COL106 - Data Structures and Algorithms

Find the largest element in an array

```
Algorithm arrayMax (A, n)
  Input : array A with n elements
  Output : largest element in the array
  currentMax ← A[0];
  for i ← 1 to n-1 do
      if currentMax < A[i] then
          currentMax ← A[i]
  return currentMax
```

pseudocode

more formal than English

less detailed than a program

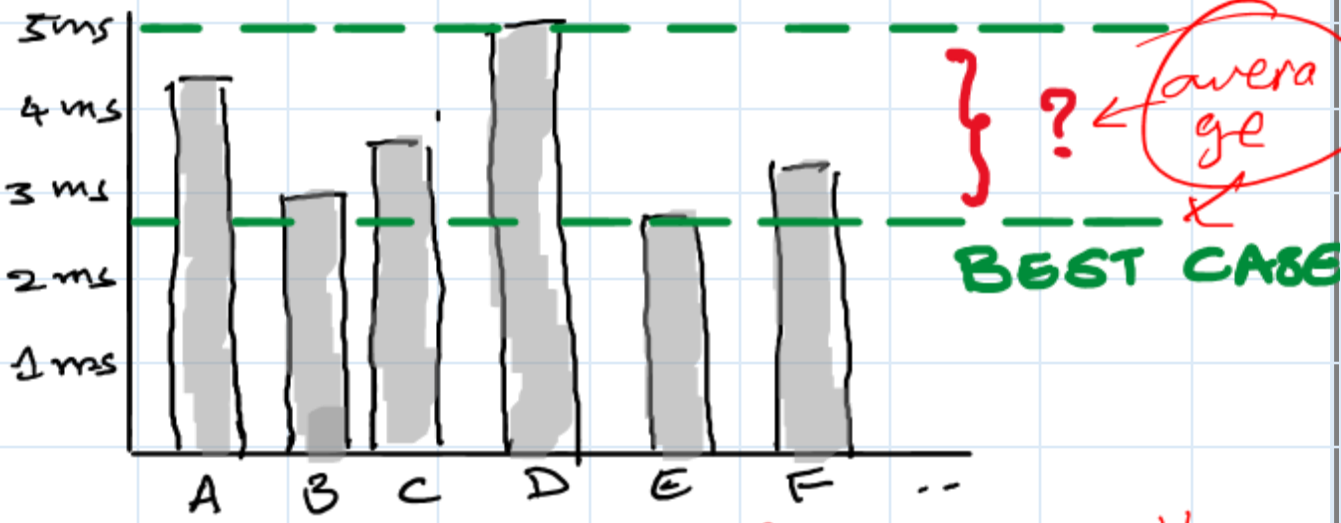
hides program design / implementation

Find the largest element in an array

How efficient ?

```
Algorithm arrayMax (A, n)
  Input : array A with n elements
  Output : largest element in the array
  currentMax ← A[0];
  for i ← 1 to n-1 do
      if currentMax < A[i] then
          currentMax ← A[i]
  return currentMax
```

EXPERIMENTAL



Even when the input array size remains same there are variations in observed runtime

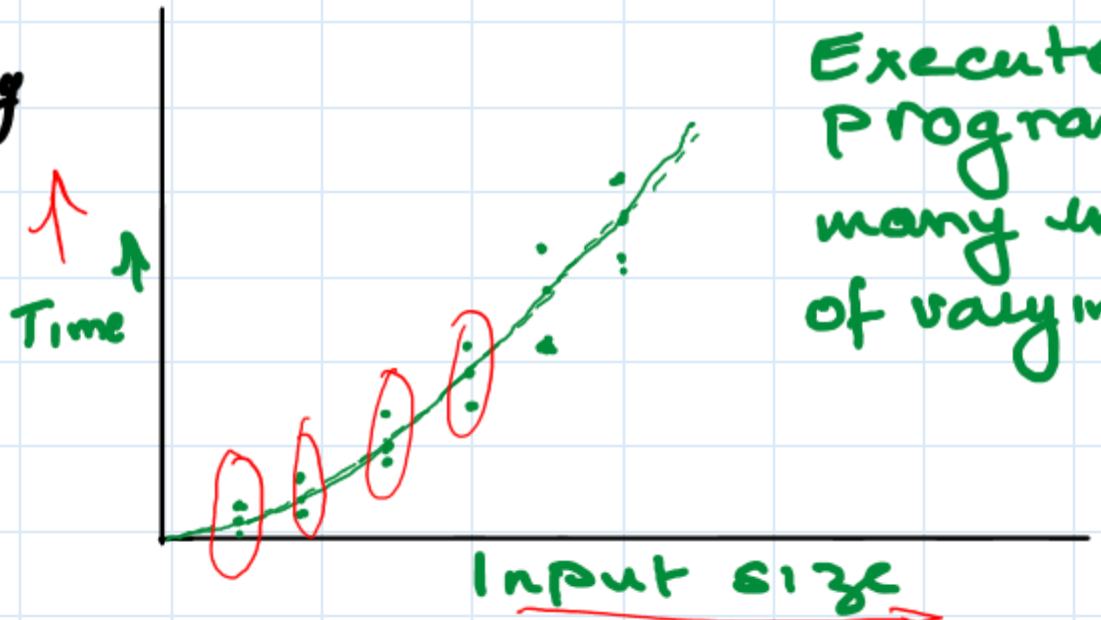
Find the largest element in an array

How efficient ?

Algorithm arrayMax (A, n)
Input : array A with n elements
Output : largest element in the array

```
currentMax ← A[0];
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

EXPERIMENTAL



Execute the program with many instances of varying size

Even when the input array size remains same there are variations in observed runtime

What drives the choice of implementation?

1. Efficiency of the algorithm

   ⇒ will the implementation choice help in speeding up the algorithm.

   ⇒ add new methods, improve state/data "structure", storage and access.

2. Resource usage (usually memory).

THEORETICAL ANALYSIS

Can we reason about the efficiency of an algorithm without implementing it?

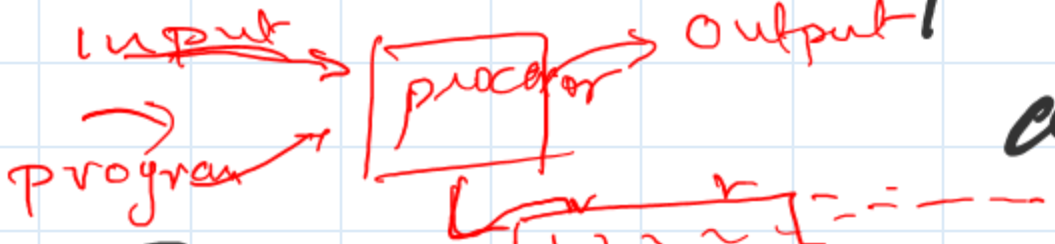
- running time influenced by the hardware or language used etc.
- the pain of achieving correct implementation.

* Algorithm instead of the program
* Characterize the running time as a function of input size n.
* Consider all possible inputs

Use an abstraction of a machine that "executes" the pseudocode/algorithm

ABSTRACTION OF A "MACHINE"

Abstraction helps to identify primitive ops.



and their cost model.

Random Access Machine (RAM) model

(1 processing unit)

⇒ Each simple operation takes 1 timestep

⇒ Unbounded collection of memory cells

- each with an address
- each can hold a bounded value
- each can be accessed in 1 timestep

How far is RAM from Reality?

	"real"	RAM
* Each operation takes constant time	✗	✓
* Memory is unbounded	✗	✓
* Memory access takes constant time	✗	✓
* Memory can hold bounded value	✓	✓

Real computers have many other sophistications (hyperthreading, vectorization, etc)

But RAM is still a "realistic" semplification.

Algorithm arrayMax (A, n)

Input : array A with n elements

Output : largest element in the array

```
1  currentMax ← A[0];
2  for i ← 1 to n-1 do
3      if currentMax < A[i] then
4          currentMax ← A[i]
5  return currentMax
```

assignment, addition/increment, comparison

1 operation.

2 operations

2n operations

2n operations

0 or 2 at each step.

(max 2(n-1))

Let a = time taken by the fastest primitive

b = time taken by the slowest primitive

worst-case

Total time of arrayMax = $T(n)$

$a(4n+5) \leq T(n) \leq b(5n+5)$

$(4n+5 \quad - \quad 5n+5)$

$\Rightarrow T(n)$ is bounded from both sides by linear functions (of input size)

What happens if we change
(i) CPU speed ?
(ii) memory cell capacity ?

Slide by Matt Stallmann included with permission.

Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

runtime quadruples when problem size doubles

Asymptotic Analysis

⇒ To focus on the "growth rate"

eg. the running time of arrayMax grows linearly with input size.

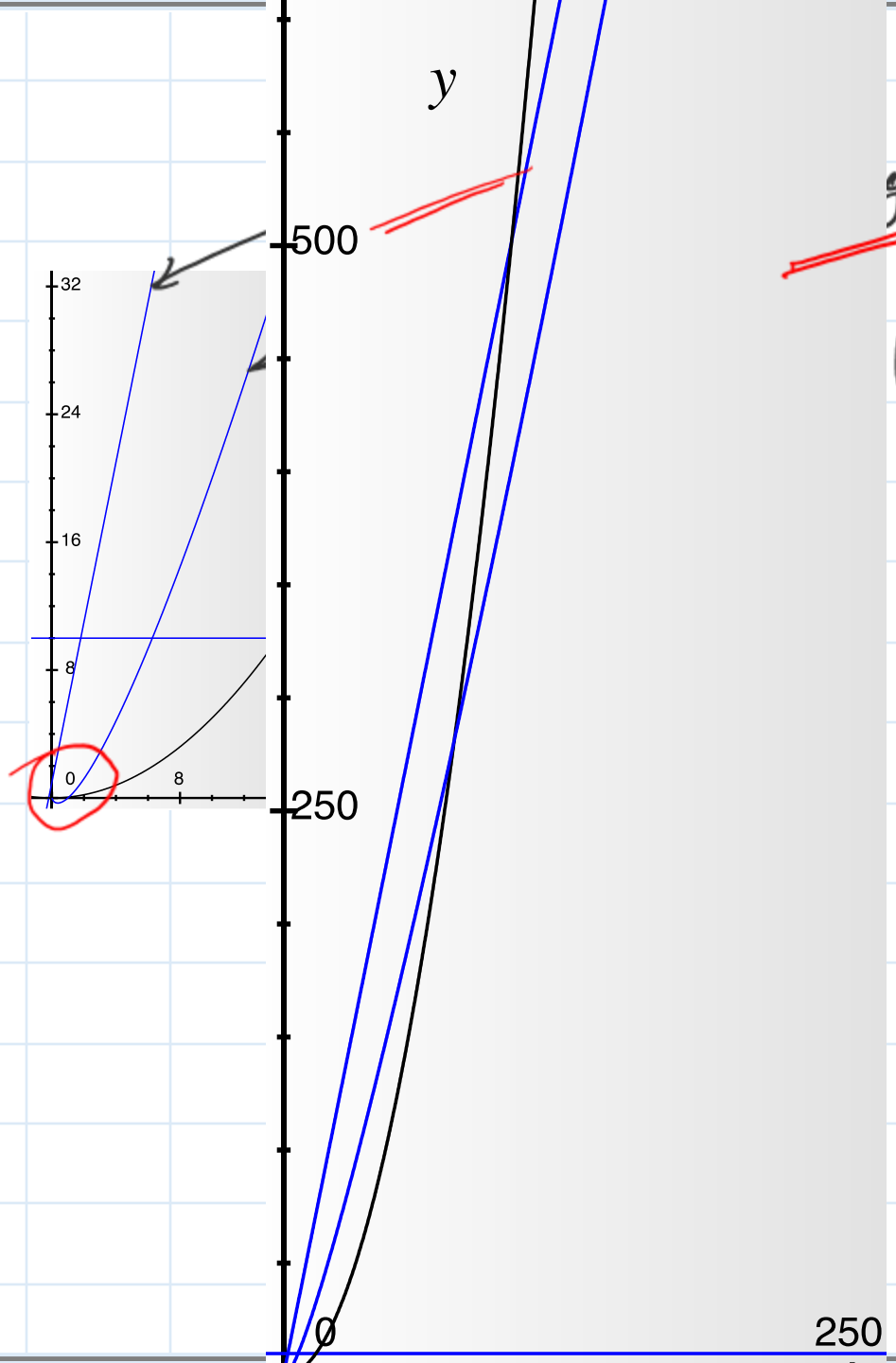
⇒ To get rid of "details" (implementation, h/w)

$4n \approx n$
$5n \approx n$
⋮

} you can "fix" constant by better h/w. but not the growth rate.

⇒ To capture the **essence** of the algorithm

How does it perform with the size of the input **in the limit**



$n \log n$

$\left(\frac{1}{20} n^2\right)$

At small input sizes the behavior of algorithms is highly dependent on the constants

Asymptotic Notation

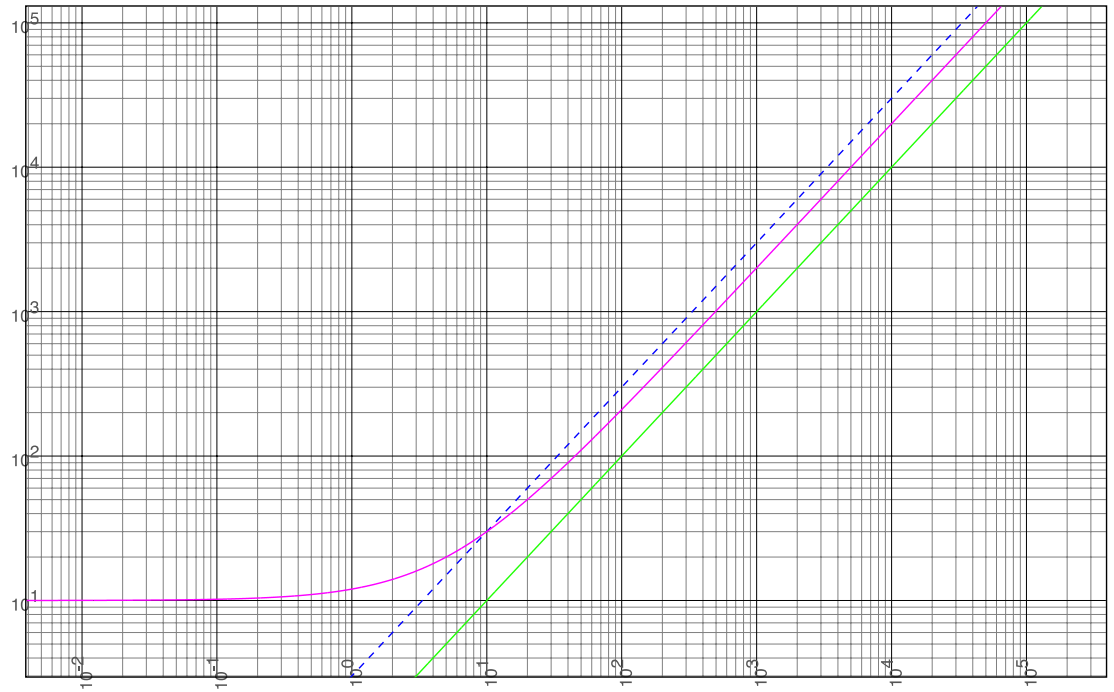
Big "Oh" notation. (O-notation)

asymptotic upper bound

$f(n)$ is $O(g(n))$ if there are two constants c and n_0 s.t.

$$f(n) \leq c\,g(n) \quad \text{for} \quad n \geq n_0$$

$f(n)$ and $g(n)$ are functions over non negative ints.



$2n+10$ is $O(n)$

$2n+10 \leq cn$

$c-2 \quad n \geq 10$

$n \geq 10/(c-2)$

let $c = 3$ and $n_0 = 10$

Asymptotic Analysis

⇒ Use O-notation to express the number of primitive operations executed as a function of input size

⇒ Comparing algorithms

$O(n)$ is better than $O(n^2)$

$O(\log n)$ is better than $O(n)$

...

Choice of Data structure

Algorithm listMax (A, n)
Input : a list with n elements
Output : largest element

```
current-Max ← get (A,0)
for i ← 1 to n-1 do
    if current-Max < get (A, i) then
        current-Max ← get (A, i)
return current-Max.
```

Does the choice of datastructure to emplement the List make a difference to the performance?

Algorithm prefixAverages(X)
Input: an n-element list of numbers X.
Output: an n-element list A of numbers s.t.
get(A,i) is the average of elements X(0...i).

```
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + get(x,i)
    set(A, i, a/(i+1))
return array A.
```

i iterations with $i = 0, 1, 2, \ldots n-1$

n iterations

⇒ What is the running time? { with array / with linked list.

⇒ Can we do better?

Algorithm BinarySearch (A, n, T)

Inputs: A is an array of n sorted elements.
T is the element value we want to locate

Output: ~~index of~~ the array element with value T.

BINARY SEARCH