1. (8 points) Suppose you have algorithms with the following runtimes. (Assume these are exact running times as a function of the input size n, not a asymptotic running times.) How much slower do these algorithms get when you double the input size?

(a) n

$$\frac{\text{time for double}}{\text{time}} = \frac{(2n)}{n} = 2$$

(b) $n \log n$

$$\frac{(2n \log(2n))}{n \log n} = \frac{2n \log 2 + 2n \log n}{n \log n}$$

$$= \frac{2\cancel{n}\overset{1}{\log 2}}{\cancel{n}\log n} + \cancel{\frac{2n \log n}{n \log n}} = \frac{2}{\log n} + 2$$

$\log_2 2 = 1$

for large n, 2

2. (7 points) Recall the definition of big O:

$f(n)$ is $O(g(n))$ if there exist positive constants n_0, c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Using this definition, prove that $\underbrace{2n+5}_{f(n)}$ is $O(\underbrace{n^2}_{g(n)})$.

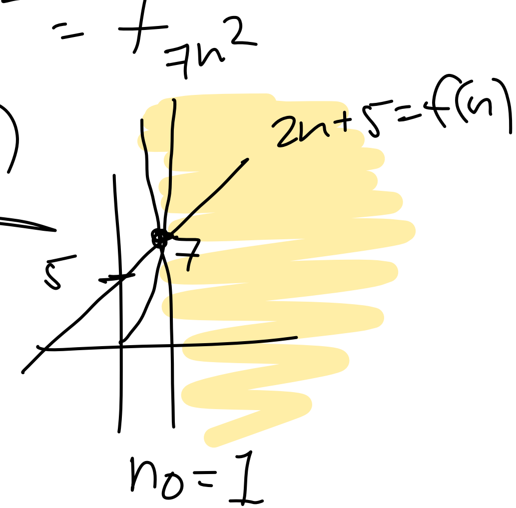
Give a c, n_0 $\quad$ $c = 7,\ n_0 = 1$

Notice that

$2(1) + 5 = 7$ and $7 \cdot (1)^2 = 7$

so we have $f(n_0) \leq c \cdot g(n_0)$

$\forall n \geq n_0$



Consider $c = 7$, $n_0 = 1$.

Notice that $2n+5 \leq 7 \cdot n^2$

for all $n \geq 1$.

Summary

When we talk about the runtime of an algorithm, we always mean:

(1) there is a $f(n)$ expressing the # of primitive operations an alg. takes on an input of size n (ex $5n\log n + 2n + 25$)

(2) give $g(n)$ such that $f(n)$ is $O(g(n))$.

ex (not exactly what we want: $5n\log n + 2n + 25$ is $O(2^n)$)

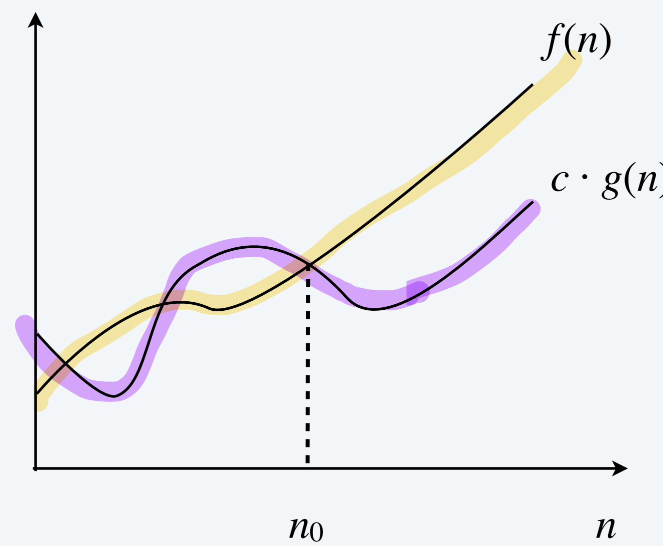
ex $5n\log n + 2n + 25$ is $O(n\log n)$

For $i = 1, 2, \ldots, n$
    For $j = i+1, i+3, \ldots, n$
        Add up array entries $A[i]$ through $A[j]$
        Store the result in $B[i, j]$
    Endfor
Endfor

Big Omega notation

Ω $\geq$

Lower bounds. $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is both $\Omega(n^2)$ and $\Omega(n)$.
- $f(n)$ is not $\Omega(n^3)$.

how would I prove $32n^2+17n+1$ is $\Omega(n^2)$?

give c, n_0 s.t. $32n^2+17n+1 \geq cn^2$ for all $n \geq n_0$.

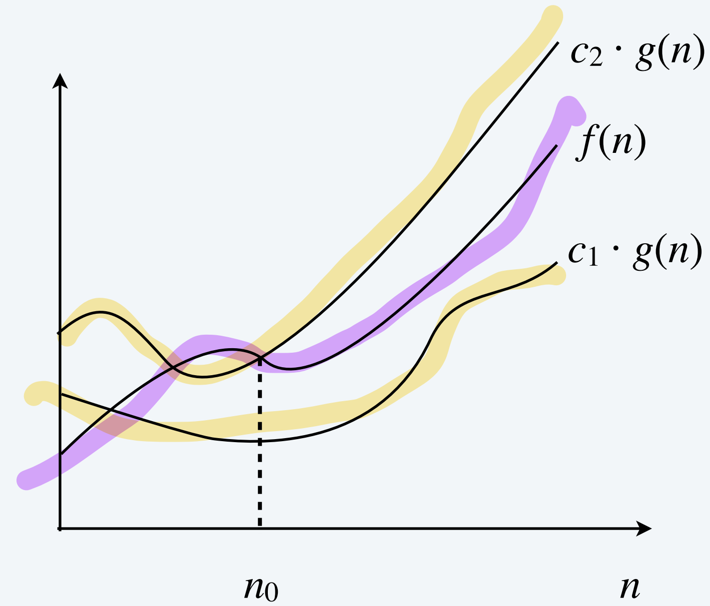
$32n^2+17n+1$ is not $\Omega(n^3)$

to prove:

argue that there are no c, n_0 s.t. $32n^2+17n+1 \geq cn^3$ for all $n \geq n_0$.

Big Theta notation

Tight bounds. $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq \underline{c_1 \cdot g(n)} \leq f(n) \leq \underline{c_2 \cdot g(n)}$ for all $n \geq n_0$.



Ex. $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is $\Theta(n^2)$.
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.

$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ <u>and</u> $f(n)$ is $\Omega(g(n))$

Is $f(n) = 32n^2 + 17n + 5$...

1. $\Theta(n)$
2. $\Theta(n^2)$
3. $\Theta(n^3)$
4. All three
5. $\Theta(n^2)$ and $\Theta(n^3)$

$f(n)$ is $\Theta(g(n))$ if both $\Omega(g(n))$ and $O(g(n))$

$g(n)$	$O(g(n))$? $\leq$	$\Omega(g(n))$? $\geq$
n	no	yes
n^2	yes	yes
n^3	yes	no

Multiplication by a constant

back at :20

Suppose I have runtime $f(n)$ and I know it is $O(g(n))$.

ex $f(n) = 3n^2$

$g(n) = n^3$

Is $b \cdot f(n) = O(g(n))$ for all constants b?

① yes

② no

for all b, $b \cdot 3n^2$ is $O(n^3)$?

no? to prove, give a counter example

$f(n) = 3n^2$

$g(n) = n^3$

$b = 2$

$b3n^2$ not $O(n^3)$.

$6n^2$ not $O(n^3)$. F

yes? there is some c, n_0 s.t. $6n^2 \leq cn^3$ for all $n \geq n_0$

✓

is 2^{n+1} $O(2^n)$?

notice that $2^{n+1} = 2^n 2$ so 2^{n+1} is $O(2^n)$

Is 2^{2n} $O(2^n)$?

hint: $2^{2n} = 2^{n+n} = 2^n 2^n$

$\bigcirc \cdot f(n)$

non-constant

Asymptotically different functions

c — constant

$\log n$ — logarithmic

n — linear

$n \log n$ — linearithmic

n^2 — quadratic

n^3 — cubic

...

n^a — polynomial

2^n

3^n

...

b^n

$n!$

exponential
inefficient

2n is not O(n!)

where does $\sqrt{n}$ fall?

$\sqrt{n} = n^{1/2}$

$n^{\log n}$

Summary

Big O is ______ for functions

Big Omega is ________ for functions

Big Theta is _________ for functions

We use Big O/Big Omega/Big Theta in order to:

Is there a worst-case input here? (Or best case?)

```
Algorithm 1
Input: integer array A of length n
for i = 1, 2, …, n :
    total = 0
    for j = 1, 2, …, n:
        total = total + A[i]
    B[i] = total
```

[1, 2, 3, 4, 5]

[5, 4, 3, 2, 8]

Worst-case: worst case for given n.

insertion sort:

for size n input $[\qquad]$ $\xleftrightarrow{\quad n \quad}$

if sorted: $\Theta(n)$

if reverse-sorted: $\Theta(n^2)$

worst-case runtime:

$f(n)$ such that alg takes $f(n)$ steps on worst-case input of size n.