# Largest Sum Subinterval & Asymptotic Analysis

# Today's Plan

- Look at a fun problem (Largest Subinterval Sum)
- Iteratively develop more efficient solutions
  - Prove some things to help us get there
- Take a step back and state precisely what we mean by efficiency
- Practice some asymptotic analysis

# Largest Subinterval Sum

**INPUT:** An array $A$ of $n$ integers (1-indexed)

**OUTPUT:** The largest sum of any subinterval. The empty interval (which we will represent as $NULL$ has sum $0$).

Example 1: Consider the array $(10,20,-50,40)$

Subinterval [1, 1] = 10

Subinterval [1,4] = 10+20-50+40 = 20

Subinterval [2, 3] = 20-50 = -30

The largest sum subinterval is $40$, corresponding to $[4,4]$

# Largest Subinterval Sum

**INPUT:** An array $A$ of $n$ integers (1-indexed)

**OUTPUT:** The largest sum of any subinterval. The empty interval (which we will represent as $NULL$ has sum $0$).

Example 2: Consider the array $(-2,3,-2,4,-1,8,-20)$

The largest sum subinterval is $12$, corresponding to $[2,6]$

# Largest Subinterval Sum

**INPUT:** An array $A$ of $n$ integers (1-indexed)

**OUTPUT:** The largest sum of any subinterval. The empty interval (which we will represent as $NULL$ has sum $0$).

Question: Is this problem interesting when the array's integers are all positive?

No! Then the answer is always the entire interval…

# Developing an Algorithm

# Algorithm with $O(n^3)$ Steps

- Let's start with an algorithm that corresponds directly to the problem definition:
  - We are looking for the latest sum of any sub-interval
    - How many total sub-intervals are there?
      - $\binom{n}{2}$ which is $\dfrac{n(n+1)}{2} = O(n^2)$
    - How long does it take to sum a sub-interval?
      - $O(n)$ (in the worst case, must sum entire array)

This brute-force algorithm takes $O(n^3)$ steps

# LargestSum(A):

$largest \leftarrow 0$

**for** $i \leftarrow 1 \ldots n$

  **for** $j \leftarrow i \ldots n$

    $sum \leftarrow 0$

    **for** $k \leftarrow i \ldots j$

      $sum \leftarrow sum + A[k]$

    $largest \leftarrow \max(sum, largest)$

**return** $largest$

Try walking through LargestSum(A) on a small example, like $A = (10,20,-50,40)$

# Algorithm with $O(n^2)$ Steps

- The last algorithm repeated a lot of work. How?
  - If $A$ had 7 integers, interval $[2,7]$ computed $[2,2]$, $[2,3]$, $[2,4]$, and so on…
    - Can we avoid this repeated work?

**Idea:** Compute and reuse a *Partial Sum* table

$$PS(j) = \sum_{i=1}^{j} A(i)$$

# Algorithm with $O(n^2)$ Steps

**Claim:** We can use $PS$ to compute the sum of any interval $(i, j)$ in $O(1)$ time. How?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $A$ | | -2 | 3 | -2 | 4 | -1 | 8 | -20 |
| $PS$ | 0 | -2 | 1 | -1 | 3 | 2 | 10 | -10 |

$PS[i]$ contains sum of all integers "up until $A[i]$", with a $0$ for the empty array.

$$PS(j) = \sum_{i=1}^{j} A(i)$$

# Algorithm with $O(n^2)$ Steps

Example: How to compute $A(3,6)$?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $A$ | | -2 | 3 | -2 | 4 | -1 | 8 | -20 |
| $PS$ | 0 | -2 | 1 | -1 | 3 | 2 | 10 | -10 |

$PS[2]$ is everything before $A[3]$

$PS[6]$ is everything up to $A[6]$

$PS(j) = \sum^{j} A(i)$

Subtract $PS[j] - PS[i-1]$

# LargestSum(A):

$PS \leftarrow partial\_sums(A)$ // we can construct this in O(n) time

$largest \leftarrow 0$

**for** $i \leftarrow 1 \ldots n$

$O(n^2)$ iterations

  **for** $j \leftarrow i \ldots n$

    $largest \leftarrow \max(largest, PS[j] - PS[i-1])$

**return** $largest$

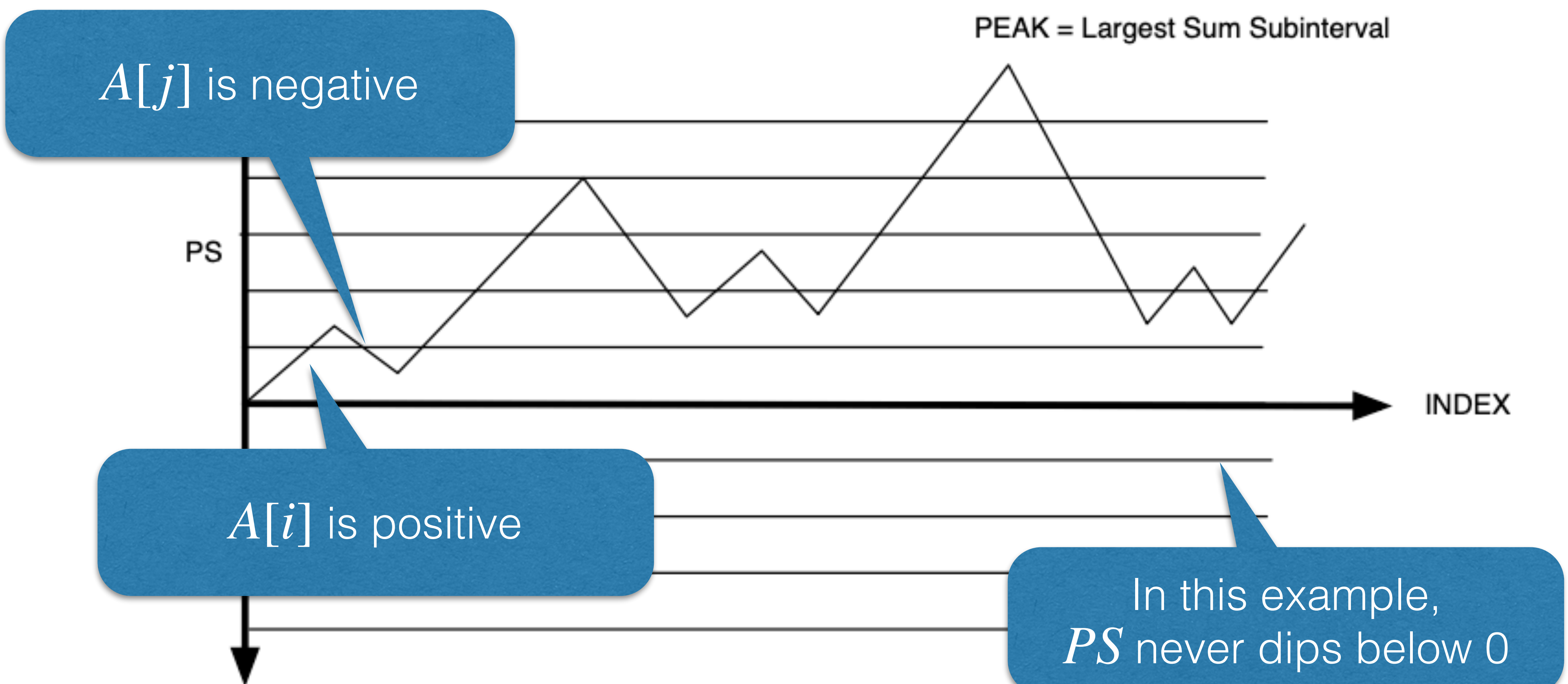
Each iteration performs $O(1)$ work

# Can We Do Even Better?

# Algorithm with $O(n)$ Steps

Let $PS(j) = \sum_{i=1}^{j} A[i]$ give the partial sum of the first $j$ integer values of $A$.

Let's visualize an example $PS(j)$



# Algorithm with $O(n)$ Steps

Observation 1: If $PS(j) \geq 0$ for all $1 \leq j \leq n$ then the largest sum subinterval is the interval $[1,k]$ where $k$ maximizes $PS(k)$.

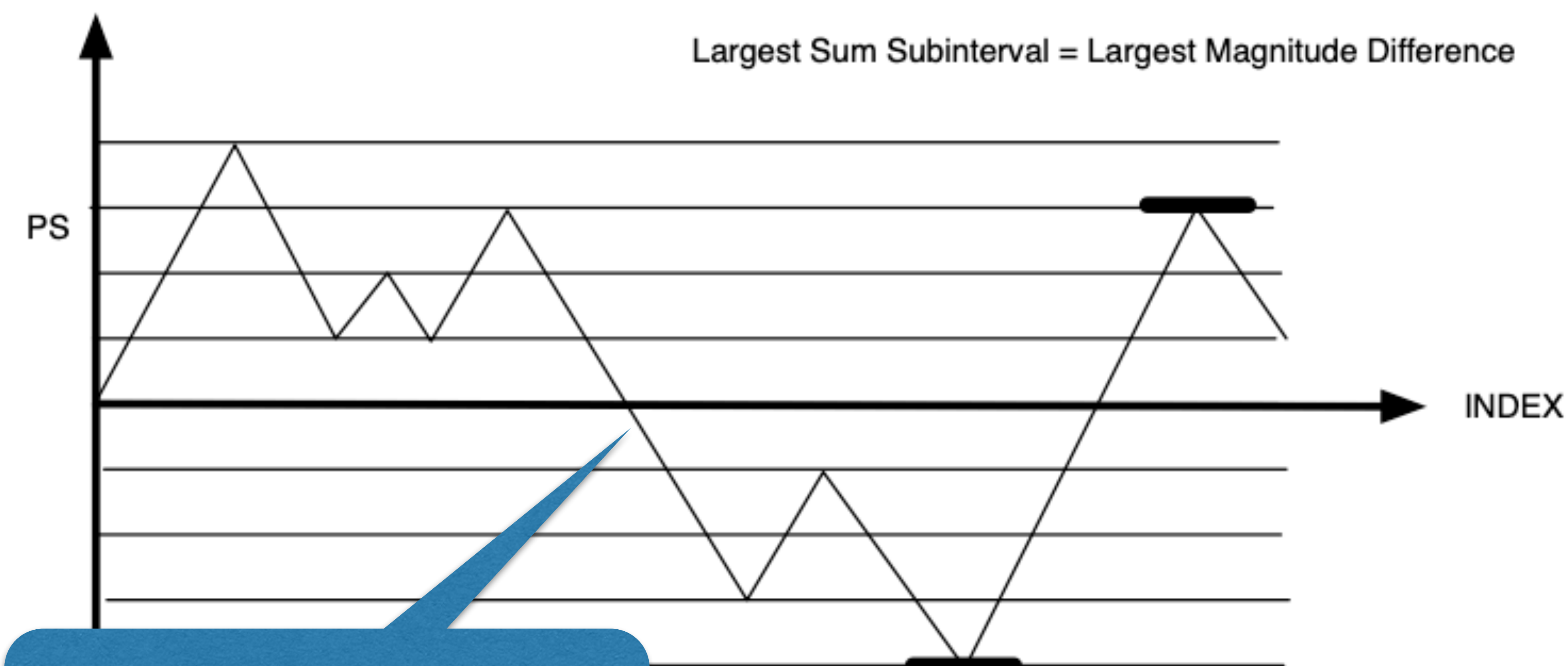
**Proof.** The proof is by contradiction.

Suppose $[1,k]$ did not give the largest sum. Then there is some other interval $[u, v]$ that has a larger sum. But shifting $u$ to $1$ cannot decrease the sum (since we would then be subtracting out 0), and shifting $v$ to $k$ cannot decrease the sum (since $k$ maximizes $PS(k)$). Thus $[u, v]$ cannot be an interval with a larger sum.

# Algorithm with $O(n)$ Steps

Let $PS(j) = \sum_{i=1}^{j} A[i])$ give the partial sum of the first $j$ integer values of $A$.

Let's visualize a second example $PS(j)$:




$PS$ dips below 0

# Algorithm with $O(n)$ Steps

Observation 2: When $PS(j)$ falls below $0$ for the first time, then the largest sum subinterval never includes $j$—it falls on one side or the other. That is, when $PS(j)$ falls below $0$ for the first time, the problem essentially "resets" with $PS(j)$ being "the new $0$".

**Proof.** The proof is by contradiction.

Suppose the largest sum subinterval $[u, v]$ contains the first point $j$ where the partial sum drops below $0$. Notice that $[u, j]$ corresponds to a negative sum. The interval $[j + 1, v]$ must be larger than $[u, v]$ since we are subtracting out a negative sum. This is a contradiction.

# LargestSum(A):

$sum, largest \leftarrow 0$

**for** $i \leftarrow 1 \ldots n$

$\quad sum \leftarrow \max(sum + A[i], 0)$

$\quad largest \leftarrow \max(sum, largest)$

**return** $largest$

This $O(n)$ algorithm follows from our previous two observations.

- We only need to worry about sums corresponding to intervals where $i$ is a new "0-point" for the partial sum and $j$ maximizes the partial sum
- Going back to our visualization, we are calculating the largest difference between some valley and a subsequent peak

# Analysis and Asymptotics

- *Why should we examine problems analytically?*
  - Analysis is independent of the algorithm's implementation, the language the program is written in, and the hardware on which the program is run
  - Theoretical efficiency almost always implies a path towards practical efficiency
    - When there is a mismatch between a theoretical model's predictions and the observed performance, there is an interesting systems problem to be solved!

My research group relies on this!

# Analysis and Asymptotics

- *Why use worst-case analysis?*
  - Worst-case is a *real* guarantee.
  - Worst-case captures efficiency reasonably well in practice. Exceptions are rare (e.g., Quicksort) and interesting.
  - Average case is hard to quantify—we often don’t know the true distribution of inputs, so what are we analyzing the average of?

# Analysis and Asymptotics

- *What does efficient actually mean?*
  - We will say an algorithm is efficient if it runs in time that is polynomial in the size of the input
  - Practical efficiency probably maxes out somewhere between $O(n \log n)$ and $O(n^3)$ , depending on the context
  - ***Not*** brute force!

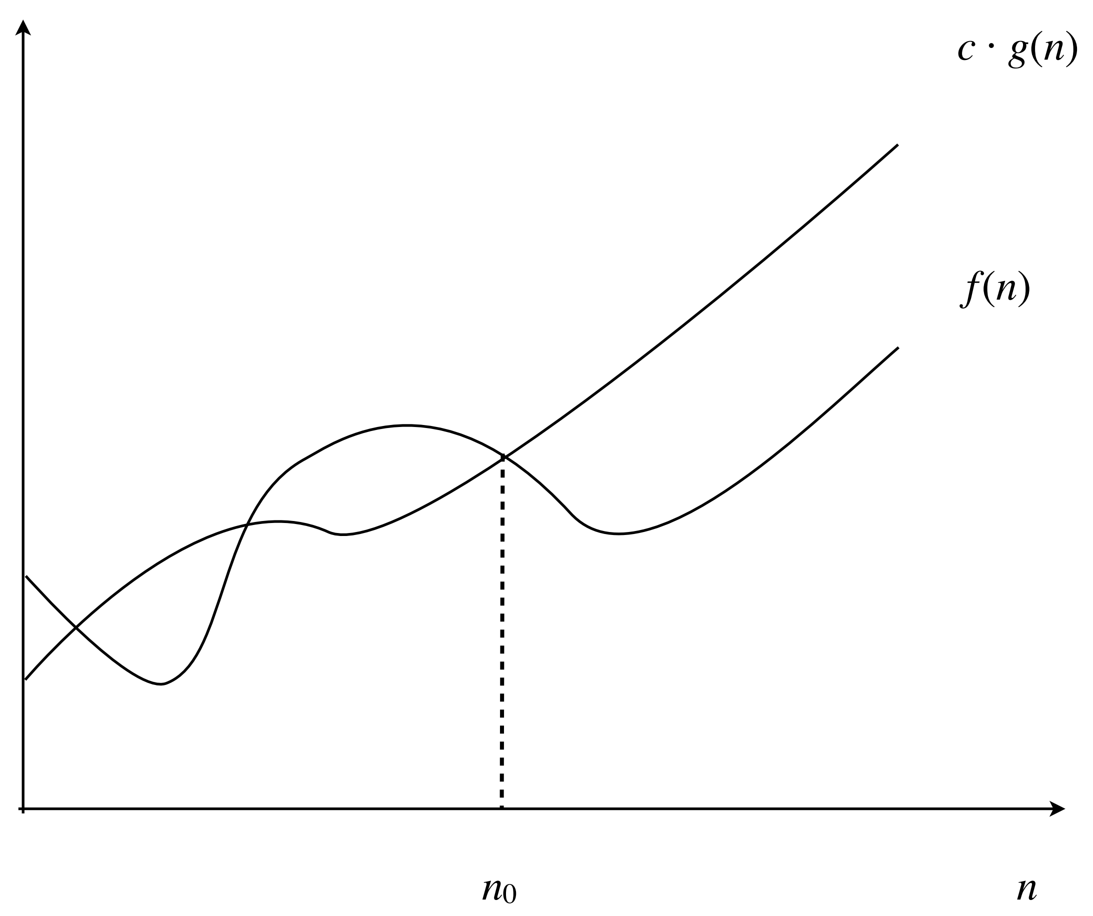
# Analysis and Asymptotics

- *Why use asymptotic analysis?*
  - Precise bounds are difficult to calculate
  - Precise runtime is dependent on external factors, often including things we don't consider or can't control (hardware, OS environment, compiler, …)
  - We often want to compare algorithms, and equivalency up to constant factors is often the right level of detail to have those conversations
    - Once we pick an efficient algorithm, we can optimize the "practical considerations"

# Asymptotic Analysis

# Big-O

**Definition** (Asymptotic upper bounds): $f(n)$ is $O(g(n))$ if and only if there exists constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq c \cdot g(n)$



# Big-O

**Definition** (Asymptotic upper bounds): $f(n)$ is $O(g(n))$ if and only if there exists constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq c \cdot g(n)$

Example:

$$
\begin{aligned}
f(n) &= 3n^2 + 17n + 8 \\
&\leq 3n^2 + 17n^2 + 8n^2 \quad \text{For } n \geq 1 \\
&= 28n^2
\end{aligned}
$$

Choosing $c = 28$ and $n_0 = 1$ means $f(n)$ is $O(n^2)$

# Class Quiz

Let $f(n) = 3n^2 + 17n\log_2 n + 1000$. Which of the following are true?

A. $f(n)$ is $O(n^2)$.

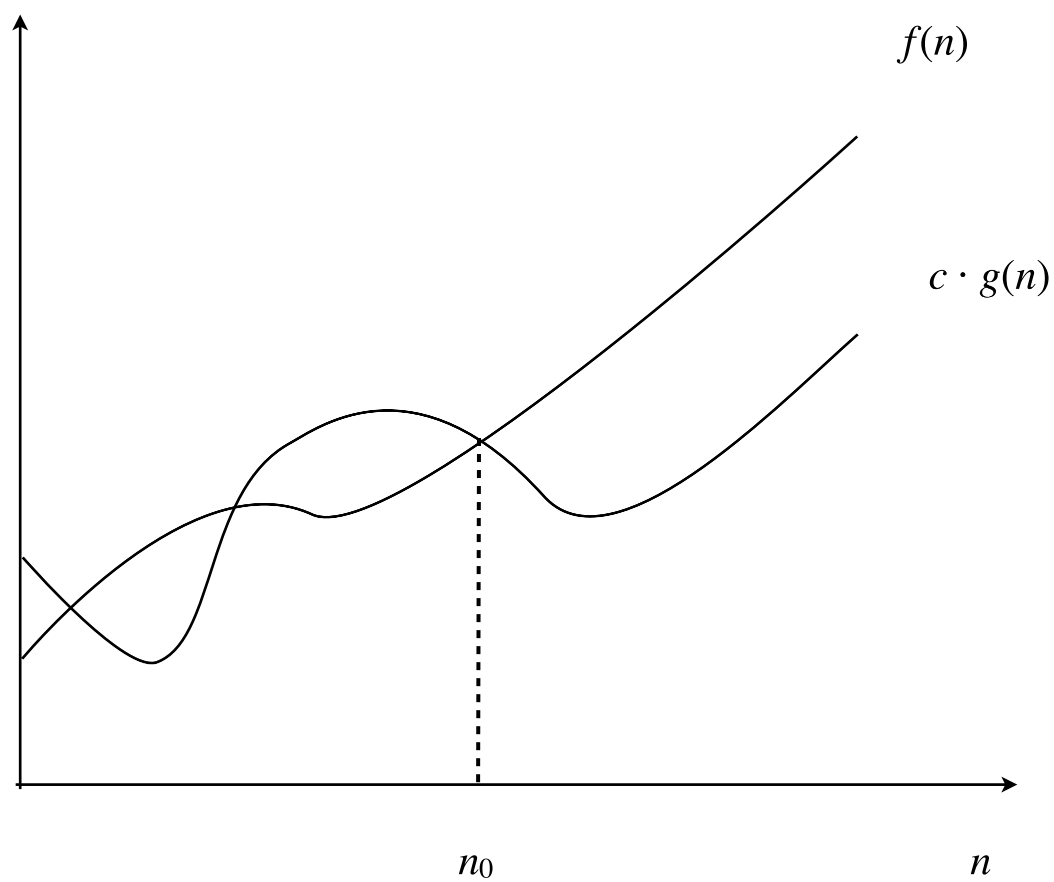
B. $f(n)$ is $O(n^3)$.

C. Both A and B.

D. Neither A nor B.

# Big-Omega

**Definition** (Asymptotic lower bounds): $f(n)$ is $\Omega(g(n))$ if and only if there exists constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \geq c \cdot g(n)$



# Big-Omega

**Definition** (Asymptotic lower bounds): $f(n)$ is $\Omega(g(n))$ if and only if there exists constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \geq c \cdot g(n)$

Example:

$$\begin{aligned} f(n) &= 3n^2 + 17n + 8 \\ &\geq 3n^2 \qquad \text{For } n \geq 0 \end{aligned}$$

Choosing $c = 1$ and $n_0 = 0$ means $f(n)$ is $\Omega(n^2)$

# Big-Theta

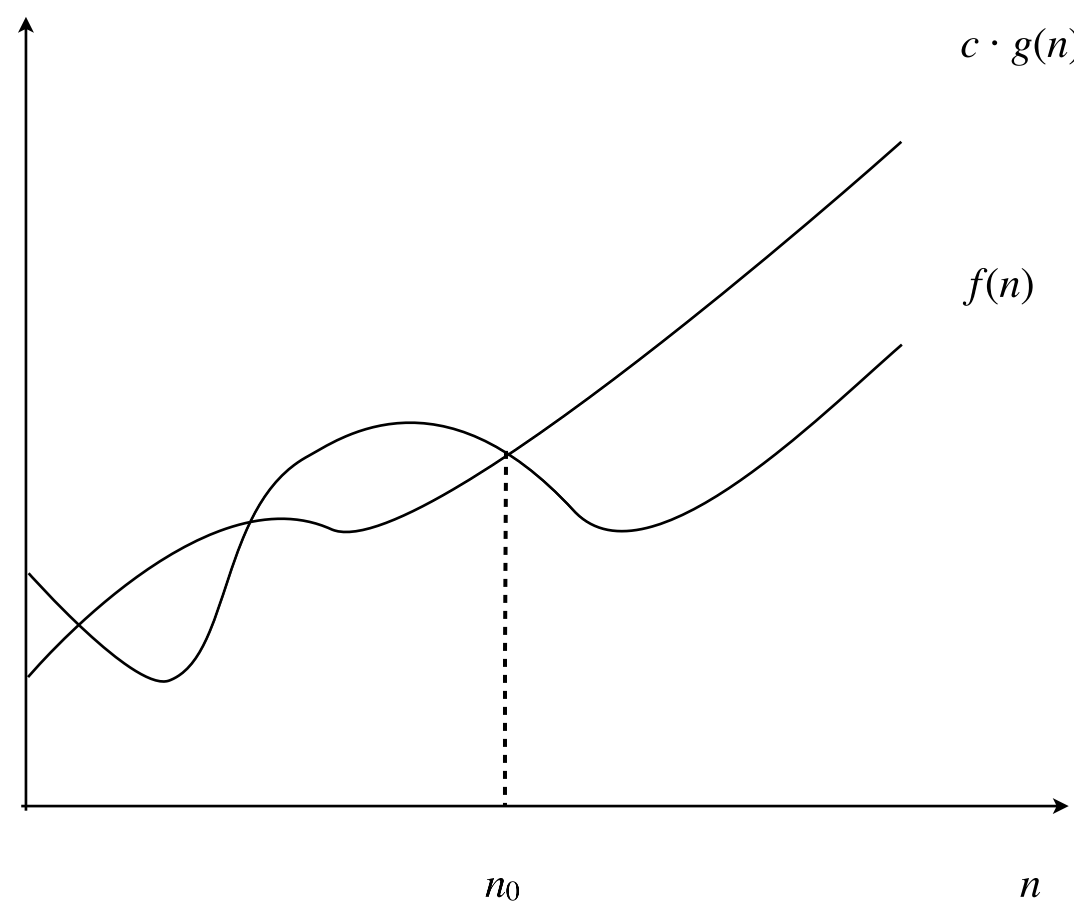
**Definition** (Asymptotic tight bounds): $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $\Omega(g(n))$

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

Ideally, we'd strive for a "tight" bounds whenever we can!

# Big Oh- Notational Abuses

- $O(g(n))$ is actually a set of functions, but the CS community writes $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$
- For example
  - $f_1(n) = O(n \log n) = O(n^2)$
  - $f_2(n) = O(3n^2 + n) = O(n^2)$
  - But $f_1(n) \neq f_2(n)$
- Okay to abuse notation in this way



# Growth of Functions

**Table 2.1** The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

| | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $1.5^n$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| $n = 10$ | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | 4 sec |
| $n = 30$ | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | 18 min | $10^{25}$ years |
| $n = 50$ | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | 11 min | 36 years | very long |
| $n = 100$ | $<$ 1 sec | $<$ 1 sec | $<$ 1 sec | 1 sec | 12,892 years | $10^{17}$ years | very long |
| $n = 1{,}000$ | $<$ 1 sec | $<$ 1 sec | 1 sec | 18 min | very long | very long | very long |
| $n = 10{,}000$ | $<$ 1 sec | $<$ 1 sec | 2 min | 12 days | very long | very long | very long |
| $n = 100{,}000$ | $<$ 1 sec | 2 sec | 3 hours | 32 years | very long | very long | very long |
| $n = 1{,}000{,}000$ | 1 sec | 20 sec | 12 days | 31,710 years | very long | very long | very long |

# Playing with Logs: Properties

- In this class, $\log n$ means $\log_2 n$, $\ln n = \log_e n$
- Constant base doesn't matter: $\log_b(n) = \dfrac{\log n}{\log b} = O(\log n)$
- Properties of logs:
  - $\log(n^m) = m \log n$
  - $\log(ab) = \log a + \log b$
  - $\log(a/b) = \log a - \log b$

$$a^{\log_a n} = n$$

**We will use this a lot!**

**Exponents**

$$n^a \cdot n^b = n^{a+b}$$

$$(n^a)^b = n^{ab}$$

# Comparing Running Times

- When comparing two functions, helpful to simplify first
- Is $n^{1/\log n} = O(1)$?
- Is $\log \sqrt{4^n} = O(n^2)$ ?
- Is $n = O(2^{\log_4 n})$?

# Comparing Running Times

- When comparing two functions, helpful to simplify first
- Is $n^{1/\log n} = O(1)$?
    - Simplify $n^{1/\log n} = (2^{\log n})^{1/\log n} = 2$ : **True**
- Is $\log \sqrt{4^n} = O(n^2)$
    - Simplify $\log \sqrt{2^{2n}} = \log 2^n = n \log 2 = O(n)$ : **True**
- Is $n = O(2^{\log_4 n})$?
    - Simplify $2^{\log_4 n} = 2^{\frac{\log_2 n}{log_2 4}} = 2^{(\log_2 n)/2} = 2^{\log_2 \sqrt{n}} = \sqrt{n}$ : **False**

# Tools for Comparing Asymptotics

- We ca use limits to show asymptotic bounds
  - If $\lim_{n \to \infty} \frac{f(x)}{g(x)} = 0$, then $f(x) = O(g(x))$
  - If $\lim_{n \to \infty} \frac{f(x)}{g(x)} = c$ for some constant $0 < c < \infty$, then $f(x) = \Theta(g(x))$

# Tools for Comparing Asymptotics

- Logs grow slowly than any polynomial:
  - $\log_a n = O(n^b)$ for every $a > 1,\ b > 0$
- Exponentials grow faster than any polynomial:
  - $n^d = O(r^n)$ for every $d > 1,\ r > 0$
- Taking logs
  - As $\log x$ is a strictly increasing function for $x > 0$, $\log(f(n)) < \log(g(n))$ implies $f(n) < g(n)$
  - E.g. Compare $3^{\log n}$ vs $2^n$
    - Taking log of both, $\log n \log 3$ vs $n$
  - Beware: when comparing logs, constants matter!