What makes an algorithm good?
Why would an algorithm be better than another?

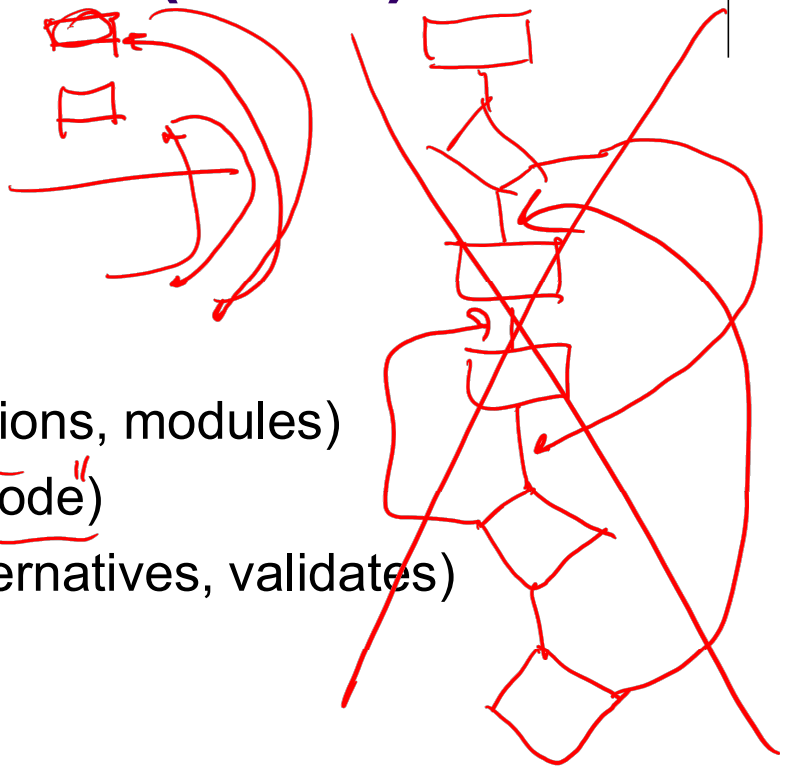
Goodness of an algorithm?

- #1 criteria: correct
- Correct results

if we do not obtain the needed is it useless?
It depends on the problem...
It may be all we can do given the problem...

- (some cases may need to compromise w/ an approximate solution)

Goodness of an algorithm? (cont.)



- #2: Desirable qualities:
  - Clear to read code and to debug
  - Code easy to understand
  - Concise code
  - Modular organization (top level, functions, modules)
  - Structured (no breaks, no spaghetti code)
  - Robust (does not crash, previews alternatives, validates)
  - Easy to maintain and revise
  - Clear/informative user interaction

Goodness of an algorithm? (cont.)

- Efficient time wise:
  - executes efficiently (with a realistic response time): TIME COMPLEXITY
- Efficient use of space (memory):
  - SPACE COMPLEXITY
- Efficiency: essential qualities to consider for large size problems!

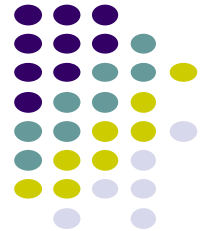
Measuring Algorithm Time Complexity

- Empirically: Seconds, miliseconds…. good idea if controlling the environment. BUT it depends on so many variables!

- Exact count of operations that get executed

Order of an algorithm – Time complexity of the algorithm

Order of an algorithm



- Gives a notion of the Time complexity of the algorithm
- This is a math based theory that is most relevant for problems and algorithms involving large numbers of data (large size of problems)

Order of an algorithm (cont.)

- The Order gives an "approximate" measure of an algorithm in terms of number of "critical operations" that are executed in the algorithm…
- "approximate" is in fact very precisely defined mathematically.

Order of an algorithm (cont.)

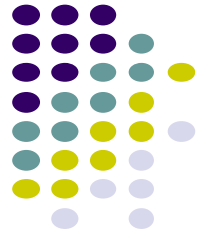
- Critical operations can be:
  - additions
  - comparisons (if statements)
  - transfer operations (assignments, swaps)
  - …

Order of an algorithm (cont.)

- The **order** is expressed as the **number of critical operations** that the algorithm executes expressed as **a function of the problem input size**

- problems **input sizes** may be:
  - dimensions of lists/ matrices,
  - the number of values to be added,
  - the number of values among which we search
  - the number of values to be sorted…

Big-O



- An algorithm is rated in terms of some **reference function**. It is said to be in the order (big-O) of some reference function:
  - O(n), O(n^2), O(log n), etc.

- Intuitively, that means that **for sufficiently large n**, the time that the algorithm will take (to execute the critical operation) will be proportional to n, n^2, log n, etc, where n is the size of the problem.

Standard Reference functions

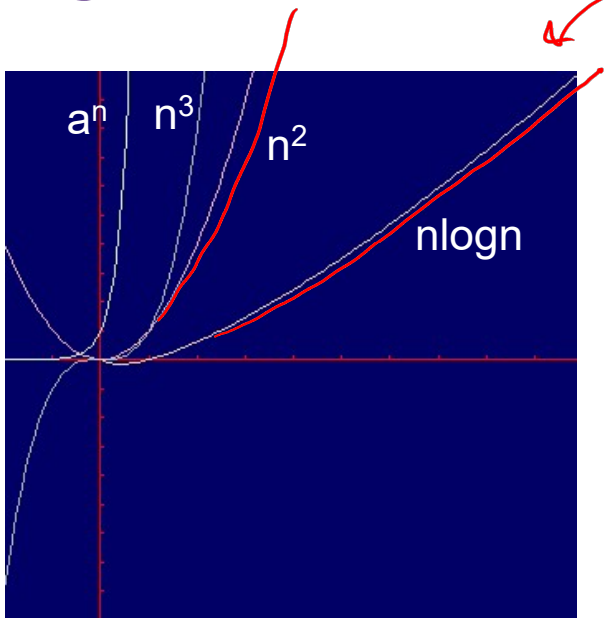
Category	Reference Function
Constant	1
Logarithmic	$\log_2(n)$
Linear	n
nlogn	$n\log_2(n)$
Quadratic	n^2
Cubic	n^3
Exponential	a^n, a>1

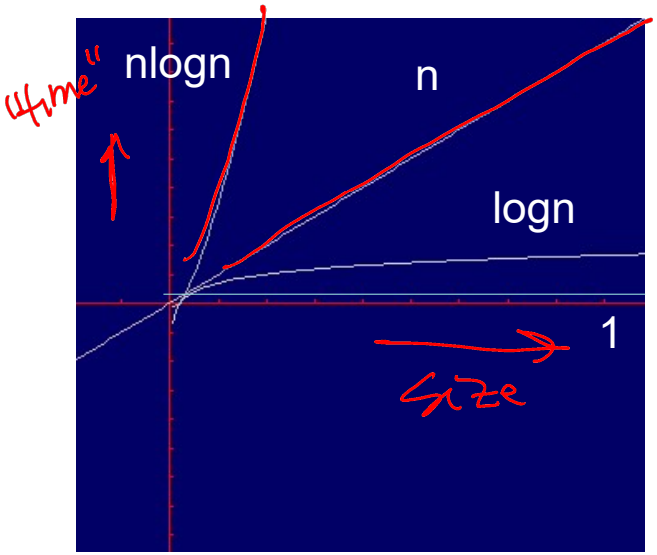
$\log_2 8 = 3$ because $2^3 = 8$

$\log_2 n = x$ when $2^x = n$

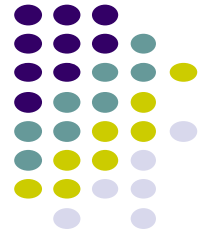
$n * \log_2 n$ ← multiplied

Comparing the standard reference functions





What could be an algorithm of order…



- O(1)? Color shoes first person in line
- O(n)? Linear search (in the worse case)
- O(n^2)? Selection sort

As n gets larger… (as the size of the problem is larger…)

n	Ex: Get first element in list always constant O(1)	Ex: Linear search O(n)	Ex: Binary search O(log n)	Ex: Selection sort $O(n^2)$	Ex: Merge sort O(n log n)
10	1	10	3.32	100	33.2
100	1	100	6.64	10,000	664
1,000	1	1000	9.96	1,000,000	9,960
10,000	1	10,000	13.28	100,000,000	132,800

Comparing algorithms:

- Which algorithm is preferred for large n if we are choosing based on time complexity?

We prefer an algorithm growing slowest as the size of the problem increases

Time needed to process n items for 6 algorithms

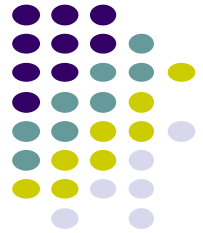
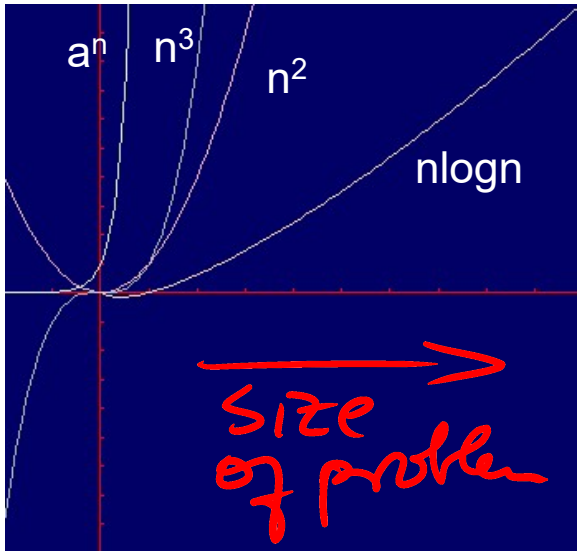
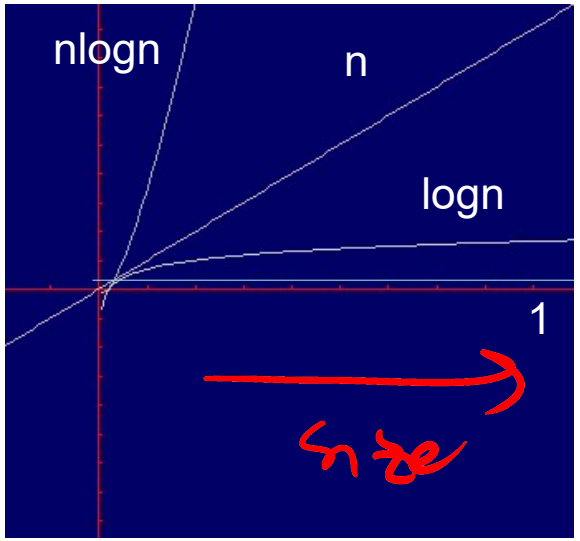


Table 4.2, Gossett

$g_i(n)$	n=10,000	n=100,000	n= 1,000,000	n=250,000,000
$\log_2(n)$	13 nanosecs	17 nanosecs	20 nanosecs	28 nanosecs
n	0.00001 secs	0.0001 ses	0.001 secs	0.25 secs
$n\log_2(n)$	0.00013 secs	0.00166 secs	0.01993 secs	6.97434 secs
n^2	0.1 secs	10.0 secs	16 mins, 40 secs	≈ 1.98 years
n^3	16 mins, 40 secs	≈ 11 days,14 hs	32 years	≈ 500 million years
2^n	≈ 6×10^{2993} years	-------Too long to contemplate -------		

Comparing the standard reference functions





(A) algorithm in O(1) is preferred as n gets larger
(B) algorithm in $O(n^2)$ is preferred as n gets larger