6.006	Lecture 1	Sept. 8, 2016

TODAY: Algorithmic Thinking/Peak Finding
- administrivia
- course overview
- peak finding problem (1D / 2D)
- divide & conquer

6.006: Introduction to Algorithms

Profs. Erik Demaine, Debayan Gupta, Ronitt Rubinfeld

TAs: Anton Anastasov, Peitong Duan, Themistoklis Gouleakis, Adam Hesterberg, Gregory Hui, Atalay Ileri, Skanda Koppula, Akshay Ravikumar, Ludwig Schmidt, Yonadav Shavit, Aradhana Sinha, Ali Vakilian, Ray Hua Wu, Parker Zhao

http://stellar.mit.edu/S/course/6/fa16/6.006/

6.006-staff@mit.edu

Handout 1:

* sign up at alg.csail.mit.edu by 5 P.M. TODAY
- recitation assignment tonight
  (note: 2 per week / 1 per lecture)
- sign up for Piazza (recommended)

- prereqs: 6.01/6.009 coreq (Python)
  & 6.042 (discrete math)
- grading: 5 psets (30%, Python 3 + LaTeX)
  Quiz 1 (20%, Oct. 18, 7:30-9:30)
  Quiz 2 (20%, Nov. 17, 7:30-9:30)
  final (30%)
- read policies: pset grace days, collaboration

Algorithm = mathematical abstraction of a computer program
= well-specified procedure for solving a computational problem

- usually: finite sequence of operations described via structured English, pseudocode, or real code ↳ preferred

History/naming: al-Khwārizmī (c. 780-850)
"al-kha-raz-mee"

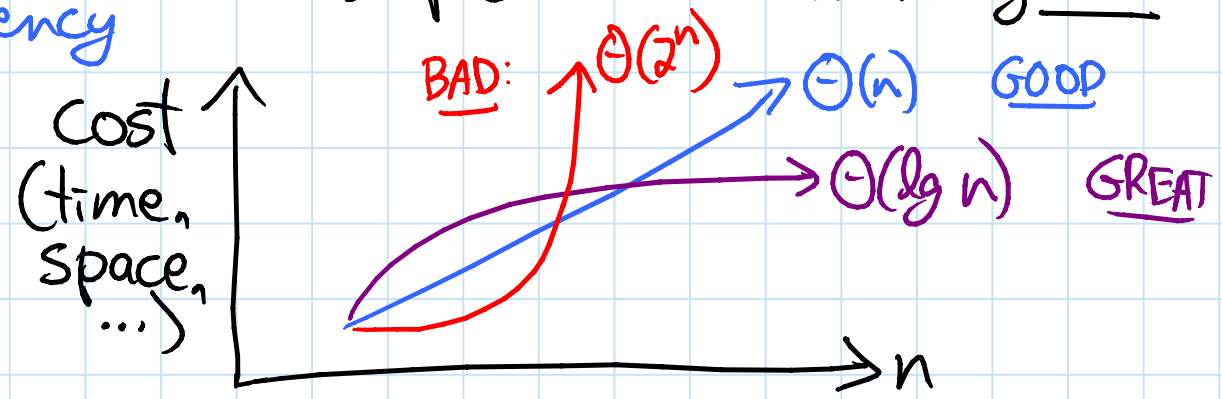
- father of algebra
- author of *The Compendious Book on Calculation by Completion and Balancing* [c. 830]
- some of the first algs: linear & quadratic eqns.

Want an algorithm that's:

- correct
- fast ≈ scalable } focus here
- small space
- general, simple, clever, ...

Scalability: measure time/space/etc. as problem size n grows $\to \infty$

"efficiency"



- **big idea**: IGNORE CONSTANT FACTORS using asymptotic notation ($\Theta, O, \Omega, o, \omega$)
  e.g. $5n^2 - 7n + 4 = \Theta(n^2)$
  $\Rightarrow$ minor details (machine instruction set, compiler optimization, ...) irrelevant
  - dominant for large n

- **polynomial time** = $\Theta(n), \Theta(n^2), \Theta(n^3), \ldots$ = "GOOD"
- **exponential time** e.g. $\Theta(2^n)$ = "BAD"

Main topics in 6.006:

- sorting
- data structures:
  - heaps
  - binary search trees
  - hashing
- graph search
- shortest paths
- techniques:
  - divide & conquer
  - dynamic programming
- continuous optimization
- geometry
- complexity (NP-complete)

sample application:

spreadsheets
Google
simulation
scheduling
file sync.
Rubik's Cube
Google Maps

(TODAY)
optimization
reals
video games
lots ∴

Peak finding: find a local maximum (or minimum) in a 1D or 2D array

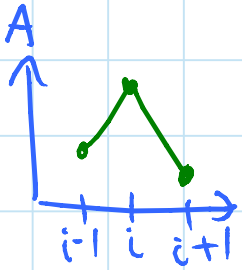
(think: blind Geordi LaForge searching for water accumulation on alien mountain range, with transporter)

1D: input = array $A[0 \cdots n-1]$
output = i such that $A[i-1] \le A[i] \ge A[i+1]$
where $A[-1] = -\infty$ & $A[n] = -\infty$

Algorithm 1: brute force
- test $A[0], A[1], \ldots, A[n-1]$ for peakyness
- $\Theta(1)$ time each $\Rightarrow O(n)$ time

Algorithm 2: max(A) is a local max
- $\Theta(n)$ time

Algorithm 3: divide & conquer
- look at element i
- if it's a peak: done, return it
- else it has a neighbor $> A[i]$
  - if $A[i-1] > A[i]$: must be peak to left
  - if $A[i+1] > A[i]$: must be peak to right
  - $\Rightarrow$ recurse on $A[:i]$ or $A[i+1:]$



(Why? walk left/right until descend or hit end $\Rightarrow$ peak)

(<i) (>i)

⇒ left with $\max\{i, n-i-1\}$ elements in the worst case

⇒ set $i = n-i-1$ to balance

⇒ $i = (n-1)/2$ = MIDDLE

Divide & conquer algorithm:

1D peak: one half

1. divide input into part(s)
2. conquer (solve) each part recursively
3. combine result(s) to solve original

- if size-n input divided into $n_1, n_2, \ldots, n_k$ then $T(n)$ = worst-case running time
  = divide time
  $+ T(n_1) + T(n_2) + \cdots + T(n_k)$ } RECURRENCE
  + combine time

- 1D peak: $T(n) = T(n/2) + \Theta(1)$
  - solve: $T(n) = T(n/2) + c$
    $= T(n/4) + c + c$
    $= \cdots$
    $= T(n/2^k) + c \cdot k$
    $= T(n/2^{\lg n}) + c \cdot \lg n$
    $= T(1) + c \lg n$
    $= O(\lg n)$

can't use $\Theta(1)$ here because of "..." — need to keep track of constant used $\lg n$ times

2D: input = $n \times n$ matrix A
output = (i, j) such that:

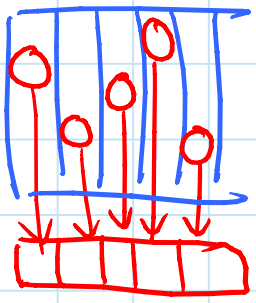
$$A[i-1, j] \le A[i, j] \ge A[i, j-1],\ A[i, j+1],\ A[i+1, j]$$

Algorithm 1: brute force
- test all n^2 elements for peakyness

$\Rightarrow O(n^2)$ time

Algorithm 2: reduce to 1D
1. take max element in each column
2. solve 1D problem on max array

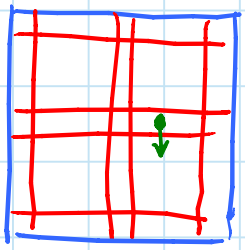
$\Theta(n^2)$ time



Algorithm 3: be lazy about (1)
- solve 1D problem on max array
- for each of $O(\lg n)$ accesses to max., compute in $\Theta(n)$ time

$\Rightarrow O(n \lg n)$ time

Algorithm 4: 2D divide & conquer



- window = first, middle, & last row & column
- if it has a peak: done (can avoid rechecking first/last row/column)
  - ↳ relative to entire array
- else: find max(window)
  - must have a larger neighbor
    - ⇒ can't be at window corners, center, ...
  - ⇒ recurse in unique subwindow containing it

Correctness: subwindow contains peak

- walk up (N/S/E/W as appropriate) from neighbor until get stuck at a peak
- walk ≥ neighbor > max ≥ subwindow boundary

⇒ never left subwindow

Time: $T(n) \le T(n/2) + O(n)$

$n \times n \to \frac{n}{2} \times \frac{n}{2}$

- solve:

$$\begin{aligned} T(n) &\le T(n/2) + c \cdot n \\ &\le T(n/4) + c \cdot \tfrac{n}{2} + c \cdot n \\ &\le T(n/8) + c \cdot \tfrac{n}{4} + c \cdot \tfrac{n}{2} + c \cdot n \\ &\le \cdots \\ &\le T(1) + c \cdot n \underbrace{\left(1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \cdots\right)}_{< 2} \\ &= O(n) \end{aligned}$$