notation:

$f(n) \in \omega(g(n))$ if **for all** constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

Here f has **higher complexity** than g.

$f(n) \in \omega(g(n))$
implies
$f(n) \in \Omega(g(n))$

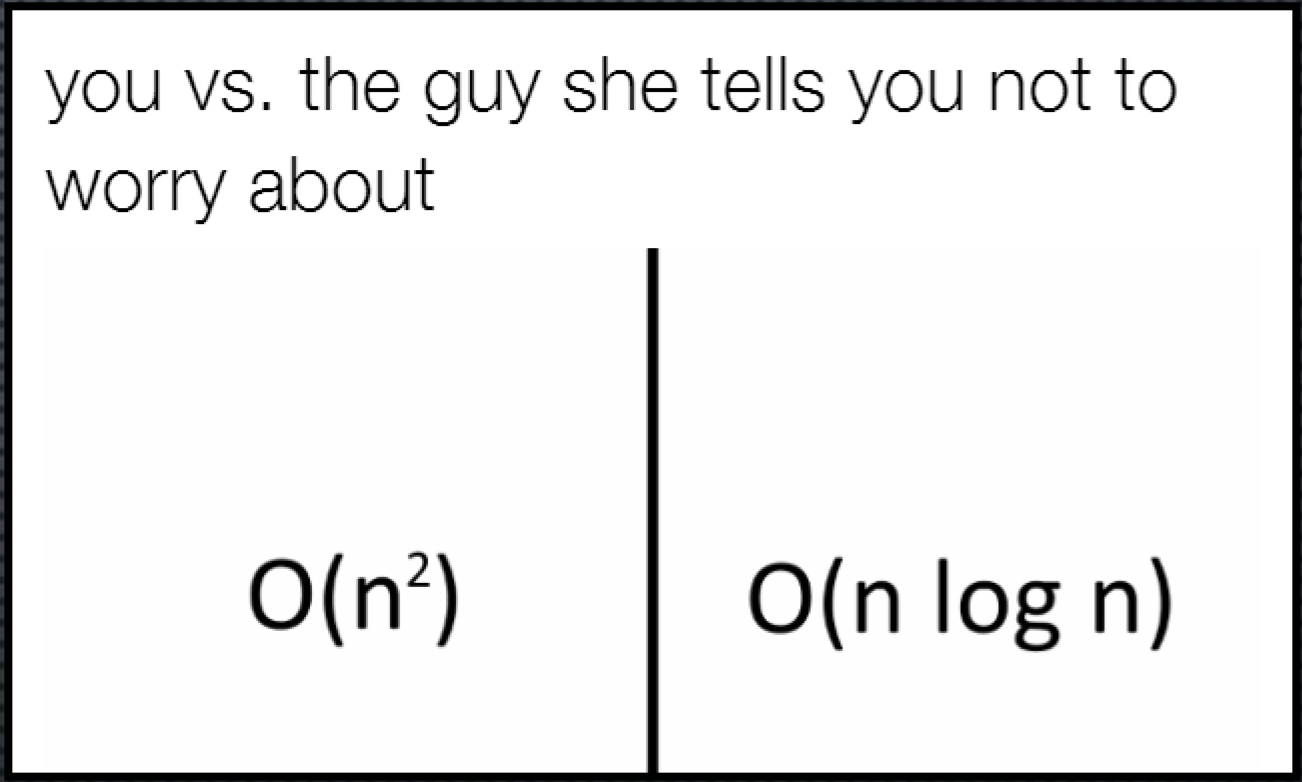
But NOT
vice versa

EXERCISE

- Which of the following are true?
- $n^2 \in O(n^3)$
- $n^2 \in o(n^3)$
- $n^3 \in \omega(n^3)$
- $\log n \in o(n)$
- $n \log n \in \Omega(n)$
- $n \log n^2 \in \omega(n \log n)$
- $n \in \Theta(n \log n)$

EXERCISE

- Which of the following are true?
- $n^2 \in O(n^3)$ YES
- $n^2 \in o(n^3)$ YES
- $n^3 \in \omega(n^3)$ NO
- $\log n \in o(n)$ YES
- $n \log n \in \Omega(n)$ YES
- $n \log n^2 \in \omega(n \log n)$ NO
- $n \in \Theta(n \log n)$ NO



you vs. the guy she tells you not to worry about

$O(n^2)$

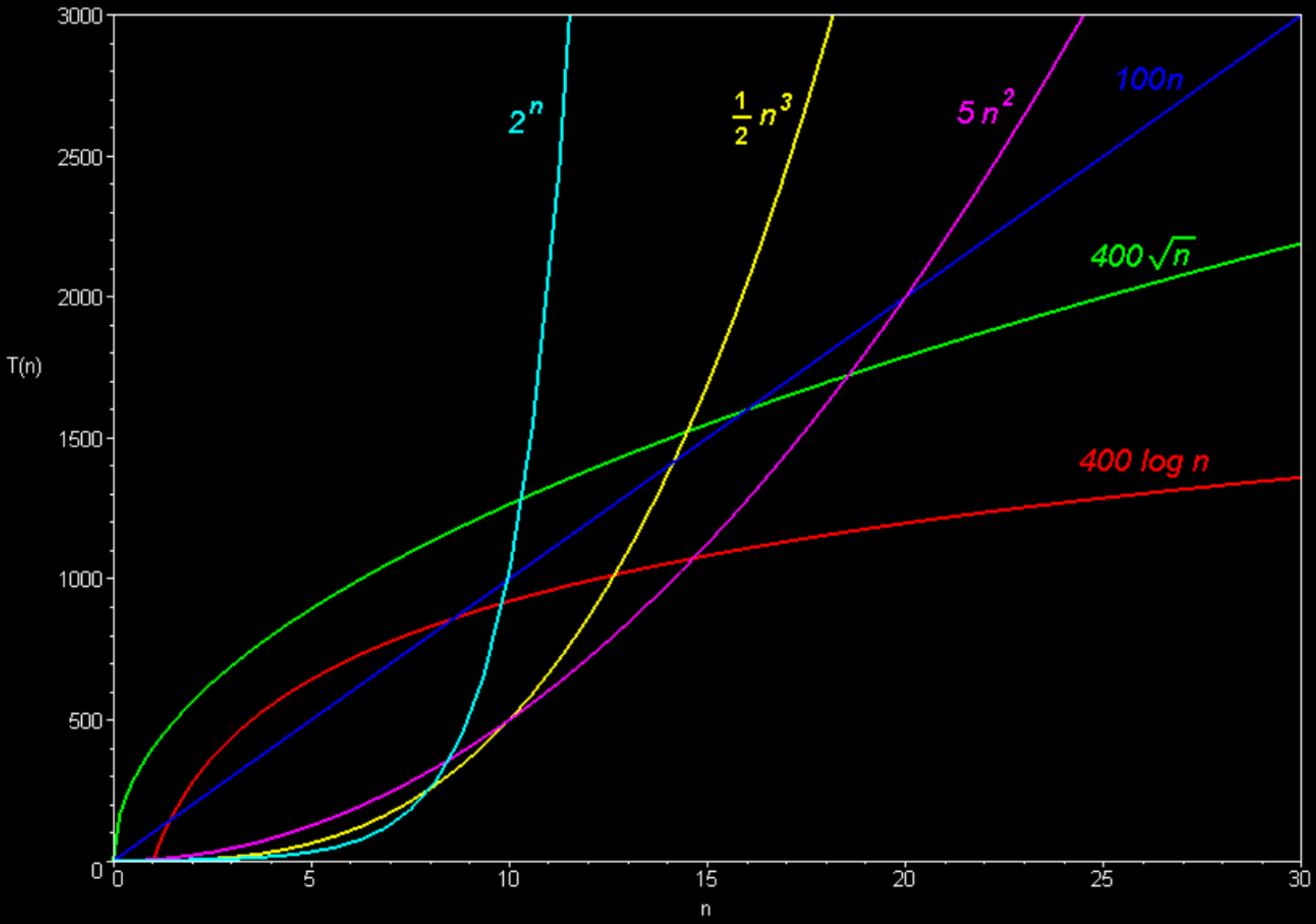
$O(n \log n)$

COMPARING GROWTH RATES

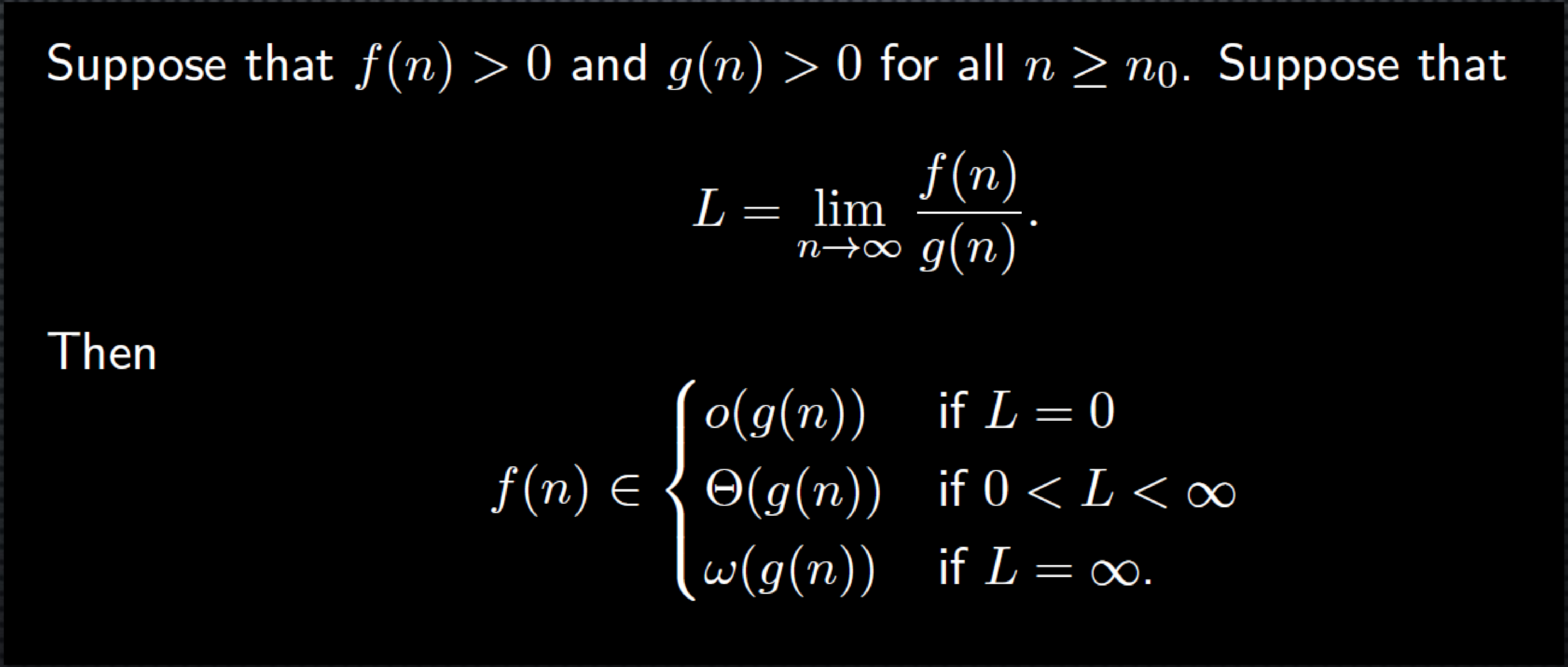
Some Common Growth Rates (in increasing order)

polynomial
- $\Theta(1)$
- $\Theta(\log n)$
- $\Theta(\sqrt{n})$
- $\Theta(n)$
- $\Theta(n^2)$
- $\Theta(n^c)$

exponential
- $\Theta(1.1^n)$
- $\Theta(2^n)$
- $\Theta(e^n)$
- $\Theta(n!)$
- $\Theta(n^n)$



LIMIT TECHNIQUE
FOR COMPARING GROWTH RATES



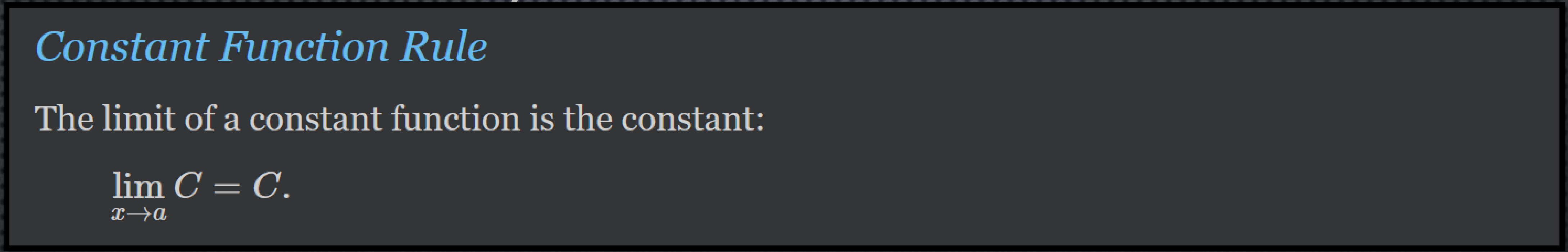
Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}.$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty. \end{cases}$$

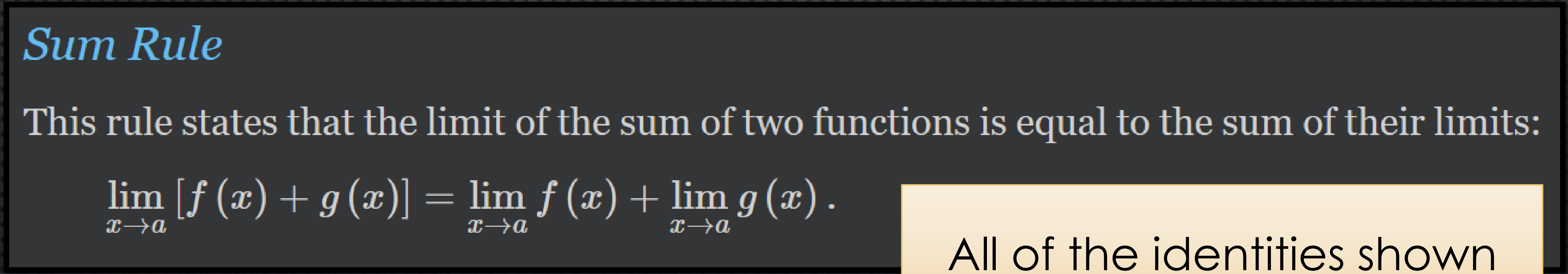
LIMIT RULES 1/3



Constant Function Rule

The limit of a constant function is the constant:

$$\lim_{x \to a} C = C.$$



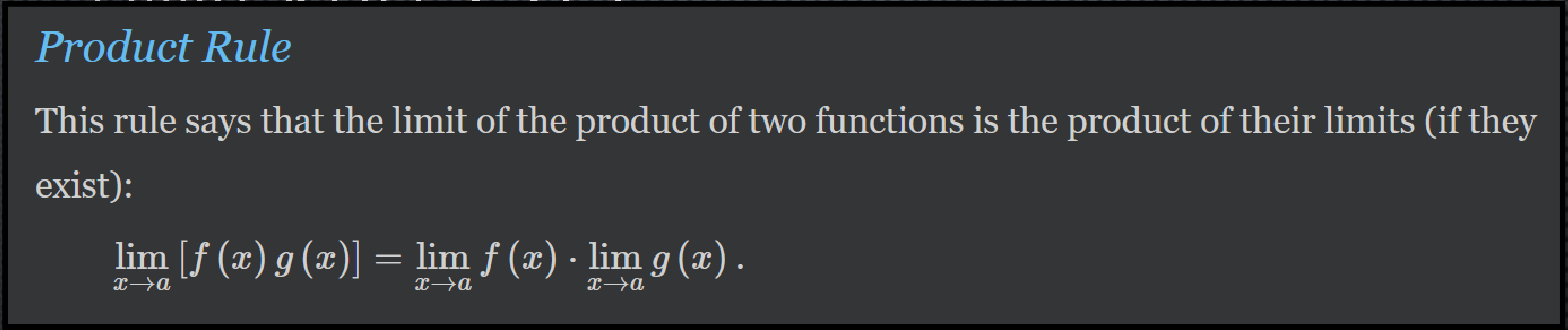
Sum Rule

This rule states that the limit of the sum of two functions is equal to the sum of their limits:

$$\lim_{x \to a} \left[f\left(x\right) + g\left(x\right) \right] = \lim_{x \to a} f\left(x\right) + \lim_{x \to a} g\left(x\right).$$

All of the identities shown hold **only if the limits exist**

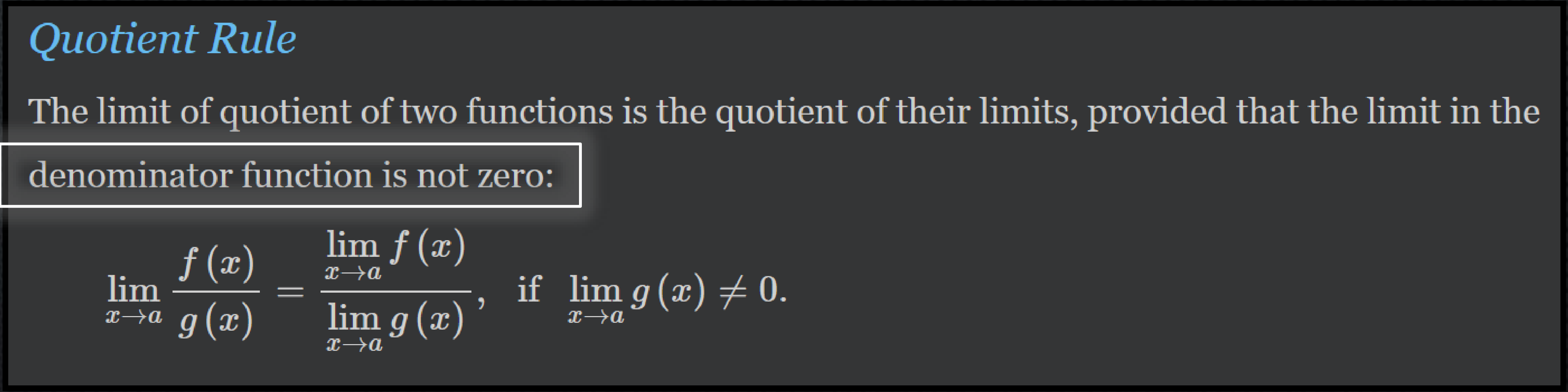
LIMIT RULES 2/3



Product Rule

This rule says that the limit of the product of two functions is the product of their limits (if they exist):

$$\lim_{x\to a}\left[f\left(x\right)g\left(x\right)\right]=\lim_{x\to a}f\left(x\right)\cdot\lim_{x\to a}g\left(x\right).$$

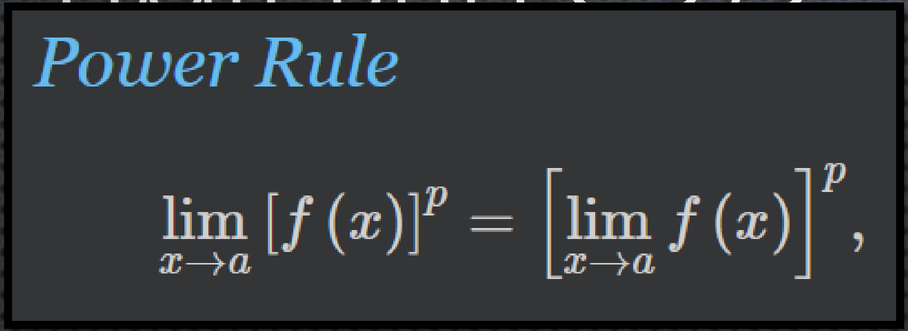


Quotient Rule

The limit of quotient of two functions is the quotient of their limits, provided that the limit in the denominator function is not zero:

$$\lim_{x\to a}\frac{f\left(x\right)}{g\left(x\right)}=\frac{\lim\limits_{x\to a}f\left(x\right)}{\lim\limits_{x\to a}g\left(x\right)},\quad\text{if}\quad\lim_{x\to a}g\left(x\right)\neq 0.$$

LIMIT RULES 2/2



Power Rule

$$\lim_{x\to a}\left[f\left(x\right)\right]^{p}=\left[\lim_{x\to a}f\left(x\right)\right]^{p},$$

Limit of an Exponential Function

$$\lim_{x\to a} b^{f(x)} = b^{\lim_{x\to a} f(x)}$$

Limit of a Logarithm of a Function

$$\lim_{x\to a} \log_b f(x) = \log_b \lim_{x\to a} f(x)$$

(Where base $b > 0$)

L'HOSPITAL'S RULE

- Often we take the limit of $\frac{f(n)}{g(n)}$ where both $f(n)$ and $g(n)$ tend to ∞, or both $f(n)$ and $g(n)$ tend to 0
- Such limits require L'Hospital's rule
  - This rule says the limit of $\boldsymbol{f(n)/g(n)}$ in this case is the same as the limit of the **derivative**
  - In other words, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{d}{dn}f(n)}{\frac{d}{dn}g(n)}$

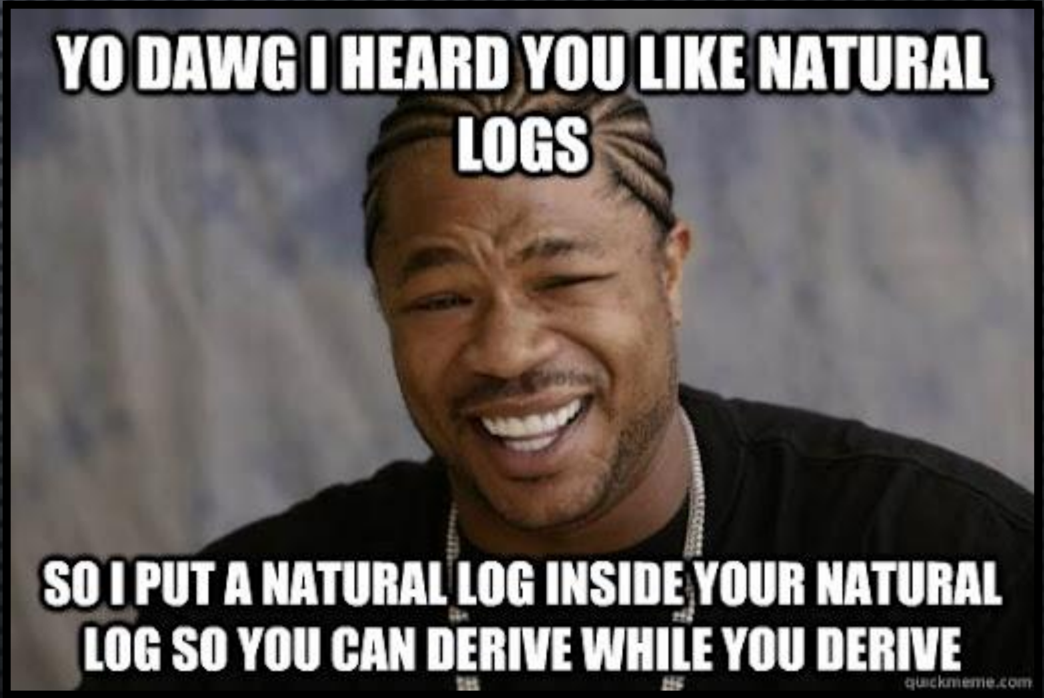
USING THE LIMIT METHOD: EXERCISE 1

- Compare growth rate of n^2 and $n^2 - 7n - 30$
- $\lim_{n\to\infty} \frac{n^2-7n-30}{n^2}$
- $= \lim_{n\to\infty} (1 - \frac{7}{n} - \frac{30}{n^2})$
- $= 1$
- So $n^2 - 7n - 30 \in \Theta(n^2)$

USING THE LIMIT METHOD: EXERCISE 2

- Compare growth rate of $(\ln n)^2$ and $n^{1/2}$

- $\lim_{n\to\infty} \frac{(\ln n)^2}{n^{1/2}} = \lim_{n\to\infty} \frac{\frac{d}{dn}(\ln n)^2}{\frac{d}{dn}n^{1/2}}$



USING THE LIMIT METHOD: EXERCISE 2

- Compare growth rate of $(\ln n)^2$ and $n^{1/2}$

- $\lim_{n\to\infty} \frac{\frac{d}{dn}(\ln n)^2}{\frac{d}{dn}n^{1/2}}$
- $= \lim_{n\to\infty} \frac{2\ln n\,(1/n)}{\frac{1}{2}n^{-1/2}}$
- $= \lim_{n\to\infty} \frac{4\ln n}{n^{1/2}}$
- $= \lim_{n\to\infty} \frac{\frac{d}{dn}4\ln n}{\frac{d}{dn}n^{1/2}}$
- $= \lim_{n\to\infty} \frac{4/n}{\frac{1}{2}n^{-1/2}}$
- $= \lim_{n\to\infty} \frac{8}{n^{1/2}}$
- $= 0$
- So, $(\ln n)^2 \in o(n^{1/2})$

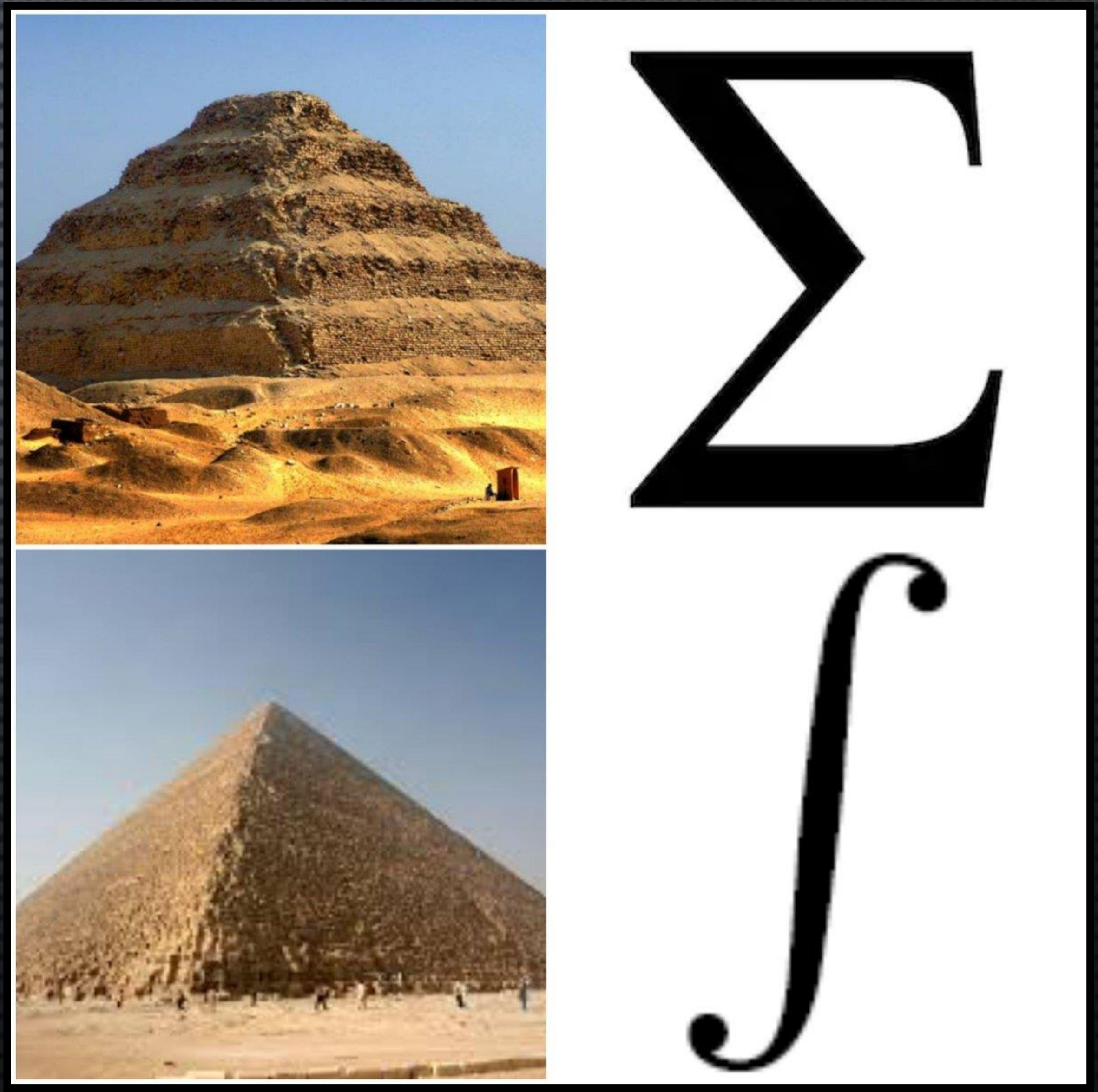
Try these at home...

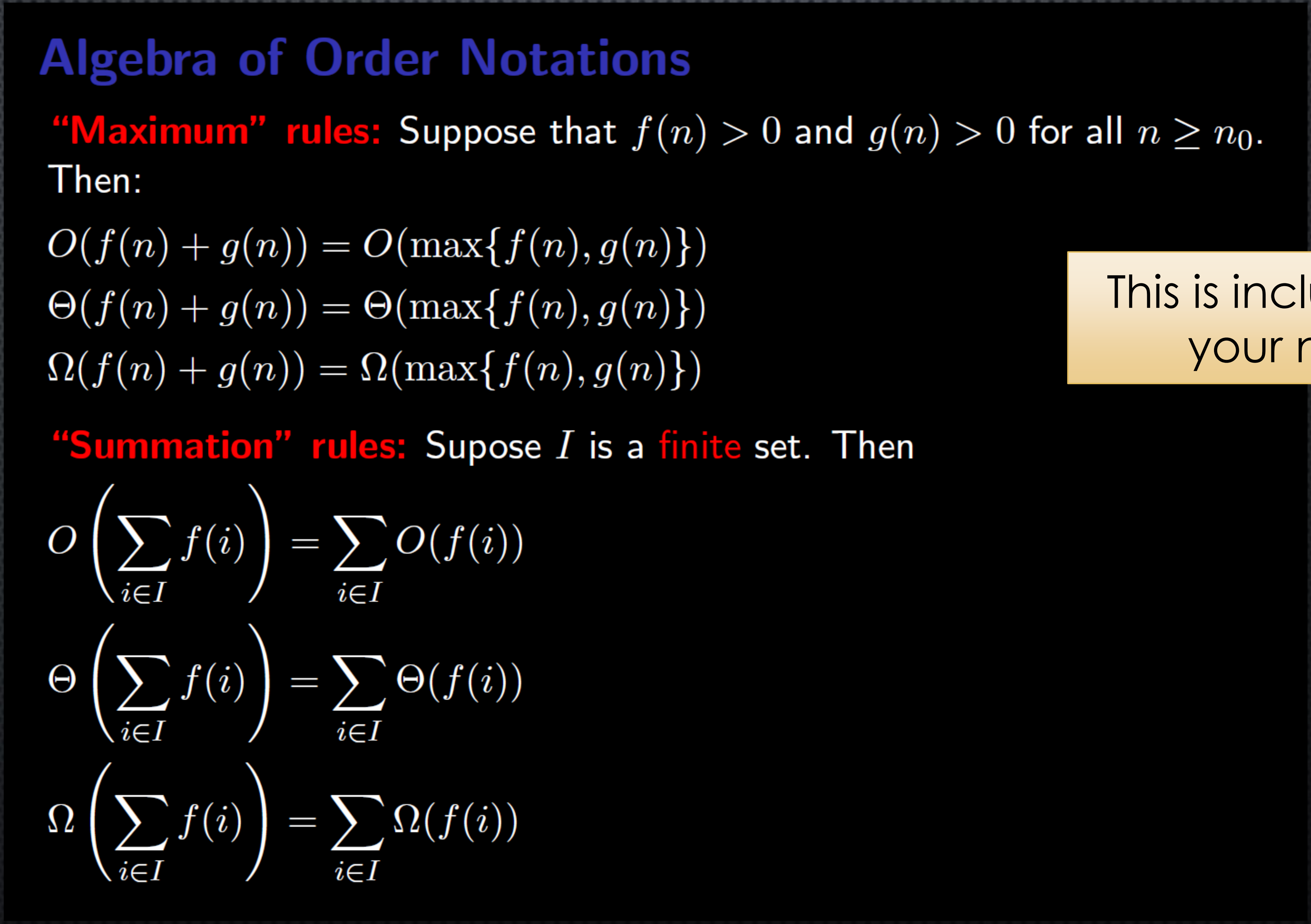
Additional Exercises

1. Compare the growth rate of the functions $(3+(-1)^n)n$ and n.

2. Compare the growth rates of the functions $f(n) = n\,|\sin \pi n/2| + 1$ and $g(n) = \sqrt{n}$.

SUMMATIONS AND SEQUENCES





Algebra of Order Notations

"Maximum" rules: Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Then:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$
$$\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$$
$$\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$$

"Summation" rules: Supose I is a finite set. Then

$$O\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} O(f(i))$$
$$\Theta\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Theta(f(i))$$
$$\Omega\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Omega(f(i))$$

This is included for your notes

Summation rules are commonly used in loop analysis.

Example:

$$\begin{aligned}\sum_{i=1}^{n} O(i) &= O\left(\sum_{i=1}^{n} i\right) \\ &= O\left(\frac{n(n+1)}{2}\right) \\ &= O(n^2).\end{aligned}$$

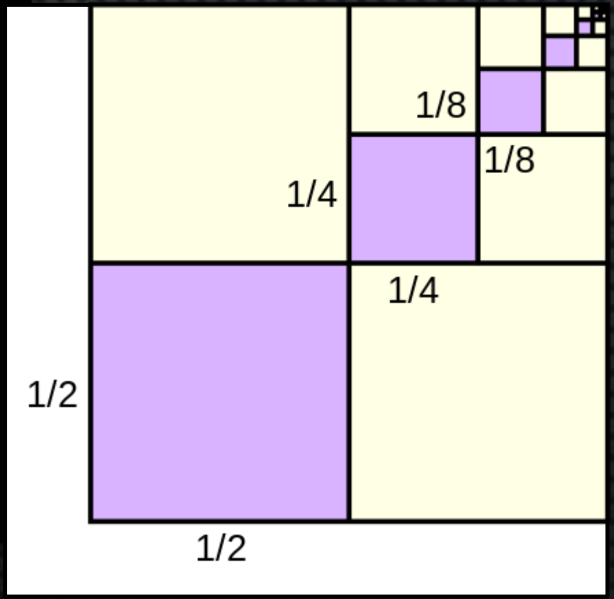
SEQUENCES

Arithmetic sequence:

$$\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2).$$

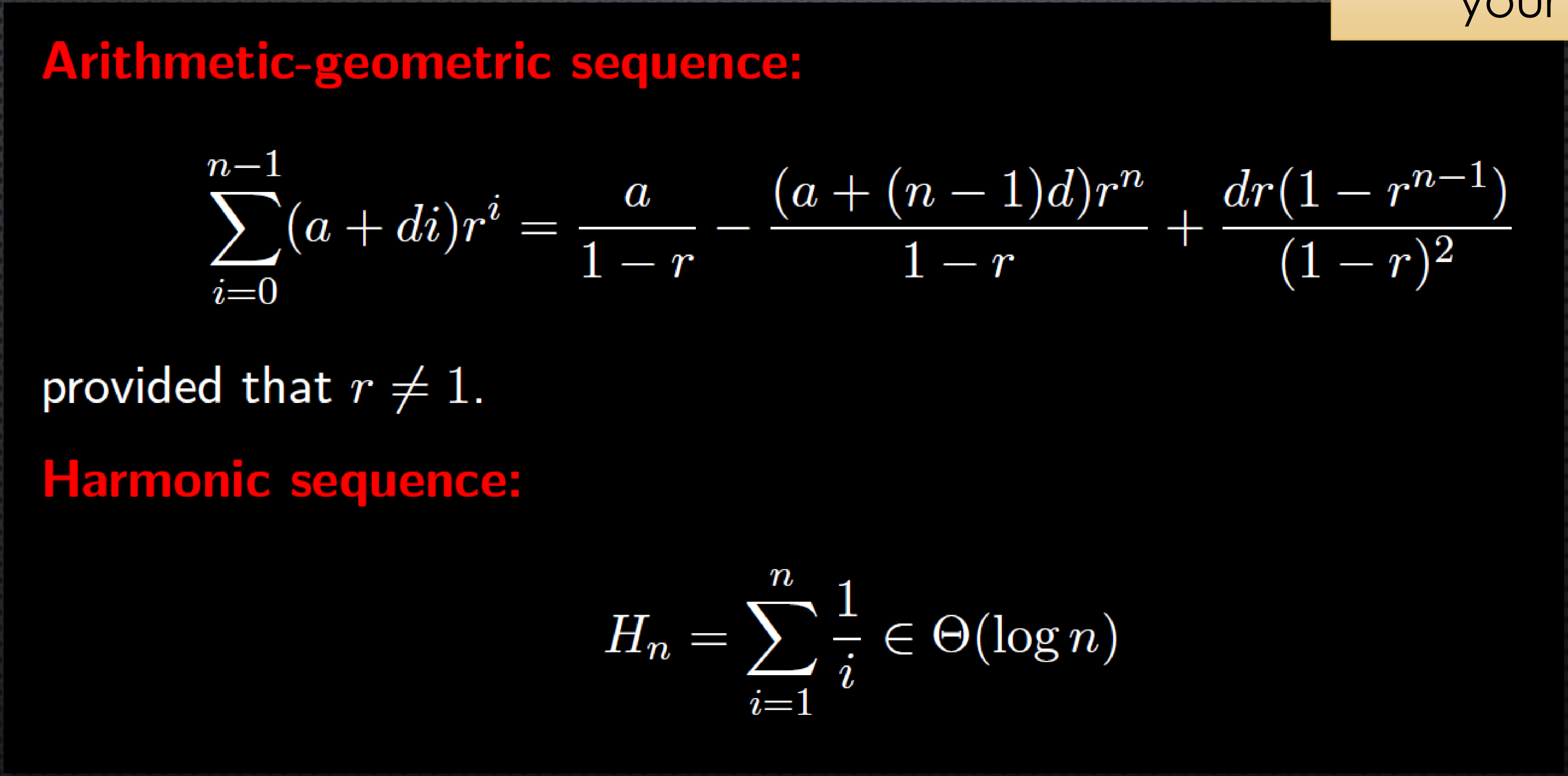
Geometric sequence:

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1. \end{cases}$$



SEQUENCES CONTINUED

This is included for your notes



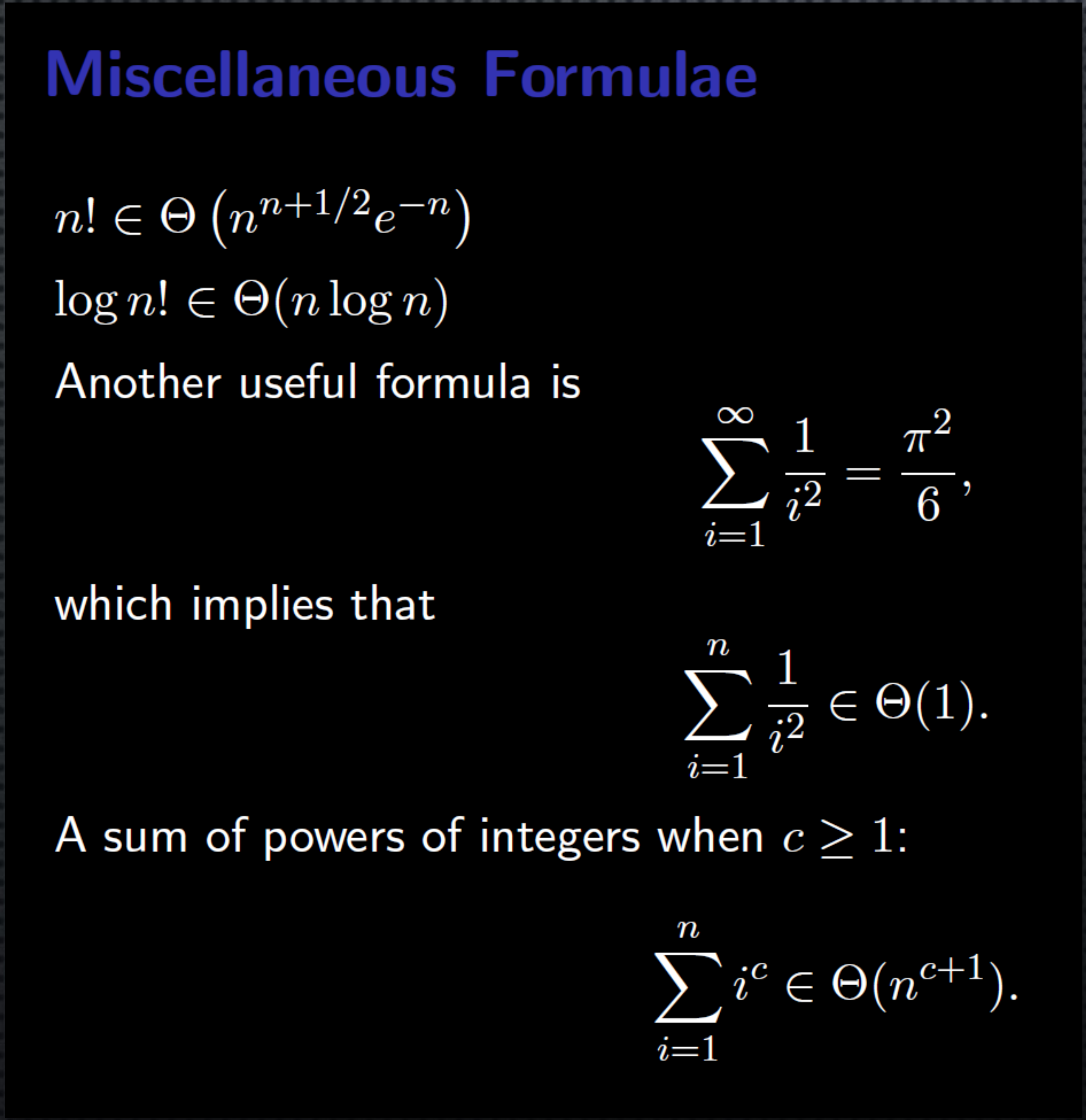
Arithmetic-geometric sequence:

$$\sum_{i=0}^{n-1}(a+di)r^i = \frac{a}{1-r} - \frac{(a+(n-1)d)r^n}{1-r} + \frac{dr(1-r^{n-1})}{(1-r)^2}$$

provided that $r \neq 1$.

Harmonic sequence:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$



Miscellaneous Formulae

$n! \in \Theta\left(n^{n+1/2}e^{-n}\right)$

$\log n! \in \Theta(n \log n)$

Another useful formula is

$$\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6},$$

which implies that

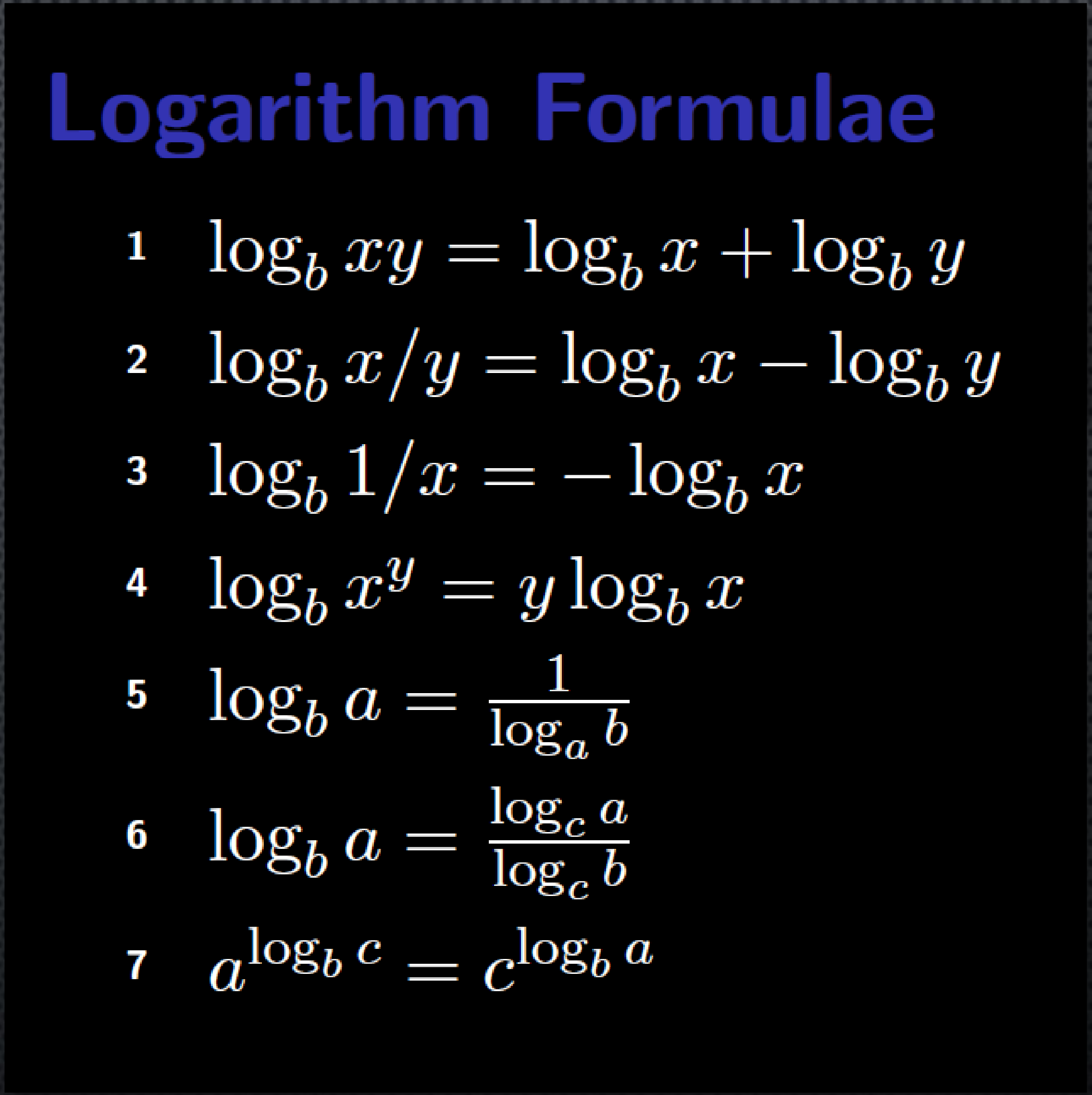
$$\sum_{i=1}^{n} \frac{1}{i^2} \in \Theta(1).$$

A sum of powers of integers when $c \geq 1$:

$$\sum_{i=1}^{n} i^c \in \Theta(n^{c+1}).$$

This is included for your notes

LOGARITHM RULES



Logarithm Formulae

1. $\log_b xy = \log_b x + \log_b y$
2. $\log_b x/y = \log_b x - \log_b y$
3. $\log_b 1/x = -\log_b x$
4. $\log_b x^y = y \log_b x$
5. $\log_b a = \dfrac{1}{\log_a b}$
6. $\log_b a = \dfrac{\log_c a}{\log_c b}$
7. $a^{\log_b c} = c^{\log_b a}$

BASE OF LOGARITHM **DOES NOT MATTER**!

- Big-O notation does not distinguish between log bases
- Proof:
  - Fix two constant logarithm bases b and c
  - From log rules, we can change from $\log_c$ to $\log_b$ by using formula: $\log_\mathrm{b} x = \log_c x / \log_c b$
  - But $\log_c b$ is a **constant!**
  - So $\log_c x \in \Theta(\log_b x)$

We typically omit the base,
and just write $\boldsymbol{\Theta(\log x)}$
for this reason

LOOP ANALYSIS

META-ALGORITHM FOR ANALYZING LOOPS

- Identify operations that require only constant time
- The complexity of a **loop** is the **sum** of the complexities of **all iterations**
- Analyze independent loops separately and add the results
- If loops are nested, it often helps to start at the innermost, and proceed outward… but,
  - sometimes you must express several nested loops together in a single equation (using nested summations),
  - and actually evaluate the nested summations… (can be hard)

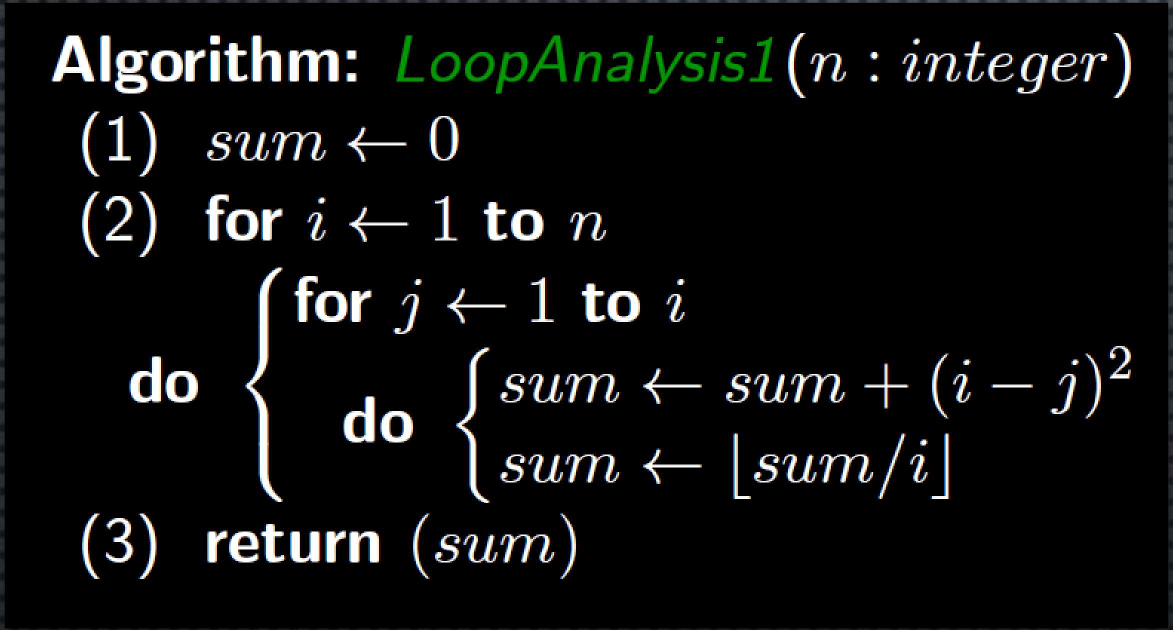
TWO BIG-O ANALYSIS STRATEGIES

- **Strategy 1**
  - Prove a O-bound and a matching Ω-bound separately to get a Θ -bound.

Often easier
(but not always)

- **Strategy 2**
  - Use Θ-bounds throughout the analysis and thereby obtain a Θ-bound for the complexity of the algorithm

EXAMPLE 1



```
**Algorithm:** LoopAnalysis1$(n : integer)$
$sum \leftarrow 0$
for $i \leftarrow 1$ to $n$
    do
        for $j \leftarrow 1$ to $i$
            do
                $sum \leftarrow sum + (i - j)^2$
                $sum \leftarrow \lfloor sum/i \rfloor$
return $(sum)$
```

Strategy 1: big-O and big-Ω bounds

```
Algorithm: LoopAnalysis1($(n : integer)$)
(1)  $sum \leftarrow 0$
(2)  for $i \leftarrow 1$ to $n$
        do { for $j \leftarrow 1$ to $i$
               do { $sum \leftarrow sum + (i - j)^2$
                    $sum \leftarrow \lfloor sum/i \rfloor$
(3)  return $(sum)$
```

We focus on the two nested **for** loops (i.e., (2)).

The total number of iterations is $\sum_{i=1}^{n} i$, with $\Theta(1)$ time per

Upper bound:

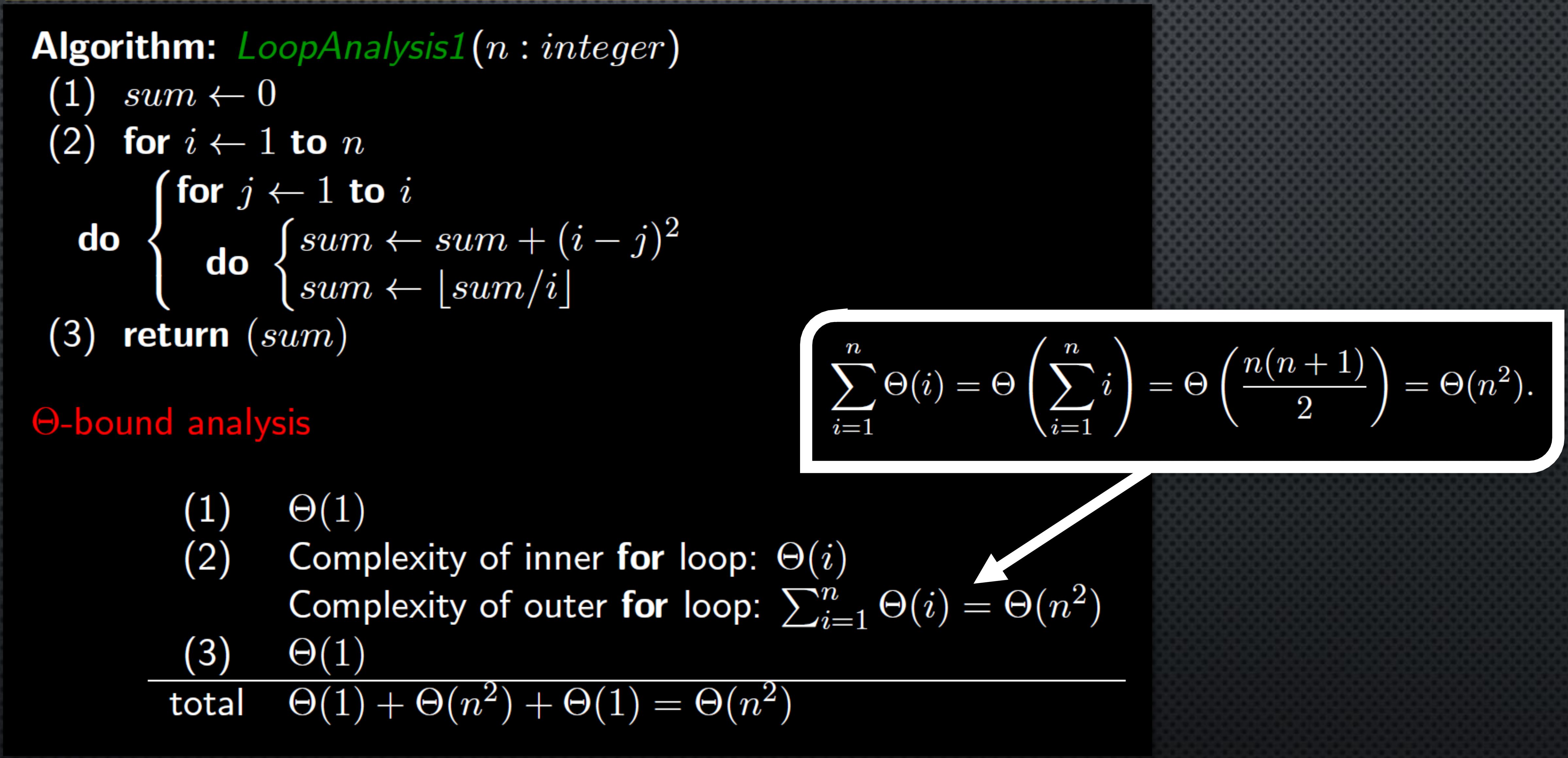
$$\sum_{i=1}^{n} O(i) \leq \sum_{i=1}^{n} O(n) = O(n^2).$$

Lower bound:

$$\sum_{i=1}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(n/2) = \Omega(n^2/4) = \Omega(n^2).$$

Since the upper and lower bounds **match**, the complexity is $\Theta(n^2)$.

Strategy 2: use Θ-bounds throughout the analysis



Algorithm: *LoopAnalysis1*$(n : integer)$

```
(1) $sum \leftarrow 0$
(2) for $i \leftarrow 1$ to $n$
    do {
        for $j \leftarrow 1$ to $i$
            do {
                $sum \leftarrow sum + (i-j)^2$
                $sum \leftarrow \lfloor sum/i \rfloor$
            }
    }
(3) return $(sum)$
```

Θ-bound analysis

(1)	$\Theta(1)$
(2)	Complexity of inner **for** loop: $\Theta(i)$
	Complexity of outer **for** loop: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
(3)	$\Theta(1)$
total	$\Theta(1) + \Theta(n^2) + \Theta(1) = \Theta(n^2)$

$$\sum_{i=1}^{n} \Theta(i) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2).$$

EXAMPLE 2

```
sum := 0;
for i := 1 to n do
  j := i;
  while j >= 1 do
    sum:= sum + i/j;
    j := floor(j/2);
print(sum)
```

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$O(1)$

Consider this loop alone… number of loop iterations?

j starts at i and is repeatedly divided by 2… this can happen only $\boldsymbol{\Theta(\log i)}$ **times**

So inner loop has runtime $\boldsymbol{\Theta(\log i)}$

And the entire inner loop is executed for $i = 1, 2, \ldots, n$

So, we have $\boldsymbol{T(n) \in \Theta\left(\sum_{i=1}^{n} \log i\right)}$

$$T(n) \in O\left(\sum_{i=1}^{n} \log i\right) \subseteq O\left(\sum_{i=1}^{n} \log n\right) \subseteq \boldsymbol{O(n \log n)}$$

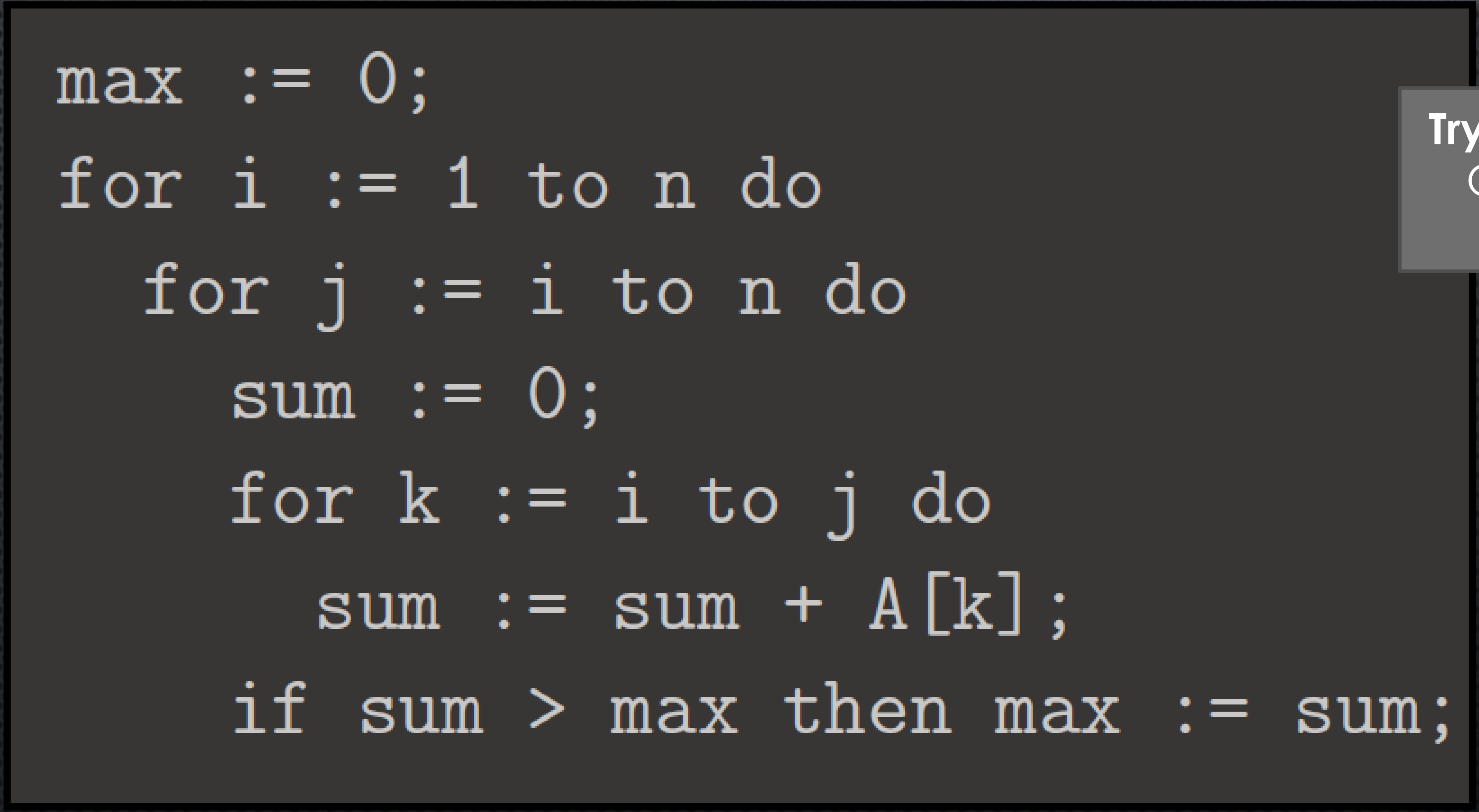
$$T(n) \in \Omega\left(\sum_{i=1}^{n} \log i\right) \subseteq \Omega\left(\sum_{i=\frac{n}{2}}^{n} \log \frac{n}{2}\right) \subseteq \boldsymbol{\Omega(n \log n)}$$

… *ANOTHER* EXERCISE IN LOOP ANALYSIS?

Olive Garden waiter: Sir, you've already had 5 baskets of breadsticks

Me:

EXAMPLE 3 (BENTLEY'S PROBLEM, SOLUTION 1)



```
max := 0;
for i := 1 to n do
  for j := i to n do
    sum := 0;
    for k := i to j do
      sum := sum + A[k];
    if sum > max then max := sum;
```

Try to analyze this yourself!
One possible solution is
given in these slides…

Strategy 1: big-O and big-Ω bounds

```
max := 0;
for i := 1 to n do
  for j := i to n do
    sum := 0;
    for k := i to j do
      sum := sum + A[k];
    if sum > max then max := sum;
```

$O(1)$

$\sum_{i=1}^{n} \dots$

$O(1)$

$O(1)$

$O(1)$

$$T(n) \in \Theta(1) + \sum_{i=1}^{n}\sum_{j=i}^{n}\left(\Theta(1) + \sum_{k=i}^{j}\Theta(1) + \Theta(1)\right)$$

$$T(n) \in \sum_{i=1}^{n}\sum_{j=i}^{n}\boldsymbol{\Theta(j-i)} \in \boldsymbol{\Theta}\left(\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)\right)$$

$$T(n) \in \boldsymbol{O}\left(\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)\right) \le O\left(\sum_{i=1}^{n}\sum_{j=i}^{n}\boldsymbol{n}\right)$$

$$\le O\left(\sum_{i=1}^{n}\sum_{\boldsymbol{j=1}}^{n} n\right)$$

$$\boldsymbol{T(n) \in O(n^3)}$$

This is the **maximum number of iterations** that could be performed in this loop

Proving a big-Ω bound…

```
max := 0;
for i := 1 to n do
  for j := i to n do
    sum := 0;
    for k := i to j do
      sum := sum + A[k];
    if sum > max then max := sum;
```

Recall:

$$T(n) \in \mathbf{\Theta}\left(\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)\right)$$

$$T(n) \in \mathbf{\Omega}\left(\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)\right)$$

$$\geq \Omega\left(\sum_{i=1}^{\boldsymbol{n/2}}\sum_{j=i}^{n}(j-i)\right)$$

$$\geq \Omega\left(\sum_{i=1}^{n/2}\sum_{j=\boldsymbol{3n/4}}^{n}(j-i)\right)$$

Intuition: $j - i$ is $\mathbf{\Omega(n)}$ in **some** iterations. How many iterations? Lots?

To get a good Ω-bound, we ask questions like: When do our **loops** have **<u>many</u> iterations**? When is our **dominant term <u>large</u>**?

Many iterations: when our $\boldsymbol{j}$ **loop does** $\mathbf{\Omega(n)}$ **iterations!** For example, when $\boldsymbol{i \leq n/2}$…

Large dominant term: when $\boldsymbol{j}$ **is much larger than** $\boldsymbol{i}$ (i.e., by a factor of n)

Proving a big-Ω bound… **continued**

```
max := 0;
for i := 1 to n do
   for j := i to n do
      sum := 0;
      for k := i to j do
         sum := sum + A[k];
      if sum > max then max := sum;
```

Recall: $T(n) \in \Omega\left(\sum_{i=1}^{n/2} \sum_{j=3n/4}^{n} (j-i)\right)$

$$\geq \Omega\left(\sum_{i=1}^{n/2} \sum_{j=3n/4}^{n} \left(\frac{3n}{4} - \frac{n}{2}\right)\right)$$

$$= \Omega\left(\sum_{i=1}^{n/2} \sum_{j=3n/4}^{n} n/4\right)$$

$$\geq \Omega\left(\frac{n}{2} \cdot \frac{n}{4} \cdot \frac{n}{4}\right) = \Omega(n^3)$$

Smallest possible value of $j - i$ for these bounds on $\boldsymbol{i, j}$

We will perform **at least this much** work in **every** iteration!

This term does **not** depend on the loop indexes, so just **multiply** by the total number of loop iterations…

Since we have $O(n^3)$ and $\Omega(n^3)$, we have **proved** $\boldsymbol{\Theta(n^3)}$

BONUS

- Study-song of the day
- Tool - Descending
- youtu.be/PcSoLwFisaw