# Analysis of Algorithms & Big-O

CS16: Introduction to Algorithms & Data Structures
Summer 2021

# How fast is this algorithm?

```
function find_least_important_seam(vals):
    dirs = 2D array with same dimensions as vals
    costs = 2D array with same dimensions as vals
    costs[height-1] = vals[height-1] // initialize bottom row of costs

    for row from height-2 to 0:
        for col from 0 to width-1:
            costs[row][col] = vals[row][col] +
                              min(costs[row+1][col-1],
                                  costs[row+1][col],
                                  costs[row+1][col+1])
            dirs[row][col] = -1, 0, or 1 // depending on min

    // Find least important start pixel
    min_col = argmin(costs[0]) // Returns index of min in top row

    // Create vertical seam of size ‘height’ by tracing from top
    seam = []
    seam[0] = min_col
    for row from 0 to height-2:
        seam[row+1] = seam[row] + dirs[row][seam[row]]

    return seam
```

# How fast is this algorithm?

```
function sum_array(array)
   // Input: an array of integers
   // Output: the sum of the integers
   if array.length = 0
      return error
   sum = 0
   for i in [0, array.length-1]:
     sum = sum + array[i]
    return sum
```

# Let’s measure it

- Implement it (in Python)
- Run it
- Time it

# Let's measure it

- Implement it (in Python)
- Run it
- Time it
- Repeat for different input sizes

# What might affect these measurements?

# Let’s try something else

- Have to look at the algorithm
- How long will it take?
- Depends on how long each operation takes
    - +
    - *
    - array[i]
- Let’s assume each operation takes the same amount of time

# How fast is this algorithm?

```
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0              1op
       return error                  1op
    sum = 0                          1op
    for i in [0, array.length-1]:    1oop
      sum = sum + array[i]           3ops per loop
     return sum                      1op
```

- Do we count "**`return error`**"?
  - depends on whether input array is empty
  - if **`array`** is empty then **`sum_array`** takes **`2 ops`**
  - if **`array`** is not empty then **`sum_array`** takes **`??? ops`**

# Run time depends on input

- Which inputs should we choose?
  - Best-case?
  - Worst-case?
  - Average-case?
- In general, worst-case
  - CS is an engineering discipline
  - If I'm building a bridge, don't care about best-case weight tolerance!

# How fast is this algorithm?

```
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0          <---- 1op
       return error              <---- 1op
    sum = 0                      <---- 1op
    for i in [0, array.length-1]:  <---- 1oop
      sum = sum + array[i]       <---- 3ops per loop
     return sum                  <---- 1op
```

- How long in non-empty case?
  - Depends on loop length, which depends on array length
  - Call the running time on an array of length n **T(n)**
  - What's T(n)?

# How fast is this algorithm?

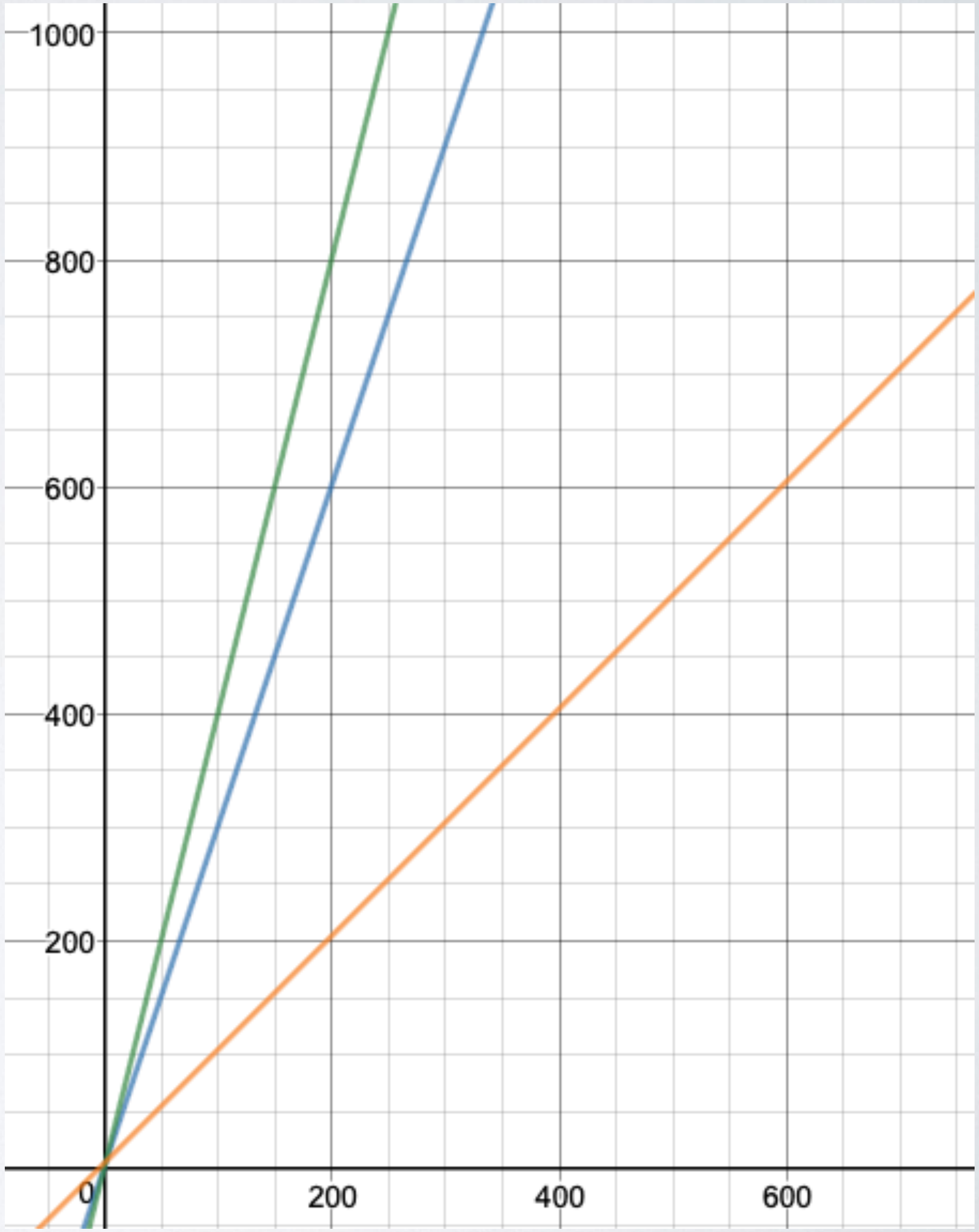
```
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0                          1op
       return error                              1op
    sum = 0                                      1op
    for i in [0, array.length-1]:                loop
      sum = sum + array[i]                       3ops per loop
     return sum                                  1op
```

- $T(n) = 3n + 3$ ops
- Do we believe this number? What assumptions did we make?
- What if array accesses take twice as long as addition?

# Could be any of these…

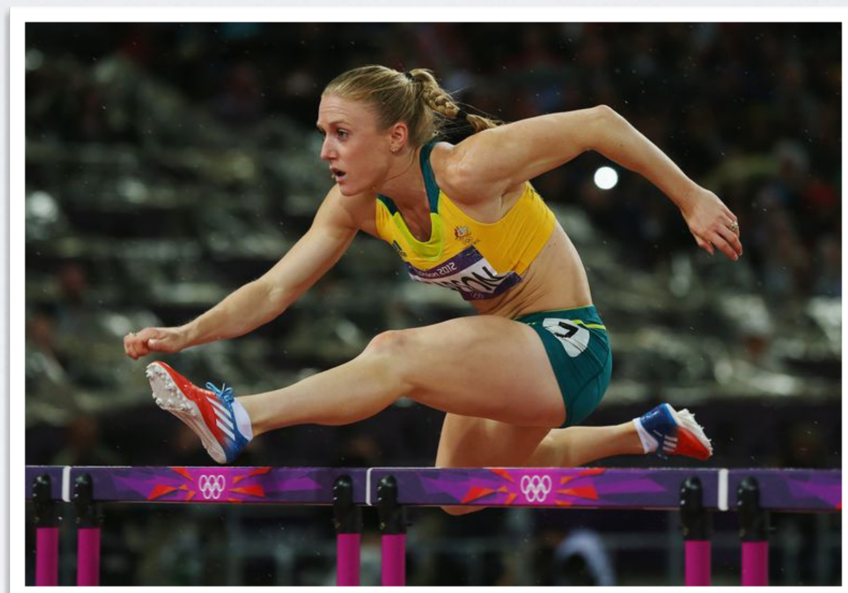
- 3n+3
- 4n+3
- n+5
- …
- What can we say for sure?

Linear



# Running Times

**Constant**
independent of input size



**Linear**
depends on input size


**Quadratic**
depends on square of input size

# **Constant** Running Time

```
function first(array):
    // Input: an array
    // Output: the first element
    return array[0]
```

**2ops**

- How many operations are executed?
  - `T(n)=2` ops
  - What if array has `100` elements?
  - What if array has `100,000` elements?
- **key observation:**
  - *running time does not depend on array size!*

# What's the running time?

```
function possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    //           between any two elements in the list
    products = []
    for i in [0, array.length):
        for j in [0, array.length):
            products.append(array[i] * array[j])
    return products
```

1op

loop

loop per loop

4ops per loop

per loop

1op

Quadratic

# What's the running time?

```
function argmax(array)
    // Input: an array
    // Output: the index of the maximum value
    index = 0
    for i in [1, array.length):
      if array[i] > array[index]:
        index = i
     return index
```

1op

loop

3ops per loop

1op per loop (sometimes)

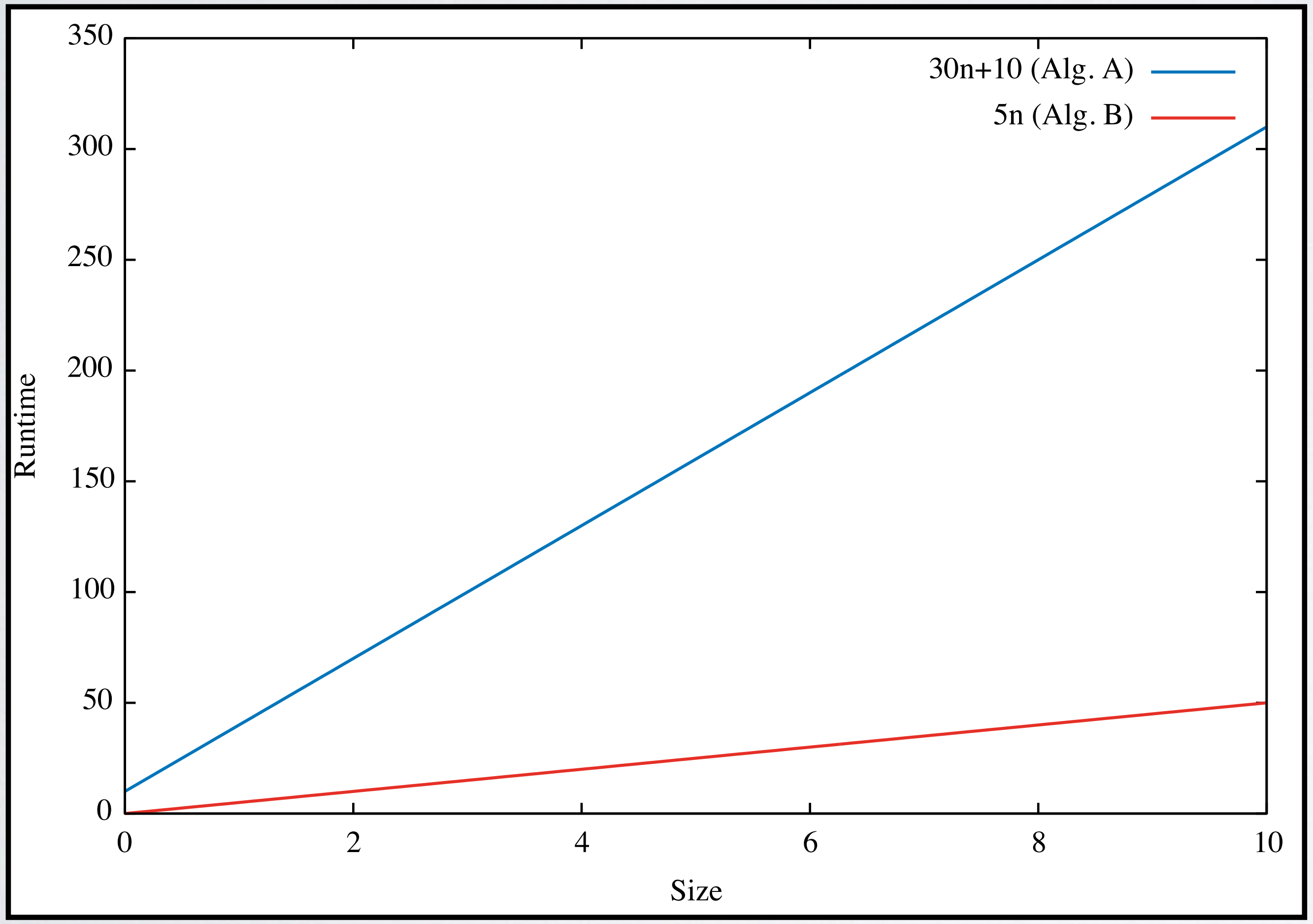
1op

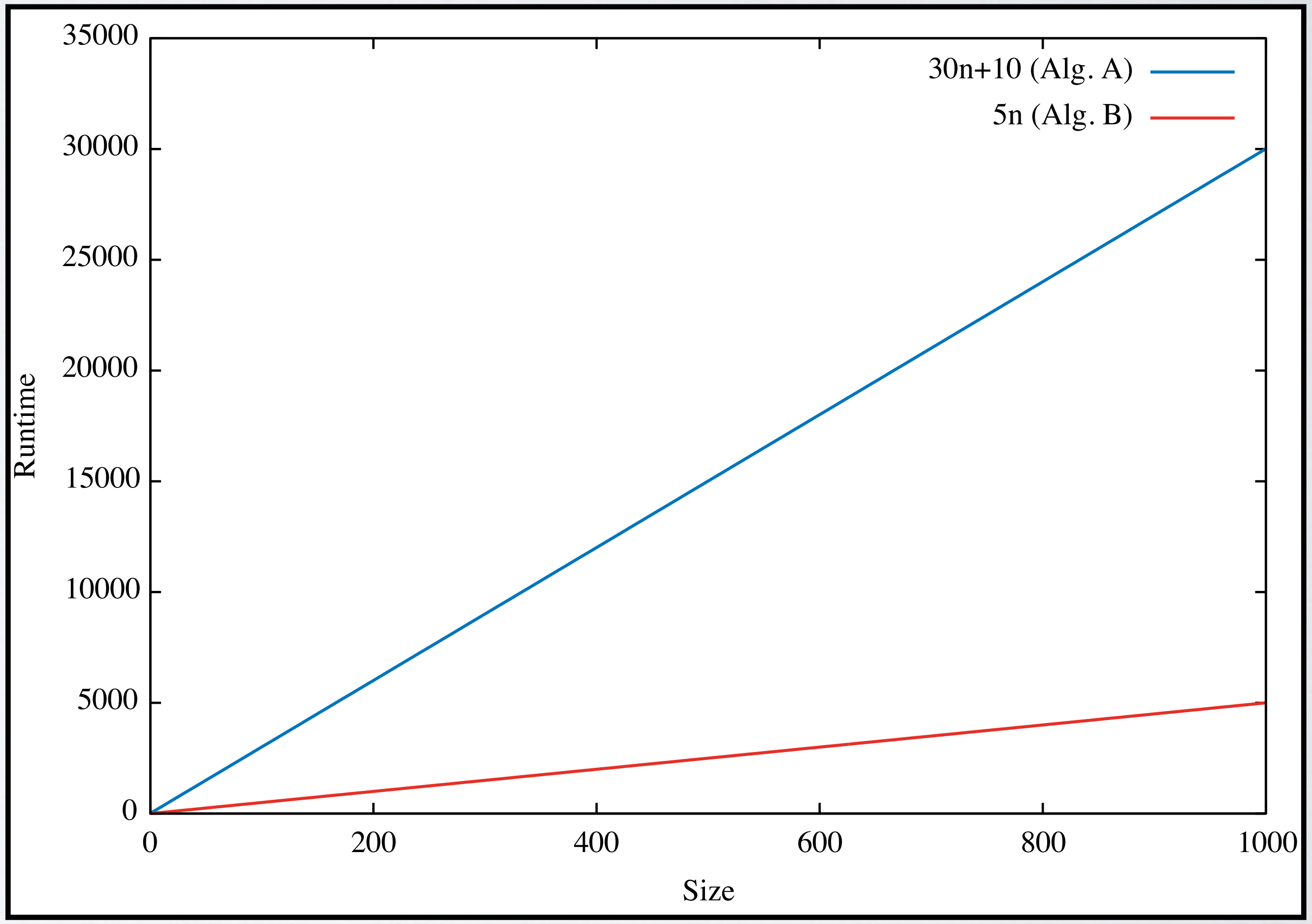
Linear

Q: how do we compare running times?

# Which Algorithm is Better?

- Algorithm **A** takes $T_A(n)=30n+10$ ops
- Algorithm **B** takes $T_B(n)=5n$ ops

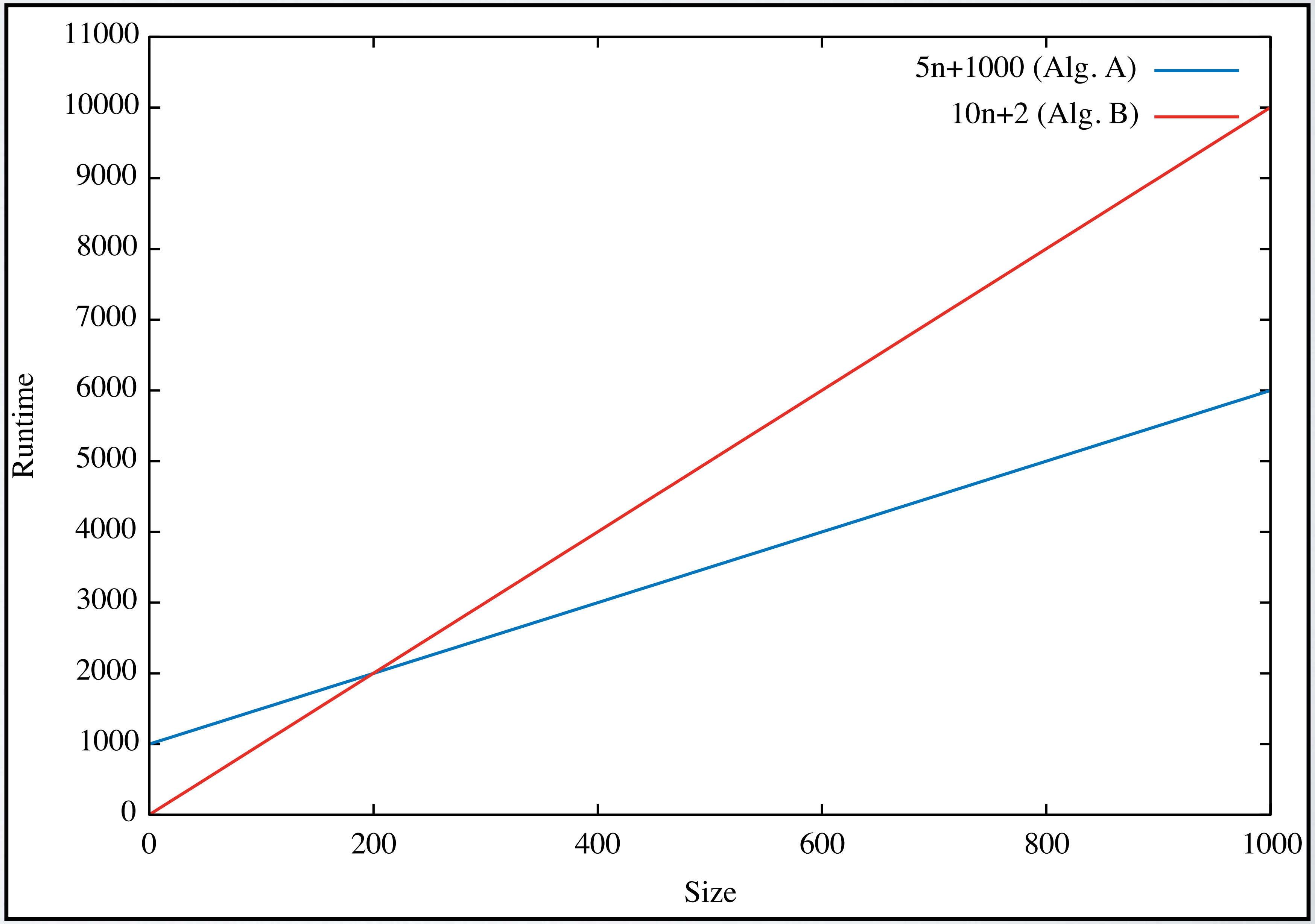


# Which Algorithm is Better?

- Alg **A** takes $T_A(n)=5n+1000$ ops
- Alg **B** takes $T_B(n)=10n+2$ ops
- It depends on $n$

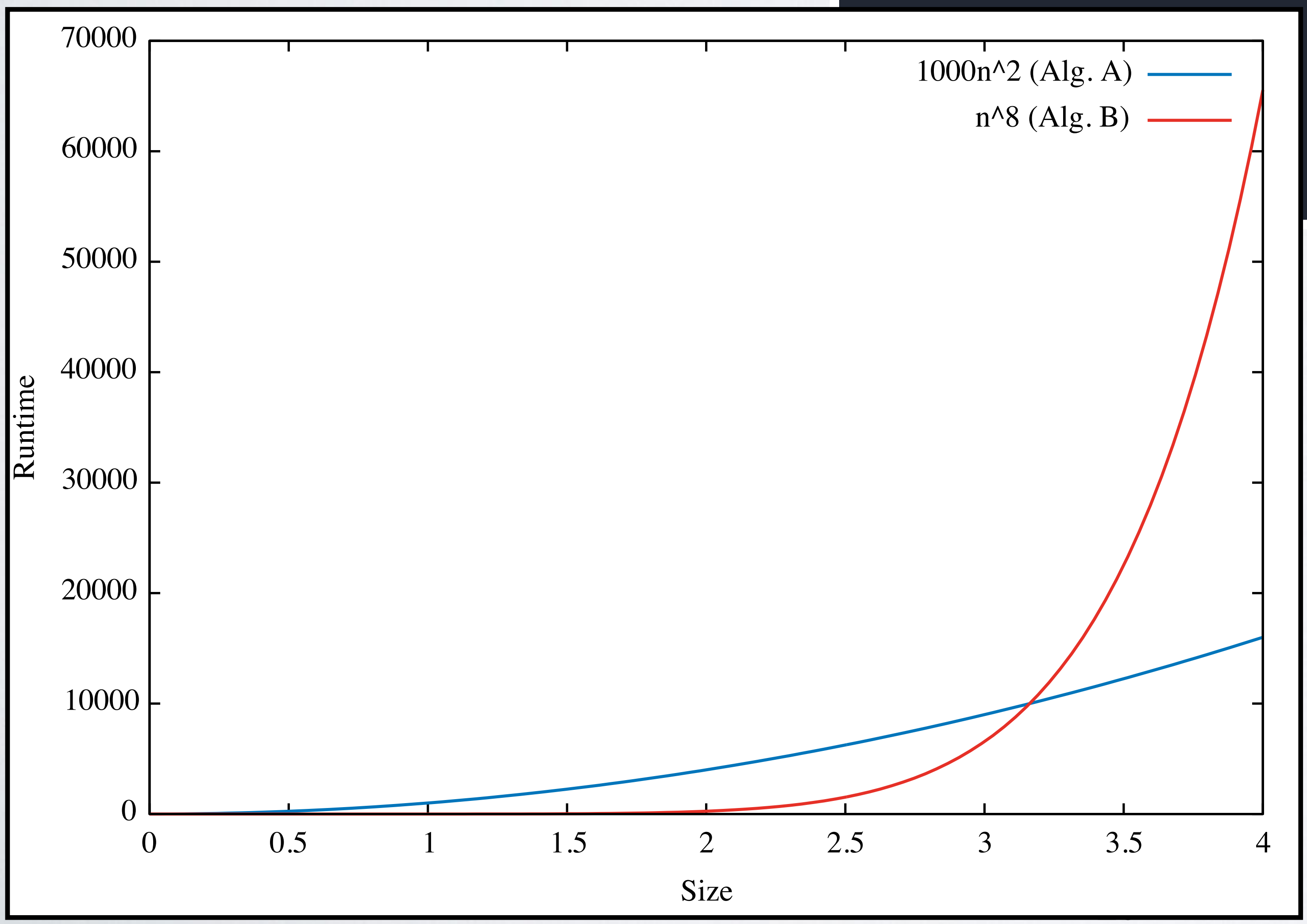
```
rtime(A) < rtime(B) ⟺ 5n+1000 < 10n+2
                    ⟺ 5n > 998
                    ⟺ n > 199.6
```

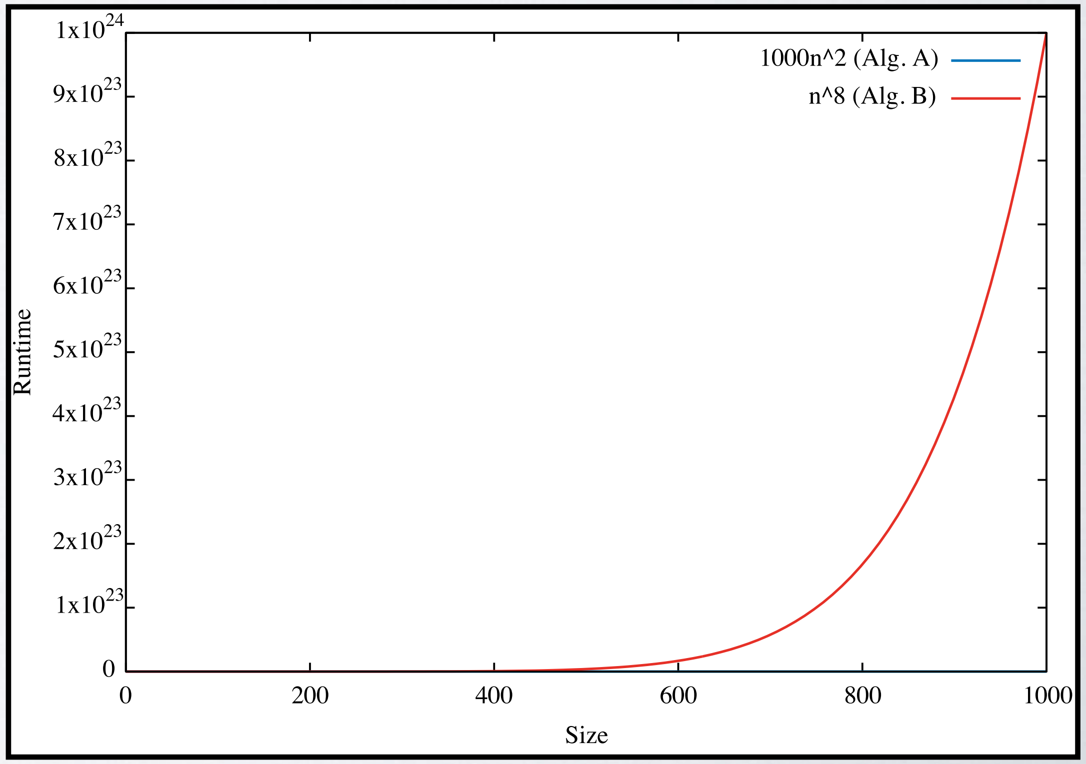


# Which Algorithm is Better?

- Alg **A** takes $T_A(n)=1000n^2$ ops
- Alg **B** takes $T_B(n)=n^8$ ops
- It depends on **n**

$$
\begin{aligned}
\texttt{rtime(A)} < \texttt{rtime(B)} &\iff 1000n^2 < n^8 \\
&\iff 1000n^2 - n^8 < 0 \\
&\iff n^2(1000 - n^6) < 0 \\
&\iff 1000 - n^6 < 0 \\
&\iff n > 1000^{1/6} \\
&\iff n > 3.16\ldots
\end{aligned}
$$



# What is Running Time?

Asymptotic worst-case running time

=

*Number of elementary operations*
*on worst-case input*
*as a function of input size n*
***when n tends to infinity***

In CS "running time" usually means asymptotic worst-case running time…but not always!
we will learn about other kinds of running times

# Comparing Running Times

***Comparing*** *asymptotic* running times

=

*$T_A(n)$ is better than $T_B(n)$ if*

*for large enough $n$*

*$T_A(n)$ grows slower than $T_B(n)$*

Q: can we formalize all this mathematically?

# Big-O

**Definition (Big-O)**: $T_A(n)$ is $O(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \leq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$'s order of growth is at most $T_B(n)$'s order of growth
- Examples
  - $2n+10$ is $O(n)$

# Big-O

- How do we find "the Big-O of something"?
  - Usually you "eyeball" it
  - Then you try to prove it
    - (most of the time in CS16 it will be simple enough that you don't need to prove it)

# Big-O Examples

> **Definition (Big-O)**: $T_A(n)$ is $O(T_B(n))$ if there exists positive constants $\mathbf{c}$ and $\mathbf{n_0}$ such that:
>
> $$T_A(n) \leq c \cdot T_B(n)$$
>
> for all $\mathbf{n} \geq \mathbf{n_0}$

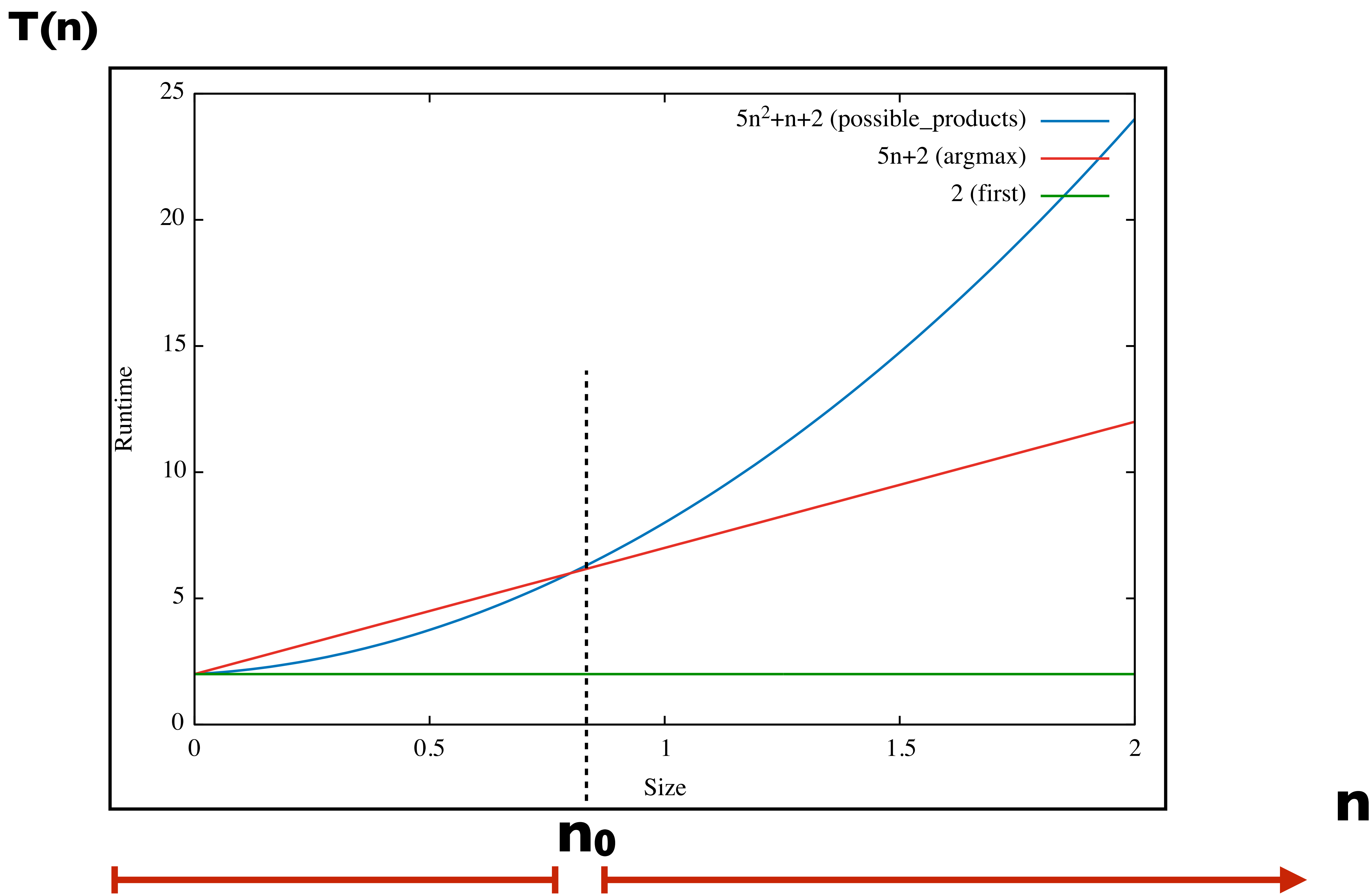
- Guess that $2n+10$ is $O(n)$. Can we prove it?
  - If we set $\mathbf{c=3}$, can we find an $\mathbf{n_0}$ such that for all $\mathbf{n} \geq \mathbf{n_0}$, $2n+10 \leq 3 \cdot \mathbf{n}$?

**c**

$$2n+10 \leq 3n \iff -n + 10 \leq 0$$
$$\iff n \geq 10$$

$\mathbf{n_0}$

# What is $\mathtt{n_0}$ ?




**We don't care what happens here** **We only care what happens here**

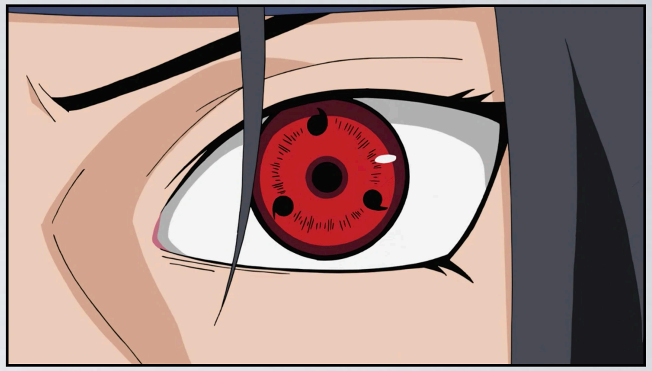
# More Big-O Examples

- $n^2$ is not $O(n)$. Why?
  - What do we have to show to prove that $n^2$ is $O(n)$?
  - We have to find a positive constant $c$,
  - and a positive constant $n_0$ such that
  - for all $n > n_0$, $n^2 \le c \cdot n$
  - This is not possible!

$$n^2 \le c \cdot n \iff n \le c$$

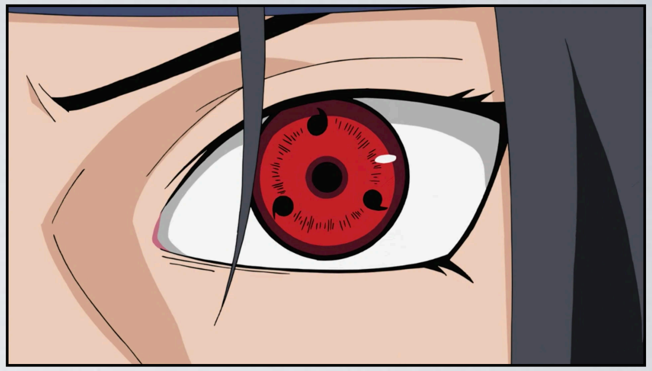
Not possible when $n$ grows & $c$ is constant

# Eyeballing Big-O



- If `T(n)` is a polynomial of degree `d`, $T(n) = an^d + bn^{d-1} + \ldots + wn + z$
  - then `T(n)` is $O(n^d)$
- In other words you can ignore
  - lower-order terms
  - constant factors
- Examples
  - $1000n^2+400n+739$ is $O(n^2)$
  - $n^{80}+43n^{72}+5n+1$ is $O(n^{80})$
- *For the Big-O, use the smallest upper bound*
  - `2n` is $O(n^{50})$ but that's not really a useful bound
  - instead it is better to say that `2n` is $O(n)$

# Eyeballing Big-O



- $n^{10}+2020$ is $O(n^{10})$ but also $O(n^{11})$,…,$O(n^{50})$,…
  - but better to say it is $O(n^{10})$
- There are at most `300` people in this room
  - there are also at most `1000`,…,`1M`, …
  - but telling me there are at most `300` is more "useful"

# More Eyeballing Big-O

- Find Big-O of 3 algorithms
  - runtime of first is $T(n)=2$
  - runtime of argmax is $T(n)=4n+2$
  - runtime of possible_products is $T(n)=4n^2+n+3$
- Replace constants with "**c**" (they are irrelevant as **n** grows)
  - first: $T(n)=c$
  - argmax: $T(n)=c_0n+c_1$
  - possible_products: $T(n)=c_0n^2+n+c_1$

# More Eyeballing Big-O

- Discard lower-order terms & constants
  - first: `T(n)=c` is `O(1)`
  - argmax: $T(n)=c_0n+c_1$ is `O(n)`
  - possible_products: $T(n)=c_0n^2+n+c_1$ is $O(n^2)$
- The convention for `T(n)=c` is to write `O(1)`

# Running Times

| $n$ | $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| 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 | $1.84 \times 10^{19}$ |
| 128 | 7 | 128 | 896 | 16,384 | 2,097,152 | $3.40 \times 10^{38}$ |
| 256 | 8 | 256 | 2,048 | 65,536 | 16,777,216 | $1.15 \times 10^{77}$ |
| 512 | 9 | 512 | 4,608 | 262,144 | 134,217,728 | $1.34 \times 10^{154}$ |

# Big-O

**Definition (Big-O)**: $T_A(n)$ is $O(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \leq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$'s growth rate is upper bounded by $T_B(n)$'s growth rate
- But what if we need to express a lower bound?
  - we use Big-$\Omega$ notation!

# Big-Omega

**Definition (Big-Ω)**: $T_A(n)$ is $\Omega(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \geq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$'s growth rate is lower bounded by $T_B(n)$'s growth rate
- What about an upper **and** a lower bound?
  - We use Big-**Θ** notation

# Big-Theta

**Definition (Big-Θ)**: $T_A(n)$ is $\Theta(T_B(n))$ if it is $O(T_B(n))$ and $\Omega(T_B(n))$.

- $T_A(n)$'s growth rate is the same as $T_B(n)$'s

# More Examples

| $T(n)$ | Big-O | Another Big-O | Big-Ω | Big-ϴ |
|---|---|---|---|---|
| $an + b$ | $O(n)$ | $O(n^{100})$ | $\Omega(n)$ | $\Theta(n)$ |
| $an^2 + bn + c$ | $O(n^3)$ | $O(n^2)$ | $\Omega(n)$ | $\Theta(n^2)$ |
| $a$ | $O(n)$ | $O(1)$ | $\Omega(1)$ | $\Theta(1)$ |
| $3^n + an^{40}$ | $O(3^n)$ | $O(50^n)$ | $\Omega(n)$ | $\Theta(3^n)$ |
| $an + b \log n$ | $O(n^2)$ | $O(n))$ | $\Omega(\log n)$ | $\Theta(n)$ |

# Key takeaways

- We can analyze algorithm running time independent of implementation
- Important thing is behavior as input grows
- We'll be doing a fair amount of running time analysis using Big-O notation and proofs
  - Big-O might seem complex at first
  - It's just formalizing the intuition from earlier—constant factors don't matter

# How fast is this algorithm?

```
function find_least_important_seam(vals):
    dirs = 2D array with same dimensions as vals
    costs = 2D array with same dimensions as vals
    costs[height-1] = vals[height-1] // initialize bottom row of costs

    for row from height-2 to 0:
        for col from 0 to width-1:
            costs[row][col] = vals[row][col] +
                              min(costs[row+1][col-1],
                                  costs[row+1][col],
                                  costs[row+1][col+1])
            dirs[row][col] = -1, 0, or 1 // depending on min

    // Find least important start pixel
    min_col = argmin(costs[0]) // Returns index of min in top row

    // Create vertical seam of size 'height' by tracing from top
    seam = []
    seam[0] = min_col
    for row from 0 to height-2:
        seam[row+1] = seam[row] + dirs[row][seam[row]]

    return seam
```

# Additional Readings

- To read more on asymptotic runtime and Big-O
  - Dasgupta et al. section 0.3 (pp. 15-17)
  - Roughgarden Part 1, Chap 2

# References

- Slide #12
  - Usain Bolt (constant): 8-time Olympic gold medalist and greatest sprinter of all time
  - Sally Pearson (linear): 2012 Olympic world champion in 100m hurdles, 2011 and 2017 World Champion
  - Wilson Kipsang (quadratic): former marathon world-record holder, Olympic marathon bronze medalist
  - Eliud Kipchoge (quadratic): 2016 Olympic marathon gold medalist, greatest marathoner of the modern era