COL106 - Data Structures and Algorithms

Asymptotic Analysis

Announcements

- Sign up on piazza
- Fill in form on pizza to grab one of the empty seats
- Lab Quiz: Jan 25
- Submit lab practice problems on moodle using vpl next week

Fend 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
```

- How to measure efficiency? ⇒
- Worst Case time Complexity" analysis

$T(n) = \max_{A : |A|=n} t(n)$

RAM model

$\leq c(4n+5)$ $\leq c(5n+5)$

Asymptotic Analysis

⇒ To focus on the "growth rate"

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

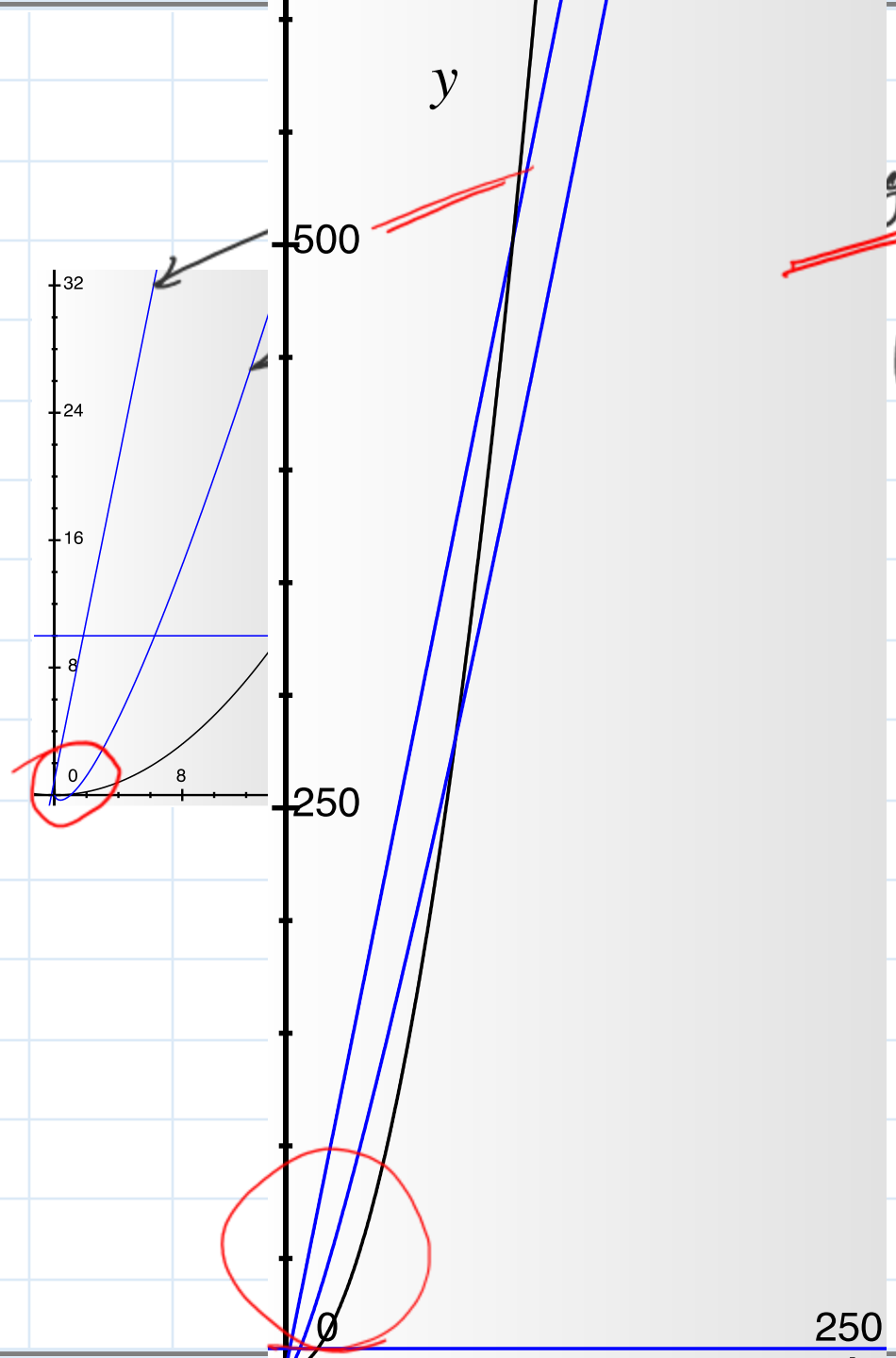
⇒ To get rid of "details" (implementation, h/w)

$4n \approx n$
$5n \approx n$
⋮

} 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**



$g\,n)$

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

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

Asymptotic Notation

Big "Oh" notation. (O-notation)

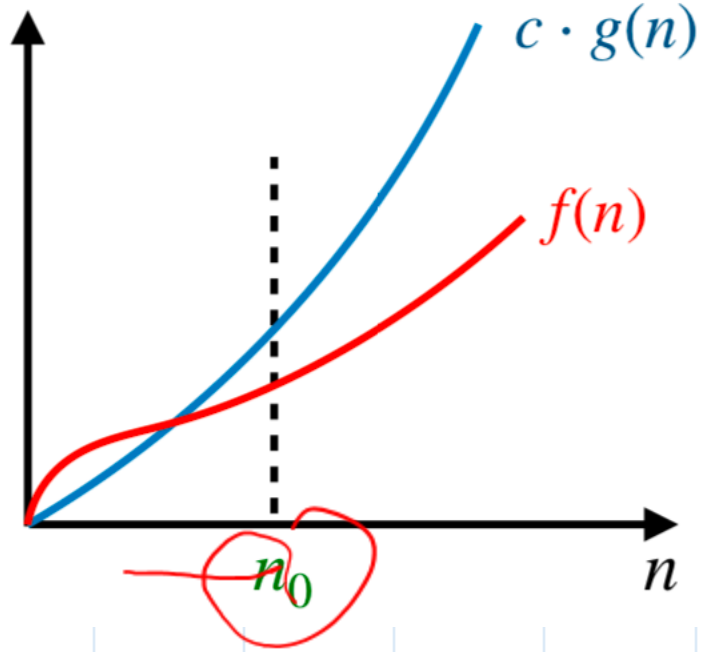
asymptotic upper bound

$f(n)$ 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.

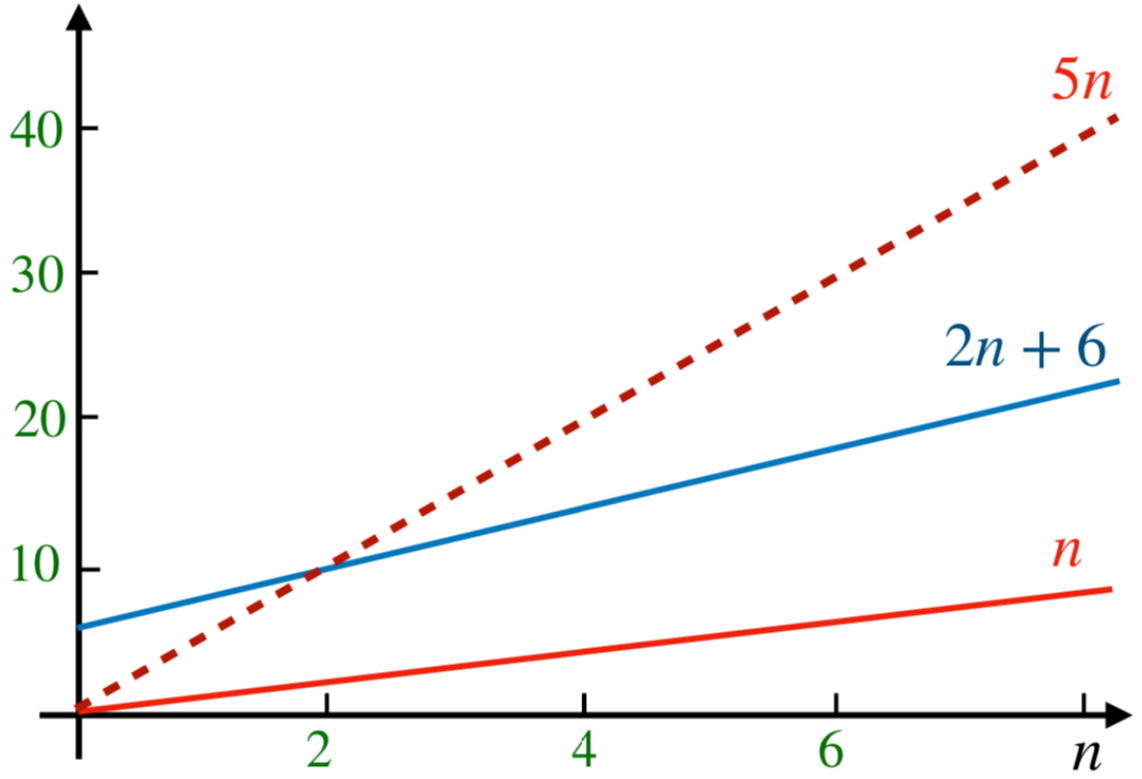
$f : \mathbb{N} \to \mathbb{R}^+$



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

$O(n)$ = $5n$

$10n$ $n^2/20$



$2n + 6 = O(n)$

$t(n) \in O(g)$

Asymptotic Analysis

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

⇒ Comparing algorithms

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

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

...

1. Multiplicative constants don't matter. $14n^2 = O(n^2)$
2. $O(n^a) \leq O(n^b)$ if $a < b$ $\quad O(n^2) \leq O(n^3)$
3. Exponential dominates polynomial.

Asymptotic Lower Bound : Big Omega

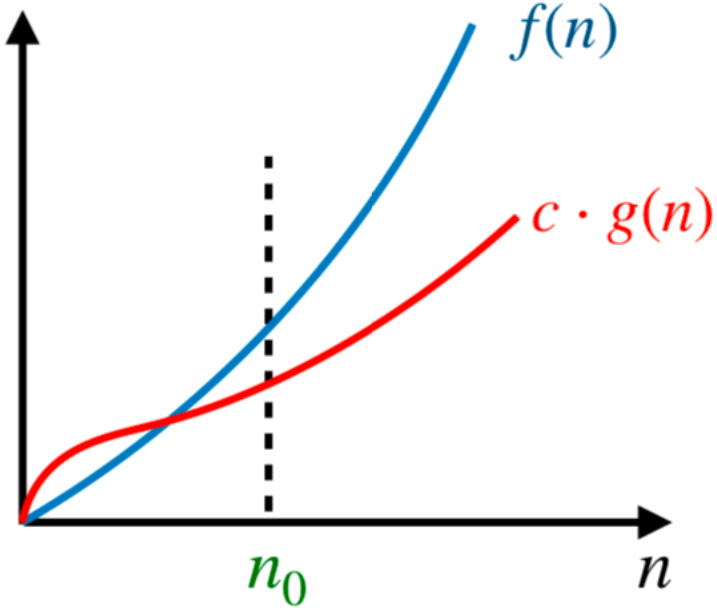
- We say $f(n)$ is $\Omega(g(n))$ if there exist constants c & n_0 s.t

$$f(n) \geq c\, g(n)$$

for $n \geq n_0$.

Big Theta

$\Theta(g(n))$



- We say $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Is arrayMax in $\Theta(n)$?

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
```

- Worst Case

1) $n^3 + 5n = \Theta(n^3)$

2) $4^{\log_2 n} = O(n^2 \log n)$

3) $8 + \lceil \log_3 \ n \rceil = \Theta(\log_5 \ n)$

4) $2n + 6 \neq \Omega(n \log n)$

5) $a_d n^d + \cdots + a_2 n^2 + a_1 n + a_0 \ = \ O(n^d), \quad \text{for } a_d > 0$

6) $n^k = O(2^n), \quad$ for each $k > 0$

7) $n \neq O(\log^k n), \quad$ for each integer $k > 0$

Exercise!

non-negative integers

$$O(f) = \{g \mid \exists [n_0, c > 0](\forall [n \geq n_0](g(n) \leq cf(n)))\}$$
$$\Omega(f) = \{g \mid \exists [n_0, c > 0](\forall [n \geq n_0](g(n) \geq cf(n)))\}$$
$$\Theta(f) = O(f) \cap \Omega(f)$$

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.
```

$O(1)$

n iterations

$O(n^2)$

$$\sum_{i=1}^{n} i \approx \frac{n(n+1)}{2}$$

$1+2+3+\cdots+n-1$

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

$A(0) = X(0)$ $\quad A(1) = \frac{X(1)+X(0)}{2}$

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, j)
    set(A, i, a/(i+1))
return array A.
```

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

n iterations

$A[i+1] \leftarrow A[i] + X[i+1]$ $\quad i+2$

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

⇒ Can we do better?

Primality Testing

Input: $n \in \mathbb{N}$

Output: yes if n is prime

Check-Prime(n)

1. if ($n \leq 1$) the return("not prime")
2. for ($i = 2$; $i < n$; i++)
   if (i divides n) then return("not prime")
3. return("prime")

- What is the input size?
- What is the running time?

A better algorithm

Check-Prime(n)

1. if ($n \leqslant 1$) then return("not prime")
2. set $i = 2$
3. while ($i \times i \leqslant n$)

   if (i divides n) then return("not prime")

   $i = i + 1$
4. return("prime")

- Running time?
- Is this a polynomial time algorithm?

$c \cdot \sqrt{n}$

$O(\sqrt{n})$