EXAMPLE: BINARY RECURSION – FIBONACCI

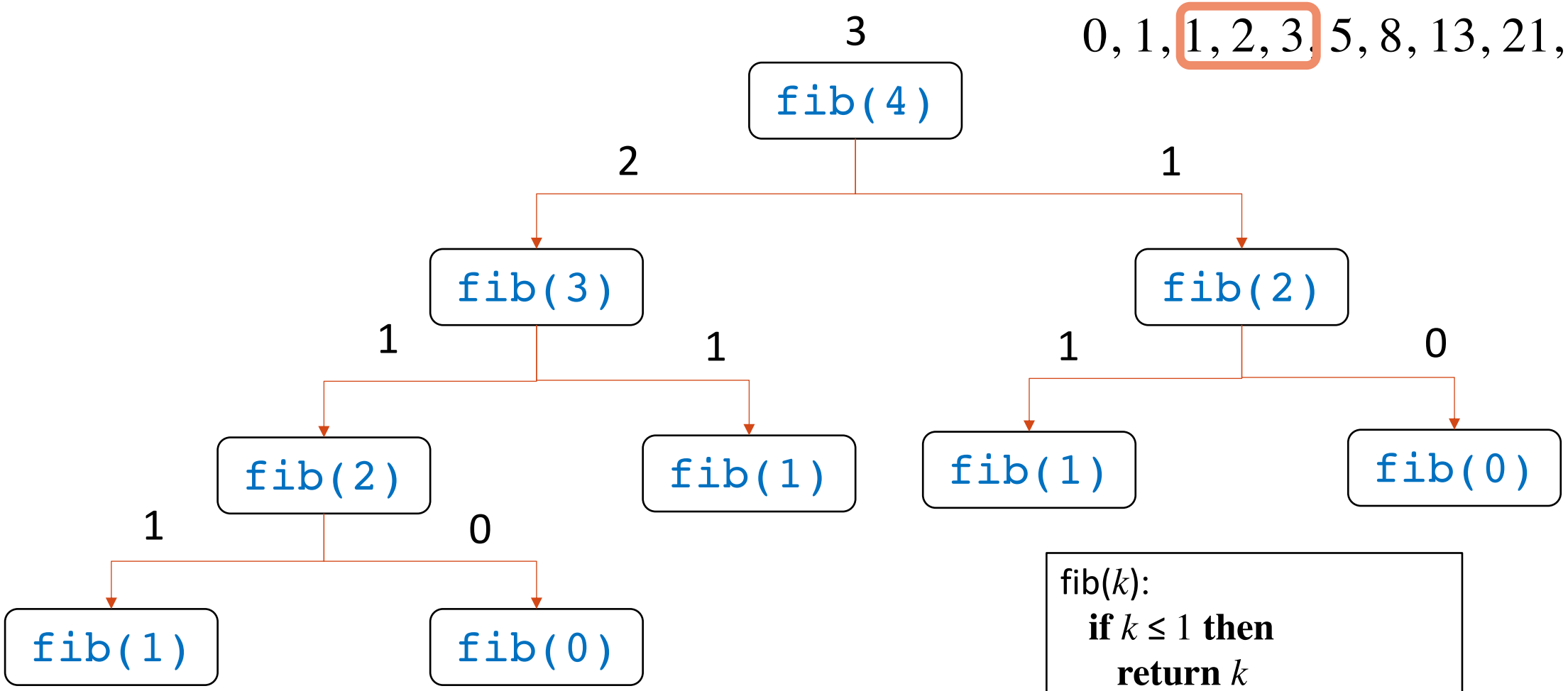
- Consider the problem of computing the k^{th} Fibonacci number.
- Recall that the Fibonacci numbers are recursively defined as:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_i = F_{i-1} + F_{i-2}$ for $i > 1$ $\quad 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$
- By directly applying this definition, we get this implementation:

Algorithm fib(k):
Input: Nonnegative integer k
Output: The k^{th} Fibonacci number F_k

```
if k ≤ 1 then
    return k
else
    return fib(k−1) + fib(k−2)
```

EXAMPLE: BINARY RECURSION – FIBONACCI

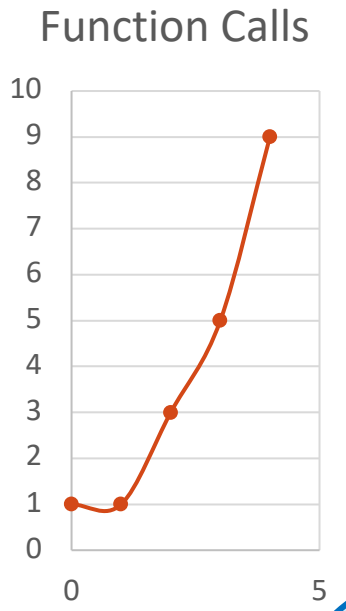
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



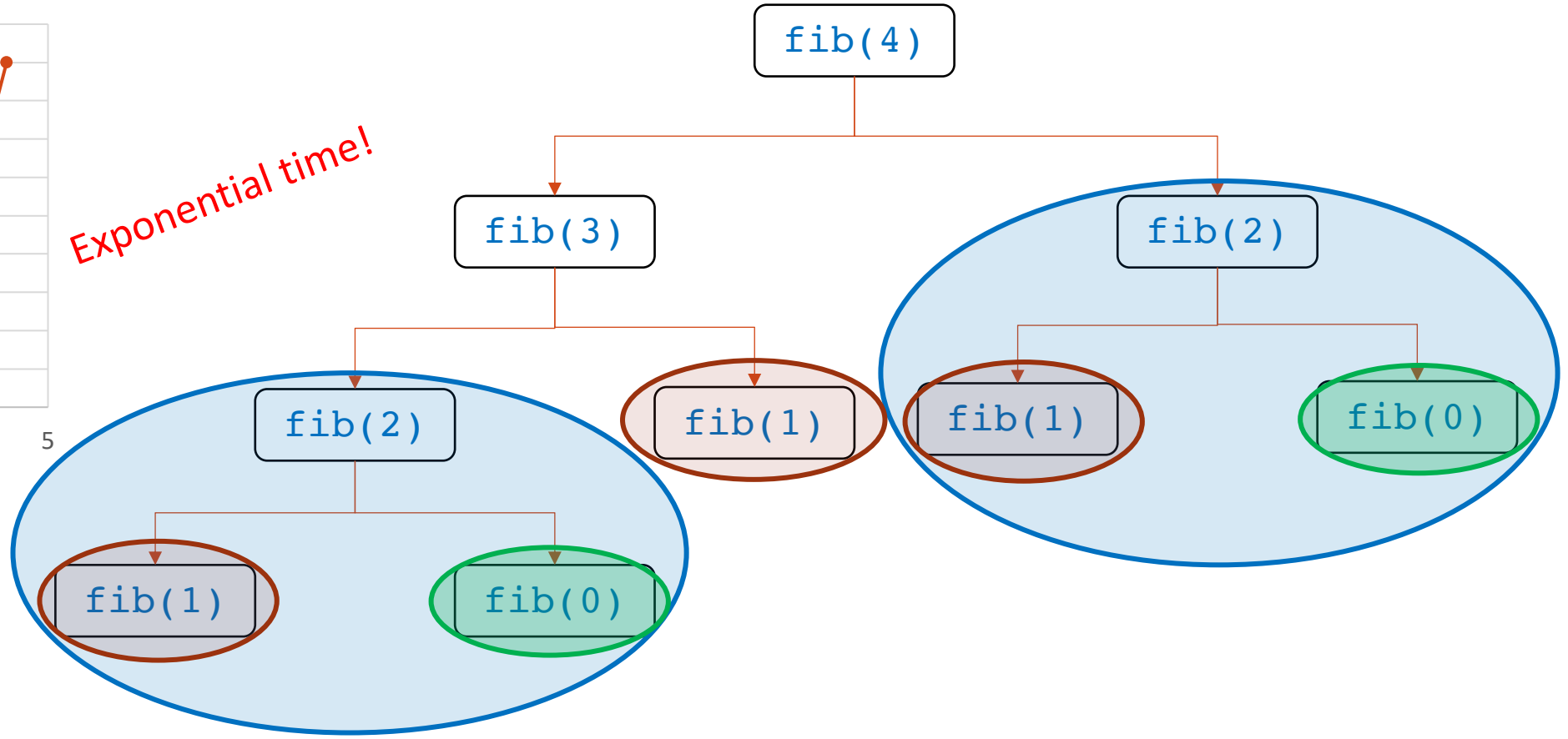

```
fib(k):
  if k ≤ 1 then
    return k
  else
    return f(k−1) + f(k−2)
```

EXAMPLE: BINARY RECURSION – FIBONACCI

- The binary implementation of computing F_k is inefficient
- *Do you notice the problem?* **Operations are being repeated!**



Exponential time!



EXAMPLE: FIBONACCI WITH LINEAR RECURSION

- Clearly, the binary-recursion implementation was inspired by the recursive formula of Fibonacci numbers: $F_i = F_{i-1} + F_{i-2}$ for $i > 1$

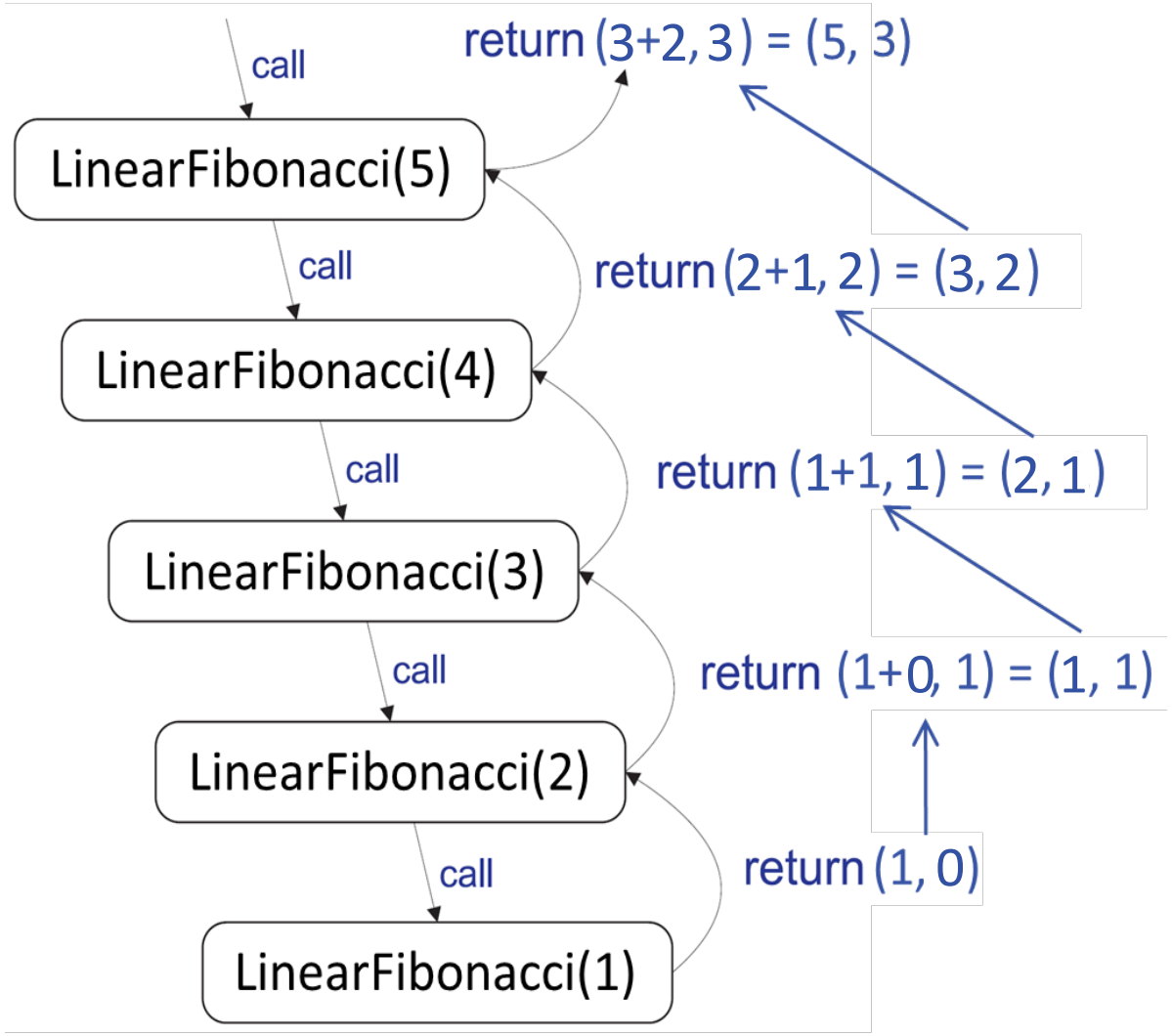
- However, we can compute F_k more efficiently using **linear recursion!**

  - To do this, define a recursive function that computes *a pair of consecutive Fibonacci numbers* (F_k, F_{k-1})

Algorithm LinearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k, F_{k-1})
    if $k \leq 1$ **then**
      return $(k,0)$
    else
      $(i, j) \leftarrow$ LinearFibonacci($k-1$)
      return $(i+j, i)$

EXAMPLE: FIBONACCI WITH LINEAR RECURSION

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

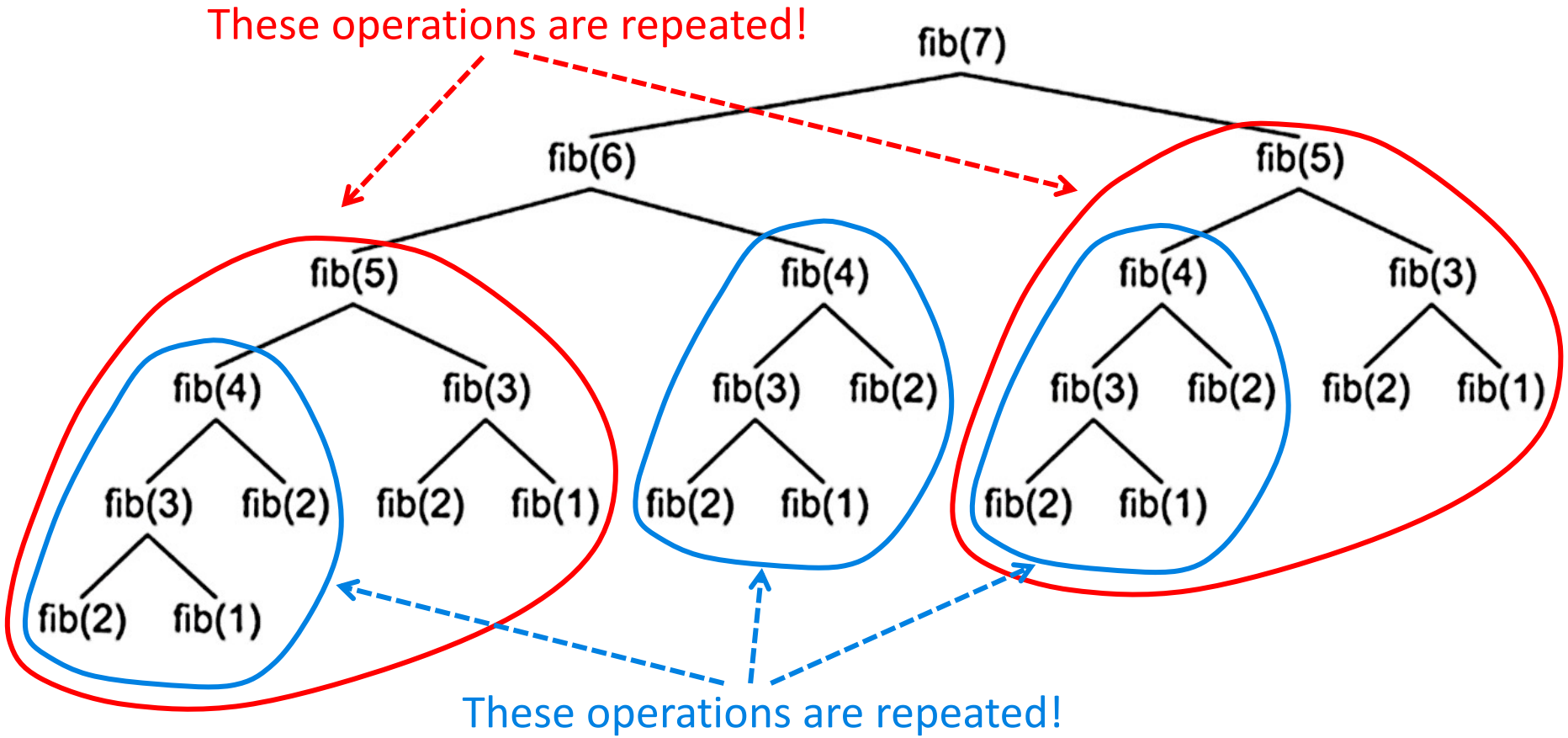


Algorithm LinearFibonacci(k):
Input: A nonnegative integer k
Output: Pair of Fibonacci numbers (F_k, F_{k-1})
if $k \leq 1$ **then**
return $(k,0)$
else
$(i, j) \leftarrow$ LinearFibonacci($k-1$)
return $(i+j, i)$

LinearFibonacci(k) requires only k−1 recursive calls!

LESSONS FROM THE FIBONACCI EXAMPLE

- When designing an algorithm, **avoid needlessly repeating the same operations** over and over!



LESSONS FROM THE FIBONACCI EXAMPLE

- It's not enough to design an algorithm that works **correctly**, it needs to work **efficiently**! Your design can make a huge difference to the runtime!

Algorithm fib(k):
  Input: Nonnegative integer k
  Output: The k^{th} Fibonacci number F_k
    if $k \leq 1$ **then**
      return k
    else
      return fib($k-1$) + fib($k-2$)

Computes Fibonacci in **exponential** time!

Algorithm LinearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k, F_{k-1})
    if $k \leq 1$ **then**
      return $(k,0)$
    else
      $(i, j) \leftarrow$ LinearFibonacci($k-1$)
      return $(i+j, i)$

Computes Fibonacci in **linear** time!

CHAPTER 4:
ALGORITHM ANALYSIS TOOLS

FUNCTIONS FOR ANALYSIS OF ALGORITHMS

1. The Constant Function
2. The Logarithm Function
3. The Linear Function
4. The N-Log-N Function
5. The Quadratic Function
6. The Cubic Function and Other Polynomials
7. The Exponential Function

THE CONSTANT FUNCTION

- This function is defined as:

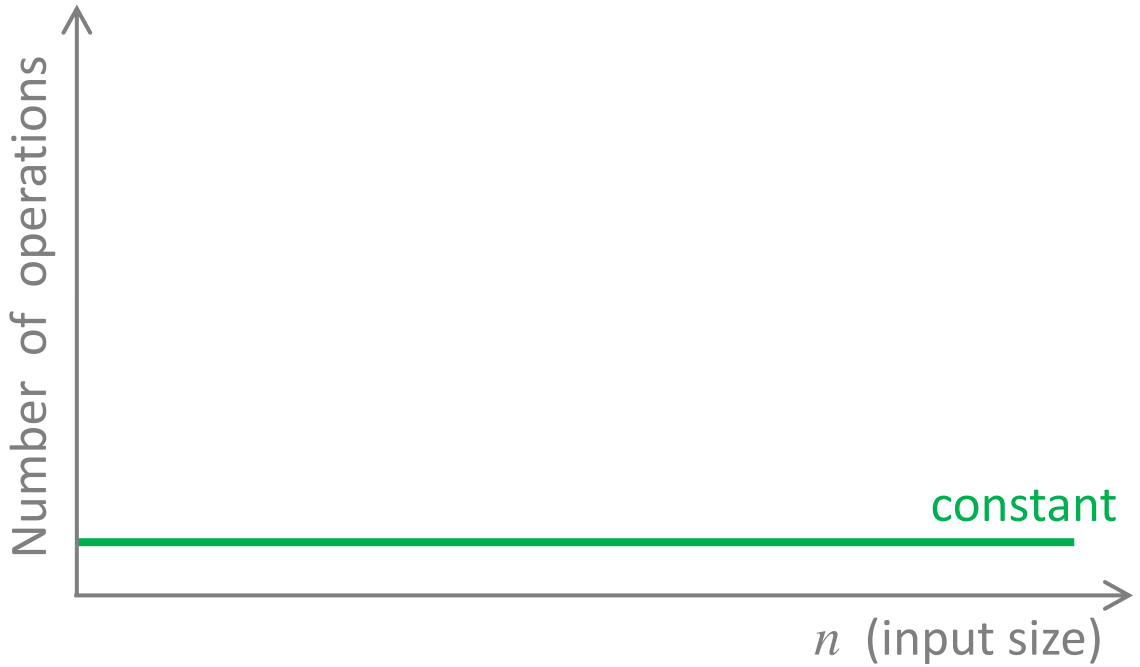
  $f(n) = c$

- The most fundamental constant function is:

  $g(n) = 1$

- Any other constant function, $f(n) = c$, can be written as a constant c times $g(n)$:

  $f(n) = cg(n)$



- For basic operations, e.g., addition or an assignment, the required number of operations is **constant**, i.e., independent of the **size of the input**, n.
  - E.g., `x = 5;` takes the same time as `x = 5296681223;`
  - E.g., `y = 5+6;` takes the same time as `y = 5296681220 + 6800741180;`

THE LOGARITHM FUNCTION

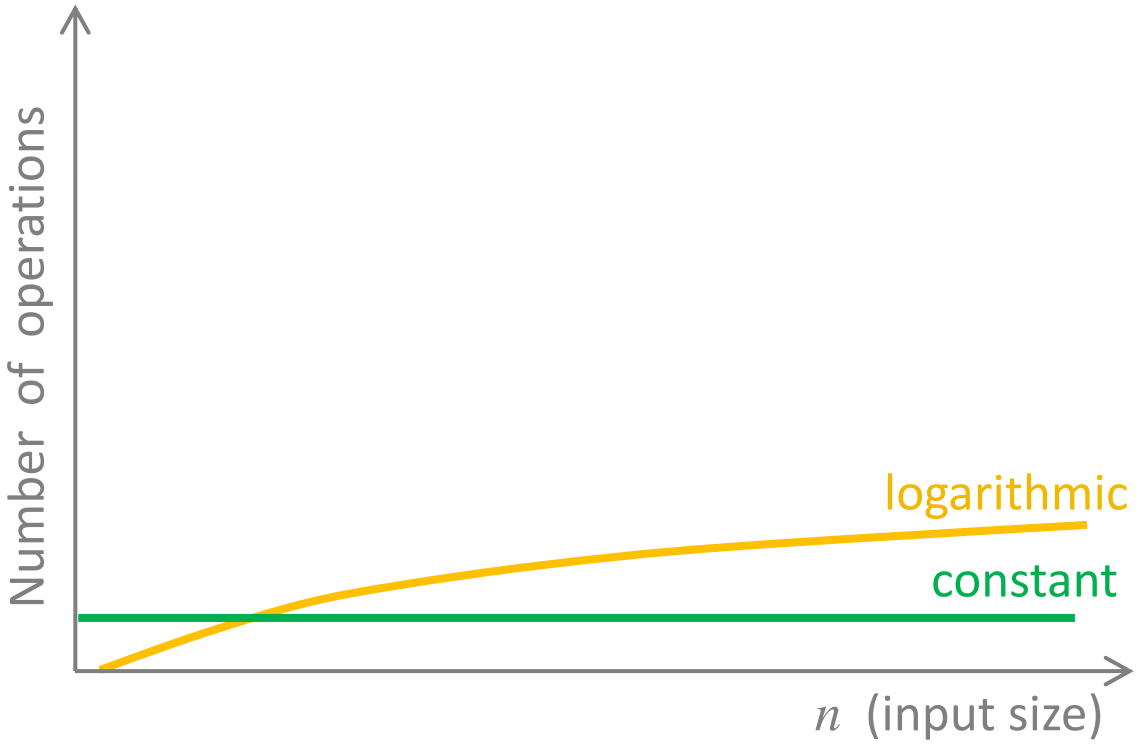
- This function is defined as:

  $f(n) = x = \log_b n$ if $b^x = n$

  b is the ***base*** of the logarithm

Example:

- Given $f(n) = \log_2 n$, x would be for:
  - $f(8) = 3$ (because $2^3 = 8$)
  - $f(16) = 4$ (because $2^4 = 16$)
  - $f(32) = 5$ (because $2^5 = 32$)



THE LOGARITHM FUNCTION

- Approximation can be used to compute $\log_b n$
  - It is equal to the number of times we can divide *n* by *b* until we get a number less than or equal to 1
  - E.g. $\log_3 27$ is 3, since 27/3/3/3 = 1
  - E.g. $\log_2 12$ is 4, since 12/2/2/2/2 = 0.75 ≤ 1
- Since computers store data in binary format, *the most common base used in computer science is **2***. Thus, by convention:

$$\log(n) = \log_2(n)$$

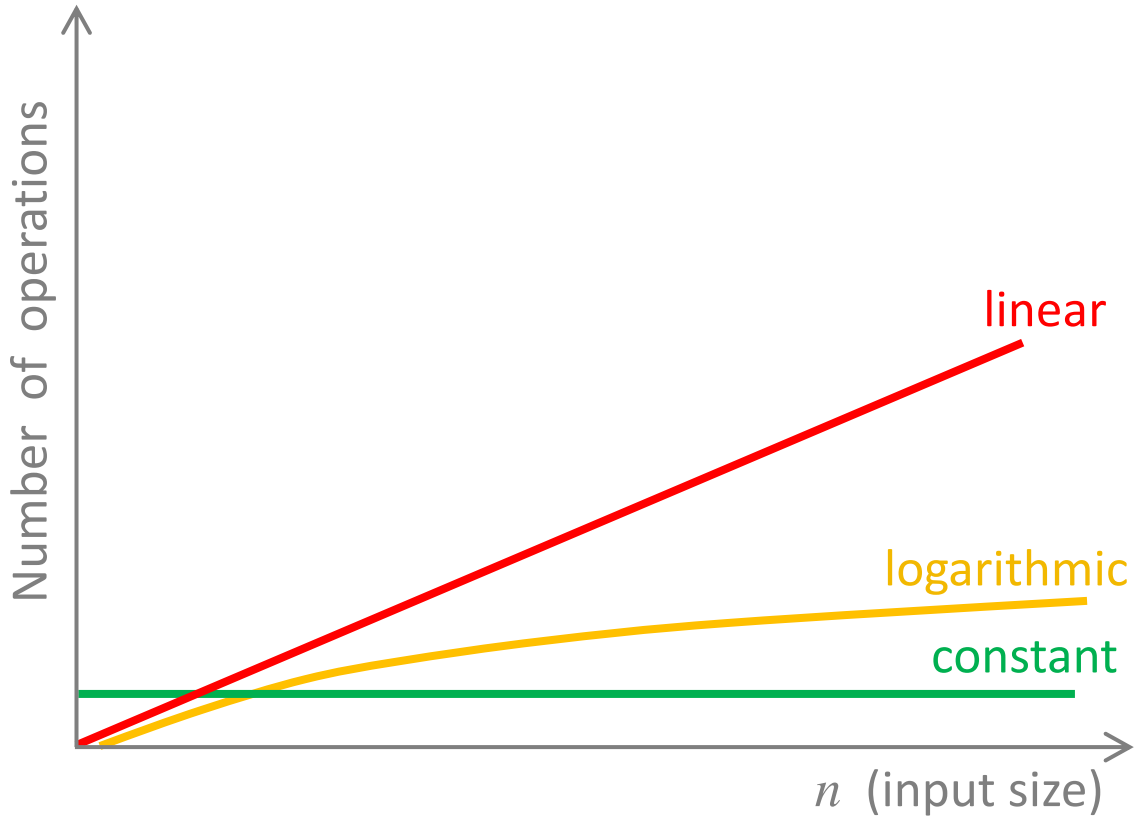
THE LINEAR FUNCTION

- This function is defined as:

$$f(n) = n$$

- E.g., comparing a number *x* to each element of an array of size *n* requires *n* comparisons.
- This function represents the best runtime we can hope to achieve when performing n iterations:

```
for(int i=0; i<n; i++){
  // code goes here...
}
```



FUNCTIONS FOR ANALYSIS OF ALGORITHMS

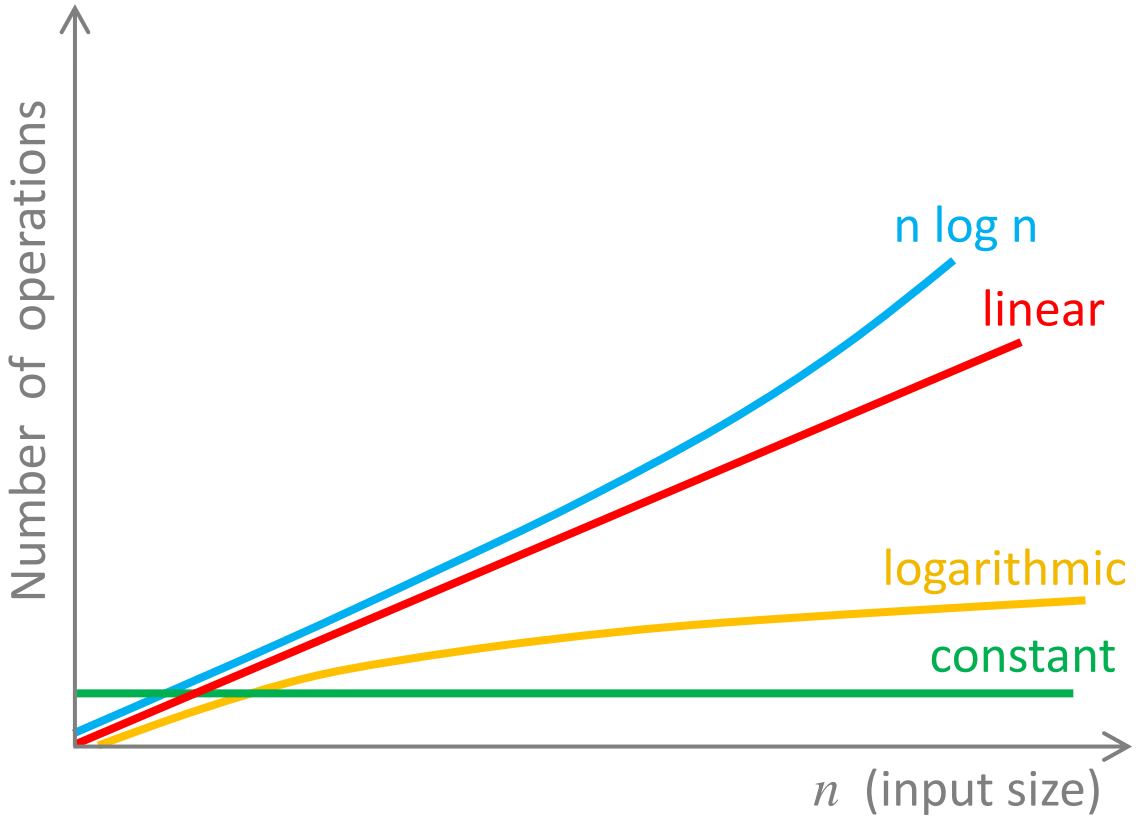
1. The Constant Function
2. The Logarithm Function
3. The Linear Function
4. The N-Log-N Function
5. The Quadratic Function
6. The Cubic Function and Other Polynomials
7. The Exponential Function

THE N-LOG-N FUNCTION

- This function is defined as:

$$f(n) = n \log n$$

- This function grows a little faster than the linear function and a lot slower than the quadratic function.



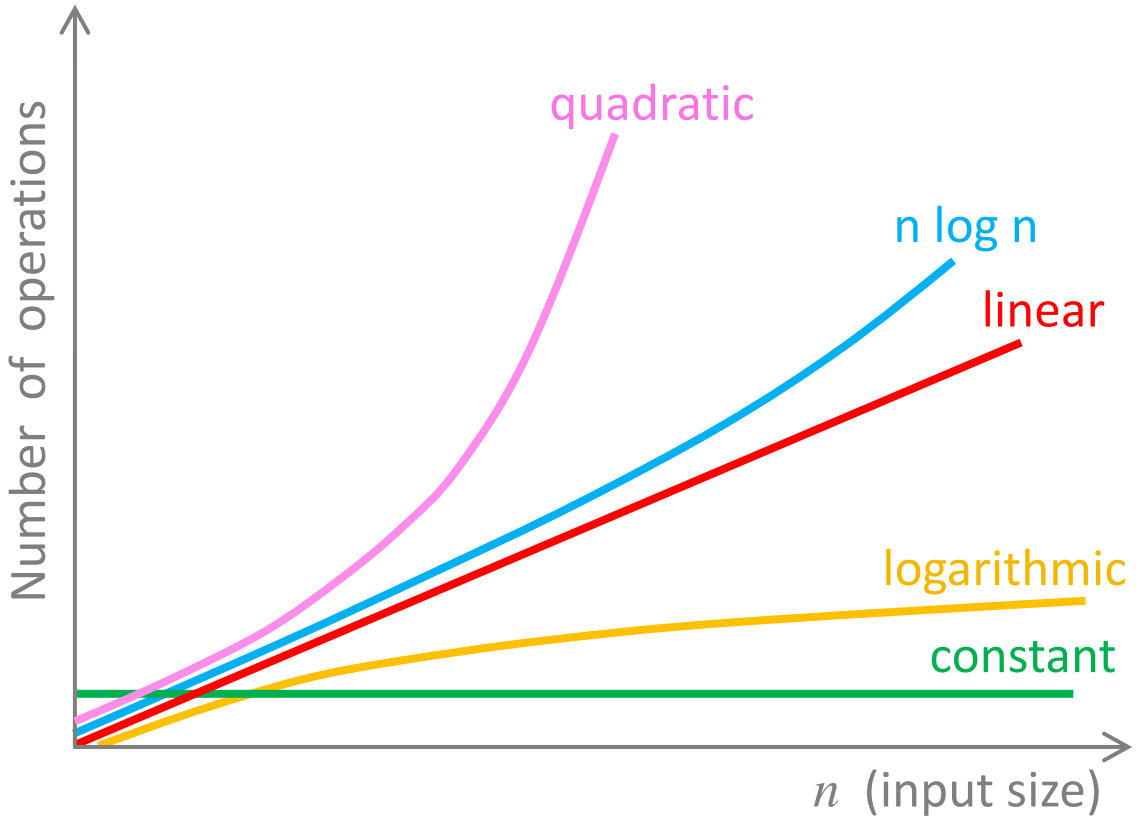
THE QUADRATIC FUNCTION

- This function is defined as:

$$f(n) = n^2$$

- This function represents the best runtime we can hope to achieve when performing n x n iterations:

```
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        //code goes here...
    }
}
```



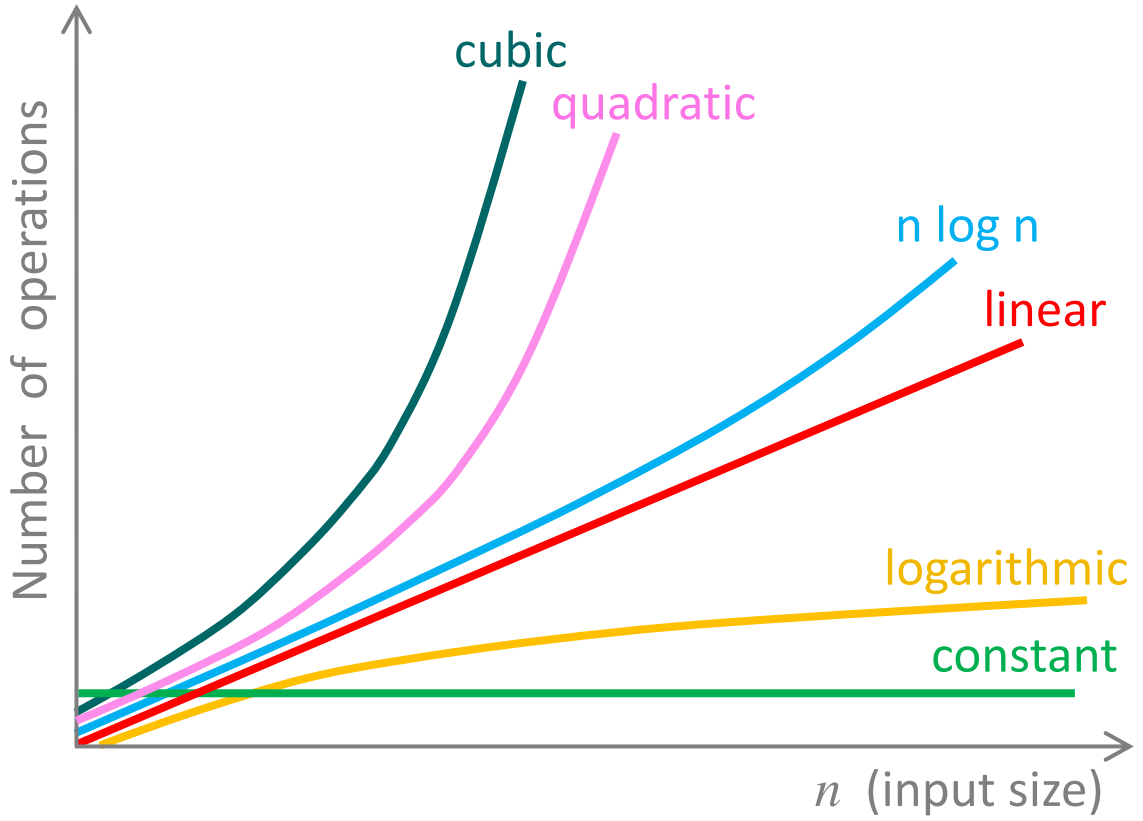
THE CUBIC FUNCTION

- This function is defined as:

$$f(n) = n^3$$

- This represents the best runtime we can hope to achieve when performing n x n x n iterations:

```
for(int i=0; i<n; i++){
  for(int j=0; j<n; j++){
    for(int k=0; k<n; k++){
      //code goes here...
    }
}
```



POLYNOMIALS

- A ***polynomial*** function is a function of the form

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d = \sum_{i=0}^{d} a_i n^i$$

  - ***Coefficients***: $a_0, a_1, \ldots, a_d$ are constants
  - ***Degree*** of the polynomial: Integer *d*, which indicates the highest power in the polynomial.

POLYNOMIALS

- Examples of polynomial functions:
  - $f(n) = 2 + 5n + n^2$
  - $f(n) = 1 + n^3$
  - $f(n) = 1$
  - $f(n) = n$
  - $f(n) = n^2$
- Thus, linear, quadratic and cubic functions are ***polynomials***.

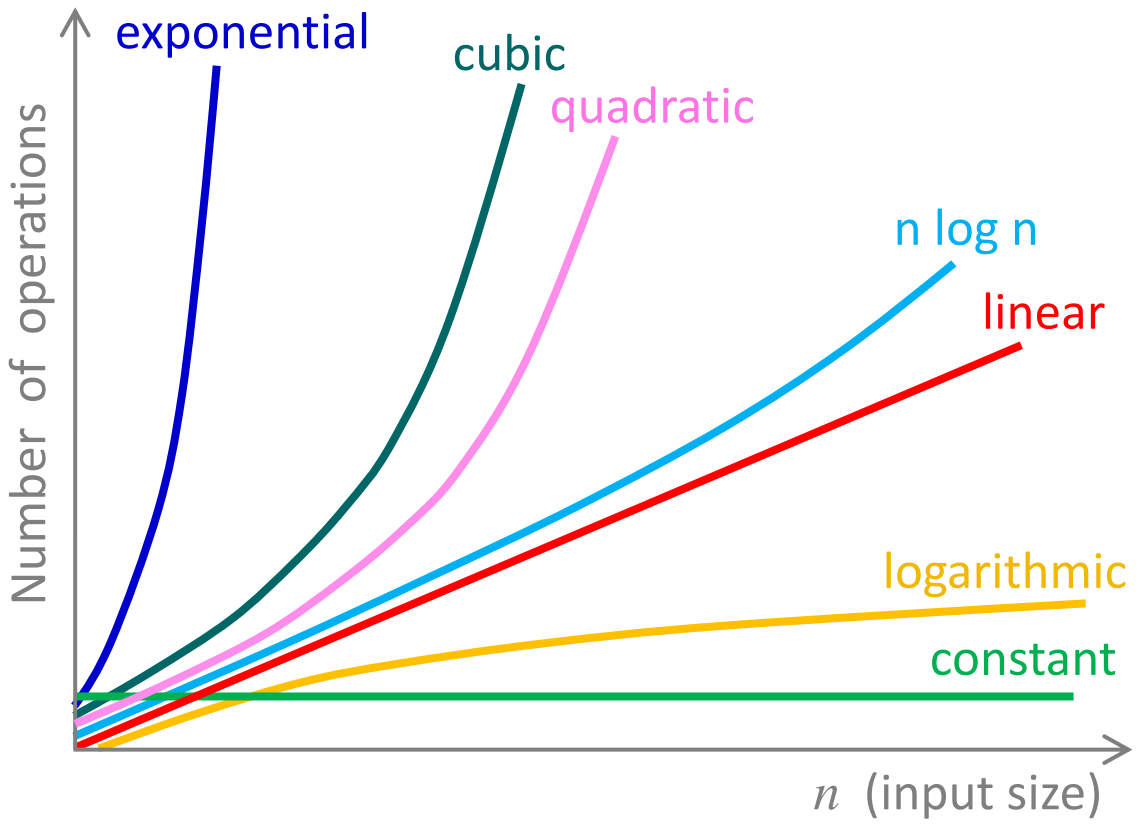
THE EXPONENTIAL FUNCTION

- This function is defined as:

$$f(n) = b^n$$

  - ***Base:*** *b* is a positive constant
  - ***Exponent:*** *n*

- In algorithm analysis, the most common base is $b = 2$



COMPARING GROWTH RATES

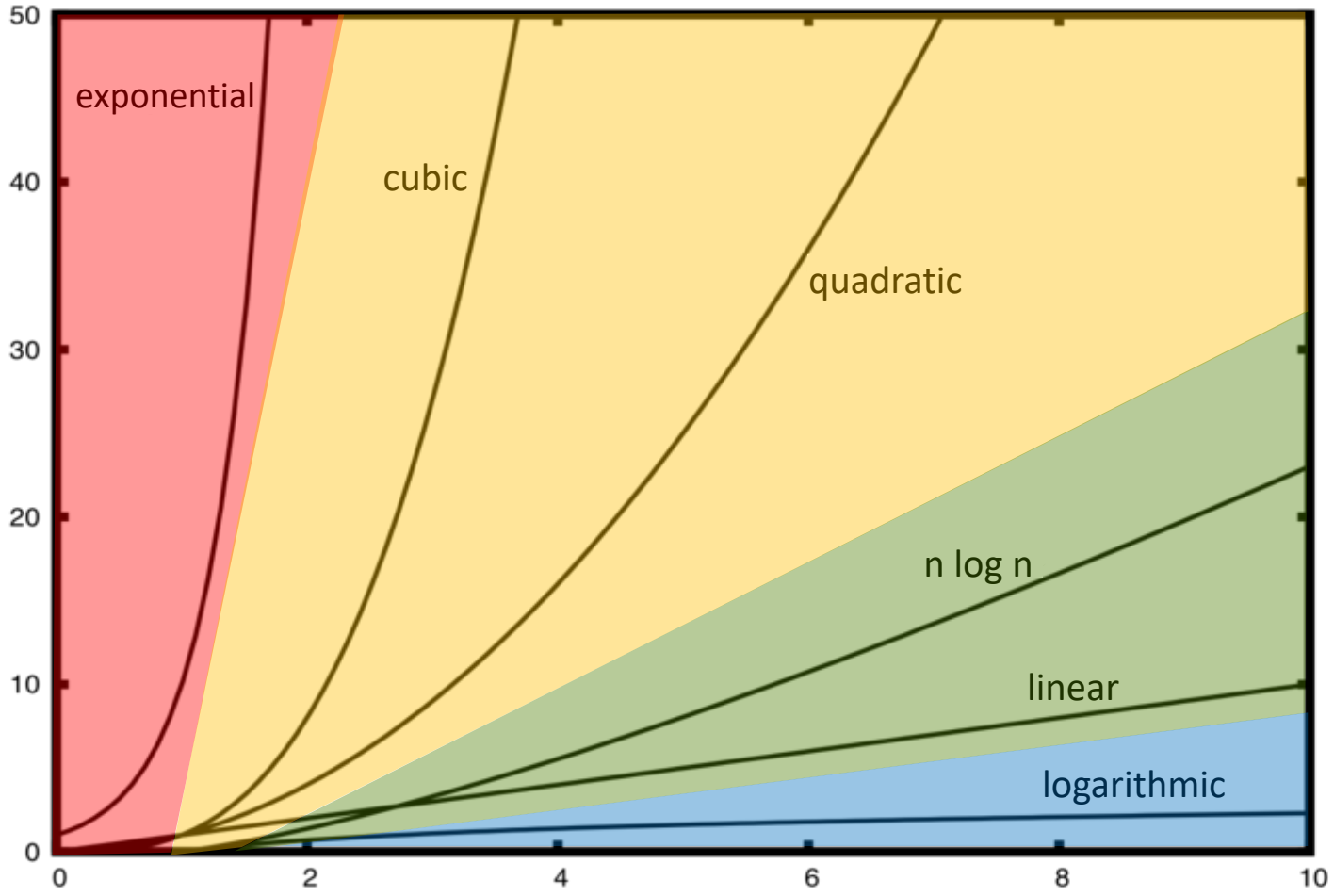
- The difference between the functions can be huge, especially if n is large!

  blue = instantaneous;

  green = fast;

  yellow = caution;

  red = super slow!



COMPARING GROWTH RATES

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

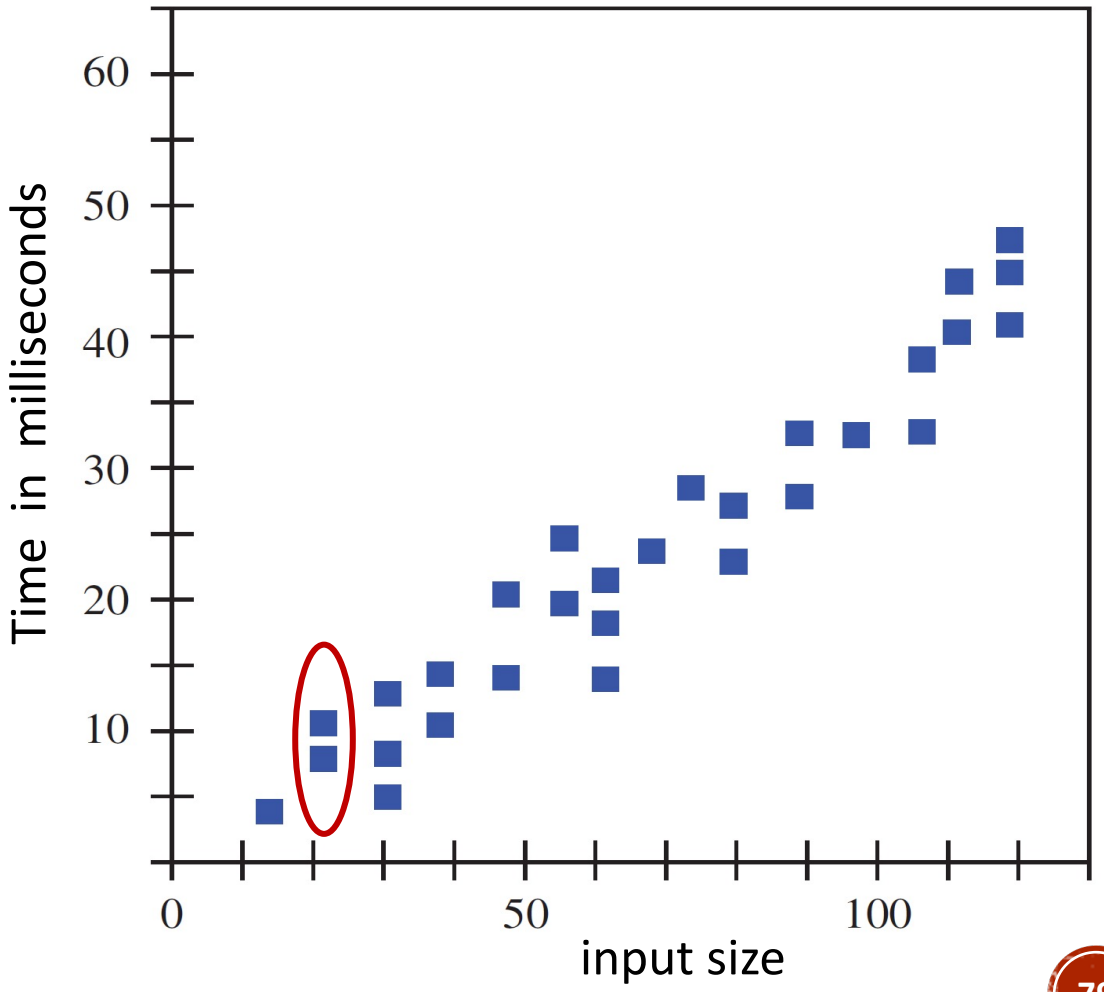
blue = instantaneous; **green** = fast; **yellow** = caution; **red** = super slow!

ANALYSIS OF ALGORITHMS

- Factors impacting the running time:
  - **Input size**
  - **Hardware environment** (processor, memory, etc.)
  - **Software environment** (e.g. operating system, programming language, compiler, interpreter, etc.)
- We will focus on the relationship between the running time of an algorithm and the **input size**

EXPERIMENTAL STUDIES

- One way to evaluate runtime is to test one algorithm on various inputs, and plot the runtime against input size
  - Note that a number of input instances might vary in runtime even if they have the same size!
  - ➢ This is because the two input instances may vary in other attributes apart from the size.



EXPERIMENTAL STUDIES

- While experimental studies of runtime are useful, they have some limitations:
  - Experiments can only be done on a **limited set of test inputs**
  - It is **hard to compare two algorithms** unless the experiments were performed in the same hardware and software environments
  - **We must implement the algorithm** in order to study its running time experimentally

THEORETICAL STUDIES

- To avoid the limitations of experimental studies, we can analyze the runtime mathematically (aka theoretically)
  - This takes into account **all possible input instances**
  - This allows us to **evaluate the relative efficiency of any two algorithms** in a way that is independent from hardware and software
  - This can be performed by studying a high-level description of the algorithm **without actually implementing it**
- It aims at associating, with each algorithm, a function $f(n)$ that characterizes the running time of the algorithm as a function of the input size n. Typically, $f(n)$ is one of the 7 functions we have seen.

COUNTING PRIMITIVE OPERATIONS

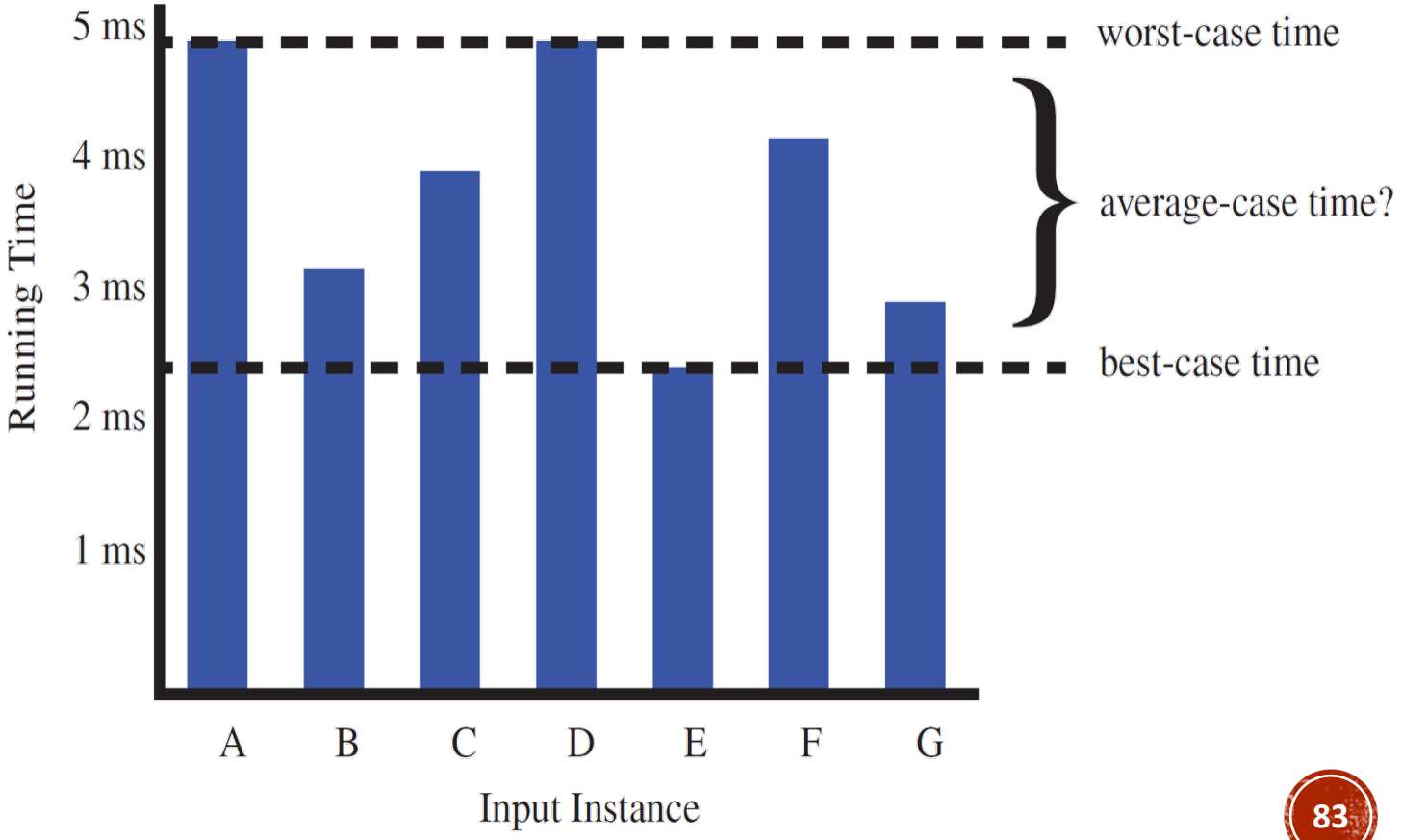
- Theoretical analysis involves counting "primitive" operations:
  - **Assigning** a value to a variable
  - **Comparing** two numbers
  - An **arithmetic operation** (e.g. adding two numbers)
  - **Calling a function**
  - **Indexing** into an array
  - Following an object **reference**
  - **Returning** from a function

Usually, theoretical analysis tends to focus on the first three.

FOCUSING ON THE AVERAGE CASE

- Theoretical analysis may express the runtime as a function of the input size by taking **the average over all possible inputs** of the same size.

- Unfortunately, such an average-case analysis is typically challenging, as it requires us to calculate the expected runtime based on a given input distribution, which usually *involves sophisticated probability theory.*



FOCUSING ON THE AVERAGE CASE

Example: what is the **average** number of primitive operations?

Algorithm Max(A, n)
  Input: An array A of n integers
    $max \leftarrow A[0]$
    for $i \leftarrow 1$ **to** $n-1$ **do**
      if $max < A[i]$ **then** n
        $max \leftarrow A[i]$

As can be seen, even for this very simple algorithm, it is not easy to figure out the average number over all possible inputs!

- The if-statement is performed n times for any A
- The number of assignments of max depends on A

Given this A, the number of assignments = 1:

10	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**

Given this A, the number of assignments = n:

10	20	30	40	50	60	70	80	90	100	110	120

Given this A, the number of assignments = $n/2$:

10	**10**	**10**	**10**	**10**	**10**	**10**	80	90	100	110	120

Given this A, the number of assignments = $n/2$:

10	20	**10**	40	**10**	60	**10**	80	**10**	100	**10**	**10**

FOCUSING ON THE WORST CASE

What if we focus our analysis on the **worst case**? What would be the number of primitive operations for this algorithm?

```
Algorithm Max(A, n)
    Input: An array A of n integers
        max ← A[0]
        for i ←1 to n-1 do
            if max < A[i] then
                max ← A[i]
```

- Our analysis can **focus on constructing a single input instance** that requires the largest number of primitive operations
- For this algorithm, the worst case is to have an input like this one, which would **require n assignments**:

10	20	30	40	50	60	70	80	90	100	110	120

- What about the primitive operations in the line (**for** $i \leftarrow 1$ **to** $n-1$ **do**)? this performs about n comparisons and n assignments. Thus, the total number of operations is:

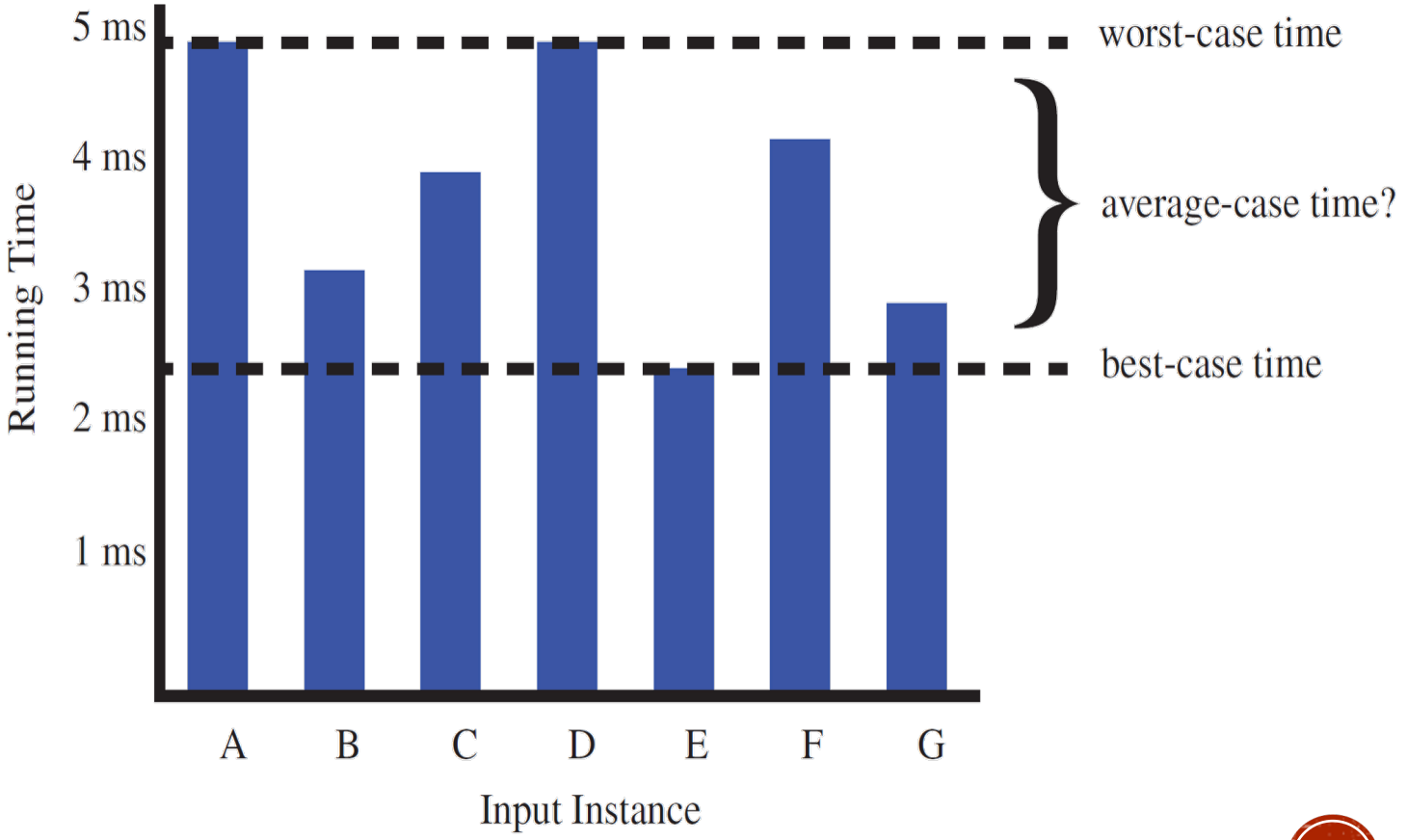
($2n$ operations in the for loop) + (n if-statements) + (n assignments of max) = $4n$

FOCUSING ON THE WORST CASE

- Worst-case analysis requires identifying the worst-case input, which is often simple

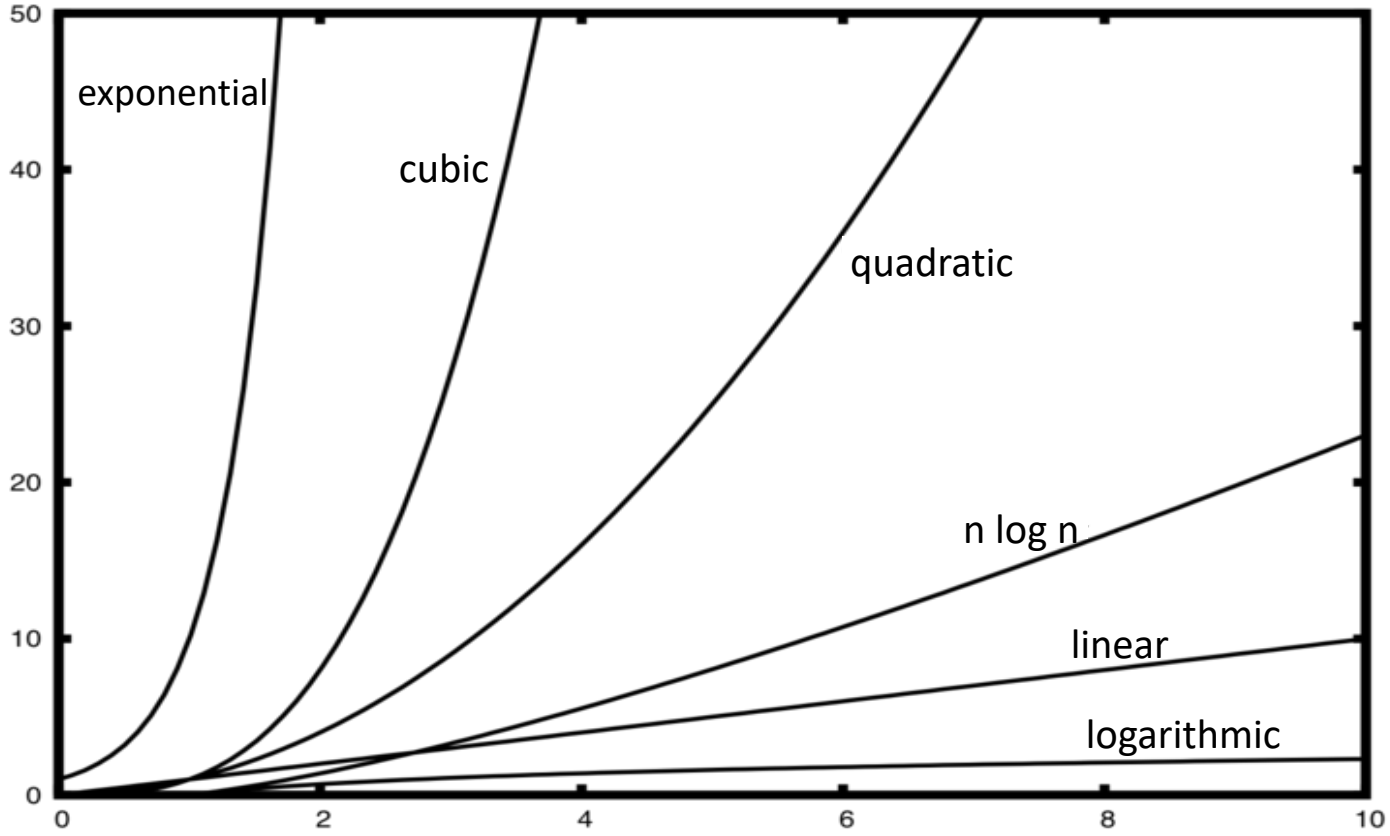
- Thus, we characterize runtime in terms of the **worst case**!
- Moreover, when designing an algorithm, we tend to focus on improving its performance given the worst possible input

- <u>The idea is</u>: If an algorithm performs well on the worst input, it performs well on all inputs!



WHAT MATTERS IS THE GROWTH RATE

- In computer science, we focus on **classifying algorithms into broad categories**, e.g., when the input size increases, how does the runtime grow?
  - linearly?
  - Logarithmically?
  - exponentially?



- We've seen an algorithm that requires $4n$ in the worst case

  but if we want to look at the **big picture**, it doesn't matter much if it is $4n$ or $20n$ or even $1000n$; **the category here is linear!**

ASYMPTOTIC NOTATION AND ANALYSIS

THE "BIG-OH" NOTATION

- We say that

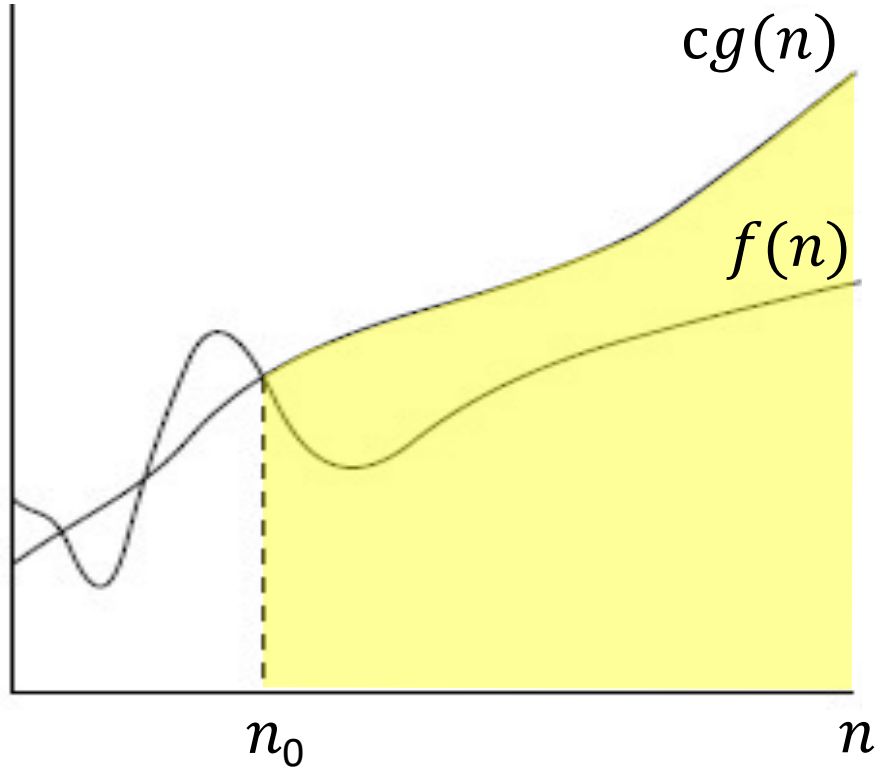
  $f(n)$ is $O(g(n))$

  which is pronounced as follows:

  "$f(n)$ is ***big-Oh*** of $g(n)$"

  The growth can be described as follows:

  $f(n) \leq cg(n)$, for $n \geq n_0$



THE "BIG-OH" NOTATION

Example: Prove that 8n−2 is O(n).

Justification:

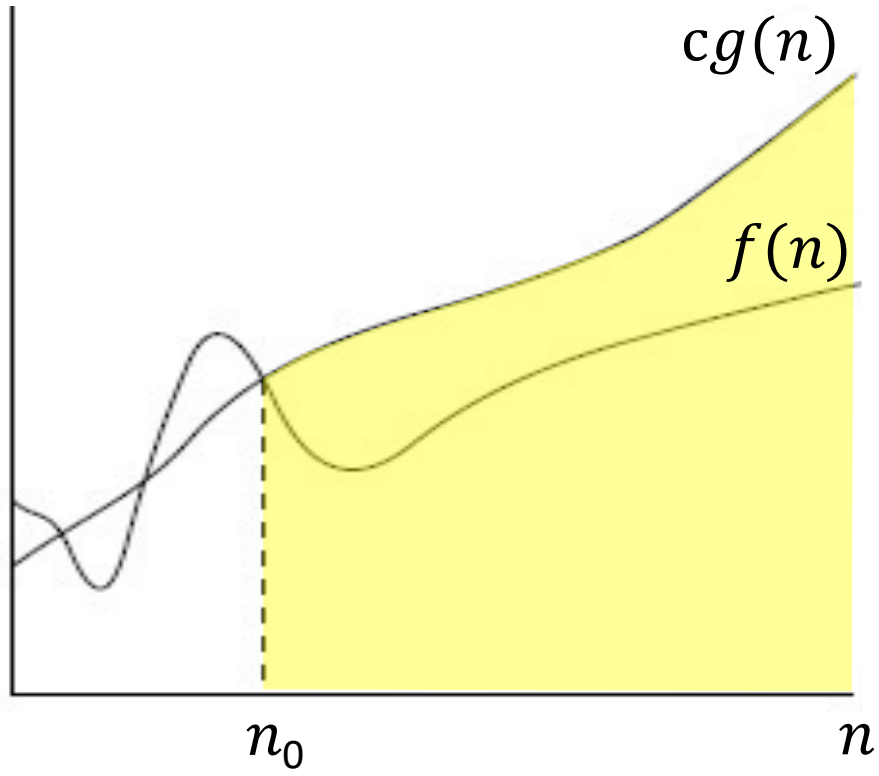
- f(n) = 8n-2, g(n) = n, and we need to find c > 0 and $n_0 \geq 1$ such that:

  $f(n) \leq cg(n)$, for $n \geq n_0$

  → 8n−2 ≤ cn, for $n \geq n_0$

- A possible choice is c = 8 and n_0 = 1, because:

  8n − 2 ≤ 8n, for n ≥ 1



THE "BIG-OH" NOTATION

Rule: If $f(n)$ is a polynomial of degree d, *i.e.*,

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d$$

and $a_d > 0$, then: $f(n)$ is $O(n^d)$