$\mathcal{O}$

The stakes are very high when two algorithms solve the same problem but scale so differently with the size of the problem (we'll call that n). We want to express this scaling in a way that:

- is simple
- ignores the differences between different hardware, other processes on computer
- ignores special behaviour for small n

Disclaimer

For those of you taking CSC165...

best bound on worst-case

- We (computer scientists) commonly refer to $\mathcal{O}$, but often mean Θ.
- What we're concerned about is the *tightest* upper bound.
- So, while technically a function that has worst case running time proportional to $n\ lg\ n$ is in $\mathcal{O}(n^2)$, we wouldn't say that.

$O(n\lg n)$

big-$\mathcal{O}$ definition

Suppose the number of "steps" (operations that don't depend on n, the input size) can be expressed as $t(n)$. We say that $t \in \mathcal{O}(g)$ if:

there are positive constants c and B so that for every natural number n no smaller than B,
$t(n) \leq cg(n)$

if $t \in \mathcal{O}(n)$, then it's also the case that $t \in \mathcal{O}(n^2)$, and all larger bounds

$$\mathcal{O}(1) \subseteq \mathcal{O}(\lg(n)) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(2^n) \subseteq \mathcal{O}(n^n) \ldots$$

We want to talk about the tightest bound, ie. the smallest upper bound

sequences

```
def silly(n):
    n = 17 * n**(1/2)
    n = n + 3
    print("n is: {}.".format(n))

    if n > 97:
        print('big!')
    else:
        print('not so big!')
        for i in range(n):
            print(i)
```

O(1)

≤ 97 → small

O(1)

constant

still O(1)

How does the running time of silly depend on n?

loops

How does the running time of this code fragment depend on **n**?

```
sum = 0
for i in range(n):
    sum += i
```

$O(n)$

time proportional to n

How does the running time of this code fragment depend on **n**?

```
sum = 0
for i in range(n//2):
    for j in range(n**2):
        sum += i * j
```

$n/2$

n^2

$$\frac{n}{2} \times n^2$$

$$= \frac{n^3}{\cancel{2}} \in O(n^3)$$

more loops

How does the running of this code fragment depend on **n**?

```
i, j, sum = 0, 0, 0
while i**2 < n:
    while j**2 < n:
        sum += i * j
        j += 1
    i += 1
```

$\rightarrow \sqrt{n}$

$\rightarrow \sqrt{n}$

$O(1)$

$\sqrt{n} \times \sqrt{n} \times O(1)$

$\rightarrow O(n)$

How does the running time of this code fragment depend on **n**?

```
i, sum = 0, 0
while i < n * n:
    sum += i
    i += 1
```

$\rightarrow$ # of times add 1 to i to get to n^2! $\rightarrow n^2$

$O(n^2)$

$n^2 - \frac{n}{2}$

conditions

How does the running time of this code fragment depend on **n**?

```
sum = 0
if n % 2 == 0:
    for i in range(n*n):
        sum += 1
else:
    for i in range(5, n+3):
        sum += i
```

n^2

worst case

$O(n^2)$

n

$O(n) \subseteq O(n^2)$

halving

How does the running time of **twoness** depend on **n**?

```
def twoness(n):
    count = 0
    while n > 1:
        n = n // 2
        count = count + 1
    return count
```

lg n $\Rightarrow$ # of times we can divide n by 2 //

working with lg

lg(n): this is the number of times you can divide n in half before reaching 1.

- refresher: $a^b = c$ means $\log_a c = b$.
- this runtime behaviour often occurs when we "divide and conquer" a problem (e.g. binary search)
- we usually assume $\lg n$ (log base 2), but the difference is only a constant:

$$2^{\log_2 n} = n = 10^{\log_{10} n} \implies \log_2 n = \log_2 10 \times \log_{10} n$$

- so we just say $\mathcal{O}(\lg n)$.

miscellaneous

n = len(L) ?

How does the running time of this code fragment depend on n?

```
for k in range(5000):
    if L[k] % 2 == 0:
        even += 1
    else:
        odd += 1
```

→ constant # of iterations

does not depend on input size.

Python list operations

How does the running time of this code fragment depend on `n` and `m`?

```
sum = 0
for i in range(n):
    for j in range(m):
        sum += (i + j)
```

Python list operations

n = # of items in list

- append O(1)
- insert O(n)
- pop O(1)
- pop(i)/remove O(n)
- indexing O(1)
- slicing for a slice of size k O(k)
- in, max, min O(n) → look at everything
- len O(1) → info is stored/maintained

shift everything over.

L[1:]

L[i:j]

k = j − i + 1

Python dict operations

most of the time
(average case)

- get item $O(1)$
- set item $O(1)$
- delete item $O(1)$
- in $O(1)$

Monday

what these are

How do we do it???