CS2100

Asymptotics +
big-O

Today

- HW3 due via git
- HW1 grade files are pushed
- HW4 - up today, due in 1 week (more in a bit)
  read carefully!
  code is on webpage
  but also in course repo

- Midterm 1: Tuesday, Feb 20
  Review in class Monday Feb. 19

Next: Asymptotic Analysis

Motivation:

How should we compare 2 programs?

- speed
- space
- compatability

⋮

Speed:

- Exact speed can depend on many variables besides the algorithm.

Issues at play:

Alternative approach:

Count primitive operations, which are smallest operations.

In addition: generally only examine worst case running time.

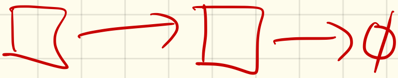
Why?

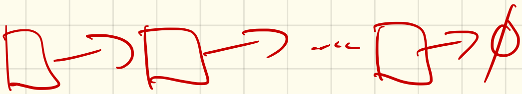
Now: How to actually compare?

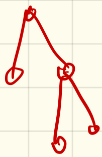
- Remember small difference may be due to processor, language, or any number of things that aren't dependent on the algorithm.

- Also: need a way to account for inputs changing

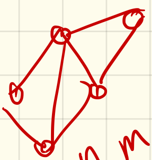
eg searching a list



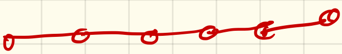




• in terms of input size - n



n, m

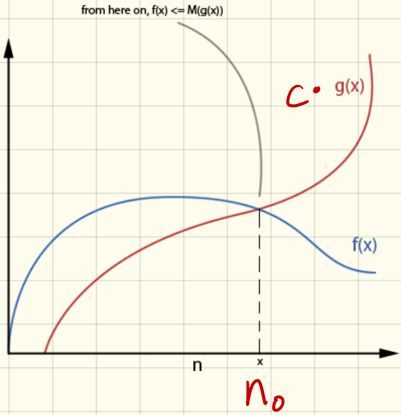


Big-O notation

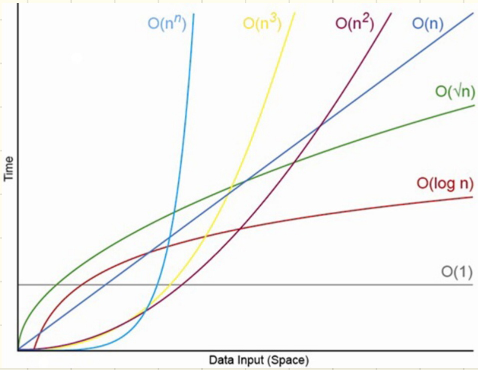
We say $f(n)$ is $O(g(n))$ if

$\forall n > n_0$, $\exists c > 0$ such that

$$f(n) \leq c \cdot g(n)$$



$5n$ is $O(n)$



Examples

① $5n$ is $O(n^2)$

Let $c = 6$, for any $n > \underset{n_0}{2}$,

$$5 \cdot n < 6 \cdot n^2$$

why? $5 < 6$ & $n < n^2$ ✓

② $5n$ is $O(n)$

Let $c = 7$

and $5n < 7n$

③ $\boxed{16n^2 + 21n}$ is $O(n^2)$

Let $c = 16 + 21 = 37$

& $n > 2$

then $16n^2 + 21n < 37n^2$

Thm: $f(n) = a_c n^c + a_{c-1} n^{c-1} + \dots + a_0$

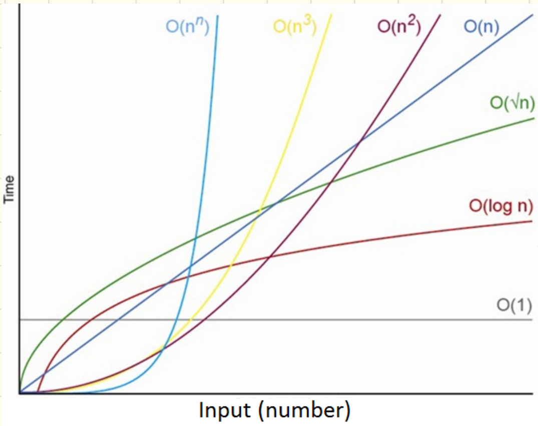
then $f(n) = O(n^c)$

Common runtimes

1. O(1) - constant time
2. O(log n) - binary search
3. O(n) - linear search
4. O(n log n) - sorting binary trees
5. O(n^2) ~ nested for loops

(polynomial) (quadratic)

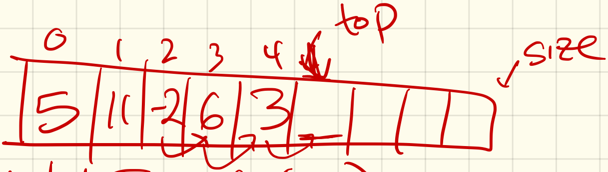
And: O(2^n), O(n!)] bad!



<u>Claim:</u> Inserting a new element at the beginning of an array is $O(n)$ time.

<u>pf:</u>

A:



addFront(2)

```
for (int i = top; i >= 0; i--)
    A[i] = A[i-1];
A[0] = 2;
```

Worst case: top $= O(\text{size})$

& this is how many iterations are in my loop

if size $= n$,

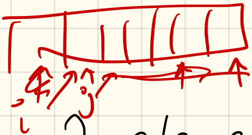
then $O(n)$

Claim: Inserting an element at the head of a list is $O(1)$ time.

- allocate new node
- copy value into it
- update next pointer
- update head pointer

↳ roughly 5 operations so $O(1)$

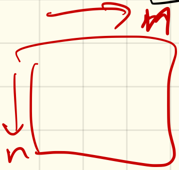
Nested for loops:



Ex: find if any 2 elements in the array are equal.

```
for (int i=0; i<n; i++)
    for (int j=i; j<n; j+1)
        if (A[i] == A[j])
            return true;
return false;
```

3 operations



Running time:

$$\sum_{i=0}^{n-1}\left[\underbrace{\sum_{j=i}^{n-1} 3}_{\underbrace{3+3+3+\cdots+3}_{n-i \text{ times}}}\right]$$

$$= \sum_{i=0}^{n-1} 3(n-i) = 3n + 3(n-1) + 3(n-2) + \cdots + 3$$

$$3n + 3(n-1) + \cdots + 3$$

$$= 3 \boxed{\sum_{i=1}^{n} i} = 3 \cdot \frac{n(n-1)}{2}$$

$$= 3\left(\frac{n^2 - n}{2}\right)$$

$$= \frac{3}{2}n^2 - \frac{3}{2} \cdot n$$

$$= O(n^2)$$

From here on out, we'll use this analysis for any function or data structure we code.

Some may be obvious:

Some harder:

Runtime of stack operations