UNDERSTANDING ALGORITHM EFFICIENCY

Goals:

- Evaluate Different **Algorithms**
  - programs, or implementations of a solutions
- Be platform agnostic
- Evaluate performance on all inputs
- What resource? **Time**

MEASURING CLOCK TIME

```
import time

tstart = time.clock()
<code>
tend = time.clock()
print(tend - tstart)
```

- Not platform
- Doesn't allow us to understand the behavior on all inputs
- Program dependent measurement.

COUNTING STEPS

- Assume certain steps take the same, fixed constant time
  - mathematical operation on "small" numbers
  - comparisons
  - assignments
  - accessing objects from memory
- Then count the number of steps as a function of input

```
def mult (a, b):
    prod = 0          — 1
    while b > 0:      — 1
        prod += a     — 1
        b -= 1        — 1
    return prod       — 1
```

#steps = 1 + 3b + 1
= 3b + 2
O(b)

```
def mult (a, b)
1 — prod = 0
1 — for _ in range (b)
1 —     prod += a
1 — return prod.
```

#steps = 1 + 2b + 1
= 2b + 2
O(b)

```
def isMem?(L, e):
    i = len(L) - 1          - 1
    while i >= 0:           - 1
        if L[i] == e:       - 1
            return True     - 1
        i -= 1              - 1
    return False            - 1
```

```
def isMem?(L, e)
    return e in L
```

$O(n)$

If e is the last element of the list then the loop executes 1 - time] Best case

If e is not in the list, the loop will execute len(L) times.] Worst case

→ Measure running time on worst case.

$\#steps = 2 + 3\,len(L)$.

$= 2 + 3n$.

$O(n)$

TOWARDS A MEASURE OF COMPLEXITY

- Measure running time in an abstract model that counts # of steps
- Measure time as a function of input size
- Get as tight a bound as possible for the running time on the worst input of a given size
- Don't be precise: get "order of" instead of "exact" growth.

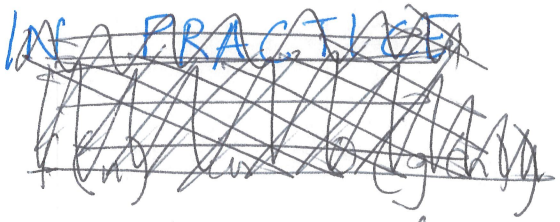
BIG O

Fix functions $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$

$f(n)$ is $O(g(n))$ if there are constants c and n_0 such that for every $n \geq n_0$

$$f(n) \leq c\, g(n).$$

$f(n)$ is $O(g(n))$ if $f(n)$ is at most a constant times $g(n)$ for all (large values of n) but some finitely many.



If $f(n)$ is $O(g(n))$ and $g(n) \leq h(n)$ then $f(n)$ is also $O(h(n))$

IN PRACTICE

$f(n)$ is $O(g(n))$ if $g(n)$ is obtained from $f(n)$ by

- ignoring additive constants
- ignoring multiplicative constants
- keeping only the terms that grows most rapidly in an expression involving a sum of terms

$\log_b a = c$ where $b^c = a$

$\frac{\log_c a}{\log_c b} = \log_b a$

$n^2 + 2n + 2 \longrightarrow O(n^2)$

$n^2 + 1000000\,n + \underline{3^{277}} \rightarrow O(n^2)$

$\log n + 10^{-20} n + 2 \rightarrow O(n)$

$n^{30} + 3^n \rightarrow O(3^n)$

$\lim_{n \to \infty} \frac{n^{30}}{3^n} \rightarrow 0$

COMPLEXITY OF COMMON PYTHON OPS

int, float

• Arithmetic ops	O(1)
• Assignment	O(1)
• Comparison (==, <=, etc.)	O(1)

List: n is len(L)

• Access (L[i])	O(1)
• Store (L[i] = ..)	O(1)
• len(L)	O(1)
• for i in L:	O(1)
• L.append(i)	O(1)
• L1 == L	O(n)
• L.remove(i)	O(n)
• in list	O(n)

```
def isContained(L1, L2)
    for e in L1:                    — O(1)
        if e not in L2:             — O(n)
            return False            — O(1)
    return True
```

$O(n) \cdot O(n) = O(n^2)$

```
def mult_rec(a, b):
    if b == 0:
        return 0
    return a + mult_rec(a, b-1)
```

$T(b)$ = # steps when b is the value of the second argument.

steps if $b = 0$, $= O(1)$

$b \neq 0$,

$T(0) = O(1)$

$$\begin{aligned} T(b) &= O(1) + T(b-1) \\ &= O(1) + O(1) + T(b-2) \\ &= O(1) + O(1) + O(1) + T(b-3) \\ &\vdots \\ &= \underbrace{O(1) + O(1) + \cdots O(1)}_{b} = O(b) \end{aligned}$$

```
def mult_fast (a,b):
    if b==0:
        return 0
    c = 2 * mult_fast (a, b//2)
    if b%2==1:
        c += a
    return c
```

$T(b)$ – running of mult_fast when 2nd arg is b.

$T(0) = O(1)$

$T(b) = O(1) + T(b/2)$

$= O(1) + O(1) + T(b/4)$

$\vdots$

$= \underbrace{O(1) + O(1) + O(1) \cdots O(1)}_{i} + T(b/2^i)$

For what value of i is $b/2^i = 1$

$b = 2^i \Rightarrow i = \log_2 b$

$T(b) \neq O(\log b)$

ANALYZING THE COMPLEXITY OF ALGORITHMS

- Analyze statements in the code
- Time for a sequence of statements is the sum of the times of each statement

$$O(f(n)) + O(g(n)) = O(f(n)+g(n))$$

- Nested loops
  - A loop running $O(f(n))$ times with body taking $O(g(n))$ takes in total $O(f(n) \cdot g(n))$.